F4 – Parallelität und Nebenläufigkeit

Prüfungsunterlagen zum Vorlesungszyklus:
„Formale Grundlagen der Informatik"
gehalten von Prof. Dr. Rüdiger Valk
Sommersemester 2001

Inhaltsverzeichnis

Kapitel 1

Einleitung

Wie der ganze Zyklus "Formale Grundlagen der Informatik", beschäftigt sich diese Vorlesung auf mathematischer Basis mit Abstraktionen, Modellbildungen und Verfahren zur Beschreibung und Analyse von Algorithmen und Prozessen. Formale Methoden spielen in der Informatik die Rolle eines "Denkzeugs", mit dem der (abstrakte) Kern einer Sache knapp und präzise beschrieben werden kann. Erst auf der Basis eines sauberen theoretischen Fundaments wird es möglich, solche Beschreibungen zu formulieren und deren Analysen vorzunehmen. Parallele und nebenläufige Programme und Prozesse sind wegen ihrer Komplexität besonders anfällig für fehlerhafte Behandlung aufgrund unpräziser Methoden. Es ist daher kein Zufall, dass "*formal methods*" in diesem Gebiet fester Bestandteil der Forschung und Entwicklung sind. Die Vorlesung führt in die wichtigsten Phänomene und Beschreibungstechniken ein. Sie werden z.T. an realen Programmen illustriert.

Zunächst werden parallele Programme und Nebenläufigkeit durch Prozessausdrücke eingeführt. Diese Darstellung ist wie die Prozess-Algebra von R. Milner [Mil99] stark auf Aktionen zentriert. Die Bedeutung der durch Grammatiken definierten Prozessausdrücke wird durch Transitionssysteme (endliche Automaten) als operationale Semantik dargestellt. Der Bezug zur Programmierung wird durch Java-Threads hergestellt. Das Kapitel ist sehr an [MK99] orientiert.

Im 3. Kapitel werden Petrinetze auf drei Abstraktionsebenen eingeführt: als elementare Netze, als kantenkonstante bzw. Platz/Transitionsnetze und als gefärbte Netze. Im Gegensatz zur prozessoralen Sprache des vorigen Kapitels weisen Netze eine Ereignis-

und Zustands-Orientierung auf. Sie sind über die operationale Semantik der Erreichbarkeitsgraphen mit ihr verbunden. Zur Vertiefung kann [Rei85], [JV87] und [GV01] herangezogen werden.

Das 4. Kapitel behandelt elementare Eigenschaften verteilter Systemmodelle wie Invarianten, Lebendigkeit, Fairness und Beschränktheit. Lebendigkeit und Beschränktheit werden am Thema der Analyse von Geschäftsprozessen (workflow) exemplarisch eingesetzt. An das schon im 2. Kapitel am Beispiel behandelte Probleme der Dateninkonsistenz durch nebenläufige Prozesse wird systematisch afgegriffen. Mittels schematischer Auftragsysteme werden die Eigenschaften der Funktionalität und maximalen Nebenläufugkeit diskutiert.

Das 5. Kapitel beginnt mit Unteilbarkeit (Atomizität) als wichtigem Begriff bei nebenläufigen Prozessen. Grundformen der Synchronisation, wie Speicher- Rendezvous- und Nachrichten-Synchronisation werden behandelt. Das Kapitel enthält auch einige klassische Konzepte und hochsprachliche Synchronisations-Konstrukte.

Das 6. Kapitel führt in parallele Algorithmen ein, d.h. in zeitgetaktete Parallelverarbeitung, wie sie auf der PRAM implementiert werden könnte, ohne diese jedoch formal einzuführen. Einen Überblick erhält man durch [Rei93]. Im 7. Kapitel werden dagegen verteilte Algorithmen ([Rei98], [Lyn96],[Mat89]) behandelt, die auf dem Prinzip von kooperierenden asynchronen Algorithmen beruhen.

Kapitel 2

Eine prozessorientierte Sprache und ihre Semantik

2.1 Prozesse als Ordnungen

Allgmein werden Prozesse als Folgen von Handlungen angesehen, wobei hier zunächst Handlungen (Aktionen) gemeint sind, die von einer Maschine, einem Prozessor, allgemein von einer Funktionseinheit ausgeführt werden.

Eigenschaften der Handlungen:

- *extensional*, d.h. durch ihre Wirkung beschreibbar
- *unteilbar* (auch atomar), d.h. sie werden vom Prozessor ununterbrochen ausgeführt
- *geordnet*
  a) durch eine totale Ordnung: sequentieller Prozess
  b) partielle Ordnung: nichtsequentieller Prozess, d.h. zeitlich/kausal unabhängige Handlungen sind möglich

Mehrere sequentielle Prozesse wirken durch *Synchronisation* zusammen und bilden so einen Gesamtprozess, der eine Menge partiell geordneter Handlungen darstellt. Kausal unabhängige Handlungen heißen *nebenläufig*.

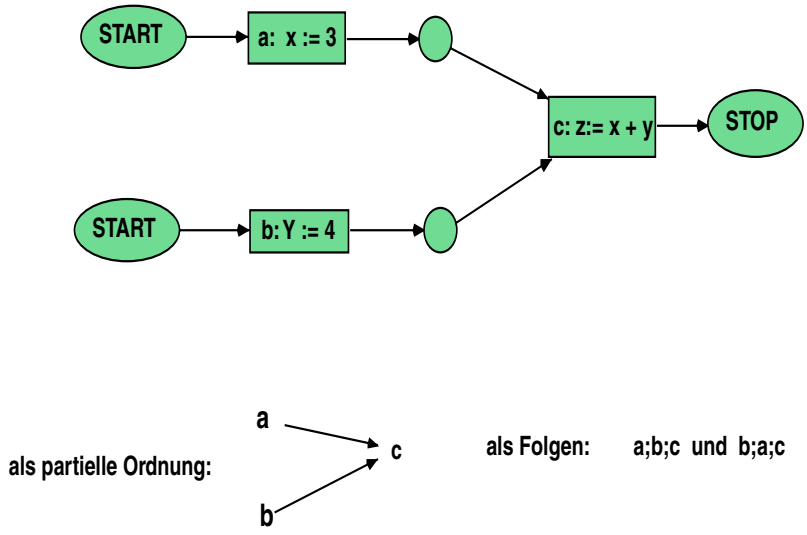


Abbildung 2.1: Nebenläufige Handlungen

Nebenläufige Prozesse werden auf (mindestens) zwei Weisen dargestellt:

a) als partielle Ordnung, wie z.B. in Abb. 2.1 als Relation {(a,c),(b,c)} oder

b) als lineare Ordnung in Form von Folgen : u:= a;b;c und v:= b;a;c.

Die Darstellungsform a) heißt *"partial order semantics"* oder *"PO-Semantik"*, während b) *"interleaving semantics"* oder *"Folgen-Semantik"* heißt.

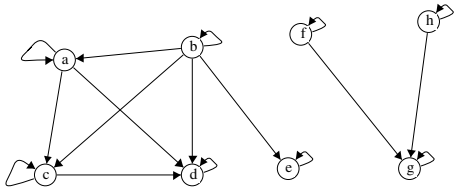
Definition 2.1 *Sei A eine Menge und $R \subseteq A \times A$ eine (binäre) Relation.*

a) (A, R) *heißt partielle Ordnung (partially ordered set, poset), falls gilt:*

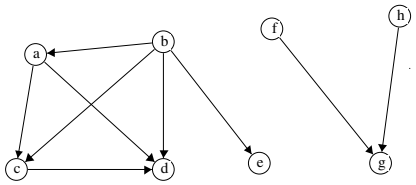
1. $\forall a, b \in A.\ (a, a) \in R$ "Reflexivität"
2. $\forall a, b \in A.\ (a, b) \in R \wedge (b, a) \in R \Rightarrow a = b$ "Antisymmetrie"
3. $\forall a, b, c \in A.\ (a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$ "Transitivität"

Schreibweise: $a \leq b$ *für* $(a, b) \in R$

b) (A, R) *heißt strikte Ordnung oder Striktordnung (partially ordered set, poset, Halbordnung), falls gilt:*



R = {(a,a),(a,c),(b,a),...}



S = R - id = {(b,a),(a,c),(b,c),...}

Abbildung 2.2: Partielle Ordnung R und Striktordnung S

1. $\forall a, b \in A.\ (a, a) \notin R$ “Irreflexivität”
2. $\forall a, b, c \in A.(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R.$ “Transitivität”

Schreibweise: $a < b$ *für* $(a, b) \in R$

c) (A, R) *heißt totale oder lineare Ordnung (totally ordered set, poset), falls gilt:*

1. (A, R) *ist partielle Ordnung*
2. $\forall a, b \in A.\ (a, b) \in R \vee (b, a) \in R$ “Vollständigkeit”

Eine Striktordnung mit 2. heißt totale oder lineare Striktordnung.

In Abb. 2.2 ist oben eine partielle Ordnung R und darunter die Striktordnug $S = R - id$ dargestellt. Striktordnungen lassen sich oft übersichtlicher als Präzedenzgraph (“Hasse-Diagramm”) darstellen, der nur die direkten Nachfolger enthält (Abb. 2.3).

Definition 2.2 *Sei* $(A, <)$ *eine strikte Ordnung und* $a, b \in A$.

1. b *heißt* direkter Nachfolger *von* a *(in Zeichen:* $a \lessdot b$*), falls:*

   $a \lessdot b :\Leftrightarrow a < b \wedge \neg\exists c \in A.\ a < c \wedge c < b$

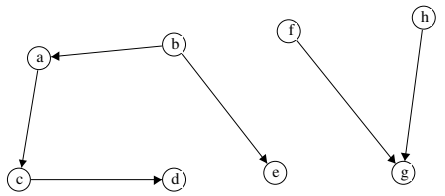


Abbildung 2.3: Präzedenzgraph Q

2. $(A, \lessdot)$ *heißt Präzedenzrelation zu* $(A, <)$.

Anmerkung:

a) $\lessdot = < - <^2$

("$-$" Mengendifferenz, $<^2 := < \circ < = \{(a,b) \mid \exists c.\, a < c \wedge c < b\}$ Relationenprodukt)

b) Gilt $< = \lessdot^+$(transitive Hülle), dann heißt $(A, <)$ kombinatorisch. In diesem Fall ist $<$ durch $\lessdot$ festgelegt. Für endliche Mengen A ist $(A, <)$ immer kombinatorisch.

c) Falls A endlich ist, muß dies nicht gelten

Beispiel: rationale Zahlen $(\mathbb{Q}, <)$, hier ist $\lessdot = \emptyset$

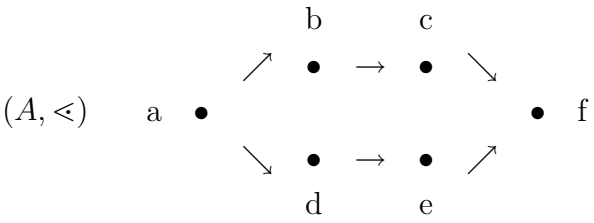
d) Ist eine Striktordnung $(A, <)$ isomorph zu einer Teilmenge von $\mathbb{N}$ (mit der von $\mathbb{N}$ geerbten Striktordnung), dann wird sie gerne mit einer Folge beschrieben:

$$\begin{array}{rcl} a_1 \to a_2 \to \ldots \to a_n & \text{mit} & a_1 a_2 \ldots a_n \quad (card(A) = n) \\ & \text{oder} & a_1; a_2; \ldots; a_n \\ a_1 \to a_2 \to \ldots & \text{mit} & a_1 a_2 \ldots \quad (card(A) = card(\mathbb{N})) \\ & \text{oder} & a_1; a_2; \ldots \end{array}$$

e) Für eine partielle Striktordnung $(A, <)$ ist
$Lin(A, <) := \{(A, <_1) | (A, <_1)$ ist eine lineare Striktordnung und $< \subseteq <_1\}$
die Menge der linearen (oder seriellen) Vervollständigung von $(A, <)$. Ist $(A, <)$ kombinatorisch mit $< = \lessdot^+$ dann wird: $Lin(A, \lessdot) := Lin(A, \lessdot^+)$ definiert.

f) Übertragung auf Folgen entsprechend d):

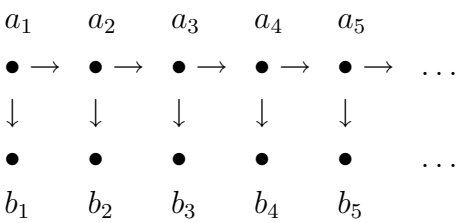
Beispiel:



$$Lin(A,\leq) = \{\ a\ b\ c\ d\ e\ f\ ,\quad a\ b\ d\ c\ e\ f\ ,\quad a\ d\ b\ c\ e\ f\ ,\quad a\ d\ b\ e\ c\ f\ ,\quad a\ d\ e\ b\ c\ f\ ,\quad a\ b\ d\ e\ c\ f\ \}$$

Konstruktionsprinzip?

Beispiel: $(A, \leq)$



$$Lin(A,\leq) = \{\ a_1\ b_1\ a_2\ b_2\ a_3\ b_3\ a_4\ b_4\ a_5\ b_5\ \ldots \quad a_1\ a_2\ b_1\ b_2\ a_3\ b_4\ a_4\ b_4\ a_5\ b_6\ \ldots \quad a_1\ a_2\ b_1\ a_3\ b_2\ b_4\ a_4\ b_4\ a_5\ b_6\ \ldots \quad \vdots$$

Konstruktionsprinzip?
(so nicht möglich, da $Lin(A, \leq)$ überabzählbar)

Prozesse mit

- PO-Semantik:
  - Darstellung: partielle Ordnung
    traces
    Kausalnetze
  - erzeugt durch System: Petrinetze

- Folgen-Semantik:

Darstellung:	Folgen (endl.,unendlich)
	temporale Logik
erzeugt durch System:	Automaten
	Turing-Maschinen
	RAM, PRAN
	Prozessalgebra

in diesem Kapitel: Folgen-Semantik

2.2 Die elementare Prozesssprache elP

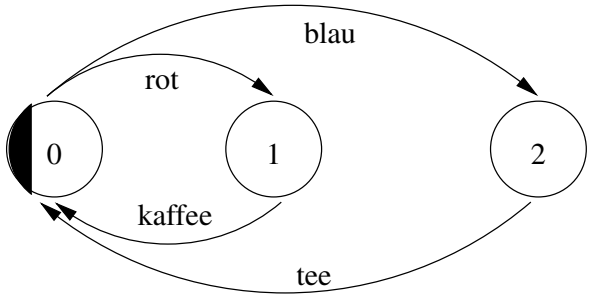
Im folgenden Kasten wird gegenübergestellt:
erste Zeile: Implementationsebene zweite Zeile: Modellebene
Übergang von oben nach unten : *Abstraktion*

Prozessor (I)	$\leftrightarrow$	Programm (II)	$\leftrightarrow$	Prozess (III)
$\updownarrow$		$\updownarrow$		$\updownarrow$
Automat, Transitionssystem	$\leftrightarrow$	PA-Ausdruck	$\leftrightarrow$	Ausführungsfolgen

Beispiel: Getränkeautomat

GETRÄNKE= (rot $\rightarrow$ kaffee $\rightarrow$ GETRÄNKE
|blau $\rightarrow$ tee $\rightarrow$ GETRÄNKE)

PA-Ausdruck (II)



Automat, Transisitionssystem (I)

rot; kaffee; rot; kaffee; blau; tee; blau; tee; blau; tee;... eine Ausführungsfolge (III)

Andere Sicht: **Semantik von Programmiersprachen**

Semantik eines Programms → Sinn, Bedeutung (hier: in einer formalen Darstellung)

Es gibt (mindestens) zwei Formen:

a) operationale Semantik
Programm (z.B. PA-Ausdruck) → operationale Semantik[1] (z.B. Automat,TS-System)

b) denotationale Semantik
Programm (z.B. PA-Ausdruck) → denotationale Semantik[2] (z.B. Funktion f)

Strukturmerkmale:

zu a): Struktur der Maschine, des Automaten

zu b): Struktur der Sprache: Syntax

Definition 2.3 *Elementare PA-Ausdrücke werden durch folgende kontextfreie Grammatik (BNF-Form-artig) definiert:*

BProcessDef:

BProcessBody AlphabetExtension$_{opt}$ *Relabel*$_{opt}$ *Hiding*$_{opt}$.

BProcessBody:

ProcessIdent = BLocalProcess

BProcessBody , Processident = BLocalProcess

BLocalProcess:

STOP

ERROR

ProcessIdent

(BChoice)

[1]Ein-/Ausgabe-Relation, z.B. alle Paare (z.B. Eingabefolge,Ausgabefolge)
[2]Ein-/Ausgabe-Funktion, z.B. alle Paare (Eingabefolge,$\underbrace{f(Eingabefolge)}_{Ausgabefolge}$)

BChoice:

BActionPrefix

BChoice | *BActionPrefix*

BActionPrefix:

BPrefixActions → *BLocalProcess*

BPrefixActions:

ActionLabel

BPrefixActions → *ActionLabel*

BCompositeDef:

|| *ProcessIdent* = *BCompositeBody* $Priority_{opt}$ $Hiding_{opt}$

BCompositeBody:

ProcessIdent $Relabel_{opt}$

(BParallelComposition) $Relabel_{opt}$

BParallelComposition:

BCompositeBody

BParallelComposition || *BCompositeBody*

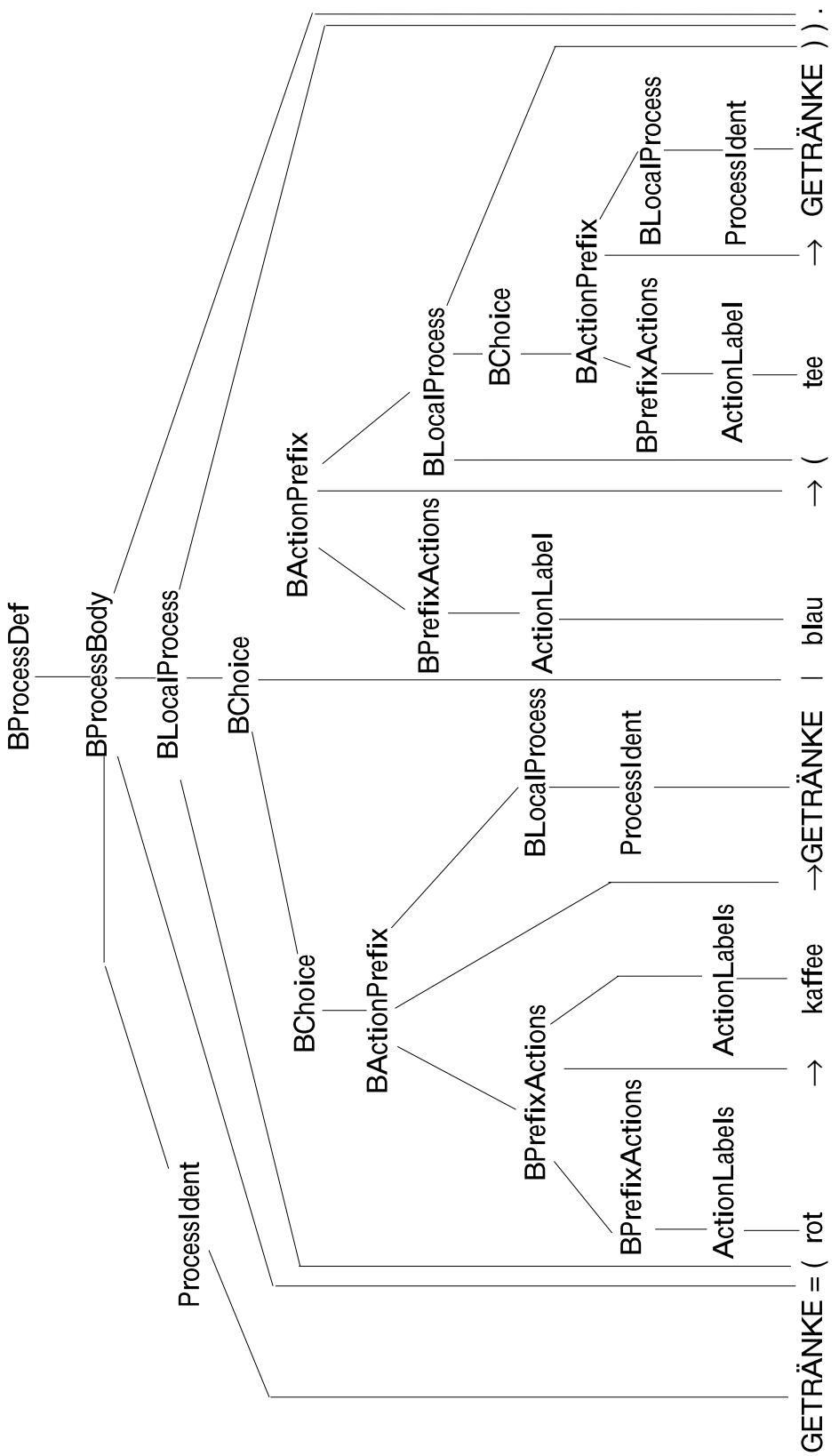
zum Vergleich: BN-Notation (Skript F1, Kap.3, 1.Seite)

```
<BComposite Body> ::= <ProcessIdent> |
                      <ProcessIdent><Relabel>|
                      (<ParallelCompositium>)|
                      (<ParallelCompositium>)Relabel
```

Achtung: Das Zeichen | gehört oben (in der Definition 2.3) zu den Terminalen, unten (in der BN-Notation) jedoch nicht. xyz_{opt} steht für "xyz kann (optional) gewählt werden oder nicht".

operationale Semantik durch nichtdeterministische endliche Automaten (NFA, Skript F1, Def.2.1)

hier: Transisitionssystem



F4/2001

Abbildung 2.4: Beispiel eines Ableitungsbaum

Definition 2.4 *Transitionssystem*

$TS = (Z, X, K, z, z_{stop})$ *also* $Z_{start} := \{z\} \subseteq Z, Z_{end} := \{z_{stop}\} \subseteq Z$
Wenn nicht benötigt, wird z_{stop} *weggelassen. Zuweilen wird auch ein besonderer Zustand* $z_{error} \in Z$ *benutzt.*

Z: Zustandsmenge		*(nicht notw. endlich)*
X: Alphabet		*Bezeichner der Aktionen*
$K \subseteq Z \times \tilde{X} \times Z$		*Transitionsrelation*
$\tilde{X} = X \cup \{\lambda\}$	λ interne *Aktion (oft auch* τ*)*	

Notation für Zustandsübergang für gegebenes TS und $a \in \tilde{X}$:

TS $\xrightarrow{\text{a}}$ TS' *gdw.*

1) $TS = (Z, X, K, z)$ mit $z \in Z$ und $TS' = (Z, X, K, z')$ mit $z' \in Z$

2) $(z, a, z') \in K$

3) $z \neq z_{error}$

Beispiel: Z=$\{0, 1, 2\}$
X=$\{\text{rot}, \text{kaffee}, \text{blau}, \text{tee}\}$
z=0
K=$\{(0, \text{rot}, 1), (0, \text{blau}, 2), (1, \text{kaffee}, 0), (2, \text{tee}, 0)\}$

Die Konstrukte von PA-Ausdrücke:

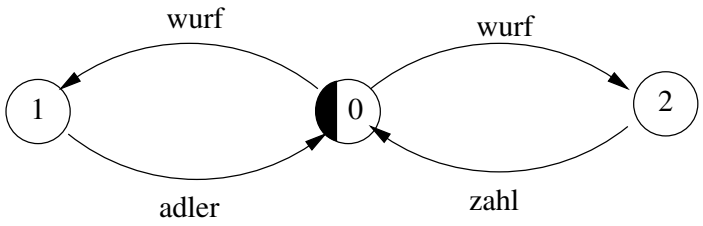
1. Aktions-Präfix:
$(X \rightarrow P)$
verhält sich nach Aktion x wie P.
Iteration der Regel als $(x \rightarrow (y \rightarrow P))$ oder $(x \rightarrow y \rightarrow P)$ (siehe Ableitungsbaum S.11)

2. Stop-Prozess:
STOP
ist ein vordefinierter Prozess ohne Aktion, der die reguläre Termination darstellt.

3. Error-Prozess:
ERROR
ist ein vordefinierter Prozess ohne Aktion, der die irreguläre Termination darstellt.

4. Auswahl (choice):
$(x \to P | y \to Q)$
verhält sich nach Aktion x wie P oder alternativ nach y wie Q.
Hierbei gibt es zwei mögliche Fälle
a) $x \neq y$: deterministische Auswahl und b) $x = y$: nichtdeterministische Auswahl

Beispiel: $LOSEN = (wurf \to adler \to LOSEN | wurf \to zahl \to LOSEN)$



allgemeiner: $(x_1 \to P_1 | x_2 \to P_2 | \ldots | x_n \to P_n)$
falls $P_1 = P_2 = \ldots = P_n = P$ auch: $(\{x_1, \ldots, x_n\} \to P)$

5. Nebenläufige Prozesse:
$(P \| Q)$
verhält sich wie, wenn P und Q unabhängig (nebenläufig, parallel) ablaufen.
Gemeinsame Aktionsbezeichner $x \in X$ werden als eine Aktion behandelt. (interne Aktionen sind natürlich nicht gemeinsam)

Beispiel:
FELIX=(schlafen → tennis → essen → FELIX)
MARIETTA=(tennis → aufgaben → klavier → MARIETTA)
‖SPIELER=(FELIX‖MARIETTA)
KLUB=(platzpflege → tennis → fest → KLUB)
‖SAMSTAG=(SPIELER‖KLUB)

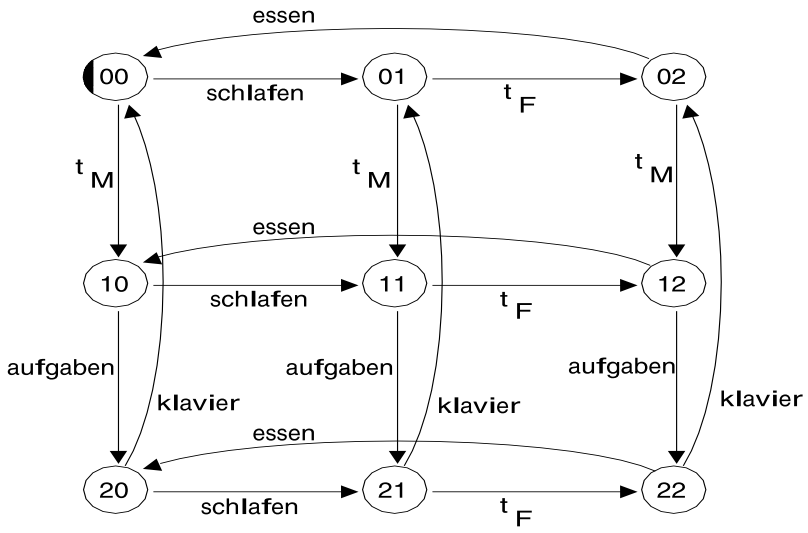
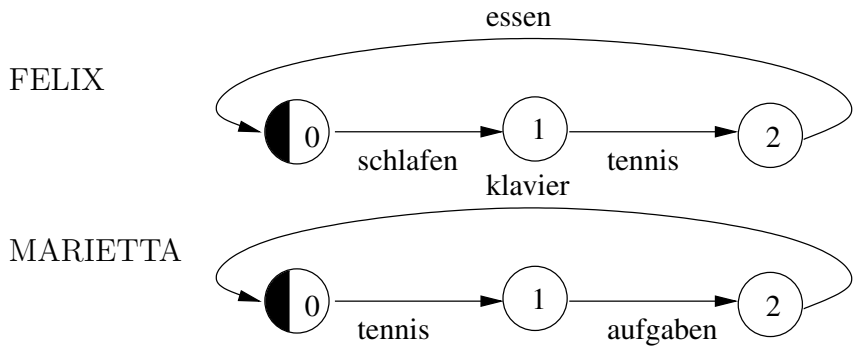


Abbildung 2.5: ‖SPIELER (wenn sie alleine Tennis spielen: Felix t_F Mariette t_M)



Aufgabe: Zeichnen sie das Transitionssystem 2.6 noch "regelmäßiger"!
Aber: i.A. bringt das wenig! → Methoden der Analyse!

Wenn die Prozeßfolgen des PA-Ausdrucks P_1 bzw P_2 durch TS_1 bzw TS_2 dargestellt werden, dann beschreibt das "Produkt-Transitions-System" $TS_1 \times TS_2$ die Folgen von $(P_1 \| P_2)$

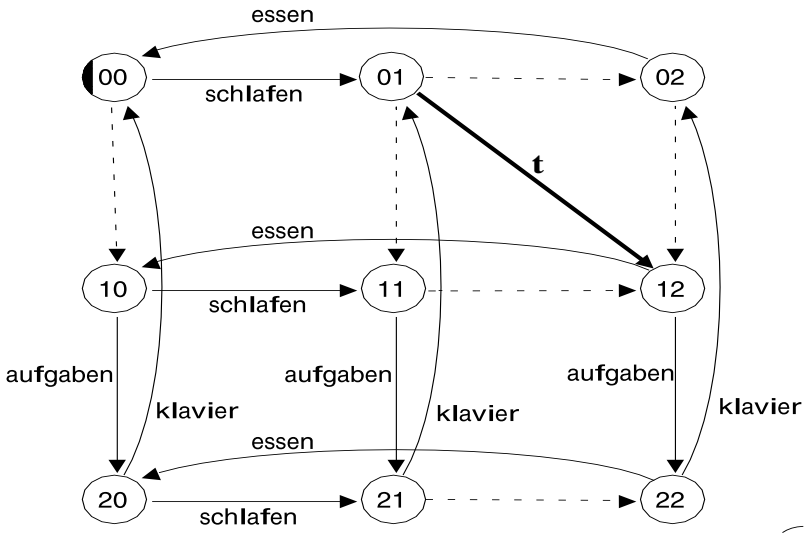


Abbildung 2.6: ||SPIELER (wenn sie zusammen Tennis spielen: Felix: t statt t_F, Marietta: t statt t_M)

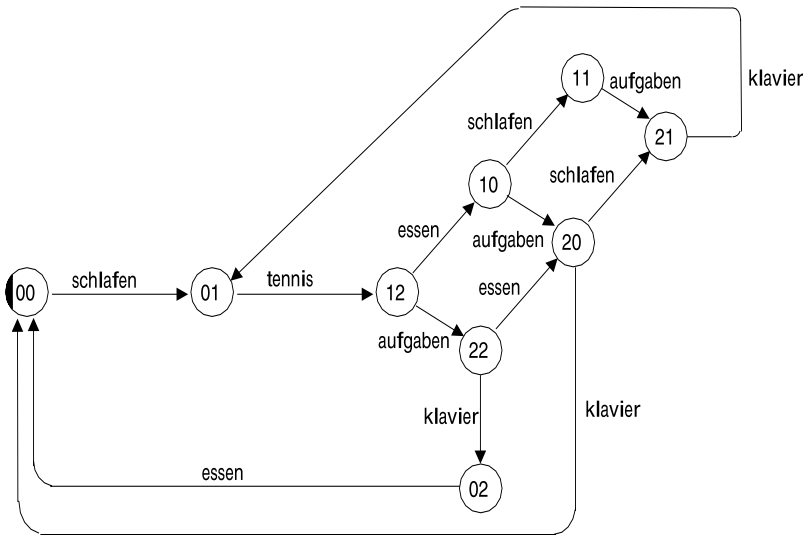


Abbildung 2.7: Wie Bild 2.6 nur neu gezeichnet

Definition 2.5 *Produkt-Transitions-System $TS_1 \times TS_2$ (vergl. Skript F1 "Produktautomat" - Theorem 2.14 und 2.18)*

$$
\begin{array}{rrll}
\textit{Geg:} & TS_1 & = & (Z_1, X_1, K_1, z_0^1, z_{stop}^1) \qquad TS_2 = (Z_2, X_2, K_2, z_0^2, z_{stop}^2) \\
& TS_1 \times TS_2 & := & (Z_1 \times Z_2, X_1 \cup X_2, K, (z_0^1, z_0^2), (z_{stop}^1, z_{stop}^2)) \\
& K & := & \{((z_1, z_2), x, (z_1', z_2'))| \\
& & \textit{wobei} & (z_1, x, z_1') \in K_1 \wedge z_2' = z_2 \textit{ falls } x \in \tilde{X}_1 - X_2 \\
& & \textit{und} & (z_2, x, z_2') \in K_2 \wedge z_1' = z_1 \textit{ falls } x \in \tilde{X}_2 - X_1 \\
& & \textit{und} & (z_1, x, z_1') \in K_1 \wedge (z_2, x, z_2') \in K_2 \textit{ falls } x \in X_1 \bigcap X_2\}
\end{array}
$$

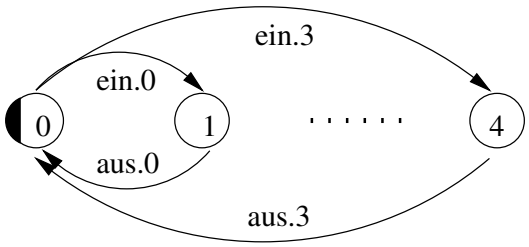
Ansätze zu einer formalen operationalen Semantik von PA-Ausdrücken sind in Abbildung 2.8 angegeben. Dort werden die entsprechenden Transitionssysteme über den syntaktischen Aufbau der PA-Ausdrücke konstruiert.

2.3 Die allgemeine Prozesssprache $\mathcal{P}$

6. indizierte Prozesse und Aktionen:

Puffer= (ein[i : 0 . . . 3] → aus[i] → PUFFER). steht für

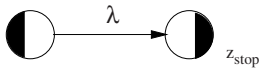
$$
\begin{array}{lllllll}
\text{PUFFER=} & (& \text{ein}[0] & \rightarrow & \text{aus}[0] & \rightarrow & \text{PUFFER} \\
& | & \text{ein}[1] & \rightarrow & \text{aus}[1] & \rightarrow & \text{PUFFER} \\
& | & \text{ein}[2] & \rightarrow & \text{aus}[2] & \rightarrow & \text{PUFFER} \\
& | & \text{ein}[3] & \rightarrow & \text{aus}[3] & \rightarrow & \text{PUFFER}).
\end{array}
$$



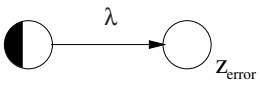
0..3 kann durch T ersetzt werden, falls

```
range T=0..3
```

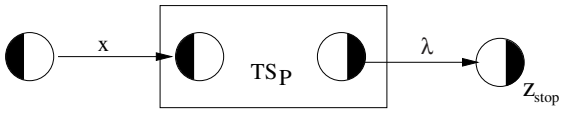
R=STOP



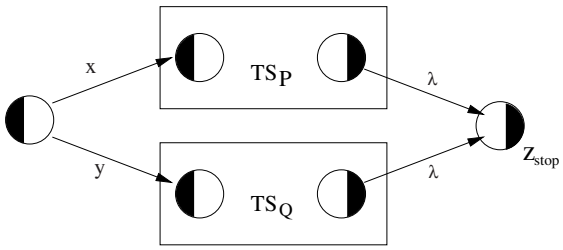
R=ERROR



R=(x→P)



R=(x→P|y→Q)



R=(P‖Q) wie in Definition 2.5

P=(x→P|y→STOP)

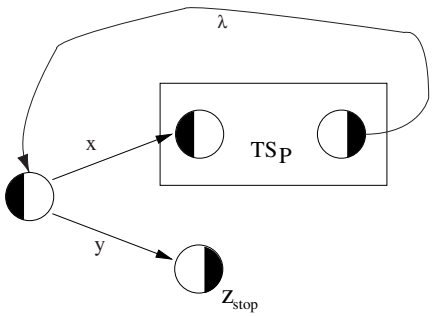


Abbildung 2.8: Formale operationale Semantik von PA-Ausdrücken Q, definiert über deren syntaktischen Aufbau (Bei diesen Konstruktionen verlieren die START- und STOP-Zustände diese Eigenschaft.) Ist jede Rekursion so darstellbar?

vereinbart wurde.

```
const N=1
range T=0..N
range R=0..2*N

SUM=(in[a:T][b:T] -> TOTAL[a+b]),
TOTAL[s:R]=(out[s] -> SUM).
```

7. Prozess-Parameter

PUFFER$(N = 3) = (ein[i : 0 \ldots N] \rightarrow aus[i] \rightarrow$ PUFFER)
oder
const N=3
PUFFER= $(ein[i : 0 \ldots N] \rightarrow aus[i] \rightarrow$ PUFFER).

8. Geschützte Aktion (guarded action/command):
$(when \quad B \quad x \rightarrow P | y \rightarrow Q)$
verhält sich wie Aktion x gefolgt von P, falls B zu wahr ausgewertet wird oder wie Aktion y gefolgt von Q.

$$
\begin{array}{rcl}
\text{COUNTDOWN}(N = 3) & = & (\text{start}\rightarrow\text{COUNTDOWN}[N]), \\
\text{COUNTDOWN}[i : 0 \ldots N)] & = & (when(i > 0)\ \text{tick}\rightarrow\text{COUNTDOWN}[i-1] \\
 & & |when(i == 0)\ \text{beep} \rightarrow \text{STOP} \\
 & & |\ \text{stop} \rightarrow \text{STOP}).
\end{array}
$$

Prozesse in Java: heavyweight process

↓

lightweight process → thread (von "Thread of control": Faden des Programmflusses)

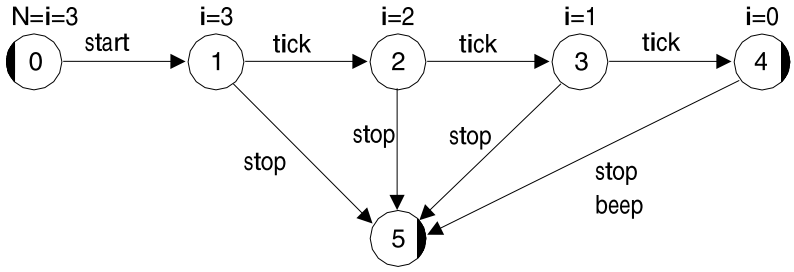


Abbildung 2.9: Transitionssystem

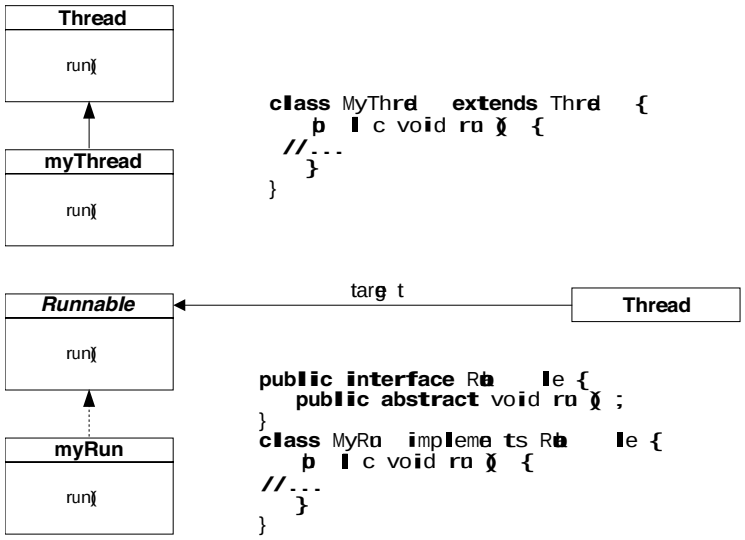


Abbildung 2.10: Darstellung eines Thread

wird erzeugt durch: Thread a =*new* MyThread();
oder Thread b = *new* MyThread(*new* MyRun());
→ Zustand:"created"

start() ruft die run() - Methode aus → Zustand: "running"

stop() terminiert Prozess; der dann nicht mehr zu starten ist → garbage collection falls nicht mehr referenziert

yield() Prozess gibt Prozess frei: Zustand "runnable" kann aber wieder aufgenommen werden durch

dispatch()

```
THREAD       = CREATED,
CREATED      = (start          → RUNNING
                |stop           → TERMINATED),
RUNNING      = ({suspend,sleep}→ NON_RUNNABLE
                |yield          → RUNNABLE
                |{stop,end}     → TERMINATED
                |run            → RUNNING),
RUNNABLE     = (suspend        → NON_RUNNABLE
                |dispatch       → RUNNING
                |stop           → TERMINATED),
NON_RUNNABLE = (resume         → RUNNABLE
                |stop           → TERMINATED),
TERMINATED   = STOP.
```

Abbildung 2.11: Lebenszyklus von Threads als PA-Ausdruck

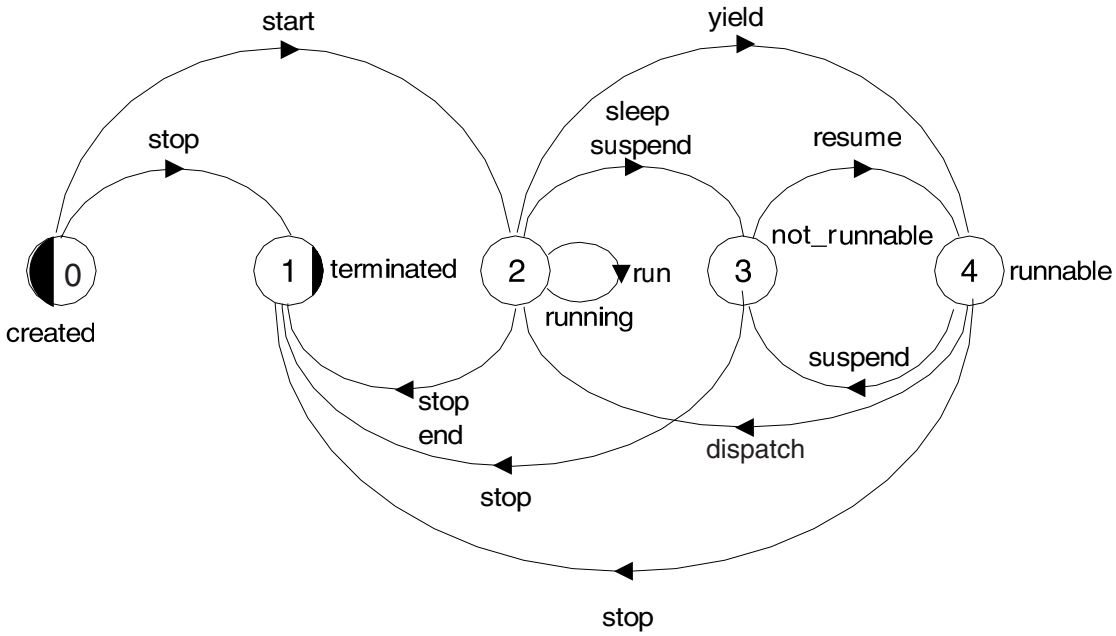


Abbildung 2.12: Lebenszyklus von Threads als Transitions-System

suspend() hält Prozess an, bis er durch

resume() wieder in den Zustand "runnable" gerät, wo er vom Prozessor wieder weiter ausgeführt werden kann

sleep(3) hält Prozess für 3 msec an, dann automatisch: resume().

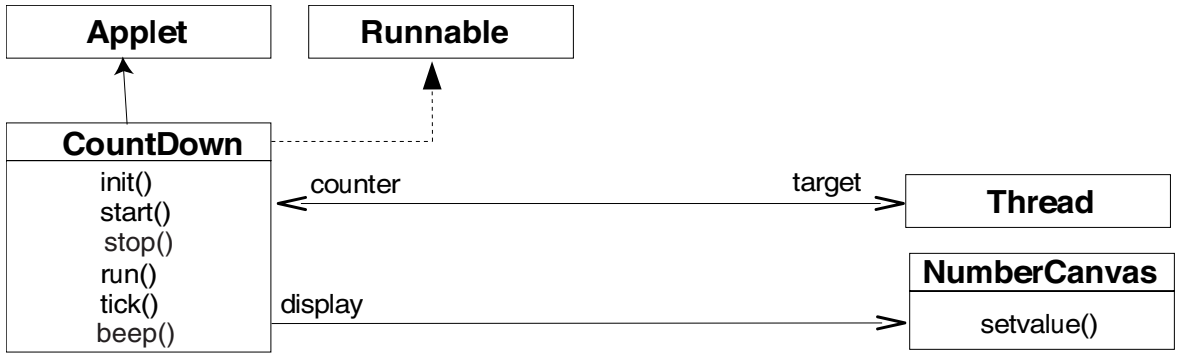


Abbildung 2.13: Klassendiagramm des Counter-Applets

```
public class NumberCanvas extends Canvas {
  // create canvas with title and optionally set background color
  public NumberCanvas(String title) {...}
  public NumberCanvas(String title, Color c) {...}

  // set background color
  public void setcolor(Color c) {...}

  // display newval on screen
  public void setvalue(int newval) {...}
}

public class CountDown extends Applet
                       implements Runnable {
  Thread counter; int i;
  final static int N = 10;
  AudioClip beepSound, tickSound;
  NumberCanvas display;

  public void init() {
    add(display=new NumberCanvas("CountDown"));
    display.resize(150,100);
    tickSound = getAudioClip(getDocumentBase(),"sound/tick.au");
```

```
    beepSound = getAudioClip(getDocumentBase(),"sound/beep.au");
  }

  public void start() {
    counter = new Thread(this);
    i = N; counter.start();   // Browser started, thread ‘counter’ mit ‘run’-Methode
  }

  public void stop() {  // z.B. wenn Nutzer die Site verlaesst - vergl. stop()
    counter = null;
  }
                                              //  COUNTDOWN (N=3) = (start-> COUNTDOWN[N]),
  public void run() {                         //
    while(true) {                             //  COUNTDOWN[i:0..N] =
      if (counter == null) return;            //  (when(i>0) tick-> COUNTDOWN[i-1]
      if (i>0) { tick(); --i;}                //  |when(i==0) beep-> STOP
      if (i==0) { beep(); return;}            //  |stop-> STOP
    }                                         //  ).
  }

  private void tick() {
    display.setvalue(i); tickSound.play();
    try{ Thread.sleep(1000);}
    catch (InterruptedException e) {}
  }

  private void beep() {
    display.setvalue(i); beepSound.play();
  }
}
```

9. Prozess-Etikette

a:P

verhält sich wie P, hat jedoch den Bezeichner a.

Jede Aktion x in P wird zu a.x in a:P.

Beispiel 2.6

$$\begin{aligned} \text{PROZ} &= (\text{belegen} \rightarrow \text{benutzen} \rightarrow \text{freigeben} \rightarrow \text{PROZ}) \\ \\ \text{a:PROZ} &= (\text{a.belegen} \rightarrow \text{a.benutzen} \rightarrow \text{a.freigeben} \rightarrow \text{a:PROZ}) \\ \text{b:PROZ} &= (\text{b.belegen} \rightarrow \text{b.benutzen} \rightarrow \text{b.freigeben} \rightarrow \text{b:PROZ}) \\ \|\text{PROZ\_PAAR} &= (\text{a:PROZ} \parallel \text{b:PROZ}) \end{aligned}$$

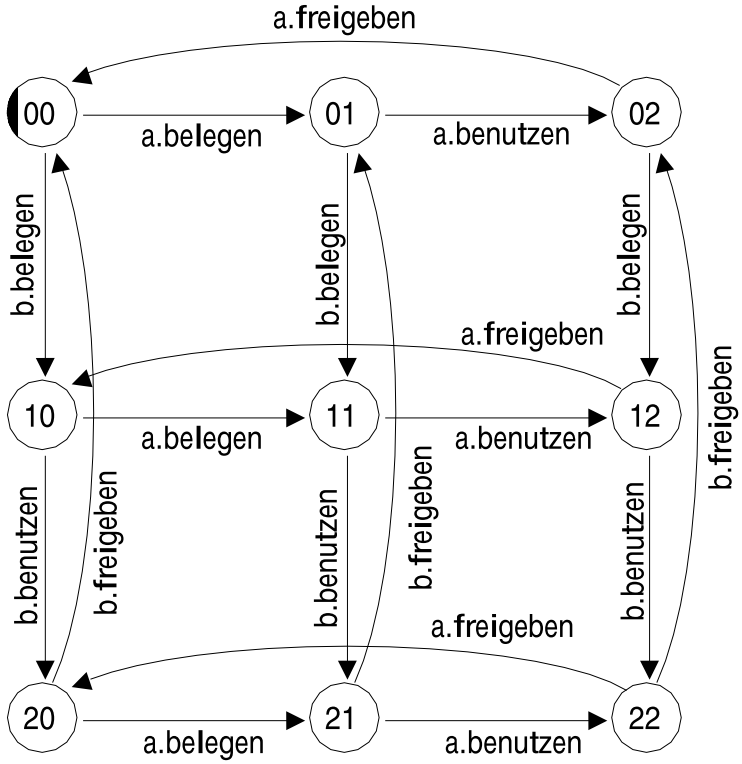


Abbildung 2.14: Transitionssystem für ||PROZ_PAAR

10. Vielfach-Etikette:
$\{a_1, \ldots, a_x\} :: P$
ersetzt jedem Aktionsbezeichner n durch $a_1.n, \ldots, a_x.n$, und außerdem $(n \to Q)$ durch $\{a_1.n, \ldots, a_x.n\} \to Q$ d.h. $(a_1.n \to Q | a_2.n \to Q | \ldots | a_x.n \to Q)$

Beispiel 2.7 Betriebsmittel

BM=(belegen → freigeben → BM)
{a,b}::BM wie
{a.belegen,b.belegen} → ({a.freigeben, b.freigeben}→ BM)
d.h. wie
((a.belegen→(a.freigeben→ BM | b.freigeben → BM)) | (b.belegen → (a.freigeben→ BM | b.freigeben → BM)))

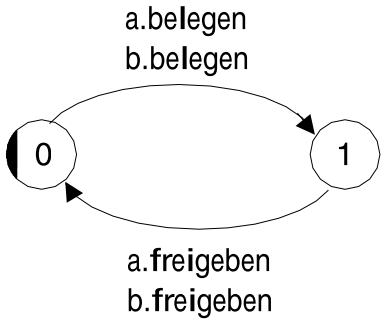


Abbildung 2.15: Transitionssystem von {a,b}::BM

11. Umetikettierung
$P/\ \{\text{neu}_1/\text{alt}_1, \ldots, \text{neu}_n/\text{alt}_n\}$
ist der PA-Ausdruck P, jedoch nach (simultaner) Ersetzung der Etiketten alt_i durch neu_i $(1 \leq i \leq n)$.

12. Verdecken
$P \setminus \{a_1, \ldots, a_x\}$
ist der PA-Ausdruck P, jedoch nach Ersetzung der Etiketten $a_1, \ldots, a_x$ durch die Bezeichner τ (oder λ) für die interne Aktion.
$P@\{a_1, \ldots, a_x\}$
ist der PA-Ausdruck $P \setminus (A - \{a_1, \ldots, a_x\})$, wobei A die Menge der Aktionsbezeichner von P ist, d.h. alle Bezeichner bis auf $\{a_1, \ldots, a_x\}$ werden zu internen Aktionen.

Zusammengesetzt ergibt sich *wechselseitiger Ausschluss* (mutual exclusion):
‖BM_PROZ=(a:PROZ ‖ b:PROZ ‖ {a,b}::BM)

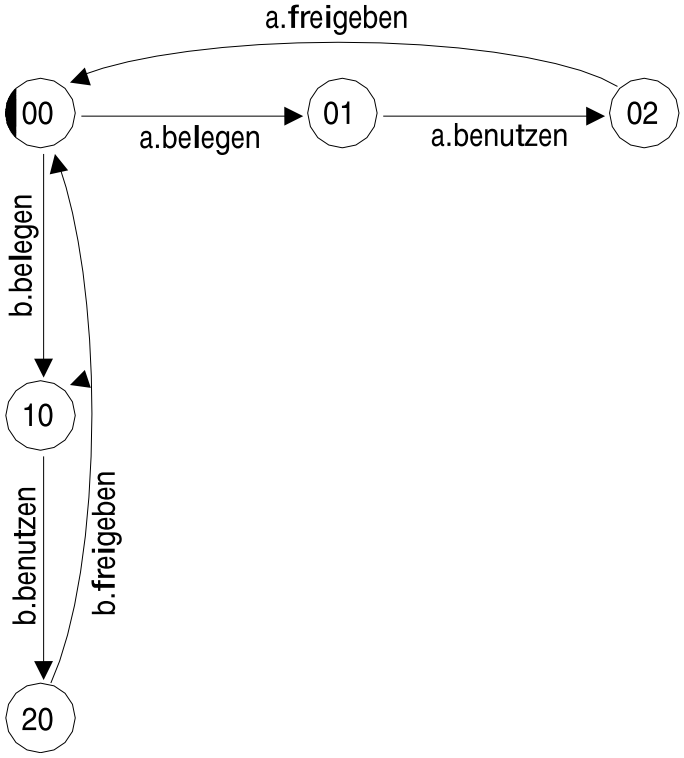
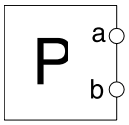


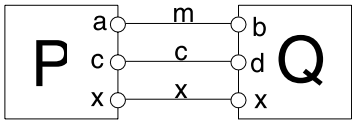
Abbildung 2.16: Transitionssystem von ‖BM_PROZ

Strukturdiagramme:

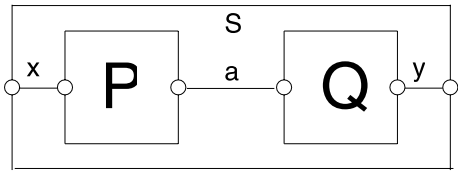
P mit $A = \{a, b\}$



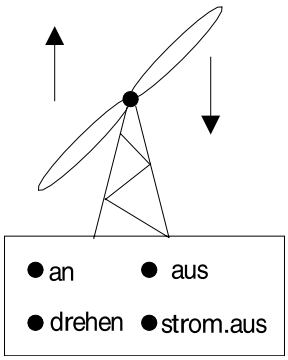
$(P\|Q)/\{m/a, m/b, c/d\}$



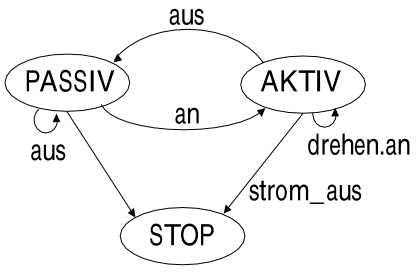
$\|S = (P\|Q)@\{x, y\}$



Beispiel 2.8



RAD = PASSIV
PASSIV = (an → AKTIV
—aus → PASSIV
—strom_aus → STOP),
AKTIV = (aus → PASSIV
—{drehen,an} → AKTIV
—strom_aus → STOP)



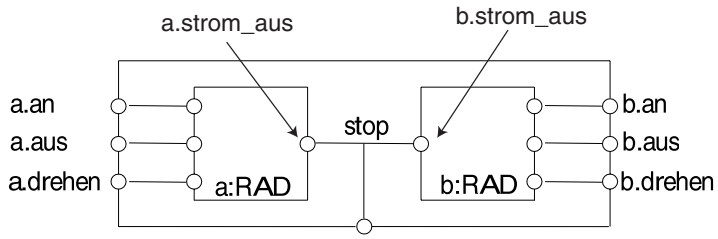


Abbildung 2.17: ‖PAARLAUF=(a:RAD‖b:RAD)/{stop/{a,b}.strom_aus}

Beispiel 2.9 *Louvre* (nach [AG93] [MK99])

Problemstellung: zu einem geschlossenem Bereich (Kunsthalle, Louvre, Ziergarten,...) soll die Personenkontrolle durch ein Programm unterstützt werden. Dazu kann auf Drehkreuze (turnstiles) an verschiedenen Eingängen zugegriffen werden.

Abbildung 2.18: Der zu kontrollierende Bereich

Zur Vereinfachung gebe es nur 2 Eingänge (West,Ost) und nur Zugänge von Personen Auf einem Bildschirm sollen die Anzahlen der Zugänge einzeln und insgesamt angezeigt werden:

West	Ost	Gesamt
13	7	20

Bei einem Ablauf ergibt sich jedoch folgendes inkonsistentes Ergebnis: (Warum ?)

West	Ost	Gesamt
20	20	31

PA-Ausdruck zum Louvre-Beispiel:

```
const N = 4
range T = 0..N
set   VarAlpha 0 {value.{read[T],write[T]}}

VAR       = VAR[0],
VAR[u:T]  = (read[u]    -> VAR[u]
            |write[v:T] -> VAR[v]).

TURNSTILE = (go     -> RUN),
RUN       = (arrive -> INCREMENT
            |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
             -> value.write[x+1]->RUN
            )+VarAlpha.
||LOUVRE  = (east:TURNSTILE || WEST:TURNSTILE
            || {east,west,display}::value:VAR)
             /{go /{east,west].go,
               end/{east,west}.end}.
```

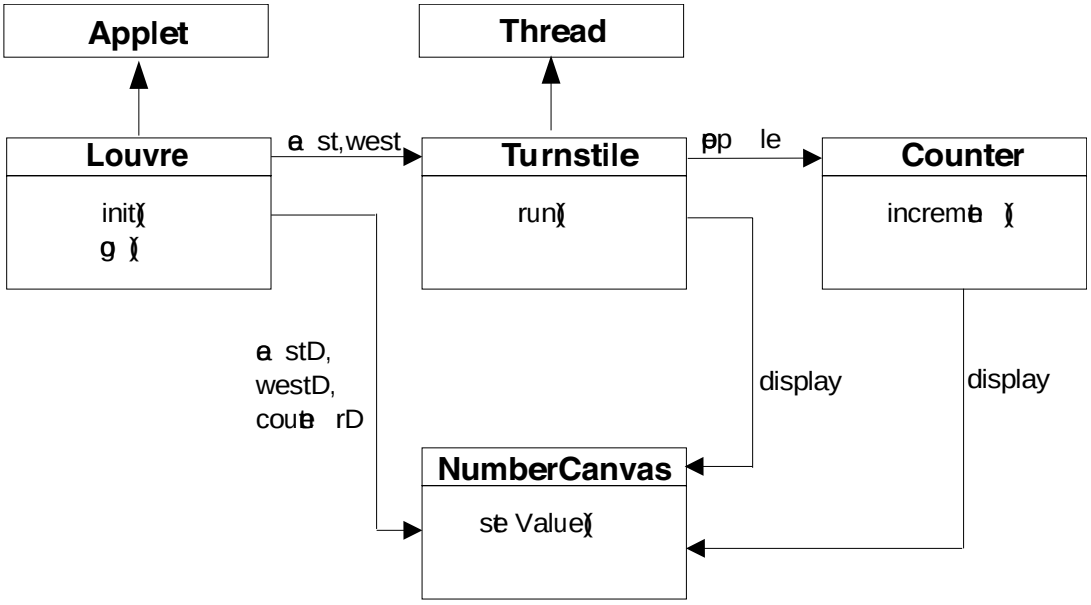


Abbildung 2.19: Klassendiagramm zu LOUVRE

```
private void go() {
  counter = new Counter(counterD);
  west = new Turnstile(westD,counter);
  east = new Turnstile(eastD,counter);
  west.start();
  east.start();
}

class Turnstile extends Thread {
  NumberCanvas display;
  Counter people;

  Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

   puplic void run() {
     try {
       display.setvalue(0);
       for (int i=1; i=1;i<=Garden.MAX;i++){
         Thread.sleep(500); //0.5 second between arrivals
         display.setValue(i);
         people.increment();
       }
     } catch (InterruptedException e) {}
   }
}

class Counter{
  int value=0;
  NumberCanvas display;

  Counter(NumberCanvas n) {
    display=n;
    display.setvalue(value);
  }

  void increment() {
    int temp = value;       //read value
    Simulate.HWinterrupt();
    value=temp+1;           // write value
    display.setvalue(value);
  }
}
```

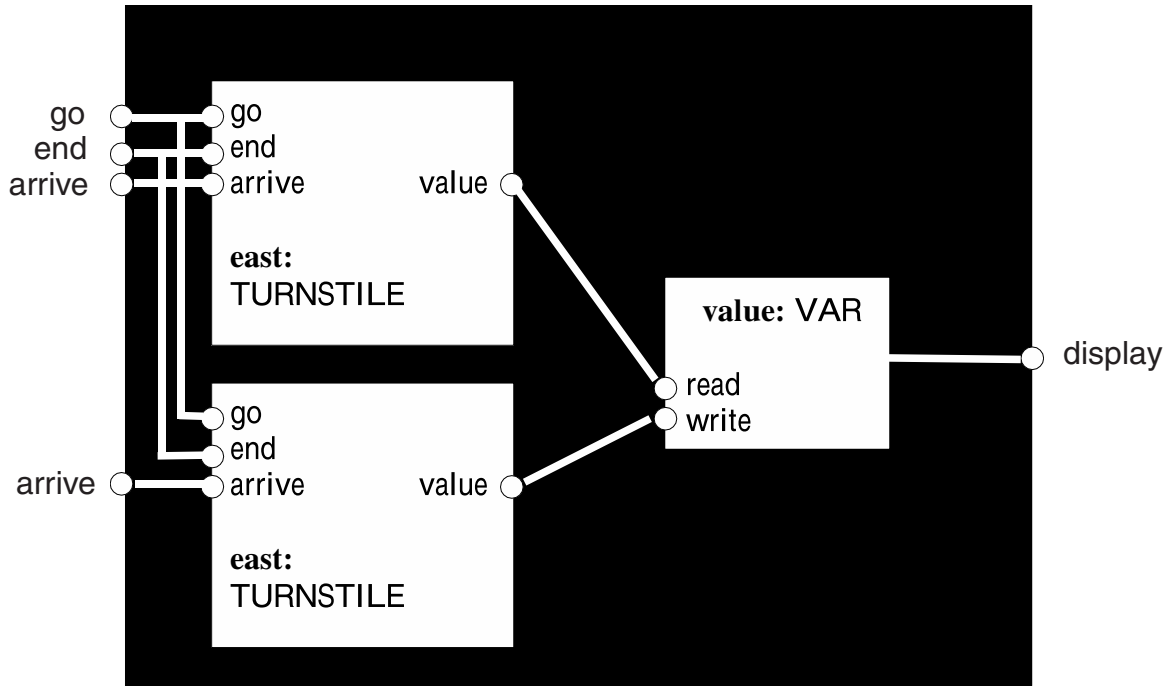


Abbildung 2.20: Strukturdiagramm zu TURNSTILE

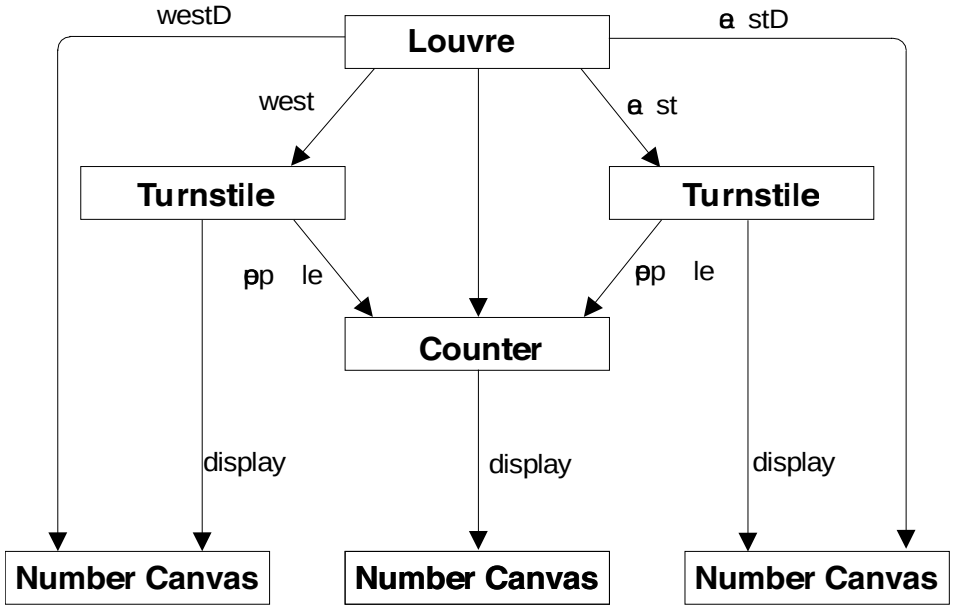


Abbildung 2.21: Objektdiagramm zu Louvre

Kapitel 3

Petrinetze

3.1 Einleitung

Nach der Prozessalgebra werden nun Petrinetze als weitere Modellierungsform von nebenläufigen Systemen behandelt.

