

1. Einleitung

Dieses Buch soll selbsterklärend sein. Damit meinen wir, dass möglichst alle vorkommenden Begriffe, auch sehr elementare, wie etwa der eines endlichen Automaten, einer Turingmaschine oder einer Registermaschine, formal exakt definiert werden. Natürlich kann das nicht auch für alle mathematischen Begriffe gelten. So werden wir Standardbegriffe, wie etwa den Logarithmus, etc., nicht weiter erläutern. In der Einleitung werden allerdings auch Konzepte der Theoretischen Informatik erst einmal ohne formale Definitionen, die später folgen, benutzt. Hier wollen wir den Versuch unternehmen, Petri-Netz-Theorie in die Theoretische Informatik einzuordnen. Dies ist ein eher „philosophisches“ Kapitel, das einen kleinen Überblick über klassische Theorien von sequentiellen, parallelen und nebenläufigen Rechnungen gibt. Es soll aufzeigen, wie sich Petri-Netz-Theorie einbettet, was sie so attraktiv gemacht hat. Das eigentliche Buch beginnt ab Kapitel 3 und ist lesbar, ohne sich um diese Einordnung in Kapitel 1 kümmern zu müssen, wenn man die Standardkonzepte der Mathematik aus Kapitel 2 nur bei Bedarf nachschlägt.

1.1 Sequentielle Rechnungen

Der klassische Untersuchungsgegenstand der Theoretischen Informatik sind sequentielle Rechnungen. Diese sind dadurch charakterisiert, dass alle Rechenschritte hintereinander ausgeführt werden. Turingmaschinen, Markov-Algorithmen, Grammatiken, oder λ -Kalküle sind Modelle sequentieller Rechnungen. Der Algorithmus- und Kalkülbegriff selbst ist in seiner klassischen Form sequentiell. Ein Algorithmus ist abstrakt gesehen eine endliche Vorschrift, bestehend aus Anweisungen, die in Abhängigkeit vom jeweiligen Zwischenergebnis einer Rechnung eindeutig den nächsten Rechenschritt festlegen. Im Gegensatz dazu stellt ein Kalkül eine endliche Auswahl von Anweisungen zur Verfügung, von denen je nach Zwischenergebnis mehrere verwendet werden dürfen. Dennoch darf auch von mehreren anwendbaren Schritten jeweils nur einer angewendet werden. Algorithmen führen damit zu deterministischen sequentiellen Rechnungen, Kalküle zu nichtdeterministischen sequentiellen Rechnungen.

Eine formale Präzisierung des Algorithmusbegriffs sind z.B. *Turingmaschinen*. Das Speichermedium einer Turingmaschine ist eine endliche Anzahl von sogenannten *Bändern*. Ein Band ist eine lineare Anordnung von potentiell unendlich vielen sogenannten *Zellen* (oder *Feldern*). Jede Zelle ist mit einem Symbol aus einer gegebenen endlichen Menge Σ von Zeichen, dem *Bandalphabet* Σ , besetzt. Um den Begriff des potentiell Unendlichen elegant in den Griff zu bekommen, kann man ein Band als ein unendliches Wort über Σ auffassen, dessen Buchstaben fast alle aus dem Symbol für „leer“, $\#$, bestehen. „Fast alle“ wird hier stets im Sinn von „alle bis auf endlich viele Ausnahmen“ benutzt. Ein Band besitzt also zwar immer einen unendlichen Vorrat an Zellen, zu jedem Zeitpunkt einer Rechnung werden aber nur endlich viele davon benutzt (allerdings ohne Beschränkung in deren Anzahl), fast alle sind leer, d.h. tragen das Sondersymbol $\#$. Zu jedem Band besitzt eine Turingmaschine einen sogenannten *Schreib-Lese-Kopf*, der die Verbindung des Programmteils der Maschine zu ihrem Speichermedium darstellt. Der Schreib-Lese-Kopf ist jeweils auf eine Zelle des Bandes ausgerichtet. Er kann das Symbol in dieser Zelle lesen, ein Symbol in die Zelle schreiben und sich auf die Nachbarzelle recht oder links begeben. Das *Programm* einer Turingmaschine ist eine endliche Folge $1 : B_1; 2 : B_2; \dots; k : B_k$ von durchnummerierten Elementarbefehlen B_j . Als einen Satz von erlaubten Elementarbefehlen kann man etwa wählen:

- Lies x auf Band i (x enthält als Wert das Symbol, das sich gerade auf der Zelle des Schreib-Lese-Kopfes von Band i befindet),
- Schreibe a auf Band i (das Symbol auf der Zelle des Schreib-Lese-Kopfes von Band i wird durch a ersetzt, dabei muss $a \in \Sigma$ gelten),
- Gehe nach rechts auf Band i (der Schreib-Lese-Kopf von Band i wird auf die rechte Nachbarzelle gesetzt),
- Gehe nach links auf Band i (analog),
- Falls $x = a$ gehe zur Zeile j (falls das zuletzt gelesene Symbol ein a ist, $a \in \Sigma$ muss gelten, so führe die Programmausführung mit dem Elementarbefehl Nummer j fort),
- HALT (bei Erreichen dieses Befehls terminiert die Programmausführung).

Ohne Einschränkung soll genau der letzte Befehl $k : B_k$ die Form $k : \text{HALT}$ besitzen und sonst kein HALT-Befehl vorkommen. Ferner soll nur zu vorhandenen Befehlen j , $1 \leq j \leq k$, gesprungen werden dürfen. Benutzt eine Turingmaschine m Bänder, so spricht man auch von einer *m-Band-Turingmaschine*. Turingmaschinen mit solch einem Befehlssatz heißen auch *deterministisch*. Lassen wir noch Elementarbefehle der Art

- Gehe zur Zeile j_1 oder j_2

hinzu, so erhalten wir auch *nichtdeterministische Turingmaschinen*. Eine korrekte Ausführung dieses Befehls besteht darin, mit Programmzeile j_1 oder mit Programmzeile j_2 fortzufahren. Zu Beginn stehen alle Schreib-Lese-Köpfe aller Bänder auf der jeweils ersten Zelle, und alle Bänder bis auf das erste sind leer. Der Inhalt des ersten Bandes soll die Form $\#w\#\dots\#\dots$ besitzen, wobei w ein Wort über dem Alphabet $\Sigma - \{\#\}$ ist.

Eine *Konfiguration* einer m -Band-Turingmaschine ist ein aktueller Schnappschuss einer Rechnung, aus dem sich die Rechnung eindeutig fortsetzen lässt. D.h., es muss angegeben werden, welche Programmzeile j , $1 \leq j \leq k$, ausgeführt werden soll, was die aktuellen Inhalte der m Bänder sind, und auf welcher Zelle jeden Bandes der Schreib-Lese-Kopf dieses Bandes steht. Dabei lässt sich der aktuelle Inhalt und die aktuelle Position des Schreib-Lese-Kopfes eindeutig durch ein endliches Wort der Form uav beschreiben. uav ist der Inhalt des Bandes, wobei der unendliche Anteil der Symbole $\#$ am rechten Ende von v weggelassen wird, und a das Symbol, auf dessen Feld sich der Schreib-Lese-Kopf befindet. Eine Konfiguration C kann also durch einen Vektor $C = (j, u_1a_1v_1, \dots, u_ma_mv_m)$ beschrieben werden. Hier ist j , $1 \leq j \leq k$, die auszuführende Programmzeile und $u_ia_iv_i$ die aktuellen Bandinhalte, $1 \leq i \leq m$. Eine *Startkonfiguration* hat also die Form

$$C_1 = (1, \underline{\#w}, \underline{\#}, \dots, \underline{\#})$$

mit dem Inputwort w auf Band 1 und leeren Bändern sonst. Eine Konfiguration $C' = (j', u'_1\underline{a'_1}v'_1, \dots, u'_m\underline{a'_m}v'_m)$ heißt *direkter Nachfolger* einer Konfiguration $C = (j, u_1\underline{a_1}v_1, \dots, u_m\underline{a_m}v_m)$, $C \vdash C'$, falls die Konfiguration $C = (j, u_1\underline{a_1}v_1, \dots, u_m\underline{a_m}v_m)$ durch korrekte Anwendung des Befehls $j : B_j$ in die Konfiguration $C' = (j', u'_1\underline{a'_1}v'_1, \dots, u'_m\underline{a'_m}v'_m)$ übergeht. Bei deterministischen Turingmaschinen kann jede Konfiguration höchstens einen direkten Nachfolger besitzen, bei nichtdeterministischen Turingmaschinen können es mehrere sein. Eine *Rechnung* $R = C_1, C_2, C_3, \dots$ ist eine endliche oder unendliche Folge $(C_i)_{i \in I}$, $I = \{1, \dots, n\}$ oder $I = \mathbb{N}$, mit $C_i \vdash C_{i+1}$ für alle i , $i+1 \in I$. Endet eine Rechnung $R = C_1, \dots, C_n$ von $C_1 = (1, \underline{\#w}, \underline{\#}, \dots, \underline{\#})$ aus in einer *Haltekonfiguration* $C_n = (k, u_1a_1v_1, \dots, u_ma_mv_m)$, so interpretieren wir $z = u_1a_1v_1$ als *das Ergebnis* der Rechnung mit Input w , $u_ia_iv_i$ für $i > 1$ hingegen als uninteressante Nebeneffekte. Eine unendlich lange Rechnung heißt auch *nicht abbrechend*, eine endliche Rechnung, die mit einer Haltekonfiguration endet, auch *terminierend*. Im Falle deterministischer Turingmaschinen existiert zu jeder Startkonfiguration genau eine Rechnung. Im Falle nichtdeterministischer Turingmaschinen können eventuell mehrere verschiedene Rechnungen mit der gleichen Startkonfiguration beginnen. Dabei ist es auch möglich, dass manche dieser Rechnungen von der gleichen Startkonfiguration aus nicht abbrechen, andere hingegen mit eventuell unterschiedlichen Ergebnissen terminieren.

