

KERNFORSCHUNGSZENTRUM KARLSRUHE

Institut für Datenverarbeitung in der Technik

KfK 2506

OBERBLICK UND VERGLEICH VERSCHIEDENER MITTEL FOR
DIE SPEZIFIKATION UND DEN ENTWURF VON SOFTWARE

J. Ludewig
W. Streng

Kernforschungszentrum Karlsruhe GmbH, Karlsruhe

Zusammenfassung:

Dieser Bericht enthält eine Sichtung und Auswertung relevanter Verfahren für die Spezifikation und den Entwurf von Software. Dabei wird ein einheitliches Beschreibungsschema angewandt. Die einzelnen Ansätze werden nach verschiedenen Kriterien vergleichend gegenübergestellt.

Abstract:

REVIEW AND COMPARISON OF TOOLS FOR SOFTWARE SPECIFICATION AND DESIGN

This report contains a collection and evaluation of relevant methods for specification and design of software, using a uniform description scheme. The various approaches are compared by several different criteria.

Inhaltsverzeichnis

	<u>Seite</u>
1. Einleitung	1
2. Die einzelnen Spezifikations- und Entwurfssysteme	4
2.1 <u>H</u> ierarchy plus <u>I</u> nput- <u>P</u> rocess- <u>O</u> utput (HIPO)	5
2.2 <u>H</u> igher <u>O</u> rd <u>e</u> r <u>S</u> oftware (HOS)	12
2.3 <u>J</u> ackson- <u>D</u> esign- <u>M</u> ethodology (JDM)	22
2.4 LOGOS	29
2.5 <u>M</u> odular <u>A</u> pproach to <u>S</u> oftware <u>C</u> onstruction, <u>O</u> peration and <u>T</u> est (MASCOT)	33
2.6 <u>M</u> odule <u>I</u> nterconnection <u>L</u> anguage 75 (MIL75)	42
2.7 <u>P</u> rogram <u>D</u> esign <u>L</u> anguage (PDL)	46
2.8 <u>P</u> roblem <u>S</u> tatement <u>L</u> anguage/ <u>P</u> roblem <u>S</u> tatement <u>A</u> nalyzer (PSL/PSA)	51
2.9 <u>S</u> tructured <u>A</u> nalysis and <u>D</u> esign <u>T</u> echnique (SADT)	55
2.10 <u>S</u> tructured <u>D</u> esign (SD)	61
2.11 <u>S</u> oftware <u>E</u> ngineering <u>F</u> acility (SEF)	66
2.12 <u>S</u> oftware <u>R</u> equirements <u>E</u> ngineering <u>P</u> rogram (SREP)	67
2.13 <u>S</u> tanford <u>R</u> esearch <u>I</u> nstitute/ <u>P</u> arnas <u>D</u> esign <u>M</u> ethodology (SRI/PARNAS)	74
2.14 <u>S</u> oftware <u>S</u> pecification and <u>E</u> valuation <u>S</u> ystem (SSES)	84
3. Vergleichende Gegenüberstellungen	94
3.1 Vergleich des Inhalts, der Form der Darstellung und der Entwicklungsmethoden	95

	<u>Seite</u>
3.2 Klassifizierung und Bewertung der Methoden und Systeme	96
3.3 Zuordnung zu den einzelnen Entwicklungsphasen	99
3.4 Die Möglichkeiten der automatischen Systeme	100
3.5 Vergleichende Beschreibung eines kleinen Entwurfs-Beispiels /2/	102
3.6 Literatur	106

Die Literatur zu den einzelnen Systemen ist in den betreffenden Abschnitten angegeben.

1. Einleitung

Ein beträchtlicher Teil der Software-Fehler entsteht in der Spezifikations- und Entwurfsphase der Software-Entwicklung. Dies ist vor allem auf einen Mangel an geeigneten formalen Hilfs- und Prüfmitteln zurückzuführen. Im Rahmen des Vorhabens 'Rechnergestützte Software Entwurfs- und Prüfmittel' sollen deshalb u.a. Methoden und Hilfsmittel für den zuverlässigen Entwurf von Anwendersoftware für Prozeßrechner sowie für seine Prüfung entwickelt werden.

Ein erster Schritt ist die Sammlung, Sichtung und Auswertung der in der Literatur bekannten, z.T. schon in der Praxis bereits eingesetzten Verfahren für die Spezifikation und den Entwurf von Software. Darauf aufbauend sollte eine Untersuchung der Anwendbarkeit dieser Verfahren für die Aufgabenstellungen der Prozeßdatenverarbeitung zur Auswahl und Definition brauchbarer Hilfsmittel führen.

Aus der Vielzahl von Methoden und Systemen haben wir die folgenden ausgewählt und betrachtet (alphabetisch):

- HIPO Hierarchy plus Input-Process-Output
- HOS Higher Order Software
- JDM Jackson Design Methodology
- LOGOS
- MASCOT Modular Approach to Software Construction, Operation, and Test
- MIL Module Interconnection Language
- PSL/PSA Problem Statement Language/Problem Statement Analyzer
- SADT Structured Analysis and Design Technique
- SD Structured Design
- SEF Software Engineering Facility
- SREP Software Requirements Engineering Program
- SRI/PARNAS Stanford Research Institute/ParnasDesign Methodology
- SSES Software Specification and Evaluation System

Diese Verfahren geben einen repräsentativen Überblick des betrachteten Gebiets. Obwohl die Entwicklung dieser Systeme und die uns zugänglichen Informationen sehr unterschiedlich sind, haben wir das folgende allgemeine Beschreibungsschema zugrunde gelegt:

1. Kurzbeschreibung
2. Anwendungsphase und -gebiet
3. Inhalt und Form der Darstellung
4. Entwicklungsmethode
5. Literatur
6. Beispiel

Die Kurzbeschreibung gibt jeweils eine zusammenfassende Charakterisierung des betrachteten Systems. In den drei folgenden Punkten des Beschreibungsschemas werden wichtige Aspekte dieser allgemeinen Charakterisierung vertieft. Im fünften Punkt wird die verwendete Literatur angegeben, und den Abschluß bildet ein Anwendungsbeispiel.

In vielen Fällen grenzt die Literatur das Feld möglicher Anwendungen der Systeme (Punkt 2) nicht ein; aus den Einsatzgebieten, die in Tabelle 3.2 c genannt sind, lassen sich aber Rückschlüsse ziehen. Die Unterscheidung von Inhalt und Form der Informationsdarstellung (Pkt.3) und der Vorgehensweise zur Gewinnung dieser Information ermöglicht eine globale Klassifizierung der betrachteten Systeme, wie sie in Tabelle 3.2 a, b angegeben ist. Es zeigt sich, daß der überwiegende Teil der verfügbaren Entwicklungssysteme im wesentlichen Hilfsmittel zur Darstellung der Informationen zur Verfügung stellt, während nur wenige Ansätze von einer Entwicklungsmethode ausgehen.

Anläßlich seines Tutorials über rechnergestützte Software-Entwicklungssysteme in London (1977) hat Prof. Teichrow für die Zukunft folgende Änderungen in der Software-Entwicklung als Voraussetzungen zur Steigerung der Softwarequalität und zur Erhöhung der Produktivität bei der Softwareerstellung angegeben.

- Die Software-Entwicklung wird verstärkt unterstützt durch neue aufeinander abgestimmte Hilfsmittel (manueller Art oder rechnergestützt).

- Ein größerer Anteil der Software-Produktion wird vom Rechner selbst übernommen.

- Die Menge der neu zu erstellenden Software wird vermindert durch den verstärkten Einsatz wiederverwendbarer Software-Bausteine.

Dieser Ausblick in die Zukunft sollte beim Lesen der nachfolgenden Systembeschreibung beachtet werden.

2. Die einzelnen

Spezifikations- und Entwurfssysteme

2.1 Hierarchy plus Input-Process-Output (HIPO)

2.1.1 Kurzbeschreibung

HIPO stellt ein Verfahren zur Unterstützung des Entwurfs und der Dokumentation von Software-Systemen dar. Es basiert auf der Voraussetzung eines modularen, hierarchischen Systemaufbaus und enthält graphische Techniken zur Beschreibung der Modulhierarchie und der einzelnen Moduln.

HIPO besteht aus zwei Grundelementen:

- Einem Diagramm zur Darstellung der hierarchischen Struktur der Moduln, d.h. einer Beschreibung der Zerlegung einer Funktion in ihre Teilfunktionen.
- Input-Process-Output Diagrammen, die jede Funktion in der Hierarchie durch ihre Eingabe, die Verarbeitung und die Ausgabe beschreiben.

Die leicht erlernbare, einheitliche graphische Darstellungstechnik erleichtert die Verständigung zwischen den verschiedenen Software-Entwicklern.

2.1.2 Anwendungsphase und Gebiet

HIPO gibt einen Rahmen für die entwicklungsbegleitende Dokumentation eines Softwaresystems.

Ausgehend von einem Übersichtsdiagramm, in dem die Hauptkomponenten eines Systems grob skizziert werden, haben die Designer die Möglichkeit, den Zerlegungsprozeß durch schrittweise Verfeinerung der HIPO-Diagramme einheitlich und verständlich darzustellen. Neben dieser Unterstützung der Entwurfsphase tragen sie auch zum Auffinden von Abhängigkeiten bei Programmänderungen in der Wartungsphase bei.

2.1.3 Inhalt und Form der Darstellung

Eine HIPO-Beschreibung besteht aus drei Diagrammart, einer Strukturübersicht (visual table of contents), Oberblicks-Diagrammen (overview diagrams) und Detaildiagrammen (detail diagrams) (siehe Bild HIPO-1).

Strukturübersicht

Diese Übersicht zeigt die hierarchische Anordnung aller Oberblicks- und Detail-Diagramme und ihre zugeordneten Namen und Markierungen (Nummern). Der Wurzel dieser baumartigen Beschreibung ist ein Diagramm zugeordnet, das die zu realisierende Gesamtfunktion darstellt. Die nächste Beschreibungsebene zerlegt diese Funktion in logische Teilfunktionen. Während der Softwareentwicklung dient die Strukturübersicht zur Beschreibung der funktionellen Zerlegung. Für das fertige Produkt kann sie als Inhaltsverzeichnis zum leichteren Auffinden von Informationen eines bestimmten Detaillierungsgrades oder eines bestimmten Diagramms dienen. Die Strukturübersicht sollte durch erläuternden Text (Legende) über die verwendeten Symbole ergänzt werden.

Oberblicks-Diagramm

Ein HIPO-Diagramm besteht - von links nach rechts - aus einem input-, einem process- und einem output-Abschnitt. Jedes Diagramm verfeinert eine Funktion in Teilfunktionen, die als nummerierte Schritte im process-Abschnitt aufgelistet werden. Der input-Abschnitt enthält Daten, die von den Teilfunktionen im process-Abschnitt benutzt werden, ihre Zuordnung wird durch Pfeile angegeben. Der output-Abschnitt enthält Daten, die von den einzelnen Teilfunktionen erzeugt oder modifiziert werden und deren Zuordnungen durch Datenflußpfeile beschrieben werden.

Oberblicks-Diagramme bilden die oberen Ebenen der Funktionshierarchie.

Detail-Diagramme

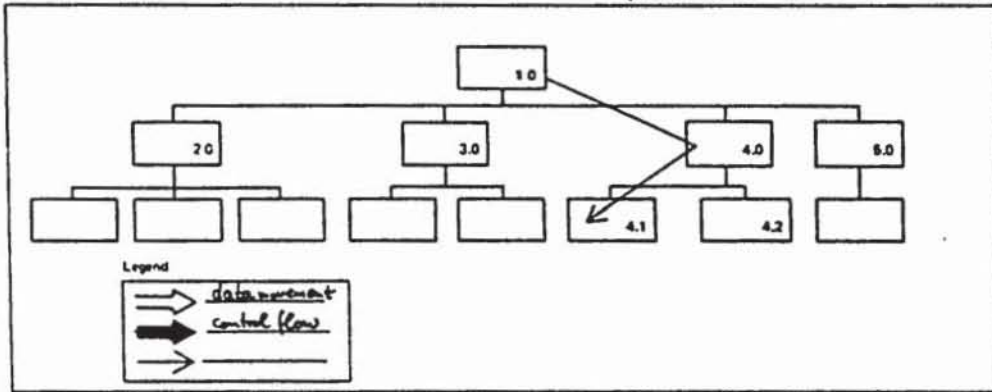
Sie sind genauso aufgebaut wie die Oberblicks-Diagramme, enthalten jedoch detailliertere Informationen über Teilfunktionen, spezielle Ein-/Ausgaben und interne Datenflüsse.

HIPO-Diagramme repräsentieren in erster Linie Datenflüsse. Vor allem in Detail-Diagrammen kann jedoch der process-Abschnitt auch Kontrollfluß-Informationen enthalten (allerdings in beschränktem Umfang). Darüberhinaus können die Detail-Diagramme durch einen Abschnitt für Verweise auf weitere Informationen oder zusätzliche Details (Modulnamen bei der Implementierung) ergänzt werden (Extended Description).

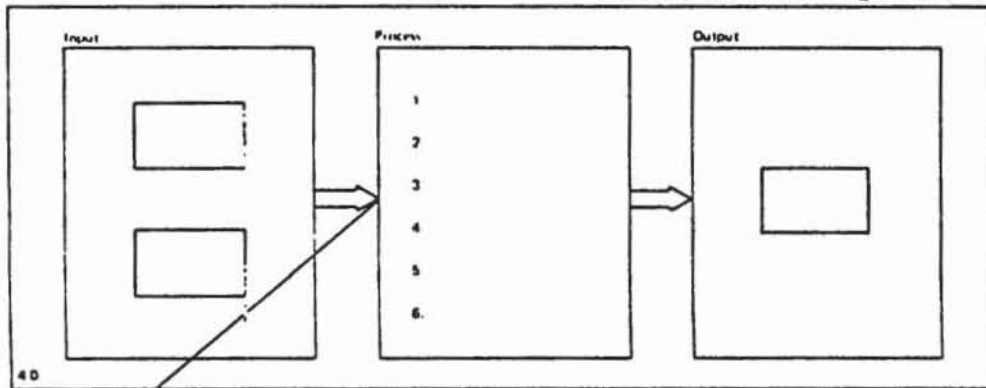
Zur Erleichterung der graphischen Erzeugung von HIPO-Diagrammen gibt es Schablonen für die verwendeten Symbole und HIPO-Arbeitsformulare zum Sammeln der in einem Diagramm enthaltenen Informationen. Es gibt auch einige Ansätze zur rechnergestützten Darstellung und Erzeugung von HIPO-Diagrammen wie z.B. das PROVAC-System von UNIVAC.

Als Nachteile von HIPO kann man das Fehlen einer Ausdrucksmöglichkeit für die Kontrollflußbeziehungen zwischen Moduln aufeinanderfolgender Hierarchieebenen sowie einer Beschreibungsmöglichkeit von Daten auf verschiedenen Abstraktionsebenen anmerken.

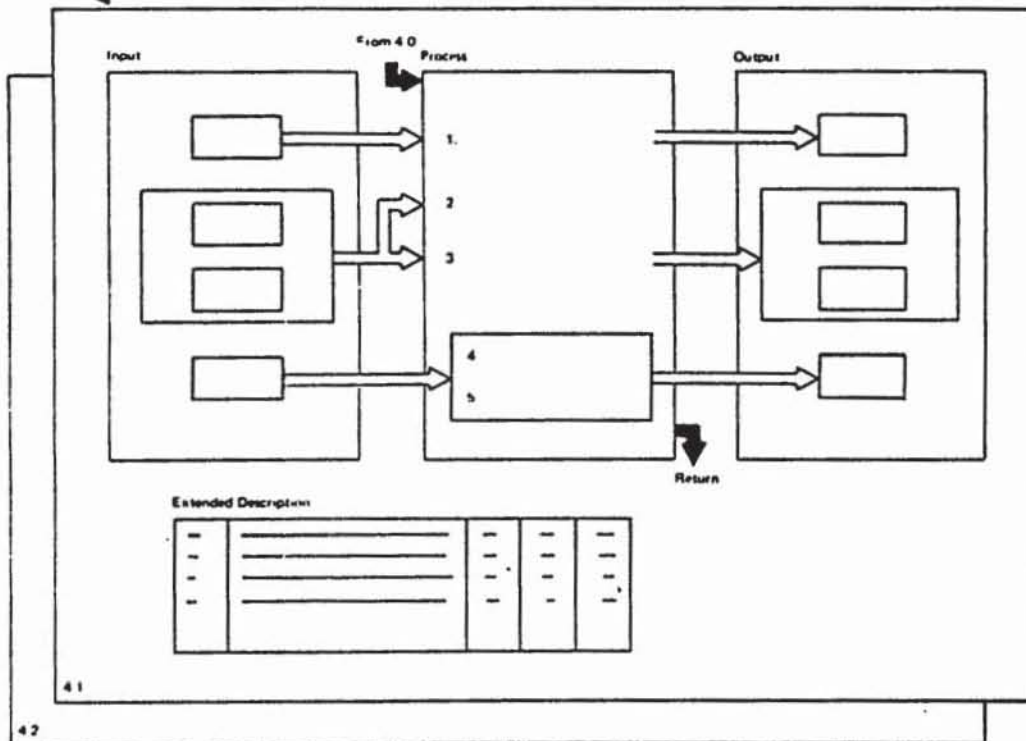
1 A Visual Table of Contents (Strukturübersicht)



2 Overview Diagrams (Überblickdiagramme)



3 Detail Diagrams (Detaildiagramme)



A typical HIPO package

2.1.4 Entwicklungsmethode

HIPO unterstützt die graphische Darstellung eines iterativen top-down-Entwicklungsprozesses, in dem die Strukturübersicht und die IPO-Diagramme parallel erstellt werden.

Die Benutzung der Methode wird erleichtert durch Regeln für den Gebrauch der verschiedenen Diagrammformen und Beispiele.

2.1.5 Literatur

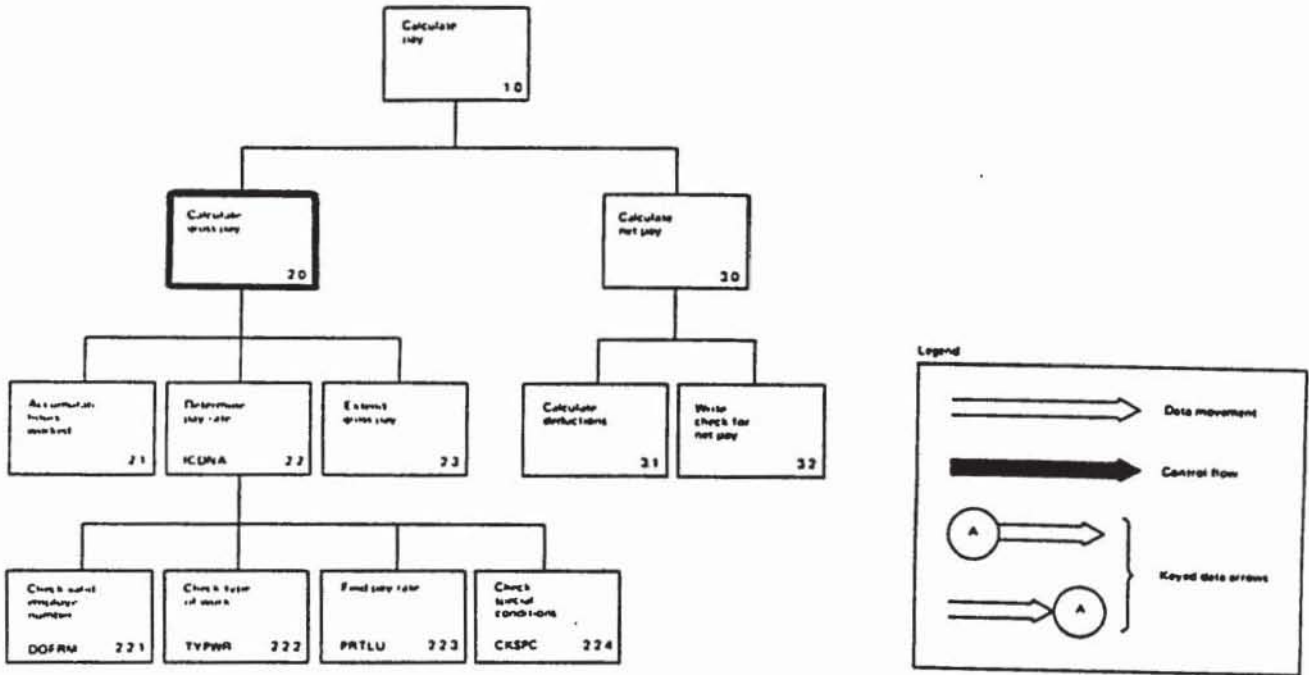
- /1/ HIPO-A Design Aid and Documentation Technique
IBM GC20-1851-1, 1975

- /2/ J.F. Stay
HIPO and integrated program Design
IBM Systems Journal, 15, (1976) pp. 143-154

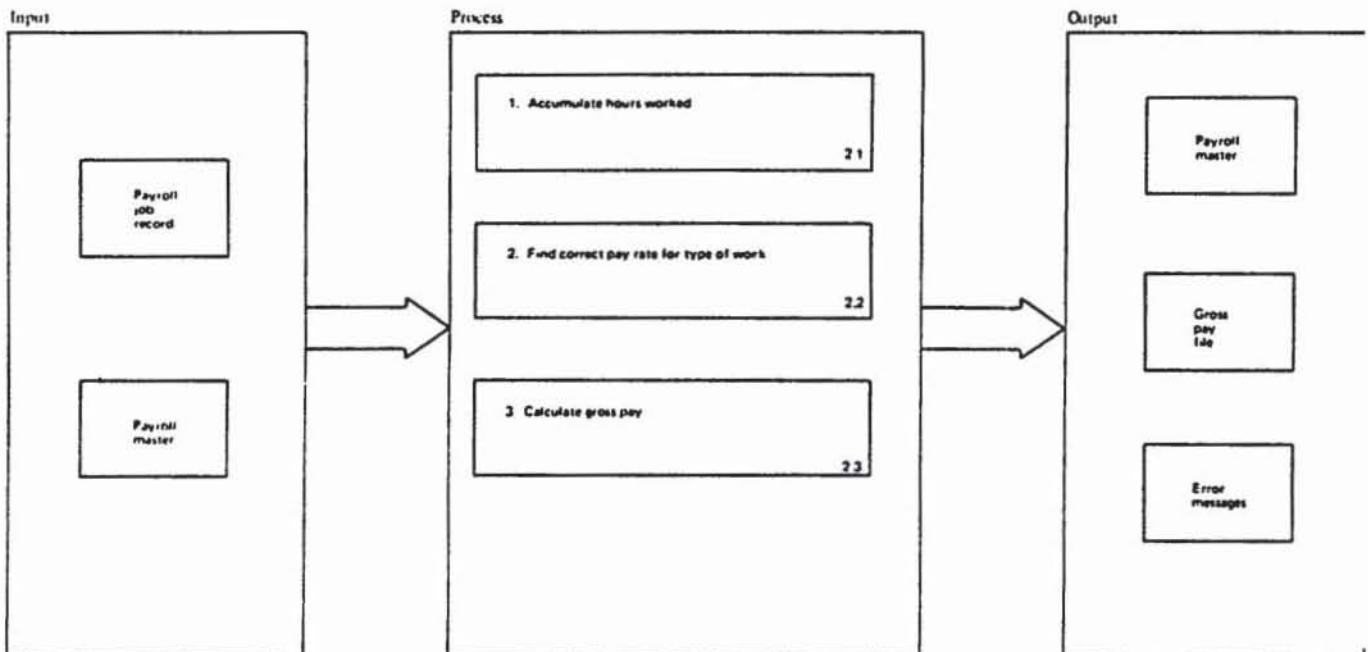
- /3/ M.N. Jones
HIPO for Developing Specifications
DATAMATION, March, 22, (1976) pp. 112-125

2.1.6. Beispiel

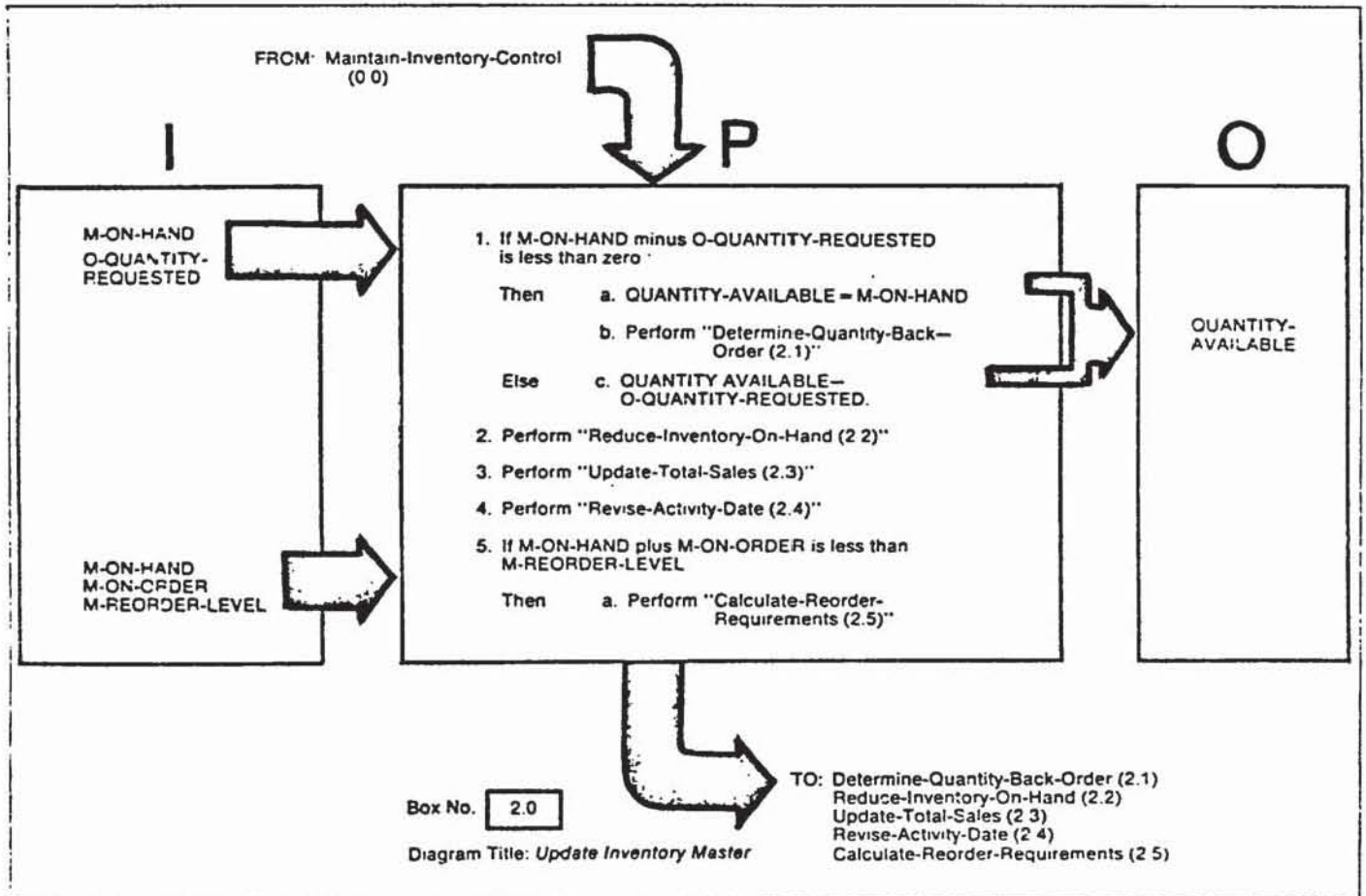
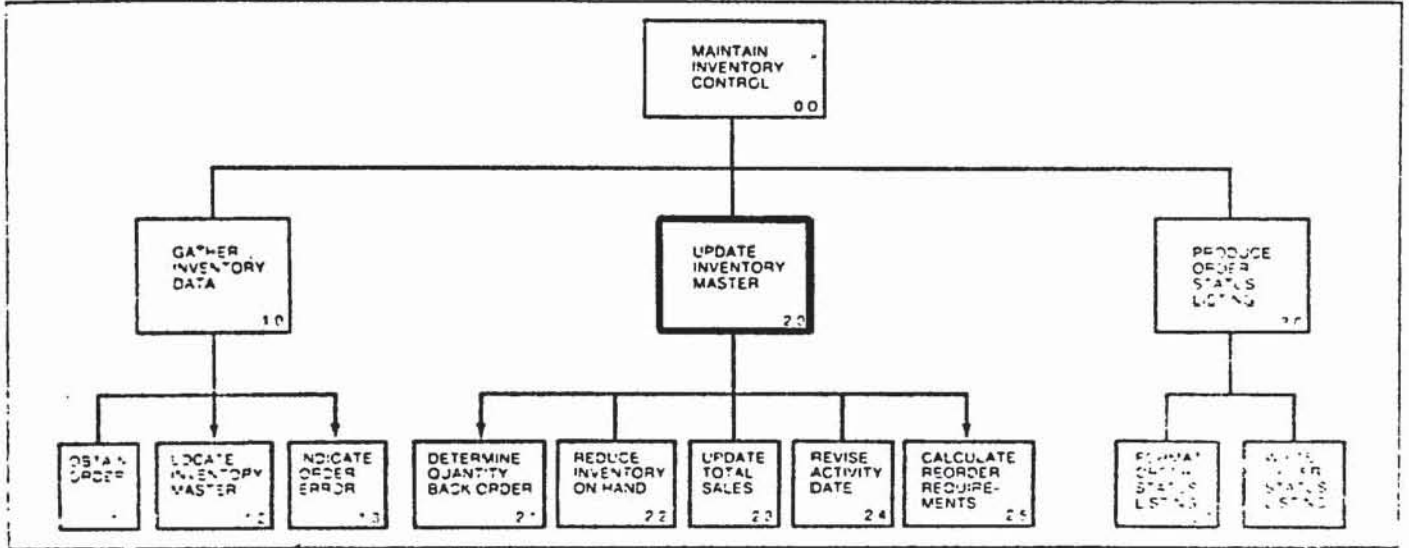
a) Strukturübersicht für das System Calculate pay /1/



Author	J. Renaker	System/Program	Payroll	Date	10/74	Page	1	of	1
Diagram ID	20	Name	Calculate Gross Pay						



b) Strukturübersicht für maintain inventory control /3/



2.2 Higher Order Software (HOS)

2.2.1 Kurzbeschreibung

Die Arbeit von Hamilton und Zeldin über HOS beruht auf Erfahrungen während der Entwicklung von Software für das APOLLO-Programm.

Es handelt sich bei HOS um einen formalen Ansatz zur Beschreibung eines Software-Systems. Ausgangspunkt ist dabei die Definition der Modulschnittstellen. Moduln können in einer Baumdarstellung, dem sogenannten control map, nur so angeordnet werden, daß sie sechs Axiomen genügen.

Diese sechs Axiome scheinen eine notwendige und hinreichende Bedingung für eine systematische Systementwicklung zu sein, die von zuverlässigen Schnittstellen ausgeht. Der Beweis für diese Behauptung steht jedoch noch aus. Durch die Axiome werden Aussagen über bestimmte Schnittstellenaspekte wie Zugriffsrechte, Aufrufstruktur und Zeitverhalten gemacht. Eine Spezifikationsprache mit dem Namen AXES wird entwickelt, deren Ausdrucksmöglichkeiten so gewählt wurden, daß die spezifizierten Schnittstellen den Anforderungen der Axiome entsprechen.

Aus den Axiomen lassen sich drei gültige Zerlegungsstrukturen für die control map ableiten. Ein durch HOS beschriebenes System kann automatisch auf Konsistenz mit den Axiomen und auf Vollständigkeit der Entwurfsbeschreibung überprüft werden.

2.2.2 Anwendungsphase und Gebiet

HOS erhebt den Anspruch, eine Methode zur formalen Beschreibung von Software in allen Phasen der Entwicklung zur Verfügung zu stellen. Da sich die wesentlichen Beschreibungsaspekte jedoch auf Modularisierungskriterien und die damit verbundene Definition von Modulschnitten beziehen, wird das Hauptanwendungsgebiet von HOS in der Entwurfsphase liegen. Die Erprobung von HOS findet u.a. auch auf dem Realzeitgebiet statt.