3.2 Netze

Petrinetze werden durch ein einfaches Beispiel eingeführt, um wesentliche Prinzipien wie *Lokalität*, *Nebenläufigkeit*, *grafische* und *formaltextuelle Darstellung* daran zu erläutern. Das Beispiel stellt die Synchronisation von Objekten dar, wie sie in vielen Anwendungen in anderer, aber prinzipiell ähnlicher Form vorkommt.

Das Beipiel stellt den Startvorgang eines Autorennens dar [GV01]. Zur Vereinfachung werden nur drei Objekte modelliert: zwei Autos und ein Starter (Abb. 3.1). Wenn die Fahrer der Wagen ihre Vorbereitungen abgeschlossen haben, geben Sie ein Fertigzeichen ("ready sign"). Hat der Starter die Fertigzeichen von allen Wagen erhalten hat, gibt er das Startsignal und die Wagen fahren los.

Man stelle sich vor, der Vorgang soll (z.B. für eine Simulation) modelliert werden. Dabei könnten die folgenden Bedingungen und Aktionen als relevant betrachtet werden:

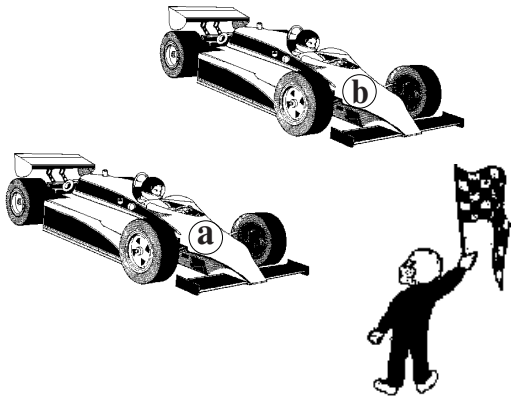


Abbildung 3.1: Start zweier Rennwagen

a) Liste der Bedingungen:	p_1:	car a; preparing for start
	p_2:	car a; waiting for start
	p_3:	car a; running
	p_4:	ready sign of car a
	p_5:	start sign for car a
	p_6:	starter; waiting for ready signs
	p_7:	starter; start sign given
	p_8:	ready sign of car b
	p_9:	start sign for car b
	p_{10}:	car b; preparing for start
	p_{11}:	car b; waiting for start
	p_{12}:	car b; running
b) Liste der Aktionen:	t_1:	car a; send ready sign
	t_2:	car a; start race
	t_3:	starter; give start sign
	t_4:	car b; send ready sign
	t_5:	car b; start race

Die Unterscheidung von “aktiven” und “passiven” Systemkomponenten ist eine wichtige (aber nicht immer eindeutige) Abstraktion. Diese wird durch Netze unterstützt:

I	Das Prinzip der Dualität in Netzen
Es gibt zwei disjunkte Mengen von Grundelementen: *P-Elemente* (state elements, Plätze, Stellen) und *T-Elemente* (Transition-Element, Transistionen). Dinge der realen Welt werden entweder als passive Elemente aufgefaßt und als P-Elemente dargestellt (z.B. Bedingungen, Plätze, Betriebsmittel, Wartepools, Kanäle usw.) oder als aktive Elemente, die durch T-Elemente repräsentiert werden (z.B. Ereignisse, Aktionen, Ausführungen von Anweisungen, Übermitteln von Nachrichten usw.).	

Anmerkung:

- Statt von Plätzen spricht man auch von "Stellen" und "S-Elementen".
- Beispielsweise kann ein Programm ein aktives oder passives Element sein, je nach Kontext.
- Beachte: In der Prozessalgebra wird diese Dualität nicht berücksichtigt, zumindest nicht explizit.

Um zu einem ausführbaren Modell zu kommen, ordnen wir den Bedingungen für den Anfangszustand Wahrheitswerte TRUE und FALSE zu. Im Anfangszustand m_1 bereiten sich die Wagen a und b auf den Start vor (d.h. $p_1 = p_{10} = T$ (TRUE)) und der Starter wartet auf die Fertigzeichen (d.h. $p_6 = T$). Der Anfangszustand ist also als Vektor m_1 darstellbar:

$$\begin{aligned}\mathbf{m}_1 = \; & [\, p_1 = T, p_2 = F, p_3 = F, p_4 = F, p_5 = F, p_6 = T, \\ & p_7 = F, p_8 = F, p_9 = F, p_{10} = T, p_{11} = F, p_{12} = F \,]\end{aligned}$$

Zwei Aktionen, nämlich t_1 und t_4, können hier stattfinden. Betrachten wir die Aktion t_1. Mit ihr gibt der Wagen a das Startzeichen und beendet die Vorbereitungsphase ($p_1 = F$). Dann wartet er auf den Start ($p_2 = T$), nachdem er das Fertigzeichen abgegeben hat ($p_4 = T$). Daraus ergibt sich der neue Zustandsvektor $\mathbf{m}_2$ als:

$$\begin{aligned}\mathbf{m}_2 = \; & [p_1 = F, p_2 = T, p_3 = F, p_4 = T, p_5 = F, p_6 = T, \\ & p_7 = F, p_8 = F, p_9 = F, p_{10} = T, p_{11} = F, p_{12} = F]\end{aligned}$$

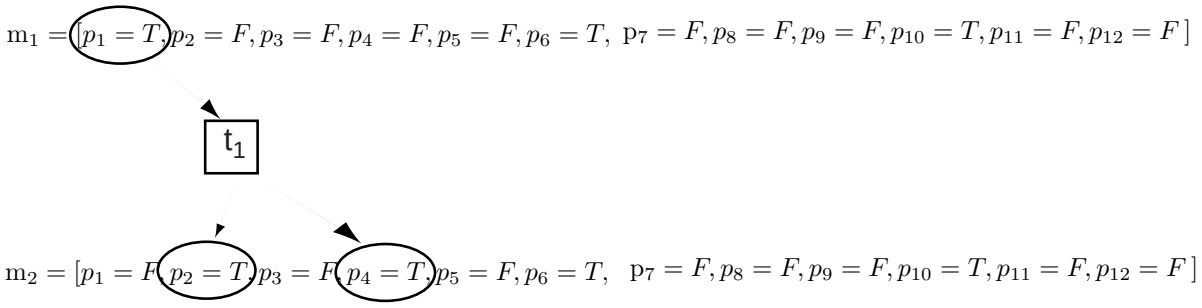


Abbildung 3.2: Lokaltät der Aktion t_1

Die graphische Darstellung dieses Zustandsüberganges in Abb. 3.2 zeigt, dass nur einige Bedingungen beteiligt sind. Sie sind in runde Grafikelemente gefasst und werden Vor- und Nachbedingung der Aktion genannt. Zusammen mit der Aktion (Rechteck) stellen sie den Übergang wesentlich einfacher und adäquater dar. Vor- und Nachbedingung nennen wir die Lokalität der Aktion. Sie allein bestimmt kausale (und zeitliche) Abhängigkeiten mit anderen Aktionen.

Die Aktion t_1 kann stattfinden, wenn p_1 gilt (TRUE) und p_2, p_4 nicht gelten (FALSE). p_1, p_2 und p_4 heißen Bedingungen der Aktion t_1, wobei p_1 *Vorbedingung* und p_2, p_3 *Nachbedingungen* heißen. Zusammen mit dem Aktionsbezeicher nennen wir die Menge $\{t_1, p_1, p_2, p_4\}$ die Lokalität von t_1.

II	**Das Prinzip der Lokalität für Petri Netze**
Das Verhalten einer Transition wird ausschließlich durch ihre *Lokalität* bestimmt, welche sich aus ihr und der Gesamtheit ihrer Eingangs- und Ausgangs-Elemente zusammensetzt.	

Die Bedeutung dieser Begriffsbildung wird deutlicher, wenn wir einbeziehen, dass die zweite Aktion in $\mathbf{m}_2$ stattfinden kann, die $\mathbf{m}_2$ nach $\mathbf{m}_3$ überführt:, wobei:

$$\begin{aligned}\mathbf{m}_3 = \; & [p_1 = F, p_2 = T, p_3 = F, p_4 = T, p_5 = F, p_6 = T, \\ & p_7 = F, p_8 = T, p_9 = F, p_{10} = F, p_{11} = T, p_{12} = F].\end{aligned}$$

Die Lokalität von t_4 ist $\{t_4$,p_{10},p_8,$p_{11}\}$. Also teilen t_1 und t_4 keine Bedingung und sind damit völlig unabhängig. Die Abb. 3.3 drückt dies auch grafisch aus. Dies ist die Basis zur Modellierung von Nebenläufigkeit in Petrinetzen.

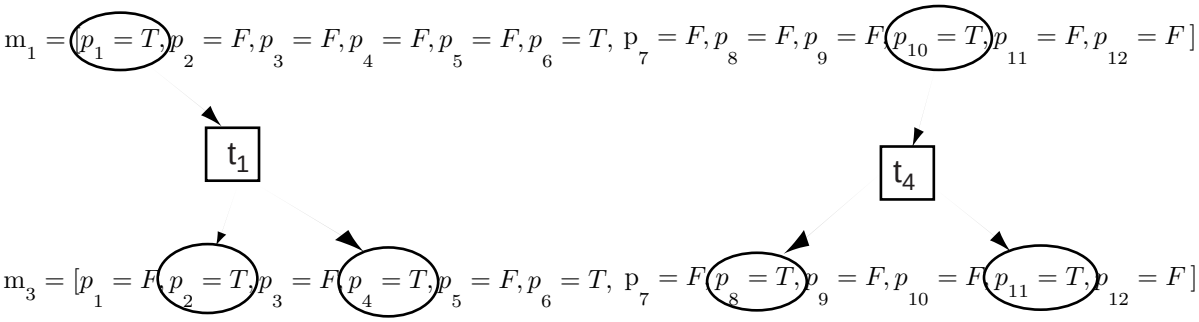


Abbildung 3.3: Concurrent actions t_1 and t_4

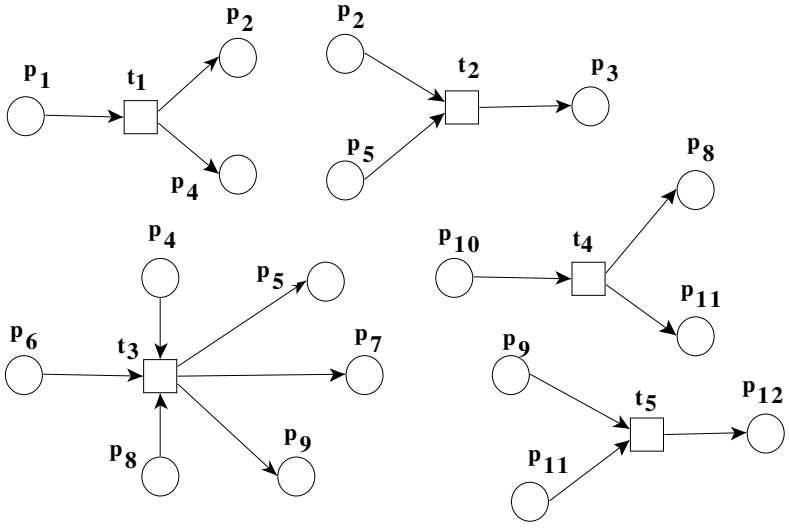


Abbildung 3.4: Einzelaktionen als Transitionen

III	Das Prinzip der Nebenläufigkeit für Petrinetze
Transitionen mit disjunkter Lokalität finden unabhängig (nebenläufig, concurrently) statt.	

Die Abb. 3.4 zeigt *alle* Aktionen mit ihren Vor- und Nachbedingungen. In dieser Form heißen sie Transitionen und die Bedingungen Plätze oder Stellen. Zusammen bilden sie ein *Netz*. Identifiziert man Stellen mit gleichem Bezeichner, so erhält man Abb. 3.5. (Einige Plätze enthalten *Marken*, die den Anfangszustand markieren, worauf wir später zurückkommen.)

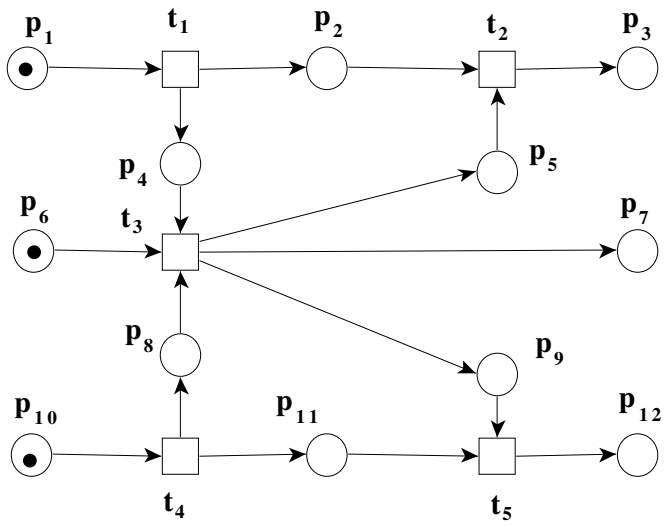


Abbildung 3.5: Netz $\mathcal{N}$ zum Beispiel

IV	Das Prinzip der grafischen Darstellung von Petrinetzen
P-Elemente (auch S-Elemente) werden durch runde grafische Elemente (Kreise, Elipsen,...) dargestellt (rund wie im Buchstaben P oder S). T-Elemente werden durch eckige grafische Elemente (Rechtecke, Balken,...) dargestellt (eckig wie im Buchstaben T). Kanten (auch "Pfeile") verbinden T-Elemente mit den P-Elementen ihrer Lokalität. Also gibt es nur Kanten zwischen T-Elementen und P-Elementen. Außerden gibt es Beschriftungen, wie Bezeichner, Gewichtungen oder Schutzbedingungen (guards).	

In vielen Fällen (z.B. in Texten, zur formalen Beschreibung oder als Datenstruktur) sind formaltextuelle Darstellungen nützlich. Eine solche folgt als mathematische Definition.

Definition 3.1 *Ein* Netz *ist ein Tripel* $\mathcal{N} = (P, T, F)$*, wobei*

- P *eine Menge von* Plätzen *(oder* Stellen*),*
- T *eine mit* P *disjunkte Menge von* Transitionen *und*
- F *die* Flussrelation $F \subseteq (P \times T) \cup (T \times P)$ *darstellt.*

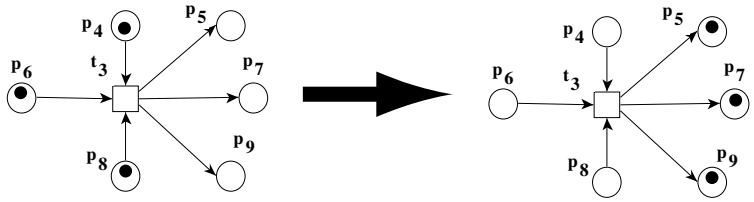


Abbildung 3.6: Schaltregel

Falls P und T endlich sind, dann heißt auch das Netz $\mathcal{N}$ endlich.

Für das Beispielnetz erhält man $P = \{p_1, \ldots, p_{12}\}$, $T = \{t_1, ..., t_5\}$,

$F = \{(p_1, t_1), (t_1, p_2), (t_1, p_4), ...\}$.

Wichtig ist, dass die graphische und mathematische Darstellung äquivalent sind.

V	**Das Prinzip der formaltextuellen Darstellung von Petrinetzen**
Zu jeder graphischen Darstellung eines Petrinetzes gibt es eine äquivalente formaltextuelle Darstellung und umgekehrt.	

Die Gültigkeit einer Bedingung wird durch eine Marke in dem entsprechenden Platz dargestellt (siehe Abb. 3.5). Die Schaltregel wird informell in Abb. 3.6 gezeigt: eine Transition kann schalten, wenn alle Eingangsstellen eine Marke enthalten. Das Ergebnis des Schaltens ist das Entfernen der Marken in den Eingangsstellen und das Hinzufügen der Marken zu den Ausgangsstellen.

Die Abbildung 3.7 zeigt alle möglichen Folgen von Transitionsereignissen. Nebenläufige (z.B. t_1 und t_4) Transitionen sind sowohl als simultaner Schritt wie auch in Folgensemantik dargestellt.

Die folgende Notation für Eingangs- und Ausgangs-Elemente ist üblich:

Definition 3.2 *Für ein Netz $\mathcal{N} = (P, T, F)$ und ein Element $x \in P \cup T$ bezeichnet $^\bullet x := \{y \in P \cup T \mid (y, x) \in F\}$ die Menge der* Eingangselemente *und $x^\bullet := \{y \in P \cup T | (x, y) \in$*

$F\}$ *die Menge der* Ausgangsselemente *von* x*. Ist* x *ein Platz, dann heißen* ${}^\bullet x$ *bzw.* $x^\bullet$ Eingangs- *bzw.* Ausgangs-Transitionen*. Für eine Menge von Elementen* $A \subseteq P \cup T$ *seien entsprechend* ${}^\bullet A := \{y | \exists x \in A : (y,x) \in F\}$ *und* $A^\bullet := \{y | \exists x \in A : (x,y) \in F\}$ *.* $loc(t) := \{t\} \cup {}^\bullet t \cup t^\bullet$*. heißt* Lokalität *von* t*.*

Beispiel: Für $A = \{t_1, p_5, p_{11}\}$ im Netz 3.5 erhält man ${}^\bullet A = \{p_1, t_3, t_4\}$ and $A^\bullet = \{p_2, p_4, t_2, t_5\}$.

3.3 Kantenkonstante und Platz/Transitions Netze

In dem Beispielnetz wurden die Objekte *Wagen a* und *Wagen b* durch einfache Marken repräsentiert. Diese sind an sich nicht zu unterscheiden. In dem Netz 3.5 werden sie nur durch ihre Lage in den unterschiedlichen Plätzen p_1 und p_{10} gekennzeichnet. In Hinblick auf eine direktere Modellierung der realen Situation und um ein kompakteres Modell zu erhalten, wäre es vorteilhaft die Rennwagen direkt durch unterscheidbare Marken (Bezeichner) a und b auf *einem Platz* darzustellen, wie dies in Abb. 3.8 durch den Platz $p_{1\&10}$ gezeigt wird. Dieser Platz repräsentiert gleichsam die Aufstellungszone für den Rennbeginn.

Unterscheidbare Marken werden als Sorten oder Typen gruppiert, die Farben heißen. Jedem Platz p wird durch $cd(p)$ ("colour domain") eine Farbe zugeordnet, hier also $cd(p_{1\&10}) = cars = \{a, b\}$ (in der grafischen Darstellung kursiv beim Bezeichner des Platzes). Andere Beispiele von Farben sind "integer" oder "boolean". Wie im Netz 3.9 zu sehen, spezifizieren Ausdrücke an Kanten wie "a", welche Marke entfernt bzw. hinzugeführt wird. Plätzen ohne Farbspezifikation wird per default die Farbe $token = \{\bullet\}$, also die bekannte Marke zugeordnet. Solche Netze heißen *kantenkonstante gefärbte Netze*, da die Kanten mit Ausdrücken über Konstanten (und nicht wie später mit Ausdrücken über Konstanten *und* Variablen) gewichtet sind. Das Netz 3.10 ist ein weiteres Beispiel, in dem auch die Nachrichten als Farben *ready* und *start* modelliert sind.

Die Kante (p_4, t_3) ist mit dem Ausdruck $rsa + rsb$ gewichtet. Dies bedeutet, dass die Menge $\{rsa, rsb\}$ von p_4 abzuziehen ist. In Netzen werden neben Mengen auch Multimengen benutzt, in denen Elemente mehrfach auftreten können. Multimengen werden als Abbildungen oder als formale Summen dargestellt.

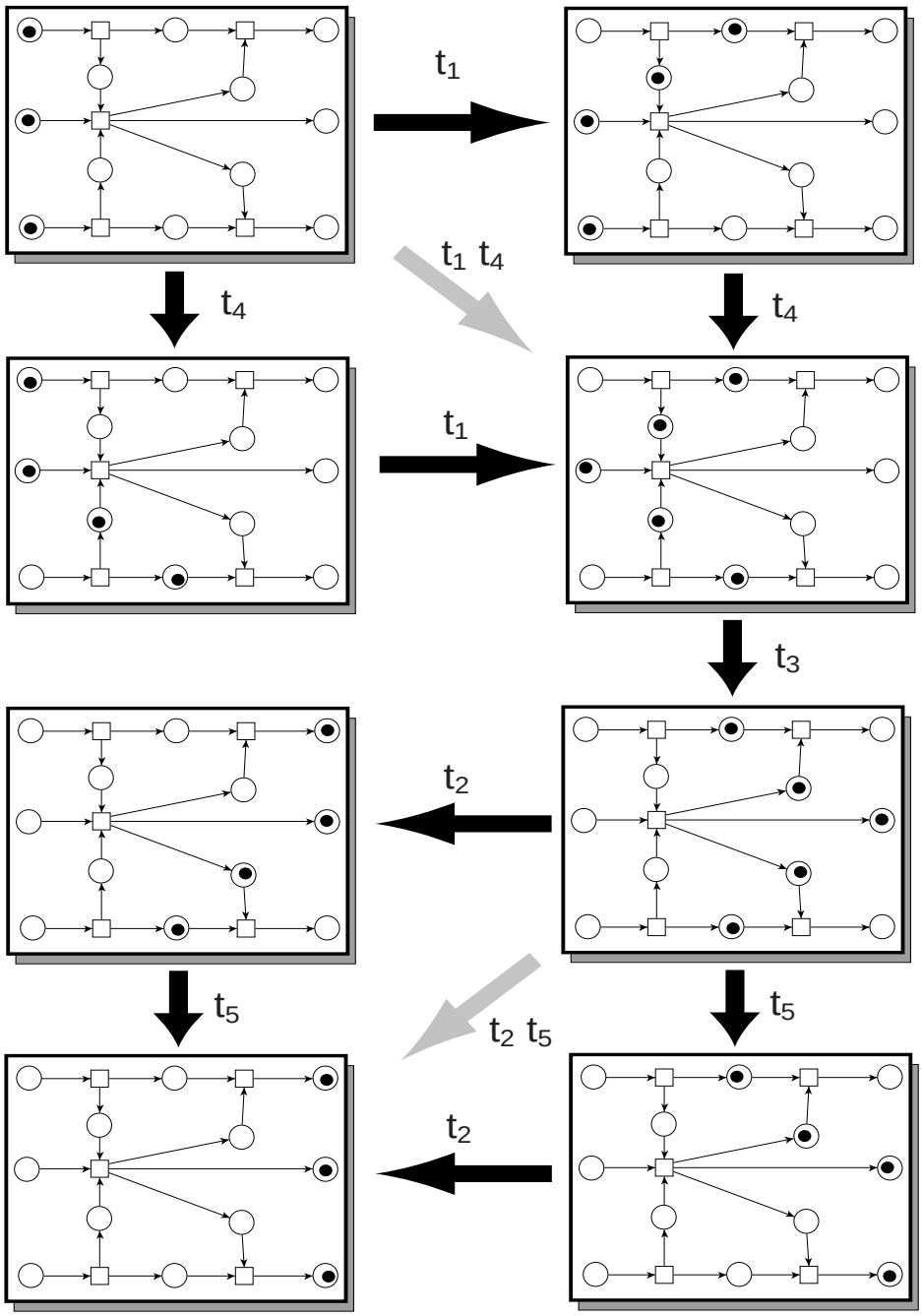


Abbildung 3.7: Schaftfolgen des Netzes 3.5

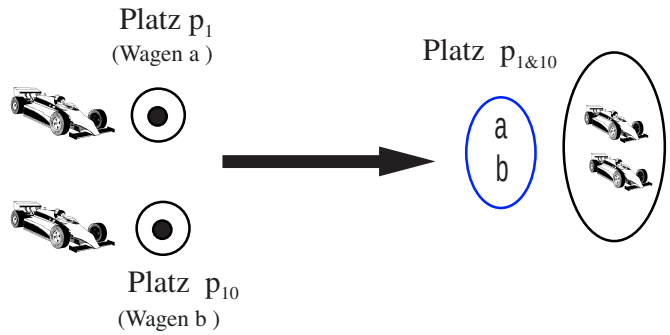


Abbildung 3.8: Übergang zu individuellen Marken

Definition 3.3 *Eine* Multimenge *(bag, multi-set) 'bg', über einer nichtleeren Menge A, ist eine Abbildung $bg : A \to \mathbb{N}$, die auch als formale Summe $\sum_{a\in A} bg(a)'a$ dargestellt wird. $Bag(A)$ bezeichnet die Menge aller Multimengen über A. Die Mengenoperationen Vereinigung, Inklusion, Differenz und Kardinalität werden wie folgt auf $Bag(A)$ als* Summe $(+)$*, Inklusion* $(\leq)$*,* Differenz $(-)$ *und Kardinalität* $(|\cdot|)$ *übertragen:*

Für Multimengen bg, bg_1 and bg_2 über A, wird definiert:

- $bg_1 + bg_2 := \sum_{a\in A}(bg_1(a) + bg_2(a))'a$
- $bg_1 \leq bg_2 :\Leftrightarrow \forall a \in A : bg_1(a) \leq bg_2(a)$
- $bg_1 - bg_2 := \sum_{a\in A}(Max(bg_1(a) - bg_2(a), 0))'a$ *und*
- $|bg| := \sum_{a\in A} bg(a)$ *ist die Mächtigkeit oder Kardinalität von bg (nur definiert, falls die Summe endlich ist) und $\emptyset$ bezeichnet die* leere Multimenge *(mit $|bg| = 0$).*

Beispiel: *$bg_1 = \{a, a, b, b, b, d\}_b$ ist Multimenge über $A = \{a, b, c, d\}$: (der Index unterscheidet die schließende Klammer von der Mengenklammer) oder als formale Summe: $bg_1 = 2'a + 3'b + d$. Mit $bg_2 = a + 2'b + c$ erhält man $bg_1 + bg_2 = 3'a + 5'b + c + d$ and $bg_1 - bg_2 = 1'a + 1'b + 0'c + 1'd = a + b + d$. Multimengen wie $\{a, b, d\}_b$, die Mengen sind, werden auch als Mengen dargestellt: $\{a, b, d\}$*

Definition 3.4 *Ein* kantenkonstantes gefärbtes Petrinetz *(KKN) wird als Tupel*

$\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, W, \mathbf{m}_0\rangle$ definiert, wobei

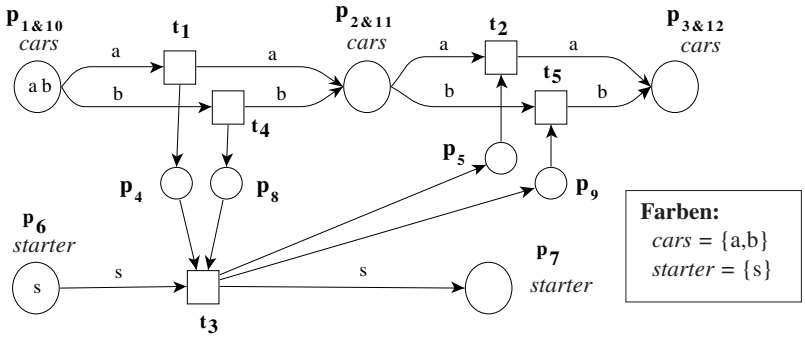


Abbildung 3.9: Das kantenkonstante gefärbte Netz $\mathcal{N}_1$

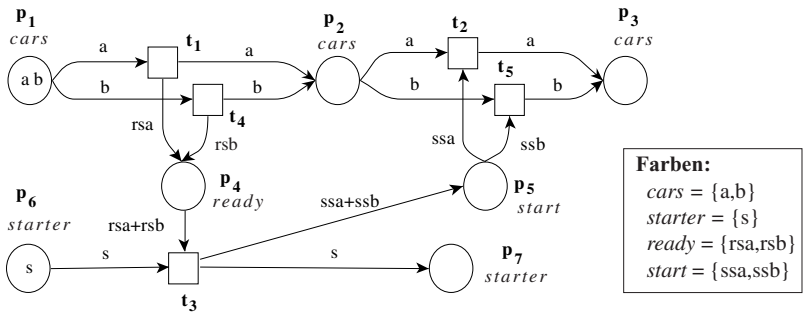


Abbildung 3.10: Das kantenkonstante gefärbte Netz $\mathcal{N}_2$

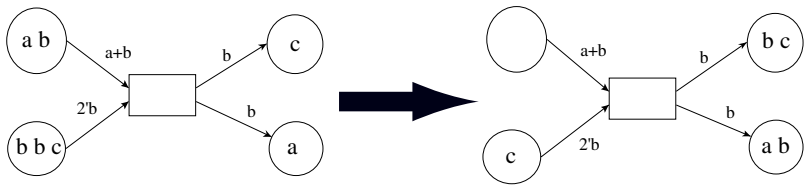


Abbildung 3.11: Schaltregel für kantenkonstante CPN

- (P, T, F) *ein endliches Netz (Def. 3.1) ist ,*
- $\mathcal{C}$ *ist eine Menge von* Farbenmengen*,*
- $cd\colon P \to \mathcal{C}$ *ist die* Farbzuweisungsabbildung *(colour domain mapping). Sie wird durch* $cd\colon F \to \mathcal{C}$, $cd(x,y) :=$ **if** $x \in P$ **then** $cd(x)$ **else** $cd(y)$ **fi** *auf* F *erweitert.*
- $W : F \to Bag(\bigcup \mathcal{C})$ *mit* $W(x,y) \in Bag(cd(x,y))$ *heißt Kantengewichtung.*
- $\mathbf{m}_0 : P \to Bag(\bigcup \mathcal{C})$ *mit* $\mathbf{m}_0(p) \in Bag(cd(p))$ *für alle* $p \in P$ *ist die Anfangsmarkierung.*

Beispiel: Für das kantenkonstante Netz 3.10 ist beispielweise $cd(p_4) = ready = \{rsa, rsb\}$, $cd((p_4,t_3)) = cd(p_4) = \{rsa, rsb\}, W(p_4,t_3)) = 1'rsa + 1'rsb \in Bag(cd(p_4)) = Bag(\{rsa, rsb\})$ und $\mathbf{m}_0(p_1) = 1'a + 1'b \in cd(p_1)$.

Definition 3.5 *a) Die Markierung eines KKN* $\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, W, \mathbf{m}_0 \rangle$ *ist ein Vektor* $\mathbf{m}$ *mit* $\mathbf{m}(p) \in Bag(cd(p))$ *für jedes* $p \in P$ *(auch als Abbildung* $\mathbf{m} : P \to Bag(\bigcup \mathcal{C})$ *mit* $\mathbf{m}(p) \in Bag(cd(p))$ *für jedes* $p \in P$ *aufzufassen).*

b) Eine Transition $t \in T$ *heißt* aktiviert *in einer Markierung* $\mathbf{m}$ *falls* $\forall p \in {}^\bullet t.\ \mathbf{m}(p) \geq W(p,t)$ *(als Relation:* $\mathbf{m} \xrightarrow{t}$*).*

c) Es sei $\widetilde{W}(p,t) := \begin{cases} W(p,t) & \text{falls } (p,t) \in F \\ \emptyset & \text{sonst} \end{cases}$

und entsprechend $\widetilde{W}(t,p) := \begin{cases} W(t,p) & \text{falls } (t,p) \in F \\ \emptyset & \text{sonst} \end{cases}$

Ist t *in* $\mathbf{m}$ *aktiviert, dann ist die* Nachfolgemarkierung *definiert durch* $\mathbf{m} \xrightarrow{t} \mathbf{m}' \Leftrightarrow \forall p \in P.\ (\mathbf{m}(p) \geq \widetilde{W}(p,t) \wedge \mathbf{m}'(p) = \mathbf{m}(p) - \widetilde{W}(p,t) + \widetilde{W}(t,p))$*. (Beachte, daßes sich um Multimengenoperationen handelt!).*

d) Definiert man $W(\bullet,t) := (\widetilde{W}(p_1,t),\ldots,\widetilde{W}(p_{|P|},t))$ *als Vektor der Länge* $|P|$*, dann kann die Nachfolgemarkierung kürzer durch Vektoren definiert werden:*

$\mathbf{m}\xrightarrow{t}\mathbf{m}' \Leftrightarrow \mathbf{m} \geq W(\bullet,t) \wedge \mathbf{m}' = \mathbf{m} - W(\bullet,t) + W(t,\bullet)$.

Dabei sind die Multimengenoperatoren komponentenweise auf Vektoren zu erweitern.

Beispiel: Für das Netz 3.10 ist die Anfangsmarkierung (als Vektor dargestellt) $\mathbf{m}_0 = (a+b,\emptyset,\emptyset,\emptyset,\emptyset,s,\emptyset)$. Da $W(\bullet,t_1) = (a,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset)$ ist die Transition t_1 in $\mathbf{m}_0$ aktiviert und die Nachfolgemarkierung ist $\mathbf{m}' = \mathbf{m}_0 + W(t_1,\bullet) - W(\bullet,t_1) =$ $(a+b,\emptyset,\emptyset,\emptyset,\emptyset,s,\emptyset) + (\emptyset,a,\emptyset,rsa,\emptyset,\emptyset,\emptyset) - (a,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset) = (b,a,\emptyset,rsa,\emptyset,s,\emptyset)$. $a+b$ bzw rsa sind hier beispielsweise die Multimengen $\{a,b\}_b$ bzw. $\{rsa\}_b$, dargestellt als formale Summen.

Die Schaltregel wird in Abb. 3.11 an einem abstrakten Beispiel erläutert, wobei Multimengen vorkommen, die keine Mengen sind!

Definition 3.6 *Die Nachfolgemarkierungsrelation von Definition 3.5 wird wie üblich auf Wörter über* T *erweitert:*

- $\mathbf{m}\xrightarrow{w}\mathbf{m}'$ *falls* w *das leere Wort* λ *ist und* $\mathbf{m} = \mathbf{m}'$*,*
- $\mathbf{m}\xrightarrow{wt}\mathbf{m}'$ *falls* $\exists\mathbf{m}'' : \mathbf{m}\xrightarrow{w}\mathbf{m}'' \wedge \mathbf{m}''\xrightarrow{t}\mathbf{m}'$ *für* $w \in T^*$ *und* $t \in T$*.*

Die Menge $\mathbf{R}(\mathcal{N}) := \{\mathbf{m}|\exists w \in T^* : \mathbf{m_0}\xrightarrow{w}\mathbf{m}\}$ *ist die Menge der* erreichbaren *Markierungen oder auch* Erreichbarkeitsmenge*.* $FS(\mathcal{N}) := \{w \in T^* \mid \exists\mathbf{m} : \mathbf{m_0}\xrightarrow{w}\mathbf{m}\}$ *ist die Menge der* Schaltfolgen *(firing sequence set) von* $\mathcal{N}$*.*

Beispiel: Es folgt die Markierungs/Transitionsfolge für die Schaltfolge $t_4,t_1,t_3,t_2,t_5 \in FS(\mathcal{N}_2)$, wobei zur Abkürzung die Multimengen als Mengen geschrieben sind.

$$\begin{pmatrix}\{a,b\}\\ \emptyset\\ \emptyset\\ \emptyset\\ \emptyset\\ \{s\}\\ \emptyset\end{pmatrix} \xrightarrow{t_4} \begin{pmatrix}\{a\}\\ \{b\}\\ \emptyset\\ \{rsb\}\\ \emptyset\\ \{s\}\\ \emptyset\end{pmatrix} \xrightarrow{t_1} \begin{pmatrix}\emptyset\\ \{a,b\}\\ \emptyset\\ \{rsa,rsb\}\\ \emptyset\\ \{s\}\\ \emptyset\end{pmatrix} \xrightarrow{t_3}$$

$$\begin{pmatrix} \emptyset \\ \{a,b\} \\ \emptyset \\ \emptyset \\ \{ssa,ssb\} \\ \emptyset \\ \{s\} \end{pmatrix} \xrightarrow{t_2} \begin{pmatrix} \emptyset \\ \{b\} \\ \{a\} \\ \emptyset \\ \{ssb\} \\ \emptyset \\ \{s\} \end{pmatrix} \xrightarrow{t_5} \begin{pmatrix} \emptyset \\ \emptyset \\ \{a,b\} \\ \emptyset \\ \emptyset \\ \emptyset \\ \{s\} \end{pmatrix}$$

Ein wichtiger Sonderfall von kantenkonstanten gefärbten Netze stellen die am besten bekannten Platz/Transitins-Netze dar. Bei ihnen ist allen Plätzen als Farbe die einelementige Menge $token = \{\bullet\}$ zugeordnet.

Definition 3.7 (I) *Ein kantenkonstantes Petrinetz (Def. 3.4) $\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, W, \mathbf{m}_0 \rangle$ heißt Platz/Transitions-Netz (P/T-Netz) oder Stellen/Transitions-Netz (S/T-Netz) falls $\mathcal{C} = \{token\} = \{\{\bullet\}\}$.*

Jede Multimenge hat hier die Form $n'\bullet$ mit $n \in \mathbb{N}$. $Bag(token)$ wird daher mit $\mathbb{N}$ identifiziert. Da diese Netzklasse wichtig ist, spezialisieren wir die Definition 3.4 explizit.

Definition 3.8 (II) *Ein Platz/Transitions-Netz (P/T-Netz) oder Stellen/Transitions-Netz (S/T-Netz) wird als Tupel $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ definiert, wobei*

- *(P, T, F) ein endliches Netz (Def. 3.1) ist ,*
- *$W : F \to \mathbb{N}$ Kantengewichtung heißt und*
- *$\mathbf{m}_0 : P \to \mathbb{N}$ die Anfangsmarkierung ist.*

Beispiel: Das Netz 3.12 ist ein P/T-Netz. Die beiden Wagen sind hier als ununterscheidbare Objekte modelliert: “zwei Wagen stehen in Startposition”. Die Schaltregel ist in den Abb. 3.13 und 3.14 an abstrakten Beispielen dargestellt.

Definition 3.9 *a) Die Markierung eines $P/T - Netzes$ $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ ist ein Vektor $\mathbf{m}$ mit $\mathbf{m}(p) \in \mathbb{N}$ für jedes $p \in P$ (auch als Abbildung $\mathbf{m} : P \to \mathbb{N}$ aufzufassen). Die Menge aller Markierungen über P (bzw. S) wird mit M_P (bzw. M_S) bezeichnet.*

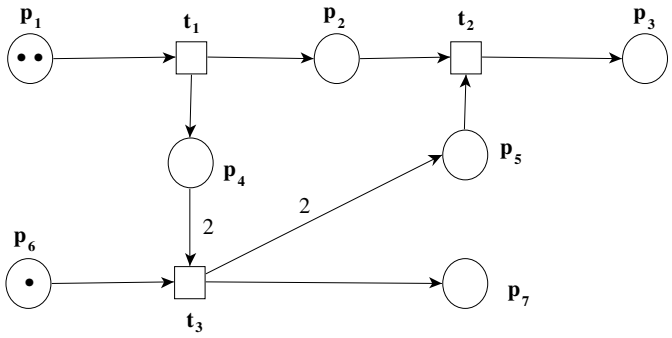


Abbildung 3.12: Platz/Transitions Netz $\mathcal{N}_3$

b) Eine Transition $t \in T$ heißt aktiviert *in einer Markierung $\mathbf{m}$ falls $\forall p \in {}^\bullet t.\ \mathbf{m}(p) \geq W(p,t)$ (als Relation: $\mathbf{m} \xrightarrow{t}$).*

c) Es sei $\widetilde{W}(p,t) := \begin{cases} W(p,t) & \text{falls } (p,t) \in F \\ 0 & \text{sonst} \end{cases}$

und entsprechend $\widetilde{W}(t,p) := \begin{cases} W(t,p) & \text{falls } (t,p) \in F \\ 0 & \text{sonst} \end{cases}$

Ist t in $\mathbf{m}$ aktiviert, dann ist die Nachfolgemarkierung *definiert durch $\mathbf{m} \xrightarrow{t} \mathbf{m}' \Leftrightarrow \forall p \in P.\ (\mathbf{m}(p) \geq \widetilde{W}(p,t) \wedge \mathbf{m}'(p) = \mathbf{m}(p) - \widetilde{W}(p,t) + \widetilde{W}(t,p))$. (Beachte, dass es sich jetzt um Operationen auf $\mathbb{N}$ handelt!).*

d) Definiert man $W(\bullet,t) := (\widetilde{W}(p_1,t), \ldots, \widetilde{W}(p_{|P|},t))$ als Vektor der Länge $|P|$ und entsprechend $W(t,\bullet) := (\widetilde{W}(t,p_1), \ldots, \widetilde{W}(t,p_{|P|}))$, dann kann die Nachfolgemarkierung einfacher durch Vektoren definiert werden:

$\mathbf{m} \xrightarrow{t} \mathbf{m}' \ \Leftrightarrow\ \mathbf{m} \geq W(\bullet,t) \ \wedge\ \mathbf{m}' = \mathbf{m} - W(\bullet,t) + W(t,\bullet))$.

Dabei sind die Operatoren auf $\mathbb{N}$ komponentenweise auf Vektoren zu erweitern.

Beispiel: Im P/T-Netz $\mathcal{N}_3$ von Abb. 3.12 ist die Anfangsmarkierung der Vektor $\mathbf{m}_0 = (2,0,0,0,0,1,0)$ oder (alternativ) die Abbildung $\mathbf{m}_0 : P \to \mathbb{N}$ mit $\mathbf{m}_0(p_1) = 2$, $\mathbf{m}_0(p_6) = 1$ und $\mathbf{m}_0(p_i) = 0$ in den anderen Fällen. Oft ist es praktisch eine Markierung als Wort zu schreiben: $\mathbf{m}_0 = p_1^2 p_6$ oder $\mathbf{m}_0 =< p_1^2 p_6 >$.

Eine Schaltfolge für das P/T-Netz $\mathcal{N}_3$ in der Vektorschreibweise ist:

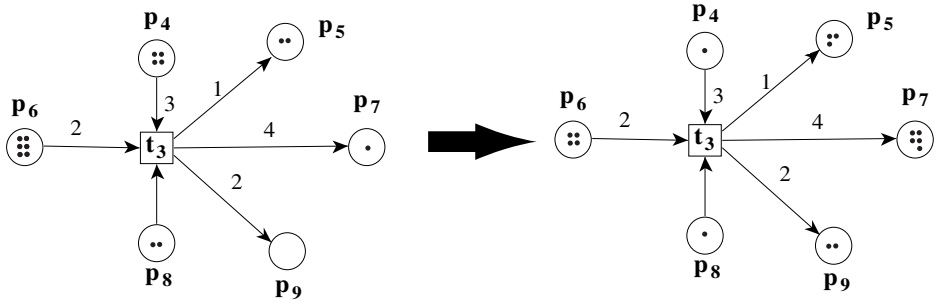


Abbildung 3.13: Schaltregel für P/T-Netze I

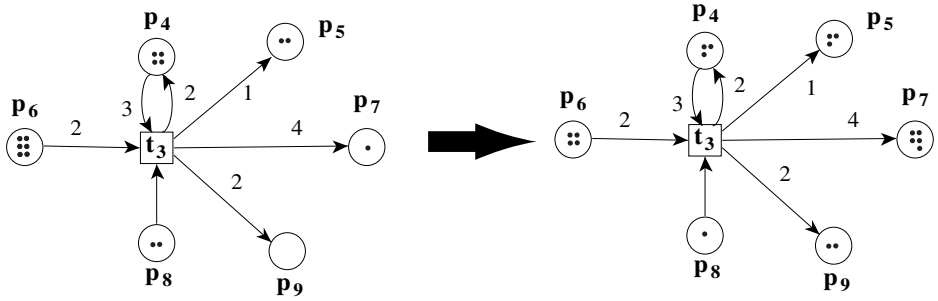


Abbildung 3.14: Schaltregel für P/T-Netze II

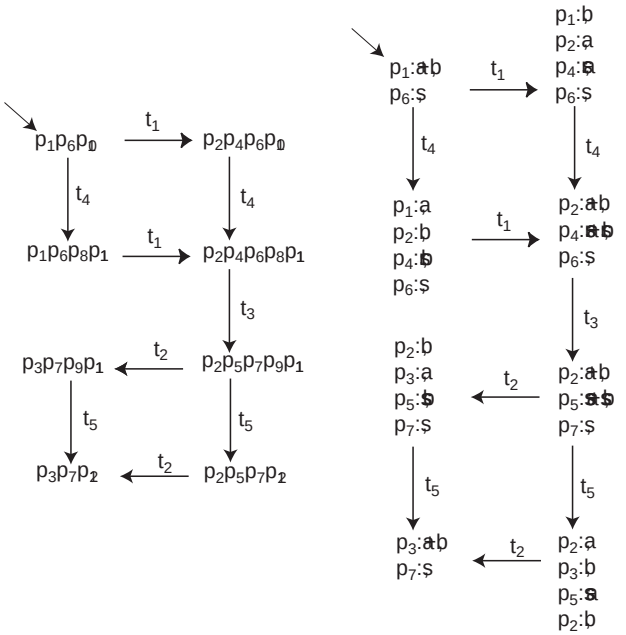


Abbildung 3.15: Erreichbarkeitsgrafen der Netze 3.5 und 3.10

$$\begin{pmatrix}2\\0\\0\\0\\0\\1\\0\end{pmatrix} \xrightarrow{t_1} \begin{pmatrix}1\\1\\0\\1\\0\\1\\0\end{pmatrix} \xrightarrow{t_1} \begin{pmatrix}0\\2\\0\\2\\0\\1\\0\end{pmatrix} \xrightarrow{t_3} \begin{pmatrix}0\\2\\0\\0\\0\\2\\1\end{pmatrix} \xrightarrow{t_2} \begin{pmatrix}0\\1\\1\\0\\0\\1\\1\end{pmatrix} \xrightarrow{t_2} \begin{pmatrix}0\\0\\2\\0\\0\\0\\1\end{pmatrix}$$

Definition 3.10 *Der Erreichbarkeitsgraph eines kantenkonstanten gefärbten Netzes $\mathcal{N}$ ist ein Tupel $RG(\mathcal{N}) := (Kn, Ka)$ mit Knotenmenge $Kn := \mathbf{R}(\mathcal{N})$ (siehe Def. 3.6) und Kantenmenge $Ka := \{(\mathbf{m}_1, t, \mathbf{m}_2) \mid \mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2\}$ vergl. Def. 3.5.*

Beispiel: Der Erreichbarkeitsgraph des P/T-Netzes 3.5 ist links in Abb. 3.15 dargestellt. Vergleiche ihn mit Abb. 3.7! Rechts in dieser Abbildung ist der Erreichbarkeitsgraph des KKN 3.10 zu sehen.

Der Erreichbarkeitsgraph eines Netzes entspricht dem Transitionssystem eines PA-Ausdruckes. Zum Vergleich modellieren wir das Beispiel des Rennwagenstarts mit einem PA-Ausdruck. Dazu verwenden wir ähnliche Namen der Aktionen wie am Anfang des Kapitels:

Liste der Aktionen:
t_1: car a; send ready sign ($send\_a$)
t_2: car a; start race ($start\_a$)
t_3: starter; give start sign ($give\_start\_sign$)
t_4: car b; send ready sign ($send\_b$)
t_5: car b; start race ($start\_a$)

Rennstart:

$$CAR\_A = send\_a \rightarrow start\_a \rightarrow STOP$$
$$CAR\_B = send\_b \rightarrow start\_b \rightarrow STOP$$
$$STARTER = give\_start\_sign \rightarrow STOP$$
$$SIGN\_A = send\_a \rightarrow give\_start\_sign \rightarrow STOP$$
$$SIGN\_B = send\_b \rightarrow give\_start\_sign \rightarrow STOP$$
$$SIGNAL\_A = give\_start\_sign \rightarrow start\_a \rightarrow STOP$$
$$SIGNAL\_B = give\_start\_sign \rightarrow start\_b \rightarrow STOP$$

$$||CARS = (CAR\_A||CAR\_B)$$
$$||STARTING = (STARTER||SIGN\_A||SIGN\_B||SIGNAL\_A||SIGNAL\_B)$$

$$||RACE = (CARS||STARTING)$$

Vergleiche den PA-Ausdruck $||RACE$ mit dem Netz 3.5. Sein Erreichbarkeitsgraf RG($\mathcal{N}$) ist isomorph zum Transitionssystem von $||RACE$ - in diesem Sinne handelt es sich um äquivalente Modelle.

3.4 Gefärbte Netze

Bei der Einführung von kantenkonstanten Netzen wurden Plätze verschmolzen. Um ein Netz noch kompakter zu machen liegt es nahe, auch Transitionen zusammenzulegen, insbesondere wenn sie verhaltensähnliche Aktionen modellieren. Wenn dabei das gleiche Verhalten dargestellt werden soll, dann ist die zusammengelegte Transition zu parametrisieren. Dies kann durch Variablen geschehen, wie dies im Netz 3.19 durch die Variablen

x und y der Fall ist. Beispielsweise modelliert die Transition t_1 im Netz 3.19 durch die Variablenbelegung $\beta_1 := [x = a, y = rsa]$ die Transition t_1 im Netz 3.10, während mit $\beta_2 := [x = b, y = rsb]$ die Transition t_4 in 3.10 simuliert wird. Natürlich wäre $\beta_3 := [x = a, y = rsb]$ keine zulässige Belegung. Dies wird durch Schutzbedingungen (guards) ausgeschlossen. Guards sind über den Variablen an einer Transition definierte Prädikate. Sie können auch als Testbedingung an einer Verzweigung eingesetzt werden. Allgemein stehen an den Kanten eines solchen gefärbten Netzes Ausdrücke über den Variablen, die mit einer zu wählenden Belegung Multimengen definieren, die dann die gleiche Rolle wie bei kantenkonstanten Netzen spielen. Die Form der Ausdrücke lassen wir hier offen, um flexibel zu bleiben. Wichtig ist nur, dass sie zu einer Belegung eine passende Multimenge liefern, d.h. eine Multimenge über der Farbe des angrenzenden Platzes. Entsprechendes gilt für Guards. Wichtig ist hier, dass sie zu einer Belegung einen Wahrheitswert liefern, der über die Zulässigkeit der Belegung entscheidet. Im übrigen ist die Definition eines gefärbten Netzes derjenigen eines kantenkonstanten Netzes ähnlich.

Definition 3.11 *Sei* $Var := \{x_1, x_2, x_3, \ldots\}$ *eine Menge von Variablen mit Wertebereichen* $dom(x)$ *für jedes* $x \in Var$. *Eine* Belegung *ist eine Zuordnung (Abbildung)* $\beta = [x_1 = u_1, x_2 = u_2, x_3 = u_3, \ldots]$ *von Werten* $u_i \in dom(x_i)$ *zu den Variablen.*

Definition 3.12 *Ein* gefärbtes Petrinetz *(coloured Petri net, CPN) wird als Tupel* $\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, Var, Guards, \widehat{W}, \mathbf{m}_0 \rangle$ *definiert, wobei gilt:*

- (P, T, F) *ist ein endliches Netz (Def. 3.1),*
- $\mathcal{C}$ *ist eine Menge von* Farbenmengen*,*
- $cd\colon P \to \mathcal{C}$ *ist die* Farbzuweisungsabbildung *(colour domain mapping). Sie wird durch* $cd\colon F \to \mathcal{C}$, $cd(x, y) :=$ **if** $x \in P$ **then** $cd(x)$ **else** $cd(y)$ **fi** *auf* F *erweitert.*
- Var *ist eine Menge von Variablen mit Wertebereichen* $dom(x)$ *für jedes* $x \in Var$.
- $Guards = \{guard_t \mid t \in T\}$ *ordnet jeder Transition* $t \in T$ *ein Prädikat* $guard_t$ *mit Variablen aus Var zu.*
- $\widehat{W} = \{W_\beta \mid \beta$ *ist Belegung von* $Var\}$ *ist eine Menge von Kantengewichtungen der Form* $W_\beta : F \to Bag(\bigcup \mathcal{C})$, *wobei* $W_\beta(x, y) \in Bag(cd(x, y))$ *für alle* $(x, y) \in F$ *gilt.*

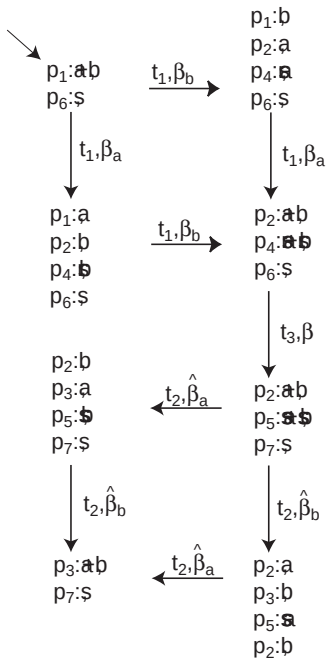


Abbildung 3.16: Erreichbarkeitsgraph des gefärbten Netzes 3.19

- $\mathbf{m}_0 : P \to Bag(\bigcup \mathcal{C})$ *mit* $\mathbf{m}_0(p) \in Bag(cd(p))$ *für alle* $p \in P$ *ist die Anfangsmarkierung.*

Beispiel: Für das gefärbte Netz 3.19 ist beispielweise $cd(p_4) = ready = \{rsa, rsb\}$, $cd((p_4, t_3)) = cd(p_4) = \{rsa, rsb\}$, $W_\beta(p_1, t_1) = 1'a \in Bag(cd(p_1)) = Bag(\{a, b\})$ für $\beta = [x = a, y = rsa]$ und $\mathbf{m}_0(p_1) = 1'a + 1'b \in cd(p_1)$, $guard_{t_1} = (x = a \wedge y = rsa) \vee (x = b \wedge y = rsb)$

Definition 3.13 *a) Die Markierung eines gefärbten Netzes (CPN)* $\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, Var, Guards, \widehat{W}, \mathbf{m}_0 \rangle$ *ist ein Vektor* $\mathbf{m}$ *mit* $\mathbf{m}(p) \in Bag(cd(p))$ *für jedes* $p \in P$ *(auch als Abbildung* $\mathbf{m} : P \to Bag(\bigcup \mathcal{C})$ *mit* $\mathbf{m}(p) \in Bag(cd(p))$ *für jedes* $p \in P$ *aufzufassen).*

b) Sei β *eine Belegung für* Var*. Die Transition* $t \in T$ *heißt* $\beta-$*aktiviert in einer Markierung* $\mathbf{m}$ *falls* $guard_t(\beta) = true$ *und* $\forall p \in {}^\bullet t.\ \mathbf{m}(p) \geq W_\beta(p, t)$ *(als Relation:* $\mathbf{m} \xrightarrow{t,\beta}$*).*

c) Es sei $\widetilde{W_\beta}(p,t) := \begin{cases} W_\beta(p,t) & \textit{falls } (p,t) \in F \\ \emptyset & \textit{sonst} \end{cases}$
und entsprechend $\widetilde{W_\beta}(t,p) := \begin{cases} W_\beta(t,p) & \textit{falls } (t,p) \in F \\ \emptyset & \textit{sonst} \end{cases}$
Ist t in $\mathbf{m}$ β *– aktiviert, dann ist die* Nachfolgemarkierung *definiert durch* $\mathbf{m} \xrightarrow{t,\beta} \mathbf{m}' \Leftrightarrow \forall p \in P.\ (\mathbf{m}(p) \geq \widetilde{W_\beta}(p,t) \wedge \mathbf{m}'(p) = \mathbf{m}(p) - \widetilde{W_\beta}(p,t) + \widetilde{W_\beta}(t,p))$. *(Beachte, daßes sich um Multimengenoperationen handelt!).*

d) Definiert man $W_\beta(\bullet,t) := (\widetilde{W_\beta}(p_1,t),\ldots,\widetilde{W_\beta}(p_{|P|},t))$ *als Vektor der Länge* $|P|$ *und entsprechend* $W_\beta(t,\bullet) := (\widetilde{W_\beta}(t,p_1),\ldots,\widetilde{W_\beta}(t,p_{|P|}))$, *dann kann die Nachfolgemarkierung einfacher durch Vektoren definiert werden:*

$\mathbf{m} \xrightarrow{t,\beta} \mathbf{m}' \ \Leftrightarrow\ \mathbf{m} \geq W_\beta(\bullet,t) \ \wedge\ \mathbf{m}' = \mathbf{m} - W_\beta(\bullet,t) + W_\beta(t,\bullet))$.

Dabei sind die Multimengenoperatoren komponentenweise auf Vektoren zu erweitern.

e) $\mathbf{m} \xrightarrow{t} \mathbf{m}' :\Leftrightarrow \exists\beta.\ \mathbf{m} \xrightarrow{t,\beta} \mathbf{m}'$

Beispiel: Für das Netz 3.19 ist die Anfangsmarkierung (als Vektor dargestellt) $\mathbf{m}_0 = (a+b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset)$. Für $\beta = [x = a, y = rsa]$ und $W_\beta(\bullet, t_1) = (a, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ ist die Transition t_1 in $\mathbf{m}_0$ aktiviert und die Nachfolgemarkierung ist $\mathbf{m}' = \mathbf{m}_0 + W_\beta[t_1, \bullet] - W_\beta(\bullet, t_1) = (a+b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset) + (\emptyset, a, \emptyset, rsa, \emptyset, \emptyset, \emptyset) - (a, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) = (b, a, \emptyset, rsa, \emptyset, s, \emptyset)$. Mit $a+b$ bzw rsa sind hier beispielsweise die Multimengen $\{a,b\}_b$ bzw. $\{rsa\}_b$ als formale Summen dargestellt.

Die Schaltregel wird in Abb. 3.17 an einem abstrakten Beispiel erläutert:

1. Wähle (wie in b)) eine Belegung β für Var(t) mit $guard_t(\beta) = true$.
2. Werte mit dieser Belegung die Ausdrücke an den Kanten zu Multimengen aus (wie in c).
3. Wende die Schaltregel für kantenkonstsante Netze (Abb. 3.11) an (wie in d)).

Eine *Markierungs-Schaltfolge* für das gefärbte Netz von Abb. 3.19 sieht folgendermaßen aus, wobei die Belegungen $\beta_a = [x = a, y = rsa]$, $\beta_b = [x = b, y = rsb]$ und $\hat{\beta}_a = [x = a, y = ssa]$, $\hat{\beta}_b = [x = b, y = ssb]$ gewählt wurden. Für die Transition t_3 ist kann eine beliebige Bindung eingesetzt werden, da alle Kantenausdrücke nur Konstante enthalten.

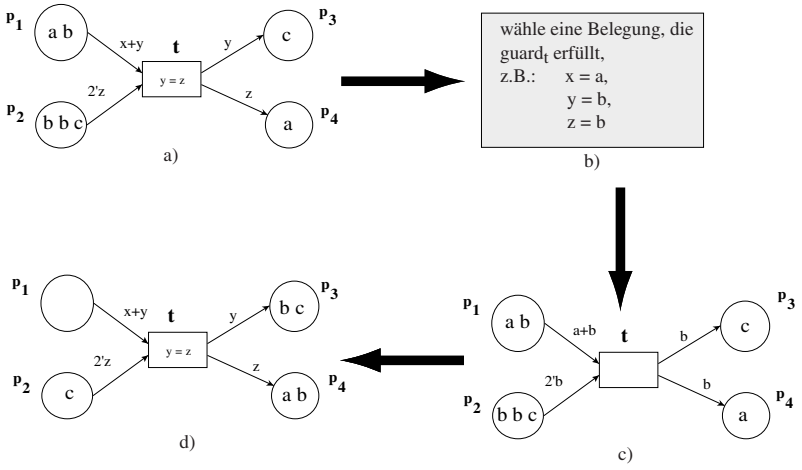


Abbildung 3.17: Schaltregel für gefärbte Netze

$$\begin{pmatrix} \{a,b\} \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \{s\} \\ \emptyset \end{pmatrix} \xrightarrow{(t_1,\beta_b)} \begin{pmatrix} \{a\} \\ \{b\} \\ \emptyset \\ \{rsb\} \\ \emptyset \\ \{s\} \\ \emptyset \end{pmatrix} \xrightarrow{(t_1,\beta_a)} \begin{pmatrix} \emptyset \\ \{a,b\} \\ \emptyset \\ \{rsa, rsb\} \\ \emptyset \\ \{s\} \\ \emptyset \end{pmatrix} \xrightarrow{(t_3,\beta)}$$

$$\begin{pmatrix} \emptyset \\ \{a,b\} \\ \emptyset \\ \emptyset \\ \{ssa, ssb\} \\ \emptyset \\ \{s\} \end{pmatrix} \xrightarrow{(t_2,\hat{\beta}_a)} \begin{pmatrix} \emptyset \\ \{b\} \\ \{a\} \\ \emptyset \\ \{ssb\} \\ \emptyset \\ \{s\} \end{pmatrix} \xrightarrow{(t_2,\hat{\beta}_b)} \begin{pmatrix} \emptyset \\ \emptyset \\ \{a,b\} \\ \emptyset \\ \emptyset \\ \emptyset \\ \{s\} \end{pmatrix}$$

Definition 3.14 *Die Nachfolgemarkierungsrelation von Definition 3.13 wird wie üblich auf Wörter über T erweitert:*

- $\mathbf{m} \xrightarrow{w} \mathbf{m}'$ *falls w das leere Wort λ ist und* $\mathbf{m} = \mathbf{m}'$,

- $\mathbf{m} \xrightarrow{wt} \mathbf{m}'$ *falls* $\exists \mathbf{m}'' : \mathbf{m} \xrightarrow{w} \mathbf{m}'' \wedge \mathbf{m}'' \xrightarrow{t} \mathbf{m}'$ *für* $w \in T^*$ *und* $t \in T$.

Die Menge $\mathbf{R}(\mathcal{N}) := \{\mathbf{m} | \exists w \in T^* : \mathbf{m_0} \xrightarrow{w} \mathbf{m}\}$ *ist die Menge der* erreichbaren *Markierungen oder auch* Erreichbarkeitsmenge. $FS(\mathcal{N}) := \{w \in T^* \mid \exists \mathbf{m} : \mathbf{m_0} \xrightarrow{w} \mathbf{m}\}$ *ist die Menge der* Schaltfolgen *(firing sequence set) von* $\mathcal{N}$.

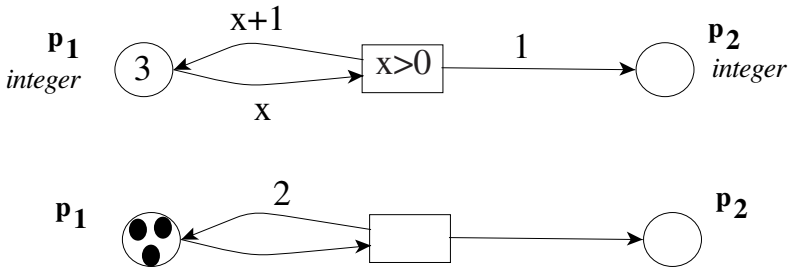


Abbildung 3.18: Gefärbtes Netz und P/T-Netz mit ähnlichem Verhalten

Vergleiche das gefärbte Netz mit dem P/T-Netz in Abb. 3.18. Beide stellen einen Zähler dar.

