Formale Grundlagen der Informatik II

Modellierung & Analyse

von Informatiksystemen

Rüdiger Valk

Arbeitsbereich Theoretische Grundlagen der Informatik (TGI)

LV 64-060	Modul IP9
Semester	WiSe 2010
Vortragende	Rüdiger Valk, Daniel Moldt und Michael Köhler-Bußmeier

Auf dieses Skript (Stand 11. November 2010) kann unter `http://www.informatik.uni-hamburg.de/TGI/lehre/` zugegriffen werden. Der Benutzername ist „lossol“ und das Passwort lautet „capetri“ (jeweils ohne Anführungsstriche). Dort sind auch pdf-Dateien der aktuellen Vorlesungsfolien zu finden.

Einige in diesem Skript benutzte Symbole:

	klein	groß
Alpha	α	
Beta	β	
Gamma	γ	Γ
Delta	δ	Δ
Lambda	λ	Λ
Pi	π	Π
Rho	ρ	
Sigma	σ	Σ
Tau	τ	
Phi	ϕ , φ	Φ
Chi	χ	
Psi	ψ	Ψ
Omega	ω	Ω

natürliche Zahlen $\mathbb{N} = \{0, 1, 2, 3, \cdots\}$
positive natürliche Zahlen $\mathbb{N}^+ = \{1, 2, 3, \cdots\}$
ganze Zahlen $\mathbb{Z} = \{\cdots -2, -1, 0, 1, 2, \cdots\}$
rationale Zahlen $\mathbb{Q}$
reelle Zahlen $\mathbb{R}$
Zeichen „Partiell“ ∂
$|M|$ bezeichnet die Mächtigkeit einer Menge M.

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 $\mathbf{m} = (x_1, \cdots, x_n) \in V$ als Vektoren bezeichnet, wobei $V = A^B$ (z.B. $A^{\{1,\cdots,n\}}$bzw. A^n), gilt, d.h. $x_1, \cdots, x_n \in A$ und $B = \{b_1, \cdots, b_n\}$ 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 ($\mathbf{m}_1 \leq \mathbf{m}_2$ und $\mathbf{m}_1 \neq \mathbf{m}_2$) (also hier ausnahmsweise nicht komponentenweise!).

Inhaltsverzeichnis

Einleitung

Große Informatiksysteme sind von zunehmender Bedeutung in Anwendungen aller Art, gleichzeitig aber wegen der Komplexität ihres Verhaltens besonders anfällig für fehlerbehaftetes Verhalten aufgrund unpräzise angewandter Methoden. Daher sind „*formal methods*“ seit langem fester Bestandteil der Forschung und Entwicklung auf dem Gebiet von Verifikation und Semantik. Diese Lehrveranstaltung verzahnt in besonderer Weise die in den informatischen Studiengängen angebotenen Inhalte der theoretischen mit solchen der praktischen Informatik, insbesondere auch solchen die aus der Befassung mit verteilter Software entstehen. So ist diese Veranstaltung, wie alle der theoretischen Informatik einerseits stark auf die Vermittlung von Methoden ausgerichtet, muss aber andererseits zentrale Inhalte des Gebietes abdecken.

Es werden exemplarisch zwei Formalismen zur Beschreibung und Analyse von Informatiksystemen behandelt: Transitionssysteme und Petrinetze. Das erstere ist mit seinem Ursprung, den „endlichen Automaten“, das konzeptionell einfachste Modell. Es erlaubt die Darstellung vieler Erscheinungen auf anschauliche Weise, ist aber bei praktischen Anwendung ineffektiv, da die Modelle sehr schnell in ihrer Größe wachsen. Petrinetze stellen das entwickelste Gebiet dar, da grundlegende Systemeigenschaften wie Verklemmungsfreiheit, Lebendigkeit und Fairness auch stellvertretend für andere Modelle behandelt und in der Forschung ausführlich untersucht wurden.

Monographien über verteilte Systeme (z.B. Tanenbaum [uMS95], Coulouris et al.[CDK02]) behandeln zwar Verfahren zur Lösung von Problemen in verteilten Systemen in großer Vielfalt und rezeptartig, die zugrunde liegenden Algorithmen werden jedoch oft nicht ausformuliert oder gar verifiziert. Das erschwert deren Verständnis ungemein und führt zu unausgereiften Lösungen in der Praxis. Algorithmische Methoden für synchrone Mehrprozessorsysteme werden durch die behandelten parallelen Algorithmen formalisiert. Typische Vertreter sind paralleles Mischen und Sortieren und deren spezifische Komplexitätsmaße.

Die Analyse und Verifikation von Informatiksystemen wird exemplarisch anhand verschiedener Formalismen zur Systemmodellierung behandelt: Transitionssysteme, Petrinetze, Prozessalgebra und parallele Random Access Maschinen. Zur formalen Verifikation solcher Modelle wird eine ebenso formale Spezifikation benötigt, um die forderten Eigenschaften darzustellen und deren Gültigkeit im System durch geeignete Methoden nachzuweisen. Prominente Spezifikationsformalismen sind Invarianten und temporale Logik. Der Nachweis der Systemspezifikation lehnt sich an klassische mathematische Beweismethoden an oder erfolgt durch Model-Checking.

Verfahren des Model-Checking kommen der von der Industrie geforderten Bereitstellung von automatischen Verfahren entgegen. Es macht sich die konzeptionelle Einfachheit von Transitionssystemen zu Nutze, bekämpft aber die enormen Komplexitätsprobleme durch immer raffiniertere Methoden. Die Anwendung von entsprechenden Tools setzt daher einige theoretische Grundkenntnisse wie temporale Logik und Erreichbarkeitsanalyse voraus. Prozessalgebra liefert kompositionale Modellierungsansätze und führt zu effektiven Verfahren der Verifikation. Dafür werden allerdings vertiefte theoretische Kenntnisse auch beim Werkzeuganwender erforderlich.

Eine Fortführung der hier behandelten Themen (wie z.B. „verteilte Algorithmen“) wird in dem Bachelor/Master Modul: Modellierung verteilter Systeme (MVS)) angeboten.

Kapitel 1

Transitionssysteme und Verifikation

Das ursprüngliche[1] 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. Außerdem wird der Bezug zu den traditionellen endlichen Automaten und regulären Mengen aufgenommen.

1.1 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.

Definition 1.1 *Ein* Transitionssystem $TS = (S, A, tr, S^0, S^F)$ *besteht aus*

a) einer Menge S von Zuständen*,*

b) einer endlichen Menge A von Aktionen *oder* Transitionen*,*

c) einer Transitionsrelation $tr \subseteq S \times A \times S$*,*

d) einer Menge $S^0 \subseteq S$ von Anfangszuständen *und*

e) einer Menge $S^F \subseteq S$ von Endzuständen*,*

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

TS heißt endlich, *falls S endlich ist. Existiert nur ein Anfangszustand, dann schreibt man auch s_0 statt $\{s_0\}$. Wird S^F in dem Tupel weggelassen, dann sind alle Zustände Endzustände, d.h. $S^F = S$.*

Definition 1.2 *Eine* Transitionsetikettenfunktion *eines Transitionssystems $TS = (S, A, tr, S^0, S^F)$ ist eine Abbildung $E_A : A \to \Sigma \cup \{\epsilon\}$. Dabei ist Σ das Etikettenalphabet und ϵ das leere Wort* [2].

Abbildung 1.1: Getränkeautomat

Beispiel 1.3 Ein Getränkeautomat wie in Abbildung 1.1 soll nach Einwurf einer Euro-Münze je nach Wahl von Tee oder Kaffee das entsprechende Getränk ausgeben. Die Abbildung 1.2 a) zeigt ein mögliches Verhalten als Transitionssystem $TS = (S, A, tr, s_0)$ mit $S = \{s_0, \cdots, s_5\}$, $A = \{1€,\text{Tee},\text{Kaffee},\text{Tee\_kochen},\text{Kaffee\_kochen},\text{Tee\_einfüllen},\text{Kaffee\_einfüllen}\}$ und $tr = \{(s_0, 1€, s_1), (s_1, \text{Tee}, s_2), \cdots\}$. Es ist sinnvoll, $S^F = \{s_0\}$ als Endzustand zu wählen. Transitionsetiketten dienen oft dazu, interne von extern sichtbaren Aktionen zu unterscheiden. So könnte man hier $\Sigma := \{1€,\text{Tee},\text{Kaffee},\text{Tee\_einfüllen},\text{Kaffee\_einfüllen}\}$ und $[E_A(a) := \epsilon$, falls $a \in \{\text{Tee\_kochen},\text{Kaffee\_kochen}\}$ und a sonst] wählen. Wählt man statt der Aktionsfolgen die entsprechenden Bilder unter E_A, dann sind $\{\text{Tee\_kochen},\text{Kaffee\_kochen}\}$ als interne Aktionen unsichtbar. Abbildung 1.2 b) zeigt eine entsprechende Darstellung.

Endliche Automaten (Abschnitt 1.3) sind endliche Transitionssysteme. Die Semantik von Petrinetzen (Kapitel 3) und State-Charts wird ebenfalls durch Transitionssysteme[3] beschrieben, hier meist ohne Endzustände. In der Prozessalgebra (Kapitel 5) werden Transitionssysteme[3]

[2] E_A kann sinnvoll auch als Homomorphismus $E_A^* : A^* \to \Sigma^*$ aufgefasst werden.
[3] nicht notwendigerweise endlichen

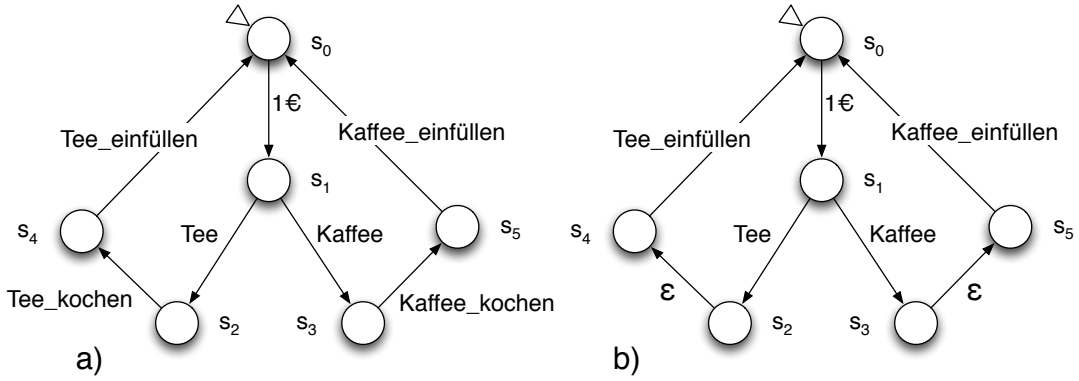


Abbildung 1.2: a) Getränkeautomat als Transitionssystem und mit internen Transitionen in b)

mit nur einem Endzustand benutzt. Beim Model-Checking (Abschnitte 1.4 und 1.5) kommen Transitionssysteme mit Endzuständen zum Einsatz, bei denen statt der Aktionen Aussagen in den Zuständen betrachtet werden.

Das dynamische Verhalten („die Semantik“) eines Transitionssystems wird durch die erreichbaren Zustände und durch seine (endlichen oder unendlichen) Transitions- bzw. Aktionsfolgen beschrieben. Statt $(s_1, a, s_1') \in tr$ wird meist die intuitivere Schreibweise $s_1 \xrightarrow{a} s_1'$ benutzt.

Definition 1.4 *Für ein Transitionssystem* $TS = (S, A, tr, S^0, S^F)$ *wird die* Transitionsrelation $s_1 \xrightarrow{a} s_1' :\Leftrightarrow (s_1, a, s_1') \in tr$ *wie folgt auf* A^* *erweitert*[4] *(dabei sei* $\epsilon \in A^*$ *das leere Wort und* $w \in A^+$, $a \in A$, $s_1, s_1' \in S$*):*

- $s_1 \xrightarrow{\epsilon} s_1$
- $s_1 \xrightarrow{wa} s_1' :\Leftrightarrow \exists s_1'' \in S : (s_1 \xrightarrow{w} s_1'') \wedge (s_1'' \xrightarrow{a} s_1')$.

Damit definieren wir

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

Entsprechend sei für Aktionsfolgen

- $AS(TS, S_1) := \{w \in A^* \mid \exists s_1 \in S_1, s_1' \in S : s_1 \xrightarrow{w} s_1'\}$ *als* Menge der von S_1 aus in TS möglichen Aktionsfolgen *und*

[4] A^* ist wie üblich die Menge aller endlichen Folgen (Wörter) über der Zeichenmenge (Alphabet) A inklusive dem leeren Wort ϵ. A^+ ist A^* ohne ϵ.

- $AS(TS) := AS(TS, S^0)$ *als* Aktionsfolgenmenge von TS *definiert.*
- $FS(TS) := \{w \in A^* \mid \exists s_1 \in S_0, s_1' \in S^F : s_1 \xrightarrow{w} s_1'\}$ *ist die* terminale Aktionsfolgenmenge von TS. *Diese Menge heißt oft auch* akzeptable Aktionsfolgenmenge *oder* akzeptierte Sprache *und wird als* $L(TS)$ *bezeichnet.* [5]
- $E_A(TS) := \{E_A^*(w) \in \Sigma^* \mid w \in FS(TS)\}$ *ist die* terminale *oder* akzeptierte Etikettensprache *von* TS. [6]

Zwei Transitionssysteme $TS_1 = (S_1, A, tr_1, S_1^0)$ *und* $TS_2 = (S_2, A, tr_2, S_2^0)$ *heißen* folgenäquivalent*, falls sie die gleiche Menge von Aktionsfolgen haben, d.* $TS_1 \sim_{AS} TS_2 :\Leftrightarrow AS(TS_1) = AS(TS_2)$

bzw. terminal folgenäquivalent *oder* akzeptanzäquivalent*, falls*

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

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 Menge S^F unendlich oft treffen muss.

Definition 1.5 *Für eine Zeichenmenge (Alphabet) A sei* A^ω *die Menge der unendlichen Folgen von Zeichen aus A und* $A^\infty := A^* \cup A^\omega$ *sei die Menge der endlichen oder unendlichen Folgen von Zeichen aus A. Eine Menge* $L \subseteq A^\omega$ *heißt* ω-Sprache. $w \in A^\omega$ *wird als* $w = a_0a_1a_2\cdots$ *dargestellt. Für eine solche Folge sei* infinite$(w) := \{a \in A | a$ *kommt in w unendlich oft vor*$\}$.

Falls für eine unendliche Folge $w = a_0a_1a_2\cdots$ *in einem Transitionssystem* $TS = (S, A, tr, S^0, S^F)$ *ein Pfad* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots$ *mit* $s_0 \in S^0$ *und* infinite$(s_0s_1s_2\cdots) \cap S^F \neq \emptyset$ *existiert, dann heißt w* akzeptiert *oder* akzeptabel. $L^\omega(TS) = \{w \in A^\omega | w$ *ist akzeptabel*$\}$ *ist die* akzeptable (oder akzeptierte) ω-Sprache *von* TS.

Beispiel 1.6 Zur Erläuterung greifen wir das Beispiel 1.3 auf, wobei zur besseren Übersicht die Transitionsbezeicher durch einfache Symbole in Abbildung 1.3 ersetzt wurden. Außerdem wurden die Transitionen $s_4 \xrightarrow{e} s_2$ und $s_5 \xrightarrow{h} s_3$ eingeführt. Sie können wie folgt interpretiert werden. In s_4 und s_5 wird geprüft, ob das Getränk heiß genug ist. Ist das nicht der Fall wird der Vorgang wiederholt. Der Zustand s_6 ist nicht für Benutzer zugänglich, sondern z.B. für das Service-Personal. Für dieses Transitionssystem $TS = (S, A, tr, s_0)$ gilt beispielsweise $R(TS) = \{s_0, s_1, s_2, s_3, s_4, s_5\}$ und $abdedfac \in AS(TS)$. Definiert man s_0 auch als Endzustand ($S^F = \{s_0\}$) dann gilt $acgiabdededf \in L(TS)$. Ein Beispiel für ein Element aus $L^\omega(TS)$ ist $abdededf(acgi)^\omega$. Dabei bedeutet $(acgi)^\omega$, dass die Folge $acgi$ unendlich oft wiederholt wird. Da bei Folgen aus $L^\omega(TS)$ der Endzustand s_0 unendlich oft durchlaufen werden muss, ist es nicht

[5] Es gilt also $FS(TS) \subseteq AS(TS)$ und $FS(TS) = AS(TS)$ falls $S^F = S$.

[6] $E_A^*(w)$ benutzt die kanonische Erweiterung $E_A^* : A^* \to \Sigma^*$ von $E_A : A \to \Sigma \cup \{\epsilon\}$ auf Wörter.

möglich, in einer Aufwärmschleife wie *de* zu bleiben, wie zum Beispiel in $acghghgiab(de)^\omega$. Dadurch kann also die Spezifikation ausgedrückt werden, dass die Aufwärmschleife immer wieder verlassen wird. Dies kann beispielsweise ein Teil der Hardwarespezifikation des Getränkeautomaten sein.

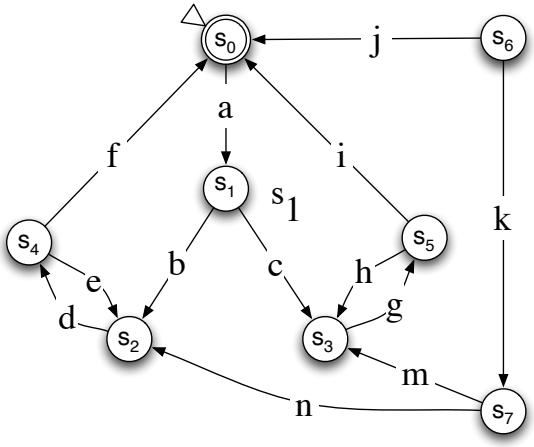


Abbildung 1.3: Getränkeautomat als Transitionssystem mit abstrakten Transitionsbezeichern

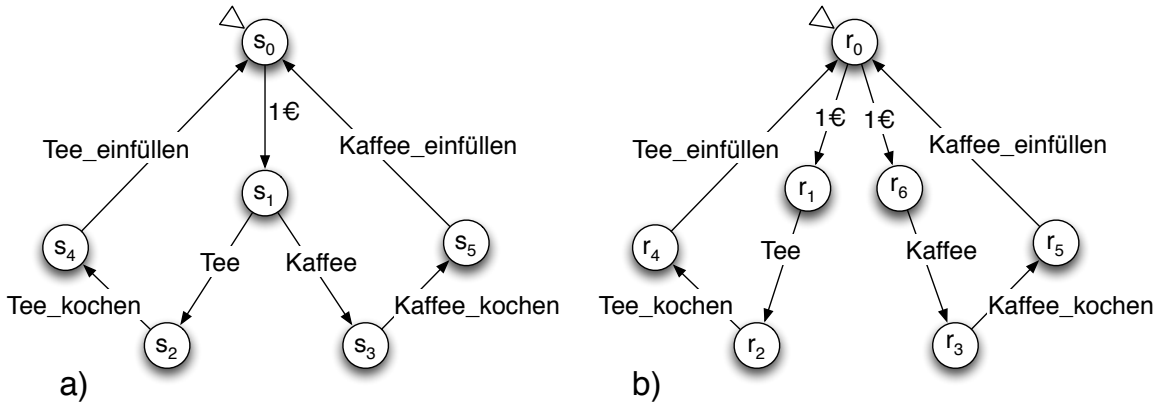


Abbildung 1.4: Zwei folgenäquivalente Transitionssysteme

Abbildung 1.4 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. Außer der im folgenden definierten *aktionsbasierten* Bisimulation wird in der Literatur auch eine *zustandsbasierte* Bisimulation

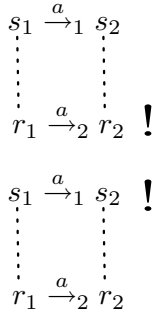
gegeben.

Definition 1.7 *Gegeben seien für $i \in \{1,2\}$ zwei Transitionssysteme $TS_i = (S_i, A, tr_i, S_i^0, S_i^F)$ mit Endzuständen (und der gleichen Aktionsmenge A)*[7]*. 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:
$s_1 \xrightarrow{a}_1 s_2 \Rightarrow \exists r_2 \in S_2 : r_1 \xrightarrow{a}_2 r_2 \wedge (s_2, r_2) \in \mathcal{B}$
$r_1 \xrightarrow{a}_2 r_2 \Rightarrow \exists s_2 \in S_1 : s_1 \xrightarrow{a}_1 s_2 \wedge (s_2, r_2) \in \mathcal{B}$

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



TS_1 und TS_2 heißen bisimilar *(in Zeichen $TS_1 \leftrightarrows TS_2$) falls eine solche Bisimulationsrelation $\mathcal{B}$ existiert. Zustände mit $(s,r) \in \mathcal{B}$ heißen* bisimilar *(in Zeichen $s \leftrightarrows r$).*

Die Bedingungen a) und c) in Definition 1.7 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 $s_1 \leftrightarrows r_1$ und $s_2 \leftrightarrows r_2$ sind durch die unterbrochenen Linien gekennzeichnet. Entsprechendes gilt für den zweiten Teil der Bedingung b) und die untere Graphik.

Wir zeigen nun, dass die (intuitiv) nicht verhaltensäquivalenten Transitionssysteme aus Abbildung 1.4 tatsächlich nicht (formal) bisimilar sind. Wären sie bisimilar, dann müssten wegen der Bedingung a) die Anfangszustände in der Relation stehen: $s_0 \leftrightarrows r_0$. Aus der ersten Bedingung b) folgt dann, dass mit $a = 1€$ auch $s_1 \leftrightarrows r_1$ oder $s_1 \leftrightarrows r_6$ 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 1.5 bisimilar. Die Bisimulations-Relation ist $\mathcal{B} = \leftrightarrows = \{(s_i, r_i) | i = 0, \cdots, 5\} \cup \{(s_1, r_6)\}$.

Satz 1.8 *Wenn zwei Transitionssysteme $TS_i = (S_i, A, tr_i, S_i^0, S_i^F)$ mit Endzuständen (und der gleichen*[7] *Aktionsmenge A und $i \in \{1,2\}$) bisimilar sind, dann sind sie auch akzeptanzäquivalent, aber nicht umgekehrt,*
d.h.: $TS_1 \leftrightarrows TS_2 \ \Rightarrow\ TS_1 \sim_L TS_2$).

[7] Das ist keine echte Einschränkung, da dies durch Obermengenbildung immer zu erreichen ist.

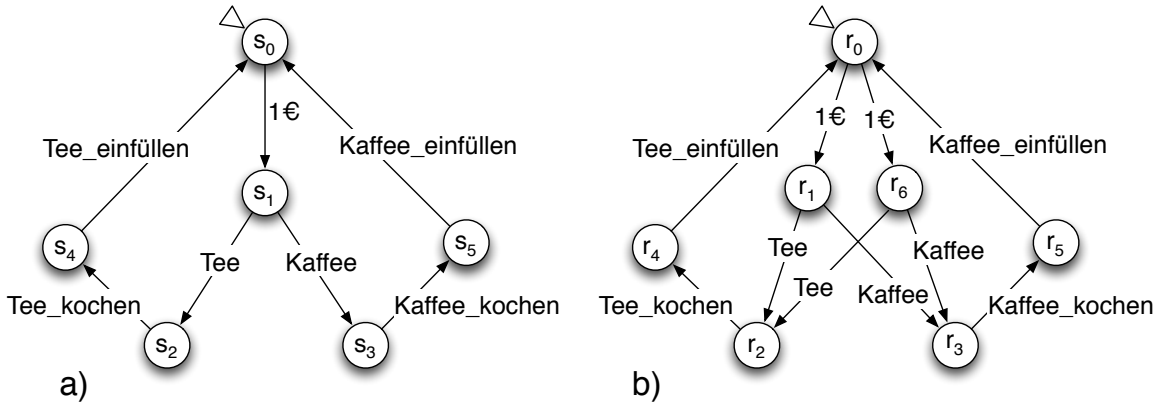


Abbildung 1.5: Zwei bisimilare Transitionssysteme

Anmerkung: Da $S_1^F = S_1$ und $S_2^F = S_2$ als Spezialfall enthalten ist, gilt auch
$TS_1 \leftrightarrows TS_2 \;\Rightarrow\; AS(TS_1) = AS(TS_2)$

Beweis:
Sei $TS_1 \leftrightarrows TS_2$. Wir beweisen zunächst $(s_0 \xrightarrow{w}_1 s_1 \;\wedge s_0 \leftrightarrows r_0) \;\Rightarrow\; (\exists r_1 : r_0 \xrightarrow{w}_2 r_1 \wedge s_1 \leftrightarrows r_1)$ 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 $s_0 \xrightarrow{va}_1 s_2$ und $s_0 \leftrightarrows r_0$. Dann gibt es ein $s_1 \in S_1$ mit $s_0 \xrightarrow{v}_1 s_1 \xrightarrow{a}_1 s_2$. Nach Induktionsvoraussetzung gibt es einen Zustand r_1 mit $r_0 \xrightarrow{v}_2 r_1$ und $s_1 \leftrightarrows r_1$. Wegen Definition 1.7 b) gibt es r_2 mit $r_1 \xrightarrow{a}_2 r_2$ und $s_2 \leftrightarrows r_2$. Also gilt $r_0 \xrightarrow{va}_2 r_2$ und $s_2 \leftrightarrows r_2$.

$$\begin{array}{ccccc} s_0 & \xrightarrow{v}_1 & s_1 & \xrightarrow{a}_1 & s_2 \\ \vdots & & \vdots & & \vdots \\ r_0 & \xrightarrow{v}_2 & r_1 & \xrightarrow{a}_2 & r_2 \;! \end{array}$$

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 $s_0 \xrightarrow{w}_1 s_1$. Also existiert nach Definition 1.7 a) ein $r_0 \in S_2^0$ mit $s_0 \leftrightarrows r_0$. Mit der vorher bewiesenen Zwischenbehauptung gibt es $r_1 \in S_2$ mit $r_0 \xrightarrow{w}_2 r_1$ und $s_1 \leftrightarrows r_1$. Nach Definition 1.7 c) muss auch $r_1 \in S_2^F$ gelten. Also erhalten wir $w \in L(TS_2)$. □

Damit folgt, dass die beiden bisimilaren Transitionssysteme von Abbildung 1.5 auch akzeptanzäquivalent sind (z.B. mit $S_1^F = \{s_0\}$ und $S_2^F = \{r_0\}$).

1.2 Produkte von Transitionssystemen

Transitionssysteme können synchronisiert werden. Dadurch erhält 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 synchronisiernde 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 *Produkttransitionssystem* genannt wird. Die zu synchronisierenden Transitionspaare werden durch eine Relation $Sync \subseteq A_1 \times A_2$ bestimmt. Je nach dem Umfang dieser Relation ist das resultierende Produkttransitionssystem 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 eingestzt wird, welcher den Durchschnitt bzw. die Vereinigung regulärer Mengen akzeptiert.

Definition 1.9 *Gegeben seien zwei Transitionssysteme* $TS_i = (S_i, A_i, tr_i, S_i^0, S_i^F)$ $(i \in \{1,2\})$ *(mit nicht notwendig gleichen Aktionsmengen* A_i*). Das* Produkttransitionssystem *von* TS_1 *und* TS_2 *wird als Transitionssystem*
$TS_1 \otimes_\gamma TS_2 = (S_1 \times S_2, A_1 \cup A_2 \cup Sync, tr_3, S_1^0 \times S_2^0, S_1^F \times S_2^F, \gamma)$ *definiert, wobei*

a) $Sync \subseteq A_1 \times A_2$ *eine* Synchronisations-Relation*,*

b) $\gamma : Sync \to A_3$ *eine Abbildung von Sync in ein* Kommunikationsalphabet[8] A_3 *ist und*

c) die Transitionsrelation tr_3 *durch folgende Regeln* $Sy1$*,* $Sy2$ *und* $Sy3$ *definiert werden.*

Die Kommunikationsabbildung γ *ist optional.*

Regel $Sy1$*:* Erste Komponente

$$\frac{s_1 \xrightarrow{a_1}_1 s_2, \quad a_1 \notin pr_1(Sync), \quad r \in S_2}{(s_1, r) \xrightarrow{a_1}_3 (s_2, r)}$$

[8] Es sei A_3 disjunkt zu A_1 und A_2.

Regel Sy2: Zweite Komponente

$$\frac{r_1 \xrightarrow{a_2}_2 r_2, \quad a_2 \notin pr_2(Sync), \quad s \in S_1}{(s, r_1) \xrightarrow{a_2}_3 (s, r_2)}$$

Regel Sy3: Kommunikation

$$\frac{s_1 \xrightarrow{a_1}_1 s_2, \quad r_1 \xrightarrow{a_2}_2 r_2, \quad (a_1, a_2) \in Sync}{(s_1, r_1) \xrightarrow{(a_1,a_2)}_3 (s_2, r_2)}$$

Dabei ist $pr_1(Sync) := \{s | \exists r \in A_2 : (s, r) \in Sync\}$ die Menge der ersten Komponenten (1. Projektion) und $pr_2(Sync)$ entsprechend definiert. Statt (a_1, a_2) kann in Regel Sy3 auch $\gamma(a_1, a_2)$ eingesetzt werden (sofern definiert). Statt $TS_1 \otimes_\gamma TS_2$ schreiben wir auch kürzer $TS_1 \otimes TS_2$, wenn Sync und γ aus dem Kontext ersichtlich sind.

Beispiel 1.10 (einfaches Sender-/Empfänger-System)
Das Beispiel von Abbildung 1.6 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-Taransitionssystem $S \otimes R$ 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 $(s_1, r_0) \xrightarrow{c_B(d)}_3 (s_0, r_1)$ 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 1.6 rechts unten). Diese Konstruktion entspricht der Beseitigung von ϵ-Übergängen, wie sie für Automaten in FGI 1 (Kap 14) behandelt wurde.

Beispiel 1.11 (gestörtes Sender-/Empfänger-System)
Um ein nicht korrektes Verhalten im vorstehenden Beispiel zu zeigen, soll nun der interne Kanal B gestört sein (Abb. 1.7). Die Störung wird dadurch dargestellt, dass der Sender das Fehlersignal $\perp$ in den Kanal schreibt. Hier ist $Sync = \{(s_B(d), r_B(d)), (s_B(\perp), r_B(\perp))\}$ 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 Beipiel die fehlerhafte Folge $r_A(d)s_C(d)r_A(d)s_C(\perp)r_A(d)s_C(d)$.

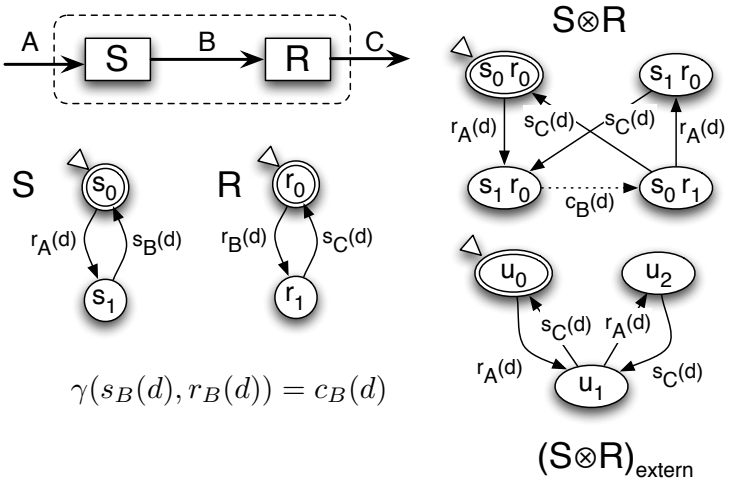


Abbildung 1.6: Ein einfaches Sender-Empfänger-System

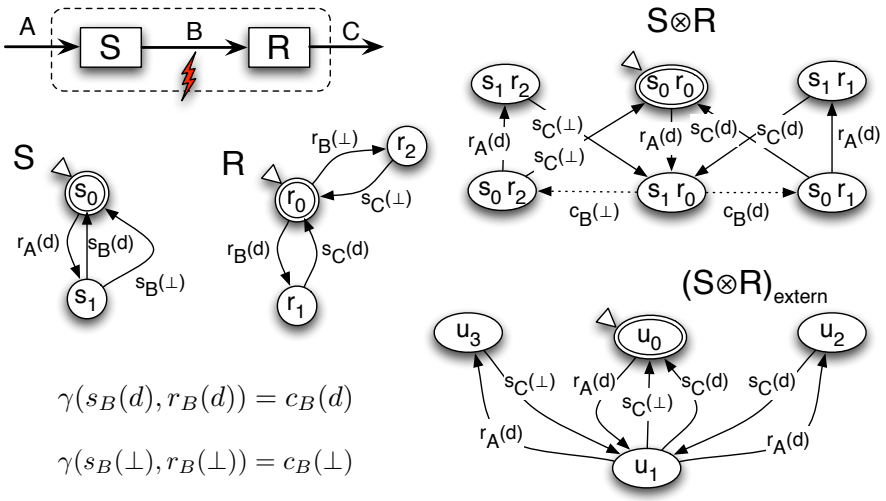


Abbildung 1.7: Ein gestörtes Sender-Empfänger-System

Beispiel 1.12 (entstörtes Sender-/Empfänger-System)
Wir entwickeln im gleichen Formalismus das „Alternierbitprotokoll“. Im ersten Schritt wird eine Bestätigung über einen ungestörten Kanal D zurückgesandt (Abbildung 1.8). 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. Dies geht in die Spezifikation des Systems als „Fairness-Bedingung“ ein. Formal wird dies dadurch ausgedrückt, dass immer der Endzustand erreicht werden muss oder (bei unendlichen Prozessen) immer wieder durchlaufen werden muss.

Beispiel 1.13 (Das Alternierbitprotokoll)
Die Spezifikation des Alternierbitprotokolls geht davon aus, dass auch der Bestätigungskanal 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 1.9. Erhält der Sender das Bit mit dem selben Wert (in den Zuständen s_3 oder s_0) zurück, dann kann er das nächtes 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 $\Delta = \{d_1, \cdots, d_k\}$ 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 1.13, Abbildung 1.9) 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.

1.3 Automaten und reguläre Sprachen

Transitionssysteme haben in Anwendungen meist eine endliche Zustandsmenge. Endliche Transitionssysteme $TS = (S, A, tr, S^0, S^F)$ sind endliche akeptierende nichtdeterministische Automaten (NFA), wie sie in der Vorlesung FGI 1 in der Form $\mathcal{A} = (Q, \Sigma, \delta, S_0, F)$ behandelt wurden. Ihre akzeptierten Sprachen $L(TS)$ bzw. $L(\mathcal{A})$ sind die regularen Sprachen, deren Definition wir hier wiederholen und auf Mengen von unendlichen Wörtern erweitern.

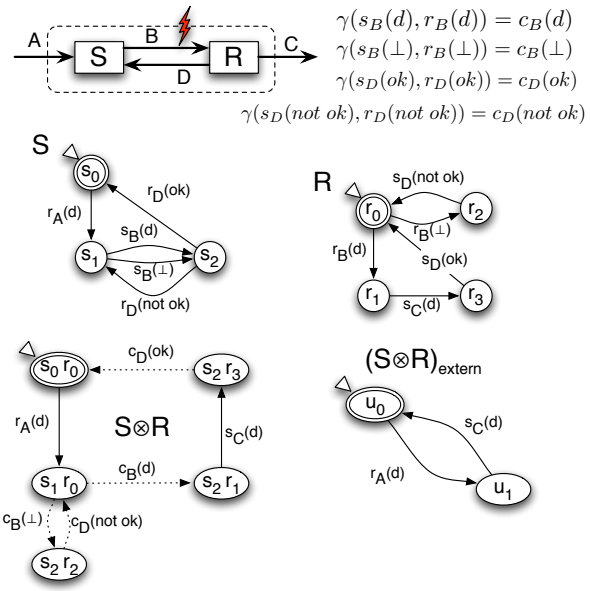


Abbildung 1.8: Entstörtes Sender-Empfänger-System

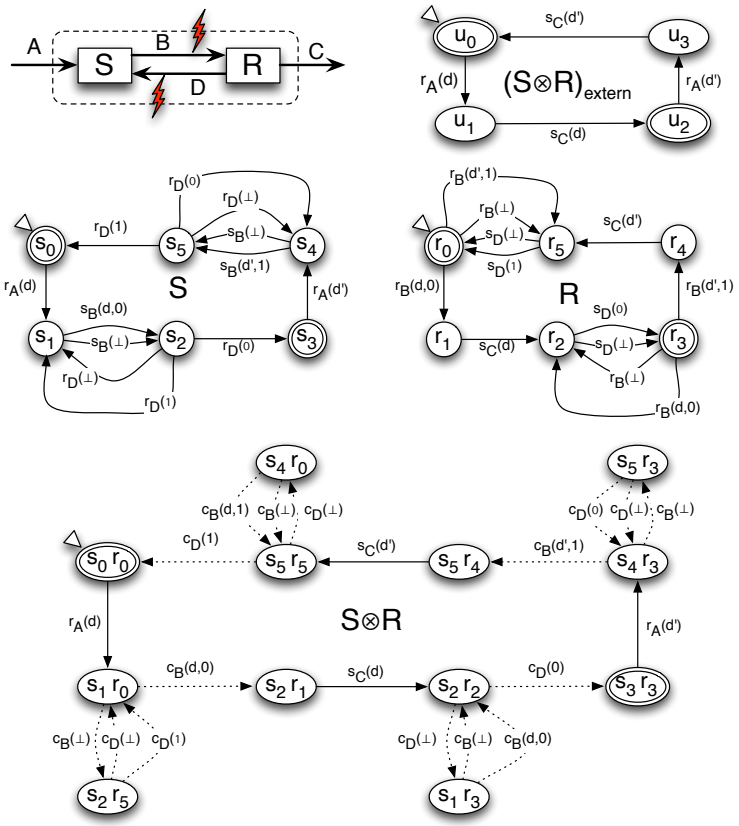


Abbildung 1.9: Das Alternierbitprotokoll

Definition 1.14 *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^*,*
   - *A^+ beschreibt die Menge M_A^+.*
4. *Nur die mit 1., 2. und 3. erzeugten Ausdrücke sind reguläre Ausdrücke. Die dadurch beschriebenen Mengen heißen* reguläre Mengen *oder Sprachen über Σ.*

Wir definieren nun das Analogon für unendliche Wörter.

Definition 1.15 *Für ein (endliches) Wort $w = a_1 a_2 \cdots a_n \in \Sigma^*$ und ein (unendliches) Wort $u = b_1 b_2 \cdots \in \Sigma^\omega$ sei $w \cdot u := a_1 a_2 \cdots a_n b_1 b_2 \cdots \in \Sigma^\omega$ die Konkatenation dieser Wörter und entsprechend für Sprachen $W \subseteq \Sigma^*$ und $U \subseteq \Sigma^\omega$ sei $W \cdot U := \{w \cdot u | w \in W, u \in U\} \subseteq \Sigma^\omega$. Ferner sei für $W \subseteq \Sigma^*$ der ω-Abschluss $W^\omega := \{w_1 \cdot w_2 \cdot w_3 \cdots | w_i \in W \backslash \{\epsilon\}, i \geq 1\}$ gegeben. Jede Menge $U \subseteq \Sigma^\omega$ heißt ω-Sprache („Omega-Sprache“).*

Ein Ausdruck der Form $G = A_1 \cdot B_1^\omega + \cdots + A_n \cdot B_n^\omega$ heißt ω-regulärer Ausdruck *über dem Alphabet Σ, falls $A_1, \cdots, A_n, B_1, \cdots, B_n$ reguläre Ausdrücke über Σ sind und kein M_{B_i} das leere Wort ϵ enthält. Dieser Ausdruck beschreibt die* ω-reguläre Sprache *$M_G := M_{A_1} \cdot M_{B_1}^\omega \cup \cdots \cup M_{A_n} \cdot M_{B_n}^\omega$.*

Beispiel 1.16 Der endliche Automat TS von Abbildung 1.3 akzeptiert die Sprache $L(TS) = (a(bd(ed)^* f + cg(hg)^* i))^*$. Sei dieser Ausdruck im Folgenden C genannt und wählt man s_6 als neuen Anfangszustand (s_0 sei weiterhin der einzige Endzustand), dann ist $L^\omega(TS) = knd(ed)^* fC^\omega + kmg(hg)^* iC^\omega + jC^\omega$.

Dieses Beispiel illustriert auch den nächsten Satz, der die Äquivalenz von regulären Sprachen und von endlichen Automaten (oder Transitionssystemen) akzeptierten Sprachen formuliert. Ein entsprechendes Ergebnis gilt auch für unendliche Wörter.

Satz 1.17 *a) Die durch endliche Transitionssysteme $TS = (S, \Sigma, tr, S^0, S^F)$ akzeptierten Sprachen $L(TS)$ sind genau die regulären Sprachen über Σ.*

b) Die durch endliche Transitionssysteme $TS = (S, \Sigma, tr, S^0, S^F)$ akzeptierten ω-Sprachen $L^\omega(TS)$ sind genau die ω-regulären Sprachen über Σ.

Der Beweis von Teil a) erfolgt in einer Richtung dadurch, dass gemäß der induktiven Definition von regulären Sprachen nichtdeterministische endliche Automaten (*NFA*s) 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. Der Beweis für ω-Sprachen in Teil b) ist ähnlich und in [BK08] zu finden. Von dort (Seite 175) stammt auch das Transitionssystem von Abbildung 1.10, das die ω-Sprache $L^\omega(TS) = c^*ab(b^+ + bc^*ab)^\omega$ akzeptiert.

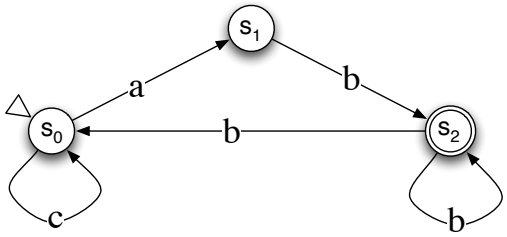


Abbildung 1.10: Transitionssystem TS mit ω-Sprache $L^\omega(TS) = c^*ab(b^+ + bc^*ab)^\omega$

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 1.18 *Gegeben seien für $i \in \{1,2\}$ zwei Transitionssysteme $TS_i = (S_i, A_i, tr_i, S_i^0, S_i^F)$ mit Endzuständen. Dann können Transitionssyteme TS_3 und TS_4 effektiv konstruiert werden, die den Durchschnitt der akzeptierten Sprachen akzeptieren:*

a) $L(TS_3) = L(TS_1) \cap L(TS_2)$ *und*

b) $L^\omega(TS_4) = L^\omega(TS_1) \cap L^\omega(TS_2)$.

Beweis:
Seien oBdA. $L(TS_1) \neq \emptyset \neq L(TS_2)$ und $L^\omega(TS_1) \neq \emptyset \neq L^\omega(TS_2)$.

a) Wir definieren TS_3 als das Transitionssystem $TS_1 \otimes_{Sync} TS_2$ mit $Sync = \{(a,a) | a \in A_1 \cap A_2\}$ und $\gamma(a,a) = a$ für $a \in A_1 \cap A_2$ und streichen alle Transitionen $(s_1, r_1) \xrightarrow{a} (s_2, r_2)$ mit $a \notin A_1 \cap A_2$, d.h. bei der Produktdefinition von Definition 1.9 auf Seite 10 kommt nur die Regel $Sy3$ zur Anwendung.

Es sei nun $w \in L(TS_1) \cap L(TS_2)$ und $w = a_1 a_2 \cdots a_n$, $n \geq 1$. Dann gilt $s_0 \xrightarrow{a_1}_1 s_1 \xrightarrow{a_2}_1 s_2 \cdots s_{n-1} \xrightarrow{a_n}_1 s_n$ mit $s_0 \in S_1^0$, $s_n \in S_1^F$ und $r_0 \xrightarrow{a_1}_2 r_1 \xrightarrow{a_2}_2 r_2 \cdots r_{n-1} \xrightarrow{a_n}_2 r_n$ mit $r_0 \in S_2^0$, $r_n \in S_2^F$. Dann ist auf Grund der Regel $Sy3$ auch $(s_0, r_0) \xrightarrow{a_1}_3 (s_1, r_1) \xrightarrow{a_2}_3$

$(s_2, r_2) \cdots (s_{n-1}, r_{n-1}) \xrightarrow{a_n}_3 (s_n r_n)$ mit $(s_0, r_0) \in S_1^0 \times S_2^0$ und $(s_n, r_n) \in S_1^F \times S_2^F$ und somit $w \in L(TS_3)$. Der umgekehrte Schluss erfolgt entsprechend.

b) Für diesen Teil des Beweises konstruieren wir TS_4 zunächst wie TS_3. Ist dann $w = a_1 a_2 \cdots \in L^\omega(TS_3)$, also $(s_0, r_0) \xrightarrow{a_1}_3 (s_1, r_1) \xrightarrow{a_2}_3 (s_2, r_2) \xrightarrow{a_3}_3 \cdots$ mit $(s_0, r_0) \in S_1^0 \times S_2^0$ und $(s_i, r_i) \in S_1^F \times S_2^F$ für unendlich viele $i \geq 1$, dann gilt entsprechendes auch für die beiden Komponenten und es ist $w \in L^\omega(TS_1) \cap L^\omega(TS_2)$.

Der Umkehrschluss ist jedoch so nicht möglich, da ausgehend von $w \in L^\omega(TS_1) \cap L^\omega(TS_2)$ und $w = a_1 a_2 \cdots$ mit $s_0 \xrightarrow{a_1}_1 s_1 \xrightarrow{a_2}_1 s_2 \cdots$ mit $s_0 \in S_1^0$, $s_i \in S_1^F$ für unendlich viele $i \geq 1$ und $r_0 \xrightarrow{a_1}_2 r_1 \xrightarrow{a_2}_2 r_2 \xrightarrow{a_n} \cdots$ mit $r_0 \in S_2^0$, $r_j \in S_2^F$ für unendlich viele $j \geq 1$ auf Grund der Regel *Sy3* auch wieder $(s_0, r_0) \xrightarrow{a_1}_3 (s_1, r_1) \xrightarrow{a_2}_3 (s_2, r_2) \xrightarrow{a_3}_3 \cdots$ mit $(s_0, r_0) \in S_1^0 \times S_2^0$ existiert, die s_i und r_j nicht für die gleichen i und j unendlich oft in einem Endzustand sein müssen! Daher konstruieren wir TS_4 aus TS_3, indem wir die Zustände (s, r) um eine Komponente $q \in \{1, 2\}$ erweitern. $(s, r, 1)$ bedeutet dabei „TS_4 wartet auf einen Zustand $(s, r, 1)$ mit $s \in S_1^F$“ und entsprechend bedeutet $(s, r, 2)$ „TS_4 wartet auf einen Zustand $(s, r, 2)$ mit $r \in S_2^F$“. Auf Grund der Definition von $w \in L^\omega(TS_1)$ und $w \in L^\omega(TS_2)$ muss dieses Ereignis eintreten und TS_4 wechselt dann den Wert von q auf den jeweils anderen Wert. Auf diese Weise werden abwechselnd Zustände (s, r, q) mit $s \in S_1^F$ und $r \in S_2^F$ unendlich oft eingenommen und es gilt mit einer entsprechenden Wahl von S_4^F die gewünschte Beziehung $w \in L^\omega(TS_4)$.

Formal definieren wir $TS_4 = (S_4, A_1 \cap A_2, tr_4, S_4^0, S_4^F)$ durch

a) $S_4 = \{(s, r, q) | (s, r) \in S_3, q \in \{1, 2\}\}$

b) $(s, r, q) \xrightarrow{a}_4 (s', r', q') :\Longleftrightarrow (s, r) \xrightarrow{a}_3 (s', r') \ \wedge \ q' := \begin{cases} 2 & \text{falls } q = 1 \wedge s \in S_q^F \\ 1 & \text{falls } q = 2 \wedge r \in S_q^F \\ q & \text{sonst} \end{cases}$

c) $S_4^0 := \{(s, r, 1) | (s, r) \in S_3^0 = S_1^0 \times S_2^0\}$

d) $S_4^F := \{(s, r, 1) | s \in S_1^F\}$

Wie bei TS_3 wird auch jedes $w \in L^\omega(TS_4)$ sowohl in TS_1 als auch in TS_2 akzeptiert. □

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 Korrektkeit eines Systems besonders wichtig, weil ein Fehlverhalten unverantwortlich oder zu teuer ist, werden Verifikationsmethoden angewandt. Eine davon ist das *Model-Checking*. Bei dieser Methode wird das *System* durch ein Transitionssystem TS_{sys} modelliert und die *Spezifikation* durch eine (temporal-)logische Formel f_{spec} festgelegt. Letztere wird häufig in ein äquivalentes Transitionssystem TS_{spec} umgewandelt (siehe Satz 1.37 auf

Seite 36) oder die Spezifikation wird direkt durch TS_{spec} festgelegt. Zu prüfen ist dann, ob alle Transitionsfolgen des Systems auch solche der Spezifikation sind, d.h.

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

oder

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

Um dies zu prüfen, wird die Beziehung $P \subseteq Q \Leftrightarrow P \cap (R \backslash Q) = \emptyset$ ausgenutzt, die für alle Mengen $P \subseteq R$ und $Q \subseteq R$ gilt. Wenn A die Grundmenge der Aktionen ist, bleibt also zu prüfen, ob

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

oder

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

gilt. Dazu wird 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 zur Größe des Transitionssystems linearer Zeit 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 $A^* \backslash L(TS_{spec})$ zu akzeptieren, muss dann nur im vervollständigten[9] Automaten die Endzustandsmenge komplementiert werden.
4. $A^{\omega} \backslash L^{\omega}(TS_{spec})$ zu erhalten, ist schwieriger. Ist die Spezifikation jedoch durch eine Formel f_{spec} gegeben, dann kann man jedoch einfach zur Negation $\neg f_{spec}$ übergehen und zu dieser Formel den akzeptierenden Automaten[10] konstruieren, der $A^* \backslash L(TS_{spec})$ oder $A^{\omega} \backslash L^{\omega}(TS_{spec})$ akzepiert.

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. Wir erläutern dies an dem fehlerhaften System von Beispiel 1.11 (Abbildung 1.7, Seite 12).

Beispiel 1.19 (Model Checking)

Abbildung 1.11 zeigt das externe Verhalten des gestörten Sender-Empfänger-Systems von Abb. 1.7 als Transitionssystem TS_{sys} (in Abb. 1.7 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.

[9] siehe FGI 1
[10] siehe Satz 1.37

Das Transitionssystem $\overline{TS}_{spec}$ in der Mitte rechts akzeptiert $A^* \backslash L(TS_{spec})$. 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.

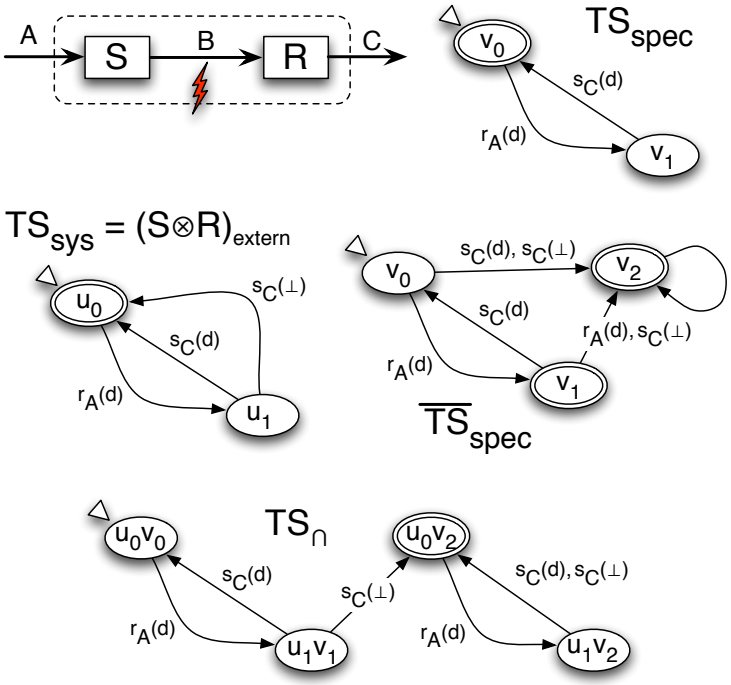


Abbildung 1.11: Model Checking für das modifizierte gestörte Sender-Empfänger-System.

1.4 Kripkestrukturen

1.4.1 Verifikation und Model-Checking

Validierung von Hardware- und Softwaresystemen wird zunehmend wichtig.

Unter **System-Validierung** versteht man:

a) traditionell: **Testen und Simulation**

- am Anfang bei einfachen Fehlern sehr wirksam,
- aber immer weniger effektiv, wenn komplexe und verborgene Fehler auftreten,
- insbesondere bei Systemen mit Asynchronität, Parallelität, Nebenläufigkeit,
- Fehler oft von speziellen Zeitparametern/Nachrichtenlaufzeiten abhängig.

b) alternativ: **formale Verifikation**
- umfassende Prüfung,
- alle Zweige des Verhaltens werden geprüft.

Wichtige Vorgehensweisen der formalen Verifikation sind:

a) Axiomatische Semantik (Hoare-Systeme, Zusicherungen, Invarianten)

b) temporale Logik

c) Analyse des Zustandsraumes (Model-Checking)

a) wird in der Vorlesung *Semantik von Programmsprachen* (Modul FGI 3) behandelt. Eine Einführung zu b) und c) wird in diesem Kapitel gegeben.

Model-Checking (auch **Zustandsraum-Analyse**) hat folgende Vor- und Nachteile:

Vorteile:

- ohne besondere Kenntnisse anwendbar,
- bei nicht korrektem System: Generierung von Abläufen, die zu den Fehlern führen.

Nachteile:

- Größe des Zustandsraumes,
- Ausdrucksfähigkeit der Spezifikationssprache

Abhilfe:

- *symbolisches* Model Checking
- *Faltung*
- Ausnutzung von Symmetrien
- Übersetzer für Spezifikationssprachen

Model Checking heißt *Analyse des Zustandsraumes*. Zustandsräume sind für praktisch alle Systemmodelle in natürlicher Weise gegeben.

Beispiele:

System	*Zustandsraum*
Programm	*Zustandsgraph*
Schaltkreis	*Zustandsgraph*
Statechart	*Transitionssystem*
Prozessausdruck	*Transitionssystem*
Petrinetz	*Markierungsgraph* = *Erreichbarkeitsgraph*

Literatur:

E.M. Clarke et al.: *Model Checking*, The MIT Press, Cambridge, 1999, [CGP99]

B. Bèrard 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]

C. Baier, J.-P. Katoen: *Principles of Model Checking*, MIT Press, 2008, [BK08]

1.4.2 Transitionssysteme in Form von 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. Im Gegensatz zu der vorher in diesem Kapitel betrachteten *Transitions*-Etikettenfunktion wird daher hier eine *Zustands*-Etikettenfunktion benutzt.

Definition 1.20 *Sei $TS = (S, A, tr, S^0, S^F)$ ein Transitionssystem. Eine* Zustandsetikettenfunktion *ist eine Abbildung $E_S : S \to \mathcal{P}(AP)$, wobei AP eine Menge von atomaren Aussagen ist.*

Beispiel 1.21 Durch eine Zustandsetikettenfunktion können den Zuständen Mengen von atomaren Aussagen zugeordned werden, die in diesem Zustand gültig sind. Beispielsweise könnte man im Transitionssystem von Abbildung 1.2 b) $AP := \{\alpha_1, \alpha_2, \alpha_3\}$ durch $\alpha_1 =$ „1€ gezahlt“,

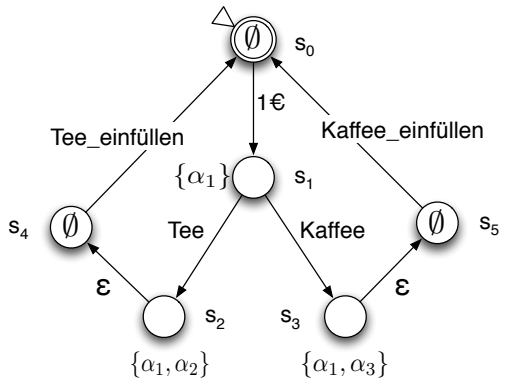


Abbildung 1.12: Getränkeautomat als Transitionssystem mit Zustandsetikettenfunktion

α_2 = „Tee gewählt“ und α_3 = „Kaffee gewählt“ definieren. Die Zuordnung $E_S(s_0) = \emptyset$, $E_S(s_1) = \{\alpha_1\}$ usw. ist in Abbildung 1.12 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.

Definition 1.22 *(Kripke-Struktur 1)*

Eine Kripke-Struktur *ist ein endliches Transitionssystem* $M := (S, A, tr, S_0, \{\})$[11] *mit einelementiger Aktionenmenge (also* $|A| = 1$*), linkstotaler*[12] *Transitionsrelation* tr *und zusätzlich einer Zustandsetikettenfunktion* $E_S : S \to \mathcal{P}(AP)$.

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 Relation $R \subseteq S \times S$ dargestellt. Sie ist linkstotal (siehe Fußnote „linkstotal“), damit jede endliche Zustandsfolge Anfangsstück einer unendlichen Folge ist, d.h. keine „Sackgassen“ entstehen können.

Definition 1.23 *(Kripke-Struktur 2)*

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

a) einer endlichen Zustandsmenge S*,*

b) einer Menge $S_0 \subseteq S$ *von Anfangszuständen,*

c) einer linkstotalen (siehe Fußnote) Transitionsrelation $R \subseteq S \times S$ *und*

[11] D.h es gibt keine Endzustände.
[12] Linkstotal bedeutet: $\forall s \in S\ \exists s' \in S : (s, s') \in R$

d) einer Zustandsetikettenfunktion $E_S : S \to \mathcal{P}(AP)$, die jedem Zustand s eine Menge $E_S(s) \subseteq AP$ von aussagenlogischen atomaren Formeln zuordnet (die in diesem Zustand gelten).

Beispiel 1.24 $E_S(3) = \{\alpha_1, \alpha_2\}$ in der Kripke-Struktur von Abb. 1.12 oder $E_S(3) = \{\neg Start, Close, \neg Heat, \neg Error\}$ in der Kripke-Struktur von Abb. 1.16.

Anmerkung: In Darstellungen und in Modelchecking-Werkzeugen werden zur besseren Lesbarkeit oft Transitionsbezeichner benutzt (wie in den Abbildungen 1.12 und 1.16). Diese gehören aber nicht zur formalen Definition.

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

Definition 1.25 *Für eine Kripke-Struktur $M := (S, S_0, R, E_S)$ sei ein* Pfad *oder eine* Rechnung *aus $s \in S$ eine unendliche Folge $\pi = s_0 s_1 s_2 \ldots \in S^\omega$ mit $s_0 = s$ und $\forall i \geq 0 : (s_i, s_{i+1}) \in R$. $SS(M) := \{\pi | \exists s \in S^0 : \pi$ ist eine Rechnung aus $s\}$ ist die Menge aller Pfade von M. Die zu π zugehörige Zustandsetikettenfolge ist dann $E_S(\pi) = E_S(s_0)E_S(s_1)E_S(s_2)\ldots \in \mathcal{P}(AP)^\omega$. Zwei Kripke-Strukturen heißen* trace-äquivalent*, wenn die Mengen ihrer Zustandsetikettenfolgen gleich sind.*

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

Zustand: $s : V \to D$ 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. $\mathcal{S}_0$ 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 1.26 *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 Anfangsbedingung $\mathcal{S}_0$ erfüllen,*

3. *$\forall s, s' \in S : R(s, s') \leftrightarrow \mathcal{R}(V, V')$ mit Belegung s für V und s' für V',*

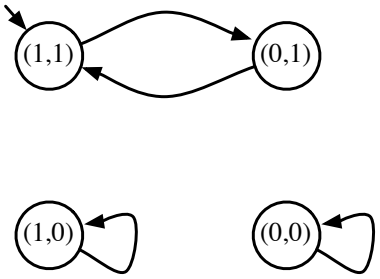


Abbildung 1.13: Kripke-Struktur zu Beispiel 1.27

4. *$E_S(s)$ Menge der Formeln, die bei Belegung s gelten.* *($v \in E_S(s)$ gdw. $s(v) = wahr$, $v \notin E_S(s)$ gdw. $s(v) = falsch$)*

Beispiel 1.27 (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:

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

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

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

$S = D \times D$,

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

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

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

Ein Pfad vom Anfangszustand: (1, 1), (0, 1), (1, 1), (0, 1), ...

1.4.3 Kripke-Strukturen von Programmen

Sequentielle Programme als Kripke-Strukturen

Exemplarisch für Programme werden nun Kripke-Strukturen für eine einfache Prgrammsprache definiert, die die elementaren Anweisungen **Zuweisung**, **Skip** (d.i. die leere Anweisung), **Hintereinaderausführung**, **bedingte Anweisung** und **Schleife** enthält.

Zur Kennzeichnung von Zuständen erhalten die Programme Zeilennummern.

Für Zeilennummern 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 Kripkestruktur.

l, l' Zeilennummer jeweils vor und nach der Anweisung

pc, pc' Befehlszähler - Variable

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

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

Hintereinanderausführung: [13]
$\mathcal{C}(l,\ (P_1; l'' : P_2),\ l') \equiv \mathcal{C}(l, P_1, l'') \vee \mathcal{C}(l'', P_2, l')$

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

Schleifen-Anweisung:
$\mathcal{C}(l,\ \textbf{while}\ b\ \textbf{do}\ l_1 : P_1\ \textbf{endwhile},\ l') \equiv$
$(pc = l \wedge pc' = l_1 \wedge b \wedge same(V)) \vee$
$(pc = l \wedge pc' = l' \wedge \neg b \wedge same(V)) \vee$
$\mathcal{C}(l_1, P_1, l)$

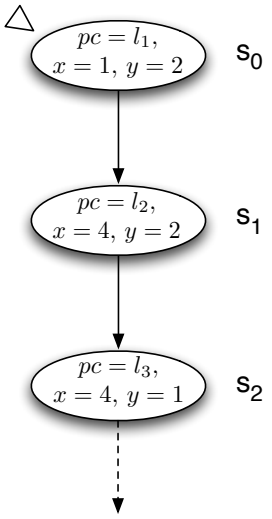


Abbildung 1.14: Anfang einer Kripke-Struktur zum Programm von Beipiel 1.28

Beispiel 1.28 (ein kleines sequentielles Programm)

Wir bilden die Formeln für die Zuweisungsfolge $l_1 : x := 2 \cdot y;\ l_2 : y := y - 1;\ l_3\ \cdots$ mit $V = \{x, y\}$ und dem Anfangswert $x = 1, y = 2$.

Anfangszustand: $\mathcal{S}_0 \equiv (x = 1 \wedge y = 2 \wedge pc = l_1)$

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 Kripkestruktur generiert:

Anfangsschritt: Bilde alle Anfangszustände, die $\mathcal{S}_0$ erfüllen. Das können mehrere sein. In unserem Beispiel (siehe Abbildung 1.14) 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 ist[14]. 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 $\{(s, s')\} = \{((pc, x, y), (pc', x', y'))\}$.

[13]Das innere Klammerpaar dient der besseren Lesbarkeit.

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

Nebenläufige Programme als Kripke-Strukturen

Zu den Anweisungen von Abschnitt 1.4.3 werden nun noch die Anweisungen **cobegin/coend** für nebenläufige Ausführung und **await** für bedingtes Warten hinzugenommen.

Die Programmbeschreibung

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

wird oft mit Zeilennummern versehen:

$$P = l : \textbf{cobegin } l_1 : P_1\ l_1' \| \ldots \| l_n : P_n\ l_n'; \textbf{coend};\ l'$$

Daraus 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:

$\mathcal{C}(l, P, l') \equiv$
$(pc = l \wedge pc_1' = l_1 \wedge \ldots \wedge pc_n' = l_n \wedge pc' = \perp) \vee$ (Initialisierung)
$(pc = \perp \wedge pc_1 = l_1' \wedge \ldots \wedge pc_n = l_n' \wedge pc' = l' \wedge \bigwedge_{i=1}^{n} (pc_i' = \perp)) \vee$ (Termination)
$(\bigvee_{i=1}^{n} (\mathcal{C}(l_i, P_i, l_i') \wedge same(V \backslash V_i) \wedge same(PC \backslash \{pc_i\})))$ (Transitionen von P_i)

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

await-Anweisung:

$\mathcal{C}(l, \textbf{await}(b), l') \equiv$
$(pc_i = l \wedge pc_i' = l \wedge \neg b \wedge same(V_i))$ "busy waiting"
$\vee\ (pc_i = l \wedge pc_i' = l' \wedge b \wedge same(V_i))$

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

1.4.4 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:

Algorithmus 1.1 (Wechselseitiger Ausschluss nach Dijkstra - Programmschema)

```
P0:     Initialisierung
        m : cobegin P||Q coend
        wobei
   P:   l_0 : while True do          Q:   l_1 : while True do
            p_i : non-critical section;        q_i : non-critical section;
            Eintrittsprotokoll                 Eintrittsprotokoll
            p_j : critical section;            q_j : critical section;
            Austrittsprotokoll                 : Austrittsprotokoll
          endwhile l'_0                      endwhile l'_1
```

Algorithmus 1.2 (Wechselseitiger Ausschluss nach Dijkstra - 1)

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

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 Forderungen 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 1.1.

Ein erster Lösungsversuch folgt der Vorstellung einer Ampel, die auf grün und rot gesetzt wird (Algorithmus 1.2). Was geht hier schief?

Der zweite Versuch in Algorithmus 1.3 führt das Anmelden eines Zutrittswunsches durch $wantP$ und $wantQ$ ein. Ein Prozess prüft, ob dieser bei dem anderen vorliegt. Was läuft hier falsch?

Der dritte Versuch in Algorithmus 1.4 versucht die Verklemmung durch das temporäre Aufheben des Zutrittswunsches zu vermeiden. Warum funktioniert das nicht?

