14 Testen nebenläufiger Systeme

14.1 Eigenschaften nebenläufiger Systeme und auftretende Fehler

Unter **nebenläufigen Softwaresystemen** werden Systeme verstanden, bei denen die Semantik der Programmiersprache keinen sequentiellen Ablauf verlangt. Das bedeutet, daß eine parallele Ausführung verschiedener Anweisungen möglich — aber nicht zwingend erforderlich — ist. Damit wird der Begriff der Nebenläufigkeit (Parallelität ist möglich) vom Begriff der (strikten) Parallelität (Gleichzeitigkeit ist zwingend vorgeschrieben) unterschieden.

Nebenläufig ausführbare Programmteile, die jeweils für sich sequentiell abgearbeitet werden, nennt man **Prozesse** oder (engl.) Tasks. Auf der obersten Beschreibungsebene eines nebenläufigen Softwaresystems ist es meist angebracht, das System als eine Menge von Prozessen zu beschreiben, die mit einer *cobegin-/coend*-Anweisung[1] zusammengefaßt sind.

Beispiel 14.1.1
Ein System aus drei Prozessen P1, P2, P3 läßt sich folgendermaßen beschreiben, wobei || das Trennzeichen für die drei Prozesse ist:

cobegin *P1* || *P2* || *P3* **coend**

Ein System nebenläufiger Prozesse hat i. allg. ein nichtdeterministisches Verhalten und erzeugt evtl. sogar undefinierte Ergebniswerte, wie folgendes Beispiel zeigt.

Beispiel 14.1.2

cobegin *x:=y* || *y:=x* **coend**

1. *Wenn y:=x vor x:=y ausgeführt wird, haben beide Variablen zuletzt den Anfangswert x_0 von Variable x.*
2. *Wenn x:=y vor y:=x ausgeführt wird, haben beide Variablen zuletzt den Anfangswert y_0 von Variable y.*

[1] *cobegin* ist die Abkürzung für „concurrent begin“, *coend* für „concurrent end“.

3. *Wenn der Anfangswert von y bei Ausführung von x:=y genau zu dem Zeitpunkt x zugewiesen werden soll, zu dem der aktuelle Wert von x (bei der Ausführung der rechten Seite von y:=x) ermittelt wird; dann ist der aktuelle Wert von x und damit der endgültige Wert von y unbestimmt oder undefiniert*[2].

Der Nichtdeterminismus ist eine Folge der (durch die Semantik der entsprechenden Programmiersprache) nicht festgelegten Reihenfolge der Ausführung einzelner nebenläufiger Anweisungen. Um diesen oft unerwünschten Nichtdeterminismus zu verhindern, sind Prozesse zu synchronisieren, d. h. sie müssen auf gewisse Zustände oder Ereignisse warten, was zu Blockierungen führen kann. Die Synchronisation kann durch Nachrichtenaustausch (message passing) erreicht werden[3]. Dabei wird das Senden und Empfangen von Nachrichten zur Synchronisation benutzt: der Prozeß, der eine Nachricht empfangen soll, muß so lange warten, bis sie abgeschickt wurde und daraufhin eingetroffen ist.

Beispiel 14.1.3 (Essende Philosophen)
In der Programmiersprache Ada läßt sich das in Kapitel 1.3 spezifizierte Verhalten der Philosophen und Gabeln wie folgt (auszugsweise) beschreiben (vgl. [TLK 92], S. 210):

FORKS : **array** *(1..5)* **of** *FORK;*

```
task body FORK is
1:  begin
2:     loop
3:        accept UP;
4:        accept DOWN;
5:     end loop;
6:  end FORK;
```

Zusätzlich sind fünf Philosophentasks vom Typ PHILOSOPHER zu erzeugen, wobei N jeweils die Werte von 1 bis 5 annimmt.

Gemäß der Semantik der Sprache Ada werden die Anweisungen der einzelnen Prozesse bzw. Tasks sequentiell ausgeführt. Allerdings gibt es Synchronisationsanweisungen, die Wartebedingungen realisieren. In diesem Beispiel muß ein Philosoph bei dem Aufruf „FORKS(N).UP" darauf warten, daß die entprechende Gabel (FORK) mit Nummer N die Anweisung „accept UP" ausführen kann (und umgekehrt). Bei erfolgreicher Synchronisierung werden beide Anweisungen quasi gleichzeitig (als **Rendezvous***) ausgeführt. Stößt ein „accept UP" auf mehrere wartende Anweisungen*

[2] wenn der Speicherzugriffsmechanismus gleichzeitige Zugriffe auf eine Variable (bzw. Speicherzelle) erlaubt

[3] aber auch durch den gegenseitigen Auschluß sogenannter „kritischer Abschnitte" mit Hilfe von Semaphoren oder Monitoren. Genaueres dazu findet sich in der in Kapitel 14.7 angegebenen Literatur.

„FORKS(N).UP“ (von Philosoph N und N-1), dann wird ein mögliches Rendezvous nichtdeterministisch ausgewählt.

```
package body PHILOSOPHER is
    EAT, THINK : constant := 10.0;
    task P;
    task body P is
0:  begin
1:     loop
2:        FORKS(N).UP;
3:        FORKS(1+N mod 5).UP;
4:        delay EAT;
5:        FORKS(N).DOWN;
6:        FORKS(1+N mod 5).DOWN;
7:        delay THINK;
8:     end loop;
9:  end P;
end PHILOSOPHER;
```

Eine **selektive Kommunikation** ergibt sich, wenn der empfangende Prozeß in alternativen Zweigen nur unter bestimmten Bedingungen von verschiedenen Prozessen Nachrichten empfängt. Diese Bedingungen werden auch **Wächter** (guards) genannt.

Beispiel 14.1.4 (Speicher mit n Plätzen)
Ein Speicher mit n Speicherplätzen läßt sich als ewig laufender Prozeß beschreiben, der auf folgende Nachrichten reagiert: ein schreibender Prozeß schickt die Nachricht „ablegen“ (eines Wertes), ein lesender (konsumierender) Prozeß schickt die Nachricht „holen“ (eines Wertes). Die beiden Wächter sind in diesem Fall „fuellung $<$ n“ und „fuellung $>$ 0“. Es gibt drei Fälle:

1. *Bei leerem Speicher (Wächter „fuellung $>$ 0“ nicht erfüllt) wird die Nachricht „holen“ eines konsumierenden Prozesses nicht akzeptiert.*

2. *Bei vollem Speicher (Wächter „fuellung $<$ n“ nicht erfüllt) wird die Nachricht „ablegen“ eines schreibenden Prozesses nicht akzeptiert.*

3. *Wenn der Speicher weder voll noch leer ist, sind beide Wächter wahr. In diesem Fall kann eine vorliegende Nachricht „ablegen“ oder „holen“ sofort akzeptiert werden. Liegen beide Nachrichtenarten vor, erfolgt eine nichtdeterministische Auswahl. Liegt keine Nachricht vor, wartet der Speicherprozeß auf die nächste Nachricht.*

Nebenläufige Programmsysteme können normalerweise alle Konstrukte enthalten, die auch in sequentiellen Systemen erlaubt sind. Daher sind entsprechende

Berechnungs- und Bereichsfehler (s. Definition 7.2.1 auf Seite 195) möglich. Besonders interessant sind hier aber die Fehler, die nur in nebenläufigen Systemen auftreten können. Es handelt sich dabei um Fehler, die mit der blockierenden Synchronisation und dadurch erzwungenem Warten zwischen Prozessen bzw. Programmteilen zu tun haben. Im Prinzip können zwei Arten von Fehlern (in Bezug auf das Warten) vorkommen: einerseits fehlendes Warten, andererseits zusätzliches Warten.
Die Fehlerart des fehlenden Wartens kommt in Teil 3 von Beispiel 14.1.2 vor: Wenn der lesende Zugriff auf Variable x (in y:=x) nicht auf den schreibenden Zugriff auf diese Variable in x:=y wartet, kann es zu einem unbestimmten (undefinierten) Wert bei der Ausführung von y:=x kommen.

Die Fehlerart des zusätzlichen Wartens eines Prozesses kann folgendes bedeuten:

(a) ein Prozeß wartet unnötig lange auf Nachrichten bzw. Zustandswechsel anderer Prozesse, aber das Warten hat schließlich ein Ende;

(b) ein Prozeß wartet unendlich lange auf Nachrichten bzw. Zustandswechsel anderer Prozesse.

Im Fall (a) liegt ein ineffizientes Verhalten vor, was höchstens bei Echtzeitsystemen (genaueres siehe Abschnitt 14.5.1) zu falschem Verhalten führt. Im Fall (b) ist der fragliche Prozeß für alle Ewigkeit blockiert. Dies ist i. allg. ein unerwünschtes Verhalten. Dabei können wieder mehrere Fälle unterschieden werden:

i. Das unendlich lange Warten ist nicht zwingend durch die Konstruktion des Gesamtsystems bedingt, sondern nur durch eine unglückliche (zufällige) Verkettung von Umständen bzw. ein unfaires Verhalten der anderen Prozesse. In diesem Falle spricht man von **Verhungern** (starvation bzw. livelock).

ii. In keinem Fall nimmt das Gesamtsystem in der Zukunft einen Zustand an, in dem das Warten des Prozesses ein Ende hat. In diesem Fall spricht man von **Verklemmung** (deadlock) des Prozesses. Wenn alle Prozesse eines Systems im Wartezustand sind, dann ist das gesamte System für alle Ewigkeit blockiert und man spricht von **globaler Verklemmung**. Warten dagegen nur einige Prozesse jeweils im Kreis auf Zustandswechsel, die nur eintreten können, wenn andere wartende Prozesse aus dieser Prozeßmenge fortschreiten, so wird von **zirkulärer Verklemmung** eines Teilsystems (von Prozessen) gesprochen.

Bei fehlendem Warten gibt es Operationsfolgen aus zwei verschiedenen Prozessen, die sich unzulässigerweise überlappen. Beim unendlich langen Warten (beim Verhungern und bei Verklemmungen) sind vorher Operationsfolgen ausgeführt worden, die nicht ausgeführt werden sollten, da sie in diese ungewollten Systemzustände geführt haben. Bei unnötigem Warten liegt dagegen der gegenteilige Fall vor: einige Operationsfolgen können nicht ausgeführt werden, obwohl sie — laut Spezifikation —

ausführbar sein sollten. In jedem Fall lassen sich die vorgestellten Fehler (fehlendes oder zusätzliches Warten) also abstrakt als **Synchronisationsfehler** bezeichnen.

Das Konzept der Synchronisationsfehler wird im folgenden formal definiert, wobei folgende Annahmen gemacht werden:

1. (a) Die Prozesse des Programmsystems P werden auf einem Monoprozessor ausgeführt (daher läßt sich eine *Folge* von Operationen beobachten), oder
   (b) die (auf einem Multiprozessor) nebenläufig ausgeführten Operationen werden — bei der Beschreibung — als Folge (sequentiell) angeordnet.
2. Es werden nur Systeme von Prozessen betrachtet, die über Nachrichtenaustausch kommunizieren (vgl. S. 379, s. dort Fußnote 3).

Das nichtdeterministische Verhalten solcher Systeme hängt nur davon ab, welche Ereignisse in den Kommunikations- bzw. Synchronisationsanweisungen (z. B. *send/receive*) auftreten. Daher kann von allen sonstigen Operationen abstrahiert werden und eine Ausführung des Programmsystems P kann als eine Folge von Synchronisationsereignissen beschrieben werden. Eine solche Folge wird **Synchronisationssequenz** bzw. kurz **SYN-Sequenz** genannt. SYN-Sequenzen werden danach unterschieden, ob sie tatsächlich in der Implementation auftreten können oder ob sie — laut Spezifikation — auftreten dürfen bzw. sollen.

DEFINITION 14.1.1 (GÜLTIGE/AUSFÜHRBARE SYN-SEQUENZ)
Sei S die Spezifikation, I die Implementation und x die Eingabe[4] *eines nebenläufigen Systems P.*

1. *Eine SYN-Sequenz f heißt* **gültig** *für P bei der Eingabe von x g. d. w. f laut Spezifikation S bei Eingabe von x in P erzeugbar sein sollte. (Andernfalls heißt f* **ungültig***).* **G(P, x)** *sei die Menge aller gültigen SYN-Sequenzen für P bei Eingabe von x.*
2. *Eine SYN-Sequenz f heißt* **ausführbar** *für P bei der Eingabe von x g. d. w. f bei Eingabe von x in I erzeugt werden kann. (Andernfalls heißt f* **unausführbar***.)* **A(P, x)** *sei die Menge aller ausführbaren SYN-Sequenzen bei Eingabe von x in die Implementation I von P.*

Wegen des nichtdeterministischen Verhaltens nebenläufiger Programmsysteme kann es zu jeder Eingabe x mehrere gültige bzw. ausführbare SYN-Sequenzen geben. Im Normalfall sollten die ausführbaren mit den gültigen SYN-Sequenzen übereinstimmen, d. h. es sollte $G(P,x) = A(P,x)$ gelten; andernfalls liegt ein Fehler vor.

[4] Zur Eingabe kann auch die Angabe des Zustands (Inhalts) einer Datei oder Datenbank gehören, außerdem kann die Eingabe bei interaktiven Programmen aus einer Folge von Eingabewerten bestehen (s. Definition 3.5.2.3 auf Seite 61).

DEFINITION 14.1.2
Ein nebenläufiges Programmsystem P enthält einen **Synchronisationsfehler** *genau dann wenn eine Eingabe x von P mit $G(P,x) \neq A(P,x)$ existiert.*

Falls $G(P,x) \neq A(P,x)$ gilt, ist mindestens eine der folgenden Aussagen wahr:

1. Es gibt mindestens eine ausführbare SYN-Sequenz f für P bei Eingabe von x, die nicht gültig ist.
2. Es gibt mindestens eine gültige SYN-Sequenz f für P bei Eingabe von x, die nicht ausführbar ist.

