CAP/SIL - EINE SYSTEMIMPLEMENTIERUNGSSPRACHE FÜR MULTI-MIKROPROZESSOR-SYSTEME AUS DER CAP-SPRACHFAMILIE

Franz J. Rammig, H. W. Röder
Universität Dortmund

Zusammenfassung: Nach einem Überblick über die CAP Sprachfamilie und deren grundlegende Konzepte werden die verschiedenen Vertreter dieser Familie vorgestellt. Dabei wird auf die Version CAP/SIL besonders eingegangen. CAP/SIL enthält einen Modultyp zur Definition getrennt übersetzbarer Betriebsmitteltypen, der einen Kontrollausdruck zur Synchronisation der Moduloperationen enthält. Die Möglichkeit des Kontrollausdrucks wird an einem relevanten Beispiel erläutert und anderen Konzepten gegenübergestellt.

1. Überblick über die CAP-Sprachfamilie

1.1 Historischer Überblick

Erste Ansätze für das CAP-Konzept reichen in das Jahr 1975 zurück. Damals war die Absicht, eine fortgeschrittene, flexible Hardware Beschreibungssprache zu entwickeln [1]. Man findet in diesem Ansatz bereits das Grundkonzept, eine alphanumerische Sprache zum Aufschreiben interpretierter, zeitbewerteter Transitionsnetze (mit einer semantischen Verankerung in Petri Netzen) anzubieten.

Auf der Basis dieses Vorschlages wurde ab 1976 die Version CAP/RTPL [2] implementiert. Neben dem Aspekt der Hardwarebeschreibung liegt hier ein Anwendungsschwerpunkt bei der Programmierung von mikroprozessororientierten Gerätecontrollern. Konsequenterweise findet man in dieser Sprache Konstrukte der postulativen Realzeitprogrammierung. Als Anlehnungssprache wurde aus verschiedenen Gründen PL/1 gewählt. Das Projekt ist inzwischen beendet, das System, bestehend aus Compiler, Interpreter, Binder, Codegenerator, Code-Optimierer, Testhilfesystem, Dokumentationssystem, Programmgenerator, ist im industriellen Einsatz.

Zur gleichen Zeit wurde eine zweite Version von CAP, CAP/SIL konzipiert. Hier galt es, eine hardwarenahe Systemimplementierungssprache für Multiprozessorsysteme zu konzipieren. Diese Aufgabe brachte es mit sich, daß für die Nebenläufigkeit auf Prozedurebene erheblich mehr Unterstützung angeboten wird. Auf Realzeitprogrammierung wurde verzichtet. Als Anlehnungssprache wurde PASCAL gewählt. Besonderes Gewicht haben im Rahmen dieses Projekts Parallelisierungsalgorithmen. Das Projekt steht kurz vor seinem Abschluß.

Seit 1980 wird an einer dritten Version von CAP gearbeitet. Hier geht es dezidiert darum, eine fortgeschrittene Hardwarebeschreibungssprache für einen größeren Bereich von Abstraktionsebenen zu implementieren. Deskriptiven Realzeitbeschreibungen kommt damit eine besondere Bedeutung zu. Neben der Spezifikation von nebenläufigen Kausalitätsstrukturen ist auch derer Realisierung durch getaktete Systeme unmittelbar beschreibbar. Diese Version von CAP

wurde CAP/DSDL genannt und lehnt sich wie CAP/SIL an PASCAL an.

1.2 Das CAP Grundkonzept

Zumindest in ihrer semantischen Verankerung sind alle CAP Dialekte nonprozedurale Sprachen. Sie können jedoch auch prozedural benutzt werden, CAP/SIL sogar nur prozedural.

"Nonprozedurale Spache" soll in diesem Zusammenhang bedeuten, daß die Anordnung der Anweisungen keinen Einfluß auf deren Ausführungszeitpunkt hat. Vielmehr ist die Ausführbarkeitsbedingung für jede Anweisung explizit anzugeben.

