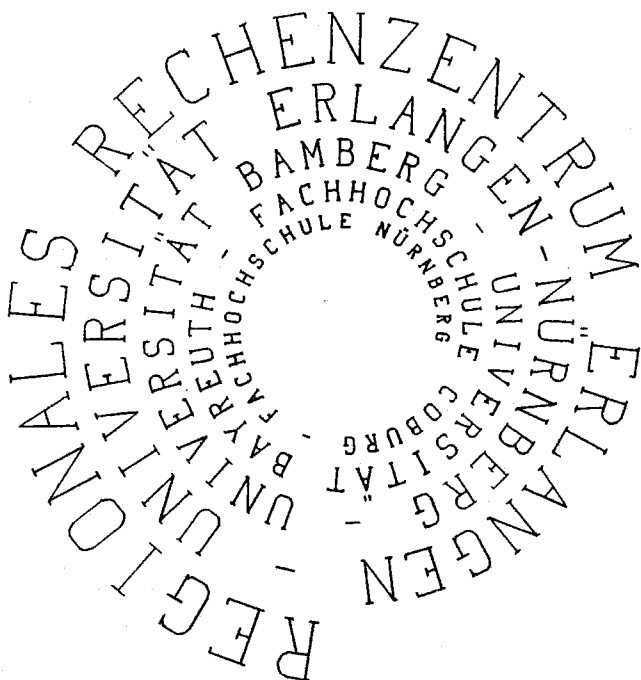


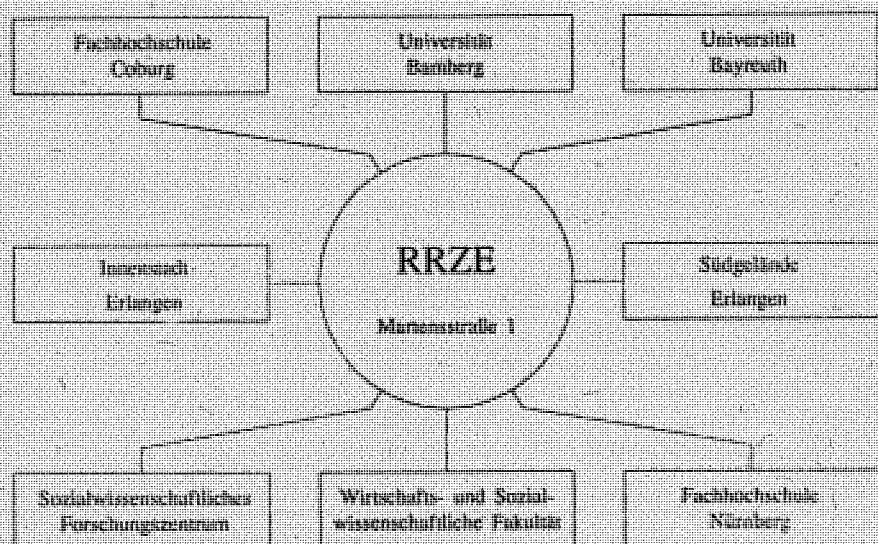
MITTEILUNGSBLATT

DES REGIONALEN
RECHENZENTRUMS ERLANGEN

HERAUSGEBER F. WOLF



Nr. 53 - ERLANGEN - JUNI 1989



Regionales Rechenzentrum Erlangen

Martensstraße 1
D-8520 Erlangen
Tel. 09131/85-7031

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

ISSN 0172-2905 (Mitteilungsblatt des Regionalen Rechenzentrums)
Redaktion dieser Ausgabe: H. Henke

MITTEILUNGSBLATT

DES REGIONALEN
RECHENZENTRUMS ERLANGEN

HERAUSGEBER F. WOLF

**Untersuchung von Sicherungsmaßnahmen
für verteilte Systeme
unter Verwendung
eines geeigneten Betriebssystemmodells**

Robert Kummer

Nr. 53 - ERLANGEN - JUNI 1989

Kummer, Robert:

Untersuchung von Sicherungsmaßnahmen für verteilte Systeme unter Verwendung eines geeigneten Betriebssystemmodells

In Prozeßsystemen mit miteinander kommunizierenden Prozessen, die auf mehrere lose gekoppelte Rechner verteilt sind, gibt es eine Vielzahl von möglichen Fehlerquellen. Angefangen von Übertragungsfehlern, über kurzzeitiges Aussetzen einzelner Prozessoren bis zum Zerstören von Programmdateien ist es eine weite Spanne. Betrachtet man insbesondere verteilte Systeme zur Steuerung technischer Prozesse, so ist noch mit einer Erweiterung der Fehlerursachen zu rechnen.

Es wird deshalb in dieser Arbeit der Versuch unternommen, ein Prozeßsystem vorzustellen, das gegenüber Fehlern eine gewisse Robustheit aufweist. Dabei wird festgestellt, daß die Problemstellung eine Aufteilung in drei Teilaufgabengebiete nahelegt: Die Erstellung eines Betriebssystemmodells, die Entwicklung eines Botschaftenalgorithmus und das geeignete Setzen von Rücksetzpunkten. Dabei ist es wichtig, daß die ausgewählten Lösungen aufeinander aufbauen.

So wird zunächst ein Betriebssystemmodell erläutert, das als logische Einheiten nur Prozesse aufweist, die über Botschaften miteinander kommunizieren, egal ob sie sich auf dem gleichen Rechner befinden oder nicht. Diese an die Spezifikationsmethode PASS (/FLEI84/) angelehnte Denkweise erlaubt eine einheitliche Behandlung aller am Prozeßsystem beteiligter Komponenten.

Als weiterer Baustein wird ein Botschaftenalgorithmus entwickelt, der insbesondere die Spracheigenschaften des Verteilten PEARL (/FHKK83/) erfüllt. Dabei ist der Grundgedanke der, daß für mehrere an einer Kommunikation beteiligten Prozesse ein zentraler Prozeß existiert, der den Nachrichtenaustausch koordiniert.

Schließlich werden Überlegungen angestellt, wie Rücksetzpunkte definiert werden können, so daß es zu keinem Dominoeffekt kommen kann. Hier spielt die Feststellung eine Rolle, daß nach erfolgreicher Beendigung einer Kommunikation kein Rücksetzen vor den Kommunikationsbeginn mehr möglich ist.

Durch das Aufeinanderaufsetzen der drei Lösungsschritte wird ein integriertes, gegenüber Fehlern robustes System erzielt. Außerdem wird untersucht, inwieweit sich dieses Gesamtsystem in das ISO/OSI-7-Schichten-Architekturmodell integrieren läßt.

Für die Überlassung der Arbeit möchte ich mich bei Herrn Prof. Dr. F. Hofmann herzlich bedanken.

Mein Dank gilt auch Herrn Prof. Dr. H. Wedekind für die Übernahme des Zweitgutachtens.

Ohne meinen hartnäckigen Betreuer, Herrn Dr. P. Holleczek, wäre dieses Werk wohl nie entstanden.

Schließlich gilt mein Dank auch der "Mafia" für die über 5 Jahre fruchtlose Zusammenarbeit und Jürgen für das mühevolle Anfertigen der Bilder.

Meiner Frau Angela gewidmet, die mich während der Jahre der Entstehung dieser Arbeit unterstützt hat.

Inhalt

1.	Einleitung	1
1.1.	Voraussetzungen und Zielsetzung	1
1.2.	Vergleich mit natürlichen Verwaltungsvorgängen	3
1.3.	Verwendete Spezifikationsmethode	5
2.	Einige beispielhafte Ansätze für ausfallsichere verteilte Systeme	7
2.1.	Software-Fehler-Toleranz in Realzeitsystemen	8
2.1.1.	Definition von Begriffen	8
2.1.2.	Erkennen und Beheben von Fehlern	10
2.1.3.	Ein Modell für Realzeitsysteme	12
2.1.4.	Kommentar	16
2.2.	Das auf Monitoren basierende "Conversation"-Schema	18
2.2.1.	Namensverbundener Recovery-Block	21
2.2.2.	Gesprächsmonitor	22
2.2.3.	Abstrakter Datentyp	23
2.2.4.	Kooperierender Recovery-Block	26
2.2.5.	Kommentar	28
2.3.	Verteilte Systeme für Rücksetzbarkeit und Zusammenbruchsicherheit	29
2.3.1.	Der "remote procedure call" als Mittel zum Nachrichtenaustausch	29
2.3.2.	Recovery-Konzept	31
2.3.3.	Maßnahmen gegen Ausfälle	33
2.3.4.	Kommentar	34
2.4.	Das "Three Phase Commit"-Protokoll	35
2.5.	Abschließende Bewertung der vorgestellten Ansätze	41

3.	Entwicklung eines Betriebssystemmodells	43
3.1.	Anforderungen an ein Betriebssystem	44
3.2.	Das konventionelle Schichtenmodell	46
3.3.	Das Betriebssystem als ein Prozeßmodell	49
3.3.1.	Abstraktes Modell	50
3.3.2.	Konkretes Modell	55
3.3.3.	Der Prozeßumschalter für die Benutzer- und Systemprozesse	60
3.3.4.	Implementiertes System	64
4.	Ein zentralistisches Modell für ein Betriebssystem-Botschaften-Protokoll	67
4.1.	Zu realisierendes Benutzerprogramm-Protokoll	69
4.2.	Das Baumverfahren	72
4.3.	Erweiterungen für die asynchrone Kommunikation	89
4.4.	Formale Beweisskizze für die Vollständigkeit des Regelwerks	95
5.	Sicherungsmaßnahmen in einem Prozeßmodell	100
5.1.	Begriffsklärung	101
5.2.	Klassen von behebbaren Fehlern	102
5.3.	Einführung von Zeitüberwachungsmaßnahmen	106
5.4.	Das Setzen von Sicherungspunkten in verteilten Systemen	115
5.5.	Zusammenhang zwischen dem Setzen von Sicherungspunkten und dem Baumverfahren	121
5.6.	Wahl der Sicherungsdaten	129
5.6.1.	Datensicherung bei Prozessen und Prozeßumschaltern	129
5.6.2.	Datensicherung bei "globalen Daten"	131
5.6.3.	Datensicherung bei Wartebereichen	133
5.7.	Reaktion nach Bemerkung eines Ausfalls	136
5.8.	Die Parallelen zum "Three Phase Commit"-Protokoll	139
5.9.	Aufwand für Sicherungs- und Rücksetzmaßnahmen	141
5.10.	Sicherungsmaßnahmen durch den Benutzer	143

6.	Erweiterungen des verteilten Systems für seine dynamische Konfigurierbarkeit und seine Einbettung in offene Systeme	146
6.1.	Rekonfigurieren eines verteilten Systems	147
6.1.1.	Definitionen	147
6.1.2.	Dynamisches Konfigurieren von Prozessen	149
6.2.	Einbettung des verteilten Systems in das ISO-7-Schichten-Modell	151
6.2.1.	Direkte Botschaftenübermittlung durch die Benutzerprozesse	152
6.2.2.	Einbettung der Prozeß-Prozeß-Kommunikation in das ISO/OSI-Referenzmodell	154
6.2.3.	Vergleich zwischen dem Baumverfahren und der ISO-Norm "Transaction Processing"	161
7.	Analyse und Diskussion des vorgeschlagenen Konzeptes	165
	Anhang 1	168
	Anhang 2	186
	Literaturverzeichnis	227

Verzeichnis der Abbildungen

Bild 2-1:	Beispiel eines Synchronisationsgraphen	13
Bild 2-2:	Kommunikationsrestriktionen, /ANKN83/	15
Bild 2-3:	Recovery-Block, /KIMM82/	19
Bild 2-4:	"Conversation", /KIMM82/	19
Bild 2-5:	Namensverbundener Recovery-Block, /KIMM82/	21
Bild 2-6:	Recovery-Block mit Gesprächsmonitor, /KIMM82/	22
Bild 2-7:	Gespräch mit abstraktem Datentyp, /KIMM82/	24
Bild 2-8:	Illegales Verschachteln von Gesprächen, /KIMM82/	25
Bild 2-9:	Restriktive Version des CRB, /KIMM82/	26
Bild 2-10:	Concurrent Recovery-Block in Concurrent-PASCAL-Umgebung, /KIMM82/	27
Bild 2-11:	2 mögliche "Two Pase Commit"-Szenarien, /GRAY79/,/CERI84/	37
Bild 2-12:	Erfolgreiches "Three Pase Commit"-Szenarium, /BEHG87/	39
Bild 3-1:	Das Betriebssystem als konventionelles Schichtenmodell	47
Bild 3-2:	Das Betriebssystem, bestehend aus Monitoren	48
Bild 3-3:	Konsolenausgabebeftrag	52
Bild 3-4:	Kommunikationsstruktur des "Abstrakten Modells"	53
Bild 3-5:	Schichtung des "Konkreten Modells"	57
Bild 3-6:	Schalenmodell des "Konkreten Modells"	59
Bild 3-7:	Beispiel für die Arbeitsweise des Prozeß-PU	63
Bild 3-8:	Treiberrahmen mit "eingehängten" Treibern	66
Bild 4-1:	Realisierung eines Protokolls durch ein "unterlagertes" Protokoll	69
Bild 4-2:	Kommunikationsstruktur für die drei Beispielprozesse	74
Bild 5-1:	Lose gekoppeltes System	102
Bild 5-2:	Übergangsautomat für das Baumverfahren und die Auszeiteinplanungen	114
Bild 5-3:	Fehler bei Prozeß B im sequentiellen Programmteil	117
Bild 5-4:	Fehler bei Prozeß B im parallelen Programmteil	117
Bild 5-5:	Phasen eines Prozesses aus <u>Benutzersicht</u> beim Nachrichtenaustausch	121
Bild 5-6:	Phasen eines Prozesses aus <u>Betriebssystemsicht</u> beim Botschaftenaustausch	122

Bild 5-7:	Abbildung der Protokollphasen aus der Benutzersicht auf die aus der Betriebssystemsicht	122
Bild 5-8:	Protokollphasen eines Prozesses mit allen Rücksetzpunkten	123
Bild 5-9:	Protokollphasen eines Prozesses mit den nicht redundanten Rücksetzpunkten	124
Bild 5-10:	Protokollphasen der Prozesse aus der Betriebssystemsicht mit Setzen von Sicherungspunkten	126
Bild 5-11:	Ergänzter Übergangsautomat aus Bild 5-2	128
Bild 5-12:	Ersetzen der globalen Daten durch einen V-Prozeß mit lokalen Daten a) indirekte Kommunikation über globale Daten b) direkte Kommunikation über V-Prozeß	132
Bild 5-13:	Ein/Ausschalten von Betriebssystem-Rücksetzpunkten und -Rücksetzen	143
Bild 5-14:	Setzen von Rücksetzpunkten durch den Benutzer	144
Bild 5-15:	Setzen von Rücksetzpunkten mit Namensliste	145
Bild 6-1:	Direkte Botschaftenübermittlung durch Prozesse	151
Bild 6-2:	Alle vorhandenen Prozesse sind direkt miteinander verbunden	153
Bild 6-3:	Alle vorhandenen Prozesse sind nicht direkt miteinander verbunden	153
Bild 6-4:	Das ISO/OSI-7-Schichten-Architekturmodell, /KEBR81/	154
Bild 6-5:	Prozeß-Prozeß-Kommunikation über ein lokales Netz	155
Bild 6-6:	Zusammenhang zwischen Sitzungen und Transportverbindungen	156
Bild 6-7:	Architekturbruch, /BAAC87/	158
Bild 6-8:	Aufrufhierarchie beim Nachrichtenaustausch aus der Sicht eines Prozesses	159
Bild 6-9:	Nachrichtenaustausch mit Hilfe standardisierter Protokolle aus der Sicht eines Prozesses	160
Bild 6-10:	Struktur der Schicht 7, bestehend aus Anwendungseinheiten, /ISOTP1/	162
Bild 6-11:	Die Anwendungseinheit als Prozeßbündel, /BEVE87/	162
Tabelle 3-1:	Prozeß-Tabelle des Prozeß-PU	61

1. Einleitung

Die Erfahrungen mit verteilten Systemen (siehe Kapitel 2 und /HEIL85/, /HERB85/) haben gezeigt, daß es notwendig und auch möglich ist, transparente verteilte Systeme zu entwickeln, die gegenüber Ausfällen eine gewisse Sicherheit aufweisen. In dieser Arbeit soll ein derartiges System vorgestellt werden. Nachfolgend wird der Rahmen und die Aufgabenstellung dafür abgesteckt.

1.1. Voraussetzungen und Zielsetzung

In verteilten Systemen, insbesondere in verteilten Prozeßsystemen zur Steuerung technischer Prozesse, ist die Problematik von Ausfällen von Komponenten oder Teilsystemen besonders akut. Da ein möglichst durchgehender Service des Systems gewährleistet werden muß, sollte ein Betriebs- und Laufzeitsystem zur Verfügung gestellt werden, das ausreichend sicher und im Fehlerfall fähig ist, geeignete Maßnahmen zu ergreifen.

Ähnliche Versuche, ausfallsichere Systeme zu entwickeln, wurden bereits unternommen. Am besten entwickelt sind Rücksetz- (Recovery-) Verfahren in Datenbanksystemen, so z.B. das "Three Phase Commit"-Protokoll (/BEHG87/). Hierbei handelt es sich um ein zentrales System, in dem ein einziger Prozeß für die Konsistenz der Datenbestände verantwortlich ist. Im folgenden werden allerdings Systeme betrachtet, die unter Umständen auf mehreren eigenständigen Rechnern verteilt sind, wobei letztere über Leitungen miteinander verbunden (lose gekoppelt) sind. In Kapitel 2 werden derartige Systeme vorgestellt.

Am Physikalischen Institut und am Regionalen Rechenzentrum der Universität Erlangen wurde im Rahmen zweier von der DFG geförderter Projekte ein Programmiersystem für verteilte Systeme entwickelt. Es wurden unter anderem eine Spezifikationsmethode (/FLEI84/), eine Erweiterung der Programmiersprache PEARL (Verteiltes PEARL, /FHKK83/), ein Betriebssystem (/KUMM83/) und ein Test- und Bediensystem (/AFHT85/) erstellt.

Aus diesen Implementationserfahrungen heraus und aus den Erfahrungen bei der Entwicklung darauf basierender Anwendungsprogramme, z.B. eines verteilten Systems zur Erfassung von Telefongesprächsdaten (/HEIL85/), wurde deutlich, daß eine Weiterentwicklung vom Verteilten Pearl zu einem zumindest teilweise "ausfallsicheren Verteilten Pearl" notwendig ist. Um ein ausfallsicheres Betriebssystem zu entwickeln, wird aber ein anderer Weg als in /KUMM83/ beschritten. Ausgehend von einem auf eigenständigen Prozessen basierenden Betriebssystemmodell (Kapitel 3), soll ein Botschaftenprotokoll entwickelt werden (Kapitel 4), das in der Lage ist, das Setzen von Rücksetzpunkten zu ermöglichen (Kapitel 5).

Folgender Grundgedanke ist dabei maßgebend: Da es sich um verteilte Systeme handelt, soll das Betriebssystem ohne gemeinsame Daten entworfen werden. Jede Komponente des Betriebssystems eines jeden Rechners soll so eigenständig wie nur möglich sein. Die Komponenten werden also zu Prozessen verselbständigt. Das in /KUMM83/ entwickelte Botschaftensystem wurde als zu undurchsichtig empfunden. Es wird deshalb ein dem "Three Phase Commit"-Protokoll ähnliches Verfahren entwickelt, das sogenannte "Baumverfahren". Dadurch ist eine leichte Unterteilung des Protokolls in Phasen (siehe Kapitel 4) möglich. Außerdem hat die Unterteilung des Protokolls den zusätzlichen Vorteil, daß das Setzen von Rücksetzpunkten leicht einfügbar ist und damit ein integraler Bestandteil des ganzen Verfahrens wird (siehe Kapitel 5). So ist zumindest das erste Problem bei Rücksetzverfahren, das Setzen von Rücksetzpunkten, vollständig gelöst. Auch das zweite Problem, das Erkennen von Ausfällen und deren Behebung durch Rücksetzen, soll ebenfalls in Kapitel 5 diskutiert werden.

Kapitel 6.1. geht auf Erweiterungsmöglichkeiten des Systems bezüglich des dynamischen Konfigurierens ein.

Um nicht nur innerhalb der Grenzen eines abgeschlossenen Systems zu verbleiben, soll in Kapitel 6.2. auch der Versuch unternommen werden, dieses verteilte System in die Welt der offenen Systeme einzubetten. Dazu wird das OSI/ISO-7-Schichten-Modell herangezogen.

Wie bereits erwähnt, versteht sich diese Arbeit als eine Fortführung der oben genannten Projekte. Aus diesem Grund spielen beim Entwurf des Modells und der Sicherungsmaßnahmen die verwendete Spezifikationsmethode PASS (Parallel Activities Specification Scheme, /FLEI84/) und die Entwicklung von Verteiltem PEARL (/FHKK83/) eine wesentliche Rolle. Dieser Umstand wird in Abschnitt 1.3. näher betrachtet. Trotz des großen Einflusses der Spezifikationsmethode auf diese Arbeit, findet sie nur unter bestimmten Aspekten Verwendung, was zu Einschränkungen und Erweiterungen der Methode führt.

Zunächst aber soll eine weitere allgemeine Motivation für diese Arbeit im nachfolgenden Abschnitt gegeben werden.

1.2. Vergleich mit natürlichen Verwaltungsvorgängen

Um im Zusammenhang mit dem nachfolgend vorgestellten Prozeß- system die Vorgehensweise beim Systementwurf plausibler zu machen, soll der Versuch unternommen werden, einen Vergleich zwischen einfachen Verwaltungsvorgängen in unserer natürlichen Umwelt und in der Betriebssystemumgebung verteilter Systeme anzustellen. Als Grundlage dieses exemplarischen Vergleichs dient folgende Tabelle:

Begriffe aus	
der natürlichen Welt	den verteilten Betriebssystemen
- Juristische Person	Benutzerprozeß
- Verwaltungsstelle, wie z.B.Kfz-Zulassungsstelle, Einwohnermeldeamt etc.	Verwaltungsprozeß für Semaphore, Gerät, etc.
- Datenschutz	Datenschutz
- Unglück bei einer Person oder Behörde	Fehler bei einem Prozeß
- Regenerieren einer Person oder Behörde	Rücksetzen eines Prozesses
- Medien, wie Post, Telefon oder mündliche Kommunikation	Botschaftenprotokoll
- Umzug einer Verwaltung in ein neues Dienstgebäude oder einer Person in eine neue Wohnung	Umkonfigurieren von Prozessen

Während die Begriffe aus dem täglichen Leben selbsterklärend sind, sei bei den Begriffen in der rechten Spalte auf einschlägige Literatur oder auf die nachfolgenden Kapitel verwiesen. Es soll hier kein inhaltlicher Vergleich durchgeführt werden - es läßt sich kaum ein Semaphore auf eine Kfz-Zulassungsstelle abbilden - , sondern lediglich um eine Darstellung bestimmter gemeinsamer Eigenschaften.

So handelt es sich bei Personen um selbständige Wesen, von denen man annehmen darf, daß sie über sich selbst und die Dinge, mit denen sie sich befassen, Bescheid wissen. Haben sie Informationslücken oder nehmen sie eine Dienstleistung in Anspruch,

so benötigen sie die Hilfe von Verwaltungsinstitutionen. Sogenannte Benutzerprozesse, also von einem Programmierer entworfene selbständige Objekte, benötigen ebenfalls Verwaltungsinstitutionen (Systemprozesse), um ihre Aufgaben abwickeln zu können. Im folgenden sollen Eigenschaften der aufgezählten Begriffe des täglichen Lebens erläutert werden; in Klammern sind die Betriebssystembegriffe dazu angegeben.

Wenn Personen (Benutzerprozesse) mit Ämtern (Verwaltungsprozessen) in Kontakt treten, so müssen letztere über Informationen (Verwaltungsdaten) dieser Personen verfügen, aber jeweils nur so viel wie unbedingt nötig. Die Kfz-Zulassungsstelle braucht keine Informationen über den Familienstand des Kfz-Halters (Der Druckerverwaltungsprozeß braucht nicht zu wissen, welche Semaphore ein Prozeß möglicherweise anspricht). Das hat natürlich vor allem Datenschutzgründe; es hat aber auch den Vorteil, daß im Falle eines Unfalls bei einer Behörde (Fehler bei einem Systemprozeß) der reibungslose Ablauf der anderen Behörden gewährleistet bleibt. Bei einer einzigen Großbehörde (allumfassendes Betriebssystem) wäre dies nicht möglich. Nachdem die Funktionstüchtigkeit der Behörde (Rücksetzen) wieder hergestellt ist, wobei auf archivierte Daten (Sicherungsdaten) zurückgegriffen werden konnte, kann der Betrieb wieder aufgenommen werden.

Kann das Gebäude, in dem die Behörde untergebracht war, nicht mehr restauriert werden, so muß das Amt in ein anderes Bauwerk (Rechner) umziehen (umkonfigurieren). Dabei müssen auch die Kommunikationswege (Leitungen zwischen den Rechnern) neu installiert werden, so daß Medien wie Post und Telefon (Botschaften) ihr Ziel wieder regulär erreichen.

Sicherlich könnten noch mehr Gemeinsamkeiten aufgezählt werden, doch es wird auch aus dieser kurzen Zusammenstellung deutlich, daß Begriffe des täglichen Lebens in ihrer gewachsenen Entwicklung auch in Prozeßsystemen Eingang finden. Insbesondere können Entwurfsentscheidungen, wie die Dezentralisierung oder das Wiederherstellen der Funktionstüchtigkeit von Behörden, übernommen werden.

1.3. Verwendete Spezifikationsmethode

Diese Arbeit orientiert sich in der Denk- und Schreibweise an der Dissertation "Ein Konzept zur Darstellung und Realisierung von verteilten Prozeßautomatisierungssystemen" von A. Fleischmann (/FLEI84/) und an den Erweiterungen von PEARL zu Verteiltem PEARL (/FHKK83/). Aus diesem Grund werden die Entwürfe und Spezifikationen in dieser Arbeit mit Hilfe der Spezifikationsmethode PASS (Parallel Activities Specification Scheme) oder den PEARL-Sprachkonstrukten TRANSMIT/RECEIVE und GUARDED REGION beschrieben. Die Methode PASS, die Prozesse als grundlegendes Darstellungs- und Strukturierungsmittel verwendet, hat den Vorteil, über ein durchgängiges Konzept zur Darstellung von Hardware- und Software-Moduln zu verfügen.

Allerdings waren für die Anwendbarkeit der Methode einige Einschränkungen und Erweiterungen notwendig. So heißt es in /FLEI84/ auf Seite 119 zum Beispiel:

"(...)

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 Synchronisations- und Kommunikationskonzepte für gemeinsame Objekte und Botschaftskonzepte in einem Programm überlagern.

(...)

Prozeßbündel und Prozeßgruppen wurden eingeführt, da es (...) 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, (...) 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. (...) 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.

(...)"

Im Gegensatz zu diesen Forderungen verschwinden bei dem hier vorgestellten Betriebssystemkonzept die gemeinsamen Objekte (Daten) gänzlich, sie werden verboten. Es wird nur mehr durch Botschaften kommuniziert (siehe Kapitel 3). Das bedeutet,

daß sich Synchronisations- und Kommunikationskonzepte für gemeinsame Objekte und Botschaftskonzepte, wie in /FLEI84/ gefordert, nicht überlagern, denn es gibt keine gemeinsamen Daten mehr. Was das oben aufgeführte Beispiel für den Zugriff auf gemeinsame Objekte betrifft, so ergibt sich kein Mehraufwand ("Overhead") bei Vorhandensein eines Monoprozessors; dort muß sowieso sequentiell gerechnet werden. Nur bei Multiprozessoren (Prozessoren mit gemeinsamem Speicher) kann es bei Lesezugriffen zu Verlangsamungen kommen, da sich die Prozesse (Prozessoren) über einen Verwaltungsprozeß koordinieren müssen.

Die ausschließliche Verwendung von Botschaften in dieser Arbeit soll nicht nur die Überlagerung von Synchronisations- und Kommunikationskonzepten für gemeinsame Objekte und von Botschaftskonzepten verhindern, sondern auch die Kommunikation eines Prozesses mit anderen Prozessen einer einheitlichen Systematik unterziehen. Diese Systematik, also ein einheitliches Konzept, ist für die leichtere Behandlung der Sicherungs- und Rücksetzproblematik nötig (siehe Kapitel 5).

Eine Erweiterung der Methode PASS findet bei dem hier vorgestellten Konzept statt, indem bei der "Verundung" von Botschaften (gleichzeitiges Senden oder gleichzeitiges Empfangen, GUARDS im Verteilten PEARL) auch eine Verknüpfung zwischen Senden und Empfangsbotschaften ermöglicht wird (siehe Kapitel 4). Da Sendekanten als Doppel- und Empfangskanten als Einfachpfeile dargestellt werden, wird dadurch allerdings ein graphisches Problem aufgeworfen!

2. Einige beispielhafte Ansätze für ausfallsichere verteilte Systeme

Es soll nachfolgend aufgezeigt werden, in welchem Rahmen sich diese Arbeit wiederfindet und welche vergleichbaren Ansätze bereits unternommen wurden, um die Problematik des Fehlverhaltens von Prozessen in verteilten Systemen in den Griff zu bekommen. Die hier vorgestellten Artikel sollen also zum einen einführen, insbesondere in die hier verwendeten Begriffe, und zum anderen bereits Wege aufzeigen, in welche Richtung eine Problemlösung zu gehen hat. Bei der Analyse der Aufsätze soll allerdings eine kritische Bestandsaufnahme nicht fehlen.

Die ausgewählten Artikel besitzen eine Gemeinsamkeit: Sie entstammen unmittelbar oder mittelbar dem "Newcastle Reliability Project". Die Ausnahme bildet lediglich Kapitel 2.4., wo kurz das "Three Phase Commit"-Protokoll (/BEHG87/) vorgestellt wird.

Seit 1979 haben sich Wissenschaftler, wie Anderson, Knight, Shrivastava oder Randell an der Universität von Newcastle bemüht, ein zuverlässiges und ausfallsicheres System zu entwerfen. Die Versuche, dieses System zu entwickeln und zu verbessern, haben zu einer Vielzahl von Publikationen geführt, so daß die Artikel von Shrivastava und Anderson/Knight nur einen kleinen Ausschnitt aus dieser Projektarbeit zeigen. Inzwischen wurden die meisten Veröffentlichungen, die in verschiedenen Zeitschriften und zu verschiedenen Tagungen vorgestellt wurden, in einem Buch (/SHRI85/) zusammengefaßt. Dadurch gelangt man zu einem guten Überblick über das Newcastle-Projekt. Da hier keine umfassende Darstellung dieses Projektes durchgeführt werden soll, wurden nur zwei Artikel herausgegriffen, um einen Einblick in die Problematik ausfallsicherer Systeme zu erlangen.

Der Artikel von Kimm entstammt zwar nicht dem Newcastle-Projekt, aber seine Vorschläge basieren auf der Erweiterung von (CONCURRENT) PASCAL, die vornehmlich wieder von Randell und auch Shrivastava stammen.

2.1. Software-Fehler-Toleranz in Realzeitsystemen

Der Artikel "A Framework for Software Fault Tolerance in Real- Time Systems" von Anderson und Knight (/ANKN83/) verfolgt zwei Ziele: Zum einen soll eine Übersicht und eine Einführung in die Problematik von Fehlern in Realzeitsystemen gegeben werden. Zum anderen sollen Ansätze gezeigt werden, wie diesen Fehlern begegnet werden könnte. Aufgrund der Allgemeingültigkeit der Begriffe werden ihre Definitionen vorangestellt, um damit auch eine Einführung in die Begriffswelt ausfallsicherer verteilter Systeme zu ermöglichen.

2.1.1. Definition von Begriffen

Ausgehend vom Begriff "Fehler" werden folgende Möglichkeiten der Fehlerunterscheidung eingeführt:

- Failure (Fehlverhalten): tritt auf, wenn das nach außen sichtbare Verhalten mit dem spezifizierten (geplanten) Verhalten nicht übereinstimmt;
- Error (erroneous state, unkorrekter Zustand): beschreibt den Zustand des Systems, der zu einem sichtbaren Fehlverhalten führen kann;
- Fault (Fehler): verursacht den unkorrekten Zustand des Systems.

Zusammengefaßt heißt das: Ein Fehler (z.B. das Durchbrennen eines Hardware-Bausteins) verursacht den unkorrekten Zustand eines Systems, der sich im sichtbaren Fehlverhalten eines oder mehrerer Prozesse äußert.

Die Vorgehensweise beim Erkennen und Beheben von Fehlern ist wie folgt (siehe auch /RAND78/):

- Fehlererkennung: Zunächst muß ein Fehler erkannt werden, z.B. durch regelmäßige Überprüfungen.
- Schadensausmaß: Es muß festgestellt werden, wieviel "zerstört" worden ist, also wie viele Prozesse und Objekte betroffen sind. Dabei muß überprüft werden, wie sich der Schaden ausgebreitet hat.

- Fehlerbehebung (Rücksetzen): Die eigentliche Schadensbehebung besteht darin, das System aus dem fehlerhaften wieder in einen korrekten Zustand zu überführen. Das geschieht in den meisten Fällen durch ein Zurücksetzen des Prozesses oder Systems in einen vorherigen korrekten Zustand ("Recovery").
- Wiederaufnahme des Betriebs: Um zu verhindern, daß der Fehler sofort wieder auftritt und damit die Aktionen hinfällig machen würde, müssen entsprechende Maßnahmen ergriffen werden. Entweder wird der Fehler behoben oder er muß zukünftig umgangen werden, z.B. durch Umkonfigurieren.

Außerdem können Fehlerzustände und Fehler nach ihrem Ausmaß beurteilt werden:

- interne Fehler: sind nur innerhalb eines Prozesses maßgebend und können von ihm selbst behoben werden;
- externe Fehler: haben zwar auch nur Auswirkungen auf einen bestimmten Prozeß, können aber von ihm allein nicht behoben werden;
- durchdringende Fehler: sind weder auf einen Prozeß beschränkt noch können sie von einem allein beseitigt werden.

Schließlich können Fehler nach ihrer "Stärke" unterschieden werden:

- andauernd: Das System kann eine bestimmte Anzahl von Fehlern verkraften (Obergrenze); wird diese überschritten, kann das System nicht standhalten;
- kurzzeitig: Das Gegenteil von andauernd.

Durch die zwei Arten der Unterscheidung lassen sich die Fehler von sehr leicht (intern, kurzzeitig) bis sehr schwer (durchdringend, andauernd) einstufen. Dadurch soll es möglich sein, verschiedene Arten von Rücksetzmaßnahmen anzuwenden. Aber dafür ist es notwendig, erkannte Fehler zu klassifizieren. Wie dies in der Praxis auszusehen hat, bleibt in diesem Artikel allerdings offen. Hat man Fehler erkannt, so muß man Maßnahmen ergreifen, um sie zu beheben. Dies soll im nachfolgenden Abschnitt untersucht werden.

2.1.2. Erkennen und Beheben von Fehlern

Zum Erkennen von Fehlern werden mehrere Verfahren vorgeschlagen. Angefangen von einfachen Hardware- über Bereichs-, Feldgrenzen- oder Invarianten-Überprüfungen kann man sich eine Vielzahl an Verfahren vorstellen.

Schwieriger ist es schon festzustellen, welches Schadensausmaß vorliegt. Dazu ist es notwendig, eine Fehler-"Geschichte" von Prozessen zu erstellen. Damit kann festgehalten werden, welchen Verlauf die Fehler hatten und welches Ausmaß sie angenommen haben. Interne Fehler sollen dabei nicht beachtet werden. Allerdings wird von Anderson/Knight festgestellt, daß die Unterscheidungskriterien fließend bzw. implementationsabhängig sind. Um eine derartige Fehlergeschichte zu erstellen, wird folgende Bit-Matrix E vorgeschlagen (/ANKN83/):

Prozeßnummer					
3	0	0	0	0	
2	0	1	1	0	
1	0	0	0	0	
<--- Zeit					

Für die Prozesse 1 bis 3 wird zu bestimmten Zeitpunkten (siehe dazu 2.1.3.) festgestellt, ob ein Fehler vorliegt (1 = Fehler, 0 = kein Fehler). Nach dieser Feststellung werden alle Bit- Einträge um eine Stelle auf der Zeitachse nach rechts geschoben und mit Nullen aufgefüllt; diese stellen damit den momentanen Zustand des Systems dar.

Bei den Rücksetz- und Wiederaufnahmemassnahmen muß unterschieden werden, ob es sich um einen internen, externen oder durchdringenden Fehler handelt. Bei den internen Fehlern ist es am einfachsten. Dafür gibt es schon genügend Verfahren: z.B. der "Recovery-Block" in PASCAL (siehe Kapitel 2.2.) oder das "ON" in PL/I.

Bei externen Fehlern muß unterschieden werden, ob der defekte Prozeß in Kontakt (Kommunikation) mit anderen Prozessen stand. Je nachdem ob dies der Fall war oder nicht, muß nur der einzelne Prozeß oder alle an der Kommunikation beteiligten Prozesse zurückgesetzt werden. Um den Aufwand, alle an der Kommunikation beteiligten Prozesse zurücksetzen zu müssen, zu verringern, ist es hilfreich, eine Unterscheidung zwischen kritischen und unkritischen Prozessen zu treffen. Es müssen dann nur die

kritischen ("wichtigen") Prozesse zurückgesetzt werden. Diese Unterscheidung kann noch durch folgende Unterteilung verfeinert werden:

- 1) Ignorieren oder übergehen des Fehlers und versuchen, die Arbeit ohne Maßnahmen fortzuführen;
- 2) Fehlerbehandlung in möglichst kurzer Zeit - in der Zwischenzeit soll so gut wie möglich weitergemacht werden;
- 3) ausführliche Fehlerbehandlung, so daß der defekte Prozeß danach normal weiterarbeiten kann.

Zum zweiten Punkt gehört auch das zwischenzeitliche Auswechseln des defekten Prozesses durch einen einfacheren Prozeß, der nicht alle Funktionen erbringen kann.

Eine wichtige Frage wird in diesem Zusammenhang angeschnitten: Wie kann man auf die wahrscheinlich falschen Meldungen und Ausgaben des defekten Prozesses reagieren?

Bei den Eingaben ist es einfacher. Man kann auf die letzten noch richtigen Daten zurückgreifen, oder man bildet einen Durchschnitt der letzten wichtigen noch erhaltenen Daten.

Zu Punkt 3) wird vor allem eine "Stand-by"-Lösung vorgeschlagen, wobei der Ersatzprozeß nicht unbedingt alle Funktionen des (teilweise) zerstörten Prozesses erfüllen muß. Letzterer kann wiederum als Ersatz für den Ersatzprozeß dienen, außer der Fehler ist zu gravierend. Eine andere Möglichkeit besteht darin, für kurzzeitige und für andauernde Fehler jeweils verschiedene Ersatzprozesse bereitzustellen.

Für durchdringende Fehler sind kaum mehr Maßnahmen möglich, höchstens ein komplettes Ersetzen der Programme oder ein vollständiges Rücksetzen aller Prozesse erscheint sinnvoll.

2.1.3. Ein Modell für Realzeitsysteme

Um ein Modell für Realzeitsysteme zu gewinnen, wird ein sogenannter "Synchronisationsgraph" G vorgestellt, der aus Programm- (P_1, \dots, P_n) und Zeitknoten (T_1, \dots, T_m) besteht. Ein Beispiel für einen Graphen gibt Bild 2-1. Ein Synchronisationsgraph soll nicht nur den Aufbau eines Realzeitsystems durch Prozesse darstellen, sondern kann auch durch die Zeitknoten die Zeitpunkte für die Prozeßsynchronisationen festlegen. Diese Zeitpunkte legen die spätest mögliche Zeit fest, zu der eine Synchronisation stattgefunden haben muß. Ein Prozeßknoten stellt im Prinzip den Programmablauf eines Prozesses zwischen zwei Synchronisationen bzw. Kommunikationen dar. Man kann damit in Bild 2-1 die Knoten P_1 , P_5 , P_6 und P_7 als parallele Prozesse verstehen, die sich zum Zeitpunkt T_4 synchronisieren.

Für den Synchronisationsgraphen gelten folgende Eigenschaften:

- G ist endlich, nicht zyklisch und gerichtet, mit einem "Rahmenanfangsknoten" (T_1) und einem "Rahmenendknoten" (T_m);
- Zeitknoten dürfen nicht doppelt vorkommen;
- nur zwei verschiedene Knotentypen können über eine gerichtete Kante miteinander verbunden werden (G ist bipartit);
- gibt es einen Pfad von T_i nach T_j , so muß gelten: $T_i < T_j$.

Eine Gefahr besteht darin, daß die Graphen sehr schnell groß und unübersichtlich werden. Dagegen wird das Argument angeführt, daß in Prozeßsystemen meistens relativ kleine zyklische Prozesse vorherrschen, so daß es möglich ist, Graphen, bestehend aus Teilgraphen, zu kreieren ("verschachtelte Graphen").

Sehr leicht läßt sich die Ähnlichkeit mit PETRI-Netzen feststellen: Zeitknoten sind als Transitionen und Prozeßknoten als Bedingungen zu verstehen. Dadurch zeichnen sich diese Synchronisationsgraphen durch die gleichen Eigenschaften wie die PETRI-Netze aus. Prozesse können nicht als geschlossene Einheiten dargestellt werden, sondern es wird nur das Zusammenspiel der Prozesse untereinander festgehalten.

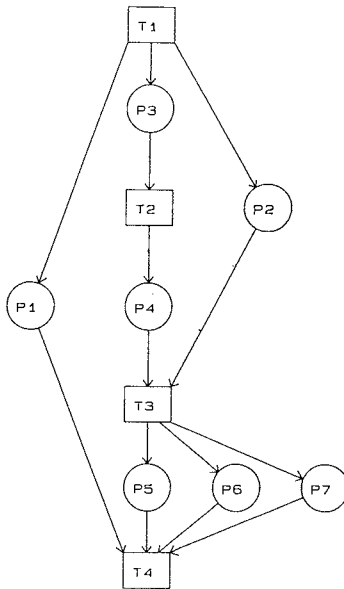


Bild 2-1: Beispiel eines Synchronisationsgraphen

Als weiterer Punkt wird die Problematik des Rücksetzens von Prozessen aufgegriffen. Dabei wird klargestellt, daß alle Prozesse, die an einer Synchronisation beteiligt sind (kurz "conversation" oder "Unterhaltung" genannt), einen gemeinsamen Rücksetzpunkt haben müssen, da es sonst zum "Domino"-Effekt (siehe /WEBE83/) kommen kann. Denn, existiert kein derartiger Punkt, so muß bei einem Fehler in einer Konversation für die darin beteiligten Prozesse erst ein gemeinsamer Wiederaufsetzpunkt gesucht werden. Dadurch kann es zu einem Aufrollen der Aktivitäten des gesamten Systems bis zu einem sehr frühen Zeitpunkt kommen.

Folgende Überlegungen sind deshalb für diesen Vorgang wichtig: Die Prozesse betreten zu verschiedenen Zeitpunkten die Unterhaltung; dabei sollen sie ihren jeweiligen Sicherungspunkt erstellen. Die Unterhaltung muß von allen beteiligten Prozessen gleichzeitig verlassen werden.

Um das Rücksetzen und die Implementation von fehlertoleranten Realzeitsystemen zu erleichtern, werden für den Synchronisationsgraphen die nachfolgenden Bedingungen, die sich gegenseitig ausschließen, erhoben. Die Zeitpunkte T_i , T_j , T_m und T_n , sowie die Prozesse P und Q bilden einen Synchronisationsgraphen, also ein Prozeßsystem, wobei die geordneten Paare (T_i, P) , (P, T_j) , (T_m, Q) und (Q, T_n) gelten. Die Kommunikationsbedingungen lauten:

Die Kommunikation (der Informationsfluß)

- von P nach Q ist nur erlaubt, wenn $T_j \leq T_m$ (siehe Bild 2-2a);
- von Q nach P ist nur erlaubt, wenn $T_n \leq T_i$ (siehe Bild 2-2b);
- zwischen P und Q ist nur erlaubt, wenn $T_j = T_n$ (siehe Bild 2-2c).

Die beiden ersten Bedingungen bedeuten eine Sequentialisierung, während Bedingung 3 den kommunizierenden Prozeßverlauf darstellt, also die Synchronisation der beiden Prozesse P und Q . Für letztere Restriktion gilt auch, daß die Prozesse, die sich dabei in einem "Austausch" befinden, Sicherungspunkte haben müssen. Das hat zur Folge: entweder wird die Kommunikation vollständig durchgeführt, oder alle Prozesse werden zu ihren Ausgangspunkten zurückgesetzt. Beim Austausch handelt es sich deshalb um eine rücksetzfähige, unteilbare Aktion (siehe auch 2.3.1.).

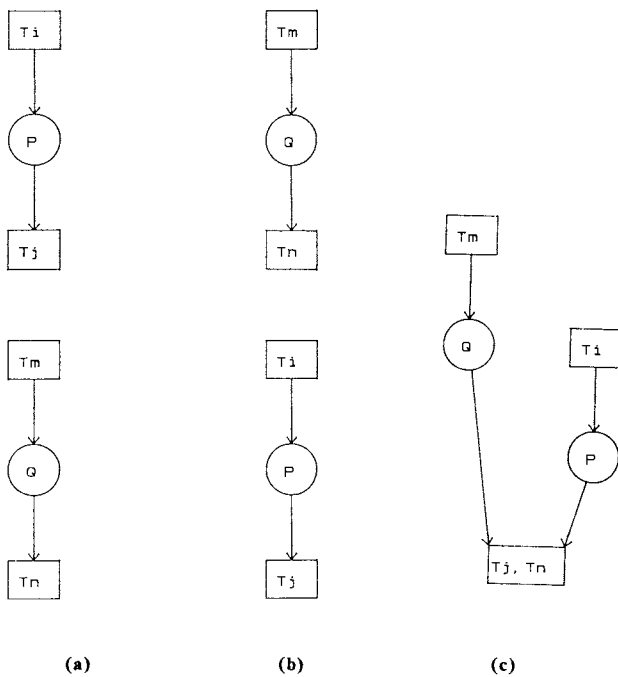


Bild 2-2: Kommunikationsrestriktionen, /ANKN83/

2.1.4. Kommentar

In der vorgestellten Arbeit werden zum einen Begriffe und prinzipielle Überlegungen dargelegt, die sich gut zur Einführung in die Materie eignen. Es wird zwar die Einschränkung gemacht, daß die eingeführten Begriffe und Sachverhalte nur für Software-Fehler, also Programmfehler gelten. Aber für das Fehlverhalten von Hardware-Bausteinen dürfen ähnliche Folgerungen gezogen werden.

Zum anderen wird ein Modell vorgestellt, mit dem Synchronisationszusammenhänge dargestellt werden können. Allerdings zeigt sich, daß die Ähnlichkeit mit den Petri-Netzen unverkennbar ist, und die Unübersichtlichkeit und Komplexität der Petri-Netze ist bereits vielfach diskutiert. Es wird zwar behauptet, daß in den meisten "praktischen Realzeitsystemen" nur begrenzte Kommunikationen mit immer wiederkehrenden Zyklen vorherrschen, aber für sehr komplexe Systeme dürfte das Modell trotzdem nicht ausreichen. In aufwendigen Systemen, wie z.B. den Protokollen des ISO-7-Schichtenmodells, ist die Anzahl wiederkehrender Zyklen im Vergleich zur Gesamtgröße des Programms relativ gering. Im Synchronisationsgraphen tauchen zwar Prozeßknoten auf, aber eine Abbildung auf die tatsächlichen Prozesse ist aufwendig. Es gilt hier die gleiche Aussage wie bei Petri-Netzen, daß die Kommunikationen zwar klar herausgestellt werden, aber die Aufteilung des Programms in (parallele) Prozesse nicht unterstützt wird. Der Graph ist also lediglich eine Beschreibungsmöglichkeit für die Synchronisation und die Kommunikation, wobei es egal ist, wie dies stattfindet, z.B. auch über gemeinsame Objekte.

Gänzlich ungeeignet ist das Modell für "remote procedure calls". Begründet wird dies mit der Aussage, daß "das Ausmaß der Parallelität beschränkt ist". Das bedeutet, daß es nur sehr wenige und dann nur zwischen einigen Prozessen Kommunikationen gibt. Außerdem stößt die Behauptung, es gäbe keine "Management- Probleme für Ressourcen in Systemen, wo Prozesse beliebig kreiert werden können", auf Widerspruch, wie die Schwierigkeiten bei der Realisierung der Realzeitprogrammiersprache ADA (/ADA83/) auch zeigen.

Problematisch erscheint ebenfalls die Behandlung der Rücksetzpunkte. So wird die Forderung aufgestellt, daß ein Prozeß bei Eintritt in eine Unterhaltung einen Rücksetzpunkt aufstellen muß und, daß alle Prozesse die Unterhaltung gleichzeitig beenden müssen (dabei werden die Rücksetzdaten wieder weggeworfen!). Es wird nicht gesagt, wie dieses gemeinsame Verlassen realisiert werden soll und was passiert, wenn ein Prozeß, der das Gespräch verlassen hat, danach "abstürzt". Der letzte Sicherungspunkt, soweit die dazugehörigen Daten noch vorhanden sind, liegt beim

Eintritt (!) in das Gespräch, so daß die anderen beteiligten Prozesse nicht mehr rücksetzbar sind, da sie bereits das Gespräch verlassen haben.

Implementationshinweise gibt es in diesem Artikel kaum, außer, daß die Existenz eines "error handlers" (Fehlerbehebungsprozesses) erwähnt wird. Es wird allerdings keine Aussage über dessen Ausfallsicherheit gemacht.

2.2. Das auf Monitoren basierende "Conversation"-Schema

Die Sprache PASCAL war zwar zunächst als reine "Lehrsprache" erfunden worden, hat aber inzwischen auch in der Praxis starken Eingang gefunden. Aufgrund der Anforderungen aus der Lehre und vor allem der Praxis ist es notwendig geworden, die Sprache weiterzuentwickeln. Gerade in Zusammenhang mit ausfallsicheren Systemen wurden bereits früh erste Versuche unternommen, PASCAL, genauer der Erweiterung CONCURRENT PASCAL, Konstrukte hinzuzufügen, um mit Fehlern fertig zu werden, die bei parallelen Prozessen verstärkt auftreten können. K. H. Kimm hat nun den Versuch unternommen, diese von Randell et. al. (/RAND78/) gemachten Vorschläge zu verbessern und weiterzuentwickeln. Seine Überlegungen sind in "Approaches to Mechanization of the Conversation Scheme Based on Monitors" (/KIMM82/) zusammengefaßt. Dabei werden vier Ansätze zur Realisierung des von Randell eingeführten "Conversation"-Schemas erläutert. Mit dem "Conversation"-Schema, nachfolgend als "Gespräch" bezeichnet, ist ein kontrollierter Datenaustausch zwischen Prozessen möglich, wobei zusätzlich Rücksetzmöglichkeiten bestehen.

Es handelt sich dabei um eine Erweiterung des "recovery block"-Konzeptes (siehe Bild 2-3), das in die Sprache CONCURRENT PASCAL integriert wurde (Die Integration in eine andere Sprache wäre auch vorstellbar.). Der "recovery block" ermöglicht das Zurücksetzen eines Prozesses an den Anfang eines Programmblocks, nachdem eine Operation aus diesem Block fehlgeschlagen hat:

Es wird der Anweisungsblock B1 ausgeführt und anschließend wird mit Hilfe des Tests T überprüft, ob die Anweisungen richtig ausgeführt wurden. Ergibt die Überprüfung einen Fehler, so wird alternativ versucht, die Anweisungsfolge B2 durchzuführen. Dieses Verfahren wird solange verfolgt, bis entweder der Test ein positives Ergebnis erbringt oder, wenn kein Anweisungsblock erfolgreich war, der "error"-Ausgang gewählt werden muß.

Das Gespräch bietet als Erweiterung die Möglichkeit, daß mehrere Prozesse gleichzeitig derartige Recovery-Blöcke ausführen und darin untereinander Daten austauschen. Dieser Datenaustausch erfolgt dabei über einen Monitor. Bild 2-4 zeigt die Kommunikation dreier Prozesse A, B, und C über den Monitor M.

```

ensure T
  by B1
  else by B2
  ....
  else by Bn
  else error.

```

Bild 2-3: Recovery-Block, /KIMM82/

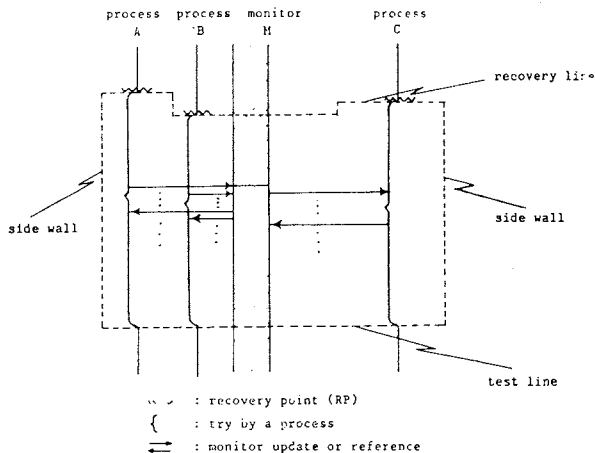


Bild 2-4: "Conversation", /KIMM82/

Ein Gespräch wird auch als eine "rücksetzbare interaktive Sitzung" bezeichnet. Es sind auch geschachtelte Gespräche möglich. Unter einer Verschachtelung versteht man dabei das gleichzeitige Ausführen von Gesprächen durch einen Prozeß, wobei die Ein- und Austrittszeitpunkte für die Gespräche voneinander unabhängig sind.

Betrifft ein Prozeß einen Recovery-Block, das geschieht unabhängig von den anderen Prozessen, so wird für ihn ein Rücksetzpunkt angelegt. Die Gesamtheit aller Rücksetzpunkte definiert eine Rücksetzlinie (siehe Bild 2-4). Solange der Prozeß sich sozusagen im "kritischen Abschnitt" befindet, hat er Zugriff auf den Monitor. Will der Prozeß den Block verlassen, wird für ihn ein Abschlußtest durchgeführt (Testlinie).

Schlägt dieser fehl, müssen alle beteiligten Prozesse zurückgesetzt werden. Die sogenannte "Seitenwand" drückt aus, daß außer A, B und C keine anderen Prozesse an der Kommunikation beteiligt sein dürfen, weder aktiv noch passiv.

Zwei mögliche Gefahren müssen bei diesem Konzept vermieden werden:

- Ein "deserter process" (**aushungernder Prozeß**) kann einen Systemstillstand provozieren, indem er sich "weigert", den Recovery-Block zu betreten, z.B. weil das Ergebnis einer IF-Abfrage den Prozeß am Block vorbeiführt. Die anderen Prozesse, die sich im Gespräch befinden, können nicht fortfahren, da sie auf den Partner warten müssen.
- Die Seitenwände sind durch "Informationsschmuggel" nach innen und/oder nach außen durchlässig.

Im folgenden sollen 4 Mechanismen beschrieben werden, das Gespräch so zu verbessern, daß obige Schwächen vermindert werden oder gar nicht mehr auftreten können. In der dargestellten Reihenfolge sind sie nach Komplexität geordnet und stellen so immer stärkere Maßnahmen dar.

