KERNFORSCHUNGSZENTRUM KARLSRUHE

Institut für Datenverarbeitung in der Technik

KfK 3060

Zur Erstellung der Spezifikation von Prozeßrechner-Software

Jochen Ludewig

von der Fakultät für Mathematik
der Technischen Universität München
genehmigte Dissertation

Kernforschungszentrum Karlsruhe GmbH, Karlsruhe

Kernforschungszentrum Karlsruhe GmbH
ISSN 0303-4003

Zusammenfassung

Gegenstand dieser Arbeit ist ein System zur Entwicklung von Prozeßrechner-Software, das den Übergang von einer informalen Aufgabenstellung auf eine formale Spezifikation und die damit verbundene Strukturierung der Programme unterstützen und so einen Beitrag zur Zuverlässigkeit der Software leisten soll.
Nach einigen Vorüberlegungen (2) werden die Konzepte entwickelt für das Begriffssystem, das den Kern bildet (3.1), für die Spezifikationssprache (3.2), für das Werkzeug zur Verarbeitung, Speicherung und Auswertung der Spezifikation (3.3) und für das Verfahren, nach dem das System angewendet werden kann (3.4). Die Präzisierung der Sprache und die Implementierung des Werkzeugs werden beschrieben (4). Für den Übergang von der Spezifikation zum Programm wird ein Ansatz dargelegt (5). Es folgt ein größeres Beispiel (6). Die vollständige Grammatik (7), Literaturverzeichnis (8) und eine Liste der Abkürzungen (9) stehen am Schluß.

On the development of specifications for process control software

This is a report on a system for the development of process-control software. The system supports the transition from an informal description of a problem to a formal specification. That includes the structuring process, which has a strong influence on software reliability.

Based on some preliminary discussion (2), a basic set of constructs is defined (3.1). The specification language (3.2) and a tool to process, store, and evaluate specifications (3.3) are described, and it is shown how to use the system (3.4). The language definition and the implementation of the tool are presented (4). For the transition from specification to actual code, a technique is outlined (5). Finally, an example is given (6). The complete syntax (7), the references (8) and a list of abbreviations (9) follow.

Inhalt
======

1. Thema und Zielsetzung

In den letzten Jahren entstand - vor allem wegen der hohen Programmkosten und wegen der großen Probleme, die immer wieder bei Software-Projekten aufgetreten sind (Boehm, 1973; Yourdon, 1972) - ein wachsendes Interesse an a l l e n Phasen der Programm-Entstehung, nachdem früher die Codierung (und damit die Programmiersprachen) ganz im Mittelpunkt gestanden hatte (Teichroew, 1974). Dabei wurde offensichtlich, daß es für die übrigen Phasen nur wenige Mittel und Methoden gibt und daß diese meist unverbunden nebeneinanderstehen, also weder begrifflich noch syntaktisch kompatible Schnittstellen haben, und in ihrer Vielfalt unüberschaubar sind (Teichroew, Hershey, Yamamoto, 1977; Ramamoorthy, So, 1977).

Die naheliegende Konsequenz war die Entwicklung i n t e g r i e r t e r Systeme für einen möglichst großen Teil des Programm-Lebenslaufs (life-cycle), wie sie z.B. das Ziel des ISDOS-Projekts (Teichroew, Sayani, 1971) und der "Software Engineering Facility" (Irvine, Brackett, 1977) ist.

Diese Arbeit befaßt sich mit Methoden und Hilfsmitteln der Programmentwicklung für Prozeßrechner. Sie umfaßt die Konzeption eines Systems, mit dessen Hilfe Spezifikationen allgemeiner Anwendungsprogramme für Prozeßrechner erstellt und in eine Programmiersprache umgesetzt werden können, und die teilweise Implementierung der zu diesem System erforderlichen Programme.

Schwerpunkte der Arbeit sind

- eine Klärung des Spezifikationsbegriffs und einiger verwandter Begriffe,
- die Entwicklung von Konzepten für eine Spezifikationssprache,
- die vollständige formale Definition der Spezifikationssprache,
- der Brückenschlag zu Programmiersprachen und Betriebssystem-Funktionen.

Die Grundaufgabe des Software-Engineering besteht darin, auf effiziente Weise Programme herzustellen, die den Erwartungen der Auftraggeber entsprechen, und diese Programme veränderten Anforderungen anzupassen (Bauer, 1972; Boehm, 1976).
Das Requirement-Engineering als Teilgebiet des Software-Engineering behandelt die Aufgabe, die Anforderungen in einer für die weitere Programmentwicklung geeigneten Form zu fixieren und dabei Mängel dieser Anforderungen erkennbar zu machen. Ein Spezifikationssystem umfaßt also sowohl konstruktive als auch analytische Ansätze, um Diskrepanzen zwischen dem Produkt und den daran geknüpften Erwartungen zu verhindern. Zusammen mit anderen Systemen, die den Übergang von einer - nicht notwendig operationellen - Programmiersprache auf ausführbaren, effizienten Code unterstützen und kontrollieren (siehe 5.10), kann es Komponente eines integrierten Entwicklungssystems für Prozeßrechner-Software sein, das den gesamten Lebenslauf des Programms abdeckt.

Generierbare Programme, die ohne eigentlichen Entwurfsvorgang aus einem Baukasten erzeugt werden (Schuchmann, 1977), sind nicht Gegenstand der Arbeit. Auch hat sie nicht das Ziel, die Herstellung extrem effizienter Programme speziell zu unterstützen.

Insgesamt soll die Arbeit also einen Beitrag dazu liefern, daß auch im Bereich der Prozeßrechner-Programmierung eine Systematik wie in anderen Ingenieur-Disziplinen entsteht, die eine Voraussetzung für den Übergang von der "Art of Computer Programming" zum "Software Engineering" ist.

2. Vorüberlegungen

2.1 Begriffe und Definitionen

Viele für diese Arbeit wichtige Begriffe sind in DIN-Normen definiert, insbesondere in DIN 66 201 ("Prozeßrechensysteme, Begriffe"), ferner in DIN 44 300 ("Informationsverarbeitung, Begriffe") und DIN 66 230 ("Dokumentation"). Soweit nichts anderes gesagt ist, werden alle dort aufgeführten Begriffe normgerecht verwendet.

"Prozeßrechner" wird hier abweichend von DIN 66 201, 6.1, nicht auf Baueinheiten beschränkt, sondern wie "Prozeßrechensystem" (ibid., 4.2) verwendet, also auf die Funktionseinheit bezogen.

Nachfolgend werden einige Begriffe diskutiert, deren Bedeutung in der Literatur schwankend ist. Auf den Spezifikationsbegriff wird in 2.2 eingegangen.

2.1.1 Programme und Programmiersprachen

Das Wort "Programm" wird entsprechend DIN 44 300 gebraucht ("eine in einer beliebigen Sprache abgefaßte vollständige Anweisung ('statement') zur Lösung einer Aufgabe mittels einer digitalen Rechenanlage"). "Software" hat dieselbe Bedeutung.

In dieser Arbeit ist von Prozeßrechner-Anwendungsprogrammen die Rede, also von Programmen,

- die für Prozeßrechner bestimmt sind (unabhängig davon, ob sie mit Hilfe anderer Rechner entwickelt werden). Brinch-Hansen (1978, S.934) zählt die Charakteristika von Prozeßrechner-Programmen auf.

- die anwendungsspezifische Probleme lösen. Dabei wird angenommen, daß die für einen Prozeßrechner-Typ stets wiederkehrenden, von der speziellen Anwendung unabhängigen Aufgaben bereits durch sogenannte Systemprogramme, z.B. Betriebssysteme, Übersetzer usw., gelöst sind.

Wo nichts anderes gesagt ist, wird "Programm" stets in diesem speziellen Sinn gebraucht.

Eine Programmiersprache ist eine formale Sprache zur Definition der Funktionen eines Programms. Ein Programm ist operationell, wenn eine (i.a. virtuelle) Maschine zu seiner Ausführung verfügbar ist. Die Sprachelemente, aus denen es aufgebaut ist, werden ebenfalls operationell genannt.

2.1.2 Methoden und Hilfsmittel

Eine Methode ist ein System von Regeln, deren Anwendung zur Lösung eines Problems beiträgt. Im Extremfall, wenn schon ihre mechanische Anwendung allein zur Lösung führt, liegt bereits ein Algorithmus vor.

Ein (Hilfs-) Mittel ist ein Werkzeug, daß die Durchführung bestimmter Arbeiten unterstützt, u.U. praktisch erst ermöglicht, z.B. das Programm zur Lösung des Vier-Farben-Problems (Appel, Haken, 1977).

2.1.3 Hierarchien

Im Software-Engineering spielen Hierarchien eine wesentliche Rolle. Der Grund dafür liegt in der großen Vereinfachung der Systembetrachtung, die mit der Einführung einer sinnvollen Hierarchie verbunden ist. Diesen Punkt hat Simon (1962) aus der Sicht der Systemtheorie diskutiert. Speziell im Hinblick auf Programm-Strukturen, wie sie die Moduln, Daten und Prozeduren bilden (Parnas, 1974), werden sie von Bauer (1972), Goos (1973) und Dijkstra (1971) besprochen.
Vielfach wird der Begriff der Hierarchie nur für Bäume verwendet; hier werden auch andere Halbordnungen als hierarchische Systeme bezeichnet.

Hat ein System (oder Subsystem) S Baumstruktur, so werden hier folgende Bezeichnungen verwendet:

S ist atomar (wenn es ein Blatt ist) oder zerfällt in n Subsysteme S1 bis Sn (die Unterbäume) und den Kopf oder Vater (die Wurzel).

Der Kopf enthält die Struktur-Information, die die Subsysteme verbindet (vgl. Simon, 1962, S.468: "The whole is more than the sum of the parts, ..."). Er repräsentiert das gesamte System.

Der Kopf hat nicht notwendig das Wesen eines "boss" (ibid.). Bei der Einführung des "secretary" läßt es Dijkstra (1971, S.135) ausdrücklich offen, ob dieser über- oder untergeordnet ist.

2.1.4 Prozedurale und nicht-prozedurale Programm-Beschreibung

In Spezifikationssprachen wird meist eine nicht-prozedurale Darstellung angestrebt, d.h. eine Beschreibung, die die Reihenfolge der Operationen offenläßt (Leavenworth, Sammet, 1974). Die nachfolgenden Überlegungen zeigen aber, daß die Festlegung von Sequenzen aus verschiedenen, teilweise auch in der Spezifikation gültigen Gründen notwendig ist.

Die Operationen eines Programms sind dynamisch verknüpft, d.h. eine Operation folgt nur auf bestimmte Operationen und ist ihrerseits von bestimmten anderen gefolgt. Diese Beziehungen lassen sich mit der Relation "läuft ab vor" als Halbordnung darstellen.

Für die Beziehung "A läuft ab vor B" kann es verschiedene Gründe geben:

a) A und B erbringen Leistungen, die laut Spezifikation in dieser Reihenfolge gefordert sind.

b) A liefert als Ergebnis direkt oder indirekt eine Eingabe für B.

c) Die Reihenfolge "A vor B" trägt zur Erhöhung einer Programmqualität bei, z.B. der Effizienz, wenn A noch im Hauptspeicher steht, aber von B verdrängt wird.

d) Die Reihenfolge ist willkürlich vom Programmierer oder vom Compiler gewählt, weil die virtuelle oder die reale Maschine sequentiellen Code verlangt.

Der Grund kann also in der Spezifikation, im Datenfluß, im Streben nach Programmqualitäten oder in der verwendeten Maschine liegen.

Offenbar stehen diese vier Gruppen hinsichtlich ihrer Relevanz auf verschiedenen Stufen: (d) ist sowohl im Sinne der Funktion als auch der Qualität irrelevant, (c) ist es hinsichtlich der Funktion; (b) ist entwurfsbedingt, (a) vorgegeben.

Also nimmt mit fallendem Abstraktionsgrad der Beschreibung die Zahl der Verknüpfungen "läuft ab vor" zu, die Halbordnung geht im Extremfall in eine strenge Ordnung über. Aber schon in der Spezifikation sind gewisse Festlegungen der Sequenz unvermeidlich.

2.2 Die Spezifikation

Dieser Abschnitt hat den Zweck, Wesen, Bestandteile und Darstellungsmöglichkeiten der Spezifikation und die Konsequenzen für ein Spezifikationssystem zu klären.

2.2.1 Der System-Lebenslauf

Die beiden ersten Phasen einer Programmentwicklung nach der ersten Idee sind die Analyse und die Entwurfsphase (preliminary design). Ihnen folgen Detaillierung, Codierung, Test, Installation und Betrieb. Diese Einteilung ist in der Literatur im wesentlichen einheitlich (Hice, Turner, Cashwell, 1974; Teichroew, 1974; DeWolf, 1977, S.18).

Gegenstand dieser Arbeit sind Analyse und Entwurfsphase. Wie alle andern Phasen sind sie zwingend notwendig; daran ändert sich auch nichts, wenn mit der Codierung begonnen wird. Nur sind dann die Anfangsphasen im Kopf des Programmierers durchgeführt worden, also nicht dokumentiert, nicht prüfbar und unkontrollierten Änderungen unterworfen. Entsprechend werden viele Fehler gemacht, die erst viel später, d.h. beim Test, im Einsatz oder nie entdeckt und behoben werden können. Darin liegt der Grund für die besondere Aufmerksamkeit, die die Anfangsphasen in den letzten Jahren gefunden haben (Boehm, 1974, S.192).

Die einzelnen Phasen des Programm-Lebenslaufs können nicht streng nacheinander durchlaufen werden; Überschneidungen und Iterationen sind unvermeidbar, die im ungünstigsten Fall eine Wiederholung aller Phasen erfordern, wenn nämlich erst im Betrieb ein Fehler der Analyse bemerkt wird (vgl. Scheidig, 1975, 3.3). Nachfolgend soll gezeigt werden, daß speziell Analyse und Entwurfsphase notwendig verschränkt sind.

2.2.1.1 Analyse und Spezifikation

Nach Hice, Turner, Cashwell (1974) dient die Analyse

- der Umsetzung der ersten Idee und der allgemeinen Angaben über das System in exakte Aussagen über die Ziele,

- der Beantwortung der Fragen zur Durchführbarkeit und Zweckmäßigkeit der Entwicklung, bevor erheblicher Aufwand investiert wird.

Aussagen zum zweiten Punkt sind nur in Extremfällen ohne wenigstens teilweise Behandlung des ersten möglich. Außerdem erfordern sie betriebswirtschaftliche Überlegungen, die nicht Gegenstand dieser Arbeit sind.

Damit bleibt als Zweck der Analyse die Feststellung der Zielsetzung für das geplante System. Das Dokument dieser Phase ist die Spezifikation; diese Bezeichnung wird hier synonym mit "Anforderungsspezifikation" ("requirement specification") verwendet, also wie bei Parnas (1977) für die "genaue Aufstellung aller Anforderungen an ein Produkt". Die Spezifikation gibt Antwort auf die Frage

Was soll mit welchem Aufwand bis wann realisiert sein?

Diese Arbeit konzentriert sich auf das 'was'. Die Fragen des Aufwands und der Terminsetzung, die in das Gebiet des Projektmanagements fallen, werden nicht untersucht. Die Antwort auf die Frage 'was' kann man als die Produkt-Spezifikation bezeichnen (Gegensatz: Spezifikation der Produkt-Entwicklung = Management-Spezifikation). Nachfolgend wird "Spezifikation" in diesem eingeschränkten Sinne gebraucht.

2.2.1.2 Entwurfsphase und Entwurf

In der Entwurfsphase wird den in der Spezifikation niedergelegten Anforderungen die Konzeption einer Problemlösung gegenübergestellt (Hice, Turner, Cashwell, 1974: "Übergang von den Ansätzen und Zielen zu einer vollständigen Systembeschreibung").
Diese Konzeption ist noch weitgehend unabhängig von den Randbedingungen der Implementierung wie Programmiersprachen und Hardwareeigenschaften. Entworfen werden l o g i s c h e Strukturen und Zusammenhänge, über die Repräsentation im Rechner wird erst bei der Detaillierung entschieden.

Das Ergebnis der Entwurfsphase ist der Entwurf; dieses Wort wird hier stets im Sinne von "Grobentwurf" gebraucht, also zur Bezeichnung der ersten Beschreibung einer Problemlösung.

Der Entwurf gibt an, wie die Aufgabe grundsätzlich bewältigt werden soll, welche Daten und Prozeduren konzeptionell für diese Lösung benötigt werden und wie diese Objekte strukturiert und verbunden sind. Erst später, im Zuge der Detaillierung, wird geklärt, wie sie praktisch realisiert werden können.

Beispiel:
Die Spezifikation verlange die Verarbeitung einer kontextfreien Sprache. Der Entwurf sieht dafür einen Keller vor. Bei der Detaillierung wird dieser Keller durch eine Liste realisiert, die bei der Codierung auf ein Feld der verwendeten Programmiersprache abgebildet wird.

2.2.2 Die Beziehungen zwischen Spezifikation und Entwurf

Geht man wie üblich von einer top-down-Entwicklung aus, so besteht das Entwerfen eines Programms darin, daß zu einer Systemspezifikation eine Menge von Subsystemspezifikationen gefunden wird. Die so entstehenden Subsysteme werden genauso behandelt, bis die Spezifikation der Zielmaschine erreicht ist; das Entwerfen erfolgt also rekursiv.
Die Spezifikation ist folglich nicht e i n Stadium der Programmdokumentation, sondern entsteht mit jedem Schritt des Entwurfs neu und bildet wieder die Grundlage des nächsten Entwurfsschritts (*). Diese Beziehung läßt sich durch folgendes Bild darstellen (Bild 2.1).

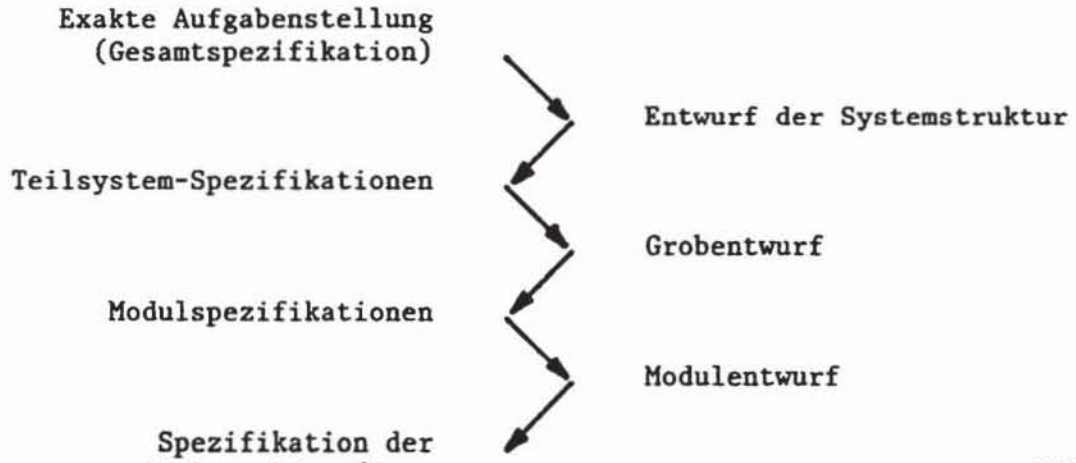


Bild 2.1
Die Programmentstehung

Aus diesem Grunde ist es sinnvoll, von einem Spezifikationsprozeß zu sprechen, der die Entwurfsschritte einschließt und in dessen Verlauf Spezifikationen entstehen, deren Detaillierungsgrad zunimmt.

Die Darstellung des Spezifikationsprozesses, wie sie oben gegeben wurde, ist eine in der Praxis nicht erreichbare Idealisierung, da sie das Vorhandensein einer vollständigen Spezifikation (siehe 2.2.4) v o r dem ersten Entwurfsschritt voraussetzt. Tatsächlich aber wird das Fehlen von Anforderungen zum Teil erst durch den Entwurf erkennbar, ja entsteht überhaupt erst durch den Entwurf. Ein anschauliches Beispiel dafür ist die Spezifikation eines Hauses: Der Kunde kann den Bodenbelag im Treppenhaus erst aussuchen, wenn die Entwurfsentscheidung zugunsten einer mehrstöckigen Bauweise gefallen ist. Für einzelne Fragen wäre noch denkbar, daß sie a priori, quasi "auf Verdacht", beantwortet werden, für alle ist dies aufgrund der hohen Zahl von Entwurfsalternativen nicht möglich.

---

* Ein interessanter Aspekt ergibt sich aus Simon (1962, S.479), wenn man für 'state description' Spezifikation setzt, für 'process description' Entwurf. Es zeigt sich, daß die oben beschriebene Beziehung keineswegs auf die Informatik oder den technisch-naturwissenschaftlichen Bereich beschränkt ist.

Daraus folgt, daß die Spezifikation im allgemeinen nicht vollständig bestimmt ist durch den vorhergehenden Entwurfsschritt, sondern daß zusätzliche Information vom Auftraggeber benötigt wird. Auch die Anforderungen werden also im Zuge der Entwicklung verfeinert, nicht nur im Sinne einer Zerteilung, sondern auch eines Ausmalens.

Die Notwendigkeit, beim Auftraggeber nachzufragen, läßt mit fortschreitender Detaillierung nach, denn die Auswirkungen der eventuell noch offenen Entscheidungen sind für den Auftraggeber immer weniger überschaubar und relevant, so daß er weder in der Lage noch bereit ist, sich damit zu befassen. Daher erreicht das Projekt ein Stadium, in dem die Anforderungen als stabil gelten können. Die Abstimmung zwischen Auftraggeber und Hersteller eines Programms erfolgt also nicht zu einem einzigen Zeit- oder Entwicklungs p u n k t , sondern während einer bestimmten P h a s e der Entwicklung. Der dabei beteiligte Hersteller braucht allerdings nicht identisch zu sein mit dem, der später die Ausführung der Arbeit übernimmt.

Bild 2.1 wird jetzt erweitert, so daß auch die schrittweise Feststellung der Anforderungen und die Verarbeitung freier Entscheidungen des Herstellers sichtbar wird. Die Entwurfsschritte drücken sich nur in den Verbindungen der Spezifikationsstadien aus.

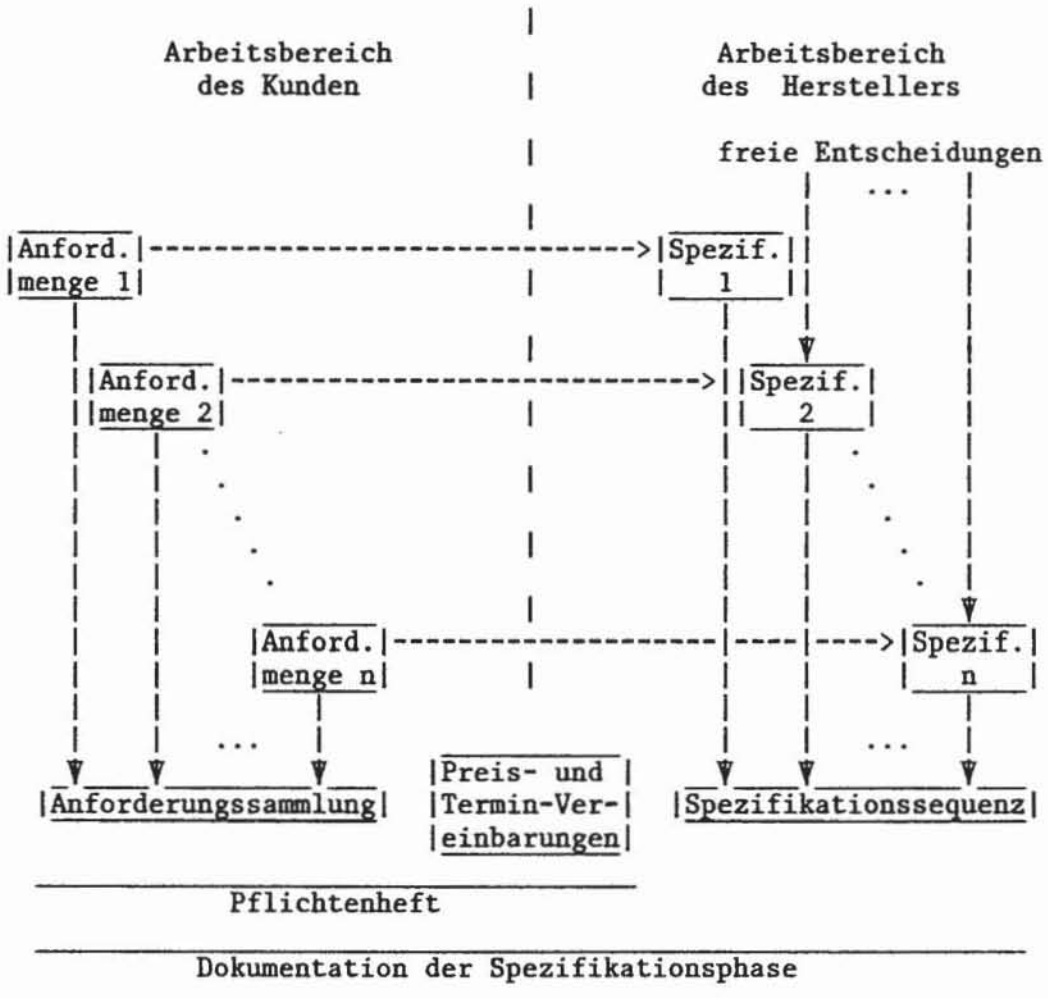


Bild 2.2 Der Spezifikationsprozeß

In Bild 2.2 ist angenommen, daß nach der Anforderungsmenge n keine weiteren Anforderungen mehr vom Kunden kommen. Aus der Spezifikation n kann dann durch weitere Verfeinerungsschritte der Code entwickelt werden.

In der Praxis der Softwareproduktion spielt das Pflichtenheft eine wichtige Rolle (2.3.1). Offenbar enthält es z u m i n d e s t die Anforderungssammlung sowie die Vereinbarungen über Preise und Termine. Nachdem festgestellt wurde, daß die Anforderungen zum Teil erst im Zuge des Spezifikationsprozesses notwendig werden, erscheint es aber fraglich, ob ein solches Pflichtenheft als Kontrakt zwischen dem Kunden und dem Hersteller ausreicht, denn sein Inhalt läßt sich ohne die Spezifikationssequenz nicht nachvollziehen. Daher sollte an die Stelle des Pflichtenheftes die Dokumentation der Spezifikationsphase treten. In dieser müssen die Anforderungen und ihre Beziehungen zur Spezifikationssequenz erkennbar sein.

2.2.3 Klassifizierung der Spezifikationsbestandteile

Ein Programm steht auf zwei Arten in Verbindung mit seiner Umgebung:

- Es wird eingesetzt und arbeitet zusammen mit anderen Programmen, Bedienern und technischen Prozessen (Einsatzaspekt).
- Es wird im Einsatz noch korrigiert und verändert. Dafür hat sich der Euphemismus "Wartung" durchgesetzt (Wartungsaspekt).

Die Produkt-Spezifikation (2.2.1.1) enthält die Anforderungen zu beiden Aspekten. Damit ergibt sich die folgende Gliederung der Spezifikation:

Bild 2.3
Gliederung der Spezifikation

Eine ähnliche Gliederung, bei der die einzelnen Teile in Beziehung zu ihren Einflüssen auf die Programmeigenschaften gesetzt sind, ist im Artikel von Boehm, Brown, Lipow (1976, S.595) enthalten.

2.2.3.1 Die Einsatzanforderungen

Die Entwicklung eines Programms ist im allgemeinen nicht Selbstzweck; vielmehr gibt es ein übergeordnetes Ziel für das System, das aus Programm, Hardware und Umgebung, also technischem Prozeß und Bedienung, entstehen soll. Da dieses Ziel normalerweise nicht auf einfache Weise erreicht werden kann, sondern nur durch ein kompliziertes Zusammenwirken der genannten Teile, ergeben sich für jedes Teil komplexe Unterziele. Die Art und Weise, wie diese ermittelt werden, hängt von der Entstehungsgeschichte des Gesamtsystems ab. Gewöhnlich sind einzelne Teile vorgegeben und können nicht mehr oder nur noch in geringem Maße modifiziert werden, z.B. der technische Prozeß, die Hardware oder Hardware plus Basissoftware einschließlich den Übersetzern.

Eine Spezifikation, die diese Ausgangssituation reflektiert, indem sie die Anforderungen an das Gesamtsystem nennt und die übrigen Komponenten beschreibt, wird komplementär genannt. Zur Implementierung müssen daraus die Anforderungen an das Programm ermittelt werden, es entsteht die direkte Spezifikation.

Das Gesamtsystem hat die folgende Struktur:

Die Verbindungen stehen für jede Art der Beeinflussung.

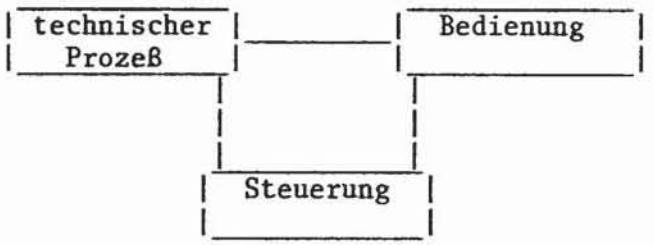


Bild 2.4 Umgebung der Steuerung

Die Steuerung zerfällt in Hardware und Software, letztere wieder in Basis- oder Systemsoftware und in Anwendungssoftware.

Hardware und Basissoftware können als virtuelle Maschine zusammengefaßt werden.

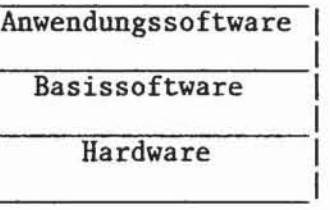


Bild 2.5 Aufbau der Steuerung

Sind auch Teile der Anwendungssoftware vorgegeben, so entsteht das nebenstehende, für die weiteren Betrachtungen zugrundegelegte Bild der Steuerung:

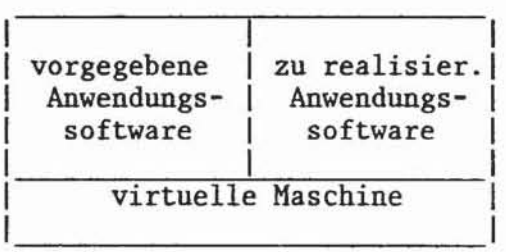


Bild 2.6 Umgebung der zu realisierenden Software

Die Einsatzanforderungen ergeben sich also aus den Verbindungen

- zur Basismaschine,
- zu anderen, kooperierenden Programmen,
- zum Bediener und zum technischen Prozeß.

Zu den Anforderungen gehören auch die Zusicherungen über das Verhalten der Umgebung, die diese Komponenten bilden.

2.2.3.2 Die Wartungsanforderungen

Die Anforderungen, die sich aus der Notwendigkeit der Programmwartung ergeben, lassen sich im allgemeinen nicht präzise vorhersehen. Daher werden nur gewünschte Programmeigenschaften (Qualitäten) spezifiziert, die erfahrungsgemäß die Wartung erleichtern.

Qualitäten, die für die Wartungsschnittstelle eine Rolle spielen, sind Modularität, Adaptierbarkeit, Portabilität, Kompatibilität. Um diese zu erhöhen, werden in der Praxis Regeln vorgegeben. Solche Konventionen betreffen z.B. die Auswahl zugelassener Programmiersprachen, die Modulgrößen, das Verbot gewisser Techniken wie selbstverändernder Programme, die Namensgebung und Form und Inhalt der Dokumentation.

2.2.4 Eigenschaften der Spezifikation

Folgende Eigenschaften sind für eine Produkt-Spezifikation anzustreben:

Sie soll i n h a l t l i c h
- zutreffend,
- vollständig und
- widerspruchsfrei sein.

Ihre D a r s t e l l u n g soll
- abstrakt,
- formal,
- präzise und
- verständlich sein

(vgl. Parnas, 1977).

Schließlich soll sie sequentiell erstellbar sein und den späteren Test des Systems vorbereiten, kurz
- entwickelbar und
- testorientiert sein.

Zur - hier nicht angestrebten - Minimalität siehe 2.2.5.

Zutreffend ist eine Spezifikation, wenn sie die Vorstellungen des Auftraggebers richtig wiedergibt. Da diese nicht objektivierbar sind, ist die Eigenschaft nicht prüfbar; sie kann aber durch das Entwicklungsverfahren der Spezifikation mehr oder minder gefördert werden.

Eine Spezifikation ist vollständig, wenn sie angibt,
- welche Einwirkungen der Umgebung auf das System zu erwarten sind und
- wie das System auf diese Einwirkungen reagieren muß.

Geht man davon aus, daß in allen Fällen, die von der Spezifikation nicht erfaßt sind, jedes Verhalten der Umgebung bzw. des Systems den Anforderungen entspricht, so ist die Spezifikation stets vollständig, aber in der Praxis kaum zutreffend.

Die Widerspruchsfreiheit bedarf keiner Erläuterung; es sei nur darauf hingewiesen, daß es sich um ein nicht entscheidbares Problem handelt.

Die Spezifikation ist abstrakt, wenn sie die Implementierung nicht vorwegnimmt. Das zu spezifizierende System soll also als "schwarzer Kasten" definiert werden. Durch Verwendung einer nicht-prozeduralen Sprache wird die Abstraktheit gefördert (siehe 2.1.4).

Eine formale Spezifikation kann maschinell verarbeitet und gespeichert werden, ihre Eigenschaften lassen sich formal überprüfen. Sie ist präzise, wenn die Semantik der Spezifikationssprache wohldefiniert ist. "Formal" und "präzise" haben also verschiedene Bedeutungen: z.B. ist eine Spezifikation in PSL (siehe 2.3.2) formal, aber nicht präzise, während eine wissenschaftliche Darstellung präzise sein kann, ohne formal zu sein. Der Umgangssprache fehlen beide Eigenschaften.

Da die Spezifikation eingeht in den Kontrakt zwischen Auftraggeber und Hersteller, ist die Verständlichkeit für beide Seiten von besonderer Bedeutung.

Die Spezifikation ist entwickelbar, wenn sie mit den zur Verfügung stehenden Mitteln sequentiell entwickelt werden kann, ausgehend von den vagen Ideen, mit denen jede Programmentwicklung in der Praxis beginnt. Dazu muß es in den frühen Phasen auch möglich sein, n a i v e Aussagen zu formulieren.
Auch für Entwickelbarkeit und Verständlichkeit gilt, daß die Begriffe zwar verwandt, aber nicht gleichbedeutend sind.

Eine Spezifikation ist testorientiert, wenn sie explizit angibt, welche Tests bei der Abnahme zweckmäßig sind. Diese Eigenschaft ist nicht notwendig, wo die Korrektheit auf andere Weise garantiert ist; sie ist illusorisch, wo die Aufschreibung der Testbedingungen oder der Test selbst einen unvertretbaren Aufwand erfordern würde, wie es z.B. bei den in 2.2.3.2 genannten Qualitäten der Fall ist.

Die genannten Ziele sind nicht völlig miteinander vereinbar (Liskov, Zilles, 1975). Herkömmliche Spezifikationen sind verständlich und - im Prinzip - vollständig, aber nicht abstrakt, formal und präzise. Parnas plädiert für diese fehlenden Eigenschaften (Parnas, 1977), kann aber mit seinem Verfahren nur relativ kleine Moduln spezifizieren, ist also weit von der Vollständigkeit entfernt. Außerdem sind seine Spezifikationen nicht leicht zu verstehen. Viele der modernen Ansätze liegen zwischen diesen beiden Polen (vgl. 2.3.2). Es ist also ein Kompromiß nötig.

2.2.5 Redundanz und Minimalität der Spezifikation

Eine minimale Spezifikation besteht nur aus Anforderungen an das Ein- und Ausgabeverhalten, enthält also keine Redundanz. Angaben über die Umgebung (Einbettung) und Ansätze zur Problemlösung gibt es in einer minimalen Spezifikation nicht. Eine solche Spezifikation läßt sich jedoch nur sehr schwer entwickeln, ändern und verstehen. In der Praxis enthält die Spezifikation daher sowohl Information über die Hintergründe der Anforderungen als auch Modelle des zu entwickelnden Systems. Dies entspricht auch der Regel, daß die Struktur der Lösung und die der Aufgabe übereinstimmen sollen (Stevens, Myers, Constantine, 1974, S.120; Jackson, 1975). Denn spätere Änderungen der Anforderungen gehen vom Problem aus.

Wo Daten gespeichert und verarbeitet werden, ist eine Spezifikation ohne Modell des Systems praktisch unmöglich. In der formalen Sprachdefinition im Anhang ist z.B. das Kontext-Attribut mit seinen Operationen ein solches Modell. Die Implementierung braucht zwar dem Modell nicht zu folgen (Parnas, 1977: "sample implementation"; Roubine, 1976), bleibt aber sicher nicht unbeeinflußt.

Eine unnötige Vorwegnahme von Entwurfsentscheidungen schränkt die Menge der möglichen Lösungen ein, schließt die optimale Lösung womöglich aus (Parnas, 1972, 1977; Liskov, Zilles, 1975). Sie ist weit verbreitet, vor allem, weil es nur wenige Spezifikationssprachen gibt und diese kaum bekannt sind (Uhrig, 1978, S.242).

Daher muß bei der Redundanz ein Kompromiß gesucht werden, der die Spezifikation entwickelbar und verständlich macht, ohne den Entwurf mehr als nötig vorwegzunehmen. Minimalität wird nicht angestrebt.

2.3 Der Stand der Technik

In diesem Abschnitt soll skizziert werden, wie heute in der Praxis Programme spezifiziert werden. Anschließend sind einige Systeme aufgezählt, die diese Arbeit beeinflußt haben.

2.3.1 Die Praxis der Spezifikation

Die Anwendung moderner Verfahren und Hilfsmittel für Spezifikation und Entwurf, wie sie in der neueren Literatur beschrieben sind, hat sich in der Praxis noch nicht durchgesetzt; es wird allenfalls experimentiert. ("Praxis" bezieht sich hier stets auf die Situation bei den Rechner-Herstellern, Software-Häusern und Forschungseinrichtungen der Bundesrepublik Deutschland; in den USA ist die Entwicklung etwas weiter.)

Über das übliche Vorgehen lassen sich nur sehr schwer Aussagen machen. Die Literatur dazu ist äußerst spärlich (z.B. mbp, 1978), meist sind die Praktiker nur im direkten Gespräch bereit, Angaben zu diesem Punkt zu machen. Grund dafür ist anscheinend weniger der Wunsch, Firmengeheimnisse zu schützen, als vielmehr das wachsende Bewußtsein, nicht mehr auf dem neuesten Stand zu sein.

Bei allen Unterschieden lassen sich folgende Merkmale festhalten:

a) Zentrales, meist einziges Dokument der Spezifikations- und Entwurfsphasen ist das Pflichtenheft, auch Lastenheft genannt (siehe 2.2.2). Darüber, welchen Inhalt ein Pflichtenheft haben muß oder nicht haben sollte, herrscht keine Übereinstimmung. Häufig sind bereits Details der Implementierung vorgeschrieben.

   Liegt die Programmentwicklung in der Hand einer einzigen Person, so bleibt die Spezifikation oft ungeschrieben; stattdessen wird im nachhinein in Form verschiedener Handbücher etwas Vergleichbares erstellt (d.h. statt der Einhaltung von Anforderungen die Beschreibung eines Produkts, also eine Umkehrung der Beziehung und damit eine starke Tendenz, nicht das Notwendige herzustellen, sondern das Gewachsene anzubieten, vgl. Heninger, 1980, IV.1).

b) Die natürliche Sprache ist meist das e i n z i g e Sprachmittel zur Spezifikation; nur selten wird sie ergänzt durch formale Sprachen. Unter diesen sind vor allem Entscheidungstabellen und Petri-Netze zu nennen (z.B. Gottschalk, 1978). Graphische Darstellungen wie HIPO oder SADT werden in steigendem Maße eingesetzt. Eine formale Spezifikation wird nur dann verwendet, wenn sie vom Problem her vorgegeben ist (z.B. bei der Implementierung eines Compilers, wo die Sprache formal definiert ist). Dies ist bei Anwendungssoftware im allgemeinen nicht der Fall.

c) Häufig kommt es vor, daß ein neues Programm ein altes ersetzen soll und "einfach" gefordert wird, daß neben neuen Fähigkeiten alle früheren erhalten bleiben sollen. Da in der Regel auch für den Vorgänger keine brauchbare Spezifikation existiert, muß das Verhalten nachgebildet werden. Das ist nicht nur dadurch problematisch, daß sich dieses nicht sicher feststellen läßt, sondern auch aufgrund der Unklarheit, welche Eigenschaften gewünscht, welche unvermeidbar gewesen waren. (Z.B. kann das alte System auf eine bestimmte Eingabe eine Fehlermeldung gebracht haben entweder, weil es nicht zur Verarbeitung in der Lage war, oder, weil diese Eingabe bewußt verboten sein sollte.)

d) Für die Feststellung der Anforderungen an ein Programm werden vielfach Schemata wie Fragebögen, Formulare, Dokumentationsregeln verwendet, die beitragen sollen zur Vollständigkeit und Verständlichkeit der Dokumentation und damit des Produkts.

e) Klare begriffliche und inhaltliche Abgrenzungen für die einzelnen Phasen des System-Lebenslaufs und die damit verknüpften Arbeiten und Dokumente fehlen.
Bei den Praktikern wächst aber das Bewußtsein, daß auf diesem Gebiet ein Defizit an begrifflichen Grundlagen, an Sprachmitteln und an Werkzeugen besteht. Damit steigt das Interesse an Arbeiten und Entwicklungen zu diesem Thema.

2.3.2 Spezifikations- und Entwurfssysteme

Die Bedeutung der Abkürzungen in diesem Abschnitt und Literaturangaben sind Kapitel 9 zu entnehmen; Beispiele und weitere Literaturhinweise finden sich in den Übersichtsarbeiten (z.B. Hershey, 1975; Boehm, 1976; Cheng, 1978; Ludewig, Streng, 1978a, 1978b; Hommel, 1978, 1980).

Eine saubere Klassifizierung der Systeme ist nicht möglich, weder nach den Zielen (z.B. Zuverlässigkeit, Produktivität, Effizienz) noch nach der Phase, in der sie eingesetzt werden können (Analyse, Entwurfsphase, Detaillierung). Sprache, Mittel und Methode werden nicht klar definiert und unterschieden. Oft fehlt das Verfahren, wird also nur gesagt, wie das fertige Modell aussehen soll (Balzert, 1978, S.49 und 61), oder es fehlt das Werkzeug, oder das Begriffssystem ist undefiniert (wenn mit natürlichen Sprachen gearbeitet wird). Die große Beliebtheit der Bezeichnung "Spezifikationssystem" trägt nicht zur Klarheit bei.

Der Jackson-Design-Methodology (JDM) liegt eine Datenstruktur-orientierte Betrachtungsweise zugrunde. Die Programmstrukur wird aus der Struktur der Ein- und Ausgabedaten abgeleitet (siehe auch 3.4.2.2). Die Tatsache, daß die JDM von vielen Praktikern als Heilslehre begrüßt wurde, zeigt, wie groß der Bedarf an Anleitungen zur Strukturierung ist.

Die Bewertungslisten für die (intermodulare) Kopplung und die (intramodulare) Bindung ("coupling" und "cohesiveness") aus Structured Design (SD) stellen einen wichtigen Versuch zur Objektivierung einer Beurteilung der Programmstruktur dar.

Ähnlich erfolgreich wie die JDM sind Structured Analysis and Design Technique (SADT) und Hierarchy plus Input Process Output (HIPO), beides Methoden zur graphischen Darstellung der Programme. Interessant ist der Dualismus von Ablauf- und Datenstrukturen in SADT, der durch die sich inhaltlich überschneidenden "Actigrams" und "Datagrams" betont wird. In der Praxis entsteht jedoch erfahrungsgemäß ein Ungleichgewicht zugunsten der Actigrams.

Die Module Interconnection Language (MIL 75) war ein interessanter, leider nicht ausgeführter Ansatz, verschiedene Sprachen für die Darstellung der inter- und der intramodularen Beziehungen zu verwenden.

In Higher Order Software (HOS) wird das Programm nach sehr strengen Regeln, Axiome genannt, aufgebaut. Durch zahlreiche Restriktionen wird eine Struktur erzwungen, die gute Voraussetzungen für die formale Verifikation bietet. Ähnliches gilt auch für die Programm-Spezifikation mit O-, V- und OV-Functions nach Parnas (Parnas, 1972).

Problem-Statement-Language/-Analyzer (PSL/PSA) ist heute das vermutlich weitestverbreitete Spezifikationssystem der Welt. Es ist pragmatisch konzipiert und bewährt sich vor allem als Dokumentationsmittel. Eine methodische Ausrichtung fehlt völlig.
Process Control Software Specification Language (PCSL) wird von einer tabellengesteuerten Version des PSA verarbeitet. Die Konzepte von PCSL sind eine Grundlage dieser Arbeit.

Software Requirements Engineering Program (SREP), das auf PSL/PSA basiert, ist speziell auf Echtzeitsysteme zugeschnitten. Durch eine externe Ursache ("stimulus") wird jeweils eine Systemreaktion hervorgerufen ("response"). Dabei vorkommende Verzweigungen in parallele oder alternative Aktionen sind zur Erzielung einer einfachen Struktur restringiert, etwa in der Art einer case-Anweisung. Ein ähnliches Konzept wird auch in dieser Arbeit verwendet.