Sei

$C = (c_1,\ldots,c_n)$ ein Tupel elementarer Bedingungen,

$D = (d_1,\ldots,d_m)$ ein Tupel von Datenelementen,

$C \cap D = \emptyset$,

P_t ein (partielles) Prädikat auf C,

$A_t : D \to D$ eine (partielle) Abbildung (Datenmanipulation) und

$B_t : C \to C$ eine (partielle) Abbildung (Bedingungsmanipulation).

Dann ist jede Anweisung t einer nonprozeduralen Sprache von der Form:

$P_t \to (A_t, B_t)$;

mit der informellen Semantik:

"Immer wenn $P_t(C)$ wahr wird, wird die Datenmanipulation A_t und die Bedingungsmanipulation B_t ausgeführt."

Die Ausführung von B_t kann zur Folge haben, daß eine (nicht notwendigerweise einelementige) Menge von Anweisungen $t_1,\ldots,t_n$ ausführbar werden. Sie können dann (auch simultan, echte Nebenläufigkeit) ausgeführt werden.

Dieses Konzept läßt sich besonders einfach auf interpretierte Petri Netze übertragen. Dabei spielen Stellen die Rolle von Bedingungsvariablen, das Ausführbarkeitsprädikat und die Bedingungstransformation sind durch die Schaltregel der Transition gegeben, während die Datentransformation durch die Interpretation beschrieben wird.

Der gesamte Datenmanipulationsteil (Deklaration von Datenobjekten, Ausdrücke, Zuweisungen) wird bei den CAP Sprachen von einer Wirtssprache entlehnt (um die babylonische Sprachverwirrung nicht

weiter zu fördern). Die gesamte Kontrollstruktur wird jedoch durch die CAP-eigene Kontrollstruktur ersetzt.

Da in der Regel strikte nonprozedurale Programmierung zu unübersichtlich wäre (nebenläufige Spaghetti-Programmierung), werden in allen CAP-Sprachen spezielle Konstrukte für die strukturierte (nebenläufige) Programmierung angeboten, die in ihrer Notation entsprechenden Konstrukten der Wirtssprache gleichen [3].

Alle CAP-Sprachen haben ein Modulkonzept mit nebenläufig aufrufbaren Moduln (aufrufende Instanz bleibt aktiv). Während in CAP/RTPL und CAP/DSDL ein relativ einfacher Abkapselungsmechanismus (Zuteilung einer einmal vorhandenen Resource nach starrem Prioritätenschema) implementiert ist, bietet CAP/SIL hier erheblich komplexere Möglichkeiten.

1.3 Besondere Eingenschaften der verschiedenen CAP-Dialekte

Hier sollen wesentliche differierende Eigenschaften von CAP/RTPL und CAP/RSDL kurz skizziert werden. Die speziellen Eigenschaften von CAP/SIL werden unter 2. dargestellt und daher hier nicht aufgeführt. Es soll an dieser Stelle betont werden, daß sich alle speziellen Eigenschaften der verschiedenen Versionen auf das Grundkonzept zurückführen lassen.

1.3.1 Besondere Eigenschaften von CAP/RTPL

CAP/RTPL kann sowohl prozedural wie auch nonprozedural benutzt werden. Es erlaubt Nebenläufigkeit auf der Statement- und Prozedurebene. Neben deskriptiver Zeitangaben (Zeitverbrauch elementarer oder komplexer Anweisungen) sind selektive postulative Angaben (einzuhaltender minimaler und maximaler Zeitverbrauch von Programmteilen und Programmen, maximaler erlaubter Speicherbedarf) möglich. Der normalen Kontrollstruktur kann eine Unterbrechungsstruktur (extern und intern auslösbare "Interrupts") überlagert werden. Weiterhin wird sogenannte implicit-Variable hervorzuheben, derer Wert kontinuierlich, von der Kontrollstruktur unabhängig, von den Werten spezifizierbarer Argumente in spezifizierbarer Weise abhängen.

