zu Kapitel 4

Stand: 31.1.2008

Einführung in die Informatik III

(für Studierende des 3. Fachsemesters)

Pflichtvorlesung für die Diplomstudiengänge "Informatik" und "Automatisierungstechnik in der Produktion" sowie wählbare Vorlesung für weitere Studiengänge, z. B. für Mathematik, Softwaretechnik, Lehramt Informatik usw.

Universität Stuttgart, Wintersemester 2007/2008

Dozent: Volker Claus

4. Nebenläufigkeit

4.1 Stellen-Transitions-Netze

4.2 Nachrichtenaustausch

Dieses Kapitel ist weitgehend im Kapitel 13 der Grundvorlesung enthalten, wurde dort aber nur sehr knapp (wenn überhaupt) behandelt. In unserer Vorlesung wird der Stoff bis 4.2.10 besprochen.

4.1 Stellen-Transitions-Netze

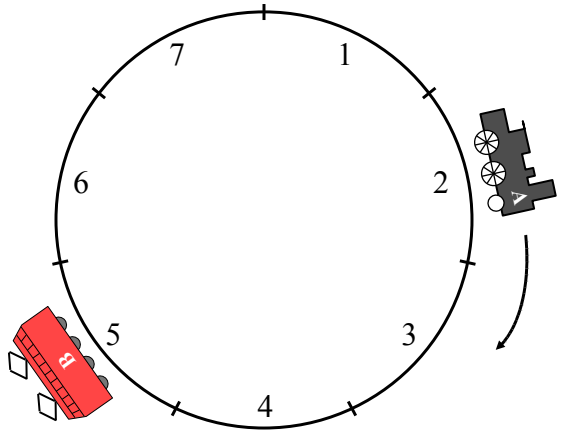
Bisher: Sequentielle Programmierung, d.h., höchstens eine Stelle im Programm wird in jedem Augenblick bearbeitet. Jeder Ablauf wird hierbei in eine Folge nacheinander auszuführender Aktivitäten zerlegt.

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. Wir beginnen mit einem Kalkül.

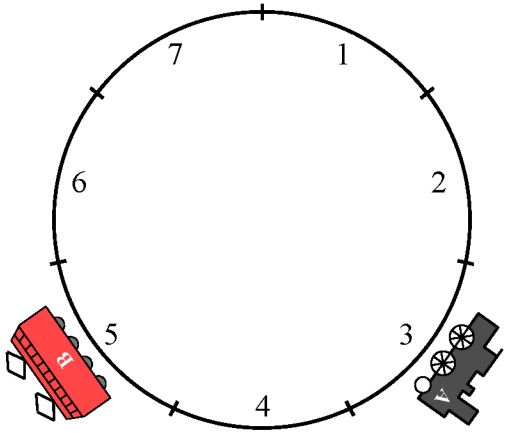
Ansatz: Verallgemeinere endliche Automaten, in denen mehrere Zustände gleichzeitig oder nacheinander aktiv sind. Als Beispiel wählen wir Züge auf einer Kreisstrecke.

4.1.1 Beispiel: Züge auf einer kreisförmigen Strecke.

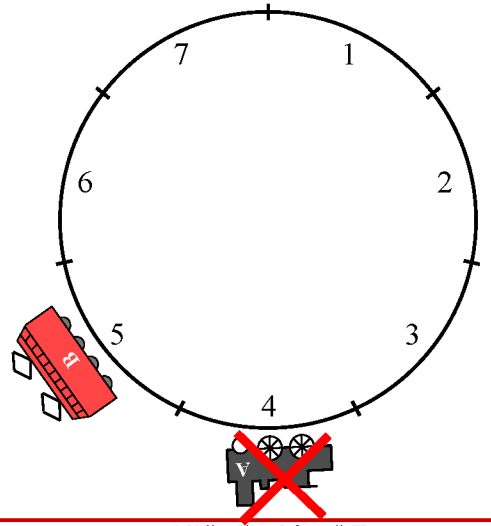
Damit die Züge nicht zusammenstoßen, verlangen wir, dass zwischen ihnen mindestens ein Streckenabschnitt frei bleibt.



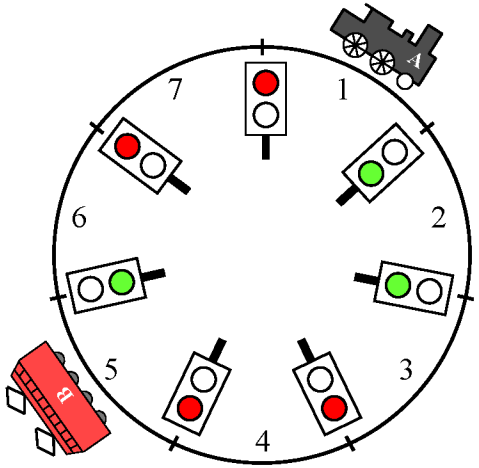
Damit die Züge nicht zusammenstoßen, verlangen wir, dass zwischen ihnen mindestens ein Streckenabschnitt frei bleibt.



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



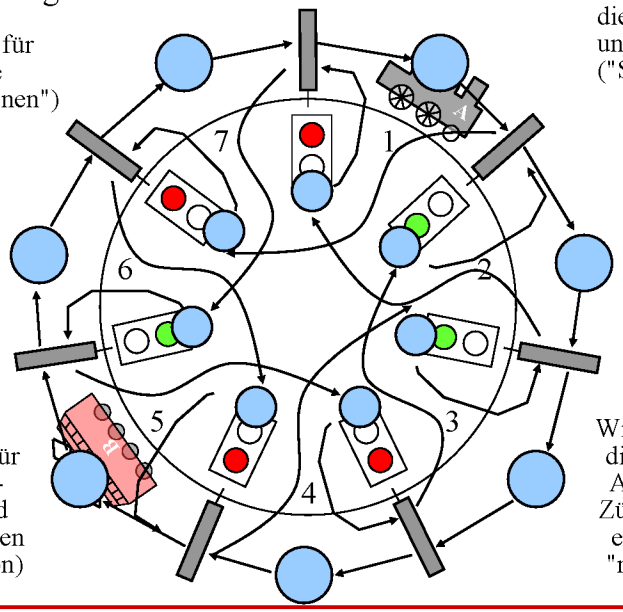
Steuerung durch Ampeln: Grünes Signal bedeutet, dass in den Streckenabschnitt hineingefahren werden darf.



Umwandlung in ein Netz:

Rechtecke für Übergänge ("Transitionen")

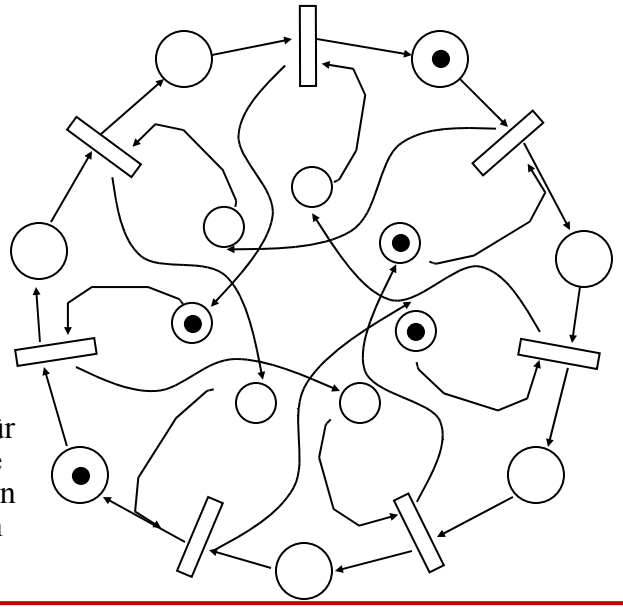
Kreise für die Strecken und Ampeln ("Stellen")



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

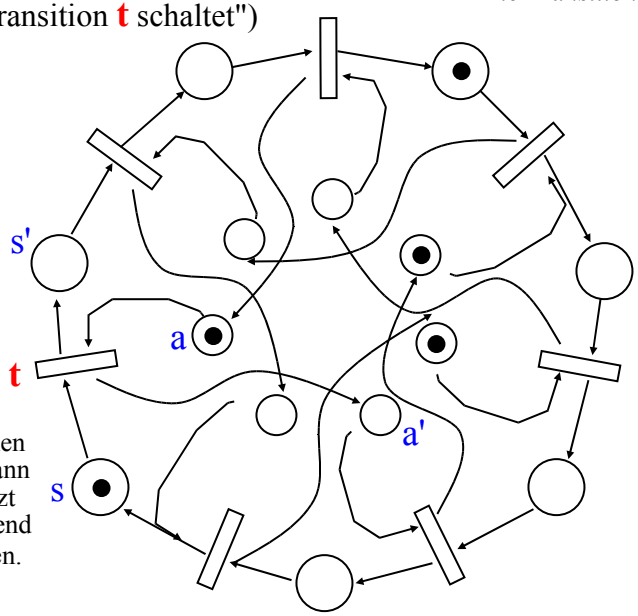
Wir lassen nun die Strecken, Ampeln und Züge weg und erhalten das "reine Netz".