Modular Approach to Software Construction, Operation and Test (MASCOT) hat ebenfalls ein Konzept zu dieser Arbeit beigetragen. Die "pools" für die ruhenden und die "channels" für die fließenden Daten kehren in 3.1.2 in ähnlicher Form als Variablen und Puffer wieder.

Matsumoto hat ein Verfahren angegeben (Matsumoto, 1977), das ganz auf der Beschreibung des technisches Systems durch Zustandsvektoren beruht und für spezielle Zwecke, z.B. die Darstellung von Schaltsequenzen, sehr vorteilhaft erscheint.

Computer-aided, Intuition guided Programming (CIP), bestehend aus einer Programmiersprache CIP-L und einem Transformationssystem, ist aus Forschungen über Breitbandsprachen hervorgegangen. CIP-L enthält neben üblichen Sprachelementen auch nicht-operationelle Elemente, die eine implementierungsunabhängige Beschreibung des Programms gestatten. Diese kann in ein operationelles, effizientes Programm transformiert werden, wobei die Erhaltung der Korrektheit sichergestellt ist (siehe 5.10).

3. Konzeption für ESPRESO

In diesem Kapitel wird ESPRESO, ein

System zur Erstellung der Spezifikation von Prozeßrechner-Software,

beschrieben. Es soll dazu beitragen, womöglich gewährleisten, daß Spezifikationen die angestrebten Eigenschaften (2.2.4) erhalten. Dazu muß ESPRESO während des Spezifikationsprozesses Führung und Unterstützung bieten. Insbesondere muß es die schritthaltende Prüfung und Dokumentation fördern.

Die Komponenten von ESPRESO sind

- eine Menge von Begriffen, aus denen Modelle der Programme und ihrer Umgebung, soweit diese beschrieben werden sollen, aufgebaut werden können ("universe of discourse", siehe 3.1),
- eine Sprache, die es gestattet, die Modelle als Spezifikationen zu formulieren (ESPRESO-S, 3.2),
- ein Werkzeug zur Speicherung, Prüfung und Verbreitung der Spezifikationen (ESPRESO-W, 3.3),
- ein Verfahren zur Erstellung der Spezifikation (3.4).

Nachfolgend werden diese vier Komponenten konzipiert. Die Reihenfolge ist so gewählt, daß die Darstellung erleichtert wird. Sie stellt nicht Schritte einer Entwicklung dar, denn die Komponenten bedingen sich gegenseitig. Die Realisierung der Sprache und des Werkzeugs folgt in Kapitel 4.

3.1 Das Begriffssystem (Modelle für die Programm-Beschreibung)

Um zu gewährleisten, daß die Komponenten des Spezifikationssystems auf die Probleme des Anwenders zugeschnitten und untereinander konsistent sind, muß zunächst ein Begriffssystem entwickelt werden, eine Sammlung von Konzepten, die dem Spezifikationssystem zugrundegelegt werden sollen.

Entsprechend den Ergebnissen von 2.2.2 wird in ESPRESO keine Unterscheidung zwischen Spezifikation und Entwurf vorgenommen. Es ist allerdings eine Kennzeichnung der Spezifikationsbestandteile anzustreben, die eine Unterscheidung der (frei gewählten) Entwurfsentscheidungen von den Anforderungen des Kunden ermöglicht.
Unterschiedliche Aspekte wie z.B. Prozeß- und Datenstrukturen werden ebenfalls zusammen dargestellt; die Vorteile einer getrennten Darstellung (Hammond, Murphy, Smith, 1978) sind nicht erkennbar.

Die Programme werden beschrieben als Systeme aus aktiven Komponenten, den Prozeduren und Blöcken (3.1.1), aus passiven Komponenten, den sog. Medien (3.1.2) und den Operationen der Prozeduren und Blöcke auf die Medien, den Aktionen (3.1.3).

3.1.1 Die funktionale Zerlegung (aktive Komponenten)

Nachfolgend werden Möglichkeiten zur Strukturierung des ausführbaren Codes von Programmen betrachtet; aus den Ergebnissen werden dann die Konzepte ausgewählt, die ESPRESO für die Spezifikation zur Verfügung stellen soll.

3.1.1.1 Die Ablaufstruktur

3.1.1.1.1
Das klassische Konzept zur Abstraktion und damit zur Strukturierung sequentieller Programme ist das Unterprogramm.
Mit der Relation 'läuft ab in', die hier zur Unterscheidung von parallelen Abläufen 'sequentiell in' genannt wird, hat das Programm dynamisch Baumstruktur. Jeder Knoten dieses Baums wird hier als Aktivität bezeichnet. Dabei ist es zunächst ohne Bedeutung, welchen Rechenvorschriften die einzelnen Aktivitäten zugeordnet sind. Da ihre Reihenfolge definiert ist, handelt es sich um einen geordneten Baum.

Beispiel für eine Ablaufstruktur mit sequentiellen Aktivitäten:

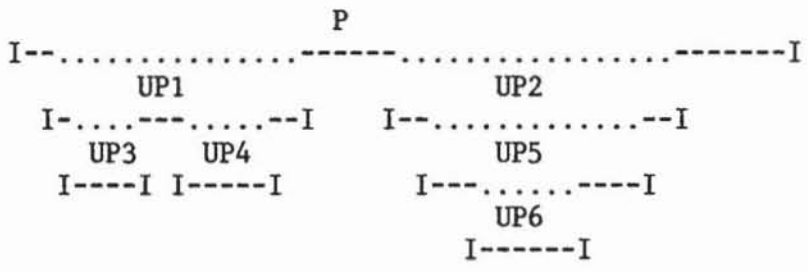


Bild 3.1 a

Der geordnete Baum dazu ist:

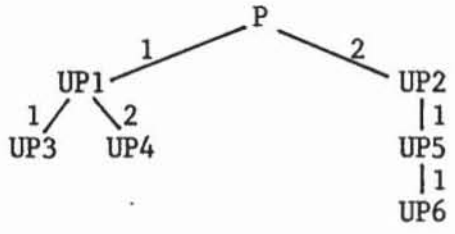


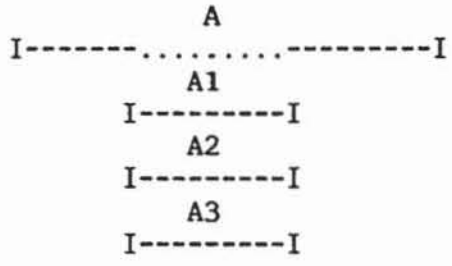
Bild 3.1 b

3.1.1.1.2
Parallele Abläufe sollen in ESPRESO auf ähnliche Weise beschreibbar sein wie die sequentiellen. Daher wird nachfolgend das Konzept der Aktivität erweitert.

Anders als bei den sequentiellen Aktivitäten entsteht bei einem nicht restringierten Konzept für asynchrone Aktivitäten keine Hierarchie, die eine sinnvolle Abstraktion darstellt. Dies gilt z.B., wenn eine Aktivität eine andere erzeugen kann und danach beide unabhängig voneinander sind. Daher wird für ESPRESO nur eine spezielle Form der Parallelität zugelassen, die an der "parallel clause" in ALGOL 68 orientiert ist.

Stehen die Aktivitäten A1 bis An zu A in der Relation 'parallel in', so laufen sie parallel ab; wie die übergeordnete Aktivität bei der Relation "sequentiell in" ruht A, bis alle Ai beendet sind. Damit entsteht auch hier ein - allerdings ungeordneter - Baum.

Beispiel für eine Ablaufstruktur mit parallelen Aktivitäten:



Der zugeordnete Baum ist:

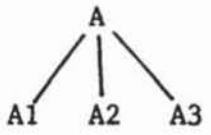


Bild 3.2

3.1.1.1.3
Für die weitere Beschreibung werden folgende Bezeichnungen eingeführt:

Wenn A1 bis An zu A in der Relation 'sequentiell in' stehen, werden sie als sequentielle Aktivitäten bezeichnet, mit A als Vater (siehe 2.1.3) bilden sie eine sequentielle Familie.

Entsprechend werden mit der Relation 'parallel in' die parallelen Aktivitäten und die parallele Familie definiert.

3.1.1.1.4
Um Unklarheiten bei der Strukturbeschreibung zu vermeiden, wird festgelegt, daß keine Aktivität Vater mehrerer Familien sein darf. Diese Bedingung kann stets durch Aufspaltung der Aktivitäten in sequentielle Aktivitäten erfüllt werden.

Beispiel für eine Ablaufstruktur mit sequentiellen und parallelen Aktivitäten (A wird durch Umformung nach A' überführt)

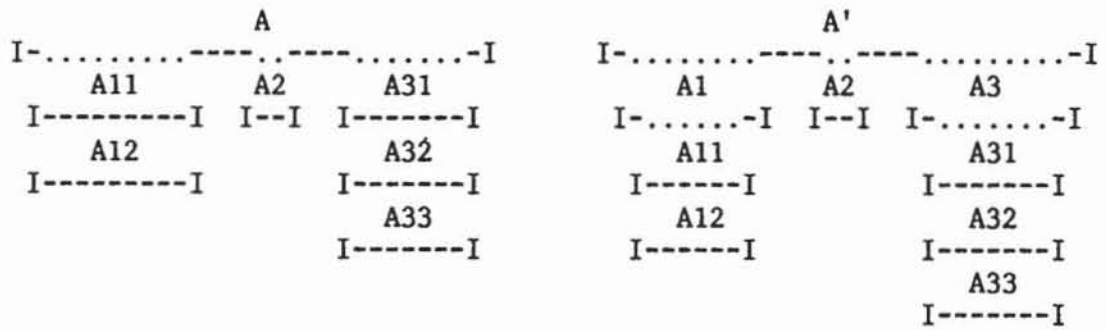


Bild 3.3

3.1.1.1.5
Die Relation 'untergeordnet' wird definiert als transitive Hülle der Vereinigung von 'sequentiell in' und 'parallel in'. Aufgrund der Regeln für diese beiden Relationen bildet 'untergeordnet' einen Wald. Ohne Einschränkung der Allgemeinheit kann gefordert werden, daß es eine einzige übergeordnete Aktivität gibt und so der Ablauf des zu beschreibenden Systems durch einen einzigen Baum, den sog. Ablaufbaum, dargestellt werden kann. Zerschneidet man diesen so, daß die parallelen Aktivitäten von ihren Vätern getrennt sind, so entstehen einzelne Bäume, die man als (Rechen-) Prozesse bezeichnet.

3.1.1.2 Die Programmstruktur

3.1.1.2.1 Rechenvorschriften

Die in 3.1.1.1 betrachteten Aktivitäten sind dynamische Objekte. Sie entstehen als Ausführungen statischer Objekte, für die nachfolgend die Bezeichnung "Rechenvorschrift" verwendet wird. Jede Rechenvorschrift kann beliebig oft ausgeführt werden, also beliebig viele Aktivitäten erzeugen. Rechenvorschrift, ausführender Prozessor (oder ausführende Prozessoren) und erzeugte Aktivität stehen also zueinander im gleichen Verhältnis wie Schallplatte, Plattenspieler und die erzeugte Musik.

Die Zuordnung zwischen Aktivitäten und Rechenvorschriften definiert eine Abbildung. Wendet man diese auf den Ablaufbaum (3.1.1.1.5) an, so entsteht ein gerichteter Graph, der sogenannte Strukturgraph. Dieser hat im allgemeinen keine Baumstruktur, da jede Rechenvorschrift von mehreren anderen oder auch rekursiv verwendet (aufgerufen) werden kann.

3.1.1.2.2 Terminale und nichtterminale Rechenvorschriften

Um die Beschreibung alternativer Ablaufvarianten zu ermöglichen, wird zur sequentiellen und parallelen Ausführung von Rechenvorschriften die alternative hinzugenommen (Fallunterscheidung).

Entsprechend den Aussagen zu Hierarchien (2.1.3) wird festgelegt:

Ein Rechenvorschrift ist entweder atomar oder verwendet Rechenvorschriften Q1 bis Qn, $Qi \neq Qk$ für $i \neq k$, nach einem von mehreren vorgegebenen Schemata. Diese Schemata sind:

a) Sequentielle Ausführung von Q1 bis Qn. Diese Struktur wurde in 3.1.1.1.3 als sequentielle Familie eingeführt (wird nachfolgend auch für Rechenvorschriften verwendet).

b) Parallele Ausführung von Q1 bis Qn. Auch hierfür wird die Bezeichnung (parallele Familie) aus 3.1.1.1.3 übernommen.

c) Ausführung genau einer Rechenvorschrift aus Q1 bis Qn entsprechend einem zugeordneten Selektor
(Relationen 'alternativ in' und 'Selektor in').

Die Strukturen a bis c erfordern alle, daß die untergeordneten Rechenvorschriften ihrem Vater exklusiv zugeordnet sind. Zur Beschreibung anderer Fälle dient d:

d) Ausführung genau einer Rechenvorschrift (Relation 'ruft auf'), die auch von anderen mit dieser und nur dieser Relation verwendet werden kann.

Strukturgraphen, die andere Strukturen enthalten, als durch (a) bis (d) zugelassen sind, können durch Einfügen zusätzlicher Rechenvorschriften angepaßt werden.

Für die Schleife (do-loop) ist keine hierarchische Relation vorgesehen. Stattdessen kann jeder Rechenvorschrift eine Variable als Wiederholbedingung (3.1.2.2) und/oder ein Trigger als Abbruchkriterium (3.1.2.4) zugeordnet werden, wodurch sie als zyklisch gekennzeichnet ist.

3.1.1.3 Prozeduren und Blöcke

3.1.1.3.1
Beim Übergang von der Spezifikation zum Entwurf könnten alle Rechenvorschriften zu Programmeinheiten (Subroutines, Procedures o.ä.) gemacht werden. Die entstehenden Einheiten wären aber zu klein, um als Gültigkeitsbereiche (siehe 3.1.5) brauchbar zu sein und um eine überschaubare Gesamtstruktur zu gewährleisten (vgl. Horning, Randell, 1973, S.26). Es ist daher zweckmäßig, die Zusammenfassung mehrerer Rechenvorschriften zu einer größeren Betrachtungseinheit zu gestatten.

Hier wird die Prozedur eingeführt. Formal ist die Abbildung der Rechenvorschriften auf Prozeduren mit Hilfe des Strukturgraphen definiert:

Eine Rechenvorschrift wird Repräsentant einer Prozedur, wenn

- ihre Ausführung die übergeordnete Aktivität bildet (entsprechend dem Hauptprogramm) oder

- sie nach 3.1.1.2.2 (d) aufgerufen wird, im Strukturgraphen also im allgemeinen mehrere Kanten auf sie gerichtet sind.

Den Kanten des Strukturgraphen folgend werden nun alle verbliebenen Rechenvorschriften den Prozeduren zugeordnet. Da sie jeweils eindeutig einem Vater zugeordnet sind, ist diese Abbildung eindeutig.

Praktisch bedeutet diese Definition, daß eine Prozedur nur einen Eingang hat und intern Baumstruktur aufweist.

Unterbäume der Prozeduren werden in Anlehnung an blockorientierte Sprachen als Blöcke bezeichnet. Diese stellen Subsysteme im Sinne von 2.1.3 dar.

3.1.1.3.2 Beispiel für Ablaufbaum und zugeordneten Strukturgraphen

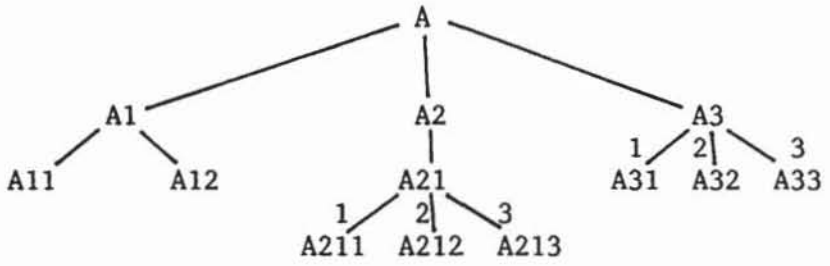


Bild 3.4 a

Dieser Baum stellt das Hasse-Diagramm der Relation 'untergeordnet' zwischen den Aktivitäten dar. Seien die Unterbäume mit den Repräsentanten A21 und A3 statisch gleich, ebenso A12 und A211. Für diese werden zusätzliche Rechenvorschriften R1 und R2 eingeführt, damit die Regeln in 3.1.1.2.2 erfüllt sind. Damit ergibt sich dieser Strukturgraph:

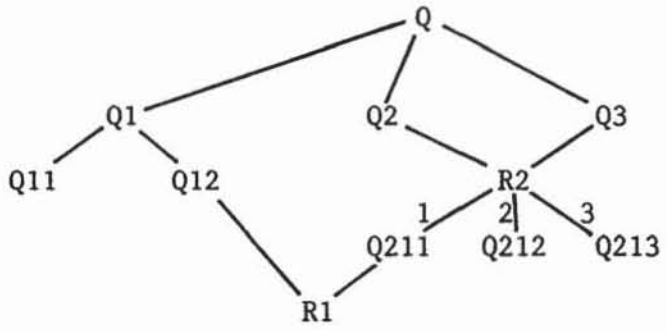


Bild 3.4 b

Die Zuordnung zwischen Aktivitäten und Rechenvorschriften ist leicht erkennbar. Q12, Q211 und Q3 dienen nur der Strukturbeschreibung, der übereinstimmende Inhalt von A12 und A211 wird aus R1, der von A21 und A3 aus R2 erzeugt.

Der Strukturgraph enthält drei Prozeduren:

P1: Q + (Q1 + (Q11) + (Q12)) + (Q2) + (Q3);
P2: R2 + (Q211) + (Q212) + (Q213); P3: (R1).

Die Blöcke sind durch Klammern markiert.

3.1.1.3.3
Zur Prozedur gehört der Begriff des Parameters. In ESPRESO wird unterschieden zwischen Eingabe-Parametern, Ausgabe-Parametern und transienten Parametern. Bei Eingabe- und Ausgabe-Parametern ist der Informationsfluß auf eine Richtung beschränkt.

Bei den Prozeduren kann es nötig sein, Prioritäten anzugeben und dadurch zu regeln, welcher Prozeß Vorrang beim konkurrierenden Zugriff auf ein Medium oder einen Prozessor haben soll. In ESPRESO wird dafür nur ein einfacher Mechanismus vorgesehen, mit dessen Hilfe die Priorität der Prozedur fest zugeordnet ist.

3.1.2 Medien

Außer den aktiven Komponenten (3.1.1) gehören zu einem Programm Daten und Betriebsmittel, die hier zusammenfassend Medien genannt werden. Verschiedene Arten von Medien werden nachfolgend definiert und im Hinblick auf die mit ihnen verbundenen Koordinierungsprobleme diskutiert.

3.1.2.1 Sprachkonzepte für die Koordinierung paralleler Aktivitäten

Greifen mehrere Aktivitäten gleichzeitig auf ein Medium zu, z.B. auf ein Peripheriegerät, einen Speicherbereich oder eine Variable, so kann es vielfältige unerwünschte Effekte geben, wenn die Zugriffe nicht koordiniert werden. Die Sprachmittel für sequentielle Programme reichen für die Koordinierung nicht aus. Daher wurden spezielle Konzepte geschaffen, z.B. Semaphore (Dijkstra, 1968b), Monitore (Brinch-Hansen, 1973; Hoare, 1974) und path expressions (Campbell, Habermann, 1974).

Alle diese Konzepte sind unabhängig von bestimmten Maschinen, aber universell, problemunabhängig und auf eine effiziente Realisierung ausgerichtet. Diese Arbeit wählt den umgekehrten Ansatz. Es werden problemorientierte Konzepte angeboten, mit denen sich die in der Praxis vorkommenden Aufgabenstellungen leicht spezifizieren lassen. Anschließend kann ihre Abbildung auf möglichst effiziente Algorithmen angestrebt werden. Effektive Algorithmen sind durch die Abbildung auf Betriebssystemaufrufe in 5.8 gegeben.

Dieses Vorgehen bietet sich bei Anwendungsprogrammen an, weil die Zahl der Aufgabentypen in der Praxis äußerst gering ist. Drei Typen treten immer wieder auf:

a) das Schema der exklusiven Verwendung von Betriebsmitteln,

b) das Lese-/Schreib-Schema,

c) das Hersteller-/Verbraucher-Schema.

(a) und (c) werden von Presser (1975, S.24) als "natural coordination needs" bezeichnet. Offenbar kommt aber b ebenso "natürlich" vor. Kompliziertere Probleme, wie sie z.B. von Agerwala (1977) zum Vergleich verschiedener Synchronisationsmittel herangezogen werden, spielen in der Praxis keine Rolle.

Die hier gewählten Synchronisationsprimitive bilden also weder ein vollständiges System im Sinne von Agerwala, Flynn (1975) noch ein minimales (vgl. Uhrig, 1978).

3.1.2.2 Variablen

Prozeßrechner-Programme arbeiten mit zeitlich veränderlichen Größen, also n a t ü r l i c h e n Variablen, die nicht erst, wie in der Stapelverarbeitung, wo die Eingabedaten a priori festliegen, durch die Substitution rekursiver Prozeduren entstehen (Bauer, 1976). Daher muß die Spezifikationssprache ein Konzept für Variablen enthalten.

Eine Variable ist ein abstrakter Speicher für ein Datum bestimmten, primitiven oder komplexen Typs. Ihr Wert ist undefiniert bis zum ersten schreibenden Zugriff, anschließend kann er beliebig oft gelesen werden.

Lese- und Schreibzugriffe sind, wenn sie nicht zeitlich zusammentreffen, völlig unabhängig voneinander. Mehrere Lesezugriffe können simultan erfolgen; Schreibzugriffe bedürfen dagegen des exklusiven Zugriffs, um ein definiertes Ergebnis zu haben. Meist ist es erwünscht, den Schreibern mindestens so gute Chancen zum Zugriff zu geben wie den Lesern. Dieses Koordinierungsschema ist in der Literatur als "2nd reader/writer problem" bekannt (Courtois, Heymans, Parnas, 1971).

Wird eine Variable mit einem Wert besetzt, der vom früheren Wert abhängt, so muß sie erst gelesen, dann geschrieben werden. Für diese Kombination wird hier die Bezeichnung "Änderung" verwendet.

Eine Variable kann als Kriterium einer Fallunterscheidung oder, wenn sie einen Wahrheitswert enthält, als Wiederholbedingung zyklischer Rechenvorschriften verwendet werden (siehe 3.1.1.2.2). In beiden Fällen handelt es sich um spezielle Fälle des lesenden Zugriffs.

Zum Begriff der Variablen vgl. Dijkstra (1968a, Remark 5). Wie die folgenden Beispiele zeigen, eignen sich Variablen auch für nicht rechnerinterne Information.

Beispiele:

a) Eichtabelle eines Meßgerätes zur Datenerfassung.
   Die Tabelle werde unregelmäßig von einem Eichprozeß neu gesetzt, von anderen Prozessen verwendet. Dabei ist es nach der Initialisierung irrelevant, in welcher Sequenz die Zugriffe ausgeführt werden.

b) Vom Rechner abgefragter Meßwert des technischen Prozesses.
   Die Schreiboperation ist hier Sache des technischen Prozesses.
   Bei Stellwerten ist es umgekehrt.

c) Schalterstellungen am Wartungsfeld und an der Prozeßperipherie.
   Der Unterschied zu (b) besteht darin, daß hier der Bediener statt des technischen Prozesses steht.
   Ein Beispiel für die umgekehrte Datenflußrichtung ist die Anzeige durch Signallampen.

3.1.2.3 Puffer

3.1.2.3.1
Die Entsprechung zum Schreiben und Lesen von Variablen ist bei materiellen Objekten scheinbar die Lieferung und Abnahme von Gegenständen. Der grundsätzliche Unterschied besteht darin, daß mit der Lieferung eine Vermehrung, mit der Abnahme eine Verminderung der vorhandenen Gegenstände untrennbar verbunden ist, ebenso auch eine Verminderung bzw. eine Vermehrung der freien Plätze, falls der Lagerraum beschränkt ist.

Hier wird als Modell der Puffer eingeführt. Ein Puffer besteht aus Plätzen, auf denen Gegenstände einer ganz bestimmten Art (Typ) gelagert werden können. Eine Lieferung an den Puffer ist nur möglich, wenn er einen freien Platz enthält, eine Abnahme nur, wenn mindestens ein Platz besetzt ist. Wenn nötig, muß ein Zugriff auf den Puffer verzögert werden, bis diese Voraussetzung besteht (Sperrverhalten).

Dieses zunächst für materielle Objekte beschriebene Modell ist in vielen Fällen auch für Daten sinnvoll. Zwar werden diese normalerweise durch den lesenden Zugriff nicht zerstört, doch sind sie anschließend oft irrelevant, und der von ihnen belegte Speicherplatz wird freigegeben, wenn die Information verarbeitet ist.
Da verlangt wird, daß der Puffer nur ganz bestimmte Gegenstände enthält, identifiziert er eine bestimmte Klasse von Objekten, so daß oft der Puffer mit dem Namen der Objektklasse bezeichnet wird.

Beispiele für Puffer:

a) "Briefkasten" eines Prozesses zur Aufnahme von Botschaften anderer Prozesse.

b) Puffer zur Aufnahme von Interrupts bei Unterbrechungssperre.

c) Meßdaten in einem Datenerfassungssystem.
   Die Daten werden von der Messung bis zur Dokumentation durch das Datenerfassungssystem geschleust, dabei transformiert, vervielfältigt und gelöscht. Jedem Stadium der Daten entspricht ein Puffer, z.B. für Rohdaten oder gefilterte Daten.

Der Zusammenhang zwischen Puffern und Rechenvorschriften ist der Beziehung zwischen Stellen und Transitionen in Petri-Netzen ähnlich.

3.1.2.3.2
Neben Lieferung und Abnahme können noch drei weitere Operationen auf Puffer nötig sein: Ein Puffer muß zu Beginn, eventuell auch später erneut initialisiert werden, und der Bestand des Puffers ("Pufferpegel") muß festgestellt werden. In Zusammenhang mit diesen beiden Operationen und in andern Fällen ist es erforderlich, den Puffer gegen alle Zugriffe anderer Prozesse zu sperren. Die genannten Operationen erhalten die Bezeichnungen "initialize", "test" und "inhibit".

3.1.2.3.3
Ein Puffer für Informationsobjekte hat entweder eine feste Zahl von Plätzen und damit einen definierten Speicherbedarf, oder er belegt den Speicher dynamisch und kann Objekte aufnehmen, solange Platz frei ist.

Ein Puffer mit n Plätzen, der zwei zyklische Prozesse koppelt, erlaubt dem liefernden einen Vorsprung von bis zu n durchlaufenen Zyklen gegenüber dem abnehmenden. Um auch die strenge Synchronisation der beiden Prozesse erzwingen zu können, werden Puffer mit null Plätzen ("Null-Puffer") zugelassen. Lieferung und Abnahme erfolgen bei einem solchen Puffer direkt ohne Zwischenspeicherung.
Die Kommunikation über einen Null-Puffer ist dem "Rendezvous" in ADA (ADA, 1979, 9.5 und 9.7) ähnlich.

3.1.2.3.4
Üblich ist bei Puffern das in 3.1.2.3.1 beschriebene Sperrverhalten. In Sonderfällen gibt es aber auch ein anderes Verhalten, das man in Anlehnung an die Vermittlungstechnik als Verlustverhalten bezeichnen kann: Ein momentan nicht erfüllbarer Zugriffswunsch bleibt nicht bestehen, sondern geht verloren. Dieses Verhalten zeigen z.B. Fernsprecheinrichtungen, wenn eine Verbindung wegen Belegung nicht hergestellt werden kann. Weitere Beispiele sind

- Interrupts bei äußerer Maskierung (Baumann, 1972),

- Prozeßmeldungen in einem überlaufenden Puffer.

Bei diesen Beispielen handelt es sich stets um eine Lieferung an den Puffer, bei der der Verlust auftritt. Formal läßt sich zwar auch die Abnahme aus einem leeren Puffer definieren, jedoch ist dies weniger anschaulich. In solchen Fällen ist immer eine spezielle Reaktion erforderlich, so daß die Test-Operation (3.1.2.3.2) das geeignete Mittel darstellt.

Bei der Verwendung dieses Modells ist allerdings zu beachten, daß durch das Verlustverhalten eine wesentliche Eigenschaft des Puffers (wie des Semaphors) verloren geht, die Kommutativität der auf ihn ausgeführten Zugriffe (vgl. die Bemerkung zu "wait" und "cause" in Dijkstra, 1971, S.125).

3.1.2.3.5
Hat der Puffer mehrere Plätze, so kann es von Bedeutung sein, in welcher Reihenfolge die Objekte entnommen werden. Meist wird erwartet, daß sich der Puffer als Warteschlange verhält (FIFO). Oft kommt es auch vor, daß die Objekte im Puffer nach ihrer Wichtigkeit geordnet werden.

3.1.2.3.6
Ein Puffer ist also durch folgende Angaben charakterisiert:

- Typ der Objekte, die er verwaltet (siehe 3.1.2.6),
- Zahl der Plätze und/oder Speicherbedarf pro Platz,
- Sperr- oder Verlustverhalten bei Lieferung an den Puffer,
- Reihenfolge der Ausgabe (nur bei mehreren Plätzen).

3.1.2.3.7
Beispiele für Puffer, wie sie in ESPRESO modelliert werden:

- Meldungspuffer

  Einheit (Typ): Meldungsstring.
  Kapazität: 20.
  Verlustverhalten.
  Organisation: FIFO.

- Ausgabeblattschreiber

  Einheit (Typ): Zeile.
  Kapazität: 1.
  Sperrverhalten.

  Der Mechanismus, der die Zeilenfortschaltung des Blattschreibers bewirkt, stellt hier den Verbraucher dar. Dieser Puffer liegt also an der Peripherie des zu spezifizierenden Systems (siehe 3.1.6).

3.1.2.4 Trigger, Starten und Beenden der Aktivitäten

Den Puffern ist nach 3.1.2.3.1 der Typ der von ihnen verwalteten Gegenstände zugeordnet. Für Fälle, in denen die Gegenstände wie die Marken in Petri-Netzen keinerlei Information transportieren als ihr bloßes Vorhandensein, werden die Trigger eingeführt. Trigger sind also spezielle Puffer, sie können auch als Auftragspuffer bezeichnet werden.

Für Trigger gibt es außer den Operationen für Puffer zwei zusätzliche: Die Ausführung einer Rechenvorschrift kann durch einen Trigger gestartet oder beendet werden. In beiden Fällen ist damit der Verbrauch einer Marke aus dem Trigger verbunden.

Starten und Beenden werden in ESPRESO nur in einer speziellen Form zugelassen: Die zu startende oder zu beendende Rechenvorschrift kann nicht zu jedem beliebigen Zeitpunkt, sondern nur an einem bestimmten Punkt ihrer Ausführung, nämlich am Anfang, beeinflußt werden, was vor allem bei zyklisch ablaufenden Rechenvorschriften (3.1.1.2.2) sinnvoll ist.

Diese Einschränkung hat Vorteile für die Klarheit der Spezifikation: Die Bindung der Ereignisse an Trigger erzwingt eine Festlegung, was zu geschehen hat, wenn die Reaktion nicht sofort erfolgen kann. Bei Ereignissen, denen der Speicher als "Gedächtnis" fehlt, bleiben solche Punkte oft unklar. Durch die Beeinflussung der Prozeduren und Blöcke nur am Beginn ist sichergestellt, daß die Hierarchie der Ablaufstruktur nicht zerstört wird. Bei der Implementierung hat dieses Konzept zudem den Vorteil, daß ein einfaches Prozeßzustandsdiagramm ausreicht (siehe 5.8.7).

Trigger dienen auch zur Beschreibung von Uhr-Funktionen. Dazu kann ihnen ein Zyklus und/oder eine Wartezeit zugeordnet werden. Sie erzeugen dann mit dem angegeben Zyklus und/oder nach der angegebenen Wartezeit selbst "Pulse". In den meisten Fällen wird es sinnvoll sein, diesen Triggern die Kapazität null und Verlust-Verhalten zuzuordnen.

Im Sinne des TRIGGER und INDUCE aus PEARL (PDV, 1977b, 2.4.1.2) kann auch die Lieferung an solche Trigger sinnvoll sein.

Beispiele für Trigger:

- Interrupts mit innerer und äußerer Maskierung.
  Bei äußerer Maskierung wird der Trigger maskiert, bei innerer Maskierung wird die vom Trigger gestartete Aktivität blockiert.

- Wartezeit in einer Aktivität.
  Der Trigger wird mit der entsprechenden Verzögerung definiert.

3.1.2.5 Betriebsmittel

Betriebsmittel sind Objekte, die von Aktivitäten belegt werden können, mit dem Ende der Aktivität jedoch ohne relevante Veränderung wieder freigegeben werden. In ESPRESO werden sie in zweierlei Bedeutung verwendet:

Reale Betriebsmittel sind alle in begrenzter Anzahl vorhandenen Systemkomponenten, die von den Aktivitäten benötigt werden. Ausgenommen sind die bereits als Prozeduren, Variablen, Puffer oder Trigger beschriebenen Objekte. Beispiele für Betriebsmittel sind Speicherplätze und Bandgeräte. Hat ein Betriebsmittel die Kapazität n, so stehen n Exemplare zur Verfügung. Eine Aktivität kann 1 bis n davon belegen.

Virtuelle Betriebsmittel sind nicht physisch vorhanden. Sie werden gebraucht als abstrakte "Behälter" für mehrere Medien, die nicht unabhängig voneinander verändert werden dürfen. (6.1.6.2 enthält mit dem Betriebsmittel "Buchführung-2" ein Beispiel.)

Die Belegung der Betriebsmittel erfolgt im gegenseitigen Ausschluß (mutual exclusion). Sie ist an die Lebensdauer der belegenden Aktivität geknüpft (vgl. 3.1.3).

3.1.2.6 Typen

Zur Beschreibung der Variablen, Puffer und Parameter werden in ESPRESO - wie in den meisten Programmiersprachen - Typen verwendet. Wie üblich sind einige elementare Typen a priori definiert:

- natürliche Zahlen (einschließlich Null),
- ganze Zahlen,
- reelle Zahlen,
- Zeichenreihen,
- Wahrheitswerte.

Aufzählungstypen können wie in PASCAL definiert werden. Komplexe Typen werden gebildet als

- Listen (Felder),
- Verbunde (Strukturen),
- Verweise (Zeiger).

3.1.3 Aktionen

In 3.1.2 wurden bereits die möglichen Operationen der aktiven Komponenten auf die Medien der einzelnen Arten beschrieben. Diese Verknüpfungen werden als Aktionen bezeichnet. Zwei Klassen sind zu unterscheiden: Die paarigen Aktionen bestehen aus zwei zusammengehörigen Teilaktionen (wie belegen und freigeben); die anderen heißen einfache Aktionen oder Transfers (wie z.B. lesen). Auch die Steueraktionen werden zu den Aktionen gezählt, also starten und beenden (3.1.2.4) sowie die Steuerung der zyklischen Ausführung durch eine Variable (3.1.2.2).

Für die drei Klassen ist die Reihenfolge der Ausführung festgelegt:

1. Steueraktionen (warten bis zum Start, prüfen auf Ende-Bedingungen),
2. erster Teil der paarigen Aktionen (belegen und sperren),
3. Transfers (lesen, schreiben, ändern, initialisieren, liefern, holen),
4. Ausführung untergeordneter Rechenvorschriften,
5. zweiter Teil der paarigen Aktionen (Sperre in den Zustand versetzen, der angetroffen worden war, und freigeben).

Die verschiedenen Steueraktionen sind auch untereinander geordnet: Sind alle drei möglichen Arten der Steuerung genutzt, so wird zunächst der Start durch einen Trigger ausgeführt, dann wird geprüft, ob ein als Abbruchkriterium dienender Trigger nicht leer ist, schließlich wird der Wert der boolschen Variablen geprüft, die der Aktivität als Wiederholbedingung zugeordnet ist.

Mehrere paarige Aktionen werden ggf. geschachtelt. Zuerst werden die Belegungen ausgeführt, dann die Sperrungen.

Die Transfers dienen vorwiegend der Weitergabe von Information, die entsprechend dem Konzept von ESPRESO nicht formal beschrieben werden kann. Doch ist eine informale Beschreibung möglich. Dazu dienen die sogenannten Zusicherungen (siehe 3.2.3.2). Die Reihenfolge der Transfers ist undefiniert.

Es sei ausdrücklich darauf hingewiesen, daß eine Rechenvorschrift nur durch eigene Aktionen (also nicht solche über- oder untergeordneter Rechenvorschriften) oder durch Parameter der Prozedur, zu der sie gehört, Information von außen erhalten oder nach außen weitergeben kann.

Die Tabelle 3.1 gibt einen Überblick der Aktionen. Die speziellen Bedeutungen sind in den betreffenden Abschnitten zu den Medien beschrieben.

Tabelle 3.1
Aktionen

Klasse	Aktion	Bezeichnung in Grammatik (7.)	wird angewandt auf Medien der Arten --> Variable	Puffer	Trigger	Betriebsmittel
Steuer-aktionen	gestartet werden	started by			x	
	beendet werden	terminated by			x	
	wiederholen falls	while	x			
paarige Aktionen	belegen	occupies				x
	sperren	inhibits		x	x	
Transfers	lesen	reads	x			
	schreiben	writes	x			
	ändern	updates	x			
	initialisieren	initializes	x	x	x	
	liefern	produces		x	x	
	holen	consumes		x	x	
	testen	tests		x	x	

3.1.4 Konstanten und Fristen

Für Größen, die dem zu spezifizierenden System fest vorgegeben sind, gibt es in ESPRESO Konstanten, denen Name und Typ zugeordnet sind. Wertangaben für Konstanten (z.B. Zahlen) sind dagegen nicht vorgesehen.

Eine Frist ist ein Zeitintervall, das näher bestimmt sein kann durch eine Variable oder Konstante und durch eine andere Frist. Einige Einheiten sind vordefiniert (sec, msec, min, h).
Bestimmen Fristen, die von Variablen abhängig sind, Verzögerung und Zyklus von Triggern, so ist der Wert zum Zeitpunkt der Initialisierung maßgeblich. Auf diesen Zeitpunkt werden anschließend auch die Uhr-Funktionen bezogen.

3.1.5 Zugriffsrechte und Gültigkeitsbereiche: Moduln

In fast allen Programmiersprachen besteht die Möglichkeit, die Gültigkeit eines Namens und damit i.a. die Verwendbarkeit des dem Namen zugeordneten Objekts (z.B. eine Variable oder Prozedur) auf einen Teil des Programms zu beschränken. Oft geschieht dies sogar automatisch, z.B. beim Schleifenzähler in ALGOL 68. Diese Beschränkung hat den Vorteil, daß die Gefahren unerwünschter Nebeneffekte durch Namenskollisionen vermindert werden. Auch wirken sie disziplinierend auf den Programmierer und erhöhen die Lokalität des Programms, eine Voraussetzung der Übersichtlichkeit.

Sind die Gültigkeitsbereiche disjunkt, so müssen Möglichkeiten zu einer Verbindung zwischen verschiedenen Gültigkeitsbereichen bestehen, z.B. durch globale Objekte (wie die Subroutines in Fortran) und Mechanismen für den Datenaustausch (Parameter-Übergabe).

Auch eine Spezifikationssprache sollte die Abgrenzung von Gültigkeitsbereichen fördern und Mittel für die sichere Kommunikation über die Grenzen dieser Bereiche hinweg bereitstellen.

Für ESPRESO kommen als Gültigkeitsbereiche zunächst die Prozeduren und Blöcke in Betracht. Das Prinzip der abstrakten Datenstrukturen (siehe 3.4.2.1) erfordert aber eine Ausweitung der Gültigkeitsbereiche über mehrere Prozeduren, um alle Operationen auf einem abstrakten Datentyp realisieren zu können. Daher wird der Modul eingeführt. Er entspricht dem "aggregate" (Bayer, 1974) und dem Modul in MODULA (Wirth, 1977).

Ein Modul ist eine Menge von

(a) Medien, Typen, Konstanten und Fristen,
(b) Prozeduren,
(c) Blöcken,
(d) eingeschachtelten Moduln.

Die Moduln bilden also Baumstrukturen. Wie bei den Aktivitäten wird ohne Einschränkung der Allgemeinheit verlangt, daß nur e i n Baum besteht, daß also ein ausgezeichneter Modul allen anderen übergeordnet ist. Die Modulnamen (d) sind uneingeschränkt gültig. Die Namen der unter (a) bis (c) genannten Objekte sind, wenn die Definition in Modul M steht und nicht in einem M untergeordneten Modul, wie folgt gültig:

(a) in M und dessen Modul-Subsystem;

(b) in M1 bis Mn und deren Modul-Subsystemen bei entsprechender Angabe zum Objekt (d.h. zur Prozedur); fehlt eine solche Angabe, wie (a);

(c) in M und in allen Moduln, zu deren Subsystem M gehört.

Beispiel:

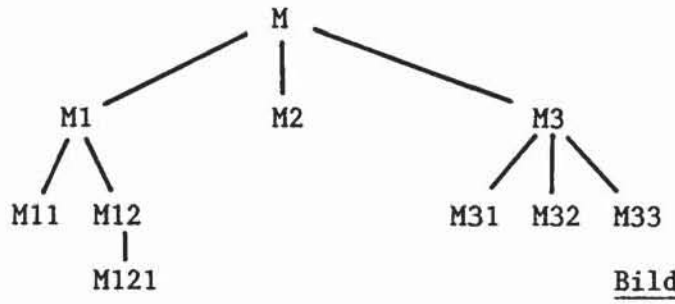


Bild 3.5 Modulbaum

In M1 seien eine Variable V, eine Prozedur P1, eine Prozedur P2 mit der Angabe "verwendbar in M3" und ein Block B definiert. V und P1 können dann in M1, M11, M12 und M121, P2 in M3, M31, M32 und M33 und B in M1 und M verwendet werden.

Da die Moduln einen Baum bilden, sind überlappende Zugriffsrechte zunächst nicht beschreibbar:

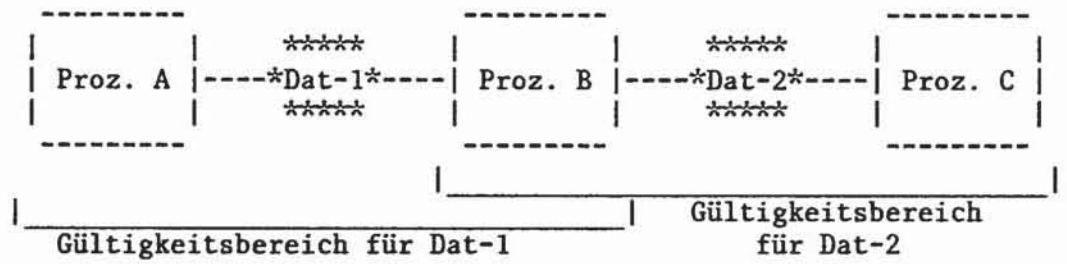


Bild 3.6 Überlappende Gültigkeitsbereiche

Für solche Situationen wird die Möglichkeit vorgesehen, die Zugriffsrechte für ein Medium auf einzelne, untergeordnete Moduln einzuschränken, und zwar differenziert nach den Zugriffsarten (z.B. Schreiben und Lesen bei Variablen). Um die Sprache nicht zu überlasten, werden dabei logisch zusammengehörige Operationen (z.B. Schreiben und Initialisieren) zusammengefaßt.

3.1.6 Weitere Angaben zur Spezifikation

Für die Implementierung eines Systems ist es notwendig zu wissen, welche spezifizierten Komponenten bereits vorhanden und nur als Teil der komplementären Spezifikation beschrieben sind. Ebenso ist eine Markierung solcher Objekte nützlich, die Beziehungen über die Grenzen des Systems nach außen haben, denn ihnen fehlt oft eine vollständige Beschreibung ihrer speziellen Umgebung (z.B. Variablen, die scheinbar nie gelesen werden). Daher besteht in ESPRESO die Möglichkeit, Medien und Prozeduren entsprechend zu kennzeichnen.

Um den Speicher- und Laufzeitbedarf des Gesamtsystems abschätzen zu können, braucht man den Bedarf der einzelnen Komponenten. In ESPRESO sind daher entsprechende Angaben vorgesehen (vgl. 3.3.2.2.9).

3.2 Die Spezifikationssprache

Zu dem in 3.1 abgeleiteten Begriffssystem muß eine Sprache, d.h. eine konkrete Grammatik, entwickelt werden. Durch die Beschränkung auf ein spezielles Begriffssystem wirkt sie disziplinierend, indem sie den Anwender zur Verwendung bestimmter Konzepte zwingt. Die Grundzüge dieser Sprache, ESPRESO-S genannt, sollen nachfolgend beschrieben werden.

Der Komplexität der Grammatik sind durch die Randbedingungen, die für das Werkzeug gelten (3.3), Grenzen gezogen. Innerhalb dieser Grenzen soll versucht werden, der Sprache möglichst viel von den Qualitäten zu geben, die Gries (1976) unter dem Begriff "Style" zusammengefaßt hat: Sie soll einfach sein und sauber definiert, eine gefällige Syntax haben und knapp, aber nicht unverständlich sein.