1.3.2 Besondere Eigenschaften von CAP/DSDL

Auch CAP/DSDL kann sowohl prozedural wie nonprozedural benutzt werden und erlaubt Nebenläufigkeit auf der Statement- und Prozedurebene. Zusätzlich wird die Beschreibung realisierender getakteter Systeme besonders unterstützt. Die deskriptiven Zeitangaben sind recht komplex (Unsicherheitsintervalle, wertespezifische Verzögerung), wogegen postulative Angaben nicht vorgesehen sind. Das "Interrupt"-Konzept unterstützt hier besonders auch Intermodulkommunikation. Das "implicit"-Konzept von CAP/RTPL findet sich mit einigen Erweiterungen auch in CAP/DSDL. Das Modulkonzept wurde so erweitert, daß die Implementation in der Version 1.0 (nicht parametrisierter) abstrakter Datentypen unmittelbar beschrieben werden kann.

1.3.3 Tabellarische Übersicht über spezielle Eigenschaften der verschiedenen CAP-Dialekte

Eigenschaft	CAP/CTPL	CAP/DSDL	CAP/SIL
Anlehnungssprache	PL/1	PASCAL	PASCAL
Nebenläufigkeit auf Statementebene	ja	ja	ja
Strukturierte Nebenläufigkeit	ja	ja	nur
Nebenläfigkeit auf Prozedurebene	Basis	Basis	elaboriert
Getrennte Übersetzung	ja	ja	ja
Postulative Realzeit	ja	nein	nein
Deskriptive Realzeit	Basis	elaboriert	nein
Parallelisierungsalgorithmen	nein	nein	ja
Getaktete Systeme	implizit	explizit	nein
Funktionale Programmierung	Basis	Basis	nein
Abstrakte Datentypen	nein	Basis	Basis
Speicherplatzvergabe	statisch	statisch	dynamisch
Interruptbehandlung	ja	ja	nein
Organisationsmodell (Haupt-)	Ereignis-steuerung	Ereignis-steuerung	Prozess-komm.

2. CAP/SIL - Eine Systemimplentierungssprache für Multi-Mikroprozessor-Systeme

2.1 Das Modulkonzept

Entwurfskriterien für CAP/SIL waren:

1. Es soll besondere Sprachunterstützung für die Lösung von Synchronisationsproblemen bei der Betriebsmittelvergabe angeboten werden,
2. bei der Benutzung von Betriebsmitteln soll Nebenläufigkeit explizit formulierbar sein, und
3. implizite Nebenläufigkeit bei der Benutzung von Betriebsmitteln soll durch Parallelisierungsalgorithmen erkennbar und realisierbar sein.

Um diese Kriterien zu erfüllen, wurde in die Wirtssprache PASCAL ein Modulkonzept in Form eines Modultyps integriert, das die Definition getrennt übersetzbarer Betriebsmitteltypen erlaubt. (Zur Laufzeit eines CAP/SIL-Programms können beliebig viele Betriebsmittel aus einem Betriebsmitteltyp inkarniert werden.)

Ein Modultyp besteht aus
[-Parameterliste]
-Operationsliste
[-External-Liste]
-Kontrollausdruck
[-Deklarationsteil für lokale Datenstrukturen]
-Operatoren
([ ] bezeichnet optionale Komponenten)

Im Kontrollausdruck sind die Synchronisationsregeln für die Benutzung der Moduloperatoren formuliert. Die Nutzung der impliziten Nebenläufigkeit kann nun durch Parallelisierungsalgorithmen erreicht werden, die alle Aufrufe von Betriebsmitteloperatoren unter Beachtung von Datenabhängigkeiten parallelisieren. Notwendige Synchronisationen werden innerhalb des Betriebsmittels vorgenommen.

2.2 Laufzeit-Organisation

Die CAP/SIL-Module und ihre Kontrollausdrücke machen eine

prozedurorientierte Struktur des zu implementierenden Betriebssystems notwendig [4]. Dabei wird ein Betriebssystemkern und/oder ein Laufzeitsystem benötigt, mit dem u.a. folgende Sprachmittel unterstützt werden:

1. Prozeduren und Funktionen (Operationen)
   Eine Operation besteht aus einem Algorithmus, lokalen Funktionen und Parametern. Sie arbeitet immer innerhalb der Umgebung eines Programms oder Moduls (Objekt) und hat Zugriff auf alle Daten und Operationen, die innerhalb des Objekts deklariert sind. Aufrufe von Operationen können synchron oder asynchron ausgeführt werden. Synchrone Operationsaufrufe entsprechen normalen PASCAL-Operationsaufrufen. Ein asynchroner Operationsaufruf liegt vor, wenn eine Export-Operation eines anderen Objekts aufgerufen wird. In diesem Fall wird die Abarbeitung der Operation als eigenständiger Prozeß gestartet. Der Prozeß hat Zugriff zu den lokalen Datenstrukturen des gerufenen Objekts, besitzt jedoch einen eigenen Stack für die Abarbeitung der Operation.