Algorithmus 1.3 (Wechselseitiger Ausschluss nach Dijkstra - 2)

P2: $wantP = wantQ = False : boolean$,
m : **cobegin** $P \| Q$ **coend**
wobei

P: l_0 : **while** True **do**
p_0 : non-critical section;
$p_1 : wantP := True$;
p_3 : **await**$(wantQ = False)$
p_4 : critical section;
$p_5 : wantP := False$;
endwhile l_0'

Q: l_1 : **while** True **do**
q_0 : non-critical section;
$q_1 : wantQ := True$;
q_3 : **await**$(wantP = False)$
q_4 : critical section;
$q_5 : wantQ := False$;
endwhile l_1'

Algorithmus 1.4 (Wechselseitiger Ausschluss nach Dijkstra - 3)

P3: $wantP = wantQ = False : boolean$,
m : **cobegin** $P \| Q$ **coend**
wobei

P: l_0 : **while** True **do**
p_0 : non-critical section;
$p_1 : wantP := True$;
p_3 : **while** $wantQ = True$ **do**
$wantP := False$;
$wantP := True$
endwhile
p_4 : critical section;
$p_5 : wantP := False$;
endwhile l_0'

Q: l_1 : **while** True **do**
q_0 : non-critical section;
$q_1 : wantQ := True$;
q_3 : **while** $wantP = True$ **do**
$wantQ := False$;
$wantQ := True$
endwhile
q_4 : critical section;
$q_5 : wantQ := False$;
endwhile l_1'

Algorithmus 1.5 (Wechselseitiger Ausschluss nach Dekker/Peterson (Dijkstra 4))

P4: $wantP = wantQ = False : boolean$,
$last = 1 \vee last = 2 : integer$,
m : **cobegin** $P \| Q$ **coend**
wobei

P:	Q:
l_0 : **while** True **do**	l_1 : **while** True **do**
[p_0 : non-critical section;]	[q_0 : non-critical section;]
$p_1 : wantP := True$;	$q_1 : wantQ := True$;
$p_2 : last := 1$;	$q_2 : last := 2$;
p_3 : **await**$(wantQ = False \vee last = 2)$;	q_3 : **await**$(wantP = False \vee last = 1)$;
[p_4 : critical section;]	[q_4 : critical section;]
$p_5 : wantP := False$;	$q_5 : wantq := False$;
endwhile l'_0	**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 1.5). Ob dieser Algorithmus das Problem löst, wird mit seiner Kripke-Struktur untersucht.

Kripkestrukturen (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 1.5 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

$$P = l : \textbf{cobegin } l_0 : P\ l'_0 \| \ldots \| l_1 : Q\ l'_1; \textbf{coend};\ l'$$

in der zuvor eingführten Notation dargestellt.

$PC = \{pc, pc_0, pc_1\}, \quad V = V_0 = V_1 = \{wantP, wantQ, last\}$
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:

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

- $\mathcal{C}(l_1, Q, l_1') \wedge same(V \backslash V_1) \wedge same(PC \backslash \{pc_1\})$ also $\mathcal{C}(l_1, Q, l_1') \wedge same(pc, pc_0)$

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

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

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

Abbildung 1.15 zeigt die zugehörige Kripkestruktur. Dabei soll eine Knotenbeschriftung (i, j, n, b_1, b_2) den Zustand $(p_i, q_j, last = n, wantP = b_1, wantQ = b_2)$ bezeichnen.

Gelten die aufgestellten Forderungen?

A) Die Befehlszähler von P und Q sind nie gleichzeitig in ihren kritischen Abschnitten. Für die Kripkestruktur 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 $\Box\neg(p_5 \wedge q_5)$ 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 $\Box(p_1 \Rightarrow \Diamond(p_5))$ ausgedrückt.

Dies kann in der Kripkestruktur von Abbildung 1.15 verifiziert werden. Der folgende Abschnitt stellt die Grundlagen für die algorithmische Lösung solcher Probleme da.

1.5 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 bereits ausgeführt wurde oder wie die folgende Spezifikation eines Aufzuges zeigt.

Beispiel 1.29 (Spezifikation eines Aufzuges (Fragment)) Die folgenden zwei Teile einer Spezifikation des gewünschten Verhaltens eines Aufzuges seien gegeben:

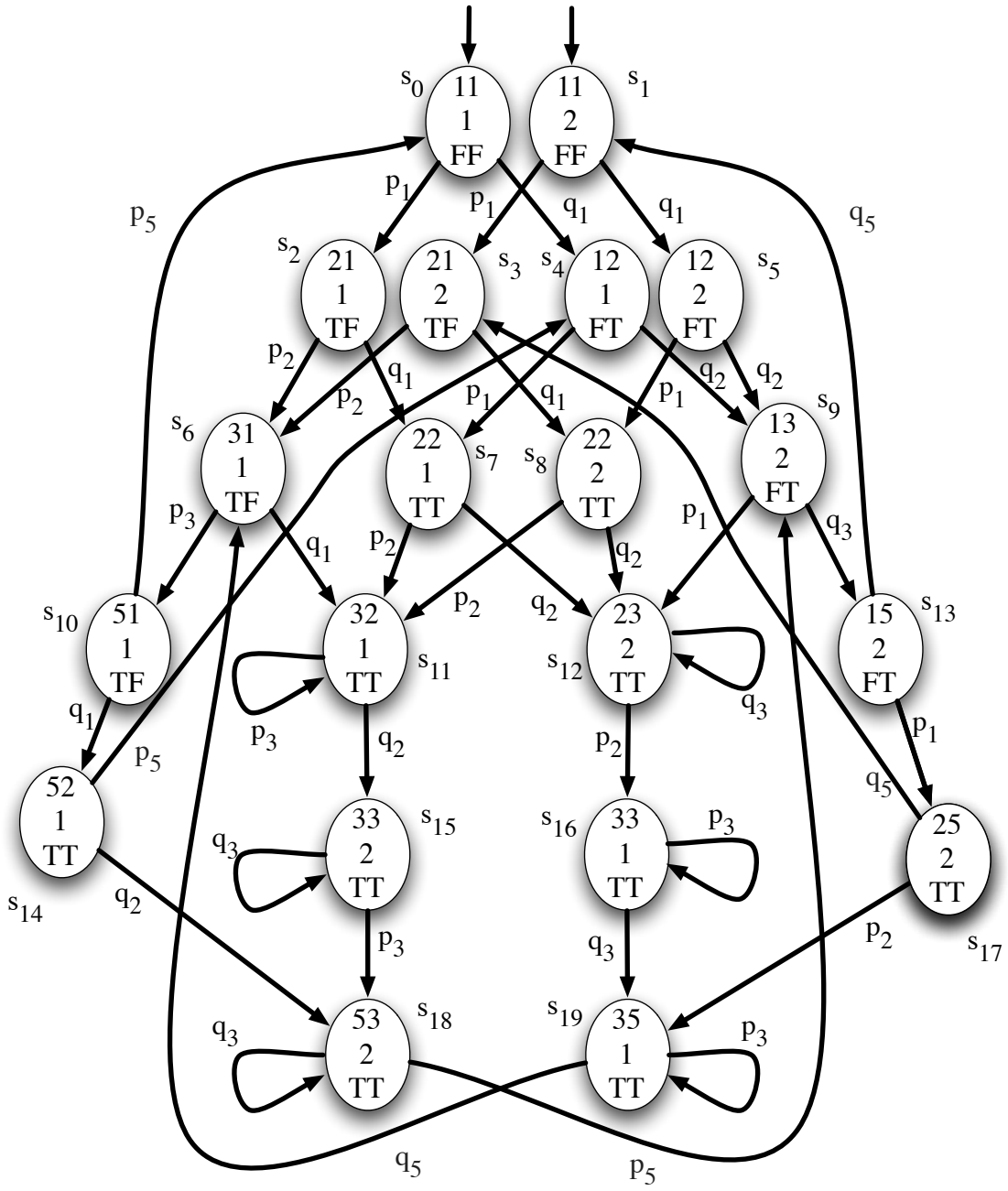


Abbildung 1.15: Kripke-Struktur zum Beispielprogramm „wechselseitiger Ausschluss“

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).

I. $\forall t, \forall n : app(n,t) \Rightarrow \exists t' > t : serv(n,t')$

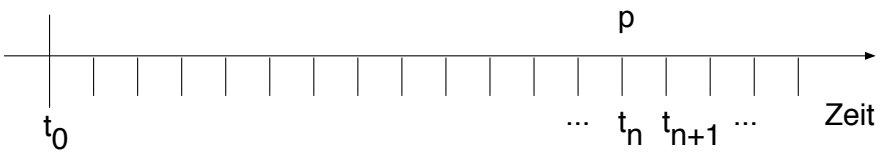
II. $\forall t, \forall t' > t, \forall n Big(\Big(app(n,t) \wedge H(t') \neq n \wedge \exists t_{trav} \,.\, t \leq t_{trav} \leq t' \wedge H(t_{trav}) = n\Big)$
$\Rightarrow \Big(\exists t_{serv} \,.\, t \leq t_{serv} \leq t' \wedge serv(n, t_{serv})\Big)\Big)$

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.

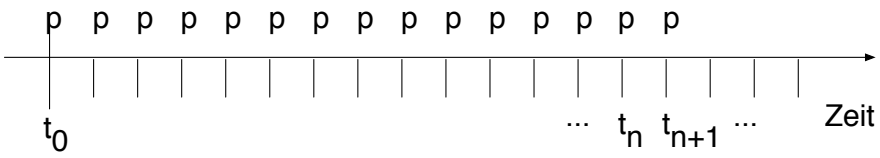
In der temporalen Logik wird der Zeit-Parameter nicht explizit benutzt. Dadurch werden die Formeln einfacher und die Entscheidungsprozeduren zur Gültigkeitsprüfung effektiv durchführbar. Amir Pnueli hat 1977 die temporale Logik erstmals für die Programmverifikation vorgeschlagen, indem er die Zeit als Programmschritte interpretierte. Von ihm stammt die „linear time logic“ LTL. 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.

Typische *temporale Quantoren* sind $\Diamond p$ und $\Box p$:

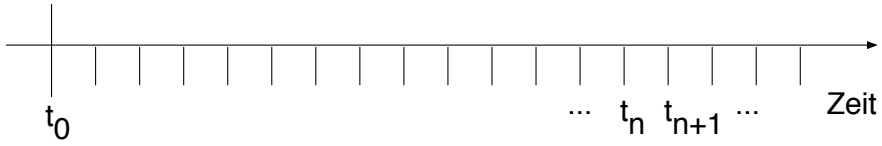
$\Diamond p$ irgendwann einmal gilt p



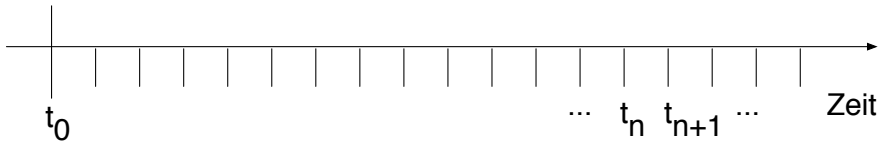
$\Box p$ von jetzt an gilt immer p



$\Diamond\Box p$ bedeutet?



$\Box\Diamond p$ bedeutet?



1.5.1 Syntax und Semantik von LTL-Formeln

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

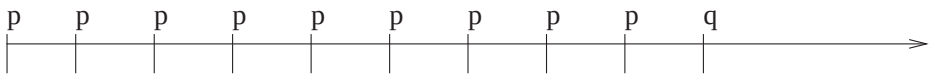
Definition 1.30 *Es sei AP eine Menge von aussagenlogischen Atomen. Dann wird die* Syntax von LTL-Formeln *wie folgt durch Backus-Naur-Form definiert:*

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

wobei $p \in AP$ ist.

LTL-Quantoren X, F, G und U beziehen sich auf unendliche Folgen $\alpha = A_0A_1A_2\cdots$ von Mengen $A_i \in \mathcal{P}(\mathcal{AP})$, 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 $\Box p$),
$p\ Uq$ „until“: es gibt einen Element der gegebenen Folge, in dem q gilt und vor diesem Element gilt immer p.



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

Definition 1.31 *Sei $\alpha = A_0A_1A_2\cdots \in \mathcal{P}(\mathcal{AP})^\omega$ eine unendliche Folge von Aussagenmengen. Dann sei für $i \in \mathbb{N}$ die Folge $\alpha^i = A_iA_{i+1}\cdots$ der i-Suffix von α. In der folgenden induktiven Definition seien $p \in AP$ und f, f_1, f_2 LTL-Formeln. $\alpha \models f$ ist zu lesen als „für die Folge α gilt f“ oder „α erfüllt f“.*

1.	$\alpha \models \mathbf{true}$.		
2.	$\alpha \not\models \mathbf{false}$.		
3.	$\alpha \models p$	$\Leftrightarrow$	$p \in A_0$.
4.	$\alpha \models \neg f$	$\Leftrightarrow$	$\alpha \not\models f$.
5.	$\alpha \models f_1 \vee f_2$	$\Leftrightarrow$	$\alpha \models f_1$ *oder* $\alpha \models f_2$.
6.	$\alpha \models f_1 \wedge f_2$	$\Leftrightarrow$	$\alpha \models f_1$ *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.	$\alpha \models f_1 U f_2$	$\Leftrightarrow$	$\exists k \geq 0\,.\,\alpha^k \models f_2$ *und für alle* $0 \leq j < k$ *gilt* $\alpha^j \models f_1$.

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

Jede LTL-Formel α definiert also eine ω-Sprache im Sinne der Definition 1.15 (Seite 15). Es genügen die Operatoren $\vee, \neg, X, U$ um alle Formeln von LTL auszudrücken:

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

Aufgabe 1.32 Beweisen Sie diese Äquivalenzen.

Eine LTL-Formel f gilt für eine unendliche Folgen $\pi = s_0s_1s_2\cdots$ von Zuständen einer Kripke-Struktur $M := (S, S_0, R, E_S)$, wenn sie für die zugehörige Zustandsetikettenfolge $E_S(\pi) = E_S(s_0)E_S(s_1)E_S(s_2)\cdots \in \mathcal{P}(AP)^\omega$ gilt. Man schreibt $M, \pi \models f$.

Definition 1.33 *Sei $M := (S, S_0, R, E_S)$ eine Kripke-Struktur und $\pi = s_0s_1s_2\cdots \in S^\omega$ eine unendliche Folge von Zuständen von M. Dann sei $M, \pi \models f :\Leftrightarrow E_S(\pi) \models f$.*

Beispiel 1.34 MW-Ofen Die Abbildung 1.16 zeigt eine Kripke-Struktur M als Systembeschreibung eines Mikrowellen-ofens. 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 $\pi = s_0s_1s_2\cdots \in S^\omega$, 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 $\pi = s_0s_1s_2\cdots = 1(25)^\omega \in S^\omega$ oder auch $(1253)^\omega \in S^\omega$.

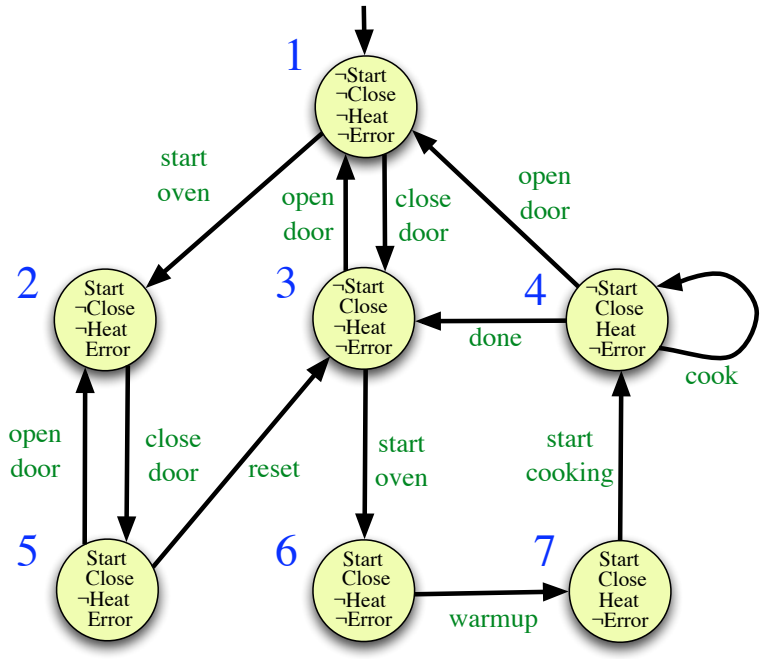


Abbildung 1.16: Kripke-Struktur für das Mikrowellenofenbeispiel

Definition 1.35 *Eine LTL-Formel f* gilt *(ist* gültig*) im Zustand $s \in S$ einer Kripke-Struktur $M := (S, S_0, R, E_S)$ falls $M, \pi \models f$ für alle Pfade $\pi = s_0 s_1 s_2 \cdots \in S^\omega$ gilt, die in s beginnen, d.h. $s_0 = s$ (in Zeichen $M, s \models f$). Sie gilt in M, falls sie in allen Anfangszuständen gilt, also: $M \models f :\Leftrightarrow \forall s \in S_0 : M, s \models f$.*

Beispiel 1.36 In Beispiel 1.34 gilt $M, s \models f$ für $s = 1$, also $M \models f$, nicht jedoch $M \models g$.

Ein für das Model-Checking fundamentaler Satz sagt, dass zu jeder LTL-Formel eine ihre Sprache akzeptierende Kripke-Struktur existiert. Diese wird auch *Büchi-Automat* genannt.

Satz 1.37 *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 $\alpha_1, \alpha_2, \alpha_3, \cdots$ 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 3.38 auf Seite 116. Beim Model-Checking - wie auf Seite 17 beschrieben - 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.

Fairnesseigenschaften wie von Definition 3.20 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: 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 3.15. Hier wird ein Existenz-Quantor für Pfade benötigt, wie er in CTL möglich ist.

1.5.2 Syntax und Semantik von CTL- und CTL*-Formeln

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

Ap „für alle Pfade, die von einem gegeben Zustand s ausgehen, gilt die Formel p",
Ep „es gibt einen Pfad, der von einem gegeben 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 jedoch eine bessere Komplexität der Model-Check-Algorithmen zu erhalten, werden diese mit den bisher betrachteten Operatoren verbunden. Dadurch ist folgende Syntaxdefinition motiviert.

Definition 1.38 *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:*

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

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.

Definition 1.39 *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. $M, s \models p \quad \Leftrightarrow \quad p \in E_S(s)$.
2. $M, s \models \neg g \quad \Leftrightarrow \quad M, s \not\models g$.
3. $M, s \models g_1 \vee g_2 \quad \Leftrightarrow \quad M, s \models g_1$ *oder* $M, s \models g_2$.
4. $M, s \models g_1 \wedge g_2 \quad \Leftrightarrow \quad M, s \models g_1$ *und* $M, s \models g_2$.
5. $M, s \models Ef \quad \Leftrightarrow \quad$ *Es gibt einen in s beginnenden Pfad π mit* $M, \pi \models f$.
6. $M, s \models Af \quad \Leftrightarrow \quad$ *Für alle in s beginnenden Pfade π 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.

In CTL müssen also vor den Pfad-Quantoren X, F, G, U immer Zustands-Quantoren A oder E stehen. Es gibt damit 8 Kombinationen:

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

Diese können alle mittels EXg, EGg, $E[g_1 U g_2]$ ausgedrückt werden:

Satz 1.40 *Es gelten die folgenden Äquivalenzen:*

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

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 1.17. Die Abbildung 1.19 auf Seite 47 zeigt die acht CTL-Operatoren als Beispiele in dieser Darstellung.

Beispiele:

- $EF(Start \wedge \neg Ready)$
  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.

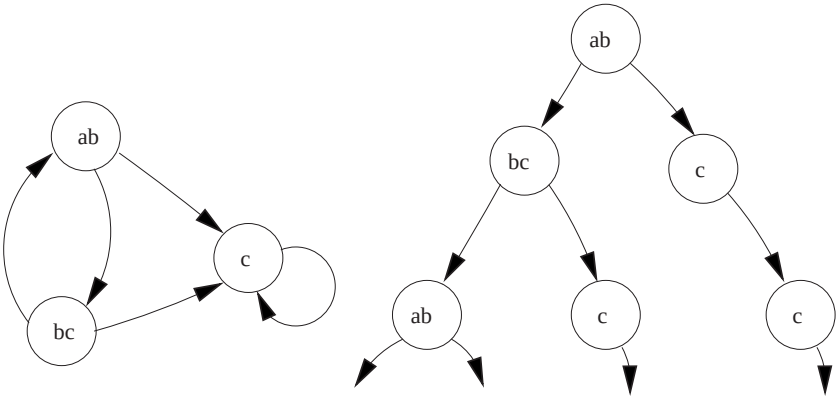


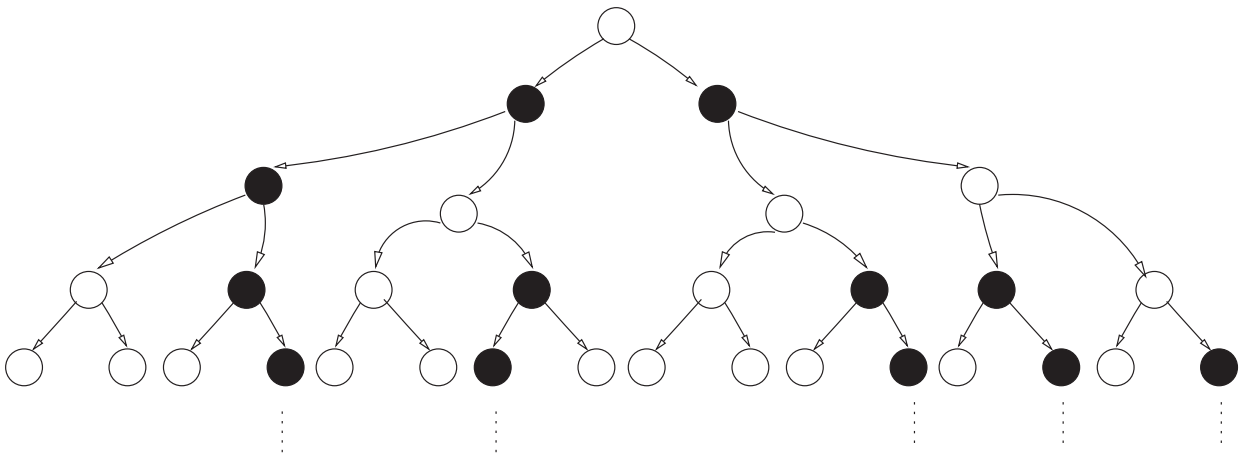
Abbildung 1.17: Abwicklung einer Kripke-Struktur

- $AG(AF\,DeviceEnabled)$
  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.

Auch in CTL kann man nicht alle LTL-Spezifikationen ausdrücken. Es gibt keine CTL-Formel, die äquivalent zur LTL-Formel

$$A(FGp)$$

ist! Sie bedeutet: „auf jedem Pfad gibt es einen Zustand, ab dem p immer gilt".

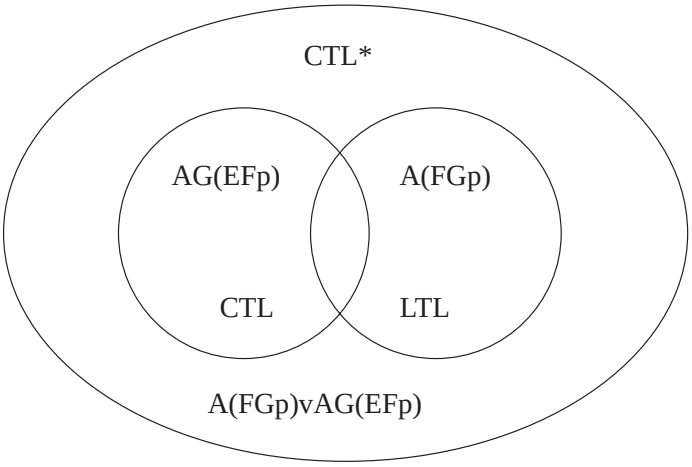


Umgekehrt gibt es keine LTL-Formel, die äquivalent zur CTL-Formel:

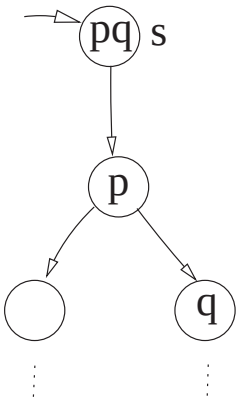
$$AG(EFp)$$

ist! Sie 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“.

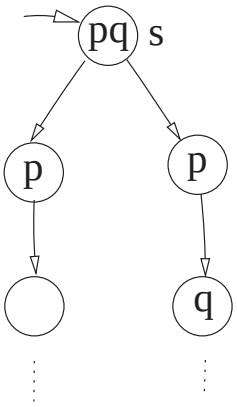
CTL^* ist eine temporale Logik, die syntaktisch beide Logiken LTL und CTL umfasst. Es gibt aber auch Formeln in CTL*, z.B. $A(FGp) \vee AG(EFp)$, die weder in CTL noch in LTL ausdrückbar sind, wodurch CTL* die Logiken LTL und CTL auch semantisch echt enthält:



Aufgabe 1.41 Gegeben sind die folgenden Kripke-Strukturen M_1 und M_2:



(a) M_1



(b) M_2

a) Gibt es Formeln in LTL, die die Strukturen unterscheiden, d.h. nur in einem Modell gelten?

b) Das gleiche für CTL.

1.5.3 Faire Kripke-Struktur

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 Kripkestrukturen 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 1.42 *(faire Kripke-Struktur)*

Eine faire *Kripke-Struktur* $M := (S, S_0, R, E_S, E_F^1, \cdots, E_F^k)$ *wird wie eine Kripke-Struktur* $M := (S, S_0, R, E_S)$ *definiert, nur dass es jetzt auch* $k \geq 1$ *Endzustandsmengen* $E_F^i \subseteq S$ *gibt. Ein Pfad* $\pi = s_0 s_1 s_2 \cdots \in SS(M)$ *heißt* fair *falls* $\text{infinite}(\pi) \cap E_F^i \neq \emptyset$ *für alle* $i \in \{1, \cdots, k\}$ *(vergl. Definition 1.5 auf Seite 6).*

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 1.39 auf Seite 37 wie folgt:

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

Beispiel 1.43

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

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

1.5.4 CTL-Model-Checking

Die *CTL-Model-Checking* Aufgabe lautet:
für eine gegebene Kripke-Struktur $M := (S, S_0, R, E_S)$ und eine gegebene CTL-Formel f ist zu berechnen:

$$\{s \in S \mid M, s \models f\}$$

Algorithmus für CTL-Model-Checking:

Erweitere $E_S(s)$ für alle $s \in S$ schrittweise zu $label(s)$. Das wird dann die Menge der 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 1.40 auf Seite 38 mittels EXg, EGg und $E[g_1 U g_2]$ ausgedrückt werden können, bleiben mit drei elementaren Operatoren im Folgenden 6 zu entwickelnde Prozeduren, die der Algorithmus jeweils aufruft:

1. $f \in AP$ ist atomar:
   Für Zustände s mit $f \in E_S(s)$ setzte $f \in label(s)$.

2. $f = \neg f_1$:
   Für Zustände s mit $f_1 \notin label(s)$ setzte $\neg f_1 \in label(s)$.

3. $f = f_1 \vee f_2$:
   Für Zustände s mit $f_1 \in label(s)$ oder $f_2 \in label(s)$ setzte $f \in label(s)$.
   (Entsprechend für $f = f_1 \wedge f_2$ und $f = f_1 \Rightarrow f_2$.)

4. $f = EXf_1$:
   Für Zustände s mit $R(s,t)$ und $f_1 \in label(t)$ setzte $f \in label(s)$.

5. $f = E[f_1 U f_2]$:
   Für Zustände s mit $f_2 \in label(s)$ setzte $f \in label(s)$.
   Für Zustände t mit $R(t,s)$ und $f_1 \in label(t)$ setze $f \in label(t)$.
   Fahre schrittweise in Gegenrichtung der Transitionen fort und setze $f \in label(s)$, falls es einen Pfad von s zu einem s' mit $f_2 \in label(s')$ gibt, auf dem für alle Zustände t davor $f_1 \in label(t)$ gilt. (Siehe Algorithmus 1.6.)

6. $f = EGf_1$:
   Zu diesem Punkt wiederholen wir in Def. 1.44 die Definition 3.16 einer strengen Zusammenhangskomponente für den Spezialfall eines gerichteten Graphen *ohne* Kantenbewertung.

Definition 1.44 *Sei $G = (K, R)$ ein gerichteter Graph, d.h.: $R \subseteq K \times K$:*

a) $A \subseteq K$ heißt Zusammenhangskomponente*, falls: $\forall a, a' \in A : aR^*a'$.*

Algorithmus 1.6 Auszeichnen mit $E(f_1 U f_2)$

```
PROCEDURE CheckEU(f1, f2)
    T := {s | f2 ∈ label(s)};
    FOR ALL s ∈ T DO label(s) := label(s) ∪ {E[f1 U f2]};
    WHILE T ≠ ∅ DO
        CHOOSE s ∈ T;
        T := T\{s};
        FOR ALL t SUCH THAT R(t, s) DO
            IF E[f1 U f2] ∉ label(t) AND f1 ∈ label(t) THEN
                label(t) := label(t) ∪ {E[f1 U f2]};
                T := T ∪ {t};
            END IF ;
        END FOR ALL ;
    END WHILE ;
END PROCEDURE ;
```

b) Sie heißt strenge Zusammenhangskomponente (SZK) *(strongly connected component: SZK), falls sie maximal ist, d.h.:* $\neg\exists k \in K \backslash A \,.\, \forall a \in A : kR^*a \wedge aR^*k$.

c) Sie heißt nichttriviale Zusammenhangskomponente*, falls:* $|A| > 1$ *oder* $\exists a \in A \,.\, aR^+a$.

Nun betrachten wir wieder die Formel: $f = EGf_1$:
Sei $M = (S, S_0, R, label)$ die im hier entwickelten CTL-Algorithmus jeweils durch die Abbildung *label* erweiterte Kripke-Struktur. Daraus konstruieren wir $M' = (S', S'_0, R', label')$ mit:

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

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 1.45 *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.*

Daraus resultiert folgender Algorithmus zur Entscheidung von EGf_1:

1. Konstruiere $M' = (S', S'_0, R', label')$.
2. Konstruiere alle SZK von M'. (Algorithmus von Tarjan mit $O(|S'| + |R'|)$ Zeitkomplexität).

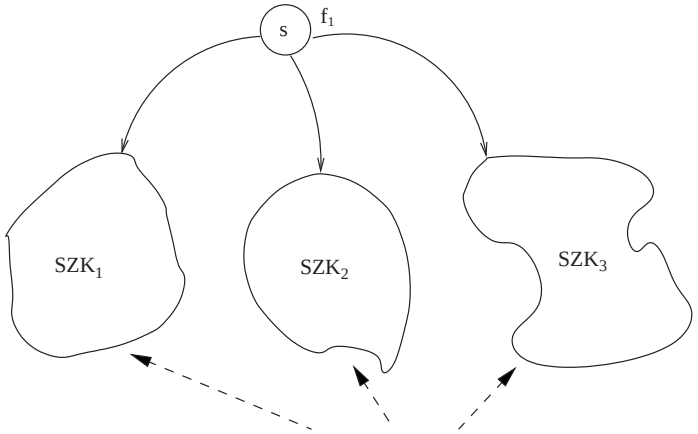


Abbildung 1.18: Strenge Zusammenhangskomponenten mit f_1

Algorithmus 1.7 Auszeichnen mit EGf_1

```
PROCEDURE CheckEGf1
    S' := {s | f1 ∈ label(s)};
    SCC := {C | C a nontrivial SCC of S'};
    T := ⋃_{C∈SCC} {s | s ∈ C};
    FOR ALL s ∈ T DO label(s) := label(s) ∪ {EGf1};
    WHILE T ≠ ∅ DO
        CHOOSE s ∈ T;
        T := T\{s};
        FOR ALL t SUCH THAT t ∈ S' AND R(t,s) DO
            IF EGf1 ∉ label(t) THEN
                label(t) := label(t) ∪ {EGf1};
                T := T ∪ {t};
            END IF;
        END FOR ALL;
    END WHILE;
END PROCEDURE;
```

3. Finde Zustände in nichttrivialen SZK.

4. Suche von diesen rückwärts alle Zustände, die dorthin führen.

   (Das sind maximal $\mathcal{O}(|S| + |R|)$ viele, vergl. Algorithmus 1.7).

Satz 1.46 *Es gibt einen Algorithmus, der für eine Kripke-Struktur $M := (S, S_0, R, E_S)$ und eine CTL-Formel f in $\mathcal{O}(|f| \cdot (|S| + |R|))$ Zeitkomplexität entscheidet, ob f für M gilt, d.h. ob $M \models f$.*

Beweis:
Wende obiges Verfahren auf die Atome von f an und fahre induktiv fort mit den Teilformeln

von f, aufsteigend mit deren Schachtelung. Die Schachtelungstiefe ist durch $\mathcal{O}(|f|)$ begrenzt. Auf jeder Ebene gibt es maximal $\mathcal{O}(|f| \cdot (|S| + |R|))$ Operationen. □

Beispiel 1.47 (MW-Ofen) CTL-Spezifikation zu der Kripke-Struktur in Abb. 1.16:

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

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

Zunächst muss f mit Hilfe der Äquivalenzen von Satz 1.40 so umgeschrieben werden, dass nur noch die Operationen enthalten sind, die der Algorithmus verarbeiten kann. Dabei steht wegen der besseren Lesbarkeit Fg für $\mathbf{true}\, U\, g$.

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

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

Zustände, die mit $EG\neg Heat$ zu markieren sind also $T = \{1, 2, 3, 5\} = S(EG\neg Heat)$. Es folgt:

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

und durch Rückwärtspfade:

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

Damit ergibt sich schließlich:

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

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

CTL-Model-Cecking mit Fairness

Definition 1.48 *Sei $M := (S, S_0, R, E_S, E_F^1, \cdots, E_F^k)$ eine faire Kripke-Struktur. Eine starke Zusammenangskomponente $C \subseteq S$ heißt fair, falls $\forall i \in \{1, \cdots, k\} : E_F^i \cap C \neq \emptyset$.*

Ferner sei $M' := (S', S_0', R', E_S', F_1, \cdots, F_k)$ mit:

$$\begin{aligned} S' &= \{s \in S \mid M, s \models_F f_1\} && (\models_F \textit{ siehe Seite 41}) \\ R' &= R_{|S' \times S'} \\ E_S' &= E_{S|S'} \\ F_i &= E_F^i \cap C \end{aligned}$$

Lemma 1.49 *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 Zusamenhangskomponente in (S', R') führt.*

Daraus folgt eine Prozedur $CheckFairEG(f_1)$ um Spezifikation in fairer Semantik zu prüfen:

$$fair := EG\,True \qquad \text{„Es gibt eine unendliche Folge"}$$

und

$$\begin{array}{lll} M, s \models_F p & \text{gdw.} & M, s \models p \wedge fair \\ M, s \models_F EXf_1 & \text{gdw.} & M, s \models EX(f_1 \wedge fair) \\ M, s \models_F E[f_1 U f_2] & \text{gdw.} & M, s \models E[f_1 U (f_2 \wedge fair)] \end{array}$$

Satz 1.50 *Es gibt einen Algorithmus, der für eine faire Kripke-Struktur $M := (S, S_0, R, E_S, E_F^1, \cdots, E_F^k)$ 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 1.51
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 setzten wir $M := (S, S_0, R, E_S, E_F^1)$ (also $k = 1$) mit

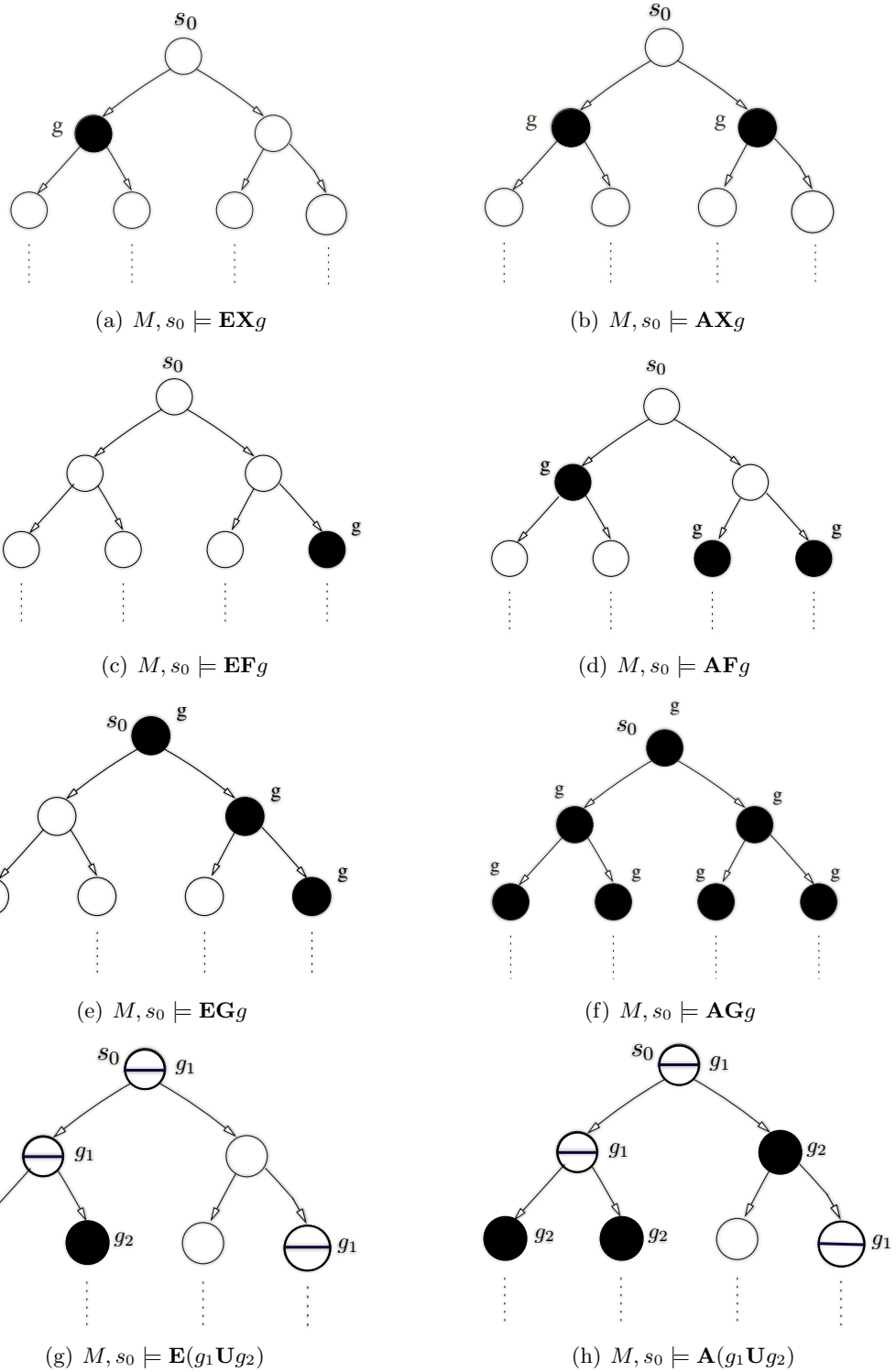
$$E_F^1 = \{s \mid s \models Start \wedge (Close \wedge \neg Error)\} = \{6, 7\}.$$

Mit $S(Start)$, $S(\neg Heat)$ wie vorher ist die ZSK $\{1, 2, 3, 5\}$ *nicht* fair, da sie disjunkt zu $E_F^1 = \{6, 7\}$ ist. Also:

$$\begin{array}{rcl} S(EG\neg Heat) & = & \emptyset \\ S(EF(Start \wedge EG\neg Heat)) & = & \emptyset \\ S(\neg EF(Start \wedge EG\neg Heat)) & = & \{1, \cdots, 7\} \end{array}$$

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



(a) $M, s_0 \models \mathbf{EX}g$ (b) $M, s_0 \models \mathbf{AX}g$

(c) $M, s_0 \models \mathbf{EF}g$ (d) $M, s_0 \models \mathbf{AF}g$

(e) $M, s_0 \models \mathbf{EG}g$ (f) $M, s_0 \models \mathbf{AG}g$

(g) $M, s_0 \models \mathbf{E}(g_1 \mathbf{U} g_2)$ (h) $M, s_0 \models \mathbf{A}(g_1 \mathbf{U} g_2)$

Abbildung 1.19: Acht CTL-Operatoren

Kapitel 2

Partielle Ordnungen

Während Prozesse von sequentiellen Prozessen durch eine totale (auch *linear* genannte) Ordnung gekennzeichnet sind, tritt für nebenläufige Prozesse an deren Stelle die partielle oder strikte Ordnung. Dieses Kapitel gibt im ersten Teil eine formale Definition solcher Ordnungen, während im zweiten Teil ihr Auftreten in verteilten Systemen betrachtet wird. 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.

2.1 Partielle und strikte Halbordnung

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 der Handlungen:

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

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

Nebenläufige Prozesse werden auf (mindestens) zwei Weisen dargestellt:

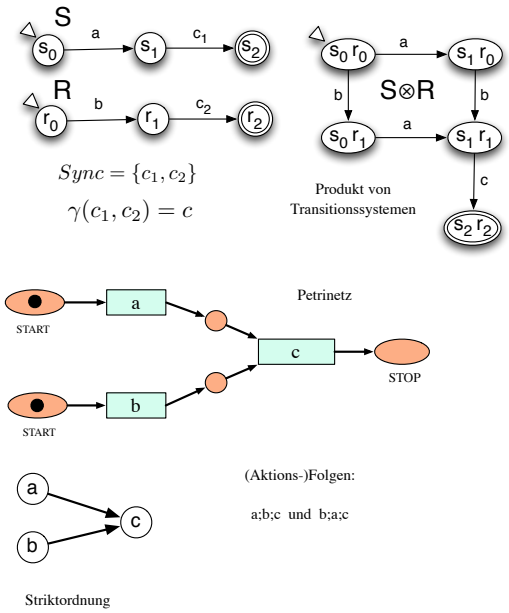


Abbildung 2.1: Nebenläufige Handlungen a und b

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

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

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

Beispiel 2.1 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 2.1 gezeichnet - ein Produkt von Transitionssystemen oder ein Petrinetz[1] verwendet werden. Seine Semantik wird durch die gezeigte Striktordnung oder die Sprache $\{abc, bac\}$ beschrieben.

Definition 2.2 *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 $(a, b) \in R$

[1] nach Lesen von Kapitel 3 betrachten.

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. 2.2 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. 2.3).

Definition 2.3 *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* $(A, <)$.

Anmerkung:

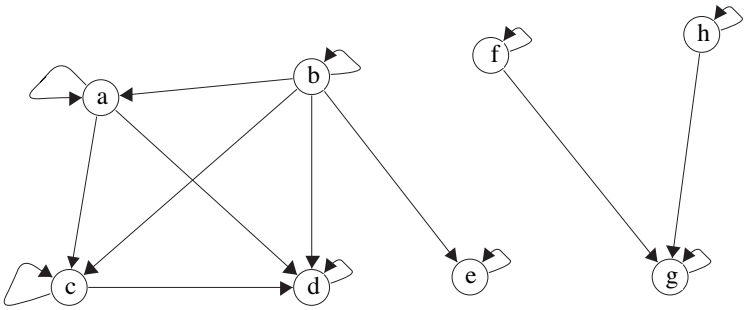
a) $\lessdot = < - <^2$
("$-$" Mengendifferenz, $<^2 := < \circ <= \{(a, b) \mid \exists c.\ a < c \wedge c < b\}$ Relationenprodukt)

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

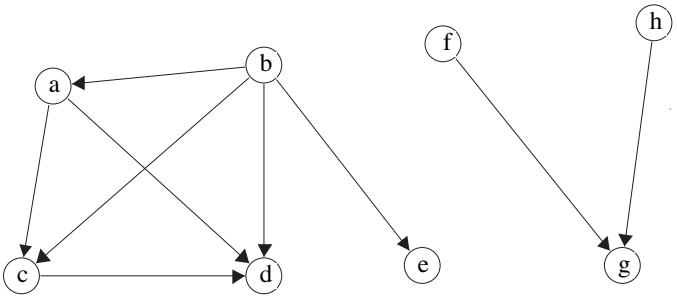
c) Falls A unendlich ist, muss dies nicht gelten:
für die rationale Zahlen $(\mathbb{Q}, <)$ gilt zum Beispiel $\lessdot = \emptyset$.

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:

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

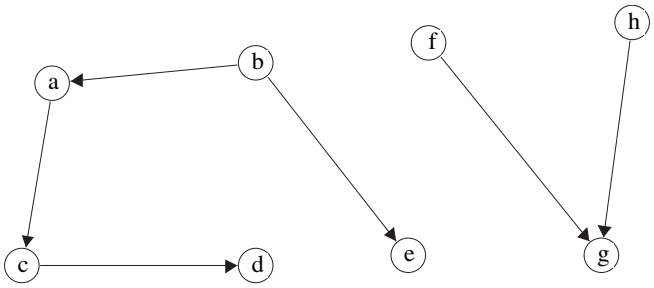


$R = \{(a,a),(a,c),(b,a),...\}$



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

Abbildung 2.2: Partielle Ordnung R und Striktordnung S

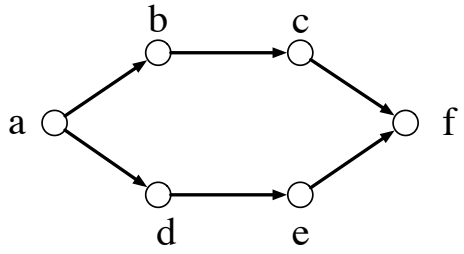


$Q = S - S^2 = \{(b,a),(a,c),(c,d),...\}$

Abbildung 2.3: Präzedenzgraph Q

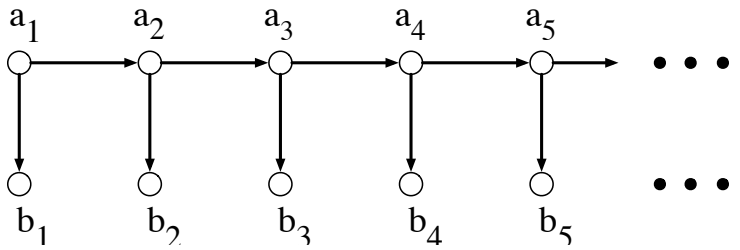
e) Für eine Striktordnung $(A, <)$ ist
$Lin(A, <) := \{(A, <_1) | <_1$ ist lineare Striktordnung mit $<\subseteq <_1\}$
die Menge der linearen Vervollständigungen von $(A, <)$. Ist $(A, <)$ kombinatorisch mit $<= \lessdot^+$ dann wird: $Lin(A, \lessdot) := Lin(A, \lessdot^+)$ definiert.

Beispiel: $(A, \lessdot)$



$Lin(A, \lessdot) = \{$ $a\ b\ c\ d\ e\ f$, Konstruktionsprinzip?
$a\ b\ d\ c\ e\ f$,
$a\ d\ b\ c\ e\ f$,
$a\ d\ b\ e\ c\ f$,
$a\ d\ e\ b\ c\ f$,
$a\ b\ d\ e\ c\ f\ \}$

Beispiel: $(A, \lessdot)$



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

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

Aufgabe 2.4 (Linearisierung)

a) Ist a, b, c, d, e, f, g, h eine Linearisierung der Striktordnung von Abb. 2.2 ?

b) Berechnen Sie für die Striktordnung S von Abb. 2.2 die Relation S^2.

2.2 Logische und vektorielle Zeitstempel

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 2.5 *Ein* Nachrichten-Modell *ist ein System von n Funktionseinheiten bzw. Prozessoren $p_0, \ldots, p_{n-1}$, 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 $\Phi = \{\phi_1, \phi_2, \ldots\}$ enthalten.

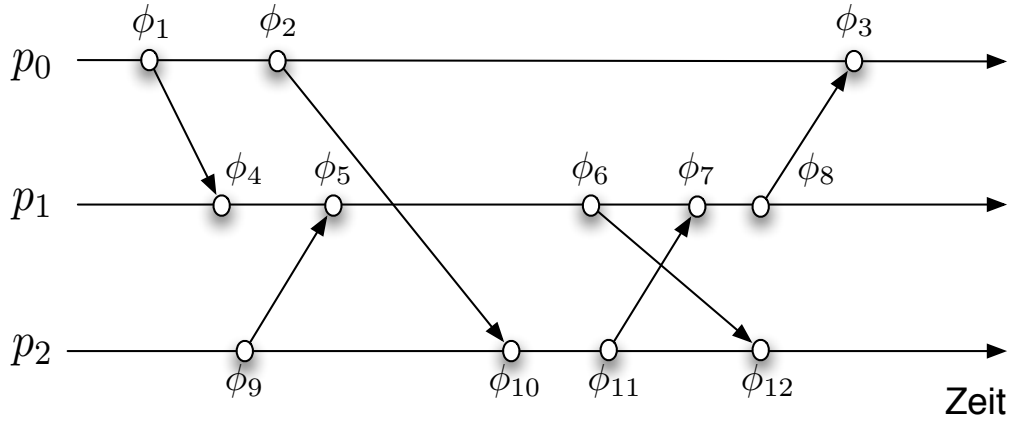


Abbildung 2.4: Nachrichten-Modell mit n=3 Prozessoren durch lokale Zeitskalen

Definition 2.6 *Es wird eine Relation* **vor** $\subseteq \Phi \times \Phi$ *definiert. Für $\phi_1, \phi_2 \in \Phi$ gelte (ϕ_1* **vor** *ϕ_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 (ϕ_1 **vor** *ϕ_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 (ϕ_1 **vor** ϕ_1) gelten. Dies kann nur daher kommen, dass (ϕ_1 **vor** ϕ_2) und (ϕ_2 **vor** ϕ_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 (ϕ_1 **vor** ϕ_2 **vor** ϕ_3 **vor** ϕ_1). Dann gilt aber gleichzeitig (ϕ_3 **vor** ϕ_1) und (ϕ_1 **vor** ϕ_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.

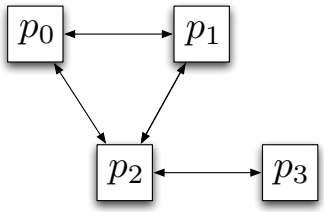


Abbildung 2.5: Banksystem mit 4 Filialen $p_0, \ldots p_3$

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. 2.6

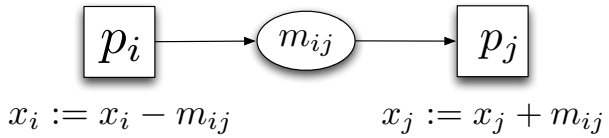


Abbildung 2.6: Übertragung von Nachrichten über Kanal

Es liege eine fortwährende Aktivität vor, d. h. jede Funktionseinheit versendet unendlich oft eine Nachricht an jede andere Funktioneinheit. Falls eine Funktionseinheit terminiert, so kann sie

ständig Nachrichten mit $m = 0$ absenden. Der Anfangszustand des Modells ist $(x_0, \ldots, x_{n-1}) = (a_0, \ldots, a_{n-1})$ und keine Nachricht ist unterwegs. $c := \sum_{i=0}^{n-1} a_i$ ist die Gesamtgeldmenge.

Beispiel 2.7

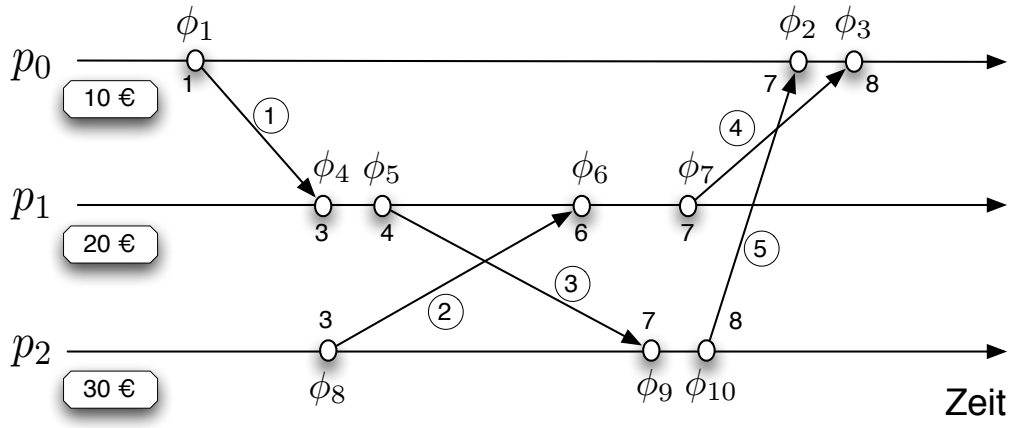


Abbildung 2.7: Zeitskala zu Beispiel 2.7

Gesamtgeldmenge : $10 + 20 + 30 = 60$
$m(\phi_8, \phi_6) = 2$
$1, 3, 4, \ldots, 8$ 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 2.8 Zeitpunkt $t = 5$

p_0	: $x_0 =$	$10 - 1$	$=$	9
p_1	: $x_1 =$	$20 + 1 - 3$	$=$	18
p_2	: $x_2 =$	$30 - 2$	$=$	28
			$\sum$	**55 !**

Verfahren 2:
Die Bankleitung bittet die Summe der abgesandten minus der Summe der eingegangenen Beträge zu x_i jeweils hinzuzuzählen.
Erwartung : $\sum = 60$.

Beispiel 2.9 Zeitpunkt $t = 5$

p_0	:	$x_0 =$	$9 + 1$	$= 10$
p_1	:	$x_1 =$	$18 - 1 + 3$	$= 20$
p_2	:	$x_2 =$	$28 + 2$	$= 30$
			$\sum$	**60**

Aber es kann auch der Fall von Abb. 2.8 eintreten, wo zum Zeitpunkt t ein unerwarterter Überschuss von 63 Geldeinheiten beobachtet wird. Solche Effekte sind durch einfache Zählverfahren nicht zu beheben, jedoch durch die im Folgenden eingeführten *Zeitstempelverfahren.*

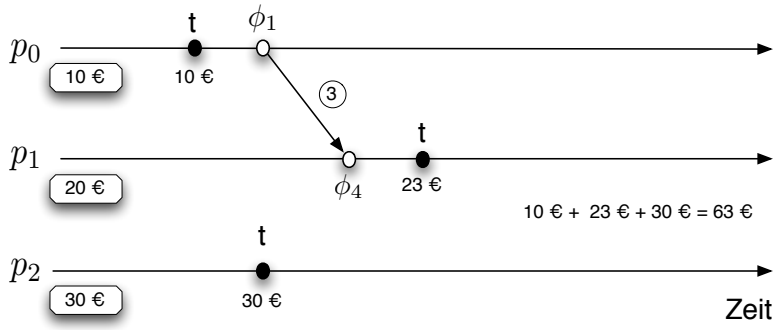


Abbildung 2.8: Im Zeitpunkt t wird virtueller Überschuss durch Nachrichten „aus der Zukunft“ beobachtet.

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

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

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

Satz 2.11 *Für die Ereignisse $\phi_1, \phi_2 \in \Phi$ gilt :*
ϕ_1 **vor** $\phi_2 \Rightarrow LT(\phi_1) < LT(\phi_2)$

Man beachte, dass das Gegenbeispiel von Abb. 2.8 hier nicht mehr auftreten kann!

Beispiel 2.12 Zeitpunkt $t = 1,5$ in Abb. 2.10:

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
			$\sum$	**57**

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

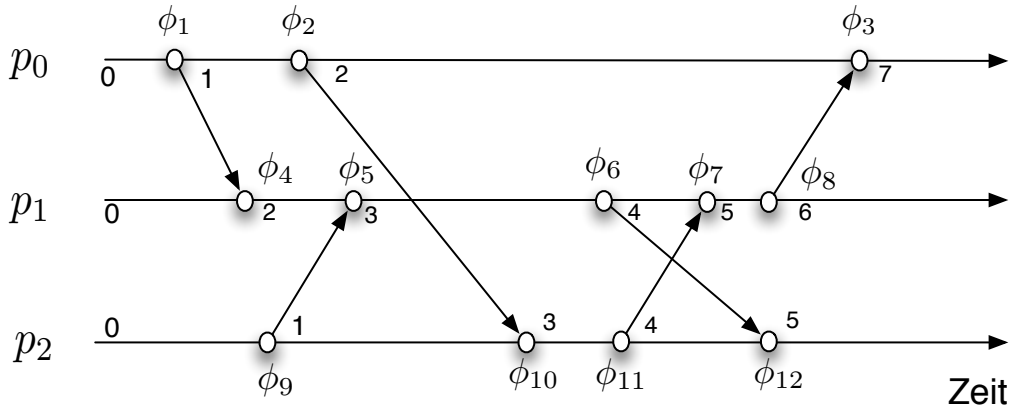


Abbildung 2.9: Senden mit logischen Zeitstempeln

Algorithmus 2.1 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.

$3 = 60$.

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.

Im Beispiel 2.12 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 2.13 *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 ist definiert*

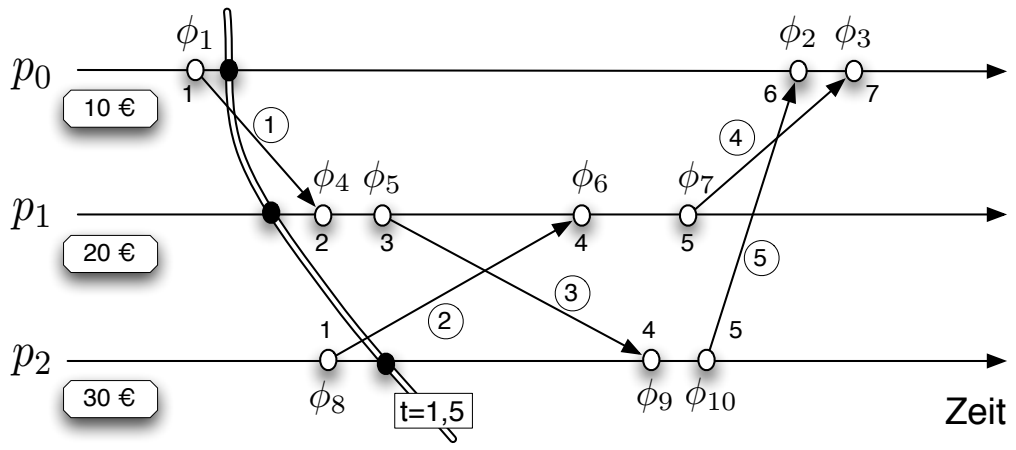


Abbildung 2.10: Zeitskala zu Beispiel 2.12

als:

$(LT(\phi), p_i) <_L (LT(\phi'), p_j)$ *genau dann, wenn*

a) $LT(\phi) < (LT(\phi')$ *oder*

b) $LT(\phi) = LT(\phi')$ *und* $i < j$.

Beispiel 2.14 $(2, p_5) <_L (3, p_5)$, $(2, p_5) <_L (3, p_4)$, $(2, p_5) >_L (2, p_4)$

Wir haben gesehen, dass gilt:

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

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

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

Diese Umkehrung gilt nicht, da $<$ - wie gesagt - keine totale Ordnung ist. Dazu noch ein Gegenbeispiel:

In der Zeitskala von Abb. 2.9 ist Folgendes zu beobachten :
Es gilt $LT(\phi_9) = 1 < LT(\phi_2) = 2$ aber nicht ϕ_9. **vor** ϕ_2 Der Grund ist, dass $LT(\phi) \in \mathbb{N}$ und $\mathbb{N}$ linear geordnet ist!
Also : Statt $\mathbb{N}$ wählen wir eine nicht linear geordnete Menge : $\mathbb{N}^k$ mit $k > 1$.

Definition 2.15 ϕ_1 *heißt* unabhängig *von* ϕ_2 *bzw.* ϕ_1 *heißt* nebenläufig *zu* ϕ_2, *geschrieben als* $\phi_1 \parallel \phi_2$, *falls gilt :*
$\phi_1 \parallel \phi_2 \Leftrightarrow \neg(\phi_1 \textbf{ vor } \phi_2) \wedge \neg(\phi_2 \textbf{ vor } \phi_1)$

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

Definition 2.16 *(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:

$\vec{v}_i[j] \mapsto \max(\vec{v}_i[j], \vec{VT}_m[j])$ *falls eine Nachricht m mit dem Vektorzeitstempel $\vec{VT}_m$ empfangen wird. Wird keine Nachricht empfangen, bleibt $\vec{v}_i[j]$ unverändert ($j \neq i$).*
(Aktualisieren der anderen Komponenten)

Definition 2.17 *(Vektor-Uhr)*
Die Abbildung $VC : \Phi \to \mathbb{N}^n$ 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. 2.11 ist $VC(\phi)$.

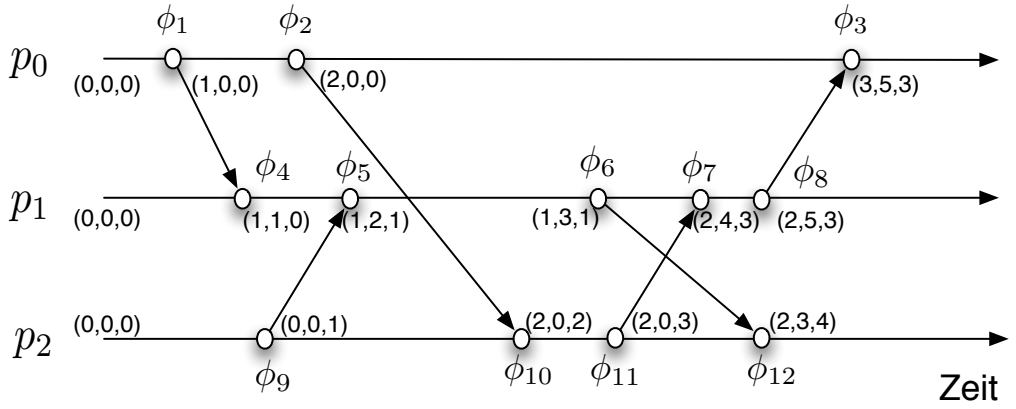


Abbildung 2.11: Zeitskala zur Vektorzeit

Definition 2.18

a) Partielle Ordnung *auf* $\mathbb{N}^n$ *:* $\quad \vec{v}_1 \leq \vec{v}_2 \Leftrightarrow \forall i \in \{1, \ldots, n\} : \vec{v}_1[i] \leq \vec{v}_2[i]$

b) Strikte Ordnung *auf* $\mathbb{N}^n$ *:* $\quad \vec{v}_1 < \vec{v}_2 :\Leftrightarrow \vec{v}_1 \leq \vec{v}_2 \wedge \vec{v}_1 \neq \vec{v}_2$

c) $\vec{v}, \vec{v}'$ *heißen* unvergleichbar*, falls* $\neg(\vec{v} \leq \vec{v}') \wedge \neg(\vec{v}' \leq \vec{v})$ *gilt.*

Satz 2.19

$$VC(\phi_1) < VC(\phi_2) \Leftrightarrow \phi_1 \textbf{ vor } \phi_2$$
$$VC(\phi_1), VC(\phi_2) \textit{ unvergleichbar } \Leftrightarrow \phi_1 \parallel \phi_2$$

Aufgabe 2.20 Der Bankleitung werden ständig alle Vektor-Zeiten $VC(\phi)$ gesandt. Kann Sie darauf ein Verfahren aufbauen, um das Bilanz-Problem zu lösen?

Kapitel 3

Platz/Transitions-Netze und Verifikation

3.1 Einleitung

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 letzen 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 in Kapitel 7 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], [Rei10] und [Rei82].

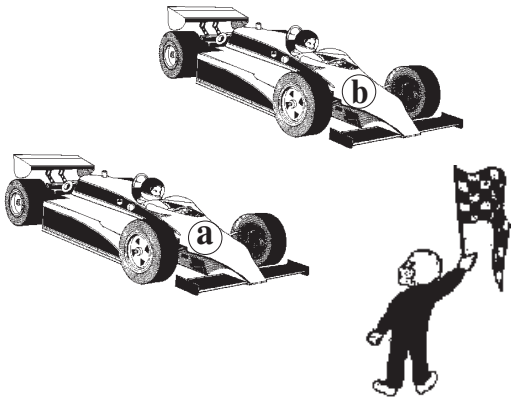


Abbildung 3.1: Start zweier Rennwagen

3.2 Netze

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

Das Beispiel stellt den Startvorgang eines Autorennens dar [GV03]. Zur Vereinfachung werden nur drei Objekte modelliert: zwei Autos und ein Starter (Abb. 3.1). Wenn die Fahrer der Wagen ihre Vorbereitungen abgeschlossen haben, geben Sie ein Fertigzeichen ("ready sign"). 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:

a) Liste der Bedingungen:

p_1:	car a; preparing for start
p_2:	car a; waiting for start
p_3:	car a; running
p_4:	ready sign of car a
p_5:	start sign for car a
p_6:	starter; waiting for ready signs
p_7:	starter; start sign given
p_8:	ready sign of car b
p_9:	start sign for car b
p_{10}:	car b; preparing for start
p_{11}:	car b; waiting for start
p_{12}:	car b; running

b) Liste der Aktionen:
t_1: car a; send ready sign
t_2: car a; start race
t_3: starter; give start sign
t_4: car b; send ready sign
t_5: car b; start race

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

I	**Das Prinzip der Dualität in Petrinetzen**
Es gibt zwei disjunkte Mengen von Grundelementen: *P-Elemente* (state elements, Plätze, Stellen) und *T-Elemente* (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.
- Beachte: In der Prozessalgebra wird diese Dualität nicht berücksichtigt, zumindest nicht explizit.

