

Formale Grundlagen der Informatik II

Modellierung Analyse

Wintersemester 2013/2014

Daniel Moldt und Rüdiger Valk



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Dieses Skript (Version vom 10. Oktober 2013) basiert auf einer Vorlage von Rüdiger Valk und wurde von M. Köhler-Bußmeier, D. Moldt und R. Valk überarbeitet und ergänzt. Auf dieses Skript kann unter der folgenden URL zugegriffen werden:
<http://www.informatik.uni-hamburg.de/TGI/lehre/>

Einige in diesem Skript benutzte Symbole:

	klein	groß	
Alpha	α		natürliche Zahlen $\mathbb{N} = \{0, 1, 2, 3, \dots\}$
Beta	β		positive natürliche Zahlen
Gamma	γ	Γ	$\mathbb{N}^+ = \{1, 2, 3, \dots\}$
Delta	δ	Δ	ganze Zahlen $\mathbb{Z} = \{\dots - 2, -1, 0, 1, 2, \dots\}$
Lambda	λ	Λ	rationale Zahlen \mathbb{Q}
Pi	π	Π	reelle Zahlen \mathbb{R}
Rho	ρ		Zeichen „Partiell“ ∂
Sigma	σ	Σ	$ M $ bezeichnet die Mächtigkeit einer
Tau	τ		Menge M .
Phi	ϕ, φ	Φ	
Chi	χ		
Psi	ψ	Ψ	
Omega	ω	Ω	

Der Begriff des Vektors wird hier weiter gefasst als in manchen Vorlesungen und Büchern der Mathematik, wo er streng auf Vektorräume wie \mathbb{Q}^n oder \mathbb{R}^n beschränkt wird. Hier werden auch Tupel $\mathbf{m} = (x_1, \dots, x_n) \in V$ als Vektoren bezeichnet, wobei $V = A^B$ (z.B. $A^{\{1, \dots, n\}}$ bzw. A^n), gilt, d.h. $x_1, \dots, x_n \in A$ und $B = \{b_1, \dots, b_n\}$ geordnet ist.

Auf A definierte Operationen werden folgendermaßen auf A^B übertragen. Für Vektoren $\mathbf{m}_1, \mathbf{m}_2 \in A^B$ werden die Operatoren $+, -, =, \leq$ jeweils komponentenweise erklärt, d.h. z.B. $\mathbf{m}_1 \leq \mathbf{m}_2$, falls $\forall b \in B : \mathbf{m}_1(b) \leq \mathbf{m}_2(b)$. Lediglich $\mathbf{m}_1 < \mathbf{m}_2$ steht für $(\mathbf{m}_1 \leq \mathbf{m}_2 \text{ und } \mathbf{m}_1 \neq \mathbf{m}_2)$ (also hier ausnahmsweise nicht komponentenweise!).

Im Folgenden werden Referenzen auf Texte gegeben, auf denen die Abschnitte dieses Skriptes u.a. beruhen. Dies ist eine Minimalangabe. Weitergehende Literatur ist in den Kapiteln zu finden.

- **Transitionssysteme, Temporallogik, Model-Checking**
[BK08],
[CGP99]: Kapitel 2 – 5,
[HR04]
- **Partielle Ordnungen**
[AW98]: Kapitel 6
- **Petrinetze und Verifikation**
[GV03]: Kapitel 2 – 5,
[Jen94] Stelleninvarianten (Seiten 113 ff),
[Rei10],
[JK09],
[Kum01]
- **Prozessalgebra**
[Fok99]: Kapitel 2 – 6
- **Parallele Algorithmen**
[Gru97]: Kapitel 4,
[Jáj92]

Inhaltsverzeichnis

1 Endliche Automatenmodelle	1
1.1 Endliche Automaten	1
1.2 Büchi-Automaten	5
2 Transitionssysteme und Kripke-Strukturen	13
2.1 Transitionssysteme	13
2.2 Bisimilarität bei reaktiven Systemen	15
2.3 Synchronisation von Transitionssystemen	19
2.4 Etikettierte Transitionssysteme	25
2.5 Kripke-Strukturen	27
2.6 Kripke-Strukturen von Programmen	31
3 Temporallogik	39
3.1 Temporale Logik	40
3.2 Linear Time Logic (LTL)	41
3.3 Computation Tree Logic (CTL)	45
3.4 Ausdrucksmächtigkeit: LTL vs. CTL	48
4 Model-Checking	51
4.1 Automatentheorie und LTL-Model-Checking	51
4.2 CTL-Model-Checking	58
4.3 CTL-Model-Checking mit Fairness	62
5 Halbordnungssemantik (Partial Order Semantics)	65
5.1 Partielle und strikte Halbordnung	66
5.2 Logische und vektorielle Zeitstempel	70
6 Petrinetze und Nebenläufigkeit	79
6.1 Petrinetze	79
6.2 Platz/Transitions-Netze	94
6.3 Fairness: 5-Philosophen	105
6.4 Prozesse von Petrinetzen	108
7 Systemverifikation durch Petrinetze	117
7.1 Verifikation beschränkter Netze	117
7.2 Der Überdeckungsgraph	126
7.3 Zustandsraumexplosion bei nebenläufigen Systemen	132

7.4 Inhibitor-Netze, Zählerprogramme und Turing-Mächtigkeit	135
7.5 Strukturelle Eigenschaften	139
8 Anwendungsmodellierung mit Petrinetzen	151
8.1 Workflow-Netze	151
8.2 Das Bankiersproblem: Sichere Zustände	160
8.3 Kantenkonstante Netze	166
8.4 Gefärbte Netze	172
8.5 Das RENEW-Werkzeug	181
8.6 Das Alternierbitprotokoll	185
9 Prozessalgebra	189
9.1 Prozess-Algebra	189
9.2 Parallele und kommunizierende Prozesse	199
9.3 Abbruch und Unterdrücken	202
9.4 Rekursion	205
9.5 Abstraktion	212
9.6 Verifikation des Alternierbitprotokolls	221
10 Parallel Algorithmen	227
10.1 Random-Access-Maschine (RAM)	229
10.2 Parallele Random-Access-Maschine (PRAM)	240
10.3 Paralleles Suchen und optimales Mischen	250
10.4 Paralleles Sortieren	257

1 Endliche Automatenmodelle

Wir betrachten im Folgenden Automaten für endliche und unendliche Wörter.

1.1 Endliche Automaten

Im Folgenden wiederholen wir Aspekte des endlichen Automaten, der in der Vorlesung FGI-1 behandelten wurde.

Definition 1.1 Ein endlicher nichtdeterministischer Automat

$$A = (Q, \Sigma, \delta, Q_0, F)$$

besteht aus den folgenden Komponenten:

- Q ist eine endliche Menge von Zuständen.
- Σ ist das Eingabealphabet.
- $\delta \subseteq (Q \times \Sigma \times Q)$ ist die Übergangsrelation.
- $Q_0 \subseteq Q$ ist die Menge der Startzustände.
- $F \subseteq Q$ ist die Menge der Endzustände.

Beispiel 1.2 Betrachten wir den NFA in Abb. 1.1, der einen Getränkeautomat modelliert. Nach Einwurf einer Euro-Münze je nach Wahl von Tee oder Kaffee wird das entsprechende Getränk ausgegeben. Es ist $Q = \{s_0, \dots, s_5\}$, $Q_0 = F = \{s_0\}$ und

$$\Sigma = \{1€, \text{Tee}, \text{Kaffee}, \text{Tee_kochen}, \text{Kaffee_kochen}, \text{Tee_einfüllen}, \text{Kaffee_einfüllen}\}.$$

Statt $(p, a, q) \in \delta$ wird meist die intuitivere Schreibweise $p \xrightarrow{a} q$ benutzt.

Die Notation $p \xrightarrow{a} q$ wird wie folgt zu $p \xrightarrow{w} q$ auf Wörter $w \in \Sigma^*$ erweitert (Sei $a \in \Sigma$, $w \in \Sigma^*$ und $p, q, r \in Q$):

- $p \xrightarrow{\epsilon} p$ für alle $p \in Q$.
- $p \xrightarrow{wa} r : \iff \exists q \in Q : (p \xrightarrow{w} q) \wedge (q \xrightarrow{a} r)$.

Die akzeptierte Sprache eines NFA A ist definiert als:

$$L(A) := \{w \in \Sigma^* \mid \exists p \in Q_0, q \in F : p \xrightarrow{w} q\}$$

Zwei NFA A_1 und A_2 heißen äquivalent falls $L(A_1) = L(A_2)$ gilt.

Die Familie aller von NFA akzeptierten Sprachen ist die Familie der *Regulären Mengen*. Jeder NFA kann durch Äquivalenzumformungen vervollständigt werden, so daß zu jeder Eingabe ein Folgezustand definiert ist, d.h. $\delta^*(q, w)$ ist für alle $q \in Q$ und alle $w \in \Sigma^*$ definiert.

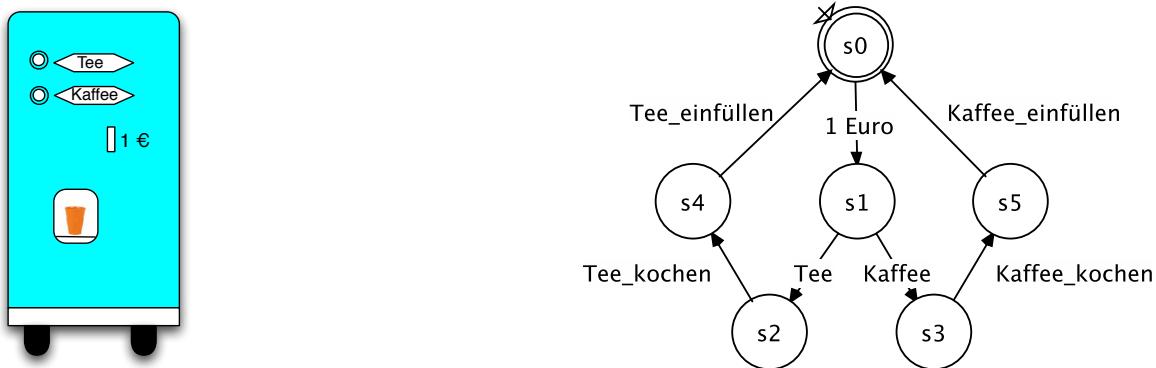


Abbildung 1.1: Getränkeautomat mit abstrakten Transitionsbezeichnern

1.1.1 Deterministische Endliche Automaten

Ein NFA A heißt deterministisch (DFA), wenn er genau einen Startzustand besitzt und die Relation δ funktional in den ersten beiden Argumenten ist, d.h. wenn $(p, a, q_1), (p, a, q_2) \in \delta$, dann gilt $q_1 = q_2$. Die Relation δ kann daher auch dann als Funktion $\delta : Q \times \Sigma \rightarrow Q$ aufgefasst werden, wobei δ i.a. keine totale Funktion ist.

Satz 1.3 *Deterministische Endliche Automaten sind äquivalent zu nichtdeterministischen.*

Beweis: Potenzautomatenkonstruktion. □

1.1.2 Abschlusseigenschaften

Satz 1.4 *Seien L_1 und L_2 reguläre Mengen, dann gilt:*

- *Die Vereinigung $L_1 \cup L_2$ ist eine reguläre Menge.*
- *Der Schnitt $L_1 \cap L_2$ ist eine reguläre Menge.*
- *Das Komplement $\Sigma^* \setminus L_1$ ist eine reguläre Menge.*
- *Der Kleene-Abschluss L_1^* ist eine reguläre Menge.*

1.1.3 Homomorphismen

Seien Σ und Γ zwei Alphabete. Ein Homomorphismus ist eine Abbildung $h : \Sigma^* \rightarrow \Gamma^*$ zwischen den Wortmengen, die die Monoidstruktur erhält, d.h. für alle $u, v \in \Sigma^*$ gilt:

$$h(u \cdot v) = h(u) \cdot h(v)$$

Also ist jeder Homomorphismus $h : \Sigma^* \rightarrow \Gamma^*$ eindeutig festgelegt, wenn wir $h(a)$ für alle $a \in \Sigma$ kennen:

$$h(w) = h(a_1 \cdots a_n) = h(a_1) \cdots h(a_n)$$

Satz 1.5 Für jede reguläre Menge $R \subseteq \Sigma^*$ und jeden Homomorphismus $h : \Sigma^* \rightarrow \Gamma^*$ ist auch das homomorphe Bild $h(R)$ regulär.

$$h(R) := \{h(w) \mid w \in R\}$$

Beweis: Zu der regulären Menge $R \subseteq \Sigma^*$ existiert ein akzeptierender DFA A . Ersetzen wir in A jede Kante $p \xrightarrow{a} q$ durch das Bild $p \xrightarrow{h(a)} q$, so erhalten wir einen verallgemeinerten FA, der $h(R)$ akzeptiert. \square

1.1.4 Reguläre Ausdrücke

Definition 1.6 Die regulären Ausdrücke über einem endlichen Alphabet Σ sind definiert durch:

1. \emptyset ist ein regulärer Ausdruck, der die (leere) Menge $M_\emptyset := \emptyset$ beschreibt.
2. Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck, der die Menge $M_a := \{a\}$ beschreibt.
3. Sind A und B reguläre Ausdrücke, welche die Mengen M_A und M_B beschreiben, dann sind induktiv folgende reguläre Ausdrücke definiert:
 - $(A + B)$ beschreibt die Menge $M_A \cup M_B$,
 - $(A \cdot B)$ beschreibt die Menge $M_A \cdot M_B$,
 - A^* beschreibt die Menge M_A^* .
4. Nur die mit 1., 2. und 3. erzeugten Ausdrücke sind reguläre Ausdrücke.

Oftmals nimmt man noch A^+ als Ausdruck hinzu. Dieser beschreibt die Menge M_A^+ . Der Ausdruck A^+ kann auch als Makro für AA^* aufgefasst werden.

Die durch reguläre Ausdrücke A beschriebenen Mengen M_A sind genau die *regulären Mengen* oder Sprachen über Σ .

Satz 1.7 Die Menge der durch endliche Automaten $A = (Q, \Sigma, \delta, Q_0, F)$ akzeptierten Sprachen ist genau die Menge der durch reguläre Ausdrücke über Σ beschriebenen Sprachen.

Beweis: Der Beweis erfolgt in einer Richtung dadurch, dass gemäß der induktiven Definition von regulären Sprachen nichtdeterministische endliche Automaten (NFAs) gekoppelt werden und dadurch der Abschluss unter Vereinigung, Produkt und * -Hülle gezeigt wird. In der anderen Richtung besteht der Beweis in der berühmten Konstruktion von Kleene. Beide Verfahren sind in den Unterlagen zur Vorlesung FGI-1 nachzulesen. \square

1.1.5 Produktautomaten

Reguläre Mengen sind abgeschlossen unter Durchschnitt. Dies wird oft (so auch in FGI-1) darauf zurückgeführt, dass sie unter Vereinigung und Komplement abgeschlossen sind. Die Produktautomaten-Konstruktion erlaubt einen direkten Beweis, was im Rahmen des Model-Checking von besonderer Bedeutung ist.

Satz 1.8 Gegeben seien die Automaten $A_i = (Q_i, \Sigma, \delta_i, Q_i^0, F_i)$ für $i = 1, 2$.

Aus A_1 und A_2 kann effektiv der Automat $A_3 = (Q_3, \Sigma, \delta_3, Q_3^0, F_3)$ konstruiert werden, der den Durchschnitt der beiden Sprachen akzeptiert: $L(A_3) = L(A_1) \cap L(A_2)$

Beweis: Die Zustände von A_3 ergeben sich als kartesisches Produkt. Wir definieren $Q_3 = Q_1 \times Q_2$, $Q_3^0 := Q_1^0 \times Q_2^0$ und $F_3 := F_1 \times F_2$. Die Übergänge werden synchronisiert:

$$(s, r) \xrightarrow[3]{a} (s', r') \iff (s \xrightarrow[1]{a} s' \wedge r \xrightarrow[2]{a} r')$$

Es sei nun $w \in L(A_1) \cap L(A_2)$ und $w = a_1 a_2 \cdots a_n$.

Dann gilt $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s_n$ mit $s_0 \in Q_1^0$, $s_n \in F_1$ in A_1 und $r_0 \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \cdots r_{n-1} \xrightarrow{a_n} r_n$ mit $r_0 \in S_2^0$, $r_n \in F_2$ in A_2 .

Dann ist nach Konstruktion auch

$$(s_0, r_0) \xrightarrow[3]{a_1} (s_1, r_1) \xrightarrow[3]{a_2} (s_2, r_2) \cdots (s_{n-1}, r_{n-1}) \xrightarrow[3]{a_n} (s_n, r_n)$$

mit $(s_0, r_0) \in Q_1^0 \times Q_2^0$ und $(s_n, r_n) \in F_1 \times F_2$ und somit ist $w \in L(A_3)$.

Der umgekehrte Schluss erfolgt entsprechend. \square

1.1.6 Entscheidungsprozeduren

Folgende Fragen sind (u.a. auch im Kontext des Model-Checkings) von Interesse

- Das *Leerheitsproblem* für NFA ist die folgende Frage:
„Gegeben ein NFA A . Gilt $L(A) = \emptyset$?“
- Das *Universalitätsproblem* für NFA ist die folgende Frage:
„Gegeben ein NFA A . Gilt $L(A) = \Sigma^*$?“
- Das *Äquivalenzproblem* für NFA ist die folgende Frage:
„Gegeben zwei NFA A_1 und A_2 . Gilt $L(A_1) = L(A_2)$?“

Lemma 1.9 Das Leerheitsproblem für NFA ist entscheidbar.

Das Universalitätsproblem für NFA ist entscheidbar.

Das Äquivalenzproblem für NFA ist entscheidbar.

Beweis: Als Übung. \square

1.2 Büchi-Automaten

Für die Beschreibung und Analyse von reaktiven Systemen werden meist unendliche Aktionsfolgen benutzt. Auch hier sind in Bezug auf Endzustände akzeptable Aktionsfolgen definiert und zwar dadurch, dass die zugehörige Zustandsfolge die Endzustandsmenge F unendlich oft treffen muss.

Ein ω -Wort $\sigma \in \Sigma^\omega$ wird als $\sigma = a_0a_1a_2\cdots$ dargestellt, wobei $a_i \in \Sigma$ für alle i .

Die Menge aller ω -Wörter über dem Alphabet Σ wird mit Σ^ω bezeichnet.

$\Sigma^\infty := \Sigma^* \cup \Sigma^\omega$ bezeichnet die Menge der endlichen oder unendlichen Folgen von Zeichen aus Σ .

Für ein ω -Wort $\sigma = a_0a_1a_2\cdots$ notieren wir die Menge der unendlich häufig vorkommenden Elemente wie folgt:

$$\text{infinite}(\sigma) := \{a \in \Sigma \mid a \text{ kommt in } \sigma \text{ unendlich oft vor}\}$$

Eine Menge $L \subseteq \Sigma^\omega$ heißt ω -Sprache.

Wir wollen ω -Sprachen jetzt mit Hilfe von Automaten definieren. Ein (nichtdeterministischer) Büchi-Automat $A = (Q, \Sigma, \delta, Q_0, F)$ besteht aus den gleichen Komponenten wie ein NFA. Die Akzeptanzbedingung muss aber anders formuliert werden, da ja das Wort nie zu Ende ist.

Ein Pfad ist eine Folge $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots$, die in einem Startzustand beginnt ($s_0 \in Q_0$).

Definition 1.10 Das ω -Wort $\sigma = a_0a_1a_2\cdots \in \Sigma^\omega$ wird von dem Büchi-Automat $A = (Q, \Sigma, \delta, Q_0, F)$ akzeptiert, wenn ein Pfad $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots$ mit $s_0 \in Q_0$ existiert, der mindesten einen Endzustand unendlich oft durchläuft, d.h. es gilt

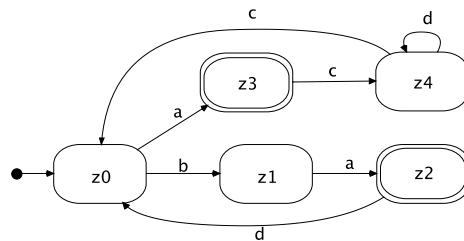
$$\text{infinite}(s_0s_1s_2\cdots) \cap F \neq \emptyset$$

$$L^\omega(A) = \{w \in A^\omega \mid \sigma \text{ wird akzeptiert}\}$$

Die Familie der von Büchi-Automaten akzeptierten ω -Sprachen wird als die Familie der ω -regulären Sprachen bezeichnet.

Beispiel 1.11 Der folgende Büchi-Automat akzeptiert die Sprache:

$$L^\omega(A) = (\{ac\}\{d\}^*\{c\} \cup \{bad\})^\omega$$



1.2.1 Deterministische Büchi-Automaten

Ein Büchi-Automat A heißt *deterministisch*, wenn A als NFA betrachtet deterministisch ist.

Folgendes Lemma zeigt die Grenzen deterministischer Büchi-Automaten.

Lemma 1.12 *Es gibt keinen deterministischen Büchi-Automaten, der die Sprache $L = \{a, b\}^* \{b\}^\omega$ akzeptiert.*

Beweis: Angenommen es gäbe doch einen deterministischen Büchi-Automaten A , der die Sprache $L = \{a, b\}^* \{b\}^\omega$ akzeptiert.

Dann würde $\sigma_0 = b^\omega$ akzeptiert. Dann gäbe es einen Präfix u_0 von σ_0 (d.h. $u_0 \in \{b\}^*$), so dass $\delta^*(s_0, u_0) \in F$ gilt.

Betrachten wir nun $\sigma_1 = u_0ab^\omega$. Das Wort σ_1 wird auch akzeptiert.

Da unendlich oft ein Endzustand eingenommen wird, gäbe es dann auch einen Präfix von σ_1 der Form u_0au_1 , so dass $\delta^*(s_0, u_0au_1) \in F$ gilt.

Setzen wir das Argument fort, so erhalten wir endliche Wörter $u_i \in \{b\}^*$, so dass $\delta^*(s_0, u_0au_1a \cdots au_n) \in F$ gilt.

Da wir nur endlich viele Zustände haben, existieren zwei i, j mit $i < j$, so dass die erreichten Zustände gleich sind:

$$\delta^*(s_0, u_0au_1a \cdots au_i) = \delta^*(s_0, u_0au_1a \cdots au_j)$$

Also bildet das Wort $(au_{i+1}a \cdots au_j)$ einen Zyklus. Da $i < j$, ist das Wort nicht-leer. Dann würde auch das folgende ω -Wort akzeptiert:

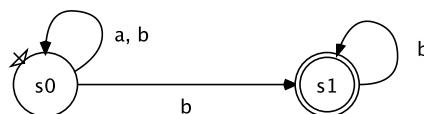
$$(u_0au_1a \cdots au_i) \cdot (au_{i+1}a \cdots au_j)^\omega$$

Dieses Wort hat allerdings unendlich viele Vorkommnisse von a , liegt also nicht mehr in $L = \{a, b\}^* \{b\}^\omega$. Widerspruch. \square

Aus dem Lemma folgt sofort folgende Trennungseigenschaft.

Satz 1.13 *Nichtdeterministische Büchi-Automaten sind echt mächtiger als deterministische.*

Beweis: Die Sprache $L = \{a, b\}^* \{b\}^\omega$ kann zwar nicht von einem deterministischen Büchi-Automaten akzeptiert werden, sie wird aber von dem folgenden nichtdeterministischen Büchi-Automaten akzeptiert.



\square

1.2.2 Abschlussoperationen auf ω -Sprachen

Wir definieren Operationen auf ω -Sprachen.

Definition 1.14 Für ein (endliches) Wort $w = a_1a_2 \dots a_n \in \Sigma^*$ und ein ω -Wort $u = b_1b_2 \dots \in \Sigma^\omega$ sei $w \cdot u := a_1a_2 \dots a_n b_1b_2 \dots \in \Sigma^\omega$ die Konkatenation dieser Wörter.

Entsprechend definiert man für Mengen $W \subseteq \Sigma^*$ und $U \subseteq \Sigma^\omega$ die Konkatenation $W \cdot U \subseteq \Sigma^\omega$ komponentenweise:

$$W \cdot U := \{w \cdot u \mid w \in W, u \in U\}$$

Ferner sei für $W \subseteq \Sigma^*$ der ω -Abschluss W^ω folgendermaßen definiert:

$$W^\omega := \{w_1 \cdot w_2 \cdot w_3 \dots \mid w_i \in W \setminus \{\epsilon\}, i \geq 1\}$$

Vereinigung, Konkatenation und ω -Abschluss sind für ω -reguläre Mengen Abschlussoperatoren.

Lemma 1.15 Wenn die Mengen $U, V \subseteq \Sigma^\omega$ zwei ω -reguläre Mengen sind und die Menge $W \subseteq \Sigma^*$ regulär ist, dann gilt:

- Die Vereinigung $U \cup V \subseteq \Sigma^\omega$ ist eine ω -reguläre Menge.
- Der ω -Abschluss $(W \setminus \{\epsilon\})^\omega$ ist eine ω -reguläre Menge.
- Die gemischte Konkatenation $W \cdot U \subseteq \Sigma^\omega$ ist eine ω -reguläre Menge.

Beweis: Als Übung. □

Auch Komplementbildung ist für ω -reguläre Mengen eine Abschlussoperation.

Lemma 1.16 Wenn die Menge $U \subseteq \Sigma^\omega$ eine ω -reguläre Menge ist, dann ist das Komplement $\Sigma^\omega \setminus U$ eine ω -reguläre Menge.

Beweis: Zu aufwendig. □

Der Konstruktionsaufwand ist aber mehr als exponentiell in der Größe des Ausgangsaufwands!

1.2.3 Büchi-Automaten und ω -reguläre Ausdrücke

Wir definieren nun das Analogon der regulären Ausdrücke für unendliche Wörter.

Definition 1.17 Seien $A_1, \dots, A_n, B_1, \dots, B_n$ reguläre Ausdrücke über Σ , wobei kein M_{B_i} das leere Wort ϵ enthält.

Ein ω -regulärer Ausdruck über dem Alphabet Σ ist ein Ausdruck der Form:

$$G = A_1 \cdot B_1^\omega + \dots + A_n \cdot B_n^\omega$$

Dieser Ausdruck beschreibt die ω -reguläre Sprache

$$M_G := M_{A_1} \cdot M_{B_1}^\omega \cup \dots \cup M_{A_n} \cdot M_{B_n}^\omega.$$

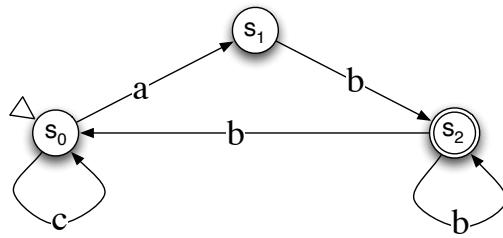


Abbildung 1.2: Büchi-Automat A mit ω -Sprache $L^\omega(A) = M_G$ mit $G = c^*ab(b^+ + bc^*ab)^\omega$

Der Büchi-Automat A in Abbildung 1.2 akzeptiert die ω -Sprache $L^\omega(A) = M_G$ mit

$$G = c^*ab(b^+ + bc^*ab)^\omega.$$

Der nächste Satz erweitert die Äquivalenz von regulären Ausdrücken und endlichen Automaten auf eine Beziehung von Büchi-Automaten und ω -regulären Ausdrücken.

Satz 1.18 Die Familie der ω -regulären Sprachen ist genau die Familie der durch die ω -regulären Ausdrücke beschriebenen Sprachen.

Beweis: Der Beweis für ω -Sprachen ist dem für reguläre Sprachen ähnlich.

Sei der ω -reguläre Ausdruck gegeben. Wir wissen bereits, wie man zu jedem regulären Ausdruck einen äquivalenten NFA konstruieren kann.

Nach Lemma 1.15 existieren Konstruktionen für Büchi-Automaten, die dann $U \cup V$, W^ω und $W \cdot U$ akzeptieren. So erzeugt man induktiv über den Aufbau des Ausdruck einen akzeptierenden Büchi-Automaten.

Sei umgekehrt ein Büchi-Automat A gegeben. Mit der Kleene-Konstruktion können wir die Menge $R_{p,q}$ aller Wörter bestimmen, die im Zustand p starten und in q enden. Diese Mengen sind jeweils regulär.

Ein ω -Wort wird genau dann akzeptiert, wenn es einen Endzustand q gibt, der auf einem Kreis liegt und der Kreis ist von einem Startzustand p aus erreichbar.

Das ist gleichbedeutend damit, dass q erreichbar ist und es einen Kreis von q nach q gibt. Alle diese Wörter haben dann folgende Darstellung:

$$L^\omega(A) = \sum_{p \in Q_0, q \in F} R_{pq} \cdot R_{qq}^\omega$$

Für Details siehe [BK08].

□

1.2.4 Produktautomat: Abschluss bzgl. Mengenschnitt

Auch die ω -reguläre Mengen sind bzgl. Durchschnitt abgeschlossen. Die Produktautomaten-Konstruktion für NFA ist aber hier nicht ausreichend. Es gilt die folgende Inklusion:

Lemma 1.19 Sei A_3 der Produktautomat der NFA A_1 und A_2 . Dann gilt

$$L^\omega(A_3) \subseteq (L^\omega(A_1) \cap L^\omega(A_2))$$

Beweis: Gegeben seien die Automaten $A_i = (Q_i, \Sigma, \delta_i, Q_i^0, F_i)$ für $i = 1, 2$. Sei A_3 der Produktautomat von A_1 und A_2 .

Ist dann $\sigma = a_1 a_2 \dots \in L^\omega(A_3)$, also

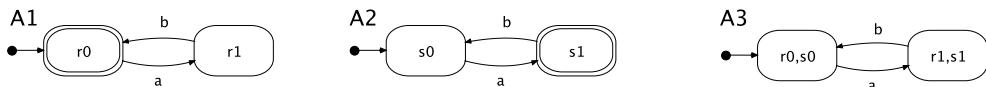
$$(s_0, r_0) \xrightarrow{a_1} (s_1, r_1) \xrightarrow{a_2} (s_2, r_2) \xrightarrow{a_3} \dots$$

mit $(s_0, r_0) \in Q_1^0 \times Q_2^0$ und $(s_i, r_i) \in F_1 \times F_2$ für unendlich viele $i \geq 1$, dann gilt entsprechendes auch für die beiden Komponenten und es ist $w \in L^\omega(A_1) \cap L^\omega(A_2)$. Also gilt stets $L^\omega(A_3) \subseteq (L^\omega(A_1) \cap L^\omega(A_2))$. \square

Der Umkehrschluss ist jedoch so nicht möglich, da die beiden Automaten ja immer zu unterschiedlichen Zeitpunkten in die Endzustände gelangen können, wie das folgende Beispiel zeigt.

Beispiel 1.20 Betrachte die beiden folgenden Automaten A_1 und A_2 . Es gilt: $L^\omega(A_1) = L^\omega(A_2) = (ab)^\omega$. Also gilt: $L^\omega(A_1) \cap L^\omega(A_2) = (ab)^\omega$.

Das Produkt A_3 enthält nun gar keine erreichbaren Endzustände. Also gilt $L^\omega(A_3) = \emptyset$.



Es ist aber möglich, die Produktautomatenkonstruktion so zu erweitern, dass der Durchschnitt akzeptiert wird:

Satz 1.21 Aus zwei Büchi-Automaten $A_i = (Q_i, \Sigma, \delta_i, Q_i^0, F_i)$, $i = 1, 2$ kann effektiv ein Büchi-Automat $A_4 = (Q_4, \Sigma, \delta_4, Q_4^0, F_4)$ konstruiert werden, der den Durchschnitt der akzeptierten ω -Sprachen akzeptiert:

$$L^\omega(A_4) = L^\omega(A_1) \cap L^\omega(A_2)$$

Beweis: Wir erweitern A_3 , indem wir die Zustände (s, r) um eine dritte Komponente i erweitern, die erzwingt, dass beide Automaten jeweils unendlich oft einen Endzustand besuchen. Die dritte Komponente gibt den Automaten an, dessen Endzustand gerade erwartet wird:

- $(s, r, 1)$ bedeutet dabei, dass A_4 darauf wartet, dass wir einen Endzustand aus A_1 einnehmen. Wird dieser durchlaufen, so wechselt die dritte Komponenten auf $i = 2$.
- $(s, r, 2)$ bedeutet entsprechend, dass wir auf einen Endzustand aus A_2 warten.

Ist $w \in L^\omega(A_1)$ und $w \in L^\omega(A_2)$, dann muss immer wieder (in jeder Komponente) ein Endzustand eingenommen werden.

Wenn wir als Endzustände nur solche mit $i = 1$ zulassen, dann erzwingen wir, dass Zustände (s, r, i) mit $s \in F_1$ und $r \in F_2$ unendlich oft im Wechsel von $i = 1$ und $i = 2$ eingenommen werden. Also gilt $L^\omega(A_1) \cap L^\omega(A_2) \subseteq L^\omega(A_4)$.

Formal definieren wir $A_4 = (Q_4, \Sigma, \delta_4, Q_4^0, F_4)$ durch

- $Q_4 = \{(s, r, i) \mid (s, r) \in Q_3, i \in \{1, 2\}\}$
- $(s, r, i) \xrightarrow[4]{a} (s', r', i') \iff (s, r) \xrightarrow[3]{a} (s', r') \wedge i' := \begin{cases} 2, & \text{falls } i = 1 \wedge s \in F_1 \\ 1, & \text{falls } i = 2 \wedge r \in F_2 \\ i, & \text{sonst} \end{cases}$
- $Q_4^0 := \{(s, r, 1) \mid (s, r) \in Q_3^0 = Q_1^0 \times Q_2^0\}$
- $F_4 := \{(s, r, 1) \mid s \in F_1\}$

Weiterhin gilt auch (wie schon bei A_3), dass jedes $w \in L^\omega(A_4)$ sowohl in A_1 als auch in A_2 akzeptiert wird: $L^\omega(A_4) \subseteq L^\omega(A_1) \cap L^\omega(A_2)$.

Insgesamt gilt also $L^\omega(A_4) = L^\omega(A_1) \cap L^\omega(A_2)$. □

1.2.5 Verallgemeinerte Büchi-Automaten

Ein *verallgemeinerter Büchi-Automat* besitzt statt einer Endzustandsmenge gleich mehrere.

Definition 1.22 Ein verallgemeinerter Büchi-Automat ist das Tupel $A = (Q, \Sigma, \delta, Q_0, \mathcal{F})$, wobei $\mathcal{F} = \{F_1, \dots, F_k\}$ eine Menge von Endzustandsmengen ist.

Das ω -Wort $\sigma = a_0 a_1 a_2 \dots \in \Sigma^\omega$ wird von einem verallgemeinerten Büchi-Automat A akzeptiert, wenn ein Pfad $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \dots$ mit $s_0 \in Q_0$ existiert, der in jeder Menge $F \in \mathcal{F}$ mindesten einen Endzustand unendlich oft durchläuft, d.h. es gilt

$$\forall F \in \mathcal{F} : \text{infinite}(s_0 s_1 s_2 \dots) \cap F \neq \emptyset$$

$L^\omega(A) = \{w \in A^\omega \mid \sigma \text{ wird akzeptiert}\}$ ist die akzeptierte ω -Sprache von A .

Satz 1.23 Zu jedem verallgemeinerten Büchi-Automaten A existiert ein Büchi-Automat A' , der die gleiche Sprache akzeptiert: $L^\omega(A) = L^\omega(A')$.

Beweis: Sei $\mathcal{F} = \{F_1, \dots, F_n\}$.

(Skizze) Wir definieren den Büchi-Automaten A' folgendermaßen: Man erzeuge n Kopien von A und starte in der 1. Kopie. Wenn A in der i . Kopie einen Endzustand aus F_i erreicht, wird in den gleichen Zustand der $(i+1)$. Kopie gewechselt. Als Endzustandsmenge wähle man F_1 .

Akzeptiert A' ein ω -Wort σ , dann müssen dabei alle n Kopien durchlaufen worden sein und in jeder Kopie muss ein Endzustand erreicht worden sein. Also hätte auch A das Wort σ akzeptiert.

Akzeptiert umgekehrt A das ω -Wort σ , dann wird mindestens ein Zustand s_i aus jedem F_i unendlich oft durchlaufen. In A' wird bei s_i in die $(i+1)$. Kopie gewechselt, bei s_n wieder zurück in die 1. Kopie. Also hätte auch A das Wort σ akzeptiert. \square

1.2.6 Entscheidungsprozeduren

Folgende Fragen sind (u.a. auch im Kontext des Model-Checkings) von Interesse:

- Das *Leerheitsproblem* für Büchi-Automaten ist die folgende Frage:
„Gegeben ein Büchi-Automat A . Gilt $L^\omega(A) = \emptyset$?“
- Das *Universalitätsproblem* für Büchi-Automaten ist die folgende Frage:
„Gegeben ein Büchi-Automat A . Gilt $L^\omega(A) = \Sigma^\omega$?“
- Das *Äquivalenzproblem* für Büchi-Automaten ist die folgende Frage:
„Gegeben zwei Büchi-Automaten A_1 und A_2 . Gilt $L^\omega(A_1) = L^\omega(A_2)$?“

Lemma 1.24 *Das Leerheitsproblem für Büchi-Automaten ist entscheidbar.*

Das Universalitätsproblem für Büchi-Automaten ist entscheidbar.

Das Äquivalenzproblem für Büchi-Automaten ist entscheidbar.

Beweis: Als Übung. \square

2 Transitionssysteme und Kripke-Strukturen

Das ursprüngliche¹ Modell des endlichen Automaten wird für die Beschreibung und Analyse von Systemverhalten in der Form von *Transitionssystemen* fortgeführt. In den verschiedenen Anwendungen sind unterschiedliche Varianten von Transitionssystemen gebräuchlich, die aber wesentliche Komponenten gemeinsam haben, nämlich Zustände und Übergänge (Transitionen) zwischen Zuständen. In diesem Kapitel werden die grundlegenden Definitionen von Transitionssystemen vorgestellt, sowie einige ihrer wichtigsten Eigenschaften und Methoden, wie Bisimulation und Synchronisation.

2.1 Transitionssysteme

Transitionssysteme bestehen aus Zuständen, darunter Anfangs- und Endzustände, sowie einer Transitionsrelation. In manchen Darstellungen der Literatur ist nur ein einziger Anfangszustand vorgesehen oder es wird auf Endzustände verzichtet.

Beispielsweise generiert eine Turingmaschine ein Transitionssystem: Die Konfigurationen der TM sind die Zustände und die Übergänge die Aktionen. Endliche Automaten sind Beispiele endlicher Transitionssysteme. Die Semantik von prozessalgebraischen Ausdrücken (Kapitel 9) und von Petrinetzen (Kapitel 6) wird ebenfalls durch Transitionssysteme beschrieben. Beim Model-Checking (Abschnitte 2.5 und 3.1) kommen Transitionssysteme zum Einsatz, bei denen die Zustände mit gültigen Aussagen etikettiert werden.

Definition 2.1 Ein Transitionssystem $TS = (S, A, tr, S^0, S^F)$ besteht aus

- einer Menge S von Zuständen,
- einer Menge A von Aktionen (auch: Transitionen),
- einer Transitionsrelation $tr \subseteq S \times A \times S$,
- einer Menge $S^0 \subseteq S$ von Anfangszuständen und
- einer Menge $S^F \subseteq S$ von Endzuständen,

TS heißt endlich, falls S und A endlich sind.

Existiert nur ein Anfangszustand, dann schreibt man auch im Tupel $TS = (S, A, tr, S^0, S^F)$ nur s_0 statt $\{s_0\}$. Wird S^F in dem Tupel weggelassen, dann sind alle Zustände Endzustände, d.h. $S^F = S$.

¹ „ursprünglich“ sowohl im historischen Sinn als auch im Rahmen der Veranstaltungen FGI-1 und FGI-2

Ein Transitionssystem ist somit also ein kantenbeschrifteter Graph (evtl. unendlich groß), bei dem einige Knoten als Start- bzw. Endzustände gekennzeichnet sind.

Statt $(s, a, s') \in tr$ wird meist die intuitivere Schreibweise $s \xrightarrow{a} s'$ benutzt. Diese erweitert sich (wie zuvor für Automaten) auch auf A^* .

Ein *unendlicher Pfad* ist eine in einem Startzustand beginnende (d.h. $s_0 \in S^0$) Folge der Form:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \dots$$

Das dynamische Verhalten („die Semantik“) eines Transitionssystems wird durch die erreichbaren Zustände und durch seine (endlichen oder unendlichen) Transitions- bzw. Aktionsfolgen beschrieben.

- $R(TS, S_1) := \{s \in S \mid \exists w \in A^*, s_1 \in S_1 : s_1 \xrightarrow{w} s\}$ ist die *Menge der von S_1 aus in TS erreichbaren Zustände* und
- $R(TS) := R(TS, S^0)$ ist die *Erreichbarkeitsmenge von TS* .

Entsprechend definiert man für Aktionsfolgen:

- $AS(TS, S_1) := \{w \in A^* \mid \exists s_1 \in S_1, s'_1 \in S : s_1 \xrightarrow{w} s'_1\}$ ist die *Menge der von S_1 aus in TS möglichen Aktionsfolgen*.
- $AS(TS) := AS(TS, S^0)$ ist die *Aktionsfolgenmenge von TS* .
- $FS(TS) := \{w \in A^* \mid \exists s_1 \in S^0, s'_1 \in S^F : s_1 \xrightarrow{w} s'_1\}$ ist die *terminale Aktionsfolgenmenge von TS* .

Diese Menge heißt oft auch *akzeptable Aktionsfolgenmenge* oder *akzeptierte Sprache* und wird als $L(TS)$ bezeichnet.

Lemma 2.2 Sei TS ein Transitionssystem, dann gilt $FS(TS) \subseteq AS(TS)$.

Es gilt $FS(TS) = AS(TS)$, falls $S^F = S$.

Beweis: Als Übung. □

Analog definieren wir für unendliche Zustandsfolgen:

- $SS(TS, S_1) := \{s_0s_1s_2 \dots \mid TS \text{ besitzt einen Pfad } s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \text{ mit } s_0 \in S_1\}$ ist die *Menge aller Zustandsfolgen von TS* .
- $SS(TS) := SS(TS, S^0)$.

2.2 Bisimilarität bei reaktiven Systemen

Bei Transitionssystemen unterscheiden wir zwei Formen der Äquivalenz: Folgenäquivalenz und Akzeptanzäquivalenz. Akzeptanzäquivalenz entspricht der bei NFA gebräuchlichen Äquivalenz.

Definition 2.3 Zwei Transitionssysteme TS_1 und TS_2 heißen folgenäquivalent, falls sie die gleiche Menge von Aktionsfolgen haben:

$$TS_1 \sim_{AS} TS_2 : \Leftrightarrow AS(TS_1) = AS(TS_2)$$

Sie heißen terminal folgenäquivalent oder akzeptanzäquivalent, falls gilt:

$$TS_1 \sim_L TS_2 : \Leftrightarrow L(TS_1) = L(TS_2)$$

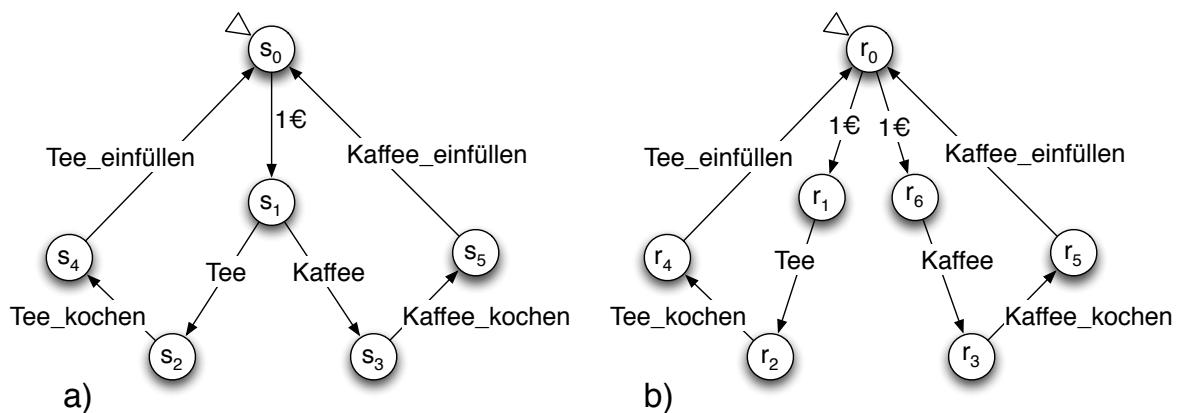


Abbildung 2.1: Zwei folgenäquivalente Transitionssysteme

Abbildung 2.1 zeigt zwei folgenäquivalente Transitionssysteme. Das Beispiel zeigt auch, dass in manchen Fällen die Folgenäquivalenz kein adäquates Mittel ist, um gleiches Systemverhalten zu definieren: Im linken System sind nach dem Münzeinwurf beide Wahlmöglichkeiten geben, was in dem rechten System nicht der Fall ist.

Eine bessere Formalisierung für gleiches Systemverhalten ist durch den Begriff der *Bisimulation* möglich.

Wir betrachten für die folgende Definition zwei Transitionssysteme mit der gleichen Aktionsmenge A . Dies ist keine echte Einschränkung, da dies durch Obermengenbildung immer zu erreichen ist.

Definition 2.4 Gegeben seien zwei Transitionssysteme $TS_i = (S_i, A, tr_i, S_i^0, S_i^F)$ für $i \in \{1, 2\}$.

Eine (aktionsbasierte) Bisimulation für (TS_1, TS_2) ist eine binäre Relation $\mathcal{B} \subseteq S_1 \times S_2$, für die folgendes gilt:

$$a) \forall s_0 \in S_1^0 : \exists r_0 \in S_2^0 : (s_0, r_0) \in \mathcal{B}$$

$$\forall r_0 \in S_2^0 : \exists s_0 \in S_1^0 : (s_0, r_0) \in \mathcal{B}$$

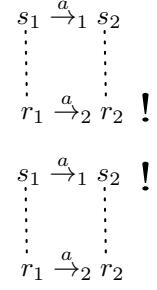
b) Für alle $(s_1, r_1) \in \mathcal{B}$ gilt:

$$s_1 \xrightarrow[1]{a} s_2 \Rightarrow \exists r_2 \in S_2 : r_1 \xrightarrow[2]{a} r_2 \wedge (s_2, r_2) \in \mathcal{B}$$

$$r_1 \xrightarrow[2]{a} r_2 \Rightarrow \exists s_2 \in S_1 : s_1 \xrightarrow[1]{a} s_2 \wedge (s_2, r_2) \in \mathcal{B}$$

$$c) \forall (s, r) \in \mathcal{B} : s \in S_1^F \Leftrightarrow r \in S_2^F$$

Zustände mit $(s, r) \in \mathcal{B}$ heißen bisimilar (in Zeichen $s \leftrightarrow r$).



TS_1 und TS_2 heißen bisimilar (in Zeichen $TS_1 \leftrightarrow TS_2$), falls eine solche Bisimulations-Relation \mathcal{B} existiert.

Die Intention der Bedingungen a) und c) in Definition 2.4 sind offensichtlich: durch sie gibt es zu jedem Anfangszustand einen bisimularen im anderen Transitionssystem. Ein Endzustand hat nur ebensolche bisimulare Partner. Die Bedingung b) wird durch die daneben stehende Graphik illustriert. Die obere dieser Graphiken entspricht dem ersten Teil der Bedingung b). Das Ausrufezeichen markiert das postulierte Element r_2 , während die anderen Elemente s_1, s_2 und r_1 zur Voraussetzung der Bedingung gehören. Die durch die Relation \mathcal{B} verbundenen Paare $s_1 \leftrightarrow r_1$ und $s_2 \leftrightarrow r_2$ sind durch die unterbrochenen Linien gekennzeichnet. Entsprechendes gilt für den zweiten Teil der Bedingung b) und die untere Graphik.

Beispiel 2.5 Wir zeigen nun, dass die (intuitiv) nicht verhaltensäquivalenten Transitionssysteme aus Abbildung 2.1 tatsächlich nicht (formal) bisimilar sind. Wären sie bisimilar, dann müssten wegen der Bedingung a) die Anfangszustände in der Relation stehen: $s_0 \leftrightarrow r_0$. Aus der ersten Bedingung b) folgt dann, dass mit $a = 1\epsilon$ auch $s_1 \leftrightarrow r_1$ oder $s_1 \leftrightarrow r_6$ gelten muss. Im ersten Fall gibt es dann zwar (wieder nach Bedingung b) zu $s_1 \xrightarrow{\text{Tee}} s_2$ die Transition $r_1 \xrightarrow{\text{Tee}} r_2$, nicht aber zu $s_1 \xrightarrow{\text{Kaffee}} s_3$ eine Transition $r_1 \xrightarrow{\text{Kaffee}} r$ für irgend ein r . Da der zweite Fall in analoger Weise zum Widerspruch führt, können die Transitionssysteme nicht bisimilar sein.

Im Gegensatz dazu sind die Transitionssysteme aus Abbildung 2.2 bisimilar. Die Bisimulations-Relation ist $\mathcal{B} = \leftrightarrow = \{(s_i, r_i) \mid i = 0, \dots, 5\} \cup \{(s_1, r_6)\}$.

Lemma 2.6 Seien TS_1 und TS_2 zwei beliebige Transitionssysteme mit $TS_1 \leftrightarrow TS_2$, dann gilt f,r alle $w \in A^*$:

$$(s_0 \xrightarrow[1]{w} s_1 \wedge s_0 \leftrightarrow r_0) \Rightarrow (\exists r_1 : r_0 \xrightarrow[2]{w} r_1 \wedge s_1 \leftrightarrow r_1)$$

Beweis: Wir beweisen dies durch Induktion über die Wortlänge $|w|$.

- Induktionsanfang $w = \epsilon$: In diesem Fall gilt $s_1 = s_0$ und r_1 kann als r_0 gewählt werden.
- Induktionsschritt $w = va$ mit $v \in A^*, a \in A$:
Sei also $s_0 \xrightarrow[1]{va} s_2$ und $s_0 \leftrightarrow r_0$. Dann gibt es ein $s_1 \in S_1$ mit $s_0 \xrightarrow[1]{v} s_1 \xrightarrow[1]{a} s_2$.

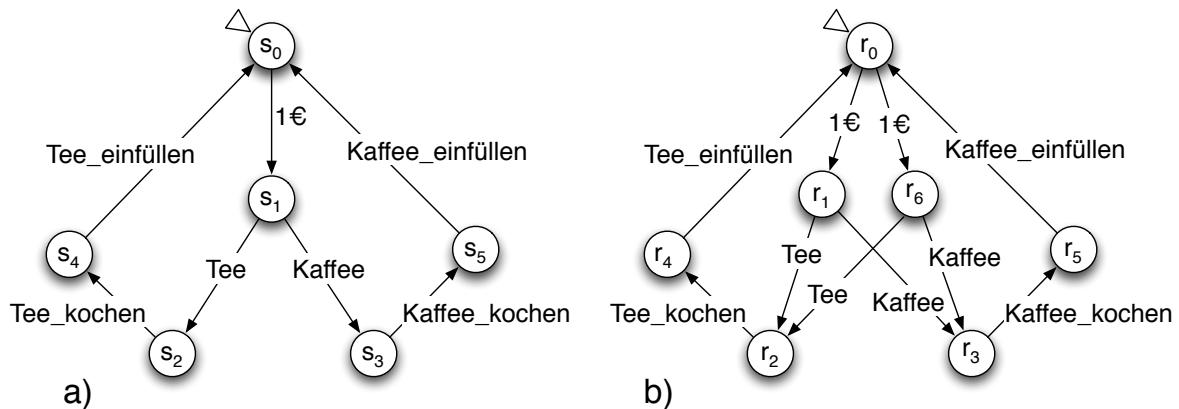


Abbildung 2.2: Zwei bisimilare Transitionssysteme

Nach Induktionsvoraussetzung gibt es einen Zustand r_1 mit $r_0 \xrightarrow{\frac{v}{2}} r_1$ und $s_1 \xleftrightarrow{v} r_1$.

Aufgrund der Bisimilarität gibt es r_2 mit $r_1 \xrightarrow{\frac{a}{2}} r_2$ und $s_2 \xleftrightarrow{a} r_2$. Also gilt $r_0 \xrightarrow{\frac{va}{2}} r_2$ und $s_2 \xleftrightarrow{va} r_2$.

$$\begin{array}{c} s_0 \xrightarrow{v} s_1 \xrightarrow{a} s_2 \\ \vdots \quad \vdots \quad \vdots \\ r_0 \xrightarrow{v} r_1 \xrightarrow{a} r_2 ! \end{array}$$

Damit ist die Aussage bewiesen. □

Satz 2.7 Seien TS_1 und TS_2 zwei beliebige bisimilare Transitionssysteme, dann hat jeder erreichbare Zustand des einen Systems einen bisimilaren Partner im anderen:

$$\begin{aligned} \forall s \in R(TS_1) : \exists r \in R(TS_2) : s \xleftrightarrow{} r \\ \forall r \in R(TS_2) : \exists s \in R(TS_1) : s \xleftrightarrow{} r \end{aligned}$$

Beweis: Wegen der Symmetrie der Aussage in TS_1 und TS_2 reicht es aus, nur eine Hälfte zu zeigen: Wir zeigen $\forall s \in R(TS_1) : \exists r \in R(TS_2) : s \xleftrightarrow{} r$.

Wenn s ein erreichbarer Zustand ist, dann existiert ein Wort w , mit dem der Zustand s von einem Startzustand $s_0 \in S_1^0$ erreicht werden kann: $s_0 \xrightarrow{w} s$.

Nach Def. 2.4 a) existiert zu s_0 ein bisimilärer Partner $r_0 \in S_2^0$ mit $(s_0, r_0) \in \mathcal{B}$.

Nach Lemma 2.6 gibt es $r \in S_2$ mit $r_0 \xrightarrow{\frac{w}{2}} r$ und $s \xleftrightarrow{} r$.

Offensichtlich ist r auch erreichbar. □

Der folgende Satz zeigt, dass Bisimilarität feiner als Folgenäquivalenz ist, denn Bisimilarität trennt auch folgenäquivalente Systeme (vgl. dazu das Beispiel in Abbildung 2.1).

Satz 2.8 Wenn zwei Transitionssysteme TS_1 und TS_2 bisimilar sind, dann sind sie auch folgen- und akzeptanzäquivalent, (aber nicht umgekehrt), d.h.:

$$TS_1 \Leftrightarrow TS_2 \Rightarrow (TS_1 \sim_L TS_2) \wedge (TS_1 \sim_{AS} TS_2)$$

Beweis: Sei $TS_1 \Leftrightarrow TS_2$. Wir beweisen nun $TS_1 \sim_L TS_2$, also $L(TS_1) = L(TS_2)$:

Zu $w \in L(TS_1)$ gibt es $s_0 \in S_1^0$ und $s_1 \in S_1^F$ mit $s_0 \xrightarrow[1]{w} s_1$.

Nach Definition 2.4 a) existiert ein $r_0 \in S_2^0$ mit $s_0 \Leftrightarrow r_0$.

Nach Lemma 2.6 gibt es $r_1 \in S_2$ mit $r_0 \xrightarrow[2]{w} r_1$ und $s_1 \Leftrightarrow r_1$.

Nach Definition 2.4 c) muss auch $r_1 \in S_2^F$ gelten. Also erhalten wir $w \in L(TS_2)$.

Da $S_1^F = S_1$ und $S_2^F = S_2$ als Spezialfall enthalten ist, gilt auch $TS_1 \Leftrightarrow TS_2 \Rightarrow AS(TS_1) = AS(TS_2)$. \square

Damit folgt, dass die beiden bisimilaren Transitionssysteme von Abbildung 2.2 auch akzeptanzäquivalent sind (z.B. mit $S_1^F = \{s_0\}$ und $S_2^F = \{r_0\}$).

Satz 2.9 Wenn \mathcal{B} und \mathcal{B}' zwei Bisimulationen zwischen TS_1 und TS_2 sind, dann ist auch die Vereinigung $(\mathcal{B} \cup \mathcal{B}')$ eine.

Beweis: Als Übung. \square

Aus dem Satz folgt, dass die *größte* Bisimulation \approx zwischen TS_1 und TS_2 existiert:

$$\approx := \bigcup_{\mathcal{B} \text{ ist Bisimulation}} \mathcal{B}$$

2.3 Synchronisation von Transitionssystemen

Transitionssysteme (und damit auch Automaten) können synchronisiert werden. Durch erhalt man Modelle für nebenläufige Systeme. Prinzipiell werden folgende Arten der Synchronisation unterschieden:

- a) *Rendezvous-Synchronisation*
- b) *Speicher-Synchronisation*
- c) *Nachrichten-Synchronisation*

Bei der Rendezvous-Synchronisation (engl. auch „handshake synchronization“) werden zu synchronisierende Aktionen ungeteilt zusammen ausgeführt. Diese Form wird meist bei Transitionssystemen angewandt. Speicher-Synchronisation erfolgt durch Schreiben und Lesen auf gemeinsam zugreifbaren Speicherbereichen, z.B. auf gemeinsamen Variablen. Nachrichten-Synchronisation wird durch das Versenden und Empfangen von Nachrichten realisiert, wobei es oft keine oberen Schranken für die Nachrichtenlaufzeit gibt. Speicher- und Nachrichten-Synchronisation können durch Rendezvous-Synchronisation dargestellt werden, indem man eine Funktionseinheit (einen „Prozess“) zwischenschaltet, der die entsprechende Verwaltung des Speicherbereiches oder des Nachrichtenkanals modelliert.

Zwei synchronisierte Transitionssysteme ergeben wieder ein Transitionssystem, das als Zustandsmenge das Mengenprodukt $S_1 \times S_2$ („kartesisches Produkt“) der ursprünglichen Zustandsmengen hat und daher *Produkt-Transitionssystem* genannt wird. Die zu synchronisierenden Transitionspaare werden durch eine Relation $\text{Sync} \subseteq A_1 \times A_2$ bestimmt. Je nach dem Umfang dieser Relation ist das resultierende Produkt-Transitionssystem mehr oder weniger wieder sequentiell, d.h. erlaubt mehr oder weniger nebenläufige Aktionen. Vollständig sequentiell ist es zum Beispiel das Produkt von endlichen Automaten („Produktautomat“), das zur Konstruktion eines Automaten eingesetzt wird, welcher den Durchschnitt bzw. die Vereinigung regulärer Mengen akzeptiert.

Definition 2.10 Gegeben seien zwei Transitionssysteme $TS_i = (S_i, A_i, \text{tr}_i, S_i^0, S_i^F)$, $i = 1, 2$ (mit nicht notwendig gleichen Aktionsmengen A_i) und eine Synchronisationsrelation $\text{Sync} \subseteq A_1 \times A_2$.

Das Produkt-Transitionssystem von TS_1 und TS_2 ist das Transitionssystem

$$TS_1 \otimes_{\text{Sync}} TS_2 = (S_1 \times S_2, A_1 \cup A_2 \cup \text{Sync}, \text{tr}_3, S_1^0 \times S_2^0, S_1^F \times S_2^F),$$

wobei die Transitionsrelation tr_3 durch die folgenden Regeln Sy1, Sy2 und Sy3 definiert wird.

1. Regel Sy1: Erste Komponente

$$\frac{s_1 \xrightarrow{a_1} s_2, \quad a_1 \notin \text{pr}_1(\text{Sync}), \quad r \in S_2}{(s_1, r) \xrightarrow{a_1} (s_2, r)}$$

Dabei ist $\text{pr}_1(\text{Sync}) := \{s \mid \exists r \in A_2 : (s, r) \in \text{Sync}\}$ die Menge der ersten Komponenten (1. Projektion) und $\text{pr}_2(\text{Sync})$ entsprechend definiert.

2. Regel Sy2: Zweite Komponente

$$\frac{r_1 \xrightarrow{a_2} r_2, \quad a_2 \notin pr_2(Sync), \quad s \in S_1}{(s, r_1) \xrightarrow{a_2} (s, r_2)}$$

3. Regel Sy3: Kommunikation

$$\frac{s_1 \xrightarrow{a_1} s_2, \quad r_1 \xrightarrow{a_2} r_2, \quad (a_1, a_2) \in Sync}{(s_1, r_1) \xrightarrow{(a_1, a_2)} (s_2, r_2)}$$

Statt $TS_1 \otimes_{Sync} TS_2$ schreiben wir auch kürzer $TS_1 \otimes TS_2$, wenn Sync aus dem Kontext ersichtlich ist.

Definieren wir eine Abbildung $\gamma : Sync \rightarrow A_3$ von Sync in ein Kommunikationsalphabet A_3 , dann kann statt (a_1, a_2) in Regel Sy3 auch $\gamma(a_1, a_2)$ eingesetzt werden. Die Aktionsmenge ist dann entsprechend nicht $A_1 \cup A_2 \cup Sync$, sondern $A_1 \cup A_2 \cup A_3$.

Aufgabe 2.11 Seien TS_1 und TS_2 zwei Transitionssysteme.

1. Welches Systemverhalten beschreibt $TS_1 \otimes_{Sync} TS_2$ für $Sync = \emptyset$?
2. Welches Systemverhalten beschreibt $TS_1 \otimes_{Sync} TS_2$ für $Sync = \{(a, a) \mid a \in A\}$?

Beispiel 2.12 (einfaches Sender-Empfänger-System)

Das Beispiel von Abbildung 2.3 zeigt zwei Transitionssysteme, die als Sender S und Empfänger R über einen Kanal B kommunizieren. Der Sender nimmt Daten d mit der Aktion $r_A(d)$ („receive“) von einem Benutzer über einen Kanal A auf und schreibt dann das Datum d in den Kanal B mit der Aktion $s_B(d)$ („send“). Diese Aktion kommuniziert mit der entsprechenden Aktion $r_B(d)$ des Empfängers, der das Datum über den Kanal C an den empfangenden Benutzer abgibt. Wir setzen also $Sync = \{(s_B(d), r_B(d))\}$ und $\gamma(s_B(d), r_B(d)) = c_B(d)$. Rechts oben ist das resultierende Produkt-Transitionssystem $S \otimes R$ dargestellt. Die gemäß Regel Sy3 konstruierte Transition ist als punktierte Linie gezeichnet.

Als „externes Verhalten“ bezeichnet man die Abläufe der extern sichtbaren Aktionen. Im vorliegenden Fall sind das die Aktionen zu den Kanälen A und C . Bezogen auf $S^F = \{(s_0, r_0)\}$ als Endzustand sollten dies alle Folgen wie $r_A(d)s_C(d)r_A(d)s_C(d) \cdots r_A(d)s_C(d)$ sein. Da $r_A(d)$ und $s_C(d)$ unabhängig voneinander sind, können aber auch Vertauschungen von diesen Aktionen in der Aktionsfolge auftreten.

Dieses Verhalten erhält man dadurch, dass die interne Transition $(s_1, r_0) \xrightarrow{c_B(d)} (s_0, r_1)$ das Etikett ϵ erhält und die Bezeichner der anderen Transitionen als Etiketten übernimmt. Dieses Verhalten ist auch die Menge der akzeptablen Folgen des Transitionssystems $(S \otimes R)_{extern}$ (Abbildung 2.3 rechts unten). Diese Konstruktion entspricht der Beseitigung von ϵ -Übergängen, wie sie für Automaten in FGI-1 behandelt wurde.

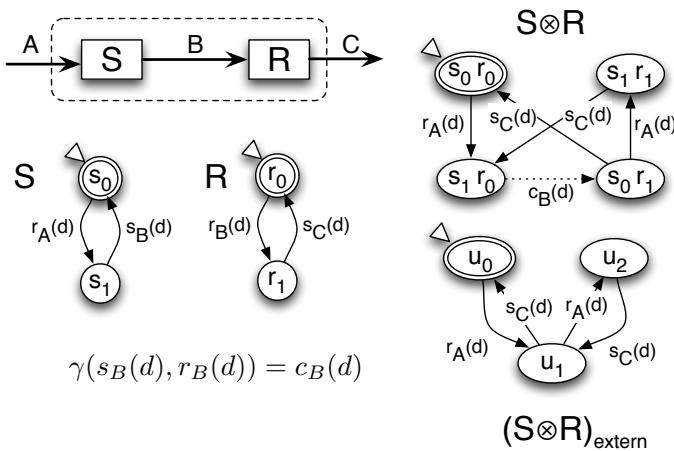


Abbildung 2.3: Ein einfaches Sender-Empfänger-System

Beispiel 2.13 (gestörtes Sender-Empfänger-System)

Um ein nicht korrektes Verhalten im vorstehenden Beispiel zu zeigen, soll nun der interne Kanal B gestört sein (Abb. 2.4). Die Störung wird dadurch dargestellt, dass der Sender das Fehlersignal \perp in den Kanal schreibt. Hier ist $\text{Sync} = \{(s_B(d), r_B(d)), (s_B(\perp), r_B(\perp))\}$ und $\gamma(s_B(d), r_B(d)) = c_B(d)$ und $\gamma(s_B(\perp), r_B(\perp)) = c_B(\perp)$. Das externe Verhalten enthält zum Beispiel die fehlerhafte Folge $r_A(d)s_C(d)r_A(d)s_C(\perp)r_A(d)s_C(d)$.

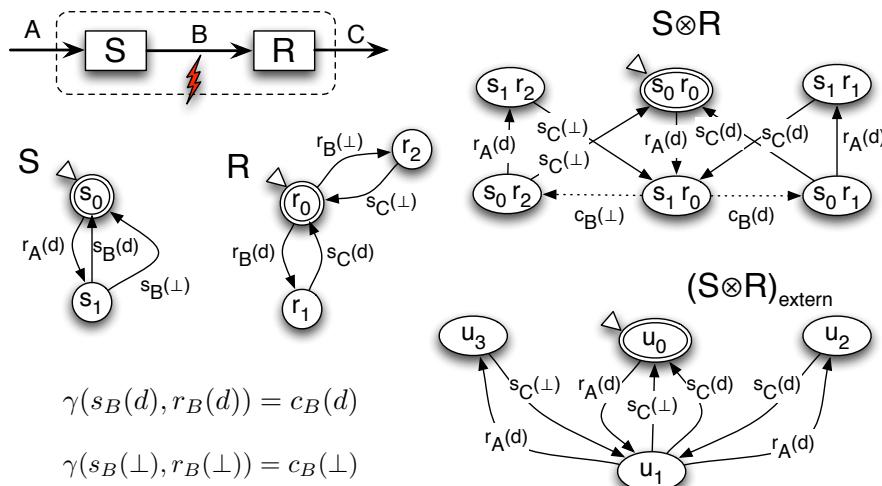


Abbildung 2.4: Ein gestörtes Sender-Empfänger-System

Beispiel 2.14 (entstörtes Sender-Empfänger-System)

Wir entwickeln im gleichen Formalismus das „Alternierbitprotokoll“. Im ersten Schritt wird eine Bestätigung über einen ungestörten Kanal D zurückgesandt (Abbildung 2.5). Wenn der Sender im Zustand s_2 ein „ok“ erhält, geht er in den Zustand s_0 und liest die nächste Eingabe. Im anderen Fall bei „not ok“ geht er nach s_1 zurück und sendet das Datum d erneut. Um das gewünschte externe Verhalten, wie durch $(S \otimes R)_{extern}$ dargestellt, zu erhalten, darf das System nicht immer in dieser Schleife bleiben. Man muss also (z.B. durch technische Massnahmen) dafür sorgen, dass der Kanal B nicht permanent gestört bleibt. Formal wird dies dadurch ausgedrückt, dass immer der Endzustand erreicht werden muss oder (bei unendlichen Prozessen) immer wieder durchlaufen werden muss.

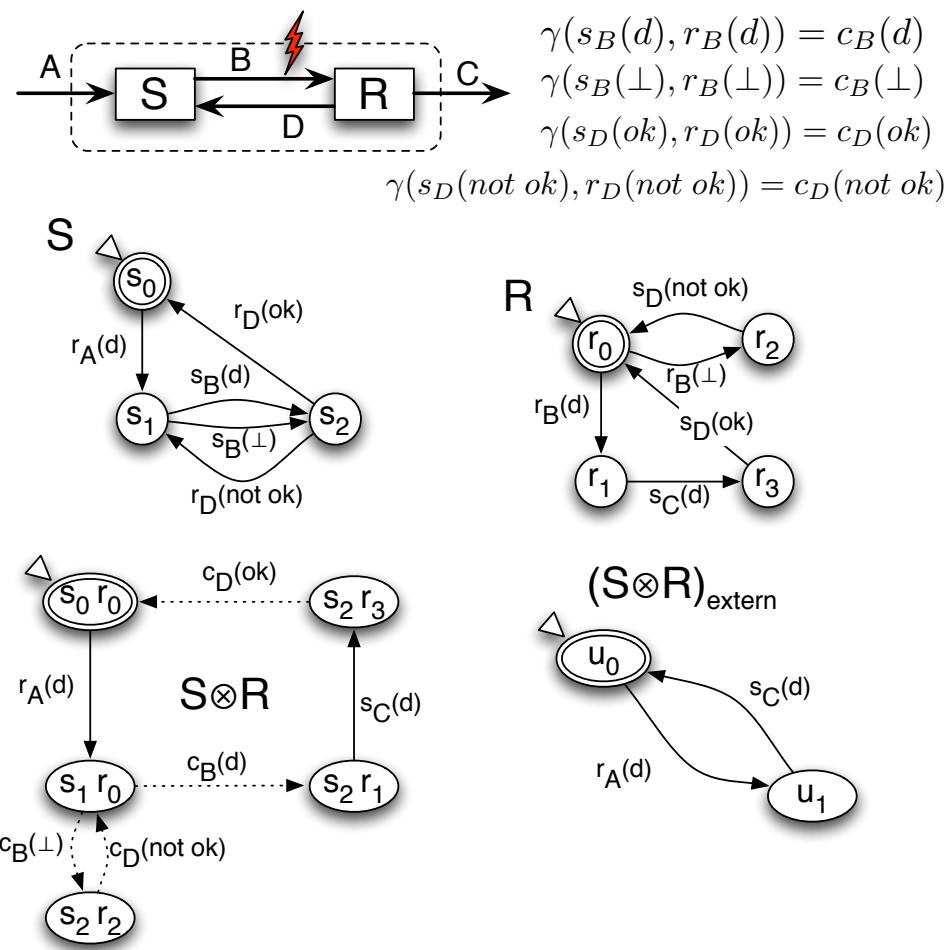


Abbildung 2.5: Entstörtes Sender-Empfänger-System

Beispiel 2.15 (Das Alternierbitprotokoll)

Die Spezifikation des Alternierbitprotokolls geht davon aus, dass auch der Bestätigungs-kanal D gestört ist, dies aber bei beiden Kanälen B und D nie permanent der Fall ist. Dazu erhält auch der Kanal D eine Bestätigungs-Prozedur. Diese kann aber mit derjenigen von B zusammengelegt werden, indem dem Datum d ein Bit mit den Werten 0 oder 1 beigefügt wird, wie z.B. in der Transition $s_1 \xrightarrow{s_B(d,0)} s_2$ in Abbildung 2.6. Erhält der Sender das Bit mit dem selben Wert (in den Zuständen s_3 oder s_0) zurück, dann kann er das nächste Datum mit alterniertem Bit versenden. Im anderen Fall sendet er das vorherige erneut (in den Zuständen s_1 oder s_4). Das externe Verhalten erhält man wieder, indem in dem Produkt-Transitionssystem $S \otimes R$ die (unterbrochen dargestellten) internen Transitionen der Kanäle B und D mit dem Etikett ϵ versieht. Das so betrachtete Produkt-Transitionssystem ist dann akzeptanzäquivalent zu $(S \otimes R)_{extern}$, das das gewünschte externe Verhalten hat.

Anmerkung: Die hier behandelten Beispiele haben den Spezialfall behandelt, dass die zu übermittelnden Daten d aus einer Datenmenge Δ stammen, die einelementig ist: $\Delta = \{d\}$. Der allgemeinere Fall einer endlichen Datenmenge $\Delta = \{d_1, \dots, d_k\}$ ist im Prinzip genauso zu behandeln. In den Zuständen des Transitionssystems muss dann das jeweilige d durch die Zustände gespeichert werden. Im Beispiel des Alternierbitprotokolls (Beispiel 2.15, Abbildung 2.6) wird der Wechsel von einem Datum zum nächsten durch den Wechsel von d zu d' dargestellt. Die Transitionssysteme werden für größere Mengen Δ unpraktikabel groß. Abhilfe schaffen hier die in den folgenden Kapitel behandelten Darstellungen durch Petrinetze und Prozessalgebra.

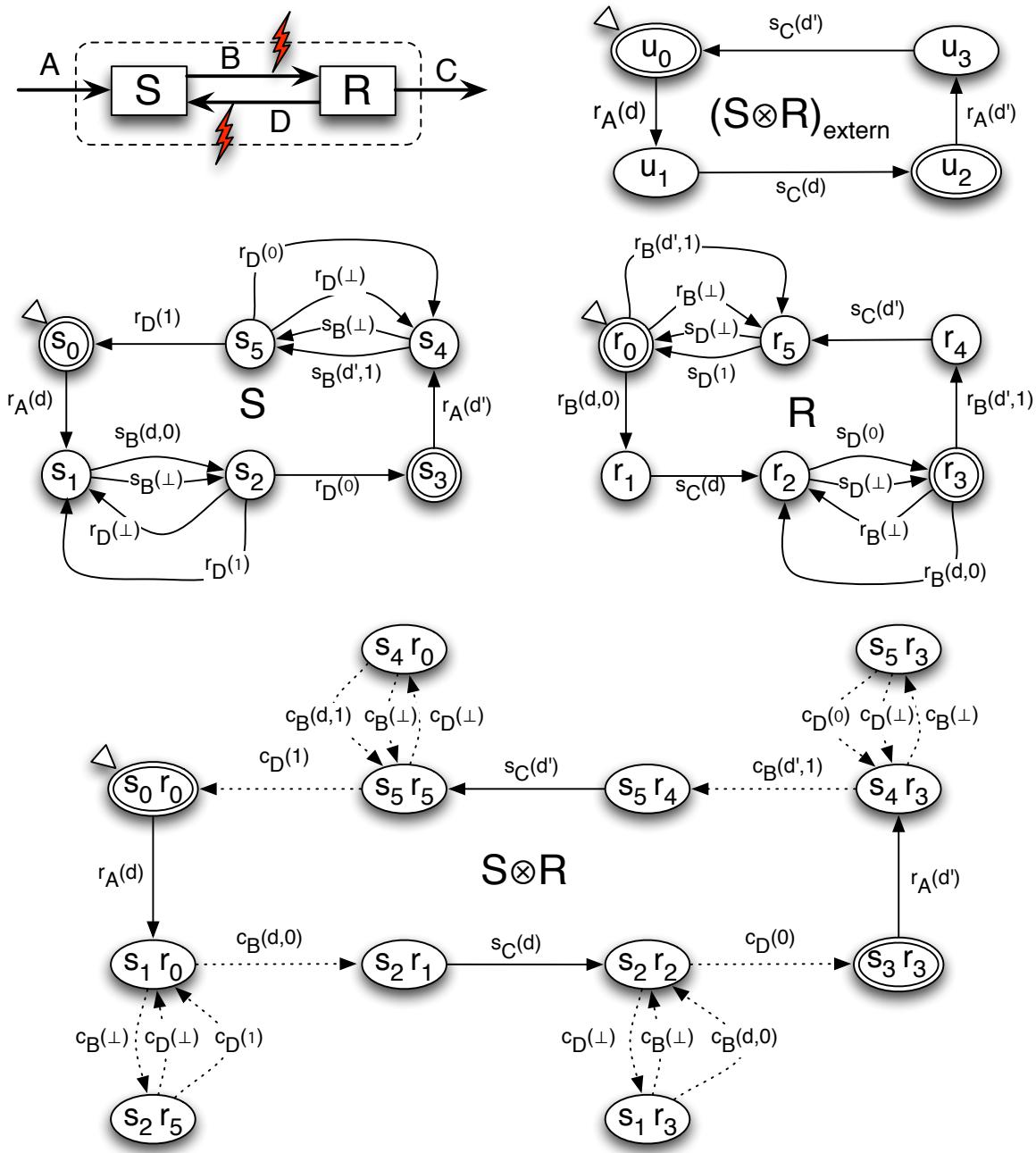


Abbildung 2.6: Das Alternierbitprotokoll

2.4 Etikettierte Transitionssysteme

Sei E eine beliebige Menge, genannt Etikettenmenge (engl. label set).

Eine *Transitionsetikettenfunktion* eines Transitionssystems TS ist eine Abbildung $E_A : A \rightarrow E$, die Aktionen auf Etikettenmenge abbildet. Wir erweitern $E_A : A \rightarrow E$ kanonisch auf Wörter $w \in A^*$:

$$E_A^*(a_1 \cdots a_n) = E_A(a_1) \cdots E_A(a_n)$$

Analog erhalten wir $E_A^\omega : A^\omega \rightarrow E^\omega$ für ω -Wörter.

Wir betrachten nun die Bilder der TS-Sprachen bzgl. der Etikettenabbildung: $E_A(TS) := E_A^*(FS(TS)) := \{E_A^*(w) \in \Sigma^* \mid w \in FS(TS)\}$ ist die *terminale Etikettsprache*.

Im Folgenden wird meist eine Transitionsetikettenfunktion $E_A : A \rightarrow E$ mit $E = \Sigma \cup \{\epsilon\}$ verwendet. Dabei ist Σ ein Etikettenalphabet und ϵ das leere Wort. E_A kann sinnvoll auch als Homomorphismus $E_A^* : A^* \rightarrow \Sigma^*$ aufgefasst werden.

Beispiel 2.16 Transitionsetiketten dienen oft dazu, interne von extern sichtbaren Aktionen zu unterscheiden. So könnte man hier die Aktionen aus $\{\text{Tee_kochen}, \text{Kaffee_kochen}\}$ als interne Aktionen deklarieren, indem man definiert:

$$E_A(a) := \begin{cases} \epsilon, & \text{falls } a \in \{\text{Tee_kochen}, \text{Kaffee_kochen}\} \\ a, & \text{sonst} \end{cases}$$

Wählt man statt der Aktionsfolgen die Bilder unter E_A , dann sind die internen Aktionen $\{\text{Tee_kochen}, \text{Kaffee_kochen}\}$ unsichtbar. Abbildung 2.7 b) zeigt eine entsprechende Darstellung.

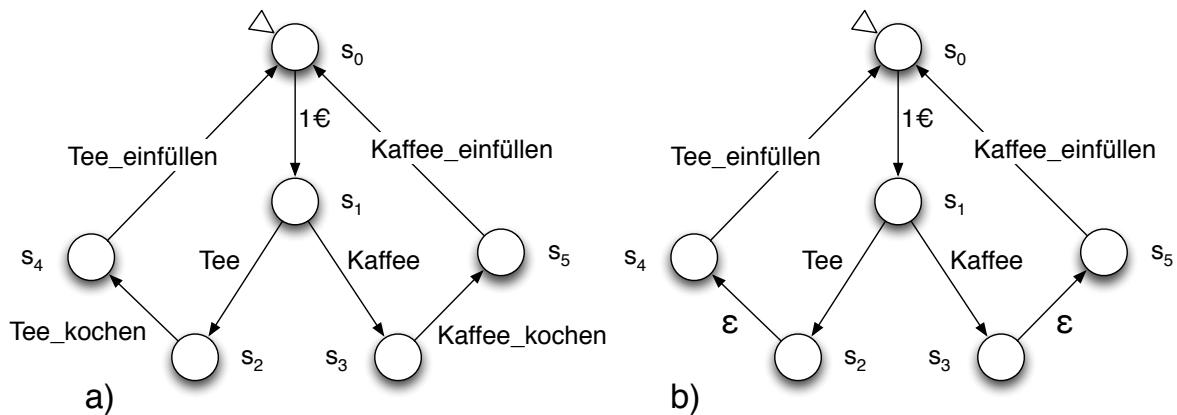


Abbildung 2.7: a) Getränkeautomat als Transitionssystem und mit internen Transitionen in b)

Analog definieren wir eine Abbildung für die Zustände. Eine *Zustandsetikettenfunktion* eines Transitionssystems $TS = (S, A, \text{tr}, S^0, S^F)$ ist eine Abbildung $E_S : S \rightarrow E$, die Zustände auf Etikettenmenge abbildet. Die *Zustandsetikettensprache* von TS ist $E_S(TS) := E_S^\omega(SS(TS)) := \{E_S^\omega(\pi) \in \Sigma^* \mid \pi \in SS(TS)\}$. Zwei TS heißen *trace-äquivalent*, wenn die Mengen ihrer Zustandsetikettenfolgen gleich sind.

2.5 Kripke-Strukturen

Wir kommen nun zu Analyseverfahren, die auf Transitionssystemen von beliebigen Systemen anwendbar sind. Wegen ihrer Wurzeln in der Logik heißen sie Kripke-Strukturen. Dabei werden weniger die Aktionsfolgen als vielmehr die Eigenschaften der Zustände betrachtet. Solche Eigenschaften werden durch atomare Aussagen in den Zuständen beschrieben.

Wir betrachteten daher eine *Zustands-Etikettenfunktion*, die jedem Zustand, die Menge der lokal gültigen atomaren Aussagen zuweist (Sei AP eine Menge von atomaren Aussagen.):

$$E_S : S \rightarrow \mathcal{P}(AP)$$

Beispiel 2.17 Durch eine Zustandsetikettenfunktion können den Zuständen Mengen von atomaren Aussagen zugeordnet werden, die in diesem Zustand gültig sind. Beispielsweise könnte man im Transitionssystem von Abbildung 2.7 b) die atomaren Aussagen $AP := \{\alpha_1, \alpha_2, \alpha_3\}$ folgendermaßen definieren:

$\alpha_1 = \text{„1€ gezahlt“}$, $\alpha_2 = \text{„Tee gewählt“}$ und $\alpha_3 = \text{„Kaffee gewählt“}$.

Die Zuordnung $E_S(s_0) = \emptyset$, $E_S(s_1) = \{\alpha_1\}$ usw. ist in Abbildung 2.8 angegeben. Das bedeutet, dass zum Beispiel bei der Analyse (Model-Checking) im Zustand s_3 nur die Eigenschaften $\{\alpha_1, \alpha_3\}$ zu prüfen sind.

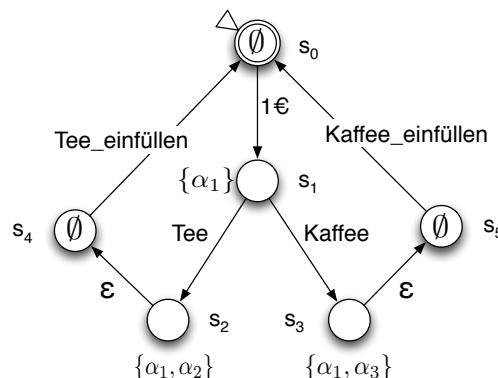


Abbildung 2.8: Getränkeautomat als Transitionssystem mit Zustandsetikettenfunktion

Eine Relation $R \subseteq X \times Y$ heißt *linkstotal*, wenn gilt: $\forall x \in X : \exists y \in Y : (x, y) \in R$.

Wir wollen nur linkstotale Transitionsrelationen betrachten, damit jede endliche Zustandsfolge Anfangsstück einer unendlichen Folge ist, d.h. keine „Sackgassen“ entstehen können. Wir können aber stets einen speziellen Sackgassen-Zustand einführen, der jedes TS mit Sackgassen in eines ohne überführt.

Eine *Kripke-Struktur* ist ein Transitionssystem $M := (S, A, tr, S_0, S^F)$ mit $S^F = \emptyset$ einelementiger Aktionenmenge (also $|A| = 1$), linkstotaler Transitionsrelation tr und zusätzlich einer Zustandsetikettenfunktion $E_S : S \rightarrow \mathcal{P}(AP)$.

Kripke-Strukturen werden üblicherweise in der folgenden Form notiert, die wir im Rest dieses Kapitels auch benutzen. Dabei wird die (hier nicht benötigte) Menge A der Aktionen weggelassen und daher die Transitionsrelation als 2-stellige, linkstotal Relation $R \subseteq S \times S$ dargestellt.

Definition 2.18 Eine Kripke-Struktur $M := (S, S_0, R, E_S)$ besteht aus

- a) einer Zustandsmenge S ,
- b) einer Menge $S_0 \subseteq S$ von Anfangszuständen,
- c) einer linkstotalen Transitionsrelation $R \subseteq S \times S$ und
- d) einer Zustandsetikettenfunktion $E_S : S \rightarrow \mathcal{P}(AP)$, die jedem Zustand s eine Menge $E_S(s) \subseteq AP$ von aussagenlogischen atomaren Formeln zuordnet.

Anmerkung: In Darstellungen und in Modelchecking-Werkzeugen werden zur besseren Lesbarkeit oft Transitionsbezeichner benutzt (wie in den Abbildungen 2.8 und 3.1). Diese gehören aber nicht zur formalen Definition.

Da eine Kripke-Struktur keine Aktionen hat, werden zur Beschreibung des Verhaltens statt $L^\omega(M)$ die unendlichen Zustandsfolgen betrachtet.

Für Transitionssysteme ist ein Pfad eine Folge $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \dots$, die in einem Startzustand beginnt. Für Kripke-Strukturen ist die Angabe von Aktionen überflüssig. Für eine Kripke-Struktur $M := (S, S_0, R, E_S)$ ist ein *Pfad* aus $s \in S$ (auch: eine *Rechnung* aus s) eine unendliche Folge:

$$\pi = s_0 s_1 s_2 \dots \in S^\omega \quad \text{mit } s_0 = s \text{ und } \forall i \geq 0 : (s_i, s_{i+1}) \in R$$

Die Menge aller Rechnungen von M ist dann (analog zu TS):

$$SS(M) := \{\pi \mid \exists s \in S^0 : \pi \text{ ist eine Rechnung aus } s\}$$

Die zu $\pi = s_0 s_1 s_2 \dots$ zugehörige Zustandsetikettenfolge ist dann:

$$E_S(\pi) = E_S(s_0) E_S(s_1) E_S(s_2) \dots \in \mathcal{P}(AP)^\omega$$

Zwei Kripke-Strukturen heißen *trace-äquivalent*, wenn die Mengen ihrer Zustandsetikettenfolgen gleich sind.

Beispiel 2.19 Die Menge aller Rechnungen der Kripke-Struktur von Abb. 2.8 ist

$$(s_0 s_1 (s_2 s_4 + s_3 s_5))^\omega$$

Die Etikettsprache ist dann:

$$\begin{aligned} & E_S((s_0 s_1 (s_2 s_4 + s_3 s_5))^\omega) \\ &= (E_S(s_0) E_S(s_1) (E_S(s_2) E_S(s_4) + E_S(s_3) E_S(s_5)))^\omega \\ &= (\emptyset \{\alpha_1\} (\{\alpha_1, \alpha_2\} \emptyset + \{\alpha_1, \alpha_3\} \emptyset))^\omega \end{aligned}$$

Hinweis: Man beachte, dass hier Mengen als Symbole verwendet werden. Man könnte die Etikettsprache (mit abgewandelten Etiketten der Form $a_X, X \in \mathcal{P}(AP)$) statt nur X) auch folgendermaßen schreiben:

$$E_S((s_0 s_1 (s_2 s_4 + s_3 s_5))^{\omega}) = (a_{\emptyset} a_{\{\alpha_1\}} (a_{\{\alpha_1, \alpha_2\}} a_{\emptyset} + a_{\{\alpha_1, \alpha_3\}} a_{\emptyset}))^{\omega}$$

Nach diesem Hinweis bleiben wir aber bei der vorherigen (und üblichen) Darstellung.

2.5.1 Darstellung von Kripke-Strukturen durch logische Formeln

Eine Kripke-Struktur kann mittels Prädikaten erster Ordnung repräsentiert werden. Die Darstellung durch logische Formeln ist für die Verarbeitung in Modelchecking-Werkzeugen wichtig und wird dort zum Beispiel für die Komplexitätsreduktion durch „binäre Entscheidungsdiagramme“ (*binary decision diagrams, BDDs*) eingesetzt.

Zustand: $s : V \rightarrow D$ ist eine Abbildung von der Menge der Variablen in eine Wertemenge. Z.B. für die Variablenmenge $V = \{v_1, v_2, v_3\}$, wird der Zustand $s = < v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5 >$ durch die zugehörige Formel $(v_1 = 2) \wedge (v_2 = 3) \wedge (v_3 = 5)$ repräsentiert. S_0 bezeichnet die Formel für den Anfangszustand.

Transition: Beziehung zwischen Werten der Variablen vor der Transition (Menge V) und Werten der Variablen nach der Transition (Menge V'). Wenn R eine Transitionsrelation ist, so bezeichnet $\mathcal{R}(V, V')$ die entsprechende Formel.

Definition 2.20 Eine Kripke-Struktur in logischer Darstellung ist ein Tupel $M := (S, S_0, R, E_S)$ mit:

1. S Menge der Belegungen,
2. $S_0 \subseteq S$ Menge von Zuständen, die die Anfangsbedingung S_0 erfüllen,
3. $\forall s, s' \in S : R(s, s') \leftrightarrow \mathcal{R}(V, V')$ mit Belegung s für V und s' für V' ,
4. $E_S(s)$ Menge der Formeln, die bei Belegung s gelten.

$$(v \in E_S(s) \text{ GDW. } s(v) = \text{wahr}, v \notin E_S(s) \text{ GDW. } s(v) = \text{falsch})$$

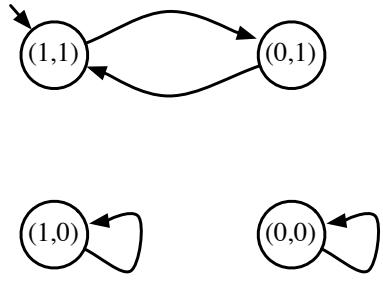


Abbildung 2.9: Kripke-Struktur zu Beispiel 2.21

Beispiel 2.21 (System mit Kripke-Struktur)

Variablenmenge: $V = \{x, y\}$,

Wertemenge: $D = \{0, 1\}$,

System: $x := (x + y) \bmod 2$,

Anfangszustand: $x = 1, y = 1$,

Das System kann mit zwei Prädikatenformeln beschrieben werden:

$$\mathcal{S}_0(x, y) \equiv x = 1 \wedge y = 1$$

$$\mathcal{R}(x, y, x', y') \equiv x' = (x + y) \bmod 2 \wedge y' = y$$

Daraus die Kripke-Struktur: $M = (S, S_0, R, E_S)$

$$S = D \times D,$$

$$S_0 = \{(1, 1)\},$$

$$R = \{((1, 1), (0, 1)), ((0, 1), (1, 1)), ((1, 0), (1, 0)), ((0, 0), (0, 0))\},$$

$$E_S((1, 1)) = \{x = 1, y = 1\}, \dots$$

Ein Pfad vom Anfangszustand: $(1, 1), (0, 1), (1, 1), (0, 1), \dots$

2.6 Kripke-Strukturen von Programmen

Exemplarisch für Programme werden nun Kripke-Strukturen für eine einfache Programmsprache definiert. Zunächst wird dies für sequentielle Programme durchgeführt und dann auf nebenläufige Programme erweitert. Illustriert wird diese Formalisierung am Beispiel eines Programmes für den wechselseitigen Ausschluss. Dieses Beispiel erläutert zudem grundsätzliche Fehlerquellen, die typisch für nebenläufige Programme sind.

2.6.1 Sequentielle Programme als Kripke-Strukturen

Es werden nun Kripke-Strukturen für eine einfache Programmsprache definiert, die die elementaren Anweisungen *Zuweisung*, *Skip* (d.i. die leere Anweisung), *Hintereinanderausführung*, *bedingte Anweisung* und *Schleife* enthält.

Zur Kennzeichnung von Zuständen erhalten die Programme Zeilenummern.

Für Zeilenummern gibt es die Variable pc (Befehlszähler, program counter). Mit $pc = \perp$ ist gemeint, dass das Programm nicht aktiv ist.

Das Prädikat $same(Y) \equiv \forall y, y' \in Y : (y' = y)$ (wobei $Y \subseteq V$) beschreibt die nicht veränderten Variablen.

Das Prädikat $pre(V)$ beschreibt die Anfangsbelegung der Variablen V .

Der Anfangszustand ist $\mathcal{S}_0 \equiv pre(V) \wedge pc = m$ mit m als Startzeilenummer.

