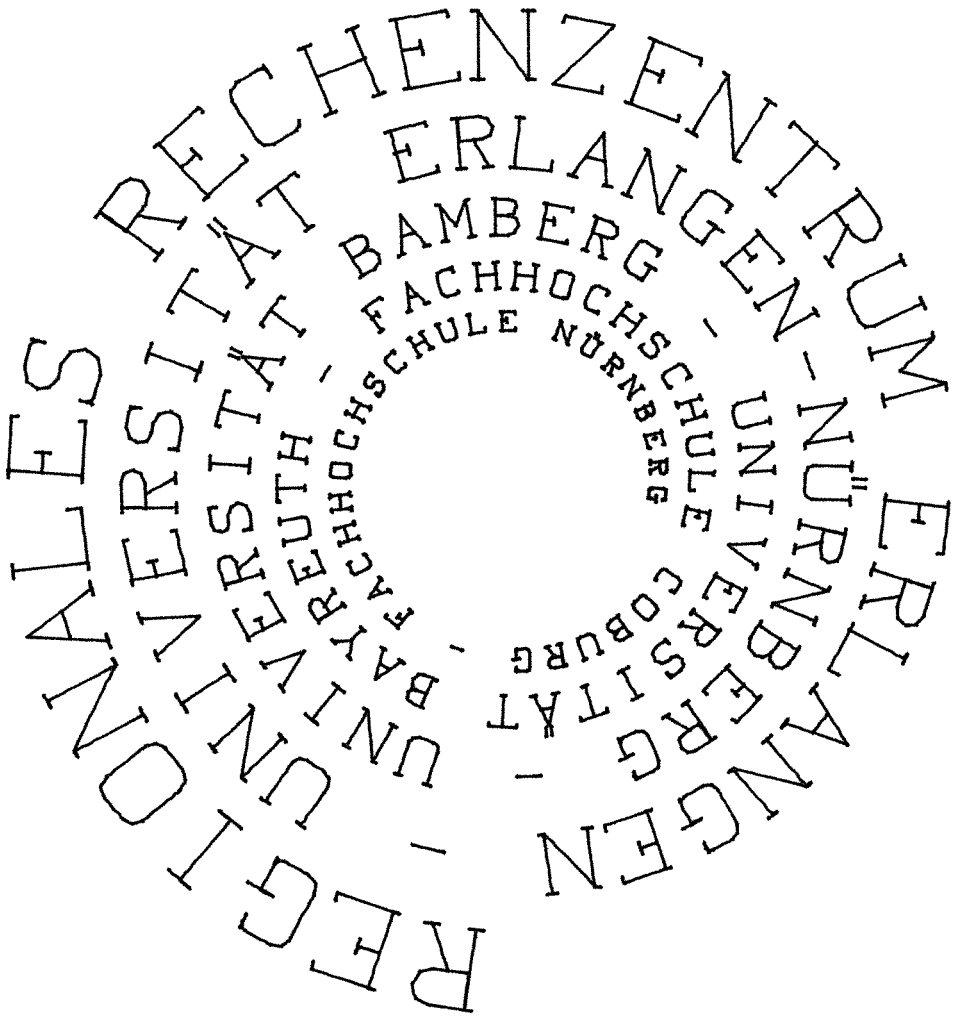
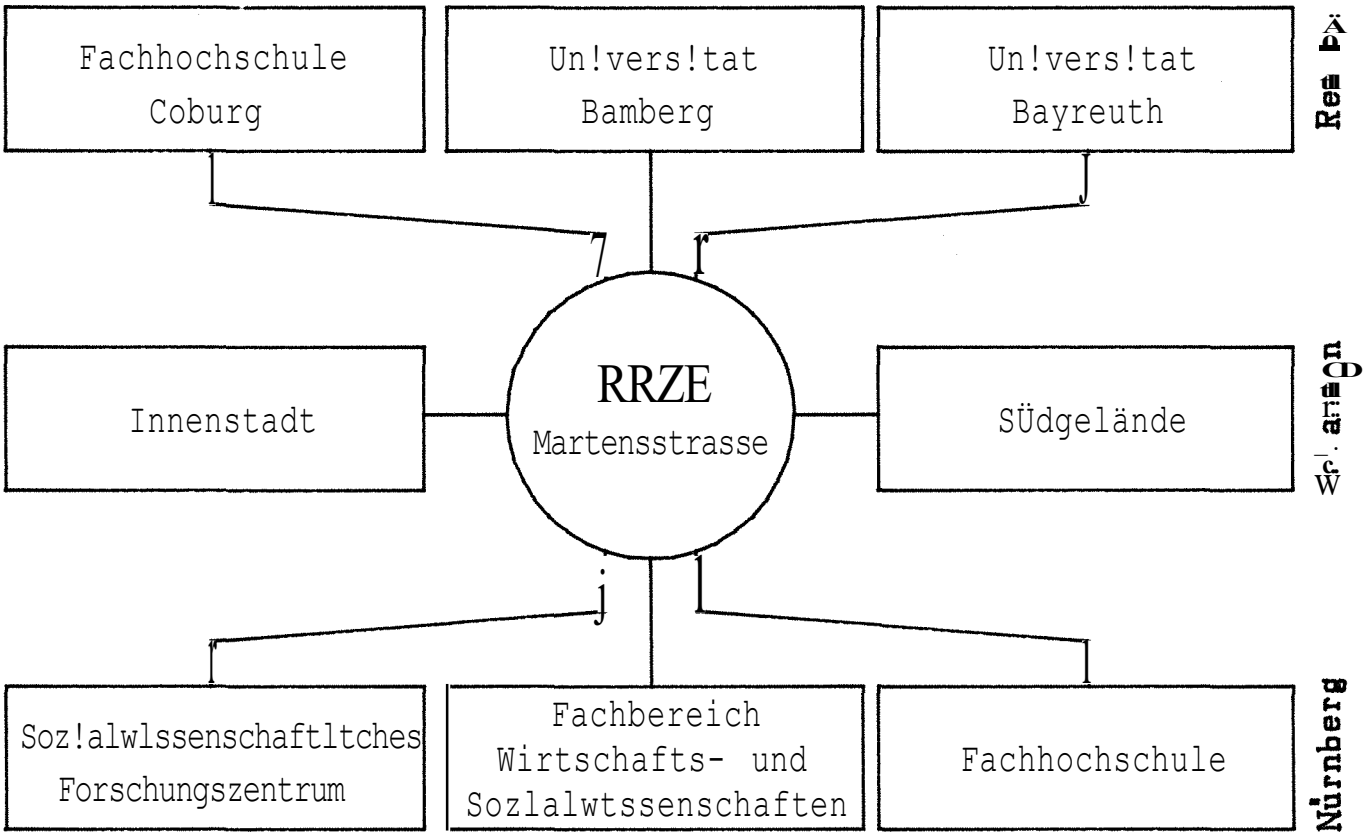
MITTEILUNGSBLATT

DES REGIONALEN RECHENZENTRUMS ERLANGEN

HERAUSGEBER F. WOLF



NR.40 - ERLANGEN - APRIL 1884



REGIONALES RECHENZENTRUM ERLANGEN

Martensstrasse 1, 8520 Erlangen, Tel., 09131/85-7031

Beteiligte Institutionen, Universität Erlangen-Nürnberg
Universität Bamberg
Universität Bayreuth
Fachhochschule Coburg
Fachhochschule Nürnberg

Redaktions H. Henke

MITTEILUNGSBLATT

DES REGIONALEN
RECHENZENTRUMS ERLANGEN

HERAUSGEBER f. WOLF

[in Konzept zur Darstellung und Realisierung van verteilten Prozessautomatisierungssystemen

Albert Fleischmann

D!ssertatlan

an der Technlscbe.a Fakultit der

Fr!edrlcb-Alexander-llnlvers!tat Erlangen-Nürnberg

NR.40 - ERLANGEN - APRIL 1984
ISSN 0172-2905

Ein Konzept zur Darstellung und Realisierung von verteilten Prozeßautomatisierungssystemen

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Albert Fleischmann

Erlangen 1983

Als Dissertation genehmigt von der
Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	28.10.1983
Tag der Promotion:	28.02.1984
Dekan:	Prof. Vetter
Berichterstatter:	Prof. Dr. F. Hofmann
	Prof. Dr. H.J. Schneider

A b s t r a c t

FLEISCHMANN, ALBERT

EIN KONZEPT ZUR DARSTELLUNG UND REALISIERUNG VON VERTEILTEN PROZESSAUTOMATISIERUNGSSYSTEMEN

In dieser Arbeit werden verschiedene Konzepte für die Synchronisation und Kommunikation paralleler Rechenprozesse vorgestellt und auf ihre Eignung für verteilte Systeme untersucht. Dabei wird besonders auf verteilte Automatisierungsprogramme eingegangen. Insbesondere wird ein Nachrichtenkonzept vorgestellt, das es erlaubt, sowohl die Kommunikation der Rechenprozesse untereinander, als auch mit dem technischen Prozeß und dem Benutzer, einheitlich darzustellen. Außerdem wird ein Spezifikationskonzept vorgestellt, mit dem verteilte Automatisierungsprogramme beschrieben werden können. Spezifikationen, die auf diesem Konzept beruhen, können auf verschiedene Fehler hin untersucht werden.

Für die Überlassung und Betreuung der Arbeit möchte ich mich bei Herrn Prof. Dr. F. Hofmann und Herrn Dr. S. Keramidis sehr herzlich bedanken.
Besonderer Dank gilt auch Herrn Dr. P. Holleczek für zahlreiche Anregungen und Hinweise.
Mein Dank gilt auch Herrn Prof. Dr. Schneider für die Übernahme des Zweitgutachtens.
Für die Hilfe beim Tippen der Arbeit möchte ich mich noch sehr herzlich bei Frau G. Brehm bedanken.

1.	Einleitung	5
2.	Spezifikation und Implementation von Programmen	7
2.1	Sequentielle Programme	10
2.1.1	Anwendungsbereiche	10
2.1.2	Strukturierungsmöglichkeiten	11
2.1.3	Anforderungen an Spezifikations- und Implementationssprachen für sequentielle Programme	12
2.2	Parallele Programme	15
2.2.1	Anwendungsbereiche	15
2.2.1.1	Dialogsysteme	16
2.2.1.2	Prozeßautomatisierungssysteme	16
2.2.2	Strukturierungsmöglichkeiten	17
2.2.3	Anforderungen an Spezifikations- und Implementationssprachen für parallele Programme	21
2.2.3.1	Beschreibung der Synchronisation und Kommunikation in Dialogsystemen	22
2.2.3.2	Beschreibung der Synchronisation und Kommunikation in Prozeßautomatisierungssystemen	22
3.	Synchronisations- und Kommunikationskonzepte	25
3.1	Auf gemeinsamen Objekten aufbauende Synchronisations- und Kommunikationskonzepte	25
3.1.1	Konzepte in Spezifikationssprachen	25
3.1.1.1	Das Softwarewerkzeug EPOS	25
3.1.1.2	Das Spezifikationskonzept nach Keramidis	29
3.1.2	Konzepte in Programmiersprachen	35
3.1.2.1	Semaphore	35
3.1.2.2	Monitore	36

3.2	Nachrichtenorientierte Synchronisations- und Kommunikationskonzepte	39
3.2.1	Konzepte in Spezifikationssprachen	40
3.2.1.1	Das Spezifikationskonzept nach Bochmann	40
3.2.1.2	Die Spezifikationssprache SDL	45
3.2.2	Konzepte in Programmiersprachen	52
3.2.2.1	Parallele Prozesse in CSP	52
3.2.2.2	Parallele Prozesse in ADA	55
3.2.2.3	Parallele Prozesse in ITP	60
3.2.2.5	Parallele Prozesse in PLITS	66
3.2.2.6	Parallele Prozesse in *JVIOD	69
3.2.2.7	Parallele Prozesse und Communication Ports nach JVIao/Yeh	72
3.2.2.8	Parallele Prozesse in DP	76
4.	Verteilte Systeme und verteilte Programme	81
4.1	Definition: verteiltes System, verteiltes Programm	81
4.2	Verteilte Systeme	83
4.2.1	Aufbau von verteilten Systemen	83
4.2.2	Anwendung von verteilten Systemen	85
4.2.2.1	Anwendung in der 'konventionellen' Programmierung	85
4.2.2.2	Anwendung bei der Steuerung technischer Prozesse	85
4.3	Verteilte Programme	87
4.3.1	Synchronisations- und Kommunikationskonzepte für verteilte Programme	87
4.3.1.1	Verteilte Programme mit gemeinsamen Objekten	88
4.3.1.1.1	Allgemeine Probleme	88
4.3.1.1.2	Probleme bei Automatisierungsprogrammen	98
4.3.1.2	Verteilte Programme und Botschaftskonzepte	99
4.3.1.2.1	Allgemeine Probleme	99
4.3.1.2.2	Probleme bei Automatisierungsprogrammen	101
4.3.2	Zusammenhang zwischen der Spezifikation und Implementation von verteilten Programmen	102

5.	Ein Konzept zur Spezifikation und Implementation verteilter Automatisierungsprogramme	105
5.1	Die Struktur von Automatisierungprogrammen	105
5.1•1	Realisierung der Kommunikation technischer Prozeß-Rechenprozesse	109
5.1•2	Realisierung der Kommunikation Benutzer-Rechenprozesse	114
5.1•3	Realisierung der Kommunikation der Rechenprozesse untereinander	117
5.2	Ein Konzept zur Spezifikation von verteilten Automatisierungsprogrammen	117
5.2.1	Die Strukturierungsmöglichkeiten des vorgeschlagenen Spezifikationskonzepts	118
5.2.1.1	Der Aufbau von Einzelprozessen	120
5.2.1.2	Der Aufbau von Prozeßbündeln	121
5.2.1.3	Der Aufbau von Prozeßgruppen	123
5.2.2	Die Beziehung von Einzelprozessen, Prozeßbündeln und Prozeßgruppen untereinander bzw. zueinander	124
5.2.3	Die Beschreibung der Bestandteile des Spezifikationskonzepts	126
5.2.3.1	Die Benutzermaschine	126
5.2.3.1.1	Definition: abstrakte Maschine	126
5.2.3.1.2	Aufgaben der Benutzermaschine	127
5.2.3.1.3	Möglichkeiten zur Beschreibung der privaten Benutzermaschine	129
5.2.3.1.4	Möglichkeiten zur Beschreibung der gemeinsamen Benutzermaschine	134
5.2.3.2	Die Kommunikationsmaschine	135
5.2.3.2.1	Das Senden	137
5.2.3.2.2	Das Empfangen	141
5.2.3.2.3	Interne Funktionen und Operationen	145
5.2.3.2.4	Bemerkungen zu den Prozeßzeigervariablen	146
5.2.3.2.5	Bemerkungen zum Wartebereich	147
5.2.3.3	Die Ablaufsteuerung	147
5.2.3.3.1	Die Darstellung der Ablaufsteuerung	148
5.2.3.3.2	Die Dynamik der Ablaufsteuerung	153

5.2.3.4	Einzelprozeß-, Prozeßbündel- und Prozeßgruppentypen	154
5.2.3.5	Der Programmentwurf	155
5.3	Prüfungsmöglichkeiten für eine Spezifikation	158
5.3.1	Vollständigkeit der Spezifikation	158
5.3.2	Erkennen bestimmter Arten von Verklemmungen	159
5.3.2.1	Verklemmungen durch Benutzermaschinenblockaden	160
5.3.2.2	Verklemmungen durch Datenblockade	161
5.3.2.3	Verklemmungen durch Kommunikationsblockaden	168
5.3.2.4	Der Zusammenhang zwischen den einzelne Verklemmungsarten	174
5.4	Sprachkonstrukte zum leichten Umsetzen einer Spezifikation in ein Programm	180
5.4.1	Beschreibung der Wirtssprache PEARL	180
5.4.2	Beschreibung der Sprachkonstrukte	182
5.4.2.1	Botschaften	182
5.4.2.2	Nichtdeterministische Kontrollanweisungen	185
5.4.3	Verteilung der Prozesse auf die Prozessoren	188
5.4.4	Einschränkungen gegenüber dem Spezifikationskonzept	188
5.5	Diskussion des vorgestellten Konzepts und erste Erfahrungen	189
6	Beispiel	191
6.1	Informelle Beschreibung des Problems	191
6.2	Programmspezifikation	192
6.3	Implementation	199
Anhang		205
Literatur		210

1 Einleitung

Die zahlreichen Möglichkeiten, Mini- und Mikrorechner als spezialisierte Rechner einzusetzen, haben dazu geführt, daß Rechnernetze in steigendem Maße auch zur Prozeßautomatisierung eingesetzt werden. Die Zerlegung komplexer Aufgabenstellungen in kleinere Teilaufgaben, die durch einzelne, diesen Teilaufgaben fest zugeordnete Rechner bearbeitet werden, hat zur Folge, daß die Struktur des verwendeten Rechnernetzes dem zu automatisierenden technischen Prozeß angepaßt werden kann. Solche Rechnernetze sind zudem zuverlässiger, effizienter und leichter erweiterbar als zentralisierte Rechnersysteme.
Für ein dezentrales Mehrrechnersystem hat sich der Begriff verteiltes System eingebürgert (siehe Kapitel 4).
Wesentlich für die Strukturierung von Automatisierungsprogrammen ist das Konzept des Prozesses. "Der Prozeß ist das beste Hilfsmittel, um auf gleichzeitig und nicht deterministisch (nicht vorhersehbar) auftretende Ereignisse der Realzeitumgebung zu reagieren" /LEVI81 /. Der abstrakte Begriff "Prozeß" ist unabhängig von der Konfiguration des verwendeten Rechnersystems (verteiltes oder zentralistisches System). Eng mit dem Prozeßkonzept verbunden sind die Probleme der Synchronisation und Kommunikation von Prozessen. Mit Hilfe von Synchronisations- und Kommunikationskonzepten organisieren die einzelnen Prozesse ihre Zusammenarbeit. In der Literatur werden zahlreiche Synchronisations- und Kommunikationskonzepte vorgeschlagen. Bei den einzelnen Konzepten wird zwischen nachrichtenorientierten und solchen mit gemeinsamen Objekten unterschieden.
Bei nachrichtenorientierten Konzepten wird die Synchronisation und Kommunikation von Prozessen durch den expliziten Austausch von Nachrichten durchgeführt.
Bei Konzepten mit gemeinsamen Objekten verwenden mehrere Prozesse (u.U. gleichzeitig) dieselben Ressourcen (z.B. Daten, Prozeduren).
In dieser Arbeit soll untersucht werden, inwieweit sich die einzelnen Synchronisations- und Kommunikationskonzepte für verteilte Systeme eignen.

Ein weiteres Problem bei Automatisierungsprogrammen ist deren Zusammenarbeit mit dem technischen Prozeß. Die Zusammenarbeit der Rechenprozesse eines Automatisierungsprogramms mit dem technischen Prozeß kann ebenfalls als Synchronisations- und Kommunikationsproblem aufgefaßt werden. Ebenso können die Ein- und Ausgaben vom bzw. zum Benutzer als Synchronisations- und Kommunikationsprobleme betrachtet werden. Beides wäre dann die Synchronisation und Kommunikation eines Automatisierungsprogramms mit der Umgebung. Zur Lösung dieser Synchronisations- und Kommunikationsprobleme werden in Automatisierungsprogrammen Ein- und Ausgabeanweisungen verwendet (siehe z.B. PEARL /DIN66853/). Es soll geprüft werden, ob es möglich ist, die Synchronisation und Kommunikation der Rechenprozesse eines Automatisierungsprogramms untereinander und mit dem technischen Prozeß mit einem einheitlichen Nachrichtenkonzept zu beschreiben.

In den letzten Jahren hat sich die Meinung durchgesetzt, daß man Programmsysteme zunächst unabhängig von ihrer Implementierung beschreiben (spezifizieren) muß. Es werden verschiedene Forderungen an solche Spezifikationskonzepte gestellt. Im Zusammenhang mit der Automatisierung von technischen Prozessen tritt zusätzlich die Forderung auf, daß der Prozeßbegriff einen integralen Bestandteil einer solchen Spezifikationsmethode bilden soll. Weiter muß es mit einer solchen Spezifikationsmethode möglich sein, die Zusammenarbeit eines Automatisierungsprogramms mit dem technischen Prozeß problemgerecht zu beschreiben.

Die Probleme mit einzelnen Synchronisations- und Kommunikationskonzepten bei verteilten Systemen werden in Kapitel 4 behandelt. Ein einheitliches Nachrichtenkonzept, mit dem die Kommunikation der Rechenprozesse untereinander und mit dem technischen Prozeß beschrieben werden kann, wird im Abschnitt 5.1 diskutiert. In den Abschnitten 5.2 bis 5.6 wird ein Konzept zur Spezifikation und Implementation von verteilten Automatisierungsprogrammen vorgestellt.

2. Spezifikation und Implementation von Programmen

Die Entwicklung von Programmen läuft in mehreren sogenannten Projektphasen ab. In der Literatur werden für die Programmentwicklung verschiedene Phaseneinteilungen vorgeschlagen /LAUB79/. Dabei werden die einzelnen Projektphasen nach den zu erstellenden Dokumenten oder den auszuführenden Tätigkeiten unterschieden.

Zur Abgrenzung der Begriffe Spezifikation und Implementation soll der 'Software-Program-Cycle' verwendet werden, der in /KKST79/ beschrieben wird. Bild 2.1 zeigt die in diesem Modell verwendete Phaseneinteilung für die Programmentwicklung.

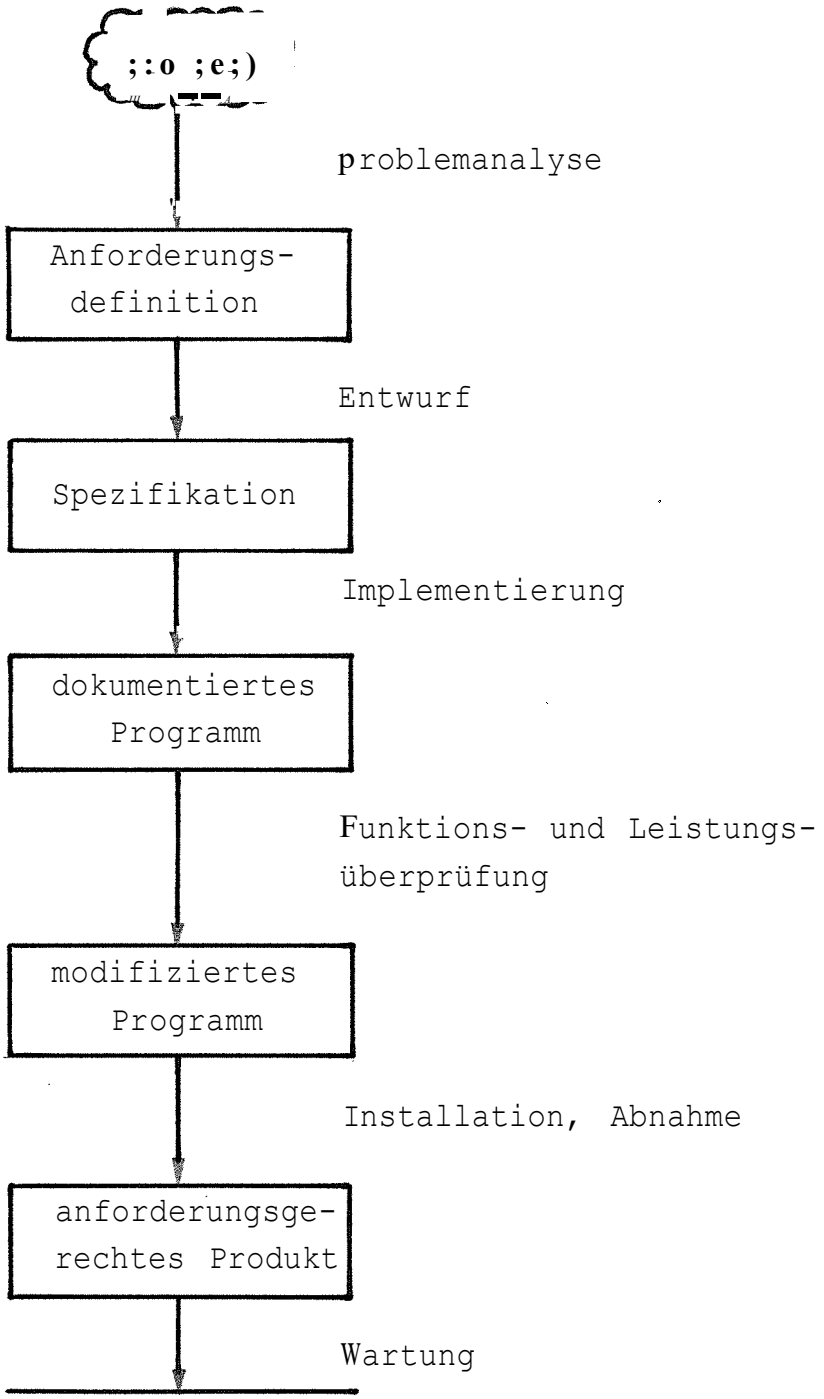


Bild 2.1: Entwicklungsphasen eines Programms nach /KKST79/.

In Bild 2.1 sind die Kanten mit den auszuführenden Tätigkeiten beschriftet und die Zustände mit den Ergebnissen dieser Tätigkeiten. Das Ergebnis des Entwurfs ist also die Spezifikation und das Ergebnis der Implementierung ein dokumentiertes Programm bzw. die Implementation. Die Sprache, in der eine Spezifikation bzw. Implementation abgefaßt ist, soll Spezifikations- bzw. Implementationssprache heißen.

Diese Arbeit befaßt sich mit der Spezifikation und Implementation einer speziellen Klasse von Anwenderprogrammen (verteilte Prozeßautomatisierung).
Beim Entwurf gilt es, ein Modell für ein Softwaresystem zu erstellen. Solch ein Modell zu bilden heißt, daß das Gesamtsystem in einzelne Teile zerlegt (z.B. Moduln) wird. Mit Hilfe von Spezifikationsmethoden werden diese Programmteile und ihre Schnittstellen beschrieben. Ein so formuliertes Programmodell wird als Spezifikation bezeichnet.
Bei der Implementierung steht nicht mehr das Gesamtsystem im Mittelpunkt der Betrachtung, sondern die bei der Spezifikation gewonnenen Teile eines Systems. Diese einzelnen Programmteile werden in einer geeigneten Programmiersprache formuliert.
Je nach dem Anwendungsgebiet, für das das zu erstellende Programmsystem bestimmt ist, ist die Verwendung unterschiedlicher Spezifikationskonzepte und Programmiersprachen vorteilhaft. Denn je nach Anwendungsgebiet werden folgende unterschiedliche "Programmtypen" verwendet:
- sequentielle Programme
- parallele Programme.

Unter einem Programm wird die exakte Beschreibung einer Berechnung verstanden, wobei eine Berechnung eine endliche Folge von Operationen ist. Eine Operation bildet eine Menge von Eingabevariablen auf eine Menge von Ausgabevariablen ab /HANS73/. Erfolgt bei der Ausführung einer Berechnung die Ausführung der Operationen streng sequentiell, so handelt es sich um ein sequentielles Programm. Die Ausführung eines sequentiellen Programms erfolgt durch einen Prozeß.
Ein Programm heißt parallel, wenn es sequentielle Programmstücke enthält, deren Operationen gleichzeitig oder überlappt ausgeführt werden. Ein paralleles Programm wird durch eine entsprechende Anzahl von Prozessen ausgeführt.
In der kommerziellen Datenverarbeitung (Verwaltung) werden überwiegend sequentielle und in der Prozeßautomatisierung parallele Programme verwendet. Impliziert die Anwendung ein paralleles Programm in dem die einzelnen Prozessezusammenarbeiten, so müssen andere Spezifikationskonzepte und Program-

miersprachen verwendet werden als bei sequentiellen Programmen. Soll ein Programmsystem entworfen werden, so ist durch den Anwendungsbereich häufig vorgegeben, ob ein sequentielles oder ein paralleles Programm erstellt werden soll. Durch diese Vorgabe wird die Vielfalt der verwendbaren Spezifikationskonzepte und auch der Programmiersprachen eingeschränkt. Soll nämlich ein paralleles Programm erstellt werden, so muß eine Programmiersprache verwendet werden, die Konzepte zur Beschreibung der Zusammenarbeit von Prozessen enthält.
In den folgenden Abschnitten sollen die Anwendungsbereiche und Strukturierungsmöglichkeiten für parallele und sequentielle Programme diskutiert werden.

2.1 Sequentielle Programme

2.1 .1. Anwendungsbereiche

Sequentielle Programme können überall dort eingesetzt werden, wo ein Programm nicht auf mehrere Ereignisse gleichzeitig warten und reagieren muß. Sequentielle Programme warten zu einem bestimmten Zeitpunkt immer nur auf ein Ereignis, z.B. auf eine Eingabe vom Bediengerät. Dieser Programmtyp wird überwiegend beim Auswerten von Daten verwendet. In der kommerziellen Datenverarbeitung ist dies z.B. das Erstellen von Lohnabrechnungen und im naturwissenschaftlichen Bereich z.B. das Auswerten von physikalischen Meßdaten. Häufig können sequentielle Programme in parallele Programme umgewandelt werden. Dies erfolgt, um die Programme schneller zu machen (Eingaben und Berechnungen können überlappt erfolgen). Die Aufgabenstellung würde dies allerdings nicht direkt erfordern, da in solchen Programmen im allgemeinen nicht auf mehrere Ereignisse gleichzeitig gewartet wird.

2.1 .2. Strukturierungsmöglichkeiten

Ein Prozeß legt die zeitliche Reihenfolge bei der Ausführung von Operationen fest. Die genaue Definition einer Operation hängt vom Abstraktionsniveau ab, auf dem ein Prozeß beschrieben werden soll. Bild 2.2 zeigt denselben Prozeß auf verschiedenen Abstraktionsniveaus /HANS73/.

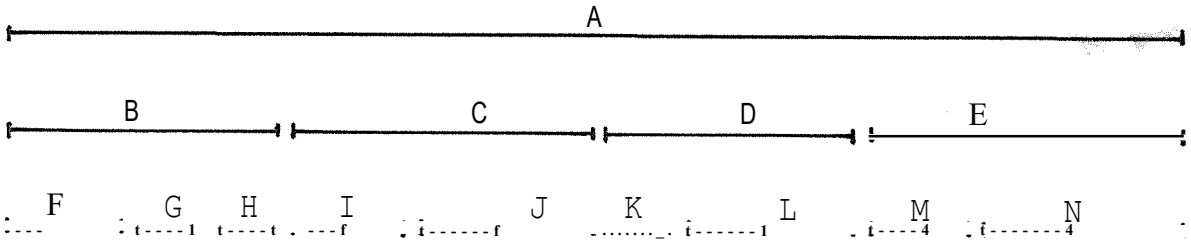


Bild 2.2 : Ein Prozeß, betrachtet auf verschiedenen Abstraktionsniveaus /HANS73/

Der gesamte Prozeß kann als eine einzige Operation (A) oder als eine Folge vieler einfacher Operationen (F bis N) gesehen werden.

Operationen manipulieren im allgemeinen den Zustand von Objekten. Die Operationen, die dasselbe Objekt manipulieren, sind charakteristisch für dieses Objekt. Die Operationen bewahren aber gewisse unveränderliche Eigenschaften eines Objekts.

Da viele Objekte im wesentlichen dieselben Verbaltenscharakteristika haben, ist es nützlich, nur eine Sammlung von Operationen zu definieren (eventuell parametrisiert), die auf mehrere Objekte anwendbar ist. Man sagt dann, zwei Objekte seien vom selben Typ, wenn sie dieselbe Sammlung von Operationen haben.

Die Objekte mit ihren Operationen werden zu einer abstrakten Maschine zusammengefaßt.

Aus diesen Überlegungen ergeben sich auch die Strukturierungskonzepte für sequentielle Programme.

Ein sequentielles Programm besteht aus einer Vorschrift, die angibt, in welcher Reihenfolge die Operationen einer abstrakten Maschine aufgerufen werden sollen (Prozeßdeklaration) und der Beschreibung der abstrakten Maschine. Bei der Beschreibung

von sequentiellen Berechnungen (sequentielle Programme) spielt es eine Rolle, auf welchem Abstraktionsniveau die abstrakte Maschine angelegt ist. Die Aufrufreihenfolge der Operationen wird in einer entsprechend langen Prozeßdeklaration beschrieben, oder die abstrakte Maschine enthält nur eine einzige sehr mächtige Operation (siehe Bild 2.2), und die Prozeßdeklaration besteht nur aus dem Aufruf dieser Operation.
Bei den Spezifikationstechniken für sequentielle Programme wird der Schwerpunkt auf eine abstrakte Maschine mit wenigen, aber komplexen Operationen gelegt.
Allerdings werden diese komplexen Operationen mit Hilfe von einfacheren Operationen beschrieben. Diese Hilfsoperationen sind außerhalb der abstrakten Maschine nicht sichtbar. Die Hilfsoperationen können natürlich auf weiteren einfachen Hilfsoperationen basieren. Dadurch entsteht innerhalb der 'sichtbaren' abstrakten Maschine eine Hierarchie von weiteren abstrakten Maschinen.
Bei der Strukturierung von abstrakten Maschinen wird im folgenden immer der Begriff 'Modul' verwendet. Moduln bestehen im allgemeinen aus Daten und Operationen auf diesen Daten und entsprechen damit den abstrakten Objekten. Die abstrakten Maschinen von sequentiellen Programmen werden durch Moduln und die Beziehung der Moduln untereinander strukturiert. Beim Entwurf wird ein Programmsystem in einzelne Moduln zerlegt. Diese Moduln können zueinander in vielfältiger Beziehung stehen ('Benutzt-'Beziehung, 'Folgt auf'-Beziehung, 'Enthält'-Beziehung usw. (siehe /KKST79/)), wobei nach /KKST79/ für die Beurteilung der Komplexität der Modulbeziehung die 'Benutzt'-Beziehung häufig verwendet wird. Ein Modul A benutzt Modul B genau dann, wenn Modul A in Modul B definierte Funktionen und Operationen aufruft oder anstößt.
Die 'Benutzt'-Struktur eines Programmsystems kann sehr vielfältig sein; sie kann vom Chaos (jeder Modul benutzt jeden) bis zur trivialen Ordnung (jeder Modul benutzt nur sich selbst) reichen. Neben den Kriterien für die Modularisierung (siehe /ALWE77/) gilt es, beim Entwurf eines Programmsystems auch eine einfache 'Benutzt'-Struktur zu erreichen. Üblicherweise wird beim Programmentwurf eine hierarchische 'Benutzt'-

Struktur angestrebt. Die verschiedenen Typen von hierarchischen 'Benutzt'-Strukturen sind in /KKST79/ beschrieben.
Die 'Benutzt'-Hierarchie muß mit der Hierarchie von abstrakten Maschinen nicht übereinstimmen..
Bei der Strukturierung von Programmen kann es durchaus vorkommen, daß die einzelenen Funktionen bzw. Operationen eines Moduls zu verschiedenen Schichten der abstrakten Maschine gehören (siehe Bild 2.3) /HABE76/.

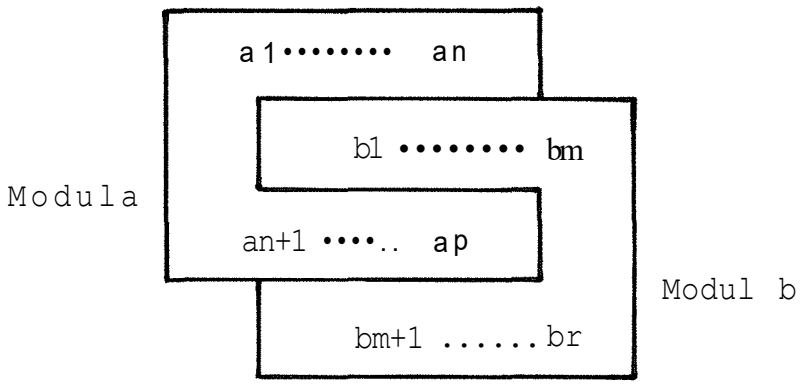


Bild 2.3 : Moduln, dessen Funktionen und Operationen verschiedenen Hierarchiestufen angehören.

2.1.3. Anforderungen an Spezifikations- und Implementationssprachen für sequentielle Programme

Die Anforderungen an Spezifikationssprachen werden aus den geforderten Eigenschaften von Spezifikationen abgeleitet.
Nach /KKST79/ sollen für eine Spezifikation folgende Qualitätsmerkmale gelten:

- Vollständigkeit und Widerspruchsfreiheit
- Minimalität
  Die Spezifikation soll alle zur Lösung des Problems notwendigen Entwurfentscheidungen enthalten und nur diese.
- Verständlichkeit

Aus diesen Qualitätsmerkmalen werden in /KKST79/ folgende Anforderungen abgeleitet:

- Formalisierbarkeit
  Die Spezifikationssprache muß es zulassen, eine Spezifikation auf Widerspruchsfreiheit und Vollständigkeit zu prüfen. Außerdem soll es möglich sein, die Implementation gegenüber der Spezifikation zu verifizieren.
- Implementationsunabhängigkeit
  Durch die Spezifikation sollen keine speziellen Algorithmen zur Problemlösung vorgegeben werden.
- Konstruierbarkeit und Rezipierbarkeit
  Für einen Programmierer muß es mit vertretbarem Aufwand erlernbar sein, seine Entwurfsideen mit Hilfe einer Spezifikationssprache zu beschreiben.
- Änderbarkeit
  Geringe Änderungen in der Anforderungsdefinition sollten deshalb nur (entsprechende) geringe Änderungen in der (entsprechenden) Spezifikation nach sich ziehen.

Die Gewichtung dieser Anforderungen an eine Spezifikationssprache hängt stark vom Anwendungsbereich für ein Programm und von der Ausbildung der Programmentwerfer bzw. Programmierer ab. Einige dieser Anforderungen können dann sogar zueinander in Widerspruch geraten.
Wird ein Betriebssystem erstellt, so kann im allgemeinen davon ausgegangen werden, daß das Entwicklungspersonal über eine entsprechende Informatikausbildung verfügt. Diese Ausbildung läßt es zu, formale Spezifikationssprachen zu verwenden, die Kenntnisse in mathematischer Logik voraussetzen.
Möchte ein Physiker ein Programm zum Auswerten seiner Meßdaten erstellen, so wäre es für ihn ein unvertretbarer Aufwand, sich Kenntnisse in mathematischer Logik anzueignen, um eine formale Spezifikationssprache verwenden zu können. Für ihn liegt der Anforderungsschwerpunkt auf der leichten Erlernbarkeit einer Spezifikationssprache, damit er sie schnell für seine Belange einsetzen kann.
Aus diesen Überlegungen läßt sich die zusätzliche Anforderung ableiten, daß eine Spezifikationssprache verschiedene Grade an

Formalisierbarkeit zulassen sollte, vor allem wenn die Spezifikationssprache in Bereichen eingesetzt werden soll, in denen allgemeinen nicht von reinen Informatikern programmiert wird (Kommerzielle Datenverarbeitung, Prozeßautomatisierung u.s.w.).
Für eine Spezifikationssprache würde dies konkret bedeuten, daß sie aus einem Rahmen besteht, in den verschiedene Besehre ibungswerkzeuge eingefügt werden können. So soll es in einem solchen Rahmen möglich sein, die Funktionen und Operationen eines Moduls entweder in Prosa oder durch formale Techniken, wie z.B. der 'algebraischen Spezifikation', zu beschreiben.

Da die Anforderungen an sequentielle Programmiersprachen für die weiteren Ausführungen nicht von Bedeutung sind, soll nur sehr kurz darauf eingegangen werden.
Die gewünschten Eigenschaften von Programmiersprachen hängen sehr stark von der speziellen Anwendung ab. Eine Auflistung der Faktoren, die für die Auswahl einer Programmiersprache von Bedeutung sind, findet sich in /SOMM82/.
Allgemein wird an eine Programmiersprache die Forderung nach einem leistungsfähigen Modulkonzept gestellt. Diese Forderung leitet sich aus den Struktur ierungsaspekten für sequentielle Programme ab.

2.2 Parallele Programme

2.2.1 Anwendungsbereiche

Parallele Programme werden eingesetzt, wenn für ein sequentielles Programm die Rechenzeit verkürzt werden soll oder wenn Programmsysteme auf sich parallel abspielende Vorgänge in ihrer 'Umwelt' reagieren müssen /LAUB76/. Im folgenden werden hauptsächlich die Anwendungen aus den zuletzt genannten Forderungen diskutiert. Solche Programmsysteme werden auch Echtzeitsysteme genannt. Bei solchen Echtzeitsystemen wird zwischen

- Dialogsystemen und
- Prozeßautomatisierungssystemen

unterschieden.

2.2.1.1. Dialogsysteme

Hier gibt ein Mensch über entsprechende Recheneinheiten Daten an ein Programm und wartet u.U. auf eine Antwort. Bild 2.4 zeigt schematisch ein Dialogsystem /LAUB76/.

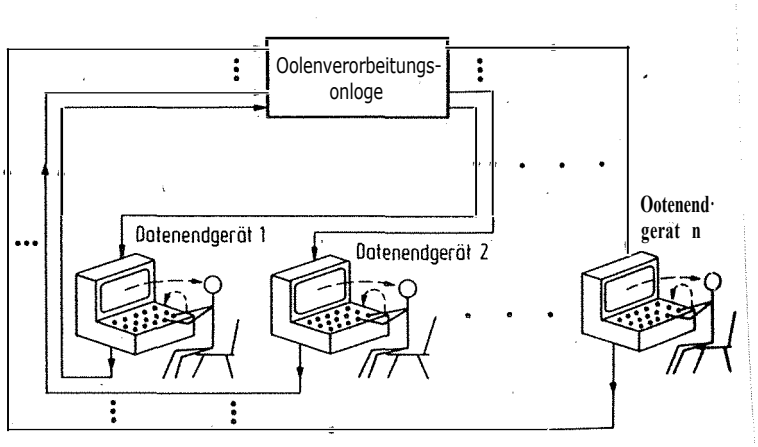


Bild 2.4: Dialogsystem /LAUB76/

Der Benutzer eines Dialogsystems kann über ein entsprechendes Ein- Ausgabegerät einen Dialog zu einem beliebigen Zeitpunkt beginnen. Es entsteht dann ein neuer Prozeß, der den Dialog mit dem "neuen" Benutzer abwickelt und die gewünschten Anweisungsfolgen ausführt.
Beispiele für Dialogsysteme sind Platzbuchungssysteme von Reisebüros oder Lagerhaltungssysteme.

2.2.1.1. Prozeßautomatisierungssysteme

Bei Prozeßautomatisierungsprogrammen muß sich der Programmablauf überwiegend nach den Vorgängen im technischen Prozeß richten /LAUB76/.
Bild 2.5 zeigt den schematischen Aufbau eines Prozeßautomati-

sierungssystems /LAUB76/.

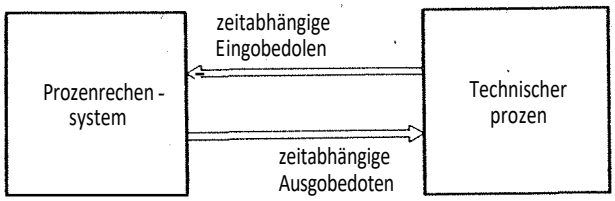


Bild 2.5 : Prozeßautomatisierungssystem /LATJB76/

Der Unterschied zwischen Dialogsystemen und Prozeßautomatisierungssystemen liegt vor allem bei den geforderten Antwortzeiten auf Ereignisse in der 'Umwelt'. Bei Dialogsystemen reicht es, wenn die Antwort auf eine Eingabe erst nach einigen Sekunden kommt, dagegen muß ein Prozeßautomatisierungssystem auf Vorfälle im technischen Prozeß oft innerhalb weniger Millisekunden reagieren.
In Prozeßautomatisierungsprogrammen ist die Anzahl der Prozesse meist statisch festgelegt.

2.2.2. Strukturierungsmöglichkeiten

Ein paralleles System wird durch mehrere parallele Prozesse ausgeführt. Jeder Prozeß führt bestimmte Operationen aus. Dadurch kann es allerdings dazu kommen, daß dieselbe Operation von verschiedenen Prozessen gleichzeitig aufgerufen wird. Es lassen sich also Operationen unterscheiden, die nur von einem Prozeß (dem 'Besitzer') aufgerufen werden und solche, die von mehreren Prozessen verwendet werden können. Objekte, die von Operationen manipuliert werden, die von mehreren Prozessen verwendet werden, bezeichnet man als gemeinsame Objekte (dieser Prozesse). Objekte, die nur von einem Prozeß verwendet werden, heißen private Objekte. Daraus ergibt sich die Grundstruktur von parallelen Programmen (siehe Bild 2.6).

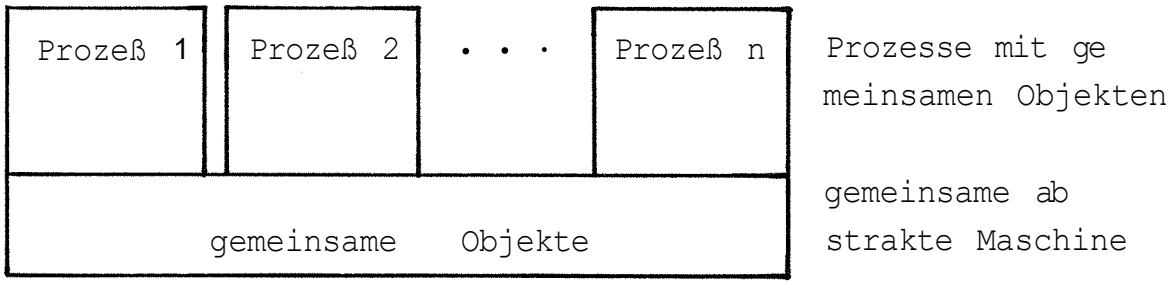


Bild 2.6: Grundstruktur von parallelen Programmen

Ein paralleles System besteht also aus mehreren Prozeßdeklarationen und einer Sammlung von privaten und gemeinsamen Objekten. Die gemeinsamen Objekte können zu einer gemeinsamen abstrakten Maschine zusammengefaßt werden. Die Objekte, die jeweils nur von einem Prozeß benutzt werden, werden zur privaten abstrakten Maschine des betreffenden Prozesses zusammengefaßt. Für die Strukturierung der privaten abstrakten Maschine gelten ähnliche Überlegungen wie bei sequentiellen Programmen, deshalb soll im folgenden nicht weiter darauf eingegangen werden. Wesentlich für die Strukturierung paralleler Programme sind die gemeinsamen Objekte und die Prozeßdeklarationen.
Bei der Strukturierung eines parallelen Programms kann der Strukturierungsschwerpunkt auf einem dieser Aspekte liegen. Wird der Schwerpunkt auf eine gemeinsame abstrakte Maschine mit entsprechend mächtigen Operationen gelegt, kann ähnlich wie bei sequentiellen Programmen verfahren werden. Allerdings ergibt sich daraus, daß die Operationen eines gemeinsamen Objekts von mehreren Prozessen u.U. gleichzeitig aufgerufen werden können, ein Problem, das bei sequentiellen Programmen nicht existiert. Um einen konsistenten Zustand eines gemeinsamen Objekts zu gewährleisten, müssen die beteiligten Prozesse über die Verwendung eines gemeinsamen Objekts, d.h. über den Aufruf diverser Operationen, Absprachen treffen, bzw. ein Objekt enthält die Vorschriften über die zulässigen Aufruffolgen der einzelnen Operationen (z.B. Pfadausdrücke /CAMP74/) direkt (Lokalitätsprinzip /KERA82/). Die Zugriffe auf ein gemeinsames Objekt müssen synchronisiert werden /KERA82/.

Bei parallelen Systemen muß nicht nur der Zugriff auf gemeinsame Objekte synchronisiert werden, sondern Prozesse müssen auch direkte Absprachen über die Ausführungsreihenfolge von Operationen treffen, die nicht zu einem gemeinsamen Objekt gehören. Diese Absprachen über Einschränkungen der Parallelität werden ebenfalls als Synchronisation bezeichnet.
Ein weiteres Problem bei parallelen Systemen ist das der Kommunikation. Damit ist gemeint, daß Prozesse Informationen austauschen müssen, damit sie "sinnvoll" weiterlaufen können. Die Probleme der Synchronisation und der Kommunikation hängen stark zusammen, da bei einem Informationsaustausch (Datenaustausch) i.a. Sychronisationsprobleme auftreten.
Wird als Strukturierungsaspekt eine abstrakte Maschine mit gemeinsamen Objekten verwendet, wobei nicht näher beschrieben wird, wie die einzelnen Prozesse aussehen bzw. wieviele es sind, so werden die gleichzeitig zulässigen bzw. sich ausschließenden Operationen auf gemeinsame Objekte direkt beim entsprechenden Objekt angegeben. Solche Synchronisationsmittel werden als ressourcenorientiert bezeichnet /GOEH81a/.
Prozesse können jedoch die Verwendung von gemeinsamen Objekten auch direkt organisieren. Die Anweisungen zur Synchronisation sind also nicht mit den Objekten verknüpft, sondern mit den Prozessen. Solche Synchronisationsmittel werden aktionsorientiert genannt /GOEH81a/.
Aktionsorientierte Synchronisationsmittel sind zur Lösung von Reihenfolgeproblemen innerhalb von Prozessen besonders gut geeignet. Gibt es zur Lösung von Synchronisationsproblemen nur ressourcenorientierte Synchronisationsmittel, müssen Reihenfolgeprobleme durch entsprechende gemeinsame Objekte gelöst werden.
Gibt es in einem parallelen Programm gemeinsame Objekte, kann das Problem der Kommunikation durch entsprechende gemeinsame Objekte gelöst werden /KERA82/. Zur Kommunikation wird der allseits bekannte Datentyp "Puffer" in seinen verschiedenen Ausprägungen verwendet. Die Zugriffe auf einen solchen Kommunikationspuffer müssen i.a. synchronisiert werden.
Werden bei der Strukturierung eines parallelen Programms keine gemeinsamen Objekte zugelassen, können nur aktionsorientierte

Synchronisationskonzepte verwendet werden. Da die einzelnen Prozesse nun keine Informationen mehr über gemeinsame Objekte austauschen können, wurden für die Kommunikation sogenannte Botschaftskonzepte eingeführt. Bei Botschaftskonzepten erfolgt der Informationsaustausch nicht über gemeinsame Objekte.
Auf welchen Strukturierungsaspekt beim Programmentwurf der Schwerpunkt gelegt wird, hängt wesentlich von der Anwendung und von der zur Verfügung stehenden Implementierungssprache ab. Wobei vor allem der letzte Punkt einer Anforderung an eine Spezifikation widerspricht, nämlich daß sie unabhängig von der Implementationssprache sein soll. Die Argumente für diese Behauptung sollen erst im Abschnitt 4-3.2 diskutiert werden. Es soll jetzt nur auf den ersten Punkt eingegangen werden.
Ein Betriebssystem für Mehrbenutzerbetrieb läßt sich besser als eine abstrakte Maschine auffassen, deren Objekte von den Benutzern (Prozesse) verwendet werden. Wie ein Benutzerprozeß aussieht, muß für ein Betriebssystem belanglos sein, deshalb verwalten die gemeinsamen Objekte sich selbst und gewährleisten damit ihren konsistenten Zustand.
Dagegen ist es vorteilhaft, bei Programmen zur Prozeßautomatisierung den Strukturierungsschwerpunkt auf Prozesse zu legen. Denn bei solchen Programmen muß vorgegeben sein, welche Aktivitäten bzw. Reaktionen von welchen Prozessen ausgeführt werden.
Bisher wurde zwischen ressourcen- und aktionsorientierten Synchronisations- und Kommunikationskonzepten unterschieden. Bei dieser Klassifikation wird als Unterscheidungsmerkmal der Ort benutzt, in dem sich die Synchronisations- bzw. Kommunikationsvorschrift befindet. Allerdings kann bei dieser Klassifikation nicht unterschieden werden, ob ein Synchronisations- und Kommunikationskonzept für gemeinsame Objekte oder für die direkte Synchronisation und Kommunikation geeignet ist. Denn ein aktionsorientiertes Konzept kann durchaus für gemeinsame Objekte gedacht sein. Ein Beispiel dafür sind Semaphore zur Synchronisation und zusätzlich globale Variable zur Kommunikation. Globale Variable werden von mehreren Prozessen benutzt und können deshalb als gemeinsame Objekte betrachtet werden (siehe Abschnitt 2.2). Semaphore zusammen mit globalen Varia-

blen sind also ein aktionsorientiertes Synchronisations- und Kommunikationskonzept, aber gedacht für Programme mit gemeinsamen Objekten. Wegen der Strukturierungsaspekte für parallele Programme, wie sie in Abschnitt 2.2 erläutert wurden, erscheint es sinnvoller, nicht zwischen aktionsorientierten und ressourcenorientierten Synchroniations- und Kommunikationskonzepten zu unterscheiden, sondern wie in /STRA81/ zwischen Konzepten mit und ohne gemeinsame Objekte. Letztere werden, wie bereits erwähnt, im weiteren Botschaftskonzepte genannt.

2.2.3. Anforderungen an Spezifikations- und Implementationssprachen für parallele Programme

Für Spezifikations- und Programmiersprachen für parallele Programme gelten die gleichen Anforderungen wie bei sequentiellen Programmen. Da bei parallelen Programmen als zusätzliche Schwierigkeit die Synchronisation und Kommunikation hinzukommt, muß es Möglichkeiten geben, die Lösung von Synchronisations- und Kommunikationsproblemen zu beschreiben.
Bei der Spezifikation und Implementation von parallelen Programmen existieren also zwei Beschreibungsprobleme:

- Die Spezifikation bzw. die Implementation der benötigten Objekte mit den darauf definierten Operationen.
- Die Spezifikation und Implementation der Synchronisationsprobleme.

Im folgenden soll nur noch auf die Anforderungen an Spezifikations- und Implementationssprachen bezüglich der Synchronisation und Kommunikation eingegangen werden. Die Anforderungen, um Objekte mit den auf ihnen definierten Operationen zu beschreiben, wurden bereits in Abschnitt 2.1.3 diskutiert.
Die Anforderungen an Konzepte zur Beschreibung der Synchronisation und Kommunikation werden wesentlich durch das Anwendungsgebiet eines Programms bestimmt. Die Synchronisations- und Kommunikationskonzepte sowohl in den Spezifikations- als auch den Programmiersprachen sind zur Beschreibung von Synchronisations- und Kommunikationsproblemen bei Dialog- oder

Prozeßautomatisierungssytemen nicht gleich gut geeignet. Deshalb werden die unterschiedlichen Anforderungen an Synchronisations- und Kommunikationskonzepte in Spezifikations- und Programmiersprachen bei der Beschreibung von Dialog- und Prozeßautomatisierungssystemen getrennt dargestellt.

2.2.3.2 Beschreibung der Synchronisation und Kommunikation in Dialogsystemen

Bei Dialogsystemen laufen die einzelnen Prozesse, die den Dialog mit einem Benutzer abwickeln, im wesentlichen unabhängig voneinander ab. Die Anzahl der Prozesse in einem Dialogsystem ist im allgemeinen nicht statisch festgelegt, d.h. es können Prozesse aus dem System verschwinden (terminiert werden) bzw. neue hinzukommen.
Die einzelnen Prozesse müssen den Zugriff zu gemeinsamen Betriebsmitteln (gemeinsame Objekte) synchronisieren.
Aus diesen allgemeinen Merkmalen für Dialogsyteme ergibt sich auch, daß der Strukturierungsschwerpunkt auf gemeinsamen Objekten liegt. Es gibt allerdings auch in Dialogsystemen gewisse Kommunikationsaspekte (z.B. Flugbuchungssysteme). Allerdings spielen diese eine untergeordnete Rolle. Sie dienen lediglich dazu, den Zugriff auf gemeinsame Daten (Objekte) zu implementieren.

2.2.3.2 Beschreibung der Synchronisation und Kommunikation in Prozeßautomatisierungssystemen.

In der Prozeßautomatisierung liegt die Aufgabenstellung zunächst nur in einer unvollständigen informellen und u.U. widersprüchlichen Form vor /LUDE81/. Eine Spezifikationstechnik für Prozeßautomatisierungsprogramme muß deshalb unvollständige und zunächst auch nur informelle Spezifikationen zulassen. Dies gilt besonders auch für Synchronisations- und Kommunikationsprobleme. Da in der Regel die zusammenhänge zwischen den einzelnen Prozessen nicht umfassend bekannt sind, muß es in

einer Spezifikationssprache möglich sein, das Synchronisationsverhalten nach und nach zu beschreiben.
Da besonders beim Entwurf von Prozeßautomatisierungsprogrammen 'Nicht-Informatiker' beteiligt sind, nämlich die Spezialisten für den zu steuernden und zu überwachenden technischen Prozeß, muß die Spezifikationssprache auch für diesen Personenkreis verständlich und in angemessener Zeit erlernbar sein (siehe auch Abschnitt 2.1.3).
In /GOEH81b/ werden die Synchronisations- und Kommunikationsprobleme geschildert, wie sie in Prozeßautomatisierungsprogrammen auftreten. Mit einem Synchronisierungskonzept müssen diese Probleme einfach und übersichtlich beschrieben werden können.

Allgemeine Synchronisationsprobleme

- Probleme des wechselseitigen Ausschlusses
- Reihenfolgeprobleme
  Zwischen den einzelnen Prozessen eines Automatisierungsprogramms können zueinander zeitliche und logische Reihenfolgebeziehungen bestehen.
- daten- und ereignisabhängige Fortsetzungsprobleme
  Ein Prozeß wird in seinem Ablauf solange unterbrochen, bis eine bestimmte Bedingung zutrifft b.z.w. ein erwartetes Ereignis eintrifft.
- Prioritäten und Reaktionszeiten
  Treffen mehrere Ereignisse ein, die für Prozesse auf demselben Prozessor bestimmt sind, so muß geregelt sein, welche Ereignisse zuerst bearbeitet werden (Prozessorvergabe bei Monoprozessoren). Durch entsprechende Prioritäten für die Behandlung von Ereignissen können gewisse Reaktionszeiten für die einzelnen Ereignisse erreicht werden.

Synchronisationsprobleme bei speziellen Automatisierungsprogrammen

In /GOEH81b/ werden vier Modellprozesse beschrieben, an Hand derer Synchronisationsprobleme ermittelt wurden. Diese Synchronisationsprobleme können in die oben genannte Klassifikation eingeordnet werden.

- Synchronisationsaufgaben bei der Betriebsüberwachung
  * Koordinierung der Verwaltung gemeinsamer Betriebsdaten
  * Synchronisierung der Protokollierung
- Synchronisationsaufgaben bei einer direkten digitalen Regelung
  * Pufferung von Protokollierungsaufträgen
  * Synchronisierung des lesenden und schreibenden Zugriffs auf gemeinsame Daten
- Synchronisation bei einer automatischen Geräteprüfung
  * Fortsetzen von Prozessen, wenn alle für die Fortsetzung notwendigen Bedingungen erfüllt sind
  * Zeitliche Überwachung des Eintritts von Ereignissen
- Synchronisation bei der Automatisierung eines Hochregallagers
  * Anstoß bzw. Fortsetzung eines Prozesses, wenn eine der für die Aktivierung notwendige Bedingung erfüllt ist
  * Pufferung von Aufträgen, die von verschiedenen Stellen eintreffen

Bei den bisherigen Anforderungen an eine Spezifikationssprache blieb unberücksichtigt, auf welchem Rechnersystem das zu entwerfende Automatisierungssystem laufen soll. Es gibt Synchronisationskonzepte, die für Monoprozessorsysteme besonders geeignet sind (z.B. Monitore), und solche, die für verteilte Rechensysteme vorteilhaft sind. In Kapitel 4. wird auf diese Problematik näher eingegangen.

3. Synchronisations- und Kommunikationskonzepte

In den folgenden Abschnitten werden einige Spezifikations- und Implementationssprachen vorgestellt, die verschiedene Synchronisations- und Kommunikationskonzepte enthalten.

3.1. Auf gemeinsamen Objekten aufbauende Synchronisations- und Kommunikationskonzepte

Seit Dijkstra 1968 /DIJK68/ die Semaphore vorgeschlagen hatte, entstanden zahlreiche Synchronisationskonzepte für Prozesse mit gemeinsamen Objekten. Nur wenige dieser Konzepte haben Eingang in eine konkrete Spezifikations- oder Programmiersprache gefunden. Eine Beschreibung und ein Vergleich zahlreicher solcher Synchronisationskonzepte findet sich in /BART82/. In den folgenden Abschnitten sollen einige Konzepte vorgestellt werden, die in konkreten Spezifikations- oder Programmiersprachen verwendet werden, also zu einer gewissen praktischen Bedeutung gelangten.

