Darstellung von Monitoren durch Petri Netze

Ralf Steinmetz*

St - 248

Stichworte: *Monitor, Petri Netz, Semaphor, parallele Programmierung, Synchronisation, Kommunikation*

Zusammenfassung: *Die Synchronisations- und Kommunikationskonzepte nehmen bei der Programmierung paralleler Prozesse eine zentrale Stellung ein. Semaphore und Monitore sind zwei bekannte Mechanismen dieser Art. In diesem Artikel wird beschrieben, wie man Monitore durch Petri Netze mit Hilfe des Semaphorkonzepts darstellen kann.*

Damit lassen sich alle von den Petri Netzen her bekannte graphische Darstellungsmöglichkeiten und Analyseverfahren auf die in Petri Netze transformierten Monitore anwenden. Dies kann für die Programmierung paralleler Prozesse sowohl zum Verständnis des implementierten Programms beitragen als auch bei der Fehlersuche auf Grund falsch programmierter Synchronisation (mit Monitoren) wesentlich helfen.

Representation of Monitors by Petri Nets

Key-words: *Monitor, Petri Net, Semaphore, Concurrent Programming, Synchronisation, Communication*

Abstract: *Interprocess communication and synchronisation is a central topic of concurrent programming. The semaphore- and the monitor-concept are well known. In this article the transformation of monitors into petri nets via the semaphore-concept is described.*

This has the advantage, that the whole theory of petri nets can be used to analyse the implemented monitors. So the programmer has the opportunity to use the descriptive power of petri nets to understand implemented concurrent programs. The Petri nets of the monitors can also be used to support the programmer debugging errors based on erroneous programmed synchronisation with monitors.

* Dipl.-Ing. Ralf Steinmetz, Institut für Übertragungstechnik, Technische Hochschule Darmstadt, Merckstr. 25, D-6100 Darmstadt

1 Einleitung

Durch die sinkenden Hardwarekosten und den steigenden Einsatz von Mehrprozessorsystemen, unter anderem als Realzeitsysteme, müssen auch die höheren Programmiersprachen den Anforderungen dieser komplexeren Umgebung angepaßt werden. Dies bedeutet für die Sprachen neben der Bereitstellung von Konstrukten zur Modularisierung (zum Beispiel Abstrakte Datentypen in MODULA-2 [14]), auch die Implementierung von Sprachelementen zur Programmierung paralleler Prozesse (zum Beispiel in MODULA-2 und ADA [2]).

Das zentrale Problem beim Umgang mit parallelen Prozessen ist das zu verwendende Konzept zur Synchronisation und Kommunikation [1], [3]. Es bieten sich hierfür verschiedene Methoden an (zum Beispiel: Busy Waiting, Semaphore, Monitore, Conditional Critical Region, Path Expresions, Austausch von Botschaften). Im folgenden sollen nur Semaphore [7] und das in MODULA-2, Concurrent Pascal [5] und Pascal Plus [6] realisierte Monitorkonzept [4], [8] betrachtet werden.

Im praktischen Einsatz erweist sich das Semaphorkonzept als sehr vielseitig, jedoch ist die Programmierung mit Semaphoren sehr fehlerträchtig und unübersichtlich [1], [10]. Deshalb wird auch von keiner höheren Programmiersprache dem Anwender ausschließlich Semaphore als Sprachelemente zur Synchronisation zur Verfügung gestellt. Monitore sind wesentlich benutzerfreundlicher und haben deshalb Eingang in mehrere Programmiersprachen gefunden. Durch die nicht sequentielle Programmierung ist es für den Programmierer sehr schwer, den korrekten Ablauf eines aus mehreren parallelen Prozessen bestehenden Programms festzustellen. Auch eine Fehlersuche auf Grund falscher Synchronisation oder Kommunikation ist bei derartigen Programmen extrem arbeits- und damit auch kostenintensiv.

Petri Netze eignen sich durch ihre inhärente Parallelität besonders gut zur Darstellung und Analyse paralleler Probleme [11], [13]. Wenn sich einzelne Konstrukte, die die Synchronisation und Kommunikation des Monitorkonzepts bilden, in Petri Netze überführen lassen, so können die Analysemechanismen von Petri Netzen zur Auswertung der Kommunikation und Synchronisation mit Monitoren verwendet werden.

0013-5704/84/8 0314–09 . $ 03.00/0 © Friedr. Vieweg & Sohn Verlagsgesellschaft mbH

Daß sich Monitore durch Semaphore darstellen [8] und Semaphore in Petri Netze überführen lassen [12] ist bereits bekannt. Dies läßt auch den direkten Schritt von Monitoren zu Petri Netzen vermuten, der in diesem Artikel beschrieben und an einem Beispiel demonstriert werden soll.

Im Anschluß an die Erläuterung des verwendeten Petri Netz-Modells wird das Semaphorkonzept und dessen Transformation in ein Petri Netz beschrieben. Der vierte Abschnitt beginnt mit der Definition des Monitorkonzepts. Dann wird die Darstellung der Monitorelemente durch Semaphore, parallel zu deren Transformation, in ein Petri Netz vorgestellt. Die so entstandenen Petri Netze können unter den angegebenen Bedingungen, wie im letzten Teil des vierten Abschnitts aufgezeigt, wesentlich vereinfacht werden. Im fünften Abschnitt wird an einem Beispiel das Zusammenwirken der einzelnen Teilnetze von den einzelnen Monitorkomponenten gezeigt.

2 Das Petri Netz

Das Umfeld der Prozeßsynchronisation und der Prozeßkommunikation besteht aus etlichen parallel und asynchron ablaufenden Vorgängen. Zur Beschreibung und Analyse derartiger Systeme eignen sich besonders gut Petri Netze [1], [13]. In der Literatur existieren viele verschiedene Petri Netz-Typen (so zum Beispiel Bedingungs/Ereignis-, Stellen/Transitionen- oder Prädikats/Ereignis-Netze [11]). Für die Bearbeitung der Problematik dieses Artikels wurde das im folgenden beschriebene Petri Netz-Modell, basierend auf dem Stellen/Transitionen-Netz, verwendet.