2. Module (Objekte)
   Ein Modul besteht aus einer Anzahl von Operationen, lokalen Datenstrukturen, Parametern und einem Kontrollausdruck für die Synchronisation der Export-Operationen. Als Inkarnierungsparameter können u.a. Modultypen für die Erzeugung lokaler Betriebsmittel und inkarnierte Objekte als globale Betriebsmittel übergeben werden.

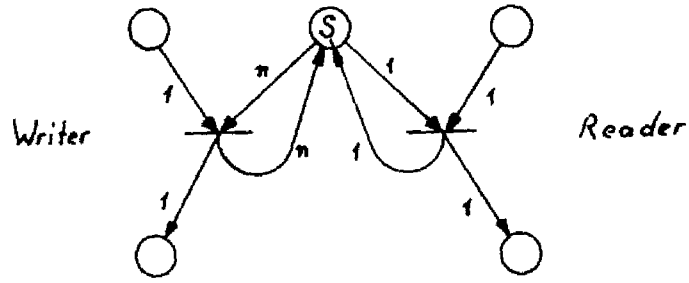
Module-Inkarnationen werden mit Hilfe einer Standardfunktion (activate) erzeugt. Diese legt eine neue Umgebung für die Aufnahme der lokalen Moduldaten an und initialisiert den Kontrollausdruck.

2.3 Der Kontrollausdruck

Der Kontrollausdruck beschreibt ein Transitionsnetz, in dem für jede Modul-Exportoperation eine Transition vorhanden sein muß. Die Synchronisationsbedingungen für die Exportoperationen entsprechen den Stellenbelegungen. Das Schalten einer Transition initiiert die Abarbeitung der zugehörigen Operation. Bei der Auswertung des Transitionsnetzes wird davon ausgegangen, daß nie

zwei Transitionen simultan schalten können (Kolateralität). Dies bedeutet keine Einschränkung der Nebenläufigkeit, da das Transitionsnetz lokal auf einem Prozessor ausgewertet werden muß. Die Auswertung beginnt, wenn die linkeste Eingangsstelle einer Transition markiert wird. Werden die linkesten Eingangsstellen mehrerer Transitionen simultan markiert, so wird das Transitionsnetz nacheinander für alle Markierungen ausgewertet.

Bei der Formulierung von Synchronisationsregeln mit möglicher Nebenläufigkeit mittels Petri-Netzen bemerkt man sehr schnell, daß eine obere Grenze für den Grad der Nebenläufigkeit angegeben werden muß. Will man z.B. das Reader/Writer-Problem lösen, so muß man die Anzahl der möglichen nebenläufigen Reader-Aktivierungen beschränken: [6]



Die Synchronisationsstelle S wird mit n Marken initialisiert. Die Aktivierung des Writer benötigt n Marken von S, die Aktivierung eines Reader je 1 Marke von S, so daß entweder 1 Writer oder maximal n Reader nebenläufig aktiv sein können.

Das Problem wird noch schwieriger, wenn man versucht Prioritäten auszudrücken. Dabei wird es dann nötig, das Fehlen von Stellen-Markierungen zu testen.

Um die Bestimmung einer oberen Grenze bei der Anzahl nebenläufiger Aktivitäten zu vermeiden und die Formulierung von Prioritäten zu ermöglichen, können Stellendeklarationen in CAP/SIL ein Markierungsprädikat enthalten, das Zählerwerte benutzt [5]. Liefert die Auswertung des Prädikats den Wert true, so ist die Stelle markiert, andernfalls nicht.