3.1.1. Konzepte in Spezifikationssprachen

3.1.1.1. Das 'Software-Werkzeug' EPOS

EPOS ist ein Spezifikations- und Entwurfssystem für Ingenieure, die Realzeitsysteme projektieren, entwickeln, in Betrieb nehmen, warten und handhaben /GEOH81b/. EPOS unterstützt die Programmstrukturierungsaspekte 'Prozesse' und 'gemeinsame Objekte'.
Es stehen in EPOS sowohl aktions- als auch ressourcenorientierte Synchronisationskonzepte zur Verfügung. Allerdings ist die Kommunikation nur über gemeinsame Objekte möglich. Bei der aktionsorientierten Synchronisation besitzen die Rechenprozesse Informationen bezüglich des Synchronisationsproblems und entscheiden, ob sie eine Ressource benützen können oder nicht. Im folgenden sollen kurz die Synchronisatonskonzepte in EPOS

erläutert werden. Die aktionsorientierten Synchronisierungskonzepte werden verwendet für
- Probleme des wechselseitigen Ausschlusses
- Reihenfolgeprobleme und
- daten- bzw. ereignisabhängige Fortsetzungsprobleme.

Für jeden dieser Problembereiche gibt es in EPOS entsprechende Synchronisationsanweisungen.
Der EXKLUSIV-Operator dient zur Beschreibung von Problemen des wechselseitigen Ausschlusses. Die Operanden des EXKLUSIV-Operators sind Aktionen, die in unterschiedlichen parallelen Rechenprozessen auftreten.

Beispiel:
Spezifikation des wechselseitigen Ausschlusses der Verarbeitungsvorgänge "Meßwertprotokoll" und "Ausfallprotokoll".

```
EXKLUSIV (Meßwertprotokoll, Ausfallprotokoll)
```

Durch diese Anweisung wird auch festgelegt, daß die Aktion "Meßwertprotokoll" gleichzeitig von verschiedenen Rechenprozesssen verwendet wird. Dies zeigt daß EPOS auch bei aktionsorientierten Konzepten von gemeinsamen Objekten ausgeht.

Durch zusätzliche Attribute können auch komplexere Beziehungen spezifiziert werden. Durch das Attribut REENTRANT wird innerhalb eines EXKLUSIV-Ausdrucks die gleichzeitige mehrfache Ausführung einer Aktion ermöglicht.

Beispiel:
Über eine schmale Brücke kann gleichzeitig nur in einer Fahrtrichtung gefahren werden. Allerdings können sich mehrere Fahrzeuge, die in dieselbe Richtung fahren, auf der Brücke befinden.
Dieses Synchronisationsproblem wird wie folgt formuliert:

```
EXKLUSIV (Überqueren-Fahrtrichtung1 REENTRANT,
          Überqueren-Fahrtrichtung2 REENTRANT).
```

Der SEQUENCE- und der CYCLE-Operator dienen dazu, Reihenfolgebeziehungen zu beschreiben, die während des Ablaufs eingehalten werden müssen. Als Operanden treten wiederum Aktionen unterschiedlicher paralleler Prozesse auf.
Mit dem Operator SEQUENCE werden Reihenfolgeprobleme beschrieben, die nur einmal erfüllt sein müssen. Mit dem Operator CYCLE werden Reihenfolgebeziehungen beschrieben, die während des gesamten Ablaufs eingehalten werden müssen.

Beispiele:

a) Von einem Puffer kann erst gelesen werden, wenn mindestens einmal etwas hineingeschrieben wurde.

```
SEQUENCE (Puffer-Schreiben, Puffer-Lesen)
```

b) Die Verarbeitung von Meßwerten kann wie folgt spezifiziert werden:

```
CYCLE (Analogwert_einlesen, Umrechnung, Grenzwertprüfung)
```

Mit Hilfe der Warteanweisungen WAIT UNTIL bzw. WAIT FOR ist es möglich, zu formulieren, daß während des Ablaufs an einer bestimmten Stelle in einem Rechenprozeß gewartet werden muß, bis ein Ereignis eingetreten ist.

Beispiele:

a) Spezifikation eines datenabhängigen Fortsetzungsproblems

```
WAIT UNTIL   Puffer-nicht-voll
   THEN    Einfüllen
WAITEND
```

b) Spezifikation eines ereignisabhängigen Fortsetzungsproblems

```
WAIT FOR     Siedemeldung
     THEN    Brenner abschalten
WAITEND
```

Die Spezifikationsangaben für die aktionsorientierte Synchronisierung werden mit Ausnahme von daten- bzw. ereignisorientierten Fortsetzungsproblemen im sogenannten SYNCHRO-Teil zu einer gemeinsamen Oberaktion zusammengefaßt. Die konjunktive Verknüpfung der Einzelspezifikationen beschreibt die notwendige Einschränkung des freien parallelen Ablaufs.

Zur Spezifikation der ressourcenorientierten Synchronisierung werden erweiterte Pfadausdrücke verwendet. Dabei wird beschrieben, wie und von welchen Aktionen Ressourcen benützt werden können, unabhängig davon, zu welchen Rechenprozessen die einzelnen Aktionen gehören.

Als Operatoren für die Pfadausdrücke stehen zur Verfügung:

A->B	A muß vor B ausgeführt werden
A+B	entweder A oder B werden ausgeführt (exklusiv)
A*	A wird wiederholt ausgeführt
A//B	A und B werden parallel ausgeführt
A REENTRANT	A kann von verschiedenen Rechenprozessen gleichzeitig benützt werden
A PRIO	die Ausführung von A wird bevorzugt behandelt.

In EPOS werden Ressourcen vom Typ DATA und vom Typ INTERFACE unterschieden. In den Ressourcen werden im OPERATION-Teil die erlaubten Ausführungsreihenfolgen durch Pfadausdrücke beschrieben. Die Operationen zum Bilden von Pfadausdrücken wurden oben beschrieben.

Beispiel:
Spezifikation des Leser/Schreiber-Problems mit Priorität für die Sehreiboperationen.

```
DATA Puffer
   .
   .
OPERATION; Schreiben    (Lesen REENTRANT + Schreiben PRIO)*
   .
   .
DATEND
```

Für die Beschreibungsmittel zur Synchronisation ermöglicht EPOS eine rechnergestützte bei der Analyse und Dokumentation der Spezifikation (siehe /GOEH81b/).

Diskussion:

Bei EPOS wurde bewußt darauf verzichtet, Darstellungsmittel für alle denkbaren Synchronisationsprobleme und für alle möglichen Strukturen von parallelen Rechenprozessen anzugeben. Dies wird vor allem bei den Pfadausdrücken deutlich, mit denen bekanntlich nur reguläre Ausdrücke /MÜLL77/ möglich sind. Hier müßten bedingte Pfadausdrücke eingeführt werden, um allgemeine Probleme beschreiben zu können.
Die Konstrukteure von EPOS behaupten, daß die verwendeten Synchronisationsmittel problemgerecht (Prozeßautomatisierungssystem) und benutzerfreundlich sind. Ein wesentlicher Vorteil von EPOS ist, daß es Rechnerunterstützungen gibt, die eine bequeme Analyse des Synchronisationsverhaltens von Programmen ermöglichen.

3.1.1.2 Spezifikationskonzept nach Keramidis

Keramidis /KERA82/ läßt als Strukturierungsaspekt nur gemeinsame Objekte zu. In diesem Konzept gibt es deshalb nur resourcenorientierte Synchronisationsmittel.
Bei dieser Spezifikationsmethode werden Moduln als abstrakte Datentypen aufgefaßt. Als abstrakter Datentyp wird dabei der Prototyp einer häufig gebrauchten Datenstruktur samt den zuge-

hörigen Operationen verstanden /KKST79/. In /KERA82/ werden durch den Abstraktionsmechanismus MODULE aus bereits vorhandenen Datentypen neue erzeugt.
In diesem Konzept wird die Synchronisation unabhängig von den ausführenden Prozessen eines parallelen Programms beschrieben. Da es in diesem Konzept keine Prozeßnamen gibt, muß die Synchronisation unabhängig vom Prozeßnamen beschrieben werden können. Dazu wird der Begriff der Aktivität eingeführt. Eine Aktivität beginnt mit dem Aufruf einer Operation durch einen Prozeß zu existieren und endet mit der Beendigung dieser Operation. Rufen zwei Prozesse gleichzeitig dieselbe oder verschiedene Operationen auf, existieren auch zwei Aktivitäten.

Eine Aktivität kann in die in Bild 3,1 gezeigten Tätigkeitszustände versetzt werden.

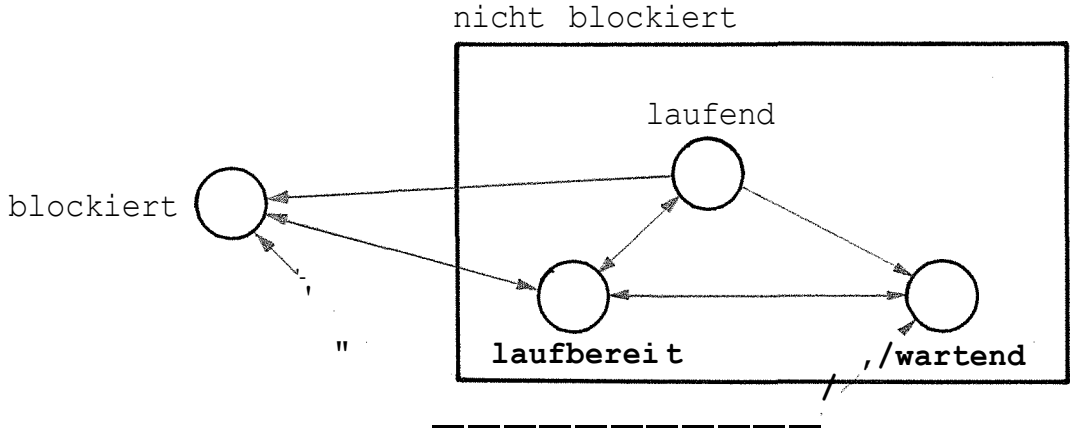


Bild 3,1 : Tätigkeitszustände einer Aktivität

Wann eine Aktivität sich in welchen Tätigkeitszuständen befindet, wird weiter unten erläutert.

Ein abstrakter Datentyp wird in /KERA82/ nach folgendem Schema aufgebaut:

```
MODULE Name (Parameterliste)
SPECIFICATION
PARAMETERS
 .
SYN
 .
 .
OPERATIONS
 .
 .
IMPLEMENTATION
 .
 .
MODULEND
```

Im weiteren wird auf den Implementierungsteil nicht weiter eingegangen. In ihm werden die abstrakten Objekte durch konkrete Objekte repräsentiert und abstrakte Operationen durch Prozeduren realisiert, die die konkreten Objekte manipulieren. Der Spezifikationsteil beschreibt die Schnittstelle eines Objekts zur Außenwelt. Er besteht aus den Abschnitten PARAMETERS, SYN und OPERATIONS.
Im SYN-Teil werden die Verträglichkeits- und Prioritätseigenschaften der Aktivitäten festgelegt, die die Operationen ausführen.
Mit der Verträglichkeitsrelation VTGL kann man definieren, in welchen Zuständen Operationen verträglich sind. Die VTGL-Relation wird wie folgt interpretiert: Wird zu einem Zeitpunkt die Operation op1 ausgeführt, dann werden alle Aktivitäten, die gleichzeitig Operationen in demselben Modul ausführen möchten, in den Zustand 'wartend' versetzt, wenn diese Operationen im momentanen Zustand nicht mit der Operation op1 verträglich sind.
Die Prioritätseigenschaft definiert - im allgemeinen zustandsabhängige - Rangordnungen unter den Aktivitäten, die sich

gegenseitig ausschließende Operationen ausführen möchten. Die Prioritätsrelation PRIO hat folgende Bedeutung: Eine bereite (genaue Definition von 'bereit' siehe /KERA82/) Aktivität A1, die eine Operation op1 ausführen möchte, kann nicht in den Zustand 'laufend' (siehe ebenfalls /KERA82/) versetzt werden, wenn es eine andere Aktivität A2 gibt, die eine Operation ausführen möchte, der im momentanen Zustand eine höhere Priorität zugeordnet ist als der Operation die von A1 ausgeführt werden soll. Prioritätenrelationen werden vorwiegend bei der Beschreibung von Schedule-Problemen eingesetzt. Sind für die Beschreibung der Synchronisation Prozeßnamen nötig, so können diese in diesem Modell durch Parameter angegeben werden. Denn sowohl die Verträglichkeits- als auch die Prioritätsrelation kann von Parametern der abstrakten Operationen abhängen.

Im OPERATION-Teil werden die für dieses Objekt erlaubten Operationen definiert. Um die Semantik der Operationen zu beschreiben, wird die sogenannte Prädikatentransformation verwendet (siehe /GRIE81 /) . Da auf diese Technik der Semantikdefinition später noch näher eingegangen wird, soll hier nur das erläutert werden, was zum Verstehen dieser Spezifikationsmethode nötig ist.

Eine abstrakte Operation op auf ein abstraktes Objekt y wird durch Angabe der Prädikate P(y) und Q(y) nach dem Muster {P(y)} op(y) {Q(y)} spezifiziert. Die Operation op(y) kann nur ausgeführt werden, wenn für den momentanen Zustand von y das Prädikat P(y) gilt. Nach der Ausführung von op(y) gilt das Prädikat Q(y). Aus schreibtechnischen Gründen wird statt {P} Operation {Q} die Notation NBL: P EFFECTS: Q verwendet. Das Prädikat P gibt die Nichtblockierungsbedingung des die Operation aufrufenden Prozesses an. Ruft ein Prozeß eine Operation auf und ist die NBL-Bedingung nicht erfüllt, so wird er in den Zustand "blockiert" versetzt.

Ist die NBL-Bedingung erfüllt, so sind die Verträglichkeits- und Prioritätsbedingungen dafür entscheidend, ob die die Operation ausführende Aktivität in den Tätigkeitszustand 'laufend' versetzt wird.

Im PARAMETERS-Teil werden die Typen der in der Spezifikation vorkommenden Größen festgelegt. Außerdem wird in diesem Teil

auch die abstrakte Repräsentation von Objekten eines Typs festgelegt. Die abstrakte Repäsentation entspricht einer mathematischen Repräsentation der abstrakten Objekte vor der Einführung des abstrakten Datentyps. Im PARAMETERS-Teil wird die abstrakte Repräsentation deswegen definiert, weil in den NBL- und EFFECTS-Teilen einer Operation Aussagen über Objekte des zu definierenden Typs gemacht werden.

Beispiel: Abstrakter Datentyp 'semaphore'

```
MODULEsemaphore (1: integer where 12! 0)
   SPECIFICATION

   PARAMETERS
     k: INTEGER WHERE k > 0
     s: SEMAPHORE AR AS INTEGER; INIT: s = 1

   OPERATIONS
     wait (s,k)
     NBL: s 2 k
     EFFECTS: s = 's-k

     signal (s,k)
     NBL: true
     EFFECTS: s = 's+k
MODULEND
```

Der Datentyp 'semaphore' wird abstrakt repräsentiert durch eine Variable vom Typ Integer (AR-Konstrukt). Mit 's' im EFFECTS-Teil einer Operation ist der Wert des Objekts vor der Ausführung der Operation gemeint.
Durch die Deklaration s1: semaphore (3) bzw. s2: sernaphore (10) kann man zwei Semaphore mit den Anfangswerten 3 und 10 deklarieren. Andererseits kann man aber auch wait (s1,5) oder wait(s1,2) aufrufen.

Diskussion

Das in /KERA82/ beschriebene Konzept berücksichtigt nur den Strukturierungsaspekt der gemeinsamen Objekte. Aus diesem Grund ist die Anwendung dieser Spezifikationsmethode nicht für alle Anwendungsbereiche gleich vorteilhaft. Dadurch, daß es nur - zwar sehr mächtige - ressourcenorientierte Synchronisationswerkzeuge gibt, lassen sich bestimmte Synchronisationsprobleme, wie sie in der Prozeßautomatisierung vorkommen, nur schwer beschreiben. Hängt z.B. die Ausführung eines Rechenprozesses vom Eintreten eines bestimmten Ereignisses ab, so läßt sich dies in /KERA82/ nur sehr aufwendig darstellen, wogegen Probleme, die bei Betriebssystemen auftreten (Verwendung gemeinsamer Betriebsmittel), mit dieser Methode sehr elegant beschrieben werden können. Allerdings kann es hier ähnliche Probleme geben, wie sie in /KEED78/ für Monitore beschrieben werden. So wird ein Prozeß immer blockiert, wenn z.B. ein von ihm gewünschtes Betriebsmittel nicht verfügbar ist. Dabei wäre es manchmal durchaus wünschenswert, einen Prozeß nicht zu blockieren, sondern vielmehr den Zugriff zu verweigern und den Prozeß zu informieren, wenn ein Betriebsmittel frei geworden ist /KEED78/.

Probleme wie bei Monitoren /KEED78/ gibt es auch bei Stapelbetriebssystemen, wenn ein Prozeß zu einem bestimmten Zeitpunkt mehrere Betriebsmittel anfordert. Denn im allgemeinen werden zumindest verschiedene Betriebsmittel von verschiedenen Moduln verwaltet. Jedes einzelne Betriebsmittel aus der gewünschten Menge wird nacheinander angefordert. Durch diese einzig mögliche Vorgehensweise bei der Belegung von Betriebsmitteln entsteht ein beträchtliches Verklemmungsrisiko, denn die Verklemmungsbedingung 'Die gewünschten Betriebsmittel werden einzeln angefordert' läßt sich nicht mehr ausschließen.

3.1 .2. Konzepte in Programmiersprachen

3.1 .2.1. Semaphore

Semaphore /DIJK68/ sind das erste Konzept, das vorgeschlagen wurde, um Synchronisationsprobleme zu lösen. Semaphore sind ein spezieller Datentyp.
Jedem Semaphor S wird eine Integer-Variable und eine Warteschlange zugeordnet. Auf einem Semaphor sind die P- und V-Operationen definiert.
Die P-Operation, angewendet auf dem Semaphor S bedeutet, daß der Wert des Semaphors um eins vermindert wird, falls S vor der Ausführung der Operation einen Wert größer als Null hatte. Gilt S = 0 vor der Ausführung der Operation, so wird der aufrufende Prozeß in die Warteschlange eingereiht.
Wird die V-Operation ausgeführt und ist die Warteschlange nicht leer, so wird ein Prozeß daraus entfernt. Falls die Warteschlange leer ist, wird die entsprechende Integervariable um eins erhöht.
Soll die V- bzw. P-Operation auf dem Semaphor S ausgeführt werden, wird dies mit V(s) bzw. P(s) notiert. Das folgende Beispiel zeigt das allseits bekannte Erzeuger-Verbraucher-Problem mit beschränktem (hier einelementigen) Puffer /RICH77/.

```
          SEMAPHORE  voll=0,  leer=1,  zustand=1

Erzeuger-Prozeß                        Verbraucher-Prozeß

BEGIN                                  BEGIN,
   ep:  Erzeuge Wart                    vp:  P(voll)
        P(leer)                              P(Zustand)
        P(Zustand)                           Entnehme Ware aus Puffer
        Transportiere Ware nach Puffer       V(Zustand)
        V(Zustand)                           V(leer)
        V(voll)                              Verbrauche Ware
        GOTO ep                              GOTO vp
END                                    END
```

Diskussion

Das Semaphorkonzept hat eine sehr niedrige Abstraktionsebene und ist direkt über der Hardware angesiedelt /KERA82/. Allerdings läßt es sich sehr einfach implementieren (siehe z.B. /KUMM81 /).
Eine Programmiersprache, in die das Semaphorekonzept integriert wurde, ist z.B. PEARL (siehe Abschnitt 5.4.1.).

3.1 .2.2. Monitore

Das Monitorkonzept wurde von Hoare /HOAR74/ entwickelt. Nach /HOAR74/ ist ein Monitor wie folgt aufgebaut:

```
monitor name =
    Deklaration der lokalen Daten:
          .
          .
          .
    Prozeduren;
       procedure name (formale Parameter)
          .
          .
       procedure name (formale Parameter)

        Initialisierung:
          BEGIN
            .
            .
          END
```

Ein Monitor enthält eine Menge von Prozeduren und lokalen Daten. Die Daten werden nach der Erzeugung eines Monitors gemäß dem Initialisierungsteil vorbesetzt. Außerhalb eines Monitors sind die lokalen Daten verborgen. Sie können nur mit

Hilfe der Prozeduren manipuliert werden.
Zu jedem Zeitpunkt kann höchstens eine Prozedur eines Monitors ausgeführt werden.
Innerhalb einer Prozedur kann sich ein Prozeß an einer sogenannten 'condition'-Variablen blockieren. Dazu ruft ein Prozeß die Operation WAIT auf. Beim Aufruf dieser Funktion wird der Name der entsprechenden condition-Variablen mit angegeben, so daß der vollständige Aufruf wie folgt aussieht:

name. WAIT.

Nach dem Aufruf von WAIT wird ein Monitor wieder freigegeben, d.h. ein anderer Prozeß kann seine gewünschte Monitorfunktion ausführen.
Das Gegenstück zur Operation WAIT ist die Operation SIGNAL. Ruft ein Prozeß P1 bei der Ausführung einer Monitorprozedur die Operation 'Name.SIGNAL' auf und ist die Menge der an 'name' blockierten Prozesse leer, so hat dies keinerlei Wirkung. Wenn allerdings an der condition-Variablen 'name' einer oder mehrere Prozesse blockiert sind (durch einen 'name.WAIT' Aufruf), so wird ein Prozeß P2 deblockiert. Jetzt muß allerdings der Prozeß P1 solange warten, bis P2 den Monitor wieder freigibt. Erst dann kann der Prozeß P1 weitermachen.

Diskussion:

Monitore sind ein ressourcenorientiertes Synchronisationskonzept. Deshalb gelten für sie bezüglich der Prozeßautomatisierung ähnliche Einwände, wie sie in Abschnitt 3.1.1.2 geschildert wurden. Sie sollen aus diesem Grund nicht nochmals dargestellt werden.
Bei der Verwendung von Monitoren zur Implementierung von Betriebssystemen verschärfen sich die ebenfalls in Abschnitt 3.1.1.2 angeführten Einwände, da Monitore für die Verwendung der Monitorfunktionen nur über einen sehr starren Synchronisationsmechanismus verfügen. Es gelten die in /KEED78/ aufgeführten und in Abschnitt 3.1.1.2 kurz dargestellten Einwände uneingeschränkt. Weitere Probleme bei der Verwendung von Moni-

toren, wie sie in Concurrent Pascal /HANS77/ definiert sind, werden in /PIEP81/ erläutert (z.B. Schachtelung von Monitoren, Überstrukturierung) und sollen nicht weiter dargestellt werden.

3.2. Nachrichtenorientierte Synchronisations- und Kommunikationskonzepte

Da in dieser Arbeit ein Synchronisationskonzept vorgestellt wird, das auf Botschaftsmechanismen beruht, sollen zunächst zahlreiche aus der Literatur bekannte Konzepte erläutert werden. Dies soll vor allem dazu dienen, das in Kapitel 5 vorgeschlagene Konzept besser einorden zu können.
Nach /LANE78/ sind für Systeme, deren Prozesse sich mit Botschaftskonzepten synchronisieren, folgende Punkte charakteristisch:

- Die Synchronisation des Zugriffs auf gemeinsame Ressourcen wird mit Warteschlangen implementiert, die den einzelnen Prozessen, die die Ressourcen verwalten, fest zugeordnet sind.
- Daten, die von mehreren Prozessen bearbeitet werden, werden ebenfalls durch Prozesse verwaltet. Diese erhalten die 'Zugriffsaufträge' durch Botschaften.
- Periphere Geräte werden als Prozesse betrachtet.
- Den Prozessen sind die Prioritäten fest zugeordnet.
- Prozesse arbeiten mit einer geringen Anzahl von Nachrichten und beenden zuerst die Bearbeitung einer Nachricht, bevor sie die nächste aus der Warteschlange nehmen.
- Die Anzahl der Prozesse ist statisch.

Es wurden verschiedene Konzepte zur Darstellung paralleler Prozesse auf der Basis von Botschaftssystemen vorgeschlagen. (/HOAR78/, /HANS78/, /ADA79/, /MAYE80/, /BOPS81/, /BOCH79/, /COOK80/, /FELD79/, /SILB79/)
Diese Konzepte unterscheiden sich vor allem in folgenden Punkten:

- Blockiert- und Nichtblockiertbedingungen /GENT81/
- Adreß- und Absenderangaben /GENT81/
- Botschaftsformate /GENT81/
- Verhalten bei Übertragungsfehlern /GENT81/
- Abstraktionsgrad

Im folgenden sollen einige der vorgeschlagenen Botschaftskon-

zepte erläutert und diskutiert werden. Die einzelnen Konzepte werden nicht untereinander verglichen, dazu sei auf die entsprechende Literatur verwiesen (/STROBO/, /ROBE81/, /STAU82/, /GENT81/, /WELS79/, /WELS81/, /KLEB83/, /STOT82/ u.s.w.)

3.2.1. Konzepte in Spezifikationssprachen

3,2.1 .1. Das Spezifikationskonzept nach G. Bochmann

Das Konzept von Bochmann /BOCH79/ wurde für die Spezifikation von DFÜ-Software entwickelt. Es dient vor allem zur Beschreibung von Übertragungsprozeduren (Protokollen). Das Konzept von Bochmann ist die Erweiterung eines Konzepts, das zur Spezifikation von parallelen Programmen dient /KELL76/.
Das Modell von Bochmann besteht im wesentlichen aus einem Petri-Netz mit Transitionen und Knoten. Die Petri-Netze werden durch eine Menge X von Variablen ergänzt. Die Aktionen At verändern Variable aus der Menge X. Der Zustand des Systems ist durch die Verteilung der Marken und die Werte der Variablen bestimmt. Eine Transition t kann dann ausgeführt werden, wenn in jeder Eingangsstelle mindestens eine Marke ist (Standardregel bei Petrinetzen) und die Übergangsbedingung Pt erfüllt ist. Feuert eine Transition, so wird die zugeordnete Aktion At ausgeführt und die Marken werden entsprechend den Regeln für die Petri-Netze neu verteilt. Ein System S (z.B. parallele Programme) kann aus mehreren Teilsystemen S1 ,S2, ...Sn bestehen. Jedes dieser Teilsysteme wird wie oben geschildert beschrieben.
Die Menge der Variablen eines Teilsystems Si wird mit Xi bezeichnet. Xi ist die Menge der lokalen Variablen eines Systems Si. Die Übergangsbedingungen Pi hängen nur von den lokalen Variablen Xi ab und die Aktionen At verändern nur die lokalen Variablen Xi. Zur Kommunikation der einzelnen Teilsysteme miteinander enthält jedes Teilsystem eine Menge von 'von außen anstoßbaren Aktionen' (distantly initiated actions). Ähnlich den lokalen Aktionen können sie den Wert der lokalen Variablen verändern. 'Distantly initiated actions' sind nicht

mit dem Feuern einer Transition verbunden und werden eine endliche Zeit nach ihrem Anstoß ausgeführt. Der Anstoß erfolgt innerhalb der lokalen Aktion eines anderen Teilsystems. Das anstoßende Teilsystem kann die 'distantly initiated action' zur Ausführung mit Parametern versorgen (nur Konstante erlaubt). Alle Aktionen innerhalb eines Teilsystems laufen unter gegenseitigem Ausschluß.
Das Anstoßen einer 'distantly initiated action' wird beschrieben durch die Anweisung:

INITIATIVE name, p1, ... pk

Mit 'name' wird die angestoßene Aktion und mit 'p1 '....'pk' werden die übergebenen Parameterwerte bezeichnet. Die anstoßende Aktion wartet nicht auf das Ausführungsende der angestoßenen Aktion. Die Ausführungsreihenfolge von 'distantly initiated actions' ist im allgemeinen verschieden von der Anstoßreihenfolge.
Die Darstellung eines Teilsystems besteht aus folgenden Teilen:

- Einern Zustandsdiagramm, das anzeigt, welche Zustandsübergänge möglich sind,
- den lokalen Daten und
- einer Tabelle, die wiedergibt, welche Prädikate welchem Zustandsübergang zugeordnet sind und welche Aktionen dann ausgeführt werden. Diese Tabelle enthält außerdem noch die Definition der 'distantly initiated actions', die in diesem Teilsystem ausgelöst werden können.

Die Beschreibung eines Gesamtsystems besteht aus einer Menge solche Teilsystembeschreibungen.
Mit dem Spezifikationskonzept nach G. Bochmann ist es prinzipiell möglich, eine Beschreibung des Verhaltens des Gesamtsystems zu erstellen. Dadurch wird es möglich, das Verhalten des Gesamtsystems bezüglich Zyklen, Verklemmungen und Lebendigkeit sowie partielle und vollständige Korrektheit zu untersuchen. Der Beweis der partiellen Korrektheit eines Systems besteht darin, herauszufinden, ob und unter welchen Umständen das

Senderteilsystem weiß, ob alle Daten, die vom Benutzer erhalten wurden, korrekt übertragen wurden und in der gleichen Reihenfolge beim Empfänger ankommen, wie sie der Benutzer an den Sender gegeben hat. Dieser Sachverhalt wird bei Bachmann durch das folgende Prädikat ausgedrückt:

p1: Producer Sequence = Consumer sequence.

Ein Sender ist in einem vollständigen Zustand, wenn in diesem Zustand p1 gilt.
Partielle Korrektheit eines Systems bedeutet dann, daß ein vollständiger Senderzustand besteht und vollständige Korrektheit bedeutet, daß nach einer endlichen Zeit immer ein vollständiger Zustand erreicht wird.

Beispiel:
Das folgende Beispiel ist die Beschreibung eines einfachen DFÜ-Protokolls. Das System besteht aus zwei Teilsystemen: dem Sender und dem Empfänger.

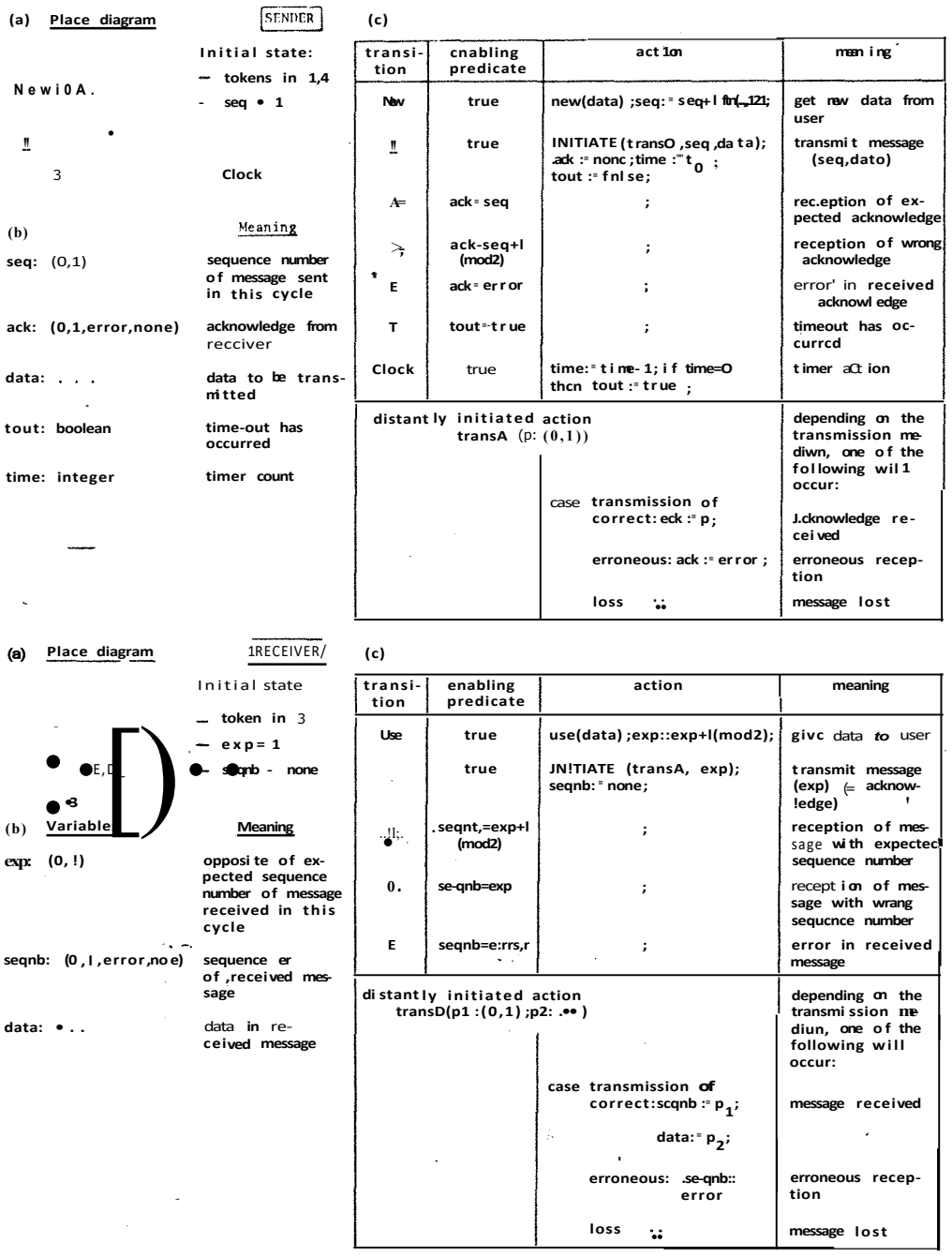


Bild 3.2. : Ein einfaches DFÜ-Protokoll /BOCH79/

In Bild 3.3 wird das Verhalten des Gesamtsystems dargestellt. Der Beweis für die Korrektheit dieses Systems findet sich in /BOCH79/ von S. 197 bis S. 198.

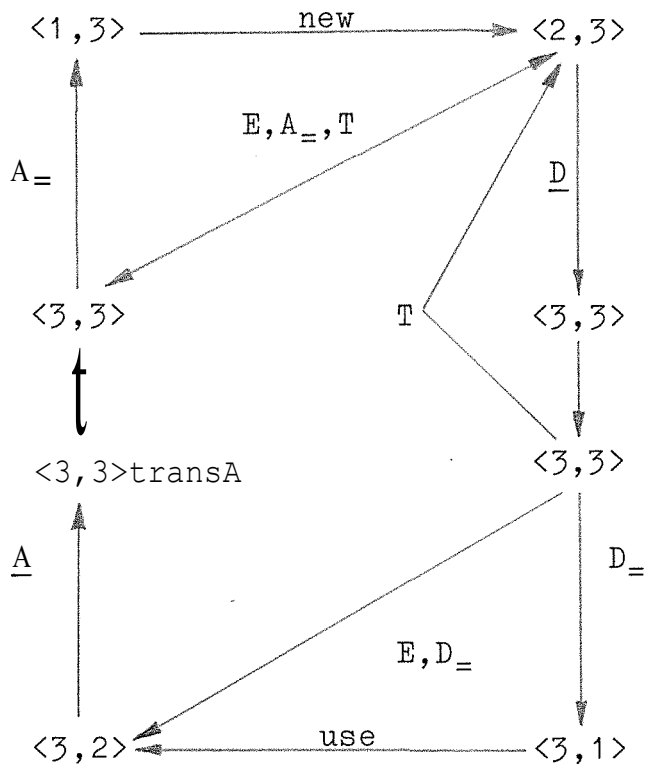


Bild 3.3: Gesamtverhalten des Systems aus Bild 3.2 /BOCH79/

Diskussion

Bei den Teilsystemen wird angegeben, welche 'distantly initiated actions' auf die lokalen Daten des Teilsystems wirken. Es wird jedoch nicht beschrieben, von welchen Teilsystemen sie unter welchen Umständen angestoßen werden. Um das Verhalten eines Teilsystems zu erkennen, muß jedes Teilsystem in die Betrachtung einbezogen werden, welches 'distantly initiated actions' beim betrachteten Teilsystem auslösen kann.
Die 'lokalen Daten' eines Teilsystems können zu beliebigen Zeitpunkten durch andere Teilsysteme verändert werde Einern Teilsystem ist es nicht möglich zu verhindern, daß bestimmte 'distantly initiated actions' ausgeführt werden. Dadurch ist es schwierig, die Konsistenz von Daten zu gewährleisten. Außerdem können dadurch keinerlei Schutzaspekte definiert werden.

Ein Teilsystem muß aktiv an Hand der Daten prüfen, ob eine bestimmte 'distantly initiated action' ausgeführt wurde.
Die Reihenfolge der Aktionen innerhalb eines Teilsystems gehört nicht zur Beschreibung eines Teilsystems wie bei /KEMA79/, sondern wird bestimmt von dem Gesamtverhalten all der Teilsysteme, die 'distantly initiated actions' in diesem Teilsystem anstoßen dürfen, d.h. die auslösenden Teilsysteme müssen über den Zustand des anzusprech nden Teilsystems informiert sein, um die Konsistenz der Daten des Zielsystems zu gewährleisten. Dies ist nicht vereinbar mit dem Modulkonzept.
(Die Spezifikation von HDLC mit dem Konzept von Bochmann /BOCH77/ umfaßt z.B. 14 Teilsysteme, welche durch diverse 'distantly initiated actions' verbunden sind. Da lokale Daten auch durch entsprechend komplexe Prozeduren verändert werden und auch in den Prozeduren 'distantly initiated actions' anstoßen werden, ist es nicht leicht, den funktionalen Ablauf von HDLC zu verstehen).
Überdies erscheint es überflüssig, sich auf Petri-Netze zu beziehen, da die Methoden zur Untersuchung Standard-Petri-Netzen wegen der eingeführten Prädikate nicht mehr anwendbar sind.
Die Schaltbedingung für die Standard-Petri-Netze kann in das Prädikat mit einbezogen werden. Dazu muß jeder Stelle ein Zähler zugeordnet werden, der anzeigt, wieviel Marken sich in einer Stelle angesammelt haben. Eine Transition kann dann schalten, wenn für die Eingangsstellen gilt, daß die Zähler größer Null sind und das der Transition zugeordnete Prädikat erfüllt ist. Die Aktion wird um das Erhöhen der Zähler aller Ausgangsstellen um eins ergänzt /HOFM81/.

3.2.1 .2 Die Spezifikationssprache SDL

Die Sprache SDL wurde von der CCITT (der Vereinigung aller Post- und Telegraphenverwaltungen) entwickelt. SDL dient zum Entwurf von programmgesteuerten Telefonvermittlungssystemen (stored programm controlled = SPC). SDL ist Teil einer Sprachenreihe, die die Erstellung eines SPC-Systems von der Anforderungsdefinition bis zum Einsatz unterstützen soll. In Bild

3.4 werden die Phasen bei der Erstellung eines SPC-Systems gezeigt, wie sie sich aus der Sicht der CCITT darstellen. In diesem Bild sind auch die beiden anderen Sprachen dieser Sprachenreihe mit ihren Bezügen zu den einzelnen Entwurfsphasen eingezeichnet (CHILL, MML). Aus diesem Bild geht auch hervor, daß SDL für den Systementwurf verwendet wird.

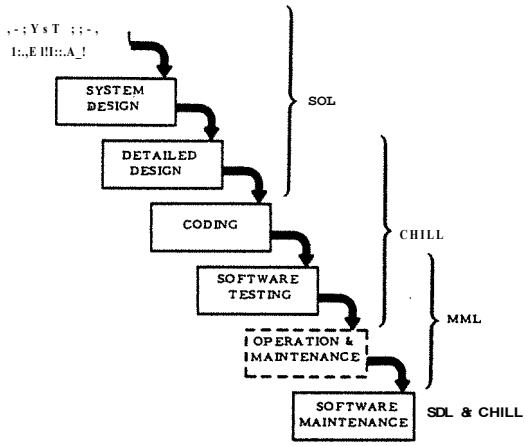


Bild 3.4 : Die Sprachen der CCITT zum Erstellen von programmgesteuerten Vermittlungssystemen /CARR82/

Ein in SDL /ROSA82/ beschriebenes Programm ist in einzelne Prozesse strukturiert. Prozesse sind in SDL die einzige Möglichkeit zur Strukturierung von Programmen. In SDL ist ein Prozeß ein Objekt, das die logische Funktion ausführt. Jeder Prozeß wird dabei als ein endlicher Zustandsautomat betrachtet, der durch einen geschlossenen gerichteten Graphen dargestellt werden kann. (siehe Bild 3.5)

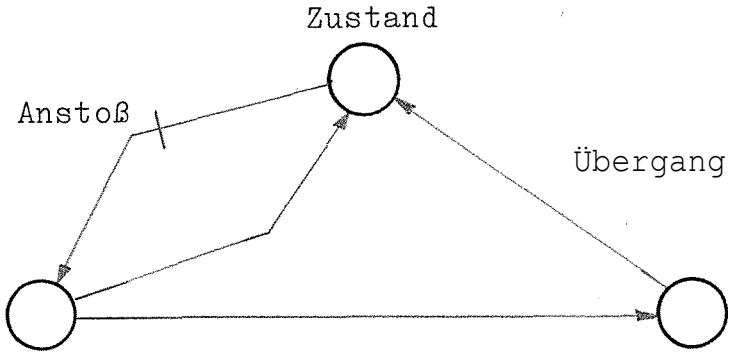


Bild 3.5 : Ein endlicher Zustandsautomat, dargestellt als gerichteter geschlossener Graph /ROSA82/.

Jeder Prozeß hat die Eigenschaft, daß jeder Zustand durch eine Reihe von entsprechenden Übergängen von jedem Zustand aus erreicht werden kann.
Prozesse untereinander können nur durch das Senden und Empfangen von sogenannten Signalen kommuniziern. Dieses Signalkonzept dient sowohl zur Synchronisation von Prozessen als auch zum Datenaustausch zwischen diesen.
Ein SDL-Prozeß wird definiert als ein Objekt, das entweder in einem Zustand auf eine Eingabe (Empfangen eines Signals) wartet, oder sich in einem Übergang befindet. In einem Zustand wartet ein Prozeß alternativ auf verschiedene Signale. Ein Prozeß bleibt in einem Zustand solange blockiert, bis eines der erwarteten Signale eintrifft.
Der Übergang ist eine Folge von Aktionen, die ausgeführt werden, wenn der Prozeß den entsprechenden Übergang ausführt. Bei einem Übergang sind drei Arten von Aktionen erlaubt:

- Ausgaben
  Dabei wird ein Signal generiert, das wieder als Eingabe dienen kann.
- Entscheidungen
  Diese Aktionsart entspricht der Fallunterscheidung in den gängigen Programmiersprachen. Dabei kann einer von mehreren Pfaden für den Übergang ausgewählt werden.
- Aufgaben
  Dies sind alle Aktionen, die weder eine Ausgabe noch eine Entscheidung sind.

Sendet ein Prozeß ein Signal durch eine Ausgabeanweisung, so ist dieses Signal immer an einen bestimmten Prozeß adressiert. Erreicht ein Signal den Zielprozeß, so wird es zurückgestellt bis dieser Prozeß wieder in einen Zustand gelangt (Übergänge in SDL sind nicht zeitlos), denn ein Prozeß kann nur in einem Zustand Signale annehmen. In jedem Zustand ist angegeben, welche Signale hier angenommen werden können. Befindet sich ein Prozeß in einem Zustand, so wird diesem eines der wartenden Signale angeboten. Gibt es keine Signale für einen Prozeß in einem Zustand, so wird er bis zum Eintreffen von entsprechenden Signalen blockiert. Die wartenden Signale werden einem Prozeß in der Ankunftsreihenfolge angeboten. Erlaubt ein angebotenes Signal einen Übergang, so wird es angenommen und der Prozeß führt den entsprechenden Übergang aus. Ist durch ein angebotenes Signal kein Übergang möglich, wird es verworfen und das nächste Signal wird angeboten.

Um zu verhindern, daß ein Signal verworfen wird, gibt es in SDL eine Sicherungsanweisung (save-concept). Signale, auf die durch eine save-Anweisung Bezug genommen wird, werden einem Prozeß nicht zum Empfangen angeboten, sondern werden aufgehoben, bis der Prozeß den nächsten Zustand erreicht. Für die Repräsentation von SDL-Programmen gibt es nach CCITT mehrere Möglichkeiten. Hier soll nur die graphische Darstellung erläutert werden.

Ein SDL-Graph besteht aus einer Anzahl von graphischen Symbolen. Diese Symbole entsprechen den verschiedenen Aktionen, den Zuständen, den Empfangs- und den Rettanweisungen. Bild 3.6 zeigt die in SDL verwendeten Symbole.

Das Symbol für einen Zustand enthält den Zustandsnamen. Gibt es in einem Prozeß mehrere Zustände mit gleichen Namen, so handelt es sich immer um denselben Zustand. Dies dient dazu, den Graphen zu vereinfachen.

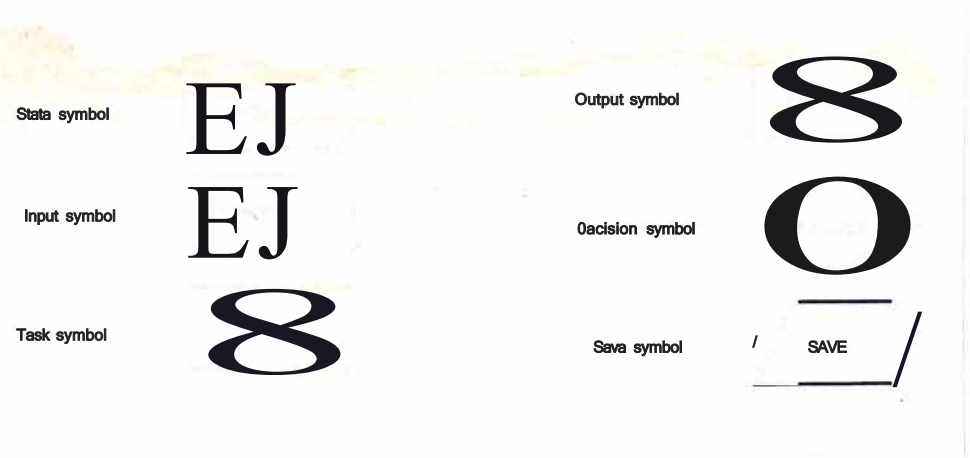


Bild 3.6 : Graphische Symbole von SDL /ROSA82/

Die Symbole für die Eingaben, Ausgaben und Sicherungen enthalten die Namen der Signale, auf die sich die entsprechenden Anweisungen beziehen. Analog sind die Symbole für Entscheidungen und Aufgaben beschriftet.
Werden diese Symbole in einem Diagramm verbunden, müssen folgende Regeln beachtet werden:

- Einern Zustandssymbol dürfen nur Eingabesymbole oder Eingabe- und Sicherungssymbole folgen.
- Jedes Eingabe- und Sicherungssymbol folgt nur auf genau ein Zustandssymbol.
- Jedem Eingabesymbol folgt nur genau ein Symbol. Dieses Symbol darf kein Eingabe- oder Sicherungssymbol sein.
- Jedem Aufgabensymbol darf nur genau ein Symbol folgen und dies darf kein Eingabe- oder Sicherungssymbol sein.
- Einern Entscheidungssymbol müssen ein oder mehrere Symbole folgen. Diese Folgesymbole dürfen keine Eingabe- oder Sicherungssymbole sein.
- Einern SichBrungssymbol darf kein Symbol folgen.

Bild 3.7 zeigt ein Beispiel für ein SDL-Diagramm. Es handelt sich dabei um einen Ausschnitt aus dem Prozeß, der die Anrufbehandlung einer Telefonvermittlung ausführt.

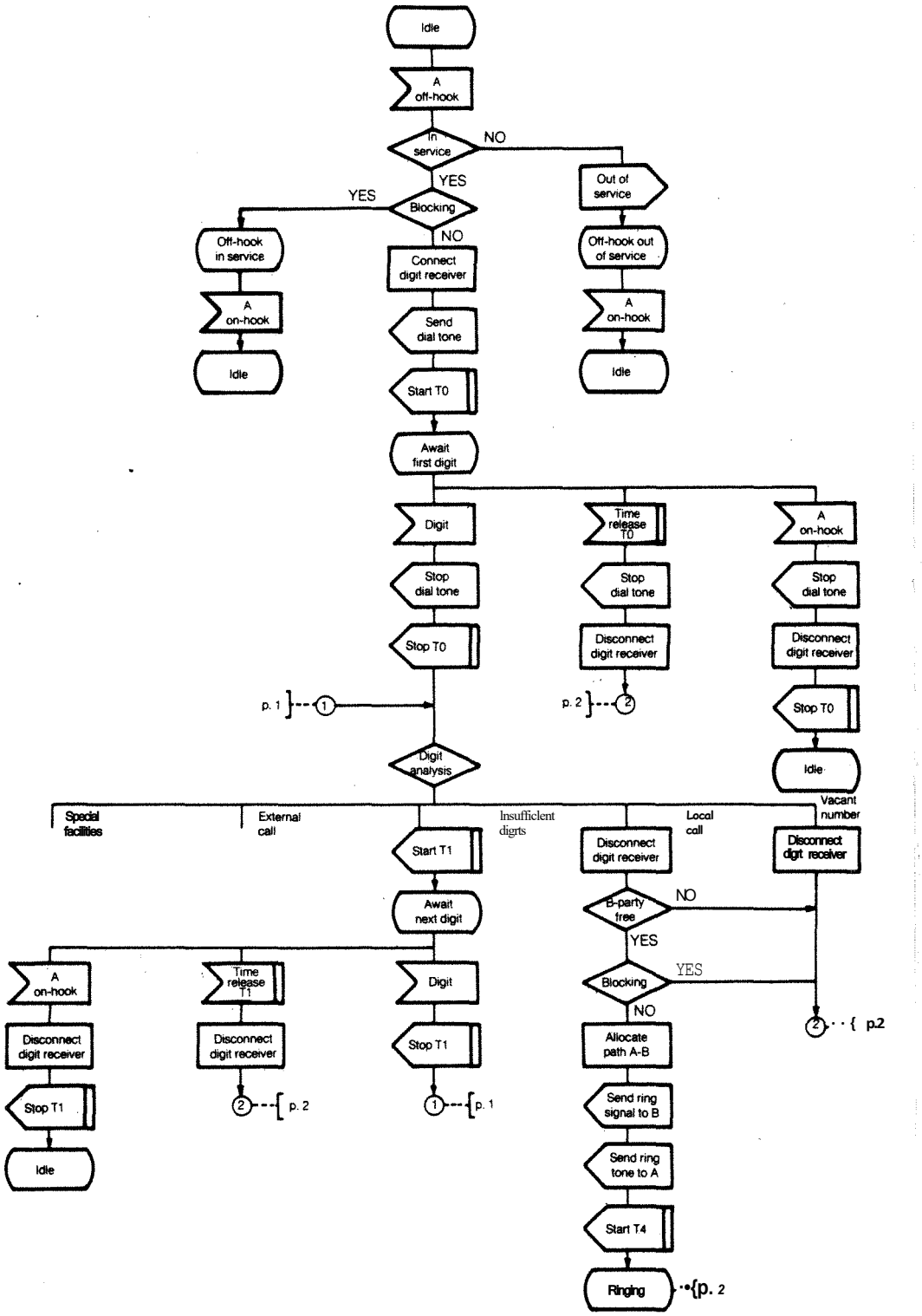


Bild 3.7 : Ausschnitt aus der Anrufbehandlung einer Telefonvermittlung /ROSA82/

Diskussion:

Beim Senden oder Empfangen von Signalen wird nur der Name des Signals angegeben, d.h. der Name eines Signals muß in einem Programm eindeutig sein; damit kann der Empfänger bzw. Sender identifiziert werden. Die eindeutigen Signalnamen können in größeren Projekten zu einer Vielzahl von Bezeichnungen führen. Einem Signal wird implizit durch den Namen der Informationsgehalt zugeordnet. Wie die Information z.B. beim Senden von Nachrichten gewonnen wird, kann in SDL nicht beschrieben werden. Dazu sind umgangssprachliche Ergänzungen nötig. Es ist nur ein asynchroner Nachrichtenaustausch möglich, d.h. ein Sender darf erst weitermachen, wenn sein Signal vom Empfänger angenommen wurde. Außerdem können in SDL Signale verbraucht werden, ohne daß ein Übergang beim Empfänger stattfindet bzw. der Sender davon benachrichtigt wird, daß die Nachricht weggeworfen wurde. Das Synchronisationsverhalten kann dadurch unübersichtlich werden.
Die Firma TEKADE, die SDL bei Entwicklung von Telefonvermittlungen verwendet, führt für die praktische Anwendung von SDL folgende Vor- und Nachteile auf /BRUE81/.
Vorteile:
- gute Dokumentationshilfe
- leicht erlernbar
- gut erweiterbar

Nachteile:
- Änderungen in der Graphik sind bei größeren Projekten aufwendig.

3.2.2 Konzepte in Programmiersprachen

3.2.2.1 Parallele Prozesse in CSP

In CSP (Communicating Sequentiell Processes), das von Hoare /HOAR78/ vorgeschlagen wurde, besteht ein Programm aus mehr - ren parallelen Prozessen.
In CSP unterscheidet man zwischen einfachen (simple) und strukturierten (structured) Anweisungen. Strukturierte Anweisungen sind:
- Alternativanweisungen (alternativ commands)
- Wiederholungsanweisungen (repetitive commands)
- Parallelanweisungen (parallel commands)

Eine Alternativanweisung ist eine Menge von sogenannten 'Guarded Commands'.
Ein Guarded Command besteht aus einem sogenannten Guard und einer Befehlsliste. Ein Guard ist eine Bedingung (Boolescher Ausdruck), die wahr sein muß, damit die dem Guard zugordnete Befehlsliste ausgeführt werden kann. Eine Bedingung kann sein:
- eine Eigenschaft der Programmvariablen. Trifft diese Eigenschaft auf die Variablen zu, so ist die Bedingung 'wahr'.
- eine erwartete Nachricht. Möchte der entsprechende Prozeß die passende Nachricht (Anzahl und Typ der Nachrichtenparameter) an den betrachteten Prozeß senden, gilt die Bedingung als wahr.
- eine logische Verknüpfung von Variableneigenschaften gefolgt von einer Empfangsanweisung. Die Bedingung ist dann wahr, wenn die Variablen die entsprechende Eigenschaft haben, und wenn der in der Empfangsanweisung angegebene Prozeß eine passende Nachricht senden möchte.

Eine Alternativanweisung kann nur ausgeführt werden, wenn mindestens ein Guarded Command in ihr ausgeführt werden kann, d.h. es muß bei mindestens einem Guard die angegebene Variableneigenschaft zutreffen, und der Quellprozeß für eine eventuell erwartete Nachricht darf nicht terminiert sein.
Ein Empfangsprozeß wartet solange, bis der angegebene Sendeprozeß sendebereit oder terminiert ist. Im ersten Fall gilt

dann das gesamte Guard als wahr, und die entsprechende Befehlsfolge wird ausgeführt. Die Alternativanweisung gilt als beendet. Im anderen Fall gilt das Guard als falsch. Sind alle Guards einer Alternativanweisung falsch, wird diese mit einer Fehlermeldung abgebrochen. Sind bei mehreren Guards die Bedingungen wahr, wird von diesen ein beliebiges zur Ausführung ausgewählt. Die restlichen Guards, deren Bedingungen ebenfalls wahr sind, haben keine Wirkung.
Eine Wiederholungsanweisung ist ebenfalls eine Menge von Guards. Der Unterschied zwischen der Wiederholungs- und Alternativanweisung ist, daß bei letzterer die Guards solange geprüft werden, bis keines mehr ausgeführt werden kann. Erst wenn keine Bedingung mehr wahr ist, gilt eine Wiederholungsanweisung als beendet.
Eine Parallelanweisung spezifiziert die gleichzeitige Ausführung mehrerer sequentieller Folgen von Anweisungen. Eine solche Folge von Anweisungen wird im weiteren mit Prozeß bezeichnet. Alle Prozesse beginnen gleichzeitig, und eine Parallelanweisung gilt erst dann als beendet, wenn alle in ihr enthaltenen Prozesse beendet sind. Die einzelnen Prozesse in einer Parallelanweisung dürfen keine gemeisamen Variablen benutzen.
Einfache Anweisungen sind :
- Wertzuweisungen,
- Eingaben,
- Ausgaben und
- das leere Kommando.

Da von den einfachen Anweisungen für die weiteren Betrachtungen nur die Ein-/Ausgaben wichtig sind, sollen nur diese näher erläutert werden.
Ein-/Ausgabeanweisungen beschreiben die Kommunikation zwischen parallelen Prozessen. In einer Eingabeanweisung wird angegeben, von welchem Prozeß die Eingabe erwartet wird (Quell- oder Sendeprozeß) und welchen Variablen die Werte der Nachrichtenparameter zugewiesen werden (Zielvariablen). In den Ausgabeanweisungen wird angegeben, an welchen Prozeß eine Nachricht gehen soll (Ziel- oder Empfangsprozeß) und welche Variablen-

werte als Nachrichtenparameter mitgesendet werden sollen. Die Anzahl und die Typen der Nachrichtenparameter bei der Ausgabeanweisung müssen mit der Anzahl und den Typen der Zielvariablen bei der Eingabeanweisung übereinstimmen.
Prozesse synchronisieren sich mit Hilfe der Ein- und Ausgabeanweisungen. Ein Prozeß, der ausgeben möchte, wird solange blockiert, bis der Zielprozeß die Nachricht will, also die entsprechende Eingabeanweisung ausführt. Ebenfalls wird ein Prozeß blockiert, der eine Nachricht erwartet, für die der Quellprozeß noch nicht sendebereit ist. Da in den Guards der Alternativanweisungen Eingabeanweisungen angegeben werden dürfen, kann ein Prozeß auf mehrere Eingaben gleichzeitig warten.

Beispiel:
Simulation eines Semaphors

```
S::val:integer; val:=0;
   *[(i:1 ..100)X(i)?V() -> val:=val+1
   [] (i:1 ..100)val>0; X(i)?P() -> val:=val-1
   ]
```

Bild 3.8: Beispiel eines CSP-Programms /HOAR78/

Diskussion:

Ein- und Ausgabeanweisungen stehen zwischen anderen Anweisungen. Dadurch ist es schwer, das Kommunikations- bzw. das Synchronisationsverhalten eines CSP-Programms zu überblicken.
Die Implementierung von Serviceprozessen (Serviceprozesse sind Prozesse, die anderen Prozessen bestimmte Hilfsfunktionen zur Verfügung stellen) wird durch das feste Schema der Sender- und Empfängerangaben erschwert. In den Guarded Commands, in denen der Serviceprozeß auf Aufträge wartet, muß jeder Prozeß, der eine Funktion verwenden will, in einem Guard aufgeführt werden. Dem Serviceprozeß muß ein auftraggebender Prozeß durch einen Nachrichtenparameter den Absender mitteilen (ähnlich der Kennung für die gewünschte Funktion), damit dieser weiß, an welchen Prozeß er eventuell die Antwort senden muß.

Dieses Problem soll mit einem Beispiel erläutert werden (siehe Bild 3.9).
Ein Prozeß S stellt die Funktionen F1 und F2 den Prozessen P1 und P2 zur Verfügung.

```
S: : FUN: integer, KEN: integer, .....
     [P1 ?(FUN, .....) -> KEN: =1                    Auftrag
     JP2? (fun, ..... ) -> KEN:=2                    annehmen
     ]
     [FUN=1  -> /* Funktion 1 ausführen */           Auftrag
     1FUN=2  -> /* Funktion 2 ausführen */           ausführen
     ]
     [KEN=1  -> P1 !(Rückgabeparameter)              Ergebnis
     JKEN=2  -> P2 !(Rückgabeparameter)              zurücksenden
     ]
]
```

Bild 3.9: Beispiel eines einfachen Serviceprozesses

In Guarded Statements sind Ausgaben nicht erlaubt. Dadurch ist es nicht möglich, eine Nachricht an irgend einen Prozeß aus einer Menge von Prozessen zu senden. Dies ist vor allem dann von Nachteil, wenn eine bestimmte Funktion von mehreren Serviceprozessen zur Verfügung gestellt wird.
In einem Guard darf nur eine Eingabeanweisung angegeben werden. Soll auf mehrere Eingaben gewartet werden, die in beliebiger Reihenfolge eintreffen können, so kann dies durch eine Wiederholungsanweisung und entsprechende Flags relativ einfach realisiert werden.

3.2.2.2 Parallele Prozesse in ADA

Ein ADA-Programm /ADA79/ besteht aus Unterprogrammen, Moduln und Blöcken. Diese Einheiten können ineinander verschachtelt sein. Unterprogramme und Blöcke werden ähnlich verwendet wie in ALGOL oder PASCAL und sollen im folgenden nicht weiter betrachtet werden.