Der Fall 1 ist ein Fehler, bei dem sehr wahrscheinlich eine falsche Ausgabe produziert wird. Im Fall 2 wird normalerweise keine falsche Ausgabe produziert; das implementierte System hat aber unzulässige Einschränkungen und sollte deshalb auch als fehlerhaft angesehen werden.
Eine SYN-Sequenz, die zu einer Verklemmung führt, sollte normalerweise laut Spezifikation ungültig sein. Wenn sie dennoch ausführbar ist, liegt mit der Verklemmung ein spezieller Synchronisationsfehler (von Fall 1) vor.
Das folgende Beispiel erläutert einen Synchronisationsfehler für Fall 2.

BEISPIEL 14.1.5 (ERZEUGER-VERBRAUCHER-PROBLEM)
Ein nebenläufiges System EV zur Lösung des Erzeuger-Verbraucher-Problems mit der Puffergröße 2 (analog zum Speicher von Beispiel 14.1.4 spezifiziert) läßt sich — wie in Abbildung 14.1 dargestellt — veranschaulichen.

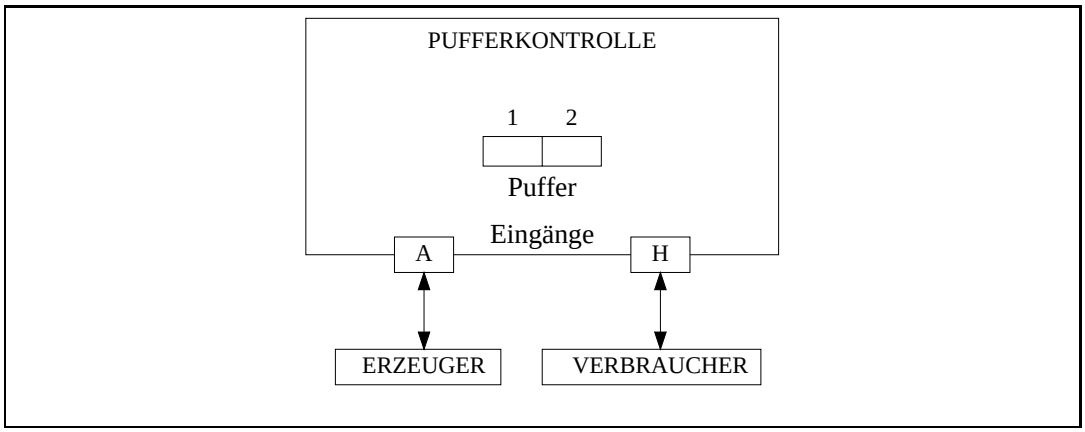


Abb. 14.1: Erzeuger-Verbraucher-System

Die Synchronisation der drei Prozesse PUFFERKONTROLLE, ERZEUGER und VERBRAUCHER erfolgt über die Operationen „Ablegen“ und „Holen“, die durch die Synchronisationsereignisse A und H dargestellt werden. (In Ada sind A und H z. B. jeweils ein entsprechendes Rendezvous.)

Wenn der Prozeß ERZEUGER dreimal „Ablegen“ aufruft und der Prozeß VERBRAUCHER dreimal „Holen“ aufruft, gibt es genau vier gültige SYN-Sequenzen (siehe Übung 14.2). Wenn die Implementation dagegen nur das Speichern eines Elements (anstatt von zwei Elementen) im Puffer erlaubt, dann ist nur die SYN-Sequenz AHAHAH ausführbar. Die vom Erzeuger abgelegten drei Elemente werden vom Verbraucher in der richtigen Reihenfolge geholt. Das Verhalten des Systems ist also korrekt. Dennoch hat das System mit Puffergröße 1 den Fehler, daß gültige SYN-Sequenzen, z. B. AHAAHH, im System nicht ausführbar sind. In diesen Fällen wird der ERZEUGER beim Ablegen unnötig blockiert (da die Teilsequenz AA nicht möglich ist), und der VERBRAUCHER muß beim Holen ebenfalls unnötig warten (da die Teilsequenz HH nicht ausführbar ist).

14.2 Grundsätzliche Modelle, Probleme und Lösungen

14.2.1 Beschreibung von nebenläufigen Programmsystemen durch Modelle

Um Test- und Analyseverfahren in allgemeiner Form vorstellen zu können, müssen Modelle vorliegen, die von den konkreten Ausprägungen der nebenläufigen Programme und Programmiersprachen abstrahieren und nur die benötigten, wesentlichen Eigenschaften der nebenläufigen Programme beschreiben. Bei sequentiellen Programmen sind dies z. B. Pfadausdrücke, endliche Automaten, Kontrollfluß- und Datenflußgraphen. Diese Modelle sind in angepaßter bzw. erweiterter Form ebenfalls für nebenläufige Systeme geeignet.

14.2.1.1 Pfadausdrücke

Zur Spezifikation der zulässigen nebenläufigen Abläufe von Prozessen bieten sich wieder Pfadausdrücke an, die gegenüber (sequentiellen) Pfadausdrücken um Konstrukte für nebenläufiges Ausführen erweitert sind.

DEFINITION 14.2.1

1. *Die Syntax der* **nebenläufigen Pfadausdrücke** *über einer Menge F ist folgendermaßen definiert:*

   (a) *Ein Pfadausdruck (nach Def. 5.1.1*[5]*) ist auch ein nebenläufiger Pfadausdruck.*

   (b) *Wenn P und Q nebenläufige Pfadausdrücke sind, so auch*

[5] Ein sequentieller Pfadausdruck nach Def. 5.1.1 ist ein leerer Pfadausdruck, ein Element aus F oder wird daraus mit Sequenz, Alternative, Option oder Wiederholung gebildet.

i. $P + Q$, *die* **Nebenläufigkeit** *von* P *und* Q,

ii. $\{P\}$, *die* **Simultanität** *von* P.

(c) *Nur ein durch (a) und (b) gebildeter Ausdruck ist ein nebenläufiger Pfadausdruck, im folgenden auch einfach* **Pfadausdruck** *genannt.*

2. *Die Bedeutung der Pfadausdrücke entspricht der Namenswahl für die verschiedenen Ausdrücke:*

(a) *Die Bedeutung von sequentiellen Ausdrücken ist in Teil 2 von Definition 5.1.1 erläutert.*

(b) i. *Bei der Nebenläufigkeit von P und Q dürfen die „Elementarereignisse aus F“, die in P und Q enthalten sind, beliebig überlappt (d. h. nacheinander oder gleichzeitig) vorkommen, allerdings muß die durch P bzw. Q vorgeschriebene Reihenfolge eingehalten werden.*

ii. *Bei der Simultanität von P dürfen die durch P spezifizierten Reihenfolgen beliebig oft nebenläufig zu sich selbst (d. h. gleichzeitig oder nacheinander) ausgeführt werden.*

14.2.1.2 Endliche Automaten und Flußgraphen

In vielen Fällen bietet sich — wie bei sequentiellen Systemen — eine Beschreibung des spezifizierten Verhaltens durch endliche Automaten (anstelle von Pfadausdrücken) an. Dabei beschränkt man sich meist auf die Beschreibung der gültigen SYN-Sequenzen, da eine Beschreibung der echten Nebenläufigkeit eine komplizierte Form von Automaten erfordert. Für das Erzeuger-Verbraucher-Problem (s. Beispiel 14.1.5) läßt sich z. B. ein endlicher Automat angeben, der die erlaubten Folgen der Operationen *Holen* und *Ablegen* beschreibt (s. Übung 14.3).

Zur Beschreibung von vorliegenden, implementierten, durch Nachrichtenaustausch kommunizierenden Systemen von parallelen Prozessen bieten sich — wie im sequentiellen Fall — Kontrollflußgraphen bzw. Datenflußgraphen für die einzelnen Prozesse an, wobei die Synchronisationen durch besondere Synchronisationskanten dargestellt werden.

Definition 14.2.2
Ein System von parallelen Prozessen $P_i, i = 1, \ldots, n$, wird durch eine Menge von synchronisierten Flußgraphen $F_i, i = 1, \ldots, n$, mit zusätzlichen Synchronisationskanten S dargestellt.

1. *Dabei besteht jeder* **synchronisierte Flußgraph** $F_i = (N_i, K_i, a_i, E_i)$ *aus:*

   - *einer Menge von Knoten N_i, wobei jeder Knoten eine Anweisung von P_i repräsentiert (die als nächstes ausgeführt werden kann)*[6]*;*
   - *einer Menge von gerichteten Kanten K_i, wobei jede Kante k von Knoten n nach Knoten m den möglichen Kontrollfluß zwischen den Anweisungen in P_i beschreibt, die zu den Knoten n und m gehören;*
   - *einem Anfangsknoten a_i (ein Knoten, der die erste von P_i auszuführende Anweisung beschreibt);*
   - *einer Menge E_i von Endknoten (Knoten, die eine letzte von P_i auszuführende Anweisung beschreiben), wobei E_i auch leer sein kann, wenn Prozeß P_i nie terminiert.*

2. *Die* **Synchronisationskanten** *von S verbinden Knoten aus verschiedenen Flußgraphen F_i und F_j, $i \neq j$. Diese Kanten sind — je nach Art der Synchronisation — gerichtet oder ungerichtet.*

   - *Eine* **gerichtete** *Synchronisationskante von Knoten m nach Knoten n bedeutet, daß die entsprechenden Anweisungen so zu synchronisieren sind, daß erst nach Ausführung der zu m gehörenden Anweisung die zu n gehörende Anweisung ausgeführt werden darf.*
   - *Eine* **ungerichtete** *Synchronisationskante zwischen Knoten m und Knoten n bedeutet, daß ein Rendezvous der zu m und n gehörenden Anweisungen erfolgen soll, d. h. die Anweisungen sollen „gleichzeitig“*[7] *stattfinden.*

3. *Bei* **reduzierten synchronisierten Flußgraphen** *F_i^r werden nur die auszuführenden Synchronisationsanweisungen*[8] *oder Strukturierungsanweisungen*[9] *durch Knoten dargestellt.*
   Eine Kante führt von einem Knoten n zu einem Knoten m in F_i^r, wenn in dem (nicht reduzierten) Flußgraphen F_i ein Weg von n nach m existiert, der (außer n und m) keine Knoten des reduzierten Flußgraphen F_i^r enthält. Die Synchronisationskanten S sind dieselben wie bei den nicht reduzierten Flußgraphen $F_i, i = 1, \ldots, n$.

[6] Faßt man die Knoten (wie bei einem Automaten) als Zustände auf, sind die Anweisungen die „Transitionen“, die eigentlich den Kanten zugeordnet werden müssen. Die hier vorgestellte Darstellung entspricht aber der üblichen Flußdiagrammnotation und erlaubt die Zuordnung der Synchronisationskanten (s. Teil 2) zu den Knoten.

[7] Bei Ausführung auf einem Monoprozessor bedeutet „gleichzeitig“, daß die in F_i bzw. F_j jeweils folgende Anweisung erst ausgeführt werden darf, wenn beide zum Rendezvous gehörenden Anweisungen ausgeführt wurden.

[8] z. B. *accept, select, delay, task begin/end*

[9] *subprogram call/begin/end/return* oder *block begin/end*, wenn das Unterprogramm oder der Block Synchronisationsanweisungen enthalten.

Bei reduzierten synchronisierten Flußgraphen konzentriert man sich bei der Beschreibung also auf die Synchronisationsanweisungen und spart damit Knoten (und somit Beschreibungsaufwand) ein.

Beispiel 14.2.1 (Essende Philosophen)
Das Verhalten von zwei essenden Philosophen (2-Philosophen-Problem) läßt sich durch die reduzierten synchronisierten Flußgraphen aus Abbildung 14.2 beschreiben. Dabei stellt Zustand P_{ij} die Anweisung Nummer j (aus Beispiel 14.1.3) von Philosoph i dar. Entsprechend bezeichnet F_{ij} die j-te Anweisung von FORK i.

Wie in Abbildung 14.2 zu sehen, ist die Darstellung von nebenläufigen Systemen als reduzierte synchronisierte Flußgraphen zwar ziemlich kompakt, es fehlt aber eine schnelle Erfassung der möglichen Zustände und Übergänge des gesamten Systems auf einen Blick.

Diese Gesamtschau erhält man, wenn die Knoten der einzelnen (reduzierten) Flußgraphen F_i als Teilzustände des gesamten Systems aufgefaßt werden. Die Zustände sind also n-Tupel, deren Komponenten die Zustände (Knoten) aus den n Flußgraphen sind (evtl. ergänzt um den Zustand „inaktiv“). Von den Variablenwerten der einzelnen Prozesse wird dabei vollständig abstrahiert, was wiederum Beschreibungsaufwand einspart (allerdings auch Beschreibungsgenauigkeit einbüßt). In dem Gesamtsystem gibt es eine **Transition** (einen Zustandsübergang) von einem Zustand $z = (z_1, z_2, \ldots, z_n)$ zu einem Zustand $z' = (z'_1, z'_2, \ldots, z'_n)$, wenn für alle $i = 1, \ldots, n$ jeweils eine der folgenden vier Bedingungen gilt:

1. (z_i, z'_i) ist ein Übergang in F_i,
2. $z_i =$ inaktiv und $z'_i = a_i =$ Anfangszustand von F_i,
3. z_i ist Endzustand von F_i und $z'_i =$ inaktiv,
4. $z'_i = z_i$ (keine Änderung in F_i).

Da im Fall 4 nichts beim i-ten Prozeß passiert, muß für mindestens ein i ($1 \leq i \leq n$) eine der Bedingungen 1, 2, oder 3 zutreffen, bei denen in F_i ein Übergang stattfindet. Im Fall 1 kann dabei der Transition t noch als **Beschriftung** die Anweisung zugeordnet werden, die im Knoten z_i von F_i als nächstes ausgeführt werden kann (beim Übergang nach z'_i).
In den Fällen 1, 2 und 3 sind die Übergänge nur erlaubt, wenn die zugehörigen Synchronisationskanten (siehe Teil 2 von Def. 14.2.2) dies zulassen. Bei einem Rendezvous zwischen Prozeß P_i und P_j muß also z. B. gleichzeitig Fall 1 für Flußgraph F_i und Flußgraph F_j vorliegen. Für ein nebenläufiges System von Prozessen P_i mit zugehörigen (reduzierten) Flußgraphen F_i ($i = 1, \ldots, n$) wird der nach obigem Konzept konstruierte endliche Automat **(reduzierter) Nebenläufigkeitsautomat** (von $\{P_i\}$ bzw. $\{F_i\}$) genannt.

