universität innsbruck
Institut für Informatik

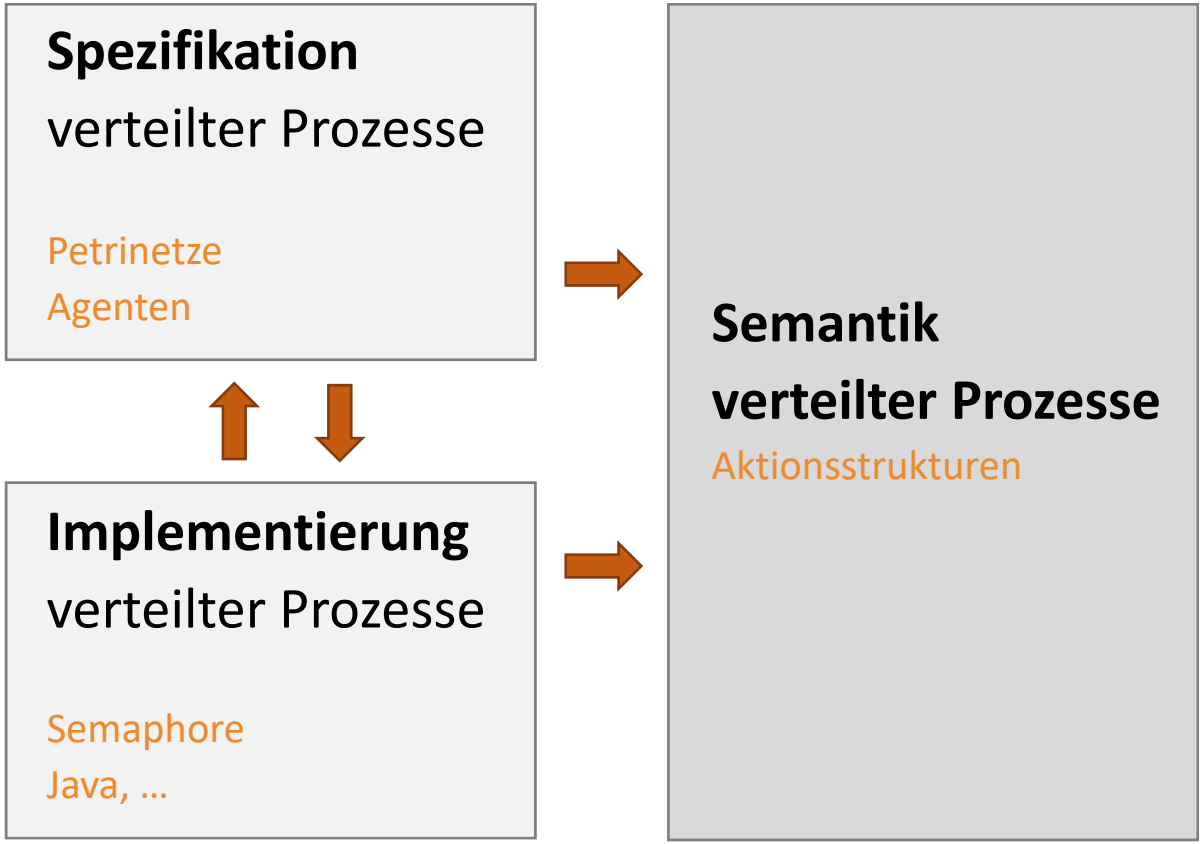
Software Engineering

5. Formale Spezifikation
Teil 2 – Nebenläufige Systeme

Ruth Breu

Übersicht

Übersicht


Spezifikation
verteilter Prozesse
Petrinetze
Agenten
Implementierung
verteilter Prozesse
Semaphore
Java, …
Semantik
verteilter Prozesse
Aktionsstrukturen

5.3.1 Semaphore

- Beispiel eines Implementierungskonzepts für nebenläufige Systeme
- Basis: Koordination nebenläufig ablaufender Prozesse durch Zugriff auf gemeinsame Variablen
- **Kritisches Betriebsmittel:** Betriebsmittel (z.B. Variable), auf das mehrere Prozesse zugreifen
- **Kritische Aktion/kritischer Abschnitt:** Kette von Aktionen, die auf ein gemeinsames Betriebsmittel zugreifen
- Prozesse müssen vor dem Eintritt in einen kritischen Abschnitt synchronisiert werden – hier: Verwendung von Semaphoren

Semaphor

Semaphor (Dijkstra, 1968): Datentyp, der aus einer integer-Variable s besteht, auf der nur die zwei Operationen P und V zulässig sind:

```
sema int s   wird mit einem Wert ≥ 0 initialisiert
P(s):        if (s > 0) s- -;
             else  blockiere aufrufenden Prozess
V(s):        s++;
             wecke einen blockierten Prozess, falls es einen gibt
```

- Jeweils nur ein Prozess kann eine Operation auf einem Semaphor ausführen
- Jeder Semaphor ist mit einer Warteschlange von blockierten Prozessen verbunden

Boolescher Semaphor

- sema bool s
- nimmt nur Werte 0, 1 (bzw. true, false) an

Beispiel 1: Gegenseitiger Ausschluss

Mutual Exclusion: zwei Programme sollen nebenläufig ausgeführt werden, aber nur jeweils ein Programm soll sich in einem kritischen Abschnitt befinden können ()

```
sema int mutex = 1;

program1() {
   while(true) {
      P(mutex);
      /* kritischer Abschnitt */
      V(mutex);
      /* unkritischer Abschnitt */
   }
}
```

```
program2() {
   while(true) {
      P(mutex);
      /* kritischer Abschnitt */
      V(mutex);
      /* unkritischer Abschnitt */
   }
}
```

auch für N Prozesse anwendbar

Beispiel 2: Erzeuger-Verbraucher

Producer/Consumer: zwei Programme sollen strikt abwechselnd auf einen Puffer zugreifen

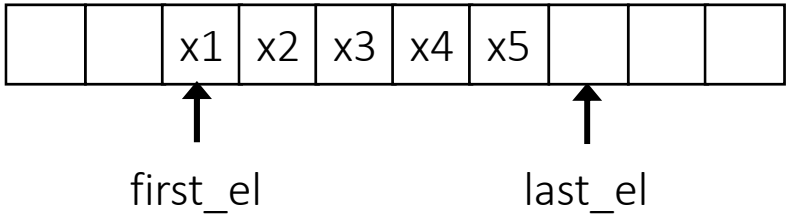
```
sema int leer = 1;
sema int voll = 0;
int puffer;

producer() {
   int nachricht;
   while(true) {
      /* produziere nachricht */
      P(leer);
      puffer = nachricht;
      V(voll);
   }
}
```

```
consumer() {
   int nachricht;
   while(true) {
      P(voll);
      nachricht = puffer;
      V(leer);
      /* verarbeite nachricht */
   }
}
```

Beispiel 3: Producer/Consumer mit N-elementigem Puffer

Gegeben ist ein Puffer mit N Elementen

```
int N;
int[ ] buffer = new int[N];
int anzahl = 0;
int first_el = 0; int last_el = 0;
```



Gegeben seien außerdem Operationen `insert` und `remove`, die den Puffer nach der FIFO-Strategie füllen bzw. leeren

```
void insert(int x) {
    buffer[last_el] = x;
    last_el = (last_el+1) % N;
    anzahl ++;
}
```

```
int remove() {
    int tmp = first_el;
    first_el = (first_el+1) % N;
    anzahl - -;
    return buffer[tmp];  }
```

Es sollen Operationen `produce` und `consume` (auf der Basis von `insert` bzw. `remove`) entwickelt werden, die sicherstellen,

1. dass sich höchstens ein Prozess in einem kritischen Abschnitt befindet
2. dass es keinen Über- oder Unterlauf des Puffers gibt

Lösung

```
int N;
int[ ] buffer = new int[N];
int anzahl = 0;
int first_el = 0;
int last_el = 0;
sema int mutex = 1;
sema int frei = N;
sema int belegt = 0;

producer() {
    int nachricht;
    while(true) {
    /* produziere nachricht */
    p(frei);
    p(mutex);
    insert(nachricht);
    v(mutex);
    v(belegt);
    }
}
```

```
consumer() {
   int nachricht;
   while(true) {
      p(belegt);
      p(mutex);
      nachricht = remove();
      v(mutex);
      v(frei);
      /* verarbeite nachricht */
   }
}
```

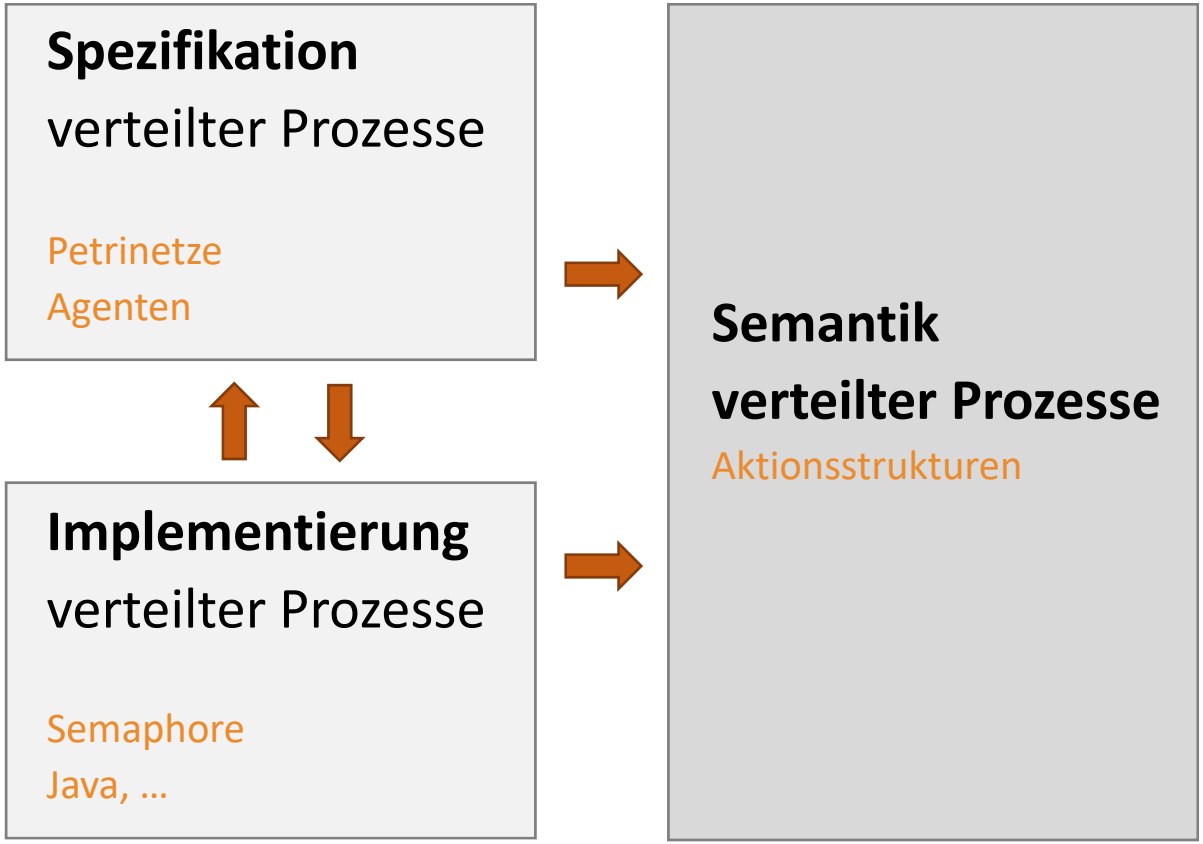
Typische Phänomene bei verteilten Systemen