Definition: Ein Petri Netz (PN) ist ein 8-tupel:
$PN = (S, T, U, V, I, g, k, m_0)$

Dabei ist:

S: Stellen; die nicht leere, endliche Menge aller Stellen eines Systems, passive Netzknoten.
$S = \{s_1, s_2, ..\}$ Symbol: Kreis

T: Transitionen; die nicht leere, endliche Menge aller Übergänge, die in einem Netz stattfinden können, aktive Netzknoten.
$T = \{t_1, t_2, ..\}$ **Symbol: Stab**

U: Kanten (Relation, Flußrelation); die endliche Menge aller gerichteten Kanten vom Typ 1 von s_i nach t_j $(i, j \in \mathbb{N})$.
$U \subseteq \{S \times T\}$ Symbol: Pfeil

V: Kanten (Relation, Flußrelation); die endliche Menge aller gerichteten Kanten von t_i nach s_j $(i, j \in \mathbb{N})$ (alle Kanten sind vom Typ 1).
$V \subseteq \{T \times S\}$ Symbol: Pfeil

I: Kanten (Inhibitor arcs, Relation, Flußrelation); die endliche Menge aller gerichteten Kanten vom Typ 2 von s_i nach t_j $(i, j \in \mathbb{N})$.
$I \subseteq \{S \times T\}$ Symbol: Strecke mit Kreis an einem Ende

g: Gewicht (Funktion, Eigenschaft einer Kante vom Typ 1); jeder Kante aus den Mengen U und V wird eine positive ganze Zahl oder unendlich zugeordnet.
$g: U \to \{\mathbb{N}, \infty\} \cup V \to \{\mathbb{N}, \infty\}$ Abk.: g(s, t) oder g(t, s)

k: Kapazität (Funktion, Eigenschaft einer Stelle); jeder Stelle wird eine positive ganze Zahl oder unendlich zugeordnet.
$k: S \to \{\mathbb{N}, \infty\}$ Abk.: k(s)

m: Markierung, (Vektor oder Funktion); jeder Stelle eine positive Anzahl von Marken (Kerne, Symbol: Punkt) zugeordnet.
$m: S \to \mathbb{N}$ Abk.: m(s)
m_0 ist die Anfangsmarkierung

Definition: **Der Vorbereich einer Transistion** ist die Menge aller Stellen, die durch eine Kante aus den Mengen U oder I mit dieser Transition verbunden ist.
$inp(t) = \{s \in S \mid (s, t) \in U \cup I\}$ Abk.: inp(t)

Definition: Der **Vorbereich 1 einer Transition** ist die Menge aller Stellen, die durch eine Kante aus der Menge U mit dieser Transition verbunden ist.
Abk.: $inp_1(t)$

Definition: Der **Vorbereich 2 einer Transition** ist die Menge aller Stellen, die durch eine Kante aus der Menge I mit dieser Transition verbunden ist.
Abk.: $inp_2(t)$

Definition: Der **Nachbereich einer Transition** ist die Menge aller Stellen, die durch eine Kante aus der Menge V mit dieser Transition verbunden ist.
$out(t) = \{s \in S \mid (t, s) \in V\}$ **Abk.: out(t)**

Definition: Eine Transition ist **aktiviert**, wenn
1. jede Stelle des Vorbereichs 1 mindestens soviele Marken enthält, wie das Gewicht der Kante zwischen dieser Transition und Stelle beträgt,
2. jede Stelle des Vorbereichs 2 keine Marke enthält und
3. jede Stelle des Nachbereichs noch soviele Marken aufnehmen kann, daß die Summe der aktuellen Markierung und des Gewichts der Kante zwischen der betrachteten Stelle und Transition die Kapazität der Stelle nicht übersteigt.

Definition: Eine Transition kann **schalten**, wenn die Transition aktiviert ist. Die Markierung m geht in die Markierung m′ über.
$inp_1(t): m'(s) = m(s) (n - (s, t)$
$out(t): m'(s) = m(s) + gt, s)$
$sonst: m'(s) = m(s)$ **Abk.: $m\,[t) > m'$**

2.1 Beispiel

An folgendem kleinen Beispiel sollen die eingeführten Begriffe verdeutlicht werden (siehe Bild 1).

Stellen: $S = \{s_1, s_2, s_3, s_4, s_5\}$
Transitionen: $T = \{t_1, t_2\}$
Kantentyp 1 von s_i nach t_j: $U = \{(s_2, t_1), (s_3, t_2), (s_4, t_1)\}$
Kanten von t_i nach s_j: $V = \{(t_1, s_3), (t_2, s_4), (t_2, s_5)\}$

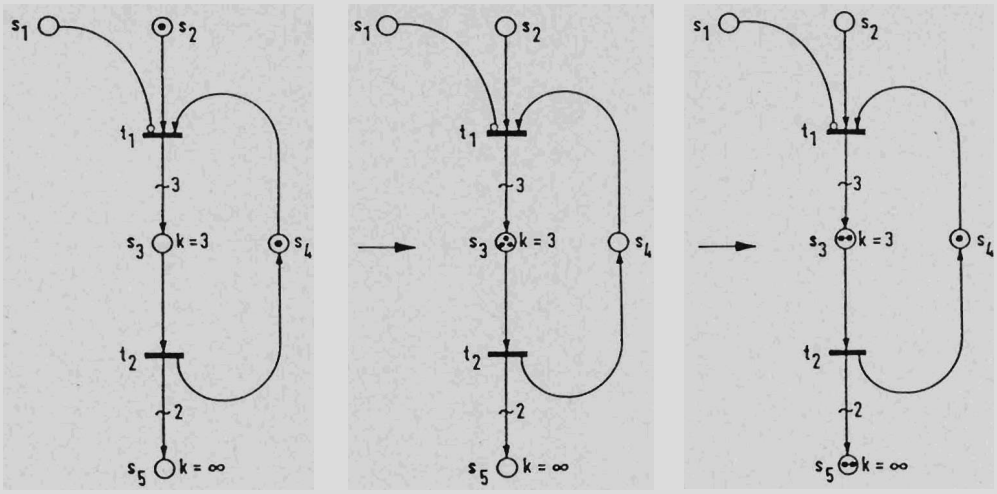


Bild 1
Beispiel eines Petri Netzes

Kantentyp 2 von s_i nach t_j: I = $\{(s_1, t_1)\}$
Gewichte: $g(t_1, s_3) = 3$, $g(t_2, s_5) = 2$, sonst $g = 1$
Kapazität: $k(s_3) = 3$, $k(s_5)$ = unendlich, sonst $k = 1$

Außerdem gilt zum Beispiel: $inp(t_1) = \{s_1, s_2\}$,
$inp_1(t_1) = \{s_2\}$ und $inp_2(t_1) = \{s_1\}$.
Markierung: $m_0(s_1, s_2, s_3, s_4, s_5) = (0, 1, 0, 1, 0)$
Die Transition t_1 ist unter m_0 aktiviert und kann schalten (siehe Bild 1), $m_0\ [t_1 > m_1$.

$$m_1(s_1, s_2, s_3, s_4, s_5) = (0, 0, 3, 0, 0)$$

Nun ist t_2 aktiviert und kann schalten: $m_1\ [t_2 > m_2$

$$m_2(s_1, s_2, s_3, s_4, s_5) = (0, 0, 2, 1, 2)$$

2.2 Anmerkungen

Läßt man als Vorbereich einer Transition nur Stellen mit endlicher Kapazität zu, so ist die Einführung des Kantentyps 2 (I) nur eine verkürzte Schreibweise (siehe Bild 2).