Bei Moduln unterscheidet man zwischen sogenannten 'Package Moduln' und 'Task Moduln'. Ein Package Modul ist eine Sammlung von Unterprogrammen und/oder Daten. Auf diesen Typ von Moduln soll ebenfalls nicht weiter eingegangen werden. Er entspricht im wesentlichem den üblichen Datenmoduln.
Ein Task Modul ist eine Sammlung von Tasks und/oder Daten. Tasks sind Programmeinheiten, die parallel aktiv sein können (Prozesse). Ein paralleles ADA-Programm ist somit eine Menge von parallelen Tasks und Prozeduren, die von den Tasks aufgerufen werden.
Die Synchronisation und Kommunikation von Tasks erfolgt durch ein Rendezvouskonzept. Bei einem Rendezvous ruft ein Prozeß (Sendeprozeß) einen sogenannten Entry-Punkt in einem anderen Prozeß auf (Empfangsprozeß). Der Aufruf eines Entry-Punkts sieht genauso aus wie ein Prozeduraufruf, d.h. es muß der Name des aufgerufenen Entry-Punkts und es können Ein- und Ausgabeparameter angegeben werden. Die Namen von Entry-Punkten sind eindeutig. Aufrufe von Entry-Punkten mit einem bestimmten Namen nimmt genau ein Prozeß an. Ein Prozeß kann aber mehrere verschieden Entry-Punkte besitzen. Der Aufruf eines Entry-Punkts (Entry-Aufruf) wird vom Empfangsprozeß durch eine sogenannte Accept-Anweisung angenommen. Eine Accept-Anweisung enthält den Namen des Entry-Aufrufs, der angenommen werden soll, und den sogenannten Accept-Rumpf. Nimmt ein Prozeß einen Entry-Aufruf an, wird der Accept-Rumpf ausgeführt. Während der Empfangsprozeß den Accept-Rumpf ausführt, bleibt bzw. wird die Sendetask blockiert. Der Empfangsprozeß nimmt den Aufruf eines Entry-Punkts erst an, wenn er zu einer entsprechenden Accept-Anweisung kommt. Der Sendetask ist durch die eindeutige Zuordnung von Entry-Punkten zu Prozessen bekannt, welche Task den Entry-Aufruf annimmt. Die Empfangstask weiß nicht, von welcher Task der Entry-Punkt aufgerufen wurde. Ruft ein Prozeß einen bestimmten Entry-Punkt auf und hat der angesprochene Prozeß eine entsprechende Accept-Anweisung noch nicht erreicht, so wird der Sendeprozeß blockiert. Erreicht eine Task eine entsprechende Accept-Anweisung, werden die eventuell angegebenen Eingabeparameter übernommen und der Accept-Rumpf dieser Accept-Anweisung ausgeführt. Danach werden eventuell angege-

bene Rückgabeparameter an die Sendetask transferiert. Dann wird die Sendetask wieder deblockiert. Umgekehrt wird die Empfangstask blockiert, wenn sie einen Entry-Punkt erreicht (Accept-Anweisung), der noch nicht von einer Task aufgerufen wurde. Die Empfangstask bleibt solange blockiert, bis der entsprechende Entry-Punkt aufgerufen wird. Dann wird die Empfangstask deblockiert und die Sendetask blockiert, bis die Empfangstask eventuelle Parameter übernommen hat, den zu dieser Accept-Anweisung gehörenden Accept-Rumpf ausgeführt hat und eventuell angegebene Rückgabeparameter an die Sendetask zurückgegeben hat. Die Blockade einer auf Empfang stehenden Task kann zeitüberwacht werden, d.h. wird innerhalb einer bestimmten Zeit nicht der zugehörige Entry-Punkt aufgerufen, wird die Task deblockiert und eine Ausnahmebehandlung eingeleitet.
Dieses Rendezvouskonzept soll mit dem folgenden Bild 3.10 noch verdeutlicht werden.

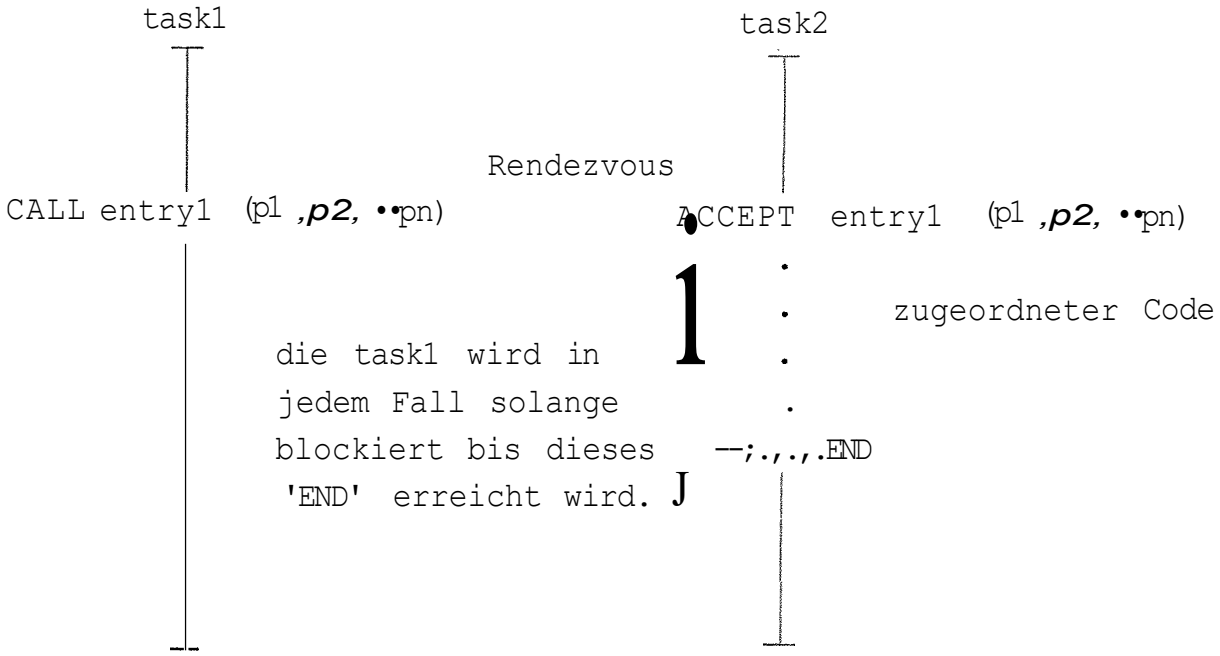


Bild 3.10: Beispiel für ein Rendezvous in ADA

Mit Hilfe der sogenannten Select-Anweisung kann eine Task gleichzeitig auf den Aufruf verschiedener Entry-Punkte warten. Eine Select-Anweisung ist eine Menge von Bedingungen, die miteinander verodert sind. Eine Bedingung kann ein bestimmter

Zustand der Daten oder eine Accept-Anweisung sein. Eine Bedingung gilt als wahr, wenn die Daten in einem entsprechenden Zustand sind oder ein entsprechender Entry-Aufruf vorliegt. Ist eine Bedingung wahr, wird das entsprechende Codestück bzw. der Code der Accept-Anweisung ausgeführt. Eine Select-Anweisung gilt als beendet, wenn eine Bedingung wahr ist und der zugehörige Code ausgeführt wurde. Ist keine der Bedingungen wahr wird der 'Else'-Zweig ausgeführt falls vom Programmierer einer vorgesehen wurde.

Beispiel:
Mehrere Prozesse können die Variable 'RESOURCE' durch den Aufruf der Prozedur 'READ' lesen. Allerdings müssen sie erst den Entry Punkt 'START' aufrufen, um das Lesen anzukündigen. Nach dem Aufruf von 'READ' wird die Variable 'RESOURCE' durch den Aufruf von 'STOP' wieder freigegeben. Es können gleichzeitig mehrere Prozesse die Variable lesen. Beschrieben kann sie allerdings nur werden, wenn kein Prozeß die Variable belegt.

```
task READEA_WAITER is
   procedure READ (V : out ELEM);
   entry WRITE(E : in ELEM);
end;

task body READEA_WAITER is
   RESOURCE : ELEM;
   READERS  : INTEGER := 0;

   entry START;
   entry STOP;

   procedure READ(V : out ELEM) is
      _ READ is a procedure, not an entry, hence concurrent calls of READ are possible
      - READ synchronizes such calls with the entry calls START and STOP
   begin
      START; V := RESOURCE; STOP;
   end;

begin
   accept WRITE(E : in ELEM) do
      RESOURCE := E;
   end;

   loop
      selact
         accept START;
         READERS := READERS + 1;
      or
         accept STOP;
         READERS := READERS - 1;
      or when READERS = 0 =>
            accept WRITE(E : in ELEM) do
               RESOURCE := E;
            end WRITE;
      end selact;
   end loop;
end READER_WRITER;
```

Bild 3.11 : Beispiel für ein ADA-Programm /ADA79/

Diskussion:

Die Sende- (Entry-Aufrufe) und Empfangsanweisungen (Accept-Anweisungen) sind über den gesammten Code verstreut; dadurch wird es schwierig, das Synchronisations- und Kommunikationsverhalten eines ADA-Programms zu überblicken /STR080/.
Ein Entry-Aufruf kann von verschiedenen Accept-Anweisungen an verschiedenen Stellen im Prozeß angenommen werden. Diesen Accept-Anweisungen kann ein unterschiedlicher Code zugeordnet sein, d.h. was in einer Empfangstask durch einen Entry-Aufruf geschieht, hängt auch davon ab, in welchen Zustand sich diese

befindet /STRA80/, nicht nur vom Namen des Entry-Punkts.
Die Task, die einen Entry-Punkt aufruft, wird in jedem Fall solange blockiert, bis der Code ausgeführt ist, der der annehmenden Accept-Anweisung zugeordnet wird. Wenn eine Task eine Accept-Anweisung ausführt, für die noch kein Entry-Aufruf vorliegt, wird diese blockiert. Trifft der Entry-Aufruf ein, wird die Empfangstask deblockiert, aber die Sendetask blokkiert, d.h. es sind zwei Prozeßumschalteraufrufe notwendig, was sehr zeitaufwendig sein kann, vor allem, wenn die beiden Tasks auf demselben Prozessor laufen /ROBE80/. Ist der Code, der den Acceptanweisungen zugeordnet ist, noch sehr lang, wird der Grad der Parallelarbeit von Tasks stark veringert /STR080/.
Tasks können in einer Accept-Anweisung nicht festlegen, von welcher Task sie einen Entry-Aufruf annehmen wollen. Einer Empfangstask ist es nicht direkt möglich, die Sendetask zu bestimmen. Allerdings ist es dadurch problemlos möglich, Serviceprozesse zu implementieren. Möchte ein neuer Prozeß den angebotenen Service nutzen, muß der Serviceprozeß nicht geändert werden.
Da die Namen von Entry-Punkten innerhab eines ADA-Programms eindeutig sein müssen, weiß die Sendetask, wer den Entry-Aufruf annimmt.
In ADA kann zwar das Eintreffen eines Entry-Aufrufs zeitüberwacht werden, es ist aber nicht möglich, das Annehmen eines Entry-Aufrufs zeitlich zu überwachen. Eine Sendetask kann nicht angeben, innerhalb welcher Zeit der Entry-Aufruf ausgeführt sein muß.

3.2.2.3 Parallele Prozesse in ITP

Ein sogenanntes 'Input Tool Process' Programm /BOPS81/ besteht aus mehreren Prozessen. Jeder Prozeß enthält mehrere Input Tools (IT) und/oder Basic Tools (BT). Input Tools können ineinander verschachtelt sein. Ein Prozeß kann als IT der äußersten Ebene betrachtet werden. Der Name dieses äußersten IT entspricht dem Prozeßnamen.

Im allgemeinen besteht ein Input Tool aus vier Komponenten:
- der Initialisierung
- der Eingaberegel
- dem Toolrumpf
- der Rückgabeinformation

Ein IT durchläuft vier Phasen:
- Die Initialisierung führt Vorbesetzungen durch
- Eintreffende Eingaben werden geprüft, ob sie nach der Eingaberegel zulässig sind
- Wenn die Eingaberegel erfüllt ist, wird der Tool-Rumpf ausgeführt
- Zuletzt wird die Rückgabeinformation über Parameter zurückgegeben

Die wichtigste Komponente eines IT ist die Eingaberegel. Die Eingaberegel beschreibt die zulässigen Eingabe- und Aufruffolgen von ITs, die dieser Input Tool enthält (ähnlich den Pfadausdrücken in /CAMP74/).
Die Eingaberegel ist ein Ausdruck über der Menge der IT-Namen, BT-Namen und Eingabebezeichnungen sowie der Operatorenmenge
$,*,;,&,+ .
T und V seien zwei Eingaben bzw. Input Tool-Namen oder Basic-Tool Namen dann gilt für die Operatoren:

1. T;V : Nach T folgt V (Sequenz)
2. T$: T wird beliebig oft wiederholt. Die Folge endet, wenn T leer ist (Definition der leeren Folge siehe weiter unten).
3. T* : T wird überhaupt nicht oder beliebig oft wiederholt.
4. T&V : Es muß T und V angestoßen werden. Die Reihenfolge ist bedeutungslos.
5. T+V : Es muß T oder V angestoßen werden (exklusives oder).

Syntaktisch leere Eingabefolgen sind nicht erlaubt.
Nach der Aktivierung eines IT führt es die Initialisierung aus. Erhält ein IT nach der Initialisierung eine Eingabe, wird geprüft, ob diese Eingabe nach der Eingaberegel als nächstes

Zeichen erlaubt ist. Wird die Eingabe nicht erwartet, wird sie zurückgewiesen (weggeworfen). Ist die Eingabe als nächste zugelassen, wird das zugehörige IT aktiviert und auf die nächsten Eingaben gewartet. Alle Folgen, für die eine angenommene Eingabe nicht gepaßt hat, werden nicht weiter betrachtet. Eine Eingaberegel ist erfüllt, wenn das Ende einer erlaubten Eingabefolge erreicht wird.
Wenn die Eingaberegel eines IT erfüllt ist, wird der dazugehörige Toolrumpf ausgeführt. Danach gilt das Input Tool als beendet. Der Name eines beendeten Tools gilt als Eingabe für das nächsthöhere IT. Diese Eingabe für das höhere IT kann ebenfalls das Ende einer Eingabefolge sein. Der dazugehörige Toolrumpf wird ausgeführt usw.. Für Basic Tools sind als Eingaben nur Nachrichten von anderen Prozessen erlaubt (siehe weiter unten).
Mit den bisherigen Möglichkeiten können als Eingaberegeln nur reguläre Ausdrücke angegeben werden. Um mächtigere Regeln angeben zu können, gibt es in ITP noch sogenannte Vor- und Nachbedingungen (Pretests und Posttests).
Eine Vorbedingung beschreibt eine bestimmte Zustandsmenge der lokalen Daten (Wertebelegung der Variablen). Die Vorbedingung ist wahr, wenn die lokalen Daten sich in einem der beschriebenen Zustände befinden. Ist die Vorbedingung wahr, dann gilt die nachfolgende Eingaberegel. Wenn die Vorbedingung falsch ist, dann gilt die leere Eingaberegel. Eine leere Eingaberegel ist also eine Eingaberegel, deren Vorbedingung falsch ist.
Eine Nachbedingung ist ebenfalls eine Bedingung über die lokalen Variablen. Trifft die Nachbedingung auf lokalen Variablen nach dem Ablauf des Toolrumpfs nicht zu, gilt das Ende dieses Toolrumpfs nicht als Eingabe für die nächst höhere Ebene.

Beispiele:
Das IT 'CD-array' nimmt die Koordinaten von einem Digitalisierer auf. Wenn der Schalter 'sample' gedrückt wird, wird ein Koordinatenpaar angenommen und in einem Feld gespeichert. Wird die Taste 'stopswitch' gedrückt, wird das Sammeln von Koordinatenpaaren beendet. Im IT 'stop' wird die Variable 'proceed' auf 'false' gesetzt. Dadurch wird die Vorbedingung von 'CD-

array' falsch und damit beendet.

```
tool CD-array ([ ] int x, y) = input (!proceed | : (collect +stop))$ end
    int i; bool proceed;
    tool collect =input sample(xx, yy) end
        int xx, yy;
        x[i] :=xx;
        y[i] :=yy;
        i :=i +1
    end
    tool stop =input stopswitch end
        proceed :=false
    end
    init i := 1, proceed :=true end
end
```

Bild 3.12 : Beispiel für ein ITP-Programm /BOPS81/
Annehmen der Koordinaten eines Digitalisierers

Der IT 'Zz' nimmt nur groß- und kleingeschriebene 'Z' an. Dies wird durch eine Nachbedingung erreicht.

```
tool Zz = input Key(c) :| c ="Z" 1+ Key(c) :| c ="z" | end
end
```

Bild 3.13 : Beispiel für ein ITP-Programm /BOOS81/
Annehmen von groß- und kleingeschriebenen 'Z'

Für die Kommunikation von Prozessen untereinander gibt es einen Botschaftsmechanismus. Prozesse empfangen Botschaften durch Basic Tools (BT). Der Unterschied zwischen Basic Tools und Input Tools ist, daß Basic Tools statt einer Eingaberegel eine Empfangsregel haben. Mit der Empfangsregel wird die Struktur der Nachricht definiert. Eine Nachricht wird nur angenommen, wenn sie die gleiche Struktur hat wie in der Empfangsregel angegeben. Basic Tools mit einer leeren Empfangsregel werden als Signale bezeichnet.
Allgemeine Struktur der Basic Tools:

```
tool NAME=receive message
  Basic Tool Rumpf
end
```

Zum Senden von Nachrichten wird folgende Anweisung verwendet:

```
send   PROCESSNAME.BASICTOOLNAME    (MESSAGE)
```

Der sendende Prozeß wird solange blockiert, bis die Nachricht angenommen wurde.
Die Angabe des Prozeßnamens, des Basic-Tool-Namens und der Botschaftsstruktur (Empfangsregel) ist optional. Dadurch sind vier Adressierungsarten möglich.

- Werden der Prozeßname und der Basic Tool Name angegeben, dann ist der Empfänger eindeutig bestimmt. Die Nachricht kann nur von dem angegebenen Prozeß mit dem angegebenen Basic Tool angenommen werden.
- Wird nur der Prozeßname angegeben, dann kann die Nachricht von jedem Basic Tool dieses Prozesses angenommen werden, falls es die gleiche Empfangsregel hat.
- Wird nur der Basic Tool Name angegeben, kann die Nachricht jeder Prozeß annehmen, der ein Basic Tool mit diesem Namen enthält.
- Wird weder ein Prozeßname noch ein Basic Tool Name angegeben, kann jeder Prozeß mit jedem Basic Tool die Nachricht annehmen; es muß nur die Empfangsregel passen.

Ein Prozeß kann angeben, von welchen Prozessen er Nachrichten annehmen will. Die Namen der Prozesse, von denen Nachrichten erwartet werden, werden in der Eingaberegel angegeben.
In ITP gibt es Variable vom Typ 'Prozeßmenge'. Diese Variablen enthalten Namen von Prozessen. Mit diesen Variablen kann angegeben werden, von welchen Prozessen eine Nachricht erwartet wird, bzw. an welche Prozesse ein Nachricht gesendet werden soll, wobei die Reihenfolge für das Senden bedeutungslos ist.

Beispiel:
Der Prozeß 'Printer' betreibt und verwaltet einen Drucker. Um zu verhindern, daß die Ausdrucke von mehreren Prozessen vermischt werden, muß der Drucker mit dem Aufruf 'first line' belegt werden. Der Drucker wird mit dem Zeichen 'EOF' wieder freigegeben.

```
to 1 printer = input (first line; 1more 1: source _,. line$)$ end
     bool more; process set source;
     tool first line = input line end
          ifmore
          then source := {sender}
          fi
     end
     tool line = receive string msg;
          more := (msg ;!, EOF);
          if more
          then lineprint(msg)
          eise skip page
          fi
     end
end
```

Bild 3.14 : Beispiel für ein ITPProgramm
Verwaltung eines Druckers /BOPS81/

Diskussion:

Da die Reihenfolge, in der ein Prozeß Nachrichten annimmt an einer Stelle beschrieben wird, erhält man schnell einen Überblick über das Empfangsverhalten eines Prozesses. Nachteilig ist, daß das Ausgabeverhalten nicht ebenfalls an einer Stelle beschrieben wird.
Es besteht keine Möglichkeit, verschiedene Nachrichten an mehrere Prozesse gleichzeitig oder alternativ zu senden. Es können nur Nachrichten mit gleichen Namen alternativ an verschiedene Prozesse gesendet werden (bei der Sendeanweisung den Prozeßnamen weglassen).
In verschiedenen Prozessen kann es ein Basic Tool mit gleichen Namen aber verschiedenen Empfangsregeln geben. Sendet ein Prozeß eine Nachricht an dieses BT, ohne daß der Zielprozeß angegeben wird, hängt es von der Empfangsregel ab, d.h. von der Nachrichtenstruktur, welche Prozesse diese Nachricht annehmen können. Dieser Einfluß der Nachrichtenstruktur auf die Menge der möglichen Empfänger kann das Kommunikationsverhalten eines Prozeßsystems unübersichtlich machen.
Das Problem, daß ein Prozeß auf mehrere Nachrichten wartet, die in beliebiger Reihenfolge eintreffen können, kann in ITP mühelos durch eine entsprechende Eingaberegel gelöst werden.

Serviceprozesse lassen sich in ITP problemlos formulieren. Die Adressierungsmöglichkeiten beim Senden von Nachrichten sind, trotz obiger Einwände, flexibler als in anderen Konzepten. Serviceprozesse können verhindern, daß sie von beliebigen Prozessen verwendet werden. In ITP kann ein Sender feststellen, welcher Prozeß seine Nachricht letztlich angenommen hat (sogenanntes 'receive primitive'). Ebenso kann ein Empfänger den Absender einer Nachricht ermitteln (sogenanntes 'sender primitive').
Es ist nur ein synchrones Senden möglich, d.h. der Sender wird solange blockiert, bis der Empfänger die Nachricht angenommen hat. Umgekehrt wird der Empfänger solange blockiert, bis eine der erwarteten Nachrichten eintrifft. Durch dieses Sende-/Empfangsverhalten wird der Grad an Parallelarbeit eingeschränkt.

3.2.2.4 Parallele Prozesse in PLITS

Ein PLITS-Programm /FELD79/ besteht aus mehreren in sich abgeschlossenen Moduln. Da aber Moduln in PLITS aktive Einheiten sein können, sind sie mit den Tasks oder Prozessen in anderen Konzepten vergleichbar. Moduln (Prozesse) kommunizieren untereinander durch Botschaften. Eine Botschaft ist eine Menge von Paaren der Art NAME~wERT (Botschaftskomponenten). Wobei 'NAME' die Bezeichnung (Name) und 'WERT' der Typ der Botschaftskomponente ist. Der Name und der Typ einer Botschaftskomponente müssen allen Moduln bekannt sein. Die verwendeten Paare werden durch folgende Anweisung global bekannt gemacht:

```
public S1:T1,S2:T2, ... Sm:Tm;
```

Dabei werden mit Si die Namen und mit Ti die zugehörigen Typen bezeichnet.
In PLITS können Variable vom Typ 'message' deklariert werden. Es ist möglich, durch eine vorherige Typdeklaration die Struktur der Variablen vom Typ 'message' festzulegen.

Beispiel für die Strukturdefinition des Typs 'message':

```
message (sj~Xj,sk~Xk .....)
```

Jedes Xi ist eine Variable vom Typ Ti und das Paar Si:Ti wurde mit public deklariert.
Jede Botschaft hat implizit noch zwei weitere Komponenten. Eine Komponente heißt 'source' und ist vom Typ 'module'. Die andere heißt 'about' und ist vom Typ 'transaction'. Mit der Komponente Source kann der Empfänger einer Botschaft den Sender ermitteln. Mit der Komponente About können beim Senden und Empfangen noch zusätzliche Bedingungen angegeben werden. Darauf soll im folgenden nicht näher eingegangen werden.
Beispiel für die Deklaration einer Variablen vom Typ 'message':

```
var BOTSCHAFT message
```

Zum Senden Nachrichten wird folgende Anweisung verwendet:

```
send M to V  about K
```

Dabei ist Meine Botschaft (eine Menge von Paaren der Art NAME~wERT), V ein Modulname (Zielprozeß) und Kein Transaktionsbezeichner. Die Namen der gesendeten Paare müssen in der Public-Anweisung des sendenden Prozesses deklariert sein.
Eine Sendeanweisung hat keine direkte Wirkung auf den sendenden Prozeß. Der Sendeprozeß wird nicht blockiert, und es werden keine Variablen verändert.
Eine Empfangsanweisung hat folgendes Aussehen:

```
receive  M   from V   about K
```

Dabei ist M der Name einer Variablen vom Typ 'message', V ist der Name des Prozesses, von dem die Nachricht erwartet wird, und K ist ein Transaktionsbezeichner. Eine Task, die eine Nachricht empfangen möchte, wird solange blockiert, bis diese Nachricht eintrifft.

Der empfangende Prozeß kann die Komponenten einer empfangenen Nachricht dadurch ansprechen, daß er den Namen der Botschaftsvariablen und den Namen der Botschaftskomponente angibt.

Beispiel:

V:=B.K Der Variablen V wird der Wert der Komponente K der Botschaftsvariablen B zugewiesen.

Beispiel für ein PLITS-Programm:
Der Prozeß 'Fibonacci' berechnet die nächste Fibonaccizahl und gibt sie an den auftraggebenden Prozeß.

```
1   Const George = mod
2     Begin
3     Public Recipient: module
4           Object: integer
5     var I, J, NexL.Fib: integer
6         Messl, Mess2: message
            .
            .
            .
7     Send message (Recipient ~ Me) To
                        Fibonacci
8     Receive Mess2 From Fibonacci
9     NexLFib : - Mess2,Object
            .
            .
            .
10    End
```

```
Const Fibonacci = mod
  Begin
  Public Recipient: module
        Object: integer
  var This, Last, Previous: integer
      Request: message
  Last:- 0
  This:= 1
  While True Do
    Begin
      Receive Request
      Previous : - Last
      Last :-This
      This : - Last + Previous
      Send message (Object ~ This) To
                        Request•Recipient
    End
  End
```

Bild 3.15 : Beispiel für ein PLITS-Programm /FELD79/

Diskussion:

In PLITS sind die Sende- und Empfangsanweisungen über den gesamten Programmtext verstreut. Dadurch ist es schwierig, einen Überblick über das Kommunikationsverhalten eines Programms zu bekommen.
In PLITS ist es nicht möglich, auf Nachrichten von mehreren bestimmten Prozessen gleichzeitig zu warten bzw. mehrere Nachrichten gleichzeitig oder alternativ an bestimmte Prozesse zu senden. Ein Prozeß kann nicht angeben, daß er nur Nachrichten von bestimmten Prozessen annehmen möchte. Ein Empfangsprozeß

weiß erst, wer der Sender ist, wenn er die Nachricht angenommen hat.
Serviceprozesse lassen sich aus den Gründen, die bereits in Kapitel 3.2.2.1 diskutiert wurden, schwer implementieren. Ein Serviceprozeß muß eine Nachricht erst annehmen, damit er weiß, welche Hilfsfunktion aufgerufen wurde. Kann diese Funktion momentan nicht ausgeführt werden, muß der Serviceprozeß die Nachricht puffern.
In PLITS wird der sendende Prozeß nicht blockiert (asynchrones Senden).

3.2.2.5 Parallele Prozesse in *MOD

Die Sprache *MOD (Star-MOD) /COOK80/ wurde für verteilte Rechensysteme entwickelt. In *MOD wird ein Rechnernetz als eine beliebige Menge von Rechnern gesehen, die mit festen Leitungen untereinander verbunden sind.
Ein Programm in *MOD besteht aus mehreren Moduln. Ein Modul kann zur Typdeklaration oder als textueller Rahmen für zusammengehörige Programmteile benutzt werden. In *MOD unterscheidet man zwischen Netzwerkmoduln und Prozessormoduln.
In einem Netzwerkmodul werden die Verbindungen zu anderen Prozessoren beschrieben, und es können alle Datentypen, Konstanten, Prozeduren und Prozesse deklariert werden, die den Prozessormoduln bekannt sein sollen.
Ein Prozessormodul enthält mehrere parallele Prozesse. Alle Prozeduren und Prozesse innerhalb eines Prozessormoduls können auf gemeinsame Variable zugreifen, wogegen die Kommunikation von Prozessen aus unterschiedlichen Prozessormoduln nur mit Botschaften möglich ist.
Ein Prozeß kann von anderen Prozessen wie eine Prozedur aufgerufen werden. Bei einem solchen Aufruf können auch Parameter übergeben werden. Diese Art von Prozessen werden Funktionsprozesse genannt. Funktionsprozesse sind wiedereintrittsfähig, d.h. bei jedem Aufruf eines solchen Prozesses wird von ihm ein Exemplar erzeugt. Es können also mehrere Exemplare eines Funktionsprozesses gleichzeitig laufen. Ist ein solcher Funktions-

prozeß mit seinen Berechnungen fertig, werden die Rückgabeparameter an den aufrufenden Prozeß gegeben. Dann wird dieses Exemplar des Funktionsprozesses wieder zerstört. Beim Aufruf eines Funktionsprozesses wird der aufrufende Prozeß solange blockiert, bis der Funktionsprozeß die Parameter zurückgegeben hat und das Exemplar zerstört wurde. Die Kommunikation mit Funktionsprozessen erfolgt also über Parameter.

Neben diesem Aufruf von Funktionsprozessen können Prozesse noch über Botschaften miteinander kommunizieren. Dazu wurden in *MOD Ports eingeführt.

Ein Port wird ähnlich deklariert wie eine Variable. Er hat einen Namen und Parameter. Jedem Port ist eine Warteschlange zugeordnet, in die Nachrichten, die über diesen Port gehen, eingekettet werden.

Das Senden einer Nachricht sieht aus wie ein Prozeduraufruf. Statt dem Prozedurnamen wird der Portname angegeben. Beim Senden einer Nachricht wird diese in die Warteschlange des entsprechenden Ports eingehängt. Der sendende Prozeß wird nicht blockiert.

Der Empfänger holt irgendwann die gewünschte Nachricht aus dem entsprechenden Port ab und führt dann den zugehörigen Portrumpf aus. Der auszuführende Portrumpf wird bei der Empfangsanweisung angegeben. Empfängt ein Prozeß an mehreren Programmstellen Nachrichten über denselben Port, so ist u.U. jeder Empfangsanweisung ein anderer Portrumpf zugeordnet. Der Empfangsprozeß wird blockiert, wenn das angesprochene Port leer ist. Ein Prozeß kann gleichzeitig auf das Eintreffen mehrerer Nachrichten warten. Ein Port kann von mehreren Prozessen angesprochen werden. Der Sendeprozeß weiß nicht, welcher Prozeß die Nachricht annimmt. Für das Empfangen von Nachrichten wird folgende Anweisung /COOK80/ verwendet:

REGIONSTATEMENT ::= region PORTIO [,PORTIO] ... ;
--[PORTBODY]
end region
PORTBODY ::= STATEMENTLIST ! PORTCASE [;PORTCASE] ...
PORTCASE ::= PORTIO: begin STATEMENTLIST
-- end PORTIO

Beispiel:

Das folgende *MüD Programm beschreibt das allgemein bekannte Philosophenproblem. Fünf Philosophen sitzen um einen Tisch, auf dem ein Topf Spaghetti steht und fünf Gabeln liegen. Jeweils eine Gabel liegt links, bzw. rechts von jedem Philosophen. Jeder Philosoph denkt nach oder ißt. Zum Essen benötigt jeder Philosoph zwei Gabeln. Er nimmt die Gabeln die links und rechts von ihm liegen. Nach dem Essen legt er die Gabeln wieder auf ihre Plätze zurück. Es gilt nun, den Zugriff auf die Gabeln so zu synchronisieren, daß keine Verklemmung entsteht.

Jedem Philosophen ist der gleiche Prozeß zugeordnet (Feld von Prozessen). Die Philosophenprozesse kommunizieren über die Ports get, got und put.

Das zugehörige *MÜD-Programm zeigt das Bild 3.16.

```
network module dining room= (phil [1], phil [2]) ,(phil [2], phil [3 J) ,
                            (phil [3 J,phi 1 [4 JJ, (phil [4 |, phil [5 J),
                            (phil[SJ ,phil(l]);   (*ring network*)
   processor module phil[SJ;
        define get, put, got;
        module at table;
             po rt get, put, got, getforks, putforks;
             import phil;
             var myfork, gotone: semaphore;
             port get(who, fork: integer);   (*get "fork" for "who"*)
             P2_il got(who: integer);        (*let "who" know*)
                  put(fork: integer);        (*give "fork" back*)
             process waitrep(who: integer);  (*local rep. for "who"*)
                  begin  p(myfork);  got(who)
                  end waitrep;
             procrn communicator;
                  begin loop
                        regfon get, put, got;
        get:              begin
                          if fork<>pid then phil(pid mod 5+1].get(who,fork)
                                     else waitrep(who)
                                     end if
                          end get;
        put:              begin
                          l i fork<>pid then phil[pid mod 5+1].put(fork)
                                     else v(myfork)
                                     end l i
                          end put;
        got:              begin
                          l l who<>pid       phil (pid mod 5+1] .got (who)
                                             v(gotone)
                                   end if
                          end got
                        end reg ion
                  end loop
             end communicator;
             erc)cedure getforks;   (*philosopher waits for both forks*)
                  begin   get(.li pid=1 l:.!:!.fi1l else pid-1);   p(gotone);
                          get(if pid=1 tL:!..lm5 tl$..!t pid);     p(gotone)
                  end getforks;
             procedure putforks;   (*give forks back and don' t wait*)
                  begin  put(pid);
                         put( if pid=1 tL:!..lm5 else pid-1)
                  end putforks;
             begin   communicator
             end at_ table;
        begin loop
              (*think*)   getforks;   (*eat *)  putforks
             .!!l!9,loop
        end phil
   end diningroom.
```

Bild 3.16 : Beispiel für ein *MÜD-Programm /CüüK80/

Diskussion:

Die Sende- und Empfangsanweisungen sind in *MOD über den ganzen Programmtext verstreut. Dadurch ist es schwierig, einen Überblick über das Kommunikationsverhalten eines Programmsystems zu bekommen.
Wartet ein Prozeß auf mehrere Nachrichten, die in beliebiger Reihenfolge eintreffen können, so läßt sich dieses Problem nur ähnlich umständlich wie in ADA lösen.
In *MOD lassen sich Serviceprozesse leicht implementieren. Dadurch, daß ein Serviceprozeß die verschiedenen Aufträge über benannte Ports bekommt, ist es möglich, daß nur Aufträge angenommen werden, die auch ausgeführt werden können. Die gleiche Hilfsfunktion kann von verschiedenen Prozessen angeboten werden. Diese Prozesse greifen auf denselben Port zu. Allerdings weiß ein Prozeß nicht, an welchen Prozeß eine Nachricht geht.
Ein Prozeß kann von einem bestimmten Port an mehreren Programmstellen Nachrichten annehmen. Den einzelnen Empfangsanweisungen können dabei unterschiedliche Portrümpfe zugeordnet sein. Die Wirkung einer Nachricht hängt also vom Zustand des Empfangsprozesses ab. Der Sender weiß nicht mehr, was seine Nachricht beim Sender bewirkt (siehe auch ADA).
Direkt können Nachrichten nur asynchron ausgetauscht werden, d.h. der sendende Prozeß wird nie blockiert.

3.2.2.6 Parallele Prozesse und Communication Ports nach Mao/Yeh

Nach Mao/Yeh /MAYE80/ besteht ein Programm aus mehreren parallelen Prozessen. Diese Prozesse haben keine gemeinsamen Objekte (z.B. gemeinsame Daten oder Monitore). Untereinander können Prozesse Informationen nur durch Botschaften austauschen. Zum Austausch von Botschaften gibt es drei Arten von Anweisungen:
- Eine Anweisung zum Aufbau einer Verbindung und zum gleich-

zeitigen Übertragen von Botschaften (connect statement).
- Eine Anweisung zum Empfangen von Nachrichten (port statement).
- Eine Anweisung zum Lösen einer Verbindung (disconnect statement).

Die Anweisung zum Aufbau einer Verbindung und gleichzeitigem Senden einer Nachricht sieht wie folgt aus:

```
connect PROZESSNAME.PORTNAME (PARAMETERLISTE);
```

Die Nachricht (Parameterliste) geht dabei über den angegebenen Port zu dem gewünschten Prozeß (Zielprozeß). Ein Prozeß, der eine Verbindung aufgebaut hat und eine Botschaft senden möchte (Quellprozeß), wird solange blockiert, bis der Zielprozeß die Verbindung löst (Verbindungsabbauanweisung). Der Zielprozeß führt im allgemeinen vor dem Lösen einer Verbindung eine Empfangsanweisung (port statement) aus. Die Anweisung zum Empfangen einer Nachricht hat folgendes Aussehen:

```
port PORTNAME  PARAMETERLISTE ;
      condition BOOLEAN EXPRESSION;
      servant   PROZESSLISTE;
      PORTRUMPF
endport;
```

Ein Prozeß kann eine Nachricht nur empfangen, wenn der beim Schlüsselwort 'condition' angegebene boolesche Ausdruck wahr ist. Dieser boolesche Ausdruck beschreibt eine Eigenschaft der lokalen Daten und ist nur sinnvoll, wenn ein Prozeß gleichzeitig über mehrere Ports Nachrichten erwartet (Nichtdeterministische Ports), denn die lokalen Variablen können dann nur noch durch das Eintreffen von anderen Nachrichten geändert werden (es gibt keine gemeinsamen Daten). Ein Prozeß kann auf mehrere Nachrichten gleichzeitig warten (Nichtdeterministische Ports). Möchte ein Prozeß Nachrichten empfangen, so wird er solange blockiert, bis eine der erwarteten Nachrichten eintrifft (über ein entsprechendes Port und von einem der

angegebenen Prozesse). Beim Annehmen einer Nachricht werden die Nachrichtenparameter in die in der Parameterliste der Empfangsanweisung (port statement) angegebenen Variablen übernommen. Danach führt der Zielprozeß den Code aus, der dieser Empfangsanweisung zugeordnet ist (Portrumpf). Empfängt ein Prozeß an mehreren Programmstellen Nachrichten über dasselbe Port so können die einzelnen Empfangsanweisungen unterschiedliche Portrümpfe haben.
Der Quellprozeß bleibt solange blockiert, bis der Zielprozeß die entsprechende Anweisung zum Lösen der Verbindung ausführt. Diese Anweisung sieht wie folgt aus:

rr (PROZESSNAME) ;

Fehlt der Prozeßname, wird die Verbindung zu dem Prozeß gelöst, desen Nachricht gerade bearbeitet wird (Ausführen des Portrumpfs) .
Der Prozeß, der die Empfangsanweisung (port statement) ausführt, wird als Meisterprozeß (master) bezeichnet. Der Prozeß der eine Nachricht sendet (connect statement), wird Dienerprozeß (servant) genannt. Die Anweisung zum Lösen einer Verbindung darf nur vom Meisterprozeß ausgeführt werden. Der Zielprozeß (Meister) kann somit den Quellprozeß (Diener) solange blockieren, wie er es für nötig hält.
Es können Variable vom Typ Prozeßidentifikator deklariert werden. Variable dieses Typs können untereinander verglichen oder durch Wertzuweisungen verändert werden. Dabei muß aber die Variable auf der rechten Seite der Zuweisung ebenfalls vom Typ Prozeßidentifikator sein. Jede Nachricht enthält implizit noch einen Parameter vom Typ Prozeßidentifikator und dem Namen 'servant-id'. Dieser Parameter enthält den Absender einer Nachricht.

Beispiel: Semaphoroperationen
Der Prozeß 'Semaphore' erwartet gleichzeitig Nachrichten über den Port P und den Port V. Allerdings kann die Nachricht P nur angenommen werden, wenn der Wert der Variablen S größer Null ist. Ein Prozeß, der dann die Nachricht P sendet, wird blok-

kiert. Die Variable enthält den momentanen Wert der Semaphore. Die Nachrichten P und V werden von Prozessen eines Prozeßfelds mit Namen Q erwartet.
Das Bild 3.17 zeigt den entsprechenden Programmcode.

```
process semaphore;
  var s: integer;
  begin
    s := 1;
    cycle
    // port V;
      servant Q[1..10];
      begin
        !!
      s : = s + 1;endport
    //portP;
      condition s>O;
      servant Q[1..10];
      begin
      !!
      s := s - 1;endport
    endcycle;
    end;

proce  Q[i];
    begin
        .... (processing without resource)
        connect semaphore.P;
        .... (using resource )
        connect semaphore.V;
        .... ( processing without resource )
 end;
```

Bild 3.17 : Beispiel für die Verendung der Comunication Ports nach Mao/Yeh /MAYE8O/

Diskussion:

Für dieses Botschaftskonzept gilt ebenfalls, daß die Anweisungen zum Nachrichtenaustausch über den gesamten Programmtext verstreut sind. Das Kommunikationsverhalten ist schwer zu überblicken (siehe auch ADA, *M0D, CSP).
Bei den Empfangsanweisungen kann noch eine Bedingung angegeben werden. Ist diese Bedingung wahr, dann kann mit dieser Anweisung eine Nachricht angenommen werden. Durch diese Möglichkeit läßt sich das Problem, daß ein Prozeß auf mehrere Nachrichten

wartet, die in beliebiger Reihenfolge eintreffen, einfach lösen.
Sollen in dem Konzept nach Mao/Yeh Serviceprozesse implementiert werden, so entsprechen diese den Meisterprozessen und die aufrufenden Prozesse den Dienerprozessen, d.h. der in der 'Benutzt'-Hierarchie höherliegende Prozeß wird als Prozeß betrachtet, der dem Serviceprozeß nachgeordnet ist. Dies ist eine unübliche Betrachtungsweise.
Es ist nur ein synchroner Botschaftsaustausch möglich. Dadurch wird der Grad an Parallelität eingeschränkt.
Außerdem wird ein Dienerprozeß in jedem Fall blockiert, auch wenn seine Nachricht schon erwartet wird. Dies führt zu zeitaufwendigen Zustandswechseln (Prozeßumschaltungen).

3.2.2.7 Parallele Prozesse in DP

In Distributed Processes (DP), einem Konzept, das von Hansen /HANS78/ vorgeschlagen wurde, besteht ein Programm aus einer festen Anzahl sequentieller, parallel arbeitender Prozesse. Ein Prozeß besteht aus Variablen, Prozeduren und der Initialisierung. Jeder Prozeß läuft auf einem eigenen Prozessor. In einem Programm gibt es keine globalen Variablen, aber globale Prozeduren. Globale Prozeduren gehören zwar zu einem bestimmten Prozeß, können aber von anderen Prozessen auch aufgerufen werden. Ruft ein Prozeß eine globale Prozedur eines anderen Prozesses auf, bezeichnet man dies als externe Anforderung.
Ein Prozeß führt zwei Arten von Anweisungen aus:
- Anweisungen innerhalb der Initialisierung,
- und Anweisungen in globalen Prozeduren.

Diese zwei Arten von Anweisungen werden überlappt ausgeführt. Ein Prozeß beginnt nach dem Start mit dem Durchführen der Initialisierung, die abgeschlossen oder durch eine nicht erfüllte Bedingung unterbrochen wird. Eine Bedingung beschreibt eine Eigenschaft der Variablen. Wird die Initialisierung unterbrochen, werden vorliegende externe Anforderungen begonnen oder weiterbearbeitet. Die Bearbeitung von globalen

Prozeduren kann ebenfalls durch nicht erfüllte Bedingungen unterbrochen werden. Die Initialisierung oder unterbrochene externe Anforderungen werden weitergeführt, wenn die entsprechende Bedingung wahr wird. Eine Bedingung kann dadurch wahr werden, daß durch andere externe Anforderungen die Variablen des Prozesses verändert werden. Die Überlappung der Initialisierung und die Bearbeitung von externen Anforderungen wird beliebig fortgesetzt oder solange, bis die Initialisierung beendet ist. Dann bearbeitet ein Prozeß nur noch externe Anforderungen. Ein Prozeß ist solange aktiv, d.h. führt die Initialisierung durch bzw. bearbeitet externe Anforderungen, bis die Initialisierung unterbrochen oder beendet wird und bis die Bearbeitung aller externer Anforderungen durch nicht erfüllte Bedingungen ebenfalls unterbrochen wurde. Ein Prozeß wird dann solange blockiert, bis weitere externe Anforderungen eintreffen. Ruft ein Prozeß eine globale Prozedur in einem anderen Prozeß auf, wird er solange blockiert, bis der gerufene Prozeß diese externe Anforderung abgeschlossen hat.
Eine Prozedur wird definiert durch seine Eingabeparameter, Ausgabeparameter, den benötigten lokalen Daten und einer Anweisungsfolge, dem Prozedurrumpf.
Aussehen einer Prozedurdeklaration:

```
proc NAME (INPUTPARAMETER   OUTPUT PARAMETER)
     DEKLARATION LOKALER VARIABLER
     ANWEISUNGSFOLGE
```

Ein Prozeß P ruft eine globale Prozedur R, die im Prozeß Q deklariert ist, mit folgender Anweisung auf:

```
call Q.R (INPUTPARAMETER   OUT PUT PARAMETER);
```

Im Konzept DP kennt also der rufende Prozeß den gerufenen Prozeß. Umgekehrt kennt der gerufene Prozeß nicht den rufenden. Beim Aufruf einer globalen Prozedur werden zuerst die Eingabeparameter übergeben. Wenn die externe Anforderung fertig bearbeitet wurde, werden die Rückgabeparameter an den rufenden Prozeß gegeben.

Um nichtdeterministisches Programmverhalten formulieren zu können, gibt es zwei Typen von Anweisungen:
- sogenannte Guarded Commands (GC)
- und sogenannte Guarded Regions (GR).

Ein GC ermöglicht es einem Prozeß, in Abhängigkeit vom Zustand der lokalen Daten (der Wertebelegung der Variablen), eine Anweisungsfolge auszuwählen. Ist keine der Alternativen möglich, da keine der angegebenen Bedingungen auf die Variablen zutrifft, so wird die Ausführung des GC entweder abgebrochen (leere Anweisung) oder eine Ausnahmebehandlung (exception) angestoßen.
Bei den Guarded Commands unterscheidet man zwei Arten:
- sogenannte If-Anweisungen
- und sogenannte Da-Anweisungen.

Bei einer If-Anweisung wird eine der möglichen Alternativen ausgewählt (wahre Bedingung) und die dazugehörige Anweisungsfolge ausgeführt. Wenn keine Bedingung zutrifft, wird entsprechend im Programm fortgefahren.
Eine Da-Anweisung ist ähnlich wie die If-Anweisung aufgebaut. Sie wird solange wiederholt, bis keine der Bedingungen wahr ist. Dann wird die nächste Anweisung des Programms ausgeführt.
Die Syntax der zwei Typen von Guarded Commands:

```
if B1:S1/B2:S2/ ..... end
do  B1 :S1 /B2:S2? .....end
```

Mit Bi wird eine Bedingung und mit Si eine Anweisungsfolge bezeichnet.
Mit den GR-Anweisungen ist es einem Prozeß möglich, auf einem bestimmten Zustand seiner Variablen zu warten. Bei den Guarded Regions unterscheidet man ebenfalls zwischen zwei Typen:
- den sogenannten When-Anweisungen
- und den sogenannten Cycle-Anweisungen.

Bei einer When-Anweisung wartet ein Prozeß solange, bis eine der angegebenen Alternativen ausgeführt werden kann. Danach wird entsprechend dem Programm weitergemacht.

Eine Cycle Anweisung ist eine When-Anweisung, die unendlich oft wiederholt wird.
Syntax der Guarded Regions:

```
when  B1:S1/B2:S2/ ..... end;
cycle B1 :S1 /B2:S2/ ..... end;
```

Dabei sei Bi eine Bedingung und Si eine Anweisungsfolge.

Beispiel:
Der Prozeß stream liest mit der globalen Prozedur 'input' des Prozesses 'cardreader' (der Code dieses Prozesses ist nicht angegeben) Lochkarten ein und gibt den Inhalt der Lochkarten zeichenweise an den Prozeß 'buffer'. Dazu ruft der Prozeß 'stream' die globale Prozedur 'send' im Prozeß buffer auf.

```
process stream
lr  array(80Jchar; n, i: int
do true:
  call cardreader.input(b)
  ü b • blanldine: skip 1
    b Ţ' blanldine: i :- l; n :- 80
      do b[n] = space: n :- n - 1 end
      do i :S n call buffer.send(b[i]); i :- i + 1 end
  end
  call buffer.send(newline)
end.
```

Bild 3.18: Beispiel für ein DP-Programm /HANS78/

Diskussion:

In DP wird von der Voraussetzung ausgegangen, daß jeder Prozeß auf einem eigenen Prozessor läuft. Dies schränkt die Möglichkeiten für die Strukturierung eines Programms in einzelne Prozesse ein, bzw. eine Änderung der Prozeßanzahl erfordert eine Änderung der Hardwarestruktur.
In DP ist einem Prozeß nicht bekannt, welcher Prozeß eine externe Anforderung stellt. Dies bringt Schwierigkeiten, wenn der aufrufende Prozeß noch eine Antwort bekommt (durch eine entsprechende externe Anforderung), nachdem seine externe

Anforderung abgeschlossen wurde. Die Identität eines rufenden Prozesses muß durch einen Parameter mitgeführt werden. Dadurch, daß einem gerufenen Prozeß der rufende Prozeß nicht bekannt ist, kann sich ein Prozeß nicht vor unbefugten externen Anforderungen schützen.
Ein Prozeß, der eine externe Anforderung stellt, wird solange blockiert, bis diese abgeschlossen ist. Dadurch wird der Grad an Parallelarbeit erheblich eingeschränkt.
Nichtdeterministisches Programmverhalten wird in DP durch Guarded Commands und Guarded Regions beschrieben. In DP gibt es aber noch eine Quelle für nichtdeterministisches Verhalten. Ein Prozeß wechselt zwischen der Initialisierung und der Bearbeitung externer Anforderungen ständig hin und her. Dieser Wechsel ist nicht in der Programmnotation explizit sichtbar /WELS80/. Dadurch wird es schwierig festzustellen, welche Folgen von externen Anforderungen möglich sind.
Ein Prozeß kann nicht mehrere externe Anforderungen gleichzeitig oder alternativ stellen (analog dem Senden mehrerer Nachrichten).

4 Verteilte Systeme und verteilte Programme

Im folgenden soll versucht werden, auf der Basis der Definitionen des Begriffs 'verteiltes System', wie er in /BOCH79/ und /DANI81/ verwendet wird, die Begriffe 'verteiltes System' und 'verteiltes Programm' voneinander abzugrenzen. Dies ist nötig, um besser auf die Problematik der Spezifikation und Implementierung von Anwenderprogrammen auf verteilten Systemen eingehen zu können. Denn Ziel dieser Arbeit ist der Entwurf von verteilten Programmen und nicht der von verteilten Systemen.

4.1 Zusammenhang zwishen verteilten Systemen und verteilten Programmen

Bei üblichen Monoprozessoranlagen wird zwischen der Systemebene und der Anwenderebene unterschieden. Eine solche Unterscheidung kann auch zwischen verteiltem System und verteilten (Anwender-)Programm gemacht werden. Die Systemebene eines verteilten Systems besteht aus den Systemkomponenten und dem Übertragungs- oder Kommunikationsmechanismus (siehe Bild 4.1).

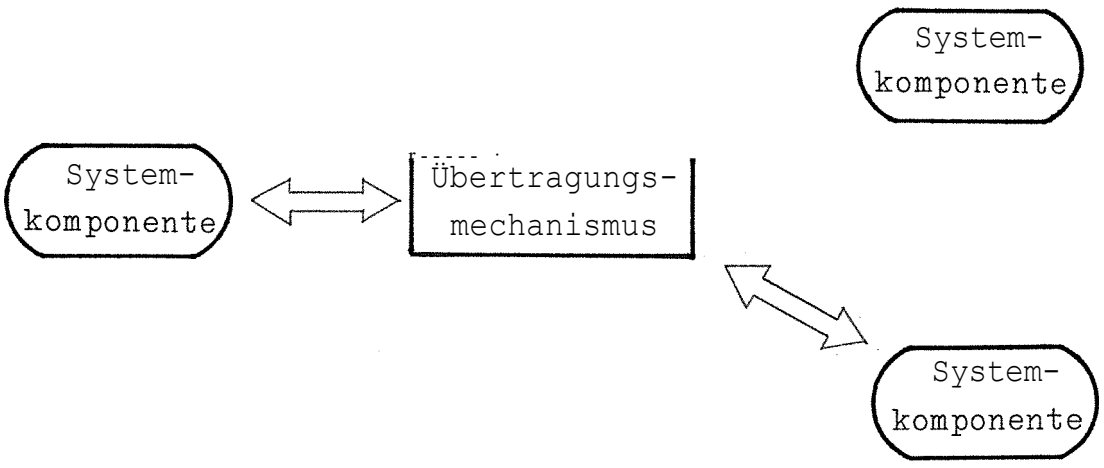


Bild 4.1: Bestandteile eines verteilten Systems

Eine Komponente stellt eine rechnende Einheit (z.B. Rechner, intelligentes Terminal) zusammen mit entsprechenden Programmen

dar, die es erlaubt, den Übertragungsmechanismus zu nutzen (entsprechende Betriebssystemteile). Die Systemkomponenten erhalten nicht nur Eingaben über den Kommunikationsmechanismus, sondern auch von der direkten Umgebung (z.B. Interrupts). Aufgabe der Systemkomponenten ist es also unter anderem, Botschaften, die über den Kommunikationsmechanismus von anderen Systemkomponenten eintreffen, und die Eingaben aus der direkten Umgebung den Anwederprogrammen zugänglich zu machen. Im Übertragungsmechanismus soll die Realisierung der Datenübertragung verborgen sein z.B. Netztopologie, Vermittlungstechnik, Protokoll.

Die Anwenderebene enthält das vom Benutzer eines verteilten Systems erstellte Programm. Bei der Erstellung von Anwenderprogrammen für verteilte Systeme wird das Anwenderprogramm in einzelne Anwenderprozesse (Benutzerprozesse) gegliedert. Diese Prozesse können auf den einzelnen Komponenten eines verteilten Systems laufen. Es ist durchaus möglich, daß mehrere Benutzerprozesse auf einer Systemkomponente laufen. Ein solches Programm soll als verteiltes Programm bezeichnet werden. Verteilte Programme sind also im allgemeinen parallele Programme. Die Anwenderprozesse benutzen die Möglichkeiten des verteilten Systems, um ihre Aktivitäten zu synchronisieren, bzw. um miteinander oder mit der Umgebung zu kommunizieren. Auf der Anwenderebene wird die Umgebung ebenfalls als eine Menge von Prozessen aufgefaßt. Diese sogenannten 'Umgebungsprozesse' kommunizieren mit den Benutzerprozessen. Die Umgebungsprozesse benutzen dazu nicht den Kommunikationsmechanismus, sondern andere Mechanismen (z.B. Interrupts), auf die hier nur durch ein Beispiel kurz eingegangen werden soll. Ein solcher Umgebungsprozeß kann z.B. eine Schrittmotorsteuerung sein, die den Sollwert für die Motorposition von einem Anwenderprozeß über eine Digitalausgabe mitgeteilt bekommt und dem Benutzerprozeß über einen Interrupt meldet, wann der Schrittmotor die gewünschte Position erreicht hat.

Dieses Einbeziehen der Umgebung ist vor allem für Anwenderprogramme wichtig, die zur Steuerung von technischen Prozessen dienen.

Bild 4.1 zeigt zusammenfassend den Zusammenhang zwischen ver-

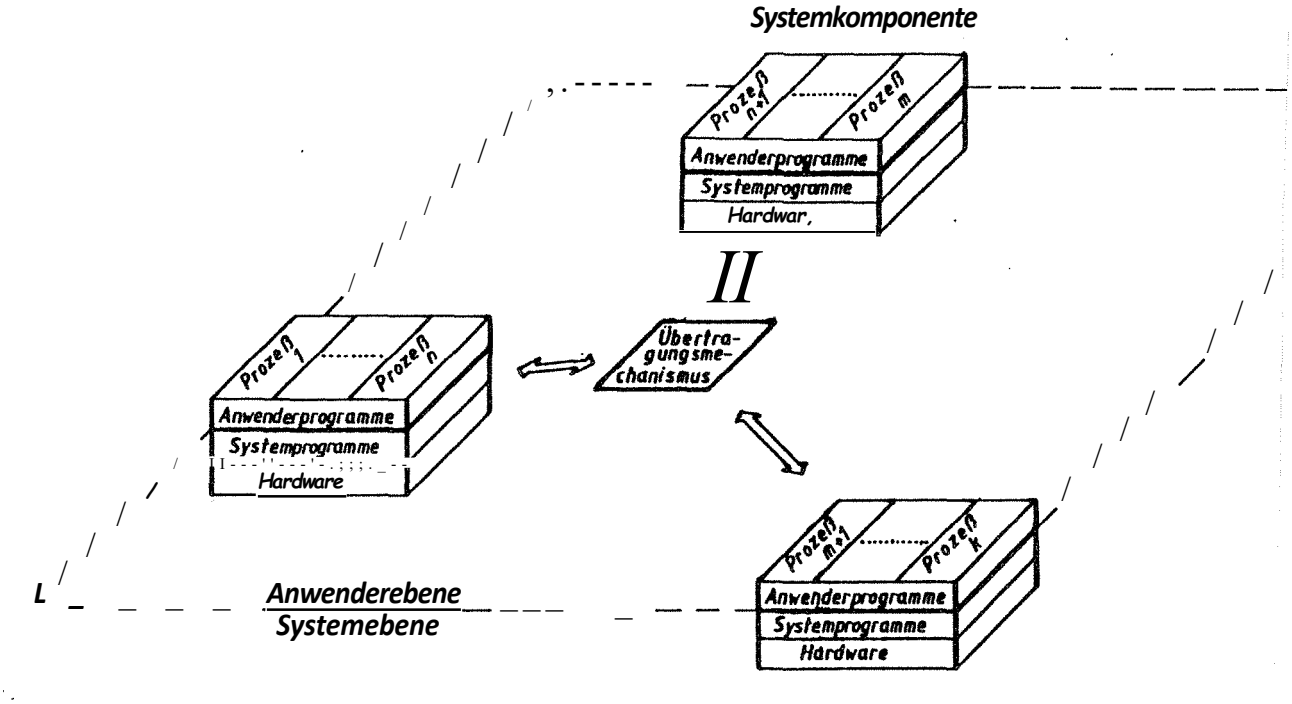


Bild 4,2 Zusammenhang verteiltes System - verteiltes Programm

4,2 Verteilte Systeme

4.2.1 Aufbau von verteilten Systemen

Im folgenden soll kurz ein Architekturmodell für verteilte Systeme vorgestellt werden /DAVI81 /. Dadurch soll der Aufbau der Systemebene von verteilten Systemen veranschaulicht werden. Allerdings soll nicht näher auf die allgemeinen Schwierigkeiten beim Entwurf von verteilten Systemen eingegangen werden, dazu sei auf die entsprechende Literatur verwiesen (/BOCH79/, /DAVI81/, /CARL78/). In dieser Arbeit sollen nur die Probleme bei der Implementierung von bestimmten Klassen von Synchronisations- und Kommunikationsmechanismen, die die Systemebene eines verteilten Systems dem Anwender zur Verfügung stellt, diskutiert werden. Bild 4.3 zeigt das Architekturmodell. Das Modell hat drei Dimensionen. Die senkrechte

turmodell. Das Modell hat drei Dimensionen. Die senkrechte Achse zeigt die Systemebene eines verteilten Systems als eine Menge von logischen Schichten. Die Probleme, die in allen logischen Schichten auftreten, sind an der nach rechts zeigenden Achse aufgetragen. An der dritten Achse sind die allgemeinen Implementierungs- und Optimierungsprobleme aufgetragen.