2.2.3 Inhalt und Form der Darstellung

Die Grundform der Darstellung ist eine mathematische Notation, die in eine graphische Baumstruktur eingebettet ist, in der jeder Knoten eine Funktion im mathematischen Sinne repräsentiert (control map). Die Beziehungen zwischen diesen Knoten (Funktion-Unterfunktion), d.h. die Schnittstellen, unterliegen gewissen Zerlegungskriterien, die im Bild HOS-1 angegeben sind.

Bild HOS-2 enthält als Beispiel eine Gegenüberstellung der HOS-control-map-Darstellung und äquivalenter Formen. Während in den control-maps Funktionen und ihre Relationen dargestellt werden, zeigen die Strukturdiagramme explizit den Kontrollfluß zwischen aufgerufenen Moduln.

Eine formale Sprache AXES ist in Entwicklung. Sie soll geeignet sein, die control-maps und Schnittstellen zu spezifizieren und eine automatische Prüfung auf Widerspruchsfreiheit mit den Axiomen ermöglichen.

COMPOSITION

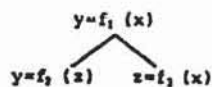


Figure a: An Example of Composition

Consider Figure a

- (1) One and only one function (f_2) controlled by f_1 receives the input data, x , from module f_1 .
- (2) One and only one function controlled by f_1 produces the output data, y , for the module f_1 . In the example of Figure a f_2 produces the output data.
- (3) All other input and output data produced by functions controlled by f_1 reside in local variables. In the example of Figure a, z is a local variable.
- (4) For each change of state of the controller, all functions controlled by f_1 have a change in state.
- (5) Every local variable must exist both as an input variable for one and only one function and as an output variable for one and only one different function on the same level.

CLASS PARTITION

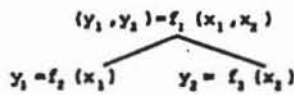


Figure b: An Example of Class Partition

Consider Figure b

- (1) The inputs for each function controlled by f_1 are received directly from f_1 .
- (2) The outputs of all functions controlled by f_1 are produced directly for f_1 .
- (3) For each change in state of a controller, all functions controlled by f_1 have a change in state.
- (4) Each input of f_1 can only be accessed by one function controlled by f_1 .
- (5) No two subfunctions access the same variable.
- (6) There is no communication between functions controlled by f_1 .

SET PARTITION

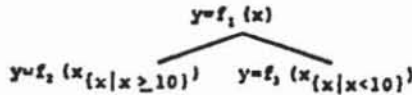
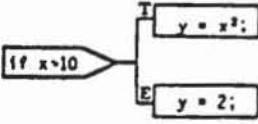


Figure c: An Example of Set Partition

Consider Figure c

- (1) Each function controlled by f_1 produces output data, y .
- (2) All functions controlled by f_1 receive input data from the same variable x .
- (3) Only one function controlled by f_1 has a change in state for a given change in state of the controller.
- (4) The values represented by the input variables of a subfunction represent a subset of values represented by the input variables of the controller.
- (5) There is no communication between functions.

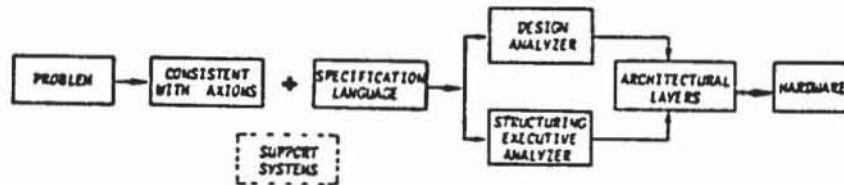
HOS-1 (Fortsetzung)

CONTROL MAP REPRESENTATION	STRUCTURED DESIGN DIAGRAM	HAL SOURCE CODE
$\forall \bar{Q} \bar{Q} = \bar{M}^{-1} \bar{Q} / S = F(\bar{M}, \bar{Q}, S)$	$\bar{Q} = \bar{M}^{-1} \bar{Q} / S;$	$\bar{Q} = \bar{M}^{-1} \bar{Q} / S;$
<p><u>COMMENTS</u></p> <ol style="list-style-type: none"> (1) An assignment statement. (2) A matrix data type (\bar{M}) vector data type (\bar{Q}) and scalar data type (S) and the availability of the corresponding set of operations for these data types are assumed. 		
$y_{\{y x^2\}} = f_{11}(x_{\{x>10\}}) \quad y_{\{y=2\}} = f_{21}(x_{\{x<10\}})$		<pre>if x > 10 then y = x^2; else y = 2;</pre>
<p><u>COMMENTS</u></p> <ol style="list-style-type: none"> (1) If f_{21} were modified so as to contain the relation $x > 20$, the analyzer would detect an extraneous path. (2) Suppose the THEN branch assigns y as shown, but the ELSE branch were to assign a variable other than y. In this case, the analyzer would detect an error. (3) If f_{21} were modified so as to assign a value to x, the analyzer would detect an error. (4) If f_{11} were modified to CALL f_{21}, the analyzer would detect an error. (5) A constant function is specified by restricting the elements of the output set (cf. node f_{21}). 		

CONTROL MAP REPRESENTATION	STRUCTURED DESIGN DIAGRAM	HAL SOURCE CODE
<p>COMMENTS</p> <ol style="list-style-type: none"> (1) Three nested levels of logic flow are shown on the design diagram. (2) Five levels of data flow are shown on the control map. (3) Each function must appear as a node. A relation determines the partitioning of the elements of the input set. If an extraneous path exists due to an inconsistency in the specification itself, the analyzer will not detect such a path. For example, suppose $\sin(x) > 10$ were modified to be $2x < 3$. In this case the analyzer would not detect the fact that f_6 will never be executed. On the other hand, all logical inconsistencies among nested relations are checked for extraneous paths. For example, suppose $\sin(x) > 10$ were modified to be $x < 2$. Here, f_6 would be shown to be logically inconsistent by the analyzer. It is not clear that the removal of all such paths is the best way to completely design a system. But, it is clear that if we know where these paths exist, we can explicitly make a design decision as to whether to remove such a path or not. (4) The use of composition referred to by node f_7 is key to the use of library modules. (5) e is a local variable. 		<pre> if sin x > 10 then if x > 3 do: then y = tan(x); else y = 2x; end; else y = x^2; </pre>
<p>SCHEDULE $F_1 > F_2 > F_3$</p> <p>COMMENTS</p> <ol style="list-style-type: none"> (1) This set of HAL statements implements the invocation of processes at a given level so that the priorities of the processes are ordered and are relatively lower than the controller process. (2) F is an array of name variables. P is an array of numbers. The values assigned to P are ordered from highest to lowest. The assignment of values (rather than a relational form of priority) is necessary because of the restrictions of HAL. (3) The analyzer assumes Designate Priority to be a lower virtual layer function, i.e., Designate Priority does not appear on the control map. (4) In this case, the analyzer checks the syntax in that the number of scheduled processes must be consistent with the number of processes assigned a priority by Designate Priority. (5) See Appendix I for a description of a real-time example using this concept. 		<pre> do; CALL Designate Priority (F) Assign P; Schedule F1 at priority P1; Schedule F2 at priority P2; Schedule F3 at priority P3; end; </pre>

2.2.4 Entwicklungsmethode

Die wesentlichen Komponenten der HOS-Methode sind im nachfolgenden Bild dargestellt:



Der Entwurf von Software wird durch die HOS-Axiome und daraus abgeleitete Modularisierungsregeln unterstützt. Eine Spezifikationsprache, die diesen Regeln genügt, erleichtert die Formulierung des Entwurfs (AXES).

HOS-AXIOME:

- AXIOM 1: A GIVEN MODULE CONTROLS THE INVOCATION OF THE SET OF VALID FUNCTIONS ON ITS IMMEDIATE, AND ONLY ON ITS IMMEDIATE, LOWER LEVEL
- AXIOM 2: A GIVEN MODULE IS RESPONSIBLE FOR ELEMENTS OF ONLY ITS OWN OUTPUT SPACE
- AXIOM 3: A GIVEN MODULE CONTROLS THE ACCESS RIGHTS TO EACH SET OF VARIABLES WHOSE VALUES DEFINE THE ELEMENTS OF THE OUTPUT SPACE FOR EACH IMMEDIATE, AND ONLY EACH IMMEDIATE, LOWER LEVEL FUNCTION
- AXIOM 4: A GIVEN MODULE CONTROLS THE ACCESS RIGHTS TO EACH SET OF VARIABLES WHOSE VALUE DEFINE THE ELEMENTS OF THE INPUT SPACE FOR EACH IMMEDIATE, AND ONLY EACH IMMEDIATE, LOWER LEVEL FUNCTION
- AXIOM 5: A GIVEN MODULE CAN REJECT INVALID ELEMENTS OF ITS OWN, AND ONLY ITS OWN, INPUT SET
- AXIOM 6: A GIVEN MODULE CONTROLS THE ORDERING OF EACH TREE FOR THE IMMEDIATE, AND ONLY THE IMMEDIATE, LOWER LEVEL

Die Entwicklung eines Entwurfs, der mit den Axiomen konsistent ist, wird durch folgende Strukturierungsregeln unterstützt:

- EVERY DECISION MADE WITHIN A MODULE IS DIRECTLY RELATED TO THE MODULE FUNCTION (e.g., a moding module would not depend on pilot response procedures)
- DATA PRODUCED PER FUNCTION IS RELATED ONLY TO MODULE FUNCTION (e.g., a navigation module should not produce information that should be produced by a guidance module)
- A MODULE SHOULD NOT SPECIFY HOW IT IS TO BE USED (e.g., a module should not schedule itself)
- A MODULE MAY ONLY INVOKE LOWER LEVEL MODULES
- A MODULE MAY ONLY INVOKE OTHER MODULES TO PERFORM ITS PARTICULAR FUNCTION (e.g., navigation does not invoke a guidance function)
- A FUNCTION (and resulting modules) SHOULD BE DESIGNED WITH SUB-FUNCTIONS, TOP-DOWN
- KNOWN INFORMATION ON A HIGHER LEVEL SHOULD NOT BE BROUGHT UP AGAIN AT A LOWER LEVEL. FOR EXAMPLE, IF A MODE IS ALREADY KNOWN, IT SHOULD NOT BE NECESSARY TO INTERROGATE THE MODE
- A FUNCTION WHICH IS REQUIRED TO DEAL WITH ASYNCHRONOUS EVENTS SHOULD BE DESIGNED TO BE ASYNCHRONOUS, i.e., ARTIFICIAL TIME CONSTRAINTS SHOULD NOT BE FORCED
- ARTIFICIAL CONTROL CONSTRAINTS SHOULD NOT BE IMPOSED ON A MODULE (e.g., if a higher level guidance function can call a common routine before being subdivided into lower level guidance functions, it is better than having that routine called individually by each lower level guidance function)
- FLEXIBILITY (developmental and real-time) IS A MAJOR CONCERN. IT SHOULD NOT BE COMPROMISED FOR EFFICIENCY (time and memory) UNLESS IT BECOMES AN ABSOLUTE NECESSITY

Ein Analysator für AXES-Spezifikationen soll in der Lage sein, neben der Syntaxprüfung eine Konsistenzanalyse zwischen den spezifizierten Schnittstellen und den HOS-Axiomen durchzuführen und geeignete Fehlermeldungen auszugeben.

2.2.5 Literatur

M. Hamilton, S. Zeldin

Higher Order Software Technique applied to a space shuttle prototype program,

Lecture Notes in Comp. Science, Vol. 19, 1974, pp. 19-31

M. Hamilton, S. Zeldin

Higher Order Software - A Methodology for Defining Software

IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, 1976, pp. 9-32

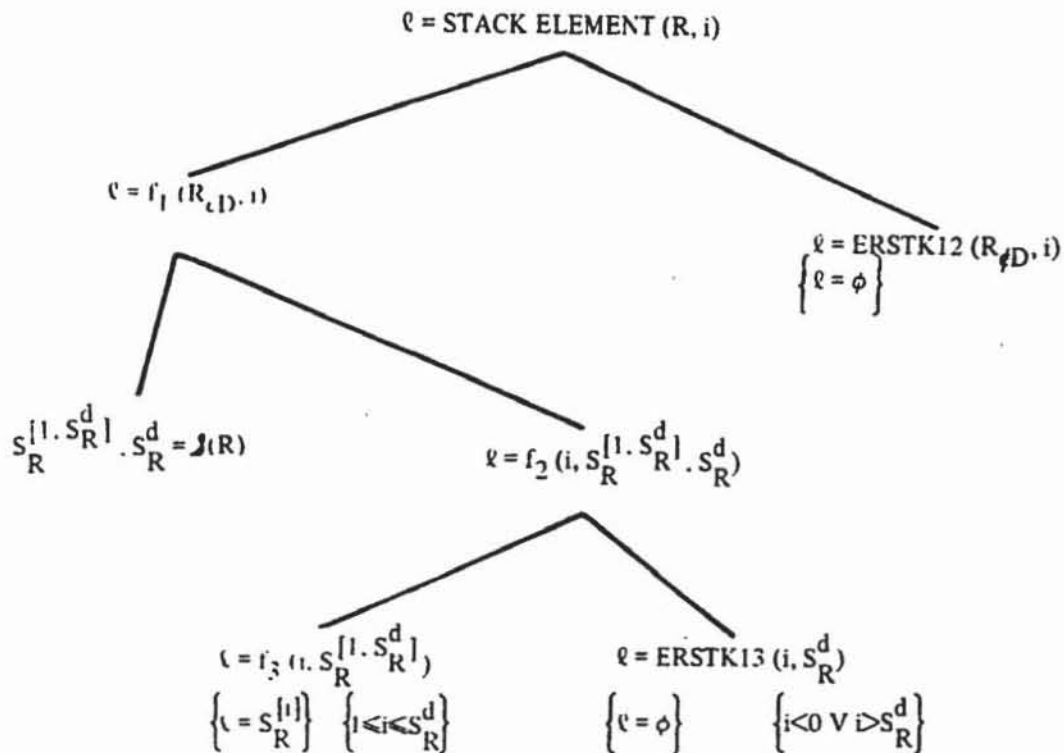
2.2.6 Beispiel

Zur Klärung des HOS-Spezifikationsbeispiels soll die Parnas Modul-Spezifikation herangezogen werden (HOS-3). Stackelement (R,i) ist ein Teil einer Modulspezifikation zur Verwaltung eines Kellers und liefert den Wert des i -ten Elements des Kellers R .

Notation:

- R - The referent to a stack (in place of stack name, SN)
- S_R - The stack referred to by R
- S_R^d - The current depth of stack S_R
- S_R^m - The maximum allowable depth for S_R
- $S_R^{[i]}$ - The i^{th} element of stack S_R
- $S_R^{[i,j]}$ - The i^{th} thru j^{th} elements of S_R
- D - The directory implied in creating, deleting, etc.

Das Beispiel HOS-4 stellt den Entwurf eines Systems zur Verarbeitung von Botschaften dar. Da dies das einzige uns zugängliche größere Anwendungsbeispiel ist, jedoch nähere Informationen fehlen, kann es nur einen groben Eindruck einer HOS-Anwendung vermitteln.



HOS-Spezifikation der Kellerooperation 'stackelement'

Function STACKELEMENT

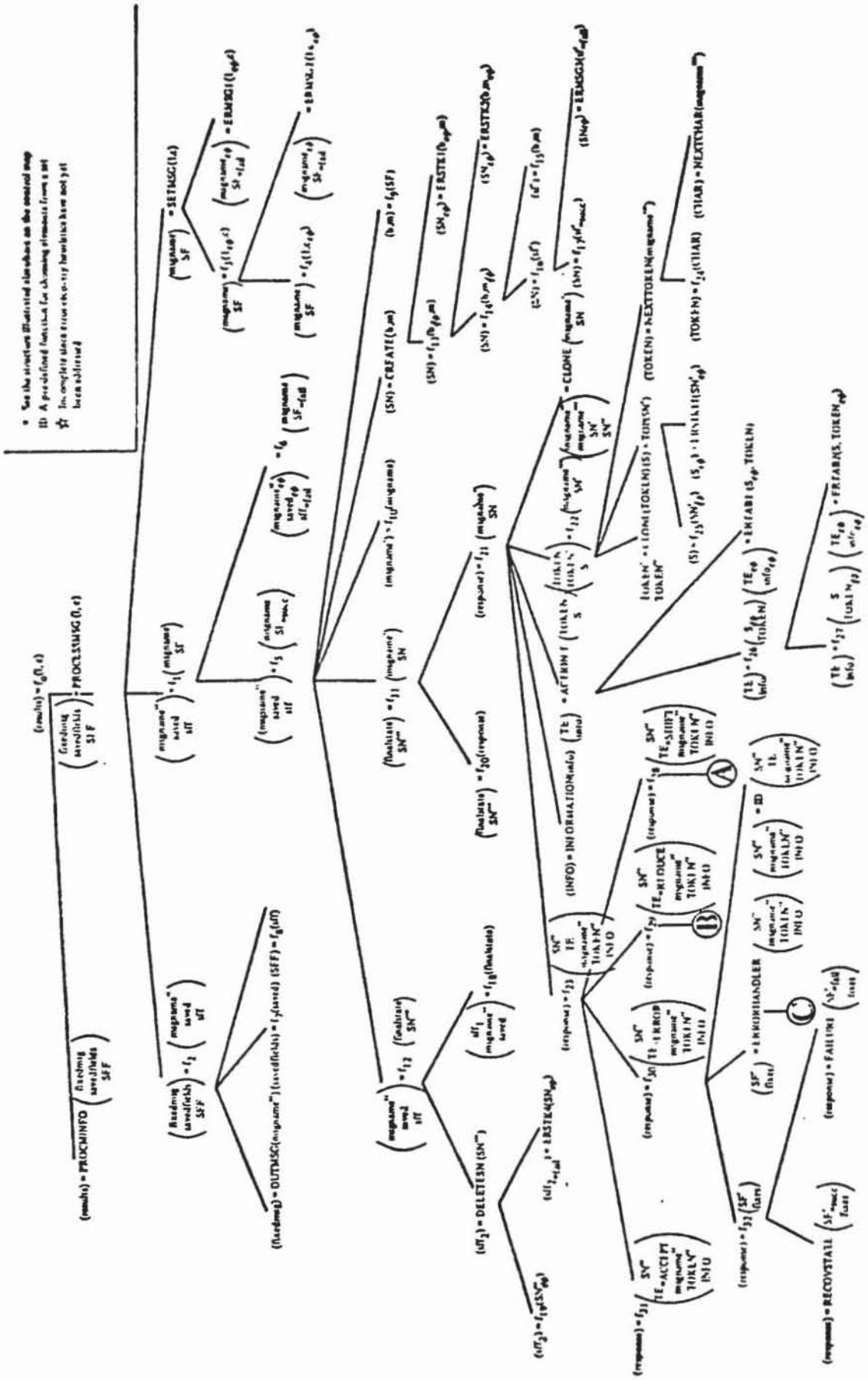
possible values: stack elements (integers whose range is set for each implementation)

initial value: none

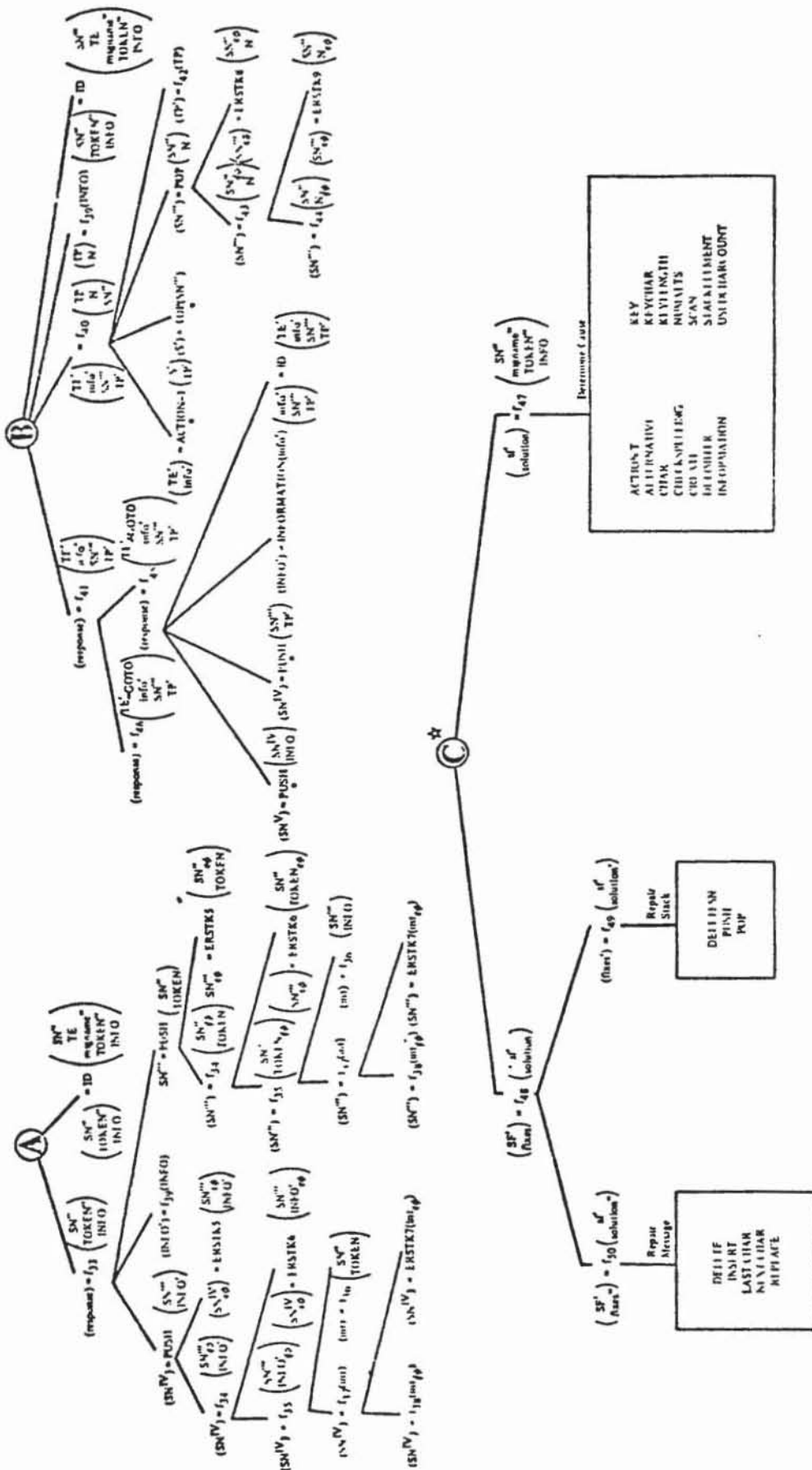
parameters: sn, a stack name
i, $1 \leq i \leq \text{DEPTH}(sn)$

effect: call ERSTK12 if sn is not a valid stack name;
call ERSTK13 if i is out of range;
returns the value of the ith element on stack sn;
i=1 gives the bottom element of the stack (placed there at the time of the stack's creation);
i=DEPTH(sn) gives the current top element;
this is included only for the convenience of the ERR module.

PARNAS-Spezifikation der Kellerooperation 'stackelement'



SANTONI, P. A., "A REPORT ON THREE METHODOLOGIES/TOOLS FOR SYSTEM CONCEPT AND DESIGN"
 NAVAL ELECTRONICS LABORATORY CENTER, SAN DIEGO, CA 92152, NELC N725, 17 NOVEMBER 1976.



2.3 Jackson-Design-Methodology (JDM)

2.3.1 Kurzbeschreibung

JDM ist eine Technik für den Programmentwurf. Ihr Kern ist die Auffassung, daß sich ein zu entwickelndes komplexes Programm darstellen läßt als eine Reihe 'einfacher Programme', die über sequentielle Files kommunizieren. Dabei ist relativ genau definiert, was ein 'einfaches Programm' ist.

Der Methode liegen zwei Ideen zugrunde:

- a) Nach den schlechten Erfahrungen mit effizienten Programmen, deren Fehler nur mangelhaft lokalisiert und beseitigt werden können, sollten Programme besser erst korrekt, dann effizient gemacht werden.
- b) Ein Programm ist nur dann flexibel und durchschaubar, wenn es die Realität adäquat repräsentiert, also nicht nur das gewünschte Ein-/Ausgabeverhalten hat, sondern auch intern ein geeignetes Modell der Realität darstellt.

Das Ziel der JDM ist entsprechend diesen Ideen die Lösung der gestellten Aufgabe durch ein geeignetes Modell, dessen Struktur und Elemente einfach sind. Für die Implementierung werden Anweisungen gegeben. Wesentlich ist dabei das Verfahren der 'Inversion eines Teilprogramms relativ zu einem anderen'. Damit ist die Umwandlung eines Programmteils in ein Unterprogramm gemeint.

2.3.2 Anwendungsphase und Gebiet

Die JDM betrifft nur den Entwurf der Software. Sie ist offenbar dann besonders geeignet, wenn Ein- und Ausgabe des Programms sich durch sequentielle Files darstellen lassen. Zwar gibt Jackson auch eine Darstellung für Interrupts an, doch scheint dabei das Verfahren überstrapaziert.

2.3.3 Inhalt und Form der Darstellung

Da es sich bei der JDM um eine Technik handelt, die nur manuell verwendet wird und in deren Zentrum keine bestimmte Darstellung, sondern eine bestimmte Betrachtungsweise liegt, spielt die Form keine wesentliche Rolle. Jackson läßt nur solche Datenstrukturen zu, die sich in geordneten Bäumen darstellen lassen. Ein hochgestellter Kreis bezeichnet Unterbäume, die alternativ sind (UNION in ALGOL 68), ein Stern solche, die beliebig oft stehen oder auch fehlen können (siehe 2.3.6).

2.3.4 Entwicklungsmethode

Der Entwurf nach der JDM geht von den Datenstrukturen in Input- und Output-Files aus. Lassen sich beide miteinander vereinbaren, so entsteht direkt ein 'einfaches Programm', das das Problem löst. Das 'einfache Programm' ist bei Jackson wie folgt definiert:

"A 'simple program' has the following attributes:

- The program has a fixed initial state; nothing is remembered from one execution to the next.
- Program inputs and outputs are serial files, which we may conveniently suppose to be held on magnetic tapes. There may be more than one input and more than one output file.
- Associated with the program is an explicit definition of the structure of each input and output file. These structures are tree structures, defined in the grammar used above. This grammar permits recursion in addition to the features shown above; it is not very different from a grammar of regular expressions.

- The input data structures define the domain of the program, the output data structures its range. Nothing is introduced into the program text which is not associated with the defined data structures.
- The data structures are compatible, in the sense that they can be combined into a program structure in the manner shown above.
- The program structure thus derived from the data structures is sufficient for a workable program. Elementary operations of the program language (possibly supplemented by more powerful or suitable operations resulting from bottom-up design) are allocated to components of the program structure without introducing any further 'program logic'."

Im allgemeinen gibt es aber Struktur-Konflikte, d.h. Ein-/- und Ausgabefiles lassen sich nicht ohne Zerstörung ihrer Strukturen überlagern ('structure clash'). In diesem Fall werden ein oder mehrere Puffer eingeführt ('intermediate files'), zwischen denen jeweils eine einfache Transformation genügt. Die Programmteile, die diese Transformation realisieren, stehen zueinander in einem Koroutinen-Verhältnis.

Zur Realisierung in den gängigen Programmiersprachen werden die Koroutinen 'invertiert', d.h. von zwei Koroutinen wird eine zum Unterprogramm der anderen. Dabei tritt als Schwierigkeit auf, daß das Unterprogramm einen activation-record z.B. in Form von own-Variablen braucht.

Auch die Fehlerbehandlung wird von der JDM erfaßt. Für den üblichen Konfliktfall, daß ein Fehler erst erkannt werden kann, wenn die Verzweigung im Programm schon erfolgt sein müßte, gibt Jackson folgende Strategie an:

- a) Program so erstellen, als ob an der Verzweigung ein 'freundlicher Geist' den Ablauf steuert, d.h. die Wahl zwischen korrekt- und inkorrekt-Alternative trifft.
- b) Mißtrauen gegen den Geist entwickeln, d.h. Prüfungen in die korrekt-Alternative einfügen, wo nötig, nach Erkennung eines Fehlers Sprünge zur inkorrekt-Alternative einsetzen.

- c) Soweit nötig dafür sorgen, daß Auswirkungen der korrekt-Alternative rückgängig gemacht werden, falls ein Fehler erkannt wurde.

In diesem Zusammenhang betont Jackson, daß der Sprung (goto) hier dem Problem genau angemessen ist und daher nicht krampfhaft vermieden werden sollte.

Für den Erfolg der JDM ist offenbar maßgeblich, daß der Anwender in einer Schulung 'Kochrezepte' bekommt, die ihn scheinbar aus seiner Unsicherheit gegenüber den modernen Verfahren der Strukturierung erlösen und für einen großen Bereich kommerzieller DV einen Weg weisen, brauchbare Programmentwürfe zu erzielen.

2.3.5 Literatur

- /1/ Jackson, M.A.
Principals of Program Design,
Academic Press, 1975
- /2/ Jackson, M.A.
'Constructive Methods of Program Design',
Proceedings of the First Conference of the European Cooperation
in Informatics, 1976,
Lecture Notes in Computer Science, No. 44, Springer-Verlag, 1976
pp. 236-262
- /3/ McGowan, C.L.; Kelly, J.R.;
A Review of Decomposition and Design Methodologies Reliable
Software
INFOTECH, London 1977

2.3.6 Beispiel

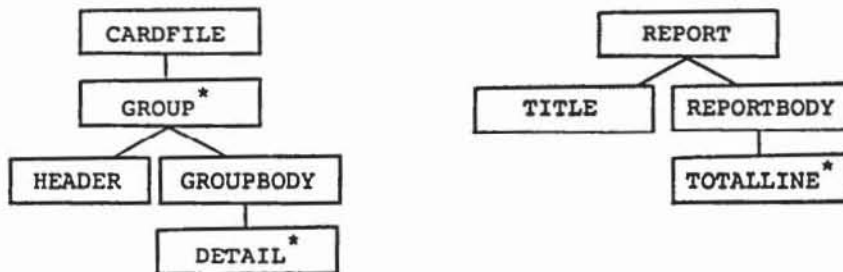
Nachfolgend ist ein kleiner Ausschnitt eines größeren, in /1/ ausführlich diskutierten Beispiels wiedergegeben. Dieser Teil zeigt die Auflösung eines Struktur-Konflikts.

Die Aufgabe sei folgende:

Ein Kartenstapel sei zu verarbeiten.

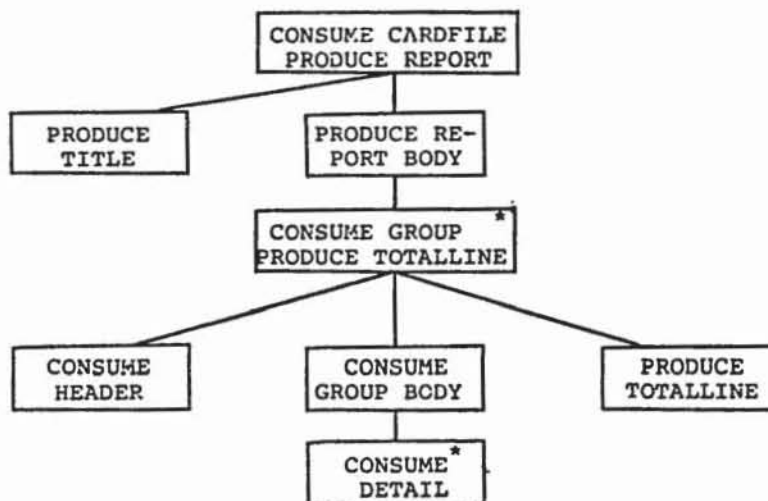
Er enthält Gruppen von Karten, deren gemeinsames Kennzeichen jeweils ein Schlüsselwort ist. Jede Gruppe beginnt mit einer Kopfkarte; die folgenden Karten enthalten jeweils eine Zahl.

Es soll ein Protokoll gedruckt werden, das die Summe der Zahlen jeder Gruppe mit den zugehörigen Schlüsselwörtern ausgibt. Die Struktur der Ein- und Ausgabedaten ist wie folgt:



JDM-2

Aus diesen kompatiblen Datenstrukturen wird die folgende Programmstruktur abgeleitet.

