Die Axiome für die τ-Aktion und Abstraktion ergeben:

$$\begin{aligned}
& \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
= \;& \tau_{\{c_3\}}(r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2)) \\
= \;& r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 Q_1) \| Q_2)) \\
= \;& r_1 \cdot \tau_{\{c_3\}}(c_3 \cdot \partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2))) \\
= \;& r_1 \cdot \tau \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2))) \\
= \;& r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2)))
\end{aligned}$$

Weiter rechnen wir:

$$\begin{aligned}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2))) \;=\; & \\
& r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 Q_1) \| (s_2 Q_2))) \\
& + s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 Q_1) \| (s_2 Q_2))) \;=\; & \\
& s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2)))
\end{aligned}$$

Damit erhalten wir als Lösung für die lineare rekursive Spezifikation E für einen Puffer der Kapazität 2:

$$\begin{aligned}
X &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_2 \| Q_1)) \\
Y &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(Q_1 \| (s_2 Q_2))) \\
Z &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 Q_1) \| (s_2 Q_2)))
\end{aligned}$$

9.5.2 Die Fairness-Regel

Es ist möglich, durch Abstraktion τ-Schleifen zu konstruieren: So führt $\tau_{\{a\}}(\langle X \,|\, X{=}aX\rangle)$ unendlich lange die τ-Aktion aus.

Definition 9.59 *Sei E eine geschützte lineare rekursive Spezifikation, C eine Teilmenge ihrer Variablen und $I \subset A$ eine Menge von Aktionen.*

- *C heißt* (Fairness-)Gruppe *(cluster) für I, falls für je zwei Rekursionsvariablen X und Y in C $\langle X|E\rangle \xrightarrow{b_1} \cdots \xrightarrow{b_m} \langle Y|E\rangle$ und $\langle Y|E\rangle \xrightarrow{c_1} \cdots \xrightarrow{c_n} \langle X|E\rangle$ 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 $\{v_1 Y_1, \ldots, v_m Y_m, w_1, \ldots, w_n\}$ ist, dann gilt

$$\tau \cdot \tau_I(\langle X|E\rangle) = \tau \cdot \tau_I(v_1\langle Y_1|E\rangle + \cdots + v_m\langle Y_m|E\rangle + w_1 + \cdots + w_n)$$

Zur Erläuterung von CFAR sei E das Gleichungssystem:

$$\begin{aligned} X_1 &= aX_2 + b_1 \\ &\vdots \\ X_{n-1} &= aX_n + b_{n-1} \\ X_n &= aX_1 + b_n \end{aligned}$$

$\tau_{\{a\}}(\langle X_1|E\rangle)$ führt τ-Transitionen aus, bis eine Aktion b_i für $i \in \{1, \ldots, n\}$ 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:

$$\tau_{\{a\}}(\langle X_1|E\rangle) \underline{\leftrightarrow}_{rb} b_1 + \tau(b_1 + \cdots + b_n)$$

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 9.5 die mit f bezeichneten Transitionen b_1, b_2 und b_3 fair schalten (Definition 6.35 b)), d.h. der Zyklus der mit a bezeichneten Transitionen einmal verlassen wird.

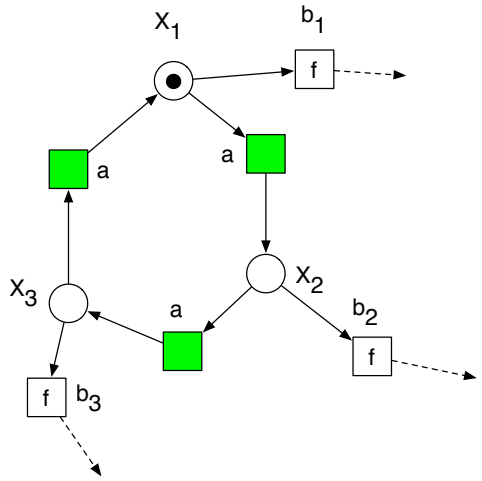


Abbildung 9.5: Faire Abstraktion als P/T-Netz mit fair schaltenden Transitionen.

Beispiel 9.60

$$X = heads \cdot X + tails$$

$\langle X|E\rangle$ 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:

$$\begin{aligned} \tau \cdot \tau_{\{heads\}}(\langle X|E\rangle) &= \tau \cdot \tau_{\{heads\}}(tails) \\ &= \tau \cdot tails \end{aligned}$$

und

$$\begin{aligned}\tau_{\{heads\}}(\langle X|E\rangle) &= \tau_{\{heads\}}(heads \cdot \langle X|E\rangle + tails)\\ &= \tau \cdot \tau_{\{heads\}}(\langle X|E\rangle) + tails\\ &= \tau \cdot tails + tails\end{aligned}$$

Anmerkung: Der Kalkül ACP_τ mit Abstraktion, geschützter linearer Rekursion und Fairnessregel ist korrekt und vollständig in Bezug auf initiale Verzweigungsbisimulation:

$$s = t \;\Leftrightarrow\; s \leftrightarrows_{rb} t$$

9.6 Verifikation des Alternierbitprotokolls

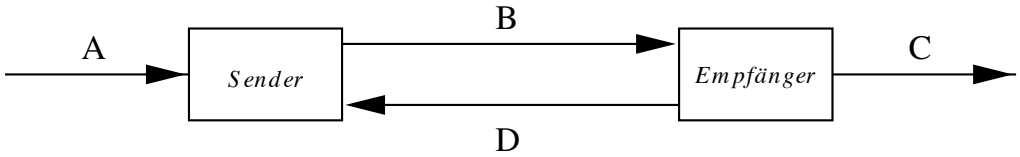
Es wird nun mit den Mitteln der Prozessalgebra ein Korrektheitsbeweis für das Alternierbitprotokoll geführt.

Es sollen Daten über einen FIFO-Kanal gesendet werden. Dazu wird ein Datum am Kanal A eingelesen: $r_{\mathrm{A}}(d)$ und dann vom Kanal C gesendet: $s_{\mathrm{C}}(d)$. Die Kanalkapazität beträgt $n = 1$, so dass jedes Datum erst bei C ausgeliefert werden muss, bevor bei A ein neues akzeptiert werden kann.