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

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

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

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

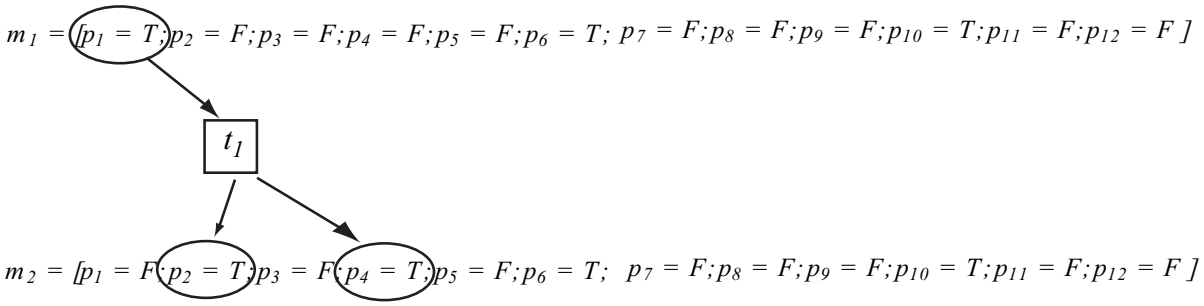


Abbildung 3.2: Lokalität der Aktion t_1

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

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

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

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

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

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

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

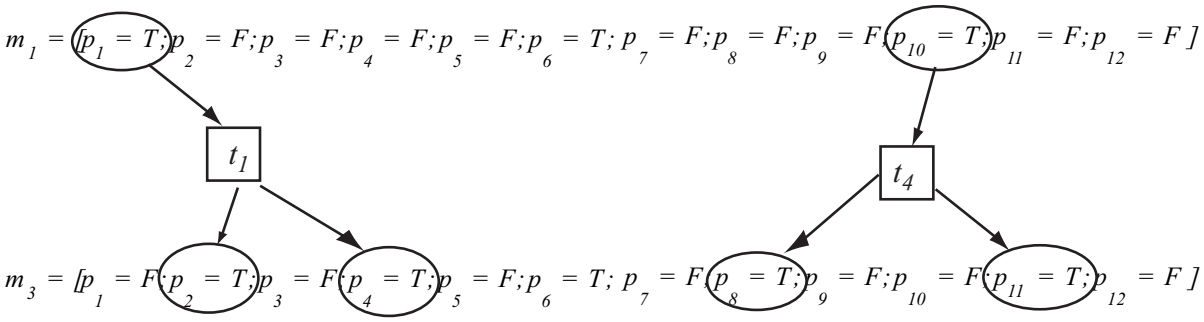


Abbildung 3.3: Concurrent actions t_1 and t_4

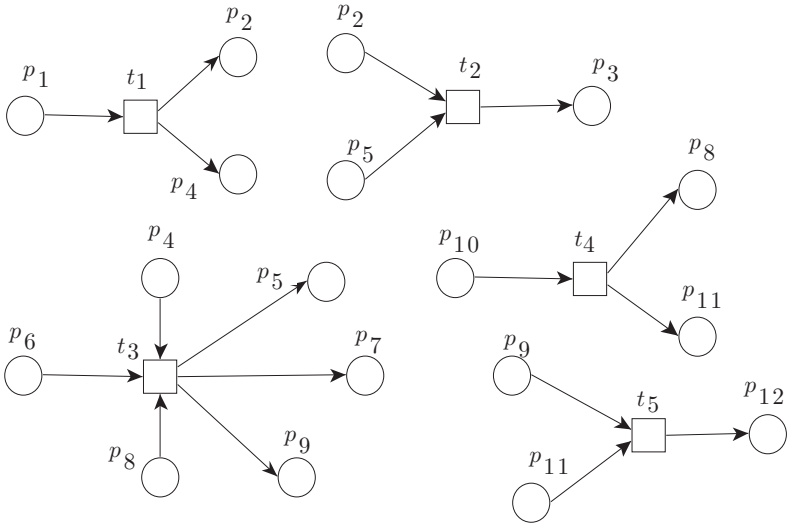


Abbildung 3.4: Einzelaktionen als Transitionen

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

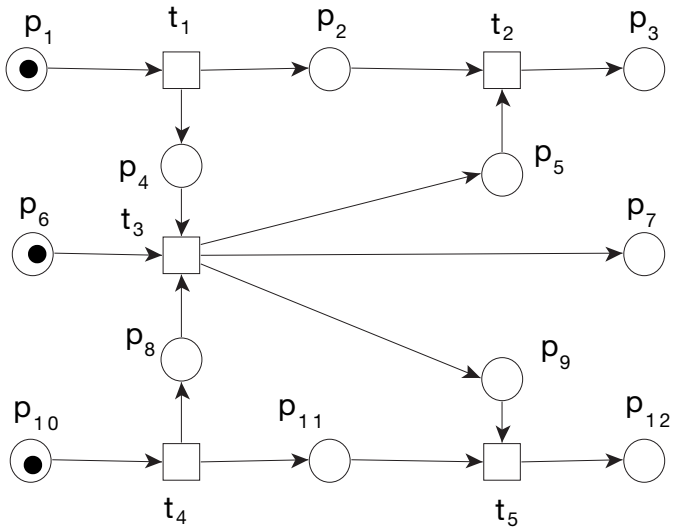


Abbildung 3.5: Netz $\mathcal{N}$ zum Beispiel

IV	Das Prinzip der grafischen Darstellung von Petrinetzen
P-Elemente (auch S-Elemente) werden durch runde grafische Elemente (Kreise, 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. Außerdem gibt es Beschriftungen, wie Bezeichner, Gewichtungen oder Schutzbedingungen (guards).	

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

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

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

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

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

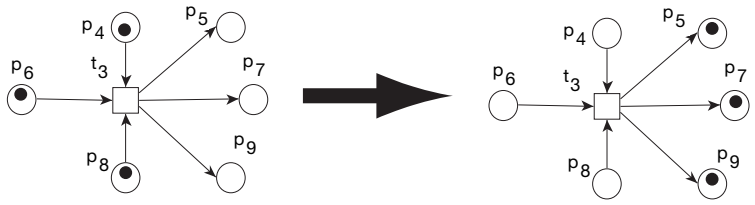


Abbildung 3.6: Schaltregel

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

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

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

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

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

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

Definition 3.2 *Für ein Netz $\mathcal{N} = (P, T, F)$ und ein Element $x \in P \cup T$ bezeichnet $^\bullet x := \{y \in P \cup T \mid (y, x) \in F\}$ die Menge der* Eingangselemente *und $x^\bullet := \{y \in P \cup T | (x, y) \in F\}$ die Menge der* Ausgangsselemente *von x. Ist x ein Platz, dann heißen $^\bullet x$ bzw. $x^\bullet$* Eingangs- *bzw.* Ausgangs-Transitionen*. Für eine Menge von Elementen $X \subseteq P \cup T$ 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.*

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

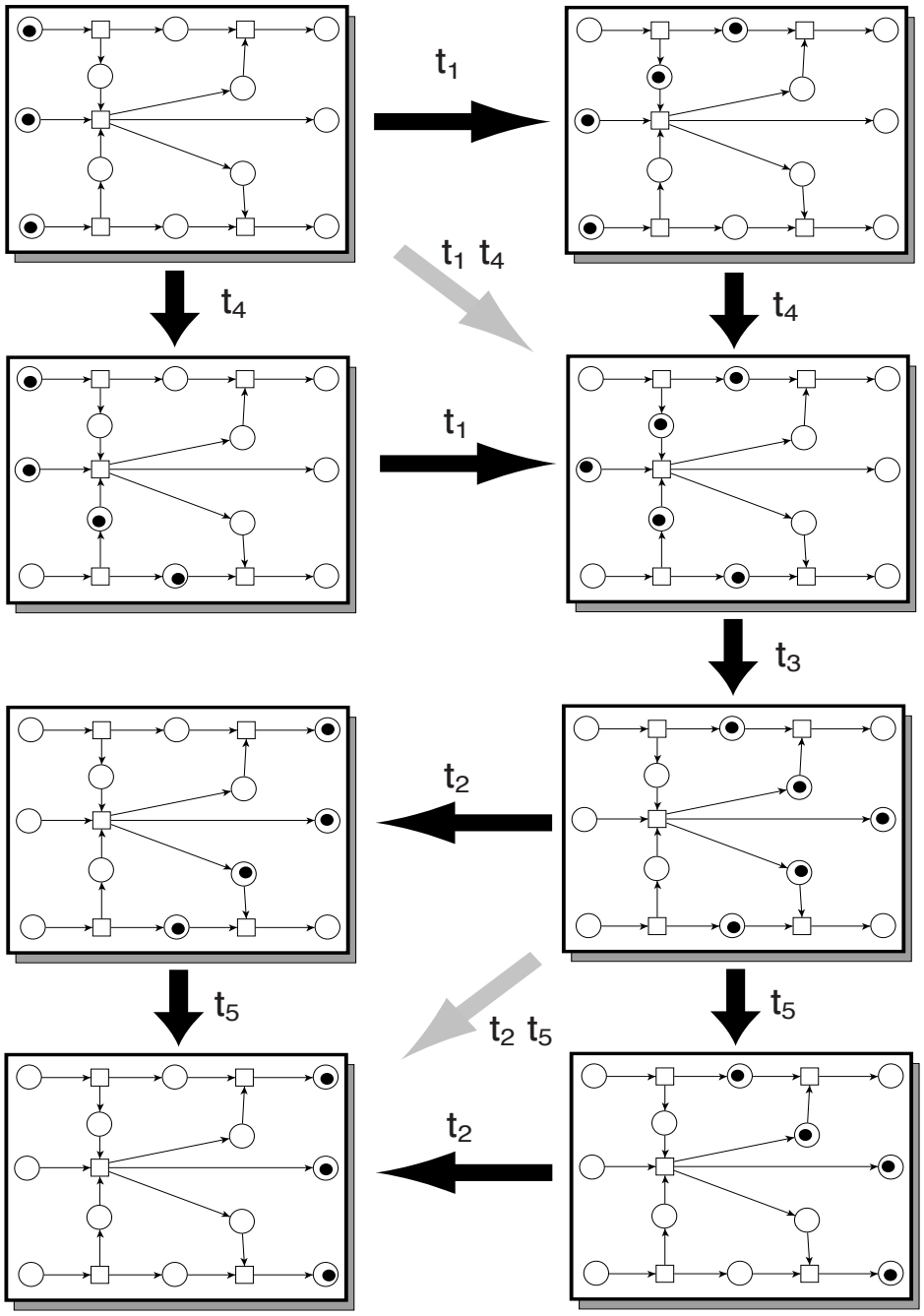


Abbildung 3.7: Schaftfolgen des Netzes 3.5

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

Die Konstruktion von Systemhierarchien durch Vergröberung (Abstraktion) und Verfeinerung ist ein wichtige Methode des Systementwurfs. Petrinetze unterstützen dies durch besondere mit ihrer Struktur kompatible Konzepte. Diese werden unhabhängig 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 vergröbernden Teiles bilden wird.

Definition 3.3 *Sei $\mathcal{N} = (P, T, F)$ ein Netz, $X := P \cup T$ und $Y \subseteq X$ eine Menge von Elementen. Dann heißt $\partial(Y) := \{y \in Y | \exists x \notin Y \,.\, x \in loc(y)\}$ der* Rand *(border) (der Menge Y). Y heißt* Platz-berandet *(place-bordered) oder* offen[1]*, wenn $\partial(Y) \subseteq P$, und* Transitions-berandet *(transition-bordered) oder* abgeschlossen[1]*, falls $\partial(Y) \subseteq T$. Um eine Vergröberung mit der Netzstruktur verträglich zu gestalten, sollten im Normalfall Platz- bzw. Transitions-berandete Mengen durch einen Platz bzw. eine Transition ersetzt werden.*

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.

Die Menge $Y = \{p_3, p_4, t_2, t_3, t_4\}$ des Netzes in Abb. 3.8 ist Transitions-berandet und wird daher zu einer Transition t_Y vergröbert. Auf diese Weise erhält man wieder ein Netz, das in Abb. 3.9 dargestellt ist. Diese Operation wird nun formalisiert.

Definition 3.4 *Sei $\mathcal{N} = (P, T, F)$ ein Netz und Y eine nicht leere Transitions-berandete Menge von Elementen. Dann heißt $\mathcal{N}[Y] = (P[Y], T[Y], F[Y])$* einfache Vergröberung *(simple abstraction) von $\mathcal{N}$ in Bezug auf Y falls gilt: $P[Y] = P \backslash Y$, $T[Y] = (T \backslash Y) \cup \{t_Y\}$, wobei t_Y ein neues Element ist, $F[Y] = \{(x, y) | x \notin Y \wedge y \notin Y \wedge (x, y) \in F\} \cup \{(x, t_Y) | x \notin Y \wedge \exists y \in Y \,.\, (x, y) \in F\} \cup \{(t_Y, x) | x \notin Y \wedge \exists y \in Y \,.\, (y, x) \in F\}$. $P[Y]$ enthält alle Plätze mit Ausnahme derjenigen aus Y. $T[Y]$ enthält alle Transitionen mit Ausnahme derjenigen aus Y und ein neues 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.*

Entsprechend, wenn Y eine Platz-berandete Menge ist, dann erhält man $\mathcal{N}[Y] = (P[Y], T[Y], F[Y])$ durch $P[Y] = (P \backslash Y) \cup \{p_Y\}$, wobei p_Y ein neues Element ist, $T[Y] = T \backslash Y$, $F[Y] = \{(x, y) | x \notin Y \wedge y \notin Y \wedge (x, y) \in F\} \cup \{(x, p_Y) | x \notin Y \wedge \exists y \in Y \,.\, (x, y) \in F\} \cup \{(p_Y, x) | x \notin Y \wedge \exists y \in Y \,.\, (y, x) \in F\}$.

[1] Offene und abgeschlossene Mengen definieren eine Topologie, die eine Formalisierung von Nachbarschaft auf der graphischen Struktur von Netzen darstellt.

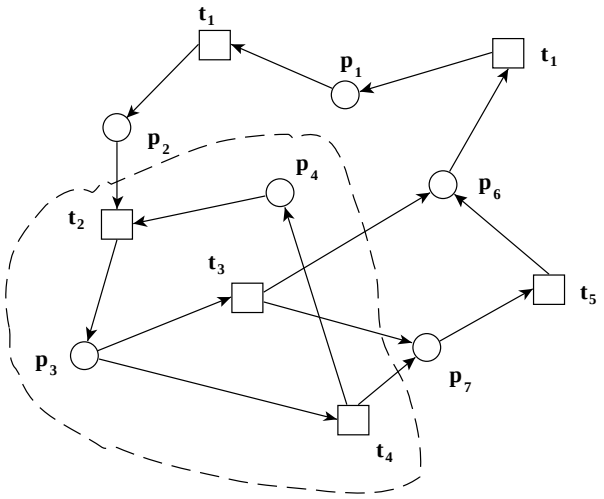


Abbildung 3.8: Eine transitionsberandete Menge

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 3.5 *a) Wenn $\mathcal{N}_2 = \mathcal{N}_1[Y]$ eine einfache Vergröberung von $\mathcal{N}_1$ für eine Platz- oder Transitions-berandete Menge Y ist, dann heißt $\mathcal{N}_1$* einfache Verfeinerung *(simple refinement) von $\mathcal{N}_2$. Für eine Menge $\{Y_1, Y_2, \ldots, Y_n\}$ von paarweise disjunkten, Platz- oder Transitions-berandeten Teilmengen von $P_1 \cup T_1$ wird $\mathcal{N}_2 = (\ldots((\mathcal{N}_1[Y_1])[Y_2])\ldots[Y_n])$* Vergröberung *(abstraction) von $\mathcal{N}_1$ genannt und $\mathcal{N}_1$ ist eine* Verfeinerung *(refinement) von $\mathcal{N}_2$. $\mathcal{N}_2$ wird durch $\mathcal{N}_2 = \mathcal{N}_1[Y_1, Y_2, \ldots, Y_n]$ bezeichnet.*

b) Eine Vergröberung $\mathcal{N}_2 = \mathcal{N}_1[Y_1, Y_2, \ldots, Y_n]$ 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 Vergröberung durch $\mathcal{N}_2 = \mathcal{N}_1[Y_1, \ldots, Y_i^{(d)}, \ldots, Y_n]$ bezeichnet werden, wobei $d = p$ bzw. $d = t$ ist und Y_i als a Platz- bzw. Transitions-berandete Menge betrachtet wird.

Abb. 3.10 zeigt eine nicht einfache Vergröberung. Das obere Netz stellt das aus der Vorlesung F4 bekannte Beispielnetz zum Start eines Autorennens dar. Dabei sind die vergröberten 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 $Y_2 = \{t_8, p_{15}, t_9\}$,

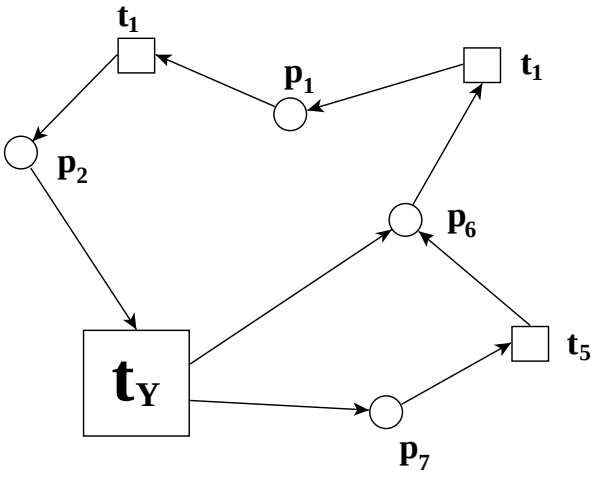


Abbildung 3.9: Vergröberung des Netzes aus Abb. 3.8

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 Vergröberung eines Netzes wieder ein Netz ist, d.h. der Definition 3.1 genügt. Die Vergröberung im unteren Teil von Abb. 3.10 hat sinngemäß das entsprechende Verhalten des Netzes darüber. Eine Vergröberung 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. 3.11 a). Intuitiv hat die Vergröberung in Abb. 3.11 d) das entsprechende Verhalten: Marken können von links nach rechts "durchlaufen". Allerdings ist dieses Netz auch Vergröberung von Abb. 3.11 b). Das Verhalten ist hier jedoch völlig verschieden.

Die Vergröberung in Abb. 3.11d) 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 3.11c dargestellt. Die Abbildungen 3.11e-h. zeigen die entsprechenden Fälle für Transitions-berandete Mengen. Es handelt sich um die *Verschmelzung* oder *Fusion* von Transitionen.

Mit Abb. 3.12 sind zwei Netze gegeben, deren Komposition durch Platzverschmelzung das Netz von Abb. 3.10 a) ergibt.

3.3 Platz/Transitions Netze

In diesem Abschnitt wird das Verhalten (die *Semantik*) der Petrinetze anhand des Modells der Platz/Transitions Netze formal eingeführt, d.h. Markierungen von Plätzen und ihre Veränderung durch das Schalten von Transitionen. Darüberhinaus wird eine kleine Erweiterung der

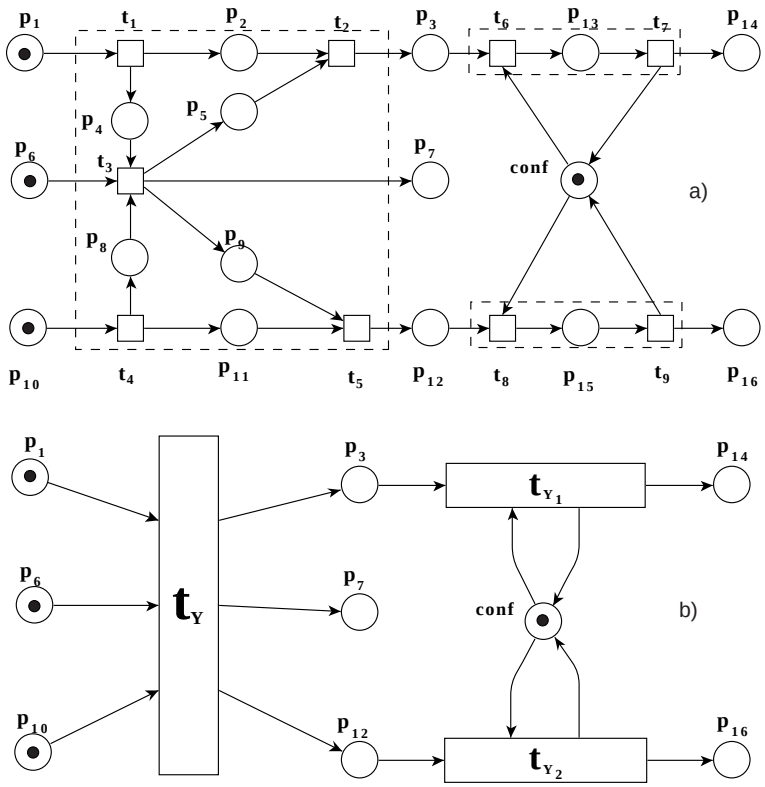


Abbildung 3.10: Eine (nicht einfache) Vergröberung

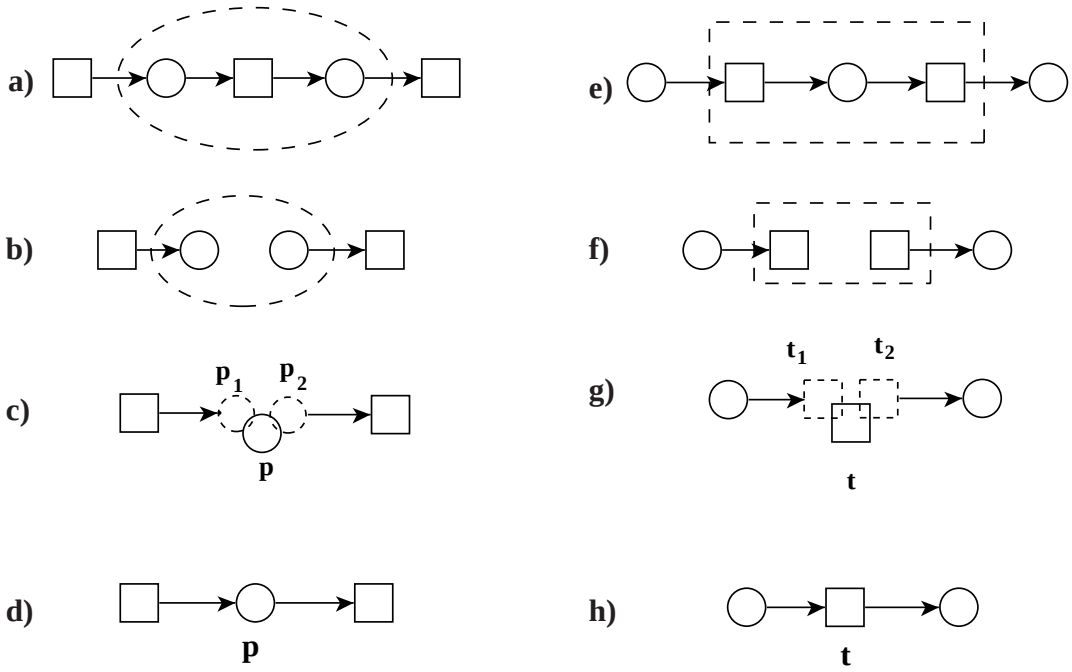


Abbildung 3.11: Vergröberung und Verschmelzung (Fusion)

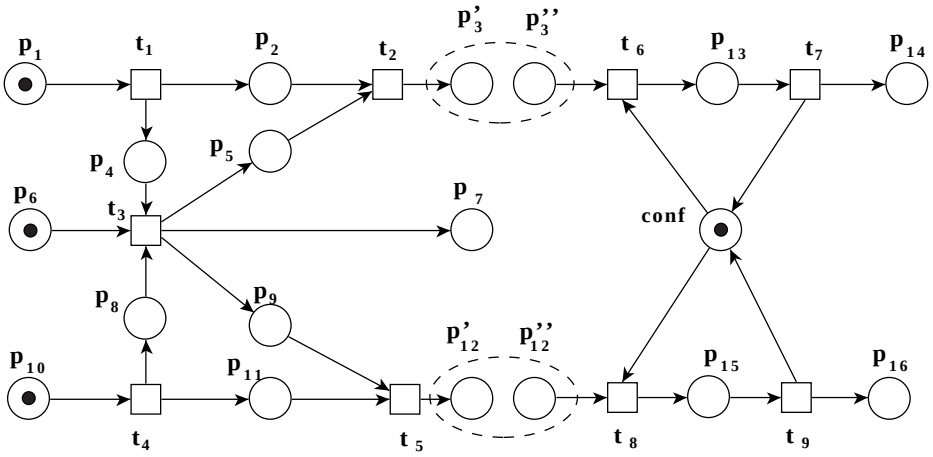


Abbildung 3.12: Komposition durch Verschmelzung (Fusion)

Netze betrachtet, die erst im Kontext ihrer Semantik sinnvoll wird: *Anfangsmarkierung* und *Kantengewicht.*

Definition 3.6 *Ein* Platz/Transitions-Netz *(*P/T-Netz*) oder Stellen/Transitions-Netz (S/T-Netz) wird als Tupel* $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ *definiert, wobei*

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

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

Beispiel: Das Netz 3.13 ist ein P/T-Netz. Die beiden Wagen sind hier als ununterscheidbare Objekte modelliert: "zwei Wagen stehen in Startposition". Die Schaltregel ist in der Abbildung 3.14 an einem abstrakten Beispiel dargestellt.

Definition 3.7 *a) Die Markierung eines* $P/T-Netzes$ $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ *ist ein Vektor* $\mathbf{m}$ *mit* $\mathbf{m}(p) \in \mathbb{N}$ *für jedes* $p \in P$ *(auch als Abbildung* $\mathbf{m} : P \to \mathbb{N}$ *aufzufassen). Die Menge aller Markierungen über* P *(bzw.* S*) wird mit* M_P *(bzw.* M_S*) bezeichnet.*

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) $\widetilde{W}(x, y) := \begin{cases} W(x, y) & \text{falls } (x, y) \in F \\ 0 & \text{sonst} \end{cases}$
ist wieder die Erweiterung von W *auf* $(P \times T) \cup (T \times P)$*.*

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

d) Definiert man $W(\bullet, t) := (\widetilde{W}(p_1, t), \ldots, \widetilde{W}(p_{|P|}, t))$ *als Vektor der Länge* $|P|$ *und entsprechend* $W(t, \bullet) := (\widetilde{W}(t, p_1), \ldots, \widetilde{W}(t, p_{|P|}))$*, dann kann die Nachfolgemarkierung einfacher durch Vektoren definiert werden:*
$\mathbf{m} \xrightarrow{t} \mathbf{m}' \ \Leftrightarrow\ \mathbf{m} \geq W(\bullet, t) \ \wedge\ \mathbf{m}' = \mathbf{m} - W(\bullet, t) + W(t, \bullet))$.
Dabei sind die Operatoren auf $\mathbb{N}$ *komponentenweise auf Vektoren zu erweitern.*

Beispiel: Im P/T-Netz $\mathcal{N}_3$ von Abb. 3.13 ist die Anfangsmarkierung der Vektor $\mathbf{m}_0 = (2, 0, 0, 0, 0, 1, 0)$ oder (alternativ) die Abbildung $\mathbf{m}_0 : P \to \mathbb{N}$ mit $\mathbf{m}_0(p_1) = 2$, $\mathbf{m}_0(p_6) = 1$ und $\mathbf{m}_0(p_i) = 0$ in den anderen Fällen. Oft ist es praktisch eine Markierung als Wort zu schreiben: $\mathbf{m}_0 = p_1^2 p_6$ oder $\mathbf{m}_0 =< p_1^2 p_6 >$.

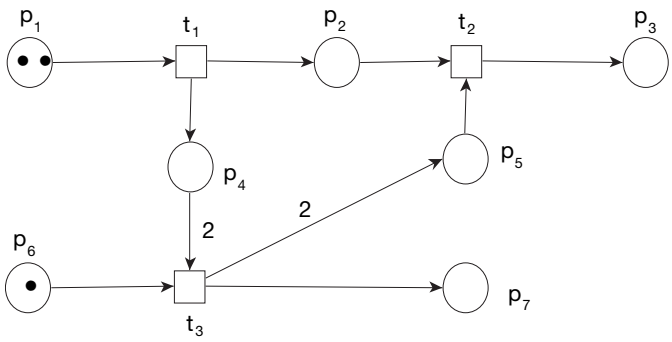


Abbildung 3.13: Platz/Transitions Netz $\mathcal{N}_3$

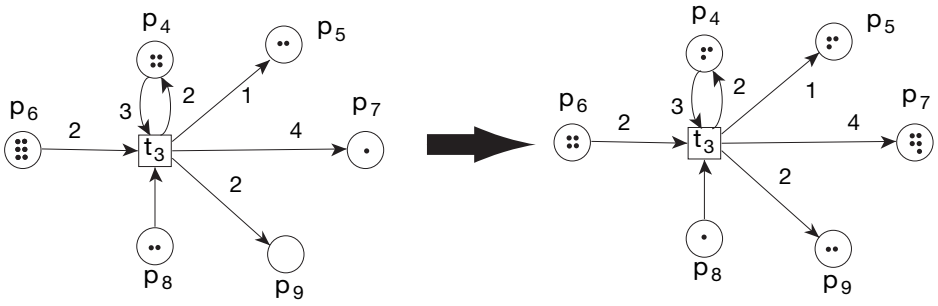


Abbildung 3.14: Schaltregel für P/T-Netze

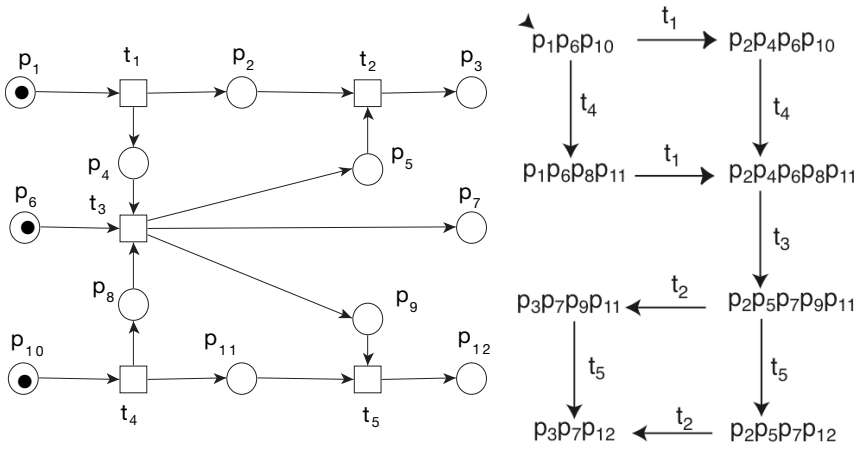


Abbildung 3.15: Erreichbarkeitsgraph des P/T-Netzes $\mathcal{N}$ von Abb. 3.5.

Definition 3.8 *Die Nachfolgemarkierungsrelation von Definition 3.7 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 $\exists \mathbf{m}_1 : \mathbf{m} \xrightarrow{w} \mathbf{m}_1$ und $FS(\mathcal{N}) := \{w \in T^* \mid \mathbf{m_0} \xrightarrow{w}\}$ ist die Menge der* Schaltfolgen *(firing sequence set) von $\mathcal{N}$.*

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

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

Werden die Markierungen als Wörter dargestellt, so erhält man:

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

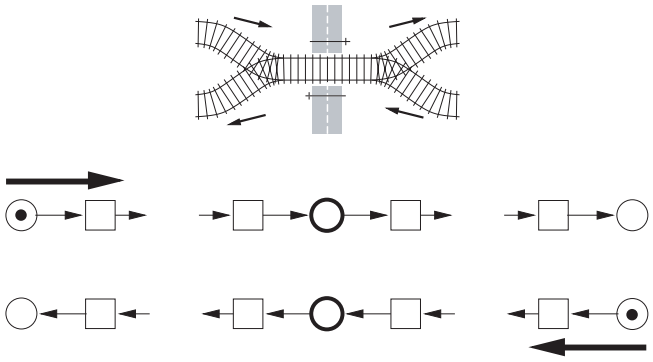
Definition 3.9 *Der Erreichbarkeitsgraph eines P/T-Netzes $\mathcal{N}$ ist ein Tupel $RG(\mathcal{N}) := (Kn, Ka)$ mit Knotenmenge $Kn := \mathbf{R}(\mathcal{N})$ und Kantenmenge $Ka := \{(\mathbf{m}_1, t, \mathbf{m}_2) \mid \mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2\}$ (siehe Beispiel in Abb. 3.15).*

Anmerkung: Der Erreichbarkeitsgraph eines P/T-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 $FS(TS) = FS(\mathcal{N})$.

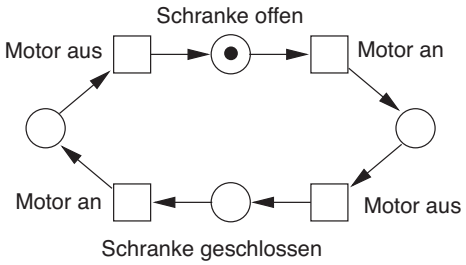
Aufgabe 3.10 (Schrankensteuerung)

Zwei Eisenbahnstrecken werden an einem beschrankten Bahnübergang auf einem einzigen Gleis geführt. (Dies ist die Standardsituation bei der AKN.)

Modellieren Sie die notwendige Synchronisation durch ein P/T-Netz, indem Sie von folgendem Netz für die beiden Gleise ausgehen. Dabei bedeutet für jedes Gleis die fett dargestellte Stelle den „kritischen Abschnitt“ (die Kreuzung). Die Marken bedeuten Züge. (Autos auf der Straße werden nicht dargestellt.) Die Transitionen stellen wichtige Aktionen des Zuges dar, wie z.B. das Vorbeifahren an einem Signal oder das Auslösen eines Sensors.



a) Lösen Sie das Problem zunächst für je einen Zug auf jeder Strecke ohne Straße und Schranken (je eine Marke wie angegeben). Signale (nicht aktivierte Transitionen) stoppen ggf. die Züge.



b) Betrachten Sie auch die Schranken und deren korrektes Öffnen und Schließen. Benutzen Sie dazu das obige Teilnetz „Motorsteuerung“ und steuern es durch Sensoren an den Gleisen.

c) Nehmen Sie zusätzlich an, dass maximal $n > 0$ (z.B. $n = 5$) Züge hintereinander auf jeder Strecke fahren können. Es sei erlaubt, dass sich mehrere Züge in einer Richtung in dem gleichen Streckenabschnitt (auf der gleichen Stelle) befinden. Dies soll so interpretiert werden, dass sich die Züge in sicherem Abstand in verschiedenen Teilen des Streckenabschnitts bewegen. Synchronisieren Sie wieder den wechselseitigen Ausschluss auf der Kreuzung und das richtige Öffnen und Schließen der Schranken. (Beispielsweise: Nicht öffnen, wenn Zug in gleicher Richtung folgt.)

3.4 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

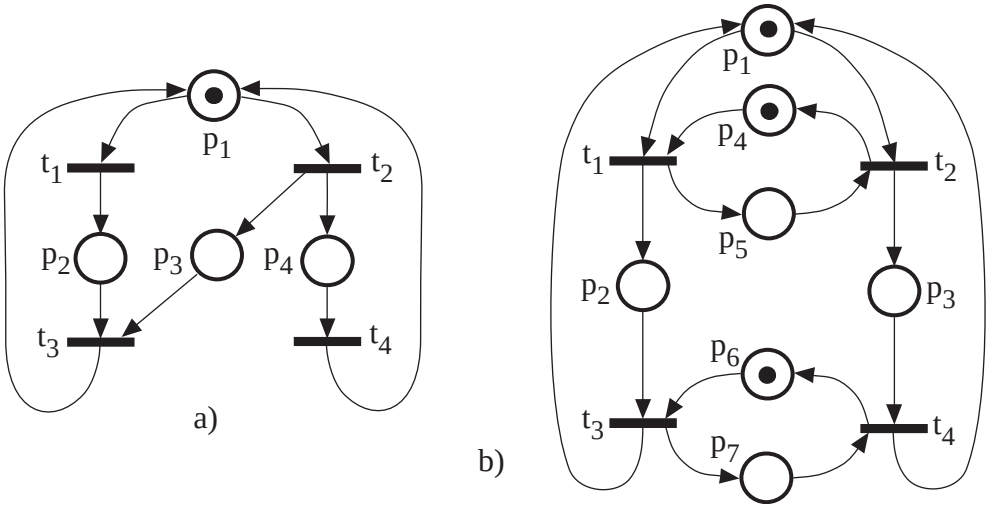


Abbildung 3.16: Beispiele: (a) ein unbeschränktes, nicht-lebendiges und nicht-reversibles Netz (b) ein lebendiges Netz, das aber bei vergrößerter Anfangsmarkierung (z.B. $\mathbf{m_0}(p_5) = 1$) nicht lebendig ist.

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.

Das Netz in Abb. 3.16 a) ist unbeschränkt, was einem „Überlauf[2]“ in realen Systemen entspricht. Darüber 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. 3.16 b) zeigt.

Für *jede* Anfangsmarkierung gelten für das Netz in Abb. 3.17 a) die folgenden Markenerhaltungsgesetze (P-Invarintengleichungen)

$$\mathbf{m}(p_1) + \mathbf{m}(p_2) + \mathbf{m}(p_3) = \mathbf{m_0}(p_1) + \mathbf{m_0}(p_2) + \mathbf{m_0}(p_3) = k_1(\mathbf{m_0})$$
$$\mathbf{m}(p_1) + \mathbf{m}(p_4) + \mathbf{m}(p_5) = \mathbf{m_0}(p_1) + \mathbf{m_0}(p_4) + \mathbf{m_0}(p_5) = k_2(\mathbf{m_0})$$
$$\mathbf{m}(p_6) = \mathbf{m_0}(p_6) = k_3(\mathbf{m_0})$$

[2]z.B. zu große Zähler, zu volle Puffer, zu viele aktivierte Prozesse

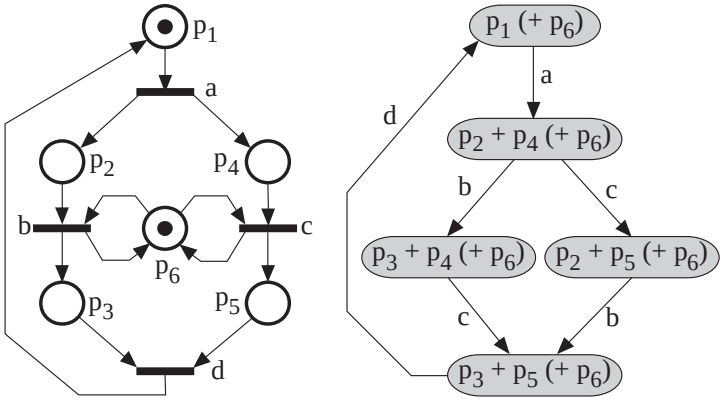


Abbildung 3.17: Beschränktes, lebendiges und reversibles Netz mit Erreichbarkeitsgraph

wobei $\mathbf{m_0}$ die Anfangsmarkierung und $\mathbf{m}$ eine beliebige erreichbare Markierung ist. $k_i(\mathbf{m_0}) \in \mathbb{Z}$ ist jeweils eine von $\mathbf{m_0}$ abhängige Konstante. Daher gelten die Ungleichungen:

$$\begin{aligned}
\mathbf{m}(p_1) &\leq \min(k_1(\mathbf{m_0}), k_2(\mathbf{m_0})) \\
\mathbf{m}(p_i) &\leq k_1(\mathbf{m_0}); i = 2, 3 \\
\mathbf{m}(p_j) &\leq k_2(\mathbf{m_0}); j = 4, 5 \\
\mathbf{m}(p_6) &= k_3(\mathbf{m_0})
\end{aligned}$$

Sie 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).

Wenn im Netz von Abb.3.16 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. 3.16 a) wählt, es ist *nicht* lebendig. Dies ist also wieder eine strukturelle Eigenschaft, die daher *strukturelle Nichtlebendigkeit* (structural non-liveness) heißt. Umgekehrt heißt ein Netz *strukturell lebendig* (structural live), wenn für es eine lebendige Markierung existiert.

Eine Markierung heißt *Rücksetzzustand* (home state), wenn sie von jeder erreichbaren Markierung erreicht werden kann. Die Anfangsmarkierung des Netzes in Abb. 3.18 a) ist kein

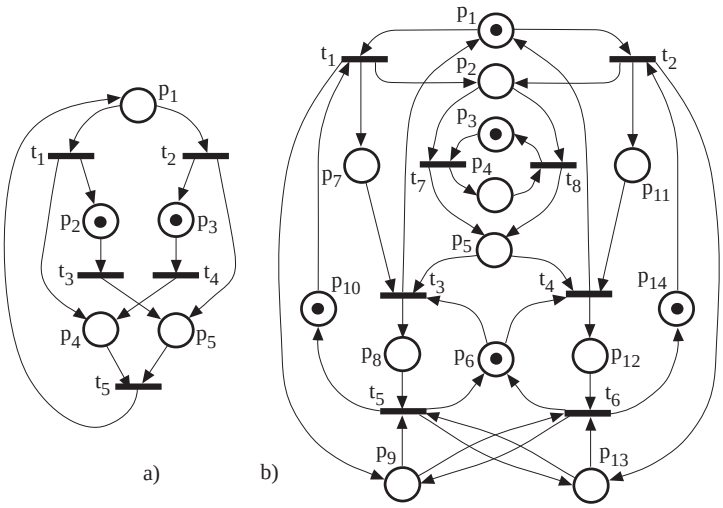


Abbildung 3.18: (a) Die Anfangsmarkierung ist kein Rücksetzzustand, jedoch alle Übrigen; (b) Die Anfangsmarkierung ist kein Rücksetzzustand, der übrige Erreichbarkeitsgraph zerfällt in zwei streng zusammenhängende Zusammenhangskomponenten (siehe Abb. 3.23).

Rücksetzzustand. Das gilt auch für das Netz in Abb 3.18 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. Durch Abb. 3.19 wird dies tatsächlich bewiesen, da alle 2^3 Kombinationen vertreten sind.

Die letzte der hier betrachteten elementaren Systemeigenschaften ist der *wechselseitige Ausschluss* (mutual exclusion). Diese Eigenschaft umfasst Bedingungen wie die Unmöglichkeit des gleichzeitigen Zugriffs zweier Roboter auf ein Objekt.

Zwei Plätze (bzw. Transitionen) befinden sich im wechselseitigen Ausschluss, wenn sie niemals gleichzeitig markiert sind (bzw. schalten). Beispielsweise gilt für das Netz in Abb. 3.17 für jede erreichbare Markierung $\mathbf{m}$ die Invariantengleichung $\mathbf{m}(p_1) + \mathbf{m}(p_2) + \mathbf{m}(p_3) = 1$. Also sind p_1, p_2, und p_3 im wechselseitigen Ausschluss.

In der Tabelle 3.1 sind die formalen Definitionen der diskutierten Eigenschaften zusammengefasst. Darin sind Notationen enthalten, die z.T. im vorandgehenden Kapitel eingeführt wurden. Um die Anfangsmarkierung $\mathbf{m_0}$ eines P/T-Netzes hervorzuheben, definiert man ein *Netzsystem* als $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$ mit $\mathcal{N} = \langle P, T, F, W \rangle$. Für ein Netzsystem $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$ wird die Erreichbarkeitsmenge durch $\mathbf{R}(\mathcal{S}) = \mathbf{R}(\mathcal{N}, \mathbf{m_0})$ bezeichnet.

Die Menge P der Plätze wird wie in der Literatur auch durch S *(„Stellen“)* bezeichnet. Ent-

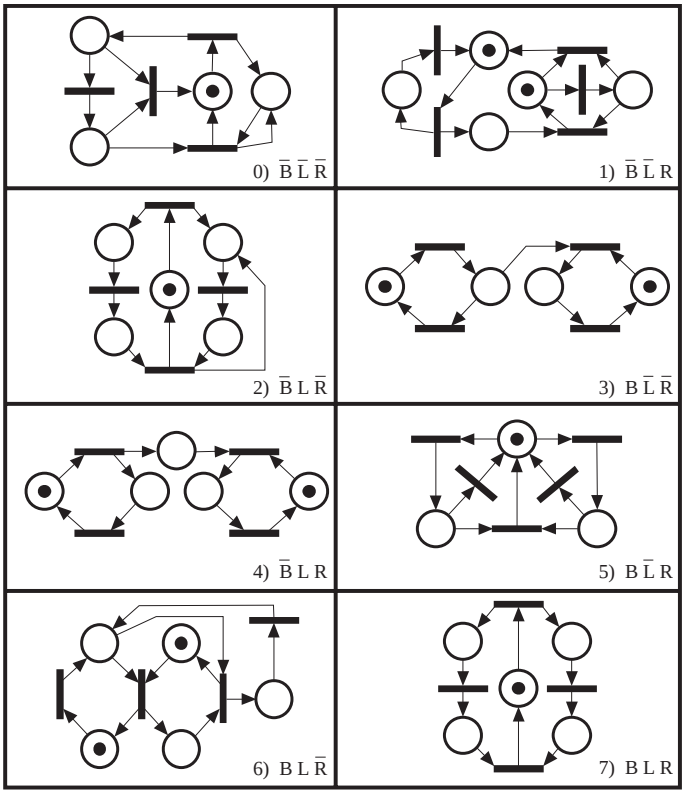


Abbildung 3.19: Beschränktheit (B), Lebendigkeit (L) und Reversibilität (R) sind unabhängige Eigenschaften

sprechend heißt das Platz/Transitions-Netz dann *Stellen/Transitions-Netz* (*S/T-Netz*).

3.5 Verifikation durch den Erreichbarkeitsgraphen

Verifikationsmethoden für Petrinetze:

a) enumerative Methoden:
   Analyse des Erreichbarkeitsgraphen

b) Transformationen
   Reduktion

c) strukturelle Analyse
   lineare Algebra-Methoden (P-Invarianten, T-Invarianten)
   graphenbasierte Methoden

(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 bzw. beschränkt, wenn alle Plätze k-beschränkt bzw. 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 *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 $\langle \mathcal{N}, \mathbf{m} \rangle$ für alle $\mathbf{m}$ beschränkt ist. $\mathcal{N}$ heißt *strukturell lebendig* (structurally live), falls $\langle \mathcal{N}, \mathbf{m} \rangle$ für mindestens ein $\mathbf{m}$ lebendig ist.

Tabelle 3.1: Systemeigenschaften eines P/T-Netzes $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$

Definition 3.11 *Der* Erreichbarkeitsgraph *eines Netzsystems $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$ ist ein gerichteter Graph* $\mathrm{RG}(\mathcal{S}) = (V, E)$*, mit Knotenmenge $V = \mathbf{R}(\mathcal{S})$ (also der Erreichbarkeitsmenge) und Kantenmenge $E = \{\langle \mathbf{m}, t, \mathbf{m'} \rangle | \mathbf{m}, \mathbf{m'} \in \mathbf{R}(\mathcal{S})$ und $\mathbf{m} \xrightarrow{t} \mathbf{m'}\}$.*

Ist das Netzsystem $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$ beschränkt, dann ist $\mathrm{RG}(\mathcal{S})$ endlich und kann konstruiert werden, z.B. durch den Algorithmus 3.1. 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.

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 3.8 zurückkommen.

Der Erreichbarkeitsgraph $\mathrm{RG}(\mathcal{S})$ eines unbeschränkten Netzsystems $\mathcal{S}$ 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 3.1 als Abbruchkriterium dient. Dies wird in folgendem Satz präzisiert.

Satz 3.12 *Ein Netzsystem $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$ ist genau dann unbeschränkt, wenn es zwei erreichbare Markierungen $\mathbf{m}_1, \mathbf{m}_2 \in \mathbf{R}(\mathcal{S})$ gibt, die folgende Bedingungen erfüllen*[3]*:*

[3] Zur Definition von $\mathbf{m} < \mathbf{m'}$ bei Vektoren siehe die Definitionen auf der Rückseite des Titelblattes und Seite

Algorithmus 3.1 (Berechnung des Erreichbarkeitsgraphen)

Input - Das Netzsystem $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$
Output - Der gerichtete Graph $\mathrm{RG}(\mathcal{S}) = (V, E)$, falls das Netzsystem beschränkt ist.

1. Initialisiere $\mathrm{RG}(\mathcal{S}) = (\{\mathbf{m_0}\}, \emptyset)$; $\mathbf{m_0}$ sei ungefärbt;
2. **while** Es gibt ungefärbte Knoten in V. **do**
   - 2.1 Wähle einen ungefärbte 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 $E := E \cup \{\langle \mathbf{m}, t, \mathbf{m}' \rangle\}$
3. Der Algorithmus terminiert mit Ergebnis. ($\mathrm{RG}(\mathcal{S})$ ist der Erreichbarkeitsgraph.)

a) $\exists \sigma \in T^* : \mathbf{m}_1 \xrightarrow{\sigma} \mathbf{m}_2$

b) $\mathbf{m}_1 < \mathbf{m}_2$

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 $\mathbf{m}_2 < \mathbf{m}_3$. Also gilt die Bedingung auch für das Paar $\mathbf{m}_2, \mathbf{m}_3$ und man erhält per Induktion $\mathbf{m}_1 \xrightarrow{\sigma} \mathbf{m}_2 \xrightarrow{\sigma} \mathbf{m}_3 \xrightarrow{\sigma} \mathbf{m}_4 \xrightarrow{\sigma} \mathbf{m}_5 \xrightarrow{\sigma} \ldots\ldots$ Wegen $\mathbf{m}_1 < \mathbf{m}_2 < \mathbf{m}_3 < \mathbf{m}_4 < \mathbf{m}_5 < \ldots\ldots$ ist das System unbeschränkt. Ist umgekehrt der Erreichbarkeitsgraph unendlich, dann gibt es[4] eine unendliche Folge $\mathbf{m}_1 \xrightarrow{*} \mathbf{m}_2 \xrightarrow{*} \mathbf{m}_3 \xrightarrow{*} \mathbf{m}_4 \xrightarrow{*} \mathbf{m}_5 \xrightarrow{*} \ldots\ldots$ mit $\mathbf{m}_1 \in \mathbf{R}(\mathcal{S})$ und $\mathbf{m}_1 < \mathbf{m}_2 < \mathbf{m}_3 < \mathbf{m}_4 < \mathbf{m}_5 < \ldots\ldots$ Also ist die Bedingung erfüllt. □

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. Das sind aussagenlogische Formeln, deren Atome Ungleichungen der Form $\sum_{p \in A} k_p \mathbf{m}(p) \leq k$ sind, wobei k_p und k rationale Konstanten und A eine Teilmenge der Plätze ist. Beispiel: $\Pi(\mathbf{m}) = (2 \cdot \mathbf{m}(p_1) + \mathbf{m}(p_2) \leq 3 \vee \mathbf{m}(p_3) \leq 1)$

Die erste Klasse von Spezifikationen heißt *Markierungs-Invarianzeigenschaften.*

Definition 3.13 *Eine* Markierungs-Invarianzeigenschaft *(marking invariance property) eines Netzsystems* $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$ *ist ein Prädikat der Form* $\forall\, \mathbf{m} \in \mathbf{R}(\mathcal{S}) : \Pi(\mathbf{m})$ *oder* $\forall\, \mathbf{m} \in \mathbf{R}(\mathcal{S})\ \exists\, t \in T : \Pi(\mathbf{m})$*, wobei* Π *ein Markierungsprädikat ist.*

Beispiele dazu sind:

109.
[4] nach dem Lemma von König

Algorithmus 3.2 (Entscheiden einer Markierungs-Invarianzeigenschaft)

Input - Der Erreichbarkeitsgraph $\mathrm{RG}(\mathcal{N}, \mathbf{m_0})$. 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 $\mathbf{R}(\mathcal{S})$ als ungefärbt.
2. **while** Es gibt einen ungefärbten Knoten $\mathbf{m} \in \mathbf{R}(\mathcal{S})$ **do**
   2.1 Wähle einen ungefärbten Knoten $\mathbf{m} \in \mathbf{R}(\mathcal{S})$ und färbe ihn.
   2.2 **if** $\mathbf{m}$ erfüllt nicht Π.
   then return FALSE (Die Eigenschaft Π ist nicht erfüllt.)
3. Return TRUE

1) *k-Beschränktheit* (k-boundedness) eines Platzes p: $\forall \mathbf{m} \in \mathbf{R}(\mathcal{S}) \,.\, \mathbf{m}(p) \leq k$.

2) *Markierungs-Ausschluss* (marking mutual exclusion) zwischen p und p':
$\forall \mathbf{m} \in \mathbf{R}(\mathcal{S}) \,.\, ((\mathbf{m}(p) = 0) \vee (\mathbf{m}(p') = 0))$[5].

3) *Verklemmungsfreiheit* (deadlock-freeness): $\forall \mathbf{m} \in \mathbf{R}(\mathcal{S}) \,.\, \exists t \in T \,.\, W(\bullet, t) \leq \mathbf{m}$.

Markierungs-Invarianzeigenschaften können mit Algorithmus 3.2 entschieden werden, dessen Zeitkomplexität linear in Bezug auf die Größe von $\mathbf{R}(\mathcal{S})$ 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 $\mathrm{RG}(\mathcal{S})$, der von $\mathbf{m_0}$ ausgeht und in einer Markierung endet, die Π nicht erfüllt.

Beispiel 3.14 Analyse von Markierungs-Invarianzeigenschaften Betrachte das Netzsystem in Abb. 3.17, für das $\mathbf{R}(\mathcal{S}) = \{p_1+p_6, p_2+p_4+p_6, p_3+p_4+p_6, p_2+p_5+p_6, p_3+p_5+p_6\}$ gilt. Die Anwendung des Algorithmus 3.2 zur Überprüfung des Markierungsausschlusses der Plätze p_5 und p_6 (d.h. $\Pi(\mathbf{m}) = (\mathbf{m}(p_5) = 0) \vee (\mathbf{m}(p_6) = 0)$) beginnt damit, dass alle Elemente von $\mathbf{R}(\mathcal{S})$ 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.

Die zweite Klasse von Spezifikationen heißt *Lebendigkeits-Invarianzeigenschaften.*

Definition 3.15 *Eine* Lebendigkeits-Invarianzeigenschaft *(liveness invariance property) eines Netzsystems $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$ ist ein Prädikat der Form*
$\forall \mathbf{m} \in \mathbf{R}(\mathcal{S}) \,.\, \exists \mathbf{m}' \in \mathbf{R}(\mathcal{N}, \mathbf{m}) \,.\, \Pi(\mathbf{m}')$*, wobei* Π *ein Markierungsprädikat ist.*

Beispiele dazu sind:

1) *Lebendigkeit von t* (liveness of t): $\forall \mathbf{m} \in \mathbf{R}(\mathcal{S}) \,.\, \exists \mathbf{m}' \in \mathbf{R}(\mathcal{N}, \mathbf{m}) \,.\, W(\bullet, t) \leq \mathbf{m}'$.

[5] = denke man sich hier mittels $\leq$ ausgedrückt.

2) $\mathbf{m}_H$ ist *Rücksetzzustand* (home state): $\forall \mathbf{m} \in \mathbf{R}(\mathcal{S}) \,.\, \exists \mathbf{m}' \in \mathbf{R}(\mathcal{N}, \mathbf{m}) \,.\, \mathbf{m}' = \mathbf{m}_H$.

3) *Reversibilität* (reversibility): $\forall \mathbf{m} \in \mathbf{R}(\mathcal{S}) \,.\, \exists \mathbf{m}' \in \mathbf{R}(\mathcal{N}, \mathbf{m}) \,.\, \mathbf{m}' = \mathbf{m_0}$.

Diese Eigenschaften lassen sich nicht wie in Algorithmus 3.2 durch lineares Durchsuchen der Erreichbarkeitsmenge überprüfen. Vielmehr muss von jeder erreichbaren Markierung eine von dieser erreichbare Markierung gefunden werden, die Π erfüllt. Dabei hilft der Begriff der *strengen Zusammenhangskomponente* eines Graphen.

Definition 3.16 *Ein* Pfad *von* $\mathbf{m}_1$ *nach* $\mathbf{m}_k$ *in einem Erreichbarkeitsgraphen* $\mathrm{RG}(\mathcal{S}) = (V, E)$ *ist eine Folge* $\mathbf{m}_1 \ldots \mathbf{m}_i \mathbf{m}_{i+1} \ldots \mathbf{m}_k$ $(k \geq 2)$ *seiner Knoten mit* $\langle \mathbf{m}_i, t_i, \mathbf{m}_{i+1} \rangle \in E$ *für alle* $i \in \{1, \ldots, k-1\}$ *und jeweils ein* $t_i \in T$. *Ein Teilgraph von* $\mathrm{RG}(\mathcal{S})$ *heißt* streng zusammenhängend *(strongly connected) 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) *(strongly connected component) von* $\mathrm{RG}(\mathcal{S})$. *Sie heißt* triviale Zusammenhangskomponente *falls sie nur aus einem Knoten ohne Schleife besteht und* 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]). Abb. 3.21 zeigt die strengen Zusammenhangskomponenten des Graphen von Abb. 3.20, von denen zwei terminal sind. Zwei seiner SZKs sind trivial.

Die Knotenmengen verschiedener SZKs sind natürlich disjunkt. Daher kann man sie in eindeutiger Weise zu einem Knoten zusammenziehen. Wenn man die verbleibenden Kanten sinngemäß überträgt, erhält man seinen *reduzierten Graphen.*

Definition 3.17 *Der* reduzierte Graph $\mathrm{RG}^c(\mathcal{S}) = (V_c, E_c)$ *eines Erreichbarkeitsgraphen* $\mathrm{RG}(\mathcal{S}) = (V, E)$ *hat als Knotenmenge die SZKs von* $\mathrm{RG}(\mathcal{S})$*, d.h.* $V_c := \{C_1, \ldots, C_r\}$. *Die Kantenmenge ist* $E_c := \{\langle C_i, t, C_j \rangle |$ *Es gibt eine Kante* $\langle \mathbf{m}_i, t, \mathbf{m}'_j \rangle \in E$ *derart, dass* $\mathbf{m}$ *bzw.* $\mathbf{m}'$ *in SZKs* C_i *bzw.* C_j *liegen und* $C_i \neq C_j$ *gilt.*$\}$

In Abb. 3.22 ist der reduzierte Graph von Abb. 3.20 dargestellt. Den beiden terminalen SZK entsprechend, hat der azyklische Graph zwei Knoten, von denen keine Kanten ausgehen. Als Anfangsknoten wird der Knoten definiert, der aus derjenigen SZK entstanden ist, die den Anfangsknoten enthält (vorausgesetzt, der Ausgangsgraph hatte einen solchen).

Da der reduzierte Graph $\mathrm{RG}^c(\mathcal{S})$ immer azyklisch ist, können seine terminalen SZKs in linearer Zeitkomplexität berechnet werden. Dies wird im Algorithmus 3.3 zur Prüfung von Lebendigkeits-Invarianzeigenschaften ausgenutzt. Dieser hat daher auch eine lineare Zeitkomplexität (in Bezug auf die Größe von $\mathrm{RG}(\mathcal{S})$). Wenn der Algorithmus erfolgreich terminiert,

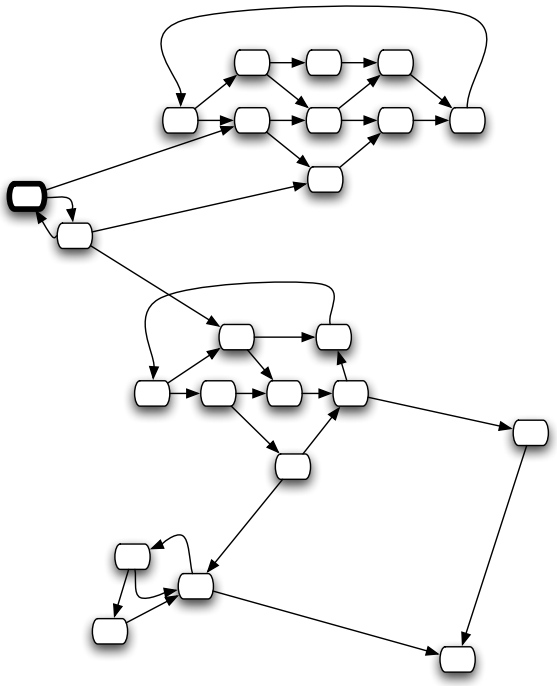


Abbildung 3.20: Ein Graph mit Anfangsknoten (fett)

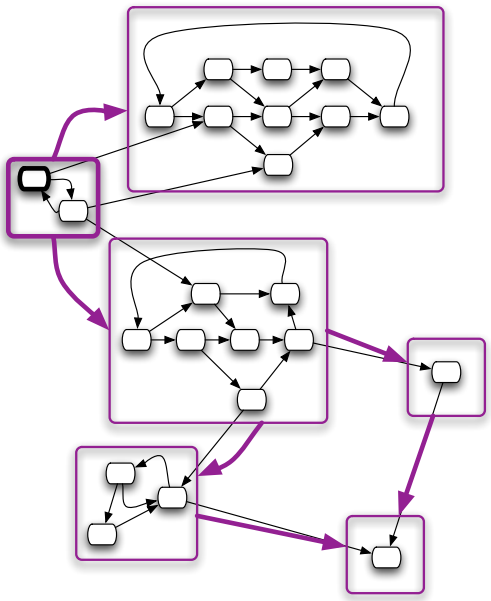


Abbildung 3.21: Der Graph von Abb. 3.20 mit seinen SZKs

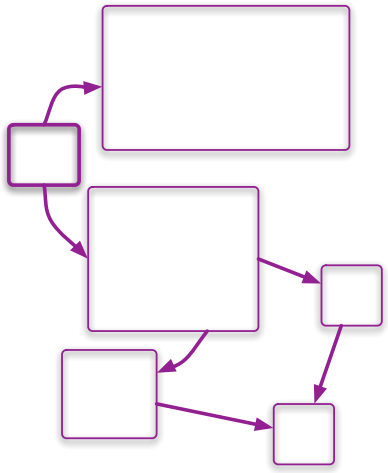


Abbildung 3.22: Der reduzierte Graph des Graphen von Abb. 3.20

Algorithmus 3.3 (Entscheiden einer Lebendigkeits-Invarianzeigenschaft)

Input - Der Erreichbarkeitsgraph $\mathrm{RG}(\mathcal{N}, \mathbf{m_0})$. 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) $C_1, \ldots, C_r$ von $\mathrm{RG}(\mathcal{N}, \mathbf{m_0})$.
2. Berechne den Graphen $\mathrm{RG}^c(\mathcal{S}) = (V_c, E_c)$ durch Wandeln der SZKs $C_1, \ldots, C_r$ in jeweils einen Knoten, d.h. $V_c = \{C_1, \ldots, C_r\}$. $\langle C_i, t, C_j \rangle \in E_c$ genau dann, wenn eine Kante $\langle \mathbf{m}, t, \mathbf{m}' \rangle \in E$ derart existiert, dass $\mathbf{m}$ in der SZK C_i und $\mathbf{m}'$ in der SZK C_j liegt und $i \neq j$ gilt.
3. Berechne die Menge F der terminalen SZKs von $\mathrm{RG}^c(\mathcal{S})$.
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

---

enthalten alle terminalen SZKs eine Markierung, die die Eigenschaft Π erfüllt. Daher existiert zu jeder erreichbaren Markierung eine Nachfolgemarkierung, die Π erfüllt. Falls der Algorithmus nicht erfolgreich terminiert, enthält mindestens eine terminale SZK eine Markierung, die die Eigenschaft Π nicht erfüllt.

Anmerkung: Man kann effektivere Algorithmen entwickeln, wenn man besondere Charakteristiken der zu prüfenden Eigenschaft *oder* des zu analysierenden Systems kennt.

Als Beispiel für den ersteren 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.

Beispiel 3.18 Analyse einer Lebendigkeits-Invarianzeigenschaft Wir betrachten das Netzsystem von Abb.3.18 b) mit dem Erreichbarkeitsgraphen von Abb. 3.23. Um den Algorithmus 3.3 auszuführen, müssen zunächst die SZKs berechnet werden. Diese sind schon in Abb. 3.23 durch die Bereiche C_1, C_2 und C_3 eingezeichnet, wobei C_2 und C_3 terminal sind. Wir prüfen die Lebendigkeitseigenschaft $(\forall \mathbf{m} \in \mathbf{R}(\mathcal{S}) \,.\, \forall t \in T \,.\, \left[\exists \mathbf{m}^t \in \mathbf{R}(\langle \mathcal{N}, \mathbf{m} \rangle) \,.\, \mathbf{m}^t \geq W(\bullet, t)\right])$.

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 Netzsystem lebendig.

Führt man Algorithmus 3.3 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.

Auch vergleichsweise kleine Komponenten eines System müssen rechnergestützt verifiziert werden. Daher hat das Forschungsgebiet *rechnergestützte Verifikation* (computer-aided verification) große praktische Bedeutung[6]. Es folgt als Beispiel ein kleines Netzsystem, das nur schwer ohne Rechnerunterstützung zu analysieren ist.

Beispiel 3.19 Abb. 3.24 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.

3.6 Fairness und Netzinvarianten

Fairness und Netzinvarianten sind zwei wichtige Erscheinungen der Verifikation von Systemen. Fairness ist verwandt mit Lebendigkeit und wird später im Zusammenhang mit temporallogischen Spezifikationen wieder aufgegriffen werden. Netzinvarianten erlauben die Verifikation von Markierungsinvarianzeigenschaften ohne die oft exponentielle Konstruktion des Erreichbarkeitsgraphen.

[6]Die *International Conference on Computer-Aided Verification* ist ein wichtiges Forum für neue Ergebnisse auf diesem Gebiet (siehe *Lecture Notes in Computer Science (LNCS)* **407** (1989), **531** (1990), **575** (1991), **663** (1992), **697** (1993), **818** (1994), **939** (1995), **1102** (1996), **1254** (1997), **1427** (1998), **1633** (1999)).