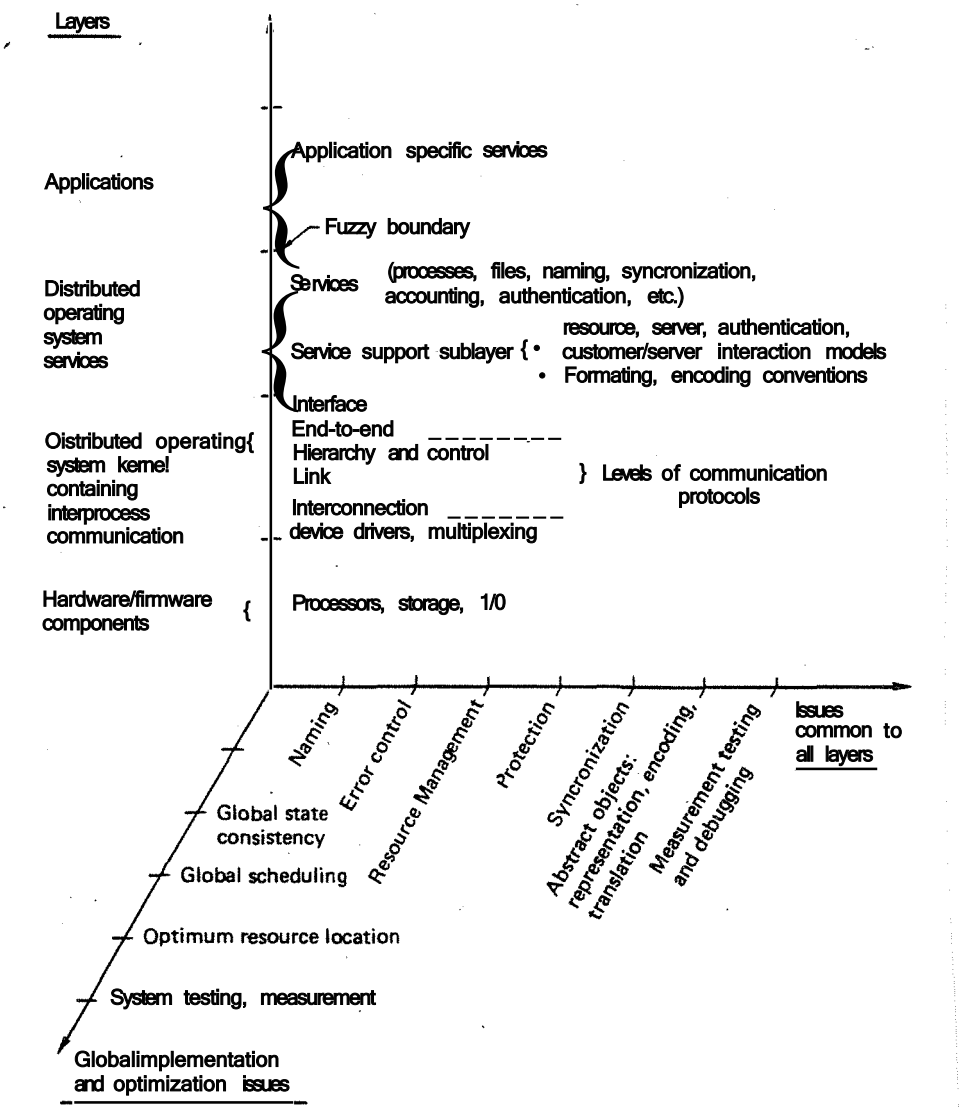


Bild 4.2 Architekturmodell von verteilten Systemen /DAVI81/

4.2.2 Anwendung von verteilten Systemen

4.2.2.1 Anwendung in der 'konventionellen' Programmierung

Von einem mehr technischen Standpunkt unterscheidet man nach /KERN81/ drei Hauptanwendungszwecke von verteilten Systemen.

- Datenverbund
  Räumlich von einander getrennte Benutzer wollen auf eine gemeinsame Datenbank zugreifen. Beispiele dafür sind Anwendungen bei Banken, Reisebüros und Organbanken.
- Funktionsverbund
  Bestimmte Spezialprogramme können von entfernten Datenstationen aufgerufen werden. Beispiele dafür sind Programme, die vor unbefugter Weitergabe geschützt werden sollen.
- Betriebsmittel- und Lastverbund
  Die Rechnerleistung soll dorthin gebracht werden, wo sie tatsächlich benötigt wird.

In der Praxis kommt natürlich keine dieser Hauptverwendungszwecke in reiner Form vor. Im allgemeinen werden als Anwendungen Mischformen auftreten.

4.2.2.2 Anwendung von verteilten Systemen bei der Steuerung technischer Prozesse

Die zahlreichen Möglichkeiten, Mini- und Mikrorechner einzusetzen, haben dazu geführt, daß verteilte Systeme in steigendem Maße in der Prozeßautomatisierung eingesetzt werden. Verteilte Systeme werden unter anderem bei der Erfassung von Betriebsdaten, bei der Laborautomation, bei der Entwicklung und Konstruktion von Fertigungsteilen (CAD), bei der Prozeßleittechnik und bei der Fernwirktechnik angewendet /LEVI81/.
Nach /LEVI81/ sprechen im wesentlichen vier Gründe für den Einsatz von verteilten Systemen bei der Prozeßautomatisierung:

1. Allgemeine Gründe
   Hier werden in /LEVI81/ die bereits in Kapitel 4.3.1 erwähnten Argumente aufgeführt, allerdings auf dem speziellen Hintergrund der Prozeßautomatisierung mit verteilten Systemen.
   - Datenverbund
     Daten, die räumlich verteilt bzw. auf verschiedenen Systemkomponenten anfallen, können von allen am Verbund beteiligten Komponenten ausgenutzt werden.
   - Funktionsverbund
     Die technischen Prozesse sind zu komplex und räumlich ausgedehnt, um von einem Rechner gesteuert und überwacht zu werden. Einzelne Rechner erledigen spezielle Aufgaben.
   - Last- und Betriebsmittelverbund
     Die anfallenden Arbeiten können auf verschiedene Systemkomponenten verteilt werden. Bestimmte Peripheriegeräte können von mehreren Systemkomponenten aus angesprochen werden.

2. Die Struktur des Problems, d.h. des zu steuernden technischen Prozesses, kann auf die Struktur des verteilten Systems abgebildet werden. Bestimmte Aufgaben können einen bestimmten u.U. spezialisierten Rechner zugeordnet werden.

3. Die Zuverlässigkeit der Hardware und der Software wird durch die Verteilung auf mehrere Rechner erhöht. Vor allem wenn durch eine entsprechende Redundanz Rechner die Aufgaben eines ausgefallenen Rechners wenigstens teilweise oder sogar vollständig übernehmen können.

4. Ein verteiltes System kann leichter an ein geändertes Aufgaben- oder Leistungsspektrum angepaßt werden.

4.3 Verteilte Programme

4.3.1 Synchronisations- und Kommunikationsmechanismen für verteilte Programme

Verteilte Programme sind, wie bereits erwähnt, im allgemeinen parallele Programme. Das wesentliche Unterscheidungsmerkmal von parallelen Programmen gegenüber sequentiellen Programmen ist, daß sich die Prozesse eines parallelen Programms synchronisieren bzw. miteinander kommunizieren müssen.
In Kapitel 2 wurden verschiedene Strukturierungsaspekte für parallele Programme und die entsprechenden Eigenschaften der Synchronisations- und Kommunikationskonzepte diskutiert.
In Kapitel 3 wurden verschiedene konkrete Synchronisations- und Kommunikationskonzepte vorgestellt, die entweder auf gemeinsamen Objekten oder auf Botschaftsmechanismen beruhen.
In diesem Abschnitt soll nun untersucht werden,

- welche Probleme es gibt, wenn in der Systemebene eines verteilten Systems verschiedene Synchronisations- und Kommunikationskonzepte implementiert werden.
- welche Effizienzprobleme es bei verteilten Programmen durch die Strukturierungsaspekte, die durch die einzelnen Synchronisations- und Kommunikationsaspekte unterstützt werden (gemeinsame Objekte, keine gemeinsamen Objekte), geben kann.

An die Synchronisations- und Kommunikationskonzepte für verteilte Programme werden somit folgende Forderungen gestellt:

- Bei deren Implementierung auf verteilten Systemen soll die Leitungsbelastung minimal sein.
- Beim Entwurf von verteilten Programmen müssen die Strukturierungsmöglichkeiten eine Programmstruktur nahelegen, die die Leitungsbelastung gering hält.

4.3.1 .1 Verteilte Programme mit gemeinsamen Objekten

4.3.1.1 .1 Allgemeine Probleme

Synchronisationskonzepte, die auf gemeinsamen Objekten beruhen (Monitore, Semaphore), sind auf Rechensysteme mit gemeinsamem Speicher ausgelegt. Um diese Konzepte in verteilten Systemen einsetzen zu können, müssen sie durch Hilfsfunktionen implementiert werden. Diese Aussage soll am Beispiel der Monitore und Semaphore verdeutlicht werden (/LEVI81 /, /KEME79/, /AMMA81 /).

Implementation von Monitoren

Ein Monitor kann mit Hilfe eines Stellvertreterprozesses und einer sogenannten Monitorhülle implementiert werden.
In Bild 4.4 ist diese Lösung dargestellt.

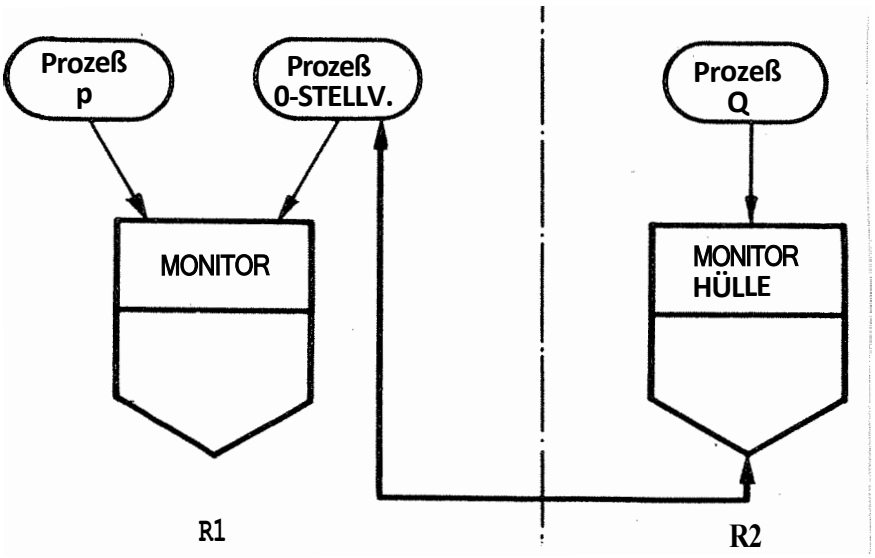


Bild 4.4 : Implementierungsmöglichkeit von Monitoren für verteilte Systeme

Ruft ein Prozeß eine Funktion aus einem Monitor auf, der sich auf einer anderen Komponente des verteilten Systems befindet, so geht dieser Aufruf an die Monitorhülle. Die Monitorhülle veranlaßt, daß der Aufruf durch den Übertragungsmechanismus an

die entsprechende Systemkomponente (Definition siehe Abschnitt 4.1) übertragen wird. Dort wird der Auftrag von einem sogenannten 'Stellvertreterprozeß' angenommen. Der Stellvertreterprozeß führt den Aufruf aus und gibt die Ergebnisse, falls er nicht im Monitor blockiert wird, an die Monitorhülle zurück. Diese reicht die Ergebnisse an den aufrufenden Prozess weiter, der dann fortgesetzt werden kann.
Da der Stellvertreterprozeß im Monitor blockiert werden kann, muß auf einer Systemkomponente für jeden Prozeß, der auf einer anderen Systemkomponente läuft und Funktionen von Monitoren aufruft, die sich auf der betrachteten Systemkomponente befinden, ein Stellvertreterprozeß vorhanden sein. Wenn ein Prozeß verschiedene Monitore auf verschiedenen Systemkomponenten benutzt, muß auf jeder dieser Systemkomponenten ein Stellvertreterprozeß existieren.
Auf jeder Systemkomponente, auf der ein Prozeß läuft, der Funktionen eines Monitors aufruft, der sich auf einer anderen Systemkomponente befindet, muß eine dazugehörige Monitorhülle existieren.
Für die konkrete Implementierung können im wesentlichen zwei Wege beschritten werden:

- Der Speicherplatz für die Stellvertreterprozesse und der Platz für die Monitorhüllen wird vom Übersetzer, vom Binder oder vom Lader statisch angelegt /KEME79/.
- Die Stellvertreterprozesse und die Monitorhüllen werden dynamisch erzeugt und der benötigte Platz wird von einer Speicherverwaltung angefordert.

Statische Speicherplatzreservierung:

Zunächst sollen die Probleme diskutiert werden, die entstehen, wenn der Speicherplatz für die Stellvertreterprozesse statisch angelegt wird. Dabei soll als erstes auf die Schwierigkeiten eingegangen werden, wenn der Speicherplatz vom Übersetzer reserviert wird. Danach soll diskutiert werden, welche Probleme sich ergeben, wenn der Speicherplatz vom Binder oder vom Lader reserviert wird.
Soll der Speicherplatz statisch durch den Übersetzer reser-

viert werden, so stellt diese Lösung einige Anforderungen an die Sprache bzw. das Übersetzungssystem. Die Anforderungen an eine Sprache und an ein Übersetzungssystem werden für diesen Fall in /KEME79/ am Beispiel von Concurrent Pascal gezeigt. Hier wird allerdings davon ausgegangen, daß ein verteiltes Programm als Ganzes übersetzt wird. Die Probleme verschärfen sich, wenn es möglich sein soll, daß Teile eines Programms getrennt übersetzt werden. Dies ist aber eine Forderung, wie sie an alle modernen Programmiersprachen gestellt wird.
Im folgenden soll versucht werden, einige dieser Probleme zu erläutern.
Wird ein Monitor übersetzt, so muß bekannt sein, wieviele Prozesse Funktionen dieses Monitors aufrufen, damit Speicherplatz für die Stellvertreterprozesse freigehalten wird.
Ein ähnliches Problem ergibt sich, wenn ein Prozeß übersetzt wird, der eine Monitorfunktion aufruft. In diesem Fall weiß der Übersetzer nicht, auf welcher Systemkomponente sich der Monitor einmal befinden wird. Befindet sich der zu übersetzende Prozeß auf demselben Prozessor wie ein von diesem benutzter Monitor, so hat bei der Übersetzung eines Monitoraufrufs ein Eintrag in die Querverweisliste zu erfolgen, ansonsten muß der Aufruf der entsprechenden Monitorhülle abgelegt werden (Bei diesen Überlegungen sol nicht mit berücksichtigt werden, die Übernahme von Funktionen auf andere Komponenten während der Laufzeit, z.B. bei einem Fehlerfall).
Es gibt zwei Möglichkeiten, dem Übersetzer diese Informationen zukommen zu lassen:

- Die Programmiersprache wird erweitert durch einen Anweisungstyp, der die Verteilung der Programmoduln auf die einzelnen Komponenten des verteilten Systems beschreibt.
- Der Benutzer gibt dem Übersetzer die benötigten Informationen durch Parameter.

Wird die Programmiersprache durch Anweisungen erweitert, die die Programmverteilung beschreiben (wie bei /KERM79/), so hat dies den Vorteil, daß die Verteilung des Programms auf die einzelen Komponenten im Programmtext dokumentiert wird. Allerdings ist nun eine Programmiersprache entstanden, die ein

Synchronisationskonzept benutzt, das für Systeme mit gemeinsamem Speicher entwickelt wurde, andererseits aber Anweisungstypen enthält, die eindeutig auf eine Programmiersprache für verteilte Systeme hinweisen.
Erhält der Übersetzer die Informationen über die Verteilung der Programmoduln auf die einzelnen Systemkomponenten durch Parameter, so hat dies den Vorteil, daß ein Programm verwendet werden kann, unabhängig davon, ob es auf einem verteilten oder einem konventionellen System laufen soll. Je nach verwendetem System muß es nur neu übersetzt werden, aber ohne daß vorher der Programmtext geändert werden muß. Allerdings muß sich zu jeder abgelegten Objektdatei gemerkt werden, mit welchen Parametern sie der Übersetzer erzeugt hat. Die Eigenschaften eines übersetzten Programmes hängen also nicht nur vom Programmtext sondern auch von Übersetzungsparametern ab, was nicht sehr dokumentationsfreundlich erscheint.
Statt des Übersetzers könnte auch der Binder die notwendigen Platzreservierungen vornehmen. Der Binder benötigt dann die Informationen über die Verteilung der Programmmodule auf die einzelnen Systemkomponenten. Dies würde vor allem die Struktur des Übersetzers verkomplizieren, da dieser alle Möglichkeiten offenhalten müßte, d.h. der abgesetzte Code müßte unabhängig davon sein, ob sich ein Prozeß und ein von diesem verwendeter Monitor auf demselben oder verschiedenen Prozessoren befindet. Ebenso würde die Aufgabe des Binders komplizierter werden. Denn dieser müßte dann nicht nur die Querverweislisten bearbeiten, sondern auch aufgrund der vom Benutzer angegebenen Programmverteilung den Programmcode ergänzen (z.B. Hinzufügen einer entsprechenden Monitorhülle).
Für die Dokumentationsfreundlichkeit gelten dieselben Argumente, wie beim Festlegen der Programmverteilung durch Übersetzerparameter.
Ähnliche Überlegungen, wie bei der Verteilung des Programms beim Binden, können für die Programmverteilung beim Laden angestellt werden.
Der benötigte Speicherplatz für die Monitorhüllen und Stellvertreterprozesse kann bei einer statischen Anzahl von Prozessen bzw. Monitoren abgeschätzt werden. Auf jeder Systemkornpo-

nente wird soviel Speicherplatz reserviert, wie für die Monitorhüllen und Stellvertreterprozesse aller Prozesse und Monitore eines verteilten Programms benötigt wird. Die Berechnung dieser oberen Grenze kann im allgemeinen nur vom Binder und Lader durchgeführt werden, da dazu alle Programmoduln notwendig sind.
Um unnötige Leitungsbelastungen zu vermeiden, muß jeder Systemkomponente bekannt sein, auf welchen Systemkomponenten sich welche Prozesse bzw. welche Monitore befinden. Jede Monitorhülle muß wissen, an welche Systemkomponente der Monitoraufruf weitergegeben werden soll, bzw. der Stellvertreterprozeß muß wissen, an welche Systemkomponente das Ergebnis eines Monitoraufrufs gesendet werden soll.
Dieses Adresierungsproblem wird durch diese Speicherplatzabschätzung nicht gelöst.

Dynamische Speicherplatzreservierung

Statt den Speicherplatz für die Stellvertreterprozesse und die Monitorhüllen durch den Übersetzer, Binder oder Lader statisch freizuhalten, können die Stellvertreterprozesse und die Monitorhüllen dynamisch erzeugt werden und der dafür benötigte Speicherplatz von einer Speicherverwaltung angefordert werden. Dies setzt natürlich voraus, daß auf den Systemkomponenten eine Speicherverwaltung zur Verfügung steht.
Jeder Komponente muß dann entweder bekannt sein, welche Monitore sie beherbergt, oder sie muß wissen, auf welcher Systemkomponente sich die einzelnen Monitore befinden.
Zunächst soll der Fall diskutiert werden, daß eine Systemkomponente 'weiß', welche Monitore sie beherbergt. Dazu muß vom Übersetzer zu jedem Programmodul eine Liste angelegt werden, in der steht, welche Monitore sich in diesem Modul befinden. Der Lader kann dann die Monitorlisten der Programmodule, die auf eine Komponente geladen werden sollen, dem System zur Verfügung stellen. Dieses Problem scheint also relativ einfach lösbar.
Ruft ein Prozeß eine Funktion eines Monitors auf, der sich auf einer andern Systemkomponente befindet, so wird bei den ein-

zelnen Systemkomponenten angefragt, ob sie den gesuchten Monitor enthalten. Dieses Vorgehen ist sehr zeitaufwendig und belastet den Übertragungsmechanismus.
Die andere Möglichkeit ist, daß einer Systemkomponente bekannt ist, auf welcher Komponente sich welcher Monitor befindet. Es ergibt sich das Problem, wie die Informationen über die Verteilung der Monitore beschafft werden können. Zum Beschaffen dieser Informationen können die bereits bei der statischen Speicherplatzreservierung diskutierten Möglichkeiten verwendet werden. Die dort aufgeführten Vor- und Nachteile gelten auch, wenn der Speicherplatz für die Stellvertreterprozesse und die Monitorhüllen dynamisch angefordert wird.
Es soll nun angenommen werden, daß der Systemebene bekannt ist, auf welcher Systemkomponente sich welcher Monitor befindet. Allerdings soll eine Komponente nicht wissen, welche Prozesse die in ihr enthaltenen Monitore benutzen.
Werden nun die Stellvertreterprozesse und die Monitorhüllen dynamisch erzeugt, so kann sich u.U. das verteilte Programm verklemmen, ohne daß aus dem Programmtext auf eine mögliche Verklemmung geschlossen werden kann. Es soll versucht werden, dies an einem Beispiel zu verdeutlichen (siehe Bild 4.5).

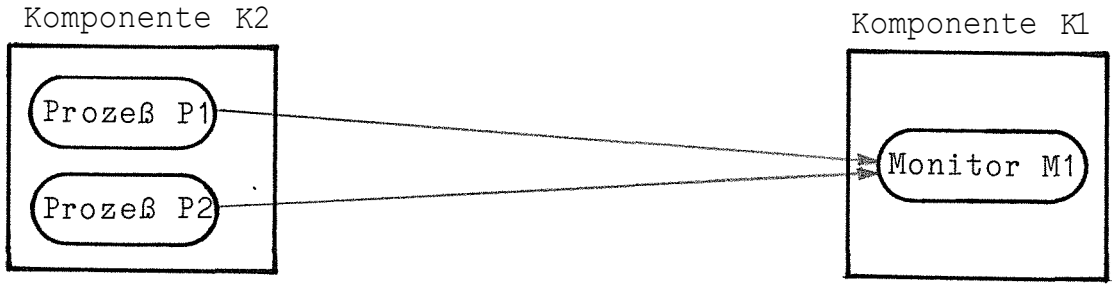


Bild 4.5 : Beispiel für die Verklemmung von Prozessen bei der dynamischen Erzeugung von Stellvertreterprozessen

Zwei Prozesse P1 und P2 auf der Komponente K2 benutzen Funktionen aus dem Monitor M1, der sich auf der Komponente K1 befindet. Auf der Komponente K1 ist soviel Speicher frei, daß ein Stellvertreterprozeß erzeugt werden kann. Auf K2 ist Platz für zwei Monitorhüllen. P1 ruft nun eine Funktion in M1 auf. Es wird eine entsprechende Monitorhülle erzeugt, dann der

entsprechende Stellvertreterprozess, der schließlich die Funktion aufruft. Der Stellvertreterprozeß von P1 wird im Monitor M1 blockiert, da er darauf wartet, daß P2 eine Funktion aufruft, die ihn wieder deblockiert. P2 ruft diese Funktion ebenfalls auf. Es wird die Monitorhülle erzeugt, aber auf K1 kann kein Stellvertreterprozeß erzeugt werden, da kein Speicherplatz mehr frei ist. Das System verklemmt sich, ohne daß aus dem Programmtext auf eine Verklemmung geschlossen werden kann. Ein ähnliches Problem tritt auf, wenn der Platz für die Monitorhüllen nicht ausreicht.
Im Realfall dürfte eine Mischung aus Speicherplatzbedarf für Stellvertreterprozesse und Monitorhüllen zur Verklemmung führen. Diese Verklemmungen können nur verhindert werden, wenn entsprechend der Programmstruktur und der Aufteilung der einzelnen Prozesse auf die Systemkomponenten auf jeder Komponente genügend Speicherplatz zur Verfügung steht.
Der Speicherplatzbedarf je Komponente ergibt sich aus der maximal möglichen Anzahl von Monitorhüllen und Stellvertreterprozessen, die gleichzeitig auf dieser Komponente existieren können. Um diesen Speicherplatzbedarf berechnen zu können, muß allerdings bekannt sein, welche Monitore von welchen Prozessen benötigt werden und welche Prozesse welche Monitore aufrufen. Allerdings kann der maximal benötigte Speicherplatz für die Stellvertreterprozesse und Monitorhüllen, wie bei der statischen Speicherplatzzuteilung, abgeschätzt werden. Dadurch entfällt dann aber der Vorteil der dynamischen Speicherverwaltung, daß weniger Speicherplatz benötigt wird. Die Adressierungsprobleme (auf welchen Systemkomponenten befinden sich welche Monitore bzw. Prozesse) können mit einer Speicherplatzabschätzung ebenfalls nicht gelöst werden.
Damit hat man dieselben Probleme wie bei der statischen Reservierung. Allerdings hat die dynamische Speicherplatzzuteilung noch den Nachteil, daß sie wesentlich aufwendiger ist als eine statische.

Ein weiteres Problem bei Monitoren auf verteilten Systemen ist die übliche Ausfallrate des Übertragungsmechanismus.
Bei der oben geschilderten Implementierung eines Monitors auf

verteilten Systemen kann ein Prozeß durch den Aufruf einer Monitorfunktion, die sich in einem Monitor befindet, der auf einem anderen Prozessor läuft, für immer blockiert werden. Wird nämlich die Leitung unterbrochen, nachdem der Funktionsaufruf übertragen wurde, so kann eine Fertigmeldung nicht mehr an den aufrufenden Prozeß zurückgegeben werden. Der aufrufende Prozeß bleibt blockiert, außer das System stellt die Leitungsunterbrechung fest und deblockiert ihn. Dem aufrufenden Prozeß sollte dann aber angezeigt werden, daß der Aufruf der Monitorfunktion nicht regulär abgeschlossen wurde. In den bisher vorgeschlagenen Monitorkonzepten sind keine Mechanismen zur Fehlerbehandlumg vorgesehen d.h. für verteilte Systeme müßten spezielle Monitore entworfen werden, die die übliche Fehleranfälligkeit des Übertragungmechanismus berücksichtigen.

Neben den oben geschilderten Implementierungsproblemen für Monitore auf verteilten Systemen kann es auch zu unnötigen Leitungsbelastungen bei Programmen mit Monitoren kommen, wenn beim Entwurf des Programms nicht mit berücksichtigt wird, daß ein verteiltes Programm entstehen soll /WEBE83/. Es soll zunächst versucht werden, diese Problematik anhand eines Beispiels zu verdeutlichen.
Ein Programm besteht aus den Prozessen P1, P2, P3 und P4 sowie einem Monitor M. Die Prozesse P1 und P2 sollen Serviceprozesse sein, die den Prozessen P3 und P4 irgendwelche Dienstleistungen zur Verfügung stellen. Die Dienstleistungsanforderungen werden den Prozessen P1 und P2 über den Monitor M mitgeteilt. Die Dienstleistungsanforderungen vom Prozeß P3 werden alle vom Prozeß P1 bearbeitet, während Anforderungen vom Prozeß P4 überwiegend vom Prozeß P2 und in einigen Fällen vom Prozeß P1 bearbeitet werden. Der Prozeß P4 weiß aber nicht, welcher Serviceprozeß konkret welche Anforderungen bearbeitet (z.B. übernimmt der Prozeß P1 einige Aufträge, um den Prozeß P2 zu entlasten). Dieses Programm soll nun auf ein verteiltes System gebracht werden, das aus zwei Systemkomponenten besteht. Aus irgendwelchen Gründen sollen die Prozesse P1 und P3 auf der einen Systemkomponente und die Prozesse P2 und P4 auf der anderen ablaufen. Das Bild 4.6 zeigt die Struktur des Bei-

spielprogramms.

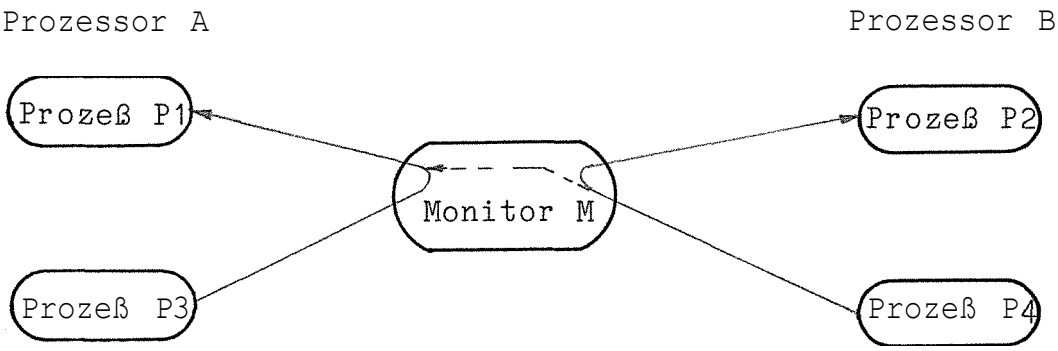


Bild 4.6 : Struktur des Beispielprogramms

Es muß nun entschieden werden, auf welcher Systemkomponente der Monitor M untergebracht werden soll. Befindet sich der Monitor M auf dem Prozessor A (Systemkomponente A), so müssen alle Anforderungen des Prozesses P4 zunächst an den Prozessor A gesendet werden, um dann in den meisten Fällen an den Prozessor B zurückgesendet zu werden, damit sie vom Prozeß P2 bearbeitet werden. Es ergibt sich eine unnötige Leitungsbelastung, die die Leistungsfähigkeit eines Anwenderprogramms erheblich einschränken kann.
Wenn der Monitor M auf der Systemkomponente B untergebracht ist, ergeben sich dieselben Probleme.
Die Programmstruktur in diesem Beispiel führt bei einem verteilten System zu einer ineffizienten Lösung.
Wird ein Programm entworfen, das als gemeinsame Objekte Monitore verwendet, so muß schon beim Entwurf berücksichtigt werden, ob dieses Programm für ein verteiltes oder kein verteiltes (zentralistisches) System gedacht ist. Programmstrukturen, die bei einem zentralistischen System durchaus angemessen sind, können bei verteilten Programmen ineffizient sein (siehe obiges Beispiel). Umgekehrt kann eine effiziente Lösung für ein verteiltes System für ein zentralistisches System sehr aufwendig sein.

Semaphore

Ähnlich wie die geschilderte Implementationsmöglichkeit für Monitore auf verteilten Systemen lassen sich auch Semaphore realisieren. Statt einer Monitorhülle gibt es Semaphorhüllen und ebenfalls Stellvertreterprozesse. Bei der Implementierung von Semaphoren nach der oben geschilderten Technik gelten für die einzelnen Vor- und Nachteile die gleichen Überlegungen wie bei den Monitoren.

Eine andere Möglichkeit, Semaphore in verteilten Systemen zu implementieren, wurde in /AMMA81 / vorgeschlagen. Bei dieser Technik wird auf jeder Systemkomponente, auf der sich ein Prozeß befindet, der ein systemweit bekanntes Semaphor benutzt, eine Repräsentation dieses Semaphors angelegt. Wird der Wert eines Semaphors verändert, wird bei allen Repräsentationen dieses Semaphors ebenfalls die Wertänderung durchgeführt. In /AMMA81/ wird allerdings nicht darauf eingegangen, was passiert, wenn auf zwei verschiedenen Repräsentationen desselben Semaphors gleichzeitig Operationen ausgeführt werden. Es soll versucht werden, dieses Problem an einem Beispiel zu verdeutlichen.

Ein netzweit bekanntes Semaphor soll den Wert eins haben. Zwei Prozesse auf verschiedenen Prozessoren führen nun gleichzeitig eine 'Request-Operation' aus (d.h. der Semaphor soll jeweils um eins erniedrigt werden). Da die Semaphore den Wert eins hat, müßte ein Prozeß blockiert werden. In /AMMA81/ heißt es, daß eine Semaphoroperation sofort auf der entsprechenden Repräsentation ausgeführt wird und der neue Semaphorwert den Repräsentationen auf den anderen Prozessoren mitgeteilt wird, d.h. im obigen Fall wären beide Request-Operationen ausführbar. Dies widerspricht der üblichen Semantik von Semaphoren. Um die Semantik von Semaphoren nicht zu verändern, müßte zwischen den Prozessoren, die eine Repräsentation eines Semaphors enthalten, zuerst ein Dialog stattfinden, der einen konsistenten Zustand des Semaphors gewährleistet. Dieser Dialog dürfte sehr aufwendig sein und zu einer großen Belastung des Übertragungsmechanismus führen. Bei dieser Implementation von 'verteilten Semaphoren' wird keine Aussage darüber gemacht, was passiert, wenn Teile des Übertragungsmechanismus

ausfallen und die Repräsentationen von einigen Semaphoren nicht mehr aktualisiert werden können.

4.3.1.1.2 Probleme bei der Prozeßautomatisierung:

Werden zur Spezifikation und Implementation von Automatisierungsprogrammen nur gemeinsame Objekte mit entsprechenden ressourcenorientierten Synchronisationskonzepten verwendet, so treten vor allem zusätzliche Probleme an den Schnittstellen Rechenprozesse/technischer Prozeß und Automatisierungsprogramm/Benutzer auf (siehe Kapitel 2.4). Zum Beschreiben der Zusammenarbeit technischer Prozeß/Rechenprozesse und Automatisierungsprogramm/Benutzer verwenden Programmiersprachen, die zur Programmierung von Prozeßautomatisierungsproblemen entwikkelt wurden, Ein-/Ausgabefunktion und Interrupts. Die Programmiersprache PEARL, die speziell zur Prozeßautomatisierung entworfen wurde, benutzt zur Kommunikation und Synchronisation mit dem technischen Prozeß (bzw. Benutzer) Ein-/Ausgabefunktionen und Sprachkonstrukte zur Behandlung von Interrupts, wogegen zur Synchronisation und Kommunikation der Rechenprozesse untereinander gemeinsame Objekte verwendet werden (Globale Variable, Semaphore).
Die Zusammenarbeit des technischen Prozesses mit den Rechenprozessen läßt sich durch gemeinsame Objekte weniger günstig beschreiben. Dies soll an einem Beispiel erläutert werden. Wenn ein technischer Prozeß in zufälligen Zeitabständen entsprechenden Rechenprozessen bestimmte Ereignisse mitteilen will, werden dazu im allgemeinen Interrupts benutzt. Durch das Eintreffen eines erwarteten Interrupts wird ein Prozeß aktiviert, der die notwendigen Reaktionen für das angezeigte Ereignis einleitet. Es wirkt umständlich, wenn ein Interrupt als gemeinsames Objekt betrachtet wird (z.B. als Semaphor, das vom technischen Prozeß erhöht und vom entsprechenden Rechenprozeß erniedrigt wird). Ebenso gibt es Interpretationsprobleme, wenn ein Rechenprozeß z.B. einen neuen Sollwert für eine Prozeßgröße an den technischen Prozeß geben will. Es ergibt sich die Frage, was das gemeinsame Objekt von Rechenprozessen

und technischem Prozeß ist, über das diese Nachricht übergeben werden soll.
Ein ähnliches Interpretationsproblem tritt auf, wenn der technische Prozeß einem Rechenprozeß einen Meßwert mitteilen will. Die gleichen Überlegungen können angestellt werden für die Zusammenarbeit des Automatisierungsprogramms mit dem Benutzer bzw. umgekehrt.

4.3.1 .2 Verteilte Programme und Botschaftskonzepte

4.3.1 .2.1 Allgemeine Probleme

Bei der Implementierung von Botschaftssystemen für verteilte Systeme werden keine Stellvertreterprozesse und Monitorhüllen benötigt. Beim Senden von Botschaften muß bekannt sein, auf welcher Komponente des verteilten Systems sich der Zielprozeß befindet. Diese Information kann dem System durch entsprechende Anweisungen zur Verfügung gestellt werden (siehe Abschnitt 4.3.1.1). Es gibt allerdings einige Schwierigkeiten, insbesondere bei der Implementierung von rein asynchronen Botschaftskonzepten.
Bei asynchronen Botschaftskonzepten wird der sendende Prozess nicht blockiert. Die Nachricht muß also zwischengespeichert werden, bis sie vom Empfänger abgeholt wird. Dazu muß auf der Systemkomponente, auf der der Sendeprozeß läuft, oder auf der Komponente, auf der sich der Empfangsprozeß befindet, der entsprechende Speicherplatz zur Verfügung stehen. Der Speicherplatzbedarf, der insgesamt bzw. auf den einzelnen Komponenten für das Zwischenspeichern benötigt wird, kann nicht statisch (z.B. zur Übersetzungszeit) festgestellt werden. Der benötigte Zwischenspeicherplatz hängt vom zeitlichen Verhalten der einzelnen Prozesse ab. Werden eine bestimmte Zeit lang sehr schnell hintereinander Nachrichten gesendet, die relativ langsam von den Empfängern entgegengenommen werden, so wird ein entsprechend großer Speicherplatz benötigt.
Bei asynchronen Botschaftsmechanismen wird also eine dynamische Speicherplatzverwaltung benötigt. Wird eine Nachricht

ten im allgemeinen nicht übereinstimmt mit der Reihenfolge, in der die Nachrichten angenommen werden, wird der Zwischenspeicherplatz in eine Folge von belegten und freien Speicherplätzen zerstückelt. Dieses bei dynamischen Speicherverwaltungen übliche Problem, kann mit entsprechenden Strategien aus der Betriebssystemtechnik gelöst werden /HOFM78/. Allerdings hängt die Effizienz all dieser Lösungen stark von den Hardwaremöglichkeiten des betrachteten Rechners ab. In jedem Fall belastet eine dynamische Speicherzuteilung die Laufzeiten der Programme.

Diese zeitliche Verzögerung stellt allerdings nicht das eigentliche Problem dar. Vielmehr kann es passieren, daß sich durch die bei asynchronen Konzepten nötige dynamische Speicherverwaltung Benutzerprogramme verklemmen, ohne daß aus dem Programmtext eine mögliche Verklemmung ersichtlich ist. Dies soll an einem Beispiel verdeutlich werden (siehe Bild 4,7).

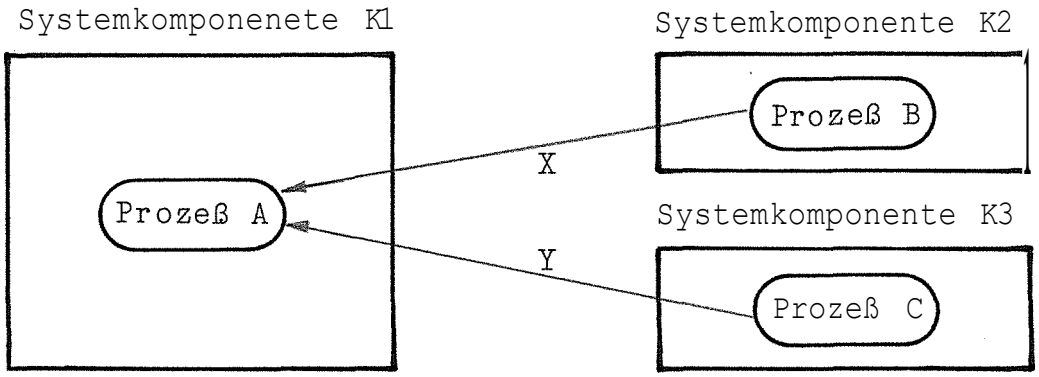


Bild 4,7: Verklemmung bei asynchronen Botschaftsmechanismen

Ein Prozeß A, der auf der Komponente K1 läuft, erhält vom Prozeß B die Nachricht X und vom Prozeß C die Nachricht Y. Der Prozeß B läuft auf der Komponente K2 und der Prozeß C auf der Komponente K3. Die gesendeten Nachrichten sollen auf der Komponente zwischengespeichert werden, auf der sich der Adressat befindet. Die Komponente K1 soll Platz zum Zwischenspeichern von zwei Nachrichten haben. Angenommen, Prozeß A möchte vom Prozeß B die Nachricht X. Prozeß B hat diese Nachricht noch nicht gesendet, deshalb wird Prozeß A blockiert, um auf das Eintreffen der Nachricht X zu warten. Bis Prozeß B die Nach-

von zwei Nachrichten haben. Angenommen, Prozeß A möchte vom Prozeß B die Nachricht X. Prozeß B hat diese Nachricht noch nicht gesendet, deshalb wird Prozeß A blockiert, um auf das Eintreffen der Nachricht X zu warten. Bis Prozeß B die Nachricht X sendet, hat Prozeß C zweimal die Nachricht Y gesendet, d.h. der gesamte Zwischenspeicherplatz auf K1 ist belegt. Möchte der Prozeß B die Nachricht X nun senden, muß er blokkiert werden, und damit hat sich das Gesamtsystem verklemmt.
Ähnliche Fälle lassen sich auch konstruieren, wenn die Zwischenspeicherung auf der Komponente erfolgt, wo sich der Sendeprozeß befindet.
Diese Verklemmung kann zwar durch einen entsprechend dimensionierten Speicherplatz hinausgeschoben, aber nicht ausgeschlossen werden. Asynchrone Botschaftskonzepte lassen sich also wegen der endlichen Speicherplatzgröße nicht ohne weiteres verklemmungssicher realisieren.
Einfacher zu Implementieren sind synchrone Botschaftsmechanismen. Bei den synchronen Konzepten wird der Sendeprozeß solange blockiert, bis ihm die Nachricht vom Empfangsprozeß abgenommen wird. Dadurch kann der benötigte Speicherplatz statisch festgestellt werden. Synchrone Botschaftsmechanismen haben allerdings gegenüber den asynchronen Konzepten den Nachteil, daß der Parallelitätsgrad eingeschränkt wird.

4.3.1.2.2 Probleme bei Automatisierungsprogrammen

Durch Botschaftskonzepte läßt sich die Zusammenarbeit von technischem Prozess bzw. Benutzer und Automatisierungsprogrammen einfacher darstellen, als dies mit gemeinsamen Objekten möglich ist (siehe Abschnitt 5.1.2).
Die Signale des technischen Prozesses können als Botschaften an einen entsprechenden Rechenprozeß betrachtet werden. Genauso kann ein Eingabeaufruf eines Rechenprozesses als die Bereitschaft interpretiert werden, eine bestimmte Botschaft annehmen zu wollen.
Das Ausgeben von neuen Sollwerten für Prozeßgrößen kann als das Senden einer entsprechenden Botschaft interpretiert wer-

den.
Bei Prozeßautomatisierungssystemen werden im allgemeinen Kenndaten des technischen Prozesses unterschiedliche Zeit lang zwischengespeichert (Protokolle). Auf die entsprechende Datei greifen u.U. mehrere Prozesse zu. Werden zur Kommunikation der Rechenprozesse untereinander nur Botschaftskonzepte verwendet, so muß diese Datei von einem Prozeß verwaltet werden (siehe Abschnitt 3.2). Dieser Verwaltungprozeß erhält die Zugriffswünsche der anderen Prozesse durch Botschaften mitgeteilt und führt die entsprechenden Operationen aus. Das Ergebnis wird an den auftraggebenden Prozeß durch Botschaften weitergereicht. Dieser Verwaltungsprozeß stellt einen Engpaß dar, da alle Zugriffe auf eine solche Datei sequentialisiert werden.
Wird für eine Prozeßautomatisierung eine Monoprozessoranlage verwendet, so sind Botschaftskonzepte für die Synchronisation und Kommunikation von Rechenprozessen weniger effizient als gemeinsame Objekte. Hier entsteht nun eine Inhomogenität zwischen dem Beschreibungskonzept für die Synchronisation und Kommunikation der Rechenprozesse untereinander und dem Konzept mit dem die Zusammenarbeit der Rechenprozesse mit dem technischen Prozeß beschrieben wird.

4.3.2. Zusammenhang zwischen der Spezifikation und Implementation verteilter Programme

Wie in Kapitel 3 dargestellt, gibt es Spezifikations- und Implementationssprachen, die die Zusammenarbeit von parallelen Prozessen entweder mit gemeinsamen Objekten oder mit Botschaftskonzepten beschreiben.
Vorweg kann gesagt werden, daß die Spezifikations- und Implemetationssprache vor allem bzgl. der verwendeten Synchronisations- und Kommunikationskonzepte aufeinander abgestimmt sein sollen.
Steht z.B. eine Programmiersprache zur Verfügung, in der es keine gemeinsamen Objekte, sondern nur ein Botschaftskonzept gibt, soll dann für die Spezifikation ein Beschreibungskonzept benutzt werden, das auf gemeinsamen Objekten beruht bzw. umge-

kehrt?
Wird z.B. zur Spezifikation eines Programms die Technik /KERA82/ benutzt, (siehe Abschnitt 3.1.1.2) und zur Implementation eine Sprache verwendet, die zur Synchronisation und Kommunikation nur Botschaften benutzt, so kann dies zu Problemen bei der Implementation führen. Die Moduln in der Spezifikation nach /KERA82/ enthalten einen sogenannten 'Syn'-Teil. Darin wird beschrieben, wie auf die einzelnen Funktionen des Moduls zugegriffen werden darf, d.h. welche Funktionen gleich-, zeitig ausgeführt werden dürfen und welche sieh gegenseitig ausschließen.
Wird ein Modul aus der Spezifikation auf einen einzelnen Prozeß der Programmiersprache abgebildet (in Sprachen mit Botschaftsmechanismen gibt es nur aktive Strukturierungselemente, was Prozessen entspricht), wobei die einzelnen Funktionen den erwarteten Botschaften entsprechen, so können nicht mehrere Funktionen gleichzeitig ausgeführt werden. Ein Prozeß führt die einzelnen Aufträge (Funktionsaufrufe) hintereinander aus. Kann ein Auftrag momentan nicht ausgeführt werden, wird er zurückgestellt, bis es durch andere eintreffende und bearbeitbare Aufträge möglich ist, daß er abgeschlossen wird.
Der in der Spezifikation angegebene Parallelitätsgrad kann also nicht erreicht werden, wenn ein Modul auf einen Prozeß abgebildet wird. Der spezifizierte Parallelitätsgrad kann nur erreicht werden, wenn der Modul durch mehrere Prozesse realisiert wird. In diesem Fall müssen auch die Daten eines Moduls aufgeteilt werden. Die einzelnen Moduln sind aber so entworfen, daß die in ihnen zusammengefaßten Daten logisch zusammengehören, d.h. werden die Daten auf mehrere Prozesse verteilt, so müssen diese im allgemeinen wieder Botschaften austauschen, um sich über den Zustand der Datenteile in anderen Prozessen zu informieren. Es ist offensichtlich, daß dies zu unübersichtlichen Implementationen führt.
Wird für die Spezifikation eine Sprache benutzt, die nur Botschaften kennt, und für die Implementation eine Sprache verwendet, die nur gemeinsame Objekte kennt, so sind die Schwierigkeiten geringer, falls kein verteiltes Programm entstehen soll (Begründung siehe vorige Abschnitte). Der Austausch von

Nachrichten und das entsprechende Blockiertverhalten kann z.B. durch globale Variable, Semaphore und Hilfsprozesse simuliert werden. Es kann aber notwendig werden, Hilfsprozesse einzuführen, wenn z.B. auf mehrere Ereignisse gleichzeitig gewartet werden soll. Die Struktur dieser Simulation ist zwar einfach, allerdings kann der Bedarf an Semaphoren und Hilfsprozessen beträchtlich sein, so daß die Implementationen sehr langsam und schwerfällig werden /BFH082/.

Weiterer Gesichtspunkte bei der Auswahl eines Spezifikationskonzepts sind das Anwendungsgebiet und das verwendete Rechnersystem.

Wie bereits im vorherigen Kapitel dargestellt, läßt sich mit Botschaftskonzepten die Zusammenarbeit zwischen steuerndem Programm und technischem Prozeß besser beschreiben. Speziell in der Prozeßautomation kann es Umstände geben, die es geeigneter erscheinen lassen, eine Spezifikationsmethode mit Botschaftskonzepten zu wählen, obwohl die Implementation in einer Programmiersprache erfolgen soll, die zur Synchronisation und Kommunikation gemeinsame Objekte benutzt. Dies ist vor allem der Fall, wenn zwischen dem technischen Prozeß und dem Rechenprozeß eine häufige und vielfältige Kommunikation nötig ist. Die Einfachheit und Klarheit der Spezifikation kann die Nachteile der Implementation wieder ausgleichen.

5 Ein Konzept zur Spezifikation und Implementation von Automatisierungsprogrammen für verteilte Systeme

In den vorherigen Kapiteln wurden die Probleme geschildert, die bei der Spezifikation und Implementation von Programmen zur Prozeßautomation auftreten, insbesondere, wenn es sich um verteilte Programme handelt. In Kapitel 4 wurde versucht, den Zusammenhang zwischen den in der Spezifikation und den in der Implementation verwendeten Synchronisationskonzepten aufzuzeigen.
Im folgenden soll ein Konzept zur Spezifikation von Programmen zur Steuerung technischer Prozesse erläutert werden. Dieses Konzept benutzt bei der Beschreibung der Synchronisation und Kommunikation einen Botschaftsmechanismus (die Begriffe Botschafts- und Nachrichtenmechanismus werden im folgenden synonym verwendet), läßt aber auch die Verwendung von gemeinsamen Objekten zu. Prozesse auf verschiedenen Systemkomponenten (Prozessoren) können nur über Botschaften kommunizieren bzw. sich synchronisieren, wogegen Prozesse, die sich auf derselben Systemkomponente befinden, auch gemeinsame Objekte benutzen können. Der vorgestellte Botschaftsmechanismus versucht, die in Kapitel 3 geschilderten Nachteile der einzelnen Botschaftskonzepte zu vermeiden und die Vorteile zu erhalten.
Um den Übergang von der Spezifikation zur Implementation zu erleichtern, wurde eine Programmiersprache um entsprechende Sprachkonstrukte erweitert.

5-1 Die Struktur von Automatisierungssystemen

Ein Automatisierungssystem besteht aus drei Komponenten /HARR82/:
- dem Benutzer
- dem Automatisierungsprogramm
- dem technischen Prozeß.

Den Zusammenhang zwischen diesen Komponenten zeigt Bild 5.1

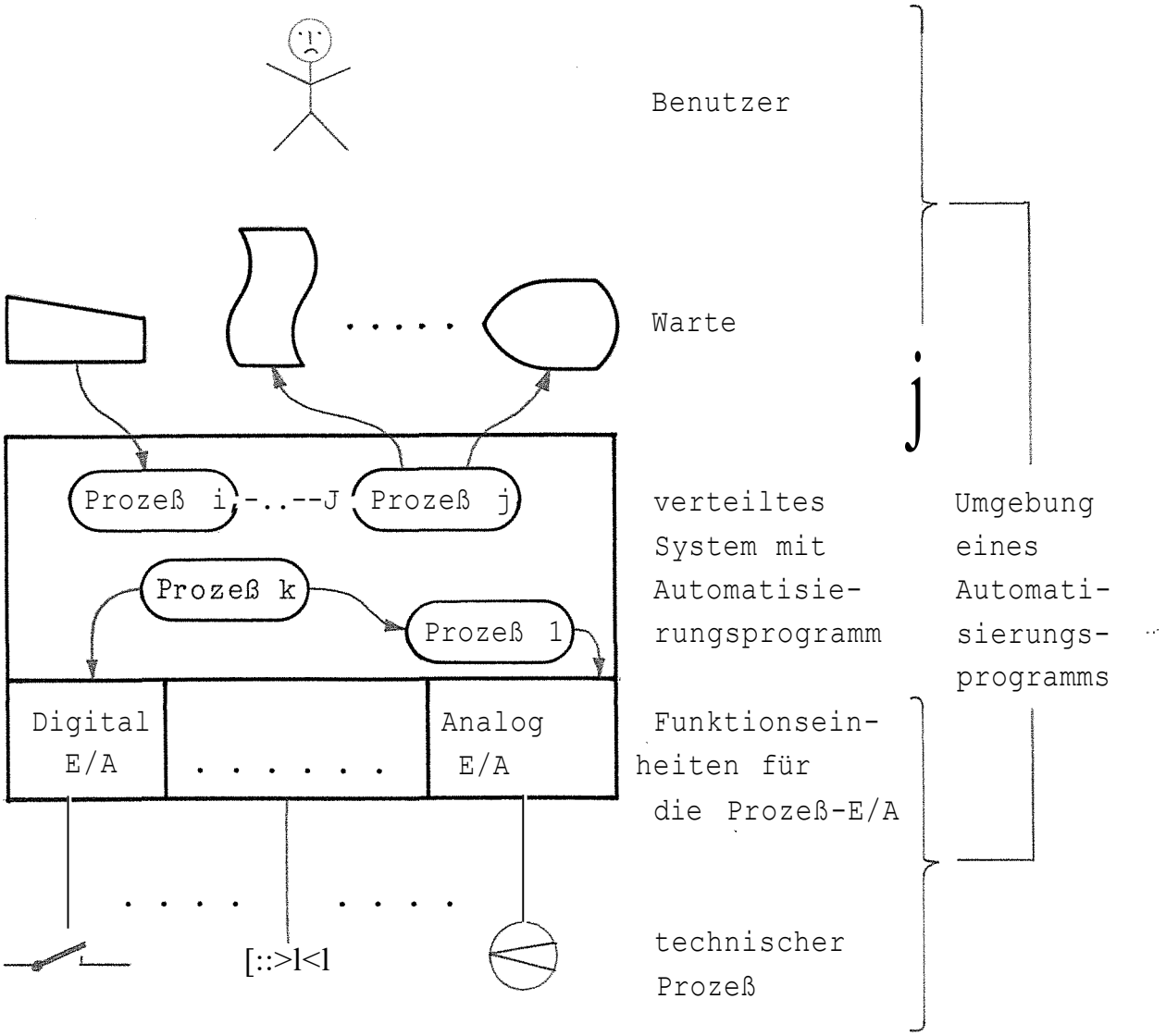


Bild 5.1 Struktur von Automatisierungssystemen

Die wesentlichen Aufgaben eines Automatisierungsprogramms sind:

- Annehmen von Meßwerten, die der technische Prozeß liefert.
- Ausgeben von Soll- und Stellwerten an den technischen Prozeß.
- Annehmen von Sollwerten vom Benutzer (Mensch).
- Ausgeben von wesentlichen Zuständen des technischen Prozesses an den Benutzer (der Zustand eines technischen Prozesses soll durch bestimmte Meßwerte, die in einem bestimmten Zeitraum an das Automatisierungsprogramm gegeben wurden, repräsentiert sein).

- Protokollieren von wesentlichen Zuständen des technischen Prozesses auf Dateien, sowie Verwalten dieser Dateien (z.B. Führen von Tagesprotokollen, Monatsprotokollen).

Aus diesen Aufgaben für Automatisierungsprogramme folgt, daß für deren Strukturierung sowohl gemeinsame Objekte als auch Prozesse notwendig sind. Gemeinsame Objekte in Automatisierungsprogrammen sind z.B. die bereits erwähnten Protokolldateien.
Wesentlich für Automatisierungsprogramme ist jedoch die Strukturierung in Prozesse (auch Rechenprozesse genannt) und die Kommunikation dieser Rechenprozesse untereinander und mit der 'Umgebung' (/LAUB76/ ,/LUDE80/). Ein Automatisierungsprogramm tauscht mit dem technischen Prozeß und dem Benutzer Daten aus (Kommunikation mit der 'Umgebung').
Mit dem Benutzer wird im allgemeinen über sogenannte 'Standard-Ein-/Ausgabegeräte' (z.B. graphischer Bildschirm, Drukker) kommuniziert.
Die Kommunikation mit dem technischen Prozeß erfolgt über Prozeß-Ein/Ausgabegeräte (Digital-Ein-/Ausgabe, Analog-Ein-/Ausgabe).
Rechenprozesse müssen untereinander ebenfalls kommunizieren bzw. sich synchronisieren. Die verschiedenen Möglichkeiten, die sich für Rechenprozesse dazu anbieten, wurden in Kapitel 3 diskutiert.
Im folgenden wird versucht, die Kommunikation
- Rechenprozeß - Rechenprozeß,
- Rechenprozeß - Benutzer und
- Rechenprozeß - technischer Prozeß

auf ein einheitliches Botschaftskonzept (Nachrichtenkonzept) zurückzuführen.
Die Nachrichten, die in einem Automatisierungsprogramm zwischen den einzelnen Kommunikationspartnern ausgetauscht werden, können auf folgenden Abstraktionsebenen betrachtet werden:
- Die logische Bedeutung einer Nachricht
- Die Realisierung einer Nachricht
- Die Implementierung einer Nachricht

Nachrichten (Botschaften) sollen aus einem Nachrichtennamen und den Nachrichtenparametern bestehen. Durch den Nachrichtennamen erhält eine Nachricht eine logische Bedeutung. Der Sender gibt damit an, was er 'sagen' will, und der Empfänger 'weiß', was mit dieser Nachricht gemeint ist. Die Nachrichtenparameter geben detailliertere Auskunft über die Nachricht. Eine exaktere Definition der Bedeutung des Nachrichtennamens und der Nachrichtenparameter folgt in späteren Kapiteln.
Hier soll nur versucht werden, diese Begriffe durch ein Beispiel zu erklären.
Es soll folgende Botschaft betrachtet werden:

```
        VENTILSTELLUNG(wert)
                       L__ Nachrichtenparameter
             1----Nachrichtenname
```

Die Nachricht mit dem Namen 'VENTILSTELLUNG' hat für den Sender bzw. den Empfänger die Bedeutung, daß er die Ventilstellung verändern soll. Der Nachrichtenparameter gibt an, wie die Ventilstellung verändert werden soll.
Durch den Nachrichtennamen erhält eine Nachricht also eine Bedeutung.
Neben der Bedeutung einer Botschaft ist auch noch die Realisierung einer Botschaft wesentlich. Als Realisierung wird dabei verstanden, wie sich eine Botschaft dem Kommunikationspartner darstellt. Eine Botschaft kann z.B. realisiert sein durch:

- Änderungen von elektrischen Spannungspegeln auf Signalleitungen.
- Ausgeben von alphanumerischen Zeichen auf entsprechende E/A-Geräte
- Ausgeben oder Ergänzen von Bildern auf graphischen Ausgabegeräten
- Eingeben von alphanumerischen Zeichen über entsprechende Eingabegeräte.
- Daten die zwischen Rechenprozessen ausgetauscht werden.

Zur Beschreibung einer Botschaftsrealisierung gehören aller-

dings keine Angaben über den Algorithmus, der z.B. ein bestimmtes Bild erzeugt oder eine Spannungsänderung auf einer bestimmten Leitung verursacht, bzw. welches Protokoll beim Datenaustausch zwischen Rechenprozessen verwendet wird. Dies ist die Implementierung der Nachricht.
Der technische Prozeß kann in verschiedene Komponenten zerlegt werden. Jede Komponente des technischen Prozesses soll eindeutig mit einem Namen gekennzeichnet sein, ebenso sollen alle Rechenprozesse und E/A-Geräte, die zur Kommunikation mit dem Benutzer dienen, mit einem eindeutigen Namen versehen sein.
Beim Senden einer Nachricht muß angegeben werden, wer diese Nachr ieht erhalten soll, bzw. beim Empfangen muß festgelegt sein, von wem eine Nachricht gewünscht wird (Adressat, Absender).
In den folgenden Abschnitten soll gezeigt werden, daß mit diesem Nachrichtenkonzept die Kommunikation unter verschiedenartigen Partnern problemgerecht dargestellt werden kann, und daß die unterschiedlichen Realisierungen und Implementierungen auch bei Automatisierungsprogrammen nicht in die Spezifikation eingehen müssen.

5.1.1 Realisierung der Kommunikation technischer Prozeß/ Rechenprozeß

Ein technischer Prozeß kann, wie schon erwähnt, in Komponenten zerlegt werden. Eine solche Komponente sendet bzw. empfängt Nachrichten von anderen Komponenten und Rechenprozessen (siehe Bild 5.2).

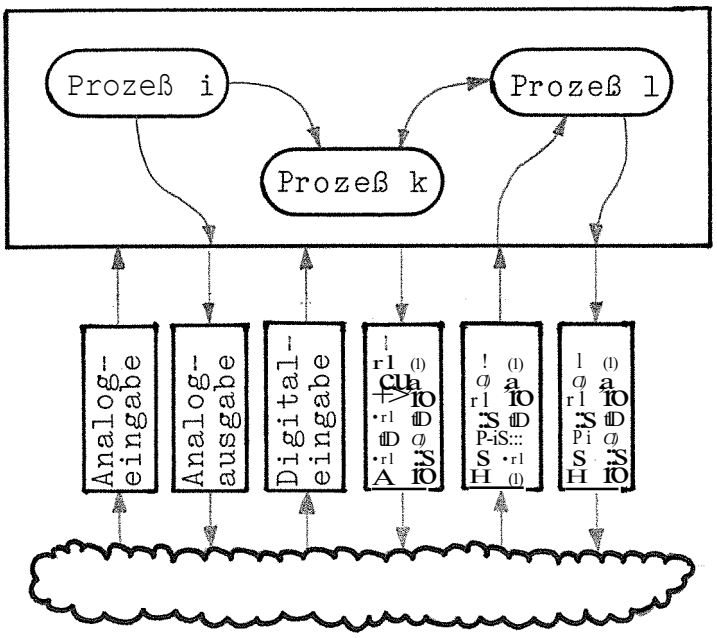


Bild 5.2 : Kommunikation Rechenprozesse/technischer Prozeß

Nachrichten zwischen Rechenprozessen und dem technischen Prozeß werden durch Prozeßsignale dargestellt. Bei diesen Signalen unterscheidet man zwischen Analog-, Digital- und Impulssignalen und die jeweilige Signalrichtung (Ein-, Ausgabe). Nach DIN 66216 werden folgende Funktionseinheiten unterschieden /LAUB76/:

- Digitaleingabe – Digitalausgaben
- Analogeingabe – Analogausgaben
- Impulseingabe – Impulsausgaben

Bei diesen Funktionseinheiten werden noch einzelne Typen unterschieden:

- Funktionseinheiten, die ein Signal erzeugen, wenn eine Ausgabe beendet ist bzw. wenn eine neue Eingabe vorliegt: Solche Funktionseinheiten werden im folgenden als aktive Funktionseinheiten bezeichnet.
- Funktionseinheiten, die kein Signal erzeugen, wenn eine Eingabe vorliegt bzw. eine Ausgabe beendet ist. Bei diesen Funktionseinheiten wird in bestimmten Zeitabständen abgefragt, ob eine neue Eingabe vorliegt bzw. eine Ausgabe beendet ist (Polling):

Im folgenden werden diese Funktionseinheiten passive Funktionseinheiten genannt.

