Erhöhung der Nebenläufigkeit in automatisch entworfenen digitalen Systemen

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
von der Fakultät für Informatik
der Universität Karlsruhe (Technische Hochschule)

genehmigte
Dissertation

von
Taisy Silva Weber
aus Natal, Brasilien

Tag der mündlichen Prüfung: 20. Mai 1986

Erster Gutachter: Prof. Dr.-Ing. Detlef Schmid

Zweiter Gutachter: Prof. Dr.-Ing. H. M. Lipp

Danksagung

Herrn Professor Dr.-Ing. D. Schmid bin ich für seine weitreichende Unterstützung, für die Schaffung einer förderlichen Arbeitsumgebung sowie für seine wertvollen Anregungen sehr zu Dank verpflichtet.

Herrn Prof.Dr.-Ing. H.M. Lipp danke ich für die Übernahme des Korreferats, sowie für die nützliche Hinweise, die zur Vervollständigung der Arbeit beitrugen.

Bei meinen Kollegen des Instituts für Informatik IV bedanke ich mich für die gute Zusammenarbeit. Herr Dr. R. Weber, Herr Dr. Rosenstiel und Herr Dr. Camposano ermöglichten dank ihrer Diskussionsbereitschaft und ihrer hilfsreichen Hinweise die Erstellung der BABEL-Sprache und den Anschluß an das CADDY-System. Bei der Universidade Federal do Rio Grande do Sul (UFRGS-Brasilien) und dem Deutschen Akademischen Austauschdienst (DAAD) bedanke ich mich für die finanzielle Unterstützung.

Sehr herzlich danke ich Frau Möller, Frau Klein, sowie den Herren Dr. Klaus Echtle und Dipl.-Inform. Michael Marhöfer für ihre stätige Hilfsbereitschaft. Ebenso bedanke ich mich sehr herzlich bei meinem Mann Raul für seine Geduld und liebevolle Sorge.

Gedruckt mit Unterstützung des Deutschen Akademischen Austauschdienstes

Zusammenfassung

Nebenläufigkeit (die gleichzeitige Aktivität mehrerer Operationen in einem digitalen System) ist eine Möglichkeit, ohne Anstieg der Technologiekosten hohe Arbeitsgeschwindigkeiten zu erzielen. Die vorliegende Arbeit soll einen Beitrag zur Lösung des Problems der Erhöhung des Nebenläufigkeitsgrades in komplexeren digitalen Systemen durch automatischen Entwurf leisten.

Ausgangspunkt dieser Arbeit ist die bisher unbefriedigende Situation bei der Beschreibung und automatischen Synthese nebenläufiger Schaltungen im Vergleich zur verbreiteten Ausnutzung von Nebenläufigkeit in den Bereichen der Rechnerarchitektur, Betriebssysteme und Programmiersprachen. Im allgemeinen wird Nebenläufigkeit erst in den letzten Phasen des automatischen Entwurfsprozesses einbezogen, was die Komplexität des Syntheseverfahrens beträchtlich erhöht. Dagegen verfolgt diese Arbeit die Idee, daß ein hoher Nebenläufigkeitsgrad mit geringer Synthesekomplexität erreicht wird, wenn Nebenläufigkeit schon in den frühesten Entwurfsphasen, nämlich der Problemanalyse und der Verhaltensbeschreibung, berücksichtigt wird.

Zur Beschreibung des Verhaltens eines Systems wird die Sprache BABEL (Beschreibungssprache für nebenläufige digitale Schaltungen) eingeführt, die eine hierarchische und strukturierte Beschreibung der Nebenläufigkeit unterstützt. Mit dem Ziel, die Zunahme der Komplexität des von der BABEL-Beschreibung ausgehenden automatischen Syntheseprozesses gering zu halten, wurden Verfahren zur Realisierung von Nebenläufigkeit durch mehrere kooperative Steuerwerke und Verfahren zur Erhöhung des Parallelitätsgrades durch Kompaktierung von Zuständen entwickelt. Um den Entwurfsprozeß zu vervollständigen, wurden die Sprache und das Syntheseverfahren an das automatische Entwurfssystem CADDY (Carlsruhe Digital Design System) angeschlossen. Die am Ende des automatischen Entwurfsvorgangs erzeugte Schaltung nutzt die im beschriebenen System vorliegende Nebenläufigkeit dann voll aus.

Inhaltsverzeichnis

Verwendete Symbole

$(x_1, \ldots, x_n)$	n-Tupel
$(x_1, \ldots, x_m)$	Menge der Elemente x_1 bis x_m
$(x \mid P)$	Menge aller x, für die P gilt
$\forall\ x : P$	Für alle x gilt P
$\forall\ x\ .\ P_1 : P_2$	Für alle x so daß P_1 gilt P_2
$\exists\ x : P$	Es gibt ein x, für das P gilt
$\exists 1\ x : P$	Es gibt genau ein x, für das P gilt
$\Rightarrow$	wenn, dann
$A \subset B$	A enthalten in B
$A \not\subset B$	A nicht in B enthalten
$A \bigcup B$	A vereinigt mit B
$A \bigcap B$	A geschnitten mit B
$\bigcup_i M_i$	Vereinigung aller Mengen M_i
$A - B$	A ohne B
$\{\ \}$	leere Menge
$\in$	Element von
$\notin$	nicht Element von
$\wedge, \vee, \neg$	und, oder, nicht
$f : A \rightarrow B$	Abbildung f der Menge A in die Menge B
$\ll$	viel kleiner
$\lvert M \rvert$	Zahl der Elemente der Menge M

Verwendete Schreibweisen

Für die Darstellung der Produktionen (Kapitel 3) und der syntaktischen Definitionen der BABEL-Grammatik (Kapitel 4) gilt die ALADIN-Notation [Kast79]:

a ::= b	Ableitung
'X'	Wortsymbole oder Sonderzeichen
[a]	Optional
a / b	Entweder a oder b
(a) *	Anzahl der Anwendung der Elemente a ist gleich oder größer als Null.
(a) +	Anzahl der Anwendung der Elemente a ist größer als Null.
(a//b)	Wiederholung mit Trennzeichen; a kommt mindestens einmal vor. Wenn es mehrere a gibt, stellt b ein Trennzeichen dar.

Verwendete Bezeichnungen

AGM	Aktion
AGMs	Menge der Aktionen eines IPM's
a_k	Anforderungsknoten
Ass	Menge der Auswahl-Steuersignale
BABEL	Beschreibungsprache für nebenläufige digitale Schaltungen
CAD	Computer aided design
CADDY	Carlsruhe Digital Design System
CAP/DSDL	Concurrent Algorithmic Programming Language/ Digital System Description Language
CC	Conflict Case
$Cs(t_i)$	Menge der Steuersignale von t_i
Cs_k	Menge der Steuersignale des Knotens k
DA	Design Automation
DSL	Digital System Specification Language
E	Menge der Kanten
E_b	Menge der bedingten Kanten
E_u	Menge der unbedingten Kanten
e_{imp}	implizite Kante einer Sequenz
fz	Folgezustand
GM	Gemeinsames Mittel
HDL	Hardware Description Language
IPM	Interprozeßmodul
IPMs	Menge der Interprozeßmoduln eines Systems
ISPS	Instruction Set Processor Specification
$id(l_i)$	Kennzeichen einer Zeile l_i einer U_{tab}
$id(t_i)$	Kennzeichen (identifier) von t_i
K1,K2,K3,K4	Kompaktierungsregeln 1, 2, 3 bzw. 4
KfA	Menge der konfliktfreien Paare von Aktionen
Kss	Menge der Aktivierungssignale
k_i	Knoten i
Lss	Menge der Laden-Steuersignale
l_i	Zeile (line) einer U_{tab}
MpA	Menge der Paare von Aktionen eines IPMs
MsA	Menge der Paare von simultanen Aktionen
O_p	Menge der Operationen
Param.	Parameter
PC	Priority Case
P_{opd}	Datenpfad
PRD	Produktion
$Pred(t_i)$	Menge der Vorgänger von t_i
Ps	Menge der Prozesse eines Systems

Sa_k	Anforderungssektion
S_{comp}	Sektion (ein Ergebnis der Komposition)
SD	Syntaktische Definition
Sk	Sektion
SLIDE	Structured Language for Interface Description and Evaluation
sM	Sequentielle Maschine
SM	Schutzmechanismus
S_{opt}	Sektion (die optimale Komposition)
ss	Steuersignal
S_t	Menge der Strukturknoten
$S_t(z_i)$	Menge der aktiven Steuersignale von z_i
$Suc(t_i)$	Menge der Nachfolger von t_i
$suc(t_i)$	Nachfolger von t_i
Sw_k	Wartesektion
T_p	Menge der Tests (Knoten) eines Verhaltensgraphen
$T_s(t_i)$	Menge der Testsignale von t_i
tP	terminaler Prozeß
transit	Überganseinheit (transition unit)
t_i	Übergangseinheit i
$type(t_i)$	Typ von t_i
U_{graph}	Übergangsgraph
U_{tab}	Übergangstabelle
$Vk(t_i)$	Menge der entsprechenden Verhaltensgraphenknoten
w	Wert eines Testsignals
w_k	Warte-Knoten
Z	Menge von Zuständen
$Zw(t_i)$	Menge der Zweige von t_i
z_i	Zustand i
α	Anfangsknoten eines Abschnitts
α_t	Eingangspunkt einer Sektion
α_*	Anfangsknoten des nächsten Abschnitts
δ	Überführungsfunktion einer sM
o	Endknoten eines Abschnitts
φ	Inzidenzabbildung eines Verhaltensgraphen
Φ	Nebenläufigkeitsgrad
$\Phi_a(a_i)$	Nebenläufigkeitsgrad eines Ablaufs a_i
$\Phi_c(c_i)$	Nebenläufigkeitsgrad eines Steuerwerks c_i
$\Phi_z(z_i)$	Nebenläufigkeitsgrad eines Zustands z_i
Π	Priorität

Einleitung

Die Ausnutzung der in digitalen Schaltungen vorhandenen Nebenläufigkeit erhöht die Geschwindigkeit eines Systems, läßt sich allerdings im allgemeinen auf Kosten einer zunehmenden Entwurfskomplexität realisieren. Die Gründe hierfür liegen hauptsächlich in der Bestimmung der nebenläufigen Elemente und der potentiellen Konflikte, die bei der Interaktion zwischen diesen Elementen auftreten können, ferner in der Erzeugung von geeigneten Mechanismen zur Vermeidung solcher Konflikte und letztlich in der Überprüfung der Korrektheit des nebenläufigen Verhaltens.

Das Hauptziel dieser Arbeit ist die Überwindung dieser Komplexitätserhöhung. Dafür wird die folgende Strategie verfolgt:

- Zerlegung des Systems während der ersten Phase des Entwurfszyklus in potentiell nebenläufige Prozesse und die Anwendung eines geeigneten Mittels zur Beschreibung dieser Prozesse auf der Verhaltensebene.

- Verfügbarkeit impliziter Mechanismen zur Interaktion zwischen den Prozessen, so daß der Entwerfer von dem aufwendigen und fehleranfälligen Entwurf der Synchronisations- und Kommunikationsschemata befreit wird.

- Anwendung einer von der Beschreibung bis zur Schaltungsstruktur gehenden automatischen Synthese, die jeden Prozeß als ein unabhängiges System behandelt und die Prozesse durch eine angepaßte Interaktionsstruktur verknüpft.

Das erste Kapitel zeigt die Phasen des Entwurfszyklus, in denen man Nebenläufigkeit behandelt muß, damit in der realisierten Schaltung ein hoher Grad von Nebenläufigkeit erreicht werden kann. Ein geeignetes Mittel zur Beschreibung von nebenläufigen Systemen wird in Kapitel 2 durch die Analyse von typischen digitalen Systemen und von verschiedenen Beschreibungsmöglichkeiten charakterisiert. Aus dieser Analyse ergibt sich das Beschreibungsmittel für nebenläufiges digitales Verhalten, welches in Kapitel 3 und 4 vorgestellt wird. Dieses Mittel verfügt über die im Gebiet der Programmiersprachen bekannten Synchronisationsprimitive und erlaubt damit die Anwendung des Verfahrens zur Überprüfung der Korrektheit, die für diese Primitive schon vorhanden sind.

Aus einer korrekten Beschreibung eines Systems wird eine entsprechende korrekte Schaltung erzeugt. Das in Kapitel 6 bis 8 vorliegende Synthesekonzept umfaßt:

- Die Synthese eines Operationswerks und eines Steuerwerks für jeden beschriebenen Prozeß. Innerhalb eines Prozesses wird Nebenläufigkeit durch Kompaktierung von Zuständen erreicht.

- Die Ergänzung der realisierten Schaltung durch einen vorher entworfenen und verifizierten Interaktionsmechanismus, welcher die in der Beschreibungsform vorhandenen Synchronisationsprimitive auf der Strukturebene umsetzt.

Kapitel 9 schließlich faßt den vorgestellten Entwurfsprozeß zusammen und gibt einen kurzen Ausblick auf weiterführende Arbeiten.

1. Nebenläufigkeit

Dieses Kapitel führt die Konzepte von Parallelität und Nebenläufigkeit in digitalen Systemen ein und zeigt die Phasen des Entwurfszyklus, in denen man Nebenläufigkeit sorgfältig behandeln muß, damit in der realisierten Schaltung ein hoher Grad von Nebenläufigkeit mit geringem Entwurfsaufwand erreicht werden kann. Zum Schluß folgt die Abgrenzung der in dieser Arbeit behandelten Themen.

1.1 *Nebenläufigkeit und Parallelität*

Die Begriffe 'Nebenläufigkeit' und 'Parallelität' finden in der Fachliteratur keine allgemein anerkannte Abgrenzung ([Sequ83], [Rem81], [Zakh84]). Oft kommen sie als Synonyme vor, noch öfter wird Nebenläufigkeit als Oberbegriff für Parallelität benutzt. Beispiele hierfür sind die gleichzeitige Aktivierung mehrerer digitaler Komponenten, die gleichzeitige Übertragung von Informationen über verschiedene Datenwege oder die gleichzeitige Durchführung unterschiedlicher Funktionen mit unabhängigen Datenbeständen. In diesem gleichzeitigen Geschehen mehrerer Ereignisse liegt die Grundidee dieser beiden Begriffe.

Hier wird Nebenläufigkeit als die gleichzeitige Aktivität mehrerer Operationen oder Mengen von Operationen gesehen, wobei unter Operationen die üblichen, mit digitalen Schaltungen realisierbaren logischen und arithmetischen Verknüpfungen zwischen Informationsträgern zu verstehen sind. Zwei oder mehrere Operationen sind dann nebenläufig, wenn die Möglichkeit besteht, daß sie gleichzeitig aktiv sein können. Falls diese nebenläufigen Operationen immer gleichzeitig aktiv sind, handelt es sich um Parallelität. Die Konzepte von Nebenläufigkeit und Parallelität als eine zeitliche Relation zwischen aktiven Operationen werden in Kapitel 3 im Rahmen des von BABEL verwendeten Verhaltensmodells definiert bzw. formalisiert.

1.2 *Ausnutzung der Nebenläufigkeit beim Entwurf*

1.2.1 *Entwurfszyklus*

Beim Entwurf digitaler Systeme können die in Tabelle 1.1 gezeigten Entwurfsphasen unterschieden werden.

PHASE	DARSTELLUNGSEBENE	FRAGESTELLUNG
Problemanalyse	Problemspezifikation	was?
Lösungsentwurf	Verhaltensbeschreibung	wie?
Synthese	Schaltungsstruktur	womit? (Komponenten)
Realisierung	Schaltung	wodurch? (Technologie)

Tabelle 1.1 - Phasen des Entwurfszyklus

Einige Autoren ([Nies83], [Rose84]) betrachten den Entwurfszyklus in den drei Ebenen Verhaltensebene, logische und physikalische Ebene. Der Verhaltensebene entsprechen die zwei oberen Phasen und der physikalischen Ebene entspricht die unterste Phase der Tabelle 1.1. Die hier angewandte Einordnung unterscheidet eindeutig die Aktivitäten und die Ergebnisse jeder Phase.

In der ersten Phase des Entwurfszyklus, der Problemanalyse, werden die Vorstellungen über ein neues digitales System konkreter entwickelt und festgehalten. Das Ergebnis dieser Phase, die Problemspezifikation, gibt die Antwort auf die Frage 'was soll gemacht werden?' und legt die Anforderungen an Leistung, Zuverlässigkeit, elektrische und mechanische Schnittstellen fest, die durch den Entwurf möglichst genau eingehalten werden müssen. Da eine vollständige Überprüfung der Korrektheit nur gegenüber einer rein formalen Spezifikation möglich ist, wäre die Anwendung eines formalen Spezifikationsmittels auf dieser Ebene ideal. Aus Mangel an solchen Mitteln wird das Problem in der Regel in einem Pflichtenheft mit Hilfe von Tabellen, Diagrammen und Graphen beschrieben und Teile der formalen Spezifikation werden in die zweite Phase verschoben.

Die zweite Phase, der Lösungsentwurf, befaßt sich mit der Frage, wie das Problem durch ein digitales System gelöst werden kann. Es gibt zur Zeit kaum maschinelle Unterstützung in dieser Phase, doch ist aufgrund der Entwicklung der künstlichen Intelligenz [Rich84] zu erwarten, daß künftig Experten-Systeme in diesem Bereich eingesetzt werden können. Das Ergebnis dieser Phase ist die Verhaltensbeschreibung des Systems. Ausgehend von

dieser Beschreibung beginnt die Implementierung, bei denen maschinelle Unterstützung insbesondere in der letzten Phase verbreiteten Einsatz findet ([AlPe81], [Dire81], [DSST83], [HaPa82], [Newt81], [SGB83], [Trim81]).

In der ersten Implementierungsphase, hier Synthese genannt, werden die digitalen Komponenten und ein Verbindungsmuster gesucht, die das Verhalten des Systems auf der strukturellen Ebene realisieren. Das Ergebnis dieser Phase, die Schaltungsstruktur, kann in verschiedenen Formen dargestellt werden, z.B. in Blockdiagrammen, Register-Transfer-Sprachen, Logikschaltplänen und anderen. Die letzte Phase, die Realisierung, befaßt sich mit der Technologie und liefert die technologische Umsetzung.

Bei der Betrachtung von Nebenläufigkeit sind die Phasen des Lösungsentwurfs und der Synthese von größter Bedeutung. In diesen zwei Phasen sind die wichtigsten Entscheidungen zu treffen, welche die Nebenläufigkeit eines Systems bestimmen. Weil der Entwerfer die Eigenschaften des zu realisierenden Systems am bestens kennt, ist der durch einen effizienten Algorithmus und eine geeignete Beschreibung erzielte Nebenläufigkeitsgrad in der Regel größer als der Grad, den ein Syntheseverfahren allein erreichen kann. Deshalb betont diese Arbeit die Beschreibung von Nebenläufigkeit. In Kapitel 2 bis 4 wird die Beschreibung von nebenläufigem Verhalten behandelt und danach (Kapitel 5 bis 8) die Synthese von Nebenläufigkeit.

1.2.2 *Nebenläufigkeit beim Lösungsentwurf*

Beim heutigen Stand der Technik liegt der Entwurf des Algorithmus im Bereich der Entwerfertätigkeiten. Der Entwerfer nutzt in der Regel aus ähnlichen Problemen, eigener Erfahrung oder der Fachliteratur (wie zum Beispiel [Kung80], [MiKo84], [Zakh84]) herausgezogene Lösungsvorschläge, Beispiele und Algorithmenmuster, die er an sein Problem anpaßt. Der passende Algorithmus muß vor der Realisierung beschrieben werden. Zu diesem Zweck braucht der Entwerfer ein geeignetes Beschreibungsmittel, welches diesen Algorithmus mit allen seinen nebenläufigen Eigenschaften auf der Verhaltensebene darstellen kann.

1.2.3 *Nebenläufigkeit auf der Verhaltensebene*

Nebenläufigkeit kann in einem digitalen System auf verschiedenen, sich nicht ausschließenden Schichten stattfinden. Auf der Verhaltensebene

können drei davon unterschieden werden, nämlich die Nebenläufigkeit zwischen Operationen, Funktionen und Prozessen (Tabelle 1.2). Diese Zerlegung in Schichten dient zur Einordnung der Beschreibungsmittel und entspricht den auf der Verhaltensebene existierenden Konzepten und Notationen zur Beschreibung von Nebenläufigkeit.

SCHICHT	BEISPIEL
Operationen	zwei parallele Addierer
Funktionen	Multiplikation von Matrizen
Prozesse	Mehrprozessorsystem

Tabelle 1.2 - Nebenläufigkeitsschichten

Die einfachste zu realisierende Schicht betrifft die Nebenläufigkeit zwischen Operationen. Diese ist direkt in digitalen Schaltungen durch mehrfache Anwendung digitaler Komponenten und deren gleichzeitige Aktivierung realisierbar. Die nächste Schicht bezieht sich auf die Nebenläufigkeit von Funktionen. Diese Klasse wird durch die gleichzeitige Durchführung von vollständigen Funktionen, die aus einer Menge primitiver Operationen bestehen, realisiert.

Die letzte Schicht ist die höchste, in der man Nebenläufigkeit beschreiben kann. Ein System wird in kooperierende Prozesse zerteilt, wobei jeder dieser Prozesse eine unabhängige, geschlossene Aufgabe ausführt. Diese Schicht schließt eine breite Skala von digitalen Systemen mit ganz verschiedener Architektur ein. Die nebenläufigen Einheiten können beispielsweise gleiche oder verschiedene Aufgaben ausführen, der Systembetrieb kann synchron oder asynchron stattfinden, und die Steuerung und die Datenbestände können entweder zentralisiert oder verteilt vorkommen. Die richtige nebenläufige Architektur auf dieser Schicht hebt die Leistung eines Systems bedeutend an [Sequ83]. Aus diesem Grund betont die Arbeit vor allem diese Art der Realisierung von Nebenläufigkeit.

Die meisten vorhandenen Beschreibungsmittel (z.B. [AlPe81], [DSST83]) decken die beiden ersten Schichten (Operationen und Funktionen) ab. Einige davon (z.B. [PaWa81]) auch die dritte, ohne allerdings zuverlässige Mechanismen zur Darstellung der Interaktionen zwischen Prozessen anzubieten, oder (wie [Noll86]) heben die direkte Beschreibung der Steuerung hervor.

Anhand einiger Beispiele von digitalen Systemem werden in Kapitel 2 die Anforderungen an Mittel zur Darstellung von Nebenläufigkeit auf der Verhaltensebene festgelegt und die verfügbaren Notationen und Mechanismen zur Beschreibung dieser Klasse von Problemen zusammengefaßt. Aus dieser Analyse ergeben sich ein Modell (Kapitel 3) und eine Beschreibungsform für nebenläufiges digitales Verhalten (Kapitel 4), die alle Schichten der Tabelle 1.2 abdecken.

1.2.4 *Ausnutzung der Nebenläufigkeit bei der Synthese*

Der Umfang der Implementierung digitaler Schaltungen reicht von rein manuellem Entwurf bis hin zur vollautomatischen Synthese. Diese Arbeit befaßt sich mit dem zweiten Fall, welcher auch als DA (design automation) bezeichnet wird. Die Vor- und Nachteile der automatischen Synthese sind in der Fachliteratur ([Clof84], [Feue83], [Nies83], [Shiv83], [Wern82]) ausführlich dargestellt und können wie folgt zusammengefaßt werden:

> Die automatische Synthese erzeugt mit großer Zuverlässigkeit korrekte Lösungen für digitale Probleme und ist bedeutend schneller als die rein manuellen Verfahren oder die rechnergestützten manuellen Verfahren (CAD). Der Preis dafür ist jedoch häufig ein weniger effizientes Ergebnis.

Im Rahmen dieser Arbeit wird der Nebenläufigkeitsgrad während der automatischen Synthese durch folgende Maßnahmen erhöht:

- Bestimmung und Implementierung von Nebenläufigkeit zwischen Operationen.

- Implementierung der Notationen und Mechanismen zur Darstellung von Nebenläufigkeit zwischen Funktionen und Prozessen, die in der Verhaltensbeschreibung vorhanden sind.

Die Grundlage der Synthese von digitalen Schaltungen wird in Kapitel 5 vorgestellt. In Kapitel 6 bis 8 werden die oben festgelegten Maßnahmen zur Erhöhung des Nebenläufigkeitsgrades in der Synthese von digitalen Systemen eingesetzt.

1.3 *Abgrenzung*

Die meisten Arbeiten aus dem Bereich des rechnergestützten Entwurfs und der automatischen Synthese befassen sich mit Konzepten und Werkzeugen für die Phasen des Entwurfszyklus, die unterhalb der strukturellen Ebene liegen ([HaPa81], [Thur82], [CaTr84]). Einige davon ([AlPe81], [DSST83]) beschäftigen sich auch mit der automatischen Synthese aus einer Verhaltensbeschreibung, ohne jedoch das nebenläufige Verhalten zwischen Prozessen (Tabelle 1.2) zu berücksichtigen. Es sind auch mehrere Mittel zur Beschreibung von Nebenläufigkeit vorhanden ([PaWa81], [SiCo83]), deren Ziel allerdings die Simulation oder Verifizierung ist, selten aber die Synthese.

Das hier vorliegende Verfahren unterscheidet sich von diesen Arbeiten dadurch, daß es ein vollständiges Konzept zum Behandeln aller Klassen von Nebenläufigkeit der Tabelle 1.2 enthält, mit dem Ziel, den höchstmöglichen Nebenläufigkeitsgrad zu erreichen.

1.4 *Die Aufgabenstellung*

Diese Arbeit befaßt sich mit der Untersuchung einer geeigneten Darstellung von nebenläufigem Verhalten digitaler Schaltungen und der entsprechenden Synthese der dabei dargestellten Systeme.

Aus der Analyse von typischen digitalen Systemen und vorhandenen Bescheibungsprachen ergibt sich die Entwurfssprache BABEL, die Nebenläufigkeit zwischen Prozessen, Funktionen und Operationen (Tabelle 1.2) darstellen kann. Zu dieser Entwurfssprache gehört ein Übersetzer, der standardisierte Eingabeinformationen für CADDY ([RoCa85], [SCR84]) erzeugt und damit die Ausnutzung einer Menge einsatzbereiter Werkzeuge erlaubt. BABEL liegt ein Stufe höher als die CADDY-Eingabesprache DSL ([CaWe84], [CaWe85a], [CaWe85b]) insofern, als BABEL die Vereinbarung mehrerer Prozesse erlaubt und ein DSL-Programm ein BABEL-System mit einem einzigen Prozeß darstellt.

Da die Synthese des Datenflusses in CADDY schon vorhanden ist ([Rose84], [CKR84]), befaßt sich diese Arbeit mit der Synthese der Steuerung. Das hier vorliegende Syntheseverfahren erzeugt für jeden BABEL-Prozeß eine Zustandsübergangstabelle. Es wird angenommen, daß es etablierte Verfahren gibt, welche Zustandsübergangstabellen in Schaltungsstrukturen umsetzen. Ein Beispiel für ein solches Verfahren ist das von CADDY verwendete System LOGE [Lipp83]. Um mehrere Prozesse zu verbinden, erzeugt die Synthese nach der Analyse von potentiellen Konflikten und der Auswahl

eines geeigneten Interaktionsschemas ein Muster zur Interaktion der Prozesse, welches die Interaktionsschaltung und das Kommunikationsprotokoll enthält.

Bild 1.4.1 zeigt die Umsetzung von Verhaltensbeschreibungen in Schaltungsstrukturen mit CADDY. Die Komponenten, die innerhalb des gepunkteten Rahmens liegen, sind Bestandteile dieser Arbeit.

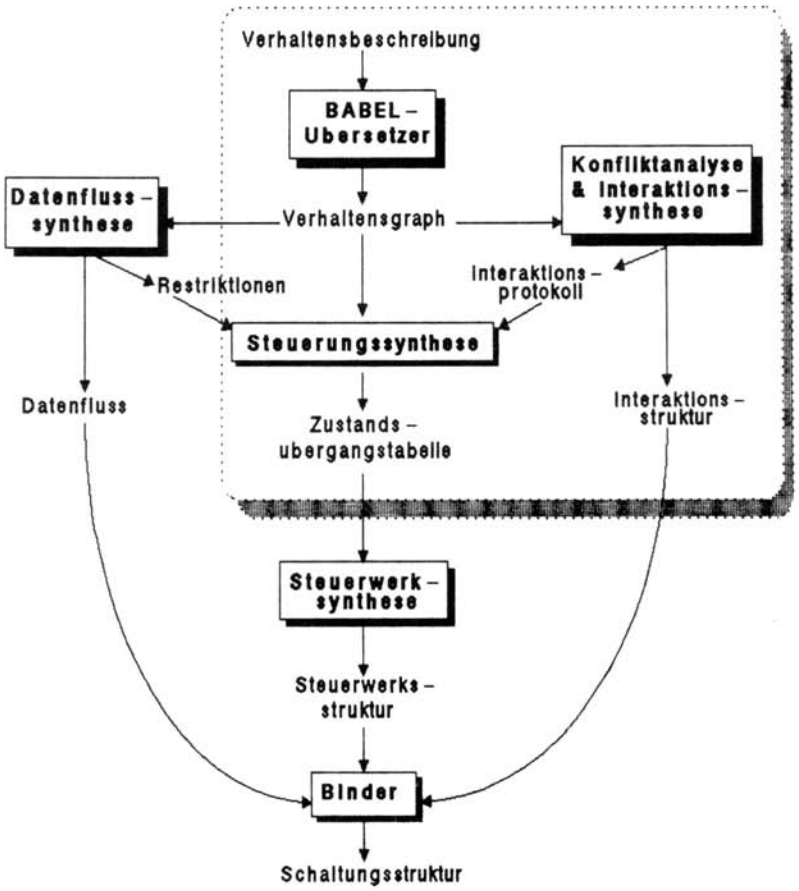


*Bild 1.4.1 - Strukturs*ynthese in CADDY mit BABEL als Eingabesprache

2. Verhaltensbeschreibung

Dieses Kapitel führt anhand einiger Beispiele von nebenläufigen Systemen Beschreibungsmittel für diese Art von Systemen auf der Verhaltensebene ein. Es findet eine sorgfältige Untersuchung der Hardwarebeschreibungssprachen statt, weil diese Beschreibungsmittel einen leichten Übergang zum automatischen Entwurf erlauben.

2.1 *Digitale Systeme mit nebenläufigem Verhalten*

Die folgenden Abschnitte führen Beispiele von digitalen Systemen ein, die typische Fälle von Nebenläufigkeit im Hardwarebereich darstellen. Anhand dieser Beispiele werden allgemeine nebenläufige Eigenschaften und einige Anforderungen an die formalen Beschreibungsmittel auf der Verhaltensebene festgelegt. Diese Beispiele dienen in Kapitel 4 auch dazu, die Anwendung von spezifischen Sprachnotationen in BABEL zu verdeutlichen.

Die hier vorliegenden Beispiele umfassen ein breites Spektrum von nebenläufigen Problemen im Hardwarebereich, welches ein Beschreibungsmittel behandeln muß. Der interessierte Leser sei auf weitere Literatur ([FoKu80], [Mold83], [HLSM82], [Pete81], [Trel82], [Zakh84]) verwiesen.

2.1.1 *Datenbearbeitung mit Pufferung*

Eine häufige Klasse von Problemen, die der Entwerfer digitaler Schaltungen behandeln muß, entsteht aus der Bearbeitung von Daten, die schneller erzeugt werden, als sie mit einem rein seriellen Verfahren verarbeiten werden können. Wenn die Erzeugungsfrequenz der Daten eine nicht beeinflußbare Größe ist, wird dieses Problem durch die Anwendung von mehreren Einheiten, die mit Puffern gemeinsamer Daten gleichzeitig arbeiten können, gelöst. Zwei Beispiele dieser Klasse seien hier vorgeführt.

Das erste Beispiel besteht aus drei voneinander unabhängigen Einheiten, die gemeinsame Daten verarbeiten (Bild 2.1.1). Die Geschwindigkeit, mit der jede Einheit ein Datenelement bearbeitet, ist unbestimmt. Die Einheit ERZEUGER liest Daten aus einer externen Quelle und speichert sie in einem Puffer (Ein-Puffer). Die Einheit TRANSFORMER bearbeitet jedes Datenelement einmal und speichert es in einem weiteren Puffer (Aus-Puffer). Die letzte Einheit (VERBRAUCHER) beendet die Bearbeitung und schreibt die Daten auf

einen externen Speicher.

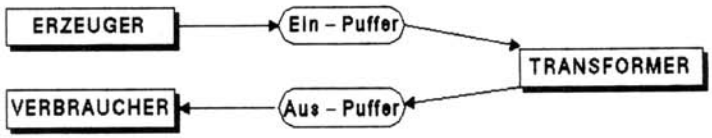


Bild 2.1.1 - Beispiel eines Systems mit drei unabhängigen Einheiten und zwei gemeinsamen Datenbeständen

In Bild 2.1.1 ist Ein-Puffer ein gemeinsames Mittel der Einheiten 'ERZEUGER' und 'TRANSFORMER' und Aus-Puffer ein gemeinsames Mittel von 'TRANSFORMER' und 'VERBRAUCHER'. Dabei benötigen diese Einheiten die Möglichkeit, Daten aus einem Puffer zu holen bzw. Daten in einen der Puffer einzufügen. Diese Aktionen (Holen und Einfügen) werden gemäß der Tabelle 2.1.1 von den entsprechenden Einheiten durchgeführt.

Einheit	Puffer	Aktion
ERZEUGER	Ein-Puffer	Einfügen
TRANSFORMER	Ein-Puffer	Holen
	Aus-Puffer	Einfügen
VERBRAUCHER	Aus-Puffer	Holen

Tabelle 2.1.1 - Verbindung zwischen den Einheiten und den Puffern des ersten Beispiels

Während der Bearbeitung können fehlerhafte Zustände auftreten, falls die Aktion 'Holen' versucht, Daten einzulesen, die im Puffer nicht vorhanden sind (d.h. der Puffer ist leer), oder falls 'Einfügen' Daten schreibt, wenn es keine freien Plätze im Puffer gibt (d.h. der entsprechende Puffer ist voll). Diese fehlerhaften Zustände können durch die bedingte Verzögerung einer Aktion vermieden werden, d.h. durch das Feststellen der ungünstigen Bedingungen (Puffer ist beim Holen leer bzw. beim Einfügen voll) und die Verzögerung der Aktionsausführung, bis die entsprechende Bedingung sich ändert.

Fehlerhafte Zustände treten auch auf, wenn zwei Einheiten gleichzeitig auf denselben Puffer zugreifen. Dabei kann ein inkonsistenter Wert aus einem Speicherplatz geholt werden, indem dieser Speicherplatz gelesen wird, während sich sein Inhalt als Folge einer Einfüge-Aktion ändert. Dieser Widerspruch kann bei gegeseitigem Ausschluß der Aktionen 'Holen' und 'Einfügen' vermieden werden.

Es sei jetzt angenommen, daß das Bearbeitungsverfahren der Einheit 'TRANSFORMER' so viel Zeit benötigt, daß es zu einem Engpaß im Datenfluß führt. Das zweite Beispiel (Bild 2.1.2) umgeht dieses Problem, indem anstatt einer zwei solcher Einheiten verwendet werden.

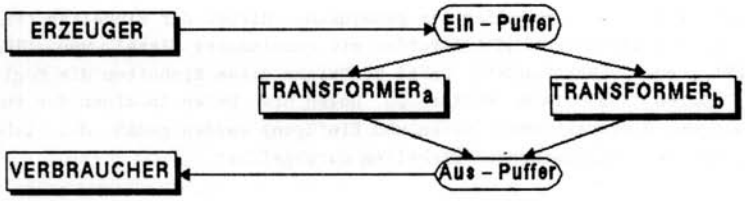


Bild 2.1.2 - Durchsatzsteigerung durch zwei Exemplare der Einheit TRANSFORMER

Die von den verschiedenen Einheiten benötigten Aktionen sind in Tabelle 2.1.2 dargestellt.

Einheit	Puffer	Aktion
ERZEUGER	Ein-Puffer	Einfügen
TRANSFORMER$_a$	Ein-Puffer	Holen
	Aus-Puffer	Einfügen
TRANSFORMER$_b$	Ein-Puffer	Holen
	Aus-Puffer	Einfügen
VERBRAUCHER	Aus-Puffer	Holen

Tabelle 2.1.2 - Verbindungen im zweiten Beispiel

Eine andere Darstellung der Tabelle 2.1.2 zeigt deutlich, welche Aktionen auf welchem Puffer durchgeführt werden können (Tabelle 2.1.3).

Puffer	Aktion	Einheiten
Ein-Puffer	Einfügen	ERZEUGER
	Holen	$TRANSFORMER_a$
	Holen	$TRANSFORMER_b$
Aus-Puffer	Holen	VERBRAUCHER
	Einfügen	$TRANSFORMER_a$
	Einfügen	$TRANSFORMER_b$

Tabelle 2.1.3 - Aktionen, die gleichzeitig aktiv sein können

Zusätzlich zu den im ersten Beispiel schon erwähnten Widersprüchen können hier fehlerhafte Zustände auch als Folge des simultanen Holens von Daten aus Ein-Puffer wie auch als Folge des gleichzeitigen Einfügens von Daten in Aus-Puffer auftreten. Ein fehlerhafter Zustand tritt beim gleichzeitigen Holen dann auf, wenn dieselben Daten mehrfach geholt werden. Beim gleichzeitigen Einfügen tritt er auf, wenn Daten verloren gehen und/oder verfälscht werden. Dieses widersprüchliche Verhalten kann durch gegenseitigen Ausschluß der von verschiedenen Einheiten ausgeführten Aktionen vermieden werden.

Für die Formulierung dieser Klasse von Problemen muß ein geeignetes Beschreibungsmittel den folgenden Anforderungen genügen:

- Beschreibung von mehreren unabhängigen Einheiten, auch von mehreren Kopien einer Einheit (z.B. die beiden TRANSFORMER-Einheiten des zweiten Beispiels) und ihren entsprechenden Verbindungen.

- Vereinbarung eines gegebenen Datenbestands (z.B. eines Puffers) als gemeinsames Mittel.

- Beschreibung der Mechanismen (hier der bedingten Verzögerung der Ausführung einer Aktion bzw. des gegenseitigen Ausschlusses), die das Auftreten von fehlerhaften Zuständen bei der Nutzung der gemeinsamen Mittel vermeiden.

2.1.2 *Prozessoren*

Das dritte Beispiel stellt einen Prozessor vor, der zwei nebenläufige Einheiten enthält (Bild 2.1.3). Eine Einheit ('FETCH') holt Befehle aus dem Speicher, die andere ('EXECUTE') holt Operanden aus dem gleichen Speicher und führt die Befehle aus. Die Einheiten arbeiten unabhängig, müssen aber zu unbestimmten Zeitpunkten auf gemeinsame Datenbestände zugreifen. Diese sind der Speicher ('memory') und der Befehlspuffer ('instruction queue'), in dem die geholten Befehle auf Dekodierung und Ausführung warten. Die Einheit, welche die Befehle ausführt, soll höhere Priorität in der Nutzung der Verbindung zum Hauptspeicher haben, damit sie die nötigen Operanden rechtzeitig holen kann.

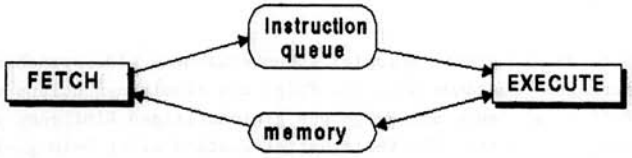


Bild 2.1.3 - Prozessor mit zwei nebenläufigen Einheiten

Zusätzlich zu den schon im letzten Abschnitt erwähnten Anforderungen an ein Beschreibungsmittel, tritt folgende Anforderung bei diesem Beispiel auch auf:

- Spezifikation der Prioritäten der Einheiten bezüglich eines von diesen Einheiten benutzbaren gemeinsamen Mittels (z.B. die Prioritätseinordnung von 'FETCH' und 'EXECUTE' bezüglich des gemeinsamen Mittels 'memory').

Das vierte Beispiel besteht aus drei gleichen Prozessoren und einem gemeinsamen Speicher (Bild 2.1.4). Jeder Prozessor in diesem Mehrprozessorsystem entspricht der Spezifikation des Beispiels 3 (Bild 2.1.3).

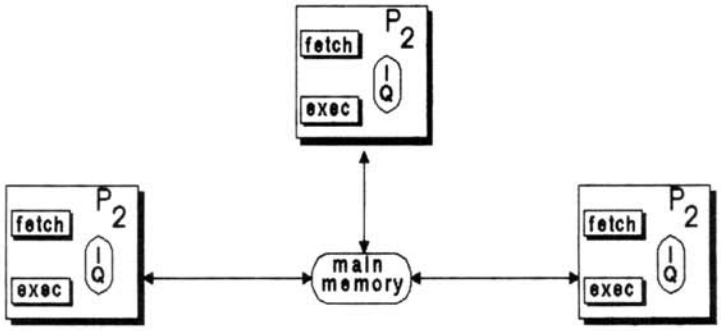


Bild 2.1.4 - Mehrprozessorsystem

Dieses System ist in zwei hierarchischen Ebenen zu betrachten. In der oberen Ebene liegen die drei Prozessoren (P_1, P_2 und P_3) und der gemeinsame Speicher ('main-memory'). In der unteren Ebene liegen die Komponenten der einzelnen Prozessoren (die zwei nebenläufigen Einheiten 'fetch' und 'exec' und der Befehlspuffer 'IQ').

Ein Beschreibungsmittel, welches dieses Beispiel beschreibt, muß die folgenden Anforderungen erfüllen:

- Beschreibung der hierarchischen Anordnung der Elemente eines Systems.

- Zuteilung jedes gemeinsamen Mittels zu einer hierarchischen Ebene (z.B. im Mehrprozessorsystem den Speicher an die obere Ebene und den Befehlspuffer an die untere Ebene).

2.1.3 *Vektoren und Matrizen*

Diese Klasse von Problemen entsteht aus der Realisierung von Algorithmen oder Verfahren, bei denen ein gewisser Grad von Parallelität a priori bekannt ist. Beispiele dazu sind die Multiplikation von Vektoren und Matrizen wie auch die Algorithmen zur schnellen Fouriertransformation (FFT - fast fourier transforms). Kung schlägt in [Kung82] die Anwendung von Reihenstrukturen (systolic arrays) für die Realisierung dieser Art von Algorithmen vor.

Ein gutes Beispiel dieser Klasse von Problemen ist die Schaltung zur Mustererkennung (*pattern matching chip*) von Foster und Kung [FoKu80] (Bild 2.1.5). Es handelt sich um einen Echtzeitvergleich eines Musters (pattern) mit einer beliebig langen Zeichenkette. Ist das Muster in der Zeichenkette enthalten, so wird dies entsprechend angezeigt.

Das System besteht aus mehrerer Zellen, die zusammen einen sog. 'Systolic'-Algorithmus ausführen. 'Systolic'-Algorithmen sind durch die folgenden Eigenschaften gekennzeichnet: (a) die Realisierung benötigt eine kleine Anzahl von einfachen Zellentypen, (b) die Verbindung zwischen den Zellen ist regulär, und (c) der Algorithmus nutzt eine Pipeline-Anordnung maximal aus.

Hier werden zwei Typen von Zellen, der Vergleicher und der Akkumulator, mehrfach verwendet. Jeder Vergleicher enthält als Eingabe abwechselnd ein Zeichen des Musters und ein Zeichen der Zeichenkette. Die Musterzeichen und die Zeichen der Zeichenkette durchlaufen die Zellen in entgegengesetzter Richtung. Wenn das Musterzeichen dem Zeichen der Zeichenkette entspricht, informiert der Vergleicher den entsprechenden Akkumulator. Jeder Akkumulator bekommt zusätzlich dazu eine Anzeige des Endes des Musters sowie eine Anzeige der Position des 'don't-care'-Zeichens X im Muster. Der Akkumulator beinhaltet ein vorläufiges Ergebnis 't'. An Ende der Kette benutzt er 't', um das von rechts nach links fließende 'Ergebnis' zu ersetzen.

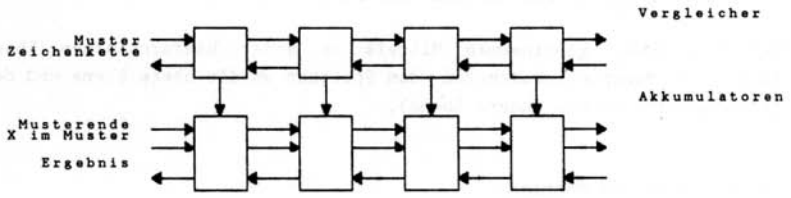


Bild 2.1.5 - Systolic Array zur Mustererkennung

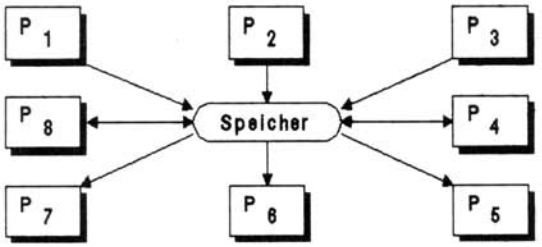
Hier ist wieder ein Beschreibungsmittel nötig, welches die mehrfache Anwendung einer beschriebenen Einheit erlaubt. Im Rahmen dieser Arbeit werden die mehrfachen Erscheinungen einer beschriebenen Einheit Verkörperungen der Einheit genannt.