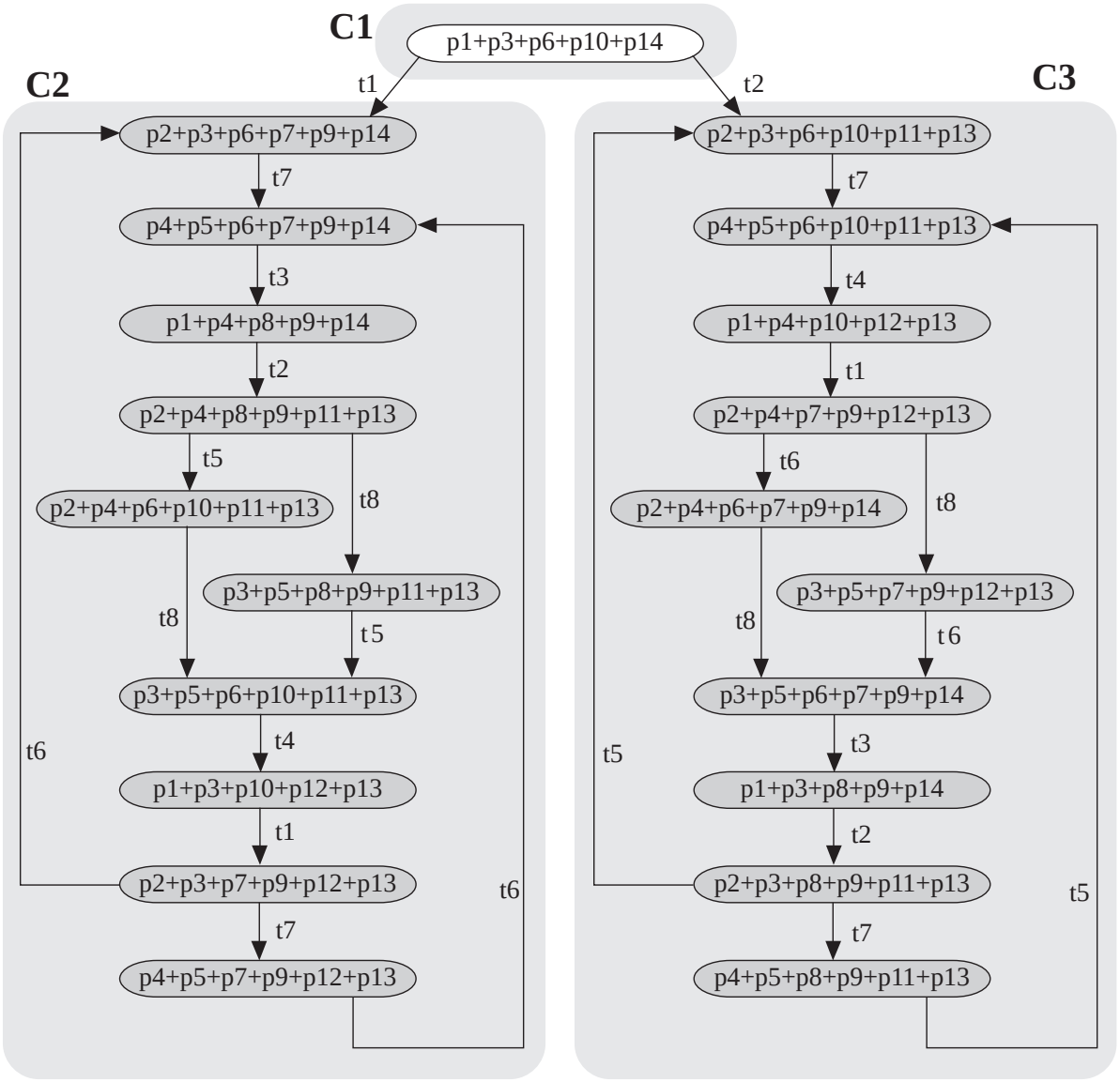


Abbildung 3.23: Erreichbarkeitsgraph des Netzsystems von Abb. 3.18.b

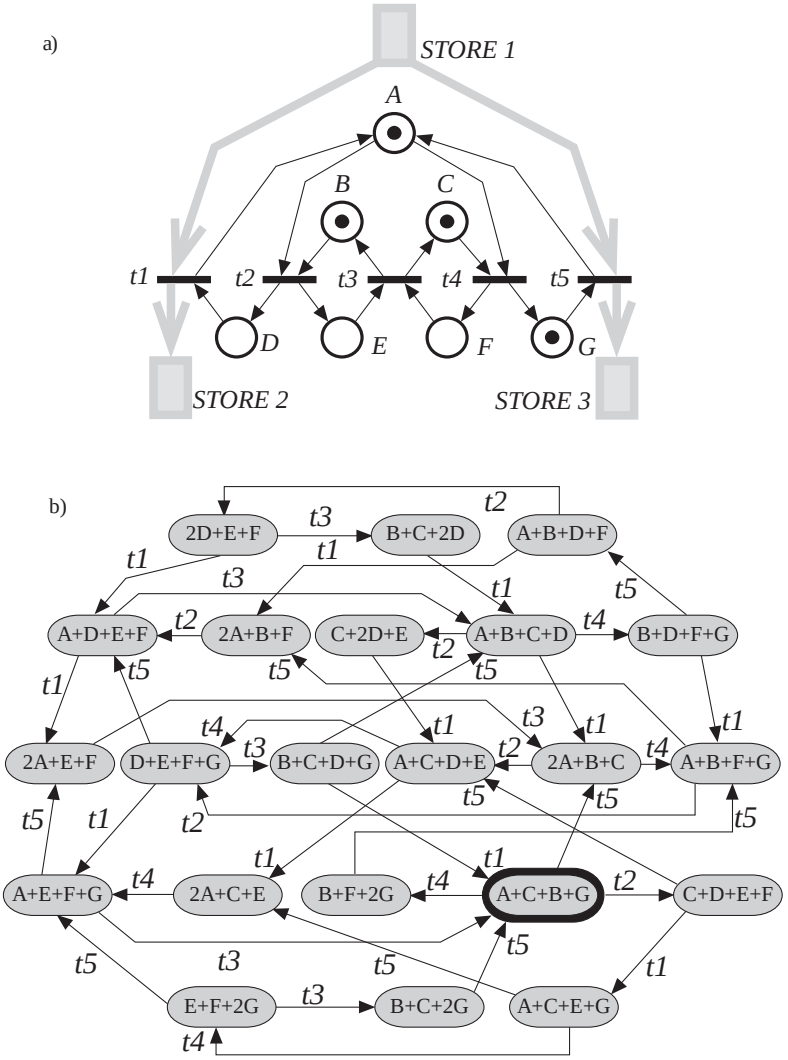


Abbildung 3.24: Ein kleines Netzsystem (a) mit komplexem Erreichbarkeitsgraph in (b)

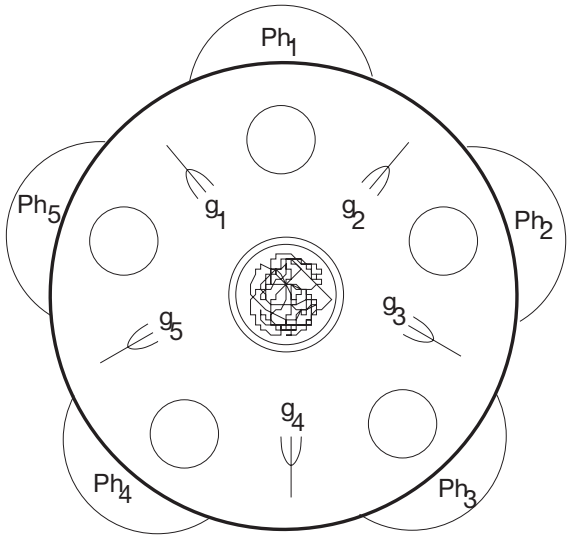


Abbildung 3.25: Die fünf Philosophen am Tisch (nach Dijkstra)

3.6.1 Fairness

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

Fünf Philosophen $Ph_1, \ldots, Ph_5$ sitzen an einem runden Tisch, in dessen Mitte eine Schüssel mit Spaghetti steht (Abb. 3.25). Jeder Philosoph Ph_i befindet sich entweder im Zustand des "Denkens" (d_i) oder "Essens" (e_i). Zum Essen stehen insgesamt nur 5 Gabeln $g_1, \ldots, g_5$ (die Betriebsmittel) zur Verfügung, jeweils eine zwischen zwei benachbarten Philosophen. Geht ein Philosoph vom Denken zum Essen über, nimmt er erst die rechte und dann die linke Gabel auf.

Abbildung 3.26 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. 3.26 dargestellten Vergröberung übergehen. Nun ist das Netz zwar lebendig, aber es besteht immer noch die Möglichkeit, dass zwei Philosophen, etwa Ph_1 und Ph_3 so die Gabel benutzen, dass Ph_2 nie die Chance hat, seine Gabeln aufzunehmen. Man sagt, für den Philosophen Ph_2 besteht die Gefahr des Verhungerns (*starvation*), oder die Philosophen Ph_1 und Ph_3 verhalten sich unfair gegenüber Ph_2.

Definition 3.20 *Ein P/T-Netz $\mathcal{N} = \langle S, T, F, W, \mathbf{m}_0 \rangle$ 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 bedeutet $F_\omega(\mathcal{N}) := \{w = a_1 a_2 a_3 \cdots \in T^\omega | \forall i \in Nat^+ : a_1 a_2 a_3 \cdots a_i \in FS(\mathcal{N})\}$ (Definition von $FS(\mathcal{N})$ siehe Definition 3.8 auf Seite 78).*

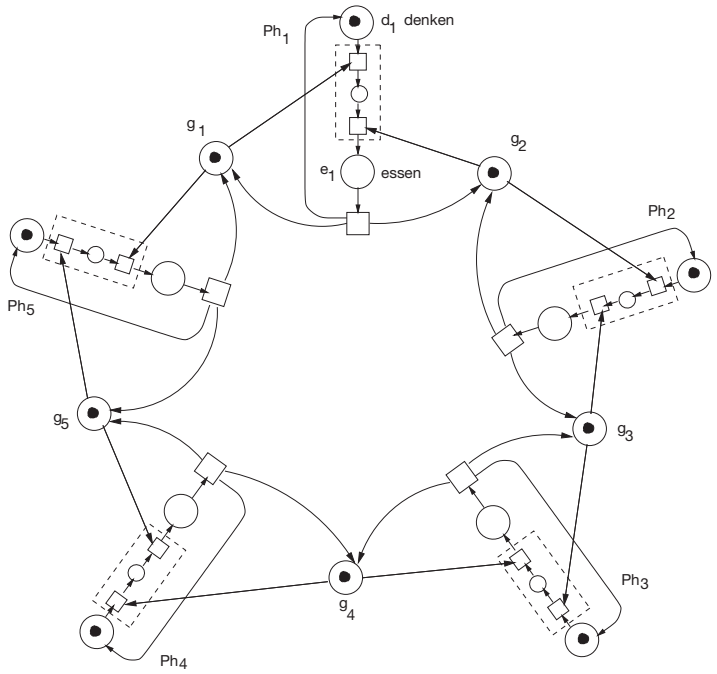


Abbildung 3.26: Die fünf Philosophen als P/T-Netz

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.3.27 kann natürlich die faire Folge

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

wie die unfaire

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

schalten. Dieses Netz entspricht in gewisser Weise dem Programm

do a $\rightarrow$ c
$\Box$ b $\rightarrow$ d
od

worin c und d Anweisungen sind, die a und b nicht verändern und letztere mit **true** initialisiert sind. Die **do**...**od** - Klammer bezeichnet eine Schleifenanweisung mit nichtdeterministischer Ausführung des Schleifenkörpers und geschützten Anweisungen (*guarded commands*).

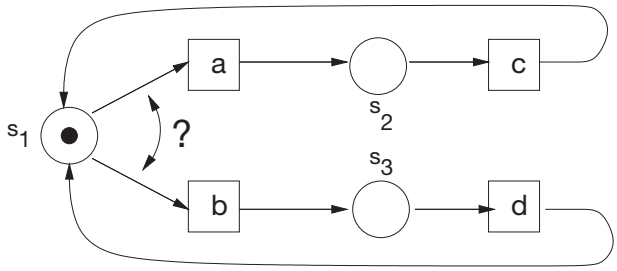


Abbildung 3.27: Netz mit unfairem Verhalten

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 3.21 *Es sei $\mathcal{N} = \langle S, T, F, W, \mathbf{m}_0 \rangle$ ein P/T-Netz und $t \in T$.*

a) t schaltet produktiv *oder* verschleppungsfrei *oder* nach der verschleppungsfreien Schaltregel *(finite delay property), wenn*

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

b) t schaltet fair*, oder* nach der fairen Schaltregel *(fair firing rule) , wenn*

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

b) 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. Die **do**...**od** - Klammer bezeichnet wieder eine Schleifenanweisung mit nichtdeterministischer Ausführung des Schleifenkörpers und geschützten Anweisungen (*guarded commands*).

```
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 3.22

a) Das Netz von Abb.3.26 ist zwar lebendig, hat aber auch unter der verschleppungsfreien Schaltregel kein faires Verhalten ($ac\ ac\ \ldots$ ist immer noch möglich). Das Netz verhält sich jedoch fair bei der fairen Schaltregel.

b) Das Netz von Abb.3.26 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 Netz von Abb.3.26 ohne Vergröberung ist nicht lebendig. Unter der fairen Schaltregel hat es jedoch ein faires Verhalten. Verhindert man die Verklemmung durch andere Maßnahmen, z.B. indem man höchstens 4 Philosophen in den Essraum lässt (Abb. 3.28), dann ist das Netz lebendig und fair bei der fairen Schaltregel.

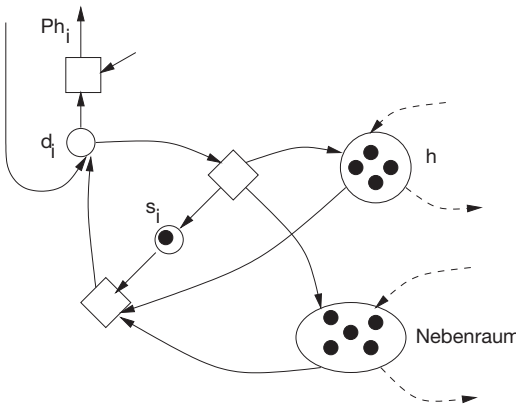


Abbildung 3.28: Philosophenproblem mit Nebenraum

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

Aufgabe 3.23 (5 Philosophen)

Es sollen Invarianten in Petrinetzen gefunden und Systemeigenschaften bewiesen werden – soweit möglich mit Hilfe der gefundenen Invarianten. Sei N das Petrinetz in der Abbildung 3.26, wobei die gestrichelten, eingefassten Gebiete eine Transition sind, sei N' das verfeinerte Netz (ohne Zusammenfassen der eingefassten Transitionen). Finden Sie für beide Netze die Invarianten und geben Sie ihnen inhaltliche Bedeutungen (z.B. „Die Zahl der Philosophen ist konstant“). Beweisen Sie für beide Netze:

a) Wenn ein Philosoph i isst, sind die Stellen g_i und g_{i+1} unmarkiert.

b) Nie essen zwei Nachbarn gleichzeitig.

c) Auf jeder Stelle liegt jeweils höchstens eine Marke.

d) Die Netze sind verklemmungsfrei – oder geben Sie eine Verklemmung an.

Zusatzaufgabe :

e) Das Netz N erlaubt verschleppungsfreie bzw. faire Schaltfolgen, in denen nicht alle Philosophen essen. Geben Sie eine Folge nach der verschleppungsfreien und eine nach der fairen Schaltregel hierfür an.

3.6.2 Netzinvarianten

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 Invarianten 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]. Dort werden Plätze wie in vielen deutschsprachigen Büchern als *Stellen* bezeichnet. Entsprechend werden die Buchstaben s und S benutzt. P/T-Netze heißen *S/T-Netze*.

Als Beispiel betrachten wir das Leser/Schreiber-Problem. Die folgende Abb. 3.29 a) zeigt ein entsprechendes Modell als S/T-Netz (oder P/T-Netz) für n Aufträge ($n \in \mathbb{N}, n > 0$), die Anfangs in der Stelle (dem Platz) *lok* liegen, was ausdrückt, dass sie noch nichts mit dem kritischen Abschnitt zu tun haben, auf den sie später lesend oder schreibend zugreifen. Durch das Schalten der Transition a bzw. d melden sie sich als Lese- bzw. Schreibaufträge an, d.h. die entsprechende Marke liegt in *la* bzw. *sa*, was “zum Lesen angemeldet” bzw. “zum Schreiben angemeldet” heißen soll. Dann folgt der Zugriff auf die kritischen Daten mit den bekannten Spezifikationen:

a) Wenn ein Schreiber schreibt, darf kein Leser lesen

b) Es darf höchstens ein Schreiber schreiben

c) Wenn ein Leser liest, darf kein Schreiber schreiben

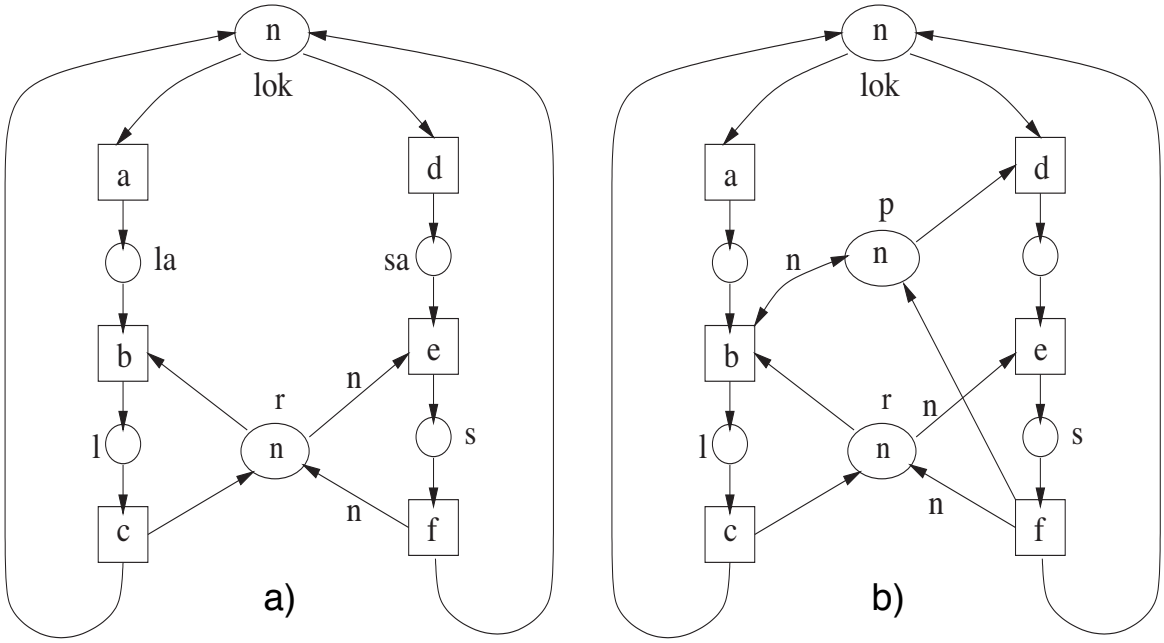


Abbildung 3.29: Leser/Schreiber-Problem in a) und mit Priorität für Schreibaufträge in b)

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

$$\mathbf{m}(s) > 0 \;\Rightarrow\; \mathbf{m}(l) = 0 \tag{3.1}$$

$$\mathbf{m}(s) \leq 1 \tag{3.2}$$

$$\mathbf{m}(l) > 0 \Rightarrow \mathbf{m}(s) = 0 \tag{3.3}$$

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 3.29 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).

Für das S/T-Netz des Leser/Schreiber-Problems in Abb. 3.29 a) gelten die Gleichungen ($n \geq 1$):

$$i_1: \quad lok + la + sa + l + s = n \tag{3.4}$$

$$i_2: \quad l + r + n \cdot s = n \tag{3.5}$$

$lok, la, \ldots$ stehen hier abkürzend für $\mathbf{m}(lok), \mathbf{m}(la), \ldots$. Gleichungen dieser Form heißen S-Invarianten-Gleichungen. Sie haben folgende allgemeine Form:

Definition 3.24 *Es sei $\mathcal{N} = \langle S, T, F, W, \mathbf{m}_0 \rangle$ ein S/T-Netz mit $S = \{s_1, \ldots, s_p\}$. Eine Gleichung der Form $\sum_{i=1}^{p} k_i \cdot \mathbf{m}(s_i) = k$ mit $k_i, k \in \mathbb{Z}$ heißt S-Invarianten-Gleichung[7], wenn sie für alle in $\mathcal{N}$ erreichbaren Markierungen $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ gilt. Abkürzend wird sie auch als $\sum_{i=1}^{p} k_i \cdot s_i = k$ geschrieben.*

Eine S-Invarianten-Gleichung ist also eine spezielle Markierungs-Invarianzeigenschaft (Definition 3.13 auf Seite 85). Der Beweis, dass obige S-Invarianten-Gleichungen für alle erreichbaren Markierungen gelten, stellen wir zurück, leiten aber aus ihnen die Synchronisationsspezifikationen ab.

- Spezifikation[8] 3.1: $s > 0 \rightarrow l = 0$ folgt aus i_2.
- Spezifikation 3.2: $s \leq 1$ folgt aus i_2.
- Spezifikation 3.3: $l > 0 \rightarrow s = 0$ folgt aus i_2.

Damit ist nachgewiesen, dass das Netz von Abb. 3.29 a) die Synchronisationsspezifikationen des Leser/Schreiber-Problems erfüllen. Wir zeigen nun, wie man die Gültigkeit der S-Invarianten-Gleichungen nachweist. Dazu betrachten wir das S/T-Netz von Abb. 3.29 a) und die S-Invarianten-Gleichung i_2:

$$i_2: \mathbf{m}(l) + \mathbf{m}(r) + n \cdot \mathbf{m}(s) = n \tag{3.6}$$

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

Als Induktionsbeweis zeigt man dann:

Beweis:
Gilt 3.6 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 3.6 auch für $\mathbf{m}_2$.

Für $t = e$ gilt

$$\begin{aligned} \mathbf{m}_2(sa) &= \mathbf{m}_1(sa) - 1 \\ \mathbf{m}_2(r) &= \mathbf{m}_1(r) - n \\ \mathbf{m}_2(s) &= \mathbf{m}_1(s) + 1, \end{aligned} \tag{3.7}$$

während alle anderen Stellen unverändert bleiben. Gilt 3.6 für $\mathbf{m}_1$, dann auch für $\mathbf{m}_2$. Dies ist für alle Transitionen durchzuführen. □

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

[7] Zur Unterscheidung zu dem bald definierten S-Invarianten-Vektor.
[8] Wieder steht $s > 0$ für $\mathbf{m}(s) > 0$ usw.

Definition 3.25 *Es sei $\mathcal{N} = \langle S, T, F, W, \mathbf{m}_0 \rangle$ ein S/T-Netz mit $S = \{s_1, \ldots, s_p\}$ und $T = \{t_1, \ldots, t_q\}$. Der Vektor $\Delta_{\mathcal{N}}(t) \in \mathbb{Z}^p$ heißt* Wirkung *der Transition $t \in T$ und ist definiert durch*

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

Die durch Aneinanderreihung der Vektoren $\Delta_{\mathcal{N}}(t_1) \ldots \Delta_{\mathcal{N}}(t_q)$ gebildete $(p \times q)$-Matrix $\Delta_{\mathcal{N}}$ heißt 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 3.26 *Wenn t die Markierung $\mathbf{m}_1$ durch Schalten in $\mathbf{m}_2$ überführt ($\mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2$), dann gilt:*

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

Der Beweis folgt direkt durch Vergleich der Definitionen 3.7 und 3.25.

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

Satz 3.27 *(Lautenbach)*
Es sei M^{tr} die Transponierte einer Matrix M und $\underline{0} \in \mathbb{Z}^{|T|}$ der Nullvektor. Ist $\mathcal{N} = \langle S, T, F, W, \mathbf{m}_0 \rangle$ ein S/T-Netz mit Inzidenzmatrix $\Delta_{\mathcal{N}}$ und $i \in \mathbb{Z}^{|S|}$ eine ganzzahlige Lösung des linearen Gleichungssystems

$$\Delta_{\mathcal{N}}^{tr} \cdot i = \underline{0}$$

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:
(durch Indukion über $\mathbf{R}(\mathcal{N})$):
Die Behauptung ist trivial für $\mathbf{m} = \mathbf{m}_0$.
Es gelte die Behauptung für $\mathbf{m}_1 \in \mathbf{R}(\mathcal{N})$, also $i^{tr} \cdot \mathbf{m}_1 = i^{tr} \cdot \mathbf{m}_0$, und es gelte $\mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2$ für eine Transition $t \in T$. 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 3.26 die Induktionsbehauptung:

$$\begin{aligned} i^{tr} \cdot \mathbf{m}_2 &= i^{tr} \cdot (\mathbf{m}_1 + \Delta_{\mathcal{N}}(t)) \\ &= i^{tr} \cdot \mathbf{m}_1 + i^{tr} \cdot \Delta_{\mathcal{N}}(t) \\ &= i^{tr} \cdot \mathbf{m}_1 \\ &= i^{tr} \cdot \mathbf{m}_0 \end{aligned}$$

□

Definition 3.28 *Jede ganzzahlige Lösung* $i \in \mathbb{Z}^{|S|} \setminus \{\underline{0}\}$ *von* $\Delta_{\mathcal{N}}^{tr} \cdot i = \underline{0}$ *heißt* S-Invarianten-Vektor[9] *des* S/T*-Netzes* $\mathcal{N}$.

Wir interpretieren Satz 3.27 anhand unseres Beispiels. Abbildung 3.30 zeigt die Inzidenzmatrix $\Delta_{\mathcal{N}}$ des Netzes von Abb. 3.29 a) und zwei Invarianten-Vektoren i_1 und i_2. Man rechne nach: $\Delta_{\mathcal{N}}^{tr} \cdot i_1 = \underline{0}$ und $\Delta_{\mathcal{N}}^{tr} \cdot i_2 = \underline{0}$.

$\Delta_{\mathcal{N}}$	a	b	c	d	e	f		i_1	i_2
lok	-1	0	1	-1	0	1		1	0
la	1	-1	0	0	0	0		1	0
sa	0	0	0	1	-1	0		1	0
l	0	1	-1	0	0	0		1	1
s	0	0	0	0	1	-1		1	n
r	0	-1	1	0	$-n$	n		0	1
j	3	3	3	2	2	2			

Abbildung 3.30: Inzidenzmatrix $\Delta_{\mathcal{N}}$ mit S-Invariantenvektor i_1, i_2 und T-Invariantenvektor j

Folglich gilt nach Satz 3.27 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 S-Invarianten-Gleichung 3.6.

Wir fassen zusammen:

Aus einem gegebenen S/T-Netz $\mathcal{N}$ können durch Berechnung aller ganzzahligen Lösungen i in $\Delta_{\mathcal{N}}^{tr} \cdot i = \underline{0}$ alle S-Invarianten-Vektoren gefunden werden. Für die Anfangsmarkierung $\mathbf{m}_0$ werden durch $i^{tr} \cdot \mathbf{m} = i^{tr} \cdot \mathbf{m}_0$ die entsprechenden S-Invarianten-Gleichung aufgestellt, die für alle $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ gelten. Mit ihnen können, wie oben gezeigt, Netzeigenschaften nachgewiesen werden.

Auch die Lösungen j von $\Delta_{\mathcal{N}} \cdot j = \underline{0}$ haben eine wichtige Interpretation für die Analyse von Systemen. (Hier $\Delta_{\mathcal{N}}$ wird nicht transponiert!)

Definition 3.29 *Jede Lösung* $j \in \mathbb{N}^{|T|} \setminus \{\underline{0}\}$ *von* $\Delta_{\mathcal{N}} \cdot j = \underline{0}$ *heißt* T-Invarianten-Vektor *des* S/T*-Netzes* $\mathcal{N}$.

[9] Im Gegensatz zur S-Invarianten-Gleichung in Def. 3.24.

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 $\psi(w) := (|w|_{x_1}, |w|_{x_2}, \ldots, |w|_{x_{|T|}})$ von dem Wort w ein T-Invarianten-Vektor. T-Invarianten beschreiben also die Häufigkeit des Schaltens jeder Transition bei reproduzierendem Verhalten[10].

Definition 3.30 *Sei $\Sigma = \{x_1, x_2, \ldots, x_k\}$ ein endliches, geordnetes Alphabet, dann heißt der Homomorphismus $\psi : \Sigma^* \to \mathbb{N}^k$* Parikh-Abbildung *und das Bild $\psi(w) := (|w|_{x_1}, |w|_{x_2}, \ldots, |w|_{x_k})$* Parikh-Vektor*, der in der i-ten Komponente angibt, wie oft das Zeichen x_i in dem Wort $w \in \Sigma^*$ vorkommt, hier mit $|w|_{x_i}$ notiert.*

Satz 3.31 *Es seien $\mathbf{m}_1, \mathbf{m}_2$ Markierungen und $w = t_{i_1} \ldots t_{i_k}$ eine Schaltfolge, die $\mathbf{m}_1$ in $\mathbf{m}_2$ überführt: $\mathbf{m}_1 \xrightarrow{w} \mathbf{m}_2$. Die Markierungen $\mathbf{m}_1$ und $\mathbf{m}_2$ sind genau dann gleich, wenn es einen T-Invarianten-Vektor $j \in \mathbb{N}^{|T|}$ derart gibt, dass jede Transition $t_i \in T$ genau $j(i)$ mal in w vorkommt, d.h.: $j(i) = \psi(w)(i)$*

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

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. 3.30 besagt also, dass die Anfangsmarkierungen $\mathbf{m}_0$ dann wieder erreicht (reproduziert) wird, wenn drei Lese- und zwei Schreibaufträge ihren Zyklus durchlaufen haben.

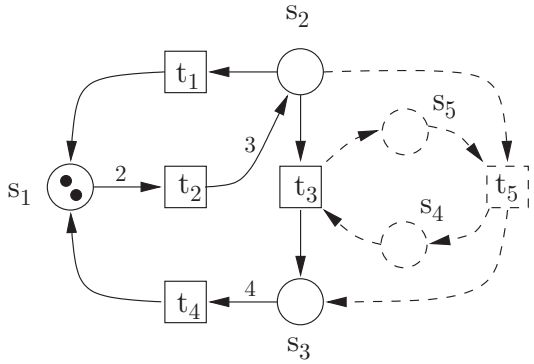


Abbildung 3.31: Netz und Teilnetz

Aufgabe 3.32 (Invarianten)

[10] In manchen Skripten und Büchern wird der Parikh-Vektor durch: $\#(w) := (\#_1(w), \#_2(w), \ldots, \#_{|T|}(w))$ dargestellt, wobei dann natürlich $\#_i(w) := |w|_{t_i}$ gilt.

Gegeben seien die S/T-Netze N und N', wobei N das Gesamtnetz aus der Abb. 3.31 ist und N' das Teilnetz ist, dessen Elemente mit durchgezogenen Linien gezeichnet sind.

Berechnen Sie mit Hilfe der Sätze 3.27 (Satz von Lautenbach) und 3.31 S-Invarianten und T-Invarianten für das Gesamtnetz N und das Teilnetz N' (falls diese existieren). Untersuchen Sie, ob es in den Netzen Schaltfolgen gibt, die die Anfangsmarkierung wiederherstellen.

3.7 Das Bankiersproblem

Die Problematik von Verklemmungen bei der Betriebsmittelvergabe wurde von Dijkstra als Problem des Bankiers dargestellt [Dij68]. Hier wurde auch der Begriff des sicheren 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 $f = (f_1, \ldots, f_n)$ und einer Zahl $g \in \mathbb{N}$. Ein Zustand einer solchen Instanz ist ein n-tupel $r = (r_1, \ldots, r_n)$ das die Restforderung der Kunden beschreibt. Anfangs gilt $r = f$. Ein Zustand heißt *sicher*, wenn er nicht notwendig zu einer Verklemmung führt.

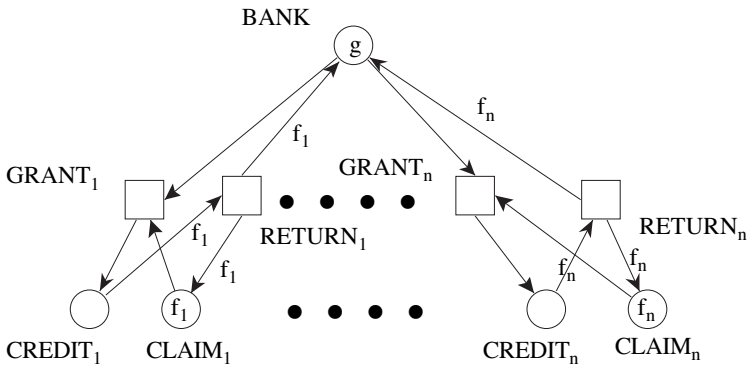


Abbildung 3.32: Das Bankiersproblem mit n Kunden

Das P/T-Netz in Abb. 3.32 modelliert das beschriebene Bankiersproblem. Der Platz *BANK* enthält so viele Marken wie das Kapital des Bankiers in Geldeinheiten umfasst. $CREDIT_i$

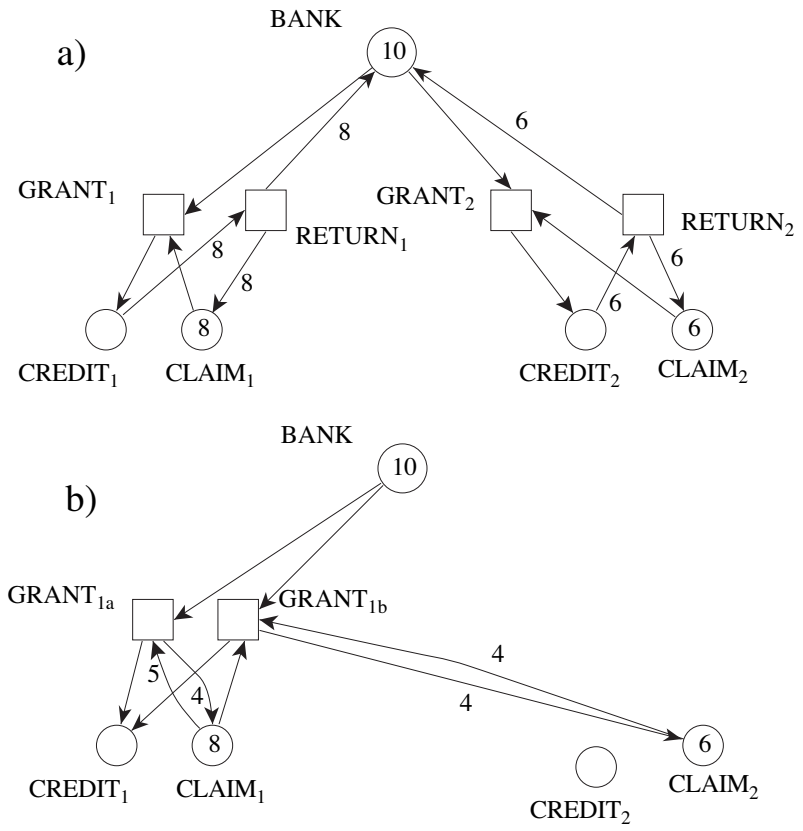


Abbildung 3.33: Eine Bankiersprobleminstanz mit 2 Kunden und Modifikation zur Vermeidung von Verklemmungen

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

Betrachten wir zwei spezielle Instanzen, nämlich $\iota_1 = (2, (8, 6), 10)$ (Abb. 3.33) und $\iota_2 = (3, (8, 3, 9), 10)$ (Abb. 3.35). An ihnen werden Erreichbarkeitsgraph und P-Invarianten erläutert.

Für die Instanz $\iota_1 = (2, (8, 6), 10)$ in Abb. 3.33a wird jeder Zustand durch eine Markierung dargestellt, d.h. durch eine 5-dimensionalen Vektor. Für alle erreichbaren Markierungen $\mathbf{m}$ gelten die folgenden 3 Gleichungen:

(Solche Gleichungen werden als P-Invarianten-Gleichungen in einem späteren Kapitel behandelt.)

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

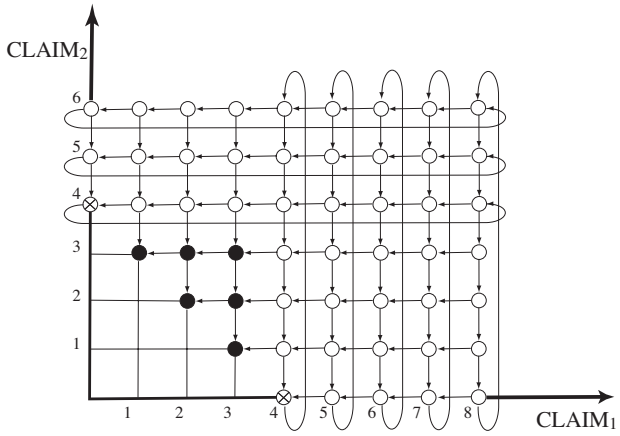


Abbildung 3.34: Erreichbarkeitsgraph des Netzes 3.33

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

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

(Sie heißen P-Invarianten-Gleichungen und werden im nächsten Abschnitt 3.6.2 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. 3.34).

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. 3.34.

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) and (4,0), (mit Kreuz).

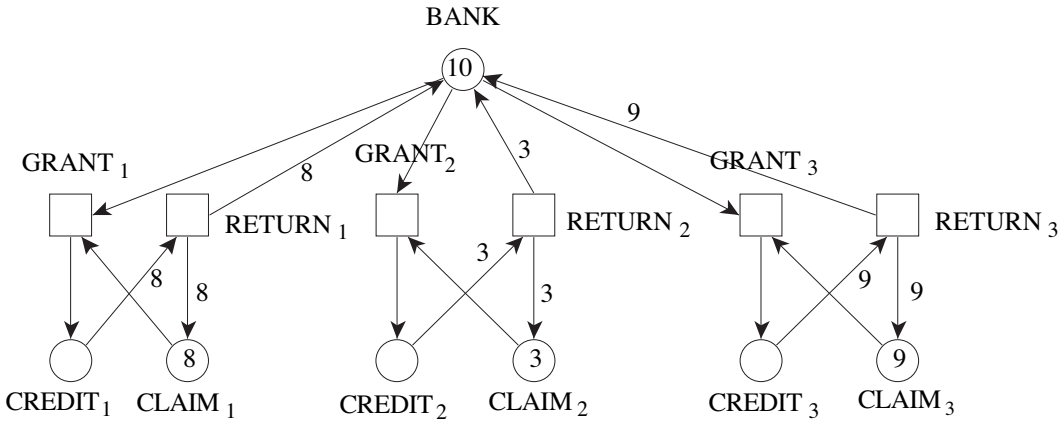


Abbildung 3.35: Eine Instanz des Bankiersproblems mit 3 Kunden

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

Wie aus Abb. 3.34 ersichtlich kann dies auch dadurch geschehen, dass Transition $GRANT_1$ nur in Markierungen aktivierbar ist, die (in mindestens einer Komponente) größer als $(4, 0)$ oder $(0, 4)$ sind. Dies wird erreicht, wenn man die Transition $GRANT_1$ durch zwei modifizierte Kopien $GRANT_{1a}$ und $GRANT_{1b}$ (Abb. 3.33b)[11] ersetzt. Diese beiden Transitionen haben den gleichen Schalteffekt wie die ursprüngliche, aber einen höheren Aktivierungs-"Schwellwert". Die entsprechende Konstruktion ist bei $GRANT_2$ anzuwenden. Der allgemeine Algorithmus ist in [VJ85] und teilweise in [JV87] S.216 ff. zu finden.

Die zweite Instanz $\iota_2 = (3, (8, 3, 9), 10)$ ist ein Beispiel aus dem Buch [BH73] über Betriebssysteme. Seine Netzdarstellung befindet sich in Abbildung 3.35. Es enthält 7 Plätze. Wegen der nun 4 Gleichungen kann der Erreichbarkeitsgraph in drei Dimensionen dargestellt werden (Abb.3.37). 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

[11] Diese 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 $RETURN_2$.

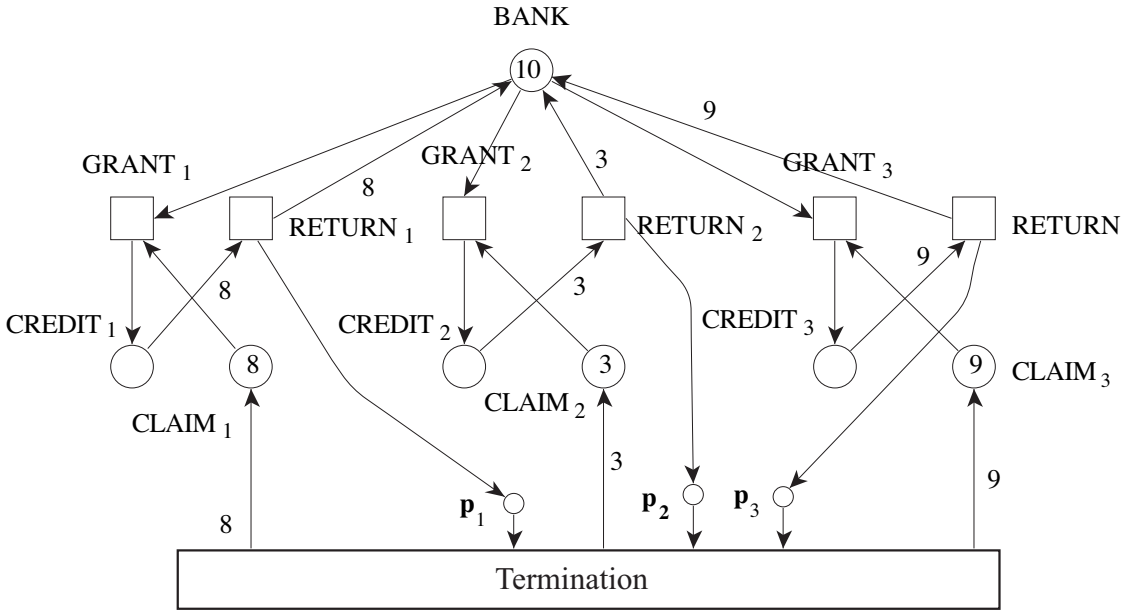


Abbildung 3.36: Bankiersnetz mit Terminationstransition

3.5 behandelt. Um die ursprüngliche Definition von Dijkstra beizubehalten, kann man wie in Abb. 3.36 eine Transition *TERMINATION* einführen, die dafür sorgt, dass alle Kunden einen Ausleihvorgang beendet haben, bevor eine neue Runde started. Um den Begriff der sicheren Markierung sinnvoll auch in dem Netz 3.35 zu benutzen, könnte man folgende Definitionen unter c) benutzen.

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

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

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

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

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

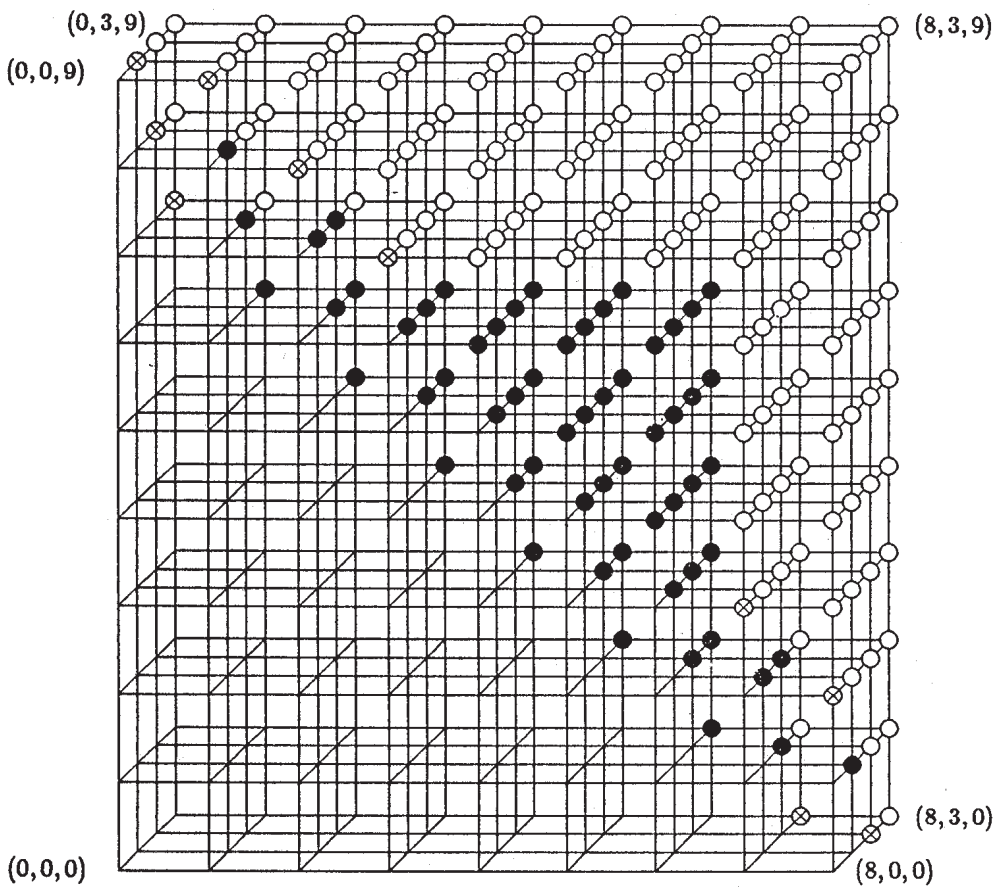


Abbildung 3.37: Erreichbarkeitsgraph des Netzes 3.35

3.8 Komplexität nebenläufiger Systeme [12]

3.8.1 Überdeckungsgraph

Der Algorithmus 3.1 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) Markenanzahlen auf einem Platz bezeichnet. Dazu werden die natürlichen Zahlen mit diesem formalen Symbol erweitert, sowie *Pseudomarkierungen* eingeführt, deren Komponenten dieses Omega enthalten.

Definition 3.33 *Es sei* $\mathbb{N}_\omega := \mathbb{N} \cup \{\omega\}$ *zusammen mit folgenden Rechenregeln:*

$$\forall n \in \mathbb{N} : \omega > n;$$

$$\forall n \in \mathbb{N}_\omega : \omega + n = \omega - n := \omega;$$

$$\forall n \in \mathbb{N}\backslash\{0\} : n \cdot \omega = \omega \cdot n := \omega; 0 \cdot \omega = \omega \cdot 0 := 0.$$

Ein Vektor[13] $\mathbf{m} \in \mathbb{N}_\omega^P$ *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* $+, -, =, \leq$ *jeweils komponentenweise erklärt, d.h.,* $\mathbf{m}_1 \leq \mathbf{m}_2$*, falls* $\forall s \in S : \mathbf{m}_1(s) \leq \mathbf{m}_2(s)$*. Lediglich* $\mathbf{m}_1 < \mathbf{m}_2$ *steht für* $(\mathbf{m}_1 \leq \mathbf{m}_2$ *und* $\mathbf{m}_1 \neq \mathbf{m}_2)$*.*

Sei $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ ein P/T-Netz. Wir konstruieren durch Algorithmus 3.4 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 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.

Abb. 3.38 zeigt ein P/T-Netz $\mathcal{N}$ mit Überdeckungsgraph.

Die Bedeutung des Überdeckungsgraphen ergibt sich aus folgenden Eigenschaften:

[12] Entnommen dem Skript *Theoretische Grundlagen der Programmierung* von M. Jantzen und diesem Skript stilistisch angepasst.

[13] Zum Begriff des Vektors siehe die Definitionen auf der Rückseite des Titelblattes.

Algorithmus 3.4 (Berechnung eines Überdeckungsgraphen)

Input - Das P/T-Netz $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$
Output - Der Überdeckungsgraph Graph $G(\mathcal{N}) = (V, E)$.

1. Initialisiere $G(\mathcal{N}) = (\{\mathbf{m_0}\}, \emptyset)$; $\mathbf{m_0}$ sei ungefärbt;
2. **while** Es gibt ungefärbte Knoten in V **do**
   2.1 Wähle einen ungefärbte 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 $X(\mathbf{m}') := \{\mathbf{m}'' \in V | \mathbf{m}'' \leq \mathbf{m}' \text{ und } \mathbf{m}'' \xrightarrow{*} \mathbf{m}'\}$;
   2.2.2 **if** $X(\mathbf{m}') \neq \emptyset$ **then** $\mathbf{m}_1(p) := \begin{cases} \omega, & \exists \mathbf{m}'' \in X(\mathbf{m}') : \mathbf{m}''(p) < \mathbf{m}'(p) \\ \mathbf{m}'(p), & \text{sonst.} \end{cases}$
   else $\mathbf{m}_1 := \mathbf{m}'$;
   2.2.3 **if** $\mathbf{m}_1 \notin V$
   then $V := V \cup \{\mathbf{m}_1\}$, wobei $\mathbf{m}_1$ ein ungefärbter Knoten sei. ;
   2.2.4 $E := E \cup \{\langle \mathbf{m}, t, \mathbf{m}_1 \rangle\}$;
3. Der Algorithmus terminiert mit Ergebnis. ($G(\mathcal{N})$ ist der Überdeckungsgraph.)

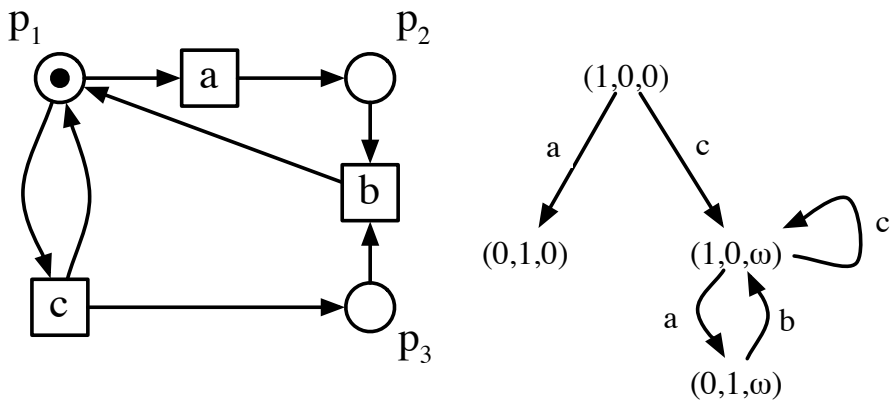


Abbildung 3.38: Ein Netz $\mathcal{N}$ mit Überdeckungsgraph

Satz 3.34 *Sei $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ 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 $\mathbf{m}_0 \xrightarrow[w]{*} \mathbf{m}_1$.

Aus dieser letzten Eigenschaft leitet sich der Name „Überdeckungsgraph“ für $G(\mathcal{N})$ ab, denn zu jedem in $\mathcal{N}$ erreichbaren Knoten $\mathbf{m} \in \mathrm{RG}(\mathcal{N})$ 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 3.35 *Der Algorithmus 3.4 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 $\mathbf{m}_0 \to \mathbf{m}_1 \to \mathbf{m}_2 \to \ldots \quad \ldots \to \mathbf{m}_i \to \mathbf{m}_{i+1} \to \ldots$ 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 $\mathbf{m}_{i_1} \xrightarrow{+} \mathbf{m}_{i_2} \xrightarrow{+} \mathbf{m}_{i_3} \xrightarrow{+} \ldots \xrightarrow{+} \mathbf{m}_{i_j} \xrightarrow{+} \mathbf{m}_{i_{j+1}} \xrightarrow{+} \ldots$ [14] 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 3.4. □

Aufgabe 3.36 Konstruieren Sie einen Überdeckungsgraphen $G(\mathcal{N})$ für das folgende P/T-Netz $\mathcal{N}$.

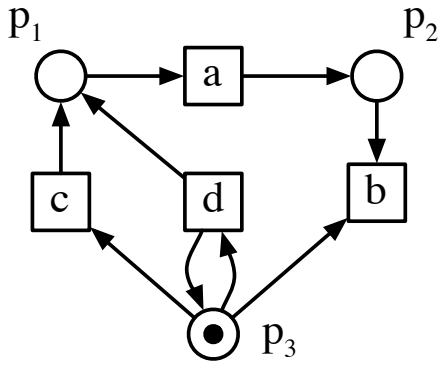


Abbildung 3.39: Netz $\mathcal{N}$ zur Aufgabe

Satz 3.37 *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.*

[14] $\xrightarrow{+}$ bedeutet wie üblich, dass ein oder mehrere Schritte der Relation vorliegen.

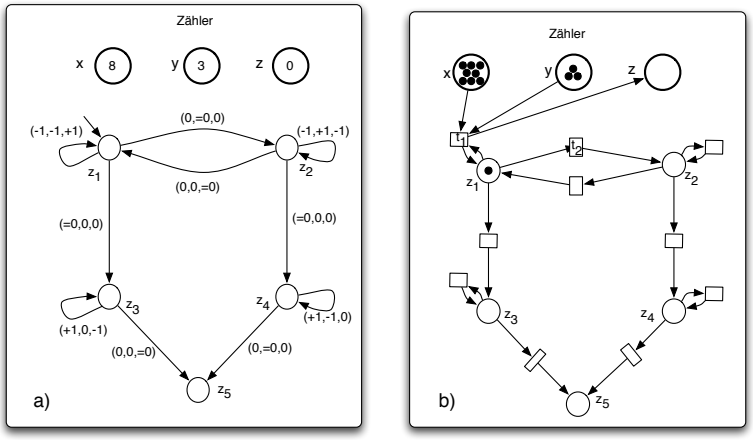


Abbildung 3.40: 3-Zählerautomat und P/T-Netz für $x := x \bmod y$

Beweis:
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.

Sei $\mathbf{m}_2 \in V$ mit $\mathbf{m}_2(s) = \omega$ für ein $s \in P$, so gibt es einen Pfad $\mathbf{m}_0 \xrightarrow[\alpha]{*} \mathbf{m}_{3,s} \xrightarrow[\beta]{*} \mathbf{m}_s \xrightarrow[t]{} \mathbf{m}_{1,s} \xrightarrow[\gamma]{*} \mathbf{m}_{2,s}$ in $G(N)$ und $\mathbf{m}_{1,s} \in \mathbb{N}_\omega^P$ mit $\mathbf{m}_s \xrightarrow{t} \mathbf{m}_{1,s}$ 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 $\mathbf{m}_3^\bullet, \mathbf{m}_1^\bullet, \mathbf{m}_2^\bullet, \ldots$ mit $\mathbf{m}_0 \xrightarrow{\alpha} \mathbf{m}_3^\bullet \xrightarrow{\beta t} \mathbf{m}_1^\bullet \xrightarrow{\gamma} \mathbf{m}_2^\bullet \ldots$.

Da $\mathbf{m}_3^\bullet(s) < \mathbf{m}_1^\bullet(s) < \mathbf{m}_2^\bullet(s) < \ldots$ gilt, ist die Platz $s \in P$ also in $\mathcal{N}$ nicht beschränkt und $\mathbf{R}(\mathcal{N})$ ist keine endliche Menge. □

3.8.2 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. Beschränktkeit, 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-Maschinen[15]. In der Abbildung 3.40 a) ist als Bei-

[15]Zählerautomaten mit nur *einem* Zähler sind mächtiger als endliche Automaten, aber weniger mächtig als Kellerautomaten, die alle kontextfreien Sprachen akzeptieren können.

spiel 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 Transitionsssystem 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 veringert.
- 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.

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 ist und danach entsprechendes im Zustand z_2 passiert. Offensichtlich wird ingesamt der Anfangswert von x so oft wie möglich um den Anfangswert von y veringert und der verbleibende Rest nach x gebracht, wonach der Automat in den Endzustand z_5 übergeht. Dadurch wird die Zuweisung $x := x$ `mod` 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 3.40 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 enthalten[16]. 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 3.40 b) das Verhalten der entsprechenden Transition $z_1 \xrightarrow{(0,=0,0)} z_2$ in a) zu geben, müsste geprüft werden, ob der Platz y keine Marke enthält. Ist ein solcher Test bei P/T-Netzen möglich?

Die Abbildung 3.41 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

[16] Sie erfüllen die S-Invariante $z_1 + z_2 + z_3 + z_4 + z_5 = 1$.

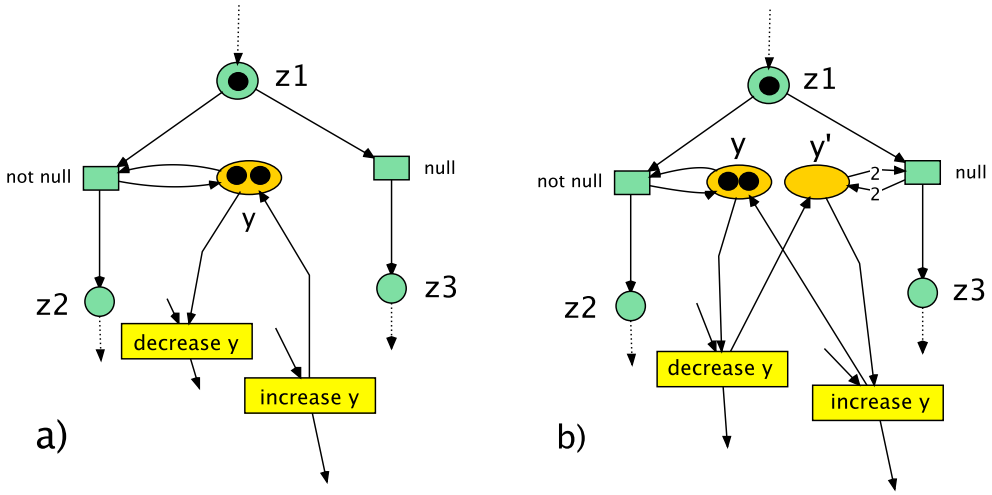


Abbildung 3.41: Nulltest-Situation für P/T-Netze in a) und Lösung in b)

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 3.41 b). Dies ist ein Platz der die S-Invarianten-Gleichung $y+y' = k$ erfüllt, wobei k wie immer aus der Anfangsmarkierung zu berechnen ist (hier $k = 2$). Folglich kann die Tansition `null` genau dann schalten, wenn $\mathbf{m}(y') \geq 2$, d.h. - wie gewünscht - $\mathbf{m}(y) = 0$ ist. Diese Konstruktion ist wegen der S-Invarianten-Gleichung jedoch nur möglich, wenn der Platz y beschränkt ist, eine Forderung, die bei der Simulation von Turing-mächtigen Zählerautomaten nicht zu erfüllen ist[17]. 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 3.42 a) eingeführt. Hier kann *per definitionem* die Transition 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 3.42 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 3.42 c) gegeben ist. Eine solche Lösung setzt die Existenz einer nicht endlichen Farbmenge voraus, wie hier die Farbe `integer` für den Platz y. Wie in Kapitel 2 diskutiert, sind gefärbte Netze mit nur endlichen Farbmengen gleichmächtig zu P/T-Netzen, da sie immer zu solchen aufgefaltet werden können. Eine Übersicht über die Turing-Mächtigkeit bei Petrinetzen gibt die Tabelle 3.2.

[17] Wenn 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.

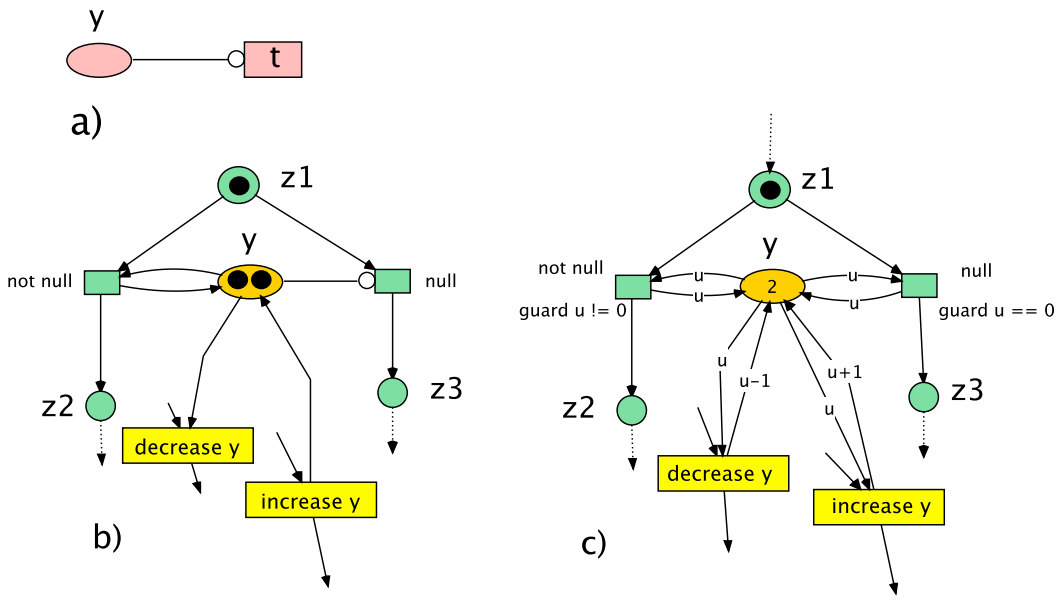


Abbildung 3.42: Nulltest in Inhibitor- und gefärbten Netzen

	Turing-mächtig?
P/T-Netze	nein
Inhibitor-Netze	ja
gefärbte Netze mit endlichen Farbmengen	nein
gefärbte Netze mit beliebigen Farbmengen	ja

Tabelle 3.2: Turing-Mächtigkeit von Petrinetzen

3.8.3 Komplexität

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 3.38 *Es gibt eine unendliche Folge von beschränkten Petri-Netzen $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3, \ldots$, deren Größe ($|P|+|T|+|F|$) linear wächst, jedoch die Größe der Erreichbarkeitsmengen $|\mathbf{R}(\mathcal{N}_1)|, |\mathbf{R}(\mathcal{N}_2)|, |\mathbf{R}(\mathcal{N}_3)|, \ldots$ wächst schneller als jede primitiv rekursive Funktion.*

Eine ebenfalls nicht primitiv rekursiv berechenbare Variante der *Ackermann-Funktion* ist folgende:

$$A(0,n) := 2n+1, A(m+1,0) := 1, A(m+1,n+1) := A(m, A(m+1,n))$$

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 3.4) für die Endlichkeit der Erreichbarkeitsmenge eines P/T-Netzes mit dem Überdeckungsgraphen nicht primitiv-rekursiv ist! Daher entsteht die Frage, ob es vielleicht bessere Verfahren gibt, mit denen dieses Problem entschieden werden kann?

Folgende Ergebnisse sind bekannt:

Satz 3.39 (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 3.40 (Lipton (1976)) *Das Beschränktheitsproblem benötigt für seine Entscheidung mindestens $O(2^{c \cdot \sqrt{n}})$ Platzbedarf.*

Ein besseres Ergebnis ist von Rosier und Yen bewiesen worden:

Satz 3.41 (Rosier&Yen (1986)) *Das Beschränktheitsproblem kann für P/T-Netze $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ mit $O\big(2^{c \cdot |P| \cdot log(|P|)} \cdot (log(|T|) + max_{x,y \in P \cup T}(|W(x,y)|)\big)$ 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 3.42 *Ein P/T-Netz* $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ *heißt* konfliktfrei*, falls es keinen* Konfliktplatz *enthält, d.h. einen Platz* p *mit* $|p^\bullet| > 1$*.*

Satz 3.43 (Rosier et. al. (1987)) *Das Beschränktheitsproblem kann für konfliktfreie Petrinetze mit* $O(n^{1,5})$ *Platzbedarf entschieden werden.*

Dieses Beispiel zeigt, dass eine alleinige Beschränkung der Kapazität der Plätzen in der Regel nicht ausreicht, um niedrige Komplexitätseigenschaften zu erreichen. 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.

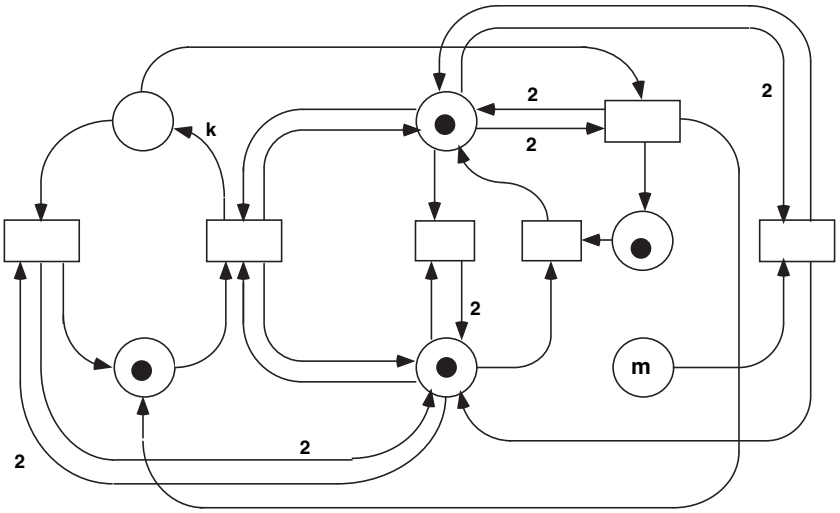
Das P/T-Netz $\mathcal{N}_7$ hat 6 Plätze, 6 Transitionen, $30 + k$ Kanten und $m + 4$ Marken in der eingezeichneten Anfangsmarkierung $\mathbf{m}_0$. Die Größen $k \geq 2$ und $m \geq 0$ sind festlegbar und die maximale Zahl von möglichen Marken bestimmt sich durch die Funktion

$$\max(m, k) := k \cdot f_k(m) + 2,$$

wobei f_k wie folgt definiert ist:

$$f_k(m) := \texttt{ IF } m = 0 \texttt{ THEN } k \texttt{ ELSE } f_k(m-1) \cdot k^{f_k(m-1)} \texttt{ FI}.$$

Es gilt: $\max(2, 2) = 4098$, aber schon $\max(3, 2) = 1048576^{103} + 2$ und $\max(2, 3) = 3^{86} + 2$.



Auch für syntaktisch stark eingeschränkte Netze, wie *markierte Graphen* oder *1-konservative Netze* sind hohe Analysekomplexitäten bekannt.

Definition 3.44 *Ein P/T-Netz* (S, T, W) *heißt* konservativ*, wenn es keine multiplen Kanten besitzt (d.h. stets* $W(x, y) \leq 1$ *gilt) und eine Funktion* $f : S \to \mathbb{N} \setminus \{0\}$ *existiert, für die gilt:* $\forall t \in T : \sum_{s \in {}^\bullet t} f(s) = \sum_{s \in t^\bullet} f(s)$*.*

Netze, bei denen $f(s) = 1$ für alle $s \in S$ ist, heißen 1-konservativ *und es gilt dann automatisch:* $\forall t \in T : |t^\bullet| = |^\bullet t|$.

Ein P/T-Netz (S, T, W) heißt markierter Graph*, wenn gilt:*

$$\forall s \in S : |^\bullet s| = |s^\bullet| = 1.$$

Satz 3.45 *Sei $\mathcal{N} = \langle P, T, F, W, \mathbf{m}_0 \rangle$ konservatives P/T-Netz, dann gilt:*
aus $f(\mathbf{m}) := \sum_{s \in S} \mathbf{m}(s) \cdot f(s)$ folgt für jede erreichbare Markierung $\mathbf{m} \in \mathbf{R}(\mathcal{N})$ stets $f(\mathbf{m}_0) = f(\mathbf{m})$.

Aufgabe 3.46 Beweise den oben angeführten Satz 3.45 und schließe daraus auf die Endlichkeit der Menge $\mathbf{R}(\mathcal{N})$.

Besteht das Problem für nur *eine* Markierung zu entscheiden, ob sie im Erreichbarkeitsgraphen vorkommt, so bezeichnet man dies als *Erreichbarkeitsproblem*.

Definition 3.47 (Erreichbarkeitsproblem)
Gegeben: *Ein P/T-Netz $\mathcal{N} := (P, T, F, W, \mathbf{m}_0)$ und eine Markierung $\mathbf{m} \in \mathbb{N}^P$.*

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

Definition 3.48 (Überdeckbarkeitsproblem)
Gegeben: *Ein P/T-Netz $\mathcal{N} := (P, T, F, W, \mathbf{m}_0)$ und eine Markierung $\mathbf{m} \in \mathbb{N}^P$.*

Frage: *Gibt es $\mathbf{m}' \in \mathbf{R}(\mathcal{N})$ mit $\mathbf{m}' \geq \mathbf{m}$?*

Das Erreichbarkeitsproblem ist entscheidbar [May84], aber selbst für 1-konservative P/T-Netze komplex.

Satz 3.49 *Das Erreichbarkeitsproblem für 1-konservative P/T-Netze ist $\mathcal{PSPACE}$-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. Die in 3.44 definierten *markierten Graphen* gehören zu letzterer Klasse.

Die bekannten Algorithmen für das Erreichbarkeitsproblem für beliebige P/T-Netze konstruieren eine abgewandelte Form des Überdeckungsgraphen 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.

Satz 3.50 *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. Einen guten Überblick über weitere Komplexitätsresultate zu Algorithmen und Problemen bei Petrinetzen ist bei Esparza und Nielsen (1994) zu finden.

Kapitel 4

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 Vorlesungen[1] behandelden 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. 4.1). Bei Speichersynchronisation wird eine SIMD-Architektur (*single instruction multiple data*) (Abb. 4.1) 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 4.1 *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,*

[1] Bachelor/Master Modul: Modellierung verteilter Systeme (MVS)

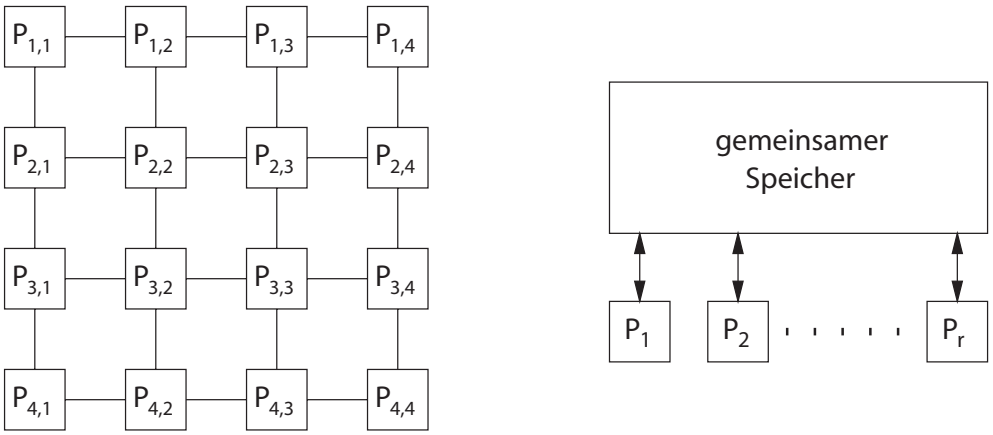


Abbildung 4.1: Prozessorkonfigurationen: 2-dimensionales Feld und PRAM

- 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 Defionition 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* Algorithmen für $\mathcal{P}$ genommen, wenn Letzeres nicht bekannt ist). Ein paralleler Algorithmus A heißt *optimal* für ein Problem $\mathcal{P}$, wenn $\mathrm{W}_A(n) :\in \Theta(T^*_{\mathcal{P}}(n))$ (äquivalente Notation: $\mathrm{W}_A(n) = \Theta(T^*_{\mathcal{P}}(n))$. 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.

4.1 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.

[2]Entnommen dem Skript *Automaten und Komplexität (AUK)* von M. Jantzen und diesem Skript angepasst.

Die *Random-Access-Maschine (RAM)* ist ein einfaches von Neumann–Modell sequentieller Rechner, das gleichmächtig 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.

4.1.1 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 $x_1, x_2, \ldots$

Der Speicher einer RAM besteht aus einer abzählbaren Folge von Registern $R_0, R_1, \ldots$. Der Index i eines Registers dient als **Adresse**. Register R_0 dient als **Akkumulator**.

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 $b_0, b_1, \ldots$ von Befehlen aus einer vorgegebenen Befehlsmenge $\mathcal{B}$.

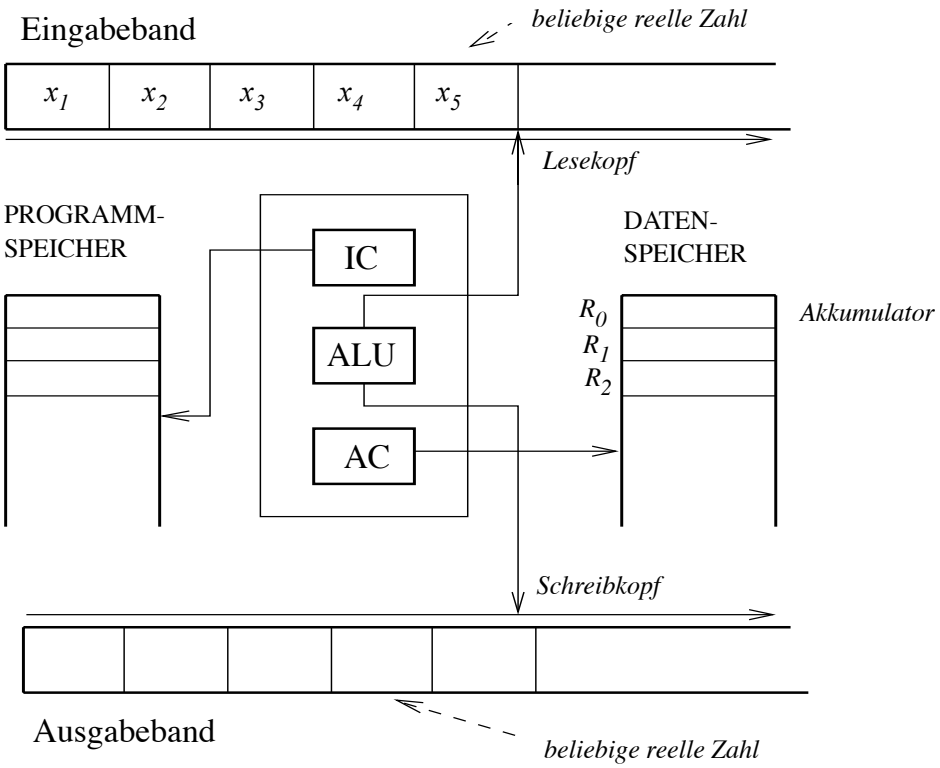


Abbildung 4.2: RAM – Random Access Machine

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	label – Befehlsadresse
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 4.1 zeigt, wie sich die Konfiguration der Maschine durch Ausführung eines Befehls ändert.

Die Befehle des Programms werden in sequentieller Ordnung ausgeführt, bis eine Halte- oder

Tabelle 4.1: 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 *location counter* (Befehlsregister) wird auf die mit b markierte Anweisung gesetzt.
10.	JGTZ b	Der *location counter* wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) > 0$ ist, sonst auf die nachfolgende Anweisung.
11.	JZERO b	Der *location counter* wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) = 0$ ist, sonst auf die nachfolgende Anweisung.
12.	HALT	Programmausführung beenden.

c – Speicherabbildung

$c(i)$ Der Inhalt des Registers R_i.

$v(a)$ – Der Wert von a

$$\begin{aligned} v(=i) &= i \\ v(i) &= c(i) \\ v(*i) &= c(c(i)) \end{aligned}$$

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 $\mathcal{P}$ vom Eingabeband stets n Zahlen $x_1, \ldots, x_n$ liest, danach berechnet und danach stets m Zahlen $y_1, \ldots, y_m$ auf das Ausgabeband schreibt, dann sagt man, dass $\mathcal{P}$ 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 4.2 In Abb. 4.3 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. 4.4 zu sehen. Aus Abb. 4.3 kann man auch ersehen, wie man die Befehle der höheren Programmiersprache in RAM–Instruktionen übersetzen kann.

2. *Akzeptieren von Sprachen:*
   Ein RAM–Programm kann man auch als einen Akzeptor einer Sprache interpretieren.

