Stand: 27.7.06

Wünschenswerte Fortsetzung:

13. Prozesse

13.1.1 Stellen-Transitions-Netze

13.1.2 Nachrichtenaustausch

13.1.3 Nebenläufigkeit in Ada

Es ist geplant, dieses Kapitel in "Informatik III" im Wintersemester 06/06 mit zu behandeln.

13.1 Stellen-Transitions-Netze

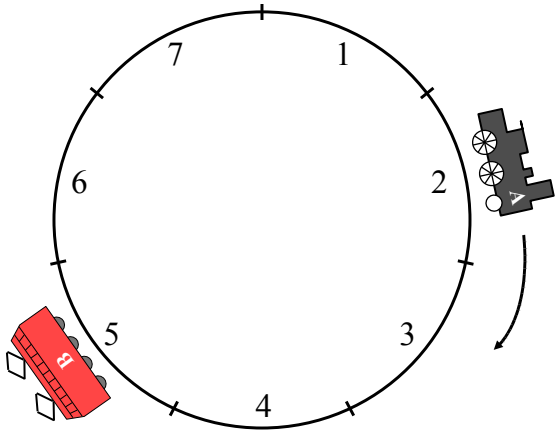
Bisher: Sequentielle Programmierung, d.h., höchstens eine Stelle im Programm wird in jedem Augenblick bearbeitet. Jeder Ablauf wird hierbei in eine Folge nacheinander auzuführender Aktivitäten zerlegt.

Im Folgenden wollen wir unabhängig voneinander ablaufende Programme (Prozesse, Objekte) und deren Kommunikation beschreiben. Man spricht von Nebenläufigkeit (engl.: concurrency) und von nebenläufigen, von verteilten und von parallelen Systemen.

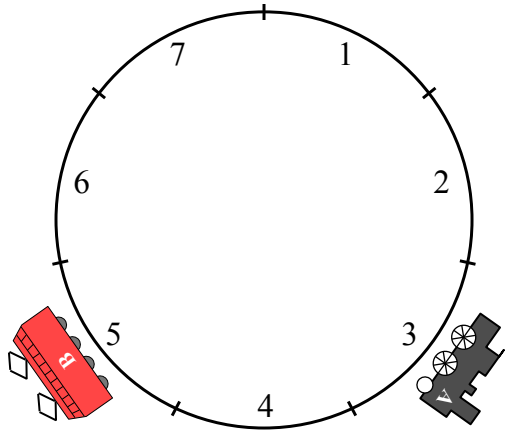
Ansatz: Verallgemeinere endliche Automaten, indem mehrere Zustände gleichzeitig oder nacheinander aktiv sind. Als Beispiel wählen wir Züge auf einer Kreisstrecke.

13.1.1 Beispiel: Züge auf einer kreisförmigen Strecke.

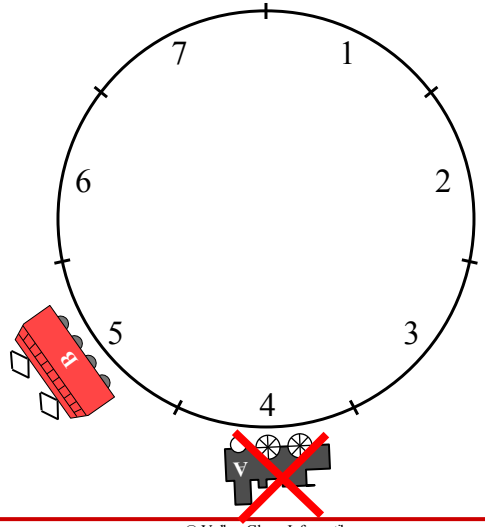
Damit die Züge nicht zusammenstoßen, verlangen wir, dass zwischen ihnen mindestens ein Streckenabschnitt frei bleibt.



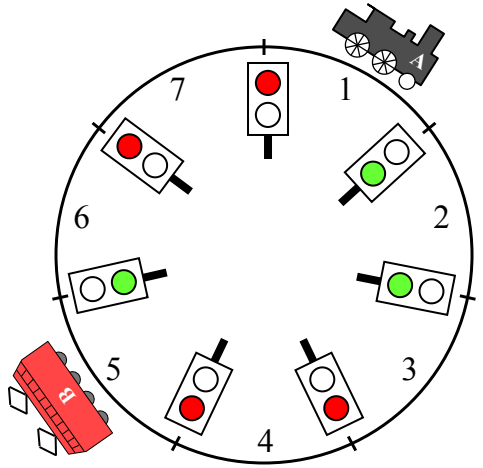
Damit die Züge nicht zusammenstoßen, verlangen wir, dass zwischen ihnen mindestens ein Streckenabschnitt frei bleibt.



Diese Situation darf also nicht eintreten. Wie kann man dies sicherstellen?



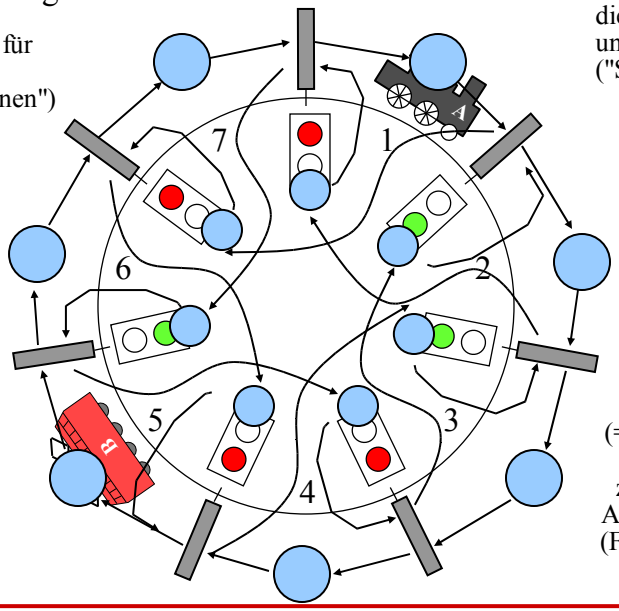
Steuerung durch Ampeln: Grünes Signal bedeutet, dass in den Streckenabschnitt hineingefahren werden darf.



Umwandlung in ein Netz:

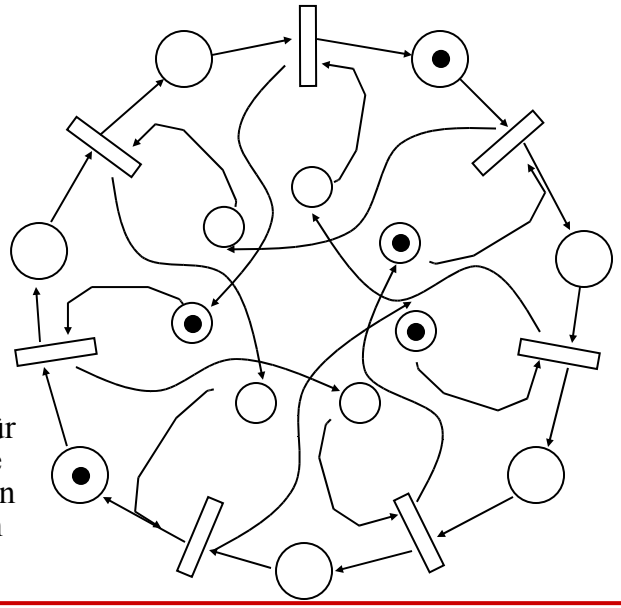
Rechtecke für Übergänge ("Transitionen")

Kreise für die Strecken und Ampeln ("Stellen")



Pfeile (=Kanten) für Vorausset-zungen und Auswirkungen (Flussrelation)

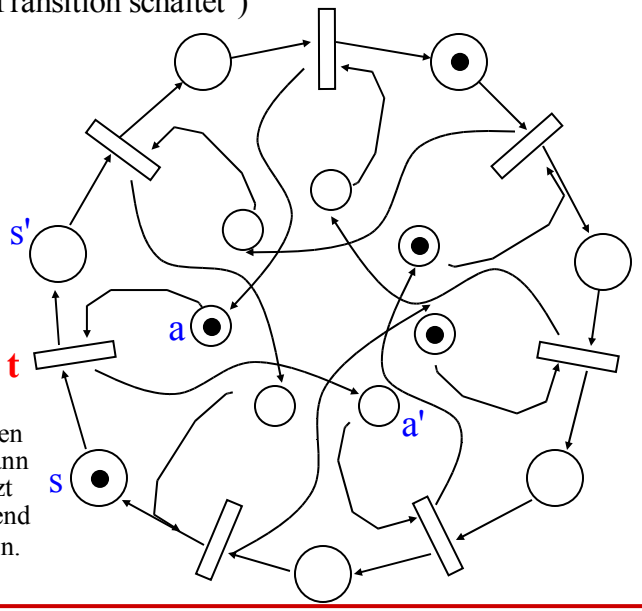
Das resultierende Netz:



Marken für mögliche Aktivitäten einfügen

Eine Aktivität durchführen ("eine Transition schaltet")

Die Transition t "schaltet":

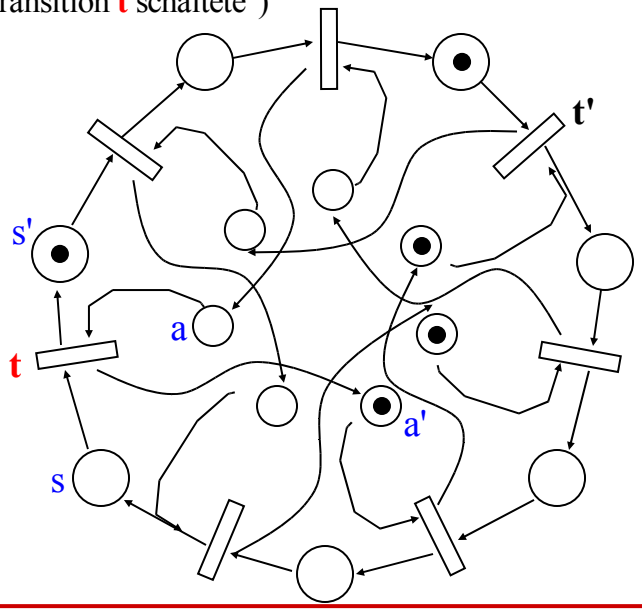


Die Marken werden dann umgesetzt entsprechend den Kanten.

Ein Zug kann von Strecke s nach Strecke s' fahren (= in Stelle s liegt eine Marke). Die Ampel steht auf grün (= in der Stelle a liegt eine Marke). Der Zug fährt nun von s nach s' (= die Transition t schaltet).

Eine Aktivität durchführen ("die Transition t schaltete")

Ergebnis des Schaltens:



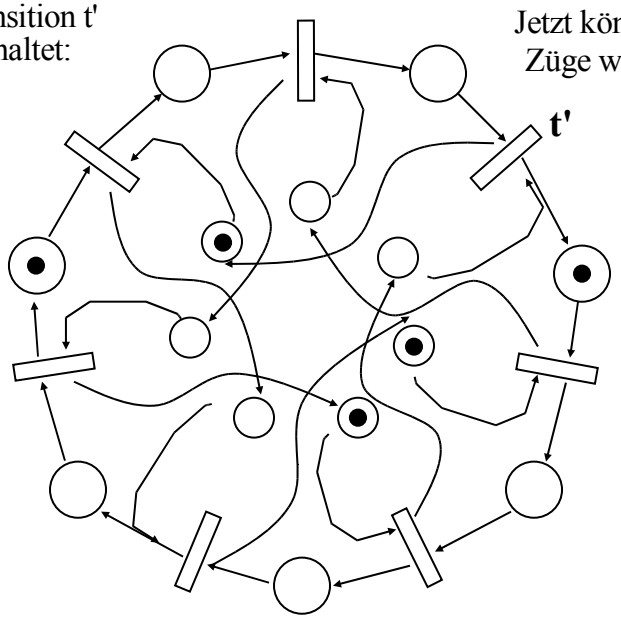
Der Zug ist nun auf der Strecke s' (= in Stelle s' liegt eine Marke). Die Ampel a steht auf rot (= in der Stelle a liegt keine Marke). Dafür wird aber a' auf grün gestellt (= in Stelle a' liegt eine Marke).

Eine Transition t schaltet bedeutet also:
Voraussetzung: Auf allen Stellen, von denen eine Kante nach t führt, muss mindestens eine Marke liegen.
Aktion: Von jeder dieser Stellen wird eine Marke abgezogen. Danach wird zu jeder Stelle, zu der eine Kante von t führt, eine Marke hinzugefügt.

Man nennt dies die "Schaltregel". Wir werden sie im Folgenden exakt definieren.

In unserem Beispiel: Der linke Zug kann nicht weiterfahren, weil die zugehörige Ampel keine Marke enthält. Aber der rechte Zug kann weiterfahren, d.h., die Transition t' kann jetzt schalten. Das Ergebnis (= die neue Verteilung der Marken) finden Sie auf der nächsten Folie.

Die Transition t' hat geschaltet:



Jetzt könnten beide Züge weiterfahren usw.

Definition 13.1.2: Stellen-Transitionsnetz (S/T-Netz)

$N = (S, T, F, K, W, M_0)$ heißt Stellen-Transitions-Netz $\Leftrightarrow$

(1) S ist eine endliche Menge (Menge der "Stellen"),

(2) T ist eine endliche Menge (Menge der "Transitionen"),

(3) $F \subseteq (S \times T) \cup (T \times S)$ ist die "Flussrelation" (Kantenmenge),

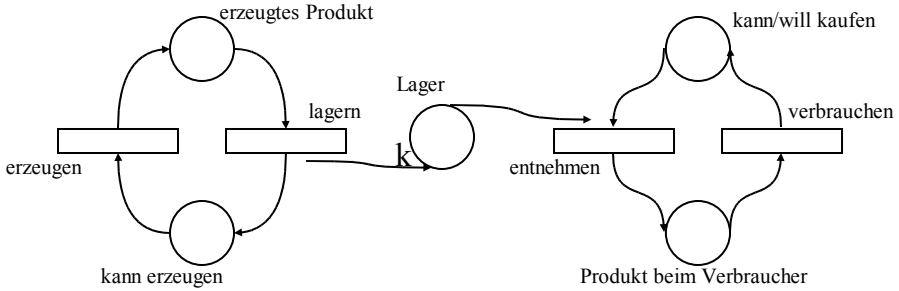
(4) $K: S \to \mathbb{N} \cup \{\infty\}$ ist die Kapazität für jede Stelle,

(5) $W: F \to \mathbb{N}$ ist die Gewichtsfunktion ("weight") der Kanten,

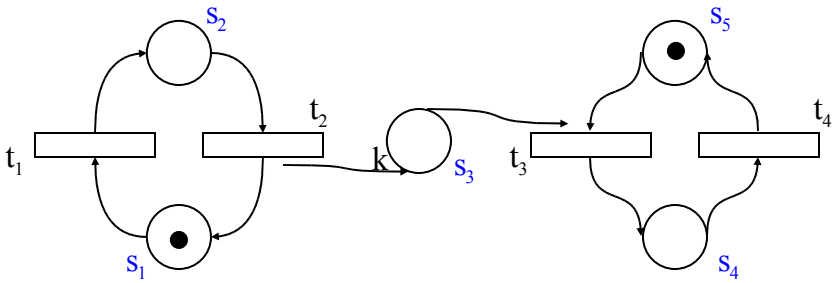
(6) $M_0: S \to \mathbb{N}_0 \cup \{\infty\}$ ist die Anfangsmarkierung, für die gelten muss: $\forall s \in S: M_0(s) \le K(s)$, d.h., in keiner Stelle dürfen mehr Marken liegen, als die Kapazität zulässt (die Markierungen schreibt man in der Regel als Vektoren).

Beispiel 13.1.3: Erzeuger-Verbraucher-Kreislauf (Producer-Consumer-Cycle)

Ein Erzeuger erzeugt ein Produkt, legt dieses in einem Lager, das maximal $k \ge 1$ Stellplätze besitzt, ab und wiederholt diesen Prozess.
Ein Verbraucher entnimmt ein Produkt aus dem Lager, konsumiert dieses und wiederholt diesen Prozess.