- Verklemmung (Deadlock)
  - Kein Prozess kann fortfahren, weil alle auf eine Bedingung warten, die nur ein anderer (ebenfalls wartender) Prozess beseitigen kann
- Aushungerung (Starvation)
  - Ein Prozess wird dauerhaft nicht mehr zur Ausführung zugelassen, obwohl er noch nicht beendet ist
- Nicht-Determinismus
  - Das Systemverhalten ist nicht vorhersehbar (z.B. durch nicht-bestimmtes Verhalten beim Umschalten der Prozesse, durch nicht-synchronisierten Zugriff auf gemeinsame Variablen)
- Nicht-Terminierung
  - Häufig führen Prozesse wiederkehrende Aufgaben durch und terminieren dadurch nicht (z.B. auf Betriebssystemebene)

Semantik nebenläufiger Systeme

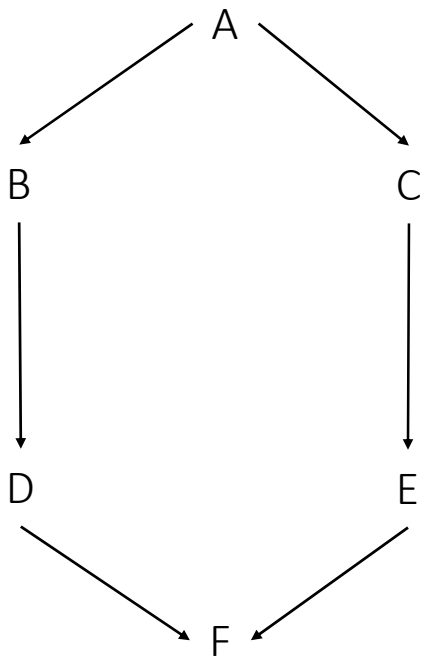
- Semantik sequentieller Systeme: Zustandsänderungen
- Um die Semantik nebenläufiger Systeme zu beschreiben, muss man ein anderes semantisches Modell wählen:
  - Nebenläufige Programme terminieren oft nicht, d.h. der zugeordnete Endzustand ist undefiniert
  - Nebenläufige Programme haben oft ein nichtdeterministisches Verhalten, d.h. der Endzustand ist nicht determiniert
  - Um Aussagen über nebenläufige Programme treffen zu können, muss der Programmablauf mit seinen Aktionen im semantischen Bereich abgebildet werden
  - ➔ Konzept der Aktionsstrukturen (Kapitel 5.3.2)
  - Das Konzept der Aktionsstrukturen dient als Theorie zur Beschreibung der Semantik nebenläufiger Programme
  - Die semantische Abbildung selbst (z.B. die Abbildung nebenläufiger Java-Programme in Aktionsstrukturen) werden wir nur skizzieren

Übersicht


Spezifikation
verteilter Prozesse
Petrinetze
Agenten
Implementierung
verteilter Prozesse
Semaphore
Java, …
Semantik
verteilter Prozesse
Aktionsstrukturen

5.3.2 Aktionsstrukturen

- Beispiel: **Prozess** des Einsteigens von vier Personen in ein zweitüriges Auto



Ereignisdiagramm

Ereignis	*Aktion*
A	Aufsperren des PKWs
B	Linke Tür öffnen, 1. Person steigt ein (durch linke Tür nach hinten)
C	Rechte Tür öffnen, 2. Person steigt ein (durch rechte Tür nach hinten)
D	Fahrer steigt ein und schließt die Tür
E	Beifahrer steigt ein und schließt die Tür
F	PKW fährt ab

„C endet, bevor E beginnt ($C \leq E$)“

Definition: Aktionsstruktur

Gegeben sei eine Menge E von Ereignissen (engl. *events*) und eine Menge A von Aktionen (engl. *actions*). Dann heißt das Tripel $p = (E_0, \leq_0, \alpha)$ Prozess oder Aktionsstruktur, falls folgende Aussagen gelten:

- $E_0 \subseteq E$
- $\leq_0$ ist eine partielle Ordnung über E_0
- $\alpha: E \rightarrow A$

Ein Prozess heißt endlich, wenn die Menge E_0 endlich ist.

NB: α muss nicht injektiv sein, d.h. Aktionen können wiederholt ausgeführt werden, werden dann aber durch unterschiedliche Ereignisse modelliert.

Im weiteren wird die Darstellung meist vereinfacht und zwischen Ereignissen und Aktionen nicht unterschieden.

Parallelität

Für einen Prozess $(E_0, \leq_0, \alpha)$ heißen zwei Ereignisse $e1, e2 \in E_0$ $(e1 \neq e2)$ parallel, wenn

$$\neg (e1 \leq e2) \wedge \neg (e2 \leq e1),$$

sonst heißen e1, e2 sequentiell. Parallele Ereignisse stehen also in der Ordnung nicht in Beziehung.

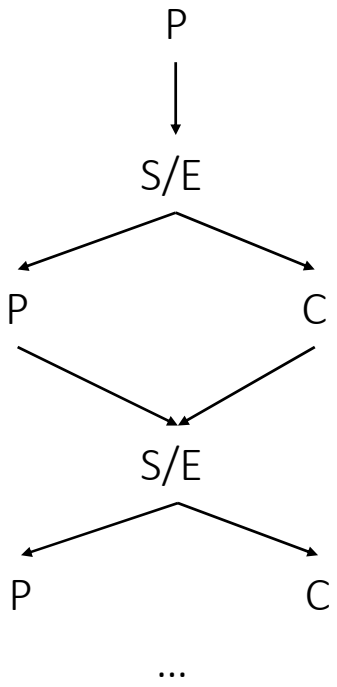
Ein Prozess $(E_0, \leq_0, \alpha)$ heißt sequentiell, wenn kein Paar von parallelen Ereignissen existiert, d.h., wenn $\leq_0$ eine lineare Ordnung ist.

Beispiel:

e1 –> e2 –> e3 –> e4 –> e5 ist ein sequentieller Prozess.

Beispiel (1)

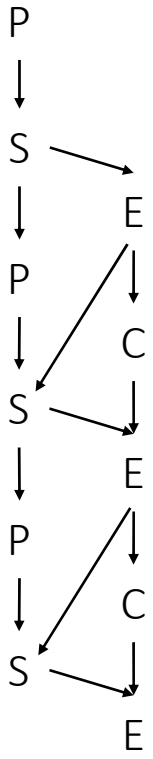
Erzeuger-/Verbraucherprozess mit synchronem Senden/Empfangen



P = Produzieren
C = Verarbeiten
S/E = Senden und Empfangen

Beispiel (2)

Erzeuger-/Verbraucherprozess mit asynchronem Nachrichtenaustausch



einelementiger Puffer

P = Produzieren
C = Verarbeiten
S = Senden
E = Empfangen

Sequentialisierung und Interleaving-Modell

Stellen Sie sich einen Beobachter vor, der die Ereignisse eines Prozesses und die entsprechenden Aktionen in einem sequentiellen Ablaufprotokoll festhält.

- Parallele Aktionen werden dabei in eine zufällige Reihenfolge gebracht
- Das Ergebnis der Beobachtung ist ein sequentieller Prozess, der einer zufällig gewählten Sequentialisierung entspricht
- Aus der Gesamtmenge dieser Beobachtungen lässt sich der Prozess eindeutig rekonstruieren

Werden die vollständigen Sequentialisierungen, und nicht die Aktionsstruktur als Modell eines Systems verwendet, spricht man von einem Interleaving-Modell.

Sequentialisierung

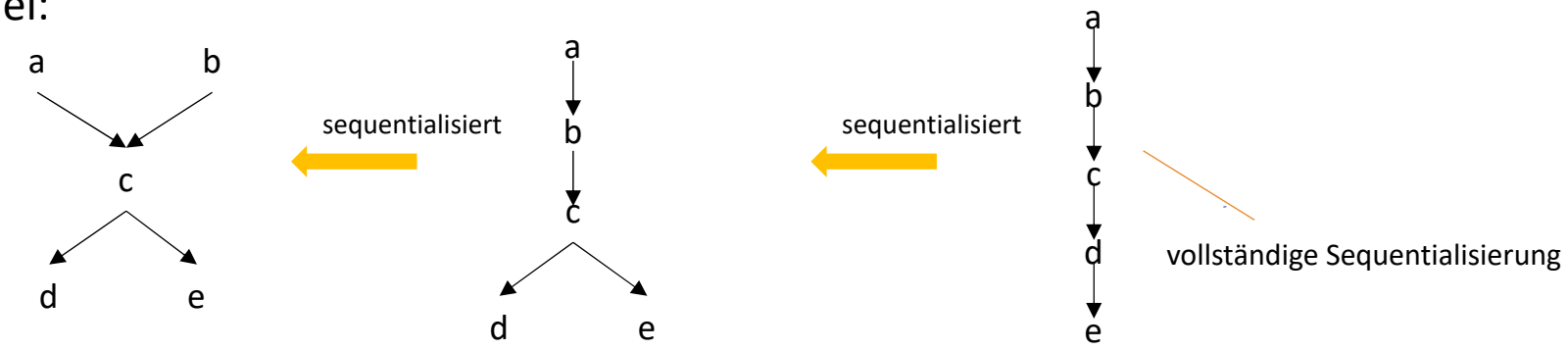
Gegenstand: Prozesse mit identischer Ereignismenge, die sich nur in der partiellen Ordnung unterscheiden.

Ein Prozess $(E_1, \leq_1, \alpha_1)$ heißt **Sequentialisierung** eines Prozesses $(E_2, \leq_2, \alpha_2)$, falls

1. $E_1 = E_2$
2. $\leq_2 \subseteq \leq_1$ (d.h. $\forall e1, e2 \in E_1: e1 \leq_2 e2 \Rightarrow e1 \leq_1 e2$)
3. $\alpha_1 = \alpha_2$

Ist dann $(E_1, \leq_1, \alpha_1)$ sequentiell, so heißt die Sequentialisierung **vollständig**.

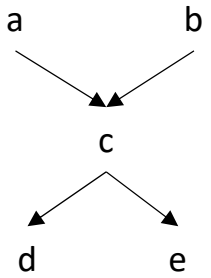
Beispiel:



Satz

Jede Aktionsstruktur $(E_1, \leq_1, \alpha_1)$ ist durch die Menge ihrer vollständigen Sequentialisierungen eindeutig bestimmt.