2.1.4 *Synchronisation mehrerer Abläufe (Rendez-vous)*

Das sechste Beispiel stellt im Gegensatz zu den anderen keinen Wettbewerb um gemeinsame Mittel dar, sondern den häufigen Fall der Synchronisation verschiedener Aktivitäten. Mehrere unabhängige Einheiten müssen sich in einem bestimmten Zustand ihres Ablaufs synchronisieren. Die Einheiten, die zuerst diesen Treffpunkt erreichen, müssen daher auf alle anderen warten. Hier ist ein Beschreibungsmittel erforderlich, welches diese Art von Synchronisation darstellen kann.

2.1.5 *Leser-Schreiberproblem*

Zu diesen Beispiel soll angenommen werden, daß viele Einheiten auf einem Speicherelement (z.B. ein Register) lesen und schreiben können. Mehrere Lesevorgänge dürfen daher gleichzeitig ablaufen, während Schreiben nur unter gegenseitigem Ausschluß durchgeführt werden darf (Bild 2.1.6). Anders als bei den Beispielen des Abschnitts 2.1.1 kann hier ein Datenelement mehrfach gelesen und bearbeitet werden (dieses kann zum Beispiel eine 'Status'-Information sein). Schreiben darf wie vorher nur im Alleingang stattfinden.



Schreiber=$\{P_1,P_2,P_3\}$ Leser=$\{P_5,P_6,P_7\}$ Scheiber/Leser=$\{P_4,P_8\}$

Bild 2.1.6 - Einheiten mit nebenläufigen Zugängen zu einem gemeinsamen Speicherelement

In Gegensatz zum 'fetch-look-ahead'-Prozessor, wo eine Einheit Vorrang zum Speicher hatte, muß hier nicht beschrieben werden, welche Einheit, sondern welche Aktion (Lesen oder Schreiben) Zugriffsvorrang hat. Eine

andere interessante Eigenschaft dieses Problems ist, daß einige Aktionen mit gegenseitigem Ausschluß durchzuführen sind (z.B. Lesen und Schreiben oder mehrere Schreiben) und andere nicht (z.B. mehrere Lesen), weil diese Aktionen keine Konflikte in der Nutzung des gemeinsamen Speicherelements verursachen.

Zur Beschreibung dieses Problems sind folgende Maßnahmen erforderlich:
- Spezifikation der Priorität der Aktionen.
- Einsetzen des gegenseitigen Ausschlusses von Aktionen, nur wenn es nötig ist (d.h. falls Widersprüche auftreten können).

2.1.6 *Gemeinsame Eigenschaften*

Die Tabellen 2.1.4 und 2.1.5 zeigen eine Zusammenfassung der den Beispielen gemeinsamen Eigenschaften. Ein Mittel, welches zur Beschreibung dieser Beispiele verwendet wird, muß Notationen zur Darstellung dieser Eigenschaften anbieten.

EIGENSCHAFTEN / BEISPIELE	1	2	3	4	5	6	7
Gemeinsame Datenbestände und/oder Betriebsmittel	X	X	X	X			X
Bedingte Verzögerung der Ausführung einer Aktion	X	X	X	X			
Gegenseitiger Ausschluß von Aktivitäten	X	X	X	X			X
Synchronisation der Abläufe						X	
Priorität			X	X			X
Merfache Verkörperung einer Einheit	X	X		X	X		
Verbindung der Einheiten	X	X	X	X	X	X	X
Hierarchische Darstellung				X			

Tabelle 2.1.4 - Gemeinsame Eigenschaften der Beispiele

Beispiel	1 :	3 Einheiten, 2 Puffer
	2 :	Pipeline
	3 :	Prozessor
	4 :	Mehrprozessorsystem
	5 :	Systolic-Array
	6 :	Rendez-vous
	7 :	Leser-/Schreiberproblem

Tabelle 2.1.5 - Liste der Beispiele

In den nächsten Abschnitten werden einige Mittel zur Beschreibung dieser Art von Problemen eingeführt.

2.2 *Beschreibung des nebenläufigen Verhaltens*

Ein digitales System kann auf verschiedenen Abstraktionsebenen und durch Anwendung verschiedener Mittel, wie zum Beispiel durch Blockdiagramme, Zeitdiagramme, Schaltnetze und Schaltwerke, algebraische Formulierungen, formale und natürliche Sprachen usw. beschrieben werden. Jedes dieser Beschreibungsmittel kann in mehreren Phasen des Entwurfszyklus (Tabelle 1.1) eingesetzt werden, obwohl für jede Ebene und jeden Zweck einige Formulierungsformen geeigneter sind als andere. Natürliche Sprache findet zum Beispiel verbreitete Anwendung in der Problemspezifikation und Blockdiagramme, Schaltnetze und Schaltwerke sind auf der Ebene der Schaltungsstruktur das übliche Beschreibungsmittel.

Von Interesse für diese Arbeit sind Beschreibungsmittel auf der Verhaltensebene, welche den Lösungsalgorithmus und das Verhalten eines Systems bezüglich seiner Außenwelt umfassen. Die Arbeiten von Rem [Rem84] und Van de Snepscheut [Snep83], in denen die Beschreibung von Schaltungen durch eine Menge von 'Traces' erfolgt, von Thurn [Thur82] durch nebenläufige Graphen, von Floyd und Ullmann [FlUl82] durch reguläre Ausdrücke, von Barbacci [Barb85] durch Anwendung von Ada und von Suzuki [Suzu85] durch Anwendung von 'Concurrent Prolog' zeigen das breite Spektrum vorhandener Möglichkeiten zur Beschreibung nebenläufiger Verhalten.

In den folgenden Abschnitten werden einige Mittel zur Beschreibung von Nebenläufigkeit wie Petri-Netze, Programmiersprachen und Hardwarebeschreibungssprachen betrachtet.

2.2.1 *Beschreibung durch Netze*

Petri-Netze [Pete77] stellen ein anerkanntes Hilfsmittel zur Beschreibung nebenläufiger Vorgänge dar. Es ist eine Vielzahl von Beschreibungsformen und Syntheseverfahren bekannt, die auf Erweiterungen und Einschränkungen der ursprünglichen Petri-Netz-Definitionen beruhen.

Ullrich [Ull76] zeigt jedoch in seiner Dissertation, daß Petri-Netze nicht zur Beschreibung größerer Probleme geeignet sind, da die Schwierigkeiten beim Nachweis der Eigenschaften 'lebendig' und 'sicher', welche die Korrektheit der Beschreibung darstellen, exponentiell mit der Zahl der Netzelemente wachsen. Um dieses Problem zu überwinden, entwarf Ullrich ein Verfahren, welches das Petri-Netz in nebenläufige, kommunizierende Zustandsgraphen zerlegt. Die Beschreibung des gesamten Problems in Form eines Petri-Netzes, welches die Steuerung der zu realisierenden Schaltung darstellt, bleibt allerdings Aufgabe des Entwerfers.

Die Komplexität der Korrektheitsüberprüfung und der Aufwand bei der Formulierung des Problems durch Petri-Netze erschweren die Anwendung dieses Beschreibungsmittels auf der Verhaltensebene. Thurn vertritt in seiner Dissertation [Thur82] sogar die Ansicht, daß die direkte Problemdarstellung in Form von Petri-Netzen dem Entwerfer nicht überlassen werden kann. Wie im folgenden gezeigt wird, tritt dieser Aufwand bei Hardwarebeschreibungssprachen im allgemeinen nicht auf; deshalb wird diese letzte Art von Beschreibung hier bevorzugt.

2.2.2 *Allgemeine Programmiersprachen*

Zahlreiche Mechanismen und Programmiersprachen, die Nebenläufigkeit unterstützen, wurden in den letzten Jahren entwickelt und finden verbreitet Anwendung auf den Gebieten der Betriebssysteme, der Verwaltung von Datenbanksystemen und bei Realzeit-Systemen. Fortschritte in der Theorie der nebenläufigen Programmierung (concurrent programming) bieten dem Benutzer Sprachnotationen an, welche die Nebenläufigkeit auf einfache Weise darstellen, die Anforderungen an Synchronisation explizit erfüllen und eine Uberprüfung der Korrektheit erleichtern. Gute Beispiele von nebenläufigen Programmiersprachen sind 'Concurrent Pascal' [Brin75], 'Modula' [Wirt77] und 'Ada' [USD81].

Höhere Programmiersprachen, wie die oben erwähnten, können auch vorteilhaft zur Beschreibung nebenläufigen digitalen Verhaltens verwendet werden. Suzuki [Suzu85], zum Beispiel, benutzt 'Concurrent Prolog' ohne

jede Anpassung, um nebenläufige digitale Systeme zu beschreiben.

Die Vorteile der direkten Anwendung etablierter Programmiersprachen liegen in folgenden Punkten:

- Die Sprachen sind gut geeignet zur reinen Verhaltensbeschreibung, da sie ausschließlich den Lösungsalgorithmus darstellen können, unabhängig von Implementierungsmechanismen auf tieferen Ebenen.

- Nebenläufigkeit und Synchronisation sind leicht in den vorhandenen Sprachnotationen darzustellen, ohne daß der Benutzer die nötigen Mechanismen selbst entwerfen muß.

- Verfahren zur Verifikation der Korrektheit des nebenläufiges Verhaltens, welches die Sprachnotationen beschreiben können, sind teilweise schon vorhanden.

Es gibt trotzdem Nachteile, welche die Anwendung von Programmiersprachen zur Beschreibung digitaler Schaltungen erschweren. Diese liegen in folgenden Punkten:

- Die Programmiersprachen berücksichtigen nicht wichtige implizite Hardwareeigenschaften und einige Besonderheiten, die beschrieben werden müssen, wie zum Beispiel die Notwendigkeit, Anschlüsse (Pins) zu vereinbaren, asynchrones Verhalten darzustellen und Zeiteinschränkungen zu spezifizieren.

- Die Programmiersprachen nehmen als Voraussetzung an, daß es mächtige Betriebssysteme oder andere ähnlich souveräne Betriebsmittel gibt, welche die verwendeten Synchronisationsprimitive realisieren, das nebenläufige Verhalten überwachen und den genauen Ablauf des Programms bestimmen (zum Beispiel der Betriebssystemlinker von 'Concurrent Pascal'). Das ist im allgemeinen im Bereich des Hardwareentwurfs nicht der Fall. Es gibt kein auf der physikalischen Ebene schon realisiertes und sofort verfügbares Betriebsmittel, welches die oben erwähnten Aufgaben eines Betriebssystems erledigt und damit Nebenläufigkeit unterstützt. Alle, das zu entwerfende System und der dazu passende Synchronisationsmechanismus, müssen letztendlich zusammen realisiert werden.

BABEL versucht, wie in Kapitel 4 gezeigt wird, die Vorteile der Programmiersprachen auszunutzen und ihre Nachteile durch Anpassung des Synchronisationsmechanismus an Hardwareeigenschaften zu vermeiden.

2.2.3 *Hardwarebeschreibungssprachen*

Wegen der wachsenden Komplexität von digitalen Systemen und des Bedarfs an guten Werkzeugen zur Beschreibung und zum Entwurf solcher Systeme wurde in der letzten Zeit eine Vielzahl von Hardwarebeschreibungssprachen (HDL) entwickelt ([Comp74], [Comp77], [Comp85]). Die meisten von ihnen finden Anwendung auf dem Gebiet der Simulation und Verifikation, einige davon in der automatischen Synthese.

Angemessene HDLs gibt es für alle Entwurfsphasen unterhalb der Spezifikationsebene (Tabelle 1.1), wobei für die Beschreibung auf den realisierungsnahen Ebenen (register transfer, gate, switching circuit, layout) eine größere Anzahl von HDLs zur Verfügung steht [SiTr81]. Viele HDLs erlauben die Beschreibung auf mehreren dieser Ebenen.

Sprachen auf der Verhaltensebene sind selten zu finden, obwohl eine Tendenz in diese Richtung bei der Untersuchung neuer Veröffentlichungen auf diesem Gebiet zu vermerken ist [DAC85]. DSL, die Eingangsprache des CADDY-Systems, und BABEL stellen reine Beispiele dieser Kategorie von Sprachen dar, die außerdem über keine Konstruktionen auf tieferen Ebene verfügen. Noch seltener sind HDLs, welche die Formulierung von Nebenläufigkeit erlauben. Einige davon werden im Abschnitt 2.3 untersucht.

Besonderes Merkmal der HDLs ist es, daß sie über die Vorteile der allgemeinen Programmiersprachen verfügen und gleichzeitig die Hardwareeigenschaften mitberücksichtigen.

2.3 *Nebenläufigkeit in Hardwarebeschreibungssprachen*

Die folgenden Hardwarebeschreibungssprachen verfügen über Notationen zur Darstellung von Nebenläufigkeit:

ISPS - 'Instruction Set Processor Specification' [Barb81] eine Sprache zur Beschreibung von Rechnern und Digitalschaltungen auf der Befehlsebene.

SLIDE - 'Structured Language for Interface Description and Evaluation ' [PaWa81] beschreibt Ein-/Ausgabesysteme, Schnittstellen und Verbindungen von digitalen Systemen.

CAP/DSDL - 'Concurrent Algorithmic Programming Language / Digital Systems Description Language' [Ramm82] ist eine Pascal-ähnliche Sprache, die ein weites Spektrum von Abstraktionsgraden zwischen Systemebene und Realisierungsebene abdeckt.

DSL - 'Digital System Specification Language' ist eine Entwurfssprache, die die Beschreibung und Synthese komplexer digitaler Funktionen erlaubt. Die neueste Version von DSL [CaWe84] verfügt über Notationen zur Spezifikation von Zeitverhalten, Schnittstellen, Moduln und ermöglicht die Vereinbarung mehrerer Sequenzen.

Das ursprüngliche Anwendungsgebiet von ISPS, CAP und SLIDE war die Simulation, während DSL von Anfang an als Entwurfssprache entwickelt wurde. ISPS dient inzwischen auch als Eingangssprache eines automatischen Entwurfssystems [DSST83].

Diese Sprachen benutzen teilweise die auf dem Gebiet der Programmiersprachen verbreiteten Konzepte von gegenseitigem Ausschluß (*mutual exclusion*), kritischen Abschnitten (*critical sections*), Semaphoren und gemeinsamen Mitteln (*shared resources*). Diese Konzepte sind in der Literatur ([BenA82], [AnSc83]) ausführlich behandelt und werden hier deshalb nicht wiederholt. Der Begriff Prozeß ist dagegen nicht mit Prozessen im Bereich der Betriebssysteme und einiger Programmiersprachen zu vergleichen. Um das Problem der Anwendung dieses Begriffes zu vereinfachen, wird in diesem Abschnitt jede unabhängige Einheit Prozeß genannt. Eine formelle Definition von Prozeß im Rahmen eines nebenläufigen Modells wird in Kapitel 3 eingeführt.

2.3.1 *Notationen zur Darstellung der Nebenläufigkeit*

Die Tabelle 2.3.1 faßt die Möglichkeiten zur Nebenläufigkeitsdarstellung in den vier Sprachen zusammen.

Komplexe Einheiten werden in ISPS als Prozeduren vereinbart. Der Ablauf dieser Prozeduren findet entweder sequentiell (NEXT) oder parallel (;) statt. Eine mit dem Attribut *process* aktivierte ISPS-Prozedur läuft nach dem Prozeduraufruf unabhängig ab. Der Entwerfer muß selbst abschätzen, wann diese Art von Prozedur fertig wird, oder er benutzt eine eingebaute Funktion (IS.RUNNING), um nach der Aktivität dieser Prozedur zu fragen. Diese Funktion ist hauptsächlich ein Hilfsmittel zur Simulation und modelliert keinen direkt auf der Schaltung realisierbaren Synchronisationsmechanismus.

Nebenläufigkeits-darstellung	ISPS	SLIDE	CAP	DSL
Prozeß-vereinbarung	Ja	Ja	-	-
paralleler vs. sequentieller Ablauf	; NEXT	; NEXT	conbegin end seqbegin end	[,] ;
Zusätzliche Möglichkeiten	-	-	Netz	mehrere Sequenzen
Aktivierung	Proze-dural	INIT	-	START

Tabelle 2.3.1 - Darstellung von Nebenläufigkeit in verschiedenen Hardwarebeschreibungssprachen

Das Prozeßkonzept der SLIDE-Sprache ist für strukturierte Hardwarebeschreibungen besser geeignet als das entsprechende Konzept in ISPS. Jede Menge von Ereignissen, die eine unabhängig gesteuerte Umgebung oder einen endlichen Automaten darstellt, wird in SLIDE 'Prozeß' genannt. Die SLIDE-Notation *INIT* bestimmt die Aktivierung eines Prozesses. Jeder Prozeß wird aktiv, wenn die durch *INIT* vereinbarten Bedingungen auftreten. Innerhalb eines SLIDE-Prozesses werden Anweisungen sequentiell oder parallel durch die Notation *NEXT* oder ; dargestellt.

In CAP ist kein Prozeßkonzept vorhanden. Ein Pascal-ähnliches Prozedurkonzept ist das einzige Mittel zur Modularisierung und Strukturierung in dieser Sprache. Der Ablauf von Anweisungen findet in CAP sequentiell oder parallel durch die Anwendung von *seqbegin* und *conbegin* statt. Diese Notationen sind *NEXT* und *;* in SLIDE ähnlich. CAP bietet noch eine unstrukturierte Möglichkeit an, um Nebenläufigkeit durch ein Netz darzustellen. Der Entwerfer kann jedoch das nebenläufige Netz mit *seqbegin* und *conbegin* einfacher darstellen [Ramm82].

In DSL gibt es das Konzept der Sequenz. Eine Sequenz, auch imperativer Teil genannt, stellt einen endlichen Automaten dar und könnte deswegen auch Prozeß genannt werden. In Kapitel 3 werden die Konzepte von Sequenz

und Prozeß vertieft und die Unterschiede zwischen beiden erklärt.

Der Ablauf von Anweisungen S_i ist in DSL innerhalb einer Sequenz parallel oder sequentiell durch die Anwendung der folgenden Sprachnotationen darzustellen:

Sequentiell - $S_1;\ldots;S_n$
Parallel - $[S_1,\ldots,S_n]$ oder FORK $S_1,\ldots,S_n$ JOIN

2.3.2 *Interaktion zwischen Prozessen*

In den vier HDL-Sprachen ist die Kooperation über gemeinsame Mittel der verwendete Interaktionsmechanismus. Um Widersprüche zu vermeiden, muß ein exklusiver Zugriff auf gemeinsam benutzte Mittel durch gegenseitigen Ausschluß vorhanden sein. Die Tabelle 2.3.2 zeigt eine Zusammenfassung der Mechanismen zur Darstellung von gegenseitigem Ausschluß in den vier Sprachen.

Interaktion zwischen Prozessen	ISPS	SLIDE	CAP	DSL
Explizite gemeinsame Mittel	-	-	-	-
Kritische Abschnitte	Ja	-	-	-
Gegenseitiger Ausschluß	Ja	Priorität	Ja	CONT STOP

Tabelle 2.3.2 - Interaktion zwischen Prozessen

Es gibt keine spezifische Notation in diesen Sprachen, um gemeinsame Mittel zu vereinbaren. In SLIDE, CAP und ISPS beschreibt der Entwerfer die gemeinsam benutzten Mittel durch Vereinbarung von globalen Variablen. Alle Prozesse haben auf jede globale Variable freien Zugriff. Es ist kein Mechanismus vorhanden, der diese Variablen gegen simultane Zugriffe schützt.

In DSL werden alle Variablen global vereinbart, so daß es keine Unter-

schiede zwischen gemeinsamen oder exklusiven Mitteln gibt. Die Variablen in DSL sind alle frei zugänglich und der Entwerfer muß sich selbst darum kümmern, daß keine Widersprüche wegen simultaner Zugriffe auftreten. Eine nützliche Hilfe bietet der DSL-Compiler dem Entwerfer insofern an, als er durch Analyse der nebenläufigen Sequenzen und parallelen Zweige bestimmt, wo Widersprüche möglicherweise eintreten können.

Eine explizite Vereinbarung von kritischen Abschnitten ist in den vier Sprachen nur bei ISPS vorhanden. Jede Prozedur oder jeder Prozeß kann das Attribut *critical* bekommen, welches die mehrfache Aktivierung dieser Prozedur verbietet. Dieser Mechanismus verbietet jedoch die gleichzeitige Aktivierung mehrerer verschiedener 'kritischer' Prozeduren nicht und ist deshalb nicht geeignet, Widersprüche durch simultane Zugriffe auf ein gemeinsames Mittel durch verschiedene Prozeduren zu vermeiden.

CAP unterstützt den gegenseitigen Ausschluß durch einen im Simulator eingebauten Mechanismus, der die mehrfache Aktivierung irgendeiner Prozedur vermeidet. Wie in ISPS geschieht ein gegenseitiger Ausschluß nur bei Exemplaren derselben Prozedur.

SLIDE bietet einen ungewöhnlichen Mechanismus an, um den gegenseitigen Ausschluß zu unterstützen. Jeder Prozeß wird unter einer expliziten Priorität vereinbart, welche seine Aktivierung und seinen Ablauf beeinflußt. Wenn ein Prozeß aktiv wird, werden alle anderen aktiven Prozesse mit geringerer Priorität beendet. Der Mechanismus garantiert gegenseitigen Ausschluß bei Prozessen derselben Ebene und bietet eine zeitliche Anordnung für die Prozeßaktivierung an. Die Nebeneffekte der abrupten Deaktivierung von Prozessen, die ein Simulator leicht überwinden kann, sind in einer realisierten Schaltung unerwünscht.

Durch die Anwendung der asynchronen Kommandos STOP und CONTINUE kann der Entwerfer in DSL den gegenseitigen Ausschluß von nebenläufigen Sequenzen nachbilden. Mit diesen Kommandos, welche die Aktivierung und Deaktivierung von Sequenzen steuern und mit Signalen, welche die Sequenzen erzeugen, kann der Entwerfer eine Art von Semaphor realisieren und damit einen gegenseitigen Ausschluß erreichen.

2.3.3 *Synchronisationsmechanismen*

Wenn zwei nebenläufige Prozesse zu einem bestimmten Zeitpunkt interagieren wollen, muß irgendeine Art von Synchronisation stattfinden. Die notwendige Synchronisation kann beispielsweise durch den gegenseitigen Ausschluß oder durch Mechanismen wie Semaphore oder Monitore verwirklicht werden.

Synchronisationsprimitive wie die Semaphore P und V wurden in SLIDE als 'SIGNAL' und 'RECEIVE' definiert und sind in dieser Sprache zur Simulation und Verifikation von E/A-Beschreibungen eingeführt. Zur Zeit stehen keine Verfahren zur korrekten Synthese dieser Primitive zur Verfügung.

Die eingebaute IS.RUNNING-Funktion in ISPS erlaubt durch Abfrage des Status des ISPS-Simulators eine Art von Synchronisation während der Simulation. Außerhalb der Simulationsumgebung (z.B bei der Synthese) stellt diese Funktion keinen nützlichen Synchronisationsmechanismus dar. Die IS.RUNNING-Funktion ist weiter nichts als ein Hilfsmittel zur Simulation und dient nicht zur Beschreibung des Verhaltens eines Systems.

Die Tabelle 2.3.3 zeigt den Mangel an vorhandenen Synchronisationsmechanismen in den hier besprochenen vier HDL's.

	ISPS	SLIDE	CAP	DSL
Synchronisations-mechanismen	Eingebaute Funktion (Simulation)	Semaphore P & V	-	-

Tabelle 2.3.3 - Synchronisationsdarstellung

2.3.4 *Bewertung*

Die letzten Abschnitte verdeutlichen, daß alle hier analysierten Sprachen einen Mangel an Notationen zur Beschreibung von wirksamen Mechanismen zur Interaktion und Synchronisation zwischen nebenläufigen Einheiten aufweisen. Dieser Mangel an geeigneten Notationen erschwert die Beschreibung des zu entwerfenden Systems sowie seine Synthese und die Überprüfung der Korrektheit seines nebenläufigen Verhaltens.

3. Verhaltensmodell

Das hier vorgeschlagene Verhaltensmodell zur Beschreibung der Nebenläufigkeit (BABEL-Modell) beruht auf dem DSL-Modell. Die Gemeinsamkeit beider Modelle liegt darin, daß ein Prozeß, die nebenläufige Einheit des BABEL-Modells, grundsätzlich einem DSL-Programm entspricht. In den folgenden Abschnitten werden die Elemente des DSL-Modells, die für den Aufbau des BABEL-Modells nötig sind, erklärt und das BABEL-Modell eingeführt.

3.1 *Das DSL-Modell*

Die Semantik eines in DSL beschriebenen digitalen Systems wird durch einen gerichteten Graphen bestimmt, welcher das Verhalten dieses Systems darstellt. Dieser Graph wird hier DSL-Graph genannt.

Der interne Bericht 'Interne Form und Semantik von DSL' [CaWe85a] enthält die ausführliche Beschreibung der Knoten und Strukturen des DSL-Graphen. Von allen Informationen eines DSL-Graphen ist hier die zeitliche Reihenfolge von Operationen von Interesse, da diese Relation den Ablauf eines Programms und damit die Synthese der Steuerung bestimmt. Im applikativen Teil eines DSL-Programms ist keine zeitliche Relation vorhanden, weil in diesem Teil die Operationen datenabhängig sind und nicht von einer externen Steuerung aktiviert werden. Deshalb beschränkt sich diese Arbeit auf den imperativen Teil des DSL-Programms.

Die hier benutzten Definitionen von Graphen und Teilgraphen entstammen der Literatur [Ebe81].

3.1.1 *Verhaltensgraph*

Die vom DSL-Übersetzer erzeugte Zwischenform ist ein eindeutiger vollständiger Verhaltensgraph.

DEF. 3.1.1 - Verhaltensgraph

> Ein Verhaltensgraph ist ein gerichteter Graph G, $G = (V,E,\varphi)$, der aus einer endlichen, nicht leeren Menge V von Knoten, einer endlichen Menge E von Kanten und eine Inzidenzabbildung φ: $E \rightarrow V \times V$

besteht. Für V und E gilt:

$$V = O_p \cup T_p \cup S_t, \qquad E = E_b \cup E_u$$

Die Menge V entsteht aus der Vereinigung der Menge O_p der Operationen, der Menge T_p der Tests und der Menge S_t der Strukturknoten. Die Menge E entsteht aus der Vereinigung der Menge E_b der bedingten Kanten und der Menge E_u der unbedingten Kanten. Die Kanten E_b und E_u sind wie folgt definiert:

$$E_b \rightarrow T_p \times V \quad \text{und} \quad E_u \rightarrow (O_p \cup S_t) \times V$$

Gibt es für zwei Knoten v und z eine Kante e mit $\varphi(e) = (v,z)$, so nennt man z einen Nachfolger (*successor*) von v und v einen Vorgänger (*predecessor*) von z. Da ein Knoten v mehrere Nachfolger und mehrere Vorgänger haben kann:

$$(\varphi(e_n) = (v,x) \ \wedge \ \varphi(e_m) = (v,y)) \text{ und}$$
$$(\varphi(e_p) = (u,v) \ \wedge \ \varphi(e_q) = (w,v))$$

erfolgt die Darstellung der Nachfolger- und Vorgänger-Mengen mit Hilfe der folgenden Funktionen:

Nachfolger von v: $SUC(v) = \{x, y\}$
Vorgänger von v: $PRED(v) = \{u, w\}$

DEF. 3.1.2 - Pfad

Eine Folge $C = (v_0, v_1, \ldots, v_n)$ von Knoten eines Verhaltensgraphen heißt ein Pfad, falls gilt:

$$v_i \in SUC(v_{i-1}), \quad 1 \leq i \leq n$$

Dabei heißt: v_0: der Anfangsknoten,
v_n: der Endknoten,
n : der Länge des Pfades.

Man spricht auch von der Menge der Knoten P_{vz}, die auf dem Pfad $C = (v,..,z)$ liegen und sagt C geht durch die Knoten von P_{vz}. Falls kein Knoten in einem Pfad mehrfach vorkommt, handelt es sich um einen Weg.

DEF. 3.1.3 - Eindeutiger Verhaltensgraph

Ein Verhaltensgraph heißt eindeutig, falls gilt:
$\forall\ v \in O_p\ ,\ |\ SUC(v)|\ = 1$

Dies bedeutet, daß alle Knoten $v \in O_p$ eines eindeutigen Verhaltensgraphen sich auf einen einzigen Nachfolger beziehen. Alle hier behandelten Verhaltensgraphen sind eindeutig.

DEF. 3.1.4 - Vollständiger Verhaltensgraph

Ein Verhaltensgraph heißt vollständig, wenn es für alle Testergebnisse, die ein Test $t \in T_p$ liefert, einen und nur einen Nachfolger gibt, der mit t durch eine bedingte Kante verbunden ist.

Die Definition 3.1.4 bedeutet, daß jedes Testergebnis eines Tests zu einem Nachfolger führt. Diese Definition schließt nicht die Möglichkeit aus, daß mehrere Testergebnisse durch eine einzige bedingte Kante zu demselben Nachfolger führen.

DEF. 3.1.5 - Zusammenhängender Verhaltensgraph

Gegeben seien zwei Knoten v und z, ein Pfad C_v durch v ($v \in P_v$, die Menge der Knoten von C_v), und ein Pfad C_z durch z ($z \in P_z$, die Menge der Knoten von C_z). Ein Verhaltensgraph G heißt zusammenhängend, wenn gilt:

$\forall\ v,z \in G,\ \exists\ C_v \wedge \exists\ C_z\ :\ P_v \bigcap P_z \neq \{\ \}$

Zwei besondere zusammenhängende Teilgraphen, Kette und Abschnitt, werden im folgenden eingeführt. Für diese Teilgraphen sind zwei Funktionen definiert, FIRST und LAST. Diese Funktionen stellen den Anfangs- bzw. Endknoten dieser Teilgraphen dar.

DEF. 3.1.6 - Abschnitt

Ein Abschnitt $A = (V',E',\varphi')$ ist ein zusammenhängender Teilgraph eines Verhaltensgraphen $G = (V,E,\varphi)$ mit einem einzigen Anfangsknoten $\alpha \in V'$, $FIRST(A) = \alpha$, und einem einzigen Endknoten $o \in V'$, $LAST(A) = o$, wobei gilt:

$\forall\ v \in (V' - \{\alpha\})\ :\ PRED(v) \subset V'$

$$\forall\ v \in (V' - \{o\}) : SUC(v) \subset V'$$

Die Vorgänger aller Knoten außer α sind Knoten des Abschnitts und die Nachfolger aller Knoten außer o sind Knoten des Abschnitts. Sei $PRED(\alpha) \subset V$ und $SUC(o) \subset V$, seien auch die Menge $ExtPred_\alpha$ der Vorgänger von α, die außerhalb A liegen, und Menge $ExtSuc_o$ der Nachfolger von o, die außerhalb A liegen:

$$ExtPred_\alpha = \{z \mid z \in PRED(\alpha) \text{ und } z \notin V'\} \text{ und}$$
$$ExtSuc_o = \{z \mid z \in SUC(o) \text{ und } z \notin V'\},$$

dann soll gelten: $|ExtSuc_o| \leq 1$ und $|ExtPred_\alpha| \leq 1$.
Das bedeutet: wenn es Nachfolger von o (oder Vorgänger von α) außerhalb des Abschnitts gibt, dann nur einen.

Ein Abschnitt, wobei jeder Knoten einen einzigen Nachfolger hat, heißt Kette.

DEF. 3.1.7 - Kette

Eine Kette $K = (V', E', \varphi')$, $V' \subset O_p$, $E' \subset E_u$, ist ein Abschnitt, wobei es, wenn $\alpha \neq o$ ist, einen einzigen offenen Weg mit maximaler Länge von α nach o gibt.

Für eine Kette K gilt: $FIRST(K) = \alpha$ und $LAST(K) = o$. Wenn eine Kette K einen einzigen Knoten v enthält, dann gilt: $FIRST(K) = LAST(K) = v$.

Zur Bestimmung der Inzidezabbildung einer Kette (und der im nächsten eingeführten DSL-Abschnitte) eignet sich die Darstellung dieses Teilgraphen als eine Produktion einer kontextfreien Grammatik, in der die Elemente aus V (Operationen, Tests und Strukturknoten) Terminalzeichen sind. Die Produktion PRD1 verdeutlicht die Inzidenzabbildung φ' einer Kette.

PRD1: Kette ::= Operation / Operation Kette

wobei Operation ein Knoten $v \in O_p$ ist. Sei gemäß der Produktion PRD1 eine Kette K_1 mit $K_1 = v\ K_2$, dann wird φ' durch die Gleichung (a) bestimmt, so daß:

(a) $SUC(v) = \{FIRST(K_2)\}$

Die Gleichungen (b) stellen die Anfangs- und Endknoten α und o von K_1 dar:

(b) $FIRST(K_1) = v$
$LAST(K_1) = LAST(K_2)$

3.1.2 *DSL-Graph*

Ein DSL-Graph ist ein eindeutiger vollständiger Verhaltensgraph, dessen Abschnitte ausschließlich durch die Produktion PRD2 erzeugt werden. Ein DSL-Graph besteht nur aus diesen Abschnitten. Die Menge V der erlaubten DSL-Knoten ist in [CaWe85a] definiert.

PRD2: DSL-Abschnitt ::= Kette / IF-Abschnitt / FOR-Abschnitt /
WHILE-Abschnitt / DO-UNTIL-Abschnitt /
CASE-Abschnitt / FORK-JOIN-Abschnitt /
DSL-Abschnitt DSL-Abschnitt

Es sei ein DSL-Abschnitt A mit $A = A_1A_2$, wobei A_1 und A_2 auch DSL-Abschnitte seien. Die Inzidenzabbildung von A wird durch die Gleichungen (c) angegeben:

(c) $FIRST(A) = FIRST(A_1)$
$LAST(A) = LAST(A_2)$
$FIRST(A_2) \in SUC(LAST(A_1))$

Bild 3.1.1 zeigt die verschiedenen Abschnitte eines DSL-Graphen. Aus diesem Bild lassen sich die entsprechenden Produktionen leicht herauslesen. Um deren Strukturen näher betrachten zu können, werden im folgenden die Produktionen und die Inzidenzabbildungen dreier DSL-Abschnitte eingeführt.

PRD3: IF-Abschnitt ::= 'IF' [Kette] Test
'THEN' DSL-Abschnitt
['ELSE' DSL-Abschnitt]
FI-Knoten

PRD4: WHILE-Abschnitt ::= 'WHILE' [Kette] Test
'DO' [DSL-Abschnitt]
OD-Knoten

PRD5: FORK-JOIN-Abschnitt ::= FORK-Knoten
(DSL-Abschnitt // ',')
JOIN-Knoten

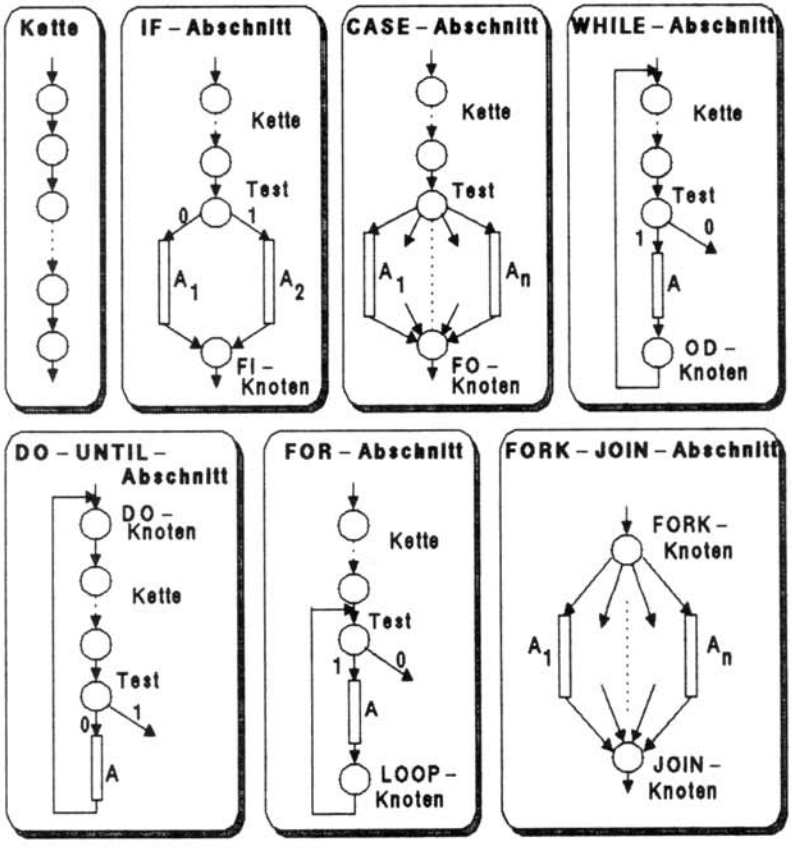


Bild 3.1.1 - Abschnitte eines DSL-Graphen (A = Abschnitt)

Die Elemente zwischen '' sind lediglich verknüpfende Elemente in der Struktur, welche die Notationen der DSL-Sprache widerspiegeln. Es gibt keinen Knoten im Verhaltensgraphen, der diese Elemente darstellt. Die Elemente zwischen [ ] sind in diesen Abschnitten optional. Bei der Produktion PRD3 ist der Test ein Knoten $t \in T_p$. Bei der Produktion PRD5 bedeutet (// ',') Wiederholung mit ',' als Trennzeichen.

Es sei laut Produktion PRD3 ein IF-Abschnitt A wie im folgenden angegeben, wobei K eine Kette, t ein Test und A_1 und A_2 DSL-Abschnitte seien. Die Inzidenzabbildung in A wird durch die Gleichungen (d) beschrieben:

A = 'IF' K t 'THEN' A_1 'ELSE' A_2 FI-Knoten
(d) FIRST(A) = FIRST(K)
SUC (LAST(K)) = {t}
SUC (t) = {FIRST(A_1), FIRST(A_2)}
SUC (LAST(A_1)) = {FI-Knoten}
SUC (LAST(A_2)) = {FI-Knoten}
LAST(A) = FI-Knoten

In (d) ist der FI-Knoten ein Strukturknoten, und die Kanten e_1 und e_2, die durch $\varphi(e_1)$ = (t,FIRST(A_1)) und $\varphi(e_2)$ = (t,FIRST(A_2)) bestimmt werden, sind bedingte Kanten. Falls K nicht vorkommt, dann ist FIRST(A) = t.

Es sei W laut PRD4 ein WHILE-Abschnitt. Die Inzidenzabbildung in W wird durch die Gleichungen (e) bestimmt.

W = 'WHILE' K t 'DO' A OD-Knoten
(e) FIRST(W) = FIRST(K)
SUC (LAST(K)) = {t}
SUC (t) = {FIRST(A),α_*}
SUC (LAST(A)) = {OD-Knoten}
SUC (OD-Knoten) = FIRST(W)
LAST(W) = t

In (e) ist der OD-Knoten ein Strukturknoten und der Knoten α_* ein Nachfolger von t außerhalb von W. Gemäß PRD2 und der Gleichung (c) ist dieser Knoten der Anfangsknoten des nächsten Abschnitts.

Es sei F laut PRD5 ein FORK-JOIN-Abschnitt, wobei alle A_i, $0 < i \leq n$, DSL-Abschnitte sind. Die Abschnitte A_i sind nebenläufig (s.u. Abs. 3.1.3) Die Inzidenzabbildung in F wird durch die Gleichungen (f) bestimmt.

F = FORK-Knoten $A_1,\ldots,A_n$ JOIN-Knoten
(f) FIRST(F) = FORK-Knoten
SUC (FORK-Knoten) = {FIRST(A_1),...,FIRST(A_n)}
SUC (LAST(A_1)) = {JOIN-Knoten}
SUC (LAST(A_n)) = {JOIN-Knoten}
LAST(F) = JOIN-Knoten

In (f) sind die FORK- und JOIN-Knoten Strukturknoten. Die Kanten zwischen dem FORK-Knoten und seinen Nachfolgern sind unbedingt (DEF. 3.1.1). Der

FORK-Knoten ist der einzige Knoten des DSL-Graphen, der mit mehreren Nachfolgern durch unbedingte Kanten verbunden ist. Im Gegensatz zum FI-Knoten, der nach der Aktivierung eines einzigen Abschnitts A_1 oder A_2 eines IF-Abschnitts erreicht wird, ist der JOIN-Knoten erst nach der Aktivierung aller nebenläufigen Abschnitte A_i zu erreichen.

Wie die Produktionen PRD2 bis PRD5 zeigen, können DSL-Abschnitte aus anderen DSL-Abschnitten entstehen. Ein Abschnitt, der andere enthält und von keinem ein Element ist, wird als Sequenz bezeichnet.

DEF. 3.1.8 - Sequenz

> Ein Abschnitt S eines Verhaltensgraphen D ist eine Sequenz, falls es keinen Abschnitt A, $A \subset D$, mit $S \subset A$ und $S \neq A$ gibt.

Eine Sequenz, die nur aus DSL-Abschnitten besteht, wird DSL-Sequenz genannt. Eine DSL-Sequenz stellt das Verhalten eines imperativen Teils eines DSL-Programms dar. Ein DSL-Graph kann mehrere Sequenzen enthalten. Zwei Sequenzen eines DSL-Graphen sind nicht zusammenhängend, d.h. es gibt keinen Pfad, der irgendein Paar v, z von Knoten, $v \in S_1$, $z \in S_2$, $S_1 \neq S_2$, enthält.

Es wird angenommen, daß jede Sequenz durch eine implizite Kante e_{imp} ergänzt wird, so daß $\varphi(e_{imp}) = (o,\alpha)$ ist, wobei o der Endknoten und α der Anfangsknoten der Sequenz ist. Diese Kante heißt implizit, weil sie nicht automatisch aus der Beschreibung eines imperativen Teils erzeugt wird. Der DSL-Übersetzer läßt absichtlich einen Nachfolger des Endknotens der entsprechenden Sequenz offen. Das dient im DSL-Graphen als Kennzeichen des strukturellen Endes einer Sequenz (diese Strategie vereinfacht das Durchlaufen eines DSL-Graphen), bedeutet aber nicht, daß der imperative Teil nur einmal auszuführen ist. Die Ergänzung durch e_{imp} macht eine Sequenz eindeutig und vollständig. Diese Eigenschaften erlauben die Synthese der Steuerung für Sequenzen.

3.1.3 *Relationen in einem DSL-Graphen*

Die Nachfolger-Relation genügt zur vollständigen Beschreibung der Ordnungsbeziehung zwischen den Knoten eines DSL-Graphen offensichtlich nicht. Vielmehr muß für alle Paare (v,z) von Knoten erkennbar sein, ob sie geordnet, alternativ oder nebenläufig sind. Für die folgenden Definitionen gilt: $e_{imp} \notin E$, wobei E der Menge der Kanten eines DSL-Graphen entspricht.

DEF. 3.1.9 - Geordnet

Gegeben seien zwei Knoten v und z in einer Sequenz. Wenn es einen Pfad von v nach z (oder von z nach v) gibt, dann sind v und z geordnet und man schreibt (v; z) (bzw. (z; v)).

Wäre $e_{imp} \in E$, dann wären alle Knoten geordnet. In den Tat sind alle Knoten in einer eindeutigen und vollständigen Sequenz (d.h. einer mit e_{imp}) von jedem anderen aus durch das mehrfache Durchlaufen des Graphen erreichbar. Von Interesse sind hier jedoch nur die Relationen, die ohne das mehrfache Durchlaufen bestimmt werden können.

DEF. 3.1.10 - Alternativ

Gegeben seien zwei nicht geordnete Knoten v, z in einer Sequenz. Wenn es einen Knoten $t \in T_p$ (Test), einen Pfad C_1 von t nach v, dessen Menge der Knoten P_{tv} ist, und einen Pfad C_2 von t nach z, dessen Menge der Knoten P_{tz} ist, gibt, so daß $P_{tv} \cap P_{tz} = \{t\}$ ist, dann sind v und z alternativ. Die alternative Relation wird durch (v<> z) dargestellt.

DEF. 3.1.11 - Nebenläufig

Gegeben seien zwei nicht geordnete Knoten v, z in einer Sequenz. Wenn es einen FORK-Knoten f, einen Pfad C_1 von f nach v, dessen Menge der Knoten P_{fv} ist, und einen Pfad C_2 von f nach z, dessen Menge der Knoten P_{fz} ist, gibt, so daß $P_{fv} \cap P_{fz} = \{f\}$ ist, dann sind v und z nebenläufig. Die nebenläufige Relation wird durch [v, z] dargestellt.

Das folgende Beispiel veranschaulicht diese Relationen.

Beispiel 3.1.1:

Bild 3.1.2 zeigt eine Sequenz mit elf Knoten (α-a & o-k), und die Tabelle 3.1.1 beinhaltet die aus diesem Bild gewonnenen gültigen Relationen zwischen den Knoten.