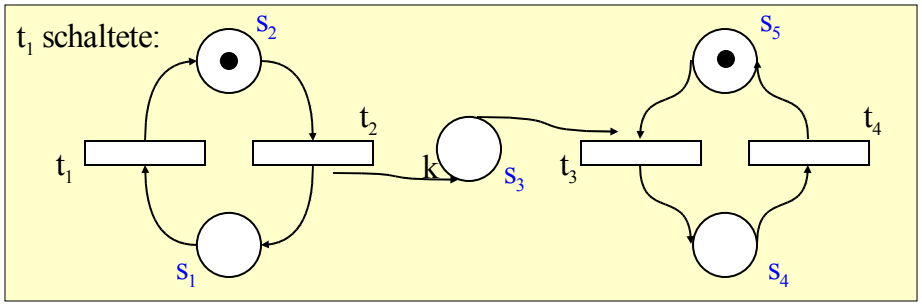


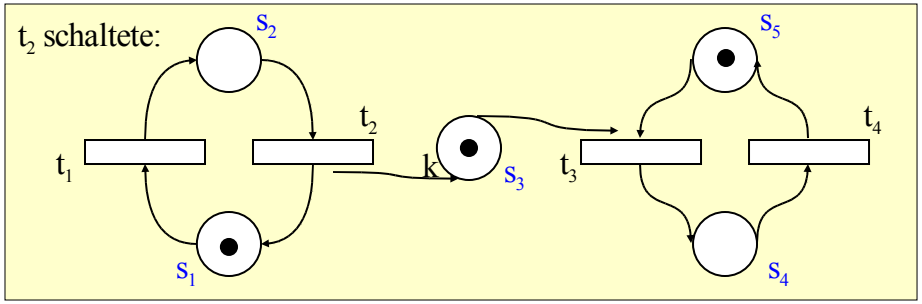
Hinweis: Das Lager hat die Kapazität k, alle anderen Stellen haben die Kapazität ∞.



Verbraucher-Erzeuger-System
mit Anfangsmarkierung $M_0 = (1,0,0,0,1)$.

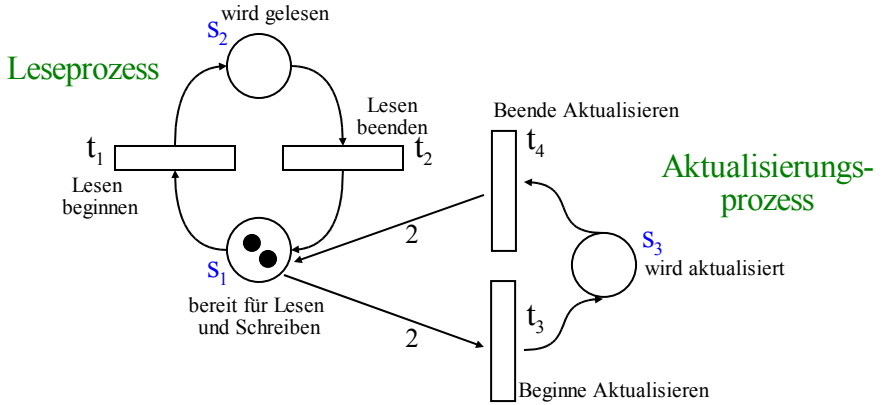
Formal: $S = \{s_1, s_2, s_3, s_4, s_5\}$, $T = \{t_1, t_2, t_3, t_4\}$, $M_0 = (1,0,0,0,1)$,
$F = \{(s_1,t_1),(s_2,t_2),(t_1,s_2),(t_2,s_1),(t_2,s_3),(s_3,t_3),(s_5,t_3),(t_3,s_4),(s_4,t_4),(t_4,s_5)\}$,
$W((x,y))=1$ für alle Kanten (x,y), $K(s_1)=K(s_2)=K(s_4)=K(s_5)=\infty$, $K(s_3)=k$.
In dieser Anfangssituation kann nur die Transition t_1 schalten.
Aus der Anfangsmarkierung (1,0,0,0,1) ensteht dann die Folge-Markierung (0,1,0,0,1). Nun kann t_2 schalten und es entsteht die Markierung (1,0,1,0,1). Jetzt können t_1 oder t_3 schalten, wobei die Markierungen (0,1,1,0,1) bzw. (1,0,0,1,0) entstehen usw.

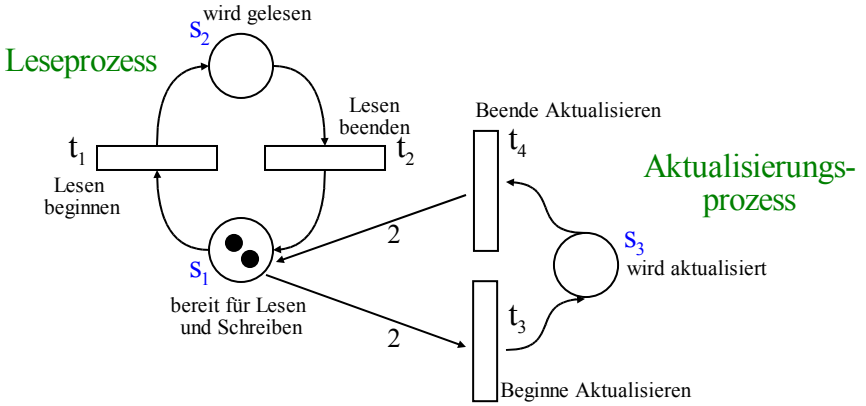




Beispiel 13.1.4: Lese-Schreib-Konflikt

Eine Datenbank steht zwei Benutzern zum Lesen zur Verfügung. Ab und zu sollen die Inalte von einem Autor aktualisiert werden; zu diesem Zeitpunkt darf nur der Autor auf die Datenbank zugreifen können. Modell hierzu?



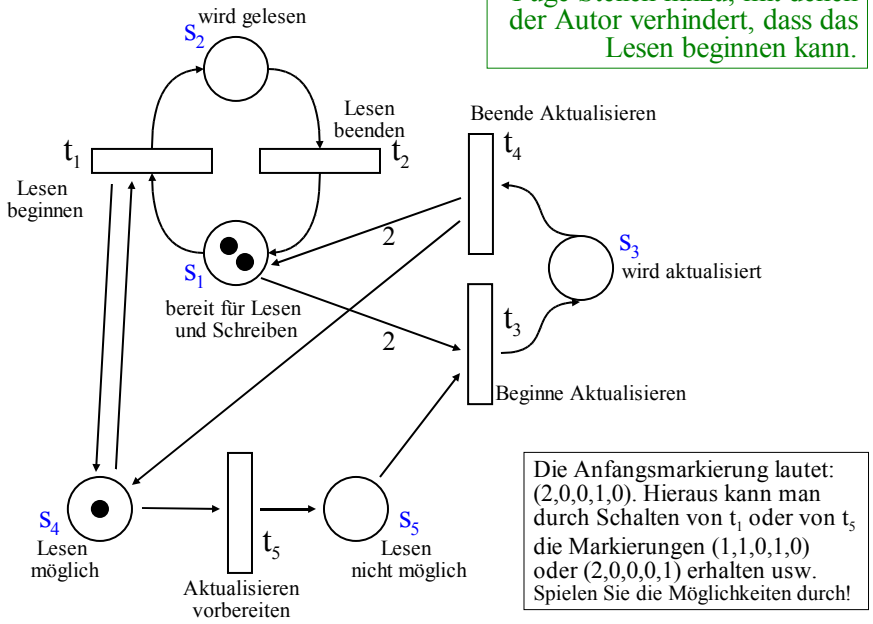


Nun kann eine "unfaire Aktionsfolge" auftreten:

$t_1\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2\, \ldots$

Hierbei ist stets mindestens eine Marke in der Stelle s_2, so dass niemals aktualisiert werden kann. Wie kann man erzwingen, dass der Autor das Lesen unterbrechen kann?

Füge Stellen hinzu, mit denen der Autor verhindert, dass das Lesen beginnen kann.



Die Anfangsmarkierung lautet: (2,0,0,1,0). Hieraus kann man durch Schalten von t_1 oder von t_5 die Markierungen (1,1,0,1,0) oder (2,0,0,0,1) erhalten usw. Spielen Sie die Möglichkeiten durch!

Fragen 13.1.5 a:

1. Erreichbarkeitsproblem: Wie kann man feststellen, ob eine angestrebte Situation (= Markierung) von einer gegebenen Situation (= Markierung) aus erreicht werden kann?
2. Beschränktheit: Wie kann man nachweisen, dass die Zahl der Marken in allen Stellen beschränkt bleibt?
3. Fairness: Wie kann man ermitteln und sicherstellen, dass keine "unfairen Aktionsfolgen" (= unfaire Folge von schaltenden Transitionen) auftreten kann?
4. Wir kann man beweisen, dass das Netz nicht in "Verklemmungen" gerät, also in eine Markierung, von der aus es nicht mehr weitergeht?
5. Wie stellt man sicher, dass das Netz nicht in "sinnlose Schleifen" (= Iteration von schaltenden Transitionen, die aus Sicht des Problems keinen Sinn machen) gerät?

Um diese Fragen beantworten zu können, müssen wir zuerst die Arbeitsweise und die erforderlichen Begriffe exakt definieren.

Definition 13.1.5 b: Begriffe und Schreibweisen

$N = (S, T, F, K, W, M_0)$ sei ein Stellen-Transitions-Netz.

(1) Jede Abbildung $M: S \to \mathbb{N}_0 \cup \{\infty\}$ heißt Markierung von N. M heißt zulässig, wenn für alle $s \in S$ gilt: $M(s) \le K(s)$.

(2) Eine Markierung schreibt man in der Regel als Spaltenvektor (oder in Texten auch als Zeilenvektor). Hierbei wird vorausgesetzt, dass die Menge der Stellen S geordnet ist: $S = \{s_1, s_2, s_3, \ldots, s_n\}$ mit $s_1 < s_2 < s_3 < \ldots < s_n$

(3) Für $x \in S \cup T$ heißen $^\bullet x = \{(y,x) \mid (y,x) \in F\}$ der Vorbereich von x und $x^\bullet = \{(x,y) \mid (x,y) \in F\}$ der Nachbereich von x.

(4) Die Flussrelation $F \subseteq (S \times T) \cup (T \times S)$ zusammen mit der Gewichtsfunktion W wird meist auf die gesamte Menge $(S \times T) \cup (T \times S)$ wie folgt fortgesetzt zu $W': S \to \mathbb{N}_0$

$W'((x,y)) :=$ if $(x,y) \in F$ then $W((x,y))$ else 0 fi.

(W' beschreibt F und W eindeutig.)

Definition 13.1.6: Arbeitsweise von S/T-Netzen

$N = (S, T, F, K, W, M_0)$ sei ein Stellen-Transitions-Netz.

(1) Eine Transition $t \in T$ heißt unter der Markierung M aktiviert $\Leftrightarrow \forall s \in {}^\bullet t: W((s,t)) \le M(s)$ und $\forall s \in t^\bullet: W((t,s)) + M(s) \le K(s)$. Man schreibt hierfür auch: $M[t>$

(2) *Schaltregel:* Es sei t eine Transition, die unter der zulässigen Markierung M aktiviert ist. Dann kann t schalten und es entsteht aus M die Folge-Markierung M' mit:

$$M'(s) = \begin{cases} M(s) & s \notin {}^\bullet t \text{ und } s \notin t^\bullet, \\ M(s) - W((s,t)) & s \in {}^\bullet t \text{ und } s \notin t^\bullet, \\ M(s) + W((t,s)) & s \notin {}^\bullet t \text{ und } s \in t^\bullet, \\ M(s) - W((s,t)) + W((t,s)) & s \in {}^\bullet t \text{ und } s \in t^\bullet. \end{cases}$$

[Wir hätten auch $M'(s) = M(s) - W'((s,t)) + W'((t,s)), \forall s \in S$ schreiben können, siehe Definition 13.1.5 (4).]

noch Definition 13.1.6:

(3) Schreibweise: Wenn t unter M aktiviert ist und nach dem Schalten von t die Markierung M' entsteht, so schreibt man $M[t>M'$ oder $M \xrightarrow{t} M'$.

(4) Fortsetzung der Relation $\cdot[>$ auf Folgen von Transitionen (also auf T^*; Wörter über T nennen wir auch Schaltfolgen):

$M[t_1 t_2 t_3 \ldots t_r> \Leftrightarrow$

es gibt Markierungen $M_1, M_2, \ldots, M_{r-1}$ mit $M[t_1 > M_1$, $M_1[t_2 > M_2$, $M_2[t_3 > M_3, \ldots, M_{r-2}[t_{r-1} > M_{r-1}, M_{r-1}[t_r >$.

noch Definition 13.1.6: Aktiviertheit und Erreichbarkeit

(5) Fortsetzung der Relation $\cdot[>\cdot$ auf Schaltfolgen (also auf Folgen von Transitionen):

$M[t_1 t_2 t_3 \ldots t_r> M' \Leftrightarrow$ es gibt Markierungen $M_1, M_2, \ldots, M_{r-1}$ mit $M[t_1 > M_1$, $M_1[t_2 > M_2$, $M_2[t_3 > M_3, \ldots, M_{r-1}[t_r > M'$.

(Im Falle r=0 muss M=M' sein.)

(6) $ERR(M) = \{M' \mid$ es existiert $t_1 t_2 t_3 \ldots t_r$ mit $M[t_1 t_2 t_3 \ldots t_r> M'\}$ heißt Erreichbarkeitsmenge bzgl. M.

(Beachte: $M \in ERR(M)$, insbesondere ist ERR(M) nie leer.)

$ERR(N) = \{M' \mid$ es existiert $t_1 t_2 t_3 \ldots t_r$ mit $M_0[t_1 t_2 t_3 \ldots t_r> M'\}$ heißt Erreichbarkeitsmenge des Netzes N.

Wenn M' in ERR(M) liegt, so sagt man auch, M' ist von M aus erreichbar.

Die Erreichbarkeit stellt die Bedeutung der S/T-Netze dar. Kennt man also den Erreichbarkeitsgraphen (siehe unten) eines S/T-Netzes im Detail, so kann man hieraus alle seine Eigenschaften ableiten. Wir werden dies an den Begriffen Beschränktheit, Lebendigkeit und Fairness demonstrieren.

Hinweise:

Oft schreibt man S/T-Netze nur in der Form $N = (S, T, F, M_0)$. In diesem Fall ist $K(s) = \infty$ für alle Stellen s und $W((x,y)) = 1$ für alle Kanten (x,y) einzusetzen.

Dies gilt auch für Zeichnungen: Fehlen die Angaben für K oder W an einer Stelle bzw. Kante, so ist unbeschränkte Kapazität bzw. Kantengewicht 1 gemeint.

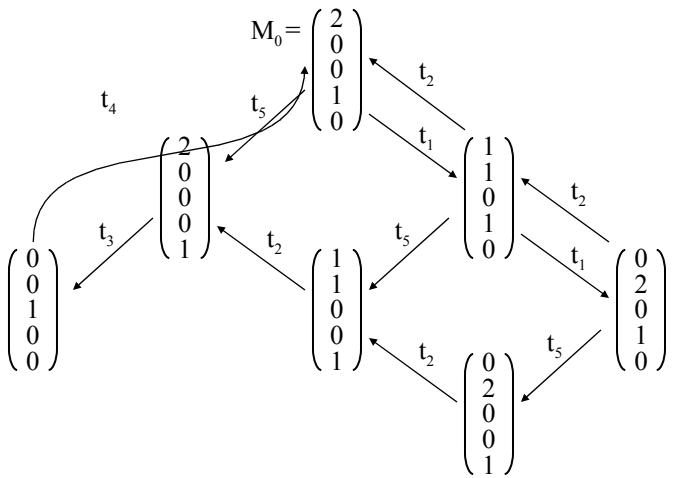
In diesem Abschnitt sprechen wir oftmals nur von einem "Netz" und meinen damit stets ein S/T-Netz.

Um die Arbeitsweise eines Netzes im Ganzen zu verstehen, konstruiert man schrittweise alle Markierungen, die man von der Anfangsmarkierung M_0 aus erreichen kann. Das Ergebnis ist der "Erreichbarkeitsgraph" des Netzes.

Dieser Graph kann unendlich oder endlich sein. Unter den endlichen Graphen interessieren vor allem diejenigen, die nicht allzu groß werden; also: Wenn d die Länge der Darstellung eines Netzes ist, dann soll die Länge der Darstellung des Erreichbarkeitsgraphen höchstens polynomiell bzgl. d sein.

Leider ist dies jedoch nur selten der Fall. Wir betrachten hierzu einige Beispiele und definieren den Erreichsbarkeitsgraphen eines Netzes formal.