Zur Vereinfachung seien auch Schlingen erlaubt. Diese lassen sich, wie in Bild 3 gezeigt, problemlos in ein Netz ohne Schlingen (zur eindeutigen Darstellung der Inzidenzmatrix für Netz-Analysen) überführen.

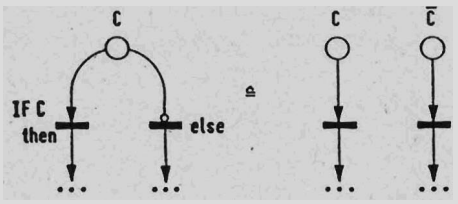


Bild 2 Netz mit Kantentyp 2 dargestellt durch ein Netz ohne Kantentyp 2

Bild 3 Die Schlinge

3 Das Semaphorkonzept und seine Darstellung durch Petri Netze

Das Semaphorkonzept [7] ist eine Synchronisationsprimitive, die auf gemeinsamen Variablen (shared variables) basiert. Die Semaphorvariable Sema ist eine Natürliche Zahl auf der zwei Operationen definiert sind:

P(Sema) = wait(Sema) : Ein Prozeß führt dann folgende Operationen aus: Falls

- Sema > 0 ist, so wird Sema um 1 erniedrigt (Sema:= Sema-1) und der Prozeß läuft weiter;
- Sema = 0 ist, so wird der Prozeß blockiert und muß warten bis Sema > 0 wird, um anschließend Sema zu erniedrigen und weiterzulaufen.

V(Sema) = signal(Sema) : Ein Prozeß führt dann folgende Operationen aus: Falls

- Sema > 0 ist, so wird Sema um 1 erhöht (Sema:= Sema+1) und der Prozeß läuft weiter;
- Sema = 0 ist, so wird Sema um 1 erhöht, der die Operation V(Sema) ausführende Prozeß läuft weiter. Falls einer oder mehrere Prozesse auf Sema > 0 warten, so wird einer wieder aktiviert.

Anmerkungen:

Die durch P(Sema) und V(Sema) ausgeführten Operationen sind unteilbar. Alle Semaphorvariablen müssen initialisiert werden. Auf eine Semaphorvariable kann immer nur eine Operation zur gleichen Zeit angewendet werden. Kann eine Semaphorvariable nur die Werte 0 und 1 annehmen, so nennt man sie binäre Semaphorvariable.

Semaphore stellen ein sehr einfaches und doch wirksames Konzept zur Prozeßsynchronisation dar. Fairness wird durch den Anwender erreicht: Wenn V(Sema) unendlich oft ausgeführt wird, wird kein Prozeß, der wegen einer Operation P(Sema) blockiert ist, unendlich lange warten [1] (Die Menge der Prozesse, die P(Sema) bein-

halten, und die Menge aller P(Sema) sei endlich). Manche Implementierungen sehen deshalb eine FIFO Warteschlange für blockierte Prozesse vor. Auch ohne FIFO-Implementierung kann sehr leicht ein Deadlock, wie im folgenden Beispiel gezeigt, spezifiziert werden:

PROZESS 1:	PROZESS 2:
$P(Sema_1)$;	$P(Sema_2)$;
$P(Sema_2)$;	$P(Sema_1)$;

Der Prozeß 1 soll die Anweisung $P(Sema_1)$ ausführen, anschließend soll der Prozeß 2 die Anweisung $P(Sema_2)$ ausführen. Zwischen den beiden P-Operationen der jeweiligen Prozesse sollen sich keine $V(Sema_1)$- und $V(Sema_2)$-Operationen befinden. Also haben sich diese beiden Prozesse gegenseitig blockiert und das System ist verklemmt.

Das Semaphorkonzept kann sehr einfach in ein Petri Netz überführt werden [12]. Die Semaphorvariable Sema wird durch eine Stelle S dargestellt. Die Operation P(Sema) wird durch eine Transition (wait(S)) dargestellt, eine der Eingangsstellen ist das Semaphor Sema (Stelle S) und die andere der vorherige Zustand des aufrufenden Prozesses (siehe Bild 4).

V(Sema) wird durch eine Transition (signal(S)) dargestellt, von der eine Ausgangsstelle S und die andere den Zustand des Prozesses nach der Aktivierung darstellt (siehe Bild 5).

Zum besseren Verständnis wird das folgende Beispiel als Petri Netz modelliert (siehe Bild 6):

```
PROGRAM mutex
PROCESS P1  (*Definition des ersten Prozesses*)
   BEGIN
   DO forever
     P(Sema);
     (*Critical Section 1: bei Ausführung dieser Operation soll nur dieser Prozeß P1 aktiv sein*)
     V(Sema);
     (*Rest*)
   OD
   END;
PROCESS P2 (* Definition des zweiten Prozesses*)
   BEGIN
   DO forever
     P(Sema);
     (* Critical Section 2: bei Ausführung dieser Operation soll nur dieser Prozeß P2 aktiv sein*)
     V(Sema);
     (* Rest *)
   OD
   END;
```

Bild 4 Semaphoroperation 'P(Sema)'

Bild 5 Semaphoroperation 'V(Sema)'

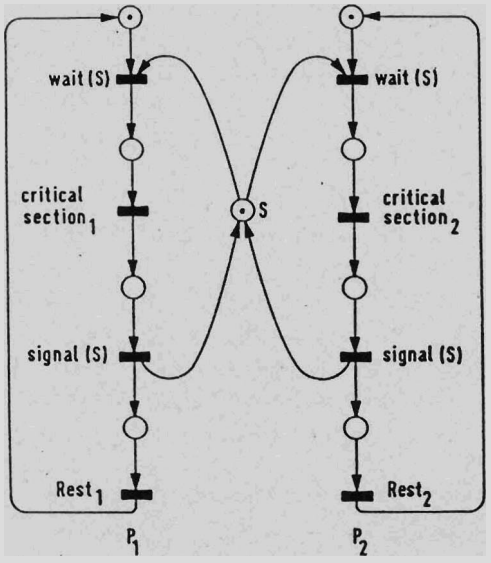


Bild 6 Beispiel: mutex

```
COBEGIN   (* Programm mutex:                          *)
  P1;P2   (* Aufruf der beiden Prozesse, die
             parallel                                 *)
COEND     (* ablaufen sollen                         *)
```

4 Das Monitorkonzept

Obwohl Semaphore eine sehr wirksame Methode zur Synchronisation paralleler Prozesse darstellen, ist diese Synchronisationsprimitive jedoch sehr unstrukturiert, und es ist schwer mit Semaphoren übersichtlich und fehlerfrei zu programmieren [10]. Eine Alternative bietet das von C. Hoare und P. B. Hansen [4], [8] vorgeschlagene Monitorkonzept.