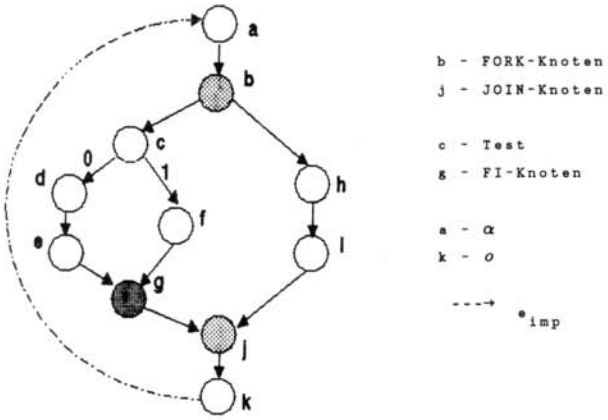


Bild 3.1.2 - Beispiel einer Sequenz

	a	b	c	d	e	f	g	h	i	j	k
a		;	;	;	;	;	;	;	;	;	;
b			;	;	;	;	;	;	;	;	;
c				;	;	;	;	,	,	;	;
d					;	◇	;	,	,	;	;
e						◇	;	,	,	;	;
f							;	,	,	;	;
g								,	,	;	;
h									;	;	;
i										;	;
j											;
k											

Tabelle 3.1.1 - Die im Bild 3.1.2 vorhandenen Relationen

Hätte man e_{imp} bei der Definition 3.1.9 genommen, wären alle Knoten im Bild 3.1.2 geordnet (z.B. f und e durch einen Pfad von f über g, j, k, a, b, c, d bis e). Dadurch wäre diese Relation bedeutungslos, da sie für alle Paare von Knoten auftreten würde.

Ende des Beispiels.

Die Relationen 'geordnet', 'alternativ' und 'nebenläufig' sind auch zwischen Abschnitten eines DSL-Graphen gültig. Zwei zusammenhängende Abschnitte A, B sind geordnet, wenn LAST(A), FIRST (B) geordnet sind. Sie sind alternativ, wenn FIRST(A), FIRST(B) alternativ sind, und letztlich sind sie nebenläufig, falls FIRST(A), FIRST(B) nebenläufig sind.

Da Sequenzen nicht zusammenhängende Teilgraphen sind, sind diese Definitionen nicht auf Sequenzen anwendbar. Die Relationen zwischen Sequenzen sind lediglich durch ihre Aktivität bestimmt. Zwei Sequenzen sind nebenläufig, wenn sie gleichzeitig aktiv sein können. Wenn sie nie gleichzeitig aktiv vorkommen, dann sind sie alternativ.

Mit Hilfe der oben definierten Relationen läßt sich der Ablaufgraph einer Sequenz einführen. Der Ablaufgraph einer Sequenz enthält die Anfangs- und Endknoten dieser Sequenz, aber keine alternativen Knoten.

DEF. 3.1.12 - Ablaufgraph

Ein Teilgraph L einer Sequenz S ist ein Ablaufgraph von S, falls gilt:

a) $\forall$ v,z $\in$ L, (v;z) oder [v,z]
b) $\forall$ L' $\subset$ S, falls L' ein Ablaufgraph ist und L $\subset$ L', dann ist L = L'.

Ein Ablaufgraph ist ein unvollständiger Verhaltensgraph. Jeder Knoten $t \in T_p$ in einem Ablaufgraphen hat einen einzigen Nachfolger, womit die Relation 'alternativ' ausgeschlossen ist. Ein Ablaufgraph stellt den Ablauf einer Sequenz dar. Ein Ablauf ist das Durchlaufen eines Ablaufgraphen.

Beispiel 3.1.2:

Bild 3.1.3 zeigt eine Sequenz mit zwei IF-Abschnitten und die daraus abgeleiteten Ablaufgraphen. Die Ablaufgraphen zeigen deutlich die vier vorhandenen Möglichkeiten zum Durchlaufen dieser Sequenz.

Im Bild 3.1.3 ist t ein Test. FI steht für FI-Knoten, der Endknoten eines IF-Abschnitts.

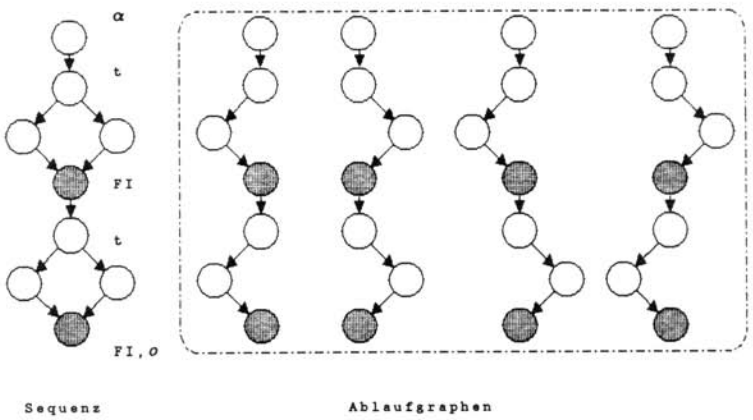


Bild 3.1.3 - Die Ablaufgraphen einer Sequenz

Ende des Beispiels.

Die Relationen 'geordnet' und 'nebenläufig' spezifizieren eine schwache zeitliche Einordnung zwischen den Knoten eines DSL-Graphen. Die geordnete Relation (v;z) bedeutet in diesem Fall, daß die Aktivierung des Knotens z entweder gleichzeitig oder nach der Aktivierung des Knotens v vorkommen kann. Die nebenläufige Relation [v,z] bedeutet, daß v und z zeitlich unabhängig voneinander sind, das heißt, die Aktivierung des Knotens z kann entweder gleichzeitig, vor oder nach der Aktivierung des Knotens v vorkommen. Die endgültige zeitliche Relation wird bei der Synthese bestimmt und ist dann fest. Wenn die Synthese für ein bestimmtes Paar von Knoten einer Sequenz eine zeitliche Einordnung wählt, dann gilt diese zeitliche Einordnung für alle Abläufe dieser Sequenz.

3.2 *Das BABEL-Modell*

Das von BABEL verwendete Modell ist eine Erweiterung des DSL-Modells und dient zur Beschreibung und Synthese von digitalen Systemen. Der Unterschied zwischen beiden Modellen liegt in der strukturierten Beschreibung von Nebenläufigkeit auf der Schicht der Prozesse (Tabelle 1.2), die das BABEL-Modell erlaubt.

3.2.1 *Modellelemente*

Ein durch ein Programm vollständig beschriebenes digitales Projekt wird im Rahmen des Modells als *System* bezeichnet. Jedes Beispiel des Abschnitts 2.1 ist in diesem Sinne ein System.

Die Hauptkomponenten eines Systems sind Prozesse. Ein Prozeß ist entweder ein terminaler Prozeß oder eine Menge von Prozessen, die gemäß einem vordefinierten Interaktionsmuster zusammenarbeiten. Im Mehrprozessorsystem des Beispiels 4 (Abschnitt 2.1.2) ist jeder Prozessor ein solcher Prozeß.

DEF. 3.2.1 - Terminaler Prozeß

> Ein terminaler Prozeß (tP) ist eine eindeutige und vollständige Sequenz oder eine Menge von solchen Sequenzen, wobei zwischen den Sequenzen dieser Menge ausschließlich die alternative Relation gilt.

Im Gegensatz zu DSL-Sequenzen (d.h. Sequenzen, die nur aus DSL-Abschnitten entstehen) ist die Sequenz (oder sind die Sequenzen) eines terminalen Prozesses fähig, mit Sequenzen anderer Prozesse zu interagieren. Ein terminaler Prozeß wird durch ein DSL-Programm mit einer einzigen Sequenz oder mehreren alternativen Sequenzen beschrieben, wobei diese Sequenzen durch neue Abschnitte und zusätzliche Knoten, die Interaktionen mit anderen Prozessen erlauben, ergänzt sind.

Im Rahmen dieser Arbeit werden hauptsächlich terminale Prozesse behandelt, die eine einzige Sequenz enthalten. Alternative Sequenzen dienen im allgemeinen dazu, um Unterbrechungen darzustellen (im Anhang A2 steht ein Beispiel dafür). Nebenläufige Sequenzen innerhalb eines terminalen Prozesses sind nicht erlaubt, weil für deren Interaktionen kein Konzept bzw. keine Notation vorhanden ist. Dafür gibt es die Möglichkeit, was eigentlich das Ziel von BABEL ist, die Nebenläufigkeit von Sequenzen durch mehrere Prozesse zu beschreiben.

Terminale Prozesse sind gemäß der Definition (Eindeutigkeit und Vollständigkeit der Sequenz) immer aktiv, was in diesem Zusammenhang bedeutet, daß es keine Operation in der Sequenz gibt, welche die Aktivität des Prozesses endgültig beendet (im Gegensatz zu allgemeinen Anwenderprogrammen, die, wenn sie nicht gerade fehlerhaft sind, immer terminieren). Die implizite Kante e_{imp} bestimmt, daß nach der Aktivierung des Endknotens o der Anfangsknoten α aktiviert wird.

Die folgenden Produktionen PRD6 und PRD7 stellen deutlich das rekursive

Konzept von Prozeß dar. Dieses Konzept wird später mit den in den nächsten Abschnitten eingeführten Elementen nocheinmal definiert (siehe DEF 3.3.2).

PRD6: Prozeß ::= tP / (Prozeß // ',')

PRD7: tP ::= (Sequenz // '<>')

Die Notation (a//b) stellt die Wiederholung von a mit b als vorhandene Relation zwischen den Elementen a dar. Das Element a kommt mindestens einmal vor. tP steht für terminalen Prozeß; ',' für nebenläufige Aktivitäten und '<>' für alternative Aktivitäten.

Die einzige gültige Relation zwischen den Prozessen eines Systems ist die nebenläufige Relation. Damit nebenläufige Prozesse zusammenarbeiten können, muß irgendeine Art von Interaktion zwischen ihnen stattfinden. Aus den beiden allgemeinen Möglichkeiten, Interaktion zwischen Prozessen darzustellen, der Kommunikation und der Kooperation [AnSc83], wurde das Kooperationsmodell gewählt. Die Gründe hierfür sind vor allem folgende:

- Das Kommunikationsmodell verlangt wegen der Speicherung der ausgetauschten Nachrichten im Sender- und Empfänger-Prozeß mehr Speicherplatz als das Kooperationsmodell.

- Die Übertragung von Nachrichten im Kommunikationsmodell ist zeitaufwendig.

Das Kooperationsmodell bestimmt, daß Prozesse keine direkte Verbindung zum Austausch von Nachrichten besitzen. Wenn sie Informationen austauschen müssen, findet das durch die geregelte Nutzung von gemeinsamen Mitteln statt.

DEF. 3.2.2 - Gemeinsames Mittel

> Jeder von mehreren Prozessen benutzbare Informationsträger (Variablen oder Anschlüsse) ist ein gemeinsames Mittel (GM).

Als Variablen verstehen sich hier die Elemente, seien es Register, Speicher, usw., die aus der VARIABLE-Vereinbarung des DSL-Programms realisiert werden können. Als Anschlüsse verstehen sich die unter INTERFACE vereinbarten Ausgangs- und Eingangsanschlüsse.

3.2.2 *Schutzmechanismus*

Wegen der unbestimmten gleichzeitigen Abläufe von mehreren Prozessen können dabei Widersprüche im Zustand der gemeinsamen Mittel auftreten. Ein Widerspruch deutet hier auf einen unbestimmten oder indeterministischen Inhalt eines gemeinsamen Mittels hin. Diese Widersprüche haben Zugriffskonflikte oder die Verflechtung der Operationen mehrerer Prozesse (interleaving) als Ursachen.

a) Zugriffskonflikte:

- Gleichzeitiges Lesen und Schreiben eines Speicherelements (unbestimmter vergänglicher Wert).
- Gleichzeitiges Schreiben eines Speicherelements (unbestimmtes Ergebnis).

b) Verflochtene Operationen mehrerer Prozesse:

Jeder Prozeß führt eine oder mehrere Operationen auf ein GM aus. Angenommen sei, daß Zugriffskonflikte dabei nicht auftreten. Widersprüche können trotzdem verursacht werden, indem die Operationen verschiedener Herkunft beim gleichzeitigen Ablauf mehrerer Prozesse in so einer Reihenfolge vorkommen, daß eine Operation die andere täuscht bzw. verbirgt. Ein typisches Beispiel dazu: ein Prozeß P1 liest ein GM und entscheidet gemäß seinem Inhalt, eine Operation (z.B. eine Addition) auf dieses GM auszuführen. Inzwischen hat der Prozeß P2 dessen Inhalt schon geändert. Zum Schluß entspricht der Inhalt des GM's weder dem von P1 noch dem von P2 erwarteten Wert.

Die serielle Ausführung der einzelnen Operationen, die auf ein GM zugreifen, kann in der Regel Zugriffskonflikte vermeiden. Diese hat allein aber keine Wirkung auf die durch verflochtene Operationen verursachten Widersprüche. Hier muß vielmehr eine ganze Reihe von Operationen eines einzigen Prozesses ununterbrochen ausgeführt werden. Beide, serielle und unterbrochene Ausführung von Operationen, lassen sich durch gegenseitigen Ausschluß der Prozesse erreichen.

Wegen des möglichen Auftretens von Widersprüchen können die Prozesse nicht ohne Einschränkungen auf ein gemeinsames Mittel zugreifen. Die Prozesse müssen zuerst den Zugriff zum gewünschten GM anfordern und dann warten, bis sie das Zugriffsrecht zu diesem GM erhalten. Ein impliziter Schutzmechanismus (Bild 3.2.1) vermeidet Widersprüche im Zustand der gemeinsamen Mittel durch die zweckmäßige Zuteilung des Zugriffsrechts an die verschiedenen Prozesse. Damit werden Operationen (oder Reihen von

Operationen), die Widersprüche verursachen können, mit gegenseitigem Ausschluß ausgeführt.

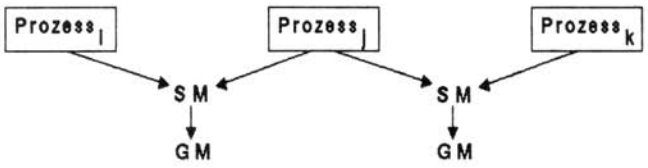


Bild 3.2.1 - Vermeidung von Widersprüchen durch implizite Schutzmechanismen (SM)

3.2.3 *Aktion*

Ein Prozeß greift auf ein GM zu, indem er Operationen und Tests, die dieses Mittel als Operanden benutzen, durchführt. Die Menge dieser Operationen und Tests, die zusammen vorkommen, entspricht einem Abschnitt im Verhaltensgraphen eines Prozesses. Dieser Abschnitt unterscheidet sich von den DSL-Abschnitten dadurch, daß sein erster Knoten eine Zugriffsanforderung auf ein GM darstellt.

DEF. 3.2.4 - Aktion

> Eine Aktion AGM ist ein Abschnitt, der den Anforderungsknoten a_k enthält. Für eine Aktion gilt:
>
> - Es gibt in einer Aktion ein und nur ein a_k und es gilt: FIRST (AGM) = a_k.

Da der Anforderungsknoten die erste Operation einer Aktion ist, verursacht die Aktivierung einer Aktion erst die Anforderung der entsprechenden GM. Der Ablauf des Prozesses wird danach verzögert bis der Prozeß das Zugriffsrecht zu diesem GM bekommt.

Die Operationen und Tests, die ein GM benutzen, sind durch die Aktionen deutlich gekennzeichnet. Um Widersprüche zu vermeiden, reicht es, wenn der Schutzmechanismus jeweils nur einem Prozeß Zugriff zu einem GM erlaubt. Der Prozeß führt dann eine Aktion durch, und während diese Aktion aktiv ist, bekommt kein anderer Prozeß das Zugriffsrecht zu diesem GM. Durch diesen gegenseitigen Ausschluß von Aktionen garantiert der

Schutzmechanismus, daß keine Widersprüche bei der Nutzung eines GM's auftreten.

Manche Aktionen verursachen keinen Widerspruch, auch wenn sie gleichzeitig aktiv sind. Diese Aktionen sind zum Beispiel solche, die den Zustand des GM's nicht ändern, oder solche, die auf verschiedene Informationsträger eines GM's zugreifen. Diese Aktionen nennt man konfliktfreie Aktionen. Konfliktfreie Aktionen können gleichzeitig aktiv sein. Erst die konfliktbehafteten Aktionen sollen mit gegenseitigem Ausschluß durchgeführt werden.

Mit der Einführung des Abschnitts 'Aktion', lassen sich die Abschnitte eines terminalen Prozesses (sog. 'BABEL-Abschnitte') vollständig definieren:

PRD8: BABEL-Abschnitt ::= DSL-Abschnitt / Aktion /
BABEL-Abschnitt BABEL-Abschnitt

BABEL-Abschnitte werden alle Abschnitte genannt, die in einem terminalen Prozeß vorkommen können, einschließlich DSL-Abschnitten (siehe PRD2).

3.2.4 *Verteilung des Zugriffsrechts*

Die von den Prozessen geforderten Aktionen werden gemäß einem für jedes System generierten Konfliktmodell und einem Prioritätsschema durchgeführt. Das Konfliktmodell eines GM's weist auf die Paare der konfliktfreien Aktionen hin und wird aus den Eigenschaften dieser Aktionen abgeleitet. Das Prioritätsschema enthält die Zugriffsprioritäten der Aktionen und der Prozesse, die zur Spezifikation des Systems gehören.

Bei gleichzeitigen Zugriffsanforderungen mehrerer Prozesse bestimmt der Schutzmechanismus gemäß dem Konfliktmodell und dem Prioritätsschema, welche Prozesse das Zugriffsrecht bekommen. Wenn die von den Prozessen geforderten Aktionen konfliktfrei sind, bekommen alle Prozesse das Zugriffsrecht. Wenn sie dagegen nicht konfliktfrei sind, bekommt nur der Prozeß mit der momentan höchsten Priorität das Zugriffsrecht.

3.2.5 *Verhalten einer Aktion*

Haben ein Prozeß oder eine Gruppe von Prozessen, die konfliktfreie Aktionen verlangen, das gemeinsame Mittel belegt, so verteilt der Schutz-

mechanismus das Zugriffsrecht erst wieder, wenn das Mittel freigegeben wird.

Ein Prozeß, der ein gemeinsames Mittel belegt, kann das Mittel in einem nicht verfügbaren Zustand vorfinden (z.B. ein Puffer ist beim 'Holen' leer - sehe Abs. 2.1.1). In diesem Fall gibt der Prozeß das Mittel frei und wartet auf eine Änderung des Zustandes. Dieses Verhalten wird durch einen Warte-Abschnitt dargestellt.

PRD9: Warte-Abschnitt ::= 'IF' Test 'THEN' Warte-Knoten
['ELSE' DSL-Abschnitt]
FI-Knoten

Es sei W laut PRD9 ein Warte-Abschnitt, wobei t ein Test und A ein DSL-Abschnitt sei. Die Inzidenzabbildung in W wird durch die Gleichungen (g) angegeben:

W = 'IF' t 'THEN' Warte-Knoten 'ELSE' A FI-Knoten

(g) FIRST(W) = t
SUC (t) = {Warte-Knoten, FIRST(A)}
SUC (Warte-Knoten) = {FI-Knoten}
SUC (LAST(A)) = {FI-Knoten}
LAST(W) = FI-Knoten

Der Operand des Tests t wird Bedingung genannt. Diese Bedingung stellt einen Zustand des gemeinsamen Mittels dar. Der Warte-Knoten (w_k) ist ein Struktur-Knoten und hat die Eigenschaft, den Ablauf des Prozesses zu unterbrechen.

Warte-Abschnitte sind nur innerhalb von Aktionen zu finden. Die folgende Produktion beinhaltet diese Eigenschaft:

PRD10: Aktion ::= Anforderungsknoten
Aktion-Rumpf

Aktion-Rumpf ::= Rumpf-Abschnitt

Rumpf-Abschnitt ::= Warte-Abschnitt / DSL-Abschnitt
Rumpf-Abschnitt Rumpf-Abschnitt

Wenn die Bedingung des Warte-Abschnitts zum Verlassen des Warte-Zustands erfüllt wird, befindet sich der wartende Prozeß in einer der folgenden Lagen:

a) *Der Nachfolger von w_k ist nicht der Endknoten der Aktion:* Der Prozeß muß, nachdem die Bedingung erfüllt wird, noch auf das Zugriffsrecht zum entsprechenden GM warten. Wenn mehrere Prozesse auf dieselbe Bedingung warten, und die Aktionen, die die Prozesse abschließen müssen, nicht konfliktfrei sind, bekommt nur ein Prozeß das Zugriffsrecht, und alle anderen auf diese Bedingung wartenden Prozesse bleiben im Wartezustand.

b) *Der Nachfolger von w_k ist der Endknoten der Aktion:* Der Prozeß benötigt keinen Zugriff zum GM mehr und kehrt nach der Änderung des Bedingungszustandes zu seinem normalen Ablauf zurück, ohne auf das Zugriffsrecht zu warten.

Bild 3.2.2 zeigt eine Skizze der allgemeinen Zustände eines Prozesses während seines Ablaufs.

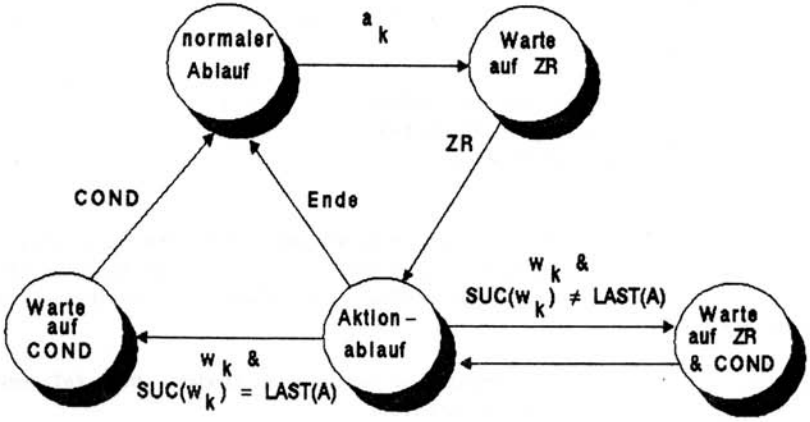


Bild 3.2.2 - Prozeßablauf mit Ausführung einer Aktion. Es bedeuten: COND - Wartebedingung, ZR - Zugriffsrecht, A - Aktion

3.3 *Interprozeßmodul*

Jedes gemeinsame Mittel sowie der implizite Mechanismus, der dieses Mittel schützt, und die Aktionen über diesem Mittel werden in einem spezifischen Element des BABEL-Modells, dem Interprozeßmodul (*interprocess module*) zusammengeschlossen.

3.3.1. *Konzept*

DEF. 3.3.1 - Interprozeßmodul

Ein Interprozeßmodul umfaßt das Tupel:

IPM = (GM, AGMs, SM)

mit den Komponenten:
GM : Gemeinsames Mittel
AGMs : Menge der Aktionen
SM : Impliziter Schutzmechanismus

Das Bild 3.3.1 zeigt eine Skizze eines IPM's, der zwei Aktionen enthält.

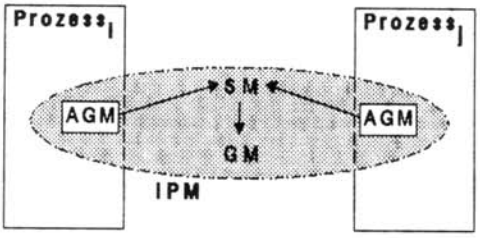


Bild 3.3.1 - Darstellung eines Interprozeßmoduls

Mit der Einführung des neuen Elements IPM wird 'Prozeß' wie folgt definiert:

DEF. 3.3.2 - Prozeß

Ein Prozeß ist ein Tupel: Prozeß = (IPMs, Ps)
mit den Komponenten:

Ps : Menge von Prozessen
IPMs : Menge von Interprozeßmoduln

Die Prozesse eines Prozesses werden hier auch als Unterprozesse

bezeichnet. Falls |Ps| = 1, dann gilt |IPMs| = 0, d.h., in einem Prozeß mit einem einzigen Unterprozeß sind Interprozeßmoduln bedeutungslos.

Der oberste Prozeß bei dieser rekursiven Definition, d.h. der Prozeß, der von keinem anderen ein Unterprozeß ist, wird System genannt. Jedes gemeinsame Mittel in einem System gehört zu einem Interprozeßmodul. Der Interprozeßmodul vermeidet durch den impliziten Schutzmechanismus Widersprüche im Zustand der gemeinsamen Mittel. Im Gegensatz zu Prozessen ist ein IPM eine passive Komponente eines Systems, die erst aktiv wird, wenn ein Prozeß eine Aktion von diesem IPM verlangt.

3.3.2 *Interprozeßmoduln und Monitore*

Das hier dargestellte Konzept des Interprozeßmoduls lehnt sich an die bekannte Monitordefinition (Dijkstra, Brinch Hansen, Hoare) [AnSc83] und die Variante von Kessels [Kess77] an.

Hoares Monitor

Ein Monitor ist ein abstrakter Datentyp, wobei die Datenstruktur ein gemeinsames Mittel darstellt. Die Prozeduren dieses Datentyps, welche die gültigen Operationen über der Datenstruktur einschließen, sind mit gegenseitigem Ausschluß auszuführen, d.h. höchstens eine Prozedur darf in jedem Moment aktiv sein. Der gegenseitige Ausschluß von Prozeduren garantiert, daß kein Widerspruch auftritt.

Kessels Monitor

Die Variante von Kessels bestimmt, daß die Bedingungen zum Verlassen eines Monitors durch eine Warte_Operation (wait) nur aus lokalen Variablen des Monitors zu erzeugen sind. Diese Bedingungen müssen als solche vereinbart werden. Die Signalisierung einer Bedingungsänderung erfolgt automatisch (ohne die Anwendung des Signalbefehls, wie in Hoares Monitor).

Diese Eigenschaften erleichtern die Implementierung dieses Konzepts, ohne seine Anwendbarkeit einzuschränken. Deshalb wurde Kessels Monitor als Muster für den IPM verwendet.

Der Hauptunterschied zwischen den Konzepten des BABEL-Interprozeßmoduls und des oben definierten Monitors und seiner Varianten ist die Regel des gegenseitigen Ausschlusses. Im klassischen Konzept sind alle Aktionen mit

gegenseitigem Ausschluß durchzuführen. Diese strenge Regel wurde hier gelockert, um von der Parallelität in der Hardware besser profitieren zu können. Dabei bestimmt das IPM-Konzept, daß zwei oder mehrere konfliktfreie Aktionen, wenn sie gleichzeitig verlangt werden, auch gleichzeitig aktiviert werden können. Die Ausschlußregel ist nur für nicht konfliktfreie Aktionen gültig.

Die Lockerung der Ausschlußregel benötigt die Erzeugung eines Konfliktmodells und die Synthese eines angepaßten und optimierten Schutzmechanismus für jedes neue System. Es ist zu erwähnen, daß der Aufwand bei der Analyse der Konfliktfälle und der Erzeugung des Konfliktmodells durch den Gewinn an Geschwindigkeit mehr als ausgeglichen wird.

Das IPM-Konzept erlaubt auch klassiche Synchronisationsprobleme, wie in den Beispielen 6 (2.1.4) und 7 (2.1.5) dargestellt, leichter in BABEL als durch Monitore zu beschreiben (Anhang A1).

Bemerkung:

> Im Softwarebereich sind konfliktfreie Aktionen (Prozeduren) seltener als in der Hardware. Das liegt daran, daß die meisten Datenstrukturen sich in einem gemeinsamen Speicher befinden, wobei die impliziten Belegungen der Speicherregister (Adressen und Daten) schon Konflikte verursachen könnten, wenn sie nicht streng sequentiell stattfinden würden. Deshalb bringt die Vernachlässigung der Ausschlußregel im Softwarebereich keinen signifikanten Gewinn.

3.4 *Hierarchie im BABEL-Modell*

Da ein Prozeß Unterprozesse enthalten kann, bilden sich im BABEL-Modell mehrere hierarchische Ebenen. Das Interaktionsschema in dieser Hierarchie, d.h. die Art, wie Prozesse aus verschiedenen Ebenen miteinander interagieren, wird im folgenden eingeführt.

3.4.1 *Hierarchische Gliederung*

Die Hierarchie in BABEL wird durch einen Baum dargestellt, bei dem die Väter nichtterminale Prozesse (oder Systeme) und die Blätter Interprozeßmoduln und terminale Prozesse sind. Jede Stufe dieses Baums stellt eine hierarchische Ebene dar.

In BABEL sind nur terminale Prozesse in der Lage, durch die Nutzung von IPM miteinander zu interagieren. Die nichtterminalen Prozesse stellen lediglich eine Menge von IPM und Prozessen dar, deren Aktivität durch die Aktivität ihrer Unterprozesse bestimmt wird. Deshalb sind nur die Blätter des Hierarchie-Baumes in Interaktionen verwickelt.

Die Interaktionen zwischen den terminalen Prozessen erfolgen nach dem folgenden Schema:

- Prozesse desselben Vaters kooperieren miteinander durch einen IPM, welcher ebenfalls Sohn dieses Vaters ist.

- Prozesse verschiedener Väter, P_i und P_j, kooperieren miteinander durch einen IPM einer höheren Ebene, dessen Vater ein gemeinsamer Vorfahr von P_i und P_j ist.

Das folgende Beispiel eines hierarchischen Systems zeigt ein solches Interaktionsschema.

Beispiel 3.4.1:

Sei ein System:

$$S_0 = (IPMs_1, Ps_1),$$
$$Ps_1 = \{S_1, S_2, proc_1\},$$
$$IPMs_1 = \{ipm_a, ipm_b\}$$

Sei außerdem:

$$S_1 = (IPMs_2, Ps_2),$$
$$Ps_2 = \{proc_2, S_3\},$$
$$IPMs_2 = \{ipm_c\}$$

und:

$$S_2 = (IPMs_3, Ps_3),$$
$$Ps_3 = \{proc_5, proc_6\},$$
$$IPMs_3 = \{ipm_e\}$$

Sei letztlich:

$$S_3 = (IPMs_4, Ps_4),$$
$$Ps_3 = \{proc_3, proc_4\},$$
$$IPMs_3 = \{ipm_d\}$$

wobei $proc_1$, $proc_2$, $proc_3$, $proc_4$, $proc_5$ und $proc_6$ terminale Prozesse seien, und ipm_a, ipm_b, ipm_c, ipm_d, und ipm_e Interprozeßmoduln.

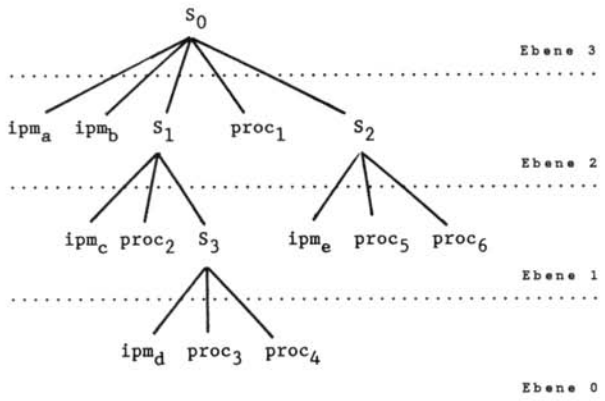


Bild 3.4.1 - Hierarchie-Baum eines Systems

- Die terminalen Prozesse von S_3 ($proc_3$ und $proc_4$) kooperieren durch ipm_d.
- Die Prozesse $proc_3$ und $proc_4$ können mit $proc_2$ durch ipm_c kooperieren.
- Die Interaktion zwischen $proc_3$ and $proc_6$ kann durch ipm_a oder durch ipm_b stattfinden.

Ende des Beispiels.

3.4.2 *Spezifikation der Priorität*

Falls mehrere Prozesse Aktionen eines gegebenen IPM's gleichzeitig verlangen, bestimmt das Prioritätschema, welcher Prozeß das Zugriffsrecht bekommt. Dieses Schema ist nur gültig, falls die verlangten Aktionen nicht konfliktfrei sind. Das Prioritätsschema umfaßt die in BABEL vom Benutzer spezifizierten Prioritäten von Aktionen und Prozessen. Eine Prioritätsspezifikation wird durch die folgende Halbordnung dargestellt:

$$Pri.spec = (\Pi, \geq)$$

wobei Π eine Folge der durch die Relation $\geq$ geordneten Prioritäten von Aktionen oder Prozessen ist.

Die Priorität von Aktionen ist der von Prozessen überlegen: unabhängig von der spezifizierten Prozeßpriorität bekommt immer der Prozeß, der die

Aktion mit der höchsten Priorität verlangt, das Zugriffsrecht zum entsprechenden IPM.

Die Prozeßpriorität stellt die Priorität der Prozesse bezüglich eines IPM's dar. Diese Spezifikation gilt nur, wenn die von diesen Prozessen verlangten Aktionen die gleiche Priorität haben. Alle Prozesse, sei es ein terminaler Prozeß oder nicht, sind bezüglich jedes von ihnen benutzbaren IPM's durch ihre Priorität geordnet. Dabei enthält jeder nichtterminale Prozeß die Spezifikation der Priorität seiner Unterprozesse.

$$Pri.spec = (\Pi, \geq)$$
$$\Pi = (\Pi proc_0, \ldots, \Pi proc_n), \quad \Pi proc_{i-1} \geq \Pi proc_i, \; 1 \geq i \geq n$$

$\Pi proc_i$ stellt für einen terminalen Prozeß $proc_i$ dessen Priorität und für einen nichtterminalen Prozeß die Folge Π seiner Unterprozesse dar. Da aber letztlich nur terminale Prozesse interagieren können, muß eine Folge Π, die nur aus Prioritäten von terminalen Prozessen entsteht, für jeden IPM bekannt sein. Das wird erreicht, indem man in der Folge Π des Vaters dieses IPM's alle Prioritäten von nichtterminalen Prozessen durch die Folge Π' ihres Unterprozesses ersetzt. Die Folge Π' übernimmt in Π den Platz der Priorität des Vaters der Prozesse in Π'.

Das folgende Beispiel erläutert die Prioritätsspezifikation von Prozessen in der BABEL-Hierarchie.

Beispiel 3.4.2:

Seien für das Beispiel 3.4.1 die folgenden Spezifikationen bezüglich des ipm_a gültig:

a) $\Pi S_0 \Rightarrow \Pi S_1 > \Pi S_2 > \Pi proc_1$
b) $\Pi S_1 \Rightarrow \Pi S_3 > \Pi proc_2$
c) $\Pi S_2 \Rightarrow \Pi proc_6 > \Pi proc_5$
d) $\Pi S_3 \Rightarrow \Pi proc_3 > \Pi proc_4$

Nach dem Einsetzen von (d) in (b), und (c) und (b) in (a), erhält man die Relation (e), die die vollständige Spezifikation der Priorität aller terminalen Prozesse bezüglich des ipm_a enthält.

e) $\Pi S_0 \Rightarrow \Pi proc_3 > \Pi proc_4 > \Pi proc_2 > \Pi proc_6 > \Pi proc_5 > \Pi proc_1$

Ende des Beispiels.

3.4.3 *Der BABEL-Verhaltensbaum*

Wegen der im BABEL-Modell vorhandenen Hierarchie wird das Verhalten eines Systems durch einen Baum von Verhaltensgraphen dargestellt. Dieser Baum wird als Verhaltensbaum bezeichnet.

Der Verhaltensbaum entspricht dem Hierarchie-Baum eines Systems, wobei die Blätter (Interprozeßmoduln und terminale Prozesse) durch den entsprechenden Verhaltensgraphen ersetzt sind.

a) Verhaltensgraph eines terminalen Prozesses

Das Verhalten jedes terminalen Prozesses wird durch einen Verhaltensgraphen dargestellt, der einer BABEL-Sequenz entspricht. Eine BABEL-Sequenz ist eine Sequenz (DEF. 3.1.7), die aus BABEL-Abschnitten (PRD6) besteht. In einer BABEL-Sequenz kommen Aktionen nicht direkt vor. Sie sind in diesem Graphen durch den entsprechenden Anforderungsknoten vertreten. Der Rest der Aktion, der Aktions-Rumpf, kommt als Teilgraph des Verhaltensgraphen in dem entsprechenden Interprozeßmodul vor.

b) Verhaltensgraph eines Interprozeßmoduls

Ein Interprozeßmodul wird durch einen nicht zusammenhängenden Verhaltensgraphen dargestellt. Jeder Teilgraph in einem IPM-Verhaltensgraphen ist ein Aktions-Rumpf (PRD10).

Der Schutzmechanismus in jedem IPM ist implizit, und es gibt deshalb dafür keine Darstellung auf der Verhaltensebene.

4. Die Sprache BABEL

BABEL ist eine Entwurfssprache zur Beschreibung und Synthese von digitalen Systemen. BABEL enthält die vollständige DSL-Grammatik und zusätzlich dazu Sprachnotationen, die es erlauben, Prozesse, Interprozeß-moduln, Aktionen, Wartebedingungen und andere Elemente des in Kapitel 3 beschriebenen Modells darzustellen.

Die Sprache wird in den folgenden Abschnitten durch syntaktische Definitionen (SD) 2.1 beschrieben und ausschließend durch die BABEL-Darstellung der Beispiele des Abschnitts 2.1 erläutert.

4.1 *Die Grammatik von BABEL*

Für die Darstellung der Grammatik gilt die ALADIN-Notation [Kast79]. Es folgt eine Kurzfassung dieser Notation:

a ::= b	Ableitung
'X'	Wortsymbole oder Sonderzeichen
[a]	Optional
a / b	Entweder a oder b
(a) *	Die Anzahl der Anwendungen der syntaktischen Konstruktion a ist gleich oder größer Null
(a) +	Die Anzahl der Anwendungen der syntaktischen Konstruktion a ist größer als Null.
(a//b)	Wiederholung mit Trennzeichen, wobei a mindestens einmal vorkommt. Wenn es mehrere a gibt, stellt b ein Trennzeichen dar.

Das Bild 4.1 zeigt die Ableitungswege der Elemente der BABEL-Grammatik. Die Elemente zwischen Punkten sind BABEL-spezifisch, alle anderen Elemente gehören zu DSL [CaWe85a].

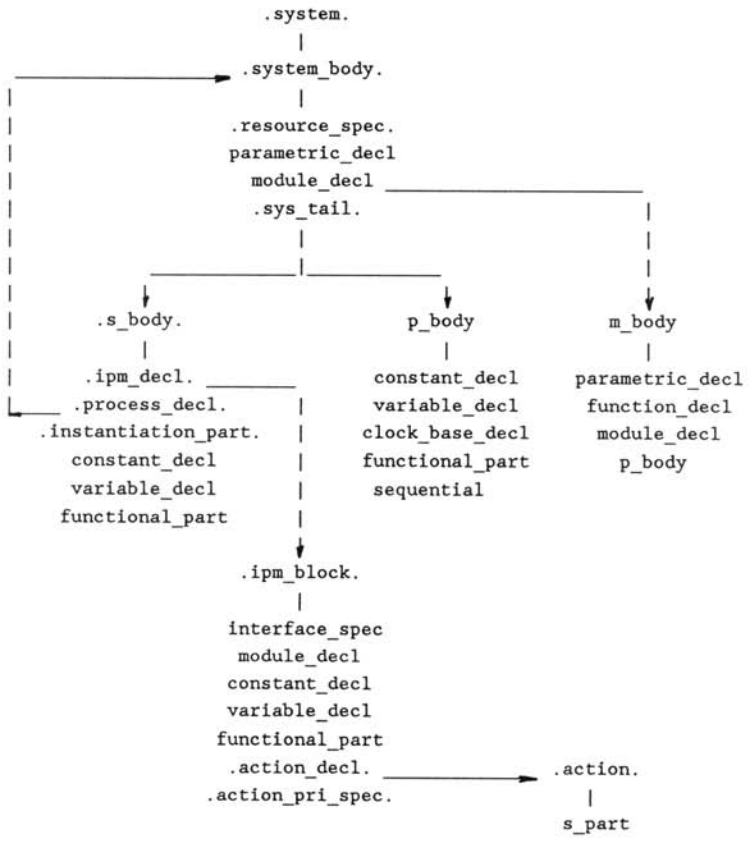


Bild 4.1 - Ableitungswege der BABEL-Grammatik

4.2 *Beschreibung eines Systems*

Das gewünschte System wird durch ein BABEL-Programm dargestellt. Dieses Programm beschreibt jeden Prozeß durch ein DSL-Programm, das durch Notationen zur Darstellung der Elemente des nebenläufigen Modells ergänzt wurde. Für die Beschreibung eines Systems gelten die folgenden syntaktischen Definitionen:

```
SD1:   system ::= 'SYSTEM'  identifier  ';'
                  system_body
                 'END' [ 'SYSTEM' ] '.'

SD2:   system_body ::= [ resource_spec   ]
                         parametric_decl
                       [ module_decl     ]
                         s_body  /  p_body

SD3:   s_body ::=   ipm_decl
                    process_decl
                    instantiation_part
                    association_part
                    priority_spec
                  [ constant_decl  ]
                  [ variable_decl  ]
                  [ functional_part]
```

In SD1 ist *identifier* ein beliebiger eindeutiger Name. In SD2 ist *p_body* der Rumpf eines terminalen Prozesses, ein DSL-Programm-Rumpf, der mit einer zusätzlichen Sprachnotation versehen ist, welche die Aktivierung von Aktionen erlaubt. In SD2 und SD3 sind *parametric_decl*, *module_decl*, *constant_decl* und *variable_decl* syntaktische Elemente von DSL mit gleicher semantischer Bedeutung wie in DSL.

Der *functional_part* eines *s_body* (SD3) unterscheidet sich allerdings von einem DSL-*functional_part*. Die Steuerbefehle (START, STOP, CONTINUE) kommen in *s_body* nicht vor und eine BABEL-spezifische Art von Operanden, die übertragenen Operanden, (s.u. Abschnitt 4.2.5) darf benutzt werden. Die Steuerbefehle können in BABEL auch die Aktivierung von Sequenzen innerhalb eines *p_body* steuern, nicht aber die von Systemen und Prozessen, da diese Elemente immer aktiv sind. Es ist zu beachten, daß die BABEL-Grammatik genau der DSL-Grammatik entspricht, wenn *resource_spec* und *s_body* im *system_body* weggelassen werden.

Grammatik eines DSL-Programms:

```
system_body ::= parametric_decl   [module_decl]   p_body
```

Die folgenden syntaktischen Elemente sind BABEL-spezifisch und dienen der System-Beschreibung:

- Typvereinbarungen (*process_decl* & *ipm_decl*)
- Verkörperung der Interprozeßmoduln und der Prozesse (*instantiation_part*)

- Verbindung der Prozesse mit den Interprozeßmoduln (*association_part*)
- Darstellung der Zugriffspriorität (*priority_spec*)

Diese syntaktischen Elemente werden in den folgenden Abschnitten eingeführt.

4.2.1 *Typvereinbarungen*

Ein BABEL_Programm verfügt über zwei mögliche Typenvereinbarungen: die erste zur Beschreibung von Interprozeßmoduln (SD4) und die zweite zur Beschreibung von Prozessen (SD5). Alle verschiedenen Prozesse und Interprozeßmoduln eines Systems sind als Typen vollständig zu beschreiben.

```
SD4:  ipm_decl::= ('INTERPROCESS' ['MODULE'] identifier ';'
                   ipm_block
                  'END' ['INTERPROCESS'] '.' ) *

SD5:  process_decl ::= ('PROCESS' identifier ';'
                        system_body
                       'END' ['PROCESS'] '.' ) +
```

Ein *process_decl* (SD5) führt zu *system_body* (SD2) und dadurch zu *s_body* (SD3) oder *p_body* (DSL-Element). Das entspricht den Definitionen von Abschnitt 3.2.1, in denen steht, daß ein Prozeß entweder ein terminaler Prozeß (*p_body*) oder eine Menge von Prozessen (hier auch als System (*s_body*) bezeichnet) sein kann.

Ein *system_body*, der sich zu einem *s_body* ableiten läßt, bildet in der Beschreibung eine neue hierarchische Ebene.

4.2.2 *Verkörperung der Interprozeßmoduln und Prozesse*

Die Verkörperung (instantiation) beschreibt, wieviele Instanzen jedes Typs von Interprozeßmoduln (SD6) und Prozessen (SD7) für das System benötigt werden. Alle verkörperten Prozesse sind automatisch aktiv.

In einer bestimmten hierarchischen Ebene eines Systems dürfen nur Prozesse und Interprozeßmoduln verkörpert werden, die zu dieser Ebene gehören.

```
SD6:  creation_part   ::= 'CREATE' (id_group // ',' ) ';'
SD7:  activation_part ::= 'ACTIVATE' ( id_group // ',' ) ';'
SD8:  id_group        ::= identifiers ':' identifier
SD9:  identifiers     ::= ( identifier // ',' )
```

In SD8 sind *identifiers* verkörperte IPM-Namen (CREATE) oder verkörperte Prozeß-Namen (ACTIVATE) und *identifier* ist ein IPM-Typ (CREATE) oder ein Prozeß-Typ (ACTIVATE), der innerhalb des laufenden Prozesses vereinbart wurde.

4.2.3 *Verbindung*

SD10 beschreibt die nötige Verbindung zwischen den verkörperten Prozessen und Interprozeßmoduln. Eine vollständige Spezifikation der Verbindung soll für jeden in der laufenden hierarchischen Ebene verkörperten Prozeß erfolgen, und die Reihenfolge der Vereinbarungen der Außenmittel (s.u. 4.4.1) eines jeden Prozesses muß der Reihenfolge der Interprozeßmoduln in der Verbindungspezifikation dieses Prozesses entsprechen.

```
SD10:  association_part ::= 'ASSOCIATE' (as_spec//',') ';'
SD11:  as_spec::= identifiers 'WITH' (identifier // 'AND')
```

In SD11 sind *identifiers* Namen von verkörperten Prozessen und *identifier* ist der Name eines verkörperten IPM's.