Definition 3.15 *Der Erreichbarkeitsgraph eines gefärbten Netzes* $\mathcal{N}$ *ist ein Tupel* $RG(\mathcal{N}) := (Kn, Ka)$ *mit Knotenmenge* $Kn := \mathbf{R}(\mathcal{N})$ *(siehe Def. 3.6) und Kantenmenge* $Ka := \{(\mathbf{m}_1, (t, \beta), \mathbf{m}_2) \mid \mathbf{m}_1 \xrightarrow{t,\beta} \mathbf{m}_2\}$ *(vergl. Def. 3.5).*

Beispiel: Der Erreichbarkeitsgraph des gefärbten Netzes 3.19 ist in Abb. 3.16 dargestellt.

Kantenausdrücke von gefärbten Netzen können auch speziell definierte Funktionen enthalten. Diese werden wie in Abb. 3.20 zweckmäßigerweise in den Deklarationsteil aufgenommen. Der Vorteil dieser Darstellung liegt in der Reduzierung der Anzahl der Variablen. Dadurch werden in diesem Beispiel Guards in Transitionen nicht benötigt.

3.5 Dijkstras Bankier und das Alternierbit-Protokoll

Beispiel 1: Das Bankiersproblem

Die Problematik von Verklemmungen bei der Betriebsmittelvergabe wurde von Dijkstra als Problem des Bankiers dargestellt [Dij68]. Hier wurde auch der Begriff des sicheren

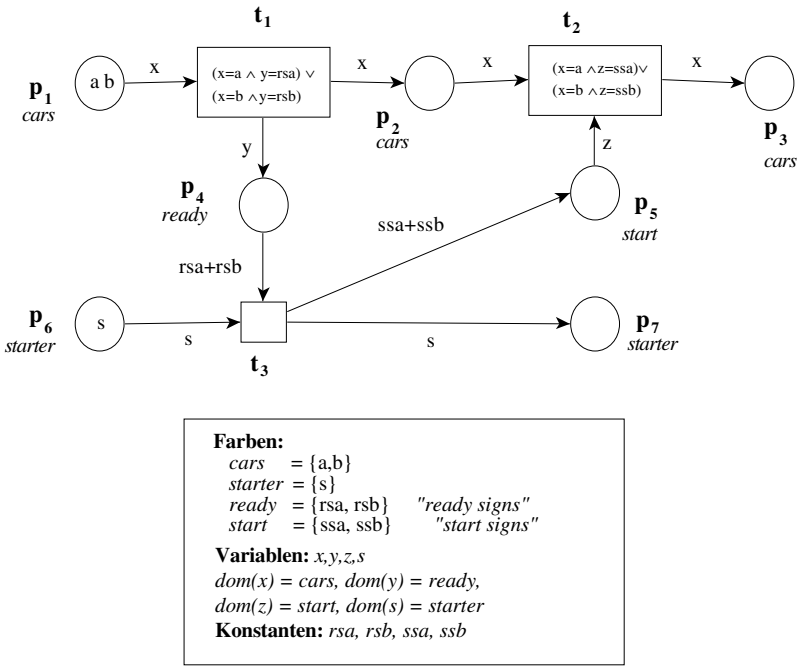


Abbildung 3.19: Gefärbtes Netz $\mathcal{N}_5$ mit Guards

Zustands eingeführt.

Ein Bankier besitzt ein Kapital g. Seine n Kunden erhalten wechselnden Kredit. Jeder Kunde muss seinen maximalen Kreditwunsch von vorneherein bekanntgeben und wird nur als Kunde akzeptiert, wenn dieser das Kapital nicht übersteigt. Kredite werden nicht entzogen. Dafür muss aber jeder Kunde versprechen, den Maximalkredit auf einmal nach endlicher Zeit zurückzuzahlen. Der Bankier verspricht, jede Bitte um Kredit in endlicher Zeit zu erfüllen. Für den Bankier besteht natürlich das Problem der Verklemmung: es kann sein, dass mehrere Kunden noch nicht ihren Maximalkredit erhalten haben, das Restkapital des Bankiers aber zu klein ist, mindestens einen Kunden total zu befriedigen, um dann nach einer Frist wieder neues Kapital zu haben.

Eine Instanz $\iota = (n,f,\mathrm{g})$ des Bankiersproblem besteht aus einer Zahl $n \in \mathbb{N}$, einem n-Tupel $f = (f_1, \ldots, f_n)$ und einer Zahl $g \in \mathbb{N}$. Ein Zustand einer solchen Instanz ist ein n-tupel $r = (r_1, \ldots, r_n)$ das das die Restforderung der Kunden beschreibt. Anfangs gilt $r = f$. Ein Zustand heißt *sicher*, wenn er nicht notwendig zu einer Verklemmung führt.

Das P/T-Netz in Abb. 3.22 modelliert das beschriebene Bankiersproblem. Der Platz *BANK* enthält so viele Marken wie das Kapital des Bankiers in Geldeinheiten umfasst.

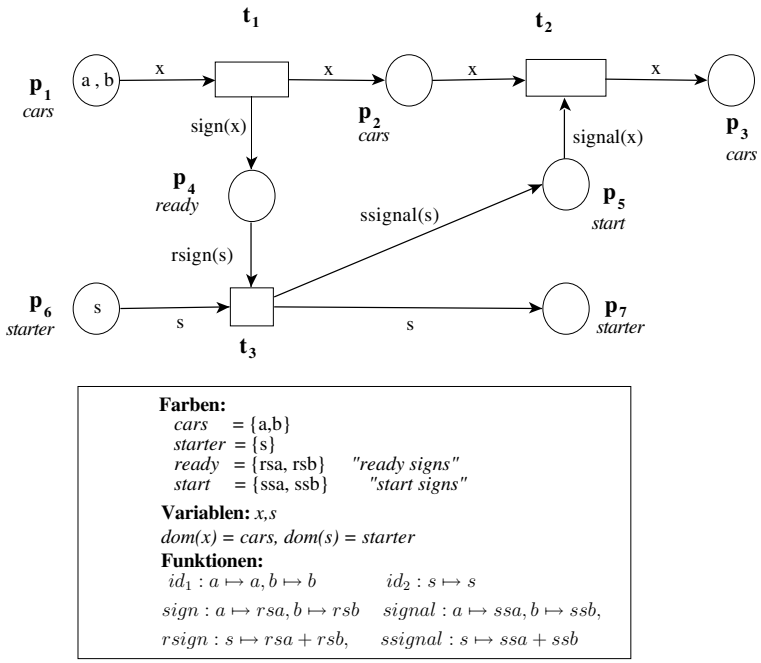


Abbildung 3.20: Gefärbtes Netz mit definierten Funktionen

$CREDIT_i$ und $CLAIM_i$ stehen für Kredit und Restforderung. Durch die Transition $GRANT_i$ erhält der Kunde i so viele Geldeinheiten, wie sie schaltet. $RETURN_i$ transferiert das Geld zurück. Diese Transition kann nur schalten, wenn der Bankier die Maximalforderung f_i des Kunden erfüllt hat. Gleichzeitig wird die Ausgangsforderung wieder hergestellt.

Betrachten wir zwei spezielle Instanzen, nämlich $\iota_1 = (2, (8, 6), 10)$ (Abb. 3.23) und $\iota_2 = (3, (8, 3, 9), 10)$ (Abb. 3.25). An ihnen werden Erreichbarkeitsgraf und P-Invarianten erläutert.

Für die Instanz $\iota_1 = (2, (8, 6), 10)$ in Abb. 3.23a wird jeder Zustand durch eine Markierung dargestellt, d.h. durch eine 5-dimensionalen Vektor. Für alle erreichbaren Markierungen $\mathbf{m}$ gelten die folgenden 3 Gleichungen:

(Solche Gleichungen werden als P-Invarianten-Gleichungen im Abschnitt 4.1.1 behandelt.)

- $\mathbf{m}[BANK] + \mathbf{m}[CREDIT_1] + \mathbf{m}[CREDIT_2] = 10$
  (Das Geld ist bei der Bank oder bei den Kunden und zwar genau 10 Einheiten

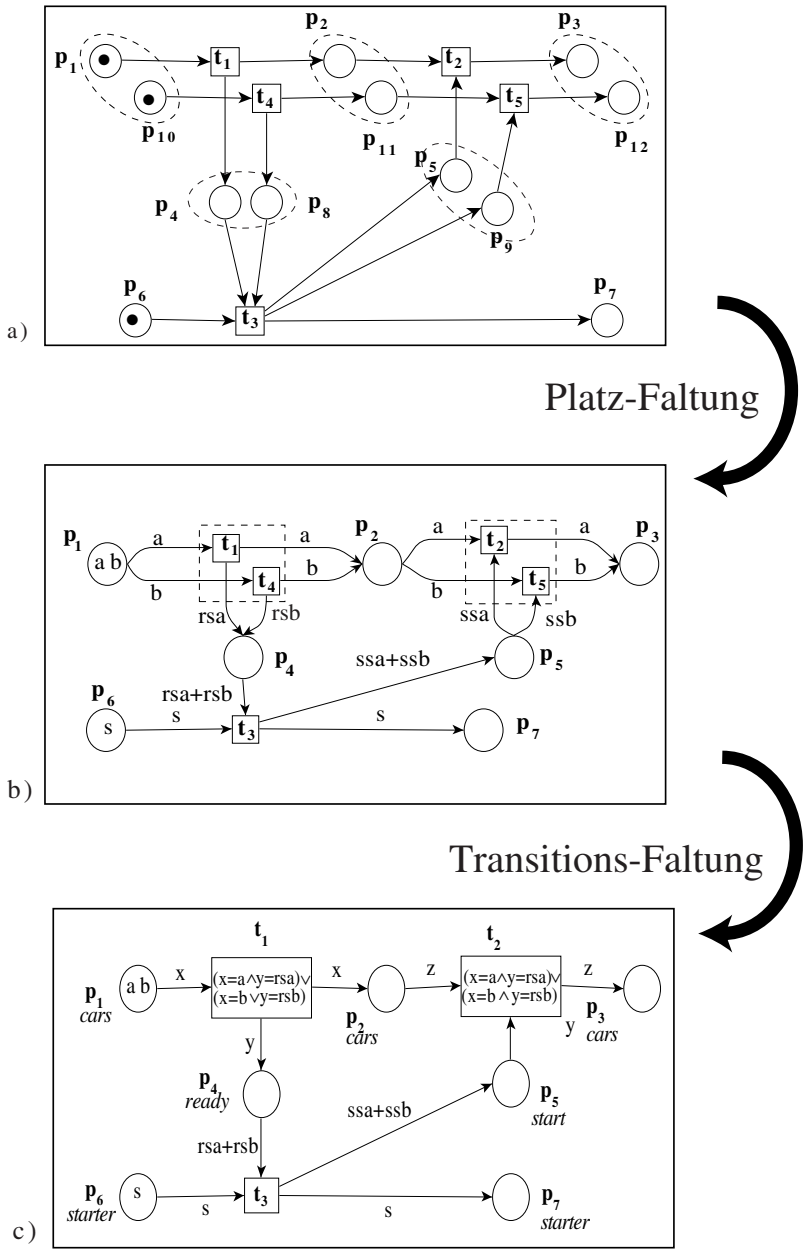


Abbildung 3.21: Netzfaltung zum kantenkonstanten und gefärbten Netz

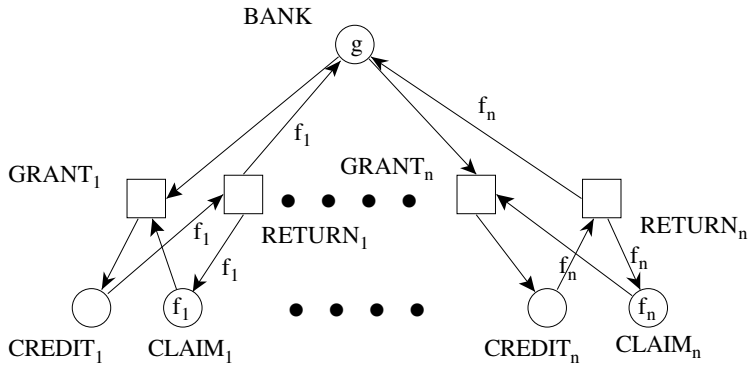


Abbildung 3.22: Das Bankiersproblem mit n Kunden

insgesamt.)

- $\mathbf{m}[CLAIM_1] + \mathbf{m}[CREDIT_1] = 8$
  (Kredit und Restforderung des Kunden 1 beträgt zusammen immer genau 8 Einheiten.)

- $\mathbf{m}[CLAIM_2] + \mathbf{m}[CREDIT_2] = 6$
  (Kredit und Restforderung des Kunden 2 beträgt zusammen immer genau 6 Einheiten.)

(Sie heißen P-Invarianten-Gleichungen und werden im nächsten Abschnitt 4.1.1 systematisch behandelt.) Also ist jede erreichbare Markierung schon durch 2 ihrer Komponenten festgelegt, nämlich $(CLAIM_1, CLAIM_2)$. Die anderen 3 Komponenten, $BANK, CREDIT_1$ und $CREDIT_2$ können mit den Gleichungen daraus berechnet werden. Dadurch kann der Erreichbarkeitsgraf in einem ebenen Gitter dargestellt werden (Abb. 3.24).

Die Anfangsmarkierung $\mathbf{m}_0 = (10, 0, 8, 0, 6)$ (wobei die Plätze folgendermaßen geordnet seien: $(BANK,\ CREDIT_1,\ CLAIM_1,\ CREDIT_2,\ CLAIM_2)$) wird auf das Paar $(\mathbf{m}_0(CLAIM_1), \mathbf{m}_0(CLAIM_2)) = (8, 6)$ reduziert. Es entspricht dem Knoten rechts oben im Grafen von Abb. 3.24.

Alle Pfade, die in diesem Knoten anfangen, entsprechen Schaltfolgen. Kanten nach links, rechts, unten und oben entsprechen jeweils dem Schalten der Transition $GRANT_1$, $RETURN_1, GRANT_2$ und $RETURN_2$. Werden die Kunden strikt nacheinander bedient,

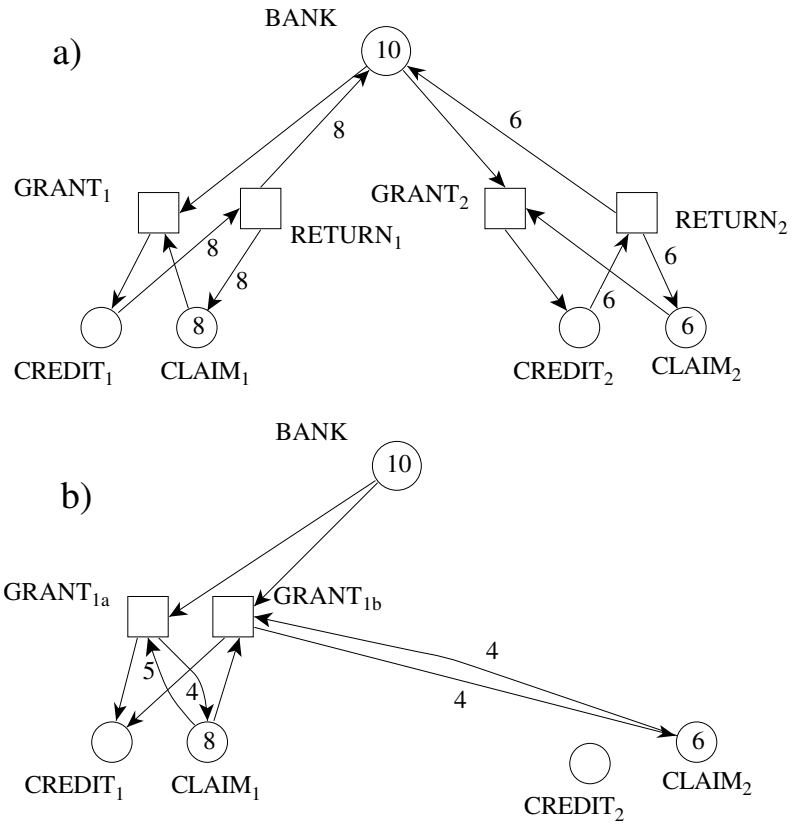


Abbildung 3.23: Eine Bankiersprobleminstanz mit 2 Kunden und Modifikation zur Vermeidung von Verklemmungen

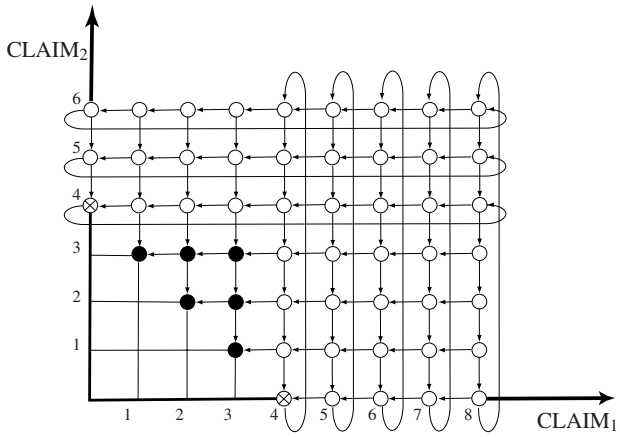


Abbildung 3.24: Erreichbarkeitsgraph des Netzes 3.23

so gibt es keine Probleme. Falls sich jedoch die Ausleihschritte überlappen, kann einer der drei Verklemmungen (1,3),(2,2) und (3,1) kommen.

Dijkstra hat darauf hingewiesen, dass es noch andere kritische Zustände gibt, nämlich diejenigen, die unvermeidlich zu einer Verklemmung führen, wie z.B. (3,3). Er nannte sie *unsicher*. Sie sind als schwarze Knoten dargestellt. In [HV87] und [VJ85] wurde gezeigt, dass *sichere Zustände* (weiße Knoten) durch ihre *minimalen Elemente* darstellbar sind: (0,4) and (4,0), (mit Kreuz).

Wie kann man unsichere Zustände vermeiden? Der Algorithmus von Dijkstra berechnet vor jedem Ausleihschritt, ob von dem dann erreichten Nachfolgezustand der Anfangszustand noch erreichbar ist.

Wie aus Abb. 3.24 ersichtlich kann dies auch dadurch geschehen, dass Transition $GRANT_1$ nur in Markierungen aktivierbar ist, die (in mindestens einer Komponente) größer als $(4,0)$ oder $(0,4)$ sind. Dies wird erreicht, wenn man die Transition $GRANT_1$ durch zwei modifizierte Kopien $GRANT_{1a}$ und $GRANT_{1b}$ (Abb. 3.23b) ersetzt. Diese beiden Transitionen haben den gleichen Schalteffekt wie die ursprüngliche, aber einen höheren Aktivierungs-"Schwellwert". Die entsprechende Konstruktion ist bei $GRANT_2$ anzuwenden. Der allgemeine Algorithmus ist in [VJ85] und teilweise in [JV87] S.216 ff. zu finden.

Die zweite Instanz $\iota_2 = (3,(8,3,9),10)$ ist ein Beispiel aus dem Buch [BH73] über Betriebssysteme. Seine Netzdarstellung befindet sich in Abbildung 3.25. Es enthält 7 Plätze. Wegen der nun 4 Gleichungen kann der Erreichbarkeitsgraf in drei Dimensionen dargestellt werden (Abb.3.27). Bezeichnend ist das Anwachsen seiner Größe. Er enthält 195 erreichbare Markierungen. Die Teilmenge von 137 sicheren Markierungen (weiße Knoten) wird von 10 minimalen Elementen (dem Residuum: weiße Knoten mit Kreuz)) erzeugt. Eine allgemeine Methode zu ihrer Berechnung wird in [HV87] gegeben. Die 58 unsicheren Markieungen sind wieder schwarz dargestellt.

Die zweite und größere Instanz hat aber auch Eigenschaften, die in der ersten nicht zu beobachten waren: sie enthält Markierungen, von denen aus Verklemmungen vermieden werden können, die aber nicht sicher sind, wenn sicher heißt, dass alle Kunden ihre Transaktionen beenden können. Beispielsweise kann von der Markierung $(4,3,6)$ ausgehend, der zweite Kunde beliebig viele Transaktionen ausführen, während die Kunden 1 und 3 nichtmal einen Ausleihvorgang vollständig zu Ende bringen können. Solche Situatio-

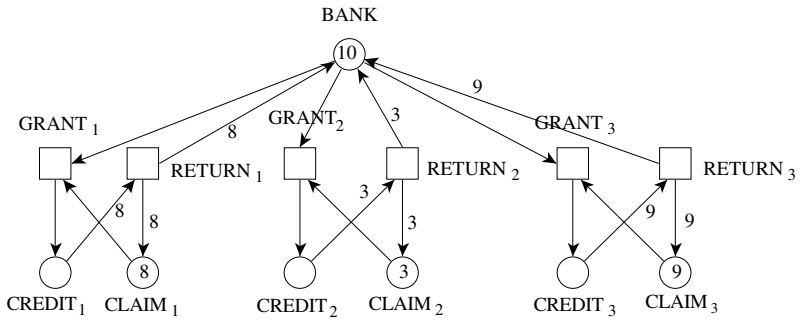


Abbildung 3.25: Eine Instanz des Bankiersproblems mit 3 Kunden

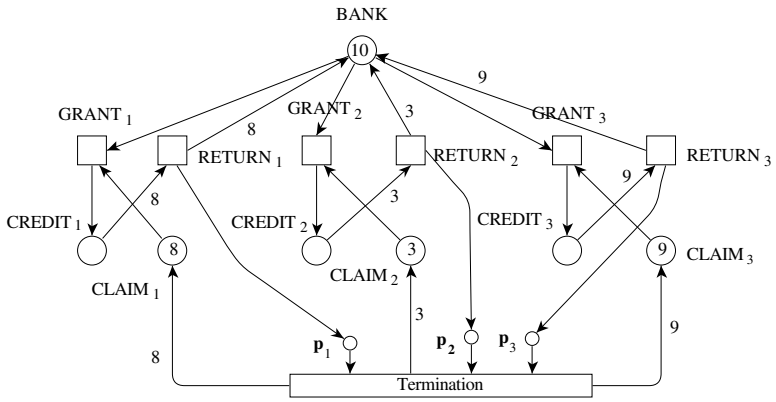


Abbildung 3.26: Bankiersnetz mit Terminationstransition

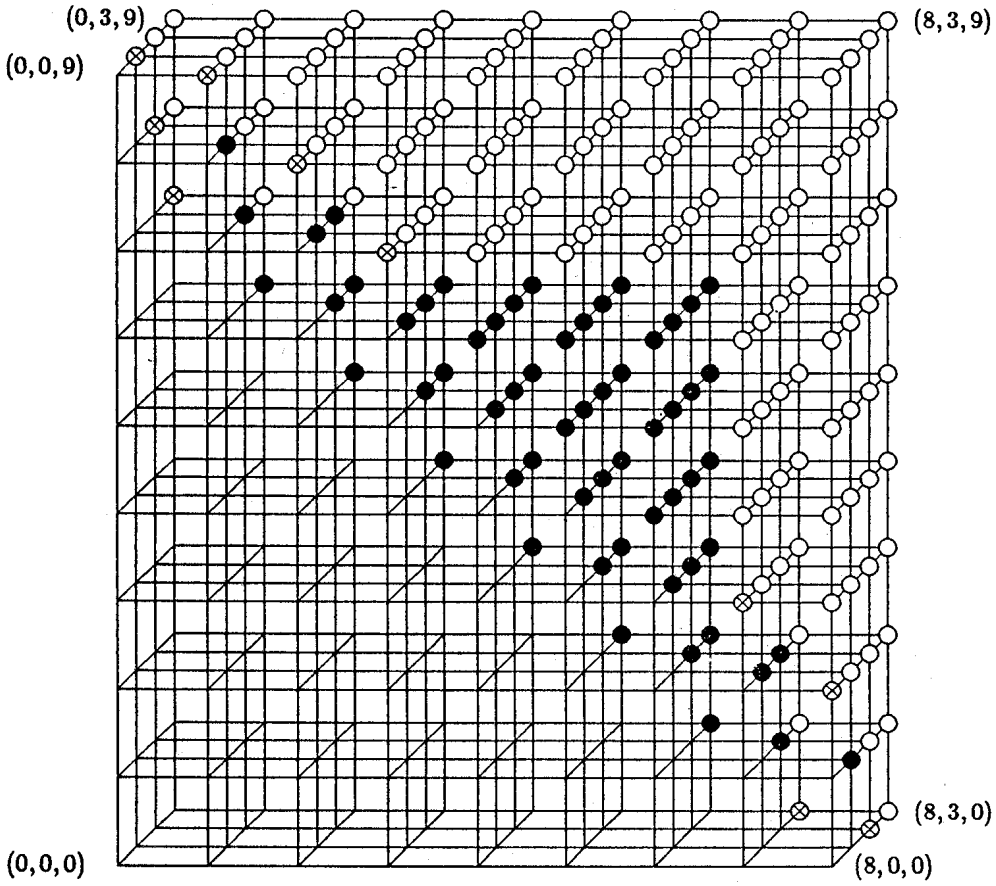


Abbildung 3.27: Erreichbarkeitsgraph des Netzes 3.25

nen heißen partielle Verklemmung. Ein Netz ohne partelle Verklemmungen heitß*lebendig*. Lebendigkeit wird im Abschnitt 4.1.2 behandelt. Um die ursprüngliche Definition von Dijkstra beizubehalten, kann man wie in Abb. 3.26 eine Transition *TERMINATION* einführen, die dafür sorgt, dass alle Kunden einen Ausleihvorgang beendet haben, bevor eine neue Runde started. Um den Begriff der sicheren Markierung sinnvoll auch in dem Netz 3.25 zu benutzen, könnte man folgende Definitionen unter c) benutzen.

Nach dieser Diskussion sind folgende Definitionen sinnvolle Übertragungen des Begriffs der Sicherheit auf Netze:

a) Eine Markierung **m** heißt sicher, wenn eine spezifizierte Endmarkierung (hier die Anfangsmarkierung) wieder erreichbar ist.

b) Eine Markierung **m** heißt sicher, wenn eine bestimmte Transition (z.B. TERMINATION) zum Schalten gebracht werden kann.

c) Eine Markierung **m** heißt sicher, wenn von ihr aus eine unendliche Schaltfolge möglich ist, die alle Transitionen des Netzes unendlich oft enthält.

Ist in a) die Anfangsmarkierung gemeint, dann wird diese Eigenschaft in der Petrinetzliteratur auch "homing property" genannt. Die Eigenschaft c) hat mit dem "fairen Verhalten" eines Netzes zu tun, das auch im Abschnitt 4.1.2 behandelt wird. Fairness und Lebendigkeit stehen in komplexen Beziehungen zueinander.

Am Ende der Diskussion des Bankiersproblem nutzen wir es für ein Beispiel eines gefärbten Netzes. In Abb. 3.28 ist das P/T-Netz 3.25 als gefärbtes Netz modelliert. Die Restforderung ist beispielsweise eine Multimenge, die nach Kunden sortiert die jeweiligen Restforderungen enthält. Die Kantenbewertung von $(CREDIT, RETURN)$ hat beispielsweise die Form eines bedingten Ausdrucks, der für die Belegung $\beta = [y = c]$ den Wert $W_\beta(CREDIT, RETURN) = 9'c = \{c, c, c, c, c, c, c, c, c\}_b$ annimmt. (Die Belegung von x ist hier ohne Wirkung - deswegen das Fragezeichen.) Der Definitionsbereich der Variablen y ist durch die Farbe *clients* gegeben, was hier sinnvoll ist, da dies eine problemadäquate Parametrisierung ist. Andererseits sieht man hier, dass der Wertebereich einer Variablen nicht die Farbe eines Platzes sein *muss*. Man hätte auch $dom(y) = \{1, 2, 3\}$ und $W_\beta(CREDIT, RETURN) = \text{case } y \text{ of } [1 \to 8'a \mid 2 \to 3'a \mid 3 \to 9'a]$ wählen können!

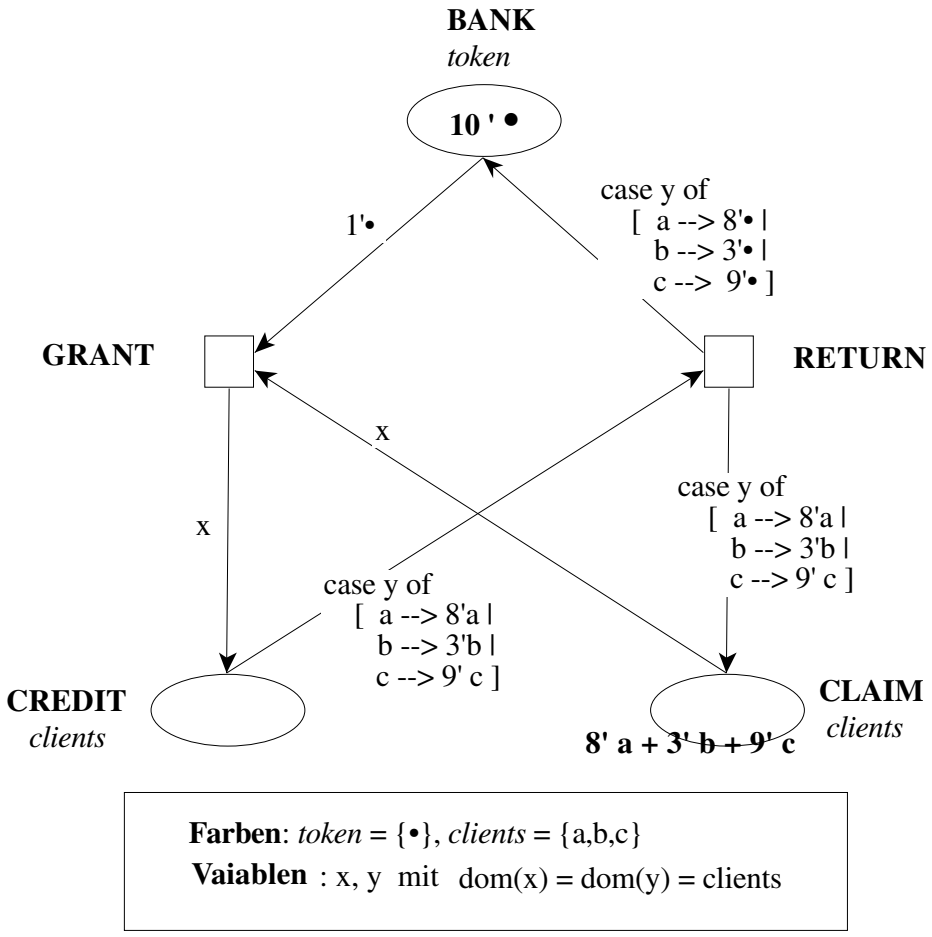


Abbildung 3.28: Faltung des Bankiersnetz 3.25 zu einem gefärbtes Netz

Beispiel 2: Das Alternierbitprotokoll

In diesem Beispiel wird das bekannte Alternierbit-Protokoll als gefärbtes Netz modelliert. Für die Übermittlung einer Nachricht zwischen zwei Arbeitsrechnern (hosts) (Abb 3.29) stehe ein Kanal zur Verfügung, auf dem in beiden Richtungen, aber nur in einer Richtung zur Zeit, eine Nachricht übermittelt werden kann (Halbduplex-Kanal) (Abb. 3.30). Bei der Übermittlung kann die Nachricht gestört werden. Durch redundante Kodierung wird aber jeder Fehler erkannt und angezeigt. Eine fehlerhafte Übermittlung wird im gefärbten Netz von Abb. 3.30 durch Schalten der Transition g statt h dargestellt, wobei die Nachricht d vernichtet und die Konstante F (für fehlerhafte Nachricht) abgeliefert wird. (Die Halbduplexeigenschaft ist nicht explizit dargestellt, da sie sich später automatisch ergibt.) Um das gewünschte fehlerfreie Protokoll zu realisieren, hat man natürlich die Möglichkeit, durch Rückmeldung den korrekten Empfang quittieren zu lassen oder bei Fehlern ein erneutes Senden herbeizuführen. Problematisch dabei ist jedoch, daß auch bei der Rückmeldung Fehler auftreten können. Erhält der Sender eine gestörte Quittung, sendet er sicherheitshalber die alte Nachricht noch einmal. Der Empfänger muß nun aber davor bewahrt werden, die Wiederholung von d_i für die nächste Nachricht d_{i+1} zu halten. Um dies zu erreichen, wird der Nachricht d ein Bit x beigepackt und vor Ablieferung der Nachricht an Y wieder entfernt. In Abb. 3.32 ist die Verfeinerung von Abb. 3.31 unter Benutzung des Halbduplex-Kanals von Abb. 3.30 als gefärbtes Netz dargestellt.

Als Besonderheit kommt dieses Protokoll zur Quittierung mit nur einem Bit x aus ([BS69]). Es heißt daher auch *Alternierbitprotokoll*.

Wird die Nachricht gestört (Transition g), dann schaltet die Transition n. Diese befördert das nicht geänderte Alternierbit x nach s_{13} und mit i nach s_{12}. Wegen $x \neq y$ wird dann die Nachricht (x, d) nochmal gesendet (Transition q), deren Kopie in s_5 aufbewahrt wurde. Transition q schaltet auch, wenn eine ordnungsgemäße Quittung durch k gestört als F ankommt. In diesem Fall wird die zum zweiten Male ankommende Nachricht durch Schalten der Transition m gelöscht und nicht an Y weitergegeben. Wir verzichten hier auf einen Beweis, der in [Hai82] nachgelesen werden kann. Dort wird mit Mitteln der temporalen Logik für die Folge $\alpha = d_1 d_2 \ldots$ der von X abgegebenen und für die Folge $\beta = d_1 d2 \ldots$ der von Y aufgenommenen Nachrichten gezeigt, daß β immer Präfix von α ist und beide unendlich sind, also zusammen: $\alpha = \beta$. Dazu muß vorausgesetzt werden, daß das Netz verschleppungfrei schaltet. Außerdem muß man annehmen, daß von kei-

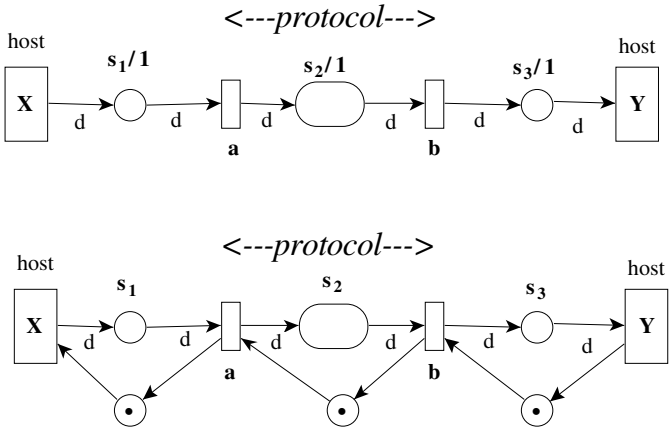


Abbildung 3.29: Spezifikation des Alternierbitprotokolls

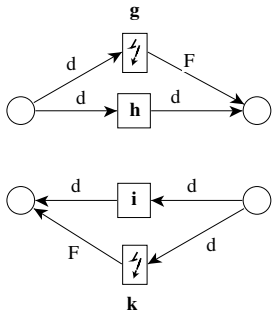


Abbildung 3.30: Halbduplex-Kanal mit Fehlererkennung

nem Zeitpunkt an der Kanal permanent gestört ist, d.h. daß die Transitionen h und i unendlich oft schalten. Dies ist zum Beispiel durch die Annahme darstellbar, daßh und i fair schalten.

Abschließend zeigen wir, wie das gefärbte Netz 3.32 zu einem einfachen Netz (P/T-Netz, dass auf jedem Platz höchstens eine Marke enthält) "aufgefaltet" werden kann. Dazu verzichten wir jedoch auf die Darstellung des Datentransports durch die Variable d. Die Abb. 3.33 zeigt zunächst die Auffaltung des äußeren Rings, in dem das Bit läuft. Jeder Platz und jede Transition tritt hier doppelt auf, was die Belegungen $[x = 1]$ und $[x = 0]$ repräsentiert. Einer Idee von [Obe81] folgend ist die geometrische Anordnung so gewählt, dass kreuzende Pfeile ein Alternieren des Bits anzeigen. Die Plätze s_{11} und s_{12} sind insofern anders, als sie eine Farbe mit 3 Elementen 0,1 und F haben. Daher werden sie jeweils durch 3 Plätze $\{s_{11}(0),\ s_{11}(1), s_{11}(F)\}$ ersetzt. Im zweiten Schritt werden in

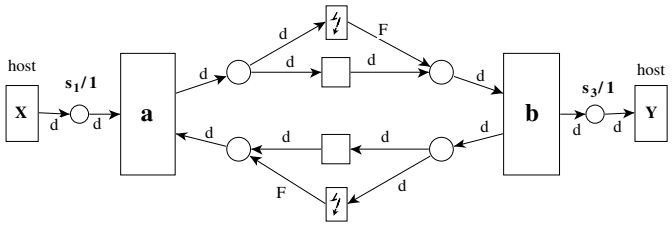
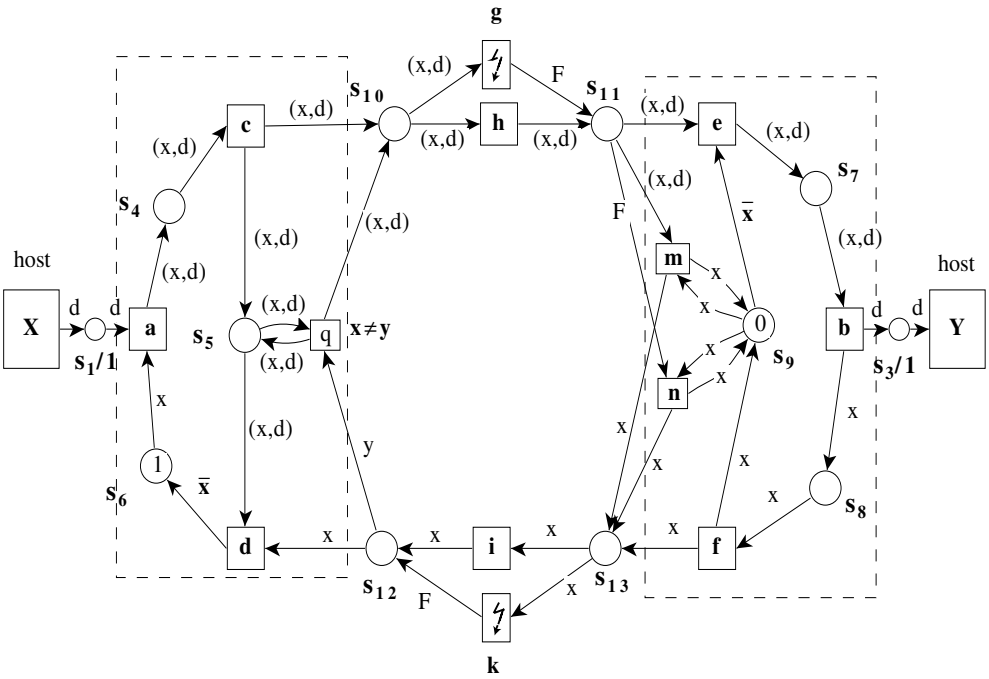


Abbildung 3.31: Kanal mit Knotenrechnern X und Y



Farben:
bit = {0,1}, *fit* = {0,1,F}, *data*
$cd(s_{11}) = cd(s_{12}) =$ *fit*, cd(s1) = cd(s3) = *data*
cd(si) = *bit* × *data* für i = 4,5,10,11,7
cd(s) = *bit* sonst
Variablen: d,x,y mit dom(x) = dom(y) = *bit*
dom(d) = *data*
Konstanten: F
Funktionen: $\bar{\cdot}: 0 \mapsto 1, 1 \mapsto 0$

Abbildung 3.32: Realisierung des Alternierbitprotokolls

Abb. 3.34 auch die restlichen Transitionen und Plätze in entsprechender Weise eingefügt.

Das Beispiel zeigt deutlich Unterschiede und Gemeinsamkeiten von gefärbten und ungefärbten Netzen. Das gefärbte Netz ist natürlich kompakter. Dafür kann in dem ungefärbten Netz der Weg des Bits samt seinen Alternierungen direkter verfolgt werden.

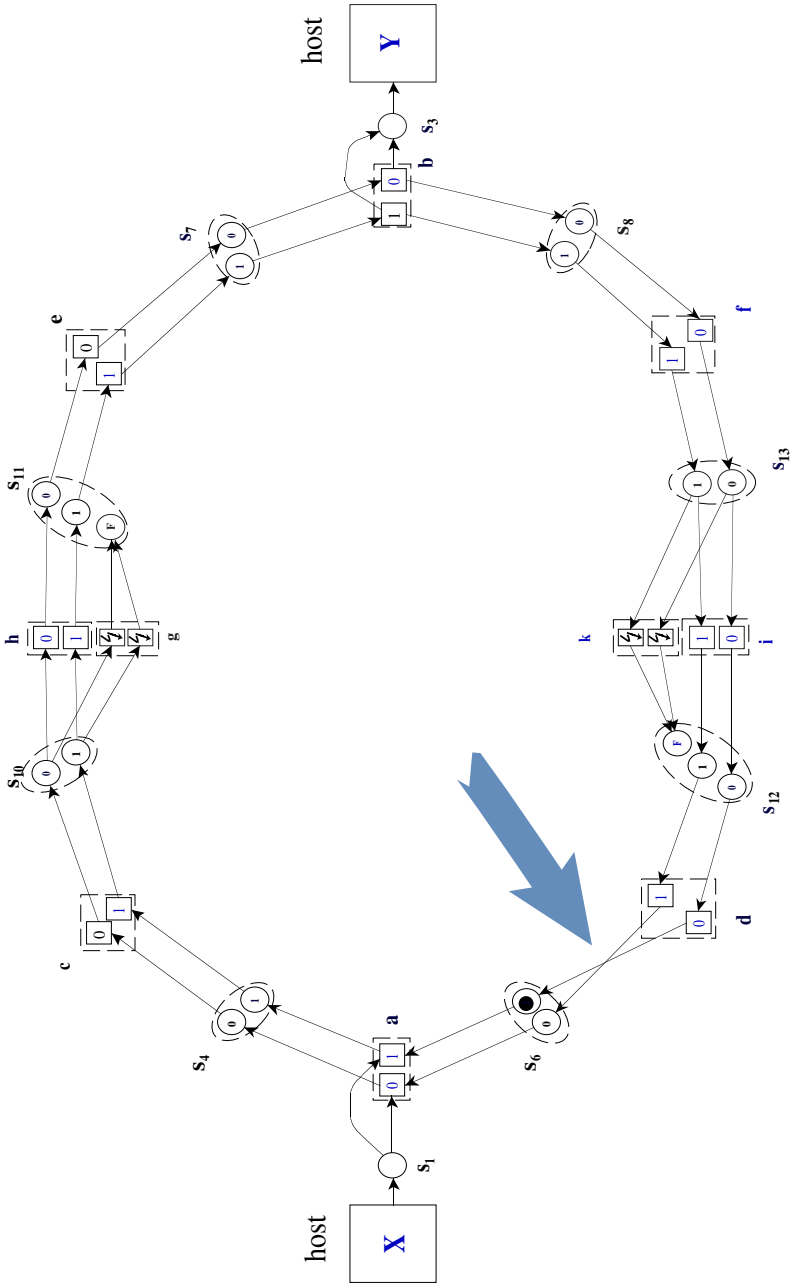


Abbildung 3.33: Auffaltung des Alternierbitprotokolls / 1.Schritt

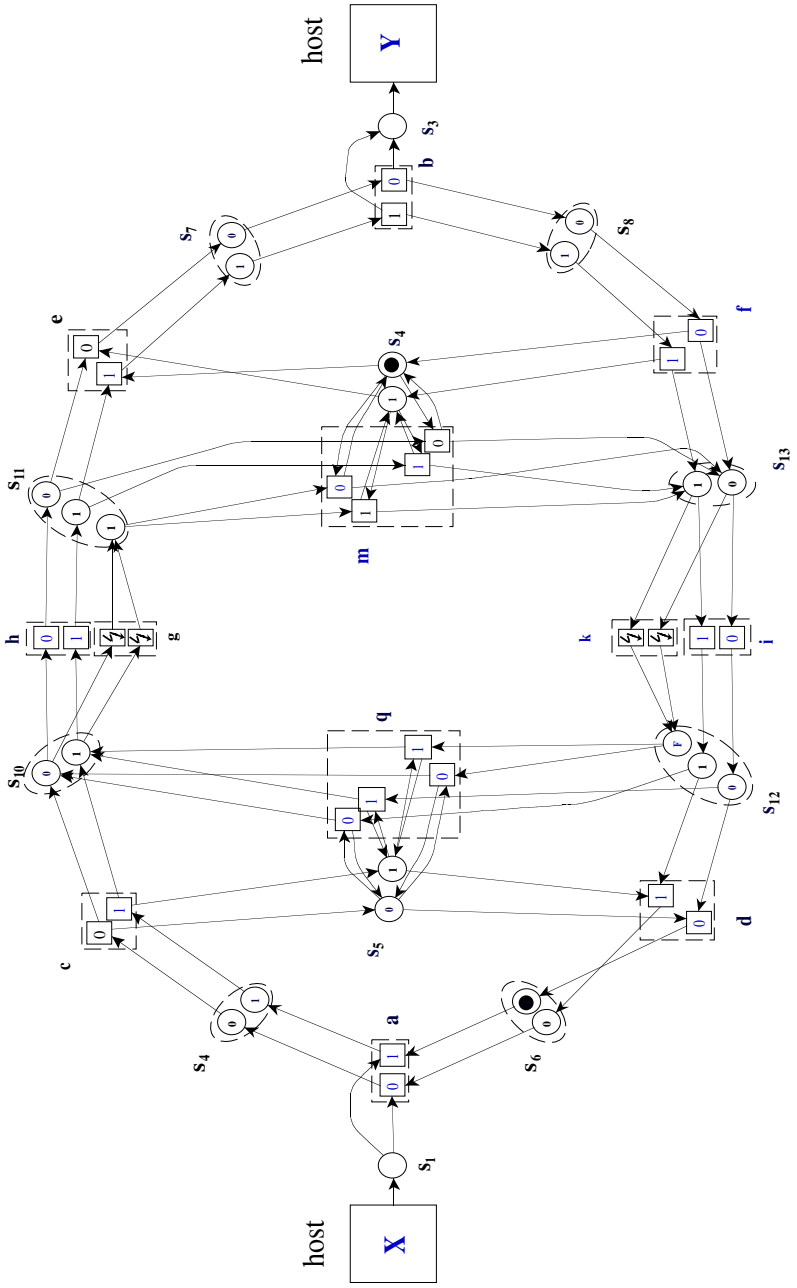


Abbildung 3.34: Auffaltung des Alternierbitprotokolls / 2.Schritt

Kapitel 4

Elementare Konsistenzeigenschaften nebenläufiger Systeme

In diesem Kapitel werden zwei Klassen von Korrektheits- bzw. Konsistenz-Begriffen behandelt: Ablaufkonsistenz, die sich auf den korrekten Steuerfluss nebenläufiger Systeme bezieht und Datenkonsistenz, die die Integrität von Daten bei nebenläufigem Zugriff betrifft.

4.1 Ablaufkonsistenz

4.1.1 Netzinvarianten

Beziehungen zwischen Programmvariablen, die bei der Ausführung eines Programmes erhalten bleiben, heißen Invarianten. Ihre Nützlichkeit zum Nachweis von Programmeigenschaften ist aus der sequentiellen Programmierung bereits hinreichend bekannt. Da das Verhalten von nebenläufigen Programmen weitaus komplexer und unübersichtlicher ist, sind Invarianten von entsprechend größerer Bedeutung, vorausgesetzt, es gelingt solche Gesetzmäßigkeiten zu erkennen. Eine Besonderheit von P-Invariante bzw. S-Invarianten liegt darin, dass sie berechenbar sind (im Gegensatz zu allgemeinen Programminvarianten). Die folgenden Abschnitte stammen aus [JV87]. Dort werden Plätze wie in vielen deutschsprachigen Büchern als *Stellen* bezeichnet. Entsprechend werden die Buchstaben s und S benutzt. P/T-Netze heißen *S/T-Netze*.

Als (weiteres) Beispiel betrachten wir das Leser/Schreiber-Problem. Die folgende Abb. 4.1 zeigt ein entsprechendes Modell als S/T-Netz (oder P/T-Netz) für n Aufträge ($n \in \mathbb{N}, n > 0$), die Anfangs in der Stelle (dem Platz) *lok* liegen, was ausdrückt, dass sie noch nichts mit dem kritischen Abschnitt zu tun haben, auf den sie später lesend oder schreibend zugreifen. Durch das Schalten der Transition a bzw. d melden sie sich als Lese- bzw. Schreibaufträge an, d.h. die entsprechende Marke liegt in *la* bzw. *sa*, was "zum Lesen angemeldet" bzw. "zum Schreiben angemeldet" heißen soll. Dann folgt der Zugriff auf die kritischen Daten mit den bekannten Spezifikationen:

Spezifikation 4.1 *a) Wenn ein Schreiber schreibt, darf kein Leser lesen*

b) Es darf höchstens ein Schreiber schreiben

c) Wenn ein Leser liest, darf kein Schreiber schreiben

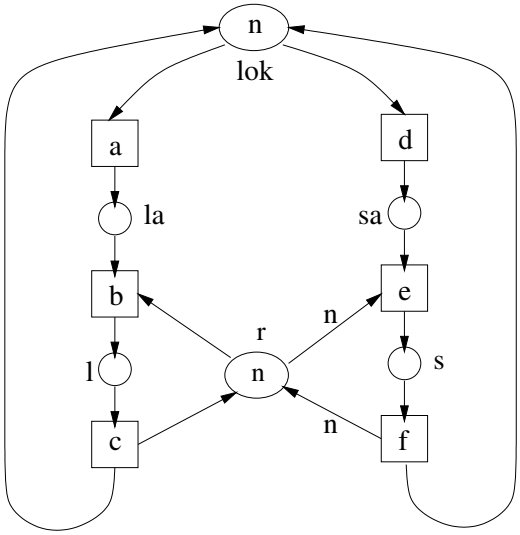


Abbildung 4.1: Leser/Schreiber-Problem für n Aufträge

Für alle erreichbaren Markierungen $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ soll also gelten:

a) $\mathbf{m}(s) > 0 \rightarrow \mathbf{m}(l) = 0$

b) $\mathbf{m}(s) \leq 1$

c) $\mathbf{m}(l) > 0 \rightarrow \mathbf{m}(s) = 0$

Wir werden die Gültigkeit dieser Bedingungen im nächsten Abschnitt beweisen.

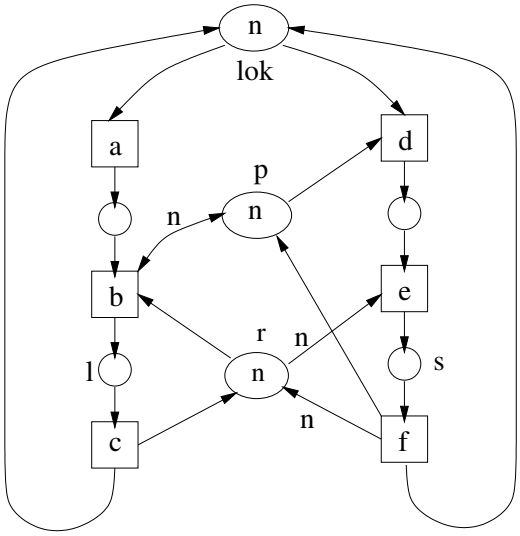


Abbildung 4.2: Priorität für Schreibaufträge

Es gibt verschiedene Varianten des Leser/Schreiber-Problems. Um eine möglichst schnelle Aktualisierung der Daten zu gewährleisten, kann man z.B. den Schreiber priorisierten Zugriff einräumen. Abbildung 4.2 zeigt die entsprechende Erweiterung: sobald mindestens ein Schreiber angemeldet ist oder schreibt, darf kein neuer Leser anfangen zu lesen (denn b ist wegen $m(p) < n$ gesperrt).

Für das S/T-Netz des Leser/Schreiber-Problems in Abb. 4.1 gelten die Invarianten ($n \geq 1$)

- i_1 : $\quad lok + la + sa + l + s = n$
- i_2 : $\quad l + r + n \cdot s = n$
  ($lok, la, \ldots$ stehen hier abkürzend für $\mathbf{m}(lok), \mathbf{m}(la), \ldots$)

Der Beweis, dass diese Invarianten für alle erreichbaren Markierungen gelten, stellen wir zurück, leiten aber aus ihnen die Synchronisationsbedingungen 4.1 ab.

a) $s > 0 \rightarrow l = 0$ (folgt aus i_2)

b) $s \leq 1$ (wegen i_2)

c) $l > 0 \rightarrow s = 0$ (wegen i_2)
(wieder steht $s > 0$ für $\mathbf{m}(s) > 0$ usw.)

Damit ist nachgewiesen, dass das Netz von Abb. 4.1 die Synchronisationsbedingungen des Leser/Schreiber-Problems erfüllen. (Wir müssen nur noch den Beweis der Gültigkeit von i_1 und i_2 nachholen.)

Wir zeigen nun, wie man die Gültigkeit der Invarianten nachweist. Dazu betrachten wir das S/T-Netz von Abb. 4.1 und die Invarianten i_2:

$$\mathbf{m}(l) + \mathbf{m}(r) + n \cdot \mathbf{m}(s) = n \tag{4.1}$$

Sie gilt für die Anfangsmarkierung $\mathbf{m}_0$

Als Induktionsbeweis zeigt man dann:

Beweis:
Gilt 4.1 für eine Markierung $\mathbf{m}_1 \in \mathbf{R}(\mathcal{N})$ und ändert eine Transition t die Markierung $\mathbf{m}_1$ durch Schalten zu $\mathbf{m}_2$ (also $\mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2$), dann gilt 4.1 auch für $\mathbf{m}_2$.

Für $t = e$ gilt

$$\begin{aligned} \mathbf{m}_2(sa) &= \mathbf{m}_1(sa) - 1 \\ \mathbf{m}_2(r) &= \mathbf{m}_1(r) - n \\ \mathbf{m}_2(s) &= \mathbf{m}_1(s) + 1, \end{aligned} \tag{4.2}$$

während alle anderen Stellen unverändert bleiben. Gilt 4.1 für $\mathbf{m}_1$, dann auch für $\mathbf{m}_2$. Dies ist für alle Transitionen durchzuführen. □

Um dies systematischer zu behandeln, definieren wir die *"Wirkung"* einer Transition.

Definition 4.2 *Es sei $\mathcal{N} = (S, T, F, W, \mathbf{m}_0)$ ein S/T-Netz mit $S = \{s_1, \ldots, s_p\}$ und $T = \{t_1, \ldots, t_q\}$. Der Vektor $\Delta_{\mathcal{N}}(t) \in \mathbb{Z}^p$ heißt* Wirkung *der Transition $t \in T$ und ist definiert durch*

$$\Delta_{\mathcal{N}}(t)(s) = -W(s,t) + W(t,s)$$

Die durch Aneinanderreihung der Vektoren $\Delta_{\mathcal{N}}(t_1) \ldots \Delta_{\mathcal{N}}(t_q)$ gebildete $(p \times q)$-Matrix $\Delta_{\mathcal{N}}$ heißt Wirkungs- oder Inzidenzmatrix. $\Delta_{\mathcal{N}}(t)$ ist dann die t-Spalte von $\Delta_{\mathcal{N}}$.

Der Name *Inzidenzmatrix* ist durch die Darstellung von $\mathcal{N}$ als Graph zu erklären, während die Bezeichnung *Wirkung* durch den folgenden Satz deutlich wird.

Satz 4.3 *Wenn t die Markierung $\mathbf{m}_1$ durch Schalten in $\mathbf{m}_2$ überführt ($\mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2$), dann gilt:*

$$\mathbf{m}_2 = \mathbf{m}_1 + \Delta_{\mathcal{N}}(t)$$

Der Beweis folgt direkt durch Vergleich der Definitionen 3.9 und 4.2.

Von Invarianten bei Programmen ist bekannt, dass sie nicht algorithmisch aus dem Programm gewonnen werden können. Einer der Vorteile der Darstellung von Synchronisations-Problemen durch S/T-Netze beruht darauf, dass Netz-Invarianten berechnet werden können. Die Grundlage dazu liefert der folgende *Satz von Lautenbach*.

Satz 4.4 *(Lautenbach)*
Es sei M' die Transponierte einer Matrix M und $\underline{0} \in \mathbb{Z}^{|T|}$ der Nullvektor. Ist $\mathcal{N} = (S, T, F, W, \mathbf{m}_0)$ ein S/T-Netz mit Inzidenzmatrix $\Delta_{\mathcal{N}}$ und $i \in \mathbb{Z}^{|S|}$ eine ganzzahlige Lösung des linearen Gleichungssystems

$$\Delta'_{\mathcal{N}} \cdot i = \underline{0}$$

dann gilt $i' \cdot \mathbf{m} = i' \cdot \mathbf{m}_0$ für alle erreichbaren Markierungen $\mathbf{m} \in \mathbf{R}(\mathcal{N})$.

Beweis:
(durch Indukion über $\mathbf{R}(\mathcal{N})$):
Die Behauptung ist trivial für $\mathbf{m} = \mathbf{m}_0$.
Es gelte die Behauptung für $\mathbf{m}_1 \in \mathbf{R}(\mathcal{N})$, also $i' \cdot \mathbf{m}_1 = i' \cdot \mathbf{m}_0$, und es gelte $\mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2$ für eine Transition $t \in T$. Aus der Voraussetzung $\Delta'_{\mathcal{N}} \cdot i' = \underline{0}$ folgt dann $\underline{0}' = (\Delta'_{\mathcal{N}} \cdot i)' = i' \cdot (\Delta'_{\mathcal{N}})' = i' \cdot \Delta_{\mathcal{N}}$ und damit $i' \cdot \Delta_{\mathcal{N}}(t) = 0$. Also gilt mit Satz 4.3 die Induktionsbehauptung:

$$\begin{aligned} i' \cdot \mathbf{m}_2 &= i' \cdot (\mathbf{m}_1 + \Delta_{\mathcal{N}}(t)) \\ &= i' \cdot \mathbf{m}_1 + i' \cdot \Delta_{\mathcal{N}}(t) \\ &= i' \cdot \mathbf{m}_1 \\ &= i' \cdot \mathbf{m}_0 \end{aligned}$$

□

Definition 4.5 *Jede ganzzahlige Löung $i \in \mathbb{Z}^{|S|} \setminus \{0\}$ von $\Delta'_{\mathcal{N}} \cdot i = \underline{0}$ heißt S-Invarianten-Vektor des S/T-Netzes $\mathcal{N}$.*

Wir interpretieren Satz 4.4 anhand unseres Beispiels. Abbildung 4.3 zeigt die Inzidenzmatrix $\Delta_{\mathcal{N}}$ des Netzes von Abb. 4.1 und zwei Invarianten-Vektoren i_1 und i_2. (Man rechne nach: $\Delta'_{\mathcal{N}} \cdot i_k = \underline{0}$.)

$\mathcal{N}$	a	b	c	d	e	f		i_1		i_2
lok	-1	0	1	-1	0	1		1		0
la	1	-1	0	0	0	0		1		0
sa	0	0	0	1	-1	0		1		0
l	0	1	-1	0	0	0		1		1
s	0	0	0	0	1	-1		1		n
r	0	-1	1	0	$-n$	n		0		1

j	3	3	3	2	2	2

Abbildung 4.3: Inzidenzmatrix $\Delta_{\mathcal{N}}$ mit S-Invarianten i_1, i_2 und T-Invariante j

Folglich gilt nach Satz 4.4 für jede von der Anfangsmarkierung $\mathbf{m}'_0 = (n, 0, 0, 0, 0, n)$ aus erreichbare Markierung $\mathbf{m}$:

$$\begin{aligned} i'_2 \cdot \mathbf{m} &= 1 \cdot \mathbf{m}(l) + n \cdot \mathbf{m}(s) + 1 \cdot \mathbf{m}(r) = \\ i'_2 \cdot \mathbf{m}_0 &= 1 \cdot \mathbf{m}_0(l) + n \cdot \mathbf{m}_0(s) + 1 \cdot \mathbf{m}_0(r) \\ &= 1 \cdot 0 + n \cdot 0 + 1 \cdot n = n \end{aligned}$$

Dies ist genau die Invariantengleichung 4.1

Wir fassen zusammen:

Aus einem gegebenen S/T-Netz $\mathcal{N}$ können durch Berechnung aller ganzzahligen Lösungen i in $\Delta'_{\mathcal{N}} \cdot i = 0$ alle Invarianten-Vektoren gefunden werden. Für die Anfangsmarkierung $\mathbf{m}_0$ werden alle durch $i' \cdot \mathbf{m} = i' \cdot \mathbf{m}_0$ die entsprechenden Invariantengleichungen aufgestellt, die für alle $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ gelten. Mit ihnen können, wie oben gezeigt, Netzeigenschaften nachgewiesen werden.

Auch die Lösungen j von $\Delta_{\mathcal{N}} \cdot j = \underline{0}$ haben eine wichtige Interpretation für die Analyse von Rechensystemen.

Definition 4.6 *Jede Lösung* $j \in \mathbb{N}^{|T|} \setminus \{\underline{0}\}$ *von* $\Delta_{\mathcal{N}} \cdot j = 0$ *heißt* T-Invarianten-Vektor *des S/T-Netzes* $\mathcal{N}$.

T-Invarianten liefern notwendige Bedingungen für die Reproduzierbarkeit von Systemen. Dabei heiße eine Markierung $\mathbf{m}$ *reproduzierbar*, wenn sie durch eine (nicht leere) Schaltfolge w von Transitionen wieder erreichbar ist. Kommt eine Transition $t_i \in T$ in w gerade $\#_i(w)$ mal vor, dann ist der Vektor $\#(w) := (\#_1(w), \#_2(w), \ldots, \#_{|T|}(w))$ ein T-Invarianten-Vektor. T-Invarianten beschreiben also die Häufigkeit des Schaltens jeder Transition bei reproduzierendem Verhalten.

Satz 4.7 *Es seien* $\mathbf{m}_1, \mathbf{m}_2$ *Markierungen und* $w = t_{i_1} \ldots t_{i_k}$ *eine Schaltfolge, die* $\mathbf{m}_1$ *in* $\mathbf{m}_2$ *überführt:* $\mathbf{m}_1 \xrightarrow{w} \mathbf{m}_2$. *Die Markierungen* $\mathbf{m}_1$ *und* $\mathbf{m}_2$ *sind genau dann gleich, wenn es einen T-Invarianten-Vektor* $j \in \mathbb{N}^{|T|}$ *derart gibt, dass jede Transition* $t_i \in T$ *genau* $j(i)$ *mal in* w *vorkommt, d.h.:* $j(i) = \#_i(w)$

Beweis:
Es gilt nach Vorraussetzung $\mathbf{m}_1 = \mathbf{m}_2 = \mathbf{m}_1 + \Delta_{\mathcal{N}}(t_{i_1}) + \Delta_{\mathcal{N}}(t_{i_2}) + \ldots + \Delta_{\mathcal{N}}(t_{i_k})$ genau dann, wenn gilt: $\underline{0} = \Delta_{\mathcal{N}}(t_{i_1}) + \Delta_{\mathcal{N}}(t_{i_2}) + \ldots + \Delta_{\mathcal{N}}(t_{i_k}) = \Delta_{\mathcal{N}} \cdot \#(w)$. □

Der T-Invarianten-Vektor j in Abb. 4.3 besagt also, dass die Anfangsmarkierungen $\mathbf{m}_0$ dann wieder erreicht (reproduziert) wird, wenn drei Lese- und zwei Schreibaufträge ihren Zyklus durchlaufen haben.

