Gliederung des Kapitels

2.2 Prozesse

~~2.2.1 Stellen-Transitions-Netze~~
2.2.2 Nachrichtenaustausch
2.2.3 Parallelität
2.2.4 Nebenläufigkeit in Ada95

2.2.2 Nachrichtenaustausch

Wie können verschiedene Prozesse Daten austauschen. Bei den S/T-Netzen müssen sich die Daten im Vorbereich der Transition befinden; sie werden "lokal" über den Nachbereich anderen Prozessen zur Verfügung gestellt.

Wir betrachten nun zwei andere Mechanismen:
- gemeinsamer Speicherbereich (shared variables),
- Aufbau von Kanälen.
Der Datenaustausch kann synchron und asynchron erfolgen.

Hierbei gehen wir sofort von einer konkreten Programmiersprache aus. Diese ist eine Erweiterung der Sprachelemente, die wir zu Beginn der Vorlesung eingeführt hatten. Wir beginnen mit dem gemeinsamen Speicherbereich.

2.2.2.1 Elementare Anweisungen: Diese übernehmen wir aus 1.1.1.5 und fügen await hinzu:

skip	Nichtstun.
X := α	Wertzuweisung. α ist ein Ausdruck. (Rechne den Ausdruck α aus und lege den erhaltenen Wert in der Variablen X ab.)
read (X)	Leseanweisung. (Lies den nächsten Wert ein und lege ihn in der Variablen X ab.)
write (α)	Schreibanweisung. (Drucke den Wert, den der Ausdruck α besitzt, aus.)
await β	Warten. *Bedeutung:* Warte, bis β wahr ist. (β ist ein Boolescher Ausdruck)
$F(X_1, ..., X_n)$	Aufruf. (Führe den Algorithmus F mit den Werten der Variablen $X_1, ..., X_n$ aus.)

2.2.2.2 Zusammengesetzte Anweisungen: Aus 1.1.1.5 übernehmen wir:

Hintereinanderausführung oder Sequenz: Wenn S_1 und S_2 Anweisungen sind, dann ist auch $S_1; S_2$ eine Anweisung.

Alternative: Wenn S_1 und S_2 Anweisungen und β ein Boolescher Ausdruck sind, dann ist auch

if β then S_1 else S_2 fi .

eine Anweisung (else skip darf man weglassen.)

Schleife: Wenn S eine Anweisung und β ein Boolescher Ausdruck sind, dann ist auch while β do S od eine Anweisung.

Hinzu kommen folgende zusammengesetzte Anweisungen:

Nichtdeterministische Auswahl für n≥2:

$(S_1$ <u>or</u> S_2 <u>or</u> S_3 <u>or</u> ... <u>or</u> $S_n)$

Nebenläufige Abarbeitung für n≥2:

$(S_1 \mid S_2 \mid S_3 \mid ... \mid S_n)$

Blöcke mit gemeinsamen Variablen:

[<u>local</u> <lokale Deklarationen>; S]

Wie in Ada lassen wir in den lokalen Deklarationen Konstanten und Initialisierungen zu. Weiterhin darf man jeder Anweisung eine Marke mittels <Name>:: voranstellen..

Unsere Beispiele haben meist die Form:

P:: [<u>local</u> <lokale Deklarationen>; $(S_1 \mid S_2 \mid S_3 \mid ... \mid S_n)$]

<u>Beispiel 2.2.2.3</u>:

```
[                 local   L: integer := 0;
                          max: constant integer := 2;

  ( P1:: while true do                  |   P2:: while true do
        await L < max;                  |       await L > 0;
        (L := L+1  or  skip)            |       (L := L-1 or skip)
        od;                             |       od;                )
]
```

Dieses Programm beschreibt den Erzeuger-Verbraucher-Kreislauf, siehe 2.2.1.3, mit der Lagergröße L, die zwischen 0 und max liegen darf.

Dieses Programms besitzt zwei Prozesse. Was ist seine Bedeutung? Ist es genau die gleiche wie die des S/T-Netzes?

Wir führen den Begrif des Zustands für Programme mit mehreren Prozessen ein. Ein Zustand gibt zu jedem Zeitpunkt an, welche Werte die Variablen besitzen und an welchen Stellen im Programm sich die Prozesse befinden.