Mittels Turingmaschinen lassen sich nun die zentralen Begriffe der Berechenbarkeitstheorie, wie *entscheidbar*, *akzeptierbar* und *berechenbar* elegant definieren:

Eine Turingmaschine M *akzeptiert* ein Wort $w \in \Sigma^*$, falls eine terminierende Rechnung mit Input w existiert. Eine Turingmaschine M *akzeptiert eine Sprache* $L \subseteq \Sigma^*$, falls M genau die Wörter $w \in L$ akzeptiert und die Wörter $w \in \Sigma^* - L$ nicht akzeptiert.

Eine deterministische Turingmaschine M *entscheidet* eine Sprache $L \subseteq \Sigma^*$, falls M für jeden Input $w \in \Sigma^*$ terminiert und genau bei jedem Input $w \in L$ mit dem Ergebnis #1 terminiert und bei jedem Input $w \in \Sigma^* - L$ mit dem Ergebnis #0.

Eine deterministische Turingmaschine M *berechnet* eine partielle Funktion $f: \Sigma^* \rightarrow \Sigma^*$ genau dann, wenn für jedes Wort $w \in \Sigma^*$ gilt:

$$\begin{aligned} f(w) \text{ ist undefiniert} &\iff \begin{array}{l} \text{die Rechnung von } M \text{ mit Input } w \\ \text{terminiert nicht, und} \end{array} \\ f(w) \text{ ist definiert} &\iff \begin{array}{l} \text{die Rechnung von } M \text{ mit Input } w \\ \text{terminiert mit Ergebnis } \#f(w). \end{array} \end{aligned}$$

Hierbei dürfen zur Akzeptanz oder Entscheidung einer Sprache L über dem Alphabet Σ oder zur Berechnung einer Funktion $f: \Sigma \rightarrow \Sigma$ auf den Zellen der Turingmaschine auch Symbole gespeichert werden, die im Alphabet Σ gar nicht vorkommen. Ferner kann man jede zahlentheoretische Funktion $f: \mathbb{N}^r \rightarrow \mathbb{N}$ mit r Argumenten auch als eine Funktion $f: \{\#, |\}^* \rightarrow \{\#, |\}^*$ auffassen, indem man den Argumentevektor $(x_1, \dots, x_r) \in \mathbb{N}^r$ als Wort $\#^{x_1} \#^{x_2} \dots \#^{x_r}$ und das Ergebnis y als Wort $\#^y$ verschlüsselt. Eine zahlentheoretische Funktion $f: \mathbb{N}^r \rightarrow \mathbb{N}$ ist damit berechenbar, falls eine Turingmaschine M_f existiert, die mit dieser Verschlüsselung f als Funktion von $\{\#, |\}^*$ nach $\{\#, |\}^*$ berechnet. Man kann etwa zeigen, dass jede berechenbare zahlentheoretische Funktion bereits von einer Turingmaschine berechnet werden kann, die nur ein Band benutzt und außer # und | keine weiteren Zeichen als Bandalphabetsymbole besitzt. Solche Turingmaschinen sollen *normiert* genannt werden.

Ein bekanntes Beispiel für eine nicht berechenbare zahlentheoretische Funktion ist die *busy-beaver* Funktion $b: \mathbb{N} \rightarrow \mathbb{N}$, definiert als $b(n) = m$, falls eine normierte Turingmaschine mit einem Programm aus maximal n Befehlen existiert, die bei Input 0 mit dem Ergebnis m terminiert, und falls keine normierte Turingmaschine mit einem Programm aus maximal n Befehlen existiert, die bei Input 0 mit einem Ergebnis größer als m terminiert. D.h. also

gerade, dass eine Rechnung von $1, \#$ aus maximal $t, \#|^m$ erreichen kann, falls die Programmlänge auf n beschränkt wird.

Eine Turingmaschine wird eindeutig durch ihr Programm festgelegt. Dieses Programm ist ein endliches Wort über einem geeigneten Alphabet, z.B. dem Alphabet mit allen Ziffern, Buchstaben und einigen Sonderzeichen wie „:“, „“, „“, mit einer dadurch eindeutig festlegbaren Zahl als Kodierung, z.B. der ASCII-Zahl dieses Programms. Eine solche elementare, eindeutige Zuordnung einer Zahl zu einer Turingmaschine, aus der man elementar die Turingmaschine wieder eindeutig rekonstruieren kann, nennt man eine *Gödelisierung*; die hierbei einer Turingmaschine M zugeordnete Zahl n auch die *Gödelnummer* $g(M)$ von M .

Jede Zahl n kann kanonisch als das Wort $|^n$ über dem einelementigen Alphabet $\{|\}$ aufgefasst werden. Damit ist jede Menge von Zahlen eine Sprache über $\{|\}$. Ein Beispiel einer unentscheidbaren Sprache ist das sogenannte *Halteproblem*, definiert als

$$H = \{n \in \mathbb{N} \mid n \text{ ist Gödelnummer einer Turingmaschine, die bei Input } n \text{ terminiert}\}.$$

Man kann auf vielfältige andere Weisen versuchen, den Begriff einer berechenbaren zahlentheoretischen Funktion oder den einer entscheidbaren Sprache zu definieren. Dies wurde zu Beginn des 20. Jahrhunderts sehr sorgfältig unternommen, mit den bereits genannten Konzepten der Markov-Algorithmen und des λ -Kalküls, oder auch mittels μ -rekursiver Funktionen oder Funktionsgleichungssystemen. In jedem Fall erhielt man die gleiche Klasse von berechenbaren Funktionen oder entscheidbaren Sprachen. Mit der *Churchschen These* werden diese Bemühungen zusammengefasst. Sie besagt, dass die intuitive Vorstellung des Berechenbaren oder Entscheidbaren exakt durch diese formalen Definitionen mittels Turingmaschinen wiedergegeben wird. Natürlich ist diese These kein Satz im mathematischen Sinn, da der Begriff der „intuitiven Vorstellung“ philosophischer Natur und nicht mathematischer Natur ist.

Neben diesen grundsätzlichen Fragen zur Präzisierung des Berechenbaren und Entscheidbaren spielt die Turingmaschine die entscheidende Rolle in der Präzisierung des Begriffs der *Komplexität* (von Algorithmen, Funktionen oder Sprachen). So *akzeptiert* eine m -Band-Turingmaschine M eine Sprache $L \subseteq \Sigma^*$ (mit $\# \notin \Sigma$) mit einer *Platzbedarfsfunktion* (bzw. einer *Schrittzahlfunktion*) $f: \mathbb{N} \rightarrow \mathbb{N}$, falls M L akzeptiert und für jedes $w \in L$ eine Rechnung von M mit Input w existiert, die bei Benutzung von maximal $f(n)$ Bandfeldern (bzw. nach maximal $f(n)$ Rechenschritten) terminiert. Hierbei ist n die Länge von w , d.h. die Anzahl der in w vorkommenden Buchstaben.

P ist die Klasse aller Sprachen, die von einer deterministischen Turingmaschine mit einem Polynom als Schrittzahlfunktion akzeptiert werden, und NP ist die Klasse aller Sprachen, die von einer nichtdeterministischen Turingmaschine mit einem Polynom als Schrittzahlfunktion akzeptiert werden. Allgemein wird $P \neq NP$ angenommen, aber es existiert kein Beweis für diese Vermutung. $EXPTIME$ bzw. $EXPSPACE$ sind die Klassen aller Sprachen, die von einer deterministischen Turingmaschine mit einer exponentiellen ($f(x) = b \cdot a^x$ für $a, b \in \mathbb{N}$) Schrittzahl- bzw. Platzbedarf Funktion akzeptiert werden können. Da die Arbeit einer jeden nichtdeterministischen Turingmaschine bei allerdings exponentiell erhöhter Rechenzeit auch von einer deterministischen Turingmaschine ausgeführt werden kann, gilt $NP \subseteq EXPTIME$. Damit gilt insgesamt $P \subseteq NP \subseteq EXPTIME \subseteq EXPSPACE$.

Man kann aber sehr leicht zeigen, dass die Rechnung einer jeden m -Band-Turingmaschine auch von einer 1-Band-Turingmaschine ausgeführt werden kann. Akzeptiert die m -Band-Turingmaschine mit einer Schrittzahlfunktion f , so genügt für die dazu äquivalente 1-Band-Turingmaschine eine quadratische Schrittzahlfunktion g mit $g(n) = a \cdot f^2(n)$ für ein $a \in \mathbb{N}$. Wird eine Sprache L über \mathbb{N} (d.h. über dem Alphabet $\{\}\}$) von einer 1-Band-Turingmaschine M mit einer Schrittzahlfunktion f akzeptiert, so existiert auch eine normierte Turingmaschine M' (d.h. mit nur $|$ und $\#$ als erlaubte Bandsymbole), die ebenfalls L akzeptiert mit einer Schrittzahlfunktion g mit $g(n) = a \cdot f(n)$ für ein $a \in \mathbb{N}$. Da für jedes Polynom bzw. jede Exponentialfunktion f auch $a \cdot f$ und $a \cdot f^2$ durch ein Polynom bzw. eine Exponentialfunktion majorisierbar sind, ist es für die Klassen P , NP , $EXPTIME$ und $EXPSPACE$ egal, ob man beliebige m -Band-Turingmaschinen oder nur normierte 1-Band-Turingmaschinen benutzt. In Kapitel 2 definieren wir der Einfachheit halber Turingmaschinen nur mit einem Band, wobei wir dann ein zweiseitig unbeschränktes Band zulassen (nur weil die formalen Definitionen dabei etwas übersichtlicher werden). Eine Konfiguration $j, u \underline{a} v$ werden wir dann auch einfach als ein Wort $ujav$ auffassen. Außerdem wollen wir eine formale Definition wählen, die auf ein Programm einer Turingmaschine verzichtet und stattdessen (dazu äquivalent) mit Maschinenzuständen arbeitet.

Eine Sprache heißt *effektiv*, falls sie in P liegt, und *ineffektiv entscheidbar*, falls sie zwar entscheidbar ist, aber nicht in P liegt.