4.1.2 Beschränktheit, Lebendigkeit und Fairness

Eine wichtige Eigenschaft von P/T-Netzen (und auch von gefärbten Netzen) ist die Eigenschaft der „Beschränktheit":

Definition 4.8 *Sei* $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ *ein P/T-Netz (siehe Def. 3.8) und* $p \in P$ *ein Platz.*

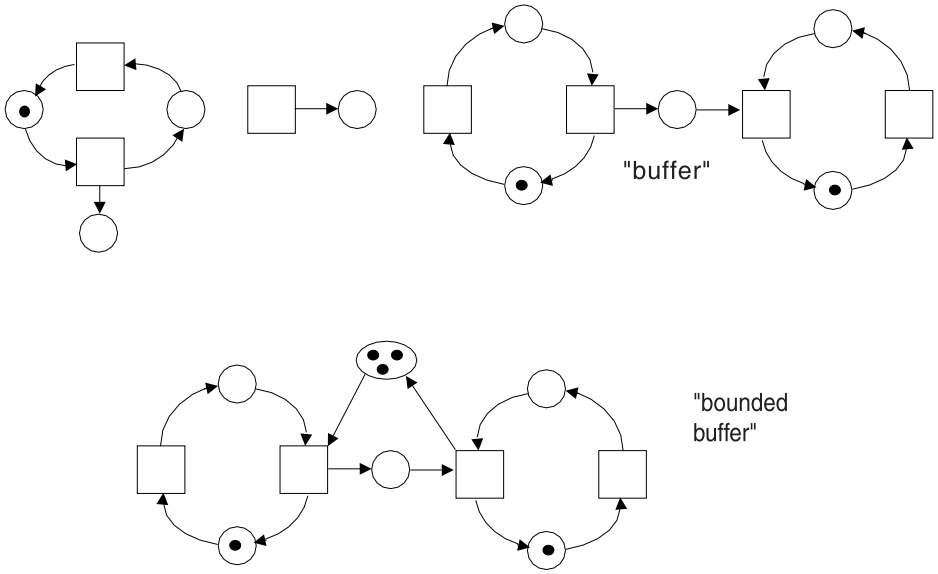


Abbildung 4.4: Beispiele für nicht beschränkte Netze (oben) und ein beschränktes Netz

a) p heißt beschränkt, falls $\exists k\ \forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) : \mathbf{m}(p) \leq k$

b) $\mathcal{N}$ heißt beschränkt, falls alle Plätze beschränkt sind.

Ein P/T-Netz $\mathcal{N}$ ist genau dann beschränkt, wenn seine Erreichbarkeitsmenge $\mathbf{R}(\mathcal{N})$ bzw. sein Erreichbarkeitsgaph $RG(\mathcal{N})$ (siehe Def. 3.10) endlich ist.

Oft ist es sinnvoll, Prozesse als unendlich zu betrachten, obwohl solche natürlich nie zu beobachten sind. Dies wurde schon im Zusammenhang mit dem Bankiersproblem deutlich. Im Rahmen der Folgensemantik werden sie als unendliche Folgen dargestellt.

Definition 4.9 *Es sei $\mathcal{N} = (S, T, F, W, \mathbf{m}_0)$ ein S/T-Netz und $\mathbf{m}_0$ eine Markierung. T^ω bezeichnet die Menge aller unendlichen Folgen von Elementen aus T. Für eine solche unendliche Folge $w \in T^\omega$, die auch ω-Folge heißt, und $n \in \mathbb{N} \setminus \{0\}$ sei $w(n) \in T$ das n-te Folgenelement und $w[n] \in T^*$ das Anfangsstück $w(1)w(2)\ldots w(n)$ von w der Länge n. Eine unendliche Folge von Transitionen $w \in T^\omega$ heißt aktiviert in $\mathbf{m}$, symbolisch $\mathbf{m}\xrightarrow{w}$, falls alle ihre Anfangsstücke aktiviert sind: $\forall n \geq 1 : \mathbf{m}\xrightarrow{w[n]}$. $F_\omega(\mathcal{N}) := \{w \in T^\omega | \mathbf{m}_0 \xrightarrow{w}\}$ ist die Menge der unendlichen Schaltfolgen von $\mathcal{N}$.*

Definition 4.10 *Es sei $\mathcal{N} = (S, T, F, W, \mathbf{m}_0)$ ein S/T-Netz und $\mathbf{m} \in M$ eine Markierung von $\mathcal{N}$.*

a) $\mathbf{m}$ *heißt* Verklemmung *(deadlock), falls keine Transition (schalten kann) aktiviert ist:*

$$\neg\exists t \in T : m \xrightarrow{t}$$

$\mathcal{N}$ *heißt* verklemmungsfrei*, falls keine erreichbare Markierung* $m \in \mathbf{R}(\mathcal{N})$ *eine Verklemmung ist.*

b) Ist für $\mathcal{N}$ *eine Menge* $M_E := \{\mathbf{m}_1, \ldots, \mathbf{m}_k\}$ *von terminalen Markierungen gegeben, dann heißt* $\mathbf{m}$ sicher oder fortsetzbar*, falls*

$$\exists w \in T^* \exists \mathbf{m}_i \in M_E : \mathbf{m} \xrightarrow{w} \mathbf{m}_i$$

c) $\mathbf{m}$ *heißt* ω*-sicher oder* ω*-fortsetzbar, falls*

$$\exists w \in T^\omega : \mathbf{m} \xrightarrow{w}$$

SICH($\mathcal{N}$*) oder SICH bezeichne die Menge aller* ω*-sicheren Markierungen von* $\mathcal{N}$*.*

Anmerkung: Abweichend von Def. 4.10 heißt eine Markierung oft sicher, wenn jede Stelle höchstens eine Marke enthält.

Beispiel 4.11 Wir betrachten das (vereinfachte) Bankiersproblem aus Abb.3.26. Von der Markierung $\mathbf{m}$ mit $\mathbf{m}(CLAIM_1) = 4$, $\mathbf{m}(CLAIM_2) = 3$, $\mathbf{m}(CLAIM_3) = 6$ kann die Schaltfolge $GRANT_2\ GRANT_2\ GRANT_2\ RETURN_2$ schalten und erreicht wieder die Markierung $\mathbf{m}$. Obwohl also in $\mathbf{m}$ eine unendliche Schaltfolge möglich ist, können die Kunden 1 und 3 nie ihr Kreditgeschäft erledigen. Bei dieser Schaltfolge befinden sich nämlich höchstens 3 Marken in $BANK$ was weder für die Restforderung 4 von Kunde 1 noch für die Restforderung 6 von Kunde 3 ausreicht. In $\mathbf{m}$ liegt also keine totale, wohl aber eine *partielle Verklemmung* vor: man sagt "$\mathcal{N}$ ist nicht *lebendig*".

Definition 4.12 *Es sei* $\mathcal{N} = (S, T, F, W, \mathbf{m}_0)$ *ein S/T-Netz und* $\mathbf{m} \in M_S$ *eine Markierung.*

a) $\mathbf{m}$ *heißt* lebendig*, falls* $\forall t \in T \forall u \in T^* \exists v \in T^* : \mathbf{m} \xrightarrow{u} \Longrightarrow \mathbf{m} \xrightarrow{uvt}$

b) $\mathcal{N}$ *heißt* lebendig*, falls* $\mathbf{m}_0$ *lebendig ist.*

c) $\mathbf{m}$ *heißt* T-fortsetzbar, *falls* $\exists w \in T^\omega : \mathbf{m} \xrightarrow{w}$ *und alle* $t \in E$ *in diesem* w *unendlich oft vorkommen.*

Ein S/T-Netz ist lebendig, wenn nach einer beliebigen Schaltfolge jede Transition wieder zum Schalten gebracht werden kann. Eine Markierung $\mathbf{m}$ ist T-fortsetzbar, wenn von $\mathbf{m}$ aus so geschaltet werden kann, dass *jede* Transition immer wieder schaltet.

Abbildung 4.5 a) zeigt ein nichtlebendiges S/T-Netz, das durch Einfügen eines "Regulationskreises" lebendig gemacht wurde (Abb. 4.5 b)). Das letztere Netz zeigt auch, dass die Menge $LEB(\mathcal{N})$ der lebendigen Markierungen eines S/T-Netzes nicht abgeschlossen[1] ist. Fügt man nämlich eine Marke zu s_4 hinzu, erhält man eine größere Markierung $\mathbf{m} > \mathbf{m}_0$, die nicht mehr lebendig ist. Andererseits ist die Menge $CONT(T)$ der T-fortsetzbaren Markierungen abgeschlossen. Zwischen lebendigen und T-fortsetzbaren Markierungen besteht folgender Zusammenhang.

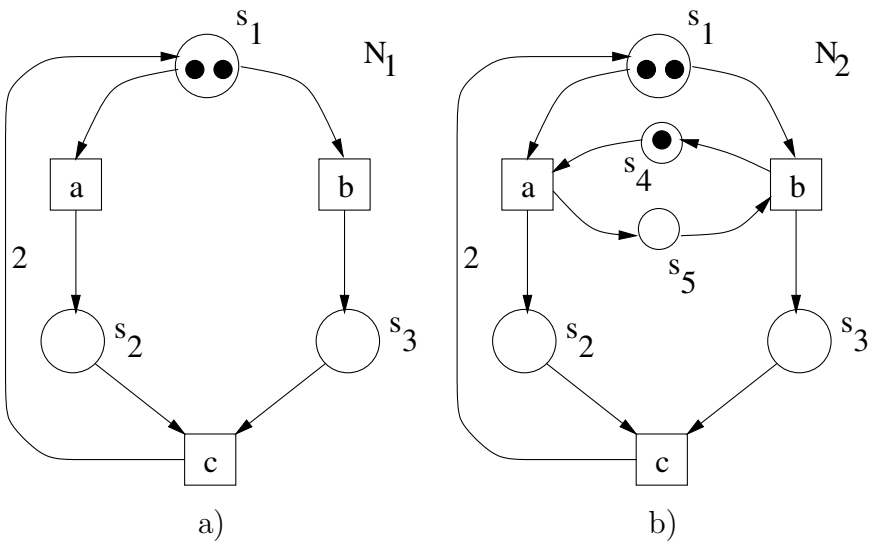


Abbildung 4.5: Nichtlebendiges S/T-Netz $\mathcal{N}_1$ und lebendiges S/T-Netz $\mathcal{N}_2$

Satz 4.13 *In einem S/T-Netz* $\mathcal{N} = (S, T, F, W, \mathbf{m}_0)$ *ist eine Markierung* $\mathbf{m}$ *genau dann lebendig, wenn alle von* $\mathbf{m}$ *aus erreichbaren Markierung* $\mathbf{m}'$ T*-fortsetzbar sind*

[1] d.h: vergrößert man die Markierung, dann bleibt die Eigenschaft nicht (bzw. doch) erhalten

Beweis:
Jede von eine lebendigen Markierung $\mathbf{m}$ *aus erreichbare Markierung* $\mathbf{m}'$ *ist wieder lebendig und daher auch* T*-fortsetzbar. Gelte nun umgekehrt* $\mathbf{m}' \in CONT(T)$ *für alle von* $\mathbf{m}$ *aus erreichbaren Markierungen* $\mathbf{m}'$*. Sei* $t \in T$*,* $u \in T^*$ *mit* $\mathbf{m} \xrightarrow{u} \mathbf{m}'$*. Dann gibt es* $w \in T^\omega$ *mit* $\mathbf{m}' \xrightarrow{w}$ *und* t *kommt unendlich oft in* w *vor. Also gibt es eine endliche Anfangsfolge* v *von* w *mit* $\mathbf{m} \xrightarrow{uvt}$*, d.h.* $\mathbf{m}$ *ist lebendig* □

Zum Begriff der Fairness betrachten wir das Problem der fünf Philosophen [Dij75], mit dem ein Betriebsmittelzuteilungsproblem besonderer Art beschrieben wird.

Fünf Philosophen $Ph_1, \ldots, Ph_5$ sitzen an einem runden Tisch, in dessen Mitte eine Schüssel mit Spaghetti steht (Abb. 4.6). Jeder Philosoph Ph_i befindet sich entweder im Zustand des "Denkens" (d_i) oder "Essens" (e_i). Zum Essen stehen insgesamt nur 5 Gabeln $g_1, \ldots, g_5$ (die Betriebsmittel) zur Verfügung, jeweils eine zwischen zwei benachbarten Philosophen. Geht ein Philosoph vom Denken zum Essen über, nimmt er erst die rechte und dann die linke Gabel auf.

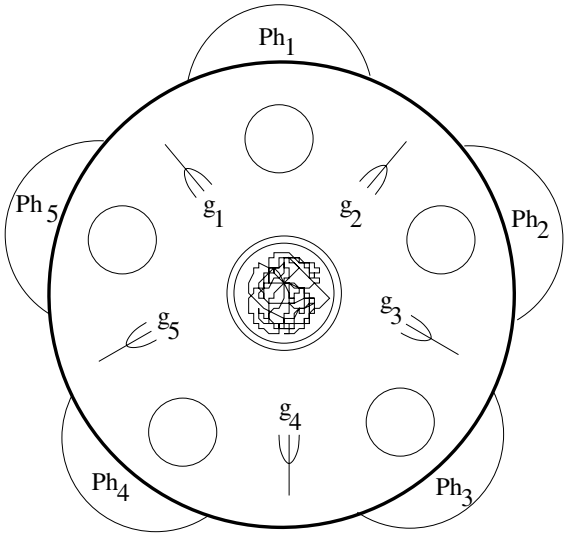


Abbildung 4.6: Die fünf Philosophen am Tisch (nach Dijkstra)

Abbildung 4.7 zeigt eine S/T-Netz-Dartstellung des Problems. Das Netz ist natürlich nicht lebendig: wenn alle fünf Philosophen ihre rechte Gabel nehmen, entsteht eine Verklemmung. Um dies zu verhindern, kann man die beiden Transitionen, die das Aufnehmen der rechten und linken Gabel darstellen, unteilbar machen, also zu der in Abb. 4.7

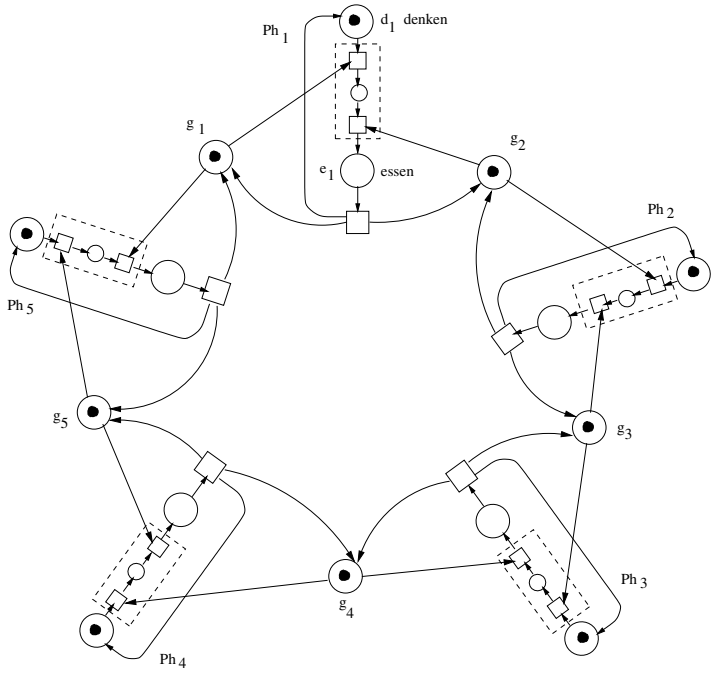


Abbildung 4.7: Die fünf Philosophen als S/T-Netz

dargestellten Vergröberung übergehen. Nun ist das Netz zwar lebendig, aber es besteht immer noch die Möglichkeit, dass zwei Philosophen, etwa Ph_1 und Ph_3 so die Gabel benutzen, dass Ph_2 nie die Chance hat, seine Gabeln aufzunehmen. Man sagt, für den Philosophen Ph_2 besteht die Gefahr des Verhungerns (*starvation*), oder die Philosophen Ph_1 und Ph_3 verhalten sich unfair gegenüber Ph_2.

Definition 4.14 *Ein S/T-Netz* $\mathcal{N} = (S, T, F, W, \mathbf{m}_0)$ *hat ein* faires Verhalten*, oder verhält sich* fair *(behaves fairly), wenn in jeder unendlichen Schaltfolge* $w \in F_\omega(\mathcal{N})$ *jeder Transition* $t \in T$ *unendlich oft vorkommt.*

Wir vergleichen die wichtigen Begriffe der Lebendigkeit und Fairneß.
a) Lebendigkeit bedeutet Freiheit von *unvermeidbaren* partiellen Verklemmungen.
b) Fairneß bedeutet Freiheit von *faktischen* partiellen Verklemmungen.
Der Unterschied zwischen lebendigen und fairem Verhalten ist also gekennzeichnet durch den Existenzquantor in a) (alle Transitionen *können* immer wieder schalten) und dem Allquantor in b) (alle Transtionen *müssen* immer wieder schalten)

Außer in einfachen Fällen hat ein System oder Netz kein faires Verhalten. In dem Netz von Abb.4.8 kann natürlich die faire Folge

$$acbd \quad acbd \quad \ldots$$

wie die unfaire

$$ac \quad ac \quad ac \quad ac \quad \ldots$$

schalten. Dieses Netz entspricht in gewisser Weise dem Programm

$$a, b := true;$$
$$\underline{do}\ a \to c \ \square\ b \to d\ \underline{od}$$

worin c und d Anweisungen sind, die a und b nicht verändern[2].

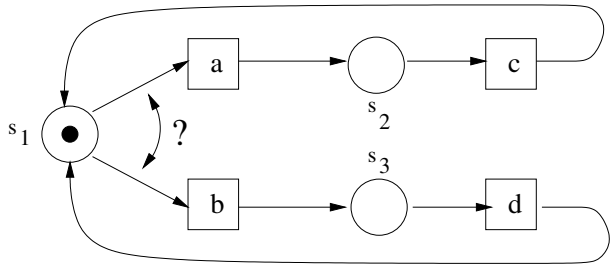


Abbildung 4.8: Netz mit unfairem Verhalten

Sowohl für die Netze wie für Programme hat man daher Formen des fairen Schaltens bzw. Programmablauf vorgeschlagen. Das Problem wird bei der Definition von Programmiersprachen heute jedoch meist noch dem Implementator zugeschrieben.

Definition 4.15 *Es sei* $\mathcal{N} = (S, T, F, W, \mathbf{m}_0)$ *ein S/T-Netz.*

a) $\mathcal{N}$ *schaltet* produktiv *oder* verschleppungsfrei *oder* nach der verschleppungsfreien Schaltregel *(finite delay property), wenn*

$$\forall t \in T \forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) \neg \exists w \in T^\omega : \mathbf{m} \xrightarrow{w} \wedge\ t \textit{ ist bei } w \textit{ permanent aktiviert } \wedge |w|_t = 0$$

[2]Zur Erklärung dieser Anweisungen siehe die Sprache *Prog* in Definition 5.1.

b) $\mathcal{N}$ *schaltet* fair*, oder* nach der fairen Schaltregel *(fair firing rule), wenn*

$$\forall t \in T \forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) \neg \exists w \in T^{\omega} : \mathbf{m} \xrightarrow{w} \wedge\ t \textit{ ist bei } w \textit{ unendlich oft aktiviert} \wedge |w|_t = 0$$

Ein Netz schaltet also nicht verschleppungsfrei (bzw. fair), wenn von einer erreichten Markierung $\mathbf{m}$ ab eine unendliche Schaltfolge w schaltet, bei der eine Transition zwar permanent, d.h. in allen durchlaufenen Markierungen (bzw. unendlich oft) aktiviert ist, aber nie schaltet.

Zu implementieren wäre diese Eigenschaft etwa durch Zähler, die über die Aktiviertheit von Transitionen Buch führen. Ab einer festgelegten Größe des Zählers würde dieser Transition dann Priorität eingeräumt.

Die verschleppungsfreie Schaltregel ist die elementarere. Sie sorgt z.B. dafür, dass unabhängige nebenläufige Anweisungen nach endlicher Zeit ausgeführt werden. Die faire Schaltregel wird oft in der Semantik nichtdeterministischer oder nebenläufiger Programme aufgenommen. Das folgende Programm[3] terminiert z.B. zwingend nur unter der fairen Schaltregel.

$$b := true;$$
$$\underline{do}\ b \rightarrow x := 1 \ \square\ b \rightarrow b := false\ \underline{od}$$

Anhand der vorstehenden Beispiele kann man die Beziehung zwischen verschleppungsfreiem bzw. fairem Schalten und lebendigem bzw. fairem Verhalten studieren.

Beispiel 4.16

a) Das Netz von Abb.4.8 ist zwar lebendig, hat aber auch unter der verschleppungsfreien Schaltregel kein faires Verhalten ($ac\ ac\ \ldots$ ist immer noch möglich). Das Netz verhält sich jedoch fair bei der fairen Schaltregel.

b) Das Netz von Abb.4.7 mit der vergröberten, und daher unteilbaren Transition ist lebendig. Es hat aber weder unter der verschleppungsfreien noch unter der fairen Schaltregel ein faires Verhalten.

[3]Zur Erklärung dieser Anweisungen siehe die Sprache *Prog* in Definition 5.1.

c) Das Netz von Abb.4.7 ohne Vergröberung ist nicht lebendig. Unter der fairen Schaltregel hat es jedoch ein faires Verhalten. Verhindert man die Verklemmung durch andere Maßnahmen, z.B. indem man höchstens 4 Philosophen in den Eßraum läßt (Abb. 4.9), dann ist das Netz lebendig und fair bei der fairen Schaltregel.

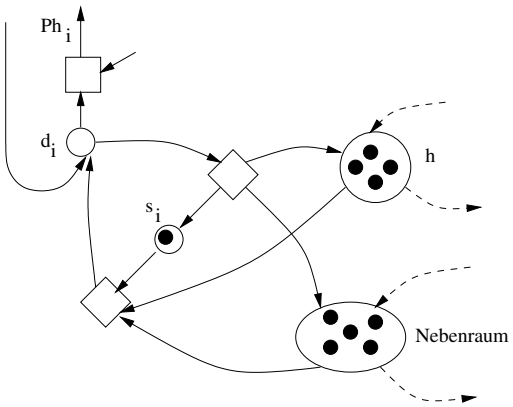


Abbildung 4.9: Philosophenproblem mit Nebenraum

In der dargestellten Anfangsmarkierung befinden sich alle Philosophen im Nebenraum. Die Stelle s_i bewirkt, dass höchstens eine Marke nach d_i und die Stelle h, dass höchstens 4 Marken nach d_1 bis d_5 gelangen.

4.1.3 Anwendung der Begriffe: Analyse von Workflow-Systemen

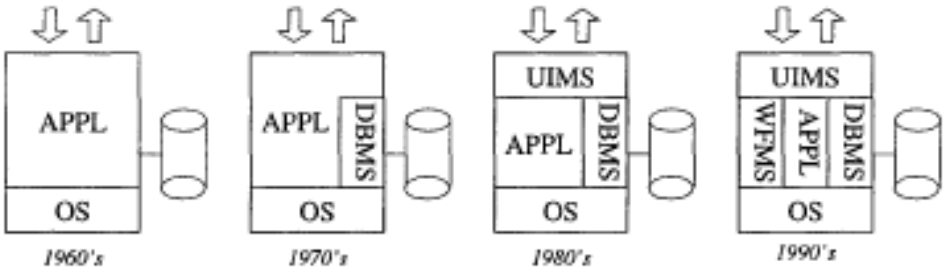


Abbildung 4.10: Workflow-Management-Systeme in historischer Perspective

Workflow-Systeme steuern und verwalten den Ablauf von Workflow-Prozessen (zu deutsch etwa "Geschäftsprozesse").

In den 60-er Jahren bestanden Informationssysteme aus einer Anzahl von Anwenderprogrammen (application systems, APPL), die direkt auf das Betriebssystem (operating system, OS) aufsetzten (vergl. Abb. 4.10). Für jedes dieser Programme wurde eine eigene Benutzerschnittstelle und ein eigenes Datenbanksystem entwickelt. In den 70-er Jahren wurde die langfristige Datenhaltung aus den Applikationsprogrammen ausgelagert. Zu diesem Zweck wurden Datenbanksysteme (database management systems, DBMSs) entwickelt. In den 80-er Jahren fand mit den Benutzeroberflächen eine ähnliche Entwicklung statt. Benutzerschnittstellensysteme (user interface management systems, UIMSs) erlaubten es, die Kommunikation mit dem Benutzer aus dem Anwendungsprogramm auszulagern. Eine ähnliche Entwicklung setzte in den 90-er Jahren mit der Entwicklung von Workflow-Programmen (workflow management systems, WFMSs) ein. Sie erlauben es, die Verwaltung von Geschäftsprozessen aus spezifischen Anwendungsprogrammen herauszuhalten [Aal00], [Aal98].

Die Aufgabe von Workflow-Systemen (manchmal auch *office logistics* genannt) ist es sicherzustellen, dass die richtigen Aktionen eines Geschäftsablaufes durch die richtigen Personen zur richtigen Zeit ausgeführt werden. Abstrakter kann man ein Workflow-System als eine Software definieren, die Geschäftsabläufe vollständig modelliert, verwaltet und steuert.

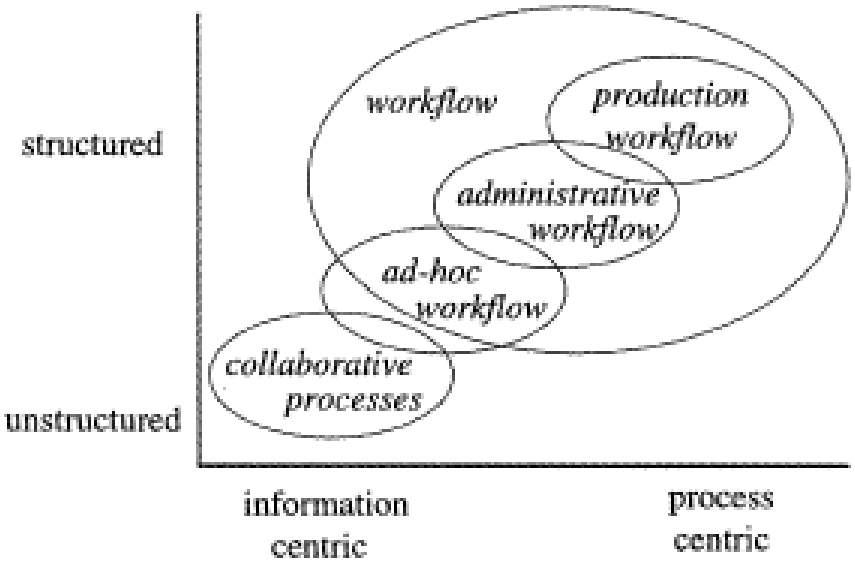


Abbildung 4.11: Workflow Prozesse und Kooperative Prozesse im Vergleich

Der Begriff Workflow-System wird häufig nicht so spezifisch gebraucht. Beispielweise wird so auch Software zur Gruppenkommunikation genannt, die lediglich die Versendung von Nachrichten und den Austausch von Information unterstützt (wie z.B. Lotus Notes, Microsoft Exchange). Die Abbildung 4.11 setzt allgemeine kooperative Prozesse und Workflow-Systeme in eine Skalierung zwischen informationszentriert und prozesszentriert bzw. weniger und mehr strukturiert.

Für Workflow-Systeme gibt es hunderte von Modellierungsansätze und Rechnerwerkzeuge, die jedoch häufig keine wohldefinierte und eindeutige Semantik besitzen. Wegen ihrer Orientierung auf Aktionen und deren Koordination bieten sich natürlich besonders Petrinetze an, deren Semantik wohldefiniert und bekannt ist. Ein solcher Ansatz nach [Aal00] und [Aal98] wird hier vorgestellt.

Im folgenden betrachten wir ein Workflowsystem zur Beschwerdebearbeitung (s. a. Abb. 4.12) :

Aktionen :

1. Aufnahme einer Beschwerde (register)
2. Fragebogen an Beschwerdeführer (send_questionnaie)

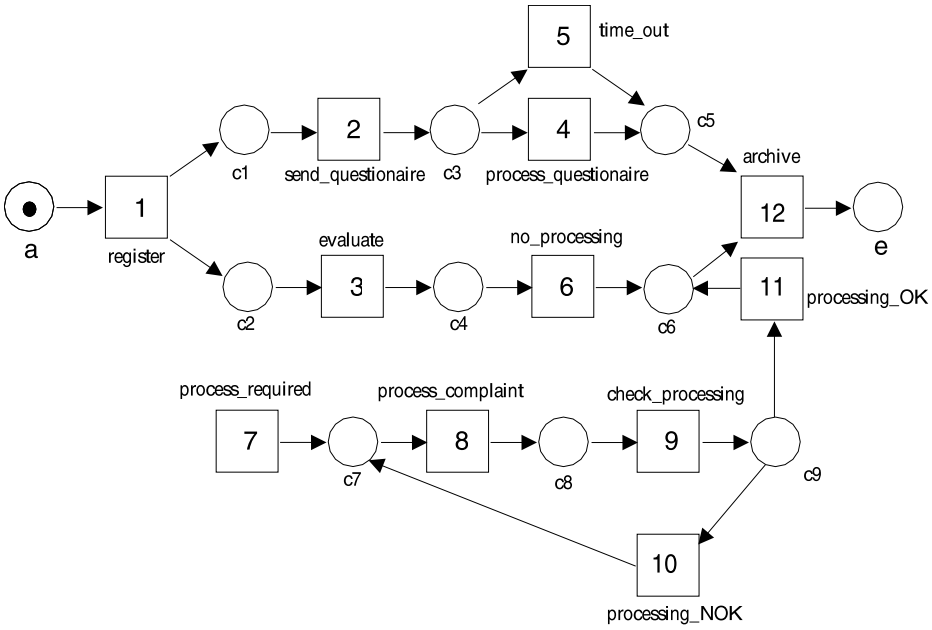


Abbildung 4.12: Ein Petrinetz für einen Vorgang zur Beschwerdenbearbeitung

3. Bewertung (evaluate) (nebenläufig zu 3.)
4. Fragebogenauswertung (process_questonaire), falls Rücklauf innerhalb von 2 Wochen, sonst:
5. Nichtberücksichtigung des Fragebogens (time_out).
6. Je nach Ergebnis der Bewertung (3.) : Aussetzung der Bearbeitung (no_processing) oder
7. Beginn der eigentlichen Prüfung (processing_required)
8. Bearbeitung der Beschwerde (process_complaint) unter Berücksichtigung des Fragebogens
9. Bewertung der Bearbeitung (check_processing) mit dem Ergebnis
10. erneute Prüfung (processing_nok) oder
11. Abschluss (processing_ok)
12. Ablage (archive)

Auch Plätze haben Bedeutung : z. B. c_2 : ,,Bewertung kann beginnen."
Falls mehrere Fälle im selben Netz dargestellt werden sollen : gefärbte Netze : Farbe $\{case_1, case_2, \ldots, case_{10}\}$ (Abb.4.13)

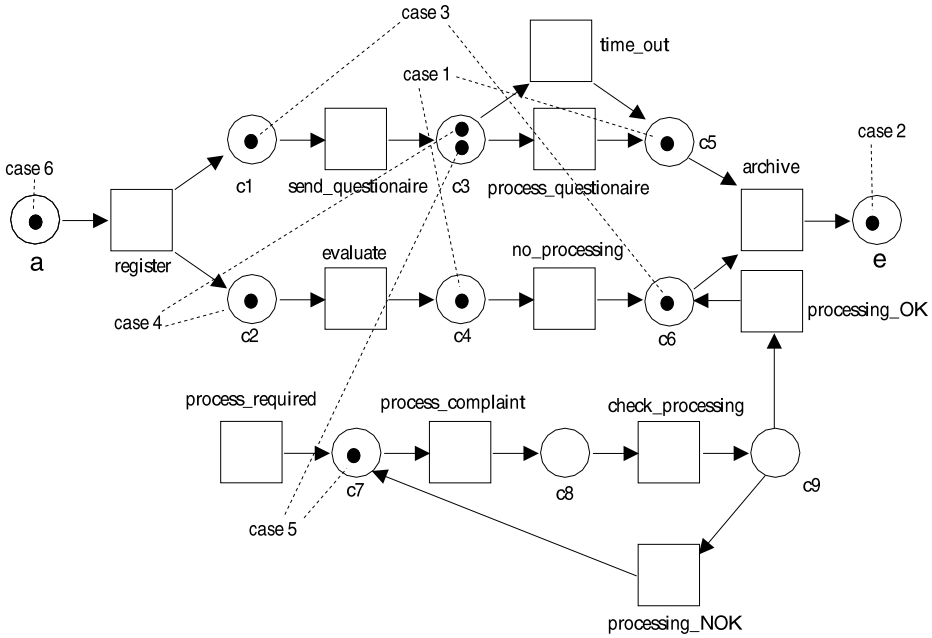


Abbildung 4.13: Gefärbtes Netz zur Darstellung mehrerer Fälle in einem Netz

Normalform solcher Netze :

a) ein Anfangsplatz a

b) ein Endplatz e

c) alle Transitionen liegen auf Pfaden zwischen a und e.

Definition 4.17 *Ein Netz $\mathcal{N} = (P, T, F)$ heißt* Work-Flow-Netz *(WF-Netz), falls*

a) es zwei besondere Plätze $\{a, e\} \subset P$ enthält mit ${}^\bullet a = \emptyset$ (,,Start", ,,Quelle", ,,Anfangsplatz") und $e^\bullet = \emptyset$ (,,Ende", ,,Senke", ,,Endplatz") und

b) alle Plätze und Transitionen auf Pfaden zwischen a und e liegen.

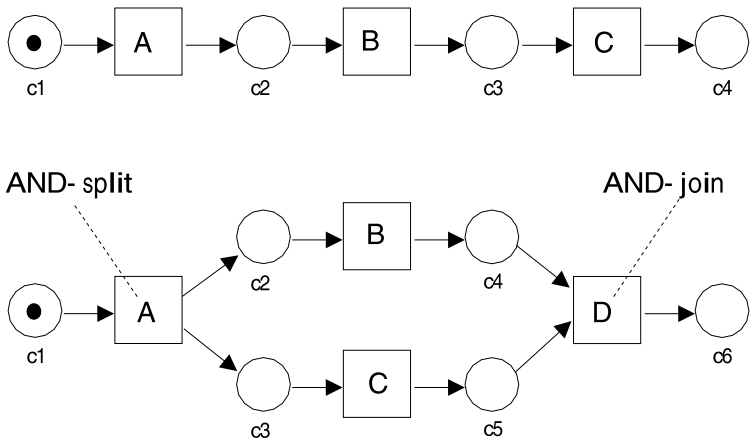


Abbildung 4.14: Sequentielle und nebenläufige Bearbeitung

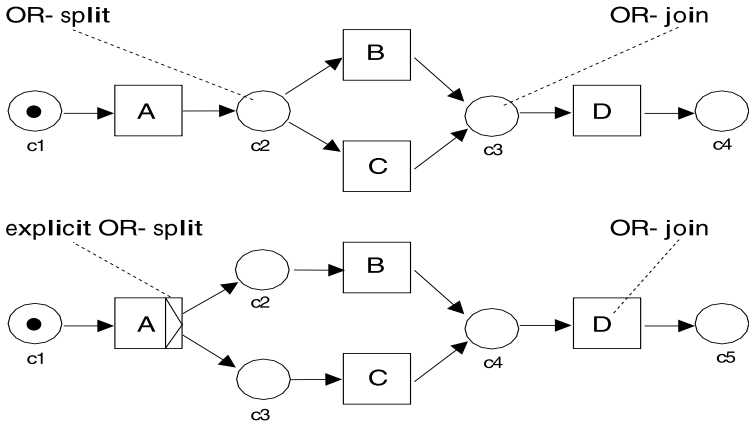


Abbildung 4.15: Alternative Bearbeitung mit und ohne expliziten Testbedingungen

typische Strukturen[4] : nebenläufige Bearbeitung und alternative Bearbeitung (mit und ohne expliziten Testbedingungen).

[4]Unter `http://www.tm.tue.nl/it/research/patterns/` ist eine Sammlung solcher Strukturmuster für Workflows zu finden.

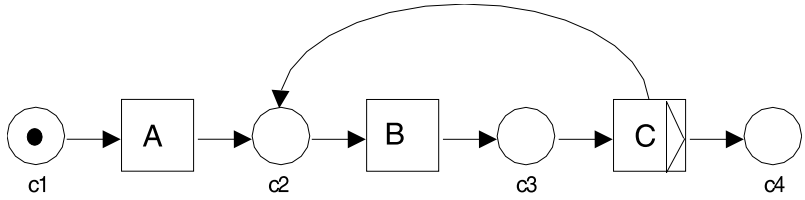


Abbildung 4.16: Iteration

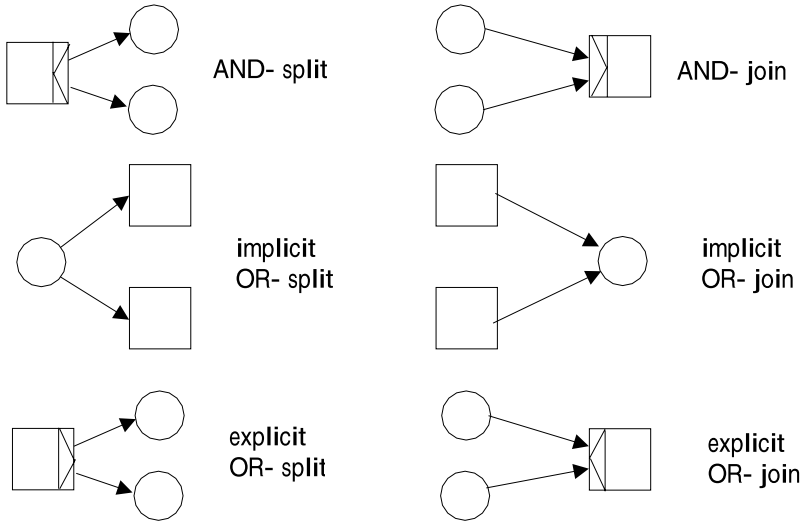


Abbildung 4.17: Graphische Symbole für Verzweigung

Ein Workflow hängt von der Interaktion mit der Umgebung ab (restriktives System, Trigger, Ressourcen). Beispiel :

- Bearbeiter ist in Urlaub oder krank
- Antwort auf eine Anfrage trifft nicht ein

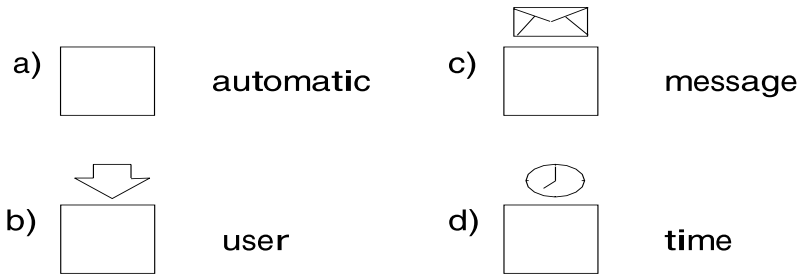


Abbildung 4.18: Trigger eines Workflowsystems

Es gibt folgende Arten von Triggern :

a) Automatisch (keine externe Eingabe notwendig)

b) Benutzer (user) : ein Bearbeiter / Benutzer / Funktionseinheit nimmt einen Auftrag an und bearbeitet ihn

c) Nachricht (message) : eine Nachricht von außen wird benötigt (Brief, Anruf, e-mail, Fax, ...)

d) Zeit (time) : es besteht eine Zeitbedingung für die Bearbeitung

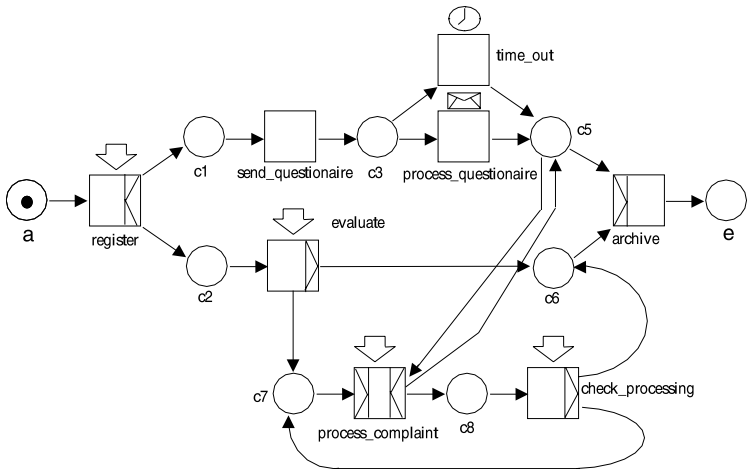


Abbildung 4.19: Workflow aus Abb. 4.12 mit Triggern

Wechselwirkung zwischen Triggern und Verzweigungen :

Beispiel : Ersetze die implizite Verzweigung in c3 der Abbildung 4.19 durch explizite !

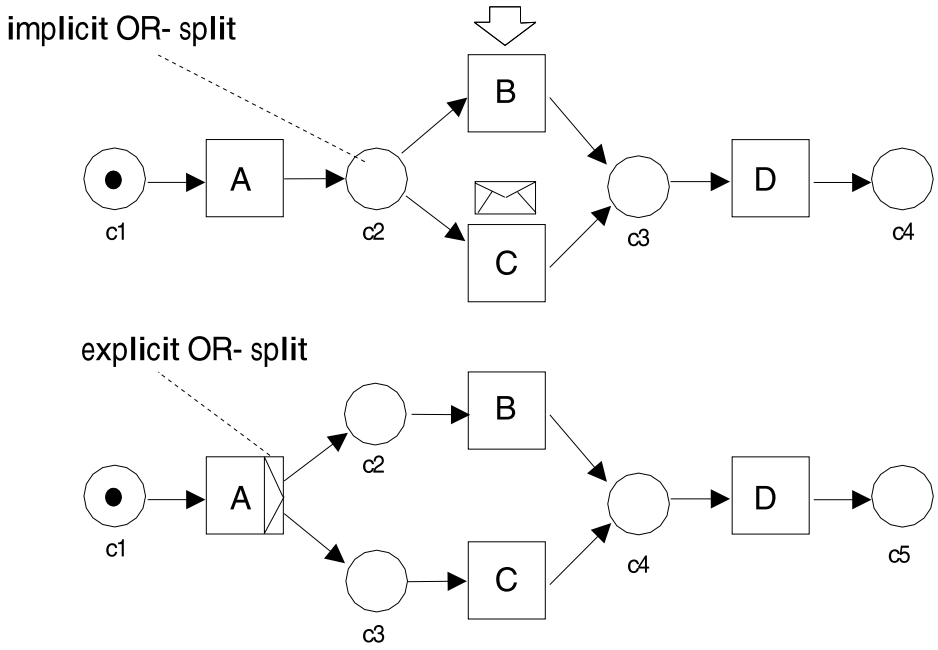


Abbildung 4.20: Unterschied zwischen impliziter Oder-Verzweigung (a) und expliziter Oder-Verzweigung (b)

anderes Problem : Zustandsinformation ,,milestones") manschmal erforderlich (Beispiel : c5 in Abb. 4.19)

Erfahrung aus der Praxis :

- Workflow-Prozesse werden oft nicht richtig verstanden (Mehrdeutigkeit, Widersprüche, Verklemmungen).
- Allein schon die (versuchsweise) Modellierung durch Petrinetze deckt Mängel auf.
- Bei fertiggestellten Petrinetzen können Mängel durch strukturelle Untersuchungen aufgedeckt werden (z. T. sogar automatisch).

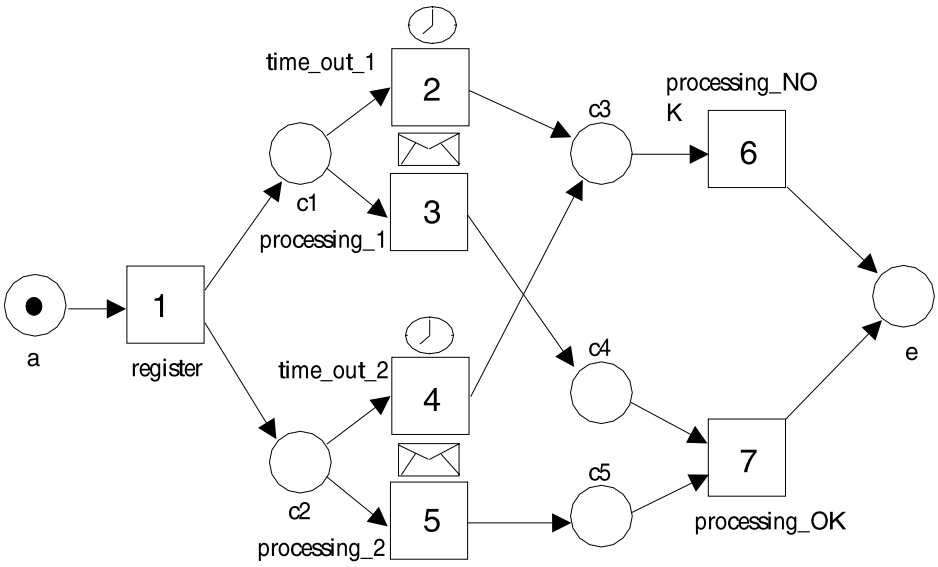


Abbildung 4.21: Problematisches Workflownetz für Beschwerdebearbeitung

Probleme :

- wenn 2 und 5 schalten : Verklemmung
- wenn 3 und 4 schalten : Verklemmung
- wenn 2 und 4 schalten : 6 wird 2x ausgeführt

Also : Modellierung des Workflow *nicht* korrekt !

Welches sind sinnvolle Anforderungen an ein korrektes Workflow-Netz ?

1. Definition 4.14 des Workflow-Netzes mit einer Marke in a (statisch zu überprüfen).
2. Für jede erreichbare Markierung terminiert des Netz mit genau einer Marke in e.
3. Jede Transition kann in einer zulässigen Schaltfolge schalten.

Definition 4.18 *Sei $\mathcal{N} = (P, T, F)$ ein Workflow-Netz (siehe Def. 4.17). Dann seien folgende Markierungen definiert :*

a) die Anfangsmarkierung $\mathbf{m}_a$ *durch :*
$\mathbf{m}_a(p) :=$ **if** $p = a$ **then** 1 **else** 0 **fi**

b) die Endmarkierung $\mathbf{m}_e$ *durch :*
$\mathbf{m}_e(p) :=$ **if** $p = e$ **then** 1 **else** 0 **fi**

Ein markiertes Workflow-Netz *(mWF-Netz) ist ein P/T-Netz $\mathcal{N} = (P, T, F, \mathbf{m}_a)$, wobei (P, T, F) ein Workflownetz ist und $\mathbf{m}_a$ wie in a) definiert ist.*
(Hier wird P/T-Netz als der Spezialfall von Def. 3.7 aufgefaßt, in dem die Kantenbewertung immer 1 ist, also: $W(x, y) = 1 \Leftrightarrow (x, y) \in F$).

Definition 4.19 *Ein markiertes WF-Netz $\mathcal{N} = (P, T, F)$ heißt* korrekt*, falls gilt :*

a) $\forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) \exists w \in T^* : \mathbf{m} \xrightarrow{w} \mathbf{m}_e$ *(vergl. Def. 3.6)*

b) $\forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) : \mathbf{m}(e) \geq 1 \Rightarrow \mathbf{m} = \mathbf{m}_e$

c) $\forall t \in T \exists \mathbf{m} \in \mathbf{R}(\mathcal{N}) : \mathbf{m} \xrightarrow{t}$ *(vergl. Def 3.9 b))*

Erklärung:

a) „$\mathcal{N}$terminiert immer"

b) $\mathbf{m}_e$ ist die einzige Möglichkeit zu terminieren, d. h. es bleiben sonst keine Marken zurück

c) keine Transition ist ,,tot".

Das mWF-Netz aus Abbildung 4.21 erfüllt c) aber nicht a) und nicht b).

Wie sind Transitionen mit Triggern hier zu behandeln ?

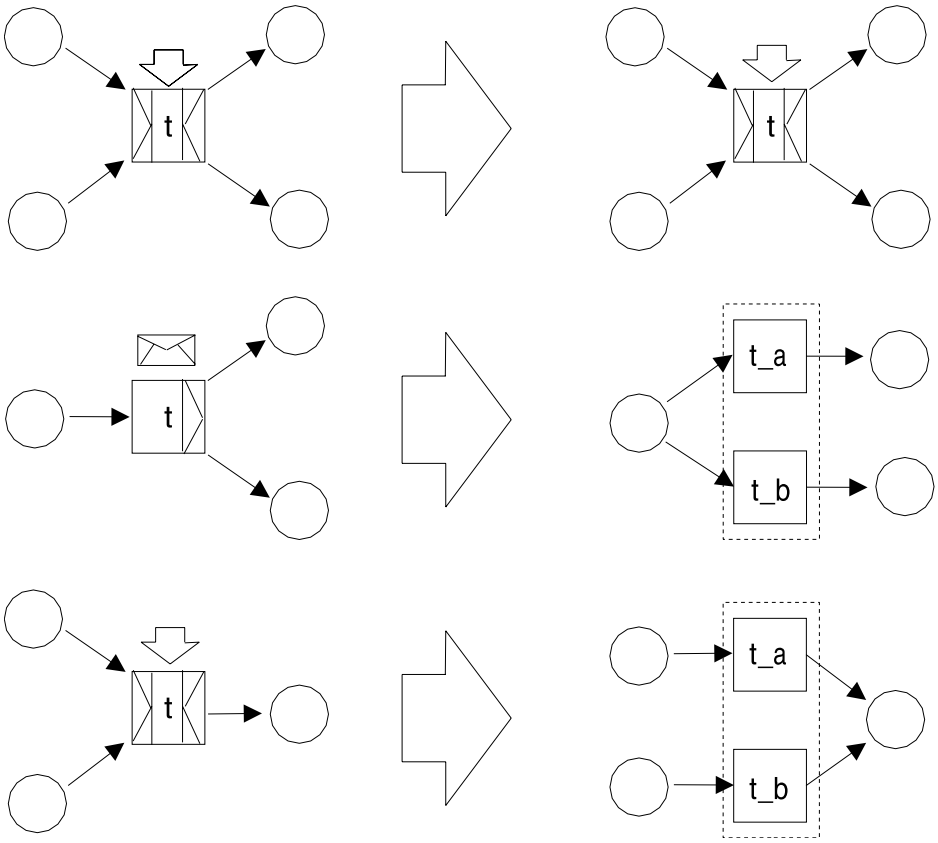


Abbildung 4.22: Übertragung von Triggern in ein mWF-Netz

Im folgenden Satz werden korrekte mWF-Netze durch die Eigenschaften ,,beschränkt" und ,,lebendig" (Def. 4.12) charakterisiert. Dazu benötigen wir eine kleine Transformation des WF-Netzes. Sie besteht in der Hinzufügung einer neuen Transition t^* von e nach a.

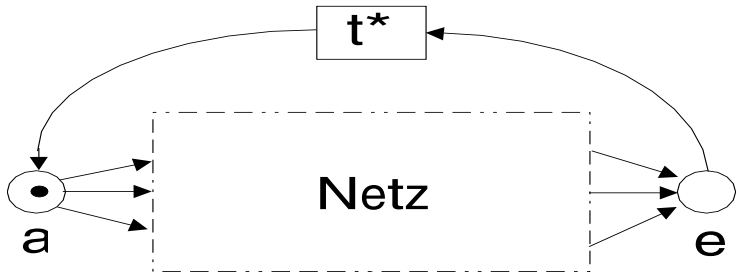


Abbildung 4.23: Transformation eines WF-Netzes

Definition 4.20 *Zu einem WF-Netz $\mathcal{N} = (P, T, F)$ definieren, wir den Abschluß $\overline{\mathcal{N}} = (\overline{P}, \overline{T}, \overline{F})$ durch*

a) $\overline{P} := P$

b) $\overline{T} := \overline{T} \cup \{t^*\}$ *(disjunkte Vereinigung)*

c) $\overline{F} := \overline{F} \cup \{(e, t^*), (t^*, a)\}$

Diese Definition wird auf mWF-Netze $\mathcal{N} = (P, T, F, \mathbf{m}_e)$ übertragen.

Theorem 4.21 *Ein markiertes WF-Netz M ist genau dann korrekt, wenn sein Abschluß $\overline{\mathcal{N}}$ lebendig und beschränkt ist.*

Beweis siehe [Aal97].

Aufgabe 4.22 Zeigen Sie, dass für das mWF-Netz $\mathcal{N}$ aus Abbildung 4.21 der Abschluss $\overline{\mathcal{N}}$ nicht beschränkt ist ! Ebenso zeige man, dass $\overline{\mathcal{N}}$ nicht beliebig ist !
Analysieren Sie das folgende mWF-Netz.

Das vorstehende markierte WF-Netz ist zwar korrekt im Sinne von Definition 4.19, es kann aber zu Fehlverhalten kommen, bei dem die Transition 2 oder 5 unendlich oft schalten und dadurch nicht die Endmarkierung erreicht wird. Um solche Fälle auszuschließen kann gefordert werden, dass sie solche ,,fair verhalten" (vergl. Def 4.14).

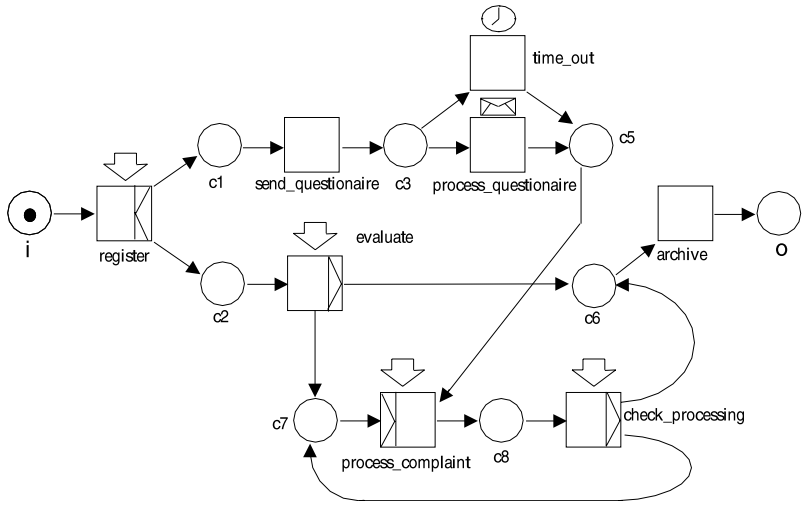


Abbildung 4.24: Ein markiertes WF-Netz zur Behandlung von Beschwerden

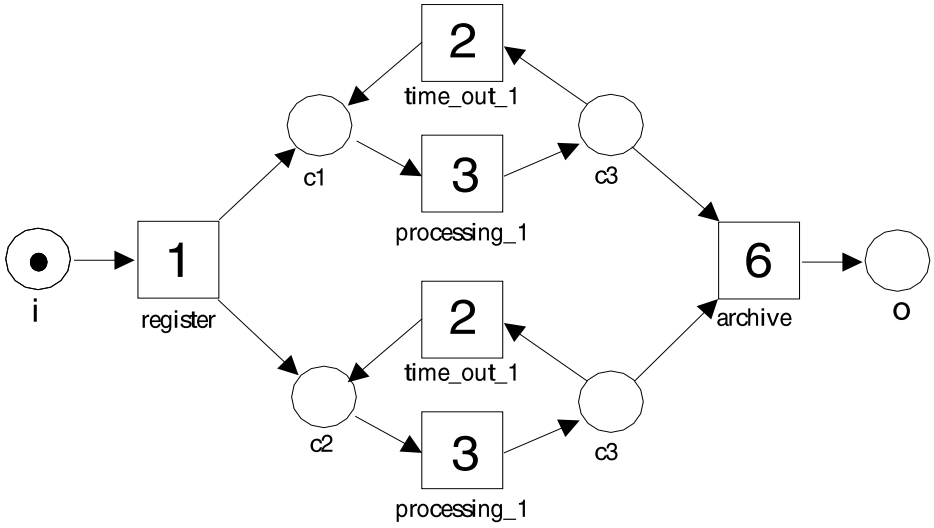


Abbildung 4.25: Ein markiertes WF-Netz zur Fairneß-AnnahmeBehandlung von Beschwerden

4.2 Datenkonsistenz

4.2.1 Schematische Auftragssysteme

Wir knüpfen an das am Ende von Kapitel 2 diskutierte Problem bei nebenläufigen Datenzugriff an. Dazu betrachten wir einige Beispiele von Programm-Fragmenten. Zur Erklärung der Anweisungen siehe die Sprache *Prog* in Definition 5.1.

Beispiel 4.23

$$
\begin{array}{lll}
P_1 & : & a_0 \ : \ z := 1 \\
 & & \underline{con}\ a_1 \ : \ x := z+1;\ a_2 : \ z := x \\
 & & \| \ b_1 \ : \ y := z+2; = b_2 : \ z := y \\
 & & \underline{noc} \\
P_2 & : & a_0 \ : \ z := 1; \\
 & & \underline{con}\ A : \ Z := Z+1 \ \| \ B : \ Z := Z+2 \ \underline{noc}
\end{array}
$$

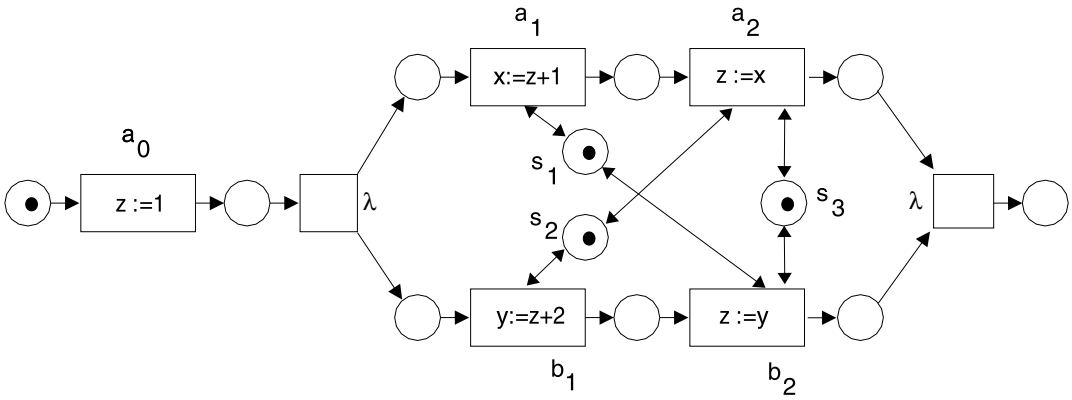


Abbildung 4.26: Netzprogramm zu P_1

Ein Ablauf des Programms P_1 kann z.B. als Schaltfolge

$$w_1 = a_0 a_1 b_1 a_2 b_2$$

des S/T-Netzes von Abb. 4.26 dargestellt werden (die mit λ benannten Hilfstransitionen wurden weggelassen). Für die ebenfalls möglichen Schaltfolgen

$$w_2 = a_0 a_1 a_2 b_1 b_2$$

bzw.

$$w_3 = a_0 a_1 b_1 b_2 a_2$$

sind die Endwerte von z gleich 4 bzw. gleich 2. Bei den Programm P_2 ist nur der w_2 entsprechende Wert möglich. Betrachtet man jedoch nur die Auswertung der rechten Seiten in P_2 als elementare Handlungen, dann kann man die gleichen Überlegungen wie bei P_1 anstellen.

Wir entnehmen dem Beispiel:

1. Bei nebenläufigen Programmen sind verschiedene Endergebnisse möglich.
2. Dies hängt auch davon ab, welche Anweisungen und Handlungen als atomar betrachtet werden.

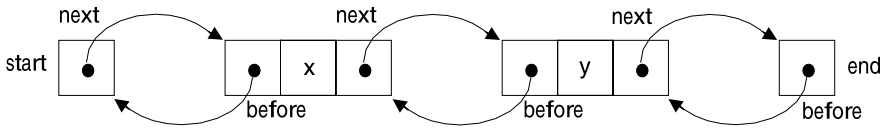


Abbildung 4.27: Doppelt verkettete Liste

Beispiel 4.24 [BR81] Die Anweisunge $delete(p)$ soll ein Listenelement p aus einer doppelt verketteten Liste wie Abb. 4.27 entfernen. Dabei seien p_1 und p_2 lokale Variablen der Anweisung.

$$\begin{array}{l} delete(p) \\ \quad \underline{con}\ a_1\ :\ p_1 := p.before\ ||\ b_1\ :\ p_2 := p.next\ \underline{noc} \\ \quad \underline{con}\ a_2\ :\ p_1.next := p_2\ ||\ b_2\ :\ p_2.before := p_1\ \underline{noc} \end{array} \tag{4.3}$$

Wendet man

$$p_3 :\ \underline{con}\ delete(x)\ ||\ delete(y)\ \underline{noc} \tag{4.4}$$

auf die Liste von Abb. 4.27 an, so erhält man bei der seriellen Ausführung von $delete(x)$ und $delete(y)$, wie gewünscht, die leere Liste von Abb. 4.28 a), bei der kollateralen Ausführung von $a_1||b_1$ und $a_2||b_2$ durch die Folge

$$w = a_1^x a_1^y b_1^x b_1^y a_2^x a_2^y b_2^x b_2^y$$

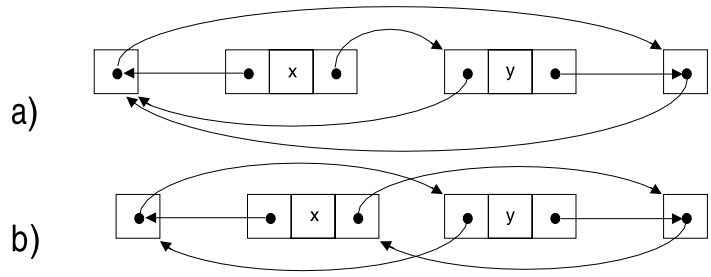


Abbildung 4.28: Ergebnis der Anwendung von 4.4: a) bei serieller Ausführung und b) bei kollateraler Ausführung

jedoch das Gebilde von Abb. 4.28 b). (Die Exponenten geben an, ob die entsprechende Anweisung zu $delete(x)$ oder $delete(y)$ gehört.)