Nachfolgend wird jede Spezifikation eines Programms in ESPRESO-S als ESPRESO-Spezifikation bezeichnet. Dabei wird abweichend von 2.2.1.1 nur die syntaktische, nicht die logische Vollständigkeit verlangt.

Die Beispiele dieses Abschnittes müssen der in 4.1 folgenden Auflistung der Sprachelemente vorgreifen. Die formale Definition steht im Anhang.

3.2.1 Vorüberlegungen

3.2.1.1 Syntax und Semantik von ESPRESO-S

Bei Programmiersprachen wird unterschieden zwischen den Regeln für die Form der Programme und den den Programmen zugeordneten Bedeutungen, zwischen Syntax und Semantik. Hesse (1976) diskutiert die beiden Begriffe und zeigt, daß die Grenzziehung willkürlich ist.

ESPRESO-S ist keine Programmiersprache; weder enthält die ESPRESO-S-Formulierung das Programm als Untermenge, da sie nicht hinreichend formalisiert und detailliert ist, noch ist sie umgekehrt im Programm enthalten, da sie z.B. Information über die Anforderungen enthält, die nicht ins Programm gelangt. Daher ist hier die Grenze noch weniger klar als bei Programmiersprachen.

Für ESPRESO-S wird folgende Definition gewählt: Die Syntax ist die Menge aller Regeln, deren Einhaltung bei der Eingabe überwacht werden soll. Syntaktische Fehler rufen Fehlermeldungen hervor und führen zur Ablehnung des Abschnitts, in dem sie festgestellt werden.

Die Syntax enthält nicht nur Regeln, wie sie z.B. in der ALGOL 60-Syntax (Naur, 1963) enthalten sind, d.h. Regeln einer kontextfreien Sprache, sondern darüber hinaus auch Kontext-Bedingungen, z.B. die Eindeutigkeit der Namen, die bei der Syntaxanalyse unmittelbar geprüft werden kann und soll.

Die Semantik ordnet einer syntaktisch korrekten ESPRESO-Spezifikation eine Bedeutung zu. Dies geschieht in zwei Phasen: Durch die Grammatik (im Anhang) wird eine Abbildung von ESPRESO-S in eine interne Form angegeben, die befreit ist von allen irrelevanten Bestandteilen. Kapitel 5 ordnet dieser dann die Formulierung in einer Programmiersprache zu. Dies setzt allerdings voraus, daß keine semantischen Fehler und Widersprüche in der Spezifikation enthalten sind, was durch Prüfmittel festgestellt werden kann (5.4).

Über die Semantik hinaus enthält eine ESPRESO-Formulierung nicht formalisierte Information in Form von Namen, Stichwörtern und Texten. Auch wenn diese Information nicht mechanisch ausgewertet werden kann, hat sie doch für die Entwickelbarkeit und Lesbarkeit erhebliche Bedeutung.

3.2.1.2 Der Zeichenvorrat

Graphische Sprachen, z.B. SADT, sind bei den Anwendern sehr beliebt, werden aber in der Regel nur von Hand verwendet, denn zum Eingeben und Editieren ist ein spezielles, aufgrund der Geräte-Abhängigkeit nicht portables Graphik-System erforderlich.

Da Portabilität für ESPRESO angestrebt wird, ist der Zeichenvorrat wie bei Programmiersprachen auf die standardisierten Zeichen (einschließlich Zeilen- und Seitenvorschub) beschränkt. Die verbleibenden Möglichkeiten, eine optische Gliederung zu erzielen, sollen bei Ausgaben des Systems ausgenutzt werden; ESPRESO-S ist jedoch - mit kleinen Ausnahmen für die Texte (3.2.3.1) - formatfrei.

Eine Unterscheidung zwischen Groß- und Kleinbuchstaben, wie sie in den Beispielen dieser Arbeit stets gemacht wird, soll möglich, aber nicht notwendig sein.

3.2.1.3 Strukturelle und relationelle Beschreibung

Das geplante System muß durch seine Teile und deren Beziehungen, also durch seine Struktur, beschrieben werden. Einfache Strukturen, insbesondere Bäume, können in linearen Sprachen entweder durch syntaktische Strukturen (strukturell) oder durch explizite Angabe der Beziehungen (relationell) dargestellt werden.

Beispiel: Besetzen einer Matrix (hier in der Form von ALGOL 68).

Nach der Deklaration (1:3, 1:2) integer feld;

haben folgende Alternativen dieselbe Wirkung:

```
feld := ( (27, 15),                 feld(1,1) := 27, feld(1,2) := 15,
         (12, -7),                  feld(2,1) := 12, feld(2,2) := -7,
         ( 0,  1) );                feld(3,1) :=  0, feld(3,2) :=  1;
```

strukturelle Form — relationelle Form

Wie das Beispiel zeigt, ist die strukturelle Form knapp und hat eine stärkere graphische Komponente, die relationelle Form ist dagegen flexibel und hat eine einfachere Grammatik.

Hohe Programmiersprachen wie ALGOL 68 bevorzugen die strukturelle Darstellung: Viele Aussagen können nur strukturell gemacht werden, z.B. kann der Gültigkeitsbereich eines Namens nur durch die Anordnung in der Aufschreibung festgelegt werden, nicht durch explizite Angabe.

In ESPRESO-S gibt es ein wichtiges Argument für die relationelle Form: Die schrittweise Erstellung der Spezifikation erfordert eine flexible Lösung, bei der es erlaubt ist, alle Aussagen in möglichst beliebiger Reihenfolge zuzulassen.

Daher wird für ESPRESO-S eine Grammatik gewählt, die die strukturelle Form bevorzugt, aber die für die kumulierende Erstellung notwendige Flexibilität bietet (siehe 3.2.5).

Das folgende Beispiel zeigt die Alternativen:

strukturelle Form

```
block abc:
   ...
 parallel
    block x1:
    reads d1;
    produces b1
    end x1
 parallel
    block x2:
    consumes b1;
    writes d2
    end x2
 end abc.
```

relationelle Form

```
block abc:
   ...
 parallel
    block x1
 parallel
    block x2
 end abc;

 block x1:
 reads d1;
 produces b1
 end x1;

 block x2:
 consumes b2;
 writes d2
 end x2.
```

Es sind also beide Formen zugelassen. Jedoch ist es nicht möglich, wie z.B. in PSL eine Beziehung beliebig von der einen oder anderen Seite aus zu beschreiben, im Beispiel also etwa die Beziehung zwischen abc und x1 in der Definition von x1 statt in der von abc.

3.2.2 Objekte und Verknüpfungen

3.2.2.1 Das Graphen-Schema

Der Sprache ESPRESO-S liegt die Vorstellung zugrunde, daß sich der Gegenstand der Beschreibung darstellen läßt als ungerichteter bipartiter Graph mit den folgenden Eigenschaften:

Die Menge der Knoten zerfällt in zwei Klassen, die der Objekte und die der Verknüpfungen. Die erste Klasse zerfällt weiter in die Mengen der Objekte einzelner Arten, die zweite in die einzelnen Relationen.

Jede Kante verbindet eine Verknüpfung mit einem Objekt. Von den Verknüpfungen einer bestimmten Relation gehen stets gleichviele Kanten aus, bei einer n-stelligen Relation n Kanten, n > 0. 1-stellige Relationen werden als Attribute bezeichnet.
Nachfolgend wird unter "Verknüpfung" nicht nur der Knoten, sondern auch die Menge der von ihm ausgehenden Kanten verstanden.

Die Kanten sind markiert. Verknüpfung und Markierung bestimmen eine Kante eindeutig. Die Markierung kann als Selektor für eine bestimmte Komponente der Verknüpfung aufgefaßt werden. Alle Verknüpfungen einer Relation haben Kanten mit gleichen Markierungen.

Bei bestimmten Relationen kann auch eine Zahl an die Stelle einer Komponente treten oder eine Komponente undefiniert bleiben. Zahlen und eine Pseudo-Konstante " undef" können also als spezielle Objekte aufgefaßt werden.

Die abstrakte Syntax von ESPRESO-S restringiert die Wahl der Objekte für die Verknüpfungen, vor allem durch Vorschriften über die Arten, die die Komponenten einer Relation haben müssen.

Beispiel:

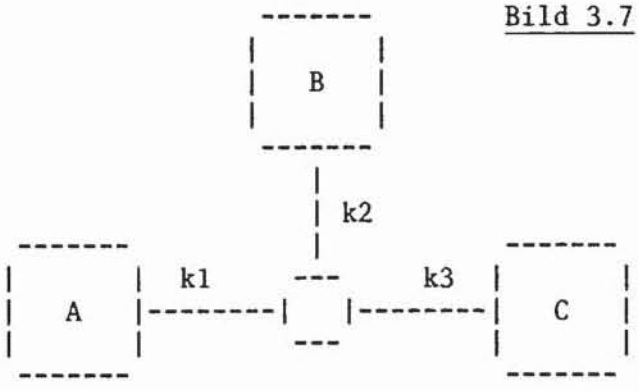


Bild 3.7 Bipartiter Graph

A, B, C sind Objekte, der Knoten dazwischen ist eine Verknüpfung.

A, B und C seien von den Arten Block, Puffer und Zusicherung.
Die Verknüpfung gehöre zur Relation 'liefert'.

Die Kanten mit den Markierungen k1, k2, k3 haben als Selektoren die Bedeutungen 'Lieferant', 'belieferter Puffer' und 'Zusicherung über die Lieferung'.

3.2.2.2 Darstellung der Objekte und Verknüpfungen

Jedem Objekt ist ein Name umkehrbar eindeutig zugeordnet. Oben wurde bereits gesagt, daß jedes Objekt eine bestimmte Art hat. Schließlich kann ein Objekt beliebig viele Texte (3.2.3.1) enthalten.

Namen und Texte sind allein keine Objekte und daher auch nicht durch Verknüpfungen mit den Objekten verbunden, sondern T e i l e von ihnen. Die Art wird formal als Attribut des Objekts aufgefaßt.

Zur Vereinfachung der Sprache und ihrer Definition wurde hier darauf verzichtet, alternative Namen für ein Objekt (" Synonyma") vorzusehen; bei einem vollen Ausbau der Sprache sollte dies nachgeholt werden.

In gewissen Fällen kann der Anwender anonyme Objekte, also Objekte ohne Namen, verwenden. Diesen sind durch ihren Kontext eindeutig bestimmte Namen zugeordnet (siehe 3.2.3.2).

Verknüpfungen haben keine Namen oder Texte. Sie sind eindeutig bestimmt durch die Relation und die Komponenten an bestimmten Selektoren, die sogenannten signifikanten Komponenten.

Die Syntax ist primär auf die Objekte gerichtet; sie werden beschrieben durch Sektionen, in denen auch ihre Verknüpfungen angegeben sind. Gewisse Sektionen, insbesondere die von Moduln und Prozeduren, sind rekursiv aufgebaut.
Das folgende Beispiel zeigt eine Sektion für ein Objekt der Art Block. Weitere solche Sektionen sind darin enthalten. In 4.1.1 wird die Syntax der Sektionen genauer dargestellt.

Beispiel:

```
block Meßwerterfassung:
text Zweck ¢ Einlesen aller Meßwerte ¢;
reads Uhrzeit;
parallel
   block Erfassung-des-Wertes-1
parallel
   block Erfassung-des-Wertes-2
parallel
   block Erfassung-des-Wertes-3
end Meßwerterfassung.
```

3.2.3 Sprachmittel zur informalen Beschreibung

3.2.3.1 Texte und Querverweise

Informale Texte spielen bei der Beschreibung von Programmen eine wesentliche Rolle, besonders in einem frühen Stadium der Entwicklung. Daher sind sie auch in ESPRESO-S vorgesehen.

Da es im allgemeinen zu einem Objekt mehrere Texte geben wird, die der Anwender einzeln eingeben, ausgeben und verändern möchte, wird jedem Text ein frei gewählter Schlüssel (Text-Selektor) vorangestellt.

Der informale Charakter der Texte schließt ihre Auswertung durch ein automatisches System aus. Eine Ausnahme bilden die in den Texten vorkommenden Namen formal definierter Objekte. Sie können als Querverweise registriert werden. In ESPRESO-S wird dies dem System dadurch erleichtert, daß solche Namen im Text durch ein vorangestelltes Sonderzeichen ("!") markiert werden müssen.

3.2.3.2 Text-Objekte

Da Texte selbst keine Objekte sind, ist es nicht möglich, sie durch Querverweise zu verbinden oder mehreren Objekten denselben Text zuzuordnen. Daher sind in ESPRESO-S Objekte spezieller Art vorgesehen, die einzig dazu dienen, Texte aufzunehmen. Im Sinne von 3.2.1.1 haben diese sog. Text-Objekte keine Semantik. Nachfolgend sind ihre wichtigsten Anwendungen beschrieben.

Die in 3.1 eingeführten Begriffe sind zur Beschreibung des geplanten Systems und seiner Schnittstellen zum technischen Prozeß und zum Bediener geeignet, soweit diese funktionell beschrieben werden können. Für globale Aussagen, z.B. über geforderte Qualitäten oder die gewählten Strukturierungsschemata, bieten diese Begriffe keine Basis. Daher werden für solche allgemeinen Angaben Text-Objekte verwendet.

Wie in 2.2.3.1 festgestellt wurde, gehört zu den Einsatzanforderungen die Beschreibung der virtuellen Maschine, auf der das geplante System laufen soll.

Sind Hardware und Basissoftware vorgegeben, so gibt es für sie auch Spezifikationen in Form von Handbüchern usw. Ein Anwender von ESPRESO wird diese Handbücher nicht in eine spezielle Spezifikationssprache übertragen, sondern wird sich vielmehr mit Verweisen auf die entsprechenden Schriften begnügen.
Wenn Hardware und Basissoftware nicht vorgegeben sind, sondern erst im Zuge des Entwicklungsprozesses ausgewählt oder entwickelt werden, ist die virtuelle Maschine selbst Gegenstand der Spezifikation und nicht von der Anwendungssoftware zu trennen.

Daher stellt ESPRESO-S für die Spezifikation der virtuellen Maschine keine speziellen formalen Mittel zur Verfügung.
Zur Beschreibung der speziellen Hardware-Konfiguration und ihres Anschlusses an Namen, die im Programm verwendet werden können, ist der in PEARL vorgesehene Systemteil gut geeignet.

Formal läßt sich in ESPRESO-S nicht ausdrücken, welchen Inhalt die Lieferung an einen Puffer hat, welcher Wert in einer Variablen erwartet wird usw. Als informaler Ersatz dienen Text-Objekte, die den Transfers und den Parameter-Deklarationen als Zusicherungen zugeordnet werden können. In dem praktisch wichtigsten Fall, daß genau ein Text benötigt wird, braucht nur dieser Text angegeben zu werden; bei der Verarbeitung wird diesem automatisch ein eindeutiger Name zugeordnet, so daß er als Text-Objekt behandelt werden kann (Beispiel siehe 3.2.5).

Text-Objekte können auch als Stichwörter verwendet werden. Sie werden dazu durch eine spezielle Relation mit den Objekten verbunden. Bei der Selektion von Objekten zur Erzeugung von Reports usw. können sie als Kriterium herangezogen werden (siehe 3.3.2.1.2).

3.2.4 Der Aufbau einer ESPRESO-Spezifikation

Nachfolgend wird zunächst beschrieben, wie eine ESPRESO-Spezifikation im einfachsten Falle dargestellt wird. Anschließend werden die für die kumulierende Erstellung notwendigen Freiheiten der Sprache zugefügt (3.2.5).

Eine vollständige ESPRESO-Spezifikation besteht aus beliebig vielen Text-Objekten (3.2.3.2) und dem Modulbaum, der alle in ESPRESO-S formalisierbaren Anforderungen enthält. Der schematische Aufbau ist im Bild 3.8 dargestellt.

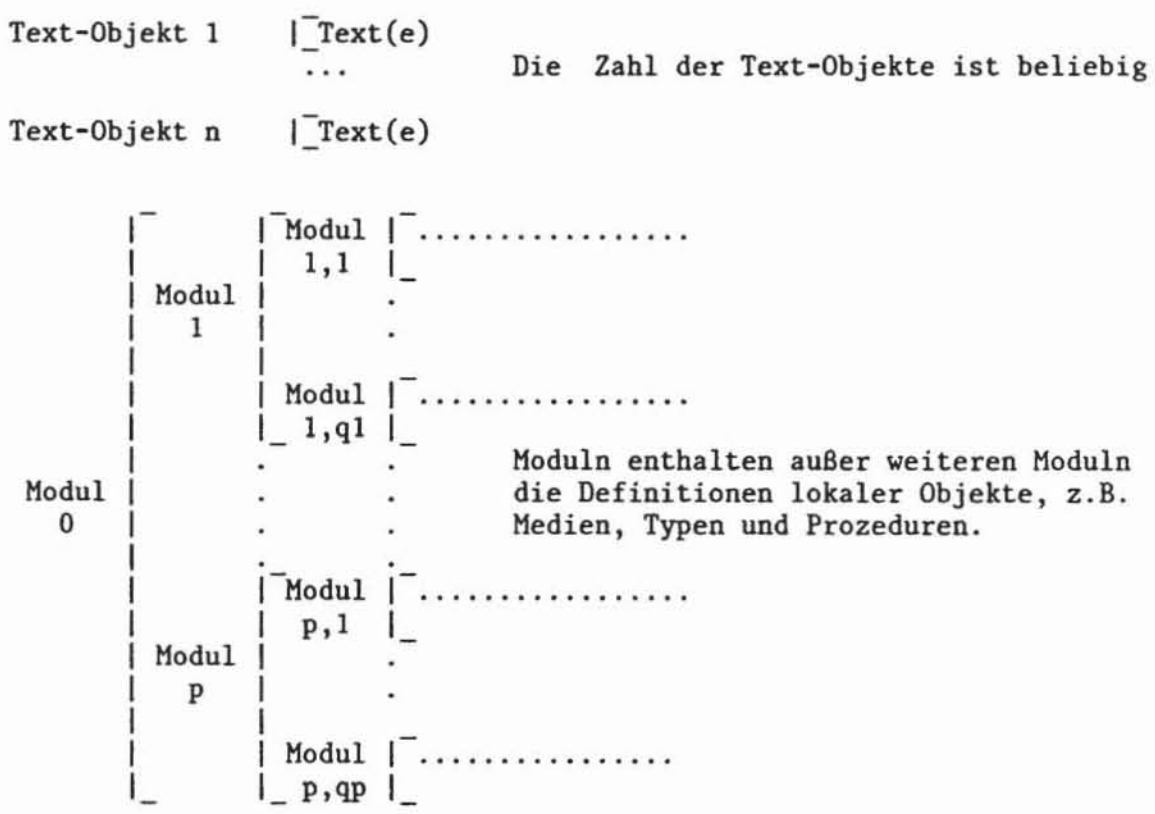


Bild 3.8 Aufbau einer ESPRESO-Spezifikation

Die in den Moduln definierten Variablen und Prozeduren werden ebenfalls hierarchisch beschrieben. Variablen können als Verbunde aus anderen Variablen aufgebaut sein, Prozeduren enthalten eingeschachtelte Blöcke (siehe Bild 3.9).

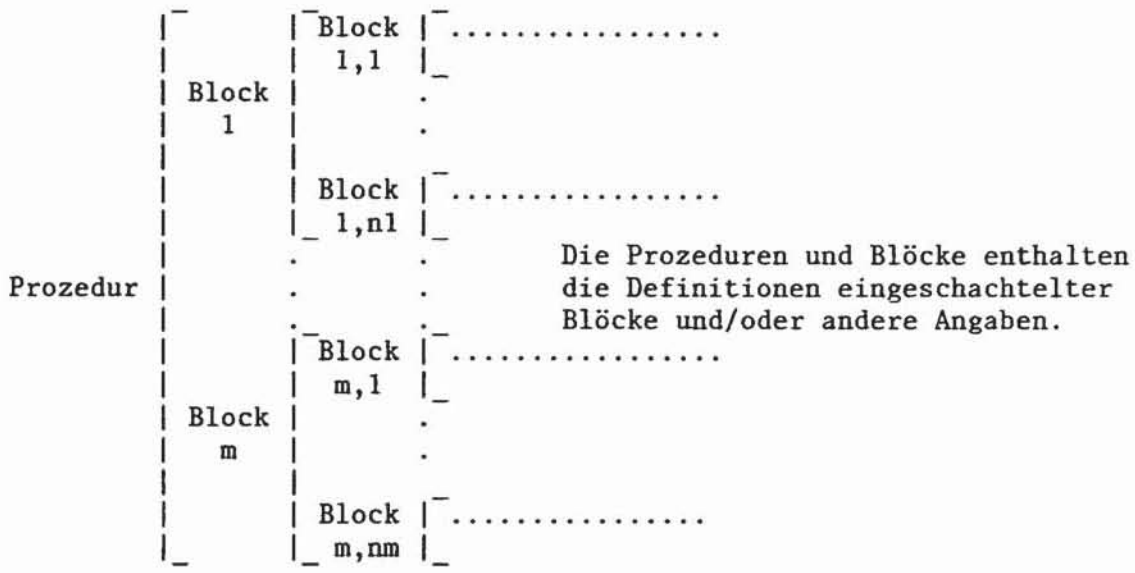


Bild 3.9 Aufbau einer Prozedur-Spezifikation

3.2.5 Lockerungen der Form

Für den praktischen Gebrauch wäre die oben beschriebene Form zu streng. Daher wird sie durch die folgenden Regeln gelockert:

a) Ein Objekt kann auch in mehreren Schritten, d.h. durch mehrere Sektionen, definiert werden.

b) Sektionen, die nicht in andere eingeschachtelt sind, werden nachfolgend als Abschnitte bezeichnet. In der beschriebenen Form bilden nur die Text-Objekte und ein einziger Modul Abschnitte; In ESPRESO-S kann dagegen jede beliebige Sektion einen Abschnitt bilden. Damit entsteht die Möglichkeit, die Festlegung hierarchischer Beziehungen zurückzustellen.

c) Die Reihenfolge der Abschnitte ist für die Semantik irrelevant.

d) In der angegebenen Form muß die Struktur vollständig sein; z.B. muß der Modul 0 stets vorhanden sein, um irgendetwas beschreiben zu können. In ESPRESO-S gilt dies nicht. Die Objekte dürfen zusammenhanglos sein oder undefiniert, d.h. nur verwendet. Allerdings ist die Vollständigkeit eine Voraussetzung zur Abbildung in die Programmiersprache (siehe 5.4).

Beispiel: Die folgende ESPRESO-Spezifikation ist syntaktisch korrekt, obwohl

- die Definition jeglicher Moduln fehlt, in die die aufgeführten Objekte einzuordnen sind,
- "Temperatur-Regelung" in zwei Sektionen definiert ist,
- alle verwendeten Variablen, Puffer und Trigger undefiniert sind.

```
procedure Temperatur-Regelung:
text Zweck: ¢ Ueberwachung der Raumtemperatur und
                Ausgabe von Stellsignalen an das Mischventil ¢;
while Sollwert-definiert;
started by Temperatur-Meßzyklus;

sequential
   block Temperatur-Messen:
   reads Raumtemperatur, Sollwert;
   writes Temperatur-Abweichung;
   end
end Temperatur-Regelung;


procedure Temperatur-Regelung:

sequential
   block Temperatur-Messen

then
   block Mischventilstellung-korrigieren:
   reads Temperatur-Abweichung;
   produces Stellpuls where ¢ nach Regelfunktion ¢
   end

end Temperatur-Regelung.
```

3.3 Das Werkzeug

Zur Prüfung, Speicherung, Auswertung und Dokumentation der Spezifikation ist ein Werkzeug, genannt ESPRESO-W, erforderlich. Seine Teile müssen auf die Sprache abgestimmt und untereinander konsistent sein, so daß der Anwender soweit wie möglich von der Verwaltung der Information entlastet wird.

Das Werkzeug soll sicherstellen, daß die mit der Sprache definierten Restriktionen vom Anwender eingehalten werden. Es soll die Aufbewahrung der Spezifikation gestatten und so eine sukzessive Vervollständigung unterstützen. Durch Mittel zur Herstellung von Kopien der gespeicherten Spezifikation soll die Aufbewahrung verschiedener Versionen oder Alternativen gefördert werden.

Der Anwender soll die Möglichkeit haben, die Spezifikation ganz oder teilweise ausgeben und unter unterschiedlichen Gesichtspunkten automatisch auswerten zu lassen. Für alle Stadien der Entwicklung müssen Prüfmittel bereitstehen.

Die Hilfsmittel müssen so komfortabel sein, daß sie die unvermeidlichen Belastungen für den Anwender gering halten und d i r e k t , d.h. nicht erst durch die Verbesserung der Produkt-Qualität, einen Ausgleich schaffen. Dadurch soll erreicht werden, daß ESPRESO vom Anwender leicht akzeptiert wird.

Abschnitt 4.2 enthält eine knappe Programmbeschreibung von ESPRESO-W; hier wird nur eine grobe und informale Spezifikation gegeben, ohne auf Details und Fragen der Realisierung einzugehen. Einzig die Speicherung der ESPRESO-Spezifikation wird schon diskutiert, da sie bis zur Schnittstelle zwischen ESPRESO-W und Anwender durchschlägt.

3.3.1 Die ESPRESO-Datei

Die Spezifikation soll schrittweise entwickelt werden. Dazu ist es notwendig, sie zu speichern, zu ergänzen und zu ändern. Die damit verbundenen Verwaltungsaufgaben soll ESPRESO-W übernehmen. Für die Speicherung bestehen folgende Möglichkeiten:

a) Der Anwender speichert die Spezifikation in einer normalen Datei und fügt - i.a. durch Editieren der Datei - alle neuen Teile zu.

b) ESPRESO-W prüft die Spezifikation, überführt sie in eine interne Darstellung und bringt sie in eine Datei (nachfolgend als ESPRESO-Datei bezeichnet), die für den Anwender völlig intransparent, also auch nicht lesbar ist.

Der erste Weg ist bei Compilern üblich, der zweite bei einigen Spezifikationssystemen. Dazwischen gibt es Kompromisse, z.B. die folgende Lösung: ESPRESO-W prüft alle Eingaben des Anwenders und speichert sie in eine interne Datei, deren Inhalt damit automatisch syntaktisch korrekt ist. Der Anwender kann diese Datei direkt lesen, aber nicht ohne Hilfe von ESPRESO-W verändern.

Für die Wahl einer Lösung sind folgende Kriterien relevant:

- Handlichkeit bei der Benutzung;
- Transparenz für den Anwender, d.h. die Möglichkeit, die Auswirkungen aller Manipulationen an der ESPRESO-Spezifikation zu überschauen;
- Möglichkeit der Prüfung einzelner Teile (und damit bei interner Speicherung auch die der Akkumulierung) einer ESPRESO-Spezifikation;
- Eignung zum Anschluß von Analyse- und Report-Funktionen;
- Effizienz und Erstellungsaufwand.

Handlichkeit und Transparenz sprechen klar für die erste Alternative; der Anwender kann direkt mit der Spezifikation arbeiten und sieht unmittelbar, was er bewirkt, wenn er entsprechende apparative Hilfen hat, z.B. ein Sichtgerät und einen Texteditor. Ist die interne Darstellung dagegen anders als die externe, so sind alle Änderungen des Datei-Inhalts mit einigem Aufwand verbunden; der geänderte Stand ist nicht unmittelbar sichtbar, sondern nur mittels der Report-Funktionen von ESPRESO-W.

Die teilweise Prüfung ist nur mit der zweiten Alternative möglich; ist eine Datei für den Anwender frei zugänglich, so kann ESPRESO-W nach irgendeiner Änderung keine Annahmen mehr machen über die Korrektheit der Spezifikation und muß erneut a l l e s prüfen.

Die Analysen können nicht auf der Grundlage der Darstellung in ESPRESO-S ausgeführt werden. Man wird in jedem Fall eine interne Repräsentation anfertigen müssen, die die logischen Verbindungen in der Spezifikation wiedergibt.

Zu Speicherbedarf und Erstellungsaufwand sind nur Vermutungen möglich. Die erste Alternative erfordert wahrscheinlich am wenigsten Speicherplatz, die zweite weniger Erstellungsaufwand, da die ESPRESO-Datei eine Standardschnittstelle zwischen allen Komponenten von ESPRESO-W bildet.

Für ESPRESO-W wird eine Kombination der beiden oben genannten Wege gewählt: Die Spezifikation wird bei der Eingabe in eine interne Darstellung, die der Benutzer nicht lesen kann, überführt (konvertiert). Diese kann ganz oder teilweise in die externe Form (ESPRESO-S) zurückgewandelt werden (dekonvertiert); was gewandelt wird, wird gleichzeitig gelöscht.

Will der Benutzer also die Spezifikation eines Objekts verändern, so dekonvertiert er sie, ändert sie in der ESPRESO-S-Form und konvertiert sie wieder. Da das Objekt in der ESPRESO-Datei gelöscht worden war, entstehen keine Inkonsistenzen.

3.3.2 Die Funktionen von ESPRESO-W

Die Funktionen von ESPRESO-W lassen sich in drei Gruppen ordnen:

- Funktionen zum Füllen einer ESPRESO-Datei
  und zur Veränderung ihres Inhalts,
- Funktionen zur Prüfung und zur Reporterzeugung,
- Funktionen zur Verwaltung der ESPRESO-Datei(en).

3.3.2.1 Funktionen, die den Inhalt der ESPRESO-Datei verändern

3.3.2.1.1 Die Konvertierung

Zur Prüfung einer ESPRESO-Spezifikation auf syntaktische Korrektheit und zur Einspeicherung in die ESPRESO-Datei dient die Konvertierungsfunktion von ESPRESO-W, kurz KONV genannt.

KONV soll folgenden Anforderungen genügen:

a) Soweit zum Zeitpunkt der Eingabe die ESPRESO-Datei bereits besetzt ist, soll KONV das gleiche Verhalten zeigen, als würden alle früher eingegebenen ESPRESO-Spezifikationen mit der neuen konkateniert und zusammen in eine leere ESPRESO-Datei gespeichert.

b) Ein Fehler in einem Abschnitt darf keine Rückwirkungen auf bereits konvertierte Abschnitte haben. Nach der Verarbeitung eines Abschnitts muß KONV also wieder im Anfangszustand sein. Im übrigen wird hier kein Aufwand zur Fehlerbehandlung betrieben (siehe 4.2.1).

c) Die wiederholte Eingabe derselben ESPRESO-Spezifikation soll keine Veränderung der ESPRESO-Datei bewirken und dieselben Fehlermeldungen hervorrufen. Diese Regelung befreit den Benutzer vom Zwang, bei jeder Eingabe zu prüfen, ob sie wirklich völlig neu ist.
   Schwierigkeiten entstehen bei Texten. Es wird festgelegt, daß ein Text ignoriert wird, wenn zum selben Objekt bereits ein Text mit demselben Selektor gespeichert ist, unabhängig vom Inhalt des alten und des neuen Textes.

Wie oben gesagt "verbraucht" KONV die Eingabe, indem es alle transformierten Teile darin löscht, was beim Auftreten eines Fehlers praktisch ist; nach fehlerfreier Eingabe ist also die Eingabedatei leer. Aus Sicherheitsgründen geschieht dies aber nicht in der eigentlichen Quelle, sondern in einer Kopie davon.

Wird KONV durch ein äußeres Ereignis vorzeitig beendet, so bleibt der Inhalt der ESPRESO-Datei unverändert.

KONV kann auch zur Syntaxprüfung ohne Veränderung in der ESPRESO-Datei verwendet werden.

3.3.2.1.2 Die Dekonvertierung

Zur Rücktransformation von der internen Form in die Sprache ESPRESO-S dient die Dekonvertierungsfunktion von ESPRESO-W, kurz DEKONV genannt.

- DEKONV wählt nach den Wünschen des Anwenders Verknüpfungen und Objekte aus der ESPRESO-Datei aus,
- wandelt diese von der internen Form in die Sprache ESPRESO-S um, wobei größtmögliche Übereinstimmung mit der in 3.2.4 beschriebenen Form angestrebt wird, und schreibt sie in eine vom Benutzer gewählte Datei,
- löscht die angegebenen Objekte und Verknüpfungen in der ESPRESO-Datei.

Die Selektion kann gesteuert werden durch Vorgabe der Objektnamen, der Arten oder von Stichwörtern.

Bei Objekten, die die Definitionen anderer Objekte einschließen können, z.B. Moduln, kann gewählt werden, ob nur das Objekt selbst oder das von ihm repräsentierte Subsystem dekonvertiert werden soll.

In der ESPRESO-Datei werden die gewählten Objekte und alle ihre Verknüpfungen gelöscht; andere Objekte, die nur ausgegeben wurden, um alle Verknüpfungen darzustellen, werden nicht gelöscht und erhalten in der Ausgabe eine entsprechende Kennzeichnung. Auf Anforderung des Anwenders werden alle Texte dekonvertiert und in der ESPRESO-Datei gelöscht, die Querverweise auf die auszugebenden Objekte enthalten. Andernfalls wird ein Hinweis auf diese Texte ausgegeben, die als Querverweise vorkommenden Namen werden nicht gelöscht.

Wird DEKONV durch ein äußeres Ereignis vorzeitig beendet, so bleibt der Inhalt der ESPRESO-Datei unverändert.

3.3.2.1.3 Die Änderung eines Objektnamens

Die Änderung eines Objektnamens mittels DEKONV und KONV wäre prinzipiell ohne weiteres möglich, aber umständlich und fehlerträchtig, da der neue Name an vielen Stellen den alten ersetzen müßte. (Ein Texteditor könnte diese Aufgabe allerdings sehr vereinfachen.) Daher wird zu diesem Zweck eine spezielle Funktion NAMAE zur Verfügung gestellt. Diese ersetzt den alten Namen überall, auch in Querverweisen, durch einen neuen.

3.3.2.2 Funktionen zur Prüfung der ESPRESO-Spezifikation auf semantische Korrektheit und zur Reporterzeugung

Prüfung und Reporterzeugung lassen sich weder logisch noch praktisch trennen, denn einerseits muß für jede Prüfung ein spezieller Report erzeugt werden, andererseits dient jeder Report auch oder ausschließlich der Kontrolle durch den Anwender.

3.3.2.2.1 Ausgabe der ESPRESO-Datei in ESPRESO-S

Dieser Report entspricht der Dekonvertierung mit dem Unterschied, daß der Inhalt der ESPRESO-Datei unverändert bleibt.

3.3.2.2.2 Prüfung auf Vollständigkeit der Definition

Es werden alle Objektnamen ausgegeben, die zu bisher nicht definierten Objekten gehören, also bisher nur in Verknüpfungen oder Querverweisen vorgekommen sind.

3.3.2.2.3 Prüfung und Ausgabe der Modulhierarchie

Durch die Syntax ist sichergestellt, daß die Moduln einen Wald bilden. Dieser wird dargestellt. Damit wird dem Anwender sichtbar, welche Moduln noch in die Hierarchie eingeordnet werden müssen, um schließlich einen einzigen Baum zu erhalten.

3.3.2.2.4 Prüfung und Ausgabe der Prozeduren-Struktur

Die Aufrufbeziehungen zwischen den Prozeduren werden in Matrixform dargestellt; sofern keine Rekursivität vorkommt, werden dabei die Prozeduren so geordnet, daß eine Dreiecksmatrix entsteht, in der auch erkennbar ist, ob die Prozeduren einen zusammenhängenden Graphen bilden.

3.3.2.2.5 Prüfung der Gültigkeitsbereiche

Für alle Medien wird festgestellt, ob ihre Gültigkeitsbereiche definiert und, soweit dies der Fall ist, ob alle Zugriffe zulässig sind. Bei den Prozeduren wird entsprechend die Legalität der Aufrufe geprüft.

3.3.2.2.6 Prüfung von Datenflüssen auf Vollständigkeit

Bei allen Variablen wird geprüft, ob sie initialisiert und gelesen werden. (Es wird nicht festgestellt, ob die Initialisierung dynamisch vor dem ersten lesenden Zugriff liegt!) Dabei werden Zerlegungen der Daten berücksichtigt, d.h. es wird z.B. keine Unvollständigkeit gemeldet, wo zwar nicht die Variable insgesamt, aber jedes ihrer Teile gelesen wird.

Bei Puffern wird geprüft, ob sie sowohl gefüllt als auch geleert werden, bei Betriebsmitteln und Prozeduren, ob sie jemals belegt bzw. aufgerufen werden.

3.3.2.2.7 Schnittstellen zwischen Teilsystemen

Teilsysteme können definiert werden durch die Kennzeichnung von Objekten mit Stichwörtern. Der Report enthält alle Beziehungen zwischen einem solchen Teilsystem und seiner Umgebung (Datenfluß, Aufrufe usw.).

3.3.2.2.8 Prüfungen zur geschachtelten Belegung von Betriebsmitteln

Werden in geschachtelten Blöcken oder untergeordneten Prozeduren Betriebsmittel belegt, so kann dies unter gewissen Voraussetzungen zu Verklemmungen führen. Falls diese Voraussetzungen gegeben sind, werden Warnungen erzeugt.

3.3.2.2.9 Prüfungen des dynamischen Verhaltens

Das dynamische Verhalten ist bei der meist zeitkritischen Prozeßrechnerprogrammierung von besonderer Bedeutung. In ESPRESO-S lassen sich die meisten für die Dynamik relevanten Informationen wie Zeitbedarf, Ausführungssequenz und gegenseitige Beeinflussung der Operationen angeben. Es besteht damit die Voraussetzung zum Anschluß eines Simulationssystems, bei dem die Verzweigungen interaktiv oder durch Zufallsentscheidungen gesteuert werden.

3.3.2.3 Funktionen zur Verwaltung der ESPRESO-Dateien

Zur Verwaltung der Dateien enthält ESPRESO-W folgende Funktionen:

EED: Einrichten einer ESPRESO-Datei.

LED: Löschen einer ESPRESO-Datei.

EDB: ESPRESO-Datei auf Band kopieren (Archivierung).

BED: Band in ESPRESO-Datei kopieren (Restaurierung).

DUPL: Duplizieren einer ESPRESO-Datei
(z.B. zur Entwicklung verschiedener Varianten).

INF: Ausgabe der Verwaltungsinformation zu einer ESPRESO-Datei.

KAT: Ausgabe eines Katalogs aller ESPRESO-Dateien.

3.4 Das Verfahren

Da ESPRESO von Beginn an nicht als methodisch neutral konzipiert wurde wie einige andere Systeme, z.B. EPOS (Biewald et al., 1979), sind in die Begriffe, die Sprache und das Werkzeug sehr viele methodische Überlegungen eingeflossen. Es bleibt daher in 3.4 nur die Anleitung, wie die in 3.1 bis 3.3 konzipierten Elemente des Systems zu verwenden sind.

Die Anleitung hat nicht den Charakter eines Kochbuchs; dieses beliebte Bild ist irreführend, weil das Kochbuch in der Regel nur Anweisungen enthält, nach denen man Produkte reproduzieren, nicht neue entwickeln kann. Genau das aber ist die Problemstellung des Software-Engineering.

3.4.1 Sammlung von Anforderungen

ESPRESO ist auf die Situation ausgerichtet, daß ein vorhandenes oder geplantes System durch Komponenten zur Prozeßdatenverarbeitung ergänzt wird oder daß solche Komponenten ersetzt werden. Im ersten Schritt muß daher die - meist informale - komplementäre Spezifikation (2.2.3.1) erstellt werden. Diese wird so weit wie möglich vervollständigt durch Aufnahme aller verfügbaren Information über die bereits definierten, eventuell schon vorhandenen Systemkomponenten.

Die in dieser Phase anfallende Information wird in Text-Objekten dargestellt und gespeichert. Durch die Querverweise in den Texten können bereits auf dieser Ebene Beziehungen ausgedrückt und verfolgt werden.

3.4.2 Die Erstellung der direkten Spezifikation

Aus der komplementären Spezifikation wird die direkte entwickelt, der Spezifikations- und Entwurfsprozeß verläuft also von außen nach innen, "outside-in". Gebräuchlicher als "outside-in" ist die eng verwandte Charakterisierung eines Entwurfsprozesses als "top-down". Der erste Begriff ist für Prozeßrechner-Programme besser geeignet, da die Spezifikation im allgemeinen nicht ein "top" darstellt, eine logisch abgeschlossene Aufgabe auf einem einzigen Abstraktionsniveau, sondern auf vielen Ebenen möglicherweise sehr verschiedene Anforderungen enthält (vgl. Punkt 5 in Parnas, 1974, S.338).

ESPRESO erfordert, daß die Formulierung der Spezifikation mit einer Strukturierung des Problems einhergeht. Im einzelnen werden die folgenden Arbeiten durchgeführt:

- Das Problem wird in logisch schwach gekoppelte Bereiche, die Moduln, gegliedert (3.4.2.1).

- Innerhalb der Moduln werden Medien und aktive Komponenten mit den Aktionen, d.h. den Zugriffen auf die Medien, definiert (3.4.2.2).

Diese beiden Verfeinerungen sind miteinander verzahnt. Wo neue Objekte definiert werden, ist jeweils zu prüfen, ob sie dem ganzen Modul zur Verfügung stehen sollen. Andernfalls werden Submoduln eingeführt, denen die Objekte zugeordnet werden können.

3.4.2.1 Die Modularisierung

Die Modularisierung soll dem Prinzip des "information hiding" (Parnas, 1972; Koster, 1977) folgen. Alle Daten sollen also in Operationen eingehüllt und nur durch diese den Prozeduren außerhalb des Moduls zugänglich sein.

Bei Prozeßrechner-Programmen, deren Kommunikation mit der Umgebung eine besonders wichtige Rolle spielt, ist zu beachten, daß auch Prozeßdaten oder die Kommandos des Bedieners in dieses Konzept passen. Praktisch bedeutet dies z.B., daß sich kein Rechenprozeß unmittelbar an den Bediener wendet, sondern dessen Eingabe wie ein Datum behandelt, auf das er über den dafür vorgesehenen Mechanismus, also über ein Dialogprogramm, zugreift. Damit braucht der Rechenprozeß keine Kenntnis zu haben von der Eingabesprache, den Fehlermeldungen usw.

In der entstehenden, durch die Datenabstraktion bestimmten Struktur spielen die apparativen Grenzen, z.B. die zwischen technischem Prozeß und Prozeßrechner oder zwischen Eingabegerät und Zentraleinheit, keine wesentliche Rolle.

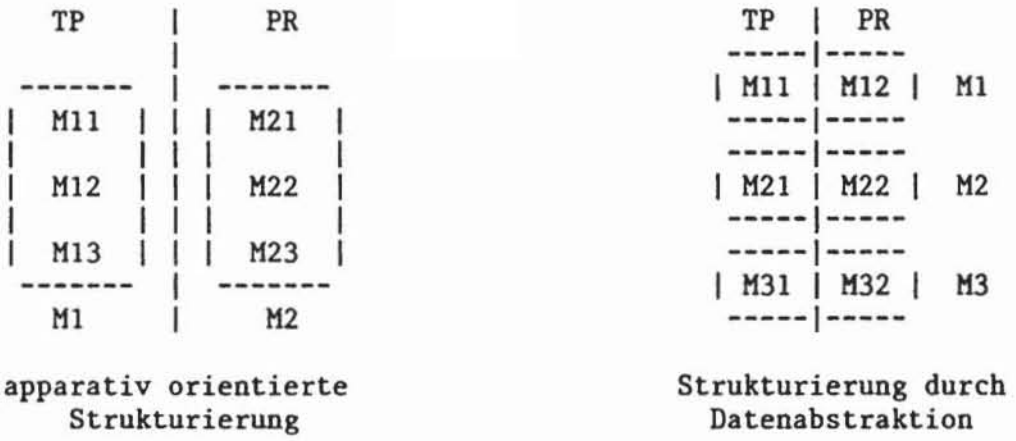


Bild 3.10 Alternative Strukturierungen

In diese Richtung zielen auch die Bewertungsmaßstäbe für Kohärenz (intramodulare Verknüpfung) und Kopplung (intermodulare Verknüpfung), die Bestandteile von Structured Design sind (siehe 2.3.2).

3.4.2.2 Die Prozeduren und Medien

Die aktiven Komponenten werden verfeinert, bis die Beschreibung vollständig ist. Dabei werden die Aktionen teilweise in die neuen terminalen Rechenvorschriften verlagert. Variablen können ebenfalls verfeinert werden. Außerdem sind die Typen von Variablen, Puffern und Parametern und die Verzögerungen und Zykluszeiten der uhrgesteuerten Trigger zu definieren.

Durch die gleichartige Behandlung sequentieller und paralleler Abläufe in ESPRESO-S gibt es anders als in den meisten Programmiersprachen keinen Druck auf den Anwender, schon frühzeitig Tasks oder Prozesse zu definieren. Er kann daher diese Entscheidungen bis zur Implementierung aufschieben und dann die nach Rechner-Konfiguration und Randbedingungen optimale Prozeßstruktur wählen (siehe 3.4.3.1). Diese Eigenschaft von ESPRESO ist besonders wichtig, da es heute möglich ist, Mikroprozessoren für Prozeßführungsaufgaben einzusetzen. Daher sollten Spezifikation und Entwurf jegliche Parallelität, die im Problem vorhanden ist, bewahren und dadurch die Möglichkeit offenlassen, das Programm auf einer beliebigen Konfiguration zu realisieren.