Die Prozedur $\mathcal{C}(l, P, l')$ liefert rekursiv für eine Programmbeschreibung P die Formel \mathcal{R} zur Repräsentation der Transitionsrelation der gewünschten Kripke-Struktur, dabei sind:

l, l' Zeilenummer jeweils vor und nach der Anweisung

pc, pc' Befehlszähler - Variable

Zuweisung:

$$\mathcal{C}(l, v \leftarrow e, l') \equiv pc = l \wedge pc' = l' \wedge v' = e \wedge same(V \setminus \{v\})$$

Skip:

$$\mathcal{C}(l, skip, l') \equiv pc = l \wedge pc' = l' \wedge same(V)$$

Hintereinanderausführung:

$$\mathcal{C}(l, (P_1; l'': P_2), l') \equiv \mathcal{C}(l, P_1, l'') \vee \mathcal{C}(l'', P_2, l')$$

Bedingte Anweisung:

$$\begin{aligned} \mathcal{C}(l, \text{if } b \text{ then } l_1 : P_1 \text{ else } l_2 : P_2 \text{ endif}, l') \equiv \\ (pc = l \wedge pc' = l_1 \wedge b \wedge same(V)) \vee \\ (pc = l \wedge pc' = l_2 \wedge \neg b \wedge same(V)) \vee \\ \mathcal{C}(l_1, P_1, l') \vee \mathcal{C}(l_2, P_2, l') \end{aligned}$$

Schleifen-Anweisung:

$$\begin{aligned} \mathcal{C}(l, \text{while } b \text{ do } l_1 : P_1 \text{ endwhile}, l') \equiv \\ (pc = l \wedge pc' = l_1 \wedge b \wedge same(V)) \vee \\ (pc = l \wedge pc' = l' \wedge \neg b \wedge same(V)) \vee \\ \mathcal{C}(l_1, P_1, l) \end{aligned}$$

²Das innere Klammerpaar dient der besseren Lesbarkeit.

Beispiel 2.22

(ein kleines sequentielles Programm)

Wir bilden die Formeln für die Zuweisungsfolge

$l_1 : x := 2 \cdot y;$

$l_2 : y := y - 1;$

$l_3 \dots$ mit $V = \{x, y\}$ und den Anfangswerten $x = 1, y = 2$.

Anfangszustand: $S_0 \equiv (x = 1 \wedge y = 2 \wedge pc = l_1)$

Transitionsrelation: $\mathcal{R} \equiv \mathcal{C}(l_1, x \leftarrow 2 \cdot y, l_2) \vee$

$\mathcal{C}(l_2, y \leftarrow y - 1, l_3)$

mit $\mathcal{C}(l_1, x \leftarrow 2 \cdot y, l_2) \equiv pc = l_1 \wedge pc' = l_2 \wedge x' = 2 \cdot y \wedge y' = y$

und $\mathcal{C}(l_2, y \leftarrow y - 1, l_3) \equiv pc = l_2 \wedge pc' = l_3 \wedge y' = y - 1 \wedge x' = x$

Daraus wird folgendermaßen der Anfang einer Kripke-Struktur generiert:

Anfangsschritt: Bilde alle Anfangszustände, die S_0 erfüllen. Das können mehrere sein. In unserem Beispiel (siehe Abbildung 2.10) ist dies nur ein einziger $s_0 = (pc, x, y) = (l_1, 1, 2)$. Wäre z eine weitere Programmvariable, dann gäbe es so viele Anfangszustände wie Werte von z .

Rekursionsschritt: Wähle einen bereits konstruierten Zustand (pc, x, y) . Bestimme (pc', x', y') derart, dass die Formel der Transitionsrelation erfüllt ist.

Im Beispiel haben wir nach dem Anfangsschritt nur $(pc, x, y) = (l_1, 1, 2)$ und $s_1 = (pc', x', y') = (l_2, 4, 2)$, da nur für einen Term der Disjunktion $pc = l_1$ erfüllt ist³. Entsprechend wird aus $s_1 = (l_2, 4, 2)$ der Nachfolgezustand $s_2 = (l_3, 4, 1)$ konstruiert.

Insgesamt gesehen entspricht die Disjunktion \mathcal{R} der Vereinigung der einzelnen Transitionsübergänge $\{(s, s')\} = \{(pc, x, y), (pc', x', y')\}$.

2.6.2 Nebenläufige Programme als Kripke-Strukturen

Zu den Anweisungen von Abschnitt 2.6.1 werden nun noch die Anweisungen **cobegin/coend** für nebenläufige Ausführung und **await** für bedingtes Warten hinzugenommen.

Die Programmbeschreibung

$$P = l : \text{cobegin } P_1 \| P_2 \| \dots \| P_n \text{ coend; } l'$$

³Würde bei der Definition der Hintereinanderausführung \vee durch \wedge ersetzt, so wäre wegen $l_1 \neq l_2$ keine gültige Belegung der Formel möglich!

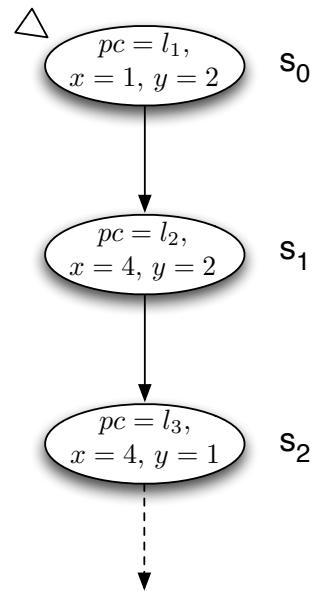


Abbildung 2.10: Anfang einer Kripke-Struktur zum Programm von Beispiel 2.22

wird oft mit Zeilenummern versehen:

$$P = l : \text{cobegin } l_1 : P_1 \parallel \dots \parallel l_n : P_n \text{; coend; } l'$$

Dafür ergeben sich die Formeln, wobei $PC = \{pc, pc_i | pc_i\text{-Befehlszähler von } P_i\}$:

$$\mathcal{S}_0(V, PC) \equiv pre(V) \wedge pc = l \wedge \bigwedge_{i=1}^n (pc_i = \perp)$$

P_i sind also am Anfang nicht aktiv. Damit ergibt sich folgende Repräsentation:

$$\begin{aligned} \mathcal{C}(l, P, l') \equiv & \\ (pc = l \wedge pc'_1 = l_1 \wedge \dots \wedge pc'_n = l_n \wedge pc' = \perp) \vee & \quad (\text{Initialisierung}) \\ (pc = \perp \wedge pc_1 = l'_1 \wedge \dots \wedge pc_n = l'_n \wedge pc' = l' \wedge \bigwedge_{i=1}^n (pc'_i = \perp)) \vee & \quad (\text{Termination}) \\ (\bigvee_{i=1}^n (\mathcal{C}(l_i, P_i, l'_i) \wedge \text{same}(V \setminus V_i) \wedge \text{same}(PC \setminus \{pc_i\}))) & \quad (\text{Transitionen von } P_i) \end{aligned}$$

V_i ist die Menge der Variablen die von Programm P_i geändert werden.

await-Anweisung:

$$\begin{aligned} \mathcal{C}(l, \text{await}(b), l') \equiv & \\ (pc_i = l \wedge pc'_i = l \wedge \neg b \wedge \text{same}(V_i)) & \quad \text{"busy waiting"} \\ \vee (pc_i = l \wedge pc'_i = l' \wedge b \wedge \text{same}(V_i)) & \end{aligned}$$

Als Beispiel behandeln wir den „wechselseitigen Ausschluss“ (*mutual exclusion*), eine Erscheinung, die von grundsätzlicher Bedeutung ist.

2.6.3 Wechselseitiger Ausschluss

Parallele Programme oder Prozesse können zum konsistenten Schreiben auf gemeinsame Daten einen *kritischen Abschnitt* enthalten, der nicht überlappend ausgeführt werden darf. Dies kommt im nicht kritischen Abschnitt nicht vor.

Der Algorithmus von Dekker/Petersen war der erste, der dies ohne betriebssystemunterstützte Operationen (wie Semaphore oder synchronized-Methoden in Java) realisierte. Die Programme sollen dabei (für den Fall zweier Prozesse P und Q) folgende Eigenschaften erfüllen:

- A) Die Befehlszähler von P und Q sind nie gleichzeitig in ihren kritischen Abschnitten.
- B) Meldet der Prozess P oder Q den Wunsch zum Eintritt in den kritischen Abschnitt an ($wantP = True$ oder $wantQ = True$), so kann er nach einer gewissen endlichen Zeit tatsächlich in seinen kritischen Abschnitt eintreten.

Bei diesen Programmen gehen wir davon aus, dass alle elementaren Anweisungen (also Zuweisungen und Tests) durch einen Speichersperrmechanismus ungeteilt (atomar) ausgeführt werden. Vorausgesetzt wird natürlich auch, dass der kritische Abschnitt nach einer gewissen Zeit auch wieder verlassen wird. A) und B) sind Minimalforderungen. Eine weitere Forderung C) ist zum Beispiel, dass der Eintritt in den kritischen Abschnitt nicht abwechselnd erfolgen muss.

Wir geben hier eine Einführung nach Dijkstra wieder und benutzen dabei das Schema von Algorithmus 2.1.

Algorithmus 2.1 (Wechselseitiger Ausschluss nach Dijkstra - Programmschema)

P0: Initialisierung $m : \text{cobegin } P \parallel Q \text{ coend}$ wobei	$P: l_0 : \text{while True do}$ $p_i : \text{non-critical section};$ $Eintrittsprotokoll$ $p_j : \text{critical section};$ $Austrittsprotokoll$ endwhile l'_0	$Q: l_1 : \text{while True do}$ $q_i : \text{non-critical section};$ $Eintrittsprotokoll$ $q_j : \text{critical section};$ $: Austrittsprotokoll$ endwhile l'_1
--	---	---

Ein erster Lösungsversuch folgt der Vorstellung einer Ampel, die auf grün und rot gesetzt wird (Algorithmus 2.2). Was geht hier schief?

Algorithmus 2.2 (Wechselseitiger Ausschluss nach Dijkstra - 1)

P1: $\text{Ampel} = \text{grün}: \{rot, grün\},$ $m : \text{cobegin } P \parallel Q \text{ coend}$ wobei	$P: l_0 : \text{while True do}$ $p_0 : \text{non-critical section};$ $p_2 : \text{await}(\text{Ampel} = \text{grün});$ $p_3 : \text{Ampel} := \text{rot};$ $p_4 : \text{critical section};$ $p_5 : \text{Ampel} := \text{grün};$ endwhile l'_0	$Q: l_1 : \text{while True do}$ $q_0 : \text{non-critical section};$ $q_2 : \text{await}(\text{Ampel} = \text{grün});$ $q_3 : \text{Ampel} := \text{rot};$ $q_4 : \text{critical section};$ $q_5 : \text{Ampel} := \text{grün};$ endwhile l'_1
--	---	---

Der zweite Versuch in Algorithmus 2.3 führt das Anmelden eines Zutrittswunsches durch $wantP$ und $wantQ$ ein. Ein Prozess prüft, ob dieser bei dem anderen vorliegt. Was läuft hier falsch?

Algorithmus 2.3 (Wechselseitiger Ausschluss nach Dijkstra - 2)

P2: $wantP = wantQ = False : \text{boolean},$ $m : \text{cobegin } P \parallel Q \text{ coend}$ wobei	$P: l_0 : \text{while True do}$ $p_0 : \text{non-critical section};$ $p_1 : wantP := True;$ $p_3 : \text{await}(wantQ = False)$ $p_4 : \text{critical section};$ $p_5 : wantP := False;$ endwhile l'_0	$Q: l_1 : \text{while True do}$ $q_0 : \text{non-critical section};$ $q_1 : wantQ := True;$ $q_3 : \text{await}(wantP = False)$ $q_4 : \text{critical section};$ $q_5 : wantQ := False;$ endwhile l'_1
---	---	---

Der dritte Versuch in Algorithmus 2.4 versucht die Verklemmung durch das temporäre Aufheben des Zutrittswunsches zu vermeiden. Warum funktioniert das nicht?

Algorithmus 2.4 (Wechselseitiger Ausschluss nach Dijkstra - 3)

P3: $wantP = wantQ = False : boolean,$ $m : \text{cobegin } P \parallel Q \text{ coend}$ wobei P: $l_0 : \text{while True do}$ $\quad p_0 : \text{non-critical section};$ $\quad p_1 : wantP := True;$ $\quad p_3 : \text{while } wantQ = True \text{ do}$ $\quad \quad wantP := False;$ $\quad \quad wantP := True$ $\quad \text{endwhile}$ $\quad p_4 : \text{critical section};$ $\quad p_5 : wantP := False;$ $\text{endwhile } l'_0$	Q: $l_1 : \text{while True do}$ $\quad q_0 : \text{non-critical section};$ $\quad q_1 : wantQ := True;$ $\quad p_3 : \text{while } wantP = True \text{ do}$ $\quad \quad wantQ := False;$ $\quad \quad wantQ := True$ $\quad \text{endwhile}$ $\quad q_4 : \text{critical section};$ $\quad q_5 : wantQ := False;$ $\text{endwhile } l'_1$
--	--

Der vierte Versuch, der ursprünglich von Dekker stammt und von Peterson auf die vorliegende Form verbessert wurde, löst die Verklemmung im 2. Algorithmus durch eine „tie break rule“, indem derjenige Prozess im Konfliktfall in den kritischen Abschnitt eintreten darf, der sich nicht zuletzt für den Eintritt angemeldet hat (Variable *last* im Algorithmus 2.5). Ob dieser Algorithmus das Problem löst, wird im Folgenden mit seiner Kripke-Struktur untersucht.

Algorithmus 2.5 (Wechselseitiger Ausschluss nach Dekker/Peterson (Dijkstra - 4))

P4: $wantP = wantQ = False : boolean,$ $last = 1 \vee last = 2 : integer,$ $m : \text{cobegin } P \parallel Q \text{ coend}$ wobei P: $l_0 : \text{while True do}$ $\quad [\quad p_0 : \text{non-critical section};]$ $\quad p_1 : wantP := True;$ $\quad p_2 : last := 1;$ $\quad p_3 : \text{await}(wantQ = False$ $\quad \quad \vee last = 2);$ $\quad [\quad p_4 : \text{critical section};]$ $\quad p_5 : wantP := False;$ $\text{endwhile } l'_0$	Q: $l_1 : \text{while True do}$ $\quad [\quad q_0 : \text{non-critical section};]$ $\quad q_1 : wantQ := True;$ $\quad q_2 : last := 2;$ $\quad q_3 : \text{await}(wantP = False$ $\quad \quad \vee last = 1);$ $\quad [\quad q_4 : \text{critical section};]$ $\quad q_5 : wantQ := False;$ $\text{endwhile } l'_1$
--	---

Kripke-Strukturen (d.h. Zustandsräume) von parallelen Programmen wachsen sehr schnell. Daher sucht man nach Methoden, um diese Größe zu reduzieren, ohne die Analysemöglichkeit einzuschränken. Dies ist im kleinen Maßstab auch bei diesem Beispiel möglich. Wir können nämlich die Zeilen p_0 , p_4 , q_0 und q_4 im Algorithmus 2.5 streichen und für dieses reduzierte Programm immer noch den wechselseitigen Ausschluss prüfen, indem wir beweisen, dass die Befehlszähler nie gleichzeitig in p_5 und q_5 sind. Das gilt auch für andere Eigenschaften.

Im Folgenden ist das (reduzierte) Programm

$$P = l : \text{cobegin } l_0 : P \parallel l'_0 \parallel \dots \parallel l_1 : Q \parallel l'_1 ; \text{coend}; l'$$

in der zuvor eingeführten Notation dargestellt.

$PC = \{pc, pc_0, pc_1\}$, $V = V_0 = V_1 = \{wantP, wantQ, last\}$
 pc_0 nimmt die Werte $\{p_1, p_2, p_3, p_5\}$ an und pc_1 die Werte $\{q_1, q_2, q_3, q_5\}$.

Der Anfangszustand ist:

$$\mathcal{S}_0(V, PC) \equiv pc = l \wedge pc_0 = \perp \wedge pc_1 = \perp$$

Die Transitionsrelation ist repräsentiert durch $\mathcal{R}(V, PC, V', PC')$ als Disjunktion der folgenden Formeln:

- $pc = l \wedge pc'_0 = l_0 \wedge pc'_1 = l_1 \wedge pc' = \perp$
 - $pc_0 = l'_0 \wedge pc_1 = l'_1 \wedge pc' = l' \wedge pc'_0 = \perp \wedge pc'_1 = \perp$
 - $\mathcal{C}(l_0, P, l'_0) \wedge \text{same}(V \setminus V_0) \wedge \text{same}(PC \setminus \{pc_0\})$ also $\mathcal{C}(l_0, P, l'_0) \wedge \text{same}(pc, pc_1)$
 - $\mathcal{C}(l_1, Q, l'_1) \wedge \text{same}(V \setminus V_1) \wedge \text{same}(PC \setminus \{pc_1\})$ also $\mathcal{C}(l_1, Q, l'_1) \wedge \text{same}(pc, pc_0)$

Dabei ist $\mathcal{C}(l_0, P, l'_o)$ die Disjunktion von folgenden Formeln:

- $pc_0 = l_0 \wedge pc'_0 = p_1 \wedge \text{True} \wedge \text{same}(V)$ (Schleifen-Anweisung)
 - $pc_0 = p_1 \wedge pc'_0 = p_2 \wedge \text{wantP}' = \text{True} \wedge \text{same}(V \setminus \{\text{wantP}\})$ (Zuweisung)
 - $pc_0 = p_2 \wedge pc'_0 = p_3 \wedge \text{last}' = 1 \wedge \text{same}(V \setminus \{\text{last}\})$ (Zuweisung)
 - $pc_0 = p_3 \wedge pc'_0 = p_3 \wedge \neg(\text{wantQ} = \text{False} \vee \text{last} = 2) \wedge \text{same}(V)$ (await-Anweisung)
 - $pc_0 = p_3 \wedge pc'_0 = p_5 \wedge (\text{wantQ} = \text{False} \vee \text{last} = 2) \wedge \text{same}(V)$ (await-Anweisung)
 - $pc_0 = p_5 \wedge pc'_0 = l_0 \wedge \text{wantP}' = \text{False} \wedge \text{same}(V \setminus \{\text{wantP}\})$ (Zuweisung)

$\mathcal{C}(l_1, Q, l'_1)$ ist entsprechend definiert.

Abbildung 2.11 zeigt die zugehörige Kripke-Struktur. Dabei soll eine Knotenbeschriftung (i, j, n, b_1, b_2) den Zustand $(p_i, q_j, \text{last} = n, \text{wantP} = b_1, \text{wantQ} = b_2)$ bezeichnen.

Gelten die aufgestellten Forderungen?

- A) Die Befehlszähler von P und Q sind nie gleichzeitig in ihren kritischen Abschnitten. Für die Kripke-Struktur heißt dies, dass in keinem Zustand der Prozess P im Zustand $p_i = p_5$ und gleichzeitig der Prozess Q im Zustand $q_j = q_5$ ist. In der temporalen Logik wird dies als $\square \neg(p_5 \wedge q_5)$ ausgedrückt.

B) Meldet der Prozess P oder Q den Wunsch zum Eintritt in den kritischen Abschnitt an ($wantP = True$ oder $wantQ = True$), so kann er nach einer gewissen endlichen Zeit tatsächlich in seinen kritischen Abschnitt eintreten. Dies bedeutet z.B. für den Prozess P in der Kripke-Struktur folgendes: von jedem Knoten mit $p_i = p_1$ stößt jeder Pfad irgendwann einmal auf einen Knoten mit $p_j = p_5$. In der temporalen Logik wird dies für den Prozess P als $\square(p_1 \Rightarrow \diamond(p_5))$ ausgedrückt.

Dies kann in der Kripke-Struktur von Abbildung 2.11 verifiziert werden. Der folgende Abschnitt stellt die Grundlagen für die algorithmische Lösung solcher Probleme dar.

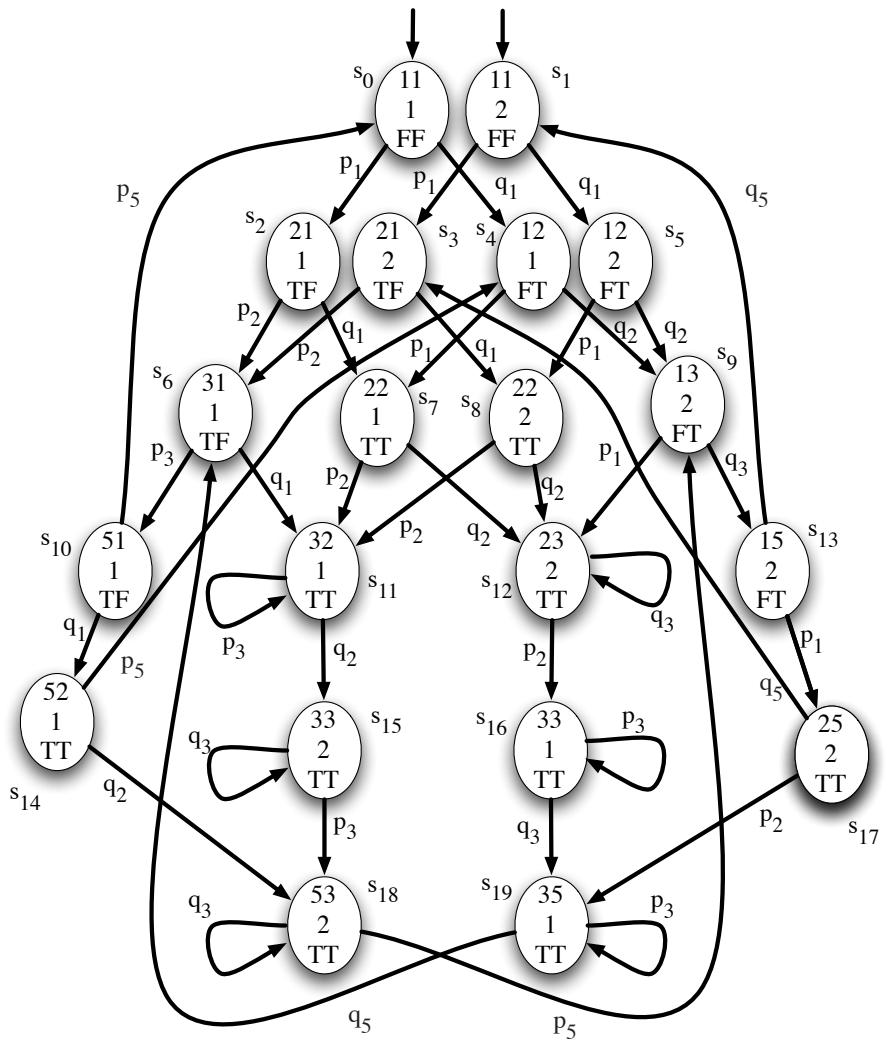


Abbildung 2.11: Kripke-Struktur zum Beispielprogramm „wechselseitiger Ausschluss“

3 Temporallogik

Die Validierung von Hardware- und Softwaresystemen wird zunehmend wichtig. Unter einer System-Validierung versteht man traditionell: **Testen und Simulation**. Dies ist am Anfang bei einfachen Fehlern sehr wirksam, ist aber im Laufe von Projekten immer weniger effektiv, da dann oftmals komplexe und verborgene Fehler auftreten. Fehler treten insbesondere bei Systemen mit Asynchronität, Parallelität, Nebenläufigkeit auf. Dies sind häufig Fehler, die von speziellen Zeitparametern/Nachrichtenlaufzeiten abhängig sind.

Eine Alternative zum Testen bietet die **formale Verifikation**, die eine umfassende Prüfung des Systemverhaltens beinhaltet. Zentraler Ansatz der formalen Verifikation sind das *Theorembeweisen* (bspw. in Zusammenhang mit der axiomatischen Semantik nach Hoare-Systeme) und die Zustandsraumanalyse (engl. Model-Checking) (meist in Zusammenhang mit temporaler Logik). Ersteres wird in der Vorlesung *Semantik von Programmiersprachen* (Modul FGI 3) behandelt. Eine Einführung zu letzterem wird in diesem Kapitel gegeben.

Model-Checking hat folgende Vor- und Nachteile: Von Vorteil ist, dass es ohne besondere Kenntnisse anwendbar ist und bei nicht korrekten Systemen Abläufe, die zu den Fehlern führen, generieren kann. Der Nachteil ist, dass praktische Systeme oftmals einen sehr großen Zustandsraum besitzen, der nicht mehr (zumindest nicht mehr unmittelbar) effizient behandelt werden kann.¹

- Literatur**
- E.M. Clarke et al.: *Model Checking*, The MIT Press, Cambridge, 1999, [CGP99]
 - C. Baier, J.-P. Katoen: *Principles of Model Checking*, MIT Press, 2008, [BK08]
 - B. Bérard et al.: *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer, Berlin, 1999, [BBF99]
 - C. Girault, R. Valk: *Petri Nets for Systems Engineering*, Part III: Verification, Springer, Berlin, 2003, [GV03]
 - M. Huth, M. Ryan: *Logic in Computer Science*, Cambridge Univ. Press, 2004, [HR04]

¹Um große Zustandsräume zu meistern, existieren einige Techniken (wie *symbolisches* Model-Checking, *Faltung*, Ausnutzung von Symmetrien), die wir hier aber nicht behandeln.

3.1 Temporale Logik

Die *temporale Logik* erlaubt es, Aussagen, die sich auf später einzunehmende Zustände beziehen, direkt auszudrücken. Solche Aussagen sind auch in der Prädikatenlogik möglich, wie dies zur Beschreibung von Lebendigkeits-Invarianzeigenschaften ausgeführt wird (siehe 7.1.2) oder wie die folgende Spezifikation eines Aufzuges zeigt.

Beispiel 3.1 (Spezifikation eines Aufzuges (Fragment))

Die folgenden zwei Teile einer Spezifikation des gewünschten Verhaltens eines Aufzuges seien gegeben:

- I. Jede Anforderung des Aufzugs wird auch erfüllt.
- II. Der Aufzug passiert kein Stockwerk mit einer nicht erfüllten Anforderung.

Die Spezifikationen I und II können mit Hilfe der Variablen t als Parameter für die ablaufende Zeit in Prädikatenlogik ausgedrückt werden, wie dies z.B. in der Physik erfolgt: $z(t) = -\frac{1}{2}gt^2$ (freier Fall des Aufzuges).

$$\begin{aligned} \text{I. } & \forall t, \forall n : app(n, t) \Rightarrow \exists t' > t : serv(n, t') \\ \text{II. } & \forall t, \forall t' > t, \forall n \left(\left(app(n, t) \wedge H(t') \neq n \wedge \exists t_{trav} . t \leq t_{trav} \leq t' \wedge H(t_{trav}) = n \right) \right. \\ & \quad \left. \Rightarrow \left(\exists t_{serv} . t \leq t_{serv} \leq t' \wedge serv(n, t_{serv}) \right) \right) \end{aligned}$$

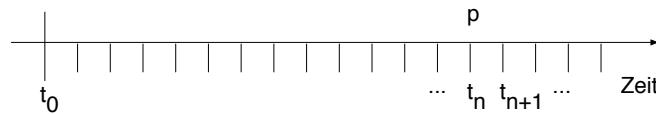
Dabei bedeuten $H(t)$ die Position des Fahrstuhls zur Zeit t , $app(n, t)$, dass eine offene Anforderung von Stockwerk n zur Zeit t besteht und $serv(n, t)$, dass der Fahrstuhl Stockwerk n bedient.

In der temporalen Logik wird der Zeit-Parameter nicht explizit benutzt. Dadurch werden die Formeln einfacher und die Entscheidungsprozeduren zur Gültigkeitsprüfung effektiv durchführbar.

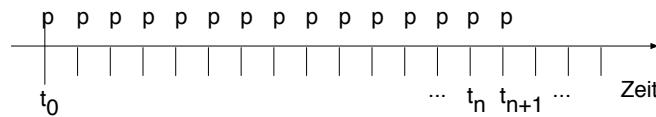
3.2 Linear Time Logic (LTL)

Amir Pnueli hat 1977 die temporale Logik erstmals für die Programmverifikation vorgeschlagen, indem er die Zeit als Programmschritte interpretierte. Von ihm stammt die *linear time logic* (LTL). Elementare temporale Quantoren sind $\diamond p$ und $\square p$:

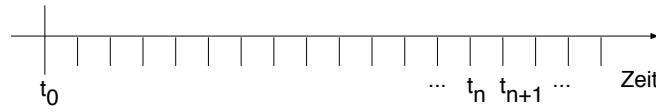
- $\diamond p$ bedeutet: Irgendwann einmal gilt p



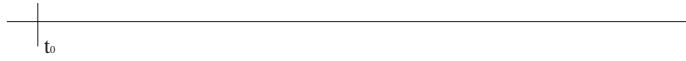
- $\square p$ bedeutet: Von jetzt an gilt immer p



- $\diamond \square p$ bedeutet: ???



- $\square \diamond p$ bedeutet: ???



3.2.1 Syntax von LTL-Formeln

Wir definieren die Syntax von LTL-Formeln in Bezug auf eine Menge AP von aussagenlogischen Elementaraussagen (atomaren Formeln).

Definition 3.2 Es sei AP eine Menge von aussagenlogischen Atomen. Dann wird die Syntax von LTL-Formeln wie folgt durch Backus-Naur-Form definiert:

$$f ::= \text{true} | \text{false} | p | (\neg f) | (f \wedge f) | (f \vee f) | (Xf) | (Ff) | (Gf) | (fUf),$$

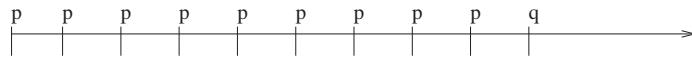
wobei $p \in AP$ ist.

3.2.2 Semantik von LTL-Formeln

LTL-Quantoren X , F , G und U beziehen sich auf unendliche Folgen $\alpha = A_0 A_1 A_2 \dots$ von Mengen $A_i \in \mathcal{P}(AP)$, d.h. jedes A_i ist eine Menge von gültigen atomaren Aussagen.

- Xp „next time“: p gilt im zweitem Element A_1 der Folge (auch $\bigcirc p$ geschrieben),
- Fp „eventually, in the future“: p gilt in einem Element einer gegebenen Folge (auch $\diamond p$),

- Gp „always, globally“: p gilt in allen Elementen einer gegebenen Folge (auch $\Box p$),
- $p \ U q$ „until“: es gibt ein Element der gegebenen Folge, in dem q gilt, und vor diesem Element gilt immer p . Illustriert:



Dies wird in der folgenden Definition induktiv auf LTL-Formeln übertragen.

Definition 3.3 Sei $\alpha = A_0 A_1 A_2 \dots \in \mathcal{P}(AP)^\omega$ eine unendliche Folge von Aussagemengen.

Für $i \in \mathbb{N}$ sei die Folge $\alpha^i = A_i A_{i+1} \dots$ der i -Suffix von α .

In der folgenden induktiven Definition seien $p \in AP$ und f, f_1, f_2 LTL-Formeln.

1. $\alpha \models \text{true}$.
2. $\alpha \not\models \text{false}$.
3. $\alpha \models p \Leftrightarrow p \in A_0$.
4. $\alpha \models \neg f \Leftrightarrow \alpha \not\models f$.
5. $\alpha \models f_1 \vee f_2 \Leftrightarrow \alpha \models f_1 \text{ oder } \alpha \models f_2$.
6. $\alpha \models f_1 \wedge f_2 \Leftrightarrow \alpha \models f_1 \text{ und } \alpha \models f_2$.
7. $\alpha \models Xf \Leftrightarrow \alpha^1 \models f$.
8. $\alpha \models Ff \Leftrightarrow \exists k \geq 0 : \alpha^k \models f$.
9. $\alpha \models Gf \Leftrightarrow \forall k \geq 0 : \alpha^k \models f$.
10. $\alpha \models f_1 U f_2 \Leftrightarrow \exists k \geq 0 . \alpha^k \models f_2 \text{ und für alle } 0 \leq j < k \text{ gilt } \alpha^j \models f_1$.

Hierbei ist $\alpha \models f$ zu lesen als „für die Folge α gilt f “ oder „ α erfüllt f “.

Die Sprache $L^\omega(f) := \{\alpha \in \mathcal{P}(AP)^\omega \mid \alpha \models f\}$ heißt: Menge der α erfüllenden Folgen über AP oder: die Sprache von α .

Jede LTL-Formel f definiert also eine ω -Sprache (nämlich $L^\omega(f)$) im Sinne der Definition 1.14 (Seite 7).

Wegen der folgenden Äquivalenzen genügen die Operatoren \vee, \neg, X, U , um alle Formeln von LTL auszudrücken:

- $f_1 \wedge f_2 \equiv \neg(\neg f_1 \vee \neg f_2)$,
- $F f \equiv \text{true} U f$,
- $G f \equiv \neg F \neg f$.

Zur Definition dieser Äquivalenz siehe Definition 3.5 (am Ende).

Aufgabe 3.4 Beweisen Sie diese Äquivalenzen.

Eine LTL-Formel f gilt für eine unendliche Folge $\pi = s_0 s_1 s_2 \dots$ von Zuständen einer Kripke-Struktur $M := (S, S_0, R, E_S)$, wenn sie für die zugehörige Zustandkettenfolge $E_S(\pi)$ gilt:

$$E_S(\pi) = E_S(s_0) E_S(s_1) E_S(s_2) \dots \in \mathcal{P}(AP)^\omega$$

Man schreibt $M, \pi \models f$.

Definition 3.5 Sei M eine Kripke-Struktur und $\pi \in SS(M)$ eine Pfad von M . Dann sei $M, \pi \models f : \Leftrightarrow E_S(\pi) \models f$.

Eine LTL-Formel f gilt (ist gültig) im Zustand $s \in S$ einer Kripke-Struktur M falls $M, \pi \models f$ für alle Pfade $\pi \in SS(M)$ gilt, die in s beginnen (notiert als $M, s \models f$).

Eine LTL-Formel f gilt in M , falls sie in allen Anfangszuständen gilt, also: $M \models f : \Leftrightarrow \forall s \in S_0 : M, s \models f$.

Gilt für alle Kripke-Strukturen M die Beziehung: $M \models f$ impliziert $M \models g$, dann notieren wir dies als $f \models g$.

Gilt für alle Kripke-Strukturen M die Beziehung: $M \models f$ gdw. $M \models g$, dann notieren wir dies als $f \equiv g$.

Beispiel 3.6 MW-Ofen Die Abbildung 3.1 zeigt eine Kripke-Struktur M als Systembeschreibung eines Mikrowellenofens. Die Transitionsbezeichner und negierten Aussagen in $E_S(s)$ dienen nur der Verdeutlichung und können wie in der Definition einer Kripke-Struktur entfallen. Die als LTL-Formel $f = G(\neg Heat U Close)$ gegebene Spezifikation gilt für M , also $M, \pi \models f$ für jede unendliche Folge $\pi = s_0s_1s_2 \dots \in SS(M)$, da beginnend mit $s_0 = 1$ immer $\neg Heat$ gilt bis $Close$ eintritt, was auch in jedem solchen Pfad tatsächlich geschieht. Dagegen gilt die Spezifikation $g = G(Start \rightarrow F Heat)$ nicht. Ein Gegenbeispiel ist die Folge $\pi = s_0s_1s_2 \dots = 1(25)^\omega \in S^\omega$ oder auch $(1253)^\omega \in S^\omega$.

Es gilt $M, s \models f$ für $s = 1$, also $M \models f$, nicht jedoch $M \models g$.

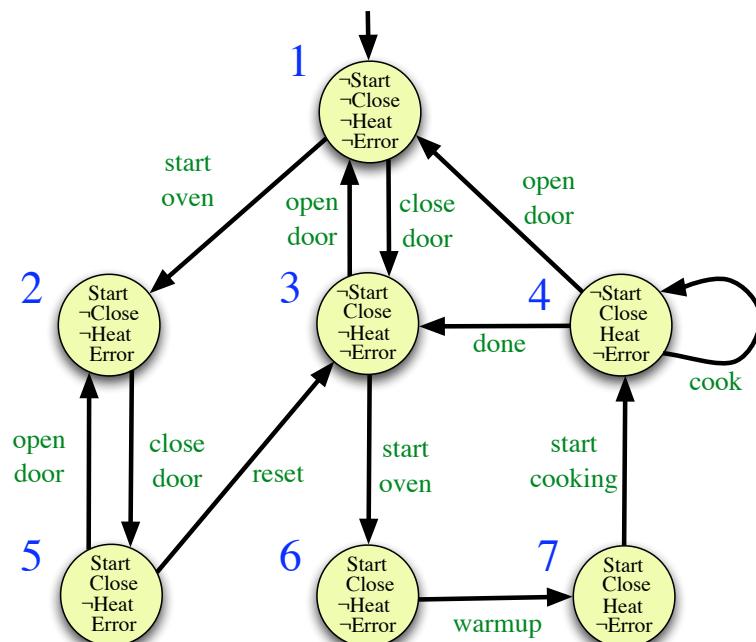


Abbildung 3.1: Kripke-Struktur für das Mikrowellenofenbeispiel

Fairnesseigenschaften wie von Definition 6.30 lassen sich gut in LTL formulieren. Jedoch nicht alle sinnvollen Eigenschaften lassen sich durch LTL-Formeln spezifizieren. Ein Beispiel ist folgende Eigenschaft: Der Aufzug kann im 3. Stock bleiben und immer die Türen auf- und zumachen. Ein weiteres Beispiel ist: Von jedem Zustand ist es möglich einen „Neustart“-Zustand zu erreichen, d.h. es gibt immer einen solchen Pfad. Diese Eigenschaft gehört zu den Lebendigkeits-Invarianzeigenschaften von Definition 7.6. Hier wird ein Existenz-Quantor für Pfade benötigt, wie er in CTL möglich ist.

Lemma 3.7 *Sind die etikettierten Zustandsfolgen zweier Kripke-Strukturen M_1 und M_2 gleich, dann können M_1 und M_2 durch keine LTL-Formel unterschieden werden, d.h. für alle LTL-Formeln ψ gilt:*

$$M_1 \models \psi \iff M_2 \models \psi$$

Beweis: Wenn $E_1(SS(M_1)) = E_2(SS(M_2))$ gilt, dann folgt:

$$\begin{aligned} & M_1 \models \psi \\ \iff & \forall \pi \in SS(M_1) : M_1, \pi \models \psi \\ \iff & \forall \pi \in SS(M_1) : E_1(\pi) \models \psi \\ \iff & \forall \sigma \in E_1(SS(M_1)) : \sigma \models \psi \\ \iff & \forall \sigma \in E_2(SS(M_2)) : \sigma \models \psi \\ \iff & \forall \pi \in SS(M_2) : E_2(\pi) \models \psi \\ \iff & \forall \pi \in SS(M_2) : M_2, \pi \models \psi \\ \iff & M_2 \models \psi \end{aligned}$$

Also sind die Modelle äquivalent. □

3.3 Computation Tree Logic (CTL)

Emerson und Halpern haben 1986 die „computation tree logic“ (CTL) eingeführt, die andere Eigenschaften hat. Später wurde dann CTL* als eine temporale Logik eingeführt, die LTL und CTL umfasst.

3.3.1 Syntax von CTL-Formeln

CTL besitzt zusätzlich *Zustandsformeln* mit Quantoren, die sich auf die von einem Zustand ausgehenden Pfade beziehen:

- Ap bedeutet: „Für alle Pfade, die von einem gegebenen Zustand s ausgehen, gilt die Formel p .“
- Ep bedeutet: „Es gibt einen Pfad, der von einem gegebenen Zustand s ausgeht und für den die Formel p gilt.“

Als gegebener Zustand fungiert dabei ein Anfangszustand oder ein Zustand, der durch eine umgebende Formel spezifiziert ist.

Um eine bessere Komplexität der Model-Check-Algorithmen zu erhalten, werden diese mit den bisher betrachteten Operatoren zu Formeln der Form EGf , $E[fUg]$ usw. verbunden. Dadurch ist folgende Syntaxdefinition motiviert.

Definition 3.8 Es sei AP eine Menge von aussagenlogischen Atomen.

Dann wird die Syntax von CTL-Zustands-Formeln wie folgt durch Backus-Naur-Form definiert, wobei $p \in AP$ und f eine CTL-Pfad-Formel ist:

$$g ::= \text{true} | \text{false} | p | (\neg g) | (g \wedge g) | (g \vee g) | (Ef) | (Af)$$

Eine CTL-Pfad-Formel wird durch

$$f ::= (Xg) | (Fg) | (Gg) | (g_1 U g_2)$$

definiert. Dabei sind g, g_1, g_2 CTL-Zustands-Formeln.

Beispiel 3.9

- $EF(Start \wedge \neg Ready)$
Es ist möglich in einen Zustand zu kommen, in dem „*Start*“ aber nicht „*Ready*“ gilt.
- $AG(Req \rightarrow AF Ack)$
Immer wenn ein Request *Req* erfolgt, dann wird er später einmal mit *Ack* bestätigt.
- $AG(AF DeviceEnabled)$
Die Aussage „*DeviceEnabled*“ gilt unendlich oft auf jedem Pfad.
- $AG(EF Restart)$
Von jedem Zustand aus ist es möglich, einen Zustand mit „*Restart*“ zu erreichen.

Die Logik CTL* trennt bei der Definition der Formeln nicht mehr zwischen Pfad- und Zustands-Formeln. Somit ist CTL syntaktisch eine Teillogik von CTL*. Aber auch LTL ist syntaktisch eine Teillogik von CTL*, nämlich indem wir jede LTL-Formel f als die CTL*-Formel Af auffassen.

3.3.2 Semantik von CTL-Formeln

Die Bedeutung von CTL-Formeln wird gerne als Beschreibung von Eigenschaften eines *Berechnungsbaumes* (computation tree) formuliert. Letzterer ist eine schleifenfreie Darstellung des Systemverhaltens. Berechnungsbäume entstehen durch „Abwickeln“ der Kripke-Struktur-Beschreibung des Systemverhaltens wie in Abbildung 3.2.

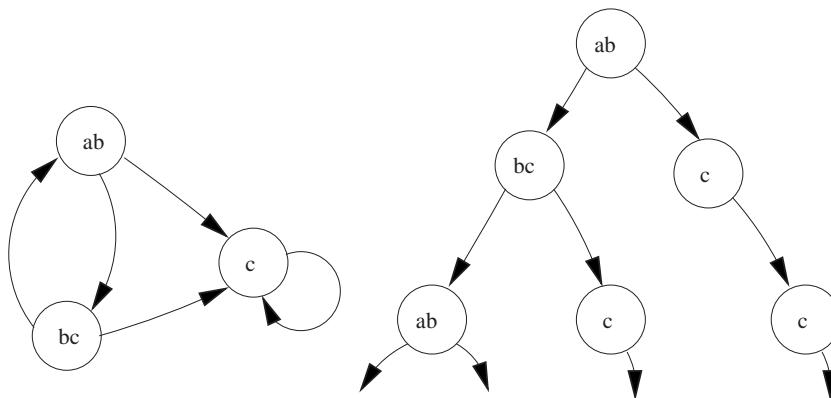


Abbildung 3.2: Abwicklung einer Kripke-Struktur

Definition 3.10 Sei $M := (S, S_0, R, E_S)$ eine Kripke-Struktur und $s \in S$ ein Zustand. Dann wird die Gültigkeit einer CTL-Zustandsformel wie folgt induktiv definiert:

1. $M, s \models p \Leftrightarrow p \in E_S(s).$
2. $M, s \models \neg g \Leftrightarrow M, s \not\models g.$
3. $M, s \models g_1 \vee g_2 \Leftrightarrow M, s \models g_1 \text{ oder } M, s \models g_2.$
4. $M, s \models g_1 \wedge g_2 \Leftrightarrow M, s \models g_1 \text{ und } M, s \models g_2.$
5. $M, s \models Ef \Leftrightarrow \text{Es gibt einen in } s \text{ beginnenden Pfad } \pi \text{ mit } M, \pi \models f.$
6. $M, s \models Af \Leftrightarrow \text{Für alle in } s \text{ beginnenden Pfade } \pi \text{ gilt } M, \pi \models f.$

Dabei ist $p \in AP$ und g, g_1, g_2 sind CTL-Zustandsformeln. f ist entsprechend der Syntax eine CTL-Pfad-Formel. Die dort enthaltenen Quantoren X, F, G und U werden analog wie in LTL definiert.

Satz 3.11 Es gelten die folgenden Äquivalenzen:

- $AXg \equiv \neg EX(\neg g)$,
- $EFg \equiv E(\text{true} \wedge Ug)$,
- $AGg \equiv \neg EF(\neg g)$,
- $AFg \equiv \neg EG(\neg g)$,
- $A[g_1 \wedge g_2] \equiv \neg E[\neg g_2 \wedge (\neg g_1 \wedge \neg g_2)] \wedge \neg EG \neg g_2$

Beweis: Als Übung. □

3.3.3 Reduktion der Operatoren

In CTL müssen vor den Pfad-Quantoren X, F, G, U immer Zustands-Quantoren A oder E stehen. Es gibt damit acht Kombinationen:

- AXg und EXg ,
- AFg und EFg ,
- AGg und EGg ,
- $A[g_1 \wedge g_2]$ und $E[g_1 \wedge g_2]$,

Aufgrund der Äquivalenzen aus Satz 3.11 können die 8 Kombination mittels $EXg, EGg, E[g_1 \wedge g_2]$ ausgedrückt werden.

Satz 3.12 Mit Hilfe der Operatoren EXg, EGg und $E[g_1 \wedge g_2]$ können alle CTL-Formeln ausgedrückt werden.

Ohne Beweis erwähnen wir folgende Eigenschaft, die die Trennschärfe der Temporallogik CTL charakterisiert.

Satz 3.13 Bisimilare Modelle können von keiner CTL-Formel unterschieden werden.

3.4 Ausdrucksmächtigkeit: LTL vs. CTL

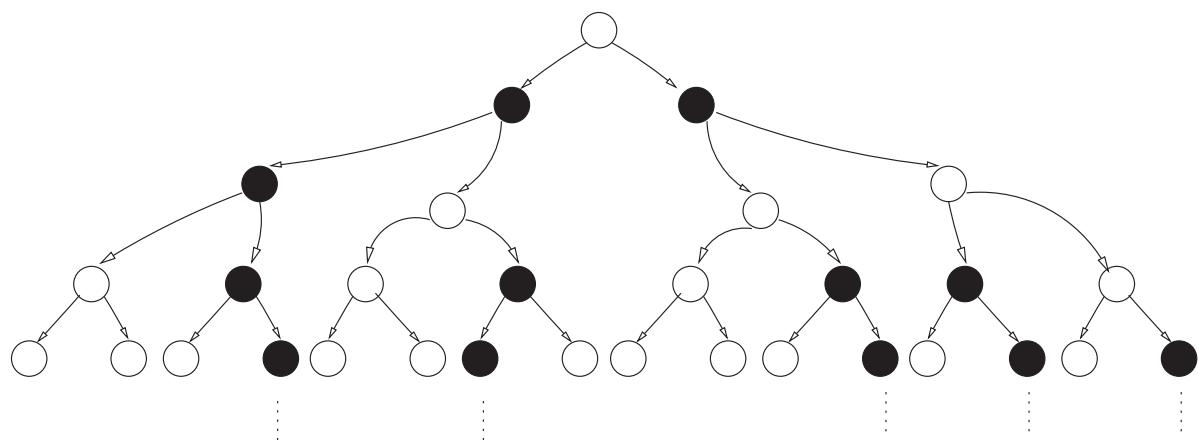
Zwei Formeln f und g heißen äquivalent (als Relationszeichen: $f \equiv g$ (siehe auch am Ende von Def. 3.5)), wenn sie durch kein Modell unterschieden werden können, d.h. wenn für jedes Modell M gilt:

$$M \models f \text{ gdw. } M \models g$$

Auch in CTL kann man nicht alle LTL-Spezifikationen ausdrücken.

Satz 3.14 Es gibt keine CTL-Formel, die äquivalent zur LTL-Formel $A(FGp)$ ist.

Die Formel $A(FGp)$ bedeutet: „Auf jedem Pfad gibt es einen Zustand, ab dem p immer gilt“.



Satz 3.15 Es gibt keine LTL-Formel, die äquivalent zur CTL-Formel $AG(EFp)$ ist.

Die Formel $AG(EFp)$ bedeutet: „Von jedem Zustand ist ein Zustand erreichbar, in dem p gilt“.

(Ist das äquivalent zu folgender Aussage: „Alle Pfade enthalten unendlich viele Zustände, in den p gilt“?)

Beweis: Wir wollen zeigen, dass es zu der CTL-Formel $\phi = AGEFp$ keine äquivalente LTL-Formel ϕ' gibt.

(1) Wir zeigen zuerst: Seien M_1 und M_2 zwei Kripke-Strukturen und ψ eine beliebige LTL-Formel. Wenn $E_1(SS(M_1)) \subseteq E_2(SS(M_2))$ gilt, dann folgt aus $M_2 \models \psi$ auch $M_1 \models \psi$. Es gilt:

$$\begin{aligned} M &\models \psi \\ \iff \forall \pi \in SS(M) : M, \pi &\models \psi \\ \iff \forall \pi \in SS(M) : E(\pi) &\models \psi \\ \iff E(SS(M)) &\subseteq L^\omega(\psi) \end{aligned}$$

Sei $M_2 \models \psi$, d.h. $E_2(SS(M_2)) \subseteq L^\omega(\psi)$. Dann ist wegen der Inklusion auch $E_1(SS(M_1)) \subseteq E_2(SS(M_2))$ auch $E_1(SS(M_1)) \subseteq E_2(SS(M_2)) \subseteq L^\omega(\psi)$ und damit gilt dann $M_1 \models \psi$.

(2) Sei nun M die Kripke-Struktur mit $S = \{s_1, s_2\}$ und den Übergängen $s_1 \rightarrow s_1$, $s_1 \rightarrow s_2$ und $s_2 \rightarrow s_2$ sowie $S^0 = \{s_1\}$. Die Zustandsetiketten sind $E(s_1) = \emptyset$ und $E(s_2) = \{p\}$. Es gilt: $M \models \phi$ für $\phi = AGEFp$.

(3) Sei M' die Einschränkung der Kripke-Struktur M auf s_1 , d.h. $S' = \{s_1\}$, $s_1 \rightarrow s_1$ und $E'(s_1) = \emptyset$. Es gilt: $M' \not\models \phi$ für $\phi = AGEFp$, denn M' besitzt nur noch den Pfad $\pi = s_1^\omega$ und es folgt $E'(\pi) = \emptyset^\omega$. Hier gilt p also nie.

Damit zeigen wir: Es gibt keine zu der CTL-Formel $\phi = AGEFp$ äquivalente LTL-Formel ϕ' .

Angenommen ϕ' wäre eine zu der CTL-Formel $\phi = AGEFp$ äquivalente LTL-Formel.

Aus der Äquivalenz folgt, dass ϕ' in M gilt, da ϕ – nach (2.) – in M gilt.

Da die Etiketten in M' und M identisch sind, ist jede ω -Folge $E(\pi)$ von M' auch eine von M . Also gilt nach (1.) auch ϕ' in M' .

Aus der Äquivalenz folgt nun, dass auch ϕ in M' – Widerspruch zu (3.).

Also gibt es keine solche äquivalente LTL-Formel ϕ' . \square

CTL^* ist eine temporale Logik, die syntaktisch beide Logiken LTL und CTL umfasst. Es gibt aber auch Formeln in CTL^* , z.B. $A(FGp) \vee AG(EFp)$, die weder in CTL noch in LTL ausdrückbar sind, wodurch CTL^* die Logiken LTL und CTL auch semantisch echt enthält (vgl. Abb. 3.3)

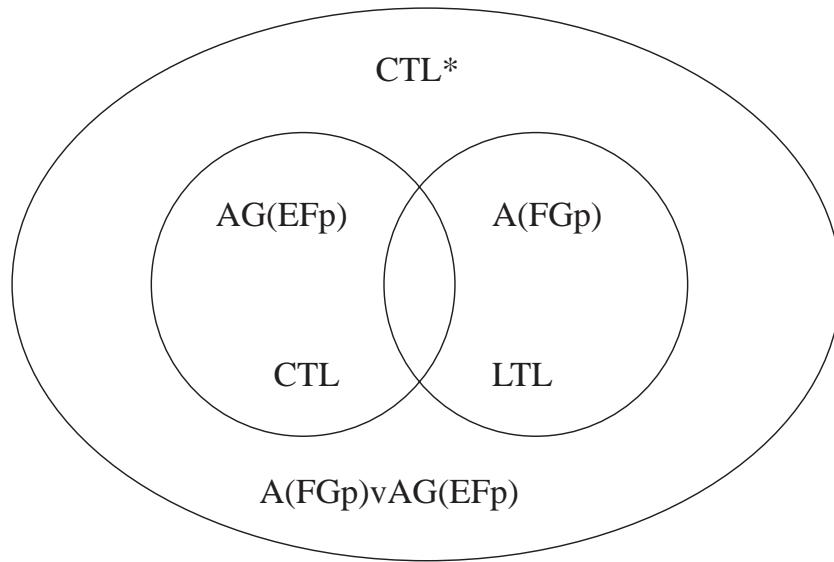
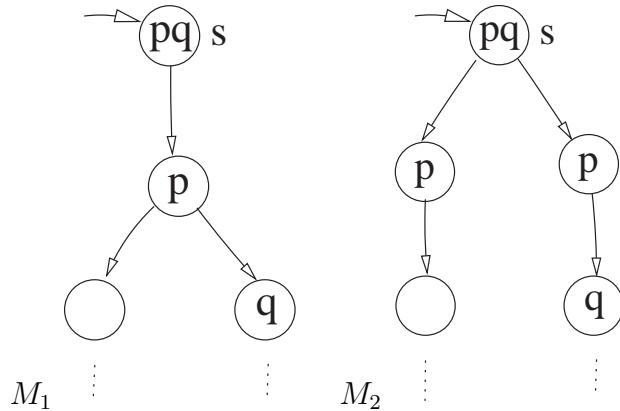


Abbildung 3.3: Beziehungen zwischen den Temporallogiken

Aufgabe 3.16 Gegeben sind die folgenden Kripke-Strukturen M_1 und M_2 :



- Gibt es Formeln in LTL , die die Strukturen unterscheiden, d.h. nur in einem Modell gelten?
- Das gleiche für CTL .

4 Model-Checking

Wir schränken uns in der folgenden Darstellung auf endliche Zustandsräume ein, da diese der algorithmischen Analyse (im Rahmen des Model-Checking) zugänglicher sind.

4.1 Automatentheorie und LTL-Model-Checking

Um Informatik-Systeme auf ihre Korrektheit zu prüfen, muss ihr Verhalten mit einer Spezifikation des Systems verglichen werden. Häufig geschieht dies aus Kosten- oder anderen Gründen weniger formal. Ist die Korrektheit eines Systems besonders wichtig, weil ein Fehlverhalten unverantwortlich oder zu teuer ist, werden Verifikationsmethoden angewandt. Eine davon ist das *Model-Checking*.

4.1.1 Model-Checking bei explizit definierter Spezifikation

Algorithmus 4.1 Model-Checking für FA

```
PROCEDURE Check( $L(TS_{sys})$ ,  $L(TS_{spec})$ )
    construct complement  $\overline{TS_{spec}}$  with  $L(\overline{TS_{spec}}) = (A^* \setminus L(TS_{spec}))$ 
    construct product  $TS_{\cap}$  with  $L(TS_{\cap}) = L(TS_{sys}) \cap L(\overline{TS_{spec}})$ 
    IF  $L(TS_{\cap}) = \emptyset$  THEN return true
    ELSE return false
END PROCEDURE
```

Bei dieser Methode wird das *System* durch ein Transitionssystem TS_{sys} modelliert und die *Spezifikation* durch ein Transitionssystem TS_{spec} . Zu prüfen ist dann, ob alle Transitionsfolgen des Systems auch solche der Spezifikation sind, d.h.

$$L(TS_{sys}) \subseteq L(TS_{spec})$$

Um dies zu prüfen, wird die für alle Mengen $P \subseteq R$ und $Q \subseteq R$ Beziehung genutzt:

$$P \subseteq Q \iff P \cap (R \setminus Q) = \emptyset$$

Wenn A die Grundmenge der Aktionen ist, bleibt also zu prüfen, ob folgendes gilt:

$$L(TS_{sys}) \cap (A^* \setminus L(TS_{spec})) = \emptyset$$

Dazu wird in Algorithmus 4.1 ein Transitionssystem TS_{\cap} konstruiert, welches diesen Durchschnitt akzeptiert.

Dieses Vorgehen hat folgende Vorteile:

1. Wie gezeigt ist ein den Durchschnitt akzeptierendes Transitionssystem mithilfe des Produkttransitionssystems relativ einfach zu konstruieren (in quadratischer Zeit).
2. In linearer Zeit (in bezug auf die Größe des Transitionssystems) kann geprüft werden, ob die akzeptierte Sprache von TS_{\cap} leer ist.
3. Oft ist das Transitionssystem (der endliche Automat) TS_{spec} deterministisch. Um das Komplement $A^* \setminus L(TS_{spec})$ zu akzeptieren, muss dann nur im vervollständigten Automaten die Endzustandsmenge komplementiert werden.
4. Ist der Durchschnitt nicht leer, dann können alle akzeptierten Folgen dieses Durchschnittes als Beispielablauf für Verletzungen der Spezifikation benutzt werden. Dies ist für die Korrektur des Systems von großem Nutzen.

Beispiel 4.1 (Model-Checking) Abbildung 4.1 zeigt das externe Verhalten des gestörten Sender-Empfänger-Systems von Abb. 2.4 als Transitionssystem TS_{sys} (in Abb. 2.4 als $(S \otimes R)_{extern}$ bezeichnet) und die Spezifikation TS_{spec} des gewünschten korrekten Verhaltens. Dabei wurde einschränkend die Annahme gemacht, dass ein Datum über den Kanal A nicht vor der Abgabe des vorangehenden über den Kanal C eingeht. Dies ist eine realistische Annahme, wie sie häufig bei Protokollen (z.B. dem Alternierbitprotokoll) vorliegt.

Das Transitionssystem \overline{TS}_{spec} in der Mitte rechts akzeptiert $A^* \setminus L(TS_{spec})$. Es ist aus TS_{spec} konstruiert worden, indem es zunächst in Bezug auf die Aktionenmenge $A = \{r_A(d), s_C(d), s_C(\perp)\}$ vollständig gemacht wurde (die Schleife in v_2 gilt in Bezug auf alle Aktionen von A). Danach wurde das Komplement $\{v_1, v_2\}$ als neue Menge von Endzuständen gewählt. Aus TS_{sys} und \overline{TS}_{spec} wurde dann TS_{\cap} konstruiert, das den Durchschnitt akzeptiert. $L(TS_{\cap})$ ist nicht leer, da z.B. die Folgen $r_A(d)s_C(\perp)$ oder $r_A(d)s_C(\perp)r_A(d)s_C(d)$ enthalten sind. Das System ist also - wie erwartet - nicht korrekt. Die beiden Folgen können bei Tests zum Auffinden des Fehlers benutzt werden.

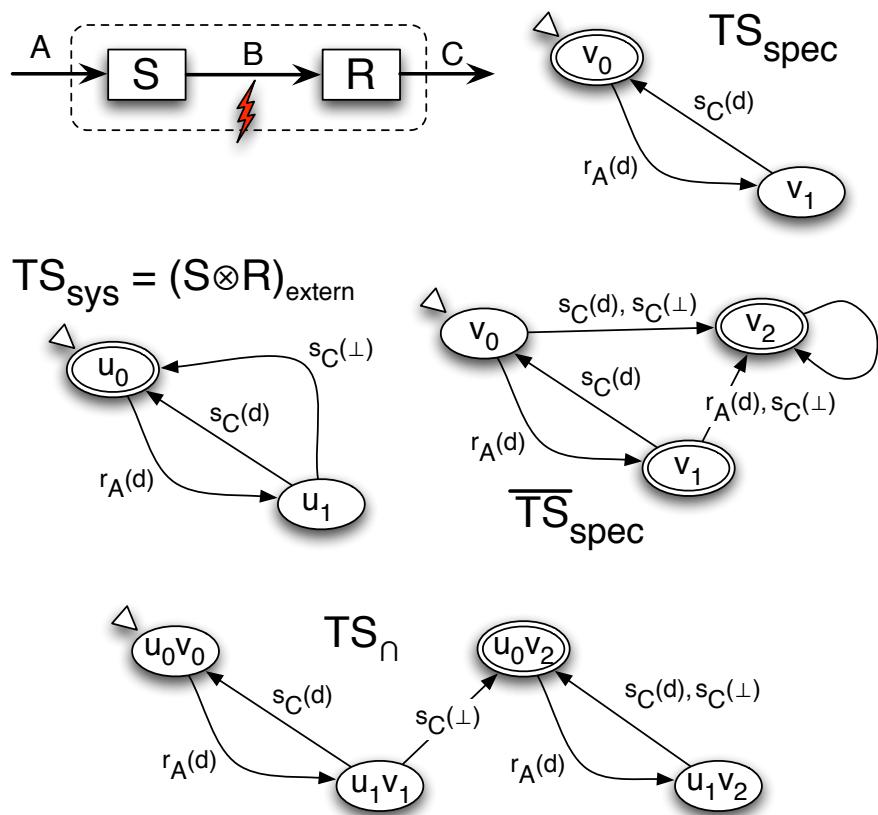


Abbildung 4.1: Model-Checking für das modifizierte gestörte Sender-Empfänger-System.

4.1.2 Model-Checking bei LTL-Spezifikationen

Ein für das Model-Checking fundamentaler Satz sagt, dass zu jeder LTL-Formel eine ihre Sprache akzeptierende Kripke-Struktur in Form eines *Büchi-Automaten* existiert.

Satz 4.2 Zu jeder LTL-Formel f kann eine Kripke-Struktur (ein Büchi-Automat) M konstruiert werden, die genau die für f gültigen Folgen akzeptiert: $L^\omega(M) = L^\omega(f)$.

Die Zeit- und Platz-Komplexität dieses Algorithmus ist $2^{\mathcal{O}(|f|)}$.

Diese obere Schranke kann nicht wesentlich verbessert werden, denn es kann eine Folge $\alpha_1, \alpha_2, \alpha_3, \dots$ von LTL-Formeln derart konstruiert werden, dass die Größe von α_i durch ein Polynom n -ten Grades beschränkt wird, die entsprechenden Kripke-Strukturen aber mindestens 2^n Zustände haben.

Dieses Ergebnis ähnelt demjenigen von Satz 7.27 auf Seite 132. Beim Model-Checking ist die Formel f_{spec} einer System-Spezifikation meist jedoch klein gegenüber der Größe des System-Transitionssystems, so dass letzteres die Komplexität des Verfahrens dominiert.

Beispiel 4.3 Für einfache LTL-Formeln f kann man die ω -reguläre Sprache $L^\omega(f)$ auch direkt angeben. Sei $p \in AP$.

LTL-Formel $f = \Box p$:

$$L^\omega(\Box p) = \left(\sum_{M \ni p, M \subseteq AP} M \right)^\omega$$

LTL-Formel $f = \Diamond p$:

$$L^\omega(\Diamond p) = \mathcal{P}(AP)^* \cdot \left(\sum_{M \ni p, M \subseteq AP} M \right) \cdot \mathcal{P}(AP)^\omega$$

Wird die Spezifikation durch eine (temporal-)logische Formel f_{spec} festgelegt, so wandeln wir diese dann in ein äquivalentes Transitionssystem TS_{spec} um (siehe Satz 4.2).

Zu prüfen ist dann analog:

$$L^\omega(TS_{sys}) \subseteq L^\omega(TS_{spec}).$$

Um dies nachzuweisen, formen wir wieder um:

$$L^\omega(TS_{sys}) \cap (A^\omega \setminus L^\omega(TS_{spec})) = \emptyset$$

Eine analoge Formulierung von Algorithmus 4.1 ist in Algorithmus 4.2 dargestellt.

Algorithmus 4.2 ist zwar effektiv (weil alle benutzten Konstruktion für Büchi-Automaten – Komplementbildung, Produktbildung, Test auf Leerheit – dies sind), es ist aber nicht sehr effizient, da die Komplementbildung für Büchi-Automaten eine enorme Zeitkomplexität aufweist: 2^{n^2} , wobei n die Anzahl der Zustände des Ausgangsautomaten ist.

Es gibt aber einen effizienteren Ansatz: Ist die Spezifikation durch eine Formel f_{spec} gegeben, dann kann man einfach zur Negation $\neg f_{spec}$ übergehen und dann zu dieser Formel

Algorithmus 4.2 LTL Model-Checking, erster Ansatz

```

PROCEDURE Check( $L(TS_{sys}), L(TS_{spec})$ )
    construct complement  $\overline{TS}_{spec}$  with  $L^\omega(\overline{TS}_{spec}) = (A^\omega \setminus L^\omega(TS_{spec}))$ 
    construct product  $TS_4$  with  $L^\omega(TS_\cap) = L^\omega(TS_{sys}) \cap L^\omega(\overline{TS}_{spec})$ 
    IF  $L^\omega(TS_\cap) = \emptyset$  THEN return true
    ELSE return false
END PROCEDURE

```

Algorithmus 4.3 LTL Model-Checking, effizientere Variante

```

PROCEDURE Check( $L(TS_{sys}), f$ )
    construct  $TS_{\neg f_{spec}}$ 
    construct product  $TS_\cap$  with  $L^\omega(TS_\cap) = L^\omega(TS_{sys}) \cap L^\omega(TS_{\neg f_{spec}})$ 
    IF  $L^\omega(TS_\cap) = \emptyset$  THEN return true
    ELSE return false
END PROCEDURE

```

den Büchi-Automaten $TS_{\neg f_{spec}}$ konstruieren, der dann $L^\omega(TS_{\neg f_{spec}}) = A^\omega \setminus L^\omega(TS_{f_{spec}})$ akzeptiert, also bereits das gewünschte Komplement (vgl. Algorithmus 4.3).

Wie groß ist also die Komplexität des Verfahrens?

Wir wissen bereits, dass zu jeder LTL-Formel f ein äquivalenter Büchi-Automat M konstruiert werden kann und dass die Zeit- und Platz-Komplexität dieses Algorithmus $2^{\mathcal{O}(|f|)}$ ist, da der Automat $2^{\mathcal{O}(|f|)}$ Zustände besitzt. Außerdem wissen wir, dass der Produktautomat A_4 Tripel als Zustände besitzt, also hier $2 \cdot |M| \cdot |TS|$ viele. Um $L^\omega(TS_\cap) = \emptyset$ festzustellen, müssen wir feststellen, ob ein erreichbarer Zustand auf einem Zyklus liegt. Dies braucht höchstens soviel Zeit und Platz, wie der Produktautomat A_4 Zustände besitzt: $\mathcal{O}(|TS| \cdot |M|)$. Insgesamt erhalten wir $\mathcal{O}(|TS| \cdot 2^{\mathcal{O}(|f|)})$ als Gesamtkomplexität.

4.1.3 Beispiel: Zustandsraumanalyse mit Maude

Das Werkzeug MAUDE erlaubt eine Analyse temporallogischer Formeln [EMS02]. Dabei kommt speziell die Logik LTL zum Einsatz. Die temporallogischen Operatoren für den LTL-Analysator werden durch das Modul MODEL-CHECKER in MAUDE-Syntax definiert.

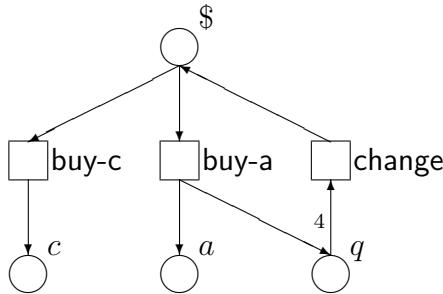


Abbildung 4.2: Ein Beispielnetz

Betrachten wir die Spezifikation des Beispielnetzes aus Abb. 4.2. Es besitzt folgende Entsprechung in der Spezifikationssprache MAUDE [CDE⁺99]:

```

mod CANDY-PN is
  sorts Place Marking .
  subsort Place < Marking .
  op _ _ : Marking Marking -> Marking [assoc comm] .
  ops $ q a c : -> Place .

  rl [buy-candy] : $ => c .
  rl [buy-apple] : $ => a q .
  rl [change] : q q q q => $ .
endm
  
```

Es existieren die Sorten der Stellen (`Place`) und der Markierungen (`Marking`), wobei `Place` eine Subsorte von `Marking` ist. Der Multimengenoperator `+` auf Markierungen wird abkürzend als Operator ohne Symbol notiert (`op _ _`). Die konkreten Stellen `$`, `q`, `a`, `c` sind Konstanten vom Typ `Place`. Die Transitionen sind als Ersetzungsregeln (`rl` [`buy-c`] usw.) beschrieben.

Das folgende Modul definiert die für eine Zustandsraumanalyse notwendigen Prädikate:

```

mod CHECK-CANDY is
  inc CANDY-PN .
  inc MODEL-CHECKER .

  subsort Marking < State .
  ops p-candy p-apple : -> Prop .
  var M : Marking .
  
```

```

eq ((M c) |= p-candy) = true .
eq ((M a) |= p-apple) = true .

op phi1 : -> Prop .
eq phi1 = <> (p-candy ∨ p-apple) .

op phi2 : -> Prop .
eq phi2 = <> p-candy .
endm
    
```

Importiert wird neben dem Modell CANDY-PN des Petrinetzes auch die Spezifikation MODEL-CHECKER. Durch `subsort Marking < State` wird definiert, dass Markierungen des Petrinetzes die Zustände der Kripke-Struktur sind. Die Übergänge werden automatisch durch die Ersetzungsregeln erzeugt. Es werden die beiden atomaren Propositionen `p-candy` und `p-apple` definiert. Dabei gilt `p-candy`, wenn in der Markierung mindestens eine Süßigkeit, d.h. eine Marke `c` vorhanden ist. Dann muss die Markierung die Form $M + c$ besitzt. Dies wird durch die Gleichung `eq ((M c) |= p-candy) = true` ausgedrückt, wobei `|=` ein Prädikat ist, das Zustände (`State`) mit Propositionen (`Prop`) in Relation setzt. Analog gilt `p-apple`, wenn in der Markierung M mindestens eine Marke `a` vorhanden ist.

Die Formel $\phi_1 = \Diamond(p\text{-candy} \vee p\text{-apple})$ drückt aus, dass irgendwann eine Marke `c` (eine Süßigkeit) oder eine Marke `a` (ein Apfel) vorhanden sein wird. Sie wird in MAUDE durch die Gleichung

$$\text{eq } \phi_1 = <> (\text{p-candy} \vee \text{p-apple})$$

notiert. Die Formel $\phi_2 = <> \text{p-candy}$ beschreibt, dass irgendwann eine Süßigkeit vorhanden sein wird.

Um zu überprüfen, ob eine Formel ϕ gilt, muss sich die Relation $M_0 \models \phi$ zu „true“ auswerten lassen. Die Gültigkeit für die Formel ϕ_1 in der Initialmarkierung $M_0 = \$+\$$ ergibt mit der folgenden Ersetzung:

```

Maude> rew [10] \$ \$ |= phi1 .
rewrite [10] in CHECK-CANDY : \$ \$ |= phi1 .
rewrites: 11 in 10ms cpu (7ms real) (1100 rewrites/second)
result Bool: true
    
```

Die Formel ϕ_2 ist nicht gültig, was man an der Ausführungsfolge, die mit $w_1 = \text{buy-apple} \cdot \text{buy-apple}$ beginnt, erkennt. Dieses Gegenbeispiel konstruiert auch MAUDE in Folge der Analyse:

```

Maude> rew [10] \$ \$ |= phi2 .
rewrite [10] in CHECK-CANDY : \$ \$ |= phi2 .
rewrites: 15 in 0ms cpu (6ms real) (~ rewrites/second)
result ModelCheckResult:
  counterexample({c a q, 'buy-apple}
  {c a q, 'buy-apple},
  {a a q q, deadlock})
    
```

4.2 CTL-Model-Checking

Die *CTL-Model-Checking* Aufgabe lautet:

- Berechne für eine gegebene (endliche) Kripke-Struktur $M := (S, S_0, R, E_S)$ und eine gegebene CTL-Formel f die Menge:

$$Sat(f) := \{s \in S \mid M, s \models f\}$$

Der Algorithmus 4.4 erweitert $E_S(s)$ für alle $s \in S$ schrittweise zu $label(s)$.

Dadurch enthält $label(s)$ alle Teilformeln von f , die in s wahr sind.

Durch Rekursion über die Schachtelungstiefe von f gilt in Schritt i : alle Teilformeln mit $i - 1$ geschachtelten CTL-Operatoren sind behandelt.

Da alle CTL-Operatoren nach Satz 3.11 mittels EXg , EGg und $E[g_1Ug_2]$ ausgedrückt werden können, bleiben nur noch zwei nicht elementare Unterprozeduren: $CheckEU(f_1, f_2)$ und $CheckEG(f_1)$.

Algorithmus 4.4

```

PROCEDURE Check(f)
  IF  $f \in AP$  THEN
    FORALL  $s \in S$  SUCH THAT  $f \in E_S(s)$  DO  $label(s) := label(s) \cup \{f\}$ ;
  IF  $f = \neg f_1$  THEN
    Check( $f_1$ );
    FORALL  $s \in S$  SUCH THAT  $f_1 \notin label(s)$  DO  $label(s) := label(s) \cup \{f\}$ ;
  IF  $f = f_1 \vee f_2$  THEN
    Check( $f_1$ ); Check( $f_2$ );
    FORALL  $s \in S$  SUCH THAT  $f_1 \in label(s) \vee f_2 \in label(s)$  DO  $label(s) := label(s) \cup \{f\}$ ;
  IF  $f = EXf_1$  THEN
    Check( $f_1$ );
    FORALL  $s \in S$  SUCH THAT  $R(s, t) \wedge f_1 \in label(t)$  DO  $label(s) := label(s) \cup \{f\}$ ;
  IF  $f = E[f_1Uf_2]$  THEN
    Check( $f_1$ ); Check( $f_2$ ); CheckEU( $f_1, f_2$ )
  IF  $f = EGf_1$  THEN
    Check( $f_1$ ); CheckEG( $f_1$ )
END PROCEDURE

```

4.2.1 Die Unterprozedur $CheckEU(f_1, f_2)$

Zunächst markieren wir alle Zustände s , die f_2 erfüllen. Dann fahren wir schrittweise in Gegenrichtung der Transitionen fort und markieren die Zustände, die f_1 erfüllen (vgl. Algorithmus 4.5). Dadurch markieren wir einen Zustand s mit f , falls es einen Pfad von s zu einem s' mit $f_2 \in label(s')$ gibt, so dass für alle Zustände t davor $f_1 \in label(t)$ gilt.

4.2.2 Die Unterprozedur $CheckEG(f_1)$

Für diese Prozedur benötigen wir den Begriff der strengen Zusammenhangskomponente, denn eine Formel f_1 gilt ja nur dann, wenn es einen Kreis in der Kripke-Struktur gibt, auf dem stets f_1 gilt.

Algorithmus 4.5 Auszeichnen mit $E(f_1 U f_2)$

```

PROCEDURE CheckEU( $f_1, f_2$ )
     $T := \{s \mid f_2 \in \text{label}(s)\}$ ;
    FORALL  $s \in T$  DO  $\text{label}(s) := \text{label}(s) \cup \{E[f_1 U f_2]\}$ ;
    WHILE  $T \neq \emptyset$  DO
        CHOOSE  $s \in T$ ;
         $T := T \setminus \{s\}$ ;
        FORALL  $t$  SUCH THAT  $R(t, s)$  DO
            IF  $E[f_1 U f_2] \notin \text{label}(t)$  AND  $f_1 \in \text{label}(t)$  THEN
                 $\text{label}(t) := \text{label}(t) \cup \{E[f_1 U f_2]\}$ ;
                 $T := T \cup \{t\}$ ;
            END IF ;
        END FORALL ;
    END WHILE ;
END PROCEDURE ;

```

Definition 4.4 Sei $G = (K, R)$ ein gerichteter Graph, d.h.: $R \subseteq K \times K$:

- a) Eine Knotenmenge $A \subseteq K$ heißt Zusammenhangskomponente, falls: $\forall a, a' \in A : aR^*a'$.
- b) $A \subseteq K$ heißt strenge Zusammenhangskomponente (SZK) (*strongly connected component: SZK*), falls sie maximal ist, d.h.: $\neg \exists k \in K \setminus A : \forall a \in A : kR^*a \wedge aR^*k$.
- c) Sie heißt nichttriviale Zusammenhangskomponente, falls sie mehr als einen Knoten enthält oder eine Schleife: $|A| > 1$ oder $\exists a \in A : aR^+a$.

Nun betrachten wir wieder die Formel: $f = EGf_1$:

Sei $M = (S, S_0, R, \text{label})$ die im hier entwickelten CTL-Algorithmus jeweils durch die Abbildung label erweiterte Kripke-Struktur. Daraus konstruieren wir $M' = (S', S'_0, R', \text{label}')$ mit:

$$\begin{aligned}
 S' &:= \{s \in S \mid M, s \models f_1\} \\
 S'_0 &:= S_0 \cap S' \\
 R' &:= R|_{S' \times S'} \\
 \text{label}' &:= \text{label}|_{S'}
 \end{aligned}$$

d.h. die „Einschränkung“ von M auf Zustände, in denen f_1 gilt. Es werden also alle Zustände und anhängende Kanten gestrichen, in denen f_1 nicht enthalten ist.

Lemma 4.5 Es gilt genau dann $M, s \models EGf_1$, wenn (1.) $s \in S'$ und (2.) es einen Pfad in M' gibt, der von s zu einer nichttrivialen strengen Zusammenhangskomponente in (S', R') führt.

Beweis: Als Übung. □

Daraus resultiert folgender Algorithmus zur Entscheidung von EGf_1 (vgl. Algorithmus 4.6):

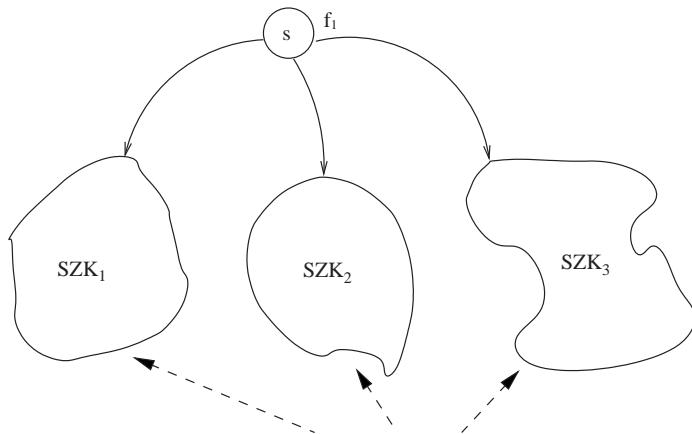


Abbildung 4.3: Strenge Zusammenhangskomponenten mit f_1

Algorithmus 4.6 Auszeichnen mit EGf_1

```

PROCEDURE CheckEG( $f_1$ )
   $S' := \{s \mid f_1 \in \text{label}(s)\};$ 
   $SCC := \{C \mid C \text{ a nontrivial SCC of } S'\};$ 
   $T := \bigcup_{C \in SCC} \{s \mid s \in C\};$ 
  FORALL  $s \in T$  DO  $\text{label}(s) := \text{label}(s) \cup \{EGf_1\};$ 
  WHILE  $T \neq \emptyset$  DO
    CHOOSE  $s \in T;$ 
     $T := T \setminus \{s\};$ 
    FORALL  $t$  SUCH THAT  $t \in S'$  AND  $R(t, s)$  DO
      IF  $EGf_1 \notin \text{label}(t)$  THEN
         $\text{label}(t) := \text{label}(t) \cup \{EGf_1\};$ 
         $T := T \cup \{t\};$ 
      END IF ;
    END FORALL ;
  END WHILE ;
END PROCEDURE ;

```

1. Konstruiere $M' = (S', S'_0, R', \text{label}')$.
2. Konstruiere alle SZK von M' . (Algorithmus von Tarjan mit $O(|S'| + |R'|)$ Zeitkomplexität.)
3. Finde Zustände in nichttrivialen SZK.
4. Suche von diesen aus rückwärts alle Zustände, die dorthin führen. (Das sind maximal $\mathcal{O}(|S| + |R|)$ viele.)

Satz 4.6 Der Algorithmus 4.4, der für eine Kripke-Struktur $M := (S, S_0, R, E_S)$ und eine CTL-Formel f entscheidet, ob f für M gilt, d.h. ob $M \models f$, besitzt die Zeitkomplexität $\mathcal{O}(|f| \cdot (|S| + |R|))$.

Beweis: Wendet man Algorithmus 4.4 auf eine Kripke-Struktur an, dann wird für jede Teilformel von f eine Subprozedur gestartet. Die Anzahl der Teilformeln ist durch $\mathcal{O}(|f|)$ begrenzt.

Für jede Teilformel benötigt die jeweilige Subprozedur maximal $\mathcal{O}(|S| + |R|)$ viele Operationen, denn die Suchverfahren für $E[-U-]$ besuchen i.w. jeden Knoten einmal und die Berechnung der SCCs für $EG-$ geschieht auch in Linearzeit. Insgesamt gibt es also maximal $\mathcal{O}(|f| \cdot (|S| + |R|))$ Operationen. \square

Beispiel 4.7 (MW-Ofen) CTL-Spezifikation zu der Kripke-Struktur in Abb. 3.1:

$$f = AG(Start \rightarrow AF Heat)$$

Es gilt immer: nach einem Zustand mit „Start“ wird später ein Zustand mit „Heat“ erreicht.

Zunächst muss f mit Hilfe der Äquivalenzen von Satz 3.11 so umgeschrieben werden, dass nur noch die Operationen enthalten sind, die der Algorithmus verarbeiten kann. Dabei steht wegen der besseren Lesbarkeit Fg für **true** $U g$.

$$\begin{aligned} AGf &\Leftrightarrow \neg EF(\neg f) \Leftrightarrow \neg EF(\neg(\neg Start \vee AF Heat)) \\ &\Leftrightarrow \neg EF(Start \wedge \neg AF Heat) \quad (AFf \Leftrightarrow \neg EG(\neg f)) \\ &\Leftrightarrow \neg EF(Start \wedge EG\neg Heat) \end{aligned}$$

Bezeichnet $Sat(Start) = \{2, 5, 6, 7\}$ die Menge der Zustände, in denen $Start$ gilt und entsprechend $Sat(\neg Heat) = \{1, 2, 3, 5, 6\}$, dann ist, um $f = EG\neg Heat$ zu behandeln, $\{1, 2, 3, 5\}$ die einzige SZK von $Sat(\neg Heat)$, d.h. der Zustand 6 wird ausgeschlossen. Zustände, die mit $EG\neg Heat$ zu markieren sind also $T = \{1, 2, 3, 5\} = Sat(EG\neg Heat)$. Es folgt:

$$Sat(Start \wedge EG\neg Heat) = \{2, 5, 6, 7\} \cap \{1, 2, 3, 5\} = \{2, 5\}$$

und durch Rückwärtspfade:

$$Sat(EF(Start \wedge EG\neg Heat)) = \{1, 2, 3, 4, 5, 6, 7\}.$$

Damit ergibt sich schließlich:

$$Sat(\neg EF(Start \wedge EG\neg Heat)) = \emptyset$$

und folglich $M, 1 \not\models AG(Start \rightarrow AF Heat)$ d.h. die Spezifikation f gilt *nicht* im Anfangszustand.

4.3 CTL-Model-Checking mit Fairness

Fairness-Spezifikationen sind für viele Anwendungen wichtig. Sie lassen sich oft in LTL ausdrücken, aber nicht in CTL, was wegen der besseren Komplexitätseigenschaften wünschenswert wäre.

Hier zwei Beispiele:

- „Eine Alternative einer sich ständig wiederholenden Alternative wird irgendwann einmal auch gewählt“ z.B. Hardware Arbiter.
- „Ein gestörter Kanal übermittelt immer wieder einmal eine Nachricht korrekt“ z.B Alternierbitprotokoll.

Eine Lösung dieses Problems besteht darin, dass die Fairness-Spezifikation auf die Kripke-Struktur verlagert wird (man spricht auch von „fairer Semantik“). Da Fairness-Bedingungen durch Endzustände in Transitionssystemen darstellbar sind, führt man Endzustände für Kripke-Strukturen ein und verlagert die Fairness-Spezifikation von der temporallogischen Formel auf das Systemtransitionssystem, also von f_{spec} auf TS_{sys} . Es wird dann CTL-Model-Checking auf die akzeptierten unendlichen Folgen angewandt, anstatt auf alle möglichen Folgen von TS_{sys} . Da eine Endzustandsmenge nur eine einzige Fairness-Spezifikation ausdrücken kann, benötigt man für mehrere solche Spezifikationen mehrere Endzustandsmengen. Dieses Modell heißt „*faire Kripke-Struktur*“ oder „*verallgemeinerter Büchi-Automat*“.

Definition 4.8 Eine faire Kripke-Struktur

$$M := (S, S_0, R, E_S, \{E_F^1, \dots, E_F^k\})$$

besteht aus einer Kripke-Struktur $M := (S, S_0, R, E_S)$ und den $k \geq 1$ Endzustandsmengen $E_F^i \subseteq S$.

Ein Pfad $\pi = s_0s_1s_2\dots \in SS(M)$ heißt fair, falls $\text{infinite}(\pi) \cap E_F^i \neq \emptyset$ für alle $i \in \{1, \dots, k\}$ gilt

Fairness wird also über eine Akzeptanzbedingung definiert, die wir bereits vom Büchi-Automaten kennen (vergl. Definition 1.10 auf Seite 5).

Bezogen auf die Einschränkung auf faire Pfade spricht man von *fairer Gültigkeit* (in Zeichen: \models_F) und ändert die Bedingungen 1, 5 und 6 von Definition 3.10 auf Seite 46 wie folgt:

1. $M, s \models_F p \iff$ Es gibt einen **fairen** Pfad, der bei s anfängt mit $p \in E_S(s)$.
5. $M, s \models_F Ef \iff$ Es gibt einen in s beginnenden **fairen** Pfad π mit $M, \pi \models f$.
6. $M, s \models_F Af \iff$ Für alle in s beginnenden **fairen** Pfade π gilt $M, \pi \models f$.

Beispiel 4.9

$$E_F^i = \{s \mid s \text{ erfüllt } \neg send_i \vee receive_i \text{ für Kanal } i\}$$

Ein fairer Pfad impliziert: in jedem Kanal wird unendlich oft empfangen, falls gesendet wird.

Definition 4.10 Sei $M := (S, S_0, R, E_S, E_F^1, \dots, E_F^k)$ eine faire Kripke-Struktur. Eine starke Zusammenhangskomponente $C \subseteq S$ heißt fair, falls $\forall i \in \{1, \dots, k\} : E_F^i \cap C \neq \emptyset$. Ferner sei $M' := (S', S'_0, R', E'_S, F_1, \dots, F_k)$ mit:

$$\begin{aligned} S' &= \{s \in S \mid M, s \models_F f_1\} & (\models_F \text{ siehe Seite 62}) \\ R' &= R|_{S' \times S'} \\ E'_S &= E_S|_{S'} \\ F_i &= E_F^i \cap C \end{aligned}$$

Lemma 4.11 Es gilt genau dann $M, s \models_F EGf_1$, wenn (1.) $s \in S'$ und (2.) es einen Pfad in M' gibt, der von s zu einer fairen, nichttrivialen starken Zusammenhangskomponente in (S', R') führt.

Daraus folgt eine Prozedur *CheckFairEG(f1)* um Spezifikation in fairer Semantik zu prüfen:

$$fair := EG \text{True} \quad \text{„Es gibt eine unendliche Folge“}$$

und

$$\begin{aligned} M, s \models_F p &\iff M, s \models p \wedge fair \\ M, s \models_F EXf_1 &\iff M, s \models EX(f_1 \wedge fair) \\ M, s \models_F E[f_1 U f_2] &\iff M, s \models E[f_1 U (f_2 \wedge fair)] \end{aligned}$$

Satz 4.12 Es gibt einen Algorithmus, der für eine faire Kripke-Struktur $M := (S, S_0, R, E_S, E_F^1, \dots, E_F^k)$ und eine CTL-Formel f in $\mathcal{O}(|f| \cdot (|S| + |R|))$ Zeitkomplexität entscheidet, ob f für M in der fairen Semantik gilt, d.h. ob $M \models_F f$.

Beispiel 4.13 Prüfe $f = AG(Start \rightarrow AF Heat)$, wobei vorausgesetzt wird, dass die Benutzer den Ofen immer korrekt bedienen. Dabei interpretieren wir „immer korrekt bedienen“ als „unendlich oft gilt $Start \wedge Close \wedge \neg Error$ “.

Daher setzen wir $M := (S, S_0, R, E_S, E_F^1)$ (also $k = 1$) mit

$$E_F^1 = \{s \mid s \models Start \wedge (Close \wedge \neg Error)\} = \{6, 7\}.$$

Entsprechend $Sat(f)$ definieren wir $Sat_F(f) := \{s \in S \mid M, s \models_F f\}$. Mit $Sat_F(Start) = Sat(Start)$, $Sat_F(\neg Heat) = Sat(\neg Heat)$ wie vorher ist die ZSK $\{1, 2, 3, 5\}$ nicht fair, da sie disjunkt zu $E_F^1 = \{6, 7\}$ ist. Also:

$$\begin{aligned} Sat_F(EG \neg Heat) &= \emptyset \\ Sat_F(EF(Start \wedge EG \neg Heat)) &= \emptyset \\ Sat_F(\neg EF(Start \wedge EG \neg Heat)) &= \{1, \dots, 7\} \end{aligned}$$

Die Spezifikation ist in der fairen Semantik erfüllt, da die Formel f im Anfangszustand gilt: $M, 1 \models_F AG(Start \rightarrow AF Heat)$.

Aufgabe 4.14 (CTL-Model-Checking) Prüfen Sie die folgende Spezifikation für das Ofenbeispiel durch den CTL-Algorithmus: $AG(Start \wedge \neg Close \wedge \neg Heat \wedge Error \Rightarrow EF \neg Error)$.

5 Halbordnungssemantik (Partial Order Semantics)

Während Prozesse von sequentiellen Prozessen durch eine totale (auch *linear* genannte) Ordnung gekennzeichnet sind, tritt für nebenläufige Prozesse (wie sie beispielsweise durch eine Synchronisation von Transitionssystemen entstehen) an deren Stelle die partielle oder strikte Ordnung.

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

Eigenschaften von Handlungen:

- *Extensionalität*: Handlungen sind durch ihre Wirkung beschreibbar.
- *Unteilbarkeit* (auch: Atomizität): Handlungen werden vom Prozessor ununterbrochen ausgeführt.
- *Anordnung*:
 - a) Ordnung durch eine totale Ordnung: sequentieller Prozess
 - b) Ordnung durch eine 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*.

Nebenläufige Prozesse können auf (mindestens) zwei Weisen dargestellt werden:

- a) als partielle Ordnung, wie z.B. in Abb. 5.1 als Relation $R = \{(a, c), (b, c)\}$ oder
- b) als lineare Ordnung in Form von Folgen: $u := a; b; c$ und $v := b; a; c$.

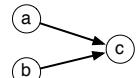
Die Darstellungsform a) heißt *partial order semantics* oder *PO-Semantik*, während b) *interleaving semantics* oder *Folgen-Semantik* heißt.

Beispiel 5.1 Um die Nebenläufigkeit von Zuweisungen a und b , wie im Programm

START $a : x := 3; b : y := 4; c : z := x + y$ **STOP**

zu modellieren, kann – wie in Abbildung 5.1 gezeichnet – ein Produkt von Transitionssystemen oder ein Petrinetz verwendet werden.

Die Semantik wird durch Folgenmenge $\{abc, bac\}$ oder durch die nebenstehende Striktordnung beschrieben:



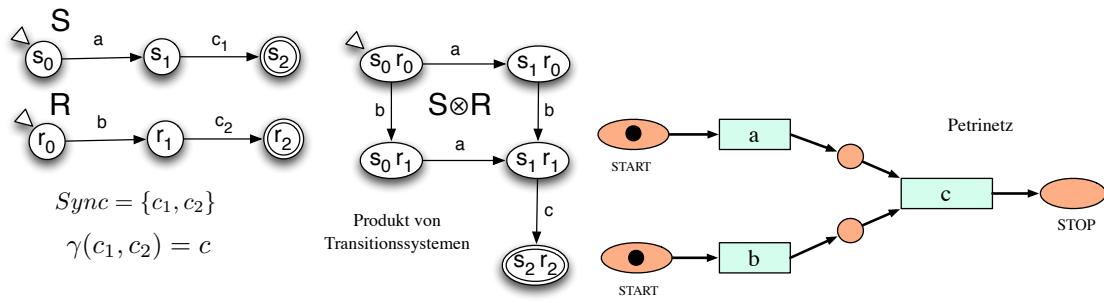


Abbildung 5.1: Nebenläufige Handlungen a und b , gefolgt von c

5.1 Partielle und strikte Halbordnung

Dieser Abschnitt gibt eine formale Definition partieller Ordnungen.

Definition 5.2 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. $\forall a \in A. (a, a) \in R$ “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:

- 1. $\forall a \in A. (a, a) \notin R$ “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), falls gilt:

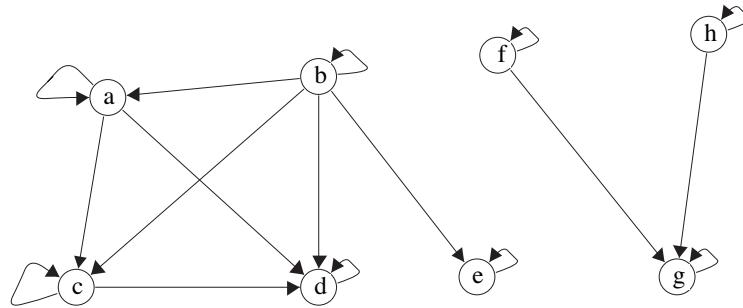
- 1. (A, R) ist partielle Ordnung.
- 2. $\forall a, b \in A. a \neq b$ impliziert $(a, b) \in R \vee (b, a) \in R$. “Vollständigkeit”

Eine Striktordnung mit 2. heißt totale oder lineare Striktordnung.