Wir analysieren das Problem mit Hilfe von "schematischen Auftragssystemen", die auf einfachen Auftragssystemen basieren:

Definition 4.25 *Ein Auftragssystem* $AS = (A, \lessdot)$ *besteht aus einer endlichen Menge* A *von* Aufträgen *und einer irreflexiven, transitiven Relation* $\lessdot \subset A \times A$*, genannt* Präzendenzrelation *(precedence relation). Für* $(a_i, a_j) \in \lessdot$ *schreiben wir auch* $a_i \lessdot a_j$ *und nennen den Auftrag* a_i *präzedent zu* a_j*.* a_i *heißt direkt präzedent zu* a_j*, wenn gilt* $a_i \lessdot a_j$*, jedoch* $a_i \lessdot a_k \lessdot a_j$ *für kein* $a_k \in A$ *gilt.*

Auftragssysteme sind also endliche Striktordnungen (Def. 2.1 b)). Das Beispiel von Abb. 4.30 als Netz mit Anfangsmarkierung dargestellt. Als Ausführungsfolgen eines Auftragssystems betrachten wir jede lineare Vervollständigung (Def. 2.2, Anmerkung e). Diese Ausführungsfolgen entsprechen den maximalen Ausführungsfolgen des entsprechenden Netzes (Kausalnetz wie z.B. in Abb. 4.30). Die Menge der Ablauffolgen, die alle Aufträge enthalten, wird mit $F_E(AS)$ bezeichnet, die Menge der Anfangsstücke dieser Folgen mit $F(AS)$.

Im Beispiel der nebenläufigen Listen-Operationen (Beispiel 4.27) war klar, dass die genannte Ausführungsfolge w als inkorrekt zu betrachten ist. Anders in Beispiel 4.23: welches oder welche der drei möglichen Resultate sollen als korrekt gelten?

Eine ähnliche, jedoch i.a. ungleich komplexere Situation finden wir bei Aufträgen, die auf große Datensätze zugreifen. Ein solcher Auftrag a_i wird auch als *Transaktion* (tran-

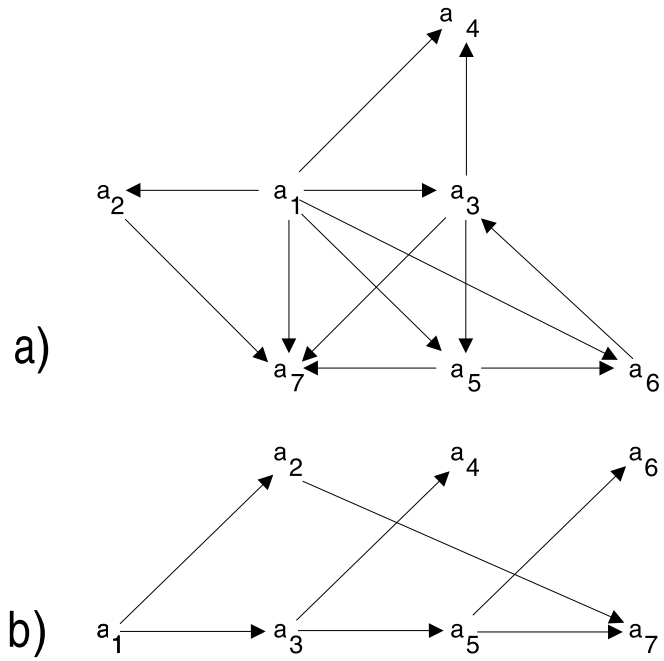


Abbildung 4.29: Präzendenzgraph G_1 in a) und zugehöriger Konnektivitätsgraph G_2 in b)

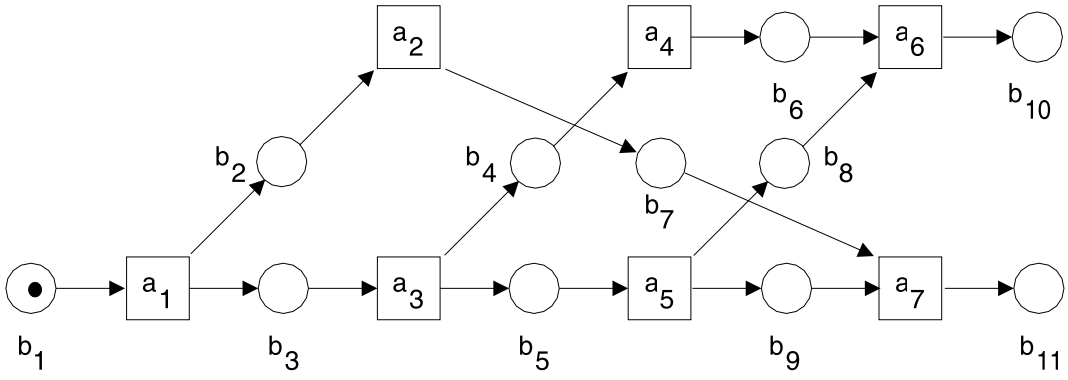


Abbildung 4.30: Auftragssystem als Kausalnetz

saction) bezeichnet. Er zergliedert sich i.a. in mehere unteilbare Teilauftträge $a_i^1, \ldots, a_i^k$, die sequentiell auszuführen sind. Aus Effektivitätsgründen (z.B. anehmbare Antwortzeit) müssen mehrere solche Aufträge oder Transaktionen $a_1, \ldots, a_n$ nebenläufig ausgeführt werden. Dabei sollen nach der Ausführung gewisse Datenbeziehungen erhalten bleiben (z.B. gleiche Adresse bei nicht getrennt lebenden Eheleuten). Man nennt dies die *Konsistenz* des Datensatzes.

Man geht davon aus, dass jede Transaktion a_i für sich korrekt arbeitet, d.h. ausgehend von einem konsistenten Datensatz einen solchen auch hinterläßt. Arbeiten nun mehere Transaktionen nebenläufig zueinander, dann entsteht nach Beendigung aller Transaktionen ein Zustand der Datei, dessen Konsistenz fraglich ist. Wären die Transaktionen seriell, d.h. nicht überlappend, ausgeführt worden, dann wäre nach unsere Annahme die Konsistenz gesichert. In Unkenntnis anderer allgemeingültiger Kriterien für Konsistenz betrachtet man einen durch nebenläufige Ausführung mehrerer Transaktionen erhaltenen Zustand der Datei als konsistent, wenn dieser Zustand auch bei einer ungeteilten Ausführung aller beteiligeten Transaktionen a_i in irgend einer Reihenfolge $a_{i_1}; a_{i_2}; \ldots; a_{i_n}$ herstellbar ist. Die ursprüngliche Auszuführungsfolge heißt dann *serialisierbar* (serializable). (Eswaran et al 76 [EGLT76]; Papadimitriou 79 [Pap79]; Bernstein et al 79 [BSW79]; Sethi 82 [Set82])

Die ungeteilte Ausführung von $< a_1; a_2 >$ und $< b_1; b_2 >$ in Beispiel 4.23 liefert in beiden Serialisierungen $z = 4$. Jede ein anderes Ergebnis liefernde Folge ist daher nicht serialisierbar. Im Beispiel 4.27 liefert z.B. w' : $a_1^x\ b_1^y\ b_1^x\ b_2^x\ a_1^y\ b_2^y\ a_2^x\ a_2^y$ wie die ungeteilte Serialisierung die leere Liste.

Im folgenden spezialisieren wir uns auf den Fall $k = 2$ und fassen eine Transaktion als Auftrag a_i eines Auftragssystems $AS = (A, \lessdot)$ auf. a_i hat also eine Verfeinerung in zwei Teilaufträgen l_i und s_i, die Lese- und Schreibauftrag von a_i heißen. Es besteht natürlich die Präzedenz $l_i \lessdot s_i$. Alle Präzedenzen $a_j \lessdot a_i$ in AS übertragen sich zu $s_j \lessdot l_i$.

Beispiel 4.26 Wir betrachten das Auftragssystem $AS = (A, \lessdot)$ aus Abb. 4.31. Die

Aufträge mögen folgende Verfeinerung in Leseaufträge l_i und s_i haben:

$l_1 : skip$	$s_1 : x, y, z := 0, 1, 2$
$l_2 : skip$	$s_2 : x := 20$
$l_3 : lo_3 := x$	$s_3 : x := lo_3 + 5$
$l_4 : lo_4 := y$	$s_4 : x, y, z := lo_4 - 1, lo_4 + 10, lo_4$
$l_5 : lo_5 := z$	$s_5 : y := lo_5$
$l_6 : lo_6 := x$	$s_6 : y := 2 \cdot lo_6$
$l_7 : write(x, y, z)$	$s_7 : skip$

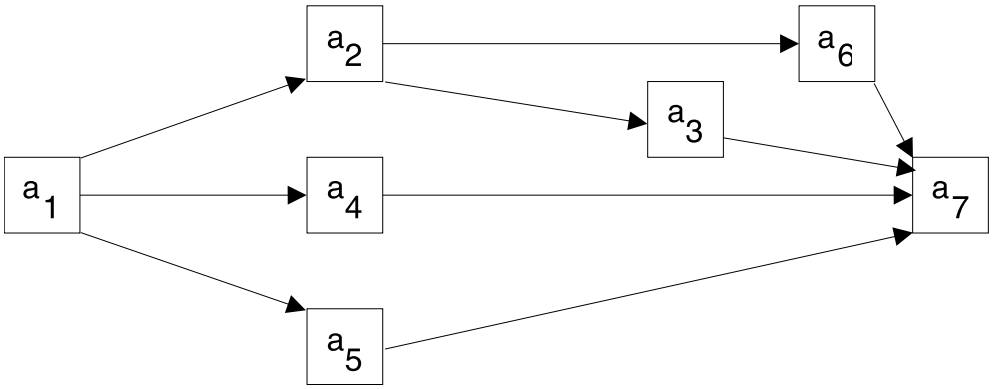


Abbildung 4.31: Ein Auftragssystem AS

Abbildung 4.32 zeigt das entsprechend verfeinerte Auftragssystem $AS' = (A', \lessdot')$, wobei mit gestrichelten Pfeilen die Zugriffe auf die global benutzten Variablen x, y und z dargestellt sind.

Die mögliche Ausführungsfolge

$$w = l_1\ s_1\ l_2\ s_2\ l_4\ l_3\ l_6\ s_4\ l_5\ s_3\ s_6\ s_5\ l_7\ s_7$$

liefert das Resultat $(x, y, z) = (25, 1, 1)$. Ist dieser Zustand konsistent?

Die ungeteilte Ausführung $< a_1 > < a_2 > < a_3 > < a_4 > < a_5 > < a_6 > < a_7 >$ liefert zwar den Zustand $(0, 0, 1)$, jedoch existiert die folgende Serialisierung $w' =< a_1 > < a_4 > < a_2 > < a_6 > < a_5 > < a_3 > < a_7 >$ mit dem gleichen Resultat $(25, 1, 1)$. Daher

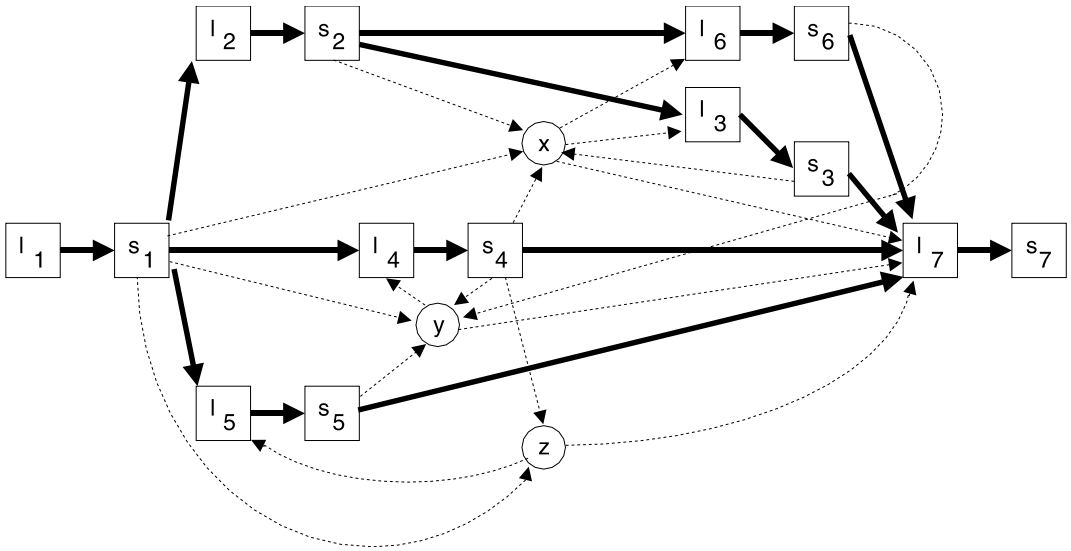


Abbildung 4.32: Verfeinertes Auftragssystem AS' mit Darstellung der Schreib/ und Lese-Zugriffe

ist das Resultat von w konsistent. Die unteilbaren Anweisungen von w' können durch Elimination der lokalen Variablen übersichtlicher wie folgt geschrieben werden:

$$
\begin{aligned}
&< a_1 > : x,y,z{:=}0,1,2\\
&< a_4 > : x\ y,z{:=}y-1,y+10,y\\
&< a_2 > : x \quad :=20\\
&< a_6 > : \quad y \quad :=2\cdot x\\
&< a_5 > : \quad y \quad :=z\\
&< a_3 > : x \quad :=x+5\\
&< a_7 > : \ \ write(x,y,z)
\end{aligned}
$$

Im folgenden wird ein Algorithmus zur Entscheidung der Serialisierbarkeiteits-Eigenschaft entwickelt. Da im allgemeinen weder die Eingangswerte noch die Aufträge bekannt sind oder feststehen, werden Lese- und Schreibaufträge nur durch die zu lesenden und zu überschreibenden Variablen gekennzeichnet. Eine konkrete Festlegung der die Wirkung der Lese- und Schreibaufträge beschreibenden Funktion nennt man Interpretation.

Definition 4.27 *Ein Auftragssystem $AS = (A, \lessdot)$ (Def. 4.25) heißt* schematisch *oder* uninterpretiert*, wenn die Auftragsmenge die Form*

$$A = \{l_1, s_1, \ldots, l_n, s_n\} \quad (n \geq 1)$$

hat. Gefordert werden auf jeden Fall die direkten Präzedenzen $l_i \lessdot s_i$ $(1 \leq i \leq n)$. Weiter können weitere direkte Präzedenzen der Form $s_i \lessdot l_j$ $(1 \leq i, j \leq n)$ $(i \neq j)$ bestehen. l_i bzw. s_i heißen Lese- bzw Schreib-Auftrag. Zusammen bilden sie den Auftrag $a_i = (l_i, s_i)$. Ferner sei eine endliche Menge $V = \{v_1, v_2, \ldots, v_p\}$ von Variablen gegeben. Zu jedem a_i $(1 \leq i \leq n)$ ist.

- *eine Menge $ein_i := \{e_1, \ldots, e_{p_i}\} \subseteq V$ von Eingangsvariablen und*
- *eine Menge $aus_i := \{u_1, \ldots, u_{q_i}\} \subseteq V$ von Ausgangsvariablen festgelegt.*

AS kann somit eindeutig durch $\lessdot$ und die Schreibweise $A = \{l_1[ein_1], s_1[aus_1], \ldots, l_n[ein_n], s_n[aus_n]\}$ dargestellt werden. Ist $A' = \{l_{i_1}, s_{i_1}, \ldots, l_{i_m}, s_{i_m}\} \subseteq A$ eine Teilmenge der Aufträge $(m \leq n)$ und $\lessdot' = \lessdot_{|A'}$, dann heißt $AS' = (A', \lessdot')$ Unterauftragssystem von AS.

Definition 4.28 *Die Vergröberung $\hat{AS} = (\hat{A}, \lessdot)$ eines schematischen Auftragssystems wie in 4.27 ist gegeben durch die Auftragsmenge*

$$\hat{A} := \{a_1, \ldots, a_n\}$$

und die Präzedenzen

$$a_i \lessdot a_j \leftrightarrow s_i \lessdot l_j \qquad (1 \leq i, j \leq n)$$

Die Mengen ein_i und aus_j werden beibehalten.

Beispiel 4.29 Dem Auftragssystem aus Beispiel 4.26 entspricht das schematische Auftragssystem

$$AS = (A, \lessdot)$$

mit

$$\begin{aligned} A = \; & \{l_1, s_1[xyz], l_2, s_2[x], l_3[x], s_3[x], l_4[y], s_4[xyz], l_5[], s_5[y], \\ & l_6[x], s_6[y], l_7[xyz], s_7\} \end{aligned}$$

(Mengenklammern von ein_i, aus_i bzw. $[\emptyset]$ wurden weggelassen). Abbildung 4.31 zeigt den Konnektivitätsgraph der Vergröberung AS.

Oft ist es nützlich, am Anfang einer jeden Ausführungsfolge einen "Initialisierungs-Auftrag" zu haben, der alle Variablen beschreibt, und am Ende einen "Ausgabeauftrag", der alle Variablen liest und an den Drucker weitergibt. Ein solches Auftragssystem heißt *vollständig*. Das schemtische Auftragssystem von Beispiel 4.26 ist vollständig.

Definition 4.30 *Ein schematisches Auftragssystem $AS = (A, \lessdot)$ wie in Definition 4.27 heißt vollständig, wenn $n \geq 3$ und*

- $a_1 = (l_1, s_1)$ *der* Initialisierungsauftrag *ist mit $ein_1 = \emptyset, aus_1 = V$ und $s_1 \lessdot l_i$ für alle $1 < i \leq n$, sowie*
- $a_n = (l_n, s_n)$ *der* Ausgabeauftrag *ist mit $ein_n = V$, $aus_n = \emptyset$ und $s_i \lessdot l_n$ für alle $1 \leq i < n$.*

Macht man ein nicht vollständiges schematisches Auftragssystem durch Hinzufügen dieser Aufträge vollständig, so sprechen wir von der Vervollständigung. *Analog sprechen wir von der* Vervollständigung einer Ausführungsfolge $w \in F_E(AS)$.

Definition 4.31 *Eine* Interpretation *I eines schematischen Auftragssystems AS mit Bezeichnung wie in Definition 4.27 ist durch eine Wertemenge D_I (kurz D) und für jedes $1 \leq i \leq n$ durch Funktionen*

$$f_i^j : D^{p_i} \to D \quad (1 \leq j \leq q_i)$$

sowie durch einen Anfangszustand $d_0 \in D^p$ gegeben. Ist $p_i = 0$, dann ist $f_i^j \in D$ eine Konstante, ist $q_i = 0$, dann entfällt f_i^j.

Ein Zustand ist ein Vektor $d \in D^p$ der Länge p und ordnet jeder Variablen $v_i \in V$ einen Wert $d_i \in D$ zu.

Zu einer Ausführungsfolge $w = w_1 w_2 \ldots w_{2n} \in F_E(AS)$ ist eine Zustandsfolge $d_0, d_1, d_2, \ldots, d_{2n}$ mit $d_i \in D^p$ folgendermaßen definiert:

- *d_0 ist der Anfangszustand*

- *Ist $w_j = l_i$ ein Leseauftrag $(1 \leq j \leq 2n)$, dann ist $d_j = d_{j-1}$ und für eine nur in a_i vorkommende Menge von lokalen Variablen $lok_i := \{lo_1, \ldots, lo_{p_i}\}$ erhalten wir die Werte:*

$$lo_1, lo_2, \ldots, lo_{p_i} := e_1, e_2, \ldots, e_{p_i}$$

- *Ist $w_j = s_i$ ein Schreibauftrag, dann entsteht d_j aus d_{j-1} durch die Zuweisung:*

$$u_1, \ldots, u_{q_i} := f_i^1(lo_1, \ldots, lo_{p_i}), \ldots, f_i^{q_i}(lo_1, \ldots, lo_{p_i})$$

Mit $res(w, I) := d_{2n}$ bezeichnen wir das Resultat von w bezüglich der Interpretation I.

Zwei Ausführungsfolgen $w_1, w_2 \in F_E(AS)$ heißen äquivalent, wenn

$$res(w_1, I) = res(w_2, I)$$

für alle *Interpretationen I von AS gilt.*

Da eine Interpretation I von AS auch eine Interpretation für jedes Unterauftragssystem AS' von AS ist, ist diese Definition der Äquivalenz auch für (nicht maximale) Ausführungsfolgen $w_1 \in F(AS)$ und $w_2 \in F(AS')$ sinnvoll und gültig.

Beispiel 4.32 Wählt man für das schematische Auftragssystem vom Beispiel 4.29 die Interpretation I mit $D_I = \mathbb{Z}$ und

$$\begin{aligned} f_1^1 &= 0, f_1^2 = 1, f_1^3 = 2, \\ f_2^1 &= 20 \\ f_3^1(lo_3) &= lo_3 + 5 \\ f_4^1(lo_4) &= lo_4 - 1, f_4^2(lo_4) = lo_4 + 10, f_4^3(lo_4) = lo_4 \\ f_5^1(lo_5) &= lo_5 \\ f_6^1(lo_6) &= 2 \cdot lo_6 \end{aligned}$$

und $d_0 = (0, 0, 0)$, dann erhält man Beispiel 4.31

(Da s_1 alle Variablen überschreibt, ist d_0 unwichtig). Für die Ausführungsfolge w in Beispiel 4.31 erhält man die Zustandsfolge:

	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}
x	0	0	0	0	20	20	20	20	0	0	25	25	25	25	25
y	0	0	1	1	1	1	1	1	11	11	11	40	1	1	1
z	0	0	2	2	2	2	2	2	1	1	1	1	1	1	1

Folglich gilt $res(w, I) = d_{14} = (25, 1, 1)$

Im folgenden soll ein notwendiges und hinreichendes Kriterium für die Äquivalenz von Ausführungsfolgen entwickelt werden.

Definition 4.33 *Es sei $AS = (A, \lessdot)$ wie in 4.27. Wir definieren für eine gegebene Ausführungsfolge $w \in F(AS)$ und $a_1, a_2 \in A$:*

$$a_1 <_w a_2, \text{ falls } a_1 \text{ vor } a_2 \text{ in } w \text{ vorkommt.}$$

Weiter sagen wir:

l_i *liest* v *von* s_j *in* w, *falls*

a) $s_j <_w l_i$

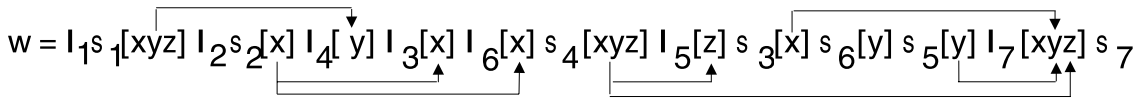
b) $w \in ein_i \cap aus_j$ *(v wird von l_i gelesen und von s_j geschrieben)*

c) für alle $a_p = (l_p, s_p), p \notin \{j, i\}$ *mit* $v \in aus_p$ *gilt*

$$s_p <_w s_j \text{ oder } l_i <_w s_p \text{ (kein dritter Auftrag schreibt dazwischen).}$$

Die Relation $W\ddot{U}(w) := \{(a_j, a_i) | \exists v \in V : l_i$ *liest* v *von* s_j *in* $w\}$ *heißt* Werteübertragunmgsrelation .

Beispiel 4.34 Für w von Beispiel 4.31 läßt sich WÜ(w) wie folgt angeben:



In einer Ausführungsfolge kann es Aufträge geben, deren Ausführung bei keiner Interpretation einen Einfluss auf das Resultat haben können. Solche Aufträge heißen nutzlos, die anderen relevant.

Definition 4.35 *Sei $AS = (A, \lessdot)$ ein schematisches und vollständidges Auftragssystem und wie $w \in F(AS)$ eine Ausführungsfolge. Die für w relevanten Aufträge definiert:*

a) Der Ausgabeauftrag ist relevant

b) Wenn $a_i \in A$ relevant ist und $(a_j, a_i) \in W\ddot{U}(w)$ gilt, dann ist auch a_j relevant.

c) Nur nach a) und b) erhaltene Aufträge heißen relevant für w, die anderen nutzlos für w.

Ein Auftrag $a_i \in A$ heißt relevant, falls er für mindestens eine Ausführungsfolge $w \in F(AS)$ relevant ist, sonst nutzlos oder inhärent nutzlos. AS heißt relevant, falls alle Aufträge relevant sind.

Beispiel 4.36 Aus Beispiel 4.34 entnehmen wir: a_7 ist relevant für w, also auch a_5, a_3 und a_4. Da a_3 bzw. a_4 relevant für w sind, gilt dies auch für a_2 bzw. a_1. a_6 ist nutzlos für w. Die von ihm beschriebene Variable y wird von a_5 überschrieben, ohne seine Ergebnisse zu benutzen. a_6 ist nicht inhärent nutzlos, denn a_6 ist relevant für $w') = l_1 s_1 l_2 s_2 \ldots l_6 s_6 l_7 s_7$. Folglich ist AS relevant.

Satz 4.37 *Zwei Ausführungsfolgen $w_1, w_2 \in F(AS)$ eines schematischen Auftragssystems AS sind* genau dann *äquivalent, wenn für ihre Vervollständigung w_1', w_2' gilt:*

a) w_1', w_2' haben die gleichen Mengen von relevanten Aufträgen und

b) für alle relevanten Aufträge a_i, a_j und jedes $v \in V$ gilt:
l_j liest v von a_i in $w_1' \leftrightarrow a_j$ liest v von a_i in w_2'

Entsprechendes gilt, falls $w_2 \in F(AS')$ für ein Unterauftragssystem AS'.

Beweis:
Der Beweis ist gleich demjenigen für den analogen Satz über die "Aquivalenz von Programm-Schemata (Luckham et al 70) mit Hilfe von "Herbrand-Interpretationen". Wir erl"autern die Hauptidee des Beweises anhand unseres Beispiels. Zunächst vervollständigen wir AS. Der Anfangszustand wird jetzt bei einer beliebigen Interpretation

I durch die Funtkionen des Initialisierungs-Auftrages bestimmt. Da wir alle Interpretationen betrachten, kommen auch alle ursprünglichen Anfangszustände vor.

Die Herbrand-Interpretation H hat als Wertebereich D die Menge aller Terme über den vorkommenden Funktionen f_j^i. Funktionsanwendungen bedeutet dann Termerweiterung. Wendet man z.B. $f(x)$ auf $f_3(f_1(x,y))$ an, dann ergibt sich $f(f_3(f1(x,y)))$. $res(w_i, H)$ ist dann ein p-Tupel solcher Terme. Die Ausführungsfolge w aus Beispiel 4.34 liefert als Resultat:

$$\begin{aligned} x &= f_3^1(lo_3) \text{ mit } lo_3 = x' \text{ und } x' = f_2^1, \text{ also } x = f_3^1(f_2^1) \\ y &= f_5^1(lo_5) \text{ mit } lo_5 = z \text{ und } z = f_4^3, lo_4 = y, y = f_1^2 \\ &= \text{ also } y = f_5^1(f_4^3(f_1^2)) \\ z &= f_4^3(lo_4) \text{ mit } lo_4 = y \text{ und } y = f_1^2, \text{ also } z = f_4^3(f_1^2) \\ &= \text{ also } res(w, H) = (f_3q(f_2^1), f_5^1(f_4^3(f_1^2)), f_4^3(f_1^2)) \end{aligned}$$

Man sieht, dass in den Resultaten $res(w_i, H)$ nur Funktionszeichen f_i^j von relevanten Aufträgen a_i vorkommen (also nicht f_6^1). Die Termbildung geschieht entsprechend der "liest v von"-Relation. Gelte also a) und b) für w_1', w_2', dann gilt

$$res(w_1', H) = res(w_2', H)$$

(Man nehme im Beispiel $w_1' := w$ und $w_2' := w'$ aus Beispiel 4.26.)

Dann gilt auch für beliebige Interpretationen I

$$res(w_1', I) = res(w_2', H),$$

da dann die Termine gemäss I auszuwerten sind (in unserem Beispiel mit I aus Beispiel 4.32 ist $x = f_3^1(f_2^1) = 20 + 5 = 25$, $y = f_5^1(f_4^3(f_1^2)) = 1$, $z = f_4^3(f_1^2) = 1$).

Also sind w_1', w_2' und damit w_1, w_2 äquivalent. Wäre umgekehrt a) oder b) verletzt, dann enthielten die Tupel $res(w_1', H)$ und $res(w_2', H)$ in einer Komponente verschiedener Terme. Damit wären w_1', w_2' und damit auch w_1, w_2 nicht äquivalent, da $res(w_1', I) \neq res(w_2', I)$ für mindestens eine Interpretation I, nämlich gerade die Herbrand-Interpretation H gilt. □

4.2.2 Serialisierbarkeit

Definition 4.38 *Eine Ausführungsfolge $w = w_1 w_2 \dots w_{2n} \in F(AS)$ eines schematischen Auftragssystems $AS = (A, \lessdot)$ heißt* seriell, *wenn für alle $1 \leq i < 2n$ gilt:*

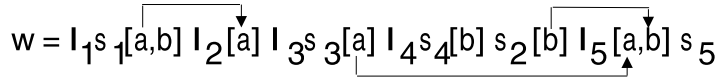
$$w_i = l_k \Rightarrow w_{i+1} = s_k$$

d.h. Leseauftrag l_k und Schreibauftrag s_k eines Auftrages a_k werden (ungeteilt) hintereinander ausgeführt.

- $w \in F(AS)$ *heißt* Serialisierung *von $w' \in F(AS)$, wenn w seriell und äquivalent zu w' ist.*
  Ersetzt man alle Paare l_i, s_i durch a_i, dann wollen wir auch die so erhaltene Folge Serialisierung von w' nennen. Diese Folge ist Ausführungsfolge der Vergröberung $\hat{AS}$ (Def. 4.28).
- $w' \in F(AS)$ *heißt* serialisierbar *(serializable), wenn w' eine Serialisierung $w \in F(AS)$ besitzt.*
- $w' \in F(AS)$ *heißt* schwach serialisierbar, *wenn w' äquivalent zu einer seriellen Ausührungsfolge $w \in F(AS')$ eines Unterauftragssystems von AS ist.*

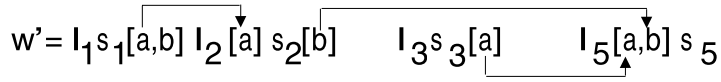
Gehen wir davon aus, dass alle Aufträge konsistente Zustände in ebensolche transformieren, dann gilt dies natürlich auch für schwach serialisierbare Ausführungsfolgen. Jede serialisierbare Folge ist schwach serialisierbar, jedoch gilt nicht die Umkehrung, wie das folgende Beispiel zeigt.

Beispiel 4.39 Für da schematische Auftragssystem in Abb. 4.33 betrachte man die folgende Ausführungsfolge w:



w' ist eine schwache Serialisierung von w (a_4 ist nutzlos in w)

Für jede Serialisierung w'' von w muss gelten:



a) $a_4 <_{w''} a_2$, da beide auf b schreiben

b) $a_2 <_{w''} a_3$, da a_2 den Wert von a liest und a_3 auf a schreibt

Aus a) und b) folgt $a_4 <_{w''} a_3$ im Widerspruch zur Präzedenz $a_3 \lessdot a_4$. Also ist w nicht serialisierbar.

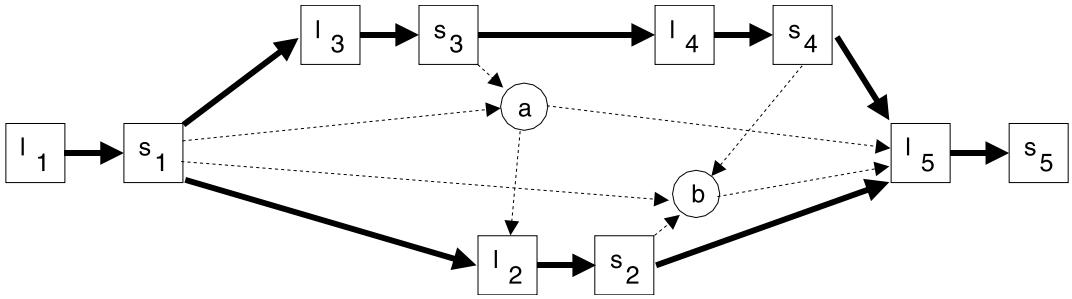
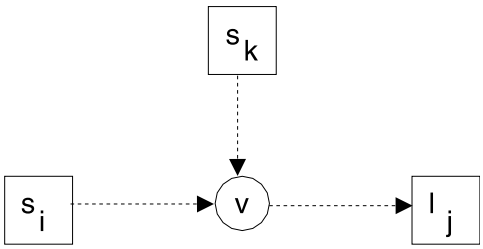


Abbildung 4.33: Schematisches Auftragssystem zu Beispiel 4.39

Wie kann geprüft werden, ob eine Ausführungsfolge w_1 schwach serialisierbar ist und somit einen konsistenten Zustand hergestellt hat? Eine (schwache) Serialisierung w_2 von w_1 muss äquivalent zu w_1 sein. Nach Satz 4.37 a) müssen ihre Vervollständigungen die gleichen relevanten Aufträge haben. Wir nehmen daher an, dass unser Auftragssystem AS vollständig war (oder gemacht wurde) und bilden das aus den für w_1 relevanten Aufträgen bestehende Unterauftragssystem $AS' = (A', \lessdot')$. Weiterhin sollen alle Aufträge in w_2 ungeteilt ausgeführt werden. zu AS' konstruieren wir daher die Vergröberung $\hat{AS}' = (\hat{A}', \lessdot')$. Als Ergebnis unserer bisherigen Schritte halten wir fest: Falls w_1 überhaupt eine schwache Serialisierung w_2 besitzt, dann muss eine solche Ausführungsfolge von $\hat{AS}'$ sein: $w_2 \in F_E(\hat{AS}')$. Nach Satz 4.37 b) müssen w_1 und w_2 außerdem die gleichen Werteübertragungsrelation haben. Wir nehmen daher W"U(w_1) als Präzedenzen zu $\lessdot'$ hinzu: $\lessdot'' := \lessdot' \cup \text{WÜ}(w_1)$. Ist $AS'' := (\hat{A}', \lessdot'')$ kein Auftragssystem (d.h. ist $\lessdot''$ nicht asymmetrisch), dann kann w_1 keine schwache Serialisierung haben. Ist AS'' jedoch ein Auftragssystem, dann muss nicht jede Ausführungsfolge $w'' \in F(AS'')$ schwache Serialisierung von w_1 sein:

Liest a_j die Variable v von a_i in w_1 und schreibt ein dritter Auftrag a_k ebenfalls auf v:



dann muss sichergestellt sein, dass a_k entweder vor a_i (also $a_k \lessdot a_i$) oder nach a_j (also $a_j \lessdot a_k$) ausgeführt wird. Wir nehmen genau eine dieser Präzedenzen in AS'' auf. Da diese Wahl später zu Zyklen in der Präzedenz-Relation führen kann, müssen wir auch die andere mögliche Wahl im Auge behalten. Präzedenzen (a_i, a_j) mit $i \in \{1, n\}$ oder $j \in \{1, n\}$ brauchen nicht hinzugenommen zu werden, da sie oder ihre Umkehrungen bereits vorhanden sind.

Wir erhalten also eine Menge von Relationen auf $\hat{A}'$, die als Graphen aufgefaßt Serialisierungsgraphen von w heißen. Ist einer dieser Graphen zyklenfrei, dann besitzt w eine Serialisierung (und nur dann).

Definition 4.40 *Sei $AS = (\{l_1, s_1, \ldots, l_n, s_n\}, \lessdot)$ ein vollständiges, schematisches Auftragssystem und $w \in F_E(AS)$ eine Ausführungsfolge.*
$\hat{AS} = (A, \lessdot), A := \{a_1, \ldots, a_n\}$ sei die Vergröberung von AS.
$\hat{AS}' = (A', \lessdot')$ sei das durch die Menge A' der für w relevanten Aufträge definierte Unterauftragssystem von AS.

Ein Graph $SG(w) = (A', \lessdot_o)$ heißt Serialisierungsgraph von w, wenn $\lessdot_o$ die transitive Hülle von $\lessdot' \cup \lessdot_1 \cup \lessdot_2$ ist, wobei folgendes gilt:

a) $\lessdot_1 := W\ddot{U}(w)_{|A'}$ *ist die Werteübertragungsrelation eingeschränkt auf A',*

b) $\lessdot_2 := \{(a_x, a_y) | a_i$ *liest v von a_j und $v \in aus_k$ und*

entweder $(a_x, a_y) = (a_k, a_j)$

oder $(a_x, a_y) = (a_i, a_k)\}$

$MSG(w)$ sei die Menge der Serialisierungsgraphen von w.

Satz 4.41 *Sei $w_1 \in F(AS)$ Ausführungsfolge eines vollständigen und relevanten schematischen Auftragssystems AS.*

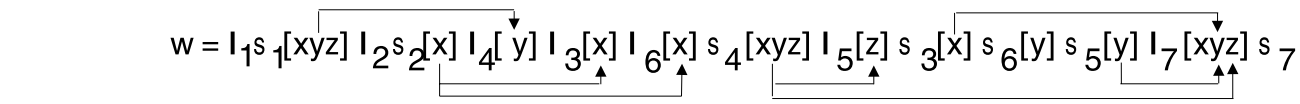
$SER(w_1) := \bigcup\{F(SG_i(w_1)) | SG_i(w_1) \in MSG(w_1)$ ist Auftragssystem$\}$ ist die Menge der Serialisierungen von w_1.

Insbesondere ist w_1 genau dann serialisierbar, wenn mindestens ein Serialisierungsgraph aus $MSG(w_1)$ ein Auftragssystem (d.h. zyklenfrei) ist.

Beweis:
Sei $w_2 \in SER(w_1)$, also $w_2 \in F_E(SG_j(w_1))$ für einen Serialisierungsgraph $SG_j(w_1)$ von w_1. $F(SG_j(w_1))$ enthält nur serielle Ausführungsfolgen. Andererseits sind die Präzedenzen gerade so definiert, dass a_j ein v von a_i genau dann in w_1 liest, wenn dies auch für w_2 gilt. Also enthalten w_1 und w_2 die gleichen relevanten Aufträge. w_1 und w_2 sind nach Satz 4.37 äquivalent. Sei umgekehrt $w_2 \in F(AS)$ eine Serialisierung von w_1. Da w_1 und w_2 äquivalent sind, haben sie nach Satz 4.37 dieselben relevanten Aufträge und die "liest v von"-Relation zwischen Aufträgen stimmt überein. Folglich muss $w_2 \in F_E(SG_i(w_1))$ für ein $SG_i(w_1) \in MSG(w_1)$ gelten. □

Hat man ein beliebiges schematischen Auftragssystem $AS = (A, \lessdot)$ und $w \in F_E(AS)$ vorliegen, dann bestimme man zunächst die für w relevanten Aufträge $A' \subset A$ (Def. 4.35). Mit Def. 4.40 und Satz 4.41 lassen sich diejenigen Auftragssysteme gewinnen, die alle Serialisierungen mit Aufträgen aus A' als Ausführungsfolgen enthalten. Damit kann auch entschieden werden, ob w schwach serialisierbar ist. Sollen die Auftragssysteme alle Aufträge von A enthalten, dann sind die für w nutzlosen Aufträge mit ihren Präzedenzen in den Serialisierungsgraphen einzufügen. Zusätzlich müssen ggf. noch Präzedenzen angebracht werden, die gewährleisten, dass diese Aufträge in jeder Ausführungsfolge nutzlos sind. w ist genau dann serialisierbar, wenn dabei ein zyklenfreier Graph, also ein Auftragssystem, entsteht.

Beispiel 4.42 Wir betrachten $AS = (A, \lessdot)$ aus Beispiel 4.29 mit der Ausführungsfolge w aus Beispiel 4.34, wo die Werteübertragungsrelation WÜ(w) angegeben ist mit:



Nach Beispiel 4.36 sind $A' = \{a_1, a_2, a_2, a_4, a_5, a_7\}$ relevant. $\hat{AS}' = (A', \lessdot')$ gemäss Def. 4.40 ist in Abb. 4.31 gegeben, wenn man dort a_6 streicht.

$$\begin{aligned} \lessdot_1 &= \mathrm{W\ddot{U}}(W)_{|A'} = \{(a_1, a_4), (a_2, a_3), (a_4, a_5), (a_4, a_7), (a_3, a_7), (a_5, a_7)\} \\ \lessdot_2 &= \{\text{entweder } (a_4, a_2) \text{ oder } (a_3, a_4), (a_4, a_3), (a_2, a_3)\} \end{aligned}$$

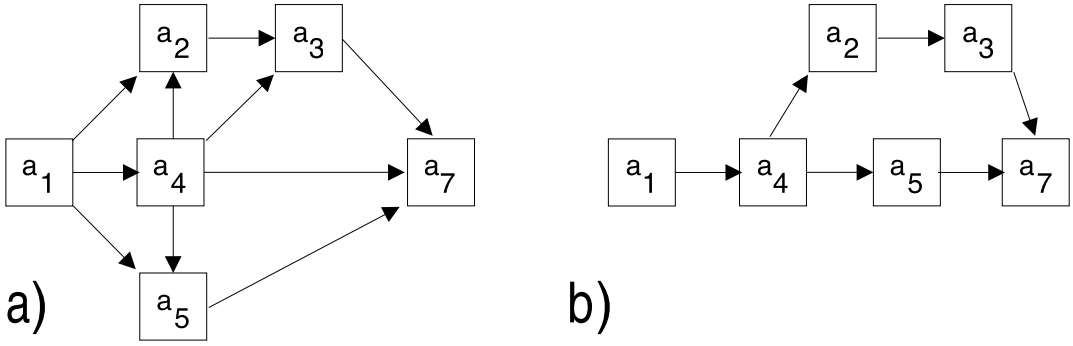


Abbildung 4.34: Serialisierungsgraph $SG_1(w)$ (ohne transitiven Kanten)

Abbildung 4.34 a) zeigt den Serialisierungsgraphen $SG_1(w)$ (für die 1. Alternative in $\lessdot_2$). $SG_2(w)$ erhält man durch Ersetzen von $a_4, a_2)$ durch (a_3, a_4). Da ein Zyklus entsteht, ist $SG_2(w)$ kein Auftragssystem.

Abbildung 4.34 b) zeigt den Konnektivitätsgraphen von $SG_1(w)$. $a_1a_4a_2a_5a_3a_7 \in F_E(SG_1(w))$ ist damit eine schwache Serialisierung von w. $F_E(SG_1(w))$ ist genau die Menge aller schwachen Serialisierungen von w aus für w relevanten Aufträgen.

Ist man an einer Serialisierung von w interessiert, dann muss der für w nutzlose Auftrag a_6 mit seinen Präzedenzen $a_2 \lessdot a_6 \lessdot a_7$ eingefügt werden. Außerdem muss man durch zusätzliche Präzedenzen sicherstellen, dass a_6 bei allen Ausführungsfolgen nutzlos ist (hier genügt z.B. $a_6 \lessdot a_5$). Das so erhaltene Auftragssystem AS_1 in Abb. 4.35 enthält alle Serialisierungen von w als Ausführungsfolgen, darunter auch die aus Beispiel 4.31 bekannte:

$$w' = a_1a_4a_2a_6a_5a_3a_7$$

.

Beispiel 4.43 Die Ausführungsfolge

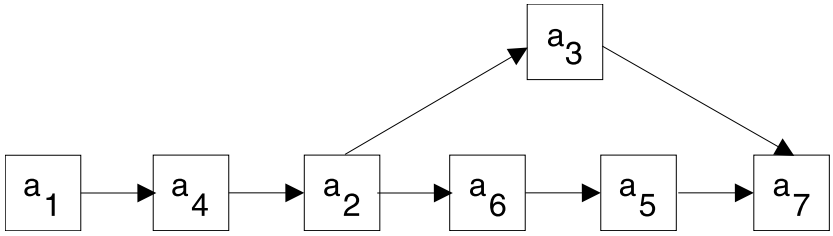


Abbildung 4.35: $SG_1(w)$ mit nutzlosem Auftrag a_6

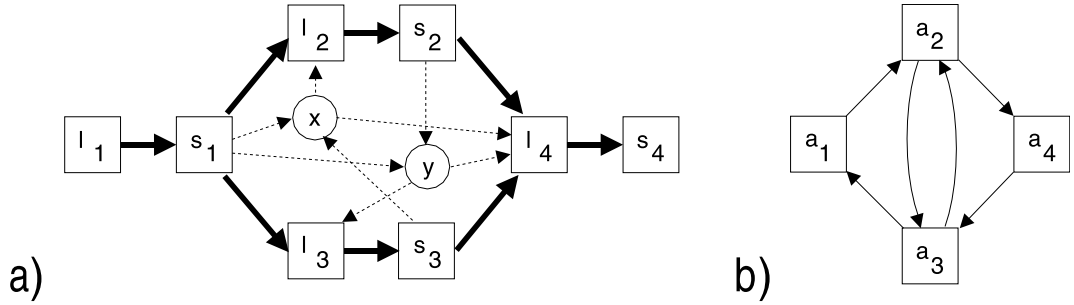


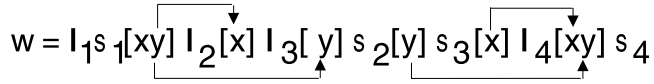
Abbildung 4.36: Auftragssystem mit zyklischem Serialisierungsgraphen

des Auftragssystems von Abb. 4.36 a) ist nicht (schwach) serialisierbar, da der (einzige) Serialisierungsgraph in Abb. 4.36 b) kein Auftragssystem ist.

Für die folgende Interpretation I_i:

$$
\begin{aligned}
&l_1 : skip\\
&s_1 : x, y := 1, 2\\
&l_2 : lo_2 := x\\
&s_3 : lo_3 := y\\
&s_2 : y := lo_2 + 1\\
&s_3 : x := lo_3 + 2\\
&l_4 : lo_4^1, lo_4^2 := x, y\\
&s_4 : skip
\end{aligned}
$$

gilt: $res(w, I_1) = (x = 4, y = 2)$. Dies ist ein konsistenter Zustand. Die Interpretation I_2 unterscheide sich von I_1 nur durch die Initialisierung $s_1' : x, y := 0, 0$. Für I_2 ist $res(w, I_2) = (x = 2, y = 1)$ jedoch inkonsitent!



Das Auftragssystem könnte aus folgendem Programm abgeleitet worden sein:

$$x, y := 1, 2;$$
$$\underline{con}\ y := x + 1 \ ||\ x := y + 2\ \underline{noc}$$

Das Problem der Serialisierbarkeiteits ist NP-vollständig (Papadimitriou 79). Damit wächst die Ausführungszeit jedes Algorithmus, der für eine Ausführungsfolge und beliebige Interpretationen feststellen soll, ob ein konsistenter Zustand erreicht wird, praktisch expotentiell. Daher wurden Teilklassen des Problems untersucht, die eine geringere, d.h. polynomiale Komplexität haen. Eine Übersicht findet man in (Papadimitriou 79; Bernstein et al 79). Dies gilt zum Beispiel schon dann, wenn man für das Auftragssystem voraussetzt, dass jede von einem Auftrag beschriebene Variable von demselben Auftrag gelesen wird (also: $aus_i \subseteq ein_i$ für alle $1 < i \leq n$). Wir behandeln hier jedoch zwei andere Teilklassen, die vom Konzept her von allgemeiner Bedeutung sind. Dabei beschränken wir uns auf Auftragssysteme, die nur Def. 4.27 vorgeschriebenen Präzedenzen enthalten und nennen diese, wie in der Literatur üblich, Transaktionen-Systeme.

4.2.3 Funktionalität

Sind in einem Auftragssystem alle Ausführungsfolgen serialisierbar, dann sind alle möglichen Resultate konsistent. Oft ist es jedoch notwendig, darüberhinaus trotz nebenläufiger Ausführungen ein einziges und damit eindeutiges Resultat aller Berechnungen sicherzustellen. Da das Resultat dann eine Funktion der Anfangswerte ist, nennen wir ein solches Auftragssystem *funktional*.

Beispiel 4.44 Für das Umordnungsprogramm von Beispiel 5.4 nehmen wir feste Anfangswerte A_0 und B_0 an. Das Programm möge dann nach $x \geq 1$ Schleifendurchläufen terminieren (den Fall $x = 0$ betrachten wir hier nicht).

Da wir an möglichst großer Nebenläufigkeit interessiert sind, formulieren wir diesen Prozess als (interpretiertes) Auftragssystem. Alle möglichen Abläufe sollen natürlich das gleiche, in Beispiel 5.4 angegebene Resultat haben.

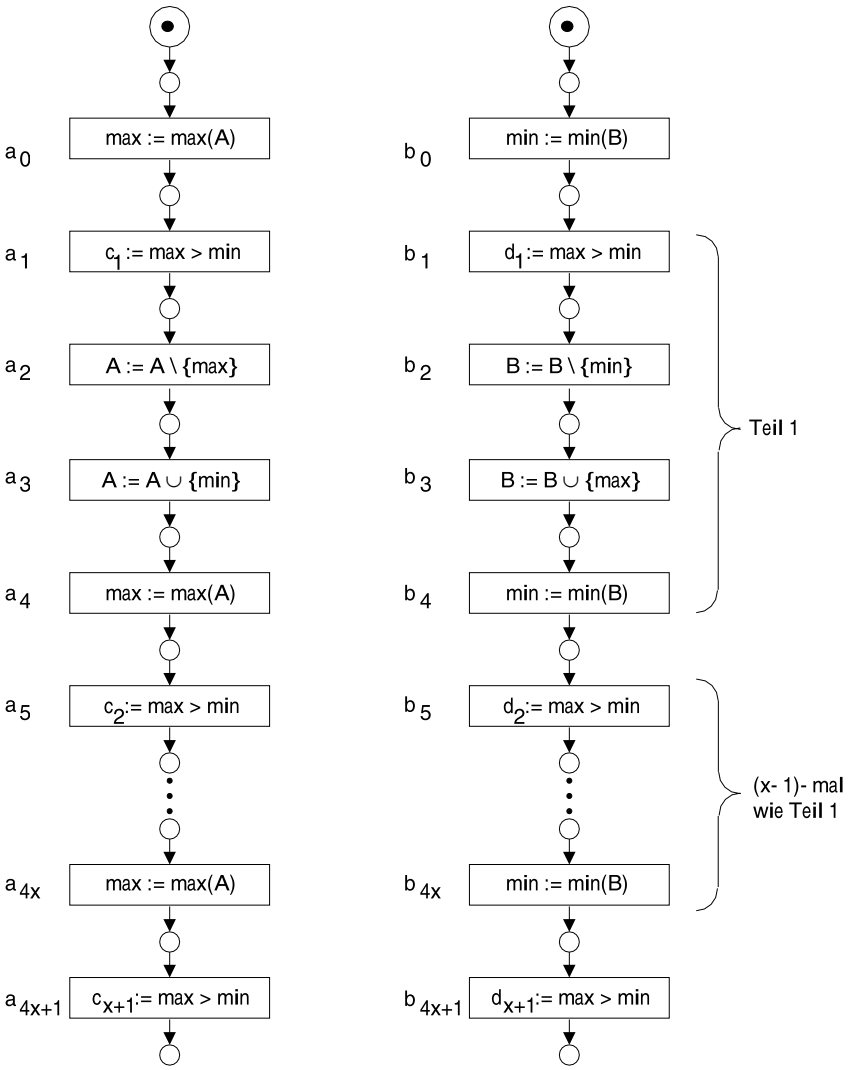


Abbildung 4.37: Interpretiertes Auftragssystem zu Beispiel 5.4

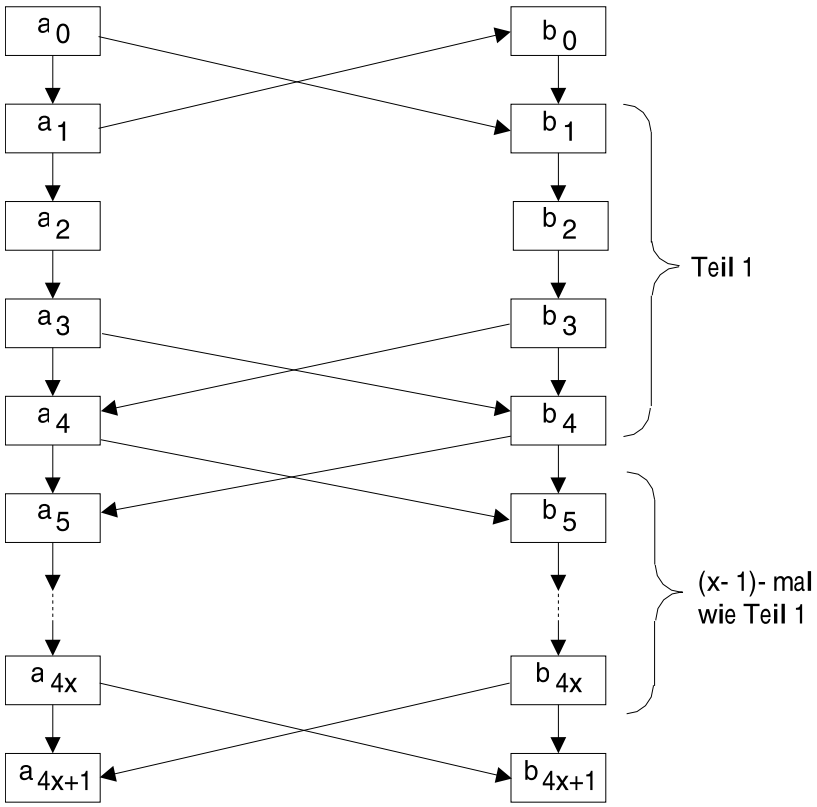


Abbildung 4.38: Für Funktionalität notwendige Präzedenzen

Da wir an unserem Verfahren zur Sicherstellung der Funktionalität interessiert sind, betrachten wir das den zugehörigen Prozess beschreibende Auftragssystem in Abbildung 4.37. Die Notwendigkeit der dort schon angegebenen Präzedenzen leuchtet unmittelbar ein. Die Aufgliederung der Aufträge in Lese- und Schreibaufträge entsprechend der Definition 4.27 eines schematischen Auftragssystems ist vorausgesetzt, aber nicht in der Abbildung dargestellt. Die Werte der verschiedenen Tests werden jeweils neuen booleschen Variablen c_i, d_j zugewiesen. Soll das Programm korrekt sein, dann muss es zumindest funktional sein.

Wir fragen, welche zusätzlichen Präzedenzen notwendig sind, um Funktionalität zu gewährleisten. Man mache sich klar, dass dazu die in Abb. 4.38 (als Konnektivitätsgraph) angegebenen Präzedenzen notwendig und hinreichend sind. Jede Ausührung des Programms von Beispiel 5.4 bei x-maligem Schleifendurchlauf ($x \geq 1$) hält diese Präzedenzen ein. Also sind die Ergebnisse aller Ausführungsfolgen (bei festen Anfangswerten A_0, B_0)

identisch und das Programm in diesem Sinne "funktional". Wir werden in diesem Abschnitt sehen, wie man diese zusätzlichen Präzedenzen von Abb. 4.38 systematisch findet. Dieses Verfahren kann dazu benutzt werden, ein nebenläufiges Programm wie in Beispiel 5.4 zu konstruieren, das größtmögliche Nebenläufigkeit zuläßt.

Bei vielen Prozessen in Rechensystemen, insbesondere bei verteilten Funktionenseinheiten ist es nicht sinnvoll, ein "Resultat" zu erwarten. Vielmehr erzeugen diese Prozesse ständig neue Werte, die sich auf die Umgebung des Prozesses auswirken. Die Folge der Werte einer Variablen nennt man *Spur*. Ist bei allen Ausführungsfolgen eines Auftragssystems die Spur jeder Variablen gleich, so nennen wir es *spurfunktional*.

Definition 4.45 *Sei AS ein schematisches Auftragssystem. AS heißt funktional, falls alle Ausführungsfolgen zueinander äquivalent sind (Def. 4.31).*

Sei $d_0, d_1, \ldots, d_{2n}$ die Zustandsfolge $w \in F(AS)$ bei einer Interpretationen I (Def. 4.31) und $v \in V$ eine Variable. Weiterhin sei $d_{i_0}, d_{i_2}, \ldots, d_{i_k}$ die Teilfolge von Zuständen, in denen der Variablen v Werte zugewiesen werden (also genau diejenigen Zustände d_{i_j} mit $w_{i_j} = s_k$ und $v \in aus_k$).

Die Folge der v-Komponenten heißt Spur (history) von v in w bei I:

$$spur(w, v, I) := d_{i_0}(v) d_{i_1}(v) \ldots d_{i_k}(v)$$

Der Vektor

$$spur(w, I) = (spur(w, v_1, I), \ldots, spur(w, v_p, I))$$

heißt Spur von w bei I.

Zwei Ausführungsfolgen $w_1, w_2 \in F(AS)$ heißen spuräquivalent, falls $spur(w_1, I) = spur(w_2, I)$ für alle Interpretationen I gilt.

AS heißt spurfunktional *(oder determiniert, determinante), falls alle Ausführungsfolgen paarweise spuräquivalent sind.*

In Satz 4.41 wurde bewiesen, dass azyklische Serialisierungsgraphen funktionale Auftragssysteme sind. Die folgenden Definitionen bereiten notwendige und hinreichende Kriterien für die Funktionaliät und Spurfunktionaliät eines Auftragssystems vor.

Definition 4.46 *Sei AS ein vollständig schematisches Auftragssystem.*

(a) Zwei Aufträge a_i und a_j von AS heißen störungsfrei, *falls $a_i \lessdot a_j$ oder $a_j \lessdot a_i$ oder $dis(a_i, a_j)$ gilt, wobei*

$dis(a_i, a_j) := (aus_i \cap aus_j = ein_i \cap aus_j = aus_i \cap ein_j = \emptyset)$ *"Bernstein"-Relation*

AS heißt störungsfrei, falls alle seine Aufträge paarweise störungsfrei sind.

(b) Ein Auftrag a_i mit $aus_i \neq \emptyset$ heißt verlustfrei. *AS heißt verlustfrei, falls alle Aufträge mit Ausnahme des Ausgabeauftrages verlustfrei sind.*

Satz 4.47 *Sei AS ein vollsändiges schematisches Auftragssystem.*

(a) AS ist genau dann funktional, wenn alle Ausführungsfolgen die gleichen relevanten Aufträge haben und diese paarweise störungsfrei sind.

(b) AS ist genau dann spurfunktional, wenn alle verlustfreien Aufträge von AS paarweise störungsfrei sind.

Beweis:
Wir beweisen (a) und erwähnen die Modifikationen für (b) in Klammern.

Es seien zunächst alle relevanten (verlustfreien) Aufträge von $AS = (A, \lessdot)$ paarweise störungsfrei.

Für beliebige Ausführungsfolgen $w_1, w_2 \in F(AS)$ müssen wir zeigen, dass sie (spur-)äquivalent sind. Seien a_i und a_j zwei beliebige relevante (verlustfreie) Aufträge von AS. Liest a_j eine Variable v von a_i in w_1, dann muss wegen $w \in aus_i \cap ein_j \neq \emptyset$ entweder $a_i \lessdot a_j$ oder $a_j \lessdot a_i$ gelten.

Zum Beispiel im ersten Fall $a_i \lessdot a_j$ muss s_i auch in w_2 vor l_j stehen, d.h. $s_i \lessdot_{w_2} l_j$. Schreibt ein dritter relevanter (verlustfreier) Auftrag a_k auf v, dann gilt nach der Definition von "a_j liest v von a_i" entweder $s_k <_{w_1} s_i$ oder $l_j \lessdot_{w_1} s_k$. Da a_k und a_i bzw. a_k und a_j störungsfrei sind, gilt $a_k \lessdot a_i$ oder $a_j \lessdot a_k$ und folglich auch $s_k <_{w_2} s_i$ oder $l_j <_{w_2} s_k$. Damit ist bewiesen, dass unter den relevanter (verlustfreien) Aufträgen der Auftrag a_j ein v von a_i bei w_1 genau dann liest, wenn dies auch bei w_2 gilt. Nach Def. 4.35 müssen

dann w_1 und w_2 auch die gleichen relevanten Aufträge haben. w_1 und w_2 sind damit äquivalent (Satz 4.37).

(Sei a_i ein verlustfreier Auftrag mit $v_i \in aus_i$. Weiterhin sei $\hat{w}_1$ bzw. $\hat{w}_1$ dasjenige Anfangsstück von w_1 bzw. w_2 das mit s_i endet. $\hat{w}_1$ und $\hat{w}_2$ haben dann die gleichen relevanten Aufträge und sind nahc Satz 4.37 äquivalent. Insgesamt sind w_1 und w_2 und damit auch w_1 und w_2 spuräquivalent.)

Sei umgekehrt AS nun funktional (spurfunktional) und a_i, a_j seien zwei für AS relevante (verlustfreie) Aufräge.

Folglich muss a_i für ein $w_i \in F_E(AS)$ relevant (a_i für ein Anfangsstück w_i eines $\hat{w}_i \in F_E(AS)$ relevant) sein. Das Analoge gelte für a_j und w_j.

Wenn weder $a_i \lessdot a_j$ noch $a_j \lessdot a_i$ gilt, so müssen wir $dis(a_i, a_j)$ beweisen.

Wir nehmen zunächst $v \in aus_i \cap ein_j \neq \emptyset$ als gegeben an und führen dies zum Widerspruch.

Es gibt serielle Ausführungsfolgen w_1 und w_2 mit $a_i <_{w_1} a_j$ und $a_j <_{w_2} a_i$. Da diese zu w_i und w_j äquivalent sind, müssen in ihnen a_i und a_j relevant sein.

a_j kann v von a_i in w_1 nicht lesen, da sonst w_1 und w_2 nicht äquivalent (spuräquivalent) sein können (Satz 4.37).

Daher existiert ein Auftrag a_k mit $a_i <_{w_1} a_k <_{w_1} a_j$ und $v \in aus_k$. Dies muss sogar für jede Ausführungsfolge w mit $a_i <_w a_j$ gelten, also: $a_i \lessdot a_k \lessdot a_j$.

Daraus folgt $a_i \lessdot a_j$ im Widerspruch zur Annahme.

Der Fall $aus_j \cap ein_i \neq \emptyset$ ist natürlich genauso zu behandeln. Der Fall $aus_j \cap aus_i \neq \emptyset$ führt mit obigem w_1 und w_2 ähnlich direkt zum Widerspruch.