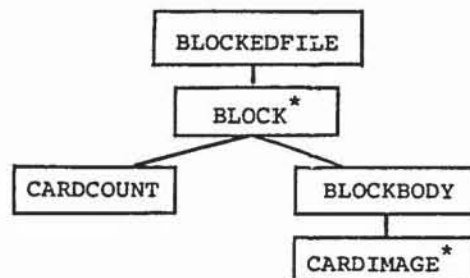


JDM-3

Das Programm läßt sich nun etwa wie folgt realisieren:

```
CARDFILE-REPORT sequence
    open cardfile; read cardfile; write title;
REPORT-BODY iteration until cardfile.eof
    total := 0; groupkey := header.key;
    read cardfile;
GROUP-BODY iteration until cardfile.eof or
    detail.key ≠ groupkey
    total := total + detail.amount;
    read cardfile;
GROUP-BODY end
    write totalline (groupkey, total);
REPORT-BODY end
    close cardfile;
CARDFILE-REPORT end
```

Die Aufgabe wird nun dadurch erschwert, daß die Eingabe in einem geblockten File steht und damit (ohne Korrespondenz zur logischen Struktur) zusätzlich folgende Struktur hat:



JDM-4

Jetzt sind die Datenstrukturen nicht mehr kompatibel, ein Struktur-Konflikt liegt vor. Die für die JDM typische Lösung besteht darin, daß ein Zwischenfile eingeführt wird, der die Eingabe des vorher entworfenen Programms PB und die Ausgabe eines zusätzlichen Teils PA enthält.

JDM-5



PA kann wie folgt aussehen:

```
PA sequence
  open blockedfile; open fileX; read blockedfile;
PABODY iteration until blockedfile.eof
  cardpointer := 1;
  PBLOCK iteration until cardpointer > block.cardcount
    write cardimage (cardpointer);
    cardpointer := cardpointer + 1;
  PBLOCK end
  read blockedfile;
PABODY end
  close fileX; close blockedfile;
PA end
```

Zur Realisierung der Programme, z.B. in ALGOL, muß PA in ein Unterprogramm von PB gewandelt werden oder umgekehrt. ("PA wird invertiert relativ zu PB").

2.4 LOGOS

2.4.1 Kurzbeschreibung

Das LOGOS-System wurde seit 1969 an der Case Western Reserve University in Cleveland, Ohio, mit Unterstützung des Verteidigungsministeriums entwickelt. Es dient zur Beschreibung von Rechner-Systemen. Dieser Ausdruck soll hier die Kombination von Hard- und Software bezeichnen. Das Ziel ist die hierarchische Darstellung der Funktionen ohne Berücksichtigung der Frage, ob die Realisierung durch Hard-, Firm- oder Software geschehen soll. Dadurch erreicht man eine uniforme Beschreibung.

Nach der jüngsten Arbeit über LOGOS, die uns vorliegt (/1/), kann die Definition eines Systems in der LOGOS-Sprache automatisch verarbeitet und auf Konsistenz geprüft werden; ein Simulator zur Ermittlung der System-Leistung war entworfen, die weitere Unterstützung der Hard- und Software-Realisierung durch LOGOS geplant.

2.4.2 Anwendungsphase und Gebiet

Wie oben geschildert soll LOGOS in der System-Entwurfsphase eingesetzt werden, und zwar insbesondere dort, wo die Unterscheidung zwischen Hard-, Firm- und Software noch nicht getroffen wurde.

Die Vorteile dieses Systems werden besonders im Grenzbereich zwischen Hard- und Software sichtbar, z.B. beim Entwurf von Gleitkommprozessoren oder bei Emulationen. Im Sinne der übrigen hier vorgestellten Entwurfssysteme geht es also um ein niedriges Abstraktionsniveau. LOGOS wird dennoch einbezogen wegen der interessanten Darstellungsweise.

2.4.3 Inhalt und Form der Darstellung

In LOGOS wird ein System dargestellt durch Datendeklarationen und zwei Graphen, den Data Graph (DG) und den Control Graph (CG). Die Graphen können hierarchisch gegliedert sein, d.h. Teil-Graphen können aus der Beschreibungsebene herausgenommen werden.

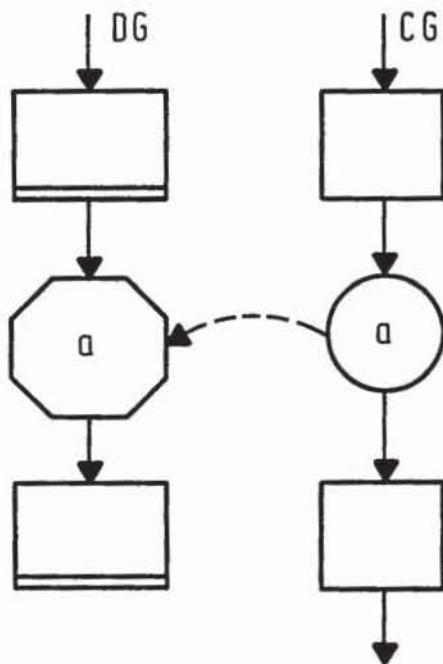
Alle Daten werden deklariert, ausgehend vom Standardtyp BIT.
Weitere Typen können rekursiv eingeführt werden, z.B. als COMPLEX
(im Sinne einer ALGOL-68-Structure), ARRAY, REFERENCE.

DG und CG sind Graphen auf der Basis von Petri-Netzen. Der DG zeigt
den reinen Datenfluß ohne alle Information zum Ablauf, also zur Aus-
wahl und Reihenfolge der Operationen. Diese Angaben sind im CG kon-
zentriert.

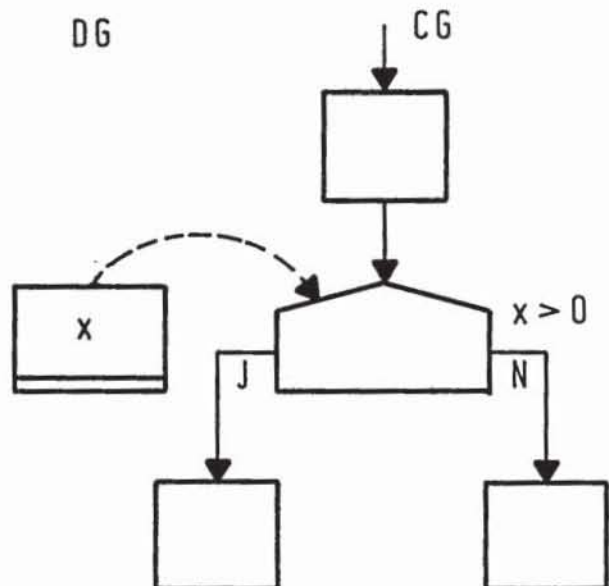
Der Zusammenhang wird hergestellt

- (a) durch Transitionen im CG, die dieselbe Bezeichnung tragen wie
Operationsknoten im DG und deren Feuern die Ausführung der Ope-
ration veranlaßt;
- (b) durch Verzweigungen im CG, die von Werten im DG gesteuert werden.

zu a:



zu b:



Für die CGen gibt es eine Reihe von control-operators (AND, OR, PREDICATE, BLOCK-HEAD und -END).

Im DG werden außer den Operationsknoten Rechtecke für Daten und Sechsecke für Referenzen (Variablen) verwendet.

Im CG lassen sich parallele und sequentielle Abläufe übersichtlich darstellen.

2.4.4 Entwicklungsmethode

LOGOS ist ein Beschreibungs- und Analysesystem; über die Entwicklungsmethode wird nichts ausgesagt. Die Möglichkeit zu hierarchischer Strukturierung unterstützt aber einen top-down-Entwurf.

2.4.5 Literatur

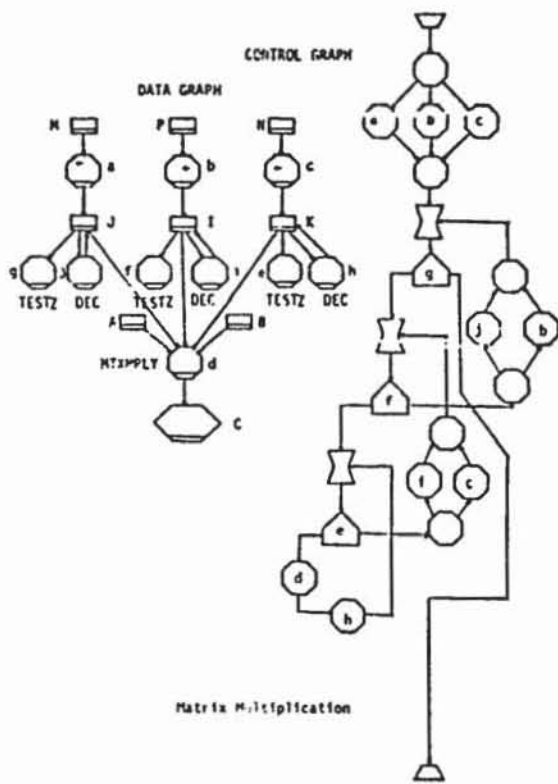
- /1/ Rose, C.W.; Albarran, M.;
Modelling and Design Description of Hierarchical
Hardware/Software Systems
12th Annual Design Automation Conference 1975,
Boston, Mass., June 23-25, pp. 421-430,
New York, N.Y.: IEEE 1975

- /2/ Rose, C.W.;
LOGOS and the Software Engineer
FJCC, Anaheim, Calif., Dec. 5-7, 1972, pp. 311-323

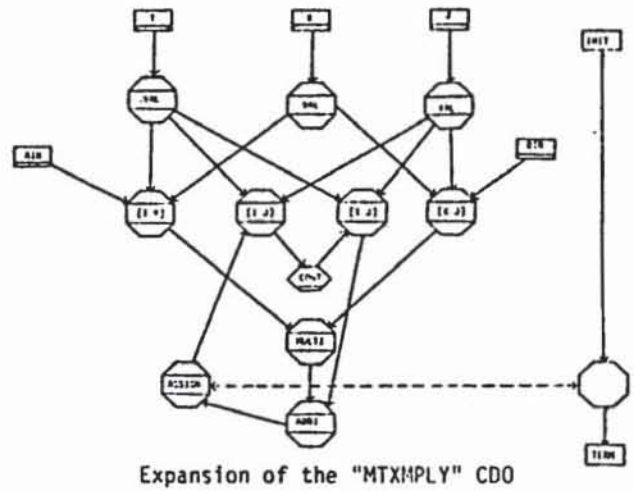
- /3/ Heath, F.G.;
Project LOGOS - A Computer-aided Design System for
integrated Software and Hardware
IEEE-Conference, Publ. No. 86. 1972

2.4.6 Beispiel

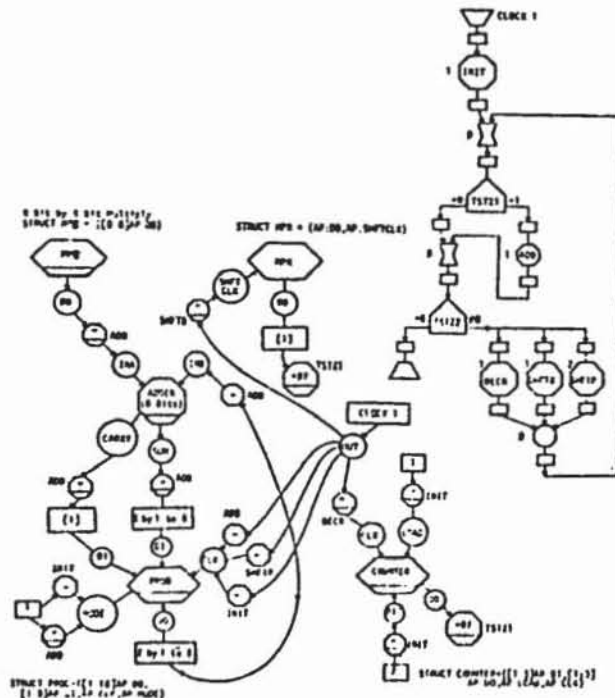
Das folgende Beispiel ist aus /1/ entnommen. Gezeigt wird eine Matrix-Multiplikation, deren Ablauf in drei Ebenen dargestellt ist: In der obersten Ebene ist der Kern der Operation, das Aufaddieren der skalaren Produkte, durch MTXMPPLY repräsentiert. Dieses Element ist dann aufgelöst, wobei noch ein nicht-primitiver Operator MULTI verbleibt. Dieser ist schließlich bis auf Bit-Ebene dargestellt.



Matrix Multiplication



Expansion of the "MTXMPPLY" CDO



Binary Multiplication

2.5 Modular Approach to Software Construction, Operation and Test

(MASCOT)

2.5.1 Kurzbeschreibung

Nach dem MASCOT-Ansatz wird der Grobentwurf eines Software-Systems für Realzeitanwendungen durch ein Netzwerk aus kooperierenden Prozessen und Datenstrukturen beschrieben und in einer graphischen Form dargestellt. Die parallelen Prozesse werden Aktivitäten genannt, als Kreise \bigcirc dargestellt und durch Pfeile mit den Datenstrukturen verbunden, auf die sie zugreifen. Die Datenstrukturen werden eingeteilt in Kanäle (I) und Speicher (\sqcup). Kanäle dienen vor allem zum Austausch von Botschaften zwischen Aktivitäten (Hersteller/Verbraucher-Beziehung), während Speicher die Zustandsdaten des Systems enthalten. Die einzige erlaubte Kommunikation zwischen Aktivitäten findet über die ihnen durch das Netzwerk zugeordneten Kanäle und Speicher statt. Da bei Realzeitanwendungen die Kommunikation zwischen Aktivitäten im wesentlichen asynchron abläuft, müssen die Kanäle und Speicher Synchronisationsmöglichkeiten enthalten.

Das MASCOT-Betriebssystem stellt zu diesem Zweck primitive Synchronisationselemente zur Verfügung (Monitor-Konzept). Kanäle und Speicher werden jedoch mit Hilfe der speziellen höheren Programmiersprache MORAL als abstrakte Datentypen derart definiert, daß ihre detaillierte Struktur und die Benutzung der primitiven Synchronisationselemente verdeckt wird. Dies wird durch spezielle Zugriffsprozeduren erreicht.

Kanäle, Speicher und ihre Zugriffsprozeduren zusammengenommen beschreiben eine wohl definierte Abstraktionsebene (virtuelle Maschine), von der ausgehend die Aktivitäten entworfen werden können.

2.5.2 Anwendungsphase und Gebiet

MASCOT unterstützt den modularen Entwurf und die Implementierung zuverlässiger Software vor allem für Realzeitanwendungen. Durch die Beschreibung des Entwurfs mit Hilfe wohl definierter Grundelemente und einer gepufferten Kommunikation zwischen den einzelnen Prozessen mit zugeordneten 'high-level'-Synchronisationsoperationen gelangt man zu einem sicheren und leicht überprüfbareren Softwaremodell für Realzeitanwendungen.

2.5.3 Inhalt und Form der Darstellung




Die größte Beschreibungseinheit des MASCOT-Entwurfsmodells ist ein MASCOT-Subsystem. Ein Subsystem besteht aus einer oder mehreren Aktivitäten, denen ein oder mehrere Kommunikationsbereiche zugeordnet sind. Die Verarbeitungsfolge einer Aktivität wird durch die Zuordnung einer sogenannten root-procedure definiert. Die formalen Parameter der root-procedure spezifizieren die Zahl und den Typ der Kommunikationsbereiche, auf die die Aktivität Zugriff hat.

Es gibt zwei Arten von Kommunikationsbereichen:

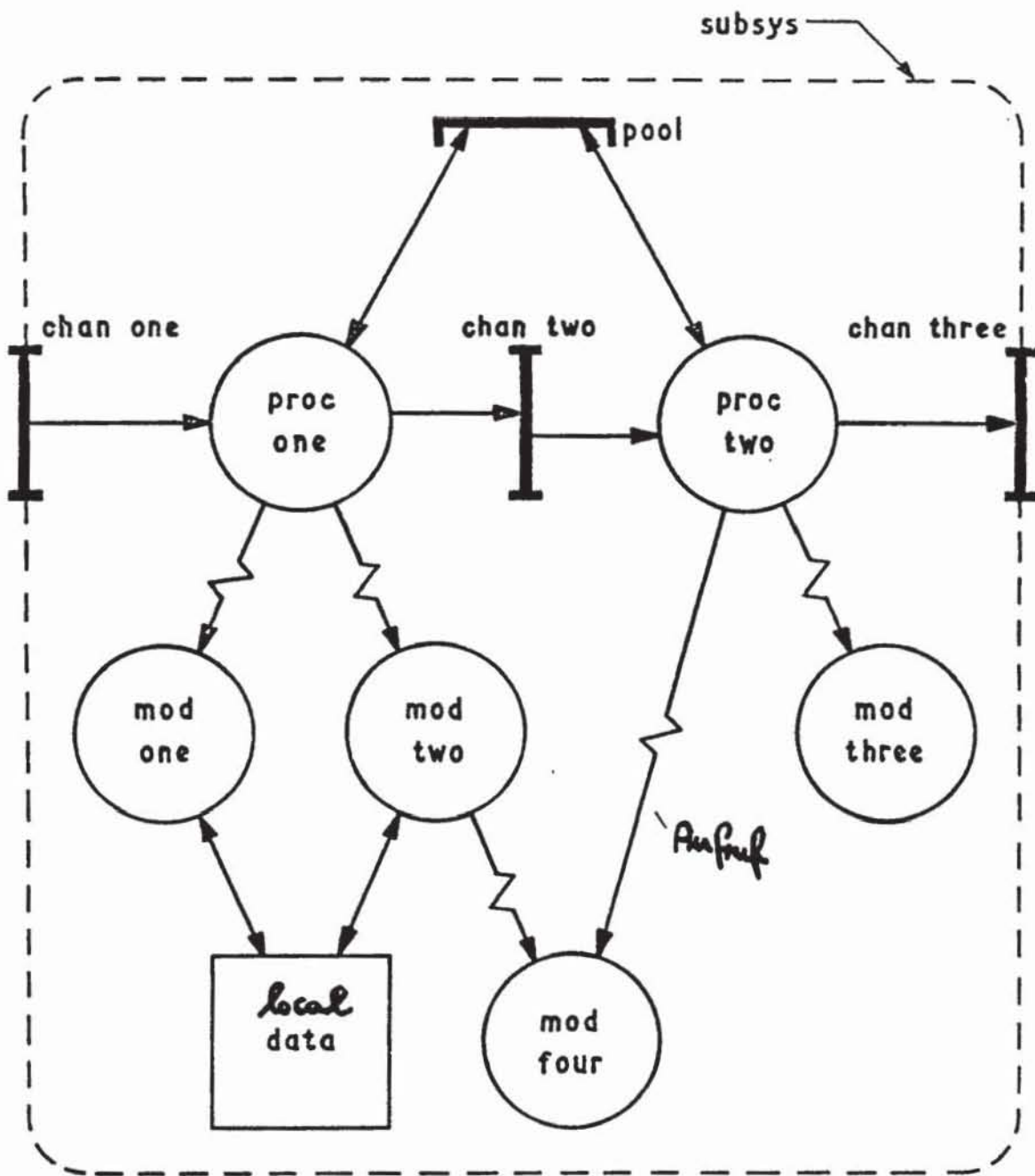
- Kanäle: Dies ist eine Datenstruktur (Puffer) zum Austausch von Botschaften zwischen kooperierenden Prozessen. Jeder Zugriff auf einen Kanal erfolgt über eine Zugriffsprozedur, die unterstützt durch das MASCOT-Betriebssystem den Datenfluß durch den Kanal überwacht.
- Speicher: Dies ist eine Datenstruktur, die Zustandsinformationen des Systems enthält und auf die von einer oder mehreren Aktivitäten aus zugegriffen werden kann.

Ein MASCOT-Subsystem besteht also aus den Systemelementen Aktivität (root-procedure), Kanal und Speicher. Diese Systemelemente können weiterhin aus MASCOT-Moduln aufgebaut sein. Ein MASCOT-Modul ist die Grundeinheit der Programmierung. Meist werden Kanäle und Speicher jeweils durch einen Modul beschrieben, während root-procedures aus mehreren Moduln zusammengesetzt sind.

Zur graphischen Darstellung des Entwurfs werden für die einzelnen Systemelemente unterschiedliche Symbole eingeführt:

- Aktivität: Sie wird als Kreis  gezeichnet, der den Namen der zugeordneten root-procedure einschließt;
- Kanal: Er wird als das Symbol  gezeichnet und der Name der zugeordneten Datenstruktur angehängt;
- Speicher: Er wird als das Symbol  gezeichnet und der Name der zugeordneten Datenstruktur angehängt.

Im nächsten Bild ist das Beispiel eines Subsystems graphisch dargestellt.



MASCOT-1

Da die Kanäle und Speicher die einzigen Schnittstellen zwischen den asynchron ablaufenden Aktivitäten bilden, spielt die Synchronisation ihres Zugriffs und deren einfache Formulierbarkeit in dem zugrundegelegten Synchronisationsmodell eine wesentliche Rolle. MASCOT bietet dafür geeignete Sprachkonzepte an, die in der Sprache MORAL eingebettet sind. MORAL ist so entworfen worden, daß sie direkt in CORAL 66 übersetzbar ist. Im Abschnitt 'Beispiel' wird eine ausführliche Anwendung von MORAL demonstriert. Hier soll nur kurz auf das Sprachkonzept einer control-queue eingegangen werden:

Eine control-queue ist eine Art Monitor, auf dem vier Operationen definiert sind: JOIN, WAIT, LEAVE, STIM. Eine Aktivität betritt eine control-queue (JOIN), wenn sie deren Überwachung benötigt. Das Betriebssystem stellt die Ausführung dieser Aktivität zurück, bis sie an der Reihe ist (FIFO). Durch die Operation LEAVE kann eine Aktivität aus der Warteschlange gelöscht werden. Eine Aktivität in der Warteschlange kann aus Synchronisationsgründen auf eine andere Aktivität warten (WAIT) oder sie aktivieren (STIM). Das Beispiel zeigt, wie diese Sprachelemente syntaktisch aufgebaut sind und im Zusammenhang verwendet werden.

2.5.4 Entwicklungsmethode

Obwohl keine spezielle Entwicklungsmethode mit MASCOT verbunden ist, wird die Vorgehensweise doch stark durch die zur Verfügung gestellten sprachlichen Hilfsmittel und das zugrundegelegte Entwurfsmodell (Netzwerk kooperierender Prozesse) bestimmt. Eine Anleitung für die Zerlegung des Systems oder Kriterien für geeignete Schnittstellen folgen daraus jedoch nicht unmittelbar.

2.5.5 Literatur

- Jackson K.; Simpson, H.R.

MASCOT - A Modular Approach to Software
construction, operation, and test.
Technical Note 778, Royal Radar Establishment.

- Jackson, K.; Harte, H.F.

The Achievement of well-structured software in real-time applications.
IFAC/IFIP workshop on real-time programming 1976.

2.5.6 Beispiel

Ein chemischer Prozeß werde durch ein Ventil reguliert, welches seine Durchflußrate bestimmt. Ein Temperaturfühler zeigt den Zustand des Prozesses an. Durch einen Rechner soll das Ventil so gesteuert werden, daß die Temperatur in festgelegten Grenzen bleibt.

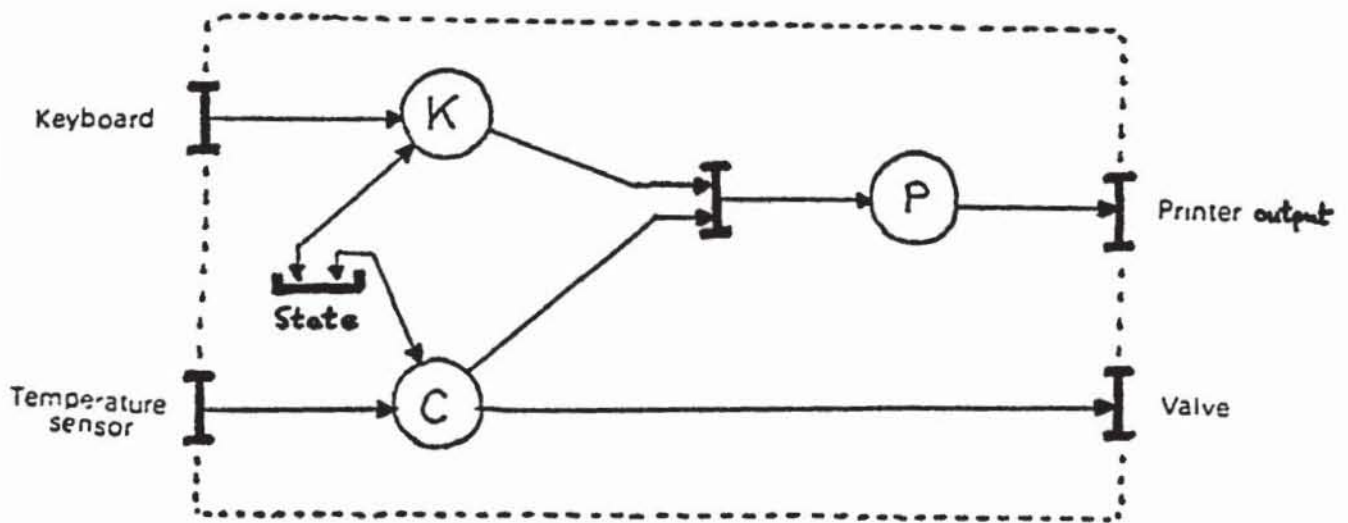
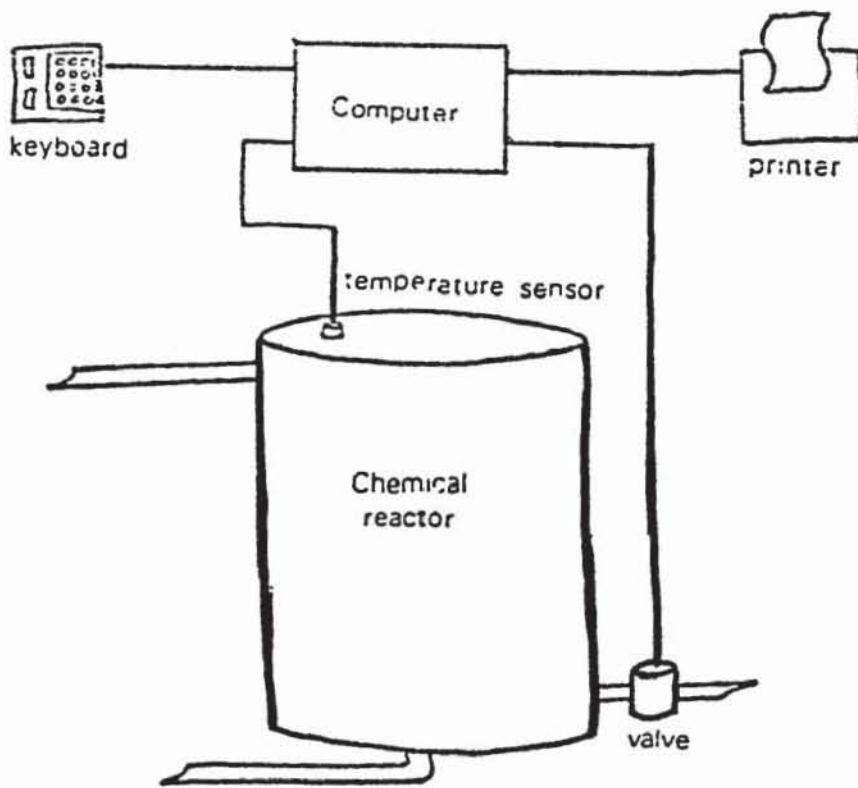
Über ein Terminal können diese Grenzwerte eingegeben werden. Auf einem Drucker wird der Zustand des Systems bei kritischen Grenzwertnäherungen und Überschreitungen der Temperatur und auf eine Abfrage hin protokolliert.

Der erste Schritt ist die Erstellung des Systemnetzwerkes (MASCOT-2). Auf dieser Abstraktionsebene wird das reale Prozeßmodell mit Hilfe der in MASCOT zur Verfügung stehenden Sprachelemente beschrieben. Dabei handelt es sich um ein statisches Modell, denn es wird noch nichts über Ablauf und Synchronisation ausgesagt. Datenstrukturen werden noch nicht konkretisiert, sondern auf Kanäle und Speicher abgebildet.

Das Diagramm MASCOT-2 stellt nur einen der möglichen Entwürfe dar. Die wesentliche Entwurfsentscheidung ist die Einführung der Aktivität P zur Ansteuerung des Druckers mit einem Kanal, in dem Druckaufträge von der Kontrollaktivität C und der Terminalaktivität K synchronisiert werden.

Der nächste Schritt ist der Entwurf der Kanäle und Speicher. Im Bild unten sind fünf Kanäle und ein Speicher enthalten (MASCOT-2).

In MORAL werden Kanäle mit dem abstrakten Datentyp 'GROUP' beschrieben durch Auflisten der verschiedenen Strukturkomponenten unterschiedlichen Typs, aus denen sich ein bestimmter abstrakter Datentyp zusammensetzt. Zusätzlich können Zugriffsprozeduren spezifiziert und Zugriffsrechte eingeschränkt werden. Dies wird am Beispiel der Spezifikation des Kanals PRINTER vom Typ PRINTCHANNEL deutlich (MASCOT-3).



MASCOT-2

```
'TYPE' 'TEMPERATURE' = 'INTEGER' (-50 'TO' 300);
'TYPE' 'VALVESETTING' = 'INTEGER' (0 'TO' 31);
'TYPE' 'PRINTREQUEST' = 'GROUP'
    'TEMPERATURE' TEMP,LOW,HIGH;
    'VALVESETTING' VALVE
'ENDGROUP';
'TYPE' 'PRINTCHANNEL'='GROUP'
'CONST' 'INTEGER' LIMIT:= 7, M:= 15;
'LOCK'
    'CONTROLQ' INQ,OUTQ;
    'INTEGER' INX,OUTX,LOSTMESSAGECOUNT;
    'ARRAY' (0 'TO' LIMIT) 'PRINTREQUEST' CYCLICQ;
'LOCK' OUTPUT
    'PROCEDURE' SEND ('REF' 'PRINTREQUEST' PR);
    'BEGIN'
        'JOIN' INQ;
        'IF'(INX = OUTX) 'MASK' M >LIMIT 'THEN'
            LOSTMESSAGECOUNT:=LOSTMESSAGECOUNT + 1
        'ELSE'
            CYCLICQ(INX'MASK'LIMIT) := [PR];
            INX := (INX + 1) 'MASK' M ;
            'STIM' OUTQ
        'FI';
        'LEAVE' INQ
    'END';
'LOCK' INPUT
    'PROCEDURE' RECEIVE ('REF' 'PRINTREQUEST' PR) ;
    'BEGIN'
        'JOIN' OUTQ ;
        'WHILE' INX = OUTX 'DO'
            'WAIT' OUTQ
        'THEN' [PR] := CYCLICQ(OUTX'MASK'LIMIT) ;
            OUTX := (OUTX+1) 'MASK' M
        'ENDLOOP' ;
        'LEAVE' OUTQ
    'END'
'ENDGROUP';
```

Die ersten Zeilen enthalten Hilfsdeklarationen, die zur Beschreibung des Typs PRINTCHANNEL benötigt werden. Die Konstanten LIMIT und M dienen zur Beschreibung des Puffers CYCLICQ, der PRINTREQUEST's enthält. Die nachfolgend deklarierten zwei MASCOT-control-queues, drei Integers und der Puffer werden durch das Sprachelement LOCK als lokale Datenstrukturen gegen Zugriffe von außen geschützt. Dem folgenden LOCK ist das Attribut OUTPUT zugeordnet. Dies bedeutet, daß nur solche Aktivitäten auf diesen geschützten Bereich zugreifen können, welche den Schlüssel Output besitzen. Der Schlüssel wird bei der Vereinbarung der root-procedure einer Aktivität festgelegt:

```
procedure keyact (ref keyboard keys (input),  
                  ref printchannel printer (output),  
                  ref statepool state (limits));
```

Ist dies die root-procedure der Aktivität K, so besitzt sie z.B. das Zugriffsrecht auf die Prozedur SEND des PRINTER-Kanals.

Aus der Deklaration des PRINTCHANNEL ist zu entnehmen, daß die Prozeduren SEND und RECEIVE den Puffer CYCLICQ als einen Ringpuffer manipulieren und mit Hilfe von MASCOT-Primitiven den Oberlauf verhindern und den Zugriff synchronisieren.