4.2.4 *Darstellung der Prozeßpriorität*

Die syntaktische Definition SD12 beschreibt die Priorität jedes Prozesses bezüglich eines bestimmten Interprozeßmoduls. Dieser Interprozeßmodul ist hier entweder ein in der laufenden hierarchischen Ebene verkörperter IPM oder ein in höherer Ebene vereinbartes Außenmittel (Abschnitt 4.4.1), dessen Verkörperungsname noch nicht bekannt ist.

```
SD12:  priority_spec ::= 'SET' ( priority // ',' ) ';'
SD13:  priority ::= 'TO' identifier
                    'PRIORITY' (pr_group // '>')
SD14:  pr_group ::=  ( identifier // '=' )
```

In SD13 ist *identifier* ein verkörperter IPM Name oder ein Außenmittel einer höheren Ebene, und in SD14 ist *identifier* ein verkörperter Prozeßname.

Wenn für einen IPM alle Prozesse die gleiche Priorität haben, entfällt die Prioritätsdarstellung für diesen IPM.

4.2.5 *Übertragene Operanden*

BABEL verfügt über eine Notation, die Referenzen zu Anschlüssen (*interfaces*) von Prozessen und Interprozeßmoduln innerhalb des *functional_part* eines *s_body* (Systemrumpf) erlaubt. Die Prozesse und Interprozeßmoduln müssen in der laufenden Ebene verkörpert sein. Mit Hilfe der übertragenen Operanden (*transported operands*) kann ein externes *interface* mit mehreren internen *interfaces* verbunden werden, wenn nötig auch über eine dazwischengeschaltete logische Struktur.

Ein übertragener Operand wird wie folgt dargestellt:

```
identifier '.' identifier
```

Der erste *identifier* ist ein verkörperter Prozeß- oder IPM-Name. Der zweite *identifier* ist ein *interface*-Name, der innerhalb des ersten Elements vereinbart ist.

4.2.6 *Beispiele zur Systembeschreibung*

Drei Programmabschnitte zeigen die Anwendung der Notationen von Abschnitt 4.2. Das erste Programm (Beispiel 4.2.1) ist die Beschreibung des Beispiels 1 aus Abschnitt 2.1.1.

Beispiel 4.2.1:

```
SYSTEM  example_1;

   INTERPROCESS buffer; ( ipm_block ) END.  (* Typ-Vereinbarung *)
   PROCESS   unit_1;    ( p_body )    END.
   PROCESS   unit_2;    ( p_body )    END.
   PROCESS   unit_3;    ( p_body )    END.
```

```
    CREATE     IN_BUFFER, OUT_BUFFER : buffer;   (* Verkörperung     *)

    ACTIVATE   ERZEUGER    : unit_1,
               TRANSFORMER : unit_2,
               VERBRAUCHER : unit_3;

    ASSOCIATE  ERZEUGER    WITH IN_BUFFER,       (* Verbindungen     *)
               TRANSFORMER WITH IN_BUFFER  AND OUT_BUFFER,
               VERBRAUCHER WITH OUT_BUFFER;
END.
```

Ende des Beispiels.

Der Interprozeßmodul *buffer* stellt die nötigen gemeinsamen Mittel der drei Einheiten dar. Dieses System (Bild 4.2) benötigt zwei Elemente von Typ *buffer*. Das erste zur Wechselbeziehung zwischen ERZEUGER und TRANSFORMER (IN_BUFFER), das zweite zur Wechselbeziehung zwischen TRANSFORMER und VERBRAUCHER (OUT_BUFFER).

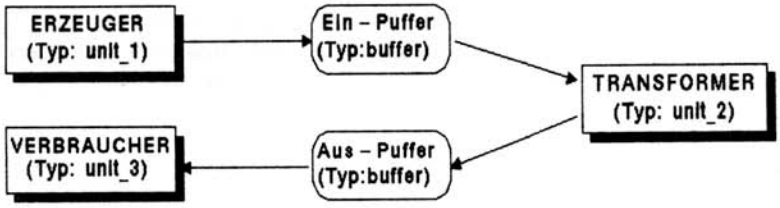


Bild 4.2 - System mit drei verkörperten Prozessen und zwei verkörperten IPM

Der nächste Programmabschnitt (Beispiel 4.2.2) enthält die Beschreibung des Beispieles 3 aus Abschnitt 2.1.2 mit einer Darstellung der Priorität. Das Bild 4.3 zeigt eine Skizze dieses Systems. In diesem Bild sind *fetch* und *execute* verkörperte Prozesse und *pc_and_queue* und *main_mem* verkörperte IPM.

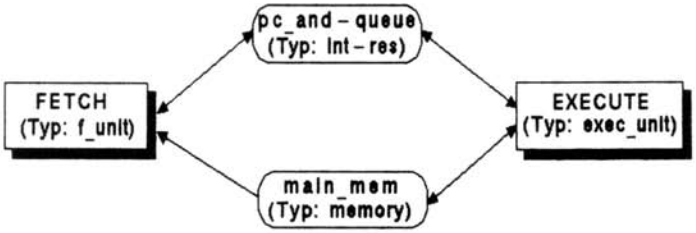


Bild 4.3 - 'fetch look ahead' Prozessor

Beispiel 4.2.2:

```
SYSTEM  processor;

        INTERPROCESS  memory;    ( ipm_block )  END.
        INTERPROCESS  int_res;   ( ipm_block )  END.
        PROCESS       f_unit;    ( p_body )     END.
        PROCESS       exec_unit; ( p_body )     END.

        (* instantiation   *)
        CREATE  pc_and_queue : int_res,  main_mem : memory;
        ACTIVATE  execute  : exec_unit,  fetch : f_unit;

        (* association     *)
        ASSOCIATE  execute WITH pc_and_queue AND main_mem,
                   fetch   WITH pc_and_queue AND main_mem;

        (* access priority to global memory      *)
        SET TO  main_mem  PRIORITY  execute > fetch;
END.
```

Ende des Beispiels.

Die Zeile:

```
SET TO  main_mem  PRIORITY  execute > fetch;
```

bestimmt, daß im Fall mehrerer gleichzeitig erfolgender Anforderungen des IPM's *main_mem* der Prozeß *execute* das Zugriffsrecht auf diesen IPM erhalten soll.

Das Beispiel 4.2.3, ein Programmabschnitt aus der Beschreibung eines Mehrprozessorsystems (siehe Abschnitt 2.1.2), zeigt die Anwendung von übertragenen Operanden. Die vollständige Beschreibung dieses Beispiels ist im Anhang A2 angegeben. Das Mehrprozessorsystem besteht aus drei gleichen Prozessoren P1, P2, und P3 und einem gemeinsamen Speicher (Bild 4.4). Jeder Prozessor ist ein verkörperter Prozeß, dessen Typ einem wie oben beschrieben System entspricht. Der Speicher ist hier nicht mehr lokal jedem Prozessor zugeordnet, sondern er ist ein gemeinsames globales Mittel. Im Bild 4.4 sind die gemeinsamen Mittel nicht dargestellt. Jeder Prozessor enthält zwei nebenläufige Einheiten (*fetch und execute*), die im Bild 4.4 als untergeordnete Elemente vom Prozessor P1 zu sehen sind. Das gesamte System hat zwei gemeinsame Anschlüsse: ein Eingangssignal (*rst*) zu allen drei Prozessoren und ein Ausgangssignal (*sys_error*), das aus drei Signalen (eines aus jedem Prozessor) entsteht. Alle anderen Anschlüsse des Systems sind im Bild 4.4 nicht dargestellt.

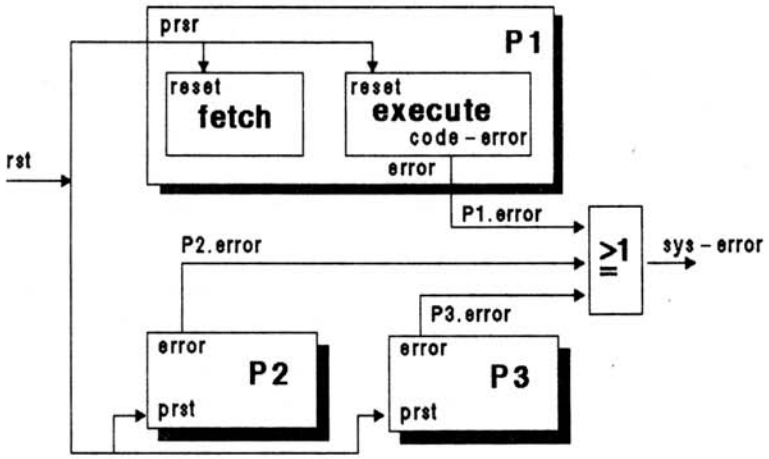


Bild 4.4 - Mehrprozessorsystem mit übertragenen Operanden. P2 und P3 haben die Struktur des Prozessors P1.

Im Bild 4.4 sind die Anschlüsse in den drei Beschreibungsebenen zu sehen. In der tiefsten Ebene findet man die Anschlüsse der verkörperten Prozesse *execute* (code_error, reset) und *fetch* (reset). Im *functional_part* des Prozeß_Typs *processor* werden diese Anschlüsse mit den Anschlüssen der zweiten Ebene (error, prst) verbunden. Nach der dreifachen Verkörperung

von *processor* werden dann die verkörperten Anschlüsse der zweite Ebene im *functional_part* des Systems mit dessen Anschlüssen (rst und sys-error) verbunden.

Beispiel 4.2.3:

```
SYSTEM multiprocessor;
   INTERFACE rst:INPUT; sys_error:OUTPUT;    (* system interfaces *)
   :
   (* processor   -    subprocesses : f_unit & exec_unit      *)
   PROCESS processor;
     :                                        (* processor interfaces *)
     INTERFACE error:OUTPUT; prst:INPUT;
     :
     PROCESS f_unit;                      (* fetch instruction unit *)
        :
        INTERFACE reset:INPUT;               (* f_unit interfaces *)
        :
     END PROCESS.                                    (* end f_unit *)
     PROCESS exec_unit;                           (* execution unit *)
        :                                   (* exec_unit interfaces *)
        INTERFACE code_error: OUTPUT; reset: INPUT;
        :
     END PROCESS.                                  (* end exec_unit *)
     (*       instantiation inside a processor-type            *)
     ACTIVATE execute : exec_unit, fetch : f_unit;
     :
     FUNCTIONAL
        (* interface connection: execute and fetch with processor *)
        error := execute.code_error,
        execute.reset := prst,       fetch.reset := prst
     END;
   END PROCESS.                                     (* end processor *)
 ACTIVATE P1,P2,P3 : processor;       (* instantiation in system *)
 :
 FUNCTIONAL
   (*     connection between processor and system interfaces     *)
   sys.error := P1.error OR P2.error OR P3.error,
   P1.prst := rst,   P2.prst := rst,   P3.prst := rst
 END;
END SYSTEM.
```

Ende des Beispiels

4.3 *Beschreibung eines Interprozeßmoduls*

Ein Interprozeßmodul ist eine Art abstrakter Datentyp, bei dem Zugriffe zu lokalen Informationsträgern, in diesem Fall gemeinsame Mittel genannt, nur durch Aktionen möglich sind. Der Rumpf eines IPM's ist durch die syntaktische Definition SD15 beschrieben:

```
SD15:   ipm_block ::= ( interface_spec ) *
                      [ module_decl     ]
                      [ constant_decl   ]
                        variable_decl
                      [ functional_part ]
                        action_decl
                      [ action_pri_spec ]
```

Ein Interprozeßmodul besteht aus folgenden Elementen:

- Vereinbarung und Initialisierung der gemeinsamen Mittel (*interface_spec & variable_decl*)
- Vereinbarung der Bedingungen (*variable_decl*)
- Beschreibung der Struktur (*functional_part*)
- Beschreibung der Aktionen über GM (*action_decl*)
- Aktionsprioritätsabschnitt (*action_pri_spec*)

4.3.1 *Vereinbarung*

a) *Vereinbarung der gemeinsamen Mittel:* Die Vereinbarung aller gemeinsamen Mittel folgt der allgemeinen Vereinbarungsregel von Informationsträgern der DSL-Sprache.

b) *Bedingungsvereinbarung:* Eine Bedingung (*condition*) entspricht einem Zustand der gemeinsamen Mittel. Die Vereinbarung einer Bedingung setzt nur den Namen der Bedingung fest. Der Inhalt wird erst in der Strukturbeschreibung (*functional_part*) durch eine *boolean expression* bestimmt. Die Bedingungsvereinbarung ist nur innerhalb eines Interprozeßmoduls erlaubt.

c) *Initialisierung der gemeinsamen Mittel:* Die Anfangswerte der zum Interprozeßmodul gehörigen gemeinsamen Mittel werden durch die DSL-Notation 'INITIAL' festgesetzt. Der Anfangswert bestimmt den Zustand der gemeinsamen Mittel nach dem Auftreten von 'power on' oder 'reset'.

4.3.2 *Strukturbeschreibung*

Die Strukturbeschreibung (*functional_part*) innerhalb eines IPM's erfolgt wie die Strukturbeschreibung in DSL. Die Steuerbefehle (START, STOP, CONTINUE), die zur Aktivierung von Sequenzen dienen, kommen aber nicht in einem IPM vor, da es dort keine Sequenz gibt.

In der Strukturbeschreibung eines IPM's werden alle vereinbarten Bedingungen durch je eine *boolean expression* über gemeinsamen Mitteln spezifiziert. Weil eine Bedingung einem Zustand der gemeinsamen Mittel entspricht (Kessels Konzept), enthält die Spezifikation einer Bedingung nur gemeinsame Mittel und Konstanten (keine exklusiven Mittel der Prozesse und auch keine Parameter der Aktionen sind zulässig).

4.3.3 *Aktionsbeschreibung*

SD16 erlaubt die Beschreibung der gewünschten Aktionen über den vereinbarten gemeinsamen Mitteln. Jede Aktion in einem Interprozeßmodul ist ein abgeschlossenes Element ohne Bezug auf externe Aktionen oder Informationsträger.

Vereinbarungen von zusätzlichen Informationsträgern sind innerhalb einer Aktion nicht möglich. Die Variablen, die als Quelle- oder Ziel-Operanden vorkommen können, sind nur die gemeinsamen Mittel und die Aktionsparameter. Die vereinbarten Bedingungen dürfen nur als Test-Variable eines IF-WAIT-Befehls vorkommen. Der Bedingungswert ändert sich indirekt durch die Änderung eines GM, welches zur Spezifikation der Bedingung gehört.

```
SD16:   action_decl ::= ( 'ACTION' identifier ';'
                        (  interface_spec ) *
                         'SEQUENCE'
                          s_part
                         'END' ['SEQUENCE']
                         'END' ['ACTION']  ';' ) +
```

In SD16 ist *identifier* ein beliebiger innerhalb eines IPM's eindeutig erkennbarer Name, und *interface_spec* erlaubt die Spezifikation der Parameter einer Aktion.

Im *s_part* einer Aktion kommt die zusätzliche Notation IF-WAIT zum bedingten Verlassen einer Aktion wegen eines ungünstigen Zustands der gemeinsamen Mittel vor.

```
'IF' condition    'THEN WAIT'
                [ 'ELSE'  s_part ]
'FI'
```

Wenn die als *condition* vereinbarte Bedingung wahr ist, muß der Prozeß, der ein Zugriffsrecht auf dieses gemeinsame Mittel hat, auf die Änderung dieser Bedingung warten.

4.3.4 *Prioritätsbeschreibung*

Durch SD17 werden die Prioritäten der Aktionen innerhalb eines Interprozeßmoduls beschrieben.

```
SD17:   action_pri_spec ::= 'PRIORITY'
                            ( pr_group // '>' ) ';'
SD14:   pr_group        ::= ( identifier // '=' )
```

In SD14 ist *identifier* jetzt der Name einer Aktion.

Wenn mehrere Aktionen eines IPM's mit unterschiedlichen Prioritäten gleichzeitig gefordert werden, erhält der Prozeß, der die Aktion mit der höchsten Priorität fordert, das Zugriffsrecht. Die Spezifikation der Prioritäten für Aktionen ist nur sinnvoll, wenn die Aktionen konfliktbehaftete Aktionen sind.

4.3.5 *Beispiel eines Interprozeßmoduls*

Der folgende Programmabschnitt führt den Interprozeßmodul *buffer* des Beispiels 1 aus Abschnitt 2.1.1 ein. Zu diesem IPM gehören zwei Aktionen, die es erlauben, Daten in dem gemeinsamen Mittel *storage* einzufügen oder aus *storage* herauszuholen. Weil *storage* leer sein kann, wenn eine Information zu holen ist, oder voll, wenn man Informationen einfügen will, sind

zwei Bedingungen *full* und *empty* vereinbart.

Beispiel 4.3.1:

```
INTERPROCESS MODULE   buffer;
                                          (* common resources *)
   VAR storage : ARRAY[15..0]  OF LOGICAL(7..0);
       head, tail, size :   LOGICAL(3..0) INITIAL = 0;
       full, empty :  CONDITION;         (* cond. declaration *)

   FUNCTIONAL  full  := (size = 15),       (* condition spec. *)
               empty := (size = 0)
   END;

   ACTION  deposit;
      INTERFACE char(7..0): INPUT;
      SEQUENCE
         IF full THEN  WAIT  FI;          (* conditional wait *)
         FORK storage[tail] := char, size := size +1  JOIN;
         tail := tail + 1
      END SEQUENCE
   END ACTION;

   ACTION  get;
      INTERFACE char(7..0): OUTPUT;
      SEQUENCE
         IF empty THEN  WAIT  FI;         (* conditional wait *)
         FORK char := storage[head], size := size -1 JOIN;
         head := head + 1
      END SEQUENCE
   END ACTION;
END INTERPROCESS.
```

Ende des Beispiels.

4.4 *Beschreibung eines Prozesses*

Ein Prozeß mit Unterprozessen ist selbst ein System, für welches alle Notationen des Abschnitts 4.2 noch gültig sind.

Ein Prozeß ohne Unterprozesse (terminaler Prozeß) ist einem DSL_Programm ähnlich. Der Unterschied zwischen beiden besteht darin, daß ein terminaler

Prozeß die Möglichkeit hat, Aktionen über gemeinsamen Mitteln zu verlangen.

4.4.1 *Außenmittel*

Die Interprozeßmodultypen, die ein Prozeß benötigt, werden durch die Vereinbarung von Außenmitteln (SD18) bestimmt. Jedes spezifizierte Außenmittel entspricht einem Exemplar eines bestimmten IPM-Typs. Von jedem IPM-Typ dürfen mehrere Außenmittel definiert werden.

```
SD18:   resource_spec ::= 'RESOURCE' ( id_group ';' )+
SD8:    id_group      ::= identifiers ':' identifier
```

In SD8 sind *identifiers* Außenmittelnamen und *identifier* ein IPM-Typ. Dieser IPM-Typ muß zur gleichen hierarchischen Ebene wie der laufende Prozeß gehören.

Die Außenmittel sind nur innerhalb desjenigen Prozesses gültig, in dem sie vereinbart sind, einschließlich aller Unterprozesse. Sie bestimmen, zu welchem IPM eine verlangte Aktion gehört. Der folgende Programmabschnitt (Beispiel 4.4.1) zeigt die Außenmittelvereinbarung des Prozesses *unit_2* des Systems *example_1*.

Beispiel 4.4.1:

```
PROCESS   unit_2;
  RESOURCE ib, ob:buffer;   (* Außenmittelvereinbarung *)
     :
END PROCESS.
:
CREATE     IN_BUFFER, OUT_BUFFER : buffer;
ACTIVATE   TRANSFORMER : unit_2;
           :
ASSOCIATE  TRANSFORMER WITH IN_BUFFER AND OUT_BUFFER;
:
```

Ende des Beispiels.

ib und *ob* sind Außenmittel vom Typ *buffer*. Die Verbindung bestimmt für den verkörperten Prozeß TRANSFORMER, daß (*ib* = *IN_BUFFER*) und (*ob* = *OUT_BUFFER*) sein soll. Das gilt nur für TRANSFORMER. Wenn es mehrere Prozesse vom Typ *unit_2* gäbe, könnten *ib* und *ob* auch anderen verkörperten IPM entsprechen.

4.4.2 *Terminaler Prozeßrumpf*

Ein terminaler Prozeßrumpf (SD19) ist ein DSL-Programm-Rumpf mit einer zusätzlichen Sprachnotation zur Anforderung von Aktionen.

```
SD19:   p_body ::= [ constant_decl   ]
                   variable_decl
                   clockbase_decl
                 [ functional_part ]
                 ( sequential_part )*

SD20:   sequential_part ::= seq_symb identifier';'
                            s_part
                           'END' [ seq_symb ] ';'

SD21:   s_part ::= s_statement [ ';' ]     /
                   s_statement ';' s_part

SD22:   seq_symb::= 'SEQUENTIAL' / 'IMPERATIVE' / 'SEQUENCE'
```

In SD19 sind *constant_decl*, *variable_decl*, *clockbase_decl* und *functional_part* syntaktische Konstruktionen von DSL mit gleicher semantischer Bedeutung wie dort. Der *s_part* (SD21) leitet sich aus allen *s_statement*'s von DSL ab. Zusätzlich tritt eine Anweisung auf, die PERFORM Anweisung (SD23), welche die Anforderung von Aktionen erlaubt.

```
SD23:   s_statement ::=   'PERFORM'
                          identifier '.' identifier
                          [ '(' expressions ')' ]
                          / DSL_s_statement
```

Der erste *identifier* in SD23 ist der Name eines gültigen Außenmittels. Der zweite ist der Name einer seiner Aktionen; *expressions* sind Parameter der Aktion.

Die gültigen Außenmittel sind die im laufenden Prozeß vereinbarten Außenmittel sowie alle Außenmittel, die in höheren Ebenen der Hierarchie vereinbart wurden. Das folgende Beispiel (Bild 4.5) zeigt die Gültigkeitsbereiche von Außenmitteln in einem System mit mehreren hierarchischen Ebenen.

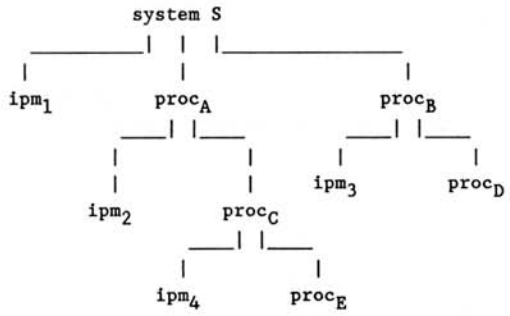


Bild 4.5 - System mit mehreren hierarchischen Ebene

Die Prozesse $proc_A$ und $proc_B$ kennen den Interprozeßmodul-Typ ipm_1 und können ihn als Außenmittel vereinbaren. Der Prozeß $proc_D$ kennt den Interprozeßmodul-Typ ipm_3, aber nicht ipm_2.

Innerhalb des Prozesses $proc_E$ können die Außenmittel von $proc_E$ (vom Typ ipm_4), von $proc_C$ (vom Typ ipm_2) und von $proc_A$ (vom Typ ipm_1) benutzt werden.

4.5 *Der BABEL-Übersetzer*

Der Übersetzer der Sprache BABEL [Webe85] hat die folgenden Ziele:

- Überprüfung der Syntax der Sprache,
- Erzeugung des BABEL-Verhaltensbaumes,
- Erzeugung von Listen und Tabellen, welche die in der Beschreibung vorhandenen Spezifikationen (Verbindungen, Verkörperungen, Priorität, ...) wiedergeben.

Außerdem prüft der Übersetzer mehrere semantische Randbedingungen wie zum Beispiel den Gültigkeitsbereich von Außenmitteln im Hierarchie-Baum, die Gültigkeit der übertragenen Operanden und die Vollständigkeit der Spezifi-

kationen von Verbindungen und Prioritäten.

Ein BABEL-Übersetzer läuft auf einer Rechneranlage Siemens 7.751 unter dem Betriebsystem BS 2000. Er wurde mit Hilfe des GAG-Systems [KHZ82] entwickelt. Alle syntaktischen Elemente aus DSL sind aus dem DSL-Übersetzer übernommen und an BABEL angepaßt.

Der BABEL-Übersetzer erzeugt zwei Ausgabedateien. Die erste besteht aus Fehlermeldungen und die zweite enthält den BABEL-Verhaltensbaum und die ergänzenden Tabellen und Listen. Das Ausgabeformat des Verhaltensbaumes entspricht für jedes der Blätter des Hierarchie-Baumes dem Ausgabeformat eines DSL-Graphen [CaWe85a] mit den zusätzlichen Darstellungen für die BABEL-spezifischen Knoten w_k (Warte-Knoten) und a_k (Anforderungsknoten). Die Ausgabeformate dieser Knoten werden im folgenden eingeführt.

Bild 4.6 zeigt die graphische Darstellung des folgenden Warte-Abschnitts in einem Aktions-Rumpf (Warte-Knoten sind ausschließlich innerhalb von Warte-Abschnitte zu finden):

```
IF condition-name THEN WAIT FI;
```

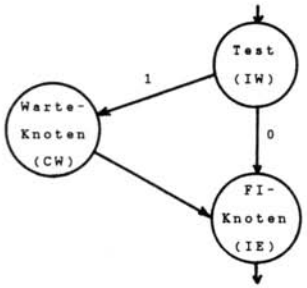


Bild 4.6 - Graphische Darstellung eines Warte-Abschnitts (IW, CM und IE sind Knotenkennzeichen)

In Bild 4.7 wird ein Teil des Verhaltensgraphen einer Aktion gezeigt, in dem dieser Warte-Abschnitt zu finden ist. Die Felder mit 'X' oder '*' haben für den entsprechenden Knoten keine Bedeutung.'n','m','k','p' und 'r' sind vom BABEL-Übersetzer erzeugte Knotenkennziffern. Der Test, der erste Knoten des Warte-Abschnitts, ist mit IW gekennzeichnet, um anzudeuten, daß mit diesem Knoten ein Warte-Abschnitt beginnt. CW steht für *conditional wait* als Kennzeichen eines Warte-Knotens.

```
Index       : 'n'     Tag : IW
Operation   : TR
Inputs      : 'condition-name' : WCN
Outputs     : 'aux_variable': AUX
Predecessor: 'p'
Condition   : 0       Successor : 'm'
Condition   : 1       Successor : 'k'

Index       : 'k'     Tag : CW
Operation   : X       Inputs : *    Outputs : *
Predecessor: 'n'
Condition   : X       Successor : 'm'

Index       : 'm'     Tag : IE
Operation   : X       Inputs : *    Outputs : *
Predecessor: 'n' 'k'
Condition   : X       Successor : 'r'
```

Bild 4.7 - Ausgabeformat eines Warte-Abschnitts

In einer Sequenz eines terminalen Prozesses gibt es außer den DSL-Knoten noch den BABEL-spezifischen Knoten a_k, der die Anforderung von Aktionen (PERFORM-Notation) darstellt. Dieser Knoten ist im Ausgabeformat des Übersetzers (Bild 4.8) mit ACALL bezeichnet. Die Felder 'inputs' und 'outputs' werden mit den Aktionsparametern belegt, und 'operation' mit dem Außenmittel (*resource-id*) und dem Aktions-Namen (*action-id*).

```
Index       : 'n'     Tag : ACALL
Operation   : 'resource-id': RES    'action-id' : ACT
Inputs      : 'input-calling-parameters'
Outputs     : 'output-calling-parameters'
Predecessor: 'p'
Condition   : X       Successor : 's'
```

Bild 4.8 - Ausgabeformat des Anforderungsknotens

In Bild 4.8 sind 'n','p' und 's' eindeutige Knotenkennziffern. 'calling-parameters' sind Parameter für die geforderte Aktion. 'resource_id' ist ein gültiger Außenmittelname und 'action-id' ist der Name der Aktion.

5. Synthese

Die Synthese ist im Rahmen dieser Arbeit die Phase des Entwurfszyklus, die aus einer Verhaltensbeschreibung eine Schaltungsstruktur erzeugt. Dieses Kapitel befaßt sich mit den Konzepten der Steuerungssynthese und führt in die nachfolgenden Kapitel 6, 7 und 8 ein.

5.1 *Synthesestrategie*

Beim systematischen Entwurf digitaler Systeme zerlegt man üblicherweise das vom Entwerfer beschriebene System in zwei Bestandteile, nämlich in einen gesteuerten Teil (Operationswerk) und in einen steuernden Teil (Steuerwerk) (Bild 5.1).

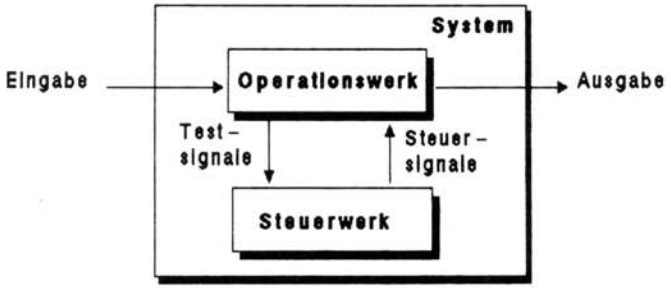


Bild 5.1 - Zerlegung eines Systems

Diese Strategie wird auch in CADDY [SCR84] angewendet. Aus dem Verhaltensgraphen einer Sequenz werden ein Operationswerk [CKR84] und ein Steuerwerk erzeugt, die das Verhalten des Systems auf der Struktur-Ebene darstellen.

Das Operationswerk ist in CADDY die Realisierung des Datenflusses, welcher der Verknüpfung der Operanden der DSL-Knoten im Verhaltensgraph entspricht. Die Synthese des Datenflusses bestimmt die digitalen Komponenten des Operationswerkes und die Steuersignale, die diese Komponenten benötigen, um die entsprechenden Operationen auszuführen.

Die Synthese der Steuerung stützt sich auf die im Verhaltensgraphen

vorhandene zeitliche Relation zwischen den Knoten und auf die von der Synthese des Datenflusses bestimmten Steuersignale jedes Knotens. Das erzeugte Steuerwerk liefert die nötigen Steuersignale so in einer geeigneten Reihenfolge, daß die Aktivität des Operationswerks dem Verhalten des beschriebenen Systems entspricht.

Die Steuerungssynthese unterteilt sich in zwei Phasen. Die erste Phase umfaßt die Transformationen, die aus einem Verhaltensgraphen eine Zustandsübergangstabelle liefern. Die zweite Phase erzeugt aus dieser Tabelle die entsprechende Schaltungsstruktur. Diese zweite Phase ist nicht Bestandteil dieser Arbeit. Es wird angenommen, daß es Verfahren gibt, welche eine Zustandsübergangstabelle in ein Steuerwerk umsetzen können, wie das beispielweise in LOGE der Fall ist [Lipp83].

5.2 *Steuerungsgrundlage*

Die Steuerwerke sind durch Schaltwerke realisiert. Schaltwerke zeichnen sich durch ihre speichernde Eigenschaft aus. Zu ihrer Beschreibung eignet sich das Modell der sequentiellen Maschinen (die Konzepte dieses Abschnitts entstammen der Literatur [Boot67]).

DEF. 5.2.1 - Sequentielle Maschine

Eine sequentielle Maschine sM ist ein Tupel $sM = (X,Y,Z,\lambda,\delta)$. Darin ist:

- X: eine endliche Menge von Eingabeelementen x_i
- Y: eine endliche Menge von Ausgabeelementen y_j
- Z: eine Menge von Zuständen z_k
- λ: eine Ausgabefunktion (λ: Z x X → Y)
- δ: eine Überführungsfunktion (δ: Z x X → Z)

Die Aufgabe einer sequentiellen Maschine besteht darin, aus einem Grundzustand heraus Eingabefolgen von Elementen einer Menge X nacheinander in Ausgabefolgen von Elementen einer Menge Y umzusetzen.

Eine sequentielle Maschine, deren Menge von Zuständen eine endliche Anzahl von Elementen enthält, nennt man endliche Maschine (*finite-state machine*). Eine endliche Maschine, die der Def. 5.2.1 entspricht, heißt Mealy-Maschine. Eine Änderung dieses Modells, welche die Ausgabefunktion λ als λ: Z → Y definiert, führt auf die Moore-Maschine. Steuerwerke sind die technische Realisierung von Mealy- oder Moore-Maschinen.

Zusätzlich zu den bisherigen Definitionen lassen sich angeben:

DEF. 5.2.2 - Zustandsübergang

Ein Zustandsübergang einer Maschine sM, $sM = (X,Y,Z,\lambda,\delta)$, ist ein geordnetes Paar (z_i,z_j) von Zuständen, für das gilt:

$$\exists\, x_k \in X \;:\; \delta(z_i,x_k) = z_j$$

In (z_i,z_j) nennt man den Zustand z_i einen Vorgänger von z_j und z_j einen Nachfolger von z_i.

DEF. 5.2.3 - Verzweigung

Eine Verzweigung tritt auf, wenn es mindestens drei Zustände z_i, z_j und z_k gibt, $z_j \neq z_k$, wobei (z_i,z_j) und (z_i,z_k) Zustandsübergänge sind.

DEF. 5.2.4 - Iteration

Eine Iteration tritt auf, wenn es eine Folge I von Zustandsübergängen gibt:

$$I = (z_0,..,z_i,..,z_n)$$

wobei: $\forall\, z_i \in I,\ (0 \leq i < n)$:
(z_i,z_{i+1}) ist ein Zustandsübergang $\wedge\ z_0 = z_n$

DEF. 5.2.5 - Ablauf

Ein Ablauf eines Steuerwerks wird durch eine Folge A von Zustandsübergängen dargestellt:

$$A = (z_0,..,z_i,..,z_n)$$

Für A gilt: $\forall\, z_i \in I,\ (0 \leq i < n)$:
(z_i,z_{i+1}) ist ein Zustandsübergang $\wedge$
z_0 ist der Startzustand $\wedge$
z_n ist ein Endzustand

Ein Ablauf ist das Durchlaufen durch ein Steuerwerk vom Startzustand bis zum Endzustand, wobei bei jeder Verzweigung nur einen Zustandübergang genommen wird.

Zur Beschreibung von sequentiellen Maschinen gibt es zahlreiche Möglichkeiten. Beispiele hierfür sind Zustandsübergangstafeln, Zustandsübergangstabellen und Zustandsübergangsdiagramme. Das Bild 5.2 zeigt das allgemeine Format einer Zustandsübergangstafel.

Z	X		
	x_1	.	x_m
z_1	$\lambda(z_1,x_1)$, $\delta(z_1,x_1)$	.	$\lambda(z_1,x_m)$, $\delta(z_1,x_m)$
.	.	.	.
z_n	$\lambda(z_n,x_1)$, $\delta(z_n,x_1)$	.	$\lambda(z_n,x_m)$, $\delta(z_n,x_m)$

Bild 5.2 - Zustandsübergangstafel

Eine für die Rechnereingabe besser geeignete Form der Beschreibung ist die Zustandsübergangstabelle (auch Übergangstabelle genannt). Diese Tabelle ist die Darstellung der Zustandsübergangstafel, deren Mengen X und Y durch ihre Belegungen ersetzt werden.

DEF. 5.2.6 - Belegung

> Ordnet man den Komponenten x_i, $1 \leq i \leq n$, einer Menge feste Werte zu, so wird das betreffende n-Tupel eine Belegung genannt.

Bei der Belegung der Eingabe- und Ausgabelemente einer sequentiellen Machine kommen ausschließlich die Werte 0, 1 und X (don't care) vor. Das Bild 5.3 zeigt ein Beispiel einer Übergangstabelle. An diesem Beispiel sieht man, daß Verzweigungen und Zustandsübergänge in dieser Art der Darstellung deutlicher hervortreten.

	Eingabebelegung				Ausgabebelegung				Folgezustand
Z	x_1	x_2	...	x_m	y_1	y_2	...	y_k	$\delta(Z,X)$
1	1	0	...	1	0	0	...	X	2
1	0	X	...	1	1	0	...	1	3
1	1	0	...	0	0	0	...	0	2
2	0	0	...	X	1	1	...	X	1
.	.	.	.	.	.	.	.	.	.

Bild 5.3 - Beispiel einer Übergangstabelle

Bei einer sequentiellen Maschine, die ein Steuerwerk beschreibt, sind die Eingabeelemente Testsignale und die Ausgabeelemente Steuersignale. Hier unterscheiden sich zwei Arten von Steuersignale: die Signale, die auf Komponente mit Speicher einwirken (Lade-Steuersignale genannt), und die Signale, die zur Auswahl einer Funktion einer kombinatorischen Schaltung dienen (Auswahl-Steuersignale). Zusätzlich dazu erwähnen die Kapitel 6 und 7 eine dritte Art von Steuersignalen, die zur Aktivierung (Starten) der Steuerwerke von Moduln benötigt wird. Die Trennung der Steuersignale ist für die Steuerungssynthese nötig, da die Transformationen zur Zustandsreduzierung diese Signale unterschiedlich behandelt müssen.

5.3 *Nebenläufigkeitsgrad*

Der Nebenläufigkeitsgrad ist eine Größe, die zur Bewertung der Steuerung eines Systems bezüglich der in diesem System realisierten Nebenläufigkeit dient.

Der Einfluß eines Steuersignals auf dem Operationswerk in einem Zustand z_i bestimmt, ob dieses Signal in diesem Zustand aktiv ist. Ein Lade-Steuersignal ist in einem Zustand aktiv, falls es eine Komponente des Operationswerks aktiviert oder deren Aktivierung erlaubt (*das Steuersignal eines Register ist z.B in einem Zustand aktiv, wenn eine Information in diesem Register geladen wird*). Ein Auswahl-Steuersignal ist in einem Zustand aktiv, wenn die gesteuerte Komponente ein Ausgangssignal liefert, die in diesem Zustand von anderer aktivierten Komponente benutzt wird (

das Steuersignal eines Multiplexers ist z.B aktiv, wenn er Signale, die das Operationswerk in diesem Moment z.B. als Eingabe für andere aktive Komponenten braucht, auswählt).

Die Menge der aktiven Steuersignale eines Zustands z_i sei $S_t(z_i)$, $S_t(z_i) \subseteq Y$. Der Nebenläufigkeitsgrad Φ_z eines Zustands z_i, $\Phi_z(z_i)$, ist das Verhältnis zwischen der Anzahl der aktiven Steuersignale in diesem Zustand und den Anzahl der Elemente der Merge Y der Steuersignale.

$$(a) \quad \Phi_z(z_i) = |S_t(z_i)| \; / \; |Y|$$

Der Durchschnittswert der Anzahl der aktiven Signale bestimmt in einem Steuerwerk c_j dessen Nebenläufigkeitsgrad Φ_c. Das folgende Verhältnis definiert diese Größe:

$$(b) \quad \Phi_c(c_j) = \frac{\sum_{i=1}^{n} \Phi_z(z_i)}{n}$$

wobei $n = |Z|$, die Anzahl der Zustände in c_j ist.

Φ_c entspricht dem Grad von gleichzeitigen Aktivitäten in einem gegebenen Moment. Φ_c zieht aber Attribute wie zum Beispiel die Häufigkeit der Aktivierung eines Zustands in einem Ablauf und die Häufigkeit dieses Ablaufs nicht in Betracht. Diese Attribute werden durch das dynamische Verhalten des Steuerwerks bestimmt und sind in den meisten Fällen durch eine statische Analyse nicht zu berechnen, da die Folge der Belegungen der Eingabeelemente unbekannt sind und die Anzahl der Abläufe wegen der Interaktionen unbestimmt ist.

Eine Alternative in dieser Richtung zur Gleichung (b) wäre, die nicht iterativen Abläufe zu berücksichtigen. Ein nicht iterativer Ablauf ist ein Ablauf (Def. 5.2.5), bei dem jede Iteration (Def. 5.2.4) nur einmal vorkommt. Die Anzahl der verschiedenen nicht iterativen Abläufe ist endlich und auch statisch zu bestimmen. Damit definiert man Φ_a, den Nebenläufigkeitsgrad eines Ablaufs, und mit dessen Hilfe redefiniert man Φ_c.

$$(c) \quad \Phi_a(a_j) = \frac{\sum_{i=1}^{m} \Phi_z(z_i)}{m}$$

$$(d)\quad \Phi_c(c_j) = \frac{\sum_{i=1}^{k} \Phi_a(a_i)}{k}$$

In (c) ist m die Anzahl der Zustände in einem nicht iterativen Ablauf a_j. In (d) ist k die endliche Anzahl der ungleichen nicht iterativen Abläufe von c_j.

Obwohl die Häufigkeiten der Abläufe in (d) nicht inbegriffen sind, haben die Zustände, die in mehreren Abläufen vorkommen, einen größeren Einfluß als die, die nur selten vorkommen. Der durch statische Analyse leicht zu berechnende Wert Φ_c reicht als Bewertungsgröße im Rahmen dieser Arbeit vollkommen aus .

Zum Schluß ist noch der Nebenläufigkeitsgrad Φ eines Systems zu definieren, das p Steuerwerke enthält. Zwei Fälle sind hier zu unterscheiden: a) alternierende Steuerwerke und b) nebenläufige Steuerwerke. Bei alternierenden Steuerwerken berechnet man den Nebenläufigkeitsgrad wie folgt:

- Die Anzahl der Ausgabeelemente der gesamten Steuerung ist gleich die Summe der Anzahl der Steuersignale aller p Steuerwerke.

$$(e)\quad y = \sum_{i=1}^{p} |Y_i|$$

- Die Anzahl der Zustände der Steuerung ist gleich die Summe der Anzahl der Zustände aller p Steuerwerke.

$$(f)\quad m = \sum_{i=1}^{p} m_i$$

Daraus ergibt sich die Gleichung (g) für die Darstellung der Nebenläufigkeitsgrad eines Systems mit alternierenden Steuerwerken:

$$(g)\quad \Phi = \frac{\sum_{i=1}^{p} \Phi_c(c_i).|Y_i|.m_i}{y.m}$$

Bei einem System mit p nebenläufigen Steuerwerke entsteht Φ aus der Summe der Φ_c aller aktiven Steuerwerke dieses Systems geteilt durch die Anzahl der Steuerwerke.

$$(h)\quad \Phi = \frac{\sum_{i=1}^{p} \Phi_c(c_i)}{p}$$

In (h) sind nur die Steuerwerke zu berücksichtigen, die unabhängig sind und gleichzeitig aktiv sein können. Die Steuerwerke, die alternierend aktiv vorkommen, werden als ein einziges betrachtet.

Die Gleichungen (g) und (h) deuten darauf, daß ein System mit nebenläufigen Steuerwerken einen größeren Nebenläufigkeitsgrad leistet als ein äquivalentes System mit alternierenden Steuerwerken.

Im allgemein benötigt man für die Interaktion zwischen den nebenläufigen Steuerwerken zusätzlich Steuersignale und Zustände. Diese müssen in (h) berücksichtigt werden. Die Unterschiede zwischen dem Φ des Systems mit Interaktionssignalen und Zuständen und des Φ des Systems gegeben durch (g) und (h) ohne diese Elemente dient zur Beunteilung der Effizienz des Interaktionsmechanismus.

6. Synthese der Steuerung

Gegeben sei ein Verhaltensgraph, der das Verhalten eines digitalen Systems mit einem einzigen terminalen Prozeß beschreibt, und ein aus dieser Beschreibung erzeugtes Operationswerk. Gesucht ist eine Spezifikation eines Steuerwerks, welches das Operationswerk korrekt steuert. Dieses Kapitel führt ein Verfahren zur Synthese dieser Spezifikation ein.

6.1 *Das Syntheseverfahren*

6.1.1 *Das Steuerwerksmodell*

Die zu erzeugende Steuerung eines Prozesses ist ein Schaltwerk, das der technischen Realisierung eines Mealy-Automaten (DEF 5.2.1) entspricht. Die Spezifikation dieses Automaten erfolgt durch eine Übergangstabelle. Die Menge der Ausgabeelemente Ss des zu erzeugenden Mealy-Automaten enthält drei Arten von Steuersignalen:

a) *Auswahl-Steuersignale (Ass):*
   Diese Signale dienen zur Auswahl einer Funktion einer kombinatorischen Schaltung (z.B: Multiplexer, Addierer).

b) *Lade-Steuersignale (Lss):*
   Sie ermöglichen das Laden einer digitalen Komponente mit Takteingang (z.B: Register, Flipflops, Zähler).

c) *Aktivierungssignale (Kss):*
   Sie dienen zur Aktivierung von Moduln und selbst ablaufenden Operationen.

Die Steuersignale Ss aktivieren ein synchrones Operationswerk, das nur einen einzigen Takt benötigt.

6.1.2 *Übersicht über das Synthesekonzept*

Das Syntheseverfahren erzeugt eine Steuerung gemäß den Daten aus zwei Spezifikationsquellen, nämlich dem Verhaltensgraphen des Systems und dem für dieses System generierten Operationswerk.

a) *Verhaltensgraph*: Der Verhaltensgraph bestimmt die Abläufe der Schaltung. Aus dem Verhaltensgraphen zieht man die zeitliche Relation zwischen den zu realisierenden Operationen und den Tests heraus.

b) *Operationswerk*: Das generierte Operationswerk bestimmt die nötigen Steuersignale für jeden realisierten Knoten des Verhaltensgraphen und deutet auf die Einschränkung in der Reihenfolge der Aktivierung der Operationen hin.

Eine korrekte Zustandsübergangstabelle zur Steuerung eines Prozesses kann durch eine eins-zu-eins Transformation der Knoten des Verhaltensgraphen in Zustände erzeugt werden.

Knoten → Zustand

Bei dieser direkten Transformation bleibt die Vorgänger-Nachfolger-Relation des originalen Verhaltensgraphen erhalten. Die nebenläufigen Abschnitte werden sequentialisiert, und alle Knoten werden dann durch die entsprechenden Steuersignale ersetzt. Diese einfache Transformation erzeugt jedoch zuviele Zustände und einen zu geringen Nebenläufigkeitsgrad.