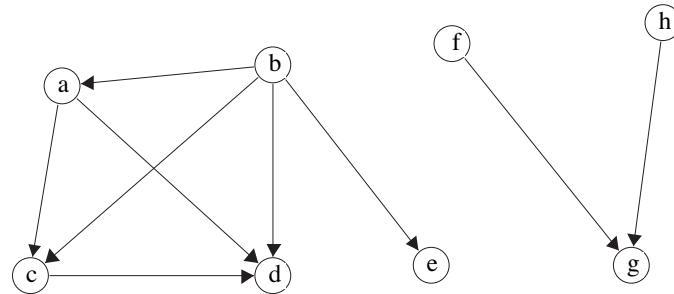
In Abb. 5.2 ist oben eine partielle Ordnung R und darunter die Striktordnung $S = R - id$ dargestellt. Striktordnungen lassen sich oft übersichtlicher als Präzedenzgraph (“Hasse-Diagramm”) darstellen, der nur die direkten Nachfolger enthält (Abb. 5.3).

Definition 5.3 Sei $(A, <)$ eine strikte Ordnung und $a, b \in A$.

1. b heißt direkter Nachfolger von a (in Zeichen: $a \lessdot b$), falls:
 $a \lessdot b : \Leftrightarrow a < b \wedge \neg \exists c \in A. a < c \wedge c < b$
2. (A, \lessdot) heißt Präzedenzrelation zu $(A, <)$.

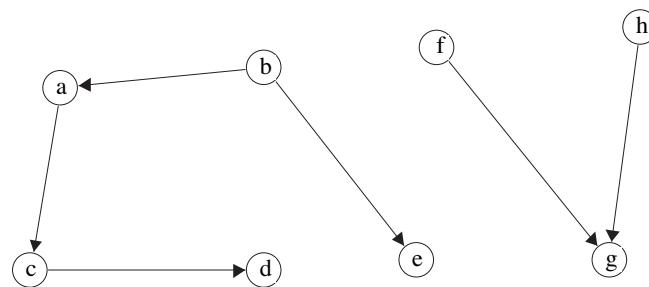


$$R = \{(a,a), (a,c), (b,a), \dots\}$$



$$S = R - id = \{(b,a), (a,c), (b,c), \dots\}$$

Abbildung 5.2: Partielle Ordnung R und Striktordnung S



$$Q = S - S^2 = \{(b,a), (a,c), (c,d), \dots\}$$

Abbildung 5.3: Präzedenzgraph Q

3. Die Menge $\circ = \{a \in A \mid \neg \exists b \in A : b < a\}$ ist die Menge der minimalen Elemente der Ordnung $<$.

Entsprechend ist $=^{\circ} = \{a \in A \mid \neg \exists b \in A : a < b\}$ ist die Menge der maximalen Elemente.

Anmerkung:

- a) Es gilt: $\lessdot = (< - <^2)$ (d.h. die Relation $<$ ohne transitiv gebildete Paare)
("—" Mengendifferenz, $<^2 := (< \circ <) = \{(a, b) \mid \exists c. a < c \wedge c < b\}$ Relationenprodukt)
- b) Gilt $\lessdot \subseteq \lessdot^+$ (transitive Hülle), dann heißt (A, \lessdot) kombinatorisch. In diesem Fall ist $<$ durch \lessdot festgelegt. Für endliche Mengen A ist (A, \lessdot) immer kombinatorisch.
- c) Falls A unendlich ist, muss dies nicht gelten:
Für die rationale Zahlen $(\mathbb{Q}, <)$ gilt zum Beispiel $\lessdot = \emptyset$.
- d) Ist eine Striktordnung $(A, <)$ isomorph zu einer Teilmenge von \mathbb{N} (mit der von \mathbb{N} geerbten Striktordnung und damit total), dann wird sie gerne mit einer Folge beschrieben:

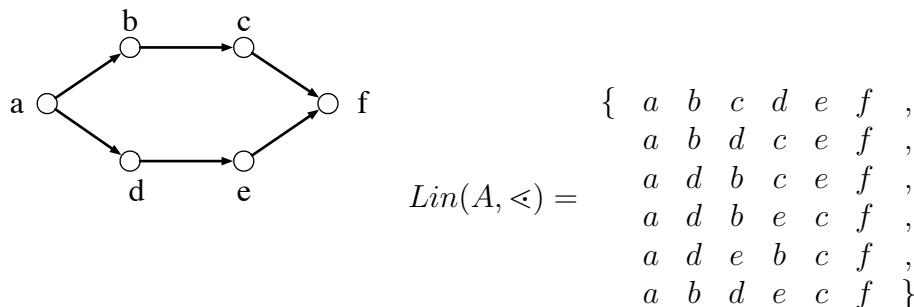
$$\begin{aligned} a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n &\quad \text{mit} \quad a_1 a_2 \dots a_n \quad (\text{card}(A) = n) \\ &\quad \text{oder} \quad a_1; a_2; \dots; a_n \\ a_1 \rightarrow a_2 \rightarrow \dots &\quad \text{mit} \quad a_1 a_2 \dots \quad (\text{card}(A) = \text{card}(\mathbb{N})) \\ &\quad \text{oder} \quad a_1; a_2; \dots \end{aligned}$$

- e) Für eine Striktordnung $(A, <)$ ist die Menge der linearen Vervollständigungen (auch: Linearisierung) von $(A, <)$:

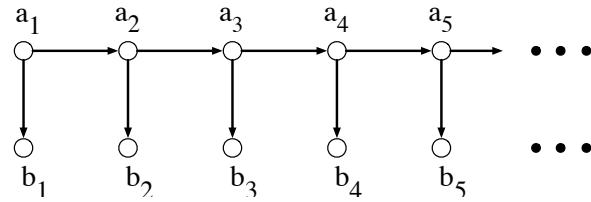
$$Lin(A, <) := \{(A, <_1) \mid <_1 \text{ ist lineare Striktordnung mit } < \subseteq <_1\} \quad (5.1)$$

Ist $(A, <)$ kombinatorisch mit $\lessdot \subseteq \lessdot^+$ dann wird: $Lin(A, \lessdot) := Lin(A, \lessdot^+)$ definiert.

Beispiel 5.4 Die Ordnung (A, \lessdot) sei durch folgenden Graph definiert (Konstruktionsprinzip?):



Beispiel: Die Ordnung (A, \lessdot) sei durch folgenden Graph definiert:



$$\begin{aligned} \text{Lin}(A, \lessdot) = \{ & \quad a_1 \quad b_1 \quad a_2 \quad b_2 \quad a_3 \quad b_3 \quad a_4 \quad b_4 \quad a_5 \quad b_5 \quad \dots \\ & a_1 \quad a_2 \quad b_1 \quad b_2 \quad a_3 \quad b_3 \quad a_4 \quad b_4 \quad a_5 \quad b_5 \quad \dots \\ & a_1 \quad a_2 \quad b_1 \quad a_3 \quad b_2 \quad b_3 \quad a_4 \quad b_4 \quad a_5 \quad b_5 \quad \dots \\ & \vdots \end{aligned}$$

Konstruktionsprinzip?
(so nicht möglich, da $\text{Lin}(A, \lessdot)$ überabzählbar)

Aufgabe 5.5 (Linearisierung)

- Ist a, b, c, d, e, f, g, h eine Linearisierung der Striktordnung von Abb. 5.2 ?
- Berechnen Sie für die Striktordnung S von Abb. 5.2 die Relation S^2 .

5.2 Logische und vektorielle Zeitstempel

In diesem Abschnitt wird die Rolle partieller Ordnungen in verteilten Systemen betrachtet. Unter anderem wird gezeigt, wie durch Zeitstempel eine strikte Ordnung zu einer totalen Ordnung (*Lamport-Ordnung* genannt) verschärft werden kann, um dadurch eine globale Sicht auf das verteilte System zu ermöglichen.

Kausalität und Zeit spielen eine wichtige Rolle beim Entwurf verteilter Systeme. Oft ist es wichtig, die relative Ordnung von Ereignissen und Aktionen zu kennen. In verteilten Systemen ist meist keine zentrale einheitliche Zeitmessung möglich. Trotzdem kann aber die relative Ordnung von Ereignissen durch sogenannte *logische Uhren* festgehalten werden, indem *logische Zeitstempel* gesetzt und den zu versendenden Nachrichten beigefügt werden. Logische Uhren wurden in [Lam78] eingeführt und werden in den Lehrbüchern [Lyn96], [AW98] und [Mat89] behandelt.

Definition 5.6 Ein Nachrichten-Modell ist ein System von n Funktionseinheiten bzw. Prozessoren p_0, \dots, p_{n-1} , die

- lokale Rechenschritte ausführen und
- Nachrichten an andere versenden.

Ein *Ereignis* ist entweder das Absenden oder das Empfangen von Nachrichten. Zur Vereinfachung wird außerdem angenommen, dass in einem Ereignis höchstens eine einzige Nachricht gesendet oder empfangen wird. Diese Ereignisse sind in der Ereignismenge $\Phi = \{\phi_1, \phi_2, \dots\}$ enthalten.

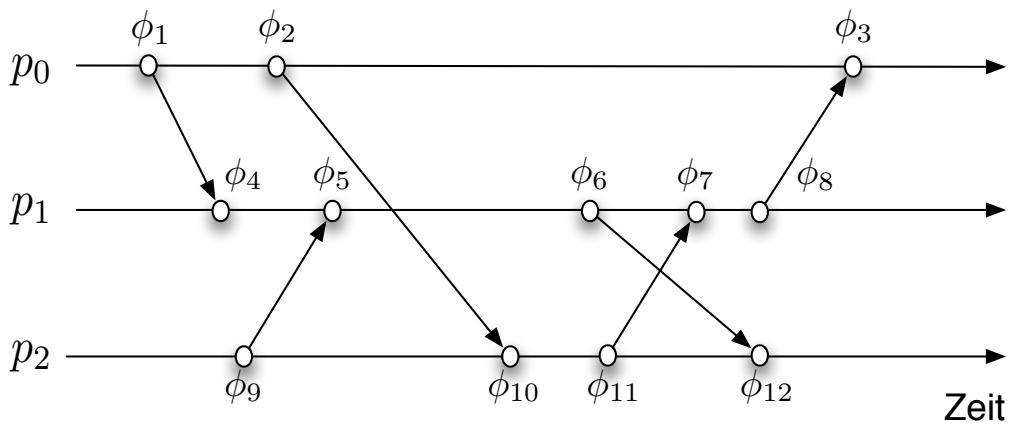


Abbildung 5.4: Nachrichten-Modell mit $n=3$ Prozessoren durch lokale Zeitskalen

Definition 5.7 Es wird eine Relation $\text{vor} \subseteq \Phi \times \Phi$ definiert. Für $\phi_1, \phi_2 \in \Phi$ gelte $(\phi_1 \text{ vor } \phi_2)$, falls folgendes gilt:

- a) Gehören ϕ_1 und ϕ_2 zu dem selben Prozessor (d.h. sie liegen auf der selben (linear geordneten) Zeitachse), dann gilt $(\phi_1 \text{ vor } \phi_2)$ genau dann, wenn ϕ_1 vor ϕ_2 auf der Zeitachse liegt.

- b) Gehören ϕ_1 und ϕ_2 zu verschiedenen Prozessoren (d.h. sie liegen auf verschiedenen Zeitachsen) und ist ϕ_1 das Sendeereignis einer Nachricht, die in ϕ_2 empfangen wird, dann gilt (ϕ_1 vor ϕ_2).
- c) Gibt es ein Ereignis ϕ mit (ϕ_1 vor ϕ) und (ϕ vor ϕ_2), dann gilt auch (ϕ_1 vor ϕ_2) (transitiver Abschluss).

(Φ, vor) ist eine strikte Ordnung, denn nach c) ist sie transitiv. Wäre sie nicht irreflexiv, dann müsste für ein Ereignis ϕ_1 die Beziehung $(\phi_1 \text{ vor } \phi_1)$ gelten. Dies kann nur daher kommen, dass $(\phi_1 \text{ vor } \phi_2)$ und $(\phi_2 \text{ vor } \phi_1)$ für ein Ereignis ϕ_2 auf einer anderen Zeitachse gilt. Sei oBdA. ϕ_1 ein Sendeereignis. Dann kann es kein Empfangsereignis sein, d.h. es muss ein Ereignis ϕ_3 auf der Zeitachse von ϕ_1 geben mit $(\phi_1 \text{ vor } \phi_2 \text{ vor } \phi_3 \text{ vor } \phi_1)$. Dann gilt aber gleichzeitig $(\phi_3 \text{ vor } \phi_1)$ und $(\phi_1 \text{ vor } \phi_3)$, was für Ereignisse auf der selben Zeitachse nach Bedingung a) ausgeschlossen ist.

Für Anwendungen in realen Systemen kann es wichtig sein, (Φ, vor) in einer linearen Ordnung zu erweitern, d. h. eine lineare Vervollständigung zu finden, die die **vor**-Relation nicht verletzt. Dazu müssen die lokale Zeitskalen in konsistente globale Zeitskalen transformiert werden. Dies wollen wir anhand des Anwendungsbeispiels eines Bankssystems darstellen.

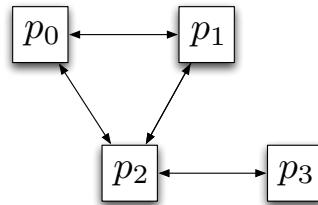


Abbildung 5.5: Banksystem mit 4 Filialen p_0, \dots, p_3

Beispiel 5.8 Funktionseinheiten (Filialen) p_i enthalten lokale Variablen x_i mit aktuellem Kontostand. Über die Kanäle wird als Nachricht eine Menge m_{ij} von p_i nach p_j transferiert (siehe Abb. 5.6).

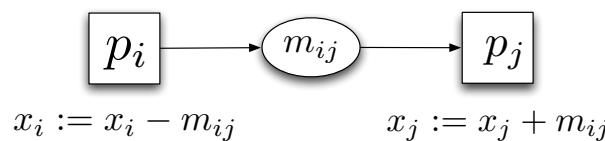


Abbildung 5.6: Übertragung von Nachrichten über Kanal

Es liege eine fortwährende Aktivität vor, d. h. jede Funktionseinheit versendet unendlich oft eine Nachricht an jede andere Funktionseinheit. Falls eine Funktionseinheit terminiert, so kann sie ständig Nachrichten mit $m = 0$ absenden. Der Anfangszustand des Modells

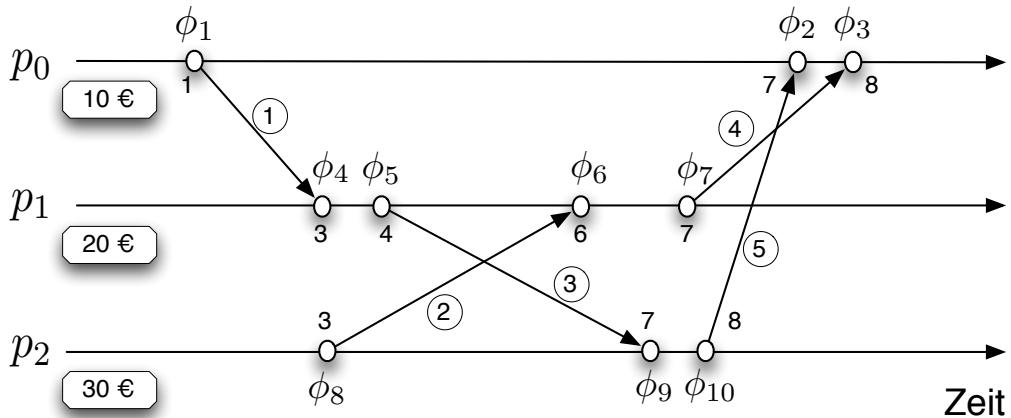


Abbildung 5.7: Zeitskala zu Beispiel 5.8

ist $(x_0, \dots, x_{n-1}) = (a_0, \dots, a_{n-1})$ und keine Nachricht ist unterwegs. $c := \sum_{i=0}^{n-1} a_i$ ist die Gesamtgeldmenge.

$$\text{Gesamtgeldmenge : } 10 + 20 + 30 = 60$$

$$m(\phi_8, \phi_6) = 2$$

1, 3, 4, ..., 8 sind die Zeitzustände der lokalen Uhren

Problem: Die Bankleitung möchte immer wieder in Intervallen die insgesamt umlaufende Geldmenge ermitteln. Diese sollte immer gleich 60 sein.

Verfahren 1:

Die Bankleitung fordert alle Funktionseinheiten auf, die Kontostände zu einem bestimmten Zeitpunkt t mitzuteilen.

Erwartung: $\sum = 60$.

Beispiel 5.9 Zeitpunkt $t = 5$

$$\begin{array}{rcl} p_0 & : & x_0 = 10 - 1 = 9 \\ p_1 & : & x_1 = 20 + 1 - 3 = 18 \\ p_2 & : & x_2 = 30 - 2 = 28 \\ \hline & & \sum 55 ! \end{array}$$

Verfahren 2:

Die Bankleitung bittet die Summe der abgesandten minus der Summe der eingegangenen Beträge zu x_i jeweils hinzuzuzählen.

Erwartung: $\sum = 60$.

Beispiel 5.10 Zeitpunkt $t = 5$

$$\begin{array}{rcl} p_0 & : & x_0 = 9 + 1 = 10 \\ p_1 & : & x_1 = 18 - 1 + 3 = 20 \\ p_2 & : & x_2 = 28 + 2 = 30 \\ \hline & & \sum 60 \end{array}$$

Aber es kann auch der Fall von Abb. 5.8 eintreten, wo zum Zeitpunkt t ein unerwarteter Überschuss von 63 Geldeinheiten beobachtet wird. Solche Effekte sind durch einfache Zählverfahren nicht zu beheben, jedoch durch die im Folgenden eingeführten *Zeitstempelverfahren*.

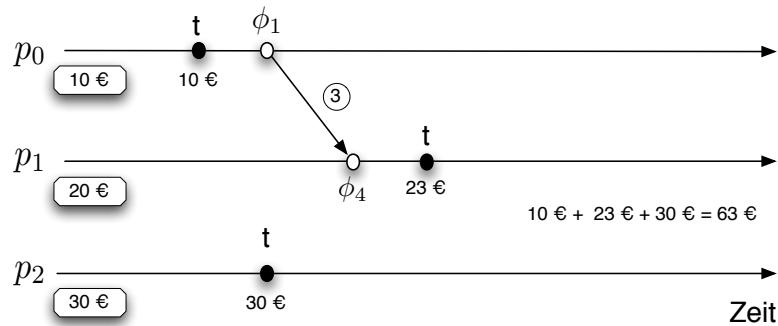


Abbildung 5.8: Im Zeitpunkt t wird virtueller Überschuss durch Nachrichten „aus der Zukunft“ beobachtet.

Es stellt sich die Frage, wie die Relation **vor** im System beobachtet werden kann. Dazu bestimmen wir eine *logische Uhr* $LT(\phi)$ mit

$$\phi_1 \text{ vor } \phi_2 \Rightarrow LT(\phi_1) < LT(\phi_2).$$

Zur Realisierung führt jede Funktionseinheit p_i eine Variable LT_i mit Anfangswert $LT_i = 0$ mit. Den Nachrichten wird der neue Wert des Sendeereignisses beigefügt (*logische Zeitstempel*). Ein Ereignis ϕ von p_i setzt LT_i auf einen um 1 größeren Wert als das Maximum des alten Wertes und eines ggf. in ϕ empfangenen Zeitstempels.

Definition 5.11 Sei ϕ ein Ereignis von p_i . Dann bezeichnet $LT(\phi)$ den von ϕ berechneten Wert von LT_i .

Satz 5.12 Für die Ereignisse $\phi_1, \phi_2 \in \Phi$ gilt:

$$\phi_1 \text{ vor } \phi_2 \Rightarrow LT(\phi_1) < LT(\phi_2)$$

Man beachte, dass das Gegenbeispiel von Abb. 5.8 hier nicht mehr auftreten kann!

Beispiel 5.13 Zeitpunkt $t = 1,5$ in Abb. 5.9:

p_0 :	$\phi = \phi_1, \phi' = \phi_2$	$c_0 = 9$
p_1 :	t liegt vor ϕ_4	$c_1 = 20$
p_2 :	$\phi = \phi_8, \phi' = \phi_9$	$c_2 = 28$
$\sum \quad 57$		

Es kommen an : 1 in ϕ_4 und 2 in ϕ_6 . Die Summe ist $57 + 3 = 60$.

Jedoch: woher weiß die Leitung, ob alle Nachrichten angekommen sind? Die Funktionseinheiten werden aufgefordert mitzuteilen, wieviele Nachrichten an welche andere Funktionseinheit abgesandt bzw. von solchen empfangen wurden.

Algorithmus 5.1 Verfahren der Bankleitung

1. Man führe logische Uhren ein.
2. Man lege ein $t \in \mathbb{Q}$ mit $t \geq 0$ fest.
3. Für jede Funktionseinheit p_i :
 - Bestimme in Bezug auf die lokale Zeit aufeinander folgende Ereignisse ϕ und ϕ' von p_i mit $LT(\phi) \leq t < LT(\phi')$, falls t nicht vor dem ersten Ereignis von p_i liegt.
 - Setze c_i auf den Wert von x_i zwischen ϕ und ϕ'
oder auf den Anfangswert, falls t vor dem ersten Ereignis von p_i liegt. Sende c_i an Leitung.
 - Sende den Wert jeder Geldsendung an Leitung, die ab ϕ' ankommt, aber einen
Zeitstempel $\leq LT(\phi)$ hat.

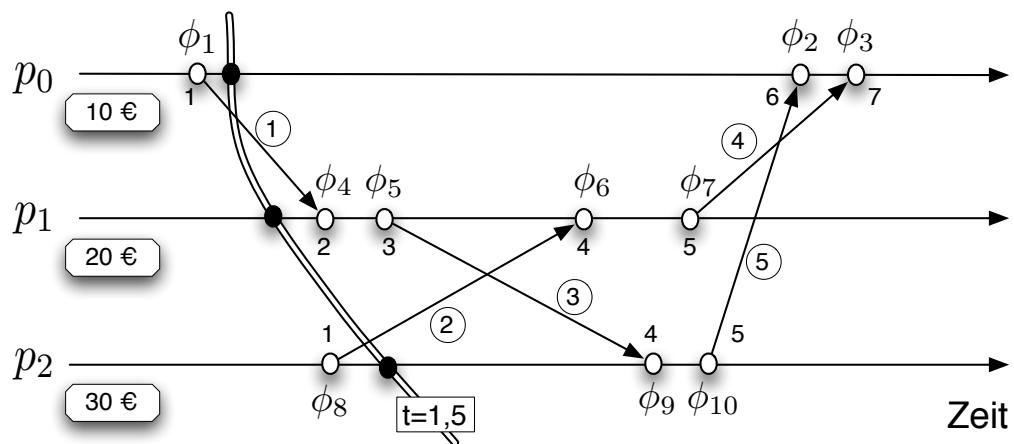


Abbildung 5.9: Zeitskala zu Beispiel 5.13

Im Beispiel 5.13 sind dies zum Zeitpunkt t=1,5:

- p_0 : eine an p_1
- p_1 : keine
- p_2 : eine an p_1

Die Relation $<$ ist i.A. keine totale Ordnung. Oft wird in verteilten Algorithmen jedoch eine totale Ordnung auf den Ereignissen benötigt, z.B. um eindeutige Prioritäten zu regeln. Zu diesem Zweck kann die Relation $<$ zu einer totalen Relation erweitert werden, indem in den Fällen, in denen zwei Ereignisse gleiche Zeitstempel haben, als „tie-break“ Regel die hier ja gegebene totale Ordnung auf den Prozessorbezeichnern hinzugenommen wird. Diese Ordnung heißt nach ihrem Finder *Lamport-Ordnung*.

Definition 5.14 Werden die Ereignisse ϕ durch ihren Zeitstempel $LT(\phi)$ und ihren Prozess(or) p_i charakterisiert, dann ist die Lamport-Ordnung $<_L$ auf den Ereignissen definiert als:

- $$(LT(\phi), p_i) <_L (LT(\phi'), p_j) \text{ genau dann, wenn}$$
- a) $LT(\phi) < LT(\phi')$ oder
 - b) $LT(\phi) = LT(\phi')$ und $i < j$.

Beispiel 5.15 $(2, p_5) <_L (3, p_5)$, $(2, p_5) <_L (3, p_4)$, $(2, p_5) >_L (2, p_4)$

Wir haben gesehen, dass gilt:

$$\phi_1 \text{ vor } \phi_2 \Rightarrow LT(\phi_1) < LT(\phi_2)$$

Stellt die Relation $<$ die vor-Relation exakt dar, d.h. gilt auch die folgende Umkehrung?

$$LT(\phi_1) < LT(\phi_2) \Rightarrow \phi_1 \text{ vor } \phi_2$$

Diese Umkehrung gilt nicht, da $<$ - wie gesagt - keine totale Ordnung ist. Dazu noch ein Gegenbeispiel: In der Zeitskala von Abb. 5.10 ist Folgendes zu beobachten: Es gilt $LT(\phi_9) = 1 < LT(\phi_2) = 2$ aber nicht $\phi_9 \text{ vor } \phi_2$. Der Grund ist, dass $LT(\phi) \in \mathbb{N}$ und \mathbb{N} linear geordnet ist.

Abhilfe: Statt \mathbb{N} wählen wir die nicht linear, aber partiell geordnete Menge \mathbb{N}^n für $n > 1$. Zur Erinnerung: Die *partielle Ordnung* auf \mathbb{N}^n ist komponentenweise definiert: $\vec{v}_1 \leq \vec{v}_2 \Leftrightarrow \forall i \in \{1, \dots, n\} : \vec{v}_1[i] \leq \vec{v}_2[i]$. Die *strikte Ordnung* auf \mathbb{N}^n schließt Gleichheit aus: $\vec{v}_1 < \vec{v}_2 : \Leftrightarrow \vec{v}_1 \leq \vec{v}_2 \wedge \vec{v}_1 \neq \vec{v}_2$. Zwei Vektoren \vec{v}, \vec{v}' heißen *unvergleichbar*, falls $\neg(\vec{v} \leq \vec{v}') \wedge \neg(\vec{v}' \leq \vec{v})$ gilt.

Definition 5.16 ϕ_1 heißt unabhängig von ϕ_2 bzw. ϕ_1 heißt nebenläufig zu ϕ_2 , geschrieben als $\phi_1 \parallel \phi_2$, falls gilt:

$$\phi_1 \parallel \phi_2 \iff \neg(\phi_1 \text{ vor } \phi_2) \wedge \neg(\phi_2 \text{ vor } \phi_1)$$

Gesucht ist eine strikte Ordnung auf Φ , die \parallel darstellt.

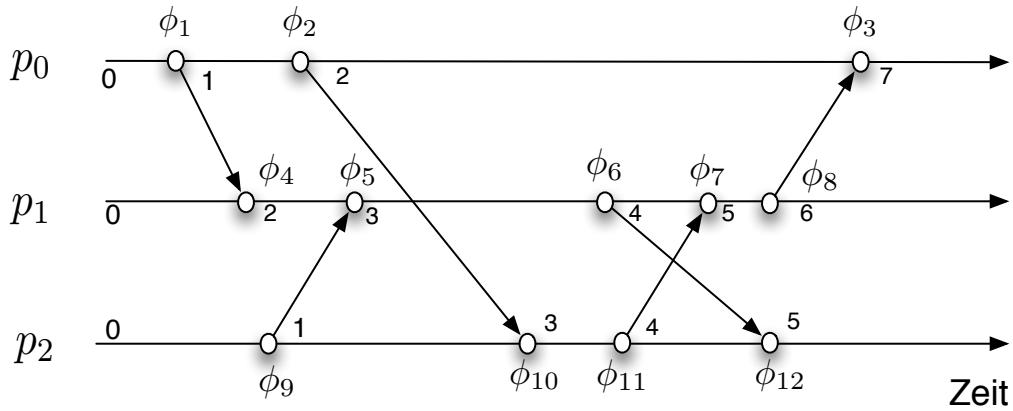


Abbildung 5.10: Senden mit logischen Zeitstempeln

Definition 5.17 (Vektorzeit, vektorieller Zeitstempel) Jede Funktionseinheit p_i führt eine Variable \vec{v}_i mit Werten in \mathbb{N}^n (lokale Vektorzeit) und dem Nullvektor $\vec{0}$ als Anfangswert.

Falls p_i ein Ereignis ϕ bearbeitet, wird der Zeitstempel \vec{v}_i wie folgt aktualisiert:

- Für die eigene Komponente i von p_i gelte:
 $\vec{v}_i[i] \mapsto \vec{v}_i[i] + 1$ (Inkrementieren des eigenen Stempels)
- Für die anderen Komponenten $j \neq i$ von p_i gelte:
 $\vec{v}_i[j] \mapsto \max(\vec{v}_i[j], \vec{V}T_m[j])$ falls eine Nachricht m mit dem Vektorzeitstempel $\vec{V}T_m$ empfangen wird. Wird keine Nachricht empfangen, bleibt $\vec{v}_i[j]$ unverändert ($j \neq i$).
(Aktualisieren der anderen Komponenten)

Die Abbildung $VC : \Phi \rightarrow \mathbb{N}^n$ wird definiert durch $VC(\phi) = \vec{v}_i$, wobei \vec{v}_i der in ϕ durch p_i berechnete Wert ist.

Beispiel: Die Anschrift an ϕ in Abb. 5.11 ist $VC(\phi)$.

Satz 5.18 Die Vektorzeit charakterisiert die **vor**-Relation, denn es gilt:

$$\begin{aligned} VC(\phi_1) < VC(\phi_2) &\iff \phi_1 \text{ vor } \phi_2 \\ VC(\phi_1), VC(\phi_2) \text{ unvergleichbar} &\iff \phi_1 \parallel \phi_2 \end{aligned}$$

Aufgabe 5.19 Der Bankleitung werden ständig alle Vektor-Zeiten $VC(\phi)$ gesandt. Kann Sie darauf ein Verfahren aufbauen, um das Bilanz-Problem zu lösen?

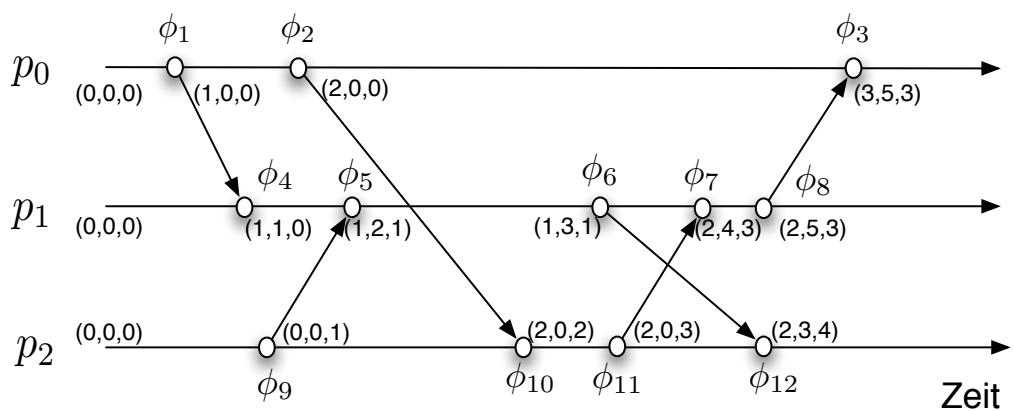


Abbildung 5.11: Zeitskala zur Vektorzeit

6 Petrinetze und Nebenläufigkeit

Dieses Kapitel führt in den Formalismus der Petrinetze ein. Wichtige ihrer Eigenschaften lassen sich folgendermaßen zusammenfassen.

- 1.) graphische und äquivalente algebraische/textuelle Darstellung
- 2.) (formal abgesicherte) Algorithmen für die Analyse
- 3.) Abstraktion und hierarchische Strukturen
- 4.) hoch entwickelte Theorie der Nebenläufigkeit (concurrency)
- 5.) Rechnerwerkzeuge für Editieren, Simulation und Analyse
- 6.) Universalität in Anwendbarkeit (Anwendungen in fast allen Gebieten)
- 7.) Varianten des gleichen Modellierungskonzeptes (Zeit-Netze, stochastische Netze, high-level, objektorientiert, ...)

Dies bewirkt unter anderem, dass sie „einfach“ zu vermitteln sind, dass Werkzeuge „einfach“ miteinander verknüpft werden können sowie die Verträglichkeit von verschiedenen Abstraktionsebenen. Die meist benutzten Vertreter der im letzten Punkt der vorstehenden Aufzählung genannten „high-level Netze“ sind die *gefärbten Netze*, bei denen es keine Einschränkung für den Datentyp der Plätze gibt. Diese werden gesondert behandelt. Wie gesagt, besteht ein großer Vorteil der Petrinetze darin, dass viele Eigenschaften dieses komplexeren Modells sehr ähnlich zu den Eigenschaften der in diesem Kapitel eingeführten elementaren Platz/Transitions-Netze sind. Empfehlenswerte Monographien zu Petrinetzen sind [GV03], [Rei82] und [Rei10].

6.1 Petrinetze

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 Beispiel stellt den Startvorgang eines Autorennens dar [GV03]. Zur Vereinfachung werden nur drei Objekte modelliert: zwei Autos und ein Starter (Abb. 6.1). Wenn die Fahrer der Wagen ihre Vorbereitungen abgeschlossen haben, geben Sie ein Fertigzeichen („ready sign“). Sobald 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:

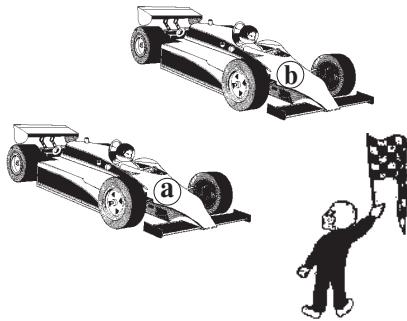


Abbildung 6.1: Start zweier Rennwagen

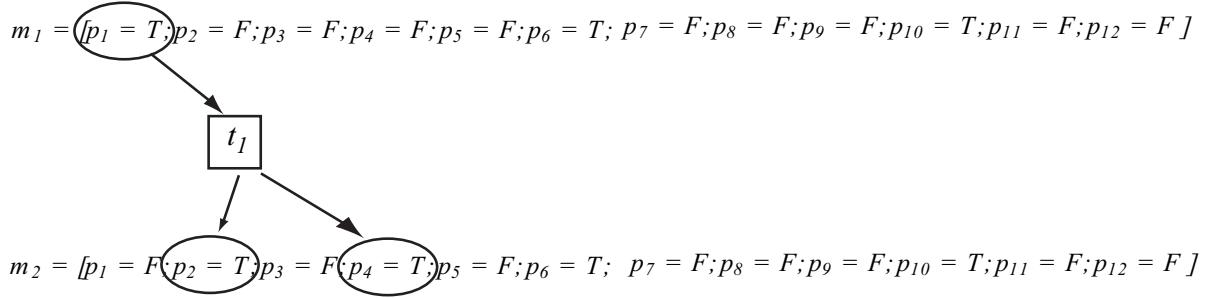
Liste der Bedingungen	Liste der Aktionen
p_1 : car a; preparing for start	
p_2 : car a; waiting for start	
p_3 : car a; running	
p_4 : ready sign of car a	
p_5 : start sign for car a	t_1 : car a; send ready sign
p_6 : starter; waiting for ready signs	t_2 : car a; start race
p_7 : starter; start sign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

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 Petrinetzen
Es gibt zwei disjunkte Mengen von Grundelementen: <i>P-Elemente</i> (state elements, Plätze, Stellen) und <i>T-Elemente</i> (Transition-Element, Transitionen). 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.


 Abbildung 6.2: Lokalität der Aktion t_1

- 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:

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

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:

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

Die graphische Darstellung dieses Zustandsüberganges in Abb. 6.2 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 Aktionsbezeichner nennen wir die Menge $\{t_1, p_1, p_2, p_4\}$ die Lokalität von t_1 .

II	Das Prinzip der Lokalität für Petrinetze
Das Verhalten einer Transition wird ausschließlich durch ihre <i>Lokalität</i> bestimmt, welche sich aus ihr und der Gesamtheit ihrer Eingangs- und Ausgangselemente zusammensetzt.	

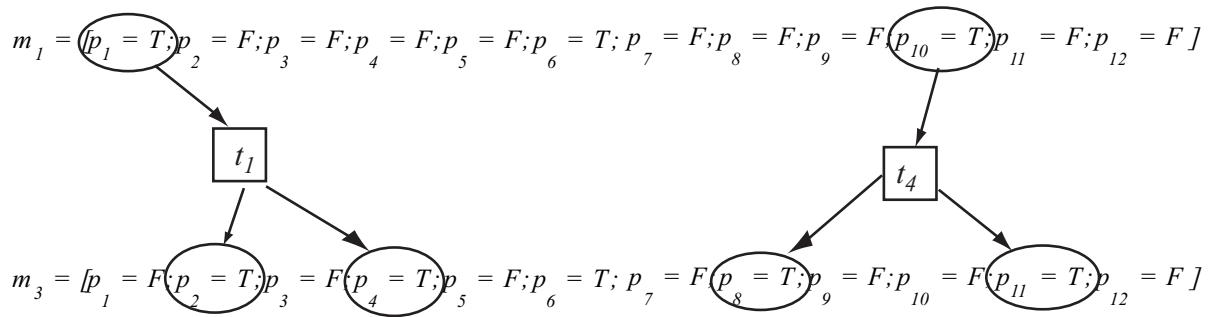


Abbildung 6.3: Nebenläufige Aktionen t_1 und t_4

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:

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

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. 6.3 drückt dies auch grafisch aus. Dies ist die Basis zur Modellierung von Nebenläufigkeit in Petrinetzen.

III

Das Prinzip der Nebenläufigkeit für Petrinetze

Transitionen mit disjunkter Lokalität finden unabhängig (nebenläufig, concurrently) statt.

Die Abb. 6.4 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. 6.5. (Einige Plätze enthalten *Marken*, die den Anfangszustand markieren, worauf wir später zurückkommen.)

IV

Das Prinzip der grafischen Darstellung von Petrinetzen

P-Elemente (auch S-Elemente) werden durch runde grafische Elemente (Kreise, Ellipsen,...) 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.

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

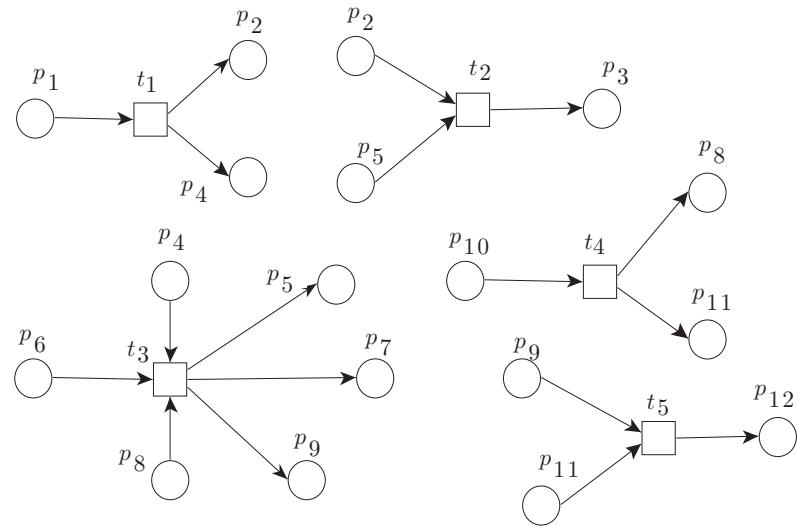
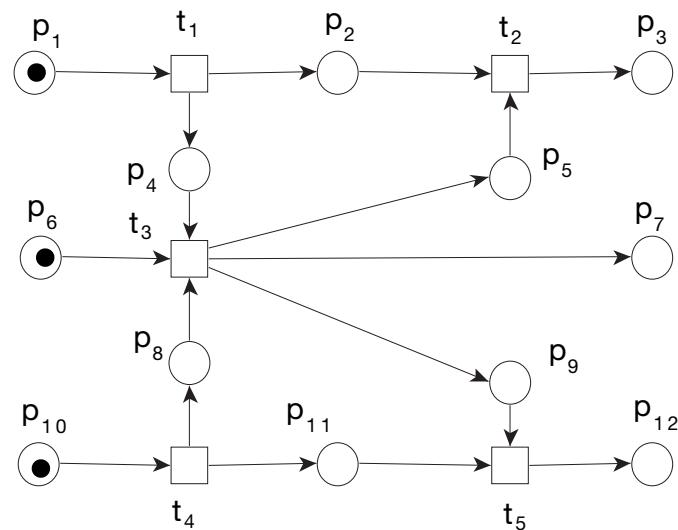


Abbildung 6.4: Einzelaktionen als Transitionen


 Abbildung 6.5: Netz \mathcal{N} zum Beispiel

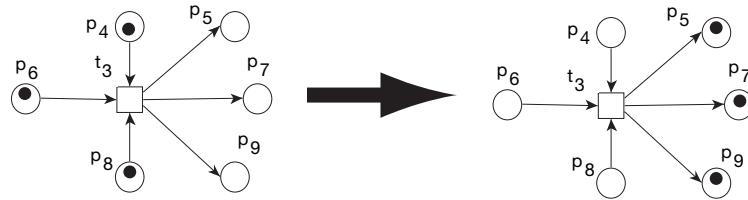


Abbildung 6.6: Schaltregel

Definition 6.1 Ein Netz ist ein Tripel $\mathcal{N} = (P, T, F)$, wobei gilt:

- P ist eine Menge von Plätzen (auch: Stellen).
- T eine Menge von Transitionen (disjunkt zu P).
- $F \subseteq (P \times T) \cup (T \times P)$ ist die Flussrelation.

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

Für das Beispielnetz erhält man $P = \{p_1, \dots, p_{12}\}$, $T = \{t_1, \dots, t_5\}$ und $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_4), \dots\}$.

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

Definition 6.2 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 $x^\bullet := \{y \in P \cup T \mid (x, y) \in F\}$ die Menge der Ausgangsselemente von x .

Für eine Menge von Elementen $X \subseteq P \cup T$ seien entsprechend

$$\bullet X := \bigcup_{x \in X} \bullet x \text{ und } X^\bullet := \bigcup_{x \in X} x^\bullet$$

$loc(y) := \{y\} \cup \bullet y \cup y^\bullet$ heißt Lokalität des Platzes oder der Transition y .

Ist x ein Platz, dann heißen $\bullet x$ bzw. x^\bullet Eingangs- bzw. Ausgangs-Transitionen. Analog für Transitionen.

Beispiel 6.3 Für $X = \{t_1, p_5, p_{11}\}$ im Netz 6.5 erhält man $\bullet X = \{p_1, t_3, t_4\}$ und $X^\bullet = \{p_2, p_4, t_2, t_5\}$.

Die Gültigkeit einer Bedingung wird durch eine Marke in dem entsprechenden Platz dargestellt (siehe Abb. 6.5). Die Schaltregel wird informell in Abb. 6.6 gezeigt: Eine

Transition kann schalten, wenn alle Eingangsstellen eine Marke enthalten und die Ausgangsstellen unbelegt sind. Das Ergebnis des Schaltens ist das Entfernen der Marken in den Eingangsstellen und das Hinzufügen der Marken zu den Ausgangsstellen.

Definition 6.4 Sei \mathcal{N} ein Petrinetz und $M \subseteq P$ eine Markierung.

Die Transition $t \in T$ ist in M aktiviert, wenn $\bullet t \subseteq M$ und $t^\bullet \cap M = \emptyset$ gilt.

Die Nachfolgemarkierung M' ergibt sich dann als $M' = (M \setminus \bullet t) \cup t^\bullet$.

Um das simultane Schalten von Transitionen zu beschreiben, wollen wir definieren, wann eine Transitionsmenge $U \subseteq T$ aktiviert ist. Dies ist der Fall, wenn der Vorbereich von U markiert, der Nachbereich von U unmarkiert ist und die Umgebungen aller Transitionen in U disjunkt zueinander sind, d.h. wenn alle Transitionen nebenläufig zueinander sind.

Definition 6.5 Sei \mathcal{N} ein Petrinetz und $M \subseteq P$ eine Markierung.

Eine Transitionsmenge $U \subseteq T$ ist in M aktiviert, wenn gilt:

1. Vorbereich markiert: $\bullet U \subseteq M$.
2. Nachbereich leer: $U^\bullet \cap C = \emptyset$.
3. Transitionen nebenläufig: $\forall t, t' \in U : t \neq t' \Rightarrow loc(t) \cap loc(t') = \emptyset$.

Die Nachfolgemarkierung M' ist definiert als $M' = (M \setminus \bullet U) \cup U^\bullet$.

Setzen wir $U = \{t\}$, so erhalten wir die bekannte sequentielle Schaltregel.

Beispiel 6.6 Die Abbildung 6.7 zeigt alle möglichen Folgen von Transitionseignissen. Nebenläufige (z.B. t_1 und t_4) Transitionen sind sowohl als simultaner Schritt wie auch in Folgensemantik dargestellt.

6.1.1 Verfeinerung und Vergrößerung von Netzen

Hierarchiebildung ist ein wesentliches Konzept zur Darstellung und Strukturierung von Systemen. Die entsprechenden Begriffe der Vergrößerung und Verfeinerung sind für Netze ohne Anschriften (Kantenbewertung, Markierung) definiert.

Die Konstruktion von Systemhierarchien durch Vergrößerung (Abstraktion) und Verfeinerung ist eine wichtige Methode des Systementwurfs. Petrinetze unterstützen dies durch besondere mit ihrer Struktur kompatible Konzepte. Diese werden unabhängig von Markierungen und speziellen Netzmodellen gebildet und daher für einfache Netze definiert. Wir beginnen mit dem Begriff des *Randes* einer Menge von Plätzen und Transitionen, der die Schnittstelle des zu vergrößernden Teiles bilden wird.

Definition 6.7 Sei $\mathcal{N} = (P, T, F)$ ein Netz, $X := P \cup T$ und $Y \subseteq X$ eine Menge von Elementen. Dann heißt $\partial(Y) := \{y \in Y \mid \exists x \notin Y . x \in loc(y)\}$ der Rand (engl. border) der Menge Y .

Y heißt Platz-berandet (place-bordered) oder offen, wenn $\partial(Y) \subseteq P$, und Transition-berandet (transition-bordered) oder abgeschlossen, falls $\partial(Y) \subseteq T$.

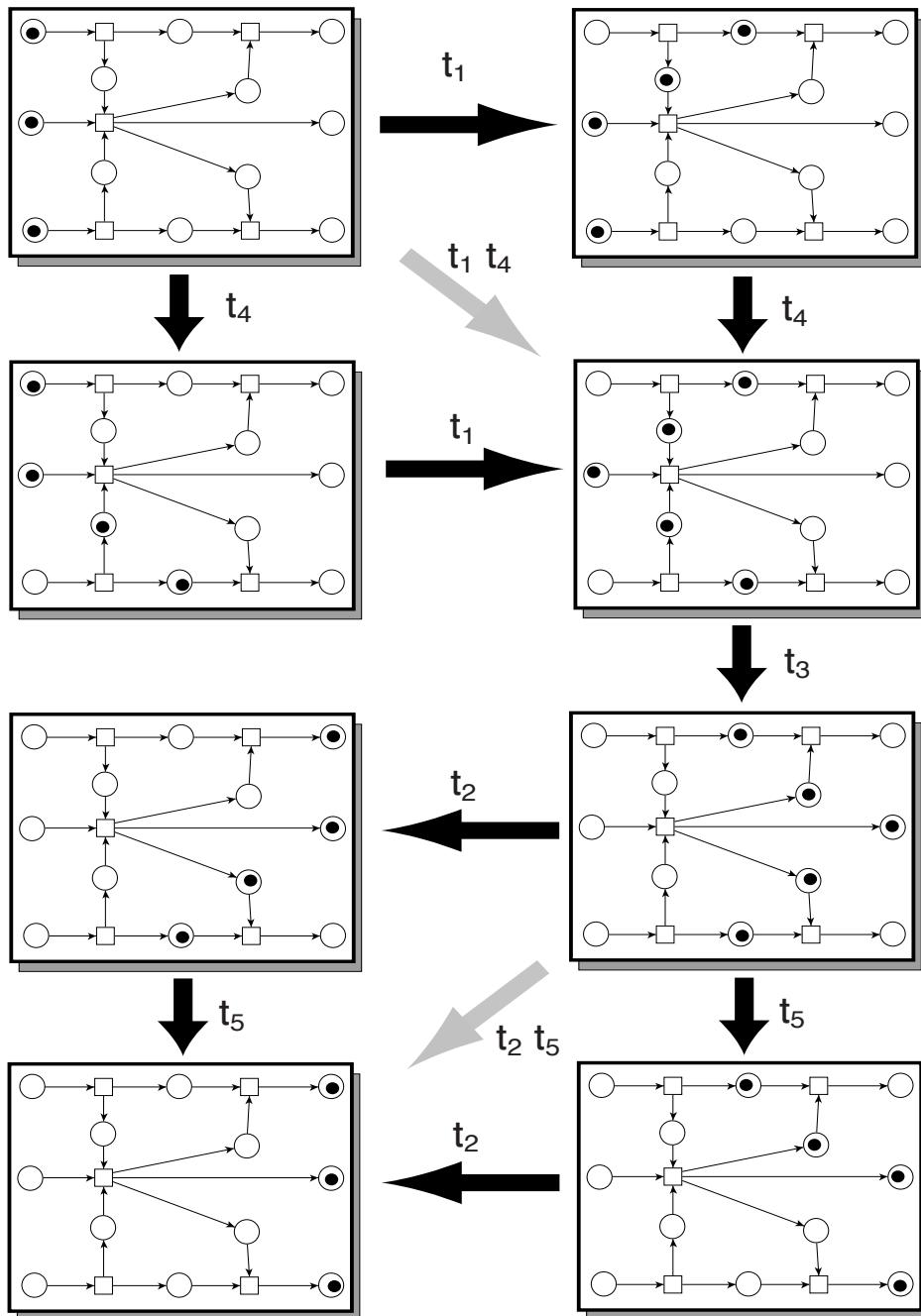


Abbildung 6.7: Schafftfolgen des Netzes 6.5

Anmerkung: Eine Menge Y kann gleichzeitig offen und abgeschlossen sein, wie z.B.: $Y := P \cup T$. In diesem Fall hängt es von der Interpretation bzw. Anwendung ab, ob Y durch einen Platz oder eine Transition ersetzt wird.

Platz-berandete Mengen heißen auch *offen*. Transitions-berandete Mengen heißen *abgeschlossen*. Die Bezeichnung ist in Anlehnung an die Topologie gewählt, denn offene und abgeschlossene Mengen definieren eine Topologie, die eine Formalisierung von Nachbarschaft auf der graphischen Struktur von Netzen darstellt.

Um eine Vergrößerung mit der Netzstruktur verträglich zu gestalten, sollten im Normalfall Platz- bzw. Transitions-berandete Mengen durch einen Platz bzw. eine Transition ersetzt werden.

Die Menge $Y = \{p_3, p_4, t_2, t_3, t_4\}$ des Netzes in Abb. 6.8 ist Transitions-berandet und wird daher zu einer Transition t_Y vergröbert. Auf diese Weise erhält man wieder ein Netz $\mathcal{N}[Y] = (P[Y], T[Y], F[Y])$, das in Abb. 6.9 dargestellt ist. $P[Y]$ enthält alle Plätze mit Ausnahme derjenigen aus Y . $T[Y]$ enthält alle Transitionen mit Ausnahme derjenigen aus Y und das neue Element t_Y . $F[Y]$ ist die Vereinigung von 3 Kantenmengen, nämlich (1) derjenigen, die kein Ende in Y haben, (2) derjenigen die von außerhalb von Y zu t_Y führen und (3) derjenigen, die von t_Y nach Außerhalb führen. Diese Operation wird nun formalisiert.

Definition 6.8 Sei $\mathcal{N} = (P, T, F)$ ein Netz und Y eine nicht leere Transitions-berandete Menge von Elementen.

Dann heißt $\mathcal{N}[Y] = (P[Y], T[Y], F[Y])$ elementare Vergrößerung von \mathcal{N} in Bezug auf Y , falls gilt:

1. $P[Y] = P \setminus Y$.
2. $T[Y] = (T \setminus Y) \cup \{t_Y\}$, wobei t_Y ein neues Element ist.
3. $F[Y] = \{(x, y) \mid x \notin Y \wedge y \notin Y \wedge (x, y) \in F\} \cup \{(x, t_Y) \mid x \notin Y \wedge \exists y \in Y . (x, y) \in F\} \cup \{(t_Y, x) \mid x \notin Y \wedge \exists y \in Y . (y, x) \in F\}$.

Wenn Y eine Platz-berandete Menge ist, dann ist $\mathcal{N}[Y] = (P[Y], T[Y], F[Y])$ analog definiert:

1. $P[Y] = (P \setminus Y) \cup \{p_Y\}$, wobei p_Y ein neues Element ist,
2. $T[Y] = T \setminus Y$,
3. $F[Y] = \{(x, y) \mid x \notin Y \wedge y \notin Y \wedge (x, y) \in F\} \cup \{(x, p_Y) \mid x \notin Y \wedge \exists y \in Y . (x, y) \in F\} \cup \{(p_Y, x) \mid x \notin Y \wedge \exists y \in Y . (y, x) \in F\}$.

Anmerkung: Die Definition von $\mathcal{N}[Y]$ ist mehrdeutig, falls Y gleichzeitig Platz- und Transitions-berandet ist. Dann schreiben wir $\mathcal{N}[Y^{(p)}]$ falls Y als Platz-berandete Menge aufgefasst wird und $\mathcal{N}[Y^{(t)}]$ im anderen Fall.

Definition 6.9 a) Wenn $\mathcal{N}_2 = \mathcal{N}_1[Y]$ eine einfache Vergrößerung von \mathcal{N}_1 für eine Platz- oder Transitions-berandete Menge Y ist, dann heißt \mathcal{N}_1 einfache Verfeinerung (simple refinement) von \mathcal{N}_2 .

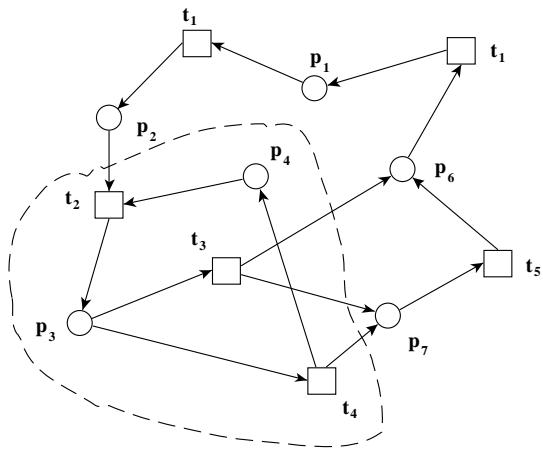


Abbildung 6.8: Eine transitionsberandete Menge

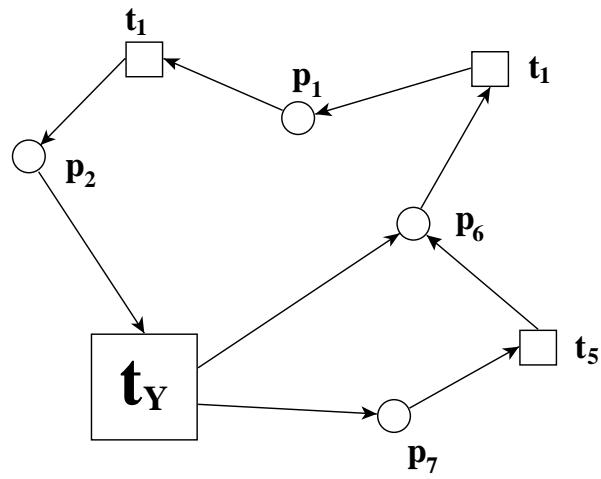


Abbildung 6.9: Vergrößerung des Netzes

Für eine Menge $\{Y_1, Y_2, \dots, Y_n\}$ von paarweise disjunkten, Platz- oder Transitions-berandeten Teilmengen von $P_1 \cup T_1$ wird $\mathcal{N}_2 = (\dots ((\mathcal{N}_1[Y_1])[Y_2]) \dots [Y_n])$ Vergrößerung (abstraction) von \mathcal{N}_1 genannt und \mathcal{N}_1 ist eine Verfeinerung (refinement) von \mathcal{N}_2 . \mathcal{N}_2 wird durch $\mathcal{N}_2 = \mathcal{N}_1[Y_1, Y_2, \dots, Y_n]$ bezeichnet.

- b) Eine Vergrößerung $\mathcal{N}_2 = \mathcal{N}_1[Y_1, Y_2, \dots, Y_n]$ von \mathcal{N}_1 wird als Faltung (folding) bezeichnet, wenn jedes Y_i entweder eine Menge von Plätzen, d.h. $Y_i \subseteq P_1$, oder eine Menge von Transitionen, d.h. $Y_i \subseteq T_1$, ist.

In der Definition einer strikten Abstraktion wird Y_i im ersten Fall durch einen Platz p_{Y_i} und im zweiten Fall durch eine Transition t_{Y_i} ersetzt. \mathcal{N}_1 wird strikte Verfeinerung von \mathcal{N}_2 genannt.

Durch folgende Konvention können Mehrdeutigkeiten vermieden werden: falls in a) oder b) eine Menge Y_i ($1 \leq i \leq n$) sowohl Platz- als auch Transitions-berandet ist, kann die Vergrößerung durch $\mathcal{N}_2 = \mathcal{N}_1[Y_1, \dots, Y_i^{(d)}, \dots, Y_n]$ bezeichnet werden, wobei $d = p$ bzw. $d = t$ ist und Y_i als a Platz- bzw. Transitions-berandete Menge betrachtet wird.

Abb. 6.10 zeigt eine nicht einfache Vergrößerung. Das obere Netz stellt das aus der Vorlesung F4 bekannte Beispielnetz zum Start eines Autorennens dar. Dabei sind die vergrößerten Mengen $Y = \{t_1, t_2, t_3, t_4, t_5, p_2, p_4, p_5, p_8, p_9, p_{11}\}$, $Y_1 = \{t_6, p_{13}, t_7\}$ und $Y_2 = \{t_8, p_{15}, t_9\}$, in der oberen Abbildung durch eine gestrichelte Linien dargestellt. Es handelt sich um drei Transitions-berandete Mengen, die zu den Transitionen t_Y , t_{Y_1} und t_{Y_2} im Netz $\mathcal{N}[Y, Y_1, Y_2]$ der unteren Abbildung verwandelt werden.

Es ist einfach zu zeigen, dass die Vergrößerung eines Netzes wieder ein Netz ist, d.h. der Definition 6.1 genügt. Die Vergrößerung im unteren Teil von Abb. 6.10 hat sinngemäß das entsprechende Verhalten des Netzes darüber. Eine Vergrößerung muss jedoch nicht eine sinnvolle Interpretation haben. Insbesondere überträgt sich der Begriff nicht zwangsläufig auf seine Semantik (d.h. die Menge seiner Prozesse). Als Beispiel betrachte man das Netzfragment in Abb. 6.11 a). Intuitiv hat die Vergrößerung in Abb. 6.11

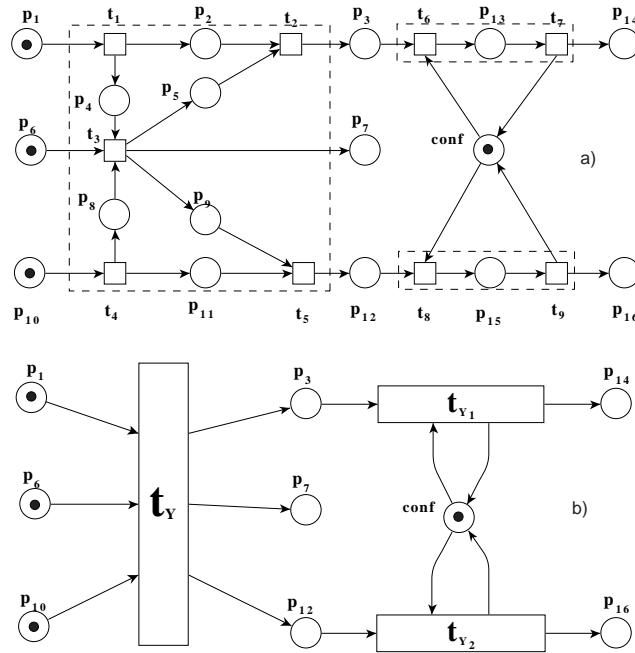


Abbildung 6.10: Eine (nicht einfache) Vergrößerung

d) das entsprechende Verhalten: Marken können von links nach rechts „durchlaufen“. Allerdings ist dieses Netz auch Vergrößerung von Abb. 6.11 b). Das Verhalten ist hier jedoch völlig verschieden.

Die Vergrößerung in Abb. 6.11d) lässt sich in diesem Fall sinnvoll interpretieren, und zwar als Verschmelzung zweier Plätze als Schnittstelle zweier Komponenten. Die Operation wird als *Verschmelzung* oder *Fusion* bezeichnet und zuweilen wie in 6.11 c) dargestellt. Die Abbildungen 6.11 e)-h). zeigen die entsprechenden Fälle für Transitions-berandete Mengen. Es handelt sich um die *Verschmelzung* oder *Fusion* von Transitionen.

Mit Abb. 6.12 sind zwei Netze gegeben, deren Komposition durch Platzverschmelzung das Netz von Abb. 6.10 a) ergibt.

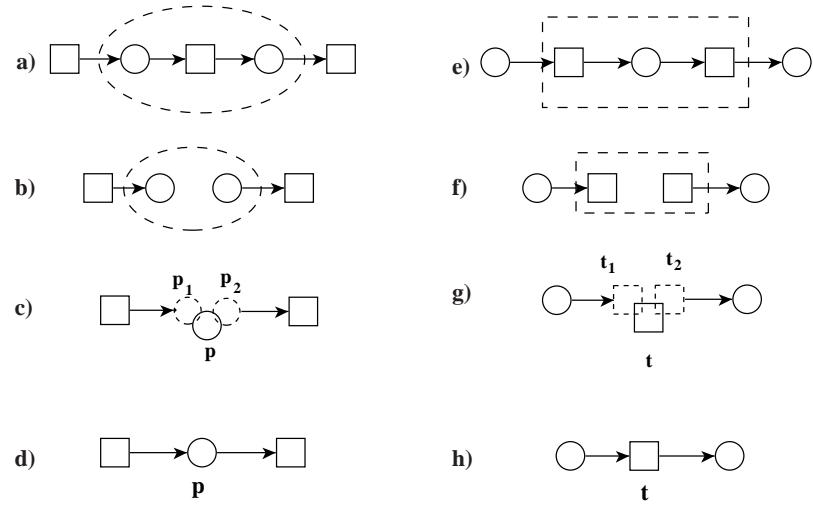


Abbildung 6.11: Vergrößerung und Verschmelzung (Fusion)

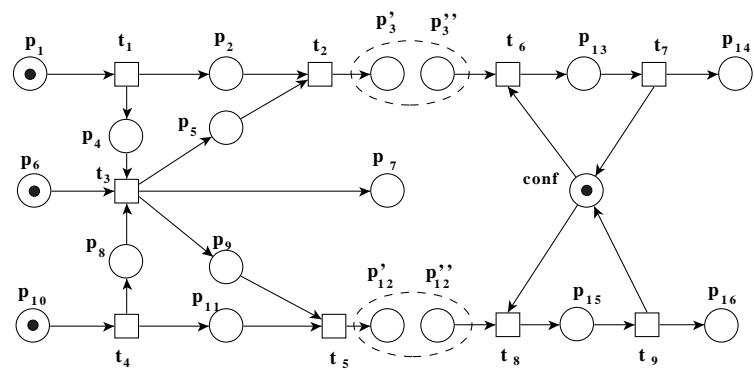


Abbildung 6.12: Komposition durch Verschmelzung (Fusion)

6.1.2 Netzmorphismen

Wie in algebraischen Theorien allgemein üblich, werden strukturerhaltende Abbildungen *Homomorphismen* oder kürzer *Morphismen* genannt. Im folgenden wird Strukturverträglichkeit für Netze eingeführt.

Unter Strukturverträglichkeit ist zu verstehen, dass bei einer Abbildung ϕ eines Netzes $\mathcal{N}_1 = (P_1, T_1, F_1)$ auf $\mathcal{N}_2 = (P_2, T_2, F_2)$ die Kanten F „zusammen“ mit den Plätzen F -erhaltend abgebildet wird:

$$\forall x, y \in (P_1 \cup T_1) : (x, y) \in F_1 \Rightarrow ((\phi(x), \phi(y)) \in F_2 \vee \phi(x) = \phi(y))$$

Mit anderen Worten, die Abbildung der Kanten ergibt sich aus der Abbildung der Knoten.

Eine Abbildung ϕ kann die Kantenrichtung umdrehen, indem einer Kante $(p, t) \in F_1$ die Kante $(\phi(p), \phi(t)) \in F_2$ zugeordnet wird, bei der $\phi(p)$ eine Transition und $\phi(t)$ ein Platz ist. Für Netzmorphismen wird ein solches Umdrehen der Richtung verboten.

Um ein solches Umdrehen der Richtung zu verhindern, wird zusätzlich noch gefordert, dass $(x, y) \in F_1 \cap (P_1 \times T_1)$ auch $(\phi(x), \phi(y)) \in F_2 \cap (P_2 \times T_2)$ oder $\phi(x) = \phi(y)$ impliziert. Analog für $(x, y) \in F_1 \cap (T_1 \times P_1)$ ¹

Ist der Netzmorphismus ϕ surjektiv, so sind die Bildknoten $P_2 \cup T_2$ aus den Konoten des Netzes \mathcal{N}_1 mit ϕ konstruierbar. Gilt zudem noch, dass jede Kante in \mathcal{N}_2 ein Abbild einer Kante in \mathcal{N}_1 ist, dann wird das Netz \mathcal{N}_2 komplett durch \mathcal{N}_1 und ϕ konstruierbar. Solche Morphismen heißen Epimorphismen.

Wir definieren für Kanten $(x, y) \in F$ die Notation $\phi(x, y) := (\phi(x), \phi(y))$.

Definition 6.10 Seien $\mathcal{N}_1 = (P_1, T_1, F_1)$ und $\mathcal{N}_2 = (P_2, T_2, F_2)$ zwei Netze.

- Eine Abbildung $\phi : (P_1 \cup T_1) \rightarrow (P_2 \cup T_2)$ heißt Netzmorphismus, falls gilt:

$$\begin{aligned} (x, y) \in F_1 \cap (P_1 \times T_1) &\Rightarrow (\phi(x), \phi(y)) \in F_2 \cap (P_2 \times T_2) \vee \phi(x) = \phi(y) \\ (x, y) \in F_1 \cap (T_1 \times P_1) &\Rightarrow (\phi(x), \phi(y)) \in F_2 \cap (T_2 \times P_2) \vee \phi(x) = \phi(y) \end{aligned}$$

- Ein Netzmorphismus ϕ heißt Faltung, falls $\phi(P_1) \subseteq P_2$ und $\phi(T_1) \subseteq T_2$ gilt.
- Ein Netzmorphismus ϕ ist ein Epimorphismus, falls ϕ surjektiv ist und für jede Kante ein Urbild existiert, d.h. für alle $f_2 \in F_2$ existiert ein $f_1 \in F_1$ mit $\phi(f_1) = f_2$. Eine Faltung mit dieser Eigenschaft heißt Epifaltung.
- Ein Netzmorphismus ϕ ist ein Netzsomorphismus, falls ϕ eine Bijektion ist und ϕ^{-1} ebenfalls ein Netzmorphismus ist.

Theorem 6.11 Seien \mathcal{N}_1 und \mathcal{N}_2 zwei endliche Netze.

- \mathcal{N}_2 ist genau dann eine Vergrößerung von \mathcal{N}_1 , wenn es einen Epimorphismus von \mathcal{N}_1 nach \mathcal{N}_2 gibt.

¹In der topologischen Terminologie wird also zusätzlich gefordert, dass offene (bzw. abgeschlossene) Mengen ebensolche Urbilder haben.

- \mathcal{N}_2 ist genau dann eine strikte Vergrößerung von \mathcal{N}_1 , wenn es eine Epifaltung von \mathcal{N}_1 nach \mathcal{N}_2 gibt.

Für den Beweis siehe [GV03, Theorem 2.5.4].

Abbildung 6.13 zeigt den Übergang von einem einfachen Netz zu einem P/T-Netz mit mehreren Marken auf einem Platz und Kantengewichten größer als Eins mit Hilfe von Netzfaltungen.²

² Mehrfache Marken und Kantengewichte sind für eine Faltung bislang so nicht definiert, denn Faltungen sind ja Abbildungen auf Petrinetzen. Es handelt sich also um eine verallgemeinerte Form der Faltung, die wir aber hier nicht mehr formal definieren wollen.

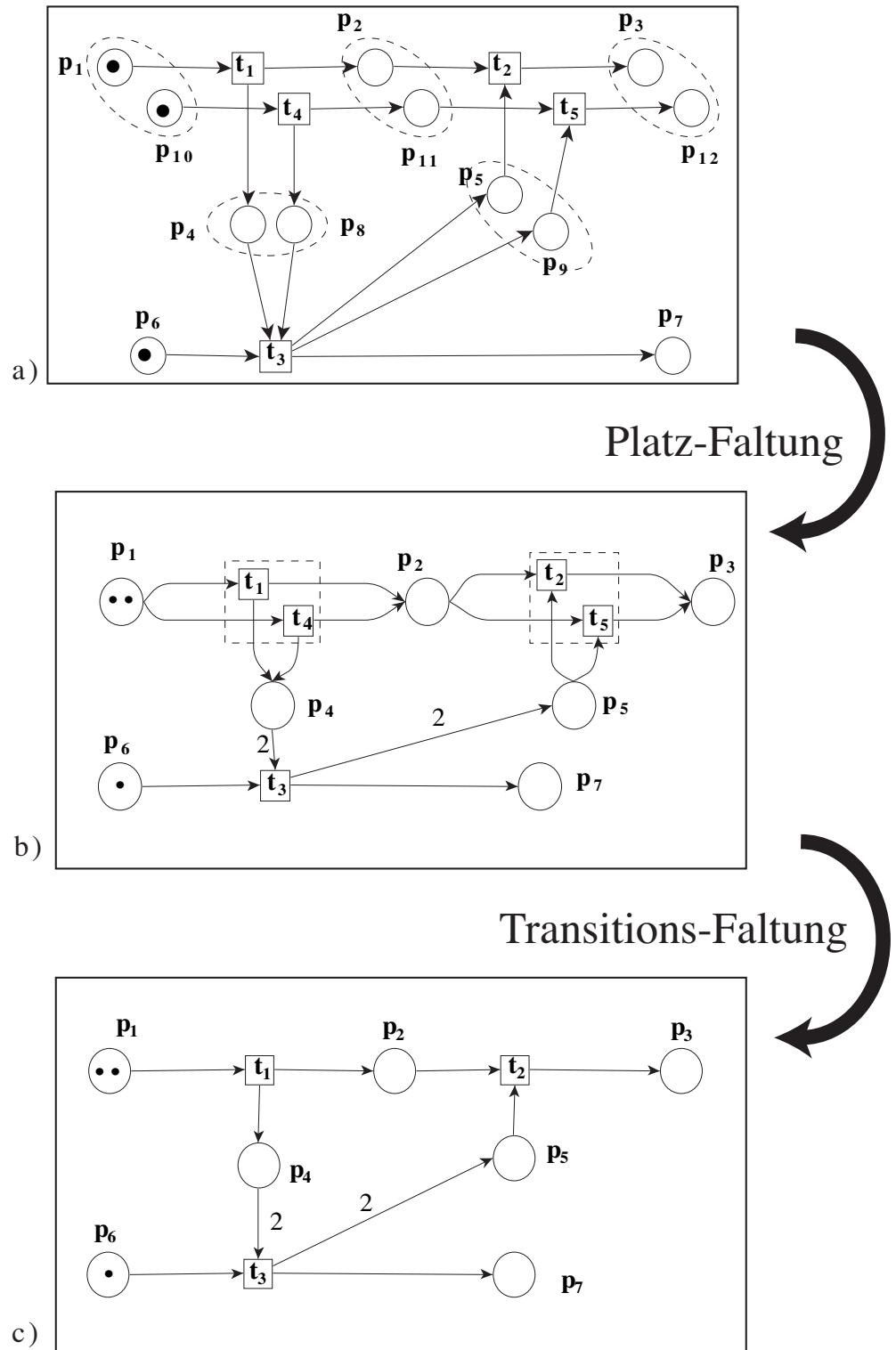


Abbildung 6.13: Netzfaltung zu Platz/Transitions-Netzen

6.2 Platz/Transitions-Netze

In diesem Abschnitt wird das Verhalten (auch: die *Semantik*) der Platz/Transitions-Netze (kurz: *P/T-Netz*) formal eingeführt.³ Dazu gehören Markierungen von Plätzen und ihre Veränderung durch das Schalten von Transitionen. Darüberhinaus wird eine kleine Erweiterung der Netze betrachtet: das *Kantengewicht*.

Definition 6.12 Ein Platz/Transitions-Netz $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ besteht aus den folgenden Komponenten:

- (P, T, F) ist ein endliches Netz.
- $W : F \rightarrow \mathbb{N} \setminus \{0\}$ ist die Kantengewichtung.
- $\mathbf{m}_0 : P \rightarrow \mathbb{N}$ ist die Anfangsmarkierung.

Ein P/T-Netz heißt *einfaches Netz*, falls $W(x, y) = 1$ für alle $(x, y) \in F$ gilt.

Um die Anfangsmarkierung \mathbf{m}_0 eines P/T-Netzes hervorzuheben, notiert man ein Netz auch in der Form $(\mathcal{N}, \mathbf{m}_0)$ mit $\mathcal{N} = (P, T, F, W)$.

Notation: Eine Markierung $\mathbf{m} : P \rightarrow \mathbb{N}$ kann auch als Tupel $\mathbf{m} \in \mathbb{N}^{|P|}$ aufgefasst werden.

Ein *Netzsystem* ist ein P/T-Netz $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ mit $P \neq \emptyset$ und $T \neq \emptyset$.

Eine nicht vorhandene Kante – d.h. (x, y) mit $(x, y) \notin F$ – können wir wie eine Kante mit dem Gewicht $W(x, y) = 0$ behandeln. Dazu definieren wir die Erweiterung von W auf $(P \times T) \cup (T \times P)$, indem wir definieren:

$$\widetilde{W}(x, y) := \begin{cases} W(x, y), & \text{falls } (x, y) \in F \\ 0, & \text{sonst} \end{cases}$$

Wir verwenden meist W sowohl für W als auch für \widetilde{W} .

Definition 6.13 Sei das P/T-Netz $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ gegeben.

- 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: $\mathbf{m} \xrightarrow{t}$).
- Ist t in \mathbf{m} aktiviert, dann ist die Nachfolgemarkierung \mathbf{m}' für alle $p \in P$ definiert durch:

$$\mathbf{m}'(p) = \mathbf{m}(p) - \widetilde{W}(p, t) + \widetilde{W}(t, p))$$

- Insgesamt notieren wir den Schaltvorgang durch $\mathbf{m} \xrightarrow{t} \mathbf{m}'$.
- Es gilt: $\mathbf{m} \xrightarrow{t} \mathbf{m}' \iff \forall p \in P. (\mathbf{m}(p) \geq \widetilde{W}(p, t) \wedge \mathbf{m}'(p) = \mathbf{m}(p) - \widetilde{W}(p, t) + \widetilde{W}(t, p))$

Die Schaltregel ist in der Abbildung 6.14 an einem Beispiel illustriert.

³Die Menge P der Plätze wird in der Literatur auch durch S (*Stellen*) bezeichnet. Entsprechend heißt das Platz/Transitions-Netz dann *Stellen/Transitions-Netz* (*S/T-Netz*).

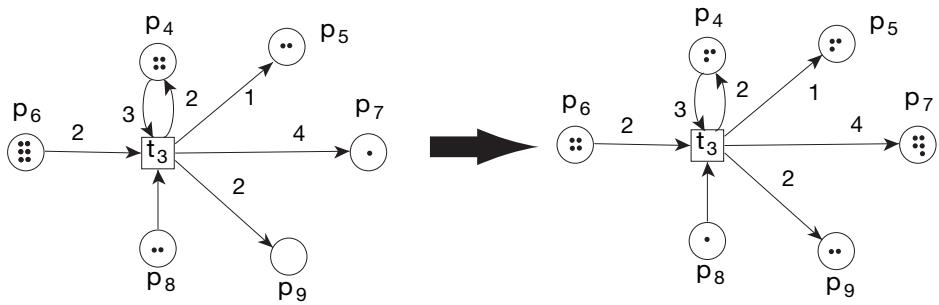


Abbildung 6.14: Schaltregel für P/T-Netze

Wir definieren den Pre-Vektor $W(\bullet, t)$ der Länge $|P|$ durch:

$$W(\bullet, t) := (\widetilde{W}(p_1, t), \dots, \widetilde{W}(p_{|P|}, t))$$

Analog ist der Post-Vektor:

$$W(t, \bullet) := (\widetilde{W}(t, p_1), \dots, \widetilde{W}(t, p_{|P|}))$$

Mit dieser Vektornotation kann die Aktivierung/Nachfolgemarkierung einfacher definiert werden:

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

Dabei werden hier die auf \mathbb{Z} definierten Operatoren $-$, $+$ und \geq in ihrer komponentenweise Erweiterung auf Vektoren aus $\mathbb{Z}^{|P|}$ verwendet.

Multimengennotation für den Vor- und Nachbereich Wir definieren die Abbildungen $\partial_0, \partial_1 : T \rightarrow P^\oplus$, die jeder Transition ihren Vorbereich bzw. ihren Nachbereich aus dem Markenmonoid P^\oplus , d.h. den Multimengen über P zuordnet. Es gilt $\partial_0(t)(p) := W(p, t)$ und $\partial_1(t)(p) := W(t, p)$, was sich auch als Multimenge darstellen lässt:

$$\partial_0(t) := \bigoplus_{p \in P} W(p, t) \cdot p \quad \text{bzw.} \quad \partial_1(t) := \bigoplus_{p \in P} W(t, p) \cdot p$$

Schaltfolgen Die Nachfolgemarkierungsrelation wird auf Wörter über T erweitert:

Definition 6.14 Sei \mathcal{N} ein P/T-Netz.

- $\mathbf{m} \xrightarrow{w} \mathbf{m}'$ 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 $t \in T$.

Eine Transitionsfolge $w \in T^*$ heißt aktiviert in \mathbf{m} (in Zeichen: $\mathbf{m} \xrightarrow{w}$), falls $\exists \mathbf{m}_1 : \mathbf{m} \xrightarrow{w} \mathbf{m}_1$

$FS(\mathcal{N}) := \{w \in T^* \mid \mathbf{m}_0 \xrightarrow{w}\}$ ist die Menge der Schaltfolgen (firing sequence set) von \mathcal{N} .

Der Erreichbarkeitsgraph Sei $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ ein P/T-Netz. Für eine Menge S^0 von Markierungen sei dann die Menge der von S^0 aus *erreichbaren* Markierungen definiert als:

$$\mathbf{R}(\mathcal{N}, S^0) := \{\mathbf{m}_2 \mid \exists w \in T^*, \mathbf{m}_1 \in S^0 : \mathbf{m}_1 \xrightarrow{w} \mathbf{m}_2\}$$

Die *Erreichbarkeitsmenge* $\mathbf{R}(\mathcal{N})$ ist $\mathbf{R}(\mathcal{N}) := \mathbf{R}(\mathcal{N}, \{\mathbf{m}_0\})$.

Für einelementige Mengen S^0 schreiben wir $\mathbf{R}(\mathcal{N}, \mathbf{m})$ anstelle von $\mathbf{R}(\mathcal{N}, \{\mathbf{m}\})$.

Falls \mathcal{N} implizit gegeben ist, schreiben wir auch $\mathbf{R}(\mathbf{m})$ für $\mathbf{R}(\mathcal{N}, \{\mathbf{m}\})$.

Definition 6.15 Der Erreichbarkeitsgraph eines P/T-Netzes \mathcal{N} ist ein Graph $RG(\mathcal{N}) := (Kn, Ka)$ mit Knotenmenge $Kn := \mathbf{R}(\mathcal{N})$ und Kantenmenge $Ka := \{(\mathbf{m}_1, t, \mathbf{m}_2) \mid \mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2\}$.

Der Erreichbarkeitsgraph eines P/T-Netzes \mathcal{N} kann auch als Transitionssystem $TS = (\mathbf{R}(\mathcal{N}), T, Ka, \{\mathbf{m}_0\})$ aufgefasst werden. Dann ist $R(TS) = \mathbf{R}(\mathcal{N})$ und $FS(TS) = FS(\mathcal{N})$.

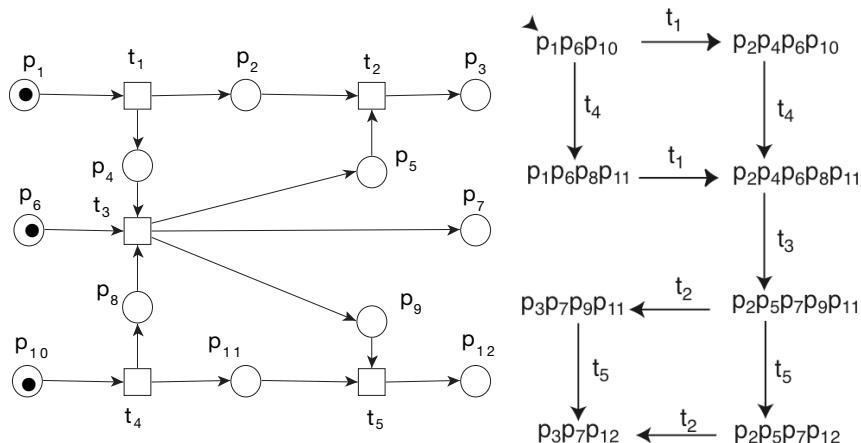
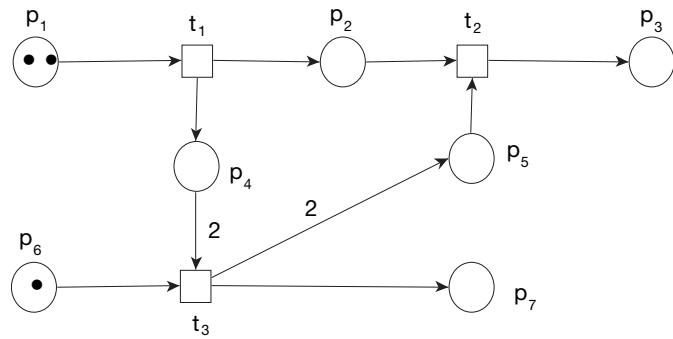


Abbildung 6.15: P/T-Netz \mathcal{N} mit Erreichbarkeitsgraph

Die Abbildung 6.15 zeigt rechts den Erreichbarkeitsgraph des linksstehenden P/T-Netzes.

Beispiel 6.16 Das Netz 6.16 ist ein P/T-Netz. Die beiden Wagen sind hier als ununterscheidbare Objekte modelliert: „zwei Wagen stehen in Startposition“.

Im P/T-Netz \mathcal{N}_3 von Abb. 6.16 ist die Anfangsmarkierung der Vektor $\mathbf{m}_0 = (2, 0, 0, 0, 0, 1, 0)$ oder (alternativ) die Abbildung $\mathbf{m}_0 : P \rightarrow \mathbb{N}$ mit $\mathbf{m}_0(p_1) = 2$, $\mathbf{m}_0(p_6) = 1$ und $\mathbf{m}_0(p_i) = 0$ in den anderen Fällen.


 Abbildung 6.16: Platz/Transitions Netz \mathcal{N}_3

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

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

Oft ist es praktisch eine Markierung als Wort zu schreiben: $\mathbf{m}_0 = p_1^2 p_6$ oder $\mathbf{m}_0 = < p_1^2 p_6 >$. Werden die Markierungen als Wörter dargestellt, so erhält man:

$$p_1^2 p_6 \xrightarrow{t_1} p_1 p_2 p_4 p_6 \xrightarrow{t_1} p_2^2 p_4^2 p_6 \xrightarrow{t_3} p_2^2 p_5^2 p_7 \xrightarrow{t_2} p_2 p_3 p_5 p_7 \xrightarrow{t_2} p_3^2 p_7$$

Eigenschaften: Monotonie, Hürde Eine zentrale Eigenschaft der P/T-Netze ist die der *Monotonie* – etwas salopp formuliert: „Mehr Marken schaden nicht.“

Lemma 6.17 Für jedes P/T Netz gilt die folgende Additivität:

$$\forall w \in T^* : \mathbf{m}_1 \xrightarrow{w} \mathbf{m}_3 \Rightarrow \forall \mathbf{m} : (\mathbf{m}_1 + \mathbf{m}) \xrightarrow{w} (\mathbf{m}_3 + \mathbf{m}) \quad (6.1)$$

(Da die Multimengenaddition kommutativ ist, könnten wir \mathbf{m} genausogut von links addieren.)

Beweis: Beweis der Behauptung (6.1) per Induktion über die Länge von w (als Übung). \square

Die Monotonie-Eigenschaft der Schaltregel folgt dann direkt.

Lemma 6.18 Für jedes P/T-Netz gilt die folgende Monotonie-Eigenschaft:

$$(\mathbf{m}_1 \xrightarrow{w} \mathbf{m}_3 \wedge \mathbf{m}_2 \geq \mathbf{m}_1) \Rightarrow \mathbf{m}_2 \xrightarrow{w} \mathbf{m}_4 \quad (6.2)$$

Beweis: Setzen wir in (6.1) speziell $\mathbf{m} := \mathbf{m}_2 - \mathbf{m}_1 \geq \mathbf{0}$, so ergibt sich (6.2):

$$\mathbf{m}_1 \xrightarrow{w} \mathbf{m}_3 \Rightarrow \mathbf{m}_2 = (\mathbf{m}_1 + \mathbf{m}) \xrightarrow{w} (\mathbf{m}_3 + \mathbf{m}) = (\mathbf{m}_3 + \mathbf{m}_2 - \mathbf{m}_1) = \mathbf{m}_4$$

D.h. die Folgemarkierung ergibt sich als $\mathbf{m}_4 = (\mathbf{m}_3 + \mathbf{m}_2 - \mathbf{m}_1)$. \square

Lemma 6.19 Definiere $\min(\mathbf{m}_1, \mathbf{m}_2)$ durch $\min(\mathbf{m}_1, \mathbf{m}_2)(p) = \min(\mathbf{m}_1(p), \mathbf{m}_2(p))$. Gilt $\mathbf{m}_1 \xrightarrow{\sigma} \text{ und } \mathbf{m}_2 \xrightarrow{\sigma}$, dann auch $\min(\mathbf{m}_1, \mathbf{m}_2) \xrightarrow{\sigma}$.

Beweis: Für Schaltfolgen der Länge 1, d.h. für eine Transition, ist dies leicht einzusehen: Aktivieren \mathbf{m}_1 und \mathbf{m}_2 die Transition t , dann auch ihr Minimum.

Beweis per Induktion. Für $\sigma = \epsilon$ gilt $\mathbf{m} \xrightarrow{\sigma}$ für jedes \mathbf{m} .

Für $\sigma = \sigma't$ sei nun $\mathbf{m}_{1,2} \xrightarrow{\sigma'} \mathbf{m}'_{1,2}$ mit der Induktionsannahme $\min(\mathbf{m}_1, \mathbf{m}_2) \xrightarrow{\sigma'} \mathbf{m}'$ für ein \mathbf{m}' .

Es gilt nun $\mathbf{m}' = \min(\mathbf{m}'_1, \mathbf{m}'_2)$, denn es gilt:

1. Gilt $\mathbf{m}_1(p) \leq \mathbf{m}_2(p)$, dann ist $\mathbf{m}'(p) = \min(\mathbf{m}_1, \mathbf{m}_2)(p) + \Delta(\sigma')(p) = \mathbf{m}_1(p) + \Delta(\sigma')(p) = \mathbf{m}'_1(p)$.
2. Gilt $\mathbf{m}_1(p) \geq \mathbf{m}_2(p)$, dann ist $\mathbf{m}'(p) = \min(\mathbf{m}_1, \mathbf{m}_2)(p) + \Delta(\sigma')(p) = \mathbf{m}_2(p) + \Delta(\sigma')(p) = \mathbf{m}'_2(p)$.

Aus $\mathbf{m}_{1,2} \xrightarrow{\sigma} \text{ folgt } \mathbf{m}'_{1,2} \xrightarrow{t}$. Damit gilt dann auch $\min(\mathbf{m}'_1, \mathbf{m}'_2) \xrightarrow{t}$. Also gilt $\mathbf{m}' \xrightarrow{t}$ und damit $\min(\mathbf{m}_1, \mathbf{m}_2) \xrightarrow{\sigma}$. \square

Eine Markierung \mathbf{m} heißt *Hürde* für $\sigma \in T^*$, falls sie die kleinste σ aktivierende Markierung darstellt, d.h. falls $\mathbf{m} \xrightarrow{\sigma}$ und $\forall \mathbf{m}' \leq \mathbf{m} : \neg \mathbf{m}' \xrightarrow{\sigma}$ gilt.

Lemma 6.20 Zu jedem $\sigma \in T^*$ ist die Hürde eindeutig bestimmt. Sie wird mit $H(\sigma)$ bezeichnet.

Beweis: Existenz: Dass es mindestens ein \mathbf{m} mit der Eigenschaft gibt, sieht man leicht. Man nehme eine beliebige Markierung, die $\sigma = t_1 \cdots t_n$ aktiviert, also bspw. $\sum_{i=1}^n W(p, t_i)$.

Von dieser Markierung entfernen wir nichtdeterministisch von irgendwelchen Stellen so lange Markierungen, wie sich an der Aktiviertheit von σ nichts ändert. Die resultierende Markierung \mathbf{m} aktiviert also σ und jede kleinere tut dies nicht mehr.

Eindeutigkeit: Gäbe es zwei verschiedene Hürden \mathbf{m}_1 und \mathbf{m}_2 . Für diese gilt $\mathbf{m}_1 \xrightarrow{\sigma} \text{ und } \mathbf{m}_2 \xrightarrow{\sigma}$, so dass nach Lemma 6.19 dann auch $\min(\mathbf{m}_1, \mathbf{m}_2) \xrightarrow{\sigma}$ gilt und da $\mathbf{m}_1 \neq \mathbf{m}_2$ folgt damit $\min(\mathbf{m}_1, \mathbf{m}_2) \not\leq \mathbf{m}_1$ oder $\min(\mathbf{m}_1, \mathbf{m}_2) \not\leq \mathbf{m}_2$ im Widerspruch zu der Annahme, dass sowohl \mathbf{m}_1 als auch \mathbf{m}_2 als Hürden bereits minimal sind. \square

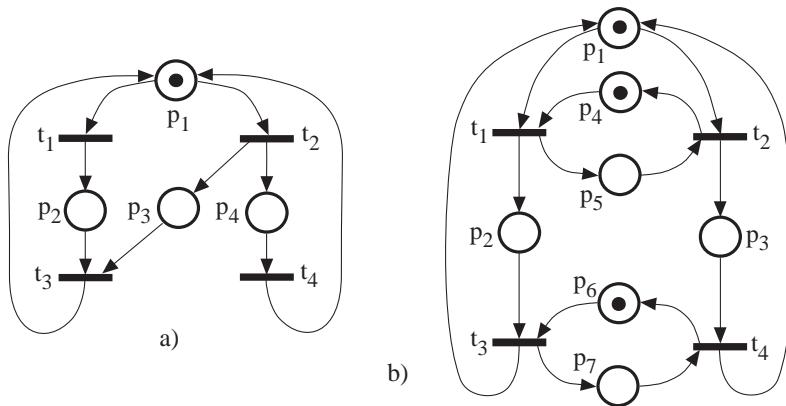


Abbildung 6.17: (a) ein unbeschränktes, nicht-lebendiges und nicht-reversibles Netz;
(b) ein lebendiges Netz, das aber bei vergrößerter Anfangsmarkierung (z.B. $m_0(p_5) = 1$) nicht lebendig ist.

6.2.1 Elementare Systemeigenschaften

In diesem Abschnitt werden einige elementare Systemeigenschaften eingeführt. Diese stellen natürlich nur eine kleine Auswahl von solchen Eigenschaften dar. Obwohl sie für P/T-Netze formuliert werden, sind sie überwiegend auch für andere Systemmodelle (darunter gefärbte Netze) formulierbar. Mit P/T-Netzen lassen sie sich jedoch besonders knapp und präzise definieren.