An der Schnittstelle technischer Prozeß - Automatisierungsprogramm (Rechenprozesse) gilt es nun folgende Punkte zu klären:
- Interpretation der Nachrichtennamen, Absenderangaben, Adressatenangaben und Nachrichtenparameter
- Realisierung bzw. Implementierung des Nachrichtenaustauschs (Kommunikation)

Eine Komponente des technischen Prozesses sendet und empfängt Nachrichten über Prozeßsignalleitungen. Mehrere Prozeßsignalleitungen können zu Leitungsgruppen zusammengefaßt werden. Bild 5.3 zeigt ein Beispiel, wie bei einer Leitungsgruppe Nachrichtenname, Adressat, Absender und Nachrichtenparameter interpretiert werden können.

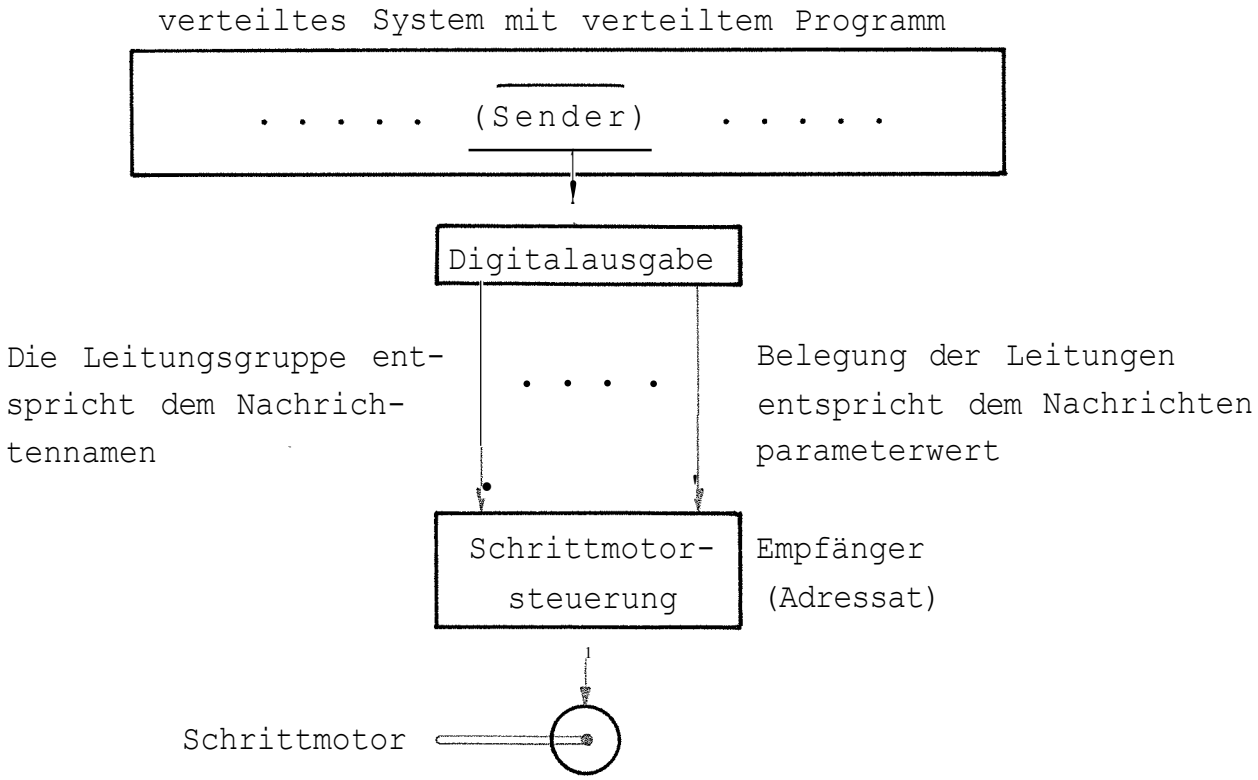


Bild 5.3 : Eine Leitungsgruppe entspricht einer Nachricht mit einem bestimmten Namen an einen bestimmten Empfänger

Der Digitalausgabe und den Leitungen ist der Botschaftsname zugeordnet. Sender dieser Nachricht sind die Rechenprozesse, die diese Digitalausgabe benutzen. Der Empfänger dieser Botschaft ist durch die Verdrahtung festgelegt. Die jeweilige Belegung der Leitungen (Bitmuster, das über das Digitalregister ausgegeben wird) entspricht dem Wert des Nachrichtenparameters.
Bei Analogein- bzw. -ausgaben entspricht die Höhe des Analogsignals dem Nachrichtenparameterwert.
Ein- und dieselbe Leitungsgruppe kann auch zum Senden und Empfangen von Nachrichten an bzw. von verschiedenen Komponenten des technischen Prozesses verwendet werden. Dazu können unterschiedliche Techniken verwendet werden.

- Zunächst wird die Leitungsgruppe eine bestimmte Zeit lang mit der Kennung des Adressaten belegt und dann erst mit der Botschaftskennung.
- Die Leitungsgruppe wird in zwei disjunkte Leitungsmengen zerlegt. Die eine Teilmenge wird mit der Kennung des Adressaten (Adreßbus) und die andere mit der Kennung der Botschaft (Datenbus) belegt.

Eine weitere Interpretationsmöglichkeit für die Prozeßsignalleitungen ist, daß jeweils eine bestimmte Belegung der Signalleitungen einem bestimmten Nachrichtennamen entspricht. Durch die Leitungsgruppe wird also der Adressat bzw. der Absender einer Nachricht festgelegt, wobei die Leitungsbelegung einem Botschaftsnamen entspricht. Bei dieser Interpretation von Signalleitungen können zu einer Nachricht keine Nachrichtenparameter mit angegeben werden.
Es können auch Mischformen dieser Interpretationsmöglichkeiten vorkommen (siehe Bild 5.4). Die Belegung eines Teils der Leitungsgruppe wird als Botschaftsname, die Belegung eines anderen Teils der Leitungen als Adressat und die Belegung der restlichen Leitungen als Botschaftsparameter aufgefaßt.

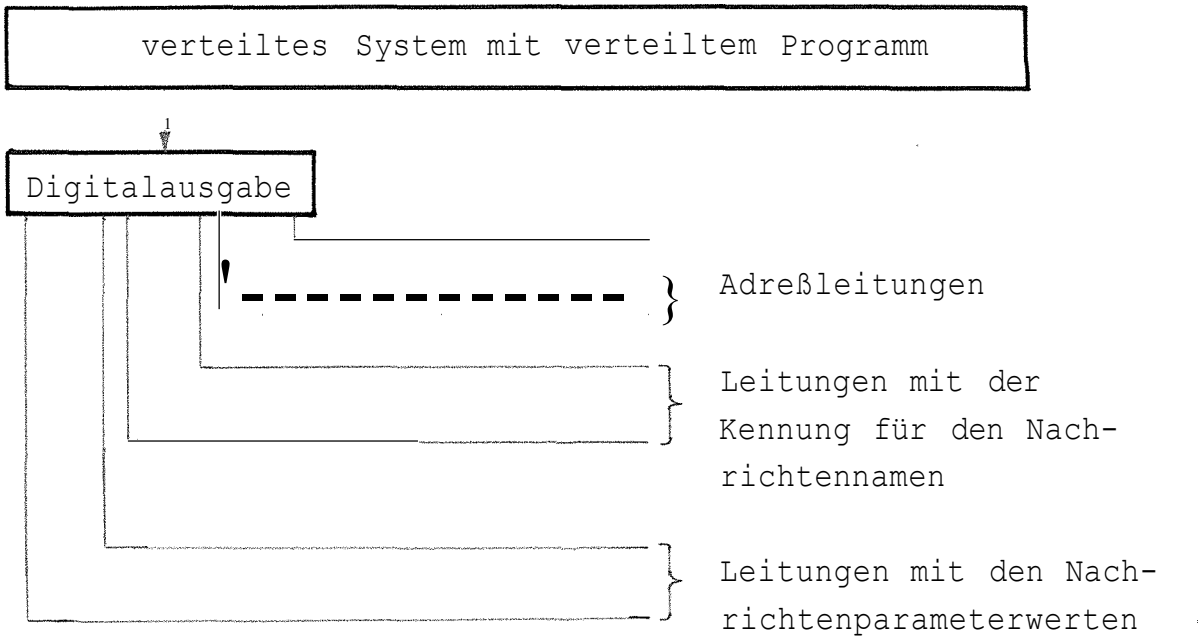


Bild 5.4 : Adreßleitungen, Nachrichtenleitungen, Parameterleitungen

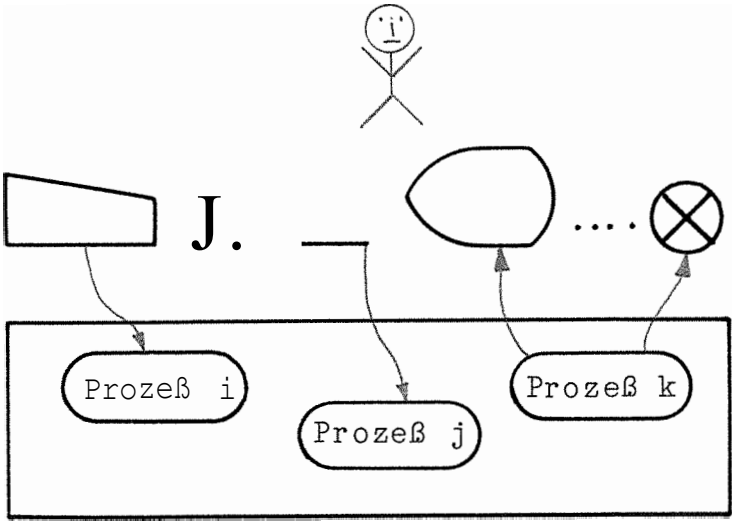
Impulseingaben bzw. -ausgaben sind Nachrichten ohne Parameter. Den Impulsleitungen wird ebenfalls der entsprechende Nachrichtenname und der entsprechende Adressat zugeordnet. Ein Impuls auf einer Leitung bedeutet: Eine Nachricht entsprechenden Namens ist eingetroffen.

Zur Kommunikation zwischen Rechenprozessen und dem technischen Prozeß werden aktive oder passive Funktionseinheiten benutzt. Die Unterscheidung zwischen aktiven und passiven Einheiten ist ein Implementationsproblem.
Bei aktiven Funktionseinheiten wird bei der Eingabe (Nachrichtenempfang) zuerst auf den Interrupt gewartet, der anzeigt, daß die dazugehörige Nachricht eingetroffen ist. Erst dann wird der Parameterwert aus dem entsprechenden Analog-Digitalwandler oder einem Digitalregister geholt. Ähnlich ist es bei der Ausgabe. Bei aktiven Funktionseinheiten wird zunächst der Parameterwert über die entsprechende Funktionseinheit ausgegeben und dann auf den Interrupt gewartet, der anzeigt, daß die Nachricht 'übertragen' ist.
Bei passiven Funktionseinheiten gibt es zwei Möglichkeiten für die Implementation des Sendens bzw. Empfangens einer Nach-

richt. Bei der ersten Möglichkeit wird eine Nachricht einfach dadurch gesendet, daß der Parameterwert über die entsprechende Funktionseinheit ausgegeben wird, ohne zu prüfen, ob die Nachricht auch angekommen ist.
Beim Empfangen wird der Wert aus der entsprechenden Funktionseinheit ausgelesen. Bei dieser Art des Eingebens und Ausgebens von Nachrichten kann es also passieren, daß die Funktionseinheit beim Senden bzw. Schreiben in einem nicht definierten Zustand ist und dadurch die Werte der Nachrichten verfälscht.
Bei der zweiten Möglichkeit wird zuerst abgefragt, ob die Funktionseinheit sich in einem definierten Zustand befindet. Erst dann wird der entsprechende Wert aus der Funktionseinheit gelesen bzw. hineingeschrieben.
Die Abfrage nach dem Zustand der Funktionseinheit erfolgt zyklisch, d.h. sie wird solange wiederholt, bis der Zustand der entsprechenden Einheit definiert ist.

5.1 .3. Realisierung der Kommunikation Benutzer/Rechenprozeß

Der Benutzer kommuniziert mit dem Automatisierungsprogramm, um Informationen über den Zustand des technischen Prozesses zu erhalten oder dem Automatisierungsprogramm neue Parameter für die Prozeßsteuerung mitzuteilen (siehe Bild 5.5).



Benutzer

Geräte zur Kommunikation mit dem Benutzer

verteiltes System mit verteiltem Automatisierungsprogramm

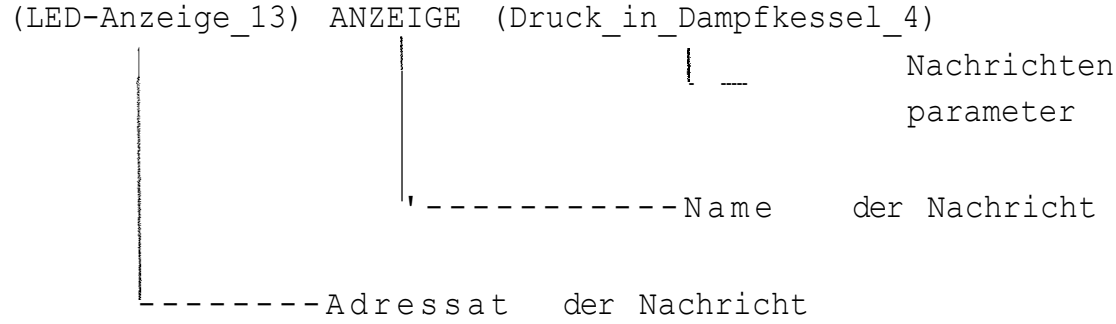
Bild 5,5 : Kommunikation Rechenprozesse/Benutzer

Für die Kommunikation Benutzer-Automatisierungsprogramm gibt es folgende Möglichkeiten:

1. Nachrichten des Automatisierungsprogramms an den Benutzer
   - optische Signale (z.B. verschiedenfarbige Kontrollampen)
   - akkustische Signale (z.B. Hupen)
   - Anzeigen (Digital-oder Analoganzeigen)
   - alphanumerische Ausgaben (z.B. Drucker)
   - graphische oder semigraphische Ausgaben (z.B. graphisches Sichtgerät, Plotter)
2. Nachrichten des Benutzers an das Automatisierungsprogramm
   - Schalterstellungen
   - alphanumerischer Text
   - Ergänzen oder Ändern von graphischen Darstellungen (interaktive Graphik).

Die Ausgabe von optischen und akkustischen Signalen bzw. die Eingaben über Schalter lassen sich genauso interpretieren wie die Ein-/Ausgaben über Signalleitungen (siehe Abschnitt 5.1.2).
Bei Ausgaben des Automatisierungsprogramms über Digital- oder Analoganzeigen wird das entsprechende Anzeigengerät durch den Nachrichtennamen und den Adressaten eindeutig bestimmt. Der Adressat entspricht dem Anzeigengerät. Der Nachrichtenname kann mehr oder weniger willkürlich gewählt werden. Sinnvoll erscheint z.B. der Nachrichtenname 'ANZEIGE'. Der anzuzeigende Wert entspricht dem Naehrichtenparameter, einschließlich der physikalischen Einheit und dem Ort im technischen Prozeß an dem der anzuzeigende Wert gemessen wurde.

Beispiel für eine Anzeigenausgabe:



Bei alphanumerischen Ausgaben kann der erklärende Text als Nachrichtenname und das Ausgabegerät als Nachrichtenadressat interpretiert werden. Die Variablen, deren Werte nach bzw. innerhalb des erklärenden Texts ausgegeben werden, entsprechen den Nachrichtenparametern.
Die graphischen Ausgaben von Automatisierungsprogrammen sind meist Kurven, die den Verlauf von Meßgrößen über der Zeit oder irgendwelchen anderen Prozeßgrößen darstellen.
Bei der Ausgabe von Kurven entspricht das Ausgabegerät dem Adressaten einer Nachricht. Das, was die Kurve 'darstellt' (z.B. Füllstand von Rührkessel 3 im Monat Mai 1982) kann als Nachrichtenname interpretiert werden. Der Verlauf der Kurve entspricht den Nachrichtenparametern.
Die semigraphischen Ausgaben sind meist schematische Darstellungen des technischen Prozesses bzw. einzelner Komponenten. Diese sogenannten Prozeßabbilder enthalten noch die momentanen Werte von bestimmten Prozeßgrößen. Diese Prozeßgrößen sind im Prozeßabbild an den Stellen eingetragen, an denen sie gemessen wurden.
Bild 5-6 zeigt ein Beispiel für ein Prozeßabbild in einer semigraphischen Darstellung /BRJA82/.

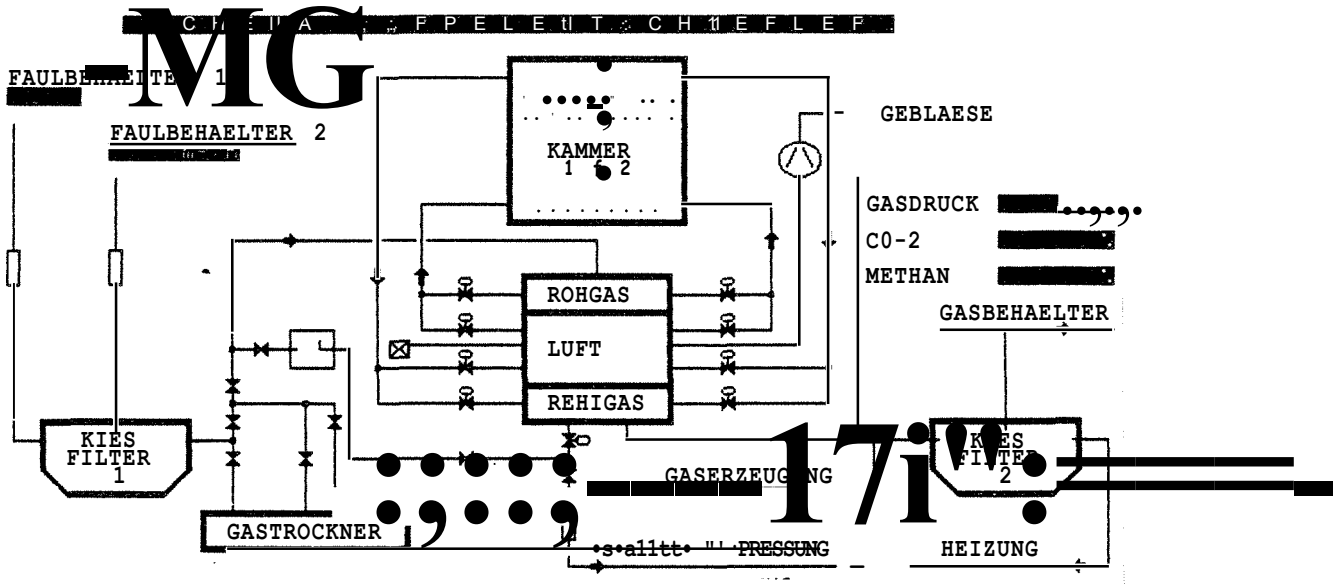


Bild 5.6 : Beispiel für semigraphische Darstellung /BRJA82/

Bei einer semigraphischen Ausgabe entspricht das Ausgabegerät ebenfalls dem Nachrichtenadressaten. Der Name der dargestellten Komponente des technischen Prozesses wird als Nachrichtenname interpretiert (in Bild 5.6 'Doppelentschwefler').
Die in der Graphik eingetragenen Meßwerte entsprechen den Nachrichtenparametern. In Bild 5.6 sind die Meßwerte durch eine helle Schrift auf dunklem Untergrund eingetragen.
Ähnlich wie sich die verschiedenen Ausgabemöglichkeiten als Realisierungen von Botschaften interpretieren lassen, können auch die verschiedenen Eingabearten als Realisierungen von Botschaften gesehen werden.

5.1 .4. Realisierung der Kommunikation zwischen Rechenprozessen

Wann und warum Rechenprozesse kommunizieren, wurde bereits in den vorangegangenen Kapiteln ausreichend diskutiert. Außerdem wurden bereits zahlreiche unterschiedliche Konzepte zur Realisierung der Kommunikation Rechenprozeß-Rechenprozeß vorgestellt.
Der Nachrichtenaustausch zwischen Rechenprozessen kann indirekt über gemeinsame Objekte (siehe /KERA82/) oder direkt über einen Nachrichtenmechanismus abgewickelt werden. Die Vor- und Nachteile dieser Realisierungen bei verteilten Systemen wurden ebenfalls in den vorigen Kapiteln ausführlich diskutiert.

5.2 Entwurf und Spezifikation von Automatisierungssystemen

Beim Entwurf eines Automatisierungsprogramms geht es darum,
- eine geeignete Aufteilung des Automatisierungsproblems auf entsprechende Strukturierungseinheiten zu finden.
- die Namen der Nachrichten zu bestimmen, die die einzelnen Strukturierungseinheiten untereinander oder mit der Umgebung austauschen.
- die Eigenschaften der Strukturierungseinheiten festzulegen.

5.2.1 Strukturierungsmöglichkeiten des vorgeschlagenen Spezifikationskonzepts

Im folgenden geht es darum, Automatisierungsprogramme zu spezifizieren. Im vorigen Abschnitt wurde bereits diskutiert, wie sich die Automatisierungsprogramme in ein Automatisierungssystem eingliedern.
Rechenprozesse in Automatisierungssystemen kommunizieren untereinander, mit dem Benutzer und dem technischen Prozeß durch Botschaften. Allerdings sollen Rechenprozesse unter bestimmten Umständen auch über gemeinsame Objekte verfügen. In dem vorzuschlagenden Spezifikationskonzept gibt es folgende Strukturierungseinheiten:

- Einzelprozesse
- Prozeßbündel
- Prozeß gruppen.

Die Strukturierungseinheit 'Einzelprozeß' ist ein Prozeß im üblichen Sinn, wogegen Prozeßbündel und Prozeßgruppen mehrere Prozesse enthalten.
Die einzelnen Strukturierungseinheiten können untereinander nur über Botschaften kommunizieren bzw. sich synchronisieren. Mit dem Benutzer und dem technischen Prozeß können Informationen ebenfalls nur über Botschaften ausgetauscht werden.
Prozesse, die über gemeinsame Objekte verfügen sollen, werden zu Prozeßgruppen oder Prozeßbündeln zusammengefaßt.
Der Unterschied zwischen einem Prozeßbündel und einer Prozeßgruppe besteht im wesentlichen darin, daß die einzelnen Prozesse einer Prozeßgruppe für andere Strukturierungseinheiten nicht sichtbar sind, wogegen die Prozesse eines Prozeßbündels von anderen Strukturierungseinheiten identifiziert werden können. Sendet z.B. ein Prozeß eine Nachricht an ein Prozeßbündel, so muß er angeben, für welchen Prozeß des Prozeßbündels die Nachricht bestimmt ist. Einem Prozeßbündel ist als Ganzem kein Name zugeordnet.
Soll die Nachricht an eine Prozeßgruppe gesendet werden, so muß nur angegeben werden, an welche Prozeßgruppe sie gehen soll (Prozeßgruppen werden als Gesamtes durch Namen identifi-

ziert). Der Sender kann nicht bestimmen, für welchen Prozeß einer Prozeßgruppe die Nachricht bestimmt ist. Eine Nachricht an eine Prozeßgruppe wird von irgendeinem Prozeß der Prozeßgruppe angenommen.

Die Prozesse eines Prozeßbündels oder einer Prozeßgruppe dürfen untereinander keine Botschaften austauschen. Sie dürfen ihre Zusammenarbeit untereinander nur über gemeinsame Objekte organisieren. Damit wird vermieden, daß sich Sychronisations- und Kommunikationskonzepte für gemeinsame Objekte und Botschaftskonzepte in einem Programm überlagern. Das Synchronisations- und Kommunikationsverhalten eines Automatisierungsprogramms wäre nur noch schwer zu überschauen.

Die Prozesse eines Prozeßbündels bzw. einer Prozeßgruppe müssen sich alle auf derselben Komponente eines verteilten Systems befinden (siehe Kapitel 4).

Prozeßbündel und Prozeßgruppen wurden eingeführt, da es in der Prozeßautomatisierung durchaus vorteilhaft sein kann, wenn mehrere Prozesse über gemeinsame Objekte verfügen.

Ein Beispiel für gemeinsame Objekte sind Dateien, die von mehreren Prozessen beschrieben werden. Läßt man nur Nachrichtenmechanismen zu, um die Zusammenarbeit von Prozessen zu beschreiben, so muß einer solchen Datei ein Verwaltungsprozeß vorgeschaltet werden. Alle Prozesse, die auf diese Datei zugreifen wollen, teilen dem Verwaltungsprozeß ihren Zugriffswunsch mit. Der Verwaltungsprozeß führt den Auftrag aus und gibt die Ergebnisse an den auftraggebenden Prozeß zurück. Ist für eine solche Datei nur ein solcher Verwaltungprozeß möglich, werden alle Zugriffswünsche sequentiell abgearbeitet. Die parallele Ausführung mehrerer z.B. lesender Zugriffe, ist nicht möglich. Durch nur einen Verwaltungsprozeß würde der Zugriff auf eine solche Datei zum Engpaß werden. Deshalb erscheint es sinnvoll, daß das Spezifikationskonzept gemeinsame Objekte zuläßt, aber mit der bereits erwähnten Einschränkung, daß Prozesse, die ein gemeinsames Objekt besitzen, zusammen mit diesem Objekt auf einem Rechner untergebracht sein müssen (siehe /LISK79/). Durch diese Einschränkung entfallen die in Kapitel 3 geschilderten Implementierungsprobleme.

5.2.1 .1 Aufbau von Einzelprozessen

In der Beschreibung der Einzelprozesse werden folgende Eigenschaften festgehalten:

- Welche Bedeutung bzw. welche Wirkung haben gesendete bzw. empfangene Nachrichten für den betrachteten Prozeß.
- Wann sendet ein Prozeß welche Nachrichten an welchen Kommunikationspartner, bzw. wann empfängt er welche Nachrichten von welchem Partner.
- Welche Berechnungen führt ein Prozeß wann aus (interne Berechnungen).

Ein Einzelprozeß besteht aus folgenden Komponenten (Bild 5.7):

- der Benutzermaschine
- der Kommunikationsmaschine
- der Ablaufsteuerung.

Diese Einteilung wurde gewählt, um bestimmte Probleme auf bestimmte Komponenten konzentrieren zu können.
In dem vorgeschlagenen Spezifikationskonzept haben somit die einzelnen Prozesse die in Bild 5.7 wiedergegebene Struktur.

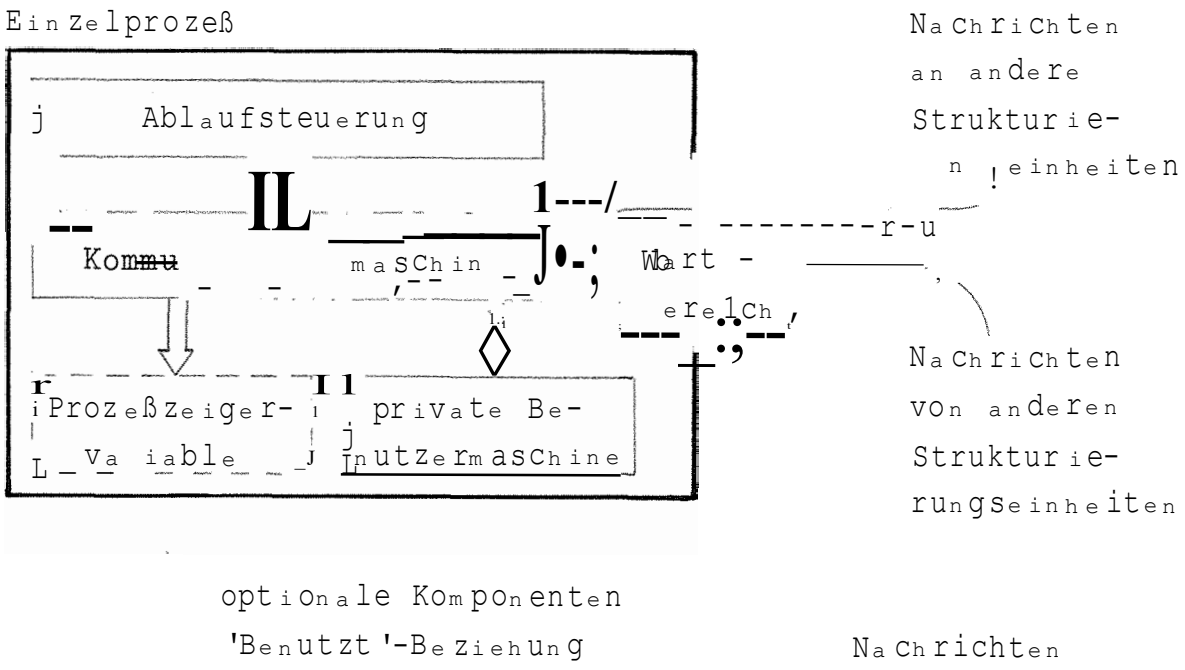


Bild 5.7 : Struktur von Einzelprozessen

Die Wirkung der eintreffenden und die Bedeutung der abgehenden Nachrichten wird durch Funktionen einer abstrakten Maschine definiert. Ebenso werden die Eigenschaften der internen Berechnungen durch Operationen dieser abstrakten Maschine festgelegt. Die Eigenschaften dieser Maschine, die im weiteren Benutzermaschine genannt wird, werden durch den Entwerfer eines Programms definiert.

Beim Senden oder Empfangen von Nachrichten kann ein Prozeß blockiert werden. Das Blockieren eines Prozesses bzw. das Abwickeln des Sendens und Empfangens von Nachrichten wird durch die sogenannte Kommunikationsmaschine ausgeführt. Die Kommunikationsmaschine überprüft die Blockiert- und Nichtblockiertbedingungen beim Senden und Empfangen von Nachrichten. Die Eigenschaften der Kommunikationsmaschine sind im wesentlichen fest vorgegeben. Der Entwerfer kann nur angeben, ob das Empfangen von Nachrichten über einen sogenannten Wartebereich erfolgen soll, d.h., ob ein Prozeß fähig ist, an ihn gerichtete Nachrichten zu puffern. Außerdem kann die Kommunikationsmaschine noch eine vom Entwerfer anzugebende Menge von sogenannten Prozeßzeigervariablen verwalten. Dieser Variablentyp kann als Werte nur Namen von Prozessen enthalten.

Durch die sogenannte Ablaufsteuerung legt der Programmentwerfer fest:
- Wann welche Nachricht an wen gesendet wird.
- Wann welche Nachrichten von wem erwartet werden.
- Wann welche internen Berechnungen durchgeführt werden.

5.2.1 .2. Aufbau von Prozeßbündeln

Ein Prozeßbündel (Bild 5.8) besteht aus:
- der gemeinsamen Benutzermaschine,
- den privaten Benutzermaschinen,
- den Kommunikationsmaschinen und
- den Ablaufsteuerungen.

Ein Prozeßbündel hat somit die in Bild 5.8 gezeigte Struktur.

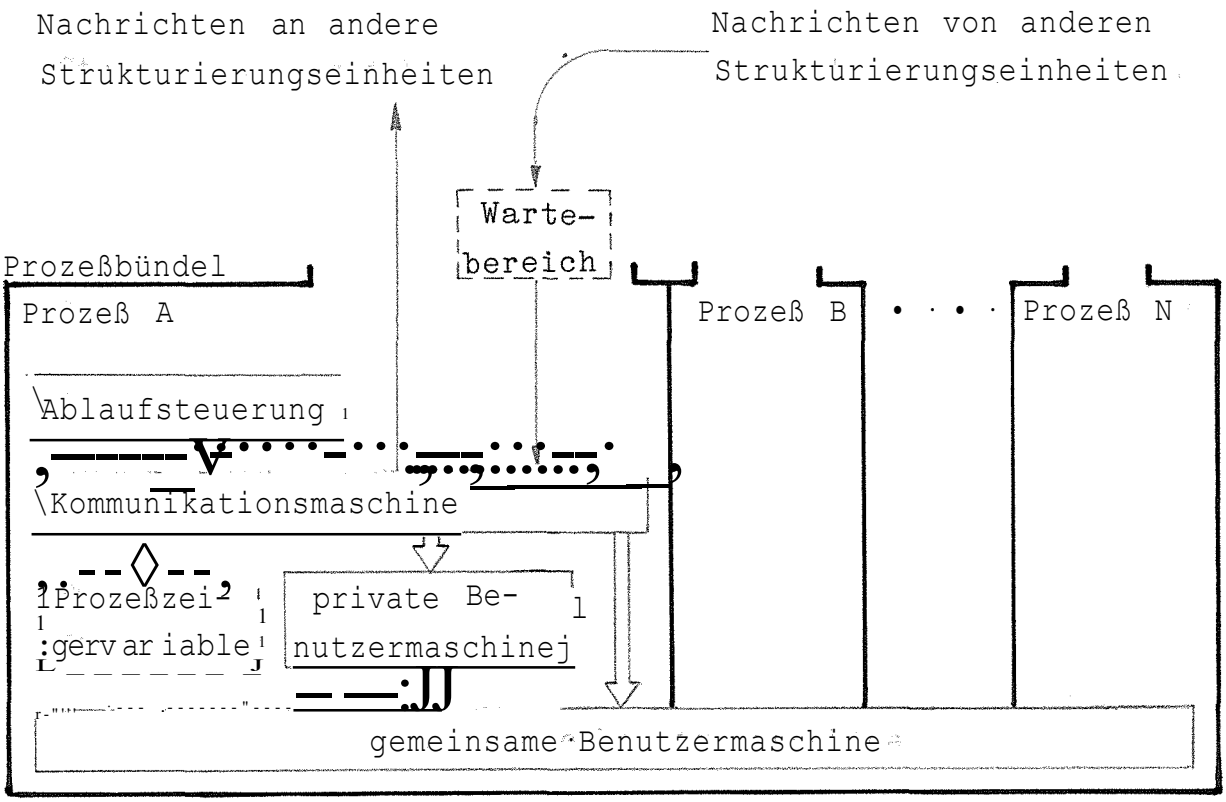


Bild 5.8 : Struktur von Prozeßbündeln

Die Wirkung der eintreffenden Nachrichten wird bei Prozeßbündeln ebenfalls durch Operationen von abstrakten Maschinen beschrieben.
Wie bereits erwähnt, besteht ein Prozeßbündel aus mehreren Prozessen, die über gemeinsame Objekte verfügen. Die gemeinsamen Objekte werden zu einer gemeinsamen Benutzermaschine zusammengefaßt (siehe Abschnitt 2.2.). Die Prozesse eines Prozeßbündels verfügen also wie die Einzelprozesse über eine von ihnen exklusiv verwendete private Benutzermaschine und zusätzlich über eine gemeinsame Benutzermaschine. Der Zugriff auf die gemeinsame Benutzermaschine kann durch ressourcenorientierte Synchronisationskonzepte geregelt werden. Die Operationen zur Beschreibung der Wirkung bzw. der Bedeutung von Nachrichten können nur auf der privaten Benutzermaschine definiert werden.
Die Ablaufsteuerung der einzelnen Prozesse eines Prozeßbündels

hat dieselben Eigenschaften wie bei Einzelprozessen.
Den einzelnen Prozessen innerhalb eines Prozeßbündels können Prioritäten zugeordnet werden.
Jeder Prozeß eines Prozeßbündels verfügt über seine eigene, von ihm exklusiv verwendete Kommunikationsmaschine. Die Kommunikationsmaschinen bei Prozeßbündeln haben dieselben Aufgaben wie bei einzelnen Prozessen (siehe vorigen Abschnitt).

5.2.1 .3. Aufbau von Prozeßgruppen

Prozeßgruppen sind ähnlich aufgebaut wie Prozeßbündel. Bild 5.9 zeigt die Struktur von Prozeßgruppen.

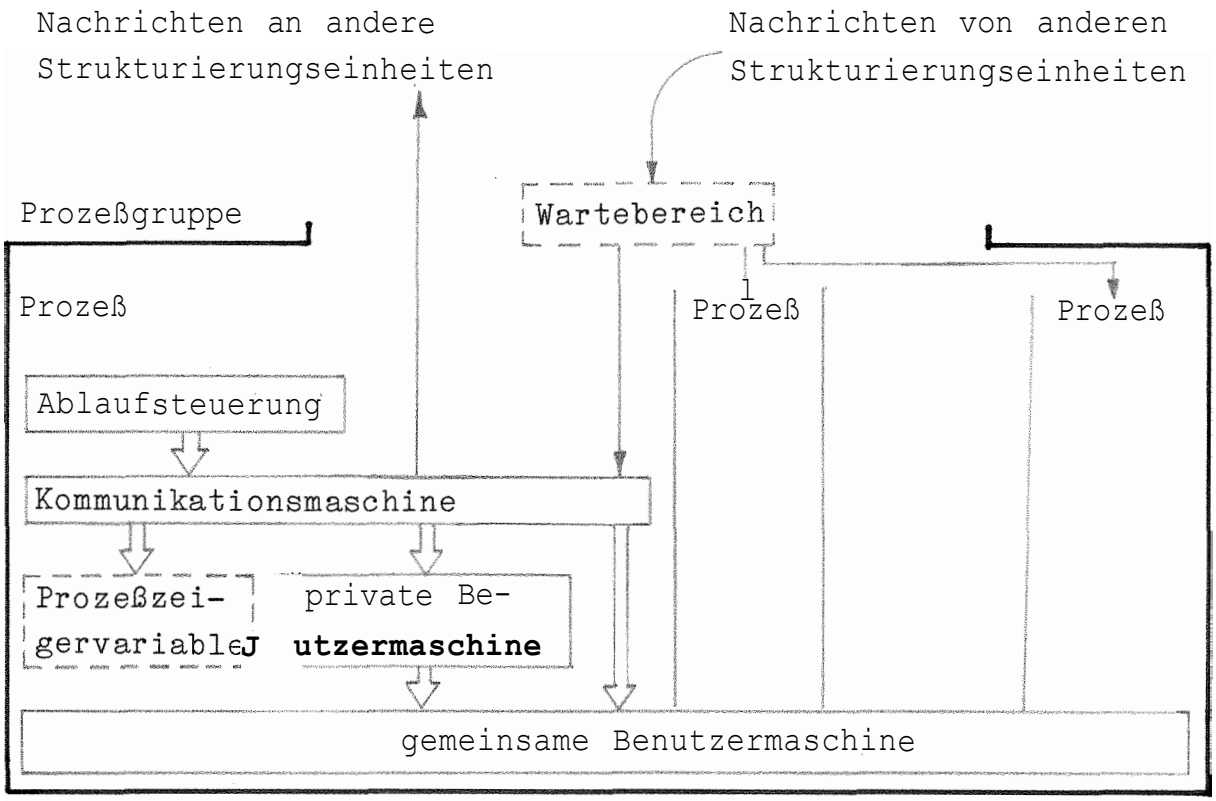


Bild 5. : Struktur von Prozeßgruppen

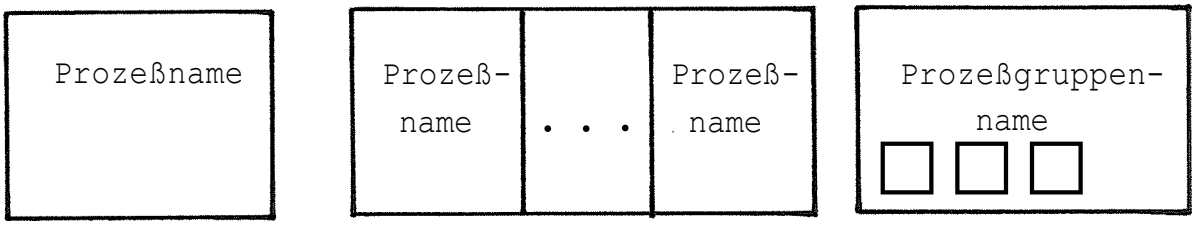
Die einzelnen Prozesse einer Prozeßgruppe verfügen jeweils über eine private und eine gemeinsame Benutzermaschine. Die Operationen zur Beschreibung der Wirkung bzw. Bedeutung von Nachrichten dürfen nur auf den jeweiligen privaten Benutzerma-

schinen definiert sein.
Wie bereits erwähnt, sind die einzelnen Prozesse einer Prozeßgruppe von außen nicht identifizierbar. Einer Prozeßgruppe kann deshalb nur als Gesamtes ein Wartebereich vorgelagert werden. Die Kommunikationsmaschinen der einzelnen Prozesse haben dieselben Eigenschaften wie bei Einzelprozessen und Prozeßbündeln, mit der Ausnahme, daß sie u.U. auf einem gemeinsamen Wartebereich arbeiten.
Die Ablaufsteuerung der einzelnen Prozesse einer Prozeßgruppe hat dieselben Aufgaben wie bei den Einzelprozessen. Wie bei den Prozeßbündeln können den einzelnen Prozessen in einer Prozeßgruppe Prioritäten zugeordnet werden.

5.2.2 Die Beziehung von Einzelprozessen, Prozeßbündeln und Prozeßgruppen untereinander bzw. zueinander

In diesem Abschnitt wird die Kommunikation von Einzelprozessen, Prozeßbündeln und Prozeßgruppen untereinander bzw. zueinander betrachtet (Kommunikationsstruktur).
Die Kommunikationsstruktur ergibt sich dadurch, daß das zu erstellende Automatisierungsprogramm in entsprechende Strukturierungseinheiten zerlegt wird und die Namen der Nachrichten festgelegt werden, die die einzelnen Strukturierungseinheiten senden bzw. empfangen können. Beim Entwurf der Kommunikationsstruktur ist es wichtig, daß der technische Prozeß miteinbezogen wird. Dazu kann der technische Prozeß ebenfalls in entsprechende Komponenten gegliedert werden, die untereinander bzw. mit den Strukturierungseinheiten Nachrichten austauschen. Eine graphische Darstellung der Kommunikationsstruktur erhält man durch eine Darstellung der einzelnen Strukturierungseinheiten als Knoten eines Graphen. Die Knoten werden mit den Prozeß- oder Prozeßgruppennamen markiert. Zu allen Prozessen bzw. Prozeßgruppen, an die der betrachtete Prozeß bzw. Prozeßgruppe Nachrichten sendet, wird eine Kante gezogen, die mit dem entsprechenden Nachrichtennamen markiert wird (siehe Bild 5.11).
Um Einzelprozesse, Prozeßbündel und Prozeßgruppen voneinander

unterscheiden zu können, werden für deren Darstellung in der Kommunikationsstruktur unterschiedliche Knotentypen verwendet. Bild 5.10 zeigt die unterschiedlichen Knotentypen.



a) Einzelprozeß b) Prozeßbündel c) Prozeßgruppen

Bild 5.10: Symbole für die Strukturierungseinheiten

Bild 5. zeigt ein einfaches Beispiel für eine Kommunikations-struktur.

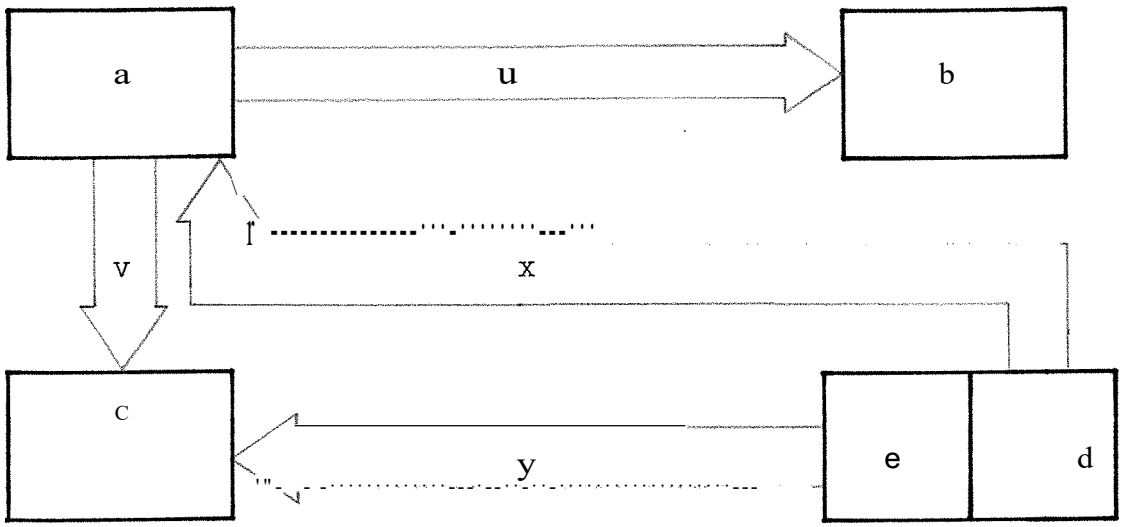


Bild 5.11 : Beispiel für eine Kommunikationsstruktur.

Der Einzelprozeß a sendet die Nachricht u an den Einzelprozeß b und die Nachricht v an die Prozeßgruppe c. Der Prozeß d aus einem Prozeßbündel sendet die Nachricht x an den Einzelprozeß a, und der Prozeß e aus dem Prozeßbündel sendet die Nachricht y an die Prozeßgruppe c.

5.2.3 Die Beschreibung der Bestandteile des Spezifikationskonzepts

5.2.3.1 Die Benutzermaschine

Bei der Beschreibung der Benutzermaschinen muß unterschieden werden, ob eine private oder eine gemeinsame Benutzermaschine definiert werden soll.
Private Benutzermaschinen können mit Hilfe von Techniken, wie sie bei der sequentiellen Programmierung üblich sind, beschrieben werden.
Bei gemeinsamen Benutzermaschinen gibt es jedoch das bereits ausführlich erläuterte Problem der Synchronisation. Die Synchronisation der Zugriffe auf eine gemeinsame Benutzermaschine soll ausschließlich in den die gemeinsame Benutzermaschine bildenden Objekten durch ressourcenorientierte Synchronisationskonzepte erfolgen. Für die Kommunikationsmaschine sind die privaten und gemeinsamen Benutzermaschinen abstrakte Maschinen, wie sie im folgenden Abschnitt definiert werden. Dadurch wird gewährleistet, daß die Kommunikationsmaschine unabhängig von der speziell gewählten Spezifikationstechnik für private und gemeinsame Benutzermaschine wird.

5.2.3.1 .1 Definition: abstrakte Maschine

Eine abstrakte Maschine stellt einem Benutzer eine Menge von Operationen zur Verfügung. Mit diesen Operationen kann der Zustand der abstrakten Maschine verändert werden. Daneben gibt es eine Menge von Funktionen, mit denen der Zustand der abstrakten Maschine abgefragt werden kann (siehe /WOER80/).

Definition:

Eine abstrakte Maschine A ist ein Quintupel (SA, FA, OA, PA, VI), wobei gilt:

a) SA ist die Menge der internen Zustände von A
b) PA ist die Menge der möglichen Zustände der Operationsparameter (Eingabeparameter) und VI die Menge der möglichen Zustände der Funktionsparameter (Rückgabeparameter)
b) OA ist eine Menge von Operationen, die den internen Zustand von OA verändern, d.h. alle a OA sind Abbildungen der Art a SA x PA SA.

   Der Folgezustand hängt nicht nur vom momentanen Zustand der Maschine ab, sondern auch von den expliziten Parametern, mit denen eine Funktion aufgerufen wird.
c) FA ist eine Menge von Funktionen f SA VI, wobei VI die Menge der möglichen Ergebnisse der Funktion f ist.

Im folgenden sollen die internen Zustände einer abstrakten Maschine als Datenzustände bezeichnet werden.

5.2.3.1 .2 Aufgaben der Benutzermaschine

Die Funktionen und Operationen der privaten und der gemeinsamen Benutzermaschine lassen sich in vier disjunkte Mengen einteilen:

- Eingabeoperationen
- Ausgabefunktionen
- interne Operationen
- interne Funktionen

Die Menge der Funktionen und Operationen der gemeinsamen Benutzermaschine läßt sich in zwei disjunkte Teilmengen zerlegen:

- interne Funktionen
- interne Operationen

Eingabeoperationen und Ausgabefunktionen

Die Operationen, die die Wirkung einer ankommenden Nachricht beschreiben, werden Eingabeoperationen genannt. Der Folgezu-

stand der Benutzermaschine hängt vom momentanen Zustand und den Nachrichtenparametern ab, die den Operationsparametern entsprechen. Die Ausgabefunktion bestimmt die Nachrichtenparameterwerte einer zu sendenden Nachricht. Diese Werte hängen nur vom momentanen Zustand der Benutzermaschine ab.

Die Eingabeoperation und die Ausgabefunktion einer Nachricht sind im allgemeinen auf den privaten Benutzermaschinen verschiedener Prozesse definiert. Die Ausgabefunktion einer Nachricht ist in den privaten Benutzermaschinen der Prozesse definiert, die diese Nachricht senden, die Eingabeoperation in den Prozessen, die die entsprechende Nachricht erwarten. Wird eine Nachricht in einem Prozeß sowohl empfangen als auch gesendet, sind in dessen privater Benutzermaschine sowohl die Eingabeoperation als auch die Ausgabefunktion definiert.
Da eine Nachricht desselben Namens von verschiedenen Prozessen gesendet bzw. empfangen werden kann, folgt, daß die Eingabeoperationen und die Ausgabefunktionen solcher Nachrichten in verschiedenen Prozessen verschieden definiert sein können.
Das Senden von Nachrichten besteht aus zwei Vorgängen:
- dem Aufruf der entsprechenden Ausgabefunktion und
- dem Übertragungsvorgang.

Diese Vorgänge werden in der wiedergegebenen Reihenfolge von der Kommunikationsmaschine ausgeführt.
Analog dazu besteht das Empfangen ebenfalls aus zwei Vorgängen, die in der angegebenen Reihenfolge von der Kommunikationsmaschine ausgeführt werden:
- dem Annahmevorgang und
- dem Aufruf der entsprechenden Eingabeoperation.

Diese Vorgänge werden detailliert in Abschnitt 5.2.4 beschrieben.

Interne Funktionen und Operationen

Mit den internen Funktionen und Operationen werden die nach außen nicht direkt sichtbaren Berechnungen eines Prozesses beschrieben. Das Ergebnis des Aufrufs einer internen Funktion

veranlaßt u.U. den Prozeß, in andere Teile seines Codes zu verzweigen. Mit den internen Operationen wird der Zustand der abstrakten Maschine von dem entsprechenden Prozeß selbst verändert, im Gegensatz zu den Eingabeoperationen, wo der Anstoß zum Aufruf von einem anderen Prozeß aus erfolgt.
Die internen Operationen und Funktionen können auf der privaten oder der gemeinsamen Benutzermaschine definiert sein.
Ist eine interne Operation oder Funktion auf der privaten Benutzermaschine definiert, so darf die private Benutzermaschine zu deren Definition auch Operationen bzw. Funktionen als Leistungen einer tiefer liegenden abstrakten Maschine benutzten. Der privaten Benutzermaschine muß es möglich sein, Operationen und Funktionen der gemeinsamen Benutzermaschine zu verwenden, damit Daten von der privaten zur gemeinsamen Benutzermaschine (und umgekehrt) geschafft werden können. Dadurch erst können Prozesse eines Prozeßbündels oder einer Prozeßgruppe untereinander Daten austauschen.
Bei den durch die private Benutzermaschine verwendeten Operationen und Funktionen der gemeinsamen Benutzermaschine kann es sich auch um Funktionen und Operationen handeln, die nicht zur obersten Schicht der gemeinsamen Benutzermaschine gehören (Funktionen und Operationen die von der Ablaufsteuerung nicht direkt aufgerufen werden).

5.2.3.1 .3 Möglichkeiten zur Beschreibung der privaten Benutzermaschine

Wie bereits in Abschnitt 2.1 diskutiert, können abstrakte Maschinen durch Moduln aufgebaut werden. Der Zusammenhang zwischen Moduln und abstrakter Maschine wurde ebenfalls in diesem Abschnitt diskutiert. Bei der Beschreibung einer abstrakten Maschine geht es also hauptsächlich um die Beschreibung von Moduleigenschaften (Objekten).
Die Eigenschaften von Moduln können informell und formal beschrieben werden. Dabei soll aber die informelle Beschreibung nur eine Vorstufe einer formalen Beschreibung sein. Bei den formalen Beschreibungstechniken werden im wesentlichen zwei

Techniken unterschieden:
- prozedurale Beschreibungstechniken und
- nichtprozedurale Beschreibungtechniken.

Beide Beschreibungstechniken ermöglichen eine hierarchische Beschreibung der Eigenschaften einer abstrakten Maschine. (siehe Kapitel 2).
Die Wahl einer geeigneten Spezifikationstechnik hängt wesentlich vom Kenntnisstand des Entwerfers ab. Informelle und prozedurale Spezifikationstechniken sind demgemäß für Nichtinformatiker schnell erlernbar.

Prozedurale Beschreibungstechniken

Unter diese Beschreibungstechniken fallen vor allem alle gängigen Programmiersprachen (z.B. PEARL, PASCAL). Bei dieser Beschreibungstechnik werden Algorithmen angegeben, die die einzelnen Operationen und Funktionen der abstrakten Maschine realisieren. Die Operationen und Funktionen werden auf Prozeduren abgebildet. Diese Prozeduren arbeiten auf im allgemeinen gemeinsamen Datenvariablen. Der Zustand der abstrakten Maschine ist durch die Wertebelegung dieser Variablen gegeben.
Die Prozeduren, die die einzelnen Funktionen und Operationen beschreiben, benutzen u.U. Hilfsprozeduren. Diese Hilfsprozeduren können als Operationen und Funktionen einer tieferliegenden abstrakten Maschine betrachtet werden.
Mit prozeduralen Beschreibungstechniken wird in erster Linie eine konkrete Realisierung angegeben und nicht die allgemeinen Eigenschaften einer abstrakten Maschine. An eine Spezifikation wird aber die Anforderung gestellt, daß nur die Eigenschaften einer abstrakten Maschine angegeben werden sollen und durch die Beschreibung keine konkrete Realisierung nahegelegt werden soll.
Ein Beispiel für eine Spezifikationssprache, die u.a. prozedurale Beschreibungstechniken benutzt, wurde in Kapitel 3 (/BOCH79/) vorgestellt.

Nichtprozedurale Beschreibungstechniken

Bei den nichtprozeduralen Beschreibungstechniken werden im

wesentlichen folgende Methoden unterschieden:
- die algebraische Spezifikation
- und die Prädikatentransformation

Mit diesen Beschreibungstechniken werden abstrakte Datentypen beschrieben. Die Beschreibung der abstrakten Datentypen kann ebenfalls hierarchisch sein (siehe folgende Ausführungen). Um einen abstrakten Datentyp zu beschreiben, werden andere, bereits definierte, abstrakte Datentypen benutzt (siehe /BAWO81 /, /PBBP82/).

Algebraische Spezifikation:
Bei der algebraischen Spezifikation werden Datentypen als algebraische Strukturen aufgefaßt, also als Mengen (Objektmengen) mit darauf definierten Operationen und Funktionen, für die ein System von Gesetzen (Axiomen) gilt /KKST79/.
Da in der Informatik alle Objektmengen endlich darstellbar sein müssen, beschränkt man sich bei der algebraischen Spezifikation auf Algebren, die folgendem Erzeugungsprinzip genügen /PBBP82/:
Jedes Element der Objektmenge kann - ausgehend von nullstelligen Operationen (Konstanten) - durch endlichmalige Anwendung von Operationen der Algebra erhalten werden.
Bei der algebraischen Spezifikation wird also die Objektmenge nicht explizit angegeben.
Die Zustände der abstrakten Datentypen und damit der abstrakten Maschinen werden durch die Ergebnisse der Funktionsaufrufe indirekt wiedergegeben.
Der erste Teil der Spezifikation eines abstrakten Datentyps besteht aus der Angabe des Werte- und Definitionsbereichs der den Datentyp charakterisierenden Funktionen. Dabei wird zwischen aufbauenden und nicht aufbauenden Funktionen unterschieden. Ein Exemplar des zu beschreibenden Datentyps wird repräsentiert durch die Hintereinanderausführung von aufbauenden Funktionen. In den Axiomen, dem zweiten Teil der Spezifikation, werden die Wirkungen der nichtaufbauenden Funktionen auf die Exemplare eines Datentyps beschrieben.
'Die Axiome werden in Form von Gleichungen gegeben, in denen

die Ausdrücke der rechten Seite aus Variablen und Konstanten der Wertebereichstypen mit Hilfe der aussagenlogischen Operatoren, der Fallauswahl und der Rekursion gebildet werden' /KKST79/.

Operationen eines abstrakten Datentyps können u.U. nur aufgerufen werden, wenn sich das betreffende Objekt in bestimmten Zuständen befindet. Bei dem in Bild 5.12 dargestellten abstrakten Datentyp handelt es sich um einen Keller, in dem nur eine bestimmte Anzahl von entsprechenden Datenobjekten gespeichert werden kann. Wird z.B. die Operation PUSH auf einen vollen Keller ausgeführt, führt dies zu einem Fehler. Dieser Teil der Spezifikation wird mit 'restrictions' bezeichnet.

Abstrakte Datentypen können parametrisiert sein (Typ-Generatoren). Das Beispiel in Bild 5.12 hat als Parameter die Objektart, die in diesem Keller gespeichert werden kann.

Ein System, das es ermöglicht, abstrakte Datentypen zu spezifizieren und die Spezifikationen verschiedenen Prüfungen zu unterziehen, ist in /MUSS80/ beschrieben.

```
     generator  FSTACK (ITEM)
functions  mtstack  :   NAT            ->   FSTACK
           push     :   FSTACK x ITEM  ->   FSTACK
           pop      :   FSTACK         ->   FSTACK
           top      :   FSTACK         ->   ITEM
           depth    :   FSTACK         ->   NAT
           limit    :   FSTACK         ->   NAT

axioms for k  NAT, i  ITEM, s  FSTACK  let
           pop (push (s' i) )  = s
           top (push (s' i) )  = i
           depth (mtstack (k))  = 0
           depth (push (s, i) ) = depth (s) + 1
           limit (mtstack (k))  = k
           limit (push (s, i))  = limit (s)

restrictions
           depth (s) ≥ limit (s)  =>  push (s' i)  = error2
           depth (s)= 0           =>   pop (s)     = error1
           depth (s)= 0           =>   top (s)     = undefined

endtype generator  FSTACK
```

Bild 5,12: Spezifikation eines Kellers beschränkter Tiefe /KKST79/

Prädikatentransformation:

Bei der Prädikatentransformation, die auf Dijkstra zurückgeht /DIJK76/, werden abstrakte Datentypen durch eine abstrakte Objektmenge und darauf definierten abstrakten Operationen und Funktionen beschrieben. Die Wirkung der Operationen wird dabei durch Prädikate über dem abstrakten Objekt beschrieben.

Gilt für ein abstraktes Objekt vor der Ausführung der Operation op das Prädikat P(y), so gilt danach das Prädikat Q(y). Dies wird nach folgendem Muster notiert:

{P(y)} op(y) {Q(y)}

Die Operation op(y) kann nur dann aufgerufen werden, wenn für

das entsprechende abstrakte Objekt das Prädikat P(y) gilt (momentanter Zustand des abstrakten Objekts y). Da die Prädikate P und Q Aussagen über das zu definierende Objekt machen, ist eine mathematische Repräsentation der abstrakten Objekte vor der Einführung des abstrakten Datentyps erforderlich /KERA82/ (siehe auch beim Abstraktionsmechanismus in CLU /LISK77/). Als Beispiel /KERA82/ soll der abstrakte Datentyp 'Warteschlange aus Elementen des Typs T mit der maximalen Kapazität n' beschrieben werden. Dabei wird vorausgesetzt, daß der abstrakte Datentyp 'Sequence of T' bereits definiert ist. Ein abstraktes Objekt des neu eingeführten Datentyps kann durch ein Objekt 'Sequence of T' repräsentiert werden, was im folgenden durch die Klausel 'AR as sequence of T' zum Ausdruck gebracht wird.

```
MODULE sequence
q, q1: queue AR as sequence of T
t' x : T
n    : integer /*maximale ws Länge/
```

{|q| < n ∧ q = q1}	end (q,t)	{q = q1•t}
{q = t.q1}	deg(q,x)	{q = q1 ∧ x = t}
{q = t.q1}	first(q,x)	{q = t •q1 ∧ x = t}

Bild 5.13 : Spezifikation einer Warteschlange mit beschränkter Kapazität

5.2.3.1 .4 Möglichkeiten zur Beschreibung der gemeinsamen Benutzermaschinen

Zur Spezifikation von gemeinsamen Benutzermaschinen eignen sich Spezifikationssprachen, die ressourcenorientierte Synchronisationskonzepte enthalten. Ressourcenorientierte Synchronisationskonzepte wurden in Abschnitt 3.1 vorgestellt. Werden die Objekte einer gemeinsamen Benutzermaschine prozedu-

ral beschrieben, können zur Synchronisation z.B. Monitore verwendet werden. Ein nichtprozedurales, sehr flexibles Konzept zur Beschreibung gemeinsamer Objekte ist das in Abschnitt 3.1.1.2 kurz vorgestellte Konzept nach Keramidis /KERA81/.