2.2.1. Namensverbundener Recovery-Block

Mit Hilfe eines dem Recovery-Block vorangestellten Namens (siehe Bild 2-5) wird der Block als "Conversation" identifiziert. Dadurch ist eine etwas leichtere Überprüfung der Zusammengehörigkeit des gleichen Gesprächs in den verschiedenen Prozessen (durch den Programmierer) möglich.

```
[conv C:]  
ensure T  
by B1  
else by B2  
....  
else by Bn  
else error.
```

Bild 2-5: Namensverbundener Recovery-Block, /KIMM82/

Mit dieser einfachen Maßnahme werden allerdings die wenigsten Probleme gelöst:

- Es kann nicht überprüft, geschweige denn verhindert werden, daß ein oder mehrere "deserter"-Prozesse existieren. Die bloße Namensgebung verpflichtet keinen Prozeß, in das Gespräch einzutreten.
- Eine "saubere" Verschachtelung von Gesprächen durch einen Prozeß kann nicht gewährleistet werden.
- Informationsschmuggel kann nicht verhindert werden, denn ein Prozeß kann jederzeit auf den Gesprächs-Monitor zugreifen, ohne sich dabei in einem Gespräch zu befinden.
- Der Abschlußtest und die Gespräche sind über alle beteiligten Prozesse verstreut und sind deshalb schwierig oder nur mit Übung lesbar.

2.2.2. Gesprächsmonitor

Der Recovery-Block wird durch die Namensangabe des verwendeten Monitors als Gesprächs-Block identifiziert (siehe Bild 2-6).

```
ensure T  
using-cm CM  
by B1  
else by B2  
....  
else by Bn  
else error.
```

Bild 2-6: Recovery-Block mit Gesprächsmonitor, /K1MM82/

Diese Maßnahme bringt gegenüber dem ersten vorgestellten Mechanismus kaum Verbesserungen, außer daß der Informationsschmuggel verhindert wird. Das ist darin begründet, daß die einzige Zugriffsmöglichkeit auf den Monitor nur innerhalb des Recovery-Blocks besteht, also kein Prozeß von außerhalb zugreifen kann.

2.2.3. Abstrakter Datentyp

Wie Bild 2-7 zeigt, wird dem Recovery-Block nicht nur der bzw. die Namen der Monitore vorangestellt, sondern auch die zum Monitor gehörigen Zugriffsprozeduren. Damit kann ein Prozeß nur dann in ein Gespräch eintreten oder eine Alternative des Recovery-Blocks ausführen, wenn er eine Monitorprozedur aufruft. Es findet also eine Abkehr von der Sichtweise, daß sich ein Monitor innerhalb eines Recovery-Blocks befinden muß, statt. Jetzt ist der Monitor die Hülle über dem Recovery-Block.

Als zusätzliche Regel gilt:

Ruft ein Prozeß die Prozedur eines Gesprächsmonitors auf, muß er gleichzeitig alle übrigen, von ihm regulär verwendeten Monitore verlassen.

Schachtelungen bezüglich der Gesprächsmonitore, mit Hilfe der Prozeduren, sind möglich. Allerdings müssen die Schachtelungstiefen verschiedener Prozesse, die gleiche Gesprächsmonitore verwenden, gleich sein. Der Stern (*) in Bild 2-8 gibt eine nicht mehr durchführbare Testlinie bei ungleicher Schachtelungstiefe an. Ergibt nämlich der Test von "CONV2"(*) ein negatives Ergebnis, so ist ein Rücksetzen auf den Beginn des Gesprächs notwendig. Das bedeutet aber, daß Gespräch "CONV3" nicht mehr "stattgefunden haben" darf. Prozeß "J" müßte also vor den Beginn des Gesprächs "CONV3" zurückgesetzt werden. Dies ist aber nicht mehr möglich, da "J" bereits seine Testlinie, also auch das Gespräch, verlassen hat.

Mit diesem Mechanismus sind nahezu alle Probleme, wie sie noch in 2.1.2. bestanden, beseitigt. Das Restproblem ist die Gefahr eines Deadlocks durch einen aushungernden Prozeß.

```

type C=
  conversation(..."nested conversations"... )
  participants
    PROCA(..."formal parameters"...);
    PROCB(.....);
    .....
  var
    CM1: c-monitor-type1;
    CM2: c-monitor-type2;
    .....
  ensure CAT      "Conversation acceptance test"
  by begin        "Primary interacting session"
    PROCA: .....
    PROCB: .....
    .....
  end
  else by begin   "Alternate interacting session"
    PROCA: .....
    PROCB: .....
    .....
  end
  .....
  else error
  begin          "Initialization"
    init CM1, CM2(...), ...
  end
  var CONVI: C;

```

Bild 2-7: Gespräch mit abstraktem Datentyp, /KIMM82/

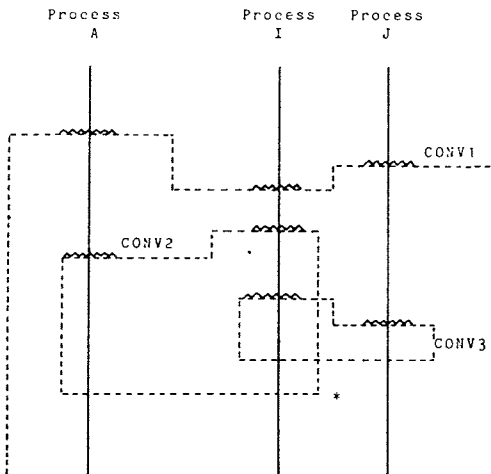


Bild 2-8: Illegales Verschachteln von Gesprächen, /KIMM82/

2.2.4. Kooperierender Recovery-Block

In dieser umfassendsten Stufe des Recovery-Blocks, dem "kooperierendem Recovery-Block" (Concurrent Recovery-Block = CRB), ist es möglich, sogenannte "Kinderprozesse" zu kreieren (siehe Bild 2-9); diese führen dann stellvertretend für den Mutterprozeß die Gespräche durch. Der "Mutterprozeß" tritt in den "Versuchsblock" (try block) ein und startet damit seine Kinderprozesse. Nachdem sich diese wieder beendet haben, kann der Akzeptanztest durchgeführt werden. Konnte dieser erfolgreich beendet werden, setzt der Mutterprozeß seinen Programmablauf fort.

```
ensure CAT      /* Acceptance test */  
by begin      /* Try block */  
    init MONITOR.1;  
    init PROCESS1.1(parameters);  
    .....  
    init PROCESSn.1(parameters)  
    end  
else by begin  /* Try block */  
    init MONITOR.2;  
    init PROCESS1.2(parameters);  
    .....  
    init PROCESSn.2(parameters)  
    end  
.....  
else error
```

Bild 2-9: Restriktive Version des CRB, /KIMM82/

Diese Form des CRB ist sehr einschränkend, da nur ein synchroner Eintritt in den Block möglich ist. Aus diesem Grund werden die "permanenten Variablen" eingeführt (siehe Bild 2-10).

Der CRB verhindert das Entstehen eines "deserter"-Prozesses, allerdings um den Preis eines sehr hohen Mehraufwands. Dieser entsteht unter anderem auch durch die "vorausschauende Ausführungsregel", mit der ein Mutterprozeß, wenn eine permanente Variable noch belegt ist, Kinderprozesse in deren Warteschlangen einreihen kann. Die Kinderprozesse können dann nacheinander, so wie sie in der Warteschlange

eingereiht sind (first-in-first-out), ihre Anweisungen ausführen.

```
const ....  
type .....  
  
permvar      "Permanent variables"  
  PV1: record      .  
        .....  
        end;        .  
  PV2: .....  
        .....  
var .....  
  
begin      "Main body of the initial process"  
  init .....  
  begin init ..... end  "Dynamic creation of"  
    .....      "children processes "  
  begin init ..... end  
    .....  
end
```

Bild 2-10: Concurrent Recovery-Block in Concurrent-PASCAL-Umgebung, /KIMM82/

Mit der "vorausschauenden Ausführungsregel" und der Möglichkeit "Kinderprozesse" zu kreieren, ist es möglich, daß ein Prozeß ("Mutterprozeß") seine Anweisungen weiter ausführt, ohne von einem langsamen oder "aushungernden" Prozeß aufgehalten zu werden.

2.2.5 Kommentar

Das auf Monitoren basierende "Conversation"-Konzept bietet, bei ausreichender Implementierung (Abstrakter Datentyp, CRB), eine sichere Art des Informationsaustausches. Beschränkt man sich auf die einfachere Realisierung des abstrakten Datentyps und nimmt aushungernde Prozesse in Kauf, so kann sogar von einer effizienten Realisierung gesprochen werden. Um einen Systemstillstand zu verhindern, könnte man den Recovery-Block noch um eine Timeout-Alternative erweitern. Dadurch würde eine Befreiung der wartenden Prozesse, die sich bereits im Block befinden, aus ihrer Wartestellung ermöglicht.

Der große Nachteil, den dieses Konzept in sich birgt, ist die Notwendigkeit, daß sich Prozesse und Monitore (und auch die permanenten Variablen beim CRB) auf dem gleichen Prozessor befinden müssen. Abhilfe können nur "Verteilte Monitore" schaffen. Das bedeutet, daß die Daten und Synchronisationsvariablen (z.B. Semaphore) über mehrere Prozessoren verteilt werden müßten, was an "globale verteilte Variable" erinnert. Solche Variable in einem konsistenten Zustand zu halten, verbunden mit dem entsprechenden Verwaltungsaufwand, ist bekanntlich kein leichtes Unterfangen.

Ein weiterer Ansatz wäre die Möglichkeit, die Monitore und ihre Prozeduren als eine Art des "remote procedure call"-Mechanismus (RPC) aufzufassen. Der Unterschied zum eigentlichen RPC liegt darin, daß alle beteiligten Prozesse zuerst in ihre jeweiligen Recovery-Blöcke eintreten müssen, bevor eine Kommunikation stattfinden kann. Gegenüber dem RPC birgt damit das Konversationsschema den Nachteil der Unflexibilität in sich, hat aber den Vorteil, daß Prozeßkommunikation und Sicherungsmaßnahmen deutlicher sichtbar sind.

2.3. Verteilte Systeme für Rücksetzbarkeit und Zusammenbruchsicherheit

In seinem Versuch, Rücksetzbarkeit und Zusammenbruchsicherheit für verteilte Systeme zu erzielen, verwendet S. K. Shrivastava (/SHRI81/) ein objektorientiertes Modell, bestehend aus mehreren logischen Schichten. Als Kommunikationskonzept wird dabei der "remote procedure call" verwendet, der es dem auftraggebenden Prozeß ermöglichen soll, auf entfernt liegenden Objekten Änderungen vornehmen zu können. Besondere Beachtung verdienen die Überlegungen des Autors, bei derartigen abgesetzten Aufrufen (Änderungen bei entfernt liegenden Objekten) Rücksetzbarkeit und eine gewisse Sicherheit vor Zusammenbrüchen zu erzielen.

2.3.1. Der "remote procedure call" als Mittel zum Nachrichtenaustausch

Um seine Überlegungen zu begründen, werden von Shrivastava (/SHRI81/) folgende Forderungen gestellt:

- Prozesse, die eine Berechnung beginnen, beenden diese entweder regulär oder dieser Versuch wird abgebrochen und hat keinen Effekt (Rücksetzfähigkeit);
- Prozesse, die für sich isoliert eine Berechnung durchführen, haben auf andere Berechnungen keinen Einfluß;
- Prozesse, die kooperierende Berechnungen durchführen, sind frei von gegenseitigen Einflüssen ("atomare Aktionen").

Diese Forderungen kann man durchaus als grundlegend für verteilte Systeme, die die Möglichkeit des Rücksetzens von Prozessen vorsehen, annehmen.

Als erste Operationen führt der Autor die Primitive "send" und "wait" ein, die nach dem Rendezvous-Prinzip funktionieren. Beide Primitive sind jeweils um eine Timeout-Komponente erweitert. Das Senden hat außerdem den Fehlerausgang "process missing", mit dem festgestellt werden kann, ob der betreffende Partner überhaupt existiert.

Basierend auf diesen beiden Konstrukten kann der "remote procedure call (RPC) eingeführt werden, der folgendermaßen implementiert werden soll (/SHRI81/):

```
send__message(processname,node,message)
  for all exceptions: invoke backward recovery
  repeat
    receive__message(processname,node,result)
    for all exceptions: invoke backward recovery
  until sequence numbers match;
-----
```

Mit der "sequence number" kann die richtige Antwort, Fragen und Antworten sind entsprechend durchnummeriert, festgestellt werden. Über die empfangenen Botschaften mit falscher Nummer, die dabei verloren gehen, wird nichts gesagt.

Der Autor versteht ein Gesamtsystem als eine Ansammlung von Prozessen und abstrakten Objekten, die auf mehrere lose gekoppelten Rechner verteilt sind. Ein abstraktes Objekt besteht dabei aus den Daten und den dazugehörigen Operationen, die die einzige Möglichkeit darstellen, die Daten zu ändern. Ein sogenannter Objektmanager stellt dieses abstrakte Objekt und die Funktionen dafür zur Verfügung.

Damit Prozesse Operationen auf entfernten Objekten durchführen können, müssen sie "remote procedure calls" absetzen, hinter denen sich dann der oben gezeigte Botschaftenaustausch verbirgt. Partner der Kommunikation ist aber nicht der Manager eines entfernten Objekts selbst, sondern der Autor stellt die Möglichkeit "Stellvertreterprozesse" als Aktivitätsträger einzuführen zur Verfügung (/SHRI81/):

```
1) create__worker (workerj,Nodei)
   unable exception: -----
```

```
2) delete__worker (workerj,Nodei)
   unable exception: -----
```

Es findet damit ein Botschaftenaustausch zwischen Prozeß und Stellvertreterprozeß statt, wobei letzterer dann die Aufrufe an den Objektmanager absetzt.

2.3.2. Recovery-Konzept

Mit Hilfe eines Sicherungspunktes kann ein Prozeß in einen "Recovery-Bereich" eintreten. In diesem Bereich kann ein Rücksetzen im Fehlerfall erfolgen oder, wenn kein Fehler gefunden wird, der Bereich normal verlassen werden.

Führt ein Prozeß einen RPC aus, so ruft er eine Prozedur des Objekt-Managers auf, der wiederum selbst zu seiner Ausführung Prozeduraufrufe durchführen kann. Dadurch entsteht eine Hierarchie von "Schnittstellen" (Schichten), wobei jede für sich einen Recovery-Bereich unabhängig von den anderen Schnittstellen haben kann. Das bedeutet, daß die Sicherungspunkte der einzelnen Schichten im Falle eines Aufrufs aufeinander abgestimmt werden müssen.

Über atomare Aktionen wurde gesagt, daß sie entweder vollständig abgeschlossen werden müssen oder im Fehlerfall keine Wirkung haben dürfen. Es liegt somit nahe, sie als einen Recovery-Bereich zu definieren. Bei Beginn einer atomaren Aktion durch einen Prozeß wird ein Rücksetzpunkt aufgestellt. Kann die Aktion erfolgreich beendet werden, so werden die Rücksetzdaten wieder verworfen. Im Falle eines Fehlers ist ein Rücksetzen notwendig. Allerdings sieht der Autor auch die Möglichkeit eines Notausgangs durch einen "fault"- (Fehler-) Ausgang vor.

Für das Rücksetzen selbst werden zwei Schemata unterschieden: das "getrennte" und das "inklusive" Rücksetzverfahren. Der Unterschied kann folgendermaßen erklärt werden:

Ein Objekt M wird durch zwei rücksetzbare Objekte A und B und durch ein nicht rücksetzbares Objekt C repräsentiert (implementiert). Aufgabe des Objektmanagers ist es, ein rücksetzbares Objekt M zur Verfügung zu stellen. Beim getrennten Verfahren erscheinen die Möglichkeiten, A und B zurückzusetzen, nur lokal für M, also nicht nach außen. Ein aufrufendes Programm kann damit voraussetzen, daß der Objektmanager die Rücksetzbarkeit von M gewährleistet. Beim inklusiven Verfahren sind die Rücksetzbarkeiten von A und B global, d.h., daß das aufrufende Programm die Rücksetzfunktionen von A und B verwendet und der Objektmanager nur das Rücksetzen von C zu gewährleisten hat.

Das inklusive Schema findet vor allem bei verschachtelten Aufrufen Anwendung, da hier für das aufrufende Programm leichter durchschaubar ist, welche (rücksetzbaren) Objekte beteiligt sind. Da es sich aber um verteilte Systeme handelt, zieht der Autor den Schluß, daß es bei abgesetzten Aufrufen einfacher wäre, ein disjunktes Schema zu verwenden, bei dem der Objektmanager eines entfernten Objekts die gesamte Rücksetzfähigkeit zur Verfügung stellt.

Wie schon erwähnt, muß, um auf abgesetzte (entfernte) Objekte zugreifen zu können, die Möglichkeit bestehen, entfernte Aktivitätsträger kreieren zu können. Diese müssen Botschaften als Realisierung des RPC, wie "erstelle Rücksetzpunkt", "lösche Rücksetzdaten" oder "rücksetzen" annehmen, ausführen und quittieren. Zwei Aussagen erscheinen in diesem Zusammenhang wichtig:

- Der Stellvertreterprozeß arbeitet als Kommandointerpretierer für seinen Hauptprozeß.
- Der Stellvertreterprozeß unterstützt das Zurücksetzen seines Hauptprozesses durch seine normalen Vorwärtsaktionen.

Das bedeutet, nicht der Stellvertreter selbst wird zurückgesetzt, sondern der Hauptprozeß und die betroffenen Datenobjekte. Läuft der Vertreter dabei falsch, wird er zerstört!

Als Nachteile für diese Verfahrensweise werden genannt:

- Der Stellvertreter "weiß" nicht, welche Objekte vom Haupt prozeß nacheinander aufgerufen werden. Er kann also nicht die Rücksetzfähigkeiten, die er lokal zur Verfügung hätte, geplant einsetzen.
- Dadurch muß der Stellvertreter, genauso wie die anderen Prozesse, alle Objekte, auf die er zugreift, rücksetzbar machen.

Als Lösung für diese Probleme wird vorgeschlagen, die Stellvertreterprozesse doch wieder selbst rücksetzbar zu machen.

Ein nicht angesprochenes Problem, nämlich, daß es bei den Objekten selbst zu Überlagerungen der Rücksetzbereiche kommt, wird dadurch aber nicht gelöst. Die Überlagerungen entstehen dadurch, daß die Stellvertreterprozesse unabhängig voneinander auf die Objekte zugreifen.

2.3.3. Maßnahmen gegen Ausfälle

Der Autor entwickelt eine Unterscheidung zwischen "rücksetzbaren" ("wiederherstellbaren") und "gesicherten" Objekten. Damit wird der eigentliche Unterschied zwischen Prozessen und Objekten klargestellt:

Objekte können gesichert werden, und man greift im Falle eines Fehlers auf die Sicherungsdaten zurück. Programme haben Rücksetzpunkte, auf die im Falle eines Fehlers zurückgesetzt wird. Wenn man also im Zusammenhang mit Objekten möglichst beide Eigenschaften wünscht, dann bedeutet dies: im Fehlerfall soll auf die gesicherten Daten und auf den Rücksetzpunkt des Objektmanagers zurückgegriffen werden.

Als Sicherungsverfahren wird das "Two Phase Commit"-Protokoll (TPC-Potokoll) nach Gray (/GRAY79/) oder ein ähnliches Protokoll vorgeschlagen (siehe Abschnitt 2.4.). Da dieses Verfahren aus dem Bereich der Datenbanksysteme stammt, ist es naheliegend, daß es für Datensicherungsmaßnahmen genutzt wird.

Der Autor stellt dabei fest, daß für (gesicherte) Objekte das inklusive Rücksetzverfahren besser geeignet wäre, da damit Sicherungsoperationen "global", also jederzeit zugänglich, sind. Da aber bisher das getrennte Verfahren bevorzugt wurde und es zu aufwendig wäre, beide Verfahren gemeinsam zu implementieren, wird auf das inklusive Verfahren verzichtet.

Außerdem werden Algorithmen für Stellvertreterprozesse angegeben, die es ermöglichen sollen, den schädlichen Auswirkungen von Zusammenbrüchen entgegenzuwirken. Dabei kann auch wieder eine Analogie zum TPC-Protokoll festgestellt werden. Erhalten Stellvertreterprozesse (oft als "light weighted processes" bezeichnet) von ihrem "Meister" keine Sicherungsnachricht, so machen sie ihre letzten Aktionen rückgängig und beenden sich. Kann der Hauptprozeß seine Sicherung durchführen, so können es auch die Stellvertreter, da sie die entsprechenden Nachrichten erhalten.

2.3.4. Kommentar

Im vorgestellten Artikel wird ein umfassendes System für widerstandsfähige "remote procedure calls" (RPC) dargestellt, das aber auch einige Nachteile in sich birgt. Was als grundsätzliche Einschränkung des RPC bekannt ist, ist das Fehlen der direkten Kommunikationsmöglichkeit zwischen (Haupt-) Prozessen. Der Nachrichtenaustausch über Objekte, die noch dazu verteilt sein können, ist relativ umständlich. Außerdem ist bisher bei Betriebssystemimplementierungen vor allem das aufwendige Kreieren und Zerstören von (Stellvertreter-) Prozessen platz- und sehr zeitaufwendig, auch wenn damit Rücksetzfähigkeiten verbunden sind.

Die Problematik beim asynchronen Zugriff auf die Objekte geht in die gleiche Richtung. Die Stellvertreterprozesse müssen sich bei jedem Zugriff untereinander koordinieren, vor allem bezüglich der Recovery-Bereiche. Denn, wie in Abschnitt 2.3.2. dargelegt, wird bei jedem asynchronen Zugriff auf ein Objekt ein eigener Recovery-Bereich begonnen, unabhängig von anderen Zugriffen. Dies verursacht Wartezeiten und einen komplexen Koordinierungsaufwand (verteilte Objekte!). Dieser Umstand wird vom Autor zugegeben, und er sucht diesbezüglich nach Optimierungsmöglichkeiten.

Die sogenannten Implementierungsanweisungen sind nicht detailliert, und es darf durchaus gefragt werden, ob die Realisierung wirklich, wie der Autor (unbelegt) vermutet, "einfach und unkompliziert" von statten gehen kann.

2.4. Das "Three Phase Commit"-Protokoll

Auf dem Gebiet der Datenbankverwaltungssysteme, insbesondere auch im Bereich der verteilten Datenbanken, gibt es bereits reiche Erfahrungen beim Einsatz von Verständigungsprotokollen. Ziel dieser Protokolle ist es, einen **gemeinsamen** Abschluß von unabhängigen Instanzen, die an einer Transaktion beteiligt sind, herbeizuführen. Unter einer Transaktion versteht man im allgemeinen den in sich abgeschlossenen, lesenden oder schreibenden Zugriff auf Datenbestände durch ein oder mehrere Verarbeitungsinstanzen. Insbesondere versteht man darunter die Überführung einer Datenbank von einem logisch konsistenten Zustand in einen anderen. Ein einfaches Beispiel stellt die Umbuchung eines Betrages X von einem Bankkonto A zu einem Bankkonto B dar. Die Umbuchung ist genau dann korrekt durchgeführt, wenn nach Abschluß der Transaktion Konto A um den Betrag X vermindert und B um X erhöht ist. Das Beispiel verdeutlicht, daß der Buchungsvorgang für einen Dritten erst dann sichtbar werden darf, wenn er abgeschlossen ist. Es liegt deshalb die Betonung auf der Eigenschaft "gemeinsam", um zu verhindern, daß etwaige, nicht aufeinander abgestimmte Zwischenstände nach außen erkennbar werden, so z.B., wenn der Betrag X vom Bankkonto A bereits abgebucht, aber noch nicht bei B eingetragen ist. Diese Forderung drückt sich auch in den von Ceri und Pelagatti (/CERI84/) formulierten notwendigen Eigenschaften für Transaktionen aus:

- Atomarität: Eine Transaktion wird entweder vollständig oder gar nicht durchgeführt. Es werden also Teilergebnisse, z.B. nach einem Fehler, wieder rückgängig gemacht.
- Beständigkeit: Die Ergebnisse einer Transaktion bleiben nach deren Abschluß immer bestehen, auch bei nachfolgenden Ausfällen.
- Nacheinanderausführbarkeit: Parallele Transaktionen sind unabhängig voneinander, so daß sie auch in beliebiger Reihenfolge ausgeführt werden können.
- Isolation: Erst nach Abschluß einer Transaktion werden die Ergebnisse bekannt gegeben. Teilergebnisse bleiben verborgen. Dadurch wird das Auftreten des sog. "Domino-Effektes" (siehe /WEBE83/) verhindert.

Ähnliche Eigenschaften wurden bereits bei der Vorstellung der Arbeit von Shrivastava (siehe Kapitel 2.3.1.) aufgezeigt, was deren Bedeutung unterstreicht. Hinzu kommt natürlich noch die Forderung, daß die Datenbestände (Datenbank) logisch konsistent

erhalten bleiben. Der Autor erwähnt dabei explizit das "Two Phase Commit"-Protokoll (Zweiphasenfreigabeprotokoll) als ein mögliches Verfahren um Transaktionen durchzuführen. Dieses Protokoll, das zum ersten Mal von Gray (/GRAY79/) vorgestellt wurde, war zunächst auf zentrale Datenbanksysteme beschränkt. Entsprechende Erweiterungen (/BEHG87/, /CERI84/) machten das Verfahren auch in verteilten (Datenbankverwaltungs-) Systemen einsetzbar und es ist wohl das heute am häufigsten verwendete. Hinzu kommen noch eine Vielzahl von erweiterten oder optimierten Protokollen, die aber letztendlich immer auf die gleiche Vorgehensweise zurückgehen. Es sollen deshalb zunächst die Grundprinzipien dargestellt werden, um dann auf Varianten und die Weiterentwicklungen, insbesondere auf das "Three Phase Commit"-Protokoll, einzugehen. Ausführliche Beschreibungen, die auch genau auf die Problematik des Recovery eingehen, finden sich u.a. in /CERI84/ und /BEHG87/.

Das Zweiphasenfreigabeprotokoll (für verteilte (Datenbank-) Systeme) zeichnet sich zunächst durch das Vorhandensein eines Koordinators ("coordinator") aus, der die Transaktion steuert bzw. beendet. Die übrigen beteiligten Instanzen (Prozesse) verhalten sich passiv und werden als Partner bzw. Teilnehmer ("participants") bezeichnet. Folgende Phasen bzw. Folge von Nachrichten werden durchlaufen bzw. ausgetauscht (siehe Bild 2-11):

In der 1. Phase sendet der Koordinator an alle Teilnehmer ein "prepare" um den bevorstehenden Transaktionsabschluß anzukündigen. Jeder Partner, der mit dem Abschluß einverstanden ist, sendet ein "ready" zurück, alle anderen ein "abort". Der Koordinator erwartet die Antworten und nach Erhalt aller, werden diese analysiert.

Mit dem Ergebnis der Analyse aus Phase 1 startet Phase 2. Befindet sich mindestens eine Ablehnung unter den Antworten, wird an alle Instanzen, die positiv antworteten, ein "abort" gesendet und die Transaktion muß verworfen werden. Haben alle Partner positiv geantwortet, so wird an alle ein "ack" oder "commit" gesendet um die Transaktion endgültig zu beenden.

Je nachdem, ob ein Teilnehmer ein "abort" oder ein "ack" erhält, verwirft oder beendet er die Transaktion. Nach Beendigung der entsprechenden Aktion, sendet er ein "ack" an den Koordinator zurück. Dieser seinerseits sammelt alle Bestätigungen auf und beschließt ebenfalls die Verarbeitung.

Da hier nur eine kurze Darstellung des Verfahrens gebracht werden kann, muß auf die Problematik des Schreibens der EOT-Sätze (EOT = End of Transaction, /REUT81/) oder das Ablaufen von Auszeiten (Timeouts) verzichtet werden.

Das Verfahren birgt allerdings auch Probleme in sich. So können bei einzelnen Partnern **Blockierungen** auftreten, wenn das Netz oder der Koordinator ausfällt und die Instanz sich in seinem sog. "Unsicherheitszustand" befindet (auf sehr übersichtliche Zustands-

diagramme sei auf /CER184/ verwiesen), wenn sie also ihr "ready" abgesendet hat und auf die endgültige Bestätigung ("commit" oder "abort") vom Koordinator wartet. Erfolgt in diesem Zustand keine Benachrichtigung, so kann die Instanz nicht entscheiden, ob die Transaktion gültig ist oder nicht.

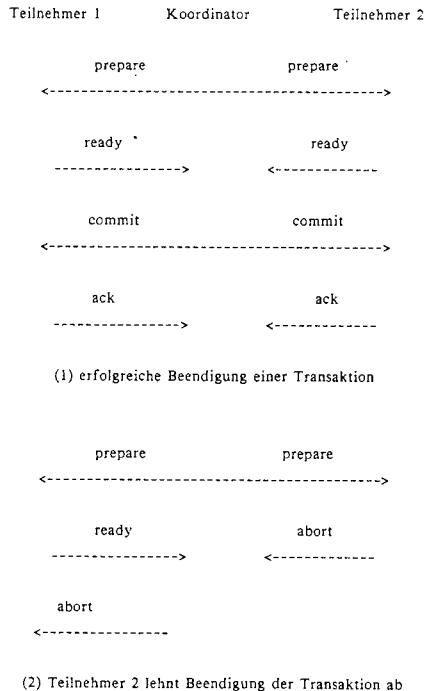


Bild 2-11: 2 mögliche "Two Phase Commit"-Szenarien, /GRAY79/, /CER184/

Für einige Systementwickler war das Verfahren auch zu aufwendig, so daß inzwischen eine Vielzahl von Varianten entwickelt wurden. Es wird bei dem Verfahren implizit angenommen, daß es sich um eine Netztopologie handelt, in der der Koordinator alle Partner erreichen kann, die Partner selbst aber nicht notwendigerweise in der Lage sind miteinander zu kommunizieren. Man spricht dabei von einer "Zentralistischen Kommunikationsstruktur" (/CER184/). Eine spürbare Reduzierung bzgl. der Anzahl der Nachrichten verspricht die Ausnutzung einer Busstruktur, bei der alle Instanzen

linear ("Lineare Kommunikationsstruktur") angeordnet sind, allerdings unter dem Verlust von Parallelität der ausführbaren Aktionen. Weitere nennenswerte Kommunikationsstrukturen sind die "verteilte" in der jeder Teilnehmer mit jedem kommunizieren kann und nicht notwendigerweise ein Koordinator vorhanden sein muß und die "hierarchische".

Für die hierarchische Kommunikationsstruktur konnten durch Vereinfachungen noch weitere Verbesserungen erzielt werden. So wurden im Systementwurf für R^* , einem experimentellen, verteilten Datenbank-Management-System, die zwei Protokolle "Presumed Abort Protocol" (PA) und "Presumed Commit Protocol" (PC) entwickelt (/MOHA86/, /CERI84/). Eines der Argumente, die für eine Optimierung sprechen, sind nach Mohan et. al. die hohen Kosten, die komplexe Verständigungsprotokolle verursachen, obwohl sie nur selten auftretende Fehler beheben und für den Normalfall zu schwerfällig sind. In R^* werden verteilte Datenbankabfragealgorithmen als ein streng hierarchischer Baum von parallelen Prozessen betrachtet, ähnlich wie dies auch in dem später noch zu betrachtenden Konzept des "Transaction Processing" geschieht (siehe Kapitel 6.2.). In einem derartigen Baum wird die Wurzel als der (oberste) Koordinator betrachtet und die Blätter lediglich als Teilnehmer. Die dazwischen liegenden Knoten erfüllen sowohl letztere Funktion gegenüber dem im Baum darüberliegenden Knoten (Wurzel) als auch die Funktion des Koordinators gegenüber den darunterliegenden Knoten (Blätter).

Das PA-Protokoll ist ein pessimistischer Ansatz, bei dem im Zweifelsfall eine negative Entscheidung für die Transaktion herbeigeführt wird. Es verkürzt das übliche Verfahren dadurch, indem der Koordinator nach Feststellen einer fehlerhaften Transaktion allen Partnern, die in der 1. Phase mit einem "ready" geantwortet haben, das "abort" sendet und die Transaktion "sofort vergißt", also auf keine weiteren Bestätigungen wartet. Geht für einen der Partner die letzte Nachricht verloren, so kann er, z.B. nach Ablauf eines Timeouts, annehmen, daß die Transaktion fehlerhaft beendet wurde.

Betrachtet man nun noch insbesondere Transaktionen, bei denen Instanzen nur Leseoperationen (keine Datenänderungen) durchzuführen haben, so können die Prozesse nach dem Absenden der Bestätigungen ("ready") sofort ihr Satzsperrn aufheben, da ihnen egal sein kann, ob die Transaktion erfolgreich war oder nicht.

Beim PC-Protokoll wird genau der umgekehrte Ansatz gewählt indem im Zweifelsfall für die positive Beendigung der Transaktion entschieden wird (optimistischer Ansatz). Es müssen dementsprechend die negativen Nachrichten ("abort") des Koordinators von den Partnerinstanzen bestätigt werden und nicht die "commit". Diese Vorgehensweise geht von der Annahme aus, daß Fehlerfälle nur selten auftreten. Auf diese Weise kann der fehlerlose Ablauf beschleunigt werden. Probleme können allerdings auftreten, wenn der Koordinator, kurz nachdem er die "prepare"-Nachrichten gesendet hat, ausfällt. Nach seinem Wiederanlauf "weiß" er von dieser Aktion nichts mehr, obwohl

alle Partner die Transaktion bestätigen. Diese Inkonsistenz kann durch Festhalten der Namen der Partner vor dem Senden der "prepare"-Nachrichten behoben werden.

Ein gänzlich anderer Weg wird bei der Weiterentwicklung des Zweiphasenfreigabeprotokolls zum Dreiphasenfreigabeprotokoll ("Three Phase Commit"-Protokoll, /BEHG87/, /CERI84/) beschritten. Insbesondere soll die Blockierung von Prozessen vermieden werden, wie sie noch beim Zweiphasenfreigabeprotokoll möglich ist. Die Vorgehensweise ist zunächst bekannt (siehe Bild 2-12):

In der 1. Phase sendet der Koordinator an alle Teilnehmer ein "prepare" um den bevorstehenden Transaktionsabschluß anzukündigen. Jeder Partner, der mit dem Abschluß einverstanden ist, sendet ein "ready" zurück, alle anderen ein "abort". Der Koordinator erwartet die Antworten, welche nach deren Erhalt alle analysiert werden.

Mit dem Ergebnis der Analyse aus Phase 1 startet Phase 2. Befindet sich mindestens eine Ablehnung unter den Antworten, wird an alle Instanzen, die positiv geantwortet haben, ein "abort" gesendet und die Transaktion muß verworfen werden. Auf Grund des "abort" wird von diesen Prozessen die Transaktion ebenfalls verworfen.

Eine Änderung tritt ein, wenn alle Partner positiv geantwortet haben: Es wird jetzt an alle Beteiligten ein "pre-commit" anstatt des "commit" gesendet.

In der 3. Phase warten die Teilnehmer auf das "pre-commit", das sie, je nach dem, ob sie es positiv bestätigen wollen mit einem "ack" oder im negativen Fall mit einem "abort" beantworten müssen. Hat der Koordinator alle "ack" eingesammelt, so sendet er abschließend sein "commit" an alle Beteiligten. War wieder mindestens eine negative Antwort dabei, so verfährt er wie in Phase 2 (senden von "abort") und verwirft die Transaktion.

Der entscheidende Unterschied zum Zweiphasenfreigabeprotokoll liegt in der Möglichkeit für die Partnerinstanzen die Transaktion in der 2. Phase doch noch abzulehnen, obwohl sie sie in der 1. noch bestätigt haben. Durch das Anhängen einer weiteren Phase und damit eines weiteren Warte- (Zwischen-) Zustandes der beteiligten Prozesse, ist eine größere Sicherheit gewährleistet, denn dieser Zwischenzustand wird bei allen Partnern festgehalten und nur wenn dieser bestätigt wird, erfolgt eine gemeinsame Beendigung. Geht in den ersten beiden Phasen eine Nachricht verloren oder es kommt zu einem Ausfall einer Instanz, so kann die Transaktion noch zurückgesetzt (verworfen) werden. Geht in der letzten Phase etwas verloren, so kann die Instanz, die deshalb blockiert wird, sich das abschließende (positive) Ergebnis noch nachholen, da sie sich auf jeden Fall den abschließenden EOT-Satz bereits gesichert hat (weiterer Zwischenzustand) und sie kann dadurch die Transaktion auch zu Ende führen.

Das Verfahren konnte hier nur verkürzt dargestellt werden, auf eine ausführliche Diskussion, vor allem auch der anderen Algorithmen, sei wieder auf /BEHG87/ und /CERI84/ verwiesen.

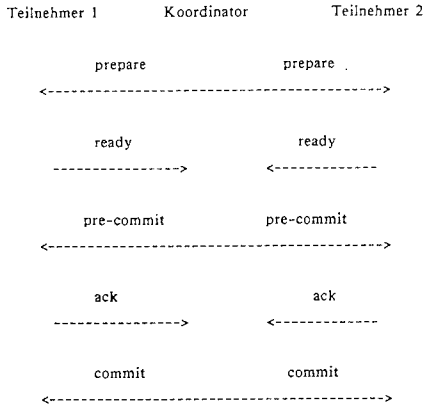


Bild 2-12: Erfolgreiches "Three Phase Commit"-Szenarium, /BEHG87/

Bei der Auswahl eines Verfahrens für diese Arbeit war das entscheidende Kriterium die absolute Sicherheit, die bei der Synchronisierung von Prozessen notwendig ist, insbesondere wenn diese Prozesse zur Steuerung technischer Prozesse eingesetzt werden. Das Dreiphasenfreigabeprotokoll erschien deshalb als die richtige Wahl, auch unter dem Gesichtspunkt des erhöhten Aufwandes. Zielen andere Vorgehensweisen vor allem auf die Optimierung, so stand bei dieser Arbeit die Überlegung, daß beim Botschaftenaustausch zwischen den unabhängigen, verteilten Prozessen ein absoluter Gleichklang notwendig ist, im Mittelpunkt. Da es in dem hier vorgestellten System an sich keine global verfügbaren Daten gibt - Daten liegen immer unter der Obhut einzelner Prozesse (siehe Kapitel 5.6.2.) - und somit der Aspekt der Integrität eines gemeinsamen (globalen) Datenbestandes nicht auftritt, reduziert sich die Aufgabenstellung für ein Kommunikationsverfahren auf den Synchronisationsaspekt, bei gleichzeitigem Datenaustausch. Diese Sichtweise wird auch in Kapitel 5.8. ("Vergleich zwischen Dreiphasenfreigabeprotokoll und Baumverfahren") nochmals aufgegriffen.

2.5. Abschließende Bewertung der vorgestellten Ansätze

Aus dem Vorangegangenen wird klar, daß in verteilten Systemen, also Systemen, in denen Prozesse auf mehrere Rechner verteilt sind, ein Bedarf an Fehlertoleranz besteht. Dabei entstehen die Probleme hauptsächlich durch die physikalische Verteilung der Komponenten und nicht durch die logische Aufteilung in unabhängige Prozesse. Dabei wird in den hier vorgestellten Ansätzen und auch in dem noch folgenden Modell immer von der Kommunikation selbst abstrahiert. Es wird lediglich vorausgesetzt, daß Kommunikationsmedien mit jeweils unterschiedlichen Fehleranfälligkeiten existieren.

Den Aufsätzen ist gemeinsam, daß zuerst ein Kommunikations- und Synchronisationsmechanismus dargestellt oder zumindest angedeutet wird, um dann darauf Rücksetzverfahren zu definieren. Auf Grund dieser Vorgehensweise liegt es nahe, daß sich ein Rücksetzverfahren, das auf einen Kommunikationsmechanismus aufgebaut wird, der auch immer einen Synchronisationsmechanismus voraussetzt, relativ elegant entwickeln läßt. Deshalb wird in der vorliegenden Arbeit auch zunächst ein Prozeßsystem bzw. ein Betriebssystemmodell (Kapitel 3) entwickelt und anschließend ein Kommunikationsmechanismus (Botschaftenprotokoll, Kapitel 4) vorgestellt. Dieser Kommunikationsmechanismus soll die Basis für ein Rücksetzverfahren (Kapitel 5) bilden, das dann relativ einfach eingefügt werden kann.

Bei Anderson/Knight (Kapitel 2.1.) werden die Kommunikation und das Rücksetzverfahren nur sehr schematisch dargestellt. Es werden keine verbindlichen Implementationshinweise gegeben. Vor allem das Festlegen der Rücksetzpunkte scheint nicht sehr durchdacht zu sein.

Anders dagegen Kimm, der von vornherein Kommunikation und Rücksetzbarkeit (das Setzen von Rücksetzpunkten) als eine Einheit sieht. Er versucht dabei den aus CONCURRENT PASCAL bekannten Recovery-Block immer mächtiger, damit natürlich auch komplexer, und dadurch immer besser für das Rücksetzen von Prozessen zu machen. Eine Trennung von Kommunikation und Rücksetzverfahren ist nicht möglich. Der Ansatz hat, dadurch, daß er den Recovery-Block in eine Programmiersprache einbettet, den nicht zu unterschätzenden Vorteil für den Programmierer, daß für ihn ein Rücksetzen deutlich sichtbar ist, also nicht eine im Betriebssystem versteckte Aktion.

Einen gegensätzlichen Weg beschreitet Shrivastava, der die Rücksetzbarkeit als eine Erweiterung des "remote procedure calls (RPC)" versteht, so daß sie dem Programmierer verborgen bleibt. Der Ansatz bietet außerdem den Vorteil, daß er den Zugriff auf Daten (durch den RPC), die Daten selbst und die Prozesse, die den Zugriff verursachen,

als ein System voneinander unabhängiger Schichten sieht. Durch diese Unabhängigkeit werden die Rücksetzfähigkeiten der einzelnen Ebenen unabhängig voneinander. Bei Kimm müssen alle Prozesse den gleichen Rücksetzmechanismus haben, da dieser in die Kommunikation eingebettet ist. Beim RPC ist die Unabhängigkeit auch deshalb leichter zu realisieren, weil schon die "Synchronisationsphilosophie" eine andere ist. Schwieriger dagegen ist es, wenn versucht wird, für alle Objekte (Daten und Prozesse) die gleiche Rücksetzfähigkeit zu erzielen.

Der Weg, der in der vorliegenden Arbeit beschritten wird, geht von einer sehr strengen Prozeß-Prozeß-Kommunikation (siehe /FHKK83/) aus, ähnlich der im Recovery-Block-Ansatz. Für diese Art der Kommunikation wird ein Protokoll entwickelt, das es ermöglicht, ein Rücksetzverfahren zu integrieren, aufzusetzen, ähnlich wie im RPC-Ansatz. Das Rücksetzverfahren bleibt im Betriebssystem verborgen. Es kann aber durch separate Fehlerausgänge bei den Kommunikationsanweisungen, wie sie für Verteiltes PEARL definiert wurden, der Recovery-Fall programmiersprachlich erfaßt werden. Wie bei Shrivastava werden Parallelen zum vorgestellten "Three Phase Commit"-Protokoll (TPCP) bzw. "Two Phase Commit"-Protokoll gezogen und im Gegensatz zu Kimm wird ein zentralistisches Botschaftenprotokoll bevorzugt. Besonders letztere Eigenschaft drängt den Vergleich mit dem TPCP auf und auch im RPC-Ansatz mit der Verwendung der "master-" (Haupt-) und "worker-" (Stellvertreter-) Prozesse ist der Zentralismus bewußt gesucht.

Bei der vorliegenden Arbeit handelt es sich damit um das Umsetzen teilweise bekannter Verfahren und Techniken auf einen strengen Kommunikationsmechanismus, wobei versucht wird, eine möglichst einfache und durchsichtige Implementation zu ermöglichen.

Drei weitere Punkte werden zusätzlich beachtet:

- Die Prozesse sollen in einer Realzeitumgebung ablaufen, also darf das Verhalten der technischen Prozesse nicht vergessen werden.
- Wie PASCAL erlaubt auch die Sprache PEARL gemeinsame ("globale") Daten zu deklarieren. Die soll ebenso in den Sicherheitsüberlegungen bedacht werden.
- Schließlich sollen das Botschaftenprotokoll und das Rücksetzverfahren vom Leitungs- und Verbindungssystem zur physikalischen Übermittlung der Protokollnachrichten und der Daten abstrahieren.

3. Entwicklung eines Betriebssystemmodells

Wie aus dem vorangegangenen Kapitel deutlich wurde, ist es, um über ausfallsichere Systeme diskutieren zu können, notwendig, zunächst Modellvorstellungen bezüglich prozeßunterstützender Laufzeitfunktionen (Betriebssystem) und Kommunikationsmöglichkeiten der Prozesse untereinander (Botschaftenprotokoll) zu entwickeln. Die drei Problemfelder **Betriebssystem**, **Botschaftenprotokoll** und **Rücksetzbarkeit** sind sehr stark abhängig voneinander. Legt man ein Betriebssystem mit ungenügenden Fähigkeiten, Prozesse zu strukturieren, zugrunde, so fällt es relativ schwer, ein Protokoll zwischen diesen "unstrukturierten", schlecht voneinander trennbaren Prozessen zu definieren. Darüberhinaus lassen sich für mehrere Prozesse schwer gemeinsame Wiederaufsetzpunkte für ein Rücksetzen festlegen, wenn die (Reihen-) Folge von Botschaften eines Protokolls nicht eindeutig ist. Da es sich hier um verteilte (Prozeß-) Systeme handelt, soll gerade auf die strikte Strukturierung (Aufteilung) des Gesamtsystems in Prozesse, auf ein durchschaubares Botschaftenprotokoll und daher auf relativ leicht handhabbare Sicherungsmaßnahmen verstärkt Wert gelegt werden.

In diesem Kapitel soll das Betriebssystemmodell vorgestellt werden. Mit Hilfe der nachfolgend aufgeführten Anforderungen und der anschließenden Darstellung von alternativen Betriebssystemkonzepten sollen ein Vorschlag für ein Betriebssystem bzw. Routinen zur Laufzeitunterstützung von Prozessen entwickelt werden.

3.1. Anforderungen an ein Betriebssystem

Ein Betriebssystem soll aus modularen Einheiten, also aus kleinen und überschaubaren Bausteinen, aufgebaut sein. Es soll die Möglichkeit der Erweiterbarkeit (open endness, /BLAA72/) haben, d.h. zusätzliche Betriebssystemfunktionen können nachträglich eingefügt werden. Der Entwurf und die Spezifikation des Betriebssystemmodells müssen so angelegt sein, daß damit Beweise für "Deadlock"- und "Livelock"-Freiheit leicht gestaltet werden können. Schon diese sehr grundlegenden Anforderungen werden von heute noch gebräuchlichen Betriebssystemen nicht oder nur teilweise erfüllt. Dabei ist es gerade im Bereich von Realzeitsystemen und verteilten Systemen wichtig, die Betriebssysteme für neuere Entwicklungen (Erweiterungen) offen zu halten. Da im folgenden vor allem die Fähigkeit der Kommunikation von Prozessen untereinander, aber auch die Einbettung in ein Realzeitsystem, Vorrang haben, werden derartige Funktionen genauer betrachtet. Daraus leiten sich die aufgeführten Anforderungen ab, die hauptsächlich mit der Entwicklung der Programmiersprache Verteiltes PEARL (/FHKK83/) entstanden. Dabei handelt es sich natürlich nur um einen möglichen Ansatz:

- Operationen auf gemeinsamen Daten durch unterschiedliche Prozesse, wenn sich die Daten im gemeinsamen Speicher, auf dem gleichen Rechner, befinden.
- Botschaftsoperationen (im Verteilten PEARL als TRANSMIT und RECEIVE bezeichnet) zur Übertragung von Nachrichten von einem Prozeß zu einem anderen mit Zeitüberwachung, wobei sich die Prozesse nicht notwendigerweise auf verschiedenen Rechnern befinden müssen.
- Nichtdeterministische Kontrollanweisungen, im Verteilten PEARL als "Guarded Statements" oder nach Dijkstra (/DIJK75/) als "Bewachte Anweisungen" bezeichnet, die Botschaftsoperationen (TRANSMIT / RECEIVE) enthalten. Die Operationen können dabei mit logischem UND (AND) und logischem exklusiven ODER (XOR) verknüpft sein. Zusätzlich sollen die Kontrollanweisungen zeitlich überwacht werden können.
- Unterstützen direkter Tasksteuerung (Aktivieren, Beenden und Fortsetzen) und Scheduling (Verwalten von Prozeß-Einplanungen, wie z.B. das Aktivieren zu einem bestimmten Zeitpunkt).
- Implizites Erzeugen von Rücksetzpunkten durch das Betriebssystem, soweit möglich; explizites Erzeugen durch den Benutzer.

- Wiederaufsetzen von einzelnen Prozessen nach sogenannten Software- oder Hardware-Crashes (/WEBE83/) an Rücksetzpunkten. Im einfachsten Fall ist das ein Neustart des Prozesses ("Fault Tolerant System", /RAND78/).
- Kommunikation mit dem Benutzer über eingegebene Kommandos und Fehlermeldungen, etc.
- Dynamisches Konfigurieren im laufenden System (siehe Kapitel 6.1.), wobei zu unterscheiden ist zwischen
 - = Ersetzen eines bestimmten Prozesses durch einen anderen Prozeß oder durch eine andere Version ("replace", /FLEI85/),
 - = Re- oder Umkonfigurieren, d.h. das Löschen eines Prozesses auf einem Prozessor und das erneute Laden auf einem anderen, soweit dies die Prozeßsteuerung erlaubt ("relocation", /KRMA85/; Ansätze dazu gibt es in /IIDS84/).
- Dynamisches Kreieren und Löschen von Prozessen (/ADA83/).

Bevor aus diesen Anforderungen ein Betriebssystemmodell abgeleitet wird, sollen zuvor vergleichbare Ansätze und deren Schwächen dargestellt werden. Dabei wird deutlich, daß diese in der Vergangenheit gewonnenen Ansätze für eine Weiterentwicklung, insbesondere in Hinblick auf Wiederaufsetzgesichtspunkte, kaum einsetzbar sind.

3.2. Das konventionelle Schichtenmodell

Bei vielen Betriebssystemmodellen ist ein Denken in "Schichten" (Ebenen) angebracht, um eine hierarchische Trennung der einzelnen Aufgabenmengen zu erzielen.

In dem hier erläuterten Modell (siehe Bild 3-1, /HERB85/) hat man eine Menge beliebiger Anwenderprozesse, die für sich eine Schicht bilden (Benutzerschicht). Eine 2. Schicht wird vom Betriebssystem gebildet, dessen Aufbau nachfolgend noch genauer erläutert wird. Die 3. Schicht bilden die Treiber, die die direkte Steuerung der Geräte, der Verbindungsleitungen, etc. übernehmen. Wir haben damit drei voneinander getrennte Schichten, die nur über eine begrenzte Anzahl von Schnittstellen Informationen austauschen. Dabei "wissen" die Schichten 1 und 3 nichts voneinander, sondern stehen nur über die Schicht 2 indirekt miteinander in Verbindung. Diese Aufteilung drängt sich förmlich auf, zumal die Entstehung der einzelnen Schichten vollkommen unabhängig voneinander ist.

Von besonderem Interesse soll die Organisation des Betriebssystems sein, die u.a. auf zwei Arten erfolgen kann.

Zum einen liegt, wie in /KUMM83/ geschildert, ein einziger Betriebssystemblock, von Dijkstra als "Sekretär" bezeichnet (/WETT84/, /DIJK71/, /HERB85/), vor. Die Hauptkritik an diesem Ansatz liegt in der Unübersichtlichkeit des Systemblocks, in dem alle Funktionen, auch wenn sie nichts miteinander zu tun haben, zusammengefaßt sind. Wie Erfahrungen (/KUMM83/) zeigten, bedeuten Erweiterungen dieses Betriebssystems ein sehr schwieriges Unterfangen. Eine Erweiterung des Systems, vor allem in Hinblick auf verteilte Systeme (siehe auch die Einleitung zu Kapitel 4), scheiterte fast an dem sehr festgefügt und unübersichtlichen Aufbau des Blocks.

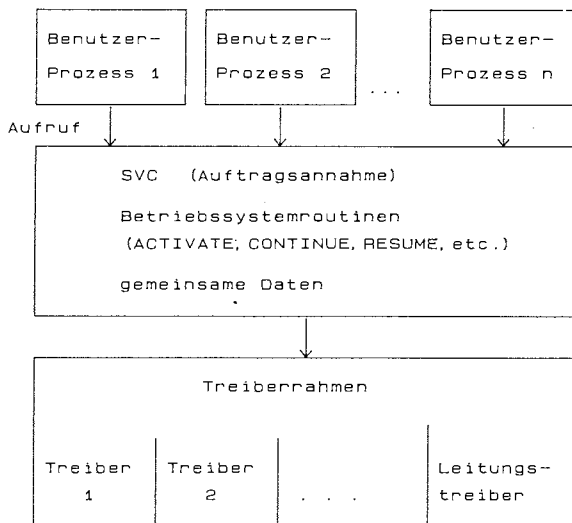


Bild 3-1: Das Betriebssystem als konventionelles Schichtenmodell

Eine zweite Organisationsform für das Betriebssystem wäre eine Struktur, bestehend aus Monitoren und abstrakten Datentypen, die durch ihre Modularisierung wesentlich flexibler wäre als die erste Form. Bild 3-2 zeigt den Aufbau eines solchen Modells, das man in Anlehnung an das 1. Modell als einen "modularisierten Sekretär" vertehen könnte. Zwischen den Monitoren sollen hierarchische Beziehungen gelten. Trotz der Übersichtlichkeit sind Nachteile in Kauf zu nehmen:

- 1) Mit der Forderung nach der hierarchischen Struktur treten Probleme auf:
Wie kann die Hierarchie durchgehalten werden, und wie können Schwierigkeiten in Zusammenhang mit den sogenannten "Nested Monitor Calls" (Verschachtelte Monitornaufrufe, /HADD77/) umgangen werden.
- 2) Die Umsetzung (Spezifikation) der Guarded Statements (-Aufträge) der Benutzerprozesse in eine Abwicklung durch Monitore ist kompliziert, da verschiedene Prozesse über die gleichen Monitorprozeduren ihre Aufträge abwickeln.

- 3) Das in dieser Arbeit gesteckte Ziel, zur Spezifikation die Methode PASS, also eine Methode mit Prozessen als Strukturierungselementen, zu verwenden, kann nicht eingehalten werden. Damit fehlt ein durchgängiges Konzept zur Darstellung aller Bestandteile des Systems, denn das Programmsystem soll als eine Menge von Prozessen gesehen werden.

Durch die Aufteilung des Betriebssystemblocks gewinnt man zwar an Übersicht, aber der Zugriff auf Betriebssystem- und Verwaltungsdaten wird erschwert, obwohl die Daten alle in einer zusammengehörigen Schicht liegen. Der Nachweis der Deadlock-Freiheit der Botschaftenabwicklung ist schwierig (siehe 1) und 2)).

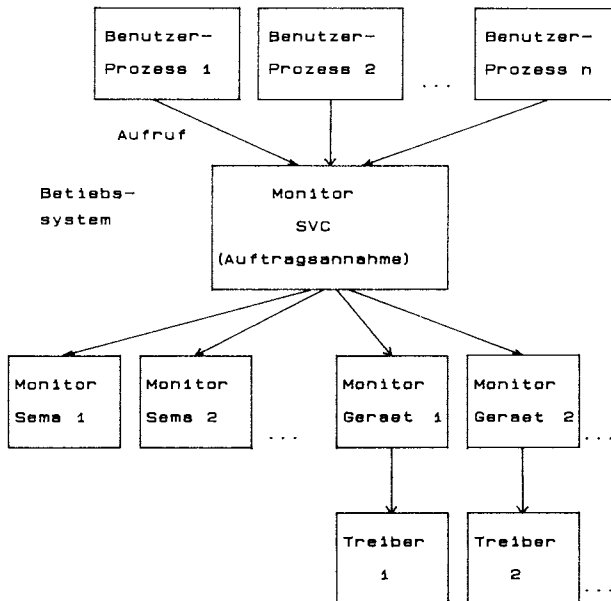


Bild 3-2: Das Betriebssystem, bestehend aus Monitoren

3.3. Das Betriebssystem als ein Prozeßmodell

War bei dem bisher vorgestellten Modell eine strikte Trennung zwischen Benutzer-, Betriebssystem- und Treiberschicht vorherrschend, so wird jetzt der Versuch unternommen, diese strikte Trennung in Schichten aufzuheben. Es bleibt zwar weiterhin das Verhältnis zwischen Auftraggeber (Benutzerprozesse) und Auftragnehmer bzw. Abwickler (Betriebssystem und Treiber) bestehen, doch wird die Art der Auftragserteilung in eine andere Form anstatt der bisherigen strengen Hierarchie gebracht. Hinzu kommt, daß das Betriebssystem, das nur aus einem festgefügteten, unflexiblen Block bestand, aufgelockert und aufgeteilt wird. Diese Vorgehensweise kann damit begründet werden, daß nicht nur ein bestimmter Botschaftsalgorithmus (siehe Kapitel 4) ins Auge gefaßt wird, sondern auch ein Rücksetzverfahren eingefügt werden soll (siehe Kapitel 5.5.). Da sich letzteres nicht nur auf die Benutzerprozesse bezieht, sondern auch auf das (aufgeteilte) Betriebssystem, erscheint ein geschlossener Block, der noch dazu alle Benutzeraufträge abwickeln soll, zu starr.