Neben der Turingmaschine spielt das Modell der *Registermaschine* in der Theoretischen Informatik eine wichtige Rolle. Es ist dem Konzept der konkreten Rechner ähnlicher als die Turingmaschine. Eine Registermaschine besitzt als Speichermedium eine endliche Anzahl von Registern, R_1, \dots, R_m . Jedes Register R_i , $1 \leq i \leq m$, kann dabei eine beliebig große natürliche Zahl $a \geq 0$ speichern. Ein Programm ist wieder eine endliche Folge $1 : B_1; 2 : B_2; \dots; k : B_k$ von durchnummerierten Elementarbefehlen. Elementarbefehle haben dabei die Form

addiere 1 in Register i ,

subtrahiere 1 in Register i (dabei soll der Registerinhalt 0 bei der Subtraktion unverändert bleiben),

falls Register i die Zahl 0 enthält, dann gehe zur Zeile j_1 , sonst gehe zur Zeile j_2 ,

HALT.

Dabei ist genau der letzte Befehl ein HALT-Befehl und es darf nur zu existierenden Zeilen gesprungen werden.

Als *Konfiguration* für eine Registermaschine bietet sich etwa ein Tupel (j, x_1, \dots, x_m) an. Dabei ist j der gerade auszuführende Befehl und x_i der aktuelle Inhalt im Register i . Es sei $C = (j, x_1, \dots, x_m)$ eine Konfiguration. Hat der j -te Befehl $j : B_j$ die Form j : addiere 1 in Register i , so ist $C' = (j+1, x_1, \dots, x_{i-1}, x_i+1, x_{i+1}, \dots, x_m)$ die direkte Nachfolgekonfiguration von C . Lautet der j -te Befehl j : subtrahiere 1 in Register i , so ist $C' = (j+1, x_1, \dots, x_{i-1}, x_i-1, x_{i+1}, \dots, x_m)$ mit $0-1 := 0$ und $(n+1)-1 := n$ die direkte Nachfolgekonfiguration. Hat der j -te Befehl die Form j : falls Register i die Zahl 0 enthält, dann gehe zur Zeile j_1 , sonst gehe zur Zeile j_2 , so ist im Falle $x_i = 0$ die direkte Nachfolgekonfiguration $C' = (j_1, x_1, \dots, x_m)$ und im Fall $x_i > 0$ gerade $C' = (j_2, x_1, \dots, x_m)$. Wir setzen $C \vdash C'$, falls C' direkte Nachfolgekonfiguration von C ist. Damit können wir wieder den Begriff einer Rechnung einführen.

Wir sagen, dass eine Registermaschine M mit m Registern eine partielle zahlentheoretische Funktion $f: \mathbb{N}^r \rightarrow \mathbb{N}$ mit $r < m$ berechnet, falls gilt

- $f(x_1, \dots, x_r)$ ist undefiniert \iff die Rechnung von M mit der Startkonfiguration $(1, x_1, \dots, x_r, 0, \dots, 0)$ (d.h. x_i in Register i für $1 \leq i \leq r$, und 0 in allen anderen Registern) terminiert nicht, und
- $f(x_1, \dots, x_r) = y$ \iff die Rechnung von M mit der Startkonfiguration $(1, x_1, \dots, x_r, 0, \dots, 0)$ terminiert mit der Haltekonfiguration $(t, y, 0, \dots, 0)$ (d.h. y im Register 1 und 0 sonst).

Es stellt sich heraus, dass die Klasse der von Turingmaschinen berechenbaren partiellen zahlentheoretischen Funktionen identisch ist mit der Klasse der von Registermaschinen berechenbaren partiellen zahlentheoretischen Funktionen.

Hierbei ist es völlig unerheblich, dass das Ergebnis einer Rechnung am Schluss in Register 1 steht. In Definition 5.4.6 werden wir später nur aus technischen Gründen verlangen, dass das Ergebnis von $f: \mathbb{N}^r \rightarrow \mathbb{N}$ im Register $r+1$ stehen soll.

Interessant ist eine Variante von Registermaschinen, die mittels LOOP-Programmen arbeiten. In LOOP-Programmen ist der bedingte Sprungbefehl verboten. Die Befehle sind auch nicht mehr durchnummeriert. Hinzu kommt stattdessen ein LOOP-Befehl mit einem Unterprogrammaufruf. Damit ist eine einfache Aufteilung eines Programms in eine Folge von Elementarbefehlen so nicht mehr möglich. Stattdessen muss man jetzt Befehle und Programme simultan definieren:

addiere 1 in Register i und
 subtrahiere 1 in Register i

sind Befehle und LOOP-Programme. Sind P und P' bereits LOOP-Programme, ist auch

$P; P'$

ein LOOP-Programm, und

loop R_i beginne P ende

ist ein Befehl und ein LOOP-Programm.

Die Semantik von loop R_i beginne P ende ist wie folgt: Besitzt das Register i zu Beginn der Ausführung dieses Befehls einen Wert x_i , so muss P genau x_i -mal hintereinander ausgeführt werden. Wir wollen die Mächtigkeit von LOOP-Programmen an einigen Beispielen zeigen.

loop R_i beginne
 subtrahiere 1 in Register i
 ende

löscht den Inhalt von Register i . Dieses LOOP-Programm werde mit

$x_i := 0$

abgekürzt. Das LOOP-Programm

$x_j := 0; x_k := 0;$
 loop R_i beginne
 subtrahiere 1 in Register i ;
 addiere 1 in Register j ;
 addiere 1 in Register k
 ende;
 loop R_k beginne
 subtrahiere 1 in Register k ;
 addiere 1 in Register i
 ende

kopiert den Inhalt von Register i in Register j , ohne den Inhalt von Register i zu verändern. Als Seiteneffekt wird dabei aber ein (Hilfs-)Register k geleert. Als k wählt man im Folgenden stets ein zuvor noch nicht benutztes Register, so dass dieser Seiteneffekt keine Wirkung hat. Dieses LOOP-Programm werde mit

$$x_j := x_i$$

abgekürzt, falls $i \neq j$ gilt. Für $i = j$ bezeichne $x_j := x_i$ ein Programm, das nichts tut, z.B. $x_j := x_j + 1$; $x_j := x_j - 1$.

$$x_i := x_j + x_k$$

bezeichne das folgende LOOP-Programm:

```

 $x_i := x_j$ ;
loop  $R_k$  beginne
  addiere 1 in Register  $i$ 
ende

```

Damit berechnet das LOOP-Programm

$$x_1 := x_1 + x_2; x_2 := 0$$

gerade die Additionsfunktion $+: \mathbb{N}^2 \rightarrow \mathbb{N}$. Die Argumente x und y für $x + y$ stehen zu Beginn der Rechnung in Register 1 und Register 2, alle weiteren eventuell notwendigen Register sind leer (tragen die Zahl 0). Am Ende der Rechnung steht dann $x + y$ in Register 1 und 0 in allen anderen.

$$x_i := x_j * x_k$$

bezeichne das folgende LOOP-Programm:

```

 $x_i := x_j$ ;  $x_h := x_k$ 
subtrahiere 1 in Register  $h$ ;
loop  $R_h$  beginne
   $x_i := x_i + x_j$ 
ende;
 $x_h := 0$ 

```

wobei x_h wieder ein Hilfsregister darstellt.

$$x_1 := x_1 * x_2; x_2 := 0$$

berechnet offensichtlich die Multiplikationsfunktion.

$$x_i := x_j^{x_k}$$

bezeichne das folgende LOOP-Programm:

```

 $x_i := 0;$ 
addiere 1 in Register  $i$ ;
loop  $R_k$  beginne
     $x_i := x_i * x_j$ 
ende

```

Offensichtlich berechnet nun

```

 $x_3 := x_1; x_1 := x_3^{x_2}; x_2 := 0; x_3 := 0$ 

```

die Potenzfunktion $f(x, y) := x^y$. Man kann auch den Prozess des Potenzierens weiter iterieren zu einer Funktion $f(x, y) := x^{x^{\dots^x}}$, induktiv definiert durch $f(x, 0) := x$ und $f(x, y+1) := f(x, y)^x$. Das folgende LOOP-Programm berechnet etwa diese iterierte Potenzierung:

```

 $x_3 := x_1;$ 
loop  $R_2$  beginne
     $x_1 := x_1^{x_3}$ 
ende;
 $x_2 := 0; x_3 := 0$ 

```

Das Ergebnis dieses recht einfachen LOOP-Programms ist nicht mehr durch eine Exponentialfunktion majorisierbar. Man kann also bereits mit sehr einfachen LOOP-Programmen extrem schnell wachsende Funktionen berechnen.

LOOP-Programme terminieren stets, da die Anzahl der Schleifendurchläufe durch einen LOOP-Befehl zu Beginn der Befehlsausführung bereits festliegt. Damit lassen sich mittels LOOP-Programmen nur *totale* zahlentheoretische Funktionen berechnen. Eine Funktion $f: \mathbb{N}^r \rightarrow \mathbb{N}$ heißt dabei total, wenn $f(x_1, \dots, x_r)$ für jedes Argumententupel $(x_1, \dots, x_r) \in \mathbb{N}^r$ auch definiert ist. Im Gegensatz dazu dürfen bei partiellen Funktionen $f: \mathbb{N}^r \rightarrow \mathbb{N}$ auch $f(x_1, \dots, x_r)$ für Argumententupel (x_1, \dots, x_r) undefiniert sein.

Wir nennen eine totale zahlentheoretische Funktion *rein hypothetisch berechenbar*, falls f zwar von einer Turingmaschine berechnet werden kann, aber von keinem LOOP-Programm.

Es ist ein Standardresultat der Theoretischen Informatik, dass die Klasse der LOOP-berechenbaren Funktionen gerade die Klasse der primitiv rekursiven Funktionen (vergleiche Definition 5.2.8) ist, die Klasse der (von Turingmaschinen) berechenbaren Funktionen aber mit der Klasse der sogenannten μ -rekursiven Funktionen übereinstimmt. Damit ist jede berechenbare, aber nicht primitiv rekursive Funktion eine rein hypothetisch berechenbare Funktion. Die Ackermannfunktion aus Definition 5.2.7 ist solch eine extrem komplexe Funktion, die nur rein hypothetisch berechenbar ist.

Man kann aber auch relativ unmittelbar eine rein hypothetisch berechenbare Funktion konstruieren. Dazu fassen wir ein jedes LOOP-Programm P mittels seines ASCII-Codes als eine Zahl $\gamma(P) \in \mathbb{N}$ auf. Aus $\gamma(P)$ lässt sich elementar das Programm P eindeutig rekonstruieren. Zu $k \in \mathbb{N}$ sei $g_k: \mathbb{N} \rightarrow \mathbb{N}$ wie folgt definiert:

Fall 1: k ist nicht die Gödelzahl $\gamma(P)$ eines LOOP-Programms P . Dann sei $g_k(n) := 0$ für alle $n \in \mathbb{N}$.