Eine bessere Lösung ist es, Zustände nur zu erzeugen, wenn sie nötig sind, d.h. beispielweise in Fällen von Konflikten und Instabilitäten und wenn Zeit-Einschränkungen vorkommen. Diese und andere Fälle, in denen Zustände nötig sind, werden in diesem Kapitel ausführlich beschrieben.

Während der Synthese der Übergangstabelle wird eine interne Darstellung des Zustandes, die Übergangseinheit (*transition unit*), verwendet. Die Transformationen, die eine Übergangstabelle mit einer kleinen Anzahl von Zuständen erzeugen, sind folgende:

- Jeder Knoten des Verhaltensgraphen wandelt sich in eine Übergangseinheit um. Diese Transformation heißt 'Erzeugung der Übergangseinheit'.

- Überflüssige Übergangseinheiten werden bei der Transformation 'Reduktion' entfernt.

- Konfliktfreie Übergangseinheiten werden durch die Transformationen 'Kompaktierung und Komposition' zusammengefaßt.

- Reduktion, Kompaktierung und Komposition wiederholen sich, bis keine Transformation mehr möglich ist.

- Zum Schluß wandelt sich jede erzeugte Übergangseinheit durch die

Fortpflanzung der Werte der Steuersignale und durch Änderung ihres Formats in einen Zustand um. Diese Transformation heißt Formatierung.

Die oben erwähnten Transformationen finden zuerst in den Sektionen der tieferen Ebenen statt. Eine Sektion entspricht einem Abschnitt des Verhaltensgraphen und wird während der Bearbeitung dieses Abschnitts erzeugt. Die Synthese der Steuerung verwendet das Konzept der Sektion:

- um die Einschränkungen des originalen Abschnitts zu erhalten.
- um die Bestimmung der Nachfolgerzustände zu vereinfachen.
- um die Kompaktierung und Komposition zwischen den Übergangseinheiten, die aus Knoten verschiedener Abschnitte erzeugt werden, zu regeln.

6.2 *Definitionen*

6.2.1 *Übergangseinheit*

Eine Übergangseinheit (*transit*) ist eine interne Darstellung eines Zustands.

DEF. 6.2.1 - Übergangseinheit

Eine Übergangseinheit ist ein Tupel:

$$t_i = (id, type, Pred, Ts, Zw)$$

mit den Komponenten:
- Kennzeichen (id), $id(t_i)=i$, $i \geq 0$
- Typ (type)
- Menge der Übergangseinheits-Vorgänger (Pred)
- Folge der Testsignale (Ts)
- Menge der Zweige (Zw)

Das Kennzeichen einer Übergangseinheit muß eindeutig sein. Die Transformation 'Formatierung' verwendet es als Zustandskennzeichen. Die Folge Ts enthält die Testsignale der Knoten des Verhaltensgraphen, aus dem die Übergangseinheit stammt. Für Ts ist die Funktion $VECTOR(Ts(t_i))$ definiert, die alle relevanten Belegungswerte dieser Folge in der Übergangseinheit t_i darstellt. Zu jedem Belegungswert w, $w \in VECTOR(Ts(t_i))$, gibt es einen entsprechenden Zweig.

DEF. 6.2.2 - Zweig

Ein Zweig einer Übergangseinheit ist ein Tupel:

Zweig = (w, Vk, suc)

mit den Komponenten:
- Belegungswert der Testsignale (w)
- Menge der entsprechenden Knoten (Vk)
- Übergangseinheits Nachfolger (suc)

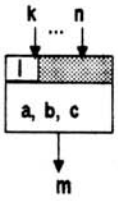
Die Menge der entsprechenden Knoten Vk einer Übergangseinheit t_i enthält die Knoten des Verhaltensgraphen, aus dem t_i stammt. Zu jedem $k_j \in Vk(t_i)$ gibt es im Operationswerk eine entsprechende Realisierung c_j. Die Menge Cs_{k_j} enthält die Steuersignale, welche die Realisierung c_j des Knotens k_j aktivieren. Die Menge $Cs(t_i)$ enthält die Steuersignale aller entsprechenden Knoten von t_i. Für ein Steuersignal ss, $ss \in Cs(t_i)$, ist die Funktion $VALUE_i(ss)$ definiert. Diese Funktion stellt den Belegungswert von ss in t_i dar.

6.2.2 *Übergangseinheits-Typen*

Es sind vier Typen zu unterscheiden:

a) *Grundeinheit (single transit):*

Eine Grundeinheit enthält kein Testsignal, $|Ts| = 0$, und hat einen einzigen Zweig, $|Zw| = 1$. Bild 6.2.1 zeigt ihre Darstellungen.



transit t_i:
id (t_i) = i
type(t_i) = single
Pred(t_i) = {k,...,n}
suc (t_i) = m
Vk (t_i) = {a,b,c}

a,b & c sind DSL-Knoten Kennzeichen

Bild 6.2.1 - Darstellungen einer 'single-transit'

Die Menge $Suc(t_i)$ der Nachfolger von t_i wird in diesem Fall wie folgt definiert: $Suc(t_i) = \{suc(t_i)\}$

b) *Testeinheit:*

In einer Testeinheit gibt es mehrere Zweige, $|Zw| > 1$, und mindestens ein Testsignal, $|Ts| \geq 1$. Die Darstellungen einer Testeinheit sind im Bild 6.2.2 zu sehen.

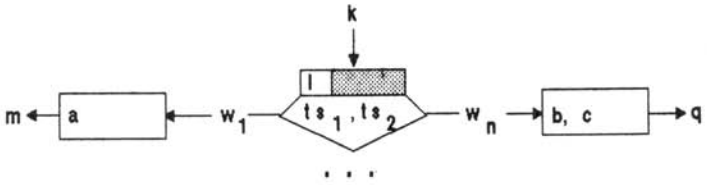
Die Menge $Suc(t_i)$ der Nachfolger und die Menge $Cs(t_i)$ der Steuersignale von t_i werden wie folgt definiert:

$$Suc(t_i) = (suc_{w1}(t_i), \ldots, suc_{wn}(t_i))$$
$$Cs(t_i) = (Cs_{w1}(t_i), \ldots, Cs_{wn}(t_i))$$

Bezüglich Vk, der Menge der entsprechenden Verhaltensgraphenknoten einer Übergangseinheit t_i, sind zwei Arten von Testeinheiten zu erkennen:

Moore: $\forall\ w_k, w_j \in VECTOR(Ts(t_i)) : Vk_{wk}(t_i) = Vk_{wj}(t_i)$

Mealy: $\exists\ w_k, w_j \in VECTOR(Ts(t_i)) : Vk_{wk}(t_i) \neq Vk_{wj}(t_i)$



transit t_i:

$id(t_i) = i$,	$type(t_i) = test$
$Pred(t_i) = (k)$,	$Ts(t_i) = (ts_1, ts_2)$
$suc_{w_1}(t_i) = m$,	$Vk_{w_1}(t_i) = (a)$
...	...
$suc_{w_n}(t_i) = q$,	$Vk_{w_n}(t_i) = (b,c)$

Bild 6.2.2 - Darstellungen einer 'test-transit'

c) *Kommunikationseinheit:*

Eine Kommunikationseinheit ist eine Grundeinheit, die nur einen entsprechenden Knoten enthält, $|Vk| = 1$, und dieser Knoten ist entweder ein Anforderungsknoten oder ein Warte-Knoten. Dieser Typ wird in Kapitel 8 behandelt.

d) *Verzögerungseinheit:*

Eine Verzögerungseinheit ist eine Grundeinheit, die keinen entsprechenden Knoten enthält, $|Vk| = 0$. Verzögerungseinheiten werden erzeugt, um die in der Beschreibungssprache vorhandene Spezifikation von CYCLEs zu erhalten.

6.2.3 *Übergangsgraph und Sektionen*

Ein Übergangsgraph ist ein gerichteter Graph, dessen Knoten Übergangseinheiten sind und in dem die Vorgänger-Nachfolger Relation durch die Mengen *Pred* und *Suc* der Übergangseinheiten bestimmt wird.

DEF. 6.2.3 - Übergangsgraph

Ein Übergangsgraph U_{graph} ist ein gerichteter Graph, $U_{graph}=(M_t,K,\varphi)$, der aus einer endlichen nicht leeren Menge M_t von Übergangseinheiten, einer endlichen Menge K von Kanten und einer Inzidenzabbildung φ: $K \rightarrow M_t \times M_t$ besteht.

DEF. 6.2.4 - Sektion

Eine Sektion $Sk = (M_t',K',\varphi')$ ist ein Teilgraph eines Übergangsgraphen $U_{graph} = (M_t,K,\varphi)$ mit einem einzigen Anfangsknoten α_t. Der Anfangsknoten α_t wird auch Eingangspunkt (*entry point*) der Sektion genannt. Für den Eingangspunkt gilt:

$$\exists! \, t_i, \; (t_i \notin M_t') \wedge (t_i \in M_t) \; : \; id(t_i) \in Pred(\alpha_t)$$

Eine Sektion wird durch die Umwandlung der Knoten eines Abschnitts in Übergangseinheiten erzeugt. Entsprechend der originalen Struktur dieses Abschnitts sind drei elementare Gruppen von Sektionen zu erkennen:

a) *Lineare Sektion:* Diese Sektion ergibt sich aus einer Kette.

b) *Verzweigte Sektion:* Diese Sektion ergibt sich aus IF-Abschnitten und CASE-Abschnitten.

c) *Iterative Sektion:* Diese Sektion ergibt sich aus WHILE-Abschnitten, FOR-Abschnitten und DO-UNTIL-Abschnitten.

Es gibt noch eine vierte Gruppe von Sektionen, die zur Aktivierung von Moduln und nicht primitiven Operationen dient. Diese wird in Kapitel 7 eingeführt. Eine fünfte und letzte Gruppe, die beim Ersetzen von Anforderungs- und Warte-Knoten durch die ensprechende Steuerung entsteht, wird in Kapitel 8 behandelt.

Obwohl Sektionen aus Abschnitten stammen und diesen ähnlich sind, können in einer Sektion mehrere Endknoten vorkommen, während ein Abschnitt nur einen einzigen Endknoten enthält. Die Endknoten einer Sektion heißen Ausgangspunkte (*exit points*) der Sektion. Für sie gilt:

- Alle Ausgangspunkte einer Sektion haben denselben Nachfolger außerhalb dieser Sektion.
- Dieser Nachfolger ist der Eingangspunkt einer anderen Sektion.

Für Sektionen definiert man die Funktionen ENTRY(Sk) und EXIT(Sk), welche die Menge der Eingangspunkte bzw. der Ausgangspunkte einer Sektion Sk darstellen. Die in den Abschnitten zu findende Struktur der Aufteilung in mehrere Ebenen (vgl. Regeln 3.2 bis 3.6 in Kapitel 3) ist auch in den Sektionen vorhanden. Eine Sektion von höherer Ebene wird auf Sektionen der tieferen Ebenen aufgebaut.

6.3 *Konfliktfälle*

Einige Relationen zwischen den Übergangseinheiten und ein Attribut der Testeinheit kennzeichnen die Konfliktfälle.

Attribut: Instabilität

Relationen: Inkompatibilität, Korrelation
Zeitabhängigkeit, Pfadabhängigkeit

a) *Instabilität*

INSTABILITY : ts → BOOLEAN

Das Attribut Instabilität weist darauf hin, ob ein Testsignal ts aus mindestens einem Eingangs-Interface ohne Zwischenspeicherung stammt. Es bedeutet: wenn eine asynchrone Änderung irgendeines Interfaces eine asynchrone Änderung eines Testsignals verursacht, dann ist dieses Testsignal instabil.

b) *Inkompatibilität*

$$\text{INCOMPATIBILITY} : \quad t_i \times t_j \rightarrow \text{BOOLEAN}$$

Gegeben seien zwei Übergangseinheiten, t_i und t_j, $Cs(t_i)$ (die Menge der Steuersignale von t_i) und $Cs(t_j)$ (die Menge der Steuersignale von t_j). Die Übergangseinheiten t_i, t_j sind kompatibel, falls:

$$Cs(t_i) \cap Cs(t_j) = \{\}$$ oder

$$\forall\, ss \in (Cs(t_i) \cap Cs(t_j)) : (ss \in Ass) \wedge (VALUE_i(ss) = VALUE_j(ss))$$

In allen anderen Fällen sind sie inkompatibel. Zwei Übergangseinheiten sind dann inkompatibel, wenn sie das gleiche Auswahl-Steuersignal mit unterschiedlichen Werten oder dasselbe Lade-Steuersignal liefern.

c) *Korrelation*

$$\text{CORRELATION} : \quad t_i \times ts \rightarrow \text{BOOLEAN}$$

Gegeben seien eine Übergangseinheit t_i, ein Testsignal ts und $Cs(t_i)$ (die Menge der Steuersignale von t_i). t_i und ts sind korrelativ, falls die Aktivität eines Signals von $Cs(t_i)$ eine Änderung in ts verursacht.

d) *Pfadabhängigkeit*

$$\text{PATH DEPENDENCE} : \quad t_i \times t_j \rightarrow \text{BOOLEAN}$$

Es sei ein Datenpfad P_{opd} eine Folge von realisierten Operationen c_i, durch die ein Operand fließt: $P_{opd} = (c_1, c_2, \ldots, c_n)$ und seien t_i und t_j Übergangseinheiten. Seien k_i ein entsprechender Knoten von t_i, $k_i \in Vk(t_i)$, und k_j ein entsprechender Knoten von t_j, $k_j \in Vk(t_j)$. Seien c_i die Realisierung von k_i und c_j die Realisierung von k_j. Dann ist Pfadabhängigkeit wie folgt definiert:

$$c_i, c_j \in P_{opd} \Rightarrow (\text{PATH DEPENDENCE}\ (t_i, t_j) = \text{TRUE})$$

Übergangseinheiten, die zu demselben Datenpfad Steuersignale liefern, sind pfadabhängig.

e) *Zeitabhängigkeit*

TIMING DEPENDENCE : t_i X t_j → BOOLEAN

Zwei Übergangseinheiten sind zeitabhängig, falls die zeitliche Relation ihrer entsprechenden Knoten eingeschränkt bei einer Zeitspezifikation vorkommt. Zeitspezifikationen werden in BABEL und DSL durch die Anwendung von CYCLE angegeben.

6.4 *Erzeugung der Übergangseinheiten*

Jeder Knoten des Verhaltensgraphen wird in eine oder mehrere Übergangseinheiten umgewandelt. Einige dieser Übergangseinheiten werden später entfernt (s.u. Reduktion). Die Knoten, die mehr als eine Übergangseinheit erzeugen können, sind folgende:

- Funktionsknoten (DSL-Knoten, deren Operation eine vom Benutzer definierte Funktion ist).

- Nicht primitive Operationen (z. B. Multiplikation oder Division).

Die Erzeugung von Übergangseinheiten aus dieser Art von Knoten wird im Kapitel 7 behandelt. Alle anderen Knoten werden gemäß den Regeln der Tabelle 6.1 umgewandelt:

Die Erzeugung einer Übergangseinheit bestimmt den Inhalt der folgenden Elemente:

- *Kennzeichen (id)*: Jede erzeugte Übergangseinheit bekommt ein eindeutiges Kennzeichen.
- *Typ (type)*: Der Typ wird gemäß der Tabelle 6.1 bestimmt.
- *Testsignale (Ts)*: Sie sind die Testsignale des entsprechenden Knotens.
- *Belegungswert der Testsignale (w)*: Ein Element von VECTOR(Ts(t_i)).
- *Entsprechender Knoten (Vk)*: Der Verhaltensgraphknoten, der t_i erzeugt.

Die Mengen *Pred* und *Suc* einer Übergangseinheit werden erst beim Einfügen dieser Übergangseinheit in den Übergangsgraphen bestimmt.

Regel	Verhaltensgraphenknoten	Erzeugte Übergangseinheit
1.	Operation	single transit
2.	Strukturknoten (außer dem FORK-Knoten)	single transit
3.	Test	test transit IF INSTABILITY(test-signal) THEN Moore ELSE Mealy
4.	Anforderungsknoten Warte-Knoten	communication transit
5.	FORK-Knoten	-

Tabelle 6.1 - Erzeugungsregeln

Die 5. Regel bedeutet, daß ein FORK-Knoten (der Anfangsknoten eines FORK-JOIN-Abschnitts) keine Übergangseinheit erzeugt. In einem Übergangsgraphen ist die Darstellung von nebenläufigen Übergangseinheiten nicht nötig. Echte Parallelität wird durch die Transformationen Kompaktierung und Komposition erreicht und durch die Zusammenfassung mehrerer Steuersignale in einem Zustand dargestellt.

6.5 *Erzeugung des Übergangsgraphen*

Die Funktionen zur Erzeugung eines Übergangsgraphen sind:

Datentyp: Übergangsgraph (G)

```
CREATE      :                  → G
INSERT      : G X transit      → G
CONCATENATE : G X G            → G
EMPTY       :                  → BOOLEAN
```

a) *CREATE*

Ein leerer Übergangsgraph wird erzeugt.

b) *INSERT*

Eine Übergangseinheit t_i wird am Ende des Übergangsgraphen eingefügt. Die Menge *Pred* von t_i und die Mengen *Suc* aller Vorgänger dieser Übergangseinheit werden gemäß der vorhandenen Vorgänger-Nachfolger-Relation zwischen den entsprechenden Verhaltensgraphenknoten vervollständigt.

Darstellung: $\quad INS\ (G_1,\ t_i) \Rightarrow G_2$

Sei A_p die Menge der Ausgangspunkte von G_1, die sich auf mindestens einen Nachfolger ungleich t_i beziehen.

$$A_p = \{t_p \in EXIT(G_1) \mid \exists\ a \in Suc(t_p)\ .\ (a \neq id(t_i))\}$$

Die folgenden Gleichungen bestimmen die Eingangs- und Ausgangspunkte von G_2:

$$ENTRY(G_2) = ENTRY(G_1)$$

$$(\forall\ b \in Suc(t_i))\ .\ (t_b \in G_1) \Rightarrow\ EXIT(G_2) = A_p$$

$$(\exists\ b \in Suc(t_i))\ .\ (t_b \notin G_1) \Rightarrow\ EXIT(G_2) = A_p \cup \{t_i\}$$

Die Menge *Suc* von t_i bleibt unbestimmt, bis alle ihre Nachfolger eingefügt sind. Falls es sich bei G_1 um einen leeren Übergangsgraphen handelt, ist t_i der Eingangs- und Ausgangspunkt von G_2.

c) *CONCATENATE*

Zwei Teilgraphen G_1 und G_2 werden beim Einfügen des Übergangsgraphen G_2 am Ende des Übergangsgraphen G_1 verkettet. Die Verkettung bestimmt die Relation zwischen G_1 und G_2 gemäß der entsprechenden Relation zwischen den ursprünglichen Abschnitten, aus denen G_1 und G_2 stammen.

Darstellung der Verkettung: $\quad CONC\ (G_1,\ G_2) \Rightarrow G_3$

Sei α_t der Eingangspunkt von G_2. Bei der Verkettung gelten die folgenden Gleichungen:

$$ENTRY(G_3) = ENTRY(G_1)$$

$$EXIT(G_3) = \{t_p \in EXIT(G_1) \mid \exists\ a \in Suc(t_p)\ .\ a \neq id(\alpha_t)\} \cup \{t_q \in EXIT(G_2) \mid \exists\ b \in Suc(t_q)\ .\ t_b \notin G_1\}$$

Die Ausgangspunkte von G_3 sind die Übergangseinheiten, die entweder zu $EXIT(G_1)$ oder $EXIT(G_2)$ gehören und sich auf Nachfolger außerhalb von G_2 bzw. G_1 beziehen.

Axiome der Verkettung:

A1: $CONC\ (CREATE,\ G) = G$
A2: $CONC(G_1, INS(G_2, t_i)) = INS(CONC(G_1, G_2), t_i)$

d) *EMPTY*

Die boolsche Funktion *EMPTY* bestimmt, ob ein Übergangsgraph leer ist oder nicht. Axiome:

A3: $EMPTY\ (CREATE) = TRUE$
A4: $EMPTY\ (INS\ (CREATE,\ t_i)) = FALSE$

6.6 *Reduktion*

Die Reduktion ist eine Transformation auf einem Übergangsgraphen, welche überflüssige Übergangseinheiten entfernt. Diese Transformation umfaßt die Anpassung der Mengen *Pred* aller Nachfolger und *Suc* aller Vorgänger der zu entfernenden Übergangseinheit.

6.6.1 *Reduktionsregeln*

Bezüglich der zu transformierenden Übergangseinheit sind drei Variationen der Reduktion zu unterscheiden. Diese Variationen werden als R1, R2 und R3 bezeichnet.

R1: *Grundeinheit ohne Steuersignal*

Sei k_i ein entsprechender Knoten von t_i, $k_i \in Vk(t_i)$. Sei t_i eine Grundeinheit ohne Steuersignale:

$$\forall\ k_i \in Vk(t_i)) \ :\ Cs_{k_i} = \{\ \}$$

R1 entfernt diese Art von Übergangseinheit:

$$CONC(INS(G_1, t_i), G_2) \overset{R1}{\Rightarrow} CONC(G_1, G_2)$$

R2: *Testeinheit mit gleichen Zweigen*

Eine Testeinheit mit gleichen Zweigen t_i wird durch folgendes Prädikat gekennzeichnet:

$$\forall\ w_i, w_j \in \mathrm{VECTOR}(\mathrm{Ts}(t_i)) :$$
$$(\mathrm{suc}_{w_i}(t_i) = \mathrm{suc}_{w_j}(t_i)) \ \wedge \ (\mathrm{Cs}_{w_i}(t_i) = \mathrm{Cs}_{w_j}(t_i))$$

Diese Art von Testeinheit wird durch eine äquivalente Grundeinheit t_j ersetzt:

$$\mathrm{CONC}(\mathrm{INS}(G_1, t_i), G_2) \overset{R2}{\Rightarrow} \mathrm{CONC}(\mathrm{INS}(G_1, t_j), G_2)$$

Für t_j gilt:

$$\mathrm{id}(t_j) = \mathrm{id}(t_i) \ \wedge \ \mathrm{Vk}(t_j) = \mathrm{Vk}_{w_i}(t_i) \ \wedge \ \mathrm{suc}(t_j) = \mathrm{suc}_{w_i}(t_i)$$

Beispiel 6.6.1:

Bild 6.6.1 zeigt ein Beispiel für die Anwendung der Reduktionsregel R2.

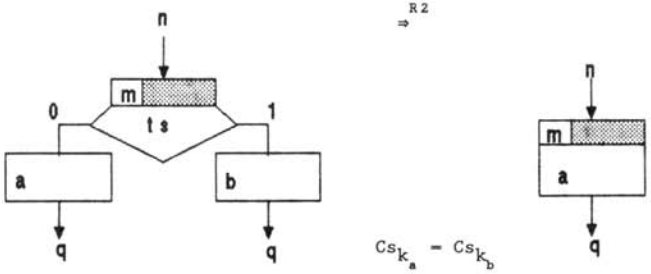


Bild 6.6.1 - Anwendung der Reduktionsregel R2

Ende des Beispiels.

R3: *Verzweigte Sektion mit gleichen Zweigen*

R3 ersetzt eine verzweigte Sektion mit gleichen Zweigen Sk_i durch eine äquivalente Sektion Sk_j:

$$\mathrm{CONC}(\mathrm{CONC}(G_1, Sk_i), G_2) \overset{R3}{\Rightarrow} \mathrm{CONC}(\mathrm{CONC}(G_1, Sk_j), G_2)$$

Die äquivalente Sektion Sk_j hat α_{t_j}, eine Grundeinheit, als Eingangspunkt. α_{t_j} ersetzt den Eingangspunkt α_{t_i} von Sk_i, einer Testeinheit. Der Nachfol-

ger von α_{t_j} ist der Eingangspunkt der Sektion irgendeines Zweiges von Sk_i.

Die Zweige einer verzweigten Sektion sind gleich, falls die Sektionen in diesen Zweigen gleich sind. Zwei Sektionen Sk_p und Sk_q sind gleich, wenn:

$\forall\ t_p \in Sk_p,\ \exists 1\ t_q \in Sk_q$:

$$Ts(t_p) = Ts(t_q) \wedge\ |Zw(t_p)| = |Zw(t_q)| \wedge$$
$$\forall\ w_i \in Ts(t_p) : Cs_{w_i}(t_p) = Cs_{w_i}(t_q) \wedge$$
$$\forall\ a \in Suc(t_p),\ \exists 1\ b \in Suc(t_q) : Cs(t_a) = Cs(t_b)$$

6.6.2 *Korrektheit der Reduktion*

Es sei ein aus einem Verhaltensgraphen V erzeugter Übergangsgraph G vor der Reduktion korrekt. Die Entfernung einer Übergangseinheit t_i durch die Anwendung von R1 hat keinen Einfluß auf die Aktivierung des Operationswerks, da t_i keine Steuersignale liefert und die Ordnung der Knoten von V im reduzierten Übergangsgraphen G erhalten bleibt. R1 verkürzt jedoch den Ablaufzyklus der Steuerung, was sie aber gegenüber der durch V dargestellten Spezifikation nicht ungültig macht, da die zeitliche Relation in V nur schwach ist (vgl. Abschnitt 3.1.3).

Es sei t_j in G eine Testeinheit mit gleichen Zweigen. In diesem Fall zeigen die Testsignale von t_j keine Auswirkung auf den Ablauf. Deshalb beeinflußt die Anwendung von R2 über t_j weder den Ablaufzyklus noch die Aktivierung des Operationswerkes. Die gleiche Betrachtung gilt für die Anwendung von R3 auf eine verzweigte Sektion mit gleichen Zweigen.

Damit wird bestimmt, daß die Anwendung der Reduktion auf einen korrekten Übergangsgraphen einen wiederum korrekten Übergangsgraphen ergibt.

6.7 *Kompaktierung*

Die Transformation 'Kompaktierung' ersetzt zwei oder mehrere durch die Nachfolger-Vorgänger-Relation verknüpfte Übergangseinheiten durch eine Menge von äquivalenten Übergangseinheiten. Diese Transformation verkleinert die Anzahl der Knoten in einem Übergangsgraphen, erhöht die Anzahl der Steuersignale in den kompaktierten Übergangseinheiten und erreicht damit einen höheren Nebenläufigkeitsgrad. Die Kompaktierung über die Grenze einer iterativen Sektion heißt Verlagerung in eine iterative Sektion (*migration into an iterative section*)

Um die Korrektheit dieser Transformation zu gewährleisten, müssen die Konfliktfälle des Abschnitts 6.3 vermieden werden, was die freie Anwendung der Kompaktierung einschränkt. Dieses Thema wird im Abschnitt 6.7.3 naher behandelt.

Bei der Kompaktierung sind zwei Fälle bezüglich der Anzahl der in diese Transformation verwickelten Übergangseinheiten zu unterscheiden: die Kompaktierung von zwei bzw. die von mehreren Übergangseinheiten.

6.7.1 *Kompaktierung von zwei Übergangseinheiten*

Zwei Übergangseinheiten t_i und t_j werden kompaktiert, falls:

$$|Pred(t_j)| = 1 \quad \wedge \quad id(t_i) \in Pred(t_j)$$

Kompaktierungsregel:

$$CONC(INS(INS(G_1,t_i),t_j),G_2) \Rightarrow^K CONC(INS(G_1,t_n),G_2)$$

Das Ergebnis der Kompaktierung ist eine neue Übergangseinheit t_n.
Bezüglich des Typs von t_j sind zwei Variationen dieser Transformation, K1 (*type(t_j)=single*) und K2 (*type(t_j)=test*), zu unterscheiden. Für die Bestimmung von t_n gemäß K1 und K2 gilt die Tabelle 6.2.

	K1	K2
t_n	type(t_j)=single	type(t_j)=test
id(t_n)	id(t_i)	id(t_i)
type(t_n)	type(t_i)	test
Pred(t_n)	Pred(t_i)	Pred(t_i)
Ts (t_n)	Ts(t_i)	Ts(t_i) $\cup$ Ts(t_j)

Tabelle 6.2 - Kompaktierungsregeln K1 und K2

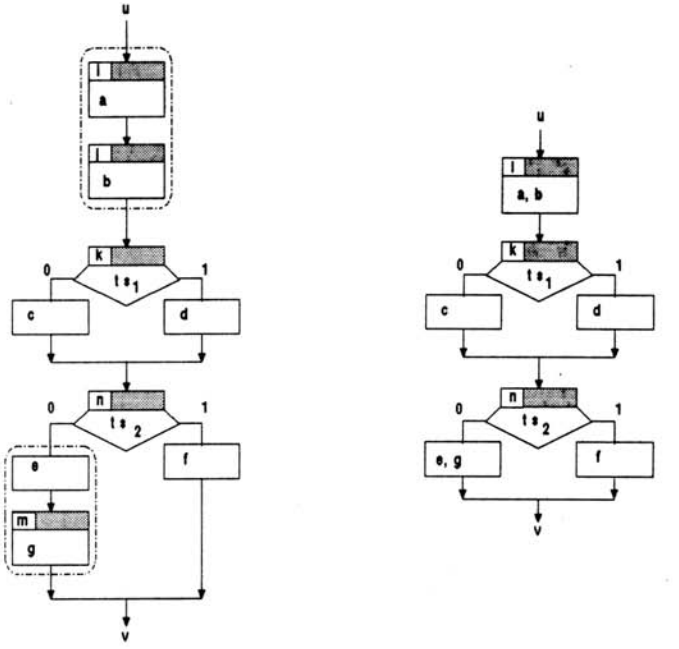
Die Attribute Vk und suc von t_n werden durch die folgenden Prädikate bestimmt:

K1: $\forall\ w_i \in VECTOR(Ts(t_i))\ .\ (suc_{w_i}(t_i) = id(t_j))\ :$
$Vk_{w_i}(t_n) = Vk_{w_i}(t_i) \bigcup Vk(t_j) \wedge$
$suc_{w_i}(t_n) = suc(t_j)$

K2: $(\forall\ w_i \in VECTOR(Ts(t_i))\ .\ (suc_{w_i}(t_i) = id(t_j))),$
$(\forall\ w_j \in VECTOR(Ts(t_j))),\ \exists\ w_i w_j \in VECTOR(Ts(t_n))\ :$
$Vk_{w_i w_j}(t_n) = Vk_{w_i}(t_i) \bigcup Vk_{w_j}(t_j) \wedge$
$suc_{w_i w_j}(t_n) = suc_{w_j}(t_j)$

Das folgende Beispiel zeigt die Anwendung der Kompaktierungsregeln K1 und K2.

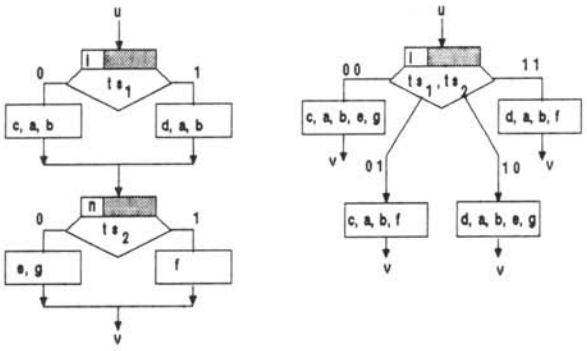
Beispiel 6.7.1:



(a) (b)

Bild 6.7.1 - Anwendung von K1 auf (t_i, t_j) und (t_n, t_m)

Es seien die fünf Übergangseinheiten t_i, t_j, t_k, t_n und t_m wie in Bild 6.7.1.a verknüpft. Zwischen diesen Übergangseinheiten kommen keine Konflikte vor. (t_i,t_j) und (t_n,t_m) werden gemäß K1 kompaktiert. Das Ergebnis ist in Bild 6.7.1.b zu sehen. Jetzt kann man K2 auf t_i und t_k im Bild 6.7.1.b anwenden. Das Bild 6.7.2.a stellt das Ergebnis dar.



(a) (b)

Bild 6.7.2 - Anwendung von K2 auf t_i und t_n

Zum Schluß wird wieder K2 auf die Übergangseinheiten im Bild 6.7.2.a angewandt. Das Bild 6.7.2.b zeigt das endgültige Ergebnis.

Ende des Beispiels.

6.7.2 *Kompaktierung mehrerer Übergangseinheiten*

Eine Menge P von Übergangseinheiten und eine Übergangseinheit t_j werden kompaktiert, falls:

$$P = (t_i \mid id(t_i) \in Pred(t_j)) \quad \wedge \quad |P| = |Pred(t_j)|$$

Kompaktierungsregel: $CONC(INS(G_1),t_j),G_2) \Rightarrow^K CONC(G_1',G_2)$.

G_1' stellt das Ergebnis der Kompaktierung dar und enthält die Menge C der kompaktierten Übergangseinheiten. Es ist zu beachten, daß $P \subset EXIT(G_1)$ ist.

Bezüglich des Typs von t_j sind zwei Variationen dieser Transformation, K3 (*type(t_j)=single*) und K4 (*type(t_j)=test*), zu unterscheiden. Die Tabelle 6.3 gilt für alle aus K3 oder K4 sich ergebenden t_n.

$\forall\ t_i \in P,\ \exists\ t_n \in C$	K3 type(t_j)=single	K4 type(t_j)=test
$id(t_n)$	$id(t_i)$	$id(t_i)$
$type(t_n)$	$type(t_i)$	test
$Pred(t_n)$	$Pred(t_i)$	$Pred(t_i)$
$Ts\ (t_n)$	$Ts(t_i)$	$Ts(t_i) \bigcup Ts(t_j)$

Tabelle 6.3 - Kompaktierungsregeln K3 und K4

Die Attribute Vk und suc jeder t_n werden durch die folgenden Prädikate bestimmt:

K3: $\forall\ t_i \in P,\ \exists\ t_n \in C$:

$$\forall\ w_i \in VECTOR(Ts(t_i))\ .\ (suc_{w_i}(t_i) = id(t_j)) :$$
$$Vk_{w_i}(t_n) = Vk_{w_i}(t_i) \bigcup Vk(t_j) \wedge$$
$$suc_{w_i}(t_n) = suc(t_j)$$

K4: $\forall\ t_i \in P,\ \exists\ t_n \in C$:

$$(\forall\ w_i \in VECTOR(Ts(t_i))\ .\ (suc_{w_i}(t_i) = id(t_j))),$$
$$(\forall\ w_j \in VECTOR(Ts(t_j))),\ \exists\ w_iw_j \in VECTOR(Ts(t_n)) :$$
$$Vk_{w_iw_j}(t_n) = Vk_{w_i}(t_i) \bigcup Vk_{w_j}(t_j)$$
$$\wedge\ suc_{w_iw_j}(t_n) = suc_{w_j}(t_j)$$

Beispiel 6.7.2:

Es seien die fünf Übergangseinheiten t_i, t_m, t_k, t_n und t_j wie in Bild 6.7.3 verknüpft. Zwischen diesen Übergangseinheiten kommen keine Konflikte vor. t_i, t_m, t_k und t_n werden gemäß K3 mit t_j kompaktiert. Das Ergebnis ist in Bild 6.7.4 zu sehen.

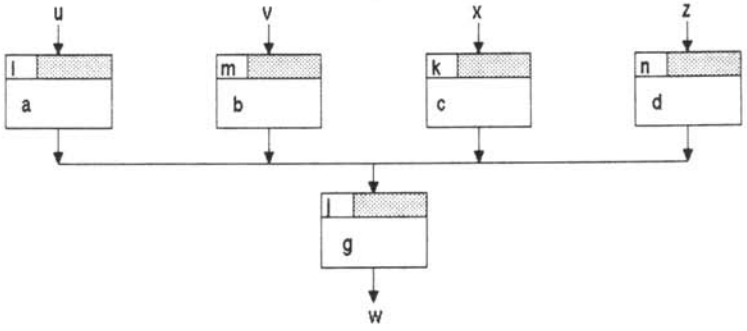


Bild 6.7.3 - Anwendung von K3

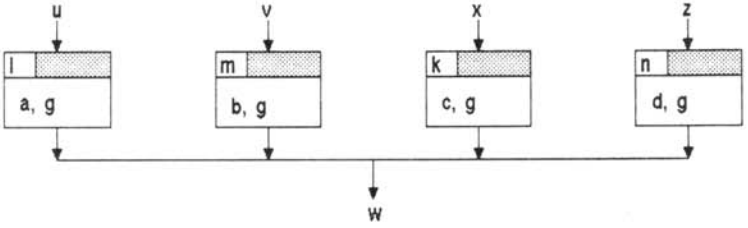


Bild 6.7.4 - Ergebnis der Anwendung von K3

Ende des Beispiels.

6.7.3 *Einschränkungen bei der Anwendung der Kompaktierung*

Beim Kompaktieren werden Übergangseinheiten zusammengefaßt, und es ergeben sich daraus neue Übergangseinheiten mit einer größeren Anzahl von Steuersignalen. Damit erreicht man Parallelität: die Steuersignale einer Übergangseinheit aktivieren gleichzeitig mehrere Komponenten des Operationswerks. Um die Korrektheit dieser gleichzeitigen Aktivierung zu garantieren, müssen die Konfliktfälle des Abschnitts 6.3 vermieden werden. Das wird erreicht, indem man die Anwendung der Kompaktierungsregeln durch die folgenden Restriktionen einschränkt:

a) Restriktion 1:

Gegeben seien t_j und P, die Menge der Vorgänger von t_j Kompaktierung ist nicht möglich, falls:

$\exists\ t_i \in P$:

$$\begin{aligned} &\text{INCOMPATIBILITY} \quad (t_i, t_j) = \text{TRUE} \ \vee \\ &\text{TIMING DEPENDENCE} \ (t_i, t_j) = \text{TRUE} \ \vee \\ &\text{PATH DEPENDENCE} \quad (t_i, t_j) = \text{TRUE} \end{aligned}$$

b) Restriktion 2:

Gegeben seien t_i, t_j, so daß t_j ein Nachfolger oder ein Vorgänger von t_i ist. Seien weiter t_j eine Testeinheit und Ts die Folge der Testsignale von t_j. Kompaktierung ist nicht möglich, falls:

$$\exists\ ts \in Ts(t_j) : \text{CORRELATION}\ (t_i, ts) = \text{TRUE}$$

c) Restriktion 3:

Gegeben seien t_i, t_j, so daß t_i eine Testeinheit und t_j der Nachfolger eines Zweigs von t_i ist. Gegeben sei auch ts, ein instabiles Testsignal von t_i. Kompaktierung ist nicht möglich, falls:

$$\exists\ w_n \in \text{VECTOR}(Ts(t_i)) : suc_{w_n}(t_i) = id(t_j) \wedge ts \text{ bestimmt } w_n$$

Das heißt: ein instabiles Testsignal ist in den Übergang von t_i nach t_j verwickelt.

Beispiel 6.7.3:

Im Bild 6.7.5 sind drei Übergangseinheiten zu sehen: eine Testeinheit t_i und zwei Grundeinheiten t_k und t_j. Für die Testsignale von t_i gilt:

$$\text{INSTABILITY}(ts_p) = \text{FALSE}, \ \text{INSTABILITY}(ts_q) = \text{TRUE}$$

Zwischen diesen Übergangseinheiten kommen keine anderen Konflikte vor.

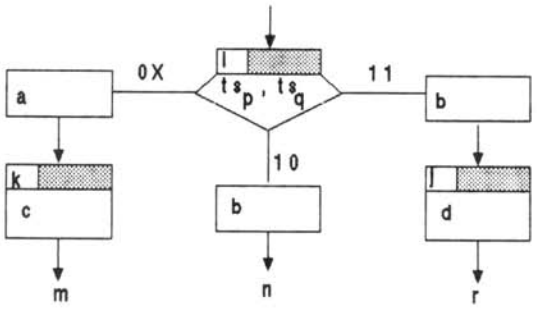


Bild 6.7.5 - Die Restriktion 3 gilt für t_i und t_j

Die Restriktion 3 gilt nicht für die Paare t_i und t_k. ts_q ist in den Übergang von t_i nach t_k nicht verwickelt, da es dabei den Wert 'don't care' annimmt. In diesem Fall kann die Kompaktierung von t_i und t_k gemäß K1 stattfinden. Das Bild 6.7.6 zeigt das Ergebnis dieser Transformation.

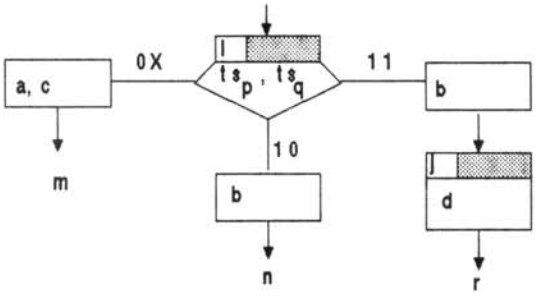


Bild 6.7.6 - Ergebnis der Anwendung von K1 über t_i, t_k

Ende des Beispiels.

Die Restriktion 3 garantiert, daß Hazards wegen eines instabilen Testsignals nicht auftreten können. Sie garantiert auch, daß eine erzeugte Moore-Testeinheit trotz der Anwendung der Kompaktierung ihren Typ behält (vgl. Erzeugungsregel 3, Tabelle 6.1).

d) Restriktion 4:

Die Verlagerung einer Übergangseinheit in eine iterative Sektion ist nicht möglich, falls die Menge der Steuersignale dieser Übergangseinheit Lade- oder Aktivierungssteuersignale enthält. Es ist dadurch verboten, daß eine einmalige Aktivierung von Ladekomponenten oder komplexeren Funktionen in einem Ablauf durch eine mehrfache Aktivierung ersetzt wird. Diese Restriktion vermeidet die unzulässige Ausbreitung von interaktiven Sektionen.

e) Restriktion 5:

Die Verlagerung einer Übergangseinheit in eine iterative Sektion ist nicht möglich, falls diese Übergangseinheit inkompatibel mit irgendeiner anderen Übergangseinheit dieser Sektion ist.

6.8 *Komposition*

Die Transformation Komposition erzeugt eine Sektion, die der Steuerung eines FORK-JOIN-Abschnitts entspricht. Die vollständige Bearbeitung jedes FORK-JOIN-Abschnitts verläuft in drei Phasen, die zweite davon ist die Komposition.

6.8.1 *Bearbeitung eines FORK-JOIN-Abschnitts*

1. Phase: Bearbeitung der Zweige

In der ersten Phase wird jeder nebenläufige Zweig eines FORK-JOIN-Abschnitts getrennt bearbeitet. Jeder Zweig erzeugt eine entsprechende Sektion, die durch die Transformationen Reduktion und Kompaktierung minimiert wird. Die erzeugten Sektionen haben die folgende Eigenschaft:

- Seien Sk_i und Sk_j zwei von verschiedenen Zweigen eines FORK-JOIN-Abschnitts erzeugte Sektionen und $t_i \in Sk_i$ und $t_j \in Sk_j$ je eine Übergangseinheit. Es gilt immer, daß kein Konflikt zwischen t_i und t_j vorkommt.

Diese Eigenschaft erlaubt die Kompaktierung von Übergangseinheiten verschiedener nebenläufiger Sektionen, ohne die Restriktionen 1, 2, 3 und 5 berücksichtigen zu müssen.

2. Phase: Komposition

Nach der Anwendung von Reduktion und Kompaktierung auf jede nebenläufige Sektion, werden diese Sektionen miteinander vermischt. Daraus ergibt sich eine Sektion, S_{comp}, mit folgenden Eigenschaften:

- Die ursprüngliche relative Einordnung der Übergangseinheiten einer nebenläufigen Sektion bleibt erhalten. Das heißt: wenn t_j in einer Sektion Sk_i ein Nachfolger von t_i ist, dann kommt t_j auch in S_{comp} nach (aber nicht unbedingt unmittelbar nach) t_i vor.

- Die verzweigten und iterativen Sektionen aller nebenläufigen Sektionen bleiben unverändert in S_{comp}. Sie werden beim Mischen zusammen mit den Grundeinheiten als unteilbare Einheiten behandelt.

S_{comp} stellt noch einen korrekten Übergangsgraphen dar. Weil die zeitliche Relation der nebenläufigen Knoten eines Verhaltensgraphen unbestimmt ist (vgl. Abschnitt 3.1.3), ist jede Einordnung der Übergangseinheiten in einer S_{comp} noch gültig.

3. Phase: Anwendung der Kompaktierung

Über S_{comp} ist zum Schluß die Transformation Kompaktierung unter Berücksichtigung aller Restriktionen anzuwenden. Es ist schon bekannt, daß die Übergangseinheiten aus einer Sektion nicht mehr zu kompaktieren sind und daß kein Konflikt zwischen Übergangseinheiten verschiedener nebenläufiger Sektionen vorkommt. Diese Erkenntnis vereinfacht die Anwendung der Kompaktierung: nur die Restriktion 4 muß noch überprüft werden.

Das Problem bei der Komposition ist es, die beste Reihenfolge für die Übergangseinheiten so zu wählen, daß die Kompaktierung zu einem Übergangsgraphen mit der kleinsten Anzahl von Zuständen führt. Dieses nicht triviale Problem ist mit hohem Aufwand verbunden. Seine Lösung und eine einfachere Alternative, die allerdings nicht immer das optimale Ergebnis liefert, dafür aber die häufigsten Fälle in der Praxis abdeckt, werden im folgenden eingeführt.

6.8.2 *Die Lösungen des Kompositionsproblems*

Eine Sektion, die zu einem Übergangsgraphen mit dem höchsten Nebenläufigskeitgrad führt, stellt eine optimale Lösung der Komposition dar. Der

Nebenläufigskeitgrad ist hier das Kriterium der Optimierung. Stattdessen kann man ein leicht überprüfbares äquivalentes Kriterium benutzen, nämlich die kleinste Anzahl von Übergangseinheiten. Die optimale Sektion wird als S_{opt} bezeichnet. Es kann mehrere S_{opt} für einen FORK-JOIN-Abschnitt geben.