Die Spezifikation des gewünschten externen Verhaltens ist:

$$E = \left\{ X = \sum_{d \in \Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot X \right\} \tag{9.1}$$

Zunächst wird das Alternierbitprotokoll als Programm (als Sender S_b und Empfänger R_b) dargestellt und dann sein Verhalten als übereinstimmend mit der geforderten Spezifikation bewiesen. Dies erfolgt indirekt über die 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.

Betrachten wir nun die Implementation.

Spezifikation des Senders für das Senden mit Bit b:

$$\begin{aligned} S_b &= \sum_{d \in \Delta} r_{\mathrm{A}}(d) \cdot T_{db} \\ T_{db} &= (s_{\mathrm{B}}(d, b) + s_{\mathrm{B}}(\bot)) \cdot U_{db} \\ U_{db} &= r_{\mathrm{D}}(b) \cdot S_{1-b} \ + \ (r_{\mathrm{D}}(1-b) + r_{\mathrm{D}}(\bot)) \cdot T_{db} \end{aligned}$$

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:

$$\begin{aligned} R_b &= \sum_{d\in\Delta}\{r_{\mathrm{B}}(d,b)\cdot s_{\mathrm{C}}(d)\cdot Q_b \\ &\quad + r_{\mathrm{B}}(d,1-b)\cdot Q_{1-b}\} + r_{\mathrm{B}}(\bot)\cdot Q_{1-b} \\ Q_b &= (s_{\mathrm{D}}(b)+s_{\mathrm{D}}(\bot))\cdot R_{1-b} \end{aligned}$$

Als externes Verhalten des Alternierbitprotokolls erhalten wir also:

$$\tau_I(\partial_H(R_0\|S_0))$$

Dabei werden durch ∂_H falsche Kommunikationspaare ausgeschlossen, d.h. wir definieren

- $\gamma(s_B(d,b), r_B(d,b)) := c_B(d,b)$
- $\gamma(s_B(\bot), r_B(\bot)) := c_B(\bot)$
- $\gamma(s_D(b), r_D(b)) := c_D(b)$
- $\gamma(s_D(\bot), r_D(\bot)) := c_D(\bot)$

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:

$$\tau_I(\partial_H(R_0\|S_0)) = \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot s_{\mathrm{C}}(d)\cdot \tau_I(\partial_H(R_0\|S_0))$$

bzw. noch direkter:

$$\tau_I(\partial_H(R_0\|S_0)) \quad \text{ist Lösung von} \quad E = \Big\{X = \sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot s_{\mathrm{C}}(d)\cdot X\Big\} \tag{9.2}$$

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:

$$\begin{aligned} R_0\|S_0 &= \textstyle\sum_{d'\in\Delta}\{r_{\mathrm{B}}(d',0)\cdot((s_{\mathrm{C}}(d')Q_0)\|S_0) \\ &\quad + r_{\mathrm{B}}(d',1)\cdot(Q_1\|S_0)\} + r_{\mathrm{B}}(\bot)\cdot(Q_1\|S_0) \\ &\quad + \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot(T_{d0}\|R_0) \end{aligned}$$

Weiter erhält man mit D4, D1, D2, D5, A6, A7:

$$\partial_H(R_0\|S_0) = \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot\partial_H(T_{d0}\|R_0)$$

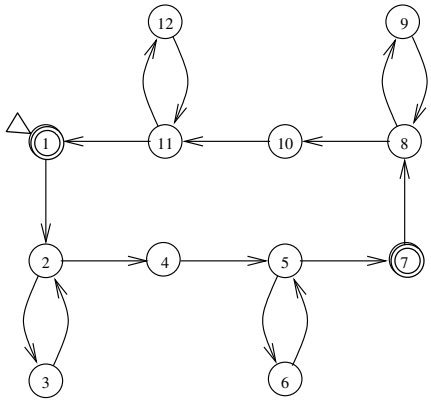


Abbildung 9.6: Transitionssystem von $\partial_H(R_0 \| S_0)$

Diese Äquivalenz entspricht dem Übergang vom Zustand 1 in den Zustand 2 im Transitions-system von $\partial_H(R_0 \| S_0)$ in Abb. 9.6.

$$\begin{aligned}
T_{d0} \| R_0 &= (s_{\mathrm{B}}(d,0) + s_{\mathrm{B}}(\bot)) \cdot (U_{d0} \| R_0) \\
&+ \textstyle\sum_{d' \in \Delta} \{ r_{\mathrm{B}}(d',0) \cdot ((s_{\mathrm{C}}(d') Q_0) \| T_{d0}) \\
&+ r_{\mathrm{B}}(d',1) \cdot (Q_1 \| T_{d0}) \} + r_{\mathrm{B}}(\bot) \cdot (Q_1 \| T_{d0}) \\
&+ c_{\mathrm{B}}(d,0) \cdot (U_{d0} \| (s_{\mathrm{C}}(d) Q_0)) + c_{\mathrm{B}}(\bot) \cdot (U_{d0} \| Q_1) \\
\partial_H(T_{d0} \| R_0) &= c_{\mathrm{B}}(d,0) \cdot \partial_H(U_{d0} \| (s_{\mathrm{C}}(d) Q_0)) \\
&\quad + c_{\mathrm{B}}(\bot) \cdot \partial_H(U_{d0} \| Q_1)
\end{aligned}$$