Die Vorstellung des (Betriebssystem-) Konzepts soll in drei Stufen erfolgen. Zunächst wird die Aufteilung des Systems (in Prozesse) erläutert, die als "abstraktes Modell" bezeichnet wird. Danach wird auf die Einbettung in ein Rechensystem (Rechner, Rechnernetz) eingegangen ("konkretes Modell") und schließlich werden noch Implementationshinweise gegeben ("Der Prozeßumschalter für die Benutzer- und Systemprozesse", "Implementiertes System").

3.3.1. Abstraktes Modell

Es darf zunächst vorausgesetzt werden, daß das Benutzerprogramm in Prozesse aufgeteilt oder zumindest als ein in sich geschlossener Prozeß verstanden werden kann.

Beim Betriebssystem wird folgende Denkweise eingeführt:

Programme, Daten, Geräte, etc., denen ein "eigenständiges (aktives) Verhalten" unterstellt werden kann, werden als "Prozesse" definiert. Dazu gehören die meisten Teile des Betriebssystems: Verwaltungsprozesse für Geräte, Semaphore, Prozeß-Signale, Zeitsignal, usw. Die so erzielte Aufteilung bewirkt, daß es letztlich im Verhalten nach außen keine Unterscheidung mehr zwischen Benutzer- und Systemprozessen gibt.

Der innere Aufbau soll bei Benutzer- und Systemprozeß ebenfalls gleich sein:

- | | | |
|--|---|---|
| - Name | } | Bekannt als Prozeß-,
oder
Gerätekontrollblock |
| - Verwaltungsdaten (z.B. Programm-Semaphor-
oder Kellerzeiger) | | |
| - Lokale Daten, soweit vorhanden (z.B. vom Entwickler definierte Prozeßdaten);
Initialwerte | | |
| - Programm (Verwaltungsroutrinen zur Semaphore- oder Gerätekontrolle, Benutzerprogramm). | | |

Eine weitere Art von Prozessen bilden die "Treiber", also diejenigen Programme, die direkt Geräte betreiben oder steuern. Ihre Ansprechpartner sind einerseits die Geräteteilungsverwaltungsprozesse und andererseits die Geräte selbst. Vorstellbar wäre auch eine Zusammenfassung von Treiber und Verwaltungsprozeß. Die Trennung erscheint aber sinnvoll, da in letzterem alle anlagenunabhängigen und im Treiber alle anlagenabhängigen Teile zusammengefaßt werden können. Da bei einer etwaigen Portierung des Gesamtsystems immer nur anlagenabhängige Teile ausgetauscht werden müssen, reduziert sich das Auswechseln auf die Treiberprozesse. Der innere Aufbau der Treiberprozesse kann gleich denen der Benutzer- und Systemprozesse sein.

Eine Sonderstellung im Gesamtsystem nehmen die "globalen Daten", wie sie z.B. in PEARL formuliert werden können, oder Dateien ein. In beiden Fällen ist ein beliebiger Zugriff durch Prozesse möglich, der vom Programmierer durch die Verwendung von Synchronisationsmechanismen (Semaphore) geregelt werden kann. Das bedeutet für das Modell zunächst einmal, daß sowohl globale Daten als auch Datei(deklaration)en frei verfügbar sind und lediglich passive Datenbestände darstellen. Sie spielen damit keine besondere Rolle. Erst bei den Überlegungen für die Wahl von Wiederaufsetzpunkten im Rahmen eines Sicherungsmodells wird für die globalen Daten ein Verwal-

tungsprozeß eingeführt (siehe Abschnitt 5.6.2.) und damit gilt wieder das für Prozesse allgemein Gesagte.

Insgesamt bedeutet dies, daß wir es nur noch mit einer Vielzahl von selbständigen (Benutzer-, System- und Treiber-) Prozessen zu tun haben, im Gegensatz zu vorher, wo wir nur die Benutzerprozesse und als Dienstleistungserbringer einen Systemprozedurblock hatten. Die große Menge an Prozessen mag eventuell stören, doch ein großer, unhandlicher Block erleichtert die Übersicht auch nicht.

Was wir bisher erreicht haben ist eine Zerlegung des Gesamtsystems in Prozesse, bestehend aus:

- Benutzerprozessen;
- Prozessen zur Verwaltung von Semaphoren;
- Prozessen zur Verwaltung von Geräten;
- Prozessen zur Steuerung von Geräten (Gerätetreiber);
- Prozessen zur Verwaltung von Unterbrechungssignalen ("Interrupts");
- Prozessen zur Aufnahme von Unterbrechungssignalen ("Interrupts", Unterbrechungstreiber);
- Prozessen zur Verwaltung von Zeitsignalen;
- Prozessen zur Aufnahme von Zeitsignalen (Uhrtreiber).

Diese Liste läßt sich, je nach Systemausbau oder Programmierumgebung, beliebig verlängern und natürlich auch verkürzen.

Die Prozesse können nicht isoliert voneinander existieren, sondern müssen Aufträge und Fertigmeldungen miteinander austauschen, sie müssen miteinander "kommunizieren". Wie das Zusammenspiel der verschiedenen Prozesse untereinander aussehen kann, zeigt folgendes Beispiel am Fall der Konsolenausgabe:

Beispiel: (Siehe Bild 3-3) Dem Verwaltungsprozeß für die Konsolenausgabe wird ein Ausgabeauftrag von einem Benutzerprozeß erteilt und dieser nimmt ihn an. Die Art und Weise der Auftragserteilung wird unten erläutert. Falls das Gerät frei ist, wird der Treiberprozeß mit dem dazugehörigen Auftrag (Treiberauftrag) gestartet. Ist die Ausgabe beendet, wird dies mittels einer Fertigmeldung vom Treiber dem Verwaltungsprozeß angezeigt. Dieser wiederum erstellt eine Fertigmeldung für den Benutzerprozeß, der dann in seinem Programm fortsetzen kann.

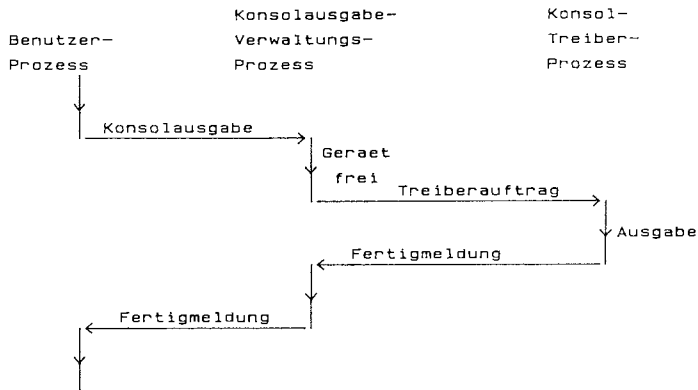


Bild 3-3: Konsolenausgabeauftrag

Im "zentralen" Bild 3-4 wird eine sogenannte Kommunikationsstruktur (siehe /FLEI84/) gezeigt. Die Prozesse werden dabei als Rechtecke gezeichnet und die "Kommunikationen", z.B. die Aufträge und Fertigmeldungen aus Bild 3-3, als Pfeile. Letztere werden im folgenden auch als "Nachrichten" oder "Botschaften" bezeichnet. Dieses Bild verdeutlicht, daß zum Gesamtsystem nicht nur die Prozesse allein gehören, sondern ebenso die Botschaften als Kommunikationsmittel zwischen den Prozessen. Dabei soll es auch einen direkten Informationsaustausch durch Kommunikation zwischen Benutzerprozessen geben (TRANSMIT/RECEIVE, GUARDED STATEMENT). Wie diese Botschaften realisiert werden, wird in den nachfolgenden Abschnitten erläutert.

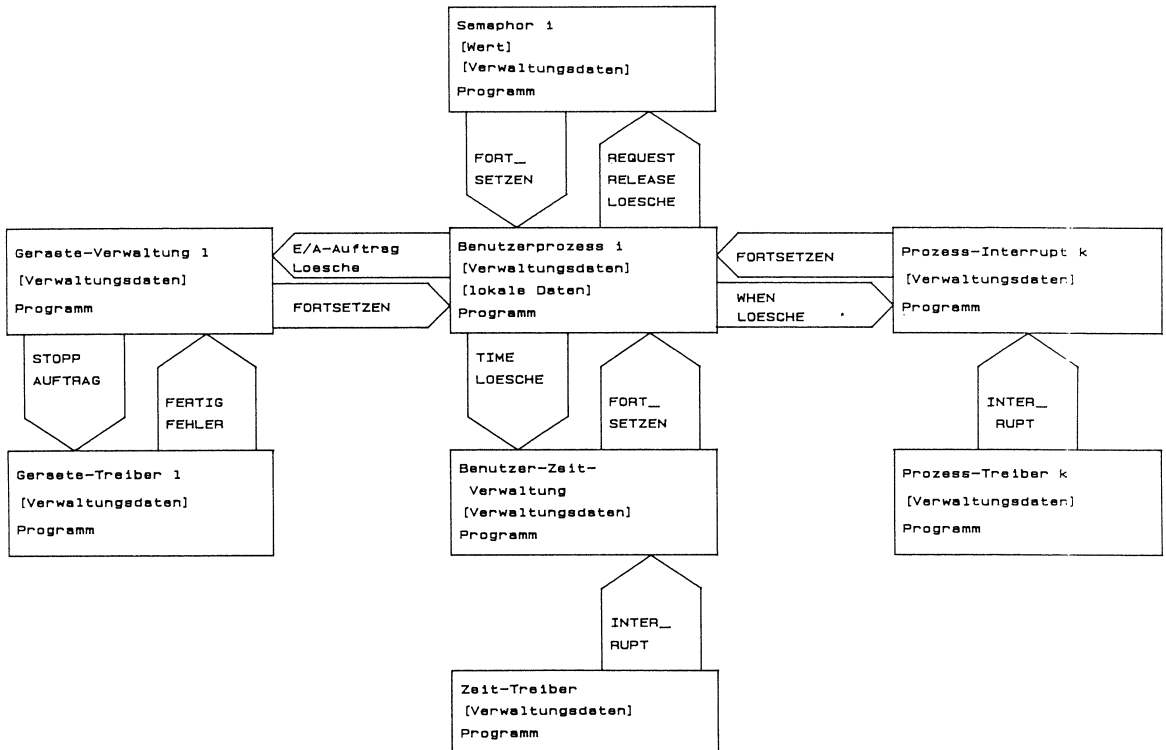


Bild 3-4: Kommunikationsstruktur des "Abstrakten Modells"

Hinzuzufügen ist lediglich noch eine Aussage über die "Art" des Botschaftenaustausches. Dabei wird prinzipiell zwischen einer "synchrone" und einer "asynchrone" Kommunikation unterschieden.

Definition: Unter "synchrone" Kommunikation versteht man das Übermitteln einer Botschaft von einem Sender zu einem Empfänger unter der Voraussetzung, daß beide gleichzeitig zum Austausch der Information bereit sind. Ist einer der beiden noch nicht bereit, bleibt der andere blockiert.

Definition: Bei der "asynchrone" Kommunikation muß die Gleichzeitigkeit der synchrone Kommunikation nicht gegeben sein. Der Sender kann seine Botschaft früher abgeben, als der Empfänger sie erwartet. Sie muß dann zwischengepuffert werden.

In /FHKK83/ wird deshalb ein sog. "Wartebereich" ("BUFFER") mit Größenangabe eingeführt. Ist dieser gefüllt, wird der Sender wieder blockiert. Der Empfänger kann für ihn bestimmte Nachrichten, soweit vorhanden, entnehmen. Senden und Empfangen sind damit zeitlich entkoppelt.

Bei der Kommunikation zwischen den Benutzer- und Systemprozessen bzw. den System- und Treiberprozessen wollen wir grundsätzlich eine synchrone Kommunikation festlegen, um eine möglichst genaue Abbildung auf die Vorgänge "Auftragserteilung - Auftragserfüllung" zu erreichen. Die Kommunikationen der Benutzerprozesse untereinander legt der Programmentwickler fest.

Im Anhang 1 befindet sich eine Spezifikation dieses Systems, also eine Darstellung der Prozesse. Dort sind auch die Bezeichnungen und Namen der Abbildung, insbesondere die Botschaften, näher erläutert.

Dieses abstrakte Modell spiegelt gewissermaßen die Anwendersicht des Systems wider. Im nächsten Abschnitt wird auf die Betriebssystemseite (konkretes Modell), und damit auf die Einbettung der Prozesse und Botschaften auf einen vollständigen Rechner (Rechensystem), eingegangen.

3.3.2. Konkretes Modell

Das in Abschnitt 3.3.1. skizzierte abstrakte Modell besteht lediglich aus den Komponenten:

- Prozesse
- Botschaften (als Kommunikationsmittel zwischen den Prozessen).

Diese beiden Mengen sollen auf einem Rechner (Rechenanlage) konkretisiert werden. Dabei laufen die Prozesse nicht isoliert auf einem Rechner ab, sondern es findet auch eine Kommunikation mit der Außenwelt statt. Wie bereits erwähnt, kann diese über Geräte (z.B. Konsole) oder Prozesssignale (Interrupts) stattfinden. Da wir auch die Kommunikation der Benutzerprozesse untereinander ermöglichen wollen, und zwar nicht nur auf einem gemeinsamen Rechner, muß es die Möglichkeit geben, über Verbindungswege (Leitungen) Daten von einem Rechner zu einem anderen zu transportieren. Es werden also "verteilte Systeme" betrachtet.

Definition: Unter "verteilten Systemen" verstehen wir (über Leitungen) lose gekoppelte Monorechnersysteme ohne gemeinsamen Speicher.

Bei der Realisierung der Prozesse selbst gibt es keine besonderen Punkte zu beachten, da sie, wie im vorhergehenden Abschnitt dargestellt, aus Daten und Programmen bestehen. Sie sind also in sich ablauffähig.

Die Kommunikation mit "ihrer Außenwelt" soll nur über Botschaften erfolgen. Bei deren Realisierung (lokal und über Verbindungen) sind zwei Anforderungen zu erfüllen:

- 1) Die Prozesse "sprechen" die Botschaften über Prozeduraufrufe "an". Es existieren also Routinen, mit denen Botschaften zusammengestellt werden können.
- 2) Das Senden und Empfangen erfolgt mit Hilfe einer sog. "**Botschaftenübermittlung**": Die von den Prozessen erstellten Botschaften müssen übertragen, also vom Sender zum Empfänger transportiert werden. Dabei bedient man sich zum einen eines Protokolls, auf das in Kapitel 4 eingegangen wird. Zum anderen müssen die Botschaften und darin enthaltene Daten physikalisch transportiert werden, wobei es primär egal ist, ob der Transport innerhalb des Rechners oder über eine Leitung stattfindet. Dies wird in Kapitel 7 näher untersucht.

Die Übermittlung läßt sich zwischen dem Senden und dem Empfangen von Botschaften unterscheiden, was in Abschnitt 3.3.4. konkretisiert wird.

Zusätzlich müssen gewisse Dienstleistungen zur Verfügung stehen:

- 3) **Speicherverwaltung:** Wie in jedem Betriebssystem muß der noch nicht belegte Speicherplatz verwaltet werden. Welchen Aufwand man dabei betreiben möchte, bleibt dem Implementierer überlassen. Im einfachsten Fall wird eine sogenannte **"Speicherzuteilung"** realisiert. Sie stellt durch eine minimale Funktionenmenge, z.B. in Form von Prozeduren, das Verwalten von vorgegebenen Pufferplätzen für Botschaften zur Verfügung. Im Prinzip müssen die Plätze nur gesucht, belegt und, wenn nicht mehr benötigt, wieder freigegeben werden.
- 4) Eine Menge reentrantfähiger, gemeinsamer Prozeduren aller oder zumindest mehrerer Prozesse soll aus Effizienzgründen in einem gemeinsamen Bereich liegen. Dazu gehören vor allem Routinen zum Belegen und Freigeben von Pufferplätzen, zum Verwalten von Warteschlangen und Kellern.

Diese Ansammlung von Dienstleistungen kann nicht als selbständiger Prozeß formuliert werden. Würde man sich z.B. die Botschaftenübermittlung in der Form eines Prozesses vorstellen, so ergäbe sich hierbei das Problem der "Doppelreferenzierung":

Bei der Prozeßkommunikation gibt es eine eindeutige Beziehung "Sender-Botschaft-Empfänger", wobei sowohl Sender als auch Empfänger jeweils eine Menge sein dürfen. Ist die Botschaftenübermittlung ein Prozeß, ein Leitungsprozeß, so muß der Sender in einer 1. Adressierungsstufe zuerst den Leitungsprozeß nennen, da dieser zunächst die Botschaft empfangen muß und in einer 2. Stufe den eigentlichen Empfänger. Außerdem wird der Übermittlung in Form einer Botschaft der Sendewunsch mitgeteilt: "Sender-Sendewunsch(Sender-Botschaft-Empfänger)-Leitungsprozeß". Es entsteht dadurch eine Botschaft in der Botschaft. Dieses Verfahren ist zu undurchsichtig, da vor allem die gewünschte eindeutige Beziehung Absender-Botschaft-Empfänger verloren geht. Eine ähnliche Problematik ergibt sich auch bei der Speicherzuteilung.

Die genannten Komponenten (Dienstleistungen) stellen sozusagen die "Basis" (Betriebssystem-Kern) des Systems dar und setzen nur noch die Hard- und Firmware, z.B. das Leitwerk oder die Befehlsentschlüsselung, voraus.

Vom bisher Gesagten kann folgende Zusammenfassung gegeben werden: Selbständige Prozesse tauschen mit Hilfe von Botschaften (Botschaftsprozeduren) Informationen aus. Die Botschaftsprozeduren und auch die Prozesse haben Zugriff auf elementare Routinen (Dienstleistungsprozeduren) zur Speicherverwaltung, zum Transport der

Botschaften, usw. Das Bild 3-5 zeigt diese 3 Schichten. Es erübrigt sich zu sagen, daß die unselbständigen Schichten (Prozeduren) "reentrant"-fähig sind und über keine eigenen Daten, außer lokalen Daten, verfügen. Sie haben also keine "Merkfähigkeit", denn diese wird nur den Prozessen zugestanden.

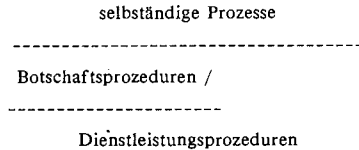


Bild 3-5: Schichtung des "Konkreten Modells"

Im konkreten Gesamtsystem fehlt nur noch ein Baustein, um Prozesse ablaufen lassen zu können: der Prozeßumschalter. Das heißt, daß das "Rest"-Betriebssystem sich auf den Prozeßumschalter reduziert. Die oben genannten Funktionen (Speicherverwaltung etc.) gehören zwar auch zum Betriebssystem, sind aber nicht eigenständig, sondern nur prozedural vorhanden.

Aus der Aufgabe mit Hilfe des dargestellten Systems technische Prozesse zu überwachen und zu steuern, folgt, daß ein System benötigt wird, mit dem man möglichst schnell auf externe Ereignisse reagieren kann. Es entsteht deshalb die Notwendigkeit, die Treiberprozesse von den übrigen Prozessen getrennt zu betrachten. Da die Treiber unabhängig voneinander und unabhängig von allen anderen Prozessen arbeiten und auf Unterbrechungssignale (externe Ereignisse) prompt reagieren müssen, gelten folgende Forderungen:

- sofortiges Erfassen einer Unterbrechung;
- Bearbeiten der Unterbrechung, unabhängig von allen anderen Treibern und Prozessen;
- Fortsetzen des unterbrochenen Programms nach Bearbeitung der Unterbrechung.

Aus dieser Liste läßt sich eine Forderung ableiten: ein Treiber muß jederzeit andere Treiber und Prozesse unterbrechen können ("gegenseitige Unterbrechbarkeit"), um das Unterbrechungssignal registrieren zu können. Danach wird der höchstpriorisierte Treiber (wichtigste Unterbrechung) fortgesetzt. Daraus resultiert die Notwendigkeit, für alle Treiber einen eigenen Prozeßumschalter einzuführen.

Diese Zusammenhänge zeigen, daß es zwei Klassen von Prozessen und deshalb zwei verschiedene Prozeßumschalter geben muß. Sowohl die Benutzer- und Systemprozesse als auch die Treiberprozesse bedienen sich eines eigenen Prozeßumschalters.

Die Entwurfsentscheidung, die Systemprozesse zum Verwalten und die Treiberprozesse zum Betreiben der Geräte zu trennen, wird durch diese Feststellungen nochmals bestätigt.

Aufgrund dieser "Klassenunterschiede" und aufgrund der unterschiedlichen "Wichtigkeit" der Klassen können wir Prioritäten festlegen. Das in Bild 3-6 gegebene Schalenmodell spiegelt diesen Sachverhalt wieder:

- Als Kern bzw. als Innerstes sieht man den Prozeßumschalter für die Treiberprozesse, im folgenden kurz als Treiber-PU bezeichnet. Er bildet die Basis für
- die Treiberprozesse, kurz Treiber, die die zeitlich problematischen Prozesse sind, da sie möglichst schnell auf externe Ereignisse reagieren müssen.
Diese beiden Schalen sind dem eigentlichen Programmsystem unterlegt. Dessen Basis ist wiederum
- der Prozeßumschalter, diesmal für die System- und Benutzerprozesse, kurz Prozeß-PU. Dieser verteilt die Prozessorzeit zwischen
- den Benutzer- und Systemprozessen, die im folgenden als die Prozesse bezeichnet werden.

Die Problematik der Prozessorvergabe soll nachfolgend noch genauer untersucht werden:

Der Prozeßumschalter für die Treiber (Treiber-PU) hat zunächst die gegenseitige Unterbrechbarkeit der Treiber zu realisieren. Ist z.B. ein Unterbrechungssignal erzeugt worden, das eine höhere Priorität als das momentan von einem Treiber bearbeitete hat, so startet der Treiber-PU den Prozeß für die wichtigere Unterbrechung. Ist diese Bearbeitung beendet, wird wieder der niedrigprioritäre Treiber fortgesetzt.

Als weitere Aufgabe des Treiber-PU ist die Unterbrechung des Prozeßumschalters für die Prozesse oder die Unterbrechung der Benutzer- und Systemprozesse zu nennen. Dies geschieht genau dann, wenn irgendeine Unterbrechung eintrifft.

Sind alle Unterbrechungssignale bearbeitet, wird grundsätzlich der Prozeß-PU fortgesetzt (siehe auch Abschnitt 3.3.3.).

Die Treiber haben die Aufgabe, die Unterbrechungssignale, die von den Standard-Ein-/Ausgabe-Geräten, dem Zeitgeber oder den technischen Apparaturen kommen, zu verarbeiten, und gegebenenfalls die Ergebnisse an die Systemprozesse über Botschaften weiterzugeben. Außerdem müssen die über die Geräteverwaltungsprozesse erhaltenen Benutzeraufträge bearbeitet werden. Für diese Fälle erhalten die Treiber den Prozessor zugeteilt.

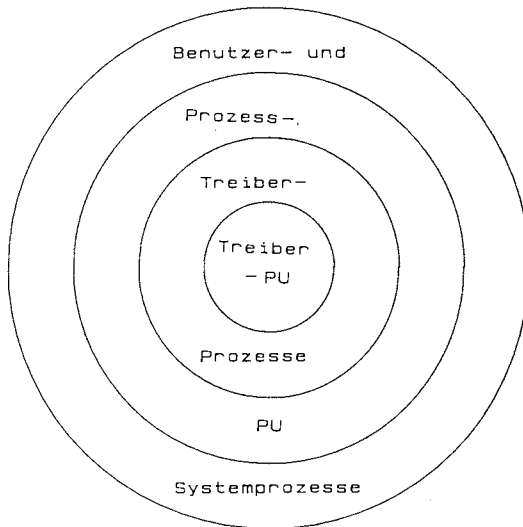


Bild 3-6: Schalenmodell des "Konkreten Modells"

3.3.3. Der Prozeßumschalter für die Benutzer- und Systemprozesse

Der Prozeßumschalter für die Prozesse (Prozeß-PU) hat demjenigen Prozeß den Prozessor zuzuteilen, der

die höchste Priorität hat und

- entweder eine Botschaft erhalten hat
- oder ablauffähig ist.

Es wird einem Prozeß also auch dann der Prozessor zugeteilt, wenn er nicht ablauffähig ist, und zwar dann, wenn er eine Botschaft erhalten hat. Damit ist gewährleistet, daß Botschaften ("Anfragen", siehe Kapitel 4) prompt beantwortet werden.

Um für die Prozessorvergabe die notwendigen Informationen zur Verfügung zu haben, benötigt der Prozeßumschalter eine Tabelle (siehe Tabelle 3-1). In dieser sind alle vorhandenen Prozesse aufgelistet und zwar in der Reihenfolge ihrer Prioritäten. Danach lassen sich 3 Prioritätsgruppen unterscheiden:

- Treiber,
- Systemprozesse und
- Benutzerprozesse.

Die Reihenfolge der Prozesse innerhalb der Gruppen ist dem Systementwickler oder Programmierer überlassen. Die Reihenfolge der Gruppen erscheint in der angegebenen Weise zweckmäßig: Die Treiber müssen höchste Priorität haben. Die Systemprozesse sollten höhere Priorität als die Benutzerprozesse haben, um einen reibungslosen Ablauf des Gesamtsystems zu ermöglichen.

Weitere Eintragungen in der Tabelle sind: der Zustand eines Prozesses (lauffähig, beendet, etc.), Adressen von Verwaltungsteilen (z.B. Adresse des Taskkontrollblocks) und Eintragungen über erhaltene Botschaften. Zu letzteren zählen u.a. auch Treiberfertigmeldungen und Zähler für eine Zeitnahme. Dadurch soll die Zeitüberwachung der Botschaften, wie sie in Kapitel 5.3. erläutert wird, verwaltet werden.

P	N	S	A	B
Priorität = Nummer	Name	Status	Adresse	Botschaften
	Treiber 1			
	Treiber 2			
	:			
(größere Zahl = geringere Priorität)	System- prozeß 1	(lauffähig, Prozesshaft _erwar- tend, etc. =	(Adresse des der zeß- kon- troll- blocks)	(entweder Adresse erhaltenen Bot- schaft oder Zähler für die interne Zeit- überwachung)
Benutzer- prozeß 1	Bot- System- prozeß 2	Zu- stand eines Pro- zesses)		
Benutzer- prozeß 2	:			

(Jede ausgesendete Botschaft wird zeitüberwacht, um festzustellen, ob die Antwort rechtzeitig eintrifft (siehe Kapitel 5.3.).)

Tabelle 3-1: Prozeß-Tabelle des Prozeß-PU

Mit Hilfe dieser Tabelle lassen sich die Aktionen, die vom Prozeß-PU sequentiell durchlaufen werden, so formulieren:

Der Prozeß-PU

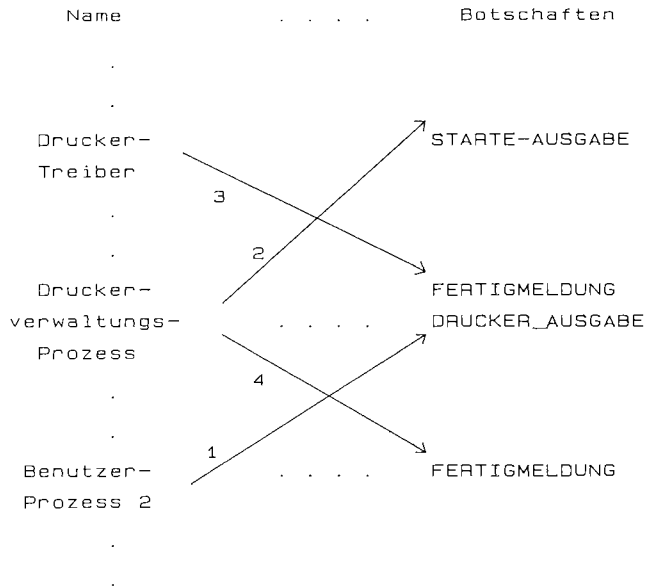
- wird aktiviert durch
 - = den Treiber-PU, wenn eine Botschaft über eine Gerätefertigmeldung oder ähnliches vorliegt oder
 - = einen Benutzer- oder Systemprozeß, wenn dieser eine Botschaft abgesendet hat und deshalb auf Antwort warten muß;
- wählt denjenigen Prozeß mit höchster Priorität aus, für den eine Botschaft eingetroffen ist oder wählt denjenigen Prozeß mit höchster Priorität aus, der ablauffähig ist;
- gibt den Prozessor an den ausgewählten Prozeß ab;
- oder läuft in einer Leerschleife, wenn kein Prozeß fortsetzbar ist.

Beim Prozeßumschalter handelt es sich damit um eine passive Prozedur, die aufgerufen wird, um nach dem höchstpriorien ablauffähigen Prozeß zu suchen und um diesen fortzusetzen. Das läßt sich noch einmal anhand eines Beispiels zeigen:

Beispiel: (Siehe auch Bild 3-7) Der Benutzerprozeß 2 wünscht eine Ausgabe auf Drucker.

Er sendet die Botschaft DRUCKER_AUSGABE mit Hilfe der Botschafts-prozeduren an den Druckerverwaltungsprozeß. Dabei wird die Botschaft in der Spalte B der Tabelle eingetragen. Danach wird der Prozeß-PU gestartet. Dieser stellt fest, daß keiner der Systemprozesse lauffähig ist, sonst wäre Benutzerprozeß 2 erst gar nicht tätig gewesen, und, daß alle auf Botschaften (Task-Aufträge) oder Treiberfertigmeldungen warten. Da für den Druckerverwaltungsprozeß ein Auftrag vorliegt, kann dieser fortgesetzt werden. Der wiederum sendet den Auftrag weiter an den Treiber, soweit dieser frei ist, und geht wieder in den Wartezustand (auf Botschaften wartend) über. Ist die Ausgabe beendet, erzeugt der Treiber eine Fertigmeldung für den Druckerverwaltungsprozeß. Der Treiber-PU startet daraufhin auf jeden Fall den Prozeß-PU und dieser setzt den Druckerverwaltungsprozeß fort, usw.

Tabelle:



- 1 : Benutzerprozeß sendet Ausgabewunsch mit den Daten an den Druckerverwaltungsprozeß
- 2 : Druckerprozeß aktiviert Ausgabe durch den Treiber
- 3 : Fertigmeldung des Treibers an den Druckerprozeß
- 4 : Botschaft des Druckerverwaltungsprozesses an den Benutzerprozeß, daß die Ausgabe beendet ist

Bild 3-7: Beispiel für die Arbeitsweise des Prozeß-PU

3.3.4. Implementiertes System

Ausgespart wurde bisher die Arbeitsweise der in Kapitel 3.3.2. ("Konkretes Modell") eingeführten Botschaftenübermittlung, die einerseits nicht als Prozeß realisiert ist, andererseits ein realitv "eigenständiges Leben" führt.

Benutzer-, System- und Treiberprozesse erstellen Aufträge bzw. deren Antworten in Form von Botschaften. Diese werden von den Botschaftsprozuren entweder direkt in die Tabelle des PU- Prozesse eingetragen oder an einen sogenannten Leitungstreiber (Ausgabe) weitergereicht. Dieses Weiterreichen geschieht über globale Daten des Treibers. Dies ist die einzige Stelle im System, an der zwei Prozesse (Benutzerprozeß und Ausgabeteil des Leitungstreibers) nicht über Botschaften miteinander kommunizieren. Der Treiber wird dabei bereits logisch als Teil der Leitung betrachtet, wie dies auch in Kapitel 7 dargestellt wird. Dieses Übergeben der Daten an den Treiber ist für den Senderprozeß bereits der Beginn der Übertragung und etwaige nachfolgende Fehler beim Treiber gelten als Übertragungsfehler. Der Treiber wickelt also selbständig das Senden der Botschaft ab. Er erhält den Prozessor über den Treiber-PU und gibt ihn über diesen wieder ab.

Ähnlich verhält es sich beim Empfangsteil des Leitungstreibers, der ebenfalls als Teil des Übertragungsmediums aufgefaßt wird. Auch wenn er eine Botschaft vollständig empfangen hat, so gilt sie noch nicht als beim Empfänger abgeliefert, sondern erst wenn er in der Tabelle des Prozeß-PU die Nachricht eingetragen hat. Anschließend werden der Treiber-PU und über diesen schließlich der Prozeß-PU aktiviert.

Der Informationsfluß beim Senden und Empfangen läßt sich damit so darstellen:

Senden : Benutzer-Prozeß (Botschaftenübermittlung) -->
Leitungstreiber (Ausgabe)

Empfangen : Leitungstreiber (Eingabe) --> Prozeß-PU
(--> Benutzer-Prozeß (Botschaftenübermittlung))

Wie bereits mehrfach erwähnt, sind die Treiberprozesse von den anderen Prozessen logisch getrennt, da sie anlagenabhängig sind und höhere Priorität haben. Deshalb wird, um eine Vereinheitlichung zu erzielen, der in /HERB85/ erläuterte "Treiberrahmen" eingeführt. In diesen werden neben den Treibern der Treiber-PU und auch die für alle Treiber gemeinsamen Initialisierungsroutinen angesiedelt (siehe Bild 3-8). Dieser Rahmen soll es ermöglichen, daß Treiber mühelos hinzugefügt ("Einhängen in den Rahmen") und auch wieder entfernt werden können ("Aushängen aus dem Rahmen").

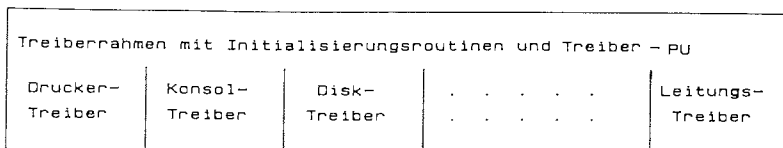


Bild 3-8: Treiberrahmen mit "eingehängten" Treibern

Trotz dieses Rahmens geht aber die Zuordnung der Treiber zu ihren jeweiligen Verwaltungsprozessen nicht verloren, außer beim Leitungstreiber, bei dem es keine Zuordnung gibt. Alle Benutzerprozesse können letzteren über die Botschaftenübermittlung bzw. Botschaftsprozeden ansprechen.

Um einen Überblick über das Verhalten des Gesamtsystems zu bekommen, ist es hilfreich, ein Schichtenmodell heranzuziehen, indem das bereits eingeführte Schalenmodell in 4 Schichten aufgebrochen wird:

1. Treiber-PU,
2. Treiberprozesse,
3. Prozeß-PU,
4. System- und Benutzerprozesse.

Die Frage, wann Elemente einer Schicht aktiv werden, wurde im einzelnen bereits beantwortet. Hier soll noch einmal eine Übersicht erfolgen:

- Der Treiber-PU wird genau dann aktiviert, wenn
 - = entweder ein Treiber auf ein Signal antworten muß;
 - = oder von einem Systemprozeß ein Auftrag für seinen Treiber ansteht;
 - = oder die Botschaftenübermittlung für den Leitungstreiber eine zu sendende Botschaft hat.
- Ein Treiber wird aktiviert, wenn er vom Treiber-PU den Prozessor erhält, um
 - = entweder einen Benutzer- (nur bei Leitungsausgabe) oder System-Auftrag abzuwickeln;
 - = oder auf ein Unterbrechungssignal zu reagieren.

- Der Prozeß-PU wird aktiviert, wenn
 - = entweder ein Prozeß sich beendet oder blockiert, weil er z.B. eine Botschaft erstellt hat, und der Prozeß damit den Prozessor abgibt (Wartestellung);
 - = oder kein Treiber mehr läuft und auf Grund dessen der Treiber-PU den Prozessor abgibt.

- Ein Benutzer- oder Systemprozeß wird nur dann aktiviert, wenn er
 - = entweder eine Botschaft erhalten hat;
 - = oder ablauffähig ist.

4. Ein zentralistisches Modell für ein Betriebssystem-Botschaften-Protokoll

In Kapitel 3 wurde nicht nur ein Betriebssystemmodell dargestellt, sondern auch ein Modell zur Kommunikation von Anwender- und Betriebssystemprozessen untereinander. Durch die Betrachtung von Realisationsgesichtspunkten, insbesondere bei den Prozeßumschaltern, konnte eine genaue Vorstellung über das gesamte Modell erzielt werden. Als Basis für dieses Modell dienen "Prozesse" und "Botschaften". Die Prozesse unterliegen in ihrer Implementation den Anforderungsspezifikationen. Die Botschaften müssen dagegen für das Gesamtsystem einheitlich realisiert werden. Diese Realisation wurde im vorangegangenen Kapitel als "Botschaftenübermittlung" oder als "BotschaftsprozEDUREN" bezeichnet. Die Übermittlung läßt sich in zwei Teile trennen: die physikalische und die logische Übermittlung. Während die physikalische Übermittlung, auf die in Kapitel 7 näher eingegangen werden soll, lediglich die "reine" Übertragung der Nachrichten (-Daten) von einem Prozeß zu einem anderen beinhaltet, befaßt sich die logische Übermittlung mit dem Protokoll zwischen den Prozessen.

Definition: Unter einem "Protokoll" versteht man ein definiertes Verfahren zwischen 2 und mehr Partnern zum Austausch von Informationen. Dieser Austausch hat den Zweck, daß nach seinem Abschluß die Partner, im Normalfall Prozesse, in der Lage sind, mit den Informationen in ihrem Programm fortzusetzen.

Wie in Kapitel 3.2. erwähnt, wurde bereits einmal am RRZE der Versuch unternommen, ein Protokollverfahren für Botschafts- und nichtdeterministische Kontrollanweisungen (GUARDED STATEMENTS) in ein bestehendes Betriebssystem zu integrieren (/KUMM83/). Bei dieser Integration sind 3 Probleme entstanden:

- Die Erweiterung eines bestehenden Betriebssystems, das nicht für verteilte Systeme ausgelegt ist, macht das gesamte System sehr unübersichtlich und schwer handhabbar.
- Die Wahl des Protokolls, nämlich des "demokratischen Verfahrens" (siehe Definition unten), erfordert einen sehr aufwendigen Algorithmus.
- Das "demokratische Verfahren" eignet sich kaum zur Integration von Sicherungsmaßnahmen, denn bereits das Setzen von Rücksetzpunkten bereitet große Schwierigkeiten.

Wie in den Arbeiten von Baacke (/BAAC87/) und Übelmesser (/UEBE87/) gezeigt wurde, ist ein "demokratisches Verfahren" viel zu schwerfällig und unübersichtlich, als daß es einem Vergleich mit dem in 4.2. vorgestellten Protokoll standhalten könnte.

Noch nachzutragen ist die Definition für das "demokratische Verfahren" (siehe auch /KUMM83/ und /UEBE87/):

Definition: Unter einem "demokratischen Verfahren" wird ein Botschaftenmechanismus auf Betriebssystemebene verstanden, in dem jeder Prozeß, der auf Anwendungsebene mit anderen Prozessen kommunizieren möchte, aktiv (Anfrage-) Botschaften an die ihm bekannten Partnerprozesse aussendet. Damit interpretiert er alle Botschaften, die wiederum von diesen Prozessen eintreffen, als direkte Antworten auf die Anfragen, egal ob es sich dabei ebenfalls um Fragebotschaften handelt. Dadurch agiert jeder Prozeß eigenständig und muß deshalb für sich selbst zu einem Ergebnis über den (nicht) erfolgreichen Abschluß der Kommunikation befinden. Da sich alle Prozesse in der gleichen Entscheidungslage mit den gleichen Rechten und der äquivalenten Schlußfolgerungslogik befinden, wird das Verfahren als "demokratisch" bezeichnet. Die Beantwortung der Frage nach der Verklemmungsfreiheit ist durch den Umstand, daß alle Prozesse nur aus ihrem Informationsstand heraus (nur Kenntnis über die direkten Partner) Entscheidungen treffen müssen, als sehr aufwendig zu betrachten.

In diesem Kapitel wird ein Protokoll vorgestellt, das zum einen den im nachfolgenden Abschnitt angeführten Anforderungen gerecht wird und, das sich zum anderen leicht um die in Kapitel 5 darzustellenden Wiederaufsetzmaßnahmen erweitern läßt.

4.1. Zu realisierendes Benutzerprogramm-Protokoll

In der Einführung zu diesem Kapitel wurde der Begriff "Protokoll" als der Informationsaustausch zwischen Prozessen definiert. Nun läßt sich aber ein Protokoll zwischen Benutzerprozessen meist nicht direkt in Form von auszutauschenden Botschaften realisieren. Der Entwickler kann beim Programmentwurf nicht sicherstellen, daß alle Prozesse, die zu einem Zeitpunkt an einer Kommunikation beteiligt sein sollen, auch wirklich zum Informationsaustausch bereit sind. Hat man, wie nachfolgend beschrieben, ein komplexes Protokoll zwischen Benutzer- und/ oder Systemprozessen durchzuführen, so ist ein "unterlagertes" Protokoll vonnöten, das das darüberliegende Protokoll "realisiert". Dies wird durch Bild 4-1 verdeutlicht. Anders ausgedrückt: Wird ein Protokoll durch ein unterlagertes (Betriebssystem-) Protokoll realisiert, so wird eine Botschaft durch eine (Folge von) Botschaft(en) des unterlagerten Protokolls realisiert. Wir wollen dafür folgende Definition verwenden.

Definition: Tauschen Benutzerprozesse Botschaften aus, so sprechen wir von "Nachrichten" eines "Benutzer- (Prozeß-) Protokolls".

Dieses Benutzerprotokoll wird durch ein "Betriebs- (System-) Protokoll", das "Botschaften" verwendet, realisiert.

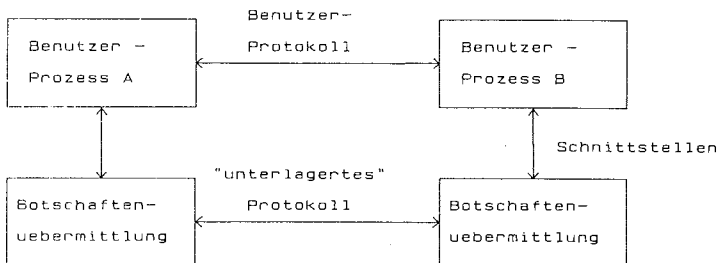


Bild 4-1: Realisierung eines Protokolls durch ein "unterlagertes" Protokoll

Der Zweck eines Benutzerprotokolls ist von Problem zu Problem verschieden und wird vom Anwendungsprogrammierer festgelegt. Das Betriebsprotokoll kann bzw. muß immer gleich sein, da eine systemweite und von den einzelnen Anwendungen unabhängige Verständigung der einzelnen Prozesse untereinander ermöglicht werden

muß.

Damit die Übermittlung von Nachrichten mittels Botschaften vonstatten gehen kann, sind noch gemeinsame "Berührungspunkte" notwendig. Diese werden als Schnittstellen bezeichnet.

Definition: Unter "Schnittstellen" ("Dienste") versteht man eine Sprachschale oder eine Menge von Prozedur- (Makro-) Aufrufen, mit deren Hilfe ein Benutzerprotokoll programmiert bzw. festgelegt werden kann.

Das Betriebsprotokoll realisiert, genau genommen, also nicht das Benutzerprotokoll, sondern die zur Verfügung gestellten Schnittstellen. Folgende Schnittstellen (Sprachschale) sollen durch das Betriebsprotokoll ermöglicht werden (siehe auch /FHKK83/):

- Senden einer Nachricht an einen anderen Prozeß
(TRANSMIT FROM HUGO TO DAGOBERT);
- Empfangen einer Nachricht von einem anderen Prozeß
(RECEIVE FROM GRETE TO DIANA);
- Alternatives Senden oder Empfangen von Nachrichten (XOR-Verknüpfung von GUARDs ("Wächter", /DIJK75/) innerhalb eines GUARDED STATEMENTS ("Bewachte Anweisung"));
- Gleichzeitiges Senden und Empfangen von Nachrichten (UND- Verknüpfung innerhalb eines GUARDs; zusätzlich zu der in /FHKK83/ vorgestellten Sprachschale sollen auch das Senden und Empfangen gleichzeitig ausführbar sein!);
- Sämtliche Anweisungen sollen durch eine Benutzerzeitüberwachung ("TIMEOUT") kontrollierbar sein;
- Dem Empfänger kann ein "Wartebereich" ("BUFFER x") vorgelagert sein, um eine asynchrone Kommunikation zu ermöglichen.

Das in diesem Kapitel erläuterte Betriebsprotokoll, das sogenannte "Baumverfahren", soll den eben genannten Anforderungen bzw. der in /FHKK83/ dargestellten Semantik der Sprachkonstrukte genügen. Im folgenden wird dieses Verfahren anhand eines Beispiels erläutert, wobei ein Regelwerk (Algorithmus) für die Ausführungsreihenfolge innerhalb des Protokolls aufgestellt wird. Das Verfahren beschränkt sich zunächst auf die rein synchrone Kommunikation ohne Zeitüberwachung. Im darauffolgenden Abschnitt wird auf die notwendigen Erweiterungen für die asynchrone Kommunikation

eingegangen. Die Zeitüberwachungsmaßnahmen werden in Kapitel 5.3. eingeführt. Der 4. Abschnitt in diesem Kapitel soll für das vorgestellte Regelwerk einen formalen Vollständigkeitsbeweis liefern, indem Schritt für Schritt auf der Basis von Mengen immer umfangreichere Operationen entsprechend der vorgestellten Regeln eingeführt werden. Unter Vollständigkeit wird dabei verstanden, daß das Verfahren sowohl deterministisch als auch endlich ist. Eine vollständige Spezifikation des Baumverfahrens findet sich im Anhang 2 (/UEBE87/).

4.2. Das Baumverfahren

Wie in Abschnitt 4.1. erläutert, soll das Betriebsprotokoll die Dienste (Sprachschale) für einen Nachrichtenaustausch realisieren. Bei Aufruf eines Dienstes durch einen Prozeß wird das Betriebsprotokoll "in Gang gesetzt" und das Verfahren versucht den Dienst zu erfüllen. Nach Abschluß des Botschaftsverfahrens und damit auch nach Beendigung des Nachrichtenaustausches wird der Prozeß wieder fortgesetzt. Mit dem Starten des Protokolls läuft für jeden Prozeß immer das gleiche Verfahren ab, für das die Grundforderungen gelten, daß es verklemmungsfrei sein und in endlicher Zeit zum Abschluß führen muß. Um diese Forderungen erfüllen zu können, wurde dem Verfahren die Idee zugrunde gelegt, daß ein zentraler Prozeß zur Verfügung stehen muß, der den Botschaftenaustausch koordiniert.

Diese zentrale Koordination hat die Vorteile, daß

- nur einer bestimmt, ob und wie es zum Nachrichtenaustausch kommt;
- sich alle anderen Prozesse passiv verhalten können, also sich selbst nicht darum bemühen müssen, ob es zum Nachrichtenaustausch kommt;
- jeder Prozeß, außer dem zentralen Prozeß, sich aus seiner Kommunikationsanweisung sofort zurückziehen kann, z.B. bei Ablauf eines Benutzer-Timeouts.

Der Nachteil liegt in der Zentralisierung, da der Nachrichten- (Botschaften-) Austausch in der Hand eines einzigen Prozesses liegt und dessen möglicher Ausfall ein Nicht-Zustandekommen der Kommunikation provoziert.

Um diesen Nachteil auszugleichen werden zwei Maßnahmen ergriffen:

- eine zusätzliche Zeitüberwachung, die in Kapitel 5.3. erläutert wird;
- eine "dynamische Bestimmung" des zentralen Prozesses.

Mit dem Begriff "dynamische Bestimmung" ist gemeint, daß bei Eintritt mehrerer Prozesse in ihre Kommunikationsanweisungen der zentrale Prozeß nicht von vornherein bestimmt ist, sondern daß unter diesen Prozessen zuerst einer "ausgewählt" wird (siehe unten).

Die gesamte Verfahrensweise wird nachfolgend an einem Beispiel dargestellt.

Beispiel: Wir gehen von 3 Prozessen (Tasks) T1, T2 und T3 aus, die außer ihrem "Restprogramm" jeweils folgende Kommunikationsanweisungen durchlaufen bzw. Dienste aufrufen:

Die Darstellung erfolgt zum einen in Form von PASS-Kommunikationsknoten (siehe Abschnitt 1.2., /FLEI84/) und zum anderen in der Notation von Verteiltem PEARL (siehe /FHKK83/).

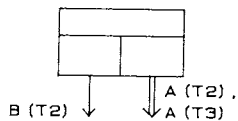
Für T1:

GUARDED REGION

GUARD RECEIVE FROM T2 TO B; REACT;

GUARD TRANSMIT FROM A TO T3; TRANSMIT FROM A TO T2; REACT;

GUARDEND;



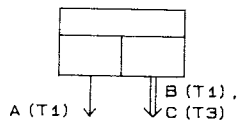
Für T2:

GUARDED REGION

GUARD RECEIVE FROM T1 TO A; REACT;

GUARD TRANSMIT FROM C TO T3; TRANSMIT FROM B TO T1; REACT;

GUARDEND;



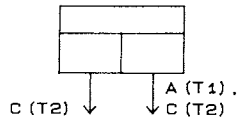
Für T3:

GUARDED REGION

GUARD RECEIVE FROM T2 TO C; REACT;

GUARD RECEIVE FROM T1 TO A; RECEIVE FROM T2 TO C; REACT;

GUARDEND;



Für die Situation, daß diese Kommunikationsanweisungen von den drei Prozessen durchlaufen werden, läßt sich eine Kommunikationsstruktur wie in Bild 4-2 darstellen.

Darin sind nochmals die Kommunikationen zwischen den Prozessen bzw. ihre Dienstaufrufe gezeigt:

- Prozeß T1 will entweder die Nachricht B von Prozeß T2 empfangen oder gleichzeitig die Nachricht A an die Prozesse T2 und T3 senden.
- Prozeß T2 will entweder die Nachricht A von Prozeß T1 empfangen oder die Nachrichten B an Prozeß T1 und C an Prozeß T3 senden.
- Prozeß T3 will entweder die Nachrichten A von Prozeß T1 und C von Prozeß T2 empfangen oder Nachricht C von Prozeß T2 empfangen (Die Empfangsanweisung für Botschaft C wird nur einmal gezeigt).

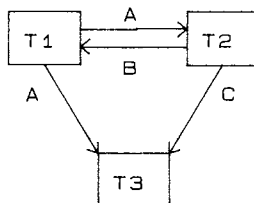


Bild 4-2: Kommunikationsstruktur für die drei Beispielprozesse

Was aus dem Bild und damit auch aus der Spezifikation nicht ersichtlich ist, ist die teilweise Inkonsistenz der Kommunikationsanweisungen: Prozeß T3 versucht, die Nachricht C von Prozeß T2 zu empfangen, die dieser nicht oder in der falschen Alternative sendet. Damit erkennt man, daß lediglich das 1. Guard für Prozeß 1, das 2. Guard für Prozeß 2 und das 1. Guard für Prozeß 3 ausführbar sind. Die alternativen Guards schlagen fehl.

Da die Prozesse ihre Aufträge unabhängig voneinander, unter Umständen auf verschiedenen Rechnern, an die Botschaftenübermittlung abgeben, ist zu Beginn der Ausführung der Kommunikationsanweisungen ein derartiger Mißstand nicht bekannt. Ein Protokoll muß also auch mit Programmierfehlern oder allgemein mit Unstimmigkeiten in den Kommunikationen fertig werden. Diese Problemstellung wird am Ende des Kapitels noch näher untersucht.

Der 1. Schritt des Verfahrens ist die Festlegung des zentralen Prozesses, im folgenden "Zentrum" genannt. Dies läßt sich einfach durch die natürliche zeitliche Reihenfolge erzielen. Da die Prozesse ihre Kommunikationsanweisungen zeitlich unabhängig voneinander betreten, bleiben zunächst die ersten Prozesse blockiert. Erst durch den letzten Prozeß ist eine Ausführung der Kommunikation möglich. Tritt dieser Prozeß in seine Anweisung ein, so wird er, da er der letzte ist, zum Zentrum dieser Kommunikation. Schwierigkeiten können dann auftreten, wenn mehrere Prozesse nahezu gleichzeitig ihre Kontrollanweisungen betreten. Dadurch entsteht das Problem, daß es mehrere Zentren geben kann. Um diesen Konflikt zu lösen, ist es naheliegend, für die Prozesse Prioritäten zu vergeben. Dies kann dynamisch oder statisch geschehen. Beim dynamischen Verfahren legt ein Zentrum z.B. durch einen Zufallszahlengenerator für sich selbst eine Priorität fest und teilt diese über eine Botschaft dem oder den anderen Zentren mit. Da dabei gleiche Werte auftreten können, muß der Algorithmus solange wiederholt werden, bis nur noch ein Zentrum übrigbleibt.

Eine schnellere Vorgehensweise verspricht die Festlegung statischer Prioritäten für alle Prozesse im gesamten System in Form von einer Tabelle. Diese liegt jedem Prozeß vor, so daß er im Falle einer Kollision mit einem anderen Zentrum sofort feststellen kann, wer im Baumverfahren fortfahren darf. Die Prioritäten müssen wie in /HOFM84/ und /UEBE87/ gezeigt, durch eine lineare Anordnung repräsentiert werden. In /UEBE87/ wird auch erläutert, wie diese Anordnung im Baumverfahren ausgenutzt wird und die Spezifikation in Anhang 2 nimmt davon Gebrauch.

Aus dem eben Erläuterten entsteht noch die Frage: Wie kann ein Zentrum feststellen, daß noch weitere Zentren existieren? Prinzipiell ist es natürlich möglich, daß ein

Zentrum bereits zu Beginn (siehe Regel 1) versucht zu ermitteln, ob es noch weitere Zentren gibt. Da es zu diesem Zeitpunkt aber sehr aufwendig ist diese zu ermitteln und da auch noch nicht sicher ist, daß es überhaupt zu einem vollständigen Baumaufbau kommt, erscheint es zweckmäßiger, die Feststellung auf die Reservierungsphase (siehe Regel 8) zu verschieben. Dort sind alle an der Kommunikation beteiligten Prozesse bekannt und die Reservierung dient auch dazu die Prozesse für die Kommunikation festzulegen. Dabei darf nur dasjenige Zentrum reservieren, das die höchste Priorität hat.

Aus dem eben Gesagten läßt sich die erste von mehreren Regeln ableiten:

Regel 1: Tritt ein Prozeß in eine Kommunikationsanweisung ein, so versucht er, als zentraler Prozeß (Zentrum) mittels Botschaften eine Kommunikation mit anderen Prozessen aufzubauen. Gelingt dies nicht, wird er blockiert und damit in einen Wartezustand versetzt. Ist er in der Lage, mit den anderen Prozessen ("Partnerprozessen") Verbindung aufzunehmen, muß er durch einen geeigneten Algorithmus sicherstellen, daß kein anderer Prozeß für die gleichen Partnerprozesse zum gleichen Zeitpunkt versucht, Zentrum zu sein.