Die den Datenstrukturen folgende Strukturierung der Prozeduren (Jackson Design Methodology, siehe 2.3.2) ist für Prozeßrechnerprogramme aufgrund der besonderen Bedeutung der Dynamik ungeeignet. Bei Programmen für die Stapelverarbeitung, z.B. bei einem Lohnabrechnungssystem, stehen die Daten zu Beginn des Laufs vollständig bereit, der Lauf ist nach definierter Zeit beendet. Damit besteht die Möglichkeit, die Daten völlig statisch zu behandeln und das Programm als Sequenz von Datentransformationen aufzufassen, wie es in der Jackson Design Method geschieht. Die Beziehungen zwischen Ein- und Ausgabe sind ihrem Wesen nach statisch; nur die Funktionsweise des Rechners, also die L ö s u n g des Problems, fügt einen dynamischen Aspekt hinzu.

Prozeßrechner-Systeme erhalten dagegen ihre Daten erst während des Laufs, der meist nicht a priori zeitlich begrenzt ist; die Reihenfolge der Ereignisse, zu denen die Eingabe gehört, ist selbst eine wichtige Information für den Prozeßrechner. Daher ist die an sich anzustrebende statische Betrachtungsweise (vgl. Gries, 1976, S.243, Punkt 1) für Prozeßrechner-Programme nicht adäquat, weil bei dieser die Dynamik nicht erst durch die Realisierung zugefügt, sondern schon durch die Aufgabenstellung vorgegeben ist.

3.4.3 Abschluß der Spezifikation

Die in 3.4.2 beschriebenen Schritte werden mit Unterstützung durch die Prüf- und Reportfunktionen kontrolliert und, wo nötig, korrigiert oder fortgesetzt, bis eine Spezifikation entstanden ist, die, soweit dies festgestellt werden kann, vollständig und widerspruchsfrei ist. Die Abstraktionsebene, bis zu der man verfeinert, hängt davon ab, welche Zielsprache verwendet werden soll. Hat man die Möglichkeit, eine hohe Programmiersprache wie ALGOL 68 einzusetzen, so kann der Übergang vor dem Entwurf von Ablauf- und Datenstrukturen wie z.B. Schleifen und Verbunden erfolgen, die ja im Programm-Code noch ausgezeichnet darstellbar sind. Ist man dagegen gezwungen, eine primitive Sprache zur Codierung zu verwenden, so sollte der Wechsel erst wesentlich später erfolgen.

Die Spezifikation muß spätestens jetzt mit dem Auftraggeber endgültig abgestimmt werden.

3.4.3.1 Vereinfachung der Prozeßstruktur

In den meisten Fällen ist es im Interesse der Effizienz sinnvoll, oft auch Vorbedingung der Realisierung mit einer bestimmten Programmiersprache oder einem bestimmten Betriebssystem, daß die Notwendigkeit zur dynamischen Generierung von Prozessen beseitigt wird und stattdessen eine feste Anzahl von Prozessen ununterbrochen existent ist. Soll die Realisierung auf einem Monoprozessor oder einem andern System erfolgen, das wesentlich weniger Prozessoren bereithält, als parallele Prozesse nötig sind, so kann es außerdem vorteilhaft sein, kollaterale Ausdrücke in sequentielle zu wandeln und dadurch Prozesse einzusparen.

Eine solche Modifikation erfolgt in drei Schritten:

a) Unterscheidung zwischen Parallelität und Kollateralität und Wandlung der letzteren in eine sequentielle Struktur, soweit sinnvoll,

b) Ausschluß der rekursiven oder iterativen Prozeßgenerierung,

c) Ausschluß der dynamischen Prozeßgenerierung überhaupt.

Unter Kollateralität wird hier die Eignung zur parallelen Ausführung verstanden, so wie in ALGOL 68. Der Implementierer muß hier solche parallelen Familien identifizieren, die aus logischen Gründen oder im Interesse der Effizienz asynchron realisiert werden müssen; alle andern kann er in sequentielle Familien umformen.

Kann durch die Umformung nach (a) oder eine Änderung der Struktur (b) erreicht werden, daß die Höchstzahl der parallelen Prozesse statisch feststeht, so ist die wesentliche Voraussetzung für (c) gegeben. Das System wird so umgeformt, daß alle Prozesse nur einmal, nämlich zum Zeitpunkt des Programmstarts, kreiert werden und anschließend ständig existent bleiben. Durch Interaktion wird dabei erreicht, daß die Prozesse nur in den vorgesehenen Zeiten wirklich aktiv sind.

Für alle praktischen Probleme reicht diese vereinfachte Struktur aus (Brinch-Hansen, 1975, Punkt E; Nehmer, Goos, 1978, S.24).

3.4.3.2 Übergang zur Zielsprache

Schließlich erfolgt der Übergang zum Code der Zielsprache, mittels eines Transformationssystems (siehe 5.) oder, wenn ein solches nicht zur Verfügung steht, von Hand.

Es ist auch möglich, eine andere Sprache zwischenzuschalten, wie es z.B. bei SADT mit FP2 versucht wurde, um den schwierigen Sprung zum Code in zwei Schritte zu zerlegen. Allerdings muß bezweifelt werden, ob normale Anwender, die meist schon einer einzigen Spezifikationssprache skeptisch gegenüberstehen, bereit sind, sich in mehrere einzuarbeiten und sie auch zu benutzen.

4. Realisierung von ESPRESO

Dieses Kapitel enthält die Konkretisierung der Sprache und des Werkzeugs, die im Kapitel 3 beschrieben wurden.

4.1 Definition der Sprache ESPRESO-S

Ein wesentlicher Unterschied zwischen ESPRESO und andern, in der Literatur beschriebenen Spezifikationssystemen ist die vollständige formale Beschreibung der Sprache ESPRESO-S und ihrer Umsetzung in die ESPRESO-Datei. Sie schützt Implementierer und Anwender vor Unklarheiten über die Korrektheit der Eingabe und über die durch eine korrekte Eingabe bewirkten Veränderungen in der ESPRESO-Datei.

Eine Erweiterung dieser Definition zur Spezifikation anderer Operationen auf die ESPRESO-Datei, z.B. der Dekonvertierung oder der Erzeugung eines Reports, ist möglich, so daß der größte Teil des Systems formal spezifiziert wäre.

Die formale Definition ist als Anhang (Kap.7) beigefügt; 4.1.3 enthält dazu einige allgemeine Bemerkungen. 4.1.1 faßt die Beschreibung der Syntax in 3.2.4 zusammen, 4.1.2 enthält tabellarische Zusammenstellungen der Arten, Attribute und Relationen.

4.1.1 Abschnitte und Sektionen

Eine Sektion ist die Beschreibung eines Objektes, des Subjekts dieser Sektion, und seiner Beziehungen zu anderen Objekten. Ist sie nicht in eine andere Sektion eingeschachtelt, so wird sie auch als Abschnitt bezeichnet. Eine ESPRESO-Spezifikation besteht aus einer durch einen Punkt abgeschlossenen Folge von Abschnitten, die jeweils durch ein Semikolon voneinander getrennt sind:

```
< Abschnitt 1>;
< Abschnitt 2>;
      .
      .
< Abschnitt n>.
```

Eine Sektion beginnt mit dem Sektionskopf, in dem Art und Name des Subjekts genannt werden. Danach kann ein Sektionsrumpf folgen, der mit einem Doppelpunkt beginnt und beliebig viele Angaben enthält. Ihm muß der Sektionsschwanz, bestehend aus end und dem Namen des Subjekts, folgen. Beispiel:

```
module Datenkontrolle:
   <Angabe 1>;
   <Angabe 2>;
      .
      .
   <Angabe m>
end    Datenkontrolle
```

Durch eine Angabe kann

- ein Text zu dem Subjekt definiert werden, z.B.
  text Zweck ¢ prüft die Daten auf Plausibilität ¢

- das Subjekt ein Attribut erhalten, z.B.
  by-priority (für ein Objekt der Art Puffer)

- eine Verknüpfung mit andern Objekten beschrieben werden:
  reads Var1 (für ein Objekt der Art Prozedur oder Block)

  Das andere Objekt wird je nach der Relation, zu der die Verknüpfung gehört, nur erwähnt (wie im vorstehenden Beispiel) oder ist seinerseits Subjekt einer eingeschachtelten Sektion:

```
comprises type Meßwerte-Block:
                   structure-of Meßstelle, Meßwert
           end  Meßwerte-Block
```

4.1.2 Tabellen der Arten, Attribute und Relationen

Tabelle 4.1 nennt die in ESPRESO-S zugelassenen Arten und die Sammelbezeichnungen, die im Text vielfach verwendet werden.

Tabelle 4.1 Die Arten der Objekte

Art	Bezeichnung in der Grammatik	Sammelbezeichnung
Text Objekt	informal	
Modul	module	
Prozedur Block	procedure block	aktive Komponenten
Eingabe-Parameter Ausgabe-Parameter Trans Parameter	inpar outpar transpar	Parameter
Variable Puffer Trigger Betriebsmittel	variable buffer trigger resource	Medien
Typ Frist Konstante	type interval constant	

Tabelle 4.2 enthält jeweils eine Zeile für eine Attribut-Auswahl. Die eingeklammerten Attribute sind Standard-Werte, die in ESPRESO-S nicht explizit angegeben werden können.

Die Art ist als Attribut bereits implizit in den Tabellen enthalten: Jedem Objekt ist eine Art als Attribut zugeordnet. Die Alternativen sind die Arten (siehe Tabelle 4.1).

Tabelle 4.2 Attribute

Nr	Attribut-Auswahl	zugelassen für Objekte d. Arten	alternative Attribute (zu den Zahlen siehe unten)
1	art	(siehe oben)	
2	ablauf	prozedur, block	sequentiell (1), parallel (2), alternativ (3), aufruf (4)
3	struktur	variable, typ	typ (5), feld (6), verbund (7), zeiger (8), aufzählung (9)
4	überlaufverh.	puffer,trigger	blockieren, verlust
5	organisation	puffer	schlange, prioritätsabhängig
6	position	prozedur, block, variable, puffer, trigger, betriebsmittel	(innen), grenze, außen
7	verwendbar	prozedur	(einfach), mehrfach

Tabelle 4.3 enthält jeweils eine Zeile für eine mehrstellige Relation oder für mehrere gleichartige Relationen (getrennt durch Schrägstriche), jeweils eine Spalte für jede Art. Die Ziffern in der Tabelle geben an, welche Arten in den verschiedenen Komponenten der Relation zulässig sind. Komponente 1 ist das Subjekt.

Einige der Relationen (r1 bis r5 und r31 bis r36) implizieren für das Subjekt bestimmte Attribute für Ablauf oder Struktur (Auswahl 2 bzw. 3). Diese Werte sind in Tabelle 4.3 als Zahlen in Klammern angegeben. Sie beziehen sich auf die entsprechenden Zahlen in Tabelle 4.2.

Die Spalte "Parameter" faßt Eingabe-, Ausgabe- und transiente Parameter zusammen. Die Relationen r8 bis r10 gelten jeweils nur für eine dieser Arten.

Anstelle einer Konstanten kann, wo diese nicht Subjekt ist, auch eine Zahl stehen.

Die letzte Spalte gibt an, welche Komponenten signifikant sind (siehe 3.2.2.2). Die mit "*" gekennzeichneten Komponenten können auch weggelassen werden.

Tabelle 4.3 Art Relationen

Relation Nr.	Bedeutung (Attr.)	Modul	Prozedur	Block	Parameter	Variable	Puffer	Trigger	Betriebsmittel	Typ	Frist	Konstante	Text-Objekt	signifikante Kompon.
01	Modul enthält	12	2	2		2	2	2	2	2	2	2		2
02	sequent. zerlegt (1)		1	12 *3										2
03	parall. zerlegt (2)		1	12										2
04	altern. zerlegt (3)		1	12								*3		2
05	hat Schlüssel (3)		1	1		2								1
06	ruft auf (4)		12	1										1
07	deklariert Parameter		1		2								*3	2
08	Eingabeparameter ist		1	1	2	3	3					3		12
09	Ausgabeparameter ist		1	1	2	3	3							12
10	trans. Parameter ist		1	1	2	3								12
11	hat Bedingung		1	1		2								1
12/13	liest/schreibt		1	1		2							*3	12
14	ändert		1	1		2							*3	12
15	initialisiert		1	1		2	2	2					*3	12
16/17	liefert/holt		1	1			2	2					*3	12
18	testet		1	1			2	2					*3	12
19	sperrt (maskiert)		1	1			2	2						12
20/21	beendet/gestart. von		1	1				2						12
22	belegt		1	1		*3			2			*3		12
23	verfügbar in	2	1						1					12
24	hat Priorität		1									2		1
25	dauert		1	1		3					12	3		1
26	hat Verzögerung		1	1		3		1			2	3		1
27	hat Periode					3		1			2	3		1
28/29	lesbar/schreibbar in	2				1								12
30	ist Verbund aus (7)					12								2
31	ist Zeiger auf (8)					1				12				1
32	Verbund-Typ aus (7)									12				12
33	ist Feld aus (6)					13				12		3		1
34	hat Werte (9)					1				1		2		12
35	hat Typ (5)				1	1	1			2		1		1
36/37	leerbar/füllbar in	2					1	1						12
38/39	kann starten/beenden	2						1						12
40	maskierbar von	2					1	1						12
41	hat Kapazität						1	1	1			2		1
42/43	stat./dyn. Sp.bedarf						1					2		1
44	hat Stichwort	1	1	1	1	1	1	1	1	1	1	1	12	12
45	hat Verweis	12	12	12	12	12	12	12	12	12	12	12	12	12

4.1.3 Zur Darstellung der Syntax durch eine Attribut-Grammatik

Entsprechend 3.2.1.1 wird unter Syntax die Menge aller Regeln verstanden, die festlegen, welche Zeichenreihen von ESPRESO-W ohne Fehlermeldung akzeptiert werden. Ausgenommen sind nur einige durch die Implementierung gesetzte Grenzen, z.B. die Zahl der möglichen Objekte oder die maximale Länge ihrer Namen.

Zur Definition einer solchen Syntax bieten sich Attribut-Grammatiken an. Sie ermöglichen nicht nur die formale Beschreibung der kontextsensitiven Syntax, sondern bieten darüber hinaus zwei Vorteile:

- Der Leser kann mühelos die der Sprache zugrundeliegende kontextfreie Grammatik erkennen, umgekehrt ist die Konstruktion der kontextsensitiven Syntax auf der Grundlage der kontextfreien einfach.

- Es erfordert nur geringen Mehraufwand, um die Grammatik so zu formulieren, daß sie nicht nur angibt, welche Eingabe korrekt ist, sondern daß außerdem das Attribut "Kontext" (siehe 7.1.3) exakt den Inhalt der ESPRESO-Datei repräsentiert.

Van-Wijngaarden-Grammatiken (Wijngaarden et al., 1975; Hesse, 1976) sind bezüglich ihrer formalen Klarheit überlegen, aber schwerer zu konstruieren und zu lesen (Marcotty et al., 1976).

Die Grammatik im Anhang hat die Form der sogenannten "extended attribute grammars", wie sie in Watt, Madsen (1977) und Watt (1979) beschrieben sind. Sie erlaubt eine besonders knappe Formulierung.

Durch die Möglichkeiten, die Grammatik zu speichern, zu editieren und - nicht zuletzt - auch sauber auszudrucken, konnte sie über einen Zeitraum von fast einem Jahr geprüft und korrigiert werden. Dadurch ist die Zahl der unentdeckten Fehler sicher sehr viel geringer, als es ohne automatische Hilfsmittel möglich gewesen wäre. Der praktische Wert einer solchen Grammatik hängt aber wesentlich von ihrer Korrektheit ab. Ein System zur Textverwaltung ist also für die Anwendung solcher Techniken notwendig (vgl. Marcotty et al., 1976, S.273).

4.1.4 Formale Eigenschaften von ESPRESO-S

Um die Implementierung von ESPRESO-W so einfach wie möglich zu halten, wurde darauf geachtet, daß ESPRESO-S sehr einfach analysierbar ist. Dies ist durch zwei Eigenschaften gewährleistet:

- Die kontextfreie Grammatik definiert eine LL(1)-Sprache (Knuth, 1971, S.102 ff.).

- Die kontextsensitive Grammatik läßt sich in einem Durchlauf, der von links nach rechts und top - down erfolgt, analysieren.

Da die Grammatik linksrekursive Produktionen enthält und solche, bei denen mehrere Alternativen gleich beginnen, sind gewisse Umformungen nötig, um die LL(1)-Eigenschaft sichtbar werden zu lassen; an den Attributen läßt sich aber ablesen, daß die dadurch entstehenden Unterschiede der Strukturbäume für den Inhalt der ESPRESO-Datei ohne Bedeutung sind.

Beispiel: Der Anhang (7.4.3) enthält die Produktion (ohne Attribute):

```
< Verbund > ::= consists-of-Symbol < Verbundvariable >
              | < Verbund > and-Symbol < Verbundvariable >.
```

Durch Einführung einer Hilfsvariablen H wird daraus:

```
< Verbund > ::= consists-of-Symbol < Verbundvariable > < H >.
< H >       ::= leer  | and-Symbol < Verbundvariable > < H >.
```

In dieser Form ist die Syntax vom Typ LL(1).

Die zweite Eigenschaft kann geprüft werden durch Inspektion aller Attribute, die in den Produktionen vorkommen. Eine Produktion hat (ohne Alternativen) die Form

$$X_0 ::= X_1 X_2 \ldots X_n$$

In dieser Grammatik gilt, daß alle vorgegebenen* Attribute von X_i für $0 < i <= n$ völlig bestimmt sind durch die vorgegebenen von X_0 und die erzeugten von X_1 bis X_{i-1}. Die erzeugten von X_0 ergeben sich aus den vorgegebenen von X_0 und den erzeugten von X_1 bis X_n. Auch die in der Grammatik vorkommenden logischen Ausdrücke mit Attributen können in dieser Sequenz ausgewertet werden.

Damit erfüllt ESPRESO-S die Bedingungen der 1-Pass-Analysierbarkeit (Bochmann, 1976, S.59) und liegt in der Hierarchie der attributierten Grammatiken weit unten (Kastens, 1978, Fig.6), ist also sehr einfach.

---

* Für "inherited" wird hier "vorgegeben", für "synthesized" "erzeugt" gesagt. Die Übersetzungen "erworben" und "abgeleitet" (Kastens, 1979, 7.1, S.11) scheinen irreführend.

4.2 Implementierung von ESPRESO-W

Realisiert wurden die wichtigsten Teile von ESPRESO-W, also die Programme KONV und DEKONV (3.3.2.1) und die Funktionen zur Verwaltung der ESPRESO-Dateien (3.3.2.3). KONV und die ESPRESO-Datei-Verwaltung wurden im Rahmen einer Diplomarbeit implementiert (Eckert, 1980).

4.2.1 Realisierungskonzepte

Für ESPRESO-W wurden folgende Programmqualitäten angestrebt (vgl. 2.2.3.2):

a) Es soll leicht zu bedienen und transparent sein, d.h. ein dem Anwender plausibles Verhalten zeigen. Ferner soll es (im Hinblick auf eine denkbare interaktive Version) nicht zu langsam arbeiten.

b) Das System soll portabel sein.

c) Es soll auch auf kleinen Rechnern installierbar sein. Als kleiner Rechner wird hier ein typischer Prozeßrechner mit 64 k Wörtern zu 16 Bits angenommen.

d) Es soll mit geringem Aufwand realisierbar sein.

e) Korrekturen, Erweiterungen und Änderungen sowohl von ESPRESO-S als auch von ESPRESO-W sollen möglichst geringen Aufwand erfordern.

Die Bedienungsfreundlichkeit (a) wird erzielt durch eine Reihe konzeptioneller Entscheidungen, die im Kapitel 3 diskutiert wurden. Als Beispiel sei hier nur die symmetrische Wirkung von KONV und DEKONV genannt.

Im Hinblick auf die übrigen Ziele wurden Realisierungskonzepte festgelegt:

- Die Implementierung verwendet als Basismaschine das PASCAL/360 System von der University at Stony Brook (Kieburtz et al., 1979). Dieses ist im Kernforschungszentrum Karlsruhe installiert (Hellmann, Ludewig, 1980). Wegen mehrerer Fehler in der derzeit vertriebenen Version war es allerdings nicht möglich, das System zu integrieren und zu testen.

  Die Verwendung von PASCAL ist günstig im Sinne von (b), (d) und (e), denn eine hohe Sprache wie PASCAL macht das Programm portabel und erleichtert die Programmierung und die Wartung. Da auch viele Kleinrechner über ein PASCAL-System verfügen, ist diese Entscheidung auch günstig im Sinne von (c).

  Die speziellen Erweiterungen des von diesem Compiler akzeptierten PASCAL-X wurden im Interesse der Portabilität nur soweit wie nötig verwendet. Schwierigkeiten machen die Modularisierung und die Ein-/Ausgabe über non-standard-files.

Standard-PASCAL (Jensen, Wirth, 1974) bietet keine Moduln als einzeln übersetzbare Einheiten und als Gültigkeitsbereiche solcher Variablen, deren Lebensdauer nicht an einen Block geknüpft sein soll. Da aber Moduln im Interesse einer klaren Struktur nicht entbehrlich sind (siehe 3.1.5), bietet PASCAL-X wie viele andere PASCAL-Systeme eine Spracherweiterung, die nicht standardisiert ist und daher die Portabilität beeinträchtigt.

Non-standard-files sind in PASCAL definiert, aber im PASCAL/360 noch nicht implementiert. Daher enthält ESPRESO-W zum Lesen aus und Schreiben in Dateien FORTRAN-Subroutines, die ebenfalls nachteilig für eine Übertragung auf andere Anlagen sind.

- Die ESPRESO-Datei soll während der Laufzeit des Programms vollständig im Hauptspeicher stehen. Dies erschwert zwar die Realisierung auf dem Kleinrechner (c), gestattet jedoch, die Daten vollständig in PASCAL zu definieren und wirkt damit positiv für (b), (d) und (e). Zu Beginn jedes Programmlaufs müssen die Daten allerdings von einem externen Speicher geholt und am Ende zurückgebracht werden.

- Die Möglichkeiten der dynamischen Variablen (Operationen new und dispose) wurden nicht verwendet. Dies erleichtert das Ein- und Ausspeichern der Listen (d), erlaubt die Behandlung des Listenüberlaufs durch das Programm (a) und trägt zur Speichereinsparung bei (b).

- Im Interesse einer leichten Realisierung (d) und Wartung (e) wurde ESPRESO-W datenorientiert modularisiert. Jede der großen Datenstrukturen, z.B. die Liste der Objekte, ist deklariert als privates Datum e i n e s Moduls und den anderen Moduln nur durch die speziellen Zugriffsoperationen zugänglich, die explizit vorgesehen sind (siehe 4.2.2).
  Globale Variablen werden nicht verwendet, Funktionen haben grundsätzlich keine Nebenwirkungen.

- Die Syntax einschließlich der Wortsymbole ist bis auf einige sehr fundamentale Schemata, die kaum jemals geändert werden dürften (z.B. der Aufbau der Sektionen aus Kopf, Rumpf und Schwanz), in Tabellenform gespeichert und daher leicht und ohne Eingriffe in die Programme änderbar.

- Ebenfalls zur Erleichterung der Realisierung ist in KONV nur eine primitive Fehlerbehandlung vorgesehen. Der Anwender erhält den Fehler präzise gemeldet, jedoch wird anschließend viel oder alles folgende überlesen, um einfach und sicher wieder aufsetzen zu können, so daß zur Erkennung aller Fehler u.U. viele Eingabeversuche notwendig sind.

- Mehrfachinterpretationen von Variablen oder Parametern werden grundsätzlich vermieden. Die einzige Ausnahme betrifft die Indizierung von Listen. Hier gilt die Konvention, daß die Numerierung mit "1" beginnt. "0" kennzeichnet den leeren Zeiger oder ein nicht vorhandenes Element in der Liste. Dadurch wird sowohl bei der Speicherung als auch beim Prozeduraufruf an vielen Stellen eine Variable eingespart.

4.2.2 Die Modulstruktur

Die Modularisierung von ESPRESO-W ist, wie oben gesagt, orientiert an den Datenstrukturen des Programms. Dies sind:

- die Kommandos zur Bedienung - Protokolle und Reports	Standard-Ein-/Ausgabe
- die ESPRESO-S-Eingabe für die Konvertierung - die ESPRESO-S-Ausgabe der Dekonvertierung	Dateien

- die Objekte, von denen noch abgespalten sind
  - die Namen
  - die Texte und Text-Selektoren
- die Verknüpfungen
	ESPRESO-Datei

- die Syntax, bestehend aus (siehe 4.2.4.2)
  - der Relationentabelle
  - der Angaben-Syntax
  - einer Liste spezieller Wortsymbole
	Syntaxtabellen

Ein gewisser Teil der Syntax ist in die Programme KONV und DEKONV eingearbeitet und daher nicht explizit gespeichert.

4.2.3 Die Schichtenstruktur

Neben PASCAL-X wurden zur Implementierung von ESPRESO-W drei weitere Sprachen verwendet:

Die IBM-Kommandosprache zur interaktiven Bedienung des Systems, die IBM-Job-Control-Language (JCL) zur Ausführung von Datei-Operationen (Einrichten, Umbenennen, Löschen) und zum Start der PASCAL-Programme, FORTRAN für die Kommunikation mit non-standard-files (siehe 4.2.1).

Betriebs- und Laufzeitsysteme lassen nur bestimmte Beziehungen zwischen den in diesen Sprachen implementierten Moduln zu: Die Aufrufe können nur in der Richtung

Kommandosprache -> JCL -> PASCAL -> FORTRAN

erfolgen. Daher ist für ESPRESO-W eine Schichtenstruktur vorgegeben:

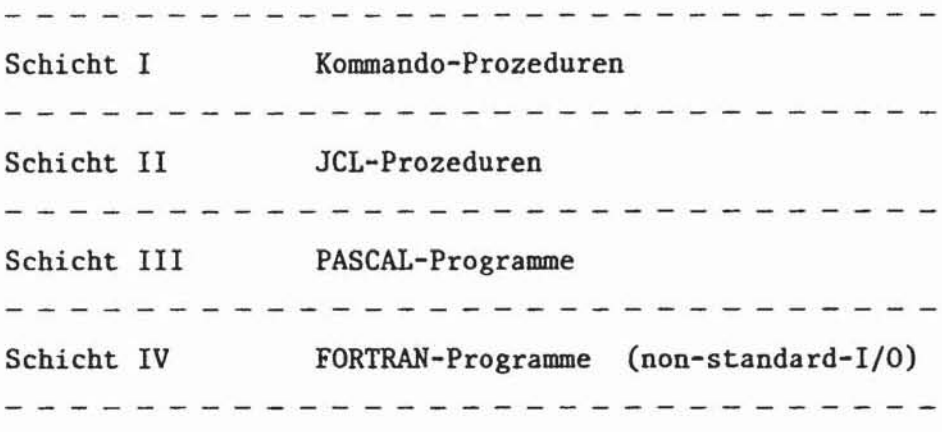


Bild 4.1 Schichtenstruktur von ESPRESO-W

Eine Schicht 0, die den Dialog mit dem Bediener führt und die Prozeduren der Schicht I aufruft, ist virtuell, denn aufgrund der Möglichkeiten, die die Kommando-Prozeduren bieten, ist es vorteilhaft, jedes Kommando als eigene Prozedur zu realisieren.

Natürlich wurde versucht, soviel wie möglich von ESPRESO-W in die Schicht III, die PASCAL-Schicht, zu bringen. Diese ist daher wie folgt weiter gegliedert:

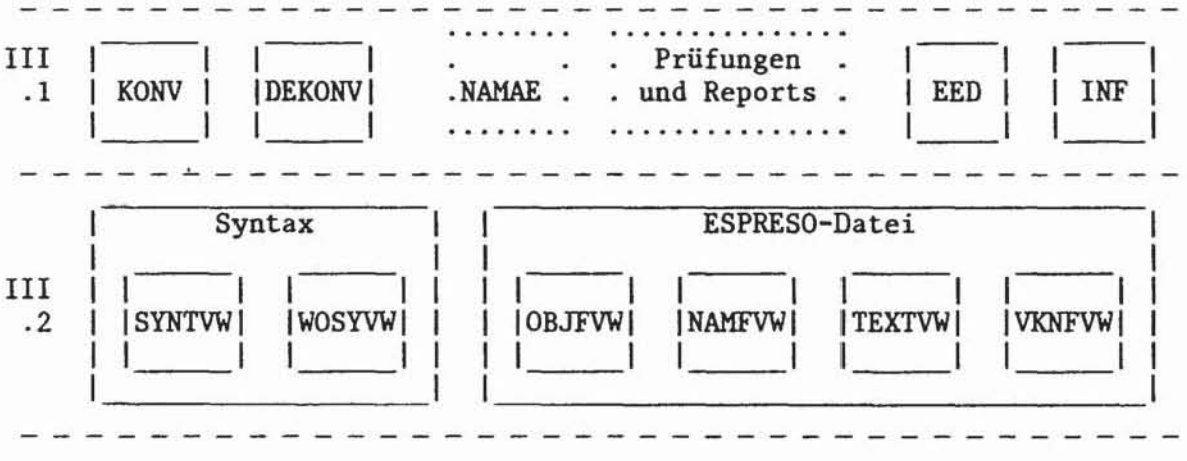


Bild 4.2 Struktur der Schicht III

Die punktierten Teile sind nicht implementiert. Einige Funktionen, z.B. LED zum Löschen einer ESPRESO-Datei, fehlen in diesem Bild, da sie völlig in den Ebenen I und II realisiert sind.

4.2.4 Kurzbeschreibung der ESPRESO-W-Programme

Dieser Abschnitt soll einen Einblick in Architektur und Organisation des Programmsystems ESPRESO-W geben. Die Moduln werden nicht streng top-down beschrieben, sondern so, daß möglichst wenige Vorwärtsbezüge notwendig sind.

4.2.4.1 Die Schichten I, II und IV

Die nicht in PASCAL-X geschriebenen Programmteile, d.h. die Schichten I, II und IV, sind auf die speziellen Erfordernisse der für die Implementierung benutzten Rechnerkonfiguration aus IBM 370-168 und IBM 3033 zugeschnitten. Bei der Installation auf einem anderen Rechner müssen sie erheblich verändert werden, soweit sie nicht - wie z.B. die Schicht IV - ganz entfallen. Sie werden daher in diesem Abschnitt nur kurz gestreift.

Prozeduren in JCL (Job-Control-Language) verwalten die Dateien und starten die aus den PASCAL-Programmen enstandenen Lademoduln. Um den Anwender vom Umgang mit dieser wenig benutzerfreundlichen Sprache zu entlasten, sind die JCL-Prozeduren völlig durch interaktiv ablaufende Kommando-Prozeduren verdeckt.

Die Kommando-Prozeduren werden vom Anwender unter TSO mit dem Namen der Funktionen (EED, KONV, ...) aufgerufen; sie prüfen zunächst die Parameter und geben sie einschließlich der Standardwerte, die der Anwender nicht verändert hat, aus. Anschließend wird die erforderliche JCL-Prozedur für die Stapelverarbeitung generiert und ihre Ausführung ausgelöst. Der Katalog der ESPRESO-Dateien wird z.T. direkt von den Kommando-Prozeduren verwaltet.

Alle Kommando-Prozeduren können mit einem Parameter HELP gestartet werden; sie geben dann über Funktion und Parameter Auskunft.

Die Schicht IV enthält FORTRAN-Subroutines für das formalisierte Lesen und Schreiben eines Standard-Records (80 Zeichen) und für das Rücksetzen der Lese- und Schreib-Zeiger (Rewind), außerdem Ein- und Ausgabeprogramme für das Retten und Rekonstruieren der ESPRESO-Datei (siehe 4.2.4.3). Sie schreiben und lesen die permanente Datei unformatiert.
Diese Subroutines sind im PASCAL-Programm gekapselt durch spezielle Prozeduren; alle übrigen Programme der Schicht III wickeln ihre Datei-Ein- und Ausgabe nur über diese Prozeduren ab.

4.2.4.2 Die Syntax-Verwaltung

Formal wäre es ausreichend, die attributierte Grammatik (7.) unverändert abzuspeichern und die ESPRESO-Datei als unstrukturierte Menge der in der Grammatik auftretenden Paare und freien Tupel zu realisieren. Abgesehen von dem sehr hohen Speicheraufwand würde dieses Vorgehen an der Ineffizienz scheitern, mit der die Operationen auf die Datei arbeiten würden. Daher werden sowohl die Grammatik als auch die ESPRESO-Datei in einer zweckmäßigeren Form angelegt.

Die Regeln der Grammatik lassen sich wie folgt einteilen:

a) Syntax der Abschnitte und Sektionen (globale Syntax-Schemata),

b) Kontextfreie Syntax der Angaben,

c) Abstrakte Syntax der Angaben, d.h. Festlegungen darüber, welche Komponenten signifikant, notwendig oder möglich sind und welcher Art jede Komponente sein darf,

d) spezielle Regeln für Sonderfälle, insbesondere für Texte,

e) Syntax der Attribute.

Die Informationen nach (a) und (d) sind bis auf die darin enthaltenen Wortsymbole ausprogrammiert, d.h. in die Programme KONV und DEKONV eingearbeitet (siehe 4.2.4.4). Alle anderen Informationen zur Grammatik sind in einer Datei in übersichtlicher, leicht änderbarer Form zusammengestellt. Der Inhalt dieser Datei ist wie folgt gegliedert:

- Ein Kopf enthält Kommentare und eine Identifikation.

- Ihm folgt eine Tabelle, die nur geringfügig verändert ist gegenüber Tabelle 4.3. Die Attribute a3 bis a6 aus Tabelle 4.2 sind ebenfalls als einstellige Relationen darin enthalten.

- Der anschließende Abschnitt gibt an, welche Relationen bestimmte Attribute implizieren und sich dadurch gegenseitig ausschließen.

- Darauf folgt eine Liste der Wortsymbole nach (a) und (d).

- Am Schluß steht die Syntax der Angaben. Diese hat z.B. für die einfache Relation r41 folgende Form:

```
r41                          Relationsnummer
W    capacity                Angabe beginnt mit diesem Wortsymbol
N    2                       Nennung der Komponente 2
F    0                       Fortsetzung bei 0 (= log. Ende)
;                            Ende der Produktion für r41
```

Ähnlich sind auch Verzweigungen, Wiederholungen usw. dargestellt.

Der Inhalt der Syntax-Datei muß geprüft, gespeichert und selektiv zugänglich gemacht werden. Dafür dient der Modul SYNTVW (Syntax-Verwaltung). Die Wortsymbole (einschließlich derer in (a) und (d)) werden vom Modul WOSYVW (Wortsymbol-Verwaltung) gesondert verwaltet.

Da die Prüfung der Syntax-Datei nur nach Änderungen erforderlich wird, ist der betreffende Teil aus SYNTVW und WOSYVW herausgelöst und in einem unabhängigen Programm SYNTCH (Syntax-Check) untergebracht. SYNTCH erzeugt ein Protokoll der Syntax-Vorverarbeitung und eine stark komprimierte Darstellung der Syntax, die SYNTVW und WOSYVW als Eingabe dient. In dieser komprimierten Form ist die Relationentabelle auf eine Menge verschiedener Vektoren aus Wahrheitswerten und eine Zuordnungstabelle reduziert.

Die Angaben-Syntax, in der die Wortsymbole durch Codes ersetzt sind, ist als Feld aus PASCAL-Records gespeichert; eine Liste ordnet jeder Relation einen (Anfangs-) Index zu.

Die Wortsymbole sind mit ihrer Länge (zur schnelleren Identifizierung) und Hinweisen auf ihre Verwendung (z.B. als Artkennzeichen) gespeichert.

4.2.4.3 Die Verwaltung der ESPRESO-Datei

Die ESPRESO-Datei enthält die Spezifikation, dargestellt durch Objekte und Verknüpfungen. Alle Informationen sind in Listen enthalten, die hier zur Unterscheidung von verketteten Listen als Files bezeichnet werden. Die Files sind als Felder von PASCAL-Records realisiert, so daß natürliche Zahlen (für die Indices) als Zeiger dienen. Freie Plätze im File sind jeweils in einer Leerliste verkettet. Die besetzten enthalten eine Statusinformation, die Markierung, die darüber Auskunft gibt, ob das Listenelement vordefiniert (7.9), schon fest oder erst vorläufig eingetragen ist, ob es als Sicherheitskopie angelegt wurde oder ob es am Ende des Programmlaufs gelöscht werden soll.

Die wesentliche Aufgabe der ESPRESO-Datei-Verwaltung ist die Bereitstellung effizienter Operationen auf die Verknüpfungen der Objekte (feststellen, ob vorhanden, eintragen, löschen, und einige weitere Operationen). Daher steht dieser Aspekt im Mittelpunkt. Die Namen und Texte sind nur im Zusammenhang mit der Ein- und Ausgabe von Bedeutung, intern können sie durch Codes ersetzt werden. Daher werden Namen (und mit ihnen auch die Schlüssel) und Texte jeweils in speziellen Moduln verwaltet, NAMFVW und TEXTVW. Die Namen stehen in einer Hash-Tabelle, die Texte werden in eine sequentielle Datei geschrieben. Wegen des ineffizienten Zugriffs bei der Ausgabe eines Texts kann dies nur eine provisorische Lösung darstellen. Die Elemente des eigentlichen Objektfiles im Modul OBJFVW sind mit den Namen und Texten verzeigert.

Um trotz der unbegrenzten Zahl von Verknüpfungen, als deren Komponente ein einziges Objekt verwendet werden kann, eine einheitliche Darstellung aller Objekte zu ermöglichen, sind die Verknüpfungen eines Objekts jeweils in (bis zu) drei verketteten Listen gespeichert. Liste 1 enthält diejenigen Verknüpfungen, in denen das Objekt Komponente 1 ist, die Listen 2 und 3 entsprechend die andern Verknüpfungen des Objekts. Eine dreistellige Verknüpfung ist also ein Knoten dreier Listen. Durch doppelte Verkettung sind Änderungen im Verknüpfungsfile sehr einfach.

Jeder Eintrag im Objekt-File und im Verknüpfungsfile enthält außer der Markierung und den verschiedenen Zeigern die Art des Objekts bzw. die Relation der Verknüpfung.

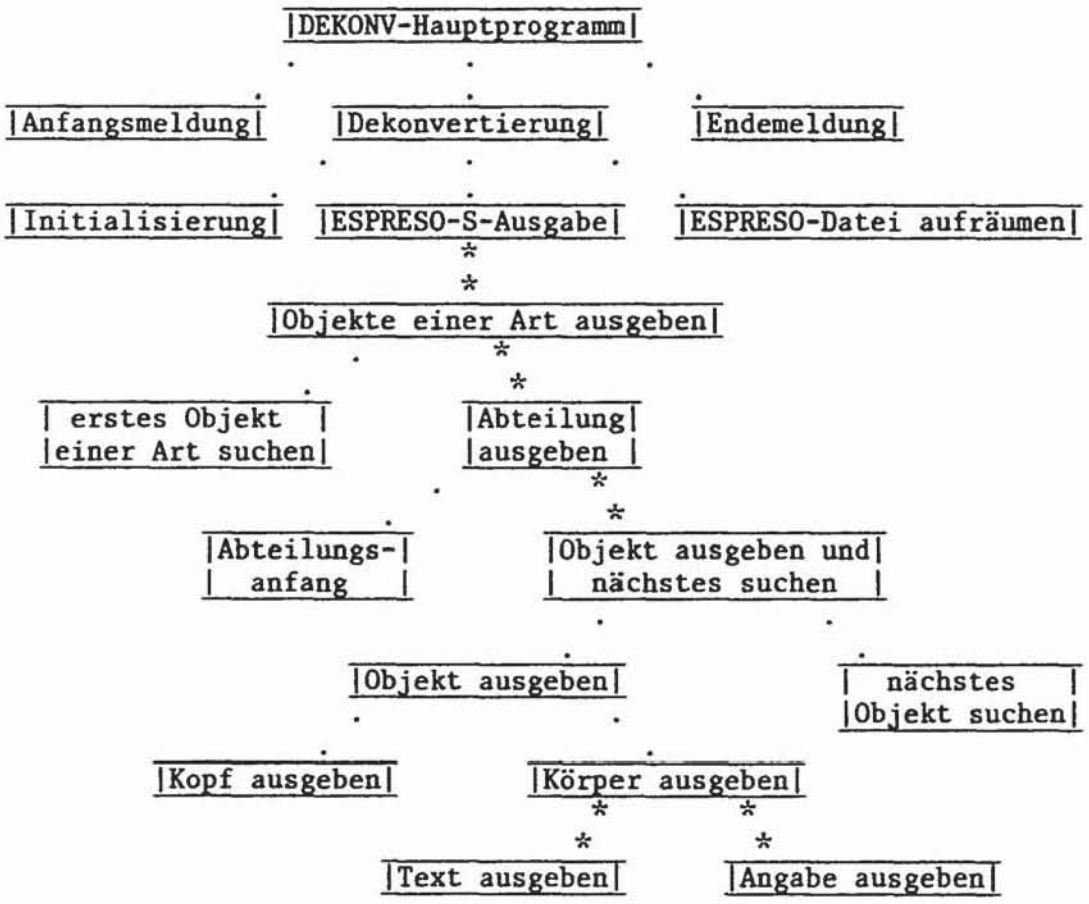
Da die ESPRESO-Datei dem Benutzer über einen Programmlauf hinaus erhalten bleiben muß, wird sie jeweils zu Beginn von einem Sekundärspeicher geholt (rekonstruiert) und zum Schluß zurückgebracht (gerettet). Diese beiden Funktionen sind wie einige weitere, die mit mehreren Files der ESPRESO-Datei arbeiten, in einem übergeordneten Modul KOMPLEX untergebracht, der selbst keine Variablen enthält, sondern nur die Operationen der Moduln OBJFVW, NAMFVW, TEXTVW und VKNFVW verwendet.

4.2.4.4 KONV und DEKONV

Die Programme KONV und DEKONV zur Konvertierung und Dekonvertierung haben ähnliche, an der Syntax von ESPRESO-S orientierte Struktur (siehe Bild 4.3). KONV ist umfangreicher und komplizierter, da Fehler in der Spezifikation erkannt und behandelt werden müssen.

Wie die Sprache ESPRESO-S sind auch die Programme zur ihrer Verarbeitung rekursiv: Innerhalb einer Angabe kann ein Objekt verarbeitet bzw. ausgegeben werden (im Bild 4.3 nicht dargestellt). Aus syntaktischen Gründen und zur Fehlerbehandlung ist es allerdings notwendig, die Abschnitte von den eingeschachtelten Sektionen zu unterscheiden.

Bild 4.3 DEKONV



Die Darstellung im Bild 4.3 folgt der Jackson-Design-Methododology. Kanten, die durch Punkte markiert sind, haben die Bedeutung "wird einmal ausgeführt". Sterne kennzeichnen, daß die Ausführung nicht, einmal oder mehrmals erfolgt.

KONV enthält als Hauptbestandteile einen Scanner (Symbolentschlüßler) und einen Parser (Zerteiler), die jeweils einen Modul bilden. Der Scanner liefert bei jedem Aufruf das nächste Grundsymbol (die Werte der in 7.6 bis 7.8 definierten, in 7.2 bis 7.5 verwendeten syntaktischen Variablen) oder einen Fehlercode und erzeugt das Protokoll. Der Parser macht mit Hilfe der Syntax-Verwaltung (siehe 4.2.4.2) die syntaktische Analyse entsprechend 7.2 bis 7.5 und modifiziert den Inhalt der ESPRESO-Datei.

Spezielle Regeln der Syntax, die zu einer schematischen Behandlung ungeeignet sind, z.B. die Prüfung hierarchischer Relationen auf Zyklenfreiheit, werden durch einen gesonderten Modul AUSNAHME bearbeitet. Dadurch sind die Sonderfälle isoliert, und der Parser bleibt übersichtlich.

KONV verarbeitet die Eingabe in einem einzigen Durchlauf. Dadurch ist es möglich, das Programm interaktiv zu betreiben, wenn der Scanner direkt von einem Eingabegerät liest. Die Beschränkung auf einen Lauf erschwert allerdings die Fehlerbehandlung. Syntaktische Fehler werden überwiegend auf der Abschnittsebene behandelt: Alle Änderungen in der ESPRESO-Datei werden innerhalb eines Abschnitts durch Kopieren des früheren Zustands so vorgenommen, daß am Ende entweder der neue Zustand fixiert oder der alte wiederhergestellt werden kann.

4.2.4.5 EED und INF

Das Programm zum Einrichten der ESPRESO-Datei arbeitet ganz ähnlich wie KONV, nur wird zu Beginn nicht eine Datei rekonstruiert, sondern leer initialisiert. Dann wird der Standard-Kontext (7.9) unter Verwendung von KONV konvertiert. Vor dem Retten erhalten alle Objekte die nur von EED verwandte Markierung "vordefiniert".

INF rekonstruiert nicht die gesamte ESPRESO-Datei, sondern nur die Verwaltungsinformation (wann von wem wie angelegt und geändert) und gibt sie im Protokoll aus.