Also sind alle relevanten (verlustfreien) Aufträge a_i, a_j $(i \neq a_j)$ störungsfrei. □

Beispiel 4.48

Für die serielle Ausführungsfolge

$$w = l_1 s_1 l_2 s_2 \ldots l_8 s_8$$

$$
\begin{array}{lll}
AS = (\{ & l_1[v_1, v_2], & s_1[v_3], \\
& l_2[v_1], & s_2[v_4], \\
& l_3[v_3, v_4], & s_3[v_1], \\
& l_4[v_3, v_4], & s_4[v_5], \\
& l_5[v_4], & s_5[v_2], \\
& l_6[v_5], & s_6[v_5], \\
& l_7[v_1, v_2, v_4], & s_7[v_4], \\
& l_8[v_1, v_3], & s_8[v_5] \ \}, \ \lessdot \ )
\end{array}
$$

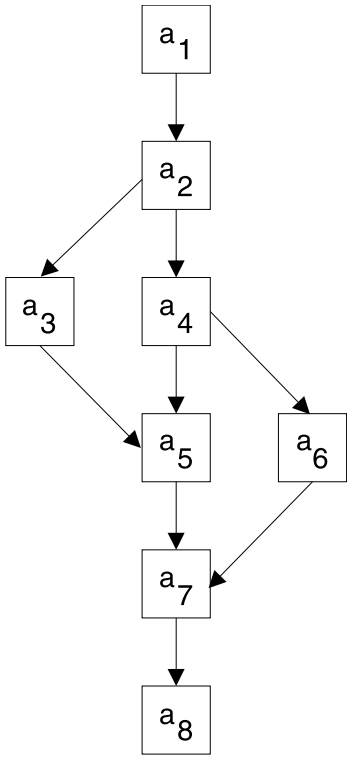


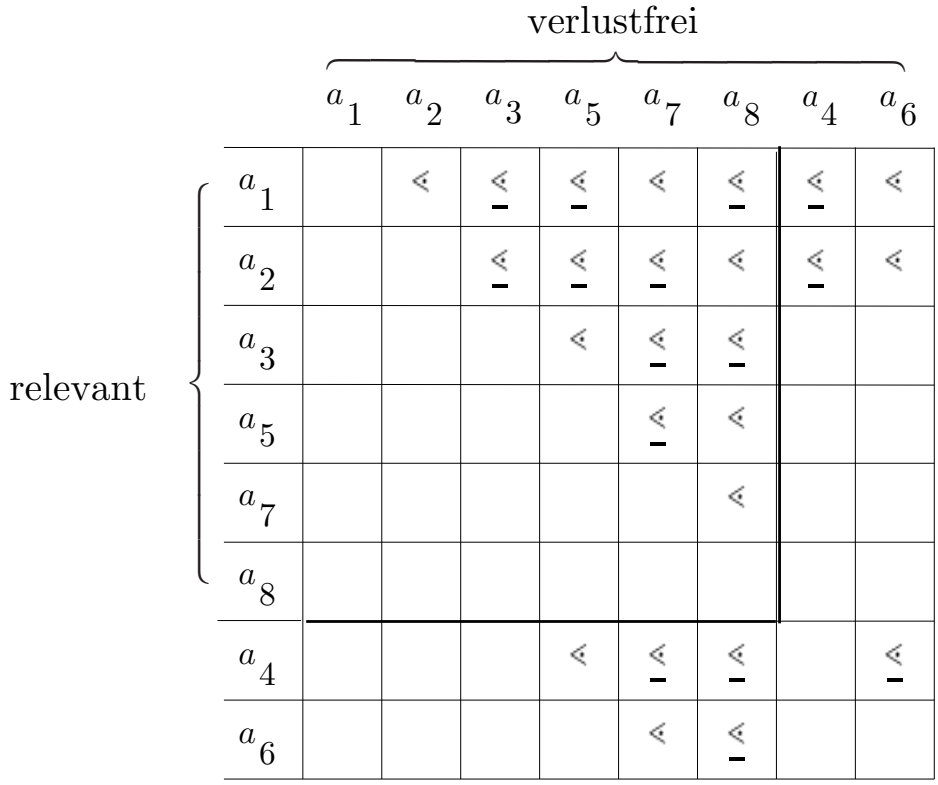
Abbildung 4.39: Ein Auftragssystem

sind die Aufträge $a_1, a_2, a_3, a_5, a_7, a_8$ relevant, nicht jedoch a_4 und a_6. (AS zwischenzeitlich vervollständigen!) a_4 und a_6 schreiben nur auf die Variable v_5, die in jeder Ausführungsfolge $w' \in F(AS)$ nachfolgend von a_8 überschriebend wird und zwar unabhängig vom alten Wert. a_4 und a_6 sind also nutzlos, was allein durch die Präzedenzen $a_4 \lessdot a_8$ und $a_6 \lessdot a_8$ sichergestellt ist.

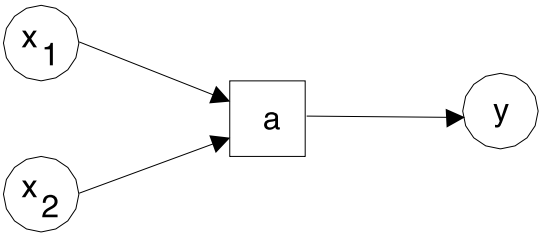
Um Satz 4.47 zu benutzen, berechnen wir die Relation $dis(a_i, a_j)$ für alle $a_i \neq a_j$ und kennzeichnen ihre Ungültigkeit durch (-) in der folgenden Tabelle. (nur oberhalb der Diagonale eingetragen!)

Da für alle relevanten (verlustfreien) Aufträge $a_i \neq a_j$ gilt: $dis(a_i, a_j) \vee a_i \lessdot a_j \vee a_j \lessdot a_i$ (d.h. falls (-), dann auch ($\lessdot$) für jeden Eintrag), ist AS funktional (und spurfunktional). Entsprechend überprüfe man, dass AS von Abb. 4.38 funktional und spurfunktional ist.

Um mehere Implikationen in einer Graphik darzustellen, hat sich folgende durch die



Netztheorie gestützte Schreibweise bewährt:

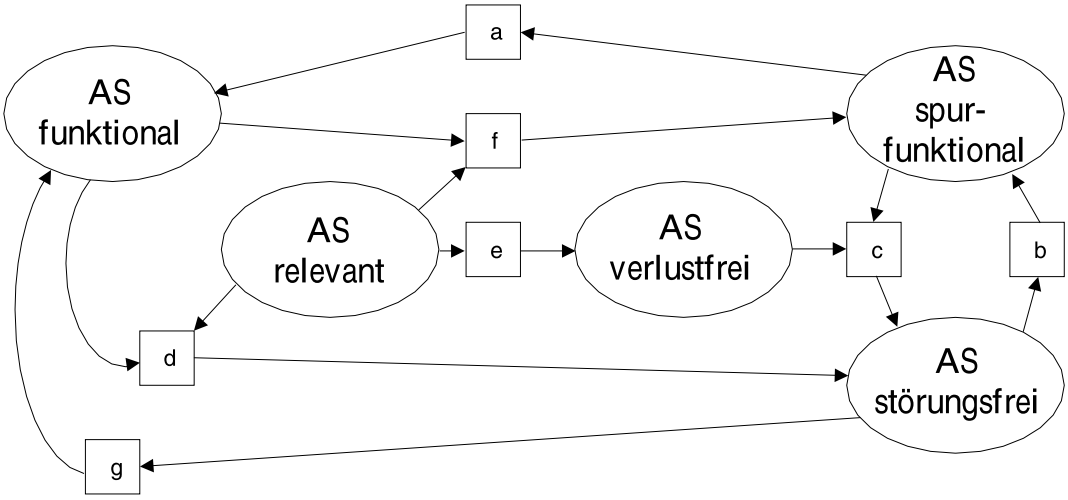


für die Behauptung a :

$x_1 \wedge x_2 \Rightarrow y$ (4.5)

Dies ist insbesondere für die folgende Übersicht unserer Ergebnisse nützlich. (An ein Schalten solcher Transitionen ist hierbei nicht zu denken!)

Satz 4.49



Beweis:

a) Spuräquivalente Folgen sind auch äquivalent.

b) Sind alle Aufträge störungsfrei, dann auch alle verlustfreien. Nach Satz 4.47 b) ist also AS spurfunktional.

c) (nach Satz 4.47 (b))

d) Sind alle Aufträge relevant und ist AS funktional, dann sind alle Aufträge störungsfrei (Satz 4.47 (a)).

e) Relevante Aufträge müssen verlustfrei sein.

f) Folgt aus d) und b).

g) Folgt aus a) und b).

□

Im Beispiel 4.48 wurde deutlich, dass nicht alle Präzedenzen eines Auftragssystems zur Gewährleistung von Funktionalität bzw. Spurfunktionalität notwendig sind. Ist dies doch der Fall, dann spricht man von einem Auftragssystem mit minimaler Präzedenzrelation oder maximaler Nebenläufigkeit bezüglich der entsprechenden Eigenschaft.

Definition 4.50 *Ein schematisches Auftragssystem $AS = (A, \lessdot)$ heißt maximal nebenläufig für Funktionalität (bzw. für Spurfunktionalität), wenn AS einerseits funktional (bzw. spurfunktional) ist, andererseits das Entfernen einer beliebigen Präzedenz $(a_i, a_j) \in \lessdot$ aus der Präzedenzrelation diese Eigenschaft zerstören würde.*

Nach folgendem Verfahren läßt sich mit Satz 4.47 ein (spur-)funktionales Auftragssystem $AS = (A, \lessdot)$ in ein für (Spur-)Funktionalität maximal nebenläufigen Unterauftragssystem $AS'(A', \lessdot')$ verwandeln, dessen Ausführungsfolgen alle (spur-)äquivalent zu denjenigen von AS sind.

Dazu bestimme man zunächst die Menge $A' \subseteq A$ der relevanten (verlustfreien) Aufträge, ersteres z.B. anhand einer beliebigen Serialisierung. Auf A' definiert man als Präzedenzrelation $\lessdot'$ die transitive Hülle von

$$\lessdot'' = \{(a_i, a_j) \in A' \times A' | a_i \lessdot a_j \text{ und } \neg dis(a_i, a_j)\} \tag{4.6}$$

$AS' = (A', \lessdot')$ hat nur relevante (verlustfreie) Aufträge, die paarweise störungsfrei sind. Nach Satz 4.47 ist also AS' funktionales (spurfunktional). Das Entfernen einer beliebigen Präzedenz $(a_i, a_j) \in \lessdot'$ würde wegen $\neg dis(a_i, a_j)$ diese Eigenschaft zerstören., Also ist AS' maximal nebenläufigs für Funktionalität (Spurfunktionalität).

Jede Folge $w_1 \in F_E(AS')$ ist äquivalent (spuräquivalent) zu einer beliebigen seriellen Folge $w_2 \in F_E(AS')$. Diese ist wiederum äquivalent (spuräquivalent) zu allen Folgen $w_3 \in F_E(AS')$.

Beispiel 4.51 Wir konstruieren zu $AS = (A, \lessdot)$ in Beispiel 4.48 ein maximal nebenläufiges Auftragssystem:

a) $AS' = (A', \lessdot')$ ist maximal nebenläufig für Funktionalität, wobei $A' = \{a_1, a_2, a_3, a_5, a_7, a_8\}$ die Menge der relevanten Aufträge ist und $\lessdot'$ aus der transitiven Hülle derjenigen Paare $(a_i, a_j) \in A' \times A'$ besteht, die in der Tabelle von Beispiel 4.48 einen Eintrag ($\underline{\lessdot}$) haben (siehe Abb. 4.40)

b) Bildet man $AS'' = (A, \lessdot'')$ mit allen Aufträgen A und $\lessdot'' = \lessdot'' \cup \{(a_4, a_8), (a_6, a_8)\}$, dann ist AS'' ebenfalls maximal nebenläufig bezüglich Funktionalität. a_4 und a_6 sind dann nähmlich nutzlos (vgl. Abb. 4.41).

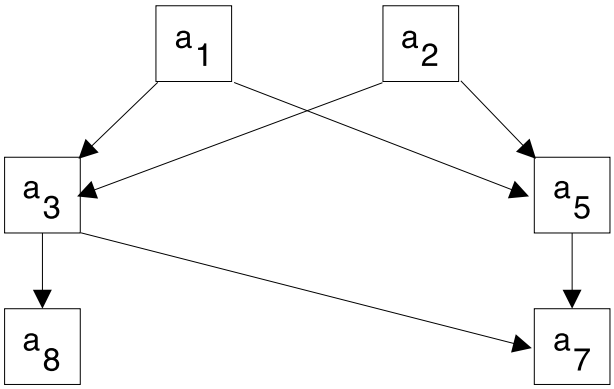


Abbildung 4.40: Für Funktionalität maximal nebenläufiges Auftragssystem

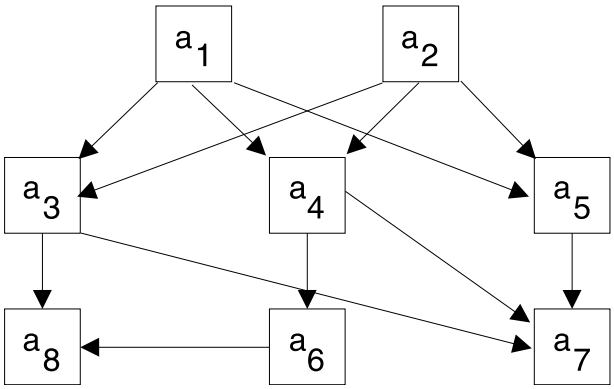


Abbildung 4.41: Für Spurfunktionalität maximal nebenläufiges Auftragssystem

c) $AS'' = (A, \lessdot'')$ ist maximal nebenläufig für Spurfunktionalität, wobei $\lessdot''$ aus der transitiven Hülle derjenigen Paare $(a_i, a_j) \in A \times A$ besteht, die in der Tabelle von Beispiel 4.41 einen Eintrag ($\underline{\leq}$) haben (alle Aufträge sind verlustfrei) (siehe Abb. 4.41).

Auch das Auftragssystem AS von Abb. 4.38 ist maximal nebenläufig für Funktionalität.

Kapitel 5

Atomizität und Kommunikation

5.1 Einleitung

Schlüsselbegriffe dieses Kapitels sind *Synchronisation*, *Atomizität* und *Kommunikation*. Bei unterschiedlicher Bedeutung stehen sie doch in starker Wechselwirkung. In den Kapiteln 2 und 3 wurden mit Prozessalgebra und Petrinetzen Modelle zur Darstellung von Synchronisation behandelt. Im vierten Kapitel spielte Synchronisation nur indirekt eine Rolle. Vielmehr wurde dort die Auswirkung von nebenläüfigen Aktionen auf die Datenkonsistenz untersucht. Die Ergebnisse dieses Kapitels untermauern die intuitive Vorstellung, dass zur Gewährleistung von Datenkonsistenz Synchronisation notwendig sein kann. Ein Beispiel für solche Synchronisationsanforderungen ist das im Abschnitt 4.1.1 dargestellte Leser/Schreiber-Problem. Geläufiger ist jedoch, dass Synchronisation zur Vermeidung von Problemen (z.B. Verklemmung, Nichtlebendigkeit) bei knappen Betriebsmitteln eingesetzt wird. Dies wurde bereits am Beispiel des Bankiersproblems (Abschnitt 3.5) behandelt.

Wir unterscheiden daher Synchronisationsformen *nach dem Ziel* als

- *Konsistenzsynchronisation* (consistency synchronisation) und
- *Betriebsmittelsynchronisation* (resource driven synchronisation).

(In der Literatur ist für den ersten Begriff auch in der Bedeutung etwas abweichend *Kooperationssynchronisation* (cooperative synchronisation) zu finden). Diese (und ggf.

weitere) Ziele können durch unterschiedliche Synchronisationsmechanismen erreicht werden und hängen von der jeweiligen Systemumgebung (Architektur, Betriebssystem, Programmsprache) ab. Generell werden wir Synchronisationsformen *nach dem Mittel* unterscheiden als

- *Speicher-Synchronisation* (shared memory synchronisation) ,
- *Rendezvous-Synchronisation* (rendezvous synchronisation) und
- *Nachrichten-Synchronisation* (message passing synchronisation).

Synchronisation wird z.B. eingesetzt, um Aktionen ohne überlappende Eingriffe von anderen und nebenläufigen Aktionen zu realisieren. Solche Aktionen werden als *unteilbar* oder *atomar* bezeichnet. Unteilbarkeit (bzw. Atomizität) ergibt sich oft in natürlicher Weise durch (unterdrückte) Kommunikation. Zudem steht die Kommunikation von Prozessen (bzw. Funktionseinheiten, Agenten) in engem Zusammenhang mit den oben erwähnten Synchronisationsformen. Die Begriffe *Synchronisation*, *Atomizität* und *Kommunikation* werden daher in diesem Kapitel im Zusammenhang behandelt.

Zu diesem Zwecke ist es nützlich eine abstrakte programmsprachliche Notation zu verwenden, in der die genannten Erscheinungen kompakter zu formulieren sind als in implementierten Sprachen wie z.B. Java, C^{++} oder Ada. Auf solche Sprachen wird aber in einem eigenen Abschnitt eingegangen. Die hier benutzte abstrakte Sprache $PROG$ basiert im Wesentlichen auf den von Dijkstra [Dij75] eingeführten *geschützten Anweisungen* (guarded commands). Diese traten schon in Kapitel 2 bei Prozessalgebra-Ausdrücken auf und stehen in enger Beziehung zur Semantik der Petrinetze. Die Sprache $PROG$ wurde unter dem Aspekt der Verifikation von D. Gries [Gri81] für sequentielle Programme behandelt. Die später hinzugefügten Rendezvous-Mechanismen orientieren sich an C. A. R. Hoare [Hoa78].

Definition 5.1 *Die Programmiersprache PROG bestehe aus folgenden Anweisungstypen:*

1. *Variablen können durch*

$$\underline{var}\ x\ :< typ >$$

vereinbart werden. Mit

$$\underline{var}\ x\ :=\ c\ :< typ >$$

kann x mit dem Wert c initialisiert werden.

2. *Eine Mehrfachzuweisung (multiple assignment) hat die Form*

$$x_1, x_2, \ldots, x_n\ :=\ e_1, e_2, \ldots, e_n$$

wobei $x_1, \ldots, x_n$ verschiedene Variable und $e_1, \ldots, e_n$ Ausdrücke sind. Sie wird ausgeführt, indem erst alle Ausdrücke $e_1, \ldots, e_n$ ausgewertet werden und dann der Wert von e_i der Variablen x_i (mit passendem Typ) zugewiesen wird. (Damit ist z.B. $x, y := x + 1\ ,\ x - 1$ nicht äquivalent zu $x := x + 1\ ;\ y := x - 1$!)

Soweit nicht anders vermerkt betrachten wir Mehrfach-Zuweisungen als Beschreibungen von unteilbaren (ununterbrechbaren) Handlungen. Ist $n = 1$, dann sprechen wir von einer Zuweisung (assignment).

3. *Sind A_1 und A_2 Anweisungen, dann beschreibt*

$$A_1\ ;\ A_2$$

die Hintereinanderausführung (sequential composition) von A_1 und A_2. skip *ist die leere Anweisung, die keine Wirkung hat.*

4. *Die geschützte Anweisung (guarded command)*

$$B\ \rightarrow\ A$$

besteht aus einem boole'schen Ausdruck B (ihrem Schutz, ihrer Schutzbedingung) (guard), und einer Anweisung A (ihrer Anweisung). A wird nur ausgeführt, wenn der (unteilbar ermittelte) Wert von B wahr ist.

5. *Geschützte Anweisungen können, in einer Auswahl-Anweisung (select command, alternative command) der Form*

$$\underline{if}\ B_1 \to A_1 \ \square\ B_2 \to A_2 \ \square\ \ldots\ \square\ B_n\ \to\ A_n\ \underline{fi}$$

mit $n \geq 1$ zusammengefaßt werden. Sie wird ausgeführt, indem genau eine beliebige, aber ausführbare, ihrer geschützten Anweisungen $B_i \to A_i$ $(l \geq i \geq n)$ ausgeführt wird. Ist keine ihrer geschützten Anweisungen ausführbar, dann ist die Auswahlanweisung nicht ausführbar Bei $n > 1$ benutzen wir die Bedingung $B_n = else$ als Abkürzung für $\neg(B_1 \vee \ldots \vee B_{n-1})$.

(Zum Beispiel können in $\underline{if}\ (x \geq y \wedge y > 0) \to z := x \ \square\ x \leq y \to z := y\ \underline{fi}$ bei den Werten $(2,1)$, $(1,1)$ bzw $(2,0)$ für (x,y) jeweils eine, zwei bzw. keine Auswahl getroffen werden.)

6. *Eine Schleifen-Anweisung (iterative command) hat die Form*

$$\underline{do}\ B_1 \to A_1 \ \square\ B_2 \to A_2 \ \square\ \ldots\ \square\ B_n \to A_n\ \underline{od}$$

mit $n \geq 1$. Es wird solange die in do od *eingeschlossene Auswahlanweisung wiederholt, wie diese ausführbar ist. Ist diese nicht mehr ausführbar, dann wird die Schleife abgebrochen und die nächste Anweisung ausgeführt.*

7. *Die Nebenlauf-Anweisung (concurrent command) hat die Form*

$$\underline{con}\ A_1 \parallel A_2 \parallel\ \ldots\ \parallel A_n\ \underline{noc}$$

mit $n \geq 1$. Sie wird ausgeführt, indem jede der Anweisungen A_i $(1 \leq i \leq n)$ unabhängig ausgeführt wird. Erst wenn alle A_i ausgeführt sind, kann die nächste Anweisung ausgeführt werden. Enthält A_i die Anweisung A_i' und A_j die Anweisung A_j', dann heißen A_i' und A_j' (wie natürlich auch A_i und A_j) nebenläufig (concurrent), falls $i \neq j$

8. *Eine Zusicherung B (assertion) ist eine Bedingung, die in der Form*

$$(*\qquad B \qquad *)$$

vor oder nach Anweisungen stehen kann. Damit wird behauptet, daß B gilt, wenn die Ausführung diese Stelle erreicht und verläßt (unabhängig von eventuellen nebenläufigen Handlungen).

Zum Abschluß noch einige Beispiele:

Beispiel 5.2

$$
\begin{array}{l}
(* \quad a, b \text{ integer}, b \geq 0 \quad *) \\
\underline{var}\ x, y\ :=\ a, b : \underline{integer}, \underline{var}\ z\ :=\ 0 :\ \underline{integer;} \\
\underline{do}\ y > 0 \wedge \text{even}(y) \quad \rightarrow y, x\ :=\ y \div 2, x + x \\
\square\ \text{odd}(y) \qquad\qquad\ \rightarrow y, z\ :=\ y - 1, z + x \\
\text{od} \\
(* \quad z = a \cdot b \quad *)
\end{array}
$$

Dieses Programm berechnet das Produkt von a und b ohne Benutzung der Multiplikations-Operation (unter Benutzung der ganzzahligen Division $\div$).

Beispiel 5.3 Gegeben seien drei sequentielle Dateien f, g, h unbekannter Länge mit Namen in alphabetischer Reihenfolge. Es soll der erste Name gefunden werden, der in allen drei Dateien auftaucht. Die Dateien werden als unbeschränkte Felder betrachtet, $<$ bezeichne die alphabetische Reihenfolge. Das folgende Programm [Gri81] löst die Aufgabe in geschickter Ausnutzung der Schleifenanweisung.

$$
\begin{array}{l}
\underline{var}\ i, j, k := 0, 0, 0 : \underline{integer}, \\
\underline{var}\ f, g, h : \underline{array}\ [0..?]\ \underline{of}\ \underline{string}; \\
\underline{do}\ f[i] < g[j] \rightarrow i := i + 1 \\
\square\ g[j] < h[k] \rightarrow j := j + 1 \\
\square\ h[k] < f[i] \rightarrow k := k + 1 \\
\underline{od} \\
(* \quad r[i] = g[j] = h[k] \quad *)
\end{array}
$$

Die Schleifenanweisung terminiert, wenn $f[i] \geq g[j]$, $g[j] \geq h[k]$, $h[k] \geq f[i]$, also mit $f[i] = g[j] = h[k]$, wie gewünscht.

Beispiel 5.4 Gegeben seien zwei disjunkte endliche Mengen A und B von ganzen Zahlen. Sie sollen so in Mengen A und B jeweils gleicher Mächtigkeit umgeformt werden,

dass alle Elemente in A kleiner als alle Elemente in B sind.

$(* \quad A = A_0 \subset \mathbb{Z}, B = B_0 \subset \mathbb{Z} :, A \text{ und } B \text{ endlich und disjunkt} \quad *)$

$\underline{var} A, B : \underline{set\ of\ integer},$
$\underline{var}\ max, min : \underline{integer};$
$max, min \rightarrow max(A), min(B);$
$\underline{do}\ max > min \rightarrow$
  $\underline{con}\ A := A \setminus \{max\}; A := A \cup \{min\}$
  $\parallel\ B := B \setminus \{min\}; B := B \cup max$
  $\underline{noc};$
  $max, min := max(A), min(B)$
$\underline{od}$

$(* \quad A \cup B = A_0 \cup B_0, A \cap B = \emptyset, |A| = |A_0|, |B| = |B_0|, max(A) < min(B) \quad *)$

Das Programm [Dij77] benutzt die Operationen max(A), min(B) für das maximale bzw. minimale Element einer Menge.

5.2 Wechselseitiger Ausschluss

Sollen bei der Ausführung von nebenläufigen Anweisungsfolgen inkonsistente Ergebnisse ausgeschlossen werden (vergl. Seite 27 oder die Einleitung zu Kapitel 4), so muss verhindert werden, dass sich gewisse Teilfolgen überlappen. Wir wollen nun versuchen, in unserer Programmiersprache PROG solche *kritischen Abschnitte* (critical regions) zu realisieren. Man spricht dann vom *wechselseitigen Ausschluss* (mutual exclusion) der kritischen Abschnitte. Anweisungen, die die Synchronisation oder Kommunikation zwischen Programmen regeln, heißen oft *Protokolle*.

Definition 5.5 *Anweisungen oder Spezifikationen, die die Synchronisation oder Kommunikation zwischen Programmen regeln, heißen auch* Protokolle *oder* Kommunikationsprotokolle.

Synchronisation von wechselseitigem Ausschluß soll nun in Form eines Programmes

$$P : A_0 \; ; \underline{con} \;\; A_1 \;\; \| \;\; A_2 \;\; \underline{noc} \tag{5.1}$$

untersucht werden. A_0 ist eine Initialisierungsanweisung. A_1 und A_2 bestehen aus einer Schleife, in der sich die Benutzung des kritischen Abschnitts (z.B. mit Zugriff auf gemeinsame Variable) abwechselt mit Anweisungen, die nur lokale Daten benutzen ("lokale Aktionen"). Eintritt und Austritt am kritischen Abschnitt werden durch die Anweisungen des *Eintrittsprotokolls* bzw. des *Austrittsprotokolls* geregelt:

$A_i : \; (i \in \{1, 2\})$
$\underline{do}\ true \rightarrow$
  $Eintrittsprotokoll$;
  $kritischerAbschnitt$;
  $Austrittsprotokoll$:
  $lokale\ Aktionen\ von\ A_i$
$\underline{od}$.

Bei diesen Programmen gehen wir davon aus, dass alle elementaren Anweisungen (also Zuweisungen und Tests) durch einen Speichersperrmechanismus nichtüberlappend ausgeführt werden.

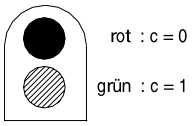
Die Programme sollen folgende Eigenschaften erfüllen:

A) Die Befehlszähler von A_1 und A_2 sind nie gleichzeitig in ihren kritischen Abschnitten.

B) Beginnt ein Programmteil A_i die Ausführung seines Eintrittsprotokolls, so kann es nach einer gewissen endlichen Zeit tatsächlich in seinen kritischen Abschnitt eintreten. Eintritts- und Austrittsprotokolle können nach endlicher Zeit verlassen werden.

Im folgenden werden vier Lösungen des Problems (z.T. nach [Dij68]) angegeben, die unterschiedlich zu bewerten sind. Das erste Programm PA orientiert sich an der Wirkungsweise von Verkehrsampeln ($c = 0$ für rot und $c = 1$ für $grün$).

Welche der Programme erfüllen die obige Spezifikation A) und B)?



PA: $c = 1$

con		‖	noc	
do	$true \rightarrow$		*do*	$true \rightarrow$
	do $c = 0 \rightarrow$ skip *od*;			*do* $c = 0 \rightarrow$ skip *od*;
	$c := 0$;			$c := 0$;
	krit. Abschnitt;			krit. Abschnitt;
	$c := 1$;			$c := 1$;
	lok. Hdlg.			lok. Hdlg.
od			*od*	

Abbildung 5.1: Programm PA

PB: $C1, C2 := 1, 1;$

con		‖	noc	
do	$true \rightarrow C1 := 0;$		*do*	$true \rightarrow C2 := 0;$
	do $C2 = 0 \rightarrow$ skip *od*;			*do* $C1 = 0 \rightarrow$ skip *od*;
	krit. Abschnitt;			krit. Abschnitt;
	$C1 := 1$;			$C2 := 1$;
	lok. Hdlg.			lok. Hdlg.
od			*od*	

Abbildung 5.2: Programm PB

```
PC: C1, C2 := 1, 1;
con                                  ||          noc
 do  true →                                  do  true →
     C1 := 0;                                    C2 := 0;
     do C2 = 0 →     C1 := 1;                    do C1 = 0 →     C2 := 1;
                     C1 := 0;                                    C2 := 0;
     od                                          od
     krit. Abschnitt;                            krit. Abschnitt;
     C1 := 1;                                    C2 := 1;
     lok. Hdlg.                                  lok. Hdlg.
 od                                          od
```

Abbildung 5.3: Programm PC

```
PD: C1, C2 := 1, 1
con                                  ||          noc
 do  true →                                  do  true →
     C1 := 0;                                    C2 := 0;
     turn := 2;                                  turn := 1;
     do (C2 = 0 ∧ turn = 2)                      do (C1 = 0 ∧ turn = 1)
       → skip od;                                  → skip od;
     krit. Abschnitt;                            krit. Abschnitt;
     C1 := 1;                                    C2 := 1;
     lok. Hdlg.                                  lok. Hdlg.
 od                                          od
```

Abbildung 5.4: Programm PD

5.3 Unteilbare Aktionen

Die in den "Lösungen" des vorangehenden Abschnitts aufgetretenen Probleme beruhen auf der Unmöglichkeit zwei aufeinander folgende Anweisungen "ungeteilt" auszuführen, das heißt so, dass keine nebenläufige Anweisung zwischen ihnen zur Ausführung kommt.

Dijkstra [Dij68] fordert, dass gewisse primitive Operationen als "unteilbare Handlungen" (indivisible actions) betrachtet werden müssen. Brinch Hansen [BH73] schreibt sogar: "Es ist unmöglich, sinnvolle Aussagen über die Wirkungen nebenläufiger Handlungen zu machen, wenn Operationen auf gemeinsamen Variablen zum selben Zeitpunkt nicht ausgeschlossen sind. Damit basiert unser Verständnis nebenläufiger Prozesse auf der Fähigkeit, ihr Wechselwirken streng sequentiell auszuführen."

Aus der Literatur können drei Definitionen für unteilbare oder atomare Handlungen entnommen werden [Lom77]:

1. Eine Handlung ist atomar, wenn die ausführende Funktionseinheit nicht die Existenz einer anderen aktiven Funktionseinheit (d.h. keinen spontanen, von ihr nicht verursachten Zustandswechsel) wahrnehmen kann, und ebenso keine andere Funktionseinheit die Ausführung der atomaren Handlung wahrnehmen kann.

2. Eine Handlung ist atomar, wenn die Funktionseinheit während ihrer Ausführung nicht mit anderen Funktionseinheiten kommuniziert.

3. Handlungen sind atomar, wenn sie bezüglich der sie bewirkenden Zustandswechsel als unteilbar und augenblicklich angesehen werden können.

Das Konzept der atomaren Handlung kann zurückverfolgt werden bis zu den Arbeiten von Floyd [Flo67] und Hoare [Hoa69] zur Korrektheit von (sequentiellen) Programmen. Die Korrektheit eines Programmes oder einer Anweisung A wird durch zwei Aussagen oder Prädikate über dem Zustandsraum ausgedrückt: einer (Eingangs-) Zusicherung Q und einer (Resultats-) Zusicherung R. A heißt partiell korrekt bezüglich Q, und R (geschrieben $(* \; Q \; *)A(* \; R \; *)$, wenn A so auf dem Zustandsraum arbeitet, dass nach seiner Ausführung R gilt, vorausgesetzt, dass vorher Q galt. (Beispiele: die Programme in den Beispielen (5.2), (5.3) und (5.4) mit angegebenen Zusicherungen Q, und R.) Damit ist die Wirkungsweise von A beschreibbar durch die Betrachtung des Zustandsraumes unmittelbar vor und unmittelbar nach der Ausführung von A. A wirkt also ohne Seiteneffekte,

oder wie eine atomare Handlung im Sinne der obigen Definition. Wir erinnern an die extensionale Beschreibung einer Handlung am Anfang des ersten Kapitels. Bei der Erweiterung der Floyd/Hoare'schen Methode der Verifikation auf nebenläufige Programme, mußte folglich im Wesentlichen das Problem gelöst werden, Wechselwirkungen mit anderen nebenläufigen Handlungen zu behandeln. Dies gelang in (Owicki et al 76) durch explizite Beweisführung, daß Wechselwirkungen die Zusicherungen P und Q gültig lassen ("sequential proofs are interference-free").

Wir wollen die Unteilbarkeit oder Atomizität von Handlungen als programmsprachliches Konzept betrachten, um dessen logische Eigenschaften unabhängig von Implementationsbesonderheiten untersuchen und benutzen zu können. Lomet schreibt [Lom77]: "Der Benutzer profitiert enorm durch die Befreiung seiner Verantwortlichkeit für dieses undurchsichtige Gebiet, was ihm erlaubt, sich auf die restlichen Programmier-Probleme zu konzentrieren."

Für jede Anweisung A in PROG sei $< A >$ die gleiche Anweisung in unteilbarer Ausführung. (Diese Notation wird seit Dijkstra [Dij78] häufig in der Literatur benutzt.)

Damit kann für die Programme P1 und P2 aus Beispiel 4.23 (Seite 99) wie im folgenden Referenz anzupassen Beispiel (5.6) durch Einführung von unteilbaren Anweisungen das einheitliche Resultat $z = 4$ für alle Prozesse sichergestellt werden.

Beispiel 5.6

$$
\begin{array}{lll}
P_1' & : & z := 1; \\
 & & \underline{con} \ < x := z + 1; z := x > \\
 & & \| \ < y := z + 2; z := y > \\
 & & \underline{noc}. \\
P_2' & : & z := 1; \\
 & & \underline{con} \ < z := z + 1 > \| < z := z + 2 > \ \underline{noc}.
\end{array}
$$

In o.g. Beispiel wäre durch

$$P_3' : \underline{con} \ < delete(x) > \| < delete(y) >$$

die unteilbare Ausführung der delete-Operation sichergestellt. Das allgemeine Problem

des wechselseitigen Ausschlusses hat nun die triviale Lösung:

$$A_i: \ \underline{do}\ true \rightarrow \ < krit.Abschnitte >; \\ \qquad\qquad\qquad\quad lokale\ Hdlg.\ von\ A_i \\ \qquad\quad \underline{od}$$

Definition 5.7 *Ist A eine Anweisung der Sprache PROG , aber nicht die Auswahl-Anweisung, dann bedeute*

$$< \quad A \quad >$$

daß A unteilbar auszuführen ist, d.h. während der Ausführung von A darf an keine in A vorkommende Variable durch eine zu A nebenläufige Anweisung A' zugewiesen werden.

Definition 5.8 *Sei A die Auswahl-Anweisung*

$$A: \ \underline{if}\ B_1 \rightarrow A_1 \ \square\ B_2 \rightarrow A_2 \ \square\ \ldots\ \square\ B_n \rightarrow A_n\ \underline{fi}$$

Dann werde $< A >$ wie folgt unteilbar ausgeführt:

1. *Gilt für ein $i \in \{1, \ldots, n\}$ $B_i = true$, dann werde für ein solches i die Anweisung $< B_i \rightarrow A_i >$ unteilbar ausgeführt.*
2. *Gilt für kein $i \in \{1, \ldots, n\}$ $B_i = true$, dann warte die $< A >$ ausführende Funktionseinheit bis (durch nebenläufige Anweisungen) Fall a) eintritt.*

Bei der im vorangehenden Abschnitt definierten Unteilbarkeit von Anweisungen handelt es sich zweifellos um ein programmier-hochsprachliches Konzept. Historisch gesehen wurden niedersprachliche und maschinennahe Kontrollstrukturen für Synchronisation viel früher entwickelt, weil auf Betriebs- bzw. Rechensystemebene nur maschinennah programmiert werden konnte.

Eine niedersprachliche Synchronisationsanweisung soll

- einfach sein, um maschinenseits eindeutig und direkt implementiert werden zu können,
- maschinenunabhängig sein, um die Portabilität von Programmen (wenigstens prinzipiell) zu erleichtern, und

- mächtig genug sein, um alle anfallenden Synchronisationsaufgaben damit bewältigen zu können.

Daraus kann man z.B. folgern, Synchronisation nur auf die Programm-Steuerung und nicht auf die Datenbehandlung einwirken zu lassen. Unsere, der Darstellung von Dijkstra [Dij68] folgenden Versuche zur Realisierung von wechselseitigem Ausschluß legen nahe, die Anweisung

$$< \underline{if}\ c > 0 \rightarrow c := c - 1\ \underline{fi} >$$

zu wählen.

Eine solche Variable c soll natürlich für die vorgesehene Signalfunktion reserviert sein und wird daher als besonderer Typ *"Semaphor"* (griech. "Zeichenträger", "Signalmast", männlich oder sächlich (vgl. DUDEN)) eingeführt.

Definition 5.9 *Ein Semaphor ist eine Variable, auf die nach ihrer Initialisierung nur durch die folgenden Anweisungen* P(sem) *und* V(sem) *zugegriffen werden darf:*

$$\begin{array}{ll} P(sem): & < \underline{if}\ sem > 0 \rightarrow sem : -sem - 1\ \underline{fi} > \\ V(sem): & < sem := sem + 1 > . \end{array}$$

Semaphor-Variablen haben nichtnegative Integerwerte (cardinal): $\{0, 1, 2, 3, \ldots\}$.

Semaphore wurden durch Dijkstra [Dij68] in die Literatur eingeführt und stammen (nach seiner Darstellung) von C. S. Scholten. P und V sind abgeleitet von den niederländischen Wörtern "proberen" und "verhogen". Zuweilen werden sie auch durch WAIT und SIGNAL ersetzt.

Das folgende Programm zeigt die Realisierung von wechselseitigem Ausschluß mit Semaphoren.

Beispiel 5.10 P: var sem:= 1 : semaphore,

con			‖			noc
A_1 :	*do*	$true \rightarrow$		A_2 :	*do*	$true \rightarrow$
		$P(sem)$;				$P(sem)$;
		krit. Abschnitt;				krit. Abschnitt;
		$V(sem)$;				$V(sem)$;
		lok. Hdlg. von A_1				lok. Hdlg. von A_2
	od				*od*	

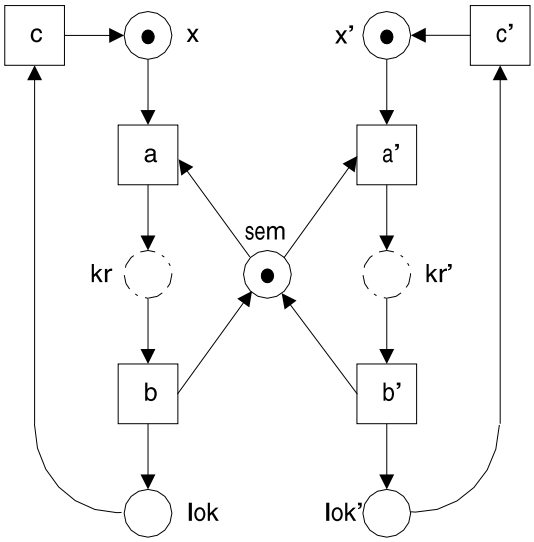


Abbildung 5.5: Netzdarstellung eines Semaphor

Abbildung 5.5 zeigt eine Netzdarstellung von Beispiel (5.10). Die Transitionen a und a' bzw. b und b' entsprechen der Wirkung der P- bzw. V-Operation auf den Semaphor *sem*. Für alle erreichbaren Markierungen $\mathbf{m}$ gilt:

$$\mathbf{m}(kr) + \mathbf{m}(kr') + \mathbf{m}(sem) = 1$$

woraus unmittelbar folgt, dass kr und kr' nicht gleichzeitig markiert sein können.

5.4 Elementare Synchronisationskonzepte

In diesem Abschnitt werden Beispiele zur Synchronisation durch Semaphore und durch andere elementare Konzepte behandelt.

Abbildung 5.6 a) zeigt ein gemeinsames Betriebsmittel “Kanal” für zwei Funktionseinheiten Sender und Empfänger als Netz. In der Verfeinerung als P/T-Netz in b) erkennt man die Unterfunktionseinheiten Erzeugen, Senden bzw. Empfangen, Verbrauchen, die die entsprechenden Handlungen in zyklischer Weise bestimmen. Das Betriebsmittel hat die Kapazität 3, d.h. der Sender kann höchstens dreimal eine Nachricht ablegen, ohne dass der Empfänger sie entnimmt.

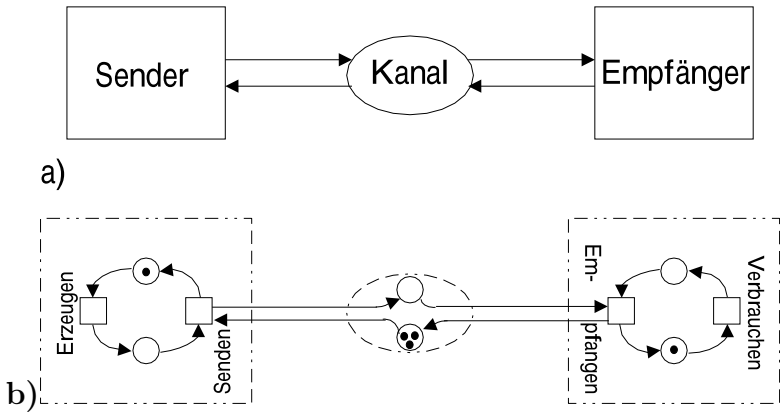


Abbildung 5.6: Synchronisation von Funktionseinheiten

Die Synchronisation wird durch die Nachricht selbst (die Marke) vorgenommen. Da ein realer Speicher oder Kanal i.a. nicht anzeigt, dass seine Füllung seiner Kapazität entspricht, muss die Synchronisation von der Datenübertragung getrennt werden.

Als Beispiel realisieren wir einen als Feld dargestellten Nachrichtenpuffer endlicher Kapazität k, der von einem Sender und einem Empfänger benutzt wird. Diese Struktur liest z.B. bei einer SPOOLing-Organisation der Ein/Ausgabe eines Rechensystems vor. Das in Abb. 5.8 dargestellte nebenläufige Programm benutzt einen Ring-Puffer der Form Abb. 5.7 als Kanal und Semaphore (Def. 5.9) zur Synchronisation, bezüglich welcher das Netz von Abb. 5.9 eine äquivalente Darstellung ist.

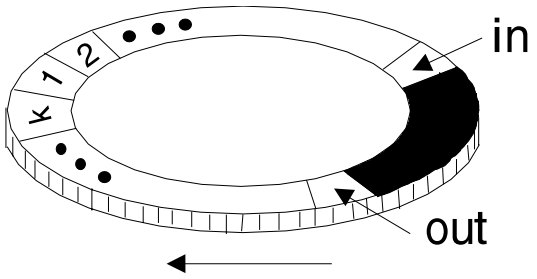


Abbildung 5.7: Ringpuffer “buffer”

Für Korrektheitsbeweise werden üblicherweise Hilfsvariable wie in und out eingeführt. Sie werden in (13) bzw. (22) wie in und out verändert, jedoch ohne die Bildung von

```
var  full:= 0, empty:= k: semaphore;
     in:=out:= l: integer; in:=out:= l:integer;
     buffer: array[l..k] of message;
     v,w: message;
con  (* Sender *)                          //        (* Empfänger *)
       do  true  →                                    do  true  →
           (10)  produce(v)                               (20)  P(full);
           (11)  P(empty)                                 (21)  w:= buffer[out]
           (12)  buffer[in]:= v;                          (22)  out:= (out+l) mod k;
           (13)  in := (in + l) mod k;                          out := out + 1
                 in := in + 1                             (23)  V(empty)
           (14)  V(full)                                  (24)  consume(w)
       od                                             od
noc
```

Abbildung 5.8: Sender/Empfänger-Programm

modulo k :

$$0 \leq \overline{in} - \overline{out} \leq k \Rightarrow \overline{in} - \overline{out} = \|in - out\|_k \tag{5.2}$$

wobei $\|in - out\|_k := (in + k - out)\ \underline{mod}\ k$.

Auch im Netz von Abb. 5.9 benutzen wir $\overline{in}$ und $\overline{out}$, um mittels der Gleichung 5.2 Aussagen über in und out zu beweisen. $\overline{in}$ und $\overline{out}$ existieren nur zu Beweiszwecken und heißen daher *Hilfsvariable*. Ihre Entfernung ändert nicht das Verhalten des übrigen Programmes bzw. Netzes.

Synchronisation bedeutet hier die zeitliche Abstimmung von Sender und Empfänger in der Weise, dass der Puffer ordnungsgemäß verwaltet wird. Das bedeutet präziser:

1. Kein Überlaufen des Puffers: $\overline{in} - \overline{out} \leq k$.
2. Kein Unterlaufen des Puffers: $\overline{in} - \overline{out} \geq 0$.
3. wenn der Sender schreibt (also (12) ausführt), ist der Puffer nicht voll (also $\|in - out\|_k < k$)

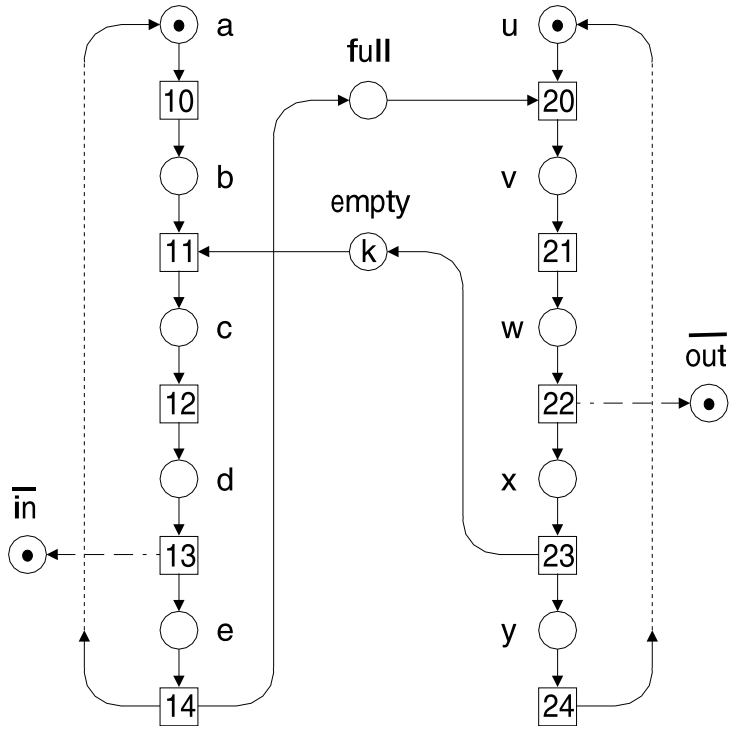


Abbildung 5.9: Sender/Empfänger-P/T-Netz

4. Wenn der Empfänger liest (also (21) ausführt), ist der Puffer nicht leer (also $\|in - out\|_k > 0$).

5. In jedem erreichbaren Zustand ist mindestens eine Anweisung ausführbar (Verklemmungsfreiheit).

Aufgabe 5.11 Bewiesen Sie diese Eigenschaften mit Hilfe von Platz-Invarianten.

Synchronisation zur Gewährleistung konsistenter Daten wird *Konsistenz-Synchronisation* genannt, im Gegensatz zur Synchronisation, bei der knappe Betriebsmittel zu verwalten sind (*Betriebsmittel-Synchronisation*). Die Implementierung von wechselseitigem Ausschluss kann je nach Situation beides sein: es kann beispielweise ein Datenbereich vor überlappendem Zugriff geschützt oder der Zugriff auf einen Drucker geregelt sein. Die Grenzen dieser Unterscheidung können aber auch weniger deutlich sein. Je nach Gesichtspunkt kann beides zutreffen. So kann die Synchronisation im Beispiel des Sender/Empfänger-Programms als Betriebsmittel-Synchronisation aufgefasst werden (das Betriebsmittel "Puffer" wird geschützt) - aber auch als Konsistenz-Synchronisation (die Daten werden ordnungsgemäß übertragen).

In jedem Fall wurden jedoch Semaphore verwendet, d.h. der Zugriff auf einen von beiden beteiligten Prozessen zugreifbaren Speicherbereich. In Hinblick auf die verwendeten Mittel wird diese Synchronisation daher als *Speicher-Synchronisation* bezeichnet. Diese Synchronisation steht im Gegensatz zur *Rendezvous-Synchronisation*, bei der sich die Partner treffen und in einer gemeinsamen Aktion die Daten austauschen oder sich synchronisieren. Eine dritte Form stellt die *Nachrichtensynchronisation* dar, bei der ein Signal oder ein Datum von einem Partner zum anderen verschickt wird.

Zur Veranschaulichung denke man an zwei Agenten (eines Geheimdienstes oder Softwareagenten), die Nachrichten austauschen müssen. Alle drei Synchronisationsformen sind üblich:

- Speicher-Synchronisation: Die Agenten verabreden einen Platz, den sie beide kennen (toter Briefkasten, gemeinsam zugreifbares Datum).
- Rendezvous-Synchronisation: Die Agenten treffen sich an einem vereinbarten Ort.
- Nachrichtensynchronisation: Die Agenden schicken die Nachricht per Bote.

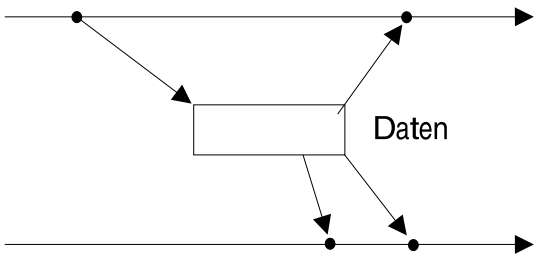


Abbildung 5.10: Speicher-Synchronisation

Abbildung 5.11: Rendezvous-Synchronisation

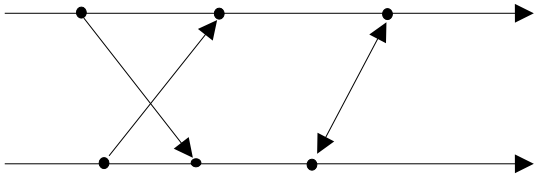


Abbildung 5.12: Nachrichten-Synchronisation (message-passing)

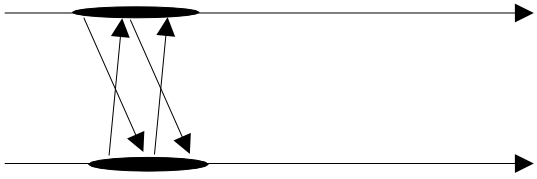


Abbildung 5.13: Rendezvous-Synchronisation als Grenzfall von Nachrichten-Synchronisation

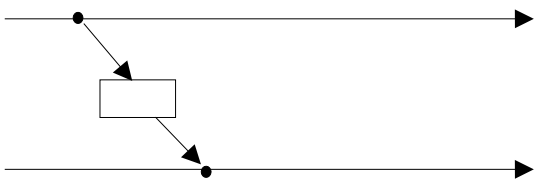


Abbildung 5.14: Nachrichten-Synchronisation als eingeschränkte Speicher-Synchronisation

Speicher-Synchronisation
$\Downarrow$
Nachrichten-Synchronisation
$\Downarrow$
Rendezvous-Synchronisation

Die bisher behandelten Beispiele benutzen alle Speicher-Synchronisation. Der Speicher war dabei etwa ein Feld oder eine Semaphorvariable.

Abbildung 5.16 zeigt das Mengensortierprogramm aus Beispiel 5.4 mit Rendezvous-Synchronisation. Jedem der nebenläufigen Programme P_1 und P_2 ist die Verwaltung einer der beiden Mengen A und B übertragen. Alle Variablen dieser Programme sind lokal. Soll P_1 auf den Wert der Variablen *min* von P_2 zugreifen, dann muß P_1 in einem Rendezvous mit P_1 diesen Wert übergeben.

P_1 signalisiert z.B. durch $P_2?\ mn$, dass es von P_2 einen Wert zu empfangen bereit ist, der auf der Variablen *mn* abgespeichert wird (Abb. 5.15). P_2 signalisiert z.B. durch $P_1!\ min$, dass es bereit ist, den Wert von *min* an P_1 zu senden. Wenn beide Handlungen zusammentreffen (und der Datentyp von *mn* und *min* gleich ist), dann werden beide unteilbar ausgeführt mit der Wirkung:

$$< mn := min >$$

Die Möglichkeit eines Rendezvous ist in der Netzdarstellung von Abb. 5.17 wie in Abb. 5.15 b) dargestellt. Kommt das Rendezvous zustande, dann haben wir die Handlung von Abb. 5.15 c).

Im allgemeinen steht vor dem Start des Programms nicht fest, welche Rendezvous zustandekommen. In [Tay83] wird sogar gezeigt, dass das Problem, dies zu berechnen,

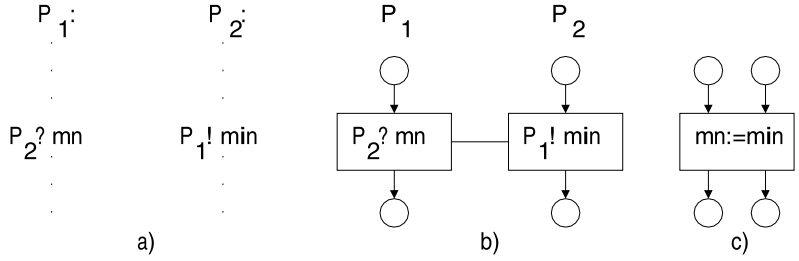


Abbildung 5.15: Rendezvous-Synchronisation

$(*A = A_0 \subset \mathbb{Z}, B = B_0 \subset \mathbb{Z}; A, B$ disjunkt, endlich und nicht leer $*)$

con

P_1:			//	P_2:		
		$max := max(A);$				$min := min(B);$
		$P_2!\ max;$				$P_1?\ mx;$
		$P_2?\ mn;$				$P_1!\ min;$
	do	$max > mn \rightarrow$			do	$mx > min \rightarrow$
		$A := A \backslash \{max\};$				$B := B \backslash \{min\};$
		$A := A \cup \{mn\};$				$B := B \cup \{mx\};$
		$max := max(A);$				$min := min(B);$
		$P_2!\ max;$				$P_1?\ mx;$
		$P_2?\ mn;$				$P_1!\ min;$
	od				od	

noc

$(*A \cup B = A_0 \cup B_0, |A| = |A_0|, |B| = |B_0|, max(A) < min(B)*)$

Abbildung 5.16: Das Programm von Beispiel 5.4 auf Seite 133 in Rendezvous-Synchronisation

NP-vollständig ist. Aus diesem Grunde kann man meist zu einem Programm in PROG nicht ein Netz wie in Abb. 5.17 angeben.

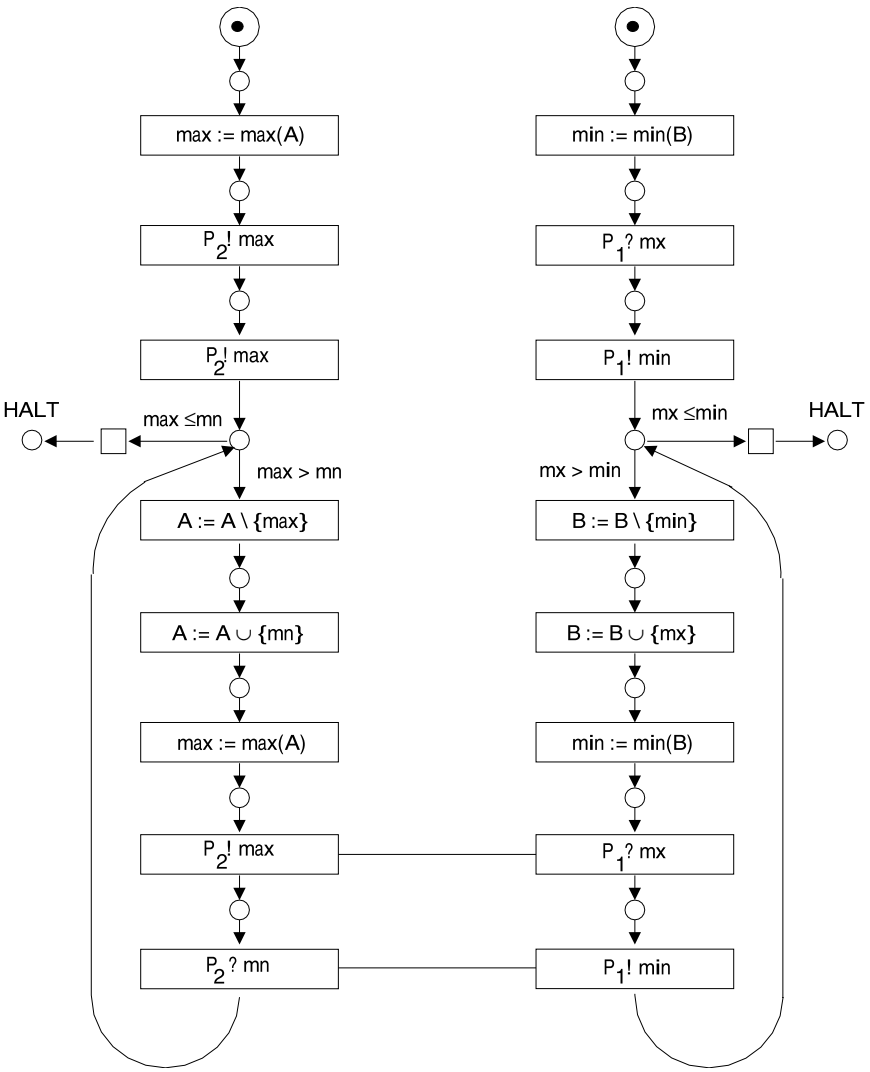


Abbildung 5.17: Das Programm von Abb. 5.16 als Netzprogramm

Rendezvous-Synchronisation wurde in der Form von "Pfad-Programmen" [CH74] und der Sprache COSY [PLS79] bekannt. Hoare hat sie dann in seiner idealen Sprache CSP benutzt [Hoa78]. Praktisch realisiert wird sie in ADA. Speichersynchronisation findet man auch im MONITOR-Konzept, welches in der Sprache Concurrent PASCAL [BH77] realisiert wurde. Wir behandeln diese Sprachen im Abschnitt 5.5.

Um die Unterschiede bzw. Gemeinsamkeiten der beiden Synchronisationsformen zu untersuchen, ist es nützlich, Programme der einen Form in der anderen zu simulieren. Dabei gehen wir von einer virtuellen Funktionseinheit aus, bei der alle Daten direkt von allen Prozessanweisungen zugreifbar sind.

Liegt ein Programm mit verteilter Datenhaltung und Rendezvous-Synchronisation vor, dann ist unmittelbar klar, wie es durch ein Programm mit Speichersynchronisation simuliert werden kann. Zunächst werden die den einzelnen nebenläufigen Prozessen zugeordneten Variablen so umbenannt, dass sie auch bei zentraler Datenhaltung lokal bleiben. Jedes Rendezvous ist dann darstellbar als Zuweisung, wobei die kommunizierenden Prozesse mit den zur Verfügung stehenden Mitteln (z.B. Semaphore) entsprechend synchronisiert werden müssen.

Soll umgekehrt ein Programm mit zentraler Datenhaltung und Speichersynchronisation durch ein Programm mit dezentraler Datenhaltung und Rendezvous-Synchronisation simuliert werden, muß man zusätzliche Prozessanweisungen einführen. Für jede von verschiedenen der ursprünglichen Prozessanweisungen gemeinsam genutzte Datenstruktur d (z. B. eine Variable, ein Feld usw.) wird eine Prozessanweisung P_d eingeführt, d ist natürlich lokal für P_d. Jede andere Prozessanweisung, die auf d zugreifen will, muß also mit P_d ein entsprechendes Rendezvous eingehen und dadurch P_d zu dem gewünschten Zugriff veranlassen.

Behandelt man so z.B. das Beispiel von Abb. 5.8, dann wird für die den Prozessanweisungen “Sender” und “Empfänger” gemeinsame Datenstruktur “buffer” eine neue Prozessanweisung “Puffer” eingeführt. Das Feld “buffer” ist nun lokal in “Puffer”. Der Zugriff wird durch die alternativ ausführbaren Rendezvous mit dem Sender und dem Empfänger realisiert. Die Synchronisation wird dabei ganz durch den Puffer durchgeführt (Abb. 5.18). Dabei darf der Schutz einer geschützten Anweisung $B \rightarrow A$ auch die Form $B = B_1; B_2$ haben, wobei B_1 eine normale boole'sche Bedingung und B_2 eine Kommunikationsanweisung ist.

$B_1; B_2 \rightarrow A$ ist ausführbar, wenn B_1 wahr und B_2 ausführbar ist. Dann wird erst B_2 und dann A ausgeführt.

Aufgabe 5.12 Wäre es richtig die Schleife im Programm *Puffer* von Abb. 5.18 folgendermaßen zu formulieren?

con Sender // Empfänger // Puffer noc

wobei:

Puffer: full := 0 : integer ;
in:= out:= l : integer;
buffer: array[l..k] of message;
do true →
if (l) $full < k$; Sender? $buffer[in]$ → (2) $full, in := full + 1, (in + l)$ mod k
□ (3) $full > 0$; Empfänger! $buffer[out]$ → (4) $full, out := full - 1, (out + 1)$ mod k
fi
od

Sender:
v: message,
do $true$ → (a) produce(v);
(b) Puffer!v
od

Empfänger:
w: message;
do $true$ → (c) Puffer?w;
(d) consume(w)
od

Abbildung 5.18: Sender/Empfänger-Programm mit Rendezvous-Synchronisation

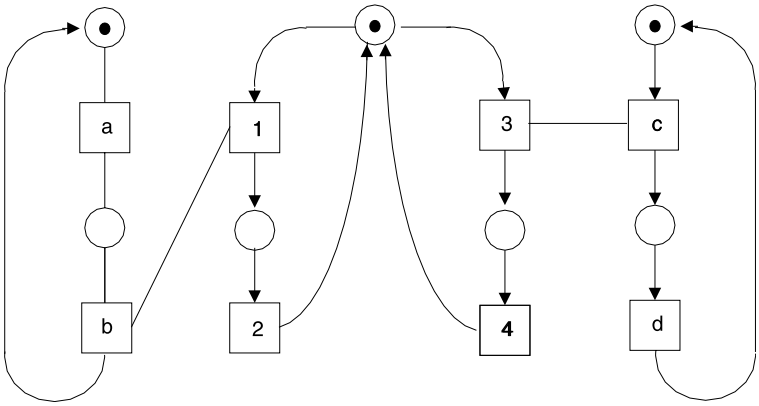


Abbildung 5.19: Synchronisationsstrukturstruktur des Programms von Abb. 5.18

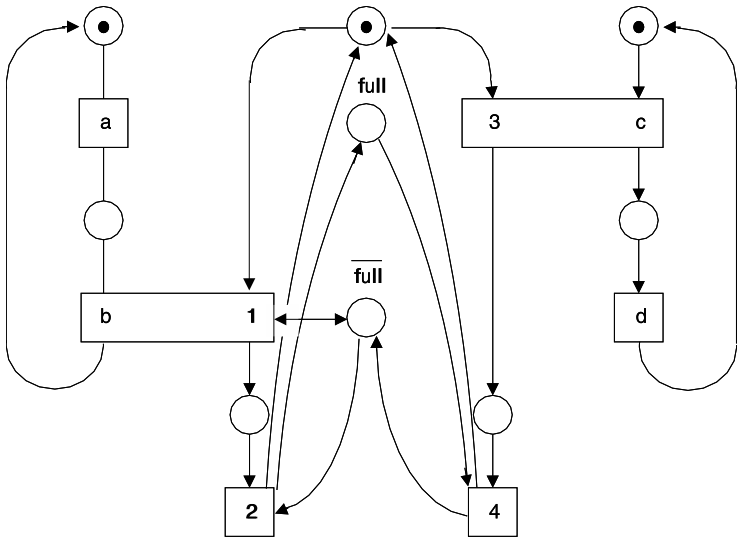


Abbildung 5.20: Kontrollstruktur mit Speicher-Synchronisation

do (l) $full < k$; Sender? $buffer[in] \rightarrow$ (2) $full, in := full + 1, (in + l)$ mod k
□ (3) $full > 0$; Empfänger! $buffer[out] \rightarrow$ (4) $full, out := full - 1, (out + 1)$ mod k
od

Das Programm von Abb. 5.18 wurde möglichst analog zu Abb. 5.8 geschrieben. Abbildung 5.20 zeigt seine Synchronisationsstruktur als Netz. Will man (wie in Abb. 5.9) die Synchronisation des Pufferzugriffs genauer darstellen, so muß man wegen der Abfrage $full < k$, eine zweite (komplementäre) Stelle $full$ mit

$$\mathbf{m}(full) + \mathbf{m}(full) = k$$

für alle erreichbaren Markierungen $\mathbf{m}$ einführen. Der Vergleich mit Abb. 5.9 zeigt, dass $full$ die Rolle von *empty* übernimmt.