Diese Äquivalenz entspricht dem Übergang vom Zustand 2 in die Zustände 3 und 4 im Transitionsgraph von Abb. 9.6.
Entsprechend erhält man für die Übergänge bis zum Zustand 7:

$$\begin{aligned}
U_{d0} \| Q_1 &= r_{\mathrm{D}}(0) \cdot (S_1 \| Q_1) \\
&\quad + (r_{\mathrm{D}}(1) + r_{\mathrm{D}}(\bot)) \cdot (T_{d0} \| Q_1) \\
&\quad + (s_{\mathrm{D}}(1) + s_{\mathrm{D}}(\bot)) \cdot (R_0 \| U_{d0}) \\
&\quad + (c_{\mathrm{D}}(1) + c_{\mathrm{D}}(\bot)) \cdot (T_{d0} \| R_0) \\
\partial_H(U_{d0} \| Q_1) &= (c_{\mathrm{D}}(1) + c_{\mathrm{D}}(\bot)) \cdot \partial_H(T_{d0} \| R_0)
\end{aligned}$$

$$\begin{aligned}
U_{d0} \| (s_{\mathrm{C}}(d) Q_0) &= r_{\mathrm{D}}(0) \cdot (S_1 \| (s_{\mathrm{C}}(d) Q_0)) \\
&\quad + (r_{\mathrm{D}}(1) + r_{\mathrm{D}}(\bot)) \cdot (T_{d0} \| (s_{\mathrm{C}}(d) Q_0)) \\
&\quad + s_{\mathrm{C}}(d) \cdot (Q_0 \| U_{d0}) \\
\partial_H(U_{d0} \| (s_{\mathrm{C}}(d) Q_0)) &= s_{\mathrm{C}}(d) \cdot \partial_H(Q_0 \| U_{d0})
\end{aligned}$$

$$\begin{aligned}
Q_0 \| U_{d0} &= (s_{\mathrm{D}}(0) + s_{\mathrm{D}}(\bot)) \cdot (R_1 \| U_{d0}) \\
&+ r_{\mathrm{D}}(0) \cdot (S_1 \| Q_0) + (r_{\mathrm{D}}(1) + r_{\mathrm{D}}(\bot)) \cdot (T_{d0} \| Q_0) \\
&+ c_{\mathrm{D}}(0) \cdot (R_1 \| S_1) + c_{\mathrm{D}}(\bot) \cdot (R_1 \| T_{d0}) \\
\partial_H(Q_0 \| U_{d0}) &= c_{\mathrm{D}}(0) \cdot \partial_H(R_1 \| S_1) \\
&\quad + c_{\mathrm{D}}(\bot) \cdot \partial_H(R_1 \| T_{d0})
\end{aligned}$$

$$\begin{aligned}
R_1\|T_{d0} &= \sum_{d'\in\Delta}\{r_{\mathrm{B}}(d',1)\cdot((s_{\mathrm{C}}(d')Q_1)\|T_{d0}) \\
&\quad + r_{\mathrm{B}}(d',0)\cdot(Q_0\|T_{d0})\} + r_{\mathrm{B}}(\bot)\cdot(Q_0\|T_{d0}) \\
&\quad + (s_{\mathrm{B}}(d,0)+s_{\mathrm{B}}(\bot))\cdot(U_{d0}\|R_1) \\
&\quad + (c_{\mathrm{B}}(d,0)+c_{\mathrm{B}}(\bot))\cdot(Q_0\|U_{d0}) \\
\partial_H(R_1\|T_{d0}) &= (c_{\mathrm{B}}(d,0)+c_{\mathrm{B}}(\bot))\cdot\partial_H(Q_0\|U_{d0})
\end{aligned}$$

Dann erhält man für die Übergänge von Zustand 7 bis zum Zustand 1:

$$\partial_H(R_1\|S_1) = \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d)\cdot\partial_H(T_{d1}\|R_1)$$

$$\begin{aligned}
\partial_H(T_{d1}\|R_1) &= c_{\mathrm{B}}(d,1)\cdot\partial_H(U_{d1}\|(s_{\mathrm{C}}(d)\cdot Q_1)) \\
&\quad + c_{\mathrm{B}}(\bot)\cdot\partial_H(U_{d1}\|Q_0)
\end{aligned}$$

$$\partial_H(U_{d1}\|Q_0) = (c_{\mathrm{D}}(0)+c_{\mathrm{D}}(\bot))\cdot\partial_H(T_{d1}\|R_1)$$

$$\partial_H(U_{d1}\|(s_{\mathrm{C}}(d)Q_1)) = s_{\mathrm{C}}(d)\cdot\partial_H(Q_1\|U_{d1})$$

$$\begin{aligned}
\partial_H(Q_1\|U_{d1}) &= c_{\mathrm{D}}(1)\cdot\partial_H(R_0\|S_0) \\
&\quad + c_{\mathrm{D}}(\bot)\cdot\partial_H(R_0\|T_{d1})
\end{aligned}$$

$$\partial_H(R_0\|T_{d1}) = (c_{\mathrm{B}}(d,1)+c_{\mathrm{B}}(\bot))\cdot\partial_H(Q_1\|U_{d1})$$

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.

$$\begin{aligned}
\{\ X_1 &= \textstyle\sum_{d'\in\Delta} r_{\mathrm{A}}(d')\cdot X_{2d'} \\
X_{2d} &= c_{\mathrm{B}}(d,0)\cdot X_{4d} + c_{\mathrm{B}}(\bot)\cdot X_{3d} \\
X_{3d} &= (c_{\mathrm{D}}(1)+c_{\mathrm{D}}(\bot))\cdot X_{2d} \\
X_{4d} &= s_{\mathrm{C}}(d)\cdot X_{5d} \\
X_{5d} &= c_{\mathrm{D}}(0)\cdot Y_1 + c_{\mathrm{D}}(\bot)\cdot X_{6d} \\
X_{6d} &= (c_{\mathrm{B}}(d,0)+c_{\mathrm{B}}(\bot))\cdot X_{5d} \\
Y_1 &= \textstyle\sum_{d'\in\Delta} r_{\mathrm{A}}(d')\cdot Y_{2d'} \\
Y_{2d} &= c_{\mathrm{B}}(d,1)\cdot Y_{4d} + c_{\mathrm{B}}(\bot)\cdot Y_{3d} \\
Y_{3d} &= (c_{\mathrm{D}}(0)+c_{\mathrm{D}}(\bot))\cdot Y_{2d} \\
Y_{4d} &= s_{\mathrm{C}}(d)\cdot Y_{5d} \\
Y_{5d} &= c_{\mathrm{D}}(1)\cdot X_1 + c_{\mathrm{D}}(\bot)\cdot Y_{6d} \\
Y_{6d} &= (c_{\mathrm{B}}(d,1)+c_{\mathrm{B}}(\bot))\cdot Y_{5d} \\
&\mid d\in\Delta\ \}
\end{aligned}$$