5. Zur Abbildung von ESPRESO-S in eine Programmiersprache

5.1 Zweck der Abbildung

Soweit ESPRESO in den Kapiteln 3 und 4 entwickelt wurde, bietet es Führung und Unterstützung bei der Anfertigung einer Spezifikation. Der nun notwendige Übergang auf eine Programmiersprache muß jedoch ganz und gar von Hand erfolgen, da das System weder Hilfe dabei leistet noch die Konsistenz von Beschreibung und Programm sicherstellt. Damit besteht die Gefahr, daß die Wirkung des Spezifikationssystems verpufft, weil der kontrollierte Anschluß an eine Programmiersprache fehlt. In diesem Kapitel wird der notwendige Brückenschlag diskutiert; die Ansätze sind jedoch nicht wie die in Kapitel 3 bis zur Implementierung geführt.

Es wird eine Abbildung angegeben, die es gestattet, eine ESPRESO-Formulierung in ein Programmskelett zu transformieren. Das vollständige Programm kann nicht erzeugt werden, da die Spezifikation auf der ESPRESO-Ebene meist nicht vollständig, jedenfalls nicht vollständig formal ist; insbesondere fehlt die Möglichkeit, die Algorithmen (im engeren Sinne) zu formulieren, denn ESPRESO-S enthält keine Darstellungen arithmetischer und logischer Operationen und Wertzuweisungen.

Die Angabe dieser Abbildung dient folgenden Zielen:

- Die in ESPRESO-S bereits ausgedrückte Information soll bei der Codierung automatisch erhalten bleiben und damit vor einer unabsichtlichen Verfälschung durch Fehlinterpretation oder Codierfehler geschützt werden.

- Durch die Abbildung auf eine wohldefinierte Programmiersprache wird die Semantik von ESPRESO-S implizit definiert.

- Durch Einsatz der Transformation wird der Codieraufwand vermindert.

Die Abbildung zielt nicht auf eine besonders effiziente Lösung; eine solche kann aus der korrekten, aber u.U. ineffizienten erzeugt werden durch Einsatz die Korrektheit erhaltender Transformationen (siehe 5.10).

5.2 Die Zielsprache

5.2.1 Anforderungen an eine Zielsprache

ESPRESO gestattet und fördert die informale Beschreibung der Inhalte von Prozeduren, Variablen, Puffern usw. und die formale Beschreibung ihrer Strukturen und Schnittstellen. Soll der Übergang von ESPRESO-S auf die Zielsprache keinen Informationsverlust bringen, so müssen für Schnittstellen und Strukturen entsprechende oder feinere Ausdrucksmittel vorhanden sein.

Vor allem muß eine Möglichkeit bestehen, die Gültigkeitsbereiche der Daten und Prozeduren einzuschränken, bei Daten nach Zugriffsarten differenziert. Dies macht eine Abbildung der Moduln auf die Codeebene überflüssig, denn sie markieren im Entwurf nur Gültigkeitsbereiche. Allerdings können Moduln auch in der Zielsprache sinnvoll sein, z.B. um das Testen und die Programm-Verwaltung zu vereinfachen.

Für die Medien sollte die Zielsprache Elemente enthalten, die den Konzepten in ESPRESO entsprechen, also Variablen, Puffer, Trigger und Betriebsmittel mit den entsprechenden impliziten Koordinierungsfunktionen. Ebenso sollte für die Parallelität ein Konzept wie in ALGOL 68 bereitstehen. Auf diese Weise sind die Betriebssystem-Aufrufe, die bei der Ausführung eines aus der ESPRESO-Spezifikation erzeugten Programms nötig sind, vollständig im Laufzeitsystem der Zielsprache verborgen.

Nachfolgend wird ESPRESO-S zunächst abgebildet auf eine hypothetische Programmiersprache, die den genannten Anforderungen entspricht. Diese Sprache ist an PASCAL orientiert und wird daher als E-PASCAL bezeichnet. In 5.6 wird die Verwendung einer verfügbaren Programmiersprache anstelle von E-PASCAL diskutiert. 5.8 gibt an, wie das Laufzeitsystem realisiert werden kann.

5.2.2 Skizze der hypothetischen Programmiersprache E-PASCAL

E-PASCAL unterscheidet sich von Standard-PASCAL (Jensen, Wirth, 1974) durch folgende Merkmale:

a) Jede Prozedur und jeder Block kann eine Schnittstellen-Deklaration enthalten, die die Außenbeziehungen explizit angibt.

   Beispiel für eine solche Deklaration:

```
interface reads V1, V2;
          writes V1, V3;
          consumes P1;
          transpar P1;
          outpar P2
interfend
```

b) In PASCAL stehen zwei verschiedene Parameter-Mechanismen zur Verfügung, by-value und by-reference. Für ein System mit Parallelität gibt es widersprüchliche Anforderungen an den Parameter-Mechanismus: Er soll sicher wie die Übergabe by-value, schnell wie der Zugriff by-reference sein und die Verwendung des aktuellen Wertes garantieren wie der call-by-name.

   Für E-PASCAL wird grundsätzlich der call-by-name verwendet, wobei die Nachteile von ALGOL 60 vermieden werden durch den Ausschluß von Prozeduren als Parameter und durch erheblich stärkere Möglichkeiten zur statischen Prüfung, die vor allem durch die Markierung der Parameter als inpar, outpar oder transpar wie in ESPRESO-S entstehen. Ausdrücke als aktuelle Parameter sind nur bei Eingabe-Parametern zugelassen. Eingabe-Parameter dürfen nur rechts, Ausgabeparameter nur links in Wertzuweisungen stehen; werden sie wieder als aktuelle Parameter verwendet, dann nur von derselben Art.

c) Für alle Variablen, die von mehreren, möglicherweise gleichzeitig ablaufenden Rechenvorschriften verwendet werden, sieht der E-PASCAL-Compiler automatisch Monitore* vor, die die Zugriffe koordinieren.

   Puffer und Trigger können in E-PASCAL als spezielle Datenstrukturen ähnlich wie Felder oder Records vereinbart werden. Sie werden ebenfalls von Monitoren verwaltet. Die Vereinbarung ist an PASCAL-Records orientiert:

```
var B1: buffer (10) of Typ1;
          fifo;
          . . .
        end;
```

   Ähnlich werden auch Betriebsmittel vereinbart.

d) In E-PASCAL kann ein Programm aus mehreren Programmen im Sinne von Standard-PASCAL bestehen; eines davon ist als Hauptprogramm gekennzeichnet. Alle andern können mit einem create-Kommando als asynchrone Prozesse gestartet werden. Sie haben Zugriff auf die im Hauptprogramm deklarierten Variablen und Puffer.

---

* "Monitor" ist hier gebraucht für irgendeine Realisierung des Sekretärsprinzips nach Dijkstra (1971).

5.3 Vorgehen zur Realisierung eines in ESPRESO-S beschriebenen Systems

Durch die Abbildung wird aus der Spezifikation eine Menge von E-PASCAL-Prozeduren erzeugt, deren Rümpfe zum Teil noch leer sind (siehe 5.5.4); die Modulstruktur geht verloren.

Den leeren Prozeduren sind Schnittstellen-Deklarationen zugeordnet, die die Außenbeziehungen der Prozeduren auf die in ESPRESO-S angegebenen Verknüpfungen beschränken. Der Ansatz sichert also die Erhaltung der Korrektheit dadurch, daß der Programmierer nur noch Programmblöcke mit definierter Umgebung bekommt. Das Innere der Blöcke kann er beliebig ausgestalten, auch durch lokale Variablen und Prozeduren, die Schnittstelle zur Umgebung ist fest vorgegeben. Medien kann der Programmierer nur durch deren Monitore ansprechen.

Der Übergang von der abgeschlossenen Spezifikation (siehe 3.4.3) zum Code wird in folgenden Schritten vollzogen:

a) Übertragung von ESPRESO-S (d.h. aus der ESPRESO-Datei) in die Programmiersprache. Das Ergebnis besteht aus einem fertigen Teil und solchen Komponenten, in denen nur die Deklarationen und Kommentare vorhanden sind.

b) Programmierung der fehlenden Teile nach Maßgabe der informalen Beschreibungen, die bei der Transformation übertragen wurden, und mit strikter Beschränkung auf die durch die vorgegebenen Deklarationen definierte Schnittstelle nach außen.

c) Plausibilitätsprüfung durch Vergleich der bereitgestellten und der tatsächlich verwendeten Zugriffsrechte. Da nur solche Zugriffe erlaubt werden, die der ESPRESO-Formulierung zufolge nötig sind, deutet sowohl die Verwendung eines nicht vorgesehenen Zugriffsrechtes, die vom Compiler festgestellt werden kann, als auch der umgekehrte Fall auf einen Fehler hin. Der Compiler sollte in diesen Fällen Warnungen erzeugen.

d) Binden aller Moduln mit dem Laufzeit- und Betriebssystem.

Weitere, hier nicht genannte Prüfungen und Erprobungen können praktisch in jedem Stadium durchgeführt werden. Wichtig ist dabei, daß Korrekturen und Änderungen stets von ganz oben durchgezogen werden, also beginnend in der ESPRESO-Spezifikation.

5.4 Konsistenz- und Vollständigkeitsanforderungen an die ESPRESO-Spezifikation

Um eine ESPRESO-Spezifikation nach E-PASCAL transformieren zu können, muß diese über die syntaktische Korrektheit hinaus die folgenden Voraussetzungen erfüllen:
(Die Nummern verweisen auf Tabelle 4.3 und den Anhang.)

- Die Gültigkeitsbereiche, die durch die Modulstruktur (r1) und die Restriktionen (r23, r28/29, r36 bis r40) definiert sind, werden entsprechend 3.1.5 eingehalten. Es wird also kein Name außerhalb seines Gültigkeitsbereichs verwendet, es sei denn in Querverweisen. Dies betrifft vor allem die Aktionen (r11 bis r22). Die Restriktionen (r28/29, r36 bis r40) können den Gültigkeitsbereich nur einschränken, nicht erweitern!

- Jeder Block ist Teil einer Prozedur. Die Prozeduren bilden unter der Relation "ruft auf" (r6) einen zusammenhängenden Graphen; genau eine Prozedur wird an keiner Stelle aufgerufen.

- Eine Prozedur oder ein Block ist nichtterminal (r2 bis r6) und/oder führt Aktionen aus (r12 bis r22).

- Alle aufgerufenen Prozeduren sind definiert, die Parameterlisten in Aufruf und Definition stimmen nach Zahl und Namen der Parameter überein. (Die Konsistenz der Übertragungsrichtung ist durch die Syntax garantiert, Typkonsistenz kann auf Codeebene geprüft werden.)

- Die Typen der Variablen und Puffer sind definiert; alle komplexen Typen sind auf elementare, d.h. vordefinierte Typen zurückgeführt.

- Variablen, die als Bedingung verwendet werden (r11), haben den Typ boolean. In einer alternativen Zerlegung (r4, r5) haben Variable und Konstanten denselben Typ.

- Zu jedem Puffer ist die Kapazität definiert. (Andernfalls ist eine Realisierung mit dynamischen Variablen nötig.)

- Allen Konstanten ist ein Wert zugeordnet. Dies geschieht durch einen Text mit einem bestimmten Selektor, z.B. "value":

```
constant C1:
         text value ¢ 17.04 ¢
end C1.
```

Typen und Werte der Konstanten müssen abhängig von ihrer Verwendung bestimmte Bedingungen erfüllen (z.B. natürliche Zahlen sein in vielen Relationen).

5.5 Übertragung der einzelnen Komponenten einer Spezifikation

Nachfolgend wird die Abbildung der Objekte und ihrer Verknüpfungen kurz beschrieben, geordnet nach den Arten der Objekte. In 5.5.5 sind die Aussagen zusammengefaßt.

Allgemein gilt, daß alle Vereinbarungen, die bei der automatischen Umsetzung erzeugt werden, global gemacht werden können, da die Einhaltung der Gültigkeitsbereiche bereits sichergestellt ist.

Objekte mit dem Attribut "außen" werden nicht abgebildet, bei denen mit Attribut "grenze" werden Transfers über die Systemgrenzen hinweg zugelassen.

5.5.1 Objekte und Verknüpfungen, die nicht abgebildet werden

Findet die Übertragung in die Programmiersprache von Hand statt, so muß durch Erhaltung und Weitergabe aller, auch der redundanten Information dafür gesorgt werden, daß der Codierer die in ESPRESO-S vorgesehenen Restriktionen einhält. Wird die Übertragung dagegen mechanisch vorgenommen, so garantiert die Transformation die Erhaltung der Korrektheit. Restriktionen, deren Einhaltung so gewährleistet ist, brauchen nicht mitgeführt zu werden.

Ein Text-Objekt (3.2.3) hat keine Entsprechung auf der Codeebene und wird daher nur zu einem Kommentar. Ist es auch als Stichwort verwendet, so kommt eine Liste der markierten Objekte hinzu. Ein Modul wird ähnlich behandelt; die ihm zugeordneten Objekte werden in einem Kommentar aufgelistet.

Alle so entstandenen Kommentare sind im Programm zusammengefaßt und durch eine Strukturierung, z.B. mit Nummern, überschaubar gehalten.

Beispiel:

```
module Personendaten:
¢ Verwaltung eines Files mit Personendaten ¢;
comprises procedure Daten-ein
      and procedure Daten-aus
      and variable  Datenbestand
end Personendaten;
```

Daraus wird (mit fiktiven Nummern):

```
(* 1.3  module Personendaten:
       Verwaltung eines Files mit Personendaten
       comprises Daten-ein          (7.3)
             and Daten-aus          (7.4)
             and Datenbestand       (4.1)          *)
```

5.5.2 Konstanten, Typen, Parameter

Konstanten-, Typ- und Parameter-Definitionen in ESPRESO-S werden umgesetzt in die entsprechenden Definitionen und Deklarationen in E-PASCAL. Das Attribut für die Struktur von Typen und Variablen (vgl. Zeile 3 in Tabelle 4.2) ist durch die Abbildung auf Typ-Deklarationen implizit abgebildet.

5.5.3 Medien

Die Medien werden abgebildet auf abstrakte Datenstrukturen, also auf Datenvereinbarungen entsprechend den gewählten Typen und auf Monitore, die darauf exklusiven Zugriff haben. Betriebsmittel und Trigger, mit deren Inhalt keine Information verbunden ist, können jeweils auf eine vom Monitor verwaltete Zahl abgebildet werden.

Eingangsverhalten und Organisation von Puffern und Triggern gehen in die Vereinbarung ihres Monitors ein (siehe 5.8.5).

Reale Betriebsmittel entstehen nicht durch ihre Deklaration, sondern sind a priori vorhanden. Im Programm können sie also nicht geschaffen, sondern nur verwaltet werden. Da aber zur Belegung und Freigabe die Betriebsmittel selbst nicht angesprochen werden müssen, kann das E-PASCAL-Programm sie wie die virtuellen verwalten und den belegenden Prozessen zuordnen.

5.5.4 Prozeduren und Blöcke

Prozeduren, die nicht wiedereintrittsfest (reentrant) sind, erhalten jeweils ein virtuelles Betriebsmittel zugeordnet; ihr Aufruf schließt seine Belegung ein (siehe 5.5.3, auch 5.9). Im übrigen spielt die Unterscheidung zwischen Prozedur und Block hier keine Rolle; nachfolgend wird nur von Rechenvorschriften gesprochen.

Zunächst wird dafür gesorgt, daß keine Rechenvorschrift sowohl Transfers als auch eine Zerlegung enthält (d.h. nichtterminal ist). Enthalte die Rechenvorschrift R die Menge der Steueraktionen und paarigen Aktionen SPA, die Menge der Transfers T und eine Zerlegung Z. R wird ersetzt durch die sequentielle Familie R1, R2, R3. R1, der Vater, enthält SPA, R2, der erste Sohn, enthält T, R3 enthält Z. Auf diese Weise können solche Rechenvorschriften beseitigt werden, so daß es nur noch nichtterminale ohne Transfers und terminale gibt.

5.5.4.1 Nichtterminale Rechenvorschriften, Steueraktionen und paarige Aktionen

Damit enthalten die nichtterminalen Rechenvorschriften nur noch Steueraktionen und paarige Aktionen. Diese sind durch die Spezifikation vollständig beschrieben und können automatisch in die Zielsprache übertragen werden. Dasselbe gilt für die Zerlegung. Folglich können die von den Transfers befreiten nichtterminalen Rechenvorschriften völlig automatisch nach E-PASCAL übertragen werden.
Das Attribut für den Ablauf (Zeile 2 in Tabelle 4.2) ist durch die Abbildung der Zerlegung implizit abgebildet.

Die Aktionen werden ausgeführt wie in 3.1.3 angegeben. Ist die Rechenvorschrift zyklisch, was durch eine Wiederholbedingung (r11) oder ein Abbruchkriterium ausgedrückt ist, so enthält der E-PASCAL-Block eine while-Schleife mit der (zuvor ausgewerteten) Bedingung als Kriterium. Diese Schleife enthält einen Block für alles weitere, was erzeugt wird. Falls eine Verknüpfung "gestartet von" besteht, beginnt der Block mit einer consume-Operation auf den betreffenden Trigger.

Bei der Maskierung von Puffern und Triggern muß zu Beginn der alte Zustand gespeichert und zum Schluß wiederhergestellt werden.

Der weitere Inhalt der Prozedur hängt von der Zerlegung der Rechenvorschrift ab: Ist diese sequentiell zerlegt (r2), so enthält der Block eine Sequenz von Prozedur-Aufrufen. Bei paralleler Zerlegung (r3) stehen alle n Aufrufe in einer "parallel-clause" (siehe 5.8.3). Die alternative Zerlegung (r4, r5) wird umgesetzt in eine Fallunterscheidung (case-Anweisung), die Aufrufe liegen in den Alternativen. Der Prozedur-Aufruf (r6) ergibt auch im Programm einen Prozedur-Aufruf (vgl. 5.9).

5.5.4.2 Terminale Rechenvorschriften, Transfers

Ein terminaler Block enthält nur Aktionen. Bis auf die Transfers ist die Abbildung der Aktionen in 5.5.4.1 beschrieben; die Transfers aber können nicht automatisch transformiert werden, da sie meist mit Manipulationen der Daten durch Rechnungen oder Vergleiche verbunden sind, die in ESPRESO-S nicht dargestellt werden können. Daher wird für die Transfers ein leerer E-PASCAL-Block erzeugt, den der Programmierer ausgestalten muß.

Um die Lokalität des Blocks so eng wie möglich zu halten und den Codierer an die in ESPRESO-S beschriebenen Außenschnittstellen zu binden, erhält der Block eine Schnittstellen-Deklaration, die der Codierer nicht erweitern darf (siehe 5.2.2 a).

5.5.5 Zusammenfassung

Es wurde gezeigt, daß die Text-Objekte nur auf Kommentare im Programm abgebildet werden. Das gilt auch für Moduln, deren Bedeutung bereits voll auf der Spezifikationsebene ausgewertet werden kann. Rechenvorschriften werden, wenn sie keine Transfers enthalten, automatisch in E-PASCAL-Prozeduren umgesetzt; andernfalls werden leere Prozedur-Rümpfe für sie erzeugt, deren Außenbeziehungen durch eine Schnittstellen-Deklaration restringiert sind.
Medien werden automatisch in die Deklarationen von Datenstrukturen umgesetzt, auf die nur durch Monitore zugegriffen werden kann. Typen und Konstanten bleiben bei der Umsetzung praktisch unverändert. Fristen gehen ein in die Initialisierungsoperationen von Triggern.

Tabelle 5.1 zeigt, wie die Relationen abgebildet werden. Sie entspricht im Aufbau der Tabelle 4.3.

Tabelle 5.1
Abbildung der Relationen

Nr.	Relation	Kommentar	Restriktion	Typ- und Datendefinition	Prozedur-Definition	Ablaufstruktur	Parameter-Besetzung	Transfer	Automatischer Zugriff
01	Modul enthält		x						
02	sequent. zerlegt					x			
03	parallel zerlegt					x			
04	alternativ zerlegt					x			
05	hat Schlüssel								x
06	ruft auf					x			
07	deklariert Parameter				x				
08/09	Ein-/Ausgabepar. ist						x		
10	trans. Parameter ist						x		
11	hat Bedingung								x
12/13	liest/schreibt							x	
14	ändert							x	
15	initialisiert							x	
16/17	liefert/holt							x	
18	testet							x	
19	sperrt(maskiert)								x
20/21	beendet/gestart. von								x
22	belegt								x
23	verfügbar in		x						
24	hat Priorität (s.u.)	x							
25	dauert	x							
26/27	hat Verzög./Periode								
28/29	lesbar/schreibbar in		x						
30	ist Verbund aus			x					
31	ist Zeiger auf			x					
32	Verbund-Typ aus			x					
33	ist Feld aus			x					
34	hat Werte			x					
35	hat Typ			x					
36/37	leerbar/füllbar in		x						
38/39	kann starten/beenden		x						
40	maskierbar von		x						
41	hat Kapazität			x					
42/43	stat./dyn. Sp.bedarf	x							
44	hat Stichwort	x							
45	hat Verweis	x							

Zu den Bezeichnungen der Spalten siehe nächste Seite. Die Priorität kann je nach Betriebssystem umgesetzt werden, siehe 5.8.2.

Die Spaltenbezeichnungen in Tabelle 5.1 bedeuten:

Kommentare	werden nicht formal abgebildet
Restriktionen	werden vor der Abbildung überprüft
Typ- und Datendefinitionen	schließen die Monitore ein
Prozedur-Definitionen	betrifft die Parameterliste
Ablaufstrukturen	Aufrufsequenzen, Schleifen, Task-Koordinierung
Parameter-Besetzung	beim Prozeduraufruf
Transfers	Zugriffe auf Daten und Betriebsmittel über Monitore, müssen vom Programmierer fertiggestellt werden
Automatischer Zugriff	Zugriffe, für die der Code automatisch aus der ESPRESO-Darstellung entwickelt werden kann

5.6 Zur Verwendung einer verfügbaren Programmiersprache

In den verfügbaren Programmiersprachen sind die genannten Sprachmittel nicht enthalten. Bei der Implementierung eines in ESPRESO-S formulierten Programms treten daher die folgenden Schwierigkeiten auf:

- ESPRESO-S kann partiell mächtiger als die Zielsprache sein, so daß, um die Abbildung zu ermöglichen, Restriktionen zugefügt werden müssen. Z.B. ist die dynamische Erzeugung von Tasks in Concurrent PASCAL und in Modula ausgeschlossen.

- Für Schutzmechanismen aus ESPRESO-S kann in der Zielsprache die Entsprechung fehlen, so daß im Code die angestrebte Sicherheit nicht gewährleistet werden kann. Z.B. gibt es für das Modulkonzept in PASCAL keine Entsprechung. Soll eine Variable in mehreren Prozeduren zugänglich sein, so muß sie in einem umfassenden Block vereinbart werden, wodurch sie im gesamten Block zugänglich wird.

- Den Sprachen liegen Annahmen über die Rechnerstruktur zugrunde, die in ESPRESO-S vermieden wurden. Mit dem Übergang auf die Programmiersprache ist also eine Einengung der möglichen Konfiguration verbunden (vgl. Brinch-Hansen, 1978, S.934, über Concurrent PASCAL und Modula).

Im Beispiel 5.7 wird auch eine Abbildung auf PASCAL-X gezeigt, eine PASCAL-Erweiterung, die sich vom Original vor allem durch die Möglichkeit unterscheidet, Moduln zu definieren ähnlich denen in Modula. Als Eingabesprache für den PASCAL-360-Compiler ist PASCAL-X gut dokumentiert (siehe 4.2.1).

5.7 Ein Beispiel zur Transformation

In die ESPRESO-Datei sei eine im Sinne von 5.4 konsistente und vollständige Systembeschreibung eingegeben worden, die u.a. die Definition einer Prozedur "Hol-Eingabe" enthält. Diese Prozedur wird hier teilweise wiedergegeben, anschließend ihr Abbild in PASCAL-X.

```
procedure Hol-Eingabe:
¢ die Meßwerte werden gelesen und geprüft. ¢;

sequential
   block Einlesen: ..... end Einlesen;

then
   block Plausib-Prüfung:
   text ¢ vergleicht die Meßwerte mit Grenzwerten und gibt
                          die korrekten Werte oder null aus. ¢;
   consumes Meßwert
      where ¢ !Meßwert liegt zwischen 0 und 150. ¢ end;
   reads Obergrenze, Untergrenze;
   produces geprüfter-Wert;
   end Plausib-Prüfung

end Hol-Eingabe;
```

Die Prozedur Hol-Eingabe ist nicht terminal. Der E-PASCAL-Code für sie kann daher vollständig automatisch erzeugt werden. Er wird im Hauptmodul eingeordnet und lautet:

```
procedure Hol-Eingabe;

interface
   calls Einlesen, plausib-Prüfung
interfend;

begin (* Hol-Eingabe *)
   (* die Meßwerte werden gelesen und geprüft. *)
   Einlesen;
   Plausib-Prüfung
end;
```

In PASCAL-X entfällt die Schnittstellen-Deklaration. Sie ist bei den automatisch erzeugten Prozeduren ohnehin entbehrlich.

Der Block Plausib-Prüfung ist dagegen terminal und enthält Aktionen, die vom Programmierer noch ausgeführt werden müssen. Er wird wie folgt abgebildet:

```
procedure Plausib-Prüfung;

interface
   consumes Meßwert
      (* assertion: !Meßwert liegt zwischen 0 und 150. *);
   reads Obergrenze, Untergrenze;
   produces geprüfter-Wert
interfend;

begin
   (* vergleicht die Meßwerte mit Grenzwerten und gibt
                        die korrekten Werte oder null aus. *)
end;
```

Das PASCAL-X-Programm wird durch die Substitution der Schnittstellen-Deklarationen wesentlich umfangreicher als das E-PASCAL-Programm:

```
external program plausib-prüfung;

procedure consume-meßwert (meßwert:  real);   external;
   (* assertion: !Meßwert liegt zwischen 0 und 150. *)

procedure read-obergrenze  (obergrenze:  real); external;

procedure read-untergrenze (untergrenze: real); external;

procedure prod-geprüfter-wert
                    (var geprüfter-wert: real); external;

begin
   (* vergleicht die Meßwerte mit Grenzwerten und gibt
                        die korrekten Werte oder null aus. *)
end;
```

Die als "external" deklarierten Prozeduren bilden jeweils einen eigenen PASCAL-X-Modul zur Verwaltung einer Variablen oder eines Puffers. Sie ersetzen sowohl die statischen Prüfungen (Zugriffsrichtung, z.B. nur schreibend) als auch die dynamische Koordinierung der Zugriffe durch Monitore in E-PASCAL.

5.8 Betriebssystem-Funktionen für E-PASCAL

5.8.1 Vorbemerkungen

Als Prozeßrechner-Programmiersprache enthält E-PASCAL Elemente, deren Ausführung die Zustandsänderung von Rechenprozessen (3.1.1.1.5) bewirkt oder bewirken kann; z.B. muß ein Prozeß, der auf einen Puffer zugreift, mit konkurrierenden Prozessen koordiniert und dabei eventuell in einen Wartezustand versetzt werden. Da die Rechenprozesse zumindest auf konventionellen Rechnern von einem Betriebssystem verwaltet werden, die Koordinierung also nicht selbständig vornehmen können, werden für solche Sprachelemente im Maschinen-Code Aufrufe des Laufzeitsystems abgesetzt. Dieser Abschnitt skizziert die Implementierung des E-PASCAL-Laufzeitsystems auf der Basis eines Standard-Betriebssystems. Er zeigt, daß E-PASCAL nur mäßige Anforderungen an das Betriebssystem stellt und mit einem einfachen Laufzeitsystem auskommt.

Die Unterscheidung zwischen Laufzeit- und Betriebssystem bedeutet nicht, daß es dafür auch getrennte Programme geben muß. Vielmehr können beide gemeinsam generiert werden, zugeschnitten auf die spezielle Anwendung.

Entsprechend der Definition in 3.1.1.1.5 entstehen und verschwinden Prozesse durch die Ausführung einer Rechenvorschrift, die Vater einer parallelen Familie ist. Zustandsänderungen der Prozesse werden von den Monitoren vorgenommen: Ein auf ein Medium zugreifender Prozeß wird, falls erforderlich, in einen Wartezustand versetzt, bis es verfügbar geworden ist. Das Laufzeitsystem muß also eingeschaltet werden, wenn der aus folgenden ESPRESO-Elementen erzeugte Code ausgeführt wird:

a) Parallele Zerlegung einer Rechenvorschrift,

b) Zugriffe auf Variablen (lesen und schreiben),

c) Zugriffe auf Puffer und Trigger (liefern, abnehmen, testen, initialisieren, starten und beenden lassen, maskieren),

d) Belegung von Betriebsmitteln (belegen).

Nicht diskutiert werden diejenigen Funktionen, die zum Start des Systems und zu seiner Beendigung sowie zur Ein- und Ausgabe nötig sind.

5.8.2 Ein Standard-Betriebssystem

Da Betriebssysteme bisher nicht standardisiert sind, wird in dieser Arbeit auf den Report des Ausschusses für Betriebssysteme im European Purdue Workshop, nachfolgend kurz TC8-Report genannt, Bezug genommen (Lalive d'Epinay, 1979). Soweit der TC8-Report lückenhaft ist, werden hier Ergänzungen vorgenommen (5.8.7).

Die Synchronisationselemente des TC8-Reports sind Datenstrukturen mit folgenden Eigenschaften: Ein Synchronisationselement besteht aus einer Liste von Prozessen (process-list) und einer Liste von Botschaften (information-list). Zu jedem Zeitpukt ist mindestens eine der Listen leer. Die einzigen zulässigen Operationen auf diesen Typ sind INC und DEC; beide haben als ersten Parameter den Namen des Synchronisationselementes, als zweiten eine Botschaft bzw. eine Variable für diese.

Zustand d. Listen	Effekt von INC(S,I)	Effekt von DEC(S,V)
information-list enthält Einträge	I aufnehmen in die information-list.	Einen der Einträge aus der information-list nehmen und in V an den Prozeß übergeben.
beide Listen leer		Den Prozeß in die process-list aufnehmen und in Wartezustand versetzen.
process-list enthält Einträge	einen Prozeß aus der Liste nehmen, diesem I übergeben, Prozeß weiterlaufen lassen.	

Dieser Abschnitt wird zeigen, daß INC und DEC stets ausreichen, wenn nicht eine dynamische Taskgenerierung erforderlich ist.

Die Synchronisationselemente werden als Semaphore bezeichnet, wo der Botschaftenmechanismus, der sie von den Semaphoren unterscheidet, nicht beansprucht wird. In diesem Fall wird das zweite Argument bei DEC und INC (für die Botschaft) weggelassen.

Der TC8-Report macht zu einigen für ESPRESO wesentlichen Punkten keine Aussagen. Es wurden daher folgende Möglichkeiten unterstellt:

- An der Schnittstelle zwischen Betriebssystem und Anwendungsprogrammen erscheinen Interrupts als INC-Operationen; in der zugeordneten Information ist die Ursache (Quelle) des Interrupts angegeben. Ein zusätzlicher Prozeß (der Teil des E-PASCAL-Laufzeitsystems sein könnte) setzt die Interrupts um in Lieferungen an die ihnen zugeordneten Trigger.

- Eine Hardware-Uhr ist vorhanden; sie erzeugt in geeigneten Intervallen Interrupts, die ebenfalls von dem oben beschriebenen Prozeß zur Auswertung der Interrupts verarbeitet werden. nach Maßgabe der Tabelle mit Startzeiten, Verzögerungen und Zyklen werden die aktiven Trigger von diesem Prozeß versorgt (vgl. Nehmer, Goos, 1978, S.24; Nehmer, 1979, S.1048). Die Uhrzeit kann jederzeit aus einer Uhrzelle gelesen werden.

- Die Priorität eines Prozesses kann in dessen process-descriptor eingetragen werden und wird bei Ausführung der REDISPATCH-Operation berücksichtigt.

5.8.3 Parallelität

Erfordert das System die dynamische Erzeugung von Rechenprozessen (vgl. 3.4.3.1), so werden die im TC8-Report nur skizzierten Funktionen "create" und "delete" erforderlich. Dieser Fall wird hier nicht weiter verfolgt, es wird vielmehr angenommen, daß die Zahl der Prozesse von Anfang an festliegt und nicht zu hoch für das Betriebssystem ist.

Jeder Prozeß erhält einen privaten Semaphor, auf den er zu Beginn eine DEC-Operation ausführt. Am Ende steht ein INC für den Semaphor des Aufrufers, dann beginnt der nächste Ausführungszyklus. Der aufrufende Prozeß startet die parallelen Unter-Prozesse, indem sie jeden der Semaphore erhöht und anschließend den eigenen pro Unter-Prozeß um eins senkt.

```
Übergeordneter Prozeß:   INC(S1), INC(S2), ... , INC(Sn);
                         DEC(S0), DEC(S0), ... , DEC(S0);

Untergeordneter Prozeß:  DEC(Si); ... (Ausführung) ... INC(S0);
```

5.8.4 Variablen-Zugriffe

Um zu verhindern, daß ein Prozeß auf eine Variable zugreift, während diese von einem andern überschrieben wird, und um sicherzustellen, daß die Schreiber nicht dauernd von Lesern blockiert werden können, ist ein Mechanismus erforderlich, der in der Literatur als Lösung des reader/writer-Problems (Courtois, Heymans, Parnas, 1971) beschrieben ist. Dazu genügen Semaphore.

5.8.5 Zugriffe auf Puffer und Trigger

Puffer sind dynamisch die kompliziertesten Objekte in ESPRESO; Trigger lassen sich etwas einfacher handhaben, da bei diesen mit den Zugriffen keine Weitergabe von Information verbunden ist. Nachfolgend werden die Trigger nur dann ausdrücklich erwähnt, wenn ihre Realisierung abweicht.

Einige der Operationen auf Puffer und Trigger lassen sich durch andere darstellen, so daß das Problem reduziert werden kann:

started-by — läßt sich darstellen als consume in einem vorangestellten zusätzlichen Block,

terminated-by — durch test und bedingtes consume am Ende des Blocks,

initializes — durch inhibit und test, abhängig vom Ergebnis dann produce oder consume,

produce mit skip-Verhalten durch inhibit, test und bedingtes produce.

Damit bleiben die Relationen produce, consume (mit block-Verhalten), test und inhibit zu realisieren. Für diese werden nachfolgend die wichtigsten Varianten angegeben.

Vier Fallunterscheidungen ermöglichen eine systematische Behandlung:

- nach dem Umfang der zu übertragenden Information (5.8.5.1),
- nach der Zahl der Plätze im Puffer (5.8.5.2),
- danach, ob der Pufferpegel abgefragt wird (test-Operation) oder ob der Puffer gesperrt wird (inhibit) (5.8.5.3),
- nach der Strategie der Objektverwaltung im Puffer (5.8.5.4).

5.8.5.1 Unterscheidung nach Umfang der Information

Trigger benötigen nur Semaphore.

produce	consume
DEC(Sf); INC(Sb)	DEC(Sb); INC(Sf)

Sf und Sb sind die Synchronisationselemente für freie und besetzte Plätze. Bei einem Trigger mit k Plätzen hat Sf den Anfangswert k, Sb den Anfangswert 0.

Puffer, deren Datentyp es erlaubt, die Speicherung in der Informationsliste eines Synchronisationselementes vorzunehmen, benötigen für die Verwaltung der leeren Plätze ebenfalls nur einen Semaphor; das andere Synchronisationselement stellt den eigentlichen Puffer dar.

DEC(Sf); INC(Sb,I)	DEC(Sb,V); INC(Sf)

I ist der übergebene Gegenstand (die Information), V ist eine Variable desselben Typs. Nachfolgend werden diese Puffer k-Puffer genannt (für "kurz").

Puffer, deren Datentyp die Speicherung in der Informationsliste nicht mehr erlaubt, benötigen für die Verwaltung der leeren und die der vollen Plätze jeweils ein Synchronisationselement. Der Speicherplatz für den Puffer ist zusätzlich anzulegen. Lieferung und Abnahme erfolgen, indem ein Index der einen Liste entnommen und nach Zugriff auf den bezeichneten Platz im Puffer der anderen zugefügt wird.

DEC(Sf,p);	DEC(Sb,p);
Puffer(p) := I;	V := Puffer(p);
INC(Sb,p);	INC(Sf,p);

p identifiziert einen Platz im Puffer, z.B. als Index. Diese Puffer heißen hier l-Puffer (für "lang").

5.8.5.2 Unterscheidung nach der Zahl der Plätze

Bei einem Puffer mit n Plätzen sind folgende Fälle für n zu unterscheiden:

n = 0	"Null-Puffer"
n = 1	"einfacher Puffer"
n > 1, begrenzt	"begrenzter Puffer"
n undefiniert	"unbegrenzter Puffer"

Der begrenzte Puffer ist der (in den Beispielen unter 5.8.5.1 angenommene) Normalfall. Er erfordert eine Verwaltung sowohl der besetzten als auch der freien Plätze. Beim einfachen Puffer wird diese Verwaltung vereinfacht, da ein einziger Platz keine Identifikation erfordert (nur für 1-Puffer von Bedeutung).

Der Nullpuffer wird am einfachsten durch einen einfachen Puffer realisiert, der durch einen Trigger für die Übergabe einer Quittung ergänzt ist ("handshaking"). Unbegrenzte Puffer sind u.U. dadurch einfacher als die begrenzten, daß die Verwaltung der leeren Plätze entfällt. Zwar gibt es streng genommen keine unbegrenzten Puffer, jedoch ist die Zahl der Elemente im Puffer oft durch die Programme begrenzt, so daß ein Pufferüberlauf nicht abgefangen werden muß.

Für k-Puffer lautet die produce-Operation in den genannten Fällen:

Null-Puffer	INC(Sb,I); DEC(Sf) (consume wie oben)
einfacher Puffer	DEC(Sf); Puffer := I; INC(Sb) oder wie begrenzter Puffer
begrenzter Puffer	DEC(Sf); INC(Sb,I)
unbegrenzter Puffer	INC(Sb,I)

Die Realisierung für Trigger und 1-Puffer ergibt sich analog; letztere sind stets begrenzt.

5.8.5.3 Unterscheidung danach, ob test oder inhibit verwendet werden

Wird der Pegel eines Puffers abgefragt, so ist eine kompliziertere Behandlung notwendig. Wie oben gesagt findet eine solche Abfrage implizit auch durch die Zugriffe initialize und produce mit skip-Verhalten statt. Ein Zähler, der durch einen Semaphor zu schützen ist, muß stets angeben, wieviele Prozesse auf Lieferungen oder Abnahmen aus einem Puffer warten und wieviele Gegenstände im Puffer liegen. Die Realisierung erfolgt bei der produce-Operation durch Vorschalten der Befehle

```
DEC(Sz); n := n + 1; INC(Sz);
```

Bei consume wird das Plus durch ein Minus ersetzt. Die Abfrage ("test") greift dann auf den aktuellen Wert von n zu:

```
DEC(Sz); result := n; INC(Sz);
```

Wird ein Puffer gesperrt (inhibit), so ist eine ähnliche Konstruktion notwendig: Jeder zugreifende Rechenprozeß benötigt zunächst den exklusiven Zugriff auf n. Die Sperrung kann also durch Belegen des Semaphors Sz erfolgen:

```
DEC(Sz); ... (sperrende Rechenvorschrift) ... INC(Sz)
```

In der sperrenden Rechenvorschrift entfallen alle DEC- und INC-Operationen auf Sz. Die Sperrung wirkt nicht auf Prozesse, die n bereits verändert haben.

Wie gesagt läßt sich produce mit skip-Verhalten realisieren durch test und bedingtes produce, wobei der Puffer solange gesperrt bleiben muß. Eine elegantere Lösung ist (für k-Puffer):

```
DEC(Sn);
if n < k
   then n := n + 1; INC(Sn); DEC(Sf); INC(Sb,I)
   else INC(Sn)
fi
```

5.8.5.4 Puffer mit definierter Ausgabestrategie

Bisher wurde angenommen, daß die Reihenfolge, in der die Objekte im Puffer an abnehmende Prozesse gereicht werden, keine Rolle spielt. In ESPRESO-S lassen sich jedoch zwei verschiedene Strategien wählen, "fifo" und "by-priority". Nachfolgend werden Erweiterungen des Puffermodells gezeigt, die das gewünschte Verhalten gewährleisten. Es ist allerdings anzunehmen, daß Realisierungen des Betriebssystems, wie es im TC8-Report beschrieben ist, in vielen Fällen bereits automatisch ein FIFO-Verhalten bieten werden, wodurch die oben angegebenen Lösungen für diesen Fall ausreichend sind.

Für ein erzwungenes FIFO-Verhalten erhält der Puffer zwei zusätzliche Zeiger, die jeweils durch einen Semaphor (oder durch einen einzigen) geschützt werden. Die Zeiger weisen auf Anfang und Ende des belegten Bereichs im Puffer.

Die Operationen auf diesen Puffer lauten dann wie folgt:

```
produce                          consume

DEC(Sf);                         DEC(Sb);
   DEC(Se);                         DEC(Sa);
      ne := ne mod k + 1;              p := na;
      p := ne;                         na := na mod k + 1;
   INC(Se);                         INC(Sa);
   Puffer (p) := V;                 V := Puffer (p);
INC(Sb)                          INC(Sf)
```

Bei prioritätsabhängiger Reihenfolge muß eine Funktion "prio" gegeben sein, die für die Einträge im Puffer die Priorität berechnet.

Der Puffer (k Plätze, Typ T) bildet dann die folgende Datenstruktur:

```
sema Sf (k), Sb (0);
type T1: record
           inhalt : T;
           next   : integer
         end;

var Puffer : array (0..k) of T1;
```

Der Puffer ist wie folgt initialisiert:

```
Puffer (0).next = 0; leer := 1;
for i := 1 to k do Puffer(i-1).next := i;
```

produce

```
DEC(Sf);
   DEC(S1);
      neu := leer;
      leer := Puffer(neu).next;
   INC(S1);
   Puffer(neu).inhalt := I;
   zeiger := 0;
   DEC(S2);
      repeat
        alt := zeiger;
        zeiger:=Puffer(alt).next;
     until (zeiger=0) or
      prio((Puffer(zeiger).inhalt)
                      < prio(V));
     Puffer (alt).next := neu;
      Puffer (neu).next := zeiger;
   INC(S2);
INC(Sb);
```

consume

```
DEC(Sb);
   DEC(S2);
      ind := Puffer(0).next;
      Puffer(0).next :=
         Puffer(ind).next;
   INC(S2);
   V := Puffer(ind).inhalt;
   DEC(S1);

     Puffer(ind).next := leer;
      leer := ind;
   INC(S1);
INC(Sf);
```

5.8.6 Belegung von Betriebsmitteln

Die Belegung erfolgt durch die Relation "belegt". Wird nur jeweils ein einziges Exemplar belegt, was insbesondere der Fall ist, wenn es sich um ein virtuelles Betriebsmittel handelt, so ist die Implementierung sehr einfach: Sie erfolgt durch das Paar DEC ... INC. Wo mehrere gleiche Betriebsmittel bereitstehen (z.B. Bandgeräte), kann als Information eine Identifikation des Einzelobjekts beigefügt werden, ähnlich wie bei der Verwaltung der 1-Puffer (5.8.5.1):

```
DEC(S,i); ... (Verwendung des Betriebsmittels i) ... INC(S,i)
```

Für die Realisierung der Uhr-Funktionen (Trigger mit Verzögerung oder Zyklus) ist ein Prozeß nötig, der bei Erreichen des gegebenen Zeitpunkts eine Lieferung an den Trigger ausführt (siehe 5.8.2). Bei der Initialisierung eines Triggers werden diesem Prozeß Verzögerung und Zyklus des Triggers mitgeteilt, bei der Freigabe (am Ende der belegenden Aktivität) streicht er den Trigger aus seiner Buchführung.

Werden mehrere gleiche Betriebsmittel angefordert, so ist ein komplizierterer Mechanismus notwendig.

```
Anfangswerte:    Synchronisationselemente: sd1(1), sd2(0), sd3(0);
                 integer vs := v0, zw := 0;
```

```
Belegung                                  Freigabe

do
   DEC(sd1);                              DEC(sd1);
   if vs >= m                             vs +:= n;
      then vs -:= m;
         INC(sd1);
        goto break
      else zw +:= 1;                      while zw > 0 do
         INC(sd1); DEC(sd2);                 INC(sd2);
         zw -:= 1; INC(sd3)                  DEC(sd3)
   fi                                     od;
od;                                       INC(sd1)
break : skip;
```

5.8.7 Zusammenfassung

Bei der Umsetzung einer ESPRESO-Spezifikation in Code werden relativ geringe Anforderungen an das Echtzeit-Betriebssystem gestellt. Es wurde gezeigt, daß die Operationen auf Synchronisationselemente (INC und DEC) genügen, wenn nicht dynamisch Prozesse kreiert werden müssen; dies erfordert zusätzlich CREATE und DELETE.