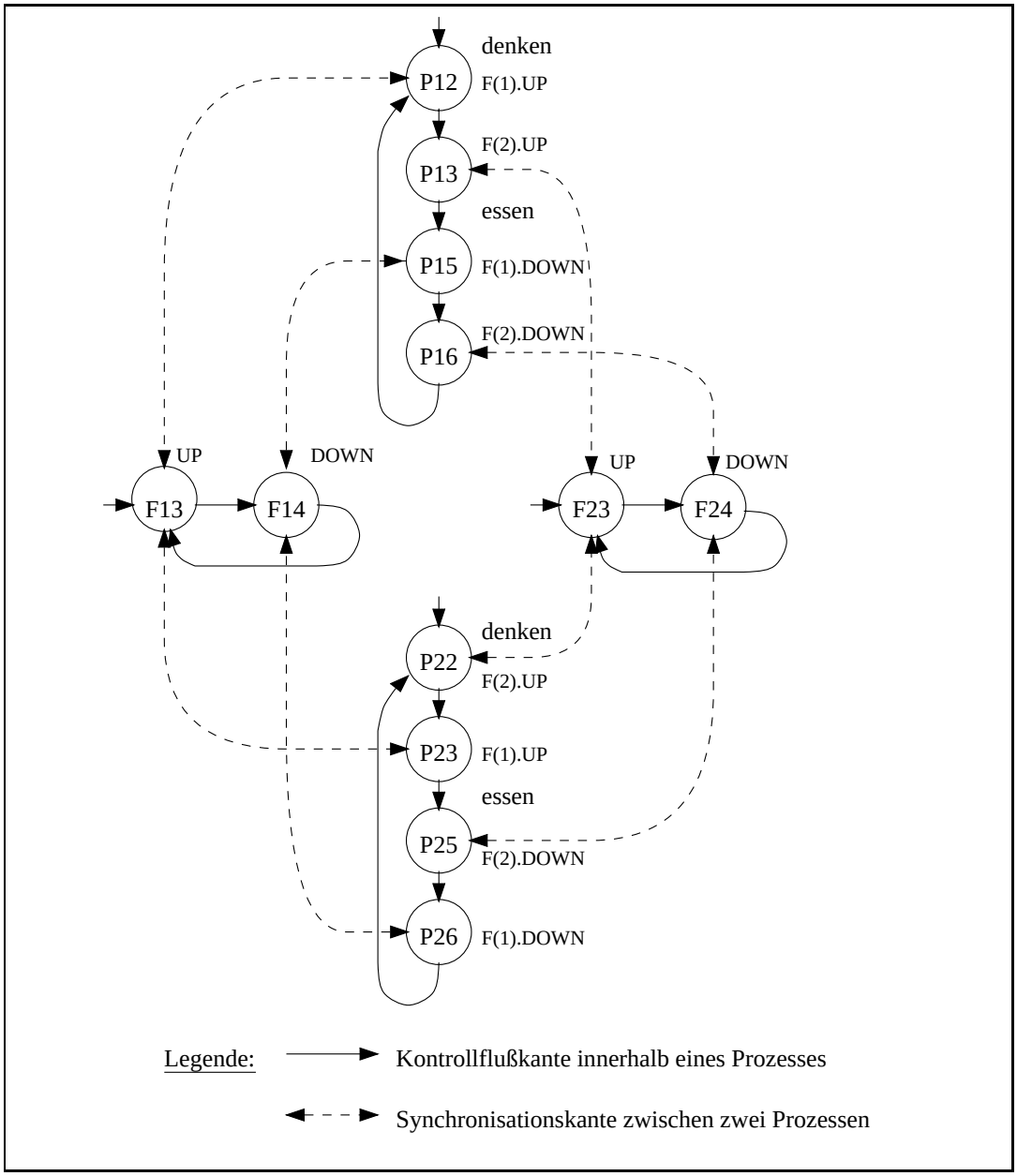


Abb. 14.2: reduzierte synchronisierte Flußgraphen für das 2-Philosophen-Problem

Beispiel 14.2.2 (Essende Philosophen)
Die reduzierten synchronisierten Flußgraphen aus Abbildung 14.2 ergeben den reduzierten Nebenläufigkeitsautomaten aus Abbildung 14.3, der das Verhalten von zwei essenden Philosophen beschreibt.

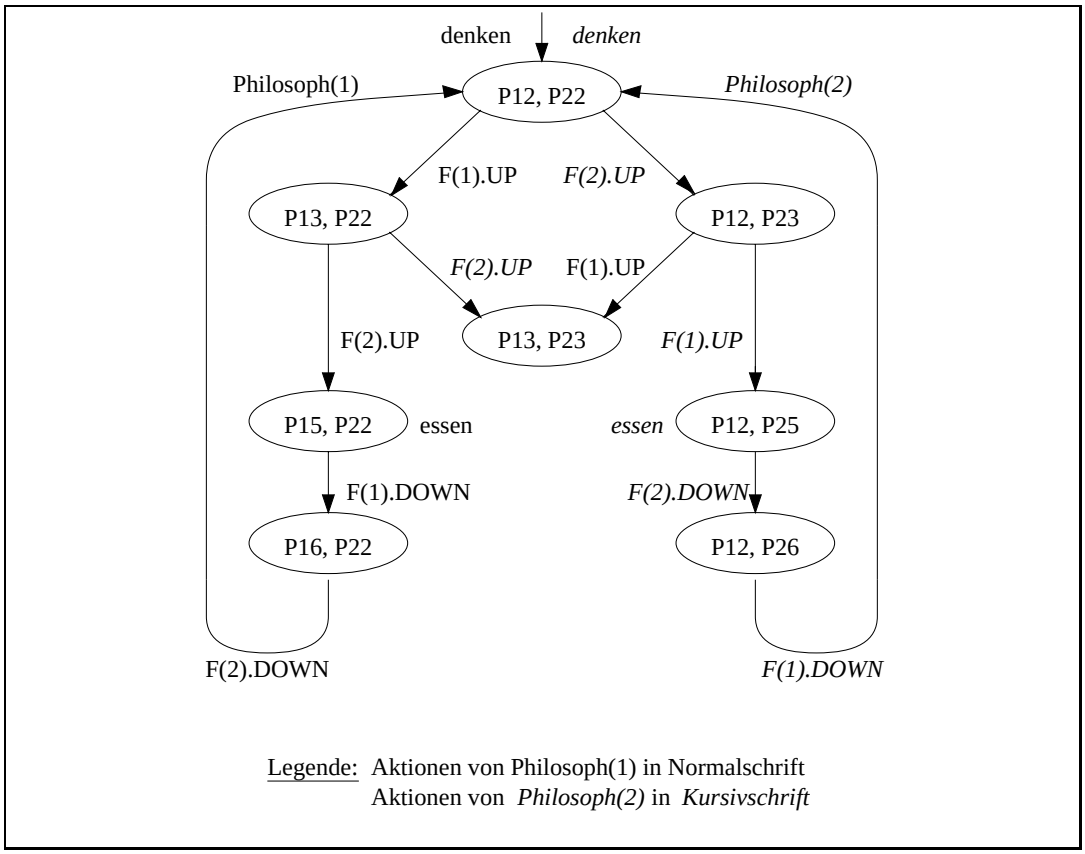


Abb. 14.3: reduzierter Nebenläufigkeitsautomat des 2-Philosophen-Problems

Bei der Modellierung mit Nebenläufigkeitsautomaten wird der Aufwand für die Analyse dadurch reduziert, daß alle Zustände mit gleichem Kontrollzustand aber unterschiedlichen Variablenwerten zu einem Zustand zusammengefaßt werden. Für eine genauere Datenflußanalyse und für eine genauere Analyse des sich verzweigenden Kontrollflusses zwischen Synchronisationsanweisungen eines Prozesses müssen die Variablenwerte aber berücksichtigt werden. Um den Analyseaufwand zu begrenzen, müssen einige Wege durch den Zustandsraum oder nur ein Teil der Zustände ausgewählt werden. Die Analyse kann daher natürlich nur gültige Aussagen über diesen Teil des Automaten (bzw. des repräsentierten Systems) machen. Dennoch kann ein solches Vorgehen nützlich sein, um sich z. B. zu vergewissern, daß kritische Systemteile oder -abläufe korrekt funktionieren.

Eine andere Möglichkeit zur Begrenzung des Analyseaufwands besteht darin, die möglichen Variablenwerte in endlichen Klassen zusammenzufassen (genaueres s. [Ber 95]).

Der Aufwand für die Konstruktion des Nebenläufigkeitsautomaten kann folgendermaßen abgeschätzt werden: wenn der i-te reduzierte synchronisierte Flußgraph F_i^r eine Anzahl von m_i Knoten bzw. Zuständen hat $(i = 1, \ldots, n)$, dann kann der Nebenläufigkeitsautomat im schlimmsten Fall $\prod_{i=1}^{n} m_i$ Zustände haben. (Das ist der Fall, in dem jeder Zustand eines Flußgraphen in jeder Kombination mit den Zuständen der anderen Flußgraphen vorkommt.) Nimmt man an, daß alle Flußgraphen etwa gleich viele Zustände — nämlich m — haben, ist die Zustandsanzahl des Nebenläufigkeitsautomaten und damit der Berechnungsaufwand (Zeitaufwand) für den Algorithmus im schlimmsten Fall von der Ordnung $O(m^n)$; der Aufwand steigt also exponentiell mit der Anzahl n der Flußgraphen (bzw. der Anzahl der Prozesse). Nur wenn der Kontrollfluß innerhalb der Flußgraphen stark eingeschränkt ist, kann man mit einem (praktisch akzeptablen) linear wachsenden Berechnungsaufwand für den Algorithmus rechnen.
Der Berechnungsaufwand kann natürlich auch dadurch reduziert werden, daß man die Anzahl n der *gleichzeitig* zu betrachtenden Flußgraphen (bzw. Prozesse) klein hält. Das ist dann möglich, wenn Prozesse praktisch nicht miteinander kommunizieren.

Aus den (reduzierten) Nebenläufigkeitsautomaten lassen sich leicht die Folgen von (Synchronisations-)Operationen bzw. -Ereignissen ableiten, die in dem beschriebenen System von parallelen Prozessen möglich sind, d. h. die ausführbaren SYN-Sequenzen im Sinne von Definition 14.1.1.

DEFINITION 14.2.3
Sei A ein (reduzierter) Nebenläufigkeitsautomat.

1. *Eine endliche Folge $z = (z_0, \ldots, z_n)$ ist eine* **Zustandssequenz** *(von A) g. d. w. für alle $i = 1, \ldots, n$ der Übergang von z_{i-1} nach z_i ein erlaubter Zustandsübergang (eine Transition) in A ist.*

2. *Eine endliche Folge $t = (t_1, \ldots, t_n)$ ist eine* **Transitionssequenz** *(von A) g. d. w. es eine Zustandssequenz $z = (z_0, \ldots, z_n)$ von A gibt mit $t_i = (z_{i-1}, z_i)$ für alle $i = 1, \ldots, n$.*

3. *Eine endliche Folge $s = (s_1, \ldots, s_n)$ ist eine* **Synchronisationssequenz** *(von A) g. d. w. es eine Transitionssequenz $t = (t_1, \ldots, t_n)$ von A gibt, so daß für alle $i = 1, \ldots, n$ gilt: s_i ist die zu t_i gehörende Beschriftung mit einem Synchronisationsereignis.*

Beispiel 14.2.3 (Essende Philosophen)
Für den Nebenläufigkeitsautomaten des Philosophenproblems aus Abbildung 14.3 ergeben sich z. B. die in Tabelle 14.1 (hier vertikal angeordneten) Sequenzen. P(i).F(j).UP bezeichnet dabei das Synchronisationsereignis, daß Philosoph i die Gabel (Fork) j aufnimmt (analog für DOWN).

Zustandssequenz	**Synchronisationssequenz**
(P12, P22)	P(1).F(1).UP
(P13, P22)	P(1).F(2).UP
(P15, P22)	P(1).F(1).DOWN
(P16, P22)	P(1).F(2).DOWN
(P12, P22)	P(1).F(1).UP
(P13, P22)	P(2).F(2).UP
(P13, P23) (Endzustand)	(Verklemmung)

Tab. 14.1 Zustands- und Synchronisationssequenz des 2-Philosophen-Problems

Mit Hilfe von SYN-Sequenzen, nebenläufigen Pfadausdrücken, endlichen Automaten, Nebenläufigkeitsautomaten und Zustands-, Transitions- und Synchronisationssequenzen kann nun das spezifizierte oder realisierte Verhalten von parallelen Programmen beschrieben werden. Damit liegt eine Basis für die statische Analyse und die dynamische Analyse (das Testen) dieser Systeme vor.

14.2.2 Probleme und Lösungen beim Testen nebenläufiger Programme

Für das Testen von nebenläufigen Programmen hat das nichtdeterministische Verhalten eine gravierende Konsequenz: ein Test mit einem Testdatum (vgl. Definition 3.5.2.2 auf S. 61) ist nicht generell reproduzierbar, da bei der Ausführung verschiedener Wege i. allg. auch verschiedene Ergebnisse erzeugt werden. Dieses Problem läßt sich nur dadurch lösen, daß zunächst die Begriffe des Tests und des Testdatums angemessener gefaßt werden und in einem zweiten Schritt Konzepte vorgestellt werden, wie Tests (im neuen Sinne) ausgeführt und reproduziert werden können.

Definition 14.2.4
Sei P ein System kommunizierender Prozesse.
Ein **(reproduzierbares) Testdatum** *t für P ist ein Tripel (x,s,y), wobei x ein Eingabedatum für P, s eine SYN-Sequenz von P bei Eingabe von x, y ein*[10] *zulässiges Solldatum bei Eingabe von x ist*[11].