Vereinfachte Syntax:

<Stellendeklaration> ::= <Stelle>[:<Markierungsprädikat>]
<Markierungsprädikat> ::= <Zähler><Vergleichsoperator><Konstante>
<Stelle> ::= <identifier>
<Zähler> ::= <identifier>
<Vergleichsoperator> ::= <|> = | ≠ |< ≤ |> ≥
<Konstante> ::= natürliche Zahl

Außer den deklarierten Stellen gibt es zur jeder Operation (Transition) eine implizite Synchronisationsstelle, die beim Operationsaufruf markiert wird. Ferner gibt es zu jeder Operation einen Zähler, der bei Operationsaktivierung inkrementiert und bei Operationsdeaktivierung dekrementiert wird. Die Operationsaufrufstelle wird mit $ Operationsname und der Zähler mit # Operationsname eindeutig festgelegt.

Die Vernetzung der Stellen und Transitionen wird in einer Anzahl von Kontrollanweisungen beschrieben. Vereinfachte Syntax:

<Kontrollanweisung> ::=
on <Stellenausdruck> <Operation> set <Stellenliste>
<Stellenausdruck> ::= <Stelle>|
<Stelle> andp <Stellenausdruck>|
<Stelle> orp <Stellenausdruck>|
(<Stellenausdruck>)
<Operation> ::= :<identifier>:|
::
<Zählerausdruck> ::= for [<Konstante>|] <Zähler> do
<Stellenliste> ::= <Stelle> [,<Stellenliste>]
<Zähler> [,<Stellenliste>]

Die Semantik der Kontrollausdrücke soll an einem komplexen Beispiel, dem Reader/Writer-Problem, erläutert werden. Scheduling-Strategie soll sein: Ein neuer Reader darf nur dann weiterarbeiten, wenn kein Writer wartet und alle Reader werden aktiviert, wenn ein Writer die Arbeit beendet. Eine Änderung der Scheduling-Strategie ist sehr einfach durch Umordnung der Kontrollanweisungen und/oder Veränderung der Eingangsstellen möglich.

```
type Buffer = module
operation read, write
Place readdone; writedone;
      nowriter : # write = Ø
      noreader : # read  = Ø
      nowritewait : # writewait = Ø
      noreadwait : # readwait  =  Ø
      waitingread : # readwait  > Ø
      waitingwrite : # writewait > Ø
on $ read andp nowriter andp nowritewait : read : set readdone;
on $ read : : set # readwait,
on $ write andp nowriter andp noreader : write : set writedone;
on $ write : : set # writewait;
on readdone andp noreader andp waitingwriter:For 1# writewait do
                                        write : set writedone
on writedone andp waiting reader : for # readwait do
                                        read : set readdone;
on writedone andp waitingwriter : for 1 # writewait do
                                        write : set writedone
   :
modulend
```

$ read und $ write sind die impliziten Operations-Synchronisationsstellen und # read, # write die Operationszähler.

Die frei angegebenen Zähler (# readwait und # writewait) können nur in Kontrollanweisungen manipuliert werden: Inkrementierung in einer <Stellenliste> und Dekrementierung in einem <Zählerausdruck>. Kontrollausdrücke, die die selbe linkeste Stelle besitzen, werden in Reihenfolge der Aufschreibung abgearbeitet.

Im Anhang 1 wird der Kontrollausdruck als erweitetes Petri-Netz dargestellt und in Anhang 2 wird eine äquivalente Darstellung als Monitor gezeigt.