5.2.3.2 Die Kommunikationsmaschine

Die Kommunikationsmaschine stellt der Ablaufsteuerung folgende Funktionen zur Verfügung:
- Senden von Telegrammen
- Empfangen von Telegrammen
- Ausführen von internen Operationen
- Ausführen von internen Funktionen

Die Kommunikationsmaschine führt diese Funktionen mit Hilfe der privaten und der gemeinsamen Benutzermaschine, dem Wartebereich und den Prozeßzeigervariablen aus.
Die Kommunikationsmaschine hat für Einzelprozesse, Prozeßbündel und Prozeßgruppen dieselben Aufgaben und Eigenschaften.
Die Möglichkeiten der Kommunikationsmaschine sind weitgehend festgelegt. Der Entwerfer eines verteilten Programms kann bei den Kommunikationsmaschinen der einzelnen Prozesse nur angeben, ob Nachrichten an diesen Prozeß über einen Wartebereich gesendet werden können (asynchrone Kommunikation) oder direkt übergeben werden müssen (synchrone Kommunikation). Der Wartebereich dient also zum Zwischenpuffern von Nachrichten. Der Entwerfer hat die Möglichkeit anzugeben, wieviel Nachrichten maximal zwischengepuffert werden können.
Außerdem kann die Kommunikationsmaschine och eine beliebige, vom Programmentwerfer festlegbare, Menge von sogenannten Prozeßzeigervariablen verwalten. Dieser Variablentyp kann als Werte nur Namen von Prozessen enthalten und dient dazu, daß der Adressat bzw. der Absender einer Nachricht indirekt angegeben werden kann.
Beim Senden und Empfangen erhält die Kommunikationsmaschine als Parameter die zu sendenden bzw. zu empfangenden Telegramme, eventuell eine Liste von Prozeßzeigervariablen und u.U.

ein Zeitintervall, das angibt, wie lange ein Prozeß beim Senden oder Empfangen maximal blockiert werden darf.
Beim Ausführen von internen Funktionen und Operationen erhält die Kommunikationsmaschine als Parameter nur die Bezeichnung der auszuführenden Funktion bzw. Operation.
Telegramme haben die durch folgende Syntax beschriebene Form und werden im weiteren als Kommunikationsausdrücke bezeichnet.

'UND', 'EXOR', '(' ')' '<' '>' und '!' sind terminale Symbole.

```
Kommunikationsausdruck::= Kommunikationsterm J
                          Kommunikationsausdruck EXOR
                          Kommunikationsterm;

Komunikationsterm      ::= Kommunikationselement !
                          Kommunikationselement UND
                          Kommunikationsterm;

Kommunikationselement  ::= (Absender-/Adreßangabe) Nachricht;

Absender/Adreßangabe   ::= direkte_Angabe ! indirekte_Angabe;

direkte_Angabe         ::= Prozeßname;

indirekte_Angabe       ::= <Prozeßzeigervariablenname>;

Nachricht              ::= Nachrichtenname(Nachrichtenparameter);
```

Gehört ein Kommunikationselement zu einem Kommunikationsausdruck, der gesendet werden soll, so wird mit der 'Absender-/Adreßangabe' der Prozeß angegeben, an den die Nachricht gehen soll (Zielprozeß). Soll ein Kommunikationsausdruck empfangen werden, so wird durch die Adreß-/Absenderangabe der Prozeß angegeben, von dem die Nachricht gewünscht wird (Quellprozeß). Die Absender-/Adreßangabe kann direkt oder indirekt erfolgen. Direkte Absender-/Adreßangabe heißt, daß der Quell- oder Zielprozeß durch einen konstanten Prozeßnamen angegeben wird.

Erfolgt die Adreß- oder Absenderangabe über Prozeßzeigervariable, so spricht man von einer indirekten Quell- bzw. Zielprozeßangabe. Die Möglichkeiten, die Werte von Prozeßzeigervariablen zu verändern bzw. zu lesen, werden in diesem Kapitel später beschrieben.
Wird in einem Kommunikationselement der Adressat bzw. der Absender direkt angegeben, so wird es als direktes Kommunikationselement, im anderen Fall als indirektes Kommunikationselement bezeichnet. Ein Kommunikationsterm, der ausschließlich direkte Kommunikationselemente enthält, wird direkter Kommunikationsterm genannt. Ein Kommunikationsausdruck, der nur aus direkten Kommunikationstermen besteht, heißt direkter Kommunikationsausdruck. Ist in einem Kommunikationsterm bzw. in einem Kommunikationsausdruck ein indirektes Kommunikationselement bzw. ein indirekter Kommunikationsterm enthalten, so sind dies indirekte Kommunikationsterme bzw. indirekte Kommunikationsausdrücke.
Enthält ein Kommunikationsausdruck direkte Kommunikationsterme, so kann zu jedem direkten Kommunikationsterm noch eine Folge von Prozeßzeigervariablen angegeben werden. Die Elemente einer solchen Folge von Prozeßzeigervariablen müssen paarweise voneinander verschieden sein. Die Bedeutung und Eigenschaften der Prozeßzeigervariablen werden zusammen mit der Beschreibung des Sendens und Empfangen von Kommunikationsausdrücken näher beschrieben.

5.2.3.2.1 Das Senden

Das Senden eines Kommunikationselementes

Kommunikationselemente werden von Prozessen gesendet (Sende- oder Quellprozesse). Sendeprozesse können Einzelprozesse oder Prozesse innerhalb von Prozeßbündeln oder Prozeßgruppen sein. Kommunikationselemente können an Einzelprozesse, an einen bestimmten Prozeß eines Prozeßbündels oder an eine Prozeßgruppe gesendet werden (Empfangs- oder Zielprozesse).

Für das Senden eines Kommunikationselementes gilt:

- Die Kommunikationsmaschine erhält beim Aufruf der Sendefunktion durch die Ablaufsteuerung als Parameter ein Kommunikationselement, das aus der zu sendenden Nachricht und dem Namen des Zielprozesses bzw. der Zielprozeßgruppe (Adressat oder Ziel) besteht. Der Adressat kann entweder ein Einzelprozeß, ein Prozeß innerhalb eines Prozeßbündels oder eine gesamte Prozeßgruppe sein.
- Der Zielprozeß bzw. die Zielprozeßgruppe erhält ein Kommunikationselement, das aus der Nachricht und dem Namen des sendenden Prozesses bzw. der sendenden Prozeßgruppe besteht (Absender oder Quelle).

Der sendende Prozeß weiß somit, an welchen Prozeß bzw. welche Prozeßgruppe eine Nachricht gesendet werden soll und der Zielprozeß weiß, von welchem Prozeß bzw. Prozeßgruppe eine Nachricht kommt (siehe Bild 5.14).

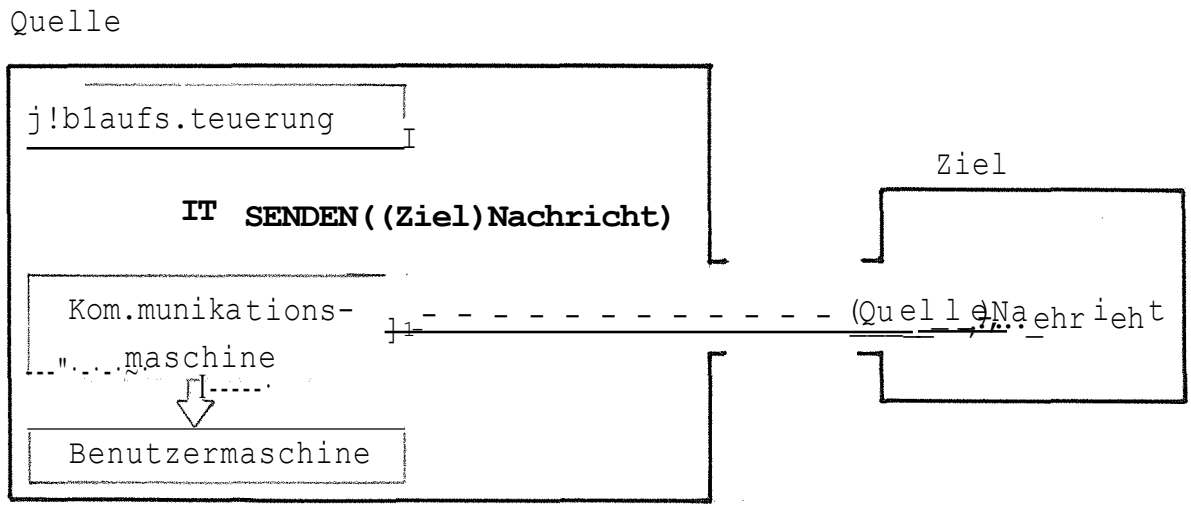


Bild 5.14 : Senden eines Kommunikationselements

Soll die Kommunikationsmaschine eines Prozesses ein Kommunikationselement senden, dann wird zuerst die entsprechende Ausgabefunktion zum Bestimmen der Nachrichtenparameter aufgerufen und dann wird der Übertragungsvorgang begonnen.
Beim Übertragen eines Kommunikationselementes werden zwei

Fälle unterschieden:
- dem Ziel ist ein Wartebereich vorgelagert,
- dem Ziel ist kein Wartebereich vorgelagert.

Ist dem Ziel ein Wartebereich vorgelagert, bedeutet 'übertragen' zu versuchen, die Nachricht und den Namen der Quelle im Wartebereich des Adressaten abzulegen. Der Übertragungsvorgang gilt dann als beendet, wenn die Nachricht und der Name des Absenders im Wartebereich abgelegt werden konnte. Der Sendeprozeß wird nur solange blockiert, bis das Kommunikationselement übertragen wurde.
Der Übertragungsvorgang kann nicht beendet werden, wenn der Wartebereich voll ist, d.h. das Kommunikationselement nicht abgelegt werden kann. In diesem Fall wird der Sendeprozeß solange blockiert, bis entweder Platz im Wartebereich wird, oder der Übertragungsvorgang wegen Zeitüberschreitung abgebrochen wird (siehe weiter unten).
Ist dem Adressaten kein Wartebereich vorgelagert, bedeutet 'übertragen', daß versucht wird, das Kommunikationselement direkt zu übergeben (Rendezvous). Wird das Kommunikationselement bereits vom Ziel erwartet, kann der Übertragungsvorgang abgeschlossen werden; der sendende Prozeß wird nur solange blockiert, bis der Zielprozeß das Kommunikationselement entgegengenommen hat. Ansonsten wird der Sendeprozeß solange blokkiert, bis der Empfänger das Kommunikationselement annimmt, oder der Übertragungsvorgang wegen Zeitüberschreitung abgebrochen wird. Der Ablaufsteuerung wird jeweils mitgeteilt, ob das Kommunikationselement gesendet werden konnte, oder ob die Zeitüberwachung abgelaufen ist. Die Ablaufsteuerung kann dann entsprechend reagieren.

Senden von Kommunikationsausdrücken

Kommunikationsausdrücke werden, ebenfalls wie Kommunikationselemente, von Prozessen gesendet. Es kann sich dabei um einen Einzelprozeß oder um einen Prozeß innerhalb eines Prozeßbündels oder einer Prozeßgruppe handeln.
Kommunikationsausdrücke bestehen, wie oben beschrieben, aus Kommunikationstermen. Der Sendeprozeß wird nun solange blok-

kiert, bis er genau einen Kommunikationsterm übertragen hat oder die Zeitüberwachung abgelaufen ist. Bei der Übertragung eines Kommunikationsterms wird bei den einzelnen Kommunikationselementen ebenfalls der Name des Adressaten durch den Namen des Absenders ersetzt, damit der Adressat 'weiß', von welchem Prozeß eine Nachricht kommt.

Ein Kommunikationsterm kann dann übertragen werden, wenn alle die in diesem Kommunikationsterm angegebenen Nachrichten zusammen mit dem Absendernamen gleichzeitig übertragen werden können. Gleichzeitig übertragen heißt: Wird eine Nachricht zusammen mit dem Absendernamen übertragen, muß gewährleistet sein, daß die anderen Nachrichten des Kommunikationsterms zusammen mit dem Absendernamen auch übertragen werden können. Der Sendeprozeß darf zwischen dem Senden des ersten und des letzten Kommunikationselements nicht blockiert werden.

Von einem Kommunikationsausdruck wird genau ein Term übertragen

Ein Kommunikationsausdruck wird von links nach rechts abgearbeitet. Kann kein Kommunikationsterm übertragen werden, wird der Sendeprozeß solange blockiert, bis es entweder möglich ist, daß ein Kommunikationsterm übertragen werden kann, oder die Zeitüberwachung abläuft (siehe weiter unten).

Dadurch, daß ein Kommunikationsausdruck von links nach rechts abgearbeitet wird, kann durch die Reihenfolge der Kommunikationsterme festgelegt werden, welcher Term übertragen werden soll, wenn mehrere Terme übertragungsbereit sind (Prioritätenrelation).

Wurde ein direkter Kommunikationsterm übertragen und wurde zu dem Kommunikationsterm als zusätzlicher Parameter eine Folge von Prozeßzeigervariablen angegeben, so werden die Adressatennamen der gesendeten Kommunikationselemente in diese Prozeßzeigervariable übernommen. Die Adressatennamen des gesendeten Kommunikationsterms werden dabei von links nach rechts in die Folge der Prozeßzeigervariablen übertragen.

Der Ablaufsteuerung wird mitgeteilt, welcher Kommunikationsterm letztlich übertragen wurde, bzw. ob die Zeitüberwachung ablief. Die Ablaufsteuerung kann dadurch entsprechend reagieren.

Zeitüberwachung beim Senden

Wie schon angedeutet wurde, kann das Senden eines Kommunikationselements oder eines Kommunikationsausdrucks zeitüberwacht werden. Ein Sendeprozeß wird nur für die angegebene Zeitdauer blockiert. Der Versuch, einen Kommunikationsausdruck zu senden, wird nach dem Ablauf der Zeitüberwachung eingestellt. Der Ablaufsteuerung wird mitgeteilt, daß der Kommunikationsausdruck nicht in der angegebenen Zeit gesendet werden konnte. Die Ablaufsteuerung kann dann entsprechend reagieren. Nach dem Ablauf der Zeitüberwachung ist ein Prozeß nicht mehr blokkiert.

Die Ablaufsteuerung kann die Sendefunktion der Kommunikationsmaschine mit einem leeren Kommunikationsausdruck und einem Zeitintervall für die Zeitüberwachung als Parameter aufrufen. Dann wird der Prozeß für die angegebene Zeitdauer blockiert.

Wird bei einem Aufruf der Sendefunktion keine Zeitdauer für die Zeitüberwachung angegeben, gilt als Voreinstellung die Zeitdauer 'unendlich'.

5.2.3.2.2. Das Empfangen

Empfangen von Kommunikationselementen

Erwartet ein Prozeß eine Nachricht, dann muß er angeben, von welchem Absender (Absender oder Quelle) die Nachricht erwartet wird (erwartetes Kommunikationselement). Beim Annehmen einer Nachricht unterscheidet man zwei Fälle:

- dem Prozeß oder der Prozeßgruppe ist ein Wartebereich vorgelagert, oder
- dem Prozeß oder der Prozeßgruppe ist kein Wartebereich vorgelagert.

Im ersten Fall durchsucht ein Prozeß bzw. die Prozesse einer Prozeßgruppe seinen bzw. ihren Wartebereich nach den erwarteten Nachrichten.

Bei Prozeßgruppen untersuchen alle Prozesse der Prozeßgruppe denselben Wartebereich nach erwarteten Kommunikationselementen. Da die einzelnen Prozesse innerhalb einer Prozeßgruppe

nicht einzeln identifiziert werden können, kann potentiell jeder Prozeß einer Prozeßgruppe ein bestimmtes Kommunikationselement annehmen. Ein Kommunikationselement, das an eine Prozeßgruppe gesendet wurde, wird also von dem Prozeß angenommen, der es als erstes erwartet.
Wird die erwartete Nachricht im Wartebereich gefunden, dann wird geprüft, ob die dieser Nachricht zugeordnete Eingabeoperation ausgeführt werden kann. Falls dies nicht der Fall ist, wird der empfangende Prozeß blockiert.
Diese Blockade hängt von den Werten der Nachrichtenparameter und vom Zustand der Benutzermaschine ab und wird Datenblockade genannt. Eine Datenblockade kann nur beim Empfangen auftreten. Die Datenblockade hält solange an, bis entweder ein gleiches Kommunikationselement mit Nachrichtenparameterwerten eintrifft, die es zulassen, die Eingabeoperation aufzurufen, oder die Zeitüberwachung abläuft (siehe weiter unten). Das Kommunikationselement, dessen Eingabeoperation nicht ausgeführt werden kann, verbleibt im Wartebereich.
Je nachdem, wie die Benutzermaschine beschrieben ist, wird die Möglichkeit der Datenblockade verschieden spezifiziert:

a) Wird die Benutzermaschine durch die Prädikatentransformation beschrieben, dann muß die entsprechende Vorbedingung gelten.
b) Wird die Benutzermaschine durch die algebraische Spezifikation beschrieben, darf der Aufruf der Eingabefunktion nicht zu einem Fehler führen.

Kann die Eingabeoperation ausgeführt werden, dann wird das Kommunikationselement angenommen, d.h.

- das Kommunikationselement wird aus dem Wartebereich entfernt,
- falls Prozeßzeigervariable angegeben sind, werden die Absendernamen in die entsprechenden Prozeßzeigervariablen übernommen,
- es wird die entsprechende Eingabeoperation ausgeführt,
- falls der Wartebereich voll war, wird ein beliebiges von den Kommunikationselementen, deren Übertragung nicht abgeschlossen werden konnte, in den jetzt nicht mehr vollen Wartebereich übernommen. Der entsprechende Sendeprozeß wird deblok-

kiert.

Danach ist das Empfangen eines Kommunikationselementes abgeschlossen.
Befindet sich das erwartete Kommunikationselement nicht im Wartebereich, wird der Prozeß blockiert.
Trifft ein erwartetes Kommunikationselement irgendwann ein, prüft die Kommunikationsmaschine, ob die entsprechende Eingabeoperation ausgeführt werden kann. Ist dies nicht der Fall, bleibt der Prozeß blockiert (Datenblockade), ansonsten wird das Kommunikationselement angenommen.
Die Blockade eines Prozesses wird spätestens nach dem Ablauf der Zeitüberwachung aufgehoben. Nicht angenommene Kommunikationselemente verbleiben im Wartebereich.
Bei den Prozessen bzw. Prozeßgruppen ohne Wartebereich wird geprüft, ob das erwartete Kommunikationselement angeboten wird. In diesem Fall wird es angenommen, wenn die entsprechende Eingabeoperation aufgerufen werden kann. Das bedeutet:
- Der Prozeß, der das Kommunikationselement anbietet, wird deblockiert,
- falls angegeben, wird der Name des Absenders in eine Prozeßzeigervariable übernommen und
- dann wird die Eingabeoperation ausgeführt.

Die Ablaufsteuerung erhält eine Rückmeldung, ob das gewünschte Kommunikationselement empfangen wurde oder ob das Empfangen wegen des Ablaufs der Zeitüberwachung abgebrochen wurde. Die Ablaufsteuerung kann dann entsprechend reagieren.

Empfangen von Kommunikationsausdrücken

Soll ein Kommunikationsausdruck empfangen werden, so müssen alle Kommunikationselemente eines Kommunikationsterms empfangen werden. Ein Prozeß darf vom Annehmen des ersten bis zum Annehmen des letzten Kommunikationselements nicht blockiert werden. Wenn ein Prozeß beginnt, Kommunikationselemente eines Kommunikationsterms anzunehmen, so muß garantiert sein, daß alle Kommunikationselemente des Kommunikationsterms zur Verfügung stehen (die Kommunikationselemente befinden sich im War-

tebereich bzw. werden angeboten) und alle entsprechenden Eingabefunktionen müssen ausgeführt werden können. Die Eingabeoperationen werden in der Reihenfolge ausgeführt, in der die entsprechenden Nachrichten im Kommunikationsterm aufgeführt sind.
Je nachdem, ob dem Prozeß oder der Prozeßgruppe ein bzw. kein Wartebereich vorgelagert ist, hat das Annehmen eines Kommunikationsterms verschiedene Auswirkungen auf den empfangenden Prozeß.
Bei Prozessen bzw. Prozeßgruppen mit Wartebereich bewirkt das Annehmen eines Nachrichtenterms:

- Die Kommunikationselemente, die den Kommunikationsterm bilden, werden aus dem Wartebereich entfernt.
- Falls der Wartebereich voll war, werden Kommunikationselemente, deren Übertragung nicht abgeschlossen werden konnte, in den nun freien Wartebereichsplätzen abgelegt. Die Übertragung dieser,Kommunikationselemente kann abgeschlossen werden.
- Falls angegeben, werden die Absender in die entsprechenden Prozeßzeigervariablen übernommen.
- Die Eingabeoperationen werden in der innerhalb des Kommunikationsterms angegebenen Reihenfolge, von links nach rechts, aufgerufen.

Bei Prozessen bzw. Prozeßgruppen ohne Wartebereich, hat das Annehmen eines Nachrichtenterms dieselbe Wirkung, mit Ausnahme der Aktionen, die den Wartebereich betreffen. Statt die Kommunikationselemente aus dem Wartebereich zu entfernen und die Kommunikationselemente, deren Übertragung nicht abgeschlossen werden konnte, in den freien Wartebereichsplätze abzulegen, werden die entsprechenden Sendeprozesse deblockiert.
Ein Kommunikationsausdruck gilt genau dann als empfangen, wenn ein Kommunikationsterm angenommen wurde. Könnten mehrere Kommunikationsterme eines Kommunikationsausdrucks angenommen werden, so wird derjenige angenommen, der im Kommunikationsausdruck am weitesten links steht.
Der Ablaufsteuerung wird beim Empfangen eines Kommunikationsausdruckes mitgeteilt, welcher Kommunikationsterm letztlich

empfangen wurde, bzw. ob die Zeitüberwachung abgelaufen ist und das Empfangen abgebrochen wurde.

Zeitüberwachung beim Empfangen

Sowohl das Empfangen von Kommunikationselementen als auch von Kommunikationstermen kann bei Prozessen mit und ohne Wartebereich zeitüberwacht werden. Der empfangswillige Prozeß wird nur für die angegebene Zeit blockiert. Kann innerhalb dieser Zeit das erwartete Kommunikationselement oder der gewünschte Kommunikationsausdruck nicht angenommen werden, da das Kommunikationselement bzw. der Kommunikationsausdruck nicht angeboten werden oder sich nicht im Wartebereich befinden bzw. die entsprechenden Eingabeoperationen nicht ausgeführt werden können, so wird der Annahmevorgang abgebrochen.
Wird beim Aufruf der Empfangsfunktion kein Kommunikationsausdruck, sondern nur ein Zeitintervall als Parameter angegeben, so wird der Prozeß nur für die angegebene Zeit blockiert.
Wird beim Empfangen keine Zeitdauer für die Zeitüberwachung angegeben, so gilt, wie beim Senden, die Voreinstellung 'unendlich'.

5.2.3.2.3 Interne Funktionen und Operationen

Die Ablaufsteuerung kann die Kommunikationsmaschine beauftragen, mit Hilfe der Benutzermaschine interne Funktionen bzw. Operationen auszuführen. Bei diesen Aufträgen erhält die Kommunikationsmaschine als Parameter nur den Namen der auszuführenden Funktion bzw. Operation.
Wird eine interne Funktion aufgerufen, erhält die Ablaufsteuerung den Funktionswert zurück. Dadurch kann die Ablaufsteuerung auf die verschiedenen Ergebnisse entsprechend reagieren.
Bei internen Operationen wird der Zustand der Benutzermaschine verändert. Allerdings kann es der Fall sein, daß der Zustand der Benutzermaschine die Ausführung einer Operation nicht zuläßt (restrictions, exceptions). Der Ablaufsteuerung wird mitgeteilt, ob eine interne Operation ausgeführt werden konnte bzw. mit welchem Fehler sie abgebrochen wurde. Die Ablauf-

steuerung kann dann entsprechende Reaktionen veranlassen.
Interne Operationen können auf der privaten oder gemeinsamen Benutzermaschine definiert sein.
Ein Einzelprozeß kann beim Aufruf einer internen Operation bzw. Funktion nicht blockiert werden. Ein Einzelprozeß verfügt nur über eine private Benutzermaschine. In einer privaten Benutzermaschine kann kein Prozeß blockiert werden (Verklemmung).
Prozesse in Prozeßbündeln und Prozeßgruppen können jedoch beim Aufruf von internen Operationen und Funktionen blockiert werden, wenn diese auf der gemeinsamen Benutzermaschine definiert sind (Synchronisation bei gemeinsamen Objekten).

5.2.3.2.4. Bemerkungen zu den Prozeßzeigervariablen

Mit den Prozeßzeigervariablen kann sich ein Prozeß die Namen von Quell- und Zielprozessen merken.
Ein Prozeß kann mit den Prozeßzeigervariablen die Adressen und Absender von Nachrichten indirekt angeben, d.h. die Nachricht wird von dem Prozeß erwartet bzw. zu dem Prozeß gesendet, dessen Name in einer bestimmten Prozeßzeigervariablen steht.
Die Prozeßzeigervariablen sind lokale Daten, die nur unter bestimmten Umständen gelesen oder beschrieben werden dürfen. Das Ändern einer Prozeßzeigervariablen ist nur zusammen mit dem Annehmen einer Nachricht möglich. Eine Prozeßzeigervariable kann nur beim Empfangen und beim Senden gelesen werden.
Durch diese Beschränkungen im Gebrauch der Prozeßzeigervariablen wird verhindert, daß sich ein Prozeß den Adressaten einer Nachricht errechnet und dadurch an beliebige Prozesse Nachrichten senden kann. Ein Prozeß kann bei Verwendung von Prozeßzeigervariablen höchstens an diejenigen Prozesse 'unbefugt' Nachrichten senden, von denen er selbst Nachrichten erhält. Es sind Felder von Prozeßzeigervariablen möglich.
Die Prozeßzeigervariablen haben als mögliche Belegung Prozeßnamen. Die Belegung aller Prozeßzeigervariablen eines Prozesses bezeichnet man als Prozeßzeigervariablenzustand.

5.2.3.2.4 Bemerkungen zum Wartebereich

Um ein höheres Maß an Parallelität zu ermöglichen, kann sich ein Prozeß bzw. eine Prozeßgruppe einen Wartebereich für Kommunikationselemente anlegen (automatische Pufferung). Bei der Definition des Wartebereichs wird nur angegeben, wieviele Kommunikationselemente in ihm Platz haben, unabhängig von der Nachrichtenlänge (Anzahl der Parameter). Die Wartebereichsgröße kann zwischen null und unendlich gewählt werden.
In der Praxis darf aber kein Wartebereich mit unendlicher Größe vorkommen, da der Speicher einer realen Rechenanlage immer begrenzt ist. An einen Prozeß oder Prozeßgruppe mit unendlich großem Wartebereich können Nachrichten asynchron gesendet werden. Der Sender wird nie blockiert. Die Probleme bei asynchronen Botschaftskonzepten wurden bereits in Kapitel 4 diskutiert (Verklemmung durch Speicherplatzbelegung). Dadurch, daß die Wartebereichsgröße vom Programmentwerfer angegeben wird, können diese Probleme nicht mehr auftreten.
Jeder Prozeß oder jede Prozeßgruppe kann sich nur einen Wartebereich anlegen. Der Wartebereich ist diesem Prozeß bzw. dieser Prozeßgruppe dann eindButig zugeordnet. Nur der Prozeß, der den Wartebereich besitzt, darf aus diesem Wartebereich Kommunikationselemente entnehmen. Die anderen Prozesse dürfen nur Nachrichten in dem Wartebereich eines anderen Prozesses ablegen. Dies heißt insbesondere, daß ein in den entsprechenden Wartebereich abgelegtes Kommunikationselement vom Sendeprozeß nicht mehr zurückgeholt werden kann. Kommunikationselemente, die sich im Wartebereich befinden, gelten noch nicht als angenommen (vgl. Abschnitt 5.2.3).
Die Belegung des Wartebereichs mit den verschiedenen Kommunikationselementen wird mit Wartebereichszustand bezeichnet.

5.2.3.3. Die Ablaufsteuerung

Durch die Ablaufsteuerung wird beschrieben, wann welche Aktionen bzw. Reaktionen ausgeführt werden.
Die Aktionen sind die Aufrufe der Funktionen der Kommunika-

tionsmaschine.
Die Reaktionen der Ablaufsteuerung sind das Auswerten der Aktionsergebnisse. Die Aktionsergebnisse sind die Rückmeldungen der Kommunikationsmaschine.

5.2.3.3.1 Die Darstellung der Ablaufsteuerung

Die Darstellung der Ablaufsteuerung erfolgt graphisch. Dabei werden zwei verschiedene Arten von Knoten unterschieden:
- die Kommunikationsknoten bzw. -zustände
- die Internknoten bzw. -zustände.

Die Intern- und Kommunikationszustände zusammen werden als Steuerknoten oder Steuerzustände bezeichnet.
Erreicht eine Ablaufsteuerung einen Kommunikationszustand, dann wird entweder die Funktion 'Senden' oder 'Empfangen' aufgerufen. Gelangt die Ablaufsteuerung zu einem Internknoten, dann wird die Kommunikationsmaschine mit dem Ausführen einer internen Funktion oder Operation beauftragt.
Die Steuerzustände sind durch gerichtete Kanten (Übergänge) verbunden.
Es werden verschiedene Kantentypen unterschieden. Welche Funktion der Kommunikationsmaschine in den verschiedenen Steuerzuständen aufgerufen wird, hängt davon ab, welcher Kantentyp von ihm wegführt.
Von einem Kommunikationsknoten können (exklusiv) entweder Sende- oder Empfangskanten wegführen. Sende- bzw. Empfangskanten sind mit einem Kommunikationsterm und eventuell mit einer Folge von Prozeßzeigervariablennamen beschriftet. Verknüpft man alle Kommunikationsterme der Sende- oder Empfangskanten, die von einem Kommunikationsknoten wegführen, mit 'EXOR', so erhält man den Kommunikationsausdruck der gesendet bzw. empfangen werden soll. Die Reihenfolge, in der die Kommunikationsterme zusammengefügt werden sollen, wird in dem zugehörigen Kommunikationsknoten vermerkt (Prioritätenrelation).
Soll zu einem direkten Kommunikationsterm noch eine Folge von Prozeßzeigervariablen angegeben werden, so werden deren Namen

nikationsausdrucks zeitüberwacht, so führt von einem Kommunikationsknoten genau eine Sende- bzw. Empfangskante mit der Beschriftung 'Zeit' weg. Die entsprechende Zeitdauer enthält die Knotenbeschriftung (siehe unten).
Graphisch wird die Sendekante als Doppellinie und die Empfangskante als einfache Linie dargestellt (siehe Bild 5.15).

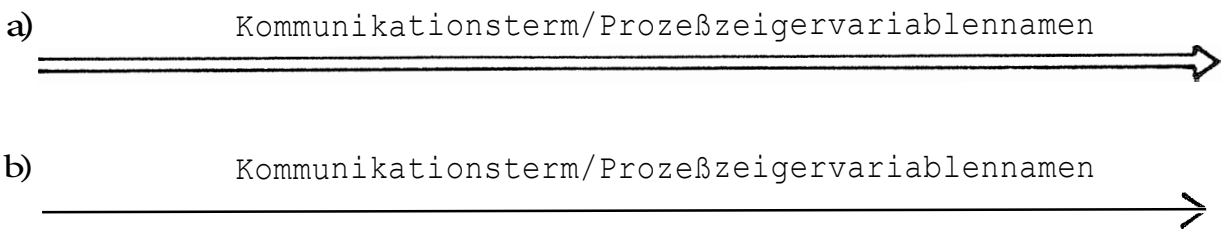


Bild 5.15 : a) Sendekante
b) Empfangskante

Wie bereits angemerkt wurde, wird der Kommunikationsknoten mit der Prioritätenrelation (Reihenfolge, in der die Kommunikationsterme zusammengefügt werden) und dem Zeitintervall für die Zeitüberwachung beschriftet. Neben diesen Angaben kann jeder Kommunikationsknoten mit einem für die betrachtete Ablaufsteuerung eindeutigen Namen (Steuerzustandsname) versehen werden.
Für diese Angaben sind in der graphischen Darstellung der Kommunikationsknoten drei Felder vorgesehen (siehe Bild 5.16).

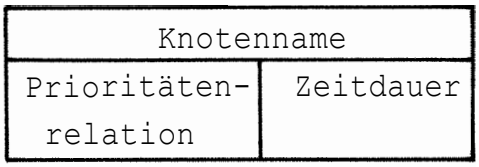


Bild 5.16 : Graphische Darstellung der Kommunikationsknoten

Ist in einem Kommunikationszustand keine Prioritätenrelation eingetragen (das entsprechende Feld ist leer), so werden die Kommunikationsterme in willkürlicher Reihenfolge zum entsprechenden Kommunikationsausdruck zusammengefügt.
Ist in einem Kommunikationsknoten keine Zeitdauer angegeben, so wird ein Zeitintervall der Länge 'unendlich' angenommen. In

diesem Fall braucht von einem Kommunikationsknoten keine Kante mit der Beschriftung 'Zeit' wegführen.
Erreicht die Ablaufsteuerung einen Internknoten, so wird die Kommunikationsmaschine (exklusiv) entweder mit dem Ausführen einer internen Funktion oder internen Operation beauftragt. Jeder Internknoten ist mit dem Namen der in diesem Zustand auszuführenden internen Operation bzw. Funktion beschriftet.
Von einen Internknoten können (exklusiv) entweder Funktionskanten oder Operationskanten wegführen. Führen von einem Internknoten Funktionskanten weg, dann wird in diesem Steuerzustand eine interne Funktion ausgeführt. Die Funktionskanten sind dann mit den möglichen Ergebnissen der auszuführenden internen Funktion beschriftet. Dabei können an einer Kante mehrere mögliche Ergebnisse stehen.
Führen von einem Internzustand Operationskanten weg, so wird in diesem Zustand eine interne Operation aufgerufen. Die wegführenden Operationskanten sind dann mit den möglichen Abschlüssen der internen Operation beschriftet.
Graphisch wird eine Operationskante als Doppellinie und eine Funktionskante als einfache Linie dargestellt (siehe Bild 5.17).

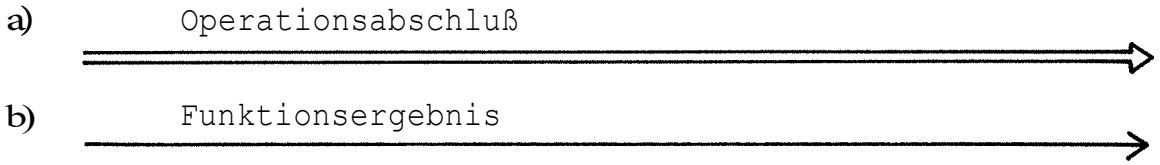


Bild 5.17: a) Operationskante
b) Funktionskante

Neben dem Namen der in einem Internzustand auszuführenden internen Funktion bzw. Operation kann jeder Internknoten mit einem für die betrachtete Ablaufsteuerung eindeutigen Namen beschriftet (siehe Bild 5.18) werden.

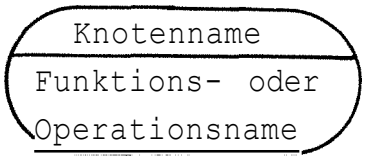


Bild 5.18 : Graphische Darstellung der Internknoten

Führt von einem Kommunikationsknoten eine Doppellinie weg, so handelt es sich um eine Sendekante; wenn sie an einem Internknoten beginnt, handelt es sich um eine Operationskante. Ähnlich ist es bei den einfachen Linien: Beginnen sie an einem Kommunikationsknoten, sind es Empfangskanten ansonsten Funktionskanten.
In jeder Ablaufsteuerung gibt es einen Anfangssteuerzustand. Dieser Steuerzustand ist mit dem Anfangssymbol (Bild 5.19) gekennzeichnet. Der Anfangssteuerzustand kann sowohl ein Kommunikationszustand als auch ein Internzustand sein.

EjAnfangssymbol

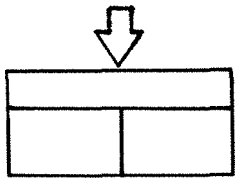


Bild 5.19: Kennzeichnung des Anfangssteuerzustands

Die Verwendung der verschiedenen Linienarten soll an einigen Beispielen erläutert werden.
Bild 5.20a zeigt einen Kommunikationszustand, in dem die Nachricht a an den Prozeß p1 gesendet werden soll. Es handelt sich hier also um einen sehr einfachen Kommunikationsausdruck, der gesendet werden soll.
Bild 5.20b zeigt einen Kommunikationszustand, in dem der Kommunikationsausdruck (p1)a EXOR (p2)a empfangen werden soll.

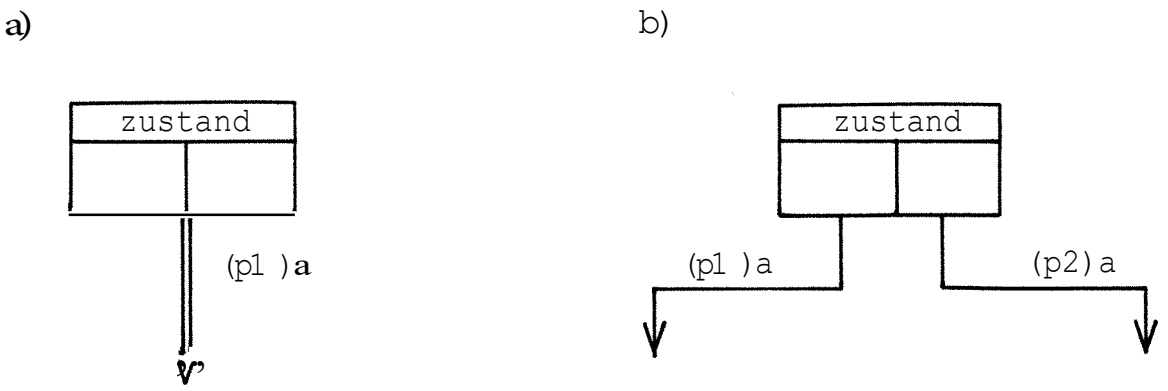


Bild 5.20 : a) Kommunikationszustand, in dem der Kommunikationsausdruck (p1)a gesendet werden soll.
b) Kommunikationszustand, in dem der Kommunikationsausdruck (p1) EXOR (p2)a empfangen werden soll

Bild 5.21a zeigt einen Internknoten, in dem eine interne Operation aufgerufen wird (es führt eine Doppellinie weg). Bild 5.21b zeigt einen Internknoten, in dem eine interne Funktion aufgerufen wird.

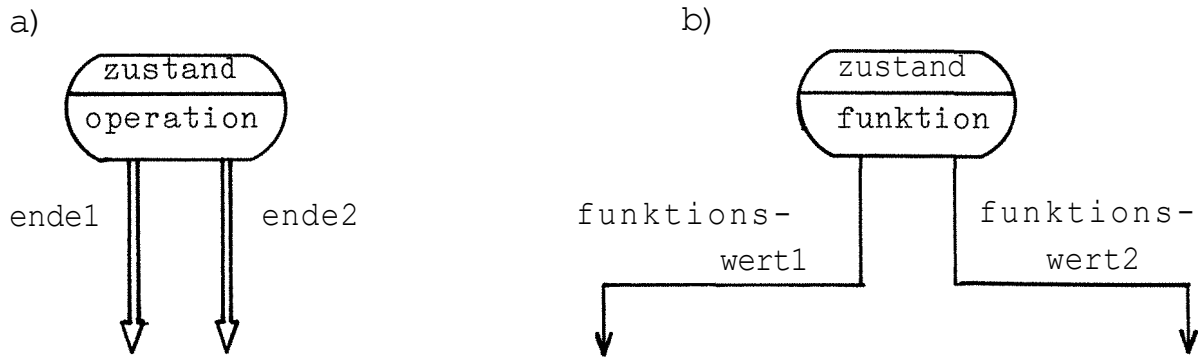


Bild 5.21 : a) Internzustand, in dem eine interne Operation ausgeführt wird
b) Internzustand, in dem eine interne Funktion ausgeführt wird

Um die Kontextabhängigkeit für die Bedeutung der einfachen oder der doppelten Kanten zu vermeiden, können für diese auch andere Strichtypen verwendet werden (z.B. gestrichelt).

Bild 5.22 zeigt ein Beispiel für eine Ablaufsteuerung.

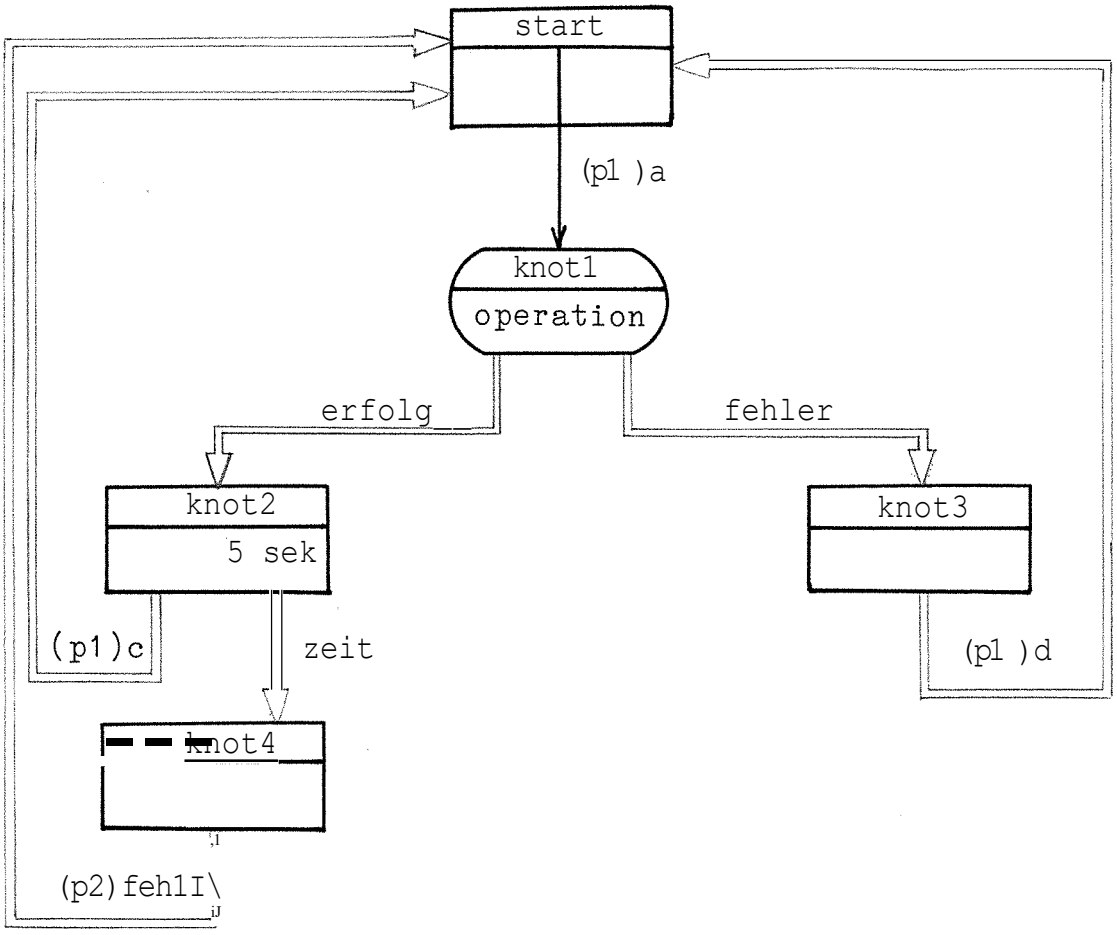


Bild 5.22 : Beispiel für eine Ablaufsteuerung

5.2.3.3.2 Die Dynamik der Ablaufsteuerung

Wird ein Prozeß gestartet, wird zunächst die Benutzermaschine initialisiert und der Prozeß befindet sich im Anfangssteuerzustand.
In einem Kommunikationszustand will ein Prozeß entweder einen Kommunikationsausdruck senden oder empfangen.
Die Beschriftung des Kommunikationsknoten (mit Ausnahme des Knotennamens) und der Sende- bzw. Empfangskanten ergibt die Parameter für den Aufruf der entsprechenden Funktion der Kommunikationsmaschine. Je nachdem, welcher Kommunikationsterm

nun gesendet oder empfangen wurde, geht die Ablaufsteuerung in den entsprechenden Steuerzustand über.
Läuft beim Senden oder Empfangen eines Kommunikationsausdrucks die Zeitüberwachung ab, so geht die Ablaufsteuerung zu dem Steuerzustand über, zu dem die Kante mit der Beschriftung 'Zeit' führt.
Gelangt eine Ablaufsteuerung zu einem Internzustand, so wird die Kommunikationsmaschine mit der Ausführung der entsprechenden internen Operation oder Funktion beauftragt. Je nachdem, mit welchem Ergebnis die interne Funktion bzw. Operation abgeschlossen wurde, geht die Ablaufsteuerung in den entsprechenden Folgezustand über. Die Ablaufsteuerung aus Bild 5.22 hat also folgendes dynamisches Verhalten.
Im Anfangszustand 'start' wird vom Prozeß p1 die Nachricht a erwartet. Trifft diese ein, wird in den Steuerzustand 'knot1' übergegangen. Dort wird die interne Operation 'operation' aufgerufen. Kann diese Operation ordnungsgemäß zu Ende gebracht werden, wird in den Steuerzustand 'knot2' übergegangen, ansonsten in den Zustand 'knot3'• Im Kommunikationszustand 'knot2' wird an den Prozeß p1 die Nachricht c gesendet. Das Senden dieses Kommunikationselements wird zeitüberwacht (5 Sekunden). Kann das Kommunikationselement innerhalb dieser Zeit übertragen werden, so geht die Ablaufsteuerung in den Steuerzustand 'start' über. Läuft die Zeitüberwachung ab, wird in den Zustand 'knot4' übergegangen. In diesem Zustand wird an den Prozeß p2 die Nachricht 'fehl' gesendet.
Im Zustand 'knot3' wird an den Prozeß p1 die Nachricht d gesendet und dann in den Zustand 'start' übergegangen.

5.2.3.4 Einzelprozeß-, Prozeßbündel- und Prozeßgruppentypen

Es kann vorkommen, daß in einem Prozeßsystem mehrere Einzelprozesse, Prozeßbündel oder Prozeßgruppen nötig sind, die auf identischen Benutzermaschinen laufen, deren Wartebereiche gleich groß sind und deren Ablaufsteuerungen bis auf die Adreß- und Absenderangaben ebenfalls übereinstimmen. Die Einzelprozesse, Prozeßbündel bzw. Prozeßgruppen eines Typs unter-

scheiden sich nur durch ihre Kommunikationspartner.
Im Spezifikationskonzept sind deshalb sogenannte Einzelprozeß-, Prozeßbündel- und Prozeßgruppentypen vorgesehen. Diese Typen sind vergleichbar mit den in Abschnitt 3.2.2.2 kurz erwähnten Type-Generatoren /ADA79/. Bei Einzelprozeß-, Prozeßbündel- und Prozeßgruppentypen werden in der Ablaufsteuerung, bei der Adreß- und Absenderangabe nicht die Namen von existierenden Prozessen und Prozeßgruppen verwendet, sondern sogenannte formale Namen. Ein Exemplar dieses Typs wird, ähnlich wie bei den abstrakten Datentypen, durch eine Deklarationsanweisung erzeugt. Durch eine Deklarationsanweisung erhält die erzeugte Struktureinheit einen eindeutigen Namen. Bei der Deklarationsanweisung wird außerdem festgelegt, welche formalen Namen auf welche konkreten Einzelprozeß-, Prozeßbündel- bzw. Prozeßgruppennamen abgebildet werden. Natürlich muß jedem formalen Namen ein konkreter Name zugeordnet werden.

5.2.3.5 Der Programmentwurf

Beim Programmentwurf wird das zu erstellende Automatisierungsprogramm in einzelne Strukturierungseinheiten zerlegt. Wie bereits mehrfach erwähnt, kann eine solche Strukturierungseinheit
- ein Einzelprozeß,
- ein Prozeßbündel oder
- eine Prozeßgruppe sein.

Der technische Prozeß wird in einzelne Komponenten aufgeteilt. Bei dieser Aufteilung muß berücksichtigt werden, welche Komponenten des technischen Prozesses mit Rechenprozessen des Automatisierungsprogramms kommunizieren. Dies ist wichtig, damit der Zusammenhang zwischen dem technischen Prozeß und dem Automatisierungsprogramm durchschaubar wird.
Ähnlich wird mit den Geräten zur Kommunikation mit dem Benutzer verfahren. Die einzelnen Ein-/Ausgabegeräte werden in die Betrachtung der Kommunikation mit einbezogen.

Durch die Kommunikationsstruktur wird die Beziehung der einzelnen Strukturierungseinheiten untereinander, zu den *Komponenten* des technischen Prozesses und zum Benutzer (E/A-Geräte) beschrieben. Wie bereits in Abschnitt 5.2. geschildert, zeigt die Kommunikationsstruktur, welche Nachrichten an wen gesendet (Strukturierungseinheit, Komponente des technischen Prozesses, E/A-Gerät) bzw. welche Nachrichten von wem empfangen werden.
Zeigt es sich bei der Zerlegung des Automatisierungsprogramms, daß einige Strukturierungseinheiten identisch aufgebaut sind, so können Typen von Strukturierungseinheiten definiert werden. Beim Entwurf von Typen von Strukturierungseinheiten und von konkreten Strukturierungseinheiten wird analog vorgegangen. Diese Vorgehensweise soll im folgenden kurz erläutert werden.
Es erscheint sinnvoll, bei den einzelnen Strukturierungseinheiten zunächst die Ablaufsteuerung bzw. Ablaufsteuerungen zu entwerfen und dann erst die Benutzermaschine zu definieren. Durch die Ablaufsteuerung ist im wesentlichen festgelegt, über welche Operationen und Funktionen die private bzw. gemeinsame Benutzermaschinen verfügen muß.
Es bleibt allerdings offen, wie beim Entwurf der gemeinsamen oder privaten Benutzermaschine vorgegangen wird. Hier kann die Vorgehensweise von der verwendeten Spezifikationstechnik abhängen.

Aus der vorgeschlagenen Vorgehensweise für den Entwurf ergibt sich ein Gerüst für die Gliederung von Spezifikationen.
Eine Spezifikation gliedert sich demnach in drei Hauptteile:

I. Kommunikationsstruktur
II. Typen von Strukturierungseinheiten
III. Deklaration und Beschreibung von Strukturierungseinheiten

Der erste Hauptpunkt enthält die Kommunikationsstruktur.
Im Hauptpunkt zwei werden die einzelnen Typen von Strukturierungseinheiten beschrieben. Die Beschreibung eines Typs wird, je nachdem, ob es sich um einen Einzelprozeß, ein Prozeßbündel oder eine Prozeßgruppe handelt, verschieden untergliedert. Die Details dieser Untergliederung finden sich im Anhang.

Der dritte Hauptpunkt zerfällt in zwei Teile:
A Deklaration von Strukturierungseinheiten
B Beschreibung von konkreten Strukturierungseinheiten

Wie eine Deklaration im Teil A aussehen kann, ist dem Anhang zu entnehmen. Ebenso kann die Gliederung der Beschreibung von Strukturierungseinheiten dem Anhang entnommen werden. Die Beschreibungen von Einzelprozessen, Prozeßbündeln und Prozeßgruppen unterscheiden sich natürlich.

Erstrebenswertes Entwurfsziel ist eine einfache Kommunikationsstruktur, d.h. die Anzahl der Strukturierungseinheiten soll klein sein, jeder Prozeß bzw. jede Prozeßgruppe soll wenige Kommunikationspartner haben und wenige verschiedene Nachrichten empfangen und senden. Diese Entwurfsziele widersprechen sich teilweise (z.B. möglichst wenige Prozesse, die möglichst wenig unterschiedliche Nachrichten empfangen bzw. senden). Hier einen Kompromiß zu finden, bleibt dem Geschick und der Erfahrung des Entwerfers überlassen.

5-3 Prüfungsmöglichkeiten für Spezifikationen

Mit dem bisher vorgestellten Verfahren ist es möglich, die Spezifikation eines Automatisierungsprogramms auf folgende Kriterien hin zu überprüfen:
- Vollständigkeit und
- bestimmte Arten von Verklemmungen.

5.3.1 Vollständigkeit der Spezifikation

Vollständigkeit der Spezifikation heißt:
- Vollständigkeit der privaten und gemeinsamen Benutzermaschine
- Vollständigkeit der Ablaufsteuerung
- Abgeschlossenheit der Kommunikation und
- Übereinstimmung von Anzahl und Typ von Nachrichtenparametern

Vollständigkeit der privaten und gemeinsamen Benutzermaschine heißt,
- daß zu jeder Nachricht, die empfangen oder gesendet wird, die entsprechende Eingabe- bzw. Ausgabefunktion definiert sein muß, d.h. es wird geprüft, ob zu allen Nachrichten, die in der Ablaufsteuerung auftauchen, eine entsprechende Funktion bzw. Operation der privaten Benutzermaschine angegeben ist.
- daß alle internen Funktionen und Operationen, die in der Ablaufsteuerung aufgerufen werden, in der privaten oder gemeinsamen Benutzermaschine definiert sind.

Die Vollständigkeit der Benutzermaschine kann statisch geprüft werden.

Die Vollständigkeit der Ablaufsteuerung bezieht sich vor allem auf die Internknoten einer Ablaufsteuerung.
Wird in einem Internknoten eine interne Funktion aufgerufen, muß gewährleistet sein, daß durch die Beschriftung der Funktionskanten alle möglichen Ergebnisse berücksichtigt werden, denn es ist nicht definiert, wie die Ablaufsteuerung sich

verhält, wenn sie ein Ergebnis erhält, für das kein Übergang vorgesehen ist.
Ähnliche Forderungen gelten bei internen Operationen. Wird in einem Internknoten eine interne Operation aufgerufen, muß für jedes mögliche Ende der Operationsausführung ein Übergang vorgesehen sein.

Abgeschlossenheit des Nachrichtenaustausches heißt,
- daß jede Nachricht, die an einen bestimmten Prozeß gesendet wird, von diesem auch in irgendeinem Kommunikationszustand erwartet wird.
- daß jede Nachricht, die von einem Prozeß erwartet wird, von einem entsprechenden Prozeß auch gesendet wird.

Die Abgeschlossenheit des Nachrichtenaustausches kann statisch geprüft werden.

Statisch können auch die Anzahl und die Typen der Nachrichtenparameter verglichen werden, d.h. die Anzahl und die Typen der Parameter einer Nachricht müssen beim Quell- und Zielprozeß übereinstimmen (wie bei den Parametern von Prozeduren).

5.3.2 Erkennen bestimmter Arten von Verklemmungen

Aus der Semantik der Kommunikationsmaschine folgt, daß
- alle Prozesse beim Senden,
- beim Empfangen und
- Prozesse in Prozeßbündeln und Prozeßgruppen auch beim Aufruf von internen Operationen und internen Funktionen

blockiert werden können.

Beim Senden kann die Blockadeursache sein, daß kein Kommunikationsterm übertragen werden kann, da die entsprechenden Wartebereiche voll sind bzw. die Zielprozesse die entsprechenden Nachrichten nicht annehmen (Kommunikationsblockade).

Beim Empfangen gibt es zwei Blockadeursachen:
- Ein Prozeß wird blockiert, wenn keiner der erwarteten Kommu-

nikationsterme zur Verfügung steht (Kommunikationsblockade).
- Ein Prozeß wird blockiert, obwohl ein erwarteter Kommunikationsterm zur Verfügung steht, aber nicht alle zugehörigen Eingabeoperationen können ausgeführt werden (Datenblockade).

Bei Prozeßbündeln oder bei Prozeßgruppen können Prozesse zusätzlich beim Aufruf von Funktionen und Operationen der gemeinsamen Benutzermaschine blockiert werden (Benutzermaschinenblockade).
Alle geschilderten Blockademöglichkeiten können dazu führen, daß sich Prozesse verklemmen. Verklemmen heißt, daß mehrere blockierte Prozesse eines Prozeßsystems nie mehr deblockiert werden. Es werden also folgende drei Arten von Verklemmungen unterschieden:
- Verklemmungen in der gemeinsamen Benutzermaschine (Benutzermaschinenverklemmung)
- Verklemmungen wegen nicht ausführbarer Eingabeoperationen (Datenverklemmung)
- Verklemmungen durch das Kommunikationsverhalten (Kommunikationsverklemmung)

Im folgenden werden die einzelnen Möglichkeiten zum Finden der verschiedenen Arten von Verklemmungen diskutiert.
Um Benutzermaschinenverklemmungen und Datenverklemmungen zu finden, können allerdings die Möglichkeiten nur skizziert werden. Es ist nicht gelungen, hier vollständige Lösungswege aufzuzeigen. Dies war auch nicht die Intention dieser Arbeit, in deren Mittelpunkt Kommunikationsaspekte stehen. Allerdings kann für die Suche nach Kommunikationsverklemmungen ein Verfahren angegeben werden.

5.3.2.1 Verklemmungen durch Benutzermaschinenblockaden

Diese Art der Verklemmung kann nur bei Prozeßbündeln und Prozeßgruppen auftreten.
Bei der Untersuchung von Verklemmungen dieses Typs wird davon ausgegangen, daß sich die Kommunikationspartner von

Prozeßbündeln und Prozeßgruppen beliebig verhalten. Benutzermaschinenblockaden können nur beim Aufruf von internen Funktionen und Operationen auftreten, wobei durch die Ablaufsteuerung die möglichen Aufrufreihenfolgen der internen Operationen und Funktionen beschrieben sind.
Die gemeinsamen Objekte, die die gemeinsame Benutzermaschine bilden, können als gemeinsame Betriebsmittel /HOFM78/ aufgefaßt werden. Eine Bibliographie über Techniken zum Finden oder Vermeiden von Verklemmungen bei der Verwendung gemeinsamer Betriebsmittel findet sich in /NEWT79/.

5.3.2.2 Verklemmungen durch Datenblockaden

In diesem Abschnitt soll eine Vorgehensweise diskutiert werden, die es gestattet, Datenverklemmungen zu finden. Dabei gibt es allerdings noch Probleme, die noch nicht befriedigend gelöst werden konnten.
Eine Datenblockade tritt dann auf, wenn die Eingabeoperation einer erwarteten und zur Verfügung stehenden Nachricht nicht ausgeführt werden kann ('Restriction'- Teil in einer algebraischen Spezifikation, Vorbedingung bei der Prädikatentransformation). Die Untersuchung auf Datenverklemmungen (Verklemmungen allein verursacht durch Datenblockade) wird unabhängig davon durchgeführt, in welcher Reihenfolge Kommunikationselemente eintreffen, d.h. die Struktur der Ablaufsteuerung bleibt unberücksichtigt. Unwesentlich ist auch, ob es sich bei den zu untersuchenden Prozessen um Einzelprozesse oder um Prozesse in Prozeßbündeln oder Prozeßgruppen handelt, denn die gemeinsame Benutzermaschine hat bei den Datenblockaden keine Bedeutung. Die Eingabeoperationen sind nur auf der jeweiligen privaten Benutzermaschine definiert.
In einer Datenblockade kann sich nur ein Prozeß befinden, der empfangen möchte. Allerdings kann diese Datenblockade Rückwirkungen auf sendende Prozesse haben. Denn wenn ein auf Empfang stehender Prozeß ohne Wartebereich einen erwarteten Kommunikationsterm nicht annehmen kann, da mindestens eine der entsprechenden Eingabeoperationen nicht ausgeführt werden kann,

so können die Quellprozesse die Nachrichten nicht übertragen.
Ist einem Prozeß ein Wartebereich vorgelagert, so bezieht sich die Datenblockade nur auf Kommunikationselemente, die sich im Wartebereich befinden. Sendende Prozesse sind dann nicht direkt betroffen, da diese die Kommunikationselemente im Wartebereich ablegen können, ohne Rücksicht darauf, ob deren Eingabeoperationen beim Zielprozeß ausgeführt werden können. Allerdings kann dadurch der Wartebereich mit Kommunikationselementen gefüllt werden, die der Prozeß erwartet, die er aber nicht annehmen kann, da die zugehörigen Eingabeoperationen nicht ausgeführt werden können.
Ist der Wartebereich durch solche Kommunikationslernente vollständig gefüllt, so können sendende Prozesse ihre Nachrichten nicht mehr im Wartebereich ablegen und werden blockiert.
Die Datenblockade eines empfangenden Prozesses wirkt sich somit indirekt auf sendende Prozesse aus.
Bei der Suche nach Verklemmungen durch Datenblockaden wird wegen obiger Argumente bei Prozessen mit bzw. ohne Wartebereich gleich vorgegangen. Da der Fall, daß einem Prozeß kein Wartebereich vorgelagert ist, bezüglich Verklemmungen der ungünstigere ist, wird bei der Suche nach Datenverklemmungen bei allen Prozessen und Prozeßgruppen kein Wartebereich angenommen. Datenblockaden können nur beim Empfangen auftreten, können aber, wie oben bereits gezeigt, Blockaden bei sendenden Prozessen verursachen. Deshalb müssen bei allen Prozessen alle Kommunikationszustände untersucht werden.