Als weitere Beispiele folgen die Deklaration des Speichers STATE als der abstrakte Datentyp STATEPOOL und die rootprocedure der Kontrollaktivität C (MASCOT-4).

```
'TYPE' 'STATEPOOL' = 'GROUP'
  'LOCK'
    'TEMPERATURE' TEMP,LOW,HIGH;
    'VALVESETTING' VALVE;
    'CONTROLQ' GUARD;
  'LOCK' LIMITS
    'PROCEDURE' RANGESET ('TEMPERATURE' L,H);
    'BEGIN' 'JOIN' GUARD;
      LOW := L; HIGH := H;
    'LEAVE' GUARD
    'END';
  'LOCK' LIMITS, VALUES
    'PROCEDURE' READ ('REF' 'PRINTREQUEST' PR,'SOURCE' S);
    'BEGIN' 'JOIN' GUARD;
      [PR] := 'EVAL' (TEMP,LOW,HIGH,VALVE);
    'LEAVE' GUARD
    'END';
  'LOCK' VALVES
    'PROCEDURE' TEMPSET (TEMPERATURE' T);
    'BEGIN'
      'JOIN' GUARD; TEMP := T; 'LEAVE' GUARD
    'END'
    'PROCEDURE' VALVESET ('VALVESETTING' V);
    'BEGIN'
      'JOIN' GUARD; VALVE := V; 'LEAVE' GUARD
    'END'
'ENDGROUP';

'PROCEDURE' CONTROL (
  'REF' 'SENSOR' THERMO (INPUT),
  'REF' 'ACTUATOR' VALVE (OUTPUT),
  'REF' 'STATEPOOL' STATE (VALUES),
  'REF' 'PRINTCHANNEL' PRINTER(OUTPUT));
'BEGIN'
  'CONST' 'INTEGER' K:= 5;
  'TEMPERATURE' TNOW, TLAST:= 0, TMEAN;
  'PRINTREQUEST' PR;
  'DO'
    TNOW:= [THERMO].READ;
    [STATE].TEMPSET(TNOW);
    [STATE].READ(PR);
    TMEAN:= (PR.LOW + PR.HIGH)/2;
    PR.VALVE:= PR.VALVE+KX' IF 'TNOW=TLAST'THEN'(TNOW-MEAN)
      'ELSE'(TNOW-MEAN)/(TNOW-TLAST)'FI';
    'IF' TNOW > PR.HIGH 'OR' TNOW < PR.LOW 'THEN'
      [PRINTER].SEND(PR)
    'FI';
    TLAST:= TNOW
  'ENDLOOP'
'END';
```

2.6. Module Interconnection Language 75 (MIL75)

2.6.1 Kurzbeschreibung

In dem bekannten Artikel von DeRemer und Kron (/1/) wurden die Module Interconnection Languages (MILs) als Sprach-Klasse eingeführt und ein spezieller Repräsentant, genannt MIL75, vorgestellt. Zugrunde liegt die Meinung, daß zur Formulierung der Zusammenhänge (Programming in the Large) andere Mittel nötig sind als zur Darstellung der Details (Programming in the Small).

MIL75 vereinigt Sprachelemente, die sonst verstreut sind über die 'Languages for programming in the Small' (LPS) und die Steuersprache für den Binder; teilweise werden die betreffenden Informationen auch nur in der nicht-formalen Dokumentation aufbewahrt. Entsprechend vereinigt der MIL75-Prozessor Funktionen des Compilers, des Binders und eines Dokumentationssystems.

Die Moduln, die im Sinne von MIL75 Atome der Beschreibung sind, werden in einem Baum angeordnet. Mit dieser Struktur sind die Beschreibungen der Zugriffsrechte auf die von den Moduln zur Verfügung gestellten 'resources' verknüpft. Als 'resources' werden offenbar die Moduln selbst bezeichnet; die Beschränkungen der Zugriffsrechte betreffen also die Berechtigung, andere Moduln aufzurufen.

2.6.2 Anwendungsphase und Gebiet

MIL75 ist ein Mittel für den top-down-Entwurf. Der MIL75-Code geht in die Implementierung ein. Modul- und Integrationstests soll das MIL75-System unterstützen.

Die Beschränkung auf Baumstrukturen bedeutet eine Entscheidung für solche Entwurfsmethoden, die auf Bäume ausgerichtet sind, z.B. nach Parnas. Auch die Verwaltung der Zugriffsrechte basiert auf dem Prinzip des Information Hiding von Parnas.

2.6.3 Inhalt und Form der Darstellung

Ein MIL75-'Programm' beschreibt eine 'module interconnection structure', die wie folgt aufgebaut ist:

T_M = (T, T_R , T_A , M, Mod, Or, Ud, UND);
darin enthalten;

T die Baumstruktur, bestehend aus der Menge der Knoten, einer Wurzel darin und der Vater-Funktion;

T_R die Zuordnung der resources, d.h. ihre Menge und die Angabe, welche resources von jedem Knoten bereitgestellt werden und werden sollen;

T_A die Zugriffsrechte, die im Baum horizontal vergeben werden (d.h. der Vater kann einem Sohn das Recht auf alle resources eines andern Sohnes verleihen);

M die Menge der Moduln, die den Blättern und eventuell auch andern Knoten durch

Mod injektiv zugeordnet sind;

Or, Ud, UND schließlich Abbildungen der resources auf die Knoten. Sie enthalten die Information, welche resources von einem Modul angeboten, benutzt und im zugehörigen Unterbaum angeboten oder benutzt und nicht dort angeboten werden.

Das MIL75-Programm ist also nicht-prozedural und hinsichtlich der Prüfung nur für die Übersetzungszeit gedacht; ein Laufzeitsystem zur dynamischen Prüfung von Zugriffen etc. ist nicht vorgesehen.

'Modul' wird hier nur im Sinne von ausführbaren Programmteilen verstanden. Das Zugriffsrecht läßt sich nicht graduell einschränken, sondern nur gewähren oder verweigern.

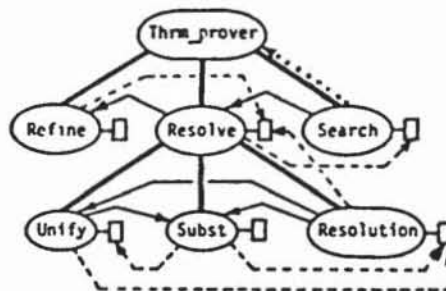
Es gelten dabei folgende Regeln:

Der Sohn hat alle Zugriffsrechte, die der Vater hat oder, die ihm vom Vater explizit eingeräumt werden.

Der Vater hat Zugriffsrecht auf alle resources, die der Sohn anbietet ('provides'). Der Sohn braucht nicht alle anzubieten, die er hat.

Der Bruder hat Zugriffsrecht auf die resources des Bruders, wenn dies vom Vater gewährt wird.

In der graphischen Darstellung in /1/ werden die Kanten des Baumes durch dicke Linien dargestellt, die Knoten durch Ellipsen. Rechtecke stellen die zugeordneten Moduln dar. Dünne Pfeile von Bruder zu Bruder drücken Zugriffsrechte aus, in Gegenrichtung und aufwärts im Baum kennzeichnen unterbrochene Pfeile die Versorgung mit einer resource. Dick punktierte Linien stehen dort, wo der Sohn seine Zugriffsrechte auch dem Vater anbietet.



2.6.4 Entwicklungsmethode

MIL75 setzt einen top-down-Entwurf voraus, impliziert aber darüber hinaus keine spezielle Entwicklungsmethode.

2.6.5 Literatur

- /1/ DeRemer, F.; Kron, H.H.
Programming-in-the-Large
versus
Programming-in-the-Small
u.a. in IEEE Transactions on Software Engineering, Vol. SE-2,
No. 2, June 1976, pp. 80-86

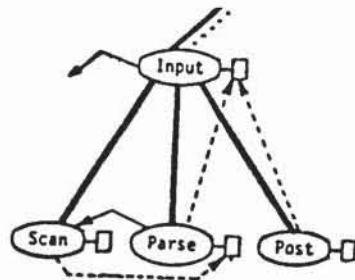
2.6.6 Beispiel

Das folgende Beispiel ist /1/ entnommen. Dort ist über den Inhalt des Programms (offenbar ein Beweis-System) nichts ausgesagt.

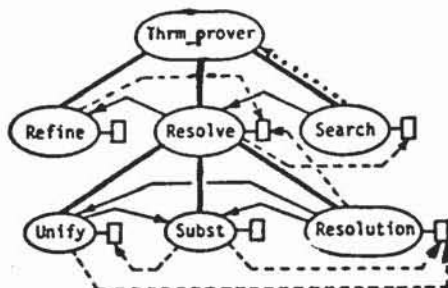
Zunächst die Beschreibung eines Unterbaums in MIL75.

```
system Input
  author 'Sharon Sickel'
  date 'July, 1974'
  provides Input_parser
  consists of
    root module
      originates Input_parser
      uses derived Parser, Post_processor
      uses nonderived Language_extensions
    subsystem Scan
      must provide Scanner
    subsystem Parse
      must provide Parser
      has access to Scan
    subsystem Post
      must provide Post_processor
```

Dieser Unterbaum läßt sich graphisch wie folgt darstellen:



Ein etwas komplexerer Ausschnitt des Systems hat folgende Darstellung:



2.7 Program Design Language (PDL)

2.7.1 Kurzbeschreibung

PDL ist der Versuch, durch Einschränkung der Syntax einer Umgangssprache ein präziseres Ausdrucksmittel zur Entwurfsbeschreibung zur Verfügung zu stellen. Es ist eine halbformale Sprache, die ihr Vokabular der englischen Sprache entnimmt und den syntaktischen Regeln einer Programmiersprache unterwirft. (Deshalb wird eine solche Sprache oft auch als strukturiertes Englisch bezeichnet, Pidgin-English.)

Für PDL existiert ein Prozessor, der die formatfreie Eingabe von PDL automatisch in ein bestimmtes Layout bringt (z.B. Einrückung und Unterstreichung von Schlüsselwörtern) und Verweise zwischen Definition und Anwendung von Entwurfssegmenten erstellt. Jeder Entwurf besteht aus einer Menge von Entwurfssegmenten in PDL, die durch eine baumartige Struktur verbunden sind. Der Hauptvorteil von PDL besteht vor allem in der rechnerunterstützten Dokumentation des Entwurfs.

PDL gibt es für die meisten Maschinen und kann für \$ 3200 bei der Firma Caine, Farber & Gordon gekauft werden.

2.7.2 Anwendungsphase und Gebiet

PDL unterstützt den Entwurf von Software-Systemen durch Bereitstellung eines Beschreibungshilfsmittels, das eng an höhere Programmiersprachen gelehnt ist.

2.7.3 Inhalt und Form der Darstellung

Eine vollständige Entwurfsbeschreibung in PDL enthält:

- ein Deckblatt mit Titel und Datum
- ein Inhaltsverzeichnis (Bild PDL-1)
- die Entwurfsbeschreibung, bestehend aus Text-, Daten- und Flußsegmenten (Bild PDL-2)

- einen Verweisbaum, der die Schachtelungsstruktur der Segmentverweise aufzeigt (Bild PDL-3)
- eine Liste, die zu jedem Segment die Seiten-/Zeilennummern angibt, an denen es benutzt wird (Bild PDL-4).

Die unterschiedlichen Segmenttypen werden durch spezielle Umrandungen gekennzeichnet. Ein Textsegment enthält ergänzende Kommentare, ein Datensegment Definitionen von Datenstrukturen.

Ein Flußsegment enthält Kontrollflußinformationen und korrespondiert etwa mit einer Prozedur in der Implementierung. Falls in einem Ausdruck eines Flußsegmentes ein Verweis auf ein anderes Flußsegment auftritt, wird die Seitenzahl, auf der dieses Segment beschrieben wird, am linken Rand dieses Ausdrucks angegeben. Die formatfrei eingegebenen Flußsegmentbeschreibungen werden durch den PDL-Prozessor in eine bestimmte, leichter lesbare Form (Einrückung) gebracht.

Der Kontrollfluß in einem Flußsegment lehnt sich in seinen Grundelementen an die Konzepte der Strukturierten Programmierung an. Die beiden elementaren Sprachelemente sind das IF und das DO-Konstrukt zur Beschreibung einer bedingten bzw. wiederholten Ausführung. Die Syntax und Anwendung dieser Sprachelemente ist aus PDL-2 zu erkennen.

2.7.4 Entwicklungsmethode

PDL unterstützt durch eine einheitliche Beschreibungsform die Methode der schrittweisen Verfeinerung. Durch die automatische Erzeugung von Verweislisten zwischen Entwurfssegmenten wird die Entwurfsprüfung erleichtert.

2.7.5 Literatur

Caine, S.H.; Gordon, E.K.

PDL - A tool for software design

National Computer Conference, 1975, Anaheim, Calif., May 19-22, Montvale, N.J.: AFIPS Pr., pp. 271-276

Verwandte Methode:

Van Leer, P.

Top-down development using a program design language

IBM Systems Journal, 15 (1976) pp. 155-170

2.7.6 Beispiel (Erläuterung unter 2.7.3)

LPG, INC. AIL DEVELOPMENT WORKBOOK (14.90)
TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION	2
PURPOSE OF SECTION	3
DICTIONARY ALGORITHMS	4
FIND DICTIONARY ENTRY	5
SEARCH ONE BLOCK	8
SEE IF MATCH	7
TOKEN SCANNING	8
BACK UP SCANNER	9
SCAN ONE TOKEN	10
SKIP BLANKS	11
SKIP COMMENT	12
SCAN IDENTIFIER	13
SCAN SPECIAL CHARACTER	14
GET NEXT CHARACTER	15
SOURCE INPUT	16
READ NEXT SOURCE CARD	17
LIST SOURCE CARD	18
MAIN PROCESSING LOOP	19
PAIR LOOP	20
PROCESS ONE STATEMENT	21
SETUP STATEMENT	22
VERIFY STATEMENT PLACEMENT	23
PROCESS IF STATEMENT	24
PROCESS PROCEDURE STATEMENT	25
PROCESS DO STATEMENT	26
PROCESS END STATEMENT	27
PROCESS END OF STATEMENT	28
DECLARATION PROCESSING	29
PROCESS DECLARATION LIST	30
SCAN DECLARATION LIST	31
SCAN DECLARATION ITEM	32
SCAN ATTRIBUTES	33
INSTALL DECLARATION ITEMS	34
INSTALL BASIC ENTRY	35
INSTALL STRUCTURE ENTRIES	36
INSTALL DECLARATION ATTRIBUTES	37
EXPRESSION AND REFERENCE PROCESSING	38
PROCESS EXPRESSION	39
PROCESS OPERAND	40
BUILD UNARY NODE	41
BUILD TOP NODE	42
PROCESS REFERENCE	43
PROCESS BASIC REFERENCE	44
FORM POSSIBLE <S4> NODE	45
PROCESS SINGLE REFERENCE	46

Inhaltsverzeichnis einer Entwurfsbeschreibung PDL-1

PROCESS EXPRESSION

```

REF
PAGE .....
*
* 1  PUSH "SCE" (START OF EXPRESSION) ONTO OPERATOR STACK
40 * 2  PROCESS OPERAND
* 3  DO WHILE NEXT TOKEN IS AN OPERATOR
* 4    DO WHILE OPERATOR IS NOT SAME AS OPERATOR ON TOP OF OPERATOR STACK AND ITS PRECEDENCE IS LESS /#
        THAN OR EQUAL TO PRECEDENCE OF OPERATOR ON THE TOP OF THE OPERATOR STACK
42 * 5    BUILD TOP NODE
* 6    POP OPERATOR STACK
* 7    ENDDO
* 8    IF NEW OPERATOR IS SAME AS TOP OPERATOR ON OPERATOR STACK
* 9      INCREMENT OPERAND COUNT IN TOP OF OPERATOR STACK BY ONE
* 10   ELSE
* 11     PUSH NEW OPERATOR AND OPERAND COUNT OF 2 ONTO OPERATOR STACK
* 12   ENDDO
40 * 13  PROCESS OPERAND
* 14  ENDDO
42 * 15  DO WHILE TOP OF OPERATOR STACK IS NOT "SCE"
* 16    BUILD TOP NODE
* 17    POP OPERATOR STACK
* 18  ENDDO
* 19  POP OPERATOR STACK
* 20  (TOP OF OPERAND STACK CONTAINS TOP NODE IN EXPRESSION)
.....

```

Ausschnitt einer 'high-level'-PDL-Beschreibung

AVERAGE OVER POINTS (RADIUS)

```

REF
PAGE .....
*
* 1  IF DEBUGGING
29 * 2    START LINE (CURRENT CYCLE)
28 * 3    PRINT POINTS IN BUFFER (CURRENT BUFFER)
* 4    ENDDO
* 5    POINTS ← 0
* 6    SX ← 0
* 7    SY ← 0
* 8    BUFFER ← PREVIOUS OF PREVIOUS BUFFER
* 9    DO FOR 5 BUFFERS
22 * 10   MOVE GOOD POINTS TO WORK BUFFER (BUFFER,RADIUS)
* 11   IF DEBUGGING
28 * 12   PRINT POINTS IN BUFFER (WORK BUFFER)
* 13   ENDDO
* 14   IF POINT COUNT OF WORK BUFFER > 0
* 15     DO FOR POINTS IN WORK BUFFER
* 16       ADD X TO SX
* 17       ADD Y TO SY
* 18     ENDDO
* 19   ADD POINT COUNT OF WORK BUFFER TO POINTS
* 20   ENDDO
* 21   BUFFER ← NEXT BUFFER
* 22   ENDDO
* 23   IF POINTS > 0
* 24     AX ← SX/POINTS
* 25     AY ← SY/POINTS
* 26   ELSE (NO DATA FOR POINT)
* 27     AX ← NEGATIVE
* 28     AY ← 0
* 29   ENDDO
.....

```

Ausschnitt einer 'low-level'-Beschreibung

CFG. INC. LOGGING DATA REDUCTION
SEGMENT REFERENCE TREES

STW

LN	DEF	SEGMENT
1	4	STW
2	11	SET DEFAULTS
3	35	FIND STARTING SECTOR
4	6	WRITE ON TAPE
5	38	CONVERT TO TANK ID
6	19	BUILD PROCESSED DATA ARRAY
7	24	INITIALIZE INPUT BUFFERS
8	31	GET POINTS
9	34	GET BATCH
10	34	READ DISK
11	32	MOVE AND COUNT POINTS
12	26	MOVE TO BUFFER
13	20	PROCESS A POINT
14	21	AVERAGE OVER POINTS
15	29	START LINE
16	28	PRINT POINTS IN BUFFER
17	22	MOVE GOOD POINTS TO WORK BUFFER
18	28	PRINT POINTS IN BUFFER
19	25	ADVANCE INPUT BUFFERS
20	31	GET POINTS
21	34	GET BATCH
22	36	READ DISK
23	32	MOVE AND COUNT POINTS
24	26	MOVE TO BUFFER
25	17	BUILD COMPRESSED DATA ARRAY
26	10	DISPLAY COMPRESSED POINTS
27	5	EXECUTE A COMMAND
28	6	WRITE ON TAPE
29	38	CONVERT TO TANK ID
30	19	BUILD PROCESSED DATA ARRAY
31	24	INITIALIZE INPUT BUFFERS
32	31	GET POINTS
33	34	GET BATCH
34	36	READ DISK
35	32	MOVE AND COUNT POINTS
36	26	MOVE TO BUFFER
37	20	PROCESS A POINT
38	21	AVERAGE OVER POINTS
39	29	START LINE
40	28	PRINT POINTS IN BUFFER
41	22	MOVE GOOD POINTS TO WORK BUFFER
42	28	PRINT POINTS IN BUFFER
43	25	ADVANCE INPUT BUFFERS
44	31	GET POINTS
45	34	GET BATCH
46	36	READ DISK
47	32	MOVE AND COUNT POINTS
48	26	MOVE TO BUFFER

PDL-3: Schachtelungsstruktur der Beschreibungssegmente

1	JP	MAIN PHASE FLUX
8	SG	MARK SUCCESSOR EDGE 7:06 7:11 7:19
21	SG	MARK LOOP ENTRY BLOCKS 3:02
39	SG	MARK LOOP MEMBERSHIP 27:12
42	SG	MARK LINE LOOP ENTRY BLOCK 21:12
4	SG	OPTIMIZE
54	SG	PERFORM JACKARD MOVEMENT 27:17
45	SG	PERFORM LOCAL CSE ELIMINATION 4:01
36	SG	PERFORM TRANSFORMATIONS 4:06
47	SG	PROCESS ASSIGNMENT FOR CSE 45:07 51:10 52:16
48	SG	PROCESS CALL FOR CSE 45:09 51:12 52:16
46	SG	PROCESS COMPUTATIONAL TRIPLE FOR CSE 45:05
17	SG	PROCESS FETCH INFORMATION 16:05 16:08 16:10 16:12 16:13 16:15 16:18
18	SG	PROCESS STORE INFORMATION 16:06
50	SG	REDUCE STRENGTH 37:18
65	SG	REDUCE STRENGTH OF ONE TRIPLE 54:07
27	SG	RESOLVE TENTATIVE BACK DIVISORS 25:09
95	GP	STRENGTH REDUCTION PART UP TRANSFORMATIONS SUBPHASE
22	SG	TRACE CALLS 4:02
23	SG	TRACE ONE NODE

PDL-4: Index für Gruppen und Segmente (Seiten-/Zeilennummern, an denen sie benutzt werden)

2.8 Problem Statement Language/Problem Statement Analyzer (PSL/PSA)

2.8.1 Kurzbeschreibung

Im Rahmen des ISDOS*-Projekts wurde an der University of Michigan unter der Leitung von Prof. Daniel Teichroew ein System entworfen und implementiert, das es gestattet, den Softwareentwurf in einer bestimmten Sprache (PSL) zu formulieren und in eine Datenbank zu speichern. Ihr Inhalt kann verändert, unter zahlreichen verschiedenen Gesichtspunkten ausgegeben und gescheckt werden mit dem Analyzer (PSA).

Ziel des ISDOS-Projekts ist es, ein Mittel zur Beschreibung und Prüfung der Programm-Spezifikation zur Verfügung zu stellen; in der Praxis hat sich aber gezeigt, daß es weniger die Spezifikation als den Entwurf unterstützt.

Der wichtigste Bestandteil des Systems ist die Datenbank. Sie enthält vom eingegebenen PSL-Text nur die relevante Information in einer nicht unmittelbar lesbaren Form. Daher ist der Zugriff nur durch PSA-Kommandos zur Report-Erzeugung möglich.

Reports enthalten über einzelne oder mehrere Namen, die frei vorgegeben oder nach verschiedenen Kriterien automatisch ausgewählt sein können, die gewünschten Angaben. Fehler wie Inkonsistenzen oder fehlende Definitionen werden darin gekennzeichnet.

Für Informationen, deren Semantik das PSL/PSA-System nicht kennt, also z.B. für den Fertigstellungstermin eines Moduls, stehen Comments und Memos zur Verfügung, deren Inhalte nicht geprüft oder verarbeitet, sondern nur gedruckt werden können.

PSL/PSA kann interaktiv (unter TSO) betrieben werden, wenn genügend Primär- und Sekundärspeicher zur Verfügung stehen. Andernfalls läuft es - wie bei uns bisher - im Batch-Betrieb.

*) Information System Design and Optimization System

2.8.2 Anwendungsphase und -gebiet

PSL/PSA läßt sich als reines Beschreibungs- und Dokumentationsmittel praktisch während der gesamten Software-Entwicklung verwenden. Seine analytischen Möglichkeiten spielen aber nur für den Entwurf eine Rolle.

Das System war zunächst für die kommerzielle Datenverarbeitung konzipiert. Inzwischen sind weitere Versionen (u.a. für die US Air-Force) geschaffen worden, die das Anwendungsgebiet erweitern. Für Echtzeit-Programmierung sind aber in PSL kaum Formulierungsmöglichkeiten und in PSA keine Prüfungen vorgesehen.

2.8.3 Inhalt und Form der Darstellung

PSL gestattet es, frei gewählte Namen bestimmter Klassen zuzuordnen und durch vorgegebene oder frei gewählte Relationen zu verknüpfen. Solche Klassen sind z.B. Process, Input, Output, Real-World-Entity oder Responsible-Problem-Definer. Relationen sind z.B. produces, uses, see-Memo. Die letzte Relation verknüpft einen Namen mit einem an anderer Stelle definierten Memo.

Eine Entwurfsbeschreibung in PSL ist ein Text mit einfacher Grammatik; seine Elemente sind Schlüsselwörter, gefolgt von der zugeordneten Information, z.B.:

```
PROCESS   :   PROBEN-ANALYSE;  
SYNONYMS  :   PROBAN, PA;  
USES      :   usw.
```

Die Ausgabe des PSA hat eine Mischform aus Text und graphischen Elementen, d.h. Namenslisten werden durch vertikale und horizontale Gliederung strukturiert, Relationen oft durch Matrizen dargestellt. Ein PICTURE-Report, der wie alle andern auf dem Drucker erzeugt werden kann, enthält Datenfluß-Graphen.

2.8.4 Entwicklungsmethode

PSL/PSA ist ein Darstellungs- und Prüfmittel; eine spezielle Entwicklungsmethode wird nicht gefördert. Allerdings wird ein top-down-Entwurf durch die Möglichkeiten des Systems unterstützt.

2.8.5 Literatur

Teichroew, D.; Hershey III E.A.

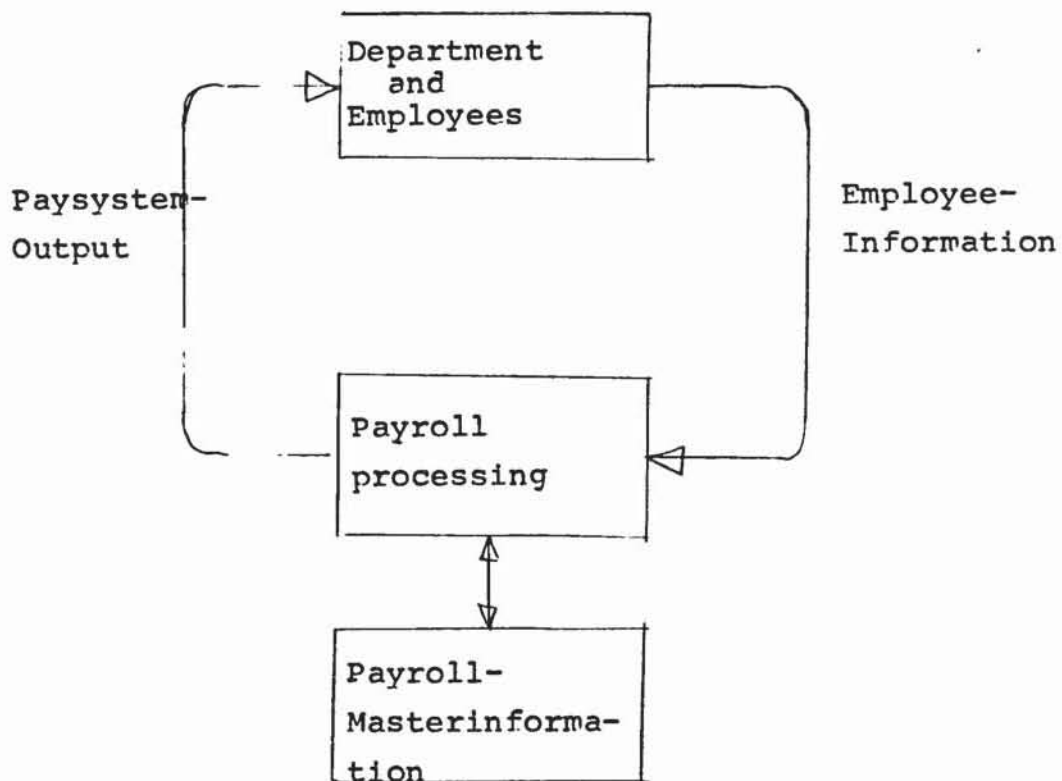
PSL/PSA: A Computer-Aided Technique for Structured Documentation
and Analysis of Information Processing Systems

IEEE Transactions on Software Engineering, Vol. SE-3, No. 1,
January 1977

2.8.6 Beispiel

Das nachfolgende Beispiel ist ein Ausschnitt des Standard-Beispiels
für PSL/PSA. Wiedergegeben ist die höchste Beschreibungsebene eines
Lohnabrechnungsprogramms.

Der Datenfluß des Gesamtsystems unter Einbeziehung der Umwelt, die
als Real-World-Entity beschrieben wird, läßt sich wie folgt darstellen:



Nachfolgend wird der PSL-Text gezeigt, der diese Struktur beschreibt
und einige zusätzliche Information gibt. (Kommentare in Form der Des-
criptions, alternativ verwendbare Synonyma.)

(PSA-1)

Der PICTURE-Report für Payroll-processing gibt die Umgebung dieses process in graphischer Form wieder (PSA-2).

Der FORMATTED-PROBLEM-Report enthält alle Informationen zu den Namen, die durch das NAME-GEN generiert worden sind (durch Parameter ALL alle Namen außer den SYNONYMS) (PSA-3).

Die einfache Struktur dieses Beispiels gestattet es nicht, die Mächtigkeit des PSL/PSA-Systems zu demonstrieren. Das vollständige Beispiel kann aber den ISDOS-Working-Papers und den Testläufen der PSL/PSA-Installation im IDT entnommen werden.