13.1.7: Konstruktion des Erreichbarkeitsgraphen des letzten S/T-Netzes aus Beispiel 13.1.4: Ausgehend von M_0 werden suksessiv alle im nächsten Schritt erreichbaren Markierungen notiert:

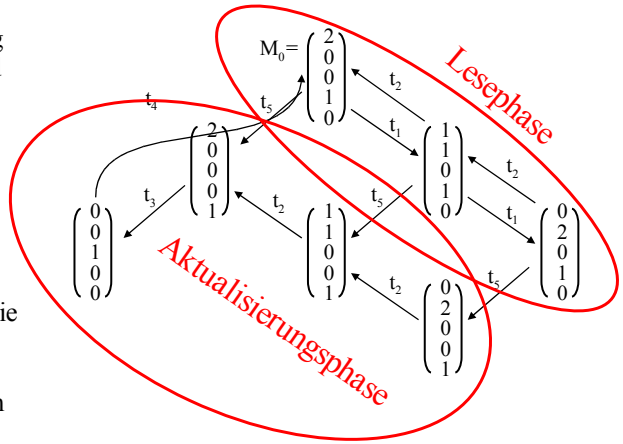


Dieser Graph besitzt einige Eigenschaften, die uns Hinweise geben, ob das angegebene Netz unser gestelltes Lese-Schreib-Problem tatsächlich löst:

1. Man erkennt die (korrekte) Trennung zwischen Lesen und Schreiben

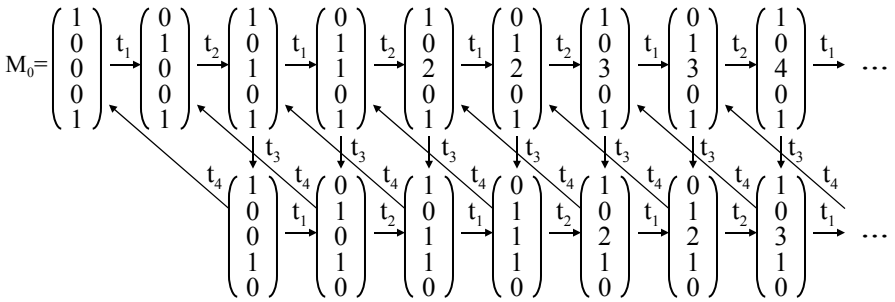
2. Man entdeckt eine "Invariante" der erreichbaren Markierungen M: $(1,1,3,1,1) \cdot M = 3$.

3. Man sieht, dass die Transitionenfolgen $(t_1t_2)^*$ und $(t_5t_3t_4)^*$ korrekt von M_0 nach M_0 führen.



4. Man erkennt, dass jede Transition irgendwann noch einmal schalten kann, dass also keine Transition irgendwann überflüssig wird.

13.1.8: Konstruktion des Erreichbarkeitsgraphen aus 13.1.3



Falls die in 13.1.3 angegebene Größe k eine natürliche Zahl ist, so besitzt der Erreichbarkeitsgraph genau 4k+2 Markierungen. Ist $k = \infty$, so ist der Ereichbarkeitsgraph unendlich groß.

Definition 13.1.9: Erreichbarkeitsgraph.

Es sei $N = (S, T, F, K, W, M_0)$ ein Stellen-Transitions-Netz mit der Erreichbarkeitsmenge ERR(N).

Der Erreichbarkeitsgraph G(N) des Netzes N ist ein gerichteter Graph mit der Knotenmenge ERR(N). Er besitzt in der Regel Mehrfachkanten, die dann aber mit *verschiedenen* Transitionen beschriftet sind. Eine Kante (M, M') mit der Beschriftung t existiert genau dann, wenn $M[t>M'$ gilt.

Formal: $G(N) = (ERR(N), \{(M, t, M') \mid M[t>M'\})$, wobei (M, t, M') eine Kante von der Markierung M zur Markierung M' mit Beschriftung t ist.

(M, t, M') zeichnet man in der Form $M \xrightarrow{t} M'$

[Hinweis: Zu jedem Netz gibt es genau einen Erreichbarkeitsgraphen. Es ist klar, wie man ihn schrittweise aufbaut.]

13.1.10: Beschränktheit eines Netzes

Ein Netz soll beschränkt heißen, wenn es eine natürliche Zahl k gibt, so dass keine Markierung im Netz erreicht werden, in der eine Stelle mehr als k Marken besitzt.

Definition: Sei $N = (S, T, F, K, W, M_0)$ ein S/T-Netz.

(1) Es sei k eine natürliche Zahl. Eine Stelle s des Netzes N heißt k-beschränkt, wenn für jede von M_0 aus erreichbare Markierung M gilt, M(s) ist nicht größer als k, d.h., $\forall M \in ERR(N): M(s) \le k$.

(2) N heißt k-beschränkt, wenn jede Stelle k-beschränkt ist, d.h., $\forall s \in S\ \forall M \in ERR(N): M(s) \le k$.

(3) s heißt beschränkt, wenn es eine natürliche Zahl k gibt, so dass s k-beschränkt ist, d.h., $\exists k \in \mathbb{N}\ \forall M \in ERR(N): M(s) \le k$.

(4) N heißt beschränkt, wenn jede Stelle von N beschränkt ist, d.h., $\exists k \in \mathbb{N}\ \forall s \in S\ \forall M \in ERR(N): M(s) \le k$.

13.1.11: Folgerung

Ein Netz ist genau dann beschränkt, wenn sein Erreichbarkeitsgraph endlich ist.

Beweis: Sei S die Menge der Stellen des Netzes N.

"$\Rightarrow$" Wenn N beschränkt ist, so gibt es ein k, so dass alle Markierungen des Erreichbarkeitsgraphen nur Komponenten besitzen, die kleiner oder gleich k sind. Dann kann es aber höchstens $(k+1)^{|S|}$ Markierungen in ERR(N) geben, d.h., der Erreichbarkeitsgraph ist endlich.

"$\Leftarrow$" Wenn der Erreichbarkeitsgraph endlich ist, dann existiert das Maximum m über alle Komponenten von Markierungen in ERR(N). Offenbar ist jede Stelle dann m-beschränkt, d.h., das Netz ist beschränkt.

13.1.12: Größe eines Netzes. Wie groß können Erreichbarkeitsgraphen werden, bezogen auf die Größe des zugehörigen Netzes?

Hierzu müssen wir zunächst festlegen, was die Größe eines Netzes ist. Wie üblich ist dies die Länge einer Darstellung.

Meist wählt man eine normierte Darstellung. Eine relativ kurze Darstellung ist z.B. die Folgende, bei der die Stellen stets mit den Zahlen von 1 bis n = |S| und die Transitionen mit den Zahlen von 1 bis m = |T| bezeichnet werden; die i-te Stelle beschreibt man, indem man vor die Nummer i ein 's' setzt; analog sei 't'j die j-te Transition:

<Zahl der Stellen>; <Zahl der Transitionen>;
<Liste der Kanten in der Form (<Bezeichnung Knoten>,
<Bezeichnung Knoten>, <Gewicht der Kante>) >;
<Kapazitäten als n-stelliger Vektor>;
<Anfangsmarkierung als n-stelliger Vektor>;;

Hierbei werden alle Zahlen binär aufgeschrieben.

Definition: Die Größe des Netzes ist Länge dieser Darstellung.

Ein Netz wird hier also als ein Wort über dem 7-elementigen Alphabet {s, t, 0, 1, ,, ;, ∞} aufgefasst.

Beispiel: In Beispiel 13.1.3 hatten wir folgendes Netz vorgestellt:
$S = \{s_1, s_2, s_3, s_4, s_5\}$, $T = \{t_1, t_2, t_3, t_4\}$, $M_0 = (1,0,0,0,1)$,
$F=\{(s_1,t_1),(s_2,t_2),(t_1,s_2),(t_2,s_1),(t_2,s_3),(s_3,t_3),(s_5,t_3),(t_3,s_4),(s_4,t_4),(t_4,s_5)\}$,
$W((x,y))=1$ für alle Kanten (x,y), $K(s_1)=K(s_2)=K(s_4)=K(s_5)=\infty$, $K(s_3)=k$.

Dieses Netz schreiben wir also für k=9 in folgender Form auf:

101;100;s1,t1,1,s10,t10,1,t1,s10,1,t10,s1,1,t10,s11,1,s11,t11,1,
s101,t11,1,t11,s100,1,s100,t100,1,t100,s101,1;∞,∞,1001,∞,∞;
1,0,0,0,1;;

Dieses Wort besteht aus 134 Zeichen. Unser Netz besitzt also die Größe 134.

Einschub: Übungsaufgaben:

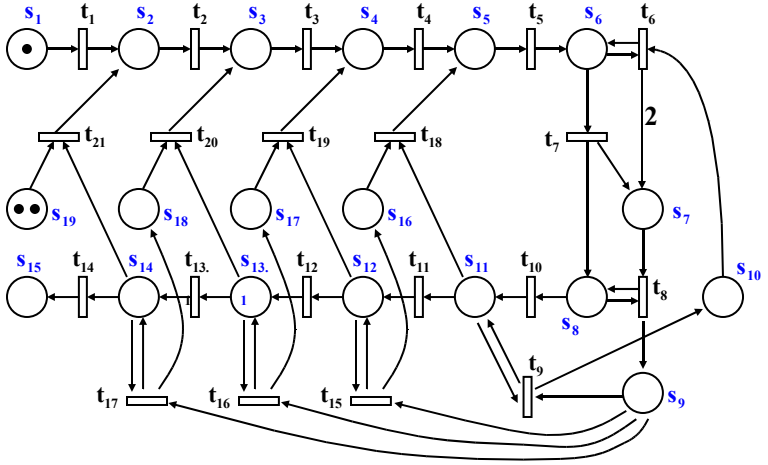
1. Welche Größen haben das "resultierende Netz" des Zug-Beispiels 13.1.1 und das letzte Netz aus Beispiel 13.1.4 (mit 5 Stellen und 4 Transitionen)?
2. Definieren Sie auf ähnliche Weise die Größe eines Erreichbarkeitsgraphen.
3. Welche Größen haben die Erreichbarkeitsgraphen der drei obigen Netze, für die die Größe bestimmt wurde? (Für zwei Beispiele wurden in 13.1.7 und 13.1.8 die Erreichbarkeitsgraphen bereits angegeben; für das Zugbeispiel müssen Sie diesen Graphen selbst konstruieren.)

4. Definieren Sie, was es bedeuten soll, dass eine Klasse von Netzen polynomiell konstruierbare Erreichbarkeitsgraphen besitzt. Wieviel Platz benötigt man für diese Konstruktionen höchstens?
5. Geben Sie unendlich große Klassen von Netzen an, deren Erreichbarkeitsgraphen sich genau mit linearem, bzw. mit quadratischem Zeitaufwand konstruieren lassen.
6. Es gibt Netze, deren Erreichbarkeitsgraphen exponentiell größer sind als die Netze selbst. Versuchen Sie, solche Netze selbst zu entdecken, oder durchsuchen Sie die Literatur hiernach.
7. Gibt es Klassen von Netzen, deren Erreichbarkeitsgraphen endlich sind, aber viel stärker als exponentiell wachsen?

Warnung zu Punkt 7: Die Erreichbarkeitsgraphen von S/T-Netzen können gewaltig wachsen.

Das S/T-Netz auf der folgenden Folie mit 19 Stellen und 21 Transitionen vollzieht die Berechnung der sog. Ackermann-Funktion A (in der fünften Stufe) nach. Dies ist eine totale berechenbare Funktion A: $\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, die schneller als jede Funktion wächst, die man nur mit elementaren Anweisungen, der Sequenz, der Alternative und der for-Schleife darstellen kann. Der Erreichbarkeitsgraph dieses S/T-Netzes ist endlich, besitzt aber mehr als

$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ (65535 mal) Knoten. Vollziehen Sie etwa 100 Schritte nach, um die Wirkungsweise des Netzes zu erahnen.



13.1.13: Lebendigkeit eines Netzes

Ein Netz soll lebendig heißen, wenn jede Transition irgendwann noch einmal schalten könnte. Genauer: Für jede Transition t muss gelten: Wenn man sich, ausgehend von M_0, in irgendeiner Markierung M befindet, dann muss von M aus eine Markierung M' erreichbar sein, unter der t aktiviert ist (also schalten kann).

Definition: Sei $N = (S, T, F, K, W, M_0)$ ein S/T-Netz.

(1) Eine Transition t des Netzes N heißt lebendig, wenn es zu jeder von M_0 aus erreichbaren Markierung M eine von M aus erreichbare Markierung M' mit M' gibt, d.h., $\forall M \in ERR(N)\ \exists M' \in ERR(M): M'[t>$.

(2) N heißt (stark) lebendig, wenn jede Transition von N lebendig ist.

Man könnte ein Netz auch lebendig nennen, wenn es immer weiterschalten kann. Dies ist gleichbedeutend damit, dass es keine von M_0 aus erreichbare Markierung gibt, zu der es keine Folge-Markierung gibt. Solche Markierungen bezeichnet man als Verklemmung.

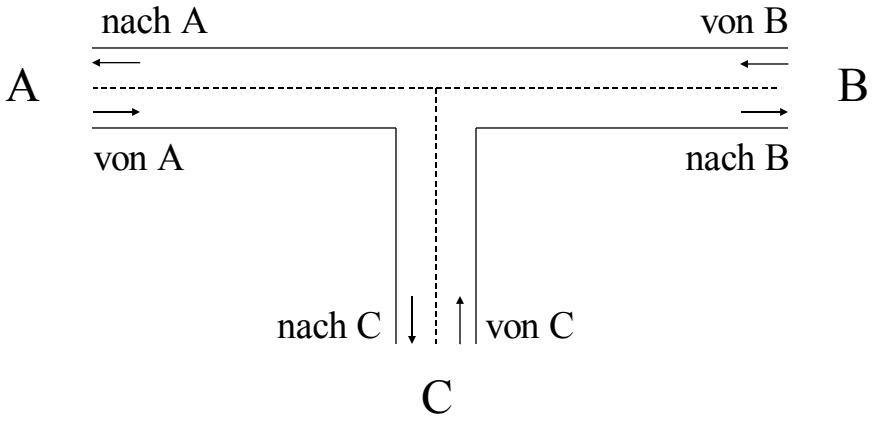
Fortsetzung der Definition:

(3) Eine Markierung M heißt Verklemmung ("Deadlock"), wenn unter M keine Transition aktiviert ist, d.h. $\neg\exists t \in T: M[t>$.

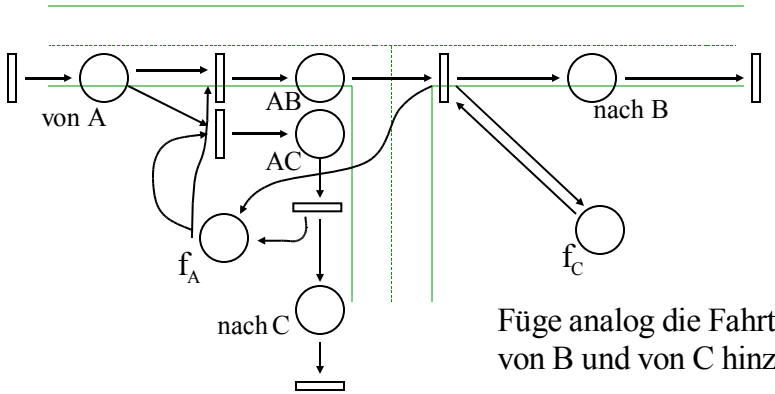
(4) Das Netz N heißt schwach lebendig, wenn es keine von M_0 aus erreichbare Verklemmung besitzt, d.h. $\forall M \in ERR(N)\ \exists t \in T: M[t>$.

13.1.14: Beispiel:

Straßenkreuzung mit Vorfahrtsregel "rechts vor links".

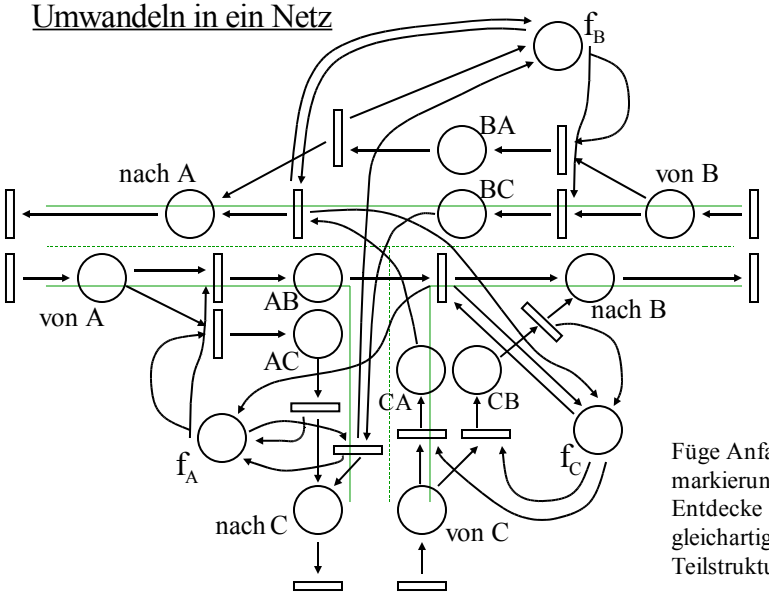


Umwandeln in ein Netz, wobei explizit "von rechts kommt niemand" durch die Stellen f modelliert wird. Wir betrachten zunächst nur die von A kommenden Fahrzeuge, für die nur wichtig ist, ob die Strecke von C zur Kreuzung frei ist (f_C).