Ein Monitor besteht aus

- einer endlichen Menge von permanenten Variablen,
- einer endlichen Menge von Prozeduren, die auf diese Variablen zugreifen können und
- einer Initialisierung der Variablen.

Die Variablen des Monitors sollen lokal sein, die Prozeduren sollen nicht auf andere globale Variablen zugreifen. Die Monitor-Prozeduren sind global zugänglich, d.h. sie können von allen Prozessen aus aufgerufen werden. Würden nun mehrere Prozesse gleichzeitig über die Prozeduren eines Monitors auf dieselben Variablen zugreifen, so wäre hier bestimmt keine Synchronisation gewährleistet. D.h. es darf sich immer nur ein Prozeß gleich-

zeitig in einem Monitor befinden. Erst wenn dieser Prozeß den Monitor verläßt, darf der nächste Prozeß eine Prozedur dieses Monitors ausführen.

Damit sich zwei Prozesse synchronisieren können, werden in einem Monitor zwei Operationen ('Wait' und 'Signal') bereitgestellt:

Führt ein Prozeß eine 'Wait'-Anweisung aus, so wird er selber blockiert und verläßt den Monitor.

Falls ein Prozeß P_1 durch eine ‚Wait'-Anweisung blockiert wurde und anschließend ein anderer Prozeß P_2 eine 'Signal'-Anweisung ausführt, so wird P_1 reaktiviert und P_2 blockiert. P_2 wird reaktiviert, sobald der Monitor freigegeben ist und hat dann Vorrang vor anderen Prozessen, die in den Monitor eintreten wollen. Wurde kein Prozeß durch eine 'Wait'-Anweisung blockiert, dann hat die Anweisung 'Signal' keine Auswirkungen.

Mit Hilfe des Monitors können auch mehr als 2 Prozesse synchronisiert werden. Dabei muß beachtet werden, daß eventuell Prozeß P_1 und Prozeß P_2 unabhängig von Prozeß P_3 und Prozeß P_4 mit den Anweisungen 'Wait' und 'Signal' arbeiten. Deshalb werden diese Anweisungen immer auf bestimmte Bedingungsvariablen ('Condition') bezogen (Wait(Condition) und Signal(Condition)).

4.1 Darstellung des Monitors durch Semaphore und Petri Netze

Damit Prozesse Prozeduren eines Monitors aufrufen können, sich aber gegenseitig anschließen, wird für jeden Monitor eine im Monitor globale boolsche Semaphorvariable 'Sema' definiert.

Bei **Eintritt eines Prozesses in den Monitor** wird die Operation P(Sema) ausgeführt:

> Falls die Semaphorvariable Sema=0 ist, muß der Prozeß warten, ansonsten wird Sema auf Null gesetzt und der Prozeß führt die Prozedur im Monitor aus.

Das Semaphor 'Sema' wird durch die Stelle S mit der Kapazität k=1 dargestellt und mit einer Marke initialisiert. Bild 7 zeigt den Aufruf der Monitorprozedur.

Nach Ausführung der Operation 'Signal(Condition)' im Monitor muß dieser Prozeß blockiert werden, falls ein anderer auf die Bedingung 'Condition' wartet. Nach Freigabe des Monitors wird der blockierte Prozeß als nächster Prozeß in den Monitor eintreten. Deshalb wird ein binäres Semaphor 'Urgent' (Stelle U), und da mehrere Prozesse warten können ein allgemeines Semaphor 'UrgentCount' (Stelle U_c), eingeführt. Vor dem **Austritt eines Prozesses aus dem Monitor** wird nun durch 'Urgent' festgestellt, ob ein bestimmter anderer Prozeß, blockiert durch ein 'Signal(Condition)', aktiviert werden muß.

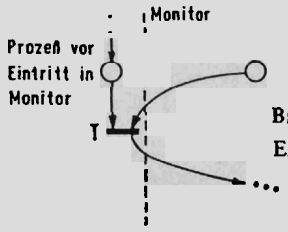
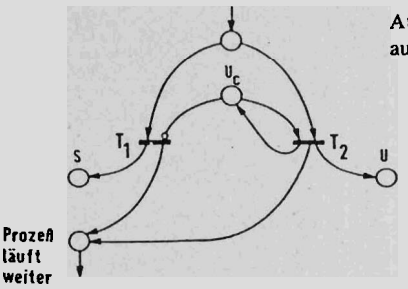


Bild 7
Eintritt eines Prozesses in den Monitor

Bild 8
Austritt eines Prozesses aus dem Monitor



Für das Semaphorkonzept bedeutet dies:

```
IF UrgentCount > 0 THEN V(Urgent)
                   ELSE V(Sema)
```

Die Stellen U und U_c haben keine Marken als Initialisierung. Bild 8 zeigt das entsprechende Petri Netz.

Für jede Bedingung 'Condition', auf welche die Operationen 'Signal(Condition)' oder 'Wait(Condition)' angewendet werden, muß ein Semaphor 'Cond' und ein Zähler 'CondCount' eingeführt werden.

Die Monitoroperation **'Wait (Condition)'** wird durch Semaphore wie folgt realisiert:

```
CondCount := CondCount + 1;
IF UrgentCount > 0 THEN V(Urgent)
                   ELSE V(Sema);
P(Cond);
CondCount := CondCount – 1;
```

In Abhängigkeit von der Stelle U_c (für UrgentCount) kann eine der beiden folgenden Transitionen (T_1 oder T_2) schalten und entweder eine Marke in die Stelle S (für Sema) oder in die Stelle U (für Urgent) setzen. In beiden Fällen wird die Markierung von C_c um eins erhöht. Der Prozeß kann erst reaktiviert werden, wenn in C (für Cond) eine Marke ist (siehe Bild 9). Die Stellen U und U_c werden mit 0 initialisiert.