Aktionsstruktur



äquivalente Repräsentation
im Interleaving-Modell

Menge von „Traces“

a – b – c – d - e
b – a – c – d – e
a – b – c – e – d
b – a – c – e – d

Teilprozesse und Präfixe

Seien $p_1 = (E_1, \leq_1, \alpha_1)$ und $p_2 = (E_2, \leq_2, \alpha_2)$ Prozesse. Gilt folgende Beziehung für p_1 und p_2:

$$E_1 \subseteq E_2,$$

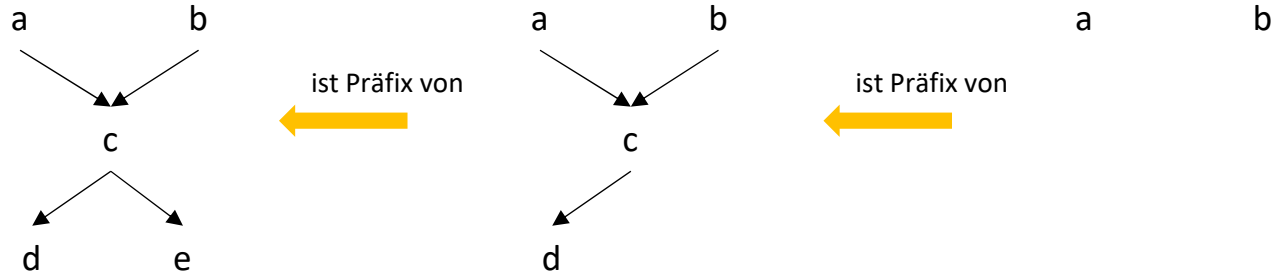
$$\alpha_2|_{E_1} = \alpha_1, \leq_2|_{E_1 \times E1} = \leq_1,$$

so heißt p_1 Teilprozess von p_2. Gilt zusätzlich die Formel

$$\forall\, e \in E_2, d \in E_1 : e \leq_2 d \Rightarrow e \in E_1,$$

so heißt der Prozess p_1 Anfang oder Präfix des Prozesses p_2.
Wir schreiben dann $p_1 \subseteq p_2$

Beispiel:



Bemerkung

- Die Präfixrelation modelliert das Fortschreiten eines Prozesses
  - Jedes Präfix eines Prozesses kann als Beschreibung eines Anfangsabschnitts des Verhaltens des Prozesses verstanden werden
- Endliche Präfixe werden zur Beschreibung unendlicher Prozesse eingesetzt
  - Ein unendlicher Prozess wird i.allg. durch die Menge seiner endlichen Präfixe charakterisiert
  - Dadurch können Eigenschaften unendlicher Prozesse durch Eigenschaften endlicher Präfixe bewiesen werden

Interpretation von Aktionen

- Die Ausführung eines (nebenläufigen) Programms kann als Aktionsstruktur aufgefasst werden. Die Aktionen sind dabei primitive Konstrukte wie Zuweisungen oder Vergleiche

Beispiel: Ausführung des Programms

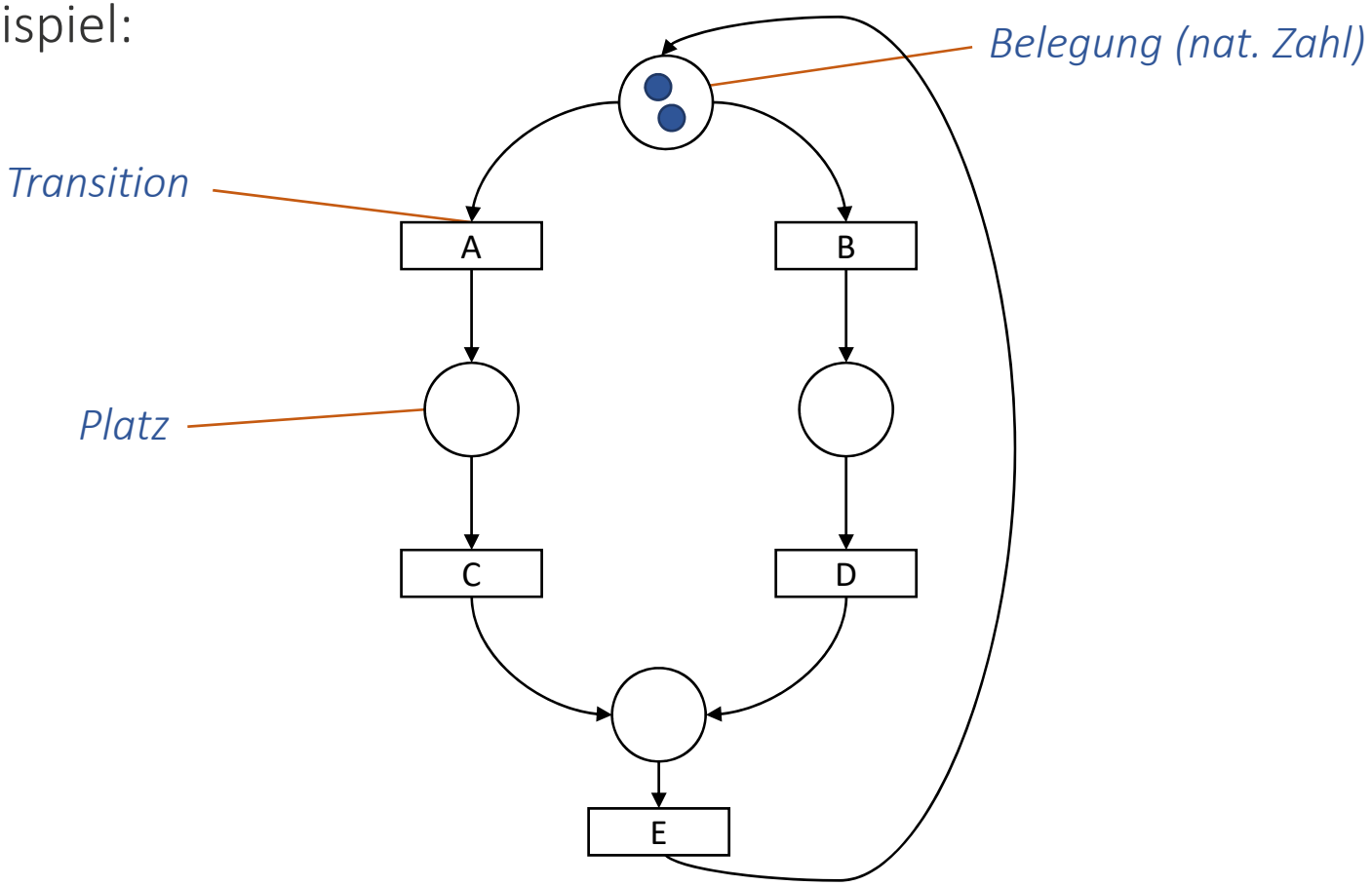
```
y = 1; x = 10;
while (x>0) {
        y * = x;
        x--;
}
```

Ereignis	Aktion
e0	`y = 1;`
e1	`x = 10;`
e2	`x > 0`
e3	`y * = x`
…	…

5.3.3 Petrinetze

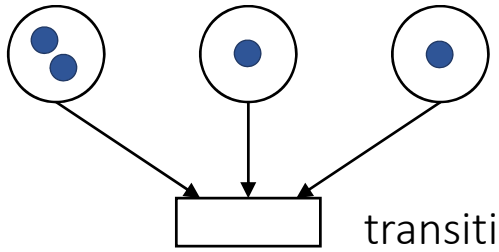
Graphische Modellierung verteilter Systeme

Beispiel:

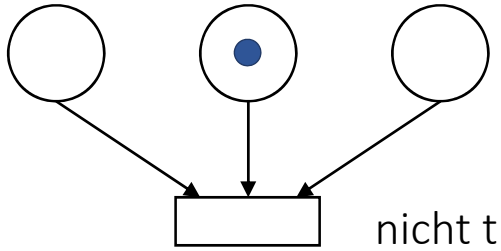


Schalten einer Transition (1)

Eine Transition „kann schalten“ (ist transitionsbereit), wenn die Belegung aller eingehenden Plätze > 0 ist.



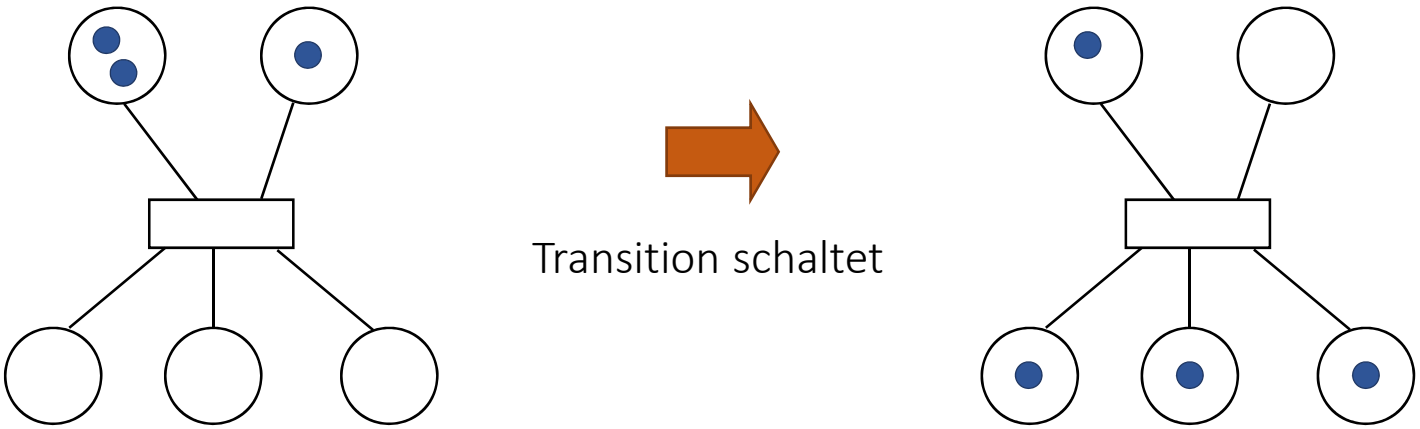
transitionsbereit



nicht transitionsbereit

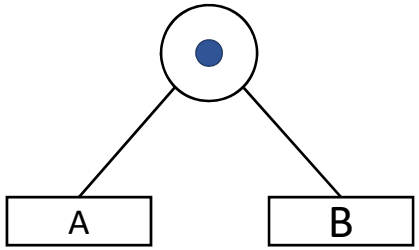
Schalten einer Transition (2)

- Beim Schalten einer Transition wird
  - die Belegung aller eingehenden Plätze um 1 erniedrigt
  - die Belegung aller ausgehenden Plätze um 1 erhöht



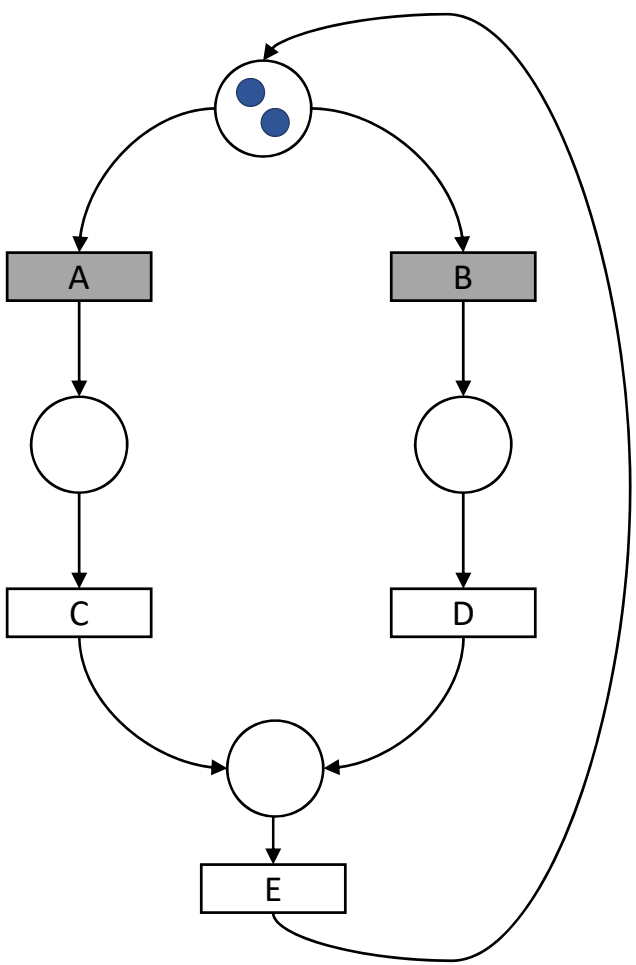
Modellierung von Parallelität

- Im allgemeinen können mehrere Transitionen gleichzeitig schalten
- Zwei Transitionen können in Konflikt stehen:

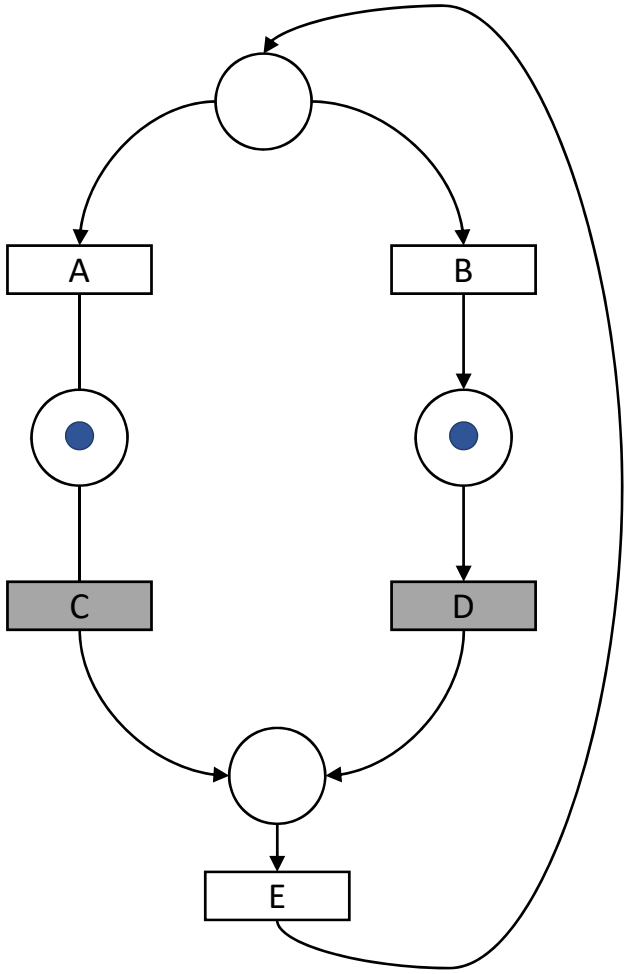


A oder B können schalten, aber nicht A und B gleichzeitig

Beispiel (1)

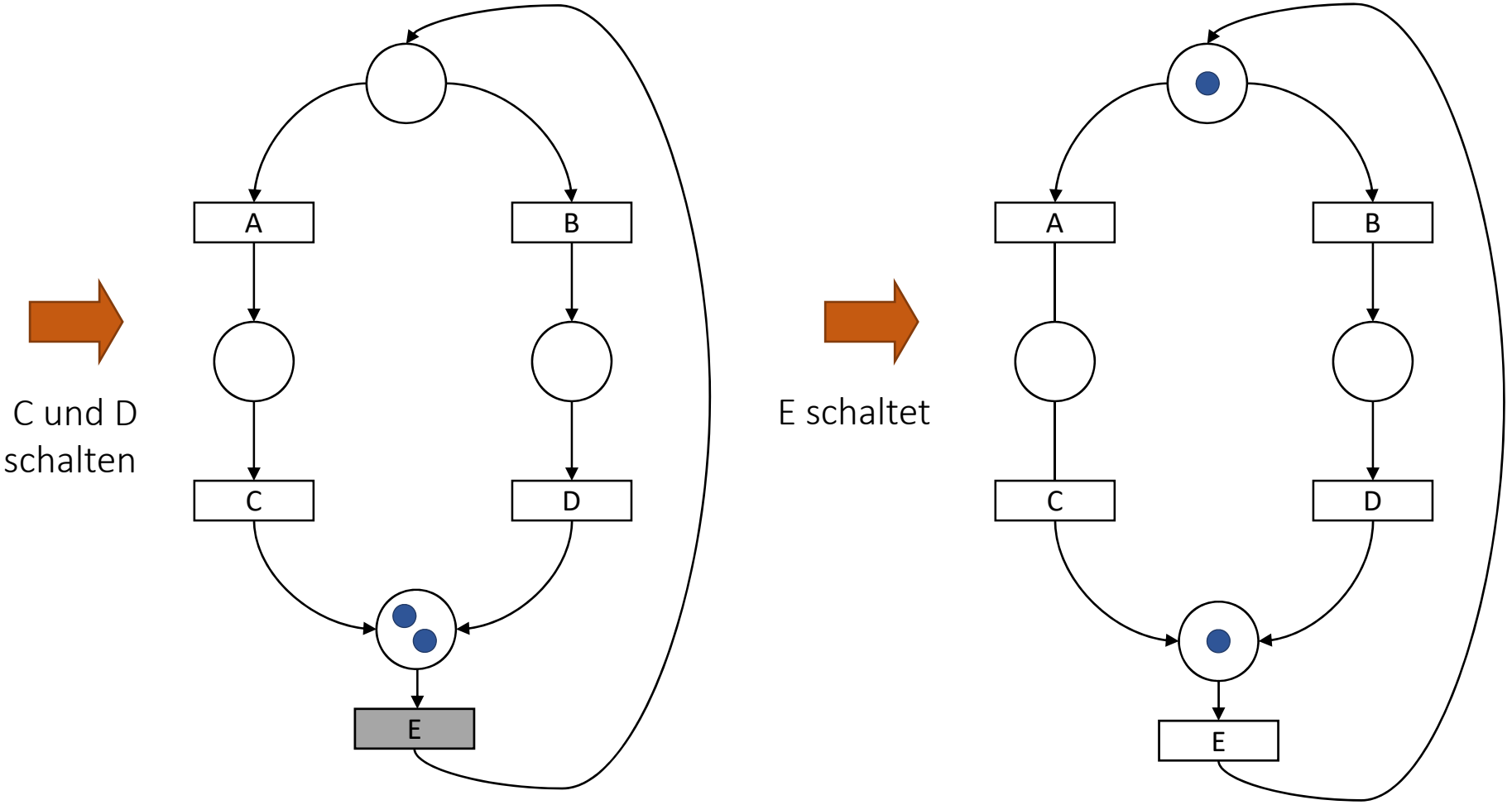


A und B schalten



(schaltbereite Transitionen sind hier grau markiert)

Beispiel (2)



Definition: Petrinetz

Gegeben sei eine Menge T und eine (dazu disjunkte) Menge P.

Ein Petrinetz ist gegeben durch ein Tupel
(T0, P0, R) mit

$T0 \subseteq T$ (Transitionen),

$P0 \subseteq P$ (Plätze),

$R \subseteq (T0 \times P0) \cup (P0 \times T0)$

Allgemein setzen wir T0 und P0 als endlich voraus.

Ein Petrinetz ist ein **bipartiter** Graph.

Definition: Belegung

Für ein Petrinetz (T0, P0, R) heißt jede Abbildung

$$b: P0 \rightarrow IN$$

eine Belegung (d.h.: den Plätzen werden Zahlen zugeordnet.

Für eine Belegung b eines Petrinetzes (T0, P0, R) heißt eine (nichtleere) Teilmenge $K \subseteq T0$ transitionsbereit, wenn gilt:

$$\forall p \in P0: \ | \{ k \in K: (p, k) \in R \} | \leq b(p)$$

Definition: Nachfolgebelegung

Für ein Petrinetz (T0, P0, R) mit Belegung b0 und eine transitionsbereite Menge $K \subseteq T0$ heißt b1 Nachfolgebelegung, falls gilt:

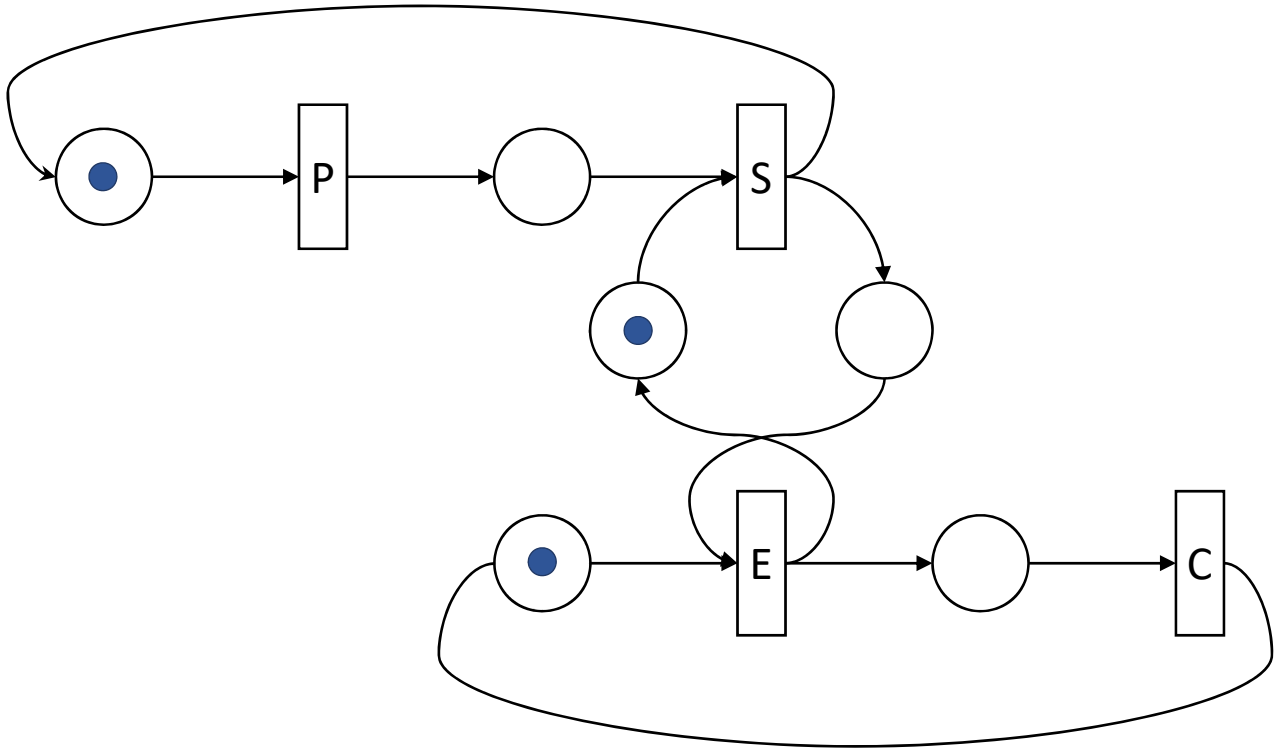
$\forall p \in P0$:

$b1(p) = |\{ k \in K: (k, p) \in R \}| - |\{ k \in K: (p, k) \in R \}| + b0(p)$

Beispiel: Petrinetz für Producer/Consumer

strikt abwechselndes Produzieren/Konsumieren

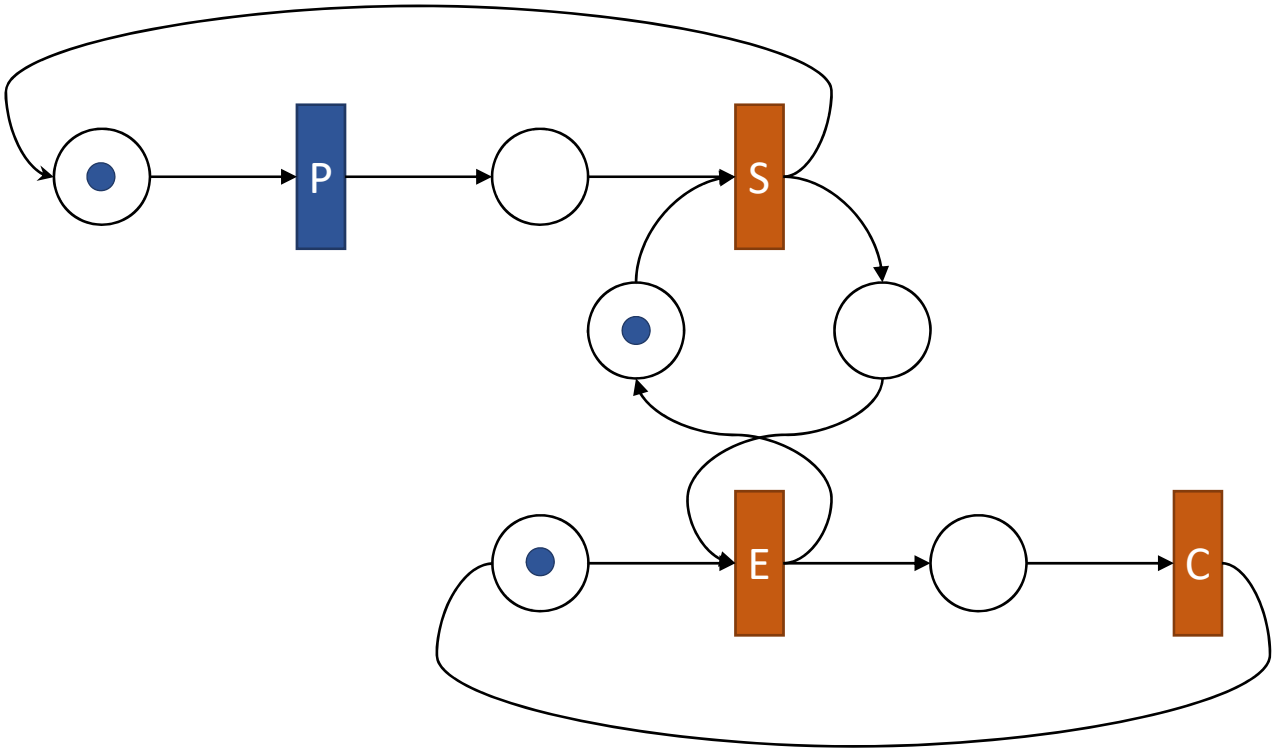
asynchrones Senden/Empfangen



Beispiel: Petrinetz für Producer/Consumer

strikt abwechselndes Produzieren/Konsumieren

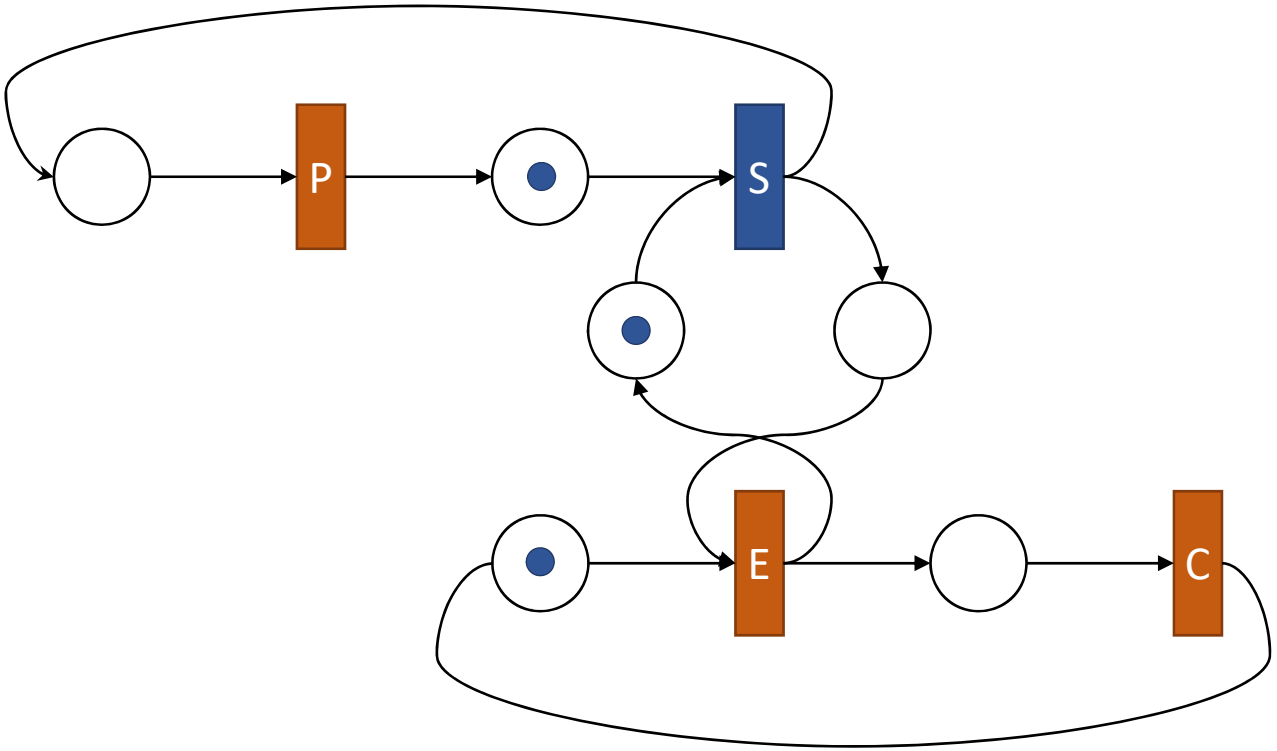
asynchrones Senden/Empfangen



Beispiel: Petrinetz für Producer/Consumer

strikt abwechselndes Produzieren/Konsumieren

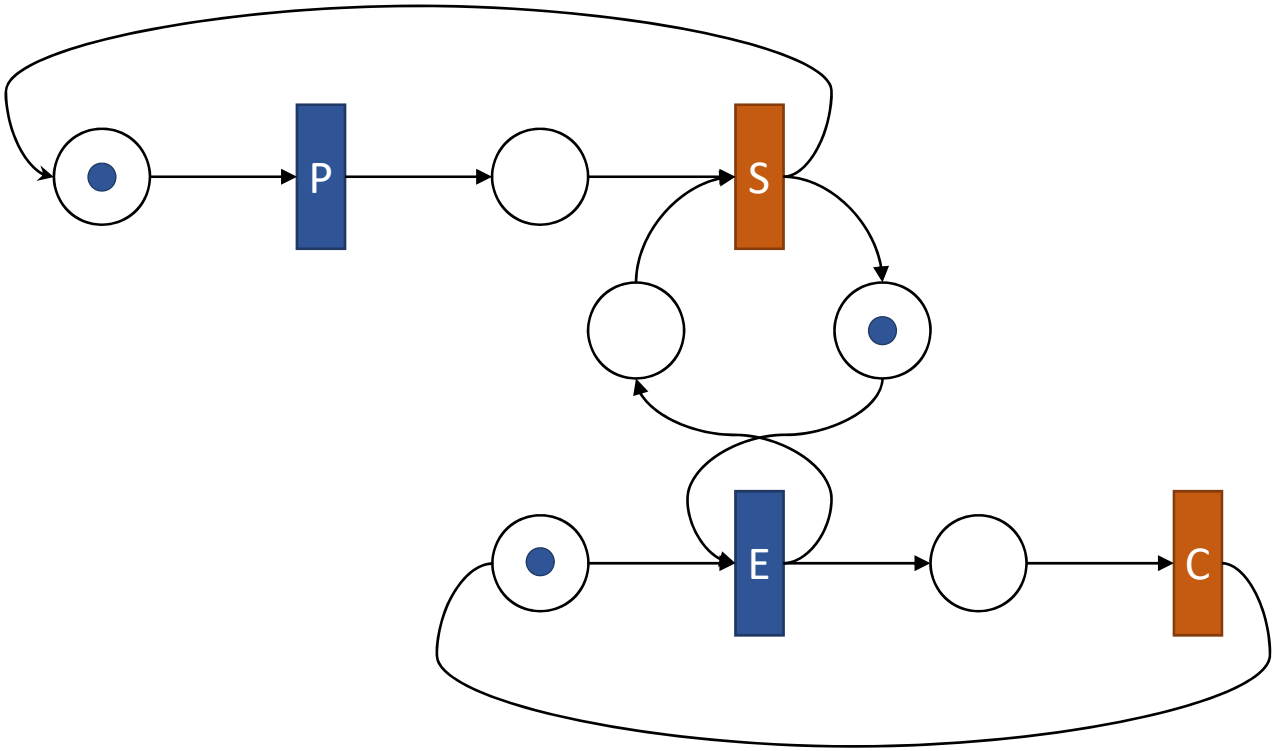
asynchrones Senden/Empfangen



Beispiel: Petrinetz für Producer/Consumer

strikt abwechselndes Produzieren/Konsumieren

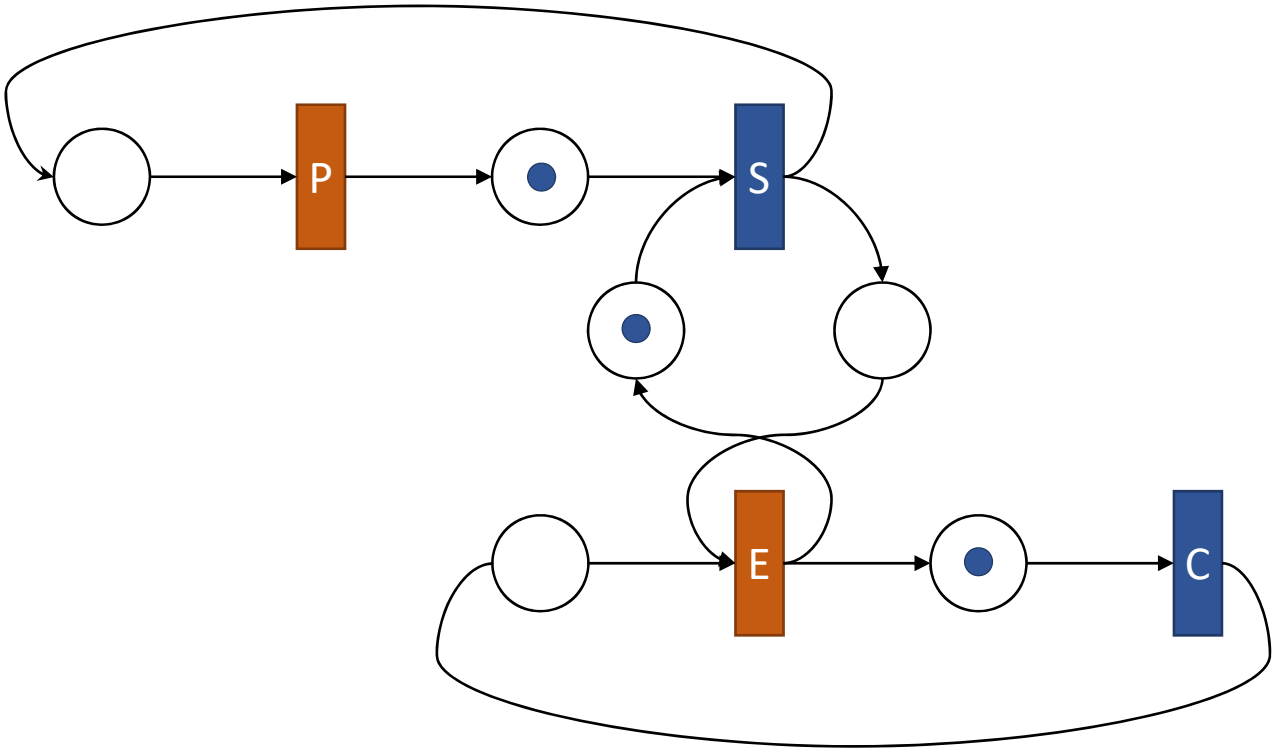
asynchrones Senden/Empfangen



Beispiel: Petrinetz für Producer/Consumer

strikt abwechselndes Produzieren/Konsumieren

asynchrones Senden/Empfangen



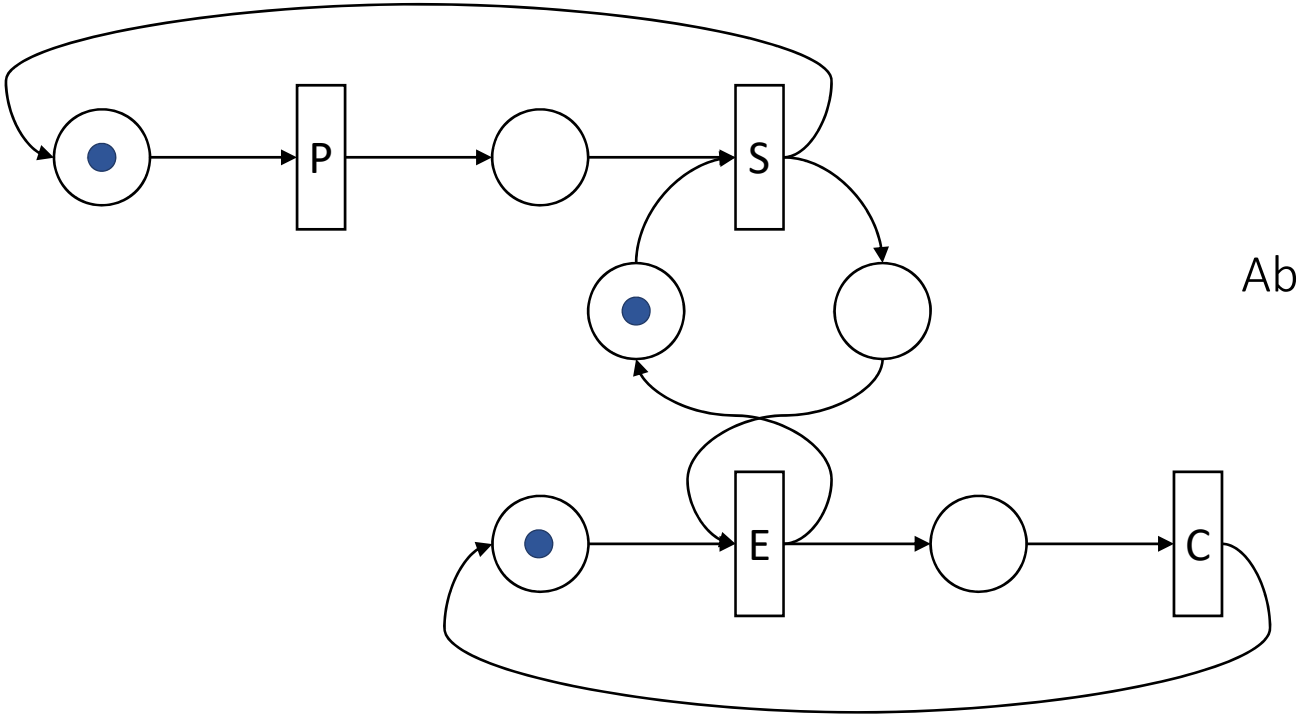
Ablauf eines Petrinetzes

- Gegeben seien
  - ein Petrinetz **N = (T0, P0, R)** und