```
            READ    1          read r1
            LOAD    1
            JGTZ    pos        } if r1 ≤ 0 then write 0
            WRITE   =0
            JUMP    endif
pos:        LOAD    1          } r2 ← r1
            STORE   2
            LOAD    1
            SUB     =1         } r3 ← r1 – 1
            STORE   3
while:      LOAD    3
            JGTZ    continue   } while r3 > 0 do
            JUMP    endwhile
continue:   LOAD    2
            MULT    1          } r2 ← r2 * r1
            STORE   2
            LOAD    3
            SUB     =1         } r3 ← r3 – 1
            STORE   3
            JUMP    while
endwhile:   WRITE   2          write r2
endif:      HALT
```

Abbildung 4.3: $f(n) = n^n$ (RAM–Programm)

Sei $\Sigma = \{a_1, \ldots, a_m\}$ 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 $\mathcal{P}$ akzeptiert eine Sprache $\mathcal{L} \subsetneq \pm^*$ folgenderweise: Das Eingabewort $w = w_1 \ldots w_k$ 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.

Dies Eingabewort wird durch ein Programm $\mathcal{P}$ akzeptiert, wenn $\mathcal{P}$ zum Schluss der Berechnung 1 in das erste Feld des Ausgabebandes schreibt und hält. Die Sprache, die $\mathcal{P}$ akzeptiert, ist die Menge der Wörter, die $\mathcal{P}$ akzeptiert.

Es ist nicht schwer zu zeigen, dass RAM–Programme für Bit-RAM's genau die rekursiv aufzählbaren Sprachen akzeptieren.

Beispiel 4.3 In Abbildung 4.6 und 4.5 ist ein Hochsprache–Programm und ein RAM-Programm abbgebildet, um die Sprache $\mathcal{L}_0 \subsetneq \{1, 2\}^*$ zu akzeptieren, deren Wörter dieselbe Anzahl von Einsen und Zweien besitzt.

```
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 4.4: $f(n) = n^n$ (Hochsprache)

4.1.2 Komplexitätsmaße für die RAM

Die parallelen Algorithmen am Ende dieses Kapitels werden in einem Pseudokode formuliert, für den die in Definition 4.1 angebenen Definitionen ausreichend sind. Das Modell der RAM ist jedoch genauer und hat präziser formulierte Definitionen für Komplexitätsmaße. Beispielsweise wird der in Definition 4.1 benutzte Begriff „Größe“ genauer gefasst.

Es gibt zwei Arten von Komplexitätsmaßen für RAMs.

Uniforme Komplexitätsmaße:

Uniformes Zeitmaß:	Uniformes Platzmaß:
1 Schritt = 1 Zeiteinheit	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 Funktion $l : \mathbb{N} \longrightarrow \mathbb{N}$, 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 4.2 (Seite 129) 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)))

```
            LOAD    =0      }
            STORE   2       }   d ← 0
            READ    1           read x
while:      LOAD    1       }
            JZERO   endwhile}   while x ≠ 0 do
            LOAD    1       }
            SUB     =1      }   if x ≠ 1
            JZERO   one     }
            LOAD    2       }
            SUB     =1      }   then d ← d – 1
            STORE   2       }
            JUMP    endif
one:        LOAD    2       }
            ADD     =1      }   else d ← d + 1
            STORE   2       }
endif:      READ    1
            JUMP    while
endwhile:   LOAD    2       }
            JZERO   output  }
            HALT            }   if d = 0 then write 1
output:     WRITE   =1      }
            HALT            }
```

Abbildung 4.5: Erkennung von $\mathcal{L}_0 \subsetneq \{1,2\}^*$ (RAM)

Definition 4.4 *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.*

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).*

Anmerkung zur vorangegangenen Definition: Die Größe $\sum_{j=1}^{s} \text{bpos}(i_{j-1}, i_j)$ muss berücksichtig werden, da sonst bei sehr trickreicher Programmierung mit Pointern ein nicht zu rechtfertigender Komplexitätsvorteil auftritt, falls sehr weit entfernte Register benutzt werden und deren Registeradressen zur Speicherung von Werten „mißbraucht“ werden. Bei normaler Programmierung tritt dieser Effekt nicht auf!

Beispiel 4.5 Betrachten wir das RAM–Programm A, das $f(n) = n^n$ berechnet (siehe Abb. 4.4 und 4.3). Es ist offensichtlich, dass für die uniformen Komplexitätsmaße die Zeitkomplexität

```
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 4.6: Erkennung von $\mathcal{L}_0 \subsetneq \{1,2\}^*$ (Hochsprache)

Tabelle 4.2: 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(input) + l(i)
	READ $*i$	l(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

$T_A(n) = \mathrm{O}(n)$ und die Platzkomplexität $T_A(n) = \mathrm{O}(1)$ 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 4.6 Die folgende Tabelle 4.3 zeigt die Komplexitätsmaße für das Programm, das die Sprache $\mathcal{L}_0$ akzeptiert.

Tabelle 4.3: 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.

4.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine

RAM und TM sind zwei sehr verschiedene Modelle. Wie schnell können sie einander simulieren ?

Satz 4.7 *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.*

Der Beweis stammt aus [HMU79]. Einen etwas ausführlicheren finden die Leser(innen) bei [Pau78]. Sehr detaillierte Beweise finden sich in [Rei90]. *Beweis:*

M habe k einseitig unendliche Bänder. Die simulierende RAM_+ (in der Abbildung 4.7 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.

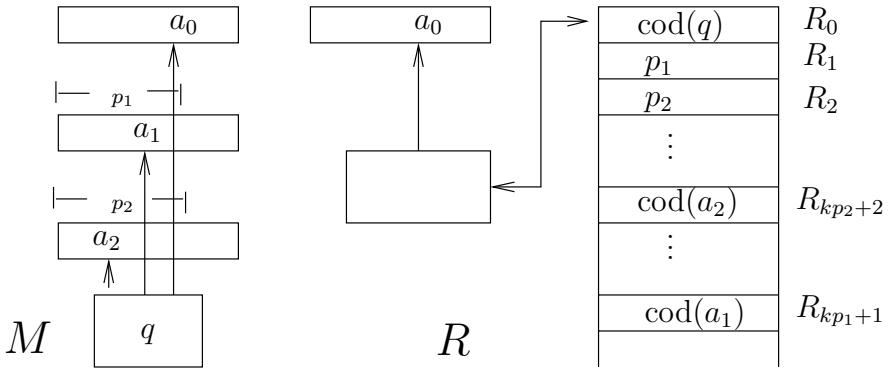


Abbildung 4.7: Zur Simulation einer TM durch eine RAM

Für jedes Tupel

$$(q, a_0, a_1, \ldots, a_k)$$

aus einem Zustand q, einem Eingabesymbol a_0 und von den Arbeitsköpfen gelesenen Zeichen a_i hat R ein kleines Programmstück, das den übergang

$$\delta(q, a_0, a_1, \ldots, a_k) = (q', a'_1, \ldots, a'_k, q_1, \ldots, q_k, a'_{k+1})$$

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. □

Satz 4.8 *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	#	#	$\cdots$	i_k	#	c_k	#	#	b	$\cdots$

wobei $i_1, i_2, \ldots, i_k$ 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, auer 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 polynomically 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 $\mathcal{M}$-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 Model 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.

4.2 Parallele Random-Access-Maschine (PRAM) [3]

Es gibt zwei grundlegende Modelle paralleler Computer mit globalem Speicher:

MIMD-Computer (*M*ultiple *I*nstructions *M*ultiple *D*ata) – jeder Prozessor führt ein spezielles (eigenes) Programm aus

SIMD-Computer (*S*ingle *I*nstructions *M*ultiple *D*ata) – 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.

4.2.1 Definition der PRAM

Definition 4.9 (PRAM) *Eine* parallele Registermaschine *(PRAM) ist eine (endlose) Folge von Prozessoren $P_1, P_2, \ldots$. 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 $R_{i,0}, R_{i,1}, \ldots$. Auf R_i kann ausschließlich P_i zugreifen. Die Kommunikation zwischen den Prozessoren geschieht über einen* globalen Speicher *$\mathcal{M} : M_0, M_1, \ldots$, ebenfalls eine unendliche Folge von Registern.*

[3] Entnommen dem Skript *Automaten und Komplexität (AUK)* von M. Jantzen und diesem Skript angepasst.

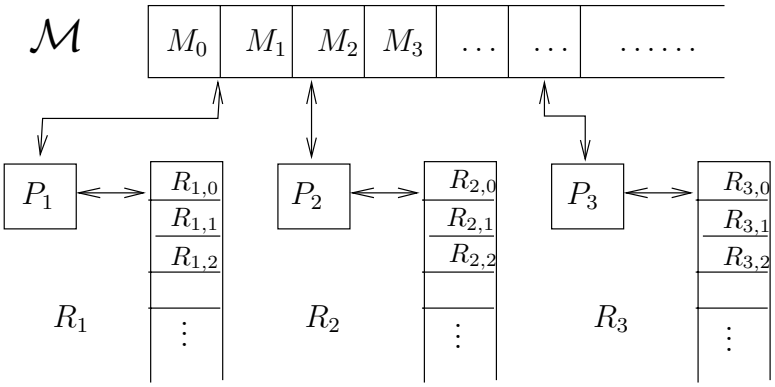


Abbildung 4.8: Parallele RAM - PRAM

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 $X = x_1, \ldots, x_n$ steht in den Registern $M_1, \ldots, M_n$ 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 $M_1, \ldots, M_m$ geschrieben.*

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.

4.2.2 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 *READ*s vor den *WRITE*s 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 (**E**xclusive **R**ead, **E**xclusive **W**rite):
Simultane *READ*s und *WRITE*s sind nicht erlaubt.

CRCW PRAM (**C**oncurrent **R**ead, **C**oncurrent **W**rite):
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$^{\text{com}}$ PRAM (**com**mon):
Ein gleichzeitiges Schreiben in ein Register M_j ist nur zulässig, wenn alle beteiligten Prozessoren versuchen denselben Wert zu schreiben.

CRCW$^{\text{arb}}$ PRAM (**arb**itrary):
Wenn einige Prozessoren versuchen, in ein Register zu schreiben, ist einer erfolgreich. Es ist aber nicht a priori festgelegt, welcher.

CRCW$^{\text{pri}}$ PRAM (**pri**ority):
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 (**C**oncurrent **R**ead, **E**xclusive **W**rite):
Gleichzeitiges Lesen erlaubt, jedoch nur exklusives Schreiben.

4.2.3 Komplexitätsmaße der PRAM

Definition 4.10 *Die uniforme (logarithmische)* Zeitkomplexität $time_M(x)$ *einer PRAM M auf eine Eingbe x ist entsprechend wie für die RAM in Definition 4.4 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 4.11 *Die* Prozessorkomplexität $proc_M(x)$ *einer PRAM M auf eine Eingabe x sei erklärt durch $proc_M(x) = max\{i \mid i = 1$ oder P_i führt auf x ein $WRITE$ 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 4.12 *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 ($O(1)$).

Überaschenderweise 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 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ 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

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$

Beweis:

$$\text{Mit } \lceil \frac{x_i}{p} \rceil \leq \frac{x_i}{p} + 1 \text{ gilt } \sum_1^t \lceil \frac{x_i}{p} \rceil = \lfloor \sum_1^t \lceil \frac{x_i}{p} \rceil \rfloor \leq \lfloor \sum_1^t (\frac{x_i}{p} + 1) \rfloor = t + \lfloor \frac{x}{p} \rfloor$$

□

Korollar 4.14 *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: *$O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$).*

(Man wähle $x = O(n),\ t = O(\log n),\ p = O\left(\frac{n}{\log n}\right)$)

Wenn die Zuordung von Daten zu Prozessoren kein Problem ist, dann kann man dieses Korollar auf PRAMs anwenden.

4.2.4 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 4.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.

Algorithmus (für EREW-PRAM):

$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{down to}}\ 0\ \underline{\text{do}}$
  $\underline{\text{par}}\ [2^l \leq j \leq 2^{l+1}-1]\ a_j \leftarrow max\{a_{2j}, a_{2j+1}\}$

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 4.9 darstellen.

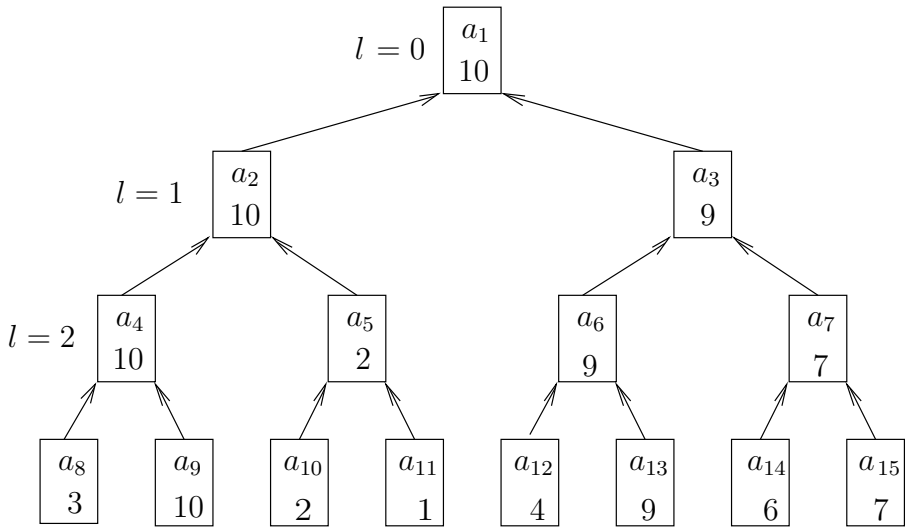


Abbildung 4.9: Maximum finden durch die PRAM

Nun genügt es, Brent's principle zu benutzen, um einen optimalen Algorithmus zu erhalten.

Beispiel 4.16 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$^{\text{arb}}$, CRCW$^{\text{com}}$, oder CRCW$^{\text{pri}}$–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: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – 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 4.17 Berechnung der ODER-Funktion auf EREW-PRAM Der folgende Algorithmus berechnet die Funktion

$$\text{ODER: } \{0,1\}^n \to \{0,1\},\ OR(x_1, \ldots, x_n) = \text{ if any } x_i = 1 \text{ then } 1 \text{ else } 0.$$

Input: $M(1), \ldots, M(n)$ – Register des globalen Speichers
Output: $M(1)$
Prozessoren: $P_1, \ldots, P_n$
Arbeitsspeicher: Y_i – jeweils lokales Register von P_i

Algorithmus für den Prozessor P_i:

```
begin t ← 0;
   Y_i ← M(i);
   while i + 2^t ≤ n
      do Y_i ← Y_i ∨ M(i + 2^t)
         M(i) ← Y_i;
         t ← t + 1
      endwhile
end
```

Analyse:
Zeitkomplexität: $\lceil \log n \rceil + 1$, Prozessorkomplexität: n

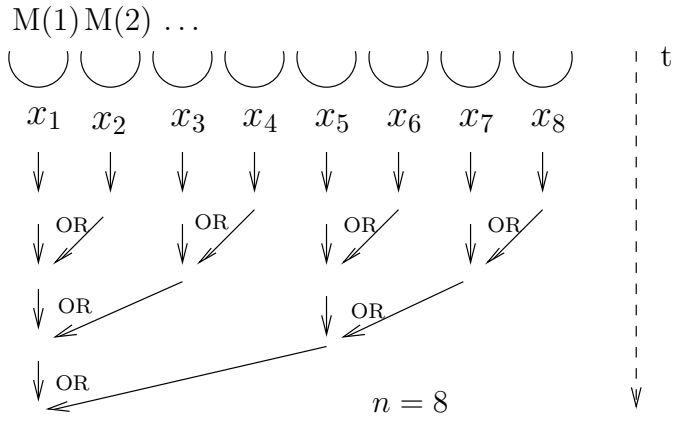


Abbildung 4.10: ODER-Berechnung durch die PRAM

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 4.18 Präfixsumme Gegeben seien $n = 2^m$ Zahlen $a_n, a_{n+1}, \ldots a_{2n-1}$. Zu berechnen sind alle partiellen Summen $b_{n+j} = \sum_{k=n}^{n+j} a_k = a_n + \ldots + a_{n+j}$,mit $j = 0, \ldots, n-1$. 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:

$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{downto}}\ 0\ \underline{\text{do}}$
  $\underline{\text{par}}\ [2^l \leq j < 2^{l+1}]\ a_j \leftarrow a_{2j} + a_{2j+1};$
$b_1 \leftarrow a_1;$
$\underline{\text{for}}\ l \leftarrow 1\ \underline{\text{to}}\ m\ \underline{\text{do}}$
  $\underline{\text{par}}\ [2^l \leq j < 2^{l+1}]$
    $b_j \leftarrow\ \underline{\text{if}}\ j \text{ odd } \underline{\text{then}}\ b_{(j-1)/2}\ \underline{\text{else}}\ b_{j/2} - a_{j+1}$

Dieser Algorithmus braucht $O(\log n)$ Zeit und $O(n)$ Prozessoren. Mit Hilfe von Brent's Prinzip läßt sich ein optimaler Algorithmus finden.

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.

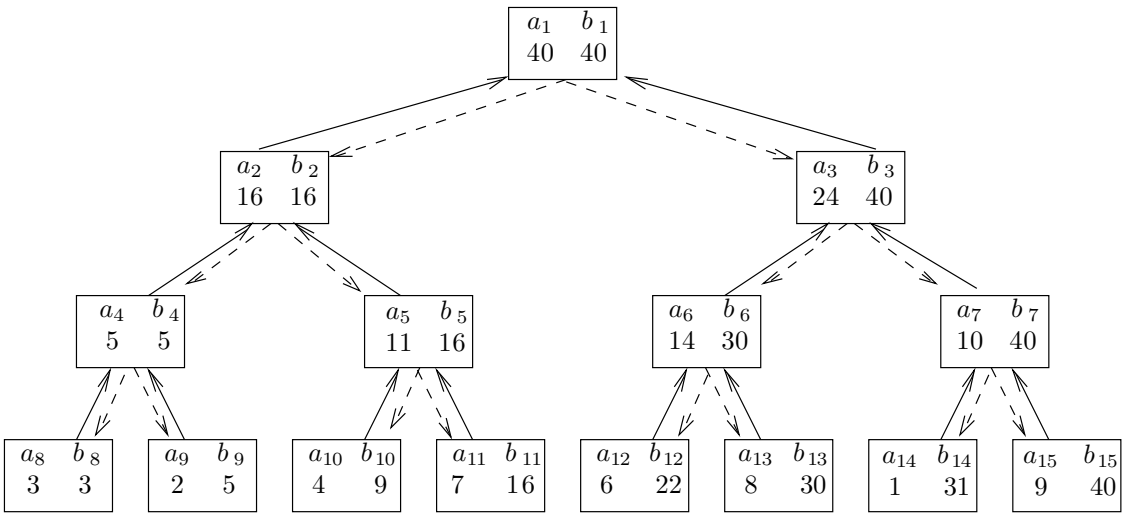


Abbildung 4.11: Präfixsummenberechnung durch die PRAM

4.2.5 PRAM–Hauptkomplexitätsklassen

Sei $\mathcal{M}$ eine beliebige PRAM–Variante. Wir bezeichnen mit

$$\mathcal{M}TimeProc(T, P)$$

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 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{M}TimeProc(T, P) \subsetneq \mathcal{M}TimeProc(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

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) $(i-1)k(n)+1, \ldots, ik(n)$ 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 besa, 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. □

Als ein Korollar bekommen wir:

Korollar 4.20
$CRCW_+\ TimeProc(\text{POL},\ \text{POL}) = \mathcal{P}$ *(Polynomialzeit)*

Beweis:
Für Polynome P, T und $k = P$ ergibt sich aus dem vorigen Satz:

$$\begin{aligned} CRCW_+\ TimeProc(T, P) &\subseteq CRCW_+\ TimeProc(O(P \cdot T), 1) \\ &= RAM_+ - Time(O(T \cdot P)) \subseteq \mathcal{P} \end{aligned}$$

□

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!

Ohne Beschränkung der Prozessoranzahl gilt aber:

Satz 4.21
$PRAM - Time(POL) = \mathcal{PSPACE}$

(Man beachte, dass eine PRAM in Zeit t bis zu $E(t)$ Prozessoren benutzen kann, wobei $E(t)$ eine exponentielle Funktion von t ist!)

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 sequencial 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.

4.2.6 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 4.22 (Kung) *Kein paralleler Algorithmus, der die Operationen $+, -, *, \div$ benutzt, kann ein Polynom n-ten Grades schneller als in $\log n$ Schritten berechnen.*

Korollar 4.23 *Kein paralleler Algorithmus kann x^n schneller als ein sequentieller Algorithmus berechnen.*

4.3 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 4.4 schrittweise verbessert, um dann als Prozedur für das parallele Sortieren in Algorithmus 4.4 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.

	Zeit $T_A(n)$	Operationen $W_A(n)$	optimal ?
Mischen 1: Satz 4.27	$O(log\ n)$	$O(n)$	ja
Mischen 2: Korollar 4.31	$O(log\ log\ n)$	$O(n\ log\ log\ n)$	nein
Mischen 3: Satz 4.32	$O(log\ log\ n)$	$O(n)$	ja
Sortieren: Satz 4.33	$O(log\ n\ log\ log\ n)$	$O(n\ log\ n)$	ja

Tabelle 4.4: Mischen und Sortieren

Definition 4.24 *Gegeben sei eine lineare Ordnung $(S, \leq)$ (Def. 2.2) und zwei Teilmengen $\tilde{A}, \tilde{B} \subseteq S$ mit $|\tilde{A}| = |\tilde{B}| = n > 0$, zu denen wir die Folgen: $A = (a_1, a_2, \ldots, a_n)$ und $B = (b_1, b_2, \ldots, b_n)$, mit $i < j \Rightarrow a_i \leq a_j \quad \wedge \quad b_i \leq b_j \quad (1 \leq j, j \leq n)$ benutzen. Die* Mischung *(merge) von A und B ist die Folge $C = (c_1, c_2, \ldots, c_{2n})$ mit $i < j \rightarrow c_i \leq c_j$, die A und B als disjunkte Teilfolgen enthält.*

Beispiel: $(2,4,4) \quad (3,4,7)$

$\searrow \quad \swarrow$

$(2,3,4,4,4,7)$

Definition 4.25 *Sei $X = (x_1, x_2, \ldots, x_t)$ eine Folge mit Elementen aus der linear geordneten Menge S. Für $x \in S$ ist der* Rang *von x in X definiert als:*
$rank(x : X) := |\{i | i \in \{1, \ldots, t\} \wedge x_i < x\}|$. Für eine weitere solche Folge
$Y = (y_1, \ldots, y_s)$ sei $rank(Y : X) := (r_1, r_2, \ldots, r_s)$ mit $r_i = rank(y_i : X)$

Beispiel: Für $X = (25, -13, 26, 31, 54, 7)$ und $Y = (13, 27, -27)$ ist $rank(Y : X) = (2, 4, 0)$.

Zur Vereinfachung nehmen wir jetzt $A \cap B = \emptyset$ an. Das Problem, die Folgen A und B zu mischen, wird dann zu:

$$\text{Bestimme für jedes } x \in A \cup B : i := rank(x : A \cup B)$$
$$\text{(dann ist } x \text{ das } (i+1)\text{-te Element } c_i \text{ in C)}$$

Wegen $rank(x : A \cup B) = rank(x : A) + rank(x : B)$ genügt uns ein Algorithmus für:

$$\text{Berechne } rank(B : A)$$

Für $b_i \in B$ kann durch binäres Suchen j_i mit $a_{j_i} < b_i < a_{j_i+1}$ in $O(\log n)$ (sequentieller) Zeit berechnet werden, d.h. $j_i = rank(b_i : A)$. Dieses Verfahren kann man parallel auf alle Elemente von B anwenden, d.h. wir erhalten einen parallelen Algorithmus mit $O(\log n)$ (paralleler) Zeit und $O(n \log n)$ Operationen.

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:

Algorithmus 4.1 (Partitionieren)

Eingabe: Zwei Felder $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_m)$ in aufsteigender Reihenfolge sortiert, wobei $\log m$ und $k(m) = \frac{m}{\log m}$ ganze Zahlen sein müssen.
Ausgabe: $k(m)$ Paare (A_i, B_i) von Teilfolgen 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** $1 \leq i \leq k(m) - 1$ **pardo**
   2.1 Berechne $rank(b_{i \log m} : A)$ durch binäre Suche und setze $j(i) = rank(b_{i \log m} : A)$
3. **for** $0 \leq i \leq k(m) - 1$ **pardo**
   3.1 $B_i := (b_{i \log m + 1}, \ldots, b_{(i+1) \log m}$;
   3.2 $A_i := (a_{j(i)+1}, \ldots, a_{j(i+1)})$
   (A_i ist leer, wenn $j(i) = j(i+1)$ gilt.)

end

A:	A_0	A_1	...	A_{k_m-1}
B:	B_0	B_1	...	B_{k_m-1}

wobei $A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_m)$ und $k_m = \frac{m}{\log m} \in \mathbb{N}$. Außerdem sollen alle $x \in A_i \cup B_i$ größer als die Elemente in $A_{i-1} \cup B_{i-1}$ sein. (Im Algorithmus 4.1 wird k_m als $k(m)$, a_{j_i+1} als $a_{j(i)+1}$ usw. geschrieben.)

Beispiel: (zum Algorithmus 4.1) Gegeben seien die folgenden zwei Felder A und B mit $m = 4$ und $k_m := k(m) = \frac{m}{\log m} = 2$

a_1 ... a_8 b_1 ... b_4
A=(4, 6, 7, 10, 12, 15, 18, 20) B=(3, 9, 16, 21)

Dann erhält man folgende Teilmengen: $A_0 = (4, 6, 7)$, $A_1 = (10, 12, 15, 18, 20)$, $B_0 = (3, 9)$ und $B_1 = (16, 21)$.

Das folgende Lemma gibt die Zeitkomplexität und Anzahl der Operationen an.

Lemma 4.26 *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 4.1 A und B in Paare von Teilfolgen (A_i, B_i) auf, so dass $|B_i| = O(\log m), \sum_i |A_i| = n$ und $C = (C_0, C_1, \ldots)$, wobei C_i die sortierte Folge ist, welche aus A_i und B_i durch Mischen entsteht. Dieser 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 $rank(b_{i \log m} : A) = j(i)$, erhalten wir $a_{j(i)} < b_{i \log m} < a_{j(i)+1}$. Dies impliziert, dass $b_{i \log m+1} > b_{i \log m} > a_{j(i)}$

und $a_{j(i)+1} > b_{i \log m}$ ist. Daher ist jedes Element von A_i und B_i größer als jedes Element von A_{i-1} oder B_{i-1}. Die Korrektheit des Algorithmus folgt hieraus unmittelbar.
Zeitanalyse: Der 1. Schritt braucht $O(1)$ sequentielle Zeit. Schritt 2 benötigt $O(\log n)$ Zeit, da die binäre Suche auf alle Elemente parallel angewendet wird. Die Gesamtzahl der für diesen Schritt auszuführenden Operationen ist $O\left((\log n) \times (\frac{m}{\log m})\right) = O(m+n)$, da $\left(\frac{m \log n}{\log m}\right) < \left(\frac{m \log(n+m)}{\log m}\right) \leq n+m$ für $n, m \geq 4$. Der 3. Schritt benötigt $O(1)$ paralleler Zeit, bei Benutzung einer linearen Anzahl von Operationen. Daher läuft dieser Algorithmus in $O(\log n)$ Zeit mit insgesamt $O(n+m)$ Operationen. □

Mit diesem Algorithmus haben wir das Mischproblem der Größe n auf Unterprobleme kleinerer Größe reduziert.

Optimaler Algorithmus für das Mischen:

- wende Algorithmus 4.1 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 4.1 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 4.27 *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 $X = (x_1, \ldots, x_n)$ von verschiedenen Elementen einer linear geordneten Menge $(S, \leq)$
- $y \in S$
- *Suchproblem:* Finde Index $i \in \{0, 1, \ldots, n\}$ mit $x_i \leq y < x_{i+1}$

Dabei seien $x_0 = -\infty, x_{n+1} = +\infty$ neue Elemente mit $-\infty < x < +\infty$ für alle $x \in S$.

binäres Suchen: Zeitkomplexität $O(\log n)$
paralleles Suchen mit $p \leq n$ Prozessoren: Prozessor P_1: zuständig für Initialisierung und Randsingularitäten. Aufteilen von X in $p+1$ Blöcke von etwa gleicher Länge. Jede parallele Runde findet $x_i = y$ oder einen y enthaltenden Block.

Algorithmus 4.2 (Paralleles Suchen für Prozessor P_j)

Eingabe: (1) Ein Feld $X = (x_1, x_2, \ldots, x_n)$ mit $x_1 < x_2 < \ldots < x_n$
(2) ein Element y
(3) die Anzahl p der Prozessoren mit $p \leq n$
(4) die Prozessornummer j mit $1 \leq j \leq p$
Ausgabe: ein Index i mit $x_i \leq y < x_{i+1}$
begin
1. **if**(j=1) **then do**
   1.1 Set $l := 0; r := n+1; x_0 := -\infty; x_{n+1} := +\infty$
   1.2.Set $c_0 := 0; c_{p+1} := 1$
2. **while**$(r - l > p)$ **do**
   2.1.**if**(j=1)**then**{set $q_0 := l; q_{p+1} := r$}
   2.2. Set $q_j := l + j\lfloor \frac{r-l}{p+1} \rfloor$
   2.3. **if**$(y = x_{q_i})$**then** {**return**(q_j); **exit**}
   else {set $c_j := 0$ **if** $(y > x_{q_j})$ and $c_j := 1$ **if** $(y < x_{q_j})$}
   2.4. **if**$(c_j < c_{j+1})$**then**{set $l := q_j; r := q_{j+1}$}
   2.5. **if**$(j = 1$ and $c_0 < c_1)$**then**{set $l := q_0; r := q_1$}
3. **if**$(j \leq r - l)$**then do**
   3.1. Case statement:
   $y = x_{l+j}$:{**return**$(l + j)$;**exit**}
   $y > x_{l+j}$:set $c_j := 0$
   $y < x_{l+j}$:set $c_j := 1$
   3.2. **if**$(c_{j-1} < c_j)$**then return**$(l + j - 1)$
end

Beispiel: (zum Algorithmus 4.2) Für die Eingabe

$$X = (2, 4, 6, \ldots, 30), y = 19, p = 2$$

hat P_1 nach Schritt 1 berechnet:

$$l = 0, r = 16, c_0 = 0, c_3 = 1, x_0 = -\infty, x_{16} = +\infty.$$

Die while-Schleife durchläuft 3 Iterationen.

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
c_1	0	0	0
c_2	1	0	0
c_3	1	1	1
l	5	7	9
r	10	10	10

Ergebnis durch P_1: **return**(q_1) mit $q_1 = 9$

Satz 4.28 *Zu der Folge $X = (x_1, x_2, \ldots, x_n)$ mit $x_1 < x_2 < \ldots < x_n$ und $y \in S$ berechnet der Algorithmus 4.2 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. □

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 4.29 *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 $rank(Y : X)$ in $O(1)$ Zeit mit $O(n)$ Operationen berechnet werden.*

Beweis:

- Bestimme $rank(y : X)$ für jedes $y \in X$ mit Algorithmus 4.2 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.

□

Nun soll $rank(B : A)$ für sortierte Folgen A mit $|A| = n$ und B mit $|B| = m$ berechnet werden. A und B sollen keine gemeinsamen Elemente enthalten.

Wieder: Berechnen durch Aufteilen von B in Blöcke der Länge von etwa $\sqrt{m}$:

Daraus: Algorithmus 4.3

Beispiel: (zum Algorithmus 4.3) $(m = 4, \sqrt{m} = 2)$

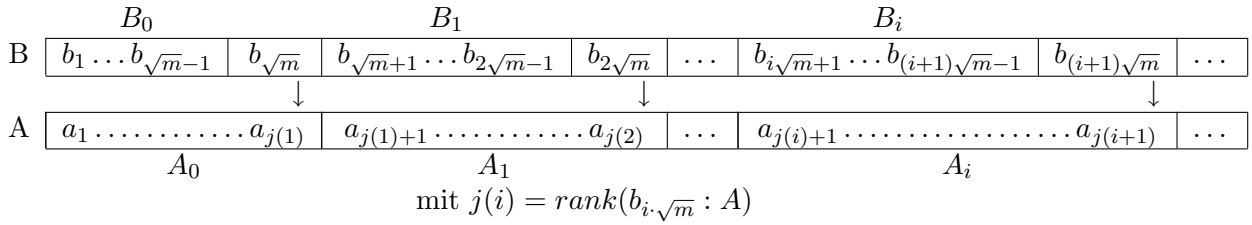


Abbildung 4.12: Aufteilung von B in Blöcke

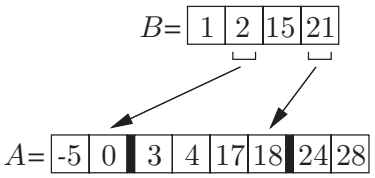
Algorithmus 4.3 (Ordne eine sortierte Folge in eine andere sortierte Folge ein)

Eingabe: Zwei Felder $A = (a_1, \dots, a_n)$ und $B = (b_1, \dots, b_m)$ in aufsteigender Reihenfolge.
Ausgabe: Das Feld $rank(B : A)$.
begin

1. Wenn $m < 4$ dann ordne die Elemente von B durch Anwendung des Algorithmus 4.2 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 4.2 ein. Dabei sei $p = \sqrt{n}$ und $rank(b_{i\lfloor\sqrt{m}\rfloor} : A) = j(i)$, für $1 \le i \le \sqrt{m}$ und $j(0) = 0$
3. Für $0 \le i \le \sqrt{m} - 1$ 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 $rank(B_i : A_i) = (0, \dots, 0)$, ansonsten berechne $rank(B_i : A_i)$ rekursiv.
4. Sei $1 \le k \le m$ ein willkürlicher Index, der kein Vielfaches von $\lfloor\sqrt{m}\rfloor$ ist und sei $i = \lfloor \frac{k}{\lfloor\sqrt{m}\rfloor} \rfloor$. Dann ist $rank(b_k : A) = j(i) + rank(b_k : A_i)$

end

- *1. Schritt*



- *2.Schritt* $j(0) = 0 \quad j(1) = 2 \quad j(2) = 6$
- *3.Schritt* $B_0 = (1) \quad A_0 = (-5, 0)$
  $B_1 = (15) \quad A_1 = (3, 4, 17, 18)$
- *4.Schritt* $rank(b_1, A) = rank(1, A) = j(0) + rank(1 : A_0) = 2$
  $rank(b_3, A) = rank(15, A) = j(1) + rank(15 : A_1) = 2 + 2 = 4$
  Also: $rank(B : A) = (\underline{2}, 2, \underline{4}, 6)$

Lemma 4.30 *Der Algorithmus 4.3 berechnet* $rank(B : A)$ *in der Zeit* $O(\log\log n)$ *mit* $O((n+m) \cdot \log\log m)$ *Operationen.*

Beweis:
Die Korrektheit wird durch Induktion über m bewiesen.

Der Induktionsanfang $m = 3$ bedeutet die Folge (b_1, b_2, b_3) in A einzuordnen. Dies erfolgt 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 $j(i) = rank(b_{i\sqrt{m}} : A)$ und $j(i+1) = rank(b_{(i+1)\sqrt{m}} : A)$ gilt $a_{j(i)} < b_{i\sqrt{m}}$ 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, das jedes Element p des Blocks B_i in den Block A_i eingefügt wird. Damit gilt $rank(p : A) = j(i) + rank(p : A_i)$, 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 4.2, wobei $p = \sqrt{n}$ gilt. Dessen Zeitkomplexität ist $O(\frac{\log(n+1)}{\log(\sqrt{n}+1)}) = O(1)$ und die Anzahl der Operationen wird durch $O(\sqrt{m}.\sqrt{n}) = O(n + m)$ beschränkt, da $2\sqrt{m}.\sqrt{n} \leq n + m$. Außerhalb der rekursiven Aufrufe benötigen Schritt 3 und 4 $O(1)$ Zeit mit $O(n + m)$ Operationen.

Sei $|A_i| = n_i$ für $0 \leq i \leq \sqrt{m} - 1$. Der zum Paar (B_i, A_i) gehörende rekursive Aufruf benötigt $T(n_i, \sqrt{m})$ Schritte. Also gilt $T(n, m) \leq max_i T(n_i, \sqrt{m}) + O(1)$ und $T(n, 3) = O(1)$. Eine Lösung dieser Rekurrenzungleichung ergibt $T(n, m) = O(\log \log m)$. Da die Anzahl der Schritte jedes rekursiven Aufrufs $O(n + m)$ ist, ergibt sich die Gesamtzahl der Operationen des Algorithmus 4.3 mit $O((n + m) \log \log m)$. □

Korollar 4.31 *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.

4.4 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 4.32 *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 4.4 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 $v = (h, j)$.

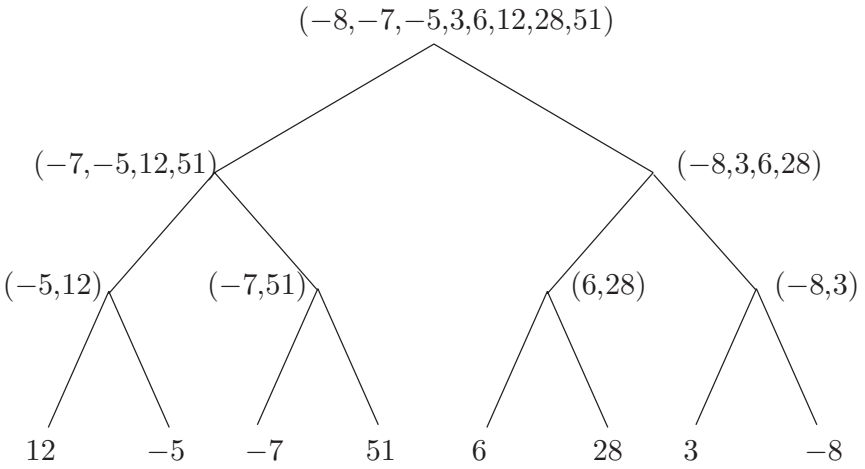


Abbildung 4.13: Binärbaum

Für obiges Beispiel wird das Ergebnis an der Wurzel $L(3, 1)$ erreicht.

Satz 4.33 *Der Algorithmus 4.4 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^hzu mischen. Also gilt nach Satz 4.32 für Operationen in

---

Algorithmus 4.4 (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 $X(2^h(j-1)+1), X(2^h(j-1)+2), \ldots, X(2^h j)$.
begin
1. **for** $1 \leq j \leq n$ **pardo**
   $L(0, j) := X(j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \leq j \leq \frac{n}{2^h}$ **pardo**
   Mische $L(h-1, 2j-1)$ und $L(h-1, 2j)$ zur sortierten Liste $L(h, j)$

end

Höhe h:

$$T(n) = O(log\, log\, 2^h) = O(log\, h) \leq O(log\, log\, n)$$

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 4.34 (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 Baumes[4] 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.

[4]Ein 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 $i \in \{0, \ldots, k-1\}$ genau $2^{2^{k-i-1}}$ Nachkommen. Die Knoten mit Höhe k haben 2 Blätter als Nachkommen.

Kapitel 5

Prozessalgebra

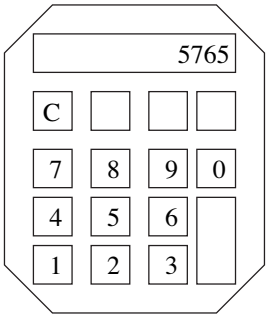
5.1 Einleitung

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 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.

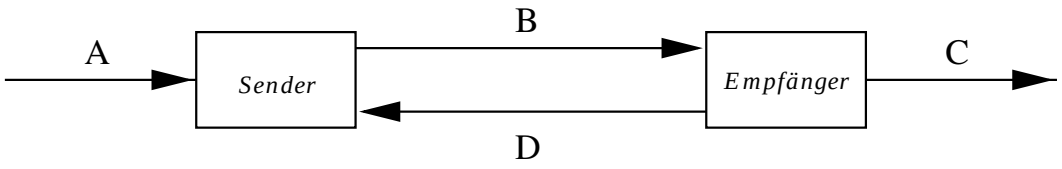
Die Darstellung in diesem Kapitel stützt sich in erster Linie auf [Fok99] und teilweise auf [Mil99]. Weitere einschlägige Literaturquellen sind: [BW90], [BV95], [Mil89] und [Bae95].

Beispiele

Taschenrechner:



Das Alternierbitprotokoll:



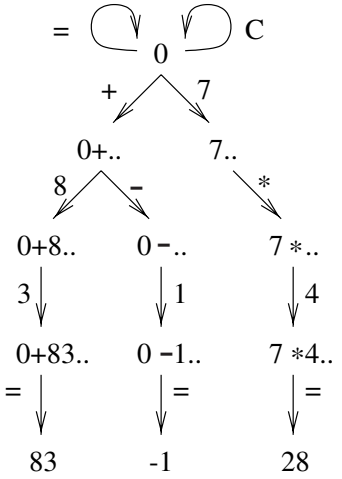
Beschreibung: siehe Beispiel 1.13 auf Seite 13.

5.2 Prozess-Graphen und elementare Prozessterme

In diesem Abschnitt werden Prozessterme und ihre Bedeutung definiert. Letzteres erfolgt durch spezielle Transitionssysteme wie in den folgenden Beispielen.

Beispiel:

Ausschnitt des Transitionssystems des Taschenrechner(beispiel)s:



Definition 5.1 *(elementare Prozess-Terme, basic process terms)*

Die Menge der elementaren Prozess-Terme *bzw.* Prozess-Ausdrücke BPA *wird aus* atomaren Aktionen $a \in A$ *sowie der* Auswahl *und* Hintereinanderausführung *gebildet:*

- *Jedes* $a \in A$ *ist ein BPA (*atomare Aktion*).*
- *Für alle* $t_1, t_2 \in BPA$ *ist* $(t_1 + t_2) \in BPA$ *(*Auswahl*: Es wird entweder* t_1 *oder* t_2 *ausgeführt).*
- *Für alle* $t_1, t_2 \in BPA$ *ist* $(t_1 \cdot t_2) \in BPA$ *(*Hintereinanderausführung, Sequenz*: Nach der ordnungsgemäßen Termination von* t_1 *wird* t_2 *ausgeführt).*

- *Nur nach diesen Regeln gebildete Terme liegen in* BPA.

*Die hier definierte Menge BPA ist die Grundmenge des auf Seite 161 eingeführten gleichnamigen Kalküls (**b**asic **p**rocess **a**lgebra)*[1]*. Durch die Regel „Sequenz (·) bindet stärker als Auswahl (+)“ können Klammern gespart werden. Der Operator Sequenz (·) kann weggelassen werden.*

Beispiel: $(((a + b) \cdot c) \cdot d) \in BPA$ oder kürzer $(a + b)cd \in BPA$ repräsentiert das Verhalten des linken Transitionssystems von Abbildung 5.1.

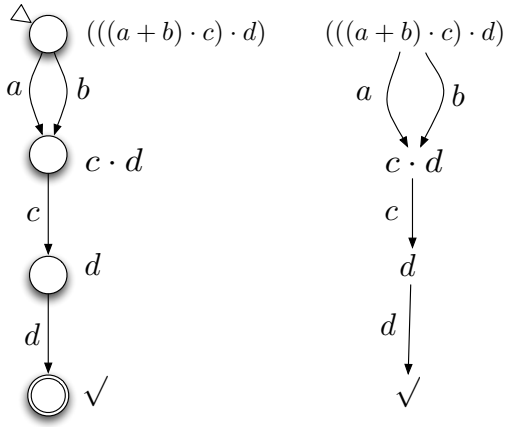


Abbildung 5.1: Prozessgraph von $((a + b)c)d)$ als Transitionssystem

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 - anders als im Beispiel des Taschenrechners - das noch bevorstehende (Rest-)Verhalten, nachdem dieser Zustand erreicht wurde. Ein Prozessgraph wird formal über folgenden Kalkül definiert, wobei (A_0) das Axiom ist. Danach werden die Schlussregeln angegeben.

Definition 5.2 *(Transitionsregeln)*

Für Prozessterme werden durch das folgende Axiom A_0 und nachfolgende Regeln Transitionsübergänge definiert, deren Zustände Prozessterme oder das Symbol $\surd$ sind.

Für $v \in A, x, y, x'y' \in BPA$ sei:

[1] Genau genommen muss also die Menge BPA vom Kalkül BPA unterschieden werden.

$$\frac{}{v \xrightarrow{v} \surd} \qquad (A_0)$$

$$\frac{x \xrightarrow{v} \surd}{x + y \xrightarrow{v} \surd} \qquad \frac{x \xrightarrow{v} x'}{x + y \xrightarrow{v} x'}$$

$$\frac{y \xrightarrow{v} \surd}{x + y \xrightarrow{v} \surd} \qquad \frac{y \xrightarrow{v} y'}{x + y \xrightarrow{v} y'}$$

$$\frac{x \xrightarrow{v} \surd}{x \cdot y \xrightarrow{v} y} \qquad \frac{x \xrightarrow{v} x'}{x \cdot y \xrightarrow{v} x' \cdot y}$$

Beispiel: Ableitung von $((a + b) \cdot c) \cdot d \xrightarrow{b} c \cdot d$ aus den Transitionsregeln:

$$\dfrac{\dfrac{\dfrac{b \xrightarrow{b} \surd}{a + b \xrightarrow{b} \surd}}{(a + b) \cdot c \xrightarrow{b} c}}{((a + b) \cdot c) \cdot d \xrightarrow{b} c \cdot d} \qquad \begin{array}{l} \dfrac{}{v \xrightarrow{v} \surd} \\[2ex] \dfrac{y \xrightarrow{v} \surd}{x + y \xrightarrow{v} \surd} \\[2ex] \dfrac{x \xrightarrow{v} \surd}{x \cdot y \xrightarrow{v} y} \\[2ex] \dfrac{x \xrightarrow{v} x'}{x \cdot y \xrightarrow{v} x' \cdot y} \end{array}$$

(links: die Ableitung im Kalkül, rechts: die benutzten Regeln)

Aufgabe 5.3 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) $(b + a)(cd)$

Definition 5.4 *Sei x ein Prozessterm. Dann heißt das Transitionssystem $TS = (S, A, tr, s_0, S^F)$* Prozessgraph von *$x$, wenn*

- *$S \subseteq BPA$ die mit dem Kalkül von Definition 5.2 aus x ableitbaren Prozessterme sind,*

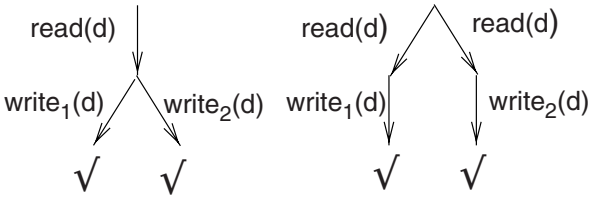
- *A die Menge der atomaren Aktionen von x,*
- *tr die im Kalkül von Definition 5.2 eingeführte Transitionsrelation,*
- $s_0 = x$ *der einzige Anfangszustand und*
- $S^F = \{\surd\}$ *der einzige Endszustand ist.*

Das Transitionssystem kann also auch als $TS = (S, A, tr, x, \surd)$ geschrieben werden. Abbildung 5.1 zeigt links einen Prozessgraphen als Transitionssystem mit Anfangs- und Endzustand. Prozessgraphen werden in den folgenden Beispielen wie in Abbildung 5.1 rechts dargestellt. Um die Graphiken übersichtlicher zu gestalten, wird oft der einzige Endzustand $\surd$ mehrfach gezeichnet.

5.3 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:



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

$$read(d)write_1(d) \text{ und } read(d)write_2(d)$$

als Schaltfolgen und 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 eine auf „Bisimulation“ beruhende Äquivalenz.

Äquivalenz durch Bisimulation

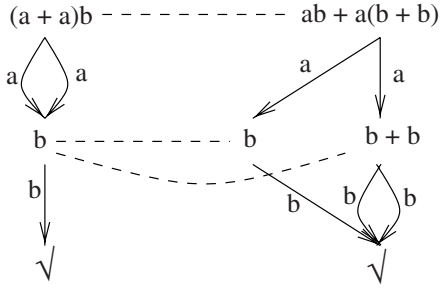
Eine *Bisimulation* ist eine binäre Relation $\mathcal{B}$ auf BPA (d.h. $\mathcal{B} \subseteq BPA \times BPA$)[2] mit folgenden Eigenschaften:

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 $p \xrightarrow{a} \surd$, dann $q \xrightarrow{a} \surd$
4. falls $p \mathcal{B} q$ und $q \xrightarrow{a} \surd$, dann $p \xrightarrow{a} \surd$

Zwei Prozesse p und q heißen *bisimilar* (Bezeichnung: $p \leftrightarrows q$), falls es eine Bisimulation $\mathcal{B}$ mit $p \mathcal{B} q$ gibt.

Lemma 5.5 *Die Bisimulation ist eine Äquivalenzrelation.*

Beispiel: $(a + a)b \leftrightarrows ab + a(b + b)$



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 $\surd \; \mathcal{B} \; \surd$ nicht explizit zu nennen.

Aufgabe 5.6 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) $((a + b)c + ac)d \; \mathcal{B} \; (b + a)(cd)$

Kongruenzäquivalenz

[2]Hier, wie auch in den folgenden Varianten von Bisimulation, sei der Fall $\mathcal{B} = \emptyset$ ausgeschlossen.

Theorem 5.7 *Die Äquivalenz der Bisimulation ist eine Kongruenz bezgl.* $\{+, \cdot\}$ *auf BPA, d.h.: falls* $s \leftrightarrows s'$ *und* $t \leftrightarrows t'$*, dann auch* $s + t \leftrightarrows s' + t'$ *und* $s \cdot t \leftrightarrows s' \cdot t'$*.*

Die Kongruenzeigenschaft folgt daraus, dass die Transitionsregeln in einer bestimmten Normalform (Panth-Form) sind.

Mit obenstehender Definition ist es aufwendig zu überprüfen, ob zwei elementare Prozessterme bisimilar sind: es müssen zunächst die Prozessgraphen und dann zwischen ihnen die Bisimulationsrelation konstruiert werden. Es wird daher jetzt ein Kalkül (**b**asic **p**rocess **a**lgebra) für eine Gleichheitsrelation zwischen elementaren Prozesstermen eingeführt, der diese Aufgabe durch die Konstruktion von Ableitungen lösen soll.

Axiome des BPA-Kalküls: Für x, y, $z \in BPA$ gelte:

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	$(x \cdot y) \cdot z$	$=$	$x \cdot (y \cdot z)$

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.

Korrektheit (soundness) Das BPA-Kalkül heißt *korrekt (sound)*, wenn für alle Terme $s, t \in BPA$ aus $s = t$ auch $s \leftrightarrows t$ folgt.

Vollständigkeit (completeness) Das BPA-Kalkül heißt *vollständig (complete)*, wenn für alle Terme $s, t \in BPA$ aus $s \leftrightarrows t$ auch $s = t$ folgt.

Satz 5.8 *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 $\sigma(s) \underline{\leftrightarrow} \sigma(t)$ 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) Gilt, da die Bisimulations-Relation eine Äquivalenz ist.

c) Gilt, da die Bisimulations-Relation eine Kongruenz ist.

□

Aufgabe 5.9

Motivieren Sie, dass folgendes Distributivgesetz nicht gilt: $x \cdot (y + z) = x \cdot y + x \cdot z$.
Hinweis: Setzen Sie $x = read(d)$ usw. in obigem Beispiel.

Satz 5.10 *Der BPA-Kalkül ist vollständig.*

Beweis:
Zu beweisen ist also, dass aus $s \underline{\leftrightarrow} t$ auch $s = t$ folgt. Zunächst wird der Beweis für die einfachere Relation $=_{\mathrm{AC}}$ bewiesen, d.h. $s \underline{\leftrightarrow} t \Rightarrow s =_{\mathrm{AC}} t$. Daraus wird dann die Behauptung des Satzes abgeleitet.

Per definitionem gelte $s =_{\mathrm{AC}} t$, 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 $s_1 + \ldots + s_k$ 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:

$$\begin{array}{lrcl}
\mathrm{R1} & x + y & =_{\mathrm{AC}} & y + x \\
\mathrm{R2} & (x + y) + z & =_{\mathrm{AC}} & x + (y + z) \\
\mathrm{R3} & x + x & \rightarrow & x \\
\mathrm{R4} & (x + y) \cdot z & \rightarrow & x \cdot z + y \cdot z \\
\mathrm{R5} & (x \cdot y) \cdot z & \rightarrow & x \cdot (y \cdot z)
\end{array}$$

Auf diese Weise erhalten wir ein Ersetzungskalkül, bei dem zwischen den Regelanwendungen Umformungen bzgl. der Relation $=_{\mathrm{AC}}$ angewendet werden können (siehe Algorithmus 5.1).

Wendet man die Regeln dieses Kalküls immer wieder auf einen elementaren Prozessterm $s \in BPA$ an, so gelangt man nach einer endlichen Zahl von Schritten zu einem elementaren Prozessterm $t \in BPA$, der nicht weiter reduziert werden kann. Allgemein heißt ein solches t in Ersetzungskalkülen *Normalform*. Das Ersetzungskalkül selbst heißt *terminierend*. Dies wird üblicherweise mit einer *Gewichtsfunktion*

$$gew : BPA \mapsto \mathbb{N}$$

bewiesen, die im vorliegenden Fall folgendermaßen definiert werden kann ($v \in A, s, t \in BPA$).

$$gew(v) := 2$$

$$gew(s + t) := gew(s) + gew(t)$$

$$gew(s \cdot t) := gew(s)^2 \cdot gew(t)$$

Da $gew(s_1) > gew(s_2) > \ldots > gew(s_q) > \ldots$ für jede Ableitung $s_1, s_2, \ldots, s_q, \ldots$ gilt, kann es keine unendlichen Ableitungen geben.

Terme in Normalform haben die Struktur $t_1 + \ldots + t_k$, 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' :

$$n \underline{\leftrightarrow} n' \Rightarrow n =_{\mathrm{AC}} n'$$

Algorithmus 5.1 (Entscheiden von $s \leftrightarrow t$ 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 163 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 163 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.)

- Falls n einen Summanden der Form a enthält, dann gilt $n \xrightarrow{a} \surd$ und wegen $n \leftrightarrow n'$ auch $n' \xrightarrow{a} \surd$. 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 $n \leftrightarrow n'$ auch $n' \xrightarrow{a} t$ mit $s \leftrightarrow t$. 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 $s =_{\mathrm{AC}} t$.

Da somit n und n' dieselben Summanden haben, gilt $n =_{\mathrm{AC}} n'$.

Um nun den Beweis von $s \leftrightarrow t \Rightarrow s = t$ zu führen, sei $s \leftrightarrow t$ 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 $s \leftrightarrow n$ und $t \leftrightarrow n'$, also insgesamt $n \leftrightarrow n'$. Für solche Normalformen wurde aber oben gezeigt: $n =_{\mathrm{AC}} n'$. Damit ergibt sich insgesamt: $s = n =_{\mathrm{AC}} n' = t$, d.h. $s = t$.

Der Begriff „Normalform“ beinhaltet, dass verschiedene Ableitungen auf die gleiche Form (modulo $=_{AC}$) führen. Dies folgt aus der Eigenschaft des Kalküls „konfluent“ zu sein, was aber hier nicht bewiesen wird.) □

Aus dem Beweis ergibt sich mit Algorithmus 5.1 ein Verfahren, mit dem man mit den Regeln R3, R4 und R5 von Seite 163 die Gültigkeit von $s \leftrightarrow t$ 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.

Satz 5.11 *Der Algorithmus 5.1 entscheidet korrekt, ob zwei Prozessterme s und t aus BPA äquivalent sind.*

Beispiel: Zu entscheiden ist: $(a + a)(cd) + (bc)(d + d) \stackrel{?}{=} ((b + a)(c + c))d$

	$s \equiv (\underline{a + a})(cd) + (bc)(d + d)$		$t \equiv ((b + a)(\underline{c + c}))d$
$\stackrel{\mathrm{A3}}{\rightarrow}$	$a(cd) + (bc)(\underline{d + d})$	$\stackrel{\mathrm{A3}}{\rightarrow}$	$\underline{((b + a)c)d}$
$\stackrel{\mathrm{A3}}{\rightarrow}$	$a(cd) + \underline{(bc)d}$	$\stackrel{\mathrm{A5}}{\rightarrow}$	$\underline{(b + a)(cd)}$
$\stackrel{\mathrm{A5}}{\rightarrow}$	$a(cd) + b(cd) \equiv n$	$\stackrel{\mathrm{A4}}{\rightarrow}$	$b(cd) + a(cd) \equiv n'$

Die beiden berechneten Normalformen n und n' sind äquivalent (modulo $=_{\mathrm{AC}}$) und daher auch die Ausgangsterme s und t.

Aufgabe 5.12 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) $((a+b)c + ac)d = (b+a)(cd)$

5.4 Parallele und kommunizierende Prozesse

Durch den *Paralleloperator (merge)* $\|$ wird die parallele (besser: nebenläufige) Ausführung der beiden Prozesse dargestellt, die er als Argument hat. In den folgenden Regeln sei $\{v, w\} \subseteq A$ und x, x', y, y' seien Prozessterme.

$$\frac{x \xrightarrow{v} \surd}{x\|y \xrightarrow{v} y} \qquad \frac{x \xrightarrow{v} x'}{x\|y \xrightarrow{v} x'\|y}$$

$$\frac{y \xrightarrow{v} \surd}{x\|y \xrightarrow{v} x} \qquad \frac{y \xrightarrow{v} y'}{x\|y \xrightarrow{v} x\|y'}$$

Zwei parallel ablaufende Prozesse kommunizieren mittels einer Kommunikationsfunktion.

Eine *Kommunikationsfunktion* $\gamma : A \times A \to A$ erzeugt für jedes Paar atomarer Aktionen a und b ihre Kommunikations-Aktion $\gamma(a, b)$.

Die Kommunikationsfunktion γ ist kommutativ und assoziativ:

$$\begin{aligned} \gamma(a,b) &\equiv \gamma(b,a) \\ \gamma(\gamma(a,b),c) &\equiv \gamma(a,\gamma(b,c)) \end{aligned}$$

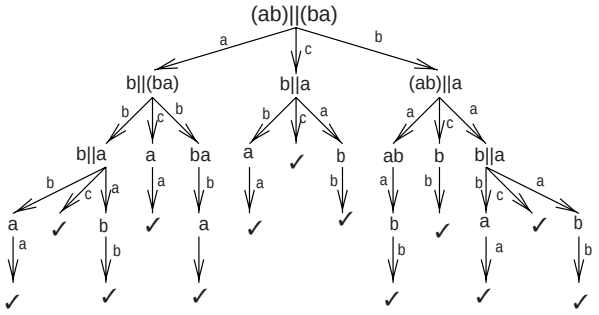
Der Paralleloperator kann eine solche Kommunikation enthalten. Sie ist eine unteilbare Aktion beider Prozesse.

$$\frac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} \surd}{x\|y \xrightarrow{\gamma(v,w)} \surd} \qquad \frac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} y'}{x\|y \xrightarrow{\gamma(v,w)} y'}$$

$$\frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} \surd}{x\|y \xrightarrow{\gamma(v,w)} x'} \qquad \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} y'}{x\|y \xrightarrow{\gamma(v,w)} x'\|y'}$$

Beispiel 5.13

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:

Der *Links-Merge-Operator (left merge)* $\between$ erlaubt die Ausführung der ersten Transition des ersten (linken) Argumentes:

$$\frac{x \xrightarrow{v} \surd}{x \between y \xrightarrow{v} y} \qquad \frac{x \xrightarrow{v} x'}{x \between y \xrightarrow{v} x' \| y}$$

Der *Kommunikations-Merge-Operator (communication merge)* $|$ stellt die Kommunikation der beiden ersten Transitionen der beiden Argumente dar:

$$\frac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} \surd}{x|y \overset{\gamma(v,w)}{\rightarrow} \surd} \qquad \frac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} y'}{x|y \overset{\gamma(v,w)}{\rightarrow} y'}$$

$$\frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} \surd}{x|y \overset{\gamma(v,w)}{\rightarrow} x'} \qquad \frac{x \xrightarrow{v} x' \quad y \xrightarrow{w} y'}{x|y \overset{\gamma(v,w)}{\rightarrow} x' \| y'}$$

Die Erweiterung der elementaren Prozessalgebra um die Operatoren *Paralleloperator (merge)* $\|$, *Links-Merge-Operator (left merge)* $\between$ und *Kommunikations-Merge-Operator (communication merge)* $|$ heißt *PAP* (*Prozessalgebra mit Parallelismus*). In ihr sollen die neuen Paralleloperatoren stärker binden als $+$, d.h.: $a \between b + a \| b$ steht für $(a \between b) + (a \| b)$. Der Paralleloperator $\|$ kann durch $\between$ und $|$ ausgedrückt werden: $s \| t \leftrightarrows (s \between t + t \between s) + s|t$.

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 5.14 *Die Äquivalenzrelation Bisimulation ist eine Kongruenzrelation in PAP, d.h.: wenn $s \underline{\leftrightarrow} s'$ und $t \underline{\leftrightarrow} t'$, dann*

- $s + t \underline{\leftrightarrow} s' + t'$,
- $s \cdot t \underline{\leftrightarrow} s' \cdot t'$,
- $s \| t \underline{\leftrightarrow} s' \| t'$,
- $s \between t \underline{\leftrightarrow} s' \between t'$ *und*
- $s | t \underline{\leftrightarrow} s' | t'$.

Axiome des PAP-Kalküls: Axiome A1, ..., A5 (Seite 161) und

$$
\begin{array}{lrcl}
\text{M1} & x \| y & = & (x \between y + y \between x) + x | y \\
 & & & \\
\text{LM2} & v \between y & = & v \cdot y \\
\text{LM3} & (v \cdot x) \between y & = & v \cdot (x \| y) \\
\text{LM4} & (x + y) \between z & = & x \between z + y \between z \\
 & & & \\
\text{CM5} & v | w & = & \gamma(v, w) \\
\text{CM6} & v | (w \cdot y) & = & \gamma(v, w) \cdot y \\
\text{CM7} & (v \cdot x) | w & = & \gamma(v, w) \cdot x \\
\text{CM8} & (v \cdot x) | (w \cdot y) & = & \gamma(v, w) \cdot (x \| y) \\
\text{CM9} & (x + y) | z & = & x | z + y | z \\
\text{CM10} & x | (y + z) & = & x | y + x | z
\end{array}
$$

Satz 5.15 *Der PAP-Kalkül ist korrekt, d.h.: $s = t \;\Rightarrow\; s \underline{\leftrightarrow} t$.*

Beweisskizze: Da die Bisimulationsäquivalenz eine Kongruenz ist, genügt es für jedes Axiom $s = t$ die Relation $\sigma(s) \underline{\leftrightarrow} \sigma(t)$ für alle Substitutionen von den Variablen aus s und t in Prozessterme zu beweisen.

Satz 5.16 *Der PAP-Kalkül ist vollständig, d.h.: $s \underline{\leftrightarrow} t \;\Rightarrow\; s = t$.*

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 $s \underline{\leftrightarrow} t$ ist, wobei s und t Normalformen s' und t' haben, dann gilt $s' =_{\mathrm{AC}} t'$, also auch $s = s' =_{\mathrm{AC}} t' = t$.

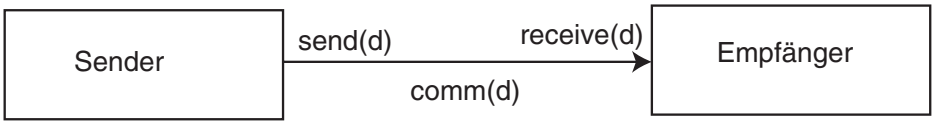


Abbildung 5.2: Kommunikation mit Sender und Empfänger

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. 5.2) 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:

$$\frac{x \xrightarrow{v} \surd \quad (v \notin H)}{\partial_H(x) \xrightarrow{v} \surd} \qquad \frac{x \xrightarrow{v} x' \quad (v \notin H)}{\partial_H(x) \xrightarrow{v} \partial_H(x')}$$

Der Bildbereich der Kommunikationsfunktion γ wird um δ erweitert:

$$\gamma : A \times A \to A \cup \{\delta\}.$$

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 5.17 *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 : die Axiome von PAP und folgende für Abbruch und Unterdrücken:

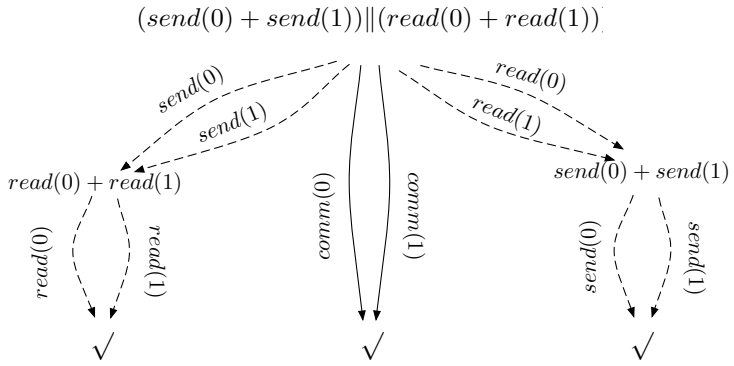
$$
\begin{array}{lrcll}
\text{A6} & x + \delta & = & x & \\
\text{A7} & \delta \cdot x & = & \delta & \\
 & & & & \\
\text{D1} & \partial_H(v) & = & v & (v \notin H) \\
\text{D2} & \partial_H(v) & = & \delta & (v \in H) \\
\text{D3} & \partial_H(\delta) & = & \delta & \\
\text{D4} & \partial_H(x + y) & = & \partial_H(x) + \partial_H(y) & \\
\text{D5} & \partial_H(x \cdot y) & = & \partial_H(x) \cdot \partial_H(y) & \\
 & & & & \\
\text{LM11} & \delta \between x & = & \delta & \\
\text{CM12} & \delta | x & = & \delta & \\
\text{CM13} & x | \delta & = & \delta &
\end{array}
$$

Beispiel 5.18 Der Prozessterm t zum einführenden Beispiel in Abb. 5.2 lautet:

$t \equiv \partial_{\{send(0),send(1),read(0),read(1)\}}((send(0) + send(1)) \| (read(0) + read(1)))$

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 5.19 *a) Bisimulation ist eine Kongruenz für ACP: wenn $s \leftrightarrows s'$ und $t \leftrightarrows t'$, dann $s + t \leftrightarrows s' + t'$, $s \cdot t \leftrightarrows s' \cdot t'$, $s \| t \leftrightarrows s' \| t'$, $s \between t \leftrightarrows s' \between t'$, $s | t \leftrightarrows s' | t'$ und $\partial_H(s) \leftrightarrows \partial_H(s')$.*

b) Der Kalkül ACP ist korrekt: $s = t \Rightarrow s \leftrightarrows t$.

c) Der Kalkül ACP ist vollständig: $s \leftrightarrows t \Rightarrow s = t$.

Beispiel 5.20

Seien $\gamma(a,b) \equiv c$ und $\gamma(a',b') \equiv c'$ zunächst die einzigen Kommunikationsaktionen zwischen Aktionen.

$$
\begin{aligned}
&(a+a')\|(b+b')\\
&\overset{\text{M1}}{=}\\
&(a+a')\between(b+b') + (b+b')\between(a+a')\\
&+ (a+a')|(b+b')\\
&\overset{\text{LM4, CM9,10}}{=}\\
&a\between(b+b') + a'\between(b+b') + b\between(a+a') + b'\between(a+a')\\
&+ a|b + a|b' + a'|b + a'|b'\\
&\overset{\text{LM2, CM5}}{=}\\
&a\cdot(b+b') + a'\cdot(b+b') + b\cdot(a+a') + b'\cdot(a+a')\\
&+ c + \delta + \delta + c'\\
&\overset{\text{A6}}{=}\\
&a\cdot(b+b') + a'\cdot(b+b') + b\cdot(a+a') + b'\cdot(a+a')\\
&+ c + c'
\end{aligned}
$$

Beispiel 5.21 (Fortsetzung)

Sei nun $H = \{a, a', b, b'\}$.

$$
\begin{aligned}
&\partial_H((a+a')\|(b+b'))\\
&=\\
&\partial_H(a\cdot(b+b') + a'\cdot(b+b') + b\cdot(a+a')\\
&+ b'\cdot(a+a') + c + c')\\
&\overset{\text{D1,2,4,5}}{=}\\
&\delta\cdot\partial_H(b+b') + \delta\cdot\partial_H(b+b') + \delta\cdot\partial_H(a+a')\\
&+ \delta\cdot\partial_H(a+a') + c + c'\\
&\overset{\text{A6,7}}{=}\\
&c + c'
\end{aligned}
$$

∂_H erzwingt also die Kommunikation zwischen a und b einerseits und zwischen a' und b' andererseits.

Aufgabe 5.22 Konstruieren Sie die Prozessgraphen zu folgenden Prozesstermen:

a) $\partial_{\{a\}}(ac)$

b) $\partial_{\{a\}}((a+b)c)$

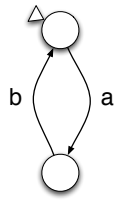


Abbildung 5.3: Transitionssystem für abab...

c) $\partial_{\{c\}}((a+b)c)$

d) $\partial_{\{a,b\}}((ab)\|(ba))$ mit $\gamma(a,b) = c$

5.5 Rekursion und Abstraktion

Bislang wurden nur Prozesse endlicher Länge spezifiziert. Unendliche Prozesse, wie z.B. der Prozess *ababab*... mit dem Transitionssystem von Abb. 5.3 werden durch Rekursion definiert.

Konkret geschieht dies z.B. durch das folgende *rekursive Gleichungssystem*:

$$\begin{aligned} X &= aY \\ Y &= bX \end{aligned}$$

Dabei sind X und Y *Rekursionsvariablen*, die zwei Zustände des Prozesses repräsentieren.

Definition 5.23 *Eine* rekursive Spezifikation *ist eine Menge von Gleichungen der Form:*

$$\begin{aligned} X_1 &= t_1(X_1, \ldots, X_n) \\ &\vdots \\ X_n &= t_n(X_1, \ldots, X_n) \end{aligned}$$

wobei $t_i(X_1, \ldots, X_n)$ Prozessterme des Kalküls ACP mit (möglichen) freien Variablen $X_1, \ldots, X_n$ sind.

Definition 5.24 *Prozesse $p_1, \ldots, p_n$ werden als eine* Lösung *einer rekursiven Spezifikation*

$$\{X_i = t_i(X_1, \ldots, X_n) \mid i \in \{1, \ldots, n\}\}$$

modulo Bisimulations-Äquivalenz bezeichnet, falls $p_i \leftrightarrows t_i(p_1, \ldots, p_n)$ für $i \in \{1, \ldots, n\}$ gilt.

Lösungen sollen eindeutig bezüglich Bisimulations-Äquivalenz sein, d.h. falls $p_1, \ldots, p_n$ und $q_1, \ldots, q_n$ zwei solche Lösungen sind, dann erwarten wir $p_i \leftrightarrows q_i$ for $i \in \{1, \ldots, n\}$.

Beispiel 5.25

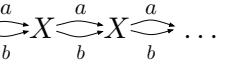
a) $X = X$ hat alle Prozesse als Lösung.

b) Alle Prozesse p mit $p \xrightarrow{a} \surd$ sind Lösungen von der rekursiven Spezifikation $\{X = a + X\}$. 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 $\{X = Xa\}$.

d) $\{X = aY, Y = bX\}$ hat als einzige Lösung den Prozess mit Prozessgraph $X \xrightarrow{a} Y \xrightarrow{b} X \xrightarrow{a} Y \xrightarrow{b} \ldots$ für X und $Y \xrightarrow{b} X \xrightarrow{a} Y \xrightarrow{b} X \xrightarrow{a} \ldots$ für Y.

e) $\{X = Y, Y = aX\}$ hat als einzige Lösung $X \xrightarrow{a} X \xrightarrow{a} X \xrightarrow{a} X \xrightarrow{a} \ldots$ für X und Y.

f) $\{X = (a + b) \between X\}$ hat als einzige Lösung 

„Einzig“ bzw. „zwei verschiedene“ ist hier jeweils modulo Bisimulation zu verstehen.

Definition 5.26 *Eine rekursive Spezifikation*

$$\{X_i = t_i(X_1, \ldots, X_n) \mid i \in \{1, \ldots, n\}\}$$

heißt geschützt (guarded) *falls die rechten Seiten ihrer Gleichungen in folgende Form*

$$a_1 \cdot s_1(X_1, \ldots, X_n) + \cdots + a_k \cdot s_k(X_1, \ldots, X_n) + b_1 + \cdots + b_\ell$$

überführt werden können, indem die Axiome des Kalküls ACP benutzt werden. Außerdem dürfen bei dieser Umformung Variable einer Rekursionsgleichung durch die entsprechende rechte Seite ersetzt werden.

Eine rekursive Spezifikation ist genau dann eindeutig (modulo Bisimulation), wenn sie geschützt ist. Zum Beispiel sind einige der rekursiven Spezifikationen von Beispiel 5.25 nicht geschützt.

Beispiel 5.27 Sei $\gamma(a, b) \equiv c$ und $\gamma(b, b) \equiv c$. $\{X{=}Y \| Z,\ Y{=}Z + a,\ Z{=}bZ\}$ ist geschützt, denn:

- $Z{=}bZ$ hat schon die gewünschte Form.
- $Y{=}Z + a$ wird transformiert, indem Z durch bZ ersetzt wird.
- $X{=}Y \| Z$ 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:

$$\begin{aligned} & (bZ + a) \| bZ \\ = \ & (bZ + a) \between bZ + bZ \between (bZ + a) + (bZ + a) | bZ \\ = \ & bZ \between bZ + a \between bZ + bZ \between (bZ + a) + bZ | bZ + a | bZ \\ = \ & b(Z \| bZ) + abZ + b(Z \| (bZ + a)) + c(Z \| Z) + cZ \end{aligned}$$

Definition 5.28 *Für eine geschützte rekursive Spezifikation E:*

$$\{X_i = t_i(X_1, \ldots, X_n) \mid i \in \{1, \ldots, n\}\}$$

soll nun

$$\langle X_i | E \rangle \qquad (i \in \{1, \ldots, n\})$$

die Lösung X_i in E bedeuten.

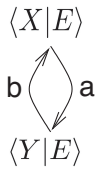
Transitionsregeln für Rekursion

$$\frac{t_i(\langle X_1|E\rangle, \ldots, \langle X_n|E\rangle) \xrightarrow{v} \surd}{\langle X_i|E\rangle \xrightarrow{v} \surd}$$

$$\frac{t_i(\langle X_1|E\rangle, \ldots, \langle X_n|E\rangle) \xrightarrow{v} y}{\langle X_i|E\rangle \xrightarrow{v} y}$$

d.h.: $\langle X_i|E\rangle$ übernimmt das Verhalten von $t_i(\langle X_1|E\rangle, \ldots, \langle X_n|E\rangle)$:

Beispiel 5.29 Sei E die geschützte rekursive Spezifikation $\{X{=}aY,\ Y{=}bX\}$. Der Prozessgraph von $\langle X|E\rangle$ ist:



Wir leiten her: $\langle X|E\rangle \xrightarrow{a} \langle Y|E\rangle$:

$$\dfrac{\dfrac{a \xrightarrow{a} \surd \qquad \dfrac{}{v \xrightarrow{v} \surd} \quad v := a}{a \cdot \langle Y|E\rangle \xrightarrow{a} \langle Y|E\rangle} \qquad \dfrac{x \xrightarrow{v} \surd}{x \cdot y \xrightarrow{v} y} \quad v := a,\ x := a,\ y := \langle Y|E\rangle}{\langle X|E\rangle \xrightarrow{a} \langle Y|E\rangle} \qquad \dfrac{a \cdot \langle Y|E\rangle \xrightarrow{v} y}{\langle X|E\rangle \xrightarrow{v} y} \quad v := a,\ y := \langle Y|E\rangle$$

Aufgabe 5.30 Leiten Sie für die folgenden drei Prozessterme den Prozessgraphen ab:

a) $\langle X|X = ab\rangle$

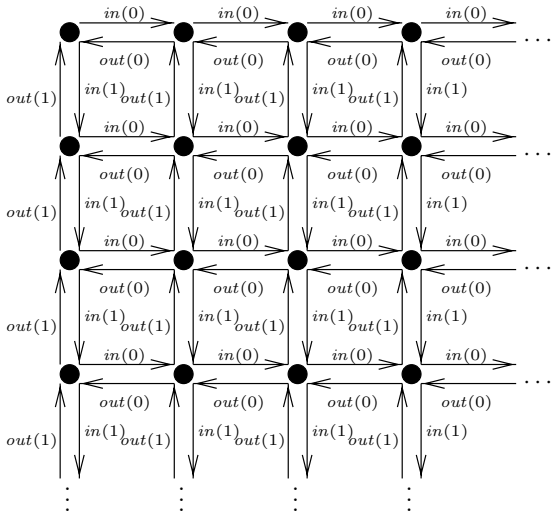
b) $\langle X|X = aXb\rangle$

c) $\langle X|X = aXb + c\rangle$

Anmerkung: ACP mit geschützter Rekursion ist eine konservative Erweiterung von ACP und Bisimulation ist eine Kongruenzrelation.

Beispiel 5.31 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:

$$X \;=\; in(0)(X \| out(0)) + in(1)(X \| out(1))$$

Aufgabe 5.32 Geben Sie eine Bisimulations-Relation für Beispiel 5.31 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.

Axiome für Rekursion:

Für eine rekursive Spezifikation E

$$\{X_i = t_i(X_1, \ldots, X_n) \mid i \in \{1, \ldots, n\}\}$$

gelte:

Rekursives Definitions Prinzip:

$$\text{RDP} \qquad \langle X_i|E\rangle \;=\; t_i(\langle X_1|E\rangle,\ldots,\langle X_n|E\rangle) \qquad (i\in\{1,\ldots,n\})$$

Rekursives Spezifikations Prinzip:

$$\begin{array}{ll}\text{RSP} & \text{Falls } y_i = t_i(y_1,\ldots,y_n) \text{ für } i\in\{1,\ldots,n\},\\ & \text{dann} \quad y_i \;=\; \langle X_i|E\rangle \qquad (i\in\{1,\ldots,n\})\end{array}$$

Korrektheit

Der Kalkül ACP mit geschützter Rekursion ist korrekt bezüglich Bisimulsation:

$$s = t \;\Rightarrow\; s \leftrightarrows t$$

RSP ist **nicht** korrekt für ungeschützte Rekursion. Beispielsweise ergibt RSP wegen $t = t$ die Gleichung

$$t \;=\; \langle X \,|\, X{=}X\rangle$$

für alle Prozessterme t.

Beispiel 5.33

$$\begin{array}{rcl}\langle Z \,|\, Z{=}aZ\rangle & \overset{\text{RDP}}{=} & a\langle Z \,|\, Z{=}aZ\rangle\\ & \overset{\text{RDP}}{=} & a(a\langle Z \,|\, Z{=}aZ\rangle)\\ & \overset{\text{A5}}{=} & (aa)\langle Z \,|\, Z{=}aZ\rangle\end{array}$$

Also gilt mit RSP,

$$\langle Z \,|\, Z{=}aZ\rangle \;=\; \langle X \,|\, X{=}(aa)X\rangle$$

Weiter gilt:

$$\begin{array}{rcl}\langle Z \,|\, Z{=}aZ\rangle & \overset{\text{RDP}}{=} & a\langle Z \,|\, Z{=}aZ\rangle\\ & \overset{\text{RDP}}{=} & a(a\langle Z \,|\, Z{=}aZ\rangle)\\ & \overset{\text{RDP}}{=} & a(a(a\langle Z \,|\, Z{=}aZ\rangle))\\ & \overset{\text{A5}}{=} & ((aa)a)\langle Z \,|\, Z{=}aZ\rangle\end{array}$$

und mit RSP:

$$\langle Z \,|\, Z{=}aZ\rangle \;=\; \langle Y \,|\, Y{=}((aa)a)Y\rangle$$

also:

$$\begin{array}{rcl}\langle X \,|\, X{=}(aa)X\rangle & = & \langle Z \,|\, Z{=}aZ\rangle\\ & = & \langle Y \,|\, Y{=}((aa)a)Y\rangle\end{array}$$

Aufgabe 5.34 Prüfen Sie für $\gamma(a,b) \equiv c$ die folgende Ableitung von

$$\partial_{\{a,b\}}(\langle X \mid X{=}aX\rangle \| \langle Y \mid Y{=}bY\rangle) = \langle Z \mid Z{=}cZ\rangle$$

$$\begin{array}{rl} & \langle X \mid X{=}aX\rangle \| \langle Y \mid Y{=}bY\rangle \\ = & \langle X \mid X{=}aX\rangle \between \langle Y \mid Y{=}bY\rangle \\ + & \langle Y \mid Y{=}bY\rangle \between \langle X \mid X{=}aX\rangle \\ + & \langle X \mid X{=}aX\rangle | \langle Y \mid Y{=}bY\rangle \\ = & a(\langle X \mid X{=}aX\rangle \| \langle Y \mid Y{=}bY\rangle) \\ + & b(\langle Y \mid Y{=}bY\rangle \| \langle X \mid X{=}aX\rangle) \\ + & c(\langle X \mid X{=}aX\rangle \| \langle Y \mid Y{=}bY\rangle) \end{array}$$

Aus

$$\begin{array}{rl} & \partial_{\{a,b\}}(\langle X \mid X{=}aX\rangle \| \langle Y \mid Y{=}bY\rangle) \\ = & c \cdot \partial_{\{a,b\}}(\langle X \mid X{=}aX\rangle \| \langle Y \mid Y{=}bY\rangle) \end{array}$$

folgt mit RSP das Ergebnis.

Sei E, E' eine geschützte rekursive Spezifikation, wobei E' aus E entsteht, indem die rechten Seiten ihrer rekursiven Gleichungen modifiziert werden, indem

- die Axiome für ACP mit geschützter Rekursion benutzt werden und
- Rekursionsvariable durch rechte Seiten entsprechender Rekursionsgleichungen ersetzt werden.

Dann kann $\langle X|E\rangle = \langle X|E'\rangle$ im Kalkül ACP mitgeschützter Rekursion für alle Rekursionsvariablen abgeleitet werden. Zum Beispiel kann

$$\langle X \mid X{=}aX{+}aX\rangle = \langle X \mid X{=}(aa)X\rangle$$

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: $((xy)z = x(yz))$.

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:

$$\frac{}{\tau \xrightarrow{\tau} \surd}$$

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 $\gamma : A' \times A' \to A \cup \{\delta\}$ 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:

$$\frac{x \xrightarrow{v} \surd \quad (v \notin I)}{\tau_I(x) \xrightarrow{v} \surd} \qquad \frac{x \xrightarrow{v} x' \quad (v \notin I)}{\tau_I(x) \xrightarrow{v} \tau_I(x')}$$

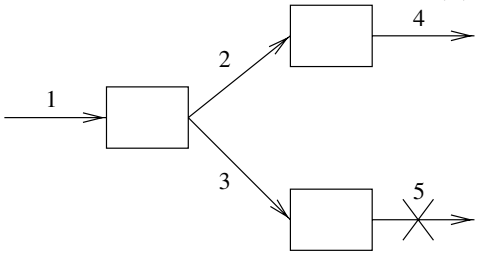
$$\frac{x \xrightarrow{v} \surd \quad (v \in I)}{\tau_I(x) \xrightarrow{\tau} \surd} \qquad \frac{x \xrightarrow{v} x' \quad (v \in I)}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')}$$

Der Kalkül ACP über A' mit τ-Transition und Abstraktionsoperator wird mit ACP_τ bezeichnet.