Die Monitoroperation **'Signal(Condition)'** wird durch Semaphore nach dem folgenden Schema realisiert:

```
UrgentCount := UrgentCount + 1;
IF CondCount > 0 THEN BEGIN
                        V(Cond);
                        P(Urgent);
                      END;
UrgentCount := UrgentCount – 1;
```

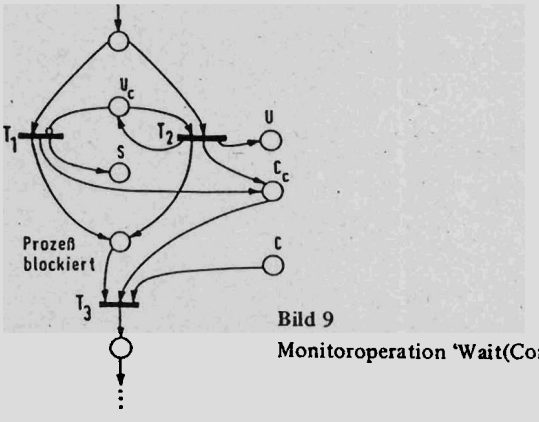


Bild 9
Monitoroperation 'Wait(Condition)'

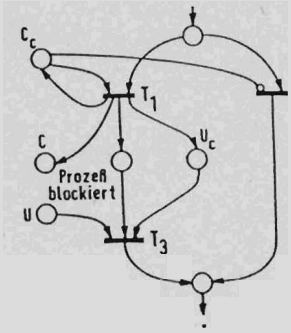


Bild 10
Monitoroperation 'Signal(Condition)'

Ein Prozeß P_1 führt die Anweisung 'Signal(Condition)' aus. Falls ein Prozeß P_2 auf die durch 'Signal(Condition)' beeinflußte Bedingungsvariable ('Condition') wartet, befindet sich in der Stelle C_c mindestens eine Marke. Somit ist, wie in Bild 10 gezeigt, nur die Transition T_1 aktiviert. Schaltet T_1, so kann der Prozeß P_2 durch Markieren der Stelle C (siehe auch Bild 9) fortgesetzt werden. Prozeß P_1 wird so lange blockiert, bis die Stelle U durch Austritt eines Prozesses aus dem Monitor oder durch Ausführung eines 'Wait(Condition)' markiert wird. Anschließend ist T_3 aktiviert. Die Monitoroperation 'Signal(Condition)' hat keine weiteren Auswirkungen, wenn kein Prozeß auf diese Bedingungsvariable wartet (die Stelle C_c ist dann nicht markiert und T_2 schaltet, siehe Bild 10).

4.2 Vereinfachungen

Die bisher betrachteten Netze enthalten für jede Monitoroperation mehrere Stellen und Transitionen. Unter bestimmten Bedingungen lassen sich diese Netze jedoch erheblich vereinfachen. Dasselbe gilt für die Darstellung durch Semaphore [8].

1. Bedingung: In einer Monitorprozedur kommt kein 'Wait(Condition)' und kein 'Signal(Condition)' vor.

1. Vereinfachungen: Während ein Prozeß diese Monitorprozedur ausführt, wird die Markierung der Stelle U_c nicht beeinflußt. Der Austritt aus dem Monitor kann deshalb ohne Berücksichtigung der Stelle U_c geschehen (siehe Bild 11).

2. Bedingung: 'Signal(Condition)' ist die letzte Anweisung in einer Monitorprozedur.

2. Vereinfachungen: 'Signal(Condition)' und der Austritt aus dem Monitor können zusammengefaßt werden. Durch Semaphore läßt sich dies wie folgt ausdrücken:

```
IF CondCount > 0 THEN V(Cond)
ELSE IF UrgentCount > 0 THEN V(Urgent)
ELSE V(Sema);
```

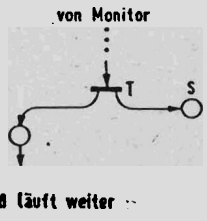


Bild 11
1. Vereinfachung bei Austritt aus dem Monitor

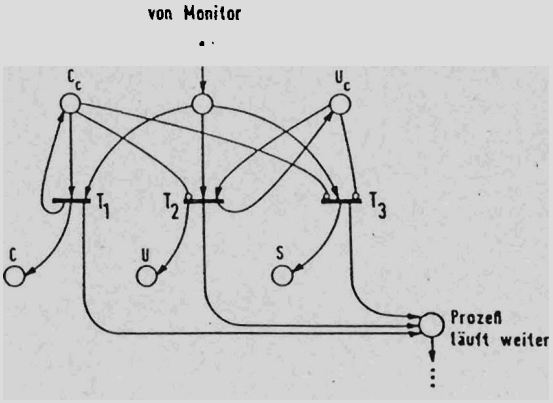


Bild 12 2. Vereinfachung bei Austritt aus dem Monitor und 'Signal(Condition)'

Im Petri Netz (Bild 12) werden diese Anweisungen paralellisiert. Das heißt, in Abhängigkeit von U_c und C_c werden in die Stellen S (über Transition T_3), U (über Transition T_2) oder C (über Transition T_1) eine Marke gesetzt.

3. Bedingung: 'Signal(Condition)' ist die letzte Anweisung in einer Monitorprozedur, und in der Prozedur kommen keine weiteren 'Signal' und 'Wait' Anweisungen vor.

3. Vereinfachungen: Zusätzlich zu den Vereinfachungen nach der 2. Bedingung (Zusammenfassung von 'Signal(Condition)' und Austritt aus dem Monitor) können die Vereinfachungen der 1. Bedingung (die Stelle U_c wurde nicht verändert) angewandt werden. Die Semaphoroperationen:

```
IF CondCount > 0 THEN V(Cond)
                 ELSE V(Sema);
```

werden im Petri Netz nach Bild 13 dargestellt.

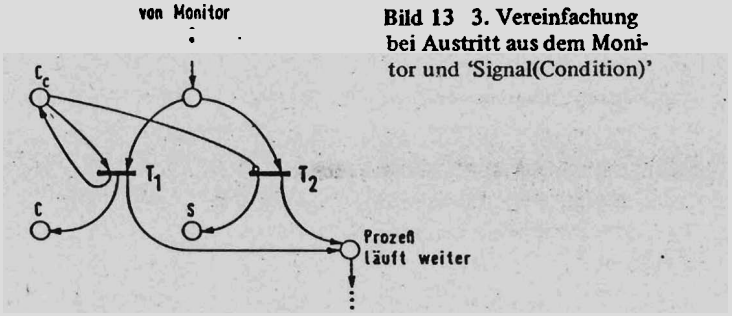


Bild 13 3. Vereinfachung bei Austritt aus dem Monitor und 'Signal(Condition)'

4. Bedingungen: Alle 'Signal(Condition)' eines Monitors sind die letzte Anweisung der jeweiligen Prozedur.

4. Vereinfachungen: Die Stellen U_c und U können weggelassen werden, weil der jeweilige Prozeß nach Ausführung von 'Signal(Condition)' den Monitor verläßt und nicht mehr als erster aktiviert werden muß. 'Signal(Condition)' wird nach Bild 13 realisiert und der Austritt aus dem Monitor nach Bild 11. Realisierung von 'Wait (condition)' mit Semaphoren:

```
IF CondCount > 0 THEN BEGIN
                        V(Sema);
                        P(Cond);
                   END;
```

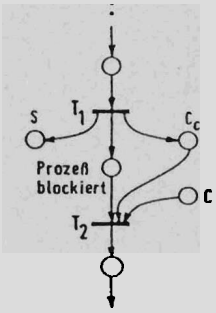


Bild 14
4. Vereinfachung bei Austritt aus dem Monitor und 'Signal(Condition)'

Der Monitor wird durch Markieren der Stelle S freigegeben: der Prozeß wird so lange blockiert, bis die Stelle C durch einen anderen Prozeß mit Ausführung der Anweisung 'Signal(Condition)' eine Marke erhält (Bild 14).