  - ein Prozess **P = (E0, ≤, α)**, wobei **α: E0 → T0**
- Für eine Anfangsbelegung **b0** heißt der Prozess P **Ablauf** des Petrinetzes N mit Endbelegung **b1**, falls P einem Verhalten des Petrinetzes mit Anfangsbelegung **b0** entspricht

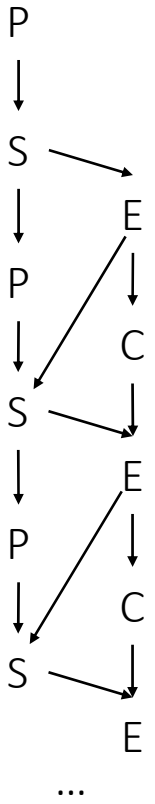
Wir schreiben dann **$b0 \rightarrow_p b1$**

- Wir sagen auch: Die Belegung **b1** ist von **b0** aus **erreichbar**

Beispiel



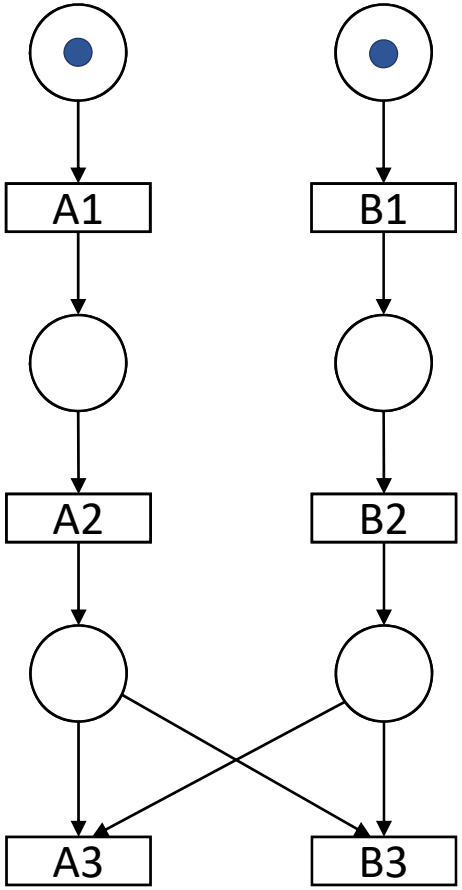
Ablauf:



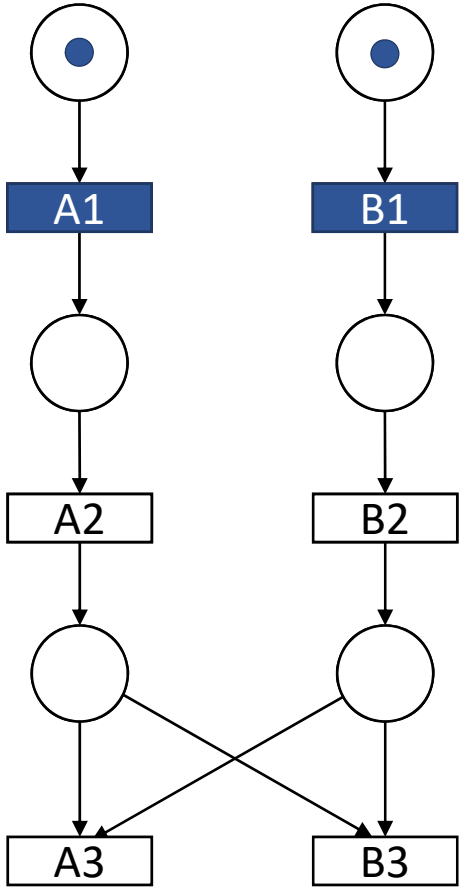
Abläufe – Präfixe und Sequentialisierungen

Ist p Ablauf eines Petrinetzes N mit Anfangsbelegung b0, so ist jedes Präfix von p und jede Sequentialisierung von p Ablauf von N mit Anfangsbelegung b0.

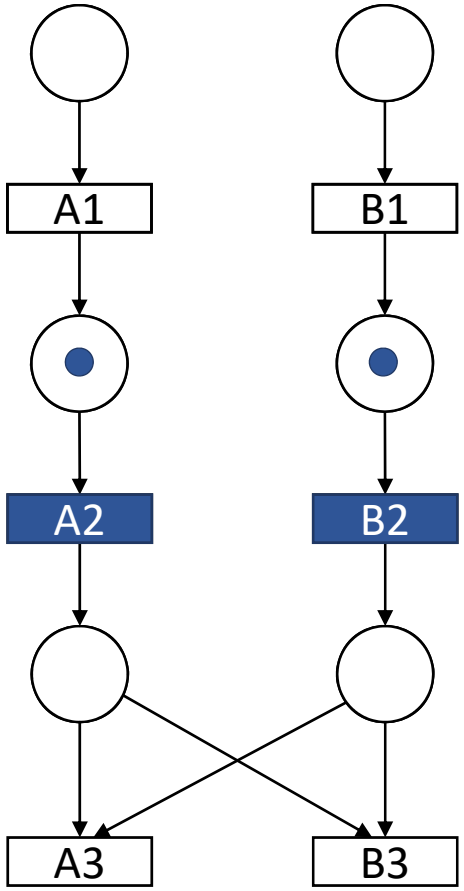
Beispiel



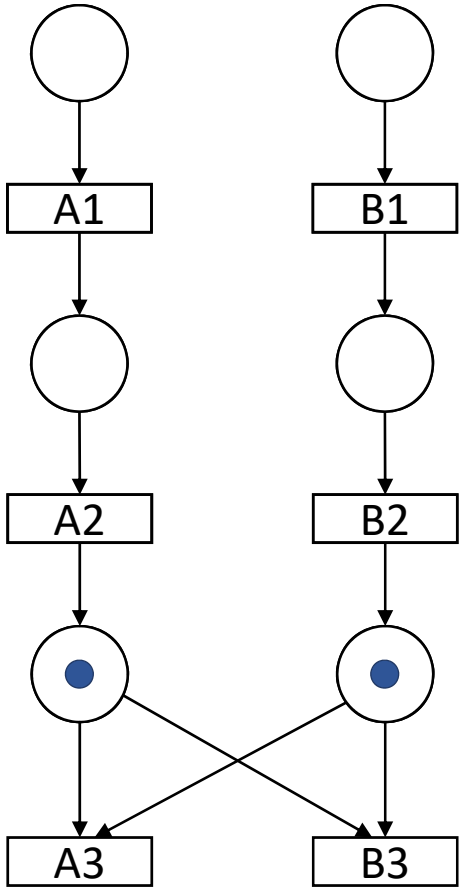
Beispiel



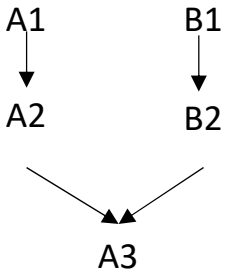
Beispiel



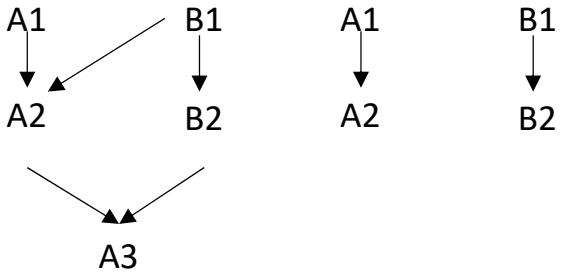
Beispiel



Abläufe z.B.