PSA VERSION 2.1R5

1977.147

BELUST IDT/GFK KARLSRUHE

A S - I S S U R C E L I S T I N G

PARAMETERS FOR: SYNU

SOURCE NOXREF UPDATE CBRFF

LINE S T M T

```
1 > /* BELUST , IDT/GFK 20.5.77 */
2 >
3 > /* DIESER PSL-TEXT ENTHAELT DIE OBERSTE BESCHREIBUNGSEBENE DES
4 > PROGRAMMS ZUR LOHNABRECHNUNG. DIE VERFEINERUNGEN DER ANGABEN
5 > SIND HIER NICHT WIEDERGEGEBENEN. */
6 >
7 > INPUT: EMPLOYEE-INFORMATION;
8 > SYNONYM: EMP-INFO, I1;
9 > DESCRIPTION;
10 > THIS INPUT REPRESENTS ALL THE NECESSARY INFORMATION TO
11 > PRODUCE THE OUTPUTS FROM THE PAYSYSTEM. ;
12 >
13 > OUTPUT: PAYSYSTEM-OUTPUTS;
14 > SYNONYM: PAYOUTS, O1;
15 > DESCRIPTION;
16 > THIS OUTPUT REPRESENTS ALL THE REQUIRED OUTPUTS OF THE
17 > TARGET PAYSYSTEM AS DEFINED BY POLICY. ;
18 >
19 > SET: PAYROLL-MASTER-INFORMATION;
20 > SYNONYM: PAY-MAST, MASTER-FILE, S1;
21 > DESCRIPTION;
22 > THIS SET CONTAINS ONE UNIT OF INFORMATION
23 > FOR EACH EMPLOYEE ON THE PAYROLL, THAT IS,
24 > THOSE EMPLOYEES WHO ARE TO RECEIVE PAYCHECKS. ;
25 >
26 > REAL-WORLD-ENTITY: DEPARTMENTS-AND-EMPLOYEES;
27 > SYNONYM: DEPT-EMP, R1;
28 > DESCRIPTION;
29 > THIS IS THE ENTITY WHICH WILL RECEIVE ALL THE OUTPUTS AND
30 > SUPPLY ALL THE INPUTS. ;
31 > GENERATES: EMPLOYEE-INFORMATION;
32 > RECEIVES: PAYSYSTEM-OUTPUTS;
33 >
34 > PROCESS: PAYROLL-PROCESSING;
35 > SYNONYM: PAYPROC, P1;
36 > DESCRIPTION;
37 > THIS PROCESS REPRESENTS THE HIGHEST LEVEL PROCESS
38 > IN THE TARGET SYSTEM. IT ACCEPTS AND PROCESSES
39 > ALL INPUTS AND PRODUCES ALL OUTPUTS. ;
40 > RECEIVES: EMPLOYEE-INFORMATION;
41 > GENERATES: PAYSYSTEM-OUTPUTS;
42 > UPDATES: PAYROLL-MASTER-INFORMATION;
43 >
44 > EOF
```

PROCESS PICTURE

PAYROLL-PROCESSING

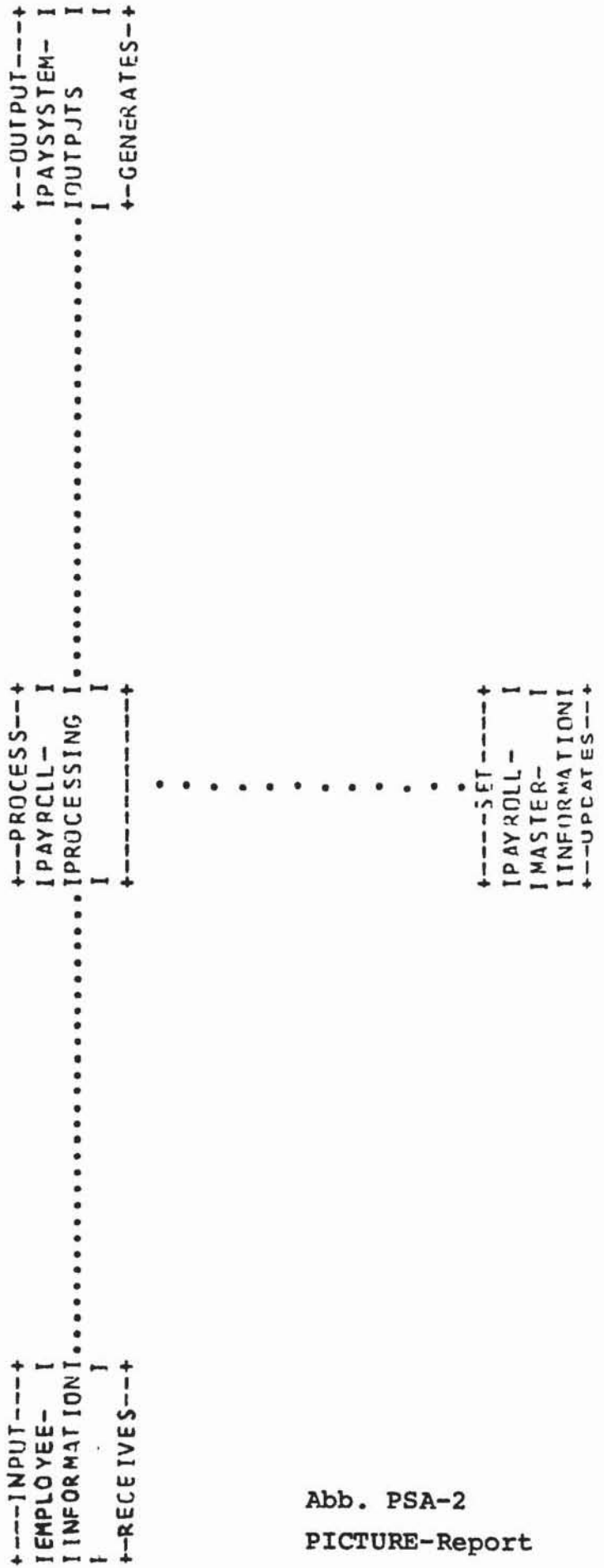


Abb. PSA-2
PICTURE-Report

FORMATTED PROBLEM STATEMENT

PARAMETERS FOR: FPS

FILE NOINDEX PRINT NOPUNCH SMARG=5 NMARG=20 AMARG=10 BMARG=25 RNMARG=70
ONE-PER-LINE DEFINE COMMENT NONNEW-PAGE NONNEW-LINE

```

1 INTERFACE                                DEPARTMENTS-AND-EMPLOYEES;
2     SYNONYMS ARE:  DEPT-EMP,
3                   RI;
4     DESCRIPTION;
5         THIS IS THE ENTITY WHICH WILL RECEIVE ALL THE OUTPUTS AND
6         SUPPLY ALL THE INPUTS.;
7     GENERATES:    EMPLOYEE-INFORMATION;
8     RECEIVES:     PAYSYSTEM-OUTPUTS;
9
10 INPUT                                       EMPLOYEE-INFORMATION;
11     SYNONYMS ARE:  EMP-INFO,
12                   II;
13     DESCRIPTION;
14         THIS INPUT REPRESENTS ALL THE NECESSARY INFORMATION TO
15         PRODUCE THE OUTPUTS FROM THE PAYSYSTEM.;
16     GENERATED BY: DEPARTMENTS-AND-EMPLOYEES;
17     RECEIVED BY:  PAYROLL-PROCESSING;
18
19 SET                                         PAYROLL-MASTER-INFORMATION;
20     SYNONYMS ARE:  MASTER-FILE,
21                   PAY-MAST,
22                   SI;
23     DESCRIPTION;
24         THIS SET CONTAINS ONE UNIT OF INFORMATION
25         FOR EACH EMPLOYEE ON THE PAYROLL, THAT IS,
26         THOSE EMPLOYEES WHO ARE TO RECEIVE PAYCHECKS.;
27     UPDATED BY:   PAYROLL-PROCESSING;
28
29 PROCESS                                     PAYROLL-PROCESSING;
30     SYNONYMS ARE:  PAYPROC,
31                   P1;
32     DESCRIPTION;
33         THIS PROCESS REPRESENTS THE HIGHEST LEVEL PROCESS
34         IN THE TARGET SYSTEM. IT ACCEPTS AND PROCESSES
35         ALL INPUTS AND PRODUCES ALL OUTPUTS.;
36     RECEIVES:     EMPLOYEE-INFORMATION;
37     GENERATES:    PAYSYSTEM-OUTPUTS;
38     UPDATES:      PAYROLL-MASTER-INFORMATION;
39
40 OUTPUT                                       PAYSYSTEM-OUTPUTS;
41     SYNONYMS ARE:  O1,
42                   PAYOUTS;
43     DESCRIPTION;
44         THIS OUTPUT REPRESENTS ALL THE REQUIRED OUTPUTS OF THE
45         TARGET PAYSYSTEM AS DEFINED BY POLICY.;
46     GENERATED BY: PAYROLL-PROCESSING;
47     RECEIVED BY:  DEPARTMENTS-AND-EMPLOYEES;
48
49 EOF EOF EOF EOF EOF

```

2.9 Structured Analysis and Design Technique (SADT)

2.9.1 Kurzbeschreibung

SADT ist eine Methode zur Unterstützung der funktionalen Analyse und des Entwurfs von Software-Systemen. Sie wurde seit 1970 entwickelt und wird von SofTech vertrieben.

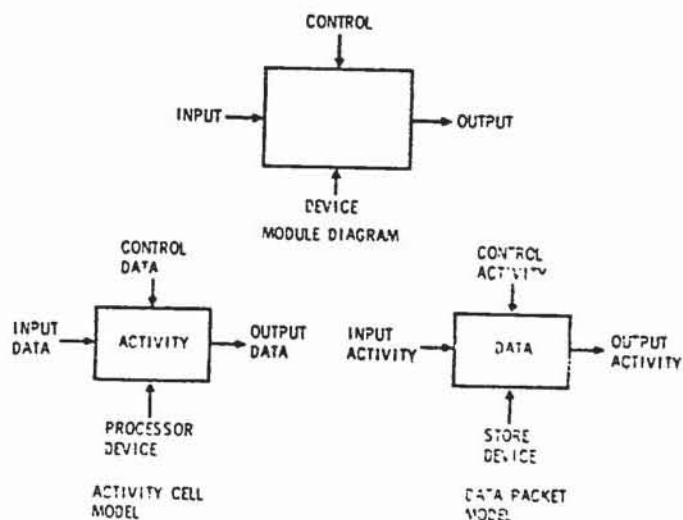
Das Ergebnis der funktionalen Analyse (Anforderungen an das System) wird in Form von Graphen, den SADT-Diagrammen, dargestellt. Das vollständige Modell einer Problembeschreibung bzw. des durch hierarchische Zerlegung abgeleiteten Entwurfs besteht aus Diagrammen, in denen die Knoten Funktionen und die Kanten Daten repräsentieren, und aus Diagrammen, in denen Knoten Daten und Kanten Funktionen repräsentieren. Zwischen diesen beiden Darstellungen bestehen Querverweise. Da die Knoten eines Diagramms wiederum durch Diagramme auf einer tieferen Ebene verfeinert sein können, wird die logische Beziehung jedes Teilsystems zum Gesamtsystem explizit ausdrückbar.

2.9.2 Anwendungsphase und Gebiet

SADT ist eine Technik zur Unterstützung der Systemanalyse und des Entwurfs und auf kein spezielles Anwendungsgebiet ausgerichtet. Neben Hilfsmitteln zur Informationsdarstellung besteht die Technik vor allem in einem Katalog von Richtlinien und Vorgehensweisen zur Anwendung dieser Hilfsmittel.

2.9.3 Inhalt und Form der Darstellung

Die grundlegenden Beschreibungseinheiten in SADT sind die actigrams (activity diagrams) und die datagrams (data diagrams). Sie werden graphisch in einer einheitlichen Form dargestellt:



Das Beschreibungsmodell von SADT basiert auf der schrittweisen Zerlegung eines Systems. Durch die actigrams und datagrams werden die beiden Hauptaspekte dieser Zerlegung, der funktionelle und der datenorientierte, explizit dargestellt.

In den actigrams werden die bei der funktionellen Zerlegung erzeugten Funktionen durch Funktionskästen und die Objekte (Daten), die transformiert werden, durch Datenpfeile repräsentiert.

In den datagrams werden die bei der datenorientierten Zerlegung definierten Komponenten von Datenstrukturen als Datenkästen auf verschiedenen Beschreibungsebenen und die Funktionen, die die einzelnen Komponenten verarbeiten, als Funktionspfeile dargestellt.

Im Bild SADT-1 ist zu sehen, daß Pfeile unterschiedliche Bedeutung haben, die dadurch markiert ist, daß sie waagrecht oder senkrecht aus dem Kasten heraus oder in ihn hinein laufen.

In den unten angegebenen Beispielen fehlt die Schnittstellenart 'DEVICE', da sie Angaben über Realisierungsmöglichkeiten der entsprechenden Informationseinheit enthält und somit meist erst in der detaillierteren Entwurfsphase benötigt wird.

Actigrams

Bild SADT-2 gibt ein Beispiel für Actigrams.

Das Diagramm mit der Knotenmarkierung A2 ist eine Verfeinerung des Knotens A0.

Eingabedaten (in die linke Seite eines Kastens hineinführende Kanten) werden in Ausgabedaten (an der rechten Seite hinausführende Kanten) transformiert. Diese Transformation kann abhängig sein von Kontrollinformation (in die obere Seite eines Kastens hineinführende Kanten). Diese Kontrollinformation kann als Zustandsinformation interpretiert werden. Ein Kasten in einem Actigram repräsentiert also eine Klasse von Funktionen, die je nach Eingabe- und Zustandsinformation verschiedene Ausgaben erzeugt.

Eine Funktion ist ausführbar, sobald alle hineinführenden Kanten belegt sind (logische UND-Verknüpfung). Daraus folgt, daß die Kanten in einem Actigram keine sequentiellen Kontrollflüsse darstellen, sondern mehrere Kästen (Funktionen) parallel ausführbar sein können. Durch Verzweigungen von Kanten mit entsprechender Markierung werden Komponenten von Datenstrukturen dargestellt.

Datagrams

Bild SADT-3 gibt ein Beispiel für Datagrams. Das Diagramm mit der Knotenmarkierung D3 ist eine Verfeinerung des Knotens D0. Datagrams sind in ihrer Form den Actigrams ähnlich. Der wesentliche Unterschied liegt in der Interpretation der Kästen und Kanten. Eingabefunktionen (in die linke Seite eines Kastens hineinführende Kanten) erzeugen Teile der durch den entsprechenden Kasten dargestellten Objekte (Daten), während Ausgabefunktionen (an der rechten Seite hinausführende Kanten) Teile davon konsumieren. Kontrollfunktionen überwachen das Erzeugen oder Konsumieren der Objekte. Kantenverzweigungen markieren Teilfunktionen.

Zwischen Datagrams und Actigrams besteht keine ein-zu-eins Abbildung in dem Sinne, daß ein Kasten in einem Datagramm durch eine Kante in einem Actigramm dargestellt wird und umgekehrt.

Eine vollständige SADT-Beschreibung enthält neben den einzelnen Diagrammen noch einen sogenannten Knotenindex (siehe SADT-4).

2.9.4 Entwicklungsmethode

Die SADT-Methode kann als top-down-Entwicklung angesehen werden. Neben der funktionellen Zerlegung und Darstellung wird die datenorientierte Zerlegung gesondert dargestellt. Dies ermöglicht eine unabhängige Überprüfung der Beschreibung. Obwohl SADT zur Unterstützung der Spezifikations- und Entwurfsphase gedacht ist, bietet es keine speziellen Hilfsmittel für die Spezifikation. Die Strukturierung des Entwurfs wird erzwungen durch die Randbedingungen der graphischen Darstellung.

2.9.5 Literatur

Ross, D.T.; Schoman, K.E.

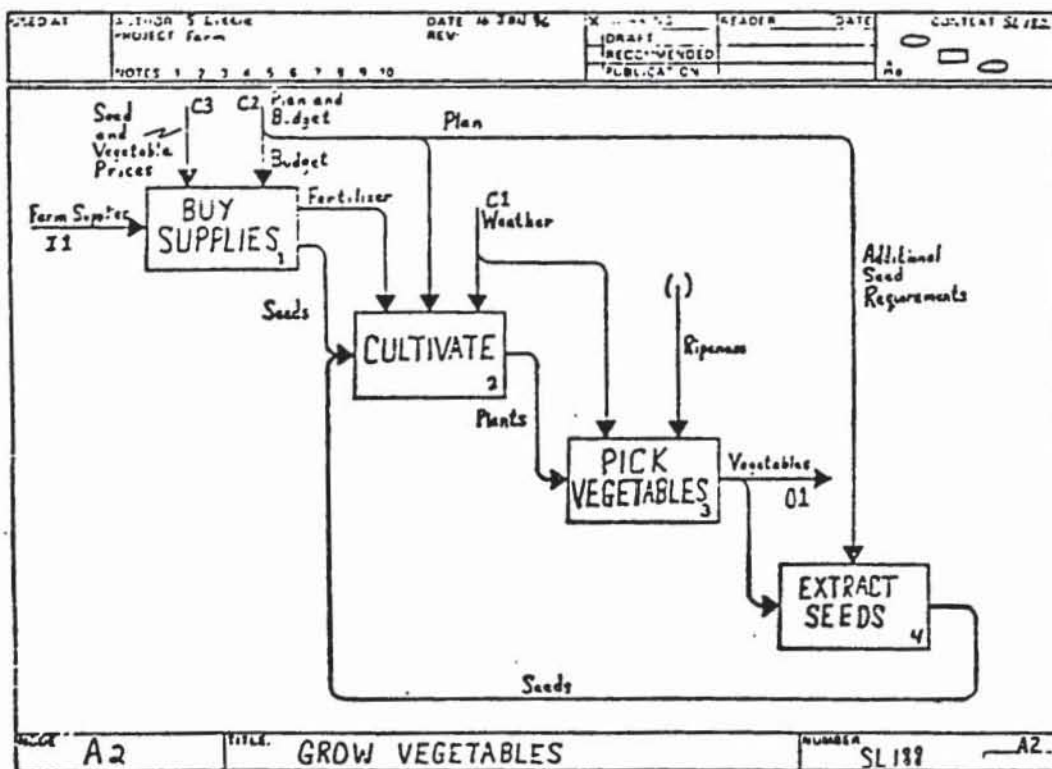
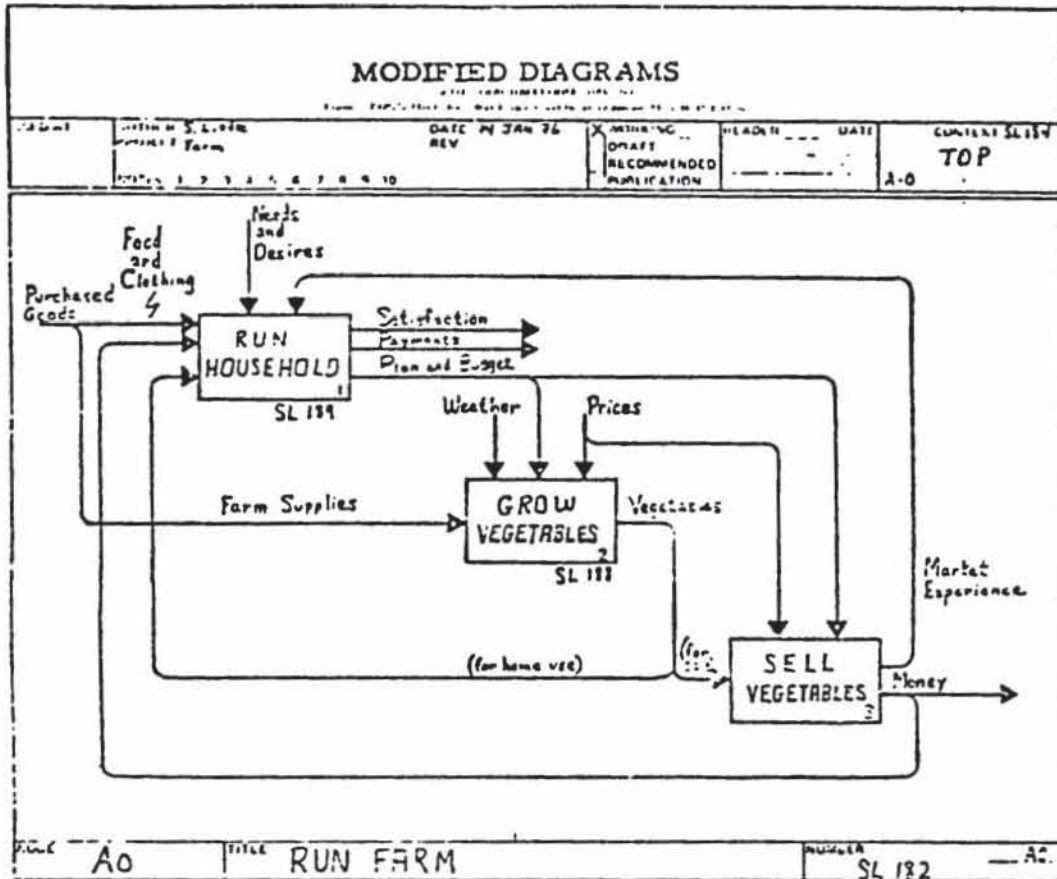
Structured Analysis for Requirements Definition

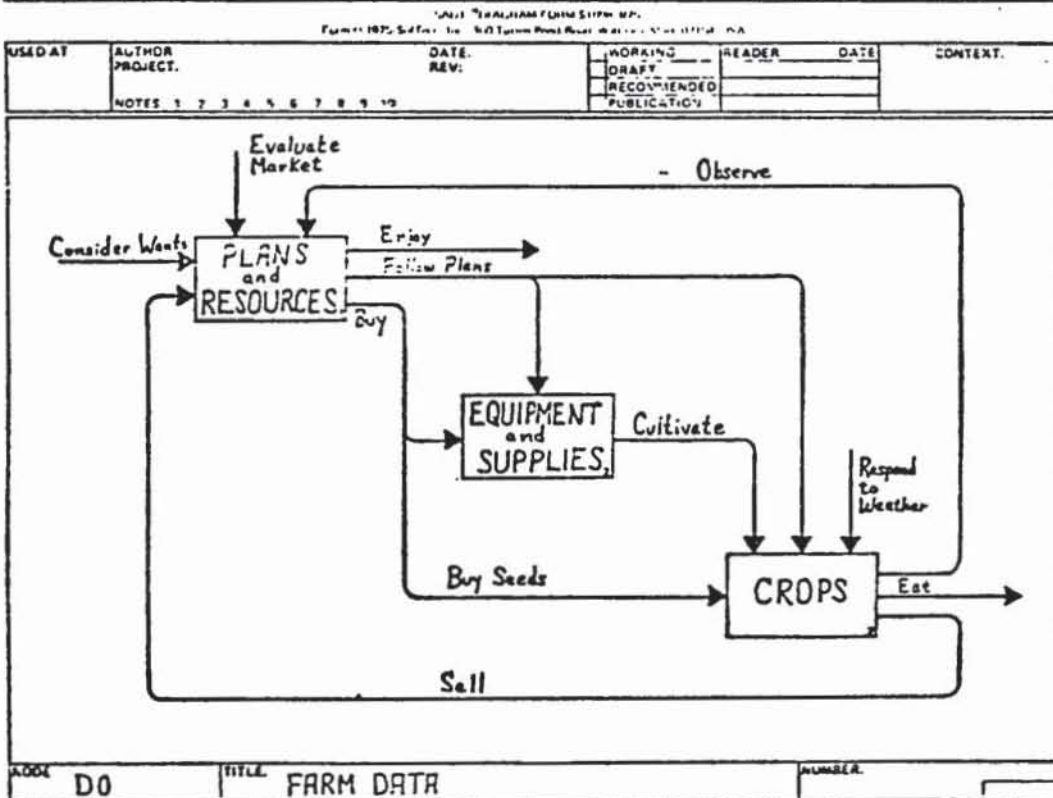
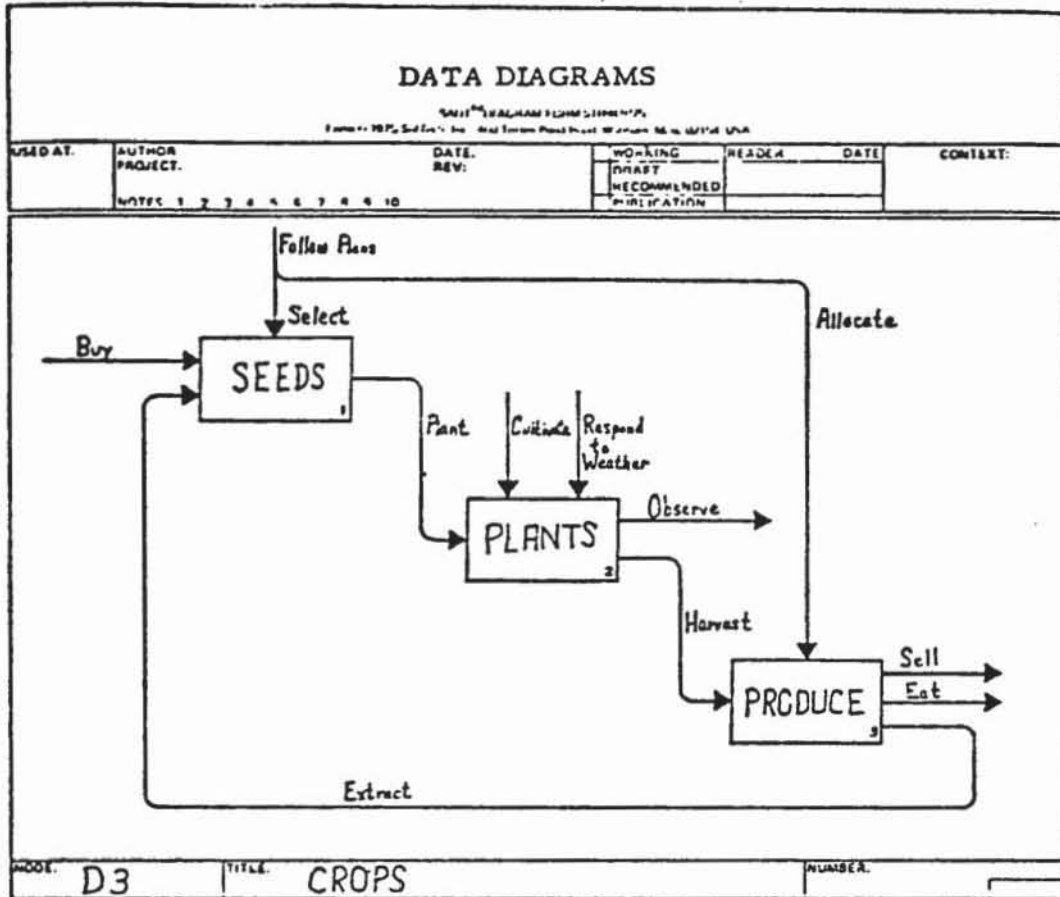
Proc. IEEE/ACM 2nd Int. Conference on Software Engineering,
San Francisco, Oct. 1976.

SofTech, Inc.

An Introduction to SADT: Structured Analysis and Design Technique, 1976

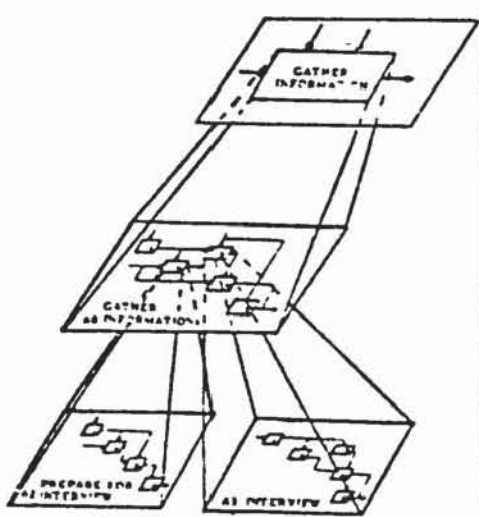
2.9.6 Beispiel: Die Actigrams und Datagrams beschreiben einen Ausschnitt aus dem Betrieb eines Bauernhofes





SADT-3 SADT DATA DIAGRAM

NODE INDEX AND CORRESPONDING
DECOMPOSITION STRUCTURE OF DIAGRAMS

<u>Node Index</u>	<u>Corresponding Decomposition Structure</u>
A -0 Gather Information (Context)	
A0 Gather Information	
A1 Coordinate and Monitor	
A2 Prepare for Interview	
A21 Confirm Data	
A22 Determine Eligibility	
A23 Prepare Notifications	
A24 Notify Unit and Member	
A3 Interview	
A31 Confirm Preparedness	
A32 Instruct Member re Options	
A33 Gather Member Choices and Data	
A34 Prepare Forms and Instructions	
A35 Instruct Member	
A4 Prepare Items	
A5 Perform Final Actions	

2.10. Structured Design (SD)

2.10.1 Kurzbeschreibung

SD wurde von Constantine im Laufe von etwa zehn Jahren entwickelt (1965-1974). Ähnlich wie bei der Jackson Design Methodology handelt es sich hier weniger um ein System als um eine Methode. Es gibt also keine Rechnerunterstützung, die Darstellungsmittel (structure charts) sind von untergeordneter Bedeutung.

SD besteht aus zwei Komponenten:

- a) Ein Bewertungsschema erlaubt die quantitative Beurteilung einer Modularisierung. Als Kriterium wird dabei die interne Bindung der Komponenten jedes Moduls herangezogen. Die interne Bindung sollte möglichst hoch sein innerhalb der Skala zufällig - logisch - zeitlich - datenbezogen - verkettet - funktionell (diese Begriffe sind einigermaßen genau definiert). Die Kopplung soll dagegen schwach sein, d.h. möglichst einfache, klare Schnittstellen (keine common-Variablen), keine Zugriffe über Modulgrenzen, keine Beeinflussung des Kontrollflusses übergeordneter Moduln oder gar Codeveränderungen.
- b) Für den Entwurf werden Hinweise und ein Darstellungsschema gegeben (siehe unten).

Ziel des SD ist es, die Erstellung wohlstrukturierter Programme mit allen ihren Vorteilen zu unterstützen und den Erfolg dabei meßbar zu machen, damit nicht wie üblich andere, leichter meßbare Größen wie Umfang oder Laufzeit den Ausschlag geben.

2.10.2 Anwendungsphase und Gebiet

SD ist eine Methode für den Programmentwurf. Sie kommt vor allem dann in Frage, wenn die Zerlegung in nur schwach gekoppelte Moduln vorrangig ist, also z.B. wenn die Arbeit auf sehr viele Programmierer verteilt werden soll. Die Methode bietet keine speziellen Regeln oder Hilfen für die Behandlung von Echtzeit-Problemen.

2.10.3 Inhalt und Form der Darstellung

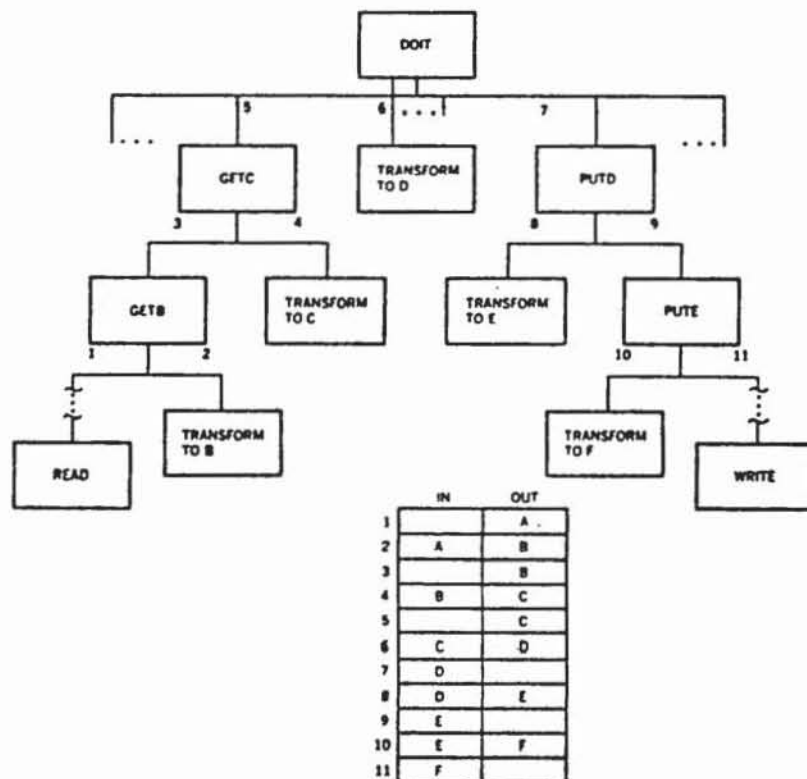
SD ist kein automatisches Verfahren. Daher gibt es keine strenge Form. Im Zuge der Entwicklung werden Datenflüsse und Modul-Hierarchien betrachtet. Als Darstellungsmittel werden Structure-charts verwendet. Dabei handelt es sich um Graphen, die die Aufrufstruktur der Moduln zeigen. Zu jedem Modul gehört eine Zeile in einer Parametertabelle (siehe Abb. SD-1).

2.10. 4 Entwicklungsmethode

SD geht von einem Datenfluß-Graphen aus. In diesem wird der Hauptstrom von der Ein- zur Ausgabe festgestellt. Im folgenden Schritt wird der Hauptstrom in die Abschnitte Eingabe, Verarbeitung und Ausgabe zerlegt. Als Kriterium gilt dabei, daß die Daten an den beiden Schnittstellen in der Form vorliegen, die dem abstrakten Verständnis von Ein- und Ausgabe am nächsten kommen. Zum Beispiel würde bei einer Meßdatenauswertung der Schnitt hinter die Plausibilitätsprüfung und Filterung gelegt, so daß nur die repräsentativen Daten die Verarbeitung erreichen. Am Ende würde die Schnittstelle vor die Aufbereitung der Werte zur Ausgabe gelegt. Diese Aufteilung kann jetzt rekursiv wiederholt werden. Auf der Eingabeseite werden wiederholt Prozesse von der Eingabe abgetrennt, auf der Ausgabeseite umgekehrt.

Es entsteht eine Modulstruktur mit der für SD charakteristischen Treppenform bei Ein- und Ausgabe. Die Verarbeitung selbst ist bei dieser Struktur oft trivial.

Abb. SD-1 (aus /1/)



Zur Kontrolle der Modularisierung werden die Skalen der Bindung und Kopplung verwendet. Geringe Bindung oder hohe Kopplung weisen auf eine schlechte Aufteilung hin.

Zu den weiteren Regeln gehört die Anpassung der Lösungs-Struktur an die Problem-Struktur. Der Einflußbereich eines Moduls (scope of effect) sollte innerhalb seines Kontrollbereichs (scope of control) liegen, d.h. er sollte nur solche Moduln beeinflussen, die in dem Unterbaum liegen, dessen Wurzel er ist.