[10] Wie in Definition 3.5.2.4 auf S. 61 sollen hier mehrere gewünschte Ergebnisse zugelassen werden.
[11] vgl. Seite 382 zum Begriff SYN-Sequenz und die Definitionen 3.5.2.3 und 3.5.2.4 auf S. 61 von Eingabedatum und Solldatum

Beim Testen mit reproduzierbaren Testdaten der Form (x, s, y) ist zu beachten, daß bei der Eingabe von x in die Implementation von P nur eine *ausführbare* SYN-Sequenz s erzeugt werden kann, die gültig oder ungültig sein kann (vgl. Definition 14.1.1). Im zweiten Fall („ungültig“) wird damit ein Synchronisationsfehler aufgedeckt (siehe Definition 14.1.2). In beiden Fällen kann das tatsächlich erzeugte Ergebnis y' mit einem Solldatum y übereinstimmen oder davon unzulässig abweichen (Fehlerfall). Wenn s eine gültige, aber nicht ausführbare SYN-Sequenz für P bei Eingabe von x ist, gibt es zwei Möglichkeiten bei dem Versuch, den Test mit (x, s, y) auszuführen:

1. der Testvorgang führt zu einer Verklemmung,
2. der Testvorgang erzeugt eine ausführbare, verklemmungsfreie SYN-Sequenz s', die verschieden von s ist.

Bei einem unveränderten System P wird beim (dynamischen) Testen meist der Fall 2 auftreten. Wenn aber das System P oder seine Umgebung verändert wird, um eine reproduzierbare Ausführung von (x, s, y) zu erzwingen, wird Fall 1 auftreten.

Die reproduzierbare Ausführung von Tests (auf einem Monoprozessor) kann durch Scheduler-, Uhr- oder Aufrufkontrolle erzwungen werden.
Beim Ansatz der **Schedulerkontrolle** wird ausgenutzt, daß der Scheduler des Betriebssystems eines Monoprozessors wesentlichen Einfluß darauf hat, welcher Prozeß als nächster für die Verarbeitung ausgewählt wird. Durch gezielte Veränderung des Schedulers kann man also eine bestimmte Reihenfolge der Prozeßaktivierungen und Prozeßsuspendierungen und damit die gewünschte SYN-Sequenz s erzeugen.
Beim Ansatz der **Uhrkontrolle** wird der zeitliche Ablauf nebenläufiger Prozesse kontrolliert. Dazu muß für alle Prozesse eine virtuelle Zeit simuliert werden, etwa von einem entsprechenden „Uhr“-Monitor. Prozesse können sich damit für eine bestimmte Anzahl von (virtuellen) Zeiteinheiten — sogenannten **Ticks** — verzögern. Jeder Tick kann dabei andere verzögerte Prozesse wieder in Gang setzen. Definiert man für jeden Prozeß an den richtigen Stellen im Programmablauf Verzögerungsanweisungen mit passenden Verzögerungszeiten (Anzahl der Ticks), kann gerade das Gewünschte erreicht werden: Im Zeitintervall i (nach dem i-ten Tick) ist genau der Prozeß aktiv, der in der geforderten SYN-Sequenz an i-ter Stelle steht.
Der Ansatz der **Aufrufkontrolle** setzt voraus, daß die kommunizierenden Prozesse ihre Synchronisation über einen Monitor oder ähnliche Konstrukte (z. B. sogenannte *entry calls* in der Sprache Ada) abwickeln. Der Monitor nimmt die Prozeduraufrufe aller Prozesse entgegen, verzögert sie aber so, daß die gewünschte Ausführungsreihenfolge der Prozedurausführungen erzeugt wird.

Die drei vorgestellten Verfahren sind unterschiedlich aufwendig, automatisierbar und einsetzbar.

Die Schedulerkontrolle hat diverse Nachteile. Sie ist

- sehr aufwendig,
- nur wenig automatisierbar (da für jede SYN-Sequenz die Eingriffe in den Scheduler neu geplant werden müssen),
- nur mit detaillierten Betriebssystemkenntnissen implementierbar,
- nur bei Ausführung auf Monoprozessoren einsetzbar.

Von Vorteil ist, daß das reale Zeitverhalten des Systems wenig verfälscht wird.

Die Uhrkontrolle hat folgende Nachteile:

- Die Spezifikation der Verzögerungsanweisungen ist kompliziert und damit fehleranfällig, da die genaue Länge der notwendigen Verzögerungen nur mit Kenntnissen über den Scheduler bestimmt werden kann. Der Test selbst kann also zusätzliche Fehler erzeugen.
- Für jede SYN-Sequenz müssen neue Verzögerungsanweisungen definiert werden.
- Durch das Konzept der virtuellen Zeit wird das reale Zeitverhalten des Systems verfälscht.

Die Aufrufkontrolle hat ebenfalls den bei der Uhrkontrolle zuletzt genannten Nachteil:

- Durch den zusätzlichen Kontrollprozeß (Monitor) wird das reale Zeitverhalten des Systems verfälscht.

Von Vorteil ist die folgende Eigenschaft:

\+ Das Verfahren ist gut zu automatisieren: Das Programmsystem P muß nur in ein System P' transformiert werden, bei dem alle Aufrufe von Prozeduren (oder „entry calls“) zu einem Kontrollprozeß (Monitor) umgeleitet werden. Dieser Kontrollprozeß vergleicht die Aufrufe mit dem nächsten auszuführenden Ereignis der vorgegebenen SYN-Sequenz und stellt einen Aufruf zurück, wenn er noch nicht an der Reihe ist.

Die Schedulerkontrolle ist also nur für den (sehr) aufwendigen Test von Monoprozessorsystemen mit harten Realzeitbedingungen — ein unwahrscheinlicher Fall — zu empfehlen. Die Uhrkontrolle ist nur beim Testen von kleinen nebenläufigen Systemen oder für die Simulation von nebenläufigen Programmen sinnvoll einsetzbar, bei denen es auf die realen Zeitbedingungen nicht ankommt. Die Aufrufkontrolle ist wegen der guten Automatisierbarkeit bei großen Programmsystemen zu empfehlen.

Sie ist sogar bei verteilten Systemen einsetzbar, wenn die Realzeitbedingungen nicht hart sind (genaueres s. Kapitel 14.5).
Das Problem des unvorhersehbaren Programmverhaltens ist mit den drei vorgestellten Ansätzen im Prinzip gelöst. In der Praxis ist die Implementierung des Monitors oder die Veränderung des Schedulers sowie die Spezifikation der zu durchlaufenden SYN-Sequenzen aber eine zeitintensive, mühsame Aufgabe für das Testpersonal. Daher werden oft nur bestimmte, besonders fehleranfällige Situationen hergestellt. Ein Beispiel dafür sind sogenannte **Streßtests**, bei denen die Reaktion des Systems auf eine sehr große Last — eine Überlast — getestet wird. Dabei wird gleichzeitig eine Vielzahl von Anforderungen an das System herangetragen; beispielsweise werden viele schreibende Prozesse erzeugt, die gleichzeitig die Nachricht *ablegen* an den Speicher aus Beispiel 14.1.4 auf S. 380 schicken.
Ein Problem ist mit den drei vorgestellten Ansätzen zum reproduzierbaren Testen neu erzeugt worden: Das Zeitverhalten des Systems wird mehr oder minder stark verfälscht. Wenn die Einhaltung von Zeitbedingungen oder zeitlichen Abhängigkeiten getestet werden soll, ist zu prüfen, ob es toleriert werden kann, daß die Zeitbedingungen durch die vorgestellten Ansätze verändert werden. Dazu sind bei der Testplanung entsprechende **Toleranzwerte** zu definieren. Falls die Toleranzwerte beim Test mit den drei vorgestellten Ansätzen nicht eingehalten werden können, ist zusätzlich Hardware einzusetzen, die ohne Zeitverfälschung die entsprechende Testablaufsteuerung übernimmt.

14.2.3 Probleme und Annahmen bei der statischen Analyse und formalen Verifikation

Bei einer statischen Analyse von nebenläufigen Programmsystemen entfallen die Probleme des nichtreproduzierbaren Testens. Im Prinzip können alle möglichen Programmpfade (SYN-Sequenzen) unter allen möglichen Bedingungen überprüft werden. Die Ergebnisse gelten unabhängig von einer bestimmten Schedulerstrategie und von evtl. wechselnden Echtzeitbedingungen.

Dennoch ist mit dieser Vorgehensweise nicht der Stein der Weisen gefunden worden, da andere Einschränkungen und Probleme vorliegen:

1. Statische Analyse beschäftigt sich i. allg. nicht mit der funktionalen Korrektheit, sondern nur mit strukturellen Eigenschaften des Programmsystems.

2. Statische Analyse und formale Verifikation machen diverse Annahmen; z. B.:

   (a) Die Anzahl der erzeugten Prozesse darf eine vorgegebene Schranke nicht überschreiten (s. Abschnitt 14.3.1, Restriktion 3).

   (b) Alle Pfade durch das Programm sind ausführbar[12].

[12] Die Bestimmung nicht ausführbarer Wege ist i. allg. zu kompliziert und generell nicht entscheidbar (siehe Satz 11.2.2 auf S. 285).

3. Die Behandlung von dynamischen Strukturen und Arrays mit variablem Index bereitet Probleme, da die einzelnen Komponenten nicht statisch identifiziert werden können. Daher müssen solche Strukturen entweder ausgeschlossen werden oder (inadäquat) als eine Einheit behandelt werden (vgl. Abschnitt 12.2.2, S. 317).

4. Die Semantik der Programmiersprache muß eindeutig definiert sein. Das ist gerade bei den Aspekten Nebenläufigkeit, Nichtdeterminismus und Zeitverhalten schwierig und nicht bei allen Sprachen vollständig gelöst (vgl. Kapitel 12.3 und 12.4 für sequentielle Programme).

5. Trotz aller Einschränkungen benötigen die statischen Analyseverfahren für Systeme von kommunizierenden Prozessen einen mit der Problemgröße exponentiell steigenden Analyseaufwand (s. Abschnitt 14.2.1, S. 390).

14.3 Statische Analyse nebenläufiger Programme

Bei der statischen Analyse geht es darum, Fehler im Programm aufzudecken, ohne das Programm mit konkreten Daten auszuführen (vgl. Kapitel 12 für sequentielle Programme). Dazu muß das nebenläufige Programm in einer gut analysierbaren Form vorliegen. Es wird also ein Modell des nebenläufigen Programmsystems benötigt, welches das Systemverhalten möglichst vollständig und korrekt wiedergibt, aber einfacher analysierbar ist als der Programmtext. Da gezeigt werden kann, daß eine exakte Modellierung die Analyse zu komplex macht (sie ist dann NP-hart oder sogar unentscheidbar), müssen Abstriche gemacht werden. Dabei ist es sinnvoll, daß das Modell alle fehlerhaften Abläufe des Programmsystems beschreibt und möglicherweise auch fehlerhafte Abläufe, die im tatsächlichen Programmsystem nicht ausführbar sind. Damit ist man auf der sicheren (übervorsichtigen) Seite.
Bei der Modellierung des Programmsystems hat man — wie im sequentiellen Fall — die Alternative, das Verhalten durch endliche Automaten mit vielen Zuständen zu beschreiben oder durch ein Modell mit nur einem Zustand — aber sehr vielen Variablen und komplexen Bedingungen für die Transformation der Variablenwerte bei einem Schritt des Berechnungsablaufs. Dieser Ansatz entspricht dem axiomatischen Ansatz bei der Programmverifikation.
Der Ansatz mit den endichen Automaten ist gut geeignet, Kontrollfluß- und Synchronisationsprobleme zu beschreiben und entsprechende Fehler (Verklemmung, Terminierungsprobleme) aufzuzeigen. Der axiomatische Ansatz ist besser für den Nachweis der partiellen Korrektheit geeignet, d. h. dafür, daß das System bei korrekter Terminierung (oder korrektem ewigen Ablauf) die richtigen Variablenwerte berechnet.

14.3.1 Statische Analyse auf der Basis von endlichen Automaten

Als Vorbereitung für die statische Analyse ist die Konstruktion des (reduzierten oder nicht reduzierten) Nebenläufigkeitsautomaten aus dem vorliegenden nebenläufigen (Ada-)Programm erforderlich (s. Kapitel 14.2). Da der Nebenläufigkeitsautomat einen kompletten Überblick über die für die Synchronisierung relevanten Zustände des Gesamtsystems liefert, können mit seiner Hilfe die folgenden Fragen beantwortet werden:

1. Kann das System in eine globale Verklemmung geraten?
2. Kann ein Prozeß des Systems unendlich lange blockiert[13] werden, d. h. muß er unendlich lange auf ein Ereignis warten, das seinen Fortschritt erlaubt?
3. Kann ein bestimmtes Rendezvous zwischen zwei Prozessen auftreten bzw. wird es stets auftreten?
4. Welche Aktionen von Prozessen können parallel ausgeführt werden bzw. werden stets parallel ausgeführt? Gibt es dabei eine **gefährliche Parallelität**, d. h. das parallele Definieren und Referenzieren oder das parallele doppelte Definieren einer Variablen in zwei verschiedenen Prozessen?

Die Fragen 1 und 3 können direkt durch Betrachtung aller Zustände des (reduzierten) Nebenläufigkeitsautomaten A beantwortet werden: Das System kann in eine globale Verklemmung geraten, wenn A einen Endzustand (Zustand ohne Nachfolger) enthält, in dem nicht alle Prozesse ihren Endzustand (bzw. den Zustand „inaktiv“) erreicht haben (vgl. Abbildung 14.3, Zustand $[P13, P23]$). Ein bestimmtes Rendezvous kann auftreten, wenn es einen Zustandsübergang in A gibt, der dieses Rendezvous beschreibt. Die Frage, ob ein bestimmtes Rendezvous stets auftreten wird, erfordert allerdings eine Nachbehandlung dieser Informationen mit graphentheoretischen Algorithmen (Propagierung der entsprechenden Informationen zum Anfangszustand von A, genaueres siehe [Tay 83], S. 375).
Frage 2 ist dann zu bejahen, wenn es einen Zyklus im Nebenläufigkeitsautomaten A gibt, d. h. wenn es eine Folge $t_1, \ldots, t_n$ von Transitionen (Zustandsübergängen) $t_i = (z_{i-1}, z_i)$ gibt $(i = 1, \ldots, n)$, die im selben Zustand $z_0 = z_n$ beginnt und endet, wobei für einen Prozeß P_j folgendes gilt: die j-te Zustandskomponente ist in allen Zuständen z_i $(i = 0, \ldots, n)$ identisch und beschreibt einen Zustand von Prozeß P_j, bei dem P_j auf das Eintreten eines Synchronisationsereignisses von einem anderen Prozeß wartet. Dies ist gerade der Fall, wenn P_j nicht im Endzustand (und nicht „inaktiv“) ist (vgl. Abbildung 14.3, „linker“ Zyklus mit identischer Zustandskomponente $P22$).
Die Frage 4 (nach parallelen Aktionen) kann ebenfalls durch genaue Betrachtung der Zustände des Nebenläufigkeitsautomaten A beantwortet werden. Für einen Zustand

[13] dies umfaßt Verklemmung und Verhungern

$z = (z_1, \ldots, z_r)$ von A, der ein System von r Prozessen beschreibt, sind jeder Zustandskomponente z_j die Anweisungen des Prozesses P_j zugeordnet, die als nächstes (in z_j) ausführbar sind. Für die Prozesse P_j und P_k sind also die den Zustandskomponenten z_j und z_k zugeordneten Anweisungen parallel ausführbar, wenn im Automat A das *nebenläufige* Verlassen der Zustandskomponenten z_j und z_k möglich ist. Das ist meistens der Fall, nur bei einem Rendezvous muß der Aufruf (*entry-call*) besonders behandelt werden (genaueres siehe [Tay 83], S. 374). Mit diesen Modifikationen können die möglichen parallelen Aktionen korrekt bestimmt werden. Die *stets* ausgeführten parallelen Aktionen erhält man wieder durch einen graphentheoretischen Algorithmus wie bei der Beantwortung von Frage 3 (Propagierung der entsprechenden Informationen zum Anfangszustand von A).