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 5.35

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 169.)

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 5.36 Weitere nichtäquivalente Prozessausdrücke:

- $a + \tau\delta$ und a
- $\partial_{\{b\}}(a + \tau b)$ und $\partial_{\{b\}}(a + b)$
- $a + \tau b$ und $a + b$

Es stellt sich also die Frage: Welche τ-Transitionen sind wirkungslos?

Antwort: diejenigen, deren Streichung nicht das Verhalten ändern, wie zum Beispiel: $a+\tau(a+b)$ und $a+b$. Nach der Ausführung von τ ist a immer noch ausführbar! Dies wird durch die folgende Definition der *Verzweigungs-Bisimulation* (branching bisimulation) formalisiert.

Definition 5.37 *Eine* Verzweigungs-Bisimulation (branching bisimulation) *ist eine binäre Relation $\mathcal{B}$ auf Prozessen, für die mit $a \in A' = A \cup \{\tau\}$ gilt:*

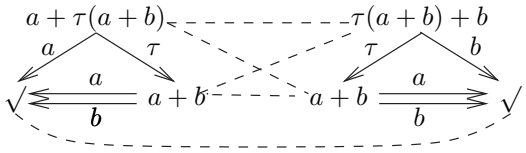
1. *Wenn $p \mathcal{B} q$ und $p \xrightarrow{a} p'$, dann*
   - *entweder $a \equiv \tau$ [3] und $p' \mathcal{B} q$*
   - *oder $q \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_0$ mit $p \mathcal{B} q_0$ und $q_0 \xrightarrow{a} q'$ mit $p' \mathcal{B} q'$*
2. *Wenn $p \mathcal{B} q$ und $q \xrightarrow{a} q'$, dann*
   - *entweder $a \equiv \tau$ und $p \mathcal{B} q'$*
   - *oder $p \xrightarrow{\tau} \cdots \xrightarrow{\tau} p_0$ mit $p_0 \mathcal{B} q$ und $p_0 \xrightarrow{a} p'$ mit $p' \mathcal{B} q'$*
3. *Wenn $p \mathcal{B} q$ und $p \xrightarrow{a} \surd$, dann $q \xrightarrow{\tau} \cdots \xrightarrow{\tau}\xrightarrow{a}\xrightarrow{\tau} \cdots \xrightarrow{\tau} \surd$.*
4. *Wenn $p \mathcal{B} q$ und $q \xrightarrow{a} \surd$, dann $p \xrightarrow{\tau} \cdots \xrightarrow{\tau}\xrightarrow{a}\xrightarrow{\tau} \cdots \xrightarrow{\tau} \surd$.*

Die τ-Folgen haben eine endliche Länge $n \geq 0$. Zwei Prozesse p und q heißen verzweigungsbisimilar (branching bisimilar)*, in Zeichen $p \leftrightarrow_b q$, wenn es eine Verzweigungs-Bisimulations-Relation $\mathcal{B}$ gibt mit $p \mathcal{B} q$.*

Beispiel 5.38

$a + \tau(a + b) \leftrightarrow_b \tau(a + b) + b$

[3] $\equiv$ bezeichnet die syntaktische Gleichheit.



Die Relation $\mathcal{B}$ ist definiert durch:

$a + \tau(a + b)\,\mathcal{B}\,\tau(a + b) + b$

$a + b\,\mathcal{B}\,\tau(a + b) + b$

$a + \tau(a + b)\,\mathcal{B}\,a + b$

$a + b\,\mathcal{B}\,a + b$

$\surd\,\mathcal{B}\,\surd$

Aufgabe 5.39

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 $a + b$. Milner [Mil89] hat gezeigt, dass man dieses Problem dadurch lösen kann, dass eine Initialisierungsbedingung gefordert wird: initiale τ-Transitionen werden nie eliminiert, das heißt in solchen Fällen wird keine Äquivalenz definiert:

- $a + \tau\delta$ und a
- $\partial_{\{b\}}(a + \tau b)$ und $\partial_{\{b\}}(a + b)$
- $a + \tau b$ und $a + b$
- b und τb

Definition 5.40 *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 $p' \leftrightarrow_b q'$.*
2. *Wenn $p\,\mathcal{B}\,q$ und $q \xrightarrow{a} q'$, dann $p \xrightarrow{a} p'$ mit $p' \leftrightarrow_b q'$.*
3. *Wenn $p\,\mathcal{B}\,q$ und $p \xrightarrow{a} \surd$, dann $q \xrightarrow{a} \surd$.*

4. *Wenn* $p \mathcal{B} q$ *und* $q \xrightarrow{a} \surd$, *dann* $p \xrightarrow{a} \surd$.

Zwei Prozesse p *und* q *heißen* initial verzweigungsbisimilar (rooted branching bisimilar), *in Zeichen* $p \leftrightarrows_{rb} q$, *wenn es eine initiale Verzweigungs-Bisimulations-Relation* $\mathcal{B}$ *gibt mit* $p \mathcal{B} q$.

Initiale Verzweigungs-Bisimulation ist wie Verzweigungs-Bisimulation eine Äquivalenzrelation. Es gilt $\leftrightarrows \subseteq \leftrightarrows_{rb} \subseteq \leftrightarrows_b$ und $\leftrightarrows = \leftrightarrows_b$ falls τ nicht vorkommt (z.B. in ACP).

Aufgabe 5.41 Bestimmen Sie für jedes der folgenden Paare von Prozesstermen, 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 $a(b+\tau c)$.

geschützte lineare Rekursion

Alle Prozessterme τs stellen eine Lösung für die Spezifikation $X = \tau X$ dar, da $\tau s \leftrightarrows_{rb} \tau\tau s$. Also ist $X = \tau X$ *ungeschützt.*

Definition 5.42 1. *Eine rekursive Spezifikation ist* linear, *wenn ihre rekursiven Gleichungen die folgende Form haben:*

$$X = a_1 X_1 + \cdots + a_k X_k + b_1 + \cdots + b_\ell \qquad (a_i, b_j \in A \cup \{\tau\})$$

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.

Satz 5.43 *Initiale Verzweigungs-Bisimulation ist eine Kongrunzrelation für* ACP_τ *und geschützter linearer Rekursion.*

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 $v \in A' = A \cup \{\tau\}, I \subseteq A$)

B1	$v \cdot \tau$	$=$	v	
B2	$v \cdot (\tau \cdot (x + y) + x)$	$=$	$v \cdot (x + y)$	
TI1	$\tau_I(v)$	$=$	v	$(v \notin I)$
TI2	$\tau_I(v)$	$=$	τ	$(v \in I)$
TI3	$\tau_I(\delta)$	$=$	δ	
TI4	$\tau_I(x + y)$	$=$	$\tau_I(x) + \tau_I(y)$	
TI5	$\tau_I(x \cdot y)$	$=$	$\tau_I(x) \cdot \tau_I(y)$	

Theorem 5.44 *ACP_τ ist korrekt in Bezug auf initiale Verzweigungsbisimulation.*

Aufgabe 5.45 Leiten sie $\tau_{\{b\}}(\langle X|X = aY, Y = bX\rangle) = \langle Z|Z = aZ\rangle$ in ACP_τ ab.

Beispiel 5.46 (Tandempuffer)

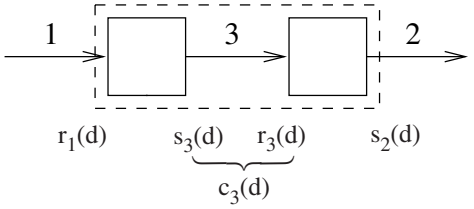
Um Ausdrücke zu vereinfachen, wird in den folgenden Beispielrechnungen eine Lösung $\langle Q|Q = r_1 s_1 Q\rangle$ einer rekursiven Spezifikation abkürzend als Q geschrieben.

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:

$$\begin{aligned} Q_1 &= \Sigma_{d\in\Delta} r_1\, s_3\, Q_1 \\ Q_2 &= \Sigma_{d\in\Delta} r_3\, s_2\, Q_2 \end{aligned}$$

wobei $d \in \Delta$ weggelassen wird, d.h. wir tun so, als ob Δ nur ein Element enthalten würde:

$$\begin{aligned} Q_1 &= r_1\, s_3\, Q_1 \\ Q_2 &= r_3\, s_2\, Q_2 \end{aligned}$$



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:

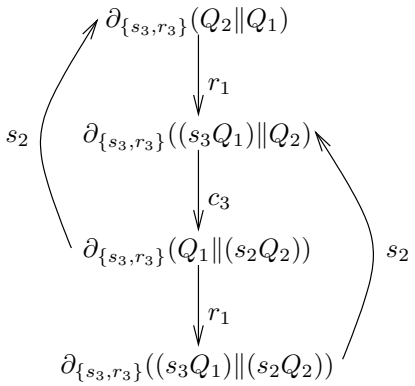
$$\begin{aligned} & Q_2 \| Q_1 \\ \overset{\text{M1}}{=} \; & Q_2 \between Q_1 + Q_1 \between Q_2 + Q_2 | Q_1 \\ \overset{\text{RDP}}{=} \; & (r_3 s_2 Q_2) \between Q_1 + (r_1 s_3 Q_1) \between Q_2 \\ & + (r_3 s_2 Q_2) | (r_1 s_3 Q_1) \\ \overset{\text{3,CM8}}{=} \; & r_3 \cdot ((s_2 Q_2) \| Q_1) + r_1 \cdot ((s_3 Q_1) \| Q_2) \\ & + \delta \cdot ((s_2 Q_2) \| (s_3 Q_1)) \\ \overset{\text{A7,A6}}{=} \; & r_3 \cdot ((s_2 Q_2) \| Q_1) + r_1 \cdot ((s_3 Q_1) \| Q_2) \end{aligned}$$

$$\begin{aligned} & \partial_{\{s_3,r_3\}}(Q_2 \| Q_1) \\ = \; & \partial_{\{s_3,r_3\}}(r_3 \cdot ((s_2 Q_2) \| Q_1) + r_1 \cdot ((s_3 Q_1) \| Q_2)) \\ = \; & \partial_{\{s_3,r_3\}}(r_3 \cdot ((s_2 Q_2) \| Q_1)) \\ & + \partial_{\{s_3,r_3\}}(r_1 \cdot ((s_3 Q_1) \| Q_2)) \\ = \; & \partial_{\{s_3,r_3\}}(r_3) \cdot \partial_{\{s_3,r_3\}}((s_2 Q_2) \| Q_1) \\ & + \partial_{\{s_3,r_3\}}(r_1) \cdot \partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2) \\ = \; & \delta \cdot \partial_{\{s_3,r_3\}}((s_2 Q_2) \| Q_1) \\ & + r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2) \\ = \; & r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2) \end{aligned}$$

Weiter rechnen wir:

$$\begin{aligned} \partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2) &= c_3 \cdot \partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2)) \\ \partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2)) &= r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 Q_1) \| (s_2 Q_2)) \\ &\quad + s_2 \cdot \partial_{\{s_3,r_3\}}(Q_2 \| Q_1) \\ \partial_{\{s_3,r_3\}}((s_3 Q_1) \| (s_2 Q_2)) &= s_2 \cdot \partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2) \end{aligned}$$

Wir fassen die gerechneten Transitionsübergänge zusammen:



Die Axiome für die τ-Aktion und Abstraktion ergeben:

$$\begin{aligned}
& \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
= \;& \tau_{\{c_3\}}(r_1 \cdot \partial_{\{s_3,r_3\}}((s_3Q_1) \| Q_2)) \\
= \;& r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3Q_1) \| Q_2)) \\
= \;& r_1 \cdot \tau_{\{c_3\}}(c_3 \cdot \partial_{\{s_3,r_3\}}(Q_1 \| (s_2Q_2))) \\
= \;& r_1 \cdot \tau \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2Q_2))) \\
= \;& r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2Q_2)))
\end{aligned}$$

Weiter rechnen wir:

$$\begin{aligned}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2Q_2))) \;=\; & \\
& r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3Q_1) \| (s_2Q_2))) \\
& + s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3Q_1) \| (s_2Q_2))) \;=\; & \\
& s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2Q_2)))
\end{aligned}$$

Damit erhalten wir als Lösung für die lineare rekursive Spezifikation E für einen Puffer der Kapazität 2:

$$\begin{aligned}
X &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
Y &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2Q_2))) \\
Z &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3Q_1) \| (s_2Q_2)))
\end{aligned}$$

Die Fairness-Regel

Es ist möglich, durch Abstraktion τ-Schleifen zu konstruieren: $\tau_{\{a\}}(\langle X \mid X{=}aX \rangle)$ führt unendlich lange die τ-Aktion aus.

Definition 5.47 *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 $\langle X|E \rangle \stackrel{b_1}{\rightarrow} \cdots \stackrel{b_m}{\rightarrow} \langle Y|E \rangle$ und $\langle Y|E \rangle \stackrel{c_1}{\rightarrow} \cdots \stackrel{c_n}{\rightarrow} \langle X|E \rangle$ für Aktionen $b_1, \cdots b_m, c_1 \cdots, c_n \in I \cup \{\tau\}$ 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 $\{v_1Y_1, \ldots, v_mY_m, w_1, \ldots, w_n\}$ ist, dann gilt

$$\begin{array}{l} \tau \cdot \tau_I(\langle X|E\rangle) \; = \\ \tau \cdot \tau_I(v_1\langle Y_1|E\rangle + \cdots + v_m\langle Y_m|E\rangle + w_1 + \cdots + w_n) \end{array}$$

Zur Erläuterung von CFAR sei E das Gleichungssystem:

$$\begin{array}{lcl} X_1 & = & aX_2 + b_1 \\ & \vdots & \\ X_{n-1} & = & aX_n + b_{n-1} \\ X_n & = & aX_1 + b_n \end{array}$$

$\tau_{\{a\}}(\langle X_1|E\rangle)$ führt τ-Transitionen aus, bis eine Aktion b_i für $i \in \{1, \ldots, n\}$ 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:

$$\tau_{\{a\}}(\langle X_1|E\rangle) \leftrightarrow_{rb} b_1 + \tau(b_1 + \cdots + b_n)$$

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 5.4 die mit f bezeichneten Transitionen b_1, b_2 und b_3 fair schalten (Definition 3.21 b)), d.h. der Zyklus der mit a bezeichneten Transitionen einmal verlassen wird.

Beispiel 5.48

$$X \; = \; \mathit{heads} \cdot X + \mathit{tails}$$

$\langle X|E\rangle$ stellt das Werfen einer idealen Münze dar, das mit *tails* endet. Von den Ergebnissen „Kopf“ wird abstrahiert: (head) $\tau_{\{heads\}}(\langle X|E\rangle)$.

$\{X\}$ ist die einzige Gruppe für $\{heads\}$ mit dem einzigen Ausgang *tails*. Daher erhält man mit der Regel CFAR:

$$\begin{array}{lcl} \tau \cdot \tau_{\{heads\}}(\langle X|E\rangle) & = & \tau \cdot \tau_{\{heads\}}(tails) \\ & = & \tau \cdot tails \end{array}$$

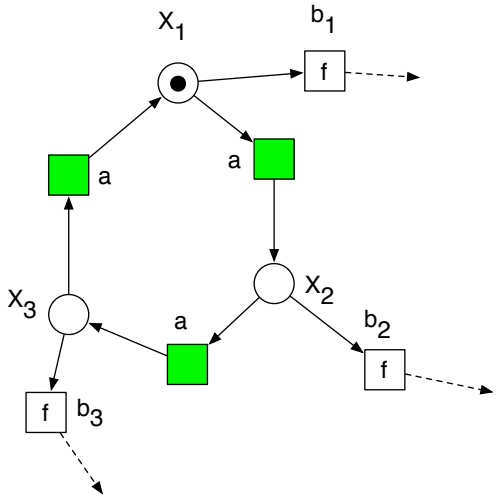


Abbildung 5.4: Faire Abstraktion als P/T-Netz mit fair schaltenden Transitionen.

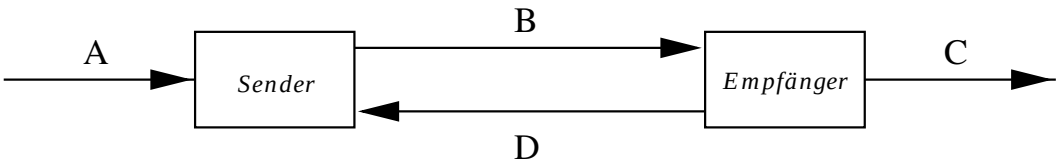
und

$$\begin{aligned}\tau_{\{heads\}}(\langle X|E\rangle) &= \tau_{\{heads\}}(heads \cdot \langle X|E\rangle + tails)\\ &= \tau \cdot \tau_{\{heads\}}(\langle X|E\rangle) + tails\\ &= \tau \cdot tails + tails\end{aligned}$$

Anmerkung: Der Kalkül ACP_τ mit geschützter linearer Rekursion ist korrekt und vollständig in Bezug auf initiale Verzweigungsbisimulation:

$$s = t \;\Leftrightarrow\; s \leftrightarrow_{rb} t$$

5.6 Verifikation des Alternierbitprotokolls



Das Alternierbitprotokoll wurde in Kapitel 1 (Abschnitt 1.13 auf Seite 13) als Transitionssystem, d.h. über den Zustandsgraphen eingeführt. Im Abschnitt 7.4.1 auf Seite 237 erfolgte dann eine Entwicklung als Petrinetz, wodurch die i.A. sehr viel kleinere Programmstruktur nachgeliefert wurde. Hier wird nun mit den Mitteln der Prozessalgebra ein Korrektheitsbeweis geführt. Dies verbindet beide Ansätze. 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 Verhaltensgleichheit (Bisimilarität) der entsprechenden Prozessgraphen, also der Transitionssysteme. Deren Zustandsexplosion 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.

Spezifikation des Senders für das Senden mit Bit b:

$$\begin{array}{lcl} S_b & = & \sum\limits_{d\in\Delta} r_{\mathrm{A}}(d)\cdot T_{db} \\ T_{db} & = & (s_{\mathrm{B}}(d,b)+s_{\mathrm{B}}(\bot))\cdot U_{db} \\ U_{db} & = & r_{\mathrm{D}}(b)\cdot S_{1-b} \;+\; (r_{\mathrm{D}}(1-b)+r_{\mathrm{D}}(\bot))\cdot T_{db} \end{array}$$

Für ein Argument u bedeutet $r_{\mathrm{X}}(u)$ bzw. $s_{\mathrm{X}}(u)$ wieder das Lesen von dem bzw. das Schreiben in den Kanal X.

Spezifikation des Empfängers für das Empfangen mit Bit b:

$$\begin{array}{lcl} R_b & = & \sum\limits_{d\in\Delta}\{r_{\mathrm{B}}(d,b)\cdot s_{\mathrm{C}}(d)\cdot Q_b \\ & & \quad + r_{\mathrm{B}}(d,1-b)\cdot Q_{1-b}\} + r_{\mathrm{B}}(\bot)\cdot Q_{1-b} \\ Q_b & = & (s_{\mathrm{D}}(b)+s_{\mathrm{D}}(\bot))\cdot R_{1-b} \end{array}$$

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

- $\gamma(s_B(d,b), r_B(d,b)) := c_B(d,b)$
- $\gamma(s_B(\bot), r_B(\bot)) := c_B(\bot)$
- $\gamma(s_D(b), r_D(b)) := c_D(b)$
- $\gamma(s_D(\bot), r_D(\bot)) := c_D(\bot)$

für $d \in \Delta, b \in \{0,1\}$. Dabei ist $\bot$ 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 $I := \{c_B(d,b), c_D(b) | d \in \Delta, b \in \{0,1\}\} \cup \{c_B(\bot), c_D(\bot)\}$.

Als Korrektheitsbeweis werden wir ableiten:

$$\tau_I(\partial_H(R_0 \| S_0)) = \sum_{d \in \Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot \tau_I(\partial_H(R_0 \| S_0))$$

Das spezifizierte Protokoll hat damit also das gewünschte Verhalten:

$$r_{\mathrm{A}}(d_0), s_{\mathrm{C}}(d_0), r_{\mathrm{A}}(d_1), s_{\mathrm{C}}(d_1), r_{\mathrm{A}}(d_2), s_{\mathrm{C}}(d_2), \ldots$$

d.h. alle Datenelemente $d_0, d_1, d_2, \ldots$ 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:

$$\begin{aligned} R_0 \| S_0 \;=\; & \textstyle\sum_{d' \in \Delta} \{ r_{\mathrm{B}}(d', 0) \cdot ((s_{\mathrm{C}}(d') Q_0) \| S_0) \\ & + r_{\mathrm{B}}(d', 1) \cdot (Q_1 \| S_0) \} + r_{\mathrm{B}}(\bot) \cdot (Q_1 \| S_0) \\ & + \textstyle\sum_{d \in \Delta} r_{\mathrm{A}}(d) \cdot (T_{d0} \| R_0) \end{aligned}$$

Weiter erhält man mit D4, D1, D2, D5, A6, A7:

$$\partial_H(R_0 \| S_0) \;=\; \textstyle\sum_{d \in \Delta} r_{\mathrm{A}}(d) \cdot \partial_H(T_{d0} \| R_0)$$

Diese Äquivalenz entspricht dem Übergang vom Zustand 1 in den Zustand 2 im Transitionssystem von $\partial_H(R_0 \| S_0)$ in Abb. 5.5.

$$\begin{aligned} T_{d0} \| R_0 \;=\; & (s_{\mathrm{B}}(d,0) + s_{\mathrm{B}}(\bot)) \cdot (U_{d0} \| R_0) \\ & + \textstyle\sum_{d' \in \Delta} \{ r_{\mathrm{B}}(d', 0) \cdot ((s_{\mathrm{C}}(d') Q_0) \| T_{d0}) \\ & + r_{\mathrm{B}}(d', 1) \cdot (Q_1 \| T_{d0}) \} + r_{\mathrm{B}}(\bot) \cdot (Q_1 \| T_{d0}) \\ & + c_{\mathrm{B}}(d,0) \cdot (U_{d0} \| (s_{\mathrm{C}}(d) Q_0)) + c_{\mathrm{B}}(\bot) \cdot (U_{d0} \| Q_1) \\ \partial_H(T_{d0} \| R_0) \;=\; & c_{\mathrm{B}}(d,0) \cdot \partial_H(U_{d0} \| (s_{\mathrm{C}}(d) Q_0)) \\ & + c_{\mathrm{B}}(\bot) \cdot \partial_H(U_{d0} \| Q_1) \end{aligned}$$

Diese Äquivalenz entspricht dem Übergang vom Zustand 2 in die Zustände 3 und 4 im Transitionsgraph von Abb. 5.5.

Entsprechend erhält man für die Übergänge bis zum Zustand 7:

$$\begin{aligned} U_{d0} \| Q_1 \;=\; & r_{\mathrm{D}}(0) \cdot (S_1 \| Q_1) \\ & + (r_{\mathrm{D}}(1) + r_{\mathrm{D}}(\bot)) \cdot (T_{d0} \| Q_1) \\ & + (s_{\mathrm{D}}(1) + s_{\mathrm{D}}(\bot)) \cdot (R_0 \| U_{d0}) \\ & + (c_{\mathrm{D}}(1) + c_{\mathrm{D}}(\bot)) \cdot (T_{d0} \| R_0) \\ \partial_H(U_{d0} \| Q_1) \;=\; & (c_{\mathrm{D}}(1) + c_{\mathrm{D}}(\bot)) \cdot \partial_H(T_{d0} \| R_0) \end{aligned}$$

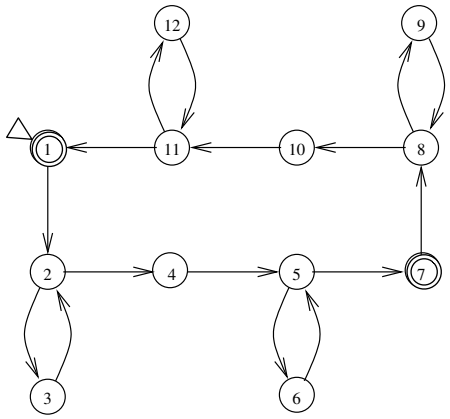


Abbildung 5.5: Transitionssystem von $\partial_H(R_0 \| S_0)$

$$
\begin{aligned}
U_{d0}\|(s_{\mathrm{C}}(d)Q_0) &= r_{\mathrm{D}}(0)\cdot(S_1\|(s_{\mathrm{C}}(d)Q_0)) \\
&\quad + (r_{\mathrm{D}}(1)+r_{\mathrm{D}}(\bot))\cdot(T_{d0}\|(s_{\mathrm{C}}(d)Q_0)) \\
&\quad + s_{\mathrm{C}}(d)\cdot(Q_0\|U_{d0}) \\
\partial_H(U_{d0}\|(s_{\mathrm{C}}(d)Q_0)) &= s_{\mathrm{C}}(d)\cdot\partial_H(Q_0\|U_{d0})
\end{aligned}
$$

$$
\begin{aligned}
Q_0\|U_{d0} &= (s_{\mathrm{D}}(0)+s_{\mathrm{D}}(\bot))\cdot(R_1\|U_{d0}) \\
&\quad + r_{\mathrm{D}}(0)\cdot(S_1\|Q_0)+(r_{\mathrm{D}}(1)+r_{\mathrm{D}}(\bot))\cdot(T_{d0}\|Q_0) \\
&\quad + c_{\mathrm{D}}(0)\cdot(R_1\|S_1)+c_{\mathrm{D}}(\bot)\cdot(R_1\|T_{d0}) \\
\partial_H(Q_0\|U_{d0}) &= c_{\mathrm{D}}(0)\cdot\partial_H(R_1\|S_1) \\
&\quad + c_{\mathrm{D}}(\bot)\cdot\partial_H(R_1\|T_{d0})
\end{aligned}
$$

$$
\begin{aligned}
R_1\|T_{d0} &= \sum_{d'\in\Delta}\{r_{\mathrm{B}}(d',1)\cdot((s_{\mathrm{C}}(d')Q_1)\|T_{d0}) \\
&\quad + r_{\mathrm{B}}(d',0)\cdot(Q_0\|T_{d0})\}+r_{\mathrm{B}}(\bot)\cdot(Q_0\|T_{d0}) \\
&\quad + (s_{\mathrm{B}}(d,0)+s_{\mathrm{B}}(\bot))\cdot(U_{d0}\|R_1) \\
&\quad + (c_{\mathrm{B}}(d,0)+c_{\mathrm{B}}(\bot))\cdot(Q_0\|U_{d0}) \\
\partial_H(R_1\|T_{d0}) &= (c_{\mathrm{B}}(d,0)+c_{\mathrm{B}}(\bot))\cdot\partial_H(Q_0\|U_{d0})
\end{aligned}
$$

Dann erhält man für die Übergänge von Zustand 7 bis zum Zustand 1:

$$
\begin{aligned}
\partial_H(R_1\|S_1) &= \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot\partial_H(T_{d1}\|R_1)\\
\partial_H(T_{d1}\|R_1) &= c_{\mathrm{B}}(d,1)\cdot\partial_H(U_{d1}\|(s_{\mathrm{C}}(d)\cdot Q_1))\\
&\quad + c_{\mathrm{B}}(\bot)\cdot\partial_H(U_{d1}\|Q_0)\\
\partial_H(U_{d1}\|Q_0) &= (c_{\mathrm{D}}(0)+c_{\mathrm{D}}(\bot))\cdot\partial_H(T_{d1}\|R_1)\\
\partial_H(U_{d1}\|(s_{\mathrm{C}}(d)Q_1)) &= s_{\mathrm{C}}(d)\cdot\partial_H(Q_1\|U_{d1})\\
\partial_H(Q_1\|U_{d1}) &= c_{\mathrm{D}}(1)\cdot\partial_H(R_0\|S_0)\\
&\quad + c_{\mathrm{D}}(\bot)\cdot\partial_H(R_0\|T_{d1})\\
\partial_H(R_0\|T_{d1}) &= (c_{\mathrm{B}}(d,1)+c_{\mathrm{B}}(\bot))\cdot\partial_H(Q_1\|U_{d1})
\end{aligned}
$$

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.

$$
\begin{aligned}
\{\quad X_1 &= \textstyle\sum_{d'\in\Delta} r_{\mathrm{A}}(d')\cdot X_{2d'}\\
X_{2d} &= c_{\mathrm{B}}(d,0)\cdot X_{4d} + c_{\mathrm{B}}(\bot)\cdot X_{3d}\\
X_{3d} &= (c_{\mathrm{D}}(1)+c_{\mathrm{D}}(\bot))\cdot X_{2d}\\
X_{4d} &= s_{\mathrm{C}}(d)\cdot X_{5d}\\
X_{5d} &= c_{\mathrm{D}}(0)\cdot Y_1 + c_{\mathrm{D}}(\bot)\cdot X_{6d}\\
X_{6d} &= (c_{\mathrm{B}}(d,0)+c_{\mathrm{B}}(\bot))\cdot X_{5d}\\
Y_1 &= \textstyle\sum_{d'\in\Delta} r_{\mathrm{A}}(d')\cdot Y_{2d'}\\
Y_{2d} &= c_{\mathrm{B}}(d,1)\cdot Y_{4d} + c_{\mathrm{B}}(\bot)\cdot Y_{3d}\\
Y_{3d} &= (c_{\mathrm{D}}(0)+c_{\mathrm{D}}(\bot))\cdot Y_{2d}\\
Y_{4d} &= s_{\mathrm{C}}(d)\cdot Y_{5d}\\
Y_{5d} &= c_{\mathrm{D}}(1)\cdot X_1 + c_{\mathrm{D}}(\bot)\cdot Y_{6d}\\
Y_{6d} &= (c_{\mathrm{B}}(d,1)+c_{\mathrm{B}}(\bot))\cdot Y_{5d}\\
|\quad d\in\Delta\quad\}
\end{aligned}
$$

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 $c_{\mathrm{B}}(d,0)\cdot X_{4d}$, also:

$$
\begin{aligned}
& r_{\mathrm{A}}(d)\cdot\tau_I(\langle X_{2d}|E\rangle)\\
=\ & r_{\mathrm{A}}(d)\cdot\tau_I(c_{\mathrm{B}}(d,0)\cdot\langle X_{4d}|E\rangle)\\
=\ & r_{\mathrm{A}}(d)\cdot\tau\cdot\tau_I(\langle X_{4d}|E\rangle)\\
=\ & r_{\mathrm{A}}(d)\cdot\tau_I(\langle X_{4d}|E\rangle)
\end{aligned}
$$

Entsprechend:

$$\begin{array}{rcl} s_{\mathrm{C}}(d) \cdot \tau_I(\langle X_{5d}|E\rangle) & = & s_{\mathrm{C}}(d) \cdot \tau_I(\langle Y_1|E\rangle) \\ r_{\mathrm{A}}(d) \cdot \tau_I(\langle Y_{2d}|E\rangle) & = & r_{\mathrm{A}}(d) \cdot \tau_I(\langle Y_{4d}|E\rangle) \\ s_{\mathrm{C}}(d) \cdot \tau_I(\langle Y_{5d}|E\rangle) & = & s_{\mathrm{C}}(d) \cdot \tau_I(\langle X_1|E\rangle) \end{array}$$

$$\begin{array}{rcl} \tau_I(\langle X_1|E\rangle) & = & \sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot \tau_I(\langle X_{2d}|E\rangle) \\ & = & \sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot \tau_I(\langle X_{4d}|E\rangle) \\ & = & \sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot \tau_I(\langle X_{5d}|E\rangle) \\ & = & \sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot \tau_I(\langle Y_1|E\rangle) \end{array}$$

$$\tau_I(\langle Y_1|E\rangle) \;=\; \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot \tau_I(\langle X_1|E\rangle)$$

Mit RSP folgt $\tau_I(\langle X_1|E\rangle) \;=\; \langle Z \mid Z = r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot Z\rangle$.

Damit ist das oben angesprochene Ziel des Beweises erreicht:

$$\tau_I(\partial_H(R_0 \| S_0)) = \sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot \tau_I(\partial_H(R_0 \| S_0))$$

Aufgabe 5.49

a) Ersetzen Sie im Beweis des Alternierbitprotokolls die Spezifikation von U_{db} durch

$$U_{db} = (r_D(b) + r_D(\bot)) \cdot S_{1-b} + r_D(1-b) \cdot T_{db}.$$

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.

5.7 Anhang: Übersicht über die Axiome der Prozesskalküle

Zur Übersicht alle Axiome und daraus abeleitete Regeln mit Seitenreferenz[4]. Wie üblich sei $v \in A$ und x, y, z seien Terme.

Bez.	Axiom			BPA	PAP	ACP	Rek.	ACP$_\tau$	Skript
A1	$x + y$	$=$	$y + x$	x	x	x		x	S. 161
A2	$(x + y) + z$	$=$	$x + (y + z)$	x	x	x		x	S. 161
A3	$x + x$	$=$	x	x	x	x		x	S. 161
A4	$(x + y) \cdot z$	$=$	$x \cdot z + y \cdot z$	x	x	x		x	S. 161
A5	$(x \cdot y) \cdot z$	$=$	$x(y \cdot z)$	x	x	x		x	S. 161
R1	$x + y$	$=_{AC}$	$y + x$						S. 163
R2	$(x + y) + z$	$=_{AC}$	$x + (y + z)$						S. 163
R3	$x + x$	$\rightarrow$	x						S. 163
R4	$(x + y) \cdot z$	$\rightarrow$	$x \cdot z + y \cdot z$						S. 163
R5	$(x \cdot y) \cdot z$	$\rightarrow$	$x \cdot (y \cdot z)$						S. 163
M1	$x \parallel y$	$=$	$(x \between y + y \between x) + x \mid y$		x	x		x	S. 167
LM2	$v \between y$	$=$	$v \cdot y$		x	x		x	S. 167
LM3	$(v \cdot x) \between y$	$=$	$v \cdot (x \parallel y)$		x	x		x	S. 167
LM4	$(x + y) \between z$	$=$	$x \between z + y \between z$		x	x		x	S. 167
CM5	$v \mid w$	$=$	$\gamma(v, w)$		x	x		x	S. 167
CM6	$v \mid (w \cdot y)$	$=$	$\gamma(v, w) \cdot y$		x	x		x	S. 167
CM7	$(v \cdot x) \mid w$	$=$	$\gamma(v, w) \cdot x$		x	x		x	S. 167
CM8	$(v \cdot x) \mid (w \cdot y)$	$=$	$\gamma(v, w)(x \parallel y)$		x	x		x	S. 167
CM9	$(x + y) \mid z$	$=$	$x \mid z + y \mid z$		x	x		x	S. 167
CM10	$x \mid (y + z)$	$=$	$x \mid y + x \mid z$		x	x		x	S. 167
A6	$x + \delta$	$=$	x			x		x	S. 168
A7	δx	$=$	δ			x		x	S. 168
D1	$\partial_H(v)$	$=$	$v, v \notin H$			x		x	S. 168
D2	$\partial_H(v)$	$=$	$\delta, v \in H$			x		x	S. 168
D3	$\partial_H(\delta)$	$=$	δ			x		x	S. 168
D4	$\partial_H(x + y)$	$=$	$\partial_H(x) + \partial_H(y)$			x		x	S. 168
D5	$\partial_H(x \cdot y)$	$=$	$\partial_H(x) \cdot \partial_H(y)$			x		x	S. 168
LM11	$\delta \between x$	$=$	δ			x		x	S. 168
CM12	$\delta \parallel x$	$=$	δ			x		x	S. 168
CM13	$x \mid \delta$	$=$	δ			x		x	S. 168
RDP	$\langle X_i \mid E \rangle$	$=$	$t_i(\langle X_1 \mid E \rangle, ..., \langle X_n \mid E \rangle)$				x	x	S. 175
RSP	$\langle X_i \mid E \rangle$	$=$	y_i, falls für alle i gilt: $y_i = t_i(y_1, ..., y_n)$				x	x	S. 175
B1	$v \cdot \tau$	$=$	v					x	S. 181
B2	$v \cdot (x + y)$	$=$	$v \cdot (\tau \cdot (x + y) + x)$					x	S. 181
TI1	$\tau_I(v)$	$=$	$v, v \notin I$					x	S. 181
TI2	$\tau_I(v)$	$=$	$\tau, v \in I$					x	S. 181
TI3	$\tau_I(\delta)$	$=$	δ					x	S. 181
TI4	$\tau_I(x + y)$	$=$	$\tau_I(x) + \tau_I(y)$					x	S. 181
TI5	$\tau_I(x \cdot y)$	$=$	$\tau_I(x) \cdot \tau_I(y)$					x	S. 181

[4]Dank an Patrick Fey

Kapitel 6

Konsistenz

In diesem Kapitel werden zwei Klassen von Korrektheits- bzw. Konsistenz-Begriffen behandelt: Ablaufkonsistenz, die sich auf den korrekten Steuerfluss nebenläufiger Systeme bezieht und Datenkonsistenz, die die Integrität von Daten bei nebenläufigem Zugriff betrifft. Die Themen werden jedoch anhand von typischen Fällen behandelt: Ablaufkonsistenz durch das Modell der Workflownetze und Datenkonsistenz durch das Modell der schematischen Auftragssysteme. Weitere Themen zur Ablaufkonsistenz wie Verklemmungsfreihet, Lebendigkeit und Fairness wurden schon früher in dieser Vorlesung behandelt. Datenkonsistenz wird meist auch durch Synchronisationsmechanismen gewährleistet, was zeigt, dass die beiden Gebiete korrelieren.

6.1 Ablaufkonsistenz: Workflow

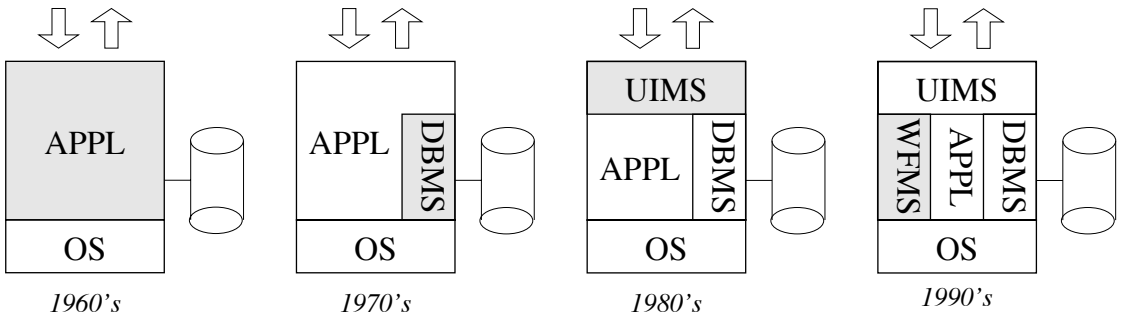


Abbildung 6.1: Workflow-Management-Systeme in historischer Perspektive

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. 6.1). 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, UIMSs) erlaubten es, die Kommunikation mit dem Benutzer aus dem Anwendungsprogramm auszulagern. Eine ähnliche Entwicklung setzte in den 90-er Jahren mit der Entwicklung von Workflow-Programmen (workflow management systems, WFMSs) ein. Sie erlauben es, die Verwaltung von Geschäftsprozessen aus spezifischen Anwendungsprogrammen herauszuhalten [Aal00], [Aal98], [AH02].

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 6.2 setzt allgemeine kooperative Prozesse und Workflow-Systeme in eine Skalierung zwischen informationszentriert und prozesszentriert bzw. weniger und mehr strukturiert.

Für Workflow-Systeme gibt es hunderte von Modellierungsansätze und Rechnerwerkzeuge, die jedoch häufig keine wohldefinierte und eindeutige Semantik besitzen. Wegen ihrer Orientierung auf Aktionen und deren Koordination bieten sich natürlich besonders Petrinetze an, deren Semantik wohldefiniert und bekannt ist. Ein solcher Ansatz nach [Aal00], [Aal98] und [AH02] wird hier vorgestellt.

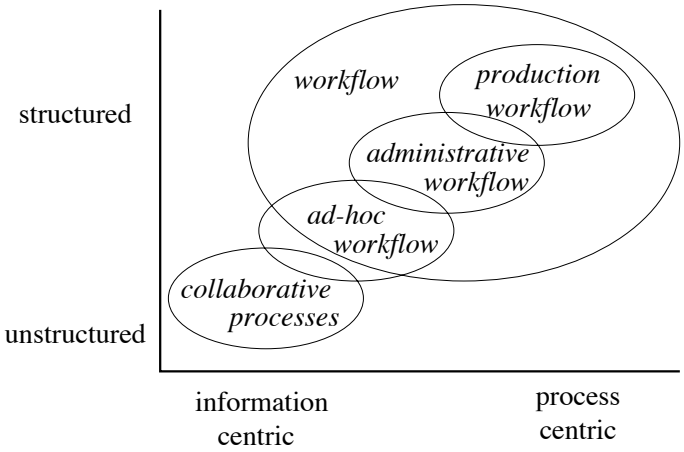


Abbildung 6.2: Workflow Prozesse und kooperative Prozesse im Vergleich

Im Folgenden betrachten wir ein Workflowsystem zur Beschwerdebearbeitung (siehe Abbildung. 6.3). Es enthält folgende Aktionen:
1. Aufnahme einer Beschwerde (register)
2. Fragebogen an Beschwerdeführer (send_questionnaire)
3. Bewertung (evaluate) (nebenläufig zu 2.)
4. Fragebogenauswertung (process_questonaire), falls Rücklauf innerhalb von 2 Wochen, sonst:

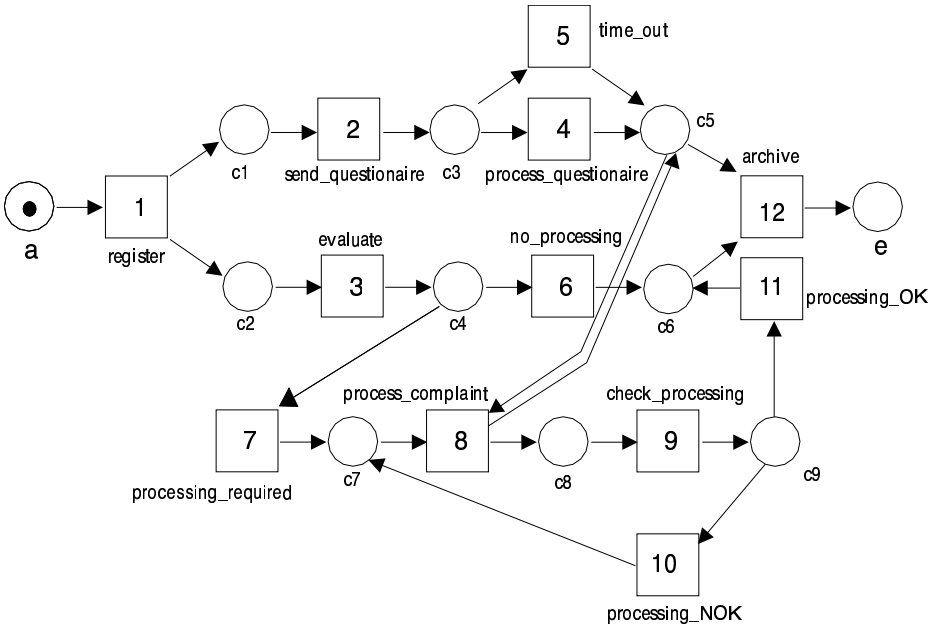


Abbildung 6.3: Ein Petrinetz für einen Vorgang zur Beschwerdebearbeitung

5. Nichtberücksichtigung des Fragebogens (time_out).
6. Je nach Ergebnis der Bewertung (3.) : Aussetzung der Bearbeitung (no_processing) oder
7. Beginn der eigentlichen Prüfung (processing_required)
8. Bearbeitung der Beschwerde (process_complaint) unter Berücksichtigung des Fragebogens[1]
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“. Es ist sinnvoll nur einen Anfangsplatz a und einen Endplatz e vorzusehen, da Anfang und Ende meist hervorgehobende Ereignisse sind. Dann sollten alle Transitionen auf Pfaden zwischen diesen liegen. Wir erhalten so folgende Normalform:

Definition 6.1 *Ein P/T-Netz* $\mathcal{N} = (P, T, F, \mathbf{m}_a)$[2] *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 Pfaden zwischen a *und* e *liegen und*

c) die Anfangsmarkierung $\mathbf{m}_a$ *durch* $[\mathbf{m}_a(p) :=$ **if** $p = a$ **then** 1 **else** 0 **fi**$]$ *sowie eine* Endmarkierung $\mathbf{m}_e$ *durch :* $[\mathbf{m}_e(p) :=$ **if** $p = e$ **then** 1 **else** 0 **fi**$]$ *festgelegt sind.*

[1] Nebenbedingung c_5

[2] Die Kantenbewertung ist 1 auf F, also: $W(x, y) = 1 \Leftrightarrow (x, y) \in F$.

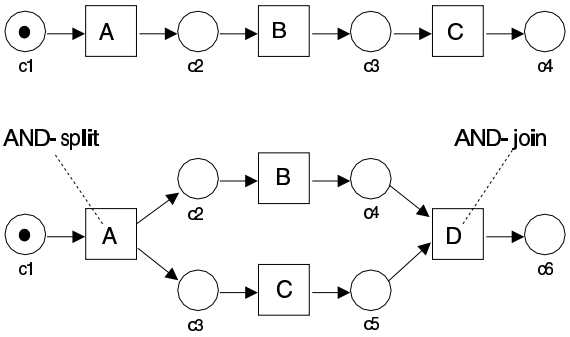


Abbildung 6.4: Sequentielle und nebenläufige Bearbeitung

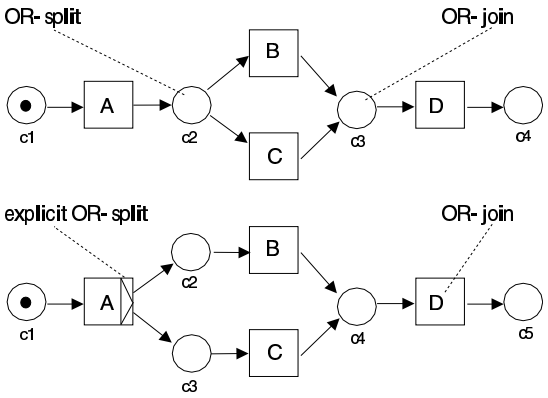


Abbildung 6.5: Alternative Bearbeitung mit und ohne expliziten Testbedingungen

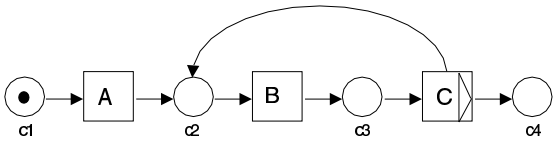


Abbildung 6.6: Iteration

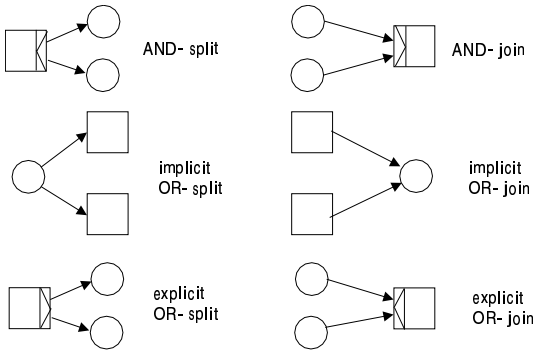


Abbildung 6.7: Graphische Symbole für Verzweigung

Typische Strukturen („patterns[3]“) 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“ .

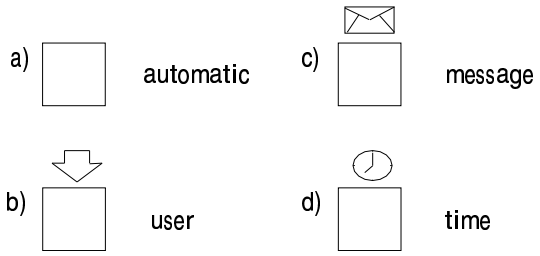


Abbildung 6.8: Trigger eines Workflowsystems

Es gibt folgende Arten von Triggern (Abbildung 6.8):

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 $c3$ der Abbildung 6.10 durch explizite). Ein anderes Problem besteht darin, dass manchmal Zustandsinformation („milestones“) erforderlich ist (Beispiel : $c5$ in Abb. 6.10).

Die Erfahrung aus der Praxis zeigt, dass

- Workflow-Prozesse oft nicht richtig verstanden werden (Mehrdeutigkeit, Widersprüche, Verklemmungen),

[3]Unter `http://www.workflowpatterns.com/patterns` ist eine Sammlung solcher Strukturmuster für Workflows zu finden.

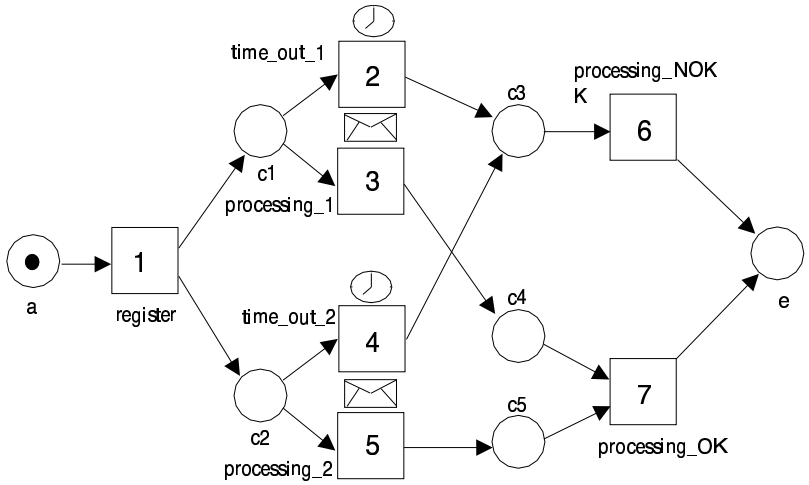


Abbildung 6.9: Problematisches Workflownetz für Beschwerdebearbeitung

- 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.

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. Wir betrachten dazu das nicht korrekte Workflow-System von Abbildung 6.9. 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 !

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 6.2 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 6.2 *Ein WF-Netz* $\mathcal{N} = (P, T, F, \mathbf{m}_a)$ *heißt* korrekt*, falls gilt :*

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

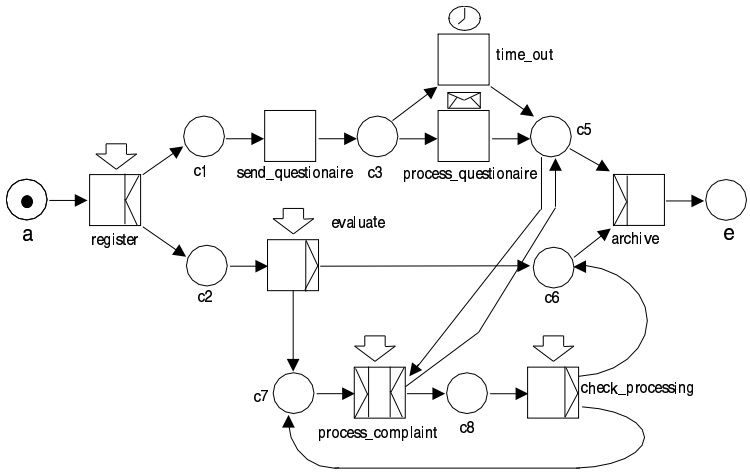


Abbildung 6.10: Workflow aus Abb. 6.3 mit Triggern

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

c) $\forall t \in T \; \exists \mathbf{m} \in \mathbf{R}(\mathcal{N}) : \mathbf{m} \xrightarrow{t}$

Das WF-Netz aus Abbildung 6.9 erfüllt c) aber nicht a) und nicht b).

Die Eigenschaft „korrektes WF-Netz“ ist ein gutes Beispiel dafür, wie eine domänenspezifische 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 3.1 auf Seite 84 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.

Die genannte Transformation des WF-Netzes besteht in der Hinzufügung einer neuen Transition t^* zwischen e und a (Abbildung 6.11):

Definition 6.3 *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) $F' := F \cup \{(e, t^*), (t^*, a)\}$

Der zugehörige Satz lautet:

Satz 6.4 *Ein WF-Netz* $\mathcal{N}$ *ist genau dann korrekt, wenn sein Abschluss* $\overline{\mathcal{N}}$ *lebendig und beschränkt ist.*

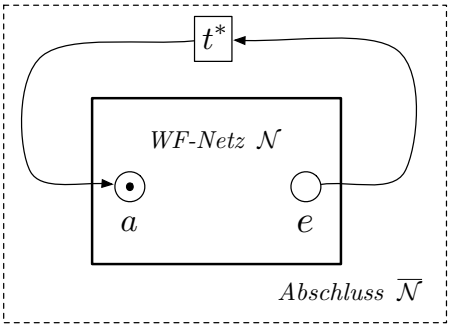


Abbildung 6.11: Transformation eines WF-Netzes

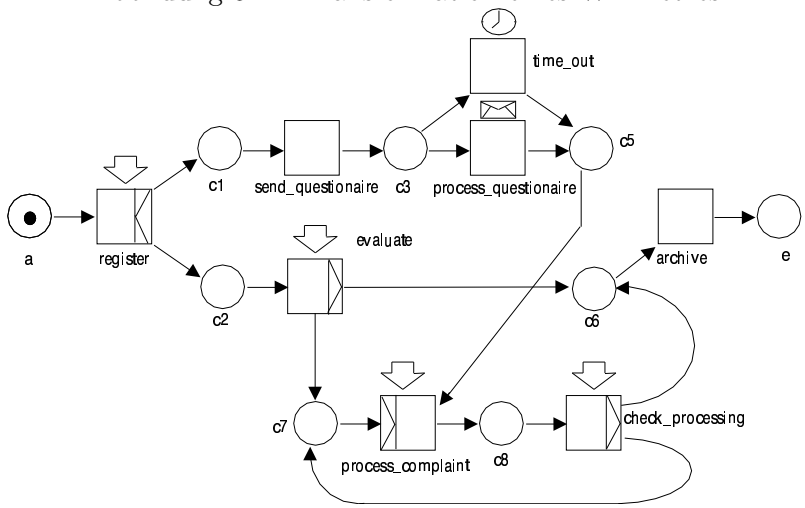


Abbildung 6.12: Ein WF-Netz zur Behandlung von Beschwerden

Der Beweis ist elementar (siehe [Aal97]) und sollte nach der Definition in Tabelle 3.1 behandelt werden.

Aufgabe 6.5 a) Zeigen Sie, dass für das WF-Netz $\mathcal{N}$ aus Abbildung 6.9 der Abschluss $\overline{\mathcal{N}}$ nicht beschränkt ist (z.B. dadurch dass $\mathbf{R}(\overline{\mathcal{N}})$ nicht endlich ist)! Ebenso zeige man (ggf. später), dass $\overline{\mathcal{N}}$ nicht lebendig ist!
b) Analysieren Sie das WF-Netz von Abb. 6.12 mit Hilfe des Verfahrens nach Satz 6.4.

6.2 Datenkonsistenz: Serialisierbarkeit

6.2.1 Schematische Auftragssysteme

Zur Einführung des Konzeptes der *Serialisierbarkeit* betrachten wir einige Beispiele von Programm-Fragmenten. Die Anweisung **cobegin** $P_1||P_2$ **coend** bedeute die nebenläufige Ausführung von P_1 und P_2mit gemeinsamen Daten (vergl. Abschnitt 1.4.4).

Beispiel 6.6 P_1 : $a_0 : z := 1$
cobegin $a_1 : x := z + 1;\ a_2 : z := x \ ||$
$b_1 : y := z + 2;\ b_2 : z := y$ **coend**
P_2 : $a_0 : z := 1;$
cobegin $A : z := z + 1 \ || \ B : z := z + 2$ **coend**

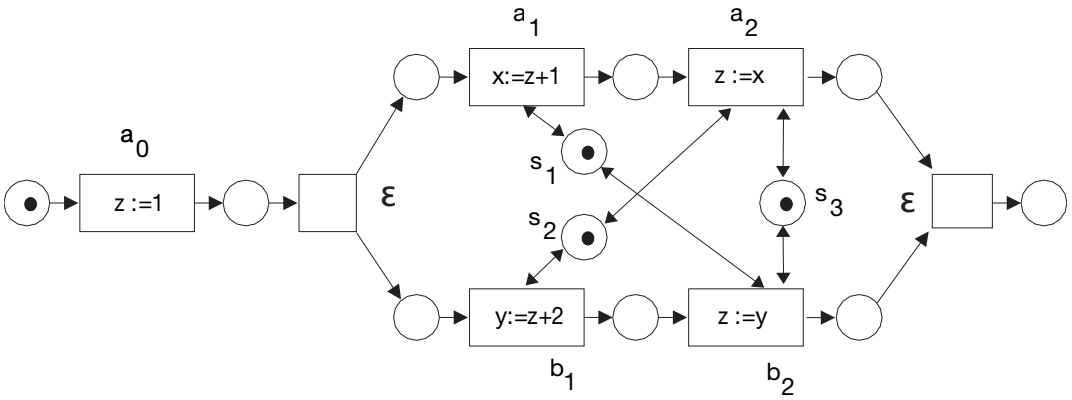


Abbildung 6.13: Netzprogramm zu P_1

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

$$w_1 = a_0 a_1 b_1 a_2 b_2$$

des P/T-Netzes von Abb. 6.13 dargestellt werden (die mit ϵ benannten Hilfstransitionen wurden weggelassen). Für die ebenfalls möglichen Schaltfolgen

$$w_2 = a_0 a_1 a_2 b_1 b_2$$

bzw.

$$w_3 = a_0 a_1 b_1 b_2 a_2$$

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

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

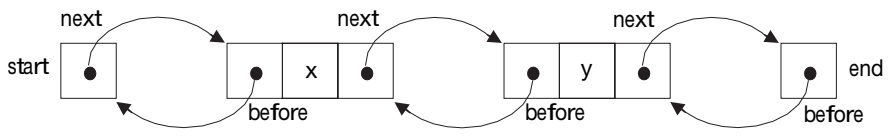


Abbildung 6.14: Doppelt verkettete Liste

Beispiel 6.7 [BR81] Die Anweisung $delete(p)$ soll ein Listenelement p aus einer doppelt verketteten Liste wie Abb. 6.14 entfernen. Dabei seien p_1 und p_2 lokale Variablen der Anweisung.

$$\begin{array}{l} delete(p) \\ \quad \textbf{cobegin } a_1 \ : \ p_1 := p.\text{before} \ || \ b_1 \ : \ p_2 := p.\text{next } \textbf{coend} \ ; \\ \quad \textbf{cobegin } a_2 \ : \ p_1.\text{next} := p_2 \ || \ b_2 \ : \ p_2.\text{before} := p_1 \ \textbf{coend} \end{array} \tag{6.1}$$

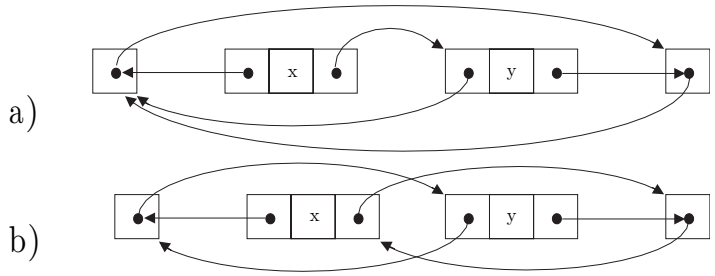


Abbildung 6.15: Ergebnis der Anwendung von 6.2: a) bei serieller Ausführung und b) bei kollateraler Ausführung

Wendet man

$$p_3 : \ \textbf{cobegin } delete(x) \ || \ delete(y) \ \textbf{coend} \tag{6.2}$$

auf die Liste von Abb. 6.14 an, so erhält man bei der seriellen Ausführung von $delete(x)$ und $delete(y)$, wie gewünscht, die leere Liste von Abb. 6.15 a), bei der kollateralen Ausführung von $a_1||b_1$ und $a_2||b_2$ durch die Folge

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

jedoch das Gebilde von Abb. 6.15 b). (Die Exponenten geben an, ob die entsprechende Anweisung zu $delete(x)$ oder $delete(y)$ gehört.)

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

Definition 6.8 *Ein Auftragssystem $AS = (A, \lessdot)$ besteht aus einer endlichen Menge A von* Aufträgen *und einer irreflexiven, transitiven Relation $< := \lessdot^+ \subset A \times A$, wobei $\lessdot$ die direkten Präzedenzen beschreibt und* $<$ Präzendenzrelation *(precedence relation) genannt wird. Für $(a_i, a_j) \in <$ schreiben wir auch $a_i < a_j$ und nennen den Auftrag a_i "präzedent zu" a_j. a_i heißt "direkt präzedent zu" a_j, wenn gilt $a_i < a_j$, jedoch $a_i < a_k < a_j$ für kein $a_k \in A$ gilt.*

Auftragssysteme sind also endliche Striktordnungen (Def. 2.2 b)). Das Beispiel von Abb. 6.16 ist in Abb. 6.17 als Netz mit Anfangsmarkierung dargestellt. Als Ausführungsfolgen eines Auftragssystems betrachten wir jede lineare Vervollständigung (Def. 2.3, Anmerkung e). Diese Ausführungsfolgen entsprechen den maximalen Ausführungsfolgen des entsprechenden Netzes (wie z.B. in Abb. 6.17). Die Menge der Ablauffolgen, die alle Aufträge enthalten, wird mit $F_E(AS)$ bezeichnet, die Menge der Anfangsstücke dieser Folgen mit $F(AS)$.

Im Beispiel der nebenläufigen Listen-Operationen (Beispiel 6.14) war klar, dass die genannte Ausführungsfolge w als inkorrekt zu betrachten ist. Anders in Beispiel 6.6: welches oder welche der drei möglichen Resultate sollen als korrekt gelten?

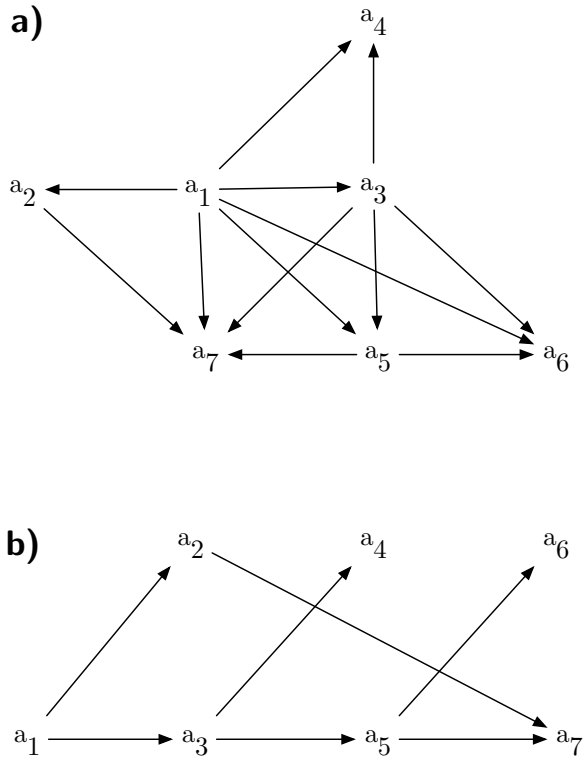


Abbildung 6.16: Präzedenzgraph G_1 in a) und zugehöriger Konnektivitätsgraph G_2 in b)

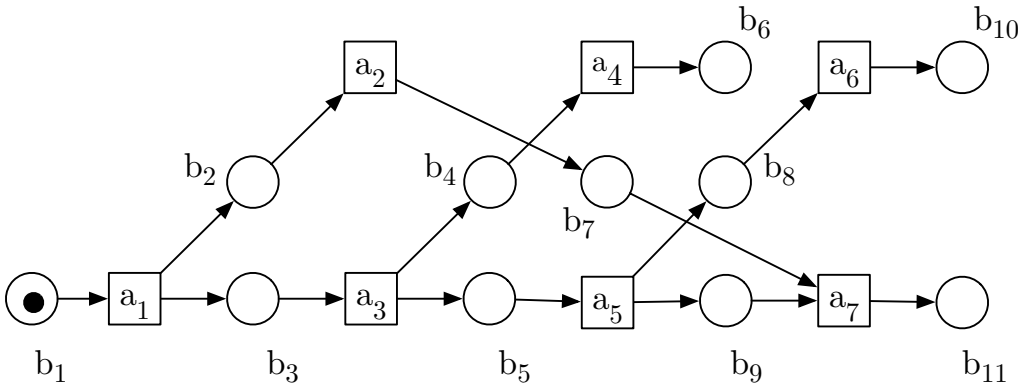


Abbildung 6.17: Auftragssystem als P/T-Netz

Eine ähnliche, jedoch i.a. ungleich komplexere Situation finden wir bei Aufträgen, die auf große Datensätze zugreifen. Ein solcher Auftrag a_i wird auch als *Transaktion* (transaction) bezeichnet. Er zergliedert sich i.a. in mehrere unteilbare Teilaufträge $a_i^1, \ldots, a_i^k$, die sequentiell auszuführen sind. Aus Effektivitätsgründen (z.B. annehmbare Antwortzeit) müssen mehrere solche Aufträge oder Transaktionen $a_1, \ldots, a_n$ nebenläufig ausgeführt werden. Dabei sollen nach der Ausführung gewisse Datenbeziehungen erhalten bleiben (z.B. gleiche Adresse bei nicht getrennt lebenden Eheleuten). Man nennt dies die *Konsistenz* des Datensatzes.

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

Die ungeteilte Ausführung von $< a_1; a_2 >$ und $< b_1; b_2 >$ in Beispiel 6.6 liefert in beiden Serialisierungen $z = 4$. Jede ein anderes Ergebnis liefernde Folge ist daher nicht serialisierbar. Im Beispiel 6.14 liefert z.B. $w' :\ a_1^x\ b_1^y\ b_1^x\ b_2^x\ a_1^y\ b_2^y\ a_2^x\ a_2^y$ wie die ungeteilte Serialisierung die leere Liste.

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

Beispiel 6.9 Wir betrachten das Auftragssystem $AS = (A, <)$ aus Abb. 6.18. Die Aufträge mögen folgende Verfeinerung in Leseaufträge l_i und s_i haben:

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

Abbildung 6.19 zeigt das entsprechend verfeinerte Auftragssystem $AS' = (A', <')$, wobei mit gestrichelten Pfeilen die Zugriffe auf die global benutzten Variablen x, y und z dargestellt sind.

Die mögliche Ausführungsfolge

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

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

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

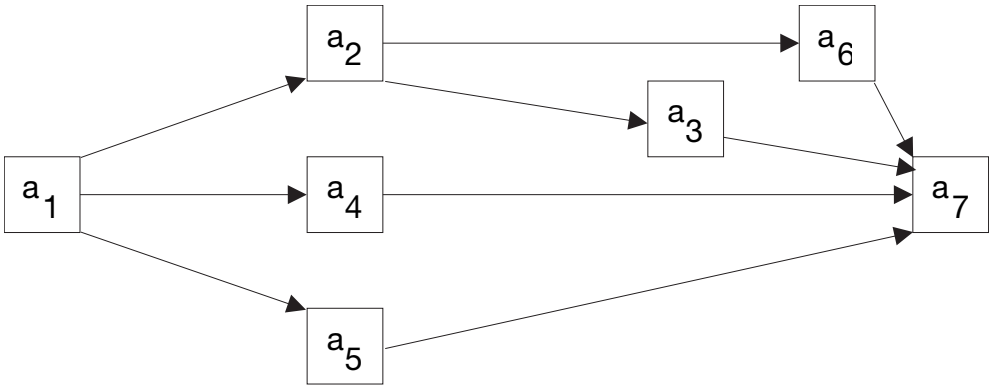


Abbildung 6.18: Konnektivitätsgraph des Auftragssystem AS

unteilbaren Anweisungen von w' können durch Elimination der lokalen Variablen übersichtlicher wie folgt geschrieben werden:

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

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

Definition 6.10 *Ein Auftragssystem $AS = (A, <)$ (Def. 6.8) heißt* schematisch *oder* uninterpretiert*, wenn die Auftragsmenge die Form*

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

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

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

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

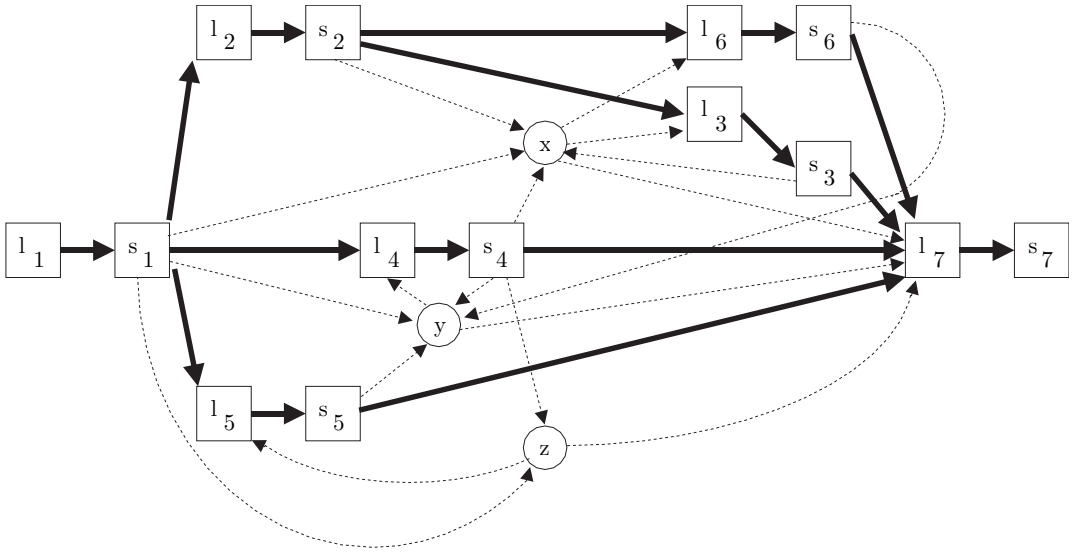


Abbildung 6.19: Verfeinertes Auftragssystem AS' mit Darstellung der Schreib/ und Lese-Zugriffe

Definition 6.11 *Die* Vergröberung $\hat{AS} = (\hat{A}, <)$ *eines schematischen Auftragssystems wie in 6.10 ist gegeben durch die Auftragsmenge*

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

und die Präzedenzen

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

Die Mengen ein_i und aus_j werden beibehalten.

Beispiel 6.12 Dem Auftragssystem aus Beispiel 6.9 entspricht das schematische Auftragssystem

$$AS = (A, <)$$

mit

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

(Mengenklammern von ein_i, aus_i bzw. $[\emptyset]$ wurden weggelassen). Abbildung 6.18 zeigt den Konnektivitätsgraph der Vergröberung von AS.

Oft ist es nützlich, am Anfang einer jeden Ausführungsfolge einen "Initialisierungs-Auftrag" zu haben, der alle Variablen beschreibt, und am Ende einen "Ausgabeauftrag", der alle Variablen liest und an den Drucker weitergibt. Ein solches Auftragssystem heißt *vollständig*. Das schematische Auftragssystem von Beispiel 6.9 ist vollständig.

Definition 6.13 *Ein schematisches Auftragssystem $AS = (A, <)$ wie in Definition 6.10 heißt vollständig, wenn $n \geq 3$ und*

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

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

Definition 6.14 *Eine* Interpretation I *eines schematischen Auftragssystems AS mit Bezeichnung wie in Definition 6.10 ist durch eine Wertemenge* D_I *(kurz* D*) und für jedes* $1 \leq i \leq n$ *durch Funktionen*

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

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

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

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

- d_0 *ist der Anfangszustand*
- *Ist* $w_j = l_i$ *ein Leseauftrag* $(1 \leq j \leq 2n)$, *dann ist* $d_j = d_{j-1}$ *und für eine nur in* a_i *vorkommende Menge von lokalen Variablen* $lok_i := \{lo_1, \ldots, lo_{p_i}\}$ *erhalten wir die Werte:*

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

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

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

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

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

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

für alle *Interpretationen* I *von AS gilt.*

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

Beispiel 6.15 Wählt man für das schematische Auftragssystem vom Beispiel 6.12 die Interpretation I mit $D_I = \mathbb{Z}$ und

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

und $d_0 = (0, 0, 0)$, dann erhält man Beispiel 6.9

(Da s_1 alle Variablen überschreibt, ist d_0 unwichtig). Für die Ausführungsfolge w in Beispiel 6.18 erhält man die Zustandsfolge:

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

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

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

Definition 6.16 *Es sei $AS = (A, <)$ wie in 6.10. Wir definieren für eine gegebene Ausführungsfolge $w \in F(AS)$ und $a_1, a_2 \in A$:*

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

Weiter sagen wir:

l_i *liest v von s_j in w, falls*

a) $s_j <_w l_i$

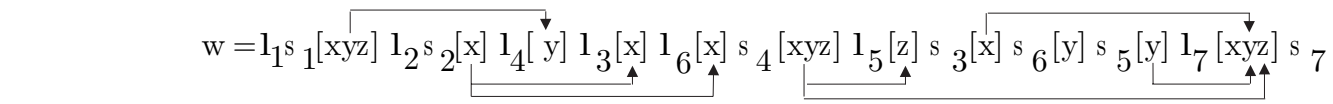
b) $v \in aus_j \cap ein_i$ *(v wird von s_j geschrieben und von l_i gelesen)*

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

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

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

Beispiel 6.17 Für w von Beispiel 6.18 lässt sich $W\ddot{U}(w)$ wie folgt angeben:



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

Definition 6.18 *Sei $AS = (A, <)$ ein schematisches und vollständiges Auftragssystem und $w \in F(AS)$ eine Ausführungsfolge. Die für w relevanten Aufträge definiert:*

a) Der Ausgabeauftrag ist relevant

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

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

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

Beispiel 6.19 Aus Beispiel 6.17 entnehmen wir: a_7 ist relevant für w, also auch a_5, a_3 und a_4. Da a_3 bzw. a_4 relevant für w sind, gilt dies auch für a_2 bzw. a_1. a_6 ist nutzlos für w. Die von ihm beschriebene Variable y wird von a_5 überschrieben, ohne seine Ergebnisse zu benutzen. a_6 ist nicht inhärent nutzlos, denn a_6 ist relevant für $w' = l_1 s_1 l_2 s_2 \ldots l_6 s_6 l_7 s_7$. Folglich ist AS relevant.

Satz 6.20 *Zwei Ausführungsfolgen $w_1, w_2 \in F(AS)$ eines schematischen Auftragssystems AS sind* genau dann *äquivalent, wenn für ihre Vervollständigung w_1', w_2' gilt:*

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

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

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

Beweis:
Der Beweis ist gleich demjenigen für den analogen Satz über die Äquivalenz von Programm-Schemata (Luckham et al 70) mit Hilfe von "Herbrand-Interpretationen". Wir erläutern die Hauptidee des Beweises anhand unseres Beispiels. Zunächst vervollständigen wir AS. Der Anfangszustand wird jetzt bei einer beliebigen Interpretation I durch die Funktionen des Initialisierungs-Auftrages bestimmt. Da wir alle Interpretationen betrachten, kommen auch alle ursprünglichen Anfangszustände vor.

Die Herbrand-Interpretation H hat als Wertebereich D die Menge aller Terme über den vorkommenden Funktionen f_j^i. Funktionsanwendungen bedeutet dann Termerweiterung. Wendet man z.B. $f(x)$ auf $f_3(f_1(x,y))$ an, dann ergibt sich $f(f_3(f_1(x,y)))$. $res(w_i, H)$ ist dann ein p-Tupel solcher Terme. Die Ausführungsfolge w aus Beispiel 6.17 liefert als Resultat:

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

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

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

(Man nehme im Beispiel $w_1' := w$ und $w_2' := w'$ aus Beispiel 6.9.)

Dann gilt auch für beliebige Interpretationen I

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

da dann die Termine gemäß I auszuwerten sind (in unserem Beispiel mit I aus Beispiel 6.15 ist $x = f_3^1(f_2^1) = 20 + 5 = 25$, $y = f_5^1(f_4^3(f_1^2)) = 1$, $z = f_4^3(f_1^2) = 1$).

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

6.2.2 Serialisierbarkeit

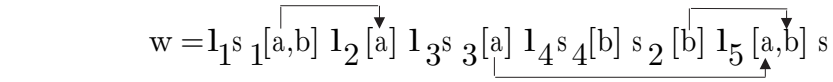
Definition 6.21 *Eine Ausführungsfolge $w = w_1 w_2 \ldots w_{2n} \in F(AS)$ eines schematischen Auftragssystems $AS = (A, <)$ heißt* seriell*, wenn für alle $1 \leq i < 2n$ gilt:*

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

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

- *$w \in F(AS)$ heißt* Serialisierung *von $w' \in F(AS)$, wenn w seriell und äquivalent zu w' ist. Ersetzt man alle Paare l_i, s_i durch a_i, dann wollen wir auch die so erhaltene Folge Serialisierung von w' nennen. Diese Folge ist Ausführungsfolge der Vergröberung $\hat{AS}$ (Def. 6.11).*
- *$w' \in F(AS)$ heißt* serialisierbar *(serializable), wenn w' eine Serialisierung $w \in F(AS)$ besitzt.*

Beispiel 6.22 Für das schematische Auftragssystem in Abb. 6.20 betrachte man die folgende Ausführungsfolge w:



Für jede Serialisierung w'' von w muss gelten:

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

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

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

Wie kann geprüft werden, ob eine Ausführungsfolge w_1 serialisierbar ist und somit einen konsistenten Zustand hergestellt hat? Eine (schwache) Serialisierung w_2 von w_1 muss äquivalent zu w_1 sein. Nach Satz 6.20 a) müssen ihre Vervollständigungen die gleichen relevanten Aufträge haben. Wir nehmen daher an, dass unser Auftragssystem AS vollständig war (oder gemacht wurde) und bilden das aus den für w_1 relevanten Aufträgen bestehende Unterauftragssystem $AS' = (A', <')$. Weiterhin sollen alle Aufträge in w_2 ungeteilt ausgeführt werden. Zu AS' konstruieren wir daher die Vergröberung $\hat{AS}' = (\hat{A}', <')$. Als Ergebnis unserer bisherigen Schritte halten wir fest: Falls w_1 überhaupt eine Serialisierung w_2 besitzt, dann muss eine solche Ausführungsfolge von $\hat{AS}'$ sein: $w_2 \in F_E(\hat{AS}')$. Nach Satz 6.20 b) müssen w_1 und w_2 außerdem die gleichen Werteübertragungsrelation haben. Wir nehmen daher $\text{WÜ}(w_1)$ als

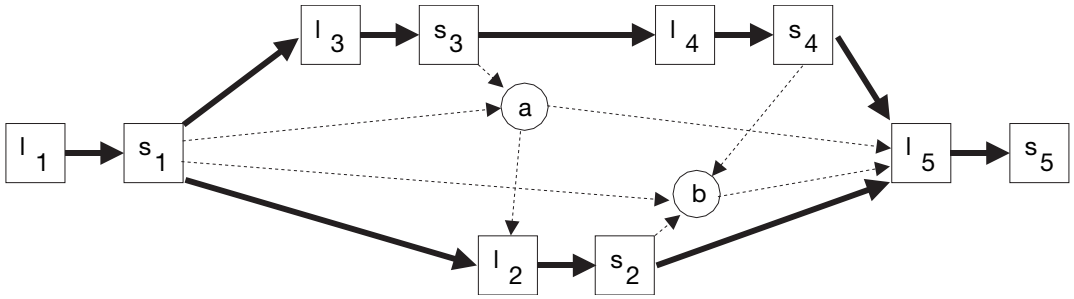
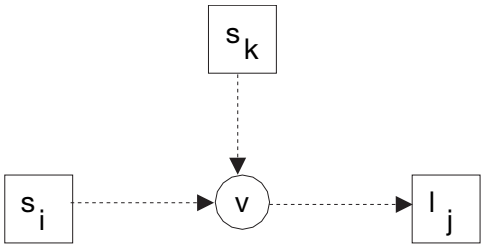


Abbildung 6.20: Schematisches Auftragssystem zu Beispiel 6.22

Präzedenzen zu $\lessdot'$ hinzu: $\lessdot'' := \lessdot' \cup \text{WÜ}(w_1)$. Ist $AS'' := (\hat{A}', <'')$ kein Auftragssystem (d.h. ist $<''$ nicht asymmetrisch), dann kann w_1 keine Serialisierung haben. Ist AS'' jedoch ein Auftragssystem, dann muss nicht jede Ausführungsfolge $w'' \in F(AS'')$ schwache Serialisierung von w_1 sein:

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



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

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

Definition 6.23 *Sei $AS = (\{l_1, s_1, \ldots, l_n, s_n\}, <)$ ein vollständiges, schematisches Auftragssystem und $w \in F_E(AS)$ eine Ausführungsfolge.*
$\hat{AS} = (A, <), A := \{a_1, \ldots, a_n\}$ sei die Vergröberung von AS.
$\hat{AS}' = (A', <')$ sei das durch die Menge A' der für w relevanten Aufträge definierte Unterauftragssystem von AS.