„maximal parallel“,
„vollständig“



„vollständig“

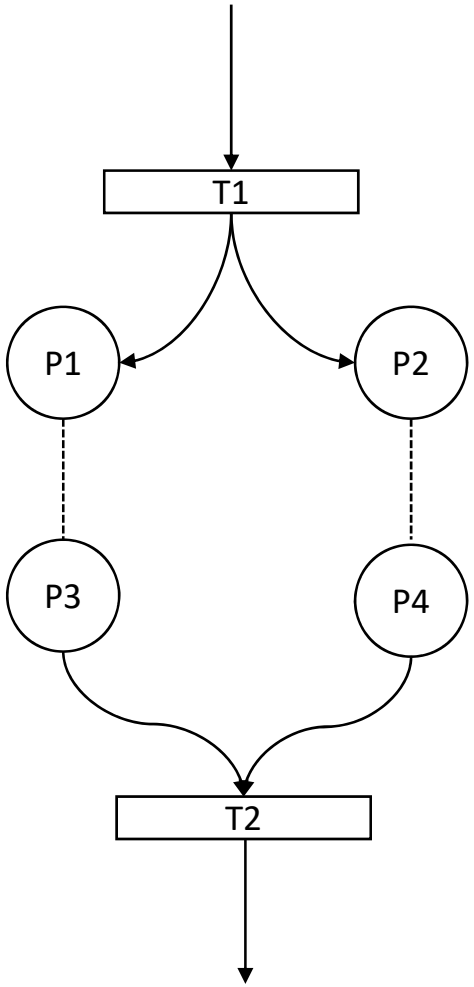
A1 -> B1 -> A2 -> B2 „sequentiell“
A1 -> A2 -> B1 -> B2 -> A3 „sequentiell“, „vollständig“

Vollständiger Ablauf

- Ein Prozess $P = (E0, \leq, \alpha)$ heißt vollständiger Ablauf eines Petrinetzes N mit Belegung b0, falls eine der beiden folgenden Bedingungen gilt:
  - Die Ereignismenge E0 ist unendlich und jedes endliche Präfix von P ist Ablauf des Netzes N mit Anfangsbelegung b0
  - Die Ereignismenge E0 ist endlich und P ist Ablauf des Netzes N für die Anfangsbelegung b0 und die Endbelegung b1, und für die Belegung b1 existiert keine nichtleere transitionsbereite Menge

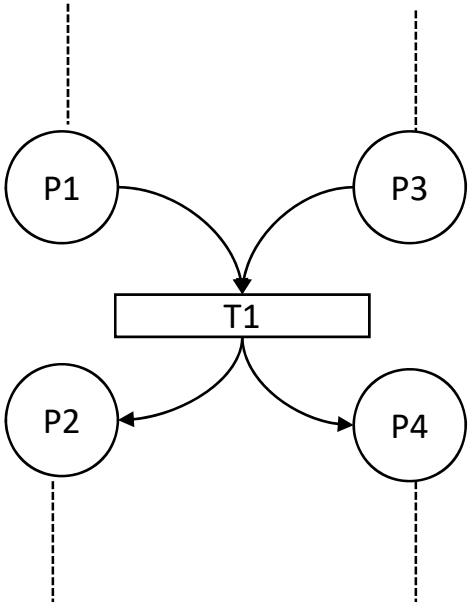
Petrinetz-Patterns (1)

Parallele Abläufe

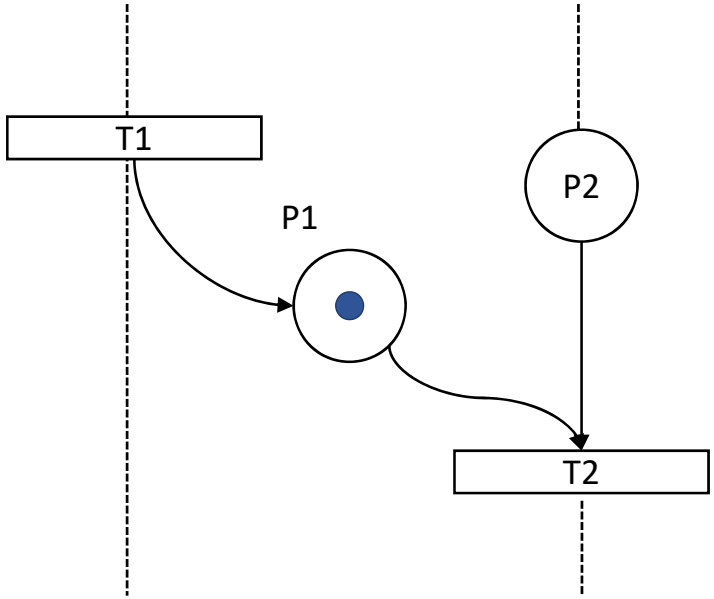


Petrinetz-Patterns (2)

Synchronisation



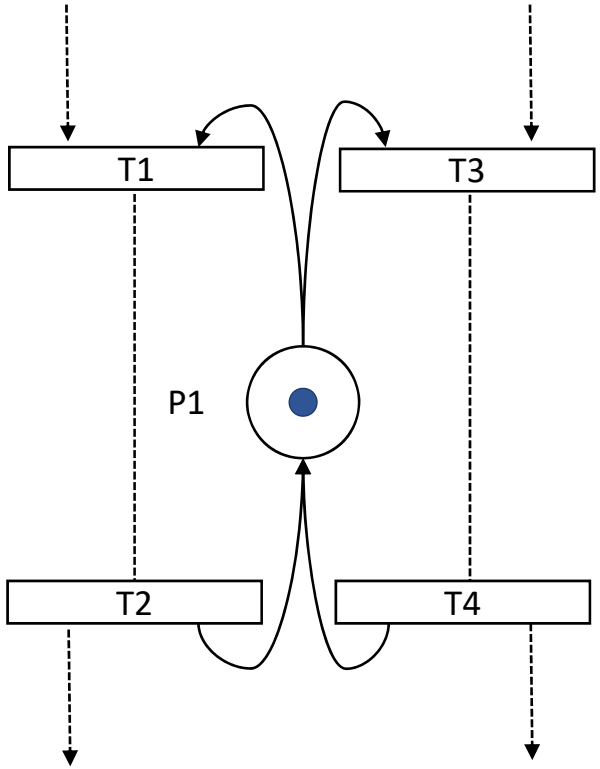
Rendez-vous



Semaphor

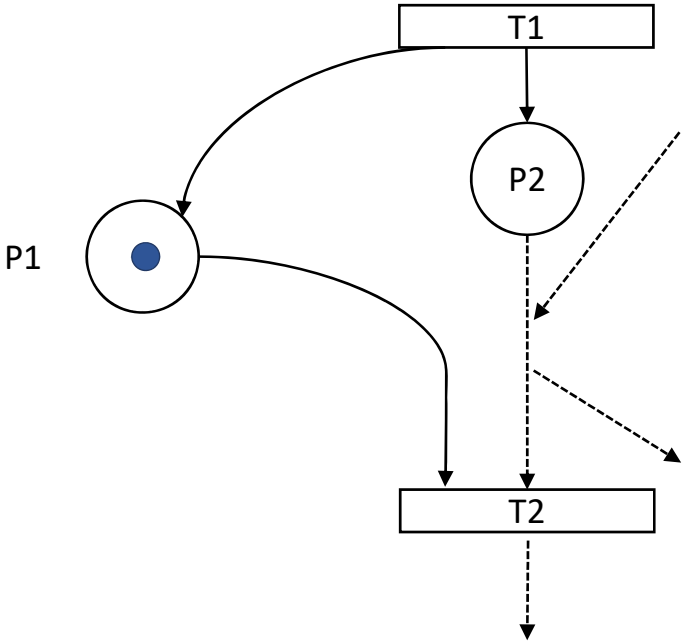
Petrinetz-Patterns (3)

Teilen von Ressourcen, gegenseitiger Ausschluss

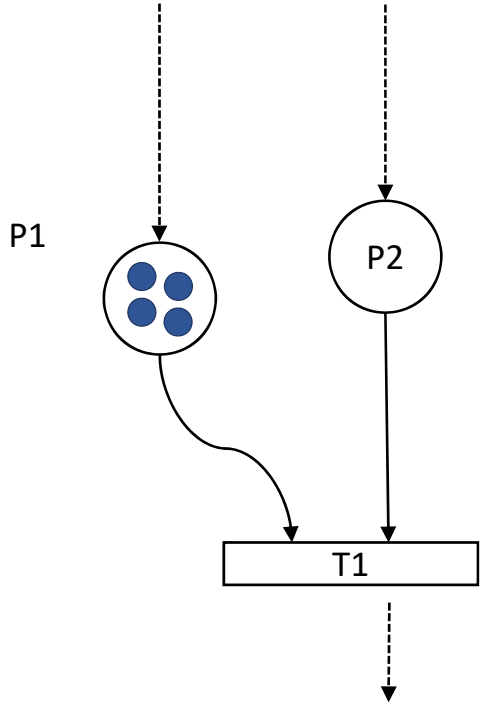


Petrinetz-Patterns (4)

Zählende Petrinetze



Zählen der durchgeführten Transitionen

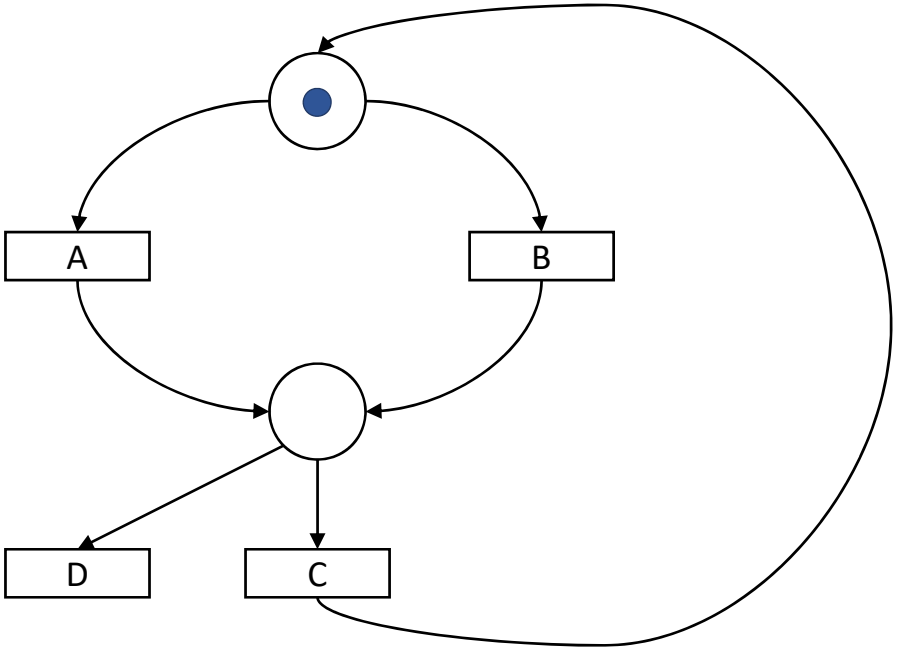


Setzen einer Schranke

Fragestellungen für Petrinetze

- Welche Belegungen sind für eine Anfangsbelegung erreichbar?
- Ist für bestimmte Transitionen ausgeschlossen, dass eine Belegung erreicht wird, in der diese Transitionen gleichzeitig transitionsbereit sind (*gegenseitiger Ausschluss*)?
- Kann eine Belegung erreicht werden, für die keine Transition mehr transitionsbereit ist (*deadlock*)?
- Existieren unendliche Abläufe, in denen eine Transition niemals auftritt (*starvation*)?
- Kann eine Belegung erreicht werden, so dass ausgehend von dieser eine bestimmte Belegung nicht mehr erreichbar ist (*Livelock*)?