Die Analyseergebnisse sind allerdings nur unter gewissen Restriktionen für die beteiligten Prozesse und ihr Zusammenspiel gültig:

1. Die Prozesse müssen leicht identifiziert werden können. Eine Identifikation durch indizierte Ausdrücke mit variablem Index (z. B. Prozeß $P(i)$ mit Variable i) oder durch eine Kette von Zeigern auf entsprechende Objekte läßt sich i. allg. nicht statisch auflösen, d. h. es ist nicht statisch feststellbar, welcher Prozeß (welches Objekt) gemeint ist.

2. Echtzeitoperationen (z. B. Verzögerungsanweisungen) können nicht statisch ausgewertet werden, es sei denn, man kennt das exakte Verhalten von Hardware, Verteilalgorithmus (Scheduler) und Systemumgebung. Da dies meist nicht der Fall ist (weil die Zeiten unvorhersehbar schwanken können) kann nur eine *defensive* Analyse erfolgen: Die Analyseergebnisse gelten unter beliebigen Annahmen über das Zeitverhalten der Komponenten, d. h. es werden Fehler oder mögliche Fehler gemeldet, die unter den konkreten Bedingungen evtl. nicht auftreten können.

3. Bei der Modellierung wird eine feste Anzahl von Prozessen vorausgesetzt. Systeme mit dynamischer Erzeugung einer variablen Anzahl von Prozessen können also nicht richtig modelliert werden. Eine akzeptable Lösung des Problems besteht aber darin, die Anzahl der Prozesse, die simultan existieren können, zu begrenzen.

4. Die Gültigkeit der Antworten hängt davon ab, ob der (reduzierte) Nebenläufigkeitsautomat alle möglichen Verhaltensweisen modelliert. Eventuell werden bedenkliche Abläufe oder erreichbare Zustände moniert, die wegen bestimmter Variablenabhängigkeiten nicht auftreten können (vgl. Einleitung, S. 395, und Bemerkung zur Aufwandsreduzierung S. 389).

14.3.2 Axiomatischer Ansatz zur statischen Analyse

Der axiomatische Ansatz zur Beschreibung des Verhaltens nebenläufiger Systeme erfordert — wie bei sequentiellen Systemen — als Basis eine Modellierung der Semantik der Programmiersprache durch Axiome und Schlußregeln. Dabei tritt wieder das Problem auf, daß die Semantik aller Aspekte einer Sprache i. allg. nur mit erheblichem Aufwand oder evtl. gar nicht formal beschrieben werden kann. Ein Beispiel dafür sind Ausnahmen (exceptions) in Ada[14].
Das Vorgehen beim Beweis der partiellen Korrektheit eines Programmsystems basiert auf dem Vorgehen beim formalen Verifizieren sequentieller Programme (vgl. Kapitel 12.4) und umfaßt folgende Schritte:
Schritt 1 (lokaler Beweis):
Für jeden Prozeß des Systems wird das gewünschte Verhalten des Prozesses *unter gewissen Annahmen über die anderen Prozesse* bewiesen.
Schritt 2 (Kooperationsbeweis):
Die Beweise für die einzelnen Prozesse werden zu einem Beweis des Verhaltens des Gesamtsystems kombiniert. *Dabei wird bewiesen, daß die Annahmen zutreffen, die ein Prozeß bei Schritt 1 über die jeweils anderen Prozesse gemacht hat.*

Die kursiv hervorgehobenen Textstellen bezeichnen die Abweichungen vom Vorgehen bei sequentiellen Systemen, die (bei Schritt 1) nur Annahmen über ihr eigenes Verhalten (z. B. bei Schleifeninvarianten) machen müssen. Bei nebenläufigen Systemen ist obiger Zusatz erforderlich. Falls die einzelnen Prozesse jederzeit auf gemeinsame Variable zugreifen können, würde der Beweis sehr aufwendig. Bei jedem Berechnungsschritt des betrachteten Einzelprozesses müßte für alle möglichen Berechnungen der anderen Prozesse gezeigt werden, daß die gewünschte Transformation der Variablenwerte erfolgt. Diesen Aufwand kann man nur vermeiden, wenn man die Benutzung von gemeinsamen (globalen) Variablen verbietet — oder auf eng beschränkte Abschnitte in den Prozessen einschränkt.

Eine Selektion von Teilen eines Beweises bei der Programmsystemverifikation führt zu dem Vorgehen der symbolischen Ausführung von Programmen.
Bei nebenläufigen Systemen besteht der Ansatz darin, in einem ersten Schritt die einzelnen Prozesse symbolisch auszuführen. Dabei wird der symbolische Berechnungszustand eines Prozesses dargestellt durch:

1. die symbolischen Werte der Variablen,

2. die Pfadbedingung,

3. den Programmzähler.

[14] Daher wird in der Literatur nur eine Teilmenge von Ada, die Sprache, ADA-CF (ADA concurrency fragment) betrachtet, die allerdings weiterhin die interessanten Konstrukte für die Prozeßsynchronisierung durch Rendezvous (*call, accept, select*) enthält (s. [Dil 90a], [Dil 90b], [GeD 84]).

Eine symbolische Ausführung des Gesamtsystems kann nun in einem zweiten Schritt dadurch erfolgen, daß man alle beim Systemstart aktiven Prozesse unabhängig voneinander bis zur jeweils ersten Synchronisationsanweisung symbolisch ausführt. (Dabei muß wieder vorausgesetzt werden, daß es eine feste Anzahl von Prozessen gibt und keine gemeinsamen globalen Variablen während dieser Berechnungen verwendet werden.) Fallunterscheidungen und Schleifen in den einzelnen Prozessen sind dabei wie im sequentiellen Fall zu behandeln, d. h. für einen gewählten Zweig ist die Pfadbedingung um die entsprechende Zweigbedingung zu erweitern.
Bei Synchronisationsanweisungen sind (im Falle von Ada) passende *entry-call*- und *accept*-Anweisungen auszuwählen (falls es mehrere Möglichkeiten gibt), die Parameterwerte sind zu übergeben, der Rumpf der *accept*-Anweisung ist auszuführen und die Ausgabeparameterwerte sind zurückzugeben. Danach können die symbolischen Berechnungen wieder wie oben fortgesetzt werden. Nur bei *select*-Anweisungen mit Terminierungsalternative gibt es dabei Probleme, die besonders zu behandeln sind (genaueres s. [GeD 84], S. 190 ff.).

Der vorgestellte Ansatz der Verwendung von symbolischen Ausführungen und Zusicherungen für die formale Verifikation von nebenläufigen Programmsystemen (genannt: **Isolationsansatz**) hat folgende Vorteile:

1. Die Zahl der symbolisch auszuführenden Wege hängt nur linear von der Schleifenstruktur und der Anzahl der *accept*-Anweisungen der einzelnen Prozesse ab. Insbesondere steigt der Aufwand nicht exponentiell mit der Anzahl der Prozesse wie beim sogenannten **Verflechtungsansatz**, bei dem der Gesamtzustand aller Prozesse [gleichzeitig] schrittweise verfolgt wird. Der Aufwand steigt höchstens mit der Anzahl verschiedener Typen von Prozessen. Es ist also z. B. egal, ob ein System mit zwei oder fünf oder 100 essenden Philosophen zu verifizieren ist.

2. Die Beweise sind einfach zu komponieren. Bei Änderung eines Prozesses sind nur sein lokaler Beweis und die betroffenen Teile des Kooperationsbeweises zu ändern. Die lokalen Beweise der anderen Prozesse gelten weiterhin.

3. Durch entsprechend formulierte (globale) Zusicherungen lassen sich auch Beweise für die Verklemmungsfreiheit des Systems o. ä. führen (sogenannte **Sicherheitsbeweise**, genaueres s. [Dil 90b], S. 655 ff.).

Der Isolationsansatz kann allerdings im schlechtesten Fall ebenfalls einen mehr als linear steigenden Aufwand bei den Kooperationsbeweisen haben. (Bei dem Produzenten/Konsumenten-Problem mit begrenztem Puffer und den essenden Philosophen ist der Aufwand allerdings linear in der Anzahl der passenden „geklammerten Abschnitte“, in denen gemeinsame Variablen von Prozessen benutzt werden.)

14.4 Dynamische Analyse nebenläufiger Programme

Eine Überprüfung von Programmen, die sich nicht auf die von statischen Analysatoren vorgegebenen Teilmengen der Programmiersprachen beschränkt, ist die dynamische Analyse — das Testen — der realen Programme in ihrer realen Umgebung (Hardware, Betriebssystem, Übersetzer, Binder, Laufzeitsystem), die allerdings nur stichprobenhaft vorgeht.
Die für einen Test benötigten Testdaten können — wie im sequentiellen Fall — aus der Programmspezifikation oder der Struktur des implementierten Programms abgeleitet werden.

14.4.1 Spezifikationsorientierte Testdatenauswahl

Für das nebenläufige Verhalten des Systems muß eine entsprechende Spezifikation vorliegen, z. B. durch nebenläufige Pfadausdrücke. Daraus lassen sich sowohl gültige als auch ungültige Synchronisationssequenzen (SYN-Sequenzen) ableiten. Entsprechendes gilt bei einer Spezifikation durch einen endlichen Automaten. Er sollte wie der Nebenläufigkeitsautomat aufgebaut sein, aber die Spezifikation des Gewünschten und nicht die Modellierung der Realisierung beschreiben.
Die Ableitung von gültigen und ungültigen SYN-Sequenzen aus einem Pfadausdruck geschieht in folgenden Schritten:

1. äquivalente Vereinfachung des Pfadausdrucks [z. B. $(A|A) = A$],
2. einschränkende Vereinfachung des Pfadausdrucks (z. B. bei Wiederholung und Simultanität),
3. Auswahl der SYN-Sequenzen.

Da bei den Vereinfachungen des Pfadausdrucks nur Möglichkeiten weggelassen werden, ist jede durch obige Schritte gebildete Sequenz also eine gültige SYN-Sequenz. Ungültige SYN-Sequenzen lassen sich konstruieren, wenn mit Hilfe der Techniken aus Kapitel 5.1 zu dem vereinfachten Ausdruck P ein unvollständiger endlicher Automat konstruiert wird bzw. der vorliegende Nebenläufigkeitsautomat verwendet wird. Durch die Vervollständigung des Automaten (mit einem „ungültigen“ Fehlerzustand F) erhält man weitere Folgen, die (zum großen Teil) ungültige SYN-Sequenzen darstellen.
Die auszuwählenden gültigen und ungültigen SYN-Sequenzen lassen sich noch dadurch reduzieren, daß — wie im sequentiellen Fall (vgl. Kapitel 5.1) — für den konstruierten endlichen Automaten nur eine Teilmenge der möglichen Übergangsfolgen als SYN-Sequenzen gewählt werden.

Die Auswahlkriterien können wie in Kapitel 5.1 gewählt werden:

- alle Zustände müssen erreicht werden,
- alle Übergänge (Transitionen) müssen benutzt werden,
- alle Paare von Übergängen müssen benutzt werden, etc.

Eine andere Möglichkeit besteht darin, den Pfadausdruck direkt zu verändern: Bei Sequenzen kann jeweils ein Element der Folge weggelassen werden, die Nebenläufigkeit von zwei Teilausdrücken kann durch die Alternative der Teilausdrücke ersetzt werden (und umgekehrt). Damit erhält man in jedem Fall ungültige SYN-Sequenzen.

Beispiel 14.4.1
$A;B;C$ wird ersetzt durch $A;B$ sowie $A;C$ oder $B;C$.
$A|B$ wird ersetzt durch $A+B$. (Statt A oder B wird also A vor B oder B vor A oder A parallel zu B ausgeführt.)
$A+B$ wird ersetzt durch $A|B$.

Mit den vorgestellten Techniken wird also zu jedem Eingabedatum x eine Menge von gültigen und ungültigen SYN-Sequenzen erzeugt (vgl. Definition 14.1.1 auf Seite 382). Die gültigen SYN-Sequenzen müssen ausführbar sein; andernfalls liegt ein Synchronisationsfehler vor. Dies ist durch die Technik des reproduzierbaren Testens zu prüfen, d. h. für ein Eingabedatum x, eine SYN-Sequenz s und das erwartete (Soll-)Ergebnis y ist das reproduzierbare Testdatum (x, s, y) auszuführen (s. Definition 14.2.4 auf S. 391).
Die ungültigen SYN-Sequenzen dürfen nicht ausführbar sein, sonst liegt ein Synchronisationsfehler vor (vgl. Def. 14.1.2 auf S. 383). Beim Versuch, ungültige Sequenzen auszuführen, kann eine Verklemmung des Systems auftreten (vgl. Seite 392). Ist die SYN-Sequenz mit x ausführbar und gültig, ist zu überprüfen, ob eine Abweichung zwischen dem Istdatum y' und dem Solldatum y vorliegt; wenn ja, liegt ein Berechnungsfehler vor.