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:

$$\begin{aligned} s_{\mathrm{C}}(d) \cdot \tau_I(\langle X_{5d}|E\rangle) &= s_{\mathrm{C}}(d) \cdot \tau_I(\langle Y_1|E\rangle) \\ r_{\mathrm{A}}(d) \cdot \tau_I(\langle Y_{2d}|E\rangle) &= r_{\mathrm{A}}(d) \cdot \tau_I(\langle Y_{4d}|E\rangle) \\ s_{\mathrm{C}}(d) \cdot \tau_I(\langle Y_{5d}|E\rangle) &= s_{\mathrm{C}}(d) \cdot \tau_I(\langle X_1|E\rangle) \end{aligned}$$

$$\begin{aligned} \tau_I(\langle X_1|E\rangle) &= \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot \tau_I(\langle X_{2d}|E\rangle) \\ &= \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot \tau_I(\langle X_{4d}|E\rangle) \\ &= \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot \tau_I(\langle X_{5d}|E\rangle) \\ &= \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot \tau_I(\langle Y_1|E\rangle) \end{aligned}$$

$$\tau_I(\langle Y_1|E\rangle) = \textstyle\sum_{d\in\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{C}}(d) \cdot \tau_I(\langle X_1|E\rangle)$$

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 9.61

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.

Anhang: Übersicht über die Axiome der Prozesskalküle

Zur Übersicht alle Axiome und daraus abgeleitete Regeln mit Seitenreferenz[2]. Wie üblich sei $v \in A$ und x, y, z seien Terme.

Bez.	Axiom			BPA	PAP	ACP	Rek.	ACP_τ	Skript
A1	$x + y$	$=$	$y + x$	x	x	x		x	S. 202
A2	$(x + y) + z$	$=$	$x + (y + z)$	x	x	x		x	S. 202
A3	$x + x$	$=$	x	x	x	x		x	S. 202
A4	$(x + y) \cdot z$	$=$	$x \cdot z + y \cdot z$	x	x	x		x	S. 202
A5	$(x \cdot y) \cdot z$	$=$	$x(y \cdot z)$	x	x	x		x	S. 202
M1	$x \parallel y$	$=$	$(x \between y + y \between x) + x \mid y$		x	x		x	S. 209
LM2	$v \between y$	$=$	$v \cdot y$		x	x		x	S. 209
LM3	$(v \cdot x) \between y$	$=$	$v \cdot (x \parallel y)$		x	x		x	S. 209
LM4	$(x + y) \between z$	$=$	$x \between z + y \between z$		x	x		x	S. 209
CM5	$v \mid w$	$=$	$\gamma(v, w)$		x	x		x	S. 209
CM6	$v \mid (w \cdot y)$	$=$	$\gamma(v, w) \cdot y$		x	x		x	S. 209
CM7	$(v \cdot x) \mid w$	$=$	$\gamma(v, w) \cdot x$		x	x		x	S. 209
CM8	$(v \cdot x) \mid (w \cdot y)$	$=$	$\gamma(v, w)(x \parallel y)$		x	x		x	S. 209
CM9	$(x + y) \mid z$	$=$	$x \mid z + y \mid z$		x	x		x	S. 209
CM10	$x \mid (y + z)$	$=$	$x \mid y + x \mid z$		x	x		x	S. 209
A6	$x + \delta$	$=$	x			x		x	S. 211
A7	δx	$=$	δ			x		x	S. 211
D1	$\partial_H(v)$	$=$	$v, v \notin H$			x		x	S. 211
D2	$\partial_H(v)$	$=$	$\delta, v \in H$			x		x	S. 211
D3	$\partial_H(\delta)$	$=$	δ			x		x	S. 211
D4	$\partial_H(x + y)$	$=$	$\partial_H(x) + \partial_H(y)$			x		x	S. 211
D5	$\partial_H(x \cdot y)$	$=$	$\partial_H(x) \cdot \partial_H(y)$			x		x	S. 211
LM11	$\delta \between x$	$=$	δ			x		x	S. 211
CM12	$\delta \parallel x$	$=$	δ			x		x	S. 211
CM13	$x \mid \delta$	$=$	δ			x		x	S. 211
RDP	$\langle X_i \mid E \rangle$	$=$	$t_i(\langle X_1 \mid E \rangle, ..., \langle X_n \mid E \rangle)$				x	x	S. 217
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. 217
B1	$v \cdot \tau$	$=$	v					x	S. 224
B2	$v \cdot (x + y)$	$=$	$v \cdot (\tau \cdot (x + y) + x)$					x	S. 224
TI1	$\tau_I(v)$	$=$	$v, v \notin I$					x	S. 224
TI2	$\tau_I(v)$	$=$	$\tau, v \in I$					x	S. 224
TI3	$\tau_I(\delta)$	$=$	δ					x	S. 224
TI4	$\tau_I(x + y)$	$=$	$\tau_I(x) + \tau_I(y)$					x	S. 224
TI5	$\tau_I(x \cdot y)$	$=$	$\tau_I(x) \cdot \tau_I(y)$					x	S. 224

[2]Dank an Patrick Fey

10 Parallele Algorithmen

Unter parallelen und verteilten Algorithmen wird die Erweiterung von klassischen, für Einprozessormaschinen konzipierten Algorithmen auf viele miteinander kooperierende Prozessoren verstanden. Parallele Algorithmen beruhen in der Regel auf Speicher- oder Rendezvous-Synchronisation und haben eine synchrone Ablaufsemantik. Charakteristisch für die in späteren 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. 10.1). Bei Speichersynchronisation wird eine SIMD-Architektur (*single instruction multiple data*) (Abb. 10.1) benutzt, deren Formalisierung das PRAM-Modell (parallel random-access machine) ist.
Die Definition der PRAM erweitert das Konzept des Einprozessormodells der RAM (*random-access machine*). Für die RAM gelten Komplexitätsmaße wie für die Turingmaschine: Zeit- und Speicherkomplexität. Für das parallele Modell der PRAM kommen noch die Prozessor- und die Operationenkomplexität hinzu.