Ein sicheres Verfahren, um S_{opt} zu finden, umfaßt die Erzeugung aller möglichen S_{comp}, die Anwendung der Kompaktierung auf jede dieser erzeugten Kompositionen und zum Schluß die Auswertung aller Ergebnisse bezüglich des Optimierungskriteriums. Die folgende Gleichung bestimmt die Anzahl As der möglichen Kompositionen. In dieser Gleichung ist m die Anzahl der zu mischenden Sektionen und n_i $(1 \leq i \leq m)$ die Anzahl der unteilbaren Einheiten von Sk_i.

$$As = \frac{(\sum_{i=1}^{m} n_i)\ !}{\prod_{i=1}^{m} (n_i!)}$$

Beispiel 6.8.1

Die Menge der möglichen Kompositionen wird schon für eine kleine Anzahl von Sektionen schwer beherrschbar. Für drei Sektionen mit zwei Einheiten je Sektion sind schon 6!/2!2!2! = 90 Kompositionen zu bearbeiten. Für vier Sektionen mit je drei Einheiten wächst die Anzahl der Kompositionen auf 12!/3!3!3!3! = 369600. Da auf jede dieser 369600 Sektionen noch die Kompaktierungsregeln anzuwenden sind, kommt man zu dem Schluß, daß dieses Verfahren unangemessen ist.

Ende des Beispiels.

Diese aufwendige Lösung läßt sich verbessern, indem man mitberücksichtigt, daß die Kompaktierung nur noch auf Übergangseinheiten verschiedener nebenläufiger Sektionen anzuwenden ist. Die S_{opt} ist ganz wahrscheinlich (aber nicht ausschließlich) unter den am meisten gemischten (*interleaved*) S_{comp} zu finden. Damit kann man alle S_{comp} streichen, die einfach durch die Permutation der Sektionen als Ganzes erzeugt werden. Daraus folgt eine extrem einfache Lösung: eine hoch gemischte S_{comp} wird direkt erzeugt und als fast optimal akzeptiert. Diese direkte Erzeugung erfolgt durch das alternierende Abholen von unteilbaren Einheiten, eine von jeder nebenläufigen Sektion. Diese alternative Lösung bestimmt nicht in jedem Fall die optimale Sektion. Sie erzeugt aber mit geringerem Aufwand und

für die häufigsten Fälle in der Praxis entweder eine fast optimale oder manchmal sogar die optimale Sektion.

Beispiel 6.8.2

Es seien drei durch Reduktion und Kompaktierung minimierte nebenläufige Sektionen (Bild 6.8.1) gegeben.

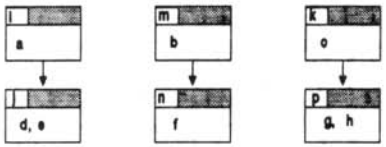
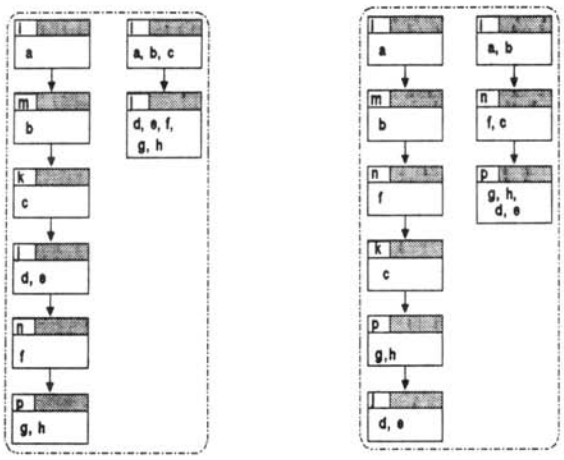


Bild 6.8.1 - Drei nebenläufige Sektionen

Bild 6.8.2 zeigt zwei mögliche von den 90 Kompositionen dieser Sektionen und die entsprechenden Ergebnisse.



(a) (b)

Bild 6.8.2 - Zwei Kompositionen

Die erste Spalte des Bilds 6.8.2.a stellt eine hoch gemischte Komposition dar, die durch alternierendes Abholen von Übergangseinheiten

erzeugt wurde. Diese Komposition führt durch Kompaktierung zu der Sektion mit der kleinsten Anzahl von Zuständen.

Ende des Beispiels.

6.9 *Anpassung des Übergangsgraphen an die Zeitspezifikation*

Nach der Anwendung aller möglichen Transformationen auf eine Sektion wird die durch CYCLE angegebene Zeitspezifikation, falls sie in der Beschreibung des Systems vorhanden ist, überprüft. Die Notation CYCLE bestimmt, wieviele Taktperioden zwischen zwei Operationen liegen müssen. Falls die erzeugte Sektion diese Spezifikation nicht erfüllt, wird sie durch das Einfügen von Verzögerungseinheiten an die Spezifikation angepaßt. Dabei zählt man zu jeder Übergangseinheit eine Taktperiode. Eine Taktperiode entpricht einem CYCLE der Spezifikation.

Diese Anpassung ist nicht immer ausführbar. Es kann passieren, daß es in der erzeugten Sektion mehr Zustände gibt, als es der Anzahl der in der Spezifikation vorhandenen CYCLEs entspricht. In diesem Fall bekommt der Benutzer eine Meldung, und die Synthese betrachtet diese Spezifikation als irrelevant.

Auf die eingefügten Verzögerungseinheiten sind die Transformationen Reduktion und Kompaktierung nicht anzuwenden.

6.10 *Formatierung*

Die Formatierung wandelt einen Übergangsgraphen in eine Übergangstabelle um. Dabei erzeugt jede Übergangseinheit einen Zustand. Mit Ausnahme der Kommunikationseinheiten, die in Kapitel 8 behandelt werden, wandelt sich jede Übergangseinheit gemäß der Tabelle 6.4 in einen Zustand um.

DEF. 6.10.1 - Übergangstabelle

Eine Übergangstabelle U_{tab} ist eine Menge von Zeilen: $U_{tab} = (l_1, \ldots, l_j, \ldots, l_m)$, wobei das Tupel *line* eine Zeile l dieser Tabelle darstellt.

line = (id, Eb, Ab, fz)

Die Komponenten von *line* sind:

- id: Zustandskennzeichen
- Eb: Eingabebelegung (Belegung der Testsignale)
- Ab: Ausgabebelegung (Belegung der Steuersignale)
- fz: Folgezustand

Eine Untermenge dieser Tabelle, in der alle Zeilen das gleiche Zustandskennzeichen enthalten, stellt einen Zustand dar. Dabei ist ein Zustand z_i in einer Übergangstabelle wie folgt definiert:

$$z_i = (l \in U_{tab} \mid id(l) = i)$$

Erzeugter Zustand z_i	Grund-einheit	Verzögerungs-einheit	Test-einheit
Anzahl der Zeilen	1	1	$\lvert Zw \rvert$
Zustandskennzeichen	$id(t_i)$	$id(t_i)$	$id(t_i)$
Testsignale	-	-	$Ts(t_i)$
Steuersignale	$ss \in Cs(t_i)$	-	$ss \in Cs(t_i)$
Folgezustände	$suc(t_i)$	$suc(t_i)$	$Suc(t_i)$

Tabelle 6.4 - Formatierung von t_i

a) *Anzahl der Zeilen*

Eine Testeinheit erzeugt einen Zustand mit so vielen Zeilen, wie es der Anzahl der in ihr enthaltenen Zweige entspricht. Die anderen Übergangseinheiten erzeugen Zustände mit nur einer einzigen Zeile.

b) *Zustandskennzeichen*

Alle Zeilen eines aus einer Testeinheit t_i erzeugten Zustands z_i bekommen das gleiche Zustandskennzeichen.

$$\forall\, l_j \in z_i : id(l_j) = id(t_i)$$

c) *Belegung der Testsignale*

Der Belegungswert aller Testsignale einer Übergangstabelle ist für einen aus einer Grund- oder Verzögerungseinheit erzeugten Zustand 'X' (don't care). Für einen Zustand, den eine Testeinheit erzeugt, gilt:

$\forall\ l_j \in z_i,\ \exists\ w_j \in$ VECTOR(Ts(t_i)) :

die Testsignale von t_i bekommen die Belegung w_j, alle andere Testsignale bekommen 'X'.

d) *Belegung der Steuersignale*

Die Menge Cs(t_i) einer Übergangseinheit t_i entspricht der Menge der aktiven Signale $S_t(z_i)$ des aus dieser t_i erzeugten Zustands z_i (vgl. Abschnitt 5.4). Die Steuersignale von Cs(t_i) bekommen in z_i ihre Tätigkeitswerte, alle andere Ausgabesignale werden in dieser Zeile mit ihren Untätigkeitswerten belegt. In einem aus einer Verzögerungseinheit stammenden Zustand sind alle Steuersignale mit ihren Untätigkeitswerten zu belegen.

Für einen Zustand z_i, den eine Testeinheit t_i erzeugt, sind die Steuersignale jeder Zeile diejenige Steuersignale, welche der entsprechende Zweig von t_i enthält.

$\forall\ l_j \in z_i,\ \exists\ w_j \in$ VECTOR(Ts(t_i)) :

$$Cs(t_j) = \bigcup_n Cs_{k_n} \quad \wedge \quad k_n \in Vk_{w_j}(t_i)$$

Bei der Belegung der Steuersignale muß man beachten, daß der Untätigkeitswert eines Auswahl-Signals, ss ∈ Ass, 'X' (don't care) ist und der eines Lade-Signals, ss ∈ Lss, die Negation seines Tätigkeitswertes ist.

e) *Folgezustand*

$\forall\ l_j \in z_i,\ \exists\ w_j \in$ VECTOR(Ts(t_i)) : $fz(l_j) = suc_{w_j}(t_i)$

Beispiel 6.10.1

Das Bild 6.10.1 zeigt eine iterative Sektion mit vier Übergangseinheiten: t_2, t_3, t_4 und t_5.

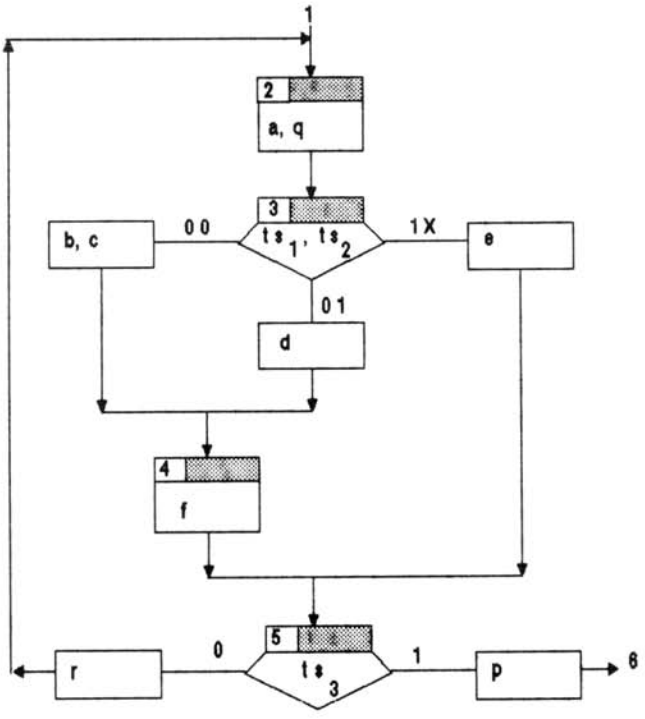


Bild 6.10.1 - Sektion eines Übergangsgraphen

Diese Sektion erzeugt die Übergangstabelle des Bildes 6.10.2. Dabei ist zum Beispiel der Zustand z_2 das Ergebnis der Formatierung der Übergangseinheit t_2. Testsignale sind in t_2 irrelevant, deshalb werden die Eingabesignale von z_2 mit 'X' belegt. Die Synthese des Operationswerks bestimmt, daß $Cs_{k_a} = \{ss_1=0, ss_2=1\}$ und $Cs_{k_q} = \{ss_k=1\}$ ist. Diese sind die Steuersignale, welche die Realisierungen der Knoten a bzw. q aktivieren. Damit bestimmt man $Cs(t_2)$, $Cs(t_2) = Cs_{k_a} \cup Cs_{k_q}$, die Steuersignale von t_2. Diese Steuersignale werden in z_2 mit den entsprechenden Tätigkeitswerten belegt und alle anderen Ausgabesignale mit ihren Untätigkeitswerten.

id	Eingabebelegung					Ausgabebelegung				fz
	ts_1	ts_2	ts_3	...	ts_m	ss_1	ss_2	...	ss_k	
1			. . .				. . .			2
2	X	X	X	...	X	0	1	.	1	3
3	0	0	X	...	X		. . .			4
3	0	1	X	...	X		. . .			4
3	1	X	X	...	X		. . .			5
4	X	X	X	...	X		. . .			5
5	X	X	0	...	X		. . .			2
5	X	X	1	...	X		. . .			6
6			. . .				. . .			..

Bild 6.10.2 - Übergangstabelle

Ende des Beispiels.

6.11 *Bewertung des Verfahrens*

Die Komplexität des in diesem Kapitel vorliegenden Verfahrens zur Synthese der Steuerung eines Prozesses und der Nebenläufigkeitsgrad des Ergebnisses sind die hier verwendeten Kriterien zur Bewertung dieses Verfahrens.

6.11.1 *Komplexität des Syntheseverfahrens*

Die Transformationen des Syntheseverfahrens sind alle lokal. Das bedeudet, daß sie entweder auf einzelne Übergangseinheiten (Erzeugung, Reduktion und Formatierung) oder höchstens auf benachbarte Übergangseinheiten (Kompaktierung und Komposition) anwendbar sind. Es ist deshalb zu erwarten, daß die Komplexität des Verfahrens linear ist. Die Anwendung des O-Kalküls zur Ermittlung der Komplexität jeder Transformation zeigt, daß dieser Erwartung erfüllt wird.

a) *Komplexität der Erzeugung der Übergangseinheiten*

Die Komplexität dieser Transformation ist für einen Verhaltensgraphen mit n Knoten O(n), da jeder Knoten nur einmal besucht wird.

b) *Komplexität der Reduktion*

Jede erzeugte Übergangseinheit muß bei der Reduktion genau einmal überprüft werden, so daß die Komplexität dieser Transformation O(n) ist.

c) *Komplexität der Kompaktierung*

Der schlimmste Fall für die Kompaktierung (der allerdings zum besten Ergebnis führt) tritt ein, wenn alle Übergangseinheiten eines Übergangsgraphen kompaktiert werden können. Zuerst werden n Übergangseinheiten paarweise kompaktiert. Daraus ergeben sich n/2 Übergangseinheiten, die wieder paarweise kompaktiert werden können. Das wiederholt sich, solange Kompaktierung noch möglich ist. Damit ist die Anzahl der Übergangseinheiten, die diese Transformation bearbeitet:

$$n + n/2 + n/4 + n/8 + \ldots < 2n-1$$

Da das O-Kalkül die Unterdrückung der Konstanten erlaubt, erreicht man, daß die Komplexität hier O(n) ist.

d) *Komplexität der Komposition*

Im Abschnitt 6.8.2 wurde schon der hohe Aufwand bei der Suche nach der optimalen Komposition erwähnt (vgl. die Berechnung der Anzahl der möglichen Kompositionen). Die vorgeschlagene alternative Lösung hat dagegen die lineare Komplexität O(n), wobei n die Summe aller Übergangseinheiten der in die Komposition verwickelten nebenläufigen Sektionen darstellt.

e) *Komplexität der Formatierung*

Die Komplexität dieser Transformation ist für einen Übergangsgraphen mit n Übergangseinheiten O(n), da jeder Knoten nur einmal besucht wird.

Die Komplexität jeder Transformation ist linear, und die gesamte Komplexität, berechnet durch die Addition der Komplexitäten aller Transformationen, ist auch linear.

6.11.2 *Einfluß auf den Nebenläufigkeitsgrad*

Die Transformationen, die den Nebenläufigkeitsgrad beinflussen, sind allein die Reduktion und die Kompaktierung, weil diese Übergangseinheiten entfernen. Um diesen Einfluß zu ermitteln, wird der Nebenläufigkeitsgrad eines Ablaufs verwendet (die Definitionen von Φ (s. 5.3) sind auch bei Übergangseinheiten anwendbar, da sie eine interne Darstellung von Zuständen sind).

Es sei $\Phi_a(a_j)$ der Nebenläufigkeitsgrad eines Ablaufs vor der Anwendung der Reduktion und a_j ein nicht iterativer Ablauf, der eine zu entfernende Übergangseinheit t_k enthält.

$$\Phi_a(a_j) = \frac{\sum_{i=1}^{n} \Phi_z(t_i)}{n}$$

Dabei ist n die Anzahl der Übergangseinheiten in a_j. Nach der Entfernung von t_k durch R1 bleiben n-1 Übergangseinheiten in a_j übrig. Der Term:

$$\sum_{i=1}^{n} \Phi_z(t_i) = \Phi_z(t_k) + \sum_{i=1}^{n-1} \Phi_z(t_i),\ t_i \neq t_k$$

bleibt unverändert, weil t_k keine Steuersignale liefert. Damit ist $\Phi_z(t_k) = 0$. Daraus ergibt sich der erhöhte Wert von Φ_a':

$$\Phi_a'(a_j) = \Phi_a(a_j) \cdot n/n\text{-}1$$

Der Einfluß auf den Nebenläufigkeitsgrad des Steuerwerks hängt von der Häufigkeit des Vorkommens von t_k in den verschiedenen Abläufen dieses Steuerwerks ab. Φ_c wird trotzdem bei jeder Anwendung von R1 erhöht, weil die Terme der Summe (d.h. die Φ_a) entweder gleich oder erhöht vorkommen.

Die Reduktionen R2 und R3 entfernen sinnlose Verzweigungen. Sie ändern trotzdem die Anzahl der Abläufe eines Steuerwerks nicht und haben deswegen keinen Einfluß auf den Nebenläufigkeitsgrad. Eine von R2 oder R3 zu entfernende Verzweigung bestimmt keinen neuen Ablauf, weil alle ihre Zweige die gleichen Steuersignale liefern. Man kann auch sagen, daß die Abläufe, in denen die veschiedenen Zweige dieser Verzweigung den einzigen Unterschied darstellen, alle gleich sind, und deshalb kommen sie nicht in der Berechnung von Φ_c vor.

Sei $\Phi_a(a_j)$ jetzt der Nebenläufigkeitsgrad eines Ablaufs vor der Anwendung der Kompaktierung, und sei $\Phi_a'(a_j)$ der Nebenläufigkeitsgrad nach der Kompaktierung, und sei a_j ein nicht iterativer Ablauf, der die zu kompaktierten Übergangseinheiten t_k und t_p enthält, so wird:

$$\Phi_a(a_j) = \frac{\sum_{i=1}^{n-2} \Phi_z(t_i) + \Phi_z(t_k) + \Phi_z(t_p)}{n}$$

$$\Phi_a'(a_j) = \frac{\sum_{i=1}^{n-2} \Phi_z(t_i) + \Phi_z(t_r)}{n - 1}$$

Es sei n die Anzahl der Übergangseinheiten im nicht kompaktierten a_j. Nach der Kompaktierung von t_k und t_p bleiben n-1 Übergangseinheiten in a_j. Weil im allgemeinen die Steuersignale von t_k und t_p disjunkt sind, gilt:

$$\Phi_z(t_r) = \Phi_z(t_k) + \Phi_z(t_p)$$

so daß: $\Phi_a'(a_j) = \Phi_a(a_j) \cdot n/n\text{-}1$

Es gibt trotzdem Fälle, in denen:

$$\max(\Phi_z(t_k), \Phi_z(t_p)) \leq \Phi_z(t_r) < \Phi_z(t_k) + \Phi_z(t_p)$$

Sie treten dann auf, wenn die Mengen der Steuersignale von t_k und t_p eine gemeinsame Untermenge von Steuersignale enthalten. In der Praxis ist aber:

$$|Cs_k \cap Cs_p| \ll |Cs_k \cup Cs_p|$$

d.h., die Anzahl der gemeinsamen Steuersignale ist viel kleiner als die Summe der aktiven Steuersignale und damit ist deren Einfluß auf Φ geringfügig. Infolgedessen kann man behaupten, daß die Anwendung der Kompaktierung den Wert von Φ_a um den Faktor n/n-1 erhöht. Der Einfluß auf den Nebenläufigkeitsgrad des Steuerwerks hängt von der Häufigkeit des Vorkommens von t_k und t_p in den verschiedenen Abläufen dieses Steuerwerks ab. Φ_c steigt bei jeder Anwendung der Kompaktierung, weil es mindestens einen Wert Φ_a gibt, der erhöht vorkommt.

7. Aktivierung eines Moduls

Ein Modul entspricht einer Menge der vom Benutzer beschriebenen Funktionen oder einer nicht primitiven Operation. Selbstgesteuerte Moduln verfügen über ein internes Steuerwerk, das nach dem in Kapitel 6 beschriebenen Verfahren erzeugt werden kann. Das folgende Kapitel behandelt das Problem der Aktivierung dieses Steuerwerks. Von Aktivierung versteht sicht hier die Zusendung zu einem Modul von einer gewissen Reihenfolge von Steuersignalen, die erlaubt, den Modul korrekt zu starten, die nötige Parametern zu übertragen, und den Modul bereit zu einem neuen Starten zu verlassen.

7.1 *Integration eines Moduls in einen Prozeß*

Ein Modul ist im Rahmen dieser Arbeit eine unabhängige Einheit, die getrennt realisiert und dann in das System integriert wird. Ein Modul entspricht entweder einer Menge der vom Benutzer beschriebenen Funktionen oder einer in der Beschreibungssprache vorhandenen nicht primitiven Operation. Die Unterschiede zwischen den Benutzerfunktionen und den nicht primitiven Operationen liegt darin, daß eine Realisierung für die letztere schon vorhanden ist.

a) *Nicht primitive Operationen:* Sie sind diejenige, die in Gegensatz zu Operationen wie +, -, OR, TRANSFER mehr als eine Taktperiode (und deshalb mehr als einen Zustand) zur Durchführung ihre Funktionen benötigen (z.B. Multiplikation, Division).

b) *Benutzerfunktionen:* Sie werden vom Benutzer durch die Notation MODULE in BABEL (oder DSL) beschrieben und wie die Operationen der Sprache verwendet. Eine MODULE-Vereinbarung definiert eine oder mehrere Funktionen. Diese verfügen in der Regel über allgemeine Anschlüsse und ein einziges Steuerwerk, deshalb kann nur eine Funktion eines Benutzermoduls jedesmal aktiviert werden. Das Verfahren des Kapitels 6 erzeugt aus der Beschreibung eines Benutzermoduls dessen interne Steuerung. Dieses Modul wird dabei genau wie ein terminaler Prozeß behandelt.

Die Integration eines Moduls in einen Prozeß umfaßt die Einbettung seines Datenflusses in das Operationswerk des Prozesses und die Integration seiner Steuerung in die Steuerung des Prozesses (Bild 7.1.1). Es gibt Moduln, die keine Steuerung benötigen. Die Integration dieser Art von Moduln ist deshalb allein eine Aufgabe der Synthese des Operationswerks.

Diese bestimmt auch, wieviele Realisierungen jedes Moduls erforderlich sind. Dabei muß man bei jedem Modulaufruf erkennen, um welche verfügbare Realisierung es sich handelt.

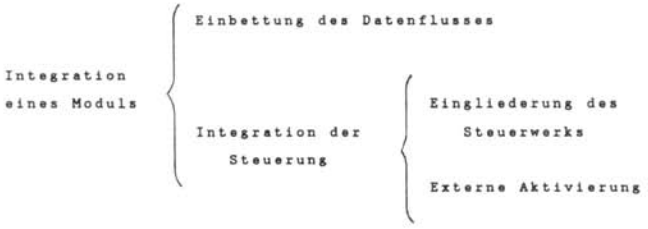


Bild 7.1.1 - Integration eines Moduls

Die Integration der Steuerung eines Moduls in die Steuerung eines Prozesses findet während der Bearbeitung der Knoten, die diesen Modul aufrufen, statt. Für jeden realisierten Modul benötigt man entweder die Spezifikation seiner internen Steuerung oder die Spezifikation der Reihenfolge der Steuersignale, welche die Aktivierung des Moduls ermöglichen. Beide Spezifikationen, wenn überhaupt vorhanden, kommen als Übergangsgraphen vor. Die Spezifikation der Aktivierung eines Moduls wird von dem Benutzer durch eine zu diesem Zweck in der Beschreibungssprache vorhandene Notation bestimmt (s.u. 7.3). Diese Aktivierung führt eine Aktivierungssektion im aufrufenden Prozeß durch.

Für die Integration der Steuerung sind zwei Strategien möglich:

a) *Eingliederung* - Der Übergangsgraph, der die interne Steuerung des Moduls darstellt, wird in die Steuerung des Prozesses eingegliedert. Die Eingliederung findet beim Ersetzen der Knoten, die diesen Modul aufrufen, statt, indem der betreffende Übergangsgraph eingesetzt wird (Bild 7.1.2).

b) *Externe Aktivierung* - Nicht die interne Steuerung, sondern eine der Aktivierungssektionen des Moduls wird eingegliedert (Bild 7.1.3). Die Steuerung des Moduls bleibt als ein getrenntes Steuerwerk erhalten und wird von der Steuerung des Prozesses durch die eingegliederte Aktivierungssektion aktiviert. In diesem Fall ist das Steuerwerk des Moduls der Steuerung des Prozesses untergeordnet.

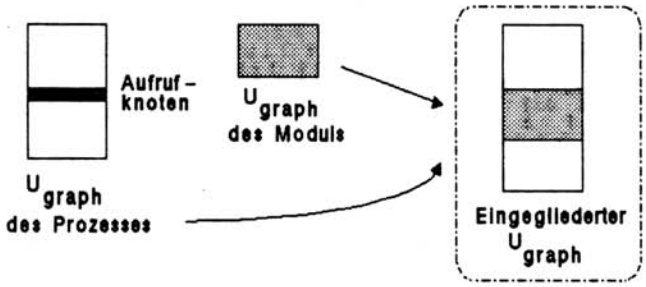


Bild 7.1.2 - Eingliederung der Steuerung

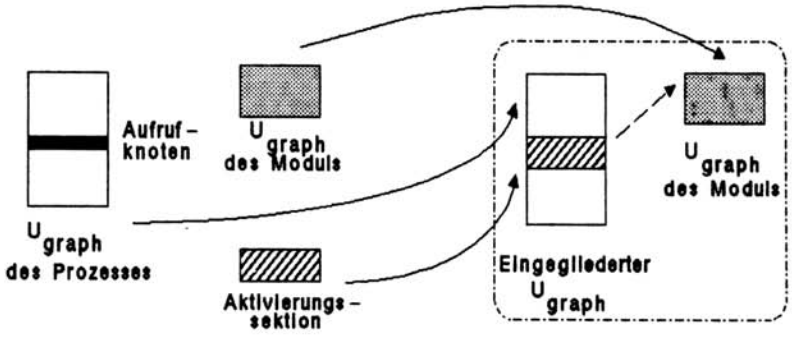


Bild 7.1.3 - Externe Aktivierung

Die Kriterien für die Wahl der Strategie der Integration der Steuerung sind folgende:

- Der Modul ist eine schon fertige vollständige digitale Komponente. Hier ist die externe Aktivierung die einzige mögliche Wahl.

- Der Modul ist noch zu realisieren. In diesem Fall kann die Häufigkeit des Modulaufrufs entscheiden, welche Strategie zur Integration besser geeignet ist. Es können auch andere Kriterien in Frage kommen, wie zum Beispiel die Optimierung der Plazierung der Komponenten und die Minimierung der Verzögerungen bei der Übertragung der Steuersignale. In der laufenden Phase des Entwurfszyklus sind jedoch diese Probleme noch nicht behandelt. Hier wäre die Anwendung einer *' knowlegde base'*

vorteilhaft, um die beste Entscheidung treffen zu können.

7.2 *Externe Aktivierung*

7.2.1 *Umfang der externen Aktivierung*

Die Aktivierung eines Moduls umfaßt:

- Das Starten des Moduls.
- Die Übertragung der Parameter.
- Die Freigabe des Moduls.

Diese Tätigkeiten werden von dem Steuerwerk des Prozesses, der diesen Modul aufruft, durchgeführt. Die Spezifikation dieser Tätigkeiten steht in der Beschreibung des Moduls unter der Notation PERFORMED FUNCTION [CaWe84]. Aus dieser Spezifikation werden je nach Anzahl der Funktionen in diesem Modul eine oder mehrere Aktivierungssektionen erzeugt. Es wird angenommen, daß:

- die Spezifikation dieser Tätigkeiten korrekt und vollständig ist,

- die Aktivierung in jedem Moment sofort nach der Freigabe des Moduls möglich ist.

Nach der Freigabe eines Moduls ist dieser Modul vom Prozeß aus gesehen inaktiv. Die Steuersignale bzw. deren Belegungswerte, die zu einer korrekten Freigabe des Moduls führen, stehen innerhalb der PERFORMED-FUNCTION-Spezifikation unter der Notation HOLD.

7.2.2 *Berechnung des Nebenläufigkeitsgrades*

Die Steuersignale einer Aktivierungssektion dienen ausschließlich dazu, einen bestimmten Modul zu aktivieren (Aktivierungssignale) oder um Parameter zu übertragen (Auswahl- oder Lade-Steuersignale). Die wirksamen aktiven Steuersignale sind während der Aktivität des Moduls letzlich nur die, die zum Steuerwerk dieses Moduls gehören. Deshalb kommen die Steuersignale einer Aktivierungssektion nicht in der Berechnung des Nebenläufigkeitsgrades vor.

Es sei $\Phi_a(a_p)$ der Nebenläufigkeitsgrad eines Ablaufs a_p, der p Übergangseinheiten umfaßt und einmal den Modul M aufruft. In der Bestimmung von $\Phi_a(a_p)$ ist die Aktivierungssektion dieses Moduls nicht inbegriffen. Den Nebenläufigkeitsgrad von a_p einschließlich der Aktivität des Moduls M stellt man als $\Phi_a(a_p^M)$ dar; er wird wie folgt bestimmt:

$$(a)\quad \Phi_a(a_p^M) = \frac{\Phi_a(a_p) \cdot p + \Phi_c(c_m) \cdot m}{p + m}$$

c_m - Steuerwerk des Moduls
m - Anzahl der Zustände von c_m
p - Länge von a_p ohne die Aktivierungssektion

Dabei ist $\Phi_c(c_m)$ der Nebenläufigkeitsgrad des Moduls M. $\Phi_c(c_m)$ beinflußt den Nebenläufigkeitsgrad des Prozesses nur durch die Abläufe, in denen Aufrufe dieses Moduls vorkommen. Dabei ist dieser Einfluß durch den Faktor $m/_{(p+m)}$ bestimmt. Dieser Faktor stellt das Verhältnis zwischen der Periode der Aktivität des Moduls und dem gesamten Ablaufzyklus dar. In anderen Worten, der Nebenläufigkeitsgrad eines Moduls zeigt nur während seiner Aktivität Wirkung auf den Nebenläufigkeitsgrad eines Prozesses. Falls ein Ablauf sequentielle Aufrufe mehrerer Moduln enthält, dann gilt die Gleichung (b).

$$(b)\quad \Phi_a(a_p^{M_1;\ldots;M_n}) = \frac{\Phi_a(a_p) \cdot p + \sum_{i=1}^{n} (\Phi_c(c_i) \cdot m_i)}{p + \sum_{i=1}^{n} m_i}$$

c_i - Steuerwerk des Moduls M_i
m_i - Anzahl der Zustände von c_i

Wenn mehrere Modulaufrufe innerhalb eines FORK-JOIN-Abschnitts vorkommen, wendet man die Transformation 'Komposition' und 'Kompaktierung' auf die Aktivierungssektionen dieser Moduln an. Daraus ergibt sich eine einzige Aktivierungssektion, die alle Moduln gleichzeitig aktiviert. Bei der Berechnung des Nebenläufigkeitsgrades eines Ablaufs a_p, der diese gleichzeitige Aktivierung einmal enthält, kommt der Faktor m_x vor, wobei:

$$(c)\quad m_x = \mathrm{Max}(m_1,\ldots,m_k)$$

In der Gleichung (c) ist m_i die Anzahl der Zustände des Moduls M_i und k die Anzahl der gleichzeitig aktiven Moduln. Der Nebenläufigkeitsgrad des Ablaufs a_p ist dann:

$$(d)\quad \Phi_a(a_p^{M_1,\cdots,M_n}) = \frac{\Phi_a(a_p)\ .\ p\ +\ \sum_{i=1}^{k}(\Phi_c(c_i)\ .\ m_i)}{p + m_x}$$

Die Anwendung von Moduln, insbesondere wenn sie häufig aktiv sind und/oder deren Aufrufe parallel vorkommen, erhöht den Nebenläufigkeitsgrad eines Prozesses und ist deshalb aus dieser Sicht vorteilhaft.

7.3 *Erzeugung der Aktivierungssektion*

Der BABEL- (oder DSL-) Übersetzer erzeugt aus jeder Spezifikation der Aktivierung eines Moduls einen Verhaltensgraphen, der einem Abschnitt entspricht. Aus diesem Abschnitt (Aktivierungsabschnitt) wird dann eine Aktivierungssektion abgeleitet. Die Umwandlung eines Abschnitts in eine Sektion, obwohl dem in Kapitel 6 präsentierten Verfahren ähnlich, unterscheidet sich davon durch die folgenden Merkmale:

- Es ist kein Datenfluß zu realisieren. Die Zieloperanden in diesem Abschnitt sind entweder die Parameter des Moduls oder Aktivierungssignale.

- Es gilt für alle aus den Knoten dieses Abschnitts erzeugten Übergangseinheiten, daß sie untereinander zeitabhängig sind. Der Grund hierfür ist, daß die Spezifikation der Aktivierung eines Moduls letztendlich eine Zeitspezifikation darstellt. Infolgedessen ist die Transformation 'Kompaktierung' auf die Übergangseinheiten einer Aktivierungssektion nicht anwendbar.

7.3.1 *Die Spezifikation der Aktivierung eines Moduls*

Die unter der Notation CONTROL beschriebene Aktivierung eines Moduls enthält die Spezifikation der Aktivierungssignale und die Spezifikation der Verfügbarkeit der Parameter und der Ready-Signale, die dieser Modul benötigt, um eine seiner Funktionen durchzuführen. Die Notation zur Beschreibung der Verfügbarkeit der Parameter ist: *identifier* := *A*, wobei *identifier* ein Eingangs- oder Ausgangsanschluß ist und 'A' für *Available* steht. Diese Notation setzt fest, daß die als Anschlüsse (*INTERFACE*) vereinbarten Parameter während des entsprechenden Zyklus verfügbar sein

sollen. Ein Ready-Signal, welches die Vollendung der Bearbeitung eines Moduls bezeichnet, wird durch die Notation: *WAIT identifier=value* bestimmt, wobei *identifier* ein vereinbarter Ausgangsanschluß sein muß.

In einem Abschnitt, der die oben beschriebene Aktivierung eines Moduls darstellt, sind ausschließlich die folgenden Knoten vorhanden:

a) *Operationsknoten*: Die vorhandenen Operationen stellen einfache Zuweisungen von Konstanten zu Aktivierungssignalen dar. Diese Konstanten sind ausschließlich die für Steuersignale erlaubten Belegungswerte (0,1,X).

b) *Spezifikationsknoten*: Sie sind eine besondere Art von Operationsknoten und dienen dazu, um die Verfügbarkeit der Parameter und der Ready-Signale zu spezifizieren. Diese Knoten sind auch Operationen in dem Sinn, daß sie Übertragungen von Informationsträgern darstellen: Übertragung der Parameter des Moduls und Übertragung der Ready-Signale aus dem Modul.

c) *Strukturknoten*: Die einzigen Strukturknoten, die in diesem Abschnitt vorkommen, sind FORK- und JOIN-Knoten. Sie stellen im Gegensatz zu der schwachen nebenläufigen Relation zwischen den Knoten in einem FORK-JOIN -Abschnitt einer Sequenz (s. 3.1.3) feste Parallelität dar.

Zusätzlich lassen sich aus der Spezifikation der Aktivierung die folgenden Elemente bestimmen:

a) *Die Aktivierungssignale*: Alle vereinbarten Eingangs-Anschlüsse, die in der Spezifikation vorkommen und keine Modulparameter darstellen, sind Aktivierungssignale für diesen Modul.

b) *Die Ready-Signale*: Alle vereinbarten Ausgangs-Anschlüsse, die in der Spezifikation in einer WAIT-Notation vorkommen, stellen Ready-Signale dar.

7.3.2 *Die Übergangseinheiten einer Aktivierungssektion*

Die Übergangseinheiten einer Aktivierungssektion werden bei der Bearbeitung der Knoten eines Aktivierungsabschnitts gemäß der Tabelle 7.1 erzeugt.

Knoten k_i	Übergangseinheit t_i type	$Ts(t_i)$	$Cs(t_i)$
Operation	single	-	Zieloperand
Strukturknoten	-	-	-
S Ein.Param.	single	-	Auswahl der entspr.Param.
e Aus.Param.	single	-	Laden der entspr.Param.
z . Ready-Sig.	test	Ready-Signal	-

Tabelle 7.1 - Übergangseinheiten einer Aktivierungssektion

Die Operationen eines Aktivierungsabschnitts wandeln sich in Grundeinheiten um. Dabei enthält die Menge der Steuersignale jeder erzeugten Übergangseinheiten t_i ausschließlich Aktivierungssignale. Diese sind die Zieloperanden dieser Operation, die in t_i mit den entsprechenden zugewiesenen Werten belegt werden.

Die Knoten, welche die Verfügbarkeit von Parametern darstellen, erzeugen auch Grundeinheiten. Die Steuersignale dieser Übergangseinheiten müssen dann in diesem Fall die entsprechenden Eingangsparameter des Moduls auswählen oder die Ausgangsparametern zweckmäßig laden.

Die Knoten, die Ready-Signale spezifizieren, sind die einzigen, die Testeinheiten erzeugen. Diese Testeinheiten haben das folgende Merkmal (Bild 7.3.1):

$$\exists\ w_r \in \mathrm{VECTOR}(Ts(t_i)) : \mathrm{suc}_{w_r}(t_i) \neq \mathrm{id}(t_i)$$
$$\forall\ w,\ w \neq w_r \ \wedge\ w \in \mathrm{VECTOR}(Ts(t_i)) : \mathrm{suc}_{w}(t_i) = \mathrm{id}(t_i)$$

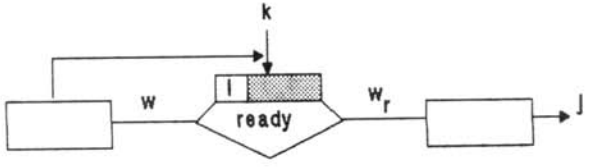


Bild 7.3.1 - Testeinheit einer Aktivierungssektion

Die Mengen Suc und Pred jeder erzeugten Übergangseinheit t_i spiegeln die Vorgänger-Nachfolger-Relation zwischen den Knoten des Aktivierungsabschnitts wider, welche die Funktionen SUC und PRED über diesen Knoten darstellen. Weil die Aktivierungssektion eines Moduls in einem Prozeß mehrfach vorkommen kann, sind das Kennzeichen (*id*) jeder erzeugten Übergangseinheit und die Inhalte deren Mengen Pred und Suc verschieblich (relocatable). Sie bekommen erst in der Kopie der Aktivierungssektion, die in den Übergangsgraphen des aufrufenden Prozesses eingefügt wird, einen festen Wert.

Die Strukturknoten erzeugen keine Übergangseinheit. Ein FORK-JOIN-Abschnitt eines Aktivierungsabschnitts hat als einziges mögliches Format:

FORK-Knoten $k_1,k_2,\ldots,k_n$ JOIN-Knoten

Die Knoten, die dabei vorkommen, erzeugen alle zusammen eine einzige Übergangseinheit, die gemäß der folgenden Regel abzuleiten ist:

t_i : type(t_i) = single
Ts(t_i) = ( )
Vk(t_i) = ($k_1,k_2,\ldots,k_n$)
Cs(t_i) = Zieloperanden der Knoten von Vk(t_i)

7.4 *Beispiel*

Das folgende Beispiel zeigt die Beschreibung der Aktivierung eines Moduls, einen aus dieser Beschreibung erzeugten Aktivierungsabschnitt und letztlich die entsprechende Aktivierungssektion. Der vom Benutzer beschriebene Modul bestehe aus zwei Funktionen *multiplication* und *square* und habe die folgenden vereinbarten Anschlüsse:

Vereinbarung der Anschlüsse:

```
INTERFACE  ck  : CLOCK;
           in_a(15..0), in_b(15..0) : INPUT;
           enable, sel_function     : INPUT;
           out_c(31..0) : OUTPUT FANOUT 3;
           ready        : OUTPUT;
```

Ende der Vereinbarung.

Die folgende Spezifikation gilt für die Aktivierung dieses Moduls:

Spezifikation der externen Aktivierung:

```
PERFORMED FUNCTION
    HOLD enable := 0 END

    out_c := #multiplication(in_a,in_b)
    CONTROL  enable:=1; [sel_function:=1, in_a:=A];
             in_b:=A; WAIT ready=1; out_c:=A
    END;

    out_c := #square(in_a)
    CONTROL  enable:=1; [sel_function:=0, in_a:=A];
             WAIT ready=1; out_c:=A
    END;
```

Ende der Spezifikation.

Solange der Anschluß *enable* mit dem Wert Null belegt wird, ist der Modul gemäß der HOLD-Spezifikation inaktiv. Die Belegung des Anschlusses *sel_function* bestimmt, welche Funktion durchzuführen ist. Je nach Funktion müssen entweder ein oder zwei Parameter übertragen werden. Der Modul setzt das Signal *ready*, sobald die ausgewählte Funktion beendet ist. Kurz danach steht das Ergebnis an den Anschlüssen *out_c* zur Verfügung. Es wird angenommen, daß diese Spezifikation korrekt und vollständig ist. Weder der Übersetzer noch die Synthese prüft deren Korrektheit, eine solche Überprüfung ist vielmehr (noch) eine Aufgabe des Entwerfers.

Der Übersetzer erzeugt einen Aktivierungsabschnitt für jede CONTROL-Spezifikation. Der folgende Verhaltensgraph (Bild 7.4.1) stellt den Aktivierungsabschnitt für die Funktion *square* dar.

```
NAME : #square              TYPE : PREFIX
INPUT  in_a(15..0) : IN     OUTPUT  out_c(31..1) : OUT

CONTROL

 Index       : 10      Tag : SO          Operation : TR
 Inputs      : 1       Outputs    : enable(0..0)   : IN
 Predecessor: *        Condition  : X    Successor : 13

 Index       : 13      Tag : PB          Operation : *
 Inputs      : *       Outputs    : *
 Predecessor: 10       Condition  : X    Successor : 11  12
```

```
Index       : 11      Tag : SO          Operation : TR
Inputs      : 0       Outputs    : sel_function(0..0) : IN
Predecessor: 13       Condition  : X    Successor : 14

Index       : 12      Tag : SO          Operation : TR
Inputs      : A       Outputs    : in_a(15..0)     : IN
Predecessor: 13       Condition  : X    Successor : 14

Index       : 14      Tag : PE          Operation : *
Inputs      : *       Outputs    : *
Predecessor: 11  12   Condition  : X    Successor : 15

Index       : 15      Tag : SO          Operation : WAIT
Inputs      : ready(0..0) : OUT         Outputs   : 1
Predecessor: 14       Condition  : X    Successor : 16

Index       : 16      Tag : SO          Operation : TR
Inputs      : A       Outputs    : out_c(31..1)    : OUT
Predecessor: 15       Condition  : X    Successor : *
```

Bild 7.4.1 - Aktivierungsabschnitt für die Funktion 'square'

Zum Schluß wird aus dem Aktivierungsabschnitt des Bildes 7.4.1 gemäß den Regeln des Abschnitts 7.3.2 die folgende Aktivierungssektion (Bild 7.4.2) abgeleitet:

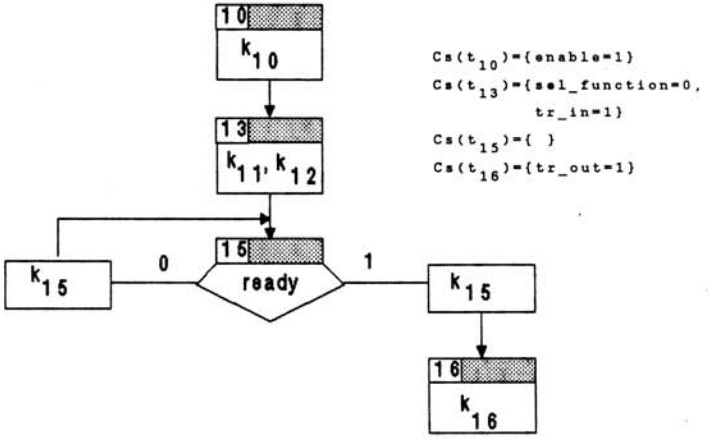


Bild 7.4.2 - Aktivierungssektion für 'square'

Im Bild 7.4.2 stellen *tr_in* und *tr_out* die erforderlichen Steuersignale zur Übertragung der Eingangs- und Ausgangsparameter dar.