Beispiel 14.4.2
Beim Erzeuger-Verbraucher-System (vgl. Beispiel 14.1.5) besteht eine Eingabe x aus der Folge der Elemente, die der Erzeuger produzieren soll, z. B. $x = (7, 5, 11, 3, 2, 6)$. Bei einer spezifizierten Puffergröße von 2 sind z. B. die folgenden SYN-Sequenzen von A (Ablegen) und H (Holen) gültig (vgl. Übung 14.3):

$$s1 = AHAHAHAHAHAH,\ s2 = AAHHAAHHAAHH.$$

Ungültig sind dagegen die Sequenzen

$$s3 = AAAHHHAAAHHH,\ s4 = AAAAAAHHHHHH.$$

In beiden Fällen müssen in den Sequenzen je sechsmal die Operationen A und H vorkommen, da sechs Elemente erzeugt (und verbraucht) werden. Sequenzen mit weniger Operationen A (und H) würden nicht alle Elemente von x abarbeiten, Sequenzen mit mehr als sechs Operationen A (und H) passen dagegen nicht zur Eingabe x, da es einen Zugriffsfehler (auf nicht vorhandene Komponenten von x) geben würde.

Die Schwierigkeit bei dem vorgestellten Ansatz zum spezifikationsorientierten Testen besteht darin, für eine Eingabe x eine Menge relevanter SYN-Sequenzen zu bestimmen (s. obiges Beispiel 14.4.2). Werden umgekehrt zuerst die SYN-Sequenzen ausgewählt, so müssen alle möglichen Systemzustände des nebenläufigen Systems berücksichtigt werden. Eine Testdatenauswahl auf Basis einer vollständigen Systemzustandsmenge ist aber aus Komplexitätsgründen nicht praktikabel. Die Spezifikation muß also abstrakt und trotzdem aussagekräftig sein, um eine relevante Testdatenauswahl mit annehmbaren Aufwand zu ermöglichen. Darüberhinaus müssen meistens unterstützende, zusätzliche und nur für den Test relevante formale Beschreibungen hinzugefügt werden (s. [Sch 88a]). Die Testdatenauswahl auf der Basis der Spezifikation stellt also hohe Anforderungen an die Testpersonen, da die Erzeugung der Testdaten (fehlersensitive Eingabedaten x, SYN-Sequenzen s, erwartete Ausgaben y) nicht (voll) automatisiert werden kann.

Die Beschränkung auf SYN-Sequenzen hat einen Mangel: Es werden für die Nebenläufigkeit $p + q$ zweier Operationen p und q nur die möglichen Sequenzen $p;q$ und $q;p$ überprüft, nicht jedoch, ob p und q wirklich nebenläufig ausgeführt werden können. Dazu müßten einfache Aufrufe von Operationen in eine Sequenz aus *start(p)* und *end(p)* (für Operation p) zerlegt werden. Ein Aufrufkontrollprozeß müßte dann für die nebenläugigen Operationen p und q feststellen, ob z. B. die Sequenz *(start(p)* σ *start(q));(end(p)* σ *end(q))* ausführbar ist[15], d. h. p und q sind potentiell nebenläufig[16].

14.4.2 Implementationsorientierte Testdatenauswahl

Auf Basis der Programmstruktur wird zunächst eine Menge von ausführbaren SYN-Sequenzen ausgewählt. Für jede ausgewählte SYN-Sequenz wird eine fehlersensitive Menge von Eingabedaten bestimmt. Dies kann — wie im sequentiellen Fall (vgl. Kapitel 11.2) — von Hand, durch automatische, aber zufällige Erzeugung und schließlich durch symbolische Ausführung erreicht werden (vgl. Kapitel 12.3 und 14.3). Für die Auswahl von SYN-Sequenzen sollte versucht werden, sequentielle Methoden wie z. B. die konventionellen Überdeckungsmetriken anzuwenden. Für nebenläufige Programme ist die Auswahl jedoch weitaus schwieriger und komplexer, da die

[15] σ ist der Shuffle-Operator, d. h. $x \ \sigma \ y = (x;y)|(y;x)$; x und y finden also in beliebiger sequentieller Reihenfolge statt.

[16] Ob die einzelnen Anweisungen von p und q nebenläufig ausführbar sind, ist allerdings nur festzustellen, wenn auch ihre Ausführung vom Aufrufkontrollprozeß gesteuert werden kann. Das setzt aber einen Eingriff in die bestehenden Programme voraus, was i. allg. nicht praktikabel ist.

Menge der SYN-Sequenzen von der Gesamtmenge aller möglichen Systemzustände abhängt, d. h. die Zustände der einzelnen sequentiellen Teilprozesse reichen nicht aus. Als Grundlage sollte daher der Nebenläufigkeitsautomat zu einem Programmsystem herangezogen werden (s. Seite 387). Bezogen auf den Graphen des Nebenläufigkeitsautomaten können nun Überdeckungskriterien definiert werden, die analog zu den Kriterien für den Kontrollflußgraphen eines sequentiellen Programms gebildet werden (vgl. Kapitel 7.2).

Definition 14.4.1 (Überdeckungkriterien nebenläufiger Programme)
Sei A der Nebenläufigkeitsautomat zu einem nebenläufigen Programm P. Dann gibt es folgende Kriterien für die Menge W der auszuführenden Wege beim Testen von Programm P:

1. **alle Wege:** *P ist gleich der Menge aller Synchronisationssequenzen von A (vgl. Teil 3 von Definition 14.2.3, S. 390).*
2. **alle Anfangswege:** *P ist gleich der Menge aller Anfangswege. (Ein Anfangsweg ist eine Synchronisationssequenz, deren zugehörige Transitionssequenz [vgl. Teil 2 von Definition 14.2.3] mit dem Anfangszustand von A beginnt.)*
3. **alle Nebenläufigkeitszustände:** *Für jeden Nebenläufigkeitszustand z von A enthält P mindestens einen Anfangsweg, dessen Transitionssequenz den Zustand z enthält.*
4. **alle Transitionen:** *Für jede Transition t von A enthält P mindestens einen Anfangsweg, dessen Transitionssequenz die Transition t enthält.*

Beispiel 14.4.3
Für das Problem der beiden essenden Philosophen (vgl. Beispiel 14.2.2 und Abbildung 14.3) gilt folgendes:

1. *Für das Kriterium „alle Nebenläufigkeitszustände“ reicht ein Weg: (P12, P22), (P13, P22), (P15, P22), (P16, P22), (P12, P22), (P12, P23), (P12, P25), (P12, P26), (P12, P22), (P13, P22), (P13, P23). Dies bedeutet: erst ißt Philosoph(1), dann Philosoph(2), dann nehmen beide die linke Gabel (Verklemmung).*
2. *Für das Kriterium „alle Transitionen“ reichen zwei Wege: ein Weg wie oben bei „alle Nebenläufigkeitszustände“, dann noch ein Weg mit der anderen Transition in den Verklemmungszustand, d. h. zum Schluß (P12, P22), (P12, P23), (P13, P23), statt (P12, P22), (P13, P22), (P13, P23).*
3. *Die Kriterien „alle Anfangswege“ und „alle Wege“ sind mit einer endlichen Menge von Wegen nicht zu erfüllen, da der Graph des Nebenläufigkeitsautomaten Zyklen enthält. Diese Kriterien sind also nur sinnvoll für Graphen ohne Zyklen bzw. die Anzahl der Schleifendurchläufe ist zu begrenzen (vgl. Abschnitt 7.2.4).*

Vor einer reproduzierbaren Testausführung mit einer Eingabe x und einer SYN-Sequenz s ist durch Vergleich mit der Spezifikation festzustellen, ob die Sequenz s gültig ist; falls nein, liegt ein Synchronisationsfehler vor. Im anderen Falle ist das Ergebnis y' der reproduzierbaren Ausführung (mit Eingabe x und gültiger Sequenz s) mit der Sollausgabe y zu vergleichen um festzustellen, ob ein Berechnungsfehler vorliegt.

Mit diesem Vorgehen können also sowohl Synchronisationsfehler als auch Berechnungsfehler aufgedeckt werden. Allerdings werden nur ausführbare ungültige SYN-Sequenzen erkannt, nicht jedoch gültige unausführbare SYN-Sequenzen. (Dazu dient das spezifikationsorientierte Testen, s. Abschnitt 14.4.1). Die Überprüfung, ob eine SYN-Sequenz gültig ist, setzt voraus, daß die Spezifikation entsprechend formal vorliegt (z. B. durch Pfadausdrücke). Die Auswahl passender Eingabedaten ist relativ schwierig und nicht automatisierbar.

14.5 Analyse von verteilten und Echtzeitsystemen

Bei den bisher betrachteten (nebenläufigen) Systemen spielten Zeiten für die Abarbeitung von Anweisungsfolgen und die Verteilung der Berechnungen auf mehrere Prozesse keine Rolle für die Korrektheit der Systeme.
Im Gegensatz dazu zeichnen sich **Echtzeitsysteme** durch **harte Zeitbedingungen** aus. Das bedeutet, daß sie nur korrekt arbeiten, wenn gewisse Anweisungen in bestimmten Zeitintervallen abgearbeitet werden. Dabei geht es nicht um **interne Zeitbedingungen**, d. h. darum, daß die Systemausgaben möglichst schnell erzeugt werden, nachdem die Eingaben vorliegen. Vielmehr geht es bei Echtzeitsystemen um **externe Zeitbedingungen**, d. h. darum, daß die Systeme in gewissen Zeitintervallen Informationen aus der Umwelt erhalten oder an sie abgeben. Die Verarbeitungsschritte des Programmsystems müssen also mit den externen Eingaben synchronisiert werden. Ein Beispiel dafür ist ein Verkehrszählungssystem, das für jeden Verkehrsteilnehmer Zähldaten aufnehmen und statistisch verarbeiten muß. Werden die Daten nicht rechtzeitig verarbeitet oder zwischengespeichert, gehen Informationen verloren und die Verkehrszählung ist falsch. Noch kritischer sind Echtzeitanforderungen bei **eingebetteten Systemen**. Das sind Systeme, die mit technischen oder biologischen Prozessen verbunden sind und die auf Ereignisse oder veränderte Zustände dieser Prozesse rechtzeitig reagieren müssen, damit der Zustand des realen Systems in tolerierbaren, spezifizierten Grenzen bleibt. Abhängig von den Eigenschaften der zu regelnden Prozesse müssen also Spezifikationen für die Reaktionszeiten des Programmsystems erstellt werden. Besonders kritisch ist die Obergrenze für die Reaktionszeit, aber Untergrenzen können auch relevant sein: Bei einem Doppelklick mit einer (Computer-)Maus darf der erste Klick nicht zu schnell (im Vergleich zum zweiten) verarbeitet werden, da der Doppelklick sonst als zwei einzelne Klicks interpretiert wird. Beispiele für eingebettete Systeme sind

Steuerungsanlagen für chemische oder thermische Prozesse, aber auch Steuerungen von Robotern, (Stahl-)Walzstraßen, Flugzeugen und Raketen, sowie Geräte zur Diagnose und Therapie in der Medizin. Bei diesen Systemen haben (Regelungs-)Fehler direkte Auswirkungen auf Leben und Gesundheit der jeweils betroffenen Menschen.

Die Verteilung der Berechnung auf mehrere Prozesse wird dann zu einem besonderen Problem, wenn es sich um **verteilte Systeme** handelt. Sie zeichnen sich durch dezentralisierte Systemkontrolle, verteilte Verarbeitungseinheiten und verteilte Betriebsmittel aus. Die räumliche, geographische Verteilung der Systemkomponenten ist dagegen kein zwingendes Kriterium für ein verteiltes System. (Außerdem ist unklar, wo die Grenze zu ziehen wäre: bei einem maximalen Abstand von 5 Metern, 10 Metern, 100 Metern?) Ein Multiprozessorsystem mit strenger Kopplung zwischen den Einzelprozessoren ist also demnach kein verteiltes System. Abstrakt formuliert besteht ein verteiltes System aus einem System kooperierender, parallel ablaufender Prozesse mit jeweils lokalem Speicher, bei dem die Prozesse keine identische Sicht des globalen Systemzustands haben. Außerdem gibt es keinen Prozeß, der dafür sorgen könnte, daß alle anderen Prozesse eine solche konsistente und identische Sicht des globalen Systemzustands erhalten (es sei denn, man toleriert eine [starke] Verlangsamung des Systemablaufs und setzt ein zuverlässiges Übertragungsmedium für Nachrichten voraus). Daher versagen die bisher vorgestellten Analysemethoden, die das Konzept eines globalen Systemzustands benutzen, bei den meisten Anwendungen.

14.5.1 Analyse von Echtzeitsystemen

Die dynamische Analyse (das Testen) von Echtzeitsystemen wirft spezielle Probleme auf:

1. Wenn externe Unterbrechungen (interrupts) zu jeder Zeit auftreten können, ist es praktisch nicht möglich, alle möglichen Eingabesituationen vorherzusagen.

2. Die Interpretation der Ausgaben und Reaktionen des Systems kann sehr schwierig sein; insbesondere kann die Instrumentierung der Programme für die Zwecke der Überdeckungsmessung oder Fehlerlokalisierung das normale Zeitverhalten verfälschen, weil der Testvorgang mit der normalen Prozeßausführung *interferiert*. Damit wird also nicht mehr das tatsächliche, sondern ein modifiziertes Programmsystem analysiert.

Problem 1 ist nicht aus der Welt zu schaffen, wenn es reale Ereignisse gibt, auf die sofort reagiert werden muß, um eine Katastrophe zu verhindern. Andernfalls sollten natürlich (bei der Programmierung) strukturierte Formen der Ereignisverarbeitung verwendet werden.

Problem 2 kann aber dadurch abgemildert oder praktisch beseitigt werden, daß zusätzliche Hardware eingesetzt wird. Dabei werden die internen Busse des Hardwaresystems „angezapft“, so daß sämtliche Programmablaufinformationen aufgezeichnet werden können. Falls der Speicher für die Aufzeichnungen eines Testlaufs ausreicht, kann dann im Nachhinein der gesamte Programmablauf noch einmal analysiert werden, ohne daß die befürchtete Interferenz mit den realen Abläufen auftritt. In realistischen Anwendungen können allerdings nicht alle Informationen gespeichert werden. Daher sind interessante Zeitausschnitte oder Prozeßzustände auszuwählen. Damit können zwar nicht alle, aber wenigstens die kritischen Systemzustände und -übergänge untersucht werden.