Hierfür müssen wir die "Stelle im Programm" definieren. Grob gesprochen ist es eine Stelle zwischen zwei elementaren Anweisungen oder Berechnungen von Bedingungen. Wir nummerieren diese Stellen einfach durch, z.B.:

```
P1:: (1) while (2) true do                P2:: (1) while (2) true do
  (3) await  L < max;                       (3) await L > 0;
      ((4) L := L+1  or (5) skip)               ((4) L := L-1 or (5) skip)
  (6) od;                                   (6) od;
```


Hier besitzt man eine gewisse Willkür. Man kann besipielsweise auch die Stelle vor der inneren nebenläufigen Anweisung markieren, also

(4)((5)L := L+1 or (5') skip)

Man kann auch die Berechnungen der Ausdrücke feiner unterteilen, also

(4)((5) L (7) := (6)L+1 or (5') skip)

Dies hängt davon ab, ob die Programme in Zeittakten bearbeitet werden oder ob es Rechnungen gibt, die von außen nicht unterbrochen werden können.

Wir formulieren nun die Menge der möglichen Zustände zu der Unterteilung, die auf der vorigen Folie angegeben wurde.

P1:: ① while ② true do
③ await L < max;
(④ L := L+1 or ⑤ skip)
⑥ od;

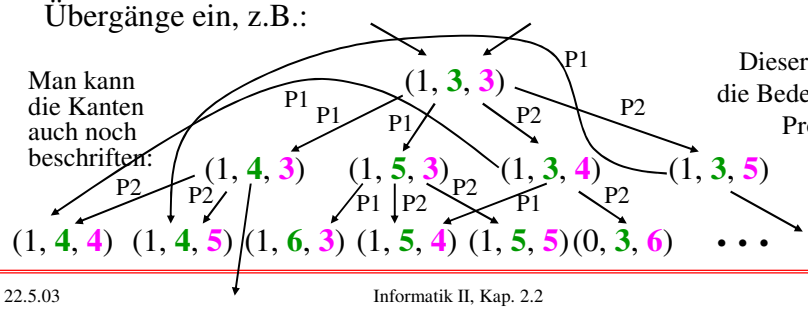
||

P2:: ① while ② true do
③ await L > 0;
(④ L := L-1 or ⑤ skip)
⑥ od;

Zustandsmenge

Z = { (a, **i**, **j**) | a ist der Wert von L, **i** ist die Stelle im Programm P1 und **j** ist die Stelle im Programm P2 }

Nun tragen wir, wie beim Erreichbarkeitsgraphen, die möglichen Übergänge ein, z.B.:



P1:: ① while ② true do
③ await L < max;
(④ L := L+1 or ⑤ skip)
⑥ od;

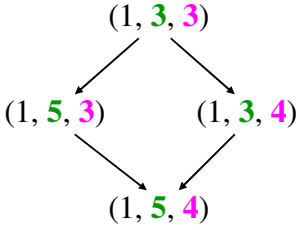
||

P2:: ① while ② true do
③ await L > 0;
(④ L := L-1 or ⑤ skip)
⑥ od;

In der Regel betrachtet man hierbei keine "Gleichzeitigkeit". Zum Beispiel ist der Übergang

(1, **3**, **3**)
↓
(1, **5**, **4**)

dann nicht zulässig, man muss vielmehr zwei Schritte in irgendeiner Reihenfolge durchführen:



2.2.2.4: Legt man die Bedeutung (Semantik) nebenläufiger Programme so fest, dass niemals zwei Programme gleichzeitig einen Schritt ausführen können, sondern stets eine Reihenfolge erzwungen wird, so spricht von einer "Interleaving"-Semantik.

Die Interleaving-Semantik lässt sich aus theoretischer Sicht leichter behandeln als eine Semantik, in der auch Gleichzeitigkeit zugelassen ist. Weiterhin beschreibt die Interleaving-Semantik genau die Verhältnisse, die bei einem Monoprozessor vorliegen, der die verschiedenen nebenläufigen Programme alleine ausführen muss (der also die Nebenläufigkeit nur vortäuscht).

2.2.2.5: Zur "Granularität" (feinkörnig / grobkörnig): Um die Zustände exakt definieren zu können, muss man festlegen, welche Anweisungsteile "nicht unterbrechbar" sind. Solche Teile werden als in sich geschlossene Einheiten angesehen, deren Ausführung von anderen Ereignissen nicht gestört wird.

Sind in unserem obigen Beispiel "L:=L+1" und "L:=L-1" nicht unterbrechbar auf, so hat nach der nebenläufigen Abarbeitung von (Q1:: L:=L+1 | Q2:: L:=L-1) die Variable L ihren Wert nicht verändert. Sind aber nur die arithmetischen Operationen und die Wertzuweisungen jede für sich nicht unterbrechbar, so kann folgende Möglichkeit bei dieser Abarbeitung auftreten:

Hole den Wert a von L bzgl. Q1; hole den Wert a von L bzgl. Q2; bilde a+1 bzgl. Q1; bilde a-1 bzgl. Q2; weise a+1 der Variablen L zu bzgl. Q1; weise a-1 der Variablen L zu bzgl. Q2.