Definition 10.1 *Komplexitätsmaße für einen Algorithmus A sind die*

- Zeitkomplexität*: maximale Anzahl $T_A(n)$ der synchronen Schritte aller Prozessoren bis zur Termination, wobei das Maximum über alle Eingaben („Probleminstanzen“) der Größe n gebildet wird,*
- Speicherkomplexität*: maximale Anzahl $S_A(n)$ der im gemeinsamen Speicher und den lokalen Speichern der Prozessoren bis zur Termination belegten Speicherzellen, wobei das Maximum über alle Eingaben („Probleminstanzen“) der Größe n gebildet wird,*
- Prozessorkomplexität*: maximale Anzahl $P_A(n)$ der bis zur Termination aktiv gewordenen Prozessoren, wobei das Maximum über alle Eingaben („Probleminstanzen“) der Größe n gebildet wird,*
- Operationenkomplexität*: maximale Anzahl $W_A(n)$ der Operationen aller Prozessoren bis zur Termination, wobei das Maximum über alle Eingaben („Probleminstanzen“) der Größe n gebildet wird, (d.h. parallele Operationen werden einzeln gezählt, “W” für “work”).*

Für die RAM ist nach dieser Definition also $P_A(n) = 1$ und $W_A(n) = T_A(n)$.
Sei $\Theta(T^*_{\mathcal{P}}(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen (oft wird auch nur das Maximum unter allen *bekannten*

[1] Bachelor/Master Modul: Intelligente kooperierende Dienste (KD)

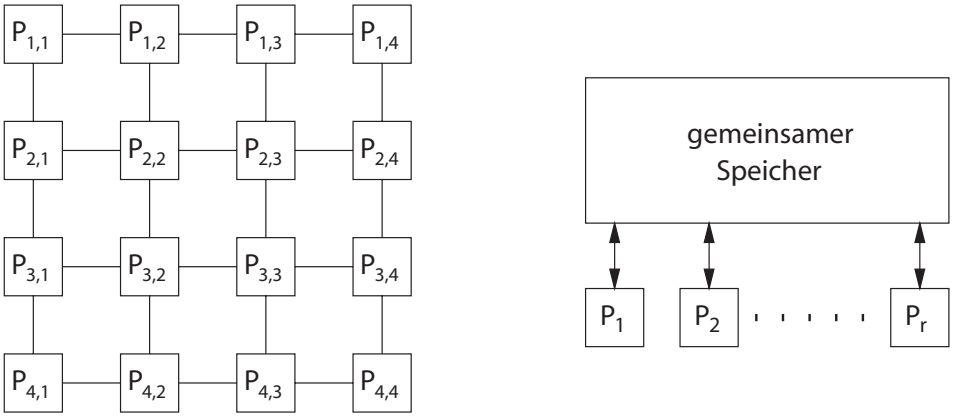


Abbildung 10.1: Prozessorkonfigurationen: 2-dimensionales Feld und PRAM

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

10.1 Random-Access-Maschine (RAM)[2]

Turing-Maschinen sind ausgezeichnete Modelle von universellen Rechnern, um die grundlegenden Probleme der Berechenbarkeit, Beschreibbarkeit und Komplexitätstheorie zu untersuchen.

In diesem Kapitel betrachten wir andere grundlegende Modelle von universellen sequentiellen und parallelen Rechnern, die entweder wichtig sind, um die Hauptprobleme des Entwurfs und der Analyse von Algorithmen zu untersuchen, oder wichtig sind zur Untersuchung von grundlegenden Problemen der Parallelität.

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

10.1.1 Definition der RAM

Das Turing–Maschinen Modell ist ausreichend, um Fragen der Berechenbarkeit zu untersuchen. Eine TM hat jedoch mit einem modernen Computer wenig gemeinsam.

Das wichtigste Beispiel eines Modells von realen sequentiellen Computern sind **Registermaschinen**. Diese Rechnermodelle besitzen einen Speicher ähnlich dem eines modernen Mikroprozessors. Der **Speicher** wie auch das **Eingabeband** bestehen jeweils aus einer Folge von Registern, die nicht nur einzelne Symbole, sondern auch ganze Zahlen beliebiger Größe speichern können. Ein **Programm** für eine Registermaschine ist mit einem in einer Assembler- oder höheren Programmiersprache geschriebenen Programm vergleichbar. Wir betrachten zwei Modelle von Registermaschinen: RAM und RASP.

Beim Modell der RAM unterscheiden wir zudem noch zwei Varianten: Die sogenannte *Bit-RAM*, bei der die Register beliebige natürliche (ganze) Zahlen enthalten dürfen. Bei der Modell–Variante der sogenannten *arithmetischen RAM* können die Speicherinhalte aus einem beliebigen Körper, wie z.B. $\mathbb{R}$ oder $\mathbb{Q}$ stammen. Weitere Modellvarianten entstehen bei Einschränkung des erlaubten Befehlssatzes.

Das Eingabeband einer RAM besteht aus einer abzählbaren Folge von Registern $x_1, x_2, \ldots$

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**.

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

Die aktuelle Konfiguration einer RAM läßt sich eindeutig aus dem Inhalt der Datenspeicher und dem Befehlszähler IC bestimmen. Am Anfang einer Berechnung sind alle Register mit 0 initialisiert.
Eine RAM wird durch ein Programm spezifiziert. Dies ist eine endliche, fortlaufend numerierte Folge $b_0, b_1, \ldots$ von Befehlen aus einer vorgegebenen Befehlsmenge $\mathcal{B}$.