Das resultierende Netz:



Marken für mögliche Situationen einfügen

Eine Aktivität bei **t** durchführen ("die Transition **t** schaltet")



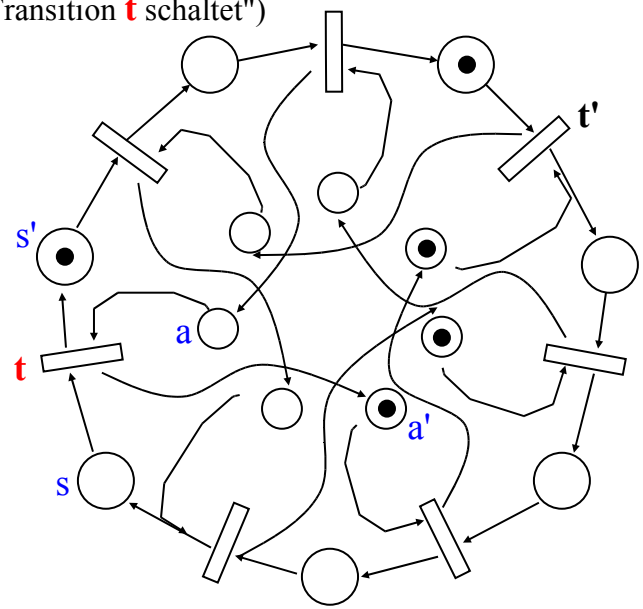
Die Marken werden dann umgesetzt entsprechend den Kanten.

Die Transition t "schaltet":

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 bei **t** durchführen ("die Transition **t** schaltet")

Ergebnis des Schaltens:



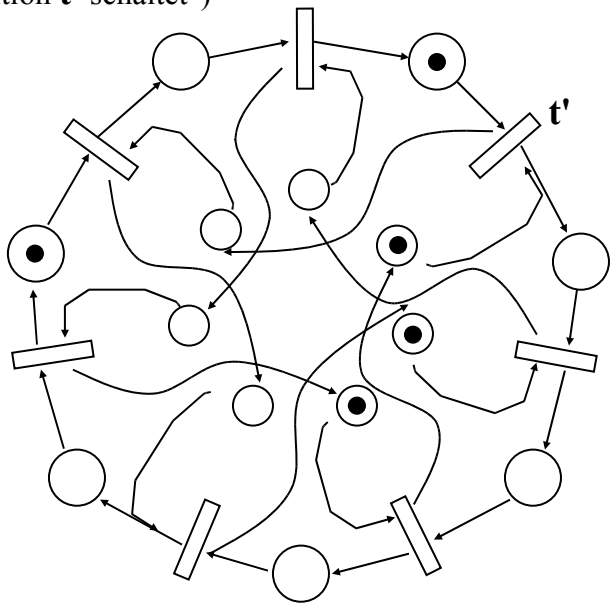
Der Zug ist nun auf der Strecke s' (= in Stelle s' liegt eine Marke). Die Ampel a steht auf rot (= in der Stelle a liegt keine Marke). Dafür wird aber a' auf grün gestellt (= in Stelle a' liegt eine Marke).

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.

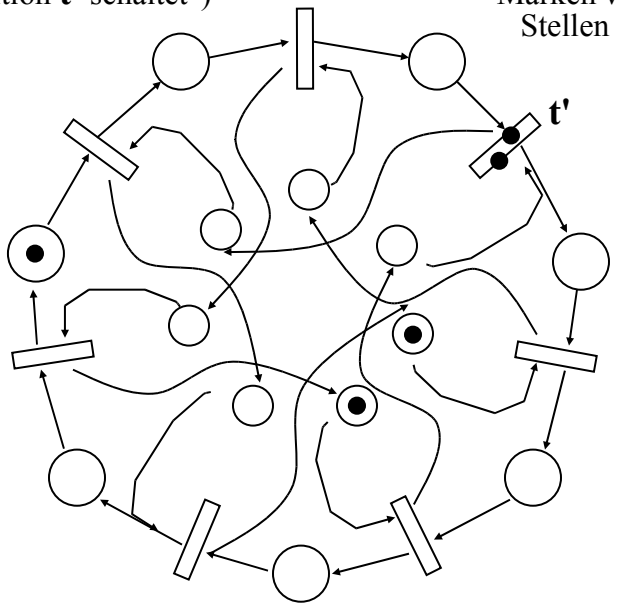
In unserem Beispiel: Der linke Zug kann nicht weiterfahren, weil die zugehörige Ampel keine Marke enthält. Aber der rechte Zug kann weiterfahren, d.h., die Transition t' kann jetzt schalten. Das Ergebnis (= die neue Verteilung der Marken) finden Sie auf der nächsten Folie.

Nächste Aktivität durchführen
("die Transition **t'** schaltet")



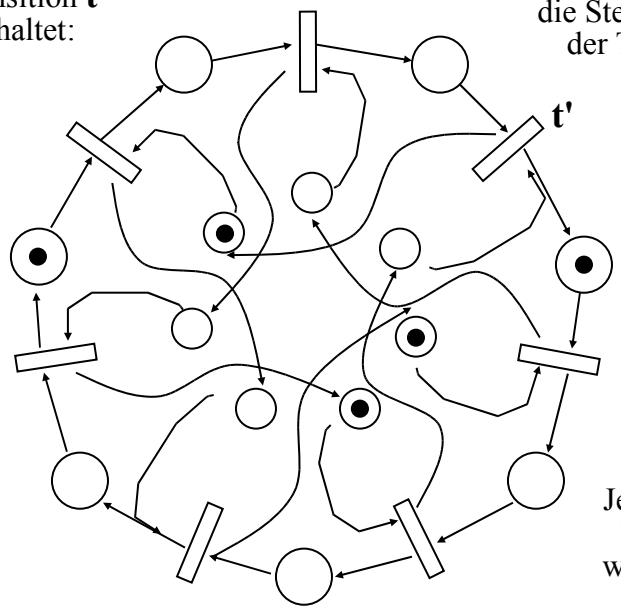
Nächste Aktivität durchführen
("die Transition **t'** schaltet")

Zwischensituation:
Marken werden von
Stellen abgezogen



Die Transition **t'**
hat geschaltet:

Marken werden auf
die Stellen hinter
der Transtition
gelegt



Jetzt können
beide Züge
weiterfahren
usw.

Definition 4.1.2: 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),

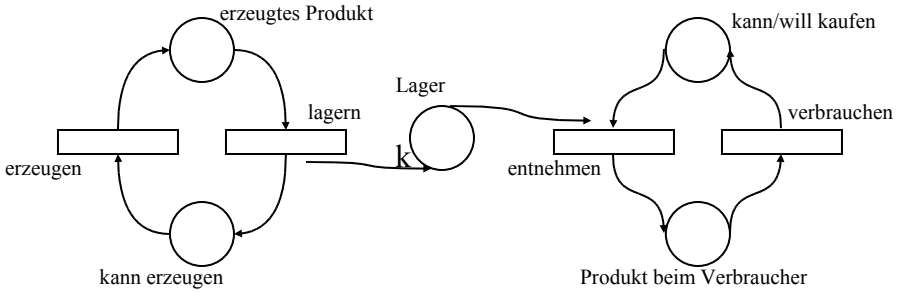
(4) $K: S \rightarrow \mathbb{N} \cup \{\infty\}$ ist die Kapazität für jede Stelle,

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

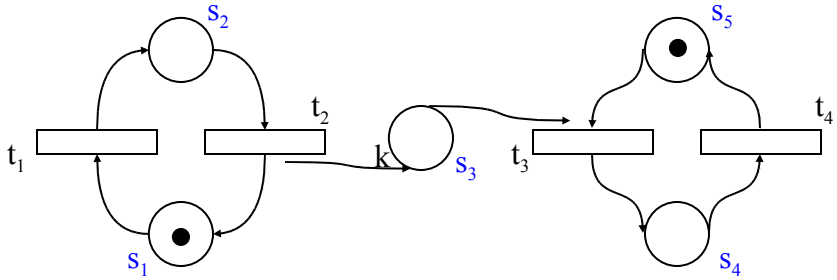
(6) $M_0: S \rightarrow \mathbb{N}_0 \cup \{\infty\}$ ist die Anfangsmarkierung, für die gelten muss: $\forall s \in S: M_0(s) \leq K(s)$, d.h., in keiner Stelle dürfen mehr Marken liegen, als die Kapazität zulässt (die Markierungen schreibt man in der Regel als Vektoren).

Beispiel 4.1.3: Erzeuger-Verbraucher-Kreislauf (Producer-Consumer-Cycle)

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

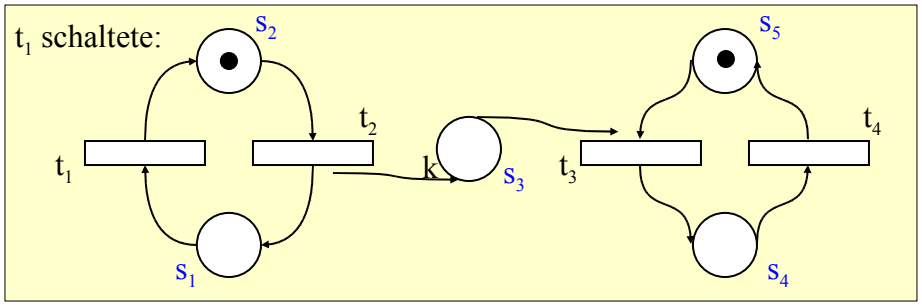


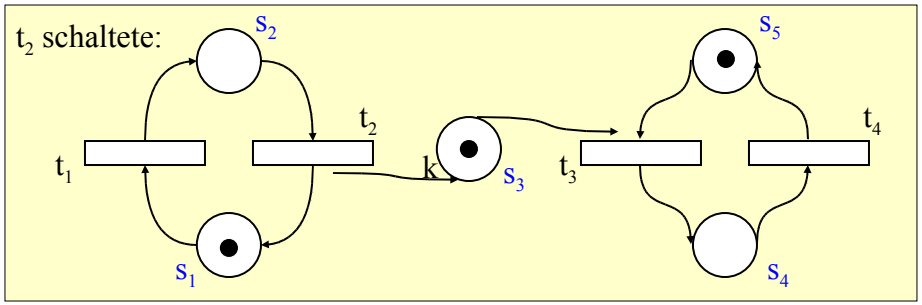
Hinweis: Das Lager hat die Kapazität k, alle anderen Stellen haben die Kapazität ∞.