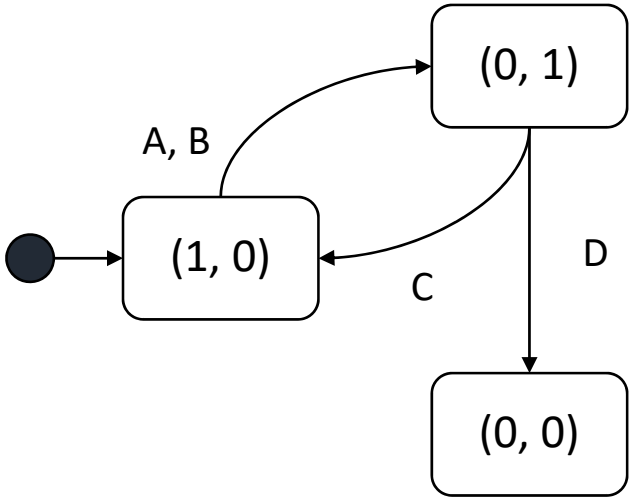
Beispiel



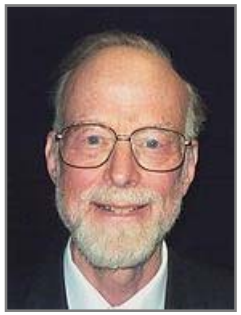
Anfangsbelegung: (1, 0)

Menge der erreichbaren Belegungen:
{(1, 0),
(0, 1),
(0, 0)}

Übergangsdiagramm für die erreichbaren Belegungen



5.3.4 Agenten



Tony Hoare

- Beschreibung von Prozessen durch programmiersprachliche Terme
- Wir sprechen von einer **Prozessalgebra**
- Die folgende Notation lehnt sich an die Sprache CSP („Communicating Sequential Processes“) von C.A.R Hoare (1976) an
- Beispiele:
  - a **or** b
  - a || b

BNF-Syntax der Sprache der Agenten

<agent> ::= skip |
<action> |
<agent> ; <agent> |
<agent> or <agent> |
<agent> || <agent> |
<agent_id> |
<agent_id> :: <agent> |
(<agent>)

<action> vorgegebene Sprache zur Beschreibung von Aktionen

<agent_id> vorgegeben Menge von Identifikatoren

Semantik von Agenten

Wie Petrinetze beschreiben Agenten Mengen von Prozessen (Aktionsstrukturen)

Term	Abläufe
skip	leerer Prozess
a	Prozess mit einem Ereignis, das mit a markiert ist
t_1; t_2	sequentielle Komposition der mit den Agenten t_1 und t_2 beschriebenen Prozesse
t_1 **or** t_2	Vereinigungsmenge der mit den Agenten t_1 und t_2 beschriebenen Prozesse (*)
t_1 || t_2	parallele Komposition der durch die Agenten t_1 und t_2 beschriebenen Prozesse
x :: t (x kommt in t vor)	entspricht dem „Aufruf“ des Agenten x, der durch die rekursive Deklaration x = t charakterisiert ist.

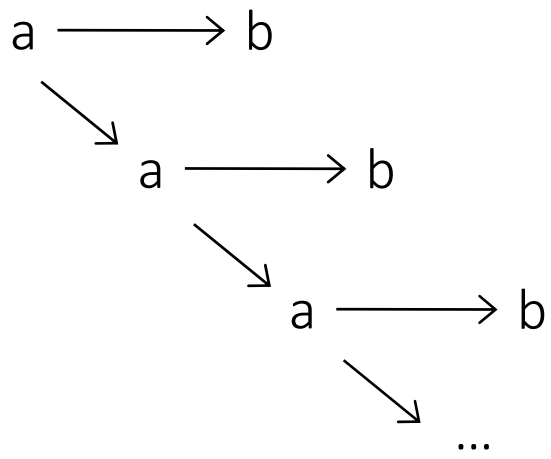
(*) d.h. entweder t_1 oder t_2 wird ausgeführt, aber nicht beide

Rekursiv definierte Agenten

Beispiele:

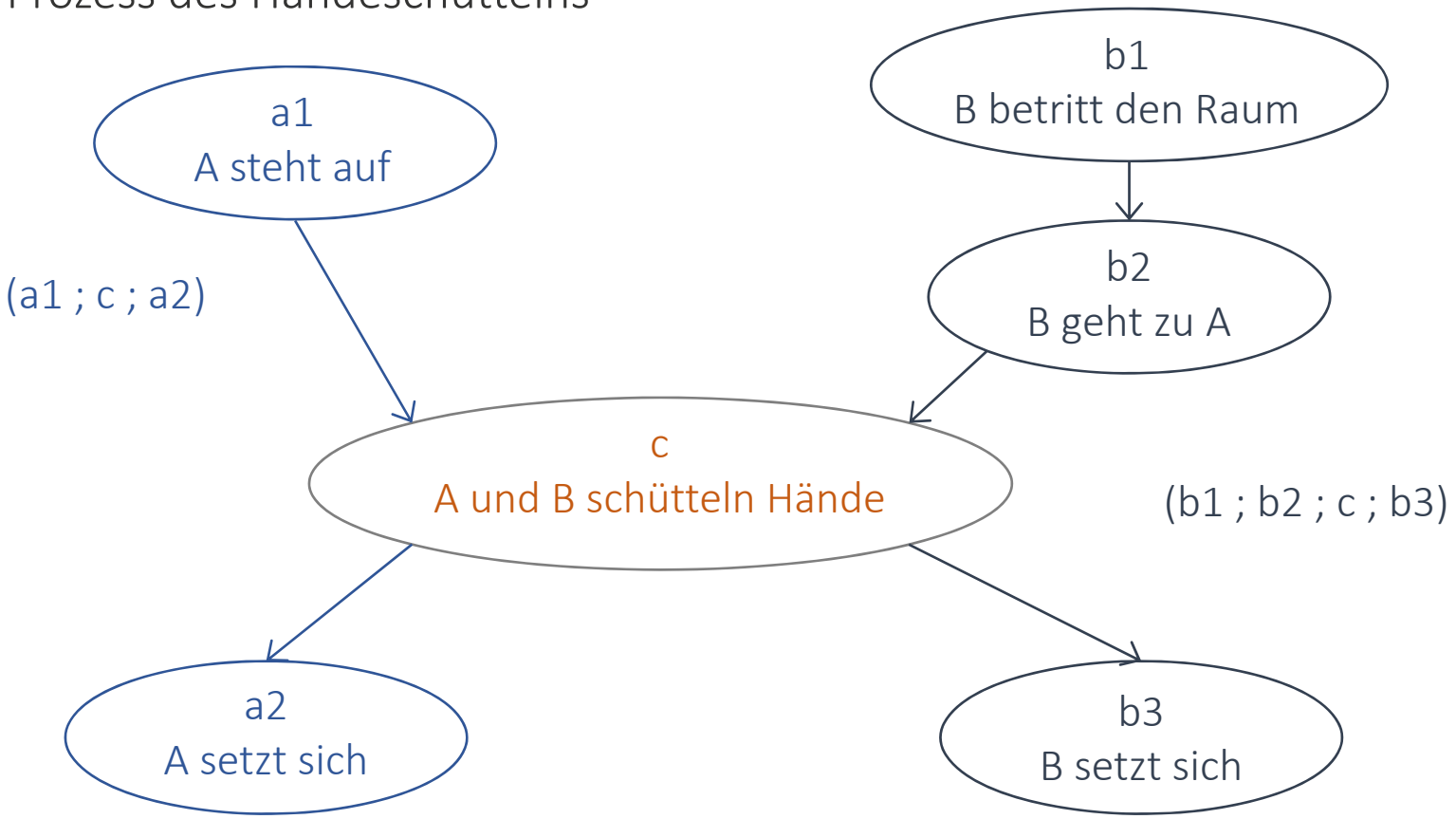
1. ampel :: (rot; gelb; grün; ampel)
2. x :: a; (b || x)

Agent (2) beschreibt den unendlichen Ablauf



Synchronisation und Koordination von Agenten

Prozess des Händeschüttelns



Zwei Prozesse führen eine Menge von Aktionen synchron aus.

Synchrone parallele Komposition

Sei S eine Menge von Aktionen

$t_1 \,||_S\, t_2$ beschreibt die parallele Komposition der Prozesse t_1 und t_2, bei der die Aktionen in S synchron ausgeführt werden.

- Beispiele:
  - Händeschütteln
    $(a1 ; c ; a2) \,||_{\{c\}}\, (b1 ; b2 ; c ; b3)$
  - Prozess mit Verklemmung
    $a ; b \,||_{\{a,b\}}\, b ; a$

Modellierung von gegenseitigem Ausschluss

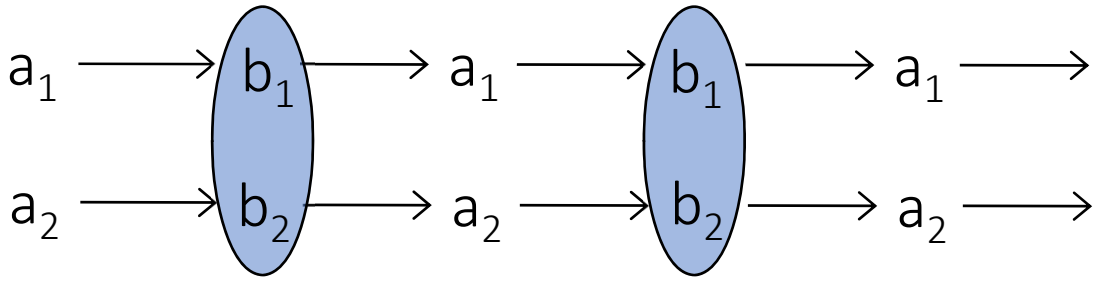
x1 :: a1 ; b1 ; x1 || x2 :: a2 ; b2 ; x2

Bedingung:

b1 und b2 sollen unter gegenseitigem Ausschluss ausgeführt werden

Möglicher Ablauf

Paare von parallelen Ereignissen
mit konfliktträchtigen Aktionen



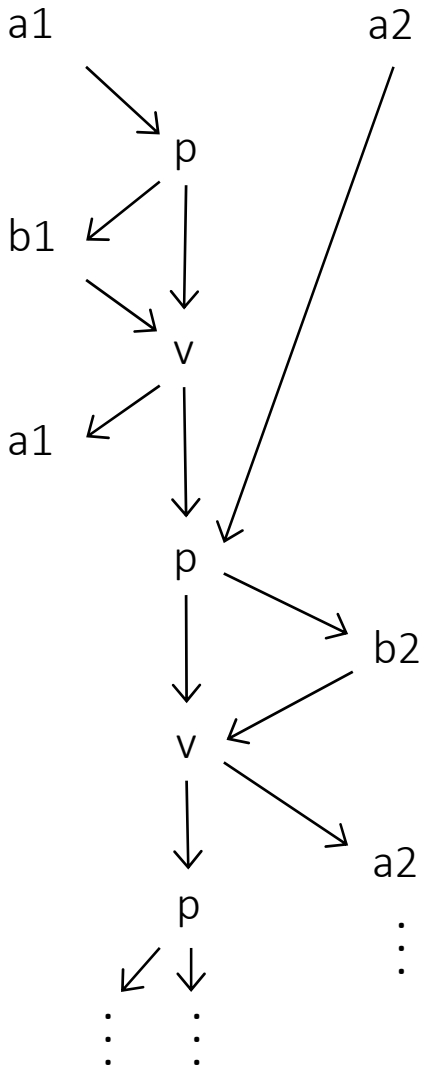
Lösung

k = y :: p ; v ; y

t1 = x1 :: a1 ; p ; b1 ; v ; x2
t2 = x2 :: a2 ; p ; b2 ; v ; x2

t = (t1 || t2) || $_{\{v,p\}}$ k

Möglicher Ablauf



Zusammenfassung Kapitel 5.3

- Aktionsstrukturen bzw. Prozesse
  - mathematische Theorie
  - dient zur **Semantikbeschreibung** von verteilten Systemen
  - Ein verteiltes System wird beschrieben als Menge von Abläufen
- Petrinetze
  - klassische graphische Spezifikationstechnik zur Modellierung von Abläufen
  - über den Ablaufbegriff mit den Aktionsstrukturen verbunden
- Agenten
  - Spezifikation von Prozessen durch Terme
  - über den Ablaufbegriff mit den Aktionsstrukturen verbunden

Verbindung von Programm und Spezifikation

- Programm (z.B. auf Basis von Semaphoren) ist korrekt bzgl. Spezifikation wenn jeder Ablauf des Programms auch Ablauf der Spezifikation ist
- Zwei Spezifikationen (z.B. Petrinetz/Agent) sind äquivalent, wenn sie die gleiche Menge an Abläufen beschreiben