Der Übergang von Speicher-Synchronisation zu Rendezvous-Synchronisation kann also auch in Netzen durch Aufspaltung dargestellt werden.

Abbildung 5.21 a) zeigt das Leser/Schreiber-Problem (Abb. 4.1) und Abb. 5.21 b) das Sender/Empfänger-Beispiel (Abb.5.9) in dieser Form. Die durch Linien verbundenen Transitionen dürfen nur gemeinsam schalten (entsprechend Abb. 5.15).

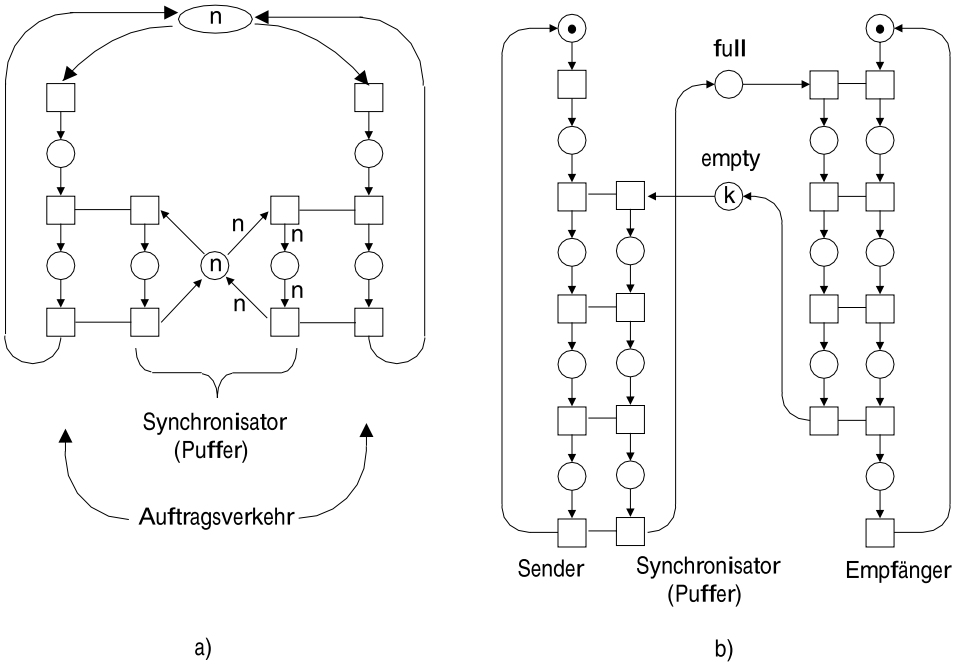


Abbildung 5.21: Rendezvous-Synchronisation bei Netzen

5.5 Höhere programmiersprachliche Synchronisationskonzepte

Wie bei den meisten Anweisungstypen in Programmiersprachen gibt es auch für die Synchronisation nebenläufiger Handlungen elementare und komplexe Operationen. Die ersteren sind maschinennäher und daher einfacher zu implementieren, die letzteren unterstützen die fehlerfreie und strukturierte Programmierung komplexer Aufgaben.

Die vorgestellten Semaphoroperationen gehören natürlich zu den eher niedersprachlichen Formen. Da Semaphore mit Hilfe der P- und V-Anweisungen von beliebigen Programmteilen aus zugreifbar sind, können sehr unübersichtliche Programme entstehen.

Wie mehrfach gezeigt, kann der nebenläufige Zugriff auf gemeinsame Variable zu fehlerhaftem Verhalten führen. Daher versuchen hochsprachliche Synchronisationsanweisungen bei nebenläufig genutzten Variablen

a) einen strukturierten und kontrollierten Zugriff zu unterstützen, sowie

b) die Lokalität zu erhöhen.

Der zweite Punkt hat leider zur Folge, daß Datenmengen oft kopiert werden müssen, wie dies z. B. bei Rendezvous-Synchronisation oder Datenflußprogrammen der Fall ist. Wir behandeln diese Konzepte der Strukturierung im folgenden für Speicher-Synchronisation und Rendezvous-Synchronisation.

Schon bei der Entwicklung sequentieller Programme wurde die Notwendigkeit erkannt, komplexe Datenstrukruren einzuühren, bei denen der Zugriff nur durch wohldefinierte Prozeduren möglich ist. Ausgehend von SIMULA 67 wurde dieses Konzept weiterentwickelt in Sprachen wie MODULA, EUCLID, CLU, ALPHARD oder ADA. Bei nebenläufigen Programmen war es dann natürlich naheliegend, den Zugriff auf solche Datenabstraktionen nur im wechselseitigen Ausschluß zuzulassen. Eine solche Datenabstraktion heißt daher "Monitor" (Brinch-Hansen [BH73]; Hoare [Hoa74]). Das Monitor-Konzept wird in Sprachen für nebenläufige Programmierung wie Concunrrent PASCAL (kurz CP) [BH73] oder Concurrent EUCLID [Hol83] benutzt.

Ein Monitor ist nach folgendem Schema aufgebaut:

- Monitorname
- Zugriffsrechte
- gemeinsame Variablen
- sichtbare Prozeduren
- unsichtbare Prozeduren
- Initialisierung.

Nur die sichtbaren Prozeduren können von nebenläufigen Prozessen aufgerufen werden, und dies per definitionem nur im wechselseitigen Ausschluß. Beim Aufruf wird der Monitorname und der Name der sichtbaren Prozedur benutzt. Durch die Zugriffsrechte wird festgelegt, welche Prozessanweisungen oder Monitore überhaupt einen solchen Aufruf enthalten dürfen. Durch Benutzung der sichtbaren Prozeduren können die gemeinsamen Daten geändert werden. Dabei können die unsichtbaren Prozeduren nützlich sein, welche aber nicht außerhalb des Monitors aufrufbar sind. Im Initialisierungsteil kann den gemeinsamen Variablen bei der Erzeugung des Monitors ein Anfangswert zugewiesen werden.

Soll ein aufrufender Prozess P, verzögert werden, so ist dies mit einer Anweisung *delay(p)* möglich. Dabei wird der aufrufende Prozess in einem Wartepool p der Kapazität l verzögert, bis ein anderer Prozess P_2 die Anweisung *continue(p)* ausführt. Wenn P_2 dann den Monitor verläßt, kann P_1 die nächste Anweisung nach *delay(p)* ausführen. Mit *empty(p)* kann abgefragt werden, ob der Pool frei oder belegt ist. Pools mit größerer Kapazität können durch Felder vom Typ Wartepool dargestellt werden.

Abbildung 5.22 zeigt das Sender/Empfänger-Programm von Abb. 5.18 als CP-Programm. Der Puffer ist zum Monitor geworden. Empfänger und Sender sind zyklische Prozessanweisungen mit Zugriffsrecht auf den Puffer. Die sichtbaren Prozeduren sind durch "entry" gekennzeichnet.

Ein Monitor stellt also - bis auf die wichtige Ausnahmeregel für *delay* und *continue* - eine unteilbare und vergröberte Anweisung im Sinne des Abschnitts 5.3 dar. Der Zugriff auf gemeinsame Variable ist streng kontrolliert. Abbildung 5.24 zeigt die Synchronisationsstruktur des Programmes von Abb. 5.22 als Netz: Monitore sind natürlich als passive Systemteile durch Kanäle (Stellen) dargestellt; der Pfeil zeigt in Zugriffsrichtung.

Als Nachfahre von Simula, Concurrent Pascal und C realisiert die Programmiersprache Java Nebenläufigkeit ebenfalls durch Speicher-Synchronisation. Nebenläufige Einheiten in Java sind Objekte, die von der speziellen Klasse Thread abgeleitet sind. Dies betrifft sogar den automatisch erzeugten Prozess, in dem die Methode main() der Hauptklasse ausgeführt wird. Die einfachste Möglichkeit, um neben dem Prozess für die Methode main() weitere Prozesse zu erzeugen, besteht darin, in eine Klasse, die von Thread erbt, die Methode run() zu überschreiben. run() wird nach dem Start des Threads automatisch aufgerufen und kann dann weitere Methoden anstoßen.

Durch die Methode suspend im Beispiel von Abb. 5.25 wird die Ausführung des Thread zeitweise ausgesetzt, um eine Eingabe zu verarbeiten. Methoden in Objekten können vor gleichzeitiger Ausführung durch das Schlüsselwort `synchronized` geschützt werden, was den wechselseitigen Ausschluß der Methode mit anderen bewirkt. Abb. 5.26 zeigt ein Beispiel.

Manchmal ist der Programmteil, der ein gemeinsam genutztes Objekt verändert, wesentlich kürzer als eine Methode des zu schützenden Objekts. Dazu ist es sinnvoller, folgende Anweisung zu benutzen.

```
synchronized (expression)
    statement
```

```
Puffer = monitor;

    var  full, empty, in, out ;integer;
         buffer : array[i. .k] of message;
         p, q: queue;

    procedure  entry send(v :message)
         begin
         if full=k then delay(p);
         buffer[in]:=v;
         full:= full+1;
         in := (in+l) mod k;
         continue(q)
         end

    procedure  entry receive (var w: message)
         begin
         if full=0 then delay(q);
         w:= buffer[out];
         full:= full-1;
         out := (out+1) mod k;
         continue(p)
         end
    begin full:= 0; empty:= k;
         in := l; out := 1
    end.

Sender = process(Puffer);
    var v: message; cycle produce(v);
         Puffer.send(v) end.
Empfänger = process(Puffer);
    var w: message; cycle Puffer.receive(w);
         consume(w) end.
```

Abbildung 5.22: Sender/Empfänger-Programm mit Monitor

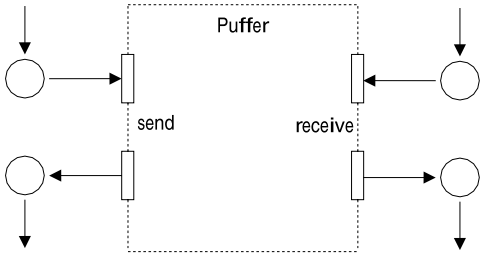


Abbildung 5.23: Monitor als unteilbare Vergröberung

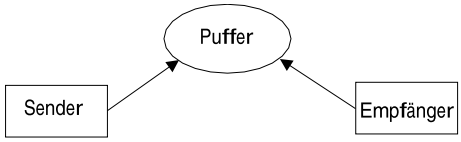


Abbildung 5.24: Netz zu Abb. 5.22

```
    Thread ballgame;

    public void  stopbutton ()
      ballgame.suspend ();
      if (askUser(
         "Wollen Sie wirklich das Spiel beenden?(y/n)"))
         ballgame.stop ();
      else
         ballgame.resume ();
}
```

Abbildung 5.25: Thread in Java

```
class Manage Buf {
private int [100] buf;
public synchonized void deposit (int item) { ...}
public synchonized int fetch () {...}
...
}
```

Abbildung 5.26: Synchronized method in Java

Hier wird `expression` zu einer Objektreferenz ausgewertet. Das Objekt wird während der Ausführung von `statement` geschützt, wobei letzteres eine einfache oder zusammengesetzte Anweisung sein kann. Die Abbildungen 5.27 und 5.28 enthalten als größeres Beispiel Klassen für ein Sender/Empfänger-Programm, das viele Ähnlichkeiten mit dem CP-Programm in Abb. 5.22 aufweist.

Die folgenden Zeilen erzeugen aus den Klassen Queue, Producer und Consumer, die Objekte buff, producer und consummer.

```
Queue buff = new Queue(100);
Producer producer = new Producer(buff);
consumer consumer = new Consumer(buff);
producer.start();
consumer.start();
```

Zunächst wird das Objekt `buff` der Klasse Queue erzeugt. Dabei wird der Parameter 100 an `size` zugewiesen. `que` wird als Feld der Länge `size = 100` erzeugt. Ebenso wird das Objekt `producer` mit dem Parameter `buff`, also mit Referenz auf das gerade erzeugte Objekt `buff` generiert. Die Anweisung `producer.start()`; bewirkt die Ausführung des Kodes von `producer`. Entsprechendes geschieht mit `consumer`.

Wegen extends Thread werden die Objekte producer und consumer nebenläufig ausgeführt. Consumer erzeugt zunächst einen Wert in new_item vom Typ integer. Dies ist nicht ausgeführt und nur durch den Kommentar `//--Create a new_item` dargestellt.

```
class Queue    {
               private int [] que;
               private int nextIn,
               nextOut,
               filled,
               queSize;

    public Queue (int size) {
        que = new int [size];
        filled = 0;
        nextIn = 1;
        nextOut = 1;
        queSize = size;
    } //** end of Queue constructor

        public synchronized void deposit (int item) {
        try {
            while (filled == queSize)
                wait ();
            que [nextIn]= item;
            nextIn = (nextIn  + 1); % queSize
            filled++;
            notify ();
        }  // ** end of try clause
        catch (InterruptedException e) {}
    }     //** end of deposit method



    public synchronized int fetch () {
        int item = 0;
        try {
            while (filled == 0)
                wait ();
            item = que [nextOut];
            nextOut = (nextOut  + 1); % queSize
            filled--;
            notify ();
        }  //** end of try clause
        catch (InterruptedException e) {}
        return item;
    }    //** end of fetch method
} //** end of Queue class
```

Abbildung 5.27: Sender/Empfänger-Programm in Java: Klasse Queue

```
class Producer extends Thread {
    private Queue buffer;
    public Producer(Queue que) {
            buffer = que;
    }
        public void run () {
            int new_item;
            while (true) {
            //-- Create a new_item
        buffer.deposit(new_item);
        }
    }
    }

class Consumer extends Thread {
    private Queue buffer;
        public Consumer (Queue que) {
            buffer = que;
        }
        public void run () {
            int stored_item;
            while (true) {
            stored_item = buffer.fetch();
            //-- Consume the stored_item
            }
        }
    }
```

Abbildung 5.28: Sender/Empfänger-Programm in Java: Klassen Producer, Consumer

Bevor die Schleife wiederholt wird, erfolgt ein Zugriff auf die Methode `deposit` von `buff`, wobei der Wert der Variable `new_item` an `item` übergeben wird. In `wait` wird die Ausführung zunächst nicht angehalten, da `filled == queSize` nicht wahr ist, denn es gilt `filled = 0` und `queSize = 100`. Dann wird der Wert von `item` in die Position `nextIn = 1` des Feldes `que` eingetragen, `nextIn` um 1 modulo `queSize` erhöht und `filled` ebenso um 1 erhöht. `notify()` stößt eine wartende und ausführbare Methode an. Bei späteren Durchläufen kann dies nur bei `wait()` in der Methode `fetch` sein, falls `filled == 0` nicht mehr gilt. Generell dürfen die Anweisungen `wait()` und `notify()` nur in synchronisierten Methoden aufgerufen werden. Der Wirkungsbereich von `notify` ist das umgebende Objekt, nur also `buff`, wobei nur eine wartende Methode angestoßen wird. Soll dies für alle wartenden Methoden gelten, so ist `notifyall` zu benutzen.

`catch (Interrupted Exception e)` ist eine Anweisung zur Ausnahmebehandlung, die hier aber nichts bewirkt. Der Wirkungsbereich der Ausnahmebehandlung wird durch `try{...}` definiert.

Das Verhalten des Objektes `consumer` der Klasse `Consumer` mit Zugriff auf die Methode `fetch()` ist sinngemäß. Zu beachten ist jedoch, daß der Wert des Parameters `item` in `buff` der Variable `stored_item` von `consumer` zugewiesen wird. Die Rückgabevariable wird durch `return item` festgelegt.

Bei der Rendezvous-Synchronisation wird das Problem der gemeinsamen Variablen in radikaler Weise gelöst: gemeinsame Variablen werden gänzlich ausgeschlossen. Kommunikation und Synchronisation werden durch direkten Nachrichtenaustausch vollzogen (vgl. Abschnitt 5.4). Wie in der Einleitung erwähnt, wird dies mit größerem Speicheraufwand und vielen Kopieroperationen erkauft. Im Programm von Abb. 5.16 sind die Variablen *max* bzw. *min* im wesentlichen Kopien der gleichen Variablen in P_1 bzw. P_2. Natürlich kann Rendezvous-Synchronisation auch durch technische Gegebenheiten, wie kommunizierende Funktionseinheiten ohne gemeinsamen Speicher, notwendig werden. Rendezvous-Synchronisation wurde systematisch durch die theoretische Programmiersprache CSP (Communicating Sequential Processcs) von Hoare [Hoa78] eingeführt. Durch OCCAM (May et al 84,85) existiert auch eine Implementation von CSP. Da CSP geschützte Anweisungen benutzt, können wir diese Sprache leicht unter Rückgriff auf die Definition von PROG (Def. 5.1) darstellen. CSP ist wie PROG, wobei alle Variablen für jede Anweisung A_i in einer Nebenlaufanweisung (5.1 g) lokal sind. Dafür werden am Ende von Schutzbedingungen und im Anweisungsteil Rendezvous-Anweisungen wie in Abb. 5.15 zugelassen. Die Abbildungen 5.16 und 5.18 zeigen CSP-Programme. (Man beachte, daß in (Hoare 78) unterschiedliche Bezeichnungen für die Kontrollstrukturen benutzt werden, und zwar $[B_1 \to A_1 \Box \ldots \Box B_n \to A_n]$ für (5.1 e), $*(B_1 \to A_1 o \ldots \Box B_n \to A_n]$ für (5.1 f) und $[P_1 :: A_1 | ... | P_n :: A_n]$ für (5.1 g).) Das Abb. 5.31 entsprechende Netz für das CSP-Programm von Abb. 5.18 ist in Abb. 5.29 dargestellt: der Puffer ist aktive Prozessanweisung geworden; die Kommunikation ist wechselseitig.

Abbildung 5.29: Netz zu Abb. 5.18

Auch die Programmiersprache ADA (Barnes [Bar80]; Horowitz [Hor83]) benutzt Rendezvous-Synchronisation. Die kommunizierenden Anweisungen heißen "tasks". Anders als in CSP braucht der gerufene Task nicht den Namen der rufenden Task zu kennen. Der Grund liegt darin, daß der gerufene Task auch eine dienstleistende Funktionseinheit (etwa eine Programmbibliothek) sein kann. ADA kennt eine der Auswahlanweisung ähnliche Anweisung, die "Select-Anweisung":

select
  when $B_1 \rightarrow A_1 or B_2 \rightarrow A_2 or ... or B_n \rightarrow A_n$
    else A_{n+1}
end select

Wie in der Auswahlanweisung wird eine Anweisung $A_i (1 \leq i \leq n)$ ausgeführt, für die die Bedingung B_i wahr ist. Ist kein B_i wahr, dann wird A_{n+1} ausgeführt. Der else-Teil mit A_{n+1} kann auch fehlen. Dann ist die Anweisung blockiert, bis ein B_i wahr ist. Es sind jedoch eine Reihe von Ausnahmeregelungen zu beachten. Zur Illustration formulieren wir in Abb. 5.30 den Puffer des CSP-Programms von Abb. 5.18 als ADA-Task.

Abbildung 5.31 zeigt eine verfeinerte Form der Rendezvous-Synchronisation. Sie wurde im V-System (Cheriton [Che84]) gewählt, das zur Kommunikation in lokalen Rechnernetzen entwickelt wurde. Den Sender kann man sich z.B. als Benutzerauftrag und den Empfänger als Dienstleistungsauftrag denken. Der letztere ist solange inaktiv oder blockiert, bis er durch eine SEND-Anweisung eines Senders aktiviert wird. Danach bleibt der Sender blockiert, bis der Empfänger den Dienstleistungsauftrag vollzogen hat. Dieser kann z.B. darin bestehen, einen Datenblock von einem Teilnehmer zum anderen zu kopieren. Gegenüber der in CSP oder ADA gewählten Form erlaubt dieser Rendezvous-Mechanismus eine direktere Anwendung von Bedienstrategien. Der Empfänger kann nämlich mehrere Anforderungen abwarten und diese nach einer gewählten Bedienstrategie (z.B. Restrotationszeit bei einer Platte) behandeln. Eine entsprechende ADA-Implementation wäre viel komplizierter.

Rendezvous-Synchronisation ist ein Spezialfall von Gruppen-Kommunikation. Hierbei ist es möglich, anstelle eines einzigen Partners mit einer ganzen Gruppe zu kommunizieren. Gruppen werden durch spezielle Gruppenbezeichner ausgewiesen. Aufträge können durch bestimmte Anweisungen einer Gruppe beitreten ("join Group") oder diese verlassen ("Leave Group") (vgl. Cheriton [Che84])), was bestimmte Rechte des Auftrages

```
task Puffer is
    entry sende (v : in.message);
    entry empfange (w; out.message);
end;
task body Puffer is
    full: integer := 0; empty: integer := k;
    in : integer := 1; out : integer := 1;
    buffer: array[1..k] of message;
beqin
    loop
        select
            when full < k    →
                accept sende(v : inmessage) do
                        buffer[in] := v
                        end;
                        full := full + 1;
                        in := (in + l) mod k
        or
            when full > 0    →
                accept empfange(w : outmessage) do
                        w := buffer[out]
                        end;
                        full := full − 1;
                        out := (out + 1) mod k
        end select;
    end loop;
end Puffer
```

Abbildung 5.30: Puffer als ADA-Task

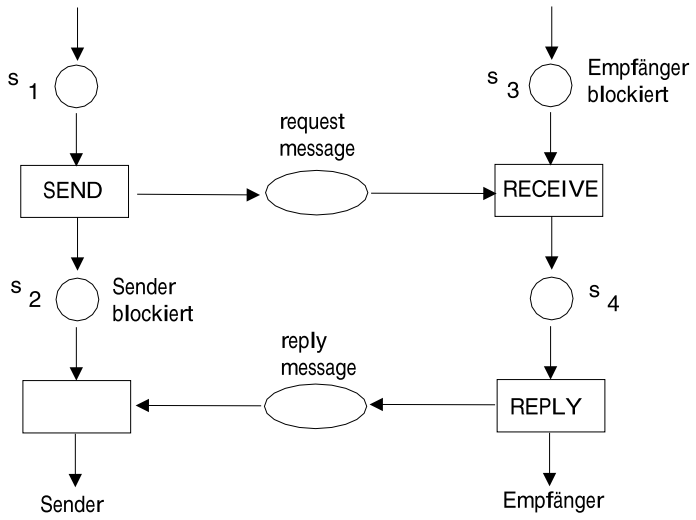


Abbildung 5.31: Rendezvous im V-System

voraussetzen kann. Als Kommunikationsform unterscheidet man eine Mitteilung (notification) von einer Anfrage (query).

Gruppen-Kommunikation ist bei verteilten Systemen sinnvoll einsetzbar. Beispielsweise kann ein Auftrag eine Gruppe von Dienstleistungsaufträgen befragen, ob sie frei sind ("freie Drucker bitte melden!"). Umgekehrt können Aufträge durch Mitteilungen ihren Zustand anderen Aufträgen bekanntgeben ("ich, der Drucker XY, bin frei"). Gruppenkommunikation ist in vielen lokalen Rechnernetzen relativ leicht zu implementieren, da alle Teilnehmer einen Gemeinschaftskanal abhören (z.B. ein Koaxial-Kabel in Ethernet).

Die Programmierung von Speichersynchronisation durch eine zwischengeschaltete Task in ADA ist nicht sehr efficient, da jeweils Rendezvous-Synchronisation eingesetzt werden muß.

Daher wurden in ADA 95 geschützte Objekte (protected objects) eingeführt, die eher mit Monitoren zu vergleichen sind. Der Zugriff auf geschützte Objekte erfolgt durch geschützte Unterprogramme oder Zugriffpunkte die wie in einer Task durch *entry* gekennzeichnet sind. Das Unterprogramm oder der Task-Body wird im wechselseitigen Ausschluß zu anderen Aufrufen ausgeführt.

Der wesentliche Unterschied von Abb. 5.32 im Vergleich zur Abbildung 5.30 ist also

```
protected BUFFER is
entry DEPOSIT (ITEM : in INTEGER);
entry FETCH (ITEM : out INTEGER);
private
    BUFSIZE : constant INTEGRER := 100;
    BUF : array (1..BUFFSITE) of INTEGER;
    FILLED : INTEGER range 0..BUFSIZE := 0;
    NEXT_IN,
    NEXT_OUT : INTEGER range 1.. BUFSIZE := 1;
end BUFFER;

protected body BUFFY is
    accept DEPOSIT (ITEM : in INTEGER)
    when  FILLED < BUFSIZE     is
    begin
    BUF (NEXT_IN) := ITEM;
    NEXT_IN := (NEXT_IN mod BUFSIZE) + 1;
    FILLED := FILLED + 1;
    end DEPOSIT;
    accept FETCH (ITEM : out INTEGER) when FILLED > 0 is
        begin
        ITEM := BUF (NEXT_OUT);
        NEXT_OUT := (NEXT_OUT mod BUFSIZE) + 1;
        FILLED := FILLED - 1;
        end FETCH;
end BUFFFER;
```

```
-- Main command loop for a command interpreter
loop
 select
     Terminal.Wait_For_Interrupt;
     Put_Line ("Interrupted");
 then abort
     -- This will be abandoned upon terminal interrupt
     Put_Line ("-> ");
     Get_Line (Command, Last);
     Process_Command (Command (1..Last));
 end select;
end loop;
-- A time-limited calculation
select
     delay 5.0;
     Put_Line ("calculation does not converge");
then abort
     -- This calculation should finish in 5.0 seconds;
     -- if not, it is assumed to diverge.
     Horribly_complicated_Recursive_Function (X, Y);
end select;
```

Abbildung 5.33: Asynchrone Select-Anweisung in ADA 95

das Fehlen der loop-Anweisung, wodurch das geschützte Objekt als nicht selbstständiger Programmteil erkennbar ist.

Außerdem ist ADA 95 durch eine asynchrone Kommunikationsmöglichkeit erweitert worden, die äsynchrone Select-Anweisung" heißt. Sie kann entweder sofort auf eine Nachricht einer anderen Task reagieren oder mit Hilfe einer delay-Anweisung nach einer spezifizierten Verzögerung.

Im ersten Teil des Beispiels von Abbildung 5.33 werden die Anweisungen im abort-Teil wiederholt ausgeführt, bis der Aufruf von Terminal.wait_For_Interrupt erfolgt.

Im zweiten Teil wird die Funktion im abort-Teil der select-Anweisung bis zu fünf Sekunden lang ausgeführt. Wenn sie bis dahin nicht terminiert, wird die select-Anweisung abgebrochen. (Quelle: [Wir99])

Kapitel 6

Parallele Algorithmen

6.1 Einleitung

Unter parallelen und verteilten Algorithmen wird die Erweiterung von klassischen, für Einprozessormaschinen konzipierten Algorithmen auf viele miteinander kooperierende Prozessoren verstanden. Parallele Algorithmen beruhen in der Regel auf Speicher- oder Rendezvous-Synchronisation (vergleiche Abschnitt **??**) und haben eine synchrone Ablaufsemantik. Charakteristisch für die in Kapitel 7 behandelten verteilten Algorithmen ist dagegen Nachrichten-Synchronisation.

Parallelen Algorithmen mit Rendezvous-Synchronisation liegt als Rechnerarchitektur eine meist reguläre Struktur (n-dimensionales Feld ($1 \leq n \leq 4$), Baum, Ring usw) von Prozessoren zu Grunde (Abb. 6.1). Bei Speichersynchronisation wird eine SIMD-Architektur (single instruction multiple data) (Abb. 6.1) benutzt, deren Formalisierung das PRAM-Modell (parallel random-access machine) ist. Komplexitätsmaße sind die

- *Zeitkomplexität*: maximale Anzahl $T(n)$ der synchronen Schritte aller Prozessoren bis zur Termination bei Probleminstanzen der Größe n,
- *Speicherkomplexität*: maximale Anzahl $S(n)$ der im gemeinsamen Speicher und den lokalen Speichern der Prozessoren bis zur Termination belegten Speicherzellen bei Probleminstanzen der Größe n,

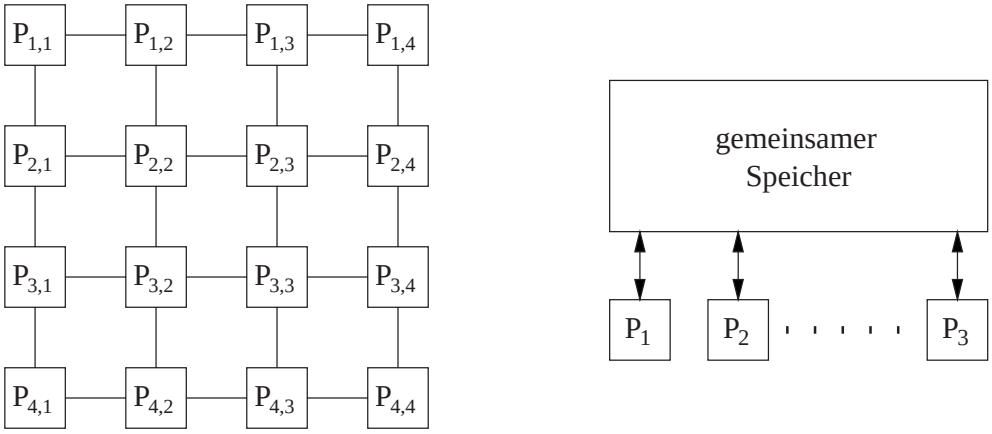


Abbildung 6.1: Prozessorkonfigurationen: 2-dimensionales Feld und PRAM

- *Prozessorkomplexität*: maximale Anzahl $P(n)$ der bis zur Termination aktiv gewordenen Prozessoren bei Probleminstanzen der Größe n,
- *Operationenkomplexität*: maximale Anzahl $W(n)$ der Operationen aller Prozessoren bis zur Termination bei Probleminstanzen der Größe n (d.h. parallele Operationen werden einzeln gezählt, "W" für "work").

Sei $\Theta(T^*(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen (oft wird auch nur das Maximum unter allen *bekannten* Algorithmen für $\mathcal{P}$ genommen, wenn ersteres nicht bekannt ist). Ein paralleler Algorithmus A heißt *optimal* für ein Problem $\mathcal{P}$, wenn $W(n) = \Theta(T^*(n))$ (äquivalente Notation: $W(n) \in \Theta(T^*(n))$. In anderen Worten: die Gesamtzahl der von A ausgeführten Operationen ist asymptotisch gleich der sequentiellen Zeitkomplexität $T^*(n)$ des Problems - unabhängig von der (parallelen) Zeitkomplexität $T(n)$ von A.

Lehrbücher zu parallelen Algorithmen sind [Jáj92], [Rei93], [Cha92] und [GS93]. Die folgenden Algorithmen sind in [Jáj92] ausführlicher beschrieben. Dabei bezeichnet $|M|$ die Mächtigkeit einer Menge M. **pardo** ist eine Anweisung zur (synchron-)parallelen Ausführung.

6.2 Paralleles Suchen und optimales Mischen

Definition 6.1 *Gegeben sei eine lineare Ordnung $(S, \leq)$ (Def. 2.1) und zwei Teilmengen $\tilde{A}, \tilde{B} \subseteq S$ mit $|\tilde{A}| = |\tilde{B}| = n > 0$. Darstellung: $A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_n)$, wobei $i < j \Rightarrow a_i \leq a_j \quad \wedge \quad b_i \leq b_j \quad (1 \leq j, j \leq n)$. Die* Mischung *(merge) von A und B ist die Folge $C = (c_1, c_2, \ldots, c_{2n})$ mit $i < j \rightarrow c_i \leq c_j$, die A und B als disjunkte Teilfolgen enthält.*

Beispiel: $(2, 4, 4) \quad (3, 4, 7)$

$\searrow \quad \swarrow$

$(2, 3, 4, 4, 4, 7)$

Definition 6.2 *Sei $X = (x_1, x_2, \ldots, x_t)$ eine Folge mit Elementen aus der linear geordneten Menge S. Für $x \in X$ ist der* Rang *von x in X definiert als:*
$rank(x : X) := |\{i | i \in \{1, \ldots, t\} \wedge x_i \leq x\}|$. Für eine weitere solche Folge $Y = (y_1, \ldots, y_s)$ sei $rank(Y : X) := (r_1, r_2, \ldots, r_s)$ mit $r_i = rank(y_i : X)$

Beispiel: Für $X = (25, -13, 26, 31, 54, 7)$ und $Y = (13, 27, -27)$ ist $rank(Y : X) = (2, 4, 0)$.

Zur Vereinfachung nehmen wir jetzt $A \cap B = \emptyset$ an. Das Problem, die Folgen A und B zu mischen, wird dann zu:

Bestimme für jedes $x \in A \cup B : rank(x : A \cup B) = i$
(dann ist x das i-te Element c_i in C)

Wegen $rank(x : A \cup B) = rank(x : A) + rank(x : B)$ genügt uns ein Algorithmus für:

Berechne $rank(B : A)$

Für $b_i \in B$ kann durch binäres Suchen j_i mit $a_{j_i} < b_i < a_{j_i+1}$ in $O(\log n)$ (sequentieller) Zeit berechnet werden, d.h. $j_i = rank(b_i : A)$. Dieses Verfahren kann man parallel auf alle Elemente von B anwenden, d.h. wir erhalten einen parallelen Algorithmus mit $O(\log n)$ (paralleler) Zeit und $O(n \log n)$ Operationen.

Algorithmus 6.1 (Partitionieren)

Eingabe: Zwei Felder $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_m)$ in aufsteigender Reihenfolge sortiert, wobei $\log m$ und $k(m) = \frac{m}{\log m}$ ganze Zahlen seinen müssen.

Ausgabe: $k(m)$ Paare (A_i, B_i) von Teilfolgen von A und B, sodass
(1) $|B_i| = \log m$
(2) $\sum_i |A_i| = n$ und
(3) jedes Element von A_i und B_i ist größer als jedes Element von A_{i-1} oder B_{i-1} für alle $1 \leq i \leq k(m) - 1$.

begin

1. Set $j(0) := 0, j(k(m)) := n$
2. **for** $1 \leq i \leq k(m) - 1$ **pardo**
   2.1 Berechne $rank(b_{i \log m} : A)$ durch binäre Suche
   und setze $j(i) = rank(b_{i \log m} : A)$
3. **for** $0 \leq i \leq k(m) - 1$ **pardo**
   3.1 $B_i := (b_{i \log m + 1}, \ldots, b_{(i+1) \log m}$;
   3.2 $A_i := (a_{j(i)+1}, \ldots, a_{j(i+1)})$
   (A_i ist leer, wenn $j(i) = j(i+1)$ gilt.)

end

Da es sequentielle Algorithmen mit linearer Zeitkomplexität gibt (d.h. $O(n)$), ist der parallele Algorithmus *nicht optimal*. Es soll daher jetzt ein optimaler, paralleler Algorithmus entwickelt werden. Als Hilfsalgorithmus bildet der folgende parallele Algorithmus Blöcke A_i und B_i von A und B:

A:	A_1	A_2	...	A_{k_m}
B:	B_1	B_2	...	B_{k_m}

wobei $A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_m)$ und $k_m = \frac{m}{\log m} \in \mathbb{N}$. Außerdem sollen alle $x \in A_i \cup B_i$ größer als die Elemente in $A_{i-1} \cup B_{i-1}$ sein. Im Algorithmus 6.1 wird k_m als $k(m)$, a_{j_i+1} als $a_{j(i)+1}$ usw. geschrieben.

Beispiel: (zum Algorithmus 6.1) Gegeben seien die folgenden zwei Felder A und B mit $m = 4$ und $k(m) = \frac{m}{\log m} = 2$

	a_1	...	a_8		b_1	...	a_4
A=(	4,	6,7,10,12,15,18,	20)	B=(	3,	9,16,	21)

Dann erhält man folgende Teilmengen: $A_0 = (4, 6, 7)$, $A_1 = (10, 12, 15, 18, 20)$, $B_0 = (3, 9)$ und $B_1 = (16, 21)$.

Das folgende Lemma gibt die Zeitkomplexität und Anzahl der Operationen an.

Lemma 6.3 *Sei C die sortierte Folge, welche man durch Mischen der sortierten Folgen A und B mit den Längen n bzw. m erhält. Dann teilt der Algorithmus 6.1 A und B in Paare von Teilfolgen (A_i, B_i) auf, so dass $|B_i| = O(\log m), \sum_i |A_i| = n$ und $C = (C_0, C_1, \ldots)$, wobei C_i die sortierte Folge ist, welche aus A_i und B_i durch Mischen entsteht. Dieser Algoritmus benötigt in $O(\log n)$ Zeit mit insgesamt $O(n + m)$ Operationen.*

Beweis:
Wir zeigen zuerst, dass jedes Element in den Teilfolgen A_i und B_i größer ist, als jedes Element von A_{i-1} oder B_{i-1}. Die zwei kleinsten Elemente von A_i und B_i sind $a_{j(i)+1}$ und $b_{i \log m+1}$, während die größten Elemente von A_{i-1} und B_{i-1}, $a_{j(i)}$ und $b_{i \log m}$ sind. Da $rank(b_{i \log m} : A) = j(i)$, erhalten wir $a_{j(i)} < b_{i \log m} < a_{j(i)+1}$. Dies impliziert, dass $b_{i \log m+1} > b_{i \log m} > a_{j(i)}$ und $a_{j(i)+1} > b_{i \log m}$ ist. Daher ist jedes Element von A_i und B_i größer als jedes Element von A_{i-1} oder B_{i-1}. Die Korrektheit des Algorithmus folgt hieraus unmittelbar.
Zeitanalyse: Der 1.Schritt braucht $O(1)$ sequentielle Zeit. Schritt 2 benötigt $O(\log n)$ Zeit, da die binäre Suche auf alle Elemente parallel angewendet wird. Die Gesamtzahl der für diesen Schritt auszuführenden Operationen ist $O\left((\log n) \times (\frac{m}{\log m})\right) = O(m + n)$, da $\left(\frac{m \log n}{\log m}\right) < \left(\frac{m \log(n+m)}{\log m}\right) \leq n+m$ für $n, m \geq 4$. Der 3.Schritt benötigt $O(1)$ paralleler Zeit, bei Benutzung einer linearen Anzahl von Operationen. Daher läuft dieser Algorithmus in $O(\log n)$ Zeit mit insgesamt $O(n + m)$ Operationen. □

Mit diesem Algorithmus haben wir das Mischproblem der Größe n auf Unterprobleme kleinerer Größe reduziert.

Optionaler Algorithmus für das Mischen:

- wende Algorithmus 6.1 an
- behandle die Paare (A_i, B_i) getrennt (es gilt $\|B_i\| = \log n$)

- falls $|A_i| \leq c \log n$ mische (A_i, B_i) in $O(\log n)$ Zeit mit einem linearen sequentiellen Mischalgorithmus
- falls $|A_i| \not\leq c \log n$, dann wende Algorithmus 6.1 an, um A_i in Blöcke der Länge $\log n$ zu zerlegen. Dazu werden $O(\log \log n)$ Zeit und $O(|A_i|)$ Operationen benötigt.

Insgesamt ergibt sich:

Satz 6.4 *Das Mischen zweier Folgen A und B der Länge n ist von einem parallelen Algorithmus in $O(\log n)$ Zeit mit $O(n)$ Operationen durchführbar.*

Methode des Algorithmus: aufteilen (partioning)
zu unterscheiden von: teile und herrsche (divide-and-conquer)

Um zu einem schnelleren Algorithmus zu kommen, betrachten wir paralleles Suchen.

- *Gegeben:* Eine linear geordnete Folge $X = (x_1, \ldots, x_n)$ von verschiedenen Elementen einer linear geordneten Menge $(S, \leq)$
- $y \in S$
- *Suchproblem:* Finde Index $i \in \{0, 1, \ldots, n\}$ mit $x_i \leq y < x_{i+1}$

Dabei seien $x_0 = -\infty, x_{n+1} = +\infty$ neue Elemente mit $-\infty < x < +\infty$ für alle $x \in S$.

binäres Suchen: Zeitkomplexität $O(\log n)$
paralleles Suchen mit $p \leq n$ Prozessoren: Prozessor P_1: zuständig für Initialisierung und Randsingularitäten. Aufteilen von X in $p + 1$ Blöcke von etwa gleicher Länge. Jede parallele Runde findet $x_i = y$ oder einen y enthaltenden Block.

Beispiel: (zum Algorithmus 6.2) Für die Eingabe

$$X = (2, 4, 6, \ldots, 30), y = 19, p = 2$$

hat P_1 nach Schritt 1 berechnet:

$$l = 0, r = 16, c_0 = 0, c_3 = 1, x_0 = -\infty, x_{16} = +\infty.$$

Die while-Schleife durchläuft 3 Iterationen.

Algorithmus 6.2 (Paralleles Suchen für Prozessor P_j)

Eingabe: (1) Ein Feld $X = (x_1, x_2, \ldots, x_n)$ mit $x_1 < x_2 < \ldots < x_n$
(2) ein Element y
(3) die Anzahl p der Prozessoren mit $p \leq n$
(4) die Prozessornummer j mit $1 \leq j \leq p$

Ausgabe: ein Index i mit $x_i \leq y < x_{i+1}$

begin

1. **if**(j=1) **then do**
   1.1 Set $l := 0; r := n + 1; x_0 := -\infty; x_{n+1} := +\infty$
   1.2.Set $c_0 := 0; c_{p+1} := 1$
2. **while**$(r - l > p)$ **do**
   2.1.**if**(j=1)**then**{set $q_0 := 1; q_{p+1} := r$}
   2.2. Set $q_j := l + j\lfloor\frac{r-l}{p+1}\rfloor$
   2.3. **if**$(y = x_{q_i})$**then {return**(q_j)**; exit}**
   else {set $c_j := 0$ if $y > x_{q_j}$ and $c_j := 1$
   if $y < x_{q_j}$}
   2.4. **if**$(c_j < c_{j+1})$**then**{set $l := q_j; r := q_{j+1}$}
   2.5. **if**$(j = 1$ and $c_0 < c_1)$**then**{set $l := q_0; r := q_1$}
3. **if**$(j \leq r - l)$**then do**
   3.1. Case statement:
   $y = x_{l+j}$:{**return**(l+j);**exit**}
   $y > x_{l+j}$:set $c_j := 0$
   $y < x_{l+j}$:set $c_j := 1$
   3.2. **if**$(c_{j-1} < c_j)$**then return**(l+j-1)

end

Iteration:	1	2	3
q_0	0	5	7
q_1	5	6	8
q_2	10	7	9
q_3	16	10	10
c_0	0	0	0
c_1	0	0	0
c_2	1	0	0
c_3	1	1	1
l	5	7	9
R	10	10	10

Ergebnis durch P_1: **return**(q_1) mit $q_1 = 9$

Satz 6.5 *Zu der Folge $X = (x_1, x_2, \ldots, x_n)$ mit $x_1 < x_2 < \ldots < x_n$ und $y \in S$ berechnet der Algorithmus 6.2 einen Index i mit $x_i \leq y < x_{i+1}$ in der Zeitkomplexität $O\left(\frac{\log(n+1)}{\log(p+1)}\right)$, wobei $p \leq n$ die Anzahl der Prozessoren ist.*

Beweis:
Zur Komplexität: In der i-te Iteration der while-Schleife wird die Länge der zu durchsuchenden Unterfolge von $s_i = r - l$ auf $s_{i+1} \leq \frac{r-1}{p+1} + p = \frac{s_i}{p+1} + p$ gesetzt, was die Länge des (p+1)-ten Blockes beschränkt. Mit $s_0 = n + 1$ löst man die rekurrente Ungleichung zu $s_i \leq \frac{n+1}{(p+1)^i} + p + 1$. Also werden $O\left(\frac{\log(n+1)}{\log(p+1)}\right)$ Iterationen der Länge $O(1)$benötigt. Schritt 3 benötigt $O(1)$ Zeit. □

Anmerkung: Der Algorithmus ist optimal, wenn p konstant ist.

Als nächstes wird ein Algorithmus zum parallelen Mischen entwickelt, der in $O(\log n \log n)$ Zeit arbeitet. Das ist erstaunlich, da schon die Maximumsbestimmung $\Omega(\log n)$ parallele Schritte erfordert (bezogen auf eine CREW PRAM, d.h. eine PRAM die parallel lesen (concurrent read), aber nur exklusiv schreiben (exclusive write) darf.)

Lemma 6.6 *Sei Y eine Folge von m Elementen einer linear geordneten Menge $(S, \leq)$ und X eine Folge von n verschiedenen Elementen mit $m \in O(n^s)$ für eine Konstante $0 < s < 1$. Dann kann $rank(Y : X)$ in $O(1)$ Zeit mit $O(n)$ Operationen berechnet werden.*

Beweis:

- Bestimme $rank(y : X)$ für jedes $y \in X$ mit Algorithmus 6.2 mit $p = \lfloor \frac{n}{m} \rfloor \in \Omega(n^{1-s})$ (vergl. F3, Def.3.8)
- Also kann $rank(y : X)$ für jedes $y \in Y$ in der Zeit $O\left(\frac{\log(n+1)}{\log(p+1)}\right) = O\left(\frac{\log(n+1)}{\log(n^{1-s})}\right) = O(1)$ berechnet werden.
- Die Anzahl der Operationen für ein Element ist $O(p) = O(\frac{n}{m})$, da $p = \lfloor \frac{n}{m} \rfloor$ und die Zeitkomplexität für jeden Prozessor $O(1)$ ist. Bei m Elementen erhält man also $O(n)$ Operationen.

□

Nun soll $rank(B : A)$ für sortierte Folgen A mit $|A| = n$ und B mit $|B| = m$ berechnet werden. A und B sollen keine gemeinsamen Elemente enthalten.

Wieder: Berechnen durch Aufteilen von B in Blöcke der Länge von etwa $\sqrt{m}$:

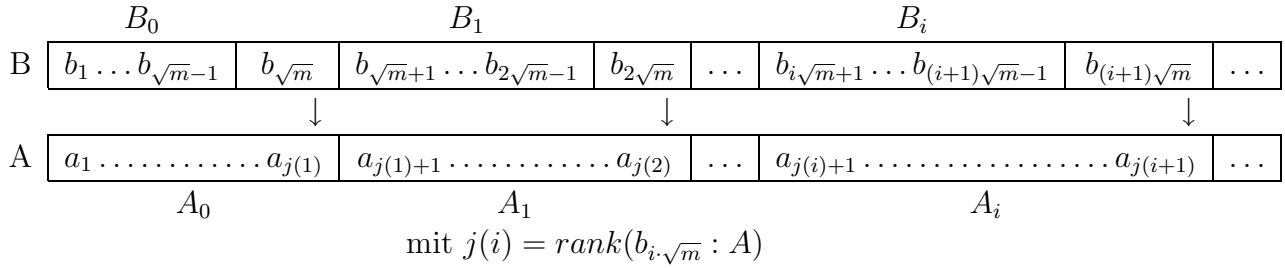
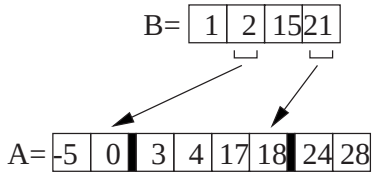


Abbildung 6.2: Aufteilung von B in Blöcke

Daraus: Algorithmus 6.3

Beispiel: (zum Algorithmus 6.3) $(m = 4, \sqrt{m} = 2)$

- *1. Schritt*



Algorithmus 6.3 (Ordne eine sortierte Folge in eine andere sortierte Folge ein)

Eingabe: Zwei Felder $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_m)$ in aufsteigender Reihenfolge.
Ausgabe: Das Feld $rank(B : A)$.
begin

1. Wenn $m < 4$ dann ordne die Elemente von B durch Anwendung des Algorithmus 6.2 mit $p = n$. Fertig.
2. Ordne die Elemente $b_{\lfloor\sqrt{m}\rfloor}, b_{2\lfloor\sqrt{m}\rfloor}, \ldots, b_{i\lfloor\sqrt{m}\rfloor}, \ldots, b_m$ in A mit Hilfe des Algorithmus 6.2 ein. Dabei sei $p = \sqrt{n}$ und $rank(b_{i\lfloor\sqrt{m}\rfloor} : A) = j(i)$, für $1 \leq i \leq \sqrt{m}$ und $j(0) = 0$
3. Für $0 \leq i \leq \sqrt{m} - 1$ sei $B_i = (b_{i\lfloor\sqrt{m}\rfloor+1}, \ldots, b_{(i+1)\lfloor\sqrt{m}\rfloor-1})$ und $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$; Wenn $j(i) = j(i+1)$ dann setze $rank(B_i : A_i) = (0, \ldots, 0)$, ansonsten berechne $rank(B_i : A_i)$ rekursiv.
4. Sei $1 \leq k \leq m$ ein willkürlicher Index, der kein Vielfaches von $\lfloor\sqrt{m}\rfloor$ ist und sei $i = \lfloor\frac{k}{\lfloor\sqrt{m}\rfloor}\rfloor$. Dann ist $rank(b_k : A) = j(i) + rank(b_k : A_i)$

end

- *2.Schritt* $j(0) = 0 \quad j(1) = 2 \quad j(2) = 6$
- *3.Schritt* $B_0 = (1) \quad A_0 = (-5, 0)$
  $B_1 = (15) \quad A_1 = (3, 4, 17, 18)$
- *4.Schritt* $rank(b_1, A) = rank(1, A) = j(0) + rank(1 : A_0) = 2$
  $rank(b_3, A) = rank(15, A) = j(1) + rank(15 : A_1) = 2 + 2 = 4$
  Also: $rank(B : A) = (\underline{2}, 2, \underline{4}, 6)$

Lemma 6.7 *Der Algorithmus 6.3 berechnet* $rank(B : A)$ *in der Zeit* $O(\log\log n)$ *mit* $O((n+m) \cdot \log\log m)$ *Operationen.*

Beweis:
Die Korrektheit wird durch Induktion über m bewiesen.

Der Induktionsanfang $m = 3$ bedeutet die Folge (b_1, b_2, b_3) in A einzuordnen. Dies erfolgt in Zeile 1. Wir nehmen jetzt an, dass die Induktionsbehauptung für alle $m' < m$ mi $m \geq 4$

gilt und beweisen, dass alle Elemente in B_i zwischen $a_{j(i)}$ und $a_{j(i+1)+1}$ liegen (für jedes i mit $0 \leq i \leq \sqrt{m} - 1$).

Jedes Element p in B_i erfüllt $b_{i\sqrt{m}} < p < b_{(i+1)\sqrt{m}}$. Da $j(i) = rank(b_{i\sqrt{m}} : A)$ und $j(i+1) = rank(b_{(i+1)\sqrt{m}} : A)$ gilt $a_{j(i)} < b_{i\sqrt{m}}$ and $b_{(i+1)\sqrt{m}} < a_{j(i+1)+1}$, und somit auch $a_{j(i)} < p < a_{j(i+1)+1}$. Diese Tatsache zeigt, das jedes Element p des Blocks B_i in den Block A_i eingefügt wird. Damit gilt $rank(p : A) = j(i) + rank(p : A)$, weil $j(i)$ die Anzahl der Elemente in A ist, die vor A_i liegen. Damit folgt die Korrektheit durch Induktion.

Nun zu den Komplexitätsschranken. Sei $T(n, m)$ die parallele Zeit, die benötigt wird, um B in A einzufügen, wobei $|B| = m$ und $|A| = n$ sei.

Schritt 2 bewirkt $\sqrt{m}$ Aufrufe des Algorithmus 6.2, wobei $p = \sqrt{n}$ gilt. Dessen Zeitkomplexität ist $O(\frac{\log(n+1)}{\log(\sqrt{n}+1)}) = O(1)$ und die Anzahl der Operationen wird durch $O(\sqrt{m}.\sqrt{n}) = O(n + m)$ beschränkt, da $2\sqrt{m}.\sqrt{n} \leq n + m$. Außerhalb der rekursiven Aufrufe benötigen Schritt 3 und 4 $O(1)$ Zeit mit $O(n + m)$ Operationen.

Sei $|A_i| = n_i$ für $0 \leq i \leq \sqrt{m} - 1$. Der zum Paar (B_i, A_i) gehörende rekursive Aufruf benötigt $T(n_i, \sqrt{m})$ Schritte. Also gilt $T(n, m) \leq max_i T(n_i, \sqrt{m}) + O(1)$ und $T(n, 3) = O(1)$. Eine Lösung dieser Rekurrenzungleichung ergibt $T(n, m) = O(\log\log m)$. Da die Anzahl der Schritte jedes rekursiven Aufrufs $O(n + m)$ ist, ergibt sich die Gesamtzahl der Operationen des Algorithmus 6.3 mit $O((n + m)\log\log m)$. □

Korollar 6.8 *Zwei sortierte Folgen der Länge n können in $O(\log\log n)$ Zeit mit $O(n \cdot \log\log n)$ Operationen gemischt werden.*

Anmerkung: Dieser Algorithmus ist nicht optimal.

6.3 Paralleles Sortieren

Schnelles Sortieren ist in Anwendungen von großer Bedeutung. Daher soll hier ein paralleler, auf Mischen beruhender Algorithmus vorgestellt werden. Im vorigen Abschnitt wurde ein paralleler Algorithmus zum Mischen zweier Folgen behandelt, der aber nicht optimal ist. Er kann aber dazu benutzt werden, um einen optimalen Algorithmus zu konstruieren, indem die Folgen A und B in Blöcke der Länge $\lceil log(log(n)) \rceil$ zerlegt werden. Das ergibt folgendes Ergebnis (siehe [Jáj92], Abschnitt 4.2.3):

Satz 6.9 *Die Aufgabe, zwei sortierte Folgen der Länge n zu mischen, kann mit einem parallelen Algorithmus in der Zeit $O(log\ log\ n)$ mit einer Gesamtzahl von $O(n)$ Operationen erledigt werden.*

Der folgende parallele Algorithmus arbeitet wie merge-sort. Die zu sortierende Folge X wird in Teilfolgen X_1 und X_2 ungefähr gleicher Länge zerlegt, die getrennt sortiert und das Ergebnis durch Mischen zusammengefügt wird. Dies kann implementiert werden, indem ein (balancierter) Binärbaum nach Abbildung 6.3 von den Blättern her erzeugt wird. Die Wurzel enthält dann die sortierte Folge. Dazu wird für jeden Knoten v die entsprechende sortierte Teilliste mit $L[v]$ bezeichnet. Der j-te Knoten der Höhe h sei $v = (h, j)$.

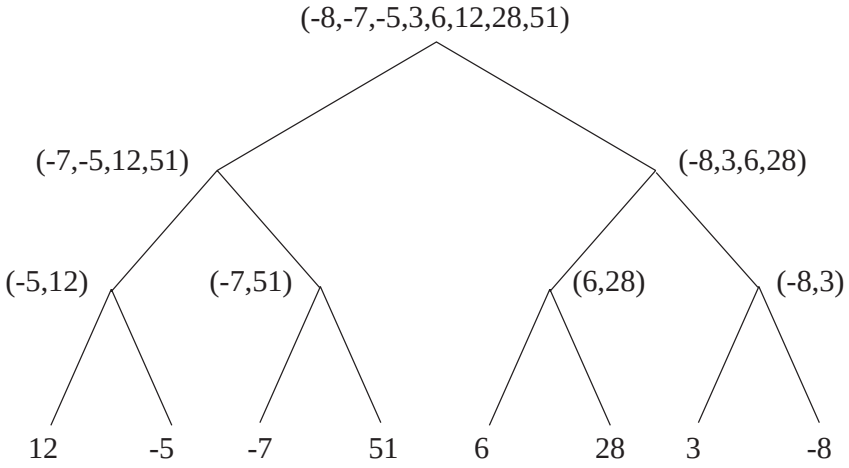


Abbildung 6.3: Binärbaum

Für obiges Beispiel wird das Ergebnis an der Wurzel $L(1, 3)$ erreicht.

Satz 6.10 *Der Algorithmus 6.4 ist mit einer Zeitkomplexität von $O(\log n \cdot \log\log n)$ und $O(n \log n)$ Operationen optimal.*

Beweis:
Die Anzahl der n Iterationen im Schritt 2 ist $O(\log n)$. Jede Iteration benötigt $O(\log\log n)$ Schritte, wenn der oben genannte optimale Mischalgorithmus benutzt wird,

Algorithmus 6.4 (Simple Merge Sort)

Eingabe: Ein Feld X der Ordnung n, wobei $n = 2^l$ für eine ganze Zahl l ist.
Ausgabe: Ein balancierter binärer Baum mit n Blättern derart, dass $L(h, j)$ für jedes $0 \leq h \leq \log n$ die sortierte Teilfolge der Elemente im Teilbaum mit Wurzel (h, j) enthält (mit $1 \leq j \leq \frac{n}{2^h}$). Damit enthält der Knoten (h, j) die sortierte Liste der Elemente $X(2^h(j-1)+1), X(2^h(j-1)+2), \ldots, X(2^h j)$.

begin

1. **for** $1 \leq j \leq n$ **pardo**
   $L(0, j) := X(j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \leq j \leq \frac{n}{2^h}$ **pardo**
   Mische $L(h-1, 2j-1)$ und $L(h-1, 2j)$ zur sortierten Liste $L(h, j)$

end

also insgesamt $O(\log n \cdot \log\log n)$ Schritte. Entsprechend berechnen sich die $O(n \log n)$ Operationen. □

Anmerkung: Es existiert ein solcher optimaler Algorithmus mit Zeitkomplexität $O(\log n)$. Dieses Ergebnis kann nicht verbessert werden, wenn $p \leq n$ Prozessoren benutzt werden. Gilt $2n \leq p \leq n^2$, dann kann $O\left(\frac{\log n}{\log\log \frac{2p}{n}}\right)$ erreicht werden.

Kapitel 7

Verteilte Algorithmen

7.1 Einleitung

Verteilte Algorithmen werden zur Steuerung von Berechnungen und Informationsverarbeitung auf vielen miteinander verbundenen Rechnern eingesetzt. Im Gegensatz zu parallelen Algorithmen sind diese Rechner jedoch in Netzwerke eingebunden, deren Verbindungen durch relativ lange Nachrichtenlaufzeiten gekennzeichnet sind. Zudem können diese Zeiten in *asynchronen* Netzwerken für jede Nachricht unterschiedlich sein - auch für verschiedene Nachrichten auf demselben Kanal. Auch wenn, wie meist angenommen, die Algorithmen auf den einzelnen Knotenrechnern determinstisch ablaufen, wird durch die unbestimmten Nachrichtenlaufzeiten ein nichtdeterministisches Gesamtverhalten erzeugt. Ein solcher Algorithmus wird als terminiert definiert, wenn

1.) alle lokalen Algorithmen in den Knoten terminiert haben und

2.) keine Nachrichten mehr unterwegs sind.

Dies zu überprüfen ist ein nichttriviales Problem (*verteilte Termination*). Die zweite Forderung entfällt bei *synchronen* verteilten Algorithmen. Ihre Abläufe werden in Runden gegliedert. In einer Runde kommen alle abgeschicketen Nachrichten im jeweiligen Zielknoten an und werden dort verarbeitet. Erst dann werden in allen Knoten neue Nachrichten versandt, womit eine neue Runde beginnt. Solche Algorithmen setzen eine

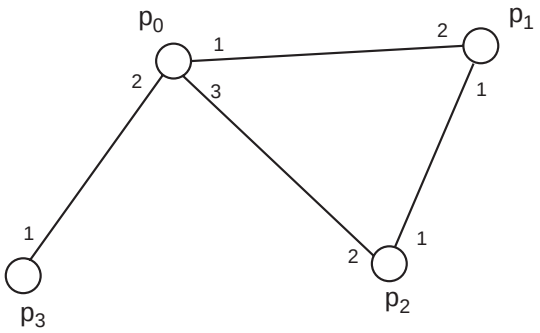


Abbildung 7.1: Netzwerk eines verteilten Algorithmus

für alle Knoten synchrone Taktung voraus, die im Allgemeinen in Rechnernetzen nicht vorausgesetzt werden kann. Wir betrachten hier vorwiegend asynchrone verteilten Algorithmen. Einen Überblick findet man in den Monographien [Mat89] [Rei98] [AW98] und [Lyn96]. Zu dem Forschungsgebiet gehören auch Verfahren der Zuverlsässigkeitsverbesserung bei gestörten Kanälen. Diese werden hier aber nicht behandelt. Es wird vorausgesetzt, dass jede abgesandte Nachricht nach endlicher Zeit beim Empfänger eintrifft. Besondere Beachtung findet ein weiteres Komplexitätsmaß, die

- *Nachrichtenkomplexität* (message complexity): die maximale Anzahl $M(n)$ der von allen Prozessoren bis zur Termination bei Probleminstanzen der Größe n versandten Nachrichten.

Definition 7.1 *Ein* Netzwerk *(eines verteilten Algorithmus) ist ein ungerichter, zusammenhängender Graph. Die an einen Knoten angrenzenden Kantenenden haben (für diesen Knoten) eindeutige Bezeichner (siehe Abb. 7.1).*

Definition 7.2 *Ein verteilter Algorithmus besteht aus einem Netzwerk (Def. 7.1), dessen $n > 0$ Knoten $p_0, p_1, \ldots p_{n-1}$ Prozessoren heißen. Jedem Prozessor ist ein deterministischer Algorithmus zugeordnet, der in einem Schritt an den Kanten vorliegende Nachrichten verarbeiten und über diese Kanten neue Nachrichten versenden kann. Der Algorithmus hat einen Anfangszustand und eine Menge von Endzuständen.*

Die Algorithmen der Prozessoren benutzen nur die Bezeichner der Kanten (Kanäle). In den Algorithmen werden jedoch oft zur einfacheren Darstellung stattdessen die Namen der Zielknoten benutzt.

7.2 Echo- und Wahlalgorithmen

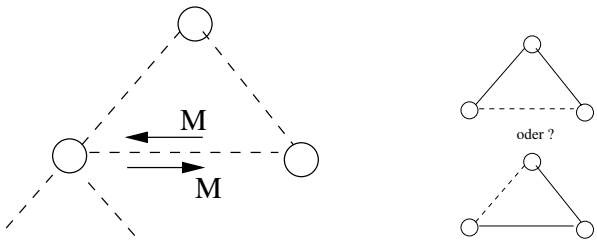
Der Echo-Algorithmus (flooding algorithm)
(Verbreiten von Nachrichten über das ganze Netzwerk, Konstruktion eines Gerüstes (spanning tree))

Verfahren:
- ein festgelegter (Wurzel-)Knoten sendet Nachrichten (Explorer) an alle Nachbarn
- ebenso die anderen Knoten nach Erhalt eines Explorers

Problem: Mehrfachbesuch eines Knotens → führt zu Zyklus statt Gerüst

Lösung: nur die zuerst ankommende Nachricht berücksichtigen.