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

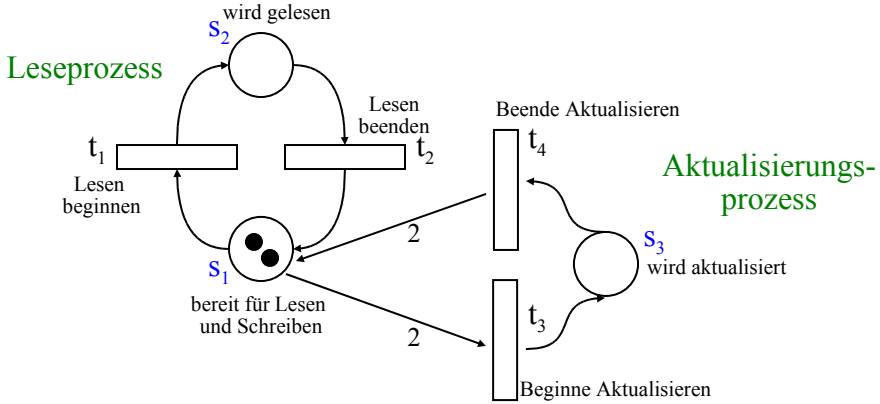
Formal: $S = \{s_1, s_2, s_3, s_4, s_5\}$, $T = \{t_1, t_2, t_3, t_4\}$, $M_0 = (1,0,0,0,1)$,
$F=\{(s_1,t_1),(s_2,t_2),(t_1,s_2),(t_2,s_1),(t_2,s_3),(s_3,t_3),(s_5,t_3),(t_3,s_4),(s_4,t_4),(t_4,s_5)\}$,
$W((x,y))=1$ für alle Kanten (x,y), $K(s_1)=K(s_2)=K(s_4)=K(s_5)=\infty$, $K(s_3)=k$.
In dieser Anfangssituation 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.

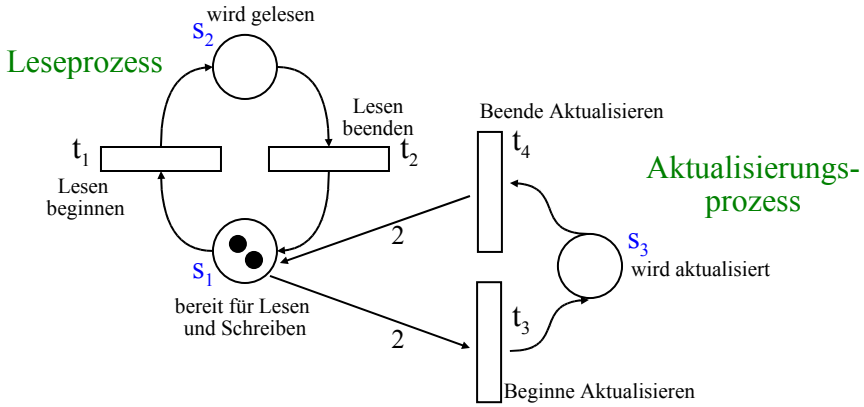




Beispiel 4.1.4: Lese-Schreib-Konflikt

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



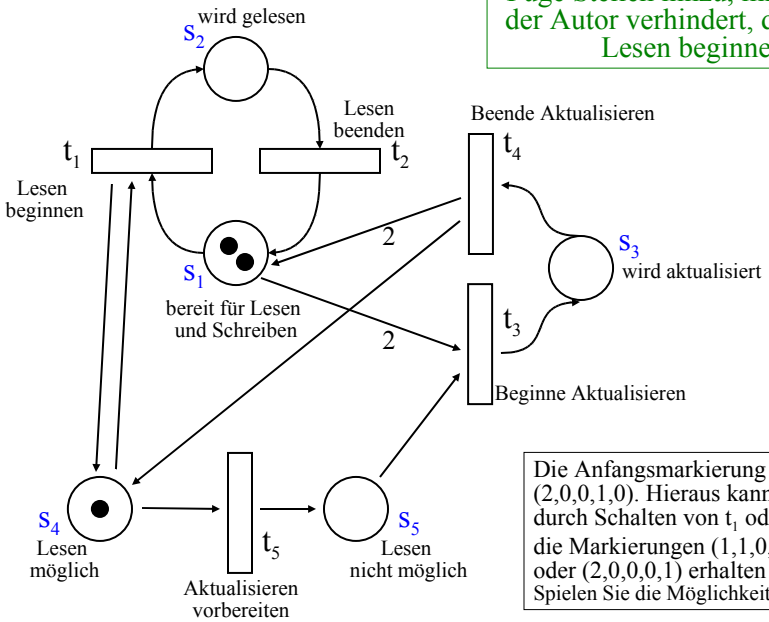


Nun kann eine "unfaire Aktionsfolge" auftreten:

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

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 4.1.5 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. Wie 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 4.1.5 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 4.1.6: 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 (zulässige) 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 4.1.5 (4).]

noch Definition 4.1.6:

(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 $[>$ 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 $M[t_1>M_1$, $M_1[t_2>M_2$, $M_2[t_3>M_3$, ..., $M_{r-2}[t_{r-1}>M_{r-1}$, $M_{r-1}[t_r>$.

noch Definition 4.1.6: Aktiviertheit und Erreichbarkeit

(5) Fortsetzung der Relation $[>$ 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 $M[t_1>M_1$, $M_1[t_2>M_2$, $M_2[t_3>M_3$, ..., $M_{r-1}[t_r>M'$.
(Im Falle r=0 muss M=M' sein.)

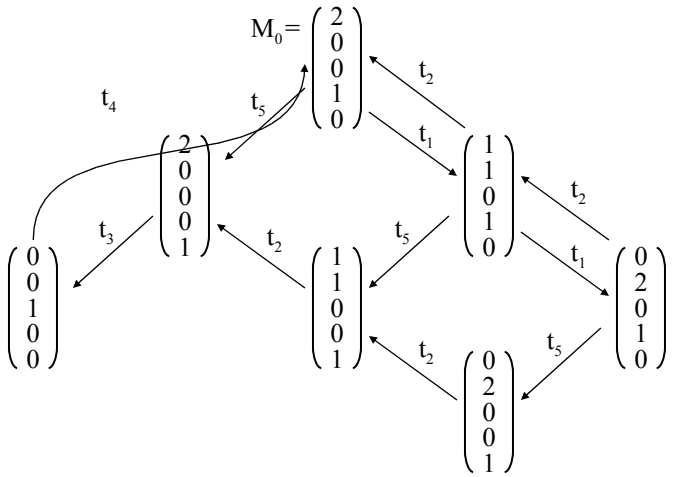
(6) $ERR(M) = \{M' \mid$ es existiert $t_1 t_2 t_3 \ldots t_r$ mit $M[t_1 t_2 t_3 \ldots t_r> M'\}$ heißt Erreichbarkeitsmenge bzgl. M.
(Beachte: $M \in ERR(M)$, insbesondere ist ERR(M) nie leer.)
Wenn M' in ERR(M) liegt, so sagt man auch, M' ist von M aus erreichbar.
$ERR(N) = \{M' \mid$ es existiert $t_1 t_2 t_3 \ldots t_r$ mit $M_0[t_1 t_2 t_3 \ldots t_r> M'\}$ heißt Erreichbarkeitsmenge des Netzes N.

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.

Die Erreichbarkeit stellt die Bedeutung der S/T-Netze dar.

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.

4.1.7: *Konstruktion des Erreichbarkeitsgraphen* des letzten S/T-Netzes aus Beispiel 4.1.4: Ausgehend von M_0 werden nacheinander alle im nächsten Schritt erreichbaren Markierungen notiert:

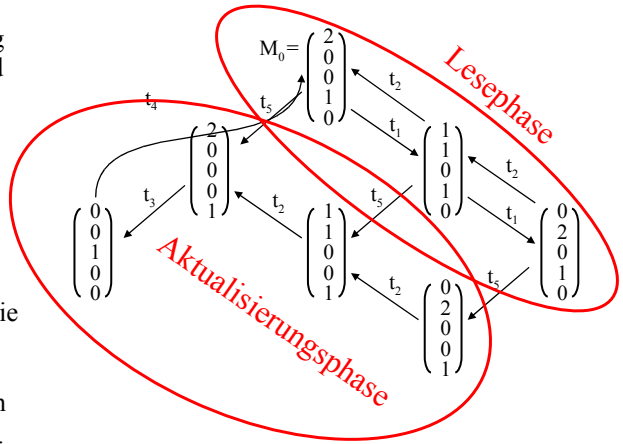


Dieser Graph besitzt einige Eigenschaften, die uns Hinweise geben, ob das angegebene Netz unser gestelltes Lese-Schreib-Problem tatsächlich löst:

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

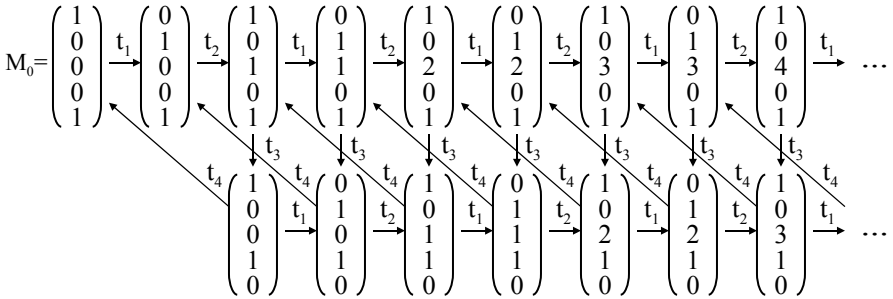
2. Man entdeckt eine "Invariante" der erreichbaren Markierungen M: $(1,1,3,1,1)\cdot M = 3$.