Ähnlich wie bei der JDM ist bei SD wichtig, daß der Anwender ein System von Regeln erhält. Offenbar unterscheiden sich SD und JDM im Schwerpunkt dieser Regeln: hier handelt es sich vorwiegend um Kriterien, die die (rückblickende) Prüfung erlauben, dort um Anweisungen, die das Vorgehen festlegen.

2.10.5 Literatur

/1/ Stevens, W.P.; Myers, G.J.; Constantine, L.L.
Structured Design
IBM System Journal 13, 1, pp. 114-139 (1974)

/2/ Yourdon, E.; Constantine, L.L.
Structured Design
Yourdon Inc., New York 1975

/3/ McGowan, C.L.; Kelly, J.R.
A Review of Decomposition and Design Methodologies
in:INFOTECH, Reliable Software, London 1977

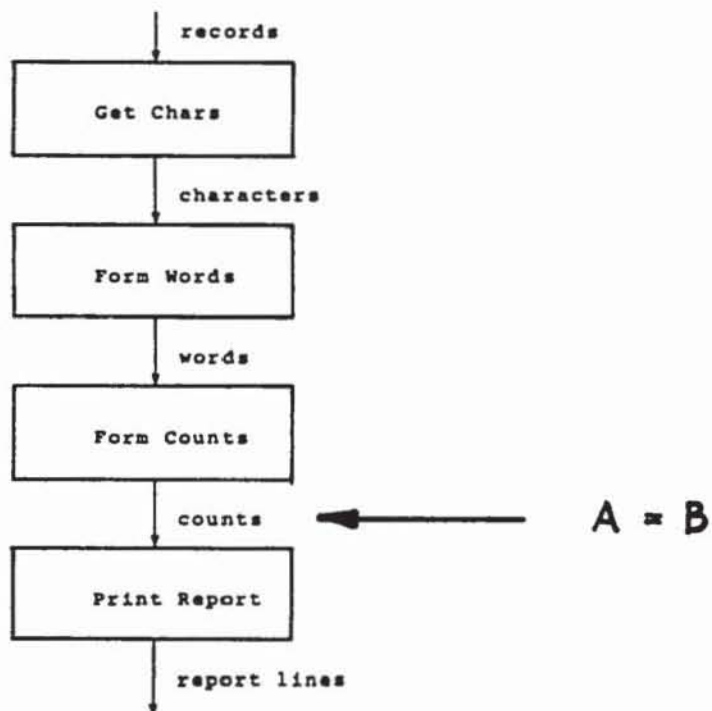
2.10.6 Beispiel

Als Beispiel soll die Lösung des von Jackson angegebenen Problems gezeigt werden (nach /3/). Ein File mit den Texten von Telegrammen (null bis n), die mit dem Wort ZZZZ abgeschlossen sind, soll verarbeitet werden. Als Ergebnis soll ein Protokoll gedruckt werden, das für jedes Telegramm eine laufende Nummer, die Zahl der Wörter und die Zahl der überlangen Wörter enthält.

Die Aufteilung des Eingabefiles in Records hat nichts mit den Grenzen der Telegramme zu tun.

Der Datenfluß kann wie folgt beschrieben werden:

SD-2

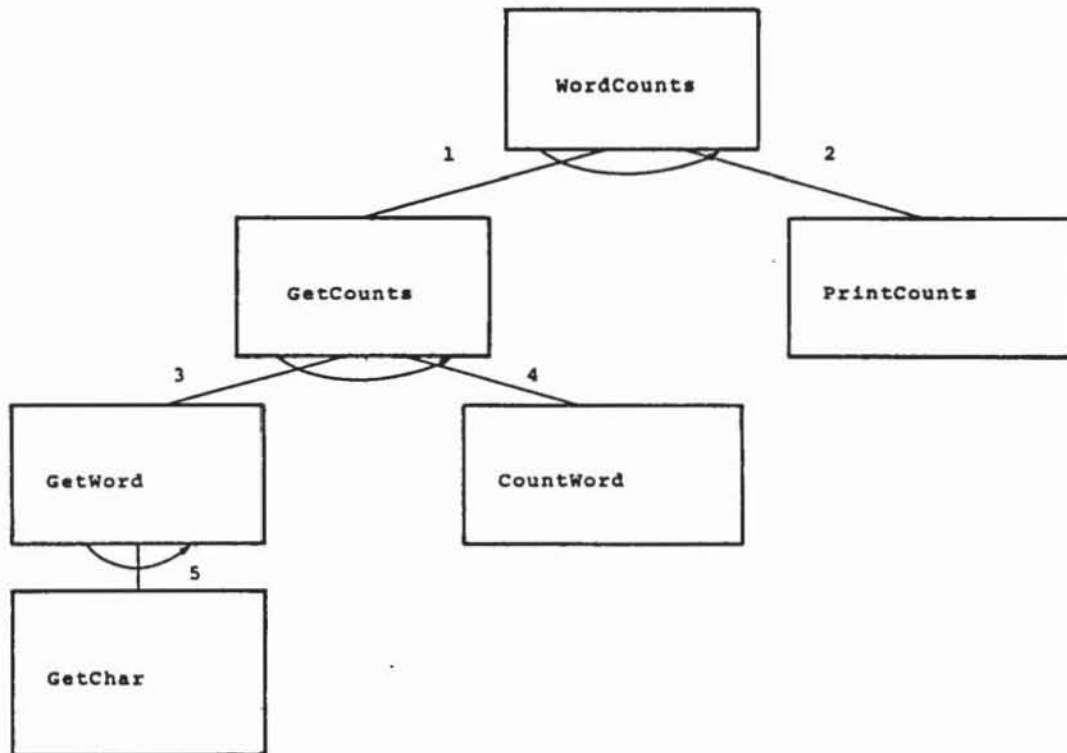


*Structured design data
flow chart*

Die Punkte A und B werden als Grenzen zwischen Eingabe, Verarbeitung und Ausgabe gewählt. In diesem Beispiel fallen A und B zusammen, es gibt also im Sinne des SD keine Verarbeitung.

Die Fortsetzung dieser Zerlegung im Eingabeteil ergibt folgenden Entwurf:

SD-3



	Input	In/Out	Output
1	-----	-----	#Words, #Oversize, EOF
2	Telegram#, #Words, #Oversize	-----	-----
3	-----	-----	Word, EOF
4	Word	#Words, #Oversize	-----
5	-----	-----	Char, EOF

Die Spalte In/Out enthält transiente Parameter. /3/ enthält ein PL/1 ähnliches Programm, das diesen Entwurf realisiert.

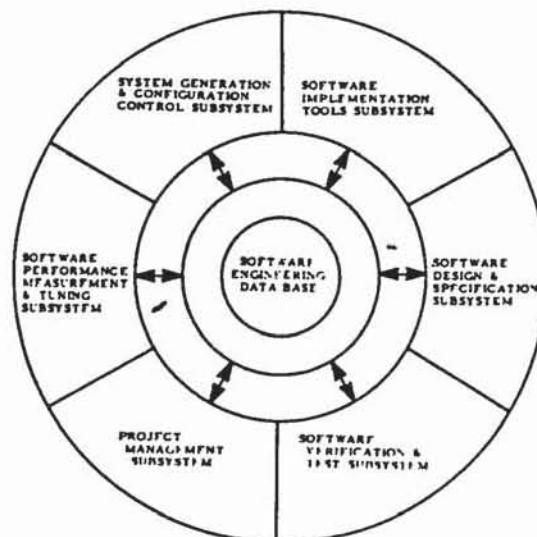
2.11. Software Engineering Facility (SEF)

2.11.1 Kurzbeschreibung

Die Software Engineering Facility (SEF) wird im Auftrag der amerikanischen Luftwaffe seit etwa 1974 von SOFTECH in Waltham, Massachusetts, entwickelt. Nach der letzten Veröffentlichung im Oktober 1976 scheint sich dieses Projekt noch im Entwurfsstadium zu befinden; es werden nur vage Angaben zur Architektur gemacht, von Anwendungserfahrungen wird noch nicht berichtet.

Ziel der SEF ist es, eine umfassende, integrierte Sammlung von Hilfsmitteln für alle Phasen der Programmentwicklung zur Verfügung zu stellen, also für Spezifikation, Entwurf, Test, Integration und die begleitende Dokumentation. Kern dieses Systems ist die Datenbank, die alle Information über die Software enthält, die in der Entwicklung ist. Damit gehört SEF zur Klasse der Datenbankorientierten Software-Entwicklungssysteme, in der auch PSL/PSA und SREP sind.

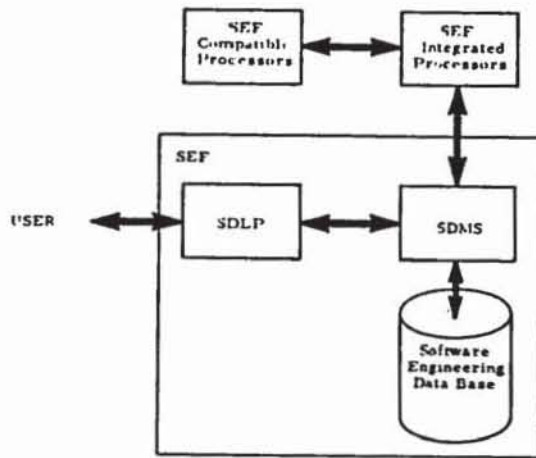
SEF-1



The Software Engineering Facility:
Subsystem View

Die Ziele der SEF sind sehr hochgesteckt. Die Realisierung soll aber dadurch erleichtert werden, daß vorhandene Werkzeuge, z.B. Übersetzer integriert werden, indem man durch spezielle Anpassungsbausteine (Interfaces) den Anschluß an die Datenbank ermöglicht.

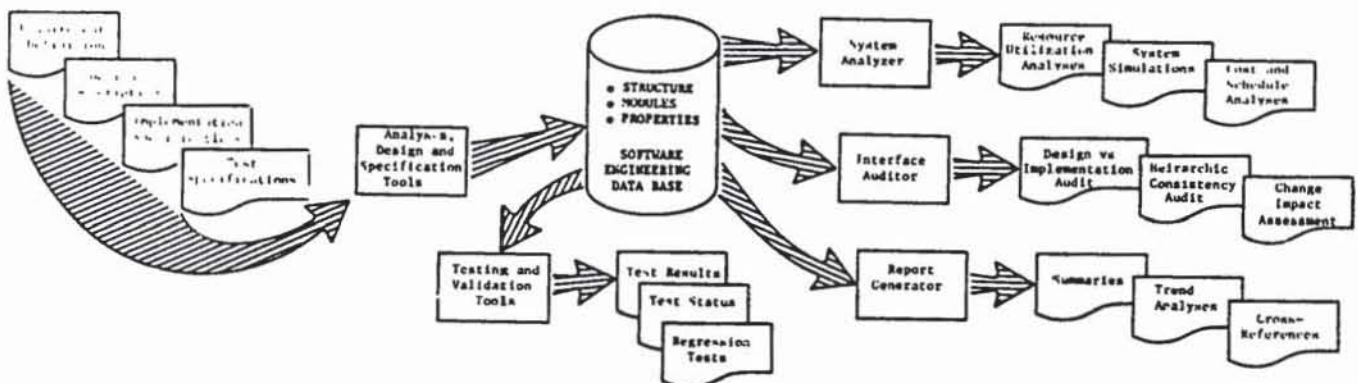
SEF-2



Processor Support View of SEF Architecture

Die Arbeitsweise von SEF ist in der folgenden Abbildung skizziert (alle aus /1/).

SEF-3



SEF: Processor View

2.11.2 Anwendungsphase und Gebiet

Entsprechend den Zielen von SEF soll dieses System während des gesamten Entwicklungsganges eingesetzt werden. Eine Ausrichtung auf eine spezielle Art von Software ist nicht erkennbar.

2.11.3 Inhalt und Form der Darstellung

Ober die Form der Darstellung gibt es keine Angaben in /1/. Gegenstand der Eingabe - und damit auch der Datenbank - sind alle in der Programmentwicklung anfallenden Informationen, also z.B. Spezifikation, Strukturbeschreibungen, Testdaten, Managementinformationen.

2.11.4 Entwicklungsmethode

In /1/ wird die Aussage gemacht, daß die Benutzung von SEF zur Einhaltung einer bestimmten Methode zwingt und dadurch die Erzeugung guter Software ('well-engineered programs') fördert. Nähere Angaben fehlen.

2.11.5 Literatur

/1/ Irvine, C.A.; Bracket, J.W.

Automated Software Engineering through Structured Data Management
Proc. IEEE/ACM 2nd Int. Conference on Software Engineering,
San Francisco, October 1976, pp. 56-61

2.11.6 Beispiel

Ein Beispiel kann wegen des Fehlens aller Angaben zur Repräsentation nicht gegeben werden.

2.12. Software Requirements Engineering Program (SREP)

2.12.1 Kurzbeschreibung

Das SREP-System wurde als Teil des Software Development Systems (SDS) von TRW für das Ballistic Missile Defense Advanced Technology Center (BMDATC) der U.S. Army entwickelt. Das System basiert praktisch und konzeptionell auf PSL/PSA und verwendet z.B. dessen Datenbankmanagement-System.

SREP besteht im wesentlichen aus einer Spezifikationsprache, der Requirements Statement Language (RSL), einem Analysator, dem Requirements Evaluation and Validation System (REVS) und einem Datenbankmanagement-System. Das Sprachkonzept von RSL umfaßt Sprachkonstrukte, die zur Beschreibung von Realzeitanwendungen geeignet sind. Zur Darstellung von Realzeit-Leistungsanforderungen können die funktionalen Anforderungen in Reaktionsnetzen (stimulus-response networks), den sogenannten R-nets, angeordnet werden. Der Analysator besteht aus einem System von Hilfsmitteln zur Prüfung der Systembeschreibung in der Datenbank.

Das Ziel von SREP ist die Erstellung einer vollständigen, konsistenten, geprüften, eindeutigen und maschinenverarbeitbaren Spezifikation der Software-Anforderungen und damit eines Dokuments, das als Grundlage für den Entwurf dienen kann.

2.12.2 Anwendungsphase und Gebiet

Neben grundlegenden Funktionen zur Unterstützung der Software-Entwicklung wurden gegenüber PSL/PSA besondere Erweiterungen für die Anwender bei Realzeitsoftware-Projekten vorgesehen. Außer an Sprachkonstrukten von RSL zeigt sich dies vor allem bei den spezifischen Analyseunterstützungen. Einen Schwerpunkt bildet hier die Simulation.

2.12.3 Inhalt und Form der Darstellung

Zur Beschreibung der Anforderungen steht die formale Sprache RSL zur Verfügung. Anstatt die Anforderungen einzelner Modulen zu spezifizieren, wird eine Anforderung als die Prozeßsequenz beschrieben, die eine bestimmte Eingabe (stimulus) zur Folge haben soll. Diese ablauforientierte Beschreibung wird durch einen formalen Strukturierungsmechanismus, die sogenannten R-Nets, dargestellt.

RSL baut auf vier Grundkonzepte auf:

- Elemente

Die Elemente sind die Objekte der Sprache. Jedes Element ist von einem bestimmten Typ, wobei jeder Typ etwas über die charakteristische Eigenschaft eines Objektes aussagt. Standard-Element-Typen in RSL sind z.B. ALPHA (die Klasse der Prozesse), DATA (die Klasse der zu verarbeitenden Informationselemente) und R-NET (die Klasse der Verarbeitungsabläufe).

- Relationen

Relationen definieren Beziehungen zwischen Elementen. Die RSL-Relationen sind binär und nicht kommutativ. INPUT TO ist z.B. eine Standardrelation zwischen einem Objekt der Klasse DATA und einem der Klasse ALPHA.

- Attribute

Attribute dienen zur Beschreibung von Elementeigenschaften. Ein Beispiel eines Attributes, das Elementen vom Typ DATA zugeordnet werden kann, ist INITIAL-VALUE.

- Strukturen

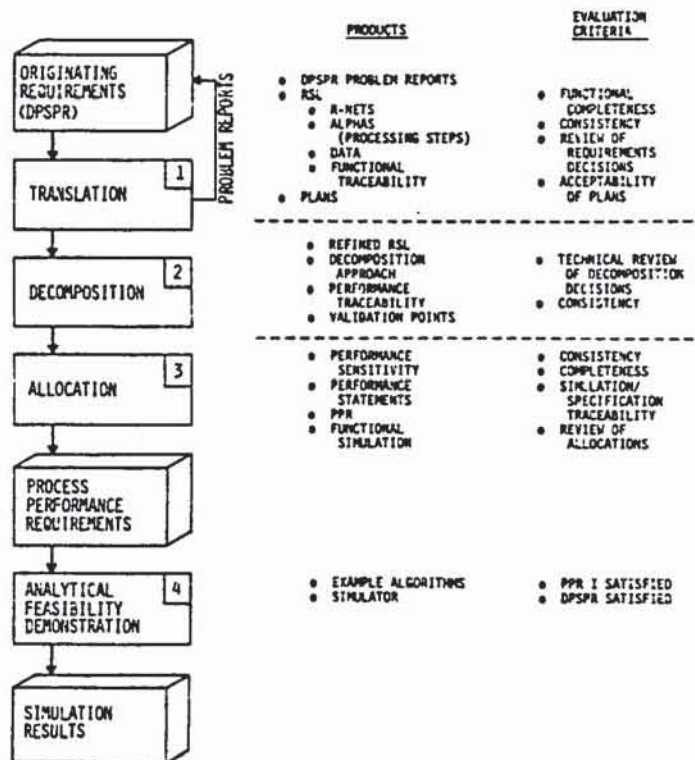
Strukturen dienen zur Beschreibung von R-Nets, d.h. zur Beschreibung des konzeptionell zweidimensionalen Graphen (Ablaufmodell) durch die eindimensionale Sprache RSL.

Zur Strukturierung der Verarbeitungspfade stehen nur die Sprachelemente AND, OR und FOREACH zur Verfügung. Auf den Verarbeitungspfaden lassen sich Leistungsanforderungen durch das Einfügen von sogenannten Validations-Punkten definieren. Diesen Punkten werden überprüfbare Genauigkeits- und Zeitbedingungen zugeordnet.

2.12.4 Entwicklungsmethode

Die Beschreibung der Anforderungen in RSL wird interaktiv oder im Batch-Betrieb dem System eingegeben. Diese Daten werden mit den bereits in der Datenbank gespeicherten Informationen vom Analysator REVS auf Konsistenz und Vollständigkeit geprüft. Außerdem können Simulationen durchgeführt werden zur Feststellung, ob die Anforderungen erfüllt sind.

Im nachfolgenden Diagramm sind die einzelnen Schritte der Erstellung, die Ergebnisse (Dokumente) der einzelnen Schritte sowie die jeweils zugeordneten Prüfungen dargestellt.



Dem Zerlegungsschritt 2 liegt im wesentlichen das Konzept der schrittweisen Verfeinerung zugrunde, das von RSL durch das Sprachkonzept SUB-NET unterstützt wird.

SREP kann als eine Weiterentwicklung von PSL/PSA aufgefaßt werden. Hervorzuheben sind hier vor allem ein weiterer Ausbau der Testmöglichkeiten durch Simulation und die Ablauf-Orientierung.

2.12.5 Literatur

Bell, T.E.; Bixler, D.

A flow-oriented requirements statement language

TRW-SS-76-02, April 1976

Davis, C.G.; Vick, C.R.

The Software Development System

Proc. IEEE/ACM 2nd International Conference on Software Engineering

San Francisco, October 1976

Alford, M.

A requirements engineering methodology for real-time processing
requirements

ibid

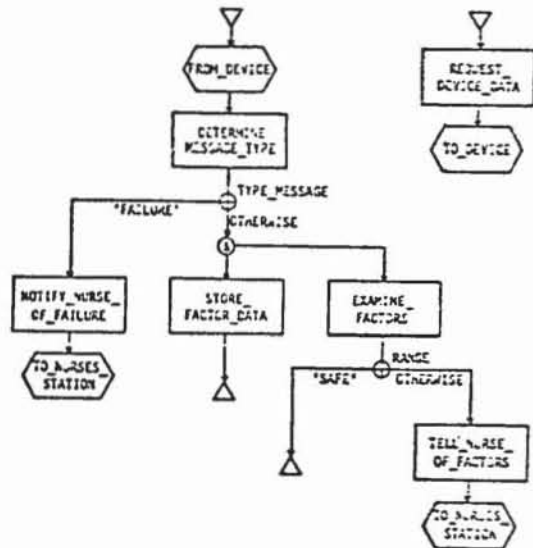
2.12.6 Beispiel /Alford/

R-NET

Krankenhausüberwachung

The problem is the following:

"(1) A patient monitoring program is required for a hospital. (2) Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood-pressure, and skin resistance. (3) The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a data base. (4) For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range is 98 to 99.5 degrees Fahrenheit). (5) If a factor falls outside of the patient's safe range, or if an analog device fails, the nurse's station is notified".



ORIGINATING_REQUIREMENT: SENTENCE_2.
 DESCRIPTION: "DEFINES ANALOG DEVICE MEASUREMENTS".
 TRACES TO: MESSAGE DEVICE_REPORT.

MESSAGE: DEVICE_REPORT.
 PASSED THROUGH: INFJT_INTERFACE FROM_DEVICE.
 MADE BY: DATA DEVICE_NUMBER, DATA TYPE_MESSAGE,
 DATA DEVICE_DATA.
 TRACED FROM: SENTENCE_2.

DATA: DEVICE_DATA.
 INCLUDES: DATA PULSE, DATA TEMPERATURE,
 DATA BLOOD_PRESSURE,
 DATA SKIN_RESISTANCE.

ENTITY_CLASS: PATIENT.
 ASSOCIATES: DATA PATIENT_NUMBER,
 DATA SAFE_FACTOR_RANGE_FILE
 FACTOR_HISTORY.

DATA: SAFE_FACTOR_RANGE.
 INCLUDES: DATA LOW_PRESSURE, DATA HI_PRESSURE,
 DATA LOW_TEMPERATURE, DATA HI_TEMPERATURE,
 DATA LOW_SKIN_RESISTANCE,
 DATA HI_SKIN_RESISTANCE.
 TRACED FROM: SENTENCE_4.

FILE: FACTOR_HISTORY.
 CONTAINS: DATA MEASUREMENT_TIME, DATA HPULSE,
 DATA HI_TEMPERATURE, DATA HBLOOD_PRESSURE,
 DATA HSKIN_RESISTANCE.
 TRACED FROM: SENTENCE_3.

ALPHA: EXAMINE_FACTORS.
 INPUTS: DATA DEVICE_DATA, DATA SAFE_FACTOR_RANGE.
 OUTPUTS: RANGE.
 DESCRIPTION: "THIS PROCESSING STEP FIRST RETRIEVES THE SAFE
 FACTOR DATA ASSOCIATED WITH THE DEVICE, COMPARES
 THE DEVICE DATA TO THE SAFE FACTOR RANGES FOR
 THE PATIENT BEING MONITORED, AND DETERMINES
 WHETHER THE FACTORS ARE IN_BOUNDS (RANGE_SAFE)
 OR OUT_OF_BOUNDS".

DATA: PATIENT_DEVICE_NUMBER.
 INCLUDED IN: DATA SAFE_FACTOR_RANGE.

2.13 Stanford Research Institute/Parnas Design Methodology (SRI/PARNAS)

2.13.1 Kurzbeschreibung

In den bekannten Papieren /Pa 72a, b/ hat Parnas seine Ideen zur formalen Spezifikation von Modulen und Entwurfskriterien für die Modularisierung von Systemen entwickelt. Parnas hat zwar schematische Aussagen über den Beschreibungsinhalt von Modulen und deren Schnittstellen gemacht, aber keine präzise Syntax und Semantik definiert. Ausgehend von seinen Überlegungen wurde am SRI die formale Spezifikationssprache SPECIAL entwickelt, in der die Konzepte von PARNAS eingebettet wurden und die eine Verifikation des Entwurfs ermöglicht. Die Sprachelemente wurden dabei so ausgewählt, daß sie die sogenannte 'SRI-Hierarchical Design Methodology' unterstützen.

Die wesentlichen Konzepte von Parnas waren

- Information hiding - Jeder Modul enthält eine wichtige Entwurfsentscheidung, die den anderen Modulen nicht bekannt ist. Die Aufteilung in Module wird dabei so gewählt, daß die Änderung einer Entwurfsentscheidung in möglichst wenigen Modulen Änderungen erfordert.
- Unabhängige Entwicklung - Durch weitgehend abstrakte und einfache Schnittstellen sollte die unabhängige Entwicklung der einzelnen Module früher beginnen als sonst.
- Verständlichkeit - Jeder Modul soll für sich verständlich sein und nicht nur das System als Ganzes.

Um diese Entwurfskriterien hat Parnas ein Spezifikationsschema entwickelt, das auf einem Zustandsmodell basiert. Eine Modulspezifikation nach Parnas kann als abstrakte Maschine aufgefaßt werden, deren Zustand und Übergangsfunktion durch zwei Arten von Funktionen charakterisiert werden, den V-Funktionen (Value- functions) und den O-Funktionen (Operations functions).

Jede V-Funktion liefert einen Wert; die Menge der V-Funktionen eines Moduls definiert den Zustand des Moduls. Jede O-Funktion beschreibt einen Zustandsübergang, indem sie neue Werte für V-Funktionen definiert. Der Zustandsübergang bei Aufruf einer O-Funktion wird beschrieben durch Zusicherungen, die Werte von V-Funktionen nach dem Aufruf mit ihren Werten vor dem Aufruf in Beziehung setzen.

Jede V-Funktion hat einen Anfangswert, der durch eine Zusicherung beschrieben wird. Aufrufe von V-Funktionen und O-Funktionen eines Moduls sind durch Ausnahme-Bedingungen (exception conditions) eingeschränkt zur Charakterisierung nicht definierter Zustandsübergänge. In einer Implementierung sind dies die Fehlerausgänge.

Die Zustandsübergänge werden im effects-Teil einer O-Funktion beschrieben. V-Funktionsnamen in Hochkommata ('...') stellen Werte von V-Funktionen vor dem Aufruf der O-Funktion dar, während V-Funktionsnamen ohne Hochkommata neue Funktionswerte nach Beendigung des Aufrufs darstellen.

Die Spezifikationssprache SPECIAL (Spezification and Assertion Language) kann als Formulierung dieses Spezifikationsschemas von Parnas betrachtet werden.

Nachfolgend wird dieses System vergleichsweise ausführlich behandelt, da die Ideen von Parnas grundlegend für die Entwicklung des gesamten Gebietes der Spezifikations- und Entwurfsebene waren (historisch und konzeptionell).

2.13.2 Anwendungsphase und Gebiet

Die SRI-Entwurfsmethode stellt formale Hilfsmittel für den funktionalen Entwurf einer großen Klasse von Systemen zur Verfügung (dabei sollen sowohl Hardware- wie auch Softwaresysteme enthalten sein).

Da keine ablauforientierten Beschreibungsmittel (Zeitbedingungen etc.) in SPECIAL enthalten sind, ist SRI für Realzeitanwendungen weniger geeignet.

2.13.3 Inhalt und Form der Darstellung

Die Spezifikationsprache SPECIAL erlaubt die Beschreibung eines Systementwurfs auf zwei Ebenen, der sogenannten Assertion-Ebene (innere Ebene) und der Spezifikations-Ebene (äußere Ebene). Auf der Assertion-Ebene wird das Verhalten eines Systems zusammen mit seinen abstrakten Eigenschaften in Form von nicht-prozeduralen Ausdrücken der Aussagenlogik und der Mengenlehre beschrieben.

Die Spezifikations-Ebene ermöglicht die Beschreibung eines Systems als eine Hierarchie von Modulen, wobei jeder Modul als eine abstrakte Maschine durch einen Zustand und Operationen für Zustandsänderungen definiert wird.

Das Verhalten eines Moduls wird durch eine formale Spezifikation der Funktionen beschrieben, die von anderen Modulen aufgerufen werden können. Der Zustand eines Moduls wird dargestellt durch die Werte seiner V-Funktionen. Die Zustandstransformationen eines Moduls werden als O-Funktionen bezeichnet. Jede Transformation wird durch eine Menge von Zusicherungen beschrieben, die die Werte von V-Funktionen vor Funktionsaufruf mit ihren Werten nach Funktionsaufruf in Beziehung setzen. OV-Funktionen sind Funktionen, die sowohl Werte haben, als auch Zustandstransformationen beschreiben. Außer zur Modulspezifikation werden Zusicherungen in SPECIAL noch zur Spezifikation von Beziehungen zwischen den Zuständen (Werte von V-Funktionen) auf verschiedenen Ebenen der Systemhierarchie benutzt. Diese Relationen werden als Abbildungsfunktionen (mapping functions) bezeichnet.

Syntax der Assertion-Ebene

Zusicherungen (Assertions) werden in SPECIAL zur Beschreibung des Systemverhaltens benutzt, z.B. zur Formulierung von Zustandstransformationen, Fehlerbedingungen, invarianten Moduleigenschaften und Bedingungen, die zu einem bestimmten Zeitpunkt der Programmausführung erfüllt sein müssen.

SPECIAL enthält eine Reihe von Objekten und Operationen zur Formulierung von Zusicherungen; u.a. können quantifizierte Ausdrücke, wie $\text{FORALL } x \text{ INSET } s: p(x)$ für alle x aus der Menge s ist $p(x)$ wahr) und Ausdrücke zur Charakterisierung von Objekten wie $\text{LET } x \mid \text{ EXISTS } z: z > 0 \text{ AND } z \text{ MOD } 2 = 0 \text{ AND } x = t(z+1) \text{ IN } f(x) + g(x)$ gebildet werden. Weiter gibt es bedingte Ausdrücke und den soge-

nannten TYPECASE Ausdruck zur Identifizierung des aktuellen Typs eines Objekts vom Vereinigungstyp:

```
TYPECASE x OF      | wobei Type x= UNION (TYPE 1, TYPE 2)
  TYPE 1 : f (x)   |
  TYPE 2 : g (x)   |
END
```

Syntax der Spezifikations-Ebene

Durch die Spezifikationsebene wird ein syntaktischer Rahmen zur Modulspezifikation festgelegt, in den die Ausdrücke der Assertions-Ebene eingebettet werden.

Die Spezifikation eines Moduls wird in sechs Abschnitte unterteilt:

```
MODULE    <symbol>
  TYPES ...
  DECLARATIONS ....
  PARAMETERS .....
  DEFINITIONS ...
  EXTERNALREFS ...
  FUNCTIONS ...
END-MODULE
```

<symbol> ist der Modulname.

- TYPES enthält die Definitionen aller Typenvereinbarungen.
- DECLARATIONS enthält alle Variablendeklarationen.
- PARAMETERS enthält die Deklarationen für symbolische Konstanten, die initialisiert werden, bevor der Modul benutzt wird, und vom Modul nicht verändert werden können (z.B. maximale Tiefe eines Kellers).
- DEFINITION enthält die Definition von Makros, die global für den gesamten Modul sind.
- EXTERNALREFS enthält externe Objektdeklarationen, die in der Spezifikation des Moduls benutzt werden.
- FUNCTIONS enthält die Definitionen für alle V-, O-, OV-Funktionen des Moduls.

Zum besseren Verständnis des Beispiels wird im folgenden noch ausführlicher auf die Syntax des FUNCTIONS-Abschnitts eingegangen.

Definition der V-Funktionen

V-Funktionen dienen zur Beschreibung des Modulzustandes. Der aktuelle Wert einer V-Funktion wird nicht explizit definiert, sondern durch Induktion: Die Spezifikation enthält den Anfangswert, und der Wert zu einem bestimmten Zeitpunkt ergibt sich aus der Folge von O-Funktionsaufrufen bis zu diesem Zeitpunkt.

Eine V-Funktion kann entweder als hidden oder visible und als primitive oder derived erklärt werden. Ist sie hidden, so kann sie nicht von anderen Modulen aufgerufen werden. Der Wert einer derived V-Funktion ist ein Ausdruck, der von den Werten anderer V-Funktionen des Moduls abgeleitet wurde, während eine primitive V-Funktion direkt einen Teil der Zustandsdefinition beschreibt.