Die wichtigsten dieser Eigenschaften sind:

- 1) Beschränktheit (*boundedness*), was die Endlichkeit des Zustandraumes bedeutet,
- 2) Lebendigkeit (*liveness*), was die potenzielle Ausführbarkeit bedeutet,
- 3) Reversibilität (*reversibility*), was diejenigen Systeme charakterisiert, die immer in den Anfangszustand zurückgesetzt werden können,
- 4) wechselseitiger Ausschluss (*mutual exclusion*), was die Unmöglichkeit von simultanen Teilmarkierungen (p-mutex) oder Transitionsausführungen (t-mutex) bedeutet.
- 5) Eine Markierung heißt *Rücksetzzustand* (*home state*), wenn sie von jeder erreichbaren Markierung erreicht werden kann.

In der Tabelle 6.1 sind die formalen Definitionen der diskutierten Eigenschaften zusammengefasst. Darin sind Notationen enthalten, die z.T. im vorangehenden Kapitel eingeführt wurden.

Beispiel 6.21 Das Netz in Abb. 6.17 a) ist unbeschränkt, was einem „Überlauf“⁴ in realen Systemen entspricht. Darauf hinaus ist es nicht lebendig. Die Vermehrung von Betriebsmitteln (resources) muss jedoch nicht zu einem lebendigen Netz führen, sondern kann sogar diese Eigenschaft zerstören, wie Abb. 6.17 b) zeigt.

⁴z.B. zu große Zähler, zu volle Puffer, zu viele aktivierte Prozesse

- (1) Sei $k \in \mathbb{N}$. Dann heißt ein Platz $p \in P$ *k-beschränkt* (*k*-bounded) in \mathcal{N} , falls $\forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) : \mathbf{m}(p) \leq k$
 - (2) p heißt *beschränkt* (bounded) in \mathcal{N} , falls $\exists k \in \mathbb{N} \forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) : \mathbf{m}(p) \leq k$
 - (3) \mathcal{N} heißt *k-beschränkt*, wenn alle Plätze *k-beschränkt* sind.
 \mathcal{N} heißt *beschränkt*, wenn alle Plätze *beschränkt* sind.
 - (4) \mathcal{N} heißt *verklemmungsfrei* (deadlock-free), falls $\forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) \exists t \in T : \mathbf{m} \xrightarrow{t}$
 - (5) t heißt *potenziell aktivierbar* in \mathbf{m} , wenn gilt: $\exists \sigma \in T^* : \mathbf{m} \xrightarrow{\sigma i} \mathbf{m}' \xrightarrow{t}$
 t heißt *lebendig* (live) in \mathcal{N} , falls $\forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) \exists \sigma \in T^* : \mathbf{m} \xrightarrow{\sigma t} \mathbf{m}'$
 - (6) \mathcal{N} heißt *lebendig*, falls alle Transitionen lebendig sind.
 - (7) $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ heißt a *Rücksetzzustand* (home state),
falls $\forall \mathbf{m}' \in \mathbf{R}(\mathcal{N}) \exists \sigma \in T^* : \mathbf{m}' \xrightarrow{\sigma} \mathbf{m}$
 - (8) \mathcal{N} heißt *reversibel* (reversible), falls $\forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) \exists \sigma \in T^* : \mathbf{m} \xrightarrow{\sigma} \mathbf{m}_0$
 - (9) *wechselseitiger Ausschluss* (mutual exclusion) in \mathcal{N} :
 p_i und p_j sind in *Markierungs-Ausschluss* (marking mutual exclusion) falls
 $\nexists \mathbf{m} \in \mathbf{R}(\mathcal{N}) : (\mathbf{m}(p_i) > 0) \wedge (\mathbf{m}(p_j) > 0)$
 t_i und t_j sind in *Schalt-Ausschluss* (firing mutual exclusion)
falls $\nexists \mathbf{m} \in \mathbf{R}(\mathcal{N}) : \mathbf{m} \geq W(\bullet, t_i) + W(\bullet, t_j)$
-
- (10) Strukturelle Eigenschaften :
 \mathcal{N} heißt *strukturell beschränkt* (structurally bounded),
falls $(\mathcal{N}, \mathbf{m})$ für alle \mathbf{m} beschränkt ist.
 \mathcal{N} heißt *strukturell lebendig* (structurally live),
falls $(\mathcal{N}, \mathbf{m})$ für mindestens ein \mathbf{m} lebendig ist.
-

Tabelle 6.1: Systemeigenschaften eines P/T-Netzes $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$

Wenn im Netz von Abb.6.17 a) die Transition t_1 schaltet, wird eine *Verklemmung* (deadlock) erreicht, d.h. eine Markierung, in der keine Transition aktiviert ist. Ein Netz heißt *verklemmungsfrei* (deadlock-free), wenn die Erreichbarkeitsmenge keine Verklemmung enthält.

Lebendigkeit (liveness) ist eine stärkere Eigenschaft. Eine Transition t heißt *potenziell aktivierbar* (potentially fireable) in einer gegebenen Markierung \mathbf{m} , wenn eine aktivierte Schaltfolge $\sigma \in T^*$ existiert, die zu einer Markierung \mathbf{m}' führt, in der t aktiviert ist. Eine Transition heißt *lebendig* (live), wenn sie in jeder erreichbaren Markierung potenziell aktivierbar ist. Ein Netz heißt *lebendig*, wenn alle Transitionen in der Anfangsmarkierung lebendig sind.

Egal wie man die Anfangsmarkierung des Netzes in Abb. 6.17 a) wählt, es ist *nicht lebendig*. Dies ist also wieder eine strukturelle Eigenschaft, die daher *strukturelle Nicht-lebendigkeit* (structural non-liveness) heißt. Umgekehrt heißt ein Netz *strukturell lebendig* (structural live), wenn für es eine lebendige Markierung existiert.

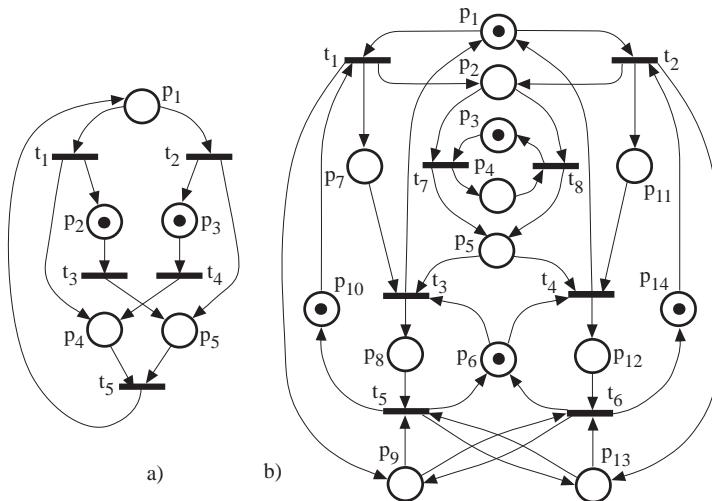


Abbildung 6.18: (a) Die Anfangsmarkierung ist kein Rücksetzzustand, jedoch alle übrigen; (b) Die Anfangsmarkierung ist kein Rücksetzzustand, der übrige Erreichbarkeitsgraph zerfällt in zwei streng zusammenhängende Zusammenhangskomponenten (siehe Abb. 7.4).

Beispiel 6.22 Die Anfangsmarkierung des Netzes in Abb. 6.18 a) ist kein Rücksetzzustand. Das gilt auch für das Netz in Abb. 6.18 b). Es ist ebenfalls lebendig und beschränkt, besitzt aber keinen Rücksetzzustand. Sein Erreichbarkeitsgraph enthält nämlich zwei verschiedene starke Zusammenhangskomponenten, die jeweils alle Transitionen enthalten. Die Menge aller Rücksetzzustände eines Netzes heißt *Rücksetzmenge (home space)*. Die Existenz von Rücksetzzuständen ist natürlich generell für Systeme wünschenswert.

Lebendigkeit, Beschränktheit und Reversibilität sind prominente und „gute“ Systemeigenschaften, und man kann die Frage stellen, ob sie voneinander unabhängig sind, d.h. jede Kombination möglich ist.

Lemma 6.23 Lebendigkeit, Beschränktheit und Reversibilität sind voneinander unabhängige Systemeigenschaften.

Beweis: Durch Abb. 6.19 wird anhand von Beispielen gezeigt, dass alle $2^3 = 8$ Kombinationen tatsächlich vertreten sind. \square

6.2.2 Das Erreichbarkeitsproblem

Das *Erreichbarkeitsproblem* ist das Problem, für eine Markierung zu entscheiden, ob sie im Erreichbarkeitsgraphen vorkommt:

Gegeben: Ein P/T-Netz $(\mathcal{N}, \mathbf{m}_0)$ und eine Markierung $\mathbf{m} \in \mathbb{N}^P$.

Frage: Gilt $\mathbf{m} \in \mathbf{R}(\mathcal{N}, \mathbf{m}_0)$?

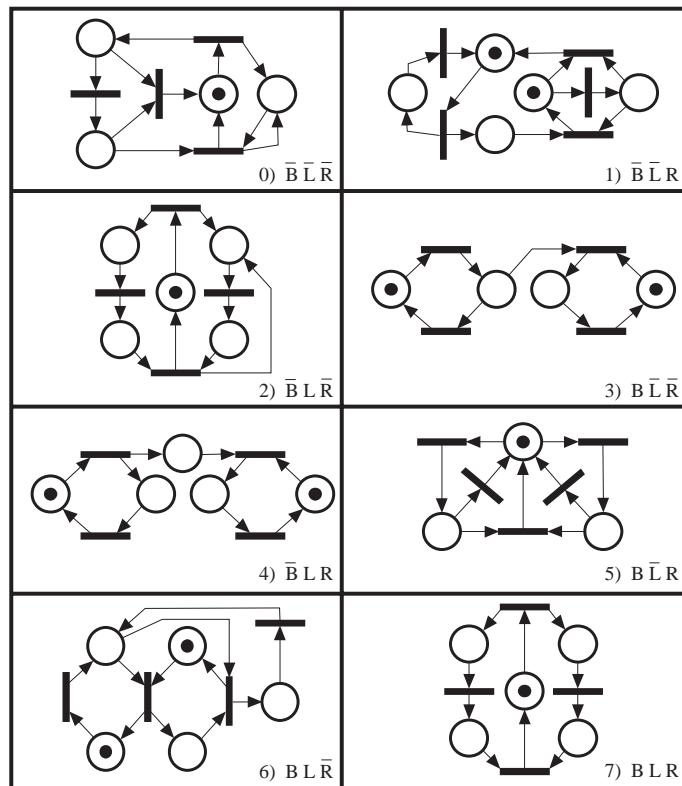


Abbildung 6.19: Beschränktheit (B), Lebendigkeit (L) und Reversibilität (R) sind unabhängige Eigenschaften

Das Überdeckbarkeitsproblem fordert nicht das exakte Erreichen einer Markierung, sondern nur das Erreichen einer Markierung, die mindestens so groß ist wie eine vorgegebene:

Gegeben: Ein P/T-Netz $(\mathcal{N}, \mathbf{m}_0)$ und eine Markierung $\mathbf{m} \in \mathbb{N}^P$.

Frage: Gibt es eine Markierung $\mathbf{m}' \in \mathbf{R}(\mathcal{N}, \mathbf{m}_0)$ mit $\mathbf{m}' \geq \mathbf{m}$?

Das Erreichbarkeitsproblem ist (sogar für unbeschränkte) P/T-Netze entscheidbar. Der Beweis ist jedoch aufgrund seines enormen Umfangs Spezialveranstaltungen vorbehalten.

Satz 6.24 (Mayr, 1984) *Das Erreichbarkeitsproblem ist auch für unbeschränkte P/T-Netze entscheidbar.*

Die bekannten Algorithmen für das Erreichbarkeitsproblem für beliebige P/T-Netze konstruieren eine abgewandelte Form des Überdeckungsgraphen (s. Abschnitt 7.2) und haben daher eine Laufzeit, die auch bei Fällen endlicher Erreichbarkeitsmengen *keine* primitiv rekursive Funktion der Eingabegröße mehr ist. Dies gilt auch für andere Modelle als Petrinetze, sofern sie ein entsprechend nebenläufiges Verhalten darzustellen erlauben. Für beschränkte Netze ist es dagegen stets möglich, den Erreichbarkeitsgraphen zu konstruieren. Das Erreichbarkeitsproblem ist daher für beschränkte Netze offensichtlich entscheidbar. Die Komplexität ist aber enorm:

Satz 6.25 *Das Erreichbarkeitsproblem für beschränkte P/T-Netze ist entscheidbar, benötigt jedoch mindestens exponentiell viel Platz.*

Eine untere Schranke $NSpace(2^{O(n)})$ wurde von Lipton 1976 bewiesen.

Auch für syntaktisch stark eingeschränkte Netze, wie *1-konservative Netze*, sind hohe Analysekomplexitäten bekannt.

Definition 6.26 Ein P/T-Netz \mathcal{N} heißt *konservativ*, wenn es *keine multiplen Kanten* besitzt (d.h. stets $W(x, y) \leq 1$ gilt) und eine Funktion $f : S \rightarrow \mathbb{N} \setminus \{0\}$ existiert, für die gilt:

$$\forall t \in T : \sum_{p \in \bullet t} f(p) = \sum_{p \in t^\bullet} f(p).$$

Netze, bei denen $f(p) = 1$ für alle $p \in P$ ist, heißen *1-konservativ* und es gilt dann automatisch: $\forall t \in T : |t^\bullet| = |\bullet t|$.

Das Erreichbarkeitsproblem ist aber selbst für beschränkte, ja selbst für 1-konservative P/T-Netze komplex.

Satz 6.27 *Das Erreichbarkeitsproblem für 1-konservative P/T-Netze ist PSPACE-vollständig.*

Nur in wenigen Teilklassen der allgemeinen P/T-Netze findet man \mathcal{NP} -Vollständigkeit und in noch wenigeren sogar deterministische Verfahren, die die Erreichbarkeitsfrage in Polynomzeit lösen.

Ein P/T-Netz \mathcal{N} heißt *azyklisch*, wenn der Netzgraph keine Zyklen enthält.

Satz 6.28 Das Erreichbarkeitsproblem für azyklische P/T-Netze ist \mathcal{NP} -vollständig.

Ohne Beweis geben wir folgendes Ergebnis an:

Satz 6.29 Das Äquivalenzproblem ist unentscheidbar, d.h. es ist unentscheidbar, ob zwei gegebene P/T-Netze \mathcal{N}_1 und \mathcal{N}_2 die gleiche Erreichbarkeitsmenge haben, d.h. ob $\mathbf{R}(\mathcal{N}_1) = \mathbf{R}(\mathcal{N}_2)$ gilt.

Einen guten Überblick über weitere Komplexitätsresultate zu Algorithmen und Problemen bei Petrinetzen ist bei Esparza und Nielsen (1994) zu finden.

6.3 Fairness: 5-Philosophen

Fairness ist eine wichtige Erscheinung der Verifikation von Systemen. Fairness ist verwandt mit Lebendigkeit und ist im Zusammenhang mit temporallogischen Spezifikationen wichtig.

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

Fünf Philosophen Ph_1, \dots, Ph_5 sitzen an einem runden Tisch, in dessen Mitte eine Schüssel mit Spaghetti steht (Abb. 6.20). 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, \dots, 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.

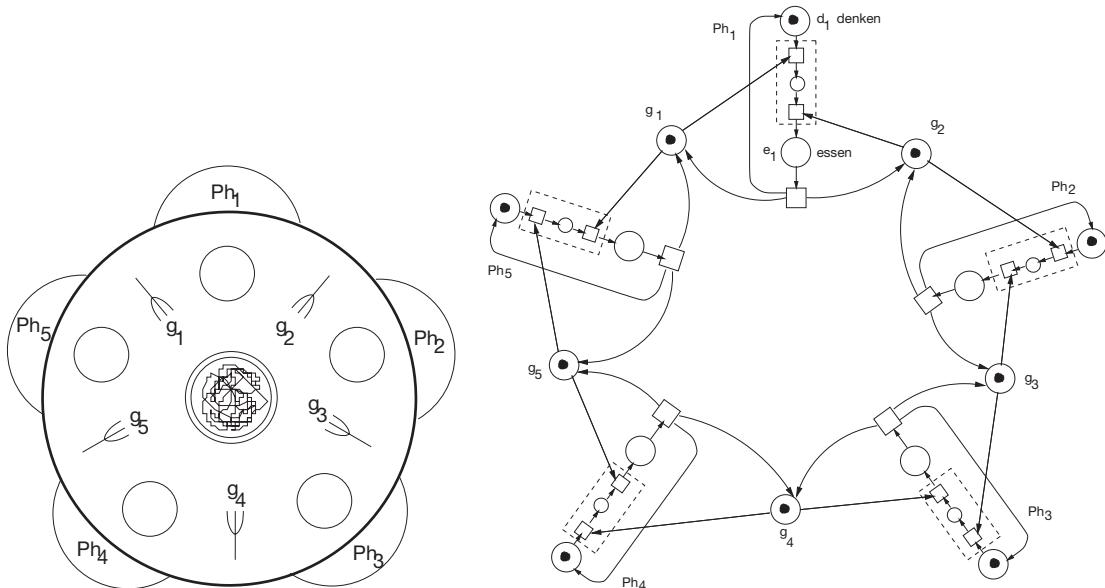


Abbildung 6.20: Die fünf Philosophen am Tisch (nach Dijkstra) und als P/T-Netz

Abbildung 6.20 zeigt eine P/T-Netz-Darstellung 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. 6.20 dargestellten Vergrößerung 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 6.30 Ein P/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 $w \in F_\omega(\mathcal{N})$ jede Transition $t \in T$ unendlich oft vorkommt.

Dabei sei $F_\omega(\mathcal{N}) := \{w = a_1 a_2 a_3 \cdots \in T^\omega \mid \forall i \geq 1 : a_1 a_2 \cdots a_i \in FS(\mathcal{N})\}$.

Zur Definition von $FS(\mathcal{N})$ siehe Definition 6.14 auf Seite 95.

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 lebendigem und fairem Verhalten ist also gekennzeichnet durch den Existenzquantor in a) (alle Transitionen *können* immer wieder schalten) und dem Allquantor in b) (alle Transitionen *müssen* immer wieder schalten).

Außer in einfachen Fällen hat ein System oder Netz kein faires Verhalten. In dem Netz von Abb.6.21 kann natürlich die faire Folge schalten:

$$acbd \quad acbd \quad \dots,$$

aber auch die unfaire:

$$ac \quad ac \quad ac \quad ac \quad \dots$$

Dieses Netz entspricht in gewisser Weise dem folgenden Programm:

```
do   a → c
    □ b → d od
```

Hierbei sind c und d Anweisungen, die a und b nicht verändern und letztere mit **true** initialisiert sind. Die **do...od** - Klammer bezeichnet eine Schleifenanweisung mit nicht-deterministischer Ausführung des Schleifenkörpers und geschützten Anweisungen (*guarded commands*).

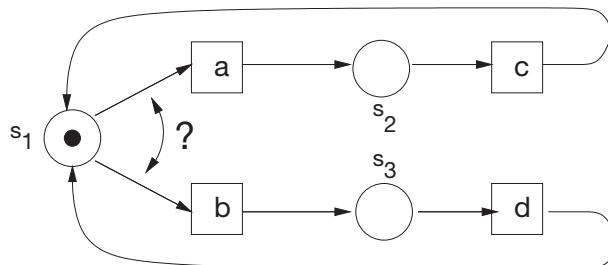


Abbildung 6.21: Netz mit unfairem Verhalten

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 6.31 Es sei $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ ein P/T -Netz und $t \in T$.

- a) *t schaltet produktiv oder verschleppungsfrei oder nach der verschleppungsfreien Schaltregel (finite delay property), wenn*

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

- b) *t schaltet fair, oder nach der fairen Schaltregel (fair firing rule), wenn*

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

- c) *Das Netz \mathcal{N} schaltet nach der produktiven bzw. fairen Schaltregel, wenn dies alle seine Transitionen tun.*

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 terminiert z.B. zwingend nur unter der fairen Schaltregel:

```
b := true;
do   b → x := 1
      □ b → b := false od
```

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

Beispiel 6.32

- a) Das 5-Philosophen-Netz von Abb. 6.20 ist zwar lebendig, hat aber auch unter der verschleppungsfreien Schaltregel kein faires Verhalten (*ac ac ...* ist immer noch möglich). Das Netz verhält sich jedoch fair bei der fairen Schaltregel.
- b) Das 5-Philosophen-Netz von Abb. 6.20 mit der vergröberten, und daher unteilbaren Transition ist lebendig. Es hat aber weder unter der verschleppungsfreien noch unter der fairen Schaltregel ein faires Verhalten.
- c) Das 5-Philosophen-Netz von Abb. 6.20 ohne Vergrößerung ist nicht lebendig. Unter der fairen Schaltregel hat es jedoch ein faires Verhalten.
- d) Verhindert man die Verklemmung durch andere Maßnahmen, z.B. indem man höchstens 4 Philosophen in den Essraum lässt (Abb. 6.22), dann ist das Netz lebendig und fair bei der fairen Schaltregel.

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.

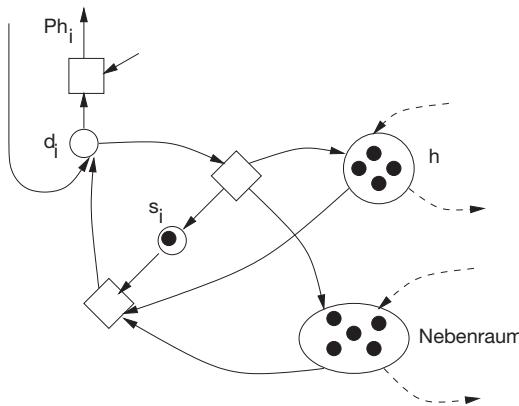


Abbildung 6.22: Philosophenproblem mit Nebenraum

6.4 Prozesse von Petrinetzen

Der Erreichbarkeitsgraph beschreibt eine zustandsorientierte Sichtweise auf das Verhalten eines Systems. Allerdings ist es nicht möglich, den Marken eine Individualität zuzusprechen. Betrachte die Abb. 6.23. In der Entwicklung durch die Schaltfolge

$$p_1 + p_1 \xrightarrow{t_1} p_1 + p_2 \xrightarrow{t_2} p_1 + p_1$$

ist nicht zu erkennen, welche Marke p_1 (von den zwei möglichen) zum Schalten von t_1 genutzt wurde. Die Struktur von Markierungen erlaubt keine Unterscheidung von Marken, sondern drückt nur ihre Multiplizität aus.

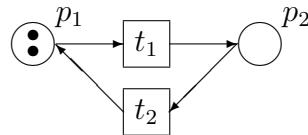


Abbildung 6.23: Ein Beispielnetz

Im Gegensatz dazu berücksichtigt die Prozess-Semantik auch die Historie der Marken. Betrachte das Netz in Abb. 6.23: Eine Verlängerung des Prozesses aus Abb. 6.24(a) kann an zwei verschiedenen Stellen stattfinden, da das Netz nicht 1-sicher ist.⁵ Es ergeben sich die Prozesse in Abb. 6.24(b) und (c). Prozesse unterscheiden die beiden Fortsetzungen, da beide Marken unterscheidbar sind.

⁵ Die Stellen der Prozessnetze aus Abb. 6.24 sind jeweils mit der Stelle des Netzes aus Abb. 6.23 beschriftet, auf die sie abgebildet werden.

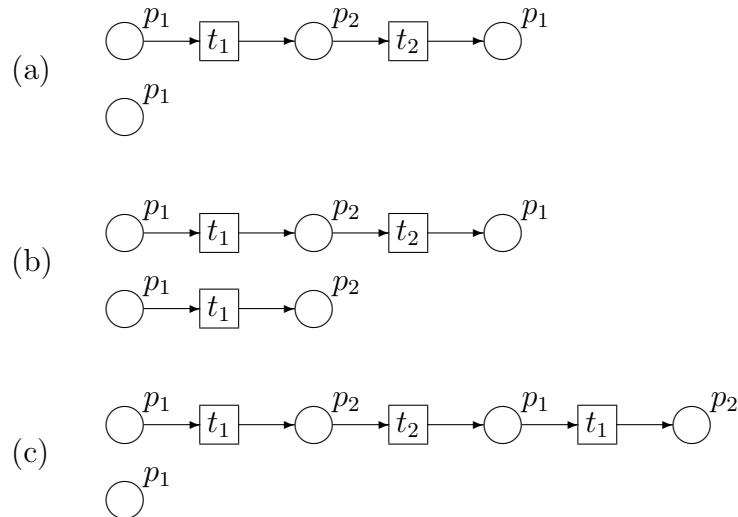


Abbildung 6.24: Prozesse: (a) Grundprozess,
 (b) Verlängerung des Prozesses – erste Variante und
 (c) Verlängerung des Prozesses – zweite Variante

6.4.1 Ereignis- und Kausalnetze

Die Prozesse eines Netzes sind Kausalnetze. Die abgeschwächte Form von Kausalnetzen sind Ereignisnetze (engl. „occurrence nets“), die – im Gegensatz zu Kausalnetzen – noch vorwärtsverzweigende Stellen erlauben, beispielsweise um Systemkonflikte in Prozessen (engl. „branching processes“) darzustellen [Eng91].

Definition 6.33 Ein Petrinetz $N = (B, E, F)$ heißt Ereignisnetz, gdw. der transitive Abschluss F^+ azyklisch ist und für alle $b \in B$ auch $|b^\bullet| \leq 1$ gilt. Gilt zusätzlich noch $|b^\bullet| \leq 1$, so ist N ein Kausalnetz.

Für ein Ereignisnetz $N = (B, E, F)$ definiere die Ordnungen $<, \leq \subseteq (B \cup E)^2$ durch $< := F^+$ und $\leq := F^*$. Die Menge der Vorgänger eines Elements $y \in B \cup E$ ist $\downarrow y := \{x \mid x < y\}$. Das Ereignisnetz N heißt *vorgänger-endlich*, wenn für jedes $b \in B$ die Vorgängermenge $(_- < b)$ endlich ist.

Da ein Ereignisnetz vorwärtsverzweigt, können Netzelemente in Konflikt stehen, nämlich dann, wenn eine Stelle $b \in B$ existiert, von der zwei Konflikttereignisse $e_1, e_2 \in b^\bullet$ ausgehen. Dies wird durch die symmetrische Konfliktrelation $\# \subseteq (B \cup E)^2$ beschrieben:

$$x \# y \iff \exists b \in B : \exists e_1, e_2 \in b^\bullet : e_1 \neq e_2 \wedge e_1 \leq x \wedge e_2 \leq y$$



Kausalnetze sind demnach immer konfliktfrei: $\# = \emptyset$.

Sei $R \subseteq A \times A$ eine symmetrische, reflexive Relation. $K \subseteq A$ heißt *Klique* bezüglich R , gdw. alle Elemente in Relation stehen, d.h. für alle $x, y \in K$ gilt $x R y$. Eine maximale Klique heißt *Bezirk*. Die Menge aller Bezirke von R wird durch $\text{BEZ}(R)$ notiert.

Sei die Relation $<$ gegeben. Die symmetrischen und reflexiven Relationen **li** und **co** sind definiert als:

$$\mathbf{li} := < \cup <^{-1} \cup id_A \quad \text{und} \quad \mathbf{co} := \bar{\mathbf{li}} \cup id_A$$

Die Menge der Bezirke bezüglich **li** werden als *Linien*, die Bezirke bezüglich **co** als *Schnitte* bezeichnet.

6.4.2 Prozesse

Am Anfang dieses Kapitels haben wir das dynamische Verhalten (auch Ablauf genannt) eines P/T-Netzes durch eine Folge von Transitionen dargestellt. In einem Prozess (auch verteilter Ablauf genannt) wird die Ausführung einer Transition durch eine Transition repräsentiert zusammen mit ihrem Vor- und Nachbereich und den verbindenden Pfeilen. In entsprechender aber umgekehrter Weise wurden die Aktionen von Abbildung 6.4 auf Seite 83 zu dem Netz in Abbildung 6.5 komponiert. Nun gehen wir von dem Netz aus und bilden die Aktionen, was dann zu einem Prozess in Form eines Kausalnetzes führt. Dabei wird wie in Abbildung 6.25 (a) ein Platz p mit k Marken durch k verschiedene

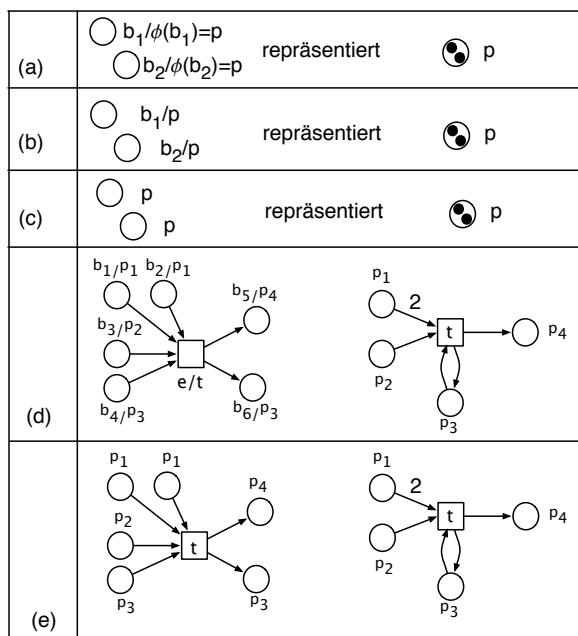


Abbildung 6.25: Aktion einer Transition

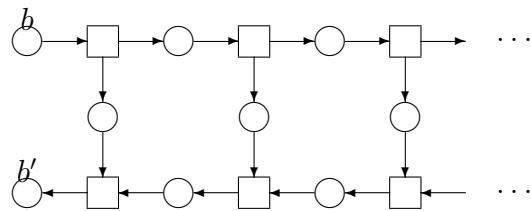


Abbildung 6.26: Ein unendlicher, nicht vorgängerendlicher Prozess

Plätze b_i mit dem Etikett $\phi(b_i) = p$ dargestellt. Die b_i sind also die eindeutigen Bezeichner der Plätze des Kausalnetzes, während die Etikette $\phi(b_i)$ nicht unbedingt eindeutig sein müssen. Abkürzend schreiben wir dafür auch b_i/p wie in (b) oder einfach nur p wie in (c) (und auch schon in Abbildung 6.24). Eine Transition t wird dann wie in (d) mit ihrem Vorbereich $\bullet t$ und ihrem Nachbereich t^\bullet dargestellt, wobei der Platz entsprechend dem Kantengewicht $W(p, t)$ bzw. $W(t, p)$ mehrfach dargestellt wird. Vor- und Nachbereich sind hier immer disjunkt, da sie die Lage der Marke vor bzw. nach dem Schalten darstellen. Da in Abbildung 6.25 (d) der Platz p_3 sowohl im Vor- als auch im Nachbereich von t liegt, entstehen dazu in der zugehörigen Aktion links daneben zwei Plätze b_4/p_3 und b_6/p_3 was die Marke in p_3 vor und nach dem Schalten repräsentiert.

In Abbildung 6.30 (b) ist dieser Weise ein Prozess des Netzes aus Abb. 6.29 konstruiert worden. Eine auf einer solchen Konstruktion basierende Prozessdefinition folgt in Definition 6.35. Zunächst wird jedoch eine Definition behandelt, bei der das Etikett eines Platzes $\phi(p)$ oder einer Transition $\phi(t)$ explizit als Abbildung vorkommt.

Nach [BM84] wird ein Prozess (engl. „run“) eines Netzes N definiert als ein Kausalnetz $R = (B, E, F_R)$ zusammen mit einem Abbildungspaar $\phi = (\phi_P : B \rightarrow P, \phi_T : E \rightarrow T)$. Diese Abbildung heißt *Faltung* und ist auch eine Faltung im Sinne von Definition 6.10, falls die Kantengewichte höchstens den Wert 1 haben.

Kausalnetze, die einen Prozess eines P/T-Netzes darstellen sollen, müssen vorgängerendlich sein, denn nur dann kann eine Bedingung in endlicher Zeit realisiert werden. Abbildung 6.26 zeigt einen nicht vorgänger-endlichen Kausalnetz, da Beispielsweise b' unendlich viele Vorgänger besitzt. Ist der Prozesse R vorgänger-endlich, so kann er als Grenzwert endlicher Prozesse dargestellt werden.

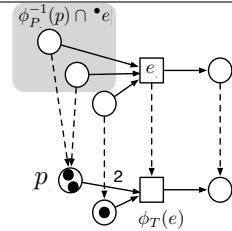
Um die Darstellung zu vereinfachen, beschränken wir die folgende Darstellung auf T -schlichte Netze. Sind die Netze T -schlicht, gilt $\bullet t \neq \emptyset$, d.h. es werden keine Marken aus dem „Nichts“ erzeugte. ${}^\circ R$ bzw. ${}^o R$ bezeichnen die minimalen bzw. maximalen Elemente eines Ereignisnetzes.

Definition 6.34 Sei $N = (P, T, F, W, \mathbf{m}_0)$ ein P/T-Netz, $R = (B, E, F_R)$ ein vorgängerendliches Ereignisnetz und ϕ ist ein Abbildungspaar aus $\phi_P : B \rightarrow P$ und $\phi_T : E \rightarrow T$, dann ist (R, ϕ) ein Verzweigungsprozess des Netzes N , falls die folgenden Eigenschaften gelten:

1. Die Struktur von R „passt“ zu N : $(x, y) \in F_R \Rightarrow (\phi(x), \phi(y)) \in F$
2. Die Stellen ${}^\circ R$ beschreiben \mathbf{m}_0 : $\forall p \in P : \mathbf{m}_0(p) = |\phi^{-1}(p) \cap {}^\circ R|$

3. Die Prozessabbildung ϕ ist verträglich mit der Kantengewichtung:

$$\begin{aligned} \forall e \in E \quad \forall p \in {}^{\bullet}\phi_T(e) : W(p, \phi_T(e)) = |\phi_P^{-1}(p) \cap {}^{\bullet}e| \\ \forall e \in E \quad \forall p \in \phi_T(e)^{\bullet} : W(\phi_T(e), p) = |\phi_P^{-1}(p) \cap e^{\bullet}| \end{aligned}$$



Ist R zudem noch ein Kausalnetz, so heißt (R, ϕ) ein Prozess des Netzes N .

Die Menge aller Verzweigungsprozesse von N wird mit $\text{BranProc}(N)$ bezeichnet, die Menge aller Prozesse mit $\text{Proc}(N)$.

Ist R ein Prozess, so muss R ein endlich verzweigendes Kausalnetz sein. Im Beispielnetz von Definition 6.34 ist $\mathbf{m}_0(p) = 2 = |\phi^{-1}(p) \cap {}^{\circ}R|$ und $W(p, \phi_T(e)) = 2 = |\phi_P^{-1}(p) \cap {}^{\bullet}e|$. Als Alternative zu Definition 6.34 kann ein Prozess (R, ϕ) auch durch eine Konstruktion erzeugt werden, die sich an den Schaltvorgängen im Netz N orientiert. Die Konstruktion erzeugt aus einem bestehenden Prozess induktiv einen weiteren, indem zum alten Prozessnetz – im Einklang mit dem Netz N – weitere Transitionen und Stellen hinzugefügt werden.

Definition 6.35 Sei $N = (P, T, F, W, \mathbf{m}_0)$ ein P/T-Netz.

1. Sei C eine Menge mit $\phi(c) \in P$ derart, dass $\phi^{-1}(p) = \mathbf{m}_0(p)$ für alle $p \in P$.
 $((C, \emptyset, \emptyset), \phi)$ ist ein Prozess (Prozessanfang). Für das Netz in Abbildung 6.27 (a) ist dies z.B. die Menge c_1, c_2, c_3, c_4 mit $\phi(c_1) = p_1, \phi(c_2) = p_1, \phi(c_3) = p_1, \phi(c_4) = p_4$ in Abbildung 6.27 (b). Streicht man hier die anderen Transitionen und Plätze, so erhält man den Anfangsprozess, der der Anfangsmarkierung entspricht.
2. Sei $(R, \phi) = ((B, E, F_K), \phi)$ ein (schon konstruierter) Prozess und $t \in T$ eine Transition. Sein maximaler Schnitt R° enthalte für jeden Eingangsstelle $p \in {}^{\bullet}t$ mindestens $W(p, t)$ Elemente b_i mit $\phi(b_i) = p$. Für den Prozess in Abbildung 6.27 (b) und die Transition c von (a) ist dies der Fall: R° enthält 2 Elemente b_i mit $\phi(b_i) = p_1$ und ein Element b_i mit $\phi(b_i) = p_4$. Es werden (außer dem Anfangsprozess) hier nur noch die Etikette angeben.
3. Der Prozess wird nun verlängert, indem eine neue Transition e mit $\phi(e) = t$ eingefügt wird, die die unter 2. genannten Plätze als Eingangsplätze enthält. Als Ausgangsplätze werden für jedes $p \in t^{\bullet}$ eine Anzahl von $W(t, p)$ neuen Plätzen b_i mit $\phi(b_i) = p$ angefügt. Dies ist für das Beispiel in Abbildung 6.27 (c) ausgeführt (hier ist $t = c$ und $p = p_3$).

Der in Abbildung 6.27 (c) gegebene Prozess kann verlängert werden (oder kürzer ausfallen). Wie die Abbildung 6.27 (d) zeigt, kann er jedoch auch ganz anders beginnen. Man beachte, dass die Transitionen a und c in (c) auf einer Linie liegen, also sequentiell schalten, während sie in (d) auf einem Schnitt liegen, d.h. nebenläufig schalten!

Damit beide Definitionen korrespondieren, ist für unendliche Prozesse nach Definition 6.34 die Eigenschaft der Vorgängerendlichkeit notwendig. Nach [GR83] ist jeder Prozess nach der Konstruktion in Definition 6.35 auch ein Prozess im Sinne der Definition 6.34 – und umgekehrt.

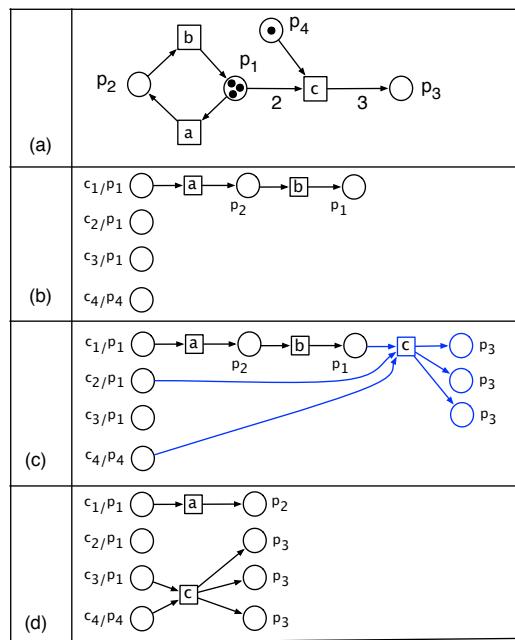
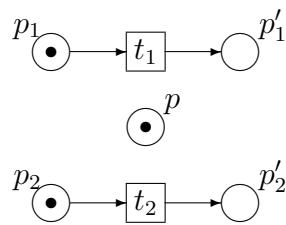
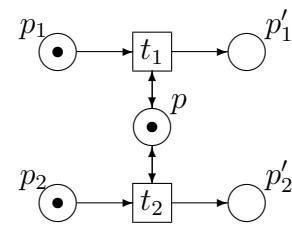


Abbildung 6.27: Zur 2. Prozessdefinition

Prozesse sind ausdrucksstark genug, um Nebenläufigkeit von Permutierbarkeit zu unterscheiden. So besitzt das Netz N_1 aus Abb. 6.28 den in Abb. 6.30 (a) dargestellten Prozess, während das Netz N_2 aus Abb. 6.29 die in Abb. 6.30 (b) und (c) dargestellten Prozesse besitzt. Die Netzknoten sind wieder nur mit ihrem Etikett, d.h. dem Bild der Abbildung ϕ beschriftet.


 Abbildung 6.28: Das Petrinetz N_1

 Abbildung 6.29: Das Petrinetz N_2

Betrachtet man die Schnitte und Linien bezüglich **li** und **co**, die sich aus der Ordnung $<$ ergeben, so ist es sinnvoll, diese auf reine Stellen bzw. auf reine Transitionsmengen zu beschränken.

Die Menge der P -Schnitte eines Prozesses R wird im Folgenden durch \mathcal{C}_R notiert. Insbesondere ergibt sich die Menge \mathcal{C}_R aller Stellen-Schnitte, kurz: P -Schnitte, als die Menge der möglichen Markierungen eines Prozesses:

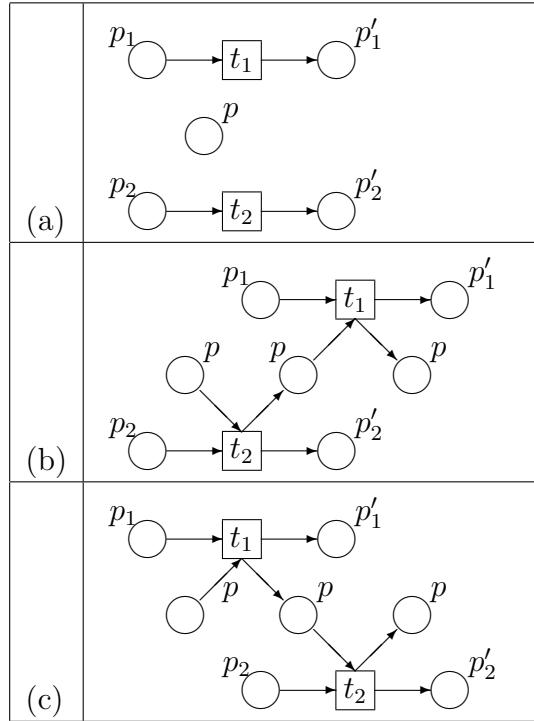


Abbildung 6.30: (a) Prozess des Netzes N_1 aus Abb. 6.28 sowie
(b) und (c) Prozesse des Netzes N_2 aus Abb. 6.29

Theorem 6.36 Sei $(R, \phi) \in \text{Proc}(N)$ ein Prozess von N , dann ist jeder Stellenschnitt C eine erreichbare Markierung:

$$C \in \mathcal{C}_R \Rightarrow \phi(C) \in RS(N)$$

Die Transitions-Schnitte, kurz: T -Schnitte, sind die nebenläufigen Transitionen.

6.4.3 Fortschritt und Fairness

Das Netz in Abbildung 6.31 a) erlaubt die Wiederholung der Aktionen a und b , d.h. $abababab\dots$ ist eine aktivierte Schaltfolge. Für jeden Zustand, der während dieser Schaltfolge erreicht wird, ist die Transition c aktiviert, ohne je zu schalten. Um zu erzwingen, dass c schaltet, müssen wir (in der Schaltfolgen-Semantik) die produktive Schaltregel (Definition 6.31) anwenden. In der Prozess-Semantik stellt sich dies in viel natürlicherer Weise als Fortschrittseigenschaft dar.

Definition 6.37 Sei ein P/T-Netz N gegeben. Ein Prozess (R, ϕ) mit (B, E, F_K) respektiert die Fortschrittseigenschaft (engl. progress) einer Teilmenge der Transition $T^p \subseteq T$, wenn der maximale Schnitt R° nur Transitionen aus $T \setminus T^p$ aktiviert:

$$\forall e \in E \quad \forall p \in P : W(p, \phi_T(e)) = |\phi_P^{-1}(p) \cap R^\circ| \Rightarrow t \notin T^p$$

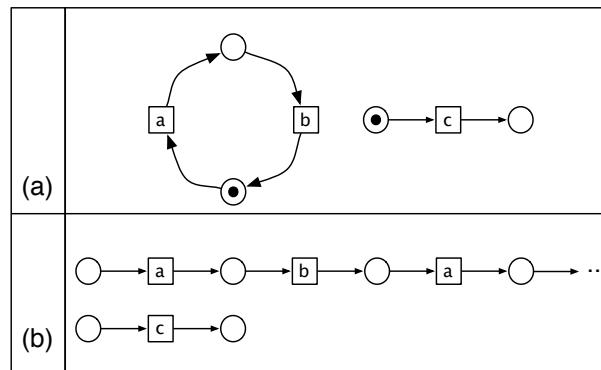


Abbildung 6.31: Zur Fortschrittseigenschaft

Eine Transition $t \in T^p$ heißt Fortschritttransition. Positiv formuliert bedeutet Fortschritt, dass eine dauerhaft aktivierte Fortschritttransition $t \in T^p$ auch schalten muss. Abbildung 6.31 b) zeigt einen Prozess (hier der einzige maximale Prozess) des Netzes aus Abbildung 6.31 a). Falls c eine Fortschritttransition ist, d.h. $c \in T^p$ muss sie in jedem Prozess vorkommen.

Die produktive Schaltregel (Definition 6.31) in der Schaltfolgen-Semantik entspricht also dem Fall $T = T^p$ als Fortschrittseigenschaft.

Eigenschaft 6.38 *Wenn N ein lebendiges Netz ist und für alle Transitionen die Fortschrittseigenschaft gilt, dann sind alle Prozesse, welche die Fortschrittseigenschaft respektieren, unendlich.*

7 Systemverifikation durch Petrinetze

Es existieren verschiedene Verifikationsmethoden für Petrinetze:

- a) enumerative Methoden: Der Erreichbarkeitsgraph wird analysiert (normaler Weise nur bei beschränkten Netzen möglich).
- b) Transformationen: Das Netz wird – unter Beibehaltung der untersuchten Eigenschaft – vereinfacht (Reduktion).
- c) strukturelle Analyse: Hierbei wird die Netzstruktur herangezogen, um Eigenschaften – unabhängig von der Initialmarkierung – nachzuweisen. Dies geschieht u.a. durch Methoden der linearen Algebra (P-Invarianten, T-Invarianten) oder durch graphenbasierte Methoden.

7.1 Verifikation beschränkter Netze

Ist das Netzsystem $(\mathcal{N}, \mathbf{m}_0)$ beschränkt, dann ist $\mathbf{R}(\mathcal{N}, \mathbf{m}_0)$ endlich und kann konstruiert werden. In diesem Fall sind die in diesem Skript behandelten Methoden des Model-Checking anwendbar, falls die zu prüfende Eigenschaft in temporaler Logik spezifiziert ist. Wir wählen hier einen elementareren Ansatz, indem wir die Eigenschaften direkt im Zustandsgraphen (das ist der Erreichbarkeitgraph) formulieren.

Der Algorithmus 7.1 liefert eine Berechnung dieses Graphen. Die Kennzeichnung der Knoten (durch Färben) in Schritt 2.1 bewirkt, dass jede Markierung nur einmal bearbeitet wird. In Schritt 2.2.3 werden nur solche Markierungen zu V hinzugefügt, die noch nicht vorhanden sind.

Algorithmus 7.1 (Berechnung des Erreichbarkeitsgraphen)

Input - Das Netzsystem $(\mathcal{N}, \mathbf{m}_0)$

Output - Der gerichtete Graph $\text{RG}(\mathcal{N}, \mathbf{m}_0) = (V, E)$, falls das Netzsystem beschränkt ist.

1. Initialisiere $\text{RG}(\mathcal{N}, \mathbf{m}_0) = (\{\mathbf{m}_0\}, \emptyset)$; \mathbf{m}_0 sei ungefärbt;
 2. **while** Es gibt ungefärbte Knoten in V . **do**
 - 2.1 Wähle einen ungefärbten Knoten $\mathbf{m} \in V$ und färbe ihn.
 - 2.2 **for** Für jede in \mathbf{m} aktivierte Transition t **do**
 - 2.2.1 Berechne \mathbf{m}' mit $\mathbf{m} \xrightarrow{t} \mathbf{m}'$;
 - 2.2.2 **if** Es gibt einen Knoten $\mathbf{m}''' \in V$ derart, dass $\mathbf{m}''' \xrightarrow{\sigma} \mathbf{m}'$ und $\mathbf{m}''' < \mathbf{m}'$.
then Der Algorithmus terminiert ohne Ergebnis.;
(Das Netzsystem ist unbeschränkt.)
 - 2.2.3 **if** Es gibt keinen Knoten $\mathbf{m}''' \in V$ derart, dass $\mathbf{m}''' = \mathbf{m}'$
then $V := V \cup \{\mathbf{m}'\}$, wobei \mathbf{m}' ein ungefärbter Knoten sei.
 - 2.2.4 $E := E \cup \{(\mathbf{m}, t, \mathbf{m}')\}$
 3. Der Algorithmus terminiert mit Ergebnis. ($\text{RG}(\mathcal{N}, \mathbf{m}_0)$ ist der Erreichbarkeitsgraph.)
-

Die Konstruktion des Erreichbarkeitsgraphen ist sehr aufwendig, da er im Verhältnis zur Größe des Netzes exponentiell viele Knoten enthalten kann (siehe [Val92]). Darauf werden wir im Abschnitt 7.3 zurückkommen.

Der Erreichbarkeitsgraph $RG((\mathcal{N}, \mathbf{m}_0))$ eines unbeschränkten Netzsystems $(\mathcal{N}, \mathbf{m}_0)$ ist nicht endlich. Karp and Miller [KM69] haben bewiesen, wie man dies durch die Bedingung feststellen kann, die in Schritt 2.2.2 von Algorithmus 7.1 als Abbruchkriterium dient. Dies wird in folgendem Satz präzisiert.

Satz 7.1 Ein Netzsystem \mathcal{N} ist genau dann unbeschränkt, wenn es zwei erreichbare Markierungen $\mathbf{m}_1, \mathbf{m}_2 \in \mathbf{R}(\mathcal{N}, \mathbf{m}_0)$ gibt, die folgende Bedingungen erfüllen:

- a) $\exists \sigma \in T^* : \mathbf{m}_1 \xrightarrow{\sigma} \mathbf{m}_2$
- b) $\mathbf{m}_1 \not\leq \mathbf{m}_2$

Beweis: Gilt die Bedingung, dann kann σ auch in \mathbf{m}_2 schalten, da ja mindestens so viele Marken wie in \mathbf{m}_1 vorhanden sind: $\mathbf{m}_1 \xrightarrow{\sigma} \mathbf{m}_2 \xrightarrow{\sigma} \mathbf{m}_3$. Außerdem gilt $\mathbf{m}_2 - \mathbf{m}_1 = \mathbf{m}_3 - \mathbf{m}_2$ und damit $\mathbf{m}_2 \not\leq \mathbf{m}_3$. Also gilt die Bedingung auch für das Paar $\mathbf{m}_2, \mathbf{m}_3$ und man erhält per Induktion $\mathbf{m}_1 \xrightarrow{\sigma} \mathbf{m}_2 \xrightarrow{\sigma} \mathbf{m}_3 \xrightarrow{\sigma} \mathbf{m}_4 \xrightarrow{\sigma} \mathbf{m}_5 \xrightarrow{\sigma} \dots$.

Wegen $\mathbf{m}_1 \not\leq \mathbf{m}_2 \not\leq \mathbf{m}_3 \not\leq \mathbf{m}_4 \not\leq \mathbf{m}_5 \not\leq \dots$ ist das System unbeschränkt.

Ist umgekehrt der Erreichbarkeitsgraph unendlich, dann gibt es nach dem Lemma von König eine unendliche Folge $\mathbf{m}_1 \xrightarrow{*} \mathbf{m}_2 \xrightarrow{*} \mathbf{m}_3 \xrightarrow{*} \mathbf{m}_4 \xrightarrow{*} \mathbf{m}_5 \xrightarrow{*} \dots$ mit $\mathbf{m}_1 \in \mathbf{R}(\mathcal{N}, \mathbf{m}_0)$ und $\mathbf{m}_1 \not\leq \mathbf{m}_2 \not\leq \mathbf{m}_3 \not\leq \mathbf{m}_4 \not\leq \mathbf{m}_5 \not\leq \dots$. Also ist die Bedingung erfüllt. \square

Theorem 7.2 Folgende Aussagen sind für gegebenes $(\mathcal{N}, \mathbf{m}_0)$ äquivalent:

1. Die Menge der erreichbaren Markierungen $R(\mathcal{N}, \mathbf{m}_0)$ ist endlich.
2. Der Erreichbarkeitsgraph $RG(\mathcal{N}, \mathbf{m}_0)$ ist endlich.
3. \mathcal{N} ist in der Markierung \mathbf{m}_0 beschränkt.
4. Es gibt keine Markierungen $\mathbf{m}_1, \mathbf{m}_2$ mit $\mathbf{m}_2 \not\geq \mathbf{m}_1$ und $\mathbf{m}_0 \xrightarrow{*} \mathbf{m}_1 \xrightarrow{*} \mathbf{m}_2$.

Beweis: Als Übung.

\square

Auch vergleichsweise kleine Komponenten eines System müssen rechnergestützt verifiziert werden. Daher hat das Forschungsgebiet *rechnergestützte Verifikation* (computer-aided verification) große praktische Bedeutung. Es folgt als Beispiel ein kleines Netzsystem, das nur schwer ohne Rechnerunterstützung zu analysieren ist.

Beispiel 7.3 Abb. 7.1 zeigt ein einfaches Netzsystem: Teile einer Produktionsanlage sollen von *STORE 1* zu *STORE 2* oder *STORE 3* transportiert werden. Das durch die Plätze $\{B, C, E, F\}$ erzeugte Teilnetz spezifiziert eine Regel, nach der die Teile auf die Lager zu verteilen sind. Der angegebene Erreichbarkeitsgraph ist schwer zu analysieren, obwohl er „strukturiert“ dargestellt ist. Man prüfe mit ihm, dass folgende Regel implementiert wurde: die Teile werden gleichmäßig auf beide Lager verteilt, es darf aber bis zu 4 konsekutive Zuteilungen zu einem Speicher geben, was langfristig wieder auszugleichen ist.

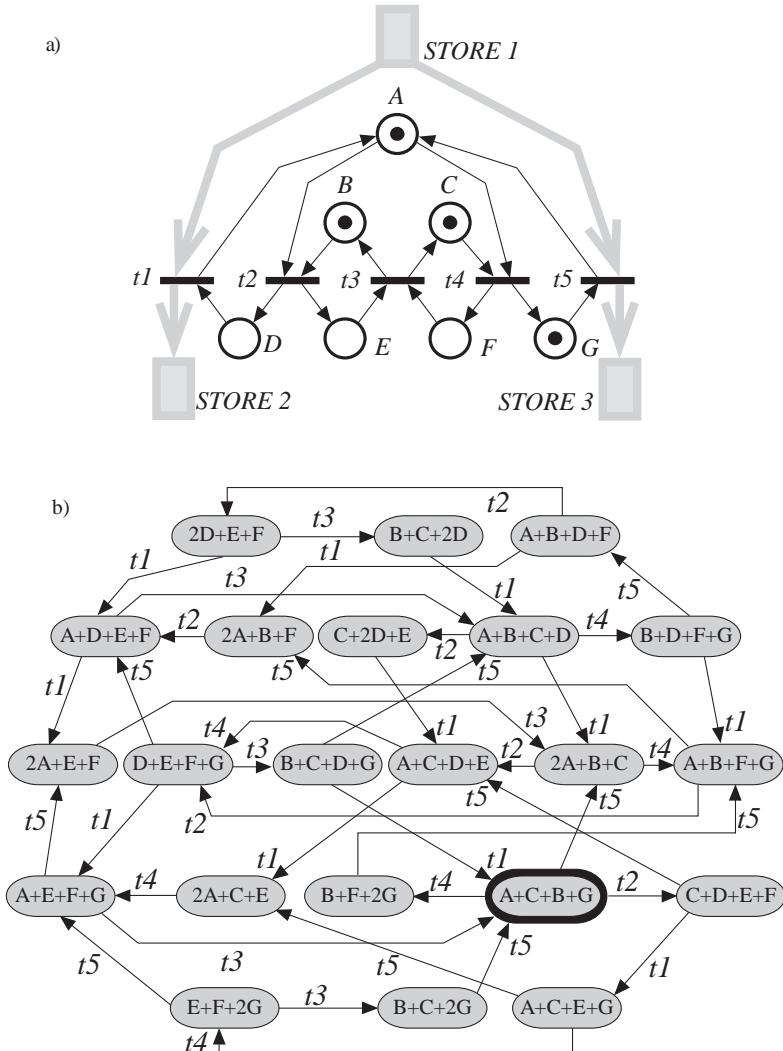


Abbildung 7.1: Ein kleines Netzsystem (a) mit komplexem Erreichbarkeitsgraph in (b)

7.1.1 Markierungs-Invarianzeigenschaft

Es wird nun ein Verfahren beschrieben, das Systemeigenschaften aus zwei Klassen in polynomieller Zeit (im Verhältnis zur Größe des Erreichbarkeitsgraphen) entscheidet. Die Systemeigenschaften werden durch *Markierungsprädikate* Π spezifiziert.

Markierungsprädikate sind aussagenlogische Formeln, deren Atome Ungleichungen der Form:

$$\Pi(\mathbf{m}) = \left[\sum_{p \in P} k_p \mathbf{m}(p) \leq k \right] \quad k_p, k \in \mathbb{Q}$$

Beispiel: $\Pi(\mathbf{m}) = (2 \cdot \mathbf{m}(p_1) + \mathbf{m}(p_2)) \leq 3 \vee \mathbf{m}(p_3) \leq 1$

Beispiel: $\Pi(\mathbf{m}) = \mathbf{m}(p_1) = 1$ (Gleichheit = denke man sich hier mittels \leq ausgedrückt.)

Die erste Klasse von Spezifikationen heißt *Markierungs-Invarianzeigenschaften* (engl. marking invariance property).

Definition 7.4 Sei Π ein Markierungsprädikat.

Eine Markierungs-Invarianzeigenschaft eines Netzsysteams $(\mathcal{N}, \mathbf{m}_0)$ ist ein Prädikat der Form:

- $\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) : \Pi(\mathbf{m})$ oder
- $\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) \exists t \in T : \Pi(\mathbf{m})$

Beispiele dazu sind:

1) *k-Beschränktheit* (*k*-boundedness) eines Platzes p :

$$\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) . \mathbf{m}(p) \leq k$$

2) *Markierungs-Ausschluss* (marking mutual exclusion) zwischen p und p' :

$$\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) . ((\mathbf{m}(p) = 0) \vee (\mathbf{m}(p') = 0))$$

3) *Verklemmungsfreiheit* (deadlock-freeness):

$$\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) . \exists t \in T . W(\bullet, t) \leq \mathbf{m}$$

Algorithmus 7.2 (Entscheiden einer Markierungs-Invarianzeigenschaft)

Input - Der Erreichbarkeitsgraph $\text{RG}(\mathcal{N}, \mathbf{m}_0)$; die Markierungs-Invarianzeigenschaft Π .

Output - TRUE falls die Eigenschaft Π erfüllt ist; FALSE falls die Eigenschaft Π nicht erfüllt ist.

1. Initialisiere alle Elemente von $\mathbf{R}(\mathcal{N}, \mathbf{m}_0)$ als ungefärbt.
 2. **while** Es gibt einen ungefärbten Knoten $\mathbf{m} \in \mathbf{R}(\mathcal{N}, \mathbf{m}_0)$ **do**
 - 2.1 Wähle einen ungefärbten Knoten $\mathbf{m} \in \mathbf{R}(\mathcal{N}, \mathbf{m}_0)$ und färbe ihn.
 - 2.2 **if** \mathbf{m} erfüllt nicht Π .

 then return FALSE (Die Eigenschaft Π ist nicht erfüllt.)
 3. Return TRUE
-

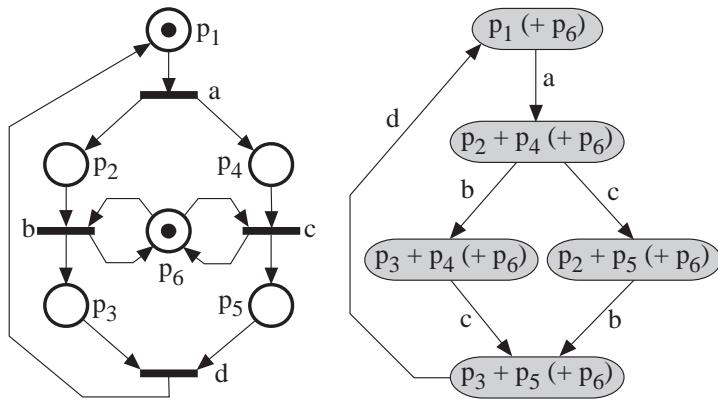


Abbildung 7.2: Beschränktes, lebendiges und reversibles Netz mit Erreichbarkeitsgraph

Markierungs-Invarianzeigenschaften können mit Algorithmus 7.2 entschieden werden, dessen Zeitkomplexität linear in Bezug auf die Größe von $\mathbf{R}(\mathcal{N}, \mathbf{m}_0)$ ist, denn jeder Knoten wird höchstens einmal aufgesucht. Falls der Algorithmus erfolgreich terminiert, dann erfüllen alle von \mathbf{m}_0 aus erreichbaren Markierungen das Prädikat Π . Falls der Algorithmus in Schritt 2.2 terminiert, dann gibt es einen Pfad in $\text{RG}(\mathcal{N}, \mathbf{m}_0)$, der von \mathbf{m}_0 ausgeht und in einer Markierung endet, die Π nicht erfüllt.

Beispiel 7.5 Wir wollen für das Netz in Abb. 7.2 testen, ob Markierungsausschluss der Plätze p_5 und p_6 gilt:

$$\Pi(\mathbf{m}) = (\mathbf{m}(p_5) = 0) \vee (\mathbf{m}(p_6) = 0)$$

Die Anwendung des Algorithmus 7.2 zur Überprüfung von $\Pi(\mathbf{m})$ beginnt damit, dass alle Elemente von $\mathbf{R}(\mathbf{m}_0)$ ungefärbt sind (Schritt 1).

Dann werden die Markierungen nacheinander geprüft bis $p_2 + p_5 + p_6$ erreicht wird. Hier wird das Prädikat Π als ungültig festgestellt und der Algorithmus terminiert mit FALSE.

7.1.2 Lebendigkeits-Invarianzeigenschaften

Die zweite Klasse von Spezifikationen heißt *Lebendigkeits-Invarianzeigenschaften* (engl. liveness invariance property).

Definition 7.6 Sei Π ein Markierungsprädikat.

Eine Lebendigkeits-Invarianzeigenschaft eines Netzesystems $(\mathcal{N}, \mathbf{m}_0)$ ist ein Prädikat der Form

$$\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) . \exists \mathbf{m}' \in \mathbf{R}(\mathbf{m}) . \Pi(\mathbf{m}')$$

Beispiele dazu sind:

- 1) *Lebendigkeit von t* (liveness of t):

$$\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) . \exists \mathbf{m}' \in \mathbf{R}(\mathbf{m}) . W(\bullet, t) \leq \mathbf{m}'$$

2) \mathbf{m}_H ist *Rücksetzzustand* (home state):

$$\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) . \exists \mathbf{m}' \in \mathbf{R}(\mathbf{m}) . \mathbf{m}' = \mathbf{m}_H$$

3) *Reversibilität* (reversibility):

$$\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0) . \exists \mathbf{m}' \in \mathbf{R}(\mathbf{m}) . \mathbf{m}' = \mathbf{m}_0$$

Diese Eigenschaften lassen sich nicht wie in Algorithmus 7.2 durch lineares Durchsuchen der Erreichbarkeitsmenge überprüfen. Vielmehr muss von jeder erreichbaren Markierung eine von dieser erreichbare Markierung gefunden werden, die Π erfüllt.

Der im folgende entwickelte Algorithmus 7.3, der Lebendigkeits-Invarianzeigenschaften prüft, greift auch das Konzept der *strengen Zusammenhangskomponente* (strongly connected component) eines Graphen zurück.

Definition 7.7 Ein Pfad von \mathbf{m}_1 nach \mathbf{m}_k in einem Erreichbarkeitsgraphen $\text{RG}(\mathcal{N}, \mathbf{m}_0) = (V, E)$ ist eine Folge $\mathbf{m}_1 \dots \mathbf{m}_i \mathbf{m}_{i+1} \dots \mathbf{m}_k$ ($k \geq 2$) seiner Knoten mit $(\mathbf{m}_i, t_i, \mathbf{m}_{i+1}) \in E$ für alle $i \in \{1, \dots, k-1\}$ und jeweils ein $t_i \in T$.

Ein Teilgraph von $\text{RG}(\mathcal{N}, \mathbf{m}_0)$ heißt streng zusammenhängend, falls er entweder nur aus einem Knoten besteht oder für je zwei verschiedene seiner Knoten \mathbf{m}_i und \mathbf{m}_j ein Pfad von \mathbf{m}_i nach \mathbf{m}_j und von \mathbf{m}_j nach \mathbf{m}_i existiert.

Seine Knotenmenge heißt Zusammenhangskomponente (ZK).

Eine maximale ZK heißt strenge Zusammenhangskomponente (SZK) von $\text{RG}(\mathcal{N}, \mathbf{m}_0)$.

Sie heißt triviale Zusammenhangskomponente, falls sie nur aus einem Knoten ohne Schleife besteht.

Sie heißt terminale strenge Zusammenhangskomponente, falls von keinem ihrer Knoten eine Kante zu einem Knoten \mathbf{m} ausgeht, der nicht in ihr liegt.

Die strengen Zusammenhangskomponenten eines gerichteten Graphen (V, E) können in der Zeitkomplexität $O(|V| + |E|)$ berechnet werden (siehe [Meh84], Algorithmus von Tarjan).

Lemma 7.8 Die Knotenmengen verschiedener SZKs sind disjunkt.

Beweis: Als Übung. □

Da die Knotenmengen verschiedener SZKs disjunkt sind, kann man sie in eindeutiger Weise zu einem Knoten zusammenziehen. Wenn man die verbleibenden Kanten sinngemäß überträgt, erhält man seinen *reduzierten Graphen*.

Seine Kantenmenge E_c ist dann die Menge aller (C_i, t, C_j) , so dass $C_i \neq C_j$ gilt (keine Schleifen) und es eine Kante $(\mathbf{m}_i, t, \mathbf{m}'_j) \in E$ gibt, wobei \mathbf{m}_i in der SZK C_i und \mathbf{m}'_j in der SZK C_j liegt.

Definition 7.9 Der reduzierte Graph $\text{RG}^c(\mathcal{N}, \mathbf{m}_0) = (V_c, E_c)$ eines Erreichbarkeitsgraphen $\text{RG}(\mathcal{N}, \mathbf{m}_0) = (V, E)$ besteht aus:

- Die SZKs von $\text{RG}(\mathcal{N}, \mathbf{m}_0)$ sind die Knotenmenge, d.h. $V_c := \{C_1, \dots, C_r\}$.
- Die Kantenmenge ist $E_c := \{(C_i, t, C_j) \mid \exists (\mathbf{m}_i, t, \mathbf{m}'_j) \in E : \mathbf{m}_i \in C_i \wedge \mathbf{m}'_j \in C_j \wedge C_i \neq C_j\}$.

Als Anfangsknoten wird der Knoten definiert, der aus derjenigen SZK entstanden ist, die den Anfangsknoten enthält (vorausgesetzt, der Ausgangsgraph hatte einen solchen).

Beispiel 7.10 Abb. 7.3 b) zeigt die strengen Zusammenhangskomponenten des Graphen von Abb. 7.3 a), von denen zwei terminal sind. Zwei seiner SZKs sind trivial.

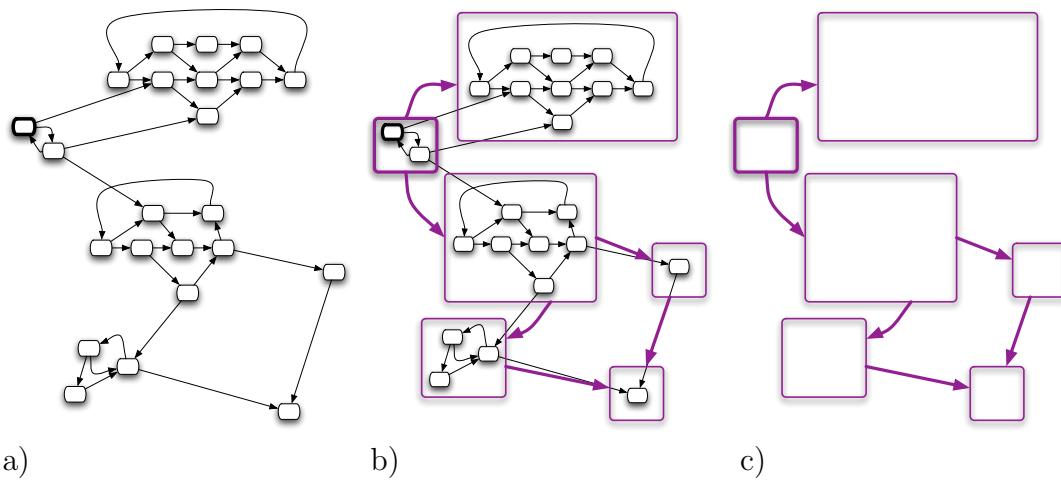


Abbildung 7.3: a) Ein Graph, b) seine SZKs und c) der reduzierte Graph

In Abb. 7.3 c) ist der reduzierte Graph von Abb. 7.3 a) dargestellt. Den beiden terminalen SZK entsprechend, hat der azyklische Graph zwei Knoten, von denen keine Kanten ausgehen.

Lemma 7.11 Der reduzierte Graph $\text{RG}^c(\mathcal{N}, \mathbf{m}_0)$ ist azyklisch.

Beweis: Als Übung. □

Da der reduzierte Graph $\text{RG}^c(\mathcal{N}, \mathbf{m}_0)$ immer azyklisch ist, können seine terminalen SZKs in linearer Zeitkomplexität berechnet werden. Dies wird im Algorithmus 7.3 zur Prüfung von Lebendigkeits-Invarianzeigenschaften ausgenutzt. Dieser hat daher auch eine lineare Zeitkomplexität (in Bezug auf die Größe von $\text{RG}(\mathcal{N}, \mathbf{m}_0)$). Wenn der Algorithmus erfolgreich terminiert, enthalten alle terminalen SZKs eine Markierung, die die Eigenschaft II erfüllt. Daher existiert zu jeder erreichbaren Markierung eine Nachfolgemarkierung, die II erfüllt. Falls der Algorithmus nicht erfolgreich terminiert, enthält mindestens eine terminale SZK eine Markierung, die die Eigenschaft II nicht erfüllt.

Als wichtigen Spezialfall erhalten wir:

Algorithmus 7.3 (Entscheiden einer Lebendigkeits-Invarianzeigenschaft)

Input - Der Erreichbarkeitsgraph $\text{RG}(\mathcal{N}, \mathbf{m}_0)$. Die Lebendigkeits-Invarianzeigenschaft Π .
Output - TRUE falls die Eigenschaft Π erfüllt ist; FALSE falls die Eigenschaft Π nicht erfüllt ist.

1. Berechne die strengen Zusammenhangskomponenten (SZKs) C_1, \dots, C_r von $\text{RG}(\mathcal{N}, \mathbf{m}_0)$.
 2. Berechne den reduzierten Graphen $\text{RG}^c(\mathcal{N}, \mathbf{m}_0) = (V_c, E_c)$ von $\text{RG}(\mathcal{N}, \mathbf{m}_0)$
 3. Berechne die Menge F der terminalen SZKs von $\text{RG}^c(\mathcal{N}, \mathbf{m}_0)$.
 4. **while** es gibt $C_i \in F$ **do**
 - 4.1 **if** C_i enthält keine Markierung \mathbf{m}' , die Π erfüllt.
then return FALSE
 - 4.2 Entferne C_i aus F
 5. Return TRUE
-

Lemma 7.12 Eine Transition $t \in T$ des Netzsysteams $(\mathcal{N}, \mathbf{m}_0)$ ist genau dann lebendig, wenn t als Anschrift in jeder terminalen SZK des reduzierten Graphen $\text{RG}^c(\mathcal{N}, \mathbf{m}_0)$ vorkommt.

Beispiel 7.13 Analyse einer Lebendigkeits-Invarianzeigenschaft

Wir betrachten das Netzsysteem von Abb.6.18 b) mit dem Erreichbarkeitsgraphen von Abb. 7.4. Um den Algorithmus 7.3 auszuführen, müssen zunächst die SZKs berechnet werden. Diese sind schon in Abb. 7.4 durch die Bereiche C_1 , C_2 und C_3 eingezeichnet, wobei C_2 und C_3 terminal sind.

Wir prüfen die Lebendigkeitseigenschaft:

$$\forall \mathbf{m} \in \mathbf{R}(\mathbf{m}_0). \forall t \in T. (\exists \mathbf{m}^t \in \mathbf{R}(\mathbf{m}). \mathbf{m}^t \geq W(\bullet, t))$$

Dazu sei t eine feste Transition und $\Pi(\mathbf{m}) = (\mathbf{m} \geq W(\bullet, t))$.

Dann wird in Schritt 4 des Algorithmus festgestellt, dass jede der beiden terminalen SZK eine Markierung \mathbf{m} enthält die Π erfüllt, d.h. dass $\Pi(\mathbf{m})$ gilt. Da dies für alle Transitionen erfolgreich durchgeführt werden kann, ist das Netzsysteem lebendig.

Führt man Algorithmus 7.3 aus, um zu prüfen ob die Markierung $\mathbf{m}_H = p_2 + p_3 + p_6 + p_7 + p_9 + p_{14}$ ein Rücksetzzustand ist, erhält man das Ergebnis FALSE, da zwar die terminale SZK C_2 die Markierung \mathbf{m}_H enthält, nicht jedoch C_3 .

Anmerkung: Man kann effektivere Algorithmen entwickeln, wenn man besondere Charakteristiken der zu prüfenden Eigenschaft oder des zu analysierenden Systems kennt.

Als Beispiel für den ersten Fall nehmen wir die Eigenschaft der Reversibilität. Dann müssen alle terminalen SZKs die Anfangsmarkierung enthalten, d.h. der Erreichbarkeitsgraph ist insgesamt streng zusammenhängend. Es genügt in diesem Fall also zu prüfen, dass es nur eine SZK gibt.

Zum zweiten Fall nehmen wir an, dass wir schon wissen, dass das System reversibel ist. Dann ist der Erreichbarkeitsgraph streng zusammenhängend. Um die Lebendigkeit einer Transition t zu analysieren, genügt es dann zu prüfen, ob t überhaupt an irgend einer Kante des Erreichbarkeitsgraphen steht.

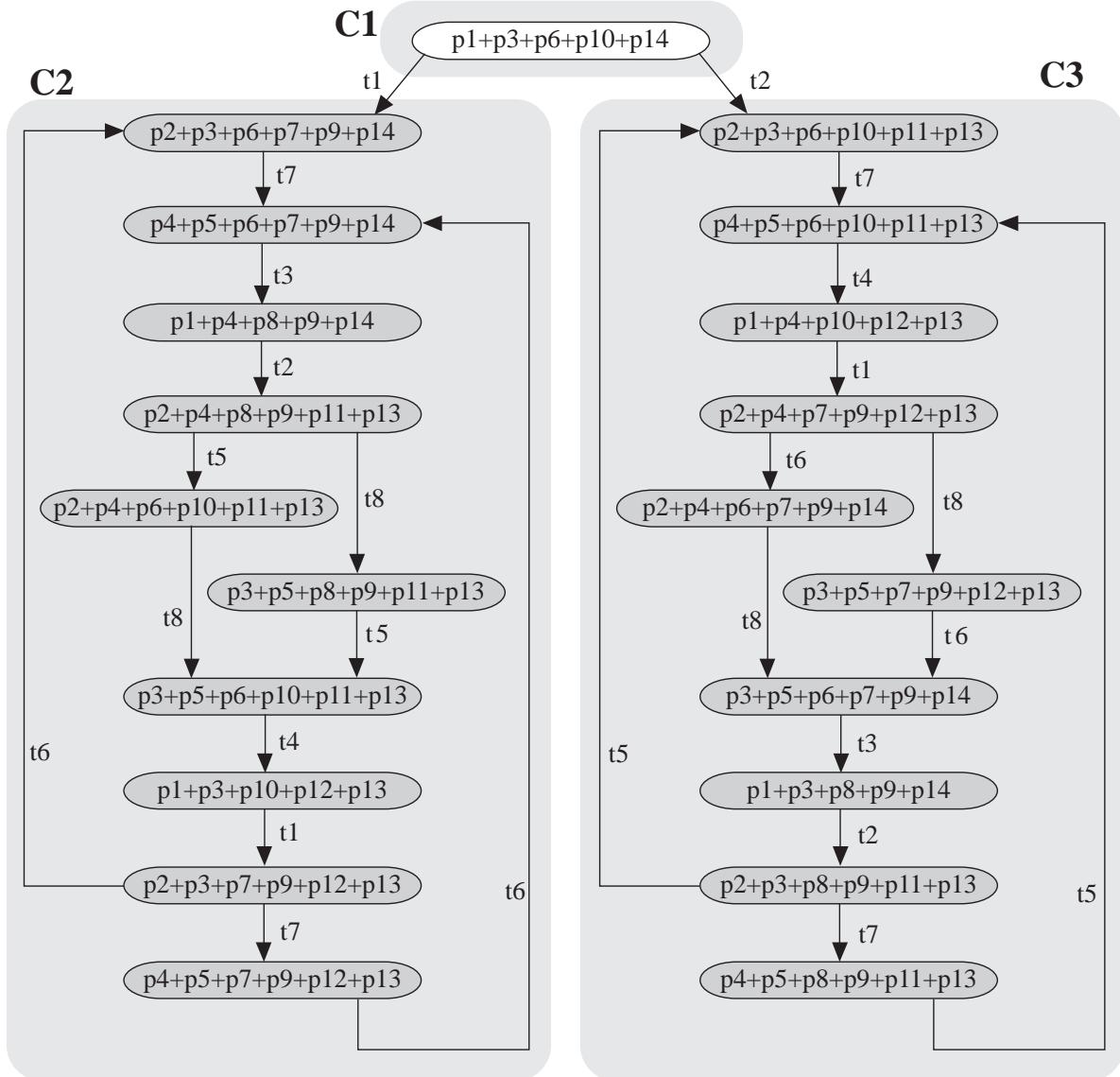


Abbildung 7.4: Erreichbarkeitsgraph des Netzsystems von Abb. 6.18.b

7.2 Der Überdeckungsgraph

Der Algorithmus 7.1 zur Berechnung des Erreichbarkeitsgraphen bricht ohne Ergebnis ab, wenn das Netz unbeschränkt ist. Das ist unbefriedigend, wenn man z.B. zur Fehleranalyse wissen muss, welche Plätze unbeschränkt sind oder wie der Erreichbarkeitsgraph in Teilen aussieht, wo sich die Unbeschränktheit nicht auswirkt.

Mit dem Symbol ω (Omega) wird im Folgenden die Möglichkeit beliebig hoher (aber natürlich endlicher) Markenzahlen auf einem Platz bezeichnet. Dazu werden die natürlichen Zahlen mit diesem formalen Symbol erweitert, sowie *Pseudomarkierungen* eingeführt, deren Komponenten dieses Omega enthalten.

Definition 7.14 Es sei $\mathbb{N}_\omega := \mathbb{N} \cup \{\omega\}$ zusammen mit folgenden Rechenregeln:

$$\forall n \in \mathbb{N} : \omega > n;$$

$$\forall n \in \mathbb{N}_\omega : \omega + n = \omega - n := \omega;$$

$$\forall n \in \mathbb{N} \setminus \{0\} : n \cdot \omega = \omega \cdot n := \omega; 0 \cdot \omega = \omega \cdot 0 := 0$$

Ein Vektor $\mathbf{m} \in \mathbb{N}_\omega^P$ wird Pseudomarkierung genannt, wenn in ihm das Symbol ω vorkommt, und für diese wird die Schaltregel formal übernommen. Eine Pseudomarkierung \mathbf{m} entspricht einer gewöhnlichen (Teil-) Markierung auf den Plätzen s mit $\mathbf{m}(s) \neq \omega$, wobei die mit ω besetzten Komponenten beliebig sind und unberücksichtigt bleiben.

Für Vektoren $\mathbf{m}_1, \mathbf{m}_2 \in \mathbb{Z}^r$ seien die Operatoren $+, -, =$ jeweils komponentenweise erklärt.

Die partielle Ordnung \leq ergibt sich komponentenweise: $\mathbf{m}_1 \leq \mathbf{m}_2$, falls $\forall p \in P : \mathbf{m}_1(p) \leq \mathbf{m}_2(p)$.

Die strikte Ordnung $\mathbf{m}_1 < \mathbf{m}_2$ (besser: $\mathbf{m}_1 \lneq \mathbf{m}_2$) steht für $(\mathbf{m}_1 \leq \mathbf{m}_2 \text{ und } \mathbf{m}_1 \neq \mathbf{m}_2)$.

7.2.1 Konstruktion des Überdeckungsgraphen

Wir definieren im folgenden den Überdeckungsgraphen. Ein Überdeckungsgraphen ist eine reudzierte Version des Erreichbarkeitsgraphen. Es wird sich zeigen, dass Ein Überdeckungsgraph stets endlich ist, während dies bei Erreichbarkeitsgraphen bekanntermaßen nicht immer der Fall ist. Somit ist es unvermeidbar, dass ein Überdeckungsgraph weniger Information über die Zustände enthält als der Erreichbarkeitsgraph. Wie wir aber sehene werden, reicht diese eingeschränkt Version bereits aus, um einige (wenn auch nicht alle) Fragen zu Petrinetzen zu entscheiden – insbesondere die Fragestellung, ob ein Netz beschränkt ist.

Sei $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ ein P/T-Netz. Wir konstruieren durch Algorithmus 7.4 einen gerichteten, kantenbeschrifteten Graphen $G(\mathcal{N}) := (V, E)$ mit $V \subseteq \mathbb{N}_\omega^P$ und Kanten $E \subseteq V \times T \times V$ und nennen ihn *Überdeckungsgraph*. Dabei schreiben wir $\mathbf{m}_1 \xrightarrow{*w} \mathbf{m}_2$, wenn es in $G(\mathcal{N})$ einen Pfad vom Knoten \mathbf{m}_1 zum Knoten \mathbf{m}_2 gibt, der die in der

Algorithmus 7.4 (Berechnung eines Überdeckungsgraphen)

Input - Das P/T-Netz $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$
Output - Der Überdeckungsgraph Graph $G(\mathcal{N}) = (V, E)$.

1. Initialisiere $G(\mathcal{N}) = (\{\mathbf{m}_0\}, \emptyset)$; \mathbf{m}_0 sei un gefärbt;
 2. **while** Es gibt ungefärbte Knoten in V **do**
 - 2.1 Wähle einen ungefärbten Knoten $\mathbf{m} \in V$ und färbe ihn.
 - 2.2 **for** Für jede in \mathbf{m} aktivierte Transition t **do**
 - 2.2.1 Berechne \mathbf{m}' mit $\mathbf{m} \xrightarrow{t} \mathbf{m}'$ und $X(\mathbf{m}') := \{\mathbf{m}''' \in V \mid \mathbf{m}''' \leq \mathbf{m}' \text{ und } \mathbf{m}''' \xrightarrow{*} \mathbf{m}'\}$;
 - 2.2.2 **if** $X(\mathbf{m}') \neq \emptyset$ **then** $\mathbf{m}_1(p) := \begin{cases} \omega, & \exists \mathbf{m}''' \in X(\mathbf{m}') : \mathbf{m}'''(p) < \mathbf{m}'(p) \\ \mathbf{m}'(p), & \text{sonst.} \end{cases}$
 - 2.2.3 **else** $\mathbf{m}_1 := \mathbf{m}'$;
 - 2.2.4 **if** $\mathbf{m}_1 \notin V$ **then** $V := V \cup \{\mathbf{m}_1\}$, wobei \mathbf{m}_1 ein ungefärbter Knoten sei. ;
 - 2.2.4 **E** := $E \cup \{(\mathbf{m}, t, \mathbf{m}_1)\}$;
 3. Der Algorithmus terminiert mit Ergebnis. ($G(\mathcal{N})$ ist der Überdeckungsgraph.)
-

Kantenfolge zusammengefügte Beschriftung $w \in T^*$ hat. Wir schreiben $\mathbf{m}_1 \xrightarrow{*} \mathbf{m}_2$, wenn es irgendeinen Pfad gibt, dessen Beschriftung uns nicht wichtig ist.

Beispiel 7.15 Abb. 7.5 zeigt ein P/T-Netz \mathcal{N} mit Überdeckungsgraph.

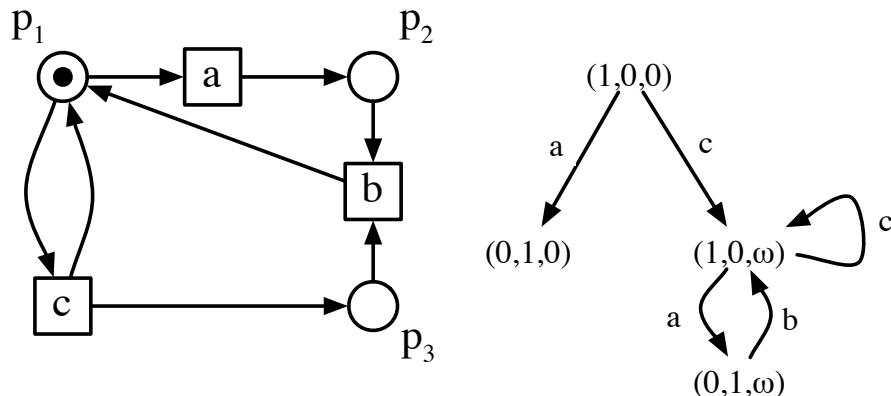


Abbildung 7.5: Ein Netz \mathcal{N} mit Überdeckungsgraph

Die Bedeutung des Überdeckungsgraphen ergibt sich aus folgenden Eigenschaften:

Satz 7.16 Sei $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ ein P/T-Netz und $G(\mathcal{N}) = (V, E)$ ein Überdeckungsgraph zu \mathcal{N} , dann ist ein Platz $p \in P$ genau dann beschränkt, wenn es keinen Knoten $\mathbf{m} \in V$ mit $\mathbf{m}(p) = \omega$ gibt.

Gilt $\mathbf{m}_0 \xrightarrow{w} \mathbf{m}$ im Netz \mathcal{N} für ein Wort $w \in T^*$, so gibt es in $G(\mathcal{N})$ einen Knoten \mathbf{m}_1 mit $\mathbf{m}_1 \geq \mathbf{m}$ und $\mathbf{m}_0 \xrightarrow{*} \mathbf{m}_1$.

Aus dieser letzten Eigenschaft leitet sich der Name „Überdeckungsgraph“ für $G(\mathcal{N})$ ab, denn zu jedem in \mathcal{N} erreichbaren Knoten $\mathbf{m} \in RG(\mathcal{N})$ gibt es einen, i.A. anderen, in

$G(\mathcal{N})$, der diesen „überdeckt“. Um diesen Sachverhalt zu beweisen, zeigen wir zunächst die Termination dieses Algorithmus.

Satz 7.17 Der Algorithmus 7.4 zur Konstruktion von $G(\mathcal{N})$ terminiert.

Beweis: Angenommen, der Algorithmus terminiere *nicht*, dann ist $|V|$ unendlich.

Begründung: Wäre $|V|$ endlich, so auch die Menge $V \times T$ und alle Paare $(m, t) \in (V \times T)$ wären einmal nach Eintritt in die **while**-Schleife ausgewählt worden und die Termination wird erreicht.

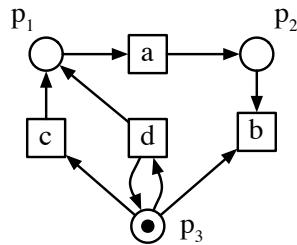
Nun sind nach Konstruktion alle Knoten $\mathbf{m} \in V$ von \mathbf{m}_0 aus erreichbar. Betrachten wir den spannenden Baum $B(\mathcal{N})$ von $G(\mathcal{N})$, der dadurch entsteht, dass diejenigen Kanten weggelassen werden, die im Schritt 2.2.4 zu einem schon existierenden Knoten gezeichnet werden. $B(\mathcal{N})$ entsteht also aus $G(\mathcal{N})$ durch Streichen gewisser Kanten.

Da $B(\mathcal{N})$ verzweigungsendlich ist (jeden Knoten verlassen maximal $|T|$ viele Kanten) aber selbst unendlich viele Knoten besitzt, gibt es in $B(\mathcal{N})$ nach dem Satz von König (1936) einen unendlichen Pfad $\mathbf{m}_0 \rightarrow \mathbf{m}_1 \rightarrow \mathbf{m}_2 \rightarrow \dots \dots \rightarrow \mathbf{m}_i \rightarrow \mathbf{m}_{i+1} \rightarrow \dots$ in dem kein Knoten \mathbf{m}_i zweimal vor kommt.

In jeder unendlichen Folge von paarweise verschiedenen Vektoren $\mathbf{m}_i \in \mathbb{N}^P$ gibt es, nach einem Satz von Dickson (1926), eine unendliche Teilfolge $\mathbf{m}_{i_1} \xrightarrow{+} \mathbf{m}_{i_2} \xrightarrow{+} \mathbf{m}_{i_3} \xrightarrow{+} \dots \xrightarrow{+} \mathbf{m}_{i_j} \xrightarrow{+} \mathbf{m}_{i_{j+1}} \xrightarrow{+} \dots$ mit der Eigenschaft $\mathbf{m}_{i_j} < \mathbf{m}_{i_{j+1}}$.

Nach Konstruktion muss $\mathbf{m}_{i_{j+1}}$ eine ω -Komponente mehr als \mathbf{m}_{i_j} haben, denn $\mathbf{m}_{i_j} \xrightarrow[B(\mathcal{N})]{*} \mathbf{m}_{i_{j+1}}$ impliziert $\mathbf{m}_{i_j} \xrightarrow[G(\mathcal{N})]{*} \mathbf{m}_{i_{j+1}}$ und wenn $\mathbf{m}_{i_j} < \mathbf{m}_{i_{j+1}}$ ist, wird $\mathbf{m}_{i_{j+1}}$ mindestens eine ω -Komponente mehr enthalten als \mathbf{m}_{i_j} . Dies führt zu dem gewünschten Widerspruch, denn nur endlich viele ω -Komponenten sind überhaupt möglich. Also terminiert die Konstruktion von $G(\mathcal{N})$ nach dem Algorithmus 7.4. \square

Aufgabe 7.18 Konstruieren Sie einen Überdeckungsgraphen $G(\mathcal{N})$ für das folgende P/T-Netz \mathcal{N} .



Satz 7.19 Für ein P/T-Netz \mathcal{N} ist $RG(\mathcal{N})$ genau dann endlich, wenn kein Knoten von $G(\mathcal{N})$ eine ω -Komponente besitzt.

Beweis: ($\text{kein } \omega \Rightarrow RG(\mathcal{N})$ ist endlich)

Kommt in keiner Komponente von Knoten aus V in $G(\mathcal{N}) = (V, E)$ die Bezeichnung ω vor, so ist $G(\mathcal{N})$ identisch mit dem Erreichbarkeitsgraph von \mathcal{N} , und, wegen der Termination des Verfahrens, ist auch die Menge $V = \mathbf{R}(\mathcal{N})$ endlich.

$(\omega \Rightarrow \mathbf{RG}(\mathcal{N}) \text{ ist unendlich})$

Sei $\mathbf{m}_2 \in V$ mit $\mathbf{m}_2(s) = \omega$ für ein $s \in P$, so gibt es einen Pfad in $G(N)$

$$\mathbf{m}_0 \xrightarrow[\alpha]{*} \mathbf{m}_{3,s} \xrightarrow[\beta]{*} \mathbf{m}_s \xrightarrow[t]{*} \mathbf{m}_{1,s} \xrightarrow[\gamma]{*} \mathbf{m}_{2,s}$$

und $\mathbf{m}_{3,s} < \mathbf{m}_{1,s}$, genauer: $\mathbf{m}_{3,s}(s) < \mathbf{m}_{1,s}(s)$.

Da das Wort w auf dem Pfad von \mathbf{m}_0 über $\mathbf{m}_{3,s}$ nach \mathbf{m}_s eine Schaltfolge im Netz \mathcal{N} ist, gibt es also Wörter $\alpha, \beta, \gamma \in T^*$ und erreichbare Markierungen $\tilde{\mathbf{m}}_3, \tilde{\mathbf{m}}_1, \tilde{\mathbf{m}}_4, \dots$ mit $\mathbf{m}_0 \xrightarrow{\alpha} \tilde{\mathbf{m}}_3 \xrightarrow{\beta t} \tilde{\mathbf{m}}_1 \xrightarrow{\gamma} \tilde{\mathbf{m}}_4 \dots$

Da $\tilde{\mathbf{m}}_3(s) < \tilde{\mathbf{m}}_1(s) < \tilde{\mathbf{m}}_4(s) < \dots$ gilt, ist die Platz $s \in P$ also in \mathcal{N} nicht beschränkt und $\mathbf{R}(\mathcal{N})$ ist keine endliche Menge. \square

Wir erhalten aus den vorangegangenen Resultaten insbesondere die folgende Aussage.