Am Ende hat L also den Wert a-1 (an Stelle des erwarteten Wertes a).

Sind überhaupt keine Anweisungsteile "nicht unterbrechbar", dann sind alle zeitlich verschachtelten Reihenfolgen möglich.

Beispiel: (R1:: X:=Y; X := X+1 | R2:: Y:=X; X:=X+1)

Man kann diese Anweisungen nun beliebig in der zeitlichen Reihenfolge ineinander stecken. Beispiele sind:

Anfangs seien X=a, Y=b	
R1:	R2:
Y holen	
	X holen
	Y zuweisen
	X holen
	um 1 erhöhen
	X zuweisen
X zuweisen	
X holen	
um 1 erhöhen	
X zuweisen	
Am Ende: X=b+1, Y=b	

Zeit

Anfangs seien X=a, Y=b	
R1:	R2:
	X holen
	Y zuweisen
Y holen	
	X holen
	um 1 erhöhen
	X zuweisen
X zuweisen	
X holen	
um 1 erhöhen	
X zuweisen	
Am Ende: X=a+1, Y=a	

(R1:: X:=Y; X := X+1 | R2:: Y:=X; X:=X+1)

Anfangs seien X=a, Y=b	
R1:	R2:
Y holen	
X zuweisen	
X holen	
	X holen
um 1 erhöhen	
	Y zuweisen
X zuweisen	
	X holen
	um 1 erhöhen
	X zuweisen
Am Ende: X=b+2, Y=b	

Zeit

Anfangs seien X=a, Y=b	
R1:	R2:
	X holen
Y holen	
	Y zuweisen
X zuweisen	
	X holen
X holen	
	um 1 erhöhen
um 1 erhöhen	
	X zuweisen
X zuweisen	
Am Ende: X=b+1, Y=a	

Prüfen Sie: Lassen sich auch *Am Ende: X=b+2, Y=b+1* oder *Am Ende: X=a+2, Y=b+1* erreichen? Wie viele verschiedene Möglichkeiten gibt es bei diesem Beispiel?

Hierbei können Konflikte auftreten. Beispielsweise muss man vermeiden, dass ein Prozess Werte in eine Variable (= in einen Speicherbereich) schreibt, während ein anderer Prozess diese Variable ebenfalls verändert. Das Gleiche gilt, wenn irgendein anderes Betriebsmittel (Eingabegerät, Drucker, Übertragungmedium, Beamer usw.) exklusiv genutzt werden muss.

Die Anweisungsfolge, die ungestört von nebenläufigen Prozessen ausgeführt werden muss, nennt man einen *kritischen Abschnitt.* Kann man während ihrer Ausführung den exklusiven Zugriff garantieren, so spricht man vom wechselseitigen oder gegenseitigen Ausschluss.

Gewisse exklusive Zugriffe lassen sich hardwaremäßig sicherstellen, z.B. der Zugriff auf eine Speicherzelle. Für Anweisungsfolgen muss man in der Praxis softwaremäßige Lösungen finden.

2.2.2.6 Definition:

Ein sequentiell abzuarbeitender Teil eines Programms heißt kritischer Abschnitt, wenn es ein Betriebsmittel gibt, das während der Ausführung dieses Programmteils von keinem anderen (hierzu nebenläufigen) Prozess genutzt werden darf. Prozesse, die auf das gleiche Betriebsmittel zugreifen wollen, *konkurrieren* um dieses Betriebsmittel und *bilden einen Konflikt.*

In einem kritischen Abschnitt darf sich zu jedem Zeitpunkt höchstens einer der konkurrierenden Prozesse befinden. Kann man sicherstellen, dass sich bei einem Konflikt zu jedem Zeitpunkt höchstens einer der konkurrierenden Prozesse in seinem kritischen Abschnitt befindet, so spricht man vom wechselseitigen Ausschluss (mutual exclusion).

2.2.2.7 Beispiel

Ein kritischer Abschnitt ist in einem Programm das Ausdrucken von Daten, wenn nur ein Drucker vorhanden ist. Im einfachsten Fall konkurrieren nur zwei Prozesse um den Drucker.