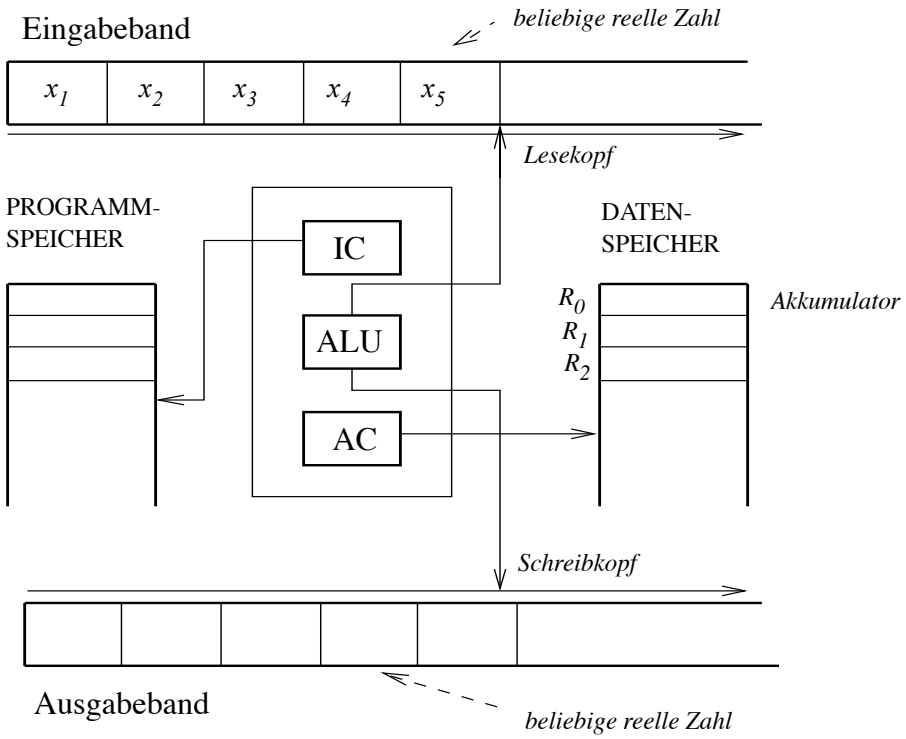


Abbildung 10.2: RAM – Random Access Machine

Operationen		Operand
READ	op	$= i$ – die Zahl i
WRITE	op	i – Inhalt des
LOAD	op	Registers R_i
STORE	op	$*i$ – indirekte Adresse
ADD	op	(Inhalt des Registers R_j, wobei j
SUB	op	der Inhalt des Registers R_i ist)
JUMP	label	
JZERO	label	label – Befehlsadresse
JGZERO	label	

Die Befehlsmenge $\mathcal{B}$ einer RAM umfaßt die vorstehend aufgelisteten Grundoperationen. *Die exakte Natur der Befehle ist nicht wichtig, solange die Befehle ähnlich den Befehlen*

von realen Computern sind. Es gibt hier **I/O–Operationen**, **arithmetische Operationen** und **JUMP–Operationen**. Außer JUMP– und I/O–Operationen werden alle mit Hilfe des Registers R_0 (Akkumulator) ausgeführt. Die folgende Tabelle 10.1 zeigt, wie sich die Konfiguration der Maschine durch Ausführung eines Befehls ändert.

Tabelle 10.1: Die Befehle einer RAM

Instruktion		Bedeutung
1.	LOAD a	$c(0) \leftarrow v(a)$
2.	STORE i	$c(i) \leftarrow c(0)$
	STORE $*i$	$c(c(i)) \leftarrow c(0)$
3.	ADD a	$c(0) \leftarrow c(0) + v(a)$
4.	SUB a	$c(0) \leftarrow c(0) - v(a)$
5.	MULT a	$c(0) \leftarrow c(0) \times v(a)$
6.	DIV a	$c(0) \leftarrow \lfloor c(0) \div v(a) \rfloor$
7.	READ i	$c(i) \leftarrow$ derzeitiges Eingabesymbol
	READ $*i$	$c(c(i)) \leftarrow$ derzeitiges Eingabesymbol. Der Kopf auf dem Eingabeband geht in beiden Fällen um ein Feld nach rechts.
8.	WRITE a	$v(a)$ wird in das Feld auf dem Ausgabeband geschrieben über dem sich gerade der Bandkopf befindet. Danach wird der Kopf um ein Feld nach rechts bewegt.
9.	JUMP b	Der *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

$$v(=i) = i$$
$$v(i) = c(i)$$
$$v(*i) = c(c(i))$$

Die Befehle des Programms werden in sequentieller Ordnung ausgeführt, bis eine Halte- oder JUMP-Instruktion vorkommt.

Im allgemeinen definiert ein RAM-Programm eine partielle Abbildung vom Eingabeband auf das Ausgabeband. Zwei Fälle sind wichtig:

1. *Berechnung von Funktionen:*
   Wenn ein RAM-Programm **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 **P** eine Funktion $f : \mathbb{N}^n \longrightarrow \mathbb{N}^m$ berechnet.

Es ist nicht schwer zu zeigen, dass RAM-Programme für Bit-RAM's jede partiell rekursiven Funktion, und nur diese, berechnen können.

Beispiel 10.2 In Abb. 10.3 gibt es ein RAM-Programm, um die Funktion $f(n) = n^n$ zu berechnen. Eine Beschreibung desselben Programms in einer höheren Programmiersprache ist in Abb. 10.4 zu sehen. Aus Abb. 10.3 kann man auch ersehen, wie man die Befehle der höheren Programmiersprache in RAM-Instruktionen übersetzen kann.

```
          READ    1          read r1
          LOAD    1
          JGTZ    pos        if r1 ≤ 0 then write 0
          WRITE   =0
          JUMP    endif
pos:      LOAD    1
          STORE   2          r2 ← r1
          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 10.3: $f(n) = n^n$ (RAM-Programm)

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

   Sei $\Sigma = \{a_1, \ldots, a_m\}$ 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 $\mathbf{P}$ 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.

```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

Abbildung 10.4: $f(n) = n^n$ (Hochsprache)

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

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

Beispiel 10.3 In Abbildung 10.6 und 10.5 ist ein Hochsprache–Programm und ein RAM-Programm abgebildet, um die Sprache $\mathcal{L}_0 \subsetneq \{1,2\}^*$ zu akzeptieren, deren Wörter dieselbe Anzahl von Einsen und Zweien besitzt.

10.1.2 Komplexitätsmaße für die RAM

Die parallelen Algorithmen am Ende dieses Kapitels werden in einem Pseudokode formuliert, für den die in Definition 10.1 angegebenen Definitionen ausreichend sind. Das Modell der RAM ist jedoch genauer und hat präziser formulierte Definitionen für Komplexitätsmaße. Beispielsweise wird der in Definition 10.1 benutzte Begriff „Größe“ genauer gefasst.

Es gibt zwei Arten von Komplexitätsmaßen für RAMs.