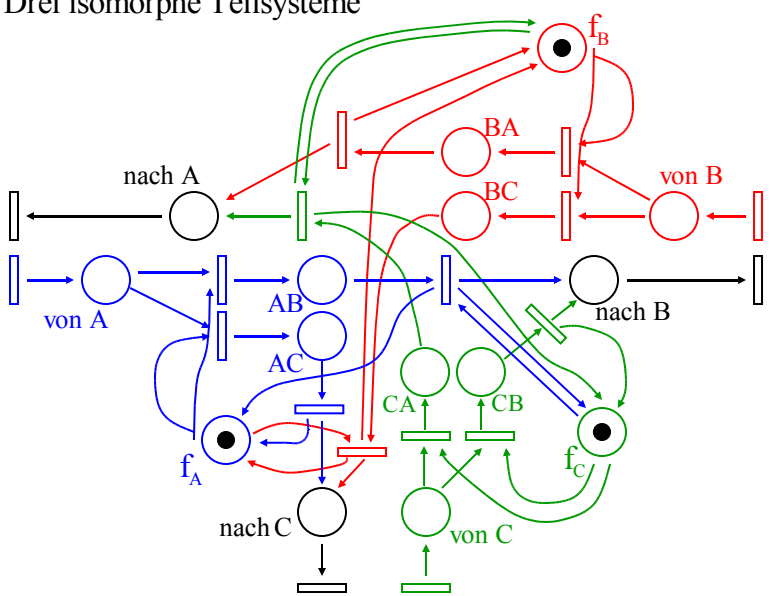
Füge analog die Fahrten von B und von C hinzu.

Umwandeln in ein Netz

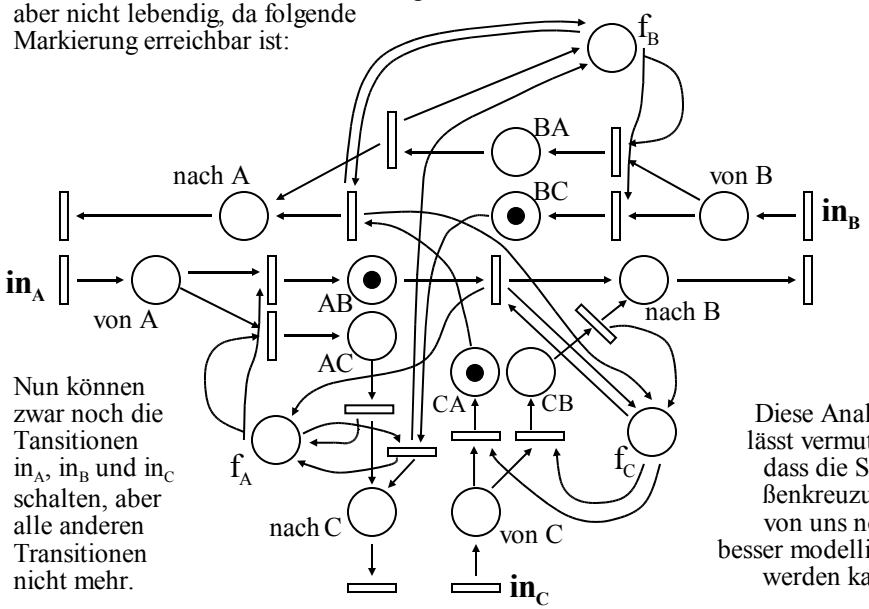


Füge Anfangsmarkierung hinzu. Entdecke gleichartige Teilstrukturen:

Drei isomorphe Teilsysteme



Dieses Netz ist zwar schwach lebendig, aber nicht lebendig, da folgende Markierung erreichbar ist:

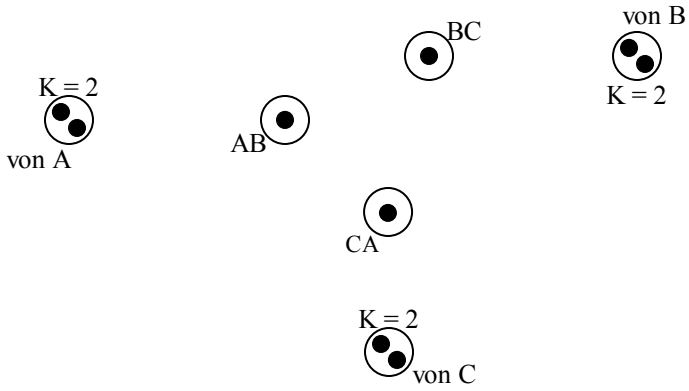


Nun können zwar noch die Tansitionen in_A, in_B und in_C schalten, aber alle anderen Transitionen nicht mehr.

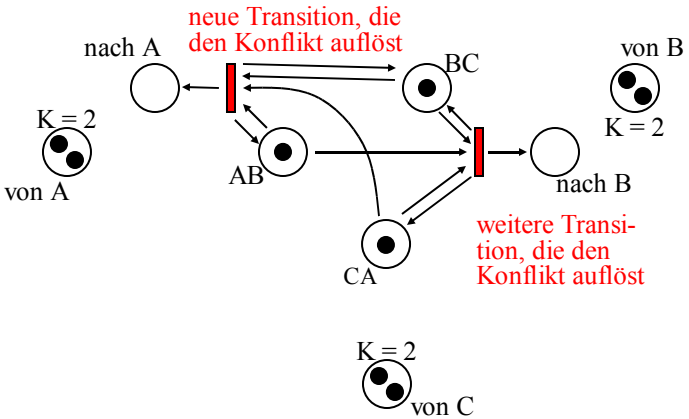
Diese Analyse lässt vermuten, dass die Straßenkreuzung von uns noch besser modelliert werden kann.

Wir sollten die Eingangs-Stellen "von A", "von B" und "von C" mit einer Kapazität, z.B. mit 2 belegen.

In diesem Fall gibt es eine Verklemmung, nämlich (wir zeigen nur den relevanten Ausschnitt aus dem Netz, also die Stellen der Verklemmung, die noch Marken tragen):



Diese Verklemmung können wir beseitigen, indem wir eine Transition einfügen, die in diesem Fall z.B. das von C nach A fahrende Fahrzeug als erstes über die Kreuzung lässt:



Wir können zusätzlich eine Transition hinzufügen, die das von A nach B fahrende Fahrzeug bevorzugt.

Als drittes können wir eine Transition hinzufügen, die das von B nach C fahrende Fahrzeug als erstes über die Kreuzung lässt.

Dies führt zu einem Netz, das relativ realistisch die Situationen an einer Kreuzung widerspiegelt. Es enthält alle Möglichkeiten, eine Verklemmung aufzulösen und unsinnige Überlastungen in gewissen Teilen des Netzes zu vermeiden.

Die Leser(innen) mögen diese Einzelheiten in das bereits vorhandene Netz eintragen und das neue Netz untersuchen.

Ende Beispiel 13.1.14

13.1.15: Wirkung des Schaltens einer Transition t_j:

$$\Delta t_j = \begin{pmatrix} W'((t_j,s_1)) - W'((s_1,t_j)) \\ W'((t_j,s_2)) - W'((s_2,t_j)) \\ W'((t_j,s_3)) - W'((s_3,t_j)) \\ \dots \\ W'((t_j,s_n)) - W'((s_n,t_j)) \end{pmatrix} \in \mathbb{N}_0^n \quad \text{mit } n = |S|$$

Wenn t_j schaltet, wird die Markierung genau um diesen Vektor verändert, d.h.: Aus $M[t_j>M'$ folgt $M' = M + \Delta t_j$.

Wir übertragen diese Aussage nun auf eine Folge von Transitionen. Hierzu sei $T = \{t_1, t_2, ..., t_m\}$.

Hierzu definieren wir für $w \in T^*$:
$\#_j w$ = Anzahl der t_j, die in w vorkommen.

Formal: $\#_j \varepsilon = 0$ und $\#_j vt = \underline{\text{if}}\ t_j = t\ \underline{\text{then}}\ \#_j v + 1\ \underline{\text{else}}\ \#_j v\ \underline{\text{fi}}$ $(\forall v \in T^*, \forall t \in T)$.

Dann gilt für jede Folge w von Transitionen mit $M[w>M'$:

$M' = M + \#_1 w \cdot \Delta t_1 + \#_2 w \cdot \Delta t_2 + \#_3 w \cdot \Delta t_3 \dots + \#_m w \cdot \Delta t_m$.

Dies formulieren wir nun in Matrizenschreibweise. Hierzu sei #w der (Spalten-) Vektor, dessen Komponenten $\#_1 w$, $\#_2 w$, ..., $\#_m w$ sind:

$$\#w = \begin{pmatrix} \#_1 w \\ \#_2 w \\ \#_3 w \\ \dots \\ \#_m w \end{pmatrix}$$

13.1.16 Definition:
Gegeben sei ein Netz $N = (S, T, F, K, W, M_0)$. Es seien $S = \{s_1, s_2, ..., s_n\}$ und $T = \{t_1, t_2, ..., t_m\}$. Die (n,m)-Matrix

$$C = (\Delta t_1, \Delta t_2, \Delta t_3, ..., \Delta t_m)$$

heißt Inzidenzmatrix des Netzes N.

13.1.17 Folgerung:

Für alle Markierungen M und M' und für alle Schaltfolgen $w \in T^*$ gilt:

Aus $M[w>M'$ folgt $M' = M + C \cdot \#w$.

13.1.18 Definition:

(1) Jeder (Zeilen-) Vektor $y \in \mathbb{Z}^n$ mit $y \cdot C = 0$ heißt S-Invariante.

(2) y heißt echte S-Invariante $\Leftrightarrow$ y ist eine S-Invariante mit nichtnegativen Komponenten und $y \neq 0$.

(3) Jeder (Spalten-) Vektor $x \in \mathbb{Z}^m$ mit $C \cdot x = 0$ heißt T-Invariante.

Beachte 13.1.17, so sieht man: Eine T-Invariante gibt an, wie oft die einzelnen Transitionen schalten müssen, damit die ursprüngliche Markierung wieder hergestellt wird.
Eine S-Invariante y besagt, dass der Wert $y \cdot M = y \cdot M'$ für alle von M aus erreichbaren Markierungen M' gleich ist.

13.1.19 Satz:

Sei N = (S, T, F, K, W, M_0) ein Netz.

(1) Sei y eine S-Invariante, dann gilt für alle M'∈ ERR(M): $y \cdot M = y \cdot M'$.

(2) Sei y eine echte S-Invariante. dann gilt für jede Stelle s von N, deren zugehörige Komponente in y positiv ist: s ist k-beschränkt für $k = y \cdot M_0$.

(3) Wenn es eine Markierung M und eine Schaltfolge w∈ T* mit M[w>M gibt, dann ist #w eine T-Invariante von N.

Dieser Satz folgt unmittelbar aus den bisherigen Ausführungen.

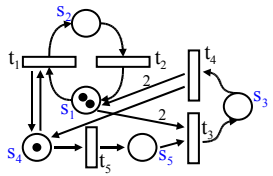
Bei (2) beachte man für die Komponente M(s):

$M(s) \le y(s) \cdot M(s) \le y_1 \cdot M_1 + y_2 \cdot M_2 + \ldots + y_n \cdot M_n = y \cdot M = y \cdot M_0 = k$

Aus der linearen Algebra wissen wir, dass die Menge der S- bzw. T-Invarianten einen Untervektorraum bildet.

Anwenden auf das Beispiel aus 13.1.4:

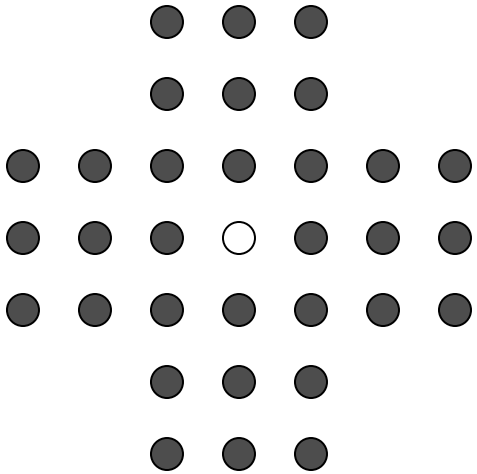
$$C = \begin{pmatrix} -1 & 1 & -2 & 2 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix} \quad M_0 = \begin{pmatrix} 2 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$



Eine echte S-Invariante lautet: y = (1, 1, 3, 1, 1). Nach Satz 13.1.19 (2) sind daher alle Stellen 3-beschränkt und somit ist auch das Netz beschränkt, siehe auch Beispiel 13.1.7.

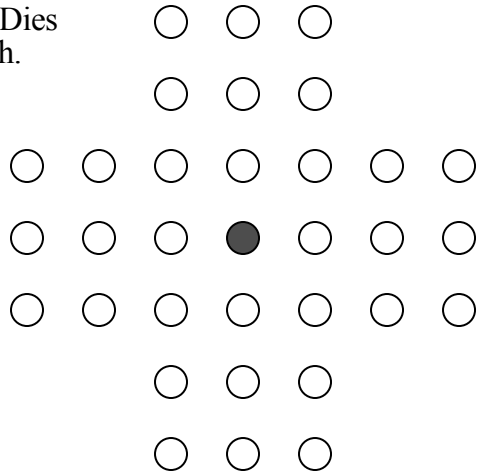
(1,1,0,0,0) und (0,0,1,1,1) sind T-Invarianten, daher verändern die Schaltfolgen t_1t_2 oder t_2t_1 bzw. $t_3t_4t_5$ oder $t_3t_5t_4$ oder $t_4t_3t_5$ oder $t_4t_5t_3$ oder $t_5t_3t_4$ oder $t_5t_4t_3$ (sofern sie schalten können) die Markierung nicht.

Beispiel 13.1.20: Solitaire-Spiel

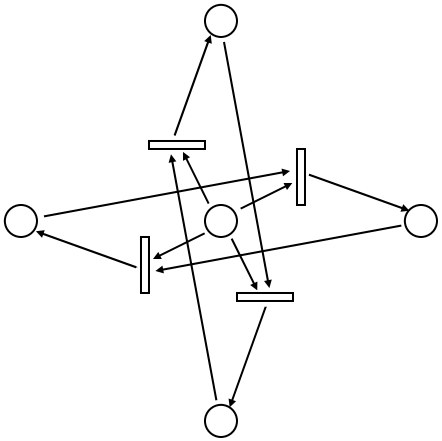


33 Plätze, 32 Steine, d.h., ein Platz bleibt frei. Ein Zug = Springe in gerader Richtung (nicht diagonal) über einen Nachbarstein auf einen freien Platz und entferne den übersprungenen Stein. Ziel: Am Ende soll nur noch ein Stein (möglichst auf einem vorgegebenen Platz) übrig sein.

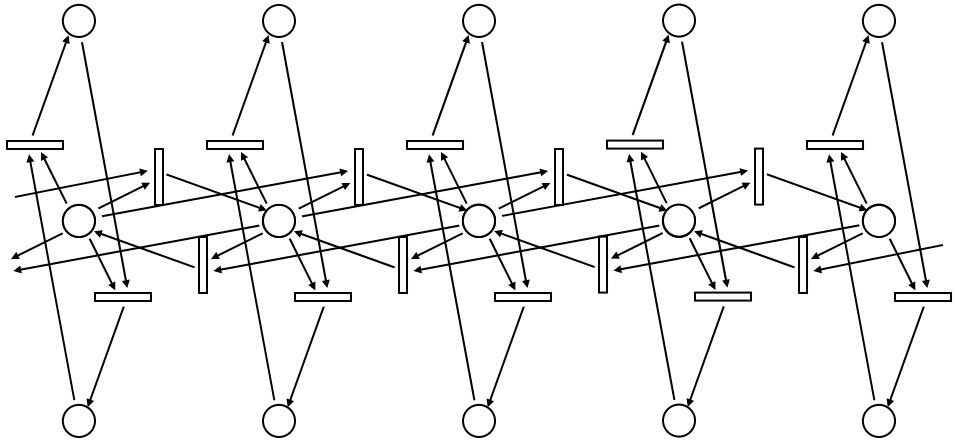
Meist versucht man, diese Endsituation zu erreichen. Dies geht tatächlich.