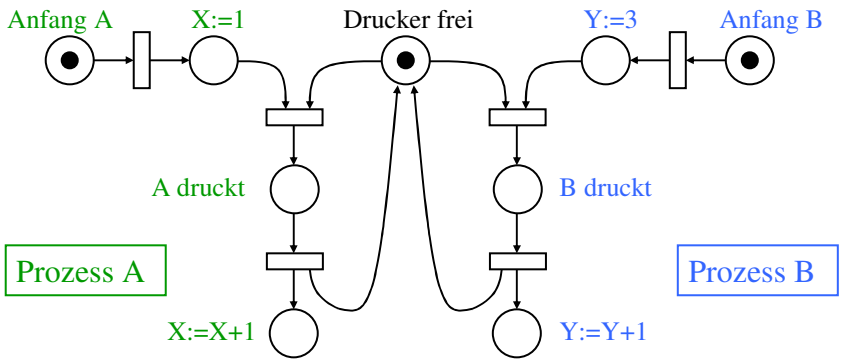
Wir beschreiben im Folgenden den wechselseitigen Ausschluss zuerst mit einem S/T-Netz. Anschließend realisieren wir ihn softwaremäßig durch geeignete Kontrollvariable in den beiden Prozessen.

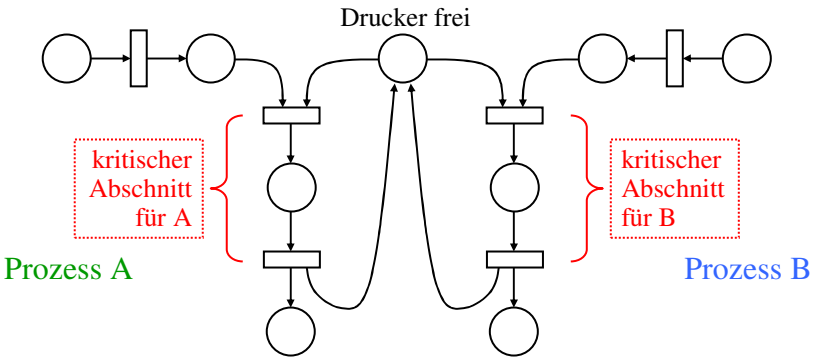
Beispielprogramm: Zwei Prozesse wollen X bzw. Y drucken.

[local X, Y : integer;
(A:: X:=1; write(X); X:=X+1 |B:: Y:=3; write(Y); Y:=Y+1)]

[local X, Y : integer;
(A:: X:=1; write(X); X:=X+1 |B:: Y:=3; write(Y); Y:=Y+1)]

Die Programmteile write(X) und write(Y) bilden für jeden der beiden Programmteile kritische Abschnitte. Für dieses Beispiel lässt sich der wechselseitige Ausschluss mit S/T-Netzen leicht modellieren, indem man dem Betriebsmittel "Drucker" eine Stelle "frei" zuordnet :





Beschreibung des wechselseitigen Ausschlusses mit einem S/T-Netz.

Für die Realisierung in der Programmierung wird die Stelle "Drucker frei" durch eine Boolesche Variable dargestellt:

Obiges Beispiel als Programm mit wechselseitigem Ausschluss:

```
[ local X, Y: integer; frei: Boolean := true;
   (A:: X:=1;                    |   B:: Y:=3;
        await frei;              |        await frei;
        frei:=false;             |        frei:=false;
        write(X);                |        write(Y);
        frei := true;            |        frei := true;
        X:=X+1                   |        Y:=Y+1 )      ]
```

Diese Realisierung ist nur dann korrekt, wenn die Anweisungsfolge " await frei; frei := false " nicht unterbrechbar ist! Anderenfalls könnten beide Prozesse (fast) gleichzeitig auf die Variable "frei" zugreifen und sie beide als true erkennen. Beide Prozesse betreten dann gleichzeitig ihre kritischen Abschnitte.

Hinweis: Wenn k Drucker für mehrere Prozesse vorliegen, so würde man im S/T-Netz die Stelle "Drucker frei" anfangs mit k Marken belegen. Bei der Übertragung in ein Programm muss man dann eine Variable

```
Drucker_frei: natural := k;
```

deklarieren. Will einer der Prozesse in seinen kritischen Abschnitt eintreten, so würde man schreiben:

```
await Drucker_frei > 0;
Drucker_frei := Drucker_frei - 1;
< kritischer Abschnitt >;
Drucker_frei := Drucker_frei + 1;
```

Im allgemeinsten Fall würde man bei der Programmierung noch prüfen, ob die Variable Drucker_frei einen maximalen Wert MAX nicht überschreiten kann, d.h., man würde vor dem Erhöhen von Drucker_frei noch await Drucker_frei < MAX einfügen.