5 Beispiel: ein Roboterarm

Am folgenden Beispiel sollen die in den vorherigen Abschnitten besprochenen Überlegungen verdeutlicht werden. Zu diesem Zweck wurde ein fremder Algorithmus ([9], Algorithmus 1 in Pascal Plus) mit mehreren Prozessen und Monitoren gewählt: ein Roboterarm.

Dieser Roboterarm kann in die 3 Richtungen (x-, y- und z-Achse) durch 3 Schrittmotore bewegt werden. Jeder Motor wird durch einen Prozeß dargestellt. Erst wenn der Arm seine Zielposition in allen 3 Koordinatenrichtungen erreicht hat, kann die nächste Bewegung stattfinden. Die Koordinaten der einzelnen Zielpunkte werden von Hand über eine Tastatur eingegeben (Prozeß 'I/0'). Der implementierte Algorithmus sieht einen weiteren Prozeß 'Cntl' vor,

- der die eingehenden Zielkoordinaten über den Monitor 'I/0' einliest,
- diese über die Monitore 'Motor 1', 'Motor 2' und 'Motor 3' den Prozessen 'Motor 1', 'Motor 2' und 'Motor 3' jeweils übergibt und
- der auf eine Bestätigung der Ausführung über Monitor 'Terminate'

wartet.

Bild 15 skizziert die Interaktionen der verschiedenen Monitore und Prozesse.

Da die verschiedenen Monitore sehr ähnlich strukturiert sind, soll hier exemplarisch das Petri Netz der Monitorprozeduren 'Mio_send' und 'Mio_receive' des Monitors 'I/0' näher erläutert werden. Die folgende Beschreibung enthält alle zur Generierung des Petri Netzes relevanten

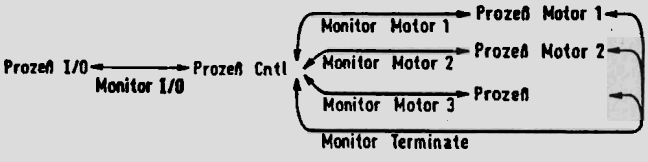


Bild 15 Interaktionen der Prozesse und Monitore des Roboterarms

Sprachkonstrukte des in MODULA-2 implementierten Algorithmus:

```
IMPLEMENTATION MODULE MIO; (*Monitor I/0*)
   . . (* Import Liste *)
VAR ..
   sender, receiver, . . : SIGNAL; (*Bedingungsvariablen *)
   oktosend, . . : BOOLEAN;
   X, Y, Z: INTEGER;
PROCEDURE MIOSEND (VAR x, y, z : INTEGER);
   BEGIN
    . .
   IF NOT oktosend THEN WAIT .. (sender, ..); END;
   oktosend := FALSE;
   X;=x; Y:=y; Z:=z;
   SEND(receiver);
    . .
END MIOSEND;

PROCEDURE MIORECEIVE (VAR x, y, z : INTEGER);
   BEGIN

   IF oktosend THEN WAIT..(receiver, ..); END;
   x:=X; y:=Y; z:=Z;
   oktosend := TRUE;
   SEND(sender);
    . .
END MIORECEIVE;

PROCEDURE MIOPUT;
    . .
PROCEDURE MIOGET;
    . .
BEGIN
 . .
oktosend:= TRUE; ..
END MIO.
```

Bild 16 zeigt das dazugehörige Petri Netz.

Der Eintritt in den Monitor (nach Bild 7) wird durch die Stelle S und die Transition T_1, T_2 und T_8, T_9 in Bild 16 dargestellt. Zusätzlich zu den monitorspezifischen Stellen und Transitionen wird die Boolsche Variable Oktosend als Stelle 0 mit der Kapazität 1 modelliert. In Abhängigkeit dieser Variablen wird die Anweisung 'Wait(sender)' ausgeführt (für die Prozedur 'Mio_send' durch Schalten der Transition T_2, für die Prozedur 'Mio_receive' durch Schalten von T_9). Für diesen Monitor trifft die Vereinfachung 4 (alle 'Signal(Condition)' eines Monitors sind die jeweils letzte Anweisung einer Monitorprozedur) zu, das heißt, die Semaphorvariablen 'Urgent' (Stelle U) und 'UrgentCount' (Stelle U_c) müssen nicht modelliert werden. Die 'Wait'-Anweisungen (Bild 9) werden durch die Netzelemente 'Prozeß blockiert', Sender, $Sender_c$, S, S_1, T_2 und T_3 bzw. durch 'Prozeß blockiert', Receiver, $Receiver_c$, S, S_3, T_8 und T_{10} modelliert. Anschließend muß die Variable 0 gesetzt werden. Dies geschieht entweder durch T_4, T_5 oder durch T_{11}, T_{12}. Für den Austritt aus dem Monitor gilt die Vereinfachung 2 ('Signal (Condition)' ist die letzte Anweisung in einer Monitor-