3. Man sieht, dass die Transitionenfolgen $(t_1t_2)^k$ und $(t_5t_3t_4)^k$ korrekt von M_0 nach M_0 führen ($\forall k \in \mathbb{N}$).



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

4.1.8: Konstruktion des Erreichbarkeitsgraphen aus 4.1.3



Falls die in 4.1.3 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 4.1.9:

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: $G(N) = (\mathrm{ERR}(N), \{(M, t, M') \mid M[t>M'\})$, 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 ausgehend von M_0 schrittweise aufbaut.]

4.1.10: 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., $\forall M \in ERR(N): M(s) \leq k$.

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

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

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

4.1.11: Folgerung

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

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

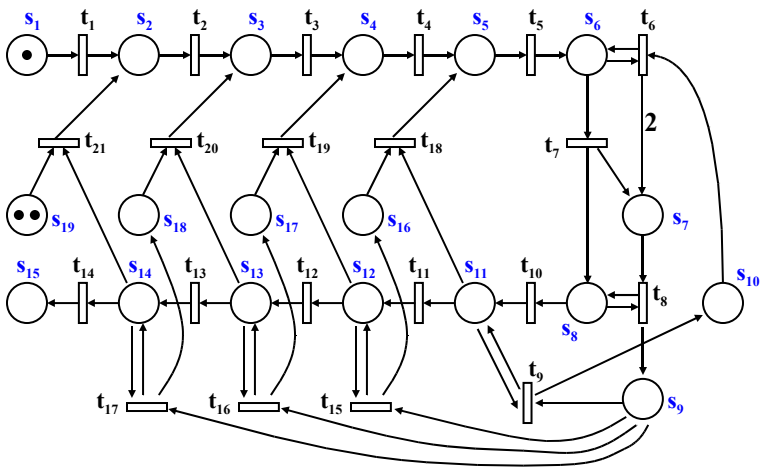
"⇒" Wenn N beschränkt ist, so gibt es ein k, so dass alle Markierungen des Erreichbarkeitsgraphen nur Komponenten besitzen, die kleiner oder gleich k sind. Dann kann es aber höchstens $(k+1)^{|S|}$ Markierungen in ERR(N) geben, d.h., der Erreichbarkeitsgraph ist endlich.

"⇐" Wenn der Erreichbarkeitsgraph endlich ist, dann existiert das Maximum m für alle Komponenten von Markierungen in ERR(N). Jede Stelle ist dann m-beschränkt, d.h., das Netz ist beschränkt.

Warnung: 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

$2^{2^{2^{\cdots^{2}}}}$ (65535 mal) Knoten. Vollziehen Sie etwa 50 Schritte nach, um die Wirkungsweise des Netzes zu erahnen.



4.1.12: 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 $M'[t>$ gibt. Formal: $\forall M \in ERR(N)\ \exists M' \in ERR(M): M'[t>$.

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

Man könnte ein Netz auch lebendig nennen, wenn es immer weiterschalten kann. Im Netz darf es dann keine "Verklemmung" geben, d.h., es darf keine von M_0 aus erreichbare Markierung ohne Folge-Markierung geben.

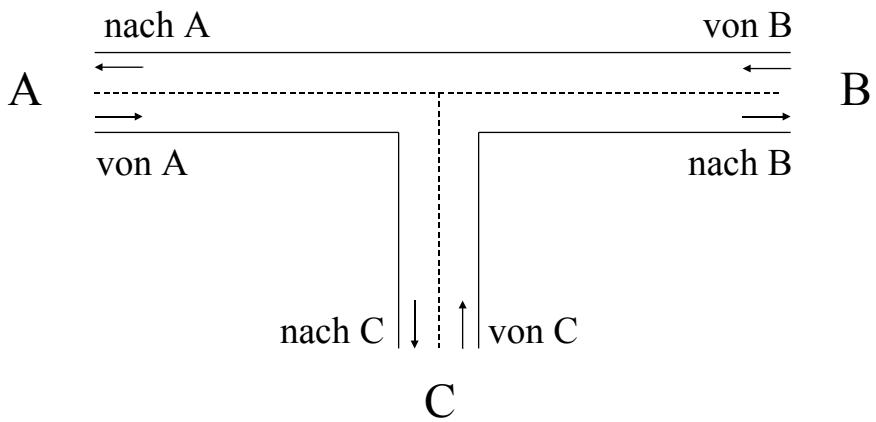
Fortsetzung der Definition:

(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. $\forall M \in ERR(N)\ \exists t \in T: M[t>$.

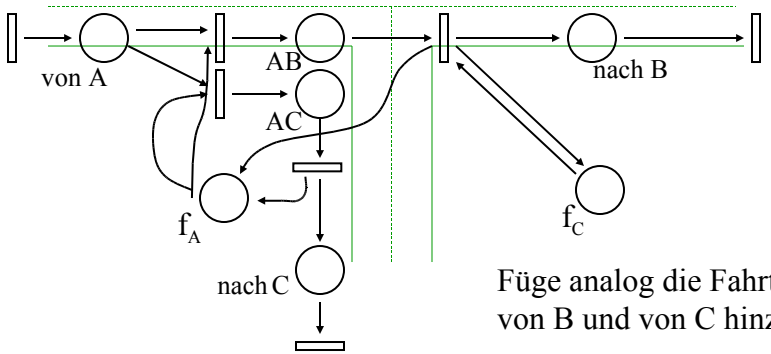
4.1.13: Beispiel:

Straßenkreuzung mit Vorfahrtsregel "rechts vor links".



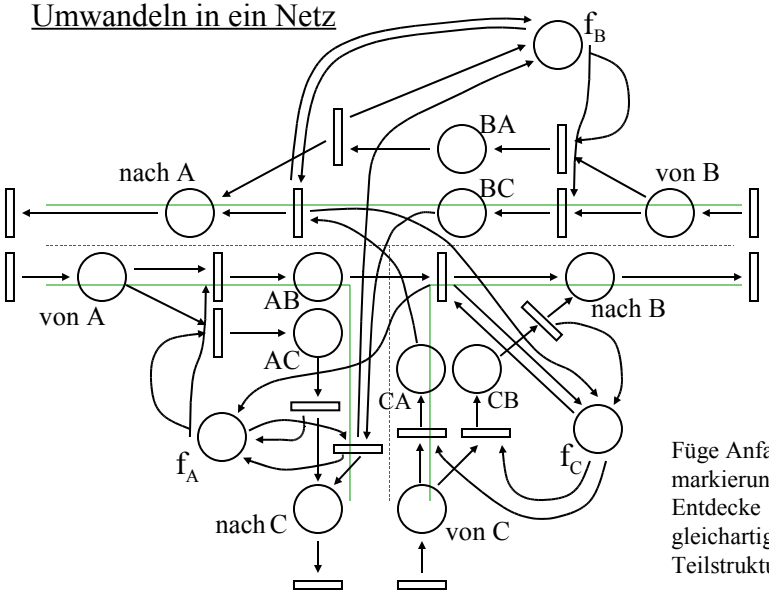
Stellen sind Straßenabschnitte, auf denen maximal ein Fahrzeug stehen kann. Unsere Stellen s sollen daher alle die Kapazität K(s) = 1 besitzen!

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).
(Beachte: Jede Stelle s besitzt hier die Kapazität K(s)=1.)



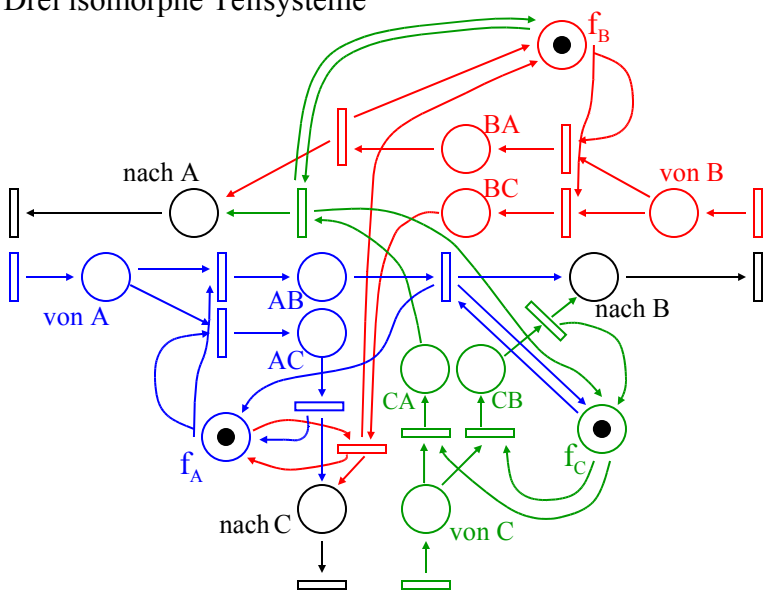
Füge analog die Fahrten von B und von C hinzu.