Korollar 7.20 *Das Beschränktheitsproblem ist entscheidbar.*

7.2.2 Aussagekraft des Überdeckungsgraphen

Jede Schaltfolge w des Netzes besitzt einen gleichbeschriebenen Pfad w im Überdeckungsgraph. Die Umkehrung gilt im allgemeinen jedoch nicht, wie das folgende Netz in Abbildung 7.6 zeigt. Das Netz hat den folgenden Erreichbarkeitsgraph:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \xrightarrow[a]{\leftarrow b} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \xrightarrow[a]{\leftarrow b} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \cdots \xrightarrow[a]{\leftarrow b} \begin{pmatrix} 1 \\ n \end{pmatrix} \xrightarrow[a]{\leftarrow b} \cdots$$

und den folgenden Überdeckungsgraph:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \xrightarrow[a]{\leftarrow b} \begin{pmatrix} 1 \\ \omega \end{pmatrix} \xrightarrow[a]{\leftarrow b} \cdots$$

Man beachte, dass es keine mit b beschriftete Kante von $\begin{pmatrix} 1 \\ \omega \end{pmatrix}$ nach $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ gibt. Nun ist beispielsweise $w = abb$ ein Kantenzug im Überdeckungsgraph, dieser ist aber nicht in \mathbf{m}_0 aktiviert.



Abbildung 7.6: Beispielnetz

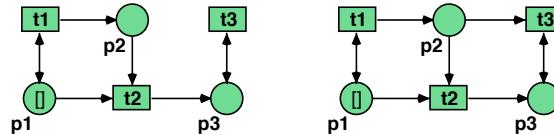
Der Überdeckungsgraph liefert somit die Möglichkeit, Nichterreichbarkeit einer Markierung nachzuweisen, denn Nichtüberdeckbarkeit impliziert Nichterreichbarkeit. Allerdings ist Erreichbarkeit nicht auf diese Art und Weise nachweisbar und auch nicht Lebendigkeit.

Theorem 7.21 Es ist mit Hilfe des Überdeckungsgraphen nicht entscheidbar,

1. ob eine Transition lebendig ist,
2. ob ein Netz lebendig ist, oder
3. ob eine Markierung \mathbf{m}' erreichbar ist.

Beweis: Dies zeigen wir durch die Angabe zweier Netzpaare mit gleichem Überdeckungsgraphen, bei dem ein Netz die Eigenschaft besitzt, das andere aber nicht.

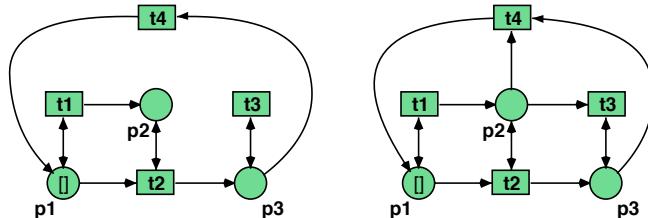
1. Im linken Netz ist t_3 lebendig, im rechten aber nicht.



$$(1, 0, 0) \xrightarrow{t_1} (1, \omega, 0) \xrightarrow{t_2} (0, \omega, 1)$$

t_1 t_3

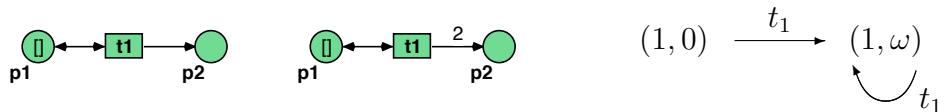
2. Das linke Netz ist lebendig, das rechte ist nach der Schaltfolge $t_1 t_2 t_3$ tot.



$$(1, 0, 0) \xrightarrow{t_1} (1, \omega, 0) \xrightleftharpoons[t_4]{t_2} (0, \omega, 1)$$

t_1 t_3

3. Im linken Netz ist die Markierung $(1, 1)^{tr}$ erreichbar, im rechten dagegen nicht.



Also sind die Eigenschaften nicht mittels Überdeckungsgraphen entscheidbar. □

7.2.3 Komplexität des Beschränktheitsproblems

Wie wir gesehen haben, ist das Beschränktheitsproblem entscheidbar. Wie groß ist der Berechnungsauflauf? Folgende Ergebnisse sind bekannt:

Satz 7.22 (Rackoff, 1978) Das Beschränktheitsproblem ist mit $O(2^{c \cdot n \cdot \log(n)})$ Platzbedarf entscheidbar.

Hierbei bezeichnet n die Größe des P/T-Netzes, d.h. z.B. die Länge seiner textuellen Beschreibung (Kodierung).

Eine untere Schranke wurde schon zuvor von Lipton gefunden:

Satz 7.23 (Lipton, 1976) *Das Beschränktheitsproblem benötigt für seine Entscheidung mindestens $O(2^{c\sqrt{n}})$ Platzbedarf.*

Ein besseres Ergebnis ist von Rosier und Yen bewiesen worden:

Satz 7.24 (Rosier und Yen, 1986) *Das Beschränktheitsproblem kann für ein gegebenes P/T-Netz $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ mit*

$$O\left(2^{c \cdot |P| \cdot \log(|P|)} \cdot (\log(|T|) + \max_{x,y \in P \cup T} (|W(x,y)|))\right)$$

Platzbedarf entschieden werden.

Für festes $|P|$ ist das Problem PSPACE-vollständig.

Erst für spezielle Teilklassen der Petrinetze kann man eine niedrigere Komplexität für dieses Entscheidungsproblem erwarten.

Definition 7.25 *Ein P/T-Netz $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ heißt konfliktfrei, falls es keinen Konfliktplatz enthält, d.h. einen Platz p mit $|p^\bullet| > 1$.*

Satz 7.26 (Rosier et. al., 1987) *Das Beschränktheitsproblem kann für konfliktfreie Petrinetze mit $O(n^{1,5})$ Platzbedarf entschieden werden.*

7.3 Zustandsraumexplosion bei nebenläufigen Systemen

Es gibt Petri-Netze mit im Verhältnis zu ihrer Größe sehr großer Erreichbarkeitsmenge, wie das folgende Ergebnis zeigt. Diese Erscheinung ist unter dem Begriff *Zustandsexplosion* bekannt.

Satz 7.27 *Es gibt eine unendliche Folge von beschränkten Petri-Netzen $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3, \dots$, deren Größe – definiert durch $(|P| + |T| + |F|)$ – linear wächst, während die Größe ihrer Erreichbarkeitsmengen $|\mathbf{R}(\mathcal{N}_1)|, |\mathbf{R}(\mathcal{N}_2)|, |\mathbf{R}(\mathcal{N}_3)|, \dots$ schneller als jede primitiv rekursive Funktion wächst.*

Eine nicht primitiv rekursiv berechenbare Funktion ist die folgende Variante der *Ackermann-Funktion*:

$$A(0, n) := 2n + 1, A(m + 1, 0) := 1, A(m + 1, n + 1) := A(m, A(m + 1, n))$$

Die Netze \mathcal{N}_i berechnen gerade immer Markenzahlen auf einem festgelegten Platz, die bis an den Wert von $A(i, 2)$ heranreichen.

Aus diesem Ergebnis folgt, dass das eben angegebene Entscheidungsverfahren (Konstruktion 7.4) für die Endlichkeit der Erreichbarkeitsmenge eines P/T-Netzes mit dem Überdeckungsgraphen nicht primitiv-rekursiv ist!

```
function hexp(n) is
    val := 1
    for i := 1 to n do
        cnt := val
        while cnt > 0 do
            val := 2 * val
            cnt := cnt - 1
        endwhile
    endfor
    return val
```

Abbildung 7.7: Hyper-exponentielles Wachstum

Wir illustrieren im folgenden die Zustandsraumexplosion an einem Beispiel. Dazu betrachten wir zunächst den Algorithmus in Abbildung 7.7. Wir indizieren die Werte der Variablen mit dem Schleifenindex i . Der Initialwert für $i = 0$ ist $val_0 = 1$. Für $i = 1$ ergibt sich $cnt_1 = 1$ und nach Durchlauf der While-Schleife $val_1 = 2$. Für $i = 2$ wird $cnt_2 = val_1 = 2$ und nach Durchlauf der Schleife ist:

$$val_2 = \underbrace{2 \cdots 2}_{cnt_2\text{-mal}} val_1 = 2^{val_1} val_1$$

Allgemein gilt die Rekursionsgleichung:

$$val_{i+1} = val_i \cdot 2^{val_i} \quad \text{und} \quad val_0 = 1$$

Wir wollen das Wachstum von val_i studieren und schätzen daher $val_i \cdot 2^{val_i}$ durch $v_i := 2^{v_i}, v_0 = 1$ von unten ab. Es ist $v_0 = 1, v_1 = 2, v_2 = 2^2, v_3 = 2^{2^2}, v_4 = 2^{2^{2^2}}$ usw. Abwickeln der Rekursion ergibt also:

$$v_n = 2^{v_{n-1}} = 2^{2^{v_{n-2}}} = 2^{2^{2^{v_{n-3}}}} = 2^{2^{2^{\dots}}} \}^{n\text{-mal}}$$

Also bestimmt das Argument n die Höhe des Exponentenstapels. Ein solches Wachstum nennt man hyper-exponentiell.

Wir können natürlich auch einen anderen Faktor als 2 wählen. Dazu müssen wir die Programmzeile $val := 2 * val$ nur gegen $val := k * val$ für ein beliebiges $k \geq 2$ austauschen. Allgemein gilt dann die Rekursionsgleichung:

$$f(i+1) = f(i) \cdot k^{f(i)} \quad \text{und} \quad f(0) = 1$$

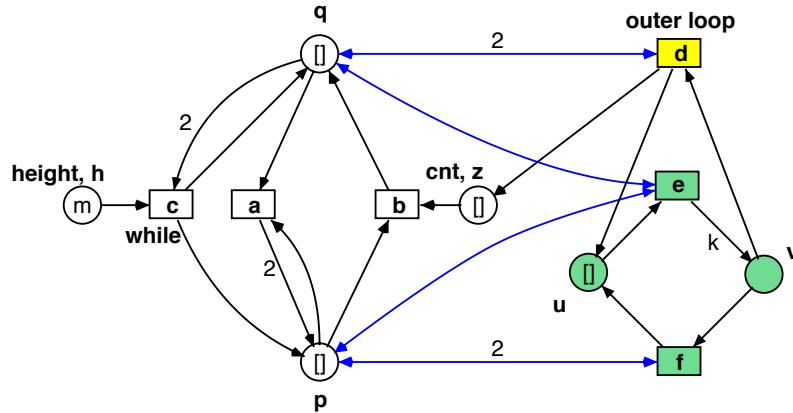


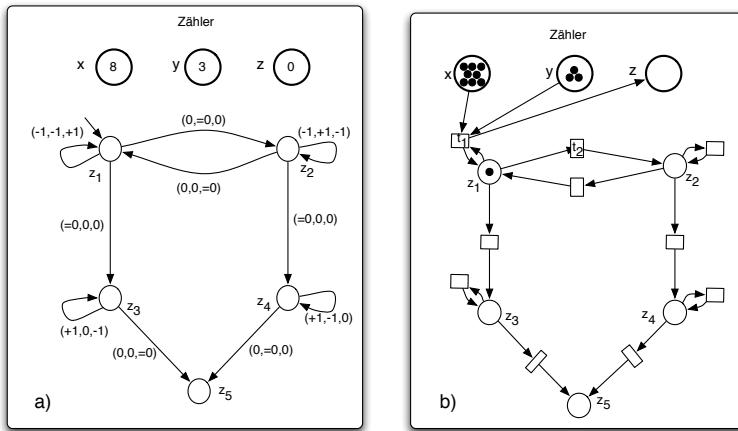
Abbildung 7.8: P/T Netz \mathcal{N}_7 mit hyper-exponentieller Beschränkung

Wir wollen jetzt ein Netz betrachten, das im wesentlichen den obigen Algorithmus zum hyper-exponentiellen Wachstum nachbildet. Das Netz \mathcal{N}_7 aus Abb. 7.8 besitzt zwei Stellen u und v , die das mit Hilfe der Transitionen e und f die Anweisung $val := k * val$ simulieren. Die Anweisung wird dabei in zwei Teile aufgespaltet:

$$tmp := k * val; val := tmp$$

Die Zählerstelle z simuliert die Anzahl der Schleifendurchläufe. Das Netz hat Operationsmodi: p^2 , pq und q^2 . Im Modus pq wird die erste Zuweisung dieser Anweisung der While-Schleife simuliert (Transition e), im Modus p^2 wird die Zweite Zuweisung simuliert. Im Modus q^2 wird durch die Transition d die Anweisung $cnt := val$ simuliert.

Dieses Netz zeigt, dass eine alleinige Beschränkung der Kapazität der Plätzen in der Regel nicht ausreicht, um niedrige Komplexitätseigenschaften zu erreichen, denn das Netz \mathcal{N}_7 hat stets eine endliche Erreichbarkeitsmenge, diese ist jedoch so groß, dass alle Petrinetz-Analysetools auf ihre Grenzen hin getestet werden können.


 Abbildung 7.9: 3-Zählerautomat und P/T-Netz für $x := x \bmod y$

7.4 Inhibitor-Netze, Zählerprogramme und Turing-Mächtigkeit

Wir diskutieren die Frage: sind Petrinetze genauso mächtig wie Turing-Maschinen? Im affirmativen Fall hätte dies den **Vorteil**, dass Petrinetze alle berechenbaren Prozeduren ausführen könnten und der Modellierer in dieser Hinsicht nicht eingeschränkt wird. Andererseits hätte dies den **Nachteil**, dass viele Eigenschaften wie bei Turing-Maschinen nicht entscheidbar sind, wie z.B. Beschränktheit, die Erreichbarkeit eines Zustandes (einer Markierung) oder Lebendigkeit.

Beginnen wir, diese Frage bei P/T-Netzen zu diskutieren. Diese sind leichter mit *Zählerautomaten* zu vergleichen als direkt mit Turing-Maschinen. Bekanntlich sind schon Zählerautomaten mit nur zwei Zählern so mächtig wie Turing-Maschinen¹. In der Abbildung 7.9 a) ist als Beispiel ein Zählerautomat mit drei Zählern x , y und z gegeben. Die Zähler haben Werte aus den natürlichen Zahlen \mathbb{N} . Gesteuert wird er durch ein Transitionssystem mit enlich vielen (hier fünf) Zuständen, wobei z_1 der Anfangszustand ist. Die Transitionen haben die Form (a, b, c) , wobei sich a , b und c jeweils auf die Zähler x , y und z beziehen. Die Einträge a , b und c haben Werte aus $\{+1, -1, 0\}$. Diese Werte bedeuten folgendes:

- Falls der Wert $+1$ ist, wird der jeweilige Zähler um eins erhöht.
- Falls der Wert -1 ist, ist die Transition nur möglich, wenn der jeweilige Zählerwert positiv ist. In diesem Fall wird der Wert um eins verringert.
- Falls der Wert 0 ist, wird der jeweilige Zählerwert nicht verändert.
- Falls der Wert $= 0$ ist, ist die Transition nur möglich, wenn der jeweilige Zählerwert Null ist. In diesem Fall wird der Wert ebenfalls nicht verändert.

¹Zählerautomaten mit nur *einem* Zähler sind mächtiger als endliche Automaten, aber weniger mächtig als Kellerautomaten, die alle kontextfreien Sprachen akzeptieren können.

Zum Beispiel bewirkt die Transition $z_1 \xrightarrow{(-1, -1, +1)} z_1$, dass die Zähler x und y um eins verringert werden, falls ihr Wert positiv ist und im selben Schritt der Zähler z um eins erhöht wird. Dies ist bei der gegebenen Anfangskonfiguration drei mal möglich, wonach nur die Transition $z_1 \xrightarrow{(0, = 0, 0)} z_2$ erfolgen kann und danach entsprechendes im Zustand z_2 passiert. Offensichtlich wird insgesamt der Anfangswert von x so oft wie möglich um den Anfangswert von y verringert und der verbleibende Rest nach x gebracht, wonach der Automat in den Endzustand z_5 übergeht. Dadurch wird die Zuweisung $x := x \bmod y$ berechnet. Da bei der Reduktion von x durch y der Wert von y verloren geht, wird er in z wieder hergestellt, was in der folgenden Phase in umgekehrter Weise geschieht.

Kann jeder Zählerautomat durch ein P/T-Netz simuliert werden? Die Abbildung 7.9 b) zeigt den Anfang einer solchen Konstruktion. Die Zähler werden durch entsprechende Plätze x , y und z dargestellt, deren Markenzahl dem Zählerinhalt entspricht. Die Plätze z_i stellen das Transitionssystem zur Steuerung dar. Sie können insgesamt nur eine Marke enthalten². Der oben beschriebenen Transition $z_1 \xrightarrow{(-1, -1, +1)} z_1$ entspricht die Netztransition t_1 . Die Wirkung der anderen Transitionen auf die Zähler müsste entsprechend ergänzt werden.

Wäre diese Konstruktion erfolgreich durchführbar, so bestünde ein gravierender logischer Widerspruch. Das Beschränktheitsproblem bei Zählerautomaten ist nämlich (wie bei Turing-Maschinen) unentscheidbar, während in diesem Kapitel gezeigt wurde, dass es für P/T-Netze entscheidbar ist. P/T-Netze können also nicht Turing-Maschinen simulieren. Wo scheitert der obige Konstruktionsversuch? Um der Transition t_2 in Abbildung 7.9 b) das Verhalten der entsprechenden Transition $z_1 \xrightarrow{(0, = 0, 0)} z_2$ in a) zu geben, müsste geprüft werden, ob der P'latz y keine Marke enthält. Ist ein solcher Test bei P/T-Netzen möglich?

Die Abbildung 7.10 a) zeigt die Situation *in nuce*. Der Ausschnitt des steuernden Transitionssystems ist durch die Plätze z_1 , z_2 und z_3 gegeben. Dabei soll die Marke in z_1 genau dann nach z_2 gebracht werden, wenn der Platz („der Zähler“) y mindestens eine Marke enthält. Beim Schalten der Transition **not null** funktioniert dies, jedoch nicht bei der Transition **null**. Eine Lösung des Problems erfolgt üblicherweise mit Hilfe der Hinzufügung eines *komplementären Platzes* y' wie in Abbildung 7.10 b). Dies ist ein Platz der die P-Invarianten-Gleichung $y + y' = k$ erfüllt, wobei k wie immer aus der Anfangsmarkierung zu berechnen ist (hier $k = 2$). Folglich kann die Transition **null** genau dann schalten, wenn $\mathbf{m}(y') \geq 2$, d.h. - wie gewünscht - $\mathbf{m}(y) = 0$ ist. Diese Konstruktion ist wegen der P-Invarianten-Gleichung jedoch nur möglich, wenn der Platz y beschränkt ist, eine Forderung, die bei der Simulation von Turing-mächtigen Zählerautomaten nicht zu erfüllen ist³. Generell ist ein solcher *Nulltest* bei P/T-Netzen nicht möglich, d.h. sie sind weniger mächtig als Turing-Maschinen.

Um Nulltests zu ermöglichen, wurden für P/T-Netze *Nulltest-Kanten* oder *Inhibitor-Kanten* wie in Abbildung 7.11 a) eingeführt. Hier kann *per definitionem* die Transition

²Sie erfüllen die P-Invariante $z_1 + z_2 + z_3 + z_4 + z_5 = 1$.

³Wenn ein Zähler beschränkt ist, kann man ihn eliminieren, indem seine Funktion in das endliche steuernde Transitionssystem integriert wird. Wenn alle Zähler beschränkt sind, ist der Zählerautomat nur so mächtig wie ein endlicher Automat.

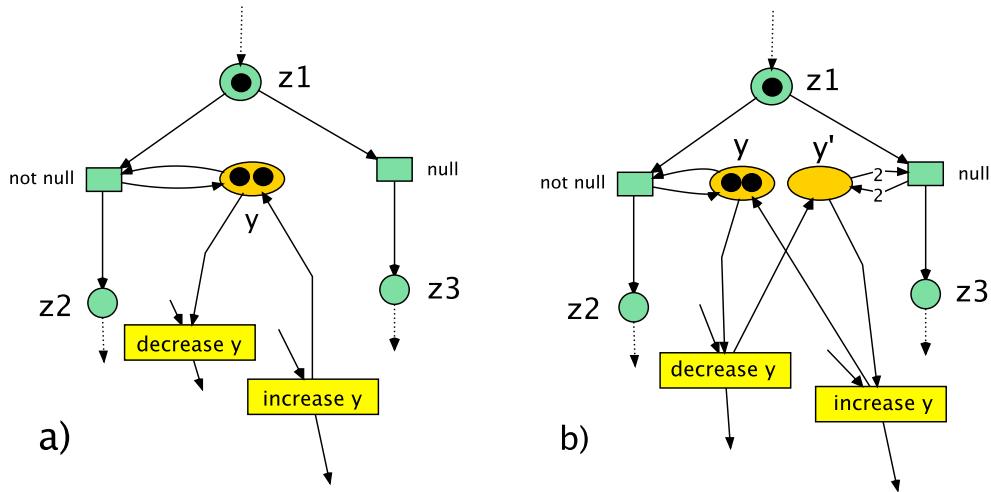


Abbildung 7.10: Nulltest-Situation für P/T-Netze in a) und Lösung in b)

	Turing-mächtig?
P/T-Netze	nein
Inhibitor-Netze	ja
gefärzte Netze mit endlichen Farbmengen	nein
gefärzte Netze mit beliebigen Farbmengen	ja

Tabelle 7.1: Turing-Mächtigkeit von Petrinetzen

t nur schalten, wenn der mit ihr über die Inhibitor-Kante verbundene Platz y leer ist. P/T-Netze, bei denen Inhibitor-Kanten erlaubt sind heißen *Inhibitor-Netze*. Eine Lösung des Nulltest-Problems für Inhibitor-Netze ist in Abbildung 7.11 b) dargestellt.

Inhibitor-Netze haben neben diesem Vorteil natürlich den Nachteil, dass viele Probleme unentscheidbar werden oder die Analysekomplexität erheblich steigt. Dies gilt auch für gefärbte Netze, für die im Vorgriff auf ein späteres Kapitel eine Lösung des Nulltest-Problems in Abbildung 7.11 c) gegeben ist. Eine solche Lösung setzt die Existenz einer nicht endlichen Farbmengen voraus, wie hier die Farbe `integer` für den Platz y . Gefärbte Netze mit nur endlichen Farbmengen sind gleichmächtig zu P/T-Netzen, da sie immer zu solchen aufgefaltet werden können. Eine Übersicht über die Turing-Mächtigkeit bei Petrinetzen gibt die Tabelle 7.1.

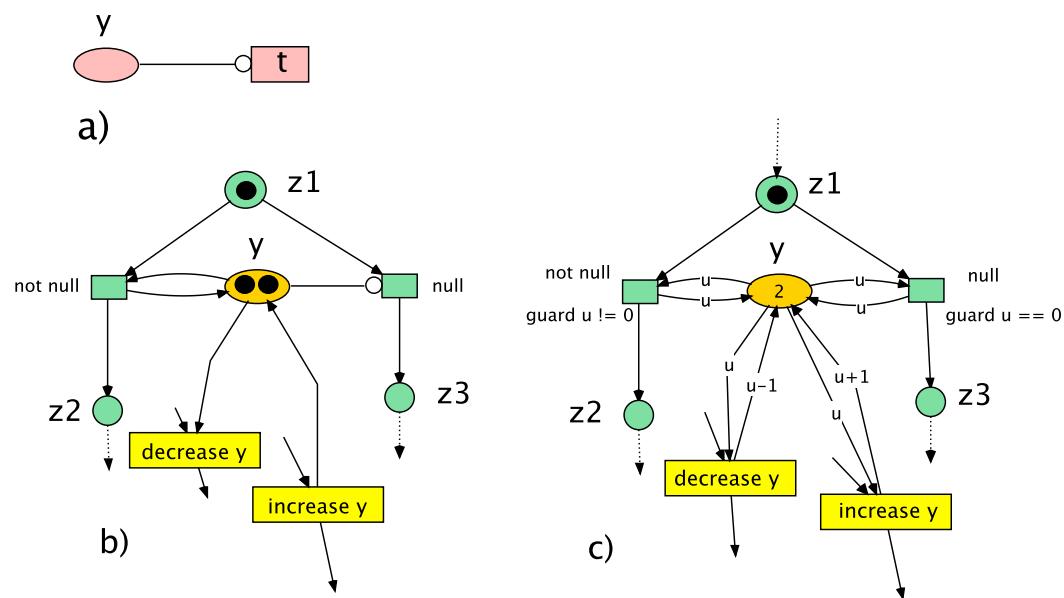


Abbildung 7.11: Nulltest in Inhibitor- und gefärbten Netzen

7.5 Strukturelle Eigenschaften

Netzvarianten erlauben die Verifikation von Markierungsvarianteneigenschaften ohne die oft exponentielle Konstruktion des Erreichbarkeitsgraphen.

7.5.1 Stellenvarianten

Beziehungen zwischen Programmvariablen, die bei der Ausführung eines Programmes erhalten bleiben, heißen Programminvarianten. 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 Varianten von entsprechend größerer Bedeutung, vorausgesetzt, es gelingt solche Gesetzmäßigkeiten zu erkennen. Eine Besonderheit von P -Invarianten (bzw. S -Invarianten) liegt darin, dass sie berechenbar sind (im Gegensatz zu allgemeinen Programminvarianten). Die folgenden Abschnitte stammen aus [JV87].

Als Beispiel betrachten wir das Leser/Schreiber-Problem. Die folgende Abb. 7.12 a) zeigt ein entsprechendes Modell als 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:

- 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

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

$$\mathbf{m}(s) > 0 \Rightarrow \mathbf{m}(l) = 0 \quad (7.1)$$

$$\mathbf{m}(s) \leq 1 \quad (7.2)$$

$$\mathbf{m}(l) > 0 \Rightarrow \mathbf{m}(s) = 0 \quad (7.3)$$

Wir werden die Gültigkeit dieser Bedingungen im nächsten Abschnitt beweisen.

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 7.12 b) 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).

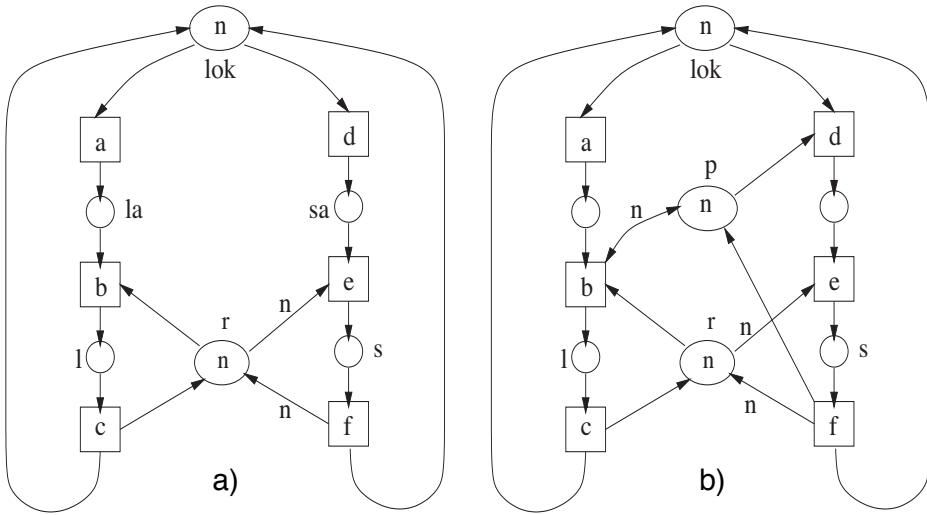


Abbildung 7.12: Leser/Schreiber-Problem in a) und mit Priorität für Schreibaufträge in b)

Für das P/T-Netz des Leser/Schreiber-Problems in Abb. 7.12 a) gelten die Gleichungen ($n \geq 1$):

$$i_1 : \quad lok + la + sa + l + s = n \quad (7.4)$$

$$i_2 : \quad l + r + n \cdot s = n \quad (7.5)$$

lok, la, \dots stehen hier abkürzend für $\mathbf{m}(lok), \mathbf{m}(la), \dots$. Gleichungen dieser Form heißen *P-Invarianten-Gleichungen*⁴.

Definition 7.28 Es sei $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ ein P/T-Netz mit $P = \{p_1, \dots, p_n\}$.

Eine Gleichung der Form $\sum_{i=1}^n k_i \cdot \mathbf{m}(p_i) = k$ mit $k_i, k \in \mathbb{Z}$ heißt *P-Invarianten-Gleichung*, wenn sie für alle erreichbaren Markierungen $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ gilt.

Abkürzend wird eine P-Invarianten-Gleichung auch als $\sum_{i=1}^n k_i \cdot p_i = k$ geschrieben.

Eine P-Invarianten-Gleichung ist also eine spezielle Markierungs-Invarianzeigenschaft (Definition 7.4 auf Seite 120). Der Beweis, dass obige P-Invarianten-Gleichungen für alle erreichbaren Markierungen gelten, stellen wir zurück, leiten aber aus ihnen die Synchronisationsspezifikationen ab.

- Spezifikation⁵ 7.1: $s > 0 \rightarrow l = 0$ folgt aus i_2 .
- Spezifikation 7.2: $s \leq 1$ folgt aus i_2 .
- Spezifikation 7.3: $l > 0 \rightarrow s = 0$ folgt aus i_2 .

⁴Zur Unterscheidung zu dem bald definierten P-Invarianten-Vektor.

⁵Wieder steht $s > 0$ für $\mathbf{m}(s) > 0$ usw.

Damit ist nachgewiesen, dass das Netz von Abb. 7.12 a) die Synchronisationsspezifikationen des Leser/Schreiber-Problems erfüllen. Wir zeigen nun, wie man die Gültigkeit der P-Invarianten-Gleichungen nachweist. Dazu betrachten wir das P/T-Netz von Abb. 7.12 a) und die P-Invarianten-Gleichung i_2 :

$$i_2 : \mathbf{m}(l) + \mathbf{m}(r) + n \cdot \mathbf{m}(s) = n \quad (7.6)$$

Sie gilt für die Anfangsmarkierung \mathbf{m}_0

Als Induktionsbeweis zeigt man dann: Gilt 7.6 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 7.6 auch für \mathbf{m}_2 .

- Für $t = e$ gilt:

$$\begin{aligned} \mathbf{m}_2(sa) &= \mathbf{m}_1(sa) - 1 \\ \mathbf{m}_2(r) &= \mathbf{m}_1(r) - n \\ \mathbf{m}_2(s) &= \mathbf{m}_1(s) + 1, \end{aligned} \quad (7.7)$$

während alle anderen Stellen unverändert bleiben.

Also: Gilt 7.6 für \mathbf{m}_1 , dann auch für \mathbf{m}_2 .

- Dies ist analog für alle Transitionen durchzuführen.

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

Definition 7.29 Es sei $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ ein P/T-Netz.

Der Vektor $\Delta_{\mathcal{N}}(t) \in \mathbb{Z}^{|P|}$ heißt Wirkung der Transition $t \in T$ und ist definiert durch

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

Die durch Aneinanderreihung der Vektoren $\Delta_{\mathcal{N}}(t_1) \dots \Delta_{\mathcal{N}}(t_{|T|})$ gebildete $(|P| \times |T|)$ -Matrix $\Delta_{\mathcal{N}}$ heißt Wirkungsmatrix 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 7.30 Wenn t die Markierung \mathbf{m}_1 durch Schalten in \mathbf{m}_2 überführt ($\mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2$), dann gilt:

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

Beweis: Der Beweis folgt direkt durch Vergleich der Definitionen 6.13 und 7.29. □

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 P/T-Netze beruht darauf, dass Netz-Invarianten berechnet werden können. Die Grundlage dazu liefert der folgende *Satz von Lautenbach*.

Definition 7.31 Es sei M^{tr} die Transponierte einer Matrix M und $\underline{0} \in \mathbb{Z}^{|T|}$ der Nullvektor.

Jede ganzzahlige Lösung $i \in \mathbb{Z}^{|P|} \setminus \{\underline{0}\}$ des linearen Gleichungssystems $\Delta_{\mathcal{N}}^{tr} \cdot i = \underline{0}$ heißt P -Invarianten-Vektor⁶ des P/T -Netzes \mathcal{N} .

Satz 7.32 (Lautenbach) Wenn $i \in \mathbb{Z}^{|P|} \setminus \{\underline{0}\}$ ein P -Invarianten-Vektor ist, dann gilt $i^{tr} \cdot \mathbf{m} = i^{tr} \cdot \mathbf{m}_0$ für alle erreichbaren Markierungen $\mathbf{m} \in \mathbf{R}(\mathcal{N})$.

Beweis: Induktion über $\mathbf{R}(\mathcal{N})$.

Anfang: Die Behauptung ist trivial für $\mathbf{m} = \mathbf{m}_0$.

Annahme: Es gelte die Behauptung für $\mathbf{m}_1 \in \mathbf{R}(\mathcal{N})$, also $i^{tr} \cdot \mathbf{m}_1 = i^{tr} \cdot \mathbf{m}_0$, und es gelte $\mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2$ für eine Transition $t \in T$.

Schritt: Aus der Voraussetzung $\Delta_{\mathcal{N}}^{tr} \cdot i = \underline{0}$ folgt dann $\underline{0}^{tr} = (\Delta_{\mathcal{N}}^{tr} \cdot i)^{tr} = i^{tr} \cdot (\Delta_{\mathcal{N}}^{tr})^{tr} = i^{tr} \cdot \Delta_{\mathcal{N}}$ und damit $i^{tr} \cdot \Delta_{\mathcal{N}}(t) = 0$.

Also gilt mit Satz 7.30 die Induktionsbehauptung:

$$\begin{aligned} i^{tr} \cdot \mathbf{m}_2 &= i^{tr} \cdot (\mathbf{m}_1 + \Delta_{\mathcal{N}}(t)) \\ &= i^{tr} \cdot \mathbf{m}_1 + i^{tr} \cdot \Delta_{\mathcal{N}}(t) \\ &= i^{tr} \cdot \mathbf{m}_1 \\ &= i^{tr} \cdot \mathbf{m}_0 \end{aligned}$$

□

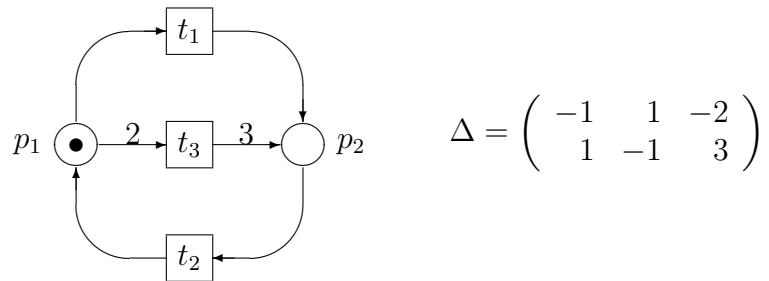


Abbildung 7.13: Beispielnetz

Aus einer P -Invarianten-Gleichung $\sum_{i=1}^n k_i \cdot \mathbf{m}(p_i) = k$ kann man aber nicht schließen, dass der Vektor (k_1, \dots, k_n) eine P -Invariante ist. Dies kann daran liegen, dass in den Schaltfolgen Transitionen, die die Eigenschaft zerstören könnten, nie aktiviert sind. Das Netz in Abbildung 7.13 erfüllt für alle aus der Anfangsmarkierung $\mathbf{m}_0 = (1, 0, 0)^{tr}$ erreichbaren Markierungen die Invariantengleichung:

$$1 \cdot \mathbf{m}(p_1) + 1 \cdot \mathbf{m}(p_2) = 1 \cdot \mathbf{m}_0(p_1) + 1 \cdot \mathbf{m}_0(p_2) = 1$$

⁶Im Gegensatz zur P -Invarianten-Gleichung in Def. 7.28.

Der zugehörige Vektor $i = (1, 1)^{tr}$ ist jedoch kein Invariantenvektor, denn

$$\Delta^{tr} i = \begin{pmatrix} -1 & 1 \\ 1 & -1 \\ -2 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Man beachte, dass für dieses Beispiel die Anfangsmarkierung gerade so gewählt ist, dass in keiner erreichbaren Markierung t_3 aktiviert ist. Für $\mathbf{m} = (2, 0, 0)$ ist t_3 aktiviert, und die Invariantengleichung ist ungültig. Eine Invariantengleichung $\sum_{i=1}^n k_i \cdot \mathbf{m}(p_i) = k$, die aus einer P -Invarianten abgeleitet wurde, gilt dagegen für jede Wahl der Initialmarkierung – für eine jeweils angepasste Konstante k .

Beispiel 7.33 Wir interpretieren Satz 7.32 anhand unseres Beispiels. Abbildung 7.14 zeigt die Inzidenzmatrix Δ_N des Leser/Schreiber Netzes von Abb. 7.12 a) und zwei Invarianten-Vektoren i_1 und i_2 .

Δ_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 7.14: Inzidenzmatrix Δ_N mit P-Invariantenvektor i_1, i_2 und T-Invariantenvektor j

Man rechne nach: $\Delta_N^{tr} \cdot i_1 = \underline{0}$ und $\Delta_N^{tr} \cdot i_2 = \underline{0}$.

Folglich gilt nach Satz 7.32 für jede von der Anfangsmarkierung $\mathbf{m}_0^{tr} = (n, 0, 0, 0, 0, n)$ aus erreichbare Markierung \mathbf{m} :

$$\begin{aligned} i_2^{tr} \cdot \mathbf{m} &= 1 \cdot \mathbf{m}(l) + n \cdot \mathbf{m}(s) + 1 \cdot \mathbf{m}(r) = \\ i_2^{tr} \cdot \mathbf{m}_0 &= 1 \cdot \mathbf{m}_0(l) + n \cdot \mathbf{m}_0(s) + 1 \cdot \mathbf{m}_0(r) \\ &= 1 \cdot 0 + n \cdot 0 + 1 \cdot n = n \end{aligned}$$

Dies ist genau die P-Invarianten-Gleichung 7.6.

Wir fassen zusammen: Aus einem gegebenen Netz \mathcal{N} können durch Berechnung aller ganzzahligen Lösungen i in $\Delta_N^{tr} \cdot i = \underline{0}$ alle P -Invarianten-Vektoren gefunden werden.

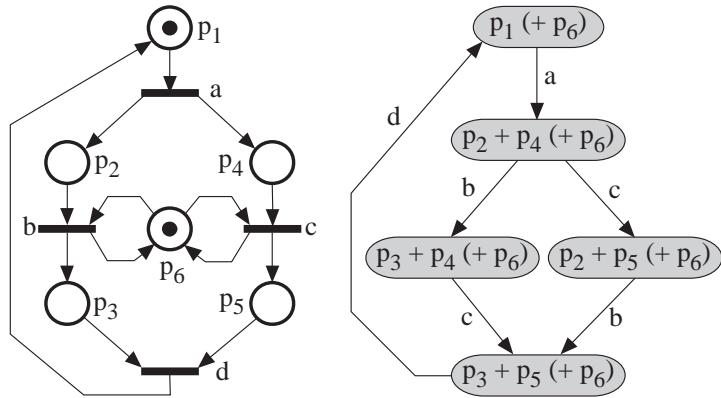


Abbildung 7.15: Beschränktes, lebendiges und reversibles Netz mit Erreichbarkeitsgraph

Für die Anfangsmarkierung \mathbf{m}_0 werden durch $i^{tr} \cdot \mathbf{m} = i^{tr} \cdot \mathbf{m}_0$ die entsprechenden P-Invarianten-Gleichung aufgestellt, die für alle $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ gelten. Mit ihnen können, wie oben gezeigt, Netzeigenschaften nachgewiesen werden.

Die Umkehrung gilt aber i.a.nicht, d.h. aus einer

Die P-Invarianten-Vektoren können u.a. herangezogen werden, um *strukturelle* Eigenschaften zu beweisen.

Beispiel 7.34 Für jede Anfangsmarkierung gelten für das Netz in Abb. 7.15 die folgenden Markenerhaltungsgesetze (P-Invarintengleichungen):

$$\begin{aligned}\mathbf{m}(p_1) + \mathbf{m}(p_2) + \mathbf{m}(p_3) &= \mathbf{m}_0(p_1) + \mathbf{m}_0(p_2) + \mathbf{m}_0(p_3) = k_1(\mathbf{m}_0) \\ \mathbf{m}(p_1) + \mathbf{m}(p_4) + \mathbf{m}(p_5) &= \mathbf{m}_0(p_1) + \mathbf{m}_0(p_4) + \mathbf{m}_0(p_5) = k_2(\mathbf{m}_0) \\ \mathbf{m}(p_6) &= \mathbf{m}_0(p_6) = k_3(\mathbf{m}_0)\end{aligned}$$

Hierbei ist \mathbf{m}_0 die Anfangsmarkierung und \mathbf{m} eine beliebige erreichbare Markierung. Die $k_i(\mathbf{m}_0) \in \mathbb{Z}$ sind jeweils eine von \mathbf{m}_0 abhängige Konstante. Daher gelten die Ungleichungen:

$$\begin{aligned}\mathbf{m}(p_1) &\leq \min(k_1(\mathbf{m}_0), k_2(\mathbf{m}_0)) \\ \mathbf{m}(p_i) &\leq k_1(\mathbf{m}_0); i = 2, 3 \\ \mathbf{m}(p_j) &\leq k_2(\mathbf{m}_0); j = 4, 5 \\ \mathbf{m}(p_6) &= k_3(\mathbf{m}_0)\end{aligned}$$

Diese Ungleichungen beweisen, dass das Netz für jede Anfangsmarkierung beschränkt ist. Dies ist natürlich eine stärkere Eigenschaft als Beschränktheit. Da sie nur von der Struktur des Netzes und nicht von der jeweils gewählten Anfangsmarkierung abhängt, heißt sie *strukturelle Beschränktheit* (structural boundedness).

Ein Netz heißt *strukturell beschränkt* (engl. structurally bounded), wenn es für jede Initialmarkierung beschränkt ist. Eine echt positive P-Invariante \mathbf{i} kann strukturelle Beschränktheit zeigen. Eine P-Invariante $\mathbf{i} \geq \mathbf{0}$ heißt überdeckend, wenn ihr Träger alle Stellen enthält.

Theorem 7.35 Besitzt das P/T Netz \mathcal{N} eine überdeckende, positive P-Invariante \mathbf{i} , d.h. $\mathbf{i}(p) > 0$ für alle p , dann ist \mathcal{N} strukturell beschränkt.

Beweis: Sei \mathbf{m}_0 eine beliebige Initialmarkierung. Mit dem Satz von Lautenbach gilt für jede erreichbare Markierung \mathbf{m} :

$$\mathbf{i} \cdot \mathbf{m} = \sum_{p \in P} \mathbf{i}(p) \cdot \mathbf{m}(p) = \sum_{p \in P} \mathbf{i}(p) \cdot \mathbf{m}_0(p)$$

Da $\mathbf{i}(p) > 0$ für jede Stelle gilt, erhalten wir:

$$\mathbf{i}(p) \cdot \mathbf{m}(p) \leq \sum_{p \in P} \mathbf{i}(p) \cdot \mathbf{m}(p)$$

Also gilt für p die Beschränkung (wieder mit $\mathbf{i}(p) > 0$):

$$\mathbf{m}(p) \leq \frac{(\mathbf{i} \cdot \mathbf{m}_0)}{\mathbf{i}(p)} \leq \mathbf{i} \cdot \mathbf{m}_0$$

Da für jedes Netz die Anfangsmarkierung fest ist, existiert also für jede Anfangsmarkierung und jeden Platz eine obere Schranke für die Markenzahl. \square

Satz 7.36 Sei \mathcal{N} konservatives P/T-Netz, dann gilt: Aus $f(\mathbf{m}) := \sum_{p \in P} \mathbf{m}(p) \cdot f(p)$ folgt für jede erreichbare Markierung $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ stets $f(\mathbf{m}_0) = f(\mathbf{m})$.

Beweis: Konservative Netze besitzen mit f eine echt positive Stellen-Invariante. \square

Satz 7.36 impliziert die Endlichkeit der Erreichbarkeitsmenge $\mathbf{R}(\mathcal{N})$.

7.5.2 Transitionsinvarianten

Für die Wirkung einer Schaltfolge ist die Reihenfolge der Transitionen unerheblich, nur die Anzahl ist entscheidend. Die *Parikh-Abbildung* liefert zu jeder Sequenz ein Tupel mit den Häufigkeiten.

Definition 7.37 Sei $\Sigma = \{x_1, x_2, \dots, x_k\}$ ein endliches, geordnetes Alphabet (mit $x_1 < x_2 < \dots < x_k$). Die Häufigkeit, mit der das Zeichen $x \in \Sigma$ in dem Wort $w \in \Sigma^*$ vorkommt, wird mit $|w|_x$ notiert.

Die Parikh-Abbildung ist der Homomorphismus $\psi : \Sigma^* \rightarrow \mathbb{N}^k$, definiert durch:

$$\psi(w) := (|w|_{x_1}, |w|_{x_2}, \dots, |w|_{x_k})$$

Der Parikh-Vektor $\psi(w)$ gibt in der i -ten Komponente an, wie oft das Zeichen x_i in dem Wort $w \in \Sigma^*$ vorkommt.⁷

Auch die Lösungen j von $\Delta_{\mathcal{N}} \cdot j = \underline{0}$ haben eine wichtige Interpretation für die Analyse von Systemen. (Hier $\Delta_{\mathcal{N}}$ wird *nicht* transponiert.)

⁷In manchen Büchern wird der Parikh-Vektor durch: $\#(w) := (\#_1(w), \#_2(w), \dots, \#_{|T|}(w))$ dargestellt, wobei dann natürlich $\#_i(w) := |w|_{t_i}$ gilt.

Definition 7.38 Jede Lösung $j \in \mathbb{N}^{|T|} \setminus \{\underline{0}\}$ von $\Delta_{\mathcal{N}} \cdot j = \underline{0}$ heißt T -Invarianten-Vektor des P/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 $|w|_{t_i}$ mal vor, dann ist der Parikh-Vektor $\psi(w) := (|w|_{x_1}, |w|_{x_2}, \dots, |w|_{x_{|T|}})$ von dem Wort w ein T -Invarianten-Vektor. T -Invarianten beschreiben also die Häufigkeit des Schaltens jeder Transition bei reproduzierendem Verhalten.

Satz 7.39 Es seien $\mathbf{m}_1, \mathbf{m}_2$ Markierungen und $w = t_{i_1} \dots t_{i_k}$ 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 $t \in T$ genau $j(t)$ mal in w vorkommt, d.h.: $j = \psi(w)$.

Beweis: Es gilt nach Voraussetzung

$$\mathbf{m}_1 = \mathbf{m}_2 = \mathbf{m}_1 + \Delta_{\mathcal{N}}(t_{i_1}) + \Delta_{\mathcal{N}}(t_{i_2}) + \dots + \Delta_{\mathcal{N}}(t_{i_k})$$

genau dann, wenn gilt:

$$\begin{aligned} \underline{0} &= \Delta_{\mathcal{N}}(t_{i_1}) + \dots + \Delta_{\mathcal{N}}(t_{i_k}) \\ &= \Delta_{\mathcal{N}} \cdot e_{t_{i_1}} + \dots + \Delta_{\mathcal{N}} \cdot e_{t_{i_k}} \\ &= \Delta_{\mathcal{N}} \cdot (|w|_{t_1} e_{t_1}) + \dots + \Delta_{\mathcal{N}} \cdot (|w|_{t_{|T|}} e_{t_{|T|}}) \\ &= \Delta_{\mathcal{N}} \cdot \psi(w) \end{aligned}$$

Mit anderen Worten: $\psi(w)$ ist eine T -Invariante. □

Für $\mathbf{m} \xrightarrow{w} \mathbf{m}$ gilt dann also $\Delta_{\mathcal{N}} \cdot \psi(w) = \underline{0}$.

Der T -Invarianten-Vektor j in Abb. 7.14 besagt also, dass die Anfangsmarkierungen \mathbf{m}_0 dann wieder erreicht (reproduziert) wird, wenn drei Lese- und zwei Schreibaufträge ihren Zyklus durchlaufen haben.

Aufgabe 7.40 (Invarianten) Gegeben seien die P/T-Netze \mathcal{N} und \mathcal{N}' , wobei \mathcal{N} das Gesamtnetz aus der Abb. 7.16 ist und \mathcal{N}' das Teilnetz ist, dessen Elemente mit durchgezogenen Linien gezeichnet sind.

Berechnen Sie mit Hilfe der Sätze 7.32 (Satz von Lautenbach) und 7.39 P -Invarianten und T -Invarianten für das Gesamtnetz \mathcal{N} und das Teilnetz \mathcal{N}' (falls diese existieren). Untersuchen Sie, ob es in den Netzen Schaltfolgen gibt, die die Anfangsmarkierung wiederherstellen.

Theorem 7.41 Sei \mathcal{N} ein beschränktes P/T-Netz, das in \mathbf{m} die unendliche Schaltfolge $w \in T^\omega$ aktiviert. Dann existiert eine T -Invariante j mit $j(t) = 0$, falls t nicht oder nur endlich oft in w vorkommt $j(t) > 0$, falls t unendlich oft in w vorkommt.

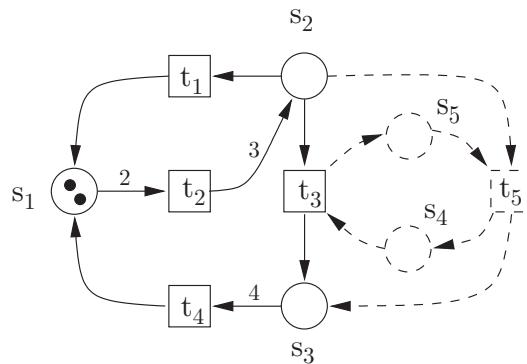


Abbildung 7.16: Netz und Teilnetz

Beweis: Für die Schaltfolge $w \in T^\omega$ gelte:

$$\mathbf{m} = \mathbf{m}_1 \xrightarrow{t_1} \mathbf{m}_2 \xrightarrow{t_2} \mathbf{m}_3 \xrightarrow{t_3} \dots$$

Sei T_ω die Menge der in w unendlich oft vorkommenden Transitionen. Sei $w = t_1 t_2 t_3 \dots$ und sei k_1 der Index, ab dem $w' = t_{k_1} t_{k_1+1} t_{k_1+2} \dots$ nur noch Transitionen aus T_ω enthält. Wir können nun w' weiter zerlegen. Wir definieren die Indexfolge k_1, k_2, k_3, \dots mit $k_1 < k_2 < k_3 < \dots$ so, dass die Schaltfolge

$$w_{k_i} = t_{k_i} t_{k_i+1} \dots t_{k_{i+1}-1}$$

alle Transitionen aus T_ω enthält.

Nun können die Markierungen \mathbf{m}_{k_i} in der Folge

$$\mathbf{m}_1 \xrightarrow{w_{k_1}} \mathbf{m}_{k_2} \xrightarrow{w_{k_2}} \mathbf{m}_{k_3} \dots$$

nicht alle verschieden sein, denn da \mathcal{N} ein beschränktes P/T-Netz ist, existieren nur endlich viele erreichbare Markierungen, so dass sich mindestens eine Markierung wiederholen muss:

$$\mathbf{m} = \mathbf{m}_{k_i} = \mathbf{m}_{k_j} \quad \text{für } i < j$$

Also ist

$$\mathbf{m}_{k_i} \xrightarrow{w_{k_i} \dots w_{k_{j-1}}} \mathbf{m}_{k_j}$$

Damit ist das Parikh-Bild $\psi(w_{ij})$ der Schaltfolge $w_{ij} := w_{k_i} \dots w_{k_{j-1}}$ nach Satz 7.39 eine T -Invariante. \square

Folgendes Theorem erlaubt es, für beschränkte Netze durch Invariantenanalyse Lebendigkeit zu verwerfen: Besitzt ein beschränktes Netz keine T -Invariante, dann kann es nicht lebendig sein.

Theorem 7.42 Sei \mathcal{N} ein beschränktes und lebendiges P/T-Netz. Dann existiert eine echt positive T -Invariante j , d.h. $j(t) > 0$ für alle $t \in T$.

Beweis: Da das Netz lebendig ist, existiert eine unendliche Schaltfolge w , in der alle Transitionen unendlich oft vorkommen. Mit Theorem 7.41 existiert dann eine echt positive T -Invariante.

Man kann dies auch direkt beweisen: Wenn \mathcal{N} lebendig ist, dann gibt es zur Initialmarkierung \mathbf{m}_0 eine Schaltfolge, in der alle Transitionen unendlich oft vorkommen. Da das Netz beschränkt ist, kommt in dieser Folge $\mathbf{m}_0 \xrightarrow{t_{i_1}} \mathbf{m}_1 \xrightarrow{t_{i_2}} \mathbf{m}_2 \dots$ mindestens eine Markierung doppelt vor: $\mathbf{m}_i \xrightarrow{w} \mathbf{m}_j$ mit $\mathbf{m}_i = \mathbf{m}_j$. Hierbei können wir w so wählen, dass alle Transitionen vorkommen. Also ist nach Satz 7.39 $\psi(w)$ eine T -Invariante. \square

Die Umkehrung des Satzes ist praktisch. Sei \mathcal{N} ein beschränktes P/T-Netz, für das keine echt positive T -Invariante j existiert, dann ist \mathcal{N} auch nicht lebendig.

7.5.3 Fallen und Siphons

Wir betrachten im folgenden nur einfache P/T-Netze, d.h. $W(x, y) \leq 1$.

Fallen Wir betrachten Stellenmengen $A \subseteq P$ mit der Eigenschaft, dass alle Transitionen t , die aus A Marken entfernen, auch wieder Marken in A generieren.

Definition 7.43 Sei $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ ein einfaches P/T-Netz.

Eine Stellenmenge $A \subseteq P$ heißt Falle (engl. trap), wenn jede Transition t , die aus A Marken entfernt, auch wieder Marken in A generiert:

$$\forall t \in T : t \in A^\bullet \Rightarrow t \in {}^*A \quad \text{bzw. kurz} \quad A^\bullet \subseteq {}^*A$$

Eine Falle $A \subseteq P$ heißt in m markiert, wenn mindestens eine Stelle in A markiert ist:

$$\sum_{p \in A} \mathbf{m}(p) > 0$$

Da jede Transition t , die Marken aus der Falle entfernt, auch wieder welche in ihr erzeugt, ist es nicht möglich, eine markierte Falle komplett zu leeren. Es ist also mindestens eine Marke in der Falle gefangen.

Lemma 7.44 Sei A eine Falle eines einfachen P/T-Netzes \mathcal{N} . Ist die Falle A in einer Markierung \mathbf{m}_0 markiert, dann bleibt sie dies auch in allen von \mathbf{m}_0 aus erreichbaren Markierungen.

$$\forall \mathbf{m}_0 : \sum_{p \in A} \mathbf{m}_0(p) > 0 \Rightarrow \forall \mathbf{m} \in R(N, \mathbf{m}_0) : \sum_{p \in A} \mathbf{m}(p) > 0$$

Beweis: Als Übung. \square

Siphons/Co-Fallen Die Umkehrung des Konzeptes *Falle* existiert auch. Es wird als *Siphon* (engl. Abfluss) oder auch als *co-Falle* bezeichnet.

Definition 7.45 Sei $\mathcal{N} = (P, T, F, W, \mathbf{m}_0)$ ein einfaches P/T-Netz.

Eine Stellenmenge $A \subseteq P$ heißt Siphon, wenn jede Transition t , die Marken in A generiert, dazu welche aus A benötigt:

$$\forall t \in T : t \in {}^*A \Rightarrow t \in A^* \quad \text{bzw. kurz} \quad {}^*A \subseteq A^*$$

Ein Siphon $A \subseteq P$ heißt markiert, wenn mindestens eine Stelle in A markiert ist.

$$\sum_{p \in A} \mathbf{m}(p) > 0$$

Bemerkung: Nach Definition ist die leere Menge sowohl ein Siphon als auch eine Falle. Ebenso ist die Stellenmenge P selbst sowohl ein Siphon als auch eine Falle.

Da ein Siphon nur durch Transition t gefüllt werden, die zum Schalten Marken aus dem Siphon benötigen, muss ein unmarkierter Siphon dies auch in allen erreichbaren Markierungen bleiben.

Lemma 7.46 Sei A ein Siphon eines einfachen P/T-Netzes \mathcal{N} . Ist der Siphon A in einer Markierung \mathbf{m}_0 unmarkiert, dann bleibt er dies auch in allen von \mathbf{m}_0 aus erreichbaren Markierungen.

$$\forall \mathbf{m}_0 : \sum_{p \in A} \mathbf{m}_0(p) = 0 \Rightarrow \forall \mathbf{m} \in R(\mathcal{N}, \mathbf{m}_0) : \sum_{p \in A} \mathbf{m}(p) = 0$$

Beweis: Als Übung. □

Aus strukturellen Eigenschaften lassen sich dynamische folgern. Mit dem folgenden Lemma können wir insbesondere die nachweisen, dass ein Netz nicht lebendig sein kann.

Lemma 7.47 Sei \mathcal{N} ein einfaches P/T-Netz ohne isolierte Plätze. Wenn \mathcal{N} ein lebendiges Netz ist, dann muss jeder nichtleere Siphon A in der Initialmarkierung markiert sein.

Beweis: Wäre der Siphon A in der Initialmarkierung unmarkiert, dann wären alle Transitionen aus A^* tot und es gilt $A^* \neq \emptyset$, da \mathcal{N} keine isolierten Plätze besitzt. Also existiert mindestens eine tote Transition. Widerspruch. □

Siphon/Trap-Eigenschaft

Wir können beide Eigenschaften – Falle und Co-Falle – sinnvoll kombinieren: Eine markierte Falle A wird nie leer. Ist A Teilmenge eines Siphons B , dann kann auch B nie leer werden.

Definition 7.48 Ein einfaches P/T-Netz \mathcal{N} hat die Siphon/Trap-Eigenschaft, wenn folgendes gilt: Jeder Siphon B von \mathcal{N} enthält eine anfangsmarkierte Falle A als Teilmenge.

Angenommen \mathcal{N} besitzt die Siphon/Trap-Eigenschaft, dann aktiviert jede erreichbare Markierung mindestens eine Transition.

Satz 7.49 Wenn \mathcal{N} die Siphon/Trap-Eigenschaft besitzt, dann aktiviert jede erreichbare Markierung von \mathcal{N} mindestens eine Transition: \mathcal{N} ist verklemmungsfrei.

Beweis: Wir zeigen zunächst: Wenn \mathbf{m} eine Verklemmung ist, dann ist die Menge $B := \{p \mid \mathbf{m}(p) = 0\}$ ein Siphon.

Sei $t \in {}^{\bullet}B$, dann muss es mindestens einen unmarkierten Platz $p \in {}^{\bullet}t$ geben, der daher auch in B liegt, denn andernfalls wäre t ja aktiviert (wegen $W(x, y) \leq 1$) und \mathbf{m} wäre keine Verklemmung. Also gilt $t \in p^{\bullet} \subseteq B^{\bullet}$, also gilt ${}^{\bullet}B \subseteq B^{\bullet}$, d.h. B ist ein Siphon.

Angenommen \mathcal{N} wäre nicht verklemmungsfrei. Da der Siphon $B := \{p \mid \mathbf{m}(p) = 0\}$ dann unmarkiert ist, enthält er erst recht keine anfangsmarkierte Falle als Teilmenge. Also besitzt \mathcal{N} nicht die Siphon/Trap-Eigenschaft. Widerspruch. \square

8 Anwendungsmodellierung mit Petrinetzen

8.1 Workflow-Netze

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. 8.1). 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, UIMSSs) 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, WFMSSs) ein. Sie erlauben es, die Verwaltung von Geschäftsprozessen aus spezifischen Anwendungsprogrammen herauszuhalten [Aal00], [Aal98], [AH02].

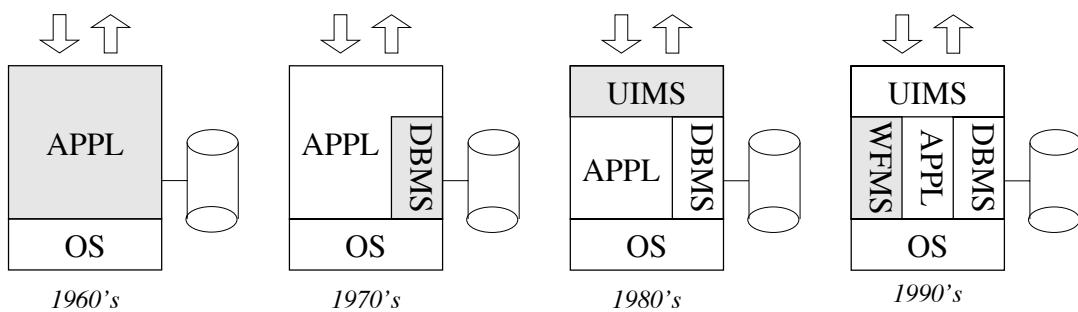


Abbildung 8.1: Workflow-Management-Systeme in historischer Perspektive

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.

Der Begriff Workflow-System wird häufig nicht so spezifisch gebraucht. Beispielsweise 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 8.2 setzt allgemeine kooperative Prozesse und Workflow-Systeme in eine Skalierung zwischen informationszentriert und prozesszentriert bzw. weniger und mehr strukturiert.

8.1.1 Modellierung durch Workflow-Netze

Für Workflow-Systeme gibt es hunderte von Modellierungsansätzen 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], [Aal98] und [AH02] wird hier vorgestellt.

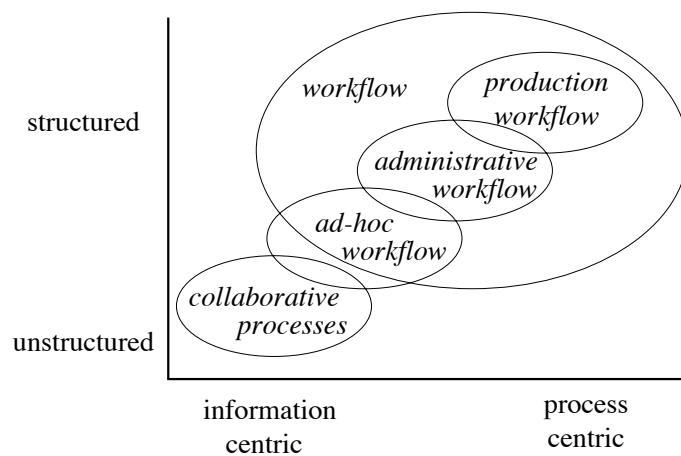


Abbildung 8.2: Workflow Prozesse und kooperative Prozesse im Vergleich

Im Folgenden betrachten wir ein Workflowsystem zur Beschwerdebearbeitung (siehe Abbildung 8.3). Es enthält folgende Aktionen:

1. Aufnahme einer Beschwerde (register)
2. Fragebogen an Beschwerdeführer (send_questionnaire)
3. Bewertung (evaluate) (nebenläufig zu 2.)
4. Fragebogenauswertung (process_questionnaire), 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

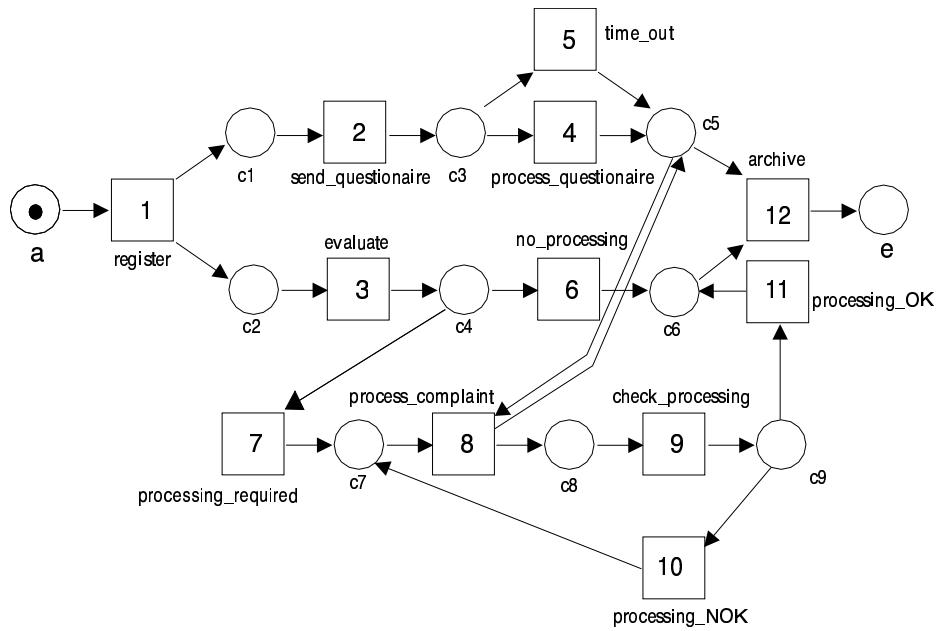


Abbildung 8.3: Ein Petrinetz für einen Vorgang zur Beschwerdebearbeitung

7. Beginn der eigentlichen Prüfung (processing_required)
8. Bearbeitung der Beschwerde (process_complaint) unter Berücksichtigung des Fragebogens¹
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. hat c_2 die Bedeutung: „Bewertung kann beginnen“. Im folgenden werden nur einfache Netze betrachtet, d.h. die Kantenbewertung ist 1 auf F , also: $W(x, y) = 1 \iff (x, y) \in F$.

Es ist sinnvoll nur einen Anfangsplatz a und einen Endplatz e vorzusehen, da Anfang und Ende meist hervorgehobene Ereignisse sind. Dann sollten alle Transitionen auf Päden zwischen diesen liegen. Wir erhalten so folgende Normalform:

Definition 8.1 Ein P/T -Netz $\mathcal{N} = (P, T, F, \mathbf{m}_a)$ heißt Workflow-Netz (WF-Netz), falls

- a) es zwei besondere Plätze $\{a, e\} \subseteq P$ enthält mit $\bullet a = \emptyset$ („Start, Quelle, Anfangsplatz“) und $e^\bullet = \emptyset$ („Ende, Senke, Endplatz“),
- b) alle Plätze und Transitionen auf Päden zwischen a und e liegen und
- c) die Anfangsmarkierung \mathbf{m}_a durch $[\mathbf{m}_a(p) := \text{if } p = a \text{ then } 1 \text{ else } 0 \text{ fi}]$ sowie eine Endmarkierung \mathbf{m}_e durch: $[\mathbf{m}_e(p) := \text{if } p = e \text{ then } 1 \text{ else } 0 \text{ fi}]$ festgelegt sind.

¹Nebenbedingung c_5

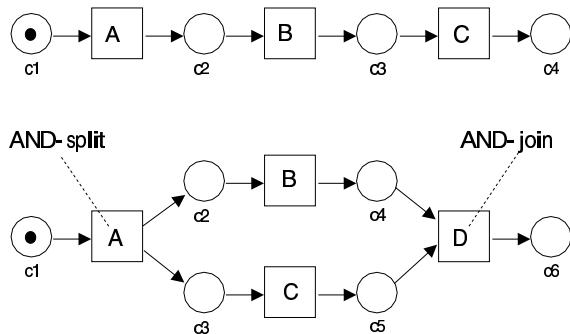


Abbildung 8.4: Sequentielle und nebenläufige Bearbeitung

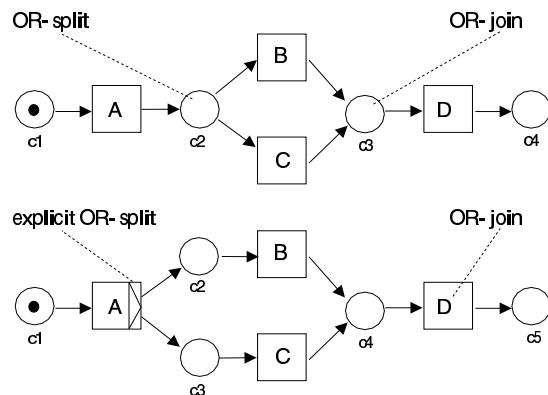


Abbildung 8.5: Alternative Bearbeitung mit und ohne expliziten Testbedingungen

Lemma 8.2 In jedem Workflow-Netz sind Quelle a und Senke e eindeutig festgelegt.

Beweis: Als Übung. □

8.1.2 Modellierungsmuster für Workflows

Typische Strukturen („patterns²“) sind sequentielle, nebenläufige und alternative Bearbeitung (letztere mit und ohne expliziten Testbedingungen). Ein Workflow kann auch von der Interaktion mit der Umgebung abhängen (restriktives System, Trigger, Ressourcen), wie zum Beispiel bei den Ereignissen: „Bearbeiter ist in Urlaub oder krank“ oder „Antwort auf eine Anfrage trifft nicht“.

Es gibt folgende Arten von Triggern (Abbildung 8.8):

²Unter <http://www.workflowpatterns.com/patterns> ist eine Sammlung solcher Strukturmuster für Workflows zu finden.

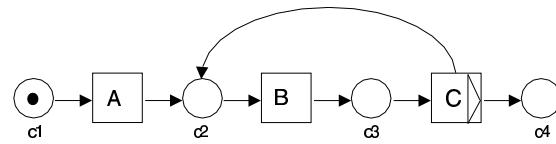


Abbildung 8.6: Iteration

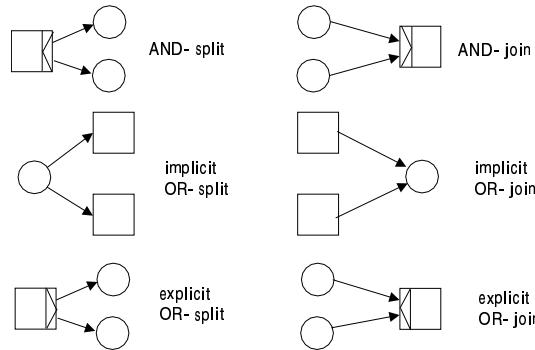


Abbildung 8.7: Graphische Symbole für Verzweigung

- a) Automatisch (keine externe Eingabe notwendig).
- b) Benutzer (user): ein Bearbeiter oder Benutzer oder eine 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.

Es kann auch Wechselwirkungen zwischen Triggern und Verzweigungen geben (ersetze beispielsweise die implizite Verzweigung in c_3 der Abbildung 8.9 durch explizite). Ein anderes Problem besteht darin, dass manchmal Zustandsinformation („milestones“) erforderlich ist (Beispiel: c_5 in Abb. 8.9).

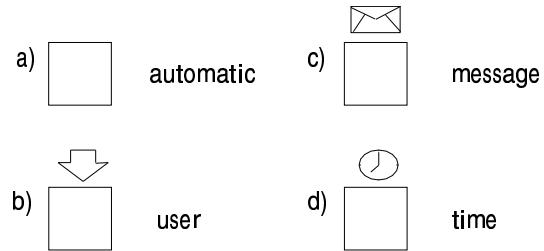


Abbildung 8.8: Trigger eines Workflowsystems

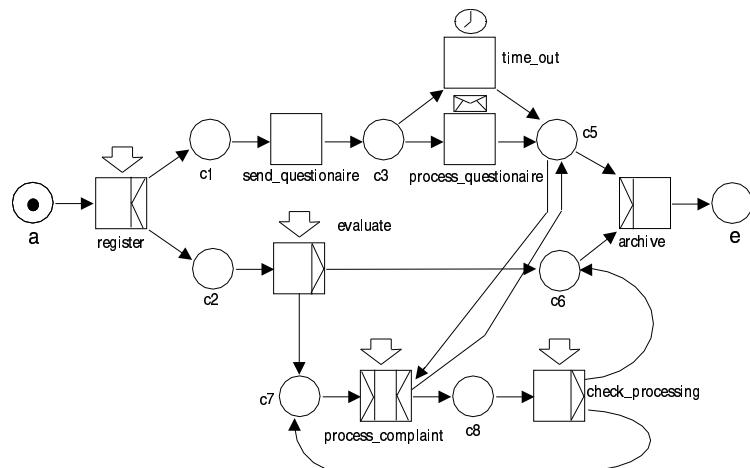


Abbildung 8.9: Workflow aus Abb. 8.3 mit Triggern

8.1.3 Korrektheit

Workflow-Systeme werden fehlerhaft entworfen, weil die abzubildenden Prozesse nicht richtig verstanden wurden. Es stellt sich die Frage, ob in diesem Sinne semantisch inkorrekte Entwürfe durch formale Methoden erkannt werden können.

Die Erfahrung aus der Praxis zeigt, dass ...

- Workflow-Prozesse oft nicht richtig verstanden werden (Mehrdeutigkeit, Widersprüche, Verklemmungen),
- allein schon die (versuchsweise) Modellierung durch Petrinetze Mängel aufdeckt und
- bei fertiggestellten Petrinetz-Modellen von Workflow-Systemen Mängel durch strukturelle Untersuchungen aufgedeckt oder durch Werkzeuge (automatisch) gefunden werden.

Beispiel 8.3 Wir betrachten dazu das nicht korrekte Workflow-System von Abbildung 8.10. Es zeigt zum Beispiel die folgenden Fälle von Fehlverhalten:

- Wenn die Transitionen 2 und 5 schalten, dann verbleibt bei Termination (d.h. wenn eine Marke in e ankommt) immer noch eine Marke in c_5 . Das deutet darauf hin, dass ein Teilprozess nicht vollendet wurde. (Die Marke kann auch beim nächsten Durchlauf zu Fehlverhalten führen.)
- Wenn die Transitionen 3 und 4 schalten, dann verbleibt bei Termination eine Marke in c_4 .
- Wenn die Transitionen 2 und 4 schalten, dann erreichen bei Termination 2 Marken den Endplatz e .

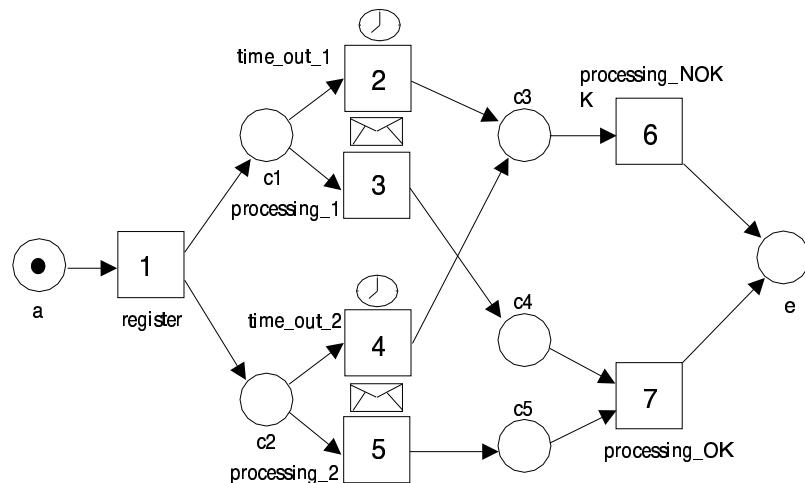


Abbildung 8.10: Problematisches Workfownet für Beschwerdebearbeitung

Da fehlerhafte Workflow-Systeme oft aus inhaltlichen Mißverständnissen entstehen, stellt sich die Frage, ob solche im Grunde semantische Fehler durch formale Kriterien vermieden werden können.

Die Definition 8.4 formalisiert dazu folgende drei Kriterien:

- a) Aus jeder erreichbaren Markierung ist eine ordnungsgemäße Termination möglich.
- b) Genau eine Marke in dem Endplatz e ist die einzige Möglichkeit zu terminieren.
- c) Jede Transition kann in einer möglichen Schaltfolge schalten, denn sonst wäre sie nutzlos.

Definition 8.4 Ein WF-Netz $\mathcal{N} = (P, T, F, \mathbf{m}_a)$ heißt korrekt, falls gilt:

- a) $\forall \mathbf{m} \in \mathbf{R}(\mathcal{N}) \exists w \in T^*: \mathbf{m} \xrightarrow{w} \mathbf{m}_e$ (Termination möglich)
- b) $\forall \mathbf{m} \in \mathbf{R}(\mathcal{N}): \mathbf{m}(e) \geq 1 \Rightarrow \mathbf{m} = \mathbf{m}_e$ (korrekte Termination)
- c) $\forall t \in T \exists \mathbf{m} \in \mathbf{R}(\mathcal{N}): \mathbf{m} \xrightarrow{t}$ (Nützlichkeit)

Das problematische WF-Netz aus Abbildung 8.10 erfüllt Eigenschaft c) – aber weder a) noch b).

8.1.4 Abschluss eines WF-Netzes

Wir transformieren ein WF-Netz, indem wir eine neue Transition t^* zwischen e und a hinzufügen (vgl. Abbildung 8.11) und so das Netz „kurzschießen“:

Definition 8.5 Für ein WF-Netz $\mathcal{N} = (P, T, F, \mathbf{m}_a)$ mit Anfangsplatz a und Endplatz e heißt $\overline{\mathcal{N}} = (P, T', F', \mathbf{m}_a)$ der Abschluss von \mathcal{N} , falls gilt:

- a) $T' := T \cup \{t^*\}$ für eine neue Transition $t^* \notin T$
- b) $F' := F \cup \{(e, t^*), (t^*, a)\}$

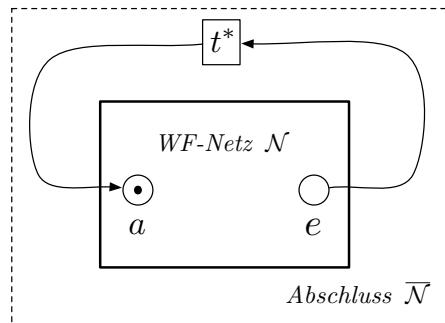


Abbildung 8.11: Transformation eines WF-Netzes: Hinzufügung einer neuen Transition t^*

Aufgabe 8.6 Zeigen Sie, dass für das WF-Netz \mathcal{N} aus Abbildung 8.10 der Abschluss $\overline{\mathcal{N}}$ nicht beschränkt ist (z.B. dadurch dass $\mathbf{R}(\overline{\mathcal{N}})$ nicht endlich ist). Ebenso zeige man, dass $\overline{\mathcal{N}}$ nicht lebendig ist.

Satz 8.7 ([Aal97]) Ein WF-Netz \mathcal{N} ist genau dann korrekt, wenn sein Abschluss $\overline{\mathcal{N}}$ lebendig und beschränkt ist.

Beweis: Als Übung. □

Aufgabe 8.8 Analysieren Sie das WF-Netz von Abb. 8.12 mit Hilfe des Verfahrens nach Satz 8.7.

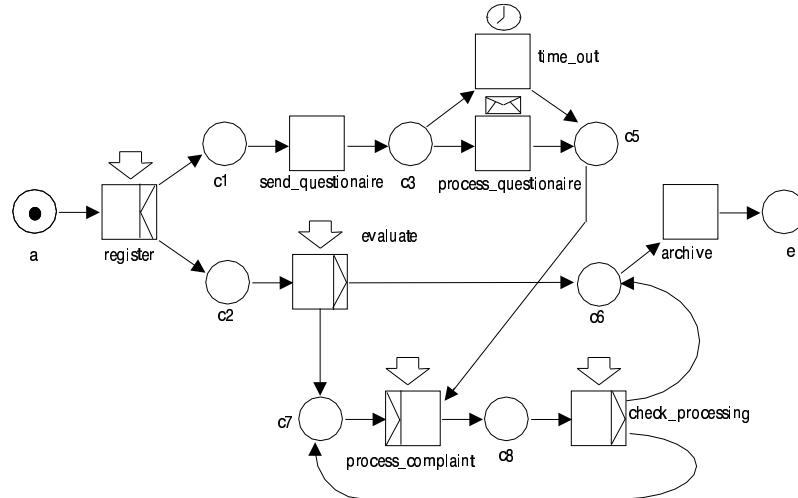


Abbildung 8.12: Ein WF-Netz zur Behandlung von Beschwerden

Die Eigenschaft „korrektes WF-Netz“ ist ein gutes Beispiel dafür, wie eine domänen spezifische Eigenschaft durch eine einfache Transformation auf eine basale Petrinetzeigenschaft zurückgeführt werden kann, die intensiv untersucht wurde und für die Software-Werkzeuge bereitstehen. Es handelt sich um die Eigenschaften „beschränkt“ und „lebendig“, die in der Tabelle 6.1 auf Seite 100 definiert werden und zu denen im selben Kapitel Entscheidungsalgorithmen entwickelt werden. Beschränktheit bedeutet, dass es eine obere Schranke für die Anzahl der Marken auf den Plätzen gibt. Diese Eigenschaft ist damit äquivalent, dass die Erreichbarkeitsmenge $\mathbf{R}(\mathcal{N})$ endlich ist. Lebendigkeit bedeutet dagegen, dass von jeder erreichbaren Markierung $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ jede Transition über eine geeignete Schaltfolge aktiviert werden kann. Intuitiv bedeutet dies, dass – egal wie sich das System verhält – nie Teile des Systems permanent ausfallen oder terminiert haben.

8.2 Das Bankiersproblem: Sichere Zustände

Die Problematik von Verklemmungen bei der Betriebsmittelvergabe wurde von Dijkstra als Problem des Bankiers dargestellt [Dij68]. Hier wurde auch der Begriff des sicheren 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 $\iota = (n, f, g)$ des Bankiersproblems besteht aus einer Zahl $n \in \mathbb{N}$, einem n -Tupel $f = (f_1, \dots, f_n)$ und einer Zahl $g \in \mathbb{N}$. Ein Zustand einer solchen Instanz ist ein n -tupel $r = (r_1, \dots, r_n)$ das die Restforderung der Kunden beschreibt. Anfangs gilt $r = f$. Ein Zustand heißt *sicher*, wenn er nicht notwendig zu einer Verklemmung führt.

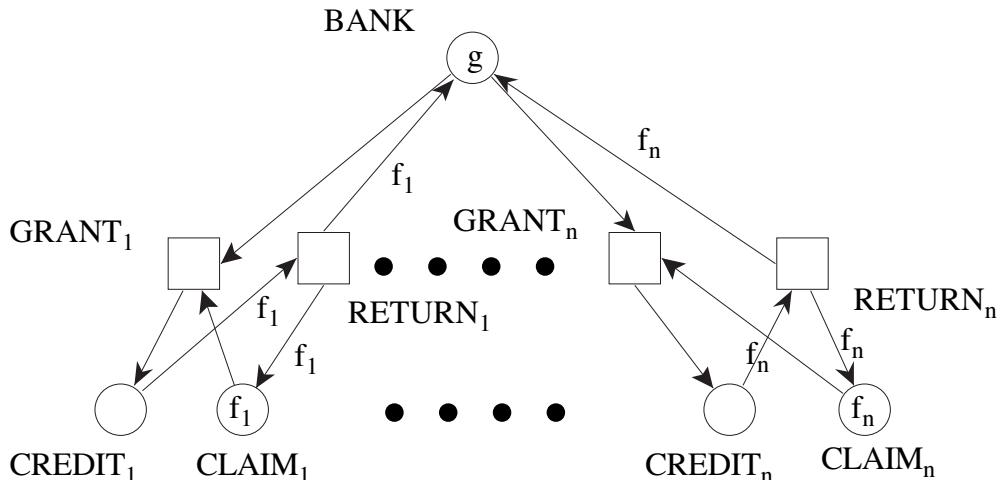


Abbildung 8.13: Das Bankiersproblem mit n Kunden

Das P/T-Netz in Abb. 8.13 modelliert das beschriebene Bankiersproblem. Der Platz *BANK* enthält so viele Marken wie das Kapital des Bankiers in Geldeinheiten umfasst. *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.

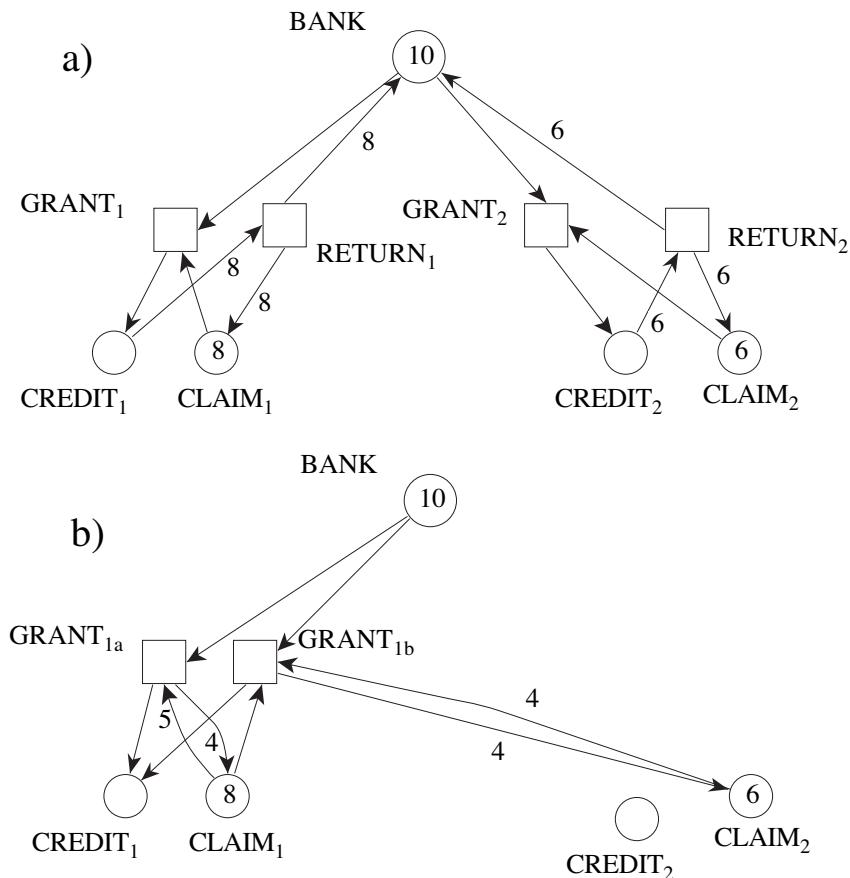


Abbildung 8.14: Eine Bankiersprobleminstanz mit 2 Kunden und Modifikation zur Vermeidung von Verklemmungen

Betrachten wir zwei spezielle Instanzen, nämlich $\iota_1 = (2, (8, 6), 10)$ (Abb. 8.14) und $\iota_2 = (3, (8, 3, 9), 10)$ (Abb. 8.16). An ihnen werden Erreichbarkeitsgraph und P-Invarianten erläutert.

Für die Instanz $\iota_1 = (2, (8, 6), 10)$ in Abb. 8.14a wird jeder Zustand durch eine Markierung dargestellt, d.h. durch einen 5-dimensionalen Vektor. Für alle erreichbaren Markierungen \mathbf{m} gelten die folgenden 3 Gleichungen:

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

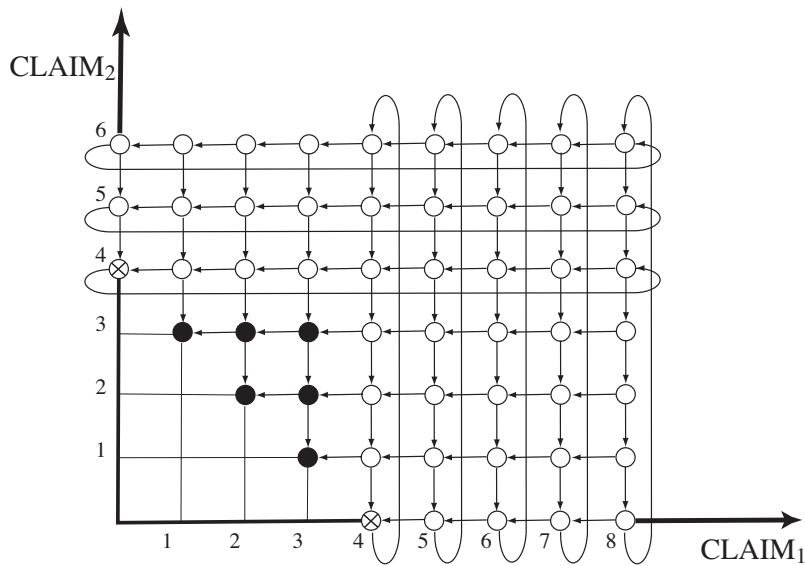


Abbildung 8.15: Erreichbarkeitsgraph des Netzes 8.14

(Sie heißen P-Invarianten-Gleichungen und werden im nächsten Abschnitt 7.5 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 Erreichbarkeitsgraph in einem ebenen Gitter dargestellt werden (Abb. 8.15).

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 Graphen von Abb. 8.15.

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, 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: $(0,4)$ und $(4,0)$, (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.

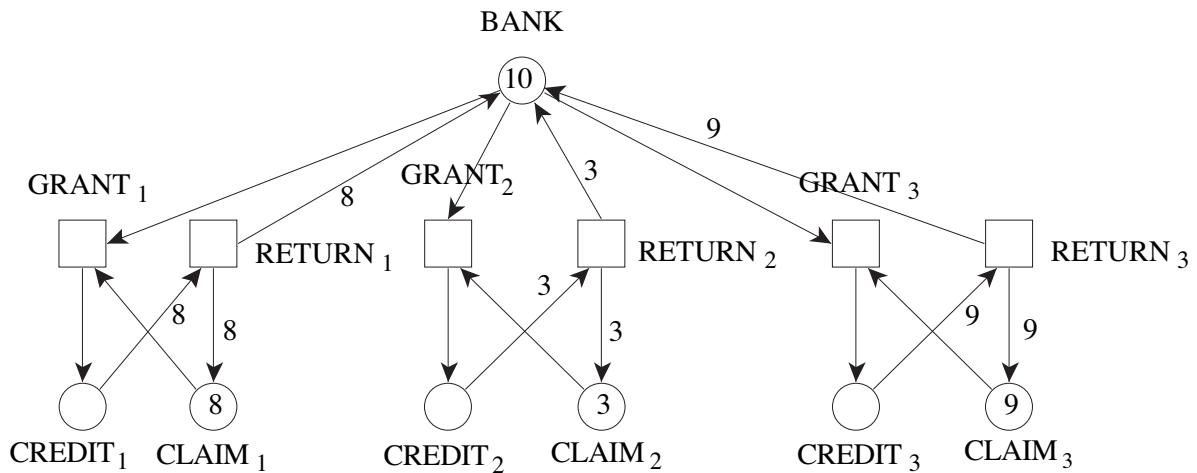


Abbildung 8.16: Eine Instanz des Bankiersproblems mit 3 Kunden

Wie aus Abb. 8.15 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. 8.14b)³ 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 8.16. Es enthält 7 Plätze. Wegen der nun 4 Gleichungen kann der Erreichbarkeitsgraph in drei Dimensionen dargestellt werden (siehe dazu auch Abb. 8.18). 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 Markierungen 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 nicht einmal einen Ausleihvorgang vollständig zu Ende bringen können. Solche Situationen heißen partielle Verklemmung. Ein Netz ohne partielle Verklemmungen heißt *lebendig*. Lebendigkeit wird im Abschnitt 7.1.2 behandelt.

³Diese Abbildung zeigt nur, wie $GRANT_1$ durch zwei Transitionen zu ersetzen ist. Entsprechendes muss auch auf $GRANT_2$ angewandt werden, nicht jedoch auf $RETURN_1$ und $RETURN_2$.

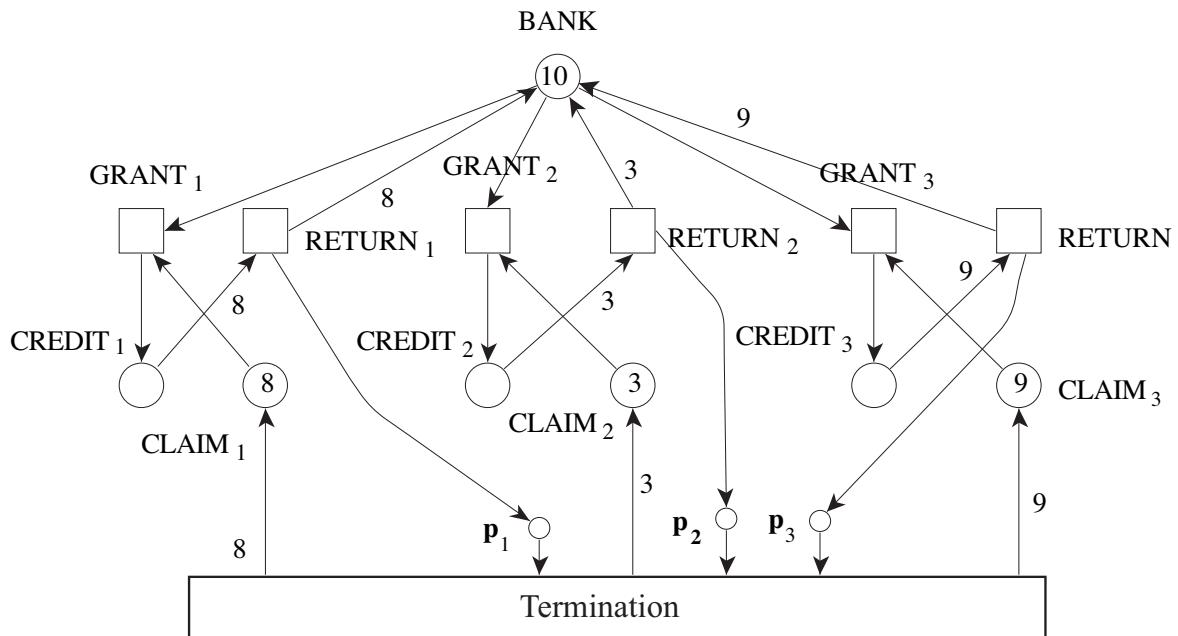


Abbildung 8.17: Bankiersnetz mit Terminationstransition

Um die ursprüngliche Definition von Dijkstra beizubehalten, kann man wie in Abb. 8.17 eine Transition *Termination* einführen, die dafür sorgt, dass alle Kunden einen Ausleihvorgang beendet haben, bevor eine neue Runde started. Die Einschränkungen der möglichen Abläufe ergibt sich aus der expliziten Synchronisation. Um den Begriff der sicheren Markierung sinnvoll auch in dem Netz 8.16 zu benutzen, könnte man folgende Definitionen unter c) benutzen, die eine sinnvolle Übertragungen des Begriffs der Sicherheit auf Netze liefern:

- Eine Markierung \mathbf{m} heißt sicher, wenn eine spezifizierte Endmarkierung (hier die Anfangsmarkierung) wieder erreichbar ist.
- Eine Markierung \mathbf{m} heißt sicher, wenn eine bestimmte Transition (z.B. *Termination*) zum Schalten gebracht werden kann.
- Eine Markierung \mathbf{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 Petrinetzzlitteratur auch „homing property“ genannt. Die Eigenschaft c) hat mit dem „fairen Verhalten“ eines Netzes zu tun. Fairness und Lebendigkeit stehen in komplexen Beziehungen zueinander.