2.4 Schlußbemerkungen

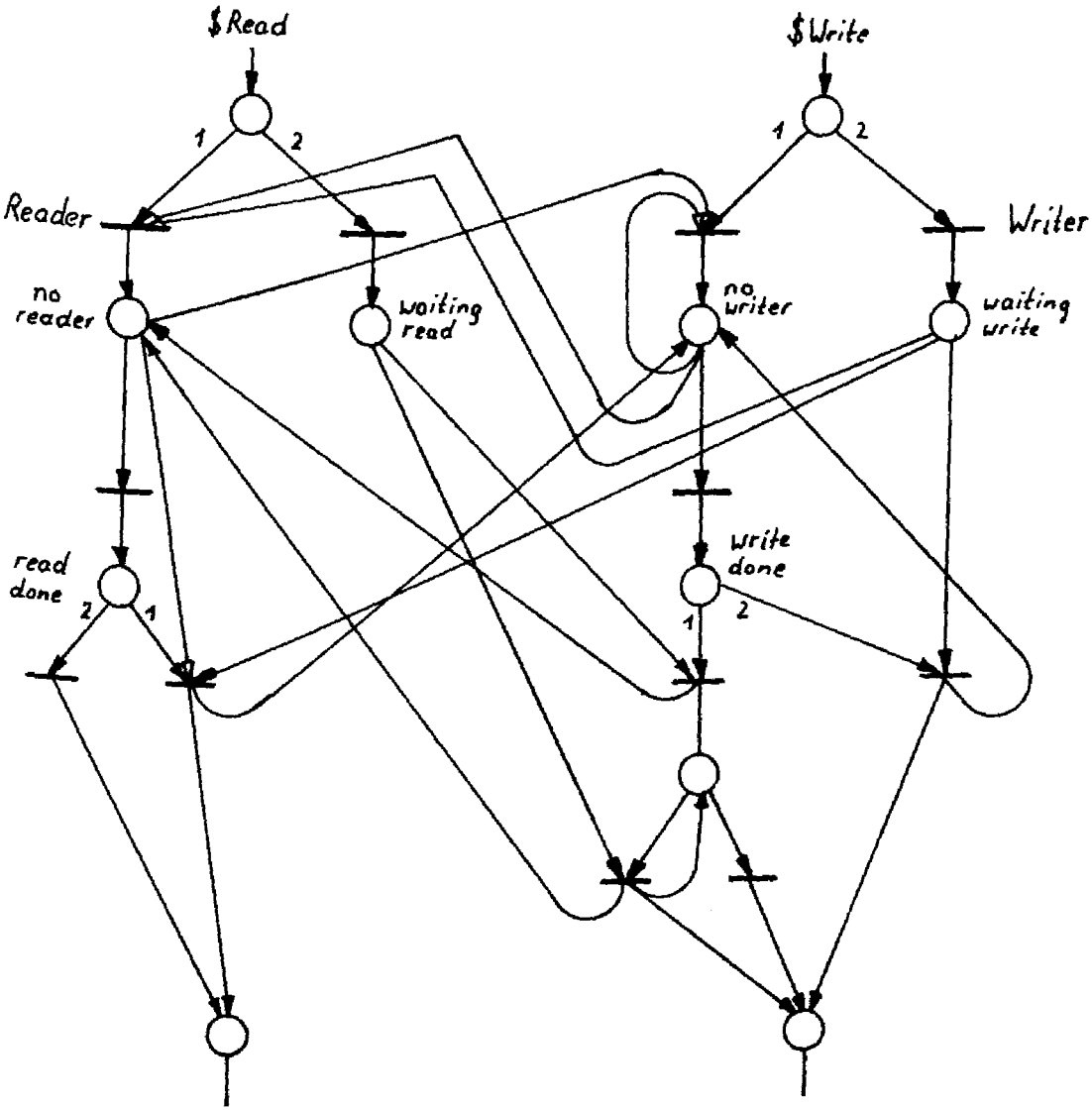
Der wesentlichste Unterschied zwischen CAP/SIL und Sprachen wie Concurrent PASCAL, MODULA, Path PASCAL usw. besteht darin, daß durch ein CAP/SIL-Programm oder -Modul ein dynamisches Prozeß-System beschrieben wird, während bei den oben genannten Sprachen

statische Prozeß-Systeme definiert werden. Ferner enthält CAP/SIL Kontrollstrukturen für die Formulierung nebenläufiger Anweisungen, durch die die Parameterversorgung für nebenläufige Operationsaufrufe erreicht wird.

Stand der Arbeiten: Ende 1980 wurden die Implementation eines CAP/SIL-Compilers und der zugehörigen Parallelisierungsalgorithmen beendet.