8. Synthese mehrerer kooperierender Prozesse

Gegeben sei ein System, das eine Menge von Prozessen umfaßt. Dabei seien die Interprozeßmoduln die Elementen des Systems, welche die Interaktion zwischen diesen Prozessen erlauben. Dieses Kapitel behandelt das Problem der Synthese der Interprozeßmoduln und deren Integration in das System.

8.1 *Synthesekonzept*

Gemäß den Definitionen des Kapitels 3 besteht ein System aus einer Menge von verkörperten Prozessen und einer Menge von verkörperten Interprozeßmoduln, wobei die Interprozeßmoduln die Elemente der Interaktion zwischen den Prozessen darstellen.

Für jeden verkörperten Prozeß wird ein Übergangsgraph (oder mehrere, wenn Moduln vorhanden sind) nach dem Syntheseverfahren des Kapitel 6 (bzw. 7) erzeugt. Der Übergangsgraph eines Prozesses enthält die Kommunikationseinheiten (vgl. Abschnitt 6.2.2), die aus den im Verhaltensgraphen des Prozesses vorhandenen Anforderungsknoten (vgl. Abschnitt 3.4.3) abgeleitet wurden. Diese Einheiten stellen keine Steuerung dar und müssen noch vor der Formatierung durch Sektionen ersetzt werden, welche die erforderlichen Anforderungen von Aktionen ausführen. Eine Sektion, die eine aus einem Anforderungsknoten a_k abgeleitete Kommunikationseinheit ersetzt, nennt man Anforderungssektion (Sa_k).

Jeder verkörperte Interprozeßmodul IPM=(GM,AGMs,SM) wird wie folgt behandelt:

- Die Synthese des Datenpfads realisiert ein einziges Operationswerk aus den Verhaltensgraphen aller Aktionen des IPM's. Dabei sind die Informationsträger des Datenpfads die gemeinsamen Mittel (GM) des IPM's.

- Die Synthese der Steuerung erzeugt für jede Aktion nach dem Verfahren des 6. Kapitels einen verschieblichen (*relocatable*) Übergangsgraphen. In diesem Übergangsgraphen können Kommunikationseinheiten vorkommen, die aus Warte-Knoten (w_k) abgeleitet wurden. Diese Einheiten müssen durch Sektionen ersetzt werden, welche dem Verhalten eines Warte-Abschnitts darstellen (vgl. Abschnitt 3.2.5). So eine Sektion nennt man Warte-Sektion (Sw_k).

Die Transformation über einen Übergangsgraphen, die eine Kommunikationseinheit durch eine Anforderungssektion Sa_k bzw. eine Wartesektion Sw_k ersetzt, heißt Expansion. Diese Transformation tritt bezüglich des entsprechenden Knotens der zu ersetzenden Kommunikationseinheit t_i in zwei Variationen E1 und E2 auf.

E1: $a_k \in Vk(t_i)$

$$CONC(INS(G_1,t_i),G_2) \overset{E1}{\Rightarrow} CONC(CONC(G_1,CONC(Sa_k,G_{Aktion})),G_2)$$

E2: $w_k \in Vk(t_i)$

$$CONC(INS(G_1,t_i),G_2) \overset{E2}{\Rightarrow} CONC(CONC(G_1,Sw_k),G_2)$$

Wobei E1 ausschließlich auf Übergangsgraphen von Prozessen und E2 auf Übergangsgraphen von Aktionen anwendbar ist. Bei der Expansion einer Kommunikationseinheit eines Prozesses wird der schon expandierte Übergangsgraph der verlangten Aktion in den Übergangsgraphen des aufrufenden Prozesses nach der entsprechenden Sa_k eingegliedert (Bild 8.1.1). Dabei bekommen die Elemente id, Suc und Pred aller Übergangseinheiten der Aktion feste Werte zugefügt.

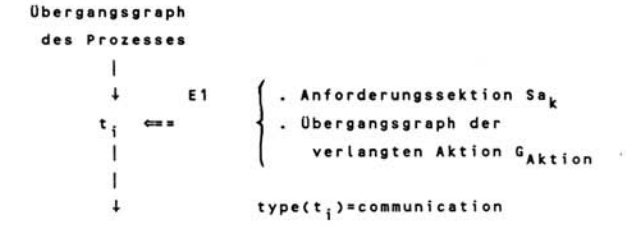


Bild 8.1.1 - Expansion einer Kommunikationseinheit

Jeder verkörperte Prozeß, der eine bestimmte Aktion verlangt, bekommt dann eine durch E1 in seinen Übergangsgraphen eingegliederte Kopie des Übergangsgraphen der verlangten Aktion. Damit wird bestimmt, daß die Realisierung eines IPM's keine eigene Steuerung benötigt, um Aktionen auszuführen. Die Ausführung einer Aktion ist lediglich eine Aufgabe des Prozesses, der diese Aktion benötigt. Deswegen kann eine Aktion mehrfach gleichzeitig aktiv sein, indem mehrere Prozesse Kopien dieser Aktion ausführen.

Um die Realisierung eines verkörperten IPM's zu vervollständigen, wird ein Schutzmechanismus (vgl. Abschnitt 3.2.2) erzeugt. Der Schutzmechanismus vermeidet Widersprüche in der Nutzung der gemeinsamen Mittel durch die zweckmäßige Verteilung des Zugriffsrechts an die Prozesse, welche Aktionen dieses IPM's durch den entsprechenden Anforderungssektionen verlangen (Bild 8.1.2)

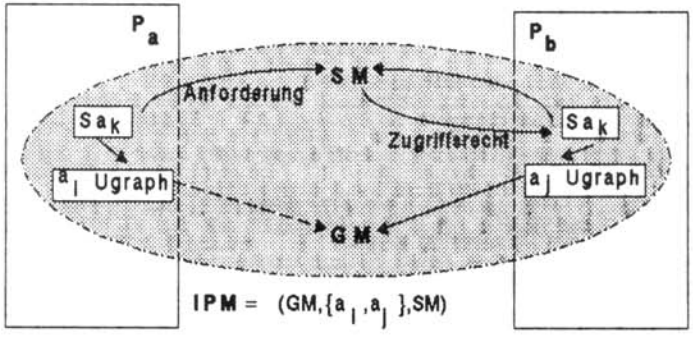


Bild 8.1.2 - Interaktion zwischen den Prozessen P_a und P_b,
a_i und a_j sind Aktionen des IPM's

Das Prioritätsschema und das Konfliktmodell jedes IPM's bestimmen die Architektur des Schutzmechanismus dieses IPM's. Die folgenden Abschnitte behandeln das Problem der Synthese des Schutzmechanismus und der mitwirkenden Anforderungs- und Warte-Sektionen.

8.2 *Das Konfliktmodell*

Das Konfliktmodell eines verkörperten IPM's weist auf die Paare der konfliktfreien Aktionen hin und wird durch die Analyse der Verhaltensgraphen der Aktionen automatisch abgeleitet. Konfliktfreie Aktionen sind solche, die keine Widersprüche verursachen können, wenn sie gleichzeitig aktiv sind. Die Ursachen der Widersprüche sind:

- Das gleichzeitige Lesen und Ändern eines GM's
- Das gleichzeitige Ändern eines GM's
- Eingeflochtene Operationen verschiedener Prozesse

Das Konfliktmodell bezieht sich auf Paare von Aktionen, obwohl bei einer Anzahl n von verkörperten Prozessen bis zu n Aktionen gleichzeitig verlangt werden können. Die Relation zwischen n Aktionen kann trotzdem auf eine Menge von 2-stelligen Relationen reduziert werden: belegt ein Prozeß schon das Mittel und aktiviert die Aktion a_i, wird der nächste Prozeß, der a_j verlangt, das Mittel nur mitbelegen dürfen, wenn das Paar (a_i,a_j) konfliktfrei ist. Kommt noch ein dritter Prozeß dazu, der a_h verlangt, bekommt er sofort das Zugriffsrecht, wenn die Paare (a_i,a_h) und (a_j,a_h) konfliktfrei sind, sonst muß er warten bis die anderen beiden Prozesse das Mittel freigeben.

8.2.1 *Umfang des Konfliktmodells*

Es seien ipm_a ein von Benutzer beschriebener IPM-Typ und $AGMs=\{a_1,\ldots,a_n\}$ die Menge der Aktionen von ipm_a. Sei ferner MpA die Menge der Paare der Aktionen von AGMs x AGMs, wobei (a_i,a_j) und (a_j,a_i) das gleiche Paar darstellt und nur einmal in MpA vorkommt. Sei letztlich a_i .ktf. a_j die Bezeichnung für ein konfliktfreies Paar $(a_i,a_j) \in MpA$, so ist die Menge der konfliktfreien Paare von Aktionen KfA, $KfA \subset MpA$, wie folgt definiert:

$$KfA = \{(a_i,a_j) \in MpA \mid a_i \ .ktf.\ a_j \}$$

Bezüglich KfA sind folgende Fälle zu berücksichtigen:

KF1: KfA = MpA (*alle Paare sind konfliktfrei*)

> Falls KfA = MpA ist, dann gilt, daß alle Aktionen von ipm_a gleichzeitig aktiv sein können, ohne Widersprüche zu verursachen. In diesem Fall ist der IPM-Typ konfliktfrei und alle seine Verkörperungen sind auch konfliktfrei. Für diese verkörperten Interprozeßmoduln wird kein Schutzmechanismus synthetisiert. Damit können die Prozesse Aktionen dieser Interprozeßmoduln ohne Abwarten des Zugriffsrecht direkt aktivieren.

KF2: KfA = { } (*kein Paar ist konfliktfrei*)

> In diesem Fall gilt, daß kein Paar von Aktionen gleichzeitig aktiv sein kann, ohne einen Konflikt zu verursachen. Die verkörperten Interprozeßmoduln von diesem Typ sind dann nur mit gegenseitigem Ausschluß zwischen alle Aktionen zu belegen.

KF3: KfA $\neq$ ( ) $\wedge$ KfA $\neq$ MpA (*einige Paare sind konfliktfrei*)

Erst durch die Analyse jedes verkörperten IPM's wird bestimmt, um welchen Konfliktfall es sich handelt.

Um den oben erwähnten Fall KF3 abzudecken, benötigt man für jeden verkörperten IPM die Menge MsA der Paare von Aktionen, die simultan verlangt werden können. Sei $Ps=\{P_1,\ldots,P_n\}$ eine Menge von verkörperten Prozessen $Ps=\{P_1,\ldots,P_n\}$, die mit einem verkörperten IPM verbunden sind, so gilt für die Menge MsA:

$$\forall\ (a_i,a_j) \in MsA : \exists\ P_i \in Ps\ .\ (P_i \text{ kann } a_i \text{ verlangen}) \wedge \exists\ P_j \in Ps,\ P_j \neq P_i\ .\ (P_j \text{ kann } a_j \text{ verlangen})$$

Beispiel 8.2.1:

Sei $AGMs=\{a_1,a_2,a_3\}$ die Menge der Aktionen eines IPM-Typs ipm_b und seien P_1 und P_2 die Prozesse, die durch eine Verkörperung Ipm_c von ipm_b miteinander interagieren. Sei angenommen, daß P_1 alle Aktionen von Ipm_c verlangen kann, und daß P_2 nur a_3 benötigt. Damit ist MsA:

$MsA = \{(a_1,a_3),(a_2,a_3),(a_3,a_3)\}$

In Ipm_c werden a_1 und a_2 nie gleichzeitig verlangt, weil ein Prozeß jedesmal nur eine Aktion ausführen kann. Das gilt auch für (a_1,a_1) und (a_2,a_2).

Ende des Beispiels.

Mit Hilfe von MsA läßt sich der Konfliktfall eines verkörperten IPM's deutlich charakterisieren.

KF3.1: MsA $\bigcap$ KfA = MsA

Der IPM ist konfliktfrei. Es entspricht die Verhaltensweise dem Fall KF1.

KF3.2: MsA $\bigcap$ KfA = ( )

Es gibt kein Paar von konfliktfreien Aktionen, die

gleichzeitig verlangt werden können. Es entpricht dem Fall KF2.

KF3.3: $MsA \bigcap KfA \neq \{\ \}$ und $MsA \bigcap KfA \neq MsA$

Es gibt mindestens ein Paar von konfliktfreien Aktionen, die gleichzeitig aktiv sein können.

Für einen bestimmnten verkörperten IPM sind letztendlich drei Fälle zu unterscheiden:

CC1 (*conflict case 1*): KF1 oder KF3.1
CC2 (*conflict case 2*): KF2 oder KF3.2
CC3 (*conflict case 3*): KF3.3

Damit sind alle Attribute definiert, die das Konfliktmodell eines verkörperten IPM's kennzeichnen.

DEF. 8.2.1 - Konfliktmodell

Das Konfliktmodell K_{modell} ist eine Tupel:

$K_{modell} = (CC, MsA, MsA \bigcap KfA)$

mit den Komponenten:
- Konfliktfall (CC)
- Menge der potentiellen simultanen Aktionen (MsA)
- Menge der simultanenen konfliktfreien Aktionen ($MsA \bigcap KfA$)

Beispiel 8.2.2:

Sei $KfA=\{(a_1,a_2),(a_1,a_3),(a_1,a_1)\}$ die Menge der konfliktfreien Aktionen des IPM-Typs des Beispiels 8.2.1. Das Konfliktmodell für Ipm_c ist:

$K_{modell} = (CC3, \{(a_1,a_3),(a_2,a_3),(a_3,a_3)\}, \{(a_1,a_3)\})$

Der Schutzmechanismus dieses IPM's muß erlauben, daß (a_1,a_3)

gleichzeitig aktiv sein können und muß vermeiden, daß (a_2,a_3) und (a_3,a_3) gleichzeitig aktiviert werden.

Ende des Beispiels.

8.2.2 *Konfliktanalyse*

Die Analyse des Konflikts stützt sich auf die Annahme, daß das gleichzeitige Lesen eines Informationsträgers (eines Registers zum Beispiel) keinen Zugriffskonflikt verursacht. Voraussetzung dafür ist, daß die Realisierung dieser Informationsträger ein geeignetes fan-out anbietet. Diese Annahme gilt nicht für adressierbaren Speicher. Dabei wird der Inhalt des Adressregisters geändert, wodurch das gleichzeitige Lesen eines Speichers nicht konfliktfrei sein kann.

Die Analyse eines Paars von Aktionen benötigt für jede Aktion a_i die folgenden Mengen:

- *DEST(a_i):* Zieloperanden der Aktion ohne die Parameter. Zieloperanden der Aktion sind alle Zieloperanden der Knoten (Operationen) dieser Aktion.

- *SOURCE(a_i):* Quelloperanden der Aktion ohne die Parameter, aber mit den Quelloperanden der in dieser Aktion benutzten Warte-Bedingungen.

Der Zieloperand eines Knotens ist der Informationsträger, der das Ergebnis der Operation dieses Knoten bekommt und damit geändert wird. Die Quelloperanden werden dagegen nur gelesen, d.h. sie werden nicht geändert (solange sie nicht ein Zieloperand irgendeiner Operation sind). Zwei Aktionen a_i, a_j sind dann konfliktfrei, falls die Informationsträger, die eine Aktion liest (SOURCE(a_i) oder SOURCE(a_j)) oder ändert (DEST(a_i) oder DEST(a_j)) von der anderen Aktion nicht gleichzeitig geändert werden können.

Ob zwei Aktionen a_i und a_j konfliktfrei sind, kann man durch das folgende Prädikat feststellen:

$$P_{ktf}(a_i,a_j) = (SOURCE(a_i) \cap DEST(a_j) = \{\}) \wedge (SOURCE(a_j) \cap DEST(a_i) = \{\}) \wedge (DEST(a_i) \cap DEST(a_j) = \{\})$$

$P_{ktf}(a_i,a_j)$ = TRUE ist eine genügende Bedingung für die Berechnung von $a_i \overset{ktf}{\underline{ktf}} a_j$. Wie später gezeigt wird, ist es aber keine notwendige

Bedingung. Um die Menge KfA zu bestimmen, muß man alle Paare von MpA nach diesem Pradikät überprüfen. Die Anzahl der Paare, welche die Komplexität des Aufbaus von KfA bestimmt, ist:

$$|MpA| = n \cdot (n + 1) / 2$$

Das folgende Beispiel zeigt die Überprüfung des Prädikats P_{ktf} für den IPM des Leser-Schreiberproblems (Abschnitt 2.1.5 und Anhang A1).

Beispiel 8.2.3:

AGMs = {read,write}
MpA = {(read,read),(read,write),(write,write)}

Die Mengen *SOURCE* und *DEST*, vom BABEL-Übersetzer berechnet, sind die folgenden:

SOURCE(read) = {shared_data}, DEST(read) = { },
DEST(write) = {shared_data}, SOURCE(write) = { }

Damit bestimmt man:
P_{ktf}(read,read) = TRUE, P_{ktf}(read,write) = FALSE
P_{ktf}(write,write) = FALSE

Für das Paar (read,read) gilt dann: read .ktf. read.

Ende des Beispiels.

Falls $P_{ktf}(a_i,a_j)$ = FALSE ist, können a_i und a_j noch konfliktfrei sein. Das geschieht beispielweise, wenn gegenseitiger Ausschluß zwischen zwei Aktionen durch die Nutzung des bedingten Wartens schon vorhanden ist. Voraussetzung dafür ist, daß immer eine der das Prädikat P_{ktf} nicht erfüllenden Aktionen (a_i,a_j), den entsprechenden Prozeß in den Warte-Zustand führt und erst wieder aktiv wird, wenn die andere Aktion den IPM schon verlassen hat. Diese Bedingung läßt sich wie folgt formulieren:

Wenn $P_{ktf}(a_i,a_j)$=FALSE, gilt noch a_i.ktf.a_j, falls die folgenden Bedingungen erfüllt werden können:

B1. Jede Aktion enthält einen Warte-Abschnitt.

B2. Die Warte-Bedingungen dieser Aktionen sind komplementär: ($condition_i$ = NOT $condition_j$). Damit wird garantiert, daß immer eine der beiden Aktionen den entsprechenden Prozeß in den Warte-Zustand führt.

B3. Die Warte-Bedingungen dürfen bis kurz vor Verlassen des IPM's nicht geändert werden. Damit wird garantiert, daß der wartende Prozeß nicht zu früh den Warte-Zustand verläßt.

$$SOURCE(condition_i) \not\subset (DEST(a_i) - DEST(PRED(LAST(a_i)))\ )$$
$$SOURCE(condition_j) \not\subset (DEST(a_j) - DEST(PRED(LAST(a_j)))\ )$$

B4. Wenn es einen Abschnitt (*head*) vor dem Warte-Abschnitt gibt, dann muß für beide Aktionen gelten:

$$(SOURCE(head(a_i)) \cap DEST(a_j) = \{\ \}) \wedge$$
$$(DEST(head(a_i)) \cap SOURCE(a_j) = \{\ \}) \wedge$$
$$(DEST(head(a_i)) \cap DEST(a_j) = \{\ \})$$

Der Abschnitt *head* und jeder der Abschnitte der anderen Aktion dürfen keine Konflikte verursachen. Um die Ausführung des Warte-Abschnitts zu garantieren, muß auch gelten:

$$SUC(LAST(head(a_i)) = FIRST(Warte\text{-}Abschnitt)$$

Die folgenden Beispiele zeigen die Anwendung der neuen Bedingungen zur Bestimmung von konfliktfreien Aktionen.

Beispiel 8.2.4:

Bei der Überprüfung der oben festgelegten Bedingungen auf den Paaren (read,write) und (write,write) des Beipiels 8.2.3 bestimmt man, indem weder *read* noch *write* Warte-Abschnitte enthalten (siehe Anhang A1), daß: KfA = {(read,read)}.

Ende des Beispiels.

Das folgende Beispiel enthält die Beschreibung des IPM-Typs Puffer des zweiten Beispiels (Abschnitt 2.1.1 und Bild 2.1.2).

Beispiel 8.2.5:

Das GM Puffer speichert nur ein Datenelement. Solange dieses Element nicht geholt wird, darf auf Puffer nicht geschrieben werden. Jedes geschriebene Element darf nur einmal geholt werden.

```
INTERPROCESS_MODULE Puffer;
     VAR buffer : LOGICAL(7..0);
         written : LOGICAL INITIAL = 0;
         full, empty: CONDITION;

     FUNCTIONAL  full  := (written = 1),
                 empty := (written = 0)    END;

     ACTION  put;
         INTERFACE  data(7..0): INPUT;
         SEQUENCE   IF full THEN  WAIT  FI;
                    buffer := data;  written := 1  END
     END ACTION;

     ACTION  get;
         INTERFACE  element(7..0): OUTPUT;
         SEQUENCE   IF empty THEN  WAIT  FI;
                    element := buffer; written := 0  END
     END ACTION;
 END INTERPROCESS.
```

SOURCE(get) = {buffer,written}, DEST(get) = {written}
SOURCE(put) = {written}, DEST(get) = {buffer,written}

Damit bestimmt man:

$$P_{ktf}(get,get) = P_{ktf}(get,put) = P_{ktf}(put,put) = FALSE$$

Warte-Abschnitte sind aber in beiden Aktionen vorhanden (B1). Bei der Überprüfung der vereinbarten Warte-Bedingungen stellt man fest, daß für die Warte-Bedingungen von *get* und *put* gilt: full = NOT empty (B2). B2 ist nicht von (get,get) und (put,put) erfüllt. Für das Paar (get,put) gilt aber auch noch, daß die Änderung der Bedingung (indirekt durch den Quelloperand) erst bei der letzten Operation vor dem Verlassen des IPM's stattfindet (B3). Damit wird bestimmt, daß: KfA = {(get,put)}

Ende des Beispiels.

Andere Fälle, in den $P^{ktf}(a_i,a_j)$=FALSE ist und die Aktionen trotzdem konfliktfrei sind, sind auch denkbar. Diese Fälle sind aber im allgemeinen ohne Kenntnis der Realisierung bei einer statischen Analyse auf der Verhaltensebene nicht feststellbar. Die Vorkenntnis der Realisierung verstößt gegen die Strategie des hier verfolgten Syntheseprozesses, steigt die Komplexität der Konfliktanalyse und bringt vermutlich kaum einen Gewinn in der Nebenläufigkeit des Systems. Diese lezte Annahme muß später noch nachgewiesen werden.

8.3 *Das Prioritätsschema*

Das Prioritätsschema eines Systems bezüglich eines verkörperten IPM's enthält die Spezifikation der Prioritäten der Aktionen und Prozesse (vgl. Abschnitt 3.4.2) und zusätzlich dazu einen Hinweis auf den Prioritätsfall. Dieser letzte ergibt sich aus der Zusammensetzung beider Prioritätsspezifikationen.

Eine Prioritätspezifikation *Pri.spec* ist eine Halbordnung, $(\Pi, \geq)$. Je nach vorhandenen Relationen zwischen den Elementen von Π (entweder Aktionen oder Prozesse) lassen sich drei Arten von Prioritäten unterscheiden:

```
gleich     - Relation: =          (Beispiel: A = B = C)
gemischt   - Relation: = und >    (        : A > B = C)
geordnet   - Relation: >          (        : A > B > C)
```

Damit sind die folgenden Prioritätsfälle (Tabelle 8.1) zu berücksichtigen:

Priorität der Aktionen	Priorität der Prozesse: gleich	gemischt	geordnet	Prioritäts-hierarchie
gleich	PC1	PC2	PC3	einstufig
gemischt	PC4	PC5	PC6	zweistufig
geordnet	PC7	PC8	PC9	zweistufig

Tabelle 8.1 - Prioritätsfälle

Für die Fälle PC1, PC2, PC3 spielt die Priorität der Aktionen keine Rolle. Die verlangten Aktionen brauchen nicht identifiziert zu werden,

was die Realisierung des Schutzmechanismus beträchtlich vereinfacht. Für alle andere Fälle gilt eine zweistufige Hierarchie: die Aktionspriorität hat Vorrang. Wenn diese nicht entscheiden kann weil z.B. zwei Prozesse die gleiche Aktion verlangen, so gilt die Prioritätseinordnung der Prozesse. Bei der Realisierung ist auch zu unterscheiden, daß PC1 eine reine zyklische Priorität darstellt; PC3, PC6 und PC9 stellen dagegen reine feste Priorität dar. Die andere Fälle sind komplexer zu realisieren, da sie eine Mischung von zyklischer und fester Priorität umfassen.

8.4 *Synthese des Interaktionsmechanismus*

Die Synthesestrategie beschränkt sich auf eine Fallunterscheidung und eine Anpassung von vorher realisierten Musterlösungen an jeden dieser Fälle. Eine Musterlösung besteht aus einem für einen bestimmten Fall generierten Schutzmechanismus und den entsprechenden Anforderungs- und Wartesektionen.

Das Konfliktmodell und das Prioritätschema bestimmen die Anzahl und die Art der Fälle, die zur Synthese eines geeigneten und wenig aufwendigen (wenn nicht sogar optimalen) Interaktionsmechanismus berücksichtigen werden müssen. Die zu unterscheidenden Fälle, durch das Produkt der Anzahl der Konfliktfälle (CC) und der Prioritätsfälle (PC) bestimmt, zählen 27. Neun davon sind trivial: falls das Konfliktmodell auf den Fall CC1 hinweist (alle Aktionen sind konfliktfrei), dann fallen der Schutzmechanismus und die Anforderungssektion aus. Das Prioritätschema verliert dabei seine Bedeutung. Zum Interaktionsmuster der trivialen Fälle gehören ausschließlich Warte-Sektionen, und zwar nur dann, wenn Warte-Abschnitte in den entsprechenden Aktionen überhaupt vorhanden sind.

Es bleiben noch 18 nichttriviale Fälle übrig. Für jeden dieser Fälle gibt es mehrere mögliche Lösungen. Einige davon werden in Anhang A4 vorgeschlagen. Diese wurden unter Berücksichtigung der folgenden Punkte realisiert:

- kleine Anzahl von Komponenten
- kleine Anzahl von Zuständen in der Anforderungs- und Wartesektion
- synchroner wie auch asynchroner Betrieb möglich
- allgemeine Lösung (leicht für verschiedene Anzahl von Prozessen anzuwenden)

Die Idee hinter diesen Vorschlägen ist die Sammlung einer Bank von Lösungen, von denen je nach Systemmerkmalen, Randbedingungen und erworbener Erfahrung die Beste angewendet werden kann.

8.5 *Bewertung des Konzepts*

Entsprechend dem Synthesekonzept verfügt die Realisierung jedes terminalen Prozesses über ein Steuerwerk. Die Steuerwerke eines Systems sind voneinander unabhängig und mit Ausnahme der Wartezustände (wait for grant, wait for condition and/or grant) prinzipiell immer aktiv. Daher sind diese Steuerwerke dann auch diejenigen im realisierten System, die bei der Berechnung des Nebenläufigkeitsgrades Φ dieses Systems berücksichtigt werden müssen (siehe 5.3). In der Berechnung von Φ sind die Steuerwerke der Moduln (siehe Kapitel 7) schon indirekt durch deren Auswirkung auf Φ_c berücksichtigt.

Das Synthesekonzept erlaubt auf der Prozeßebene die Realisierung der Nebenläufigkeit, die schon in der Beschreibung des Systems vorhanden ist. Bei der Expansion der Anforderungs- und Warteknoten und der Eingliederung des Schutzmechanismus in das Operationswerk sollte allerdings die Komplexität des zu realisierenden Systems bzw. die Anzahl der Komponenten und deren Verbindungen nicht beträchtlich erhöht werden. Durch die Konfliktanalyse und die sorgfältige Auswahl und Anpassung eines zu jedem Fall geeigneten Interaktionsmusters wird diese Komplexität gering gehaltet. Es bleibt trotzdem eine Frage der Größe und des Ziels des Systems, ob es sich lohnt, Nebenläufigkeit (d.h. Geschwindigkeit) auf Kosten der Erhöhung der Anzahl der Komponenten (d.h Chipfläche) zu gewinnen.

Die Komplexität des Syntheseverfahrens selbst ist hier geringer als die des automatischen Suchens nach potentiellen nebenläufigen Einheiten in einer nicht strukturierten Beschreibung. Das Konzept ist auch vorteilhafter als die in DSL vorhandene Möglichkeit, mehrere Sequenzen zu beschreiben und die globalen Variablen als gemeinsame Mittel zu verwenden (und der daraus folgende Bedarf an Schutz aller globalen Variablen gegen Zugriffskonflikte). Die geringere Synthesekomplexität liegt bei BABEL daran, daß das Suchen nach Konflikten auf die Interprozeßmoduln begrenzt wurde und geeignete Interaktionsmechanismen schon vorhanden sind.

9. Abschließende Betrachtungen

9.1 *Ausblick*

Diese Arbeit stellt ein Konzept zur besseren Ausnutzung der in Hardware verfügbaren Nebenläufigkeit bei der automatischen Synthese digitaler Systeme vor. Ein wichtiger Punkt ist dabei die Annahme, daß ein höherer Grad an Nebenläufigkeit ohne wesentliche Zunahme der Synthesekomplexität erreicht wird, wenn Nebenläufigkeit schon in der frühesten Phase des Entwurfszyklus, nämlich bei der formalen Beschreibung des Systems, berücksichtigt wird.

Um dieses Konzept zu verwirklichen, wurde eine Entwurfssprache entwickelt, welche die Beschreibung des Verhaltens von nebenläufigen Prozessen und deren Interaktionen auf eine voll strukturierte und hierarchische Weise erlaubt. Durch automatische Synthese wird diese Beschreibung in digitale Strukturen umgesetzt. Dabei ermöglicht die Realisierung mehrerer Steuerwerke eine effektive Nutzung der Nebenläufigkeit zwischen den synthetisierten Prozessen. Innerhalb eines dieser Prozesse erreicht die Kompaktierung von Zuständen eine höhere Parallelität zwischen den Operationen, was in signifikanter Weise zur Erhöhung des Nebenläufigkeitsgrads des realisierten Systems beiträgt.

Auf der Verhaltensebene erfolgt die Interaktion zwischen Prozessen durch die Anwendung von Interprozeßmoduln, einem abstraken Datentyp, dessen Konzept aus dem Kessels-Monitor abgeleitet wurde. Damit die Realisierung dieser Interaktionen die Komplexität des realisierten Systems (d.h. die Anzahl und die Größe der Komponenten und deren Verbindungen) nicht beeinträchtigt, wurden statt eines allgemeinen Interaktionsmechanismus mehrere, weniger aufwendige Strukturen für diesen Mechanismus vorgeschlagen. Gemäß der Merkmale jedes beschriebenen Systems kann eine dieser Strukturen ausgewählt and angepaßt werden.

Von existierenden Beschreibungsmitteln und Syntheseverfahren unterscheidet sich das hier entwickelte Werkzeug dadurch, daß es mächtige Sprachmittel zur Formulierung von Nebenläufigkeit umfaßt und die Erhöhung des Nebenläufigkeitgrads auf drei Schichten (Schicht der Prozesse, der Funktionen und der Operationen) während des ganzen Entwurfszyklus verfolgt.

9.2 *Ansätze zu weiteren Untersuchungen*

Weitere Untersuchungen könnten sich auf folgende Punkte beziehen:

- Anwendung anderer Synchronisationsmodelle und Vergleich der Ergebnisse. Kriterien für diesen Vergleich könnten bei der Verhaltensbescheibung die Einfachheit der Anwendung (z.B. Übersichtlichkeit, Abdeckung von Hardwareeigenschaften, geringere Fehleranfälligkeit und minimale Einschränkungen usw.) sowie die vorhandenen Möglichkeiten zur Überprüfung der Korrektkeit sein. Beim automatischen Entwurf könnten die Vergleichskriterien der geringste Übersetzer- und Syntheseaufwand oder die geringste Komplexität der realisierten Schaltungen sein.

- Anschluß an ein Simulationssystem, welches die Simulation der in dieser Arbeit formulierten Nebenläufigkeit gestattet.

- Erweiterung des Verfahrens oder Entwurf neuer Verfahren zum Verhaltenstest mehrerer nebenläufigen Prozesse und deren Interaktion [Webe86].

Aus der inneren Sicht der Problemstellung würden sich folgende Weiterführungen anbieten:

- Erweiterung des Konfliktsmodells, um einen größeren Bereich von Verträglichkeit zwischen Aktionen abzudecken.

- Verfeinerung des Verfahrens zur Auswahl und Anpassung der Interaktionsstrukturen. Damit wäre es nötig, jeden der 18 Fälle (Abschnitt 8.4) weiter zu unterteilen (beispielweise bezüglich des Verhältnisses zwischen der Anzahl der Prozesse und der Anzahl der verlangten Aktionen oder anderer noch zu entwickelder Kriterien). Für jeden der unterteilten Fälle könnten eventuell noch effizientere Strukturen entworfen werden.

Literaturverzeichnis

AlPe81 Allen,J.; Penfield,P.
VLSI design automation activities at MIT. IEEE Trans. on Circuits and Systems CAS-28,7 (July 1981), 645-653

AnSc83 Andrews,G.R.; Schneider,F.B.
Concepts and notations for concurrent programming. Computing Surveys 15,1 (March 1983), 3-43

Barb81 Barbacci, M.R.
Instruction set processor specifications (ISPS): the notation and its applications. IEEE Trans. on Computers C-30,1 (January 1981), 24-40

Barb85 Barbacci, M.R. et alli
Ada as a hardware description language: an initial report. CHDL85 - Computer Hardware Description Lan- guages and their Applications, Tokio, IFIP85, (August 1985), 272-302

BenA82 Ben-Ari,M.
Principles on Concurrent Programming. Prentice-Hall, Englewood Cliffs, N.J., 1982

Boot67 Booth, Taylor L.
Sequential Machines and Automata Theory. John Wiley and Sons, Inc., New York (1967)

Brin75 Brinch Hansen, P.
The programming language Concurrent Pascal. IEEE Trans. Softw Eng. SE-1, 2 (June 1975), 199-206

CaTr84 Camposano,R.; Treff,L.
STRUDEL - Eine Sprache zur Spezifikation der Struktur digitaler Schaltungen. Interner Bericht Nr. 7/84, Universität Karlsruhe, Institut für Informatik IV, 1984

CaWe84 Camposano, R.; Weber, R.
DSL - Eine Sprache zur Spezifikation digitaler Schaltungen. Interner Bericht Nr. 24/84, Universität Karlsruhe, Institut für Informatik IV, 1984

CaWe85a Camposano, R.; Weber, R.
Semantik und interne Form von DSL. Interner Bericht 3/85, Fakultät für Informatik, Universität Karlsruhe, 1985

CaWe85b Camposano, R.; Weber, R.
Compilation and internal representation of digital systems specification in DSL. XI Conferencia Latinoamericana de Informatica Porto Alegre, Brasil, 1985

ClOf84 Cleemput,W.M. van; Ofek,H.
Design automation for digital systems. Computer (October 1984), 114-122

Comp74 Computer
Special Issue on CHDL's. Computer, vol.7, no.12 (December 1974)

Comp77 Computer
HDL Applications. Computer, vol.10, no.6 (June 1977)

Comp85 Computer
Hardware Description Languages. Computer, vol.18, no. 2 (February 1985)

CKR84 Camposano, R,; Kunzmann, A.; Rosenstiel, W.
Automatic data path synthesis from behavioural level descriptions in DSL. VLSI: Algorithms and Architectures, Edited by P. Bertolazzi, F. Lucio, North Holland, 1985

Dire81 Director,S.W.; Parker,A.C.; Siewiorek,D.P.; Thomas,D.E.
A design methodology and computer aids for digital VLSI systems. IEEE Trans. on Circuits & Systems CAS-28,7 (July 1981), 634-644

DAC85 Design Automation Conference
22th Design Automation Conference, Las Vegas, (1985)

DSST83 Director,S.W,; Shen,J.P.; Siewiorek,D.P.; Thomas,D.E.
The CMU-DA/CAD project. DEE, Carnegie-Mellon University Pittsburgh, PA 15213 (1983), 1-46

Ebe81 Ebert, J.
Effiziente Graphenalgorithmen. Akademische Verlagsgeselschaft, Wiesbaden, 1981

Feue83 Feuer,M.F.
VLSI design automation: an introdution. Proc. of the IEEE 71,1 (January 1983), 5-9

FlUl82 Floyd, R.W.; Ullman, J.D.
The compilation of regular expressions into integrated circuits. Journal of the ACM, Vol.29, No.3 (July 1982), 603-622

FoKu80 Foster,M.J.; Kung,H. T
The design of a special purpose VLSI chips. Computer (January 1980), 26-40

HaPa81 Hafer,L.; Parker,A.C.
A formal method for the specification, analysis, and design of register-transfer level digital logic. 18th Design Automation Conference, IEEE, (1981), 846-853

HaPa82 Hafer,L.H.; Parker,A.C.
Automated synthesis of digital hardware. IEEE Trans. on Comp. C-31,2 (February 1982), 93-109

HLSM82 Haynes,L.S.; Lau,R.L.; Siewiorek,D.P.; Mizell,D.W.
A survey of highly parallel computing. Computer (January 1982), 9-24

Kast79 Kastens, Uwe.
ALADIN - Eine Definitionssprache für attributierte Grammatiken. Interner Bericht Nr. 7/79, Universität Karlsruhe, Fakultät für Informatik, 1979

Kess77 Kessels, J.L.W.
An alternative to event queues for synchronization in monitors. Commun. ACM 20,7 (July 1977), 500-504

Kung80 Kung,H.T.
The structure of parallel algorithms. Advances in Computer, 19 (1980), 65-112

Kung82 Kung,H.T.
Lecture notes for advanced course on VLSI architecture. Advanced course on VLSI architecture, Bristol (July 1982)

KHZ82 Kastens,U.; Hutt,B.; Zimmermann,E.
GAG: A Pratical Compiler Generator. Lecture Notes in Computer Science 141, Springer Verlag, 1982

Lipp83 Lipp,H.M.
Methodical aspects of logic synthesis. Proc. of the IEEE 71,1 (January 1983), 88-97

MiKo84 Miklosko,J.M.; Kotov,V.E. (Editors)
Algorithms, Software and Hardware of Parallel Computers. Springer-Verlag, Berlin, 1984

Mold83 Moldovan,D.I.
On the design of algorithms for VLSI systolic array. Proceedings of the IEEE 71,1 (January 1983), 113-120

Newt81 Newton,A.R.; Pederson,D.O.; Vincentelli,A.L.; Sequin,C.H.
Design aids for VLSI: the Berkeley perspective. IEEE Trans. on Circuits & Systems CAS-28,7 (July 1981), 666-680

Noll86 Nolle,M.
Eine Beschreibungssprache zum strukturierten Entwurf von komplexen digitalen Steuerwerken mit Nachweis der Lebendigkeit. Dissertation, VDI Reihe 10 Nr.55, Düsseldorf: VDI Verlag, 1986

Nies83 Niessen,C.
Hierarchical design methodologies and tools for VLSI chips. Proc. of the IEEE 71,1 (January 1983), 66-75

PaWa81 Parker,A.C; Wallace,J.J.
SLIDE: an I/O harware descriptive language. IEEE Trans. on Computers C-30,6 (June 1981), 423-439

Pete77 Peterson, J.L.
Petri Nets. Computing Surveys, Vol. 9/3 (September 1977), 223-225

Pete81 Peterson,J.G.
Keys to successful VLSI system design. VLSI - Systems and Computations (1981), 21-28

Ramm82 Rammig, F.
CAP/DSDL - Anwenderbeschreibung, Hardwarebeschreibung CAP/DSDL Dok-nr 44.2.1.1-6, Dortmund (Juli 1982), 1-142 1-142

Rem81 Rem, M.
The VLSI challenge: complexity bridling. VLSI 81: Very Large Scale Integration, Academic Press, London (1981)

Rem84 Rem, M.
Concurrent computations and VLSI circuits. Working material, International Summer School Control Flow and Data Flow: Concepts of Distributed Programming (August 1984)

Rich84 Rich,E.
The gradual expansion of artificial intelligence. Computer (May 1984),4-12

Rose84 Rosenstiel, W.
Synthese des Datenflusses digitaler Schaltungen aus formalen Funktionsbeschreibungen. Dissertation, Fakultät für Informatik, Universität Karlsruhe, VDI-Verlag, (1984)

RoCa85 Rosenstiel,W.; Camposano,R.
Synthesizing circuits from behavioural level specifications. CHDL85 - Computer Hardware Description Lan- guages and their Applications, Tokio, IFIP85, (August85), 391-403

Sequ83 Sequin,C.H.
Managing VLSI complexity: an outlook. Proc. of the IEEE 71,1 (January 83), 149-166

Shiv83 Shiva,S.G.
Automatic hardware synthesis. Proc. of the IEEE 71,1 (January 1983), 76-87

SiCo83 Singh,J.; Collins,R.M.
An environment for simulating concurrent systems. ICCAD83: International Conference on Computer Aided Design, Santa Clara, California, (September 1983), 215-216.

SiTr81 Singh,A.K.; Tracey,J.H.
Development of comparision features for computer hardware description languages. Computer Hardware Description Languages and their Applications: Breuer, Hartenstein (eds) North-Holland Publishing Company,IFIP (1981)

Snep83 Snepscheut,J.L.A. van de
Deriving Circuits from Programs. Third CALTECH Conference on Very Large Scale Integration, Springer-Verlag (1983) 241-256

Suzu85 Suzuki,N.
Concurrent Prolog as an Efficient VLSI Design Language. Computer 18,2 (February 1985), 33-40

SCR84 Schmid, D.; Camposano, R.; Rosenstiel, W.
Automatischer Entwurf hochintegrierter. Schaltungen aus Beschreibungen der Schaltungsfunktion. Informatik Fachberichte 88 (Hrsg.: W.Brauer), Springer 1984

SGB83 Siewiorek,D.P.; Giuse,D.; Birmingham,W.P.
Proposal for research on DEMETER: a design methodology and environment. Interner Bericht, CMU, Pittsburgh, (Feb 1983)

Thur72 Thurber,K.J. et al.
A systematic approach to the design of digital bussing structures. Fall Joint Computer Conference (1972), 719-740

Thur82 Thurn,K.
Ein Beschreibungsverfahren zum Entwurf digitaler Steuerungen für nebenläufige Vorgänge. Dissertation, Karlsruhe 1982

Trel82 Treleaven,P.C.
VLSI processor architectures. Computer (June 1982), 33-45

Trim81 Trimberger,S.; Rowson,J.A.; Lang,C.R.; Gray,J.P.
A structured design methodology and associated software tools. IEEE Trans. on Circuits and Systems CAS-28,7 (July 1981), 618-633

Ull76 Ullrich, G.
Der Entwurf von Steuerstrukturen für parallele Abläufe mit Hilfe von Petri-Netzen. Dissertation, Fakultät für Informatik, Universität Hamburg, 1976

USD81 U.S. Department of Defense
Programming Language Ada: Reference Manual. Lecture Notes in Computer Science, vol.106, Springer-Verlag, New-York, 1981

Webe85 Weber, T.S.
Strukturierte Beschreibung von Nebenläufigkeit in digitalen Schaltungen. Interner Bericht Nr. 8/85, Universität Karlsruhe, Institut für Informatik IV, 1985

Webe86 Weber, R.F.
Ein Testerzeugungsverfahren für digitale Schaltungen mittels einer Hardwarebeschreibungssprache. Dissertation, Universität Karlsruhe, Institut für Informatik IV, 1986

Wern82 Werner,J.
The silicon compiler: panacea, wishful thinking, or old hat? VLSI Design (Sept/Oct 1982), 46-52

Wirt77 Wirth, N.
Modula, a language for modular multiprogramming. Softw. Pract. Exp. 7 (1977), 3-35

Zakh84 Zakharov, Vasilli
Parallelism and array processing. IEEE Transactions on Computers, Vol C-33, no 1 (January 1984)

A N H A N G

Abschnitt A1 enthält Teile der Beschreibungen des Beispiels 6 (Abschnitt 2.1.4) und des Beispiels 7 (2.1.5). Abschnitt A2 enthält die vollständige Beschreibung des Beispiels 4 (Abschnitt 2.1.2), und Abschnitt A3 zeigt einen Teil der Übersetzerausgabe für dieses letzte Beispiel. Zum Schluß werden in A4 noch einige Lösungen für den Interaktionsmechanismus (Abschnitt 8.4) vorgeschlagen.

A1. *BABEL Beschreibung zweier Synchronisationsprobleme*

In diesem Abschnitt werden die BABEL-Lösungen für zwei häufige Synchronisationsprobleme vorgeführt.

Beim ersten handelt es sich um die Beschreibung eines Zusammentreffens mehrerer Prozesse (Beispiel 6). Da BABEL das Kooperationsmodell verwendet, ist dieses Problem durch ein gemeinsames Mittel zu lösen. Hier wurde ein Zähler (counter) als gemeinsames Mittel verwendet. Der Anfangswert des Zählers ist die Anzahl der Prozesse. Jeder Prozeß, der den Treffpunkt erreicht, zieht 1 von diesem Wert ab, und prüft, ob der Zähler den Wert null enthält. Wenn nein, geht der Prozeß in den Wartezustand; wenn ja, befinden sich schon alle anderen Prozesse im Wartezustand. Der laufende Prozeß lädt wieder den Anfangswert in den Zähler und kann den Treffpunkt verlassen. Gleichzeitig merken die wartenden Prozesse die Änderung des gemeinsamen Mittels und verlassen den Treffpunkt.