Fall 2: k ist die Gödelzahl $k = \gamma(P)$ eines LOOP-Programms P . P gestartet mit n in Register 1 und 0 in allen anderen Registern terminiere mit m in Register 1, dann sei $g_k(n) := m$.

Die Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$ mit

$$f(n) := g_n(n) + 1$$

ist nun rein hypothetisch berechenbar. Dass f überhaupt berechenbar ist, sieht man etwa wie folgt: Um $f(n)$ zu berechnen, kann eine geeignete (allerdings recht komplexe) Turingmaschine M zuerst überprüfen, ob n eine erlaubte Gödelzahl $\gamma(P)$ eines LOOP-Programms P ist. Falls ja, so interpretiert die Turingmaschine die Programmausführung von P gestartet mit der Konfiguration $(1, n, 0, \dots, 0)$. Terminiert die Programmausführung mit $(t, x_1, x_2, \dots, x_r)$ – bei $t = \text{HALT}$ als einzigem Haltebefehl und bei r Registern –, so setzt M nun $f(n)$ auf $x_1 + 1$.

Dass f nicht LOOP-berechenbar sein kann, ist hingegen unmittelbar klar: Angenommen es existiere ein LOOP-Programm P_f , das f berechnet. Es sei $n_f := \gamma(P_f)$. Da P_f f berechnet, gilt also $g_{n_f}(n) = f(n)$ für alle $n \in \mathbb{N}$. Aus der Definition von f ersehen wir aber $f(n) = g_n(n) + 1$. Insbesondere gilt

$$g_{n_f}(n_f) = f(n_f) = g_{n_f}(n_f) + 1,$$

ein Widerspruch.

Im Gegensatz hierzu ist der (viel ältere) Nachweis, dass die Ackermannfunktion berechenbar, aber nicht LOOP-berechenbar ist, anders strukturiert: Es kann gezeigt werden, dass die Ackermannfunktion „letztlich“ schneller wächst als jede LOOP-berechenbare Funktion. Damit passt der Ausdruck „rein hypothetisch berechenbar“ sehr exakt auf die Ackermannfunktion.

Ein Beispiel von formalen Kalkülen in der Theoretischen Informatik sind etwa *Grammatiken*. Eine Grammatik G besitzt zwei disjunkte Alphabete Σ (von sogenannten *terminalen Symbolen*) und V (von sogenannten *Variablen*) mit einem ausgezeichneten Symbol S (für *Start*) in V , sowie eine endliche Liste von Regeln $\{R_1, \dots, R_k\}$. Eine Regel R_i ist dabei ein Paar $R_i = (P_i, Q_i)$ von Wörtern $P_i, Q_i \in (\Sigma \cup V)^*$. Eine Regel (P_i, Q_i) wird auf ein Wort $w \in (\Sigma \cup$

$V)^*$ angewendet, indem nichtdeterministisch ein Vorkommen eines Teilwortes P_i in w durch Q_i ersetzt wird. Formal: Ein Wort $w' \in (\Sigma \cup V)^*$ heißt *direkter Nachfolger* von einem Wort $w \in (\Sigma \cup V)^*$, falls Wörter $u, v \in (\Sigma \cup V)^*$ existieren, so dass $w = uP_iv$ und $w' = uQ_iv$ gilt. Wörter spielen hier also die Rolle einer Konfiguration. Damit ist wie zuvor ein Rechnungsbegriff als eine Folge von direkt nachfolgenden Wörtern erklärt. Grammatiken *generieren* per Definition genau die Wörter $u \in \Sigma^*$, zu denen eine Rechnung vom Startwort S aus existiert. D.h., $L := \{w \in \Sigma^* \mid \text{es existiert eine Rechnung in } G \text{ von } S \text{ nach } w\}$ ist die von G erzeugte Sprache.

Ein Hauptsatz der Theoretischen Informatik besagt, dass genau die Sprachen von einer Grammatik erzeugbar sind, die auch von einer Turingmaschine akzeptiert werden. Die berühmte Chomsky-Hierarchie beschreibt, welche Sprachklassen von speziellen Grammatiken (d.h. mit eingeschränkten Regeln, etwa nur Regeln der Art (P_i, Q_i) mit $P_i \in V$ für kontextfreie Grammatiken) erzeugt werden, und welche Sprachklassen von eingeschränkten Varianten von Turingmaschinen akzeptiert werden.

1.2 Parallele Rechnungen

Die meisten existierenden Rechner besitzen eine sogenannte „von-Neumann-Architektur“: Eine sequentiell arbeitende zentrale Recheneinheit steuert diverse Peripherieeinheiten. Diese Zentraleinheit ist dabei der „Flaschenhals“ bei den Versuchen, die Rechnungen zu beschleunigen. Eine Abhilfe ist es, den Peripheriegeräten selbst eigenständige Rechenleistungen mitzugeben, die aber stets noch weitgehend kontrolliert von der zentralen Recheneinheit bleiben.

Mit dem Paradigma der parallelen Rechnung versucht man, diesen Flaschenhals wie folgt zu vermeiden: Eine Rechenaufgabe wird jetzt von mehreren gleichberechtigten Rechnern parallel behandelt. Das in der von-Neumann-Architektur unbekannte Problem der Koordinierung mehrerer prinzipiell gleichberechtigter Recheneinheiten tritt jetzt in den Vordergrund. Denkbar ist, dass eine Recheneinheit als *primus inter pares* diese Koordinationsaufgabe für alle übernimmt, oder dass – theoretisch viel anspruchsvoller – auch die Gesamtkoordination dezentral auf alle Recheneinheiten verteilt stattfindet.

Es wurden in den letzten Jahrzehnten vielfältige unterschiedliche Architekturen von Parallelrechnern entwickelt und auch gebaut. Unterschiede ergeben sich z.B. in den Kommunikationsmöglichkeiten und Speicherzugriffsmöglichkeiten. Ein interessantes Konzept ist eine Kommunikationsstruktur mittels Graphen: Jeder Knoten des Graphen ist ein selbständiger Rechner, jede Kante eine Kommunikationsmöglichkeit. Ein Knoten mit einem hohen Grad (das

ist die Anzahl der mit diesem Knoten verbundenen Kanten) besitzt viele direkte Nachbarrechner, mit denen er unmittelbar kommunizieren kann. Die in den 80er Jahren beliebten Transputer besaßen den Grad 4. D.h. jeder Transputer konnte mit 4 anderen Transputern (in beiden Richtungen) gleichzeitig Nachrichten austauschen. Transputer könnten zu einem beliebigen Graphen mit maximalem Grad 4 pro Knoten verschaltet werden.

Interessante theoretische Fragen sind die nach wegoptimalen Graphen, d.h. nach solchen Graphen eines festen, maximalen Grades, in dem (im Mittel oder im worst case) die Weglängen der kürzesten Wege zwischen je zwei Knoten minimal sind.

Unterschiedliche Parallelarchitekturen erhält man auch je nachdem, ob jeder Rechner ein eigenes lokales (privates) Speichermedium besitzt, oder ob alle Rechner auf ein zentrales (öffentliches) Speichermedium zugreifen. In einer PRAM-Architektur (Parallele-Random-Access-Maschine) z.B. können mehrere Rechner gleichzeitig auf eine zentrale Speichereinheit zugreifen, wobei die „Kosten“ (etwa in Zeit gemessen) für alle Rechner gleich und unabhängig vom Speicherplatz sein sollen. Ferner wird unterschieden, ob alle Rechner simultan das gleiche Programm ausführen, oder jeder Rechner ein anderes. Ebenso, ob alle Rechner von einer zentralen Uhr global getaktet werden, oder jeder Rechner seine eigene, lokale Taktung besitzt.

Von besonderem theoretischen, ja sogar philosophischem Interesse ist das Parallelrechnerkonzept der *zellularen Automaten*. Der zellulare Automat spielt für die Parallelrechnung eine ähnlich bedeutende Rolle wie die der Turingmaschine für die Theorie sequentieller Rechnungen.

Ein zellulärer Automat besteht aus einer zweidimensionalen Anordnung von Zellen, etwa wie auf einem Schachbrett. Jede Zelle selbst kann man sich als einen endlichen Automaten vorstellen (etwa eine Turingmaschine ohne Band), der mit den Automaten seiner Nachbarzellen kommuniziert. Der Nachbarschaftsbegriff ist homogen: alle Zellen besitzen die gleiche Nachbarschaftsstruktur. Alle Automaten arbeiten synchron. Zum Zeitpunkt t liest jeder Automat die gerade aktuellen Zustände seiner Nachbarautomaten, und ändert deterministisch in Abhängigkeit dieses Zustandsvektors synchron mit allen anderen Automaten zum Zeitpunkt $t + 1$ seinen Zustand. Auf allen Zellen liegt stets der gleiche endliche Automat, allerdings darf er sich auf jeder Zelle in einem anderen aktuellen Zustand befinden. Es wird in den Automaten nicht zwischen Speicherinhalt und Programmzeile unterschieden, nur nach internen Zuständen. Formal ist ein endlicher Automat ein Tupel $A = (K, I, \delta)$ bestehend aus einer endlichen Menge K von internen Zuständen, einem Inputalphabet I und einer (deterministischen) Übergangsfunktion $\delta: K \times I \rightarrow K$, die angibt, mit welchem Zustandswechsel A bei Eintreffen eines Signals $x \in I$ in einem Zustand $s \in K$ reagiert. In einem zellularen Automaten trägt also jede Zelle Kopien eines endlichen Automaten A . Als Input erhält der Auto-

mat jeweils die aktuellen Zustände seiner Nachbarautomaten. D.h., $I = K^n$ gilt, falls jede Zelle mit genau n Nachbarzellen verbunden ist. Standardnachbarschaften sind etwa

- die von-Neumann-Nachbarschaft: jede Zelle hat die vier Zellen unmittelbar links, rechts, oberhalb und unterhalb von sich als Nachbarn, plus sich selbst.
- die Moore-Nachbarschaft: jede Zelle besitzt zusätzlich zu den von-Neumann-Nachbarn auch die vier ihr diagonal benachbarten Zellen als Nachbarn.