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

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

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

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

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

Satz 6.24 *Sei* $w_1 \in F(AS)$ *Ausführungsfolge eines vollständigen und relevanten schematischen Auftragssystems* AS.

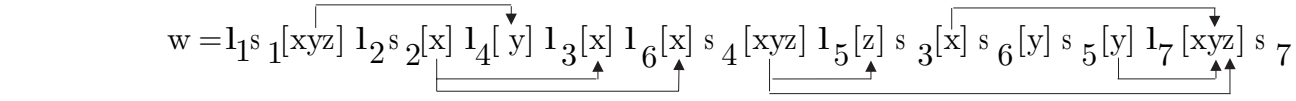
$SER(w_1) := \bigcup\{F(SG_i(w_1)) | SG_i(w_1) \in MSG(w_1)$ *ist Auftragssystem*$\}$ *ist die Menge der Serialisierungen von* w_1.

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

Beweis:
Sei $w_2 \in SER(w_1)$, also $w_2 \in F_E(SG_j(w_1))$ für einen Serialisierungsgraph $SG_j(w_1)$ von w_1. $F(SG_j(w_1))$ enthält nur serielle Ausführungsfolgen. Andererseits sind die Präzedenzen gerade so definiert, dass a_j ein v von a_i genau dann in w_1 liest, wenn dies auch für w_2 gilt. Also enthalten w_1 und w_2 die gleichen relevanten Aufträge. w_1 und w_2 sind nach Satz 6.20 äquivalent. Sei umgekehrt $w_2 \in F(AS)$ eine Serialisierung von w_1. Da w_1 und w_2 äquivalent sind, haben sie nach Satz 6.20 dieselben relevanten Aufträge und die "liest v von"-Relation zwischen Aufträgen stimmt überein. Folglich muss $w_2 \in F_E(SG_i(w_1))$ für ein $SG_i(w_1) \in MSG(w_1)$ gelten. □

Hat man ein beliebiges schematischen Auftragssystem $AS = (A, <)$ und $w \in F_E(AS)$ vorliegen, dann bestimme man zunächst die für w relevanten Aufträge $A' \subset A$ (Def. 6.18). Mit Def. 6.23 und Satz 6.24 lassen sich diejenigen Auftragssysteme gewinnen, die alle Serialisierungen mit Aufträgen aus A' als Ausführungsfolgen enthalten. Sollen die Auftragssysteme alle Aufträge von A enthalten, dann sind die für w nutzlosen Aufträge mit ihren Präzedenzen in den Serialisierungsgraphen einzufügen. Zusätzlich müssen ggf. noch Präzedenzen angebracht werden, die gewährleisten, dass diese Aufträge in jeder Ausführungsfolge nutzlos sind. w ist genau dann serialisierbar, wenn dabei ein zyklenfreier Graph, also ein Auftragssystem, entsteht.

Beispiel 6.25 Wir betrachten $AS = (A, <)$ aus Beispiel 6.12 mit der Ausführungsfolge w aus Beispiel 6.17, wo die Werteübertragungsrelation WÜ(w) angegeben ist mit:



Nach Beispiel 6.19 sind $A' = \{a_1, a_2, a_2, a_4, a_5, a_7\}$ relevant. $\hat{AS}' = (A', <')$ gemäß Def. 6.23 ist in Abb. 6.18 gegeben, wenn man dort a_6 streicht.

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

Abbildung 6.21 a) zeigt den Serialisierungsgraphen $SG_1(w)$ (für die 1. Alternative in $\lessdot_2$). $SG_2(w)$ erhält man durch Ersetzen von (a_4, a_2) durch (a_3, a_4). Da ein Zyklus entsteht, ist $SG_2(w)$ kein Auftragssystem.

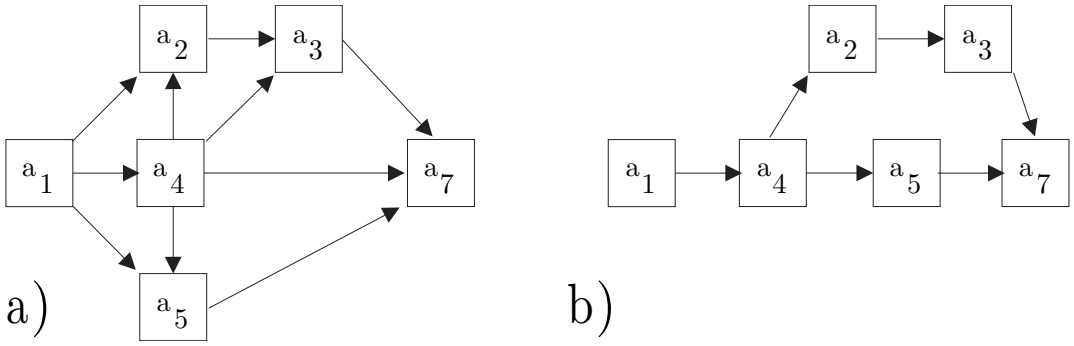


Abbildung 6.21: Serialisierungsgraph $SG_1(w)$ (ohne transitiven Kanten)

Abbildung 6.21 b) zeigt den Konnektivitätsgraphen von $SG_1(w)$. $a_1a_4a_2a_5a_3a_7 \in F_E(SG_1(w))$ ist damit eine Serialisierung der relevanten Aufträge von w. $F_E(SG_1(w))$ ist genau die Menge aller schwachen Serialisierungen von w aus für w relevanten Aufträgen.

Ist man an einer Serialisierung (aller Aufträge) von w interessiert, dann muss der für w nutzlose Auftrag a_6 mit seinen (direkten) Präzedenzen $a_2 \lessdot a_6 \lessdot a_7$ eingefügt werden. Außerdem muss man durch zusätzliche Präzedenzen sicherstellen, dass a_6 bei allen Ausführungsfolgen nutzlos ist (hier genügt z.B. $a_6 \lessdot a_5$). Das so erhaltene Auftragssystem AS_1 in Abb. 6.22 enthält alle Serialisierungen von w als Ausführungsfolgen, darunter auch die aus Beispiel 6.18 bekannte:

$$w' = a_1a_4a_2a_6a_5a_3a_7$$

.

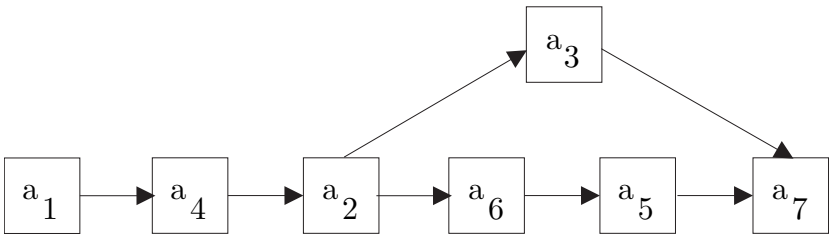


Abbildung 6.22: $SG_1(w)$ mit nutzlosem Auftrag a_6

Beispiel 6.26 Die Ausführungsfolge des Auftragssystems von Abb. 6.23 a) ist nicht serialisierbar, da der (einzige) Serialisierungsgraph in Abb. 6.23 b) kein Auftragssystem ist.

Für die folgende Interpretation I_i:

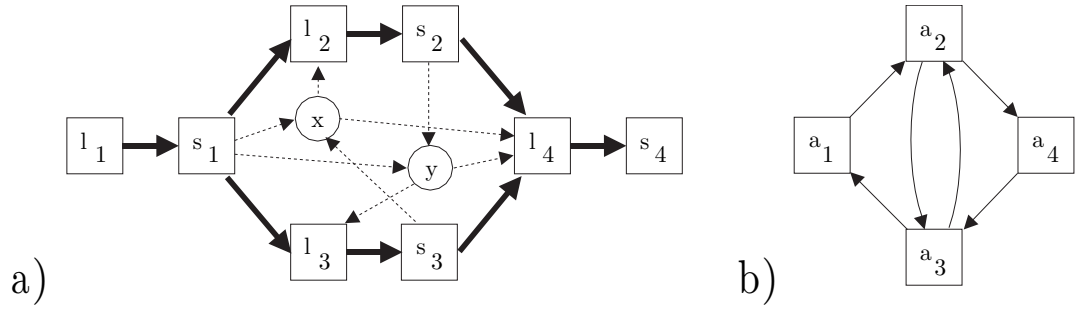


Abbildung 6.23: Auftragssystem mit zyklischem Serialisierungsgraphen

$$w = l_1 s_1 [xy]\ l_2 [x]\ l_3 [y]\ s_2 [y]\ s_3 [x]\ l_4 [xy]\ s_4$$

$l_1 : skip$
$s_1 : x, y := 1, 2$
$l_2 : lo_2 := x$
$s_3 : lo_3 := y$
$s_2 : y := lo_2 + 1$
$s_3 : x := lo_3 + 2$
$l_4 : lo_4^1, lo_4^2 := x, y$
$s_4 : skip$

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

$x, y := 1, 2;$
cobegin $y := x + 1 \ || \ x := y + 2$ **coend**

Das Problem der Serialisierbarkeit ist NP-vollständig (Papadimitriou 79). Damit wächst die Ausführungszeit jedes Algorithmus, der für eine Ausführungsfolge und beliebige Interpretationen feststellen soll, ob ein konsistenter Zustand erreicht wird, praktisch exponentiell. Daher wurden Teilklassen des Problems untersucht, die eine geringere, d.h. polynomiale Komplexität haben. Eine Übersicht findet man in (Papadimitriou 79; Bernstein et al 79). Dies gilt zum Beispiel schon dann, wenn man für das Auftragssystem voraussetzt, dass jede von einem Auftrag beschriebene Variable von demselben Auftrag gelesen wird (also: $aus_i \subseteq ein_i$ für alle $1 < i \leq n$).

6.2.3 Funktionalität

Sind in einem Auftragssystem alle Ausführungsfolgen serialisierbar, dann sind alle möglichen Resultate konsistent. Oft ist es jedoch notwendig, darüberhinaus trotz nebenläufiger Ausführungen ein einziges

und damit eindeutiges Resultat aller Berechnungen sicherzustellen. Da das Resultat dann eine Funktion der Anfangswerte ist, nennen wir ein solches Auftragssystem *funktional*.

Für das Umordnungsprogramm *Min-Max* von Beispiel 6.35 (auf Seite 221) nehmen wir feste Anfangswerte A_0 und B_0 an. Das Programm möge dann nach $x \geq 1$ Schleifendurchläufen terminieren (den Fall $x = 0$ betrachten wir hier nicht). Da wir an möglichst großer Nebenläufigkeit interessiert sind, formulieren wir diesen Prozess als (interpretiertes) Auftragssystem. Alle möglichen Abläufe sollen natürlich das gleiche, in Beispiel 6.35 angegebene Resultat haben.

Da wir an unserem Verfahren zur Sicherstellung der Funktionalität interessiert sind, betrachten wir das den zugehörigen Prozess beschreibende Auftragssystem in Abbildung 6.27. Die Notwendigkeit der dort schon angegebenen Präzedenzen leuchtet unmittelbar ein. Die Aufgliederung der Aufträge in Lese- und Schreibaufträge entsprechend der Definition 6.10 eines schematischen Auftragssystems ist vorausgesetzt, aber nicht in der Abbildung dargestellt. Die Werte der verschiedenen Tests werden jeweils neuen booleschen Variablen c_i, d_j zugewiesen. Soll das Programm korrekt sein, dann muss es zumindest funktional sein.

Wir fragen, welche zusätzlichen Präzedenzen notwendig sind, um Funktionalität zu gewährleisten. Man mache sich klar, dass dazu die in Abb. 6.27 (als Konnektivitätsgraph) angegebenen Präzedenzen notwendig und hinreichend sind. Jede Ausührung des Programms von Beispiel 6.35 bei x-maligem Schleifendurchlauf ($x \geq 1$) hält diese Präzedenzen ein. Also sind die Ergebnisse aller Ausführungsfolgen (bei festen Anfangswerten A_0, B_0) identisch und das Programm in diesem Sinne "funktional". Wir werden in diesem Abschnitt sehen, wie man diese zusätzlichen Präzedenzen von Abb. 6.27 systematisch findet. Dieses Verfahren kann dazu benutzt werden, ein nebenläufiges Programm wie in Beispiel 6.35 (auf Seite 221) zu konstruieren, das größtmögliche Nebenläufigkeit zulässt.

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

Definition 6.27 *Sei AS ein schematisches Auftragssystem. AS heißt* funktional, *falls alle Ausführungsfolgen zueinander äquivalent sind (Def. 6.14).*

Sei $d_0, d_1, \ldots, d_{2n}$ die Zustandsfolge von $w \in F(AS)$ bei einer Interpretationen I (Def. 6.14) und $v \in V$ eine Variable. Weiterhin sei $d_{i_0}, d_{i_2}, \ldots, d_{i_k}$ die Teilfolge von Zuständen, in denen der Variablen v Werte zugewiesen werden (also genau diejenigen Zustände d_{i_j} mit $w_{i_j} = s_k$ und $v \in aus_k$). Die Folge der v-Komponenten heißt Spur (history) von v in w bei I:

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

Der Vektor

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

heißt Spur von w bei I.

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

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

In Satz 6.24 wird bewiesen, dass azyklische Serialisierungsgraphen funktionale Auftragssysteme sind. Die folgenden Definitionen bereiten notwendige und hinreichende Kriterien für die Funktionalität und Spurfunktionalität eines Auftragssystems vor.

Definition 6.28 *Sei AS ein vollständig schematisches Auftragssystem.*

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

$$dis(a_i, a_j) := (aus_i \cap aus_j = ein_i \cap aus_j = aus_i \cap ein_j = \emptyset) \quad \text{“Bernstein”-Relation}$$

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

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

Satz 6.29 *Sei AS ein vollständiges schematisches Auftragssystem.*

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

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

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

Es seien zunächst alle relevanten (verlustfreien) Aufträge von $AS = (A, <)$ paarweise störungsfrei. Für beliebige Ausführungsfolgen $w_1, w_2 \in F(AS)$ müssen wir zeigen, dass sie (spur-)äquivalent sind. Seien a_i und a_j zwei beliebige relevante (verlustfreie) Aufträge von AS. Liest a_j eine Variable v von a_i in w_1, dann muss wegen $v \in aus_i \cap ein_j \neq \emptyset$ entweder $a_i < a_j$ oder $a_j < a_i$ gelten.

Zum Beispiel im ersten Fall $a_i < a_j$ muss s_i auch in w_2 vor l_j stehen, d.h. $s_i <_{w_2} l_j$. Schreibt ein dritter relevanter (verlustfreier) Auftrag a_k auf v, dann gilt nach der Definition von “a_j liest v von a_i” entweder $s_k <_{w_1} s_i$ oder $l_j <_{w_1} s_k$. Da a_k und a_i bzw. a_k und a_j störungsfrei sind, gilt $a_k < a_i$ oder $a_j < a_k$ und folglich auch $s_k <_{w_2} s_i$ oder $l_j <_{w_2} s_k$. Damit ist bewiesen, dass unter den relevanten (verlustfreien) Aufträgen der Auftrag a_j ein v von a_i bei w_1 genau dann liest, wenn dies auch bei w_2 gilt. Nach Def. 6.18 müssen dann w_1 und w_2 auch die gleichen relevanten Aufträge haben. w_1 und w_2 sind damit äquivalent (Satz 6.20).

(Sei a_i ein verlustfreier Auftrag mit $v_i \in aus_i$. Weiterhin sei $\hat{w}_1$ bzw. $\hat{w}_1$ dasjenige Anfangsstück von w_1 bzw. w_2 das mit s_i endet. $\hat{w}_1$ und $\hat{w}_2$ haben dann die gleichen relevanten Aufträge und sind nach Satz 6.20 äquivalent. Insgesamt sind $\hat{w}_1$ und $\hat{w}_2$ und damit auch w_1 und w_2 spuräquivalent.) Sei umgekehrt AS nun funktional (spurfunktional) und a_i, a_j seien zwei für AS relevante (verlustfreie) Aufträge.

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

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

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

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

a_j kann v von a_i in w_1 nicht lesen, da sonst w_1 und w_2 nicht äquivalent (spuräquivalent) sein können (Satz 6.20).

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

Daraus folgt $a_i < a_j$ im Widerspruch zur Annahme.

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

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

Beispiel 6.30

$AS = (\{$

$l_1[v_1, v_2]$,	$s_1[v_3]$,
$l_2[v_1]$,	$s_2[v_4]$,
$l_3[v_3, v_4]$,	$s_3[v_1]$,
$l_4[v_3, v_4]$,	$s_4[v_5]$,
$l_5[v_4]$,	$s_5[v_2]$,
$l_6[v_5]$,	$s_6[v_5]$,
$l_7[v_1, v_2, v_4]$,	$s_7[v_4]$,
$l_8[v_1, v_3]$,	$s_8[v_5]$ $\}$, $\lessdot$)

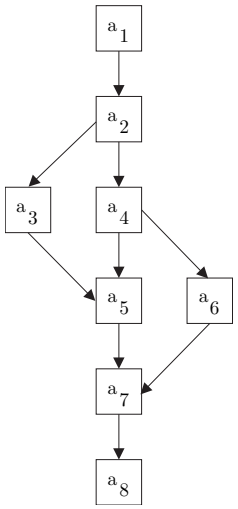


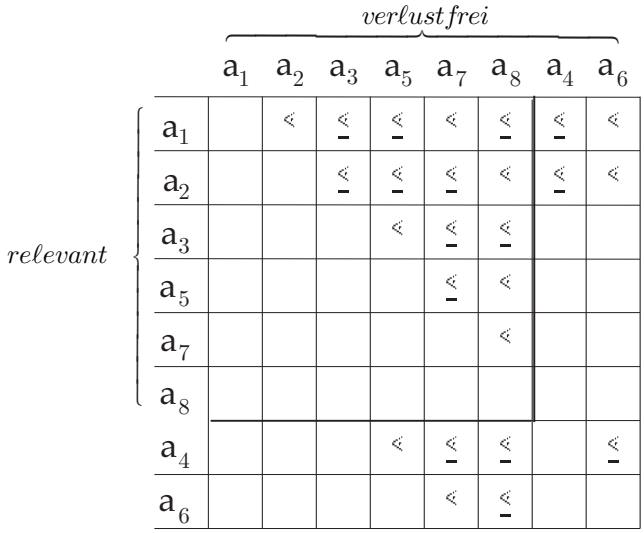
Abbildung 6.24: Ein schematisches Auftragssystem

Für die serielle Ausführungsfolge

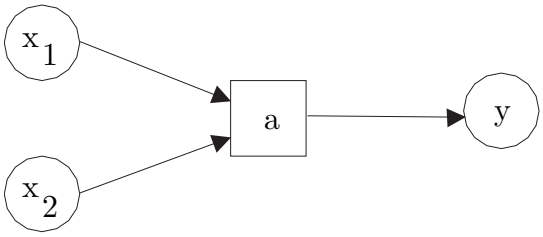
$$w = l_1 s_1 l_2 s_2 \ldots l_8 s_8$$

sind die Aufträge $a_1, a_2, a_3, a_5, a_7, a_8$ relevant, nicht jedoch a_4 und a_6. (AS zwischenzeitlich vervollständigen!) a_4 und a_6 schreiben nur auf die Variable v_5, die in jeder Ausführungsfolge $w' \in F(AS)$ nachfolgend von a_8 überschrieben wird und zwar unabhängig vom alten Wert. a_4 und a_6 sind also nutzlos, was allein durch die Präzedenzen $a_4 \lessdot a_8$ und $a_6 \lessdot a_8$ sichergestellt ist. Um Satz 6.29 zu benutzen, berechnen wir die Relation $dis(a_i, a_j)$ für alle $a_i \neq a_j$ und kennzeichnen ihre Ungültigkeit durch (-) in der folgenden Tabelle. (Nur oberhalb der Diagonale eingetragen!)

Da für alle relevanten (verlustfreien) Aufträge $a_i \neq a_j$ gilt: $dis(a_i, a_j) \vee a_i < a_j \vee a_j < a_i$ (d.h. falls (-), dann auch ($\lessdot$) für jeden Eintrag), ist AS funktional (und spurfunktional). Entsprechend überprüfe man, dass AS von Abb. 6.27 funktional und spurfunktional ist.



Um mehrere Implikationen in einer Graphik darzustellen, hat sich folgende durch die Netztheorie gestützte Schreibweise bewährt:

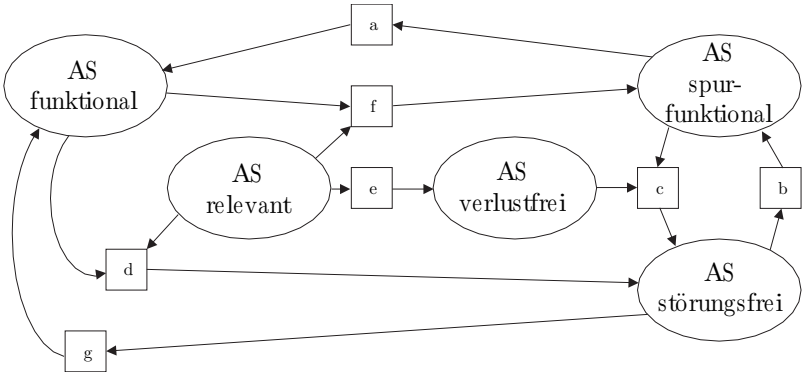


für die Behauptung a : (6.3)

$x_1 \wedge x_2 \Rightarrow y$

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

Satz 6.31



Beweis:

a) *Spuräquivalente Folgen sind auch äquivalent.*

b) *Sind alle Aufträge störungsfrei, dann auch alle verlustfreien. Nach Satz 6.29 b) ist also AS spurfunktional.*

c) *(nach Satz 6.29 (b))*

d) *Sind alle Aufträge relevant und ist AS funktional, dann sind alle Aufträge störungsfrei (Satz 6.29 (a)).*

e) *Relevante Aufträge müssen verlustfrei sein.*

f) *Folgt aus d) und b).*

g) *Folgt aus a) und b).*

□

Im Beispiel 6.30 wurde deutlich, dass nicht alle Präzedenzen eines Auftragssystems zur Gewährleistung von Funktionalität bzw. Spurfunktionalität notwendig sind. Ist dies doch der Fall, dann spricht man von einem Auftragssystem mit minimaler Präzedenzrelation oder maximaler Nebenläufigkeit bezüglich der entsprechenden Eigenschaft.

Definition 6.32 *Ein schematisches Auftragssystem $AS = (A, \lessdot)$ heißt* maximal nebenläufig *für Funktionalität (bzw. für Spurfunktionalität), wenn AS einerseits funktional (bzw. spurfunktional) ist, andererseits das Entfernen einer beliebigen Präzedenz $(a_i, a_j) \in \lessdot$ aus der Präzedenzrelation diese Eigenschaft zerstören würde.*

Nach folgendem Verfahren lässt sich mit Satz 6.29 ein (spur-)funktionales Auftragssystem $AS = (A, <)$ in ein für (Spur-)Funktionalität maximal nebenläufigen Unterauftragssystem $AS'(A', <')$ verwandeln, dessen Ausführungsfolgen alle (spur-)äquivalent zu denjenigen von AS sind.

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

$$\lessdot'' = \{(a_i, a_j) \in A' \times A' | a_i \lessdot a_j \text{ und } \neg dis(a_i, a_j)\} \tag{6.4}$$

$AS' = (A', <')$ hat nur relevante (verlustfreie) Aufträge, die paarweise störungsfrei sind. Nach Satz 6.29 ist also AS' funktional (spurfunktional). Das Entfernen einer beliebigen Präzedenz $(a_i, a_j) \in \lessdot'$ würde wegen $\neg dis(a_i, a_j)$ diese Eigenschaft zerstören., Also ist AS' maximal nebenläufig für Funktionalität (Spurfunktionalität).

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

Beispiel 6.33 Wir konstruieren zu $AS = (A, <)$ in Beispiel 6.30 ein maximal nebenläufiges Auftragssystem:

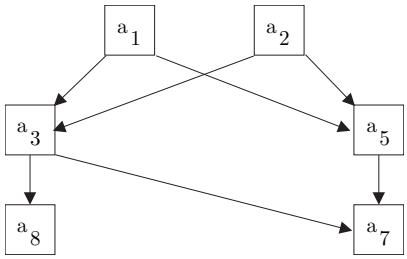


Abbildung 6.25: Für Funktionalität maximal nebenläufiges Auftragssystem

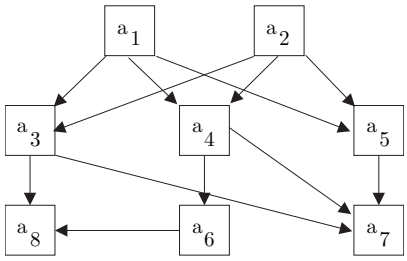


Abbildung 6.26: Für Spurfunktionalität maximal nebenläufiges Auftragssystem

a) $AS' = (A', <')$ ist maximal nebenläufig für Funktionalität, wobei $A' = \{a_1, a_2, a_3, a_5, a_7, a_8\}$ die Menge der relevanten Aufträge ist und $<'$ aus der transitiven Hülle derjenigen Paare $(a_i, a_j) \in A' \times A'$ besteht, die in der Tabelle von Beispiel 6.30 einen Eintrag ($\trianglelefteq$) haben (siehe Abb. 6.25)

b) Bildet man $AS'' = (A, <'')$ mit allen Aufträgen A und $\lessdot'' = \lessdot' \cup \{(a_4, a_8), (a_6, a_8)\}$, dann ist AS'' ebenfalls maximal nebenläufig bezüglich Funktionalität. a_4 und a_6 sind dann nämlich nutzlos (vgl. Abb. 6.26).

c) $AS'' = (A, <''\ ')$ ist maximal nebenläufig für Spurfunktionalität, wobei $<''\ '$ aus der transitiven Hülle derjenigen Paare $(a_i, a_j) \in A \times A$ besteht, die in der Tabelle von Beispiel 6.26 einen Eintrag ($\trianglelefteq$) haben (alle Aufträge sind verlustfrei) (siehe Abb. 6.26).

Auch das Auftragssystem AS von Abb. 6.27 ist maximal nebenläufig für Funktionalität.

Aufgabe 6.34 (Funktionalität)

Betrachten Sie das folgende Fragment eines Programmes:

$x := a + b$;
cobegin $z := x * a \;||\; r := p/z \;||\; u := r * b \;||\; v := b/x$ **coend**;
$u := a * a$;
$s := z/r$;
$u := x/a$;

Das Semikolon bedeutet wie üblich die Hintereinanderausführung, während

cobegin $anw1 \parallel anw2 \parallel anw3 \parallel anw4$ **coend**

die Parallelausführung der Einzelanweisungen bedeute. Bezeichnen Sie die Zuweisungen in der angegebenen Reihenfolge der Aufschreibung als Aufträge a_1 bis a_8 und konstruieren Sie das schematische Auftragssystem AS entsprechend dieser Bedeutung des Programmablaufs.

a) Entscheiden Sie mit Satz 6.29, ob AS funktional bzw. spurfunktional ist. Falls nicht, soll eine minimale Anzahl von Präzedenzen eingefügt werden, um dies zu erreichen.

b) Konstruieren Sie daraus das für Funktionalität bzw. Spurfunktionalität maximal nebenläufige Auftragssystem.

Beispiel 6.35

Gegeben seien zwei disjunkte endliche Mengen A und B von ganzen Zahlen. Sie sollen so in Mengen A und B jeweils gleicher Mächtigkeit umgeformt werden, dass alle Elemente in A kleiner als alle Elemente in B sind.

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

var max, min : **integer**;
var A, B : **set of integer**;
$max, min := max(A), min(B)$;
while $max > min$ **do**
conbegin $A := A \setminus \{max\}; A := A \cup \{min\}$ $\parallel$
$B := B \setminus \{min\}; B := B \cup \{max\}$
coend;
$max, min := max(A), min(B)$
endwhile

$\{A \cup B = A_0 \cup B_0, A \cap B = \emptyset, |A| = |A_0|, |B| = |B_0|, max(A) < min(B)\}$ (Nachbedingung)

Das Programm (nach [Dij77]und [Gri81]) benutzt die Operationen $max(A)$, $min(B)$ für das maximale bzw. minimale Element einer Menge.

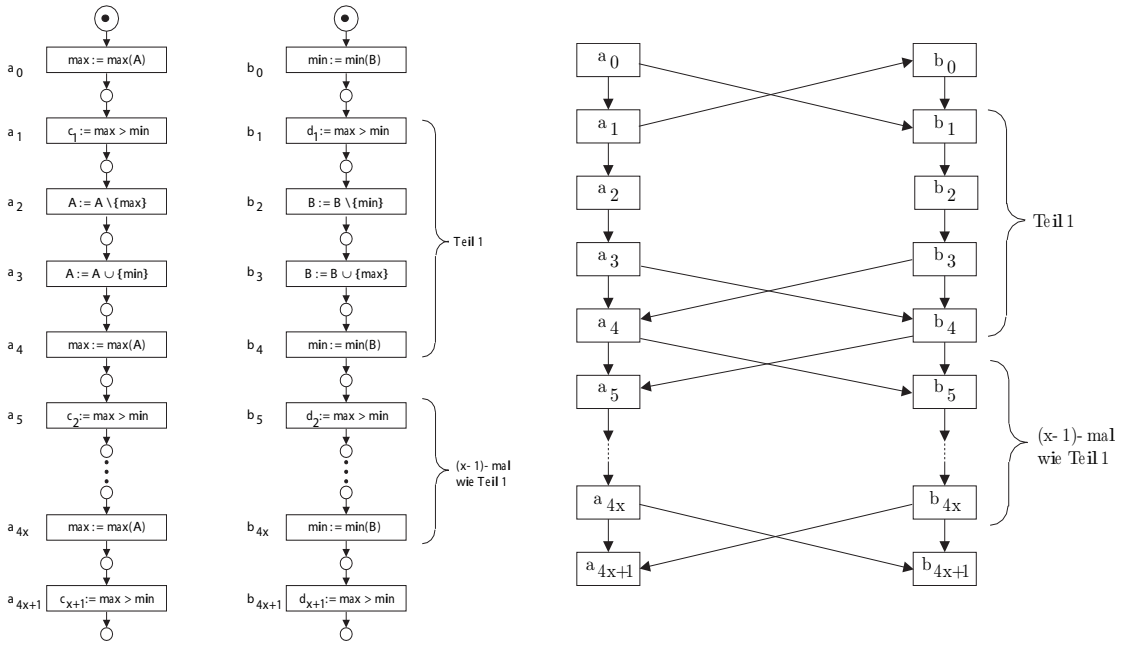


Abbildung 6.27: Interpretiertes Auftragssystem zu Beispiel 6.35
und zusätzliche für Funktionalität notwendige Präzedenzen.

Kapitel 7

Höhere Petrinetze

In diesem Kapitel werden Platz/Transitions-Netze (Kapitel 3) zu solchen Netzen erweitert, die statt der einfachen, „schwarzen“ Marken, allgemeinere Datentypen wie **integer** erlauben. Dies erlaubt große Netze in sehr viel kleinerer und kompakterer Form darzustellen, wobei viele Eigenschaften der Platz/Transitions-Netze übertragbar sind. Dies gilt jedoch nicht für die Eigenschaft der Beschränktheit, die (wie in Abschnitt 3.8.2 gezeigt) hier nicht entscheidbar ist. Enthält das Netz jedoch nur endliche Datentypen (die hier *Farben* genannt werden) so sind gefärbte Netze gleichmächtig zu Platz/Transitions-Netzen (die Beschränktheit ist dann z.B. entscheidbar) und der genannte Kompaktifizierungs-Vorteil ist dennoch möglich.

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.

In diesem Zusammenhang wird auch das Werkzeug RENEW kurz beschrieben, das den graphischen Entwurf unterstützt und eine Ausführung des Schaltens erlaubt.

Bücher zu gefärbten Netzen sind insbesondere [GV03] (Kapitel 1 bis 4), [JK09], [Rei10] und der Artikel [Kum01] zu Referenznetzen.

7.1 Kantenkonstante Netze

In dem Beispielnetz von Abbildung 3.5 (auf Seite 68) wurden die Objekte *Wagen a* und *Wagen b* durch einfache Marken repräsentiert. Diese Marken sind an sich nicht zu unterscheiden. In dem Netz 3.5 werden sie nur durch ihre Lage in den unterschiedlichen Plätzen p_1 und p_{10} gekennzeichnet. In Hinblick auf eine direktere Modellierung der realen Situation und um ein kompakteres Modell zu erhalten, wäre es vorteilhaft die Rennwagen direkt durch unterscheidbare Marken (Bezeichner) a und b auf *einem Platz* darzustellen, wie dies in Abb. 7.1 durch den Platz $p_{1\&10}$ gezeigt wird. Dieser Platz repräsentiert gleichsam die Aufstellungszone für den Rennbeginn.

Unterscheidbare Marken werden als Sorten oder Typen gruppiert, die Farben heißen. Jedem Platz p wird durch $cd(p)$ (“colour domain”) eine Farbe zugeordnet, hier also $cd(p_{1\&10}) = cars = \{a, b\}$ (in der grafischen Darstellung kursiv beim Bezeichner des Platzes). Andere Beispiele von Farben sind “integer”

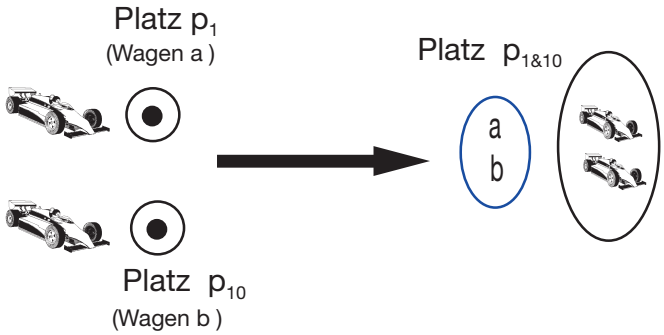


Abbildung 7.1: Übergang zu individuellen Marken

oder "boolean". Wie im Netz 7.2 zu sehen, 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. Das Netz 7.3 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 7.1 *Eine* Multimenge*(bag, multi-set) 'bg', über einer nicht leeren Menge A, ist eine Abbildung $bg : A \to \mathbb{N}$, die auch als formale Summe $\sum_{a\in A} bg(a)'a$ dargestellt wird. $Bag(A)$ bezeichnet die Menge aller Multimengen über A. Die Mengenoperationen Vereinigung, Inklusion, Differenz und Kardinalität werden wie folgt auf $Bag(A)$ als* Summe $(+)$*, Inklusion* $(\leq)$*,* Differenz $(-)$ *und Kardinalität* $(|\cdot|)$ *übertragen:*

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

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

Beispiel: *$bg_1 = \{a, a, b, b, b, d\}_b$ ist Multimenge über $A = \{a, b, c, d\}$: (der Index unterscheidet die schließende Klammer von der Mengenklammer) oder als formale Summe: $bg_1 = 2'a + 3'b + d$. Mit $bg_2 = a + 2'b + c$ erhält man $bg_1 + bg_2 = 3'a + 5'b + c + d$ 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 7.2 *Ein kantenkonstantes gefärbtes Petrinetz (KKN) wird als Tupel $\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, W, \mathbf{m}_0\rangle$ definiert, wobei*

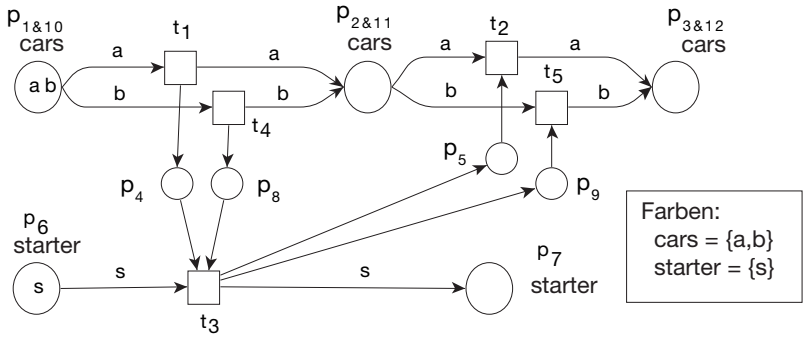


Abbildung 7.2: Das kantenkonstante gefärbte Netz $\mathcal{N}_1$

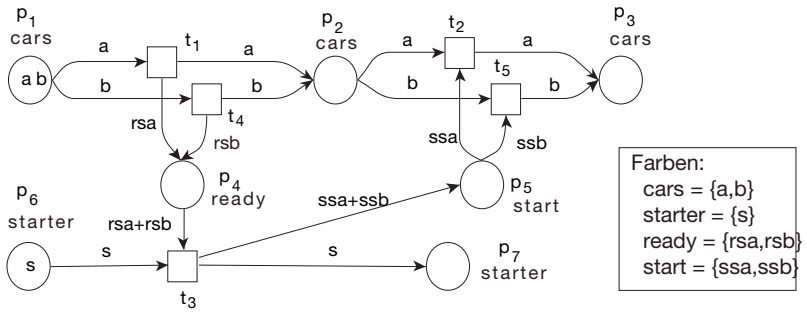


Abbildung 7.3: Das kantenkonstante gefärbte Netz $\mathcal{N}_2$

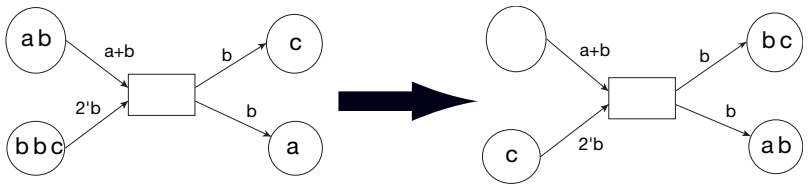


Abbildung 7.4: Schaltregel für kantenkonstante CPN

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

Beispiel: Für das kantenkonstante Netz 7.3 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 7.3 *a) Die Markierung eines KKN* $\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, W, \mathbf{m}_0\rangle$ *ist ein Vektor* $\mathbf{m}$ *mit* $\mathbf{m}(p) \in Bag(cd(p))$ *für jedes* $p \in P$ *(auch als Abbildung* $\mathbf{m} : P \to Bag(\bigcup \mathcal{C})$ *mit* $\mathbf{m}(p) \in Bag(cd(p))$ *für jedes* $p \in P$ *aufzufassen).*

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

c) $\widetilde{W}(x,y) := \begin{cases} W(x,y) & \text{falls } (x,y) \in F \\ 0 & \text{sonst} \end{cases}$
ist die Erweiterung von W *auf* $(P \times T) \cup (T \times P)$.

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

d) Definiert man $W(\bullet, t) := (\widetilde{W}(p_1,t), \ldots, \widetilde{W}(p_{|P|},t))$ *als Vektor der Länge* $|P|$ *(und sinngemäß ebenso* $W(t,\bullet)$ *), dann kann die Nachfolgemarkierung kürzer durch Vektoren definiert werden:* $\mathbf{m} \xrightarrow{t} \mathbf{m}' \Leftrightarrow \mathbf{m} \geq W(\bullet,t) \wedge \mathbf{m}' = \mathbf{m} - W(\bullet,t) + W(t,\bullet)$.
Dabei sind die Multimengenoperatoren komponentenweise auf Vektoren zu erweitern.

Beispiel: Für das Netz 7.3 ist die Anfangsmarkierung (als Vektor dargestellt)
$\mathbf{m}_0 = (a+b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset)$. Da $W(\bullet, t_1) = (a, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ ist die Transition t_1 in $\mathbf{m}_0$ aktiviert und die Nachfolgemarkierung ist $\mathbf{m}' = \mathbf{m}_0 + W(t_1, \bullet) - W(\bullet, t_1) =$
$(a+b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset) + (\emptyset, a, \emptyset, rsa, \emptyset, \emptyset, \emptyset) - (a, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) = (b, a, \emptyset, rsa, \emptyset, s, \emptyset)$. $a+b$ bzw rsa sind hier beispielsweise die Multimengen $\{a,b\}_b$ bzw. $\{rsa\}_b$, dargestellt als formale Summen.

Die Schaltregel wird in Abb. 7.4 an einem abstrakten Beispiel erläutert, wobei Multimengen vorkommen, die keine Mengen sind!

Definition 7.4 *Die Nachfolgemarkierungsrelation von Definition 7.3 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* $\exists \mathbf{m}_1 : \mathbf{m} \xrightarrow{w} \mathbf{m}_1$ *und* $FS(\mathcal{N}) := \{w \in T^* \mid \mathbf{m_0} \xrightarrow{w}\}$ *ist die Menge der* Schaltfolgen *(firing sequence set) von $\mathcal{N}$.*

Damit ist die Semantik eines kantenkonstanten gefärbten Petrinetzes durch ein Transitionssystem definiert. Dies wird in der Anmerkung auf Seite 227 explizit gegeben.

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

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

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

Definition 7.5 *Der Erreichbarkeitsgraph eines kantenkonstanten gefärbten Netzes $\mathcal{N}$ ist ein Tupel* $RG(\mathcal{N}) := (Kn, Ka)$ *mit Knotenmenge* $Kn := \mathbf{R}(\mathcal{N})$ *(siehe Def. 7.4) und Kantenmenge* $Ka := \{(\mathbf{m}_1, t, \mathbf{m}_2) \mid \mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2\}$ *(siehe Beispiel in Abb. 7.5).*

Anmerkung: 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 $FS(TS) = FS(\mathcal{N})$.

Die Zusammenfassung von Plätzen beim Übergang von einfachen Netzen zu kantenkonstanten Netzen ist im oberen Teil von Abbildung 7.28 (auf Seite 244) als Vergröberung von Plätzen („Platz-Faltung“) dargestellt. Entsprechend zeigt Abbildung die 7.27 (auf Seite 243) den Übergang von einem einfachen Netz zu einem P/T-Netz mit mehreren Marken auf einem Platz und Kantengewichten größer als eins.

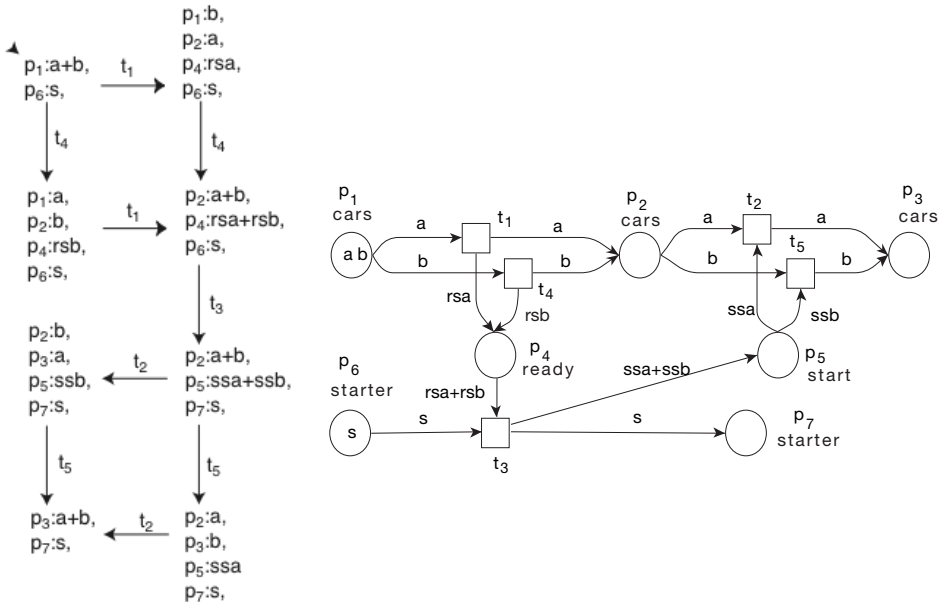


Abbildung 7.5: Erreichbarkeitsgraphen des kantenkonstanten Netzes von Abb.7.3

7.2 Gefärbte Netze

Bei der Einführung von kantenkonstanten Netzen wurden Plätze verschmolzen. Um ein Netz noch kompakter zu machen liegt es nahe, auch Transitionen zusammenzulegen, insbesondere wenn sie verhaltensähnliche Aktionen modellieren. Wenn dabei das gleiche Verhalten dargestellt werden soll, dann ist die zusammengelegte Transition zu parametrisieren. Dies kann durch Variablen geschehen, wie dies im Netz 7.8 durch die Variablen x und y der Fall ist. Beispielsweise modelliert die Transition t_1 im Netz 7.8 durch die Variablenbelegung $\beta_1 := [x = a, y = rsa]$ die Transition t_1 im Netz 7.3, während mit $\beta_2 := [x = b, y = rsb]$ die Transition t_4 in 7.3 simuliert wird. Natürlich wäre $\beta_3 := [x = a, y = rsb]$ keine zulässige Belegung. Dies wird durch Schutzbedingungen (guards) ausgeschlossen. Guards sind über den Variablen an einer Transition definierte Prädikate. Sie können auch als Testbedingung an einer Verzweigung eingesetzt werden. Allgemein stehen an den Kanten eines solchen gefärbten Netzes Ausdrücke über den Variablen, die mit einer zu wählenden Belegung Multimengen definieren, die dann die gleiche Rolle wie bei kantenkonstanten Netzen spielen. Die Form der Ausdrücke lassen wir hier offen, um flexibel zu bleiben. Wichtig ist nur, dass sie zu einer Belegung eine passende Multimenge liefern, d.h. eine Multimenge über der Farbe des angrenzenden Platzes. Entsprechendes gilt für Guards. Wichtig ist hier, dass sie zu einer Belegung einen Wahrheitswert liefern, der über die Zulässigkeit der Belegung entscheidet. Im übrigen ist die Definition eines gefärbten Netzes derjenigen eines kantenkonstanten Netzes ähnlich.

Definition 7.6 *Sei* $Var := \{x_1, x_2, x_3, \ldots\}$ *eine Menge von Variablen mit Wertebereichen* $dom(x)$ *für jedes* $x \in Var$. *Eine* Belegung *ist eine Zuordnung (Abbildung)* $\beta = [x_1 = u_1, x_2 = u_2, x_3 = u_3, \ldots]$ *von Werten* $u_i \in dom(x_i)$ *zu den Variablen.*

Definition 7.7 *Ein* gefärbtes Petrinetz *(coloured Petri net, CPN) wird als Tupel* $\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, Var, Guards, \widehat{W}, \mathbf{m}_0 \rangle$ *definiert, wobei gilt:*

- (P, T, F) *ist ein endliches Netz (Def. 3.1),*
- $\mathcal{C}$ *ist eine Menge von* Farbenmengen,
- $cd\colon P \to \mathcal{C}$ *ist die* Farbzuweisungsabbildung *(colour domain mapping). Sie wird durch* $cd\colon F \to \mathcal{C}$, $cd(x,y) := \textbf{if } x \in P \textbf{ then } cd(x) \textbf{ else } cd(y) \textbf{ fi}$ *auf* F *erweitert.*
- Var *ist eine Menge von Variablen mit Wertebereichen* $dom(x)$ *für jedes* $x \in Var$.
- $Guards = \{guard_t \mid t \in T\}$ *ordnet jeder Transition* $t \in T$ *ein Prädikat* $guard_t$ *mit Variablen aus* Var *zu.*
- $\widehat{W} = \{W_\beta \mid \beta$ *ist Belegung von* $Var\}$ *ist eine Menge von Kantengewichtungen der Form* $W_\beta : F \to Bag(\bigcup \mathcal{C})$, *wobei* $W_\beta(x,y) \in Bag(cd(x,y))$ *für alle* $(x,y) \in F$ *gilt.*
- $\mathbf{m}_0 : P \to Bag(\bigcup \mathcal{C})$ *mit* $\mathbf{m}_0(p) \in Bag(cd(p))$ *für alle* $p \in P$ *ist die Anfangsmarkierung.*

Beispiel: Für das gefärbte Netz 7.8 ist beispielweise $cd(p_4) = ready = \{rsa, rsb\}$, $cd((p_4, t_3)) = cd(p_4) = \{rsa, rsb\}$, $W_\beta(p_1, t_1) = 1'a \in Bag(cd(p_1)) = Bag(\{a, b\})$ für $\beta = [x = a, y = rsa]$ und $\mathbf{m}_0(p_1) = 1'a + 1'b \in cd(p_1)$, $guard_{t_1} = (x = a \wedge y = rsa) \vee (x = b \wedge y = rsb)$

Definition 7.8 *a) Die Markierung eines gefärbten Netzes (CPN)* $\mathcal{N} = \langle P, T, F, \mathcal{C}, cd, Var, Guards, \widehat{W}, \mathbf{m}_0 \rangle$ *ist ein Vektor* $\mathbf{m}$ *mit* $\mathbf{m}(p) \in Bag(cd(p))$ *für jedes* $p \in P$ *(auch als Abbildung* $\mathbf{m} : P \to Bag(\bigcup \mathcal{C})$ *mit* $\mathbf{m}(p) \in Bag(cd(p))$ *für jedes* $p \in P$ *aufzufassen).*

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

c) $\widetilde{W}_\beta(x,y) := \begin{cases} W_\beta(x,y) & \text{falls } (x,y) \in F \\ 0 & \text{sonst} \end{cases}$
ist wieder die Erweiterung von W_β *auf* $(P \times T) \cup (T \times P)$.

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

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

$\mathbf{m} \xrightarrow{t,\beta} \mathbf{m}' \Leftrightarrow \mathbf{m} \geq W_\beta(\bullet, t) \wedge \mathbf{m}' = \mathbf{m} - W_\beta(\bullet, t) + W_\beta(t, \bullet)$. *Dabei sind die Multimengenoperatoren komponentenweise auf Vektoren zu erweitern.*

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

Beispiel: Für das Netz 7.8 ist die Anfangsmarkierung (als Vektor dargestellt) $\mathbf{m}_0 = (a+b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset)$. Für $\beta = [x = a, y = rsa]$ und $W_\beta(\bullet, t_1) = (a, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ ist die Transition t_1 in $\mathbf{m}_0$ aktiviert und die Nachfolgemarkierung ist $\mathbf{m}' = \mathbf{m}_0 + W_\beta[t_1, \bullet] - W_\beta(\bullet, t_1) = (a+b, \emptyset, \emptyset, \emptyset, \emptyset, s, \emptyset) + (\emptyset, a, \emptyset, rsa, \emptyset, \emptyset, \emptyset) - (a, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) = (b, a, \emptyset, rsa, \emptyset, s, \emptyset)$. Mit $a + b$ bzw rsa sind hier beispielsweise die Multimengen $\{a, b\}_b$ bzw. $\{rsa\}_b$ als formale Summen dargestellt.

Die Schaltregel wird in Abb. 7.6 an einem abstrakten Beispiel erläutert:

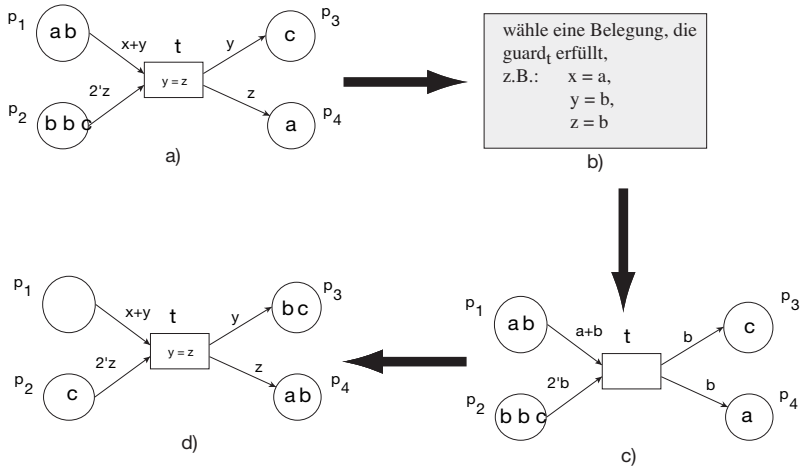


Abbildung 7.6: Schaltregel für gefärbte Netze

1. Wähle (wie in b)) eine Belegung β für Var(t) mit $guard_t(\beta) = true$.
2. Werte mit dieser Belegung die Ausdrücke an den Kanten zu Multimengen aus (wie in c).
3. Wende die Schaltregel für kantenkonstante Netze (Abb. 7.4) an (wie in d)).

Eine *Markierungs–Schaltfolge* für das gefärbte Netz von Abb. 7.8 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.

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

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

Definition 7.9 *Die Nachfolgemarkierungsrelation von Definition 7.8 wird wie üblich auf Wörter über T erweitert:*

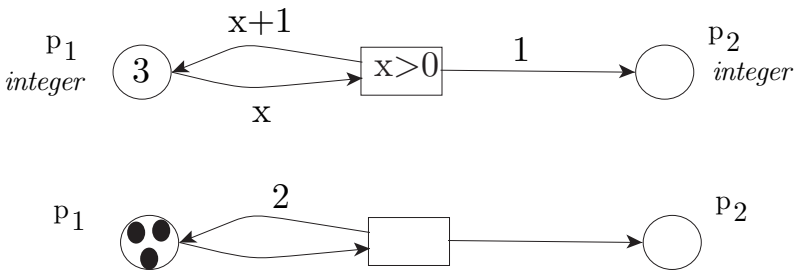


Abbildung 7.7: Gefärbtes Netz und P/T-Netz mit ähnlichem Verhalten

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

Die Menge $\mathbf{R}(\mathcal{N}) := \{\mathbf{m} | \exists w \in T^* : \mathbf{m_0} \xrightarrow{w} \mathbf{m}\}$ *ist die Menge der* erreichbaren *Markierungen oder auch* Erreichbarkeitsmenge. *Eine Transitionsfolge* $w \in T^*$ *heißt aktiviert in* $\mathbf{m}$ *(in Zeichen:* $\mathbf{m} \xrightarrow{w}$*), falls* $\exists \mathbf{m}_1 : \mathbf{m} \xrightarrow{w} \mathbf{m}_1$ *und* $FS(\mathcal{N}) := \{w \in T^* \mid \mathbf{m_0} \xrightarrow{w}\}$ *ist die Menge der* Schaltfolgen *(firing sequence set) von* $\mathcal{N}$.

Der Erreichbarkeitsgraph eines gefärbten Netzes $\mathcal{N}$ *ist ein Tupel* $RG(\mathcal{N}) := (Kn, Ka)$ *mit Knotenmenge* $Kn := \mathbf{R}(\mathcal{N})$ *(siehe Def. 7.9) und Kantenmenge* $Ka := \{(\mathbf{m}_1, (t, \beta), \mathbf{m}_2) \mid \mathbf{m}_1 \xrightarrow{t,\beta} \mathbf{m}_2\}$ *(vergl. Def. 7.8) (als Beispiel siehe Abb. 7.8).*

Wie in der Anmerkung auf Seite 227 ist der Erreichbarkeitsgraph als Transitionssystem (ohne Endzustände) aufzufassen.

Vergleiche das gefärbte Netz mit dem P/T-Netz in Abb. 7.7. Beide stellen einen Zähler dar.

Kantenausdrücke von gefärbten Netzen können auch speziell definierte Funktionen enthalten. Diese werden wie in Abb. 7.9 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.

Nachdem nun gefärbte Netze eingeführt sind, betrachten wir wieder die Abbildungen 7.27 und 7.28 (auf den Seiten 243 und 244). Im unteren Teil ist die Vergröberung von Transitionen zu erkennen. Dadurch wird auch graphisch deutlich, dass den Übergängen von einfachen Netzen über kantenkonstante Netze zu P/T-Netzen bzw. gefärbten Netzen die Zusammenfassung (Vergröberung) von zunächst Plätzen und dann Transitionen entspricht.

7.3 Das RENEW-Werkzeug

Das am Arbeitsbereich TGI in Hamburg entwickelte Werkzeug RENEW [KWD] ermöglicht die graphische Darstellung der hier behandelten Netzmodelle. Eine Werkzeugleiste ist in der Abb. 7.10 dargestellt. RENEW kann auch für die Ausführung der erzeugten Netze benutzt werden. Als Beschriftungssprache wurden Elemente der Programmsprache 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.

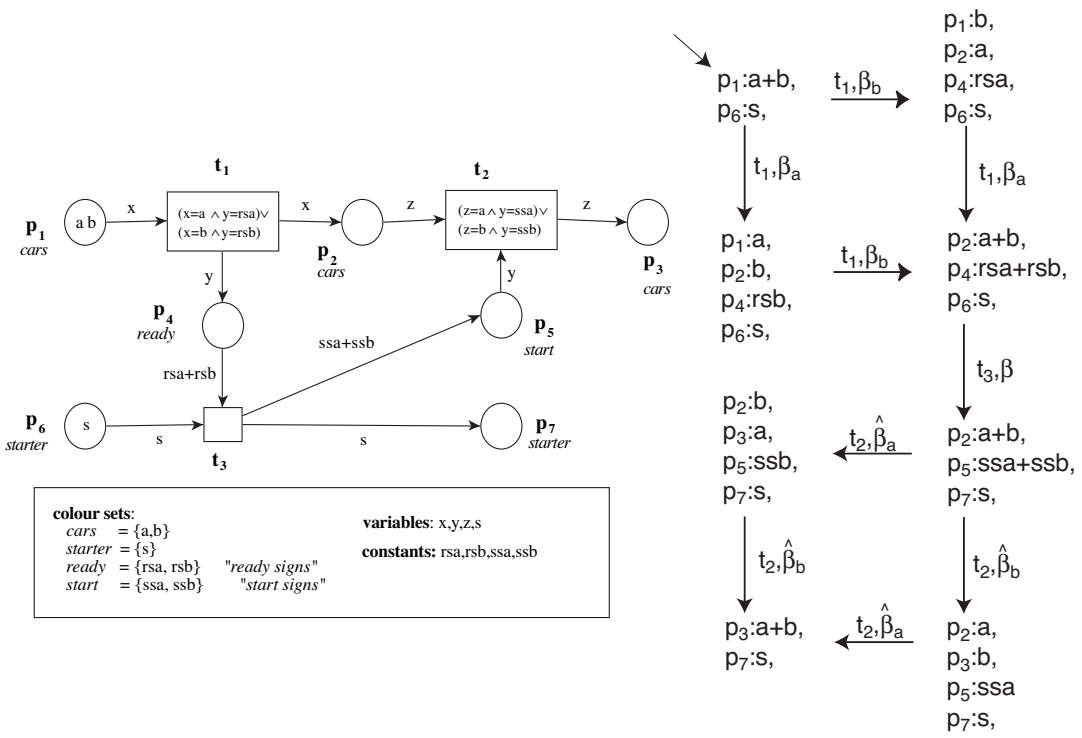


Abbildung 7.8: Gefärbtes Netz $\mathcal{N}_5$ mit Guards und sein Erreichbarkeitsgraph

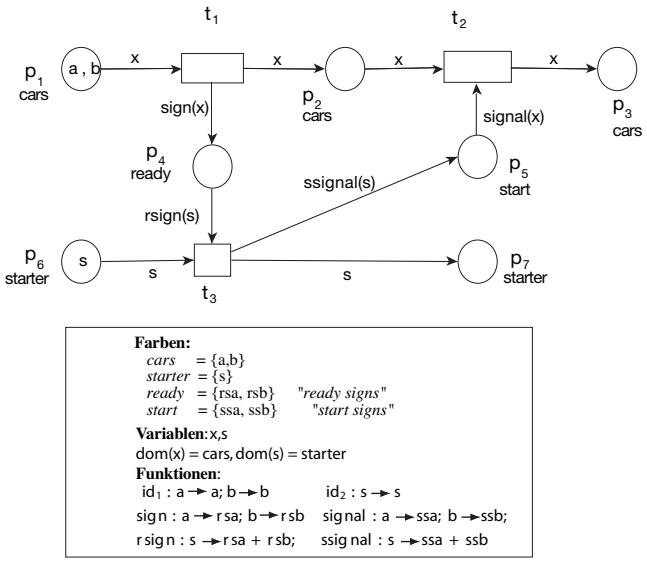


Abbildung 7.9: Gefärbtes Netz mit definierten Funktionen

Abbildung 7.10: Werkzeugleiste des RENEW-Werkzeugs

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 Zeitschranken für das Schalten benutzt werden. Um etwas über andere Petrinetz-Tools zu erfahren, siehe die Internetseite *The Petri Nets World* mit der URL [Pet]. Über sie ist auch Information zu Literatur zu Petrinetzen, Forschungsgruppen und -projekte zugänglich.

Die Abbildung 7.11 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 gemeisamen 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 7.12. Geschäftsleute werden durch *individuelle Marken* `"A"` und `"B"` dargestellt, ebenso wie ihr jeweiliges Guthaben im Platz `money`. Wird eine Darstellung mit Bezeichner und Wert gewünscht, so kann man

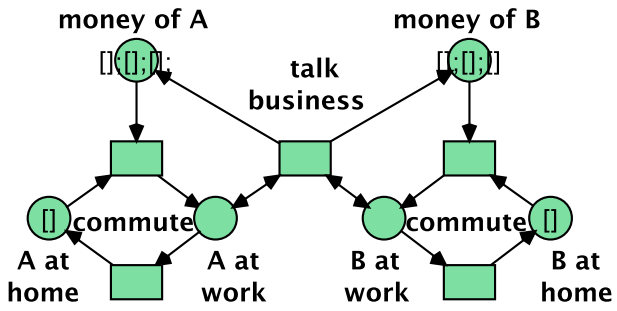


Abbildung 7.11: Das Leben zweier Geschäftsleute

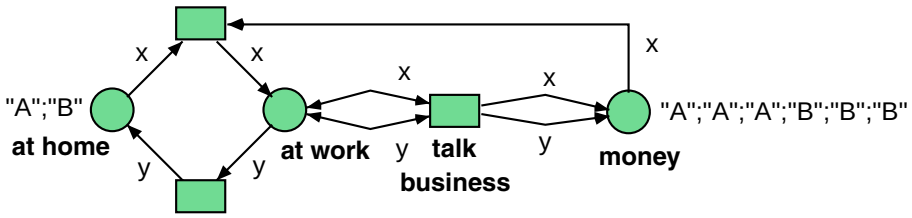


Abbildung 7.12: Faltung von Netz 7.11 zu einem gefärbten Netz

wie in Abb. 7.14 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.

Ein ernst zu nehmender Einwand gegen das gefärbte Netz aus Abb. 7.13 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. 7.15 bereiten wir eine Teilung des Systems aus Abb. 7.13 vor. Zwei der Transitionen werden doppelt in verschiedenen Teilen des Netzdiagramms 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. 7.16 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.

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.

Weil wir textuelle Anschriften für die Kanäle verwendet haben, können wir mehrere Synchronisationen gleichzeitig anfordern, ohne dass das Diagramm seine Klarheit verliert. In Abb. 7.17 wurde das Netz aus Abb. 7.16 so umstrukturiert, dass von der Transition `talk business` zwei Aufrufe des Kanals `deposit` getätigt werden. Dies veranlasst die Transition `deposit`, zweimal zu schalten, was das gewünschte Verhalten in unserem Beispiel ist.

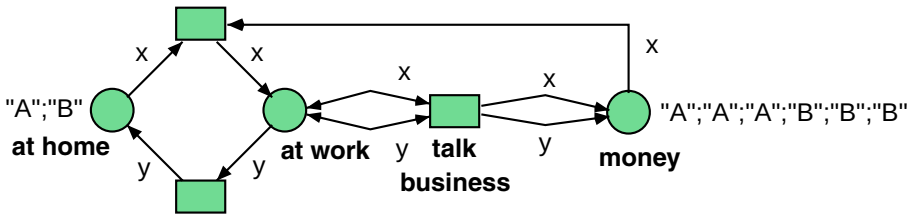


Abbildung 7.13: Faltung von Netz 7.11 zu einem gefärbten Netz

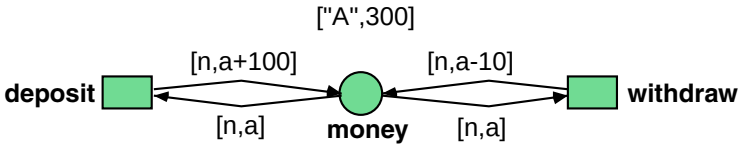


Abbildung 7.14: Ein nicht ganz so einfaches Bankkonto

Jetzt können wir die Lösung aus Abb. 7.17 mit dem gefärbten Netz zur Modellierung eines Bankkontos aus Abb. 7.14 kombinieren und jedes Konto als Wertepaar repräsentieren. Abb. 7.18 zeigt das Resultat. Beachtenswert ist, wie die Anzahl der Geldeinheiten, die ein- oder auszuzahlen ist, als zweiter Parameter über den Kanal übergeben wird. Gegenüber gewöhnlichen Petrinetzen fügen synchrone Kanäle die Fähigkeit hinzu, über gleichzeitige, nicht nur über aufeinanderfolgende oder über nebenläufige Handlungen zu sprechen (Rendez-Vous-Synchronisation).

Üblicherweise werden die beiden Netze in zwei verschieden Fenstern wie in Abbildung 7.19 dargestellt. Die Abbildung 7.20 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[1]. 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.

[1] Dadurch wird die Bindung zu $[x = "B", a = 3, acc = account[5]]$ erweitert.

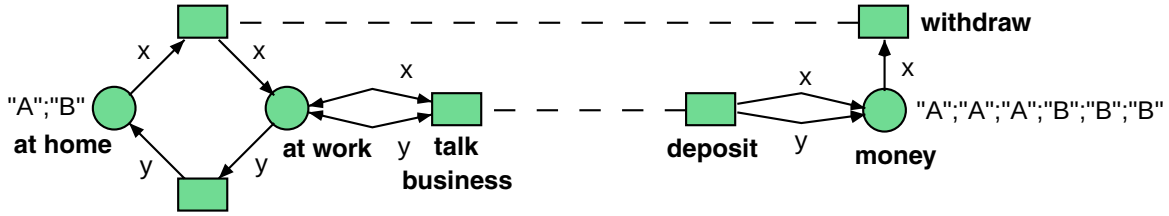


Abbildung 7.15: Personen und Konten werden geteilt

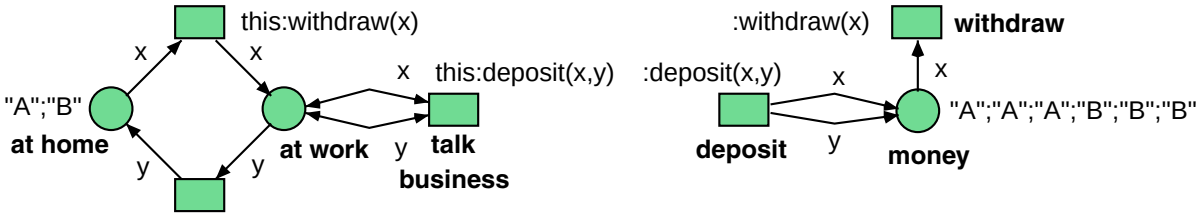


Abbildung 7.16: Textuelle Anschriften kennzeichnen Synchronisation

7.4 Zwei Anwendungsbeispiele

In den folgenden Abschnitten werden zwei Anwendungsbeispiele behandelt. Das *Alternierbitprotokoll* steht für die gesamte Klasse der Kommunikationsprotokolle. Diese Klasse ist stark aktionsorientiert. Das entwickelte Netz stellt das Bit und die Daten als Marken eines gefärbten Netzes dar. Das Modell ist sehr viel kleiner als die entsprechende Darstellung durch Transitionssysteme in Kapitel 1. Das Beispiel der *Datenbank-Manager* ist schließlich ein schönes Beispiel eines gefärbten Netzes mit einer größeren Anzahl von gefärbten Marken.

ht

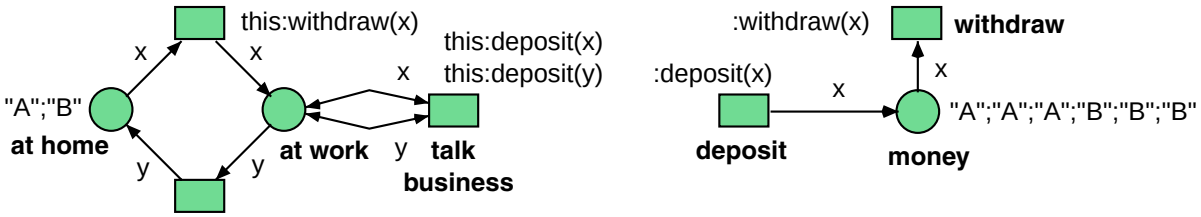


Abbildung 7.17: Wiederverwendung eines Kanals

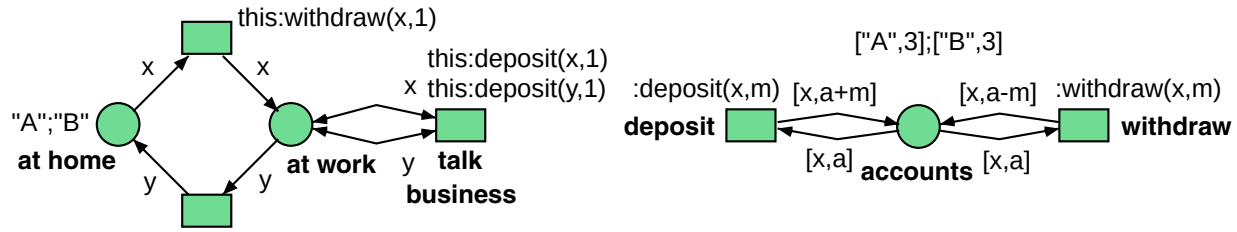


Abbildung 7.18: Buchführung mit Algebra

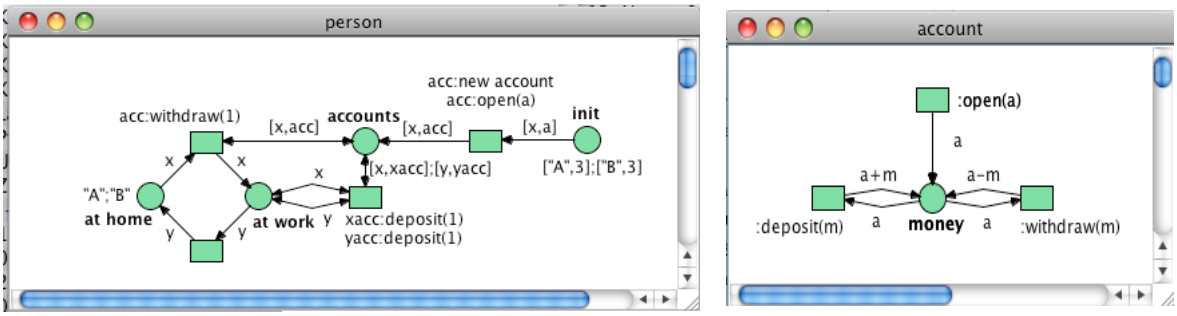


Abbildung 7.19: Buchführung mit Algebra in zwei Fenstern

7.4.1 Das Alternierbitprotokoll

In diesem Beispiel wird das *Alternierbitprotokoll* als gefärbtes Netz modelliert[2]. 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 7.21 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 $1, 2, 3, \ldots$. 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 Transittionen `a`, `h` und `b` korrekt übermittelt, damit sie eine Funktionseinheit `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,

[2]Im Abschnitt 1.13 auf Seite 13 wurde es als Transitionssystem, d.h. über seinen Zustandsgraphen entwickelt. Hier geschieht dies über die Programm- oder Steuer-Struktur, die i.A. sehr viel kleiner in ihrem Datenvolumen ist.

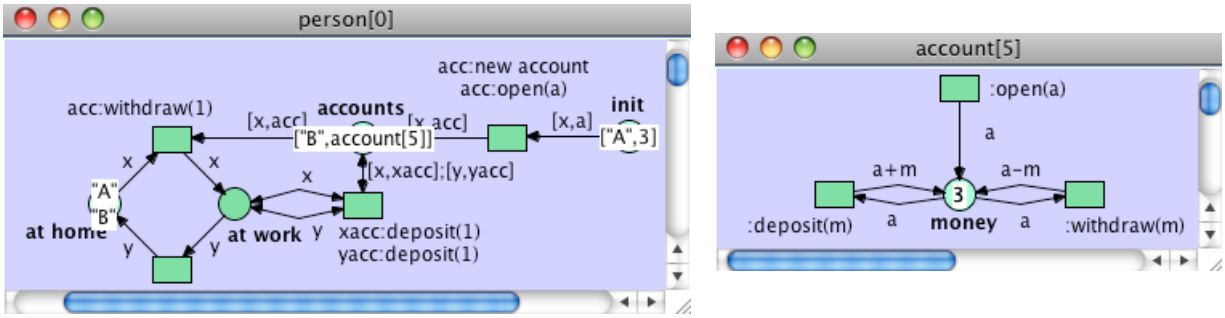


Abbildung 7.20: Buchführung mit Algebra in zwei Fenstern in Simulation

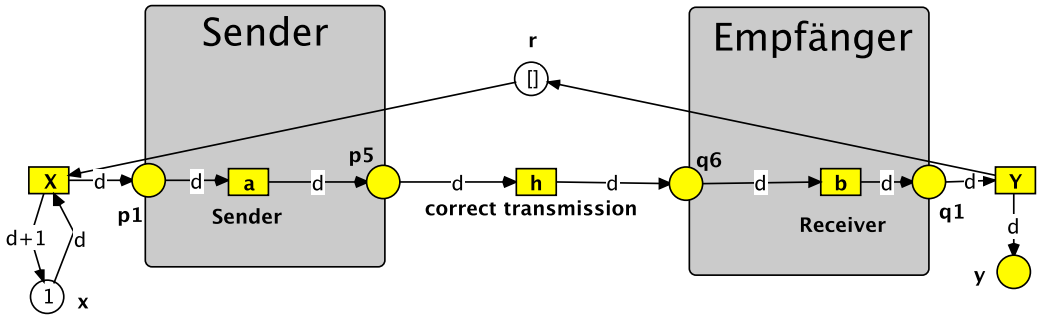


Abbildung 7.21: Spezifikation abp-1 des Alternierbitprotokolls

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 nicht betrachtet wird. Das Netz abp-1 kann also als Spezifikation der Aufgabenstellung angesehen werden.

Im Netz abp-2 (Abb. 7.22) 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 Protokollschichten 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 endgültigen Lösung wird mit abp-4 ein zu abp-3 verhaltensäquivalentes Protokoll eingeführt. Dem Datum `d` wird durch die Transition `a` ein Bit `x` beigefügt, das anfangs den Wert 1 hat. Bei fehlerfreier Übermittlung schaltet die Transition `e`, da `q4` 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

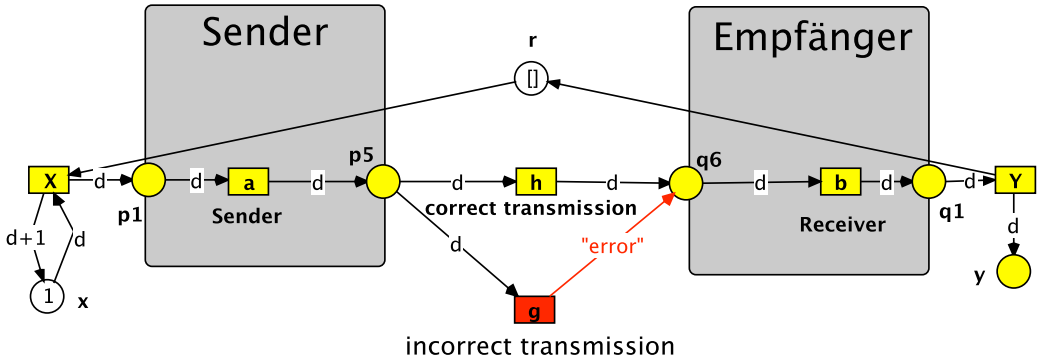


Abbildung 7.22: Übermittlung mit Fehlern: Netz abp-2

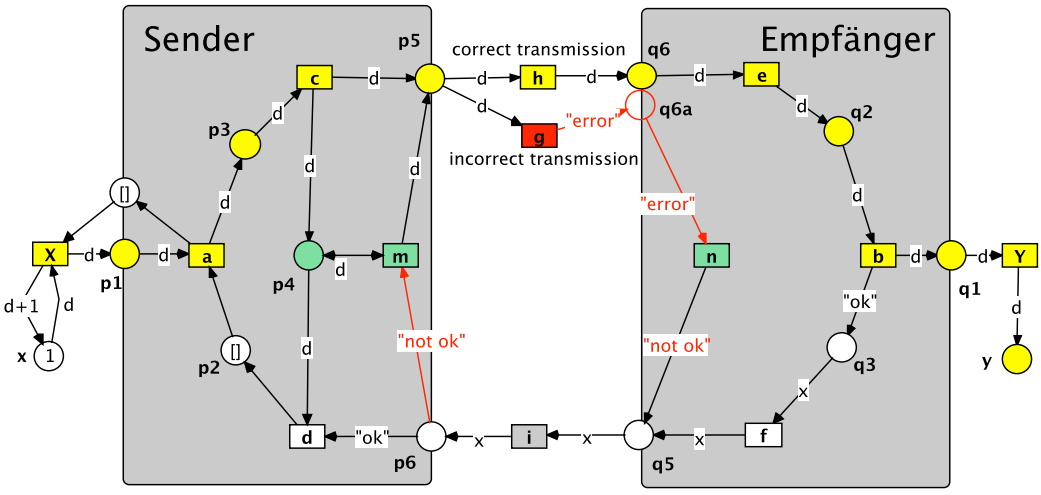


Abbildung 7.23: Übermittlung mit Quittung: Netz abp-3

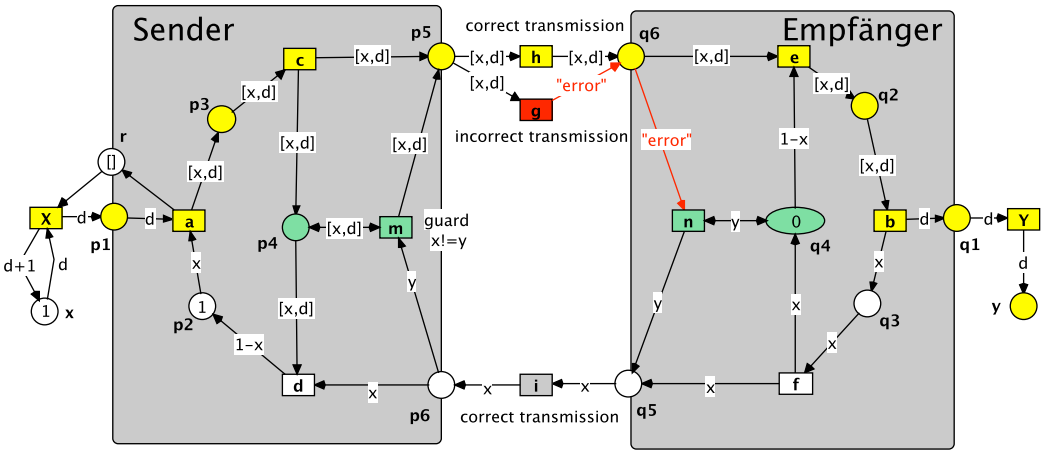


Abbildung 7.24: Übermittlung mit Quittung durch Alternierbit: Netz abp-4

`x` verschieden ist (`guard x!=y`). Um die nächste Datenübermittlung einzuleiten, wird das Ausgangsbit komplementiert nach `p2` gelegt. Der Vorgang wiederholt sich mit dem jeweils komplementären Bitwert.

Nun ist es nur noch ein kleiner Schritt zur endgültigen Lösung. Im Netz abp-5 (Abb. 7.25) ist nun auch im Quittungs-Kanal eine Störung möglich. Um den Integer-Test `guard x!=y` beibehalten 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 verdopplete Ankunft beim Empfänger erfolgen. Dies kann der Empfänger jedoch daran erkennen, dass das Bit nicht gewechselt hat. Die verdopplte Sendung wird durch den Guard `x==y|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 im Kapitel 4 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 verfiziert.

7.4.2 Das Beispiel der Datenbank-Manager

Eine Menge von $n > 0$ Datenbanken soll von n Prozessen, genannt *Datenbank-Manager*, $DBM = \{d_1, d_2, \ldots, d_n\}$ 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[3].

[3] nach [Jen87]

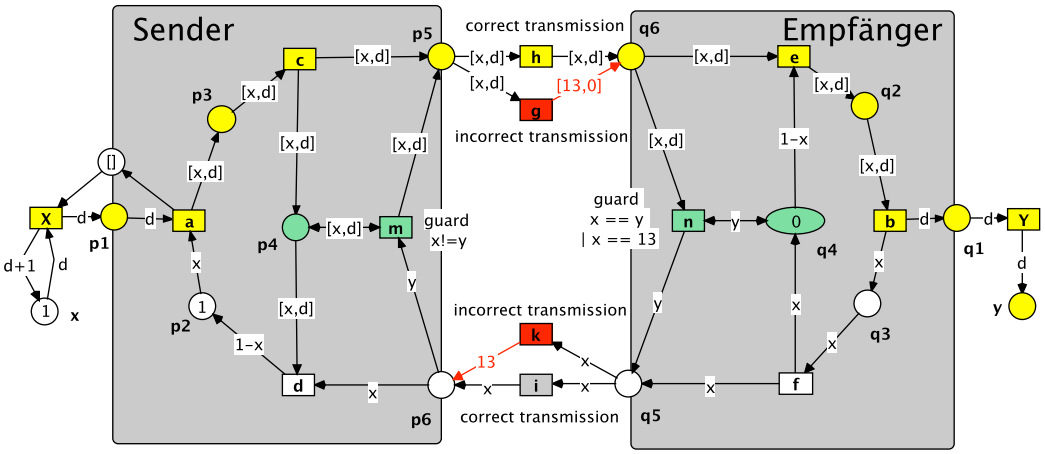


Abbildung 7.25: Das Alternierbitprotokoll abp-5

Dabei sollen weder die Datenbanken noch ihre Aktualisierung dargestellt werden, sondern nur der Nachrichtenaustausch.

Zustände der Manager : *inactive*, *waiting*, *performing*

Nachrichten : $MS = \{(s,r) | s,r \in DBM \wedge sr\}$

Zustände der Nachrichtenkanäle : *unused*, *sent*, *received*, *acknowledged*

wechselseitiger Ausschluss : *exclusion*

Spezifikation für das gefärbte Netz von Abb. 7.26:

Farben : $DBM = \{d_1, d_2, \ldots, d_n\}$

$MS = \{(s,r) | s,r \in DBM \wedge sr\}$

$E = \{e\}$

Variablen : $Var = \{e, r, s\}$

$dom(e) = E, \quad dom(r) = dom(s) = DBM$

Funktionen :

$MINE : DBM \rightarrow Bag(MS) \qquad MINE(s) := \sum_{r \neq s} (s,r)$

$REC : MS \rightarrow DBM \qquad REC((s,r)) := r$

$ABS : DBM \rightarrow E \qquad ABS(s) := e$

$$\textit{Anfangsmarkierung}: \mathbf{m}_0(p) := \begin{cases} DBM & \text{falls } p = inactive \\ MS & \text{falls } p = unused \\ \{e\} & \text{falls } p = exclusion \\ \emptyset & \text{sonst} \end{cases}$$

Nicht alle Funktionen dieser Spezifikation werden in der graphischen Darstellung von Abb. 7.26 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

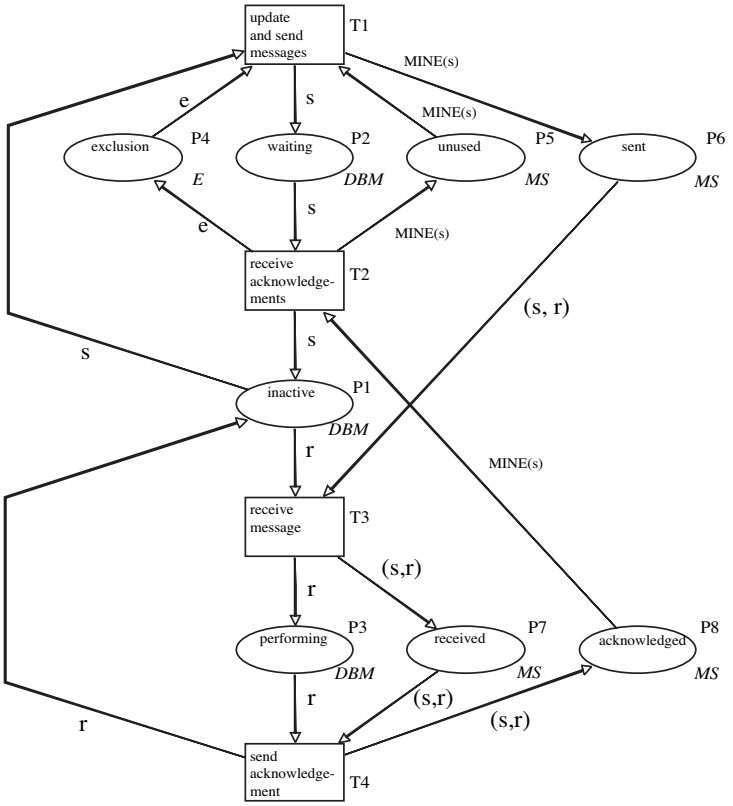


Abbildung 7.26: Datenbank-Manger

7.7 von gefärbten Netzen zu definieren. Beispielweise ergibt sich für $(x, y) = (T2, unused)$ und die Belegung $\beta = [s = d_1]$ der Wert $W_\beta(T2, unused) = MINE(d_1)$.

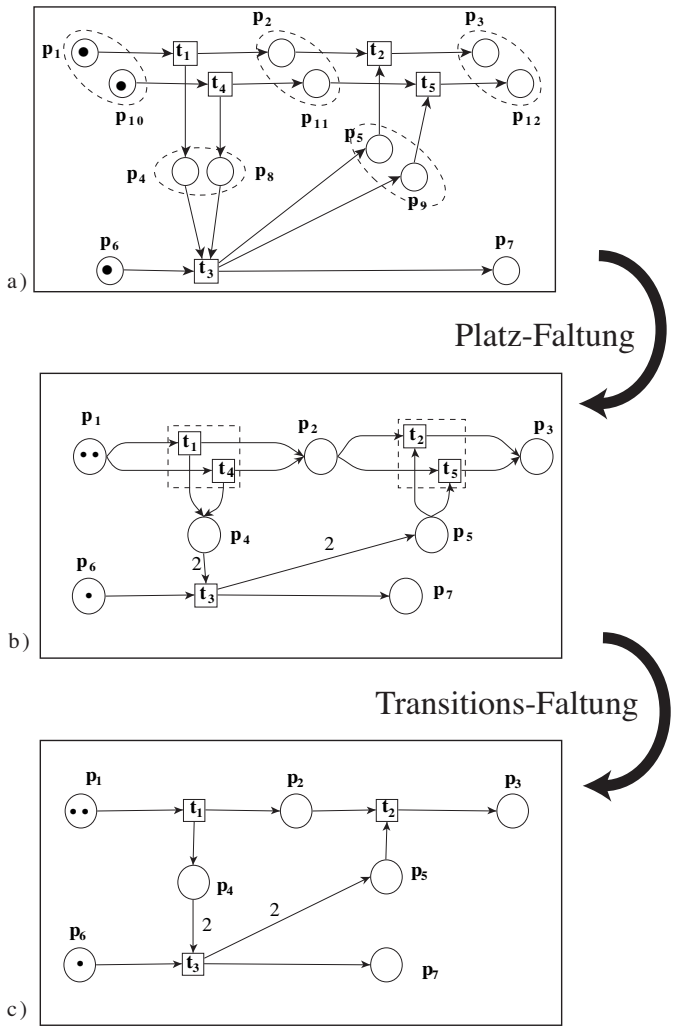


Abbildung 7.27: Netzfaltung zu Platz/Transitions-Netzen

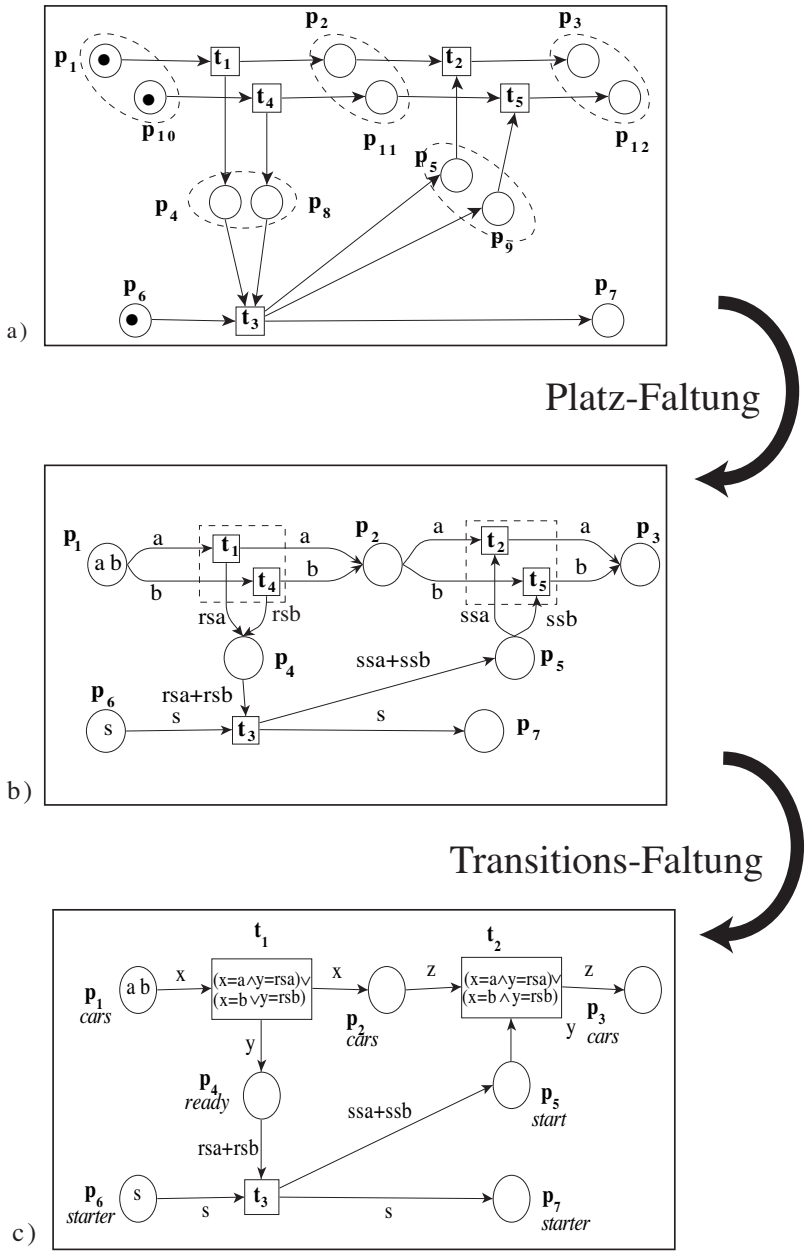


Abbildung 7.28: Netzfaltung zum kantenkonstanten und gefärbten Netz

Anhang A

Referenzen auf englische Literatur

Hier werden Referenzen auf Texte gegeben, die in englischer Sprache den Abschnitten und Kapiteln dieses Skriptes entsprechen. Dies ist eine Minimalangabe. Weitergehende Literatur ist in den Kapiteln zu finden.

Kapitel 1: Transitionssysteme und Verifikation

[BK08]

[CGP99] chapters 2 - 5

[HR04]

Kapitel 2: Partielle Ordnungen

http://mathworld.wolfram.com/StrictOrder.html

[AW98] chapter 6

Kapitel 3: Platz/Transitionsnetze und Verifikation

[GV03] chapters 2 - 5

[Jen92] the data base managers example (pages 21 ff)

[Jen94] place invariants (pages 113 ff)

Kapitel 4: Parallele Algorithmen

[Gru97] chapter 4

[Jáj92]

Kapitel 5: Prozessalgebra

[Fok99] chapters 2 - 5, 6.1

Kapitel 6: Konsistenz

[AH02]

[EGLT76]

Kapitel 7: Höhere Petrinetze

[GV03] chapters 2 - 4

[JK09]

[Kum01] reference nets

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. Bèrard, 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.

[BR81] E. Best and B. Randell. A Formal Model of Atomicity in Asynchronous Systems. *Acta Informatica*, 16:93–124, 1981.

[BS69] K.A. Bartlett and R.A. Scantlebury. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM*, 12:260–261, 1969.

[BSW79] P.A. Bernstein, D.W. Shipman, and W.S. Wong. Formal Aspects in Serializability in Database Concurrency Control. *IEEE SE-5*, 3:203–216, 1979.

[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.

[CDH92] S. Christensen and N. Damgaard Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. Technical Report DAIMI PB–390, Aarhus University, 1992.

[CDK02] G. Coulouris, J. Dollimore, and T. Kindberg. *Verteilte Systeme: Konzepte und Design*. Pearson Studium, 2002.

[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.

[Dij77] E.W. Dijkstra. A Correctness Proof for Communicating Processes - A small Exercise. Technical report, EDW-604, 1977.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. ACM*, 19:624–633, 1976.

[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.

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[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. Hauschildt and R. Valk. Safe States in Banker-like Resource Allocation Problems. *Information and Computation*, 75:232–263, 1987.

[Jáj92] J. Jájá. *An introduction to parallel algorithms*. Addison-Wesley Publ. Co., 1992.

[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.

[Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[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.

[Pap79] C.H. Papadimitriou. The Serializability of Concurrent Data Base Updates. *J. ACM*, 26(4):631–653, 1979.

[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*. Leitfäden der Informatik. Vieweg+Teubner, 15 July 2010. 248 pages; ISBN 978-3-8348-1290-2.

[Set82] R. Sethi. Useless Actions Make a Difference: Strict Serializability of Database Updates. *J. ACM*, 29(2):349–403, 1982.

[uMS95] A. S. Tanenbaum und M. van Steen. *Verteilte Systeme: Prinzipien und Paradigmen*. Pearson Studium, 1995.

[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