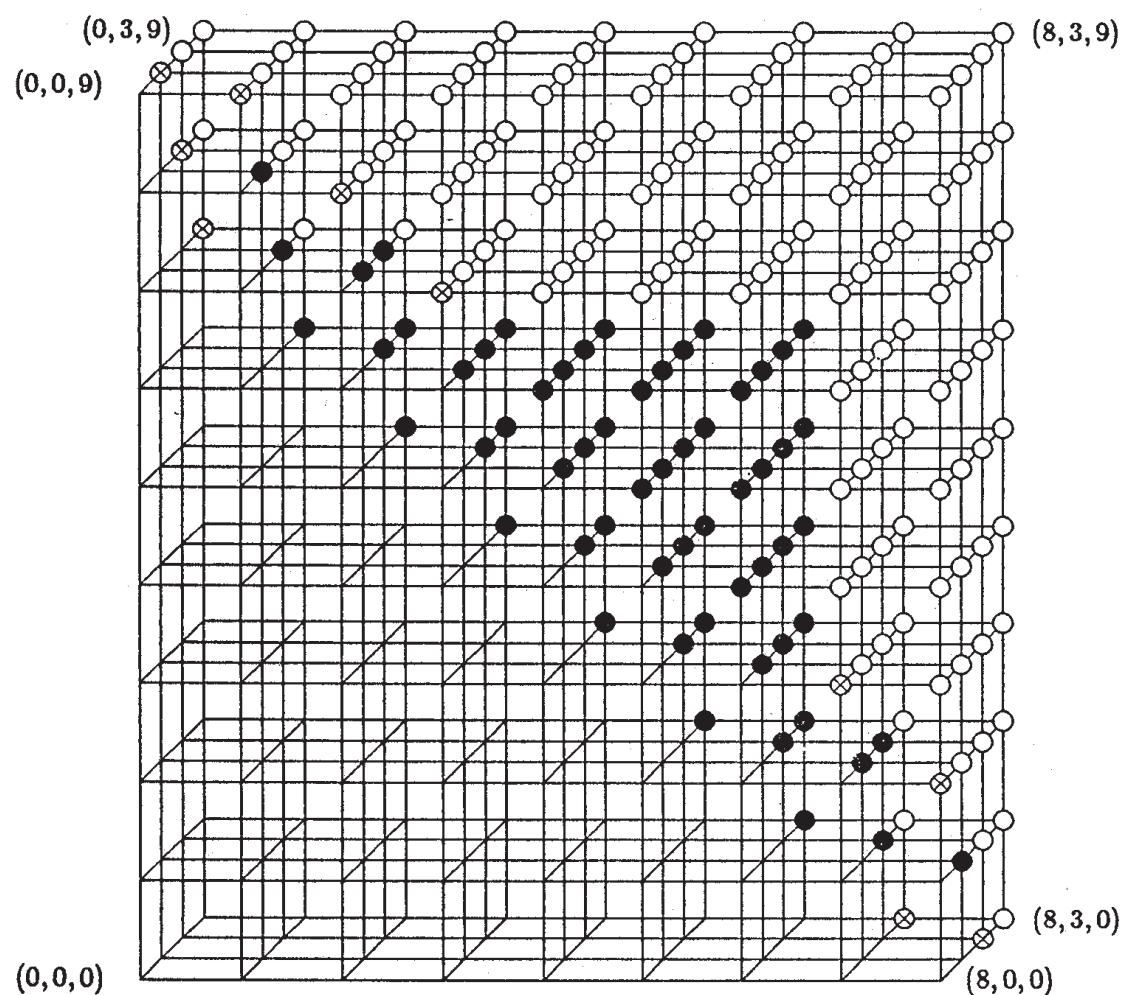


Abbildung 8.18: Erreichbarkeitsgraph des Netzes 8.16

8.3 Kantenkonstante Netze

In diesem Kapitel werden Platz/Transitions-Netze zu solchen Netzen erweitert, die statt der einfachen, „schwarzen“ Marken, allgemeinere Datentypen wie `integer` o.ä. erlauben. Dies erlaubt große Netze in sehr viel kleinerer und kompakterer Form darzustellen, wobei viele Eigenschaften der Platz/Transitions-Netze übertragbar sind.⁴ Bücher zu gefärbten Netzen sind insbesondere [GV03] (Kapitel 1 bis 4), [JK09], [Rei10] und der Artikel [Kum01] zu Referenznetzen.

Als erster Schritt beim Übergang von Platz/Transitions- zu gefärbten Netzen werden *kantenkonstanten Netze* betrachtet, die zwar schon die genannten Farbmengen haben, aber noch keine Verschmelzung von ähnlichen Transitionen erlauben. Dies wird erst im nächsten Schritt durch die Einführung von Variablen als Kantengewicht möglich (Abschnitt 8.4). Anschließen wird in Abschnitt 8.5 das Werkzeug RENEW kurz beschrieben, das den graphischen Entwurf unterstützt und eine Ausführung des Schaltens erlaubt.

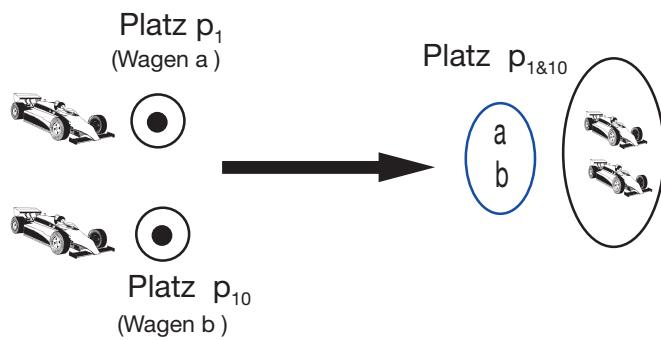
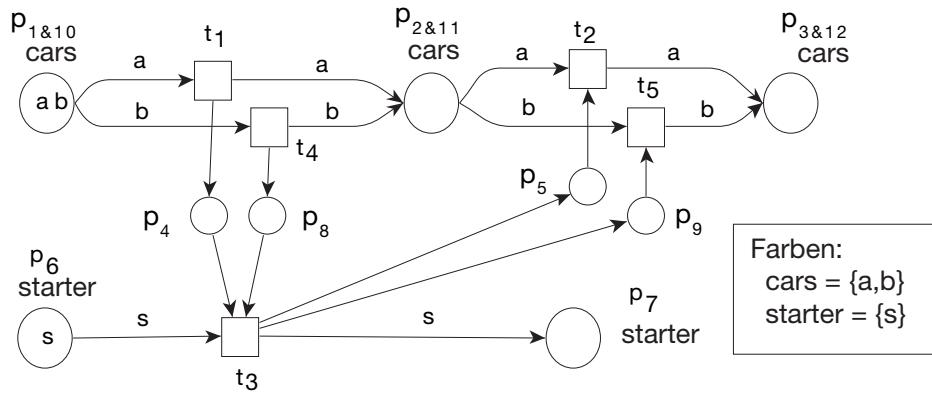


Abbildung 8.19: Übergang zu individuellen Marken

In dem Beispielnetz von Abbildung 6.5 (auf Seite 83) wurden die Objekte *Wagen a* und *Wagen b* durch einfache Marken repräsentiert. Diese Marken sind an sich nicht zu unterscheiden. In dem Netz 6.5 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. 8.19 durch den Platz $p_{1\&10}$ gezeigt wird. Dieser Platz repräsentiert gleichsam die Aufstellungszone für den Rennbeginn.

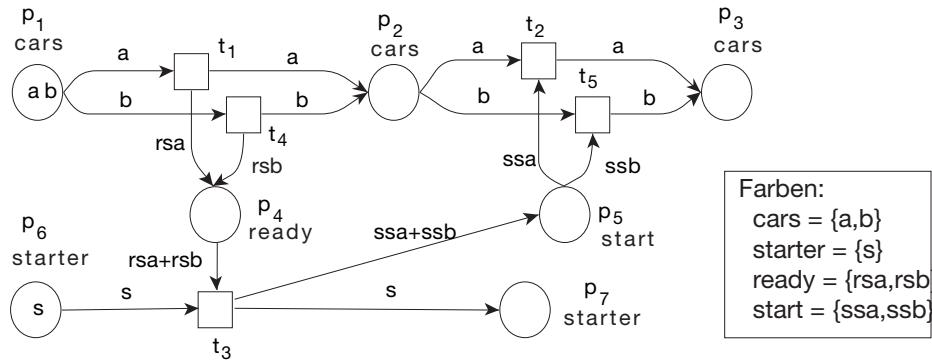
Unterscheidbare Marken werden als Sorten oder Typen gruppiert, die Farben heißen. Jedem Platz p wird durch $cd(p)$ (engl. colour domain) eine Farbe zugeordnet, hier also $cd(p_{1\&10}) = cars = \{a, b\}$ (in der grafischen Darstellung kursiv beim Bezeichner des Platzes). Andere Beispiele von Farben sind `integer` oder `boolean`. Wie im Netz 8.20 zu se-

⁴ Dies gilt jedoch nicht für die Eigenschaft der Beschränktheit, die (wie in Abschnitt 7.4 gezeigt) hier nicht entscheidbar ist. Enthält das Netz jedoch nur endliche Datentypen (die hier *Farben* genannt werden), so sind gefärbte Netze gleichmäßig zu Platz/Transitions-Netzen (die Beschränktheit ist dann z.B. entscheidbar) und der genannte Kompaktifizierungs-Vorteil bleibt bestehen.


 Abbildung 8.20: Das kantenkonstante gefärbte Netz \mathcal{N}_1

hen, spezifizieren Ausdrücke an Kanten wie “ a ”, welche Marke entfernt bzw. hinzugefügt 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.


 Abbildung 8.21: Das kantenkonstante gefärbte Netz \mathcal{N}_2

Das Netz 8.21 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.

Definition 8.9 Eine Multimenge bg (engl. bag, multi-set), über einer nicht leeren Menge A , ist eine Abbildung $bg : A \rightarrow \mathbb{N}$, die auch als formale Summe $\sum_{a \in A} bg(a)'a$ dargestellt wird.

\emptyset bezeichnet die leere Multimenge mit $\emptyset(a) = 0$ für alle $a \in A$.

$Bag(A)$ bezeichnet die Menge aller Multimengen über A .

$|bg| := \sum_{a \in A} bg(a)$ ist die Mächtigkeit (oder Kardinalität) von bg .

Die Mengenoperationen Vereinigung, Inklusion und Differenz werden punktweise auf $Bag(A)$ als Summe (+), Inklusion (\leq) und Differenz ($-$) übertragen:

- $bg_1 + bg_2 := \sum_{a \in A} (bg_1(a) + bg_2(a))'a$
- $bg_1 \leq bg_2 : \iff \forall a \in A : bg_1(a) \leq bg_2(a)$
- $bg_1 - bg_2 := \sum_{a \in A} (Max(bg_1(a) - bg_2(a), 0))'a$

Beispiel 8.10 $bg_1 = \{a, a, b, b, b, d\}_b$ ist Multimenge über $A = \{a, b, c, d\}$: (der Index b 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 $bg_1 + bg_2 = 3'a + 5'b + c + d$ und $bg_1 - bg_2 = 1'a + 1'b + 0'c + 1'd = a + b + d$.

Multimengen wie $\{a, b, d\}_b$, die Mengen sind, werden auch als Mengen dargestellt: $\{a, b, d\}$

Definition 8.11 Ein kantenkonstantes gefärbtes Petrinetz (KKN) wird als Tupel $\mathcal{N} = (P, T, F, \mathcal{C}, cd, W, \mathbf{m}_0)$ definiert, wobei

- (P, T, F) ein endliches Netz (Def. 6.1) ist,
- \mathcal{C} ist eine Menge von Farbenmengen,
- $cd: P \rightarrow \mathcal{C}$ ist die Farbzuweisungsabbildung (colour domain mapping). Sie wird durch $cd: F \rightarrow \mathcal{C}$, $cd(x, y) := \text{if } x \in P \text{ then } cd(x) \text{ else } cd(y) \text{ fi}$ auf F erweitert.
- $W: F \rightarrow Bag(\bigcup \mathcal{C})$ mit $W(x, y) \in Bag(cd(x, y))$ heißt Kantengewichtung.
- $\mathbf{m}_0: P \rightarrow Bag(\bigcup \mathcal{C})$ mit $\mathbf{m}_0(p) \in Bag(cd(p))$ für alle $p \in P$ ist die Anfangsmarkierung.

Für das kantenkonstante Netz 8.21 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 8.12 Sei $\mathcal{N} = (P, T, F, \mathcal{C}, cd, W, \mathbf{m}_0)$ ein KKN.

a) Die Markierung des KKN \mathcal{N} ist ein Vektor \mathbf{m} mit $\mathbf{m}(p) \in Bag(cd(p))$ für jedes $p \in P$.

(Dies kann auch als Abbildung $\mathbf{m}: P \rightarrow Bag(\bigcup \mathcal{C})$ mit $\mathbf{m}(p) \in Bag(cd(p))$ für jedes $p \in P$ aufgefasst werden.)

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) Die Erweiterung von W auf $(P \times T) \cup (T \times P)$ ist definiert als:

$$\widetilde{W}(x, y) := \begin{cases} W(x, y) & \text{falls } (x, y) \in F \\ \emptyset & \text{sonst} \end{cases}$$

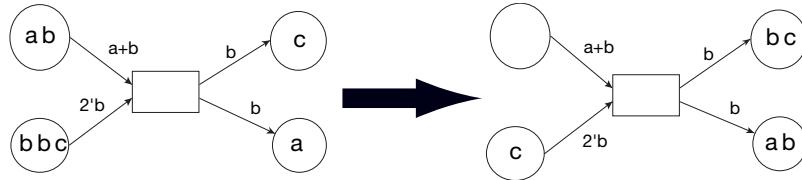


Abbildung 8.22: Schaltregel für kantenkonstante Netze

Ist t in \mathbf{m} aktiviert, dann ist die Nachfolgemarkierung definiert durch $\mathbf{m}'(p) = (\mathbf{m}(p) - \widetilde{W}(p, t)) + \widetilde{W}(t, p))$. (Beachte, dass es sich um Multimengenoperationen handelt!)

d) Wir definieren $W(\bullet, t) := (\widetilde{W}(p_1, t), \dots, \widetilde{W}(p_{|P|}, t))$ als Vektor der Länge $|P|$ – und sinngemäß ebenso $W(t, \bullet)$.

Aktivierung ergibt sich dann als: $\mathbf{m} \xrightarrow{t} \iff \mathbf{m} \geq W(\bullet, t)$.

Die Nachfolgemarkierung ist $\mathbf{m}' = \mathbf{m} - W(\bullet, t) + W(t, \bullet)$.

Dabei sind die Multimengenoperatoren komponentenweise auf Vektoren zu erweitern.

Beispiel 8.13 Für das Netz 8.21 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

$$\begin{aligned} \mathbf{m}' &= \mathbf{m}_0 + W(t_1, \bullet) - W(\bullet, t_1) \\ &= (a + b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset) + (\emptyset, a, \emptyset, rsa, \emptyset, \emptyset, \emptyset) - (a, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \\ &= (b, a, \emptyset, rsa, \emptyset, s, \emptyset) \end{aligned}$$

$a + b$ bzw rsa sind hier beispielsweise die Multimengen $\{a, b\}_b$ bzw. $\{rsa\}_b$, dargestellt als formale Summen.

Die Schaltregel wird in Abb. 8.22 an einem abstrakten Beispiel erläutert, wobei Multimengen vorkommen, die keine Mengen sind!

Definition 8.14 Die Nachfolgemarkierungsrelation von Definition 8.12 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 $t \in T$.

Für eine Menge S^0 von Markierungen sei dann $\mathbf{R}(\mathcal{N}, S^0) := \{\mathbf{m}_2 | \exists w \in T^*, \mathbf{m}_1 \in S^0 : \mathbf{m}_1 \xrightarrow{w} \mathbf{m}_2\}$ die Menge der von S^0 aus erreichbaren Markierungen und $\mathbf{R}(\mathcal{N}) := \mathbf{R}(\mathcal{N}, \{\mathbf{m}_0\})$ die Erreichbarkeitsmenge von $\mathbf{R}(\mathcal{N})$.

Falls \mathcal{N} implizit gegeben ist, schreiben wir auch $\mathbf{R}(\mathbf{m})$ für $\mathbf{R}(\mathcal{N}, \{\mathbf{m}\})$. Eine Transitionsfolge $w \in T^*$ heißt aktiviert in \mathbf{m} (in Zeichen: $\mathbf{m} \xrightarrow{w}$), falls $\exists \mathbf{m}_1 : \mathbf{m} \xrightarrow{w} \mathbf{m}_1$.

$FS(\mathcal{N}) := \{w \in T^* \mid \mathbf{m}_0 \xrightarrow{w}\}$ ist die Menge der Schaltfolgen (firing sequence set) von \mathcal{N} .

Der Erreichbarkeitsgraph eines KKN \mathcal{N} ist ein Tupel $RG(\mathcal{N}) := (Kn, Ka)$ mit Knotenmenge $Kn := \mathbf{R}(\mathcal{N})$ und Kantenmenge $Ka := \{(\mathbf{m}_1, t, \mathbf{m}_2) \mid \mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2\}$

Damit ist die Semantik eines kantenkonstanten gefärbten Petrinetzes durch ein Transitionssystem definiert: Der Erreichbarkeitsgraph eines kantenkonstanten gefärbten Netzes \mathcal{N} kann als Transitionssystem $TS = (\mathbf{R}(\mathcal{N}), T, Ka, \{\mathbf{m}_0\})$ aufgefasst werden. Dann ist $R(TS) = \mathbf{R}(\mathcal{N})$ und $FS(TS) = FS(\mathcal{N})$.

Beispiel 8.15 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.

$$\begin{pmatrix} \{a, b\} \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \{s\} \\ \emptyset \end{pmatrix} \xrightarrow{t_4} \begin{pmatrix} \{a\} \\ \{b\} \\ \emptyset \\ \{rsb\} \\ \emptyset \\ \{s\} \\ \emptyset \end{pmatrix} \xrightarrow{t_1} \begin{pmatrix} \emptyset \\ \{a, b\} \\ \emptyset \\ \{rsa, rsb\} \\ \emptyset \\ \{s\} \\ \emptyset \end{pmatrix} \xrightarrow{t_3} \begin{pmatrix} \emptyset \\ \{a, b\} \\ \emptyset \\ \emptyset \\ \{ssa, ssb\} \\ \emptyset \\ \{s\} \end{pmatrix} \xrightarrow{t_2} \begin{pmatrix} \emptyset \\ \{b\} \\ \{a\} \\ \emptyset \\ \{ssb\} \\ \emptyset \\ \{s\} \end{pmatrix} \xrightarrow{t_5} \begin{pmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \{a, b\} \\ \emptyset \\ \emptyset \\ \{s\} \end{pmatrix}$$

Abbildung 8.23 zeigt beispielhaft den Erreichbarkeitsgraphen des kantenkonstanten Netzes von Abb. 8.21.

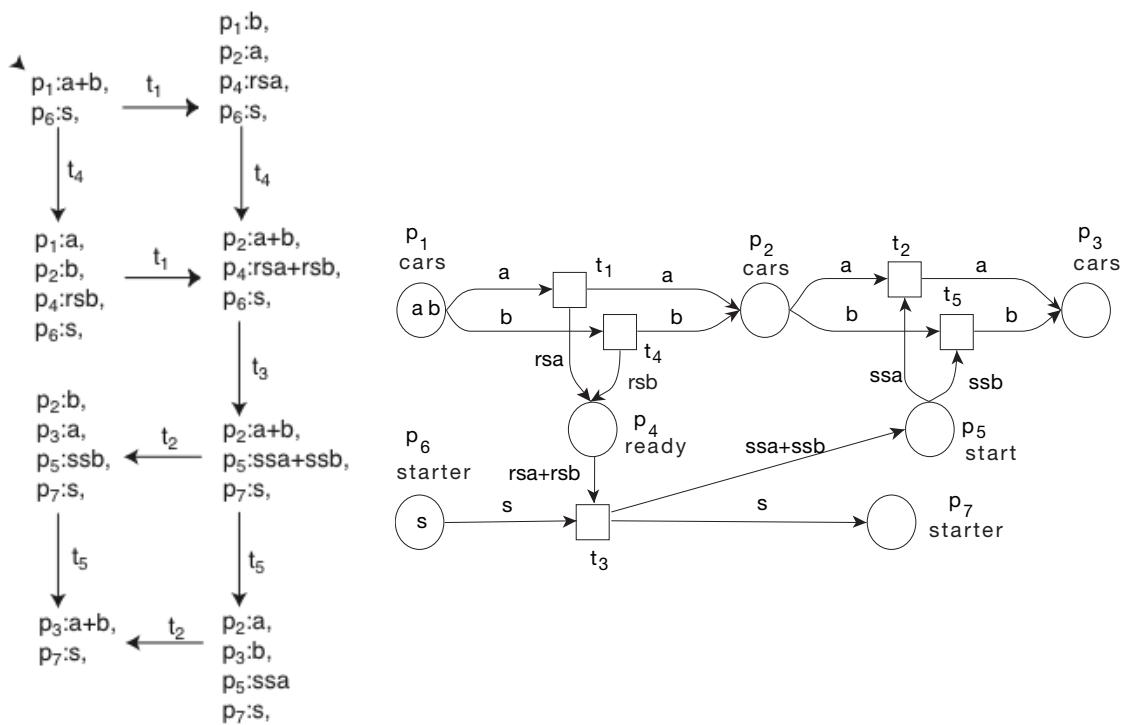


Abbildung 8.23: Erreichbarkeitsgraphen des kantenkonstanten Netzes von Abb.8.21

8.4 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.

Die Zusammenfassung von Plätzen beim Übergang von einfachen Netzen über kantenkonstante Netze zu gefärbten Netzen ist im oberen Teil von Abbildung 8.24 als Vergrößerung von Plätzen („Platz-Faltung“) dargestellt.

Die Zusammenfassung kann durch Variablen geschehen. Beispielsweise modelliert die Transition t_1 im Netz 8.25 durch die Variablenbelegung $\beta_1 := [x = a, y = rsa]$ die Transition t_1 im Netz 8.21, während mit $\beta_2 := [x = b, y = rsb]$ die Transition t_4 in 8.21 simuliert wird.

Definition 8.16 Sei $Var := \{x_1, x_2, x_3, \dots\}$ eine Menge von Variablen mit Wertebereichen $dom(x)$ für jedes $x \in Var$. Eine Belegung ist eine Zuordnung (Abbildung) $\beta = [x_1 = u_1, x_2 = u_2, x_3 = u_3, \dots]$ von Werten $u_i \in dom(x_i)$ zu den Variablen.

Natürlich wäre $\beta_3 := [x = a, y = rsb]$ keine zulässige Belegung, denn 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 8.17 Ein gefärbtes Petrinetz (engl. coloured Petri net, CPN) wird als Tupel

$$\mathcal{N} = (P, T, F, \mathcal{C}, cd, Var, Guards, \widehat{W}, \mathbf{m}_0)$$

definiert, wobei gilt:

- (P, T, F) ist ein endliches Netz (Def. 6.1),
- \mathcal{C} ist eine Menge von Farbenmengen,
- $cd: P \rightarrow \mathcal{C}$ ist die Farbzueweisungsabbildung (colour domain mapping). Sie wird durch $cd: F \rightarrow \mathcal{C}$, $cd(x, y) := \text{if } x \in P \text{ then } cd(x) \text{ else } cd(y) \text{ fi}$ 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.

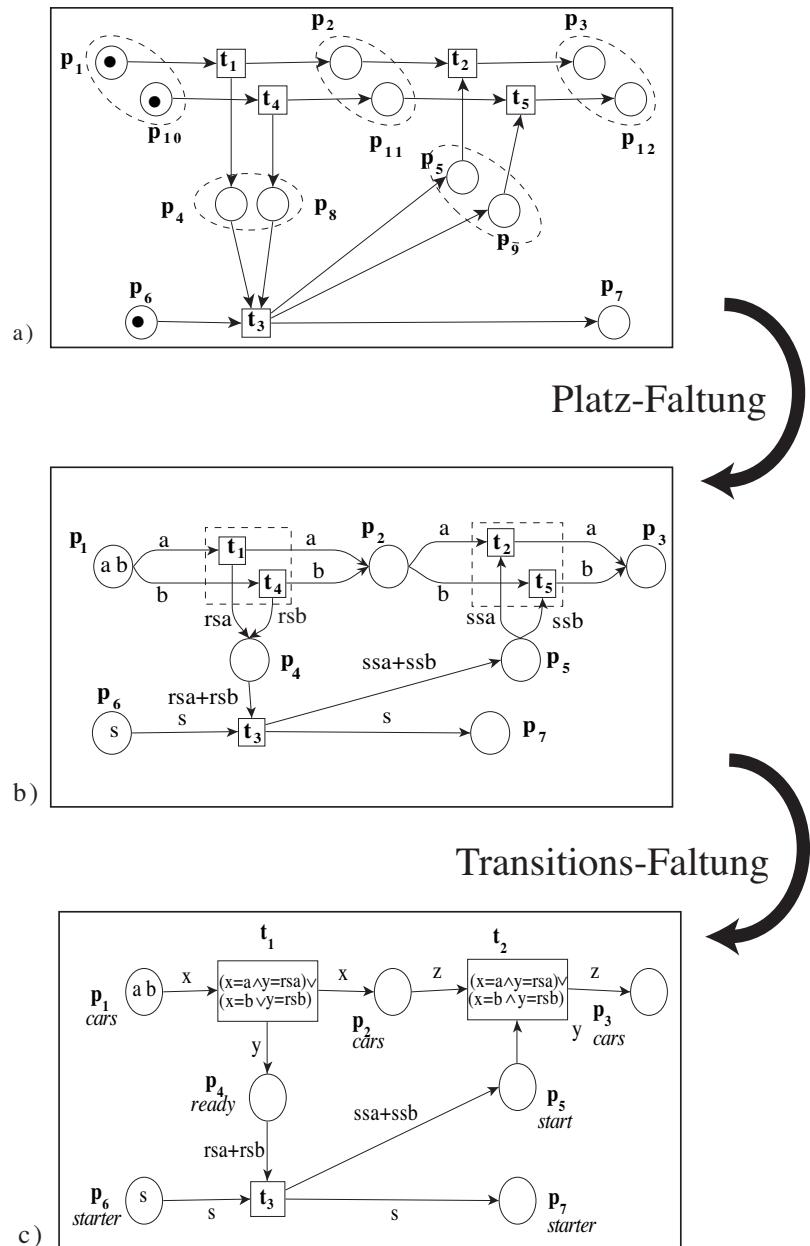


Abbildung 8.24: Netzfaltung zum kantenkonstanten und gefärbten Netz

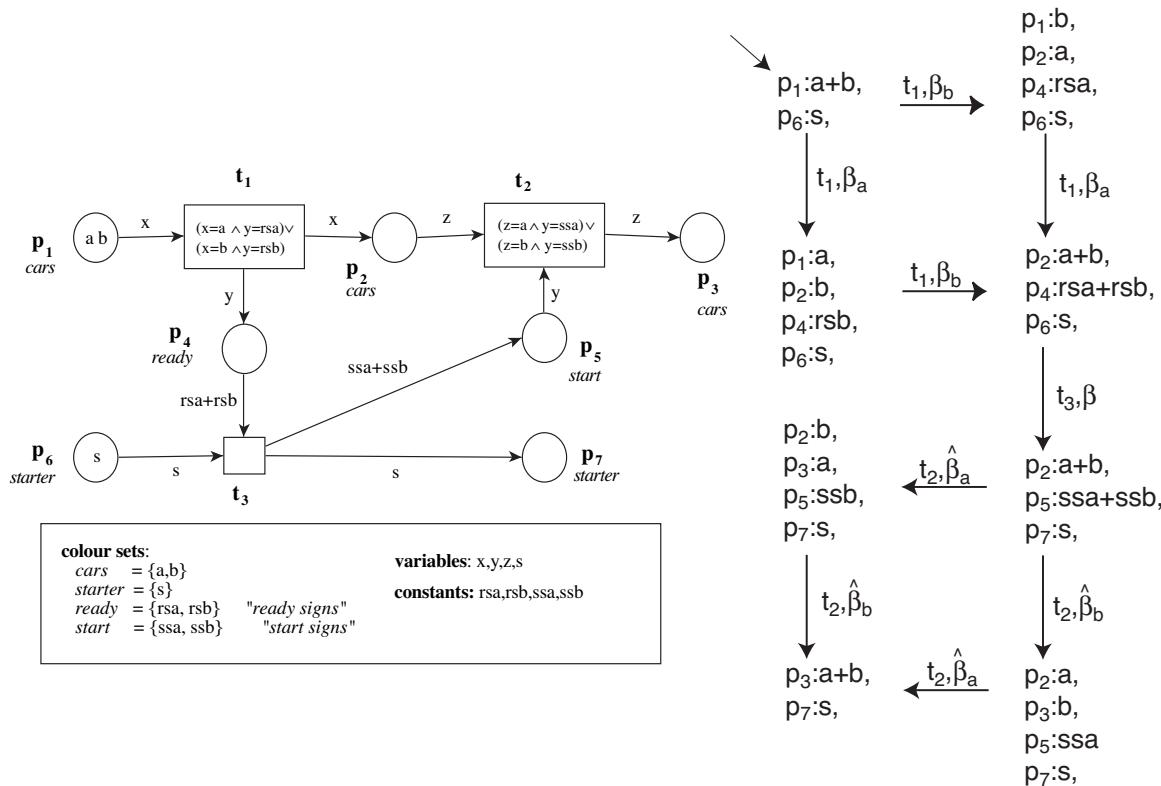


Abbildung 8.25: Gefärbtes Netz und sein Erreichbarkeitsgraph

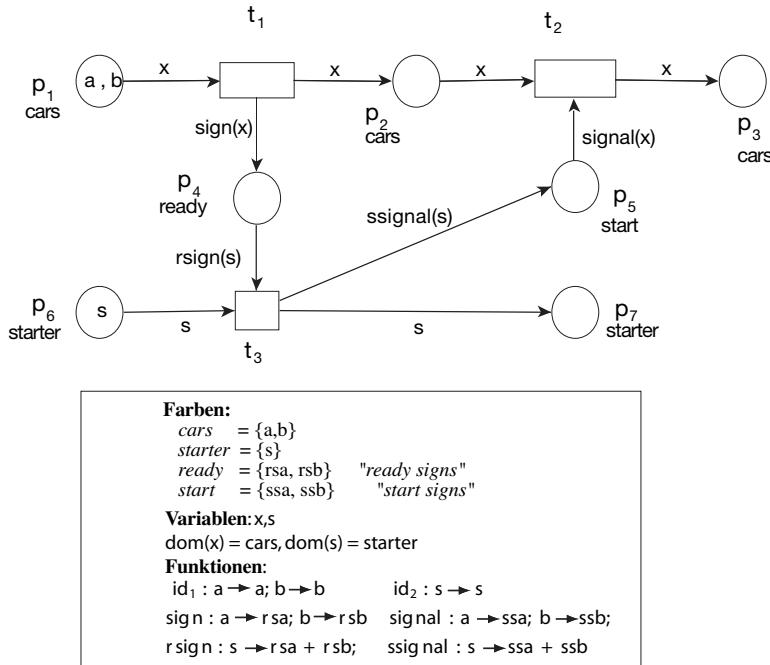


Abbildung 8.26: Gefärbtes Netz mit definierten Funktionen

- $\widehat{W} = \{W_\beta \mid \beta \text{ ist Belegung von } \text{Var}\}$ ist eine Menge von Kantengewichtungen der Form $W_\beta : F \rightarrow \text{Bag}(\bigcup \mathcal{C})$, wobei $W_\beta(x, y) \in \text{Bag}(\text{cd}(x, y))$ für alle $(x, y) \in F$ gilt.
- $\mathbf{m}_0 : P \rightarrow \text{Bag}(\bigcup \mathcal{C})$ mit $\mathbf{m}_0(p) \in \text{Bag}(\text{cd}(p))$ für alle $p \in P$ ist die Anfangsmarkierung.

Kantenausdrücke von gefärbten Netzen können auch speziell definierte Funktionen enthalten. Diese werden wie in Abb. 8.26 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.

Definition 8.18 Sei $\mathcal{N} = (P, T, F, \mathcal{C}, \text{cd}, \text{Var}, \text{Guards}, \widehat{W}, \mathbf{m}_0)$ ein gefärbtes Netz.

- Die Markierung des gefärbten Netzes \mathcal{N} ist ein Vektor \mathbf{m} mit $\mathbf{m}(p) \in \text{Bag}(\text{cd}(p))$ für jedes $p \in P$
 (auch als Abbildung $\mathbf{m} : P \rightarrow \text{Bag}(\bigcup \mathcal{C})$ mit $\mathbf{m}(p) \in \text{Bag}(\text{cd}(p))$ für jedes $p \in P$ aufzufassen).
- Sei β eine Belegung für Var . Die Transition $t \in T$ heißt β -aktiviert in einer Markierung \mathbf{m} , falls $\text{guard}_t(\beta) = \text{true}$ und $\forall p \in \bullet t. \mathbf{m}(p) \geq W_\beta(p, t)$
 (als Relation: $\mathbf{m} \xrightarrow{t, \beta}$).
- Die Erweiterung von W_β auf $(P \times T) \cup (T \times P)$:

$$\widetilde{W}_\beta(x, y) := \begin{cases} W_\beta(x, y) & \text{falls } (x, y) \in F \\ \emptyset & \text{sonst} \end{cases}$$

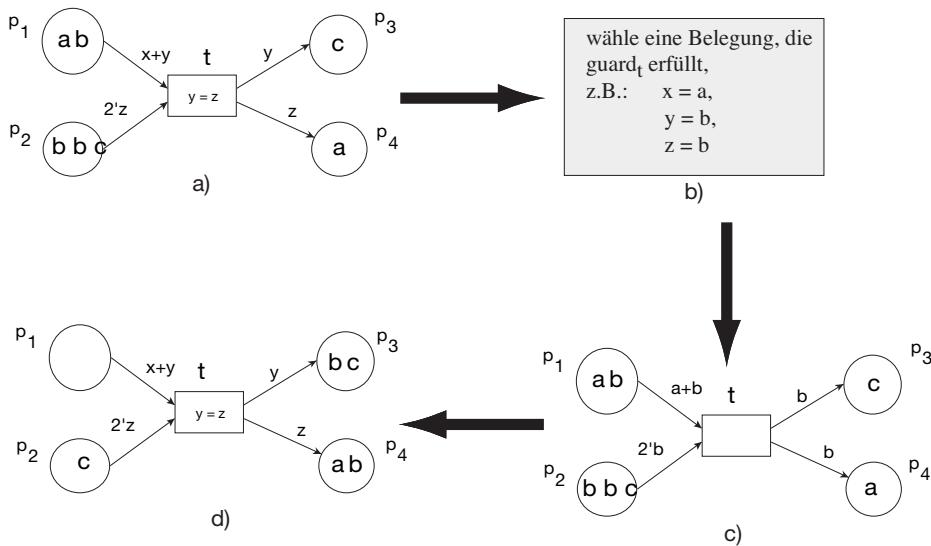


Abbildung 8.27: Schaltregel für gefärbte Netze

Ist t in \mathbf{m} β -aktiviert, dann ist die Nachfolgemarkierung definiert durch $\mathbf{m}'(p) = \mathbf{m}(p) - \widetilde{W}_\beta(p, t) + \widetilde{W}_\beta(t, p)$. (Man beachte, dass es sich um Multimengenoperationen handelt.)

- d) Definiert man $W_\beta(\bullet, t) := (\widetilde{W}_\beta(p_1, t), \dots, \widetilde{W}_\beta(p_{|P|}, t))$ als Vektor der Länge $|P|$ und entsprechend $W_\beta(t, \bullet) := (\widetilde{W}_\beta(t, p_1), \dots, \widetilde{W}_\beta(t, p_{|P|}))$, dann kann die Nachfolgemarkierung einfacher durch Vektoren definiert werden: $\mathbf{m}' = \mathbf{m} - W_\beta(\bullet, t) + W_\beta(t, \bullet)$. Dabei sind die Multimengenoperatoren komponentenweise auf Vektoren zu erweitern.
- e) Schalten wird auch ohne Belegung notiert: $\mathbf{m} \xrightarrow{t} \mathbf{m}' : \iff \exists \beta. \mathbf{m} \xrightarrow{t, \beta} \mathbf{m}'$

Die Schaltregel für CPN entspricht der für KKN, wenn man sich (für den Moment des Schaltens) alle Ausdrücke durch die Belegung β als ausgewertet vorstellt und dann mit der Schaltregel für KKN schaltet. Diese Idee illustriert Abb. 8.27.

1. Für die Transition t in Abb. 8.27 a) wählen eine Belegung β für $Var(t)$ mit $guard_t(\beta) = true$ (wie in Abb. 8.27 b)).
2. Wir werten mit dieser Belegung die Ausdrücke an den Kanten zu Multimengen aus (wie in Abb. 8.27 c)).
3. Wir wenden die Schaltregel für kantenkonstante Netze (Abb. 8.22) an (wie in Abb. 8.27d)).

Die Definition von Schaltfolgen, erreichbare Markierungen usw. ergeben sich dann ganz analog zu KKN:

Definition 8.19 Die Nachfolgemarkierungsrelation von Definition 8.18 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 $\mathbf{R}(\mathcal{N}) := \{\mathbf{m} \mid \exists w \in T^* : \mathbf{m}_0 \xrightarrow{w} \mathbf{m}\}$ ist die Menge der erreichbaren Markierungen (auch: Erreichbarkeitsmenge).

Eine Transitionsfolge $w \in T^*$ heißt aktiviert in \mathbf{m} (in Zeichen: $\mathbf{m} \xrightarrow{w}$), falls gilt: $\exists \mathbf{m}_1 : \mathbf{m} \xrightarrow{w} \mathbf{m}_1$

$FS(\mathcal{N}) := \{w \in T^* \mid \mathbf{m}_0 \xrightarrow{w}\}$ ist die Menge der Schaltfolgen von \mathcal{N} .

Der Erreichbarkeitsgraph eines gefärbten Netzes \mathcal{N} ist der Graph $RG(\mathcal{N}) := (Kn, Ka)$ mit der Knotenmenge $Kn := \mathbf{R}(\mathcal{N})$ (siehe Def. 8.19) und der Kantenmenge $Ka := \{(\mathbf{m}_1, (t, \beta), \mathbf{m}_2) \mid \mathbf{m}_1 \xrightarrow{t, \beta} \mathbf{m}_2\}$.

Der Erreichbarkeitsgraph ist ein Transitionssystem (ohne Endzustände).

Beispiel 8.20 Für das gefärbte Netz 8.25 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 $\mathbf{m}_0(p_1) = 1'a + 1'b \in cd(p_1)$, $guard_{t_1} = (x = a \wedge y = rsa) \vee (x = b \wedge y = rsb)$.

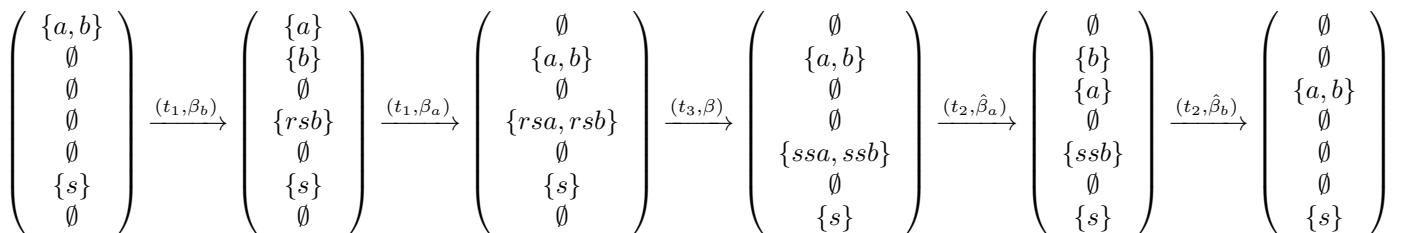
Für das Netz 8.25 ist die Anfangsmarkierung: $\mathbf{m}_0 = (a + b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset)$.

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

$$\begin{aligned} \mathbf{m}' &= \mathbf{m}_0 + W_\beta[t_1, \bullet] - W_\beta(\bullet, t_1) \\ &= (a + b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset) + (\emptyset, a, \emptyset, rsa, \emptyset, \emptyset, \emptyset) - (a, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \\ &= (b, a, \emptyset, rsa, \emptyset, s, \emptyset) \end{aligned}$$

Mit $a+b$ bzw rsa sind hier beispielsweise die Multimengen $\{a, b\}_b$ bzw. $\{rsa\}_b$ als formale Summen dargestellt.

Eine Markierungs-Schaltfolge für das gefärbte Netz von Abb. 8.25 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 kann eine beliebige Belegung eingesetzt werden, da alle Kantenausdrücke nur Konstante enthalten.



Der Erreichbarkeitsgraph des gefärbten Netzes ist in Abbildung 8.25, rechts dargestellt.

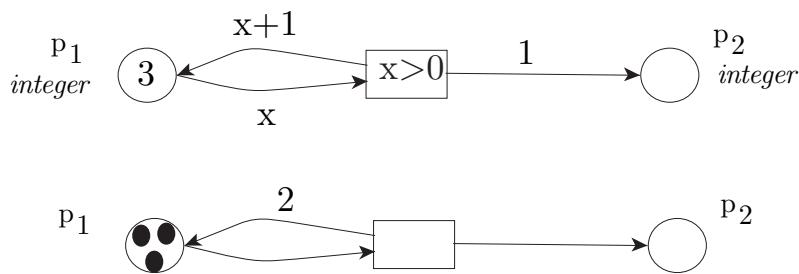


Abbildung 8.28: Gefärbtes Netz und P/T-Netz mit ähnlichem Verhalten

8.4.1 Ausdrucksmächtigkeit

Gefärbte Netze mit der Farbe *integer* besitzen einen engen Bezug zu P/T-Netzen. Vergleiche das gefärbte Netz mit dem P/T-Netz in Abb. 8.28: Beide stellen einen Zähler dar. Der entscheidende Unterschied ist aber, dass im CPN nur eine Marke mit dem Wert $n = 3$ auf der Zählerstelle liegt – nicht aber $n = 3$ Marken. Daher kann der Wert des Zählers als Ganzes erfasst werden, insbesondere auch auf den Wert $n = 0$ getestet werden. Also kann man mit CPN auch Zählerautomaten modellieren (vgl. Abschnitt 7.4), woraus eine Vielzahl von Unentscheidbarkeitsresultaten folgt. Das zentrale Resultat ist das folgende:

Satz 8.21 *Das Erreichbarkeitsproblem ist für CPN unentscheidbar.*

Gefärbte Netze mit ausschließlich endlichen Farbmengen sind dagegen nicht mächtiger als P/T-Netze, da sie immer zu solchen aufgefaltet werden können (vgl. Abbildung 8.24). Eine Übersicht über die Turing-Mächtigkeit bei Petrinetzen gibt die Tabelle 7.1.

8.4.2 Beispiel: Der Datenbank-Manager

Das Beispiel *Datenbank-Manager* [Jen87] ist ein schönes Beispiel eines gefärbten Netzes mit einer größeren Anzahl von gefärbten Marken.

Eine Menge von $n > 0$ Datenbanken soll von n Prozessen, genannt *Datenbank-Manager*, $DBM = \{d_1, d_2, \dots, d_n\}$ so verwaltet werden, dass sie immer den gleichen Inhalt haben. Um dies zu erreichen, sollen die Manager miteinander kommunizieren. Jeder Manager kann seine eigene Datenbank aktualisieren. Dabei muss er an jeden anderen Manager eine Nachricht senden, die diesen über die Aktualisierung informiert. Der Manager muss warten, bis alle anderen diese Nachricht erhalten, die Aktualisierung durchgeführt und eine entsprechende Rückmeldung zurückgesandt haben. Erst dann kehrt der Manager in den Zustand *inactive* zurück.

Dabei sollen weder die Datenbanken noch ihre Aktualisierung dargestellt werden, sondern nur der Nachrichtenaustausch.

Zustände der Manager : *inactive*, *waiting*, *performing*

Nachrichten : $MS = \{(s, r) | s, r \in DBM \wedge s \neq r\}$

Zustände der Nachrichtenkanäle : *unused, sent, received, acknowledged*
wechselseitiger Ausschluss : *exclusion*

Spezifikation für das gefärbte Netz von Abb. 8.29:

$$\text{Farben} : \quad DBM = \{d_1, d_2, \dots, d_n\}$$

$$MS = \{(s, r) | s, r \in DBM \wedge s \neq r\}$$

$$E = \{e\}$$

$$\text{Variablen} : \quad Var = \{e, r, s\}$$

$$dom(e) = E, \quad dom(r) = dom(s) = DBM$$

Funktionen :

$$MINE : DBM \rightarrow Bag(MS) \quad MINE(s) := \sum_{r \neq s} (s, r)$$

$$REC : MS \rightarrow DBM \quad REC((s, r)) := r$$

$$ABS : DBM \rightarrow E \quad ABS(s) := e$$

$$\text{Anfangsmarkierung: } \mathbf{m}_0(p) := \begin{cases} DBM & \text{falls } p = \text{inactive} \\ MS & \text{falls } p = \text{unused} \\ \{e\} & \text{falls } p = \text{exclusion} \\ \emptyset & \text{sonst} \end{cases}$$

Nicht alle Funktionen dieser Spezifikation werden in der graphischen Darstellung von Abb. 8.29 benutzt. Sie sind jedoch nützlich um die Funktion $W_\beta(x, y) \in Bag(cd(x, y))$ (wobei $(x, y) \in F$) aus der Definition 8.17 von gefärbten Netzen zu definieren. Beispielsweise ergibt sich für $(x, y) = (T2, \text{unused})$ und die Belegung $\beta = [s = d_1]$ der Wert $W_\beta(T2, \text{unused}) = MINE(d_1)$.

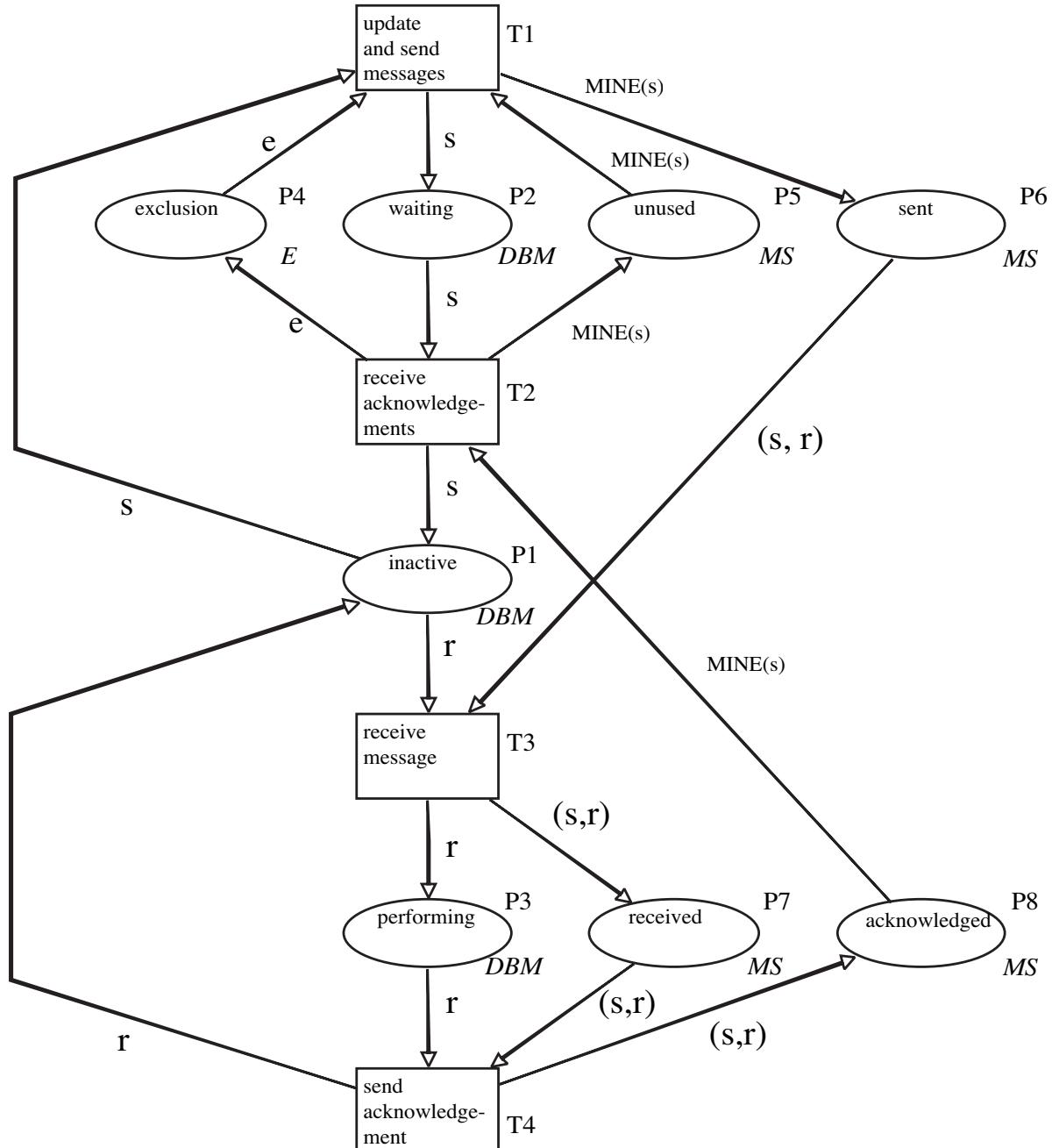


Abbildung 8.29: Datenbank-Manger

8.5 Das RENEW-Werkzeug

Das am Arbeitsbereich TGI in Hamburg entwickelte Werkzeug RENEW [KWD] ermöglicht die graphische Darstellung der hier behandelten Netzmodelle. Die RENEW-Werkzeugeleiste ist in der Abb. 8.30 dargestellt.



Abbildung 8.30: Werkzeugeleiste des RENEW-Werkzeugs

RENEW kann auch für die Ausführung der erzeugten Netze benutzt werden. Als Beschriftungssprache wurden Elemente der Programmiersprache Java [GJS97] gewählt, die in vielen ihrer Eigenschaften Petrinetze sinnvoll ergänzt. Das Werkzeug ist selbst in Java geschrieben und durch Transitionen können Java-Klassen ausgeführt werden. *Ausführung* wird in der Petrinetzliteratur auch als *Simulation* bezeichnet. Dies bedeutet die Simulation des formalen Modells und nicht eines realen Weltausschnittes. Letzteres gilt natürlich auch, wenn das Petrinetz den realen Weltausschnitt hinreichend genau darstellt. Dazu können auch Zeitschränken für das Schalten benutzt werden. Eine Übersicht zu Petrinetz-Tools gibt die Internetseite *The Petri Nets World* [Pet]. Über sie ist auch Information zu Literatur zu Petrinetzen, Forschungsgruppen und -projekte zugänglich.

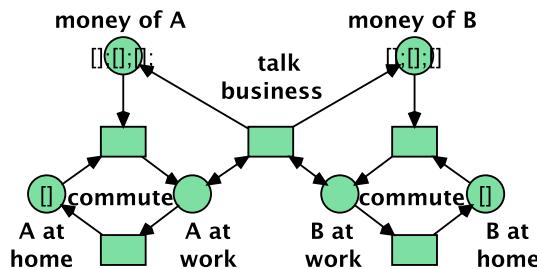


Abbildung 8.31: Das Leben zweier Geschäftsleute

Die Abbildung 8.31 zeigt ein P/T-Netz in RENEW (nach [Kum01]). Seine 3 Marken im Platz **money of A** werden als $[] ; [] ; []$ dargestellt. Dieses Netz kann folgendermaßen interpretiert werden: zwei Geschäftsleute A und B können jeweils von zu Hause (**at home**) zum Arbeitsplatz (**at work**) wechseln. Die Fahrt kostet Geld. Dieses (und mehr) kann jedoch beim gemeinsamen Handel (**talk business**) wieder verdient werden. Abstrakt gesehen handelt sich hier also um zwei Betriebsmittel verbrauchende und erzeugende Funktionseinheiten. Durch Faltung erhält man das gefärbte Netz in Abbildung 8.32. Geschäftsleute werden durch *individuelle Marken* "A" und "B" dargestellt, ebenso

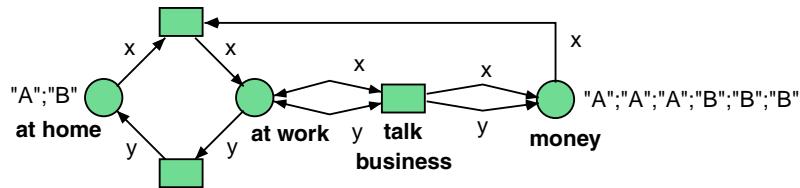


Abbildung 8.32: Faltung von Netz 8.31 zu einem gefärbten Netz

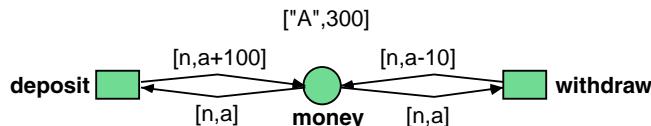


Abbildung 8.33: Ein nicht ganz so einfaches Bankkonto

wie ihr jeweiliges Guthaben im Platz **money**. Wird eine Darstellung mit Bezeichner und Wert gewünscht, so kann man wie in Abb. 8.33 vorgehen. Die Kantenbeschriftungen sind hier 2-Tupel (in JAVA-Notation), deren erste Komponente den Kontoinhaber und deren zweite Komponente den Wert oder den zu verändernden Wert darstellen.

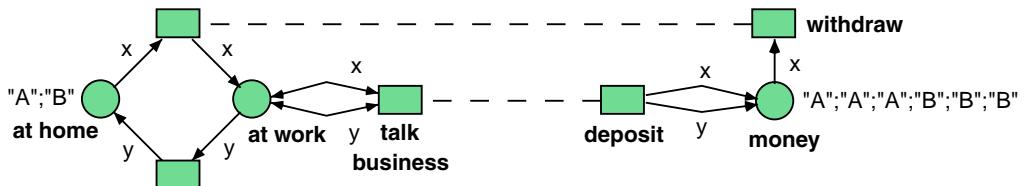


Abbildung 8.34: Personen und Konten werden geteilt

Ein ernst zu nehmender Einwand gegen das gefärbte Netz aus Abb. 8.32 ist, dass die Bewegung von Geld und die Bewegung der Personen zu stark miteinander verwoben sind. Auf den ersten Blick ist es nicht klar, welche Teile des Netzes welchen Aspekt beschreiben.

In Abb. 8.34 bereiten wir eine Teilung des Systems aus Abb. 8.32 vor. Zwei der Transitionen werden doppelt in verschiedenen Teilen des Netzdigramms aufgezeichnet. Um die Schaltsemantik des Netzes korrekt beizubehalten, müssten die Transitionen wieder verschmolzen werden.

Wir gehen einen Schritt weiter und deuten die beabsichtigte Bedeutung des Netzes durch eine textuelle Anschrift an. In Abb. 8.35 bedeutet die Anschrift `this:withdraw(x)`, dass in diesem Netz (Englisch *this net*) eine andere Transition vorhanden sein sollte, die die Auszahlung von Geld vom Konto von **x** übernehmen kann. Wir geben dabei die Variable am Ende der Anschrift an, um klarzumachen, welche Information umhergereicht werden muss.

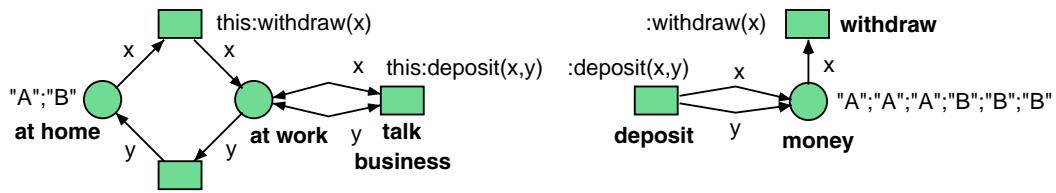


Abbildung 8.35: Textuelle Anschriften kennzeichnen Synchronisation

Solche Anschriften werden als synchrone Kanäle bezeichnet, weil sie die Transitionen zwingen, synchron zu schalten und weil sie den Fluss von Informationen kanalisieren. Die Autoren von [CDH92] haben dieses Konzept für höhere Petrinetze eingeführt. Gegenüber gewöhnlichen Petrinetzen fügen synchrone Kanäle die Fähigkeit hinzu, über gleichzeitige, nicht nur über aufeinanderfolgende oder über nebenläufige Handlungen zu sprechen (Rendez-Vous-Synchronisation).

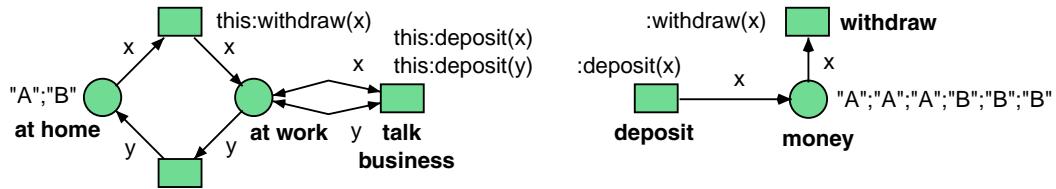


Abbildung 8.36: Wiederverwendung eines Kanals

Weil wir textuelle Anschriften für die Kanäle verwendet haben, können wir mehrere Synchronisationen gleichzeitig anfordern, ohne dass das Diagramm seine Klarheit verliert. In Abb. 8.36 wurde das Netz aus Abb. 8.35 so umstrukturiert, dass von der Transition **talk business** zwei Aufrufe des Kanals **deposit** getätigten werden. Dies veranlasst die Transition **deposit**, zweimal zu schalten, was das gewünschte Verhalten in unserem Beispiel ist.

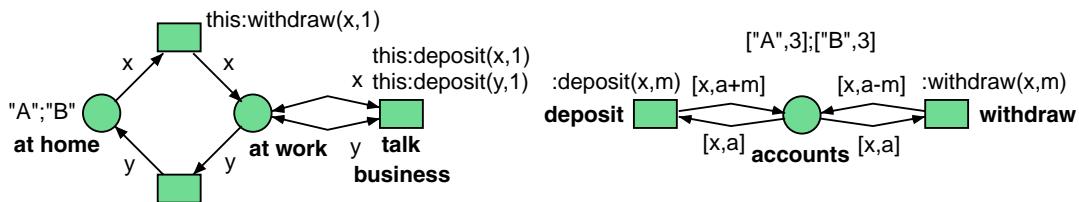


Abbildung 8.37: Buchführung mit Algebra

Jetzt können wir die Lösung aus Abb. 8.36 mit dem gefärbten Netz zur Modellierung eines Bankkontos aus Abb. 8.33 kombinieren und jedes Konto als Wertepaar repräsentieren. Abb. 8.37 zeigt das Resultat. Beachtenswert ist, wie die Anzahl der Geldeinheiten, die ein- oder auszuzahlen ist, als zweiter Parameter über den Kanal übergeben wird.

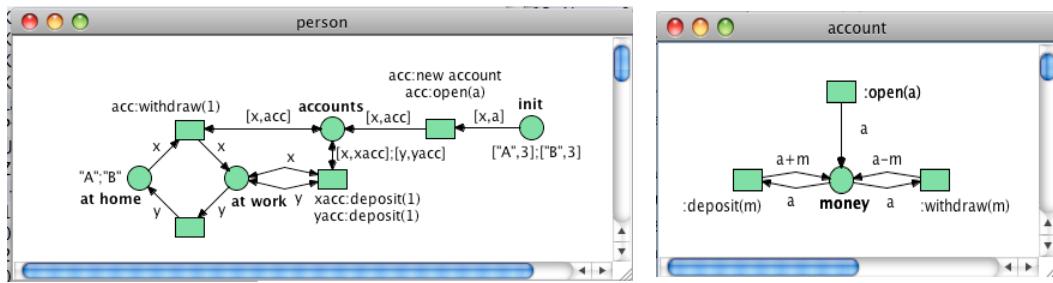


Abbildung 8.38: RENEW: Buchführung mit Algebra in zwei Fenstern

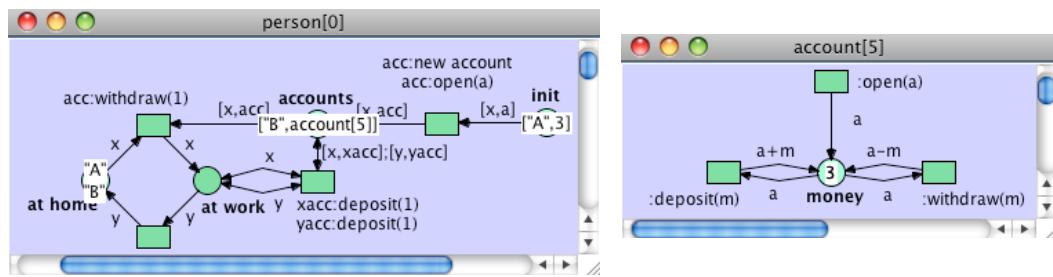


Abbildung 8.39: RENEW (Simulationsmodus): Buchführung mit Algebra in zwei Fenstern

Üblicherweise werden die beiden Netze in zwei verschiedenen Fenstern wie in Abbildung 8.38 dargestellt. Die Abbildung 8.39 zeigt das Netz `person` nach einem Simulationsschritt. Beim Schalten der Transition rechts oben wurde mit der Bindung $[x = "B", a = 3]$ ein Exemplar `account[5]` des Musters `account` erzeugt und an die Variable `acc` gebunden. Dadurch wird die Bindung zu $[x = "B", a = 3, acc = account[5]]$ erweitert. Im Platz `accounts` verfügt nun der Geschäftsmann "B" über eine Referenz auf sein Konto `account[5]` mit Inhalt 3. Beim nochmaligen Schalten dieser Transition würde entsprechendes für den Geschäftsmann "A" gelten, natürlich mit der Referenz auf ein eigenes Exemplar von `account`. Diese Fähigkeit von RENEW Exemplare von Mustern zu bilden und die Referenzen wie Marken zu behandeln ist eine über übliche gefärbte Netze hinausführende Eigenschaft, die aber nicht Gegenstand dieser Vorlesung ist.

8.6 Das Alternierbitprotokoll

Das *Alternierbitprotokoll* steht für die gesamte Klasse der Kommunikationsprotokolle. Diese Klasse ist stark actionsorientiert. Das entwickelte Netz stellt das Bit und die Daten als Marken eines gefärbten Netzes dar.

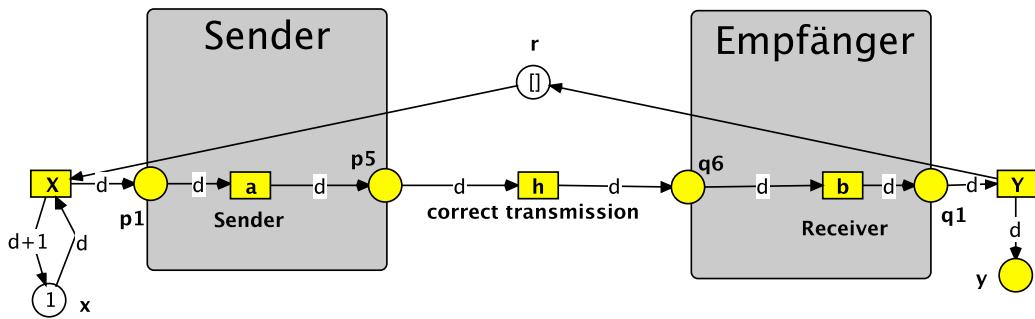


Abbildung 8.40: Spezifikation abp-1 des Alternierbitprotokolls

In diesem Beispiel wird das *Alternierbitprotokoll* als gefärbtes Netz modelliert. Für die Übermittlung einer Nachricht zwischen zwei Arbeitsrechnern (hosts) stehe ein Kanal zur Verfügung, auf dem in beiden Richtungen, aber nur in einer Richtung zur Zeit, eine Nachricht übermittelt werden kann (Halbduplex-Kanal). Bei der Übermittlung kann die Nachricht gestört werden. Durch redundante Kodierung wird aber jeder Fehler erkannt und angezeigt. Das Alternierbitprotokoll soll diese Fehler durch redundante Übermittlung verdecken und so den Benutzern einen fehlerfreien Kanal zur Verfügung stellen. Dazu muss natürlich vorausgesetzt werden, dass der Kanal nicht dauerhaft gestört bleibt, sondern immer wieder einmal ein Datum korrekt übermittelt.

Im folgenden wird das Protokoll schrittweise entwickelt. Zur Einführung in die Systemumgebung ist in Abbildung 8.40 die ungestörte Übermittlung von Daten von einem „Sender“ zu einem „Empfänger“ dargestellt. Die Transition **X** modelliert eine Funktionseinheit (z.B. eine Person, ein Prozessor), die Daten d zur Übermittlung an den Sender übergibt. Um dies in RENEW ausführbar zu machen, sind dies hier die Werte 1, 2, 3, Das Protokoll soll jedoch für beliebige Daten korrekt arbeiten, d.h. das Protokoll darf nicht auf den Inhalt der Daten zugreifen, um z.B. die Folge als Sequenznummern zu benutzen. Einige oder alle Daten können auch den gleichen Wert haben. Diese Daten werden über die Transitionen **a**, **h** und **b** korrekt übermittelt, damit sie eine Funktioneinheit **Y** (z.B. eine Person, ein Prozessor) empfangen kann. Der mit einer anonymen („schwarzen“) Marke $[]$ markierte Platz **r** dient nur dazu, dass die Plätze der mittleren Reihe maximal eine Marke enthalten. Dadurch ist gewährleistet, dass die Daten im Ausgangskanal **q1** in der gleichen Reihenfolge erscheinen wie im Eingangskanal **p1**. Darüberhinaus wird für das Protokoll gefordert, dass nicht eher ein $n + 1$ -ter Wert in **p1** erscheint als in **q1** der Wert n beobachtet wurde. Andere Protokolle erlauben hier eine beschränkte Überlappung. Dies ist eine Weiterentwicklung, die zur Vereinfachung hier

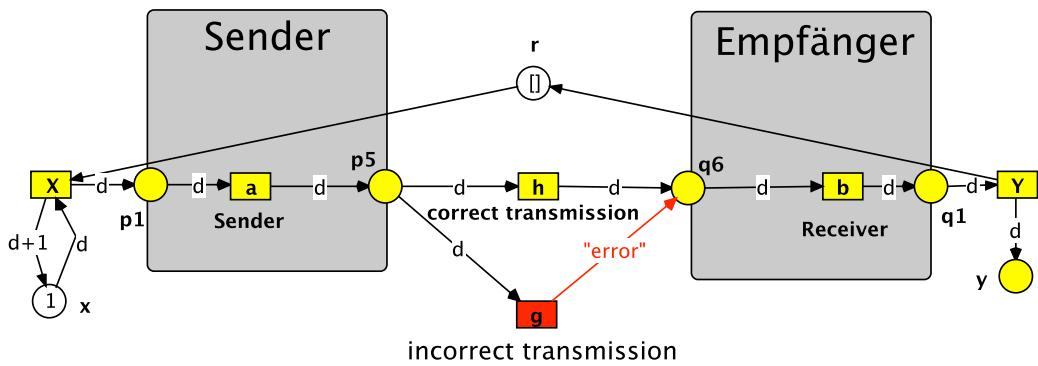


Abbildung 8.41: Übermittlung mit Fehlern: Netz abp-2

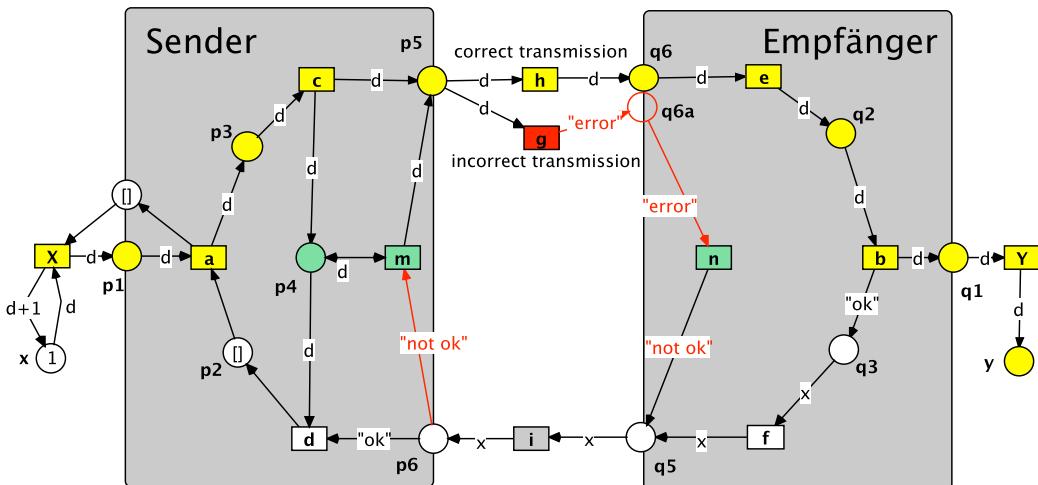


Abbildung 8.42: Übermittlung mit Quittung: Netz abp-3

nicht betrachtet wird. Das Netz abp-1 kann also als Spezifikation der Aufgabenstellung angesehen werden.

Im Netz abp-2 (Abb. 8.41) kann der Kanal Daten fehlerhaft übermitteln. Dies wird durch das alternative Schalten der Transition g modelliert, die die Fehlermeldung "error" erzeugt. Dies muss in Anwendungen durch unterliegende Protokollsichten implementiert werden, bzw. durch ein *timeout* im Fall von Datenverlust. Das Netz abp-2 hat ein von der Spezifikation abp-1 verschiedenes Verhalten.

Ein der Spezifikation entsprechendes Verhalten wird durch das Netz abp-3 durch die Rücksendung einer positiven ("ok") oder negativen ("not ok") Quittung erreicht. Bei einer negativen Quittung wird das im Platz p4 gespeicherte Datum solange wieder versandt, bis eine positive Quittung erfolgt. Das Netz erfüllt jedoch nicht die Spezifikation, wenn auch die Rücksendung über die Transition i fehlerhaft ist.

Zur Vorbereitung der endgltigen Lsung wird mit abp-4 ein zu abp-3 verhaltensquivalentes Protokoll eingefhrt. Dem Datum d wird durch die Transition a ein Bit x beigeftigt,

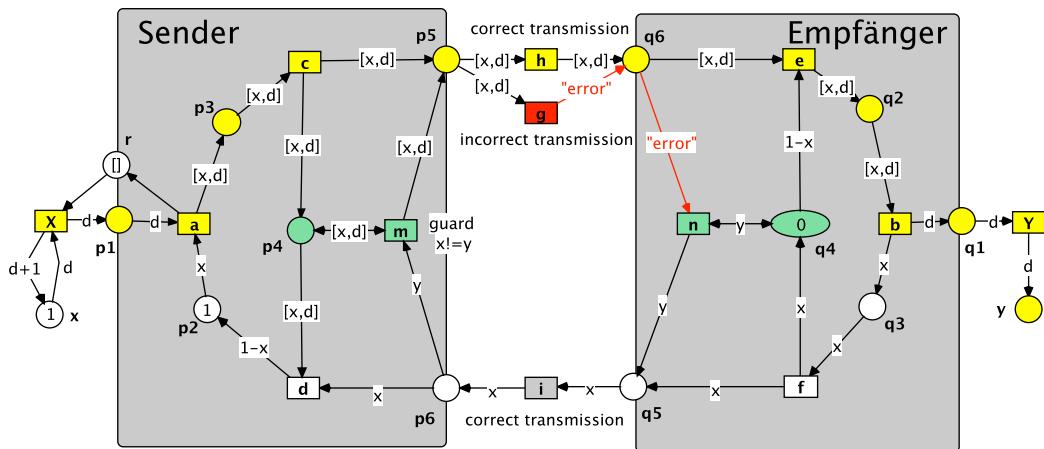


Abbildung 8.43: Übermittlung mit Quittung durch Alternierbit: Netz abp-4

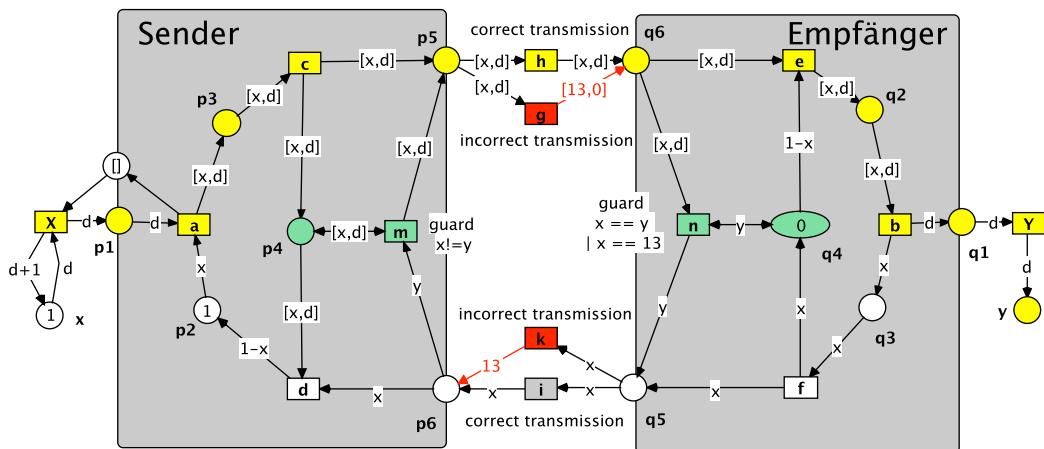


Abbildung 8.44: Das Alternierbitprotokoll abp-5

das anfangs den Wert 1 hat. Bei fehlerfreier Übermittlung schaltet die Transition e , da q_4 den komplementären Wert $1 - x$ (anfangs 0) enthält. Danach wird durch die Transition b das Bit wieder gelöscht und das Datum d wie vorher abgeliefert. Die Quittung erfolgt durch das (nicht geänderte) Bit x . Im Fehlerfall wird jedoch durch die Transition n mit dem komplementären Bit (anfangs 0) quittiert, wodurch die erneute Sendung eingeleitet wird. Dies erkennt der Sender dadurch, dass das Quittungs-Bit y von dem gespeicherten Bit x verschieden ist (guard $x!=y$). Um die nächste Datenübermittlung einzuleiten, wird das Ausgangsbit komplementiert nach p_2 gelegt. Der Vorgang wiederholt sich mit dem jeweils komplementären Bitwert.

Nun ist es nur noch ein kleiner Schritt zur endgültigen Lösung. Im Netzwerk abp-5 (Abb. 8.44) ist nun auch im Quittungs-Kanal eine Störung möglich. Um den Integer-Test guard $x!=y$ beizubehalten zu können, wurde statt der Fehlermeldung "error" die (Unglücks-)Zahl 13 gewählt, d.h. auch bei einem Fehler in diesem Kanal wird das Datum d erneut

verschickt. Nun tritt aber folgendes Problem auf. Falls das Datum vorher störungsfrei übermittelt wurde, würde in diesem Fall eine verdoppelte Ankunft beim Empfänger erfolgen. Dies kann der Empfänger jedoch daran erkennen, dass das Bit nicht gewechselt hat. Die verdoppelte Sendung wird durch den Guard $x==y \mid x==13$ (d.h. $x = y \vee x = 13$) in der Transition **n** gelöscht.

Als Besonderheit kommt dieses Protokoll zur Quittierung mit nur einem Bit **x** aus ([BS69]) - daher der Name *Alternierbitprotokoll*. Ein Beweis dafür, dass die Lösung korrekt ist, kann z.B. dadurch erfolgen, dass das Netz abp-5 als verhaltensäquivalent zu seiner Spezifikation abp-1 bewiesen wird. Verhaltensäquivalenz kann z.B. durch Bisimulation formalisiert werden. Dies wird im Rahmen der Prozessalgebra durchgeführt. Eine alternative Verifikationsmethode ist das *Model-Checking*. Hierbei wird die Spezifikation anhand des Erreichbarkeitsgraphen untersucht, der im vorliegenden Fall etwa 70 Zustände hat. Es wurden auf diese Weise bereits Systeme mit 10^{50} Zuständen verifiziert.

9 Prozessalgebra

Die Prozessalgebra wurde aus der Automatentheorie entwickelt, um nebenläufige und reaktive Prozesse und Systeme beschreiben, modellieren und verifizieren zu können. Dabei wurden wie bei endlichen Automaten Zustände festgelegt und für Aktionen oder Folgen von Aktionen spezifiziert, welches der Nachfolgezustand ist. Die *elementare Prozessalgebra* entspricht der Modellierung eines einzigen Automaten. Im Unterschied zur traditionellen Automatentheorie wird aber eine algebraische Behandlung (vergleichbar zu den regulären Ausdrücken) und der Aspekt der Beobachtbarkeit und Verhaltensäquivalenz reaktiver Systeme betont. Nebenläufige und kommunizierende Systeme werden durch mehrere Automaten beschrieben, die mittels Rendezvous-Synchronisation gekoppelt werden.

Die Darstellung in diesem Kapitel stützt sich in erster Linie auf [Fok99] und teilweise auf [Mil99]. Weitere einschlägige Literaturquellen sind: [BW90], [BV95], [Mil89] und [Bae95].

9.1 Prozess-Algebra

In diesem Abschnitt werden Prozessterme und ihre Bedeutung definiert. Letzteres erfolgt durch spezielle Transitionssysteme wie in den folgenden Beispielen.

9.1.1 Prozessterme

Die syntaktische Darstellung von Prozessen geschieht durch *Prozess-Terme*. Die im folgenden definierte Menge *BPA* ist die Grundmenge des auf Seite 194 eingeführten gleichnamigen Kalküls (**b**asic **p**rocess **a**lgebra)¹.

Definition 9.1 Die Menge der elementaren Prozess-Terme bzw. Prozess-Ausdrücke BPA wird aus atomaren Aktionen $a \in A$ sowie der Auswahl und Hintereinanderausführung gebildet:

- Atomare Aktion: Jedes $a \in A$ ist in BPA.
- Auswahl: Für alle $t_1, t_2 \in BPA$ ist $(t_1 + t_2) \in BPA$
- Sequenz: Für alle $t_1, t_2 \in BPA$ ist $(t_1 \cdot t_2) \in BPA$
- Nur nach diesen Regeln gebildete Terme liegen in BPA.

Bindungsstärke: Der Sequenzoperator \cdot bindet stärker als der Auswahloperator $+$. Durch diese Konvention können Klammern gespart werden. Der Sequenzoperator kann weggelassen werden, d.h. wir notieren kurz $(t_1 t_2)$ statt $(t_1 \cdot t_2)$.

¹Genau genommen muss also die Menge BPA vom Kalkül BPA unterschieden werden.

9.1.2 Prozessgraphen

Wir verbinden im folgenden mit jedem Term einen Prozess. Der Auswahl-Term ($t_1 + t_2$) bedeutet dann: Es wird entweder t_1 oder t_2 ausgeführt. Die Sequenz ($t_1 \cdot t_2$) bedeutet: Nach der ordnungsgemäßen Termination von t_1 wird t_2 ausgeführt.

Beispiel 9.2 Der BPA-Term $((a + b) \cdot c) \cdot d$ (oder kürzer $(a + b)cd$) repräsentiert das Verhalten des linken Transitionssystems von Abbildung 9.1.

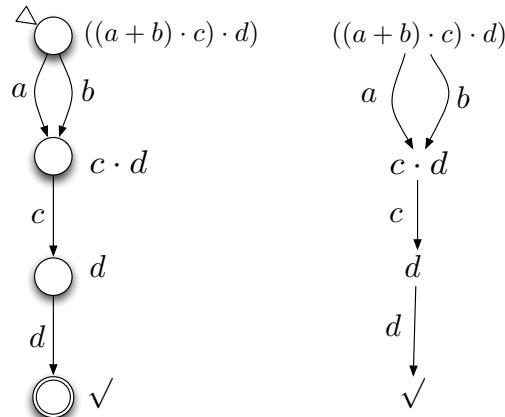


Abbildung 9.1: Prozessgraph von $((a + b)c)d$ als Transitionssystem

Als *Prozessgraphen* bezeichnen wir ein solches Transitionssystem, dessen Zustände Prozessterme sind. Ein solcher Prozessterm repräsentiert das Verhalten des Transitionssystems mit ihm als Anfangszustand. Die Zustände eines Prozessgraphen beschreiben also das noch bevorstehende (Rest-)Verhalten, nachdem dieser Zustand erreicht wurde. Der Prozessgraph definiert eine *operationale Semantik* der Prozessalgebra.

Ein Prozessgraph wird formal über einen Kalkül definiert, wobei (A_0) das Axiom ist. Die Regeln definieren Transitionsübergänge zwischen Zuständen, die entweder Prozessterme oder das Symbol \checkmark sind. Das Symbol \checkmark kennzeichnet die korrekte Termination.

Definition 9.3 Sei t ein Prozessterm. Dann heißt das Transitionssystem $TS(t) = (S, A, tr, s_0, S^F)$ Prozessgraph von t , wenn gilt:

- $S \subseteq BPA$ sind die mit dem Transitionsskalkül aus t erreichbaren Prozessterme.
- A ist die Menge der atomaren Aktionen von t .
- tr die durch den Transitionsskalkül eingeführte Transitionsrelation.
- $s_0 = t$ ist der einzige Anfangszustand.
- $S^F = \{\checkmark\}$ ist der einzige Endzustand.

Abbildung 9.1 zeigt links einen Prozessgraphen als Transitionssystem mit Anfangs- und Endzustand. Prozessgraphen werden in den folgenden Beispielen wie in Abbildung 9.1

$$\begin{array}{c}
 \overline{v \xrightarrow{v} \checkmark} \quad (A_0) \\
 \frac{x \xrightarrow{v} \checkmark}{x + y \xrightarrow{v} \checkmark} \quad (T_{+R}^\checkmark) \quad \frac{x \xrightarrow{v} x'}{x + y \xrightarrow{v} x'} \quad (T_{+R}) \\
 \\
 \frac{y \xrightarrow{v} \checkmark}{x + y \xrightarrow{v} \checkmark} \quad (T_{+L}^\checkmark) \quad \frac{y \xrightarrow{v} y'}{x + y \xrightarrow{v} y'} \quad (T_{+L}) \\
 \\
 \frac{x \xrightarrow{v} \checkmark}{x \cdot y \xrightarrow{v} y} \quad (T^\checkmark) \quad \frac{x \xrightarrow{v} x'}{x \cdot y \xrightarrow{v} x' \cdot y} \quad (T.)
 \end{array}$$

 Abbildung 9.2: BPA-Transitionsregeln ($v \in A, x, y, x'y' \in BPA$)

rechts dargestellt. Um die Graphiken übersichtlicher zu gestalten, wird oft der einzige Endzustand \checkmark mehrfach gezeichnet. (Mehrfach gezeichnete Knoten bezeichnen aber immer nur ein einziges Objekt.)

Beispiel 9.4 Ableitung von $((a + b) \cdot c) \cdot d \xrightarrow{b} c \cdot d$ aus den Transitionsregeln:

$$\begin{array}{c}
 b \xrightarrow{b} \checkmark \qquad \overline{v \xrightarrow{v} \checkmark} \\
 \hline
 a + b \xrightarrow{b} \checkmark \qquad \frac{y \xrightarrow{v} \checkmark}{x + y \xrightarrow{v} \checkmark} \\
 \hline
 (a + b) \cdot c \xrightarrow{b} c \qquad \frac{x \xrightarrow{v} \checkmark}{x \cdot y \xrightarrow{v} y} \\
 \hline
 ((a + b) \cdot c) \cdot d \xrightarrow{b} c \cdot d \quad \frac{x \xrightarrow{v} x'}{x \cdot y \xrightarrow{v} x' \cdot y}
 \end{array}$$

(links: die Ableitung im Kalkül, rechts: die benutzten Regeln)

Die Operatoren \cdot und $+$ werden in der offensichtlichen Weise auch auf Prozessgraphen erweitert. Wir betrachten also die alternative Komposition von Prozessgraphen $TS(s) + TS(t)$ und die sequentielle Komposition $TS(s) \cdot TS(t)$.

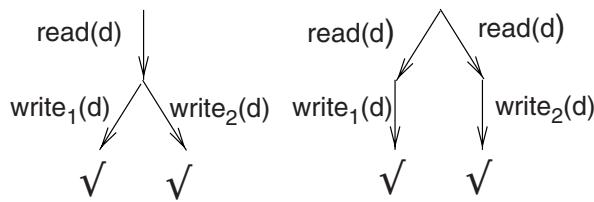
Aufgabe 9.5 Leiten Sie die Prozessgraphen der drei folgenden Prozessausdrücke mit dem Kalkül ab.

- a) $((a + b)c + ac)d$
- b) $(a(b + b))(c + c)$
- c) $(b + a)(cd)$

9.1.3 Bisimulation und Äquivalenz

Um das Verhalten eines Systems mit dem Verhalten eines anderen Systems oder einer Spezifikation zu vergleichen, wurde der Begriff der wechselseitigen Simulation oder Bisimulation eingeführt.

Beispiel 9.6 Der linke Prozess liest d . Dann wird entschieden, ob d auf Platte 1 oder Platte 2 geschrieben wird. In dem anderen Prozess wird die Entscheidung vor dem Lesen getroffen.



Beide Prozesse haben die gleichen Schaltfolgen:

$$read(d) write_1(d) \quad \text{und} \quad read(d) write_2(d)$$

Die Prozesse sind daher „schaltfolgenäquivalent“ (trace equivalent).

Diese Art der Äquivalenz ist jedoch häufig nicht angemessen, z.B. wenn die Platte 1 ausfällt. Dann würde der erste Prozess d bei jedem Ablauf auf Platte 2 schreiben - im Gegensatz zum anderen Prozess, der in eine Verklemmung geraten kann.

Dies ist die Motivation, für einen auf „Bisimulation“ beruhenden Äquivalenzbegriff.

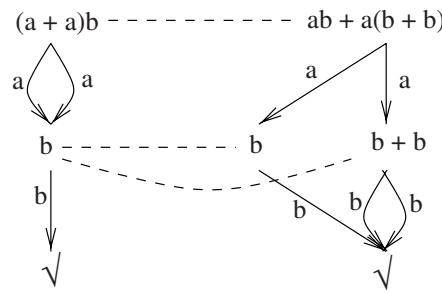
Definition 9.7 Eine Bisimulation ist eine binäre Relation \mathcal{B} auf BPA (d.h. $\mathcal{B} \subseteq BPA \times BPA$) mit folgenden Eigenschaften:

1. falls $p \mathcal{B} q$ und $p \xrightarrow{a} p'$, dann $q \xrightarrow{a} q'$ mit $p' \mathcal{B} q'$
2. falls $p \mathcal{B} q$ und $q \xrightarrow{a} q'$, dann $p \xrightarrow{a} p'$ mit $p' \mathcal{B} q'$
3. falls $p \mathcal{B} q$ und $p \xrightarrow{a} \checkmark$, dann $q \xrightarrow{a} \checkmark$
4. falls $p \mathcal{B} q$ und $q \xrightarrow{a} \checkmark$, dann $p \xrightarrow{a} \checkmark$

Zwei Prozesse p und q heißen bisimilar (Bezeichnung: $p \sqsubseteq q$), falls es eine Bisimulation \mathcal{B} mit $p \mathcal{B} q$ gibt.

Abweichend zu der Definition 2.4 wird in Definition 9.7 nicht der Anfangszustand in Relation gesetzt, so dass $\mathcal{B} = \emptyset$ stets eine mögliche Bisimulation ist. Da aber auch hier gilt, dass $(\mathcal{B} \cup \mathcal{B}')$ zweier Bisimulationen \mathcal{B} und \mathcal{B}' eine ist, können wir stets die maximale Bisimulation $\bigcup_{\mathcal{B} \text{ ist Bisimulation}} \mathcal{B}$ verwenden. Da dann zumindest $\sqrt{\mathcal{B}} \checkmark$ gilt, können wir den Fall $\mathcal{B} = \emptyset$ ausschliessen.

Beispiel 9.8 Es gilt: $(a + a)b \sqsubseteq ab + a(b + b)$



Diese Bisimulation \mathcal{B} wird durch folgende Paare definiert: $(a+a)b \mathcal{B} ab + a(b+b)$ sowie $b \mathcal{B} b$ und $b \mathcal{B} b+b$. In der Prozessalgebra ist es üblich, das immer geltende Paar $\sqrt{\mathcal{B}} \sqrt{\mathcal{B}}$ nicht explizit zu nennen.

Lemma 9.9 *Die Bisimulation ist eine Äquivalenzrelation.*

Aufgabe 9.10 Beweisen Sie die folgenden drei Bisimulationspaare mit Hilfe der Definition:

- a) $((a+a)(b+b))(c+c) \mathcal{B} a(bc)$
- b) $(a+a)(bc) + (ab)(c+c) \mathcal{B} (a(b+b))(c+c)$
- c) $((a+b)c + ac)d \mathcal{B} (b+a)(cd)$

9.1.4 Bisimulation als Kongruenz

Dass die Bisimulation eine Äquivalenz ist, reicht uns aber noch nicht, wenn wir mit BPA kompositionale Argumentationen wollen: Angenommen wir haben ein System, dass wir durch die Auswahl $(s+t)$ beschreiben. Ersetzen wir das Subsystem s durch ein System s' , von dem wir wissen, dass es sich genauso wie s verhält (d.h. es gilt $s \leftrightarrow s'$), dann soll sich am Verhalten des Gesamtsystems auch nichts verändern:

$$s \leftrightarrow s' \text{ impliziert } (s+t) \leftrightarrow (s'+t)$$

Dieses Konzept ist in der Algebra unter dem Begriff der *Kongruenz* bekannt.

Definition 9.11 Eine Äquivalenzrelation \simeq auf der Menge X heißt *Kongruenz*, wenn für alle $x, y, x', y' \in X$ und alle Operatoren f gilt:

$$\text{Wenn } x \simeq x' \text{ und } y \simeq y', \text{ dann auch } f(x, y) \simeq f(x', y').$$

Ganz allgemein wollen wir also, dass die Bisimulation eine Kongruenz bzgl. der BPA-Operatoren $+$ und \cdot sein soll:

$$\text{Wenn } s \leftrightarrow s' \text{ und } t \leftrightarrow t', \text{ dann auch } (s+t) \leftrightarrow (s'+t') \text{ und } (s \cdot t) \leftrightarrow (s' \cdot t').$$

Theorem 9.12 Bisimulation ist eine Kongruenz auf BPA.

Beweis: (Idee) Die Kongruenzeigenschaft folgt daraus, dass die Transitionsregeln in einer bestimmten Normalform (Panth-Form) sind. \square

9.1.5 Der BPA-Kalkül

Mit obenstehender Definition ist es aufwendig zu überprüfen, ob zwei elementare Prozesterme bisimilar sind: es müssen zunächst die Prozessgraphen (die i.a. sehr groß werden) und dann zwischen ihnen die Bisimulationsrelation konstruiert werden.

Es wird daher jetzt ein Gleichungskalkül (**b**asic **p**rocess **a**lgebra) für eine Gleichheitsrelation zwischen elementaren Prozesstermen eingeführt, der diese Aufgabe durch die Konstruktion von Ableitungen lösen soll.

In diesem Kalkül lässt sich fast wie gewohnt rechnen. Es ist allerdings zu beachten, dass die geltenden Axiome anders als die bekannter Algebren (wie z.B. Gruppen, Körper, Mengenalgebren) sind.