Generell sind aber auch beliebige Nachbarschaftsbeziehungen definierbar. Zur Vereinfachung geben wir den Zellen Koordinaten (x, y) im \mathbb{Z}^2 . Eine Nachbarschaft N ist nun ein endlich-dimensionaler Vektor $N = (v_1, \dots, v_n)$ von Richtungen $v_i \in \mathbb{Z} \times \mathbb{Z}$. In einer Zelle $z = (x, y) \in \mathbb{Z}^2$ sind damit genau die Zellen $z + v_1, \dots, z + v_n$ Nachbarn. Die von-Neumann-Nachbarschaft wird also durch den 5-dimensionalen Vektor $((0, 0), (-1, 0), (1, 0), (0, -1), (0, 1))$ beschrieben, die Moore-Nachbarschaft durch den 9-dimensionalen Vektor $((0, 0), (-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1))$.

Es sei $I = \{1, \dots, n\} \times \{1, \dots, n\}$ ein endliches Quadrat in \mathbb{Z}^2 . Ein zellulärer Automat auf I (d.h. jede Zelle in I soll den gleichen endlichen Automaten tragen) ist damit beschreibbar durch

- eine Nachbarschaft $N = (v_1, \dots, v_n)$ mit $v_i \in \mathbb{Z}^2$,
- eine endliche Zustandsmenge K ,
- eine lokale Übergangsfunktion $\delta: K^n \rightarrow K$.

Eine Konfiguration C ist jetzt die Beschreibung der abstrakten Zustände aller Automaten auf I , $C: I \rightarrow K$. Eine Konfiguration C' ist direkter Nachfolger einer Konfiguration C , $C \vdash C'$, falls für alle $z \in I$ gilt:

$$C'(z) = \delta(C(z + v_1), C(z + v_2), \dots, C(z + v_n)).$$

Hierbei kann es zu Problemen bei der Randbehandlung kommen: $z + v_i$ muss nicht innerhalb von I liegen. In diesem Fall soll $C(z + v_i)$ stets auf einen ausgezeichneten Zustand $\#$ aus K gesetzt werden. $\#$ ist der sogenannte „Ruhezustand“, er entspricht genau der leeren Zelle bei Turingmaschinen.

Das hier vorgestellte Konzept eines zellulären Automaten auf einem fixen Quadrat I ist aber für beliebige Rechnungen unzureichend. Es fehlt noch ein potentiell unendliches Speichermedium. Hier wird die Vorstellung verfolgt, dass bei Bedarf am Rand von I neue Zellen angefügt werden können. Mathematisch elegant lässt sich das präzisieren, indem man $I = \mathbb{Z}^2$, also unendlich

groß, setzt. Als Konfiguration werden nur solche Abbildungen $C: \mathbb{Z}^2 \rightarrow K$ zugelassen, für die $C(z) = \#$ für fast alle $z \in \mathbb{Z}^2$ gilt. D.h., nur in einem endlichen Teilgebiet des \mathbb{Z}^2 befinden sich Zellen nicht im Ruhezustand. Ferner wird noch $\delta(\#, \dots, \#) = \#$ gefordert. D.h. ein Automat, dessen Nachbarn alle im Ruhezustand sind, bleibt selbst im nächsten Takt im Ruhezustand. Da zu jeder Konfiguration C stets genau eine direkte Nachfolgekonfiguration existiert, sind alle Rechnungen deterministisch. Der Übergang von einer Konfiguration zur eindeutigen Nachfolgekonfiguration wird als Takt interpretiert. Da bei $C \vdash C'$ C' aus C durch simultane Anwendung der lokalen Übergangsfunktion δ in allen Zellen entsteht, haben wir modelliert, dass alle Automaten in allen Zellen simultan im Takt schalten. Zu beachten ist, dass zu jedem Taktschritt nur endlich viele der Zellen nicht im Ruhezustand $\#$ sind. Anders gesagt, jeweils nur endlich viele der Zellautomaten sind aktiviert. Als *Muster* eines zellularen Automaten wird eine Konfiguration ohne die Zellen im Ruhezustand aufgefasst, wobei von den absoluten Koordinaten der Zellen abstrahiert wird. Muster sind damit stets endliche 2-dimensionale Objekte. Betrachten wir ein Beispiel. Es sei $K = \{0, 1, 2\}$, 0 soll der Ruhezustand $\#$. Ein Muster ist etwa



Es kann z.B. aus der Konfiguration $C: \mathbb{Z}^2 \rightarrow K$ mit $C((0,0)) = 1$, $C((1,0)) = 1$, $C((2,-1)) = 1$, $C((3,0)) = 2$ und $C(z) = 0$ sonst entstanden sein, oder aus jeder anderen Konfiguration C' , die aus C durch eine Verschiebung, $C'(z) = C(z+b)$ für ein $b \in \mathbb{Z} \times \mathbb{Z}$, hervorgeht.

Zellulare Automaten waren ein frühes Modell zur Untersuchung nach theoretischen Fragen zur Selbstorganisation. So kann man die Frage stellen, ob in zellularen Automaten eine Verdopplung von Mustern möglich ist. Eine erstaunliche Antwort bietet der zellulare Automat, der stets modulo einer Primzahl zählt. Er ist in der Lage, ein jedes Muster zu reproduzieren. Dabei ist letztlich sogar die gewählte Nachbarschaftsbeziehung irrelevant. Wir wollen nur ein Beispiel vorstellen. Es sei Z_p der zellulare Automat mit

- von-Neumann-Nachbarschaft $N = (v_1, \dots, v_5)$,
- $K = \{0, 1, \dots, p-1\}$, 0 sei der Ruhezustand $\#$, p irgendeine Primzahl,
- $\delta: K^5 \rightarrow K$ sei definiert als $\delta(s_1, s_2, s_3, s_4, s_5) = \sum_{i=1}^5 s_i \bmod p$.

Z_p vervielfacht nach endlich vielen Schritten jedes endliche Muster. Abbildung 1.1 gibt ein Beispiel mit fünf Mustern. Die Muster sind jeweils als Bild dargestellt. 0 ist weiß, also „unsichtbar“, $p-1$ ist schwarz, g für $0 < g < p-1$ ist ein Grauwert, hell für g nahe 0, dunkel für g nahe $p-1$. Im Beispiel ist $p = 13$, d.h. alle Muster sind Grauwertbilder mit Grauwerten $\{0, 1, \dots, 12\}$.

Üblich sind zwar in der Bildverarbeitung z.B. 16 Grauwerte, aber wir brauchen hier eine Primzahl. Das Muster M_0 ist das Ausgangsmuster, die Muster M_1 und M_2 zeigen, wie sich das Muster M_0 nach einem bzw. zwei Takten verändert. D.h. $M_0 \vdash M_1 \vdash M_2$ gilt. M_{168} und M_{169} zeigen das Muster nach $13^2 - 1 = 168$ und $13^2 = 169$ Schritten. Hier haben sich jetzt 5 Kopien von M_0 gebildet. Diese Verfünfachung gelingt für jedes Muster.

Es ist wohl eher eine bizarre Eigenschaft dieser modulo einer Primzahl zählenden zellularen Automaten, dass sie stets beliebige Muster vervielfältigen. Weniger zeigen sich in diesem Beispiel aber selbstorganisierende Prinzipien. Von Neumann stellte bereits in den 50er Jahren des 20. Jahrhunderts die Frage nach „künstlichem Leben“, das er als Organisationsformen definierte, die

- sich selbst reproduzieren können, und
- die Rechnungen beliebiger Turingmaschinen simulieren können.

Der erste Punkt allein wird auch von „leblosem“ Kristallwachstum erfüllt. Mit dem zweiten Punkt sollen die selbstreproduzierenden Organisationsformen auch sinnvolle Aufgaben erledigen können. Ulam schlug Muster in zellularen Automaten als solche Organisationsformen vor. Von Neumann gelang es nun, einen zellularen Automaten mit der von-Neumann-Nachbarschaft und 28 Zuständen sowie ein endliches Muster M zu konstruieren, so dass folgendes galt:

- Es existiert ein einfaches Muster S (bestehend nur aus einer Zelle), so dass eine Rechnung von einer Komposition $M \circ S$ der beiden Muster M und S aus in zwei räumlich getrennten Vorkommen von M resultiert.
- Zu jeder Turingmaschine T existiert ein Muster $\gamma(T)$ (das „Gödelmuster von T “), und zu jedem Inputwort w für T existiert ein Gödelmuster $\gamma(w)$, so dass eine Rechnung von einer Komposition $M \circ \gamma(T) \circ \gamma(w)$ der drei Muster M , $\gamma(T)$ und $\gamma(w)$ in einem Muster $M \circ \gamma(T) \circ \gamma(u)$ resultiert, wobei u genau das Ergebnis der Rechnung von T gestartet mit Input w ist.

In einem gewissen Sinn ist dieses Muster M zur Selbstreproduktion (bei „Anstoß“ durch eine zusätzliche Inputzelle S) fähig und zur Berechnung einer jeglichen berechenbaren Funktion ($\gamma(T)$ und $\gamma(w)$ sind dabei „triviale“ Muster, die die Berechnung nicht selbst leisten, sondern nur M mitteilen, welche berechenbare Funktion mit welchem Argument M gerade berechnen soll). Damit ist dies ein Beispiel von abstraktem „Artificial Life“.

Es stellt sich die Frage, ob mit diesen mächtigen Eigenschaften zellulare Automaten prinzipiell mehr berechnen können als Turingmaschinen. Die Antwort ist aber Nein. Die Rechnung beliebiger Turingmaschinen kann in zellularen

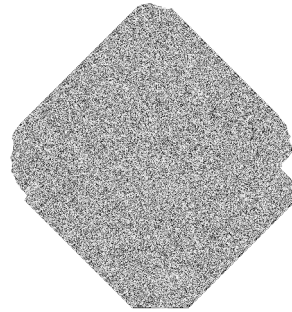
M_0 : M_1 : M_2 : M_{168} : M_{169} :

Abb. 1.1. Ein Beispiel für Reproduktion durch einen zellularen Automaten. Die fünf Bilder M_0 , M_1 , M_2 , M_{168} und M_{169} zeigen das Muster am Anfang und nach 1, 2, 168 und 169 Taktschritten.