Umwandeln in ein Netz

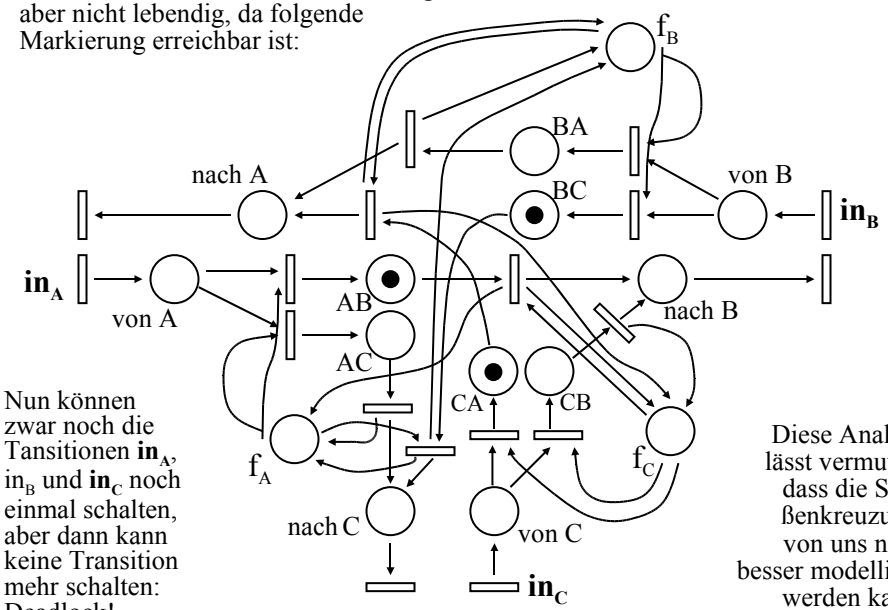


Füge Anfangsmarkierung hinzu. Entdecke gleichartige Teilstrukturen:

Drei isomorphe Teilsysteme



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

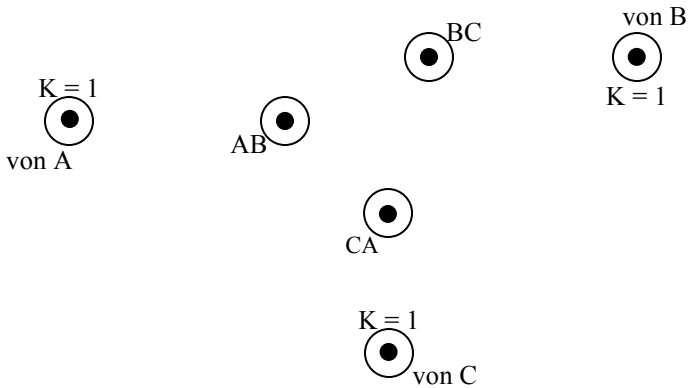


Nun können zwar noch die Tansitionen in_A, in_B und in_C noch einmal schalten, aber dann kann keine Transition mehr schalten: Deadlock!

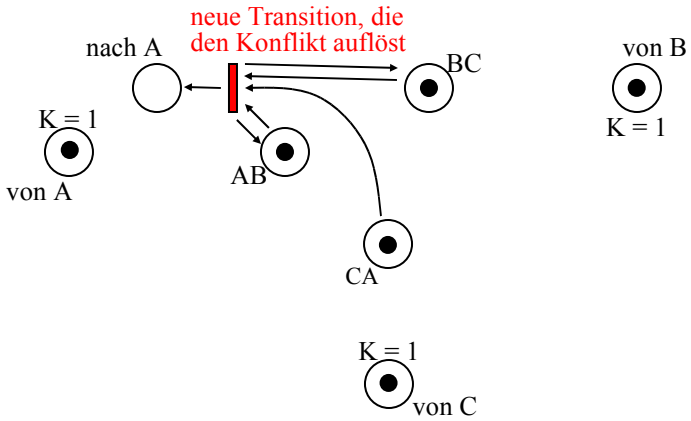
Diese Analyse lässt vermuten, dass die Straßenkreuzung von uns noch besser modelliert werden kann.

Wir hatten alle Stellen, also auch die Eingangs-Stellen "von A", "von B" und "von C" mit der Kapazität 1 belegt.

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

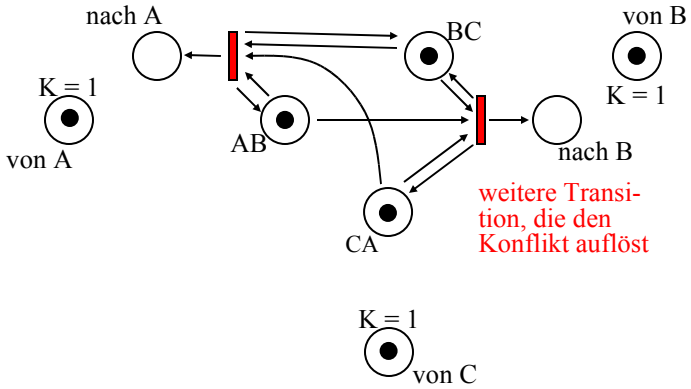


Diese Verklemmung lässt sich beseitigen, indem eine Transition eingefügt wird, die genau die Stellen, die die Verklemmung bewirken, als Vorbereich hat, und ihnen eine Folgemarkierung erlaubt. In obigem Fall z.B.: das von C nach A fahrende Fahrzeug darf nun über die Kreuzung fahren:



Wir können zusätzlich eine Transition hinzufügen, die das von A nach B fahrende Fahrzeug bevorzugt (rechts eingefügte Transition):

In obigem Fall darf nun auch das von A nach B fahrende Fahrzeug über die Kreuzung fahren. Beachten Sie, dass die anderen beiden Fahrzeuge warten, d.h., ihre Marken werden wieder auf BC und CA zurückgelegt.



Als drittes können wir eine Transition hinzufügen, die das von B nach C fahrende Fahrzeug im Konfliktfall 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 4.1.13

4.1.14: Wirkung des Schaltens einer Transition t_j:

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

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 $T = \{t_1, t_2, ..., t_m\}$.

Wir definieren für $w \in T^*$ den Anzahlvektor #w mit den Komponenten: $\#_j w$ = Anzahl der t_j, die in w vorkommen.

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

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.

4.1.15 Definition:

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

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

heißt Inzidenzmatrix des Netzes N.

4.1.16 Folgerung:

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

Wenn $M[w>M'$ möglich ist, dann gilt: $M' = M + C \cdot \#w$.

4.1.17 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 4.1.16, 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 $y \cdot M = y \cdot M'$ für alle von M aus erreichbaren Markierungen M' gilt.

4.1.18 Satz:

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

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

(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 $w \in T^*$ mit $M[w>M$ gibt, dann ist #w eine T-Invariante von N.

Dieser Satz folgt unmittelbar aus den bisherigen Ausführungen.

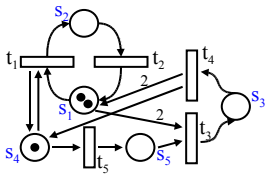
Bei (2) beachte man für die Komponente M(s):

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

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

Anwenden auf das Beispiel aus 4.1.4:

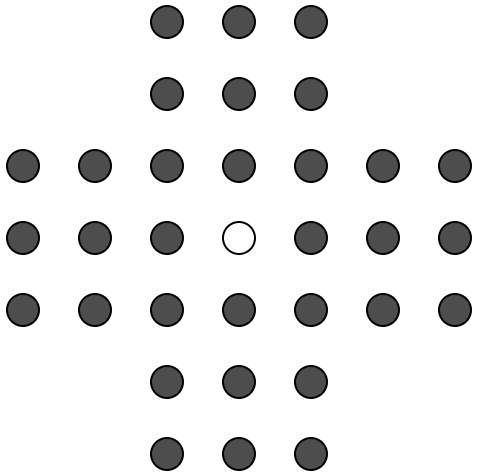
$$C = \begin{pmatrix} -1 & 1 & -2 & 2 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix} \quad M_0 = \begin{pmatrix} 2 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$



Eine echte S-Invariante lautet: y = (1, 1, 3, 1, 1). Nach Satz 4.1.18 (2) sind daher alle Stellen 3-beschränkt und somit ist das Netz beschränkt.

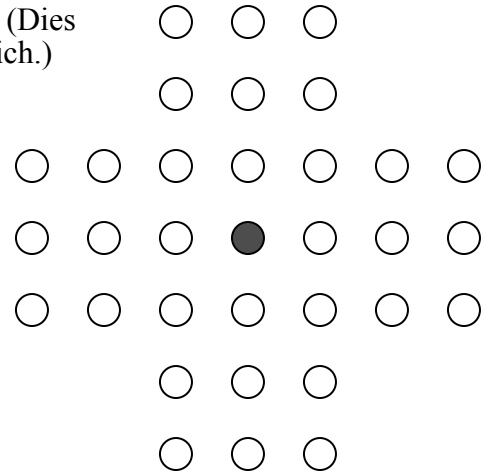
(1,1,0,0,0) und (0,0,1,1,1) sind T-Invarianten, daher verändern die Schaltfolgen t_1t_2 oder t_2t_1 bzw. $t_3t_4t_5$ oder $t_3t_5t_4$ oder $t_4t_3t_5$ oder $t_4t_5t_3$ oder $t_5t_3t_4$ oder $t_5t_4t_3$ (sofern sie schalten können) die Markierung nicht.

Illustration 4.1.19: Solitaire-Spiel

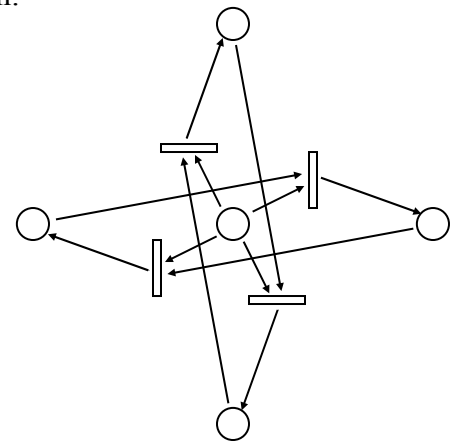


33 Stellen, 32 Steine, d.h., eine Stelle bleibt frei. Ein Zug = Springe in gerader Richtung (nicht diagonal) über einen Nachbarstein auf eine freie Stelle und entferne den übersprungenen Stein. Ziel: Am Ende soll nur noch ein Stein (möglichst auf einer vorgegebenen Stelle) übrig sein.