Wegen der Probleme des Testens von Echtzeitsystemen bietet sich also die statische Analyse oder formale Verifikation solcher Systeme an. Dazu sind natürlich Modelle, Sprachen und Techniken erforderlich, mit denen die Echtzeitanforderungen beschrieben werden können. Beispiele dafür sind synchrone Datenflußsprachen wie LUSTRE, asynchrone Spezifikationen wie ATP, zeitbehaftete Automaten oder Petri-Netze oder kommunizierende Echtzeitzustandsmaschinen (communicating real-time state machines, CRSM), die auch Eigenschaften der Umgebung spezifizieren können und ausführbar sind (genaueres siehe [BuV 95], [FMM 94], [KeG 92]).

Ein Problem bei der Spezifikation von Zeitbeschränkungen ist die Behandlung der Zeit selbst. Synchrone Modelle verwenden nur diskrete Zeitpunkte, z. B. Zehntelsekunden. Asynchrone Modelle lassen dagegen beliebig dicht folgende Zeitpunkte zu — was für manche Anwendungen erforderlich ist, da ein minimaler Abstand zwischen realen Ereignissen nicht immer garantiert werden kann.
Damit Zeitangaben und das Verhalten der Systeme (über einen Zeitraum hinweg) formal behandelt werden können, ist eine logische Notation für Zeit notwendig. Dazu reicht die Sprache der klassischen Logik nicht aus, vielmehr ist eine sogenannte **temporale Logik** (oder für absolute Zeitangaben eine **Echtzeitlogik** [Real Time Logic, RTL]) nötig. Es gibt dazu verschiedene Ansätze, um über zukünftige oder vergangene Ereignisse reden zu können, z. B. die temporalen Operatoren *Futur* und *Past* (Vergangenheit) (genaueres siehe [FMM 94]; [GFB 93], Kap. 3 und [JaM 94]).

Eine Spezifikation des Zeitverhaltens mit Formeln der temporalen Logik kann für verschiedene Zwecke benutzt werden:

1. Widersprüche in der Spezifikation aufdecken:
   Widersprüche liegen dann vor, wenn die Formeln nicht erfüllbar sind, d. h. in keinem konkreten Fall (bei einer bestimmten zeitlichen Anordnung von Ereignissen) richtig sein können.

2. Überprüfung, ob ausführbare Ereignisfolgen eines vorliegenden Systems gültig sind, d. h. ob sie die Spezifikation in Form der temporalen Formeln erfüllen (engl.: history checking):
   Die Ereignisfolgen werden dabei meistens aus entsprechenden endlichen Automaten (mit Zeitangaben) abgeleitet.

3. Testdatenerzeugung für ein vorliegendes System:
   Dazu können die Ereignisfolgen und Variablenwerte benutzt werden, die beim Versuch, die Erfüllbarkeit der Formeln zu beweisen, konstruiert werden müssen — bei Zweck 1.

Das Vorgehen bei Zweck 1 (Widersprüche aufdecken) und Zweck 2 (history checking) führt i. allg. zu unentscheidbaren Problemen. Sie sind aber dann entscheidbar, wenn die Wertebereiche der Variablen endlich sind oder wenn eine Quantifikation über die Zeit („zu allen Zeitpunkten gilt ... “) verboten wird. Die Modelle und Spezifikationen sind also entsprechend einzuschränken, was in vielen Fällen möglich ist.
Bei komplexen Systemen ist die Lösung der (entscheidbaren) Probleme allerdings meist immer noch zu aufwendig. Daher muß wieder versucht werden, die Spezifikation zu modularisieren, etwa mit objektorientierten Techniken (wie Klassenbildung, Vererbungsbeziehungen und generischen Klassen).

Für den Modul-, Integrations- und Systemtest von Echtzeitsystemen gilt im Prinzip das gleiche wie bei zeitunkritischen Systemen (vgl. Kapitel 13). Die Integrationsschritte sind nur weiter zu differenzieren:

Schritt 1: Testen der Module und Komponenten in einer simulierten Umgebung
Da ein Test der (noch unzuverlässigen) Software mit der realen Umgebung zu kostspielig und gefährlich ist, sind die folgenden Aspekte bzw. Komponenten zu simulieren: Eingaben aus der realen Umgebung, Hardware-Funktionen und das Programm zur Steuerung aller Systemteile.

Schritt 2: Testen mit dem Steuerungsprogramm
Reichte bei Schritt 1 für jedes Modul oder jede Komponente eine einfache Simulation der entsprechenden Steuerprogrammteile, wird nun das komplette Steuerungsprogramm eingebunden. Damit sollten die Komponenten noch einmal getestet werden und eventuell eine weitere Zusammenfassung von Komponenten.

Schritt 3: Teilsystemtest
Bei großen Systemen mit vielen Komponenten gibt es meistens (zur Bündelung von Dateneingabe und -ausgabe) Kommunikationsmultiplexer. Diese sind beim Testen der Teilsysteme einzubinden. Dabei sind neue, zufällig erzeugte Folgen und Mischungen von Aktionen von Teilkomponenten zu testen. (Bei den ersten beiden Schritten wurden genau bekannte Arten und Folgen von Aktionen getestet, um die Bewertung der Testergebnisse zu vereinfachen.) Die zeitliche Dichte bzw. Menge von Systemeingaben wird nicht mehr beschränkt, damit auch die Systemreaktion auf Überlast getestet wird.

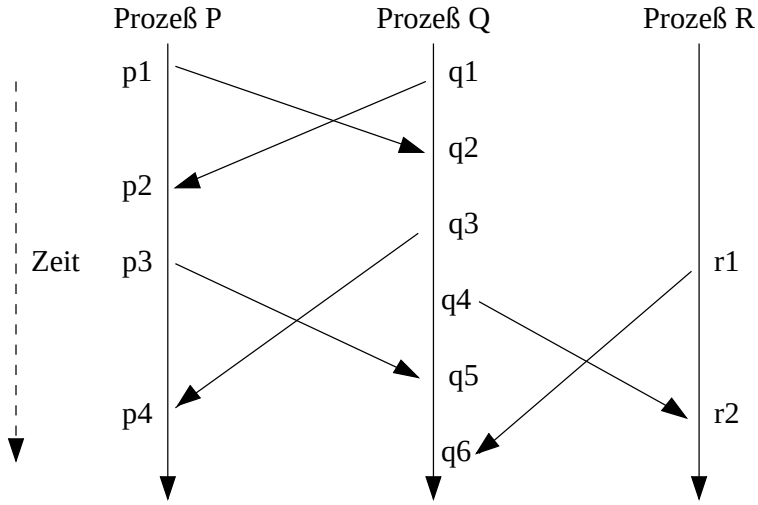
Schritt 4: Integrations- und Systemtest
Beim (Teil-)Systemtest sind hier die simulierten Eingaben durch echte Eingaben zu ersetzen. Das bezieht sich auf die Form der Eingaben, aber auch auf die Auswahl der Daten aus einem realen Betrieb — falls es schon vorher ein ähnliches System gab. Zuletzt sind die Teilsysteme zu integrieren, insbesondere die Echtzeitteilsysteme (die als

Schnittstelle Sensoren und Aktoren haben) mit den Anwendungsteilsystemen. Dabei ist speziell zu testen, ob die Anwendungsteilsysteme richtig auf Unterbrechungen durch die Echtzeitteilsysteme reagieren.

Schritt 5: Akzeptanztest/Abnahmetest
Bei diesem Test geht es nicht nur um die funktionale Korrektheit, sondern insbesondere um die globale Einhaltung der Leistungskriterien.

14.5.2 Analyse verteilter Systeme

Verteilte Systeme können nur mit Hilfe von Nachrichten kommunizieren, die jeweils ein Prozeß sendet und ein anderer Prozeß empfängt. Verteilte Systeme haben daher eine ähnliche Eigenschaft wie Systeme in der Relativitätstheorie: Es gibt keinen globalen Zustand, keine globalen Uhren und somit keine globale Zeit mit einer totalen Ordnung. Daher ist für je zwei Ereignisse der Art *senden* oder *empfangen* nicht immer klar, ob sie nacheinander oder exakt gleichzeitig stattfinden. Stattdessen muß eine partielle zeitliche Ordnung von Ereignissen angenommen werden (die mit **Präzedenz** bezeichnet wird).



Legende: Quer bzw. diagonal verlaufende Pfeile führen von einem Sendeereignis zu einem Empfangsereignis, z. B. von p1 nach q2.

Abb. 14.4: Beispiel einer Präzedenzrelation „$\rightarrow$“ (nach [ChY 83], S. 712)

Für die Präzedenzrelation „$\rightarrow$“ und alle Ereignisse e, f, g gilt folgendes:

1. Wenn e und f zum selben Prozeß gehören und e vor f stattfindet, dann gilt $e \rightarrow f$. (Beispiel: p1 $\rightarrow$ p2 in Abbildung 14.4)

2. Wenn e ein Sendeereignis in P_i für P_j ist und f das entsprechende Empfangsereignis in P_j ist, dann gilt: $e \rightarrow f$. (Beispiel: p1 $\rightarrow$ q2 in Abbildung 14.4)

3. $e \rightarrow f$ und $f \rightarrow g$ impliziert $e \rightarrow g$ (Beispiel: $p1 \rightarrow q2$, $q2 \rightarrow q3$, also $p1 \rightarrow q3$ in Abb. 14.4)

Gegenüber der temporalen Logik (TL) hat die Verwendung der Präzedenzrelation folgende Vorteile:

- es gibt keine komplizierten globalen Invarianten und nicht so viele temporale Operatoren (wie in TL),
- bessere Verständlichkeit der Präzedenzrelation.

Gegenüber den SYN-Sequenzen hat die Präzedenzrelation folgende Vorteile:

- es ist keine totale Ordnung nötig,
- manche Eigenschaften (z. B. echte Nebenläufigkeit) sind nicht durch Sequenzen ausdrückbar,
- die Verklemmunsgfreiheit ist beweisbar.

Für das Testen verteilter Systeme ergeben sich eine Reihe neuer Probleme bzw. Fragen:

Steuerungsproblem: Wie kann das jeweilige Testobjekt gezielt beeinflußt und wie können die entsprechenden Reaktionen und Ausgaben aufgezeichnet werden, wenn Testobjekt und Testsubjekt (der Prozeß, der den Test steuert) auf verschiedenen Prozessoren des verteilten Systems residieren?

Beobachtungsproblem: Wie können Verfälschungen des Systemverhaltens (durch Messen und Beobachten beim Testen) verhindert oder zumindest begrenzt werden?

Informationsmengenproblem: Wie kann die beim Testen anfallende sehr große Informationsfülle verarbeitet oder reduziert werden?

Testdatenerzeugungsproblem: Welche speziellen Tests sind bei der Integration verschiedener Prozesse nötig und sinnvoll?

Die Steuerungs- und Beobachtungsprobleme können durch eine entsprechende Hard- und Softwarekonfiguration gelöst werden: Für jeden Prozessor und die darauf residierenden Prozesse des zu testenden Systems gibt es einen speziellen Test- und Diagnoseprozeß. Damit kann der Modul- und Integrationstest der Prozesse auf diesem Prozessor ohne Verzögerungs- und Beobachtungsprobleme durchgeführt werden.

Für den Integrationstest von Prozessen auf verschiedenen Prozessoren und den Gesamtsystemtest müssen die entsprechenden Testeingaben auf die einzelnen Prozessoren verteilt werden und die auf den einzelnen Prozessoren ermittelten Reaktionen und Ausgaben zentral ausgewertet werden. Dazu ist ein zentraler Testprozeß (auf irgendeinem Prozessor) notwendig, der diesen gesamten Testablauf steuert. Bei einer reinen Softwarelösung des Testproblems, d. h. bei dem üblichen Instrumentieren der Programme, um Ablaufinformationen zu erzeugen, ist das Beobachtungsproblem nicht gelöst: Durch die zusätzlichen Verzögerungen im Programmablauf auf den einzelnen Prozessoren können neue Synchronisierungsfehler auftreten oder bestehende Synchronisierungsfehler aufgehoben (maskiert) werden. Dieses Problem kann nur durch zusätzliche Hardware gelöst werden, welche die entsprechenden Ereignisse ohne Zeitverzögerung aufzeichnet (vgl. Lösung bei Echtzeitsystemen auf S. 406).
Bei der Wiederholung von Tests können natürlich immer noch Probleme bei einer echten nichtdeterministischen Auswahl zwischen Alternativen entstehen (z. B. bei *select*-Anweisungen mit mehreren erfüllten Rendezvous-Bedingungen) oder bei minimalen Zeitverschiebungen. Dieses Problem läßt sich mit lokalen Monitoren (je einer pro Prozessor) lösen, die eine einmal vorgegebene Synchronisierungssequenz erzwingen, indem sie zu früh eintreffende Ereignisse (z. B. bei *send/receive*) entsprechend zurückstellen. (Dieses Problem tritt auch schon bei nicht verteilten nebenläufigen Programmsystemen auf, vgl. Aufrufkontrolle in Abschnitt 14.2.2 für den Monoprozessorfall).
Wie bei Echtzeitsystemen kann das Informationsmengenproblem dadurch gelöst werden, daß nur kritische (d. h. wichtige) Ereignisse aufgezeichnet werden. Das sind Ereignisse, die andere Prozesse beeinflussen können wie z. B. Prozeßstart und Prozeßende, Nachrichtensenden (*send*) und Nachrichtenerhalten (*receive*). Beim Programmieren sollten ebenfalls entsprechende Ereignisse als wichtig deklariert werden, da die programmierende Person am besten weiß, was wichtig ist.
Für das Testdatenauswahlproblem können bei verteilten Systemen spezielle Kriterien für den Integrationstest angegeben werden.