Zunächst sollen diejenigen Kommunikationszustände betrachtet werden, in denen das Empfangen nicht zeitlich überwacht wird.
Bei diesen Kommunikationszuständen wird die Beschriftung der Empfangskanten untersucht, ob die Kommunikationsterme Nachrichten enthalten, deren Eingabeoperationen nicht immer ausführbar sind. Enthält ein Kommunikationsterm mindestens eine solche Nachricht, könnte es in dem betrachteten Kommunikationszustand zu einer Datenblockade kommen. Solche Kommunikationszustände werden als datenblockadegefährdete Zustände bezeichnet. Da angenommen wird, es würde keinem Prozeß bzw. keiner Prozeßgruppe ein Wartebereich vorgelagert sein, folgt,

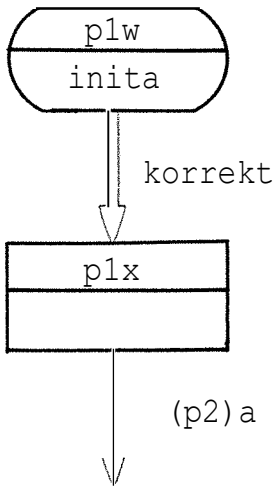
daß die entsprechenden Quellprozesse ebenfalls blockiert werden.
Ob es zu einer Datenblockade kommt, hängt vom Zustand der privaten Benutzermaschine des betrachteten empfangenden Prozesses (Datenzustand), von den Werten der Nachrichtenparameter und damit vom Datenzustand des Absenders einer Nachricht ab.
Für jeden dieser, für eine Datenblockade gefährdeten Zustände, muß nun untersucht werden,

- welche Datenzustände möglich sind, wenn beim Empfänger der betrachtete Kommunikationszustand erreicht wird,
- welche Wertebereiche die einzelnen Parameter annehmen können, d.h. bei den Quellprozessen muß nachgesehen werden, in welchen Kommunikationszuständen sie sein können, wenn sie die entsprechenden Nachrichten absenden und welche Wertebereiche dabei für die Nachrichtenparameter (Datenzustände) möglich sind.
  Bei Nachrichten ohne Nachrichtenparameter entfällt diese Untersuchung.

Das Finden aller möglichen Datenzustände, wenn sich derbetrachtete Prozeß in einem bestimmten Kommunikationszustand befindet, dürfte im allgemeinen ein schwieriges Unterfangen sein, da die Vorgeschichte eines Prozesses berücksichtigt werden muß, d.h. welche Zweige der Ablaufsteuerung ein Prozeß wie oft durchlaufen hat. Zur Lösung dürften sich hier Konstrukte wie die Schleifeninvarianten, die bei der Programmverifikation üblich sind, anbieten.
Einen Eindruck davon, wie die Suche nach Datenverklemmungen vonstatten gehen kann, soll das nun folgende sehr einfache Beispiel vermitteln.

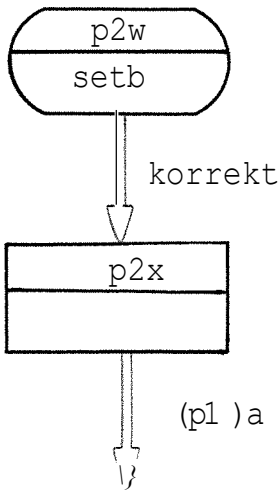
In dem in Bild 5.23 dargestellten Beispiel soll der Zustand p1x des Prozesses p1 auf Datenblockaden bzw. -verklemmungen untersucht werden.

Prozeß: p1




```
/*Ausschnitt aus der Benut-
  zermaschine von Prozeß P1*/
3. BENUTZERMASCHINE
3.1. AUSGABEFUNKTIONEN: .........
3.2. EINGABEOPERATIONEN: ..,a, ...
3.3. INTERNE OPERATIONEN:inita,.
3.4. INTERNE FUNKTIONEN: .........
3.5. BENUTZERMASCHINE:
          .
     var: TYP integer
          .
      inita:
      {TRUE} inita {var=0}
          .
     a: (par)
      {var+par.(6} a {var= 'var+par}
          .
          .
```

Prozeß: p2




```
/*Ausschnitt aus der Benut-
  zermaschine von Prozeß P2*/
3. BENUTZERMASCHINE
3.1. AUSGABEFUNKTIONEN: ..,a,.
3.2. EINGABEOPERATIONEN: .....
3.3. INTERNE OPERATIONEN:setb
3.4. INTERNE FUNKTIONEN: .... .
3.5. BENUTZERMASCHINE:
          .
      b: TYP integer
          .
       setb:
       {TRUE} setb {b=1}
          .
     a: (par)
      {TRUE} a {par=b }
          .
          .
```

Bild 5,23 : Beispiel für eine Untersuchung auf Datenverklemmungen

Im Zustand p1x erwartet der Prozeß p1 vom Prozeß p2 die Nachricht a. Die dieser Nachricht a im Prozeß p1 zugordnete Einga-

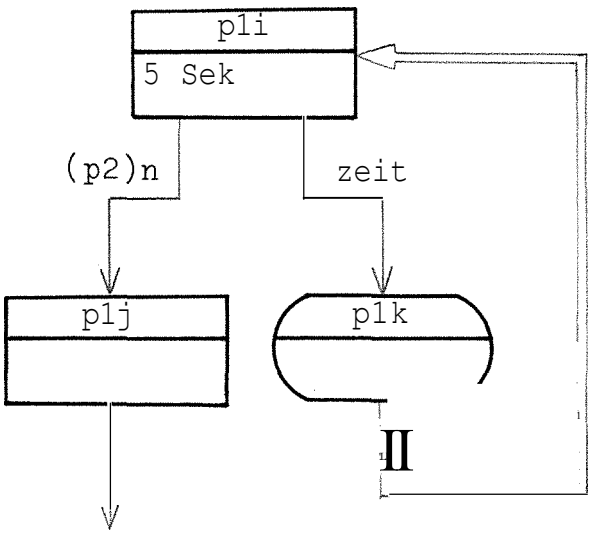
beoperation kann nur ausgeführt werden, wenn die Benutzermaschine in einem Zustand ist, in dem der Wert der Integervariablen var kleiner 6 bleibt, wenn zum momentanen Wert der Variablen var der Wert des Nachrichtenparameters par dazugezählt wird. Daraus folgt, daß der Prozeß P1 im Zustand p1x in eine Datenblockade geraten kann.
Als einzigen Vorgängerzustand hat p1x den Zustand p1w. In diesem Steuerzustand wird die Variable var auf null gesetzt. Erreicht also der betrachtete Prozeß p1 den Steuerzustand p1x, so hat var immer den Wert Null und die Nachricht a kann angenommen werden, wenn der Nachrichtenparameter par einen Wert kleiner gleich sechs hat. Nun müssen im Prozeß p2 alle Kommunikationszustände betrachtet werden, in denen p2 an p1 die Nachricht a schickt.
Der einzige solche Kommunikationszustand soll der Zustand p2x sein. Dessen einziger Vorgängerzustand ist der Steuerzustand p2w. In diesem Zustand wird durch die interne Operation setb die Variable b auf den Wert eins gesetzt.
Die Nachricht a enthält einen Parameter. Die Ausgabefunktion der Nachricht a setzt den Wert dieses Parameters gleich dem Wert der Variablen b, d.h. da die Variable b immer den Wert eins hat, wenn sich der Prozeß p2 im Zustand p2x befindet, hat auch der Nachrichtenparameter par den Wert eins.
Da hier vorausgesetzt wird, daß der Prozeß p2 in den Steuerzustand p2x kommt, kann der Prozeß p1 die Eingabeoperation von a ausführen. Es kann im Zustand p1x des Prozesses p1 zu keiner Datenverklemmung kommen. Die Frage, ob der Prozeß p2 die Nachricht a sendet, wird hier nicht behandelt (dazu siehe den nächsten Abschnitt).

Bisher wurden nur diejenigen Kommunikationszustände untersucht, bei denen das Empfangen nicht zeitüberwacht wurde.
Allerdings müssen auch die Kommunikationszustände, in denen das Empfangen zeitüberwacht wird, geprüft werden. Hier wird zwar der empfangende Prozeß nur für eine maximale Zeitspanne blockiert, aber die entsprechenden Sendeprozesse können sich verklemmen.
Der empfangende Prozeß führt immer den Übergang 'zeit' (siehe

Abschnitt 5.2.3) aus, da durch den Datenzustand die entsprechenden Eingabeoperationen von angebotenen Nachrichten nicht ausgeführt werden können. Da davon ausgegangen wird, daß keinem Prozeß ein Wartebereich vorgelagert ist, bleiben diesendenden Prozesse u.U für immer blockiert. Aus diesen Überlegungen folgt nun, daß die Kommunikationszustände, in denen ein Prozeß zeitüberwacht Kommunikationsausdrücke empfängt, genauso geprüft werden, wie die Kommunikationszustände, in denen nicht zeitüberwacht empfangen wird.

Da die Voraussetzungen bei der Suche nach Datenverklemmungen sehr streng sind, kann es durchaus sein, daß bei den Untersuchungen, die mit diesen Voraussetzungen angestellt werden, Verklemmungen gefunden werden, die in Wirklichkeit nicht auftreten. Durch ein entsprechendes Kommunikationsverhalten kann eine Datenverklemmung überhaupt nicht zustande kommen. Es werden aber alle Datenverklemmungen, bei beliebigen Kommunikationsverhalten, gefunden.
Für die Praxis heißt dies, daß die mit obigen Voraussetzungen gefundenen Datenverklemmungen nochmals eigens betrachtet werden müssen, um 'scheinbare' Verklemmungen auszusondern. Allerdings dürfte es schwer sein, dafür Regeln anzugeben, da hier zahlreiche Umstände berücksichtigt werden müssen. In diesem Fall spielt ebenfalls die Erfahrung des Entwerfers eine wesentliche Rolle.
Im folgenden Beispiel soll gezeigt werden, daß eine mit den obigen Voraussetzungen mögliche Verklemmung in Wirklichkeit nicht auftreten muß, da durch das spezielle Kommunikationsverhalten die Datenverklemmung nicht zustande kommen kann. Wird der Übergang mit der Beschriftung 'zeit' nicht mitberücksichtigt, wird mit obigen Voraussetzungen eine Datenverklemmung festgestellt. Wird jedoch die Zeitüberwachung in die Betrachtung mit einbezogen, kann die vermeintliche Verklemmung nicht auftreten.

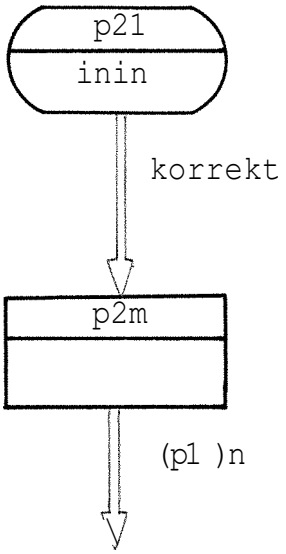
Prozeß: p1




```
/*Ausschnitt aus der Benutzer-
  maschine von Prozeß P1     */
3. BENUTZERMASCHINE
3.1. AUSGABEFUNKTIONEN: .........
3.2. EINGABEOPERATIONEN: .. ,n, ...
3.3. INTERNE OPERATIONEN:setn, ..
3.4. INTERNE FUNKTIONEN: ........
3.5. BENUTZERMASCHINE:
     .
var TYP integer
     .
setn:
  TRUE   setn   var=1

n: (par)
  var+par 10   n   var='var+par
     .
     .
```

Prozeß: p2




```
/*Ausschnitt aus der Benut-
  zermaschine von Prozeß P2*/
 3. BENUTZERMASCHINE
3.1. AUSGABEFUNKTIONEN: .. ,n,.
3.2. EINGABEOPERATIONEN: .....
3.3. INTERNE OPERATIONEN:inin
3.4. INTERNE FUNKTIONEN: .... .
3.5. BENUTZERMASCHINE:
      .
 nn: TYP integer
      .
 inin:
   TRUE   inin   nn=2

 n: (par)
   TRUE   n   par=var
      .
      .
```

Bild 5.24: Beispiel für eine 'scheinbare' Verklemmung

Erreicht der Prozeß p1 den Kommunikationszustand p1i, so wird vom Prozeß p2 die Nachricht n erwartet. Das Empfangen dieser Nachricht wird mit einer Zeitdauer von fünf Sekunden zeitüber-

wacht. Es wird nun festgestellt, daß im Zustand p1i die zu n gehörige Eingabeoperation nicht immer ausgeführt werden kann, d.h. es kommt zu einer Datenblockade, wenn der Prozeß p2 die Nachricht n anbietet. Der Zustand p21 soll der einzige Kommunikationszustand sein, in dem der Prozeß p2 die Nachricht n an den Prozeß p1 senden möchte.
Bei einer Untersuchung dieses Sachverhalts wird unter den obigen Voraussetzungen festgestellt, daß es zu einer Verklemmung kommt. Der Prozeß p1 kann die Eingabeoperation der Nachricht n nicht ausführen und p2 kann die Nachricht deshalb nicht senden. Die Prozesse p1 und p2 sind verklemmt. Allerdings ist nun leicht zu sehen, daß es wegen der vorher nicht mit berücksichtigten Zeitüberwachung zu keiner Verklemmung kommen kann, da nach dem Ablauf der Zeitüberwachung die Variable var in einen Zustand versetzt wird, in dem die Eingabeoperation von n in jedem Fall ausgeführt werden kann.

5.3.2.3 Verklemmungen durch Kommunikationsblockaden

Die Ursachen für Verklemmungen durch das Kommunikationsverhalten (Kommunikationsverklemmung) sind das Senden und Empfangen von Kommunikationsausdrücken. Dabei werden mögliche Datenblokkaden oder Benutzermaschinenblockaden nicht mitberücksichtigt. Kommt es beim Senden zu einer Blockade, so können die Ursachen dafür sein:
- Einer oder mehrere Wartebereiche bei den Adressaten sind voll.
- Die Kommunikationselemente werden von den Adressaten momentan nicht angenommen.

Ähnliches gilt bei Empfangsblockaden:
- Keiner der erwarteten Kommunikationsterme befindet sich im Wartebereich.
- Keiner der erwarteten Kommunikationsterme wird angeboten.

Bei der Suche nach Verklemmungen, die nur durch das Kommunikationsverhalten bedingt sind, müssen die Struktur der Ablaufsteuerungen, die Wartebereiche und die Prozeßzeigervariablen

betrachtet werden.
Ein Prozeß Pi (Ein Einzelprozeß oder ein Prozeß innerhalb eines Prozeßbündels oder einer Prozeßgruppe) befindet sich zu jedem Zeitpunkt t in einem Steuerzustand sf (die Übergänge sollen zeitlos sein).
Der Wartebereich eines Prozesses (oder Prozeßgruppe) Pi ist zu jedem Zeitpunkt entweder leer oder mit Kommunikationselementen belegt. Die Belegung des Wartebereichs wird Wartebereichszustand genannt. Der Wartebereich eines Prozesses Pi befindet sich zu einem Zeitpunkt t in einem Zustand wi.
Die Prozeßzeigervariablen eines Prozesses Pi sind zu einem Zeitpunkt t mit entsprechenden Werten belegt. Eine Prozeßzeigervariable kann leer sein (d.h. sie enthält keinen Prozeßnamen) oder den Namen eines Prozesses bzw. einer Prozeßgruppe enthalten. Die Belegung der Prozeßzeigervarablen des Prozesses Pi zum Zeitpunkt t wird mit Prozeßzeigerzustand z{ bezeichnet.

Mit Si wird die Menge aller Steuerzustände, mit Wi die Menge aller Wartebereichszustände und mit Zi die Menge aller Prozeßzeigerzustände des Prozesses Pi bezeichnet.

In einem Automatisierungsprogramm gibt es n Prozesse. Der Gesamtsteuerzustand s^t zum Zeitpunkt t ist dann durch folgendes n-Tupel definiert:

$$s^t = (s_1^t, \ldots, s_n^t)$$

wobei $s^t \in S$ und $S = (S1 \times \ldots \times Sn)$ ist. Mit S wird also die Menge aller Gesamtkommunikationszustände bezeichnet.
In einem Automatisierungsprogramm seien m Prozessen bzw. Prozeßgruppen Wartebereiche vorgelagert. Der Gesamtwartebereichszustand zum Zeitpunkt t ist dann durch folgendes m-Tupel definiert:

$$w^t = (w_1^t, \ldots, w_m^t)$$

wobei $w^t \in W$ und $W = (W1 \times \ldots \times w1)$ ist. Mit W wird also die Menge aller Gesamtwartebereichszustände bezeichnet.
In einem Automatisierungsprogramm gibt es o Prozesse, die über Prozeßzeigervariable verfügen. Der Gesamtprozeßzeigerzustand zum Zeitpunkt t ist dann durch folgendes o-Tupel definiert:

$$z^t = (z_j^t, \ldots, z_k^t),$$

wobei z\:z und Z=(Zjx...xzk) ist. Mit Z wird also die Menge aller Gesamtprozeßzeigerzustände bezeichnet.

Der Gesamtzustand g eines Automatisierungsprogramms zum Zeitpunkt t wird durch folgendes Tripel definiert:

$$g^t = (s^t, w^t, z^t).$$

Beim Start eines Automatisierungsprogramms befinden sich alle Prozesse im Startsteuerzustand, alle Wartebereiche sind leer und alle Prozeßzeigervariablen sind unbelegt. Der Gesamtzustand eines Automatisierungsprogramms beim Start wird mit dem Index 1 bezeichnet.
Ausgehend vom Anfangszustand g^1 können dann alle erreichbaren Gesamtzustände gemäß den in den Abschnitt 5.2 genannten Regeln abgeleitet werden.

Die Menge aller möglichen Gesamtzustände ist endlich.
Begründung:

1. Die Menge der Prozesse in einem Prozeßsystem ist endlich. Ebenso ist die Menge der Steuerzustände eines Prozesses endlich. Daraus folgt, daß auch die Menge aller möglichen Gesamtsteuerzustände endlich ist.
2. Die Anzahl der Prozesse bzw. Prozeßgruppen, denen Wartebereiche vorgelagert sind, ist ebenfalls endlich (siehe Punkt 1). Jeder Prozeß empfängt nur eine endliche Menge von Nachrichten mit unterschiedlichen Namen von einer endlichen Menge von Prozessen. Dies heißt, daß auch die Menge aller möglichen Gesamtwartebereichszustände endlich ist.
3. Die Anzahl der Prozesse, die über Prozeßzeigervariable verfügen, ist natürlich endlich. Ebenso ist die Menge der Prozeßzeigervariablen pro Prozeß endlich. Da jeder Prozeß nur von einer endlichen Menge von Prozessen Nachrichten empfängt bzw. nur an eine endliche Menge von Prozessen Nachrichten sendet, ist auch die Menge der Gesamtprozeßzeigerzustände endlich.

Aus den Punkten eins, zwei und drei folgt nun, daß auch die Menge aller möglichen Gesamtzustände endlich ist und damit auch die Menge der von einem Startgesamtzustand aus erreichbaren Gesamtzustände.

Durch die obige Definition des Gesamtsteuerzustands läßt sich auch ex akter definieren, wann sich Prozesse verklemmen.
Es wird der Gesamtzustand g^x erreicht. In diesem Gesamtzustand soll der Gesamtsteuerzustand $s^x = (s_1, \cdots s_n)$ sein. Sind vom Gesamtzustand g^x nur Gesamtzustände erreichbar, für die gilt, daß sich die Steuerzustände einiger Prozesse gegenüber ihren Steuerzuständen in s^x nicht mehr verändern, so haben sich diese Prozesse verklemmt.
Wie für ein Prozeßsystem ausgehend vom Gesamtstartzustand alle erreichbaren Gesamtzustände konstruiert werden, soll mit dem nachfolgenden Beispiel gezeigt werden.

Beispiel:
Ein Programm besteht aus den zwei Prozessen P1 und P2. Dem Prozeß p1 ist ein Wartebereich mit zwei Plätzen vorgelagert. Kein Prozeß verfügt über Prozeßzeigervariable. Bild 5.25 zeigt die Ablaufsteuerungen der beiden Prozesse.

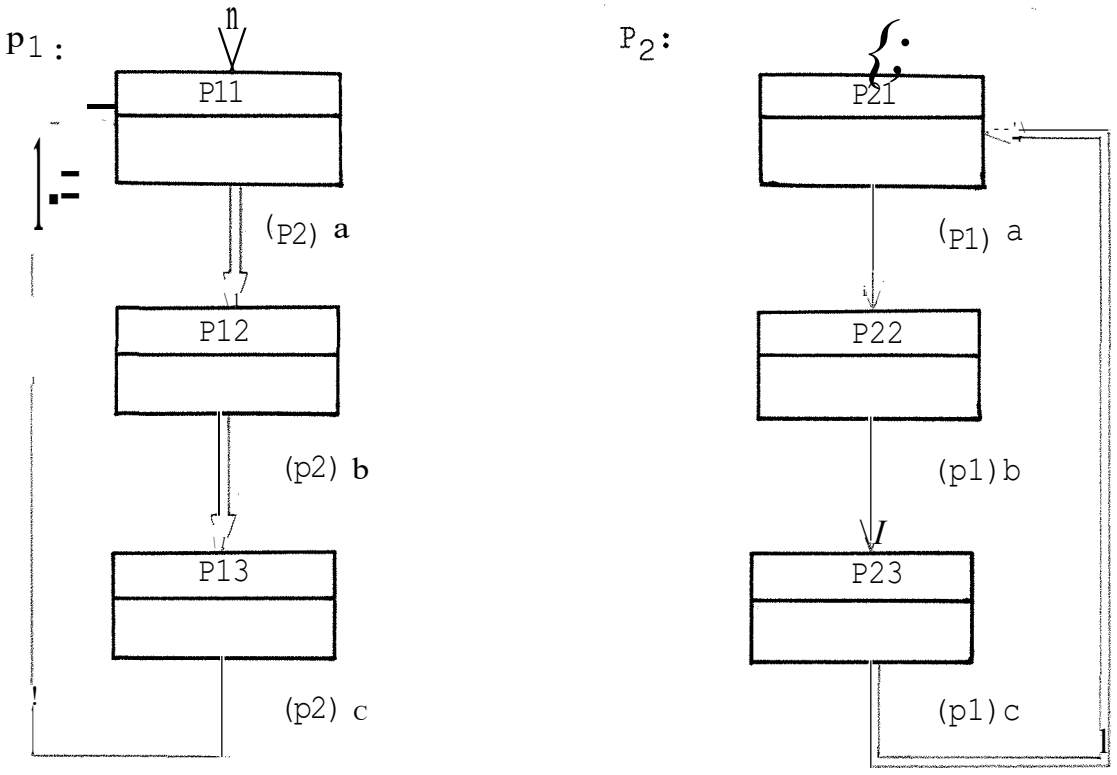
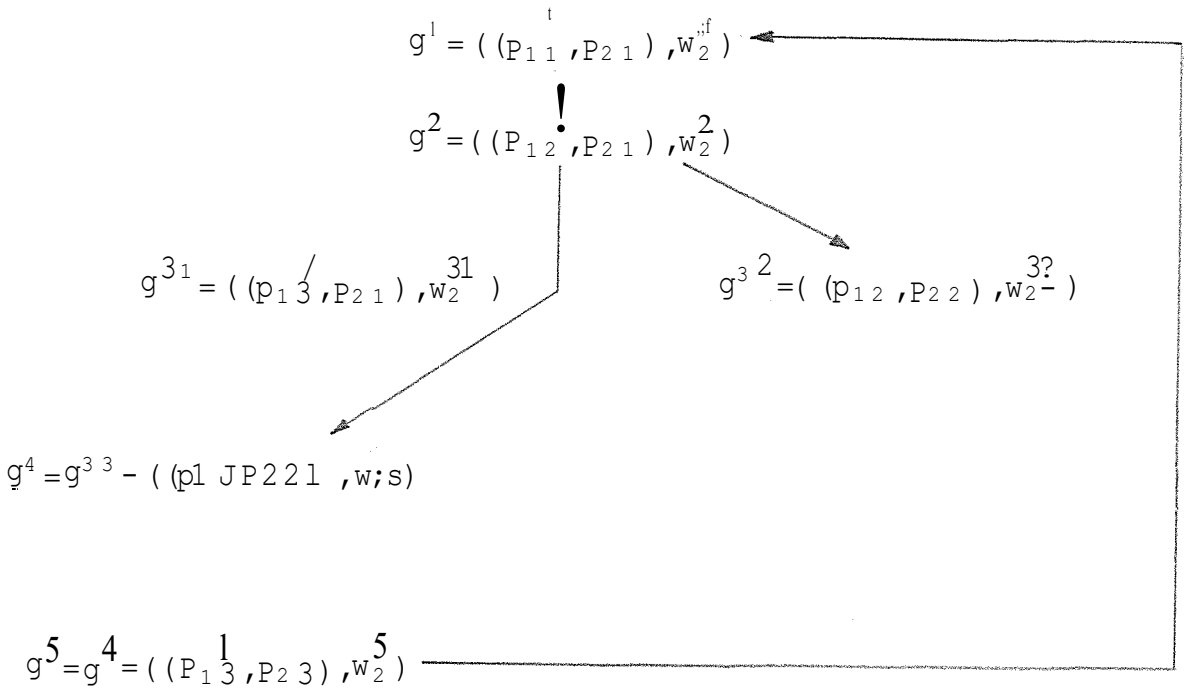


Bild 5.25 : Ablaufsteuerungen der Prozesse P1 und P2

Aus schreibtechnischen Gründen werden im folgenden bei der Beschreibung der Gesamtzustände die Komponenten mit dem Gesamt-

schreibung der Gesamtzustände die Komponenten mit dem Gesamtprozeßzeigerzustand (es gibt keine Prozeßzeigervariable) weggelassen und das Tupel das den Gesamtsteuerzustand beschreibt, direkt in den Gesamtzustand eingesetzt.
Der Anfangszustand ist $g^1 = ((p1_1, p_1), w1)$. Das Bild 5.26 zeigt alle erreichbaren Folgezustände.



$w_2^{//}$: Wartebereich ist leer
$w_2^{2..}$: Wartebereich ist mit (p_1)a belegt
w_2^{31} : Wartebereich belegt mit (p_1)a und (p_1)b
w_2^{32} : Wartebereich ist leer
w_2^{33} : Wartebereich ist mit (p_1)b belegt
w_2^5 : Wartebereich ist leer

Bild 5.26 : Erreichbare Gesamtzustände der Prozesse P_1 und P2

Im Gesamtzustand g^1 befinden sich beide Prozesse im Anfangssteuerzustand und der Wartebereich ist leer. Der einzig mögliche Gesamtzustand nach g^1 ist $g^2 = ((p_{12}, p_{21}), w_2)$. In diesem Gesamtzustand hat der Prozeß P_1 das Kommunikationselement

befindet sich nun im Steuerzustand P_{12}.
Nach dem Gesamtzustand g^2 sind drei Gesamtzustände möglich:

1. $g^{31} = ((P_{13}, P21), wt^1)$
   Der Prozeß P_1 hat das Kommunikationselement $(p_1)b$ im Wartebereich abgelegt und befindet sich nun im Steuerzustand P_{13}. Im Wartebereich befinden sich nun die Kommunikationselemente $(p_1)a$ und $(p_1)b$. Der Prozeß P_2 ist noch in keinen anderen Steuerzustand übergegangen.

2. $g^{32} = ((P1_2, P_{22}), w)$
   Der Prozeß p_2 hat aus dem Wartebereich das Kommunikationselement $(p_1)a$ entfernt und befindet sich nun im Steuerzustand $p2_2$. Der Wartebereich ist leer. Der Prozeß $P1$ befindet sich noch im Steuerzustand P_{12}.

3. $g^{33} = ((P_{13}, P_{22}), w)$
   Der Prozeß P_1 hat das Kommunikationselement $(p_1)b$ im Wartebereich abgelegt und der Prozeß P_2 hat das Kommunikationselement $(p1)a$ aus dem Wartebereich entfernt. Im Wartebereich befindet sich somit nur das Kommunikationselement $(P_1)b$.

Der Gesamtzustand g^{33} ist zugleich auch der Gesamtzustand der nach dem Gesamtzustand g^{31} möglich ist, d.h. $g^{33} = g^4$. Denn wenn im Gesamtzustand g^{31} der Prozeß p_2 das Kommunikationselement $(p_1)a$ aus dem Wartebereich entfernt und somit in den Steuerzustand p_{22} übergeht, entspricht der neue Gesamtzustand g^4 dem Gesamtzustand g^{33}.
Im Gesamtzustand g^5, der den Gesamtzuständen g^{32} und g^{33} folgt, ist der Wartebereich leer und P_1 wartet auf die Nachricht c vom Prozeß P_2. Nachdem der Prozeß P_2 diese Nachricht gesendet hat bzw. sie von P_1 empfangen wurde, wird wieder ein Gesamtzustand erreicht, der mit g^1 identisch ist.

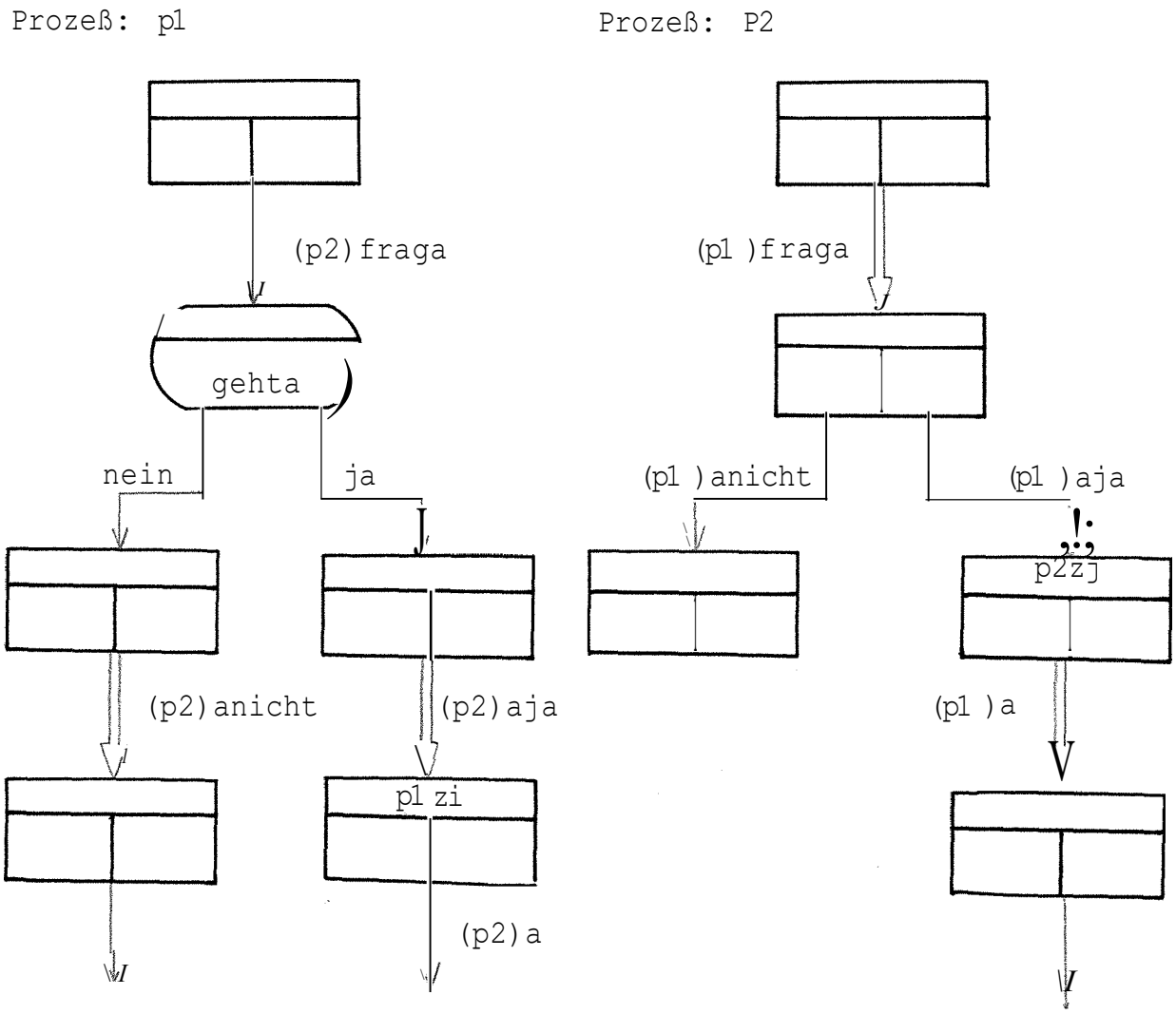
5.3.2.1. Der Zusammenhang zwischen den einzelnen Verklemmungsarten

Behauptung:
Konnte bei einem System nachgewiesen werden, daß es weder Benutzermaschinen-, Daten- noch Kommunikationsverklemmungen enthält, so enthält es auch insgesamt keine Verklemmungen, d.h. durch das Zusammenwirken von mehreren Blockadeursachen kann es zu keinen Verklemmungen kommen.

Begründung:
Da bei den einzelnen Verklemmungsuntersuchungen immer von den ungünstigsten Fällen ausgegangen wurde und die einzelnen Blokkierungsursachen unabhängig voneinander sind, kann es durch das Zusammenwirken von Benutzermaschinenblockade, Datenblockaden und Komm·unikationsblockaden zu keinen Verklemmungen kommen.
Bei der Suche nach Benutzermaschinenverklemmungen wird von allen durch die Ablaufsteuerungen erlaubte Aufrufreihenfolgen für interne Operationen und Funktionen ausgegangen. Diese Untersuchung ist also unabhängig von Daten- oder Kommunikationsblockaden. Deshalb kann es durch ein spezielles durch die Ablaufsteuerung erlaubtes Kommunikationsverhalten nicht zu Verklemmungen kommen.
Bei der Untersuchung auf Datenverklemmungen wird ebenfalls der ungünstigste Fall angenommen. Es werden keine Wartebereiche und Zeitüberwachungen angenommen.
Wenn in einem Kommunikationsausdruck, der empfangen werden soll, eine Nachricht enthalten ist, deren Eingabeoperationen nicht immer ausführbar sind, so wird der entsprechende Kommunikationszustand bereits als datenblockadegefährdet eingestuft. Es wird dann untersucht, welche Datenzustände in diesem Kommunikationszustand möglich sind. Bei den entsprechenden Sendeprozessen werden alle Kommunikationszustände berücksichtigt, in denen die entsprechende Nachricht an den zu untersuchenden Prozeß gesendet wird. Dadurch wird die Untersuchung auf Datenverklemmungen unabhängig davon, welche Gesamtzustände erreichbar sind.

Die Untersuchung auf Kommunikationsverklemmungen ist unabhängig von allen anderen Verklemmungsarten.
Die obige Behauptung stellt allerdings nur eine hinreichende Bedingung für die Verklemmungsfreiheit dar, es ist keine notwendige. Denn treten z.B. in einem Prozeßsystem Datenverklemmungen auf, so können diese durch ein entsprechendes Kommunikationsverhalten vermieden werden. Umgekehrt, wenn Kommunikationsverklemmungen auftreten, können diese durch den Zustand der Benutzermaschine überhaupt nicht zustande kommen.
Das Zusammenwirken von Kommunikationsverhalten und Zustand der Benutzermaschine soll an Hand von Beispielen verdeutlicht werden.
Das erste Beispiel (Bild 5.27) zeigt, wie eine Datenverklemmung durch ein entsprechendes Kommunikationsverhalten aufgehoben wird.

Beispiel 1:



```
/* Ausschnitt aus der Benutzer-
   maschine                    */

3. BENUTZERMASCHINE
3.1. AUSGABEFUNKTIONEN:anicht, ..
     aja, ...
3.2. EINGABEOPERATIONEN:a,fraga,

3.3. INTERNE OPERATIONEN: ..... .
3.4. INTERNE FUNKTIONEN:gehta, ..
3.5. BENUTZERMASCHINE:

var:  TYP integer
merk: TYP integer

gehta:
  IF merk+var 10 THEN
     RETURN:ja
  ELSE
     RETURN:nein
  FIN

a: (par)
 {var+par<10} a {var='var+par}

fraga: (par)
 {TRUE} fraga {merk=par}
```

```
/*Ausschnitt aus der Benut-
  zermaschine               */

3. BENUTZERMASCHINE
3.1. AUSGABEFUNKTIONEN:a,
     fraga, ....
3.2. EINGABEOPERATIONEN:
     aja,anicht, ..... .
3.3. INTERNE OPERATIONEN: .. .
3.4. INTERNE FUNKTIONEN: ... .
3.5. BENUTZERMASCHINE:

nn: TYP integer

fraga(par)
 {TRUE} fraga {par=nn}

anicht:
  SKIP

aja:
  SKIP
```

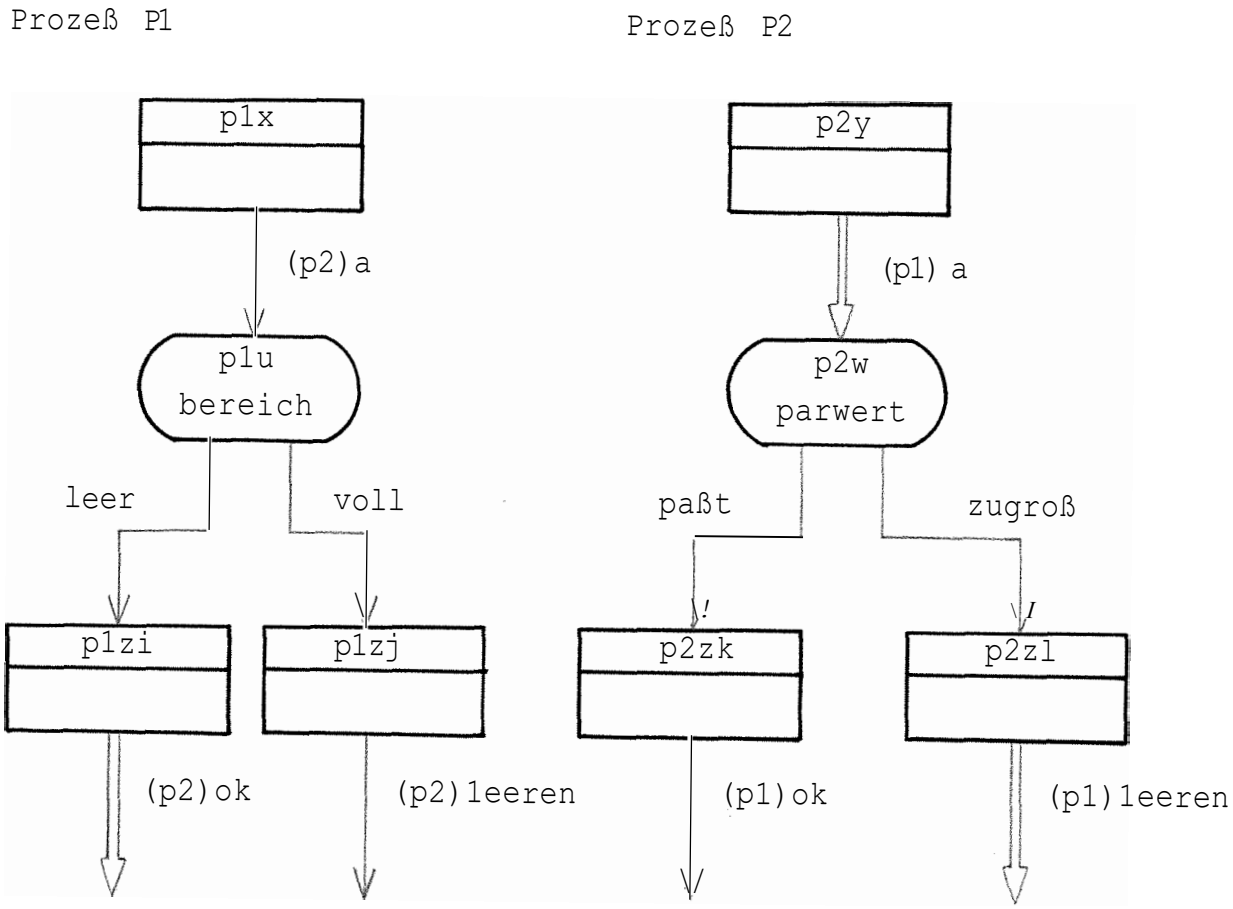
Bild 5.27 : Beispiel, wie durch ein entsprechendes Kommunikationsverhalten Datenverklemmungen nicht zustandekommen.

Der Kommunikationszustand p2zj im Prozeß p2 soll der einzige sein, in dem der Prozeß p2 die Nachricht a an den Prozeß p1 sendet. Dem Prozeß p1 soll kein Wartebereich vorgelagert sein. Wird der Zustand p1zi im Prozeß p1 daraufhin untersucht, ob es in ihm zu einer Datenverklemmung kommen kann, so stellt man

fest, daß dies der Fall sein kann, da z.B. der Wert der Variablen nn im Prozeß p2 größer zehn sein kann, wenn sich dieser im Zustand p2zj befindet.
Allerdings kann man aus der Ablaufsteuerung der beiden Prozesse sehen, daß der Prozeß p1 erst nachfragt, ob die Eingabeoperation der Nachricht a ausgeführt werden kann. Die Nachricht a sendet der Prozeß p2 nur, wenn der Prozeß p1 die Anfrage von Prozeß p2 entsprechend beantwortet hat.

Das nächste Beispiel (Bild 5.28) zeigt wie eine Kommunikationsverklemmung wegen des Datenzustands nicht zustande kommen kann.

Beispiel 2:



```
/* Ausschnitt aus der Benutzer-
   maschine von Prozeß P1      */
3. BENUTZERMASCHINE
3.1. AUSGABEFUNKTIONEN: .........
3.2. EINGABEOPERATIONEN: .. ,a, ....
3.3. INTERNE OPERATIONEN: ........
3.4. INTERNE FUNKTIONEN:bereich,.
3.5. BENUTZERMASCHINE:

var: TYP integer

a: (par)
  {TRUE} a -tvar=par

bereich:
  IF var 10 THEN
     RETURN:leer
  ELSE
     RETURN:voll
  FIN
```

```
/* Ausschnitt aus der Benutzer-
   maschine von Prozeß P2      */
3. BENUTZERMASCHINE
3.1. AUSGABEFUNKTIONEN: .. ,a, .. .
3.2. EINGABEOPERATIONEN: ..... .
3.3. INTERNE OPERATIONEN: ..... .
3.4. INTERNE FUNKTIONEN:parwert
3.5. BENUTZERMASCHINE:

b: TYP integer

a(par)
 TRUE  a  par=b

parwert:
  IF n 10 THEN
     RETURN: paßt
  ELSE
     RETURN: zugroß
  FIN
```

Bild 5.28 : Beispiel für eine Kommunikationsverklemmung die durch den Datenzustand nicht zustande kommt

Wird nur das Kommunikationsverhalten berücksichtigt, so können sich die Prozesse p1 und p2 verklemmen.
Der Prozeß p1 soll sich im Kommunikationszustand p1x und p2 in p2y befinden. Der Prozeß p2 sendet an den Prozeß p1 die Nachricht a. Der Prozeß p1 geht in den Internzustand p1u und der Prozeß in den Internzustand p2w über.
Wird nur das Kommunikationsverhalten der beiden Prozesse berücksichtigt, so kann der Prozeß p1 in den Kommunikationszustand p1zi und p2 in den Kommunikationszustand p2zl übergehen. Da nun p1 ausschließlich von p2 und p2 ausschließlich von p1 eine Nachricht erwarten, verklemmen sich die beiden Prozesse. Bezieht man allerdings in diese Betrachtung die Datenzustände mit ein, so sieht man, daß es zu dieser Kommunikationsverklem-

mung nicht kommen kann. Denn geht der Prozeß p1 in den Zustand p1i über so geht der Prozeß p2 immer in den Zustand p2k über, und somit kann es zu keiner Kommunikationsverklemmung kommen.

5.4 Sprachkonstrukte zum Umsetzen einer Spezifikation in ein Programm

Es soll nun ein Konzept vorgestellt werden, das für eine Programmiersprache sowohl gemeinsame Objekte, als auch einen komfortablen Botschaftsmechanismus vorsieht /FHKK82/,/FHKK83/. Spezifikationen mit Einzelprozessen und Prozeßbündeln können, was das Synchronisations- und Kommunikationsverhalten betrifft, manuell relativ einfach und mehr oder weniger schematisch in ein Programm umgesetzt werden. Die Probleme beim Umsetzen der sequentiellen Programmteile von der Spezifikation in ein Programm werden allerdings nicht näher diskutiert, da diese wesentlich von der verwendeten Spezifikationstechnik für die sequentiellen Teile abhängen. Die notwendigen Sprachkonstrukte zur Synchronisation und Kommunikation wurden in die Programmiersprache PEARL /DIN66253/ eingebettet.

5.4.1 Beschreibung der Wirtssprache PEARL

PEARL ist eine höhere Programmiersprache, deren Sprachumfang Echtzeitelemente enthält. Dadurch eignet sich PEARL zur Formulierung von Aufgaben der Prozeßautomatisierung. In PEARL ist ein Modulkonzept verwirklicht. Ein Programm besteht aus einem oder mehreren Moduln, die getrennt übersetzbar sind. Ein Modul enthält einen Systemteil und/oder einen Problemteil. Der Systemteil beschreibt die benötigten Hardwarekonfiguration, der Problemteil enthält die Algorithmen, die zur Lösung eines Problems verwendet werden, und besteht aus einer Folge von Spezifikationen und Deklarationen.
Durch eine Spezifikation erhält ein Modul Zugriffsrechte auf Variable und Prozeduren, die in anderen Moduln deklariert sind. Im Zusammenhang mit PEARL wird unter Spezifikation nicht die Beschreibung von Programmeigenschaften verstanden, sondern mit einer sogenannten Spezifikation werden in einem PEARL-Modul globale Variable, Prozeduren, Semaphore und Tasks bekannt gemacht, die in einem anderen Modul deklariert sind.
Ein PEARL-Programmsystem besteht meistens aus mehreren Tasks.

Eine Task kann als die Deklaration eines selbständig, parallel zu anderen Prozessen ablaufenden Rechenprozesses aufgefaßt werden.
Die Tasks unterliegen als Rechenprozesse einer zentralen Prozeßverwaltung, dem PEARL-Betriebssystem (PBS).
Die Tasks können sich in verschiedenen Zuständen befinden. PEARL bietet Sprachkonstrukte, mit denen ein Zustandswechsel von Tasks veranlaßt werden kann. Die möglichen Zustände einer Task sind:

- ruhend
  Die Task ist im System vorhanden und belegt außer dem beim Laden zugeteilten Speicherplatz keine Betriebsmittel.

- ablauffähig
  Der Task ist ein Prozessor zugeteilt oder
  die Task wartet auf ein Betriebsmittel, das vom Betriebssystem vergeben wird oder
  die Task benutzt ein zugeteiltes Betriebsmittel.

- angehalten
  Eine Task geht durch eine Anhalte- (SUSPEND) oder Aussetzanweisung (PREVENT) in diesen Zustand über. Aus diesem Zustand kann eine Task nur durch eine andere Task oder durch ein erwartetes Ereignis in einen anderen Zustand gelangen.

- wartend
  Eine Task kann in den Zustand wartend gelangen, wenn ihr Ablauf durch Semaphore gesteuert wird. Eine Task kann nur durch Eigeninitiative, beim Ausführen von Semaphoreoperationen, wartend werden.

PEARL-Tasks synchronisieren sich über Semaphore und tauschen Daten über globale Variable aus. Deshalb ist PEARL für Rechensysteme mit gemeinsamem Speicher gut geeignet.
Neben diesen Möglichkeien, den Ablauf der parallelen Prozesse zu steuern, gibt es in PEARL noch umfangreiche Ein-/Ausgabeanweisungen. Mit dieser Anweisungsart wird die Kommunikation mit

einem zu steuernden technischen Prozeß bzw. mit dem Benutzer abgewickelt.
Außerdem gibt es sogenannte Interrupt-Einplanungen (Schedules). Mit diesem Anweisungstyp können Prozesse auf bestimmte Ereignisse warten, z.B. kann eine Task auf einen Interrupt mit der Bedeutung 'Füllstand erreicht' warten und dann entsprechende Reaktionen einleiten.

5.4.2. Beschreibung der Sprachkonstrukte zur Synchronisation und Kommunikation

5.4.2.1. Botschaften

In PEARL gibt es bereits durch entsprechende Ein/Ausgabeanweisungen und Interrupt-Einplanungen Möglichkeiten zur Kommunikation bzw. Synchronisation von Automatisierungsprogrammen mit dem technischen Prozeß.
Das hier vorgeschlagene PEARL für verteilte Automatisierungsprogramme geeignet sein soll, wurde es um Botschaftsmechanismen erweitert. PEARL-Tasks (Rechenprozesse) können also untereinander Daten mit Hilfe von Botschaften austauschen /FHKK82/,/FHKK83/,/KLEBE83/,/KUMM83/.
Damit man bei einer Task schnell einen Überblick erhält, mit welchen anderen Tasks sie Botschaften austauscht, wurde der sogenannte Taskkopf, mit dem eine PEARL-Task eingeleitet wird, erweitert. In den Taskkopf wurde der diese Task betreffende Teil der Kommunikationsstruktur mit aufgenommen.
Bisher enthielt der Taskkopf im wesentlichen den Tasknamen und die Priorität der Task, sowie Angaben, ob eine Task speicherresident ist, und ob die Task auch in anderen PEARL-Moduln bekannt sein soll. Bild 5.29 zeigt die Syntax des erweiterten Taskkopfs, wobei die erste Zeile den ursprünglichen Taskkopf darstellt.

```
taskname":" "TASK"  "PRIO" prioritaet   "GLOBAL"  "RESIDENT"
          ["TRANSMITS" sendebotschaftsname(n)
                       [parameterbeschreibung]
                     "TO" empfaengername(n)
                        {,sendebotschaftsname(n)
                         [parameterbeschreibung]
                     "TO", empfaengername(n)
                        } ... }
          [11RECEIVES"  empfangsbotschaftsname(n)
                       [parameterbeschreibung]
                     "FROM" sendername(n)
                       {, empfangsbotschaftsname(n)
                          parameterbeschreibung
                     "FROM" sendername(n)
                       } ... }
                     "NOBUFFER" /
                     "BUFFER" wartebereichsgroesse] "'"
```

Bild 5.29: Syntax des erweiterten Taskkopfs.

Mit der TRANSMITS-Klausel werden die Sendebotschaften und mit der RECEIVES- Klausel die Empfangsbotschaften vereinbart. Analog zum Spezifikationskonzept besteht eine Botschaft aus dem Botschaftsnamen und einer optionalen Parameterliste. Diese setzt sich zusammen aus dem Parameternamen (bzw. einer Parameterliste), dem der Parametertyp folgt. Als Parametertyp sind in der Spracherweiterung elementare PEARL-Datentypen erlaubt. In der Spezifikation ist jeder Nachricht eine Funktion oder Operation der Benutzermaschine zugeordnet. In der Sprach-erweiterung ist eine Botschaft vergleichbar mit einem Objekt einer bestimmten Datenstruktur. Mit den entsprechenden Klauseln im Taskkopf werden Datenobjekte vereinbart, die denselben Namen wie die Nachricht haben, wobei die Namen der Nachrichtenparameter den Namen der Strukturkomponenten entsprechen. Mit Botschaften kann somit genauso umgegangen werden wie mit den sogenannnten PEARL-Datenstrukturen (Es ist zu beachten, daß mit 'Datenstrukturen' in PEARL Objekte bestimmter Struktur

gemeint sind und nicht wie in PASCAL Muster für bestimmte Datentypen.). Wie den Nachrichten die in der Spezifikation angegebene Funktion oder Operation zugeordnet wird, soll später beschrieben werden.
Bei der TRANSMITS-Klausel folgt der Botschaftsvereinbarung das Schlüsselwort "TO" und der bzw. die möglichen Empfängernamen. Bei der RECEIVES-Klausel steht an Stelle des Schlüsselwortes "TO" das Schlüsselwort "FROM", danach folgen der bzw. die möglichen Sendernamen. Anschließend an die RECEIVES-Klausel folgt die Wartebereichsvereinbarung. Bei der Angabe von NOBUFFER existiert kein Wartebereich. Bei der Angabe von BUFFER existiert ein Wartebereich endlicher Größe.
Botschaften, die in der TRANSMITS-Klausel vereinbart sind, dürfen an die angegebenen Empfänger gesendet, und solche die in der RECEIVES-Klausel vereinbart sind, dürfen von den angegebenen Sendern empfangen werden. Zum Senden bzw. Empfangen von Botschaften gibt es die Sende- bzw. Empfangsanweisungen. Die Darstellung dieser Anweisungstypen ähnelt den PEARL-Ein-/Ausgabeanweisungen.

Sendeanweisung:
"TRANSMIT" "FROM" botschaftsname "TO" empfängername";"

Mit der TRANSMIT-Anweisung wird diejenige Botschaft, die durch den angegebenen Namen identifiziert wird, mit ihren Parametern gesendet. Letztere haben ihren Wert per Zuweisung erhalten, d.h. die Funktion, die in der Spezifikation die Parameterwerte implizit bestimmt, muß durch eine entsprechende Anzahl von Wertzuweisungen, die der Sendeoperation vorausgehen, explizit realisiert werden. Ansonsten gilt für die Semantik der Sendeanweisung dasselbe wie für das Senden in der Spezifikation. Die Empfangsanweisung ist analog aufgebaut.

Empfangsanweisung:"
"RECEIVE" "FROM" sendernamen "TO" botschaftsname";"

Mit der RECEIVE-Anweisung werden Botschaften aus dem Wartebereich bzw. falls kein Wartebereich vorhanden ist, von anderen

Tasks anstehende Botschaften übernommen. Das heißt, ihre Parameterwerte werden in die Strukturkomponenten, der Datenstruktur, die dieser Empfangsbotschaft entspricht, kopiert. In der PEARL-Erweiterung wird also jeder Nachricht beim Empfangen eine konstante Eingabeoperation, nämlich das Übernehmen der Nachrichtenparameter, zugeordnet. In der Spezifikation wird beim Empfang unterschiedlicher Nachrichten implizit eine entsprechende Operation ausgelöst. Bei der Umsetzung der Spezifikation muß diese implizit ausgelöste Operation explizit durch eine entsprechende Anweisungsfolge, die der Empfangsanweisung folgt, realisiert werden.

5.4.2.2 Nichtdeterministische Kontrollanweisungen

Mit den bisherigen Sende- und Empfangsanweisungen kann jeweils nur eine Botschaft an einen Prozeß gesendet bzw. nur eine Botschaft von einem bestimmten Prozeß erwartet werden. Deshalb wurden die von Dijkstra /DIJK75/ eingeführten nichtdeterministischen Kontrollanweisungen (Guarded Statements) in PEARL eingebaut (siehe auch Kapitel 3). Mit dieser Anweisungsart wird es möglich, auf verschiedene Nachrichten von verschiedenen Prozessen alternativ zu warten bzw. verschiedene Nachrichten an verschiedene Prozesse alternativ zu senden.
Den Aufbau der nichtdeterministischen Kontrollanweisungen, wie sie in PEARL eingebaut sind, zeigt Bild 5.30.

```
guarded-statement:  "GUARDED"   region / command
                    "GUARDEND"  "!"

region:  "REGION"
         guards
         ["TIMEOUT"  "AFTER"  duration-ausdruck
                      "REACT"   anweisungsfolge]

command:  "COMMAND"
          guards
          ["OUTREACT"  anweisungsfolge]

guards:  "GUARD"  guard-element,{guard-element}  ...
                "REACT"  anweisungsfolge
         {"GUARD"  guard-element,   guard-element   ...
                 "REACT"   anweisungsfolge }  ...

guard-element:   "WHEN"  interrupt /
                  request-anweisung /
                  get-anweisung /
                  receive-anweisung /
                  release-anweisung /
                  put-anweisung /
                  transmit-anweisung
```

Bild 5.30 : Aufbau der nichtdeterministischen Kontroll-anweisungen

Die Guarded Statements werden von oben nach unten, die Guards von links nach rechts abgearbeitet. Ein Guarded Statement entspricht in der Spezifikation einem Kommunikationsausdruck und ein Guard einem Kommunikationsterm.
Jedem Kommunikationszustand in der Spezifikation wird also ein entsprechendes Guarded Statement in der Implementation zugeordnet.
Wie aus der Syntax ersichtlich ist, kann in einem Guarded Statement eine Überwachungszeit (TIMEOUT) angegeben werden.

Die Priorität der einzelnen Übergänge in der Spezifikation wird durch die Notationsreihenfolge der Guards realisiert.
In der Spezifikation wird auch die Kommunikation und Synchronisation mit einem technischen Prozeß bzw. mit dem Benutzer durch Botschaften beschrieben. Für diese Kommunikation und Synchronisationsprobleme gibt es in PEARL bereits Anweisungsarten, nämlich Ein-/Ausgabeanweisungen und Interrupt-Einplanungen. Diese Anweisungsarten sind ebenfalls als Guard-Elemente (siehe Bild 5.30) erlaubt. Bei der Umsetzung einer Spezifikation in ein PEARL-Programm muß also entschieden werden, ob eine Botschaft in der Spezifikation auf eine Botschaft in PEARL, eine Ein-/Ausgabeanweisung oder eine Interrupt-Einpl.anung abgebildet wird, wobei zur Interprozeßkommunikation im allgemeinen Botschaften und zur Kommunikation und Synchronisation mit dem technischen Prozeß bzw. dem Benutzer Ein-/Ausgabeanweisungen und Interrupt-Einplanungen verwendet werden.
Bei den Botschaftsoperationen müssen, wie im vorigen Abschnitt geschildert, die den einzelnen Botschaften zugeordneten Funktionen und Operationen explizit durch PEARL-Anweisungsfolgen realisiert werden. Um dies auch bei Guarded Statements zu ermöglichen, kann bei jedem Guard ein sogenannter REACT-Zweig angegeben werden. In diesem REACT-Zweig dürfen beliebige Folgen von PEARL-Anweisungen vorkommen. In der Spezifikation dürfen von einem Kommunikationsknoten nur entweder Sende- oder Empfangskanten wegführen. Analog dazu dürfen in einem Guarded Statement als Guard-Elemente entweder nur GET-, RECEIVE-Anweisungen und Interrupt-Einplanungen oder nur PUT- und TRANSMIT-Anweisungen vorkommen.
Wie bereits erwähnt, gibt es in PEARL Semaphore und globale Variable. Befinden sich zwei Prozesse auf demselben Prozessor, so können sie auch über globale Variable und Semaphore kommunizieren bzw. sich synchronisieren. Dazu kann der Nachrichtenaustausch mit diesen Möglichkeiten simuliert werden /BEHF82/. Um zu ermöglichen, daß mehrere solcher 'simulierter' Nachrichten alternativ erwartet, bzw. alternativ gesendet werden können, sind auch Semaphoroperationen als Guard-Elemente erlaubt, wobei die REQUEST-Operation zu den Eingaben und die RELEASE-Operationen zu den Ausgaben zählen. Dadurch, daß die

Semaphoroperationen als Guard-Elemente zugelassen sind, können auch andere Synchronisationskonzepte einfach nachgebildet werden (siehe /KLEB83/).

5.4.3 Verteilung der Prozesse auf die Prozessoren

Als Grundlage zur Verteilung eines Programms auf die einzelnen Prozessoren werden die PEARL-Moduln verwendet /FHKK83/. In PEARL enthalten die Moduln einen Systemteil. Die Konfiguration der Programmverteilung fließt in den Systemteil ein.

Bei der Verteilung muß berücksichtigt werden:
- Prozesse, die auf verschiedenen Prozessoren laufen sollen, müssen in verschiedenen Moduln liegen,
- die Angabe der logischen Verbindung (zu anderen Prozessoren), über die ein entfernter Prozeß zu erreichen ist (im Systemteil).

Die Verteilung auf die verschiedenen Prozessoren wird durch das Laden der, die entsprechenden Prozesse enthaltenden Moduln, vorgenommen.

5.4.4 Einschränkungen gegenüber dem Spezifikationskonzept

Wie aus den vorigen Abschnitten ersichtlich ist, wurden die Möglichkeiten des Spezifikationskonzepts bzgl. den Strukturierungs- und Synchronisationsmöglichkeiten durch die PEARL-Erweiterung eingeschränkt.
Die PEARL-Erweiterung hat gegenüber dem Spezifikationskonzept folgende Einschränkungen:
1. Es gibt keine Prozeßgruppen.
2. Es gibt keine Einzelprozeß-, Prozeßbündel- bzw. Prozeßgruppentypen
3. Es gibt keine Datenblockaden.