Automaten ausgeführt werden und umgekehrt: Jeder zellulare Automat kann von einer Turingmaschine simuliert werden. Dies ist ganz offensichtlich, da man die global synchronisierten Arbeiten aller Automaten (zu jedem Zeitpunkt einer Rechnung sind nur endlich viele aktiv) leicht auf einer normalen Rechenanlage simulieren kann, die selbst wiederum als Konkretisierung des Konzepts des sequentiellen Rechenparadigmas nicht mehr zu leisten vermag als eine Turingmaschine. Es ist sogar so, dass zellulare Automaten prinzipiell nicht „deutlich“ schneller rechnen können als Turingmaschinen: Es ist durch diese Art von Parallelrechnung nur maximal ein quadratischer Gewinn an Rechengeschwindigkeit möglich. Dies sieht man leicht wie folgt:

Betrachten wir etwa die Moore-Nachbarschaft. Zum Zeitpunkt $t = 0$ sei genau eine Zelle Z_0 aktiv, alle anderen seien im Ruhezustand $\#$. Dann sind zum Zeitpunkt $t = 1$ maximal alle Zellen Z in der Moore-Nachbarschaft $N(Z_0)$ um Z_0 aktiv, das sind genau 9 Zellen. Zum Zeitpunkt $t = 2$ können dann maximal alle Zellen in der Moore-Nachbarschaft $N(N(Z_0))$ um $N(Z_0)$ aktiv sein, also 25 Zellen. Zum Zeitpunkt t sind also maximal die Zellen in $N^t(Z_0)$, der t -fachen Moore-Nachbarschaft um Z_0 , bestehend aus $(2t + 1)^2$ Zellen, aktiv. Die Anzahl der zusätzlich in einer Rechnung zur Verfügung stehenden Zellen wächst also nur quadratisch mit der Rechenzeit. Dies gilt modulo konstanter Faktoren für jede Nachbarschaft.

Halten wir also fest, dass der Übergang vom sequentiellen Modell der Turingmaschine zum parallelen Modell der zellularen Automaten maximal einen quadratischen Gewinn in der Rechenzeit ergibt. Da die Polynome und Exponentialfunktionen bei Quadratur weiterhin von Polynomen und Exponentialfunktionen majorisierbar sind, ändern sich nicht einmal die Komplexitätsklassen P , NP , $EXPTIME$ und $EXPSPACE$, wenn man anstelle von Turingmaschinen zellulare Automaten als Rechner zulässt.

Ein ganz andersartiger Ansatz für parallele Berechnungen liegt im Konzept des *Quantenrechners* vor. Hierbei soll das quantenphysikalische Prinzip der *Überlagerung von Zuständen* ausgenutzt werden. Ein *Qubit* (Quantenbit) ist dabei eine Speicherzelle, die nicht nur zwei Zustände, 0 und 1, speichern kann, sondern auch Überlagerungen $\alpha 0 + \beta 1$ von 0 und 1. α und β sind hierbei komplexe Zahlen mit $\alpha^2 + \beta^2 = 1$. Die quantenphysikalische Interpretation ist, dass ein Qubit gleichzeitig 0 und 1 speichern kann, jeweils mit unterschiedlichen *Amplituden* α und β . Erst beim Auslesen entscheidet sich das Qubit eindeutig für einen der überlagerten Zustände 0 oder 1. und zwar für 0 mit der Wahrscheinlichkeit α^2 und für 1 mit der Wahrscheinlichkeit $\beta^2 = 1 - \alpha^2$. Dies entspricht dem Prinzip, dass Quantenvorgänge erst durch eine Beobachtung aus einem überlagerten Zustand in einen exakten, messbaren Zustand übergehen. Ein *Quantenregister* Q der Länge n ist nun eine physikalische Anordnung von n Qubits derart, dass in Q gleichzeitig alle 2^n Zustände $s \in \{0, 1\}^n$ speicherbar sind und Q erst durch den Akt des Ausle-

sens in einen festen dieser 2^n Zustände übergeht. Ein *Quantenrechner* kann nun in einem Schritt alle überlagerten 2^n Zustände verschieden manipulieren. Damit ist prinzipiell ein exponentieller Gewinn in der Rechengeschwindigkeit möglich. Es ist noch nicht klar, ob Quantenrechner jemals technisch realisierbar sein werden. Einige theoretische Grenzen sind jedoch bekannt: So kann ein Quantenrechner nicht mehr berechnen als ein klassischer Rechner. Auch ist prinzipiell höchstens ein exponentieller Gewinn in der Rechengeschwindigkeit bei Quantenrechnern gegenüber von-Neumann-Rechnern möglich, und das auch nur bei manchen algorithmischen Aufgaben. Zu diesen Aufgaben gehört aber die Primfaktorzerlegung einer Zahl: Mit einem Algorithmus von Shor kann mit einem geeigneten Quantenrechner jede Zahl n in n Schritten in ihre Primfaktoren zerlegt werden. Dass mit diesem Algorithmus dann viele verwendete Verschlüsselungsverfahren in Handel, Industrie und Militär auf einem Quantenrechner effektiv lösbar würden, trägt nicht unerheblich zum Interesse an Quantenrechnern bei.

Man hat auch parallel arbeitende Grammatiken studiert. Hierbei werden alle anwendbaren, lokalen Ersetzungen in einem Schritt simultan ausgeführt. Natürlich ist dabei das offensichtliche Problem von Überlappungen zu lösen. So erzeugt die einfache Regel (aba, cd) angewandt auf das Wort $ababa$ zwei direkte Nachfolger (im Sinne sequentieller Grammatiken), nämlich $cdba$, falls das erste Vorkommen von aba durch cd ersetzt wird, und $abcd$, falls das zweite Vorkommen von aba ersetzt wird. Was soll aber unter einer simultanen Anwendung der Regel auf beide Vorkommen verstanden werden? Lässt man nur kontextfreie Regeln der Form (P, Q) mit $P \in V$ zu, so stellt sich dieses Problem nicht. Man erhält dann die sogenannte Klasse der *Lindenmeyersysteme* mit interessanten Anwendungen etwa in der Biologie oder der Theorie der Formalen Sprachen.

1.3 Nebenläufige Rechnungen

Man spricht häufig von *verteilten Rechnungen* bei Modellen von parallelen Rechnungen ohne einen gemeinsamen Takt zur Ausführung der Einzelrechnungen. Nehmen wir wieder eine einfache Grammatik als Beispiel. Die Regeln $R_1 = (ab, aa)$ und $R_2 = (ba, bb)$ können auf das Wort $abcba$ an zwei Stellen angewandt werden. Direkte Nachfolger von $abcba$ sind somit $aacba$ und $abcbb$. In einem Modell von getakteten Parallelrechnungen wäre $aacbb$ der einzige direkte Nachfolger von $abcba$. In verteilten Rechnungen wären alle drei Wörter $aacba$, $abcbb$ und $aacbb$ direkte Nachfolger von $abcba$. Welches sind nun die direkten Nachfolger von $abab$? In sequentiellen Rechnungen sind das $aaab$, $abbb$ und $abaa$, da R_1 an zwei und R_2 an einer Stelle in $abab$ angewendet werden kann. Beide Anwendungen von R_1 sind in $abab$ auch gleichzeitig möglich,

da sie sich nicht gegenseitig beeinflussen. Das gilt aber nicht für eine gemeinsame Anwendung von R_1 und R_2 , da eine Anwendung von R_2 die von R_1 verhindert. Statt von gleichzeitig spricht man auch von *nebenläufig*. Damit ist R_1 im Wort *abab* an zwei Stellen nebenläufig anwendbar (R_1 ist zu sich selbst nebenläufig, dies wird auch als auto-nebenläufig bezeichnet), die beiden Regeln R_1 und R_2 sind zueinander in *abab* nicht nebenläufig. Direkte Nachfolger von *abab* in verteilten Rechnungen sind also *aaab*, *abbb*, *abaa* und *aaaa*. Man spricht nun von *nebenläufigen Rechnungen* bei Untersuchungen von verteilten Rechnungen, wenn auf den Aspekt der Nebenläufigkeit Wert gelegt werden soll.

So, wie Turingmaschinen ein mathematisch formales Modell für sequentielle Rechnungen sind, sind *Petri-Netze* ein mathematisch formales Modell für nebenläufige Rechnungen. Nur: Bei Petri-Netzen wird die Ähnlichkeit zu Rechnern weitgehend aufgegeben. Dies ist auch nicht verwunderlich, da hier ja verteilte Rechnungen ohne zentrale Recheneinheit modelliert werden. Eher besitzen Petri-Netzen eine Ähnlichkeit zu zellularen Automaten, allerdings ohne homogene Nachbarschaftsstruktur. Vielmehr wird die Nachbarschaft durch generelle Graphstrukturen beschrieben.

Ohne die formale Definition eines Petri-Netzes in Kapitel 3 vorwegnehmen zu wollen, kann man ein Petri-Netz als einen endlichen, gerichteten, zweifarbenen Graphen auffassen. Zweifarben bedeutet dabei, dass alle Knoten des Graphen mit genau einer von zwei Farben so eingefärbt sind, dass keine Kante zwei Knoten mit gleicher Farbe verbindet. Die beiden Farben werden *Place* und *Transition* genannt. Knoten mit der Farbe „Place“ (bzw. „Transition“) werden auch als Places (bzw. Transitionen) bezeichnet. Graphisch werden Places stets als Kreise, Transitionen hingegen als Balken oder Rechtecke dargestellt. Transitionen werden formale Gegenstücke zu Agenten in verteilten Systemen, Places hingegen zu Speicherzellen. Dazu darf jeder Place eine Anzahl $n \geq 0$ von sogenannten *Token* speichern. Alle Places, von denen gerichtete Pfeile zu einer Transition t hinführen, werden als *Inputplaces* oder *Vorbedingungen* von t bezeichnet. Alle Places zu denen Pfeile von t aus hinführen, heißen die *Outputplaces* oder *Nachbedingungen* von t . Dabei darf ein Place auch gleichzeitig Input- und Outputplace einer Transition sein. Ein Inputplace, von dem genau k gerichtete Pfeile zu t führen, sei kurzfristig k -facher Inputplace genannt (diese Bezeichnung wird später nicht weiter verwendet), analog für k -fache Outputplaces. Alle Transitionen spielen nun weitgehend unabhängig das folgende *Tokenspiel* auf dem Petri-Netz: liegen auf allen Inputplaces von t Token, und zwar mindestens k Token auf jedem k -fachen Inputplace, so heißt t *feuerbar*. Eine feuerbare Transition t feuert, indem sie in einem Schritt k Token von jedem k -fachen Inputplace entfernt und m Token auf jeden m -fachen Outputplace legt. Abbildung 1.2 zeigt ein Beispiel. Die Anzahl der aktuellen Token auf einem Place wird durch eine entsprechende Anzahl von Punkten auf diesem Kreis graphisch dargestellt.