Syntax einer V-Funktion: *)

```
VFUN <symbol> <formalargs>→<declaration>;  
    [DEFINITIONS {<definition>;}+ ]  
    [HIDDEN/EXCEPTIONS {<expression>;}+ ]  
    [INITIALLY/DERIVATION <expression> ;]
```

Die erste Zeile erklärt die formalen Argumente und das Ergebnis der V-Funktion. Die zweite Zeile enthält lokale Makrodefinitionen. Die dritte Zeile spezifiziert die Funktion als hidden oder visible. Ist die Funktion visible, so enthält die dritte Zeile das Schlüsselwort EXCEPTIONS gefolgt von booleschen Ausdrücken, die Fehlerbedingungen angeben. Ist die Funktion primitive, so enthält die vierte Zeile das Schlüsselwort INITIALLY, gefolgt von einer Zusicherung über den Anfangswert der Funktion. Ist die Funktion derived, so enthält sie das Wort DERIVATION, gefolgt von einem Ausdruck für den Anfangswert.

Als Beispiel betrachten wir die V-Funktionen eines Moduls zur Verwaltung eines Kellers.

*) [...] optional
 {...} + beliebig oft, jedoch mindestens einmal
 .../... alternativ

Die Zustandsinformation ist in den V-Funktionen "stack" und "ptr" enthalten.

```
VFUN ptr() INTEGER i ;
      INITIALLY i = 0 ;

VFUN stack (INTEGER i) INTEGER j;
      HIDDEN;
      INITIALLY j=?;
                                     (?=UNDEFINED)
```

Der Wert von ptr bezeichnet die Tiefe des Kellers.

Definition der O- und OV-Funktionen

Syntax einer O-Funktion

```
OFUN <symbol> <formalargs>;
      [DEFINITIONS { <definition>;}+]
      [EXCEPTIONS  { <expression>;}+]
      [EFFECTS     { <expression>;}+]
```

Die Syntax einer OV-Funktion weicht davon nur in der ersten Zeile ab:

```
OVFUN <symbol> <formalargs> → <declaration>;
```

Der EFFECTS-Abschnitt beschreibt die Zustandstransformationen durch Zusicherungen. Diese verknüpfen die Werte der V-Funktionen vor und nach dem Aufruf der O- und OV-Funktionen. Die Werte von V-Funktionen nach dem Aufruf werden durch vorangestellte Hochkommata markiert.

Als Beispiel einer O-Funktion sei die Funktion "push" angegeben, die die oben definierten V-Funktionen benutzt:

```
OFUN push (INTEGER j);
      EXCEPTIONS ptr () > = maxsize;
      EFFECTS 'stack ( 'ptr () ) = j;
              'ptr () = ptr () + 1;
```

"maxsize" ist die maximal erlaubte Kellertiefe.

2.13.4 Entwicklungsmethode

Die SRI-Entwurfsmethode basiert auf der Spezifikationsprache SPECIAL. Für die Verwendung dieses Hilfsmittels wurde ein schrittweises Vorgehen bei der Softwareentwicklung vorgeschlagen:

Schritt 0: Schnittstellen-Definition

In diesem Schritt sollen die geforderten Schnittstellen aus der Sicht des Benutzers beschrieben werden. Diese Schnittstellen werden einer Menge von Modulen zugeordnet, die jeweils Objekte eines bestimmten Typs verwalten.

Schritt 1: Hierarchische Zerlegung des Systems

Die Module werden verschiedenen Ebenen einer hierarchischen Struktur zugeordnet.

Schritt 2: Modul-Spezifikation

In diesem Schritt werden die Module nach dem durch SPECIAL definierten formalen Spezifikationsschema beschrieben.

Schritt 3: Abbildungsfunktionen

Für alle Module, die nicht der niedrigsten Entwurfsebene zugeordnet sind, wird eine Abbildungsfunktion definiert, die den Zustand eines Moduls einer bestimmten Ebene durch die Zustände von Modulen einer niedrigeren Ebene beschreibt.

Dabei werden V-Funktionen einer höheren Ebene durch Ausdrücke expandiert, die V-Funktionen einer niedrigeren Ebene enthalten. Diese Ausdrucksmittel (Mapping function expressions) sind als Sprachkonstrukte im SPECIAL enthalten.

Zur Zeit werden am SRI automatische Prüfmittel entwickelt, die gewisse Analysen für einen in SPECIAL formulierten Entwurf ermöglichen, u.a. Hierarchy-manager, Specification analyzer, Mapping function analyzer, Model consistency checker.

2.13.5 Literatur

- /Ro 75/ Robinson, L. et.al.
On Attaining Reliable Software for a Secure Operating System
Intern. Conference on Reliable Software,
21.-23. April, 1975, Los Angeles, CA.
New York, IEEE 1975, pp. 267-284
- /Neu 76/ Neumann, P.G. et.al.
Software Development and Proofs of Multi Level Security
2nd Intern. Conference on Software Engineering,
13.-15. October, 1976, San Francisco, CA.
- ^s
/RoRou
76/ Robinson, L.; Roubine, O.
SPECIAL - A Specification and Assertion Language
Stanford Research Institute, Menlo Park, CA.
- /Pa 72a/ Parnas, D.L.
A Technique for Software Module Specification with
Examples
CACM 15, 5 (1972) pp. 330-336
- /Pa 72b/ Parnas, D.L.
On the Criteria to be used in Decomposing Systems into
Modules
CACM 15, 12 (1972), pp. 1053-1058

2.13.6 Beispiel

Im folgenden ist die Spezifikation eines Moduls angegeben, der eine Menge von Kellern als einen abstrakten Datentyp verwaltet.

Erläuterungen:

STACK_NAME: Typbezeichner für die einzelnen Keller

nstacks : Anzahl der augenblicklich existierenden Keller

empty (s) : Testprädikat für einen leeren Keller s.

s : bezeichnet einen Keller

i : bezeichnet den Wert eines Zeigers

j : bezeichnet den Wert eines Kellerelements

\$ (...) : Kommentare

MODULE stacks

TYPES

stack_name: DESIGNATOR;

DECLARATIONS

INTEGER i, j;
stack_name s;

PARAMETERS

INTEGER maxsize \$(maximum size of a given stack) ,
maxstacks \$(maximum number of stacks allowed) ;

DEFINITIONS

INTEGER nstacks IS CARDINALITY({ stack_name s | ptr(s) \neq ? }):

BOOLEAN empty(stack_name s) IS ptr(s) = 0;

FUNCTIONS

VFUN ptr(s) -> i:
INITIALLY
i = ?;

VFUN stack(s: i) -> j:
HIDDEN;
INITIALLY
j = ?;

VFUN top(s) -> j:
EXCEPTIONS
ptr(s) = ?;
empty(s):
DERIVATION
stack(s, ptr(s));

OVFUN create_stack() -> s:
EXCEPTIONS
nstacks \geq maxstacks;
EFFECTS
s = NEW(stack_name):
ptr(s) = 0;

OFUN delete_stack(s):
EXCEPTIONS
ptr(s) = ?;
EFFECTS
ptr(s) = ?;
FORALL i: stack(s, i) = ?;

OFUN push(s: j):
EXCEPTIONS
ptr(s) = ?;
ptr(s) \geq maxsize;
EFFECTS
stack(s, ptr(s)) = j:
ptr(s) = ptr(s) + 1;

OVFUN pop(s) -> j:
EXCEPTIONS
ptr(s) = ?;
empty(s):
EFFECTS
j = top(s):
ptr(s) = ptr(s) - 1;
stack(s, ptr(s)) = ?;

END_MODULE

2.14 Software Specification and Evaluation System (SSES)

2.14.1 Kurzbeschreibung

Das SSES wird von Science Applications, Inc. für die NASA entwickelt. Es besteht im wesentlichen aus der Entwurfssprache SSL, einem strukturierten Preprocessor für eine FORTRAN-Erweiterung, einem Datenbanksystem und einem automatischen Testsystem.

SSES ist ein rechnergestütztes System zur Erstellung einer vollständigen, eindeutigen und widerspruchsfreien Entwurfsspezifikation.

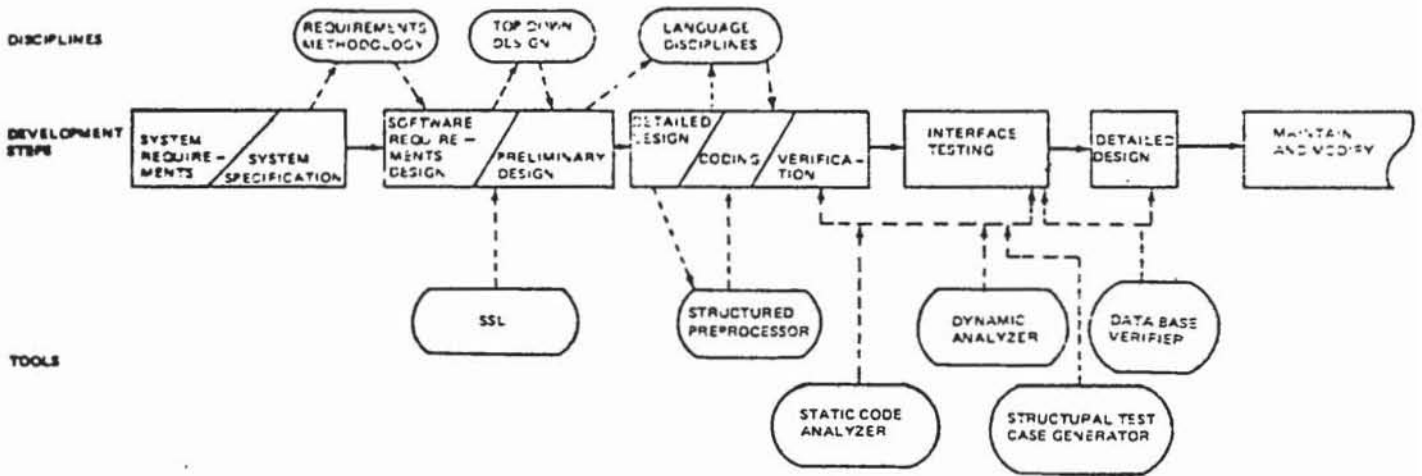
Die nichtprozedurale Entwurfssprache SSL erlaubt die explizite Beschreibung von Informationen, wie sie normalerweise in einem funktionellen Blockdiagramm eines Software-Systems enthalten sind (statische Struktur). Darüber hinaus können Datenstrukturen, Modulschnittstellen und Ein-/Ausgabe-Variablen eines Moduls beschrieben werden. Elementare Beschreibungseinheit ist der Modul. Ein oder mehrere Moduln können zu Subsystemen zusammengefaßt werden. Aufgrund der nichtprozeduralen Eigenschaft von SSL wird der Kontrollfluß innerhalb eines Moduls nicht spezifiziert, das dynamische Verhalten eines Moduls kann jedoch durch die Definition von Zusicherungen (Assertions) beschrieben werden.

Im Vergleich zu anderen rechnergestützten Systemen beschränken sich die Prüfmittel von SSES auf die Analyse des erzeugten Codes. Durch die formale Beschreibung der Modulschnittstellen und die Formulierung von Zusicherungen werden jedoch im Entwurf Vergleichsdaten bereitgestellt (in der Datenbank), die das Testsystem unterstützen. Das Testsystem enthält Möglichkeiten der statischen und dynamischen Analyse und zur Testdatenerzeugung. Um aus der Entwurfsbeschreibung in SSL gut strukturierte Programme zu entwickeln, wurde eine geeignete FORTRAN-Erweiterung definiert. Ein wesentliches Kriterium für die Erweiterung war dabei die leichtere Analysierbarkeit des erzeugten Codes.

2.14.2 Anwendungsphase und Gebiet

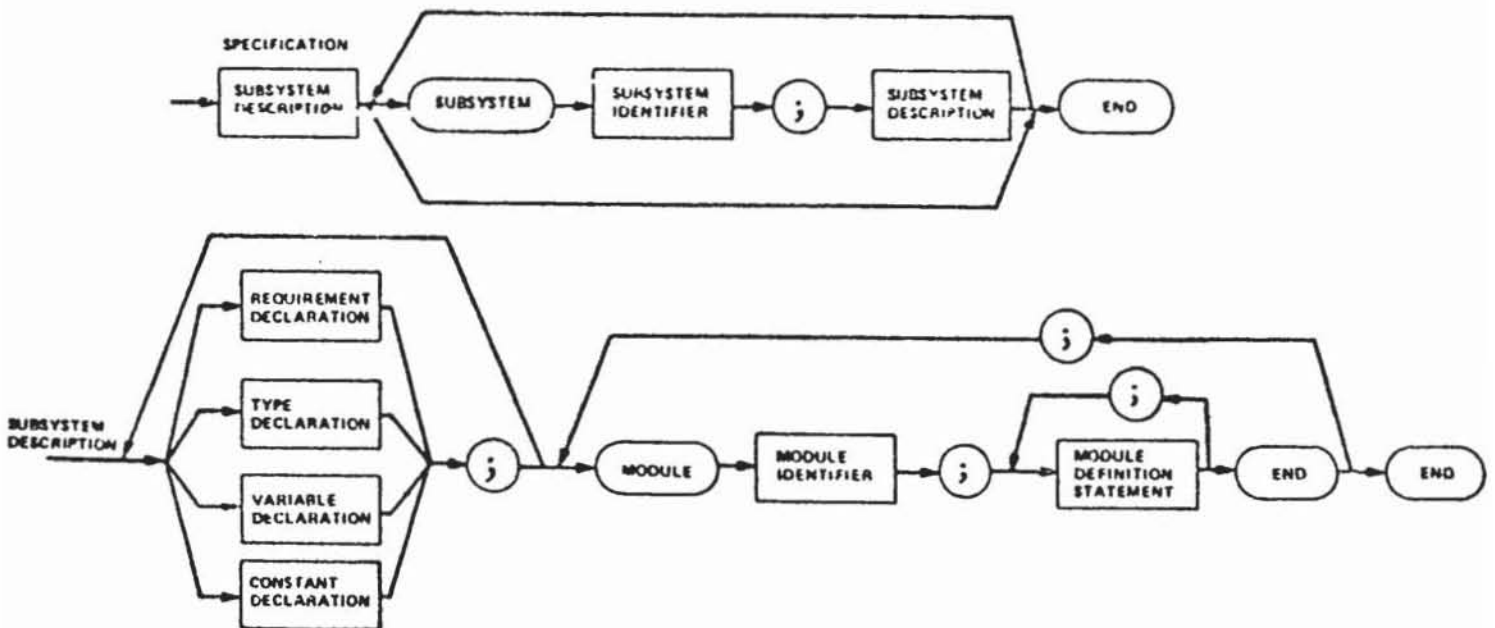
Das Anwendungsgebiet ist nicht genauer umrissen, jedoch werden auch Realzeitanwendungen in Betracht gezogen.

Die Anwendungsphase der einzelnen SSES-Komponenten läßt sich aus folgendem Schema entnehmen:

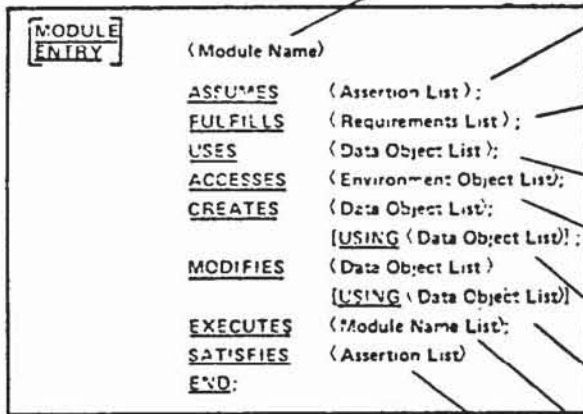


2.14.3 Inhalt und Form der Darstellung

Zur Beschreibung des Entwurfs steht die Entwurfssprache SSL zur Verfügung. Neben einer formalen Syntax (kontextfrei BNF) existiert eine mengentheoretische Beschreibung der Semantik. Eine Entwurfsbeschreibung besteht aus Subsystemen, die wiederum aus Modulen zusammengesetzt sind:



Die Syntax für eine Modulspezifikation wird im folgenden angegeben:



Identifikation eines neuen Moduls durch seinen Namen, es wird zwischen zwei Modultypen unterschieden.

Ausgabe von Zusicherungen (assertions), die vor Modulausführung gültig sein müssen.

Aufzählungen der Anforderungen, die durch diesen Modul realisiert werden.

Auflisten der Eingabedaten.

Auflisten von Elementen der Software-Umgebung, die benutzt werden.

Ausgabedaten, die initialisiert werden.

Ausgabedaten, die verändert werden.

Angabe der aufrufbaren Moduln.

Angabe von Zusicherungen (assertions), die nach Modulausführung gültig sein müssen.

Man erkennt, daß SSL eine minimale Möglichkeit enthält, die Hardwareschnittstellen zu beschreiben (ACCESS). Es existieren Sprachmittel zur Beschreibung von Datenstrukturen und zur Angabe von Eingabedaten (USES) und Ausgabedaten (CREATES, MODIFIES). Das EXECUTES-Statement kann näher spezifiziert werden durch Sprachmittel, die eine bedingte, wiederholte oder rekursive Ausführung von Moduln ausdrücken lassen. Ein Modulname, der durch MODULE spezifiziert ist, bezeichnet einen Modul, der nur innerhalb des Subsystems benutzt werden kann, in dem er deklariert wurde. Ein durch ENTRY spezifizierter Modul kann nur von anderen Subsystemen benutzt werden.

Beispiel einer Modulbeschreibung in SSL:

```
MODULE SORT (N:INTEGER) ;
  /* MODULE TO SORT ARRAY */
  /* ARRAY IS INITIALIZED FROM CARD READER */

  ASSUMES          N > 0 ;
  FULFILLS         ORDERED _ VALUE;
  ACCESSES        CARD_READER;
  MODIFIES        SARRAY USING N;
  SATISFIES       FORALL (I:INTEGER)
                  I > 0 AND I ≤ N-1
                  AND
                  SARRAY [I] ≥ SARRAY [I+1]

END;
```

2.14.4 Entwicklungsmethode

Die Entwurfsbeschreibung in SSL dient als Grundlage für die Implementierung und kann in einer Datenbank abgespeichert werden.

Als Anleitung für den Entwurf werden folgende Regeln angegeben:
(Sie sollen die Software-Struktur beeinflussen)

- o Subsysteme sollen zur Zerlegung des Systems in Abstraktionsebenen dienen.
- o Die Moduln innerhalb eines Subsystems haben keine gemeinsamen Daten (Files, COMMON) mit Moduln in anderen Subsystemen.
- o ENTRY-Moduln in einem Subsystem werden nur von Moduln anderer Subsysteme aufgerufen.
- o Ein Modul, der von einem Modul innerhalb eines Subsystems direkt aufgerufen wird, darf nie direkt von Moduln anderer Subsysteme aufgerufen werden.
- o Alle Subsysteme erfüllen eine oder mehrere der folgenden Eigenschaften:
 - Information hiding: Das Subsystem faßt Entwurfsentscheidungen zusammen, die eine große Änderungswahrscheinlichkeit haben.
 - Betriebsmittelverwaltung: Das Subsystem hat exklusiven Zugriff auf die Peripherie oder bestimmte Datenstrukturen.
 - logisch vollständig: Das Subsystem ist ein wiederverwendbarer Baustein.
 - Realzeitprozeß: Das Subsystem ist eine asynchrone Task in einer Realzeitanwendung.

SSES ist von seinem Ansatz her (formale Beschreibungssprache, Datenbankorientierung) mit den rechnergestützten Systemen PSL/PSA und SREP verwandt, besitzt aber nicht deren Mächtigkeit. Neben erweiterten Beschreibungsmöglichkeiten besitzen jene Systeme vor allem Prüfmittel auf Spezifikations-Entwurfsebene.

2.14.5 Literatur

Austin, S.L. et. al.

SSL - A Software Specification Language

Science Applications, Inc. Huntsville

Report SAI-77-537-HU, 1976

Hodges, B.C.; Ryan, J.P.

A system for automatic software evaluation

Proc. IEEE/ACM 2nd Int. Conference on Software Engineering

San Francisco, October 1976, pp. 617-623

2.14.6 Beispiel: Telegrammverarbeitung /Au 76/

The example of this section was selected to demonstrate both the descriptive level of SSL and as many language elements as possible. The requirement of the problem may be stated as follows :

"A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer where it is to be processed. The words in the telegram are separated by sequences of blanks and each telegram is delimited by the word 'ZZZ'. The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words 'ZZZ' and 'STOP' are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word."

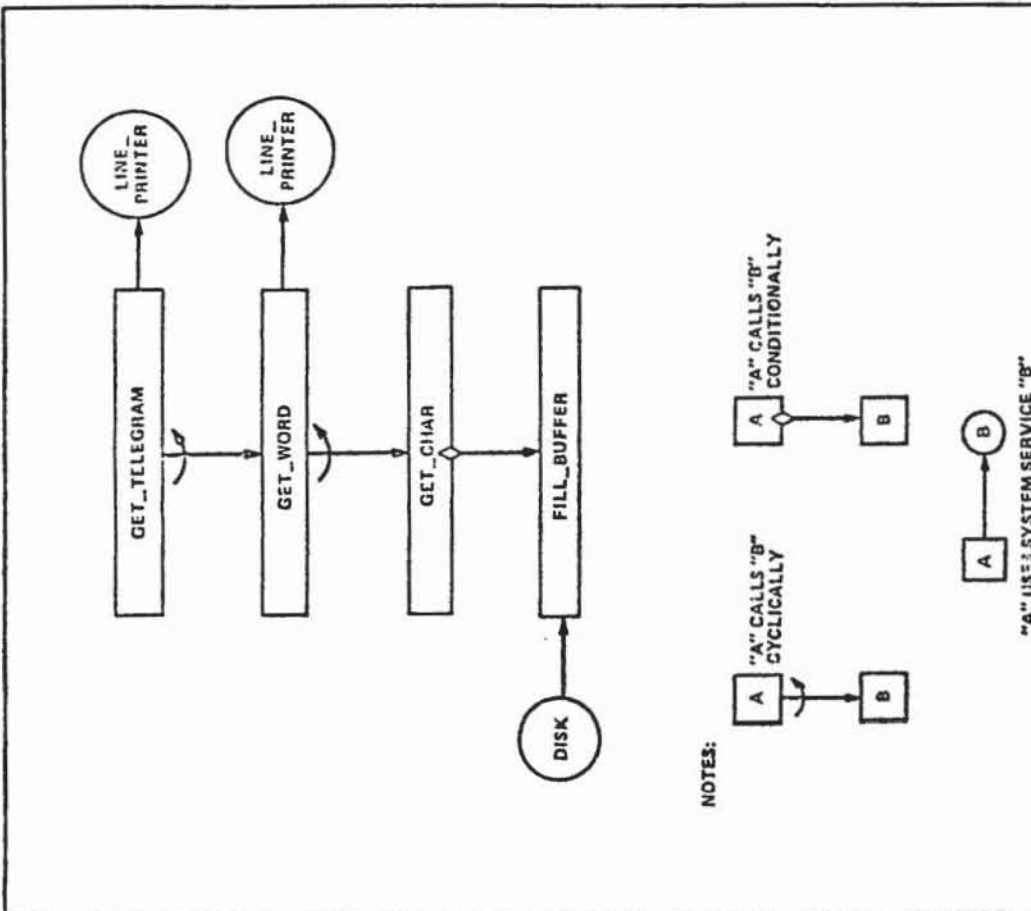
To complete the problem statement, several assumptions are necessary. The following alternatives were selected for the purpose of this exposition:

- The character stream from which the telegrams are constructed resides on a drum having fixed length records; the record length itself is left as an implementation option.
- The chargeable word count is the value to be printed and overlength words count as one word.
- If a physical end of file is encountered before the logical end of the data stream, an error message and the partial telegram is printed.

The software is organized into four modules as indicated by Figure 4.44a. The purpose of each module is given in Table.

Figure 4.44b contains the SSL description of the telegram processor. The right margin of the statement listing contains reference notes to subsections containing detailed descriptions of the language elements used.

A careful examination of Figure 4.44b will indicate an interesting application of the subsystem capability. The subroutines GET_CHAR and FILL_BUFFER occupy a separate subsystem with the sole purpose of handling file I/O. The characteristics of the device on which the telegrams are stored are encapsulated within these two modules.



MODULE DESCRIPTIONS FOR EXAMPLE	
<u>MODULE</u>	<u>PURPOSE</u>
GET_TELEGRAM	Collects words belonging to each telegram and prints them in a neat manner along with the chargeable word count.
GET_WORD	Collects characters into words and prints error messages denoting over-length word or physical record end of file.
GET_CHAR	Returns the next character in the telegram file.
FILL_BUFFER	Enters the next physical record from the drum into the character buffer.

Figure SSES-1 : Module Structure Chart for Example

```
/* beginning of main subsystem preamble */

requirement
  transductions
  collect in print;
  output
  telegram, charge_count
  end;

variable telegram:text;
  charge_count:integer:
    for print;
    subjectto charge_count ≥ 0
  word_count:integer:
    for print;
    subjectto word_count ≥ charge_count;
  word:array [1..12] of char;
    for print;
  eof_flag:boolean;
    for print

end; /* end of main subsystem preamble */

/* main routine to collect words and */
/* print telegram with chargeable word count */

module get_telegram;
  fulfills print;
  creates telegram, charge_count using word;
  creates word_count;
  modifies word_count;
  uses eof_flag;
  accesses line_printer;
  executes cyclically get_word;
  satisfies
    eof_flag or word_count = 0
  end;

/* subroutine to collect characters into */
/* words */

module get_word;
  fulfills collect;
  executes cyclically i_o.get_char(a_char:char:eof_flag);
  creates word, eof_flag;
  accesses line_printer /*prints error messages */
  end

end; /* end of main subsystem */
```

Figure SSES-2 : SSL Description for Example

```
/* beginning of i_o subsystem preamble */  
  
subsystem i_o;  
  
requirement  
    input character_file;  
    transductions  
    read in separate;  
    output a_char, eof_flag  
    end;  
  
/* parameterize record length */  
constant record_length = integer;  
type character_record = array [1..record_length] of char;  
variable character_file:sequence of character_record;  
    for read;  
    buffer:character_record;  
    for separate;  
    a_char:char;  
    for separate;  
    char_index:1..record-length;  
    for separate;  
    eof_flag:boolean;  
    for separate  
  
end: /* end of subsystem preamble */  
  
  
/* subroutine to fetch next */  
/* character from file */  
  
entry get_char (a_char; eof_flag);  
    fulfills separate;  
    executes conditionally fill_buffer;  
    modifies char_index;  
    creates a_char using buffer [char_index], eof_flag;  
    creates character_file, char_index;  
    satisfies eof_flag implies a_char = buffer [char_index],  
    end;  
  
/* subroutine to fetch next physical */  
/* record from character file */  
  
module fill_buffer;  
    fulfills read;  
    assumes char_index = record_length;  
    accesses disk;  
    creates buffer, eof_flag using character file ;  
    satisfies  
        eof_flag implies buffer = character_file  
    end  
  
end /* end of subsystem */  
end; /* end of specification */
```

Figure SSES-2 : SSL Description for Example (continued)

3. Vergleichende Gegenüberstellungen

In den folgenden Abschnitten werden Informationen über einzelne Systeme nach bestimmten Kriterien zusammenfassend gegenübergestellt. Da bestimmte Systeme zu bestimmten Kriterien keine Aussagen enthalten (weil z.B. diese Aspekte nicht beschreibbar sind) oder diese nicht zugänglich waren, bezieht sich ein Teil dieser Gegenüberstellungen jeweils nur auf Teilmengen der betrachteten Systeme.

3.1 Vergleich des Inhalts, der Form der Darstellung und der Entwicklungsmethoden

Betrachtet man die einzelnen Ansätze unter den Gesichtspunkten des Inhalts, der Form der Darstellung und der Methode, durch die der Inhalt der Systembeschreibung erlangt wird, so kommt man zu folgendem Ergebnis:

1. Methoden, die für die gleiche Phase der Software-Entwicklung konzipiert wurden, unterscheiden sich im wesentlichen nicht durch die Art des Inhalts, den sie beschreiben, obwohl unterschiedliche Systemaspekte einbezogen werden.
2. Die Methoden unterscheiden sich stark in der Form der Darstellung.
3. Die Methoden unterscheiden sich auch in der Vorgehensweise (Methode), wobei diese jedoch in den meisten Fällen nicht sehr detailliert beschrieben wird.
4. Inhalt, Form der Darstellung und Methode sind im wesentlichen unabhängig und können daher unabhängig ausgewählt werden.
5. Diejenigen Verfahren, die die Entwurfsmethode in den Vordergrund stellen, sind vorwiegend manuell, während ein Teil der anderen, die Darstellung betonenden Verfahren, die maschinelle Speicherung und Verarbeitung erlauben. Dies hat den Vorteil, daß der Inhalt nach verschiedenen Kriterien geordnet und ausgegeben werden kann (begleitende Dokumentation).

Die beschriebenen Systeme lassen sich drei Gruppen zuordnen:

1. Entwurfsmethoden,
2. Darstellungsmethoden ohne Mittel zur automatischen Analyse,
3. Darstellungsmethoden mit Mitteln zur automatischen Analyse.

Dies ist in der folgenden, entsprechend geordneten Tabelle erkennbar:
(Tab. 3.2.a)

Tabelle 3.2.b gibt die Eignung der verschiedenen Systeme für die Lösung von Aufgaben an, die charakteristisch sind für DV-Projekte. Dabei sind die letzten fünf Spalten nur für Systeme mit Rechnerunterstützung relevant.