Aufgabe des Zentrums ist es, sich einen "Überblick" über die Kommunikationsausdrücke seiner Partnerprozesse in Zusammenhang mit seinen eigenen Kommunikationsanweisungen zu verschaffen. Mit Hilfe dieses Überblicks kann er dann feststellen, wer mit wem und wie kommuniziert. Der Überblick soll eine bestimmte Form haben, nämlich die eines Baumes. Es wird also ein Baum erstellt, der in übersichtlicher Weise darstellen soll, wie die Kommunikationsverhältnisse sind. Um diesen Überblick zu erlangen, muß das Zentrum Botschaften aussenden, für die es dann (Antwort-) Botschaften erhält. Dabei besteht zwischen dem Zentrum und seinen Partnerprozessen ein genau geregeltes Verhältnis: das Zentrum ist aktiv, d.h. es sendet Fragen aus, und die Partnerprozesse sind passiv, sie geben nur Antworten. Man könnte in diesem Zusammenhang von einem Art "Remote Procedure Call"-Mechanismus sprechen: Das Zentrum führt abgesetzte Prozeduraufrufe (= Fragebotschaften) aus, die die Partnerprozesse ausführen und jeweils mit einem Ergebnis (= Antwortbotschaft) abschließen. Dabei kann das Zentrum mehrere Aufrufe absetzen und dann auf die Ergebnisse warten.

Ein weiterer wichtiger Punkt ist das Verhältnis zwischen den Bäumen und Kommunikationsanweisungen des Zentrums, das die Bäume aufbaut. Da die einzelnen Wächter bzw. die einzelnen Kanten der Kommunikationsknoten logisch mit XOR (exklusives Oder) verknüpft sind, dürfen sie nur alternativ ausgeführt werden. Es darf also insgesamt nur genau ein Wächter oder keiner ausgeführt werden. Letzterer Fall tritt ein, wenn in jedem Guard mindestens eine Kommunikation nicht ausführbar ist.

Da die Guards unabhängig voneinander sind, bedeutet dies, daß für jedes ein eigener Baum aufgebaut wird. Die Unabhängigkeit der Bäume und deren dynamischer Aufbau geht sogar soweit, daß sich für jeden Baum ein eigener Prozeß, ein sogenannter "Unterprozeß" einführen läßt. Diese Unterprozesse können unabhängig voneinander und parallel ihren Baumaufbau betreiben. Allerdings müssen in einer Implementierung (siehe /UEBE87/) die Unterprozesse nicht notwendigerweise realisiert werden. Das eben Erläuterte läßt sich in folgender Regel zusammenfassen:

Regel 2: Der zentrale Prozeß kreiert für jeden seiner alternativen Wächter (Kommunikationskanten) einen zentralen Unterprozeß. Jeder dieser Unterprozesse, die mit den natürlichen Zahlen ohne die Null durchnummeriert sind (die Anzahl der Wächter ist die größte vorkommende Nummer), versucht mit Hilfe von Frage-Botschaften aktiv einen Baum aufzubauen. Die Partnerprozesse (Partnerunterprozesse) verhalten sich passiv und geben nur Antwort-Botschaften.

Fortsetzung des Beispiels:

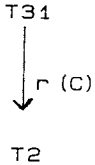
In unserem Beispiel gibt es drei Bewachte Anweisungen der drei Prozesse T1, T2 und T3. Daraus folgt, daß die Unterprozesse T11, T12, T21, T22, T31 und T32 existieren, wobei T31 und T32 aktive Zentren für den Baumaufbau sind. Letztere müssen zunächst bei den Partnerprozessen nachfragen, ob entsprechende Kommunikationsanweisungen vorliegen. Bei den etwaigen Unterprozessen können sie noch nicht fragen, da sie über deren Existenz noch nicht Bescheid wissen. Unterprozeß T31 stellt also an Prozeß T2 die Frage: "Existiert eine Kommunikationsanweisung, in der die Botschaft C an Prozeß T3 gesendet wird". "Gesendet" heißt es aus der Sicht von T2. T32 stellt an die Prozesse T1 und T2 die Fragen: "Existiert eine Anweisung, in der A an T3 gesendet wird und existiert eine Anweisung, in der C an T3 gesendet wird".

Bemerkung: Sowohl für das Senden als auch für das Empfangen sei zunächst vorausgesetzt, daß die Botschaften unverfälscht und ohne Verzögerung übermittelt werden.

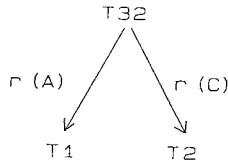
Als Antwort von T2 erhält der Unterprozeß T31 die Auskunft, daß Unterprozeß T22 die Botschaft C an T3 sendet. Unterprozeß T32 erhält Antworten von T1 und T2: "T12 sendet Botschaft A an T3 und T22 sendet Botschaft C an T3".

Die Ergebnisse des Frage-Antwort-Spiels spiegeln sich natürlich auch in den Bäumen wider. Bevor die Fragen gestellt werden, kann man von einem sog. "Anfangsbaum" ausgehen, in dem alle zu Anfang bekannten Kommunikationen dargestellt sind:

Baum für T31:

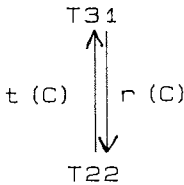


Baum für T32:

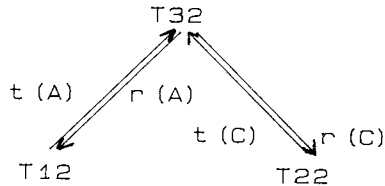


"r(A)" bedeutet, daß eine Botschaft A empfangen (receive) wird. Die Pfeilrichtung gibt dabei an, von welchem Prozeß aus die Anweisung zu lesen ist. Nach Erhalt der Antworten können die Bäume folgendermaßen ergänzt werden:

Baum für T31:

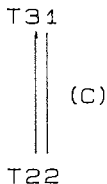


Baum für T32:

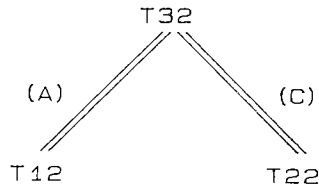


"t(A)" steht für "Senden der Botschaft A" (transmit). Da sich in allen Kanten der Bäume jeweils die Sende- und Empfangsanweisungen entsprechen, lassen sich folgende Kurzformen angeben, wobei die Transferrichtung keine Rolle mehr spielt, da es nur auf das Feststellen der Kommunikationspartner ankommt:

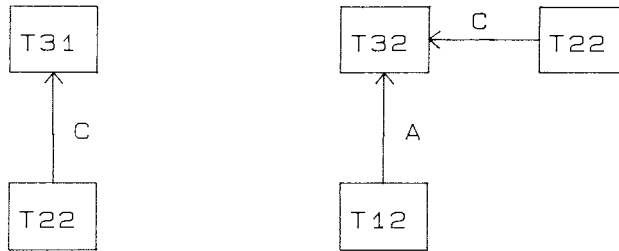
Baum für T31:



Baum für T32:

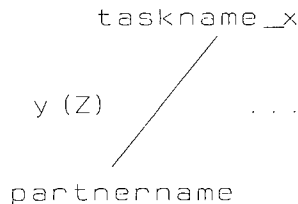


Synonym dafür kann man auch Kommunikationsstrukturen angeben:



Die am Beispiel gezeigte Vorgehensweise schlägt sich in folgenden Regeln nieder:

Regel 3: (Initialregel) Ein Unterprozeß kommuniziert mit einem oder mehreren Partnerprozessen (logische UND-Verknüpfung). Dies läßt sich in einem "Anfangsbaum" darstellen:



Es muß mindestens eine Kante existieren. "x" steht für die Unterprozeßnummer, "y" für die Sende-/Empfangsrichtung (transmit, Senden oder receive, Empfangen) und "Z" steht für den (vom Benutzer gewählten Nachrichten-) Botschaftsnamen. "taskname_x" ist im Baum der Wurzelknoten, und die Blätter werden mit den Namen der Partnerprozesse ("partnername") beschriftet.

Regel 4: Die zentralen Unterprozesse erstellen Botschaften, mit denen sie bei den in den Blättern angegebenen Prozessen das Vorhandensein entsprechender Kommunikationsanweisungen (Beschriftungen der Kanten) erfragen. Die Partnerprozesse können die Frage auf zwei verschiedene Arten beantworten:

- 1) Die gefragte Anweisung ist nicht vorhanden, damit scheidet der zentrale Unterprozeß aus dem Verfahren aus.

- 2) Die gefragte Anweisung kommt mindestens einmal vor, damit kann die entsprechende Kante vervollständigt werden. Mit der Antwort werden alle weiteren Kommunikationspartner des befragten Partnerprozesses mitgeliefert, sodaß der (Initial-) Baum entsprechend ergänzt werden kann.

Bemerkung zu 2): In /UEBE87/ wird die Vervollständigung einer Kante auch als "Sättigung" oder "Verholzen" der Kante bezeichnet.

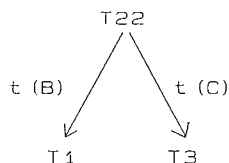
Vor allem letzterer Begriff macht recht plastisch die Vorgehensweise deutlich. Vor dem Nachfragen ist noch nicht klar, ob der Ast Teil des Baumes ist. Mit der erhaltenen Antwort wird entweder festgestellt, daß der Ast (die Kommunikationsanweisung) nicht gesättigt werden kann und deshalb der Baum aus dem Verfahren ausscheidet oder, daß der Ast fest zum Baum gehört und nicht mehr entfernbar (verholzt) ist.

In einem nächsten Schritt oder bereits mit Hilfe der Regel 4 muß sich nun der zentrale Unterprozeß über die möglichen Kommunikationspartner seiner bisherigen Blätter, der unmittelbaren Partnerprozesse, informieren. Genau das ist die Eigenschaft des zentralen Verfahrens, daß er über alle Kommunikationen Bescheid wissen muß.

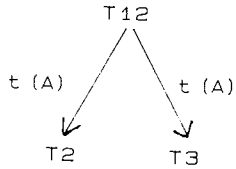
Fortsetzung des Beispiels:

Die Unterprozesse T31 und T32 kennen also ihre unmittelbaren Partner und können jetzt von diesen erfahren, wer wiederum deren Partner sind. Die Unterprozesse T22 und T12 werden aufgefordert, ihre eigenen (Teil-) Bäume zu senden, soweit bereits vorhanden. Als Antworten erhält man Bäume mit dem Namen des nachgefragten Unterprozesses als Wurzelbezeichnung und als Blattbezeichnungen deren Kommunikationspartner.

Der Unterprozeß T22 liefert den Antwortbaum:

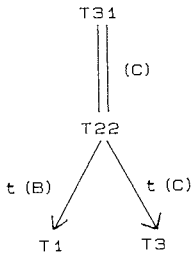


Der Unterprozeß T12 liefert den Antwortbaum:

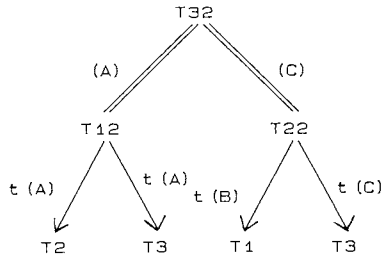


Damit lassen sich die Bäume von T31 und T32 folgendermaßen ergänzen:

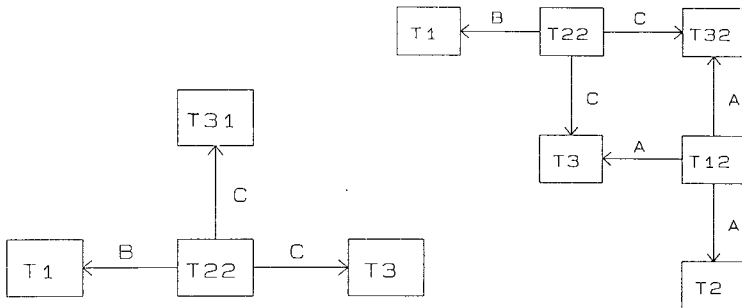
Baum für T31:



Baum für T32:



Die entsprechenden Kommunikationsstrukturen sehen so aus:



Betrachten wir zunächst den Baum bzw. die Kommunikationsstruktur für Prozeß T32. Es fällt auf, daß der antwortende Prozeß T12 mit den Prozessen T2 und T3 jeweils über die Botschaft A kommuniziert (linker Teilbaum) und, daß Unterprozeß T22 mit den Prozessen T3 und insbesondere T1 über die Botschaften C und B kommuniziert (rechter Teilbaum). Informell heißt das, daß T22 "behauptet", mit T1 über B zu kommunizieren, T12

aber nichts davon "weiß". Da wiederum T32 unbedingt mit T12 in Verbindung steht und aus jedem Prozeß wegen der XOR-Verknüpfung jeweils nur genau ein Unterprozeß (Wächter) ausgewählt werden darf, kann T22 an T1 keine Botschaft senden!

Das Verfahren ist also in der Lage, die bereits bei der Vorstellung des Beispiels geschilderte Inkonsistenz zu entdecken. Aufgrund dieser Tatsache muß der zentrale Unterprozeß T32 an den zentralen Prozeß T3 melden, daß für ihn keine Kommunikationen ausführbar sind.

Betrachtet man die Situation bei Unterprozeß T31, so sind als neue Kommunikationspartner die Prozesse T1 und T3 hinzugekommen, zu denen von T22 aus die Botschaften B und C gesendet werden. Dabei besteht zwischen den Ästen T31 \leftarrow T22 und T22 \leftarrow T3 offensichtlich kein Widerspruch, so daß man annehmen kann, daß es sich um die gleichen Kommunikationsanweisungen handelt.

Die im Beispiel dargestellten Schritte lassen sich wie folgt zusammenfassen:

Regel 5: (Iterationsregel, Verallgemeinerung der Regel 3) Ein Baum besteht aus einer Wurzel und einer beliebigen Anzahl von Knoten und Blättern, die miteinander über Äste verbunden sind. Die Äste repräsentieren die Kommunikationsanweisungen, bestehend aus Botschaftsnamen und Sende-/Empfangsrichtung (z.B. $y(Z)$) oder nur aus Botschaftsnamen bei gesättigten Ästen. Die Wurzel ist eine zentrale Unterprozeßnummer. Bei den Knoten handelt es sich um Unterprozeßnummern, die Blätter oder Knoten als Nachfolger haben und deren Vorgänger Knoten oder die Wurzel sind. Blätter haben Knoten als Vorgänger und keine Nachfolger. Ein Blatt ist entweder Prozeß- oder Unterprozeßnummer.

Kommt ein Blatt im ganzen Baum noch nicht als Knoten (Unterprozeßnummer) oder als Prozeßnummer, als Teil einer Unterprozeßnummer eines Knotens, vor, so muß Regel 6 angewendet werden.

Regel 6: (Inkonsistenzregel, Verallgemeinerung der Regel 4) Für jedes durch Regel 5 gefundene Blatt wird beim betreffenden (Unter-) Prozeß nachgefragt, ob die entsprechende Kommunikationsanweisung vorhanden ist, und wenn ja, welche weiteren Kommunikationsanweisungen sich im Wächter befinden (Senden des Teilbaums). Der neue Teilbaum wird an den bestehenden Baum angefügt und die darin enthaltenen Prozeßnummern mit den (Unter-) Prozeßnummern des bisherigen Baums verglichen (Regel 5 und Regel 7).

Dabei muß immer folgender Satz gelten: Ein durch den Teilbaum neu entstandenes Blatt mit ungesättigter Kante darf im übrigen Baum nicht vorkommen!

Letztere Aussage kann so erläutert werden: Ein Knoten verfügt nur über gesättigte Kanten. Taucht dieser Unterprozeß oder der entsprechende Prozeß wieder als Blatt, d.h. als Nachfolger eines Knotens, mit ungesättigter Kante auf, so steht dies im Widerspruch zu der Tatsache, daß er bereits Knoten ist.

Fortsetzung des Beispiels:

Im Baum für T32 war T12 bereits Knoten, als erneut die ungesättigte Kante zwischen T22 und T1 auftauchte. Durch diese Inkonsistenz schied der Unterprozeß T32 aus.

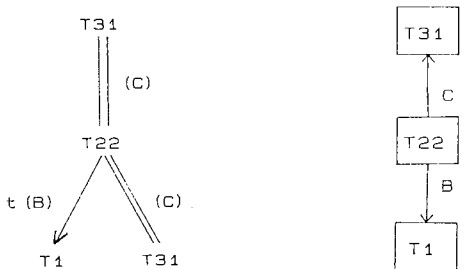
Die Regeln 5 und 6 bewirken, daß ein Baum für einen Unterprozeß aufgebaut wird. Regel 7 stellt abschließend den vollständigen Baum fest:

Regel 7: (Vollständigkeitsregel) Wurde nach Regel 6 kein Widerspruch gefunden und sind alle Blätter des neuen Teilbaums bereits als Knoten im gesamten Baum vorhanden, so ist der Baum vollständig aufgebaut. Im Widerspruchsfall scheidet der Baum aus. Gibt es noch ungesättigte Kanten, werden wieder die Regeln 5 und 6 angewandt. Ein vollständiger Baum zeichnet sich dadurch aus, daß er nur gesättigte Kanten hat. Dies bedeutet, daß alle an der Kommunikation beteiligten Prozesse sich im Baum befinden.

Regel 7 besagt, daß der zentrale Unterprozeß jetzt alle seine Partner "kennt".

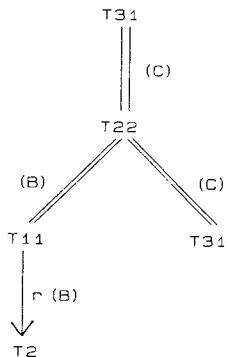
Fortsetzung des Beispiels:

Es wurde als letztes festgestellt, daß es im Baum für Unterprozeß T31 eine gesättigte Kante $T31 \xrightarrow{C} T22$ und eine ungesättigte Kante $T22 \xrightarrow{B} T1$ gibt. Wir haben deshalb folgenden Baum und folgende Kommunikationsstruktur:

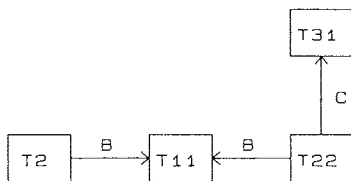


In Anwendung der Regeln 5 und 6 muß beim Prozeß T1 nachgefragt werden, um welchen Unterprozeß es sich handelt und wie seine Partnerprozesse lauten. Als erstes Ergebnis erhalten wir Unterprozeß T11 und als weitere Antwort bekommen wir als Partner den Prozeß T2.

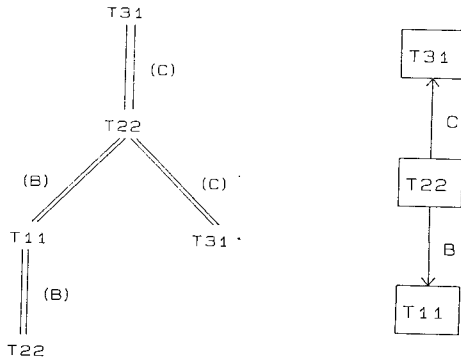
Damit ergibt sich dieser Baum:



Die entsprechende Kommunikationsstruktur sieht so aus:



Bei der Überprüfung des so entstandenen Baumes erkennt man, daß man die ungesättigte Kante $T11 \xrightarrow{(B)} T2$ mit der gesättigten Kante $T22 \xrightarrow{(B)} T11$ abgleichen kann. Wir können damit von folgenden Bildern ausgehen:



Dieser Baum ist nach Regel 7 vollständig, und es sind keine weiteren Frage-Botschaften notwendig. Das Kommunikationsverhalten wird durch die soeben festgestellte Kommunikationsstruktur deutlich, aus der wir erkennen, daß Unterprozeß T22 in einer UND-Verknüpfung die Nachrichten B und C jeweils an die Unterprozesse T11 und T31 sendet.

Unterprozeß T31 kann melden, daß er einen vollständigen Baum hat. Das Zentrum T3 stellt fest, daß keine andere Alternative existiert, und informiert deshalb per Botschaft seine Partnerprozesse T1 und T2 über den möglichen Nachrichtenaustausch. In unserem einfachen Beispiel kann somit folgender Ablauf von Botschaften erzeugt werden:

- T1 und T2 werden über den bevorstehenden Nachrichtenaustausch über ihr 1. bzw. 2. Guard informiert. Sie sperren deshalb ihre anderen Alternativen für einen etwaigen Nachrichtenaustausch.
- T1 und T2 melden an T3, daß sie bereit sind.
- T3 fordert T1 und T2 endgültig zum Nachrichtenaustausch auf.
- Da lediglich Prozeß T2 Sendeanweisungen durchläuft, sendet er die Nachrichten an die Prozesse T1 und T3 und kann anschließend fortsetzen.
- Nach Erhalt der Nachrichten setzen T1 und T3 in ihrem Programm fort.

Diese am Beispiel dargestellte Folge von Botschaften läßt sich folgendermaßen zusammenfassen:

Regel 8: Hat ein zentraler Unterprozeß einen vollständigen Baum erzielt, so übergibt er diesen an seinen zentralen Prozeß. Hat dieser mehrere Unterprozesse zur Auswahl, so muß er einen auswählen und die darin enthaltenen (Unter-) Prozesse vom möglichen Nachrichtenaustausch informieren. Ist einer der angesprochenen Prozesse nicht mehr dazu bereit, scheidet der Baum aus. Sind alle Prozesse mit der "Reservierung" einverstanden, so können alle beteiligten Prozesse zum endgültigen Nachrichtenaustausch aufgefordert werden, der darauffolgend stattfindet.

Zusammenfassend kann man feststellen, daß das Baumverfahren ein **endliches und deterministisches** Verfahren ist. Endlich deshalb, weil es nur endlich viele Prozesse in einem Prozeßsystem gibt und damit auch nur endlich viele Kommunikationen zu einem Zeitpunkt stattfinden können. Deterministisch, da die Prozesse zu einem Zeitpunkt nur in einmaliger Kommunikation zueinander treten können, die in einem Baum fixiert wird. Diese Vollständigkeit (endlich und deterministisch) soll in Abschnitt 4.4. noch an Hand einer Beweisskizze gezeigt werden.

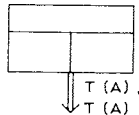
Es existiert lediglich **eine einzige** (sinnvolle) Implementationseinschränkung:

Einschränkung: Zwei Prozesse dürfen nicht über die gleiche Nachricht mit gleicher Kommunikationsrichtung (TRANSMIT / RECEIVE) in jeweils dem gleichen Guard mehr als einmal miteinander kommunizieren. Es ist also folgendes verboten:

GUARDED REGION

```
:  
    GUARD TRANSMIT FROM A TO T; TRANSMIT FROM A TO T;  
    REACT;  
:  
GUARDEND;
```

oder

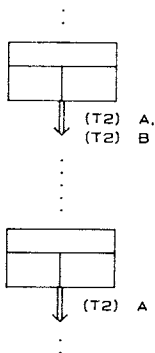


Abschließend soll noch einmal die im Beispiel festgestellte Inkonsistenz in den Kommunikationsanweisungen der drei Prozesse diskutiert werden. Wie erläutert, kann möglicherweise ein Programmierfehler vorliegen. Andererseits kann es aber auch an der Ausführungsreihenfolge der Kommunikationsanweisungen gelegen haben. Das würde bedeuten, daß zwar die einzelnen Anweisungen der verschiedenen Prozesse alle zueinander passen, aber Prozesse in bestimmten Situationen nicht zueinander passende Anweisungen durchlaufen, z.B. weil ein Prozeß statt eines "Then-Zweiges" einen "Else-Zweig" ausführt. Zu Verklemmungen, wie am obigen Beispiel gezeigt, muß das noch lange nicht führen.

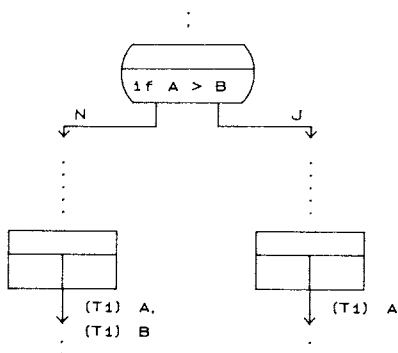
Zu dieser Problematik sei noch ein weiteres Beispiel gegeben.

Beispiel:

Prozess T1 :



Prozess T2 :



Werden die Kommunikationsanweisungen in der richtigen Reihenfolge durchlaufen, so ergeben sich offensichtlich keine Probleme. Wird aber einer der beiden Prozesse, z.B. Prozeß T2, veranlaßt, aufgrund irgendeiner Bedingung, z.B. wegen der Abfrage "IF A > B", die Kommunikationsanweisungen in einer anderen Reihenfolge zu durchlaufen, so kann es zu einem Stillstand von T1 und T2 kommen. Es läßt sich auch hier wieder argumentieren, daß es sich um einen Programmierfehler handelt, denn schließlich sollten beim Programmentwurf alle Möglichkeiten eingeplant werden. Haben allerdings die eben gezeigten Kommunikationsanweisungen zusätzliche TIMEOUT-Ausgänge, so hat der Entwickler Vorsorge getroffen, es liegt kein Fehler vor.

Zusammenfassend heißt das: Wenn ein Unterprozeß seinem Hauptprozeß meldet, daß es aufgrund von Widersprüchen in seinem Baum zu keiner Kommunikation kommen kann, so muß noch kein Programmierfehler vorliegen. Liegen alternative und ausführbare Wächter (vollständige Bäume) oder TIMEOUT-Anweisungen vor, dann kann diese Situation beabsichtigt sein. Entscheiden kann dies nur der Hauptprozeß selbst, allerdings auch wieder mit der Einschränkung, daß seine Informationen sich lediglich auf seine Unterprozesse (Kommunikationsanweisungen) beschränken. Es ist möglich, daß einer seiner Partnerprozesse den Stillstand aufzulösen vermag. Nur ein Überblick über das gesamte System könnte hier Abhilfe schaffen. Dafür müßte aber ein großer Aufwand betrieben werden, der das System in nicht mehr vertretbaren Ausmaßen belasten würde.

4.3. Erweiterungen für die asynchrone Kommunikation

Das bisherige Regelwerk (Abschnitt 4.2.) war nur auf die rein synchrone Kommunikation abgestimmt. In diesem Kapitel werden die Vorschriften dahingehend erweitert, daß auch Prozesse, die mit einem "Wartebereich" (BUFFER) versehen sind, an der Kommunikation teilnehmen können. Zum besseren Verständnis müssen noch einige semantische Eigenschaften des in /FHKK83/ und /FLEI84/ vorgestellten Wartebereich-Konzeptes dargelegt werden.

Für jeden Prozeß kann ein Wartebereich mit der Anweisung "BUFFER X" deklariert werden, wobei X eine ganze positive Zahl größer der Null ist. Damit ergeben sich folgende Eigenschaften bzw. Folgerungen daraus:

- Da für den Wartebereich keine Typgebundenheit herrscht, kann jeder Senderprozeß dort seine Nachrichten in beliebiger Anzahl, maximal bis X, hinterlegen. Kein Prozeß kann daran gehindert werden, den Wartebereich zu füllen und anderen Sendern dadurch die Möglichkeit zu nehmen, ebenfalls Nachrichten einzutragen.
- Der Empfänger darf nur asynchron empfangen, d.h. er darf nur aus seinem Wartebereich lesen. Sind, wie eben geschildert, nur Nachrichten eines bestimmten Prozesses im Wartebereich und wollen ein anderer Sender und der besagte Empfänger eine Nachricht austauschen, so sind beide bei vollem Wartebereich blockiert.

Diese beiden Eigenschaften zeigen, daß die hier vorgestellte asynchrone Kommunikation keine echte Erweiterung der synchronen ist. Diese wäre erst durch einen typgebundenen Wartebereich möglich. Ein einfaches, in /UEBE87/ vorgestelltes Beispiel zeigt eine weitere Problematik:

Beispiel:

TASK1:BUFFER 1

```
      :  
      GUARDED REGION  
      GUARD TRANSMIT FROM A TO TASK2;  
      RECEIVE FROM TASK2 TO B; REACT;  
      GUARDEND;
```

TASK2:BUFFER 1

```
      :  
      GUARDED REGION  
      GUARD TRANSMIT FROM B TO TASK1;  
      RECEIVE FROM TASK1 TO A; REACT;  
      GUARDEND;
```

Wenn wir annehmen, daß die Wartebereiche leer sind, kommt keine Kommunikation zustande. TASK1 versucht erst dann in Verbindung zu einem anderen Prozeß zu treten, wenn die gewünschte Nachricht im Wartebereich vorzufinden ist, genauso wie TASK2.

Das Beispiel zeigt, daß hier nicht einmal der typegebundene Wartebereich weiterhilft. Der Grund dafür liegt im Vermischen von TRANSMIT und RECEIVE und in der Behandlung der asynchronen Kommunikation. Genau genommen handelt es sich lediglich um ein asynchrones Empfangen.

Das Senden dagegen ist immer synchron: hat der Empfänger die Anweisung

- NOBUFFER, so muß der Empfänger auf der entsprechenden Empfangsanweisung stehen,
- BUFFER X, so muß Platz im Wartebereich sein,

ansonsten wird der Sender blockiert.

Für obiges Beispiel und für alle Situationen, in denen über Wartebereich bei gleichzeitigem Senden asynchron empfangen werden soll, gibt es 2 Fälle zu unterscheiden:

- alle die zu empfangenden Nachrichten sind bereits im Wartebereich, eine Kommunikation kann stattfinden,
- eine der zu empfangenden Nachrichten fehlt im Wartebereich, eine Kommunikation kann nicht stattfinden.

Fazit: Es wird bei Vorhandensein eines Wartebereichs nicht darauf Rücksicht genommen, ob bei gleichzeitigem (UND-Verknüpfung) Senden und Empfangen die entsprechenden Partnerprozesse die für das Empfangen nötigen Nachrichten bereithalten oder nicht, sondern es wird lediglich überprüft, ob die Nachricht bereits im Puffer eingetragen ist!

Dieses Fazit wird für die folgenden Überlegungen zugrunde gelegt. Das bedeutet, daß beim Senden nicht der Partnerprozeß, sondern dessen Wartebereich als Kommunikationspartner auftritt, da dort die Nachrichten hinterlegt werden müssen. Um das Baumverfahren möglichst konsistent zu erweitern, werden zunächst zwei Regelungen getroffen:

Regel 9: Hat ein Prozeß einen Wartebereich (WB), so hat er auch einen sogenannten Meldebereich (MB). In ihm werden alle Prozesse "notiert", die einmal versucht haben, im Wartebereich eine Nachricht zu hinterlegen, aber scheiterten, da dieser bereits voll war. Wird wieder Platz im WB frei, werden alle im MB befindlichen Prozesse davon in Kenntnis gesetzt, damit sie sich um diesen Platz wieder bewerben können.

Der Sinn der Regel ist es, zu verhindern, daß der "Partner- (Prozeß-) Wartebereich" zu einem Zentrum werden kann. Die Eigenschaft, Zentrum zu sein, bleibt also nach wie vor den aktiven Teilen eines Prozeßsystems, nämlich den Prozessen überlassen, während Datenstrukturen wie der WB passiv bleiben. Das Löschen des MB und das in Kenntnis Setzen der Senderprozesse übernimmt der Prozeß, zu dem der MB und der WB gehören. Er ist sowieso aktiv, da er gerade in diesem Moment ein Element aus dem WB entnommen hat.

Während Regel 9 sicherstellt, daß ein Wartebereich innerhalb des Baumverfahrens passiv bleibt, also lediglich Antwortgeber ist, definiert Regel 10, welches Verhalten der WB aus der Sicht des Zentrums an den Tag legt, da kein Unterprozeß vorhanden ist.

Regel 10: Ist innerhalb des Baumverfahrens ein Wartebereich beteiligt, zu dem gesendet bzw. in den eine Nachricht hinterlegt werden soll, so trägt er die Bezeichnung "partnername_0" im Baum des Zentrums, wenn der Prozeß, dem er zugeordnet ist, den Namen "partnername" trägt. Damit besitzt der Wartebereich eine Unterprozeßnummer, ohne daß der betreffende Prozeß selbst eine Kommunikationsanweisung zur Ausführung anstehen haben muß. Das Gleiche gilt für Prozeßgruppen (/FLEI84/), wobei "partnername" dem Namen der Gruppe entspricht.

Durch diese Regel ist es möglich, den WB aus der Sicht des Zentrums wie einen Unterprozeß zu behandeln, mit dem einzigen Unterschied, daß bei diesem nicht nach weiteren Kommunikationspartnern nachgefragt werden muß.

Regel 11: Erhält das Zentrum auf die Frage nach einer Empfangsanweisung (aus der Sicht des Partnerprozesses) eine Unterprozeßnummer 0 als Antwort, so ist die entsprechende Kante gesättigt, und es braucht nicht mehr weiter nachgefragt zu werden.

Schließlich muß unter Berücksichtigung des oben erzielten Fazits noch eine Regel aufgestellt werden, wie ein Wartebereich zu behandeln ist, aus dem gelesen wird.

Regel 12: Ist ein Prozeß mit Wartebereich am Baumverfahren beteiligt und soll er in diesem Zusammenhang aus dem WB mittels einer Empfangsanweisung (RECEIVE) lesen, so hat er nur nachzusehen, ob die gewünschte Nachricht vorhanden ist oder nicht. Ist sie nicht vorhanden, provoziert er einen Abbruch des Verfahrens.

Ist der Baum vollständig aufgebaut, muß für die "Reservierungsphase" noch folgendes gelten.

Regel 13: Erhält ein Prozeß mit Wartebereich eine Benachrichtigung über einen bevorstehenden Nachrichtenaustausch (Reservierung), so hat er die mitgelieferte Anzahl an Pufferplätzen für zu empfangende Nachrichten zu belegen. Führt er seinerseits Empfangsanweisungen aus, so muß er sich (nochmals) vergewissern, ob die gewünschten Nachrichten (noch) vorhanden sind. Ist eines von beiden nicht möglich, muß er eine negative Rückmeldung erzeugen.

Die Regeln sollen zusammenfassend an einem Beispiel verdeutlicht werden:

Beispiel: Gegeben seien zwei Prozesse:

TASK1:NOBUFFER

:

GUARDED REGION

GUARD TRANSMIT FROM A TO TASK2;

RECEIVE FROM TASK2 TO B; REACT;

GUARDEND;

TASK2:BUFFER 2

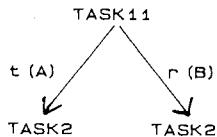
```

:
GUARDED RREGION
GUARD RECEIVE FROM TASK1 TO C;
    TRANSMIT FROM B TO TASK1; REACT;
GUARDEND;
```

Inhalt vom Wartebereich der TASK2:

1. Element: FROM TASK1 TO C,
2. Element: leer.

TASK2 sei blockiert, da TASK1 nicht bereit war. TASK1 betritt seine Kommunikationsanweisung und erhält folgenden Anfangsbaum für den zentralen Unterprozeß TASK11:



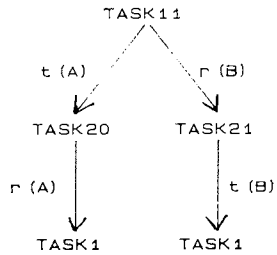
Unterprozeß TASK11 fragt bei TASK2 nach den entsprechenden Kommunikationsanweisungen. TASK2 stellt fest, daß

- TASK11 $\tau(A)$ >TASK2 ausführbar ist, da noch ein Platz im WB vorhanden ist,
- TASK11 $\tau(B)$ >TASK2 ausführbar ist, da ein TASK21 $\tau(B)$ >TASK1 existiert,
- TASK2 $\tau(C)$ >TASK1 ausführbar ist, da der entsprechende Eintrag im WB vorhanden ist.

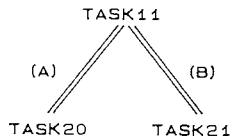
Die zweite Task kann also folgende Antworten zurücksenden:

TASK20 $\tau(A)$ >TASK1 und TASK21 $\tau(B)$ >TASK1.

Der neue Baum sieht so aus:



und zusammengefaßt:



Der Baum ist bereits vollständig. Die Reservierung und schließlich der Nachrichtenaustausch können eingeleitet werden.

4.4. Formale Beweisskizze für die Vollständigkeit des Regelwerks

Insgesamt wurden 13 Regeln aufgestellt, nach denen sowohl eine synchrone als auch eine asynchrone Kommunikation zwischen Prozessen, wie in Abschnitt 4.1. gefordert, möglich ist. Dadurch können Prozesse auf dem gleichen und auf entfernten Rechnern Nachrichten blockierungsfrei miteinander austauschen. Zunächst wurde das Baumverfahren nur für die synchrone Kommunikation erläutert und dabei wurde deutlich, daß es auf der Benutzer- bzw. Programmierenebene durchaus zum Blockieren von Prozessen kommen kann, ohne auf der Betriebssystemebene Verklemmungen hervorzurufen. Dabei ist es egal, ob tatsächlich Programmierfehler oder lediglich momentane Fehlläufe der Prozesse vorliegen.

Die notwendigen Ergänzungen für die asynchrone Kommunikation haben sich nahtlos in das Regelwerk eingefügt, so daß zur Laufzeit effektiv kein Mehraufwand beim Baumaufbau betrieben werden muß. Aufgrund der Regeln (Algorithmus) kann für die an einer Kommunikation beteiligten Prozesse ein Übergangsautomat angegeben werden. Dieser wird allerdings erst in Abschnitt 5.3. dargestellt, um die dort eingeführten Zeitüberwachungsmaßnahmen zu berücksichtigen.

Was den Aufwand für das Baumverfahren insgesamt betrifft, so erscheint er vor allem bei recht einfachem Nachrichtenaustausch (TRANSMIT / RECEIVE) als relativ hoch. Der Nutzen dieser Vorgehensweise wird allerdings erst in Kapitel 5 deutlich, im Zusammenhang mit den Sicherungsmaßnahmen.

Es wurde zwar wiederholt betont, daß es offensichtlich ist, daß das Regelwerk vollständig ist, dennoch soll nachfolgend ein formaler Beweis skizziert werden, der die Endlichkeit und Determiniertheit des Baumverfahrens zeigt. Es werden Mengen definiert, auf denen an Hand der Regeln Operationen eingeführt werden. Aus diesem Grund werden nachfolgend Schritt für Schritt die Regeln (abgekürzt durch "R") nochmals durchgegangen und so der Beweis aufgebaut.

R1: Es wird ein zentraler Prozeß mit Namen "taskname" angenommen.

R2: Unter der Annahme, daß die Bewachte Anweisung des zentralen Prozesses m Wächter enthält, mit $m \geq 1$, werden m zentrale Unterprozesse kreiert. Im folgenden wird einer ("x", mit $1 \leq x \leq m$) der Unterprozesse betrachtet.

R3: Eine Kante eines Baumes kann folgendermaßen beschrieben werden:

kante = (taskname_x, y, Z, partername),

mit taskname = Name des Zentrums, y = transmit XOR receive,

x = Unterprozeßnummer, mit $1 \leq x \leq m$, Z = Botschaftsname,

partername = Name des Partnerprozesses.

Der "Anfangsbaum" des Unterprozesses x läßt sich als folgende Menge darstellen:

Anfangsbaum = { kante(i) / $1 \leq i \leq n$ },

mit n = Anzahl der Elemente (Kommunikationsanweisungen) des Wächters.

Der Anfangsbaum entspricht der "Menge der ungesättigten Kanten" ("Kanten"),

während die "Menge der gesättigten Kanten" ("Kanten_g") zu Anfang leer ist:

Kanten = Anfangsbaum, Kanten_g = {}.

R4: Es werden Fragebotschaften für die n ungesättigten Kanten erstellt und an die Partnerprozesse gesendet. Es sind 2 verschiedene Antworten möglich (der Kommunikationspartner wird im folgenden mit KP abgekürzt):

- leerer Antwortbaum i = {} : die gefragte Kommunikationsanweisung kommt beim Partnerprozeß nicht vor

=> der Unterprozeß scheidet aus dem Verfahren aus.

- Antwortbaum i = { kante(j) / $1 \leq j \leq k_i(h)$ mit $1 \leq i \leq n$ AND KP an der i-ten Kante hat in seinem h-ten Wächter k KPs AND $1 \leq h \leq m'$ AND $\exists !: (1 \leq l \leq k_i(h) \text{ AND } kante(l) = kante(i))$ },

wobei kante(l) = kante(l') heißt, daß $Z = Z'$, $y = y'$ und taskname = partername[_w'] und partername[_x] = taskname' = > w = x' und x = w'.

Der Formalismus für Antwortbaum i besagt, daß der Kommunikationspartner die gesuchte Kommunikationsanweisung in seinem h-ten Guard, das selbst k Kommunikationsanweisungen aufweist, gefunden hat und genau eine der Anweisungen (" $\exists !$ ", d.h. "es existiert genau ein"), der Anweisung i entspricht.

Durch das Einfügen der Antwortbäume entstehen gesättigte Kanten, die folgendermaßen beschrieben werden:

kante_g = (taskname_x, y, Z, partername_h),

mit $1 \leq h \leq m'$;

oder:

kante_g = kante [partername_h \ partername],

was bedeutet, daß eine gesättigte Kante einer ungesättigten entspricht, wenn "partername_h" durch "partername" ersetzt wird.

Der Einbauvorgang vollzieht sich in 3 Schritten:

1. Kanten := Kanten \cup Antwortbaum i, mit $1 \leq i \leq n$;
2. Bilden der Menge der gesättigten Kanten:

$$\begin{aligned} \text{Kanten_g} = \{ & \text{kante}(i) / 1 \leq i \leq n \text{ AND } \exists! l: (1 \leq l \leq k_j(h) \text{ AND KP an} \\ & \text{der } i\text{-ten Kante hat im } h\text{-ten Wächter } k \text{ KPs AND kante}(l) = \\ & \text{kante}(i)) \} \cup \\ & \{ \text{kante}(l) / 1 \leq l \leq k_j(h) \text{ mit } 1 \leq i \leq n \text{ AND KP an der } i\text{-ten} \\ & \text{Kante hat im } h\text{-ten Wächter } k \text{ KPs AND } \exists! l': (1 \leq l' \leq k'_j(h') \\ & \text{mit } 1 \leq j \leq n \text{ AND KP an der } j\text{-ten Kante hat im } h'\text{-ten} \\ & \text{Wächter } k' \text{ KPs AND } j \neq i \text{ AND kante}(l) = \text{kante}(l')) \}. \end{aligned}$$

3. Bilden der Menge der ungesättigten Kanten:

$$\begin{aligned} \text{Kanten} = \{ & \text{kante}(l) / 1 \leq l \leq k_j(h) \text{ mit } 1 \leq i \leq n \text{ AND KP an der } i\text{-ten} \\ & \text{Kante hat im } h\text{-ten Wächter } k \text{ KPs AND kante}(l) \neq \text{kante}(i) \\ & \text{AND } \exists! l': (1 \leq l' \leq k'_j(h') \text{ mit } 1 \leq j \leq n \text{ AND KP an der} \\ & j\text{-ten Kante hat im } h'\text{-ten Wächter } k' \text{ KPs AND } j \neq i \text{ AND kante}(l) \\ & = \text{kante}(l')) \}. \end{aligned}$$

Bei der Menge der gesättigten Kanten handelt es sich jeweils um ein Kantenpaar aus verschiedenen Antwortbäumen, für die gilt: $\text{kante}(l) = \text{kante}(l')$.

R5: Durch den Einbauvorgang entstehen 2 disjunkte Mengen.

$$\begin{aligned} \text{Kanten_g} = \{ & \text{kante}(l) / \text{kante} = (\text{taskname_x}, y, Z, \text{partnername_h}) \text{ AND } 1 \leq l \\ & \leq n \text{ XOR } 1 \leq l \leq k_j(h) \text{ mit } 1 \leq j \leq n \text{ XOR } 1 \leq j \leq k'_j(h) \\ & \text{AND KP an der } j'\text{-ten Kante hat im } h\text{-ten Wächter } k \text{ KPs} \}; \\ \text{Kanten} = \{ & \text{kante}(l) / \text{kante} = (\text{taskname_x}, y, Z, \text{partnername}) \text{ AND } 1 \leq l \leq \\ & k_j(h) \text{ mit } 1 \leq j \leq n \text{ XOR } 1 \leq j \leq k'_j(h) \text{ AND KP an der } j'\text{-ten} \\ & \text{Kante hat im } h\text{-ten Wächter } k \text{ KPs} \}. \end{aligned}$$

Die Kanten aus jeweils einer der beiden Mengen (über die ganze Baumtiefe) unterscheiden sich in mindestens einer Komponente.

R6: Die Regel 4 wird nun mit einer Erweiterung solange wiederholt, bis entweder der Baum vollständig aufgebaut ist oder der Baum wegen eines Widerspruchs ausscheidet. Die Erweiterung besagt, daß nicht die Antwortbäume i (dies gilt nur für den Initialbaum), sondern die von $k_j(h)$ gesucht werden, mit: $1 \leq j \leq n \text{ XOR } 1 \leq j \leq k'_j(h) \text{ AND KP an der } j'\text{-ten Kante hat im } h\text{-ten Wächter } k \text{ KPs}$ (Menge der ungesättigten Kanten). Dabei muß gelten:

Satz: $\{ \text{partnername} / (\text{taskname_x}, y, Z, \text{partnername}) \in \text{Kanten} \} \cap$
 $\{ \text{partnername}' / (\text{taskname_x}, y, Z, \text{partnername}'_h) \in \text{Kanten_g} \} = \emptyset.$

Der Satz besagt, daß die Schnittmenge aus der Menge der Partnerprozeßnamen, die in den ungesättigten Kanten vorkommen, und der Menge der Partnerprozeßnamen, die in den gesättigten Kanten vorkommen, immer leer sein muß.

R7: Nach Abschluß des Baumverfahrens gilt:

Kanten = {} und

Kanten_g = { (taskname_x, y, Z, partnername_h) }.

Die Menge der ungesättigten Kanten ist leer und in der Menge der gesättigten Kanten sind alle Kommunikationskanten enthalten.

Die Menge der Kommunikationspartner läßt sich beschreiben mit:

{ taskname / kante[taskname \ taskname_x] \in Kanten_g }

R8: Es stehen jetzt m' Wächter (Kanten_g) zur Auswahl, mit $0 \leq m' \leq m$. Gilt $m' = 0$, so ist keine Kommunikation möglich, ansonsten muß eine Auswahl getroffen werden.

R9: Ein Wartebereich kann nie Zentrum werden.

R10: Der Antwortbaum gemäß Regel 4 (R4) besteht aus nur einer Kante und hat folgendes Aussehen:

Antwortbaum i = { kante_w = (taskname_x, "r", Z, partnername_0) / $1 \leq i \leq n$: kante(i) = kante_w }.

R11: Die Menge der gesättigten Kanten wird erweitert:

Kanten_g = Kanten_g \cup { kante_w / $\exists ! j, h: 1 \leq l \leq k_j(h)$ AND kante_w = kante(l) }.

Bei Vorhandensein von Wartebereichen müssen die bisher 2 disjunkten Mengen mit einer Schnittmenge akzeptiert werden:

- Teilmenge von Kanten_g = { (taskname_x, "t", Z, partnername_0) };

- Teilmenge von Kanten = { (taskname_x, "r", Z, taskname_w) / $w > 0$ }.

Das bedeutet nur, daß ein Partnerprozeß mit Wartebereich ($y = "t", w = 0$) auch mit einem Senden ($y = "r", w > 0$) an der Kommunikation beteiligt ist.

R12: Wie in Regel 11 (R11) sind die beiden Mengen nicht mehr disjunkt:

- Teilmenge von Kanten_g = { (taskname_x, "r", Z, partnername_0) };
- Teilmenge von Kanten = { (taskname_x, "t", Z, taskname_w / w > 0) }.

R13: Es müssen die vorhandenen Sendeanweisungen mit "partnername__0" vom Zentrum gezählt werden.

Zentral für diesen Beweis ist der mit R6 aufgestellte Satz, wonach die Schnittmenge der Menge der gesättigten und der Menge der ungesättigten Kanten leer sein muß. Da mit jedem neuen Antwortbaum diese beiden Mengen leicht angepaßt werden können, ist die Überprüfbarkeit auf eine Schnittmenge schnell durchgeführt. Die Endlichkeit des Verfahrens ist durch die stetige Verminderung der Kardinalität der Menge der ungesättigten Kanten gewährleistet. Die Verminderung kommt dadurch zustande, daß es nur endlich viele Prozesse gibt und deshalb nicht immer wieder neue Kommunikationspartner auftreten können. Das Verfahren ist deterministisch, da in der Menge der gesättigten Kanten sämtliche Kommunikationen erfaßt sind und die Kommunikationen zwischen zwei Partnern nur eine begrenzte Anzahl aufweisen kann.

5. Sicherungsmaßnahmen in einem Prozeßmodell

Wurden bisher ein Betriebssystemmodell und ein Botschaftenalgorithmus vorgestellt, um ein möglichst überschaubares Gesamtsystem zu erzielen, so sollen jetzt Maßnahmen untersucht werden, dieses System stabil zu gestalten. Sowohl Betriebssystem als auch Botschaftenalgorithmus erlauben ein "Deadlock"- und "Livelock"-freies Verhalten für den sogenannten "Normalfall". Normalfall heißt hier, daß sich die Prozesse so verhalten wie sie programmiert sind und, daß die Botschaften auch in der gewünschten Zeit dorthin gelangen, wohin sie gesendet worden sind. Aufgrund der blockierungsfreien Implementierung kommt es nur dann zu einem nicht erwünschten Verhalten des Systems, wenn der Entwickler "fehlerhaft" programmiert hat.

Ein neuer Aspekt ist allerdings das Eintreten nicht vorher bestimmbarer Ereignisse, wie z.B. der Verlust von Botschaften oder das Zerstören von Programmdateien. Es darf an dieser Stelle der Ausspruch, daß "durch Umkippen eines einzigen Bits schon alle Information zerstört sein kann", ruhig zitiert werden. Dazu gehört auch die Aussage (/HOFM87/), daß die heute häufigsten Fehler sogenannte "transiente" Fehler sind. Damit sind vor allem Hardware-Fehler gemeint, die einmalig und nicht reproduzierbar auftreten, ohne daß es sich hierbei um schwerwiegende oder andauernde Fehler handelt. Ebenso ist der Gesichtspunkt, daß Botschaften verloren gehen können, wichtig, vor allem dann, wenn man sich öffentlicher Netze oder Netze, deren Verhalten nicht kontrollierbar ist, bedient.

In diesem Kapitel werden deshalb Maßnahmen erläutert, die solche Fehler erfaßbar und wenn irgend möglich auch behebbar machen sollen. Diese Maßnahmen werden als Sicherungs- und Rücksetzfunktionen (Recovery) bezeichnet. In Abschnitt 1 sollen zunächst die dafür notwendigen Begriffe erläutert werden, und Abschnitt 2 beinhaltet eine Untersuchung, welche Fehler feststellbar und behebbar sind. Eine Darstellung möglicher Sicherungsmaßnahmen findet sich in Abschnitt 4, und zwar losgelöst von den Betriebssystem- und Botschaftskonzepten. Zuvor werden aber noch in einem gesonderten Kapitel Aussagen bezüglich des zeitlichen Verhaltens eines Botschaftenalgorithmus vorgestellt. In Abschnitt 5 wird schließlich die Integration der Sicherungsmaßnahmen in das Baumverfahren durchgeführt. Die Abschnitte 6 bis 8 dienen zur Untersuchung des Verhaltens im Fehlerfall, und befassen sich damit, welche Daten mit welchem Aufwand gesichert werden müssen. Kapitel 9 und 10 diskutieren schließlich die Parallelen zum "Three Phase Commit"-Protokoll sowie Überlegungen, wie dem Benutzer selbst Rücksetzmaßnahmen zur Verfügung gestellt werden können.

5.1. Begriffsklärung

Es ist zu untersuchen, was man unter einem "fehlerhaften" oder "zerstörten" Prozeß versteht. Im Prinzip gibt es 2 Arten (/LAMP78/), die fließend ineinander übergehen:

- **Malfunction** ("falsches Funktionieren"): Darunter versteht man das falsche Reagieren oder das bewußt fehlerhafte Verhalten eines Prozesses. Diese Art des Fehlers ist sehr schwer erfaßbar, denn dem falsch reagierenden Prozeß muß "verständlich" gemacht werden, daß er fehlerhaft ist. Diese Problematik rückt in die Nähe des "Halteproblems".
- **Failing, Failure, Defect** (Ausfall, "Nicht-mehr-funktionieren"): Darunter versteht man, daß ein Prozeß nicht mehr reagiert, also im einfachsten Fall, daß er ausgefallen ist. Ursachen für einen derartigen Fehler können sein: (kurzzeitiger) Stromausfall beim Prozessor, defekte Leitung(en) (z.B. durch Abreißen), Überschreiben der eigenen Daten oder Programme (nahe am Malfunction), etc.

Definition: Unter **Recovery** (Wiedergutmachen, Wiederaufsetzen, Rücksetzen) versteht man das Wiederherstellen eines betriebsfähigen Zustandes eines Prozesses (Prozeßsystems) nach Erkennen eines Defektes. Der betriebsfähige Zustand kann im einfachsten Fall ein Neustart des Prozesses sein, genauer ist allerdings ein Wiederanlaufen an einem sogenannten **Wiederaufsetzpunkt** (Sicherungs- oder Rücksetzpunkt).

Darunter versteht man eine gewählte Stoppstelle im Programm, an der der Zustand (Verwaltungsdaten, z.B. Befehlszähler) und die lokalen Daten eines Prozesses sichergestellt werden. Beim Wiederaufsetzen werden diese Daten restauriert und der Prozeß an dieser Stelle fortgesetzt.

5.2. Klassen von behebbaren Fehlern

Um Fehler klassifizieren zu können, muß zunächst der Aufbau eines Prozeßsystems analysiert werden.

Gegeben sei ein Netz von Rechnern (Prozessoren), die beliebig miteinander über Leitungen verbunden sind (siehe Bild 5-1). Dabei muß nicht jeder Rechner mit jedem Rechner verbunden sein. Die Verbindungsleitungen dienen lediglich als Transportmedium. Derartige Systeme werden als "lose gekoppelt" bezeichnet (siehe auch die Definition in 3.3.2.). Auf den Rechnern selbst befindet sich ein Prozeßsystem, wie in Kapitel 3 beschrieben, bestehend aus System-, Benutzer- und Treiber-Prozessen, sowie den Prozeßumschaltern und dem Leitungstreiber. Letzterer soll zumindest über ein einfaches "Routing" verfügen, um ankommende Botschaften, die nicht für den Rechner bestimmt sind, weiterzuleiten. Wird ein "intelligentes" Netz verwendet, z.B. ein lokales oder ein öffentliches Netz, so übernimmt dieses die Aufgabe des Routing (siehe auch Abschnitt 7.1.).

Definition: Unter "Routing" soll die Möglichkeit verstanden werden, Botschaften über ein (Rechner-) Netz zu übertragen, ohne daß dabei Start- und Zielknoten direkt miteinander verbunden sein müssen. Die dazwischenliegenden (Rechner-) Knoten sind in der Lage, die Botschaften weiterzureichen.

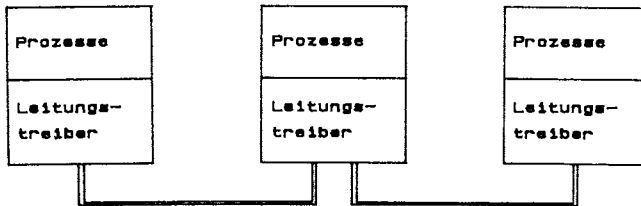


Bild 5-1: Lose gekoppeltes System

Bei Vorhandensein eines solchen Systems sind folgende Ausfälle, gestuft nach ihrer Wirkung, vorstellbar:

- Störung oder Ausfall einer, mehrerer oder sogar aller Leitungen:
 Dabei muß unterschieden werden, ob zwei Prozesse noch "eingeschränkt" oder überhaupt nicht mehr miteinander kommunizieren können.
 Unter "eingeschränkter Kommunikation" versteht man, daß zwei Prozesse nur mehr auf dem Umweg über mindestens einen Zwischenknoten (Prozessor) miteinander Botschaften austauschen können.
 Ist ein Weitervermitteln von Botschaften (Routing) sowieso vorgesehen, ist nur der Fall des abgetrennten Prozesses (Processors) von Bedeutung.
- Defekt eines Leitungstreibers: Dieser Fall ist gleichbedeutend mit dem Ausfall sämtlicher ankommender und abgehender Leitungen zu und von dem betreffenden Rechner.
- Ausfall eines oder mehrerer Prozesse auf einem Rechner.
- Ausfall eines ganzen Rechners: Damit sind nicht nur alle Prozesse, sondern auch die Prozeßumschalter und der Leitungstreiber außer Betrieb.
- Ausfall ganzer Teile des verteilten Systems oder Ausfall des ganzen Systems.