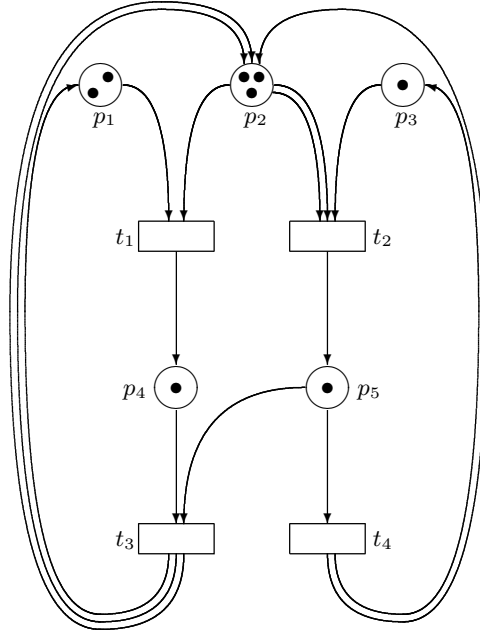


Abb. 1.2. Ein Petri-Netz N

In N sind aktuell alle vier Transitionen feuerbar, t_1 und t_2 auch nebenläufig, ebenso wie t_1, t_2 und t_3 oder t_1, t_2 und t_4 . t_3 und t_4 sind aber nicht nebenläufig feuerbar, da ein Feuern von t_3 (oder t_4) den einzigen Token von p_5 entfernt und damit das Feuern der anderen Transition t_4 (oder t_3) unmöglich macht.

Durch ein nebenläufiges Feuern von t_1, t_2 und t_3 erhalten wir das Petri-Netz aus Abbildung 1.3, durch ein nebenläufiges Feuern von t_1, t_2 und t_4 das aus Abbildung 1.4.

Hierbei müssen nicht stets mehrere nebenläufig feuerbare Transitionen feuern. Auch ein einzelnes Feuern ist gestattet, was der Idee Rechnung trägt, verteiltes Verhalten ohne globalen Takt modellieren zu wollen.

Es bietet sich an, die Places eines Petri-Netzes fest durchzunummerieren, etwa als p_1, \dots, p_n . Damit kann man mit einem einfachen n -dimensionalen Vektor $s \in \mathbb{N}^n$ über \mathbb{N} beschreiben, wieviele Token welcher Place aktuell trägt. Diese sogenannten *Markierungen* oder *Zustände* stellen damit die Konfigurationen eines Petri-Netzes dar, da durch sie allein das potentielle zukünftige Verhalten des Petri-Netzes festgelegt ist. Wir wollen mit $s \vdash_t s'$ bzw. $s \vdash_{t_1, t_2} s'$ ausdrücken, dass das Feuern von t im Zustand s möglich ist und im neuen Zustand s' resultiert, bzw. dass im Zustand s ein nebenläufiges Feuern

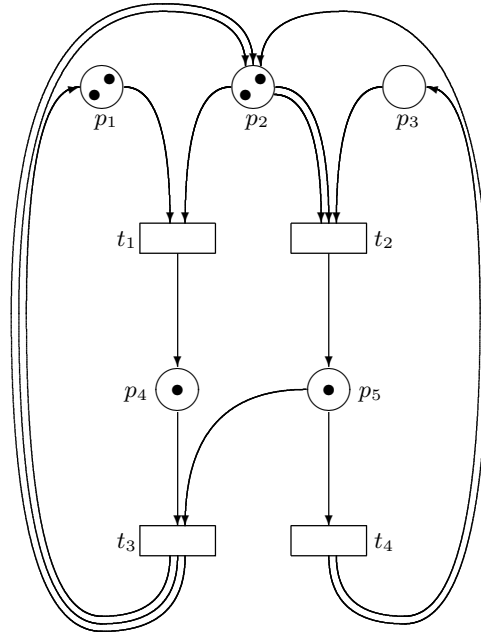


Abb. 1.3. N nach Feuern von t_1, t_2, t_3

von t_1, t_2 möglich ist und im neuen Zustand s' resultiert. Dann ist für das Netz N aus Abbildung 1.2, das sich im Zustand $(2, 3, 1, 1, 1)$ befindet, beispielsweise folgende Rechnung möglich:

$$\begin{aligned} (2, 3, 1, 1, 1) &\vdash_{t_1, t_2} (1, 0, 0, 2, 2) \vdash_{t_3} (2, 2, 0, 1, 1) \\ &\vdash_{t_4} (2, 3, 1, 1, 0) \vdash_{t_1, t_1} (0, 1, 1, 3, 0). \end{aligned}$$

Die Rechnung kann im Zustand $(0, 1, 1, 3, 0)$ nicht fortgesetzt werden, da jetzt keine Transition mehr feuierbar ist.

Während die Modelle der Turingmaschine und des zellularen Automaten unterschiedliche Rechenparadigmen modellieren – einmal sequentielle, einmal parallele Rechnungen – so sind sie doch bezüglich ihrer prinzipiellen Berechenbarkeitseigenschaften äquivalent. Sogar hinsichtlich ihrer Rechengeschwindigkeit unterscheiden sie sich im wesentlichen nur durch einen quadratischen Gewinn. Bei Petri-Netzen ist die Situation anders. Petri-Netze sind kein universelles Modell für Berechenbarkeit. Zwar kann man mit Turingmaschinen auch die Rechnungen von Petri-Netzen simulieren – natürlich unter Verzicht auf die Ausdruckbarkeit von Nebenläufigkeit und verteilten Rechnungen –, umgekehrt geht das aber nicht. Der Grund dafür ist eine gewisse Monotonie

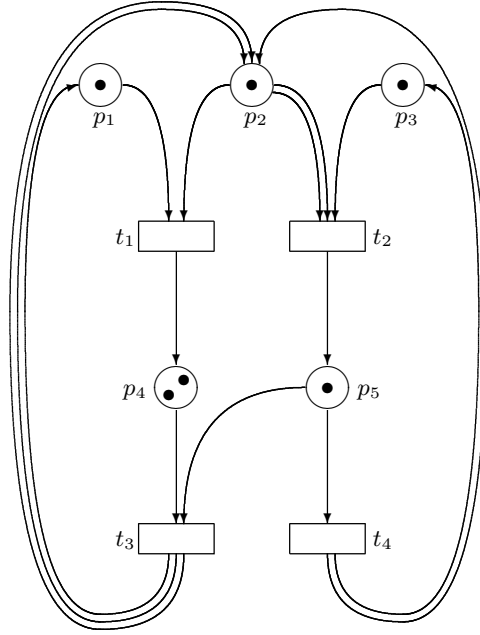


Abb. 1.4. N nach Feuern von t_1, t_2, t_4

im Verhalten von Petri-Netzen: Ist eine Transition in einem Zustand $s_1 \in \mathbb{N}^n$ feuerebar, so auch in jedem größeren Zustand $s_2 \geq s_1$. Gilt $s_1 \vdash_t s'_1$, $s_2 \geq s_1$, so gilt auch $s_2 \vdash_t s'_1 + (s_2 - s_1)$. Insbesondere können Petri-Netze nicht Befehle

j : falls $R_i = 0$ dann gehe zu j_1 sonst gehe zu j_2

simulieren. Ein formaler Beweis dafür ist allerdings aufwendig, da der Begriff „simulieren“ erst geeignet präzisiert werden muss und man sicher sein sollte, auch exotische Simulationskonzepte mit berücksichtigt zu haben. So gelingt der Beweis auch nur indirekt: Da das Konfigurationserreichbarkeitsproblem (die Frage, ob für zwei Konfigurationen C und C' eine Rechnung von C nach C' existiert) für Registermaschinen unentscheidbar ist, sollte das entsprechende Erreichbarkeitsproblem für Markierungen (die Konfigurationen von Rechnungen in Petri-Netzen) auch unentscheidbar sein, falls Petri-Netze beliebige Rechnungen von Registermaschinen simulieren könnten. Wir werden aber zeigen, dass das Erreichbarkeitsproblem für Petri-Netze entscheidbar ist. Also haben Petri-Netze nicht die gleichen Berechenbarkeitsfähigkeiten wie Turingmaschinen oder Registermaschinen. Die Frage nach der Entscheidbarkeit des Petri-Netz-Erreichbarkeitsproblems war viele Jahre offen. Der Nachweis der Entscheidbarkeit gelang schließlich Mayr in seiner Dissertation. Allerdings ist der dazu verwendete Algorithmus nicht primitiv rekursiv, d.h. er ist nur

hypothetisch berechenbar. Bis jetzt konnte weder ein primitiv rekursives Entscheidungsverfahren für das Erreichbarkeitsproblem gefunden werden, noch konnte gezeigt werden, dass das Erreichbarkeitsproblem eventuell gar nicht primitiv rekursiv lösbar ist. Wir werden aber zeigen können, dass **EXPSPACE** eine untere Schranke der Komplexität des Erreichbarkeitsproblems ist. D.h., ein jeder Algorithmus, der das Erreichbarkeitsproblem löst, braucht für eine Antwort, ob ein Zustand s in einem Petri-Netz von einem Zustand s_0 aus erreichbar ist, in fast allen Fällen mindestens exponentiellen Speicherplatz (gemessen in der „Größe“ von s , s_0 und N). (Zu beachten ist, dass hier stets endlich viele Ausnahmen erlaubt sein müssen, da ein Algorithmus ja für endlich viele Instanzen des Problems die Lösung fest gespeichert haben kann.) Obwohl wegen der Monotonie des Tokenspiels das Erreichbarkeitsproblem letztendlich lösbar ist, entziehen sich Petri-Netze einem einfachen mathematischen Zugang. Im Gegenteil, viele Petri-Netz-Fragen liegen an der Grenze des Entscheidbaren und Unentscheidbaren, andere an der Grenze des Berechenbaren und nur hypothetisch Berechenbaren. Einige Beispiele sollen das erläutern.