Der IPM 'rendez_vous' enthält das GM 'counter' und die Abzieh-Aktion 'synchronizer'. Wenn ein Prozeß seinen Treffpunktzustand erreicht, verlangt er die Ausführung dieser Aktion. Das IPM-Konzept bestimmt, daß, sobald der letzte Prozeß den 'counter' auf null bringt, alle wartenden Prozesse befreit werden, ohne zum IPM zurückzukehren (die Warteoperation ist die letzte Operation vor dem Endknoten der Aktion).

```
(*  EXAMPLE :  rendez_vous - synchronization of 7 processes *)

SYSTEM  example_6;
```

```
INTERPROCESS_MODULE   rendez_vous;
     VAR  counter :  LOGICAL(2..0) INITIAL=7;        (* GM *)
          notall  :  CONDITION;
     FUNCTIONAL  notall := (counter > 0) END;

     ACTION  synchronizer;
         SEQUENCE  counter := counter - 1;
                   IF notall THEN WAIT ELSE counter := 7 FI;
         END
     END ACTION;
END INTERPROCESS.
:
CREATE  rv : rendez_vous;
ACTIVATE  ... (Verkörperungen von proc_a, proc_b, proc_c,
               proc_n, proc_m, proc_p and proc_k)
ASSOCIATE  proc_a, proc_b, proc_c, proc_n,
           proc_m, proc_p, proc_k  WITH    rv;

END SYSTEM.
```

Das zweite Beispiel in diesem Abschnitt ist das bekannte Leser-Schreiber-Problem (2.1.5). Eine bestimmte Anzahl von Prozessen arbeitet mit gemeinsamen Daten. Die Prozesse sind von zwei Kategorien, die 'Leser' und 'Scheiber' genannt werden. Der Zugang zu den gemeinsamen Daten ist so zu regeln, daß entweder beliebig viele Leser oder genau ein einziger Schreiber diese Daten bearbeiten können. Abhängig von der Art des zu realisierenden Systems kann es nötig sein, den Lesern oder Schreibern Vorrang einzuräumen. Der folgende Auszug aus einem BABEL-Programm zeigt einen IPM, der dieses Synchronisationsproblem beschreibt. Dieser IPM enthält nur ein gemeinsames Mittel und zwei Aktionen im Gegensatz zu einer Lösung durch einen Monitor, die vier Prozeduren, zwei Bedingungen und zwei zusätzliche Zähler benötigt [Kess77].

Der Vorrang für Leser oder Schreiber ist in BABEL leicht durch Aktionspriorität zu spezifizieren. Im folgenden Beispiel haben die Schreiber (Prozesse, die die Aktion 'write' verlangen) Vorrang vor den Lesern.

```
INTERPROCESS_MODULE   shared_item;

     VAR shared_data : LOGICAL(7..0) INITIAL= 0;
```

```
    ACTION  read;
    INTERFACE  a(7..0) : OUTPUT;
               SEQUENCE   a := shared_data  END
    END ACTION;

    ACTION  write;
    INTERFACE  b(7..0) : INPUT;
               SEQUENCE   shared_data := b  END
    END ACTION;

    PRIORITY  write > read;
END INTERPROCESS.
```

Die Einfachheit dieser Beschreibung liegt an der von BABEL verwendeten Ausschlußregel. Zwei oder mehrere aktive Kopien der *read*-Aktion sind konfliktfrei und dürfen gleichzeitig aktiviert werden. Die Aktionen *write* sind dagegen nicht konfliktfrei untereinander, die Aktionen (write,read) auch nicht, sie dürfen deswegen auch nicht gleichzeitig aktiv sein. Der für dieses Beispiel automatisch erzeugte Schutzmechanismus garantiert, daß das gemeinsame Mittel ausschließlich durch einen einzigen Schreiber oder mehrere Leser belegt werden kann.

A2. *BABEL Beschreibung eines Mehrprozessorsystems*

Das System 'multiprocessor' besteht aus drei gleichen Prozessoren und einem gemeinsamen Speicher. Jeder Prozessor arbeitet mit "fetch look ahead". Dafür werden zwei Unterprozesse benötigt: ein Unterprozeß, um die Befehle aus dem Speicher zu holen, und ein zweiter, um die Operanden zu holen und die Befehle auszuführen. Da alle Prozessoren gleich sind, genügt es, einen einzigen Prozeß-Typ ('processor') zu beschreiben, der dreimal verkörpert wird. Der Speicher ist ein gemeinsames Mittel und wird als IPM-Typ ('memory') vereinbart. Die Verkörperung dieses Speichers bekommt den Namen 'main-mem'.

Die Typen 'processor' und 'memory' gehören in diesem System zur gleichen hierarchischen Ebene. Der Typ 'processor' ist ein nichtterminaler Prozeß, und seine zwei Unterprozesse ('f-unit' und 'exec-unit') bilden eine neue untergeordnete Ebene. In dieser Ebene stellt der IPM-Typ 'int-res' die gemeinsamen Mittel dar. Nach der Beschreibung dieser drei Elemente müssen sie verkörpert und verbunden werden, um die Beschreibung des Prozeß-Typs 'processor' zu vervollständigen.

```
SYSTEM  multiprocessor;

  (* global pins   *)
  INTERFACE clock : CLOCK;  rst : INPUT;  error : OUTPUT;

  (* global memory *)
  INTERPROCESS_MODULE  memory;
     VAR sp  : ARRAY [255..0] OF LOGICAL(7..0);
         mar, mdr: LOGICAL(7..0);   (* address & data reg. *)

     ACTION read_mem;
       INTERFACE  a(7..0):INPUT;  d(7..0):OUTPUT;
           SEQUENCE   mar:=a;  mdr:=sp[mar];  d:=mdr END
     END;

     ACTION write_mem;
       INTERFACE  a(7..0):INPUT;  d(7..0):INPUT;
           SEQUENCE  mar:=a;  mdr:=d;  sp[mar]:=mdr END
     END;
  END INTERPROCESS.

  (*             **     processor   **                  *)
  (* data and instructions are stored in the ipm 'memory'  *)
  PROCESS  processor;
    RESOURCE mm : memory;                  (* external ipm *)
    INTERFACE  error : OUTPUT;  pck, prst  : INPUT;

    (*  *int_res* : Internal resources of 'processor'      *)
    INTERPROCESS_MODULE   int_res;
       VAR tail, head, size : LOGICAL(3..0)  INITIAL=0;
           pc   : LOGICAL(7..0)  INITIAL=0;
           reset: LOGICAL        INITIAL=0;
           queue: ARRAY[15..0] OF LOGICAL(7..0);
           empty, full : CONDITION;

       STRUCTURE  empty := size=0,  full := size=15  END;

       ACTION read_queue;
         INTERFACE v(7..0):OUTPUT;
         SEQUENCE  IF empty THEN WAIT FI;
                   v:=queue[head];
                   FORK  size:=size-1,  head:=head+1  JOIN
         END
       END;
```

```
   ACTION read_pc;
     INTERFACE v(7..0):OUTPUT;
     SEQUENCE  IF full THEN WAIT FI;
               v:=pc;
               FORK  pc:=pc+1,  reset:=0  JOIN  END
   END;

   ACTION write_queue;
     INTERFACE  v(7..0):INPUT;
     SEQUENCE  IF reset=0
                  THEN  queue[tail]:=v;
                  FORK size:=size+1, tail:=tail+1 JOIN
               FI  END
   END;

   ACTION update_pc;
     INTERFACE  v(7..0):INPUT;
     SEQUENCE
        FORK tail:=0, head :=0,
             size:=0, reset:=1,  pc  :=v  JOIN  END
   END;
END INTERPROCESS.

(*  ** fetch instruction unit **    *)
PROCESS   f_unit;
   RESOURCE    pq : int_res;
   INTERFACE   reset, ck : INPUT;
   VAR  ad   : LOGICAL(7..0)  ;
        code : LOGICAL(7..0)   INITIAL = 0;
        halt : LOGICAL ;
   CLOCKBASE ck;

   FUNCTIONAL
       IF (code > 3) AND (code < 8)
          THEN halt:=1  ELSE halt:=0  FI,
       IF reset THEN RESET main
                ELSE IF halt  THEN STOP  main  FI  FI
   END FUNCTIONAL;

   SEQUENTIAL main;
           PERFORM  pq.read_pc(ad);
           PERFORM  mm.read_mem(ad,code);
           PERFORM  pq.write_queue(code)
   END;
END PROCESS.  (* end f_unit *)
```

```
(*         **       execution unit     **            *)
(*   main_task -   decodes and executes the codes     *)
(*   int_proc  -   communicates with the I/O devices  *)
 PROCESS   exec_unit;
  RESOURCE   pq : int_res;
  INTERFACE  in_pins(7..0)       : INPUT;
             out_pins(7..0)      : OUTPUT;
             rd, wr, code_error : OUTPUT;
             done_read, done_write, ck, reset  : INPUT;

  (*  carriers to both tasks        *)
  VAR  ready, error    : LOGICAL       INITIAL=0;
       read, write     : LOGICAL       INITIAL=0;
       ac, ir          : LOGICAL(7..0) INITIAL=0;
       d, opa          : LOGICAL(7..0);
       int, two_bytes, zero,signal  : LOGICAL;
  CLOCKBASE ck;

  FUNCTIONAL                  (* global functional part *)
     int := done_write OR done_read,
     rd  := read,           wr  := write,
     code_error:= error,    two_bytes:= ir(3),
     zero:= ac=0,           signal :=  ac(7),
     IF reset THEN RESET main_task
              ELSE IF int   THEN STOP  main_task FI,
                   IF ready THEN CONTINUE main_task FI FI,
     IF  int   THEN  START int_proc FI,
     IF  ready OR reset  THEN  STOP  int_proc FI
  END FUNCTIONAL;

  SEQUENTIAL main_task;
     error := 0;
     PERFORM  pq.read_queue(ir);
     IF two_bytes  THEN  PERFORM  pq.read_queue(opa) FI;
     CASE ir(3..0) OF
           0: :                                  (*NOP  *)
           1:  ac:= NOT (ac) :                   (*NEG  *)
           2:  ac:= 0 :                          (*CLEAR*)
           3:  ac:= ac + 1 :                     (*INC  *)
           8:  PERFORM  mm.read_mem(opa,ac):     (*LOAD *)
           9:  PERFORM  mm.write_mem(opa,ac) :   (*STR  *)
          10:  PERFORM  mm.read_mem(opa,d);      (*ADD  *)
               ac:=ac + d :
```

```
            11:  PERFORM  pq.update_pc(opa) :      (*JUMP *)
            12:  IF zero                           (*JZ   *)
                    THEN PERFORM pq.update_pc(opa) FI:
            13:  IF signal                         (*JS   *)
                    THEN PERFORM pq.update_pc(opa) FI:
            14:  read := 1;                        (*IN   *)
                 WHILE NOT(ready) DO OD; ready := 0:
            15:  FORK write:=1, out_pins:=ac JOIN; (*OUT  *)
                 WHILE NOT (ready) DO OD; ready := 0 :
        4,5,6,7: error := 1 :                      (*error*)
        FO
    END SEQUENTIAL;   (*   main *)

    SEQUENTIAL int_proc;
           IF write  THEN  [out_pins:='Z', write := 0]
                     ELSE  [ac := in_pins, read := 0]  FI;
           ready :=1
    END SEQUENTIAL;  (* int_proc  *)
    END PROCESS.     (* exec_unit *)

   (* instantiation and association inside   'processor'  *)
    CREATE     pc_and_queue : int_res;
    ACTIVATE   execute : exec_unit,   fetch : f_unit;
    ASSOCIATE  execute WITH pc_and_queue,
               fetch   WITH pc_and_queue;

   (* access priority to the global memory      *)
    SET TO  mm   PRIORITY  execute > fetch;

  FUNCTIONAL
    error := execute.code_error,
    execute.reset := prst,
    fetch.ck := pck
  END;
  END PROCESS. (* end    processor *)

(* instantiation and association  inside the system   *)
 CREATE  main_mem : memory;
 ACTIVATE   P1,P2,P3  : processor;
 ASSOCIATE  P1,P2,P3  WITH  main_mem;

(* access   priority  to main_mem  *)
 SET TO  main_mem  PRIORITY   P1 > P2 = P3;
```

```
(* structural part of the system  *)
 FUNCTIONAL
   error := P1.error  OR P2.error OR  P3.error,
   P1.prst := rst,    P2.prst := rst,    P3.prst := rst,
   P1.pck := clock,   P2.pck := clock,   P3.pck := clock
 END FUNCTIONAL;
END.
```

Dieses Beispiel zeigt die Mächtigkeit der Sprache in der Darstellung nebenläufiger Prozesse auf verschiedenen hierarchischen Ebenen. Die klare Trennung zwischen exklusiven und gemeinsamen Mitteln und die hierarchische Gliederung der Prozesse vereinfacht die Beschreibung und die Schaltungssynthese.

A3. *Übersetzerausgabe*

Wegen des Umfangs der vollständigen Übersetzerausgabe werden im folgenden nur ein paar ausgewählte Abschnitte betrachtet. Das Symbol "*" ist ein originales Ausgabesymbol und bedeutet, daß das entsprechende Element fehlt. Die folgenden Symbole gehören nicht zu der originalen Ausgabe:

   ':' in der ersten Spalte bedeutet, daß Zeilen fehlen.
   '@' bedeutet, daß nach diesem Symbol eingefügte Sätze Erklärungen sind.

```
SYSTEM : multiprocessor
======
:
INTERPROCESS MODULE
==================
IPM NAME : memory              @ In dieser Ebene gibt es nur einen IPM
:
VARIABLE  sp(7..0) : LOG_M[255..0]    mar(7..0) : LOG_R
          mdr(7..0) : LOG_R
END VARIABLE
WAIT_CONDITION   *                          @ Es gibt keine Bedingung
:
ACTION                                  @ Zwei Aktionen sind vereinbart
ACTION NAME : read_mem
PARAMETER    a(7..0) : IN        d(7..0) : OUT
TESTED CONDITION : *
SOURCE OPERAND                                      @ Quelleoperanden
```

```
    sp(7..0)[mar..mar] : LOG_M   mar(7..0) : LOG_R
    mdr(7..0) : LOG_R
DESTINATION OPERAND                                    @ Zieloperanden
    mar(7..0) : LOG_R            mdr(7..0) : LOG_R
OPERATION                      @ Hier steht der Aktionsverhaltensgraph
:
END ACTION read_mem

ACTION NAME : write_mem
:
END ACTION write_mem
END IPM memory

PROCESS
=======
PROCESS NAME : processor             @ Ein Prozeß mit Unterprozessen
============
EXTERNAL RESOURCE                          @ vereinbarte Außenmittel
  mm  :  IPM TYPE memory
REFERED IPM ACTION                              @ verwendete Aktionen
  mm  :  write_mem, read_mem
:
INTERPROCESS MODULE
==================
IPM NAME : int_res
:
VARIABLE
 tail(3..0): LOG_R INITIAL = 0    head(3..0): LOG_R INITIAL=0
 size(3..0): LOG_R INITIAL = 0    pc(7..0): LOG_R INITIAL = 0
 reset(0..0): LOG_R INITIAL= 0    queue(7..0): LOG_M[15..0]
 empty(0..0) : WCN                       @ WCN - Kennzeichen für die
 full(0..0) : WCN                        @  vereinbarten Bedingungen
END VARIABLE

WAIT_CONDITION                   @ Liste der vereinbarten Bedingungen
CONDITION NAME : empty
SOURCE OPERAND      size(3..0) : LOG_R
END CONDITION empty

CONDITION NAME : full
SOURCE OPERAND      size(3..0) : LOG_R
END CONDITION full

FUNCTIONAL                          @ Die Definition einer Bedingung
FUNCTIONAL CONDITION  *
```

```
OPERATION                          @ Verhaltensgraph der Bedingung 'empty'
 Index      : 74      Tag : SO      Operation  : EQL
 Inputs     : size(3..0) : LOG_R    0
 Outputs    : empty(0..0) : WCN     Conditions : X
:
END FUNCTIONAL

ACTION
ACTION NAME : read_queue
PARAMETER   v(7..0) : OUT
TESTED CONDITION                   @ Innerhalb dieser Aktion ist die
    empty                          @ Bedingung "empty" verwendet
SOURCE OPERAND
    queue(7..0)[head..head] : LOG_M      size(3..0) : LOG_R
    head(3..0) : LOG_R
DESTINATION OPERAND
    size(3..0) : LOG_R        head(3..0) : LOG_R

OPERATION                          @ Teilgraph einer IF-WAIT (3 Knoten)
 Index      : 76       Tag : IW          Operation  : TR
 Inputs     : empty(0..0)  : WCN
 Outputs    : SYMB_18(0..0): AUX         Predecessor: *
 Condition  : 0        Successor : 77
 Condition  : 1        Successor : 83

 Index      : 83       Tag : CW          Operation  : X
 Inputs     : *        Outputs    : *
 Predecessor: 76       Condition  : X    Successor : 77

 Index      : 77       Tag : IE          Operation  : X
 Inputs     : *        Outputs    : *
 Predecessor: 76  83   Condition  : X    Successor : 78
 :
END ACTION read_queue
:
END IPM int_res

PROCESS
=======
PROCESS NAME : f_unit
=============
EXTERNAL RESOURCE     pq : IPM TYPE int_res
REFERED IPM ACTION
  pq : read_pc, write_queue       mm : read_mem
:
```

```
END PROCESS f_unit

PROCESS NAME : exec_unit
=============
EXTERNAL RESOURCE     pq : IPM TYPE int_res
REFERED IPM ACTION
  pq : read_queue, update_pc      mm : write_mem, read_mem
:
SEQUENTIAL : main_task
==========
VARIABLE    d(7..0) : LOG_R       opa(7..0) : LOG_R
END VARIABLE

OPERATION                        @ Darstellung der PERFORM-Konstruktion
:
 Index       : 31        Tag : ACALL                @ Kennzeichen ACALL
 Operation   : pq : RES  read_queue : ACT
 Inputs      : *         Outputs    : ir(7..0) : LOG_R
 Predecessor: 30         Condition  : X          Successor : 33
:
END SEQUENTIAL main_task
:
END PROCESS exec_unit

INSTANTIATION
=============
  pc_and_queue  :  IPM TYPE int_res
  execute  :  PROC TYPE exec_unit
  fetch  :  PROC TYPE f_unit
PROCESS CONNECTION
==================
  execute  :  pc_and_queue
  fetch  :  pc_and_queue
INTERPROCESS CONNECTION
======================
  pc_and_queue  :  execute, fetch
PRIORITY
========
  mm  :     execute   >   fetch
:
FUNCTIONAL
==========
:
```

```
OPERATION                          @ Ein Knoten mit übertragenem Operand
 Index       : 106       Tag : SO          Operation  : TR
 Inputs      : execute.code_error(0..0) : OUT_T
 Outputs     : error(0..0) : OUT           Conditions : X
:
END FUNCTIONAL
END PROCESS processor

INSTANTIATION
-------------
  main_mem  :  IPM TYPE memory
  P1  :  PROC TYPE processor
  P2  :  PROC TYPE processor
  P3  :  PROC TYPE processor
PROCESS CONNECTION
------------------
  P1  :  main_mem
  P2  :  main_mem
  P3  :  main_mem
INTERPROCESS CONNECTION
-----------------------
  main_mem  :  P1, P2, P3, P4
PRIORITY
--------
  main_mem  :   P1   >   P2 = P3
:
END SYSTEM multiprocessor
```

A4. *Musterlösungen für den Interaktionsmechanismus*

In diesem Abschnitt wird eine Familie von Musterlösungen für die Realisierung des Interaktionsmechanismus vorgeschlagen (vgl. Abschnitt 8.4). Alle Schaltungen sind allgemein für eine Anzahl von n Prozessen und von m Aktionen konzipiert. Bessere Lösungen für einzelne Fälle lassen sich allerdings finden (z.B wenn n=2 ist). Es ist deshalb ratsam, daß auch diese Lösungen Bestandteile einer Bibliothek der Interaktionsmuster werden.

a) *Musterlösung für den Fall CC2.PC3*

Der Fall CC2.PC3 bedeutet, daß alle Aktionen mit gegenseitigem Ausschluß durchzuführen sind und die gleiche Priorität haben; die Prozesse sind bezüglich ihrer Prioritäten ausschließlich durch die Relation > geordnet. In diesem Fall ist keine Identifikation für Aktionen erforderlich. Die hier vorgeschlagene Musterlösung bildet den elementaren Baustein der Familie und umfaßt das Netzwerk des Bildes A4.1, welches durch eine 'daisy chain' [Thur72] zu realisieren. ist.

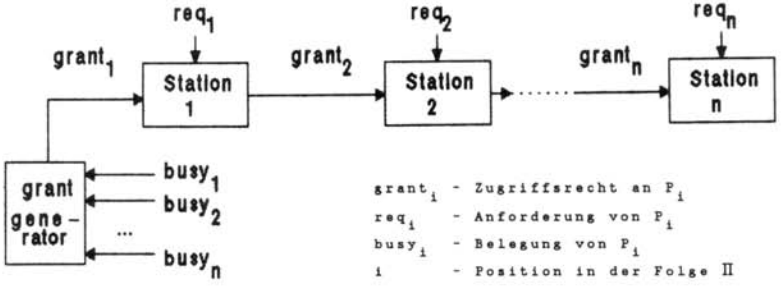


Bild A4.1 - Schutzmechanismus für CC2.PC3

Der Index steht für die Position jedes Prozesses in der Folge Π. Daher entspricht die erste Station dem Prozeß mit der größten Priorität. Das Muster für die Stationen 1 bis n-1 ist im Bild A4.2 abgebildet. Die Station n benötigt keine Realisierung, weshalb das Anforderungssignal req_n überflüssig ist. Den Zugriffsrechtserzeuger (grant_generator) zeigt das Bild A4.3.

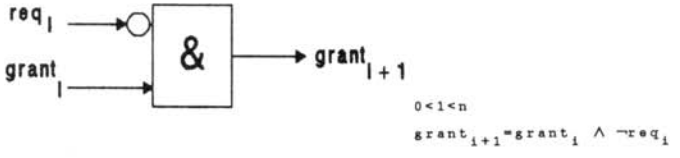
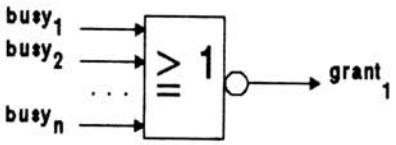


Bild A4.2 - Station i der Schutzkette



$grant_1 = \neg(busy_1 \vee .. \vee busy_n)$

Bild A4.3 - Zugriffsrechtserzeuger

Die entsprechende Anforderungssektion jedes Prozesses P_i (Bild A4.4) liefert das Anforderungs- und das Belegungssignal (req_i und $busy_i$) und testet das Zufriffsrechtssignal ($grant_i$). In Bild A4.4 sind schon in jedem Zweig die erforderlichen Steuersignale (req und busy) berücksichtigt.

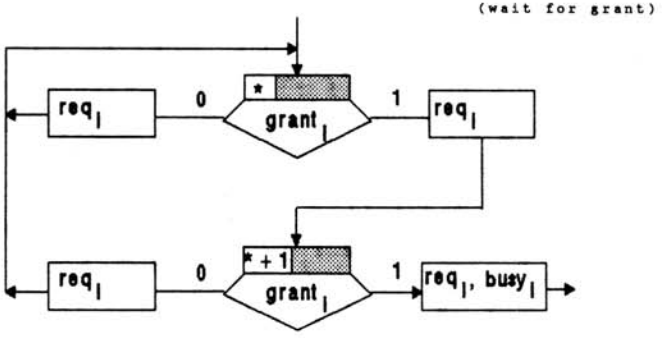


Bild A4.4 - Anforderungssektion für den Prozeß P_i

Die dargestellte Anforderungssektion entsteht aus zwei Übergangseinheiten. Nach der Expansion (siehe Abschnitt 8.1) kann man die Tranformation 'Kompaktierung' unter Berücksichtigung der Restriktionen 4 und 5 (Verlagerung in eine iterative Sektion) anwenden.

Die Interaktion zwischen den Anforderungssektion und der Schutzkette läuft wie folgt:

- Wenn keine Prozesse den IPM belegen, haben alle Signale 'grant' den logischen Wert '1'.

- Ein Prozeß P_i, der den IPM belegen will, liefert ein Anforderungssignal (req_i) und testet das Zugriffsrechtssignal ($grant_i$). P_i bleibt solange in diesem Zustand, bis er das Zugriffsrecht bekommt ($grant_i$=1).

- req_i sperrt alle Zugriffsrechtssignale mit Prioritätsindex größer als i (d.h. kleinere Prioritäten).

- Hat P_i das Zugriffsrecht bekommen, muß er noch einmal dieses Signal testen. Das doppelte Testen garantiert, daß P_i den IPM nicht belegt, falls ein Prozeß mit größerer Priorität gleichzeitig oder kurz nachher Zugriff zum IPM verlangt.

- Wird das Zugriffsrecht bestätigt, belegt der Prozeß durch das Signal $busy_i$ den IPM.

- $busy_i$=1 macht $grant_1$=0. Damit werden alle 'grant' den Wert Null annehmen. $busy_i$ muß aktiv bleiben, solange P_i den IPM belegt (d.h. alle Zustände der aktivierten Aktion müssen auch $busy_i$ liefern).

- Beim Verlassen des IPM's setzt P_i $busy_i$ wieder auf Null. Damit wird $grant_1$ gleich '1', und der Prozeß mit der größten Priorität, der auf das Zugriffsrecht wartet, darf die Folge fortsetzen.

Einschränkung bei der Anwendung dieser Lösung:

$1 < i \leq n$:

$$\text{delay } (busy_i \text{ until } grant_{i-1}) < 2.\ clock\_periode_{i-1}$$

Nachteile der Lösung:

- Es dauert mindestens 2 Zyklen (clock periodes) bis ein Prozeß einen IPM belegen kann.

- Angenommen, es haben alle n Prozesse die gleiche Taktfrequenz, so dauert es im schlimmsten Fall n Zyklen bis ein verfügbarer IPM belegt wird. Der schlimmste Fall tritt dann ein, wenn zuerst P_n, nachher P_{n-1}, nachher P_{n-2}, u.s.w den IPM verlangen, bis endlich P_1 den IPM verlangt.

Um diese Musterlösung zu vervollständigen, fehlt nur noch die Wartesektion. Eine Wartesektion ersetzt einen Warteknoten einer Aktion. Je nach Lage dieses Knotens (siehe Abschnitt 3.2.5 und Bild 3.2.2) sind bei der

Expansion zwei mögliche Wartesektionen zu verwenden. Diese sind im Bild A4.5 bzw. A4.6 gezeigt. Bild A4.5 zeigt rechts die Darstellung einer Wartesektion ohne Rückkehr in die Aktion.

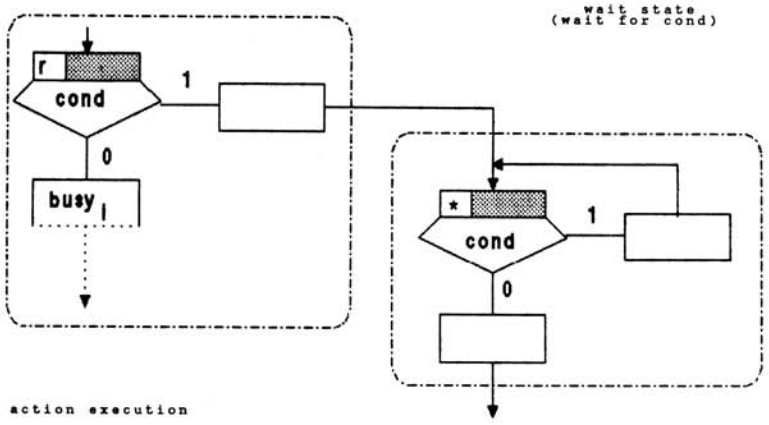


Bild A4.5 - Wartesektion ohne Rückkehr in die Aktion

Bild A4.6 zeigt die Darstellung einer Wartesektion mit Rückkehr in die Aktion. Die rechts des Bildes A4.6 abgebildete Wartesektion funktioniert nach dem gleichen Prinzip wie die Anforderungssektion (Bild A4.4), d.h, das Zugriffssrechtssignal muß 2-mal bestätigt werden. Zum Verlassen dieses Wartezustandes (siehe Bild 3.2.2) muß jetzt allerdings die Bedingung (cond=0) und das Zugriffsrechtssignal ($grant_i$) vorhanden sein. Das Anforderungssignal (req_i) soll erst auf '1' gesetzt werden, wenn 'cond' = 0 ist (d.h die Bedingung zum Verlassen des Wartezustandes erfüllt ist).

Es gibt zwei mögliche Strategien für die Behandlung der Anforderung eines Prozesses, der sich in diesem Wartezustand befindet: entweder bleibt er bei seiner gewöhnlichen Priorität oder nimmt er eine Priorität größer als die der anderen Prozesse an. Im ersten Fall ist nichts in der Schutzkette zu ändern. Dabei ist req_i=req_j und $grant_i$=$grant_j$. Im zweiten Fall kann man zum Beispiel die Kette verdoppeln, und i wird 2.j. Diese aufwendige Lösung läßt sich je nach Anzahl von Prozessen verbessern, die sich gleichzeitig im Wartezustand befinden können.

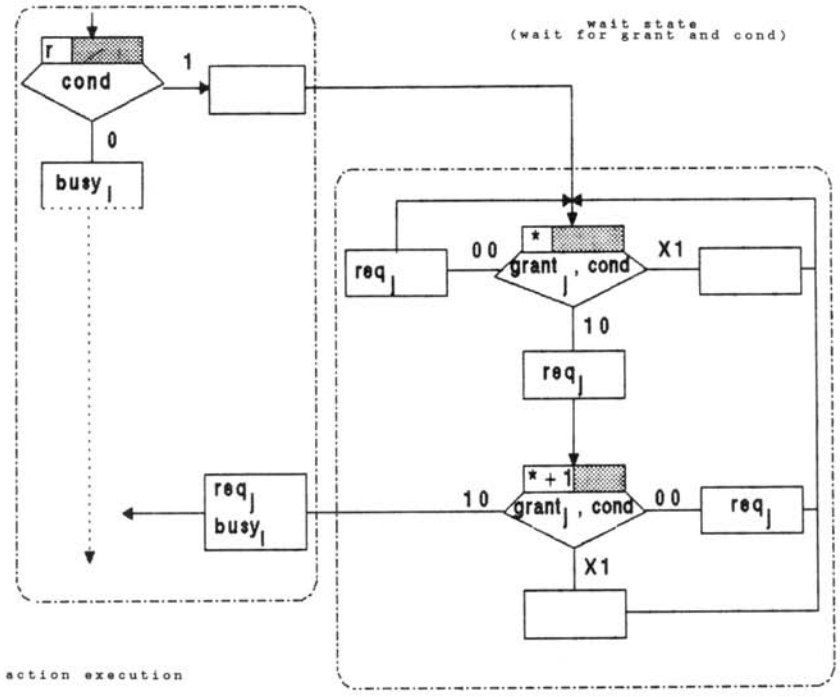


Bild A4.6 - Wartesektion in P_i mit Rückkehr in die Aktion

b) *Musterlösung für den Fall CC2.PC9*

Der Fall CC2.PC9 ist ein weiterer mit fester Priorität. Hier sind alle Aktionen mit gegenseitigem Ausschluß durchzuführen und die Prozesse und Aktionen sind bezüglich ihrer Prioritäten ausschließlich durch die Relation > geordnet. In diesem Fall sind unterschiedliche Anforderungssignale für jede der Aktionen erforderlich. Die hier vorgeschlagene Musterlösung umfaßt die gleiche Schutzkette wie Fall CC2.PC3 (Bild A4.1), aber mit einer größeren Anzahl von Stationen.

Es sei ein IPM mit m Aktionen angenommen, die von n Prozessen verlangt werden können. Das folgende Bild zeigt eine Schutzkette mit m.n Stationen, die dieses Prioritätsproblem löst.

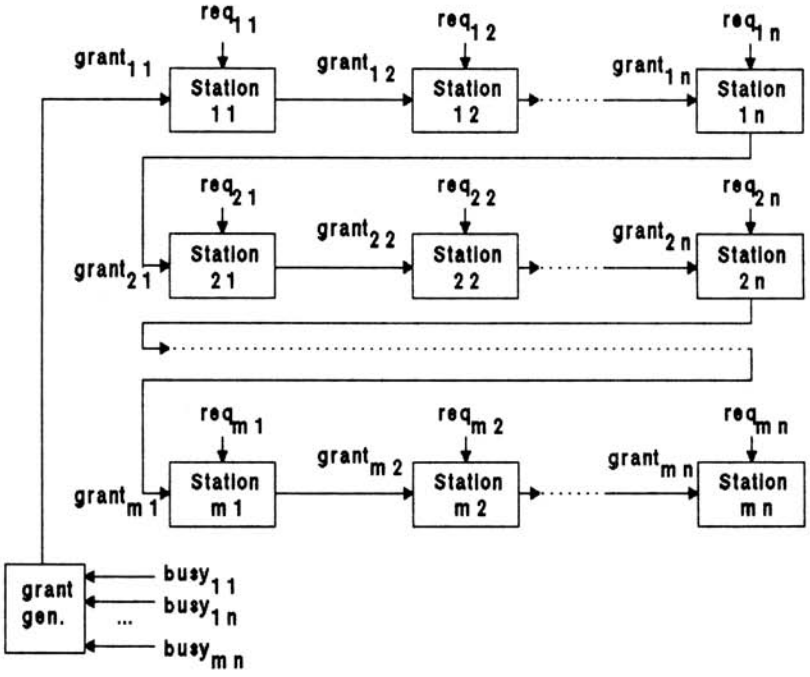


Bild A4.7 - Schutzkette für CC2.PC9

Der erste Index steht für die Position jeder Aktion in der Folge Π der Aktionsprioritäten, und der zweite Index steht für die Position jedes Prozesses in der Folge Π der Prozeßprioritäten. Daher entspricht die erste Station dem Prozeß mit der größten Priorität, der die Aktion mit der größten Priorität verlangt. Die Stationen 11 bis m.(n-1) sind genau die Stationen des Falles CC2.PC3 (Item a). Die Station m.n benötigt auch hier keine Realisierung. Der Zugriffsrechtserzeuger (grant_generator), die Anforderungssektion und die Wartesektionen haben auch hier den in den Bilder A4.3 bis A4.6 dargestellten Aufbau.

Falls nicht alle Prozesse alle Aktionen verlangen können, wird diese Kette entsprechend verkürzt. Für jeden Prozeß sind nur Stationen für diejenigen Aktionen vorhanden, die er benötigt.

c) *Musterlösung für den Fall CC2.PC6*

Die Aktionen mit gleichen Prioritäten werden in diesem Fall als eine einzige betrachtet, und die Lösung für CC2.PC9 (Item b) wird mit einer verkürzten Kette verwendet. Sei zum Beispiel die Spezifikation für Aktionspriorität A1 > A2 = A3 > A4, so besteht die zu realisierende Kette aus 3 Reihen von Stationen: eine Reihe für A1, eine zweite für A2 und A3, und eine letzte für A4. Die Anforderungen nach A2 und A3 brauchen nicht unterschieden zu werden.

d) *Musterlösung für den Fall CC2.PC1*

Die Schaltung für diesen Fall ist der elementare Grundstein für zyklische Priorität. Zyklische Priorität bedeutet hier, daß jedesmal ein anderer Prozeß bei der Verteilung des Zugriffsrechts die größte Priorität hat. Das Bild A4.8 zeigt einen einstufigen Schutzring mit n Stationen, der die zyklische Priorität realisiert. Eine Ringstation ist im Bild A4.9 abgebildet.

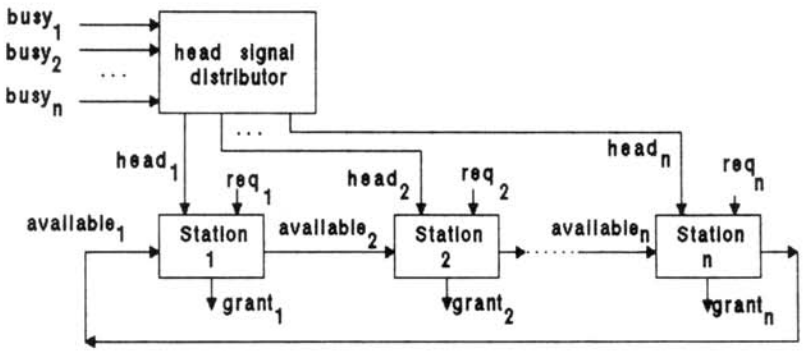


Bild A4.8 - Schutzring für CC2.PC1

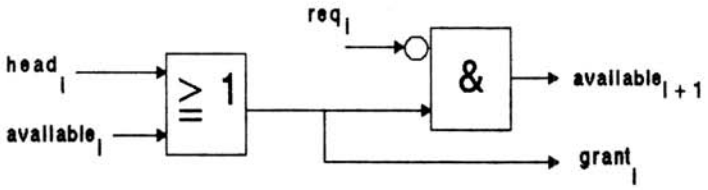


Bild A4.9 - Ringstation für P_i

Der Ring funktioniert wie folgt:

- Nur eines der Signale 'head' ist aktiv (logisch '1'). Eine Station i mit zugeteiltem $head_i$ hat die größte Priorität, solange $head_i$ aktiv ist.

- Das aktive Signal 'head' bestimmt die Aktivität des entsprechenden Signals 'grant' (falls $head_i$=1 ist, dann ist auch $grant_i$=1).

- Von der Station abgesehen, welche das aktive Signal 'head' bekommt, funktioniert der Ring genau wie die Kette des Falles CC2.PC3 (Bild A4.1).

- Sobald ein Prozeß den IPM freigibt, teilt der Verteiler des Signals 'head' einer anderen Station die größte Priorität zu.

Eine mögliche Realisierung für den Verteiler zeigt das Bild A4.10.

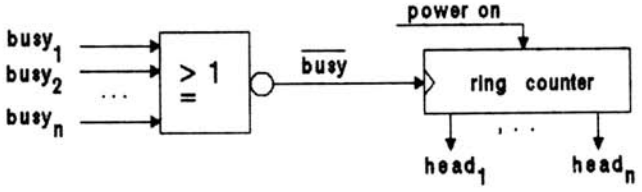


Bild A4.10 - Prioritätsverteiler

Bei jeder Freigabe des IPM's (durch die Negation irgendeines Signals $busy_i$) wird der Ring-Zähler weitergeschaltet, so daß $head_{i+1}$ nach $head_i$ aktiviert wird. In anderen Worten: der Prozeß mit der größten Priorität

vor der Belegung des IPM's bekommt nach der Freigabe dieses IPM's die kleinste Priorität.

Der Schutzring erlaubt weiterhin die Anwendung der Anforderungs- und Wartesektionen, die in den Bildern A4.4 bis A4.6 abgebildet sind. Der Schutzring ist im Vergleich zu einer Schutzkette für die gleiche Anzahl von Prozessen aufwendiger (größere Anzahl digitaler Komponenten). Bis eine bessere Lösung zur Verfügung steht, muß der Anwender Bescheid darüber wissen und eventuell fest geordnete Prioritäten zwischen den Prozesse spezifizieren, falls das nicht gegen die Anforderungen des Systems verstößt.

e) *Musterlösung für den Fall CC2.PC2*

Hier handelt es sich um gemischte Prozeßprioritäten. Es wird angenommen, daß eine Gruppe ein einziger Prozeß oder eine Menge von Prozessen mit gleicher Priorität sei, die in einer Spezifikation vorkommen. Die Gruppen sind dann geordnet:

$G_1 > G_2 > \ldots > G_n$

Realisierung:

- Jede Gruppe verfügt über eine Gruppenstation. Eine Gruppenstation ist eine Station wie in Bild A4.2 abgebildet. Alle Gruppenstationen sind in Kette verknüpft (Bild A4.1). Da es sich hier um eine Kette von Gruppenstationen handelt, werden die Signale dieser Stationen 'group-req' und 'group-grant' genannt.

  - 'group-req' entsteht aus der Operation ODER (OR) aller Signale 'req' der Elemente der Gruppe.
  - Jede Gruppe liefert ein entsprechendes Signal 'group-busy'. Dieses Signal entsteht aus der Operation ODER (OR) aus allen 'busy'-Signale der Elemente der Gruppe.
  - group-grant_1 entsteht aus der NOR-Verknüpfung aller 'group-busy'-Signale.

- Zusätzlich zu der Gruppenkette gibt es für jede Gruppe, deren Anzahl von Elemente größer als eins ist, einen Ring (A4.8 und A4.9), wobei der Prioritätsverteiler entsprechend angepaßt wurde (Bild A4.11).

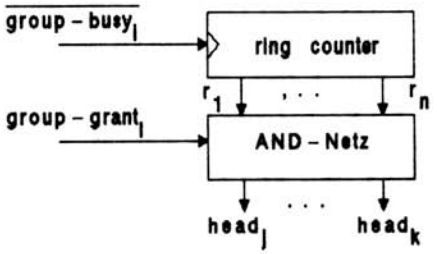


Bild A4.11 - Prioritätsverteiler der Gruppe G_i

Der Sperre des 'head'-Signals (durch eine AND-Netz) dient dazu, um den Gruppering zu deaktivieren.

f) *Musterlösung für den Fall CC3.PC9*

CC3 weist darauf hin, daß einige Paare von Aktionen des IPM's konfliktfrei sind und deswegen gleichzeitig aktiv sein dürfen (Abschnitt 8.2.1). Die Realisierung der CC3-Fälle wurde dadurch eingeschränkt, daß eine Aktion nicht aktiviert werden darf, wenn der IPM schon belegt ist, sogar wenn diese Aktion konfliktfrei mit den schon aktiven Aktionen ist.

Die hier vorgeschlagene Lösung bezieht sich auf die Musterlösung für den Fall CC2.PC9 (Item b). Dabei werden je nach Inhalt der Menge der konfliktfreien Aktionen KfA einige Reihen der Kette komprimiert und/oder die Verbindungen zwischen den Reihen neu hergestellt.

Die Struktur der Reihe einer Aktion ist wie folgt festgelegt:

- Falls $(a_i,a_i) \in$ KfA, dann enthält die Reihe i für die Anforderungen aller Prozesse eine einzige Station und req_{i1}=req_{i2}=...=req_{in} und $grant_{i1}$=$grant_{i2}$=...=$grant_{in}$.

- Falls nicht, dann ist die Reihe i eine Kette von Stationen (eine für jeden Prozeß, der die Aktion a_i verlangen kann).

Es folgt ein Schema für die Verbindung der Reihen für zwei bis drei Aktionen ($m \leq 3$), wobei $a_1 > a_2 > a_3$.

Reihe 2: $(a_1,a_2) \in KfA \Rightarrow grant\text{-}in_2 = grant\text{-}in_1$
$(a_1,a_2) \notin KfA \Rightarrow grant\text{-}in_2 = grant\text{-}out_1$

Reihe 3: $(a_2,a_3) \in KfA \Rightarrow grant\text{-}in_3 = grant\text{-}in_2$
$(a_2,a_3) \notin KfA$:
$(a_1,a_3) \in KfA \Rightarrow grant\text{-}in_3 = grant\text{-}out_2 \vee (grant\text{-}in_1 \wedge \text{NOT } grant\text{-}in_2)$
$(a_1,a_3) \notin KfA \Rightarrow grant\text{-}in_2 = grant\text{-}out_1 \wedge grant\text{-}out_2$

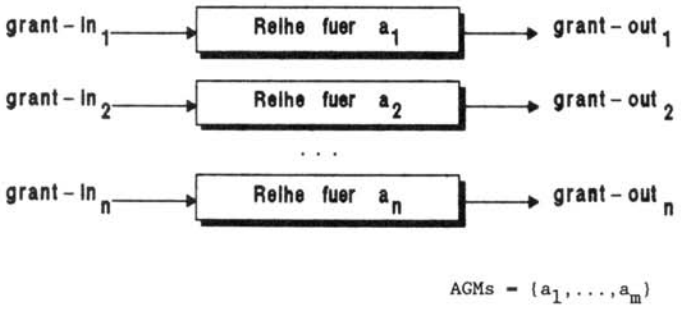


Bild A4.12 - Schema für CC3.PC9

L e b e n s l a u f

Taisy Silva Weber, geb. Pipolo Batista da Silva

30.01.1953	geboren in Natal, Brasilien, als Tochter des Meteorologen Edemar B. da Silva und seiner Frau Carminella geb. Pipolo.
1960-1964	Besuch der Schule '1. de Maio' in Porto Alegre.
1965-1971	Besuch des Gymnasiums 'Candido Jose de Godoi' in Porto Alegre.
1972	Aufnahmeprüfung bei der UFRGS (Universidade Federal do Rio Grande do Sul) und Beginn des Studiums der Elektrotechnik.
1973-1974	Nebentätigkeit als Programmiererin an der 'Pontificia Universidade Catolica do Rio Grande do Sul' und später bei der Firma 'Siderurgica Riograndense'.
1975-1976	Nebentätigkeit als Lehrerin für Elektrotechnik an der Technischen Schule 'Parobe'.
1976	Tätigkeit als Werkstudentin bei der Firma VARIG.
Dez. 1976	Studienabschluß in Elektrotechnik an der UFRGS.
1977-1979	'Mestrado' in Informatik an der UFRGS.
Seit 1978	Wissenschaftliche Angestellte an der UFRGS, zuerst als Hilfsdozent in der Elektrotechnik, nach dem Mestradoabschluß als 'Assistente' und später als 'Adjunto' in der Informatik ('Assistente' und 'Adjunto' sind Bezeichnungen für die zweite und dritte Stufe der Laufbahn eines brasilianischen Hochschullehrers).
Seit 1982	Beurlaubt zur Promotion an der Universität Karlsruhe als DAAD-Stipendiat.