Wie im vorhergehenden Abschnitt (5.1.) erläutert, kann bei einem Defekt eines Prozesses zwischen "Malfunction" und "Failing" unterschieden werden. Im folgenden soll nur mehr auf das "Failing" eingegangen werden, das definiert wurde als die Möglichkeit, daß ein Prozeß, der in Ordnung ist, dem defekten Prozeß "klar machen" kann, daß er fehlerhaft ist. Beim "Malfunction", wenn also der defekte Prozeß falsch reagiert, muß damit gerechnet werden, daß er sich z.B. gegen Fehlermeldungen "wehrt".

Für die einzelnen Ausfallarten sollen noch kurz die Reaktionsmöglichkeiten erläutert werden:

Bei der Auflistung der Ausfälle wurden als erstes die defekten Leitungen erwähnt. Solange noch irgend eine Verbindung zwischen zwei Prozessen besteht, gibt es keinen zwingenden Grund für Maßnahmen. Besteht keine Verbindung mehr, so müssen die Prozesse so gut wie möglich in ihrem Programm fortsetzen. Hierbei sind vom Programmierer vorgesehene Benutzer-Timeouts hilfreich, um zu gewährleisten, daß die Prozesse nicht "ewig" in Kommunikationsanweisungen steckenbleiben (siehe dazu Abschnitt 4.3.). Den Defekt bei einer Leitung kann nur ein Mitglied des Personals für das Operating beheben.

Beim Defekt eines Leitungstreibers könnte man sich folgende Vorgehensweise vorstellen, wobei vorausgesetzt wird, daß zumindest noch einige primitive Funktionen ausführbar sind. Der Treiber überprüft jede ankommende Datenfolge auf ein bestimmtes Bitmuster. Trifft dieses Muster ein, so bedeutet dies, daß der Treiber sich selbst stoppt, neu lädt und dann wieder in einen normalen Funktionsmodus übergeht. Die Funktionen **Überprüfen**, **Stoppen**, **Laden** und **Starten** sollten, um möglichst sicher zu sein, in Hard- oder Firmware realisiert sein. Wird ein Leitungstreiber als fehlerhaft erkannt, so kann er dadurch "gezwungen" werden, sich selbst wieder neu zu initialisieren.

Einzelne Prozesse können ausfallen, indem ihre Daten oder Anweisungen durch sie selbst oder andere Programme (teilweise) zerstört werden. Da z.B. Prozesse in Kleinrechnern zumeist nicht in gegenseitig geschützten Speicherbereichen liegen, ist ein Zerstören durchaus möglich. Es muß nicht unbedingt ein Programmierfehler vorliegen. Ein Rücksetzen ist dabei auf die gleiche Art und Weise durchzuführen wie bei einem Programmierfehler.

Die Vorgehensweise bei Ausfall eines ganzen Rechners kann man sich ähnlich wie die beim Leitungstreiber vorstellen. Der Leitungstreiber muß nur dahingehend erweitert werden, daß er ein Initialisierungsprogramm lädt und startet, das die Initialisierung des Rechners, also der Benutzer-, System- und Treiberprozesse sowie der Prozeßumschalter, vornimmt. Damit können Fehler wie kurzzeitiger Stromausfall, Überschreiben eines großen Teils des Speichers durch einzelne Programme, u.ä. behoben werden.

Fallen in einem gekoppelten System mehrere Rechner aus, so ist im Prinzip die gleiche Vorgehensweise wie bei Ausfall eines einzelnen Rechners möglich. Fällt das System für längere Zeit aus, so muß der Bediener das System neu starten und initialisieren. Dabei wird bei einer Initialisierung von Prozessen davon ausgegangen, daß sie immer mit ihren zuletzt gültigen Rücksetzdaten gestartet werden.

Bislang wurde noch nicht auf die "Peripherie" des Prozeßsystems bzw. auf die Geräte, die außer Prozessoren und deren Verbindungsleitungen existieren, eingegangen. Dabei ist zu unterscheiden zwischen:

- peripheren Geräten, die als "sekundär" bezeichnet werden können.

Sekundäre Geräte sind Geräte, die lediglich zur Protokollierung o.ä. benutzt werden, z.B. Drucker oder Bildschirm. Sie sind also für den Fortbestand des Systems nicht lebenswichtig.

- peripheren Geräten, die als "primär" bezeichnet werden können.

Primäre Geräte sind Geräte, die zur Datenspeicherung, insbesondere der Sicherheitsdaten, oder als Informationsquelle dienen. Beispiele dafür sind der Plattenpeicher oder die Tastatur eines Terminals.

Bei der Unterscheidung zwischen primär und sekundär ist nicht die Art des Gerätes, sondern dessen Verwendung maßgeblich. Ein Plattenlaufwerk, das Statistikdaten aufnimmt, ist sekundär, dasjenige dagegen, das Daten des technischen Prozesses aufnimmt, ist primär. Sind an einem Terminal Benutzereingaben sehr wichtig, die Ausgaben auf Bildschirm dem Benutzer aber jederzeit anders zugänglich, so ist die Tastatur primär, der Bildschirm nur sekundär.

- primären und sekundären technischen Prozessen und ihren Verbindungsleitungen zu den Prozessoren (Rechnern);

Rücksetzmaßnahmen, insbesondere von Programmen können sich im Normalfall nicht auf Geräte oder technische Prozesse beziehen. Wird aber z.B. ein Drucker-Treiber zurückgesetzt, so kann möglicherweise ein Rücksetzsignal vom Treiber über die Leitung zum Drucker gesendet werden, ähnlich der Vorgehensweise beim Leitungstreiber. Die Druckgeräte, die heute im Gebrauch sind, verfügen aber zumeist noch nicht über derartige Möglichkeiten. Das Rücksetzen eines Gerätes bleibt zumeist dem Bediener überlassen, so daß im folgenden auf eine nähere Betrachtung bei Geräten verzichtet wird.

5.3. Einführung von Zeitüberwachungsmaßnahmen

Um komplexe Protokolle abwickeln zu können, ist eine sinnvolle Kontrolle des Botschaftenaustausches notwendig. Als sinnvolle und hier angewandte Kontrolle wird die Zeitüberwachung gesehen. Untersuchungen, vor allem in den Arbeiten von Lamport u.a. (/LAM978/, /LAMP82/) und Fischer u.a. (/FISC85/), haben ergeben, daß ohne Überwachung, insbesondere mit Hilfe von Auszeiten ("Timeouts"), kein endliches Protokoll existiert, so daß mehrere an einer Kommunikation beteiligte Prozesse zu einem gleichen Ergebnis gelangen.

In dem hier vorgestellten Protokoll sind mehrere Stufen von Zeitüberwachungsmaßnahmen notwendig:

Zeitüberwachung auf

- 1) Benutzerebene,
- 2) Betriebssystemebene - 1. Stufe,
- 3) Betriebssystemebene - 2. Stufe.

Die Kontrolle auf Benutzerebene ermöglicht, daß Operationen, die der Entwickler in einem bestimmten Zeitraum als abgeschlossen wünscht, entweder vollständig ausgeführt oder ohne Auswirkungen abgebrochen werden. Dies entspricht der Definition einer "atomaren Aktion", wie sie auch von Shrivastava (siehe Abschnitt 2.3.1.) angewandt wird.

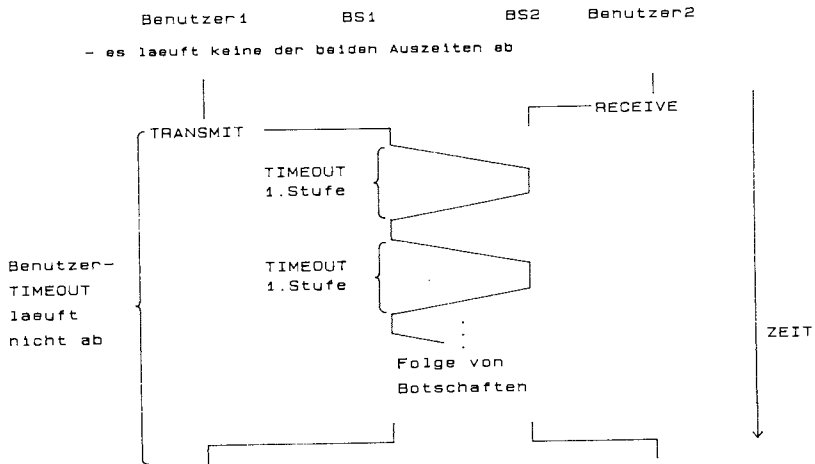
Dabei ist es gleichgültig, warum die Operation nicht in der gegebenen Zeit ausgeführt werden konnte. Es muß lediglich gewährleistet sein, daß die Nichtausführbarkeit sichtbar wird, z.B. durch die "TIMEOUT"- oder "OUTREACT"-Ausgänge der GUARDED STATEMENTS (/FHKK83/).

Ein (klassisches) Beispiel ist in der Verbindungsaufbauphase des Transportprotokolls T.70 nach CCITT (ISO-Schicht 4, /CCII83/) zu finden. Mit dem Erstellen eines Verbindungsaufbauwunsches startet der Senderprozeß eine Uhr, um diesen Aufbauversuch bzw. die Reaktion des Empfängers zu überwachen.

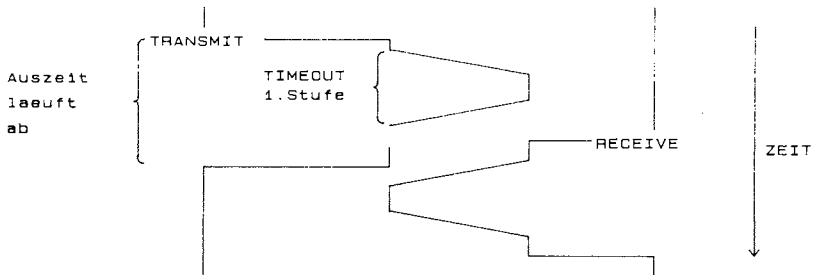
Für das Erkennen von defekten Prozessen ist dieser Timeout (Auszeit) ungeeignet, da eine Unterscheidung zwischen dem nicht mehr Reagieren (defekter Prozeß) und dem zu späten Reagieren auf dieser Ebene nicht möglich ist. Außerdem existiert unterhalb des Benutzerprotokolls noch das Betriebssystemprotokoll, z.B. das Baumverfahren, so daß die notwendige Information für diese Unterscheidung dort besser erfaßbar ist.

Mit der Zeitüberwachung auf Benutzerprogrammebene kann z.B. kontrolliert werden, ob die mit "TRANSMIT" abgesendete Nachricht auch in der angegebenen Zeit angenommen wird. Ähnlich darf man sich die Zeitüberwachung der 1. Stufe auf der Betriebssystemebene vorstellen. Im Baumverfahren ist nur das Zentrum aktiv und sendet "Anfragen" aus, auf die es Antworten erwartet. Die Partnerprozesse verhalten sich passiv und geben nur Antworten. Da auf jede Anfrage genau eine Antwort kommen muß, läßt sich dieses Frage-Antwort-Spiel vom Zentrum zeitlich leicht kontrollieren. Reagiert der Partnerprozeß nicht in der angegebenen Zeit, darf vermutet werden, daß eine Fehlersituation vorliegt.

Es läßt sich ein Zusammenhang zwischen der Auszeit auf Benutzerebene und der Auszeit auf Betriebssystemebene der 1. Stufe herstellen, da der Nachrichtenaustausch in eine Folge von Botschaften abgebildet wird. Es sind dabei 3 Fälle zu unterscheiden:



- es läuft die Benutzerauszeit ab



- es läuft der Betriebssystem-Timeout 1. Stufe ab



Im letzten Fall läuft zwangsläufig auch die Benutzerauszeit ab, was bedeutet, daß der Senderprozeß durch den TIMEOUT-Zweig die Anweisung verlassen wird. Wie der Fehlerfall zu behandeln ist, wird in den nächsten Abschnitten diskutiert.

Die Betriebssystemauszeit der 1. Stufe sorgt dafür, daß der Botschaftenaustausch innerhalb des Baumverfahrens kontrolliert werden kann. Er ist aber zu keiner Kontrolle in der Lage, falls ein Prozeß blockiert wird. Wenn wir nochmals den Fall 2 von oben betrachten, so wollte Benutzer 1 ein TRANSMIT absetzen, aber BSI bekam einen negativen Bescheid, da kein entsprechendes RECEIVE vom Benutzer 2 vorlag. Fehlt jetzt ein Benutzer-Timeout, so bleibt Benutzer 1 so lange blockiert bis Benutzer 2 die Empfangsanweisung durchläuft oder für immer, wenn er das nicht tut. Letzteres kann vom Entwickler auch beabsichtigt sein. Eine Blockierung hat aber dann keinen Sinn, wenn Benutzer 2 einen Zusammenbruch erfährt. Das würde bedeuten, daß Benutzer 1 auf ewig blockiert und Benutzer 2 auf ewig zerstört bliebe. Aus diesem Grund wird ein Timeout 2. Stufe eingeführt, der im Falle der Blockierung eines Prozesses eingeschaltet wird. Sein Ablauf bewirkt, daß der blockierte Prozeß beim Partnerprozeß wieder nachfragt. Für das dynamische Verhalten bei Verwendung eines Timeouts 2. Stufe können wieder 3 Fälle unterschieden werden:

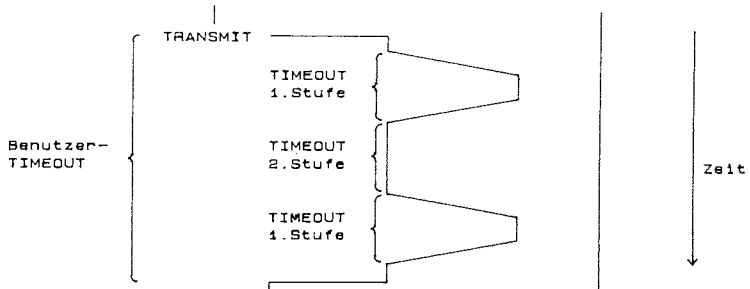
B1

BS1

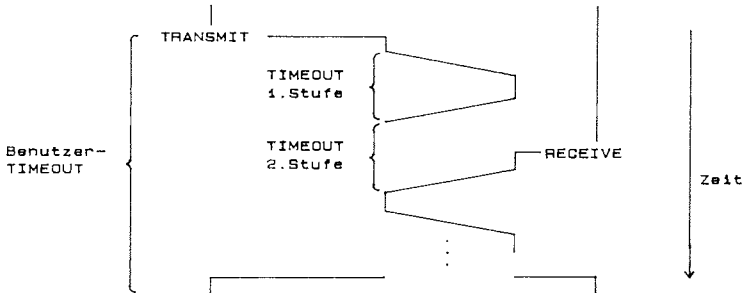
BS2

B2

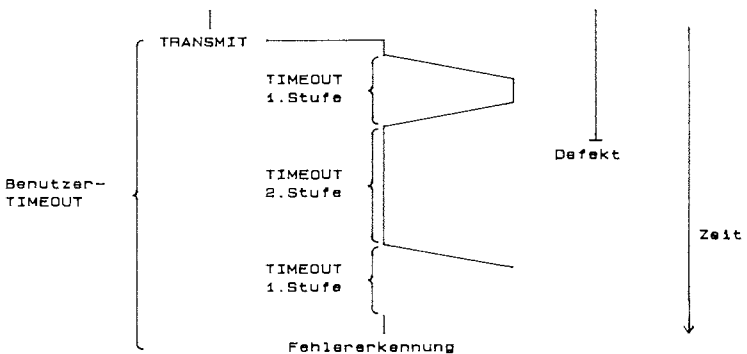
- es läuft die Auszeit der Stufe 2 ab und Benutzerprozess 2 ist nicht defekt



- es führt Benutzerprozess 2 eine RECEIVE-Anweisung vor Ablauf des Timeout 2. Stufe durch



- es läuft die Auszeit der Stufe 2 ab und Benutzerprozess 2 bzw. BS2 ist defekt



Aus Fall 3 wird deutlich, daß es immer der Timeout 1. Stufe ist, der den Ausfall bemerkt. Die Auszeit 2. Stufe bewirkt nur, daß ein Prozeß als Zentrum wieder aktiv wird.

Zusammengefaßt können folgende zwei Aussagen getroffen werden:

- Jede durch einen Prozeß ausgesendete Botschaft, auf die eine Antwort erwartet wird, wird mittels des BS-Timeouts der 1. Stufe überwacht.
- Wird ein Prozeß in seiner Kommunikationsanweisung blockiert, da sein oder seine Partner nicht zum Nachrichtenaustausch bereit sind, so wird der BS-Timeout der 2. Stufe gestartet.

Beim Botschaftenaustausch gilt also für die Betriebssystem-Zeitüberwachungen der 1. und 2. Stufe grundsätzlich, daß immer nur genau eine von beiden läuft! Lediglich die Uhren des Benutzer-Timeouts und der Auszeiten auf Betriebssystemebene können gleichzeitig laufen. Diese Überwachungsmaßnahmen ermöglichen eine korrekte und vor allem endliche Abwicklung des Protokolls. Außerdem können damit, wie festgestellt, defekte Prozesse erkannt werden und das ist wiederum Voraussetzung für mögliche Rücksetzmaßnahmen.

Die Feststellung, daß Zeitüberwachungen notwendig sind, führt auch zu Überlegungen darüber, wie groß diese Auszeiten sein müssen. Beim Timeout der 2. Stufe kann im gesamten System immer die gleiche Zeit verwendet werden, denn es geht um eine "Weckzeit", die unabhängig vom Verhalten der Kommunikationspartner und von den Verbindungswegen ist. Irgendwelche Geschwindigkeitsüberlegungen sind nicht nötig.

Beim Betriebssystem-Timeout der 1. Stufe ist es naheliegend, verschiedene Zeiten zu verwenden. Als Einflußgrößen sind dabei zu nennen:

- die Prozessorgeschwindigkeiten der beteiligten Rechner;
- die Länge der Verbindungswege;
- und vor allem die Übertragungsgeschwindigkeiten, die für die einzelnen Verbindungen verwendet werden.

Es ist deshalb ein weites Spektrum von der Verbindung zweier Prozesse auf dem gleichen, möglicherweise sehr schnellen Rechner und der langen, langsamen Verbindung zwischen zwei, möglicherweise sehr langsamen Rechnern gegeben. Es muß also individuell für jeden Fall eine geeignete Zeit festgelegt werden.

Bei der Vorstellung des Betriebssystemmodells wurde ein Timeout-Prozeß eingeführt. Dieser dient aber nur zur Abwicklung der Benutzer-Timeouts (siehe Anhang 1). Natürlich muß auch die Kommunikation zwischen Benutzer- und Timeout-Prozeß zeitlich überwacht werden, und zwar wieder durch die Betriebssystem-Timeouts der 1. und 2. Stufe.

Implementationsgesichtspunkte

Es steht noch die Frage offen, wie diese Betriebssystem-Timeouts verwaltet werden. In Abschnitt 3.3.3. wurde die Tabelle für den Prozeßumschalter für die Benutzer- und Systemprozesse (Prozeß-PU) eingeführt. In ihr werden auch für einen Prozeß ankommene Botschaften (Antworten) hinterlegt. Da für jede ausgesendete Botschaft (Anfrage) eine Zelle bereits vor Eintreffen der Antwort reserviert werden kann, kann in ihr der System-Timeout der 1. Stufe eingetragen werden. Trifft die Botschaft vor Ablauf der Zeit ein, so wird der Timeout durch die (Adresse der) Botschaft selbst überschrieben. Läuft zuvor die Auszeit ab, muß eine entsprechende Eintragung im Platzhalter erfolgen. Das Eintragen der Timeout-Einplanung in die Tabelle kann der Prozeß selbst durchführen.

Beim Betriebssystem-Timeout der 2. Stufe kann die gleiche Vorgehensweise angewendet werden, nur mit dem Unterschied, daß keine Botschaft erwartet wird. Meldet sich der Partner-Prozeß innerhalb der Zeit selbständig, wird der Timeout überschrieben, ansonsten wird mittels einer Botschaft überprüft, ob der Partner noch reagiert.

Da die Auszeiteintragen in der oben genannten Tabelle abgelegt werden, ist es naheliegend, den Prozeß-PU mit den Signalen der Systemuhr zu versorgen. Der Prozeß-PU dekrementiert bei jedem Signal die jeweils eingetragenen Timeouts um eine Zeiteinheit und überprüft, ob der Zähler zu Null geworden ist. In diesem Fall erzeugt der Prozeß-PU eine Art "stellvertretende" Botschaft und trägt sie anstatt der erwarteten Botschaft ein. Dies ist für den Prozeß das Kennzeichen, daß der Partnerprozeß nicht reagiert hat.

Die am Ende von Kapitel 3.3.4. angegebene Übersicht über die Aktivitätsreihenfolgen der einzelnen Elemente des Prozeßsystems kann für den Prozeß-PU folgendermaßen ergänzt werden:

- Der Prozeß-PU wird aktiv, wenn
 - = entweder ein Prozeß sich beendet oder blockiert, weil er z.B. eine Botschaft erstellt hat, und der Prozeß damit den Prozessor abgibt (Wartestellung);
 - = oder kein Treiber mehr läuft und auf Grund dessen der PU-Treiber den Prozessor abgibt;
 - = oder ein Zeitsignal eingetroffen ist. Es werden alle eingetragenen Betriebssystem-Timeouts dekrementiert und im Fall, daß die Zeit für eine bestimmte Botschaft abgelaufen ist, wird eine entsprechende Fehlerbotschaft für den betroffenen Prozeß erstellt und eingetragen.

Zur Integration der Auszeiten in das Baumverfahren

Der Ablauf des Baumverfahrens unter Berücksichtigung der Auszeiteinplanungen kann mit Hilfe eines Mealy- bzw. Übergangsautomaten grob dargestellt werden. Die Zustände sind dabei die Wartestellen des zentralen Prozesses. Das Eingangsalphabet sind die eintreffenden Antwortbotschaften von den Partnerprozessen oder das Ablaufen der Auszeiten und das Ausgabealphabet sind die zu treffenden Entscheidungen des Zentrums. Es läßt sich damit der Automat ("M") als ein Tripel, bestehend aus Ausgabe- ("A"), Eingabealphabet ("E") und Zustände ("Z") darstellen. In den Zuständen Z3 bis Z5 kann auch der Betriebssystem-Timeout 1. Stufe ablaufen. Da hier aber nur der normale Ablauf interessieren soll, wurde auf die herausführenden Kanten aus den besagten Zuständen verzichtet. Für diese Fälle hat das Zentrum gesondert zu reagieren, wie es z.B. in Abschnitt 5.7. beschrieben ist.

Nach Regel 8 wird nach erfolgreichem Baumaufbau der eigentliche Datenaustausch durch das Aussenden der "Reservierungs"-Botschaften eingeleitet und durch das Senden der Daten abgeschlossen. Das bedeutet, daß der Benutzer-Timeout nach dem Aussenden der Aufforderungen zum Datenaustausch nicht mehr wirksam werden darf, was auch in Bild 5-2 berücksichtigt wird, so daß der Übergang (E15/A15) zum Zustand Z5 und das Verlassen des Zustandes über E16/A16 als eine **atomare Einheit** zu verstehen ist. Das Eintreffen des Benutzer-Timeouts wird deshalb nur mehr registriert, aber nicht mehr ausgeführt. Dieser Zustand und seine Ein- und Ausgänge dienen zum Datenaustausch, der entweder vollständig oder überhaupt nicht ausgeführt werden soll. In letzterem Fall kann dann der Benutzer-Timeout wieder wirksam werden.

M = (A, E, Z), mit

A : A1 = Kreieren der Unterprozesse, Starten des Benutzer-Timeouts

A2 = Unterprozesse beenden, Verlassen der Kommunikationsanweisung über Timeout-Zweig

A3 = Betriebssystem-Timeout 2. Stufe starten

A4 = Reservierungen aussenden, Betriebssystem-Timeout 1. Stufe starten

A5 = Registrieren

A6 = Merken

A7 = Zurückziehen der Reservierung an alle anderen

A8 = Initialisieren des Baumès (nur beim 1. Durchlauf), Feststellen der Partnerprozesse, Auswahl eines Prozesses, Nachfrage nach Baum (ungesättigte Kante), Betriebssystem-Timeout 1. Stufe starten

A9 = Rückmeldung, daß kein Baumaufbau möglich

A10 = Betriebssystem-Timeout 1. Stufe stoppen, Baum einbauen, Erstellen der gesättigten und ungesättigten Kanten, Überprüfen auf Widerspruchsfreiheit

A11 = Widerspruch

A12 = Kein Widerspruch => Baum vollständig?

A13 = Ja => gesättigten Baum an Zentrum melden, Unterprozeß beenden

A14 = Nein

A15 = Aufforderung zum Datenaustausch und eigene Daten senden, Betriebssystem-Timeout 1. Stufe starten

A16 = Benutzer-Timeout verwerfen, Kommunikationsanweisung verlassen

A17 bis A21 = keine Reaktion

E : E1 = Betreten der Kommunikationsanweisung

E2 = Benutzer-Timeout

E3 = letzte Rückmeldung, kein Baum zur Auswahl

E4 = letzte Rückmeldung, ein Baum auswählbar

E5 = Reservierungsantwort

E6 = Benutzer-Timeout

E7 = mindestens eine negative Reservierungsantwort oder Benutzer-Timeout abgelaufen

E8 = Unterprozeß kreiert

E9 = kein Antwortbaum verfügbar

E10 = Antwortbaum verfügbar

E11 bis E14 = keine Eingabe vorhanden

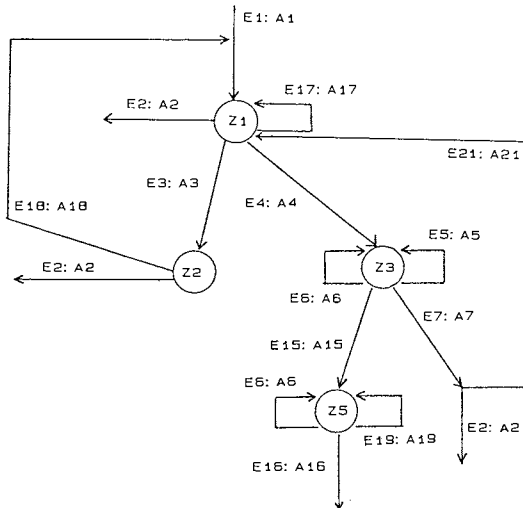
E15 = alle Reservierungsantworten positiv

E16 = alle Nachrichten empfangen

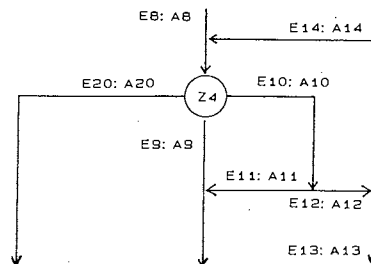
E17 = Rückmeldung von Unterprozeß

E18 = Betriebssystem-Timeout 2. Stufe
 E19 = Daten empfangen
 E20 = Hauptprozeß beendet Unterprozeß
 E21 = Benutzer-Timeout noch nicht abgelaufen

Z : Z1 = Warten auf Benutzer-Timeout oder Rückmeldung von Unterprozeß
 Z2 = Warten auf Benutzer- oder Betriebssystem-Timeout 2. Stufe
 Z3 = Warten auf Reservierungsantwort, Benutzer- oder Betriebssystem-Timeout 1. Stufe
 Z4 = Warten auf Antwortbaum oder Betriebssystem-Timeout 1. Stufe
 Z5 = Warten auf Nachrichten (Daten) für Empfangsanweisungen, Benutzer-Timeout oder Betriebssystem-Timeout 1. Stufe



Automat für den zentralen Hauptprozeß



Automat für den zentralen Unterprozeß

Bild 5-2: Übergangsautomat für das Baumverfahren und die Auszeiteinplanungen

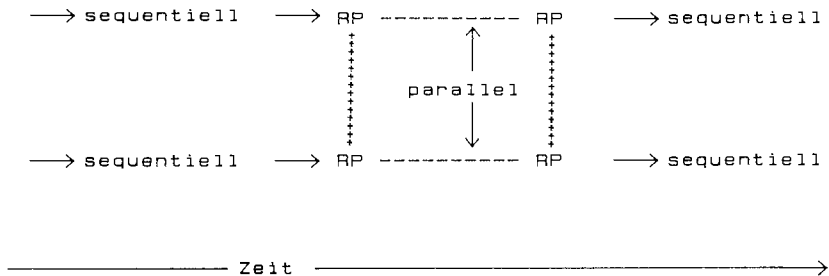
5.4. Das Setzen von Sicherungspunkten in verteilten Systemen

Im vorhergehenden Abschnitt wurden die Möglichkeiten dargestellt, wie Fehler bzw. Ausfälle von Prozessen in einem verteilten System erkannt werden können und wie, wenn ein Ausfall erkannt wird, der betreffende Prozeß, seine mit ihm kommunizierenden Partner und damit das Gesamtsystem, wieder in einen betriebsfähigen Zustand gebracht werden können. Mit Sicherheit ist letzteres möglich, wenn man den Prozeß neu startet, was aber kaum wünschenswert ist. Probleme tauchen auf, wenn dieser Prozeß mit anderen Prozessen kommuniziert hat, denn dann müssen diese Kommunikationen auch wiederholt werden. Dazu sind die Partnerprozesse aber nicht in der Lage, da sie regulär in ihrem Programm fortfahren wollen. Als Lösung bietet sich an, sogenannte Sicherungs- (Rücksetz-, Wiederaufsetz-, "Recovery"-) Punkte (kurz **RP**) zu setzen, an denen wichtige Daten (siehe Kapitel 5.6.) gesichert werden. Der Prozeß kann dann von einem derartigen Punkt aus fortgesetzt werden.

Die Wiederaufsetzpunkte dürfen aber nicht beliebig gesetzt werden. So kann es bei einem falschen Setzen und als Folge von Prozeßkommunikationen zum sogenannten "Domino"-Effekt (/WEBE83/, /ANKN83/) kommen, bei dem sich die Prozesse gegenseitig, möglicherweise bis zu ihren Startpunkten, "aufrollen".

Damit ein Prozeß feststellen kann, ob er fehlerhaft ist oder nicht, gibt es u.a. zwei unterschiedliche Ansätze: entweder der betreffende Prozeß überprüft in einer Art Selbsttest eine bestimmte Datenstruktur auf Richtigkeit, oder ein Kommunikationspartner, z.B. das Zentrum, stellt fest, ob der Prozeß noch funktionstüchtig, d.h. kommunikationsfähig, ist. Im folgenden soll nur letzterer Ansatz verfolgt werden.

Bei der Wahl von Rücksetzpunkten für Sicherungsmaßnahmen wird von folgender Überlegung ausgegangen: Programme führen zu einer Kommunikationsanweisung aus, in denen sie Daten mit anderen Prozessen austauschen, im folgenden als parallele Programmteile bezeichnet. Zum anderen führen sie sequentielle Programmteile aus, in denen sie nur auf ihre lokalen Daten zugreifen. Es erscheint deshalb zweckmäßig, die Rücksetzpunkte so zu legen, daß sie zwischen den sequentiellen und parallelen Programmstücken liegen. Graphisch läßt sich der Sachverhalt so darstellen:



Die Rücksetzpunkte der Prozesse A und B müssen dabei gemeinsam gesetzt werden, was durch die "+"-Zeichen angedeutet ist.

Will man eine syntaktische Regel, z.B. für eine Laufzeitprozedur, angeben, nach der die Anweisungen für das Setzen der Rücksetzpunkte abgelegt werden, so hat sie folgenden Aufbau:

Programm ::= { RP , sequentiell / RP , parallel } + RP
 mit RP = Recovery-Punkt,
 + = 1, 2, ...-faches Vorkommen,
 , = aufeinanderfolgend,
 / = alternativ.

Außerdem sollte es für den Benutzer die Möglichkeit geben, selbst Sicherungspunkte definieren zu können, z.B. durch Anweisungen in einer Spezifikations- oder höheren Programmiersprache. Ein entsprechender Vorschlag wird in Abschnitt 5.10 diskutiert.

Besteht ein Prozeß nur aus sequentiellen Anweisungen, so liegt es sowieso in der Hand des Programmierers, Rücksetzpunkte selbst einzuführen und dafür zu sorgen, daß dieser Prozeß von anderen Prozessen überwacht wird.

Für die Situation, daß ein Ausfall eines Prozesses bemerkt wird, gibt es für den Rückgriff auf die Sicherungsdaten zwei Fälle zu unterscheiden:

- 1) Bild 5-3: Prozeß A will eine Kommunikation zu Prozeß B aufbauen, aber dieser meldet sich nicht. Also ist B während seines sequentiellen Ablaufes abgestürzt, und es muß auf die Daten, die nach der Beendigung seiner letzten parallelen Aktivität gesichert wurden, zurückgegriffen werden. B muß deshalb seine sequentiellen Anweisungen wiederholen. A bleibt solange blockiert, bis entweder sein Benutzer-Timeout abläuft oder B "aufgeholt" hat.

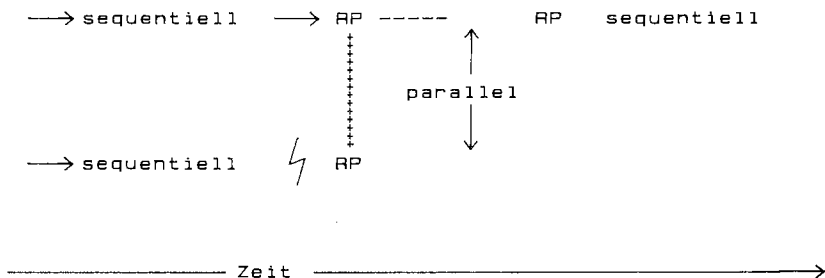


Bild 5-3: Fehler bei Prozeß B im sequentiellen Programmteil

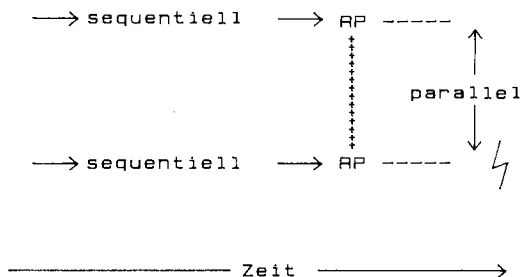


Bild 5-4: Fehler bei Prozeß B im parallelen Programmteil

- 2) Bild 5-4: Prozeß A und B stehen in Kommunikation miteinander, und B stürzt dabei ab. A und B müssen bei einem Wiederaufsetzen auf jeweils den Punkt vor Beginn des parallelen Programmteils zurück und die dort gesicherten Daten wieder holen.

Aus den beiden Beispielen wird deutlich, daß man für eine Vereinfachung oder Optimierung auch auf den Rücksetzpunkt vor den Kommunikationsanweisungen verzichten könnte (Fall 2). Stürzt ein Prozeß während der Kommunikation ab, so muß er aber auch auf den Punkt vor den sequentiellen Anweisungen zurückgesetzt werden.

Im folgenden soll bewiesen werden, daß die vorgesehenen Rücksetzpunkte und die Bedingungen, wo sie zu setzen sind, für das nicht Zustand kommen des Domino-Effektes ausreichend sind.

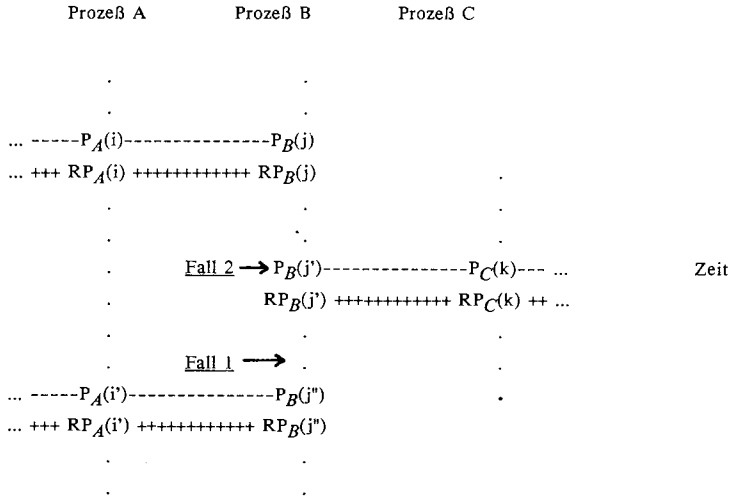
Beweisskizze:

Betrachtet werden 3 Prozesse A, B und C, die miteinander in Kommunikation stehen, wobei A und C auch noch mit anderen Prozessen Nachrichten austauschen können. Es werden nur die Wiederaufsetzpunkte nach Abschluß einer Kommunikation betrachtet. Dem Beweise liegt die Forderung zu Grunde, daß bei Rücksetzen eines fehlerhaften Prozesses lediglich der (die) direkt mit diesem Prozeß in Kommunikation stehende(n) Prozeß (Prozesse) zurückgesetzt werden muß (müssen) und keine anderen Prozesse des gesamten Prozeßsystem davon betroffen sind. So wird im Beweis wieder unterschieden zwischen einem Fehler im sequentiellen Teil eines Prozesses und im parallelen Teil, wobei nur im letzteren die Partnerprozesse ebenfalls zurückgesetzt werden müssen.

Die Abkürzungen und Zeichen haben folgende Bedeutung:

- $P_A(i)$: paralleles Programmstück (Kommunikation) des Prozesses A zum Zeitpunkt i;
- $RP_A(i)$: Rücksetzpunkt des Prozesses A zum Zeitpunkt i (nach Abschluß der Kommunikation zum Zeitpunkt i);
- für Prozeß A gilt für die Zeitpunkte folgende Relation: $i < i'$;
- für Prozeß B gilt für die Zeitpunkte folgende Relation: $j < j' < j''$;
- für Prozeß C gibt es nur einen zu betrachtenden Zeitpunkt k.

Der zeitliche Ablauf sei folgender:



Es sind zwei Fälle bei Programmfehlern zu unterscheiden:

Fall 1: Fehler im sequentiellen Programmteil:

- 1.: B wird auf RP_B(j) zurückgesetzt und RP_B(j') ist nicht gesetzt => Widerspruch zu der Forderung, daß RP_B(j') gesetzt sein muß;
- 2.: B wird auf RP_B(j) zurückgesetzt und RP_B(j') ist gesetzt => nur wenn (j' <= j) => Widerspruch;

also muß B auf RP_B(j') zurückgesetzt werden, was keine Auswirkungen auf andere Prozesse hat.

Fall 2: Fehler im parallelen Programmteil:

- 1.: B wird auf RP_B(j) zurückgesetzt, vor P_B(j') => unproblematisch;
- 2.: B wird auf RP_B(j) zurückgesetzt, während P_B(j') => unproblematisch;
- 3.: B wird auf RP_B(j) zurückgesetzt, nach P_B(j') => siehe Fall 1;

also muß B auf RP_B(j) zurückgesetzt werden, was keine Auswirkungen auf andere Prozesse hat, außer auf C, der auf RP_C(k) zurückgesetzt werden muß, da er direkt mit B kommuniziert.

Die kurze Beweisskizze kann natürlich nur einen geringen Einblick in die Problematik des Dominoeffekts erbringen. In der Arbeit von Bathelt (/BATH86/) wird ein umfangreiches analytisches Prozeßmodell zu Grunde gelegt, mit dessen Hilfe und darauf basierender exakter Beweise Kriterien für die Dominoeffektfreiheit abgeleitet werden. Die Quintessenz dieser Kriterien ist die Forderung nach der Berechenbarkeit von Konstanten, die ausdrücken, daß bestimmte Rechenschritte endlich sind. Derartige Rechenschritte sind z.B. das Zurücksetzen von Aktionen nach einem erkannten Fehler, die Aktionen zwischen zwei Rücksetzpunkten oder die Aktionen zwischen der Veränderung eines Variablenwertes und der Überprüfung seiner Richtigkeit. Mit obigem Beweisansatz sollte deshalb der Versuch unternommen werden, zu zeigen, daß sämtliche Aktionen, sowohl der parallelen, als auch der sequentiellen Programmteile aller an einem Prozeßsystem beteiligter Prozesse endlich, also durch Konstanten ausdrückbar sind.

5.5. Zusammenhang zwischen dem Setzen von Sicherungspunkten und dem Baumverfahren

In Kapitel 4.2. wurde ein Botschaftenprotokoll, das Baumverfahren, mit Hilfe von Regeln erläutert. Diese Regeln sorgen dafür, daß ein (Zentrums-) Prozeß, der kommunizieren will und dieses Verfahren anwendet, gewisse zeitlich aufeinanderfolgende Phasen durchläuft. In Bild 5-5 sind die Phasen eines Prozesses aus der Benutzersicht dargestellt. Dabei bleibt das Baumverfahren verborgen, da es während des Blockiert-Zustandes, quasi in der Betriebssystemsicht, abläuft. Es gibt insgesamt drei Phasen, die jeweils durch den zweiten Index identifiziert werden. Der 1. Index deutet an, daß es sich um die Benutzersicht handelt.

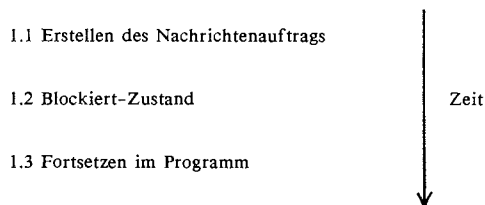


Bild 5-5: Phasen eines Prozesses aus Benutzersicht beim Nachrichtenaustausch

Betrachtet man die Phasen eines Zentrums aus der Betriebssystemsicht, so zeigt Bild 5-6, daß das Baumverfahren nur einen Teil, wenn auch den Hauptteil, ausmacht. Der 1. Index besagt, daß es sich um die Betriebssystemsicht (2) handelt und der 2. Index deutet wieder die aus Bild 5-5 bekannten drei Phasen an. Hierbei kann Phase 2 in zwei Unterphasen unterteilt werden.

Bild 5-7 zeigt die Abbildung der Phasen der Benutzer- auf die der Betriebssystemsicht.

Um nun den Zusammenhang zwischen diesen Phasen und den in Abschnitt 5.4. eingeführten Sicherungspunkten herzustellen, erinnern wir uns, daß wir die Punkte vor und nach der Kommunikation eingesetzt haben. Das bedeutet für die Phasen aus der Benutzersicht, daß wir die Sicherungspunkte vor der Phase 1.1 und nach der Phase 1.3 setzen müssen.

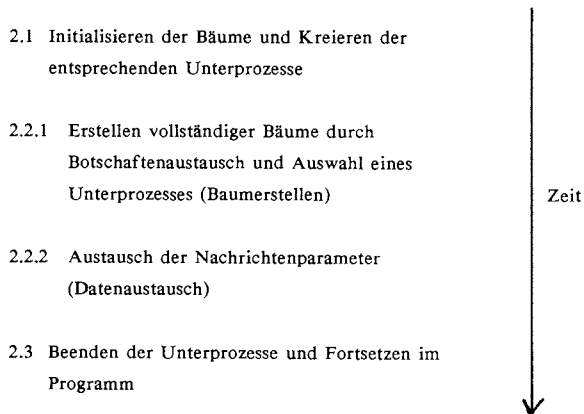


Bild 5-6: Phasen eines Prozesses aus Betriebssystemsicht beim Botschaftenaustausch

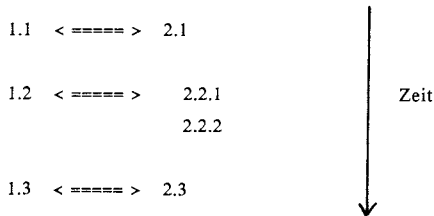


Bild 5-7: Abbildung der Protokollphasen aus der Benutzersicht auf die aus der Betriebssystemsicht

Entsprechendes gilt für die Phasen in der Betriebssystemsicht. Dort werden die Rücksetzpunkte vor 2.1 und nach 2.3 gesetzt. Die Phase 2.2 wird in zwei unterschiedliche Phasen unterteilt, in denen zwei verschiedene Arten von Kommunikationen stattfinden, nämlich "Baumerstellen" und "Datenaustausch". Wenden wir wieder den Grundsatz an, daß vor und nach einer Kommunikation ein Sicherungspunkt zu setzen ist, so ergeben sich vier zusätzliche Rücksetzpunkte. Bild 5-8 zeigt als Zwischenergebnis die Phasen aus der Benutzer- und der Betriebssystemsicht, erweitert um die Rücksetzpunkte (kurz RP), sowie die dazugehörigen Abbildungen aufeinander.

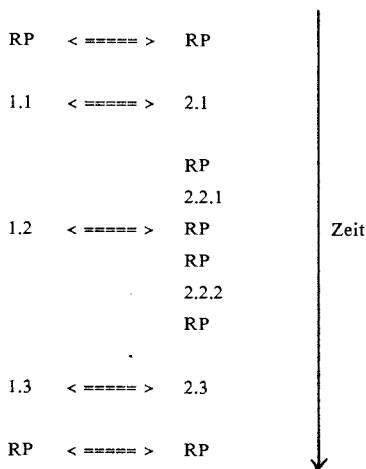


Bild 5-8: Protokollphasen eines Prozesses mit allen Rücksetzpunkten

Es ist offensichtlich, daß einige Sicherungspunkte redundant sind, insbesondere wenn sie direkt aufeinanderfolgen. So ist ein Rücksetzpunkt zwischen den Phasen 2.2.1 und 2.2.2 überflüssig. Die Phasen in der Betriebssystemschicht sind dadurch charakterisiert, daß die Phasen 2.1 und 2.3 als sequentielle und die Phasen 2.2.1 und 2.2.2 als parallele Programmteile aufzufassen sind. Aus Abschnitt 5.4. wissen wir, daß der Sicherungspunkt zwischen dem sequentiellen und dem parallelen Abschnitt weggelassen werden kann. Da die Phase 2.1 noch dazu sehr kurz ist, kann der Sicherungspunkt vor 2.2.1 gelöscht werden. Da nach Phase 2.3 ebenfalls ein sequentieller Programmabschnitt folgt, ist auch der Sicherungspunkt nach 2.3 redundant. Die gleiche Argumentation gilt für den Sicherungspunkt vor 2.1, da davor ebenfalls ein sequentieller Abschnitt liegt.

Damit erhalten wir Bild 5-9, in dem die Anzahl der Sicherungspunkte auf ein notwendiges Maß reduziert ist. Der Rücksetzpunkt vor 1.1 ist dabei ebenfalls weggelassen worden, da er vor einem parallelen Abschnitt nicht notwendig ist. Das Bild zeigt auch, daß die Abbildung der Phasen nicht unbedingt auch eine Abbildung der Rücksetzpunkte aufeinander bewirkt.

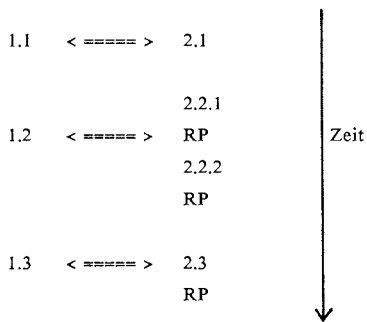


Bild 5-9: Protokollphasen eines Prozesses mit den nicht redundanten Rücksetzpunkten

Betrachten wir die beiden noch verbliebenen Wiederaufsetzpunkte aus der BS-Sicht, so kann folgendes festgestellt werden:

Der Rücksetzpunkt nach 2.2.2, kurz 2.2.2RP genannt, ist deswegen notwendig, weil damit die Kommunikation und die Synchronisation zwischen den Prozessen endgültig abgeschlossen wird. Der nachfolgende "Absturz" von einer der beteiligten Tasks darf keine Rückwirkung mehr auf die anderen haben! Es ist sozusagen ein "Point of no return".

Der Wiederaufsetzpunkt vor 2.2.2, kurz RP2.2.2 genannt, ist lediglich sinnvoll, nicht aber unbedingt notwendig. RP2.2.2 ermöglicht, daß das Baumverfahren nicht wiederholt werden muß, sondern im Fehlerfall nur die sogenannte Datenaustauschphase von 2.2.2. Läßt man diesen Rücksetzpunkt weg, müssen sowohl Baumverfahren als auch Datenaustausch wiederholt werden.

Als Fazit können wir feststellen:

2.2.2RP ist notwendig, RP2.2.2 ist sinnvoll.

Die letzte noch zu beantwortende Frage ist, wie das Setzen eines Rücksetzpunktes und das Sichern der Daten in das Baumverfahren bzw. in den Datenaustausch integriert werden können. Dazu läßt sich folgende Festlegung treffen:

Ergänzung zu Regel 8:

Das Baumverfahren endet nach Regel 8 mit der Reservierung der beteiligten Kommunikationspartner durch das Zentrum und dem darauffolgenden Datenaustausch. Das Protokoll muß nun dahingehend ereitert werden, daß, nachdem ein Prozeß erfolgreich seine Daten gesendet bzw. empfangen hat, er den Rücksetzpunkt 2.2.2RP setzen und den erfolgreichen Abschluß von Datenaustausch und Sicherungspunktsetzen dem Zentrum melden muß. Für den RP2.2.2 gilt lediglich, daß das Zentrum den Baum vollständig aufgebaut hat und den Sicherungspunkt setzen kann.

Es ist die Aufgabe des Zentrums, die soeben genannten Abschlußmeldungen von den Partnerprozessen zu sammeln und für den Fall, daß alle jeweils eine positive Meldung gesendet haben, allen mitzuteilen, daß sie in ihrem Programm fortsetzen können.

Dieser Sachverhalt ist in Bild 5-10 festgehalten, in dem nur die Betriebssystemebene gezeigt wird. Wichtig dabei ist, daß die Prozesse beim Setzen des Sicherungspunktes 2.2.2RP den alten Sicherungspunkt, der (irgendwann) davor gesetzt wurde, noch nicht löschen. Das dürfen sie erst tun, wenn sie vom Zentrum die positive Abschlußmeldung erhalten haben (Phase 2.2.3). Würden sie ihn löschen, könnten sie im Falle einer negativen Abschlußmeldung nicht mehr zurücksetzen.

Zum Schluß seien noch die möglichen Vorgehensweisen in Fehlerfällen aufgegriffen:

Kommt es in 2.1, 2.2.1, 2.2.2 oder schon früher (sequentieller Programmteil) zu einem Fehler, so müssen die Prozesse auf den letzten Sicherungspunkt zurückgesetzt werden.

Kommt es in 2.2.3 zu einem Fehler, so kann folgendermaßen verfahren werden:

- 1) ein Partnerprozeß erhält die Benachrichtigung, z.B. wegen eines Leitungsfehlers, nicht: der Prozeß fordert sie nochmals an und erhält sie daraufhin vom Zentrum;
- 2) das Zentrum ist abgestürzt: alle Prozesse setzen auf ihren letzten Rücksetzpunkten auf;
- 3) ein Partnerprozeß stürzt ab: er kann auf den Punkt 2.2.2RP zurückgesetzt werden, ohne daß die anderen Prozesse zurücksetzen müssen.

aus der Sicht des Zentrums:

aus der Sicht der Partnerprozesse:

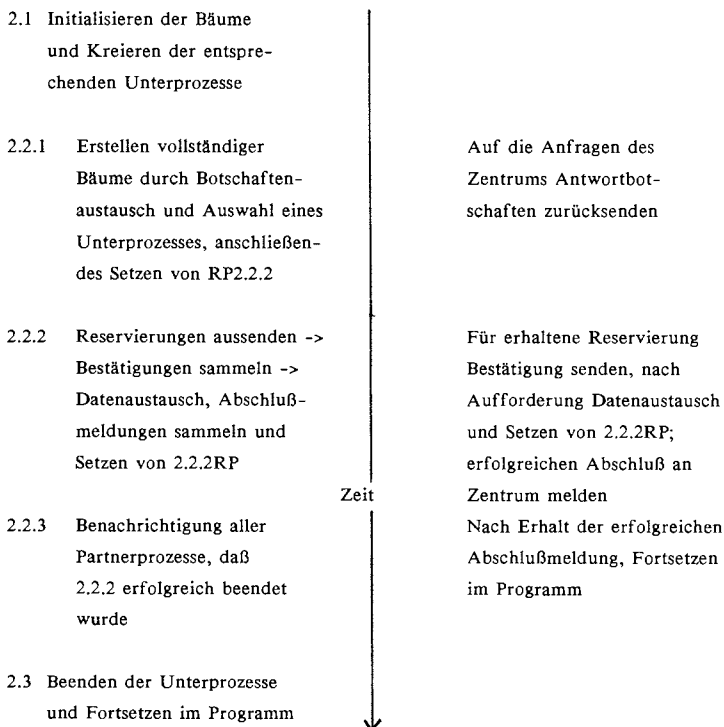


Bild 5-10: Protokollphasen der Prozesse aus der Betriebssystemsicht mit Setzen von Sicherungspunkten

Zum Abschluß von Abschnitt 5.4. wurde ein Mealy-Automat angegeben, in dem die Zustände des Zentrums innerhalb des Baumverfahrens und die möglichen Auszeiten aufgezeigt wurden. Das Setzen von Rücksetzpunkten ist lediglich eine konsistente Ergänzung der Zustandsübergänge in bzw. aus dem Zustand Z5, wobei auf Grund der Erweiterung der Regel 8 der Zustand Z5, wie unten gezeigt, erweitert wird. So wird bei der Auswahl eines Baumes bzw. bei der Reservierung der Partnerprozesse der Rücksetzpunkt RP2.2.2 gesetzt. Der Wiederaufsetzpunkt 2.2.2RP dagegen kann, nachdem alle positiven Transfermeldungen von den Partnerprozessen eingetroffen sind, noch vor dem Wegsenden der Fortsetzbotschaften eingefügt werden. Das Setzen der Rücksetzpunkte verlängert also nur die Übergänge, ändert aber an den Zustandsü-

bergängen selbst nichts. Das wird auch im Bild sichtbar, das sich rein äußerlich nicht ändert, sondern nur die Beschriftung der Zustände und Kanten erfährt eine Erweiterung.

Durch das Setzen der Rücksetzpunkte wird bereits die in Abschnitt 5.4. gemachte Feststellung erhärtet, daß es sich beim Zustand Z5 und seinen Ein-/Ausgängen bzw. bei der Phase 2.2.2 aus Bild 5-10 um eine **atomare Einheit** handelt, die jetzt durch die Rücksetzpunkte geklammert wird. Als Basis für diese Tatsache dient allerdings nicht mehr nur die Regel 8, sondern die Ergänzung zu Regel 8.