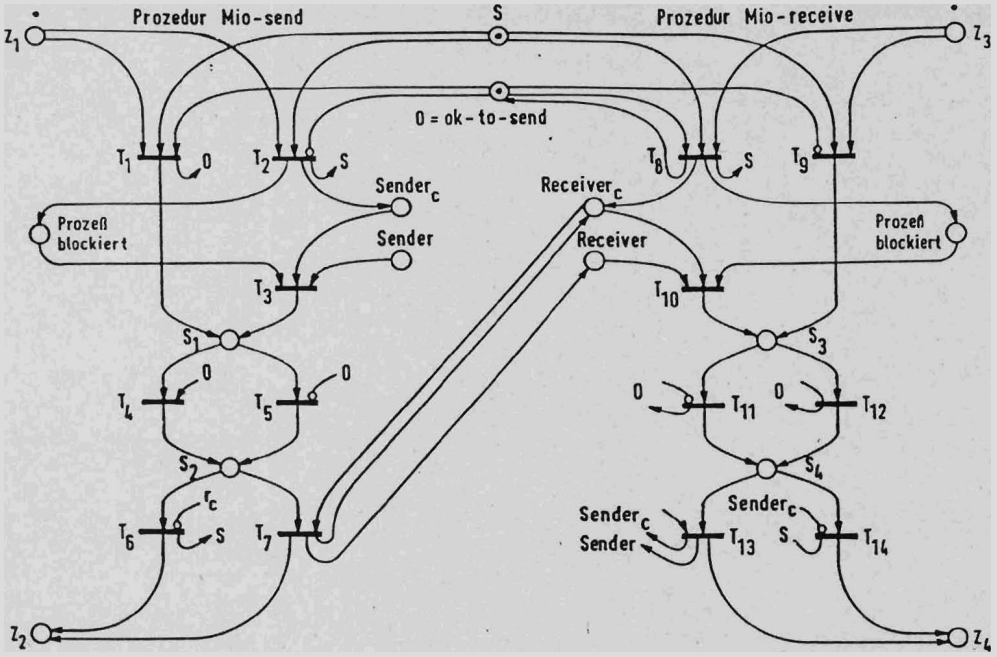


Bild 16
Petri Netz zu den Prozeduren Mio-send und Mio-receive des Monitors 'I/O'

prozedur). Außerdem sollen U_c und U nicht existieren (siehe weiter oben). Damit entsteht ein Petri Netz nach Bild 16 für Mio-send' und 'Mio-receive'. Durch die ähnliche Strukturierung der anderen Monitorprozeduren ergeben sich sehr ähnliche Netze.

6 Schlußbemerkungen

Das angeführte Beispiel wurde in MODULA-2 implementiert. Dabei wurde kein Wert auf die Optimierung des angegebenen Algorithmus gelegt. Das Verständnis und die Überprüfung der korrekten Funktionsfähigkeit der Kommunikation und Synchronisation des Roboterarmbeispiels (Programmierung in MODULA-2) haben ohne die Betrachtung des Petri Netzes einen unvorhergesehenen Aufwand mit sich gebracht. Dies ist vor allem durch die Programmierung paralleler Prozesse bedingt. Durch die Übersetzung der Monitore in Petri Netze (mit ihren graphischen Darstellungsmöglichkeiten) wurde wesentlich an Übersichtlichkeit gewonnen. Die korrekte Funktion des Gesamtnetzes konnte durch eine Petri Netz Analyse gezeigt werden: So hatte zum Beispiel der Erreichbarkeitsgraph des Netzes keine unvorhergesehenen Blätter, d.h. es gibt damit auch keinen Zustand der absoluten Systemblockierung. Unter der Beachtung des verwendeten Petri Netz-Modells ließen und lassen sich viele Analysen auf Petri Netz-Ebene auf die modellierten Monitore anwenden.

Auch Realisierungen bekannter Problemstellungen (des 'Consumer-Producer'-, 'Bounded Buffer'- und Philosophen-Problems) mit Monitore lassen sich so überprüfen. Allerdings müssen zur kompletten Analyse derartiger Probleme auch die Prozesse selbst in Petri Netze umgesetzt werden. Es ergeben sich hier bei Datenabhängigkeit der Prozeßinstances noch Modellierungsschwierigkeiten, weil die Struktur der Petri Netze statisch ist und zum Analysezeitpunkt bekannt sein muß. Bis jetzt ließen sich jedoch alle untersuchten Prozesse soweit vereinfachen, daß eine Untersuchung der Synchronisation und Kommunikation über Monitore möglich war. Bei der vorgestellten Betrachtung wurde, entsprechend der Realisierung in MODULA-2, keine Warteschlange für die Prozesse vor dem Eintritt in den Monitor berücksichtigt. Unter der Voraussetzung, daß dem Anwender die Hilfsmittel:

graphische Ausgabe der Petri Netze, Analyseprogramme und eine automatisierte Übersetzung von Monitor in Petri Netz zur Verfügung stehen, kann das in diesem Artikel beschriebene Verfahren

- durch eine Unterstützung des Verständnisses fertiger Programme und
- durch eine Verkürzung der Fehlerkorrekturphase

bei der Programmierung mit Monitoren sinnvoll eingesetzt werden.

Für die Unterstützung dieser Arbeit möchte sich der Autor bei Herrn Prof. Dr.-Ing. K. Hoffmann für wertvolle Diskussionen und bei Herrn Dr.-Ing. G. Rücklé für Hinweise bei der Realisierung der Beispielprogramme in MODULA-2 bedanken.

Literatur

[1] *G. Andrews, F. Schneider:* Concepts and Notations for Concurrent Programming. In: Computing Surveys, Vol 15, No. 1, März 1983, S. 3–43.

[2] Reference Manual for the ADA Programming Language. US Dept. of Derence, MIR Std 1815a, 1983.

[3] *M. Ben-Ari:* Principles of Concurrent Programming. Prentice-Hall Inc., Ennglewood Cliffs, N.J., USA, 1982.

[4] *P. Brinch Hansen:* Concurent Programming Concepts. In: ACM Computing Surveys, Vol 5, 1973, S. 223–245.

[5] *P. Brich Hansen:* The Programming Language Concurrent Pascal. In: IEEE Trans. On Software Engineering, Vol. 2, 1975, S. 199–207

[6] *W. Bustard, J. Welsh:* Pascal-Plus – Another Language for Modular Programming. In: Software-Practice and Experience, Vol. 9, 1979, S. 947–957.

[7] *E. Dijkstra:* Cooperating Sequential Processes. In: F. Genuys (Ed.) Programming Languages, Academic Press, New York, 1968, S. 43–112.

[8] *C. A. R. Hoare:* Monitors: An Operating System Structuring Concept. In: Comm. Of The ACM, Vol. 17, No. 10, Oktober 1974, S. 549–557.

[9] *J. Kerridge, D. Simpson:* Three solutions for a Robot Arm Controller Using Pascal-Plus, Occam and Edison. In: Software-Practice and Experience, Vol. 14, 1984, S. 3–15.