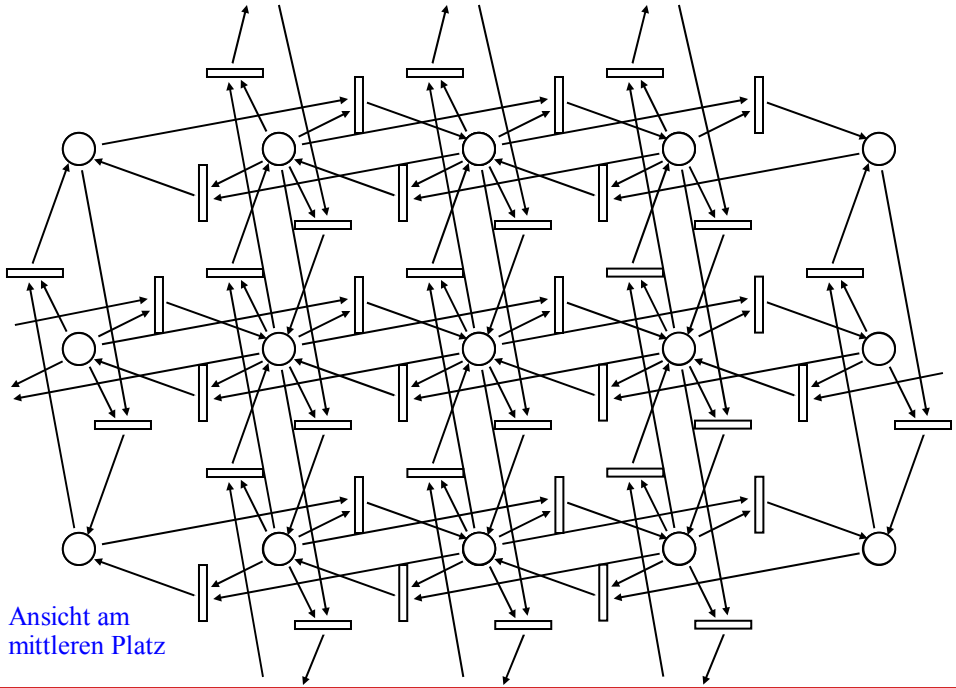


Solitaire als S/T-Netz formulieren. Betrachte einen Platz:

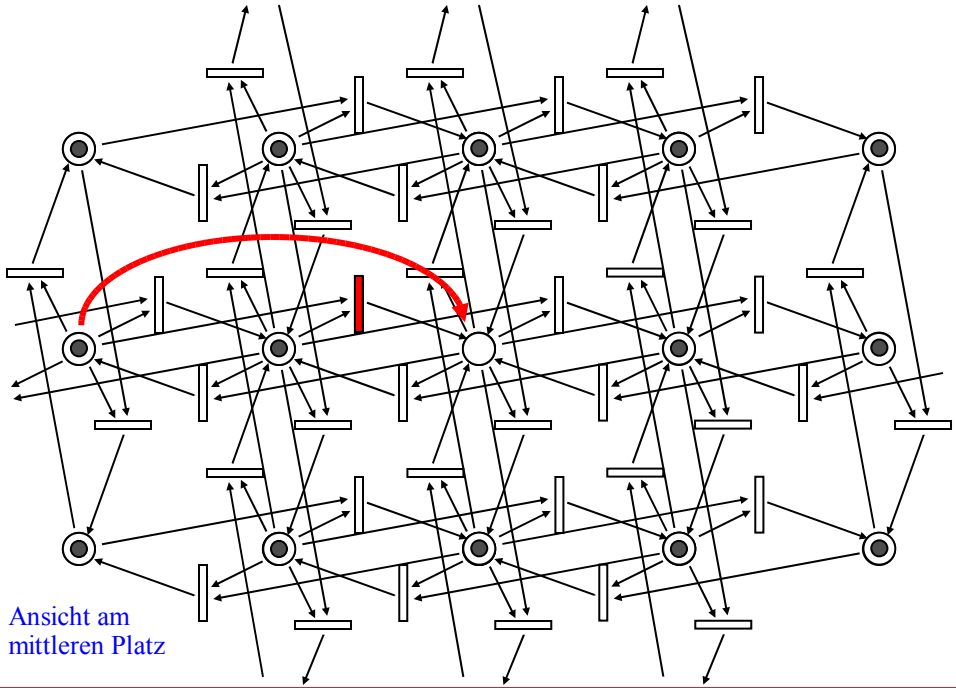


Plätze werden also Stellen, Transitionen sind das Überspringen.
Dieses Muster muss nun an jeder Stelle eingefügt werden.

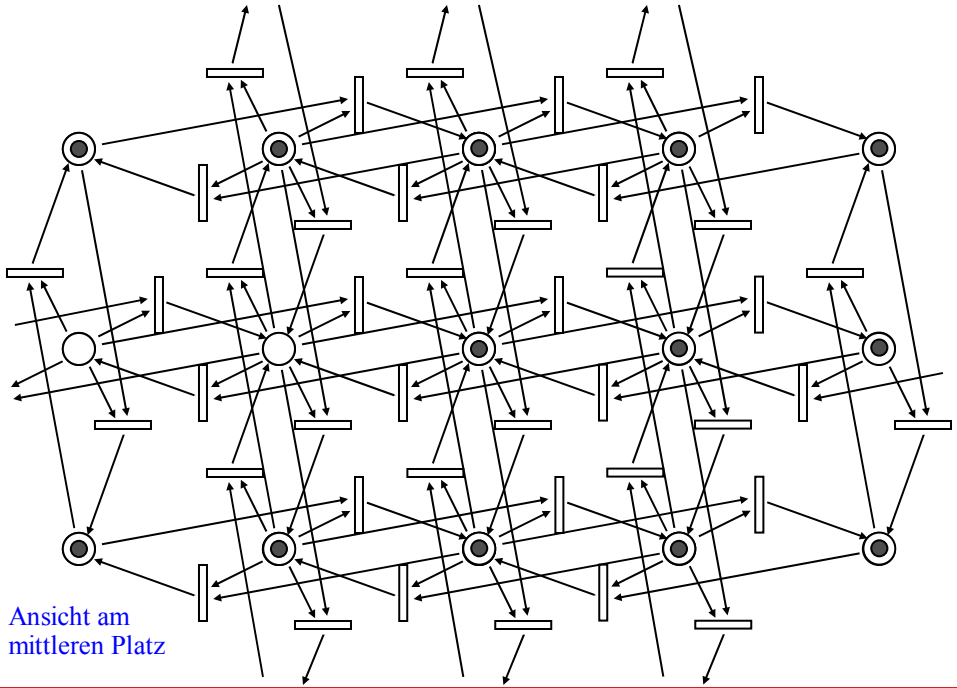




Ansicht am
mittleren Platz



Ansicht am
mittleren Platz



Ansicht am mittleren Platz

Das Ziel des Spiels besteht nun darin nachzuweisen, dass von der Anfangsmarkierung
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1

13.2.1 Elementare Anweisungen: Diese übernehmen wir aus 2.1.5 und fügen await hinzu:

skip	Nichtstun.
$X := \alpha$	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.)

13.2.2 Zusammengesetzte Anweisungen: Aus 2.1.5 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 or S_2 or S_3 or ... or S_n)

Nebenläufige Abarbeitung für n≥2:

(S_1 | S_2 | S_3 | ... | S_n)

Blöcke mit gemeinsamen Variablen:

[local <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:: [local <lokale Deklarationen>; (S_1 | S_2 | S_3 | ... | S_n)]

Beispiel 13.2.3:

```
[ 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 13.1.3, 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 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;

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

④(⑤L := L+1 or ⑤' skip)

Man kann auch die Berechnungen der Ausdrücke feiner unterteilen, also

④(⑤ L ⑦ := ⑥L+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.

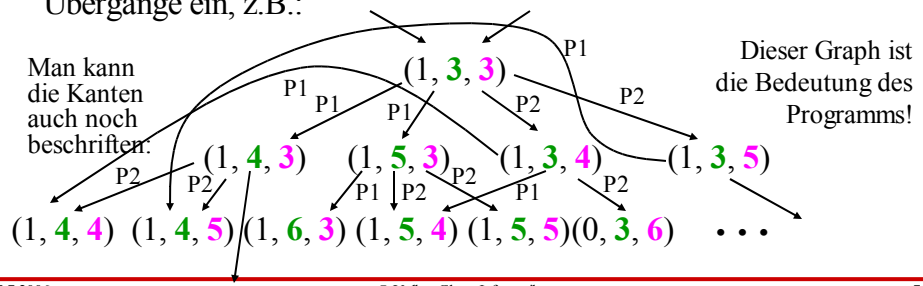
Wir formulieren nun die Menge der möglichen Zustände zu der Unterteilung, die auf der vorigen Folie angegeben wurde.

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

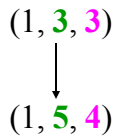
Z = { (a, **i**, **j**) | a ist der Wert von L, **i** ist die Stelle im Programm P1 und **j** ist die Stelle im Programm P2 }

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

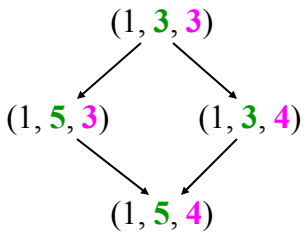


P1:: ① while ② true do ③ await L < max; (④ L := L+1 or ⑤ skip) ⑥ od;	P2:: ① while ② true do ③ await L > 0; (④ L := L-1 or ⑤ skip) ⑥ od;

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



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



13.2.4: 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).

13.2.5: 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:

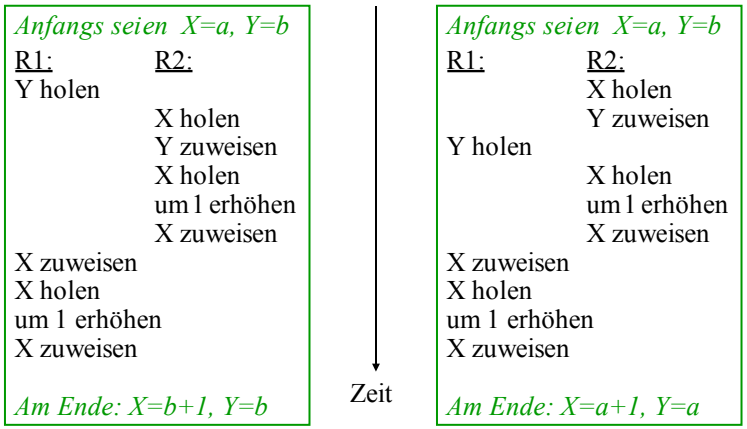
Hole den Wert a von L bzgl. Q1; hole den Wert a von L bzgl. Q2; bilde a+1 bzgl. Q1; bilde a-1 bzgl. Q2; weise a+1 der Variablen L zu bzgl. Q1; weise a-1 der Variablen L zu bzgl. Q2.

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

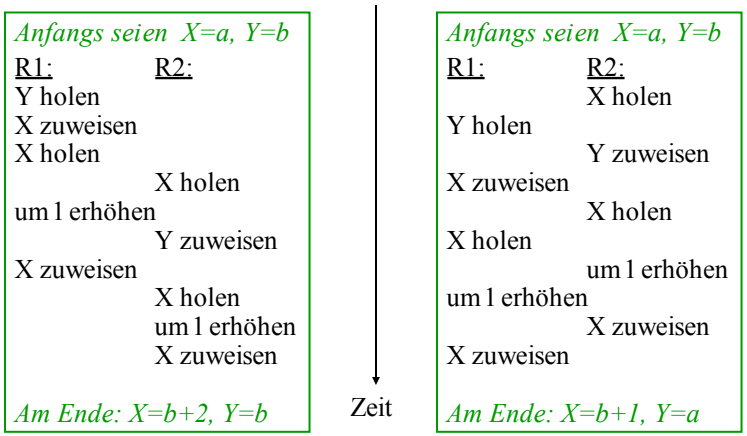
Sind überhaupt keine Anweisungsteile "nicht unterbrechbar", dann sind alle zeitlich verschachtelten Reihenfolgen möglich.

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:



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



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.

13.2.6 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).

13.2.7 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.

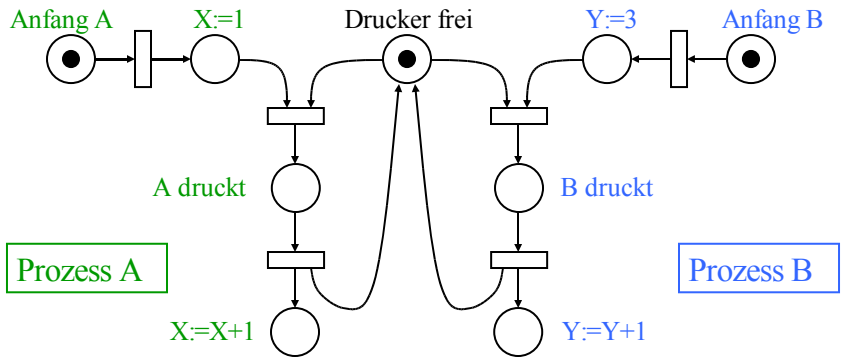
Wir beschreiben im Folgenden den wechselseitigen Ausschluss zuerst mit einem S/T-Netz. Anschließend realisieren wir ihn softwaremäßig durch geeignete Kontrollvariable in den beiden Prozessen.

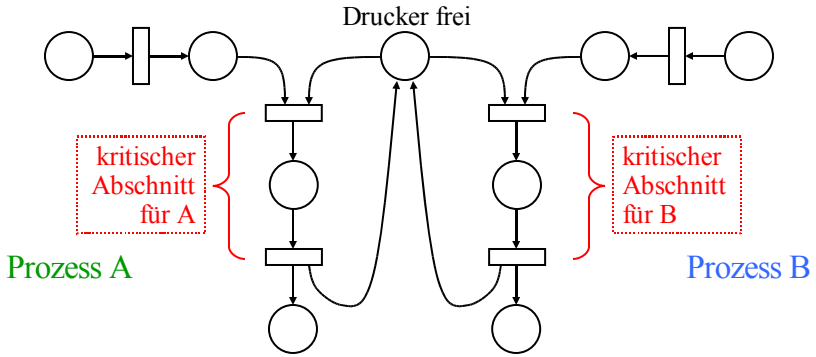
Beispielprogramm: Zwei Prozesse wollen X bzw. Y drucken.

[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.

13.2.8 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. Vorgänger waren bereits bei den alten Griechen in Gebrauch.

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.

13.2.8 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?

Dieses Programm verhindert zwar, dass sich beide Prozesse gleichzeitig in ihrem kritischen Abschnitt befinden können, aber wenn die Anweisungsfolgen (wie oft üblich) in einer unendlichen Schleife stehen, dann können die beiden Prozesse nur abwechselnd ihren kritischen Abschnitt betreten, und beide Prozesse müssen dauernd aktiv bleiben, sonst wartet der andere Prozess ewig:

Unbrauchbare Lösung

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

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


13.2.9 Problemformulierung: Gesucht wird also eine Softwarelösung, bei der jeder Prozess unabhängig vom anderen ist, außer in dem Fall, dass beide in einem gleichen Zeitraum in ihren kritischen Abschnitt eintreten wollen. Hierzu gibt es diverse Lösungen in der Literatur (z.B.: T. Dekker 1965, G. L. Peterson 1981, S. Owicki und L. Lamport 1982).

Versuchen Sie zunächst selbst, eine Lösung für Vorbereitung A bzw. B und Nachbereitung A bzw. B des allgemeinen Problems zu finden:

```
[ local <Deklarationen>;
  (A:: while true loop                |   B:: while true loop
         Vorbereitung A;              |         Vorbereitung B;
         kritischer Abschnitt A;      |         kritischer Abschnitt B;
         Nachbereitung A;             |         Nachbereitung B;
       end loop;                      |       end loop;    )     ]
```


Wir geben hier nur die Lösung von Peterson an. Jeder Prozess hat hierbei eine eigene Boolesche Variable PrA bzw. PrB, die den Wunsch, in den kritischen Abschnitt einzutreten, signalisiert. Zusätzlich gibt es eine Variable "dran", die dem anderen Prozess den Vortritt lässt, sofern dieser auch gerade in den kritischen Abschnitt will.

Diese Lösung erfüllt die geforderten Eigenschaften:

- Es kann sich zu jedem Zeitpunkt nur ein Prozess in seinem kritischen Abschnitt befinden.
- Jeder Prozess kann in seinen kritischen Abschnitt gelangen unabhängig davon, wo sich der andere Prozess befindet oder ob er noch aktiv ist.
- Es tritt keine Verklemmung auf.

13.2.10 Die Lösung von Peterson (1981):

(or ist hier das Oder in einem Booleschen Ausdruck)

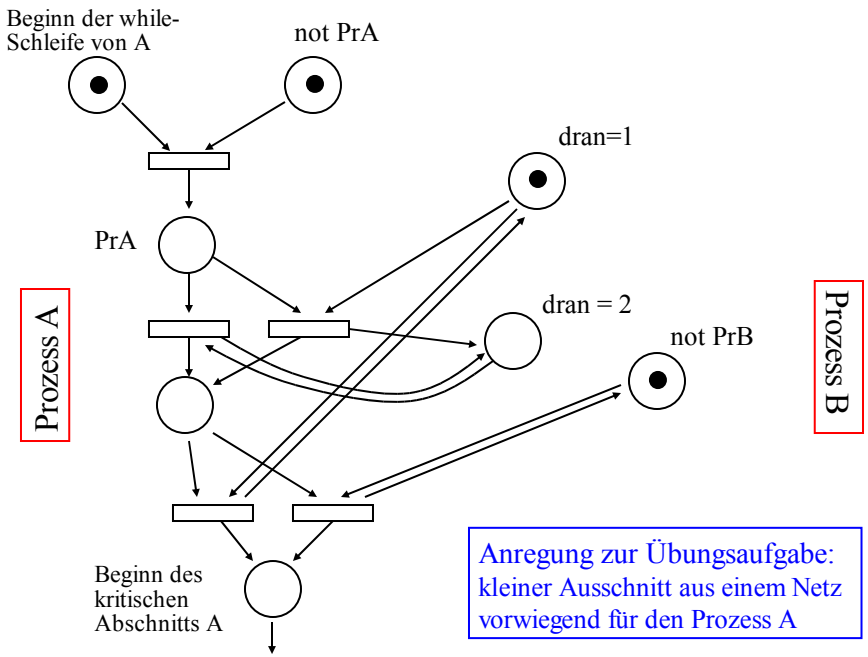
```
[ local  dran: Integer:=1;  PrA, PrB: Boolean:= false;