$M = (A, E, Z)$, mit

A : A1 bis A14 = keine Änderung

A15 = Setzen von RP2.2.2, Aufforderung zum Datenaustausch und eigene Daten senden, Betriebssystem-Timeout 1. Stufe starten

A16 = Benutzer-Timeout verwerfen, Setzen von 2.2.2RP, Fortsetzbotschaft an alle Partner, Kommunikationsanweisung verlassen

A17 bis A21 = keine Reaktion

E : E1 bis E14 = keine Änderung

E15 = alle Reservierungsantworten positiv

E16 = alle Nachrichten und Transferendemeldungen empfangen

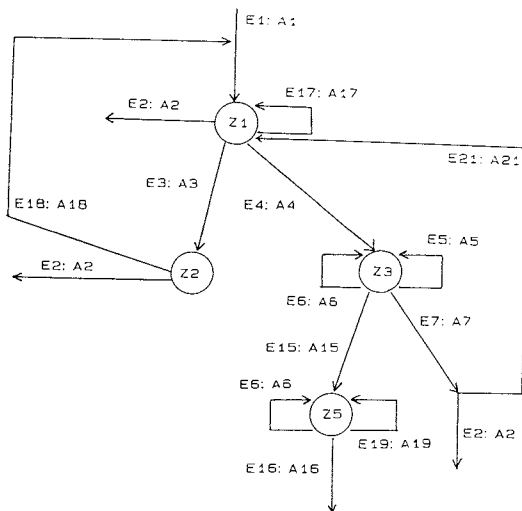
E17 bis E18 = keine Änderung

E19 = Daten oder Transferendemeldungen empfangen

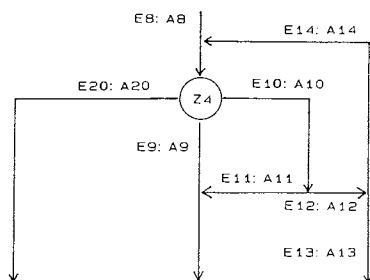
E20 bis E21 = keine Änderung

Z : Z1 bis Z4 = keine Änderung

Z5 = Warten auf Nachrichten (Daten) für Empfangsanweisungen, Transferendemeldungen von Partnerprozessen, Benutzer-Timeout oder Betriebssystem-Timeout 1. Stufe



Automat für den zentralen Hauptprozeß



Automat für den zentralen Unterprozeß

Bild 5-11: Ergänzter Übergangsautomat aus Bild 5-2

5.6. Wahl der Sicherungsdaten

5.6.1. Datensicherung bei Prozessen und Prozeßumschaltern

In den Abschnitten 4 und 5 wurde festgelegt, wie Sicherungspunkte zu setzen sind, um Prozesse nach einem Fehler wieder in einen konsistenten Zustand zu bringen. Das setzt voraus, daß Datenbestände existieren, in denen die Prozesse so gespeichert sind, daß sie in diesen Zustand überführt werden können. Als Medien zum Speichern sind heutzutage Platten oder sonstige "Hard-Disk"-Geräte gebräuchlich. Auf diesen müssen sich zunächst die Programmläufe befinden, da die Prozesse überall im Programm fortsetzbar sein müssen. Außerdem können beliebige Programmteile eines Prozesses zerstört worden sein, so daß alle Anweisungen gesichert sein müssen. Sie brauchen aber nicht bei jedem Rücksetzpunkt gesichert werden, da sie unverändert bleiben, so daß eine einmal erstellte Plattenkopie ausreicht.

Anders verhält es sich bei dem zweiten Bestandteil eines Prozesses, den Daten. Bei diesen kann unterschieden werden zwischen den vom Programmierer deklarierten Programmdateien und den Verwaltungsdaten. Zu letzteren gehören Befehlszähler ("program counter"), Kellerzeiger ("stack pointer") oder Adreßbasisregister. Handelt es sich bei dem Prozeß um ein Zentrum, so ist es vor allem bei Rücksetzpunkt RP2.2.2 auch sinnvoll den Baum bzw. die Kommunikationspartner zu speichern. Umgekehrt merken sich die Partner den Namen des Zentrums. Im folgenden wird zwischen den beiden Datenarten nicht weiter unterschieden, da sie beide für den Fortbestand eines Prozesses von elementarer Bedeutung sind. Die Daten müssen bei Durchlaufen eines Rücksetzpunktes alle jedesmal neu gesichert werden, da sie ständig Änderungen unterworfen sind. Ist der Anteil der geänderten Daten zur Gesamtzahl der Daten relativ gering, so erscheint es jedoch sinnvoll, veränderte Daten zu kennzeichnen und nur diese zu sichern. Dabei ist genau abzuschätzen, ob der Aufwand, die Daten zu kennzeichnen, nicht größer wird als die unveränderten Daten mitzusichern.

Zusammenfassend kann man feststellen, daß bei den Prozessen eine einmalig erstellte Sicherung der Anweisungen und jeweils eine bei jedem Rücksetzpunkt zu erstellende Datensicherung auf peripherem Speicher vorhanden sein muß, um einen konsistenten Wiederanlauf eines Prozesses zu gewährleisten. Eine Ausnahme bildet lediglich der Übergang von Phase 2.2.2 zur Phase 2.2.3 bei einer Kommunikation (siehe Bild 5-9), bei der sowohl der alte als auch der neue Datenbestand vorhanden sein muß, um noch ein Rücksetzen vor den Datenaustausch (Phase 2.2.2) zu ermöglichen (siehe

auch 5.5.).

Da auch die Treiber als Prozesse dargestellt werden, können die oben geschilderten Festlegungen auf alle selbständigen Teile, und das sind nur die Prozesse, angewendet werden.

Bei der Vorstellung des Modellbetriebssystems wurde grundsätzlich zwischen den Prozessen und den Botschaften unterschieden (siehe Kapitel 3). Letztere werden durch die "Botschaftsprozeden" realisiert. Bei den Prozeduren müssen nur die Anweisungen gesichert sein, da etwaige lokale Daten logisch gesehen zu den aufrufenden Prozessen gehören und auch bei fast allen Implementierungen im Keller der jeweiligen Task liegen.

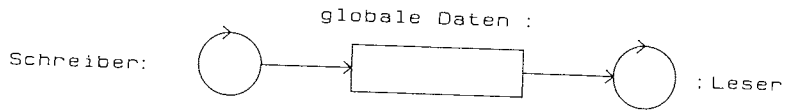
Es gibt aber auch Elemente in dem vorgestellten Prozeßsystem, die sich zunächst einmal nicht in dieses Prozeßschema integrieren lassen, wie z.B. "globale Daten" oder "Wartebereiche". Diese sollen in den folgenden Abschnitten diskutiert werden. Noch in dieses Unterkapitel gehören die "Prozeßumschalter", sowohl für Prozesse als auch für Treiber. Die Prozeßumschalter bestehen aus ihren Anweisungen und benötigen die in 3.5. eingeführte zentrale Prozeßtabelle, in der alle Prozesse und deren Zustände, erhaltene Botschaften, etc. verzeichnet sind. Der Programmcode muß wieder als Ganzes gespeichert sein. Bei der Tabelle ist es sicherlich zu umständlich, bei jeder Änderung die ganze Tabelle zu sichern, sondern es ist naheliegend, nur die Zeile (Eintrag), die zu einem Prozeß gehört, zu retten. Damit schließt das Sichern der Daten eines Prozesses auch das Sichern seines Eintrags in der zentralen Prozeßtabelle mit ein. Auf diese Art und Weise bleiben die Prozesse weiterhin unabhängig voneinander, denn ein Sichern der vollständigen Tabelle hätte eine verdeckte Abhängigkeit provoziert.

5.6.2. Datensicherung bei "globalen Daten"

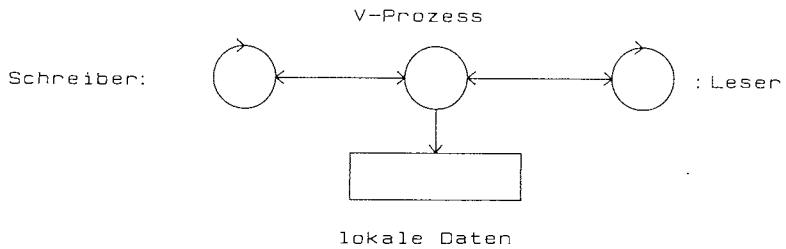
Da als Kommunikationsmittel nur mehr Botschaften erlaubt sind, können auch sogenannte "globale Daten", auf die Prozesse unabhängig voneinander zugreifen, nicht mehr in dieser Form erlaubt werden. Eine derart relativ große Einschränkung hat in Zusammenhang mit Rücksetzverfahren auch eine sehr praktische Bedeutung (siehe unten). Globale Daten bleiben zwar dem Programmierer erlaubt, aber der beliebige Zugriff, sowohl schreibend als auch lesend, wird in eine für den Programmierer nicht sichtbare "andere Form gebracht" bzw. implementiert. Für die globalen Daten wird ein (Verwaltungs-) Prozeß definiert, dessen "lokale" Daten den "globalen" Daten entsprechen. Der Zugriff auf diese nunmehr lokalen Daten wird nur diesem Prozeß erlaubt. Da der Zugriff auf globale Daten als indirekte Kommunikation zwischen Prozessen verstanden wird, liegt es nahe, diese "indirekte" Kommunikation auf eine Folge von "direkten" Kommunikationen der Prozesse mit dem Verwaltungsprozeß abzubilden.

Bild 5-12 verdeutlicht diesen Sachverhalt anhand eines Beispiels, wobei Synchronisationsaspekte außer acht bleiben, da sie nach Definition von PEARL (/DIN81/) in der Hand des Programmierers liegen. Beim Schreiber-Leser-Problem entsteht die indirekte Kommunikation dadurch, daß der schreibende Prozeß die Daten verändert und der Leser diese dann verarbeitet. Schaltet man einen Verwaltungsprozeß dazwischen, so kommuniziert der Schreiber zunächst direkt mit dem V-Prozeß (Verwaltungsprozeß). Dieser ändert die Daten und schließlich kommuniziert der Leser direkt mit dem V-Prozeß, um die geänderten Daten zu erhalten.

Der V-Prozeß wurde allerdings nicht nur aus "optischen" Gesichtspunkten eingeführt, um ein möglichst konsistentes Prozeßsystem zu erzielen, sondern hat auch für die Datensicherung praktische Bedeutung. Die globalen Daten stellen sich jetzt nur noch in Form eines Prozesses mit lokalen Daten dar, mit dem über Botschaften kommuniziert wird. Dadurch kann wieder das in Abschnitt 5.6.1. allgemein für Prozesse Gesagte gelten, so daß für globale Daten beim Setzen von Rücksetzpunkten keine weiteren Maßnahmen zu treffen sind.



a) indirekte Kommunikation über globale Daten



b) direkte Kommunikation über V-Prozeß

Bild 5-12: Ersetzen der globalen Daten durch einen V-Prozeß mit lokalen Daten

5.6.3. Datensicherung bei Wartebereichen

Nach der Spezifikationsmethode PASS (/FLEI84/) hat man zwischen den Wartebereichen (in PEARL "BUFFER") einer Prozeßgruppe und denen einzelner Prozesse zu unterscheiden. Ein Einzelprozeß kann aber als Spezialfall einer Prozeßgruppe, die nur einen einzigen Prozeß beinhaltet, aufgefaßt werden. Im folgenden wird deshalb nicht mehr zwischen den einzelnen Wartebereichtypen unterschieden.

Da es sich bei den Daten bzw. Nachrichten, die sich in einem Wartebereich befinden, in gewissem Sinne um globale Daten handelt, auf die Prozesse nur eingeschränkt Zugriff haben, läge es zunächst wieder nahe, einen Verwaltungsprozeß einzuführen. Der Zugriff stellt sich dann allgemein so dar, daß Senderprozesse Nachrichten im Wartebereich hinterlegen und Empfängerprozesse diese wieder entnehmen. Man kann damit von einem Erzeuger-Verbraucher-Problem sprechen. Das Verhalten des Verwaltungs(V)-Prozesses und seiner Kommunikationspartner soll nun nachfolgend analysiert werden.

Betrachtet man zunächst nur das Senden von Nachrichten, also das Eintragen in den Wartebereich, so nimmt ein Verwaltungsprozeß stellvertretend die Nachrichten an und nimmt die Eintragung vor. Da es sich beim Wartebereich um die lokalen Daten des V-Prozesses handelt, gilt hier wieder das Gleiche, was bei den globalen Daten (siehe 5.6.2.) erläutert wurde. Ein Sicherungspunkt müßte also nach jeder abgeschlossenen Kommunikation gesetzt werden; dabei würde auch der Inhalt des Wartebereichs gerettet. Für das Herauslesen (Empfangen) von Nachrichten aus dem Wartebereich ist die gleiche Vorgehensweise nötig.

Das bedeutet aber insgesamt, daß sämtliche Zugriffe (Schreiben und Lesen) auf den Wartebereich "sequentialisiert" werden müssen, da der V-Prozeß immer nur mit einem Partner kommunizieren kann. Aber genau das widerspricht der Definition der asynchronen Kommunikation über einen Wartebereich.

Dieser Sequentialisierungseffekt kommt bei den globalen Daten nicht zum Tragen, da auf dem Monoprozessor alle Rechenschritte sowieso nacheinander ausgeführt werden müssen (siehe auch Kapitel 1.3.). Anders sind die Verhältnisse bei einem Wartebereich, da zum einen der Wartebereich netzweit angesprochen werden kann und sich dadurch die Sequentialisierung durch das ganze Netz fortpflanzen würde. Zum anderen wird immer auf verschiedenen Wartebereichsplätzen geschrieben und gelesen, so daß ein paralleler Zugriff auf den Wartebereich notwendig ist. Aber das wird durch einen überlagerten Verwaltungsprozeß verhindert.

Verzichtet man allerdings auf die Verwendung eines zusätzlichen Verwaltungsprozesses, so bietet sich eine einfachere und effizientere Lösung an:

Bei den einzutragenden Nachrichten geht man von der Überlegung aus, daß nach vollständigem Aufbau eines Baumes durch das Zentrum zunächst eine Reservierung (siehe Regel 8 und Ergänzung zu Regel 8) der benötigten Plätze durch das Zentrum erfolgen muß. Ist dies geglückt, können die Nachrichten gesendet und in den Wartebereich eingetragen werden.

Ähnliches gilt für die zu lesenden Nachrichten aus dem Wartebereich einer (eines) Prozeßgruppe (Prozesses): Es muß zunächst eine Reservierung der benötigten Nachrichten durch die Empfänger erfolgen, und erst dann können die Nachrichten aus dem Wartebereich entnommen werden.

Aufgrund dieser Überlegungen müssen die Sicherungspunkte jeweils nach dem Eintragen und nach dem Herauslesen der Daten gesetzt werden, denn nur zu diesen Zeitpunkten werden tatsächliche Veränderungen im Wartebereich vorgenommen. Nach dem Reservieren werden keine Rücksetzpunkte benötigt, da die Reservierungen wieder zurückgezogen werden können und sie auch keine Veränderungen im Wartebereich vornehmen. Anhand von drei möglichen Fehlersituationen kann gezeigt werden, daß der gezeigte Lösungsansatz ausreicht.

Entweder entsteht ein Fehler bei einem Senderprozeß, bei einem Empfängerprozeß oder beim Wartebereich selbst. Es darf dabei vorausgesetzt werden, daß ein Fehler bei einem der Beteiligten jeweils vor einer Reservierung oder nach Fertigstellung der Sicherung keinen Effekt hat:

- Muß ein Sender zurückgesetzt werden, bevor er die Daten senden konnte, so werden auch seine Reservierungen vom betreffenden Zentrum zurückgezogen bzw. das Zentrum muß eine Neureservierung vornehmen, die die vorherige hinfällig macht. Im Gegensatz zu Prozessen muß der Wartebereich an sich nicht zurückgesetzt werden, da sich die Reservierungen nur auf einen Teil des Wartebereichs beziehen und sozusagen nur dieser Teil ein Rücksetzen erfährt. Die anderen Prozesse sind davon nicht betroffen.
- Bei einem lesenden Prozeß gilt das Gleiche wie beim Sender, nur, daß die Reservierung für das Lesen gilt.
- Muß der Wartebereich zurückgesetzt werden, so ist dies bei Nichtvorhandensein einer Reservierung unproblematisch, da keine Prozesse davon betroffen sind.

Da eine Reservierung an sich keine Datenbestandsänderung bedeutet, wird bei einer Sicherung darauf keine Rücksicht genommen. Wird also ein Wartebereich nach einem Fehler restauriert, so sind etwaige Reservierungen nicht mehr vorhanden.

Treffen Daten, für die keine Reservierungen vorliegen, von einem Senderprozeß unmittelbar nach einem Rücksetzen ein, so wird dem zentralen Prozeß ein Fehler signalisiert, da keine Reservierungen vorliegen. Das führt dazu, daß das Zentrum und alle an der Kommunikation beteiligten Prozesse ein Rücksetzen durchführen müssen. Danach kann das Zentrum sich erneut um Plätze im Wartebereich bewerben, also Reservierungen vornehmen.

Im Unterschied zu Prozessen, die immer nur an einer Kommunikation zu einem Zeitpunkt ("Wächter") beteiligt sein können, hat der Wartebereich möglicherweise mehrere Reservierungen von verschiedenen Zentren parallel vorliegen. Das hat bei einem Rücksetzen des Wartebereichs größere Auswirkungen, da dadurch nicht nur mehrere Zentren, sondern damit auch eine größere Anzahl an Prozessen betroffen sind.

Für Reservierungen von Empfängerprozessen gilt beim Rücksetzen eines Wartebereichs das Gleiche wie das eben Geschilderte.

5.7. Reaktion nach Bemerken eines Ausfalls

Mit den in Abschnitt 5.2. eingeführten Zeitüberwachungsmaßnahmen auf der Betriebssystemebene kann der Botschaftenaustausch überwacht werden. Das Zentrum sendet "Frage"-Botschaften aus und erwartet dafür Antworten von den Partnerprozessen. Dieser Vorgang kann kontrolliert werden, und für den Fall, daß ein Partner in der gewünschten Zeit nicht antwortet, muß das Zentrum geeignete Maßnahmen zur Fehlerbehebung ergreifen.

Bei diesen Aktionen wird, wie auch in Kapitel 7.1. erläutert, vorausgesetzt, daß das Transportmedium, das die Botschaftenübermittlung durchführt, über Mittel zum "Routing" verfügt.

Die Maßnahmen umfassen zwei Schritte:

- 1) Das Zentrum muß zunächst feststellen, ob überhaupt noch eine Verbindung zu dem Prozeß bzw. zu dem Prozessor, auf dem dieser läuft, besteht. Entweder stehen ihm dafür Dienste des Transportmediums zur Verfügung oder er versucht z.B. Kontakt zu einem anderen Prozeß auf dem gleichen Rechner aufzunehmen. Wird festgestellt, daß die Verbindung unterbrochen ist, bleibt nur die Möglichkeit, für den Benutzer oder Sytembediener eine Fehlermeldung zu erzeugen.
- 2) Hat das Zentrum festgestellt, daß noch eine Verbindung zum entsprechenden Rechner besteht, muß es versuchen, den fehlerhaften Prozeß oder Prozessor wieder in einen definierten Zustand zu bringen. Dabei sind zwei Fälle zu unterscheiden:
 - Es muß festgestellt werden, ob der Leitungsanschluß noch funktioniert. Das kann durch Spezialnachrichten bewerkstelligt werden, in der Art von Prüfbotschaften, auf die der zu untersuchende Leitungstreiber geeignet zu reagieren hat (siehe 5.2.). Ist der Leitungsanschluß defekt, muß der ganze Rechner mit allen seinen Prozessen zurückgesetzt werden. Das ist deshalb notwendig, weil nach Ausfall des Anschlusses nicht klar ist, in welchem Zustand sich die Prozesse befinden.

- Ist der Leitungsanschluß noch in Ordnung, so handelt es sich lediglich um einen Fehler beim Partnerprozeß. Dieser muß mit den zuletzt sichergestellten Rücksetzdaten, einschließlich der Eintragung in der zentralen Tabelle des Prozeßumschalters, neu geladen und fortgesetzt werden.

Zusätzlich zu den Maßnahmen, um den zerstörten Prozeß wieder lauffähig zu machen, müssen alle weiteren an der Kommunikation beteiligten Prozesse, die keinen Fehler aufweisen, aufgefordert werden, bei sich selbst ein Rücksetzen auf den letzten definierten Zustand durchzuführen. Sind sie damit fertig, wird das Zentrum darüber informiert. Das Zentrum muß für sich selbst auch ein Rücksetzen durchführen.

Die eben beschriebene Vorgehensweise gilt auch für den Fall, daß sich der defekte Prozeß auf dem gleichen Prozessor wie das Zentrum befindet, das den Fehler bemerkt hat. Die Probleme mit dem Leitungsanschluß treten dabei nicht auf.

Eine gänzlich andere Problematik muß im Zusammenhang mit der Prozeßdatenverarbeitung beachtet werden, wenn es also darum geht, Daten eines technischen Prozesses zu sichern. Muß ein Benutzerprozeß zurückgesetzt werden, verstreicht "relativ viel" Zeit, da er von einem Peripheriespeicher neu geladen werden muß. Während dieser Zeit gehen Daten eines technischen Prozesses verloren, da eine Produktionsanlage oder sonstige Anlagen dabei nicht zum Stillstand kommen. Für die Problemstellung der **Datenerfassung** bieten sich drei verschiedene Lösungsansätze an:

- 1) Der Benutzer kümmert sich selbst um das Retten der Prozeßdaten, z.B. durch Hardware-Module. Solche Module sollten in der Lage sein, Prozeßdaten aufzunehmen und zu speichern. Sie geben die Daten nur dann weiter, wenn der Rechner zur Aufnahme bereit ist. Dabei muß der Speicher so groß sein, daß er alle Daten aufnehmen kann, die maximal während einer Annahmepause des Rechners anfallen.
- 2) Es ist nicht notwendig, die verlorengehenden Daten zu erhalten, z.B. wenn immer nur die neuesten Daten von Interesse sind.
- 3) Da auch die Prozeß-E/A von einem Verwaltungsprozeß überwacht wird (die Daten werden vom Treiberprozeß entgegengenommen und per Botschaft zum verarbeitenden Benutzerprozeß gesendet), könnte dieser die Daten, die vom technischen Prozeß kommen, aufnehmen und dann zum inzwischen reparierten Prozeß weitergeben. Das entspricht dem 1. Lösungsansatz mit dem Unterschied, daß es sich hier um eine Software-Lösung handelt.

Bei dieser Lösung sind zwei Alternativen zu unterscheiden:

- Der Benutzerprozeß konnte die Empfangsanweisung noch vor seiner Zerstörung absetzen: der E/A-Verwaltungsprozeß ist dann sowieso in der Lage, die Daten zwischenspeichern. Bei ihm liegt der Eingabeauftrag vor, und er muß nur darauf warten, daß er seine Daten an den Benutzerprozeß senden kann.
- Es werden Daten empfangen, obwohl kein Eingabeauftrag vorliegt. Der Verwaltungsprozeß muß die Daten zwischenspeichern, bis ein Benutzerprozeß sie abholt. Das entspräche dem sogenannten "Type ahead" bei Terminals.

Dieser Ansatz hat insgesamt den Nachteil, daß bei einem Gesamtausfall des Rechners keine Möglichkeit besteht, die Daten in irgendeiner Weise zu sichern, da der Verwaltungsprozeß sich mit auf dem Prozessor befindet.

Aus allen Ansätzen wird deutlich, daß bei einem Rücksetzen von Prozessen Sorge dafür getragen werden muß, daß der technische Prozeß ohne Probleme weiterlaufen kann. Bei Prozeßsteuerungen, die heutzutage im Gebrauch sind, wird als Lösung bei Ausfällen ein "Hot-stand-by" oder zumindest ein "Cold-stand-by" genutzt. Würde man allerdings die 1. vorgeschlagene Lösung, also einen Hardware-Modul, evtl. mit eigener gesicherter Stromversorgung, wählen, so käme man sicherlich auf eine billige Lösung.

Anders verhält es sich bei der **Steuerung** technischer Prozesse. Fällt hier ein Steuerungs-, Verwaltungs- oder Treiberprozeß aus, so gibt es nur die Möglichkeit, daß ein Ersatzprozeß solange einspringt, bis der betroffene Prozeß repariert ist. Erkennt also ein Zentrum, daß ein Prozeß defekt ist und ein Rücksetzen initiiert werden muß, so hat er den Stellvertreter sofort zu starten, wobei dieser über die aktuellen Daten oder zumindest über eine Kopie der letzten Wiederaufsetzdaten im Hauptspeicher verfügen muß.

5.8. Die Parallelen zum "Three Phase Commit"-Protokoll

In Kapitel 2.4. wurde das "Three Phase Commit"-Protokoll (TPCP) nach Bernstein u.a. (/BEHG87/) bzw. Ceri und Pelagatti (/CERI84/) kurz vorgestellt, das vor allem in Datenbanksystemen Anwendung findet, um den gemeinsamen Abschluß einer Transaktion zu ermöglichen. In diesem Abschnitt soll ein Vergleich zwischen dem TPCP und dem Baumverfahren mit Setzen von Sicherungspunkten, kurz Baumverfahren genannt, gezogen werden.

Im Baumverfahren hat das Zentrum die Aufgabe mit Hilfe der Erstellung eines Baumes die Kommunikation zu allen Partnerprozessen aufzubauen. Ist dies gelungen, kann das Zentrum die Reservierungen aussenden, die Bestätigungen einsammeln und den Nachrichtenaustausch mit Setzen der Sicherungspunkte zu initiieren. In der Terminologie des TPCP bedeutet dies, daß eine Transaktion beendet werden soll und der "coordinator" daraufhin ein "prepare" an alle an der Transaktion Beteiligten aussendet. Letztere bestätigen durch ein "ready" ihre Abschlußbereitschaft und werden durch das "pre-commit" zum Beenden aufgefordert. Der Unterschied zwischen TPCP und Baumverfahren liegt darin, daß das Zentrum (Koordinator) erst durch den Baumaufbau feststellen muß, wer die Partnerprozesse sind.

Kommt beim Baumverfahren der Nachrichtenaustausch zustande, d.h. alle Prozesse, einschließlich des Zentrums, melden einen erfolgreichen Abschluß von Nachrichtenaustausch und Setzen der Sicherungspunkte, so kann das Zentrum an alle Beteiligten die "Fortsetz"-Nachricht senden. Das entspricht im TPCP dem Senden der "ack" durch die Partner und dem abschließenden Aussenden des "commit" durch den Koordinator mit der die Transaktion verlassen wird.

Sowohl im TPCP als auch im Baumverfahren sind die Aktionen zwischen dem "prepare" ("Reservierung der Partner") und dem "commit" ("Fortsetz"-Nachricht) als atomar zu betrachten, d.h. entweder wird die Aktionsfolge erfolgreich und vollständig durchgeführt oder es wird ohne Auswirkungen auf das Gesamtsystem auf den Ausgangspunkt zurückgesetzt (s.u.).

Im Fehlerfall, wenn sich einer der Prozesse nicht mehr meldet oder einen nicht erfolgreichen Abschluß des Nachrichtenaustausches bzw. der Transaktion zurücksendet, veranlassen sowohl das Baumverfahren als auch das TPCP ein Zurücksetzen aller beteiligten Prozesse.

Trotz offensichtlicher Gemeinsamkeiten zwischen dem Baumverfahren und dem "Three Phase Commit"-Protokoll in ihrer Vorgehensweise und Abfolge von Botschaften, gibt es Unterschiede. Diese liegen im Zweck der beiden Verfahren begründet. Beim TPCP steht die gemeinsame Änderung von Datenbeständen im Vordergrund, wobei es vornehmlich um die Konsistenz der Daten und nicht um den Zustand der an der Transaktion beteiligten Prozesse geht. Gibt es einen Fehler bei einem Prozeß, so ist die allerdringlichste Aktion das "Konsistenthalten" der Daten. Den zerstörten Prozeß zu restaurieren ist zweitrangig.

Ganz im Gegensatz dazu ist es das wichtigste Ziel beim Baumverfahren, einen zerstörten Prozeß wieder zum Laufen zu bringen. Im Einklang damit steht die Forderung, daß alle an der Kommunikation beteiligten Prozesse nicht "auseinanderlaufen", also die Synchronität der Kommunikation bzw. der Prozesse erhalten bleibt. Das genau ist auch die Forderung, die durch das Sprachkonstrukt "Guarded Region" erhoben wird: das **gleichzeitige** Ausführen aller Kommunikationen eines "Guards". Gemeinsame Daten wie beim TPCP fehlen gänzlich.

Zusammenfassend heißt das, daß TPCP und Baumverfahren zwar die gleiche Abfolge von Botschaften und Zustandsübergängen (Protokoll) verwenden, aber zum einen, um gemeinsame Daten konsistent zu halten und zum anderen, um die Synchronität der beteiligten Prozesse zu gewährleisten.

5.9. Aufwand für Sicherungs- und Rücksetzmaßnahmen

Mit den in diesem Kapitel eingeführten Maßnahmen zum Setzen von Sicherungspunkten und dem damit verbundenen Sichern von Daten ist ein gewisser Mehraufwand ("Overhead") verbunden. Es stellt sich die Frage, wie dieser zusätzliche Aufwand gegenüber den Kosten, die bei Verzicht auf derartige Maßnahmen entstehen können, zu bewerten ist und, ob in Bezug auf diesen Gesichtspunkt das erstellte Verfahren gerechtfertigt ist.

Bei einer Aufwandsabschätzung von Rücksetzmaßnahmen sind nach Weber (/WEBE83/) zwei konträre Einflußgrößen gegeneinander abzuwägen:

- 1) Zum einen sind die Aufwendungen für das Erstellen der Sicherungspunkte zu berücksichtigen. Darunter ist vor allem das Abspeichern der (geänderten) Daten auf einen sicheren Datenträger zu verstehen. Diese Größe soll mit "S" (Sichern) bezeichnet werden.
- 2) Zum anderen ist der Verlust abzuschätzen, der entsteht, wenn ein Prozeß und damit auch seine Daten zerstört werden und alle seine bisherigen Aktionen damit hinfällig werden. Diese Größe soll mit "V" (Verlust) bezeichnet werden.

Um einen Vergleich dieser beiden Größen durchzuführen, muß man sich zunächst darüber im klaren sein, ob man überhaupt auf irgendwelche Maßnahmen zurückgreifen möchte. Verzichtet man auf Vorkehrungen, so bleibt der betreffende Prozeß zerstört und ist ab diesem Zeitpunkt nicht mehr ansprechbar. Prozesse, die mit ihm ohne eine Benutzerzeitüberwachung kommunizieren wollen, bleiben deshalb blockiert.

Entscheidet man sich dagegen für mindestens die Möglichkeit, bei einem Fehler die Prozesse neu zu starten, so müssen ihre Programme und Initialdaten gesichert sein. Zu diesem erhöhten Aufwand an peripherem Speicher kommt noch der Zeitaufwand, den ein Partnerprozeß verbraucht, um den zerstörten Prozeß neu zu laden und alle an der Kommunikation beteiligten Prozesse neu starten. Allerdings ist zu bedenken, daß durch den Neustart eines Prozesses dieser auch mit seinen Kommunikationen von neuem beginnt, während andere, nicht an der Kommunikation beteiligte Prozesse weiterhin im Programm fortfahren.

Ist man bereit, außer dem Aufwand für das Speichern von Programm und Initialdaten auch noch den Aufwand für Rücksetzpunkte auf sich zu nehmen, ist der Wiederanlauf

eines zerstörten Prozesses zu einem späteren Zeitpunkt als dem Programmstart möglich. In die Abschätzung der Aufwandsgröße S gehen die beiden Komponenten Speicherplatz und Zeit ein. Speicherplatz wird zusätzlich für die zu sichernden Daten benötigt, die aus den Verwaltungsdaten und den vom Benutzer definierten Programmdateien bestehen. Dieser Speicherplatz muß verdoppelt werden, da das Baumverfahren für einen gewissen Zeitraum das Halten sowohl der alten als auch der neuen Sicherungsdaten verlangt (siehe Kapitel 5.5.). Der Faktor Zeit spielt deshalb eine Rolle, weil ein bestimmter Zeitbedarf zum Sichern der Daten benötigt wird, der einerseits vom Umfang der Daten, andererseits von den Geschwindigkeiten von Primär- und Sekundärpeicher abhängt.

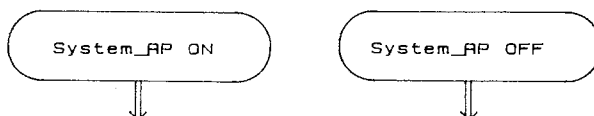
Weit schwieriger abzuschätzen ist der Verlust V , denn dieser Wert ist nur dann quantifizierbar, wenn die Aufgabe bzw. die Funktion des Prozesses bekannt ist. Allgemeingültigkeit dürfte dabei die Aussage haben, daß reine Protokollprozesse oder redundante Prozesse einen wesentlich kleineren Wert als überwachende oder zentrale nicht redundante Steuerungsprozesse haben.

Zusammenfassend heißt das: Erlaubt man eine für jeden Prozeß separate Entscheidung, ob Sicherungspunkte gesetzt werden sollen oder nicht, so muß bei der Entscheidung für einen Rücksetzpunkt " $S < V$ " gelten.

5.10. Sicherungsmaßnahmen durch den Benutzer

Bei der bisherigen Vorgehensweise erstellen das Betriebssystem bzw. die Botschafts-prozeduren "automatisch" die Sicherungspunkte. Bei Bemerken eines Ausfalls eines Prozesses wird ebenfalls selbständig ein Rücksetzen des betreffenden Prozesses und aller an der Kommunikation beteiligten Prozesse versucht. Will man eine größere Transparenz für den Benutzer oder Programmierer erreichen, können Kommandos zum Ein- und Ausschalten dieser Betriebssystemfähigkeiten zur Verfügung gestellt werden.

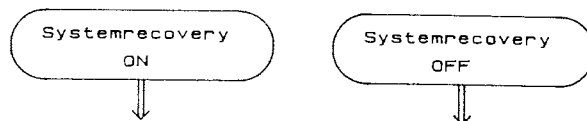
in PASS:



in Programmiersprache: SYSTEMRPON

SYSTEMRPOFF

in PASS:



in Programmiersprache: SYSTEMRECOVERYON

SYSTEMRECOVERYOFF

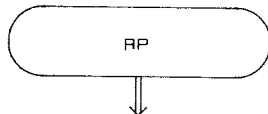
Bild 5-13: Ein/Ausschalten von Betriebssystem-Rücksetzpunkten und -Rücksetzen

Die in Bild 5-13 mit Hilfe von PASS und den Sprachkonstrukten eingeführten Operationen sollen diese Möglichkeiten darstellen. Die Anweisungen gelten jeweils nur für den Prozeß, in dem sie angegeben sind. Würde man erlauben, daß diese Anweisungen für mehrere Prozesse gelten, würde die Übersichtlichkeit sehr schnell darunter leiden, denn dieses Ein/Ausschalten ist vom dynamischen Ablauf des Prozesses abhängig. Man könnte damit nie sicher sagen, ob bei den angesprochenen Prozessen die Rücksetzpunkte eingeschaltet sind oder nicht.

Ist das Setzen von Rücksetzpunkten ausgeschaltet, muß im Falle eines Wiederaufsetzens auf die zuletzt gesicherten Daten oder, falls keine vorhanden sind, auf die Initialdaten zurückgegriffen (Neustart des Prozesses) werden. Ist das Rücksetzen ausgeschaltet, verbleibt ein fehlerhafter Prozeß im zerstörten Zustand. In dieser Situation ist es sinnvoll, daß das Zentrum eine Fehlermeldung, z.B. "Prozeß XYZ defekt" oder "Verbindung zum Prozeß XYZ unterbrochen", erzeugt.

Zusätzlich zu den erwähnten Steuerkommandos, die dem Programmierer ermöglicht werden, kann auch das explizite Setzen von Sicherungspunkten zur Verfügung gestellt werden (siehe "RP"-Kommando, Bild 5-14). Dies ist dann sinnvoll, wenn ein Prozeß sehr lange sequentielle Programmphasen, also Programmphasen ohne Kommunikation durchläuft und es zu aufwendig wäre, nach dem Wiederaufsetzen diese Programmsequenzen sämtlich zu wiederholen.

in PASS:



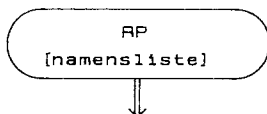
in Programmiersprache: RECOVERYPPOINT

Bild 5-14: Setzen von Rücksetzpunkten durch den Benutzer

Eine Erweiterung wäre noch dahingehend vorstellbar, daß eine Liste von Prozeßnamen beim "RP"-Kommando angegeben wird. Das würde bedeuten, daß nicht nur der Prozeß mit der "RP"-Anweisung selbst, sondern auch die in der Liste angegebenen Prozesse einen Sicherungspunkt setzen. Problematisch ist dabei die Darstellung dieser Anweisung, die ja einen gemeinsamen Sicherungspunkt der beteiligten Prozesse erzeugen will und damit auch Kommunikationen zwischen diesen Prozessen voraussetzt. Eine Darstellung in PASS in der Form eines Kommunikationsknotens scheidet aus, da dies auch Kommunikationsknoten bei den Partnerprozessen voraussetzen würde. Das ließe sich durch Kommunikationen über TRANSMIT und RECEIVE (Sende-/Empfangskanten) leichter erreichen.

Bleibt also nur die in Bild 5-15 gewählte Darstellungsform als "interne Operation". Die in der Namensliste aufgeführten Prozeßnamen können alle im Gesamtsystem vorhandenen Prozesse, darunter auch Systemprozesse, wie z.B. Geräteverwaltungsprozesse, sein.

in PASS:



in Programmiersprache: RECOVERYPOINT ([namensliste])

mit namensliste ::= { processname } *

* ::= 0, 1, 2, ...-faches Vorkommen

Bild 5-15: Setzen von Rücksetzpunkten mit Namensliste

Die Darstellung als "interne Operation" ist allerdings auch fraglich, da implizit durch das Betriebssystem Kommunikationen mit den Partnerprozessen durchgeführt werden, um den gemeinsamen Rücksetzpunkt zu erstellen. Da diese Aktionen aus Benutzersicht nicht sichtbar sind, kann es zu Überlagerungen beim Setzen der Sicherungspunkte durch die Anweisung "RECOVERYPOINT(...)" und durch den Nachrichtenaustausch über TRANSMIT und RECEIVE kommen.

Aus diesen Gründen erscheint es sinnvoller, ein Benutzer programmiert einen Nachrichtenaustausch über TRANSMIT/RECEIVE zwischen einem Steuerungsprozeß und denjenigen Prozessen, von denen er einen Sicherungspunkt wünscht, unter impliziter Nutzung der Systemsicherungspunkte.

6. Erweiterungen des verteilten Systems für seine dynamische Konfigurierbarkeit und seine Einbettung in offene Systeme

Mit dem in den vorangegangenen Kapiteln erzielten Gesamtkonzept für ein verteiltes System ist der Endpunkt für mögliche Entwicklungen noch nicht erreicht. Besonders zwei Problemfelder drängen sich noch auf: Zum einen sind Überlegungen anzustellen, wie der Aufbau eines verteilten Systems zur Laufzeit dynamisch geändert werden kann ("Rekonfigurierbarkeit"). Dies wird im 1. Abschnitt dieses Kapitels näher behandelt. Zum anderen existieren innerhalb der Standardisierungsorganisation ISO ("International Organization for Standardization") Bestrebungen, eine Norm für "verteiltes Rechnen" zu entwickeln. Deshalb wird im 2. Abschnitt auf die Einbettungsproblematik des in dieser Arbeit vorgestellten Systems in das IOS-Referenzmodell eingegangen und es werden Parallelen zur noch nicht verabschiedeten ISO-Norm "Transaction Processing" gezogen.

6.1. Rekonfigurieren eines verteilten Systems

Es wurde im letzten Kapitel auf die Möglichkeiten eingegangen, einen zerstörten Prozeß wieder herzustellen. Die Maßnahmen, einen Prozeß wieder in einen definierten Zustand zu bringen, sind aber überhaupt nur dann sinnvoll, wenn der betreffende Prozessor, auf dem dieser Prozeß läuft, sich in einem funktionstüchtigen Zustand befindet. Grundvoraussetzung für ein Rücksetzen ist also immer der technisch einwandfreie Zustand des Rechners. In diesem Kapitel sollen deshalb Hinweise gegeben werden, wie verfahren werden könnte, wenn ein Prozessor für längere Zeit ausfällt und deshalb nach einer Ersatzlösung (Änderung der Konfiguration) gesucht werden muß. Im ersten Abschnitt werden einige Definitionen gegeben und im zweiten Abschnitt wird auf mögliche Maßnahmen eingegangen.

6.1.1. Definitionen

Zur Begriffsklärung folgen zunächst einige Definitionen.

Definition: Unter "statischer Konfiguration" versteht man die Zusammenstellung eines ablauffähigen Prozeßsystems auf einem selbständigen Prozessor, bestehend aus Zentraleinheit, Speicher und notwendiger Peripherie. Der Begriff gilt ebenso für eine Menge derartiger Prozeßsysteme, wenn die Prozessoren über ein Netz miteinander verbunden sind (lose gekoppelt).

Definition: Unter den "Mindestanforderungen" eines Prozesses werden die Anforderungen verstanden, die ein Prozeß an die vorhandene Hardware und Software eines Prozessors mindestens stellt, um regulär ablaufen zu können. Die Hardware und Software eines Prozessors werden auch als dessen "Eigenschaften" bezeichnet. Die Eigenschaften eines Prozessors sind z.B. die Geschwindigkeiten von Zentraleinheit und Speicher, das Vorhandensein von Peripherieanschlüssen oder von bestimmten Programmen.

Letztere Definition bedeutet, daß beim zunächst statischen Konfigurieren darauf zu achten ist, daß jeder Prozeß auf einem Rechner (Prozessor) installiert wird, der seinen Anforderungen entspricht.

Wenn wir voraussetzen, daß wir über ein Netz von Prozessoren, einschließlich der Peripherieeinheiten, verfügen, so gibt es Rechner, auf denen ein Prozeß nicht geladen werden kann, da die Mindestanforderungen nicht erfüllt sind, und es gibt Rechner, die den Bedingungen entsprechen. Daraus folgt die nächste Definition:

Definition: Aufgrund der Mindestanforderungen eines Prozesses lassen sich die Prozessoren eines Netzwerks in drei verschiedene "Klassen" unterteilen:

- Die "Standardklasse" ist die Menge der Prozessoren, die genau die Eigenschaften besitzen, die der Prozeß für seinen Ablauf benötigt. Die Prozessoreigenschaften entsprechen also genau den Mindestanforderungen.
- Die "Oberklasse" ist die Menge der Prozessoren, die mächtigere Eigenschaften besitzen, als für den Prozeß notwendig sind. Die Prozessoreigenschaften übersteigen also die Mindestanforderungen.
- Die "Unterklasse" ist die Menge der Prozessoren, denen Eigenschaften fehlen, die für diesen Prozeß zu dessen Ablauf notwendig wären. Die Prozessoreigenschaften erfüllen also die Mindestanforderungen nicht.

Definition: Unter "dynamischem Konfigurieren" ("Um-" oder "Rekonfigurieren") eines Prozesses wird das erneute Laden und Starten des Prozesses auf einem Rechner verstanden, der nicht dem Prozessor entspricht, auf dem der Prozeß ursprünglich nach der statischen Konfiguration installiert war.

Im nächsten Abschnitt sollen die Probleme eines dynamischen Konfigurierens näher beleuchtet werden. Aus den Definitionen wird aber bereits deutlich, daß aufgrund der Forderungen, daß die Eigenschaften eines Prozessors den Mindestanforderungen eines Prozesses genügen müssen, einem Umkonfigurieren enge Grenzen gesetzt sind.

6.1.2. Dynamisches Konfigurieren von Prozessen

Bei den oben gegebenen Definitionen wurde festgestellt, daß Prozessoren über gewisse Eigenschaften verfügen. Betrachtet man verteilte Systeme zur Steuerung und Überwachung technischer Prozesse, so werden die Eigenschaften der einzelnen Prozessoren sehr unterschiedlich und umfangreich. Es müssen z.B. die notwendigen Anschlüsse zu bestimmten technischen Prozessen für bestimmte Prozessoren verfügbar sein. Das kann dazu führen, daß zahlreiche Rechner eines Prozeßsystems in die Unterklasse eines Prozesses geraten, da sie zwar in fast allen Punkten den Mindestanforderungen genügen, aber nicht über den nötigen Anschluß zu einem bestimmten technischen Prozeß verfügen. Aus diesem Beispiel und aus der Definition der Klassen wird ersichtlich, daß für das Umkonfigurieren eines Prozesses nur die Standard- und Oberklassen in Frage kommen.

Versucht man, Algorithmen zum dynamischen Konfigurieren von Prozessen zu implementieren, so sind zwei Voraussetzungen zu erfüllen:

- Der Konfigurationsalgorithmus muß über alle Eigenschaften der Prozessoren Bescheid wissen.
- Ebenso müssen die Mindestanforderungen der Prozesse eines Prozeßsystems bekannt sein.

Aus den beiden Informationsmengen können jeweils für jeden Prozeß die Klassen definiert werden. Ein Konfigurationsalgorithmus muß dann bei Ausfall eines Rechners, den oder die Prozesse auf Rechnern der Standard- oder Oberklasse laden und starten. Dabei muß er nicht nur über die Programme und Initialdaten der Prozesse, sondern auch über die letzten Sicherungsdaten verfügen.

Bislang wurden Prozesse nur für sich selbst betrachtet. Verfügen aber Prozesse auf einem Prozessor über gemeinsame Objekte oder, wie in der Programmiersprache PEARL, über gemeinsame Synchronisationsmechanismen (Semaphore), so darf der einzelne Prozeß nicht mehr für sich allein umkonfiguriert werden. In der Spezifikationsmethode PASS gibt es entsprechend für Prozesse mit gemeinsamen Daten die Strukturierungsmittel Prozeß-Bündel und -Gruppe. Deshalb müssen die oben aufgeführten Definitionen und Aussagen auf diese beiden Begriffe übertragen werden. Das bedeutet, daß nur mehr Prozeß-Bündel und -Gruppen rekonfiguriert werden können.

Zum Schluß seien noch zwei alternative Ansätze erwähnt, die etwas andere Vorgehensweisen verfolgen:

- Zum einen wäre ein dynamisches Konfigurieren per Programm vorstellbar, d.h. dem Programmierer werden Anweisungen zur Verfügung gestellt, mit denen er einen Prozeß isolieren, löschen und durch einen neuen Prozeß ersetzen kann. Dieser Vorschlag wird in /IIDS84/ gemacht, wobei allerdings die oben genannten Bedingungen, wie z.B. das Einhalten der Mindestanforderungen eines Prozesses, einfach vorausgesetzt werden.
- Zum anderen wäre das Konfigurieren über eine Benutzerschnittstelle des verteilten Betriebssystems denkbar: Ein Benutzer kann mit Hilfe von Anweisungen, die er über die Dialogschnittstelle eingibt, Prozesse isolieren, löschen, neu laden, etc. Diese Vorgehensweise setzt allerdings eine ziemliche Mächtigkeit des Betriebssystems voraus, da vor allem Betriebssystemprozesse existieren müssen, die Prozesse dynamisch einrichten und starten können.

Zusammenfassend kann festgestellt werden, daß dem Umkonfigurieren von Prozessen bzw. Prozeßsystemen in verteilten Systemen zur Steuerung technischer Prozesse sehr enge Grenzen gesetzt sind. Der Hauptgrund für die Schwierigkeiten liegt in den Mindestanforderungen der Prozesse, aber auch zeitliche Aspekte, z.B. wie schnell das Umkonfigurieren durchgeführt werden kann, spielen eine Rolle. Voraussetzung insgesamt ist ein genügend mächtiger Botschaftenalgorithmus. Inwieweit das Baumverfahren dazu ausreicht, müßte noch untersucht werden.

6.2. Einbettung des verteilten Systems in das ISO-7-Schichten-Modell

In der Einführung zu Kapitel 4 wurde zwischen der logischen und der physikalischen Übermittlung von Botschaften unterschieden. Dabei wurde festgestellt, daß die logische Übertragung durch ein genau definiertes Protokoll zwischen den an der Kommunikation beteiligten Prozessen bewerkstelligt wird. Ein Beispiel für ein derartiges Protokoll wurde mit dem Baumverfahren vorgestellt. Von der physikalischen Übermittlung wurde in Absatz 3.3.4. ("Implementiertes System") angenommen, daß Prozesse in der Lage sind, direkt ihre Botschaften an das Übertragungsmedium abzugeben, wie es auch Bild 6-1 zeigt.

Dieser Sachverhalt soll im folgenden Abschnitt näher untersucht werden. Die Überlegungen müssen sich aber auf relativ einfache Übertragungsmedien beschränken. Um diese Einschränkungen zu beseitigen, soll mit Hilfe des in Abschnitt 2 eingeführten ISO/OSI-Schichtenmodells eine Erweiterung für offene Systeme durchgeführt werden. Abschließend wird noch auf die Entwicklungen in der ISO/OSI-Normung bezüglich der Schicht 7 eingegangen und ein Vergleich mit dem hier vorgestellten Modell gezogen.

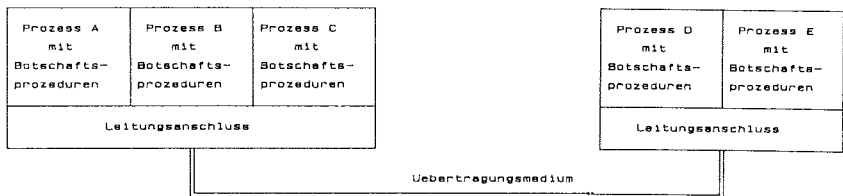


Bild 6-1: Direkte Botschaftenübermittlung durch Prozesse

6.2.1. Direkte Botschaftenübermittlung durch die Benutzerprozesse

Anwenderprozesse wickeln als Protokoll das Baumverfahren ab, um damit miteinander kommunizieren zu können. Setzt man Bild 6-1 voraus, so erstellt ein Prozeß eine Botschaft und ruft eine (Leitungs-) Prozedur auf, durch die er die Botschaft zu seinem Partner (-Prozeß) senden kann. Letzterer befindet sich auf einem Rechner, dessen Leitungstreiber die Botschaft entgegennimmt und beim Adressaten hinterlegt. Diese Vorgehensweise setzt voraus, daß alle Prozesse bzw. Prozessoren direkt miteinander verbunden sind, wie Bild 6-2 zeigt.

Setzt man dagegen die Verknüpfung der Prozessoren, wie in Bild 6-3 dargestellt, voraus, so muß die Leitungs-E/A oder der Leitungstreiber fähig sein, Botschaften weiterzuvermitteln. Dieser Mechanismus wird als "Routing" ("Weitervermitteln", siehe Kapitel 5.2.) bezeichnet. Für den Fall eines total vermaschten Netzes wie in Bild 6-2 wird auch dann ein Routing notwendig, wenn eine der Leitungen ausfällt und damit die Verbindung zwischen zwei Prozessen abreißt. Um einen möglichst konsistenten Zustand des Gesamtsystems zu erzielen, wird deswegen ein Routing als gegeben vorausgesetzt.

Bemerkung: Ein Routing muß vom Baumverfahren vorausgesetzt werden, da das Zentrum alle Prozesse, die sich in der gleichen Kommunikation befinden, erreichen können muß (siehe Kapitel 3), ohne aber über direkte Verbindungen zu jedem Prozeß zu verfügen.

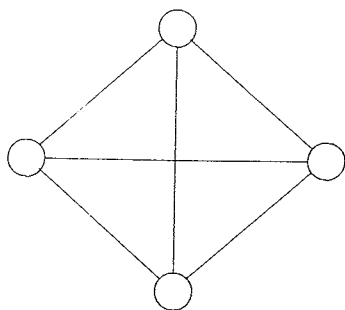


Bild 6-2: Alle vorhandenen Prozesse sind direkt miteinander verbunden

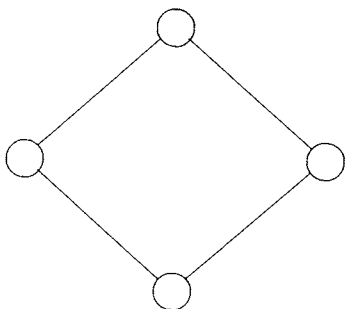


Bild 6-3: Alle vorhandenen Prozesse sind nicht direkt miteinander verbunden

6.2.2. Einbettung der Prozeß-Prozeß-Kommunikation in das ISO/OSI-Referenzmodell

Die Versuche, unterschiedliche Rechner mit unterschiedlichen Betriebssystemen miteinander zu verbinden, um damit auch entfernte Rechner den Benutzern in Zugriff zu stellen, hat zu verschiedenen Normungsbemühungen geführt. Die Normvorschläge der ISO, die sich im OSI-Modell ("Open Systems Interconnection") niederschlagen, haben sich dabei weitestgehend durchgesetzt. Das Modell läßt sich mit der 7-Schichten-Architektur von Bild 6-4 darstellen, das u.a. in /KEBR81/ sehr ausführlich erläutert ist. Es soll hier auf die Charakteristika der einzelnen Schichten nur soweit eingegangen werden, wie es erforderlich ist, das Protokoll zur Prozeßkommunikation einzuordnen.

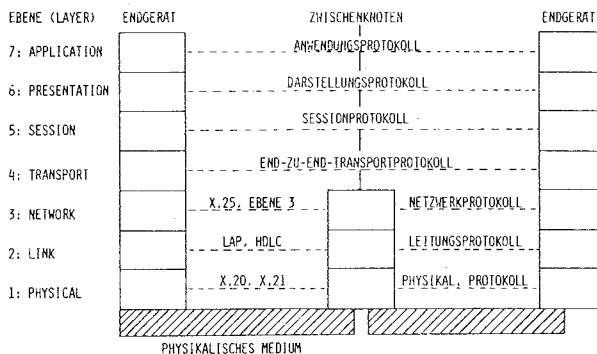


Bild 6-4: Das ISO/OSI-7-Schichten-Architekturmodell, /KEBR81/

Während in der Schicht 1 ("Physical Layer") nur der rein technische Anschluß definiert wird, bietet bereits die Schicht 2 ("Link Layer") die Möglichkeit, Datenpakete zwischen zwei direkt miteinander verbundenen Geräten (Rechnern) auszutauschen.

Im vorhergehenden Abschnitt hatten wir in Bild 6-2 die Fähigkeit der Kommunikation von Prozessen über direkte Verbindungen eingeführt. Das bedeutet, daß die Prozeßkommunikation mindestens die Ebene 2 des ISO-Referenzmodells voraussetzt.

Einen Schritt weiter geht die 3. Schicht ("Network Layer"), mit deren Hilfe zwei Rechner (Prozesse) miteinander in Verbindung treten können, ohne dabei direkt miteinander verbunden zu sein. Es wird damit eine sogenannte **Punkt-zu-Punkt-Ver-**

bindung aufgebaut, mit 0, 1, 2 und mehr Knoten (-Rechnern) dazwischen, so daß als eine wesentliche Eigenschaft dieser Schicht das Routing zu nennen ist. Als ein bekanntes Beispiel für eine Netzwerk-Norm gilt "X.25", wie sie auch im Paketvermittlungsnetz DATEX-P der Deutschen Bundespost realisiert ist.

Legen wir für die Prozeß-Prozeß-Kommunikation das Bild 6-3 zu Grunde, so hat das zur Folge, daß wir mindestens die Ebene 3 des Referenzmodells voraussetzen müssen.

Die Einführung der Transportschicht (Schicht 4, "Transport Layer") dient dazu, von der Netztopologie, wie sie z.B. noch im Aufbau der DATEX-P-Adressen sichtbar ist, zu abstrahieren. Mit ihr können sogenannte Ende-zu-Ende-Verbindungen auf- und abgebaut werden, um einen völlig transparenten Datenverkehr zwischen zwei Endgeräten, z.B. Prozessoren (Prozesse), zu erzielen.

In /KEBR81/ werden die Schichten 1 bis 4 im "Transportsystem" und die Schichten 5 bis 7 im "Anwendersystem" zusammengefaßt. Durch diese Unterscheidung wird deutlich, daß die Transportschicht letztendlich die Schicht ist, die den Datentransport bewerkstelligt, während die darüberliegenden Schichten bereits anwendungsbezogen sind. Ersetzt man in Bild 6-3 die Verbindungen zwischen den Prozessoren durch ein "lokales Netz", wie z.B. LAN 20, ETHERNET, etc., so stellt ein derartiges Netz eine Ebene 4 zur Verfügung (siehe Bild 6-5).

Dadurch kann die Prozeß-Prozeß-Kommunikation nochmals um eine Ebene angehoben werden.