Nach Vorschlag des TC8-Reports bewirkt CREATE den Übergang vom Prozeßzustand UNDEFINED nach INACTIVE; DELETE wirkt umgekehrt. Für ESPRESO ist der Zustand INACTIVE ohne Bedeutung, da von den fünf vorgesehenen Übergängen von und nach den übrigen Zuständen (BLOCKED, READY und RUNNING) nur zwei verwendet werden, denn Prozesse können sich nicht gegenseitig beenden. Daher wäre folgendes vereinfachte Schema ausreichend:

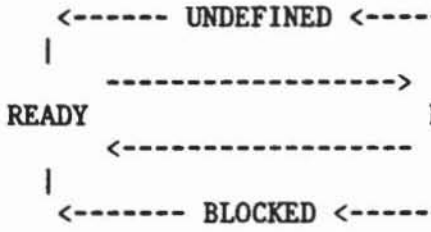


Bild 5.1 vereinfachtes Prozeßzustands-Diagramm

5.9 Optimierung

Ein E-PASCAL-Compiler sollte den erzeugten Code durch folgende Maßnahmen optimieren:

- Prozeduren, die an nur einer Stelle aufgerufen werden, können als Makros behandelt werden. Dies macht Aufruf, Parameterübergabe und Rücksprung überflüssig.

- Monitore können durch direkten Zugriff auf die Daten ersetzt werden, wo alle Prüfungen statisch möglich sind. Dazu ist es hinreichend, daß die Schreibzugriffe auf eine Variable in Prozessen liegen, die niemals parallel ablaufen zu anderen, ebenfalls auf diese Variable zugreifenden Prozessen. Das gilt z.B., wenn eine Variable nur während der (allein ablaufenden) Initialisierung des Programms gesetzt wird.

- Bei Puffern sind in vielen Fällen Semaphore entbehrlich, wo nur ein Prozeß als Lieferant und/oder als Abnehmer auftritt. Ein Beispiel ist der Puffer mit FIFO-Verhalten, der in 5.8.5.4 angegeben ist. Handelt es sich nur um zwei Prozesse, die durch diesen Puffer gekoppelt sind, so können die Semaphore Se und Sa und die Operationen darauf entfallen, denn ne und na sind jeweils lokal zu einem Prozeß.

Im Laufzeitsystem genügt für alle Variablen ein einziger Monitor, der entsprechend parametrisiert ist. Das gleiche gilt für jede der in 5.8.5 genannten Klassen von Puffern und Triggern.

5.10 ESPRESO und CIP

Auch wenn ESPRESO durch ein Mittel zur Umsetzung in eine Programmiersprache erweitert ist, deckt es nur den ersten Teil der Programmentwicklung ab, denn das erzeugte Programm ist unvollständig und in den meisten Fällen extrem ineffizient. Daher sind weitere Mittel erforderlich, um das Ziel, ein korrektes, vollständiges und praktisch brauchbares Programm, zu erreichen.

Hier bieten sich Systeme zur Programmentwicklung durch formale Transformation, als deren Vertreter hier CIP (Bauer, 1979) betrachtet wird, zur Fortsetzung an. Dieser Abschnitt untersucht, wie der Übergang von ESPRESO nach CIP erfolgen kann (5.10.1) und welche konzeptionellen Probleme dabei noch zu lösen sind (5.10.2).

5.10.1 Anwendungsbereiche

ESPRESO dient zur Sammlung der Information, die schließlich insgesamt eine (nicht völlig formale) Spezifikation bildet. CIP hat eine zwar im allgemeinen nicht operationelle, aber vollständige und formale Spezifikation als Ausgangspunkt. Stellt man den Anteil der formalisierten Information einer Spezifikation über der Zeit dar, so entsteht eine Linie, die mit der Idee bei 0 beginnt, mit der formalen Spezifikation, spätestens jedoch mit dem codierten Programm und der Wahl der Zielmaschine, bei 100 endet. Systeme wie ESPRESO und CIP sind, wie oben angedeutet, jeweils in einem bestimmten Teilgebiet anwendbar.

Bild 5.2 Grad der Formalisierung

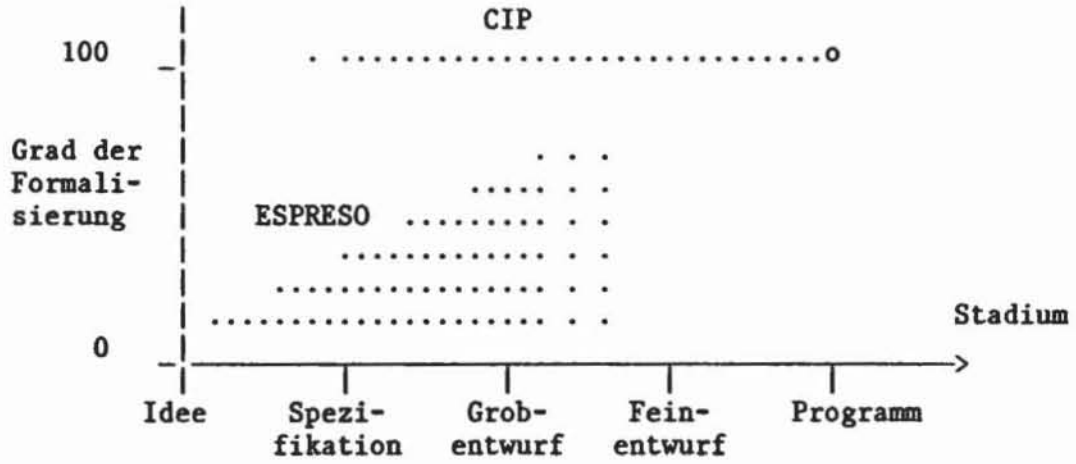


Bild 5.2 enthält zwei Aussagen zum Übergang von ESPRESO nach CIP:

- Da es d i e Spezifikation nicht gibt (vgl. 2.2.2), ist auch kein Punkt in der Programmentwicklung a priori ausgezeichnet für den Wechsel. Vielmehr gibt es einen Grenzbereich, in dem beide Systeme anwendbar sind.

- Die für CIP erforderliche vollständige Formalisierung kann mit ESPRESO nicht erreicht werden, solange Sprachmittel für die Algorithmen fehlen. Daher bedarf eine in die Sprache CIP-L (Bauer et al., 1978) oder in eine andere Programmiersprache übertragene Spezifikation noch der Ergänzung.

Da ESPRESO bisher nicht auf CIP oder ein entsprechendes System abgestimmt ist, enthält es auch Ansätze, die andernfalls besser in jenes andere System passen würden. Z.B. wäre es vorzuziehen, die in 3.4.3.1 beschriebenen Transformationen unter voller Kontrolle durch ein Transformationssystem vorzunehmen.

5.10.2 Begriffliche Besonderheiten der Prozeßrechnerprogramm-Spezifikation

Lautet die Fragestellung bei der Spezifikation eines typischen Programms für die Stapelverarbeitung "Was e r z e u g t (oder berechnet) das Programm?", so lautet sie bei einem Echtzeitsystem "Was t u t (oder bewirkt) das System?".

Die Sprache CIP-L ist, soweit sie bisher bekannt gemacht wurde, auf Probleme ausgerichtet, die durch Stapelverarbeitung gelöst werden können. Daher wird in erster Linie mit F u n k t i o n e n (im mathematischen Sinne) spezifiziert, deren Werte gewissen Bedingungen relativ zu den Eingangsgrößen genügen. Der Zusammenhang ist statisch, die Zeit spielt keine Rolle.

Demgegenüber ist die Zeit für ESPRESO von besonderer Bedeutung: Reaktionen des Systems auf Zustandsänderungen in der Umgebung müssen innerhalb einer gewissen Zeit erfolgen, Ein- und Ausgaben erfolgen dynamisch. Daher muß zu jeder zeitlich unabhängigen Komponente eines Echtzeit-Programms angegeben werden, wann sie ausgeführt wird.

Aus diesem Grund hat im Begriffssystem von ESPRESO die P r o z e d u r eine zentrale Stellung. Sie ist statisch und damit der Beschreibung und Analyse zugänglich, hat andererseits durch die Ablaufstrukturen und die Koordinierungsfunktionen dynamische Beziehungen, die dem Echtzeitaspekt Rechnung tragen (vgl. Balzer, Goldman, 1979, S.59).

Es ist zu erwarten, daß durch die vorgesehenen Erweiterungen von CIP, wie sie sich aus der Arbeit von Broy (1980) ergeben, die für ESPRESO erforderliche prozedurale Ausstattung verfügbar wird.

6. Beispiel und Kritik

6.1 Ein Beispiel: die Paketverteilanlage

6.1.1 Zur Auswahl des Beispiels

Ein Beispiel sollte klein, verständlich und praxisnah sein, alles wesentliche zeigen und nicht gerade so ausgesucht sein, daß es nur die positiven Seiten vorführt. In der Regel lassen sich nicht alle diese Ziele erreichen. Das hier gewählte Problem wurde im Rahmen des PDV-Arbeitskreises "Systematische Entwicklung von PDV-Systemen" als Standard-Beispiel ausgewählt. Es genügt den oben genannten Kriterien bis auf den Nachteil, daß kontinuierliche Elemente im technischen Prozeß fehlen. Die Gegenüberstellung mit anderen Spezifikationssprachen (Hommel, 1980) ermöglicht den direkten Vergleich.

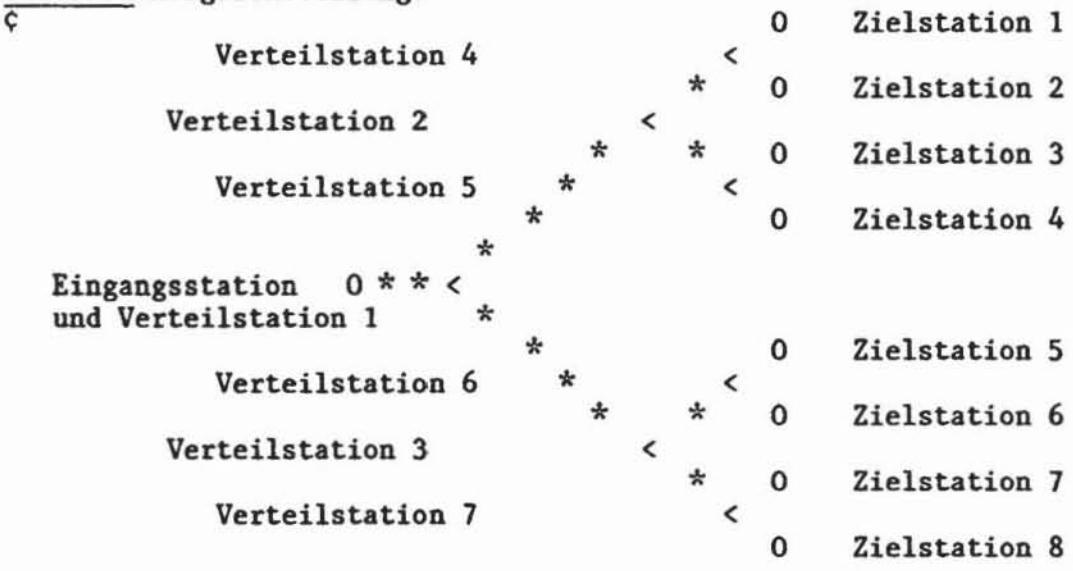
Da ESPRESO die E r s t e l l u n g der Spezifikation unterstützen soll, wird hier nicht nur die resultierende Formulierung gezeigt, sondern der Prozeß ihrer Entstehung verfolgt. Dadurch wird erkennbar, wie sich die Spezifikation formt in Wechselwirkung zwischen Aufgabenstellung und Sprachmitteln. Diese Wechselwirkung ist beabsichtigt und führt zu einer relativ übersichtlichen Formulierung.

Die Spezifikation hält sich streng an die Aufgabenstellung; das bedeutet vor allem, daß die zahlreichen denkbaren, aber in der Aufgabe ausgeschlossenen Defekte der Anlage nicht berücksichtigt werden.

6.1.2 Verbale Formulierung der Aufgabe

Die Aufgabe kann zunächst durch Text-Objekte in ESPRESO-S aufgenommen werden; sie ist bis auf "redaktionelle" Änderungen unverändert.

<u>informal</u> Aufgabenstellung:



Für eine Paketverteilungsanlage, bei der die Pakete auf acht verschiedene Zielstationen gelenkt werden, soll das Steuersystem entwickelt werden. Es wird angenommen, daß innerhalb der Anlage Pakete weder verlorengehen noch hinzukommen. Auch arbeiten sämtliche Komponenten der Anlage stets fehlerfrei, und die Pakete können sich nicht verklemmen. Allerdings muß damit gerechnet werden, daß die Pakete nicht alle gleichschnell durch die Anlage laufen. Das Steuersystem, d.h. der Prozeßrechner, ist schnell im Vergleich zu den Bewegungsvorgängen im technischen Prozeß (Bewegungen der Weichen und Pakete), es werden also keine Probleme verursacht durch die Verarbeitungszeiten im Rechner.
¢ end;

informal Eingangsstation:
¢ Die Eingangsstation besteht aus einem Freigabeorgan mit den Teilen F1 und F2. F2 hält das einlaufende Paket so lange fest, bis das Meldeorgan die Ankunft an das Steuersystem gemeldet hat und dieses mit Hilfe des Leseorgans das Codezeichen aufgenommen hat. Bei fehlendem oder unzulässigem Paketcode wird eine spezielle Zielstation angesteuert (hier: Zielstation 8).
Danach gibt das Steuersystem einen Auftrag an das Freigabeorgan, die Sperre F2 gibt den Weiterlauf für das Paket frei und das Beschleunigungsteil F1 neigt sich. Dadurch gleitet das Paket weiter. Gleichzeitig wird das nächste Paket am Einlaufen gehindert, bis die Sperre F2 wieder eingetreten ist.
Bei der Behandlung des nachfolgenden Pakets ist zu prüfen, ob sich sein Ziel von dem des Vorläufers unterscheidet. In diesem Fall ist die Freigabe seines Weiterlaufs so lange zu verzögern, daß die Lenkorgane zwischen den beiden Paketen sicher umgestellt werden können. Im andern Fall können die Pakete unmittelbar aufeinander folgen.
¢ end;

informal Verteilstation:
¢ Ein- und Ausgangspunkte jeder Verteilstation sind mit Lichtschranken versehen. Diese können aufgrund ihrer technischen Ausführung auch dicht aufeinanderfolgende Pakete mit Sicherheit einzeln erkennen. Die Meldungen werden im Steuersystem zur Laufwegverfolgung jedes einzelnen Pakets ausgewertet. Dadurch kann in Verbindung mit dem bereits ermittelten Ziel der Steuerauftrag für das nächste Lenkorgan ermittelt und ausgegeben werden.
Beim Ausgeben des Steuerauftrages ist darauf zu achten, daß alle Vorläufer die betreffende Verteilstation passiert haben. Die Verteilstation selbst muß frei sein, d.h. zwischen Ein- und Ausgangslichtschranke darf sich kein Paket befinden.
Tritt aufgrund unterschiedlicher Geschwindigkeiten der Fall ein, daß ein Paket, für das die Verteilstation umzustellen ist, den Eingangsmeldepunkt erreicht, bevor der Vorläufer die Station verlassen hat, muß die Umstellung unterbleiben. Er wird zum Falschläufer und bekommt das Ziel seines Vorläufers. Für jeden Falschläufer soll genau eine Meldung mit Soll- und Ist-Ziel ausgegeben werden.
¢ end;

informal Basismaschine:
¢ Die Aufgabe enthält auch zwei Angaben zur Basismaschine:
- Als Konfiguration ist ein einzelner Prozeßrechner vorzusehen.
- Die Programmierung erfolgt in einer höheren Sprache.

Auf die ESPRESO-Spezifikation haben diese Angaben keinen Einfluß.
¢ end.

6.1.3 Erste Formulierung des Steuersystems

```
module Paketverteil-Anlage:
comprises
   procedure Steuerung:
   ¢ Steuerung verarbeitet die Signale aus dem technischen Prozeß,
     steuert das Freigabeorgan und erzeugt, wenn notwendig, Fehler-
     meldungen.
   ¢;
   end
and
   trigger Eingangsmeldung:
   ¢ wird erhöht, wenn ein Paket in die Eingangsstation einläuft ¢;
   end
and
   variable Adresse:
   ¢ verfügbar, wenn die !Eingangsmeldung eingetroffen ist ¢;
   end
and
   variable Freigabe:
   ¢ wird vom Rechner nur geschrieben, vom Prozeß "gelesen" ¢;
   values Einlaufstellung, Auslaufstellung;
   end
and
   trigger Einlauf-VS1:
   ¢ wird erhöht, wenn Paket in die Verteilstation 1 einläuft ¢;
   end
and
   trigger Auslauf-VS1:
   ¢ wird erhöht, wenn Paket links oder rechts aus der Verteilstation 1
     ausläuft ¢;
   text unklar
   ¢ Nach Beschreibung der !Eingangsstation ist es nicht nötig, zwischen
     linkem und rechtem Ausgang zu unterscheiden. Falls die Zusammen-
     fassung der beiden Signale nicht möglich ist, sind zwei zusätzliche
     Blöcke nötig.
   ¢;
   (* "!Eingangsstation" ist ein Querverweis, siehe 3.2.3.1 *)
   end
and
   trigger Einlauf-VS2:
   ...
   ...              (* VS2 bis VS7 entsprechend VS1 *)
   ...
   end Auslauf-VS7
end Paketverteil-Anlage.
```

Zur Auslassung für die Trigger Einlauf-VS2 bis Auslauf-VS7 siehe 6.2 e.

In dieser Spezifikation wurde bisher nur die - in ESPRESO-S selbstverständliche - Zuordnung der Interrupts zu Triggern vorgenommen. Im übrigen ist die Aufgabenstellung noch unverändert.

6.1.4 Modularisierung

Aus der räumlichen Anordnung des Systems folgt direkt die naheliegende Modularisierung (vgl. 3.4.2.1):

```
module Paketverteil-Anlage:

comprises
   module Eingangsstation
and
   module VS1
and
   module VS2
   ...
and
   module VS7
and
   procedure Steuerung:
   parallel Eingangsbearbeitung
   parallel VS1-Verwaltung
   parallel VS2-Verwaltung
   ...
   parallel VS7-Verwaltung
   end Steuerung
end    Paketverteil-Anlage.
```

Trigger und Variablen, die oben als Teile des Hauptmoduls beschrieben wurden, werden nun den einzelnen Teilen zugeordnet (z.B. kommen Einlauf-VS1 und Auslauf-VS1 in den Modul VS1 und sind damit nur dort verfügbar). In jedem dieser Submoduln gibt es einen Block, der die Steuerung an dem jeweiligen Punkt übernimmt.

Für die Kommunikation zwischen diesen Moduln gibt es zwei Möglichkeiten, die zentrale, bei der ein zusätzlicher Modul oder der Modul "Eingangsstation" als einziger mit den übrigen Moduln kommuniziert, oder die dezentrale, bei der die Moduln untereinander kommunizieren. Die zentrale Lösung hat Nachteile, vor allem im Falle fehllaufender Pakete. Daher wird hier die dezentrale Lösung gewählt.

In Analogie zum physischen Fluß der Pakete, z.B. von der Verteilstation 1 zur Station 2, gibt es Puffer, durch die die Begleitinformation zu den Paketen übermittelt wird, im Beispiel also zwischen VS1 und VS2:

```
buffer Weitergabe-1-2:
¢ enthält Angaben zu Paketen, die !VS1 verlassen,
  aber !VS2 noch nicht erreicht haben ¢;
produce restricted-to VS1;
consume restricted-to VS2
end Weitergabe-1-2.
```

Dieser Puffer wird global im Modul Paketverteil-Anlage definiert. Durch die Restriktionen ist jedoch der Zugriff beschränkt auf die Moduln VS1 und VS2. Entsprechende Puffer werden zwischen allen direkt verbundenen Verteilstationen definiert, auch zwischen Eingangsstation und VS1.

6.1.5 Eingangsstation und Verteilstationen

Es folgt nun der wesentliche Teil der Arbeit, die Definition der Eingangsstation und der Verteilstationen.

```
module Eingangsstation:

comprises
    trigger   Eingangsmeldung
and variable Adresse
and variable Freigabe
and
    block      Eingangsbearbeitung:
    while      Automatik;
    started-by Eingangsmeldung;
    produces   Weitergabe-E-1;
    reads      Adresse;
    writes     Freigabe
    end
end Eingangsstation.
```

Die jetzt in diesem Modul definierten Objekte entfallen natürlich im übergeordneten. Es wird hier darauf verzichtet, diese Verschiebungen jeweils im Detail darzustellen. Auch werden hier und nachfolgend weniger Texte zu den Objekten angegeben, als in einer echten Spezifikation, der der begleitende Kommentar fehlt, sinnvoll wäre.

Als Repräsentant der Eingangsstationen wird nachfolgend VS2 beschrieben; VS3 ist völlig entsprechend. VS1 und VS4 bis VS7 sind geringfügig verschieden, VS1 wegen seiner Sonderstellung zur Eingangsstation, die übrigen, da sie keine Begleitinformation weiterreichen, aber eventuell Meldungen erzeugen müssen.

```
module VS2:

text unklar
¢ aus der Aufgabe ist nicht ersichtlich, wie die Umschaltung der Weiche
  erfolgt. Hier wird die Weiche als Variable !Weiche-2 definiert, wie
  schon die Freigabe. ¢;

comprises
    trigger Einlauf-VS2
and trigger Auslauf-VS2
and
    block VS2-Verwaltung:
    ¢ überwacht Verteilstation 2, stellt, wenn nötig und möglich, die
      Weiche 2 um, ändert bei Fehlläufern die Begleitinformation, falls
      nötig, und gibt sie weiter an die nächste Station. ¢
    end
and
    variable  Weiche-2:
    ¢ repräsentiert die reale Weiche im technischen Prozeß ¢;
    values    links, rechts
    end
end VS2.
```

6.1.6 Verfeinerung der Prozeduren

6.1.6.1 Eingangsbearbeitung

Die Struktur des Blocks Eingangsbearbeitung geht aus der Aufgabenstellung hervor, die den Ablauf vorgibt: (1) Paket läuft ein, (2) Paket wird freigegeben und an VS1 weitergemeldet, (3) Eingangsstation wird für das nächste Paket bereitgemacht. Unklar ist nur, wie die Wartezeit auszudrücken ist. Offenbar muß die Dauer bis zur Freigabe des nächsten Pakets einen bestimmten Mindestwert haben, wenn die Adressen unterschiedlich sind. Daraus folgt, daß nach Schritt (1) ein Warteschritt einzufügen ist, dessen Dauer von der Adreßsequenz und davon abhängt, wieviel Zeit bereits seit der letzten Freigabe vergangen ist.

```
block       Eingangsbearbeitung:
while       Automatik;
started-by Eingangsmeldung;
sequential
   Adresse-lesen:
   reads     Adresse, Zeit;
   updates   Vorgänger-Adresse, Vorgänger-Startzeit;
   writes    Wartezeit;
   produces Weitergabe-E-1;
   end Adresse-lesen

then
   Freigeben:
   delay  Wartezeit of sec;
   (* dies ist eine in ESPRESO-S vorgesehene implizite Beschreibung
      eines Triggers, der den Block nach einer Wartezeit startet.     *)
   writes Freigabe where ¢ := Auslaufstellung ¢;
   end Freigeben

then
   Vorbereiten:
   text unklar
   ¢ Die Aufgabe läßt offen, ob die Eingangsstation nach vorgegebener
     Zeit oder nach Quittierung des Pakets durch VS1 wieder in die
     Ausgangstellung gehen soll. Hier wird der (sinnvolle) zweite Fall
     angenommen.
   ¢;
   started-by Quittung-1-E;
   writes     Freigabe where ¢ := Einlaufstellung ¢
   end Vorbereiten

end Eingangsbearbeitung;


module Paketverteil-Anlage:
comprises
   trigger Quittung-1-E:
   ¢ bestätigt den Einlauf in !VS1 und beendet damit den Wartezustand
     in der !Eingangsbearbeitung ¢;
   produce restricted-to VS1;
   consume restricted-to Eingangsstation;
   end Quittung-1-E
end    Paketverteil-Anlage.
```

6.1.6.2 Eine repräsentative Verteilstation

Der schwierigste Teil der Aufgabe ist die Definition der Verteilstationen. In diesen laufen jeweils zwei durch den physischen Prozeß gekoppelte Prozesse ab, der Einlauf und der Auslauf der Pakete. Asynchron werden außerdem Pakete gemeldet.

Hier wird zunächst eine Spezifikation entwickelt, wie sie ohne lange Vorüberlegungen entsteht. Eine elegantere Lösung folgt unter 6.1.8.

Offensichtlich sind für die Ein- und Ausgangsmeldungen zwei parallele Prozesse erforderlich. Weniger leicht ist die Einordnung der Ankündigungen neuer Pakete, die von VS1 weitergereicht werden. Nach einiger Überlegung wird klar, daß ein spezieller Prozeß für diesen Zweck streng vom Auslaufprozeß kontrolliert würde, so daß es einfacher ist, ihn gleich dort zu integrieren.
Für die Buchführung über ein- und auslaufende Pakete sind Variablen und Trigger nötig, deren Inhalt zu jedem Zeitpunkt konsistent sein muß. Daher werden sie logisch zusammengefaßt im Betriebsmittel "Buchführung", dessen Belegung den exklusiven Zugriff sichert.

```
block VS2-Verwaltung:

parallel
   VS2-Einlauf:
   while      Automatik;
   started-by Einlauf-VS2;
   occupies   Buchführung-2;

   sequential
      VS2-E1:
      produces Pakete-in-Weiche-2; (* zählt die Pakete in der Weiche, *)
      tests    akt-Paket-2;        (* prüft, ob Weiche schon umge-    *)
      writes   Weiche-2-frei       (* stellt, notiert Ergebnis.       *)
         where ¢ := true, falls !akt-Paket-2 nicht leer ¢

      controlled-by Weiche-2-frei;

      either key true for
         VS2-E11:  end             (* keine Aktion notwendig          *)

      or key false for
         VS2-E12:
         consumes Weitergabe-1-2;     (* holt nächste Anmeldung       *)
         reads    Weichenstellung-2;  (* stellt Stand der Weiche fest *)
         produces akt-Paket-2
            where ¢ das Soll-Ziel wird unverändert übernommen,
                    das Ist-Ziel richtet sich nach !Weichenstellung-2.
                    Sind die beiden unterschiedlich, so handelt es
                    sich um einen Fehlläufer.
                  ¢;
         end VS2-E12
      end VS2-E1
```

```
   then
      VS2-E2:
      controlled-by Weichenstellung-2;

      either key links for
         VS2-E2-links:
         consumes akt-Paket-2;
         produces Weitergabe-2-4;
         end VS2-E2-links

      or key rechts for
         VS2-E2-rechts:
         consumes akt-Paket-2;
         produces Weitergabe-2-5;
         end VS2-E2-rechts
      end VS2-E2

   end VS2-Einlauf


parallel
   VS2-Auslauf:
   while      Automatik;
   started-by Auslauf-VS2;
   occupies   Buchführung-2;        (* symmetrisch zu VS2-Einlauf   *)

   consumes Pakete-in-Weiche-2;
   tests    Pakete-in-Weiche-2;
   writes   Weiche-2-leer where ¢ := true, falls leer ¢;

   controlled-by Weiche-2-leer;
   either key true for
      VS2-A1:                       (* Ist die Weiche leer, so wird *)
      consumes Weitergabe-1-2;      (* das nächste Paket erwartet,  *)
      produces akt-Paket-2;         (* notiert und die Weiche ggf.  *)
      writes   Weiche-2,            (* gestellt.          Die neue  *)
               Weichenstellung-2;   (* Position wird gespeichert.   *)
      end VS2-A1
   or key false for
      VS2-A2: ¢ Umschalten nicht möglich ¢ end
   end VS2-Auslauf

end VS2-Verwaltung.
```

6.1.6.3 Die anderen Verteilstationen

Der Modul VS1 unterscheidet sich, wenn man analoge Namen für die Objekte voraussetzt, nur durch Erzeugung der Quittung in VS1-E1:

```
block VS1-E1:
produces Pakete-in-Weiche-1, Quittung-1-E;
tests    akt-Paket-1;
...
```

Weiter analog zu VS2.

VS4-Einlauf bis VS7-Einlauf unterscheiden sich am Ende von VS2-Einlauf:

```
block VS4-E2:
consumes akt-Paket-4;
writes   Fehlläufer-in-4;

controlled-by Fehlläufer-in-4;
either key true for
   Meldung-aus-VS4:                (* Hier fehlt noch die Angabe,     *)
   produces Meldungen;             (* woraus die Meldung erzeugt wird *)
   end
or key false for
   VS4-E2-OK: (* alles klar, kein Fehlläufer *) end
end VS4-E2.
```

6.1.6.4 Die Erzeugung von Meldungen

Es fehlt noch ein Prozeß, der den Meldungspuffer abbaut:

```
module Paketverteil-Anlage:
comprises

   module Meldungserzeugung:
   comprises
      block Meldungen-drucken:
      while     Automatik;
      consumes  Meldungen;
      produces  Druckausgabe;
      end
   and
      buffer Druckausgabe:        (* Dieser Puffer wird scheinbar    *)
      interface;                  (* nur beliefert, da der Abnehmer, *)
      capacity 1;                 (* d.h. hier ein Hardware-System,  *)
      of-type Zeile;              (* nicht beschrieben ist. Daher    *)
      end                         (* ist "Druckausgabe" als "inter-  *)
                                  (* face" gekennzeichnet.           *)
   end Meldungserzeugung

and
   procedure Steuerung:
   parallel Meldungen-drucken
   end Steuerung
and
   buffer Meldungen:
   consume restricted-to Meldungserzeugung;
   capacity 10;
   of-type Zeile;
   end

end Paketverteil-Anlage.
```

6.1.7 Die Medien in VS2

Die in 6.1.6.2 genannten Medien werden, soweit sie noch nicht definiert sind, nachfolgend beschrieben. Sie sind lokal zu VS2. Die Objekte in den anderen Moduln ergeben sich analog.

```
module VS2:
comprises
   resource Buchführung-2:
   ¢ sichert die Konsistenz von !Pakete-in-Weiche-2, !Weitergabe-1-2
     und !akt-Paket-2 ¢;
   end
and
   buffer akt-Paket-2:
   ¢ zur Speicherung der Information über Pakete, die in !VS2
     angemeldet sind ¢;
   capacity 1;
   of-type Begleitinformation
   end
and
   variable gleiche-Richtung-2:
   of-type boolean;
   text ¢ true, wenn die Richtung des einlaufenden Pakets für diese
            Weiche mit der des Vorgängers übereinstimmt. ¢
   end
and
   variable Weichenstellung-2:
   ¢ zur Speicherung des Weichenstandes ¢;
   values links, rechts;
   end
and
   type Begleitinformation:
   structure-of Soll-Ziel, Ist-Ziel
   end
and
   type Soll-Ziel: of-type Zielstationen end
and
   type Ist-Ziel:  of-type Zielstationen end
and
   type Zielstationen:
   values ZS1, ZS2, ZS3, ZS4, ZS5, ZS6, ZS7, ZS8
   end
and
   variable Weiche-2-frei:
   of-type boolean;
   text ¢ true, wenn die Weiche laut Buchführung frei ist. ¢;
   end
and
   trigger Pakete-in-Weiche-2:
   ¢ Zähler für die Pakete, die von !VS2-Einlauf registriert,
     von !VS2-Auslauf aber noch nicht abgebucht sind ¢;
   end
end VS2.
```

6.1.8 Eine Vereinfachung der Block-Struktur

Betrachtet man die entstehende Struktur im Hinblick auf den Grenzfall, daß die Weiche zwischen zwei Paketen nur sehr kurz frei wird, so fällt auf, daß die Umschaltung erfolgt, während das Paket schon in die Weiche einläuft. Sie kann also ebensogut in allen Fällen von der Eingangsverwaltung vorgenommen werden, da nach Aufgabenstellung der Rechenprozeß keine nennenswerte Zeit beansprucht. Diese Lösung hat eine erheblich einfachere Struktur, da der Puffer akt-Paket-2 durch eine Variable ersetzt ist und die Auslaufverwaltung nur noch Pakete abbucht. Das macht auch das abstrakte Betriebsmittel "Buchführung" überflüssig.

```
block VS2-Verwaltung:

parallel
   VS2-Einlauf:
   started-by Einlauf-VS2;
   consumes   Weitergabe-1-2;
   writes     akt-Paket-2;                    (* ist jetzt Variable *)
   produces   Pakete-in-Weiche-2;

   sequential
      VS2-Umschalt-Test:
      inhibits Pakete-in-Weiche-2;
      tests    Pakete-in-Weiche-2;
      writes   Weiche-2-frei
      end

   then
      VS2-Umschaltung:
      controlled-by Weiche-2-frei;

      either key true for
         VS2-Weiche-stellen:
         reads  akt-Paket-2;
         writes Weiche-2, Weichenstellung-2
         text Optimierung ¢ durch weitere Verfeinerung kann hier
                            verhindert werden, daß eine Ausgabe
                            erfolgt, wenn die Stellung der Weiche
                            nicht verändert wird.                ¢;
         end VS2-Weiche-stellen

      or key false for
         VS2-Weiche-belegt: (* keine Aktion möglich *) end

      end VS2-Umschaltung
```

```
   then
      VS2-Weitergabe:
      controlled-by Weichenstellung-2;

      either key links for
         VS2-Weitergabe-links:
         reads    akt-Paket-2;
         produces Weitergabe-2-4;
         end VS2-Weitergabe-links

      or key rechts for
         VS2-Weitergabe-rechts:
         reads    akt-Paket-2;
         produces Weitergabe-2-5;
         end VS2-Weitergabe-rechts
      end VS2-Weitergabe

   end VS2-Einlauf


parallel
   VS2-Auslauf:
   started-by Auslauf-VS2;
   consumes   Pakete-in-Weiche-2;
   end VS2-Auslauf

end VS2-Verwaltung.
```

Fügt man die oben skizzierten Teile zusammen und ergänzt das fehlende sinngemäß, so entsteht eine vollständige, weitgehend formale Beschreibung des Systems, die als Spezifikation bezeichnet werden kann. Zweifellos sind bereits Entwurfsentscheidungen getroffen (z.B. hinsichtlich der Strukturierung). Dennoch lag keine vollständige Spezifikation vor; diese konnte erst auf der jetzt erreichten Detaillierungsebene von einigen Unklarheiten befreit werden.
Es bestätigt sich damit, daß die Auffassung, Spezifikation und Entwurf könnten sequentiell durchgeführt werden, praktisch nicht aufrecht erhalten werden kann. Vielmehr handelt es sich um zwei miteinander verzahnte Tätigkeiten.

6.2 Kritik

Eine Kritik an ESPRESO kann sich an den Forderungen aus Kapitel 2 und an Erfahrungen orientieren. Da Erfahrungen eines Einsatzes von ESPRESO in einem echten Software-Projekt noch nicht vorliegen, können hier nur das Beispiel (6.1) und einige ähnliche, nicht wiedergegebene Versuche, ESPRESO anzuwenden, zugrundegelegt werden.

Nachfolgend sind einige Beobachtungen aufgelistet. Die Reihenfolge stellt keine Bewertung dar.

a) Bei vielen Anforderungen, z.B. der Verständlichkeit, ist nur eine subjektive Beurteilung möglich. Der Verfasser selbst kann dazu nur feststellen, daß die Ziele während der gesamten Entwicklung beachtet und mit ESPRESO im wesentlichen erreicht wurden.

b) Die objektivierbaren Anforderungen sind bis auf drei Ausnahmen erfüllt: Es fehlen spezielle Mittel für die Vorbereitung von Tests in der Spezifikationsphase (Testorientierung), und die Verknüpfung verschiedener Versionen ist unbefriedigend. Schließlich fehlen spezielle Möglichkeiten in ESPRESO, die Zuordnung zwischen Anforderung und dem daraus entstehenden Code erkennbar zu machen. Der erste Punkt erfordert nicht unbedingt eine Erweiterung des Systems. Durch Konventionen für Texte (Vorgabe bestimmter Selektoren für die Testplanung) kann eine befriedigende Lösung erreicht werden. Der zweite Mangel ist durch eine prinzipiell einfache Erweiterung des Verwaltungssystems für die ESPRESO-Dateien behebbar. Die größten Schwierigkeiten macht die Verknüpfung zwischen den verschiedenen Ebenen. Hier können nur die relativ primitiven Mittel (Texte mit Verweisen) empfohlen werden.

c) Bei der Bearbeitung des Beispiels und bei allen andern Versuchen, ESPRESO-S anzuwenden, zeigte sich folgender charakteristischer Ablauf: Zunächst wird das Problem naiv in ESPRESO-S gefaßt. Dabei werden Unklarheiten der Aufgabenstellung erkennbar, die durch Nachfragen oder, wo dies nicht möglich ist, durch eine Entscheidung des Spezifizierenden beseitigt werden. So entsteht langsam ein vollständiges Bild des Problems. Dies führt zu einem besseren Verständnis der Aufgabe und damit oft zu einer Neufassung der ESPRESO-Spezifikation, die wesentlich transparenter wird.
Es ist denkbar, daß sich dieser Prozeß mehrfach wiederholt, wenn das Problem sehr umfangreich ist; bei allen bisherigen Versuchen hat eine Überarbeitung ausgereicht.

d) Die in 3.1 geforderte Unterscheidung zwischen Anforderungen und freien Entscheidungen wird in ESPRESO nicht durch spezielle Sprachmittel unterstützt oder gar erzwungen. Die einzige Möglichkeit ist die Speicherung aller Anforderungen in Pseudo-Objekten und Verbindung dieser Objekte mit den anderen durch Verweise. Diese Lösung ist unbefriedigend, die Sprache sollte in diesem Punkt erweitert werden.

e) Völlig ungelöst bleibt in ESPRESO-S das Problem der Indizierung und der Selektion. So enthält die Sprache keine Möglichkeiten, Elemente aus Feldern zu bezeichnen oder gleichartige Betriebsmittel zu unterscheiden. Auch gibt es keine Zusammenfassung sehr ähnlicher Objekte, z.B. der Verteilstationen in der Paketverteilungsanlage. In dieser Richtung wäre eine Erweiterung der Sprache zu wünschen. Das wesentliche Problem dabei ist, daß dann ein Merkmal der Sprache, der Verzicht auf jede Darstellung der Arithmetik, kaum aufrechterhalten werden kann. Die Sprache würde dadurch wesentlich an Klarheit verlieren.

f) Eine Vereinfachung der Darstellung oben könnte erreicht werden, wenn es für gewisse, häufig wiederkehrende Situationen Sonderregelungen gäbe, die die Darstellung ohne Nachteile straffen würden. Z.B. gibt es einige leere Blöcke, die nur zur Aufzählung aller Alternativen spezifiziert werden. Für solche leeren BLöcke könnte in der Syntax ein spezielles Objekt vorgesehen werden, so daß nicht jedesmal ein neues Objekt definiert werden muß.

g) In 5.10.2 wurde begründet, warum ESPRESO nicht den Begriff der Funktion, sondern den der Prozedur in den Mittelpunkt stellt. An einigen Stellen, z.B. bei Selektoren, ist das Fehlen des Funktionsbegriffs jedoch, wie Punkt f zeigt, ungünstig. Es bleibt zu untersuchen, wie dieses Problem gelöst werden kann; eine Möglichkeit wäre, auf die Verzweigungen in ESPRESO zu verzichten, eine andere, den Funktionsbegriff hinzuzunehmen.

h) Eines der Ziele, auch die Umgebung des geplanten Systems in ESPRESO spezifizierbar zu machen, wurde nicht erreicht. Technischer Prozeß und Bediener können nur durch Texte beschrieben werden. Eine Lösung dieses Problems würde auch eine interessante Anknüpfung an das von Baumann (1978) präsentierte Konzept ergeben, das Prozeßrechnerprogramm zunächst mit einem von einem zweiten Rechner (oder allgemeiner: Rechner-System) simulierten Prozeß zu testen, bevor es am realen Prozeß eingesetzt wird.

i) Das in ESPRESO angewandte Konzept, die Spezifikation mit der Strukturierung des Problems zu verbinden, versagt, wenn die Voraussetzungen anders sind als in 2.2.2 angenommen, d.h. wenn sehr viele Anforderungen gesammelt werden können und müssen, bevor irgendwelche Überlegungen zur Struktur möglich sind. Ein Beispiel ist die Spezifikation eines Telefonsystems (Taylor, 1980). In solchen Fällen kann nur ein dem speziellen Problem angepaßtes Spezifikationssystem echte Hilfe leisten.

j) Durch die Bindung der Medien an die - statischen - Moduln gibt es in ESPRESO-S keine Möglichkeit, lokale Variablen von rekursiven Prozeduren zu beschreiben. Dazu sind verschiedene Erweiterungen denkbar, z.B. die wahlweise Bindung der Variablen auch an Prozeduren.

k) Im Beispiel (6.1) hat sich ein - in andern Fällen weitaus schwerer wiegendes - Problem gezeigt, die Verwaltung von Bildinformation. Die Verwendung von Zeichnungen verschiedenster Art ist gerade bei den typischen Einsatzgebieten des Prozeßrechners nicht verzichtbar. Sie gefährdet aber das Prinzip, alle Information mit ESPRESO-W zu verwalten. Es muß geprüft werden, auf welche Weise Information, deren Eingabe und/oder Speicherung durch den Rechner Schwierigkeiten macht, mit dem Inhalt der ESPRESO-Datei verknüpft werden kann.

7. (Anhang) Erweiterte Attribut-Grammatik für ESPRESO-S

Inhalt

7.1 Vorbemerkungen

7.1.1 Notation

7.1.1.1 Symbole und ihre Darstellungen:

Klammern für syntaktische Variablen mit ihren Attributen	< ... >
Klammern für Tupel (nur innerhalb spitzer Klammern)	(...)
Produktionszeichen	::=
Trennzeichen für Alternativen	]
Schlußzeichen für Produktionen	.
senkrechter Pfeil von oben (vorgegebenes Attribut)	↓
senkrechter Pfeil von unten (erzeugtes Attribut)	↑
geschweifte Klammern (für Mengen)	{ ... }
eckige Klammern (für Zeichenreihen)	[...]
Mengendifferenz, Konkatenation	\, +
Enthaltensein und Nichtenthaltensein	∈, ∉
Leere Menge	∅
Vereinigung und Schnitt von Mengen	∪,∩
Überlagerung (siehe 7.1.4.1)	ϕ
Leere Zeichenreihe in Produktionen	ε

Zur leichteren Unterscheidung sind in der Grammatik Attribut-Variablen mit großen ("VARIABLE"), Attribut-Konstanten mit kleinen ("variable") und die syntaktischen Variablen mit großen und kleinen Buchstaben geschrieben ("Variablen-Verwendung").

7.1.1.2 Darstellungsschema

Alle (Meta-) Produktionen haben folgende Form:

```
< ... >
 ::= < ... >  ...  < ... >
   ] < ... >  ...  < ... >
   .
   .
   ] < ... >  ...  < ... >.
```

Die spitzen Klammern enthalten syntaktische Variablen mit ihren Attributen. Syntaktische Variablen, denen unmittelbar terminale Zeichen oder Zeichenreihen zugeordnet sind und die daher keine Attribute haben, z.B. "Semikolon" oder "end-Symbol" (siehe 7.7.2, 7.8), stehen o h n e spitze Klammern. Verweise auf andere Abschnitte dieses Anhangs und auf Attribute und Relationen in den Tabellen 4.2, 4.3 sind in Schrägstriche geschlossen, z.B. /5.1/, /a2/, /r21/. Die Kapitelnummer (7.) fehlt dann. Kommentare stehen in Klammern mit Stern, z.B. (* Kommentar *).

7.1.1.3 Attribut-Variablen und ihre Wertebereiche

Die Attribut-Variablen geben bereits durch ihren Namen Auskunft über ihre Bedeutung mit Ausnahme des ständig vorkommenden Kontexts K (siehe 7.1.3). Die nachfolgende Liste ist formal unnötig, sie dient nur dem besseren Verständnis.

Alle Attribut-Variablen können durch eine angehängte Ziffer speziell unterschieden werden. Am Wertebereich ändert diese Ziffer nichts. Die Ziffer "0" deutet dabei auf eine hierarchische Beziehung hin, sie markiert den "Vater".

K		(Links-) Kontext (Umgebung)
NAME, VERWEIS		Name (für beliebiges Objekt)
ANFANG		Anfang eines Namens oder Name
MODUL, PROZEDUR, BLOCK, VARIABLE, PUFFER, TRIGGER, PARAMETER, BETRMTTL, TYP, FRIST, KONST, TEXT-OBJEKT		NAME (für ein Objekt der betreffenden Art oder Arten)
PUGGER	$\in$	{PUFFER,TRIGGER}
TYABLE	$\in$	{TYP,VARIABLE}
PROCK	$\in$	{PROZEDUR,BLOCK}
MEDIUM	$\in$	{VARIABLE,PUFFER, TRIGGER,BETRMTTL}
ANZAHL		Ziffernfolge oder VARIABLE oder KONSTANTE
SELEKTOR		NAME (für Texte)
TEXT		Zeichenreihe ohne Cent
ZEICHEN		Einzelzeichen
ZUGRIFF	$\in$	{lesen, schreiben, ändern, initialisieren, liefern, holen, testen, starten, beenden, maskieren, belegen}
RESTRIKT	$\in$	{lese-r, schreib-r, liefer-r, hol-r, start-r, beende-r, maskier-r, verwend-r}
ART	$\in$	{modul, prozedur, block, eingabe-parameter, ausgabe-parameter, trans-parameter, variable, puffer, trigger, betriebsmittel, typ, frist, konstante, text-objekt}
ARTEN		{ART1, ... , ARTn}, n > 0.