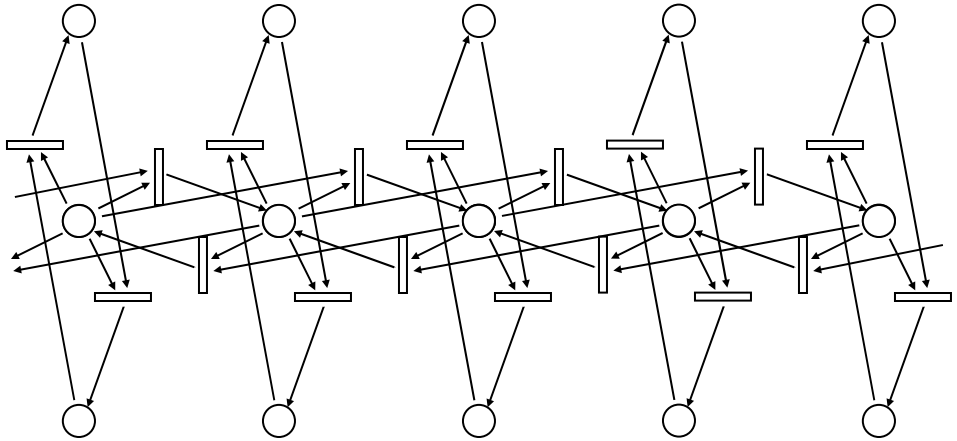
Meist versucht man, diese Endsituation zu erreichen. (Dies geht tatsächlich.)

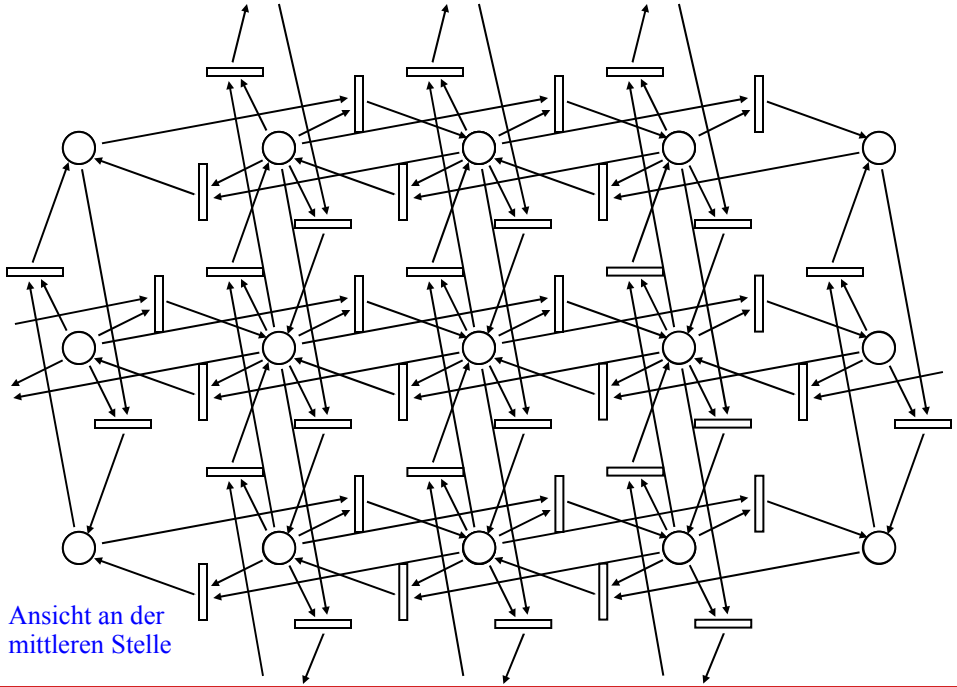


Solitaire als S/T-Netz formulieren. Betrachte eine Stelle mit ihren Nachbar-Stellen und formuliere das Überspringen durch Transitionen:

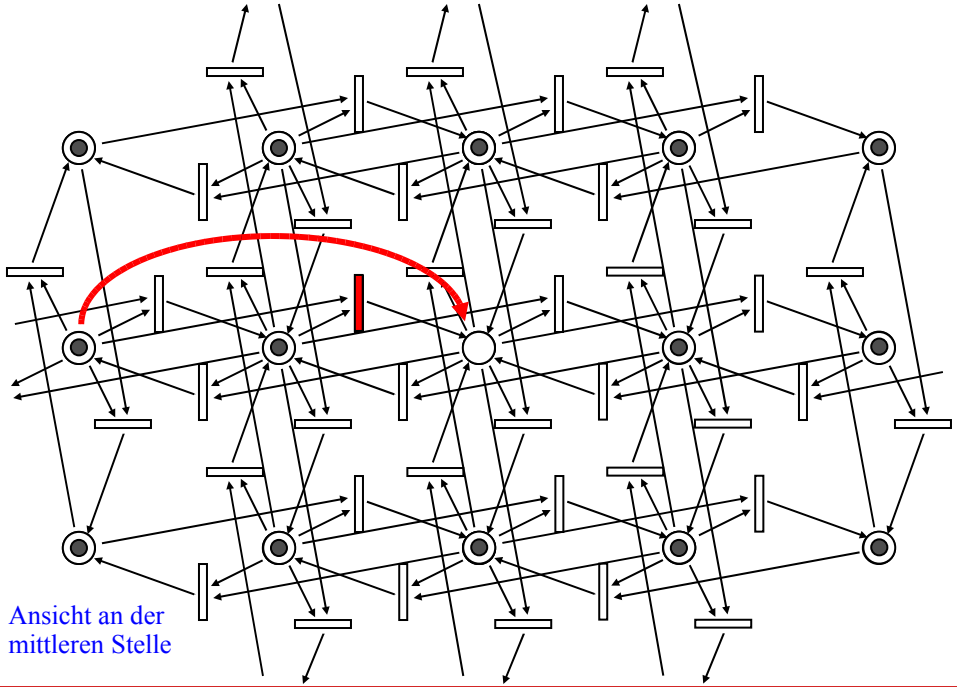


Dieses Muster muss nun an jeder Stelle eingefügt werden.

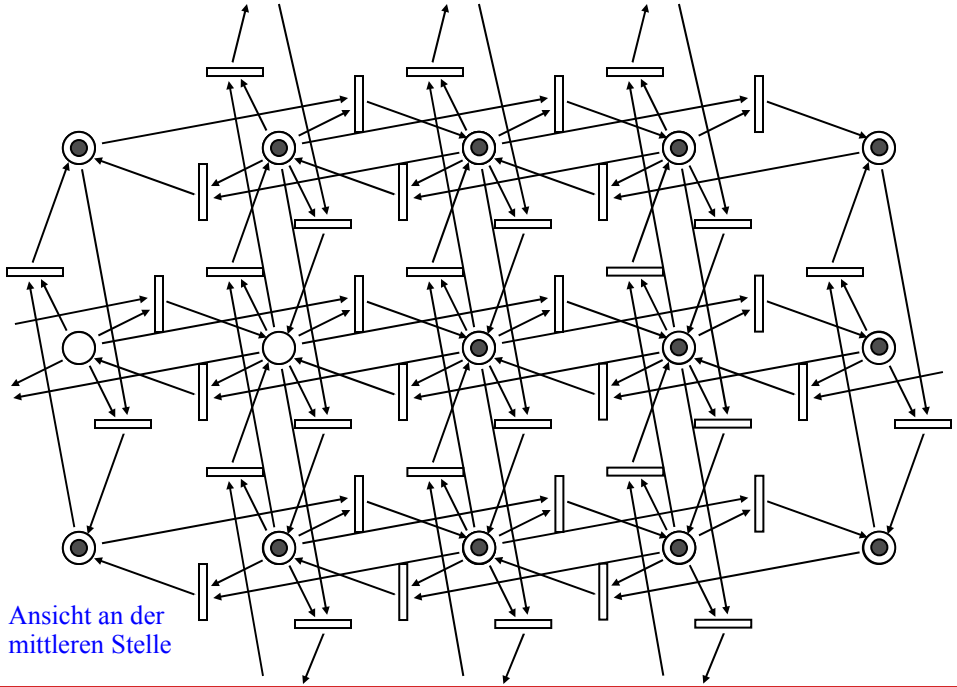




Ansicht an der mittleren Stelle



Ansicht an der mittleren Stelle



Ansicht an der mittleren Stelle

Das Ziel des Spiels besteht nun darin nachzuweisen, dass von der Anfangsmarkierung
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
eine vorgegebene Markierung, zum Beispiel
(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
erreicht werden kann.
Zugleich ist eine Transitionenfolge zu finden, die diese Erreichbarkeit bewirkt.

4.1.20 Abschließende Hinweise:

Die Erreichbarkeit ist entscheidbar, allerdings i. A. nur mit einem gewaltigen Zeitaufwand.
Das Gleiche gilt für die Beschränktheit. Beachten Sie, dass Satz 4.1.18 nur ein hinreichendes Kriterium enthält. Der Nachweis der Beschränktheit erfolgt mit sog. Überdeckungsgraphen. Diese kann man auch für die Untersuchung der Lebendigkeit einsetzen. Lebendigkeit ist nicht entscheidbar.

Anschaulich gesprochen heißt eine unendlich lange Schaltfolge fair, wenn sie jede Transition, die unendlich oft aktiviert ist, auch irgendwann schaltet.
Fairness definieren wir hier nicht formal. Das Problem, ob ein beliebiges S/T-Netz nur faire Schaltfolgen besitzt, ist nicht entscheidbar.

Hinweise zum umfangreichen Üben/Durchdenken (dies führt über diese Vorlesung hinaus):

(1) Schreiben/skizzieren Sie ein Paket "ST_Netz" in Ada, das die Stellen-Transitionsnetze darstellt.

(2) Formulieren Sie hierzu eine Prozedur, die zu einem gegebenen S/T-Netz den Erreichbarkeitsgraphen (bis zu einer vorzugebenden Größe) aufbaut.

(3) Implementieren Sie die Aufstellung der Inzidenzmatrix und die Berechnung von S- und T-Invarianten (sofern sie existieren).