Problem:
Nachrichten kommen
beide zuerst:



Lösung: Quittung (`<reject>`, echo)

Problem: Termination Lösung: Quittung (Echo) an Wurzel

Lemma 7.3 *Unter den gegebenen Voraussetzungen berechnet der Algorithmus 7.1 ein Gerüst (spanning tree) als Teilgraph des Netzwerkes. Für jeden Knoten p_i zeigt dann der Wert der Variable* parent *die Richtung zur Wurzel an, während die Werte von* children *die Wege zu den Blättern weisen.*

Algorithmus 7.1 Modifizierter Echoalgorithmus um in einem Netzwerk ein Gerüst mit gegebener Wurzel zu konstruieren: Algorithmus für Prozessor p_i mit $0 \leq i \leq n-1$.

Initially *parent* equals nil, and *children* and *other* equal to empty set.

upon receiving no message:
    if $i = r$ and *parent* is nil then // root has not sent M
        send M to all neighbors
        parent:= i

upon receiving M from neighbor p_j:
    if *parent* is nil then // p_i has not received M before
        parent:= j
        send (`<parent>`) to p_j
        send M to all neighbors except p_j
    else send (`<reject>`) to p_j

upon receiving (`<parent>`) from neighbor p_j:
    add j to *children*
    if *children* $\cup$ *other* contains all neighbors except *parent* then
        terminate

upon receiving (`<reject>`) from neighbor p_j:
    add j to *other*
    if *children* $\cup$ *other* contains all neighbors except *parent* then
        terminate

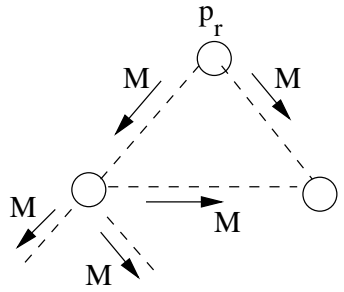


Abbildung 7.2: Mehrfachbesuch von Nachrichten in einem Knoten

Beweis:
(Skizze)
Vorbemerkung:

- $parent = nil \vee parent = j \in \mathbb{N}$. Wenn *parent*$= j$, dann bleibt *parent*$= j$.
- Am Anfang gilt $children = \emptyset$. Später verliert *children* nie ein Element.
- Die Menge der Kanten (p_i, p_j) mit $parent_j = i$ bilden einen gerichteten Teilgraph B, der nie eine Kante verliert.
- Nach endlicher Zeit wird für solche Kanten $j \in children_j$ erreicht.

Es gilt folgende Invariante:

($parent_r = nil \wedge B$ enthält weder Knoten noch Kanten) $\vee$
($parent = r \wedge B$ besteht nur aus p_r) $\vee$
(B ist Baum mit Wurzel p_r und für seine Kanten $p_i \longrightarrow p_j$ wurde p_i vor p_j zum ersten Mal von einer Nachricht erreicht.)

Zu zeigen ist:

- Die Invariante gilt zu Beginn.
- Die Invariante wird in keinem Schritt verletzt.
- B entält irgendwann alle Knoten, alle p_i terminieren und keine Nachrichten sind mehr unterwegs. □

Modifizierte Aufgabenstellung: falls nur Nachrichten verteilt werden sollen (ohne ein Gerüst zu markieren) muss die Nachricht $<$ *parent* $>$ nicht von der Nachricht $<$ *reject* $>$ unterschieden werden. Sie heissen dann oft $<$ *echo* $>$. M heisst zuweilen auch

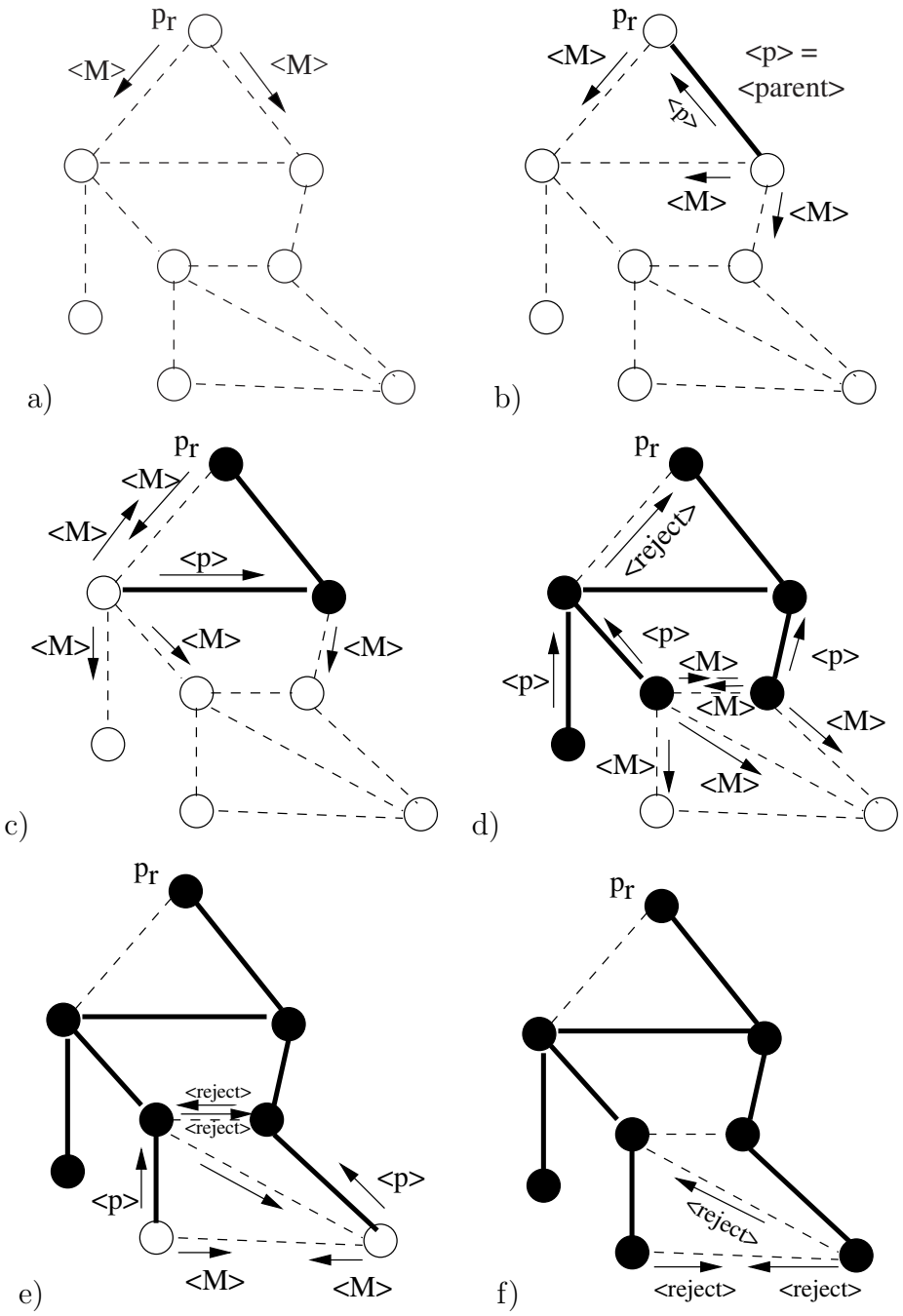


Abbildung 7.3: Ein Ablaufbeispiel zu Algorithmus 7.1

$<$ *explorer* $>$:

$$\left.\begin{array}{l}\texttt{<parent>}\\ \texttt{<reject>}\end{array}\right\} \rightarrow \texttt{<echo>} \qquad M :\rightarrow \texttt{<explorer>}$$

Maximale Anzahl der Nachrichten M in einem Netzwerk mit m Kanten und n Knoten:

$$\underbrace{2m}_{\text{2x alle Kanten}} - \underbrace{(n-1)}_{\text{Gerüstkanten}}$$

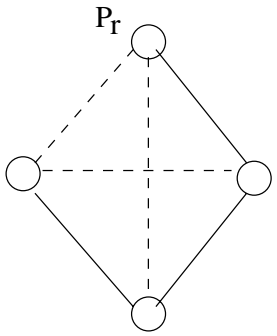
Beispiel: Im Netzwerk von Abbildung **??** gilt $m = 11, n = 8$, d.h. : $2m - (n-1) = 15$

tatsächlich: 13

Maximales m: $\frac{n(n-1)}{2}$ also $2\frac{n(n-1)}{2} - (n-1) = (n-1)^2$. Damit haben wir folgenden Satz:

Satz 7.4 *Es existiert ein asynchroner Algorithmus, um in einem Netzwerk mit m Kanten und n Knoten, um zu einem gegebenen Wurzelknoten ein Geüst zu konstruieren, dessen Nachrichtenkomplexität $O(m)$ bzw. $O(n^2)$ ist.*

BFS-Gerüst: BFS-Baum (breath-first search tree), d.h. die Entfernung der Knoten im Gerüst entspricht der kürzesten Entfernung im Netzwerk (Breitensuche)



DFS-Gerüst: DFS-Baum (depth-first search tree)

Abbildung 7.4: Gegenbeispiel zum BFS-Gerüst

Altenative zum Algorithmus 7.1: statt in neuen Knoten *alle* Nachbarn zu informieren, jetzt *nur einen* Nachbarn informieren:
Ergebnis: DFS-Gerüst.

(Tatsächliche Voraussetzung des Alg.7.2: der Wurzelknoten hat Nachbarn. Diese Einschränkung ist aber leicht zu beseitigen.)

Algorithmus 7.2 Modifizierter Echoalgorithmus um in einem Netzwerk ein Gerüst mit gegebener Wurzel und *Tiefensuche* zu konstruieren: Algorithmus für Prozessor p_i mit $0 \leq i \leq n-1$.

initially *parent* equals nil, *children* is empty and *unexplored* includes all the neighbors of p_i.

upon receiving no message:
    if $i = r$ and *parent* is nil then
        parent:= i
        let p_j be a processor in *unexplored*
        remove p_j from *unexplored*
        send M to p_j

upon receiving M from neighbor p_j:
    if *parent* is nil then // p_i has not received M before
        parent:= j
        remove p_j from *unexplored*
        if *unexplored* $\neq \emptyset$ then
            let p_k be a processor in *unexplored*
            remove p_k from *unexplored*
            send M to p_k
        else send (<parent>) to *parent*
    else send (<reject>) to p_j

upon receiving (<parent>) or (<reject>) from neighbor p_j:
    if received (<parent>) then add j to *children*
    if *unexplored*= $\emptyset$ then
        if *parent*$\neq i$ then send (<parent>) to *parent*
        terminate // DFS sub-tree rooted at p_i has built
    else
        let p_k be a processor in *unexplored*
        remove p_k from *unexplored*
        send M to p_k

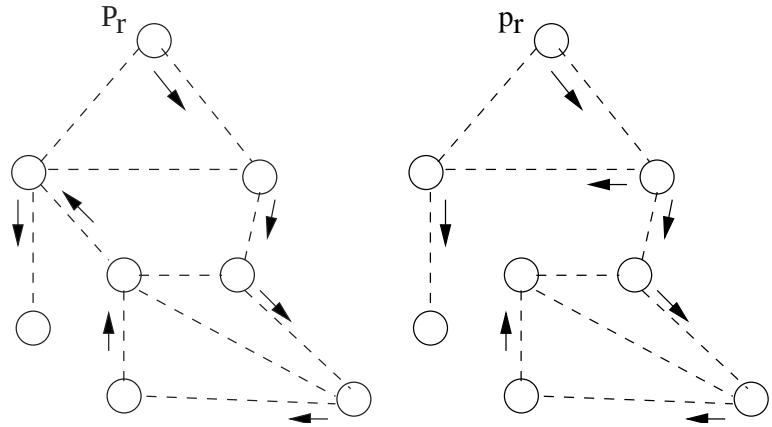


Abbildung 7.5: Beispiele für den Ablauf von Algorithmus 7.2

Satz 7.5 *Es existiert ein asynchroner Algorithmus um für ein Netzwerk mit m Kanten und n Knoten zu einem gegebenen Wurzelknoten ein DFS-Gerüst zu konstruieren (Alg. 7.2). Seine Nachrichtenkomplexität ist $O(m)$.*

Satz 7.6 *Es existiert ein asynchroner Algorithmus (Alg. 7.3) der für ein Netzwerk mit m Kanten und n Knoten ein Gerüst konstruiert,* ohne den Wurzelknoten vorher zu spezifizieren. *Seine Nachrichtenkomplexität ist $O(n \cdot m)$.*

Wichtige Vorraussetzung: Knoten haben *eindeutige* Bezeichner (z.B.: $id \in \mathbb{N}$).

Zeile 1+2: jeder Knoten kann spontan starten
Zeile 3: 'leader' ist er selbst

Jeder der n Knoten versucht ein DFS-Gerüst mit de Nachrichtenkomplexität vom Alg.7.3, also $O(m)$ zu konstruieren.

Variante:Übernehmen statt zerstören. (siehe Mattern [Mat89])

Termination nur für totalen Leader (Zeile 21). die anderen Knoten warten auf Nachrichten, *explizite* Termination durch Alg.7.1!

Problem: Wahl eines Repräsentanten (leader election)

Wahlproblem

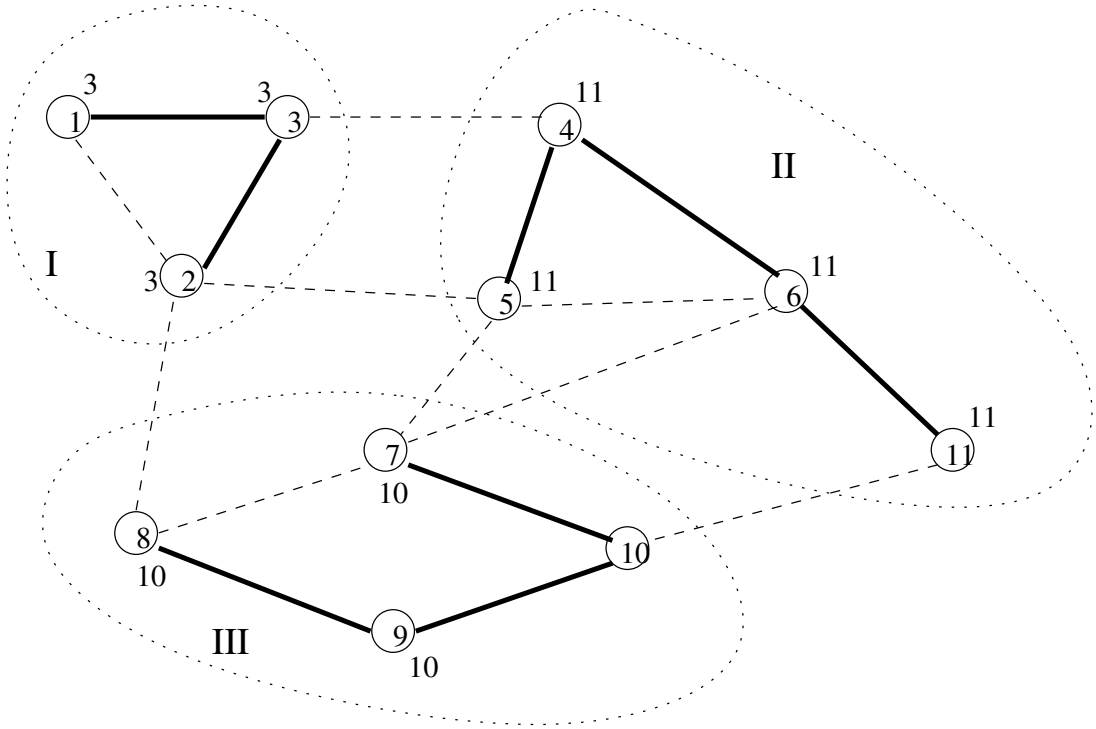


Abbildung 7.6: Möglicher Zwischenzustand bei Ablauf von Algorithmus 7.3

Algorithmus 7.3 Modifizierter Echoalgorithmus um in einem Netzwerk ein Gerüst *ohne* vorgegebene Wurzel zu konstruieren: Algorithmus für Prozessor p_i mit $0 \leq i \leq n-1$.

Initially *parent* equals nil, *leader* is 0, *children* is empty and *unexplored* includes all the neighbors of p_i.

upon receiving no message:
    if *parent* is nil then // wake up spontaneously
        leader :=id; *parent* := i
        let p_j be a processor in *unexplored*
        remove p_j from *unexplored*
        send (*leader*) to p_j

upon receiving (*new-id*) from neighbor p_j:
    if *leader* < *new-id* then // switch to new tree
        leader:=*new-id*; *parent*:= j
        unexplored:= all the neighbors of p_i except p_j // reset unexplored
        if *unexplored* $\neq \emptyset$ then
            let p_k be a processor in *unexplored*
            remove p_k from *unexplored*
            send (*leader*) to p_k
        else send (<parent>) to *parent*
    else if *leader* = *new-id* then send (<already>) to p_j // already in same tree
                // otherwise, *leader* > *new-id* and the DFS for *new-id*-is-.stalled

upon receiving (<parent>) or (<already>) from neighbors p_j:
    if received (<parent>) then add j to *children*
    if *unexplored*= $\emptyset$ then
        if *parent*$\neq i$ then send (<parent>) to *parent*
        else terminate as root of the spanning tree
    else
        let p_j be a processor in *unexplored*
        remove p_j from *unexplored*
        send leader to p_j

- sehr allgemein
- mit anderen bekannten Problemen verwandt

Bestimmung eines eindeutigen Repräsentanten in einer Gruppe von Knoten (Prozessen) in einem verteilten System.

Anwendung bzw. verwandte Probleme

- Verklemmung meherer Prozesse
- Monitor
- Koordinator
- Erneuerung eines Tokens

→ beruht meist auf Symmetriebrechnung

anderes Problem: konstruiere symmetrischen Algorithmus aus einem asymmetrischen → Symmetrisierung:

- Vorteile: einfacher zu implementieren
  fehlertoleranter
- Voraussetzung für spezielles Gerüstproblem (Wurzel ist bekannt!)
- Wird gelöst durch allgemeines Gerüstproblem (Wurzel wird ermittelt)

etwas spezieller: Maximumsproblem: Knoten linear geordnet (durch Werte, Bezeichner). Bestimme maximalen Knoten!

In Symmetria, einem vollkommen föderalistischen Staat, existiert keine permanente Hauptstadt. Die Gesamtzahl der Städte ist unbekannt, jedoch sind diese durch ein Straßennetz miteinander verbunden, über welches auch die Kommunikation in Form von Boten läuft. Zwar leben die Einwohner von Symmetria glücklich und zufrieden in ihrem Zustand der perfekten Anarchie, jedoch erfordert eine lästige internationale Verpflichtung, dass ein fremder Staatspräsident seinen Staatsbesuch in der Hauptstadt des

besuchten Staates absolviert. Dieses wird in Symmetria nun jeweils extra zu diesem Zweck temporär bestimmt, der Einfachheit wegen soll es die Stadt mit dem dicksten Bürgermeister sein. Glücklicherweise zeigt sich bei ganz genauem Nachmessen stets, daß keine zwei Bürgermeister gleich dick sind. Das Problem für die symmetrianischen Verwaltungsbeamten besteht nun darin, ein möglichst effizientes Schema zu organisieren, durch das die Hauptstadt eindeutig ermittelt und allen Bürgermeistern bekannt gegeben werden kann, so daß diese den eintreffenden Staatsgast zur Hauptstadt geleiten können[1]. Da der mit dem Flugzeug aufs Geratewohl in irgendeiner Stadt gelandete Staatsgast meistens umständlich zur neuen Hauptstadt weitergeleitet werden muß überlegen sich die Einwohner von Symmetria eine naheliegende Optimierung: Es wird nicht bereits dann eine Hauptstadt gewählt, wenn sich der Staat gast ankündigt, sondern erst dann, wenn diese gelandet ist und somit ein akuter Bedarf entsteht, bewirbt sich diese Stadt als Hauptstadt. Da es möglicher andere "gleichzeitige" Bewerber um den Status der Hauptstadt gibt (nämlich dann, wenn ein weiterer Staatsgast irgendwo anders eintrifft), es aber niemals zwei oder mehr Hauptstädte gleichzeitig geben darf, muß eine Wahl unter den Kandidaten stattfinden. Da die Symmetrianer ein praktisch veranlagtes Volk sind, lösen sie dies so, dass der Bürgermeister einer Stadt, die sich um den Status bewirbt, statt seines wahren Körperumfanges den hundertfachen Wert angibt, ansonsten jedoch das bisherige bewährte Verfahren beibehalten wird. Auf diese Weise ist sichergestell, daß nur eine sich bewerbende Stadt Hauptstadt werden kann.

Wahlalgorithmus:

- Die Endzustände jedes Knotens sind aufgeteilt in "ausgewählt"(elected) und "nicht ausgewählt" (not-elected). Wird ein Zustand dieser Gruppe erreicht, dann wird die Gruppe nicht mehr verlassen.

- In jeder zulässigen Berechnung erreicht genau ein Knoten einem Zustand der "ausgewählt"-Gruppe, die anderen aber die andere Gruppe.

[1]Der Umfang der Bürgermeister kann sich dynamisch verändern; zur eindeutigen Bestimmung einer Hauptstadt ist es schließlich unerheblich, ob dann, wenn die Hauptstadt bestimmt ist, der Bürgermeister noch immer der umfangreichste ist.Damit es nicht zu Inkonsistenzen kommt, sollte ein Bürgermeister allerdings stets die gleichen Angaben machen, wenn er während eines Wahlvorganges mehrfach gefragt werden sollte!

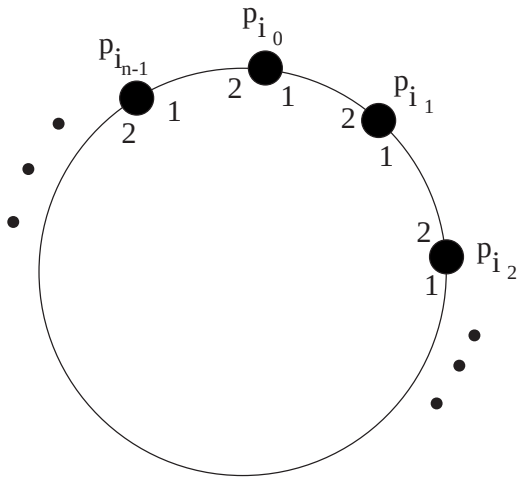


Abbildung 7.7: Netzwerk mit Ringstruktur

Ein Algorithmus heißt *anonym*, wenn die Knoten nicht unterscheidbar sind
d.h. gleiche Algorithmen in allen Knoten
gleiche Prozess-Bezeichner
usw

Ein Algorithmus heißt *uniform*, wenn die Anzahl n der Knoten nicht bekannt/benutzt wird.

also z.B. anonym *und* nicht uniform:
alle Algorithmen gleich für ein $n \in \mathbb{N}$, aber verschieden für verschiedenene $n \in \mathbb{N}$.

Wie die folgenden Sätze zeigen, gibt es keine nicht uniformen und ananoyme Auswahl-Algoritmen. Das gilt sogar für den synchronen Fall!

Satz 7.7 *Es gibt keinen synchronen, nicht-uniformen anonymen Auswahl-Algorithmus.*

Beweis:
Nach k-Runden haben alle Knoten den selben Zustand erreicht, denn

- sie hatten gleichen Anfangszustand,
- gleiche Nachrichten waren unterwegs und

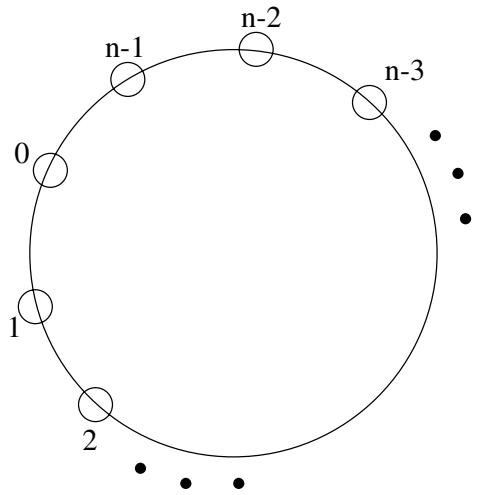
- gleiche Folgezustände wurden pro Runde erreicht.

Also bei Termination: alle sind ausgewählt oder keiner! □

Korollar 7.8 *Es gibt keine asynchronen, anonymen Auswahl-Alg. in beliebigen Netzwerken (egal ob uniform oder nicht uniform).*

Ein $O(n^2)$ Auswahl-Algorithmus (für asynchrone, nicht anonyme Ringe):

Algorithmus 7.5 ist ein Auswahlalgorithmus in einem asynchronen und gerichteten Ring. Wie die folgende Überlegung zeigt, werden im ungünstigsten Fall $O(n^2)$ Nachrichten verschickt:



$n-1$: n Nachrichten
$n-2$: $n-1$ Nachrichten
$n-3$: $n-2$ Nachrichten
⋮ : ⋮
2 : 3 Nachrichten
1 : 2 Nachrichten
0 : 1 Nachrichten

insgesamt: $\sum_{i=0}^{n-1}(n-i)+n = \Theta(n^2)$ Nachrichten

Abb. 7.8 zeigt den Algorithmus als gefärbtes Netz, im Stil von [Rei98] konstruiert mit dem Renew-Werkzeug. Die Graphik des Ringes dient nur der Veranschauung (und Animation bei der Simulation). Das (Petri)Netz fasst die Information vieler Knoten zusammen. Ein Paar $[i, j]$ als Marke bedeutet folgendes: i ist der Name des Knotens (Prozessors) p_i, j ist der Bezeichner des Knotens p_j, den p_i als temporären Kandaten für die Auswahl (leader) gewählt hat. Die Transition zwischen den Plätzen *outgoing* und *candidate* bindet ein solches Paar in den Variablen $[myid, id]$ und bringt die Menge von Paaren $(t.prod(t.neighbors(myid), id)$ nach *candidate*. Dies ist die Menge der Paare $[k, j]$, wobei

Algorithmus 7.4 ein $O(n^2)$ Auswahl-Algorithmus für asynchrone, nicht-anonyme, gerichtete Ringe

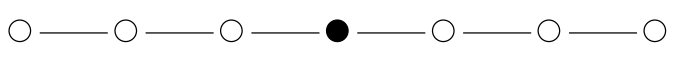
Algorithmus für Prozessor p_i mit Bezeichner i:
- sende eigene Bezeichner an linken Nachbar
- bei Empfang einer Zahl, sende sie an linken Nachbar,
  falls sie größer als der eigene Bezeichner ist.
  falls sie kleiner ist: nichts tun
  falls sie gleich ist: gehe in Endzustand "ausgewählt" und teile dies den anderen mit.

Kommentar: Der Algorithmus ist uniform.

k ein Nachfolger im Netzwerk von i (genauer p_i) und j der Bezeichner id von j ist. (In diesem Beispiel ist diese Menge immer einelementig. ($t.prod(t.neighbors(myid), id)$ ist eine eingebaute Funktion, die die Struktur des Netzwerkes als gerichteter Graph enthält. Sie kann für jedes Beispielnetz verändert werden.) Die Transition stellt also die Übermittlung des Kandidaten j von i an den Nachfolger von i dar.

Ein $O(n \log n)$ Auswahl-Algorithmus (für asynchrone, nicht anonyme Ringe):

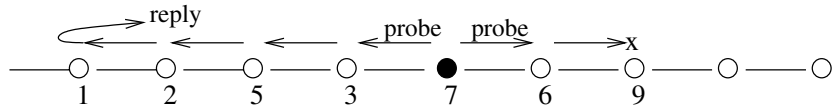
Definiere: *k-Nachbarschaft*: Menge der Knoten mit *maximalen* Abstand k.



$k = 3$
$2k + 1 = 7$ Knoten

In Phase $l = 1, 2, 3, \ldots$ ist ein Repräsentant (temporärer Leader) einer 2^l-Nachbarschaft zu bestimmen.

$l = 2 \quad 2^l = 4 \quad$ "Sonde"



Analyse der Nachrichtenzahl:

- Anzahl der Nachrichten vom einem Knoten in Phase l : $4 \cdot 2^l$
- Anzahl der Knoten, die Nachrichten aussenden in Phase $k+1$: $\frac{n}{l^k+1}$ (so viele temporäre Leader!)

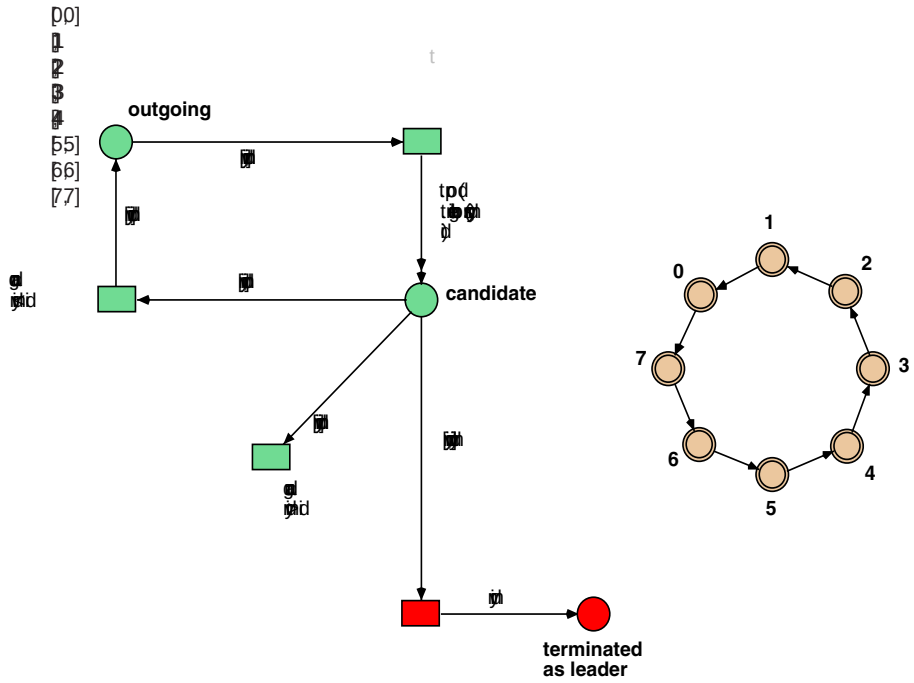
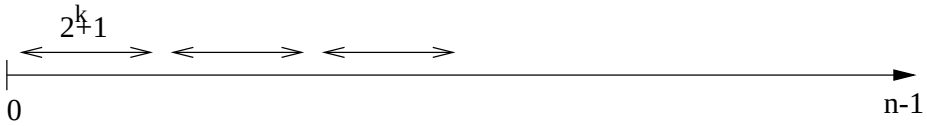


Abbildung 7.8: Der $O(n^2)$ Ring-Wahlalgorithmus als gefärbtes Netz (Renew)



- nur noch ein leader, wenn $n = 2^k + 1$, d.h. $k = \log n - 1$, also insgesamt:

$$\underbrace{4n}_{\text{Phase 0}} + \underbrace{\sum_{l=1}^{\log n - 1} \left(4 \cdot 2^l \cdot \frac{n}{2^{l-1}+1} \right)}_{\textit{Phase 1 bis Ende}} \underbrace{+\ n}_{Termination} \leq 8n \log n$$

Satz 7.9 *Der asynchrone Wahlalgorithmus 7.5 hat die Nachrichtenkomplexität* $O(n \log n)$.

Abb. 7.9 zeigt den Algorithmus 3.1 als gefärbtes Netz in ähnlicher Weise wie Abb. 7.8. $2'[i, l]$ in der Anfangsmarkierung bedeutet 2 Nachrichten des Knotens i, die er nach verschiedenen Richtungen einer l-Nachbarschaft versenden wird. Anders als im Algorithmus 7.5 wird die Nachbarschaftsgröße in einer neuen Runde einfach verdoppelt (und nicht als 2^l berechnet).

Algorithmus 7.5 ein $O(n * logn)$ Auswahl-Algorithmus für asynchrone, nicht-anonyme, ungerichtete Ringe

Algorithmus für Prozessor p_i, $0 \leq i < n$] Initially, *asleep*=true

upon receiving no message:
    if *asleep* then
        asleep:=false
        send <probe,id,0,1 > to left and right

upon receiving <probe,j,l,d > from left (resp., right):
    if $j = id$ then terminate as the leader
    if $j > id$ and $d < 2^l$ then // forward the message
        send <probe,j,l,$d + 1$ > to right (resp., left)
    if $j > id$ and $d \geq 2^l$ then // reply to the message
        send <reply,j,l > to left (resp., right)
                // if $j < id$, message is swallowed

upon receiving <reply,j,l> from left (resp., right):
    $j \neq id$ then send <reply,j,l > to right (resp., left) // forward the reply
    else // reply is for own probe
        if already received <reply,j,l > from right (resp., left) then
            send <probe,id,$l + 1$,0 > // temporary leader at end of phase l

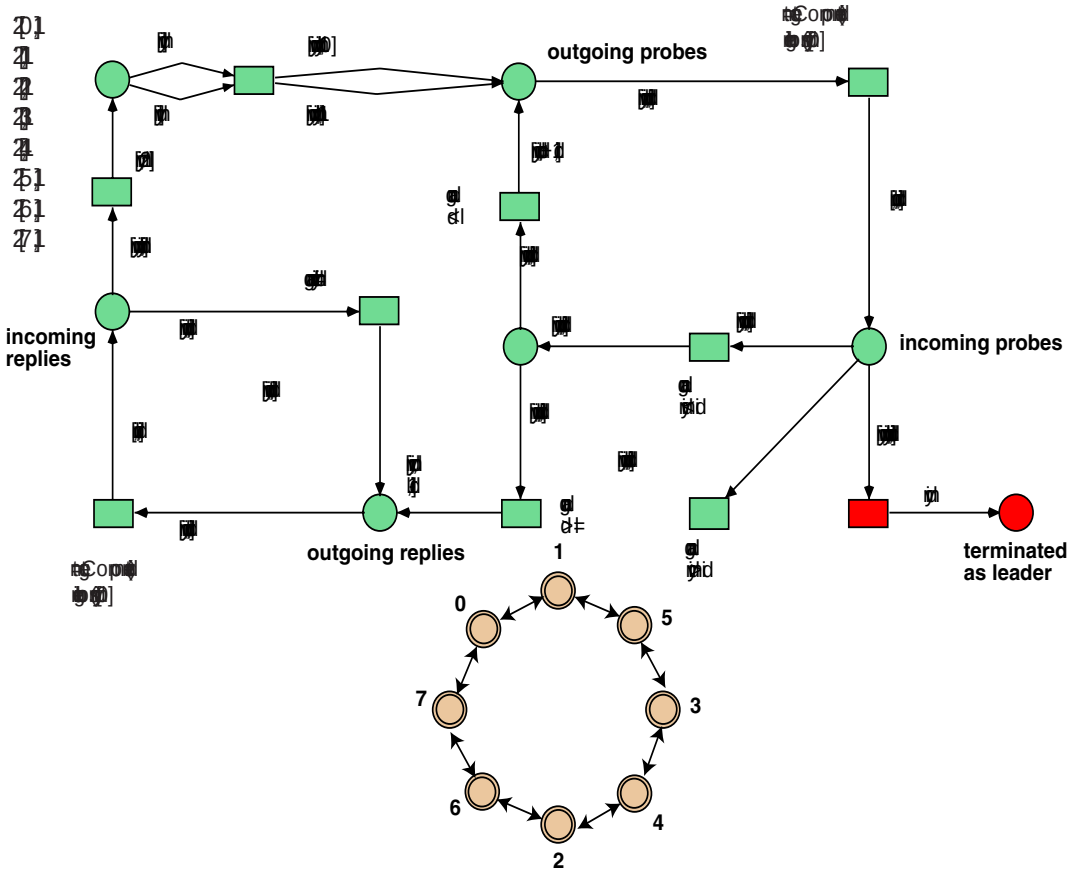


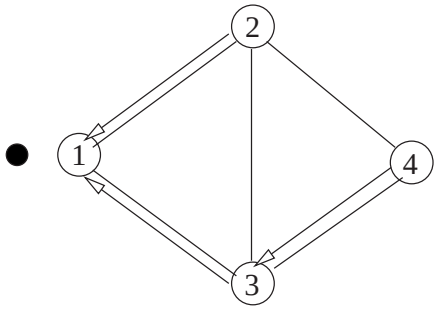
Abbildung 7.9: Der $O(n * log\ n)$ Ring-Wahlalgorithmus als gefärbtes Netz (Renew)

7.3 Verteilter wechselseitiger Ausschluss

Wie beim wechselseitigen Ausschluss mit Speichersynchronisation enthalten die Prozessoren in ihrem Algorithmus einen kritischen Abschnitt und es soll erreicht werden, dass nie zwei verschiedene diesen gleichzeitig benutzen. Der in Abb. 7.10 dargestellte verteilte Algorithmus benutzt ein Gerüst auf dem Netzwerk, dessen Wurzelknoten den kritischen Abschnitt betreten darf. Jeder Knoten (Prozessor) kennt die Richtung zum Wurzelknoten. Diese wird durch Paare $[i, j]$ in der Menge $\bar{N}$ im Platz *compass* dargestellt: vom Knoten i geht es in Richtung j zur Wurzel. Am Anfang sind die Knoten inaktiv, d.h. $1, 2, 3$ und 4 liegen in dem Platz *quiet*.

In Abbildung 7.11 ist eine lokalisierte Form des Netzes von Abb. 7.10 dargestellt, d.h. das Netz ist für jeden Knoten separat gezeichnet. Dafür kann die Variable x jeweils nur die jeweilige Knotennummer als Wert annehmen. Beispieleweise kann x in dem mit 1 bezeichneten Teilnetz nur die Belegung $[x = 1]$ annehmen. Dadurch wären natürlich weitere Vereinfachungen möglich. Andererseits sind die Verbindungskanäle nur exemplarisch für Nachrichten von Knoten 1 zu Knoten 2 für das Verschieben des Tokens und von Knoten 2 zu Knoten 1 für die Aktualisierung von *pending* dargestellt. Die übrigen Verbindungen sind entsprechend zu ergänzen.

Beispiel: Baum



$$\begin{aligned} N &= \{(1,2),(1,3),(3,4)\} \\ \overline{N} &= \{(2,1),(3,1),(4,3)\} \\ &= \text{Kompass (im Graphen eingezeichnet)} \end{aligned}$$

a) Das Token ist in Knoten 1.
b) Knoten 4 werde aktiv: $(4,4) \in$ pending $\rightarrow (3,4) \in$ pending (siehe Abb.
c)$\rightarrow (1,3) \in$ pending und $\rightarrow$ token nach 3
d)$\rightarrow$ token nach 4

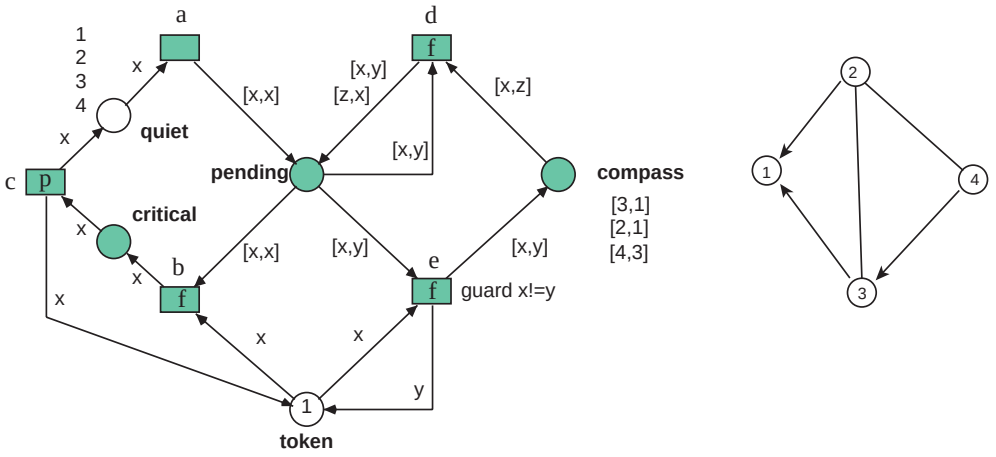


Abbildung 7.10: Verteilter wechselseitiger Ausschluss: treemutex

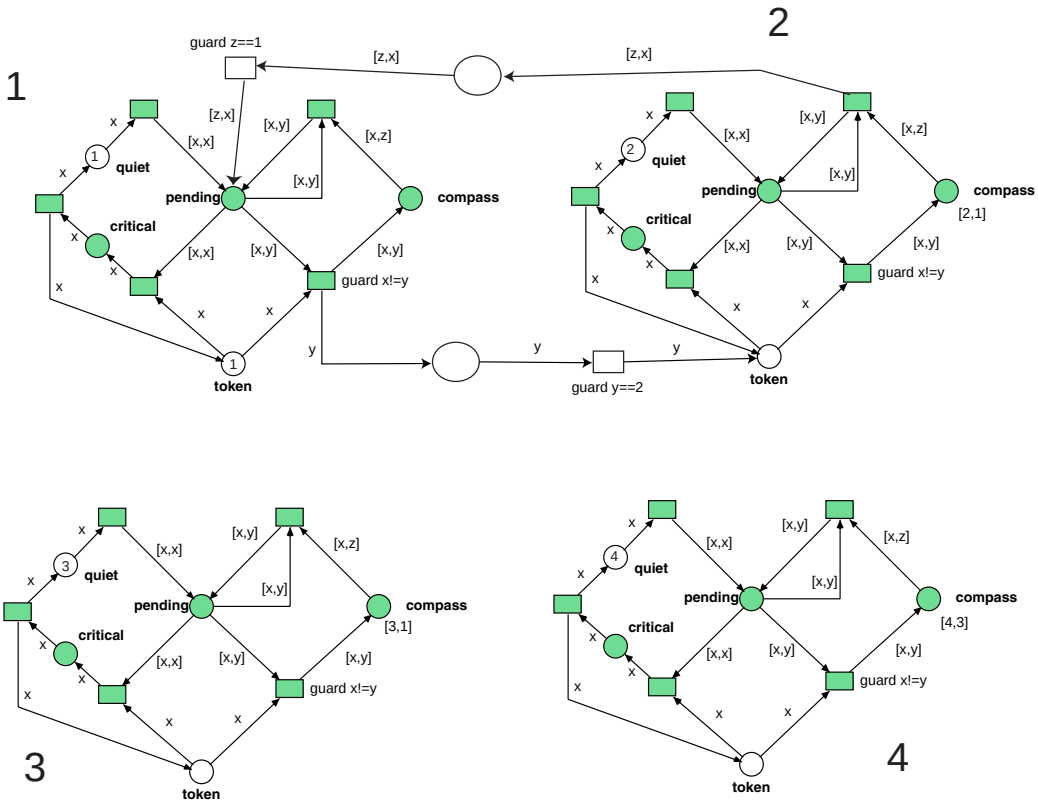


Abbildung 7.11: Lokalisierte Form des Netzes von Abb. 7.10

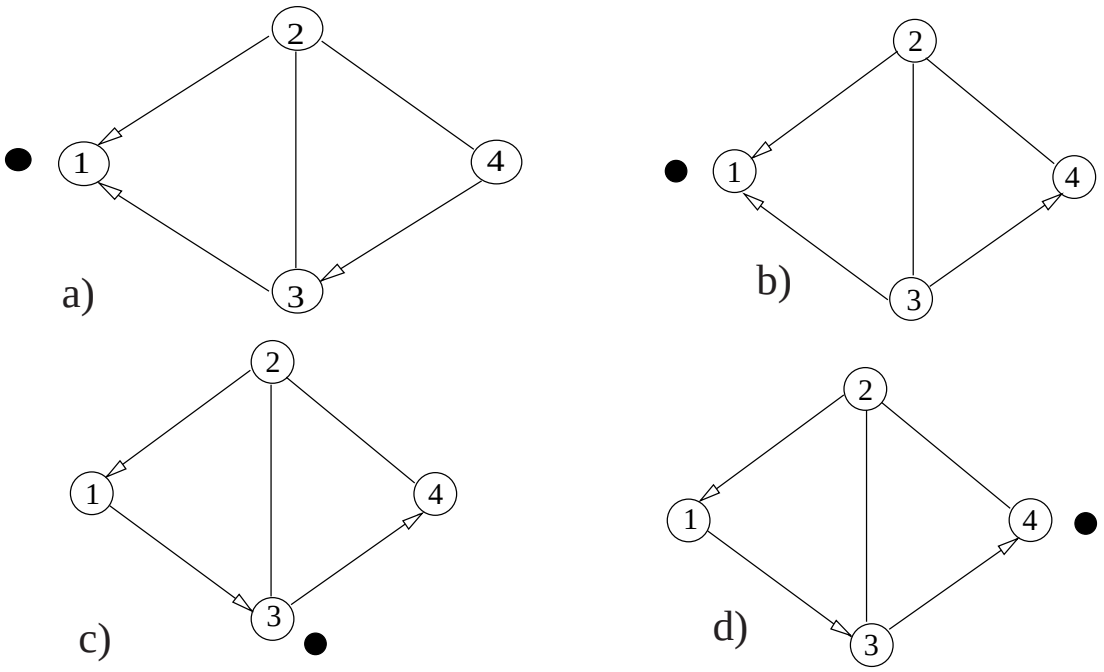


Abbildung 7.12: Zwischenzustände eines Ablaufs von Algorithmus 7.10 für das Netzwerk von Seite 202.

erweiterte Schaltregel:

- Transition a: **normal** (sie muss nicht schalten):
  der Zustand *quiet* kann, muss aber nicht verlassen werden;
  graphisch: [-]
- Transition c: **produktiv** (siehe Definition 4.15 a) auf Seite 83):
  der kritische Abschnitt muss mal freigegeben werden;
  graphisch: [p]
- Transitionen b,d,e: **fair** (siehe Definition 4.15 b) auf Seite 83):
  alle Anforderungen sollen nach endlicher Zeit berücksichtigt werden;
  graphisch: [f]

Lokaler wechselseitiger Ausschluss in Netzwerken

Voraussetzung:

- beliebiges, ungerichtetes Netzwerk
- jeder Netzwerkknoten hat einen Zustand, "critical"
- je zwei Knoten sind *genau dann* benachbart, wenn sie ein Betriebsmittel teilen.

Problem: jeder Knoten kann aktiv werden
(durch einen Zustandsübergang: "quiet"→ "pending" und geht er nach endlicher Verzögerung in "critical" mit *allen* Betriebsmitteln, die ihm zugeordnet sind.)

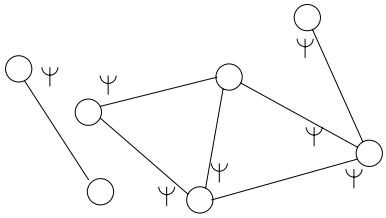
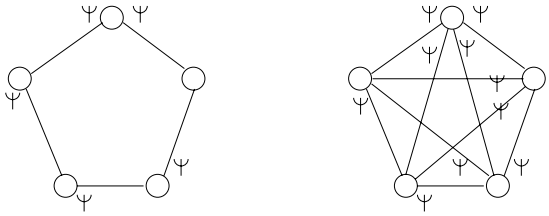


Abbildung 7.13: Betriebsmittelzuordnung: a) ein allgemeiner Fall



b) c)

Abbildung 7.14: Betriebsmittelzuordnung: b) die 5 Philosophen, c) globaler wechselseitiger Ausschluss

Darstellung des gemeinsamen Betriebsmittels von u und v:

$$(u, v) \text{ wenn bei } u, \ (v, u) \text{ wenn bei } v.$$

Durch $r(x)$ werden mit c alle BM belegt und durch d wieder freigegeben.

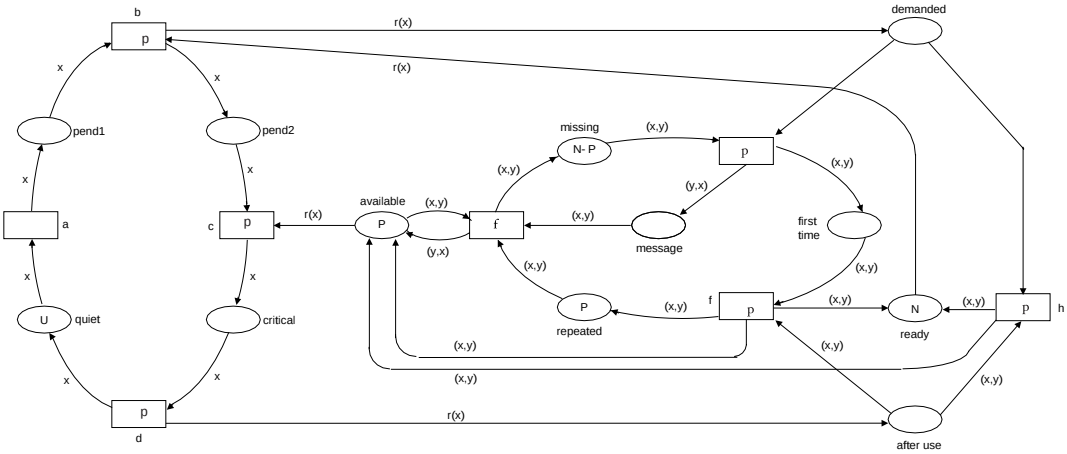
(größere Darstellung des Netzes in Abbildung 7.15)

Jedes BM (u, v) bzw. (v, u) ist immer in einem von drei Zuständen:

1. $(u, v) \in$ 'repeated': BM bei u und v war der letzte Nutzer
   Durch eine Nachricht (u, v) in 'message' kann die Transition g des BM; an v übergeben.

2. $(u, v) \in$ 'missing': BM bei v. Falls u das BM wünscht, wird die Anforderung $(u, v) \in$ 'demanded' nach 'messages' gebracht, und (u, v) von 'missing' nach 'first time'.

3. $(u, v) \in$ 'first time': das BM ist von u angefordert und wird nicht abgegeben, bevor es durch u genutzt wurde.



sort	$site$	$N = N^{-1}$
sort	$neighbors = site \times site$	$N_1 = U$
const	$U : set\ of\ sites$	$P \cup P^{-1} = N$
const	$N, P : set\ of\ neighbors$	$P \cap P^{-1} = \emptyset$
fct	$r : site \to set\ of\ neighbors$	$xP^+y \to x \neq y$
var	$x, y : site$	$r(x) = (x) \times N(x)$

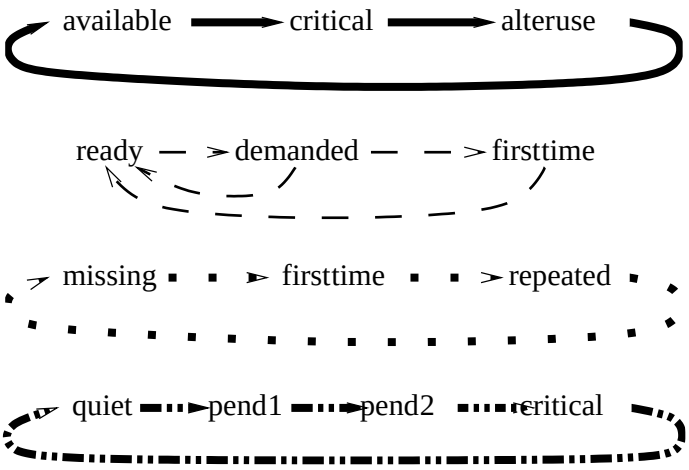
Hier Invarianten als Flüsse darstellbar:

Betriebsmittel:

Nachrichten:

Zustände 1:

Zustände 2:



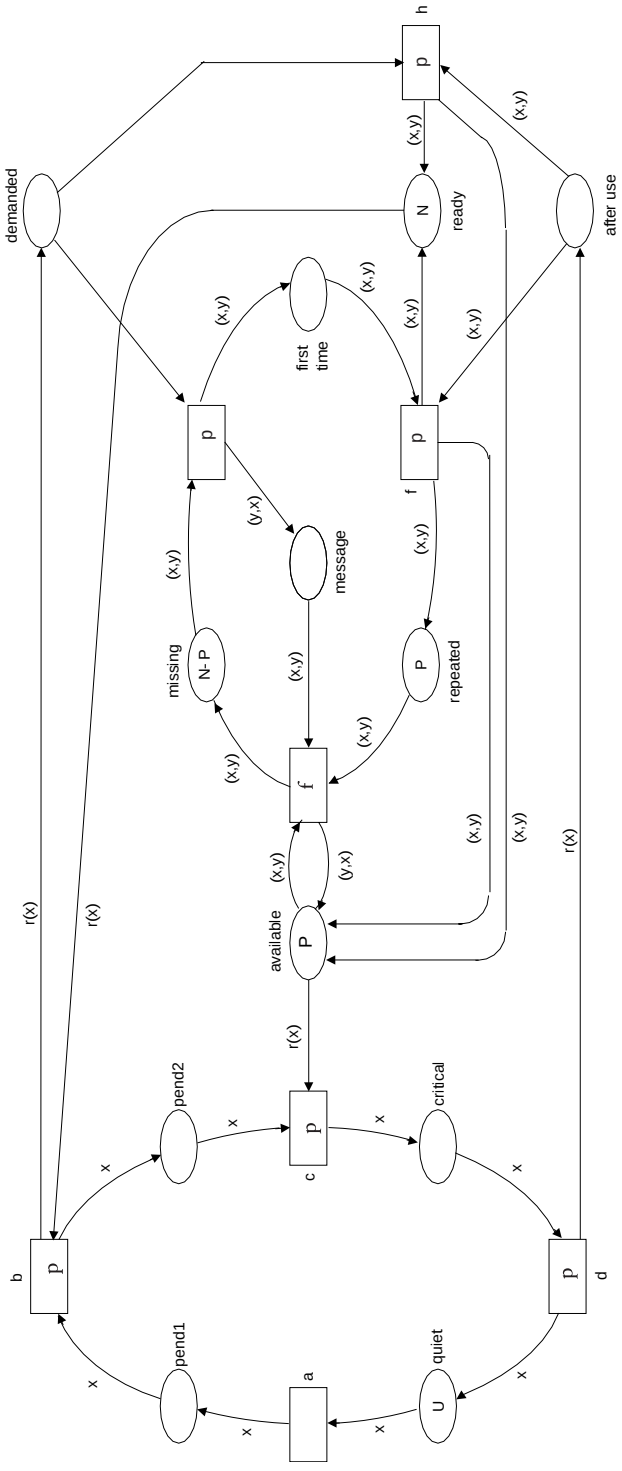


Abbildung 7.15: Netz für lokalen wechselseitigen Ausschluss

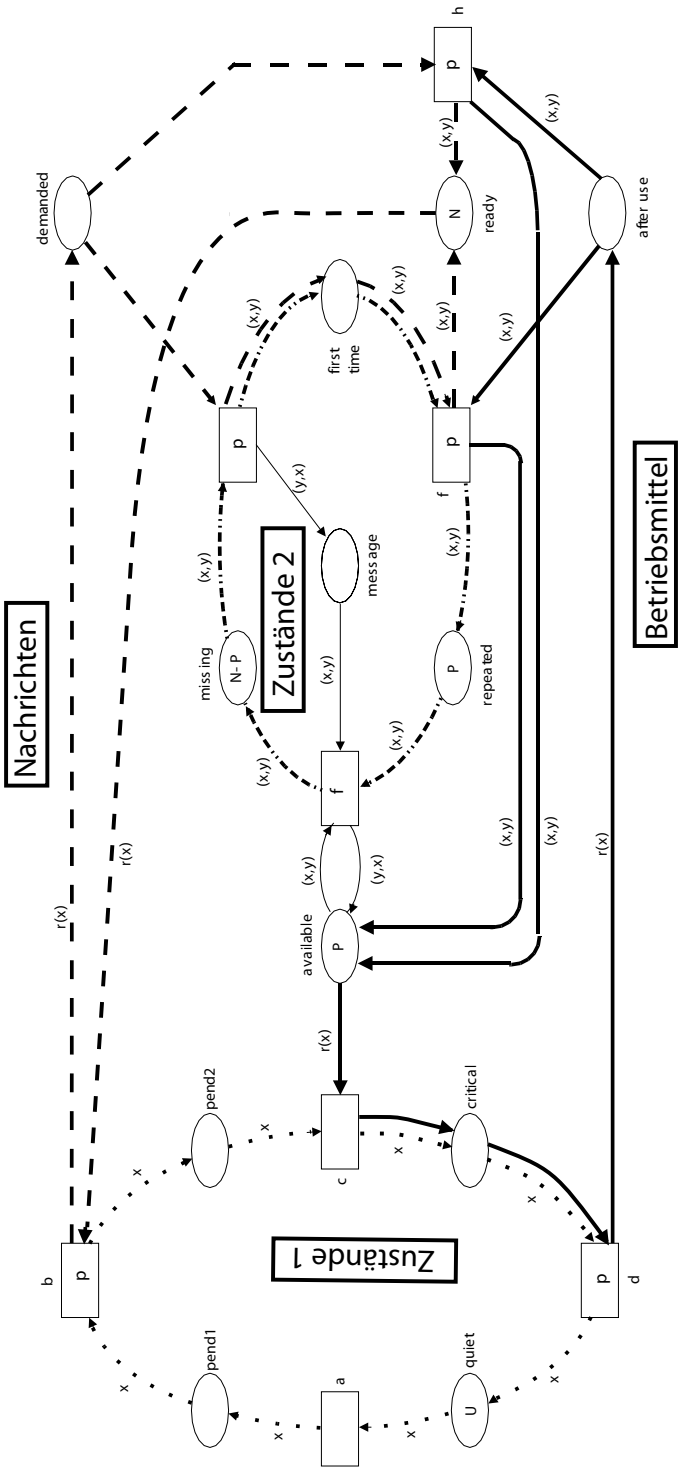


Abbildung 7.16: Netz für lokalen wechselseitigen Ausschluss mit Invarianten

Literaturverzeichnis

[Aal97] W.v.d. Aalst. Verification of Workflow Nets. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *LNCS*, pages 407–426. Springer-Verlag, Berlin, 1997.

[Aal98] W.v.d. Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8:21–66, 1998.

[Aal00] W.v.d. Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In W.v.d. Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management*, volume 1806 of *LNCS*, pages 161–183. Springer-Verlag, Berlin, 2000.

[AG93] Burns A. and Davis G.L. *Concurrent Programming.* Addison-Wesley, 1993.

[AW98] H. Attiya and J. Welch. *Distributed Computing.* McGraw-Hill, 1998.

[Bar80] J.G.P. Barnes. An Overview of ADA. *Software-Practice and Experience*, 10:851–887, 1980.

[BH73] P. Brinch-Hansen. *Operating System Principles.* Prentice Hall, Englewood Cliffs, 1973.

[BH77] P. Brinch-Hansen. *The Architecture of Concurrent Programs.* Prentice Hall, Englewood Cliffs, 1977.

[BR81] E. Best and B. Randell. A Formal Model of Atomicity in Asynchronous Systems. *Acta Informatica*, 16:93–124, 1981.

[BS69] K.A. Bartlett and R.A. Scantlebury. A note on reliable full-duplex transmission over half-duplex links. *Comm. ACM*, 12:260–261, 1969.

[BSW79] P.A. Bernstein, D.W. Shipman, and W.S. Wong. Formal Aspects in Serializability in Database Concurrency Control. *IEEE SE-5*, 3:203–216, 1979.

[CH74] R.H. Campbell and A.N. Habermann. The specification of Process Synchronization by Path Expressions. In *Lecture Notes in Computer Science*. Springer, 1974.

[Cha92] P. Chaudhuri. *Parallel algorithms*. Prentice Hall, 1992.

[Che84] D.R. Cheriton. The V-Kernel. *IEEE Software*, 1:19–46, 1984.

[Dij68] E.W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press N.Y., 1968.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18:453–457, 1975.

[Dij77] E.W. Dijkstra. A Correctness Proof for Communicating Processes - A small Exercise. Technical report, EDW-604, 1977.

[Dij78] E.W. Dijkstra. Finding the correctness proof of a concurrent program. In *Lecture Notes in Computer Science*. Springer, 1978.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. ACM*, 19:624–633, 1976.

[Flo67] R.W. Floyd. Assigning meanings to programs. In *Proc. AMS Symp. in Applied Math, Amer. Math. Soc.*, pages 19–31, 1967.

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[GS93] A. Gibbons and P. Spirakis. *Lectures on parallel computation*. Cambridge University Press, 1993.

[GV01] C. Girault and R. Valk. *Petri Nets for Systems Engineering - A Guide to Modelling, Verification, and Applications*. Springer, 2001.

[Hai82] B.T. Hailpern. Verifying concurrent processes using temporal logic. In *Lecture Notes in Computer Science 129*. Academic Press N.Y., 1982.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12:576–583, 1969.

[Hoa74] C.A.R. Hoare. Monitors, an Operating System Structuring Concept. *Comm. ACM*, 17:549–557, 1974.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Comm. ACM*, 21:666–677, 1978.

[Hol83] R.C. Holt. *Concurrent Euclid, the Unix System and Tunis*. Addison-Wesley, 1983.

[Hor83] E. Horowitz. *Fundamentals of Programming Languages*. Springer, 1983.

[HV87] D. Hauschildt and R. Valk. Safe states in banker-like resource allocation problems. *Information and Computation*, 75:232–263, 1987.

[Jáj92] J. Jájá. *An introduction to parallel algorithms*. Addison-Wesley Publ. Co., 1992.

[JV87] E. Jessen and R. Valk. *Rechensysteme – Grundlagen der Modellbildung*. Springer-Verlag, Berlin, 1987.

[Lom77] D.B. Lomet. Process structuring, synchronization and recovery using atomic actions. *ACM Sigplan Notices*, 12:128–137, 1977.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mat89] F. Mattern. *Verteilte Basisalgorithmen*. Springer, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π - Calculus*. Cambridge Univerity Press, 1999.

[MK99] J. Magee and J. Kramer. *Concurrency - State Models & Java Programs*. John Wiley & Sons, 1999.

[Obe81] H. Oberquelle. More on the readability of net representations. *Petri Nets and Related System Models*, 9:5–8, October 1981.

[Pap79] C.H. Papadimitriou. The Serializability of Concurrent Data Base Updates. *J. ACM*, 26(4):631–653, 1979.

[PLS79] P.R. Torrigiani P.E. Lauer and M.W. Shields. COSY - A System Specification Language Based on Paths and Processes. *Acta Informatica*, 12:109–158, 1979.

[Rei85] W. Reisig. *Petrinetze*. Springer, 1985.

[Rei93] J. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[Rei98] W. Reisig. *Elements of Distributed Algorithms*. Springer, 1998.

[Set82] R. Sethi. Useless Actions Make a Difference: Strict Serializability of Database Updates. *J. ACM*, 29(2):349–403, 1982.

[Tay83] R.N. Taylor. Complexity of Analyzing the Synchronization Structure of Concurrent Programs. *Acta Informatica*, 19:57–84, 1983.

[VJ85] R. Valk and M. Jantzen. The Residue of Vector Sets with Applications to Decidability Problems in Petri Nets. *Acta Informatica*, 21:643–674, 1985.

[Wir99] N. Wirth. Concurrency. In R.W. Sebesta, editor, *Concepts of Programming Languages*. Addison-Wesley, 1999.