Es sei die Erreichbarkeitsmenge $\mathcal{E}(N, s_0)$ eines Petri-Netzes N die Menge aller Zustände, die durch Rechnungen von N ausgehend von einem festen Startzustand s_0 in N erreicht werden können. Die Frage, ob ein gegebener Zustand s in $\mathcal{E}(N, s_0)$ liegt, ist entscheidbar. Die Frage, ob für zwei Petri-Netze N_1 , N_2 mit gleicher Anzahl von Places aber $\mathcal{E}(N_1, s_0) = \mathcal{E}(N_2, s_0)$ gilt, ist unentscheidbar. Ein Petri-Netz N heißt k -beschränkt, falls $\mathcal{E}(N, s_0) \subseteq \{1, \dots, k\}^n$ ist, d.h. falls keine Rechnung in N von s_0 aus mehr als maximal k Token auf einen Place legen kann. N heißt beschränkt, falls ein k existiert, so dass N k -beschränkt ist. Für beschränkte Petri-Netze ist $\mathcal{E}(N, s_0)$ stets endlich und aus N (und s_0) berechenbar. Damit ist das Gleichheitsproblem $\mathcal{E}(N_1, s_0) = \mathcal{E}(N_2, s_0)$ für beschränkte Petri-Netze N_1 und N_2 stets entscheidbar. Wir werden aber zeigen, dass kein primitiv rekursiver Algorithmus dieses Gleichheitsproblem lösen kann. Das Gleichheitsproblem für beschränkte Petri-Netze ist sogar eines der wenigen bekannten „natürlichen“ Fragestellungen, von denen bekannt ist, dass sie zwar lösbar, aber nur rein hypothetisch lösbar sind.

Was ist nun der adäquate Rechnungsbegriff für Petri-Netze? Wir haben einen direkten Nachfolger s' eines Zustands s durch Feuern einiger nebenläufiger Transitionen erklärt, wie etwa $(1, 0, 0, 2, 2)$ direkter Nachfolger von $(2, 3, 1, 1, 1)$ durch Feuern von $\{t_1, t_2\}$ im Beispiel war. Eine Rechnung ist hier eine Folge $s_1 T_1 s_2 T_2 \dots s_n T_n s_{n+1}$ von Zuständen s_1, \dots, s_{n+1} und Mengen von im jeweiligen Zustand nebenläufigen Transitionen T_1, \dots, T_n , so dass $s_i \vdash_{T_i} s_{i+1}$ für $1 \leq i \leq n$ gilt. Da mit s_i und T_i stets auch s_{i+1} eindeutig festgelegt ist (bei Wissen, welche Transitionen in welchem Zustand feuern, legt das Tokenspiel deterministisch den neuen Zustand fest), genügt $s_1, T_1, T_2, \dots, T_n$ als Angabe einer Rechnung, oder einfacher, das Wort $T_1 \dots T_n$ und der

Anfangszustand s_1 . Ein solches Wort $T_1 \dots T_n$ heißt auch *Step-Feuersequenz* von s_1 aus. Erlaubt man in jedem Rechenschritt nur die Ausführung einer Transition, d.h. $|T_i| = 1$ soll gelten, erhält man sogenannte *Feuersequenzen*. Es ist für die meisten theoretischen Konzepte irrelevant, ob man mit Feuersequenzen oder Step-Feuersequenzen arbeitet. Petri-Netze wollen ja nicht Konzepte der Gleichzeitigkeit untersuchen, sondern solche der Nebenläufigkeit. Und die Nebenläufigkeit zweier Transitionen in einem Zustand wird nicht durch das Feuern der einen aufgehoben, ist also mittels Feuersequenzen im Prinzip genauso gut studierbar wie mittels Step-Feuersequenzen.

Es empfiehlt sich häufig, unterschiedliche Transitionen eines Petri-Netzes als unterschiedliche Realisierungen desselben Agenten aufzufassen – ganz analog wie im Beispiel von verteilten Grammatiken im Übergang $abab \vdash aaaa$ zwei Instanzen der gleichen Regel (ab, aa) an zwei Stellen angewendet werden. Dies modelliert man leicht durch zusätzliche, nicht notwendig injektive Beschriftungen der Transitionen. Zwei unterschiedliche Transitionen mit der gleichen Beschriftung werden also von einer äußeren Warte als identisch aufgefasst. Dabei soll es erlaubt sein, mittels spezieller Namen (τ für unsichtbar, dabei wird τ wie das leere Wort behandelt) Transitionen für die äußere Sichtweise zu verbergen. Die Menge aller Beschriftungen aller vom Startzustand aus feuerbaren (Step-) Feuersequenzen ist dann die (Step-) Sprache eines Petri-Netzes.

Damit ist das theoretisch gut untersuchte und auch mächtige Werkzeug der Formalen Sprache in der Petri-Netz-Theorie anwendbar. Wir werden z.B. zeigen, dass Petri-Netz-Sprachen als Abschluss von Dyck-Sprachen unter einigen einfachen, klassischen Sprachoperationen gewonnen werden können.

In der Theorie der Petri-Netz-Sprachen werden Konzepte der Theorie sequentieller Rechnungen, hier Wörter bzw. Feuersequenzen, zur Beschreibung nebenläufiger Konzepte verwendet. Dabei stellt sich sofort die Frage, ob als Konzept einer Rechnung eines Petri-Netzes nicht auch ein inhärent nebenläufiges Modell gewählt werden sollte. Ein solches inhärent nebenläufiges Rechnungskonzept, nämlich *Pomsets*, werden wir vorstellen. In einem Pomset wird von den feuerbaren Transitionen abstrahiert, unter Beibehaltung der Kausalitäts- und dynamischen Nebenläufigkeitsstruktur des Netzes. In dieser sogenannten „true concurrency“ Semantik von Petri-Netzen ersetzen alle zulässigen Pomsets die Sprache der feuerbaren Feuersequenzen. Feuersequenzen werden im Gegensatz dazu als ein *Interleaving* Modell von Petri-Netzen bezeichnet. Wir werden zeigen, dass solch eine „nebenläufige Sprachtheorie“ von Pomsets ganz ähnliche algebraische Charakterisierungen erlaubt wie die Interleaving Sprachtheorie von Feuersequenzen.

Petri-Netze wurden in den 60er Jahren des 20. Jahrhunderts an der GMD, Bonn, und am MIT, Cambridge, USA, im Gedankenaustausch mit der GMD entwickelt. Die Forscher der ersten Stunden waren C.A. Petri, H. Genrich

und K. Lautenbach an der GMD, Holt und Commoner von der Firma Applied Data Research, und Hack und Patil am MIT. Der Name „Petri-Netze“ zu Ehren Petris wurde von Holt eingeführt. In seiner Dissertation untersucht Petri Fragen der Kommunikation in nebenläufigen Systemen und stellt hier sogenannte „Aktionsnetze“ vor, in denen bereits die Grundelemente der späteren Petri-Netze in anderer Notation verwendet werden. An der GMD begannen Arbeiten zu einer Invariantentheorie von Petri-Netzen und zur Klärung der Phänomene wie Kausalität und Nebenläufigkeit. Am MIT nahmen die Untersuchungen eine andere Richtung. Es wurden Verbindungen von Petri-Netzen zu asynchronen Schaltwerken, die Komplexität von Algorithmen auf speziellen Teilklassen von Petri-Netzen sowie Petri-Netz-Sprachen untersucht. In nur wenigen Jahren wurde die Petri-Netz-Theorie die erfolgreichste Methode zum Studium verteilter Rechnungen mit Tausenden von Publikationen und zahlreichen Forschergruppen in aller Welt.

Ein Lehrbuch, das auch nur annähernd den Versuch unternehmen würde, die wichtigsten Ergebnisse darzustellen, bräuchte mehrere Bände von einigen tausend Seiten an Umfang. Wir beschränken uns hier nur auf einen kleinen Ausschnitt, nämlich fundamentale Ergebnisse zu Fragen der Entscheidbarkeit, Unentscheidbarkeit und Komplexität von Petri-Netz-Problemen, sowie eine Verallgemeinerbarkeit der Semantik sequentieller Petri-Netz-Rechnungen (d.h. der klassischen Interleaving Petri-Netz-Sprachtheorie) auf true-concurrency Modelle, und hier auch nur Pomset-Modelle. Interessante Untersuchungen wie etwa zur axiomatischen Theorie von Kausalität, Verbindungen zu asynchronen Schaltwerken, Algebren von weiteren Petri-Netz-Prozessen, semantische Äquivalenzen, wie etwa Bisimulation, behavioristische Simulation, Trace-Äquivalenz, etc., bleiben völlig unberücksichtigt. Die Auswahl hat sich auch danach gerichtet, dass viele Resultate aus der amerikanischen Schule in Lehrbüchern in Deutschland nur wenig berücksichtigt wurden, abgesehen von dem Buch von P. Starke [Sta90]. So sind viele der hier aufgeführten Resultate nur aus der Originalliteratur bekannt und in Lehrbüchern kaum vertreten. Dies gilt etwa für die Entscheidbarkeit des Erreichbarkeitsproblems, die im Speicherbedarf mindestens exponentielle Komplexität des Erreichbarkeitsproblems, die nur rein hypothetische Entscheidbarkeit des Gleichheitsproblems beschränkter Petri-Netze, die algebraische Charakterisierung von Pomset-Sprachen. Der Grund, dass diese Ergebnisse bisher kaum in Lehrbüchern zu finden sind, liegt nicht in deren Bedeutung (diese ist im Gegenteil sehr hoch), sondern in der Schwierigkeit der dazu notwendigen Beweise.

Petri-Netze

Priese, L.; Wimmel, H.

2008, IX, 374 S., Softcover

ISBN: 978-3-540-76970-5