7.1.2 Interpretation

In der Literatur (Watt, Madsen, 1977, S.9 ff.) ist angegeben, wie die erweiterte Attribut-Grammatik zu interpretieren ist. Nachfolgend wird nur eine sehr knappe Zusammenfassung gegeben, die auf diese spezielle Grammatik zugeschnitten ist.

7.1.2.1 Attribute zur Kontext-Beschreibung

Die Grammatik besteht aus Produktionen und - ganz überwiegend - Metaproduktionen, aus denen durch konsistente Ersetzung der Attribut-Variablen Produktionen entstehen. Die Struktur ist wie in der BNF, wobei die syntaktischen Variablen in den spitzen Klammern attributiert sein können. Eine Klammer enthält links die syntaktische Variable, dann eine (eventuell leere) Folge von Attribut-Variablen oder gelegentlich auch Attribut-Konstanten, die jeweils links durch einen senkrechten Pfeil markiert sind. Für die erweiterten Attribut-Grammatiken ist es kennzeichnend, daß an Stelle der Attribute auch Ausdrücke aus Attributen und Operatoren (siehe 7.1.4) stehen können.

Die Richtung des Pfeils gibt an, ob der aktuelle Wert des Attributs vorgegeben ist (vorgegebenes Attribut: Pfeil kommt von oben) oder durch die weitere Zerlegung erzeugt wird (erzeugtes Attribut: Pfeil kommt von unten). Diese Richtungsangabe dient nur der Übersicht, formal ist sie redundant.

Beispiele:

a) < Name ↑ NAME > (vgl. 7.7.1)

Die syntaktische Variable "Name" hat das Attribut NAME. Wird aus "Name" z.B. die Zeichenreihe "xyz" abgeleitet, so hat NAME den Wert xyz. NAME ist also ein erzeugtes Attribut.

b) < test zyklusfrei ↓ K ↓ FRIST ↓ FRIST1 > (vgl. 7.4.8)

K ("Kontext", siehe 7.1.3) und FRIST, FRIST1 (für zwei Namen, die jeweils für eine Frist stehen) sind die vorgegebenen Attribute dieses Ausdrucks, erzeugte Attribute kommen nicht vor. Aus der einzigen Produktion, die auf diesen Ausdruck anwendbar ist (siehe 7.4.8), geht hervor, daß nur der leere String produziert wird, falls die aktuellen Werte der syntaktischen Variablen nicht in eine Sackgasse führen, weil die FRIST1 zyklisch definiert ist, also als Bestandteil ihrer selbst. Dies wäre ein Fehler.

Es handelt sich also um einen Test. Dies ist durch den Beginn mit "test" gekennzeichnet. Solche Variablen haben meist nur vorgegebene Attribute, in einigen Fällen wird der Kontext verändert.

Die Nichtterminale < test wahr >, < wobei wahr > und < fall wahr > produzieren ebenfalls nur die leere Zeichenreihe. In der Grammatik steht anstelle von "wahr" stets ein logischer Ausdruck. Ergibt dieser nicht "wahr", sondern "falsch", so liegt eine Sackgasse vor. Bei "test" kennzeichnet dies einen semantischen Fehler der Eingabe, bei "fall" muß die nächste Alternative versucht werden. Der Ausdruck hinter "wobei" hat den Charakter einer Wertzuweisung und ergibt wahr, falls der Wert definiert ist. Andernfalls handelt es sich um einen Fehler. Beispiele aus 7.3 und 7.4 sind

< test NAME1 = NAME2 >,

< fall ART = modul >,

< wobei K2 = K1 >.

c) < Block-Sektion ↓ K1 ↑ NAME ↑ K2 > (vgl. 7.2.1)

Durch die Block-Sektion wird Kontext K1 erweitert zu Kontext K2. Außerdem wird für den Gebrauch in übergeordneten Produktionen der Name des Blocks mittels der Attribut-Variablen NAME nach oben gereicht.

7.1.2.2 Attribute zur Bildung von Schemata

Um die Länge der Grammatik zu vermindern und einheitliche Strukturen erkennbar zu machen, wurde in einigen Fällen vom Prinzip abgewichen, die attributierte Grammatik aus der kontextfreien einfach durch Zufügen der Attribute zu erzeugen. Stattdessen sind verschiedene syntaktische Variablen in Schemata zusammengefaßt.

Beispiel: Alle Objekte werden durch eine Sektion beschrieben; die Sektionen haben alle die gleiche Grundstruktur. Daher wird eine einzige syntaktische Variable mit einem Attribut für den Typ verwendet, z.B. < Sektion ↑ ART ↓ K1 ↑ NAME ↑ K2 >. Eine Puffer-Sektion ist also < Sektion ↑ puffer ↓ K1 ↑ NAME ↑ K2 >.

Ähnlich wie mit ART bilden syntaktische Variablen mit ARTEN, ZUGRIFF und RESTRIKT Schemata, siehe z.B. 4.2.3, 5.1 und 5.2 in diesem Anhang.

7.1.3 Der Kontext als Attribut

Fast alle syntaktischen Variablen haben K, den (Links-) Kontext, als Attribut, und zwar meist zweifach, als vorgegebenes und als durch die Ableitung daraus erzeugtes. In andern Grammatiken wird K meist als "Environment" oder "Umgebung" bezeichnet.

Die Grammatik ist so formuliert, daß K als formaler Repräsentant der Information in der ESPRESO-Datei betrachtet werden kann. Entsprechend den Definitionen in 3.2.1.1 definiert damit die Grammatik, welche Information einer ESPRESO-Spezifikation zugeordnet ist.

K enthält unterschiedliche Arten der Information:

- die Zuordnung der Objekte, d.h. ihrer Namen, zu Arten,
- die Texte und ihre Verbindungen mit Objekten und Textselektoren,
- die Verknüpfungen der Objekte untereinander,
- Eigenschaften der Objekte (Attribute).

All diese Information wird dargestellt durch n-Tupel ($n \in \{2,3,4\}$), hier Verknüpfungen genannt.
Das erste Element darin ist stets das Merkmal, das die Bedeutung des Tupels angibt. Die Menge der Merkmale ist durch die Grammatik vorgegeben. Sie ordnen die Verknüpfungen in Klassen, die Relationen. Dem Merkmal folgen ein bis drei Elemente, von denen mindestens eines signifikant ist; die übrigen sind, falls vorhanden, abhängig und in der Grammatik durch einen Pfeil ("→") von den signifikanten getrennt. Tupel ohne abhängige Elemente werden freie Tupel genannt, die anderen Paare.

Attribute sind spezielle Paare, in denen das einzige abhängige Element kein Objektname, sondern einer vorgegebenen Menge entnommen ist.

Beispiele:

> freies Tupel: "(verweis,Name1,Selektor1,Name2)"
> bedeutet: In einem Text des Objekts "Name1", der den Textselektor "Selektor1" hat, ist ein Verweis auf das Objekt "Name2" enthalten. Alle Tupel, die mit "verweis" beginnen, haben die gleiche Struktur und Interpretation.
>
> Attribut: "(art,Name1)→variable"
> sagt aus, daß das Objekt "Name1" die einer vorgegebenen Menge entnommene Art "variable" hat.

Die signifikanten Elemente kennzeichnen eine Information eindeutig. Stimmen also von zwei Paaren die linken Seiten überein, so dürfen sich die rechten nicht widersprechen. In vielen Fällen stehen aber die abhängigen Elemente nicht oder nicht genau fest, z.B. wenn die Eingabe lautet:

'block' Name1: 'inhibits' Pugger1 'end'

"Pugger1" kann nach dem Kontext entweder ein Puffer oder ein Trigger sein. Es gilt also:

(art,Pugger1)→x, x ∈ {puffer,trigger},

was in dieser Grammatik verkürzt wird zu:

(art,Pugger1)→{puffer,trigger}

Statt eines abhängigen Elementes kann also im Kontext auch eine Menge alternativ in Frage kommender Elemente stehen. Zugunsten der Übersichtlichkeit sind in der Grammatik Mengenklammern meist weggelassen, wo die Menge nur e i n Element hat.

Offenbar folgt aus s→a1, a1 ∈ M1 und s→a2, a2 ∈ M2,
daß s→a, a ∈ M = M1 ∩ M2, wobei M nicht leer sein darf.

Paare sind also im Kontext notwendig für die Syntaxprüfung; die freien Tupel sind dagegen für die Syntax irrelevant, sie dienen nur zur Beschreibung der in die ESPRESO-Datei zu speichernden Information.

Die Elemente der Tupel sind Zeichenreihen, und zwar

- Namen von Objekten,
- Textselektoren,
- Texte,

] vom Anwender gewählt

- Arten und Relationsnamen,
- andere Merkmale zur Kennzeichnung des Informationstyps und für Objekteigenschaften,
- Platzhalter für fehlende Information.

] durch die Grammatik vorgegeben

7.1.4 Funktionen für die Grammatik

In der Grammatik werden neben den üblichen Mengenoperationen, also Vereinigung ("∪"), Schnitt ("∩"), Differenz ("\") und für geordnete Mengen (Zeichenreihen) die Konkatenation ("+"), einige spezielle Funktionen verwendet, die nachfolgend definiert werden.

Dabei werden folgende Bezeichnungen verwendet:

S(K) = { s | existiert a : (s→a) ∈ K } (signif. Teile der Paare),

A(K,s), der abhängige Teil, ist nicht definiert, falls s ∉ S(K).
Sonst existiert ein a mit (s→a) ∈ K, und es gilt: A(K,s) = a.
A(K,s,i) ist die i-te Komponente von A(K,s), falls A(K,s) definiert ist und mindestens i Komponenten hat; sonst undefiniert.

7.1.4.1
ϕ, bezeichnet als Überlagerung, ist nur partiell definiert.
Sei f ein freies Tupel, p=s→a ein Paar. Dann gilt:

$$K \phi f = K \cup \{f\},$$

$$K \phi p = \begin{cases} K \cup \{p\}, & \text{falls } s \notin S(K), \\ (K \setminus \{s \to a1\}) \cup \{s \to a2\}, \text{ mit } a1 = A(K,s) \text{ und } a2 = a \cap a1, & \text{falls } s \in S(K) \text{ und } a2 \neq \emptyset, \\ \text{undefiniert sonst.} & \end{cases}$$

ϕ dient praktisch zum Hinzufügen im Kontext, entsprechend der Verarbeitung des Eingelesenen. Dadurch, daß ϕ nur partiell definiert ist, ist ausgedrückt, daß es Eingaben gibt, die in K nicht widerspruchsfrei gespeichert werden können. Sei z.B. die Eingabe

Bei beliebigem Linkskontext K entsteht ein Widerspruch, der dadurch reflektiert wird, daß der entstehende Kontext nicht definiert ist:

$$K \phi (art,abc) \to block \phi (art,abc) \to module$$

("ϕ" ist nicht assoziativ, ein Ausdruck mit mehreren "ϕ" wird von links nach rechts ausgewertet.)

7.1.4.2
"undef" ist das neutrale Element bei der Bildung der Schnittmenge:

$$a \cap undef = a \qquad \text{für alle Mengen a.}$$

"undef" vertritt in Paaren abhängige Elemente, die noch völlig unbestimmt sind, z.B. weil die betreffende Angabe fehlt.

7.1.4.3
"alle(K)" wird an die Stelle irrelevanter Elemente der Tupel gesetzt:

$$(E1, \ldots, Ei, alle(K), Ej, \ldots, En) = \{ t \mid \text{existiert } x: t = (E1, \ldots, Ei, x, Ej, \ldots, En) \in K \cup S(K) \}$$

7.1.4.4
Die Relationen "gleich" ("="), "ungleich" ("≠"), "Element von" ("∈"), "nicht Element von" ("∉") haben die übliche Bedeutung. Die mit ihnen gebildeten Ausdrücke haben entweder den Wert "wahr" oder den Wert "falsch".

7.1.5 Eine vollständige Produktion als Beispiel

Der Ausschnitt einer Spezifikation könnte wie folgt aussehen:

```
(1)  'block' ABC:
      ...
(2)  'sequential' ALPHA: ... 'end' ALPHA
(3)  'then'       BETA:  ... 'end' BETA
(4)  'then'       GAMMA: ... 'end' GAMMA;
      ...
(5)  'end' ABC.
```

Die Zeilen 3 und 4 werden nach folgender Produktion aus 7.4.2.4.1 analysiert:

```
< nachfolgende Blöcke        ↓ K1 ↓ BLOCK1 ↓ PROCK          ↑ K2 >
 ::= ε < wobei K2 = K1 ϕ (sequent,BLOCK1)→(PROCK,undef) >          /r2/
   ] < nachfolgender Block  ↓ K1 ↓ BLOCK1 ↓ PROCK ↑ BLOCK ↑ K  >
     < nachfolgende Blöcke ↓ K ϕ (sequent,BLOCK1)→(PROCK,BLOCK)
                              ↓ BLOCK  ↓ PROCK          ↑ K2 >.   /r2/
```

Diese relativ komplizierte Produktion ist wie folgt zu interpretieren:

"nachfolgende Blöcke" ist leer oder ein nachfolgender Block, gefolgt von weiteren nachfolgenden Blöcken. Dies ist die kontextfreie Struktur. Im Beispiel werden nach der zweiten Alternative zunächst Zeilen 3 und 4 erzeugt, dann nur 4. Schließlich wird die erste Alternative angewandt.

Als Attribute treten in der Produktion auf:

- PROCK — für die übergeordnete Prozedur oder den übergeordneten Block (im Beispiel ABC),
- BLOCK1 — für einen zuvor schon definierten Block (im Beispiel erst ALPHA, dann BETA),
- BLOCK — für den ersten auf BLOCK1 folgenden Block, (im Beispiel erst BETA, dann GAMMA),
- K1, K und K2 für verschiedene Stadien des Kontexts.

K1, PROCK und BLOCK1 sind durch den (Links-) Kontext vorgegeben und treten daher als vorgegebene Attribute auf. BLOCK ist der Name des aus "nachfolgender Block" erzeugten Objekts, also dort ein erzeugtes Attribut. Seinem Nachfolger dient BLOCK seinerseits wieder als vorgegebenes Attribut.

"wobei K2 = ..." definiert den Wert von K2 und produziert keine Terminalzeichen (siehe 7.5.3).

Die Überlagerung fügt dem Kontext, wie er an "nachfolgende Blöcke" weitergereicht wird, eine Verknüpfung der Relation "sequentiell in" zu. Sollte sich im Zuge der Verarbeitung des Blocks ABC zeigen, daß die signifikanten Elemente (sequent,BETA) bereits mit anderen abhängigen Elementen als (ABC,GAMMA) oder (ABC,undef) im Kontext stehen, so ist die Spezifikation fehlerhaft.

```
(*************************************************************************)
(***   2. in Kap.7 (Anh.)  Globale Struktur der ESPRESO-Eingabe   *****)

(* Wurzel der Grammatik ist die < ESPRESO-Eingabe >. Sie umfaßt nicht
   nur die aktuelle Eingabe, sondern durch den Linkskontext alles, was
   schon in der Datei angesammelt ist.                                  *)


(***   2.1 in Kap.7 (Anh.)  Abschnitte und Sektionen   ***************)

< ESPRESO-Eingabe                              ↑ K  >
 ::= ε < wobei K = standard-kontext >
   ] < ESPRESO-Eingabe                         ↑ K1 >
     < ESPRESO-Ausdruck       ↓ K1            ↑ K  >.

(* standard-kontext enthält die vordefinierten Objekte und
   Verknüpfungen, siehe 7.9.

   Der Umfang des durch K bezeichneten Kontext ist beschränkt durch
   die Implementierung, dadurch auch die Länge der ESPRESO-Eingabe.   *)

< ESPRESO-Ausdruck             ↓ K1            ↑ K2 >
 ::= < Abschnitt               ↓ K1            ↑ K2 > Punkt
   ] < Abschnitt               ↓ K1            ↑ K  > Semikolon
     < ESPRESO-Ausdruck        ↓ K             ↑ K2 >.

< Abschnitt                    ↓ K1            ↑ K2 >
 ::= < Sektion ↑ ART           ↓ K1 ↑ NAME ↑ K2 >.                    /2.2/

< Block-Sektion                ↓ K1                 ↑ NAME ↑ K2 >
 ::= < eingebettete Sektion    ↓ K1 ↓ {block}       ↑ NAME ↑ K2 >.

< eingebettete Sektion         ↓ K1 ↓ ARTEN ↑ NAME ↑ K2 >
 ::= < Sektion ↑ ART           ↓ K1           ↑ NAME ↑ K2 >           /2.2/
     < test ART ∈ ARTEN >.


(***   2.2 in Kap.7 (Anh.)  Aufbau der Sektionen   *******************)

< Sektion              ↑ ART ↓ K1 ↑ NAME ↑ K2 >
 ::= < Sektionskopf    ↑ ART ↓ K1 ↑ NAME ↑ K2 >                        /3./
   ] < Sektionskopf    ↑ ART ↓ K1 ↑ NAME ↑ K  > Doppelpunkt            /3./
     < Sektionsrumpf   ↓ ART ↓ K  ↓ NAME ↑ K2 >                        /4./
     < Sektionsschwanz            ↓ NAME      >.                       /3./
```

```
(*************************************************************************)
(***   3. in Kap.7 (Anh.)  Der Sektionskopf und -schwanz   ***********)

< Sektionskopf      ↑ ART ↓ K ↑ NAME ↑ K ϕ (art,NAME)→ART >          /a1/
 ::= < Art-Kennung ↑ ART                                  >
     < Name                    ↑ NAME                     >        /7.1/
     < test (vordefiniert,NAME) ∉ K >.                     (* vgl. 7.9 *)


< Art-Kennung ↑ ART >
::= informal-Symbol  < wobei ART = text-objekt       >
  ] module-Symbol    < wobei ART = modul             >
  ] procedure-Symbol < wobei ART = prozedur          >
  ] block-Symbol     < wobei ART = block             >
  ] inpar-Symbol     < wobei ART = eingabe-parameter >
  ] outpar-Symbol    < wobei ART = ausgabe-parameter >
  ] transpar-Symbol  < wobei ART = trans-parameter   >
  ] variable-Symbol  < wobei ART = variable          >
  ] buffer-Symbol    < wobei ART = puffer            >
  ] trigger-Symbol   < wobei ART = trigger           >
  ] resource-Symbol  < wobei ART = betriebsmittel    >
  ] type-Symbol      < wobei ART = typ               >
  ] interval-Symbol  < wobei ART = frist             >
  ] constant-Symbol  < wobei ART = konstante         >.


< Sektionsschwanz  ↓ NAME1 >
 ::= end-Symbol
     < Name        ↑ NAME2 >                                    /7.1/
     < test  NAME1 = NAME2 >.

(* Zum Namen im Sektionsschwanz vgl. 7.10.                          *)


(*************************************************************************)
(***   4. in Kap.7 (Anh.)  Der Sektionsrumpf (Angaben zum Objekt)   ***)

< Sektionsrumpf          ↓ ART ↓ K1 ↓ NAME ↑ K2 >
 ::= < Angabe            ↓ ART ↓ K1 ↓ NAME ↑ K2 >
   ] < Angabe            ↓ ART ↓ K1 ↓ NAME ↑ K  > Semikolon
     < Sektionsrumpf     ↓ ART ↓ K  ↓ NAME ↑ K2 >.

< Angabe                 ↓ ART ↓ K1 ↓ NAME ↑ K2 >
 ::= < fall ART = modul >
     < Modul-Angabe          ↓ K1 ↓ NAME ↑ K2 >                  /4.1/
   ] < fall ART = prozedur >
     < Prozedur-Angabe       ↓ K1 ↓ NAME ↑ K2 >                /4.2.1/
   ] < fall ART = block >
     < Block-Angabe          ↓ K1 ↓ NAME ↑ K2 >                /4.2.2/
   ] < fall ART 6{eingabe-parameter,
                  ausgabe-parameter,
                  trans-parameter}  >
     < Parameter-Angabe      ↓ K1 ↓ NAME ↑ K2 >                  /4.7/

(* Die Produktion ist auf der nächsten Seite fortgesetzt.           *)
```

```
   ] < fall ART = variable >
     < Variablen-Angabe       ↓ K1 ↓ NAME ↑ K2 >                    /4.3/
   ] < fall ART = puffer >
     < Puffer-Angabe          ↓ K1 ↓ NAME ↑ K2 >                  /4.4.1/
   ] < fall ART = trigger >
     < Trigger-Angabe         ↓ K1 ↓ NAME ↑ K2 >                  /4.4.2/
   ] < fall ART = betriebsmittel >
     < Betriebsmittel-Angabe  ↓ K1 ↓ NAME ↑ K2 >                    /4.5/
   ] < fall ART = typ >
     < Typ-Angabe             ↓ K1 ↓ NAME ↑ K2 >                    /4.6/
   ] < fall ART = frist >
     < Frist-Angabe           ↓ K1 ↓ NAME ↑ K2 >                    /4.8/
   ] < fall ART = konstante >
     < Konstanten-Angabe      ↓ K1 ↓ NAME ↑ K2 >                    /4.9/
   ] < fall ART 6 {prozedur,block,variable,
                   puffer,trigger,betriebsmittel} >
     < Positionsattribut      ↓ K1 ↓ NAME ↑ K2 >
   ] < Textangabe             ↓ K1 ↓ NAME ↑ K2 >                    /6.1/
   ] < Markierung             ↓ K1 ↓ NAME ↑ K2 >                    /6.2/
   ] ε < wobei K2 = K1 >.

< Positionsattribut           ↓ K1 ↓ NAME ↑ K2 >
 ::= interface-Symbol
     < wobei K2 = K1 ϕ (position,NAME)→schnittstelle >              /a6/
   ] external-Symbol
     < wobei K2 = K1 ϕ (position,NAME)→außen        >.             /a6/


(***   4.1 in Kap.7 (Anh.)  Modul-Angaben   ***************************)

< Modul-Angabe              ↓ K1 ↓ MODUL        ↑ K2 >
 ::= < Obj-Def-in-Modul     ↓ K1 ↓ MODUL        ↑ K2 >.

< Obj-Def-in-Modul          ↓ K1 ↓ MODUL
                              ↑ K2 ϕ (in-modul,NAME)→MODUL >        /r1/
 ::= comprises-Symbol
     < lokales Objekt       ↓ K1 ↓ MODUL ↑ NAME ↑ K2 >
   ] < Obj-Def-in-Modul     ↓ K1 ↓ MODUL        ↑ K  >
     and-Symbol
     < lokales Objekt       ↓ K  ↓ MODUL ↑ NAME ↑ K2 >.

< lokales Objekt            ↓ K1 ↓ MODUL       ↑ NAME ↑ K2 >
 ::= < eingebettete Sektion ↓ K1 ↓ {modul}     ↑ NAME ↑ K2 >        /2.1/
     < test Modul-Hierarchie ↓ K1 ↓ MODUL      ↓ NAME      >
   ] < eingebettete Sektion ↓ K1 ↓ {prozedur,block, variable,
                                    puffer,trigger,betriebsmittel,
                                    typ,frist,konstante}
                                              ↑ NAME ↑ K2 >.       /2.1/

< test Modul-Hierarchie     ↓ K ↓ MODUL  ↓ NAME >
 ::= < fall (in-modul,MODUL) ∉ S(K) >
     < test NAME ≠ MODUL >
   ] < test Modul-Hierarchie ↓ K ↓ A(K,(in-modul,MODUL)) ↓ NAME >
     < test NAME ≠ MODUL >.
```

```
(***   4.2 in Kap.7 (Anh.)  Prozedur- und Block-Angaben   ************)

(***   4.2.1 in Kap.7 (Anh.)  Prozedur-Angaben   **********************)

< Prozedur-Angabe                  ↓ K1 ↓ PROZEDUR ↑ K2 >
 ::= < Block-Angabe                ↓ K1 ↓ PROZEDUR ↑ K2 >          /4.2.2/
   ] < Parameter-Zuordnung         ↓ K1 ↓ PROZEDUR ↑ K2 >
   ] < Liste-zugr-ber-Moduln       ↓ K1 ↓ PROZEDUR ↑ K2 >           /5.2/
   ] < Prioritätszuordnung         ↓ K1 ↓ PROZEDUR ↑ K2 >
   ] < Verwendbarkeit              ↓ K1 ↓ PROZEDUR ↑ K2 >.

< Parameter-Zuordnung        ↓ K1 ↓ PROZEDUR
          ↑ K2 ϕ (parameter,PARAMETER)→(PROZEDUR,TEXT-OBJEKT) >     /r7/
 ::= parameters-Symbol
     < Parameter-Definition ↓ K1 ↑ PARAMETER ↑ TEXT-OBJEKT ↑ K2 >
   ] < Parameter-Zuordnung  ↓ K1 ↓ PROZEDUR               ↑ K  > Komma
     < Parameter-Definition ↓ K  ↑ PARAMETER ↑ TEXT-OBJEKT ↑ K2 >.

< Parameter-Definition       ↓ K1 ↑ PARAMETER ↑ TEXT-OBJEKT ↑ K2 >
 ::= < eingebettete Sektion  ↓ K1 ↓ {eingabe-parameter,
                                     ausgabe-parameter,
                                     trans-parameter}
                             ↑ PARAMETER            ↑ K  >    /2.1/
     < Zusicherungsteil      ↓ K  ↓ [*]+PARAMETER
                                    ↑ TEXT-OBJEKT ↑ K2 >. /4.2.3/

< Prioritätszuordnung        ↓ K1 ↓ PROZEDUR
                      ↑ K2 ϕ (priorität,PROZEDUR)→ANZAHL >      /r24/
 ::= priority-Symbol
     < Zahlenangabe          ↓ K1 ↑ ANZAHL ↑ K2 >.                /4.9/

< Verwendbarkeit  ↓ K ↓ PROZEDUR ↑ K ϕ (reentrant,PROZEDUR) >      /a7/
 ::= reentrant-Symbol.


(***   4.2.2 in Kap.7 (Anh.)  Block-Angaben   ********************)

< Block-Angabe                   ↓ K1 ↓ PROCK ↑ K2 >
 ::= < Wiederhol-Bedingung       ↓ K1 ↓ PROCK ↑ K2 >
   ] < Verzögerung               ↓ K1 ↓ PROCK ↑ K2 >          /4.4.2/
   ] < Aktion                    ↓ K1 ↓ PROCK ↑ K2 >          /4.2.3/
   ] < Prozedur-Block-Zerlegung  ↓ K1 ↓ PROCK ↑ K2 >          /4.2.4/
   ] < Prozedur-Aufruf-Angabe    ↓ K1 ↓ PROCK ↑ K2 >
   ] < Frist-Definition          ↓ K1 ↓ PROCK ↑ K2 >.           /4.8/
   ] < Frist-Definition          ↓ K1 ↓ PROCK ↑ K2 >.           /4.8/

< Wiederhol-Bedingung            ↓ K1 ↓ PROCK
                          ↑ K2 ϕ (while,PROCK)→VARIABLE >      /r11/
 ::= while-Symbol
     < Nennung ↓ {variable}      ↓ K1 ↑ VARIABLE ↑ K2 >.        /5.1/
```

```
< Prozedur-Aufruf-Angabe          ↓ K1 ↓ PROCK
                                    ↑ K2 ϕ (ablauf,PROCK)→aufruf >      /a2/
 ::= < aufgerufene Prozedur       ↓ K1 ↓ PROCK ↑ K  >
   ] < Parameter-Besetzung        ↓ K  ↓ PROCK ↑ K2 >.

< aufgerufene Prozedur          ↓ K1 ↓ PROCK
                                 ↑ K2 ϕ (aufruf,PROCK)→PROZEDUR >       /r6/
 ::= calls-Symbol
     < Nennung ↓ {prozedur} ↓ K1 ↑ PROZEDUR ↑ K2 >.                     /5.1/

< Parameter-Besetzung                    ↓ K1 ↓ PROCK     ↑ K4 >
 ::= sets-Symbol
     < Nennung ↓ {eingabe-parameter} ↓ K1 ↑ PARAMETER ↑ K2 >            /5.1/
     to-Symbol
     < Nennung ↓ {puffer,variable,konstante}
                                     ↓ K2 ↑ NAME      ↑ K3 >           /5.1/
     < wobei K4 = K3 ϕ (inpar,PROCK,PARAMETER)→NAME >                    /r8/
   ] gets-Symbol
     < Nennung ↓ {ausgabe-parameter} ↓ K1 ↑ PARAMETER ↑ K2 >            /5.1/
     in-Symbol
     < Nennung ↓ {puffer,variable}   ↓ K2 ↑ NAME      ↑ K3 >           /5.1/
     < wobei K4 = K3 ϕ (outpar,PROCK,PARAMETER)→NAME >                   /r9/
   ] associates-Symbol
     < Nennung ↓ {trans-parameter}   ↓ K1 ↑ PARAMETER ↑ K2 >            /5.1/
     with-Symbol
     < Nennung ↓ {variable}          ↓ K2 ↑ VARIABLE  ↑ K3 >           /5.1/
     < wobei K4 = K3 ϕ (transpar,PROCK,PARAMETER)→VARIABLE >.          /r10/


(***   4.2.3 in Kap.7 (Anh.)  Aktionen   ********************************)

< Aktion                              ↓ K1 ↓ PROCK ↑ K2 >
 ::= < Zugriffssymbol   ↑ ZUGRIFF                      >
     < Gegenstandsliste ↓ ZUGRIFF ↓ K1 ↓ PROCK ↑ K2 >.

< Zugriffssymbol ↑ ZUGRIFF >
 ::= reads-Symbol          < wobei ZUGRIFF = lesen          >        /r12/
   ] writes-Symbol         < wobei ZUGRIFF = schreiben      >        /r13/
   ] updates-Symbol        < wobei ZUGRIFF = ändern         >        /r14/
   ] initializes-Symbol    < wobei ZUGRIFF = initialisieren >        /r15/
   ] produces-Symbol       < wobei ZUGRIFF = liefern        >        /r16/
   ] consumes-Symbol       < wobei ZUGRIFF = holen          >        /r17/
   ] tests-Symbol          < wobei ZUGRIFF = testen         >        /r18/
   ] inhibits-Symbol       < wobei ZUGRIFF = maskieren      >        /r19/
   ] terminated-by-Symbol  < wobei ZUGRIFF = beenden        >        /r20/
   ] started-by-Symbol     < wobei ZUGRIFF = starten        >        /r21/
   ] occupies-Symbol       < wobei ZUGRIFF = belegen        >.       /r22/

< Gegenstandsliste       ↓ ZUGRIFF ↓ K1 ↓ PROCK ↑ K2 >
 ::= < Zugriffsgruppe    ↓ ZUGRIFF ↓ K1 ↓ PROCK ↑ K2 >
   ] < Gegenstandsliste  ↓ ZUGRIFF ↓ K1 ↓ PROCK ↑ K  > Komma
     < Zugriffsgruppe    ↓ ZUGRIFF ↓ K  ↓ PROCK ↑ K2 >.
```

```
< Zugriffsgruppe          ↓ ZUGRIFF ↓ K1 ↓ PROCK          ↑ K4 >
 ::= < fall ZUGRIFF ∈
       {lesen,schreiben,ändern,initialisieren,liefern,holen,testen} >
     < Medien-Verwendung ↓ ZUGRIFF ↓ K1 ↑ MEDIUM          ↑ K2 >
     < Zusicherungsteil ↓ K2 ↓ [*]+ZUGRIFF+[*]+PROCK+[*]+MEDIUM
                                          ↑ TEXT-OBJEKT ↑ K3 >
     < wobei K4 = K3 ϕ (ZUGRIFF,PROCK,MEDIUM)→TEXT-OBJEKT >
   ] < fall ZUGRIFF ∈ {maskieren,beenden,starten} >
     < Medien-Verwendung ↓ ZUGRIFF ↓ K1 ↑ MEDIUM          ↑ K2 >
     < wobei K4 = K2 ϕ (ZUGRIFF,PROCK,MEDIUM) >
   ] < fall ZUGRIFF = belegen >
     < Belegung          ↓ ZUGRIFF ↓ K1 ↓ PROCK          ↑ K4 >.

< Belegung              ↓ ZUGRIFF ↓ K1 ↓ PROCK          ↑ K4 >
 ::= multiple-Symbol
     < Zahlenangabe                ↓ K1        ↑ ANZAHL ↑ K2 >   /4.9/
     of-Symbol
     < Nennung ↓ {betriebsmittel}  ↓ K2  ↑ BETRMTTL     ↑ K3 >   /5.1/
     < wobei K4 = K3 ϕ (belegen,PROCK,BETRMTTL)→ANZAHL >          /r22/
   ] < Nennung ↓ {betriebsmittel}  ↓ K1  ↑ BETRMTTL     ↑ K2 >   /5.1/
     < wobei K4 = K2 ϕ (belegen,PROCK,BETRMTTL)→undef  >.         /r22/

(* Die zugehörige Relation ist abhängig von ZUGRIFF, siehe oben.      *)


< Medien-Verwendung ↓ ZUGRIFF              ↓ K1 ↑ MEDIUM ↑ K3 >
 ::= < fall ZUGRIFF ∈ {lesen,schreiben,ändern} >
     < Nennung ↓ {variable}                ↓ K1 ↑ MEDIUM ↑ K3 >   /5.1/
   ] < fall ZUGRIFF = initialisieren >
     < Nennung ↓ {variable,puffer,trigger} ↓ K1 ↑ MEDIUM ↑ K3 >   /5.1/
   ] < fall ZUGRIFF ∈ {liefern,holen,maskieren,testen} >
     < Nennung ↓ {puffer,trigger}          ↓ K1 ↑ MEDIUM ↑ K3 >   /5.1/
   ] < fall ZUGRIFF ∈ {starten,beenden} >
     < Nennung ↓ {trigger}                 ↓ K1 ↑ MEDIUM ↑ K3 >.  /5.1/

< Zusicherungsteil  ↓ K1 ↓ TEXT-OBJEKT1  ↑ TEXT-OBJEKT  ↑ K2 >
 ::= where-Symbol
     < Zusicherung  ↓ K1 ↓ TEXT-OBJEKT1  ↑ TEXT-OBJEKT  ↑ K2 >
   ] ε  < wobei K2 = K1 >
        < wobei TEXT-OBJEKT = undef >.

< Zusicherung       ↓ K1 ↓ TEXT-OBJEKT1  ↑ TEXT-OBJEKT  ↑ K2 >
 ::= < eingebettete Sektion
                   ↓ K1 ↓ {text-objekt} ↑ TEXT-OBJEKT  ↑ K2 >    /2.1/
   ] < Text        ↓ K1 ↓ TEXT-OBJEKT1 ↓ []           ↑ K2 >    /6.1/
     < wobei TEXT-OBJEKT = TEXT-OBJEKT1 >.

(* Die zweite Alternative beschreibt die sog. anonyme Zusicherung, bei
   der der Text automatisch Namen und (leeren) Selektor erhält.       *)
```

```
(***   4.2.4 in Kap.7 (Anh.)  Prozedur- und Block-Zerlegung   *********)

< Prozedur-Block-Zerlegung        ↓ K1 ↓ PROCK ↑ K3 >
 ::= < sequentielle Zerlegung     ↓ K1 ↓ PROCK ↑ K2 >          /4.2.4.1/
     < wobei K3 = K2 ϕ (ablauf,PROCK)→sequentiell >               /a2/
   ] < parallele Zerlegung        ↓ K1 ↓ PROCK ↑ K2 >          /4.2.4.2/
     < wobei K3 = K2 ϕ (ablauf,PROCK)→parallel >                  /a2/
   ] < alternative Zerlegung      ↓ K1 ↓ PROCK ↑ K2 >          /4.2.4.3/
     < wobei K3 = K2 ϕ (ablauf,PROCK)→alternativ >.               /a2/


(***   4.2.4.1 in Kap.7 (Anh.)  sequentielle Zerlegung   **************)

< sequentielle Zerlegung                ↓ K1 ↓ PROCK          ↑ K2 >
 ::= sequential-Symbol
     < erster sequentieller Block       ↓ K1 ↓ PROCK ↑ BLOCK ↑ K  >
     < nachfolgende Blöcke    ↓ K ↓ BLOCK  ↓ PROCK          ↑ K2 >.

< erster sequentieller Block            ↓ K1 ↓ PROCK ↑ BLOCK ↑ K2 >
 ::= < sequentieller Block              ↓ K1         ↑ BLOCK ↑ K2 >
     < test Sequenz                     ↓ K1 ↓ PROCK ↓ BLOCK >.

< test Sequenz                          ↓ K  ↓ PROCK ↓ BLOCK >
 ::= < fall (ablauf,PROCK)  ∉ S(K) >
   ] < test (sequent,BLOCK) ϵ S(K) >.

(* Ein Sohn kann also nur zu bereits vorhandenen hinzugefügt werden,
   wenn er an einen vorhandenen angehängt wird, nicht umgekehrt.      *)

< nachfolgende Blöcke       ↓ K1 ↓ BLOCK1 ↓ PROCK          ↑ K2 >
 ::= ε < wobei K2 = K1 ϕ (sequent,BLOCK1)→(PROCK,undef) >          /r2/
   ] < nachfolgender Block  ↓ K1 ↓ BLOCK1 ↓ PROCK ↑ BLOCK ↑ K  >
     < nachfolgende Blöcke  ↓ K ϕ (sequent,BLOCK1)→(PROCK,BLOCK)
                            ↓ BLOCK  ↓ PROCK          ↑ K2 >.   /r2/

< nachfolgender Block       ↓ K1 ↓ BLOCK1 ↓ PROCK ↑ BLOCK ↑ K2 >
 ::= then-Symbol
     < sequentieller Block  ↓ K1                   ↑ BLOCK ↑ K2 >.

< sequentieller Block       ↓ K1                   ↑ BLOCK ↑ K2 >
 ::= < Block-Sektion        ↓ K1                   ↑ BLOCK ↑ K2 >  /2.1/
     < test (parallel  ,BLOCK) ∉ S(K1) >
     < test (alternativ,BLOCK) ∉ S(K1) >
     < test Block-Hierarchie ↓ K2          ↓ PROCK ↓ BLOCK >.  /4.2.4.4/


(***   4.2.4.2 in Kap.7 (Anh.)  parallele Zerlegung   ***************)

< parallele Zerlegung    ↓ K1 ↓ PROCK          ↑ K2 >
 ::= < parallele Blöcke  ↓ K1 ↓ PROCK          ↑ K2 >.

< parallele Blöcke       ↓ K1 ↓ PROCK          ↑ K2 >
 ::= < paralleler Block  ↓ K1 ↓ PROCK          ↑ K2 >
   ] < paralleler Block  ↓ K1 ↓ PROCK          ↑ K  >
     < parallele Blöcke  ↓ K  ↓ PROCK          ↑ K2 >.
```

```
< paralleler Block ↓ K1 ↓ PROCK ↑ K2 ϕ (parallel,BLOCK)→PROCK >      /r3/
 ::= parallel-Symbol
     < Block-Sektion          ↓ K1     ↑ BLOCK ↑ K2 >                /2.1/
     < test (sequent    ,BLOCK) ∉ S(K1) >
     < test (alternativ,BLOCK) ∉ S(K1) >
     < test Block-Hierarchie ↓ K2 ↓ PROCK ↓ BLOCK >.           /4.2.4.4/

(***   4.2.4.3 in Kap.7 (Anh.)  alternative Zerlegung   **************)

< alternative Zerlegung  ↓ K1 ↓ PROCK          ↑ K2 >
 ::= < Selektor          ↓ K1 ↓ PROCK          ↑ K2 >
   ] either-Symbol
     < Alternativen      ↓ K1 ↓ PROCK          ↑ K2 >.

< Selektor    ↓ K1 ↓ PROCK ↑ K2 ϕ (schlüssel,PROCK)→VARIABLE >      /r5/
 ::= controlled-by-Symbol
     < Nennung ↓ {variable} ↓ K1   ↑ VARIABLE ↑ K2 >.              /5.1/

< Alternativen           ↓ K1 ↓ PROCK          ↑ K2 >
 ::= < Alternative       ↓ K1 ↓ PROCK          ↑ K2 >
   ] < Alternative       ↓ K1 ↓ PROCK          ↑ K  > or-Symbol
     < Alternativen      ↓ K  ↓ PROCK          ↑ K2 >.

< Alternative            ↓ K1 ↓ PROCK
                    ↑ K2 ϕ (alternativ,BLOCK)→(PROCK,KONST) >      /r4/
 ::= < Kriterium         ↓ K1          ↑ KONST ↑ K  >
     < Alternativ-Block  ↓ K  ↓ PROCK ↑ BLOCK ↑ K2 >.

< Kriterium                   ↓ K1 ↑ KONST ↑ K2 >
 ::= key-Symbol
     < Nennung ↓ {konstante} ↓ K1 ↑ KONST ↑ K2 >                   /5.1/
     for-Symbol
   ] ε  < wobei K2 = K1 >
        < wobei KONST = undef >.

< Alternativ-Block           ↓ K1 ↓ PROCK ↑ BLOCK ↑ K2 >
 ::= < Block-Sektion         ↓ K1         ↑ BLOCK ↑ K2 >           /2.1/
     < test (sequent ,BLOCK) ∉ S(K1) >
     < test (parallel,BLOCK) ∉ S(K1) >
     < test Block-Hierarchie ↓ K2 ↓ PROCK ↓ BLOCK      >.     /4.2.4.4/


(***   4.2.4.4 in Kap.7 (Anh.)  Test der Block-Hierarchie   **********)

< test Block-Hierarchie          ↓ PROCK ↓ BLOCK ↓ K >
 ::= < test PROCK ≠ BLOCK >
     < test Block-Hierarchie-2   ↓ PROCK ↓ BLOCK ↓ K >.

< test Block-Hierarchie-2        ↓ PROCK ↓ BLOCK ↓ K >
 ::= < fall (sequent   ,PROCK) ∉ S(K) >
     < fall (parallel  ,PROCK) ∉ S(K) >
     < fall (alternativ,PROCK) ∉ S(K) >
   ] < fall (sequent   ,PROCK) ∈ S(K) >
     < test Block-Hierarchie  ↓ A(K,(sequent,   PROCK),1) ↓ BLOCK ↓ K >
   ] < fall (parallel  ,PROCK) ∈ S(K) >
     < test Block-Hierarchie  ↓ A(K,(parallel,  PROCK),1) ↓ BLOCK ↓ K >
   ] < test Block-Hierarchie  ↓ A(K,(alternativ,PROCK),1) ↓ BLOCK ↓ K >.
```

```
(***   4.3 in Kap.7 (Anh.)  Variablen-Angaben   **********************)

< Variablen-Angabe                ↓ K1 ↓ VARIABLE ↑ K2 >
 ::= < Restriktion ↑ RESTRIKT     ↓ K1 ↓ VARIABLE ↑ K2 >              /5.2/
     < test RESTRIKT 6 {lese-r,schreib-r} >
   ] < Variablen-Zerlegung        ↓ K1 ↓ VARIABLE ↑ K2 >.

< Variablen-Zerlegung  ↓ K1 ↓ VARIABLE ↑ K2 >
 ::= < Typ-Zerlegung   ↓ K1 ↓ VARIABLE ↑ K2 >                       /4.6/
   ] < Verbund         ↓ K1 ↓ VARIABLE ↑ K  >
     < wobei K2 = K ϕ (struktur,VARIABLE)→verbund >.                /a3/

< Verbund              ↓ K1 ↓ VARIABLE ↑ K2 >
 ::= consists-of-Symbol
     < Verbundvariable ↓ K1 ↓ VARIABLE ↑ K2 >
   ] < Verbund         ↓ K1 ↓ VARIABLE ↑ K  >
     and-Symbol
     < Verbundvariable ↓ K  ↓ VARIABLE ↑ K2 >.

< Verbundvariable              ↓ K1 ↓ VARIABLE0
                      ↑ K2 ϕ (var-verbund,VARIABLE)→VARIABLE0 >    /r30/
 ::= < eingebettete Sektion  ↓ K1 ↓ {variable} ↑ VARIABLE ↑ K2 >    /2.1/
     < test Var-Hierarchie   ↓ K2 ↓ VARIABLE0  ↓ VARIABLE      >.

< test Var-Hierarchie        ↓ K  ↓ VARIABLE0 ↓ VARIABLE >
 ::= < fall (var-verbund,VARIABLE0) ∉ S(K) >
     < test VARIABLE0 ≠ VARIABLE >
   ] < test Var-Hierarchie   ↓ K  ↓ A(K,(var-verbund,VARIABLE0))
                                          ↓ VARIABLE >
     < test VARIABLE0 ≠ VARIABLE >.


(***   4.4 in Kap.7 (Anh.)  Puffer- und Trigger-Angaben   ************)

(***   4.4.1 in Kap.7 (Anh.)  Puffer-Angaben   ***********************)

< Puffer-Angabe                   ↓ K1 ↓ PUFFER ↑ K2 >
 ::= < Puffer-Trigger-Angabe      ↓ K1 ↓ PUFFER ↑ K2 >          /4.4.3/
   ] < Puffertyp                  ↓ K1 ↓ PUFFER ↑ K2 >
   ] < statischer Speicherbedarf  ↓ K1 ↓ PUFFER ↑ K2 >
   ] < dynamischer Speicherbedarf ↓ K1 ↓ PUFFER ↑ K2 >
   ] < Pufferorganisation         ↓ K1 ↓ PUFFER ↑ K2 >.


< Puffertyp          ↓ K1 ↓ PUFFER       ↑ K2 >
 ::= < Typzuordnung ↓ K1 ↓ PUFFER ↑ TYP ↑ K2 >.                    /5.1/

< statischer Speicherbedarf  ↓ K1 ↓ PUFFER ↑ K2 ϕ (smr,PUFFER)→ANZAHL >
 ::= stat-memory-requ-Symbol                                       /r42/
     < Zahlenangabe          ↓ K1 ↑ ANZAHL ↑ K2 >.                 /4.9/

< dynamischer Speicherbedarf ↓ K1 ↓ PUFFER ↑ K2 ϕ (mpi,PUFFER)→ANZAHL >
 ::= memory-per-item-Symbol                                        /r43/
     < Zahlenangabe          ↓ K1 ↑ ANZAHL ↑ K2 >.                 /4.9/
```

```
< Pufferorganisation                    ↓ K1 ↓ PUFFER ↑ K2 >
 ::= fifo-Symbol
     < wobei K2 = K1 ϕ (überlauf-verhalten,PUFFER)→fifo   >          /a5/
   ] by-priority-Symbol
     < wobei K2 = K1 ϕ (überlauf-verhalten,PUFFER)→prior  >.         /a5/


(***   4.4.2 in Kap.7 (Anh.)  Trigger-Angaben   ***********************)

< Trigger-Angabe                    ↓ K1 ↓ TRIGGER ↑ K2 >
 ::= < Puffer-Trigger-Angabe        ↓ K1 ↓ TRIGGER ↑ K2 >            /4.4.3/
   ] < Verzögerung                  ↓ K1 ↓ TRIGGER ↑ K2 >
   ] < Periode                      ↓ K1 ↓ TRIGGER ↑ K2 >.

< Verzögerung ↓ K1 ↓ NAME
                    ↑ K2 ϕ (verzög,NAME)→(FRIST,ANZAHL) >           /r26/
 ::= delay-Symbol < Zeitangabe ↓ K1 ↑ FRIST ↑ ANZAHL ↑ K2 >.        /4.8/

(* NAME ϵ {TRIGGER,PROZEDUR}                                        *)

< Periode      ↓ K1 ↓ TRIGGER
                    ↑ K2 ϕ (periode,TRIGGER)→(FRIST,ANZAHL) >      /r27/
 ::= cycle-Symbol < Zeitangabe ↓ K1 ↑ FRIST ↑ ANZAHL ↑ K2 >.        /4.8/


(***   4.4.3 in Kap.7 (Anh.)  Angaben zu beiden Arten   ***************)

< Puffer-Trigger-Angabe             ↓ K1 ↓ PUGGER ↑ K2 >
 ::= < Restriktion ↑ RESTRIKT       ↓ K1 ↓ PUGGER ↑ K2 >            /5.2/
     < test RESTRIKT ϵ {liefer-r,hol-r,start-r,maskier-r} >
   ] < Kapazitätsangabe             ↓ K1 ↓ PUGGER ↑ K2 >
   ] < Pufferverhalten              ↓ K1 ↓ PUGGER ↑ K2 >.

< Kapazitätsangabe  ↓ K1 ↓ NAME ↑ K2 ϕ (kapazität,PUGGER)→ANZAHL >  /r41/
 ::= capacity-Symbol
     < Zahlenangabe ↓ K1            ↑ ANZAHL ↑ K2 >.                /4.9/

< Pufferverhalten                   ↓ K1 ↓ PUGGER ↑ K2 >
 ::= skip-input-Symbol
     < wobei K2 = K1 ϕ (überlaufverhalten,PUGGER)→skip  >           /a4/
   ] block-input-Symbol
     < wobei K2 = K1 ϕ (überlaufverhalten,PUGGER)→block >.          /a4/


(***   4.5 in Kap.7 (Anh.)  Betriebsmittel-Angaben   **************************)

< Betriebsmittel-Angabe             ↓ K1 ↓ BETRMTTL ↑ K2 >
 ::= < Liste-zugr-ber-Moduln        ↓ K1 ↓ BETRMTTL ↑ K2 >          /5.2/
   ] < Kapazitätsangabe             ↓ K1 ↓ BETRMTTL ↑ K2 >.       /4.4.3/
```

```
(***   4.6 in Kap.7 (Anh.)  Typ-Verfeinerung   ************************)

< Typ-Angabe              ↓ K1 ↓ TYP        ↑ K2 >
 ::= < Typ-Zerlegung      ↓ K1 ↓ TYP        ↑ K2 >
   ] < Typ-Verbund        ↓ K1 ↓ TYP        ↑ K  >
     < wobei K2 = K ϕ (struktur,TYP)→verbund >.                    /a3/

< Typ-Zerlegung           ↓ K1 ↓ TYABLE       ↑ K2 >
 ::= < Typzuordnung       ↓ K1 ↓ TYABLE ↑ TYP ↑ K  >               /5.1/
     < wobei K2 = K ϕ (struktur,TYABLE)→typ >                      /a3/
     < test übergeordnet ↓ K1 ↓ TYABLE ↓ TYP      >
   ] < Feldangabe         ↓ K1 ↓ TYABLE       ↑ K  >
     < wobei K2 = K ϕ (struktur,TYABLE)→feld >                     /a3/
   ] < Zeiger             ↓ K1 ↓ TYABLE       ↑ K  >
     < wobei K2 = K ϕ (struktur,TYABLE)→zeiger >                   /a3/
   ] < Wertangabe         ↓ K1 ↓ TYABLE       ↑ K  >
     < wobei K2 = K ϕ (struktur,TYABLE)→aufzählung >.              /a3/

< Feldangabe ↓ K1 ↓ TYABLE ↑ K2 ϕ (feld,TYABLE)→(ANZAHL,TYP) >
 ::= contains-Symbol                                               /r33/
     < Zahlenangabe   ↓ K1 ↑ ANZAHL       ↑ K  >                   /4.9/
     of-type-Symbol
     < Typnennung     ↓ K           ↑ TYP ↑ K2 >                   /5.1/
     < test übergeordnet ↓ K2 ↓ TYABLE ↓ TYP >.

< Zeiger          ↓ K1 ↓ TYABLE   ↑ K2 ϕ (zeiger,TYABLE)→TYP >     /r31/
 ::= pointer-to Symbol
     < Typnennung ↓ K1      ↑ TYP ↑ K2 >.                          /5.1/

< Wertangabe                  ↓ K1 ↓ TYABLE
                         ↑ K2 ϕ (werte,TYABLE)→KONSTANTE >        /r34/
 ::= values-Symbol
     < Nennung ↓ {konstante} ↓ K1      ↑ KONSTANTE ↑ K2 >         /5.1/
   ] < Wertangabe            ↓ K1 ↓ TYABLE         ↑ K  > Komma
     < Nennung ↓ {konstante} ↓ K       ↑ KONSTANTE ↑ K2 >.        /5.1/

< Typ-Verbund       ↓ K1 ↓ TYP ↑ K2 >
 ::= structure-of-Symbol
     < Verbundtyp   ↓ K1 ↓ TYP ↑ K2 >
   ] < Typ-Verbund  ↓ K1 ↓ TYP ↑ K  > Komma
     < Verbundtyp   ↓ K  ↓ TYP ↑ K2 >.

< Verbundtyp      ↓ K1 ↓ TYP0     ↑ K2 ϕ (komponente,TYP0,TYP1) >  /r32/
 ::= < Typnennung ↓ K1     ↑ TYP1 ↑ K2 >                          /5.1/
     < test übergeordnet ↓ K2 ↓ TYP0 ↓ TYP1 >.
```

```
< test übergeordnet              ↓ K ↓ TYABLE ↓ TYP >
 ::= < fall (art,TYABLE)→variable ∈ K >
   ] < test                          TYABLE ≠ TYP >
     < test nicht untergeordnet ↓ K ↓ TYABLE ↓ TYP >.

< test nicht untergeordnet       ↓ K ↓ TYP0   ↓ TYP1 >
 ::= < fall (typ,TYP1)  ∉ S(K) >
     < fall (feld,TYP1) ∉ S(K) >
     < fall (komponente,TYP1,alle(K)) = ∅ >          (* vgl. 7.1.4.3 *)
   ] < fall (typ,TYP1)  ∈ S(K) >
     < test übergeordnet         ↓ K ↓ TYP0   ↓ A(K,(typ ,TYP1)) >
   ] < fall (feld,TYP1) ∈ S(K) >
     < test übergeordnet         ↓ K ↓ TYP0   ↓ A(K,(feld,TYP1)) >
   ] < wobei (komponente,TYP1,TYP2) ∈ (komponente,TYP1,alle(K)) >
     < test übergeordnet         ↓ K ↓ TYP0   ↓ TYP2 >
     < test nicht untergeordnet  ↓ K \ (komponente,TYP1,TYP2)
                                   ↓ TYP0   ↓ TYP1 >.


(***   4.7 in Kap.7 (Anh.)  Parameter-Angaben   **********************)

< Parameter-Angabe     ↓ K1 ↓ PARAMETER       ↑ K2 >
 ::= < Typzuordnung    ↓ K1 ↓ PARAMETER ↑ TYP ↑ K2 >.              /5.1/


(***   4.8 in Kap.7 (Anh.)  Frist-Verfeinerung   **********************)

< Frist-Angabe              ↓ K1 ↓ FRIST ↑ K2 >
 ::= < Frist-Definition     ↓ K1 ↓ FRIST ↑ K2 >
     < test zyklusfrei      ↓ K2 ↓ FRIST ↓ A(K2,(dauert,FRIST),2) >.

< Frist-Definition          ↓ K1 ↓ NAME
                           ↑ K2 ϕ (dauert,NAME)→(ANZAHL,FRIST) >    /r25/
 ::= takes-Symbol
     < Zeitangabe           ↓ K1 ↑ ANZAHL ↑ FRIST ↑ K2 >.

< Zeitangabe                ↓ K1 ↑ ANZAHL ↑ FRIST ↑ K2 >
 ::= < Zahlenangabe         ↓ K1 ↑ ANZAHL         ↑ K  >            /4.9/
     of-Symbol
   , < Nennung ↓ {frist}    ↓ K           ↑ FRIST ↑ K2 >.           /5.1/

< test zyklusfrei           ↓ K ↓ FRIST ↓ FRIST1 >
 ::= < fall (dauert,FRIST1) ∉ S(K) >
     < test FRIST ≠ FRIST1 >
   ] < test zyklusfrei      ↓ K ↓ FRIST ↓ A(K,(dauert,FRIST1),2) >
     < test FRIST ≠ FRIST1 >.
```

```
(***   4.9 in Kap.7 (Anh.)  Konstanten-Angaben   **********************)

< Konstanten-Angabe              ↓ K1 ↓ KONSTANTE       ↑ K2 >
 ::= < Typzuordnung              ↓ K1 ↓ KONSTANTE ↑ TYP ↑ K2 >.      /5.1/

< Zahlenangabe                        ↓ K1 ↑ ANZAHL ↑ K2 >
 ::= < Anzahl                              ↑ ANZAHL      >          /7.3/
     < wobei K2 = K1 >
   ] < Nennung ↓ {variable,konstante} ↓ K1 ↑ ANZAHL ↑ K2 >.         /5.1/

(* Konstanten und Zahlen dienen hier nur der Angabe natürlicher Zahlen,
   z.B. für die Zahl der belegten Betriebsmittel usw. Wertangaben
   für reelle Zahlen, Strings usw. können nur in Form von Texten
   gemacht werden.                                                  *)


(*************************************************************************)
(***   5. in Kap.7 (Anh.)  Produktionen für mehrere Arten   ************)

(***   5.1 in Kap.7 (Anh.)  Typzuordnung und Nennung   *****************)

(* Die Verwendung eines Namens in irgendeinem Kontext setzt im allge-
   meinen voraus, daß das mit diesem Namen bezeichnete Objekt von
   ganz bestimmter Art oder von einer aus mehreren möglichen Arten ist.
   Dies wird in der Grammatik durch die Überlagerung in der Produktion
   zu Nennung dargestellt.                                          *)

< Typzuordnung    ↓ K1 ↓ NAME ↑ TYP ↑ K2 ϕ (typ,NAME)→TYP >        /r35/
 ::= of-type-Symbol
     < Typnennung ↓ K1        ↑ TYP ↑ K2                  >.

(* NAME ∈ {PARAMETER,VARIABLE,PUFFER,TYP}                          *)

< Typnennung           ↓ K1 ↑ TYP ↑ K2 >
 ::= < Nennung ↓ {typ} ↓ K1 ↑ TYP ↑ K2 >.

< Nennung ↓ ARTEN ↓ K ↑ NAME ↑ K ϕ (art,NAME)→ARTEN >              /a1/
 ::= < Name           ↑ NAME                        >.            /7.1/


(***   5.2 in Kap.7 (Anh.)  Restriktion des Zugriffs   ***************)

< Restriktion             ↑ RESTRIKT ↓ K1 ↓ NAME ↑ K2 >
 ::= < Restriktionssymbol ↑ RESTRIKT                  >
     restricted-to-Symbol
     < Modulliste         ↓ RESTRIKT ↓ K1 ↓ NAME ↑ K2 >.

< Restriktionssymbol ↑ RESTRIKT >
 ::= read-Symbol      < wobei RESTRIKT = lese-r    >              /r28/
   ] write-Symbol     < wobei RESTRIKT = schreib-r >              /r29/
   ] produce-Symbol   < wobei RESTRIKT = liefer-r  >              /r37/
   ] consume-Symbol   < wobei RESTRIKT = hol-r     >              /r36/
   ] inhibit-Symbol   < wobei RESTRIKT = maskier-r >              /r40/
   ] start-Symbol     < wobei RESTRIKT = start-r   >              /r38/
   ] terminate-Symbol < wobei RESTRIKT = beende-r  >.             /r39/
```

```
< Liste-zugr-ber-Moduln          ↓ K1 ↓ NAME ↑ K2 >
 ::= available-in-Symbol
     < Modulliste  ↓ verwend-r  ↓ K1 ↓ NAME ↑ K2 >.                  /r23/

< Modulliste       ↓ RESTRIKT   ↓ K1 ↓ NAME
                                      ↑ K2 ϕ (RESTRIKT,NAME,MODUL) >
 ::= < Nennung ↓ {modul}        ↓ K1 ↑ MODUL ↑ K2 >                 /5.1/
   ] < Nennung ↓ {modul}        ↓ K1 ↑ MODUL ↑ K  > Komma           /5.1/
     < Modulliste  ↓ RESTRIKT   ↓ K  ↓ NAME  ↑ K2 >.

(* Die zugehörige Relation ist abhängig von RESTRIKT, siehe oben.     *)


(***   5.3 in Kap.7 (Anh.)  Leere Produktionen   **********************)

< wobei wahr > ::= ε.
< test wahr >  ::= ε.
< fall wahr >  ::= ε.


(*************************************************************************)
(***   6. in Kap.7 (Anh.)  Texte und Markierungen   *******************)

(***   6.1 in Kap.7 (Anh.)  Texte   ***********************************)

(* Texte sind keine Objekte. Daher können sie auch nicht durch Rela-
   tionen mit Objekten verknüpft werden. Vielmehr wird der Text als
   Bestandteil des Objekts aufgefaßt.                                  *)


< Textangabe              ↓ K1 ↓ NAME                  ↑ K2 >
 ::= text-Symbol
     < Text-Selektor                  ↑ SELEKTOR          >
     < Text                ↓ K1 ↓ NAME ↓ SELEKTOR      ↑ K2 >.

< Text-Selektor ↑ SELEKTOR >
 ::= < Name     ↑ SELEKTOR >                                        /7.1/
   ] ε < wobei SELEKTOR = [] >.

< Text                    ↓ K1 ↓ NAME ↓ SELEKTOR                ↑ K2 >
 ::= < fall (text,NAME,SELEKTOR) ∉ S(K1) >
     Cent
     < Text ohne Rufzeichen                       ↓ []    ↑ TEXT1      >
     < Rufzeichengruppen ↓ K1 ↓ NAME ↓ SELEKTOR ↓ TEXT1 ↑ TEXT  ↑ K  >
     Cent
     < wobei K2 = K ϕ (text,NAME,SELEKTOR)→TEXT >
   ] < fall (text,NAME,SELEKTOR) ∈ S(K1) >
     Cent < Textzeichen-Folge > Cent
     < wobei K2 = K1 >.

< Textzeichen-Folge >
 ::= < Textzeichen ↑ ZEICHEN >                                     /8./
     < Textzeichen-Folge >
   ] ε.
```

```
< Rufzeichengruppen      ↓ K1 ↓ NAME ↓ SELEKTOR ↓ TEXT1 ↑ TEXT2 ↑ K2 >
 ::= < Rufzeichengruppe  ↓ K1 ↓ NAME ↓ SELEKTOR ↓ TEXT1 ↑ TEXT  ↑ K  >
     < Rufzeichengruppen ↓ K  ↓ NAME ↓ SELEKTOR ↓ TEXT  ↑ TEXT2 ↑ K2 >
   ] ε  < wobei K2 = K1 >
        < wobei TEXT2 = TEXT1 >.

< Rufzeichengruppe       ↓ K1 ↓ NAME ↓ SELEKTOR ↓ TEXT1 ↑ TEXT2 ↑ K2 >
 ::= < Verweis           ↓ K1 ↓ NAME ↓ SELEKTOR ↓ TEXT1 ↑ TEXT  ↑ K2 >
     < Verweis-Schwanz                          ↓ TEXT  ↑ TEXT2      >
   ] Rufzeichen
     < Verweis-Schwanz              ↓ TEXT1+[Rufzeichen] ↑ TEXT2      >
     < wobei K2 = K1 >.

< Verweis ↓ K ↓ NAME ↓ SELEKTOR ↓ TEXT ↑ TEXT+[Rufzeichen]+VERWEIS
                          ↑ K ϕ (verweis,NAME,SELEKTOR,VERWEIS) >  /r45/
 ::= Rufzeichen < Name ↑ VERWEIS >.                                /7.1/

(* Zu "Rufzeichen" siehe Kommentar unter 8.                          *)
(* Zwischen Rufzeichen und Name sind Leerzeichen nicht erlaubt.      *)

< Verweis-Schwanz                    ↓ TEXT1           ↑ TEXT2 >
 ::= < Textzeichen außer Rufzeichen und Buchstabe
                                        ↑ ZEICHEN          >    /8./
     < Text ohne Rufzeichen          ↓ TEXT1+[ZEICHEN] ↑ TEXT2 >
   ] ε < wobei TEXT1 = TEXT2 >.

< Text ohne Rufzeichen               ↓ TEXT1           ↑ TEXT2 >
 ::= < Textzeichen außer Rufzeichen       ↑ ZEICHEN          >    /8./
     < Text ohne Rufzeichen          ↓ TEXT1+[ZEICHEN] ↑ TEXT2 >
   ] ε < wobei TEXT1 = TEXT2 >.


(***   6.2 in Kap.7 (Anh.)  Markierungen   ****************************)

< Markierung           ↓ K1 ↓ NAME ↑ K2 >
 ::= keywords-Symbol
     < Stichwortangabe ↓ K1 ↓ NAME ↑ K2 >
   ] < Markierung      ↓ K1 ↓ NAME ↑ K  > Komma
     < Stichwortangabe ↓ K  ↓ NAME ↑ K2 >.

< Stichwortangabe                ↓ K1 ↓ NAME
                        ↑ K2 ϕ (stichwort,NAME,TEXT-OBJEKT) >     /r44/
 ::= < Nennung ↓ {text-objekt} ↓ K1 ↑ TEXT-OBJEKT ↑ K2 >.         /5.1/
```

```
(*************************************************************************)
(***   7. in Kap.7 (Anh.)  Namen, Wortsymbole und Zahlen   ***********)

(* Dieser und der folgende Abschnitt enthalten alle Produktionen, die
   nicht durch Leerzeichen verändert werden dürfen; sonst sind Leer-
   zeichen überall zulässig und bleiben (außer in Texten) ohne Ein-
   fluß. Eine hinsichtlich der Leerzeichen exakte Grammatik ergibt sich
   also, wenn in allen vorangegangenen Produktionen, in 7.6.1 jedoch nur
   in der ersten, hinter jeder syntaktischen Variablen angefügt wird:
                     < Leerzeichenfolge >.

   Zusätzlich wird dann benötigt:
   < Leerzeichenfolge >
    ::= Leerzeichen < Leerzeichenfolge >
      ] ε.

   Kommentare werden wie Leerzeichen behandelt (siehe 7.8).          *)


(***   7.1 in Kap.7 (Anh.)  Namen   ***********************************)

< Name                              ↑ NAME >
 ::= < Buchstabe  ↑  ZEICHEN                >                        /8./
     < Namensende ↓ [ZEICHEN] ↑ NAME >.

< Namensende      ↓ ANFANG                  ↑ NAME >
 ::= < Namenszeichen          ↑ ZEICHEN          >                   /8./
     < Namensende     ↓ ANFANG+[ZEICHEN] ↑ NAME >
   ] ε < wobei NAME = ANFANG >.

(* Die Länge der Namen ist durch die Implementierung begrenzt.       *)


(***   7.2 in Kap.7 (Anh.)  Wortsymbole   *****************************)

(* Wortsymbole sind in der Grammatik einheitlich dargestellt durch
   "xxx-Symbol", wobei xxx eine Zeichenfolge vertritt, die den Regeln
   für Namen genügt. ("xxxsymbol" bezeichnet   k e i n   Wortsymbol.)
   Während sie in den Beispielen dieser Arbeit durch Unterstreichung
   markiert sind, werden sie bei üblicher Eingabe in Hochkommata ge-
   setzt.                                                            *)

xxx-Symbol
 ::= Hochkomma < Name ↑ xxx > Hochkomma.                             /7.1/

(* Durch konsistente Ersetzung von xxx sind daraus alle Produktionen
   für Wortsymbole konstruierbar (siehe auch 7.10).                  *)


(***   7.3 in Kap.7 (Anh.)  Zahlen   **********************************)

< Anzahl       ↑ [ZEICHEN]+ANZAHL >
 ::= < Ziffer ↑  ZEICHEN          >                                  /8./
     < Anzahl             ↑ ANZAHL >
   ] < Ziffer ↑  ZEICHEN          >                                  /8./
     < wobei               ANZAHL = [] >.
```

```
(**************************************************************************)
(***   8. in Kap.7 (Anh.)  Einzelzeichen   ***********************************)

(* Die Sonderzeichen (siehe unten) und das Zeichen "Cent" werden hier
   nicht auf die eigentlichen Terminalzeichen abgebildet, um Verwechs-
   lungen zwischen Symbolen und Metasymbolen zu vermeiden.          *)


< Textzeichen            ↑ ZEICHEN  >
 ::= < Namenszeichen     ↑ ZEICHEN  >
     < Sonderzeichen     ↑ ZEICHEN  >.

< Namenszeichen  ↑ ZEICHEN          >
 ::= < Buchstabe ↑ ZEICHEN          >
   ] < Ziffer    ↑ ZEICHEN          >
   ] Minus < wobei ZEICHEN = Minus >.


< Buchstabe  ↑ ZEICHEN     >
 ::= A < wobei ZEICHEN = A >
   ] B < wobei ZEICHEN = B >
   ] C < wobei ZEICHEN = C >
     ...
     ...
   ] Z < wobei ZEICHEN = Z >
   ] a < wobei ZEICHEN = a >
     ...
     ...
   ] z < wobei ZEICHEN = z >.


< Ziffer     ↑ ZEICHEN     >
 ::= 0 < wobei ZEICHEN = 0 >
   ] 1 < wobei ZEICHEN = 1 >
     ...
     ...
   ] 9 < wobei ZEICHEN = 9 >.


< Sonderzeichen          ↑ ZEICHEN                 >
 ::= Hochkomma     < wobei ZEICHEN = Hochkomma     >
   ] Dollar        < wobei ZEICHEN = Dollar        >
   ] Komma         < wobei ZEICHEN = Komma         >
   ] Punkt         < wobei ZEICHEN = Punkt         >
   ] Semikolon     < wobei ZEICHEN = Semikolon     >
   ] Leerzeichen   < wobei ZEICHEN = Leerzeichen   >
   ] Kommentar     < wobei ZEICHEN = Leerzeichen   >
   ] Doppelpunkt   < wobei ZEICHEN = Doppelpunkt   >
   ] Rufzeichen    < wobei ZEICHEN = Rufzeichen    >
   ] Stern         < wobei ZEICHEN = Stern         >
   ] Schrägstrich  < wobei ZEICHEN = Schrägstrich  >
   ] Fragezeichen  < wobei ZEICHEN = Fragezeichen  >
   ] Plus          < wobei ZEICHEN = Plus          >.

(* "Kommentar" ist wie in der Metasyntax dieser Grammatik definiert
   (siehe 7.1.1.2).                                                 *)
```

```
< Textzeichen außer Rufzeichen  ↑ ZEICHEN >
 ::= < Textzeichen              ↑ ZEICHEN >
     < test                       ZEICHEN ≠ Rufzeichen >.

< Textzeichen außer Rufzeichen und Buchstabe ↑ ZEICHEN >
 ::= < Textzeichen außer Rufzeichen          ↑ ZEICHEN >
     < test        ZEICHEN ∉ {A,B,C, ... ,Z,a, ... ,z} >.


(*************************************************************************)
(***   9. in Kap.7 (Anh.)  Standard-Kontext   ***************************)

(* Der Standard-Kontext wird hier beschrieben durch eine ESPRESO-
   Spezifikation, deren Verarbeitung den Standard-Kontext ergibt.
   Alle darin vorkommenden Objekte und Verknüpfungen erhalten den
   Status "vordefiniert" (siehe 4.2.4.3).

       'type' integer;  'type' count;  'type' real;  'type' string;

       'type' boolean: 'values' true, false 'end' boolean;
                       'constant' true;  'constant' false;

       'interval'    h:   60 'of'  min 'end'   h;
       'interval'  min:   60 'of'  sec 'end' min;
       'interval'  sec: 1000 'of' msec 'end' sec;
       'interval' msec.                                              *)


(*************************************************************************)
(***   10. in Kap.7 (Anh.)  Lockerungen der Syntax   ********************)

(* block-Symbol kann entfallen in Prozedur-Zerlegungen,
   d.h. hinter 'sequential' und 'then', 'parallel', 'either' und 'or'.

   variable-Symbol kann entfallen in Variablen-Zerlegungen,
   d.h. hinter 'consists-of'.                                        *)

(* text-Symbol kann entfallen, wenn der Text als erste Angabe des
   Sektionsrumpfes steht. Da der Selektor auch leer sein kann, ist
   es also zulässig, den Rumpf mit einem Text zu beginnen.           *)

(* Bei Wortsymbolen kann das zweite Hochkomma entfallen, wenn ein
   Sonderzeichen (siehe 7.8) außer Hochkomma oder ein Cent folgt.    *)

(* Der Name am Ende der Sektion kann entfallen, wenn die Sektion keine
   eingeschachtelten Sektionen enthält und keinen Abschnitt bildet.  *)
```