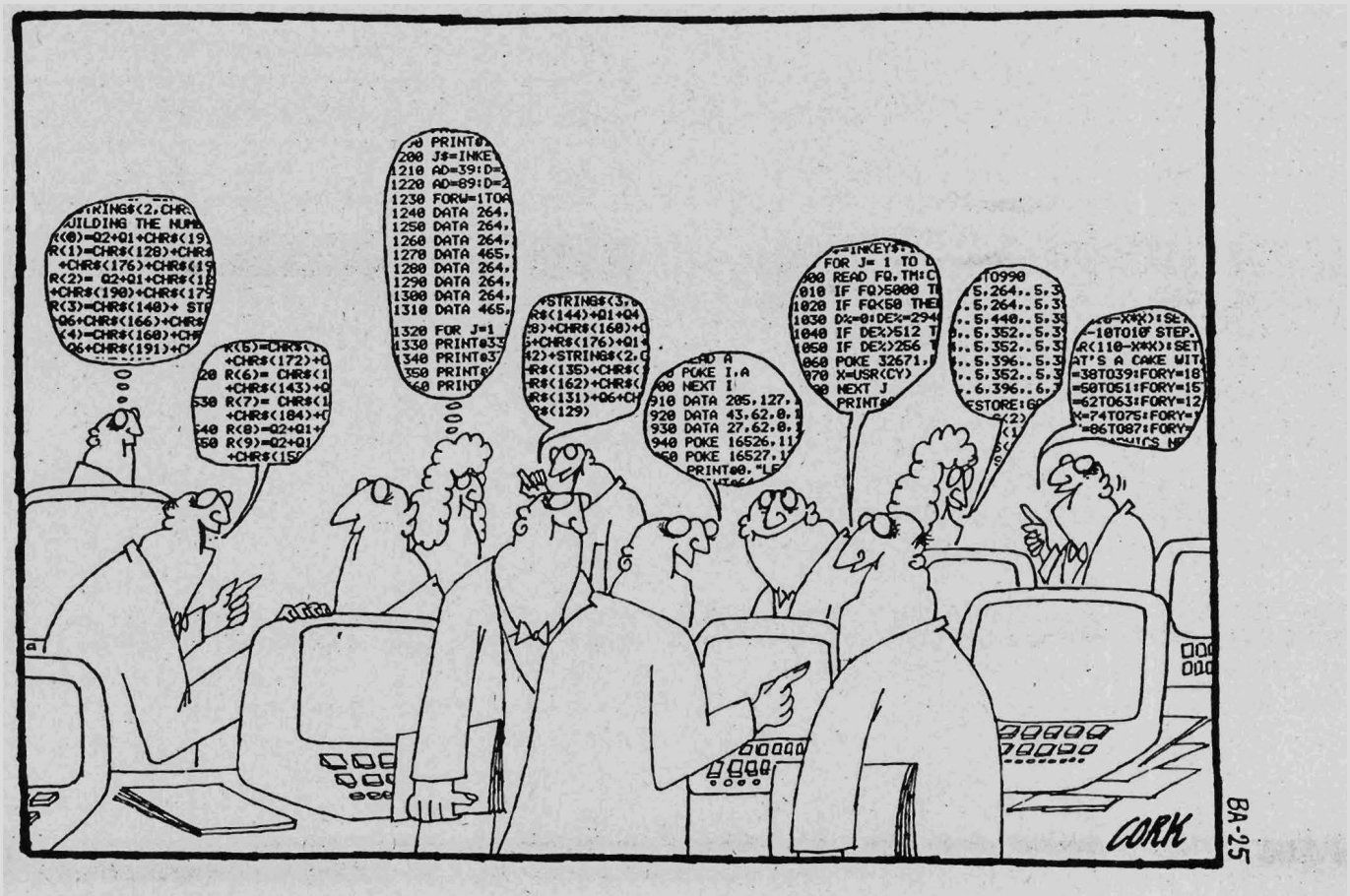
[10] *J. Nehmer:* Systemarchitektur von Realzeitsystemen. In: Informatikspektrum, Vol. 7, 1984, S. 65–72.

[11] *J. Peterson:* Petri Net Theory and the Modeling of Systems. Prentice-Hall Inc., Ennglewood Cliffs, N. J., USA, 1981.

[12] *R. Prinoth:* Semaphorkonzepte und ihre Darstellung durch Petri-Netze. Gesellschaft für Mathematik und Datenverarbeitung, Interner Bericht, 1979.

[13] *W. Reisig:* Petrinetze. Springer-Verlag, Berlin Heidelberg New York, 1982.

[14] *N. Wirth:* Programming in Modula. Springer-Verlag, Berlin Heidelberg New York, 1982.



Angewandte Informatik

Die „Angewandte Informatik" (bis einschl. 1970 „elektronische Datenverarbeitung") ist eine Datenverarbeitungsfachzeitschrift gehobenen Niveaus, in der theoretisch und praktisch orientierte Beiträge aus dem weiten Bereich der Anwendung von Computern veröffentlicht werden. Sie ist

- anwendungsorientiert deshalb, weil es sich bei den relevanten Aspekten von Informatik oder automatisierter Datenverarbeitung nicht um theoretisierende Erwägungen handelt, sondern um praktische (und theoretische) Probleme, deren Lösung uns alle angeht;
- forschungsorientiert insoweit, als „Forschung" auf dem Gebiet der Informatik bzw. automatisierten Datenverarbeitung bei den großen Computer- und Softwareherstellern und privaten Großanwendern ebenso stattfindet wie bei den Hoch- und Fachhochschulen der öffentlichen Hand;
- entwicklungsorientiert aus demselben Grunde;

Letztlich will die „Angewandte Informatik" einen Beitrag zur stetigen Weiterentwicklung der automatisierten Datenverarbeitung leisten, und gemäß ihrer eigenen Struktur und Entwicklung sieht sie ihre Möglichkeiten dazu in erster Linie in einer Synthese gewissermaßen zwischen den Belangen der Praxis und den Erkenntnissen und Forderungen der Theorie: daher (seit 1971) **Angewandte Informatik.**

Die Fachbeiträge stammen überwiegend von Informatik-Fachleuten und Datenverarbeitungs-Experten aus dem Anwender-, dem Hersteller- und dem Hochschulbereich, und zwar von praxisorientierten Wissenschaftlern und wissenschaftsorientierten Spezialisten etwa im gleichen Maße. Hinzu kommen Autoren aus Randbereichen und Nachbardisziplinen von Informatik und automatisierter Datenverarbeitung, die das Gebiet auch aus ungewohnten Blickwinkeln beleuchten.

Die Leserschaft setzt sich aus all jenen zusammen, die in Praxis, Wissenschaft und Studium beruflich auf qualifiziertem Niveau maßgeblich mit Informatik und Datenverarbeitung zu tun haben. Insbesondere handelt es sich dabei um Inhaber leitender Funktionen in diesem Bereich sowie um qualifizierte Spezialisten. Erheblich ist auch der Anteil der Leser aus den Nachbargebieten der Informatik – ein Beweis für die Vielfalt der in der „Angewandten Informatik" behandelten Themen.

Reprint

Eigenes Exemplar

Eigenes Exemplar

Angewandte Informatik

applied informatics