2.2.2.8 a Definition (das Semaphor, eingeschränkte Form)

Eine Variable vom Typ natural, die eine Menge von maximal MAX Ressourcen "bewacht", zusammen mit den nicht unterbrechbaren Operationen "Warten und Erniedrigen" und "Warten und Erhöhen" bezeichnet man als Semaphor. (Im Falle MAX=1 kann man auch eine Variable vom Typ Boolean verwenden, siehe oben; im Falle MAX = ∞ entfällt das Warten vor dem Erhöhen.)

Erläuterung des Namens: Unter einem Semaphor versteht man einen Signalmast, auch "Flügeltelegraph" genant. Solche Masten wurden ab 1790 für die optische Übermittlung von Nachrichten benutzt. Ab 1840 (in Europa ab 1850) wurden sie rasch durch die elektrische Nachrichtenübertragung ("Telegraph") verdrängt. Es gibt sie noch als "Windtelegraphen" in der Schifffahrt.

Hinweis: Das allgemeine Semaphorkonzept wurde 1968 von dem niederländischen Informatiker E. W. Dijkstra eingeführt. Neben der Kontrollvariablen S besitzt jedes solche Semaphor eine Warteschlange, in die alle Prozesse nacheinander eingetragen werden, die zur Zeit noch nicht auf das Betriebsmittel zugreifen können. Das Semaphor aktiviert die Prozesse in der Warteschlange, sobald ein angefordertes Betriebmittel frei ist.

In der Literatur bezeichnet man die Operation "Warten und Erniedrigen", also await S > 0; S := S - 1 auch als P-Operation der Semaphorvariablen S (nach dem niederländischen Wort Passeer = Betreten) oder als Warteoperation. Die andere Operation heißt V-Operation (vom niederländischen Verlaat = Verlassen) oder Signaloperation.

2.2.2.8 b Definition (das Semaphor, ausführliche Form)

Ein Semaphor besteht aus einer Variablen S des Typs natural (oder 0..MAX), einer Warteschlange W(S) für Prozesse und folgenden beiden nicht-unterbrechbaren Operationen P und V:

```
procedure P (S: in out natural);
begin  if S > 0 then S := S-1;
       else  <Stoppe diesen Prozess>;
             <trage ihn in die Warteschlange W(S) ein>; end if;
end P;

procedure V (S: in out natural);
begin  S := S+1;
if not isempty(W(S)) then <Wähle einen Prozess A aus W(S) aus>;
       <aktiviere dessen Ausführung ab der P-Operation in A,
        durch die A gestoppt wurde>;  end if;
end V;
```

Obiges Beispiel als Programm mit allgemeinem Semaphor:

```
[ local X, Y: integer; S: semaphor;
   (A:: X:=1;                  |   B:: Y:=3;
        P(S);                  |        P(S);
        write(X);              |        write(Y);
        V(S);                  |        V(S);
        X:=X+1                 |        Y:=Y+1 )      ]
```

Semaphore sind eine Standardtechnik, um den wechselseitigen Ausschluss zu realisieren, bzw. allgemein, um eine gegebene Menge von Betreibemitteln mehreren auf sie zugreifenden Prozessen zur Verfügung zu stellen, ohne dass eine Verklemmung eintreten kann.
(Vgl. "Scheduler" in Vorlesungen über Betriebssysteme.)

Problem: Kann man durch ein Programm, das nur unsere Anweisungen benutzt (also keine allgemeinen Semaphore kennt), den wechselseitigen Ausschluss sicherstellen, falls keine unterbrechbaren Operationen vorliegen (nur das Schreiben in eine Variable sei nicht-unterbrechbar)?

Wie muss man also das bisherige Programm

```
[ local X, Y: integer; frei: Boolean := true;
   (A:: X:=1;                  |   B:: Y:=3;
        await frei;            |        await frei;
        frei:=false;           |        frei:=false;
        write(X);              |        write(Y);
        frei := true;          |        frei := true;
        X:=X+1                 |        Y:=Y+1 )      ]
```

abändern? Oder kann man dies gar nicht garantieren??

Vorschlag: eine Variable einführen, die nur die Werte PA und PB annehmen kann:

```
[ local X, Y: integer;
        type prozess is (PA, PB); Nr: prozess := PA;

  (A:: X:=1;                  |   B:: Y:=3;
       Nr := PB;              |       Nr := PA;
       await Nr = PA;         |       await Nr = PB;
       write(X);              |       write(Y);
       Nr := PB;              |       Nr := PA;
       X:=X+1                 |       Y:=Y+1 )      ]
```

Jeder Prozess gibt dem anderen Prozess die Berechtigung, als erster in den kritischen Abschnitt einsteigen zu können. Ist dies ein Konzept, das Fehler vermeidet?