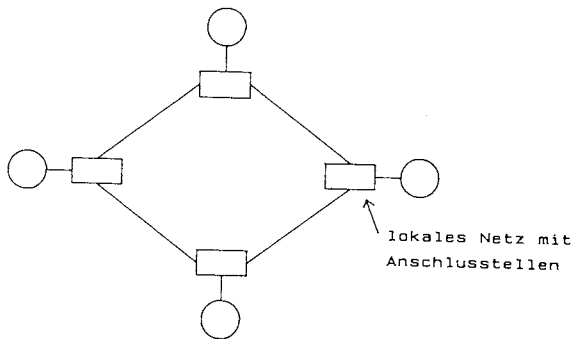


Bild 6-5: Prozeß-Prozeß-Kommunikation über ein lokales Netz

Aufgrund dieser Maßnahme wird auch deutlich, daß ein Protokoll zur Realisierung von Prozeßkommunikationen im Sinne des ISO-Architekturmodells in das Anwendersystem gehört und ein Transportsystem, gleich welcher Mächtigkeit, vorausgesetzt werden kann. Im einfachsten Fall existieren nur direkte Prozessor-Prozessor-Verbindungen, wie am Anfang des Kapitels gezeigt.

Bei lokalen Netzen ("Local Area Network", LAN) dürfen also Schicht-4-Protokolle vorausgesetzt werden. Anders verhält es sich bei Weitverkehrsnetzen ("Wide Area Network", WAN), wie z.B. Datex-P, bei denen lediglich Ebene 3 realisiert ist. Aus diesem Grund sind bereits zahlreiche Hersteller und Entwickler dazu übergegangen, die Transportschicht als Produkt anzubieten (/OTSS87/, /AFHH83/). Oft geht die Entwicklung einer Transportschicht mit der gleichzeitigen Implementierung (/OTSS87/) der Sitzungsschicht ("Session Layer", Schicht 5) einher. Wie der Name schon sagt, dient die Sitzungsschicht zum Erstellen von zeitlich befristeten Sitzungen, die unabhängig von der oder den unterlagerten Transportverbindungen sind. Erbringt die Schicht 4 eine Unabhängigkeit von der Topologie des unterlagerten Netzes, so ermöglicht die Schicht 5 eine zeitliche Unabhängigkeit vom Bestehen der Verbindungen. Wie Bild 6-6 zeigt, kann eine Sitzung mit oder während einer Transportverbindung auf- und wieder abgebaut werden.

Auf die Darstellungsschicht ("Presentation Layer", Schicht 6) soll nicht näher eingegangen werden, da sie lediglich die einheitliche Übertragung von Daten zur Aufgabe hat und damit weder für den Zeitablauf noch auf den eigentlichen Transport von Daten einen Einfluß hat.

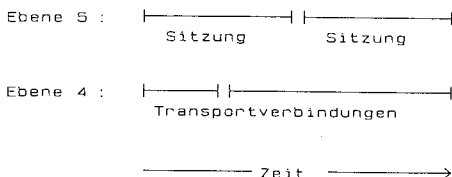


Bild 6-6: Zusammenhang zwischen Sitzungen und Transportverbindungen

Wenn wir ein Protokoll zur Realisierung von Prozeß-Prozeß-Kommunikation einordnen wollen, so bleibt nur noch die Schicht 7 ("Application Layer", Anwendungsschicht). Diese dient zur Unterstützung der unterschiedlichsten Anwendungsprobleme, so daß es, im Gegensatz zu den übrigen Ebenen des Referenzmodells, keine einheitliche

Norm für die Ebene 7 gibt. Es handelt sich vielmehr um ein Bündel von Normen, die sich einzig und allein der Anwendungsunterstützung widmen, während die unterlagerten Schichten Darstellungs- (Ebene 6), Zeit- (Ebene 5), Netz- (Ebene 4) und Datenübertragungs- (Ebene 3) Unabhängigkeit gewährleisten. Auf die Struktur der Schicht 7 und die einzelnen Normen soll in Abschnitt 3 näher eingegangen werden, insbesondere soll ein Vergleich mit dem Baumverfahren durchgeführt werden.

Wurde in diesem Kapitel bisher gezeigt, wie ein Protokoll vom direkten Zugriff auf die Leitung entbunden werden kann, um dadurch die Fähigkeiten der unterlagerten Schichten zu nutzen, so gibt es im Rahmen der internationalen Normung inzwischen schon wieder entgegengesetzte Entwicklungen. Bei der Standardisierung des "Manufacturing Automation Protocol" (MAP, /MAP21/) , einem Protokoll zur Prozeßautomatisierung, waren bisher alle 7 Schichten vorgesehen. Um aber den offensichtlich nicht ausreichenden Durchsatz zu erhöhen, wird im neuesten Normvorschlag (Version 3.0 statt bisher 2.1, /RZE87/) als Erweiterung ein "direkter Durchgriff" auf die Schicht 2 angeboten. Damit reduziert man zwar die Durchlaufzeit durch die Schichten, aber man verliert die Darstellungs-, Zeit- und Netzunabhängigkeit. Man langt also wieder bei der in Abschnitt 6.2.1. geschilderten direkten Botschaftenübermittlung durch die Prozesse selbst an.

Zum Schluß sei noch auf die Implementierungsproblematik eingegangen, die durch das Ansiedeln des Baumverfahrens in der Ebene 7 entsteht. In der Arbeit von Baacke (/BAAC87/) wurde der Versuch unternommen, ein in ein Betriebssystem integriertes Protokoll ("demokratisches Verfahren", /KUMM83/) aus dem Betriebssystem heraus in eine eigene Schicht 7 zu transferieren. Dabei waren folgende, sich widersprechende Anforderungen miteinander zu verknüpfen:

- Das Protokoll ist in das Betriebssystem integriert, um, wie in Bild 6-1 dargestellt, einen möglichst direkten Zugriff auf das Übertragungsmedium (Leitungsebene) zu haben.
- Beim Zugang auf ein öffentliches Netz (z.B. DATEX-P) möchte man sich der von Herstellern angebotenen Programme, die die Schichten 4 und 5 realisieren, bedienen. Diese Programme in das Betriebssystem zu integrieren wäre ein zu aufwendiges Unterfangen.

Will man nun einerseits das bestehende Betriebssystem nicht verändern, andererseits aber Hersteller-Software verwenden, so resultiert daraus ein Architekturbruch (siehe Bild 6-7, /BAAC87/, /HOLL87/):

Anwenderprozesse ("BP"), aus der Sicht des Betriebssystems, rufen Funktionen des

Betriebssystems ("E7") auf, um Nachrichten auszutauschen. Dieses wiederum setzt aus seiner Sicht Schnittstellenaufrufe an Anwenderprozesse (Schicht 5 und 4, "E5" und "E4") ab. Da letztere aber das Betriebssystem zum Versenden der Botschaften benutzen ("E3"), wird jenes indirekt sein eigener Auftraggeber!

Aus diesem Grund ist es notwendig, die Anteile der Schicht 7 aus dem Betriebssystem heraus in eine eigene Schicht zu verlagern.

Die gleiche Problematik ergibt sich beim Baumverfahren, nur, daß sich dabei eine einfachere Lösung findet. Es ist nicht notwendig, Betriebssystem und Protokollverfahren aufwendig zu entflechten, da eine klare Trennung zwischen Betriebssystem (Kapitel 3) und Protokoll (Kapitel 4) bereits stattgefunden hat. In Anlehnung an die Bilder 3-5 und 6-1 ergibt sich aus der Sicht eines Prozesses die in Bild 6-8 gezeigte Aufrufhierarchie:

Ein Prozeß ruft die (reentrant-fähige) Prozedur BAUMVERFAHREN auf und gibt als Prozedurparameter die Kommunikationsanweisung (TRANSMIT/RECEIVE, GUARDED STATEMENT) mit. Das Baumverfahren wickelt sein Protokoll ab und ruft dabei wiederholt die Leitungsprozedur auf, um Botschaften zu senden (und zu empfangen). Ist das Baumverfahren abgeschlossen, wird der Prozeß in seinem Programm fortgesetzt.

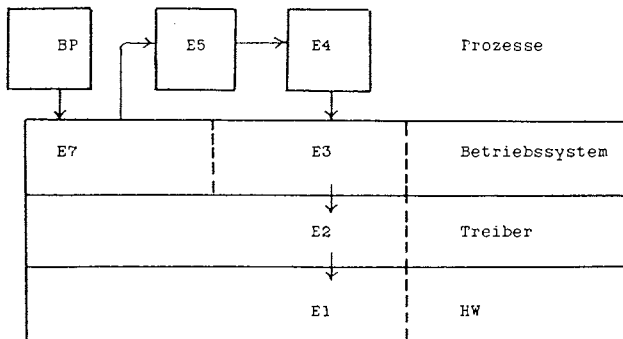


Bild 6-7: Architekturbruch, /BAAC87/

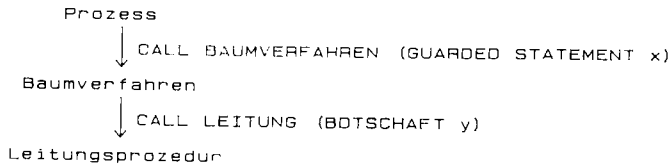


Bild 6-8: Aufrufhierarchie beim Nachrichtenaustausch aus der Sicht eines Prozesses

Der entscheidende Schritt, der zu tun ist, ist der, daß nahezu alle Komponenten aus Bild 6-8 soz. "eine Referenzstufe höher transferiert" werden. Damit ist gemeint, daß "Prozeduren als Prozesse" und "Prozedurparameter als Botschaften" betrachtet werden (siehe Bild 6-9):

Die Prozedur BAUMVERFAHREN wird zu einem Prozeß verselbständigt. Dadurch wird die vom Anwenderprozeß gewünschte Kommunikation selbst zum Nachrichtenparameter einer Kommunikation mit dem BAUMVERFAHREN. Dieses wiederum erstellt Nachrichten, deren Parameter den bisherigen Botschaften entsprechen. Die Prozesse für die Schichten 4 und 5 können dadurch problemlos angesprochen werden. Der Leitungsanschluß muß ebenfalls als Prozeß deklariert werden, indem er, wie in Kapitel 3 bei "Geräten" üblich, als "Verwaltungsprozeß" NETZANSCHLUSS und als "Treiberprozeß" definiert wird.

Trotz dieses "Umdefinierens" von Prozeduren und Prozedurparametern ist die Abwicklung des Baumverfahrens und die Prozeß-Prozeß-Kommunikation gewährleistet, auch wenn letztere nur noch "indirekt" erfolgt. Der grundlegende Unterschied liegt in der Behandlung der vom Baumverfahren erstellten Botschaften, die jetzt nur mehr Parameter von Nachrichten sind. Welche Änderung dies in der "semantischen Bedeutung" der Botschaften hat, wird im folgenden Abschnitt diskutiert.

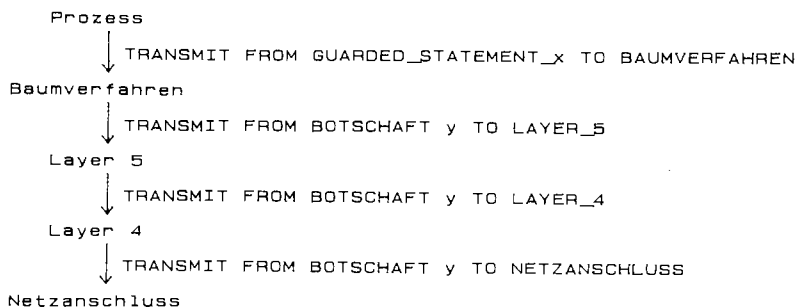


Bild 6-9: Nachrichtenaustausch mit Hilfe standardisierter Protokolle aus der Sicht eines Prozesses

6.2.3. Vergleich zwischen dem Baumverfahren und der ISO-Norm "Transaction Processing"

Wie bereits oben festgestellt, widmet sich die Schicht 7 der Unterstützung der Anwendungen. Da sich Anwendungsprobleme zum Teil sehr gravierend voneinander unterscheiden, wurde erst gar nicht der Versuch unternommen, eine einheitliche Schicht 7 zu entwerfen, sondern es wurden verschiedene anwendungsspezifische Normen entwickelt. Diese Normen, als "Anwendungsdienstelemente" ("Application Service Elements", ASE) bezeichnet, werden in zwei Klassen unterteilt: "Allgemeine Anwendungsdienstelemente" ("Common ASE", CASE) und "Begrenzte (Spezielle) Anwendungsdienstelemente" ("Specific ASE", SASE). Zu letzteren gehören der bereits normierte Nachrichtendienst ("Mail", X.400), der in der Normung sehr weit fortgeschrittene Dateidienst ("File Transfer, Access and Management", FTAM) und der Auftragsdienst ("Remote Job Entry", RJE).

Zu den allgemeinen Anwendungsdienstelementen gehören:

- ACSE ("Association Control Service Element"), das den Verbindungsauf- und -abbau von zwei Prozessen, die miteinander kommunizieren, ermöglicht;
- CCR ("Commitment, Concurrency and Recovery"), das zur Unterstützung paralleler Aktivitäten miteinander kommunizierender Prozesse dient;
- Rose ("Remote Operations Service Element"), das das Einrichten und die Ausführung abgesetzter Operationen unterstützt;
- TP ("Transaction Processing"), das die ACSE- und CCR-Dienste einschließt und durch zusätzliche Datentransfermöglichkeiten **verteiltes Rechnen** ermöglicht.

Diese Dienste sind zur Zeit in der Normdiskussion, wobei vor allem das TP noch sehr heftig diskutiert wird.

Bei der Vorstellung der Schicht-7-Dienstelemente darf nicht vergessen werden, daß für die Ebene 7 auch ein eigenes Modell entwickelt wurde. Im Gegensatz zu den anderen Schichten steht für die Schicht 7 kein geschlossener Block (Prozeß) zur Verfügung, sondern sie besteht aus sogenannten "Application Entities" ("Anwendungseinheiten"). Diese wiederum setzen sich aus Anwendungsdienstelementen zusammen, wobei deren Zusammensetzung von der Anwendung abhängig ist (siehe Bild 6-10). Das bedeutet, daß für jeden Anwenderprozeß mindestens eine eigene Anwendungseinheit

zur Verfügung stehen muß.

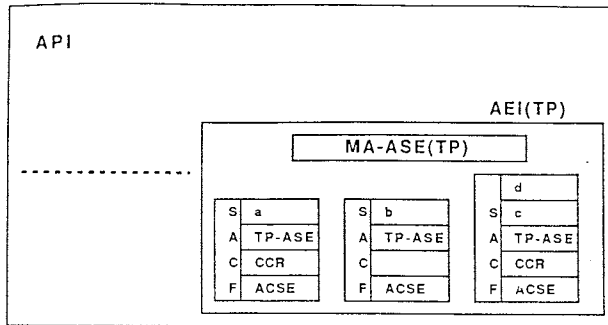
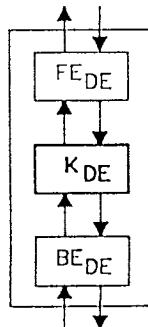


Bild 6-10: Struktur der Schicht 7, bestehend aus Anwendungseinheiten, /ISOTP1/

Bemerkung: Auch aus diesem Grund ist es sinnvoll, die Prozedur Baumverfahren in einen Prozeß umzuwandeln (wie in Kapitel 6.2.2. gezeigt). Dieser Prozeß läßt sich leicht in der Funktion eines allgemeinen Anwendungsdienstelementes in eine Anwendungseinheit integrieren. In dem Artikel von Bever und Fleischmann (/BEVE87/) wird ein Konfigurationskonzept vorgeschlagen, bei dem die allgemeinen Anwendungsdienstelemente prinzipiell als Prozesse formuliert werden und die Anwendungseinheit damit zum Prozeßbündel wird (siehe Bild 6-11).



Dienstelement mit Dienstelementkern, Front End und Back End Prozeß

Bild 6-11: Die Anwendungseinheit als Prozeßbündel, /BEVE87/

Neben dieser architektonischen Eingliederung des Baumverfahrens muß noch ein Vergleich zwischen den Diensten und Protokollen der ISO-Normung und denen des Baumverfahrens durchgeführt werden. Die Dienste beim Baumverfahren werden durch die erweiterte Sprachschale von PEARL, insbesondere durch die Anweisungen TRANSMIT/RECEIVE und GUARDED REGION/COMMAND angegeben. Von den allgemeinen Anwendungsdienstelementen ist vor allem das "Transaction Processing" von Bedeutung, da in diesem Dienste für **verteiltes Rechnen** angeboten werden. Verteiltes Rechnen bedeutet, daß der Schwerpunkt mehr auf der parallelen Ausführung von Anweisungen liegt, als auf der Kommunikation zwischen den Prozessen, die diese Berechnungen durchführen. Dieser Ansatz geht also in Richtung des "remote procedure calls" und nicht in Richtung Prozeß-Prozeß-Kommunikation. Dieses Ergebnis wird auch in /HESS88/ festgestellt, wo der Sprachvorschlag für ein erweitertes PEARL mit den Schnittstellen von TP (/ISOTP1/, /ISOTP2/, /ISOTP3/) und DIPE ("Distributed Interactive Processing Environment", /ECMA85/) verglichen wird.

Beim Vergleich der Protokolle von Baumverfahren und TP muß bei letzterem darauf hingewiesen werden, daß die Fähigkeiten der Sitzungsschicht sehr intensiv genutzt werden. Die Schicht 5 bietet die Möglichkeit "Minor" und "Major Synchronization Points" zu setzen. Durch die "Major Synchronization Points" können Rücksetzpunkte definiert werden, bei denen eine Datensicherung durchgeführt wird. So kann bei Zusammenbrüchen von Verbindungen oder Anwenderprogrammen auf diese Daten zurückgegriffen werden. Die "Minor Synchronization Points" ermöglichen den Kommunikationspartnern, sich gegenseitig zuzusichern, daß ihre Verbindung noch in einem ordnungsgemäßen Zustand ist. Mit Hilfe dieser Schicht-5-Dienstelemente kann im TP-Vorschlag ein gesichertes verteiltes Rechnen erzielt werden.

Neben den Dienstelementen zum Austausch von Daten und zum Setzen von Synchronisationspunkten existieren im TP-Vorschlag auch Elemente für ein "Two Phase Commit"-Protokoll (/GRAY79/). Dieses Protokoll ist der Vorgänger bzw. die Vorstufe des in 2.4. vorgestellten "Three Phase Commit"-Protokolls. In der Normdiskussion ist allerdings noch offen, ob dabei die Dienstelemente von CCR genutzt werden sollen, das auch "Commit und Recovery"-Elemente enthält, oder, ob eventuell im TP ein eigenes "Two Phase Commit"-Protokoll realisiert werden muß.

Der Zweck des Commit-Protokolls im "Transaction Processing" ist jedoch zunächst ein anderer als im Baumverfahren: Es werden Sicherungspunkte gesetzt oder ein Rücksetzen durchgeführt, um die Anwendungseinheiten (Schicht-7-Prozesse) in einem konsistenten Zustand zu halten, aber nicht die Anwenderprozesse selbst. Damit letztere synchron bleiben, werden die Synchronisationselemente, also die aus der Schicht 5 bekannten "Major Synchronization Points", verwendet.

Im Gegensatz dazu war beim Baumverfahren in seiner prozeduralen Fassung das TPCP benutzt worden, den Anwenderprozessen selbst eine synchrone Kommunikation untereinander zu erlauben und für sie einen fehlerfreien Zustand zu gewährleisten. Für den Übergang des Baumverfahrens in einen Prozeß bedeutet das, daß auch hier das TPCP nur dazu dienen kann, die Prozesse "BAUMVERFAHREN", die an der Kommunikation beteiligt sind, rücksetzbar zu machen. Was die Rücksetzfähigkeit der Anwenderprozesse angeht, so ist darüber zu diskutieren, inwieweit das Verteilte PEARL dahingehend ergänzt werden muß. Aber das führt wieder zum Vergleich zwischen den angebotenen Diensten von Verteiltem PEARL und Transaction Processing (/HESS88/).

Zwischen dem TP-Vorschlag und dem Baumverfahren existiert noch eine weitere Ähnlichkeit: in beiden werden Bäume verwendet. Allerdings dient der Baum im TP zur Festschreibung der Kommunikationsstruktur, die vom Verbindungsaufbau bis zum Verbindungsabbau gilt. Die Knoten und Blätter eines Baumes stellen die an der Kommunikation beteiligten Prozesse dar, wobei nur jeweils die direkt miteinander verbundenen Nachbarknoten (Nachbarprozesse) Daten austauschen dürfen.

Im Verteilten PEARL ist dagegen eine beliebig vermaschte Kommunikationsstruktur möglich. Die Bäume im Baumverfahren stellen nur die Beziehungen zwischen dem Zentrum und den übrigen Kommunikationspartnern für einen relativ kurzen Zeitpunkt dar. Sie haben nicht die Aufgabe, die allgemeine Verbindungsstruktur festzulegen.


7. Analyse und Diskussion des vorgeschlagenen Konzeptes

Bei Rechnern oder Systemprogrammen, die heutzutage verwendet werden, ist zwar eine relativ hohe Verfügbarkeit vorauszusetzen, aber dennoch kann man keine absolut sicheren Maschinen und darauf ablaufende Routinen gewährleisten. Deshalb ist man bereits früh dazu übergegangen, Programme dahingehend zu erweitern, daß zu bestimmten Zeitpunkten oder nach gewissen wichtigen Aktionen, Sicherungskopien von Daten angefertigt werden. Sind derartige Erweiterungen nicht verfügbar, so hat sich zumindest der Anwender daran gewöhnt, selbst Rettkopien herzustellen, spätestens nach einem fatalen Fehler, der ihm wichtiges vernichtet hat. Deshalb "sollte" es bei Klein- und Großrechnern üblich sein, daß man auf eine "alte" Kopie unter Verlust der seit Erstellung dieser Kopie gemachten Änderungen zurückgreifen kann.

Bei der Entwicklung physisch verteilter Systeme war man zunächst damit beschäftigt, überhaupt Rechner und Programme mit Hilfe physikalischer Leitungen und Protokolle miteinander zu verbinden. Diese verteilten Konfigurationen auch robust gegen Fehler zu machen, war dagegen eine später einsetzende Tendenz. Dabei ist eine ähnliche Vorgehensweise, wie oben beschrieben, nötig: Es werden Daten gesichert, auf die im Fehlerfall zurückgegriffen werden kann.

Das bedeutet auch, wenn auf eine Protokollierung der Aktionen seit der letzten Sicherung verzichtet wird, daß diese Aktionen alle wiederholt werden müssen. Deshalb erscheint es sinnvoll, zusätzlich zu den Rettdaten die Schritte der Rechenaktionen zu sichern. Dennoch wurde in dieser Arbeit auf eine derartige Protokollierung explizit verzichtet. Dies wäre prinzipiell auch für verteilte Systeme und für das in dieser Arbeit vorgestellte Modell möglich. Aber als zusätzliches Problem kommt noch die Notwendigkeit, technische Prozesse zu steuern, hinzu. Wie in Kapitel 5.7. festgehalten wurde, ist es zwar möglich, Software- Prozesse, also Programme, zurückzusetzen, technische Prozesse dagegen, schreiten unverändert voran. Ein einfaches Beispiel sei dafür nachfolgend gegeben.

Beispiel: Es wird ein Systemausschnitt mit einem beliebigen programmierten Prozeß und einem technischen Prozeß in der Form eines digitalen Signalgebers, der entweder den Wert "0" oder den Wert "1" liefert, gezeigt.

Programm:	Signalgeber liefert die Werte:	
Rücksetzpunkt	0	
	0	
Feststellen eines Fehlers	.	
	.	
Beginn der Rücksetzaktion	0	
	1	
Ende der Rücksetzaktion und Beginn der Wiederho- lung der Programmschritte	1	
	.	
	.	
Alle Schritte wiederholt	1	

Greift man beim Rücksetzen nicht nur auf die letzten Sicherungsdaten zurück, sondern auch auf die protokollierten Aktionen, so rechnet in der Phase, in der die Programmschritte wiederholt werden, das Programm noch mit den "0"-Werten, obwohl der Signalgeber bereits die "1"-Werte liefert.

Dieses einfache Beispiel zeigt, daß die Programme immer mit den aktuellsten Daten der technischen Prozesse versorgt werden müssen. Sie werden deshalb auf ihre letzten Sicherungsdaten zurückgesetzt, aber nach dem Wiederanlaufen sind durchaus andere Ausführungsreihenfolgen möglich.

Die in dieser Arbeit gewählten Sicherungsmaßnahmen reichen also aus, die Erfordernisse technischer Prozesse zu erfüllen (Kapitel 5). Außerdem ist durch das gemeinsame Setzen eines Rücksetzpunktes mehrerer Prozesse gewährleistet, daß die Prozesse im Fehlerfall die gleiche Sicht auf die letzte festgehaltene Situation behalten.

Um eine gleiche, d.h. gemeinsame, Sicht auf die Kommunikation und Synchronisation zu gewährleisten, wurde in Kapitel 4 ein zentraler Botschaftenalgorithmus eingeführt. Er zeichnet sich dadurch aus, daß nur einzelne, dynamisch ausgewählte Prozesse die Kommunikation steuern. Zum einen ist dadurch das Protokoll übersichtlich, zum anderen aber entsteht keine Abhängigkeit von einem einzigen Prozeß.

Schließlich wurde in Kapitel 3 die Basis für ein einheitliches Protokoll geschaffen, indem ein (Betriebssystem-) Modell entworfen wurde, das nur noch aus den Komponenten **Prozeß** und **Botschaft** besteht. Dadurch wird eine Vereinheitlichung aller Komponenten des gesamten Systems und aller weiteren Maßnahmen, wie z.B. der Datensicherung, erreicht. Diese Vorgehensweise hat nicht nur den Vorteil, daß spezielle Komponenten, wie "globale Daten", in das Konzept nahtlos eingefügt werden können (Abschnitt 5.6.), sondern, daß auch eine Weiterentwicklung im Rahmen des ISO/OSI-7-Schichten-Architekturmodells möglich ist (Kapitel 6.2.).

Versucht man einen Vergleich mit den in Kapitel 2 vorgestellten Konzepten, so ist das deshalb schwierig, weil sie fast alle, genauso wie der "Transaction Processing"-Vorschlag (siehe Abschnitt 6.2.3.), den "remote procedure call" (RPC) als Kommunikationsmittel einsetzen. So stehen der RPC und der Botschaftenmechanismus im Verteilten PEARL als Kommunikationsmöglichkeiten nebeneinander. Sie sind zwar größtenteils aufeinander abbildbar, bei Shrivastava sind die Botschaften Grundlage zur Realisierung des RPC-Mechanismus (siehe 2.3.1.), doch bleiben Unterschiede. Deshalb darf es nicht zur "Pflicht" werden, sich für einen der beiden Mechanismen entscheiden zu müssen, sondern es muß von Anwendungs- zu Anwendungsfall offen gelassen werden, welcher von beiden besser geeignet ist. Genauso wie man abzuwägen hat, welcher spezifische Rechner oder welche spezifische Programmiersprache günstiger für ein bestimmtes Problem ist.

Die Problematik des Sicherns und Rücksetzens von Prozessen ist allerdings unabhängig von der Wahl des Konzeptes, wenn die grundlegenden Erfordernisse eingehalten werden:

- Keine gegenseitige Einflußnahme von Prozessen außerhalb der Synchronisations- und Kommunikationsanweisungen;
- Einhalten des Prinzips der "atomaren Aktion": Kommunikationen werden entweder vollständig oder überhaupt nicht durchgeführt;
- Setzen von gemeinsamen Wiederaufsetzpunkten durch alle an der Kommunikation beteiligten Prozesse.

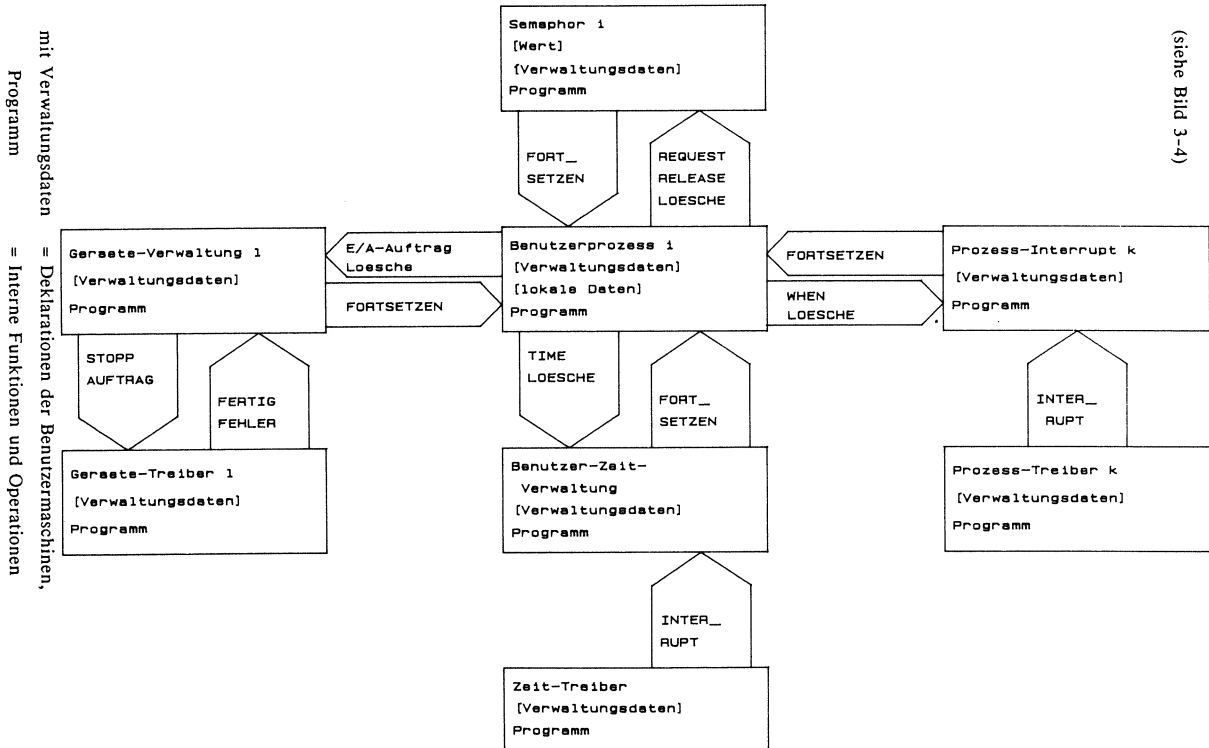
Anhang 1

"Spezifikation eines anlagenunabhängigen Betriebssystems als ein System kommunizierender Prozesse"

Die nachfolgende Spezifikation erfolgt mit Hilfe der Methode PASS (/FLEI84/), wobei die Beschreibung der Benutzermaschinen in PASCAL-ähnlicher Notation erfolgt.

1 Kommunikationsstruktur

(siehe Bild 3-4)



mit Verwaltungsdaten
= Deklarationen der Benutzermaschinen,
= Interne Funktionen und Operationen

2 Typen von Strukturierungseinheiten

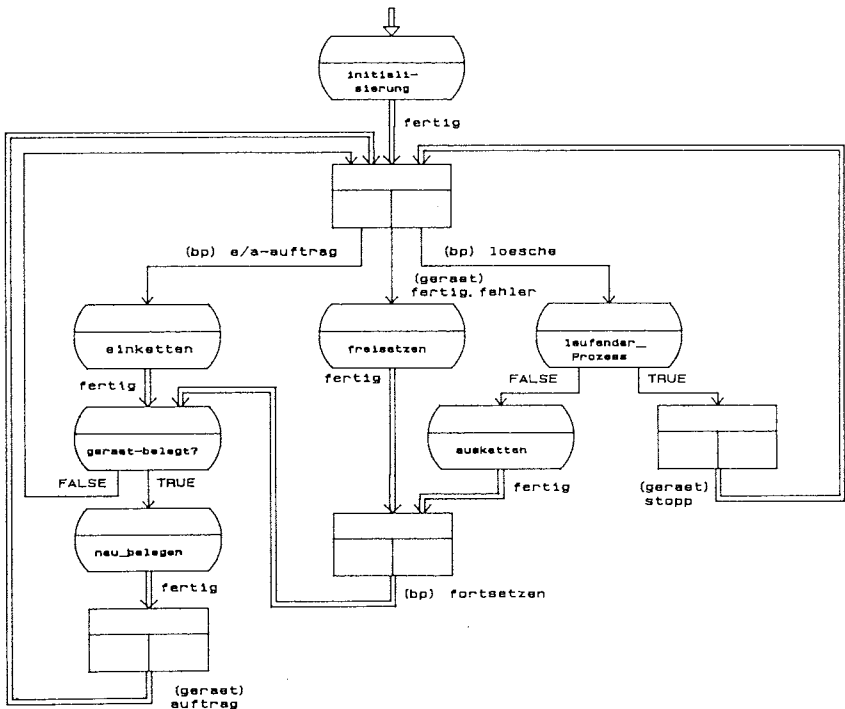
geraeteinterrupt EINZELPROZESSTYP (benutzerprozeß, geraet)

(* Dieser Prozeß verwaltet jeweils ein Ein- oder Ausgabegerät,
das zu einem Zeitpunkt nur von einem Benutzerprozeß belegt
werden kann *)

1 KOMMUNIKATIONSMASCHINE

PUFFER: 0

2 ABLAUFSTEUERUNG



3 BENUTZERMASCHINE

3.1 INTERNE OPERATIONEN : ausketten, einketten, freisetzen, initialisierung,
neu_belegen

3.2 INTERNE FUNKTIONEN : geraet_belegt?, laufender_prozeß?

3.3 EINGABEOPERATIONEN : e/a-auftrag, fehler, fertig, lösche

3.4 AUSGABEFUNKTIONEN : auftrag, fortsetzen, stopp

3.5 DEFINITION DER BENUTZERMASCHINE:

(* Deklarationen *)

TYPE bp (* Benutzerprozeß *)

```
    = RECORD      name           : CHAR(8);
                   vorgänger, nachfolger : ^bp;
                   prio           : INTEGER;
                   e/a-auftrag      : adresse;
                   (* Zeiger auf E/A-Treiber-Auftragsblock *)
                   belegt           : ^bp;
    END;
```

```
VAR   benpro, device      : ^bp;
       fehler             : INTEGER;
       lname              : CHAR(8);
```

(* Interne Operationen *)

ausketten;

(* Suchen des Kontrollblocks, so daß gilt benpro^.name = lname *)

benpro^.nachfolger^.vorgänger := benpro^.vorgänger;

benpro^.vorgänger^.nachfolger := benpro^.nachfolger;

DISPOSE (benpro);

(* fehler := Abbruch *)

END;

```

einketten;

(* Der Benutzerprozeß, festgehalten durch den Zeiger 'benpro', muß gemäß seiner
   Priorität eingekettet werden, so daß gilt:
   benpro^.vorgänger^.prio < benpro^.prio <= benpro^.nachfolger^.prio *)
END;

freisetzen;
benpro := device^.belegt;
device^.belegt := NIL;
DISPOSE (benpro);
END;

initialisierung;

(* Aus Einfachheitsgründen wird darauf verzichtet, für den Interruptkontrollblock
   eine eigene Struktur zu definieren. Da die Komponenten 'nachfolger' und
   'vorgänger' typgebunden sind, hätte ein eigener Kontrollblock zur Folge,
   daß man bei diesen Komponenten Fallunterscheidungen einführen muß. *)
NEW (device);
WITH device^ DO
    vorgänger := device^;
    nachfolger := device^;
    belegt    := NIL;
END;
END;

neu_belegen;
WITH device^ DO
    belegt := nachfolger;
    nachfolger := nachfolger^.nachfolger;
    nachfolger^.vorgaenger := device;
END;
END;

(* Interne Funktionen *)

geraet_belegt?
RETURN( device^.belegt=NIL ANDdevice^.nachfolger <> device );

laufender_prozeß? ( lname : char(8) );
RETURN ( device^.belegt^.name = lname );

```

(* Eingabeoperationen *)

```
e/a-auftrag ( iname : CHAR(8); aprio : INTEGER; auftrag: adresse );  
NEW (benpro);  
WITH benpro^ DO  
    name      := iname;  
    prio      := aprio;  
    e/a-auftrag := auftrag;  
END;  
END;
```

```
fehler ( gfehler : INTEGER );  
fehler := gfehler;  
END;
```

```
fertig;  
fehler := 0;  
END;
```

```
lösche ( name : char(8) );  
lname := name;  
END;
```

(* Ausgabefunktionen *)

```
auftrag ( device^.belegt^.e/a-auftrag : adresse );  
END;
```

```
fortsetzen ( VAR gfehler : INTEGER );  
gfehler := fehler;  
END;
```

```
stopp;  
END;
```

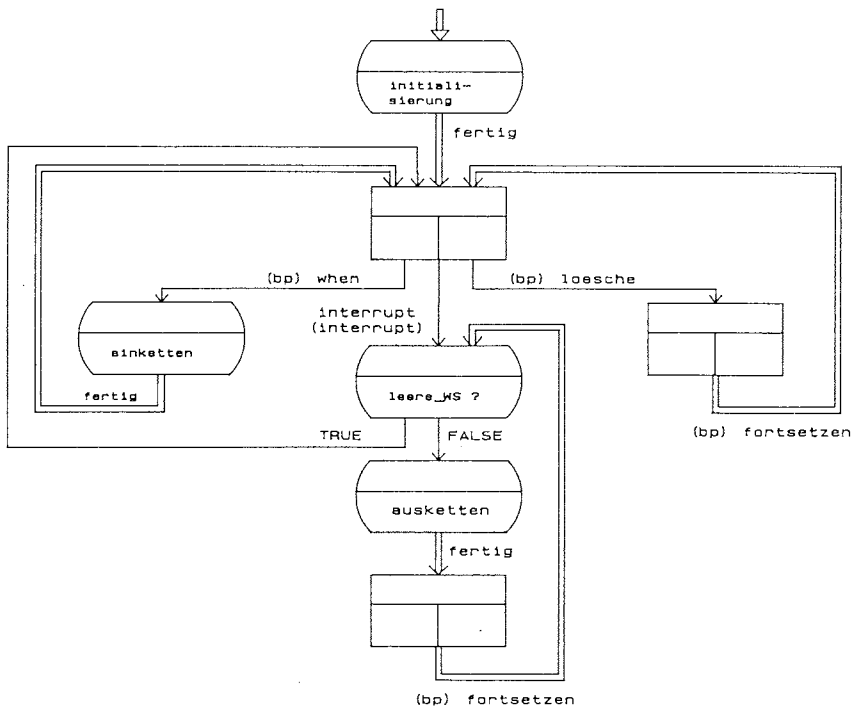
prozeßinterrupt EINZELPROZESSTYP (benutzerprozeß, interrupt)

(* Verwaltung eines Prozeß-Interrupts, so daß beliebig viele Prozesse auf das Eintreffen eines Interrupts warten können *)

1 KOMMUNIKATIONSMASCHINE

PUFFER: 0

2 ABLAUFSTEUERUNG



3 BENUTZERMASCHINE

3.1 INTERNE OPERATIONEN : ausketten, einketten, initialisierung

3.2 INTERNE FUNKTIONEN : leere_WS?

3.3 EINGABEOPERATIONEN : interrupt, lösche, when

3.4 AUSGABEFUNKTIONEN : fortsetzen

3.5 DEFINITION DER BENUTZERMASCHINE:

(* Deklarationen *)

TYPE bp (* Benutzerprozeß *)

```
    = RECORD      name           : CHAR(8);
                  vorgänger, nachfolger : ^bp;
                  fortsetzen       : INTEGER;
                  (* mit ACTIVATE = 1, CONTINUE = 2 *)
    END;
```

```
VAR    benpro, intrupt      : ^bp;
        ufortsetzen        : INTEGER;
```

(* Interne Operationen *)

```
ausketten;
benpro := intrupt^.nachfolger;
ufortsetzen := benpro^.fortsetzen;
WITH intrupt^ DO
    benpro^.nachfolger^.vorgänger := intrupt;
    nachfolger := benpro^.nachfolger;
END;
DISPOSE (benpro);
END;
```

```

einketten;
WITH intrupt^ DO
    benpro^.nachfolger := nachfolger;
    nachfolger^.vorgänger := benpro;
    nachfolger := benpro;
    benpro^.vorgänger := intrupt;
END;
END;

initialisierung;
(* Aus Einfachheitsgründen wird darauf verzichtet, für den Interrupt-Kontrollblock
   eine eigene Struktur zu definieren. Da die Komponenten 'nachfolger' und
   'vorgänger' typgebunden sind, hätte ein eigener Kontrollblock zur Folge,
   daß man bei diesen Komponenten Fallunterscheidungen einführen muß. *)
NEW (intrupt);
WITH intrupt^ DO
    vorgänger := intrupt^;
    nachfolger := intrupt^;
END;
END;

(* Interne Funktionen *)

leere_ws?
RETURN ( intrupt^.nachfolger = intrupt );
END;

(* Eingabeoperationen *)

interrupt;
(* Signalisieren, daß ein Interrupt eingetroffen ist. *)
END;

lösche ( lname CHAR(8) );
(* Suchen des Kontrollblocks, so daß gilt benpro^.name = lname *)
benpro^.nachfolger^.vorgänger := benpro^.vorgänger;
benpro^.vorgänger^.nachfolger := benpro^.nachfolger;
ufortsetzen := 0; (* nicht definiert *)
DISPOSE (benpro);
END;

```

```
when ( rname : CHAR(8); rfortsetzen : INTEGER );  
  NEW (benpro);  
  benpro^.name := rname;  
  benpro^.fortsetzen := rfortsetzen;  
END;
```

(* Ausgabefunktionen *)

```
fortsetzen ( VAR ufortsetzen : INTEGER );  
END;
```

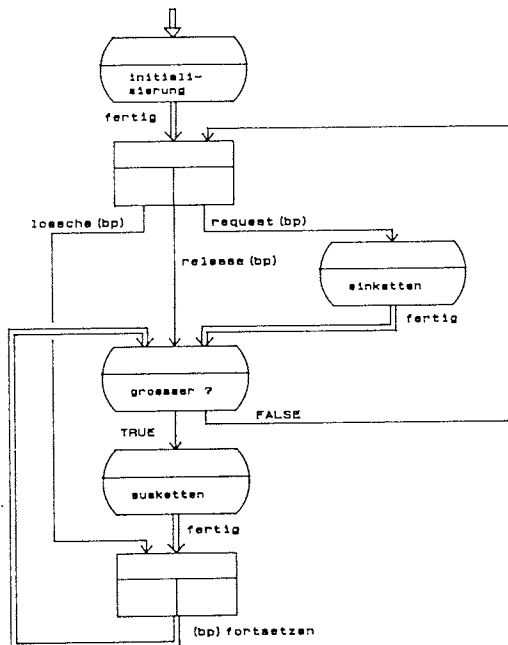
semaphor EINZELPROZESSTYP (benutzerprozeß)

(* Beschreibung einer P/Vc-Variablen und ihrer Operationen *)

1 KOMMUNIKATIONSMASCHINE

PUFFER: 0

2 ABLAUFSTEUERUNG



3 BENUTZERMASCHINE

3.1 INTERNE OPERATIONEN : ausketten, einketten, initialisierung

3.2 INTERNE FUNKTIONEN : größer?

3.3 EINGABEOPERATIONEN : lösche, release, request

3.4 AUSGABEFUNKTIONEN : fortsetzen

3.5 DEFINITION DER BENUTZERMASCHINE:

(* Deklarationen *)

```
TYPE bp (* Benutzerprozeß *)
    = RECORD      name           : CHAR(8);
                  wert, prio      : INTEGER;
                  vorgänger, nachfolger : ^bp;
    END;
```

```
VAR    benpro, pv__sema : ^bp;
```

(* Interne Operationen *)

```
ausketten;
benpro := pv__sema^.nachfolger;
WITH pv__sema^ DO
    wert := wert - benpro^.wert;
    benpro^.nachfolger^.vorgänger := pv__sema;
    nachfolger := benpro^.nachfolger;
END;
DISPOSE (benpro);
END;

einketten;
(* Der Benutzerprozeß, festgehalten durch den Zeiger 'benpro', muß gemäß seiner
   Priorität eingekettet werden, so daß gilt:
   benpro^.vorgänger^.prio < benpro^.prio <= benpro^.nachfolger^.prio *)
END;
```

initialisierung;

```
(* Aus Einfachheitsgründen wird darauf verzichtet, für den Semaphorkontroll-
block eine eigene Struktur zu definieren. Da die Komponenten 'nachfolger'
und 'vorgänger' typgebunden sind, hätte ein eigener Kontrollblock zur
Folge, daß man bei diesen Komponenten Fallunterscheidungen einführen muß.
*)
```

```
NEW (pv_sema);
```

```
WITH pv_sema^ DO
```

```
  wert := 0; (* oder ein vom Benutzer gewünschter Initialwert *)
```

```
  vorgänger := pv_sema^;
```

```
  nachfolger := pv_sema^;
```

```
END;
```

```
END;
```

```
(* Interne Funktionen *)
```

```
größer?
```

```
IF pv_sema^.nachfolger /= pv_sema
```

```
  THEN RETURN ( pv_sema^.wert >= pv_sema^.nachfolger^.wert)
```

```
  ELSE RETURN ( FALSE );
```

```
END;
```

```
(* Eingabeoperationen *)
```

```
lösche ( lname CHAR(8) );
```

```
(* Suchen des Kontrollblocks, so daß gilt benpro^.name = lname *)
```

```
benpro^.nachfolger^.vorgänger := benpro^.vorgänger;
```

```
benpro^.vorgänger^.nachfolger := benpro^.nachfolger;
```

```
DISPOSE (benpro);
```

```
END;
```

```
release ( rwert : INTEGER );
```

```
pv_sema^.wert := pv_sema^.wert + rwert;
```

```
END;
```

```

request ( rwert, rprio : INTEGER; rname : CHAR(8) );
NEW (benpro);
WITH benpro^ DO
    name := rname;
    prio := rprio;
    wert := rwert;
END;
END;

(* Ausgabefunktionen *)

fortsetzen;
END;

```

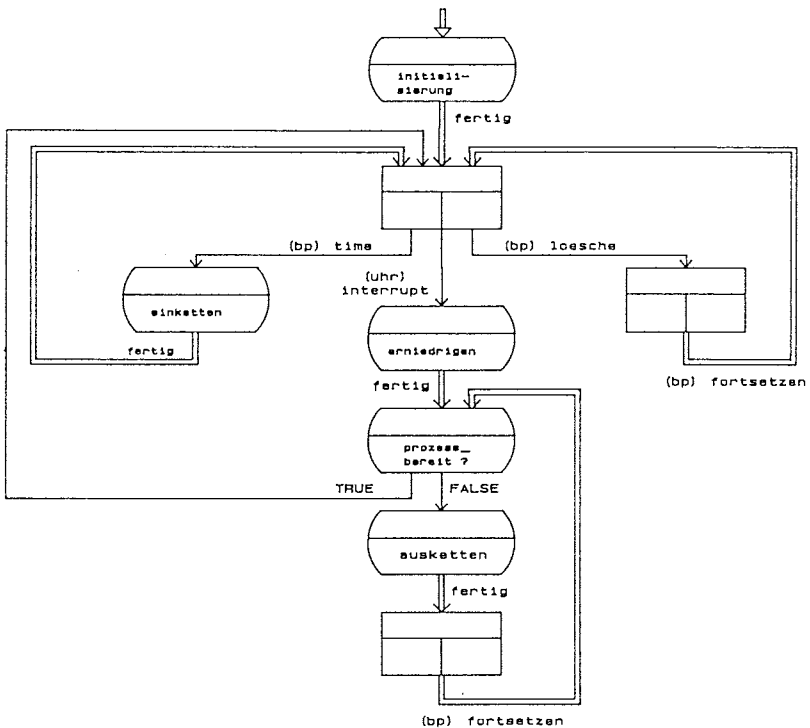
zeitinterrupt EINZELPROZESSTYP (benutzerprozeß, uhr)

(* Dieser Prozeß soll Benutzereinplanungen der Art AFTER, ALL und AT ermöglichen, wobei auch die Kombinationen AFTER/ALL und AT/ALL erlaubt sind *)

1 KOMMUNIKATIONSMASCHINE

PUFFER: 0

2 ABLAUFSTEUERUNG



3 BENUTZERMASCHINE

3.1 INTERNE OPERATIONEN : ausketten, einketten, erniedrigen, initialisierung,
umketten

3.2 INTERNE FUNKTIONEN : prozess__bereit?

3.3 EINGABEOPERATIONEN : interrupt, lösche, time

3.4 AUSGABEFUNKTIONEN : fortsetzen

3.5 DEFINITION DER BENUTZERMASCHINE:

(* Deklarationen *)

TYPE bp (* Benutzerprozeß *)

```
    = RECORD      name           : CHAR(8);
                  vorgänger, nachfolger : ^bp;
                  art             : INTEGER;
                  (* mit AT = 1, AFTER = 2, ALL = 10, AT-ALL = 11,
                     AFTER-ALL = 12 *)
                  all             : DURATION;
                  fortsetzen      : INTEGER;
                  (* mit ACTIVATE = 1, CONTINUE = 2 *)
                  timer           : DURATION;
    END;
```

```
VAR   benpro, timer      : ^bp;
      ufortsetzen        : INTEGER;
```

(* Interne Operationen *)

```
ausketten;
benpro := timer^.nachfolger;
ufortsetzen := benpro^.fortsetzen;
WITH timer^ DO
    benpro^.nachfolger^.vorgänger := timer;
    nachfolger := benpro^.nachfolger;
END;
```

```

IF benpro^.art > 9
THEN benpro^.timer := benpro^.all;
    (* Einketten, so daß gilt: benpro^.vorgänger^.timer <
        benpro^.timer <= benpro^.nachfolger^.timer *)
ELSE DISPOSE (benpro);
FIN;
END;

einketten ( benpro : ^bp );
(* Einketten, so daß gilt:
    benpro^.vorgänger^.timer < benpro^.timer <= benpro^.nachfolger^.timer *)
END;

erniedrigen;
(* für alle eingeketteten Benutzerprozesse:
    benpro^.timer := benpro^.timer - 1; *)

initialisierung;
(* Aus Einfachheitsgründen wird darauf verzichtet, für den Interruptkontrollblock
    eine eigene Struktur zu definieren. Da die Komponenten 'nachfolger' und
    'vorgänger' typegebunden sind, hätte ein eigener Kontrollblock zur Folge,
    daß man bei diesen Komponenten Fallunterscheidungen einführen muß. *)
NEW (timer);
WITH timer^ DO
    vorgänger := timer^;
    nachfolger := timer^;
END;
END;

(* Interne Funktionen *)

prozess_bereit?
RETURN ( timer^.nachfolger^.timer > 0 );
END;

(* Eingabeoperationen *)

interrupt;
(* Signalisieren, daß ein Interrupt eingetroffen ist. *)
END;

```

```

lösche ( lname : char(8) );
(* Suchen des Kontrollblocks, so daß gilt benpro^.name = lname *)
benpro^.nachfolger^.vorgänger := benpro^.vorgänger;
benpro^.vorgänger^.nachfolger := benpro^.nachfolger;
DISPOSE (benpro);
ufortsetzen := 0; (* nicht definiert *)
END;

time ( ename : char(8); eart, efortsetzen : INTEGER; eall, etimer : DURATION
);
NEW (benpro);
WITH benpro^ DO
    name      := ename;
    art       := eart;
    fortsetzen := efortsetzen;
    IF eart > 9
    THEN all := eall; (* in kleinsten Zeiteinheiten *)
    FIN;
    CASE eart OF
        1, 11 : timer := etimer - REALTIME (MOD 24 HRS);
        2, 12 : timer := etimer;
        10 : timer := eall;
    FIN;
END;
END;

(* Ausgabefunktionen *)

fortsetzen ( VAR ufortsetzen : INTEGER );
END;

```

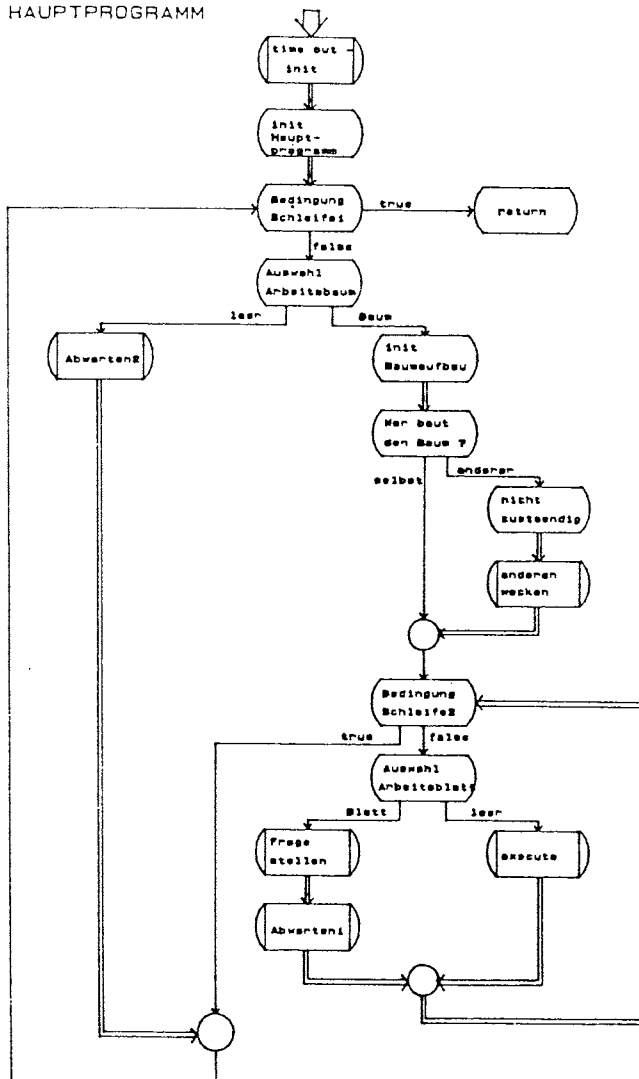
Anhang 2

"Spezifikation des Baumverfahrens für die synchrone Kommunikation"

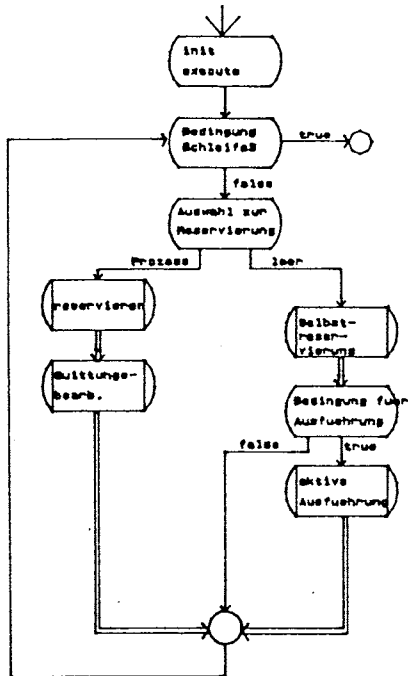
Die hier wiedergegebene Spezifikation des Baumverfahrens behandelt nur die synchrone Kommunikation von Benutzerprozessen über TRANSMIT / RECEIVE, GUARDED COMMAND und GUARDED REGION. Sie ist ebenso in der Diplomarbeit von E. Übelmesser (/UEBE87/) enthalten. Dort sind auch die für den Entwurf der Spezifikation gemachten Entwurfsentscheidungen angegeben.