3.2 Klassifizierung und Bewertung der Methoden und Systeme

	Regeln für den Entwurf	Mittel für die Darstellung	automatische Mittel zur Analyse
Entwurfsmethoden	JDM	+	-
	SD	+++	-
Darst.methoden mit autom. Analysen	SRI*	++	++
	HOS*	+++	+++
Darstellungsmeth.	MIL 75	+	++*
	PDL	+	++
	PSL/PSA	+	+++
	SEF*	+	+++
	SREP	+	+++
	SSES	+	+++*
Darstellungsmeth.	HIPO	+	+*
	LOGOS	+	+
	MASCOT	+	+
	SADT	+	-
	+ schwache Regeln (top-down)	+ lose Regeln	- nicht enthalten
	++ Einschränkungen	++ Sprache oder Diagramme	+ nur zur Speicherung
	+++ strenge Regeln	+++ masch. verb. Sprache	++ Speicherung und einfache Analysen
			+++ Speicherung und komplexe Analysen

Tabelle 3.2.a

*in der Entwicklung

	Überblick-Information	Management-Information	automatische Dokumentation	leichte Verwendbarkeit	graphische Darstellung	Verständlichkeit	für Echtzeit-Probleme geeignet	die Freiheit des Entwurfs erhaltend	Ablaufstrukturen darstellbar	Datenstrukturen darstellbar	Konsistenzprüfungen	Report-Erzeugung	Simulationen	Erweiterbarkeit	Portabilität
JDM	*			*	*	**	*	*	**	*					
SD	**			*	**	**	**	*	**						
SRI			**	*		*		**	*	**	**	*			
HOS	**		**		*	*	***	**	**	*	**	*			
MIL 75	**		*	**		**		*	**	*	**	*	*		
PDL	***	*	**	***	*	***	**	*	***	**	**	*		*	**
PSA/PSL	**	*	***	**	*	**	*	**	*	***	**	**		*	**
SEF															
SPEP	*	**	***	**	**	**	***	**	**	***	**	***	**	***	*
SSES	**		***	**		**	*	*	*	*	**	*			
HIPO	***		*	***	***	**	*	*	*	*		*			
LOGOS					***	*	***		***		*		*		
MASCOT	***			*	***	**	***	**	**						
SADT	***			**	***	**	*	**	*	*					

in Planung

Bewertung: *** starke
 ** mittlere
 * schwache
 keine Unterstützung

Tabelle 3.2.b

	Anwendungsgebiet für das System erstellt wurde	formales Darstellungsmittel	Stand der Entwicklung
HOS	große Systeme mit Parallelarbeit	HOS-Meta-Language	in Entwicklung
MIL 75	große Softwaresysteme	MIL 75	nicht bekannt
PDL	kommerzielle Softwaresysteme	PDL	im Einsatz
PSL/PSA	Informationssysteme	PSL	im Einsatz
SEF	große Softwaresysteme	SDL	in Entwicklung
SPEP	Raketenabwehr-System	R-Netze, PSL	in der Erprobung
SSES	große Softwaresysteme	SSL	in Entwicklung
HIPO	kommerzieller Einsatz	hierarchy- and IPO-charts	im Einsatz
LOGOS	Hard-/Software-Entwurf (maschinennah)	Datenfluß- und Ablaufdiagramme	im Einsatz
MASCOT	Radar-Systeme	Schemata aus Activities, Pools und Channels	im Einsatz
SADT	kommerzieller Einsatz	Actigramme und Datagramme	im Einsatz

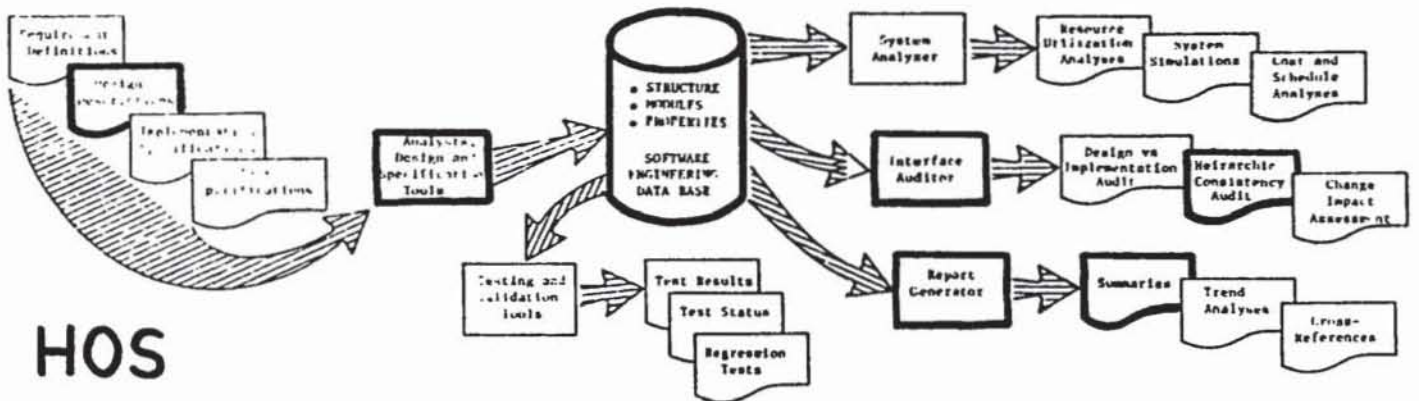
manuell

Tabelle 3.2.c

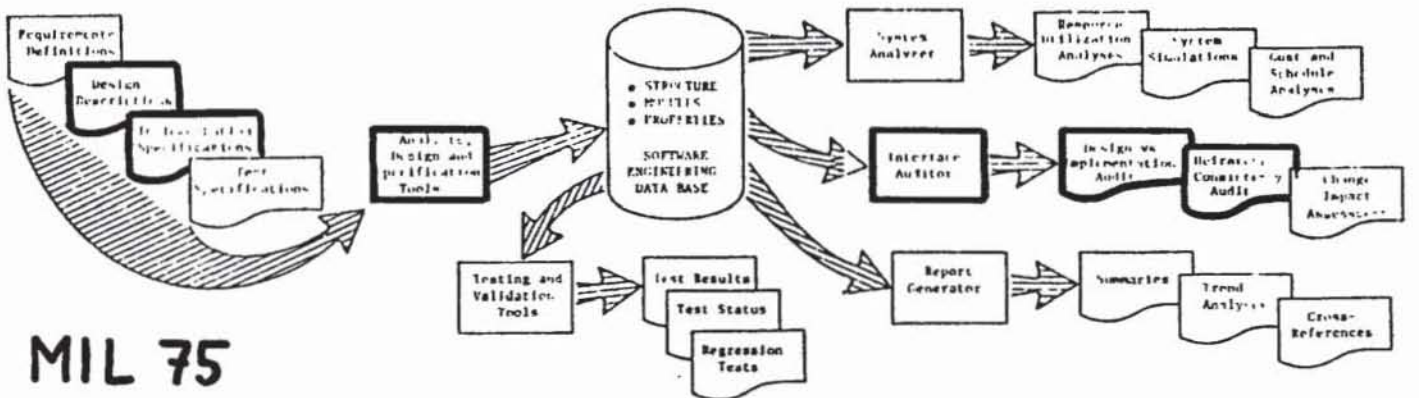
3.4. Die Möglichkeiten der automatischen Systeme

Die Fähigkeiten des geplanten SEF-Systems umfassen alle wesentlichen Möglichkeiten bestehender automatischer Systeme. Daher kann SEF zur Grundlage eines Vergleichs gemacht werden. Nachfolgend sind im SEF-Schema die in den einzelnen Systemen realisierten Teile hervorgehoben.

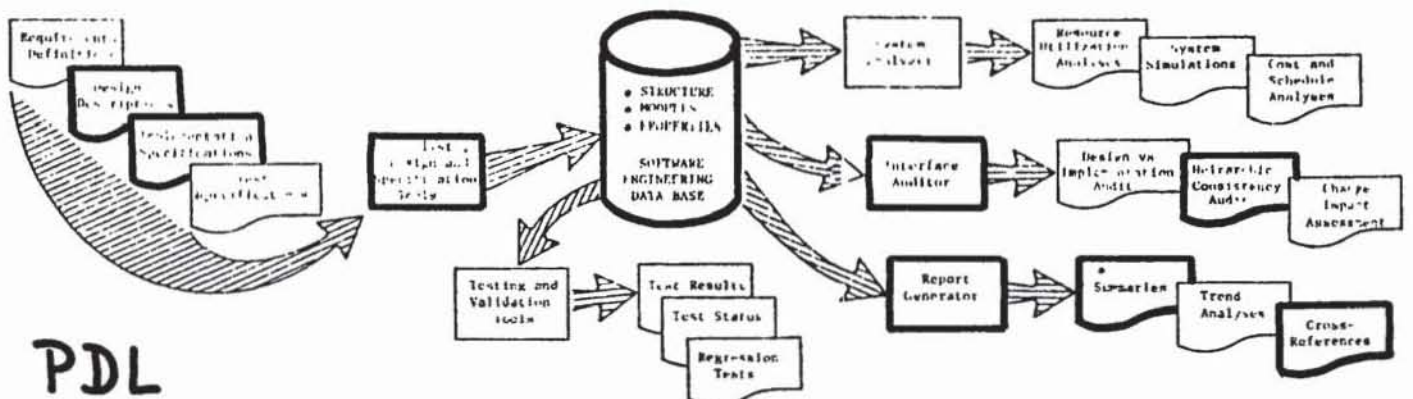
Selbstverständlich sind die Systeme nur bedingt vergleichbar.



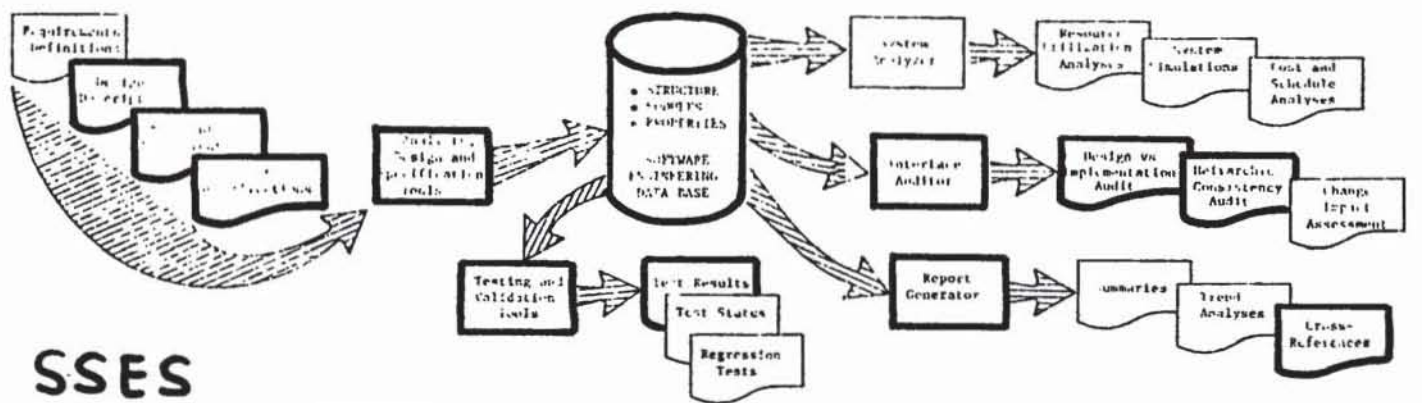
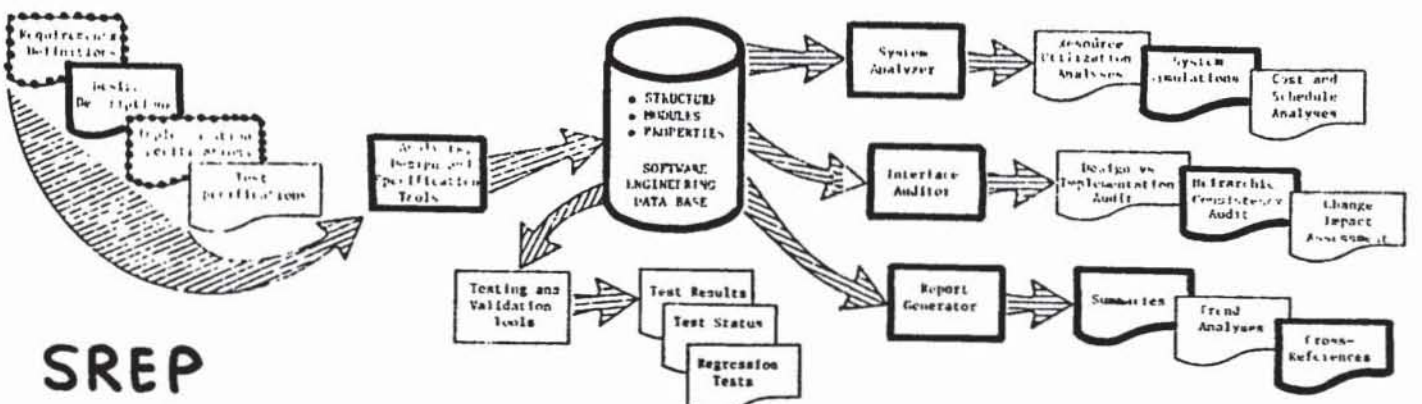
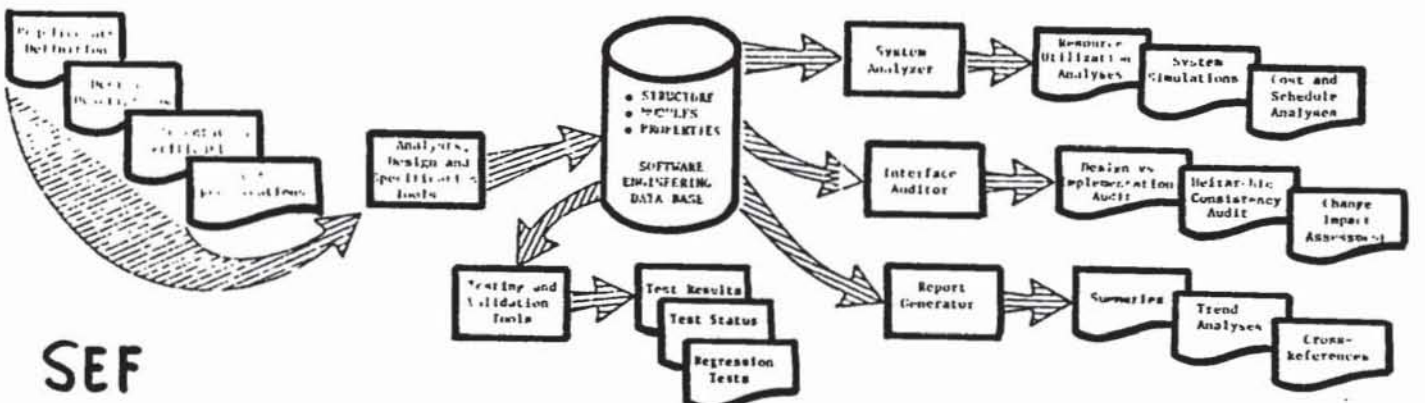
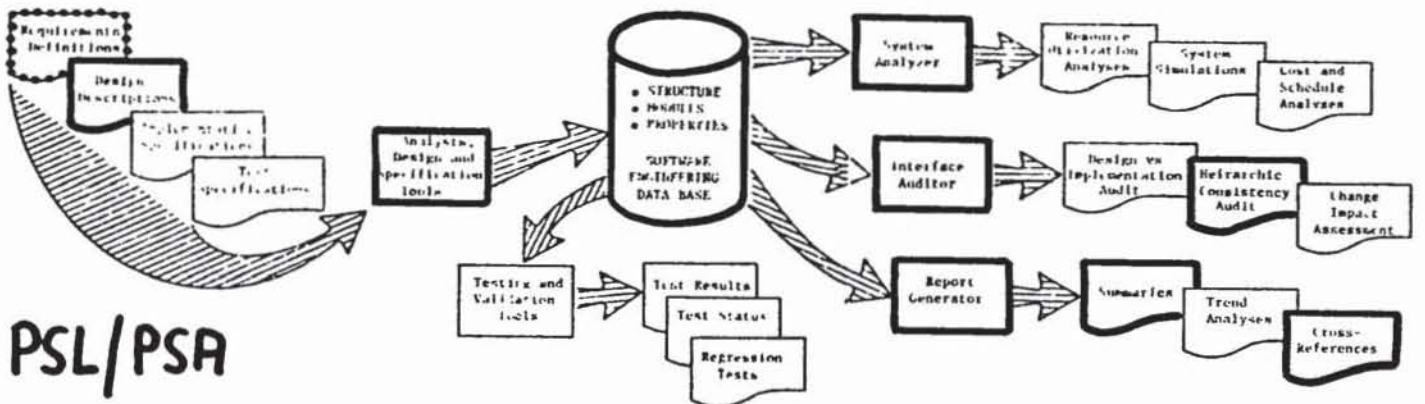
HOS



MIL 75



PDL

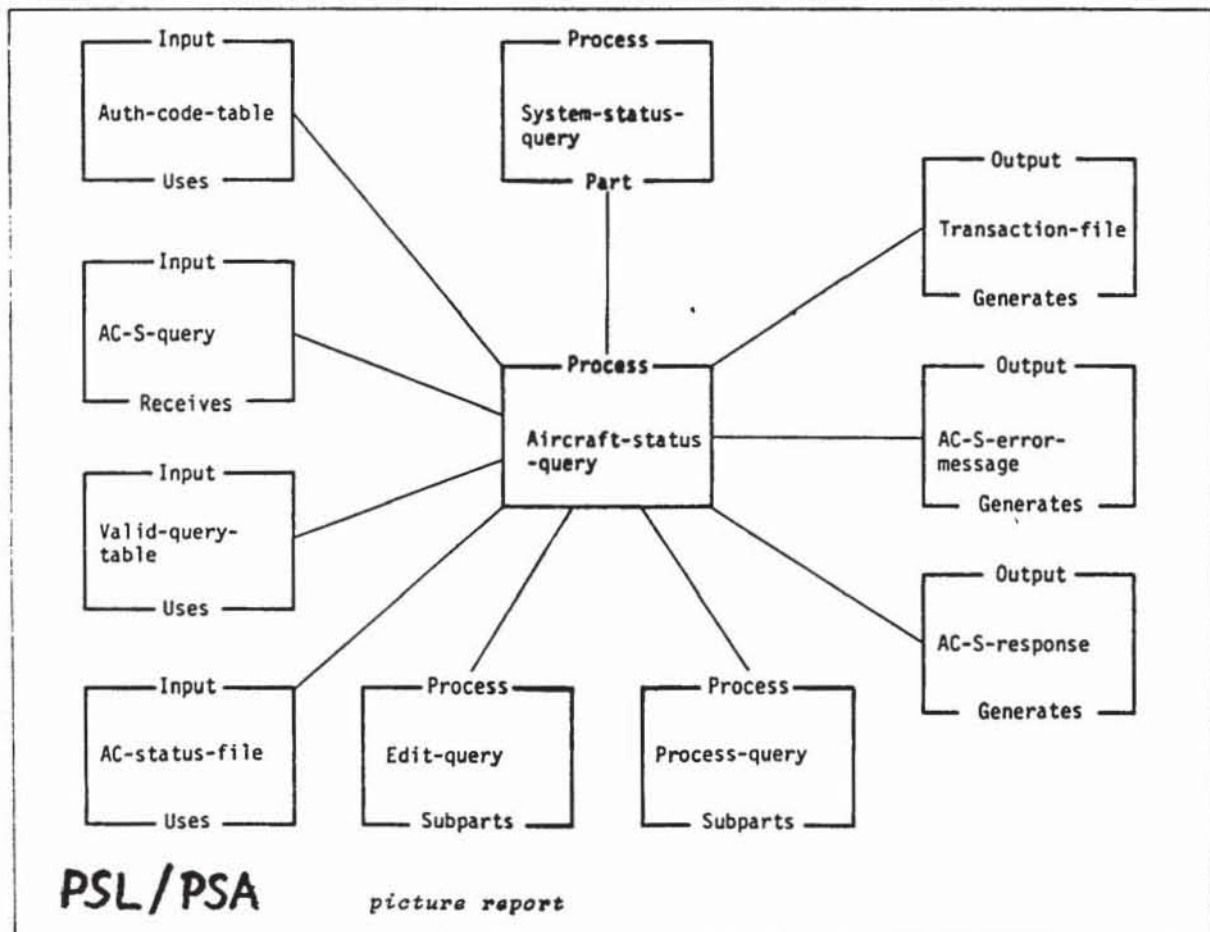


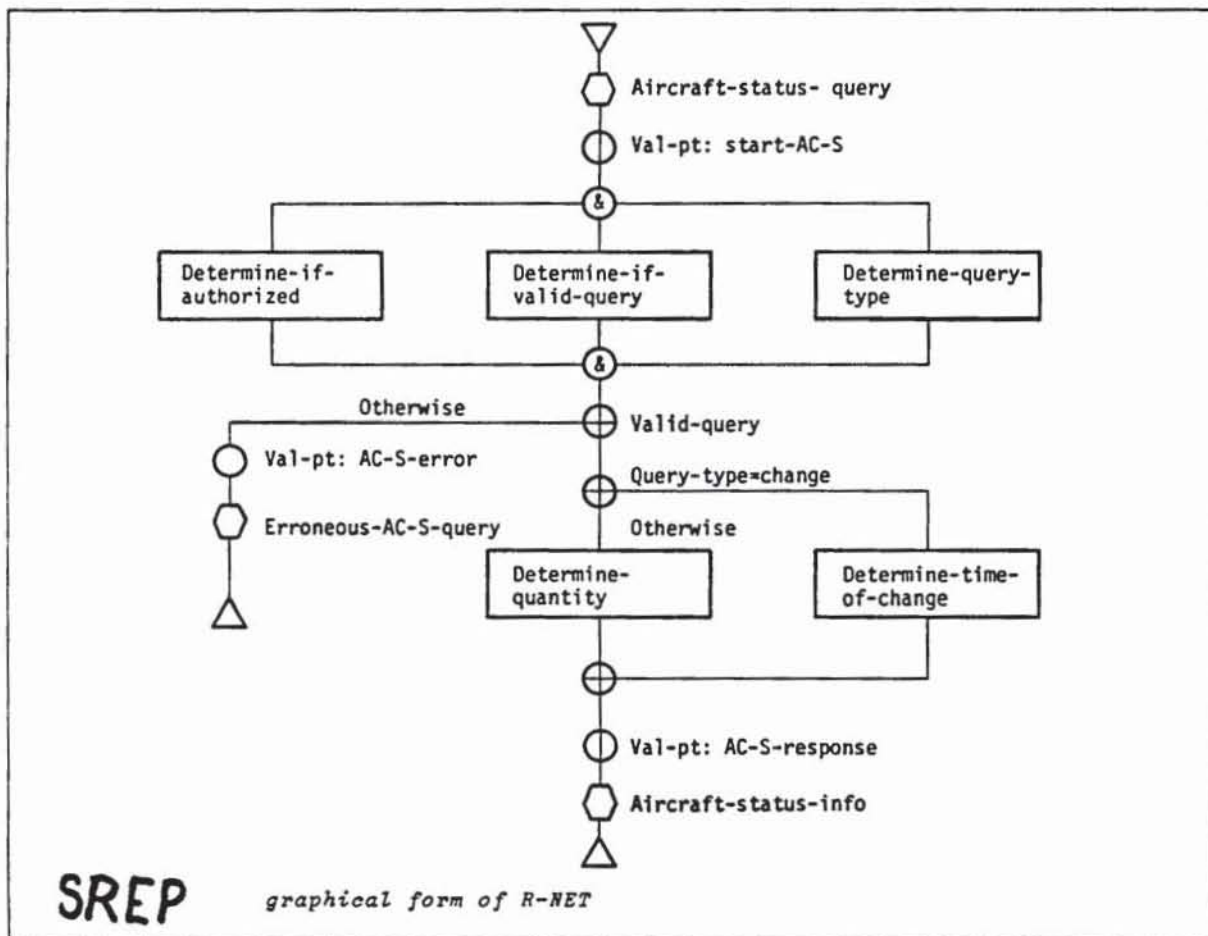
3.5 Vergleichende Beschreibung eines kleinen Entwurfs-Beispiels /2/

Ein Teil eines kleinen Entwurfs-Beispiels (Abfrage des Flugzeugzustandes) wird in PSL/PSA, SREP, HIPO, SD und PDL beschrieben.

Process:	Aircraft-status;
Synonyms are:	AC-S;
Attributes are:	AC-S-response-time 5 sec; AC-S-error-time 1 sec;
Part of:	System status;
Subparts are:	Edit-query; Process-query
Receives:	AC-S query;
Generates:	Transaction-file; AC-S-response; AC-S-error-message;
Uses:	AC-status-file; Auth-code-table; Valid-query-table;
Triggered by:	Aircraft-status-query;
Responsible problem definer:	B E Boehm
Security:	Unclassified;

PSL/PSA *formatted problem statement*

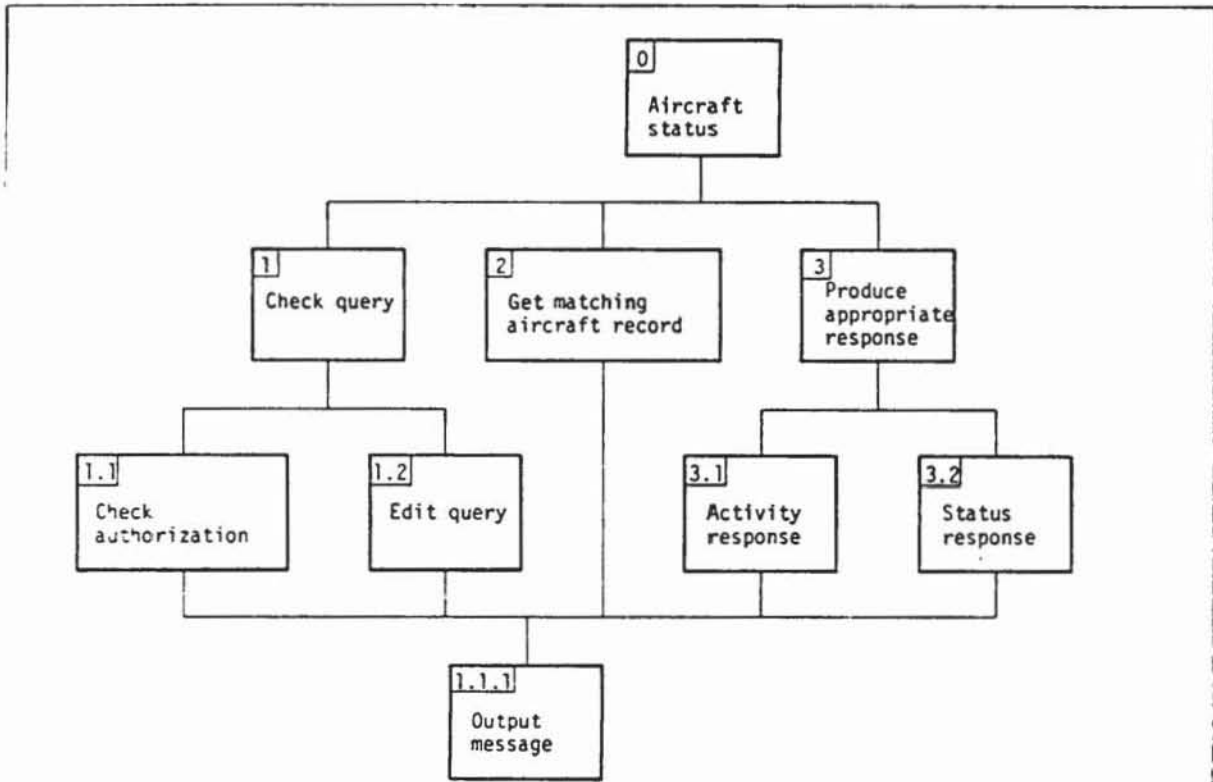




```

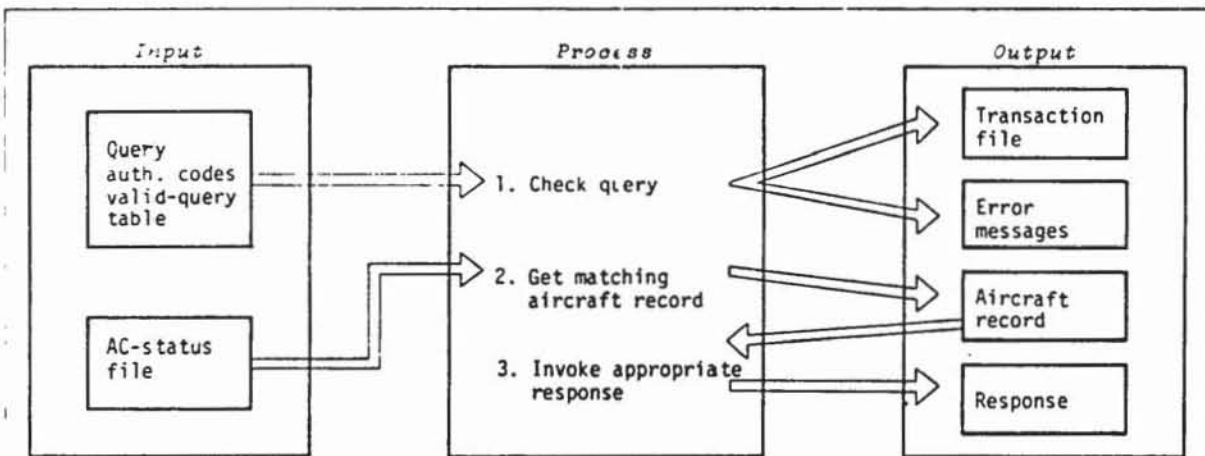
R-NET: Aircraft-status
Abbreviated by: AC-S
Entered by: "B W Boehm"
Enabled by:
  Input-interface: aircraft-status-query
Structure:
  Input-interface: aircraft-status-query
  Validation point: start-AC-S
  Do alpha: determine-if-authorized and
    Alpha: determine-if-valid-query and
    Alpha: determine-query-type
  End
  Do (valid-query)
    Do (query-type = change)
      Alpha: determine-time-of-change
    Otherwise
      Alpha: determine-quantity
    End
  Validation point: AC-S-response
  Output-interface: aircraft-status-info
  Terminate
  Otherwise
    Validation point: AC-S-error
    Output-interface: erroneous-AC-S-query
    Terminate
  End
End
  
```

SREP *text form of R-NET*



HIPO

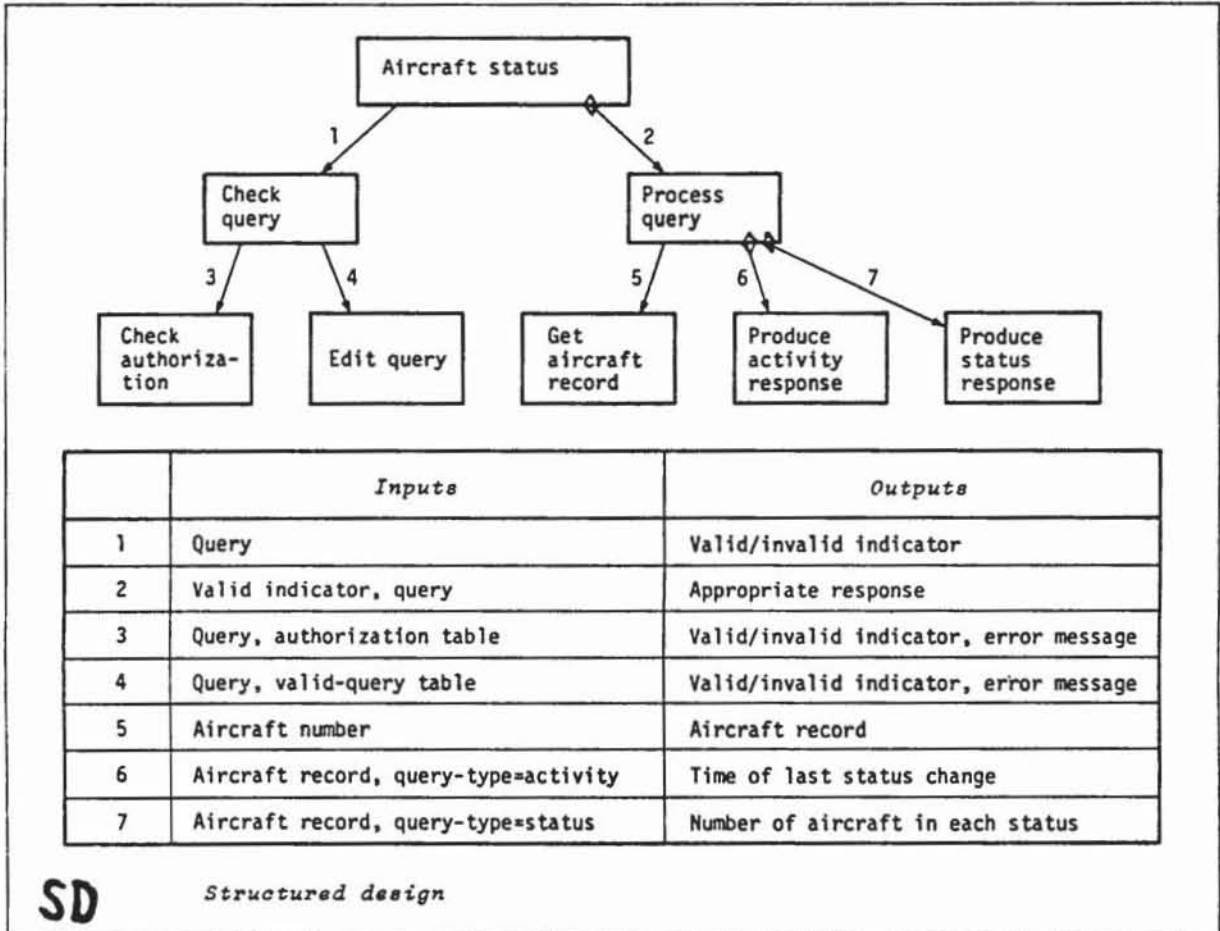
hierarchy chart



<i>Extended description</i>	<i>Chart</i>
1. Read query, output query, check authorization, edit for query type, aircraft number	1
2. Obtain matching aircraft record	2
3. Examine query type status: response shows number of A/C in each status change: response shows time of last status change	3

HIPO

input-process-output chart



```

Main
  Do while more queries
    Call process
  Enddo

Process
  Read query
  Output query
  If authorization OK
    If matching aircraft number
      Case (query-type = )
        Status:      output quantity in each status
        Change:      output time of last status change
        Otherwise:   output "invalid query type"
      Endcase
    Else output "invalid A/C number"
    Else output "invalid authorization code"
  Endif

```

PDL *Program design Language*

3.6 Literatur

Die Literatur zu den einzelnen Systemen ist in den betreffenden Abschnitten angegeben. Die nachfolgend genannten Artikel befassen sich mit mehreren Systemen. Hershey gibt eine nicht ganz aktuelle, aber breite Bibliographie zu sehr vielen Systemen.

/1/ Hershey, E.A.

A Survey of System Design Aids

INFOTECH: Software, London, November 1975

/2/ Boehm, B.W.

Software Requirements and Design Aids

INFOTECH: Reliable Software, London, März 1977

/3/ McGowan, C.L.; Kelly, J.R.

A review of Decomposition and Design Methodologies

INFOTECH: Reliable Software, London, März 1977