Für die Einschränkungen gibt es folgende Gründe:

zu 1.
In PEARL gibt es keine Ansätze für Prozeßgruppen. Deren Einführung hätte tiefere Eingriffe in das Konzept von PEARL nach sich gezogen. Deshalb wurde darauf verzichtet.

zu 2.
Hier gelten ähnliche Überlegungen wie bei Punkt 1. In PEARL sind keine Typen von Tasks möglich.

zu 3.
Um Datenblockaden zu realisieren, müßte das Betriebssystem Zugriff auf die Programmvariablen haben. Dies dürfte nur mit großem Aufwand möglich sein. Da das Betriebssytem Zugriff auf Programmvariable hat, würde es in der Benutzt-Hierarchie auf einer höheren Ebene liegen als die Programmvariablen. Dies würde den bisherigen Gepflogenheiten entgegenstehen.

5.5 Diskussion des vorgestellten Spezifikations- und Implementationskonzepts und bisherige Erfahrungen

Das vorgeschlagene Konzept zur Spezifikation und Implementation von verteilten Automatisierungsprogrammen stellt einen pragmatischen Ansatz dar. Dies heißt vor allem, daß der Aspekt der Verifikation mehr oder weniger nicht beachtet wurde. Denn zur Verifikation wären ein formales Modell für das Spezifikationskonzept und eine Programmiersprache notwendig, deren Semantik formal durch ein logisches Formelsystem zu beschreiben wären.
Da keine neue Sprache zur Prozeßautomatisierung entworfen werden sollte bzw. konnte, wurde die Sprache PEARL als Wirtssprache für die entsprechenden Anweisungen verwendet. Die Semantik von PEARL ist nicht formal definiert.
Die bisherigen Erfahrungen liegen vor allem im Bereich der Datenfernübertragung. Das vorgestellte Spezifikationskonzept wurde vor allem zum Entwurf von DFÜ-Protokollen verwendet (/HART81/, /KRAG82/, /JORD83/). Dabei zeigte sich, daß die Spezifikationstechnik für den Programmentwurf hilfreich war.

Die spezifizierten DFÜ-Protokolle wurden allerdings nicht mit dem erweiterten PEARL implementiert. Hier zeigte sich, daß auch eine Umsetzung der Spezifikation in ein FORTRAN-Programm sehr schnell und nahezu schematisch möglich ist. Die Ablaufsteuerung erleichterte das Testen wesentlich.
In /BOEM82/ wurde die Spezifikationstechnik zum Entwurf von Programmen zur Meßwerterfassung in der Experimentalphysik verwendet.
In diesem Fall wurde das Spezifikationskonzept von einem Physiker verwendet. Dabei zeigte sich, daß es auch für Nichtinformatiker schnell erlernbar ist.
In /FLEI81/ wurde das Spezifikationskonzept zum Entwurf eines Betriebssystembausteines (Treiber für ein "komplexes" Gerät) verwendet. Dieser Treiber arbeitet interruptgesteuert. Mit der beschriebenen Spezifikationstechnik konnten die verschiedenen Interrupts und die nötigen Reaktionen gut beschrieben werden. Der Treiber wurde in Assembler implementiert.
Bisher gibt es allerdings kaum Rechnerunterstützung beim Erstellen von Spezifikationen nach dem vorgestellten Konzept. Erste Ansätze wurden in /LEIN82/ gemacht. Allerdings liegt dieser Arbeit eine noch etwas andere graphische Darstellung der Ablaufsteuerung zugrunde.

6 Beispiel

6.1 Informelle Problembeschreibung

Das folgende Beispiel soll die Anwendung des Spezifikationskonzepts und dessen Umsetzung in ein PEARL-Programm zeigen. Es handelt sich um die Steuerung einer Ampel mit Hilfe eines verteilten Programms. Bei diesem Beispiel ist es so, daß drei Prozessoren verwendet werden. Auf jedem Prozessor befindet sich nur eine Task (Prozeß). Natürlich sind für eine solche einfache Ampelsteuerung nicht drei Prozessoren notwendig und ökonomisch auch nicht vertretbar. Aber dieses Beispiel zeigt die Anwendung des Spezifikationskonzepts in einer überschaubaren Anwendung, aber es enthält trotzdem wesentliche Probleme der Prozeßautomatisierung mit verteilten Programmen.
An einer Kreuzung soll der Verkehr durch Ampeln bedarfsgerecht gesteuert werden, d.h. daß registriert wird, aus welcher Richtung ein Fahrzeug kommt und dann sollen die Ampeln entsprechend geschaltet werden. Das Bild 6.1 zeigt das zu lösende 'Verkehrsproblem'.

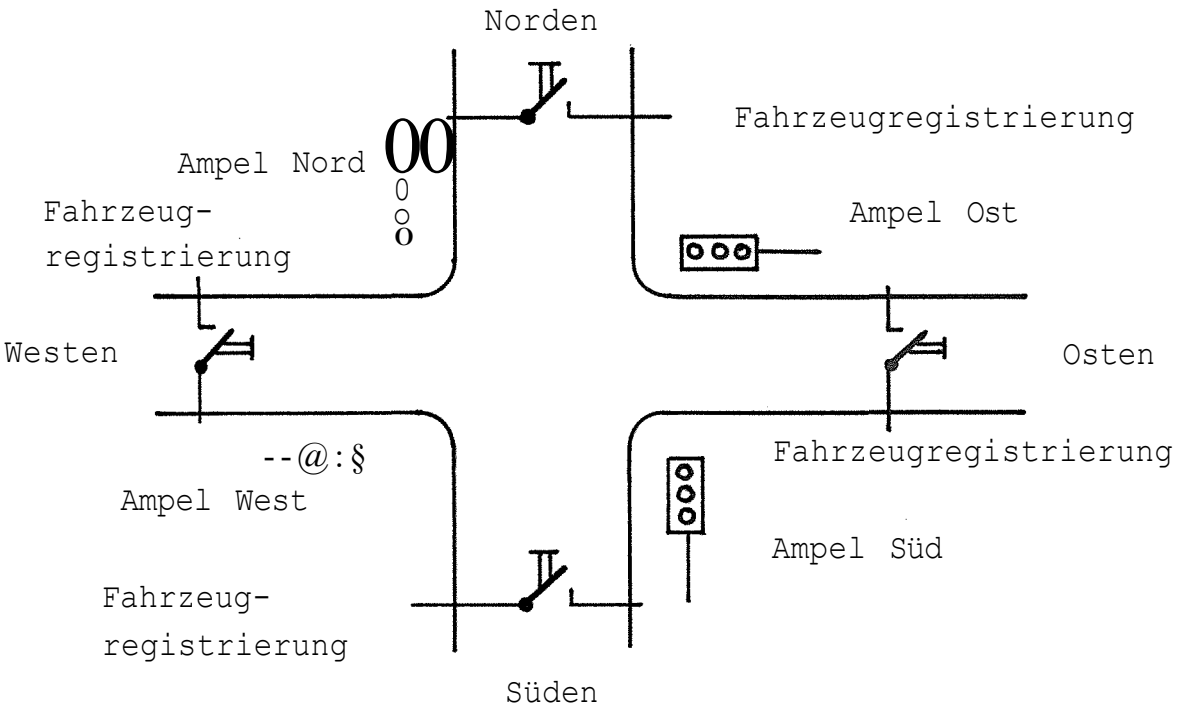


Bild 6.1 : Zu lösendes Verkehrsproblem

Im Normalfall zeigen alle vier Ampeln gelbes Blinklicht. Wird ein Fahrzeug registriert, wird die Ampel so geschaltet, daß das Fahrzeug passieren kann. Natürlich wird die entsprechende Gegenrichtung ebenfalls auf grün geschaltet. Die Ampelnbleiben dann mindestens 10 Sekunden in dieser Schaltsituation, bzw. bleiben es solange, solange Fahrzeuge aus den Richtungen registriert werden, für die freie Fahrt geschaltet ist. Danach zeigen alle Ampeln wieder das gelbe Blinklicht, außer aus den bisher gesperrten Richtungen wurde ein oder mehrere Fahrzeuge registriert. Dann wird die Ampel umgeschaltet, d.h. die bisher freien Richtungen werden gesperrt und die bisher gesperrten freigegeben. Die momentanen Schaltsituationen an den Ampeln sollen auf einem Drucker protokolliert werden.

Alle vier Ampeln werden durch eine Gruppe von Signalleitungen (12 Leitungen) angesteuert. Jede Belegung der Leitungen entspricht einer bestimmten Schaltsituation der Ampeln. Jede Belegung der Leitungen wird als eigene Botschaft aufgefaßt. So bedeutet z.B. die Leitungsbelegung mit dem Bitmuster '010010010010', daß alle Ampeln auf gelb geschaltet werden.

Die Fahrzeuge werden durch Induktionsschleifen registriert. In dem Modellprozeß sind die Induktionsschleifen durch Schalter ersetzt. Die Stellung eines jeden Schalters induziert eine entsprechende Botschaft an das Automatisierungsprogramm. Jeder Schalter kann an das Automatisierungsprogramm eine Botschaft senden. Geht der Schalter von einer bestimmten Stellung in eine bestimmte andere Stellung über (z.B. ansteigende Flanke), gilt eine Nachricht als gesendet.

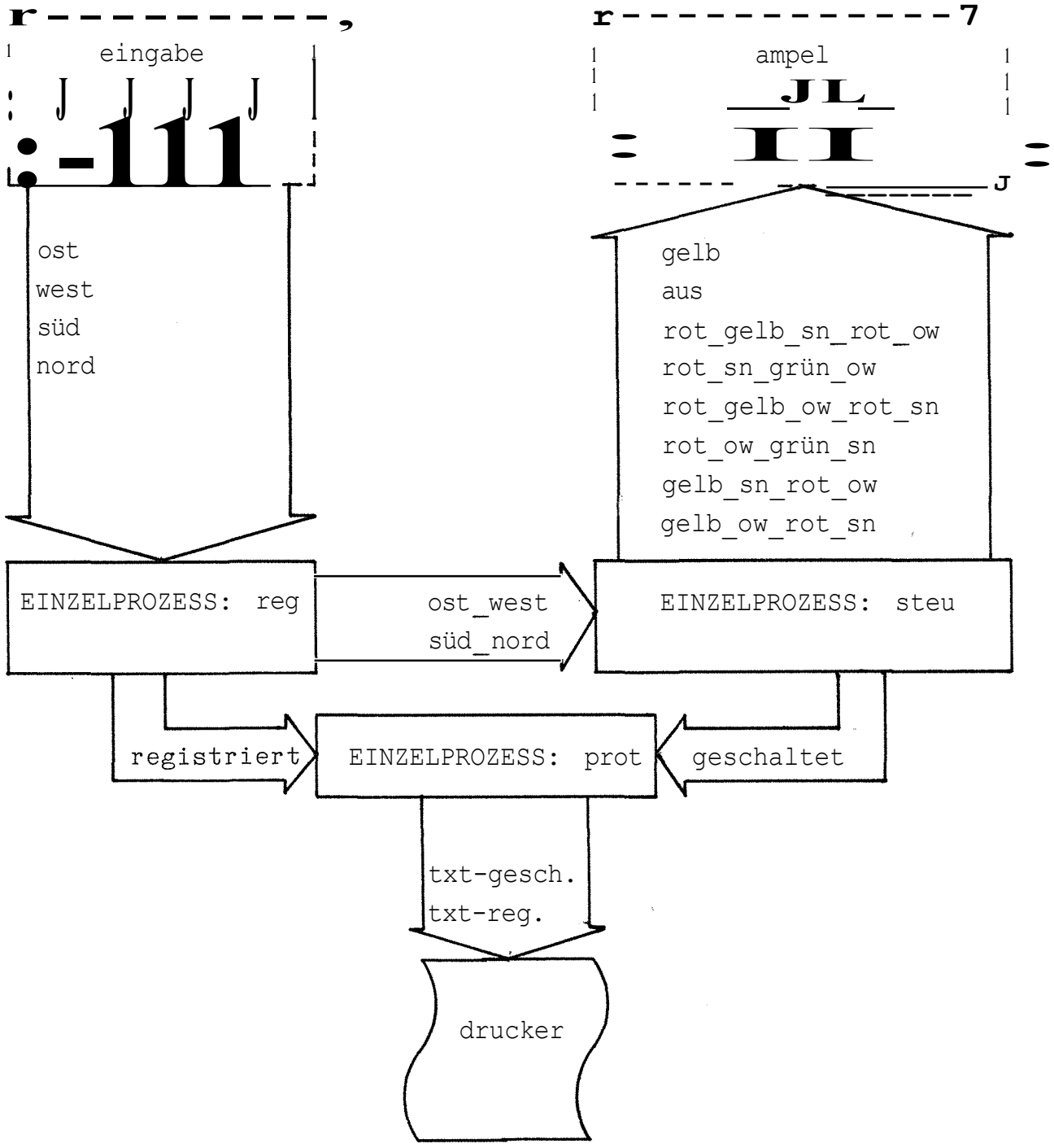
6.2 Programmspezifikation

Die Spezifikation des Ampelsteuerprogramms ist wie im Anhang geschildert aufgebaut. Da die Benutzermaschinen für die einzelnen Prozesse sehr einfach sind, wurde für deren Beschreibung die Umgangssprache verwendet.

PROZESSYSTEM: ampelsteuerung

I. KOMMUNIKATIONSSTRUKTUR

Komponenten des technischen Prozesses



III. ELEMENTE EINES PROZESSYSTEMS

1. DEKLARATIONEN

keine

2. DEFINITION WEITERER STRUKTURIERUNGSEINHEITEN

EINZELPROZESS: steu

1. KOMMUNIKATIONMASCHINE;

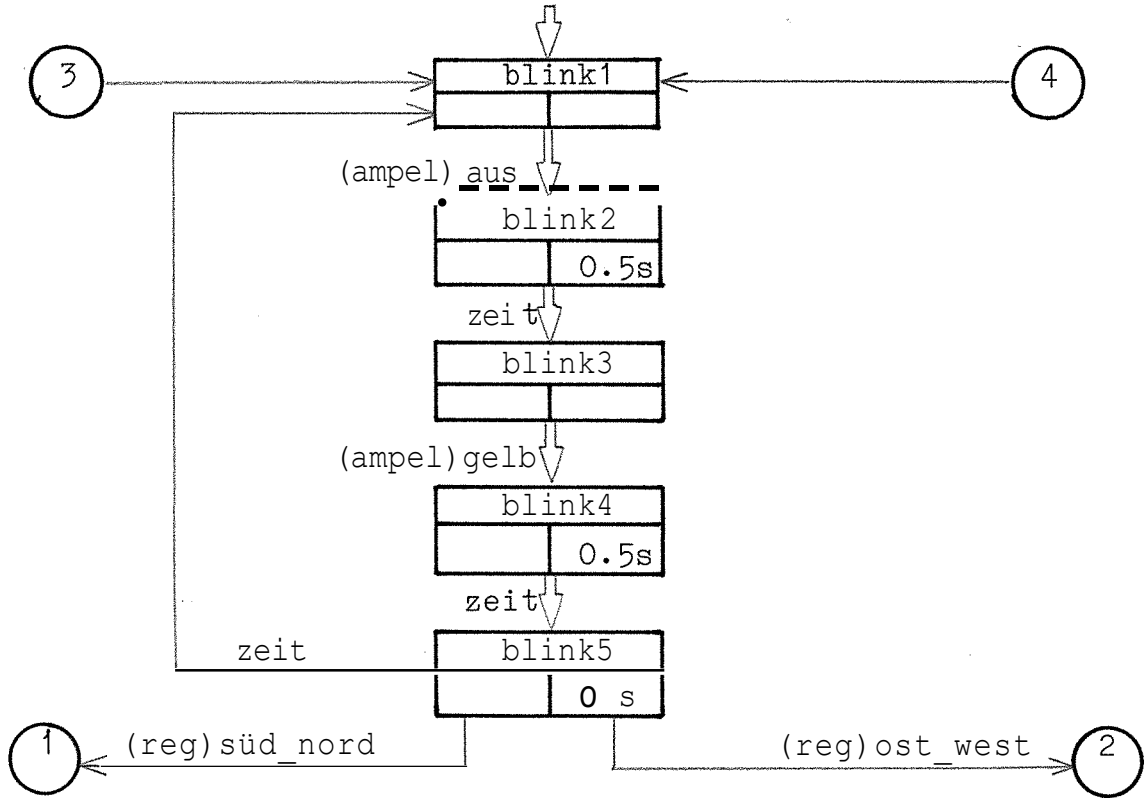
WARTEBEREICH: keiner

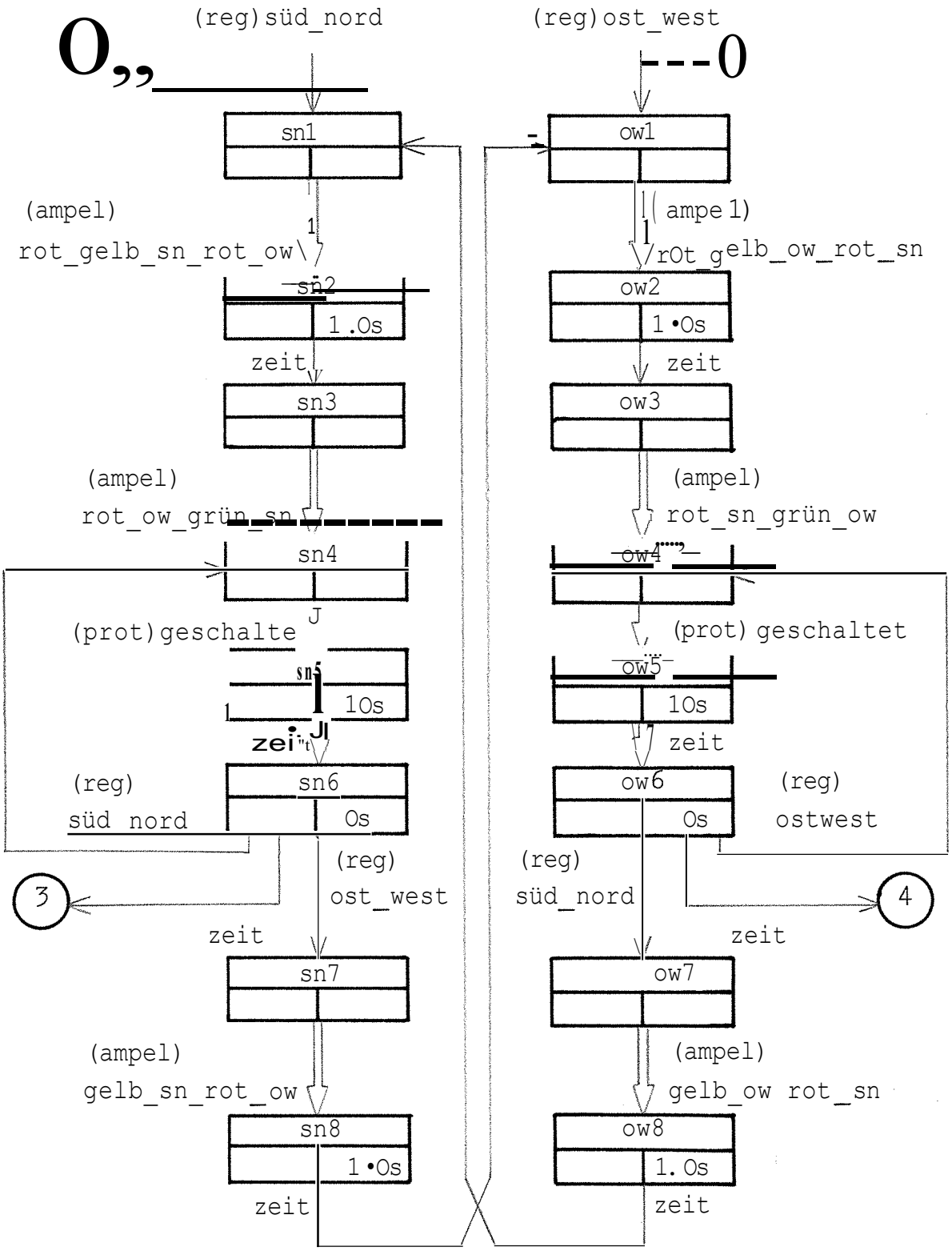
PROZESSZEIGERVARIABLE: keine

2. ABLAUFSTEUERUNG:

Die Ablaufsteuerung des Einzelprozesses 'steu' paßt nicht auf eine Seite. Deshalb wird ein zusätzliches graphisches Symbol für die Darstellung der Ablaufsteuerung eingeführt. Kanten können statt an Steuerzuständen auch an Verbindungsstellen beginnen bzw. enden. Verbindungsstellen werden durch Kreise dargestellt, die mit einer Nummer beschriftet sind.

Kanten, die in einem Kreis mit derselben Nummer beginnen bzw. enden, gehören miteinander verbunden (Verbindungsstellen) d.h. wird eine Ablaufsteuerung über mehrere Seiten dargestellt, darf es nur jeweils zwei Verbindungsstellen mit derselben Nummer geben. Zu einer führt eine Kante hin und von der anderen führt eine Kante weg.




(reg)süd_nord
(reg)ost_west
sn1
ow1
(ampel)
rot_gelb_sn_rot_ow
ampe 1)
rOt_gelb_ow_rot_sn
sn2
1.Os
ow2
1•Os
zeit
sn3
ow3
(ampel)
rot_ow_grün_sn
(ampel)
rot_sn_grün_ow
sn4
ow4
(prot)geschalte
(prot) geschaltet
sn5
10s
ow5
10s
zeit
(reg)
süd nord
sn6
Os
ow6
Os
(reg)
ostwest
(reg)
ost_west
(reg)
süd_nord
3
4
zeit
zeit
sn7
ow7
(ampel)
gelb_sn_rot_ow
(ampel)
gelb_ow rot_sn
sn8
1•Os
ow8
1. Os
zeit
zeit

3. BENUTZERMASCHINE:

3.1. AUSGABEFUNKTIONEN: gelb,
aus,
rot_gelb_sn_rot_ow,
rot_ow_grün_sn,
rot_gelb_ow_rot_sn,
rot_sn_grün_ow,
gelb_sn_rot_ow,
gelb_ow_rot_sn
geschaltet

3.2. EINGABEOPERATIONEN: süd_nord, ost_west

3.3. INTERNE OPERATIONEN:

3.4. INTERNE FUNKTIONEN:

3.5. DEFINITION DER BENUTZERMASCHINE:

zu den Eingabeoperationen

alle Botschaften sind ohne Botschaftsparameter, und allen empfangenen Nachrichten ist eine leere Eingabe zugeordnet

zu den Ausgabefunktionen

alle gesendeten Botschaften haben keine Nachrichtenparameter, deshalb ist ihnen allen die leere Ausgabefunktion zugeordnet.

Es gibt keine internen Funktionen und Operationen

PROSESSEND steu

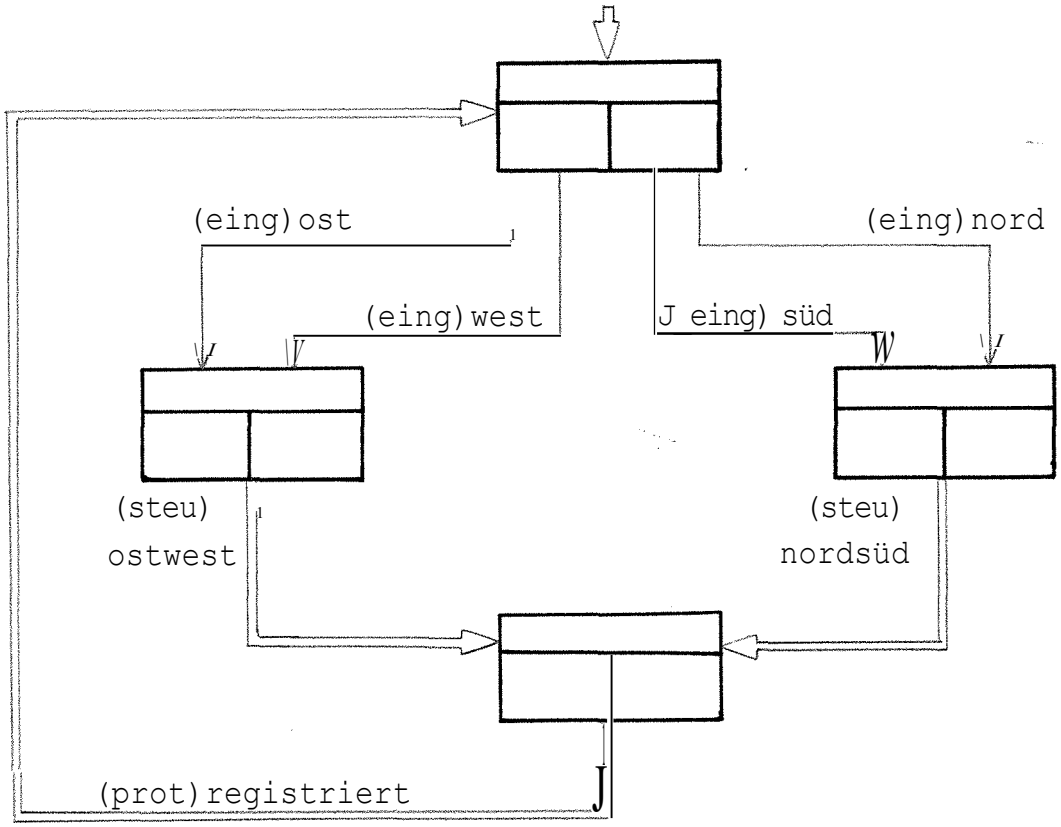
PROSESS: reg

1. KOMMUNIKATIONSMASCHINE:

WARTEBEREICH: keiner

PROZESSZEIGERVARIABLE: keine

2. ABLAUFSTEUERUNG:



BENUTZERMASCHINE

3. BENUTZERMASCHINE:

3.1. AUSGABEFUNKTIONEN: ostwest, nordsüd, registriert

3.2. EINGABEOPERATIONEN: ost, west, süd, nord

3.3. INTERNE OPERATIONEN:

3.4. INTERNE FUNKTIONEN:

3.5. DEFINITION DER BENUTZERMASCHINE:

zu den Eingabeoperationen

alle Botschaften sind ohne Botschaftsparameter, und allen empfangenen Nachrichten ist eine leere Eingabe zugeordnet.

zu den Ausgabefunktionen

alle gesendeten Botschaften haben keine Nachrichtenparameter, deshalb ist ihnen allen die leere Ausgabefunktion zugeordnet.

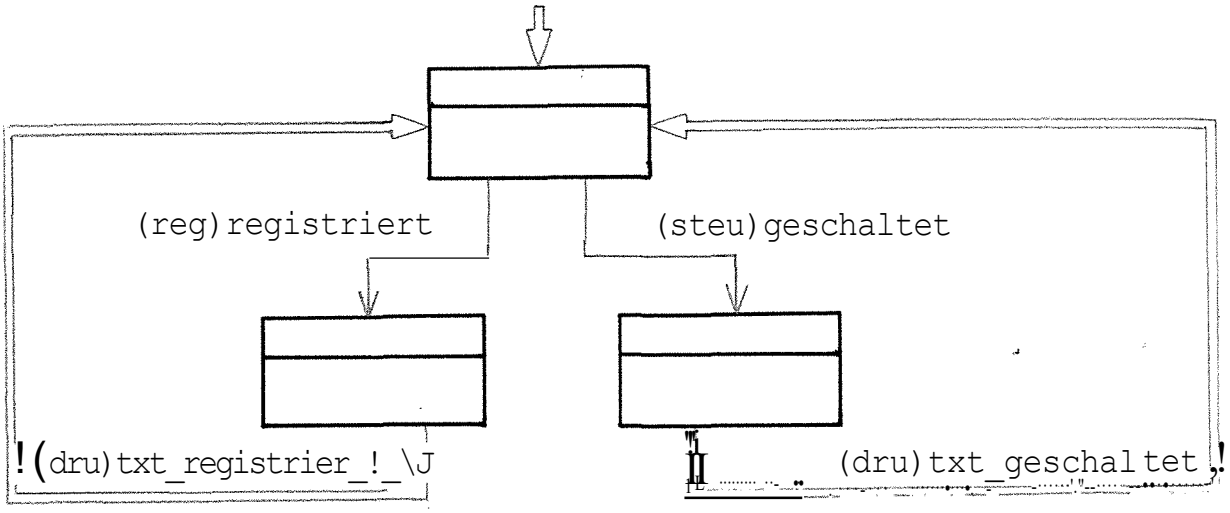
Es gibt keine internen Funktionen und Operationen.

PROZESSENDE: reg

PROZESS: prot

1. KOMMUNIKATIONSMASCHINE
   WARTEBEREICH: keiner
   PROZESSZEIGERVARIABLE: keine

2. ABLAUFSTEUERUNG:



3.1 BENUTZERMASCHINE

3. BENUTZERMASCHINE:

3.1. AUSGABEFUNKTIONEN: txt_registriert, txt_geschaltet

3.2. EINGABEOPERATIONEN: registriert, geschaltet

3.3. INTERNE OPERATIONEN:

3.4. INTERNE FUNKTIONEN:

3.5. DEFINITION DER BENUTZERMASCHINE:

txt_registriert:
Auf dem angegebenen Gerät wird der Text 'Auto registriert' ausgegeben.

txt_geschaltet:
Auf dem angegebenen Gerät wird der Text 'Ampel geschaltet' ausgegeben.

Zu den Eingabeoperationen
Alle Botschaften sind ohne Botschaftsparameter und allen

empfangenen Nachrichten ist eine leere Eingabe zugeordnet

zu den Ausgabefunktionen
alle gesendeten Botschaften haben keine Nachrichtenparameter, deshalb ist ihnen allen die leere Ausgabefunktion zugeordnet.

Es gibt keine internen Funktionen und Operationen.

PROZESSENDE: prot.

6.3 Implementation

Da jeder dieser Einzelprozesse auf einem eigenen Prozessor laufen soll, wurden sie bei der Implementierung in verschiedenen PEARL-Moduln untergebracht.
Die Implementierung der Einzelprozesse 'reg' und 'prot' ist selbstdokumentierend. Welche Anweisungsfolgen welchen Zuständen in der Ablaufsteuerung entsprechen ist, an den entsprechenden Empfangs- und Sendeanweisungen im Programmtext leicht zu erkennen. Bei der Implementierung des Einzelprozesses 'steu' wurden die Zustandsnamen in der Ablaufsteuerung vor den entsprechenden Anweisungsfolgen im Programmtext als Kommentar eingefügt.

```
MODULE( AMPEL);
SYSTEM;

AMPEL: <- ECBOIGC3)*2*1,12;
PROT  : <-CPU*  2;
REG   : ->CPU*  1;
PROBLEM;

SPC  < PROT,REG)  TASK;
SPC  AMPEL  DVC  SINK;


STEU:  TASK GLOBAL
       TRANSMITS GESCHALTET TO PROT
       RECEIVES ( osn-vEST' SUEDNORO ),FROM REG
       NOBUFFER;

/* KONSTANTEN ZUM ZUSAMMENSTELLEN DER KENNUNGEN
   FUER DIE EINZELNEN NACHRICHTEN AN DIE KOMPONENTE
   •AMPEL'  DES TECHNISCHEN PROZESSES                    */
DCL NIX    INV BIT(3)  INITC  'OOO·•BI);
DCL GELB   INV BIT(3)  INITC  '010181);
DCL ROT    INV BITC3 )  INIT(  '001'81);
DCL GRUEN  INV BIT(3)  INITC  '100•81);
DCL BUNT   INV BITC3)  INIT(  '011'81U

DCL PHASE BIT  ( 12);
DCL BLINKDAUER  INV DUR  INIT(500  MSEC);
DCL GELBDAUER  INV DUR  INIT(2  SEC>;
DCL ROTGRUENDAUER  INV DUR  INIT<7  SEC);


BLINK:  REPEAT
           /*  ZUSTAND  18LINK11  */
           /* NACHRICHT 1ÄUS1AUf8AUEN *  /
           PHASE:= NIX CAT NIX CAT NIX CAT.Nrx;
           /* UND SENDEN*  /
           SEND FROM PHASE TO AMPEL;
           /* ZUSTAND '8LINK2t  * /
           AFTER BLlNKDAUER RESUME;
           /* ZUSTAND '8LINK31  */
           PHASE:= GELB CAT GELB CAT GELB CAT GELB;
           SEND FROM PHASE TO AMPEL;
           /* ZUSTAND 'BLINK4t  */
           AFTER BLINKDAUER RESUME;

           /* ZUSTAND 'BLINKS•  */
           GUARDED COMMAND
             GUARO  RECEIVE FROM REG TO SUEDNORO;
               REACT  GOTO SUENO;
             GUARD  RECEIVE FROM REG TO OSTWEST;
               REACT  GOTO OSTWE;
           GUARDENO;

        END;
```

```
          /* ZUSTAND 'SNl•  *I
SUENO:    PHASE:= ROT CAT BlJNT CAT ROT CAT BUNT;
          SENO FROM PHASE TO AMPEL;
          j* ZUSTAN6 jSN2•  *I  ---
          AFTER GELBOAUER RESUME;
          /* ZUSTAND •SN3•  *I
          PHASE:= ROT CAT GRUEN CAT ROT CAT GRUEN;
          SEND FROM PHASE TO AMPEL;
          REPEAT
            /* ZUSTAND •SN4t  */
            TRANSMIT FROM GESCHALTET TO PROT;
            /* ZUSTAND •SNS•  *I
            AFTER ROTGRUENDAUER RESUME;

            /* ZUSTAND •SN6•  */
            GUAROED COMMAND
              GUARD  RECEIVE FROM REG TO OSTWEST;
                /* ZUSTAND tSN7' *  /
                REACT PHASE:= ROT CAT GELB CAT ROT CAT GELB;
                      SEND FROM PHASE TO AMPEL;
                      /* ZUSTAND •SN8•  *I
                      AFTER GELBDAUER RESUME;
                      GOTO OSTwE;
              GUARD  RECEIVE FROM REG TO SUEDNOPD;
                REt\CT
              OUTREACT GOTO BLINK;
            GUARDEND;

          END;
```

```
            /* ZUSTAND 9 0W1' */
OSTWE:      PHASE:= BUNT CAT ROT CAT BUNT CAT ROT;
            SEND FROM PHASE TO AMPEL;
            /* ZUSTAND •ow2• */
            AFTER GELBOAUER RESUME;
            /* ZUSTAND •OW3 1 *I
            PHASE:= GRUEN CAT ROT CAT GRUEN CAT ROT;
            SEND FROM PHASE TO AMPEL;
            /* ZUSTAND •OW4• *I
            REPEAT
              /* ZUSTAND 1 0W4 • */
              TRANSMIT FROM GESCHALTET TO PROT;
              /* ZUSTAND •OW5t */
              AFTER ROTGRUENDAUER RESUME;

             .I* ZUSTAND •OW6• */
              GUARDED COMMAND
                GUARD  RECEIVE FROM REG TO SUEDNORO;
                  /* ZUSTAND •OW7' */
                  REACT PHASE :: GELB CAT ROT CAT GELB CAT ROT;
                        SEND FROM PHASE TO AMPEL;
                        /* ZUSTAND 1 0W8 • */
                        AFTER GELBOAUER RESUME;
                        GOTO SUENO;
                GUARD  RECEIVE FROM REG TO OSTWEST;
                  REACT
                OUTREACT GOTO BLINK;
              GUARDEND;
            ENO;

END;   I* OF TASK STEU */

MODENO;
```

```
MODULE< ERFASS);
SYSTEM;
     WO:  -> ITR (15) ;
     OW:  ->  ITR(14H
     NS:  ->  ITR(13>;
     SN:  - > ITR (12) ;
     PROT:  <-CPU*  2;
     STFU:  <-CPU*  1;

PROBLEM;

   SPC  ( STEU,PROT)  TASK;
   SPC  (WO,OW,SN,NS)  INTERRUPT;

REG:  TASK GLOBAL
      TRANSMITS  ( OSTWEST,SUEONORO)  TO STEU,
                   REGISTRIERT  TO PROT;


      REPEAT

        GUARDED REGION
          GUARD  WHEN  W;
           REACT
                   TRANSMIT FROM OSTWEST  TO STEU;

          GUARD  WHEN OW;
           REACT
                   TRANSMIT FROM OSTWEST  TO STEU;

          GUARD  WHEN NS;
           REACT
                   TRANSMIT FROM SUEDNORO TO STEU;

          GUARD  WHEN SN;
           REACT
                   TRANSMIT FROM SUEDNORO TO STEU;
        GUARDENO;

        TRANSMIT FROM REGISTRIERT TO PROT;
      END;

END;  I* OF TASK REG*/

MODEN□;
```

```
.MODULE  ( PROTOK );
 SYSTEM;

   REG:  -> CPU* 1;
   STEU: ->CPU*  2;

 PROBLEM;

   SPC  ( STEU,REG)  TASK;
   SPC  PRINT FILE;

 PROT:  TASK GLOBAL
        RECEIVES REGISTRIERT FROM REG,
                 GESCHALTET FROM STEU,
        NOBUFFER;

   DCL  PLUS   INV CHARC5)   INIT('+++++t);
   DCL  MINUS  INV CHAR(5)   INIT( ,_____,);
   DCL  REGI   INV CHAR(11)  INITC'REGISTRIERT•);
   DCL  GESCH  INV CHAR C10) INIT<•GESCHALTET•1;

     REPEAT

      GUARDEO REGION

        GUARD  RECEIVE FRON REG TO REGISTRIERT;
          REACT  PUT FROM  (PLUS,REGI)  TO
                 PRINT THRU  (LINE,A(S),A(11));

        GUARD  RECEIVE FROM STEU TO GESCHALTET;
          REACT  PUT FROM  (MINUS,GESCH)  TO
                 PRINT THRU  (LINE,A(5) ,AC10));

        GUARDENO;

     END;

  END;  I* TASK ENDE  PROT */

 MODENO:
```

Anhang

Gliederungsgerüst für eine Spezifikation

I. KOMMUNIKATIONSSTRUKTUR

/* graphische Darstellung der Kommunikationsstruktur wie in Abschnitt 5.2.7 beschrieben */

II. TYPEN VON STRUKTURIERUNGSEINHEITEN

typname EINZELPROZESSTYP: (p1,p2, ... ,pn)
1. KOMMUNIKATIONSMASCHINE:
PUFFER: zahl
PROZESSZEIGERVARIABLE: v1 ,v2, ... vs
2. ABLAUFSTEUERUNG:
/* graphische Darstellung wie in Abschnitt 5.2.5.1 beschrieben*/
3. BENUTZERMASCHINE:
/* Bei den Punkten 3.1 bis 3.4 werden die Operationen und Funktionen der Benutzermaschine entsprechend ihrer Art aufgelistet.*/
3.1 AUSGABEFUNKTIONEN: af1, ... ,afm
3.2 EINGABEOPERATIONEN: eo1 , ... eon
3.3 INTERNE OPERATIONEN: io1, ... ios
3.4 INTERNE FUNKTIONEN: if1, ... ifr
3.5 DEFINITION DER BENUTZERMASCHINE:
/* in einer vom Entwerfer zu wählenden Technik*/

typname EINZELPROZESSTYP:
.
.
.
prozeßbündeltypname PROZESSBÜNDELTYP:
A. PROZESSE DES PROZESSBÜNDELTYPS
typname1 PROZESS (p1, ... pm)
0. PRIORITÄT: zahl
1. KOMMUNIKATIONSMASCHINE:
PUFFER: zahl
PROZESSZEIGERVARIABLE: v1 , ... vk

```
   2. ABLAUFSTEUERUNG:
      /* siehe Einzelprozeßtypen */
   3. PRIVATE BENUTZERMASCHINE:
      /*Beiden Punkten 3.1 bis 3.4 werden die Operationen
        und Funktionen der Benutzermaschine entsprechend
        ihrer Art aufgelistet.*/
      3.1 AUSGABEFUNKTIONEN: af1, ..... ,afm
      3.2 EINGABEOPERATIONEN: eo1 , ..... eon
      3.3 INTERNE OPERATIONEN: io1, ..... ios
      3.4 INTERNE FUNKTIONEN: if1, ..... ifr
      3.5 DEFINITION DER BENUTZERMASCHINE:
        /* in einer vom Entwerfer zu wählenden Technik*/
      .
      .
   typname2 PROZESS: (p1, ...pr)
      .
      .
B. GEMINSAME BENUTZERMASCHINE:
   1. INTERNE OPERATIONEN: io1, ..... ios
   2. INTERNE FUNKTIONEN: if1, ..... ifr
   3. DEFINITION DER BENUTZERMASCHINE:
      /* in einer vom Entwerfer zu wählenden Technik*/

prozeßbündeltypname PROZESSBÜNDELTYP:
 .
 .
 .
typname PROZESSGRUPPENTYP: (p1, ...pn)
A. PROZESSE:
   PROZESS:
   0. PRIORITÄT: zahl
   1. KOMMUNIKATIONSMASCHINE:
      PUFFER: zahl
      PROZESSZEIGERVARIABLE: v1, ...vk
   2. ABLAUFSTEUERUNG:
      /* siehe Einzelprozeßtypen */
   3. PRIVATE BENUTZERMASCHINE:
      /*Beiden Punkten 3.1 bis 3.4 werden die Operationen
        und Funktionen der Benutzermaschine entsprechend
        ihrer Art aufgelistet.*/
```

3.1 AUSGABEFUNKTIONEN: af1, ... ,afm
3.2 EINGABEOPERATIONEN: eo1 , ... eon
3.3 INTERNE OPERATIONEN: io1, ... ios
3.4 INTERNE FUNKTIONEN: if1, ... ifr
3.5 DEFINITION DER BENUTZERMASCHINE:
/* in einer vom Entwerfer zu wählenden Technik*/
PROZESS:
.
.

B. GEMINSAME BENUTZERMASCHINE
1. INTERNE OPERATIONEN: io1, ... ios
2. INTERNE FUNKTIONEN: if1, ... ifr
3. DEFINITION DER BENUTZERMASCHINE:
/* in einer vom Entwerfer zu wählenden Technik*/

typname PROZESSGRUPPENTYP: (p1, ... pm)
.
.

III. ELEMENTE EINES PROZESSYSTEMS:
1. DEKLARATIONEN:

name EINZELPROZESS (k1, ... kn) TYP typname (p1, ... pn)

prozeßbündeltypname PROZESSBÜNDEL:
name PROZESS (k1, ... km) TYP typname (p1, ... pm)
.
.
/* Jeder Prozeß eines Prozeßbündeltyps muß in der Deklaration eines entsprechenden Prozeßbündels vorkommen
*/
.

name PROZESSGRUPPE (k1, ... kl) TYP typname (p1, ... pl)

2. DEFINITION WEITERER STRUKTURIERUNGSEINHEITEN
name EINZELPROZESS
1. KOMMUNIKATIONSMASCHINE:
PUFFER: zahl
PROZESSZEIGERVARIABLE: v1 ,v2, ...vs
2. ABLAUFSTEUERUNG:
/* graphische Darstellung wie in Abschnitt 5.2.5.1 beschrieben*/
3. BENUTZERMASCHINE:
/*Beiden Punkten 3.1 bis 3.4 werden die Operationen und Funktionen der Benutzermaschine entsprechend ihrer Art aufgelistet. */
3.1 AUSGABEFUNKTIONEN: af1, ... ,afm
3.2 EINGABEOPERATIONEN: eo1, ...eon
3.3 INTERNE OPERATIONEN: io1, ...ios
3.4 INTERNE FUNKTIONEN: if1, ...ifr
3.5 DEFINITION DER BENUTZERMASCHINE:
/* in einer vom Entwerfer zu wählenden Technik*/

PROZESSBÜNDEL:
A. PROZESSE:
name PROZESS:
1. KOMMUNIKATIONSMASCHINE:
PUFFER: zahl
PROZESSZEIGERVARIABLE: v1 ,v2, ...vs
2. ABLAUFSTEUERUNG:
/* graphische Darstellung wie in Abschnitt 5.2.5.1 beschrieben*/
3. BENUTZERMASCHINE:
/*Beiden Punkten 3.1 bis 3.4 werden die Operationen und Funktionen der Benutzermaschine entsprechend ihrer Art aufgelistet. */
3.1 AUSGABEFUNKTIONEN: af1, ... ,afm
3.2 EINGABEOPERATIONEN: eo1 ,...... eon
3.3 INTERNE OPERATIONEN: io1, ...ios
3.4 INTERNE FUNKTIONEN: if1, ...ifr
3.5 DEFINITION DER BENUTZERMASCHINE:
/* in einer vom Entwerfer zu wählenden Technik*/

B. GEMEINSAME BENUTZERMASCHINE:
   1. INTERNE OPERATIONEN: io1, ... ios
   2. INTERNE FUNKTIONEN: if1, ... ifr
   3. DEFINITION DER BENUTZERMASCHINE:
      /* in einer vom Entwerfer zu wählenden Technik*/

name PROZESSGRUPPE:
A. PROZESSE:
   PROZESS:
   1. KOMMUNIKATIONSMASCHINE:
      PUFFER: zahl
      PROZESSZEIGERVARIABLE: v1 ,v2, ...vs
   2. ABLAUFSTEUERUNG:
      /* graphische Darstellung wie in Abschnitt 5.2.5.1 beschrieben*/
   3. BENUTZERMASCHINE:
      /*Beiden Punkten 3.1 bis 3.4 werden die Operationen und Funktionen der Benutzermaschine entsprechend ihrer Art aufgelistet.*/
   3.1 AUSGABEFUNKTIONEN: af1, ... ,afm
   3.2 EINGABEOPERATIONEN: eo1 ,..... eon
   3.3 INTERNE OPERATIONEN: io1, ... ios
   3.4 INTERNE FUNKTIONEN: if1, ... ifr
   3.5 DEFINITION DER BENUTZERMASCHINE:
      /* in einer vom Entwerfer zu wählenden Technik*/
   PROZESS:
      .
      .
B. GEMEINSAME BENUTZERMASCHINE:
   1. INTERNE OPERATIONEN: io1, ... ios
   2. INTERNE FUNKTIONEN: if1, ... ifr
   3. DEFINITION DER BENUTZERMASCHINE:
      /* in einer vom Entwerfer zu wählenden Technik*/

.
.
.

Literaturverzeichnis

/ADA79/ J. D. Ichbiah et. al.
Preliminary ADA Reference Manual
ACM Sigplan Notices, Juni 1979

/AMM81/ M. Ammann
PEARL für verteilte Systeme
Informatik Fachberichte 39
Fachtagung Prozeßrechner 1981
Springer Verlag, Berlin 1981

/ALWE77/ W. Altmann, D. Weber
Programmiermethodik
Arbeitsberichte des IMMD der Universität Erlangen
März 1977

/BAW081/ F. L. Bauer, H. Wössner
Algorithmische Sprache und Programmentwicklung
Springer Verlag, Berlin 1981

/BEHF82/ R. Besold, P. Holleczek, A. Fleischmann
Ein Programm zur Meßwerterfassung bei einem einfachen kernphysikalischen Experiment
nicht veröffentlichte Notizen, 1982,
Physikalisches Institut III der Universtät Erlangen

/BOCH78/ G. Bochmann
Architecture of Distributed Computer Systems
Lecture Notes in Computer Science,
Springer Verlag, Berlin 1979

/BOEM82/ F. Böml
Untersuchung der Elektronenemission von Festkörperoberflächen nach Wechselwirkung mit schnellen Ionen Bau einer Ultraviolett-Gasentladungslampe und Erstellung eines Programmsystems für Ultraviolett-Photoelektronen Spektroskopie
Diplomarbeit am Physikalischen Institut II der Universität Erlangen, 1982

/BOPS81/ J. Bos, R. Plasmeijer, J. Stroet
Process Communication Based on Input Specifications
ACM Transactions on Programming Languages and Systems, Juli 1981

/BRJA82/ R. Brehm et. al.
PROKON - Ein universelles, frei parametrierbares PEARL-System
PEARL-Rundschau, Dezember 1982

/BRUE81/ P. Brück, J. Robra
Verfahren und Hilfsmittel für die Software-Entwicklung von Nebenstellenanlagen
TE-KA-DE Technische Mitteilungen, 1981

/CAMP74/ R. H. Campbell, A. H. Habermann
The Specification of Process Synchronisation by Path Expressions
Lecture Notes in Computer Science Vol. 16, S 89-102, Springer Verlag, Berlin 1974

/CARL78/ E. Carlson, M Smyly
Practical Problems in a Distributed Application
Proc. of the National Computer Conference, 1978

/CARR82/ C. Carrellei, R.J. Roche
CCITT Languages for SPC Switching Systems
IEEE Transactions on Communications, Juni 1982

/COOK80/ R. P. Cook
*MOD - A Language for Distributed Programming
IEEE Transactions on Software Engineering
November 1980

/DAVI81/ D. W. Davies et. al
Distributed Systems - Architecture and Implementation
Lecture Notes in Computer Science (105)
Springer Verlag, Berlin 1981

/DIJK68/ E. W. Dijkstra
Cooperating Sequential Processes
aus: Programming Languages, S 43-112,
Academic Press, New York 1968

/DIJK75/ E. W. Dijkstra
Guarded Commands, Nondeterminacy and Derivation of Programms
Communication of the ACM, August 1975

/DIJK76/ E. W. Dijkstra
A Discipline of Programming
Prentice Hall, Englewood Cliffs 1976

/DIN66253/Programmiersprache PEARL
DIN 66253, 1981

/FELD79/ J. A. Feldman
High Level Programming for Distributed Computing
Communications of the ACM, Juni 1979

/FHKK82/ A. Fleischmann, P. Holleczek, G. Klebes, R. Kummer
Ein Mechanismus zur Kommunikation für verteilte Systeme in PEARL
PEARL-Rundschau, Dezember 1982

/FHKK83/ A. Fleischmann, P. Holleczek, G. Klebes, R. Kummer
Synchronisation und Kommunikation verteilter Automatisierungsprogramme
Erscheint im August oder September 1983 in der Angewandten Informatik

/FLEI81/ A. Fleischmann
Ein Konsoltreiber für ein PEARL-Programmiersystem
nicht veröffentlichte Notizen, 1981
Physikalisches Institut III der Universität Erlangen

/GENT81/ W. M. Gentleman
Message Passing between Sequential Processes: The Reply Primitive and the Administrator Concept
Software-Practice and Experience,
Vol. 11, S 435-466, 1981

/GOEH81a/ P. Göhner
Spezifikation der Synchronisierung paralleler Rechenprozesse
Informatik-Fachberichte 39
Fachtagung Prozeßrechner 1981
Springer Verlag, Berlin 1981

/GOEH81b/ P. Göhner
Ingenieurgerechte Spezifikation der Synchronisierung paralleler Rechenprozesse
Dissertation, Universität Stuttgart,
Institut für Regelungstechnik und Prozeßautomatisierung, 1981

/GRIE81/ D. Gries
The Science of Programming
Springer Verlag, Berlin 1981

/HABE76/ A. N. Habermann et. al.
Modularization and Hierarchy in a Family of Operating Systems
Communications of the ACM, Mai 1976

/HANS73/ P. Brinch-Hansen
Operating System Principles
Prentice Hall, Englewood Cliffs, 1973

/HANS77/ P. Brinch-Hansen
The Architecture of Concurrent Programms
Prentice Hall, Englewood Cliffs 1977

/HANS78/ P. Brinch-Hansen
Distributed Processes: A Concurrent Programming Concept
Communications of the ACM, November 1978

/HART81/ I. Hartmann
Entwicklung eines Systembausteins für Rechnerkopplungszwecke an der CYBER 173 des RRZE
Diplomarbeit am IMMD IV der Universität Erlangen

/HARR82/ M. Harrer
Ein PEARL-Kommunikationsprogramm für den Einsatz von Sichtgeräten
PEARL-Rundschau, Dezember 1982

/HOAR74/ C. A. R. Hoare
Monitors: An Operating System Structuring Concept
Communications of the ACM, Oktober 1974

/HOAR78/ C. A. R. Hoare
Communicating Sequential Processes
Communications of the ACM, November 1978

/HOFM81/ F. Hofmann
mündliche Mitteilung

/JORD83/ R. Jordan
Erstellung und Vergleich eines Filetransferprotokolls für Kleinrechner in verschiedenen Programmiersprachen
Diplomarbeit am IMMD IV der Universität Erlangen

/KEED78/ J. L. Keedy
On Structuring Operating Systems with Monitors
The Australien Computer Journal, Februar 1978

/KEME79/ H. Kernen
Zur Programmierung verteilter Systeme
Informatik Fachberichte 22
Kommunikation in verteilten Systemen
Springer Verlag, Berlin 1979

/KELL76/ R. M. Keller
Formal Verification of Parallel Programs
Communications of the ACM, Juli 1976

/KERA82/ S. Keramidis
Eine Methode Zur Spezifikation und korrekten Implementierung von asynchronen Systemen
Arbeitsberichte des IMMD der Universität Erlangen
Juni 1982

/KERN81/ H. Kerner, G. Bruckner
Rechnernetzwerke; Systeme, Protokolle und das ISO Architekturmodell
Springer Verlag, Wien 1981

/KKST79/ Kimm et. al.
Einführung in Software-Engineering
de Gruyter Verlag, Berlin 1979

/KLEB83/ G. Klebes
Entwicklung eines botschaftsorientierten Synchronisationsverfahrens für parallele Prozesse und seine Realisierung im Rahmen eines Realzeitprogrammiersystems
Diplomarbeit am IMMD II der Universität Erlangen

/KRAG82/ G. Kragl
Entwicklung eines Sicherungsverfahrens für ein Filetransferprotokoll zwischen Kleinrechnern und der CYBER des RRZE
Diplomarbeit am IMMD IV der Universität Erlangen

/KUMM83/ R. Kummer
Entwicklung von Betriebssystembausteinen für Guarded Regions und Botschaftsoperationen ihm Rahmen eines Realzeitprogrammiersystems
Diplomarbeit am IMMD IV der Universität Erlangen

/LANE78/ H. C. Lauer, R. M. Needham
On the Duality of Operating System Structures
ACM Operating System Reviews, April 1979

/LEIN82/ M. Leinfelder
Ein graphisch-interaktives Hilfsmittel zur Erstellung von Transitionsdiagrammen
Diplomarbeit am immd IV der Universität Erlangen

/LAUB76/ R. Lauber
Prozeßautomatisierung I
Springer Verlag, Berlin 1976

/LAUB79/ R. Lauber
Modelle zur Beschreibung des Entwurfs von Prozeßautomatisierungssystemen
Regelungstechnik 1979, Heft 12

/LEVI81/ P. Levi
Betriebssysteme für Realzeitanwendungen
Datakontext Verlag, Köln 1981

/LISK77/ B. Liskov et. al.
Abstraction Mechanisms in CLU
Communications of the ACM, August 1977

/LISK79/ B. Liskov
Primitives for Distributed Computing
ACM Proc. of the 7th Symposium on Operating System Principles, 1979, S. 33-42

/LUDE81/ J. Ludewig
PCSL und ESPRESO - Zwei Ansätze zur Formalisierung der Prozeßrechnerspezifikation
Informatik Fachberichte Nr. 39
Fachtagung Prozeßrechner 1981
Springer Verlag, Berlin 1981

/MAYE80/ T. W. Mao, R. T. Yeh
Communication Port: A Language Concept for Concurrent Programming
IEEE Transactions of Software Engineering, März 1980

/MÜLL77/ H. Müller
Automatentheorie I
Vorlesung an der Universität Erlangen, WS 1977/78

/MUSS80/ D. R. Musser
Abstract Data Type Specification in the AFFIRM System
IEEE Transactions on Software Engineering
Januar 1980

/NEWT79/ G. Newton
Deadlock Prevention, Detection and Resolution: An Annotated Bibliography
Operating Systems Reviews,
Vol. 13, Nr. 2, S 33-44, 1979

/PIEP81/ F. Pieper
Concurrent Pascal - Eine Kritik
Elektronische Rechenanlagen, Heft 3, 1981

/PBBP82/ P. Pepper et. al.
Abstrakte Datentypen: Die algebraische Spezifikation von Rechenstrukturen
Informatik Spektrum, Juli 1982

/RICH77/ L. Richter
Betriebssyteme
Teubner Verlag, Stuttgart 1977

/ROBE81/ E. S. Roberts et. al.
Task Management in ADA - A Critical Evaluation for Real-Time Multiprocessors
Software-Practice and Experience,
Vol. 11, S 1019-1051, 1981

/ROSA82/ A. Rockström, R. Saracco
SDL - CCITT Specification and Description Language
IEEE Transactions on Communications, Juni 1982

/SILB79/ A. Silberschatz
Communication and Synchronisation in Distributed Systems
IEEE Transactions on Software Engineering,
November 1979

/SOMM82/ I. Sommerville
Software Engineering
Addison-Wesely Publishing Company
London 1982

/STAU82/ J. Staunstrup
Message Passing Communication Versus Procedure Call Communication
Software-Practice and Experience
Vol. 12, S 223-234, 1982

/STOT82/ P. P. Stotts
A Comperative Survey of r.nncurrent Programming languages
ACM Sigplan Notices, Oktober 1982

/STR080/ J. Stroet
An Alternative to the Communication Primitives in ADA
ACM Sigplan-Notices, Dezember 1980

/WEEE83/ K. Weber
private Mitteilung

/WELS79/ J. Welsh et. al.
A Comparison of two notations for Process Communication
Proceedings of the Symposium on Language Design and Programming Methodology
Sydney, September 1979

/WELS81/ J. Welsh, A. Lister
A Comperative Study of Task Communication in ADA
Software-Practice and Experience
Vol. 11, S 257-290, 1981

Bisher sind folgende Mitteilungsblätter des Regionalen Rechenzentrums Erlangen erschienen (Tiiel für 1-19: Mitteilungsblatt des Rechenzentrums der Universität Erlangen-Nürnberg):

1	-		Allgemeine Information; November 1968
2	-		Jahresbericht 1968; Februar 1969
3	-	J. Kettler:	Über den Umgang mit dem ALGOL-Compiler der CD3300 unter dem Betriebssystem MASTER; September 1969
4	-	J. Kettler:	Einführung in die ALGOL-Programmierung für die Rechenanlage CD3300; Januar 1970
5	-		Jahresbericht 1969; Februar 1970
6	-		Jahresbericht 1970; Februar 1971
7	-	H. Zemanek:	Zukunftsaspekte der Informationsverarbeitung; Oktober 1971
8	-		Allgemeine Informationen; Januar 1972
9	-	G. Seybold:	EDI - ein einfacher Dialoginterpreter, Diplomarbeit; Januar 1972
10	-		Jahresbericht 1971; Januar 1972
11	-	G. Görz:	LISP 1.5 - Handbuch für die CD3300; März 1972
12	-	J. Schönhut:	Graphemsystem und Wortkonstituenz - Dokumentation eines linguistischen Programmsystems; August 1972
13	-	C. Endres:	PUMA - Programmierter Unterricht unter dem Betriebssystem MASTER; März 1973
14	-		Jahresbericht 1972; März 1973
15	-	E. Seele/ F. Wolf	Darstellung thematischer Karten mit Schnelldrucker und Plotter auf der CD3300; April 1973
16	-		Jahresbericht 1973; Februar 1974
17	-	H. Lehr:	METEOR - Beschreibung eines interpretativen Systems unter LISP 1.5 für Transformationen über linearen Listen; März 1974
18	-		Jahresbericht 1974; Februar 1975
19	-		Jahresbericht 1975; Februar 1976
20	-		Jahresbericht 1976; Juni 1977
21	-		Jahresbericht 1977; April 1978
22	-	M. Abel:	Einführung in die Programmierung FORTRAN an den Rechenanlagen TR440 und CYBER; Oktober 1978
23	-	G. Görz/ G. Büttner	Computer Spiele; November 1978
24	-		Benutzerhandbuch; November 1978

25 - 10 Jahre Rechenzentrum; November 1978

26 - Jahresbericht 1978; Februar 1979

27 - F. Wolf: REGAL - Rechnergestützte Grundausbildung in ALGOL; März 1979

28 - Jahresbericht 1979; Juli 1980

29 - J. Schönhut: Erlanger Grafik System (EGS 1.5); Februar 1980

30 - Jahresbericht 1980; Mai 1981

31 - G. Bachbauer: Handbuch für die Anwendung statistischer Programmsammlungen; Oktober 1981

32 - Jahresbericht 1981; Februar 1982

33 - M. Abel: TV - Ein System zur Textgestaltung; Februar 1983

34 - Jahresbericht 1982; April 1983

35 - Jahresbericht 1982 über die Rechenanlagen der Medizinischen Fakultät; Juni 1983

36 - Benutzerhandbuch für die Rechenanlagen der Medizinischen Fakultät; Juni 1983

37 - J. Schönhut: Erlanger Grafik-System (EGS 2.0m); Juli 1983

38 - H. Henke: Die Benutzung der CYBER über die Erlanger Mikrorechnersysteme; August 1983

39 - Jahresbericht 1983; April 1984

40 - A. Fleischmann: Ein Konzept zur Darstellung und Realisierung von verteilten Prozeßautomatisierungssystemen; April 1984