(4) Schlagen Sie in der Literatur nach und implementieren Sie weitere Analyseverfahren (z.B. den Aufbau eines Überdeckungsgraphen, den Farkas-Algorithmus, den Nachweis der Erreichbarkeit mittels Backtracking usw.).

4.2 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 zu Beginn der Grundvorlesung eingeführt wurden. Wir beginnen mit dem gemeinsamen Speicherbereich.

4.2.1 Elementare Anweisungen (diese übernehmen wir aus Abschnitt 2.1.5 der Grundvorlesung 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.)

4.2.2 Zusammengesetzte Anweisungen (siehe wiederum Abschnitt 2.1.5 der Grundvorlesung):

Hintereinanderausführung oder Sequenz: Wenn γ_1 und γ_2 Anweisungen sind, dann ist auch $\gamma_1;\gamma_2$ eine Anweisung.

Alternative: Wenn γ_1 und γ_2 Anweisungen und β ein Boolescher Ausdruck sind, dann ist auch

if β then γ_1 else γ_2 fi .

eine Anweisung (else skip darf man weglassen.)

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

Hinzu kommen folgende zusammengesetzte Anweisungen (d. h., wenn $\gamma, \gamma_1, \gamma_2, .., \gamma_n$ Anweisungen sind, dann sind auch die folgenden Konstrukte ... Anweisungen):

Nichtdeterministische Auswahl für n≥2:

(γ_1 or γ_2 or γ_3 or ... or γ_n)

Nebenläufige Abarbeitung für n≥2:

(γ_1 | γ_2 | γ_3 | ... | γ_n)

(Die nebenläufigen Anweisungen γ_i nennt man auch "Prozesse").

Blöcke mit gemeinsamen Variablen:

[local <lokale Deklarationen>; γ]

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

Hinweis: Unsere Beispiele haben meist die Form:

P:: [local <lokale Deklarationen>; (γ_1 | γ_2 | γ_3 | ... | γ_n)]

Beispiel 4.2.3:

```
[ 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 4.1.3, mit der Anzahl L der Elemente im Lager, die zwischen 0 und max (=Kapazität) 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 dies 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 beispielsweise 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 und ob es Berechnungsteile gibt, die von außen nicht unterbrochen werden können.

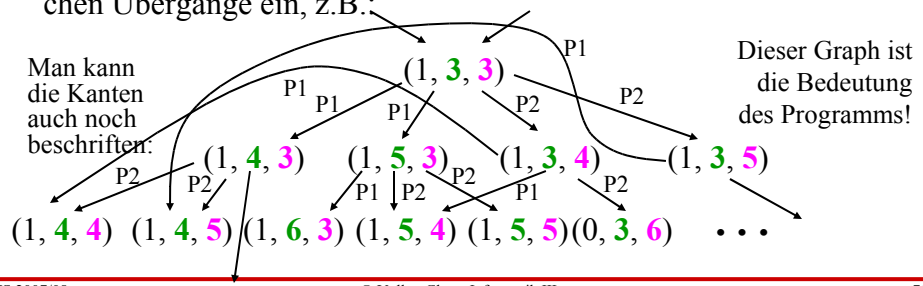
Wir formulieren nun die Menge der möglichen Zustände zu der Unterteilung, die auf der vorigen Folie angegeben wurde.

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 4.1.6 die möglichen Übergänge ein, z.B.:

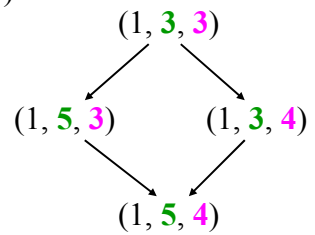


P1:: ① while ② true do ③ await L < max; (④ L := L+1 or ⑤ skip) ⑥ od;	P2:: ① while ② true do ③ await L > 0; (④ L := L-1 or ⑤ skip) ⑥ od;

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

(1, 3, 3)
↓
(1, 5, 4)

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



4.2.4: Legt man die Bedeutung (Semantik) nebenläufiger Programme so fest, dass zwei Prozesse niemals gleichzeitig einen Schritt (= Wechsel des Zustands) 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 und in Wahrheit die Prozesse verzahnt sequenziell ausführt).

4.2.5: 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, 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:

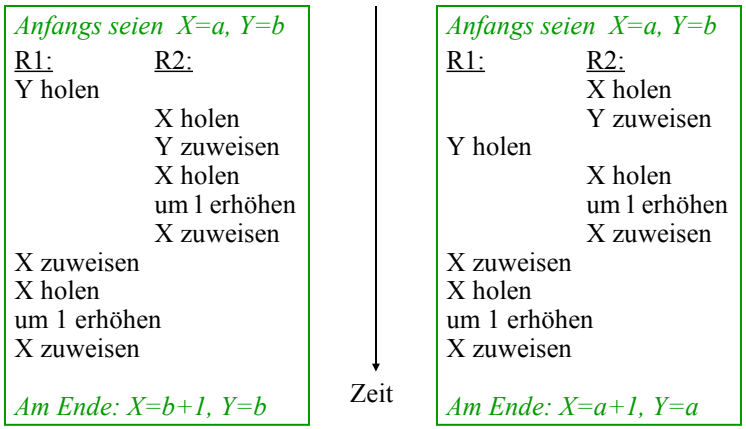
Hole den Wert a von L bzgl. Q1; hole den Wert a von L bzgl. Q2; bilde a+1 bzgl. Q1; bilde a-1 bzgl. Q2; weise a+1 der Variablen L zu bzgl. Q1; weise a-1 der Variablen L zu bzgl. Q2.

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

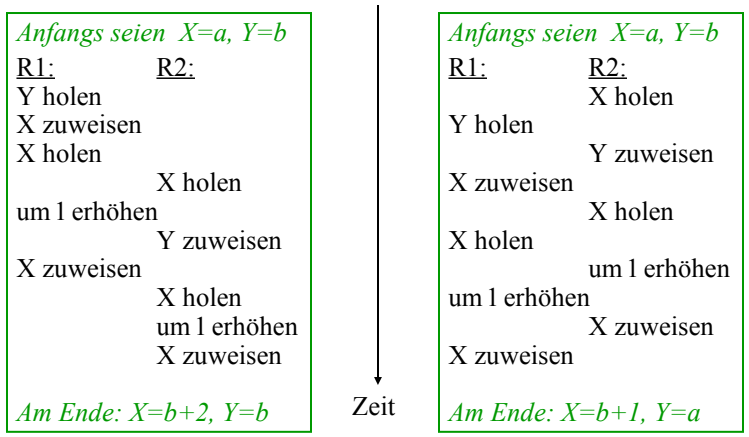
Sind überhaupt keine Anweisungsteile "nicht unterbrechbar", dann sind alle zeitlich verschachtelten Reihenfolgen möglich.

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. Vier 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 einem nebenläufigen Prozess ausgeführt werden muss, nennt man einen *kritischen Abschnitt*. In der Regel muss hierbei ein Betriebsmittel (z. B. der Prozessor) exklusiv genutzt werden.

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

4.2.6 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 für ein Betriebsmittel 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).

4.2.7 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: "Stellen" sind die elementaren Anweisungen der Prozesse, "Transitionen" sind die Übergänge zu den jeweils nächsten Berechnungen.

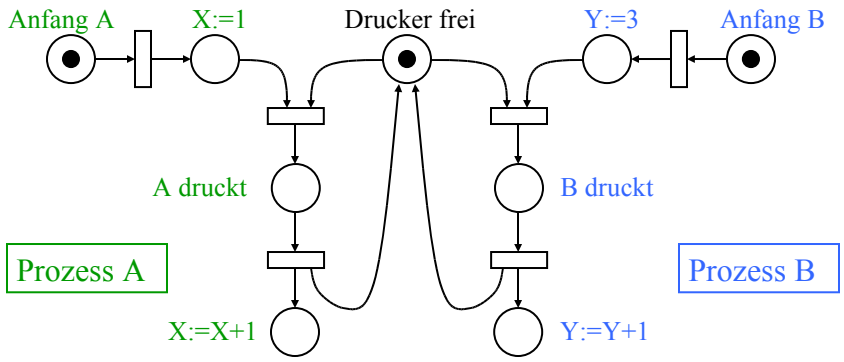
Anschließend realisieren wir den wechselseitigen Ausschluss softwaremäßig durch geeignete Kontrollvariable in den beiden Prozessen.

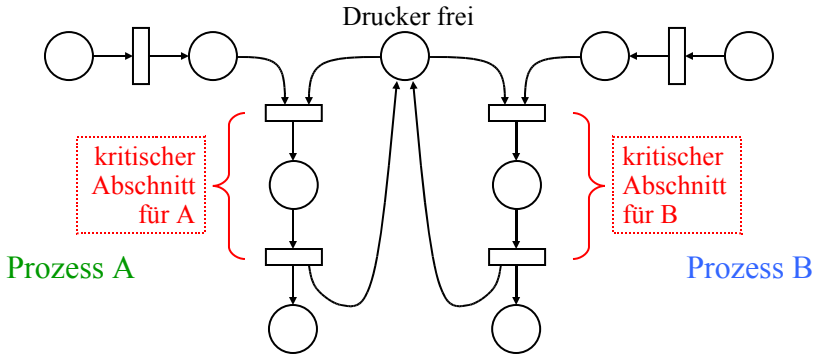
Beispielprogramm: Zwei Prozesse wollen X bzw. Y drucken.

[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 würden dann fälschlicherweise gleichzeitig ihre kritischen Abschnitte betreten können.

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.

4.2.8 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.)

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 um 500 v. Chr. in Gebrauch, vor allem als Feuerzeichen-Übertragung nachts.

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 <u>await</u> 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.

<u>4.2.8 b Definition</u> (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 Betriebsmitteln mehreren auf sie zugreifenden Prozessen zur Verfügung zu stellen, ohne dass eine Verklemmung eintreten kann.
(Vgl. "Scheduler" in Vorlesungen über Betriebssysteme.)

<u>Problem:</u> Kann man durch ein Programm, das nur unsere Anweisungen benutzt (also keine allgemeinen Semaphore kennt), den wechselseitigen Ausschluss sicherstellen, <u>falls keine unterbrechbaren Operationen vorliegen</u> (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: Führe eine Variable ein, die nur die Werte PA und PB annehmen kann.

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

 (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  type prozess is (PA, PB);
        Nr: prozess := PA;  X, Y: Integer;

 (A:: while true do                |   B:: while true do
        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
      od                           |      od    )          ]
```


4.2.9 Problemformulierung: Gesucht wird also eine Softwarelösung, bei der jeder Prozess unabhängig vom anderen ist, außer in dem Fall, dass beide zur gleichen Zeit 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, für "Vorbereitung A" bzw. B und "Nachbereitung A" bzw. B eine Lösung des allgemeinen Problems zu finden:

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


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.