(A:: while true loop                    B:: while true loop
       PrA := true;                            PrB := true;
       dran := 2;                              dran := 1;
       await (not PrB) or (dran=1);            await (not PrA) or (dran=2);
       < kritischer Abschnitt A >;             < kritischer Abschnitt B >;
       PrA := false;                           PrB := false;
       ...                                     ...
     end loop;                               end loop;    )              ]
```

In der Vorlesung wird die Arbeitsweise dieses Vorgehens genauer erläutert. Machen Sie sich diese Arbeitsweise an einem Ablaufdiagramm klar!

Zu diesem Programm kann man ein Stellen-Transitions-Netz zeichnen, das die Arbeitsweise genau widerspiegelt. Einen Beweis für die Korrektheit der Peterson-Lösung kann man dann über dieses S/T-Netz führen.

Übungsaufgabe: Konstruieren Sie dieses S/T-Netz zu der oben angegebenen Peterson-Lösung.

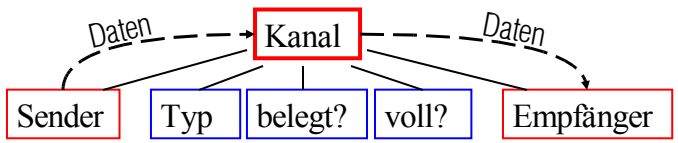


Anregung zur Übungsaufgabe:
kleiner Ausschnitt aus einem Netz vorwiegend für den Prozess A

13.2.11 Kanäle

Wir haben einige Aspekte des Nachrichtenaustausches vorgestellt, der über einen gemeinsamen Speicherbereich erfolgt. Anders funktioniert das Telefonieren: Dort wird jedem solchen Nachrichtenaustausch ein eigener Kanal zur Verfügung gestellt, der nach dessen Beendigung einer anderen Kommunkation zugeordnet werden kann.

Anstelle des Ablegens von Informationen in einem gemeinsamen Speicherbereich betrachten wir nun also die Nutzung von Kanälen, über die eine Verbindung zwischen zwei Partnern hergestellt werden kann.



uni-/ bidirektional, synchron / asynchron

Für Kanäle erlauben wir übergreifend den Datentyp
"channel [1..max] of T",
in den Daten d vom Typ T mittels CH ← d vom Sender hineingelegt und aus dem Daten dieses Typs vom Empfänger mittels CH → X der Variablen X zugewiesen werden können. Benutzen zwei Prozesse den gleichen Kanal, so muss einer Sender und einer Empfänger sein und es können Daten nur vom Sender an den Empfänger geschickt werden.

Ein Kanal ist wie eine Warteschlange organisiert und er besitzt in der Regel eine Kapazität. Die Daten, die zuerst hineingesteckt werden, kommen auch als erste wieder heraus (FIFO-Prinzip), und die Warteschlange kann meist nur die begrenzte Zahl "max" von Daten aufnehmen.

Mit einem Kanal muss weiterhin eine Boolesche Variable "belegt" verbunden sein, die einen Kanal nicht frei gibt, sofern er derzeit benutzt wird.

Die Anweisung CH ← X in einem Prozess P besagt also:

Wenn der Kanal CH nicht belegt ist, so wird er als belegt gekennzeichnet und P ist der Sender für CH;
wenn CH belegt ist und bisher nur der Empfänger dem Kanal zugeordnet ist, so wird P der Sender von CH;
wenn CH belegt ist und schon zwei Partner besitzt, so muss P unter diesen Partnern der Sender sein.

Trifft eine dieser drei Bedingungen zu, so wird geprüft, ob die Daten (hier: der Wert von X) vom Typ T sind und ob CH noch Platz für die Aufnahme eines weiteren Datums besitzt.

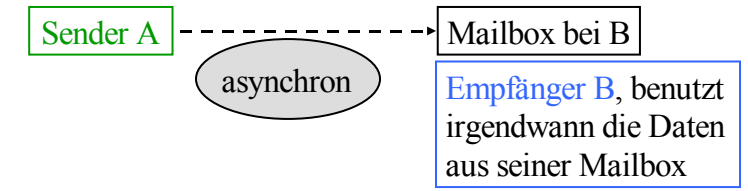
Trifft auch dies zu, so wird der Wert von X in den Kanal gelegt; anderenfalls erfolgt eine geeignete Fehlermeldung.

Analog ist die Anweisung CH → X in einem Prozess P zu interpretieren.

Will man einen Datenaustausch zwischen zwei Prozessen installieren, so muss man zwei Kanäle verwenden (wie beim Telefonieren). Will man den Zustand der Kanäle noch überwachen oder unabhängig von den Daten Kontrollinformationen übertragen, so muss man einen oder zwei weitere Kanäle hinzunehmen (wie beim Telefon).

Man muss noch festlegen, ob eine synchrone Verbindung besteht (wenn A sendet, so muss B zeitgleich empfangen; A kann erst weiterarbeiten, wenn B alle Daten empfangen hat) oder ob die Daten aynchron überliefert werden (z.B. in eine Mailbox gelegt werden; allerdings muss dann die Mailbox mit A oder mit dem Kanal synchron zusammenarbeiten).

13.2.12: Hieraus folgt: Eine asynchrone Kommunikation zwischen zwei Prozessen kann man durch zwei synchrone Kommunikationen zwischen drei Prozessen simulieren:



Ein solcher Fall liegt beim Erzeuger-Verbraucher-Kreislauf vor (siehe 13.1.3): Der Erzeuger schickt asynchron seine Produkte an den Verbraucher. Fasst man das Lager als zusätzlichen Prozess auf, so lässt sich dieser Vorgang synchron darstellen (siehe nächste Folie).

Wir wollen diese Ausführungen nun mit einem Beispiel beenden, an dem die prinzipielle Arbeitsweise von Kanälen abgelesen werden kann. Das Thema des Nachrichtenaustausches wird in Vorlesungen über Betriebssysteme, Verteilte Systeme und Sichere Systeme weiter vertieft.

Als Beispiel wählen wir den Erzeuger-Verbraucher-Kreislauf.

13.2.13 Beispiel EL, LV: channel [1..1] of T;

```
local X: T;
while true do
"erzeuge X";
EL ← X
od
```

Erzeuger

```
local Z: T; k: 0..10 :=0;
puffer: array(1..10) of T;

while true do
    ( if k<10 then
          EL → Z; k:=k+1;
          puffer(k) := Z fi
    or
    if k>0 then
          LV ← puffer(k);
          k := k-1 fi )
od
```

Lager

```
local Y: T;
while true do
LV → Y;
"verbrauche Y"
od
```

Verbraucher

Noch einige Begriffe:

Geblockte Übertragung: Daten werden meist nicht einzelnen, sondern in größeren Einheiten (Blöcken) übertragen. Dies erhöht vor allem die Effizienz der Übertragung.

Gepackte Daten: Daten werden oft noch komprimiert, damit sie weniegr Platz benötigt. Beim Empfänger müssen sie dann wieder "entpackt" werden (Beispiel: zip-Files).

Synchron: Der Empfänger übernimmt die Daten, während der Sender sendet.

Asynchron: Die Daten werden irgendwo gepuffert, bis der Empfänger sie abholt. Das Puffern kann auch im Kanal integriert sein.

Blockierendes Senden: Der Sender kann erst weiterarbeiten, wenn alle gesendeten Daten entweder beim Empfänger angekommen sind oder vom Puffer des Kanals aufgenommen wurden.

Blockierendes Empfangen: Der Empfänger kann erst weiterarbeiten, wenn alle gesendeten Daten bei ihm abgespeichert sind.

Den Datenaustausch mittels synchronem blockierendem Senden und blockierendem Empfangen bezeichnet man als Rendezvous. Dies ist in Ada realisiert, siehe 13.3.

13.3 Nebenläufigkeit in Ada

13.3.1 Einführende Begriffe

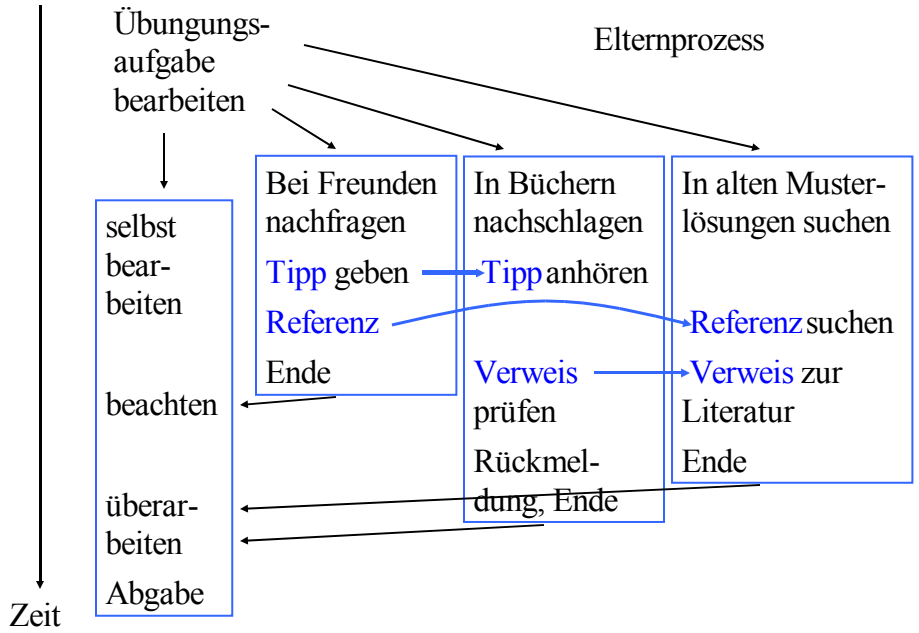
Prozess = Abarbeitung eines Algorithmus, wobei nur immer höchstens eine Stelle im Algorithmus aktiv ist.

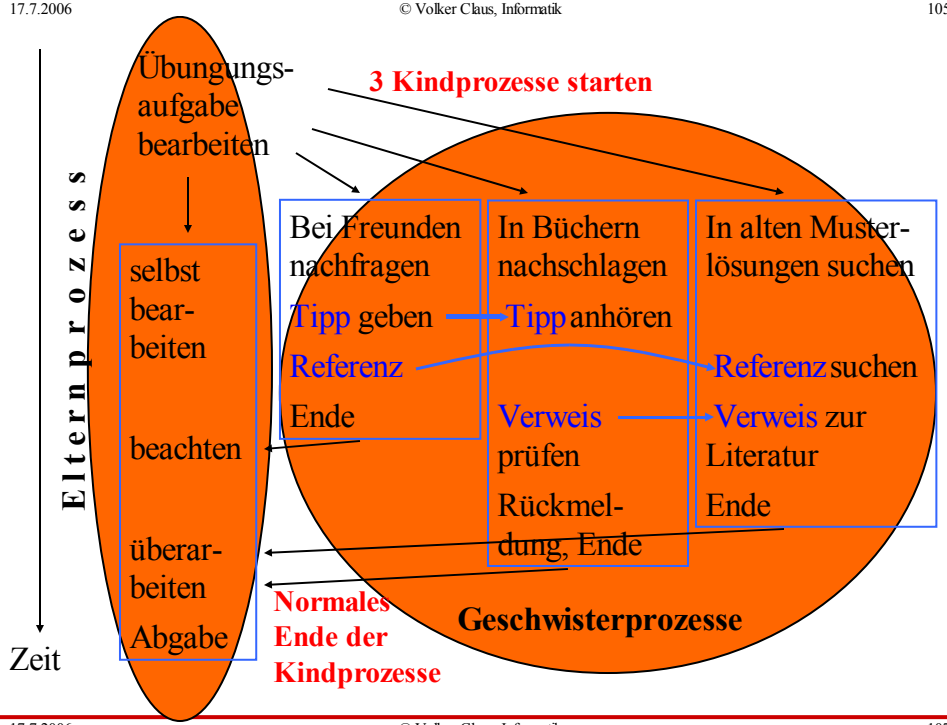
Einen Prozess kann man mit einem einzelnen Prozessor (Monoprozessor) abarbeiten. Er läuft sequentiell ab. Es können viele Prozesse gleichzeitig aktiv sein (Nebenläufigkeit).

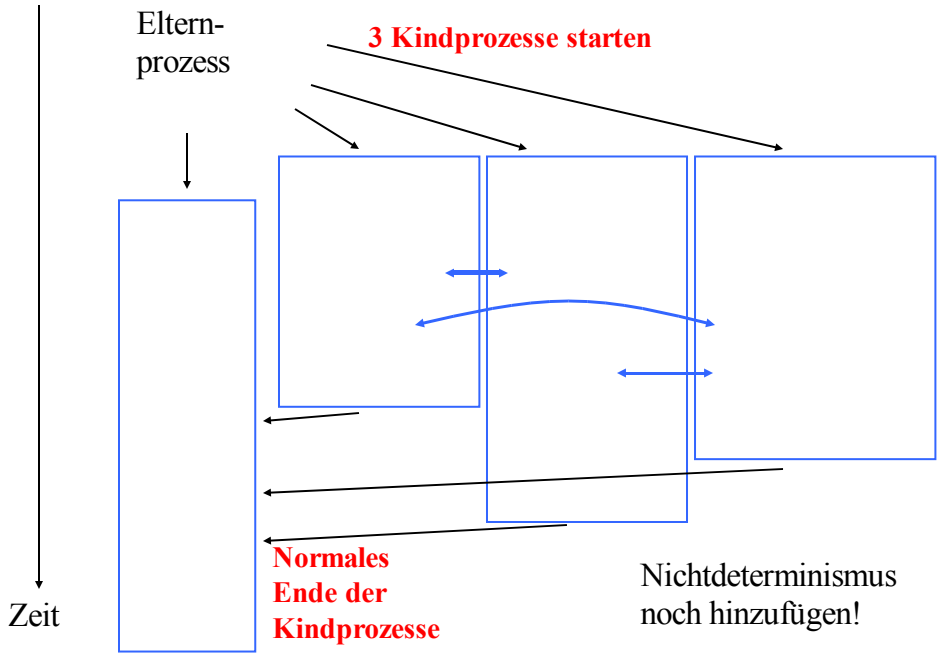
Das Programmstück, das einen Prozess beschreibt, nennen wir *Prozesseinheit*. Schlüsselwort in Ada hierfür: task. Sie kann in einem Deklarationsteil vereinbart werden. Sie ist in die Spezifikation und die Implementierung (task body) geteilt. Der Nachrichtenaustausch zwischen Prozessen erfolgt implizit durch Kanäle (zwischen einem Entry-Aufruf und einem accept).

Eine Prozesseinheit wird in Ada gestartet, sobald ihre Deklaration im Programmablauf erreicht wird (impliziter Start); es gibt in Ada keine Anweisung der Form "starte Prozess X".
Die Einheit, in der eine Prozesseinheit deklariert wird, heißt *Elternprozess*; andere im gleichen Deklarationsteil vereinbarte Prozesseinheiten heißen *Geschwisterprozesse*.

In einer Prozesseinheit kann es mehrere Stellen geben, an denen eine Synchronisation oder ein Datenaustausch erfolgen soll. Diese Stellen bezeichnet man als Entry-Schnittstellen; sie erhalten einen (Entry-) Namen und der Ablauf, der bei der Synchronisation erfolgen soll, wird in einer accept-Anweisung genau ausformuliert. Die accept-Anweisung trägt den Entry-namen; für eine Synchronisation wird sie wie eine Prozedur aufgerufen, erhält evtl. aktuelle Parameter und kann Werte über out-Variablen zurückgeben.







Wir benötigen also Sprachelemente für

Deklaration eines Prozesses: task (Spezifikation und Rumpf).

Starten eines Prozesses: Erfolgt implizit mit der Deklaration.

Normales Ende eines Prozesses: Erreichen von end
(ein Elternprozess endet aber frühestens, wenn alle seine Kindprozesse beendet sind) oder eigene Beendigung mittels terminate (in select-Anweisung).

Datenaustausch zwischen den Prozessen:
entry für die Spezifikation des Austausches
entry-Aufruf und accept für die Programmstellen

Gewaltsames Abbrechen eines Prozesses: abort

Nichtdeterministische Auswahl: select ... or ... or ... end select

Warten in nichtdeterministischen Alternativen: delay [until]

13.3.2 a Syntax