8. Literaturverzeichnis

Die Angaben sind alphabetisch nach dem (ersten) Verfasser, dann in der Reihenfolge des Erscheinungsjahrs geordnet. Bei allen Zitaten wird auf das Erscheinungsjahr Bezug genommen, auch wenn es sich um einen Tagungsbeitrag handelt.
Bei Arbeiten, die in Form von Berichten erschienen sind, ist der Titel nicht unterstrichen.

Einige Verlage sind wie folgt abgekürzt:

AP	Academic Press, London, New York.
NHPC	North Holland Publishing Company, Amsterdam, New York, Oxford.
Springer	Springer Verlag, Berlin, Heidelberg, New York.

Primärberichte enthalten unveröffentlichte Informationen von vorläufigem und betriebsinternem Charakter. Eine Zurverfügungstellung der Berichte ist nach entsprechender einzelvertraglicher Vereinbarung über die Nutzung des darin enthaltenen know how (know-how-Vertrag) möglich. Entsprechende Anfragen sind an die Abteilung Patente und Lizenzen des KfK zu richten.

ADA (1979):
Preliminary ADA reference manual.
SIGPLAN Notices, 6, No.14.

Agerwala, T. (1977):
Some extended semaphore primitives.
Acta Informatica, 8, 201-220.

Agerwala, T., M. Flynn (1975):
On the completeness of representation schemes
for concurrent processes.
Conf. on Petri-Nets and Related Methods,
MIT, Cambridge, Mass., Juli 1975, 16 Seiten (keine Seitenzahlen).

Alford, M. (1977):
A requirements engineering methodology
for real-time processing requirements.
IEEE Trans. Software Eng., SE-3, 60-69.

Appel, K., W. Haken (1977):
The solution of the four-color-map problem.
Scientific American, 237, No.4, 108-121.

Balzer, R., N. Goldman (1979):
Principles of good software specification and their implications
for specification languages.
in Proceedings of Specification of Reliable Software (SRS),
IEEE Cat. No. 79 CH 1402-9C, pp.58-67.

Balzert, H. (1978):
Vergleichende Betrachtung modularer Sprachkonzepte.
in Alber, K. (Hrsg.) (1978): Programmiersprachen.
5. Fachtagung der GI, Braunschweig, März 1978. Springer, pp.45-72.

Bauer, F.L. (1972):
Software Engineering.
in Freiman, C.V.: Proc. of the IFIP Congress 71, NHPC, pp.530-538;
nachgedruckt in Bauer (1973), pp. 522-545.

Bauer, F.L. (ed.) (1973):
Software Engineering.
Lecture Notes in Computer Science, Vol.30, Springer.

Bauer, F.L. (1976):
Variables considered harmful.
in Bauer, F.L., Samelson, K. (ed.):
Language hierarchies and interfaces,
Lecture Notes in Computer Science, Vol.46, Springer, pp.230-241.

Bauer, F.L., M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner,
H. Partsch, P. Pepper, H. Wössner (1978):
Towards a wide spectrum language
to support program specification and program development.
SIGPLAN Notices, 13, No.12, 15-24.
nachgedruckt in Bauer, Broy (1979), pp.543-552.

Bauer, F.L. (1979):
Program development by stepwise transformation - the project CIP.
in Bauer, Broy (1979), pp.237-266.

Bauer, F.L., M. Broy (ed.) (1979):
Program construction.
Lecture Notes in Computer Science, Vol.69, Springer.

Baumann, R. (1972):
Interrupt handling in real-time control systems.
Proc. of the 2nd Europ. Seminar on Real-Time Programming,
Erlangen, pp.46-52.

Baumann, R. (1978):
Computer-aided design and implementation of control algorithms.
in Niemi, A. (ed.): IFAC 78. Pergamon Press; Vol.I, pp.649-655.

Bayer, R. (1974):
Aggregates: A software design method and its application
to a family of transitive closure algorithms.
TU München, Inst. für Informatik, Bericht Nr.7432.

Biewald, J., P. Göhner, R. Lauber, H. Schelling (1979):
EPOS - a specification and design technique for
computer controlled real-time automation systems.
4th Intern. Conf. on Softw. Eng., München, 1979,
IEEE Cat. No. 79 CH 1479 - 9C, pp.245-250.

Bochmann, G.V. (1976):
Semantic evaluation from left to right.
Commun. ACM, 19, 55-62.

Boehm, B.W. (1973):
Software and its impact: a quantitative assessment.
DATAMATION, 19, No.5, 48-59.

Boehm, B.W. (1974):
Some steps toward formal and automated aids
to software requirements analysis and design.
in Rosenfeld (1974), pp.192-197.

Boehm, B.W. (1976):
Software engineering.
IEEE Trans. Comput., C-25, 1226-1241;
nachgedruckt in Teichroew (1977).

Boehm, B.W., J.R. Brown, M. Lipow (1976):
Qantitative evaluation of software quality.
2nd Intern. Conf. on Software Engineering,
San Francisco, 1976, pp.592-605.

Brinch-Hansen, P. (1973):
Concurrent programming concepts.
ACM Computing Surveys, 5, 223-245.

Brinch-Hansen, P. (1975):
The programming language Concurrent PASCAL.
IEEE Trans. Software Eng., SE-1, 199-207.

Brinch-Hansen, P. (1978):
Distributed processes: a concurrent programming concept.
Commun. ACM, 21, 934-941.

Broy, M. (1980):
Transformation parallel ablaufender Prozesse.
Dissertation, TU München;
ebenfalls erschienen als Bericht TUM-I 8001.

Caine, S.H., E.K. Gordon (1975):
PDL - A tool for software design.
AFIPS Conf. Proc., Vol.44 (NCC 1975), pp.271-276.

Campbell, R.H., A.N. Habermann (1974):
The specification of process synchronisation by path expressions.
Lecture Notes in Computer Science, Vol.16, Springer, pp.89-102.

Cheng, L.L. (1978):
Program design languages - an introduction.
Report No. ESD-TR-77-324, Electronic Systems Division,
Hanscom Air Force Base, MA 01731.

Combelic, D. (1978):
Experience with SADT.
AFIPS Conf. Proc., Vol.24 (NCC 1978), pp.631-633.

Courtois, P.J., F. Heymans, D.L. Parnas (1971):
Concurrent control with 'readers' and 'writers'.
Commun. ACM, 14, 667-668.

DeRemer, F., H.H. Kron (1976):
Programming-in-the-large versus programming-in-the-small.
IEEE Trans. Software Eng., SE-2, 80-86.

DeWolf, J.B. (1977):
Requirements specification and preliminary design
for real-time systems.
in: Proc. of COMPSAC 77, Chicago, pp.17-23.

Dijkstra, E.W. (1968a):
Cooperating sequential processes.
in Genuys, F. (ed.): Programming languages. AP, pp.43-112.

Dijkstra, E.W. (1968b):
The structure of the THE-multiprogramming-system.
Commun. ACM, 11, 341-346.

Dijkstra, E.W. (1971):
Hierarchical ordering of sequential processes.
Acta Informatica, 1, 115-138;
nachgedruckt in Hoare, Perrot (1972), pp.72-93.

Eckert, K. (1980):
Implementierung eines Spezifikationssystems
für Prozeßrechnersoftware.
Primärbericht, unveröffentlicht (siehe S.132).

Goos, G. (1973):
Hierarchies.
in Bauer (1973), pp.29-46.

Gottschalk, W. (1978):
Enwurfsmethodik für die Prozeßautomatisierung.
in Hommel (1978), pp.300-313.

Gries, D. (1976):
Some comments on programming language design.
in Schneider, H.J., M. Nagl (Hrsg.): Programmiersprachen.
4. Fachtagung der GI, Erlangen, März 1976, Springer, pp.235-252.

Hamilton, M., S. Zeldin (1976):
Higher Order Software - A methodology for defining software.
IEEE Trans. Software Eng., SE-2, 9-32.

Hammond, L.S., D.L. Murphy, M.K. Smith (1978):
A system for analysis and verification of software design.
in: Proc. of COMPSAC 78, Chicago, pp.42-47.

Hellmann, A., J. Ludewig (1980):
PASCAL/360 im KfK.
Primärbericht, unveröffentlicht (siehe S.132).

Heninger, K.L. (1980):
Specifying software requirements for complex systems:
new techniques and their applications.
IEEE Trans. Software Eng., SE-6, 2-13.

Hershey, E.A. III (1975):
A survey of system design aids.
London, ohne nähere Quellenangabe nachgedruckt in Teichroew (1977).

Hesse, W. (1976):
Vollständige Beschreibung von Programmiersprachen
mit zweischichtigen Grammatiken.
TU München, Inst. für Informatik, Bericht Nr.7623

Hice, G.F., W.S. Turner, L.F. Cashwell (1974):
System development methodology.
American Elsevier, New York.

Hoare, C.A.R. (1972):
Towards a theory of parallel programming.
in Hoare, Perrot (1972), pp.61-71.

Hoare, C.A.R., R.H. Perrot (ed.) (1972):
Operating systems techniques.
Proc. of a seminar held at Queen's University, Belfast, 1971. AP.

Hoare, C.A.R. (1974):
Monitors: An operating system structuring concept.
Commun. ACM, 17, 549-557.

Hommel, G. (Hrsg.) (1978):
Verfahren und Hilfsmittel für Spezifikation
und Entwurf von Prozeßautomatisierungssystemen.
IDT/PDV-Fachgespräch, Karlsruhe, Juni 1978. KfK-PDV 154.

Hommel, G. (Hrsg.) (1980):
Vergleich verschiedener Spezifikationsverfahren
am Beispiel einer Paketverteilanlage.
KfK-PDV 186.

Horning, J.J., B. Randell (1973):
Process structuring.
ACM Computing Surveys, 5, 5-30.

Irvine, C.A., J.W. Brackett (1977):
Automated software engineering through structured data management.
IEEE Trans. Software Eng., SE-3, 34-40.

Jackson, K., H.F. Harte (1976):
The achievement of well-structured software
in real-time applications.
Proc. of the IFAC/IFIP workshop on real-time programming,
Rocquencourt, Frankreich, Juni 1976. pp.229-238.

Jackson, M. (1975):
Principles of program design.
AP.

Jensen, K., N. Wirth (1974):
PASCAL: user manual and report.
Lecture Notes in Computer Science, Vol.18, Springer.

Kastens, U. (1978):
Ordered attributed grammars.
Interner Bericht Nr. 7/78, Institut für Informatik II,
Universität Karlsruhe.

Kastens, U. (1979):
ALADIN - Eine Definitionssprache für attributierte Grammatiken.
Interner Bericht Nr. 7/79, Fakultät für Informatik,
Universität Karlsruhe.

Kieburtz, R.B., W. Barabash, C.R. Hill (1979):
STONY BROOK PASCAL/360, User's Guide - Release 2 -
State University of New York at Stony Brook.

Knuth, D.E. (1971):
Top-down syntax analysis.
Acta Informatica, 1, 79-110.

Koster, C.H.A. (1977):
Visibility and Types.
in Cousot, P.(ed.): Machine oriented languages Bulletin 6,
(IFIP WG 2.4), IRIA, Le Chesnay, Frankreich; pp.37-48.

Lalive d'Epinay (ed.) (1979):
TC 8 up to date report.
European Workshop on Industrial Computer Systems (EWICS),
technical committee on real-time operating systems, Paper I-1-8.

Leavenworth, B.M., J.E. Sammet (1974):
An overview of nonprocedural languages.
SIGPLAN Notices, 9, No.4, 1-12.

Liskov, B.H., S. Zilles (1975):
Specification techniques for data abstractions.
IEEE Trans. Software Eng., SE-1, 7-19.

Ludewig, J., W. Streng (1978a):
Überblick und Vergleich verschiedener Mittel
für die Spezifikation und den Entwurf von Software.
KfK 2506.

Ludewig, J., W. Streng (1978b):
Methods and tools for software specification and design - a survey.
Vortrag im European Purdue Workshop, TC for Safety and Security,
Zürich, April 1978, Paper No.149.

Ludewig, J. (1980):
PCSL - a process control software specification system.
KfK 2874.

Marcotty, M., H.F. Ledgard, G.V. Bochmann (1976):
A sampler of formal definitions.
Computing Surveys, 8, 191-276.

Matsumoto, Y. (1977):
A method of software requirements definition in process control.
in: Proc. of COMPSAC 77, Chicago, pp.128-132.

mbp (1978):
DIPOL-Workshop Okt.-Dez. 77, Abschlußbericht.
Mathematischer Beratungs- und Programmierdienst, Dortmund.

Naur, P. (ed.) (1963):
Revised report on the algorithmic language ALGOL 60.
Numerische Mathematik, 4, 420-453.

Nehmer, J., G. Goos (1978):
Computerized safeguarding of nuclear power plants:
a case for reliable software.
in Hibbard, P.G., S.A. Schumann (eds.):
Constructing quality software. NHPC, pp.11-28.

Nehmer, J. (1979):
The implementation of concurrency for a PL/I-like language.
Software Practice and Experience, 9, 1043-1057.

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

Parnas, D.L. (1974):
On a 'buzzword': hierarchical structure.
in Rosenfeld (1974), pp.336-339.

Parnas, D.L. (1977):
The use of precise specifications in the development of software.
in Gilchrist, B. (ed.) (1977): Information Processing 77.
NHPC, pp.861-867.

PDV (1977a):
Basic PEARL Sprachbeschreibung.
PDV-Bericht KFK-PDV 120.

PDV (1977b):
Full-PEARL Sprachbeschreibung.
PDV-Bericht KFK-PDV 130.

Presser, L. (1975):
Multiprogramming coordination.
ACM Computing Surveys, 7, 21-44.

Ramamoorthy, C.V., H.H. So (1977):
Survey of principles and techniques of software requirements and specifications.
in Software Engineering Techniques, Vol.2, Infotech Intern. Ltd.,
Nicholson House, Maidenhead, Berkshire, England, pp.265-318.

Rosenfeld, J.L. (ed.) (1974):
Information Processing 74.
NHPC.

Ross, D.T., K.E. Schoman (1977):
Structured analysis for requirements definition.
IEEE Trans. Software Eng., SE-3, 6-15.

Roubine, O. (1976):
The design and use of specification languages.
SRI Techn. report CSL-48.

Scheidig, H. (1975):
Einige Überlegungen zur Modularisierung
von Systemen und Systemprogrammiersprachen.
TU München, Inst. für Informatik, Bericht Nr.7506.

Schuchmann, H.R. (1977):
Programme aus Fertigteilen - oder: Ist ein Baukasten-
Ansatz als universelle Programmiertechnik praktisch sinnvoll?
Elektron. Rechenanlagen, 19, 58-63.

Simon, H.A. (1962):
The architecture of complexity.
Proc. of the American Philosophical Society, 106, 467-482.

Stay, J.F. (1976):
HIPO and integrated program design.
IBM Systems Journal, 15, 143-154.

Stevens, W.P., G.J. Myers, L.L. Constantine (1974):
Structured Design.
IBM Systems Journal, 13, 115-139.

Taylor, B. (1980):
A method for expressing the functional requirements
of real-time systems.
in Haase, V. (ed.): Proc. of IFAC/IFIP-WRTP, Graz,
14.-16. April 1980. Pergamon Press, Oxford (in Vorbereitung).

Teichroew, D. (1974):
Improvements in the system life cycle.
in Rosenfeld (1974), pp.972-978.

Teichroew, D. (1977):
Computer-aided software development.
Infotech State of the Art Tutorial, London, Februar 1977.

Teichroew, D., E.A. Hershey III (1977):
PSL/PSA: a computer-aided technique for structured
documentation and analysis of information processing systems.
IEEE Trans. Software Eng., SE-3, 41-48.

Teichroew, D., E.A. Hershey III, Y. Yamamoto (1977):
Computer-aided software development technology.
in Reliable Software. Infotech Intern. Ltd., Nicholson House,
Maidenhead, Berkshire, England, pp.129-140.

Teichroew, D., H. Sayany (1971):
Automation of system building.
DATAMATION, 17, No.8, 25-30.

Uhrig, J.L. (1978):
System requirements specification for real-time systems.
in: Proc. of COMPSAC 78, Chicago, pp.241-246.

Watt, D.A., O.L. Madsen (1977):
Extended attribute grammars.
Report no.10, University of Glasgow, Computing Departement.

Watt, D.A. (1979):
An extended attribute-grammar for PASCAL.
SIGPLAN Notices, 14, No.2, 60-74.

Wijngaarden, A. van, et al. (1975):
Revised report on the algorithmic language ALGOL 68.
Acta Informatica, 5, 1-236.

Wirth, N. (1977):
Modula: a language for modular multiprogramming.
Software Practice and Experience, 7, 3-35.

Yourdon, E. (1972):
Design of on-line computer systems.
Prentice Hall, Englewood Cliffs, New Jersey.

Yourdon, E., L.L. Constantine (1975):
Structured Design.
Yourdon Inc., New York.

9. Verzeichnis der verwendeten Abkürzungen und Literaturhinweise zu einigen Spezifikationssystemen

CIP	Computer-aided, Intuition-guided Programming (Bauer, 1979)
EPOS	Entwurfsunterstützendes, PEARL-orientiertes Spezifikationssystem (Biewald et al., 1979)
ESPRESO	System zur Erstellung der Spezifikation von Prozeßrechner-Software (siehe 3.)
ESPRESO-S	ESPRESO-Sprache (siehe 3.2)
ESPRESO-W	ESPRESO-Werkzeug (siehe 3.3)
FP2	Functions - Processes - Flowcharts - Programs (Combelic, 1978)
HIPO	Hierarchy Plus Input - Process - Output (Stay, 1976)
HOS	Higher Order Software (Hamilton, Zeldin, 1976)
JDM	Jackson Design Methodology (Jackson, 1975)
MASCOT	Modularal Approach to Software-Construction, Operation and Test (Jackson, Harte, 1976)
MIL 75	Module Interconnection Language (DeRemer, Kron, 1976)
PCSL	Process Control Software Specification Language (Ludewig, 1980)
PDL	Program Design Language (Caine, Gordon, 1975)
PEARL	Process and Experiment Automation Realtime Language (PDV, 1977a, 1977b)
PSL/PSA	Problem Statement Language / Problem Statement Analyzer (Teichroew, Hershey, 1977)
SADT	Structured Analysis and Design Technique (Ross, Schoman, 1977)
SD	Structured Design (Yourdon, Constantine, 1975)
SEF	Software Engineering Facility (Irvine, Brackett, 1977)
SREP	Software Requirements Engineering Program (Alford, 1977)