Uniforme Komplexitätsmaße:

Uniformes Zeitmaß:	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

```
            LOAD    =0          } d ← 0
            STORE   2           }
            READ    1             read x
while:      LOAD    1           } while x ≠ 0 do
            JZERO   endwhile    }
            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 10.5: Erkennung von $\mathcal{L}_0 \subsetneq \{1,2\}^*$ (RAM)

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 10.2 (Seite 243) aufgelistet und hängen von der Länge der Operanden ab:

Operand a	Kosten t(a)
=i	l(i)
i	l(i) + l(c(i))
∗i	l(i) + l(c(i)) + l(c(c(i)))

Definition 10.4 *Die* uniforme (logarithmische) Zeitkomplexität *eines RAM-Programms A ist die maximale Anzahl $T_A(n)$ der Summen der uniformen (logarithmischen) Kosten aller Schritte des Programms bis zur Termination, wobei das Maximum über alle Eingaben der uniformen (logarithmischen) Größe n gebildet wird.*
Die uniforme (logarithmische) Platzkomplexität *eines RAM-Programms A ist die maximale Summe $S_A(n)$ der uniformen (logarithmischen) Kosten aller Register, die dieses Programm bis zur Termination addressiert, wobei das Maximum über alle Eingaben der uniformen (logarithmischen) Größe n gebildet wird.*