```
single_task_declaration ::=
        task defining_identifier [is task_definition];
task_definition ::= {task_item} [private {task_item}]
        end [task_identifier]
task_item ::= entry_declaration | representation_clause
task_body ::= task body defining_identifier is
        declarative_part
        begin
        handled_sequence_of_statements
        end [task_identifier];
```

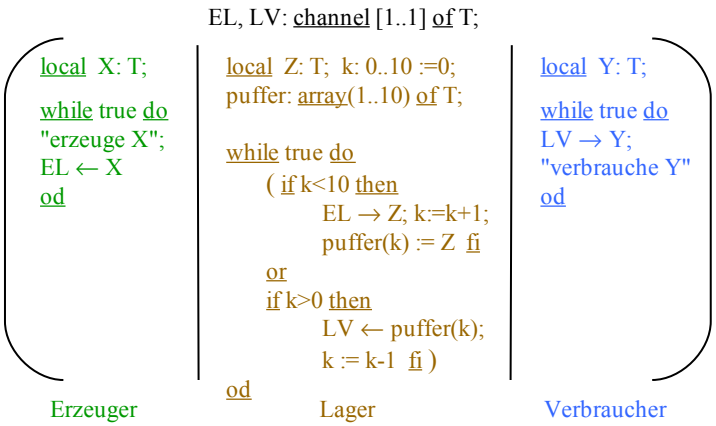

13.3.2 b Syntax (*Zu "statement" siehe 2.8.5 ff)*

```
entry_declaration ::= entry defining_identifier
        [(discrete_subtype_definition)] parameter_profile;
(parameter_profile beschreibt den Parameterteil in einer
Programmeinheit, vgl. 3.1.1)
entry_call_statement ::= entry_name [actual_parameter_part];
(actual_parameter_part  beschreibt die aktuellen
Parameter)
accept_statement ::=
        accept entry_direct_name [(entry_index)]
        parameter_profile
        [do handled_sequence_of_statements
        end [entry_identifier]];
(zu handled_sequence_of_statements siehe 3.1.1,
 insbesondere ist diese Folge von Anweisungen nie leer)
entry_index ::= expression
```


13.3.2 c Syntax

```
select_statement ::= selective_accept | timed_entry_call |
        conditional_entry_call | asynchronous_select
selective_accept ::= select [guard] select_alternative
        { or [guard] select_alternative }
        [ else sequence_of_statements ] end select;
guard ::= when condition =>
select_alternative ::= accept_alternative | delay_alternative |
        terminate_alternative
accept_alternative ::= accept_statement [sequence_of_statements]
delay_alternative ::= delay_statement [sequence_of_statements]
terminate_alternative ::= terminate;
delay_statement ::= delay_until_statement |
        delay_relative_statement
delay_until_statement ::= delay until delay_expression;
delay_relative_statement ::= delay delay_expression;
```

13.3.3 Beispiel: Der Nachrichtenaustausch erfolgt in Ada über Kanäle. Wir übertragen daher Beispiel 13.2.13:



DasProgramm wird zu procedure ... und die drei Prozesse werden zu task Erzeuger, task Lager und task Verbraucher.

Die Kanäle spielen keine direkte Rolle. Ihre Namen können als Aufrufstellen im jeweiligen Prozess verwendet werden, sofern kein Namenskonflikt entsteht.

Die Sendeoperation "←" wird durch einen Entryaufruf zu der entsprechenden Stelle ersetzt, die Empfangsoperation "→" wird zu einer accept-Anweisung.

Verwendet man die gleichen Bezeichner, so wird
aus EL ← X; der Aufruf Lager.EL(X); und
aus EL → Z; wird accept EL(U: in Float) do Z := U; end;
(hier wurde willkürlich U als Name für den formalen Parameter der Aufrufstelle gewählt; dieser muss nun natürlich auch in der Spezifikation von task Lager für diesen entry benutzt werden).

Wir fügen für den Abbruch noch eine Boolesche Variable Ende und für die Interaktion eine Character-Variable C hinzu.


```
procedure Erzeuger_Verbraucher is
        Ende: Boolean := false; C: Character;
        task Erzeuger;
        task Lager is
                entry EL(U: in Float);
        end;
        task Verbraucher;
                entry LV(W: in Float);
        end;
        < Die drei Taskrümpfe, siehe unten, hier einfügen >
begin   Put("Erzeuger, Lager und Verbraucher sind aktiv.");
        while not Ende loop get(C); Ende := C='0'; end loop;
        Put("Erzeuger, Lager und Verbraucher enden nun.");
end Erzeuger_Verbraucher;
```


Anstelle der Endlosschleife verwenden wir die Schleife mit der Bedingung "not Ende". Als "Produkt" nehmen wir reelle Zahlen.

```
task body Erzeuger is
X: Float := 1.0;
begin
while not Ende loop        -- Schleife, um ständig Zahlen mit der
    X := Sin(X+1.0);       -- Variablen X neu zu erzeugen.
    Lager.EL(X);           -- Entry-Aufruf: X wird ans Lager gesandt.
end loop;
end Erzeuger;              -- Schluss, falls "Ende" den Wert true
erhält.
```

```
task body Lager is                    -- fast wörtliche Übertragung nach Ada
Z: Float; K: Integer range 0..10 := 0;
Puffer: array(1..10) of Float;
begin
while not Ende loop                   -- statt der Endlosschleife
   select                             -- "(" wird zu select
      when K < 10 =>                  -- das if wird in ein when umgewandelt
         accept EL(U: in Float) do Z := U; end EL;
         K := K+1; Puffer(K) := Z;
   or
      when K > 0 =>                   -- das if wird in ein when umgewandelt
         Verbraucher.LV(Puffer(K));
         K := K-1;
   end select;                        -- ")" wird zu end select
end loop;
end Lager;                            -- Schluss, falls "Ende" den Wert true erhält.
```



```
task body Verbraucher is

Y: Float;
begin
while not Ende loop
      accept LV(W: in Float) do Y := W; end LV;
      < hier können Anweisungen zur Verarbeitung von x
folgen>
end loop;
end Verbraucher;
```


13.3.4: Synchronisation und Kommunikation zweier Prozesse in Ada (1):

Die Interaktion zweier Prozesse erfolgt in Ada durch "Entry-Schnittstellen". Ein Entry ist eine durch accept bezeichnete Stelle in einer Prozesseinheit. Diese kann wie ein Unterprogrammruf von einem anderen Prozess aufgerufen werden, allerdings erfolgt keine Verzweigung des Programmablaufs zu dieser Stelle, sondern es wird eine Synchronisation vorbereitet: Der aufrufende Prozess wartet an der Entry-Aufrufstelle solange, bis der gerufene Prozess die Entry-Schnittstelle erreicht hat (= Synchronisation). Dann wird der hinter accept zwischen do und end stehende Programmteil, der wie ein Unterprogrammrumpf aufgebaut sein kann, ausgeführt. Erst danach trennen sich die beiden Prozesse wieder, d.h., sie fahren unabhängig von einander mit ihrer jeweils nächsten Anweisung fort. Diesen Vorgang bezeichnet man als *Rendezvous*.

Ein Entry kann von einem beliebigen anderen Prozess aufgerufen werden. Rufen mehrere Prozesse gleichzeitig die gleiche Entry-Schnittstelle, so werden diese Aufrufe nacheinander abgearbeitet, wobei der wechselseitige Ausschluss für den do-end-Programmteil garantiert wird, d.h., dieses Programmstück kann nicht durch weitere Entry-Aufrufe unterbrochen werden.

Synchronisation und Kommunikation zweier Prozesse in Ada (2):

Trifft umgekehrt ein Prozess auf einen seiner Entry-Schnittstellen (also auf ein accept), so wartet er dort, bis ein anderer Prozess dieses Entry aufruft. Erfolgt dieser Aufruf, so wird der zwischen do und end stehende Programmteil ausgeführt (in dieser Zeit wartet der aufrufende Prozess nichtstuend) und anschließend trennen sich die beiden Prozesse wieder.

Man beachte, dass das Rendezvous mit dem Ende der accept-Anweisung endet. Danach folgende Anweisungen wirken sich nur auf die Wartezeit weiterer aufrufender Prozesse aus. In unserem Beispiel 13.3.3 endet das Rendezvous an der Entry-Schnittstelle EL in der Prozesseinheit Lager mit dem end nach dem accept. Die anschließend folgenden Anweisungen K := K+1; Puffer(K) := Z; gehören nicht mehr zum Rendezvous.

Hierdurch wird offensichtlich eine *Synchronisation* erreicht. Zugleich können durch den Entry-Aufruf, der aktuelle Parameter enthalten kann, Daten an den gerufenen Prozess übergeben werden und umgekehrt können am Ende des do-end-Programmstücks Daten zum rufenden Prozess zurückfließen, falls out-Variablen übergeben wurden. Dadurch wird eine *Kommunikation* zwischen den Prozessen realisiert.

Synchronisation und Kommunikation zweier Prozesse in Ada (3):

Durch diesen Mechanismus kann es vorkommen, dass ein rufender Prozess oder ein Prozess, der für einen Aufruf bereit ist, ewig wartet. (Dann ist der Programmierer "selbst schuld", siehe aber unten (5).)

Man beachte die *Asymmetrie des Rendezvous-Mechanismus:* Während der rufende Prozess genau weiß, mit wem er eine Kommunikation durchführt (der Entry-Aufruf besteht ja aus dem Namen des Tasks, gefolgt von einem Punkt und dem Namen der Entry-Schnittstelle; fehlt der Name des Tasks, so ist stets der Elternprozess gemeint), kennt der gerufene Prozess den Namen seines Partners nicht. Um die rufenden Prozesse korrekt bedienen zu können, muss mit jeder Entry-Schnittstelle eine Warteschlange für rufende Prozesse verbunden sein: Ein Prozess, der auf ein accept trifft, führt ein Rendezvous mit dem ersten in der Warteschlange stehenden Prozess durch (oder wartet, bis ein Aufruf eintrifft).

In der Syntax gibt es die Möglichkeit, Entries als geheim einzustufen: task_definition ::= ... [private {task_item}] ... . Dies wird man dann tun, wenn diese Entry-Schnittstellen nur von Unterprozessen genutzt werden sollen.

Synchronisation und Kommunikation zweier Prozesse in Ada (4):

Weiterhin lässt die Syntax bei der entry-Deklaration einen diskreten Untertyp [(discrete_subtype_definition)] und beim accept-statement einen [(entry_index)] zu. Diese beiden Varianten werden nur wichtig, wenn man "Entry-Familien" benutzt (dies sind viele Entries, die alle eine ähnliche Spezifikation besitzen).

Hinweise zur accept-Anweisung: Diese enthält keinen Deklarationsteil; man kann aber einen hinzufügen, indem man zwischen do und end Blöcke verwendet. Der do-end-Teil kann fehlen; in diesem Fall dient der Entry-Aufruf nur zur Synchronisation. Weiterhin kann es sinnvoll sein, die gleiche Entry-Schnittstelle an mehreren Stellen im Prozess zu platzieren; es sind daher zu einem Entry-Namen mehrere accept-Anweisungen erlaubt.
Wie üblich sind goto- und exit-Anweisungen, die von außen in eine accept-Anweisung gelangen oder eine accept-Aweisung verlassen, verboten. Allerdings ist ein return erlaubt, wodurch die umfassende Prozedur und zugleich das laufende Rendezvous beendet wird. Es gibt viele weitere Einschränkungen, siehe hierzu die Ada-Literatur.

Synchronisation und Kommunikation zweier Prozesse in Ada (5):

Natürlich muss es Mechanismen geben, um ein "unverschuldetes" unendliches Warten zu beenden. In Ada sind dies für einen rufenden Prozess:

timed_entry_call der Form
select <Entry-Aufruf> <und evtl. weitere Anweisungen>
or <delay-Alternative> <und evtl. weitere Anweisungen> end select

Bedeutung: In der delay-Alternative kann man eine Zeit vorgeben; hat bis dahin das Rendezvous des "Entry-Aufruf" nicht begonnen, so wird der Aufruf abgebrochen und die hinter der delay-Alternative aufgeführetn Anweisungen werden durchgeführt.

conditional_entry_call der Form
select <Entry-Aufruf> <und evtl. weitere Anweisungen>
or <Folge von Anweisungenm, evtl. leer> end select

Bedeutung: Ist der gerufene Prozess nicht zum sofortigen Rendezvous bereit, so wird der or-Teil ausgeführt.

Ebenso gibt es Abbruchmöglichkeiten für den Prozess, der an einer Entry-Schnittstelle auf ein Rendezvous wartet.

13.3.5: (Eingeschränkter) Nichtdeterminismus in Ada (1)

Es gibt die Select-Anweisung mit den or-Alternativen. Diese Anweisung ist *vorwiegend für die nichtdeterministische Behandlung von Entry-Aufrufen bzw. accept-Anweisungen* vorgesehen, siehe Syntax 13.3.2 c.

In der Praxis ist Nichtdeterminsimus (also willkürliches Verhalten von Prozessen an gewissen Stellen des Kontrollflusses) schwer kalkulierbar. Insbesondere können nicht alle Folgemöglichkeiten durchprobiert oder vorhergesehen werden. Entscheidet sich ein Prozess falsch, so kann er in Verklemmungen geraten.

Die select-Anweisung besitzt daher Varianten, um eine Entscheidung bzgl. der Kommunikation hinauszuzögern, bis bessere Informationen vorliegen, um eine Entscheidung, die nicht erfolgreich war, rückgängig zu machen oder durch eine andere zu ersetzen oder um eine eingeschlagene Alternative gewaltsam abzubrechen. Hierfür kann man einschränkende Bedingungen (sog. Wächter, "guards") vor die Alternativen setzen (when ... => ...), man kann die Auswahl davon abhängig machen, ob innerhalb einer Zeitspanne (delay <Zeit in Sekunden>) eine Alternative nicht erfolgreich war, man kann den eigenen Prozess zum Abbruch anbieten (terminate) oder man kann einen Prozess gewaltsam abbrechen (abort).

(Eingeschränkter) Nichtdeterminismus in Ada (2)

Betrachte ein einfaches Beispiel:

```
select
    accept Bildschirmausgabe (...) do ... end; ...
or
    when <Bildschirm an> => delay 300.0;
    <rufe Bildschirmschoner> ...
end select;
```

Liegt für 300 Sekunden kein Rendezvous mit dieser Entry-Schnittstelle "Bildschirmausgabe" vor, so wird die zweite Alternative ausgewählt.

Steht in einer Alternative das Sprachsymbol terminate, so wird dies einem "übergeordneten" Prozess, der zum Ende kommen möchte, als Möglichkeit angeboten, den laufenden Prozess an dieser Stelle zu beenden.

Bei den vielen Möglichkeiten muss man zwischen der rufenden (aktiver Prozess) und der gerufenen Seite (passiver Prozess) unterscheiden, für die ein Sprachelement unterschiedliche Bedeutung haben kann. Einzelheiten siehe Ada-Literatur und zum Teil im Programmierkurs.

13.3.6 Beispiel (nach M. Nagl)

Wir geben ein Beispiel für objektorientiertes Vorgehen in Ada an. Dieses stammt aus dem Buch von Manfred Nagl, "Softwaretechnik mit Ada95", dort Kapitel 5.4.

Ziel ist der Entwurf eines Warnsystems, das mit unterschiedlichen Dringlichkeitsstufen arbeitet. Das System wird zunächst in konventioneller Weise präsentiert und anschließend nach objektorientiert neu geschrieben.
(Hier wird nur eine Skizze des "Warnsystems" vorgestellt. Nähere Erläuterungen erfolgen in der Vorlesung, nachlesbar im o. g. Buch, welches einen guten Einblick in das praktische Programmieren mit Ada gibt.)

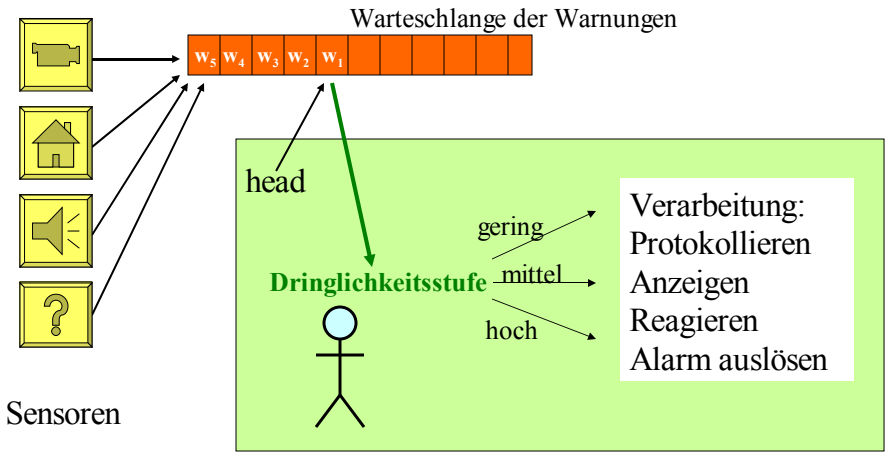
Warnsystem, konventionelle Darstellung

Gegeben seien zwei Pakete, die wir im Folgenden benutzen:

package Kalender is ...
    Hier wird ein Typ Zeit bereitgestellt (Jahr, Monat, Tag, Stunde, Minute, Sekunde, Hunderstel-Sekunde), und eine parameterlose Funktion Uhrzeit vom Ergebnistyp Zeit.

package Personalverwaltung is ...
    Hier wird ein Typ Person bereitgestellt sowie eine parameterlose Funktion Aufsichtführender vom Ergebnistyp Person.

Warnsystemskizze