ABLAUFSTEUERUNG

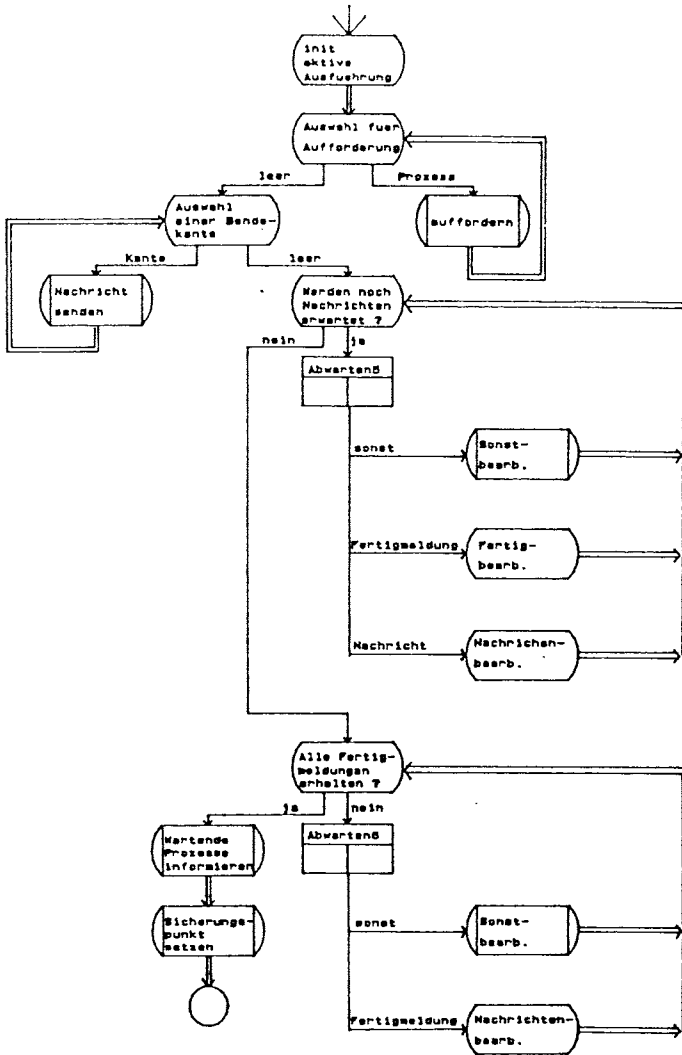
HAUPTPROGRAMM



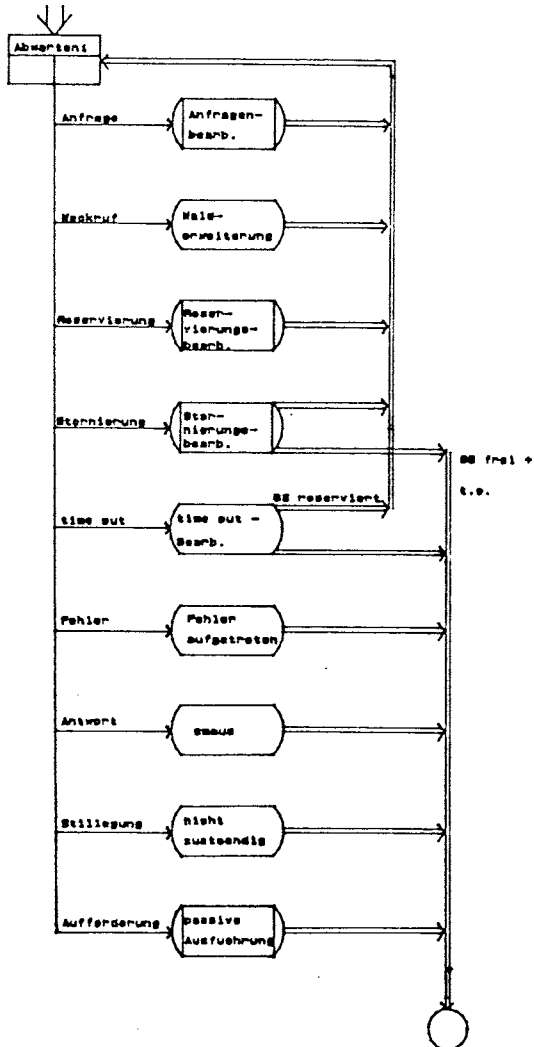
EXECUTE



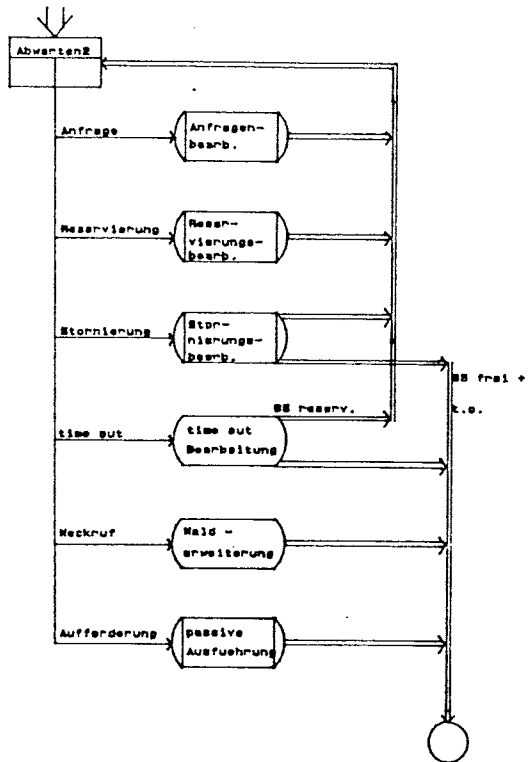
AKTIVE AUSFUEHRUNG



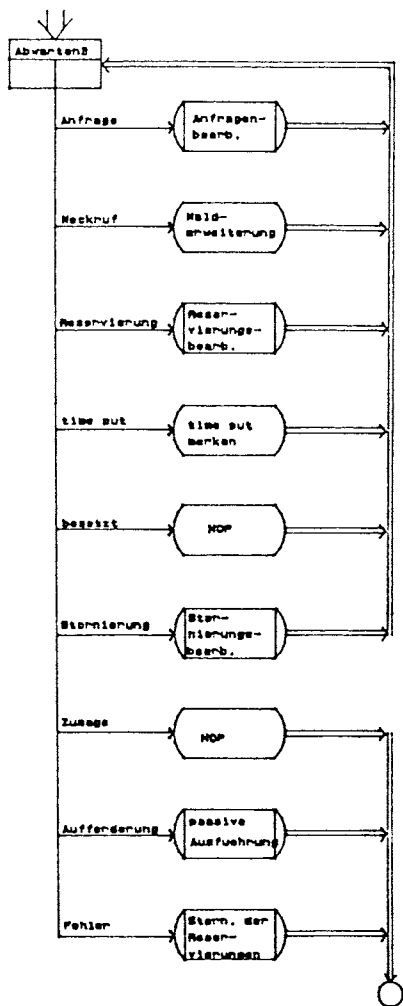
ABWARTEN1



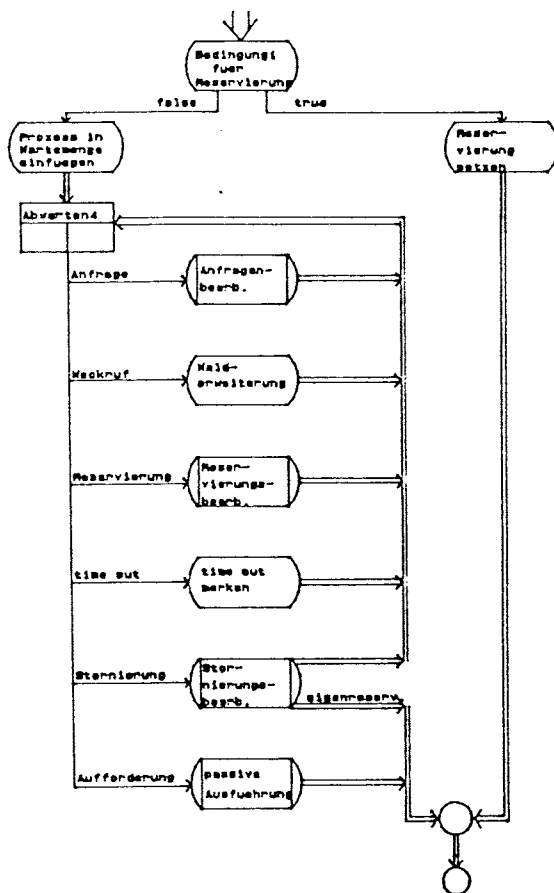
ABWARTEN2



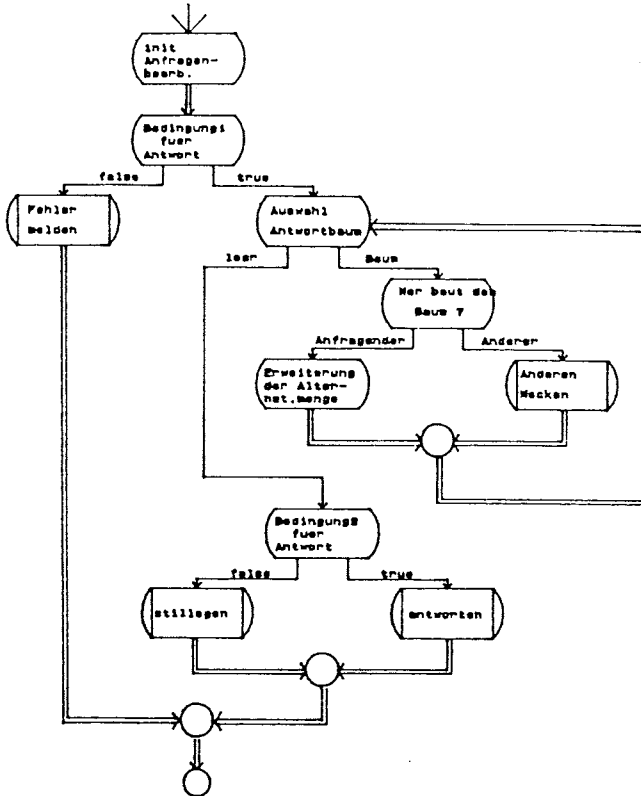
QUITTUNGSBEARBEITUNG



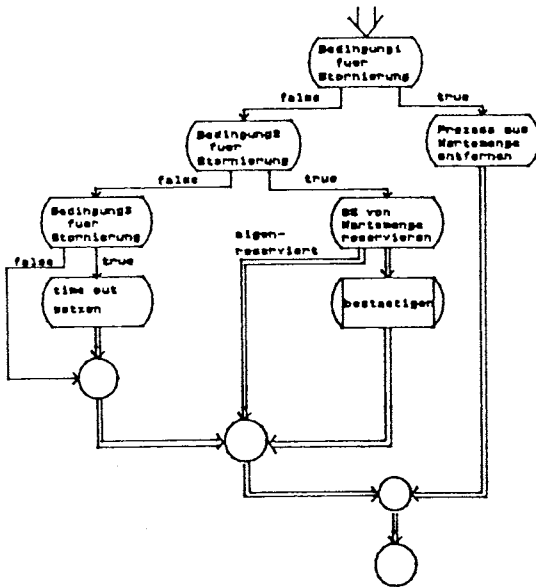
SELBSTRESEERVIERUNG



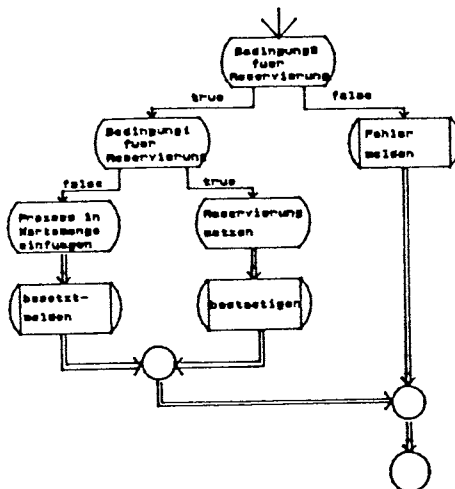
ANFRAGENBEARBEITUNG



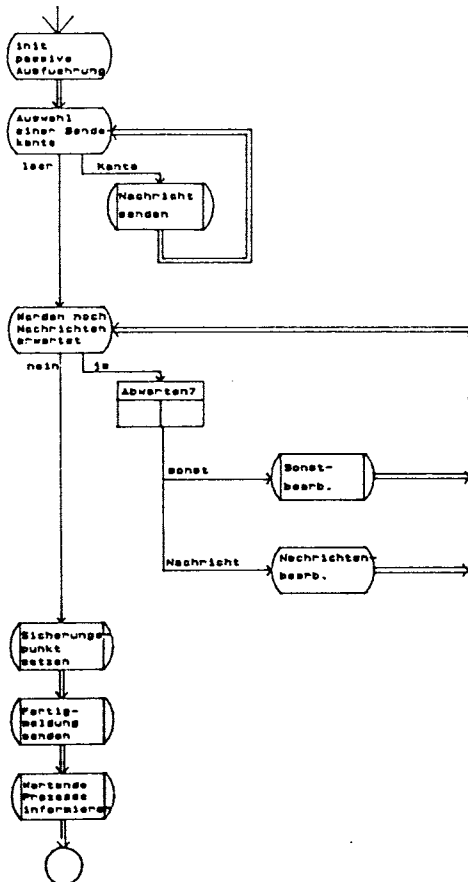
STORNIERUNGSBEARBEITUNG



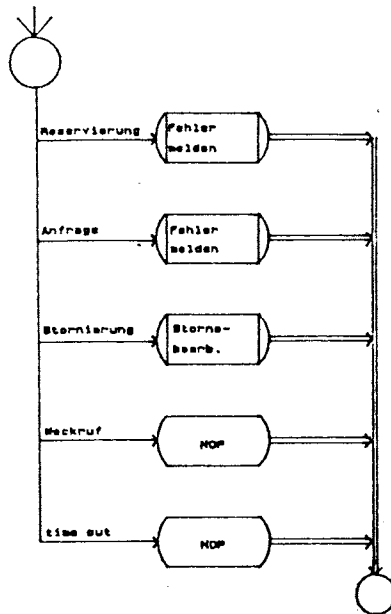
RESERVIERUNGSBEARBEITUNG



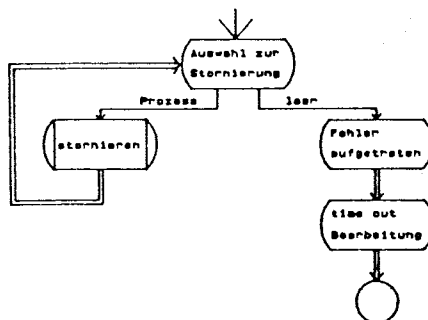
PASSIVE AUSFUEHRUNG



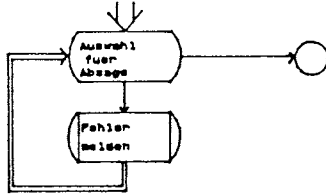
SONSTBEARBEITUNG



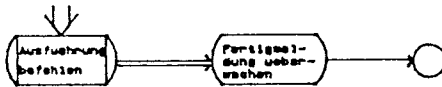
STORNIERUNG DER RESERVIERUNGEN



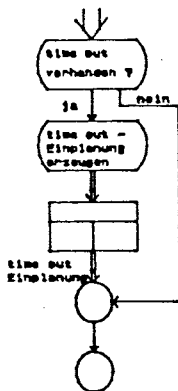
WARTENDE PROZESSE INFORMIEREN



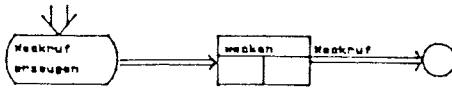
AUFFORDERN



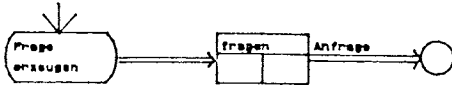
TIME OUT INIT



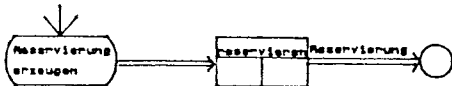
ANDEREN WECKEN



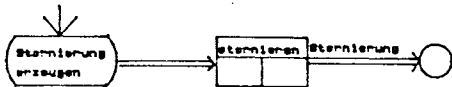
FRAGE STELLEN



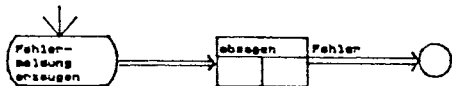
RESERVIEREN



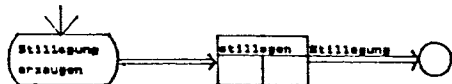
STORNIEREN



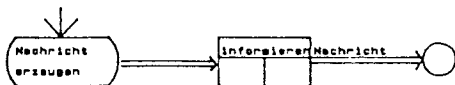
FEHLER MELDEN



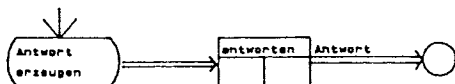
STILLEGEN



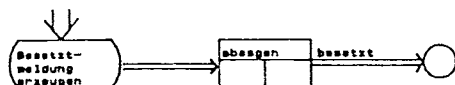
NACHRICHT SENDEN



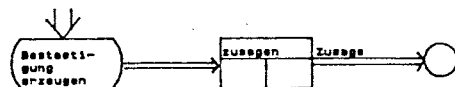
ANWORTEN



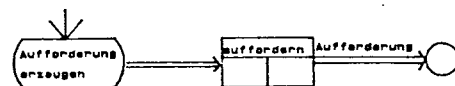
BESETZTMELDEN



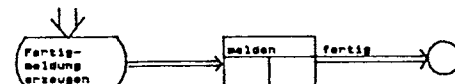
BESTÄTIGEN



AUSFÜHRUNG BEFEHLEN



FERTIGMELDUNG SENDEN



BENUTZERMASCHINE

1 Interne Operationen

cmaus, Erweiterung der Alternativenmenge, Fehler aufgetreten, Fertigmeldung überwachen, Fertigbearbeitung, GS von Wartemenge reservieren, init aktive Ausführung, init Anfragenbearbeitung, init Baumaufbau, init execute, init Hauptprogramm, init passive Ausführung, Nachrichtenbearbeitung, nicht zuständig, Prozeß aus Wartemenge entfernen, Prozeß in Wartemenge einfügen, Reservierung setzen, time out Bearbeitung, time out merken, time out setzen, Walderweiterung,

Antwort erzeugen, Aufforderung erzeugen, Besetztmeldung erzeugen, Bestätigung erzeugen, Fehlermeldung erzeugen, Fertigmeldung erzeugen, Frage erzeugen, Nachricht erzeugen, Reservierung erzeugen, Stillegung erzeugen, Stornierung erzeugen, time out Einplanung erzeugen, Weckruf erzeugen.

2 Interne Funktionen

Alle Fertigmeldungen erhalten?, Auswahl Antwortbaum, Auswahl Arbeitsbaum, Auswahl Arbeitsblatt, Auswahl einer Sende-Kante, Auswahl für Absage, Auswahl für Aufforderung, Auswahl zur Reservierung, Auswahl zur Stornierung Bedingung für Ausführung, Bedingung1 für Antwort, Bedingung2 für Antwort, Bedingung1 für Reservierung, Bedingung2 für Reservierung, Bedingung1 für Stornierung, Bedingung2 für Stornierung Bedingung3 für Stornierung, Bedingung Schleife1, Bedingung Schleife2, Bedingung Schleife3, time out vorhanden, Wer baut den Baum?, Werden noch Nachrichten erwartet?.

3 Eingabeoperationen

Anfrage, Antwort, Aufforderung, besetzt, Fehler, fertig, Nachricht, Stillegung, Reservierung, Stornierung, time out, Weckruf, Zusage.

4 Ausgabefunktionen

Anfrage, Antwort, Aufforderung, besetzt, Fehler, fertig, Nachricht, Stillegung, Reservierung, Stornierung, time out Einplanung, Weckruf, Zusage.

5 Definition der Benutzermaschine

TYPES

```
(* Wald für die Bearbeitung eines GS *)
Arbeitswald = Set of Arbeitsbaum

(* Repräsentation einer Guarded Region *)
Ursprungswald = RECORD
    UWSSet of Ursprungsbaum
    time : INTEGER (* für evtl. time out - Ausgang *)
    toadr : ADDRESS (* Fortsetzadresse bei time out *)
ENDRECORD

(* Datenstruktur für die Bearbeitung eines Guards *)
Arbeitsbaum = RECORD

    (* Guardnr. des bearbeitenden Baumes *)
    gnr : INTEGER

    (* Menge der noch zu überprüfenden Kanten *)
    L : Laub

    (* Menge der bereitsüberprüften Kanten *)
    S : Stamm

    (* Anhand von PH und PB wird die zentrale
    Fehlerbedingung überprüft *)

    (* Menge der Zielprozesse aller Kanten von S *)
    PH : Set of Verholzter Prozeß

    (* Multimenge der Zielprozesse aller Kanten von L *)
    PB : Set of Nicht Verholzter Prozeß
ENDRECORD
```

(* Repräsentation eines einzelnen Guards mit Elementen
für die Walderweiterung mit einem solchen Baum und
für die Ausführung eines Baumes *)

Ursprungsbaum = RECORD

(* Nummer des durch den Baum repräsentierten Guards *)
gnr : INTEGER

(* Repräsentation der Kommunikationsausdrücke *)
L : Laub

(* Fortsetzungsadresse *)
adr : ADDRESS

(* Vermutliches Zentrum für die Bearbeitung *)
Zentrum : Prozeßnr

(* Maxcopy und Aktcopy werden für die Walderweiterung
benötigt *)
(* Anzahl der vom verfahren erstellten Kopien des Baumes *)
Maxcopy : INTEGER

(* Anzahl der noch vorhandenen Kopien *)
Aktcopy : INTEGER

ENDRECORD

Stamm = Set of Ast (* Multimenge *)

Laub = Set of Blatt (* Multimenge *)

(* Ein Prozeß, der als Zielprozeß in einem Ast vorkommt,
wird hier registriert *)

Verholzter Prozeß = RECORD

(* Prozeßnummer des verholzten Prozesses *)

pnr : Prozeßnr

(* Nummer des beteiligten GS. Wichtig für spätere Reservierung *)

gsnr : INTEGER

(* Nummer des beteiligten Guards. Wichtig für
die Ausführungsaufforderung *)

gnr : INTEGER

ENDRECORD

(* Alle Prozesse, die als Zielprozeß in einem Blatt vorkommen,
werden hier registriert *)

Nicht verholzter Prozeß = RECORD

pnr : Prozeßnr

mcnt: INTEGER (* Multimegenzähler *)

ENDRECORD

(* Kommunikationskante, deren Ausführbarkeit bereits
nachgewiesen wurde *)

Ast = RECORD

(* Guardnr. des Startprozesses *)

Startind: INTEGER

(* Kommunikationskante *)

k : Kante

(* Guardnr. des Zielprozesses *)

Zielind : INTEGER

ENDRECORD

(* Kommunikationskante, deren Ausführbarkeit noch
nicht nachgewiesen wurde *)

Blatt = RECORD

(* Guardnr. des Startprozesses *)

ind : INTEGER

(* Kommunikationskante *)

k : Kante

ENDRECORD

(* Repräsentation einer Kommunikationskante *)

Kante = RECORD

(* Startprozeß der Kante *)

Start : Prozeßnr

(* Richtung des Informationsflusses

transmit oder receive *)

dir : Richtung

(* Zielprozeß der Kante *)

Ziel : Prozeßnr

ENDRECORD

(*In einer solchen Menge, wird die Erwiderung auf eine
Frage aufgebaut *)

Antwort = Set of ElemAntwort

(* Einzelelement einer Antwort *)

ElemAntwort = RECORD

(* Nummer des Guards, das die angefragte Kante enthält *)

ind : INTEGER

(* sonstige Blätter, die in diesem Guard mit der
Kante verknüpft sind *)

L : Laub

ENDRECORD

Prozeßnr= INTEGER

Richtung = (transmit, receive)

(* Die Wesentlichen Zustände des Baumverfahrens; Ausführung
ist mit "Reservierung und Ausführung" gleichzusetzen *)

Phase = (Ausbau, Ausführung)

(* Struktur der Botschaftstypen des Baumverfahrens *)

Botschaft = RECORD

 (* Empfänger der Botschaft *)

 Emp : Prozeßnr

 (* Absender der Botschaft *)

 Abs : Prozeßnr

 (* Typ der Botschaft *)

 Typ : Bot.type

 case tag3 : Bot.type of

 Weckruf, Frage : K : Kante

 Reservierung, Sornierung : gsnr : INTEGER

 Ant : RECORD

 Altmenge : Antwort

 gsnr : INTEGER

 ENDRECORD

 Nachricht : RECORD

 st : STRING

 ENDRECORD

 Ausführung : gnr : INTEGER

 Einplanung : time : INTEGER

 ENDCASE

ENDRECORD

Bot.type = (Weckruf, Frage, Reservierung, Sornierung, Fehler-
 meldung, Stillegung, Besetzmeldung, Bestätigung,
 Antwort, Ausführung, Nachricht, Fertigmeldung
 Einplanung)

ENDTYPE

DECLARATION

CONSTANT

time out Prozeß : Prozeßnr

ENDCONST

VAR

(* Anzahl der Nicht Fertigen Prozesse *)
NFPAnz : INTEGER

(* Datenstruktur, von der die Aktivitäten des Verfahrens
ausgehen *)
W : Arbeitswald

(* Grundlage der passiven Reaktionen *)
UW : Ursprungswald

(* Eigene Prozeßnummer *)
epn : Prozeßnr

(* Flags zum Verlassen bestimmter Verfahrenszustände *)
time out, time out Merker, executed, error, zuständig : BOOL

(* Merker für die aktuelle Verfahrensphase *)
Zustand : Phase

(* Menge der bereits reservierten Prozesse *)
RM : Set of Prozeßnr

(* Menge für den Aufbau der Antworten *)

AM : Antwort

(* Reservierungsanzeiger für das eigene GS *)

reserviert von : Prozeßnr

(* Wartemenge für weitere Reservierungswünsche *)

WM : Set of Prozeßnr

(* Nummer des gerade in Bearbeitung befindlichen GS *)

gsnr : INTEGER

(* Anzahl der Receive-Kanten, für die in der Ausführungsphase
noch keine Nachricht eingetroffen ist *)

OAnzahlR : INTEGER

ENDVAR

ENDDECLARATION

Interne Operationen

cmaus

```
(* Erstelle einer Arbeitskopie *)
create BK ∈ W
BK := B

UB.Maxcopy := UB.Maxcopy - 1

FORALL eA ∈ message.Ant DO

    message.Ant := message.Ant \ { eA }
    UB.Maxcopy := UB.Maxcopy + 1

    (* Erweitern der Menge der verholzten Prozesse *)
    BK.PH := BK.PH ∪ { ( message.Abs, message.Ant.gsnr, eA.ind ) }
    BK.PB := BK.PB \ { message.Abs }

    (* Verholzen der nachgefragten Kante *)
    create s ∈ BK.S
    s.Startind := l.ind
    s.k := l.k
    s.Zielind := eA.ind

    FORALL b ∈ eA.L DO

        (* Erstellen des Pendants der Kante von b *)
        k.Ziel := b.k.Start
        k.Start := b.k.Ziel
        k.nnr := b.k.nnr
        IF b.k.dir = transmit THEN
            k.dir := receive
        ELSE
            k.dir := transmit
        ENDIF
    ENDIF
END
```

```

(* Suchen nach dem Pendant im Laub von BK *)
IF ( select ll ∈ BK.L : ll.k = k ) ≠ ∅ THEN

    (* Verholzen der Kante mit dem pendant von k *)
    create ss ∈ BK.S
    ss.Startind := ll.ind
    ss.k := ll.k
    ss.Zielind := b.ind

    BK.PB := BK.PB \ { ll.k.Ziel }
    BK.L := BK.L \ { ll }
ELSE
    (* b dem Laub zufügen *)
    BK.L := BK.L ∪ { b }
ENDIF

(* Entfernen von b *)
eA.L := eA.L \ { b }
ENDFORALL

(* Baum auf Fehler untersuchen *)
IF BK.PB ∩ BK.PH ≠ ∅ THEN
    error := TRUE
    BK := B (* BK rücksetzen *)
    IF message.Ant = ∅ THEN
        (* B aus W entfernen *)
        W := W \ { B }
    ENDIF
ELSE
    IF message.Ant ≠ ∅ THEN
        BK.copied := TRUE
        W := W ∪ { BK }
        UB.Aktcopy := UB.Aktcopy + 1
    ELSE
        B := BK
        error := FALSE
    ENDIF
ENDIF
ENDFORALL

RETURN
END

```

Ermittlung von OAnzahlR

```
(* Setzen des Zählers für die Bearbeitung der erwarteten Nachrichten *)
OAnzahlR := 0

FORALL l ∈ UB.L : (l.k.dir = receive) do
    OAnzahlR := OAnzahlR + 1
ENDFORALL

RETURN
END
```

Erweiterung der Alternativenmenge (l : Blatt, UB : Ursprungsbaum,
Altmenge : Antwort)

```
(* Es wurde ein Guard (UB) gefunden, in dessen Laub die Kante
einer Anfrage in einem Blatt (l) vorkommt.
Die Menge der Alternativen einer Antwort AM wird deshalb um
UB erweitert. *)

(* erzeuge Element der Alternativenmenge *)
create (eA ∈ AM)

(* Guardnr der Alternative setzen *)
eA.ind := UB.gnr

(* l wird nicht in da Laub der elementaren Antwort eA aufgenommen *)
eA.L := UB.L \ { l }

RETURN
END
```

Fehler aufgetreten

```
(* Das Fehlerflag wird in verschiedenen Situationen gesetzt
- die nachgefragte Kante war beim Zielprozeß nicht vorhanden
- Der Prozeß wollte ein falsches GS reservieren
- ein Reservierungswunsch wurde endgültig abgelehnt *)
error := TRUE

RETURN
END
```

Fertigbearbeitung

```
(* Bearbeitung des Zählers für die Kontrolle der Fertigmeldungen  
Die Anzahl der nicht fertigen Prozesse wird um 1 verringert *)  
NFPAnz := NFPAnz - 1
```

RETURN

END

Fertigmeldung überwachen

```
(* Aktualisierung des Zählers für die Bearbeitung der Fertigmeldung *)  
NFPAnz := NFPAnz + 1
```

RETURN

END

GS von Wartemenge reservieren

```
select pnr ∈ WM  
reserviert von := pnr  
WM := WM \ { pnr }
```

RETURN

END

init aktive Ausführung

```
(* Auswahl des eigenen guards *)  
select UB ∈ UW : UB.gnr = B.gnr
```

Ermittlung von OAnzahlR

RETURN

END

init Anfragenbearbeitung

(* Löschen der Alternativenmenge AM, in der die auf die gestellte
Frage zurückzugebende Antwort aufgebaut wird *)

AM := 0

RETURN

END

init Baumaufbau

(* Setzen der Flags für die Baumbearbeitung *)

zuständig := TRUE

error := FALSE

Zustand := Aufbau

RETURN

END

init execute

(* Initialisierung der Reservierungs- und Ausführungsphase *)

Zustand := Reservierung

(* Vorbesetzung der Menge der bereits reservierten Prozesse *)

RM := \emptyset

(* Initialisierung des Kontrollzählers für die Ausführungsphase *)

NFPAnz := 0

RETURN

END

init Hauptprogramm

(* In der Initialisierung des Hauptprogrammes werden die Datenbereiche für die einzelnen Arbeitsschritte und deren korrekten Abfolge vorbelegt.

Dies sind:

- Errichtung des Arbeitswaldes für die Aufbauphase
- Initialisierung der Flags für die korrekte Reihenfolge
- Vorbesetzen der Datenstrukturen für Reservierungen
- Vorbereiten von Walderweiterungen *)

(* Erzeugen eines Arbeitswaldes aus dem Ursprungswald *)
FORALL UB \in UW DO

 (* Neuen Arbeitsbaum anlegen *)
 create (B \in W)
 B.gnr := UB.gnr
 B.L := UB.L
 B.S := \emptyset

 (* Initialisierung der Mengen B.PH und B.PB für die
 Überprüfung der zentralen Fehlerbedingung *)
 B.PH := { epn } ;besser \emptyset
 B.PB := \emptyset
 FORALL l \in B.L DO
 B.PB := B.PB \cup { l.k.Ziel }
 ENDFORALL

 (* Initialisierung der Steuergrößen für die Walderweiterung
 UB.Maxcopy := 1 (* Es wurde bisher nur ein Baum mit
 Basis UB in W aufgenommen *)
 UB.Aktcopy := 1 (* Aktuell existiert nur ein Baum mit
 Basis UB in W
 ENDFORALL

(* Initialisierung der Schleifenbedingung1 *)
executed := FALSE
time out := FALSE

(* Initialisierung der time out Bearbeitung *)
time out Merker := FALSE

(* Initialisierung des Wartebereiches für Reservierungen *)
reserviert von := NIL
WM := \emptyset

 RETURN
END

init passive Ausführung

(* Auswahl des richtigen guards *)
select UB ∈ UW : UB.gnr = message.gnr

Ermittlung von OAnzahlR

RETURN

END

Nachrichtенbearbeitung

(* Bearbeitung des Zählers für die Kontrolle der erwarteten
Nachrichten. Die Anzahl der offenen receive-Kanten wird
um 1 verringert *)
OAnzahlR := OAnzahlR - 1

RETURN

END

nicht zuständig

(* Der Prozeß ist selbst nicht Zentrum des Baumaufbaus *)
zuständig := FALSE

RETURN

END

Prozeß aus Wartemenge entfernen (pnr : Prozeßnr)

(* Dies wird ausgeführt, wenn von pnr ein Störnierungswunsch
eintraf *)
WM := WM \ {pnr}

RETURN

END

Prozeß in Wartemenge einfügen (pnr:Prozeßnr)

(* Eintrag des Zentrums pnr in die Wartemenge WM, da seinem
Reservierungswunsch nicht stattgegeben werden konnte *)
WM := WM ∪ { pnr }

RETURN

END

Reservierung setzen (pnr : Prozeßnr)

(* Reservierung des eigenen GS vom Zentrum pnr *)
reserviert von := pnr

RETURN

END

time out Bearbeitung

(* Der time out wird gesetzt, wenn keine Reservierung
für das Guard vorliegt. *)

IF reserviert von = NIL THEN

time out := TRUE

ENDIF

RETURN

END.

time out merken

(* Ein time out wird zunächst nur registriert, wenn

- eine Reservierung für das eigene GS vorliegt

- sich der Prozeß in der Reservierungsphase befindet *)

time out Merker := TRUE

RETURN

END

timeoutsetzen

(* Ein eventuell registrierter time out darf zuschlagen, wenn

- Aufgrung einer Stornierung das eigene GS nicht mehr belegt ist

- die Reservierungsphase aufgrund eines Fehlers verlassen

wurde und das eigene GS nicht reserviert ist *)

time out := time out Merker

RETURN

END

Walderweiterung

(* Erweiterung des Arbeitswaldes als Reaktion auf einen Weckruf.
Ein Baum qualifizierter Baum UB wird erneut in den Arbeitsbaum
aufgenommen, wenn
- keine Kopie mehr in W vorliegt (UB.Aktcopy = 0)
- mehr Kopien erstellt wurden, als noch in W vorhanden sind
(UB.Maxcopy > UB.Aktcopy) *)

FORALL UB \in UW : (select l \in UB.L : message.k = l.k) \neq 0
AND (UB.Maxcopy > UB.Aktcopy) OR UB:Aktcopy = 0 DO

(* Neuen Arbeitsbaum anlegen *)

create (B \in W)

B.gnr := UB.gnr

B.L := UB.L

B.S := \emptyset

(* Initialisierung der Mengen B.PH und B.PB für die
Überprüfung der zentralen Fehlerbedingung *)

B.PH := { epn }

B.PB := \emptyset

FORALL l \in B:L DO

B.PB := B.PB \cup { l.k.Ziel }

ENDFORALL

(* Initialisierung der Steuergrößen für die Walderweiterung

UB.Maxcopy := 1 (* Es wurde bisher nur ein Baum mir
Basis UB in W aufgenommen *)

UB.Aktcopy := 1 (* Aktuell existiert nur ein Baum mit
Basis UB in W *)

ENDFORALL

RETURN

END

Erzeuge Antwort

```
Message := bereitstellen ( message.Abs, Antwort)
Message.Abs := epn
Message.Emp := message.abs
Message.Typ := Antwort
Message.Antwort.Altmenge := Altmenge
Message.Antwort.gsnr := gsnr
RETURN
```

END

Erzeuge Aufforderung

```
Message := bereitstellen ( HP.pnr, Aufforderung)
Message.Abs := epn
Message.Emp := HP.pnr
Message.Typ := Aufforderung
Message.gnr := HP.gnr
RETURN
```

END

Erzeuge Besetztmeldung

```
Message := bereitstellen ( message.Abs, besetzt)
Message.Abs := epn
Message.Emp := message.Abs
Message.Typ := besetzt
RETURN
```

END

Erzeuge Bestätigung

```
Message := bereitstellen ( reserviert von, Zusage )
Message.Abs := epn
Message.Emp := reserviert von
Message.Typ := Zusage
RETURN
```

END

Erzeuge Fehlermeldung (pnr: Prozeßnr)

```
Message := bereitstellen ( message.Abs, Fehlermeldung )
Message.Abs := epn
Message.Emp := pnr
Message.Type := Fehlermeldung
RETURN
```

END

Erzeuge Fertigmeldung

```
Message := bereitstellen ( Zentrum, Fertig )
Message.Abs := epn
Message.Emp := Zentrum (* Zentrum muß bei Eingang der
                        Aufforderung gemerkt werden *)
Message.Type := Fertig
RETURN
```

END

Erzeuge Frage

```
Message := bereitstellen ( l.k.Ziel, Frage )
Message.Abs := epn
Message.Emp := l.k.Ziel
Message.Type := Frage
Message.k := l.k
RETURN
```

END

Erzeuge Nachricht

```
Message := bereitstellen ( l.k.Ziel, Nachricht )
Message.Abs := epn
Message.Emp := l.k.Ziel
Message.Type := Nachricht
Message.nnr := l.k.nnr
select N ∈ NP : N.nnr := l.k.nnr (*Nachrichtenpool*)
Message.st := N.st
RETURN
```

END

Erzeuge Reservierung

```
Message := bereitstellen ( P.pnr, Reservierung)
Message.Abs := epn
Message.Emp := P.pnr
Message.Typ := Reservierung
Message.gsnr := P.gsnr
RETURN
```

END

Erzeuge Stillelegung

```
Message := bereitstellen ( message.Abs, Stillelegung)
Message.Abs := epn
Message.Emp := message.Abs
Message.Typ := Stillelegung
RETURN
```

END

Erzeuge Stornierung

```
Message := bereitstellen ( P.pnr, Stornierung)
Message.Abs := epn
Message.Emp := P.pnr
Message.Typ := Stornierung
Message.gsnr := P.gsnr
RETURN
```

END

Erzeuge time out - Einplanung

```
Message := bereitstellen ( time out Prozeß, Einplanung)
Message.Abs := epn
Message.Emp := time out Prozeß
Message.Typ := Einplanung
Message.time:= UW.time
RETURN
```

END

Erzeuge Weckruf

```
Message := bereitstellen ( Zentrum, Weckruf )
Message.Abs := epn
Message.Emp := Zentrum
Message.Typ := Weckruf
select l ∈ UB.L : Zentrum = l.k.Ziel
Message.k := l.k
RETURN
```

END

Interne Funktionen

AlleFertigmeldungenerhalten ?

RETURN NFPAnz = 0

END

Auswahl Antwortbaum

(* Das ausgewählte Element wird zusätzlich aus der Menge entfernt *)
IF select (select B \in WH : message.k = B.k) $\neq \emptyset$ THEN

WH := WH \setminus { B }

ENDIF

RETURN B

END

Auswahl Arbeitsbaum

(* Das ausgewählte Element wird zusätzlich aus der Menge entfernt *)
IF select B \in W $\neq \emptyset$ THEN

W := W \setminus { B }

ENDIF

RETURN B

END

Wer baut den Baum? (pnr : Prozeßnr, UB : Ursprungsbaum)

(* Diese Funktion betimmt das zuständige Zentrum
für den Baumaufbau *)

RETURN MAX (pnr, UB.Zentrum)

END

Auswahl Arbeitsblatt

```
(* Das ausgewählte Element wird zusätzlich aus der Menge entfernt *)
IF select l ∈ B.L ≠ ∅ THEN

    B.L := B.L \ { l }

ENDIF
RETURN l
END
```

Auswahl zur Reservierung

```
(* Die Auswahl muß gemäß der vorgeschriebenen Ordnung erfolgen *)
IF ( select ph ∈ B.PH : ph.pnr ≠ epn ) ≠ ∅ THEN

    B.PH := B.PH \ { ph }
    RM := RM ∪ { ph }

ENDIF
RETURN ph
END
```

Auswahl zur Stornierung

```
(* Das ausgewählte Element wird zusätzlich aus der Menge entfernt *)
IF ( select P ∈ RM ) ≠ ∅ THEN

    RM := RM \ { P }

ENDIF
RETURN P
END
```

Werden noch Nachrichten erwartet ?

```
RETURN OAnzahlR = 0

END
```


Auswahl einer Sendekante

```
(* Das ausgewählte Element wird zusätzlich aus der Menge entfernt *)
IF ( select l ∈ B.L : l.k.dir = transmit ) ≠ ∅ THEN

    UB.L := UB.L \ { l }

ENDIF
RETURN l
END
```

Auswahl für Absage

```
(* Das ausgewählte Element wird zusätzlich aus der Menge entfernt *)
IF ( select ph ∈ WM ) ≠ ∅ THEN

    WM := WM \ { ph }

ENDIF
RETURN ph
END
```

Bedingung für Ausführung

```
(* Sicherstellen, daß das GS tatsächlich
noch nicht ausgeführt wurde *)

RETURN executed
END
```

Auswahl für Aufforderung

```
(* Das ausgewählte Element wird zusätzlich aus der Menge entfernt *)
IF ( select ph ∈ RM ) ≠ ∅ THEN

    RM := RM \ { ph }

ENDIF
RETURN ph
END
```

Bedingung Schleife1

(* Die äußere Schleife des Hauptprogramms, die auf dem gesamten Arbeitswald operiert, wird nur verlassen, wenn ein GS aufgeführt wurde, oder ein time out den Abbruch erzwingt *)

RETURN executed OR time out

END

Bedingung Schleife2

(* Die Schleife des Hauptprogrammes, welche die einzelnen Arbeitsbäume abarbeitet, wird im Vergleich zu Schleife1 auch dann verlassen, wenn

- ein Fehler beim Baumaufbau aufgetreten ist
- der Prozeß für den Aufbau des Baumes nicht zuständig ist *)

RETURN executed OR error OR NOT zuständig OR time out

END

Bedingung Schleife3

(* In dieser Schleife befindet sich die Reservierungs- und Ausführungsphase des Verfahrens. *)

RETURN executed OR error OR time out

END

Bedingung1 für Antwort

(* Ist die angefragte Kante vorhanden ? *)

select UB \in UW : select l \in UB.L : l.k = message.k

RETURN UB = NIL

END

Bedingung2 für Antwort

(* Ist die Alternativenmenge der Antwort leer *)

RETURN Altmenge = \emptyset

END

Bedingung1 für Reservierung

(* Ist das GS unreserviert ? *)

RETURN reserviert von = NIL

END

Bedingung2 für Reservierung

(* Wünscht der Prozeß auch das GS zu reservieren, das momentan
bearbeitet wird ? *)

RETURN message.gsnr \neq gsnr

END

Bedingung1 für Stornierung

(* Wird GS vom stornierenden Prozeßbelegt ? *)

RETURN reserviert von = message.Abs

END

Bedingung2 für Stornierung

(* Ist die Wartemenge leer ? *)

RETURN WM = 0

END

Bedingung3 für Stornierung

RETURN Zustand = Ausführung

END

time out vorhanden?

(* Diese Bedingung prüft, ob das GS einen
time out - Ausgang besitzt *)

RETURN UB.time out \neq 0

END

Literaturverzeichnis

- /ACS861/ ISO: "Draft International Standard 8649/2. Information Processing Systems - OSI - Service Definition for Common Application Service Elements - Part 1: Association Control", 1986
- /ACS862/ ISO: "Draft International Standard 8650/2. Information Processing Systems - OSI - Protocol Specification for Common Application Service Elements - Part 2: Association Control", 1986
- /ADA83/ Reference Manual for Ada Programming Language. U.S. Dep. Defense, 1983
- /AFHH83/ C. Andres, A. Fleischmann, U. Hillmer, P. Holleczeck, R. Kummer: "Pflichtenheft für die Basis-Dienste des Erlanger DFN-Anschlusses", RRZE IAB Nr. 193, Erlangen 1983
- /AFHT85/ C. Andres, A. Fleischmann, P. Holleczeck, M. Trautner, W. Mühlbauer: "Testen von verteilten Programmen zur Prozeßautomatisierung", Angewandte Informatik, 1985
- /AKS85/ Siemens, AKS-Team: "Allgemeine Kommunikationsschnittstelle", Karlsruhe, Erlangen 1985
- /ALS87/ ISO: "Draft Proposal Standard 9545. Information Processing Systems - OSI - Application Layer Structure", 1987
- /ANKN83/ T. Anderson, J. C. Knight: "A Framework for Software Fault Tolerance in Real-Time Systems", IEEE Transactions on Software Engineering Vol. 9, No. 3, 1983
- /BAAC87/ Christian Baacke: "Ein Betriebssystemkonzept für Verteilte Anwendungen im Rahmen des ISO/OSI-Referenzmodells", Diplomarbeit am IMMD IV, Erlangen 1987
- /BATH86/ Peter Bathelt: "Der Dominoeffekt in der Fehlerbehandlung von Prozess-Systemen", Arbeitsbericht des IMMD, Erlangen 1986

- /BEHG87/ P. A. Bernstein, V. Hadzilacos, N. Goodman: "Concurrency Control and Recovery in Database Systems", Addison-Wesley P. C., Reading 1987
- /BEST81/ E. Best, B. Randell: "Formal Model of Atomicity in Asynchronous Systems", Acta Informatica 16, Springer Berlin 1986
- /BEVE87/ M. Bever, A. Fleischmann: "Ein Konfigurationskonzept für die Anwendungsebene des ISO Referenzmodells für offene (Büro-) Systeme" Beitrag zur GI - 17. Jahrestagung "Computerintegrierter Arbeitsplatz im Büro", Informatik-Fachberichte, München 1987
- /BFPA87/ M. Bever, M. Feldhoffer, S. Pappe: "OSI Services for Transaction Processing", Beitrag zum "TP Workshop", Asilomar 1987
- /BLAA72/ G. A. Blaauw: "Computer-Architektur", Elektronische Rechenanlagen, Vol. 14, Heft 4, 1972
- /BUCK83/ G. N. Buckley, A. Silberschatz: "An Effective Implementation for the Input-Output Construct of CSP", ACM TOPLAS Vol. 5, No. 2, 1983
- /CCII83/ Deutsche Bundespost: "Rahmenwerte für TELETEX Endgeräte, Ttx-Transportprotokoll T.70", FTZ, Darmstadt, Mai 1983
- /CCR851/ ISO: "Draft Proposal 9804/2. Information Processing Systems - OSI - Definition of Common Application Service Elements - Part 1: Commitment, Concurrency and Recovery", 1985
- /CCR852/ ISO: "Draft International Standard 9805. Information Processing Systems - OSI - Specification of Protocols for Common Application Service Elements - Part 2: Commitment, Concurrency and Recovery", 1985
- /CERI84/ Ceri S., Pelagatti G.: "Distributed databases, principles and systems", McGraw-Hill, New York, 1984
- /DIJK71/ E. W. Dijkstra: "Hierarchical Ordering of Sequential Processes", Acta Informatica 1, 1971
- /DIJK75/ E. W. Dijkstra: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", Communications of the ACM Vol. 18, No. 8, 1975

- /DIN81/ Programmiersprache PEARL. DIN 66253, 1981

- /ECMA72/ European Computer Manufacturers Association: "Standard ECMA-72 Transport Protocol", 1982

- /ECMA75/ European Computer Manufacturers Association: "Standard ECMA-75 Session Protocol", 1982

- /ECMA85/ European Computer Manufacturers Association: "OSI Distributed Interactive Processing Environment (DIPE)", März 1985

- /FHKK83/ A. Fleischmann, P. Holleczeck, G. Klebes, R. Kummer: "Synchronisation und Kommunikation verteilter Automatisierungsprogramme", Angewandte Informatik, 1983

- /FISC85/ M. J. Fischer, N. A. Lynch, M. S Paterson: "Impossibility of Distributed Consensus with One Faulty Process", Journal of the ACM Vol. 32, No. 2, 1985

- /FLEI84/ Albert Fleischmann: "Ein Konzept zur Darstellung und Realisierung von verteilten Prozeßautomatisierungssystemen", Mitteilungsblatt des RRZE, Erlangen 1984

- /FLEI85/ Albert Fleischmann: Private Mitteilungen, 1985

- /GRAY79/ J. N. Gray: "Notes on Data Base Operating Systems" in "Operating Systems - An Advanced Course", Lecture Notes in Computer Science, Springer 1979.

- /HADD77/ B. K. Haddon: "Nested Monitor Calls", SIGOPS, Vol. 11, No.4, 1977

- /HANS77/ Per Brinch Hansen: "The Architecture of Concurrent Programs", 1977

- /HEIL85/ Elfriede Heilmeyer: "Erfassung von Telefongesprächsdaten mit einem verteilten System von Mikrorechnern", Diplomarbeit am IMMD IV, Erlangen 1985

- /HERB85/ Harald Herberth: "Zeitüberwachung in einem Betriebssystem für verteilte Systeme als Grundlage für Maßnahmen zur Fehlerbehandlung", Studienarbeit am IMMD IV, Erlangen 1985

- /HESS88/ Gabriele Heß: "Ein Vergleich der verteilten Anwendungen am RRZE mit internationalen Ansätzen", Diplomarbeit am IMMD IV, Erlangen 1988
- /HOAR85/ C. A. R. Hoare: "Communicating Sequential Processes", Prentice/Hall International, Englewood Cliffs, New Jersey 1985
- /HOFM83/ Fridolin Hofmann: "Koordinierung und verteilte Systeme", Beitrag zur PEARL-Tagung, 1983
- /HOFM84/ Fridolin Hofmann: "Betriebssysteme: Grundkonzepte und Modellvorstellungen", B. G. Teubner Stuttgart 1984
- /HOFM87/ Fridolin Hofmann: "Betriebssystemaspekte verteilter Systeme", Vorlesung an der Universität Erlangen-Nürnberg, Wintersemester 1987/88
- /HOLL87/ Peter Holleczeck: "Verteilte Anwendungen im DFN - Durchführbarkeitsstudie und prototypische Verwirklichung", Forschungsantrag im Rahmen des Deutschen Forschungsnetzes, Erlangen 1987
- /IIDS84/ E. Fahr, H.-G. Barbian (Dornier System), U. Bügel, G. Bonn (IITB): "Benutzerhilfsmittel für PEARL im Einsatz auf Mehrrechner-Systemen", Tagungsband zur PEARL-Tagung, 1984
- /ISOB84/ ISO: "International Standard 7498. Information Processing Systems - OSI - Basic Reference Model", 1984
- /ISOC85/ ISO: "Definition of Common Application Service Elements: Basic Kernel Subset", Draft Proposal, 1985
- /ISOS84/ ISO: "Basic Connection Oriented Session Protocol Specification", 1984
- /ISOTP1/ ISO: "Information Processing Systems - OSI - Distributed Transaction Processing - Part 1: Model", 1987
- /ISOTP2/ ISO: "Information Processing Systems - OSI - Distributed Transaction Processing - Part 2: Transaction Processing Service Definition", 1987
- /ISOTP3/ ISO: "Information Processing Systems - OSI - Distributed Transaction Processing - Part 3: Transaction Processing Protocol Specification", 1987

- /ISOT83/ ISO: "Connection Oriented Transport Protocol Specification", 1983
- /KEBR81/ H. Kerner, G. Bruckner: "Rechnernetzwerke - Systeme, Protokolle und das ISO-Architekturmodell", Springer Wien 1981
- /KIMM82/ K. H. Kimm: "Approaches to Mechanization of the Conversation Scheme Based on Monitors", IEEE Transactions on Software Engineering Vol. 8, No. 3, 1982
- /KRMA85/ J. Kramer, J. Magee: "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering Vol. 11, No. 4, 1985
- /KUMM83/ Robert Kummer: "Entwicklung von Betriebssystembausteinen für Guarded Regions und Botschaftsoperationen im Rahmen eines Realzeitprogrammiersystems", Diplomarbeit am IMMD IV, Erlangen 1983
- /LAM978/ Leslie Lamport: "Time, Clocks and the Ordering of Events in a Distributed System", Communications of the ACM Vol. 21, No. 7, 1978
- /LAMP78/ Leslie Lamport: "The Implementation of Reliable Distributed Multiprocess Systems", Computer Networks 2, 1978
- /LAMP82/ L. Lamport, R. Shostak, M. Pease: "The Byzantine Generals Problem", ACM TOPLAS Vol. 4, No. 3, 1982
- /LAMP84/ Leslie Lamport: "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems", ACM TOPLAS Vol. 6, No. 2, 1984
- /LAUE78/ H. C. Lauer, R. M. Needham: "On the Duality of Operating System Structures", Proceedings Second International Symposium on Operating Systems, IRIA, Oct. 1978, reprinted in Operating Systems Review, 13, 2 April 1979
- /LISK83/ B. Liskov, R. Scheifler: "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM Toplas Vol. 5, No. 3, 1983
- /MAP21/ General Motors Corporation: "MAP 2.1", General Motors Technical Center, Warren, Michigan

- /MOHA83/ C. Mohan, R. Strong, S. Finkelstein: "Method of Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors", Proceedings of the 2. ACM Symposium on Principles of Distributed Computing, 1983
- /MOHA86/ C. Mohan, B. Lindsay, R. Obermarck: "Transaction Management in the R* Distributed Database Management System", ACM Transactions on Database Systems 11, Nr. 4, 1986
- /OTSS87/ International Business Machines Corporation (IBM): "Open Systems Transport und Session Support (OTSS)", Release 2.0, Stuttgart 1987
- /PRE861/ ISO: "Draft International Standard 8822. Information Processing Systems - OSI - Connection oriented presentation service definition", 1986
- /PRE862/ ISO: "Draft International Standard 8822. Information Processing Systems - OSI - Connection oriented presentation protocol specification", 1986
- /RAND78/ B. Randell, P. A. Lee, P. C. Treleaven: "Reliability Issues in Computing System Design", Computing Surveys Vol. 10, Nr. 2, 1978
- /REUT81/ Andreas Reuter: "Fehlerbehandlung in Datenbanksystemen", Carl Hanser Verlag, München, 1981
- /ROS871/ ISO: "Draft International Standard 9072/1. Information Processing Systems - Text Communication - Remote Operations - Part 1: Model, Notation and Service Definition", 1987
- /ROS872/ ISO: "Draft International Standard 9072/2. Information Processing Systems - Text Communication - Remote Operations - Part 2: Protocol Specification", 1987
- /RZEH87/ Helmut Rzehak: "Die Abwicklung von Realzeit-Aufträgen in MAP-Netzen", Beitrag zum Workshop über Realzeitsysteme "PEARL 87", München 1987
- /SHRI81/ Santosh K. Shrivastava: "Structuring Distributed Systems for Recoverability and Crash Resistance", IEEE Transactions on Software Engineering Vol. 7, No. 4, 1981

- /SHRI85/ Santosh K. Shrivastava: "Reliable Computer Systems - Collected Papers of the Newcastle Reliability Project", Springer Berlin 1985
- /UEBE87/ Erich Übelmesser: "Eine implementationsnahe Spezifikation eines Botschaftenprotokolls für verteilte Systeme", Diplomarbeit am IMMD IV, Erlangen 1987
- /WEBE83/ Karl Weber: "Modellierung von Fehlverhalten mit Berücksichtigung paralleler Abläufe" Arbeitsbericht des IMMD, Erlangen 1983
- /WEBE85/ Karl Weber: Private Mitteilungen, 1985
- /WETT84/ Horst Wettstein: "Architektur von Betriebssystemen", Hanser, München 1984
- /WILL84/ R. Williamson, E. Horowitz: "Concurrent Communication and Synchronization Mechanisms", Software Practice and Experience, Vol. 14 (2), 1984