Anhang 1

Petri-Netz Semantik des Beispiel-Kontrollausdrucks



Erweiterte Petri-Netz-Darstellung des Reader/Writer Beispiels

—▷ bezeichnet Test auf Ø Marken (Inhibitor Eingang)

—▶ zieht 1 Marke von der Stelle ab

gibt Priorität der Ausgänge bei Konflikten an

Bzgl. Petri-Netz Darstellungen mit Inhibitor-Arcs und Prioritäten siehe [7].

Anhang 2

Monitor-Semantik des Beispiel-Kontrollausdrucks
Vergleiche [8], [9]).

```
monitor Scheduler;
var readreg, writereg : condition;
    # read, # write, # readwait, # writewait : Ø ... maxint,
procedure $read ;
begin if (⌀# write=Ø) and (⌀# writewait=Ø))
      then # read := # read+1
      else begin,
           # readwait := # readwait+1;
           readreg.wait
           end
end (*$ read *);
procedure $write ;
begin if (⌀# write=Ø) and (⌀# read=Ø))
      then # write := 1
      else begin
           # writewait := # writewait+1;
           writegeg.wait;
           end
end (* $ write*);
procedure readdone;
begin # read := read-1;
      if (⌀# read=Ø) and (≠# writewait > Ø))
      then begin
           # writewait := writewait-1;
           writereg.signal;
           end
end (*readdone*)
procedure writedone;
```

```
begin # write := # write-1;
      if # readwait > ø
      then while # readwait > ø do begin
                                      # readwait := # readwait-1;
                                      readreg.signal;
                                      end
      else if # writewait > ø
           then begin
                # writewait := # writewait-1
                writereg.signal
                end
end (*writedone*)
begin
# read := ø;
# write := ø;
# readwait := ø;
# writewait := ø;
end (*monitor scheduler*)
```

Die Signal-Operation muß so implementiert sein, daß der signalisierende Prozeß aktiv bleibt.

Skizze der Monitoraufrufe:

```
scheduler.$read                    scheduler.$write
        :                                  :
code der Operation                 code der Operation
      read                               write
        :                                  :
scheduler.readdone                 scheduler.writedone
```

3. Literatur

[1] Rammig, F. J.: DIGITEST II: An integrated Structural and Behavioral Language. Proceedings of 1975 International Symposium on Computer Hardware Description Languages and their Applications. New York 3. 9. - 5. 9. 1975

[2] Rammig, F. J.: An Introduction to the Concurrent Algorithmic Programming Language CAP or Looking at CAP with the Revised Iroman's Eyes. Forschungsbericht Nr. 80 der Abt.

Informatik der Universität Dortmund (1979)

[3] Rammig, F. J.: Structured Parallel Programming with a Highly Concurrent Programming Language. Proceedings of AICA '80 Bologna 29. - 31. Okt. 1980

[4] Lauer, H. C.; Needham, R. M.: On the Duality of Operating Systems Structures. Proc. Second International Symposium on Operating Systems, IRIA, Oct. 1978, reprinted in Operating Systems Review, Vol. 13, No. 2 (1979), pp. 3 - 19

[5] Gerber, A. J.: Process Synchronisation by Counter Variables. Operating Systems Review, Vol. 11, No. 4 (1977), pp. 6 - 17

[6] Kosaraju, S. R.: Limitations of Dijkstra's semaphore primitives and Petri nets. Tech. Rep. 25, Johns Hopkins Univ., Baltimore, Md., May 1973, 5 pp.; also in Operating Systems Review, Vol. 7, No. 4 (1973) pp. 122 - 126

[7] Hack, M.: Petri Net Languages. MIT, Project MAC, Computation Structures Group Memo 124 (1975)

[8] Hoare, C. A. R.: Monitors: an Operating System Structuring Concept. Communications of the ACM, Vol. 17, No. 10 (1974)

[9] Brinch-Hansen, P.: Operating Systems Principles. Prentice Hall, 1973

Dr. Franz J. Rammig
Universität Dortmund
Abteilung Informatik
Postfach 50 05 00
4600 Dortmund 50

Horst Röder
Universität Dortmund
Abteilung Informatik
Postfach 50 05 00
4600 Dortmund 50