```
package Warnsystem is
type Dringlichkeit is (gering, mittel, hoch);
type Warnung (D: Dringlichkeit) is
   record  Ankunftszeit: Kalender.Zeit;
           Nachricht: String;
           case D is
           when gering => null;
           when mittel | hoch =>
              Verantwortlicher: Personalverwaltung.Person;
              case D is
              when hoch => Alarm_ausgeloest: Kalender.Zeit;
              when others => null;
              end case;
           end case;
   end record;
```


-- Fortsetzung von *package* Warnsystem *is*

```
type Anzeigegeraet is (Drucker, Bildschirm, Wandanzeige);
procedure Anzeigen (W: in Warnung; AG: in Anzeigegeraet);
procedure Mitprotokollieren (W: in Warnung);
procedure Alarm_Ausloesen (W: in out Warnung);
procedure Reagieren (W: in out Warnung);
end Warnsystem;
```

Eine Prozedur soll beispielhaft implementiert werden. Wir betrachten hier "Reagieren", die für jede Dringlichkeitsstufe anders arbeitet; sie ist im "package body Warnsystem is ..." zu definieren:


```
procedure Reagieren (W: in out Warnung) is
begin
   W.Ankunftszeit := Kalender.Uhrzeit;
    Mitprotokollieren (W);
   Anzeigen (W, Drucker);
   case W.D is
        when gering => null;
        when mittel | hoch =>
            W.Verantwortlicher:=Personalverwaltung.Aufsichtführender;
            Anzeigen(W, Bildschirm);
            case W.D is
            when hoch =>  Anzeigen (W, Wandanzeige);
                          Alarm_Ausloesen (W);
            when others => null;
            end case;
   end case;
end Reagieren;
```


Nachteile:

Unübersichtlich.

Schwierig zu korrigieren

und schwierig zu warten; jede Korrektur oder Erweiterung erfordert vermutlich eine komplette Neuübersetzung.

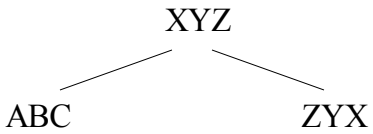
Beispiel: Man möchte eine neue Dringlichkeit "Notfall" höchster Dringlichkeit einführen.

Neuer Ansatz in Ada mit tagged Typen.

13.3.7: Warnsystem, objektorientierte Darstellung

Vorbemerkung: tagged Typen in Ada, siehe 4.5.1. Schema:

```
type XYZ is tagged record ... end record;
type ABC is new XYZ with record ... end record;
type ZYX is new XYZ with null record;
```



All dies kann offen oder geheim stattfinden:

```
type Vor is tagged record ... end record;
type Nach is new Vor with record ... end record;   oder
type Nach is new XYZ with private;   und später steht dann:
private  type Nach is new Vor with record ... end record;

type Vor is tagged private;          und später steht dann:
private  type Vor is tagged record ... end record;
```

Die abgeleiteten Typen aus Vor *können wiederum offen oder geheim deklariert werden usw.*

Variablen in Ada müssen einen festen Datentyp haben:

```
type Vor is tagged record H: Integer; end record;
type Nach is new Vor with record K: Float; end record;
type Aehnlich is new Vor with null record;
X: Vor := 17;  Y: Nach := (50, 33.6);  Z: Aehnlich := 8;
begin  X := Y;        -- verboten wegen strenger Typisierung in Ada
       X := Vor(Y);       -- erlaubt, "Konversion" (=Projektion)
       Y := (X with 415.37);  -- erlaubt, Erweitern
       Z := X;        -- verboten wegen strenger Typisierung in Ada
       Z := (X with null record);    -- erlaubt, Erweitern
       X := Vor(Z);       -- erlaubt, "Konversion" (=Projektion)
       ...          -- stets alle hinzukommenden Komponenten auflisten!
end;
```



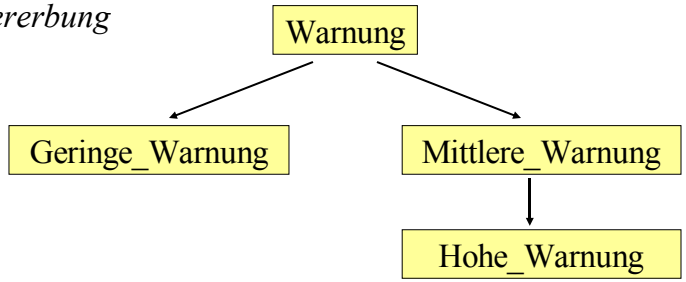
```
package Neues_Warnsystem is
type Anzeigegeraet is (Drucker, Bildschirm, Wandanzeige);
type Warnung is tagged
   record  Ankunftszeit: Kalender.Zeit;
           Nachricht: String;
   end record;
procedure Anzeigen (W: in Warnung; AG: in Anzeigegeraet);
procedure Mitprotokollieren (W: in Warnung);
procedure Reagieren (W: in out Warnung);

type Geringe_Warnung is new Warnung with null record;
procedure Reagieren (G: in out Geringe_Warnung);

type Mittlere_Warnung is new Warnung with
   record  Verantwortlicher: Personalverwaltung.Person;
   end record;
procedure Reagieren (M: in out Mittlere_Warnung);
```

```
-- Fortsetzung von package Neues_Warnsystem is
type Hohe_Warnung is new Mittlere_Warnung with
   record  Alarm_ausgeloest: Kalender.Zeit;
   end record;
procedure Reagieren (H: in out Hohe_Warnung);
procedure Alarm_Ausloesen (H: in out Hohe_Warnung);
end Neues_Warnsystem;
```

Vererbung :




```
package body Neues_Warnsystem is

procedure Reagieren (W: in out Warnung) is
begin  W.Ankunftszeit := Kalender.Uhrzeit;
       Mitprotokollieren (W);
end Reagieren;

procedure Reagieren (G: in out Geringe_Warnung) is
begin  Reagieren (Warnung(G));
       Anzeigen (Warnung(M), Drucker);
end Reagieren;

procedure Reagieren (M: in out Mittlere_Warnung) is
begin  Reagieren (Warnung(M));
       M.Verantwortlicher:=Personalverwaltung.Aufsichtführender;
       Anzeigen (Warnung(M), Bildschirm);
end Reagieren;
```



```
procedure Reagieren (H: in out Hohe_Warnung) is
begin  Reagieren (Mittlere_Warnung(H));
       Anzeigen (Warnung(H), Wandanzeige);
       Alarm_Ausloesen (H);
end Reagieren;

procedure Anzeigen (W: in Warnung; AG: in Anzeigegeraet)
       is separate;

procedure Mitprotokollieren (W: in Warnung) is separate;

procedure Alarm_Ausloesen (H: in out Hohe_Warnung)
       is separate;
…

end Neues_Warnsystem;
```


Will man nun eine neue Dringlichkeitsstufe "Notfall" einführen, so kann man zum Beispiel im package "Neues_Warnsystem" einen Typ mit Prozeduren

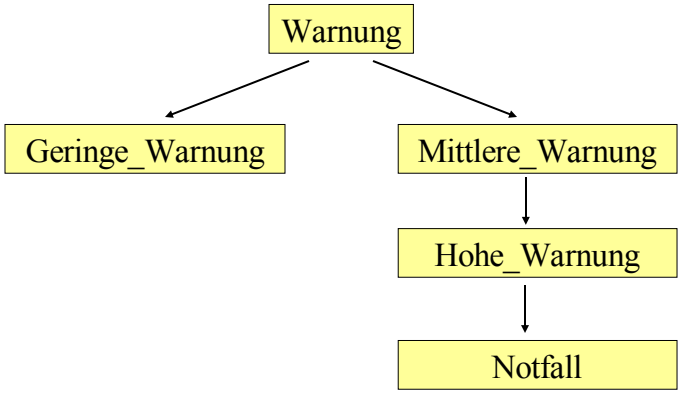
```
type Notfall is new Hohe_Warnung with
   record Rettungsdienst_alarmiert: Kalender.Zeit;
   end record;
procedure Rettungsdienst_rufen (N: Notfall);
procedure Reagieren (N: in out Notfall);
```

hinzufügen.

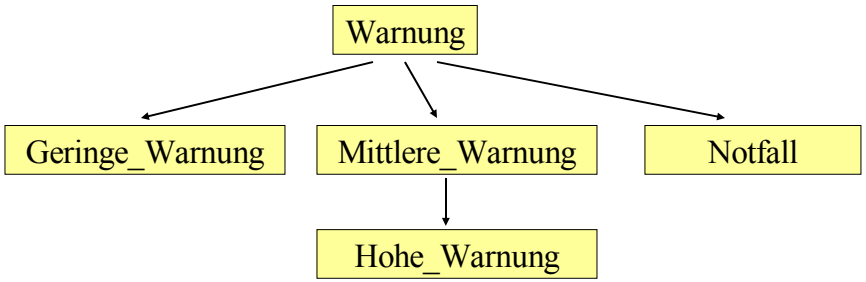
Oder man kann ein neues Paket einführen, das auf dem bisherigen package "Neues_Warnsystem" aufbaut:

```
with Neues_Warnsystem;
package Aktuelles_Warnsystem is
type Notfall is new Neues_Warnsystem.Hohe_Warnung
   with private;
procedure Rettungsdienst_rufen (N: in Notfall);
procedure Reagieren (N: in out Notfall);
procedure Anzeigen (N: in Notfall;
   AG: Neues_Warnsystem.Anzeigegeraet);
   -- diese Prozedur Anzeigen muss nun neu formuliert werden,
   -- indem z.B. die Anzeige blinkt oder farbig dargestellt wird.
private
   type Notfall is new Neues_Warnsystem.Hohe_Warnung
   with record Rettungsdienst_alarmiert: Kalender.Zeit;
        end record;
end Aktuelles_Warnsystem;
```


Neue Vererbungshierarchie:



Man kann das Paket "Aktuelles_Warnsystem" auch direkt auf dem Paket "Warnsystem" aufsetzen, wie es im Buch von Nagl geschieht. Dann kann man folgende Vererbungshierarchie implementieren:



Nun haben wir die Objekte einschließlich Vererbung dargestellt. Wir möchten jedoch eine einheitliche Verarbeitung haben, d.h., es soll für jede eingehende Warnung (egal welcher Dringlichkeitsstufe) die zugehörige Prozedur "Reagieren" ausgeführt werden.

Hierfür muss es einen Datentyp geben, der die (disjunkte) Vereinigung aller Datentypen ist, die aus einem Typ "T" durch Vererbung abgeleitet werden können.

In Ada wird dieser Typ beschrieben durch `T'Class`.
In unserem Beispiel ist also mit den Typen Warnung, Geringe_Warnung, Mittlere_Warnung, Hohe_Warnung und Notfall zugleich der Typ `Warnung'Class` als Vereinigung dieser Typen definiert.

Somit können wir nun eine einheitliche Prozedur einführen:

procedure Warnungsbehandlung (X: in out Warnung'Class) is
begin ... Reagieren (X); ... end;

Diese Prozedur können wir überall dort deklarieren, wo der Typ "Warnung" und die zu jedem hieraus abgeleiteten Typ gehörigen Prozeduren "Reagieren" sichtbar sind.
Zur Laufzeit wird für jede eintreffende Warnung Y
Warnungsbehandlung(Y);
aufgerufen. Der "tag" der Warnung legt den Typ von Y fest. Zur Laufzeit wird dann die zugehörige Prozedur "Reagieren" ermittelt und mit Y ausgeführt (dynamisches Binden).

Das Warnsystem wird nun wie folgt realisiert:

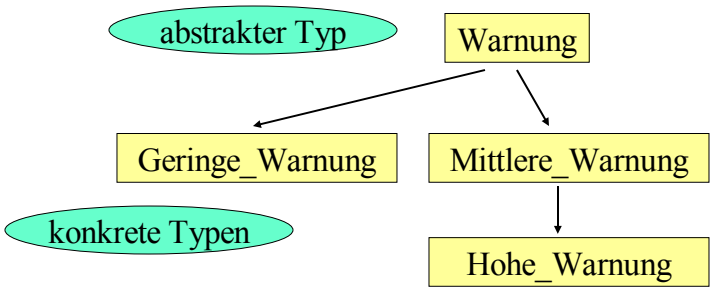
Sofern die Sensoren Signale empfangen, senden sie eine Warnung an das Rechnersystem, wobei in der Warnung mindestens zwei Komponenten gefüllt sind: (der String) Nachricht und der "tag", der den Datentyp angibt.

Diese werden automatisch in eine Warteschlange in der Halde eingetragen; das System fragt diese Schlange (Datentyp queue über dem Typ Warnung'Class) ständig auf neue Einträge ab. Auf das erste Element zeigt eine Zeigervariable "head" vom Typ

```
type Ref_Warnung is access Warnung'Class;
... head: Ref_Warnung; Y: Warnung'Class; ...
while ... loop ...;
  if not isemptyqueue then Y := head;
     Warnungsbehandlung(Y); head := head.next; ... end if; ...
```


Eine andere Möglichkeit, zur Laufzeit die Objekte der Variablen an die richtigen Prozeduren usw. zu binden, verwendet so genannte abtrakte Typen, vgl. 4.5.2 (dies hat nichts mit 'abstrakten Datentypen' zu tun!).

Man kann hier eine "Wurzel" der Objekthierarchie festlegen zusammen mit Operationen, die zunächst "abstrakt" bleiben und erst bei der Definition der konkreten Typen ausgefüllt werden. In der Syntax wurde dieser Fall bereits durch basic_declaration ::= abstract_subprogram_declaration in 4.1.10 berücksichtigt. Wir erhalten die Vererbung:




```
package Basisalarmsystem is
type Warnen is abstract tagged null record;
procedure Reagieren (Y: in out Warnen) is abstract;
end Basisalalarmsystem;

with Kalender; with Personalverwaltung;
with Basisalarmsystem;
package W_System is
type Anzeigegeraet is (Drucker, Bildschirm, Wandanzeige);
type Geringe_Warnung is new Basisalarmsystem.Warnen with
        record  Ankunftszeit: Kalender.Zeit;
        Nachricht: String;
        end record;
procedure Anzeigen
        (W: in Geringe_Warnung; AG: in Anzeigegeraet);
procedure Mitprotokollieren (W: in Geringe_Warnung);
procedure Reagieren (W: in out Geringe_Warnung);
```

```
type Mittlere_Warnung is new Basisalarmsystem.Warnen with
record  Ankunftszeit: Kalender.Zeit;
        Nachricht: String;
        Verantwortlicher: Personalverwaltung.Person;
end record;
procedure Reagieren (M: in out Mittlere_Warnung);
procedure Mitprotokollieren (W: in Mittlere_Warnung);

type Hohe_Warnung is new Mittlere_Warnung with
record  Alarm_ausgeloest: Kalender.Zeit;
end record;
procedure Reagieren (H: in out Hohe_Warnung);
procedure Alarm_Ausloesen (H: in out Hohe_Warnung);
end W_System;
```

In Reagieren kann nun ein Aufruf *Mitprotokollieren(Mittlere_Warnung(H));* aufttreten. Überlegen Sie sich, wie man die Prozedur "Anzeigen" mitnutzen kann. Formulieren Sie den "package body" zu W_System aus.