Axiome des BPA-Kalküls Für $x, y, z \in BPA$ gelte:

$$\begin{array}{ll} A1 & x + y = y + x \\ A2 & (x + y) + z = x + (y + z) \\ A3 & x + x = x \\ A4 & (x + y) \cdot z = x \cdot z + y \cdot z \\ A5 & (x \cdot y) \cdot z = x \cdot (y \cdot z) \end{array}$$

Substitution Eine *Substitution* σ ist eine Abbildung der Variablen in Terme. Sie wird induktiv auf Terme erweitert, indem alle Variablen v des Terms durch $\sigma(v)$ ersetzt werden. Beispiel: für $\sigma(u) := (u + v) \cdot w, \sigma(v) := u + u, \sigma(w) := w$ gilt dann $\sigma(u \cdot v + u) = ((u + v) \cdot w) \cdot (u + u) + (u + v) \cdot w$.

Schlussregeln des BPA-Kalküls

- (SUBSTITUTION) Für $s = t$ und eine Substitution σ gelte $\sigma(s) = \sigma(t)$.
- (ÄQUIVALENZ)
 - $t = t$ für alle $t \in BPA$
 - falls $s = t$, dann $t = s$
 - falls $s = t$ und $t = u$, dann $s = u$
- (KONTEXT)

Falls $s = s'$ und $t = t'$, dann $s + t = s' + t'$ und $s \cdot t = s' \cdot t'$.

Um die Relation *bisimilar* mittels des BPA-Kalküls zu berechnen, muss natürlich bewiesen werden, dass zwei Terme genau dann bisimilar sind, wenn sie äquivalent sind. Traditionell wird diese Eigenschaft in zwei Teilen als *Korrektheit* und *Vollständigkeit* des BPA-Kalküls formuliert und bewiesen.

Definition 9.13 (Korrektheit (soundness), Vollständigkeit (completeness))
Das BPA-Kalkül heißt korrekt (sound), wenn für alle Terme $s, t \in BPA$ aus $s = t$ auch $s \sqsubseteq t$ folgt.

Das BPA-Kalkül heißt vollständig (complete), wenn für alle Terme $s, t \in BPA$ aus $s \sqsubseteq t$ auch $s = t$ folgt.

Satz 9.14 Der BPA-Kalkül ist korrekt.

Beweis: Es wird hier nur die Beweisstruktur dargestellt. Wie fast immer bei Beweisen für die Korrektheit eines Kalküls ist zu zeigen:

- 1.) Die Axiome sind korrekt.
- 2.) Die Regeln überführen korrekte Terme in korrekte Terme.

zu 1.): Erfolgt mit 2 a): Substitution

zu 2.):

- a) Substitution: Es ist $\sigma(s) \leftrightharpoons \sigma(t)$ für jedes Axiom $s = t$ zu beweisen, wobei σ eine Substitution ist, die alle Variablen in s und t auf elementare Prozessterme abbildet. Dabei ist auf die Definition der Bisimulation zurückzugreifen. Dies wird hier nicht ausgeführt.

Informell steht dahinter in Bezug auf die Axiome A1 … A5 folgende Argumentation:

- A1: Die Terme $s + t$ und $t + s$ stellen beide eine Auswahl zwischen s und t dar.
 - A2: Die Terme $(s + t) + u$ und $s + (t + u)$ stellen beide eine Auswahl zwischen s , t und u dar.
 - A3: Eine Auswahl zwischen t und t ist eine Wahl für t .
 - A4: Die Terme $(s + t) \cdot u$ und $s \cdot u + t \cdot u$ stellen beide eine Auswahl zwischen s und t dar, worauf u ausgeführt wird.
 - A5: Die Terme $(s \cdot t) \cdot u$ und $s \cdot (t \cdot u)$ stellen beide die Aktion s dar, gefolgt von t und dann von u .
- b) Die Äquivalenzregeln gelten, da Bisimulation eine Äquivalenz ist.
 - c) Die Kontextregel gilt, da Bisimulation-Relation eine Kongruenz ist.

□

Aufgabe 9.15 Motivieren Sie, dass folgendes Distributivgesetz nicht gilt: $x \cdot (y + z) = x \cdot y + x \cdot z$.

Hinweis: Setzen Sie $x = \text{read}(d)$ usw. in obigem Beispiel.

Satz 9.16 Der BPA-Kalkül ist vollständig.

Beweis: Zu beweisen ist also, dass aus $s \leftrightharpoons t$ auch $s = t$ folgt. Zunächst wird der Beweis für die einfachere Relation $=_{\text{AC}}$ bewiesen, d.h. $s \leftrightharpoons t \Rightarrow s =_{\text{AC}} t$. Daraus wird dann die Behauptung des Satzes abgeleitet.

Per definitionem gelte $s =_{\text{AC}} t$, wenn der Ausdruck nur mittels der Axiome A1 (Kommutativität) und A2 (Assoziativität) abgeleitet werden kann. Jede Äquivalenzklasse bezüglich dieser Relation wird durch einen Ausdruck aus „Summanden“ der Form $s_1 + \dots + s_k$ dargestellt, wobei s_i entweder eine atomare Aktion a ist oder die Form $t_1 \cdot t_2$ hat.

Die übrigen Axiome werden in Ersetzungsregeln mit der Richtung „links nach rechts“ umgeformt:

$$\begin{array}{lll} \text{R1} & x + y & =_{\text{AC}} y + x \\ \text{R2} & (x + y) + z & =_{\text{AC}} x + (y + z) \\ \text{R3} & x + x & \rightarrow x \\ \text{R4} & (x + y) \cdot z & \rightarrow x \cdot z + y \cdot z \\ \text{R5} & (x \cdot y) \cdot z & \rightarrow x \cdot (y \cdot z) \end{array}$$

Auf diese Weise erhalten wir ein Ersetzungskalkül, bei dem zwischen den Regelanwendungen Umformungen bzgl. der Relation $=_{\text{AC}}$ angewendet werden können (siehe Algorithmus 9.1).

Wir zeigen im folgenden die Existenz von Normalformen:

Wendet man die Regeln dieses Kalküls immer wieder einen Prozessterm $s \in BPA$ an, so gelangt man nach einer endlichen Zahl von Schritten zu einem Prozessterm $t \in BPA$, der nicht weiter reduziert werden kann.

Man kann sogar zeigen, dass die Normalform trotz des Nichdeterminismus eindeutig bestimmt ist. Dies folgt aus der Eigenschaft des Ersetzungskalküls *konfluent* zu sein – was aber hier nicht bewiesen wird.

Dass der Ersetzungskalkül *terminierend* ist weist man mit einer *Gewichtsfunktion*

$$gew : BPA \mapsto \mathbb{N}$$

nach, die Eigenschaft hat, dass jede Reduktion das Gewicht echt verringert.

Wir definiert $gew : BPA \mapsto \mathbb{N}$ folgendermaßen ($v \in A, s, t \in BPA$):

$$\begin{aligned} gew(v) &:= 2 \\ gew(s + t) &:= gew(s) + gew(t) \\ gew(s \cdot t) &:= gew(s)^2 \cdot gew(t) \end{aligned}$$

Da $gew(s_1) > gew(s_2) > \dots > gew(s_q) > \dots$ für jede Reduktion $s_1, s_2, \dots, s_q, \dots$ gilt, kann es keine unendlichen Ableitungen geben.

Terme in Normalform haben die Struktur $t_1 + \dots + t_k$, wobei jedes t_i eine atomare Aktion $a \in A$ ist oder die Form $a \cdot s$ ($a \in A$, s in Normalform) hat. Durch Induktion über ihre Länge beweist man für Normalformen n und n' :

$$n \xrightarrow{\underline{\underline{\cdot}}} n' \Rightarrow n =_{\text{AC}} n'$$

- Falls n einen Summanden der Form a enthält, dann gilt $n \xrightarrow{a} \checkmark$ und wegen $n \xrightarrow{\underline{\underline{\cdot}}} n'$ auch $n' \xrightarrow{a} \checkmark$. Also ist a auch in n' als Summand enthalten.
- Falls n einen Summanden der Form $a \cdot s$ enthält, dann gilt $n \xrightarrow{a} s$ und wegen $n \xrightarrow{\underline{\underline{\cdot}}} n'$ auch $n' \xrightarrow{a} t$ mit $s \xrightarrow{\underline{\underline{\cdot}}} t$. Also ist $a \cdot t$ in n' als Summand enthalten. Da s und t in Normalform, aber kleiner als n und n' sind, folgt durch Induktion $s =_{\text{AC}} t$.

Da somit n und n' dieselben Summanden haben, gilt $n =_{\text{AC}} n'$.

Um nun den Beweis von $s \xrightarrow{\underline{\underline{\cdot}}} t \Rightarrow s = t$ zu führen, sei $s \xrightarrow{\underline{\underline{\cdot}}} t$ angenommen.

s und t können durch das Ersetzungssystem zu Normalformen n und n' reduziert werden. Dies könnte auch durch den Kalkül geschehen, d.h. es gilt: $s = n$ und $t = n'$.

Aus der Korrektheit des Kalküls folgt $s \xrightarrow{\cdot} n$ und $t \xrightarrow{\cdot} n'$, also insgesamt $n \xrightarrow{\cdot} n'$.

Für solche Normalformen wurde aber oben gezeigt: $n =_{AC} n'$.

Damit ergibt sich insgesamt: $s = n =_{AC} n' = t$, d.h. $s = t$. \square

Anmerkung: Der Begriff *Normalform* beinhaltet, dass verschiedene Ableitungen auf die gleiche Form (modulo $=_{AC}$) führen. Dies ist insofern bemerkenswert, da die Konstruktion der Normalform i.a. nicht deterministisch abläuft.

Beispielsweise erlaubt der Term $t = ((a + a) + (b + b))c$ folgende Umformungen:

$$((a + a) + (b + b))c \xrightarrow{R3} (a + (b + b))c \xrightarrow{R3} (a + b)c \xrightarrow{R4} ac + bc$$

und

$$\begin{aligned} ((a + a) + (b + b))c &\xrightarrow{R4} (a + a)c + (b + b)c \\ &\xrightarrow{R4} (ac + ac) + (b + b)c \\ &\xrightarrow{R4} (ac + ac) + (bc + bc) \\ &\xrightarrow{R3} ac + (bc + bc) \\ &\xrightarrow{R3} ac + bc \end{aligned}$$

Die Ersetzungsfolgen sind hier sogar unterschiedlich lang.

Man kann jedoch für diesen Ersetzungskalkül zeigen, dass die Normalform (trotz des Nichdeterminismus) stets eindeutig bestimmt ist, man also von *der* Normalform sprechen kann.

Aus dem Beweis ergibt sich mit Algorithmus 9.1 ein Verfahren, mit dem man mit den Regeln R3, R4 und R5 von Seite 196 die Gültigkeit von $s \xrightarrow{\cdot} t$ bzw. $s = t$ entscheiden kann. Dieses Verfahren hat eine lineare Zeitkomplexität und ist sehr viel besser als eines, das auf der Definition der Bisimulation beruht.

Algorithmus 9.1 (Entscheiden von $s \xrightarrow{\cdot} t$ bzw. $s = t$)

Input - Zwei Prozessterme s und t .

Output - TRUE falls $s = t$; FALSE falls die Eigenschaft $s = t$ nicht erfüllt ist.

1. Wende die Regeln R3, R4 und R5 von Seite 196 solange wie möglich auf s an.
 2. Nenne das Ergebnis n (n ist ein Prozessterm in Normalform).
 3. Wende die Regeln R3, R4 und R5 von Seite 196 solange wie möglich auf t an.
 4. Nenne das Ergebnis n' (n' ist ein Prozessterm in Normalform).
 5. Falls $n =_{AC} n'$, gebe TRUE aus, sonst FALSE.
(Dieser Schritt kann mit den Regeln R1 und R2 durchgeführt werden (wobei auf Termination zu achten ist) oder durch andere Verfahren der Textverarbeitung.)
-

Satz 9.17 Der Algorithmus 9.1 entscheidet korrekt, ob zwei Prozessterme s und t aus BPA äquivalent sind.

Beispiel 9.18

Zu entscheiden ist: $(a + a)(cd) + (bc)(d + d) \stackrel{?}{=} ((b + a)(c + c))d$

$$\begin{array}{ll} s \equiv (a + a)(cd) + (bc)(d + d) & t \equiv ((b + a)(c + c))d \\ \xrightarrow{\text{A3}} a(cd) + (bc)(\underline{d + d}) & \xrightarrow{\text{A3}} \underline{((b + a)c)d} \\ \xrightarrow{\text{A3}} a(cd) + \underline{(bc)d} & \xrightarrow{\text{A5}} \underline{(b + a)(cd)} \\ \xrightarrow{\text{A5}} a(cd) + b(cd) \equiv n & \xrightarrow{\text{A4}} b(cd) + a(cd) \equiv n' \end{array}$$

Die beiden berechneten Normalformen n und n' sind äquivalent (modulo $=_{AC}$) und daher auch die Ausgangsterme s und t .

Aufgabe 9.19 Leiten Sie die folgenden drei Äquivalenzen mit dem Kalkül ab.

- a) $((a + a)(b + b))(c + c) = a(bc)$
- b) $(a + a)(bc) + (ab)(c + c) = (a(b + b))(c + c)$
- c) $((a + b)c + ac)d = (b + a)(cd)$

9.2 Parallele und kommunizierende Prozesse

Durch den *Paralleloperator* (*merge*) \parallel wird die parallele (besser: nebenläufige) Ausführung der beiden Prozesse dargestellt, die er als Argument hat.

Ein Term kann jetzt bspw. die Form $(a + b)\|(cd)$ haben.

In den folgenden Regeln sei $\{v, w\} \subseteq A$ und x, x', y, y' seien Prozessterme.

$$\frac{x \xrightarrow{v} \checkmark}{x\|y \xrightarrow{v} y} \quad \frac{x \xrightarrow{v} x'}{x\|y \xrightarrow{v} x'\|y}$$

$$\frac{y \xrightarrow{v} \checkmark}{x\|y \xrightarrow{v} x} \quad \frac{y \xrightarrow{v} y'}{x\|y \xrightarrow{v} x\|y'}$$

Zwei parallel ablaufende Prozesse kommunizieren mittels einer Kommunikationsfunktion.

Eine *Kommunikationsfunktion* $\gamma : A \times A \rightarrow A$ erzeugt für jedes Paar atomarer Aktionen a und b ihre Kommunikations-Aktion $\gamma(a, b)$.

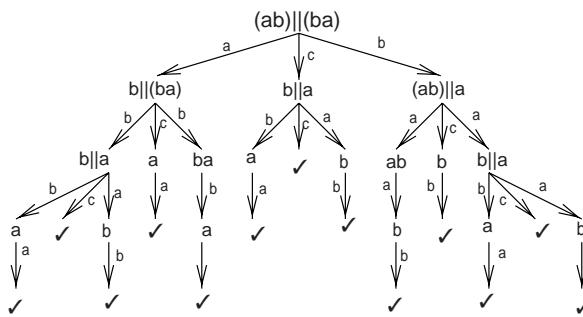
Die Kommunikationsfunktion γ ist kommutativ und assoziativ:

$$\begin{aligned}\gamma(a, b) &\equiv \gamma(b, a) \\ \gamma(\gamma(a, b), c) &\equiv \gamma(a, \gamma(b, c))\end{aligned}$$

Der Paralleloperator kann eine solche Kommunikation enthalten. Sie ist eine unteilbare Aktion beider Prozesse.

$$\begin{array}{c} \frac{x \xrightarrow{v} \checkmark \quad y \xrightarrow{w} \checkmark}{x\|y \xrightarrow{\gamma(v,w)} \checkmark} \quad \frac{x \xrightarrow{v} \checkmark \quad y \xrightarrow{w} y'}{x\|y \xrightarrow{\gamma(v,w)} y'} \\[10pt] \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} \checkmark}{x\|y \xrightarrow{\gamma(v,w)} x'} \quad \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} y'}{x\|y \xrightarrow{\gamma(v,w)} x'\|y'} \end{array}$$

Beispiel 9.20 Der Prozessgraph von $(ab)\|(ba)$ mit $\gamma(x, y) = c$ für alle $x, y \in \{a, b\}$:



Links-Merge und Kommunikations-Merge .

Um eine Axiomatisierung mit dem Paralleloperator zu erhalten, werden zwei Hilfsoperatoren benötigt: *left-merge* $x \sqcup y$ und *communication merge* $x|y$.

Der *Links-Merge-Operator* (*left merge*) \sqcup erlaubt die Ausführung der ersten Transition des ersten (linken) Argumentes:

$$\frac{x \xrightarrow{v} \checkmark}{x \sqcup y \xrightarrow{v} y} \quad \frac{x \xrightarrow{v} x'}{x \sqcup y \xrightarrow{v} x' \| y}$$

Der *Kommunikations-Merge-Operator* (*communication merge*) $|$ stellt die Kommunikation der beiden ersten Transitionen der beiden Argumente dar:

$$\begin{array}{ll} \frac{x \xrightarrow{v} \checkmark \quad y \xrightarrow{w} \checkmark}{x|y \xrightarrow{\gamma(v,w)} \checkmark} & \frac{x \xrightarrow{v} \checkmark \quad y \xrightarrow{w} y'}{x|y \xrightarrow{\gamma(v,w)} y'} \\[10pt] \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} \checkmark}{x|y \xrightarrow{\gamma(v,w)} x'} & \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} y'}{x|y \xrightarrow{\gamma(v,w)} x' \| y'} \end{array}$$

Die Erweiterung der elementaren Prozessalgebra um die Operatoren *Paralleloperator* (*merge*) \parallel , *Links-Merge-Operator* (*left merge*) \sqcup und *Kommunikations-Merge-Operator* (*communication merge*) $|$ heißt *PAP* (*Prozessalgebra mit Parallelismus*).

In ihr sollen die neuen Paralleloperatoren stärker binden als $+$, d.h.: $a \sqcup b + a \parallel b$ steht für $(a \sqcup b) + (a \parallel b)$. Der Paralleloperator \parallel kann durch \sqcup und $|$ ausgedrückt werden: $s \parallel t \Leftrightarrow (s \sqcup t + t \sqcup s) + s|t$.

Anmerkung: PAP ist eine konservative Erweiterung von BPA, d.h. die neuen Transitionsregeln verändern nicht die alten. Anders ausgedrückt bedeutet dies, dass der auf BPA eingeschränkte Prozessgraph unverändert bleibt.

Satz 9.21 Die Äquivalenzrelation Bisimulation ist eine Kongruenzrelation in PAP, d.h.: wenn $s \Leftrightarrow s'$ und $t \Leftrightarrow t'$, dann gilt:

- $s + t \Leftrightarrow s' + t'$,
- $s \cdot t \Leftrightarrow s' \cdot t'$,
- $s \parallel t \Leftrightarrow s' \parallel t'$,
- $s \sqcup t \Leftrightarrow s' \sqcup t'$ und
- $s|t \Leftrightarrow s'|t'$.

Axiome des PAP-Kalküls: Axiome A1, ..., A5 (Seite 194) und

$$M1 \quad x\|y = (x\llcorner y + y\llcorner x) + x|y$$

$$LM2 \quad v\llcorner y = v \cdot y$$

$$LM3 \quad (v \cdot x)\llcorner y = v \cdot (x\|y)$$

$$LM4 \quad (x + y)\llcorner z = x\llcorner z + y\llcorner z$$

$$CM5 \quad v|w = \gamma(v, w)$$

$$CM6 \quad v|(w \cdot y) = \gamma(v, w) \cdot y$$

$$CM7 \quad (v \cdot x)|w = \gamma(v, w) \cdot x$$

$$CM8 \quad (v \cdot x)|(w \cdot y) = \gamma(v, w) \cdot (x\|y)$$

$$CM9 \quad (x + y)|z = x|z + y|z$$

$$CM10 \quad x|(y + z) = x|y + x|z$$

Satz 9.22 Der PAP-Kalkül ist korrekt, d.h.: $s = t \Rightarrow s \xrightarrow{\cdot} t$.

Beweisskizze: Da die Bisimulationsäquivalenz eine Kongruenz ist, genügt es für jedes Axiom $s = t$ die Relation $\sigma(s) \xrightarrow{\cdot} \sigma(t)$ für alle Substitutionen von den Variablen aus s und t in Prozessterme zu beweisen.

Satz 9.23 Der PAP-Kalkül ist vollständig, d.h.: $s \xrightarrow{\cdot} t \Rightarrow s = t$.

Beweisskizze: Dies kann man beweisen, indem man die Axiome für PAP in ein (Term-)Ersetzungskalkül modulo + verwandelt. Jeder Prozessterm über PAP ist in Normalform reduzierbar. Wenn $s \xrightarrow{\cdot} t$ ist, wobei s und t Normalformen s' und t' haben, dann gilt $s' =_{AC} t'$, also auch $s = s' =_{AC} t' = t$.

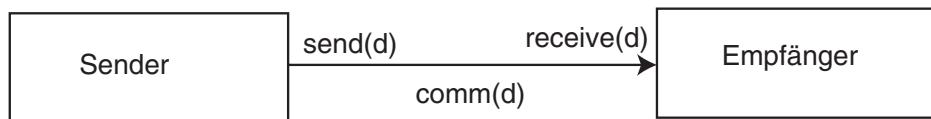


Abbildung 9.3: Kommunikation mit Sender und Empfänger

9.3 Abbruch und Unterdrücken

Abbruch (auch ‘‘deadlock’’) und Unterdrücken (auch ‘‘Verdecken’’, encapsulation) dienen dazu, Teile einer Kommunikation (wie $send(d)$ und das zugehörige $receive(d)$) zu einer Operation (z.B. $comm(d)$, vgl. Abb. 9.3) zu verschmelzen. Darüberhinaus können diese Operationen als Einzelaktionen unterbunden werden.

Der *Abbruchoperator* δ zeigt kein sichtbares Verhalten. Es gibt daher auch dazu keine Transitionsregel.

Die Operation *Unterdrücken* ∂_H , mit $H \subseteq A$, benennt alle Aktionen aus H , die bei ihm als Argument auftreten, in δ um:

$$\frac{x \xrightarrow{v} \checkmark \ (v \notin H)}{\partial_H(x) \xrightarrow{v} \checkmark} \quad \frac{x \xrightarrow{v} x' \ (v \notin H)}{\partial_H(x) \xrightarrow{v} \partial_H(x')}$$

Der Bildbereich der Kommunikationsfunktion γ wird um δ erweitert:

$$\gamma : A \times A \rightarrow A \cup \{\delta\}.$$

Das soll bedeuten: wenn a und b nicht kommunizieren, dann soll $\gamma(a, b) \equiv \delta$ gelten.

Die Operation Unterdrücken erzwingt Kommunikation. Beispielsweise kann $\partial_{\{a,b\}}(a \| b)$ nur als $\gamma(a, b)$ ausgeführt werden (falls $\gamma(a, b) \neq \delta$).

Definition 9.24 Die Erweiterung des Kalküls PAP durch die nachstehenden Axiome für Abbruch und Verdeckung wird mit ACP bezeichnet (algebra of communicating processes).

Axiome des Kalküls ACP ACP besteht aus den Axiomen von PAP und den folgenden für Abbruch und Unterdrücken:

$$\begin{array}{ll} \text{A6} & x + \delta = x \\ \text{A7} & \delta \cdot x = \delta \end{array}$$

$$\begin{array}{ll} \text{D1} & \partial_H(v) = v (v \notin H) \\ \text{D2} & \partial_H(v) = \delta (v \in H) \\ \text{D3} & \partial_H(\delta) = \delta \\ \text{D4} & \partial_H(x + y) = \partial_H(x) + \partial_H(y) \\ \text{D5} & \partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y) \end{array}$$

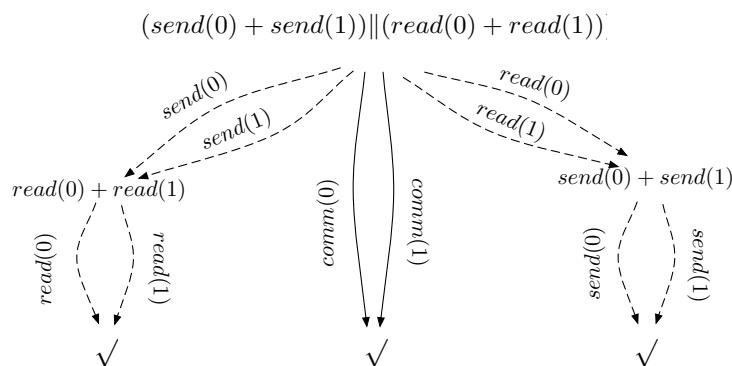
$$\begin{array}{ll} \text{LM11} & \delta \mathbb{L} x = \delta \\ \text{CM12} & \delta | x = \delta \\ \text{CM13} & x | \delta = \delta \end{array}$$

Beispiel 9.25 Der Prozessterm t zum einführenden Beispiel in Abb. 9.3 lautet:

$$t \equiv \partial_{\{send(0), send(1), read(0), read(1)\}}((send(0) + send(1)) \parallel (read(0) + read(1)))$$

mit $\gamma(send(d), read(d)) = comm(d)$ für $d \in \{0, 1\}$.

Die folgende Abbildung zeigt den zugehörigen Prozessgraphen, falls der Unterdrücken-Operator $\partial_{\{send(0), send(1), read(0), read(1)\}}$ weggelassen wird. Seine Hinzufügung bewirkt das Streichen der unterbrochenen Pfeile.



Theorem 9.26 a) Bisimulation ist eine Kongruenz für ACP: wenn $s \underline{\leftrightarrow} s'$ und $t \underline{\leftrightarrow} t'$, dann $s + t \underline{\leftrightarrow} s' + t'$, $s \cdot t \underline{\leftrightarrow} s' \cdot t'$, $s \parallel t \underline{\leftrightarrow} s' \parallel t'$, $s \mathbb{L} t \underline{\leftrightarrow} s' \mathbb{L} t'$, $s | t \underline{\leftrightarrow} s' | t'$ und $\partial_H(s) \underline{\leftrightarrow} \partial_H(s')$.

b) Der Kalkül ACP ist korrekt: $s = t \Rightarrow s \underline{\leftrightarrow} t$.

c) Der Kalkül ACP ist vollständig: $s \underline{\leftrightarrow} t \Rightarrow s = t$.

Beispiel 9.27 Seien $\gamma(a, b) \equiv c$ und $\gamma(a', b') \equiv c'$ zunächst die einzigen Kommunikationsaktionen zwischen Aktionen.

$$\begin{aligned}
 & (a + a') \parallel (b + b') \\
 \stackrel{\text{M1}}{=} & (a + a') \mathbb{L} (b + b') + (b + b') \mathbb{L} (a + a') + (a + a') \parallel (b + b') \\
 \stackrel{\text{LM4, CM9,10}}{=} & a \mathbb{L} (b + b') + a' \mathbb{L} (b + b') + b \mathbb{L} (a + a') + b' \mathbb{L} (a + a') + a|b + a|b' + a'|b + a'|b' \\
 \stackrel{\text{LM2, CM5}}{=} & a \cdot (b + b') + a' \cdot (b + b') + b \cdot (a + a') + b' \cdot (a + a') + c + \delta + \delta + c' \\
 \stackrel{\text{A6}}{=} & a \cdot (b + b') + a' \cdot (b + b') + b \cdot (a + a') + b' \cdot (a + a') + c + c'
 \end{aligned}$$

Sei nun $H = \{a, a', b, b'\}$.

$$\begin{aligned}
 & \partial_H((a + a') \parallel (b + b')) \\
 = & \partial_H(a \cdot (b + b') + a' \cdot (b + b') + b \cdot (a + a') + b' \cdot (a + a') + c + c') \\
 \stackrel{\text{D1,2,4,5}}{=} & \delta \cdot \partial_H(b + b') + \delta \cdot \partial_H(b + b') + \delta \cdot \partial_H(a + a') + \delta \cdot \partial_H(a + a') + c + c' \\
 \stackrel{\text{A6,7}}{=} & c + c'
 \end{aligned}$$

∂_H erzwingt also die Kommunikation zwischen a und b einerseits und zwischen a' und b' andererseits.

Aufgabe 9.28 Konstruieren Sie die Prozessgraphen zu folgenden Prozesstermen:

- a) $\partial_{\{a\}}(ac)$
- b) $\partial_{\{a\}}((a + b)c)$
- c) $\partial_{\{c\}}((a + b)c)$
- d) $\partial_{\{a,b\}}((ab) \parallel (ba))$ mit $\gamma(a, b) = c$

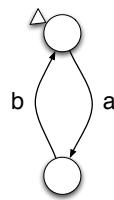


Abbildung 9.4: Transitionssystem für abab...

9.4 Rekursion

Bislang wurden nur Prozesse endlicher Länge spezifiziert. Unendliche Prozesse, wie z.B. der Prozess $ababab\dots$ mit dem Transitionssystem von Abb. 9.4 werden durch Rekursion definiert.

Konkret geschieht dies z.B. durch das folgende *rekursive Gleichungssystem*:

$$\begin{array}{rcl} X & \stackrel{!}{=} & aY \\ Y & \stackrel{!}{=} & bX \end{array}$$

Dabei sind X und Y Rekursionsvariablen, die zwei Zustände des Prozesses repräsentieren.

Definition 9.29 Eine rekursive Spezifikation für die Variablen X_1, \dots, X_n ist ein System von Gleichungen der Form:

$$\begin{array}{rcl} X_1 & \stackrel{!}{=} & t_1(X_1, \dots, X_n) \\ \vdots & & \vdots \\ X_n & \stackrel{!}{=} & t_n(X_1, \dots, X_n) \end{array}$$

Hierbei sind die $t_i(X_1, \dots, X_n)$ ACP-Terme ACP, welche die Variablen X_1, \dots, X_n enthalten dürfen.

Notationshinweis: Wir verwenden das Symbol $\stackrel{!}{=}$ in den Gleichungen $X_n \stackrel{!}{=} t_n(X_1, \dots, X_n)$, um zu zeigen, dass es sich hierbei um etwas anderes handelt als um die Aussagen $s = t$ im Kalkül.

Wenn durch den Kontext eine Verwechslung ausgeschlossen ist, notieren wir aber $\stackrel{!}{=}$ auch als $=$.

Definition 9.30 Prozesse p_1, \dots, p_n werden als eine Lösung einer rekursiven Spezifikation

$$\{X_i \stackrel{!}{=} t_i(X_1, \dots, X_n) \mid i \in \{1, \dots, n\}\}$$

(modulo Bisimulations-Äquivalenz) bezeichnet, falls $p_i \xrightarrow{\cdot} t_i(p_1, \dots, p_n)$ für $i \in \{1, \dots, n\}$ gilt.

Mit den Prozessen p_1, \dots, p_n sind hier ihre Prozessgraphen gemeint. Damit dann der Ausdruck $t_i(p_1, \dots, p_n)$ Sinn ergibt, verstehen wir die Operationen $+$ und \cdot auch als Operationen auf Prozessgraphen.

Wir wollen, dass Lösungen eindeutig bezüglich Bisimulations-Äquivalenz sind, d.h. falls p_1, \dots, p_n und q_1, \dots, q_n zwei solche Lösungen sind, dann erwarten wir $p_i \xrightarrow{\sim} q_i$ für alle $i \in \{1, \dots, n\}$.

Dies muss aber nicht für jede Spezifikation der Fall sein.

Beispiel 9.31

- a) $X \stackrel{!}{=} X$ hat alle Prozesse als Lösung.
- b) Alle Prozesse p mit $p \xrightarrow{a} \vee$ sind Lösungen von der rekursiven Spezifikation $\{X \stackrel{!}{=} a + X\}$.
Beispiel $p \equiv a + cd$: Die linke Seite ist nach Einsetzen ($X \equiv p \equiv a + cd$) bisimilar zur rechten Seite nach Einsetzen ($a + X \equiv a + a + X$).
- c) Alle nichtterminierenden Prozesse sind Lösungen von der rekursiven Spezifikation $\{X \stackrel{!}{=} Xa\}$.
- d) $\{X \stackrel{!}{=} aY, Y \stackrel{!}{=} bX\}$ hat als einzige Lösung den Prozess

$$\begin{aligned} X &\xrightarrow{a} Y \xrightarrow{b} X \xrightarrow{a} Y \xrightarrow{b} \dots \text{ für } X \text{ und} \\ Y &\xrightarrow{b} X \xrightarrow{a} Y \xrightarrow{b} X \xrightarrow{a} \dots \text{ für } Y. \end{aligned}$$
- e) $\{X \stackrel{!}{=} Y, Y \stackrel{!}{=} aX\}$ hat als einzige Lösung $X \xrightarrow{a} X \xrightarrow{a} X \xrightarrow{a} \dots$ für X und Y .
- f) $\{X \stackrel{!}{=} (a + b)\mathbb{L} X\}$ hat als einzige Lösung:

$$X \xrightarrow[a]{b} X \xrightarrow[a]{b} X \xrightarrow[a]{b} X \xrightarrow[a]{b} \dots$$

„Einzig“ bzw. „zwei verschiedene“ ist hier jeweils modulo Bisimulation zu verstehen.

Geschützte rekursive Spezifikation

Wir definieren jetzt einen Spezialfall rekursiver Spezifikation, um Existenz und Eindeutigkeit zu garantieren.

Definition 9.32 Eine rekursive Spezifikation

$$\{X_i \stackrel{!}{=} t_i(X_1, \dots, X_n) \mid i \in \{1, \dots, n\}\}$$

heißt geschützt (guarded), falls die rechten Seiten ihrer Gleichungen mit Hilfe der ACP-Kalküls in die folgende Form überführt werden können:

$$a_1 \cdot s_1(X_1, \dots, X_n) + \dots + a_k \cdot s_k(X_1, \dots, X_n) + b_1 + \dots + b_\ell$$

Außerdem dürfen bei dieser Umformung Variable einer Rekursionsgleichung durch die entsprechende rechte Seite ersetzt werden.

Beispiel 9.33 Sei $\gamma(a, b) \equiv c$ und $\gamma(b, b) \equiv c$.

$\{X \stackrel{!}{=} Y \| Z, Y \stackrel{!}{=} Z + a, Z \stackrel{!}{=} bZ\}$ ist geschützt, denn:

- $Z \stackrel{!}{=} bZ$ hat schon die gewünschte Form.
- $Y \stackrel{!}{=} Z + a$ wird transformiert, indem Z durch bZ ersetzt wird.
- $X \stackrel{!}{=} Y \| Z$ wird transformiert, indem Y durch $bZ + a$ und Z durch bZ ersetzt wird und der so erhaltene Term $(bZ + a) \| bZ$ mittels der Axiome transformiert wird:

$$\begin{aligned} & (bZ + a) \| bZ \\ &= (bZ + a) \perp\!\!\!\perp bZ + bZ \perp\!\!\!\perp (bZ + a) + (bZ + a) | bZ \\ &= bZ \perp\!\!\!\perp bZ + a \perp\!\!\!\perp bZ + bZ \perp\!\!\!\perp (bZ + a) + bZ | bZ + a | bZ \\ &= b(Z \| bZ) + abZ + b(Z \| (bZ + a)) + c(Z \| Z) + cZ \end{aligned}$$

Satz 9.34 Eine rekursive Spezifikation besitzt genau dann eine eindeutige Lösung (modulo Bisimulation), wenn sie geschützt ist.

Zum Beispiel sind einige der rekursiven Spezifikationen von Beispiel 9.31 nicht geschützt. Wegen der Existenz und der Eindeutigkeit können wir von *der* Lösung sprechen.

Definition 9.35 Für eine geschützte rekursive Spezifikation E über den Variablen X_1, \dots, X_n bezeichnet

$$\langle X_i | E \rangle$$

die Lösung der Variablen $X_i, i = 1, \dots, n$.

Transitionsregeln für Rekursion

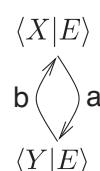
$$\frac{\begin{array}{c} t_i(\langle X_1 | E \rangle, \dots, \langle X_n | E \rangle) \xrightarrow{v} \sqrt{} \\ \hline \end{array}}{\langle X_i | E \rangle \xrightarrow{v} \sqrt{}}$$

$$\frac{\begin{array}{c} t_i(\langle X_1 | E \rangle, \dots, \langle X_n | E \rangle) \xrightarrow{v} y \\ \hline \end{array}}{\langle X_i | E \rangle \xrightarrow{v} y}$$

d.h.: $\langle X_i | E \rangle$ übernimmt das Verhalten von $t_i(\langle X_1 | E \rangle, \dots, \langle X_n | E \rangle)$:

Beispiel 9.36

Sei E die geschützte rekursive Spezifikation $\{X = aY, Y = bX\}$. Der Prozessgraph von $\langle X | E \rangle$ ist:



Wir leiten her: $\langle X|E \rangle \xrightarrow{a} \langle Y|E \rangle$:

$$\begin{array}{c}
 \frac{a \xrightarrow{a} \checkmark \quad \frac{v \xrightarrow{v} \checkmark}{v := a}}{a \cdot \langle Y|E \rangle \xrightarrow{a} \langle Y|E \rangle \quad \frac{x \xrightarrow{v} \checkmark}{x \cdot y \xrightarrow{v} y} \quad v := a, \quad x := a, \quad y := \langle Y|E \rangle} \\
 \hline
 \langle X|E \rangle \xrightarrow{a} \langle Y|E \rangle \quad \frac{a \cdot \langle Y|E \rangle \xrightarrow{v} y}{\langle X|E \rangle \xrightarrow{v} y} \quad v := a, \quad y := \langle Y|E \rangle
 \end{array}$$

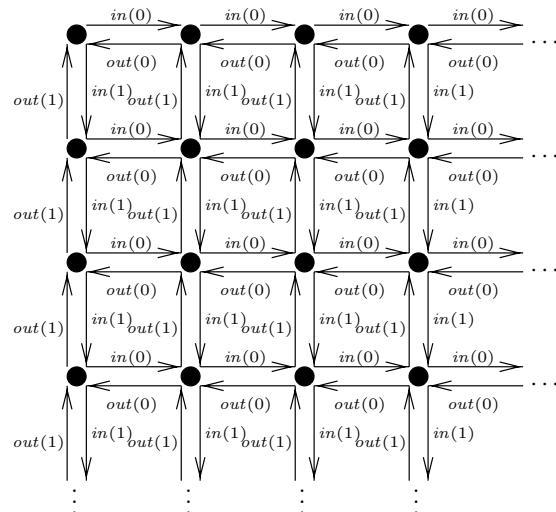
Aufgabe 9.37 Leiten Sie für die folgenden drei Prozessterme den Prozessgraphen ab:

- a) $\langle X|X = ab \rangle$
- b) $\langle X|X = aXb \rangle$
- c) $\langle X|X = aXb + c \rangle$

Anmerkung: ACP mit geschützter Rekursion ist eine konservative Erweiterung von ACP und Bisimulation ist eine Kongruenzrelation.

Beispiel 9.38 Multimenge (bag) über $\{0, 1\}$.

Die Elemente 0 and 1 werden durch $in(0)$ und $in(1)$ in die Multimenge eingefügt und können durch $out(0)$ und $out(1)$ in beliebiger Reihenfolge entfernt werden.



Eine rekursive Spezifikation ist:

$$X = in(0)(X \parallel out(0)) + in(1)(X \parallel out(1))$$

Aufgabe 9.39 Geben Sie eine Bisimulations-Relation für Beispiel 9.38 an. Dabei ist das gezeichnete Transitionssystem mit dem Zustand links oben als Anfangszustand zu

versehen. Die Bisimulation soll zwischen diesem Transitionssystem und dem Prozessgraphen einer Lösung $\langle X|E \rangle$ der rekursiven Spezifikation konstruiert werden.

Definition 9.40 Sei E eine rekursive Spezifikation E der Form:

$$\{X_i = t_i(X_1, \dots, X_n) \mid i \in \{1, \dots, n\}\}$$

Folgende Axiome gelten für ACP mit Rekursion:

- **Rekursives Definitionsprinzip:**

$$\langle X_i|E \rangle = t_i(\langle X_1|E \rangle, \dots, \langle X_n|E \rangle) \quad (i \in \{1, \dots, n\}) \quad (\text{RDP})$$

- **Rekursives Spezifikationsprinzip:**

$$\begin{aligned} & \text{Falls } y_i = t_i(y_1, \dots, y_n) \text{ für } i \in \{1, \dots, n\}, \\ & \text{dann } y_i = \langle X_i|E \rangle \quad (i \in \{1, \dots, n\}) \end{aligned} \quad (\text{RSP})$$

Beispiel 9.41

$$\begin{aligned} \langle Z \mid Z=aZ \rangle &\stackrel{\text{RDP}}{=} a\langle Z \mid Z=aZ \rangle \\ &\stackrel{\text{RDP}}{=} a(a\langle Z \mid Z=aZ \rangle) \\ &\stackrel{\text{A5}}{=} (aa)\langle Z \mid Z=aZ \rangle \end{aligned}$$

Also gilt mit RSP (denn wir haben eine Gleichung mit Z der Form $x = aax$):

$$\langle Z \mid Z=aZ \rangle = \langle X \mid X=(aa)X \rangle$$

Weiter gilt:

$$\begin{aligned} \langle Z \mid Z=aZ \rangle &\stackrel{\text{RDP}}{=} a\langle Z \mid Z=aZ \rangle \\ &\stackrel{\text{RDP}}{=} a(a\langle Z \mid Z=aZ \rangle) \\ &\stackrel{\text{RDP}}{=} a(a(a\langle Z \mid Z=aZ \rangle)) \\ &\stackrel{\text{A5}}{=} ((aa)a)\langle Z \mid Z=aZ \rangle \end{aligned}$$

Also gilt mit RSP (denn wir haben eine Gleichung mit Z der Form $y = aaay$):

$$\langle Z \mid Z=aZ \rangle = \langle Y \mid Y=((aa)a)Y \rangle$$

Also:

$$\begin{aligned} \langle X \mid X=(aa)X \rangle &= \langle Z \mid Z=aZ \rangle \\ &= \langle Y \mid Y=((aa)a)Y \rangle \end{aligned}$$

Beispiel 9.42 Wir wollen für $\gamma(a, b) \equiv c$ die folgende Gleichung beweisen:

$$\partial_{\{a,b\}}(\langle X \mid X=aX \rangle \| \langle Y \mid Y=bY \rangle) = \langle Z \mid Z=cZ \rangle$$

Ansatz: Es gilt:

$$\begin{aligned}
 & \langle X \mid X=aX \rangle \parallel \langle Y \mid Y=bY \rangle \\
 = & \langle X \mid X=aX \rangle \perp\!\!\!\perp \langle Y \mid Y=bY \rangle \\
 + & \langle Y \mid Y=bY \rangle \perp\!\!\!\perp \langle X \mid X=aX \rangle \\
 + & \langle X \mid X=aX \rangle \parallel \langle Y \mid Y=bY \rangle \\
 = & a(\langle X \mid X=aX \rangle \parallel \langle Y \mid Y=bY \rangle) \\
 + & b(\langle Y \mid Y=bY \rangle \parallel \langle X \mid X=aX \rangle) \\
 + & c(\langle X \mid X=aX \rangle \parallel \langle Y \mid Y=bY \rangle)
 \end{aligned}$$

Aus

$$\begin{aligned}
 & \partial_{\{a,b\}}(\langle X \mid X=aX \rangle \parallel \langle Y \mid Y=bY \rangle) \\
 = & c \cdot \partial_{\{a,b\}}(\langle X \mid X=aX \rangle \parallel \langle Y \mid Y=bY \rangle)
 \end{aligned}$$

folgt mit RSP das Ergebnis.

Theorem 9.43 *Der Kalkül ACP mit geschützter Rekursion ist korrekt bezüglich Bisimulation:*

$$s = t \Rightarrow s \xrightarrow{\quad} t$$

RSP ist **nicht** korrekt für ungeschützte Rekursion. Beispielsweise folgt mit RSP aus $t = t$, dass t das Muster der Spezifikation $X = X$ besitzt, d.h. dass t nach RSP eine Lösung ist:

$$t = \langle X \mid X=X \rangle$$

Da diese Gleichung für alle Prozessterme t gilt, erhalten wir also $t_1 = \langle X \mid X=X \rangle$ und $t_2 = \langle X \mid X=X \rangle$ und hieraus folgt direkt $t_1 = t_2$ und das für beliebige Terme t_1, t_2 . Mit anderen Worten: Aus RSP würde für ungeschützte Rekursion folgen, dass alle Terme gleich sind, es also nur einen einzigen Term gibt.

Bemerkung: Der Kalkül ACP mit geschützter Rekursion ist jedoch *nicht* vollständig. Wir werden daher die Form der Rekursion weiter einschränken.

Definition 9.44 *Eine rekursive Spezifikation ist linear, wenn ihre rekursiven Gleichungen die folgende Form haben:*

$$X = a_1X_1 + \cdots + a_kX_k + b_1 + \cdots + b_l \quad a_i, b_j \in A$$

Die linearen rekursiven Spezifikationen sind vollständig axiomatisierbar.

Theorem 9.45 (Vollständigkeit) *Das Kalkül ACP mit den Axiomen RDP und RSP ist eine vollständige Axiomatisierung für Bisimulation.*

Umformungen in rekursiven Spezifikationen

Sei E, E' geschützte rekursive Spezifikationen, wobei E' aus E entsteht, indem die rechten Seiten der Gleichungen folgendermaßen modifiziert werden:

- Die Axiome für ACP mit geschützter Rekursion dürfen benutzt werden.
- Rekursionsvariablen dürfen durch die rechte Seiten ihrer Rekursionsgleichung ersetzt werden.

Dann kann $\langle X | E \rangle = \langle X | E' \rangle$ im Kalkül ACP mit geschützter Rekursion für alle Rekursionsvariablen abgeleitet werden.

Zum Beispiel kann

$$\langle X | X = aX + aX \rangle = \langle X | X = (aa)X \rangle$$

abgeleitet werden, indem

- erst A3 angewandt wird: $(x + x = x)$,
- dann X durch die rechte Seite aX ersetzt wird,
- und dann zuletzt A5 angewandt wird: $((xy)z = x(yz))$.

$$X = aX + aX \rightsquigarrow X = aX \rightsquigarrow X = a(aX) \rightsquigarrow X = (aa)X$$

9.5 Abstraktion

Wird ein spezifiziertes System implementiert, so führt es i.A. mehr Aktionen aus, als in der Spezifikation vorgegeben sind. Diese Aktionen sind zwar für den internen Ablauf wichtig, für den Benutzer oder Auftraggeber jedoch nicht relevant. Um dies zu beschreiben, werden *stille* (silent), *spontane* oder *interne* Aktionen eingeführt. Ihre Bezeichnung ist meist τ (Tau).

Der stille Systemschritt τ kann ohne Vorbedingung ausgeführt werden und terminiert dann erfolgreich:

$$\overline{\tau \xrightarrow{\tau} \checkmark}$$

Der Kalkül ACP über A' mit τ -Transition und Abstraktionsoperator wird mit ACP_τ bezeichnet.

Die Transitionsregeln werden nun so erweitert, dass sie τ enthalten, d.h. die neue Menge von Aktionen ist $A' := A \cup \{\tau\}$ mit einer neuen Kommunikationsfunktion $\gamma : A' \times A' \rightarrow A \cup \{\delta\}$ derart, dass das Bild immer δ ist, falls im Argument ein τ steht, d.h. es findet keine Kommunikation mit der internen Aktion statt.

Der *Abstraktions-Operator* τ_I , mit $I \subseteq A$, benennt alle Aktionen aus I , die er als Argument führt, in τ um:

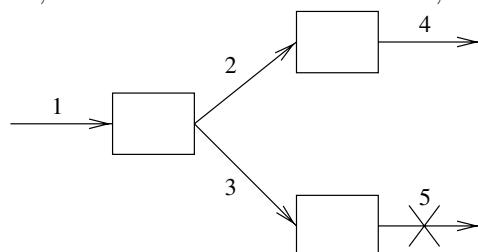
$$\begin{array}{c} \frac{x \xrightarrow{v} \checkmark (v \notin I)}{\tau_I(x) \xrightarrow{v} \checkmark} \quad \frac{x \xrightarrow{v} x' (v \notin I)}{\tau_I(x) \xrightarrow{v} \tau_I(x')} \\ \\ \frac{x \xrightarrow{v} \checkmark (v \in I)}{\tau_I(x) \xrightarrow{\tau} \checkmark} \quad \frac{x \xrightarrow{v} x' (v \in I)}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')} \end{array}$$

Wenn in einer Folge abc von Aktionen b intern ist, dann ist $a\tau c$ (für die externe Spezifikation) äquivalent zu ac , d.h. stille Aktionen sind (extern) wirkungslos.

Wie das folgende Beispiel zeigt, kann nicht generell still gleich wirkungslos gesetzt werden.

Beispiel 9.46

Daten d werden über Kanal 1 empfangen ($r_1(d)$) und über Kanal 2 oder 3 weitergeleitet ($c_2(d), c_3(d)$). Im ersten Fall wird d über Kanal 4 weitergeleitet ($s_4(d)$), im zweiten Fall ist dies jedoch nicht möglich, da der Kanal 5 blockiert ist, d.h. $s_5(d)$ ist blockiert.



Dieses Verhalten wird durch folgenden Prozessausdruck beschrieben:

$$\partial_{\{s_5(d)\}}(r_1(d) \cdot (c_2(d) \cdot s_4(d) + c_3(d) \cdot s_5(d)))$$

$$= r_1(d) \cdot (c_2(d) \cdot s_4(d) + c_3(d) \cdot \delta)$$

(Die Umformung erfolgt mit den Regeln D1,D2,D4,D5 von Seite 203.)

Spezifiziert man $c_2(d)$ und $c_3(d)$ als interne Aktionen, so erhält man

$$r_1(d) \cdot (\tau \cdot s_4(d) + \tau \cdot \delta).$$

Werden beide stille Aktionen τ gestrichen, so erhält man mit $r_1(d) \cdot (s_4(d) + \delta)$ einen Prozess ohne Verklemmung. Er wird daher als nicht (Verhaltens-)äquivalent betrachtet.

Beispiel 9.47 Weitere nichtäquivalente Prozessausdrücke:

- $a + \tau\delta$ und a
- $\partial_{\{b\}}(a + \tau b)$ und $\partial_{\{b\}}(a + b)$
- $a + \tau b$ und $a + b$

Wenn still nicht generell mit wirkungslos gleichgesetzt werden, dann stellt sich die Frage: Welche τ -Transitionen sind wirkungslos?

Antwort: Die τ -Transitionen, deren Streichung nicht das Verhalten ändert.

Zum Beispiel: $a + \tau(a + b)$ und $a + b$, denn nach der Ausführung von τ ist a immer noch ausführbar.

Dies wird durch die folgende Definition der *Verzweigungs-Bisimulation* (branching bisimulation) formalisiert. (Hierbei bezeichne \equiv die syntaktische Gleichheit.)

Definition 9.48 Eine Verzweigungs-Bisimulation ist eine binäre Relation \mathcal{B} auf Prozessen, für die mit $a \in A' = A \cup \{\tau\}$ gilt:

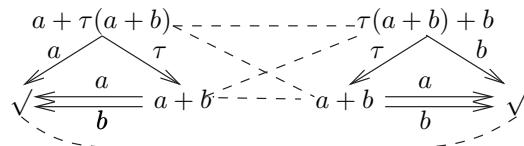
1. Wenn $p \mathcal{B} q$ und $p \xrightarrow{a} p'$, dann
 - entweder $a \equiv \tau$ und $p' \mathcal{B} q$
 - oder $q \xrightarrow{\tau} \dots \xrightarrow{\tau} q_0$ mit $p \mathcal{B} q_0$ und $q_0 \xrightarrow{a} q'$ mit $p' \mathcal{B} q'$
2. Wenn $p \mathcal{B} q$ und $q \xrightarrow{a} q'$, dann
 - entweder $a \equiv \tau$ und $p \mathcal{B} q'$
 - oder $p \xrightarrow{\tau} \dots \xrightarrow{\tau} p_0$ mit $p_0 \mathcal{B} q$ und $p_0 \xrightarrow{a} p'$ mit $p' \mathcal{B} q'$
3. Wenn $p \mathcal{B} q$ und p ist terminiert, dann $q \xrightarrow{\tau} \dots \xrightarrow{\tau} q_0$, so dass $p \mathcal{B} q_0$ und q_0 ist terminiert.
4. Wenn $p \mathcal{B} q$ und q ist terminiert, dann $p \xrightarrow{\tau} \dots \xrightarrow{\tau} p_0$, so dass $p_0 \mathcal{B} q$ und p_0 ist terminiert.

Die τ -Folgen haben eine endliche Länge $n \geq 0$.

Zwei Prozesse p und q heißen verzweigungsbisimilär (branching bisimilar), in Zeichen $p \underline{\leftrightarrow}_b q$, wenn es eine Verzweigungs-Bisimulations-Relation \mathcal{B} gibt mit $p \mathcal{B} q$.

Beispiel 9.49

$$a + \tau(a + b) \underline{\leftrightarrow}_b \tau(a + b) + b$$



Die Relation \mathcal{B} ist definiert durch:

$$a + \tau(a + b) \mathcal{B} \tau(a + b) + b$$

$$a + b \mathcal{B} \tau(a + b) + b$$

$$a + \tau(a + b) \mathcal{B} a + b$$

$$a + b \mathcal{B} a + b$$

$$\checkmark \mathcal{B} \checkmark$$

Aufgabe 9.50

1. Geben Sie eine Verzweigungs-Bisimulations-Relation an, die zeigt dass $\tau(\tau(a + b) + b) + a$ und $a + b$ verzweigungsbisimilar sind.
2. Begründen Sie, warum $\tau a + \tau b$ und $a + b$ nicht verzweigungsbisimilar sind.

Die Verzweigungs-Bisimulations-Relation ist zwar eine Äquivalenzrelation, aber keine Kongruenz bezüglich BPA. Beispielsweise sind τa und a verzweigungsbisimilar, aber nicht $\tau a + b$ und $a + b$.

Ansatz: Einige Terme werden nicht mehr als äquivalent angesehen, indem wir die Bedingungen verschärfen.

Milner [Mil89] hat gezeigt, dass man dieses Problem dadurch lösen kann, dass eine Initialisierungsbedingung gefordert wird: Initiale τ -Transitionen werden nie eliminiert
Dann wird in folgenden Fällen wird keine Äquivalenz erzeugt:

- $a + \tau\delta$ und a
- $\partial_{\{b\}}(a + \tau b)$ und $\partial_{\{b\}}(a + b)$
- $a + \tau b$ und $a + b$
- b und τb

Definition 9.51 Eine initiale Verzweigungs-Bisimulation (rooted branching bisimulation) ist eine binäre Relation \mathcal{B} auf Prozessen, für die mit $a \in A' = A \cup \{\tau\}$ gilt:

1. Wenn $p \mathcal{B} q$ und $p \xrightarrow{a} p'$, dann $q \xrightarrow{a} q'$ mit $p' \xleftrightarrow{b} q'$.
2. Wenn $p \mathcal{B} q$ und $q \xrightarrow{a} q'$, dann $p \xrightarrow{a} p'$ mit $p' \xleftrightarrow{b} q'$.
3. Wenn $p \mathcal{B} q$ und p terminiert, dann terminiert auch q .
4. Wenn $p \mathcal{B} q$ und q terminiert, dann terminiert auch p .

Zwei Prozesse p und q heißen initial verzweigungsbisimilar (rooted branching bisimilar), in Zeichen $p \xleftrightarrow{rb} q$, wenn es eine initiale Verzweigungs-Bisimulations-Relation \mathcal{B} gibt mit $p \mathcal{B} q$.

Initiale Verzweigungs-Bisimulation ist (wie Verzweigungs-Bisimulation auch) eine Äquivalenzrelation.

Das folgende Lemma zeigt, dass Initiale Verzweigungs-Bisimulation feiner als Verzweigungs-Bisimulation ist.

Lemma 9.52 Zwischen den Äquivalenzen gelten die Beziehungen:

$$\underline{\leftrightarrow} \subseteq \underline{\leftrightarrow}_{rb} \subseteq \underline{\leftrightarrow}_b$$

Falls τ in den Termen nicht vorkommt (wie z.B. in ACP), dann gilt sogar:

$$\underline{\leftrightarrow} = \underline{\leftrightarrow}_{rb} = \underline{\leftrightarrow}_b$$

Aufgabe 9.53 Bestimmen Sie für jedes der folgenden Paare von Prozessstermen, ob es bisimilar, initial verzweigungsbisimilar oder verzweigungsbisimilar ist bzw. nicht ist - möglichst mit stichhaltiger Begründung:

- a) $(a + b)(c + d)$ und $ac + ad + bc + bd$,
- b) $(a + b)(c + d)$ und $(b + a)(d + c) + a(c + d)$,
- c) $\tau(b + a) + \tau(a + b)$ und $a + b$,
- d) $c(\tau(b + a) + \tau(a + b))$ und $c(a + b)$ sowie
- e) $a(\tau b + c)$ und $a(b + \tau c)$.

9.5.1 Geschützte lineare Rekursion

Betrachten wir lineare Rekursion im Zusammenhang mit τ -Aktionen.

Alle Prozesssterme τs stellen eine Lösung für die Spezifikation $X = \tau X$ dar, da $\tau s \xrightarrow{rb} \tau \tau s$.

Die τ -Aktion reicht also nicht aus, um eine Eindeutigkeit zu garantieren. Wir müssen also den Begriff der geschützten Rekursion anpassen.

Definition 9.54 1. Eine rekursive Spezifikation ist linear, wenn ihre rekursiven Gleichungen die folgende Form haben:

$$X = a_1 X_1 + \cdots + a_k X_k + b_1 + \cdots + b_\ell \quad (a_i, b_j \in A \cup \{\tau\})$$

2. Eine lineare rekursive Spezifikation E ist geschützt, wenn es keine unendliche Folge von τ -Transitionen der folgenden Form gibt:

$$\langle X | E \rangle \xrightarrow{\tau} \langle X' | E \rangle \xrightarrow{\tau} \langle X'' | E \rangle \xrightarrow{\tau} \cdots$$

Die geschützten linearen rekursiven Spezifikationen sind genau diejenigen rekursiven Spezifikationen, die eine eindeutige Lösung modulo initialer Verzweigungs-Bisimulation haben.

Wieder geben wir eine Axiomatisierung des erweiterten Kalküls ACP_τ an.

ACP_τ bezeichne jetzt den Kalkül ACP mit τ -Aktion, Abstraktionsoperator, geschützter linearer Rekursion und den folgenden Axiomen:

Axiome für Abstraktion (mit $v \in A' = A \cup \{\tau\}, I \subseteq A$)

$$\begin{array}{ll} \text{B1} & v \cdot \tau = v \\ \text{B2} & v \cdot (\tau \cdot (x + y) + x) = v \cdot (x + y) \end{array}$$

$$\begin{array}{ll} \text{TI1} & \tau_I(v) = v \ (v \notin I) \\ \text{TI2} & \tau_I(v) = \tau \ (v \in I) \\ \text{TI3} & \tau_I(\delta) = \delta \\ \text{TI4} & \tau_I(x + y) = \tau_I(x) + \tau_I(y) \\ \text{TI5} & \tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y) \end{array}$$

Satz 9.55 *Initiale Verzweigungs-Bisimulation ist eine Kongruenzrelation für ACP_τ und geschützter linearer Rekursion.*

Theorem 9.56 *Der Kalkül ACP_τ mit Abstraktion und geschützter linearer Rekursion ist korrekt und vollständig in Bezug auf initiale Verzweigungsbisimulation.*

Aufgabe 9.57 Leiten sie $\tau_{\{b\}}(\langle X | X = aY, Y = bX \rangle) = \langle Z | Z = aZ \rangle$ in ACP_τ ab.

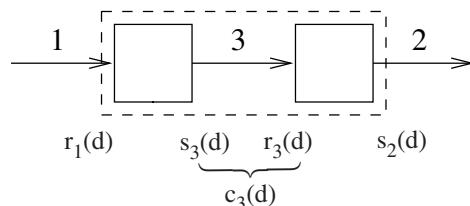
Beispiel 9.58 (Tandempuffer)

Konvention: Um Ausdrücke zu vereinfachen, wird in den folgenden Beispielrechnungen eine Lösung $\langle Q | Q = r_1 s_1 Q \rangle$ einer rekursiven Spezifikation abkürzend als Q geschrieben. Betrachtet werden zwei in Serie angeordnete Puffer der Kapazität 1 die mit Kanälen 1, 2 und 3 verbunden sind. $r_i(d)$ bzw. $s_i(d)$ bezeichne das Schreiben bzw. Lesen vom Kanal i . Δ ist eine endliche Menge von Daten. Das Verhalten ist:

$$\begin{array}{ll} Q_1 & = \sum_{d \in \Delta} r_1 s_3 Q_1 \\ Q_2 & = \sum_{d \in \Delta} r_3 s_2 Q_2 \end{array}$$

wobei $d \in \Delta$ weggelassen wird, d.h. wir tun so, als ob Δ nur ein Element enthalten würde:

$$\begin{array}{ll} Q_1 & = r_1 s_3 Q_1 \\ Q_2 & = r_3 s_2 Q_2 \end{array}$$



Die Puffer Q_1 und Q_2 der Kapazität 1 arbeiten parallel und sind durch die Aktion $\gamma(s_3, r_3) = c_3$ synchronisiert:

$$\tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(Q_2 \| Q_1))$$

Wir beweisen, dass das gemeinsame Verhalten dasjenige eines Puffers der Kapazität 2 ist:

$$\begin{aligned} X &= r_1 Y \\ Y &= r_1 Z + s_2 X \\ Z &= s_2 Y \end{aligned}$$

Dazu überlegen wir, was die Lösung dieses rekursiven Gleichungssystems E ist.

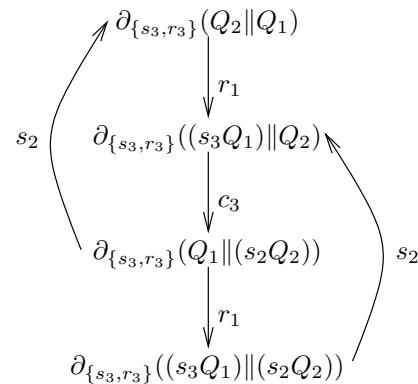
Als erstes berechnen wir:

$$\begin{aligned} & Q_2 \| Q_1 \\ \stackrel{\text{M1}}{=} & Q_2 \llbracket Q_1 + Q_1 \llbracket Q_2 + Q_2 \| Q_1 \\ \stackrel{\text{RDP}}{=} & (r_3 s_2 Q_2) \llbracket Q_1 + (r_1 s_3 Q_1) \llbracket Q_2 \\ & + (r_3 s_2 Q_2) \mid (r_1 s_3 Q_1) \\ \stackrel{\text{LM3,CM8}}{=} & r_3 \cdot ((s_2 Q_2) \| Q_1) + r_1 \cdot ((s_3 Q_1) \| Q_2) \\ & + \delta \cdot ((s_2 Q_2) \parallel (s_3 Q_1)) \\ \stackrel{\text{A7,A6}}{=} & r_3 \cdot ((s_2 Q_2) \| Q_1) + r_1 \cdot ((s_3 Q_1) \| Q_2) \\ \\ & \partial_{\{s_3, r_3\}}(Q_2 \| Q_1) \\ = & \partial_{\{s_3, r_3\}}(r_3 \cdot ((s_2 Q_2) \| Q_1) + r_1 \cdot ((s_3 Q_1) \| Q_2)) \\ = & \partial_{\{s_3, r_3\}}(r_3 \cdot ((s_2 Q_2) \| Q_1)) \\ & + \partial_{\{s_3, r_3\}}(r_1 \cdot ((s_3 Q_1) \| Q_2)) \\ = & \partial_{\{s_3, r_3\}}(r_3) \cdot \partial_{\{s_3, r_3\}}((s_2 Q_2) \| Q_1) \\ & + \partial_{\{s_3, r_3\}}(r_1) \cdot \partial_{\{s_3, r_3\}}((s_3 Q_1) \| Q_2) \\ = & \delta \cdot \partial_{\{s_3, r_3\}}((s_2 Q_2) \| Q_1) \\ & + r_1 \cdot \partial_{\{s_3, r_3\}}((s_3 Q_1) \| Q_2) \\ = & r_1 \cdot \partial_{\{s_3, r_3\}}((s_3 Q_1) \| Q_2) \end{aligned}$$

Weiter rechnen wir:

$$\begin{aligned} \partial_{\{s_3, r_3\}}((s_3 Q_1) \| Q_2) &= c_3 \cdot \partial_{\{s_3, r_3\}}(Q_1 \| (s_2 Q_2)) \\ \partial_{\{s_3, r_3\}}(Q_1 \| (s_2 Q_2)) &= r_1 \cdot \partial_{\{s_3, r_3\}}((s_3 Q_1) \| (s_2 Q_2)) \\ &+ s_2 \cdot \partial_{\{s_3, r_3\}}(Q_2 \| Q_1) \\ \partial_{\{s_3, r_3\}}((s_3 Q_1) \| (s_2 Q_2)) &= s_2 \cdot \partial_{\{s_3, r_3\}}((s_3 Q_1) \| Q_2) \end{aligned}$$

Wir fassen die gerechneten Transitionsübergänge zusammen:



Die Axiome für die τ -Aktion und Abstraktion ergeben:

$$\begin{aligned}
 & \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
 = & \tau_{\{c_3\}}(r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2)) \\
 = & r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2)) \\
 = & r_1 \cdot \tau_{\{c_3\}}(c_3 \cdot \partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2))) \\
 = & r_1 \cdot \tau \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2))) \\
 = & r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2)))
 \end{aligned}$$

Weiter rechnen wir:

$$\begin{aligned}
 \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2))) &= \\
 & r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 Q_1) \| (s_2 Q_2))) \\
 & + s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
 \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 Q_1) \| (s_2 Q_2))) &= \\
 & s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2)))
 \end{aligned}$$

Damit erhalten wir als Lösung für die lineare rekursive Spezifikation E für einen Puffer der Kapazität 2:

$$\begin{aligned}
 X &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
 Y &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2))) \\
 Z &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 Q_1) \| (s_2 Q_2)))
 \end{aligned}$$

9.5.2 Die Fairness-Regel

Es ist möglich, durch Abstraktion τ -Schleifen zu konstruieren: So führt $\tau_{\{a\}}(\langle X \mid X=aX \rangle)$ unendlich lange die τ -Aktion aus.

Definition 9.59 Sei E eine geschützte lineare rekursive Spezifikation, C eine Teilmenge ihrer Variablen und $I \subset A$ eine Menge von Aktionen.

- C heißt (Fairness-)Gruppe (*cluster*) für I , falls für je zwei Rekursionsvariablen X und Y in C $\langle X \mid E \rangle \xrightarrow{b_1} \dots \xrightarrow{b_m} \langle Y \mid E \rangle$ und $\langle Y \mid E \rangle \xrightarrow{c_1} \dots \xrightarrow{c_n} \langle X \mid E \rangle$ für Aktionen $b_1, \dots, b_m, c_1, \dots, c_n \in I \cup \{\tau\}$ gilt.
- a und aX heißen Ausgang (*exit*) der Gruppe C falls:
 1. a oder aX ein Summand auf der rechten Seite der Rekursionsgleichung für eine Rekursionsvariable in C ist, und
 2. im Fall aX zusätzlich $a \notin I \cup \{\tau\}$ oder $X \notin C$ gilt.

Die Fairness-Regel CFAR: Falls X in einer Gruppe C für I mit Ausgängen $\{v_1 Y_1, \dots, v_m Y_m, w_1, \dots, w_n\}$ ist, dann gilt

$$\tau \cdot \tau_I(\langle X \mid E \rangle) = \tau \cdot \tau_I(v_1 \langle Y_1 \mid E \rangle + \dots + v_m \langle Y_m \mid E \rangle + w_1 + \dots + w_n)$$

Zur Erläuterung von CFAR sei E das Gleichungssystem:

$$\begin{aligned} X_1 &= aX_2 + b_1 \\ &\vdots \\ X_{n-1} &= aX_n + b_{n-1} \\ X_n &= aX_1 + b_n \end{aligned}$$

$\tau_{\{a\}}(\langle X_1 | E \rangle)$ führt τ -Transitionen aus, bis eine Aktion b_i für $i \in \{1, \dots, n\}$ ausgeführt wird.

Faire Abstraktion bedeutet, dass $\tau_{\{a\}}(\langle X_1 | E \rangle)$ nicht für immer in einer τ -Schleife bleibt, d.h. irgend wann wird einmal ein b_i ausgeführt:

$$\tau_{\{a\}}(\langle X_1 | E \rangle) \xrightarrow{\perp rb} b_1 + \tau(b_1 + \dots + b_n)$$

Anfangs führt $\tau_{\{a\}}(\langle X_1 | E \rangle)$ die Aktionen b_1 oder τ aus. Im letzteren Fall wird nach einer Reihe von möglichen τ -Aktionen ein b_i ausgeführt. Dies entspricht für $n = 3$ der Situation, dass in dem P/T-Netz von Abbildung 9.5 die mit f bezeichneten Transitionen b_1 , b_2 und b_3 fair schalten (Definition 6.31 b)), d.h. der Zyklus der mit a bezeichneten Transitionen einmal verlassen wird.

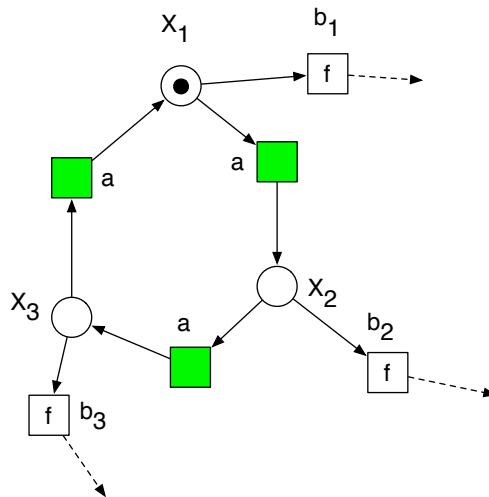


Abbildung 9.5: Faire Abstraktion als P/T-Netz mit fair schaltenden Transitionen.

Beispiel 9.60

$$X = \text{heads} \cdot X + \text{tails}$$

$\langle X | E \rangle$ stellt das Werfen einer idealen Münze dar, das mit *tails* endet. Von den Ergebnissen „Kopf“ wird abstrahiert: (head) $\tau_{\{\text{heads}\}}(\langle X | E \rangle)$.

$\{X\}$ ist die einzige Gruppe für $\{\text{heads}\}$ mit dem einzigen Ausgang *tails*. Daher erhält man mit der Regel CFAR:

$$\begin{aligned} \tau \cdot \tau_{\{\text{heads}\}}(\langle X | E \rangle) &= \tau \cdot \tau_{\{\text{heads}\}}(\text{tails}) \\ &= \tau \cdot \text{tails} \end{aligned}$$

und

$$\begin{aligned}
 \tau_{\{heads\}}(\langle X|E \rangle) &= \tau_{\{heads\}}(heads \cdot \langle X|E \rangle + tails) \\
 &= \tau \cdot \tau_{\{heads\}}(\langle X|E \rangle) + tails \\
 &= \tau \cdot tails + tails
 \end{aligned}$$

Anmerkung: Der Kalkül ACP $_{\tau}$ mit Abstraktions, geschützter linearer Rekursion und Fairnessregel ist korrekt und vollständig in Bezug auf initiale Verzweigungsbisimulation:

$$s = t \Leftrightarrow s \underline{\leftrightarrow}_{rb} t$$

9.6 Verifikation des Alternierbitprotokolls

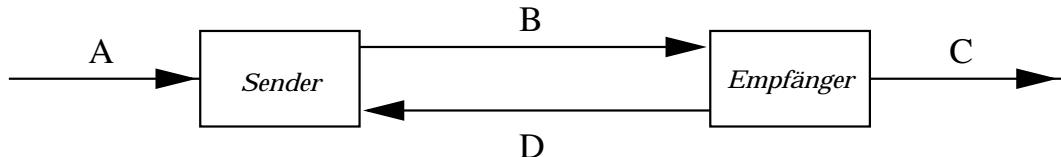
Es wird nun mit den Mitteln der Prozessalgebra ein Korrektheitsbeweis für das Alternierbitprotokoll geführt.

Es sollen Daten über einen FIFO-Kanal gesendet werden. Dazu wird ein Datum am Kanal A eingelesen: $r_A(d)$ und dann vom Kanal C gesendet: $s_C(d)$. Die Kanalkapazität beträgt $n = 1$, so dass jedes Datum erst bei C ausgeliefert werden muss, bevor bei A ein neues akzeptiert werden kann.

Die Spezifikation des gewünschten externen Verhaltens ist:

$$E = \left\{ X = \sum_{d \in \Delta} r_A(d) \cdot s_C(d) \cdot X \right\} \quad (9.1)$$

Zunächst wird das Alternierbitprotokoll als Programm (als Sender S_b und Empfänger R_b) dargestellt und dann sein Verhalten als übereinstimmend mit der geforderten Spezifikation bewiesen. Dies erfolgt indirekt über die Verhaltengleichheit (Bisimilarität) der entsprechenden Prozessgraphen, also der Transitionssysteme. Deren Zustandsexploration wird jedoch durch die in der Prozessalgebra mögliche Parametrisierung der verschiedenen Eingaben vermieden. Der entsprechende Zustandsgraph wird zwar (für den stark vereinfachenden Fall eines einelementigen Datentyps Δ der Eingabe) graphisch dargestellt, dies dient jedoch nur der Erläuterung und ist nicht Teil des Beweises.



Datenelemente d werden von einem Sender über einen störanfälligen Kanal B zu einem Empfänger gesandt. Aufgabe des Protokolls ist es, durch wiederholtes Senden die Störung zu kompensieren. Dazu fügt der Sender den Datenelementen alternativ ein Bit 0 oder 1 hinzu. Wenn der Empfänger das Datenelement korrekt erhalten hat, sendet er das Bit über den (ebenfalls störanfälligen) Kanal D an den Sender als Quittung zurück. Falls die Nachricht gestört war, sendet er jedoch das vorangehende Bit zurück.

Der Sender wiederholt das Senden eines Datenelementes mit Bit b solange, bis er eine Quittung b erhält. Dann sendet er das nächste Datenelement mit Bit $1 - b$ bis er $1 - b$ als Quittung erhält.

Betrachten wir nun die Implementation.

Spezifikation des Senders für das Senden mit Bit b :

$$\begin{aligned} S_b &= \sum_{d \in \Delta} r_A(d) \cdot T_{db} \\ T_{db} &= (s_B(d, b) + s_B(\perp)) \cdot U_{db} \\ U_{db} &= r_D(b) \cdot S_{1-b} + (r_D(1-b) + r_D(\perp)) \cdot T_{db} \end{aligned}$$

Für ein Argument u bedeutet $r_X(u)$ bzw. $s_X(u)$ wieder das Lesen von dem bzw. das Schreiben in den Kanal X .

Spezifikation des Empfängers für das Empfangen mit Bit b :

$$\begin{aligned} R_b &= \sum_{d \in \Delta} \{r_B(d, b) \cdot s_C(d) \cdot Q_b \\ &\quad + r_B(d, 1-b) \cdot Q_{1-b}\} + r_B(\perp) \cdot Q_{1-b} \\ Q_b &= (s_D(b) + s_D(\perp)) \cdot R_{1-b} \end{aligned}$$

Als externes Verhalten des Alternierbitprotokolls erhalten wir also:

$$\tau_I(\partial_H(R_0 \| S_0))$$

Dabei werden durch ∂_H falsche Kommunikationspaare ausgeschlossen, d.h. wir definieren

- $\gamma(s_B(d, b), r_B(d, b)) := c_B(d, b)$
- $\gamma(s_B(\perp), r_B(\perp)) := c_B(\perp)$
- $\gamma(s_D(b), r_D(b)) := c_D(b)$
- $\gamma(s_D(\perp), r_D(\perp)) := c_D(\perp)$

für $d \in \Delta, b \in \{0, 1\}$. Dabei ist \perp die gestörte Nachricht. H besteht also aus allen Aktionen, die auf der linken Seite dieser Definition vorkommen. τ_I abstrahiert von den internen Aktionen in $I := \{c_B(d, b), c_D(b) | d \in \Delta, b \in \{0, 1\}\} \cup \{c_B(\perp), c_D(\perp)\}$.

Als Korrektheitsbeweis werden wir ableiten:

$$\tau_I(\partial_H(R_0 \| S_0)) = \sum_{d \in \Delta} r_A(d) \cdot s_C(d) \cdot \tau_I(\partial_H(R_0 \| S_0))$$

bzw. noch direkter:

$$\tau_I(\partial_H(R_0 \| S_0)) \text{ ist Lösung von } E = \left\{ X = \sum_{d \in \Delta} r_A(d) \cdot s_C(d) \cdot X \right\} \quad (9.2)$$

Das spezifizierte Protokoll hat damit also das gewünschte Verhalten:

$$r_A(d_0), s_C(d_0), r_A(d_1), s_C(d_1), r_A(d_2), s_C(d_2), \dots$$

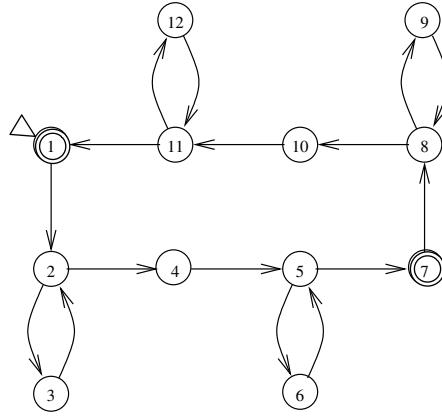
d.h. alle Datenelemente d_0, d_1, d_2, \dots werden in der richtigen Reihenfolge (und ohne Verlust oder Verdoppelung) übertragen.

Als erster Schritt leitet man unter Benutzung der Axiome M1, RDP, LM4, CM9, CM10, LM3, CM8, A6, A7 ab:

$$\begin{aligned} R_0 \| S_0 &= \sum_{d' \in \Delta} \{r_B(d', 0) \cdot ((s_C(d')Q_0) \| S_0) \\ &\quad + r_B(d', 1) \cdot (Q_1 \| S_0)\} + r_B(\perp) \cdot (Q_1 \| S_0) \\ &\quad + \sum_{d \in \Delta} r_A(d) \cdot (T_{d0} \| R_0) \end{aligned}$$

Weiter erhält man mit D4, D1, D2, D5, A6, A7:

$$\partial_H(R_0 \| S_0) = \sum_{d \in \Delta} r_A(d) \cdot \partial_H(T_{d0} \| R_0)$$


 Abbildung 9.6: Transitionssystem von $\partial_H(R_0||S_0)$

Diese Äquivalenz entspricht dem Übergang vom Zustand 1 in den Zustand 2 im Transitionssystem von $\partial_H(R_0||S_0)$ in Abb. 9.6.

$$\begin{aligned}
 T_{d0} \| R_0 &= (s_B(d, 0) + s_B(\perp)) \cdot (U_{d0} \| R_0) \\
 &+ \sum_{d' \in \Delta} \{ r_B(d', 0) \cdot ((s_C(d') Q_0) \| T_{d0}) \\
 &+ r_B(d', 1) \cdot (Q_1 \| T_{d0}) \} + r_B(\perp) \cdot (Q_1 \| T_{d0}) \\
 &+ c_B(d, 0) \cdot (U_{d0} \| (s_C(d) Q_0)) + c_B(\perp) \cdot (U_{d0} \| Q_1) \\
 \partial_H(T_{d0} \| R_0) &= c_B(d, 0) \cdot \partial_H(U_{d0} \| (s_C(d) Q_0)) \\
 &+ c_B(\perp) \cdot \partial_H(U_{d0} \| Q_1)
 \end{aligned}$$

Diese Äquivalenz entspricht dem Übergang vom Zustand 2 in die Zustände 3 und 4 im Transitionssgraph von Abb. 9.6.

Entsprechend erhält man für die Übergänge bis zum Zustand 7:

$$\begin{aligned}
 U_{d0} \| Q_1 &= r_D(0) \cdot (S_1 \| Q_1) \\
 &+ (r_D(1) + r_D(\perp)) \cdot (T_{d0} \| Q_1) \\
 &+ (s_D(1) + s_D(\perp)) \cdot (R_0 \| U_{d0}) \\
 &+ (c_D(1) + c_D(\perp)) \cdot (T_{d0} \| R_0) \\
 \partial_H(U_{d0} \| Q_1) &= (c_D(1) + c_D(\perp)) \cdot \partial_H(T_{d0} \| R_0) \\
 U_{d0} \| (s_C(d) Q_0) &= r_D(0) \cdot (S_1 \| (s_C(d) Q_0)) \\
 &+ (r_D(1) + r_D(\perp)) \cdot (T_{d0} \| (s_C(d) Q_0)) \\
 &+ s_C(d) \cdot (Q_0 \| U_{d0}) \\
 \partial_H(U_{d0} \| (s_C(d) Q_0)) &= s_C(d) \cdot \partial_H(Q_0 \| U_{d0}) \\
 Q_0 \| U_{d0} &= (s_D(0) + s_D(\perp)) \cdot (R_1 \| U_{d0}) \\
 &+ r_D(0) \cdot (S_1 \| Q_0) + (r_D(1) + r_D(\perp)) \cdot (T_{d0} \| Q_0) \\
 &+ c_D(0) \cdot (R_1 \| S_1) + c_D(\perp) \cdot (R_1 \| T_{d0}) \\
 \partial_H(Q_0 \| U_{d0}) &= c_D(0) \cdot \partial_H(R_1 \| S_1) \\
 &+ c_D(\perp) \cdot \partial_H(R_1 \| T_{d0})
 \end{aligned}$$

$$\begin{aligned}
 R_1 \| T_{d0} &= \sum_{d' \in \Delta} \{ r_B(d', 1) \cdot ((s_C(d') Q_1) \| T_{d0}) \\
 &\quad + r_B(d', 0) \cdot (Q_0 \| T_{d0}) \} + r_B(\perp) \cdot (Q_0 \| T_{d0}) \\
 &\quad + (s_B(d, 0) + s_B(\perp)) \cdot (U_{d0} \| R_1) \\
 &\quad + (c_B(d, 0) + c_B(\perp)) \cdot (Q_0 \| U_{d0}) \\
 \partial_H(R_1 \| T_{d0}) &= (c_B(d, 0) + c_B(\perp)) \cdot \partial_H(Q_0 \| U_{d0})
 \end{aligned}$$

Dann erhält man für die Übergänge von Zustand 7 bis zum Zustand 1:

$$\begin{aligned}
 \partial_H(R_1 \| S_1) &= \sum_{d \in \Delta} r_A(d) \cdot \partial_H(T_{d1} \| R_1) \\
 \partial_H(T_{d1} \| R_1) &= c_B(d, 1) \cdot \partial_H(U_{d1} \| (s_C(d) \cdot Q_1)) \\
 &\quad + c_B(\perp) \cdot \partial_H(U_{d1} \| Q_0) \\
 \partial_H(U_{d1} \| Q_0) &= (c_D(0) + c_D(\perp)) \cdot \partial_H(T_{d1} \| R_1) \\
 \partial_H(U_{d1} \| (s_C(d) Q_1)) &= s_C(d) \cdot \partial_H(Q_1 \| U_{d1}) \\
 \partial_H(Q_1 \| U_{d1}) &= c_D(1) \cdot \partial_H(R_0 \| S_0) \\
 &\quad + c_D(\perp) \cdot \partial_H(R_0 \| T_{d1}) \\
 \partial_H(R_0 \| T_{d1}) &= (c_B(d, 1) + c_B(\perp)) \cdot \partial_H(Q_1 \| U_{d1})
 \end{aligned}$$

Insgesamt ergeben sich 12 Gleichungen (die den 12 Zuständen entsprechen) und mit RSP:

$$\partial_H(R_0 \| S_0) = \langle X_1 | E \rangle$$

wobei E die folgende lineare rekursive Spezifikation ist.

$$\left\{
 \begin{array}{lcl}
 X_1 &=& \sum_{d' \in \Delta} r_A(d') \cdot X_{2d'} \\
 X_{2d} &=& c_B(d, 0) \cdot X_{4d} + c_B(\perp) \cdot X_{3d} \\
 X_{3d} &=& (c_D(1) + c_D(\perp)) \cdot X_{2d} \\
 X_{4d} &=& s_C(d) \cdot X_{5d} \\
 X_{5d} &=& c_D(0) \cdot Y_1 + c_D(\perp) \cdot X_{6d} \\
 X_{6d} &=& (c_B(d, 0) + c_B(\perp)) \cdot X_{5d} \\
 Y_1 &=& \sum_{d' \in \Delta} r_A(d') \cdot Y_{2d'} \\
 Y_{2d} &=& c_B(d, 1) \cdot Y_{4d} + c_B(\perp) \cdot Y_{3d} \\
 Y_{3d} &=& (c_D(0) + c_D(\perp)) \cdot Y_{2d} \\
 Y_{4d} &=& s_C(d) \cdot Y_{5d} \\
 Y_{5d} &=& c_D(1) \cdot X_1 + c_D(\perp) \cdot Y_{6d} \\
 Y_{6d} &=& (c_B(d, 1) + c_B(\perp)) \cdot Y_{5d} \\
 | d \in \Delta \}
 \end{array}
 \right.$$

Durch die Anwendung von τ_I auf $\langle X_1 | E \rangle$ werden die Kommunikationsschleifen zu τ -Schleifen, die durch CFAR eliminiert werden.

Beispielsweise bilden X_{2d} und X_{3d} eine τ -Gruppe I mit Ausgang $c_B(d, 0) \cdot X_{4d}$, also:

$$\begin{aligned} & r_A(d) \cdot \tau_I(\langle X_{2d} | E \rangle) \\ = & r_A(d) \cdot \tau_I(c_B(d, 0) \cdot \langle X_{4d} | E \rangle) \\ = & r_A(d) \cdot \tau \cdot \tau_I(\langle X_{4d} | E \rangle) \\ = & r_A(d) \cdot \tau_I(\langle X_{4d} | E \rangle) \end{aligned}$$

Entsprechend:

$$\begin{aligned} s_C(d) \cdot \tau_I(\langle X_{5d} | E \rangle) &= s_C(d) \cdot \tau_I(\langle Y_1 | E \rangle) \\ r_A(d) \cdot \tau_I(\langle Y_{2d} | E \rangle) &= r_A(d) \cdot \tau_I(\langle Y_{4d} | E \rangle) \\ s_C(d) \cdot \tau_I(\langle Y_{5d} | E \rangle) &= s_C(d) \cdot \tau_I(\langle X_1 | E \rangle) \\ \tau_I(\langle X_1 | E \rangle) &= \sum_{d \in \Delta} r_A(d) \cdot \tau_I(\langle X_{2d} | E \rangle) \\ &= \sum_{d \in \Delta} r_A(d) \cdot \tau_I(\langle X_{4d} | E \rangle) \\ &= \sum_{d \in \Delta} r_A(d) \cdot s_C(d) \cdot \tau_I(\langle X_{5d} | E \rangle) \\ &= \sum_{d \in \Delta} r_A(d) \cdot s_C(d) \cdot \tau_I(\langle Y_1 | E \rangle) \\ \tau_I(\langle Y_1 | E \rangle) &= \sum_{d \in \Delta} r_A(d) \cdot s_C(d) \cdot \tau_I(\langle X_1 | E \rangle) \end{aligned}$$

Mit RSP folgt $\tau_I(\langle X_1 | E \rangle) = \langle Z | Z = r_A(d) \cdot s_C(d) \cdot Z \rangle$.

Damit ist das oben angesprochene Ziel des Beweises erreicht:

$$\tau_I(\partial_H(R_0 \| S_0)) = \sum_{d \in \Delta} r_A(d) \cdot s_C(d) \cdot \tau_I(\partial_H(R_0 \| S_0))$$

Aufgabe 9.61

- a) Ersetzen Sie im Beweis des Alternierbitprotokolls die Spezifikation von U_{db} durch

$$U_{db} = (r_D(b) + r_D(\perp)) \cdot S_{1-b} + r_D(1-b) \cdot T_{db}.$$

Interpretieren Sie dies inhaltlich und formal für den Beweis.

- b) Modellieren Sie im Modell des Alternierbitprotokolls die (gestörten) Kanäle als eigene Funktionseinheiten K und L , an die - im Gegensatz zur behandelten Form - die Daten ungestört übergeben werden. Formulieren Sie die Spezifikation des geänderten Modells.

Anhang: Übersicht über die Axiome der Prozesskalküle

Zur Übersicht alle Axiome und daraus abeleitete Regeln mit Seitenreferenz². Wie üblich sei $v \in A$ und x, y, z seien Terme.

Bez.	Axiom	BPA	PAP	ACP	Rek.	ACP τ	Skript
A1	$x + y = y + x$	x	x	x		x	S. 194
A2	$(x + y) + z = x + (y + z)$	x	x	x		x	S. 194
A3	$x + x = x$	x	x	x		x	S. 194
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$	x	x	x		x	S. 194
A5	$(x \cdot y) \cdot z = x(y \cdot z)$	x	x	x		x	S. 194
M1	$x \parallel y = (x \parallel y + y \parallel x) + x y$		x	x		x	S. 201
LM2	$v \parallel y = v \cdot y$		x	x		x	S. 201
LM3	$(v \cdot x) \parallel y = v \cdot (x \parallel y)$		x	x		x	S. 201
LM4	$(x + y) \parallel z = x \parallel z + y \parallel z$		x	x		x	S. 201
CM5	$v w = \gamma(v, w)$		x	x		x	S. 201
CM6	$v (w \cdot y) = \gamma(v, w) \cdot y$		x	x		x	S. 201
CM7	$(v \cdot x) w = \gamma(v, w) \cdot x$		x	x		x	S. 201
CM8	$(v \cdot x) (w \cdot y) = \gamma(v, w)(x \parallel y)$		x	x		x	S. 201
CM9	$(x + y) z = x z + y z$		x	x		x	S. 201
CM10	$x (y + z) = x y + x z$		x	x		x	S. 201
A6	$x + \delta = x$			x		x	S. 203
A7	$\delta x = \delta$			x		x	S. 203
D1	$\partial_H(v) = v, v \notin H$			x		x	S. 203
D2	$\partial_H(v) = \delta, v \in H$			x		x	S. 203
D3	$\partial_H(\delta) = \delta$			x		x	S. 203
D4	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$			x		x	S. 203
D5	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$			x		x	S. 203
LM11	$\delta \parallel x = \delta$			x		x	S. 203
CM12	$\delta \parallel x = \delta$			x		x	S. 203
CM13	$x \delta = \delta$			x		x	S. 203
RDP	$\langle X_i E \rangle = t_i(\langle X_1 E \rangle, \dots, \langle X_n E \rangle)$				x	x	S. 209
RSP	$\langle X_i E \rangle = y_i, \text{ falls für alle } i \text{ gilt:}$ $y_i = t_i(y_1, \dots, y_n)$				x	x	S. 209
B1	$v \cdot \tau = v$					x	S. 216
B2	$v \cdot (x + y) = v \cdot (\tau \cdot (x + y) + x)$					x	S. 216
TI1	$\tau_I(v) = v, v \notin I$					x	S. 216
TI2	$\tau_I(v) = \tau, v \in I$					x	S. 216
TI3	$\tau_I(\delta) = \delta$					x	S. 216
TI4	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$					x	S. 216
TI5	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$					x	S. 216

²Dank an Patrick Fey

10 Parallele Algorithmen

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 und haben eine synchrone Ablaufsemantik. Charakteristisch für die in späteren Vorlesungen¹ 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. 10.1). Bei Speichersynchronisation wird eine SIMD-Architektur (*single instruction multiple data*) (Abb. 10.1) benutzt, deren Formalisierung das PRAM-Modell (parallel random-access machine) ist.

Die Definition der PRAM erweitert das Konzept des Einprozessormodells der RAM (*random-access machine*). Für die RAM gelten Komplexitätsmaße wie für die Turingmaschine: Zeit- und Speicherkomplexität. Für das parallele Modell der PRAM kommen noch die Prozessor- und die Operationenkomplexität hinzu.

Definition 10.1 Komplexitätsmaße für einen Algorithmus A sind die

- Zeitkomplexität: maximale Anzahl $T_A(n)$ der synchronen Schritte aller Prozessoren bis zur Termination, wobei das Maximum über alle Eingaben („Probleminstanzen“) der Größe n gebildet wird,
- Speicherkomplexität: maximale Anzahl $S_A(n)$ der im gemeinsamen Speicher und den lokalen Speichern der Prozessoren bis zur Termination belegten Speicherzellen, wobei das Maximum über alle Eingaben („Probleminstanzen“) der Größe n gebildet wird,
- Prozessorkomplexität: maximale Anzahl $P_A(n)$ der bis zur Termination aktiv gewordenen Prozessoren, wobei das Maximum über alle Eingaben („Probleminstanzen“) der Größe n gebildet wird,
- Operationenkomplexität: maximale Anzahl $W_A(n)$ der Operationen aller Prozessoren bis zur Termination, wobei das Maximum über alle Eingaben („Probleminstanzen“) der Größe n gebildet wird, (d.h. parallele Operationen werden einzeln gezählt, „W“ für „work“).

Für die RAM ist nach dieser Definition also $P_A(n) = 1$ und $W_A(n) = T_A(n)$.