4.2.10 Die Lösung von Peterson (1981):

(OR ist hier das Oder in einem Booleschen Ausdruck)

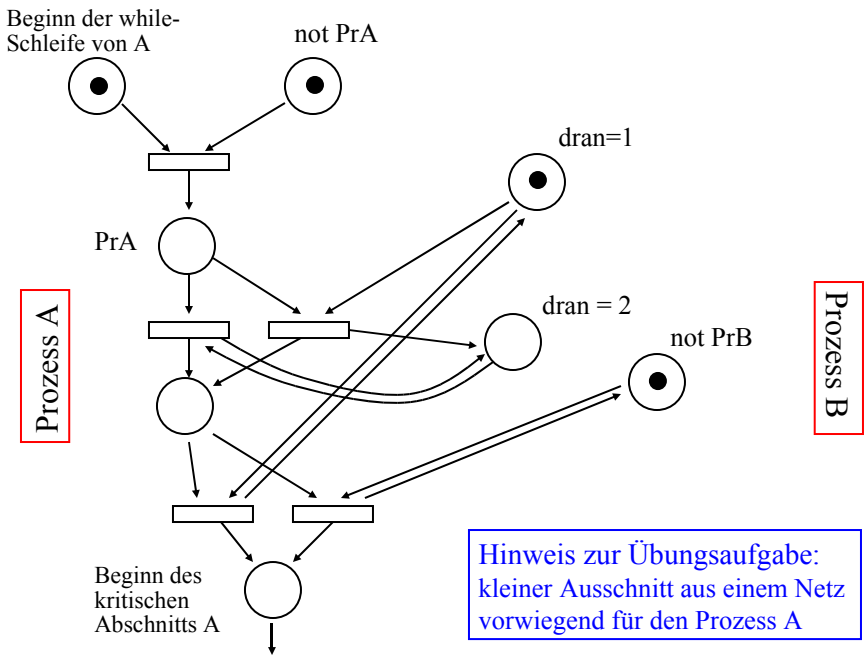
```
[ local  dran: Integer:=1;  PrA, PrB: Boolean:= false;

(A:: while true do                    B:: while true do
       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;
       ...                                   ...
    od                                    od                )        ]
```

In der Vorlesung wird an der Tafel 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.

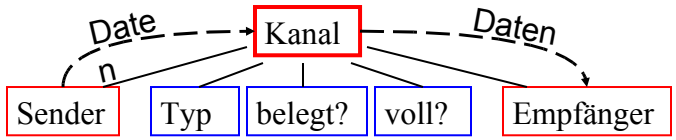


Hinweis zur Übungsaufgabe:
kleiner Ausschnitt aus einem Netz
vorwiegend für den Prozess A

4.2.11 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 Kommunikation 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 seiner Variablen X zugewiesen werden können. (CH ist eine Variable des eingeführten Datentyps.). 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.

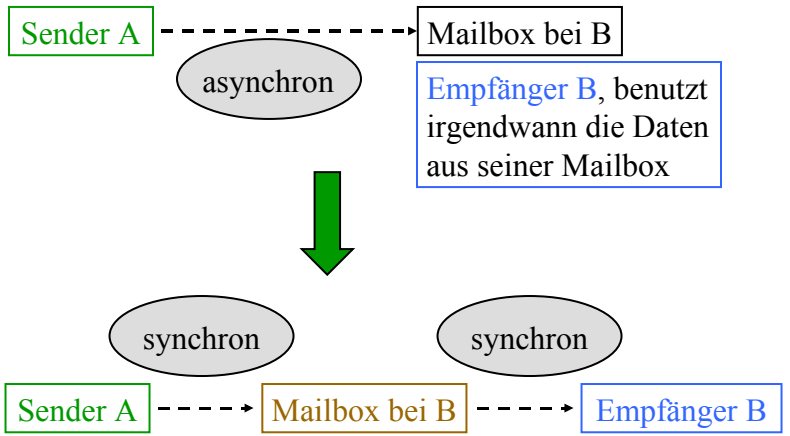
Trifft auch dies zu, so wird der Wert von X in den Kanal gelegt; anderenfalls erfolgt eine geeignete Fehlermeldung.

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).

4.2.12: 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 4.1.3): 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 mit der Kapazität 10 des Lagers (hier bedeutet "or" die nicht-deterministische Auswahl aus 4.2.2; die erzeugten und verbrauchten Elemente sind vom Datentyp T).

4.2.13 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 weniger Platz benötigen. 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 und bereits im Programmierkurs geübt worden.

5. Zusammenfassung der Vorlesung

5.1 Was wurde vermittelt?

5.2 Was fehlt?

5.3 Was baut später auf dieser Vorlesung auf?

Hier einige Stichwörter.

5.1 Was wurde vermittelt?

Zentrales Thema: "Programmierparadigmen":

Objektorientierung.

Funktionales Vorgehen.

Maschinennahe Aspekte und Übersetzung.

(Ein Hauch von) Nebenläufigkeit.

Im Vordergrund stehen die Modelle und abstrakten Begriffe, auch wenn konkret Scheme und Java gelehrt wurden, was bereits recht viel Zeit kostete. Die tatsächlichen Möglichkeiten können nur durch zusätzliche Vertiefungen vermittelt werden. Sinnvoll wäre daher ein "vielsprachiges" Software-Praktikum, welches zurzeit aber nicht realisierbar ist, sowie eine Theorie-Veranstaltung zu Algorithmen und Kalkülen.

Wichtig sind die konkreten Verfahren, vor allem:

- Sortieren durch Mischen und mit Hilfe von Bäumen
- Teilwortsuche (naiv, Automaten, Knuth-Morris-Pratt, Boyer-Moore)
- Median in linearer Zeit
- Erzeuger-Verbraucher-System (Threads und S/T und Programm)
- Zerlegung in Primfaktoren und n-te Primzahl
- Wurzelberechnungen mit beliebiger Genauigkeit
- Symbolisches Differenzieren
- Potenzmengen
- Attributierung, z. B. für arithmetische Ausdrücke
- Peterson-Verfahren

5.2 Was fehlte noch?

Weitere Paradigmen:
- Prädikatives Programmieren
- Gleichzeitigkeits-Programmierung (systolische Arrays, zelluläre Ansätze, Schaltkreise)
- Spezifizieren, Modellieren (z. B. mit UML)
- Heuristiken (z. B. evolutionäre Algorithmen)
- Grundkonzepte der Betriebssysteme und der Datenbanken

Weitere Standardbeispiele, zum Beispiel
- Anagramme (konnte nur in den Übungen angesprochen werden)
- Komplette Analyse des Teilworterkennungs-Problems
- Flüsse in Graphen
- Zuordnungsprobleme, Scheduling
- Heuristiken für TSP (Problem des Handlungsreisenden)
- Basisalgorithmen für data mining

5.3 Was baut später auf dieser Vorlesung auf?

Die Vorlesung "Einführung in die Informatik III" soll die "Grundlagen"-Vorlesungen ab dem 5. Semester vorbereiten. Die Hörer(innen) haben einen erweiterten Einblick in die Algorithmik und die gängigen Sprachen erhalten, in den Aufbau von Informatiksystemen und in die prinzipiellen Herangehensweisen.

Wichtig ist die Erfahrung, dass man Probleme mit recht unterschiedlichen Ansätzen in Angriff nehmen und dann mit unterschiedlicher Qualität lösen kann.

Weiterhin wurde viel Wert auf das Handwerkliche gelegt: In kommenden Vorlesungen sind stets (kleine) Programme zu schreiben - und hier wird durchaus eine gewisse "beherrschte Vielsprachigkeit" erwartet.

5.4 Schluss:

Fortbildungsveranstaltung für Informatik interessierte Studierende

Einladung
zu einem lehrreichen Vortrag über

E-Weihnachten

Donnerstag, **6.12.07**, um **16:53 Uhr** im Hörsaal 38.01, Informatikgebäude, Universitätsstr. 38

Anfang Dezember wird der Referent vergangener Jahre vor den Studierenden der Universität Stuttgart einen fundamentalen Vortrag aus der himmlischen Informatik halten. Noch unerforschte Fragen wie "Wo liegen die Wurzeln der Weihnachtsbäume?" oder "Gehört eine Kinder verträgliche Rutenführung zur outcome-Orientierung jeder Knecht-Ruprecht-Ausbildung?" bleiben erneut unbeantwortet und werden daher auch nicht angesprochen. Da der Vortragende im Dezember zeitlich stark beansprucht ist, bitten wir alle Hörerinnen und Hörer, besonders pünktlich zu erscheinen, bescheiden ihre Plätze einzunehmen, sich äußerst still zu verhalten und den Vortrag nicht durch Besserwisserei zu stören.

Wir wünschen eine erfolgreiche Vorbereitung auf die Vordiplomsklausur am 5. März 2008 um 9:00 Uhr.