```
BEGIN
   d ← 0;
   READ x;
   WHILE x ≠ 0 DO
     BEGIN
       IF x ≠ 1 THEN d ← d - 1 ELSE d ← d + 1;
       READ x
     END;
   IF d = 0 THEN WRITE 1
END
```

Abbildung 10.6: Erkennung von $\mathcal{L}_0 \subsetneq \{1,2\}^*$ (Hochsprache)

Tabelle 10.2: Kosten einer RAM-Instruktion

1.	LOAD a	t(a)
2.	STORE i	l(c(0)) + l(i)
	STORE $*i$	l(c(0)) + l(i) + l(c(i))
3.	ADD a	l(c(0)) + t(a)
4.	SUB a	l(c(0)) + t(a)
5.	MULT a	l(c(0)) + t(a)
6.	DIV a	l(c(0)) + t(a)
7.	READ i	l(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

Das logarithmische Platzmaß *eines Registers (während eines Programmablaufs) ist die maximale Länge $l(i)$ aller Zahlen i, die in diesem Register gespeichert wurden (während des Programmablaufs).*

Beispiel 10.5 Betrachten wir das RAM-Programm A, das $f(n) = n^n$ berechnet (siehe Abb. 10.4 und 10.3). Es ist offensichtlich, dass für die uniformen Komplexitätsmaße die Zeitkomplexität $T_A(n) = \mathrm{O}(n)$ 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 10.6 Die folgende Tabelle 10.3 zeigt die Komplexitätsmaße für das Programm, das die Sprache $\mathcal{L}_0$ akzeptiert.

Tabelle 10.3: Programm–Komplexitätsmaße für $\mathcal{L}_0$

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n \log n)$
Platzkomplexität	$O(1)$	$O(\log n)$

In vielen Fällen ist es notwendig, die logarithmischen Komplexitätsmaße zu benutzen, um realistische Ergebnisse zu erhalten.

Z.B. wissen wir schon, dass jede Turing–Maschine durch einen Counterautomaten mit 2 Countern simuliert werden kann (in denen sehr große Zahlen gespeichert sind). d. h. jede partiell rekursive Funktion kann durch ein RAM-Programm berechnet werden, dessen Platzkomplexität O(1) für das uniforme Platzkomplexitätsmaß ist! (In diesem Fall ist die uniforme Platzkomplexität offensichtlich nicht realistisch. Die logarithmische Platzkomplexität liefert viel realistischere Ergebnisse.)

Manchmal bezeichnet man die RAM mit unserer Menge von Befehlen als RAM_* und die RAM *ohne die Operation MULT und DIV* als RAM_+. Sind die in den Registern speicherbaren Zahlen stets natürliche Zahlen, so spricht man von einer *Bit-RAM*. Können rationale oder reelle Zahlen benutzt werden, so spricht man von einer *arithmetischen RAM*.

Die `MULT`–Operation ist sehr mächtig. Z.B. kann man mit n `MULT`–Operationen die Funktion $f(n) = n^{2^n}$ berechnen. Die Zeitkomplexität solcher Folge von `MULT`–Operationen ist $O(n)$ bezüglich des uniformen Komplexitätsmaßes und $O(2^n)$ – exponentiell mehr – bzgl. des logarithmischen Zeitkomplexitätsmaßes.

Es wird später für die RAM_+ gezeigt, dass auch das uniforme Zeitkomplexitätsmaß Ergebnisse produziert, die nicht sehr schlecht sind.

RAMs sind geeignete Modelle, um die Komplexität von algebraischen und kombinatorischen Optimierungs–Problemen zu untersuchen.

10.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine

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

Satz 10.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*

Beweis: M habe k einseitig unendliche Bänder. Die simulierende RAM_+ (in der Abbildung 10.7 mit R bezeichnet) verhält sich auf dem Ein- und Ausgabeband genau wie M. Die Darstellung der Beschriftung der Arbeitsbänder der TM geschieht folgendermaßen: R_0 speichert den Zustand von M, $R_j, 1 \leq j \leq k$, die Position p_j des Kopfes auf Band j und R_{kp+j} das Symbol a_j der Zelle p_j des j–ten Bandes, kodiert als eine natürliche Zahl.

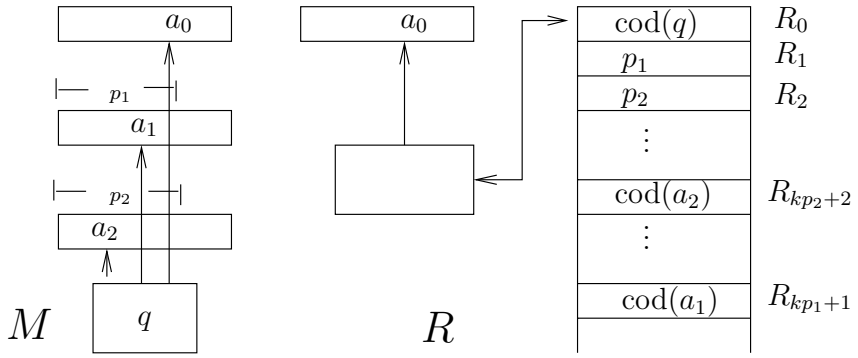


Abbildung 10.7: Zur Simulation einer TM durch eine RAM

Für jedes Tupel

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

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. □

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

Satz 10.8 *Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.*

Beweis: M speichert den Inhalt der Register von R auf dem ersten Band als

#	#	i_1	#	c_1	#	#	i_2	#	c_2	#	#	$\cdots$	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, außer den Inhalt des Register 32, auf das 5te Band. dann kopiert M die Zahl auf dem zweiten Band auf das erste, gleich rechts von ##100000#. Danach kopiert M den Inhalt des fünften Bandes auf das erste Band, ganz rechts.
3. Falls es kein Register 32 auf dem ersten Band gibt, geht M zum letzten und schreibt dann ##100000# und dahinter den Inhalt von Band 2.

Zur Zeitkomplexität: Die Länge des Wortes auf dem ersten Band ist höchstens $T(n)$ – die logarithmische Zeitkomplexität von R. Die Zeit, die M braucht, um einen Befehl von R zu simulieren ist auch $O(T(n))$ und deshalb ist die gesamte Zeit der Simulation $O(T^2(n))$. □

Man kann auch ein allgemeines Resultat zeigen: Jede im logarithmischen Maß T(n)–zeitbeschränkte RAM_* (d. h. auch mit Multiplikation und Division) kann durch eine $\mathrm{O}(T^2(n))$ 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:

$$\begin{aligned}\mathrm{RAM}_+\text{–Time}(POL) &= \text{RAM-Time}_{log}(POL) = \mathcal{P} \\ &= \text{DTM–Time}(POL) = \mathrm{DTM}_1\text{–Time}(POL)\end{aligned}$$

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.

10.2 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.

10.2.1 Definition der PRAM

Definition 10.9 (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.*
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.*

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

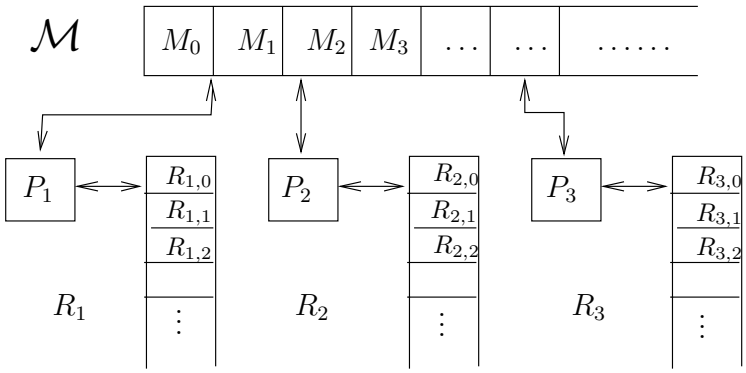


Abbildung 10.8: Parallele RAM - PRAM

Beim Erkennen von Sprachen kann die Entscheidung, ob eine Eingabe akzeptiert oder verworfen wird, auch durch die Art der Halte-Instruktion von P_1 geschehen.
Die PRAM wird spezifiziert durch ein Programm für jeden Prozessor – eine Folge von Instruktionen, die er auf eine Eingabe ausführt. Dies geschieht schrittweise synchron mit den anderen Prozessoren. Dabei gelte:

Die Programme sind für alle Prozessoren identisch!

Eine Berechnung *startet, indem jeder Prozessor synchron mit den Anderen die erste Instruktion seines Programmes ausführt. Sie endet, wenn P_1 eine* Halte-Instruktion *erreicht. Die Beschriftung des globalen Speichers zu diesem Zeitpunkt spezifiziert die Ausgabe.*

Bemerkung: Selbst wenn alle Prozessoren das gleiche Programm ausführen, bedeutet dies nicht, dass sie zu jedem Zeitpunkt vollkommen identische Instruktionen ausführen (und damit die Parallelität keinen Vorteil brächte). Da Prozessoren ihre PID zur Verfügung steht, kann ihr Verhalten von dieser abhängen, und sie können somit mit unterschiedlichen Daten rechnen! P_i kann beispielsweise seine PID als Adresse benutzen, um das i-te Eingabesymbol zu lesen, falls $i \leq n$.
Außerdem kann sich der Programmablauf einzelner Prozessoren aufgrund von unterschiedlichen Sprüngen bei *JUMP*-Instruktionen unterscheiden. Dies kann dazu führen, dass die Prozessoren in einem Schritt unterschiedliche Arten von Instruktionen ausführen.

10.2.2 Konflikte beim Speicherzugriff

Die einzige Schwierigkeit, die bei PRAMs auftritt, sind Konflikte beim gleichzeitigen Zugriff mehrerer Prozessoren auf ein Register des globalen Speichers.

Das Lesen eines Registers M_j, auf dem simultan auch eine Schreiboperation ausgeführt wird, ist unkritisch – wir vereinbaren nämlich, dass in jedem synchronen Schritt alle *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.

10.2.3 Komplexitätsmaße der PRAM

Definition 10.10 *Die uniforme (logarithmische)* Zeitkomplexität $time_M(x)$ *einer PRAM M auf eine Eingabe x ist entsprechend wie für die RAM in Definition 10.4 definiert, wobei die synchronen Schritte bis zur Termination gerechnet werden. Die Zeitkomplexität $T_M(n)$ von M ist wieder das Maximum aller $time_M(x)$, wobei das Maximum über alle Eingaben x der uniformen (logarithmischen) Größe n gerechnet wird.*

Anders als bei den Pseudokode-Programmen am Ende dieses Kapitels ist die Anzahl der benutzten Prozessoren unendlich, wobei meist nur endlich viele davon effektiv genutzt werden. Das sind diejenigen, die einen Schreibbefehl ausführen und dadurch das Ergebnis beeinflussen können. Die anderen Prozessoren lesen nur „still“ mit und werden bei einer Implementation nicht berücksichtigt. Deshalb definieren wir:

Definition 10.11 *Die* Prozessorkomplexität $proc_M(x)$ *einer PRAM M auf eine Eingabe x sei erklärt durch* $proc_M(x) = max\{i \mid i = 1 \text{ oder } P_i \text{ fuehrt auf } x \text{ ein } WRITE \text{ aus}\}$.