Sei $\Theta(T_{\mathcal{P}}^*(n))$ 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

¹Bachelor/Master Modul: Intelligente kooperierende Dienste (KD)

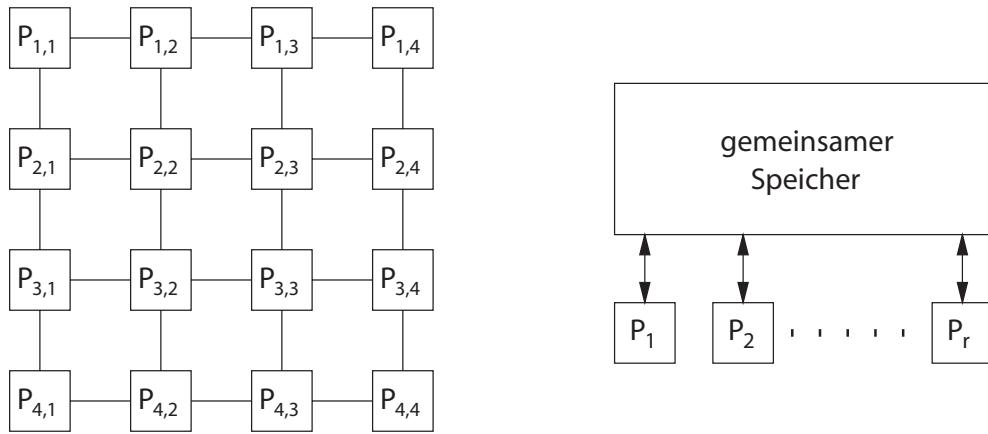


Abbildung 10.1: Prozessorkonfigurationen: 2-dimensionales Feld und PRAM

Algorithmen für \mathcal{P} genommen, wenn Letzteres nicht bekannt ist). Ein paralleler Algorithmus A heißt *optimal* für ein Problem \mathcal{P} , wenn $W_A(n) \in \Theta(T_{\mathcal{P}}^*(n))$ (äquivalente Notation: $W_A(n) = \Theta(T_{\mathcal{P}}^*(n))$). In anderen Worten: die Gesamtzahl der von A ausgeführten Operationen ist asymptotisch gleich der sequentiellen Zeitkomplexität $T_{\mathcal{P}}^*(n)$ des Problems - unabhängig von der (parallelen) Zeitkomplexität $T_A(n)$ von A .

10.1 Random-Access-Maschine (RAM)²

Turing-Maschinen sind ausgezeichnete Modelle von universellen Rechnern, um die grundlegenden Probleme der Berechenbarkeit, Beschreibbarkeit und Komplexitätstheorie zu untersuchen.

In diesem Kapitel betrachten wir andere grundlegende Modelle von universellen sequentiellen und parallelen Rechnern, die entweder wichtig sind, um die Hauptprobleme des Entwurfs und der Analyse von Algorithmen zu untersuchen, oder wichtig sind zur Untersuchung von grundlegenden Problemen der Parallelität.

Die *Random-Access-Maschine (RAM)* ist ein einfaches von Neumann-Modell sequentieller Rechner, das gleichmäßig zur Turing-Maschine ist. Es dient allgemein zum Entwurf und zur Analyse von Algorithmen für sequentielle Rechner und wird hier zur Vorbereitung einer Erweiterung zu einem parallelen Modell behandelt.

Die *parallele Random-Access-Maschine (PRAM)* ist das wichtigste Modell für den Entwurf und die Analyse von parallelen Algorithmen. Zellulare Automaten, Schaltkreise und Neuronale Netze repräsentieren weitere grundlegende Modelle von parallelen Computern, die dazu dienen, die grundlegenden Probleme der parallelen Rechner zu untersuchen und zu illustrieren.

10.1.1 Definition der RAM

Das Turing-Maschinen Modell ist ausreichend, um Fragen der Berechenbarkeit zu untersuchen. Eine TM hat jedoch mit einem modernen Computer wenig gemeinsam.

Das wichtigste Beispiel eines Modells von realen sequentiellen Computern sind **Registermaschinen**. Diese Rechnermodelle besitzen einen Speicher ähnlich dem eines modernen Mikroprozessors. Der **Speicher** wie auch das **Eingabeband** bestehen jeweils aus einer Folge von Registern, die nicht nur einzelne Symbole, sondern auch ganze Zahlen beliebiger Größe speichern können. Ein **Programm** für eine Registermaschine ist mit einem in einer Assembler- oder höheren Programmiersprache geschriebenen Programm vergleichbar. Wir betrachten zwei Modelle von Registermaschinen: RAM und RASP.

Beim Modell der RAM unterscheiden wir zudem noch zwei Varianten: Die sogenannte *Bit-RAM*, bei der die Register beliebige natürliche (ganze) Zahlen enthalten dürfen. Bei der Modell-Variante der sogenannten *arithmetischen RAM* können die Speicherinhalte aus einem beliebigen Körper, wie z.B. \mathbb{R} oder \mathbb{Q} stammen. Weitere Modellvarianten entstehen bei Einschränkung des erlaubten Befehlssatzes.

Das Eingabeband einer RAM besteht aus einer abzählbaren Folge von Registern x_1, x_2, \dots .

Der Speicher einer RAM besteht aus einer abzählbaren Folge von Registern R_0, R_1, \dots . Der Index i eines Registers dient als **Adresse**. Register R_0 dient als **Akkumulator**.

²Entnommen dem Skript *Automaten und Komplexität (AUK)* von M. Jantzen und diesem Skript angepasst.

Die aktuelle Konfiguration einer RAM lässt sich eindeutig aus dem Inhalt der Datenspeicher und dem Befehlszähler IC bestimmen. Am Anfang einer Berechnung sind alle Register mit 0 initialisiert.

Eine RAM wird durch ein Programm spezifiziert. Dies ist eine endliche, fortlaufend nummerierte Folge b_0, b_1, \dots von Befehlen aus einer vorgegebenen Befehlsmenge \mathcal{B} .

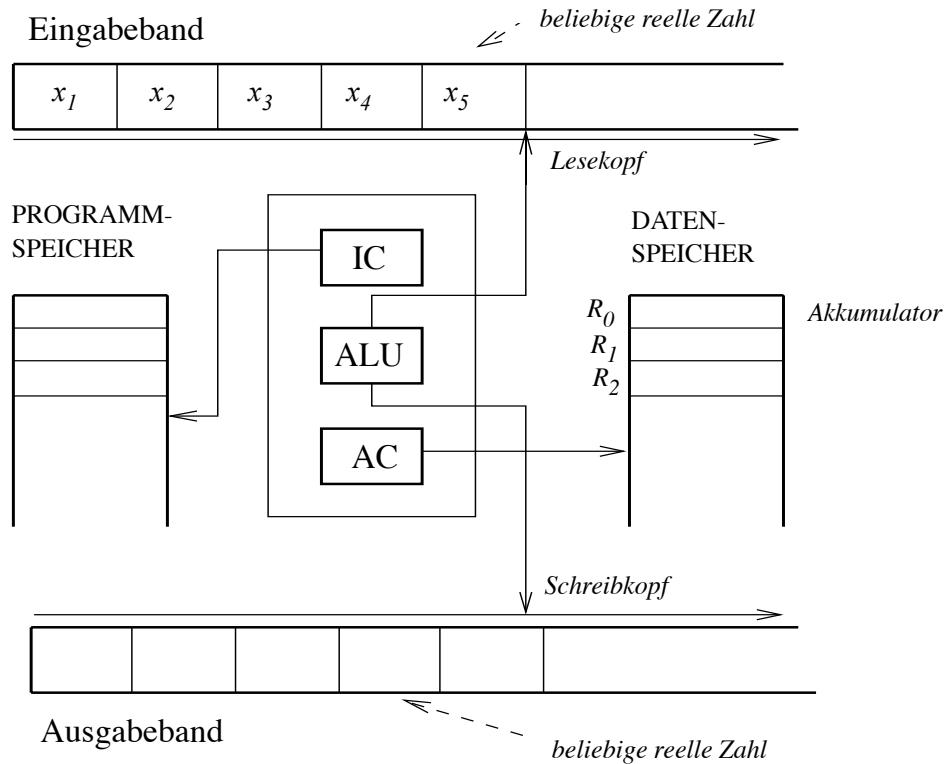


Abbildung 10.2: RAM – Random Access Machine

Operationen		Operand
READ	op	= i – die Zahl i
WRITE	op	i – Inhalt des
LOAD	op	Registers R_i
STORE	op	* i – indirekte Adresse
ADD	op	(Inhalt des Registers R_j , wobei j
SUB	op	der Inhalt des Registers R_i ist)
JUMP	label	
JZERO	label	
JGZERO	label	

Die Befehlsmenge \mathcal{B} einer RAM umfasst die vorstehend aufgelisteten Grundoperationen. Die exakte Natur der Befehle ist nicht wichtig, solange die Befehle ähnlich den Befehlen

von realen Computern sind. Es gibt hier **I/O–Operationen**, **arithmetische Operationen** und **JUMP–Operationen**. Außer JUMP– und I/O–Operationen werden alle mit Hilfe des Registers R_0 (Akkumulator) ausgeführt. Die folgende Tabelle 10.1 zeigt, wie sich die Konfiguration der Maschine durch Ausführung eines Befehls ändert.

Tabelle 10.1: Die Befehle einer RAM

Instruktion	Bedeutung
1. LOAD a	$c(0) \leftarrow v(a)$
2. STORE i	$c(i) \leftarrow c(0)$
STORE $*i$	$c(c(i)) \leftarrow c(0)$
3. ADD a	$c(0) \leftarrow c(0) + v(a)$
4. SUB a	$c(0) \leftarrow c(0) - v(a)$
5. MULT a	$c(0) \leftarrow c(0) \times v(a)$
6. DIV a	$c(0) \leftarrow \lfloor c(0) \div v(a) \rfloor$
7. READ i	$c(i) \leftarrow$ derzeitiges Eingabesymbol
READ $*i$	$c(c(i)) \leftarrow$ derzeitiges Eingabesymbol. Der Kopf auf dem Eingabeband geht in beiden Fällen um ein Feld nach rechts.
8. WRITE a	$v(a)$ wird in das Feld auf dem Ausgabeband geschrieben über dem sich gerade der Bandkopf befindet. Danach wird der Kopf um ein Feld nach rechts bewegt.
9. JUMP b	Der <i>location counter</i> (Befehlsregister) wird auf die mit b markierte Anweisung gesetzt.
10. JGTZ b	Der <i>location counter</i> wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) > 0$ ist, sonst auf die nachfolgende Anweisung.
11. JZERO b	Der <i>location counter</i> wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) = 0$ ist, sonst auf die nachfolgende Anweisung.
12. HALT	Programmausführung beenden.

$$\begin{array}{lll}
 \text{c} - \text{Speicherabbildung} & \xrightarrow{v(a) - \text{Der Wert von } a} & \begin{array}{l} v(=i) = i \\ v(i) = c(i) \\ v(*i) = c(c(i)) \end{array} \\
 c(i) \text{ Der Inhalt des Registers } R_i. & &
 \end{array}$$

Die Befehle des Programms werden in sequentieller Ordnung ausgeführt, bis eine Halte- oder JUMP-Instruktion vorkommt.

Im allgemeinen definiert ein RAM-Programm eine partielle Abbildung vom Eingabeband auf das Ausgabeband. Zwei Fälle sind wichtig:

1. Berechnung von Funktionen:

Wenn ein RAM-Programm \mathbf{P} vom Eingabeband stets n Zahlen x_1, \dots, x_n liest, danach berechnet und danach stets m Zahlen y_1, \dots, y_m auf das Ausgabeband schreibt, dann sagt man, dass \mathbf{P} eine Funktion $f : \mathbb{N}^n \longrightarrow \mathbb{N}^m$ berechnet.

Es ist nicht schwer zu zeigen, dass RAM-Programme für Bit-RAM's jede partiell rekursiven Funktion, und nur diese, berechnen können.

Beispiel 10.2 In Abb. 10.3 gibt es ein RAM-Programm, um die Funktion $f(n) = n^n$ zu berechnen. Eine Beschreibung desselben Programms in einer höheren Programmiersprache ist in Abb. 10.4 zu sehen. Aus Abb. 10.3 kann man auch erkennen, wie man die Befehle der höheren Programmiersprache in RAM-Instruktionen übersetzen kann.

READ	1	read r1		
LOAD	1	}		
JGTZ	pos		if $r1 \leq 0$ then write 0	
WRITE	=0			
JUMP	endif			
pos:	LOAD	1	}	$r2 \leftarrow r1$
	STORE	2		
	LOAD	1	}	$r3 \leftarrow r1 - 1$
	SUB	=1		
	STORE	3		
while:	LOAD	3	}	while $r3 > 0$ do
	JGTZ	continue		
	JUMP	endwhile		
continue:	LOAD	2	}	$r2 \leftarrow r2 * r1$
	MULT	1		
	STORE	2	}	$r3 \leftarrow r3 - 1$
	LOAD	3		
	SUB	=1		
	STORE	3		
JUMP	while	}		
endwhile:	WRITE			2
endif:	HALT			

Abbildung 10.3: $f(n) = n^n$ (RAM-Programm)

2. Akzeptieren von Sprachen:

Ein RAM-Programm kann man auch als einen Akzeptor einer Sprache interpretieren.

Sei $\Sigma = \{a_1, \dots, a_m\}$ ein Alphabet. (Das Symbol $a_i, 1 \leq i \leq m$ wird durch die natürliche Zahl i kodiert (Schreibweise: $a_i^{(k)} = i$)).

Ein RAM-Programm \mathbf{P} akzeptiert eine Sprache $\mathcal{L} \subseteq \Sigma^*$ folgenderweise: Das Eingabewort $w = w_1 \dots w_k$ wird in den Feldern des Eingabebandes gespeichert, w_i wie die Zahl $w_i^{(k)}$ im i -ten Feld und im $(k+1)$ -ten Feld wird 0 (Markierung des Ende des Eingabewortes) gespeichert.

```

BEGIN
  READ r1;
  IF r1 ≤ 0 THEN WRITE 0
  ELSE
    BEGIN
      r2 ← r1;
      r3 ← r1 - 1;
      WHILE r3 > 0 DO
        BEGIN
          r2 ← r2 * r1;
          r3 ← r3 - 1
        END;
      WRITE r2
    END
  END

```

Abbildung 10.4: $f(n) = n^n$ (Hochsprache)

Dies Eingabewort wird durch ein Programm \mathbf{P} akzeptiert, wenn \mathbf{P} zum Schluss der Berechnung 1 in das erste Feld des Ausgabebandes schreibt und hält. Die Sprache, die \mathbf{P} akzeptiert, ist die Menge der Wörter, die \mathbf{P} akzeptiert.

Es ist nicht schwer zu zeigen, dass RAM-Programme für Bit-RAM's genau die rekursiv aufzählbaren Sprachen akzeptieren.

Beispiel 10.3 In Abbildung 10.6 und 10.5 ist ein Hochsprache-Programm und ein RAM-Programm abgebildet, um die Sprache $\mathcal{L}_0 \subsetneq \{1, 2\}^*$ zu akzeptieren, deren Wörter dieselbe Anzahl von Einsen und Zweien besitzt.

10.1.2 Komplexitätsmaße für die RAM

Die parallelen Algorithmen am Ende dieses Kapitels werden in einem Pseudokode formuliert, für den die in Definition 10.1 angegebenen Definitionen ausreichend sind. Das Modell der RAM ist jedoch genauer und hat präziser formulerte Definitionen für Komplexitätsmaße. Beispielsweise wird der in Definition 10.1 benutzte Begriff „Größe“ genauer gefasst.

Es gibt zwei Arten von Komplexitätsmaßen für RAMs.

Uniforme Komplexitätsmaße:

Uniformes Zeitmaß: 1 Schritt = 1 Zeiteinheit	Uniformes Platzmaß: 1 Register = 1 Platzeinheit
---	--

Diese Maße sind einfach, aber wenn eine RAM sehr lange Zahlen verarbeitet, sind diese Maße weniger geeignet. Daher wird oft für RAMs das *logarithmische Komplexitätsmaß* verwendet. Dieser Wert ergibt sich als Summe der logarithmischen Kosten der Einzelschritte bzw. Register. Um diese Kosten zu bestimmen, benutzen wir die folgende

```

        LOAD   =0      }   d ← 0
        STORE 2       }
        READ   1       }   read x
while:   LOAD   1       }   while x ≠ 0 do
        JZERO endwhile } 
        LOAD   1       }   if x ≠ 1
        SUB   =1       }
        JZERO one      }
        LOAD   2       }   then d ← d - 1
        SUB   =1       }
        STORE 2       }
        JUMP  endif   }
one:     LOAD   2       }   else d ← d + 1
        ADD   =1       }
        STORE 2       }
endif:   READ   1       }
        JUMP  while   }
endwhile: LOAD   2       }   if d = 0 then write 1
        JZERO output  }
        HALT
output:  WRITE =1       }
        HALT
    
```

 Abbildung 10.5: Erkennung von $\mathcal{L}_0 \not\subseteq \{1, 2\}^*$ (RAM)

Funktion $l : \mathbb{N} \rightarrow \mathbb{N}$, die die Länge der Binärdarstellungen der Zahlen bestimmt:

$$l(i) = \begin{cases} \lfloor \log i \rfloor + 1 & i \neq 0 \\ 1 & i = 0 \end{cases}$$

Die Kosten eines einzelnen Schrittes (RAM-Instruktion) sind in Tabelle 10.2 (Seite 235) aufgelistet und hängen von der Länge der Operanden ab:

Operand a	Kosten t(a)
=i	$l(i)$
i	$l(i) + l(c(i))$
*i	$l(i) + l(c(i)) + l(c(c(i)))$

Definition 10.4 Die uniforme (logarithmische) Zeitkomplexität eines RAM-Programms A ist die maximale Anzahl $T_A(n)$ der Summen der uniformen (logarithmischen) Kosten aller Schritte des Programms bis zur Termination, wobei das Maximum über alle Eingaben der uniformen (logarithmischen) Größe n gebildet wird.

Die uniforme (logarithmische) Platzkomplexität eines RAM-Programms A ist die maximale Summe $S_A(n)$ der uniformen (logarithmischen) Kosten aller Register, die dieses Programm bis zur Termination addressiert, wobei das Maximum über alle Eingaben der uniformen (logarithmischen) Größe n gebildet wird.

```

BEGIN
    d ← 0;
    READ x;
    WHILE x ≠ 0 DO
        BEGIN
            IF x ≠ 1 THEN d ← d - 1 ELSE d ← d + 1;
            READ x
        END;
    IF d = 0 THEN WRITE 1
END
    
```

Abbildung 10.6: Erkennung von $\mathcal{L}_0 \subseteq \{1, 2\}^*$ (Hochsprache)

Tabelle 10.2: Kosten einer RAM-Instruktion

1.	LOAD a	$t(a)$
2.	STORE i	$l(c(0)) + l(i)$
	STORE $*i$	$l(c(0)) + l(i) + l(c(i))$
3.	ADD a	$l(c(0)) + t(a)$
4.	SUB a	$l(c(0)) + t(a)$
5.	MULT a	$l(c(0)) + t(a)$
6.	DIV a	$l(c(0)) + t(a)$
7.	READ i	$l(\text{input}) + l(i)$
	READ $*i$	$l(\text{input}) + l(i) + l(c(i))$
8.	WRITE a	$t(a)$
9.	JUMP b	1
10.	JGTZ b	$l(c(0))$
11.	JZERO b	$l(c(0))$
12.	HALT	1

Das logarithmische Platzmaß eines Registers (während eines Programmablaufs) ist die maximale Länge $l(i)$ aller Zahlen i , die in diesem Register gespeichert wurden (während des Programmablaufs).

Beispiel 10.5 Betrachten wir das RAM-Programm A , das $f(n) = n^n$ berechnet (siehe Abb. 10.4 und 10.3). Es ist offensichtlich, dass für die uniformen Komplexitätsmaße die Zeitkomplexität $T_A(n) = O(n)$ und die Platzkomplexität $T_A(n) = O(1)$ ist. Die Zeitkomplexität wird durch die `while`-loop und die `MULT`-Instruktion dominiert. Diese Operation wird n -mal ausgeführt.

Bevor die `MULT`-Instruktion das i -te mal ausgeführt wird, enthält der Akkumulator n^i und das Register 1 enthält n .

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

Die logarithmischen Kosten der i -ten MULT–Operation sind deshalb $l(n^i) + l(n) \approx (i+1) \log n$ und die Gesamtkosten aller MULT–Operationen sind

$$\sum_{i=1}^{n-1} (i+1) \log n = O(n^2 \log n).$$

Beispiel 10.6 Die folgende Tabelle 10.3 zeigt die Komplexitätsmaße für das Programm, das die Sprache \mathcal{L}_0 akzeptiert.

Tabelle 10.3: Programm–Komplexitätsmaße für \mathcal{L}_0

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n \log n)$
Platzkomplexität	$O(1)$	$O(\log n)$

In vielen Fällen ist es notwendig, die logarithmischen Komplexitätsmaße zu benutzen, um realistische Ergebnisse zu erhalten.

Z.B. wissen wir schon, dass jede Turing–Maschine durch einen Counterautomaten mit 2 Countern simuliert werden kann (in denen sehr große Zahlen gespeichert sind). d. h. jede partiell rekursive Funktion kann durch ein RAM–Programm berechnet werden, dessen Platzkomplexität $O(1)$ für das uniforme Platzkomplexitätsmaß ist! (In diesem Fall ist die uniforme Platzkomplexität offensichtlich nicht realistisch. Die logarithmische Platzkomplexität liefert viel realistischere Ergebnisse.)

Manchmal bezeichnet man die RAM mit unserer Menge von Befehlen als RAM_* und die RAM ohne die Operation *MULT* und *DIV* als RAM_+ . Sind die in den Registern speicherbaren Zahlen stets natürliche Zahlen, so spricht man von einer *Bit-RAM*. Können rationale oder reelle Zahlen benutzt werden, so spricht man von einer *arithmetischen RAM*.

Die MULT–Operation ist sehr mächtig. Z.B. kann man mit n MULT–Operationen die Funktion $f(n) = n^{2^n}$ berechnen. Die Zeitkomplexität solcher Folge von MULT–Operationen ist $O(n)$ bezüglich des uniformen Komplexitätsmaßes und $O(2^n)$ – exponentiell mehr – bzgl. des logarithmischen Zeitkomplexitätsmaßes.

Es wird später für die RAM_+ gezeigt, dass auch das uniforme Zeitkomplexitätsmaß Ergebnisse produziert, die nicht sehr schlecht sind.

RAMs sind geeignete Modelle, um die Komplexität von algebraischen und kombinatorischen Optimierungs–Problemen zu untersuchen.

10.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine

RAM und TM sind zwei sehr verschiedene Modelle. Wie schnell können sie einander simulieren?

Satz 10.7 Eine $T(n)$ -zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$ -zeitbeschränkt ist und die im logarithmischen Maß $O(T(n) \log T(n))$ -zeitbeschränkt ist.

Beweis: M habe k einseitig unendliche Bänder. Die simulierende RAM_+ (in der Abbildung 10.7 mit R bezeichnet) verhält sich auf dem Ein- und Ausgabeband genau wie M . Die Darstellung der Beschriftung der Arbeitsbänder der TM geschieht folgendermaßen: R_0 speichert den Zustand von M , R_j , $1 \leq j \leq k$, die Position p_j des Kopfes auf Band j und R_{kp+j} das Symbol a_j der Zelle p_j des j -ten Bandes, kodiert als eine natürliche Zahl.

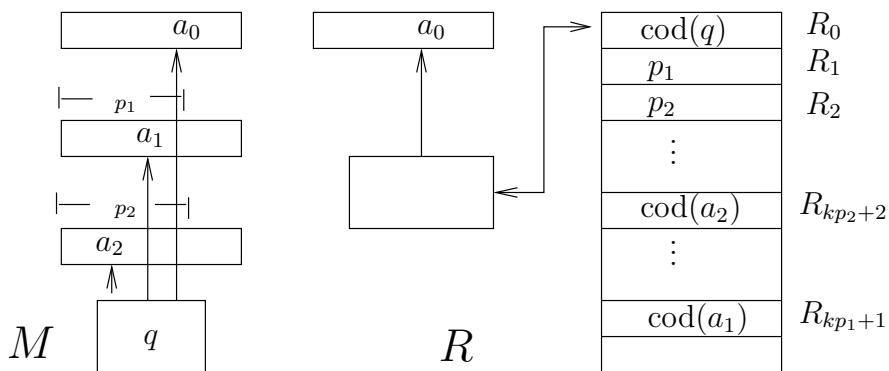


Abbildung 10.7: Zur Simulation einer TM durch eine RAM

Für jedes Tupel

$$(q, a_0, a_1, \dots, a_k)$$

aus einem Zustand q , einem Eingabesymbol a_0 und von den Arbeitsköpfen gelesenen Zeichen a_i hat R ein kleines Programmstück, das den Übergang

$$\delta(q, a_0, a_1, \dots, a_k) = (q', a'_1, \dots, a'_{k'}, q_1, \dots, q_{k'}, a'_{k'+1})$$

simuliert.

(Er speichert q' in R_0 , a'_i in R_{kp_i+i} , $p_j + q_j$ in R_j . Gleichzeitig wird $a'_{k'+1}$ gedruckt.)

Um einen Schritt der TM zu simulieren, benötigt die RAM eine konstante Anzahl elementarer Schritte, um den Übergang zu simulieren. Die logarithmischen Kosten eines Befehls lassen sich abschätzen durch die maximal möglichen Werte für die p_j , und diese

Kosten sind durch $O(\log T(n))$ beschränkt. Damit lässt sich die logarithmische Zeitkomplexität durch

$$O(T(n) \log T(n))$$

abschätzen. □

Der Beweis stammt aus [HMU79]. Einen etwas ausführlicheren finden die Leser(innen) bei [Pau78]. Sehr detaillierte Beweise finden sich in [Rei90].

Satz 10.8 Eine im logarithmischen Maß $T(n)$ -zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$ -zeitbeschränkte 5-Band DTM M simuliert werden.

Beweis: M speichert den Inhalt der Register von R auf dem ersten Band als

#	#	i_1	#	c_1	#	#	i_2	#	c_2	#	#	#	\dots	i_k	#	c_k	#	#	b	\dots
---	---	-------	---	-------	---	---	-------	---	-------	---	---	---	---------	-------	---	-------	---	---	---	---------

wobei i_1, i_2, \dots, i_k die Adressen der bislang benutzten Register sind und die Zeichenketten c_j den Inhalt des entsprechenden Registers R_{i_j} kodieren. # ist ein Trennsymbol. Der Inhalt des RAM-Akkumulators ist auf dem zweiten Band gespeichert. Zwei Bänder werden benutzt, um das Ein- und Ausgabeband von R zu simulieren. Das 5te Band ist das Arbeitsband.

Für jeden Befehl von R besitzt M eine Menge von Zuständen, die diesen Befehl simulieren. Wir betrachten die Befehle ADD *25 und STORE 32:

Simulation von ADD *25:

1. M sucht auf dem ersten Band nach dem Register 25, d. h. nach dem String ##11001. Falls gefunden, kopiert M den Inhalt dieses Registers auf Band 5. Falls nicht gefunden, hält M .
2. M sucht auf dem ersten Band nach dem Register, dessen Index auf dem fünften Band gespeichert ist. Falls gefunden, kopiert M den Inhalt dieses Registers auf Band 5. Wenn nicht, schreibt M 0 auf Band 5.
3. M addiert die Zahlen auf dem zweiten und fünften Band und schreibt das Resultat auf Band 2.

Simulation von STORE 32:

1. M sucht auf dem ersten Band nach Register 32 (d. h. nach ##100000#).
2. Falls gefunden, kopiert M alles von dem ersten Band, was rechts von ##100000# steht, außer den Inhalt des Register 32, auf das 5te Band. dann kopiert M die Zahl auf dem zweiten Band auf das erste, gleich rechts von ##100000#. Danach kopiert M den Inhalt des fünften Bandes auf das erste Band, ganz rechts.
3. Falls es kein Register 32 auf dem ersten Band gibt, geht M zum letzten und schreibt dann ##100000# und dahinter den Inhalt von Band 2.

Zur Zeitkomplexität: Die Länge des Wortes auf dem ersten Band ist höchstens $T(n)$ – die logarithmische Zeitkomplexität von R . Die Zeit, die M braucht, um einen Befehl von R zu simulieren ist auch $O(T(n))$ und deshalb ist die gesamte Zeit der Simulation $O(T^2(n))$. □

Man kann auch ein allgemeines Resultat zeigen: Jede im logarithmischen Maß $T(n)$ -zeitbeschränkte RAM_* (d. h. auch mit Multiplikation und Division) kann durch eine $O(T^2(n))$ zeitbeschränkte DTM simuliert werden.

Man kann für alle unsere Modelle von Turing-Maschinen, für RAMs mit logarithmischen Zeitmaß, und für RAM_+ mit uniformen Zeitmaß auch zeigen, dass jede von diesen Maschinen jede andere mit „*polynomiellem Zeitverlust und konstantem Platzverlust*“ simuliert.

Diese Resultate schaffen die Grundlage für folgende These auf der die moderene Komplexitätstheorie begründet ist:

Invariance Thesis: *There exists a standard class of machine models, which includes among others all variants of Turing Machines, all variants of RAM's and RASP's with logarithmic time measures, and also the RAM's and RASP's in the uniform time measure and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomially bounded overhead in time, and constant factor overhead in space.*

Die Maschinen-Modelle, die der „Invariance Thesis“ genügen, bilden die *erste Maschinenklasse*.

Die Resultate aus diesem und vorhergehendem Kapitel implizieren auch, dass die folgenden zwei Komplexitätsklassen, die sehr wichtig sind, nicht von dem einzelnen Modell aus der ersten Maschinenklasse abhängen:

\mathcal{P}

die Klasse der Sprachen (algorithmischen Probleme), die man in Polynomialzeit erkennen (lösen) kann und

\mathcal{PSPACE}

die Klasse der Sprachen (algorithmischen Probleme), die man in polynomiellen Platz erkennen (lösen) kann.

Sei POL die Klasse aller Polynome. Für eine Maschinenklasse \mathcal{M} sei $\mathcal{M}\text{-Time}(POL)$ die Menge aller Sprachen, die die polynomialzeit-beschränkten Maschinen von \mathcal{M} erkennen können.

Nach den vorangegangenen Resultaten ergibt sich:

$$\begin{aligned} \text{RAM}_+ \text{-Time}(POL) &= \text{RAM-Time}_{\log}(POL) = \mathcal{P} \\ &= \text{DTM-Time}(POL) = \text{DTM}_1 \text{-Time}(POL) \end{aligned}$$

Die RAM ist ein Modell eines sequentiellen Rechners, das sehr einfach, aber auch sehr wichtig ist. Um die Probleme und Methoden des Entwurfs und der Analyse von Algorithmen für moderne sequentielle Rechner zu untersuchen, genügt es praktisch, das RAM-Modell zu benutzen. Natürliche Frage: Gibt es natürliche Modifikationen des RAM-Modells, die wichtige Modelle von parallelen Rechnern liefern?

Die Antwort ist positiv. Ein solches Modell, die PRAM, behandeln wir im nächsten Abschnitt.

10.2 Parallele Random-Access-Maschine (PRAM)³

Es gibt zwei grundlegende Modelle paralleler Computer mit globalem Speicher:

MIMD-Computer (*Multiple Instructions Multiple Data*) – jeder Prozessor führt ein spezielles (eigenes) Programm aus

SIMD-Computer (*Single Instructions Multiple Data*) – in jedem Schritt ist die *Instruktionsart* für alle Prozessoren einheitlich.

PRAMs sind heutzutage das Hauptmodell paralleler Rechner für Entwurf und Analyse paralleler Algorithmen. Dafür gibt es drei Gründe:

1. PRAM abstrahieren von Ebenen algorithmischer Komplexität, die Synchronisation, Zuverlässigkeit, Lokalität von Daten, Netztopologie und Kommunikation betreffen, und erlaubt so den Entwerfern von Algorithmen, sich auf die grundlegenden Berechnungsschwierigkeiten zu konzentrieren.
2. Viele der Entwurfs-Paradigmen haben sich als bestechend robust herausgestellt; sie passen auch zu Modellen außerhalb des PRAM-Bereichs.
3. Neuere Ergebnisse haben gezeigt, dass PRAMs effizient auf high-interconnection networks emuliert werden können.

MIMD- und SIMD-Computer mit globalem Speicher können einander leicht und schnell simulieren, wenn man nur einen geeigneten Formalismus verwendet. Für Beispiele benutzen wir beide Modelle, um Komplexitätsresultate festzuhalten, verwenden wir jedoch das folgende formale Modell.

10.2.1 Definition der PRAM

Definition 10.9 (PRAM) Eine parallele Registermaschine (PRAM) ist eine (endlose) Folge von Prozessoren P_1, P_2, \dots . Der Index i von P_i dient zur Identifikation und wird als PID (Prozessor-ID) bezeichnet. Prozessoren sind RAMs. P_i besitzt einen lokalen Speicher R_i , bestehend aus Registern $R_{i,0}, R_{i,1}, \dots$. Auf R_i kann ausschließlich P_i zugreifen. Die Kommunikation zwischen den Prozessoren geschieht über einen globalen Speicher $\mathcal{M} : M_0, M_1, \dots$, ebenfalls eine unendliche Folge von Registern.

Die Instruktionen $READ_j$ und $WRITE_j$ von P_i transportieren Daten zwischen dem lokalen Akkumulator $R_{i,0}$ von P_i und dem Register M_j des globalen Speichers.

Alle anderen Befehle von P_i betreffen den lokalen Speicher von P_i . Um die PID verwenden zu können, gibt es eine zusätzliche Operation $LOAD(PID)$, mit der ein Prozessor seine PID in seinen Akkumulator lädt.

Ein- und Ausgabekonventionen: Eine Eingabe $X = x_1, \dots, x_n$ steht in den Registern M_1, \dots, M_n des globalen Speichers, d.h. M_j speichert x_j . Das Register M_0 enthält die Länge n der Eingabe. Eine Ausgabe der Länge m wird in M_1, \dots, M_m geschrieben.

³Entnommen dem Skript *Automaten und Komplexität (AUK)* von M. Jantzen und diesem Skript angepasst.

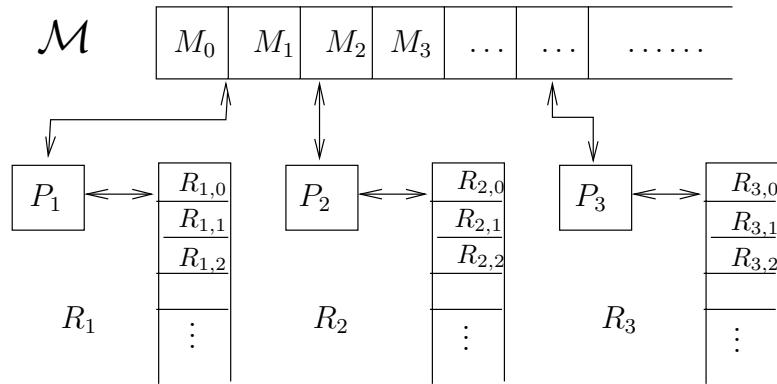


Abbildung 10.8: Parallele RAM - PRAM

Beim Erkennen von Sprachen kann die Entscheidung, ob eine Eingabe akzeptiert oder verworfen wird, auch durch die Art der Halte-Instruktion von P_1 geschehen.

Die PRAM wird spezifiziert durch ein Programm für jeden Prozessor – eine Folge von Instruktionen, die er auf eine Eingabe ausführt. Dies geschieht schrittweise synchron mit den anderen Prozessoren. Dabei gelte:

Die Programme sind für alle Prozessoren identisch!

Eine Berechnung startet, indem jeder Prozessor synchron mit den Anderen die erste Instruktion seines Programmes ausführt. Sie endet, wenn P_1 eine Halte-Instruktion erreicht. Die Beschriftung des globalen Speichers zu diesem Zeitpunkt spezifiziert die Ausgabe.

Bemerkung: Selbst wenn alle Prozessoren das gleiche Programm ausführen, bedeutet dies nicht, dass sie zu jedem Zeitpunkt vollkommen identische Instruktionen ausführen (und damit die Parallelität keinen Vorteil brächte). Da Prozessoren ihre PID zur Verfügung steht, kann ihr Verhalten von dieser abhängen, und sie können somit mit unterschiedlichen Daten rechnen! P_i kann beispielsweise seine PID als Adresse benutzen, um das i -te Eingabesymbol zu lesen, falls $i \leq n$.

Außerdem kann sich der Programmablauf einzelner Prozessoren aufgrund von unterschiedlichen Sprüngen bei JUMP-Instruktionen unterscheiden. Dies kann dazu führen, dass die Prozessoren in einem Schritt unterschiedliche Arten von Instruktionen ausführen.

10.2.2 Konflikte beim Speicherzugriff

Die einzige Schwierigkeit, die bei PRAMs auftritt, sind Konflikte beim gleichzeitigen Zugriff mehrerer Prozessoren auf ein Register des globalen Speichers.

Das Lesen eines Registers M_j , auf dem simultan auch eine Schreiboperation ausgeführt wird, ist unkritisch – wir vereinbaren nämlich, dass in jedem synchronen Schritt alle *READs* vor den *WRITEs* ausgeführt werden. Versuchen dagegen mehrere Prozessoren gleichzeitig in ein globales Register zu schreiben, muss eine Vereinbarung getroffen werden, wie sich dessen Inhalt verändert. Wir unterscheiden drei grundlegende PRAM-Modelle:

EREW PRAM (Exclusive Read, Exclusive Write):

Simultane *READs* und *WRITEs* sind nicht erlaubt.

CRCW PRAM (Concurrent Read, Concurrent Write):

Simultanes Lesen ist uneingeschränkt erlaubt, simultanes Schreiben ist ebenfalls erlaubt.

Es gibt verschiedene Modelle von *CRCW PRAM*. Sie unterscheiden sich durch die Art, wie der Inhalt eines Registers nach einer simultanen Schreiboperation festgelegt wird:

CRCW^{com} PRAM (common):

Ein gleichzeitiges Schreiben in ein Register M_j ist nur zulässig, wenn alle beteiligten Prozessoren versuchen denselben Wert zu schreiben.

CRCW^{arb} PRAM (arbitrary):

Wenn einige Prozessoren versuchen, in ein Register zu schreiben, ist einer erfolgreich. Es ist aber nicht a priori festgelegt, welcher.

CRCW^{pri} PRAM (priority):

Wenn einige Prozessoren versuchen, in ein Register zu schreiben, ist der mit dem kleinsten Index erfolgreich (Man sagt, dieser besitzt die höchste Priorität).

CREW PRAM (Concurrent Read, Exclusive Write):

Gleichzeitiges Lesen erlaubt, jedoch nur exklusives Schreiben.

10.2.3 Komplexitätsmaße der PRAM

Definition 10.10 Die uniforme (logarithmische) Zeitkomplexität $time_M(x)$ einer PRAM M auf eine Eingabe x ist entsprechend wie für die RAM in Definition 10.4 definiert, wobei die synchronen Schritte bis zur Termination gerechnet werden. Die Zeitkomplexität $T_M(n)$ von M ist wieder das Maximum aller $time_M(x)$, wobei das Maximum über alle Eingaben x der uniformen (logarithmischen) Größe n gerechnet wird.

Anders als bei den Pseudokode-Programmen am Ende dieses Kapitels ist die Anzahl der benutzten Prozessoren unendlich, wobei meist nur endlich viele davon effektiv genutzt werden. Das sind diejenigen, die einen Schreibbefehl ausführen und dadurch das Ergebnis beeinflussen können. Die anderen Prozessoren lesen nur „still“ mit und werden bei einer Implementation nicht berücksichtigt. Deshalb definieren wir:

Definition 10.11 Die Prozessorkomplexität $proc_M(x)$ einer PRAM M auf eine Eingabe x sei erklärt durch $proc_M(x) = \max\{i \mid i = 1 \text{ oder } P_i \text{ führt auf } x \text{ ein } WRITE \text{ aus}\}$.

Die Prozessorkomplexität $P_M(n)$ von M ist das Maximum aller $proc_M(x)$, wobei das Maximum über alle Eingaben („Probleminstanzen“) der uniformen (logarithmischen) Größe n gebildet wird,

Für ein gegebenes algorithmisches Problem \mathcal{P} ist es eine wichtige Aufgabe den besten Algorithmus für PRAM zu finden, der \mathcal{P} lösen kann. Die Lösung hängt oft von der Anzahl der Prozessoren ab, die man benutzen kann. Z. B. benötigen alle sequentiellen Algorithmen um n Zahlen zu sortieren, mindestens $\Omega(n \log n)$ Zeit. Es gibt aber einen PRAM-Algorithmus, der dieses in $O(\log n)$ Zeit leistet. Dieser benötigt allerdings $O(n^2)$ Prozessoren.

Definition 10.12 Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x :

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

bezeichnet. Die Operationenkomplexität $W_M(n)$ ist dann wieder das Maximum aller $work_M(x)$, wobei das Maximum über alle Eingaben x der uniformen (logarithmischen) Größe n gerechnet wird.

In der Regel ist das Prozessor-Zeit-Produkt viel größer als das exaktere, aber schwerer zu berechnende Work. Das Prozessor-Zeit-Produkt ist also eine (grobe) Abschätzung von Work nach oben.

Wir definieren wieder, dass eine PRAM für ein algorithmisches Problem \mathcal{P} optimal ist, wenn $W_M(x) = O(T(n))$ gilt, wobei $T(n)$ die Zeitkomplexität des schnellsten RAM-Algorithmus für \mathcal{P} ist.

Das Ziel beim Entwerfen von PRAM-Algorithmen ist es, für wichtige algorithmische Probleme optimale PRAM-Algorithmen zu konstruieren, deren Zeitkomplexität so klein wie möglich ist – am Besten (fast) konstant ($O(1)$).

Überraschenderweise gibt es für viele wichtige algorithmische Probleme deterministische oder probabilistische PRAM-Algorithmen, die optimal und sehr schnell sind – mit Zeitkomplexität $O(\log n)$, $O(\log \log n)$, $O(\log \log \log n)$, $O(\log^* n)$ und $O(1)$.

Versucht man Zeit durch Erhöhung der Parallelität zu sparen, so steht man vor dem *Problem der Parallelisierung*: Ist eine Verringerung der Laufzeit um ein beliebiges k möglich? Mit diesem Problem werden wir uns im Folgenden beschäftigen.

Satz 10.13 (Brent's scheduling principle) Wenn man eine Berechnung auf $m = \max_i x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i -ten Schritt ist, dann kann man auf

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \left\lfloor \frac{x}{p} \right\rfloor$$

ausführen, wobei $x = \sum x_i$

Beweis:

$$\text{Mit } \left\lceil \frac{x_i}{p} \right\rceil \leq \frac{x_i}{p} + 1 \text{ gilt } \sum_{i=1}^t \left\lceil \frac{x_i}{p} \right\rceil = \left\lceil \sum_{i=1}^t \left\lceil \frac{x_i}{p} \right\rceil \right\rceil \leq \left\lceil \sum_{i=1}^t \left(\frac{x_i}{p} + 1 \right) \right\rceil = t + \left\lfloor \frac{x}{p} \right\rfloor$$

□

Korollar 10.14 Wenn man eine Berechnung, die $O(n)$ sequentielle Zeit braucht, auf n Prozessoren in $O(\log n)$ Zeit ausführen kann, dann kann man diese Berechnung auf $O\left(\frac{n}{\log n}\right)$ Prozessoren in Zeit $O(\log n)$ ausführen – d.h. in optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$.

Beweis: Man wähle $x = O(n)$, $t = O(\log n)$, $p = O\left(\frac{n}{\log n}\right)$. □

Wenn die Zuordnung von Daten zu Prozessoren kein Problem ist, dann kann man dieses Korollar auf PRAMs anwenden.

10.2.4 Beispiele für PRAM-Algorithmen

Um die Algorithmen zu beschreiben, benutzen wir den parallelen Ausdruck

$$\underline{\text{par}} [a \leq i \leq b] S_i$$

um zu beschreiben, dass die Anweisungen S_i für alle i mit $a \leq i \leq b$ parallel ausgeführt werden sollen.

Beispiel 10.15 Maximum finden Gegeben: $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \dots, a_{2n-1}$. Gesucht wird ihr Maximum.

Algorithmus (für EREW-PRAM):

```
for l ← m - 1 down to 0 do
    par [2l ≤ j ≤ 2l+1 - 1] aj ← max{a2j, a2j+1}
```

Erklärung:

Der Algorithmus benötigt $O(m) = O(\log n)$ Zeit und $\frac{n}{2}$ Prozessoren. Die Berechnung kann man für $m = 3$ durch den Berechnungsbaum von Abbildung 10.9 darstellen.

Nun genügt es, Brent's principle zu benutzen, um einen optimalen Algorithmus zu erhalten.

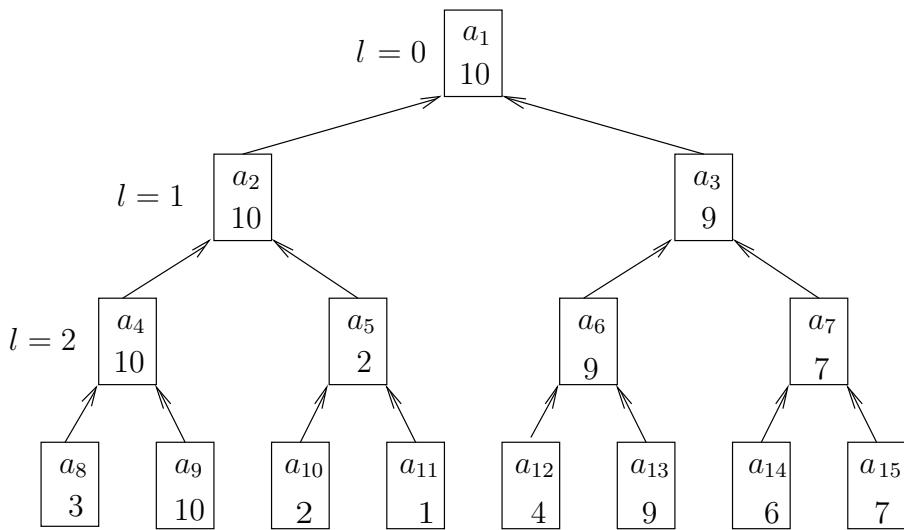


Abbildung 10.9: Maximum finden durch die PRAM

Beispiel 10.16 Maximum finden Der vorhergehende parallele Algorithmus scheint zeitoptimal zu sein. Das hieße, dass das Maximum nicht schneller als in $O(\log n)$ Zeit berechnet werden kann. Das folgende Beispiel zeigt, dass solches „common sense reasoning“ falsch sein kann. Dieses Beispiel zeigt außerdem die Mächtigkeit des parallelen Schreibens.

Folgender Algorithmus (für CRCW^{arb}, CRCW^{com}, oder CRCW^{pri}-PRAM) kann mit n^2 Prozessoren das Maximum (Minimum) von n Zahlen in der Zeit $O(1)$ finden.

Prozessoren: P_{ij} , $1 \leq i, j \leq n$;

Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \dots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i , unter denen das Maximum gefunden werden soll;

Arbeitsspeicher: $Y(1), \dots, Y(n)$ – Register des globalen Speichers;

Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij} :

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

Man beachte, daß in allen parallelen Schreibanweisungen der gleiche Wert geschrieben wird, und daher der *common-write* Modus nicht wirklich ausgenutzt wird!

Beispiel 10.17 Berechnung der ODER-Funktion auf EREW-PRAM Der folgende Algorithmus berechnet die Funktion

ODER: $\{0, 1\}^n \rightarrow \{0, 1\}$, $OR(x_1, \dots, x_n) = \text{if any } x_i = 1 \text{ then } 1 \text{ else } 0$.

Input: $M(1), \dots, M(n)$ – Register des globalen Speichers

Output: $M(1)$

Prozessoren: P_1, \dots, P_n

Arbeitsspeicher: Y_i – jeweils lokales Register von P_i

Algorithmus für den Prozessor P_i :

```

begin  $t \leftarrow 0$ ;
       $\underline{Y_i \leftarrow M(i)}$ ;
      while  $i + 2^t \leq n$ 
        do  $Y_i \leftarrow Y_i \vee M(i + 2^t)$ 
           $M(i) \leftarrow Y_i$ ;
           $t \leftarrow t + 1$ 
      endwhile
    end

```

Analyse:

Zeitkomplexität: $\lceil \log n \rceil + 1$, Prozessorkomplexität: n

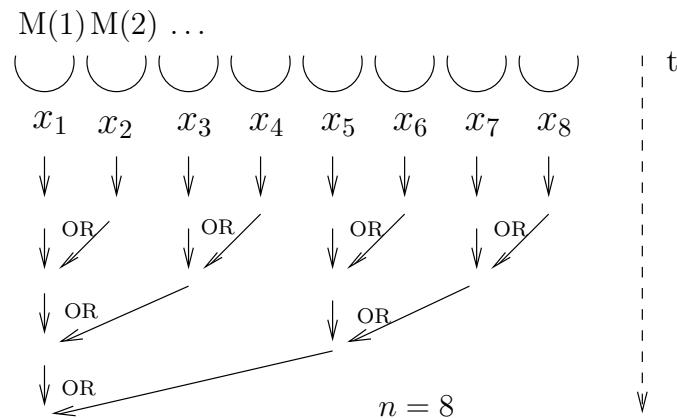


Abbildung 10.10: ODER-Berechnung durch die PRAM

Benutzen wir Brent's principle, bekommen wir einen optimalen Algorithmus mit $O(\log n)$ Zeit und $O\left(\frac{n}{\log n}\right)$ Prozessoren.

Beispiel 10.18 Präfixsumme Gegeben seien $n = 2^m$ Zahlen $a_n, a_{n+1}, \dots, a_{2n-1}$. Zu berechnen sind alle partiellen Summen $b_{n+j} = \sum_{k=n}^{n+j} a_k = a_n + \dots + a_{n+j}$, mit $j = 0, \dots, n-1$. Wir zeigen, dass es für dieses Problem einen $O(\log n)$ -Algorithmus für CREW-PRAM gibt. Dieser Algorithmus wird sehr oft benutzt, um schnelle Algorithmen für PRAM zu erstellen.

Algorithmus für CREW-PRAM:

```

for  $l \leftarrow m - 1$  downto 0 do
  par [ $2^l \leq j < 2^{l+1}$ ]  $a_j \leftarrow a_{2j} + a_{2j+1}$ ;

```

```

 $b_1 \leftarrow a_1;$ 
for  $l \leftarrow 1$  to  $m$  do
    par [ $2^l \leq j < 2^{l+1}$ ]
         $b_j \leftarrow$  if  $j$  odd then  $b_{(j-1)/2}$  else  $b_{j/2} - a_{j+1}$ 
    
```

Dieser Algorithmus braucht $O(\log n)$ Zeit und $O(n)$ Prozessoren. Mit Hilfe von Brent's Prinzip lässt sich ein optimaler Algorithmus finden.

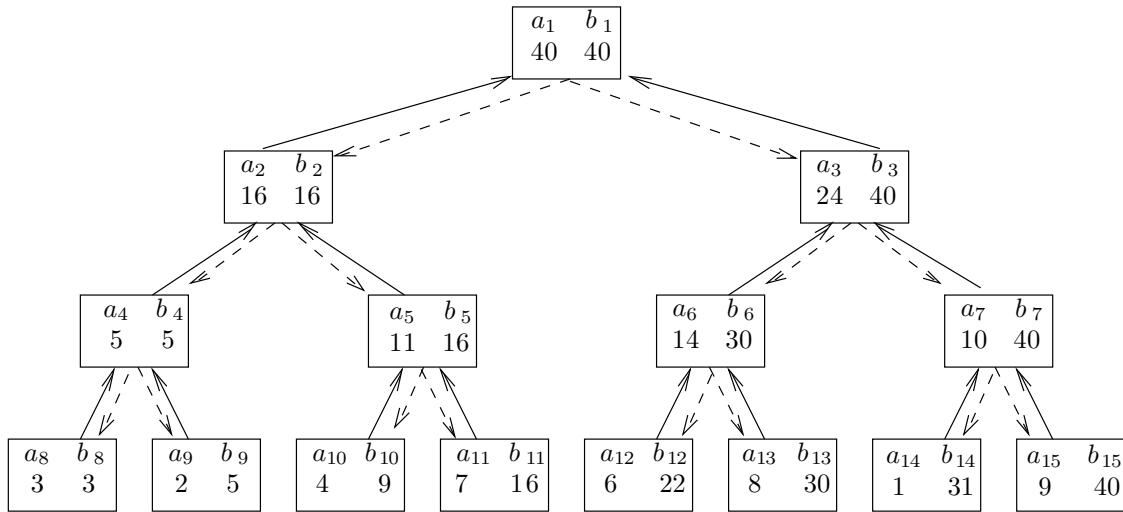


Abbildung 10.11: Präfixsummenberechnung durch die PRAM

Erste Phase (erste for-Schleife, durchgezogene Pfeile): Berechnung von a_i , $1 \leq i \leq n-1$.
Zweite Phase (zweite for-Schleife, unterbrochene Pfeile): Jeder Vater schickt seinen b -Wert seinen beiden Kindern. Das rechte Kind behält diesen als seinen b -Wert. Das linke Kind subtrahiert von diesem b -Wert den a -Wert des rechten Bruders.

10.2.5 PRAM-Hauptkomplexitätsklassen

Sei \mathcal{M} eine beliebige PRAM-Variante. Wir bezeichnen mit

$$\mathcal{M}\text{TimeProc}(T, P)$$

die Menge der Funktionen bzw. Sprachen, die von einer T -Zeit- und P -Prozessorbeschränkten PRAM vom Typ \mathcal{M} berechnet werden können.

Zwischen Prozessor- und Zeitkomplexität bei PRAMs besteht die folgende Beziehung:

Satz 10.19 Seien $T, P : \mathbb{N} \rightarrow \mathbb{N}$ Funktionen und $k : \mathbb{N} \rightarrow \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:

$$\mathcal{M}\text{TimeProc}(T, P) \subsetneqq \mathcal{M}\text{TimeProc}\left(c \cdot k \cdot T, \left\lceil \frac{P}{k} \right\rceil\right)$$

Beweis: Um eine P -Prozessorbeschränkte PRAM M mit $\lceil \frac{P}{k} \rceil$ Prozessoren zu simulieren, konstruieren wir eine PRAM M' mit P' Prozessoren, die folgendermaßen arbeitet:

P'_1 berechnet auf Eingabe n zunächst $k(n)$. Anschließend simuliert jeder Prozessor P'_i die Prozessoren von M , die die Prozessorkennung (PID) $(i-1)k(n)+1, \dots, ik(n)$ haben. Jeder synchrone Schritt von M wird in $k(n)$ Phasen simuliert, in denen nacheinander für jeden korrespondierenden Prozessor sein entsprechender Schritt nachgeahmt wird. Dazu unterteilt P'_i seinen lokalen Speicher, um die lokalen Daten der zu simulierenden Prozessoren speichern zu können. Zur Simulation eines Schrittes eines Prozessors P_j lädt P'_i die zugehörigen Operanden in seine Operationsregister.

Bei CRCW-PRAMs muss dabei sichergestellt werden, dass die alten Inhalte der Register während der gesamten Simulation eines Schrittes zur Verfügung stehen und im Fall der Prioritätsregel beim simultanen Schreiben der Prozessor mit höchster Priorität am Ende den neuen Inhalt bestimmt.

Zu diesem Zweck kann man jedes Register durch ein Tripel von Registen ersetzen, mit denen der alte Inhalt, der momentane neue Inhalt und der Prozessor, der bislang die höchste Priorität besaß, dargestellt werden.

Zum Lesen wird das erste Register eines Tripels mit dem alten Inhalt benutzt. Vor einem Schreiben in $CRCW^{pri}$ -PRAM wird überprüft, ob der momentan simulierte Prozessor eine höhere Priorität besitzt, und nur in diesem Fall geschrieben. \square

Als ein Korollar bekommen wir:

Korollar 10.20

$$CRCW_+ \text{ TimeProc(POL, POL)} = \mathcal{P} \quad (\text{Polynomialzeit})$$

Beweis: Für Polynome P, T und $k = P$ ergibt sich aus dem vorigen Satz:

$$\begin{aligned} CRCW_+ \text{ TimeProc}(T, P) &\subseteq CRCW_+ \text{ TimeProc}(O(P \cdot T), 1) \\ &= RAM_+ - \text{Time}(O(T \cdot P)) \subseteq \mathcal{P} \end{aligned}$$

\square

Das bedeutet, dass die PRAMs, die nur polynomiell viele Prozessoren benutzen und nur in Polynomialzeit arbeiten, nicht mächtiger sind als die sequentiellen RAMs, die auch in Polynomialzeit arbeiten!

Wenn wir die Prozessoranzahl aber exponentiell beschränken, dann erhalten wir eine größere Klasse. Wir definieren dazu:

$$PRAM - \text{Time}(t(n)) := CRCW_+ \text{ TimeProc}(t(n), 2^{t(n)})$$

Satz 10.21

$$PRAM - \text{Time}(POL) = \mathcal{PSPACE}$$

Dies liegt zum einen daran, dass man jede TM, die mit der konstruierbaren Platzkomplexität $s(n) \geq \log(n)$ arbeitet, durch eine PRAM mit $O(s)$ Prozessoren in der Zeit $O(s)$ simulieren kann:

$$PSPACE(s(n)) \subseteq PRAM - \text{Time}(O(s(n)))$$

Zum anderen kann man jede PRAM, die mit der konstruierbaren Zeitkomplexität $t(n) \geq \log(n)$ und mit $2^{t(n)}$ Prozessoren arbeitet, auf einer TM simulieren, die $O(t^2(n))$ Platz benötigt:

$$PRAM - Time(t(n)) \subseteq PSpace(t^2(n))$$

Der vorhergehende Satz besagt, dass Polynomialzeit für PRAMs genauso mächtig ist, wie Polynomialplatz für RAMs!

Dasselbe gilt auch für andere Modelle paralleler Computer, z.B. auch für APM. Diese Resultate sprechen für folgende These auf der die moderne parallelkomplexitätstheorie begründet ist.

Parallel Computation Thesis *There exists a standard class of (parallel) machine models, which includes among others all variants of PRAM machines, and also APM, for which polynomial time is as powerful as polynomial space for sequential machines (from the first machine class).*

Die Maschinen-Modelle, die dieser These genügen, bilden die sogenannte *zweite Maschinenklasse*.

Es scheint, dass *PSPACE* viel mächtiger ist als \mathcal{P} , aber niemand hat das bisher bewiesen. Deshalb ist die Frage

$$\mathcal{P} \stackrel{?}{=} \mathcal{PSPACE}$$

eines der wichtigsten und bedeutendsten Probleme der Informatik.

10.2.6 Grenzen der Parallelität

Natürliche Frage: Gibt es ein natürliches und einfaches Problem, für das es keinen parallelen Algorithmus gibt, der schneller ist als der schnellste sequentielle Algorithmus?

Satz 10.22 (Kung) *Kein paralleler Algorithmus, der die Operationen $+, -, *, \div$ benutzt, kann ein Polynom n -ten Grades schneller als in $\log n$ Schritten berechnen.*

Korollar 10.23 *Kein paralleler Algorithmus kann x^n schneller als ein sequentieller Algorithmus berechnen.*

	Zeit $T_A(n)$	Operationen $W_A(n)$	optimal ?
Mischen 1: Satz 10.27	$O(\log n)$	$O(n)$	ja
Mischen 2: Korollar 10.31	$O(\log \log n)$	$O(n \log \log n)$	nein
Mischen 3: Satz 10.32	$O(\log \log n)$	$O(n)$	ja
Sortieren: Satz 10.33	$O(\log n \log \log n)$	$O(n \log n)$	ja

Tabelle 10.4: Mischen und Sortieren

10.3 Paralleles Suchen und optimales Mischen

Während die bisher behandelten parallelen Algorithmen relativ einfach waren und mehr der Erläuterung des Maschinenmodells dienten, wird in den beiden letzten Abschnitten dieses Kapitels gezeigt, wie ein prominentes Problem, nämlich das Sortieren, nach nicht elementaren Überlegungen mit parallelen Algorithmen exponentiell beschleunigt werden kann. Dabei ist die Anzahl der Prozessoren immer von der Ordnung $O(n)$. Zunächst werden Algorithmen zum „Mischen“ entsprechend Tabelle 10.4 schrittweise verbessert, um dann als Prozedur für das parallele Sortieren in Algorithmus 10.4 eingesetzt zu werden.

Anders als bisher benutzen wir einen Pseudokode, der aber bei Bedarf leicht auf eine PRAM übertragen werden kann. So gehen auch oft Lehrbücher zu parallelen Algorithmen vor [Jáj92], [Rei93], [Cha92] und [GS93]. Die folgenden Algorithmen sind in [Jáj92] ausführlicher beschrieben. (Wie üblich bezeichnet $|M|$ die Mächtigkeit einer Menge M .) **pardo** ist eine Anweisung zur (synchron-)parallelen Ausführung.

Definition 10.24 Gegeben sei eine lineare Ordnung (S, \leq) (Def. 5.2) und zwei Teilmengen $\tilde{A}, \tilde{B} \subseteq S$ mit $|\tilde{A}| = |\tilde{B}| = n > 0$, zu denen wir die Folgen: $A = (a_1, a_2, \dots, a_n)$ und $B = (b_1, b_2, \dots, b_n)$, mit $i < j \Rightarrow a_i \leq a_j \wedge b_i \leq b_j$ ($1 \leq i, j \leq n$) benutzen. Die **Mischung (merge)** von A und B ist die Folge $C = (c_1, c_2, \dots, c_{2n})$ mit $i < j \rightarrow c_i \leq c_j$, die A und B als disjunkte Teilstufen enthält.

Beispiel: $(2, 4, 4) \quad (3, 4, 7)$

$$\begin{array}{ccc} & \searrow & \swarrow \\ & (2, 3, 4, 4, 4, 7) & \end{array}$$

Definition 10.25 Sei $X = (x_1, x_2, \dots, x_t)$ eine Folge mit Elementen aus der linear geordneten Menge S . Für $x \in S$ ist der Rang von x in X definiert als: $\text{rank}(x : X) := |\{i | i \in \{1, \dots, t\} \wedge x_i < x\}|$. Für eine weitere solche Folge $Y = (y_1, \dots, y_s)$ sei $\text{rank}(Y : X) := (r_1, r_2, \dots, r_s)$ mit $r_i = \text{rank}(y_i : X)$

Beispiel: Für $X = (25, -13, 26, 31, 54, 7)$ und $Y = (13, 27, -27)$ ist $\text{rank}(Y : X) = (2, 4, 0)$.

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 $x \in A \cup B : i := \text{rank}(x : A \cup B)$
 (dann ist x das $(i + 1)$ -te Element c_i in C)

Wegen $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$ genügt uns ein Algorithmus für:

Berechne $\text{rank}(B : A)$

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. $j_i = \text{rank}(b_i : A)$. 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.

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_0	A_1	...	A_{k_m-1}
B:	B_0	B_1	...	B_{k_m-1}

wobei $A = (a_1, a_2, \dots, a_n)$, $B = (b_1, b_2, \dots, b_m)$ 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 10.1 wird k_m als $k(m)$, a_{j_i+1} als $a_{j(i)+1}$ usw. geschrieben.)

Algorithmus 10.1 (Partitionieren)

Eingabe: Zwei Felder $A = (a_1, \dots, a_n)$ und $B = (b_1, \dots, b_m)$ in aufsteigender Reihenfolge sortiert, wobei $\log m$ und $k(m) = \frac{m}{\log m}$ ganze Zahlen sein müssen.

Ausgabe: $k(m)$ Paare (A_i, B_i) von Teilstücken von A und B , so dass

- (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** $1 \leq i \leq k(m) - 1$ **par do**
 - 2.1 Berechne $\text{rank}(b_{i \log m} : A)$ durch binäre Suche und setze $j(i) = \text{rank}(b_{i \log m} : A)$
3. **for** $0 \leq i \leq k(m) - 1$ **par do**
 - 3.1 $B_i := (b_{i \log m + 1}, \dots, b_{(i+1) \log m})$
 - 3.2 $A_i := (a_{j(i)+1}, \dots, a_{j(i+1)})$
 $(A_i$ ist leer, wenn $j(i) = j(i + 1)$ gilt.)

end

Beispiel: (zum Algorithmus 10.1) Gegeben seien die folgenden zwei Felder A und B mit $m = 4$ und $k_m := k(m) = \frac{m}{\log m} = 2$

$$A = (4, 6, 7, 10, 12, 15, 18, 20) \quad B = (3, 9, 16, 21)$$

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 10.26 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 10.1 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 $C = (C_0, C_1, \dots)$, wobei C_i die sortierte Folge ist, welche aus A_i und B_i durch Mischen entsteht. Dieser Algorithmus 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 $\text{rank}(b_{i \log m} : A) = j(i)$, 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 $O\left((\log n) \times \left(\frac{m}{\log m}\right)\right) = O(m + n)$, 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. \square

Mit diesem Algorithmus haben wir das Mischproblem der Größe n auf Unterprobleme kleinerer Größe reduziert.

Optimaler Algorithmus für das Mischen:

- wende Algorithmus 10.1 an
- behandle die Paare (A_i, B_i) getrennt und parallel (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 sequentiellen Mischalgorithmus
- falls $|A_i| \not\leq c \log n$, dann wende Algorithmus 10.1 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 10.27 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 (partitioning)

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 $X = (x_1, \dots, x_n)$ von verschiedenen Elementen einer linear geordneten Menge (S, \leq)
- $y \in S$
- *Suchproblem:* Finde Index $i \in \{0, 1, \dots, n\}$ 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.

Algorithmus 10.2 (Paralleles Suchen für Prozessor P_j)

Eingabe: (1) Ein Feld $X = (x_1, x_2, \dots, x_n)$ mit $x_1 < x_2 < \dots < x_n$
 (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  $q_0 := l; q_{p+1} := r$ }
        2.2. Set  $q_j := l + j \lfloor \frac{r-l}{p+1} \rfloor$ 
        2.3. if( $y = x_{q_j}$ ) 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

```

Beispiel: (zum Algorithmus 10.2) Für die Eingabe

$$X = (2, 4, 6, \dots, 30), y = 19, p = 2$$

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.

Iteration:	1	2	3	
q_0	1	1	1	
q_1	5	6	8	
q_2	10	7	9	
q_3	16	10	10	
c_0	0	0	0	Ergebnis durch P_1 : return (q_1) mit $q_1 = 9$
c_1	0	0	0	
c_2	1	0	0	
c_3	1	1	1	
l	5	7	9	
r	10	10	10	

Satz 10.28 Zu der Folge $X = (x_1, x_2, \dots, x_n)$ mit $x_1 < x_2 < \dots < x_n$ und $y \in S$ berechnet der Algorithmus 10.2 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 $i+1$ -ten Iteration der while-Schleife wird die Länge der zu durchsuchenden Unterfolge von $s_i = r - l$ auf $s_{i+1} \leq \frac{r-l}{p+1} + p = \frac{s_i}{p+1} + p$ 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. \square

Anmerkung: Der Algorithmus ist optimal, wenn p konstant ist.

Als Nächstes wird ein Algorithmus zum parallelen Mischen entwickelt, der in $O(\log \log n)$ Zeit arbeitet. Das ist erstaunlich, da schon die Maximumbestimmung $\Omega(\log n)$ parallele Schritte erfordert (bezogen auf eine CREW PRAM, d.h. eine PRAM die parallel lesen (concurrent read), aber nur exklusiv schreiben (exclusive write) darf. Diese PRAM-Modelle werden in der Hauptstudiumsveranstaltung AUK genauer erklärt.)

Lemma 10.29 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 $\text{rank}(Y : X)$ in $O(1)$ Zeit mit $O(n)$ Operationen berechnet werden.

Beweis:

- Bestimme $\text{rank}(y : X)$ für jedes $y \in X$ mit Algorithmus 10.2 mit $p = \lfloor \frac{n}{m} \rfloor \in \Omega(n^{1-s})$ (vergl. Skript zu F3)
- Also kann $(y : X)$ 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.

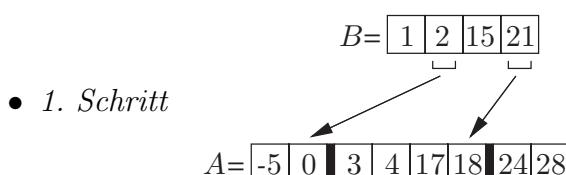
\square

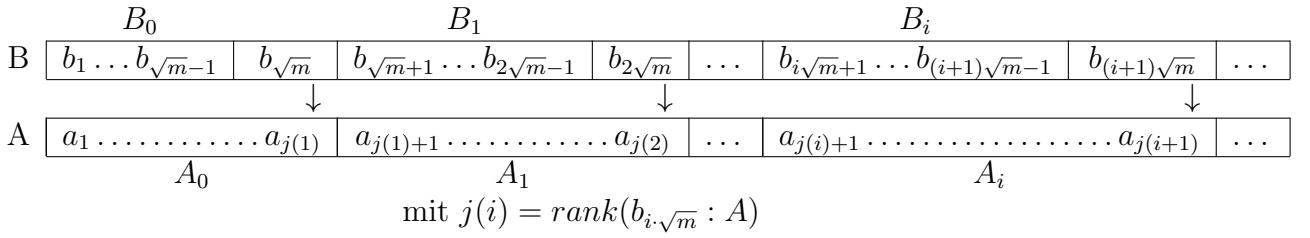
Nun soll $\text{rank}(B : A)$ für sortierte Folgen A mit $|A| = n$ und B mit $|B| = m$ berechnet werden. A und B sollen keine gemeinsamen Elemente enthalten.

Wieder: Berechnen durch Aufteilen von B in Blöcke der Länge von etwa \sqrt{m} :

Daraus: Algorithmus 10.3

Beispiel: (zum Algorithmus 10.3) ($m = 4, \sqrt{m} = 2$)




 Abbildung 10.12: Aufteilung von B in Blöcke

Algorithmus 10.3 (Ordne eine sortierte Folge in eine andere sortierte Folge ein)

Eingabe: Zwei Felder $A = (a_1, \dots, a_n)$ und $B = (b_1, \dots, b_m)$ in aufsteigender Reihenfolge.

Ausgabe: Das Feld $\text{rank}(B : A)$.

begin

1. Wenn $m < 4$ dann ordne die Elemente von B durch Anwendung des Algorithmus 10.2 mit $p = n$. Fertig.
2. Ordne die Elemente $b_{\lfloor \sqrt{m} \rfloor}, b_{2\lfloor \sqrt{m} \rfloor}, \dots, b_{i\lfloor \sqrt{m} \rfloor}, \dots, b_m$ in A mit Hilfe des Algorithmus 10.2 ein. Dabei sei $p = \sqrt{n}$ und $\text{rank}(b_{i\lfloor \sqrt{m} \rfloor} : A) = j(i)$, für $1 \leq i \leq \sqrt{m}$ und $j(0) = 0$
3. Für $0 \leq i \leq \sqrt{m} - 1$ sei $B_i = (b_{i\lfloor \sqrt{m} \rfloor + 1}, \dots, b_{(i+1)\lfloor \sqrt{m} \rfloor - 1})$ und $A_i = (a_{j(i)+1}, \dots, a_{j(i+1)})$; Wenn $j(i) = j(i+1)$ dann setze $\text{rank}(B_i : A_i) = (0, \dots, 0)$, ansonsten berechne $\text{rank}(B_i : A_i)$ 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 $\text{rank}(b_k : A) = j(i) + \text{rank}(b_k : A_i)$

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 $\text{rank}(b_1, A) = \text{rank}(1, A) = j(0) + \text{rank}(1 : A_0) = 2$
 $\text{rank}(b_3, A) = \text{rank}(15, A) = j(1) + \text{rank}(15 : A_1) = 2 + 2 = 4$
 Also: $\text{rank}(B : A) = (\underline{2}, 2, \underline{4}, 6)$

Lemma 10.30 Der Algorithmus 10.3 berechnet $\text{rank}(B : A)$ 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 einzurichten. Dies erfolgt mit Zeile 1. Wir nehmen jetzt an, dass die Induktionsbehauptung für alle $m' < m$ mit $m \geq 4$ 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 $j(i) = \text{rank}(b_{i\sqrt{m}} : A)$ und $j(i+1) = \text{rank}(b_{(i+1)\sqrt{m}} : A)$ gilt $a_{j(i)} < b_{i\sqrt{m}}$ und $b_{(i+1)\sqrt{m}} < a_{j(i+1)+1}$, und somit auch $a_{j(i)} < p < a_{j(i+1)+1}$. Diese Tatsache zeigt, dass jedes Element p des Blocks B_i in den Block A_i eingefügt wird. Damit gilt $\text{rank}(p : A) = j(i) + \text{rank}(p : A_i)$, 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 10.2, 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 $O(\sqrt{m} \cdot \sqrt{n}) = O(n + m)$ beschränkt, da $2\sqrt{m} \cdot \sqrt{n} \leq n + m$. 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 10.3 mit $O((n + m) \log \log m)$. \square

Korollar 10.31 *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.

10.4 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 10.32 *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 10.4 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 $v = (h, j)$.

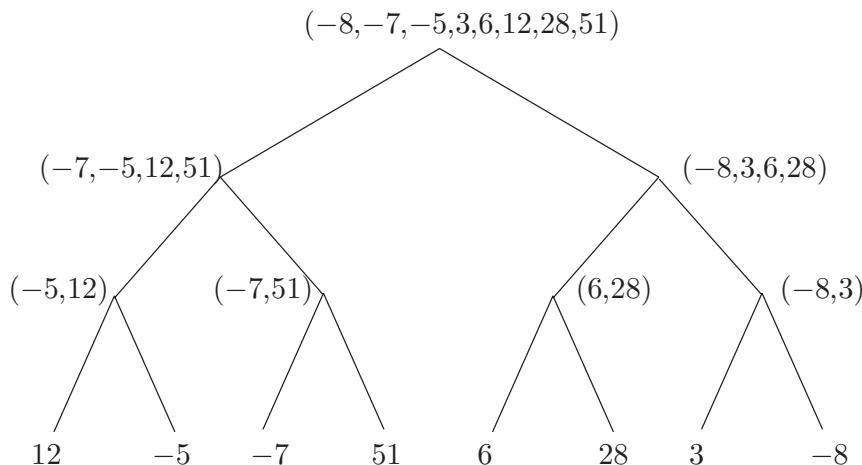


Abbildung 10.13: Binärbaum

Für obiges Beispiel wird das Ergebnis an der Wurzel $L(3, 1)$ erreicht.

Satz 10.33 *Der Algorithmus 10.4 ist mit einer Zeitkomplexität von $O(\log n \cdot \log \log n)$ und $O(n \cdot \log n)$ Operationen optimal.*

Beweis: In Höhe h sind $\frac{n}{2^h}$ Folgen der Länge 2^h zu mischen. Also gilt nach Satz 10.32 für Operationen in Höhe h :

$$T(n) = O(\log \log 2^h) = O(\log h) \leq O(\log \log n)$$

Algorithmus 10.4 (Parallel 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 $X(2^h(j-1)+1), X(2^h(j-1)+2), \dots, X(2^hj)$.

```

begin
    1. for  $1 \leq j \leq n$  par do
         $L(0, j) := X(j)$ 
    2. for  $h = 1$  to  $\log n$  do
        for  $1 \leq j \leq \frac{n}{2^h}$  par do
            Mische  $\tilde{L}(h-1, 2j-1)$  und  $L(h-1, 2j)$  zur sortierten Liste  $L(h, j)$ 
end
```

und

$$W(n) = O(2^h \cdot \frac{n}{2^h}) = O(n).$$

Auf alle $\log n$ Ebenen bezogen ergibt dies $T(n) = O(\log n \cdot \log \log n)$ und $W(n) = O(n \cdot \log n)$. □

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.

Aufgabe 10.34 (parallele Algorithmen)

Gegeben sei ein Feld mit $n > 0$ Elementen von ganzen Zahlen. Es sollen verschiedene parallele Algorithmen zur Bestimmung eines maximalen Elementes entwickelt werden. Dabei sollen jeweils die *Zeitkomplexität* $T(n)$, die *Prozessorkomplexität* $P(n)$ und die *Operationenkomplexität* $W(n)$ betrachtet werden.

- a) Hier sei $n = 2^k$. Entwickeln Sie mit der Vorstellung eines binären Baumes einen Algorithmus mit $P(n) = \mathcal{O}(n)$, $W(n) = \mathcal{O}(n)$ und $T(n) = \mathcal{O}(\log n)$. Ist der Algorithmus optimal?
- b) Hier sei $n = 2^{2^k}$. Entwickeln Sie mit der Vorstellung eines doppel-logarithmischen Baumes⁴ einen Algorithmus mit $P(n) = \mathcal{O}(n)$ und $T(n) = \mathcal{O}(\log \log n)$. Ist der Algorithmus optimal? Kann er optimal gemacht werden, falls er es noch nicht ist?
- c) Entwickeln Sie mit der Vorstellung einer $(n \times n)$ -Matrix einen Algorithmus mit $P(n) = \mathcal{O}(n^2)$ und $T(n) = \mathcal{O}(1)$. Hierbei ist jedoch von Prozessoren auszugehen, die gleichzeitig eine Variable lesen und beschreiben können (*CRCW-PRAM*). Letzteres soll aber nur dann erfolgen, wenn die beteiligten Prozessoren den gleichen Wert schreiben wollen.

⁴Ein doppel-logarithmischer Baum mit $n = 2^{2^k}$ Blättern ist folgendermaßen definiert. Die Wurzel (Höhe 0) hat $2^{2^{k-1}} = \sqrt{n}$ Nachkommen, diese wiederum $2^{2^{k-2}}$ Nachkommen usw. Allgemein haben Knoten mit Höhe $i \in \{0, \dots, k-1\}$ genau $2^{2^{k-i-1}}$ Nachkommen. Die Knoten mit Höhe k haben 2 Blätter als Nachkommen.

Literaturverzeichnis

- [Aal97] W.M.P. van der Aalst. Verification of Workflow Nets. volume 1248 of *Lecture Notes in Computer Science*, pages 407–426, Toulouse, France, 1997. Springer-Verlag.
- [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.
- [AH02] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management - Models, Methods and Systems*. The MIT Press, 2002.
- [AW98] H. Attiya and J. Welch. *Distributed Computing*. McGraw-Hill, 1998.
- [Bae95] J. Baeten. Applications of process algebra. In *Cambridge Tracts in Theoretical Computer Science 17*. Oxford University Press, 1995.
- [BBF99] B. Bérard, M. Bidoit, and A. Finkel. *Systems and Software Verification; Model-Checking Techniques and Tools*. Springer, 1999.
- [BH73] P. Brinch-Hansen. *Operating System Principles*. Prentice Hall, Englewood Cliffs, 1973.
- [BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT press, 2008.
- [BM84] Eike Best and Agathe Merceron. Frozen tokens and d-continuity: a study in relating system properties to process properties. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 188 of *Lecture Notes in Computer Science*, pages 48–61, Berlin, Germany, 1984. Springer-Verlag.
- [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.
- [BV95] J. Baeten and C. Verhoef. Concrete process algebra. In *Handbook of Logic in Computer Science, Vol. IV*, pages 149–268. Oxford University Press, 1995.
- [BW90] J. Baeten and P. Weijland. *Process Algebra*. Cambridge University Press, 1990.
- [CDE⁺99] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. *Maude: Specification and Programming in Rewriting Logic. Maude System documentation*, 1999.
- [CDH92] S. Christensen and N. Damgaard Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. Technical Report DAIMI PB-390, Aarhus University, 1992.

- [CGP99] E.M. Clarke, J.O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, 1999.
- [Cha92] P. Chaudhuri. *Parallel algorithms*. Prentice Hall, 1992.
- [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.
- [EMS02] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In *Electronic Notes in Theoretical Computer Science*, volume 71. Elsevier, 2002.
- [Eng91] Joost Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.
- [Fok99] Wan Fokkink. *Introduction to Process Algebra*. Springer-Verlag, 1999.
- [GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.
- [GR83] Ursula Goltz and Wolfgang Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983.
- [Gru97] J. Gruska. *Foundations of Computing*. International Thompson Computer Press, 1997.
- [GS93] A. Gibbons and P. Spirakis, editors. *Lectures on parallel computation*. Cambridge University Press, 1993.
- [GV03] C. Girault and R. Valk. *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications*. Springer-Verlag, Berlin, 2003.
- [HMU79] J. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Pearson Addison-Wesley Publishing Company, Inc. Wesley, 1979. 3rd edition 2007, T HOP.
- [HR04] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge Univ. Press, Cambridge, 2004.
- [HV87] D. Hauschmidt 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.
- [Jen87] K. Jensen. Coloured Petri Nets. In W. Reisig W. Brauer and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 248–299, Berlin, 1987. Springer-Verlag.
- [Jen94] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2: Analysis Methods*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1994.
- [JK09] K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, Berlin, 2009.

- [JV87] E. Jessen and R. Valk. *Rechensysteme – Grundlagen der Modellbildung*. Springer-Verlag, Berlin, 1987.
- [KM69] R.M. Karp and R.E. Miller. Parallel Program Schemata. *Journal of Computer Sciences*, 3:147–195, 1969.
- [Kum01] O. Kummer. Introduction to Petri Nets and Reference Nets. *Sozionik Aktuell, Internetzeitschrift: <http://www.informatik.uni-hamburg.de/TGI/forschung/projekte/sozionik/journal/index.html>*, 1, 2001.
- [KWD] Olaf Kummer, Frank Wienberg, and Micheal Duvigneau. Renew – The Reference Net Workshop. WWW page at <http://renew.de/>. Contains the documentation of Renew and a more technical introduction to reference nets.
- [Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–564, 1978.
- [Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [Mat89] F. Mattern. *Verteilte Basisalgorithmen*. Springer, 1989.
- [May84] E.W. Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM Journal on Computing*, 13(3):441–460, 1984.
- [Meh84] K. Mehlhorn. *Graph algorithms and NP-completeness*. Springer-Verlag, Berlin, 1984.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Mil99] R. Milner. *Communicating and Mobile Systems: the π - Calculus*. Cambridge University Press, 1999.
- [Pau78] W. J. Paul. *Komplexitätstheorie*. Teubner, Stuttgart, 1978. T PAU.
- [Pet] Petri Nets World. WWW page at <http://www.daimi.au.dk/PetriNets/>. The central source for all information regarding Petri nets.
- [Rei82] W. Reisig. *Petrinetze - eine Einführung*. Springer, Berlin, 1982.
- [Rei90] K. R. Reischuk. *Einführung in die Komplexitätstheorie*. Teubner, Stuttgart, 1990. T REI.
- [Rei93] J. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.
- [Rei10] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Vieweg+Teubner, 2010.
- [Val92] A. Valmari. A Stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.
- [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.

Index

- A^∞ , 5
 A^ω , 5
 $M \models f$, 43
 $M, \pi \models f$, 43
 \leftrightarrow_b , 213
 \leftrightarrow , 192
 δ , 202
 \equiv , 43, 48
 \mathbb{L} , 200
 \mathcal{C}_R , 113
 \parallel , 199
 \models erfüllt, 42
 \models_F , 62
 ω -Sprache, 5
 ω -reguläre Sprache, 7
 ω -regulärer Ausdruck, 7
 \parallel , 75
 ∂ , 85
 ∂_0 , 95
 ∂_1 , 95
 ∂_H , 202
 \leftrightarrow_{rb} , 214
 τ_I , 212
 $f \equiv g$, 43
Äquivalenzproblem, 4, 11

Abbruch, 202
abgeschlossene Menge, 85
abgeschlossene Menge, 85
Abschluss, 158
Abstraktion, 216
Abstraktions-Operator, 212
ACP, 203
Aktion, 13
Aktionsfolgenmenge, 14
Aktivierung, 85
akzeptable Aktionsfolgenmenge, 14
akzeptanzäquivalent, 15

akzeptierte ω -Sprache, 5, 10
akzeptierte Sprache, 1, 14
Alternierbitprotokoll, 23, 185, 221
Analyse des Zustandsraumes, 39
Anfangsbedingung, 29
Anfangszustand, 13
arithmetische RAM, 229
Auswahl, 189
azyklisch, 103

Büchi-Automat, 5, 54
Befehle einer RAM, 231
Befehlszähler, 31
Belegung, 29
Belegungen, 29
Berechnungsbaum, 46
beschränkt, 99, 100
Bezirk, 110
bisimilar, 16, 192
Bisimulation, 15, 192
Bit-RAM, 229
BPA-Kalkül, 194
Brents Prinzip, 243

CFAR, Fairness-Regel, 218
co-Falle, 149
cobegin, 35
coend, 35
CPN (formal), 175
CRCW PRAM, 242
CREW PRAM, 242
CTL*, 49
CTL-Model-Checking, 58
CTL-Pfad-Formel, 45
CTL-Zustands-Formeln, 45

Dekker/Peterson, 35
direkter Nachfolger, 66
einfache Verfeinerung, 87

- einfache Vergrößerung, 87
einfaches Netz, 94
elementare Vergrößerung, 87
elementarer Prozess-Ausdruck, 189
elementarer Prozess-Term, 189
Endliche Automaten, 1
endlicher nichtdeterministischer Automat,
 1
endliches Netz, 84
Endzustand, 13
Epifaltung, 91
Ereignis, 70
Ereignisnetz, 109
EREW PRAM, 242
Erreichbarkeitsmenge, 14, 96, 169, 177
Erreichbarkeitsproblem, 101

fair, 106
faire Kripke-Struktur, 62
faire Schaltregel, 107
faire starke Zusammenhangskomponente,
 63
fairer Pfad, 62
faires Verhalten, 106
Fairness, 62
Falle, 148
Faltung, 88, 91, 111
Flussrelation, 84
Folgen-Semantik, 65
folgenäquivalent, 15
formale Verifikation, 39
formaltextuelle Darstellung, 84
Formeln, 29
Fortschrittseigenschaft, 114

gültige LTL-Formel, 43
Gültigkeit, 46
gefäßtes Netz, 79
gefäßtes Petrinetz, 175
geschützte lineare Rekursion, 215
geschützte rekursive Spezifikation, 206
grafische Darstellung, 82

Hürde, 98
Handlung, 65
Hintereinanderausführung, 189
Homomorphismus, 2

infinite(w), 5
Inhibitor-Kante, 136
Inhibitor-Netz, 137
initiale Verzweigungs-Bisimulation, 214
interleaving semantics, 65
interne Aktion, 212
Inzidenzmatrix, 141

 k -beschränkt, 100, 120
kantenkonstantes Petrinetz, 168
Kausalnetz, 109
KKN, 168
Klique, 110
Kommunikationsalphabet, 20
Kommunikationsfunktion, 199
komplementärer Platz, 136
Komplexitätsmaße für PRAMs, 242
konfliktfrei, 131
Konfliktplatz, 131
Konfliktrelation, 109
Kongruenz, 193
konservativ, 103
Korrektheit, 194
Kripke-Struktur, 27, 28
kritischer Abschnitt, 33

Lautenbach, Satz von, 141
lebendig, 100
Lebendigkeit, 99, 100
Lebendigkeit von t , 121
Lebendigkeits-Invarianzeigenschaft, 121
Lebendigkeits-Invarianzeigenschaften, 121
Leerheitsproblem, 4, 11
lineare Vervollständigung, 68
Linie, 110
Links-Merge-Operator, 200
linkstotal, 27
locality, principle, 81
logarithmische Platzkomplexität, 234
logarithmische Zeitkomplexität, 234
logarithmisches Platzmaß, 234
logische Uhr, 73
logische Zeitstempel, 73
lokale Vektorzeit, 76
Lokalität einer Transition, 84
LTL, 41, 56

markiert, 148, 149

- Markierungs-Ausschluss, 100, 120
 Markierungs-Invarianzeigenschaft, 120
 Markierungsprädikat, 120
 merge, 250
 MIMD-Computer, 240
 Mischung, 250
 Model-Checking, 39, 51, 58
 Monoidstruktur, 2
 Multimenge, 167

 Nachrichten-Modell, 70
 Nachrichten-Synchronisation, 19, 227
 nebenläufig, 75
 nebenläufig, 65
 Nebenläufigkeit (concurrency), 82
 Netz, 84
 Netzisomorphismus, 91
 Netzmorphismus, 91
 Netzsystem, 94
 nichtriviale Zusammenhangskomponente, 59
 Nulltest, 136
 Nulltest-Kante, 136

 offene Menge, 85
 offene Menge, 85
 Operationenkomplexität, 227, 243
 optimaler paralleler Algorithmus, 228

 P-Elemente, 80
 P-Invarianten-Gleichung, 140
 P-Invarianten-Vektor, 142
 P/T-Netz, 94
 PAP, 200
 parallel random-access machine, 227
 paralleler Algorithmus, 227
 paralleles Mischen, 254
 paralleles Sortieren, 257
 Paralleloperator (merge), 199
 Parikh-Abbildung, 145
 Parikh-Vektor, 145
 partial order semantics, 65
 partielle Ordnung, 66, 75
 Pfad, 28, 122
 Platz, 84
 Platz-berandete Menge, 85
 Platz/Transitions-Netz, 94

 PO-Semantik, 65
 potenziell aktivierbar, 100
 Präfixsumme, 246
 Präzedenzgraph, 66
 Präzedenzrelation, 66
 PRAM, 240
 PRAM-Modell, 227
 Produkt-Transitionssystem, 19
 produktiv, 107
 produktive Schaltregel, 107
 Programminvariante, 139
 Prozess, 112
 Prozessgraph, 190
 Prozessorkomplexität, 227, 243

 Rücksetzzustand, 100, 122
 RAM, 229, 237
 Rand einer Menge, 85
 random-access machine, 227, 229
 Rang, 250
 rank, 250, 251
 Rechnung, 28
 reduzierter Graph, 122
 Registermaschinen, 229
 reguläre Menge, 1, 3
 regulärer Ausdruck, 3
 rekursive Spezifikation, 205
 rekursives Gleichungssystem, 205
 Rendezvous-Synchronisation, 19
 RENEW, 181
 Residuum, 163
 reversibel, 100
 Reversibilität, 99, 122
 Rucksetzzustand, 99

 Schalt-Ausschluss, 100
 Schalten, 85
 Schnitt, 110
 SIMD-Computer, 240
 Siphon, 149
 Siphon/Trap-Eigenschaft, 150
 Speicher-Synchronisation, 19
 Speicherkomplexität, 227
 Spezifikation, 51
 spontane Aktion, 212
 Sprache von α , 42
 Stelle, 84

- streng zusammenhängend, 122
strenge Zusammenhangskomponente, 122
strikte Ordnung, 75
strikte Verfeinerung, 88
Striktordnung, 66
strukturell beschränkt, 100, 144
strukturell lebendig, 100
strukturelle Eigenschaften, 100
strukturelle Nichtlebendigkeit, 100
Synchronisations-Relation, 19
Syntax von LTL-Formeln, 41
System, 51
SZK, 122

T-Element, 80
T-Invarianten-Vektor, 146
Tandempuffer, 216
temporale Logik, 39, 40
temporale Quantoren, 41
Temporallogik, 39
terminal folgenäquivalent, 15
terminale Aktionsfolgenmenge, 14
terminale Etikettensprache, 25
terminale strenge Zusammenhangskomp.,
 122
Testen und Simulation, 39
Theorembeweisen, 39
totale oder lineare Striktordnung, 66
trace-äquivalent, 26, 28
Transition, 13, 84
Transitions-berandete Menge, 85
Transitionsetikett, 25
Transitionsetikettenfunktion, 25
Transitionskalkul, 190
Transitionsregeln, 190
Transitionsregeln für Rekursion, 207
Transitionsrelation, 13
Transitionssystem, 13
Transitionssystem, endliches, 13
Transitionssysteme, etikettierte, 25
trap, 148
triviale Zusammenhangskomponente, 122

Überdeckbarkeitsproblem, 103
Überdeckungsgraph, 127
unabhängig, 75
unendlicher Pfad, 14

uniforme Platzkomplexität, 234
uniforme Zeitkomplexität, 234
Universalitätsproblem, 4, 11
Unterdrücken, 202

Validierung, 39
vektorieller Zeitstempel, 76
Vektoruhr, 76
Vektorzeit, 76
verallgemeinerter Büchi-Automat, 10
Verdecken, 202
Verfeinerung, 88
Vergrößerung, 88
Verifikation, 39
Verklemmung, 100
verklemmungsfrei, 100, 120
verschleppungsfrei, 107
verschleppungsfreie Schaltregel, 107
Verzweigungs-Bisimulation, 213
Verzweigungsprozess, 111
Vollständigkeit, 194
vorgänger-endlich, 109

wechselseitiger Ausschluss, 33, 99, 100
Wirkung, 141
Wirkungsmatrix, 141
work, 243

Zählerautomat, 135
Zeitkomplexität, 227, 242
ZK, 122
Zusammenhangskomponente, 59, 122
Zustandsetikettenfunktion, 25
Zustandsetikettensprache, 26
Zustandsexplosion, 132
Zustandsfolgen, 14