Definition 14.5.1 (Überdeckungskriterien für verteilte Systeme)
Folgende Kriterien sind alternativ anwendbar:

1. **jedes Kommunikationsereignis:**
   Für je zwei Prozesse P_i und P_j muß jede mögliche Kommunikation zwischen P_i und P_j (mittels send/receive oder Rendezvous) in einem Testfall für P_i und P_j vorkommen.

2. **jede Kommunikationssequenz:**
   Für je zwei Prozesse P_i und P_j muß jede mögliche Sequenz von Kommunikationsereignissen für P_i und P_j, bei der sich keine Zustände wiederholen[17] *(außer*

[17] Dabei ist vorauszusetzen, daß der Nebenläufigkeitsautomat endlich viele Zustände hat.

dem Anfangs- und Endzustand des Teilsystems aus P_i und P_j), in einem Testfall für P_i und P_j vorkommen.

3. **jeder zeitbeschränkte Zweig:**
   Für je zwei Prozesse P_i und P_j müssen Zweige in einem Prozeß P_i, die bei Überschreitung bzw. Unterschreitung einer Auszeit (time out) zu durchlaufen sind, in einem Testfall für P_i (und dem Prozeß P_j, der die Auszeit verursacht) vorkommen.

4. **jede Prozessoraktivierungssequenz:**
   Für jede Aufgabe A und jede Folge $P_1, \ldots, P_k$ von Prozessen, die auf entsprechenden Prozessoren aktiviert werden müssen, um die Aufgabe A zu erfüllen, ist eine entsprechende Folge von Prozessoraktivierungen zu testen. (Dies ist nur notwendig, wenn die Prozesse $P_1, \ldots, P_k$ nicht dauernd aktiviert sind, sondern zwischenzeitlich ausgelagert werden, weil es zu wenig Prozessoren gibt.)

5. **jedes Paar nichtsynchronisierbarer Kommunikationsereignisse:**
   Für jeden Prozeß P_i, der mit zwei anderen Prozessen P_j und P_k über Kommunikationsereignisse e und f kommuniziert, bei denen die Reihenfolge (e vor f oder f vor e) nicht determiniert ist, sollten beide Ereignisreihenfolgen getestet werden.

14.6 Übungen

Übung 14.1:
Geben Sie ein sequentielles Programm (für einen Monoprozessor) an, welches die strikt parallele Ausführung der folgenden n Zuweisungen (bei der zuerst gleichzeitig die rechten Seiten der Zuweisungen ausgewertet und dann diese Werte den linken Seiten zugewiesen werden) simuliert: $x_1 := y_1; x_2 := y_2; \ldots ; x_n = y_n$.
Dabei kann eine Variable $y_k, k = 1, \ldots, n$ mit einer der Variablen $x_i, i = 1, \ldots, n$ übereinstimmen.

Übung 14.2:
Geben Sie die gültigen SYN-Sequenzen für das Erzeuger-Verbraucher-Problem aus Beispiel 14.1.5 an, bei dem dreimal A (Ablegen) und dreimal H (Holen) aufgerufen wird. Begründen Sie Ihre Angaben.

Übung 14.3:
Geben Sie einen endlichen Automaten an, der die gültigen (erlaubten) SYN-Sequenzen der Operationen H (Holen) und A (Ablegen) beim Erzeuger-Verbraucher-Problem beschreibt. Nehmen Sie im Unterschied zu Beispiel 14.1.5 an, daß der Erzeuger und der Verbraucher jeweils beliebig oft die Operation H bzw. A aufrufen darf.
Wie viele Zustände sind erforderlich, wenn man den Fall beschreiben will, daß jeweils genau dreimal die Operation A und dreimal die Operation H ausgeführt wird (s. Beispiel 14.1.5)?

Übung 14.4:

Betrachten Sie die Nebenläufigkeit der Pfadausdrücke $P = (a; b; c)$ und $Q = (d; e)$.

(a) Welche (zehn) verschiedenen sequentiellen Reihenfolgen sind für $P + Q$ erlaubt? (Eine Reihenfolge ist z. B. $a; d; e; b; c$.)

(b) Welche weiteren (15) Reihenfolgen sind bei $P + Q$ erlaubt, bei denen ein oder zwei Paare von „Ereignissen“ aus P und Q gleichzeitig stattfinden? (Ein Beispiel ist $d; (a + e); b; c$. Dabei finden also a und e „gleichzeitig“ statt.)

Übung 14.5:

(a) Geben Sie analog zu den Beispielen 14.2.1 und 14.2.2 die reduzierten synchronisierten Flußgraphen und den Nebenläufigkeitsautomaten A für das 2-Philosophen-Problem an, wenn die Philosophen die Gabeln in beliebiger Reihenfolge (also nicht „links vor rechts“) aufnehmen und ablegen dürfen.

(b) Geben Sie alle zu Aufgabe (a) gehörigen Synchronisationssequenzen, die *nicht* in eine Verklemmung führen, in Form eines Pfadausdrucks an (s. Def. 5.1.1).

(c) Geben Sie analog zu Aufgabe (b) alle Synchronisationssequenzen an, die zum Schluß in eine Verklemmung führen. Benutzen Sie dazu sowohl sequentielle als auch nebenläufige Pfadausdrücke (s. Def. 5.1.1 und Def. 14.2.1).

(d) Geben Sie Synchronisationssequenzen zu Aufgabe (a) an, bei denen Philosoph 1 bzw. Philosoph 2 verhungert — oder zeigen Sie, daß dies nicht vorkommen kann.

Übung 14.6:

Benutzen Sie den Nebenläufigkeitsautomaten A aus Aufgabe (a) von Übung 14.5, um alle Tests bzw. Wege von A zu bestimmen, welche die Kriterien *alle Nebenläufigkeitszustände* bzw. *alle Transitionen* erfüllen (s. Teil 3 und 4 von Definition 14.4.1, vgl. Beispiel 14.4.3).

Übung 14.7:

(a) Geben Sie für Abbildung 14.4 alle Ereignispaare (e, f) an, die in der Präzedenzrelation $e \to f$ stehen und für die gilt:

i. e gehört zu Prozeß P, f zu Prozeß Q;

ii. e gehört zu Prozeß Q, f zu Prozeß R;

iii. e gehört zu Prozeß P, f zu Prozeß R.

(b) Geben Sie für Abbildung 14.4 und jedes Paar von Prozessen jeweils ein Ereignispaar (e, f) an, das *nicht* in der Präzedenzrelation $e \to f$ steht.

14.7 Verwendete Quellen und weiterführende Literatur

Die ersten **Sprachkonstrukte für Nebenläufigkeit** bzw. Parallelität, *fork* und *join*, stammen von Conway (1963). Dijkstra schlug die strukturierte Klammerung mit *cobegin-coend* und **Semaphore** zur Implementierung von kritischen Abschnitten vor (1968) sowie **Wächter** (guards) für die nichtdeterministische, selektive Auswahl (1975). Hoare propagierte 1974 die **Monitore** und 1978 mit den kommunizierenden sequentiellen Prozessen (communicating sequential processes, **CSP**) das Konzept der selektiven Kommunikation (vgl. [SoM 87], Kapitel 9.1; [ZöH 88], Kapitel 2, 3, 5). Eine Implementation des begrenzten Speichers als „bounded buffer" findet man in [AnS 83], S. 31. Die Fehlerarten **Blockierung**, **Verklemmung** (deadlock) und **Verhungern** (livelock) werden vor allem im Zusammenhang mit Betriebssystemen und Petrinetzen betrachtet (siehe z. B. [Ric 85], Kap. 2.3; [Sta 90], Kap. 6 und 12; [Ger 84]). Tai hat die Konzepte der **Synchronisationssequenzen** und **Synchronisationsfehler** formalisiert und das Erzeuger-Verbraucher-Problem beispielhaft erläutert (s. [Tai 85], S. 311 f.).
Der **(nebenläufige) Pfadausdruck** als Erweiterung des (sequentiellen) Pfadausdrucks wurde von Seehusen definiert (s. [See 87]), der Shuffle-Operator von Shaw (s. [Sha 80]). **Reduzierte synchronisierte Flußgraphen** (*flowgraphs*) verwendet Taylor (s. [Tay 83], S. 364 f.) zur Beschreibung von Ada-Programmen; Long/Clarke definieren entsprechende Prozeßinteraktionsgraphen (*task interaction graphs*), bei denen jedem Knoten ein (Ada-ähnlicher) Pseudocode zugeordnet ist, der aus den auszuführenden Prozeßanweisungen besteht (s. [LoC 89], S. 44 ff.). Gerichtete Synchronisationskanten wurden von Taylor/Osterweil (als spezielle Kanten in *process augmented flowgraphs*) eingeführt (s. [TaO 80], S. 267 f.). Von dem Flußgraphen wurde der Begriff des **Nebenläufigkeitszustandsgraphen** (*concurrency state graph*) von Taylor abgeleitet bzw. von Long und Clarke als *task interaction concurrency graph* (**Nebenläufigkeitsautomat** in unserem Sinne) bezeichnet (s. [Tay 83], S. 366 f.; [LoC 89], S. 46 ff.; vgl. auch [Yo& 89], S. 202). Die Konstruktion des Nebenläufigkeitsautomaten basiert auf Vorschlägen von Taylor, sowie Long und Clarke (s. [Tay 83], S. 369 ff.; [LoC 89], S. 48). Die von mir definierten Begriffe **Zustands-, Transitions- und Synchronisationssequenz** entsprechen ungefähr dem von Taylor et al. vorgestellten Begriff der *concurrency history* (s. [Tay 83], S. 367), letzterer beschreibt allerdings nur die Folge der von einer Transitionssequenz durchlaufenen Zustände.
Bei der Ausführung von Programmen auf einem Monoprozessor reicht die serialisierte Beschreibung von parallelen Prozessen aus, die Tai eingeführt hat (s. [Tai 85], S. 311). Die Frage der korrekten Modellierung von fehlerhaften und nicht fehlerhaften Abläufen wird von Young et al. diskutiert (s. [Yo& 89], S. 201 f.). Sie haben auch das System der **essenden Philosophen** durch ein Ada-Programm und daraus abgeleitete Nebenläufigkeitsautomaten beschrieben (siehe [Yo& 89], [TLK 92]; vgl.

auch [LoC 89]). Dabei wurde vorgeschlagen, die Anzahl der Zustände zu reduzieren (bzgl. *rendezvous*, *delay*, und Prozeßaktivierung/-terminierung).
Das Konzept des **reproduzierbaren Tests** und die Ansätze zur reproduzierbaren Ausführung von Tests (Scheduler-, Uhr-, Aufrufkontrolle) stammen von Brinch Hansen und Tai (s. [Bri 78], [Tai 85]), die Tripel-Darstellung für reproduzierbare Testdaten von Schippers (s. [Sch 88a]). Zusätzlich hat Tai einen Ansatz vorgeschlagen (*accept control*), der die besonderen Probleme löst, die durch sich überlappende Ankünfte und Akzeptierungen von *entry calls* in Ada-Programmen entstehen (s. [Tai 85], S. 316). Weiterentwicklungen und Implementierungen der Verfahren (speziell für Ada-Programme) wurden von Brindle et al., Tai et al. und LeBlanc/Mellor-Crummey vorgestellt (s. [BTM 89], [TCO 91], [LeM 87]). Auf die Vor- und Nachteile der **statischen Analyse** hat z. B. Taylor hingewiesen (s. [Tay 84], S. 127–129). Auf das Problem der unklaren Semantik von nebenläufigen Programmiersprachen geht z. B. German in Bezug auf Ausnahmen (*exceptions*) in Ada ein (s. [Ger 84], S. 776).
Taylor, Long und Clarke haben **Analysetechniken** für Nebenläufigkeitsautomaten entwickelt, mit denen die Fragen nach möglichen Verklemmungen, Rendezvous und parallelen Aktionen beantwortet werden können (s. [Tay 83]; [LoC 89], Kap. 4). Eine empirische Untersuchung des Berechnungsaufwands für diese und andere Techniken (zur Feststellung von Verklemmungen) hat Corbett durchgeführt (s. [Cor 96]); dabei wurde festgestellt, daß keine Technik in allen Aspekten Vorteile gegenüber den anderen Verfahren aufweist. **Temporale Logik** wird in den Ansätzen von Young et al. verwendet; damit kann nicht nur über Zustände (wie bei [CES 86]), sondern auch über Ereignisfolgen argumentiert werden (s. [Yo& 89], S. 203 f.). Genaueres zur Technik der **Aufteilung (parceling)**, mit der bei der Konstruktion und Analyse des Nebenläufigkeitsautomaten die Komplexität reduziert wird, findet man bei Taylor (s. [Tay 83], S. 371 ff.).
Der axiomatische Ansatz zur **formalen Verifikation** und **symbolischen Ausführung** von nebenläufigen Programmen beruht auf den Arbeiten von Gerth/De Roever, die ein Beweissystem für eine Teilmenge von Ada und einen Beweis für das Erzeuger-Verbraucher-System angeben, sowie auf den Arbeiten von Dillon (s. [GeD 84], [Dil 90a], [Dil 90b]). Die komplizierte Semantik von Ada, insbesondere für Ausnahmen (*exceptions*), hat schon Hoare kritisiert (siehe Zitat bei [GeD 84], S. 159).
Petri-Netze anstelle von endlichen Automaten verwenden Morgan/Razouk für die Modellbildung und Analyse. Sie verweisen auf Projektionen, Reduktionen und Selektionen zur Verkleinerung bzw. Begrenzung des Zustandsraumes (s. [MoR 87]). Eine Kombination der Techniken *Analyse endlicher Automaten* und *axiomatische Verifikation* und ihre Anwendung zur Modellierung und Verifikation von Protokollen für asynchrone Botschaftenaustauschsysteme (*message passing systems*) stellen Joseph et al. vor (s. [JRT 86]). Eine Kombination der direkten symbolischen Ausführung der Einzelprozesse anhand der nicht reduzierten Flußgraphen und der statischen Analyse des Nebenläufigkeitsautomaten schlagen Young und Taylor vor (s. [YoT 86]). Ein ähnlicher Ansatz wird von Stavely et al. verfolgt, wobei die Suche nach tatsächlich

erreichbaren Zuständen bzw. Knoten, die Verklemmungen oder gefährlichen Parallelismus von Aktionen repräsentieren, durch Boolesche Ausdrücke (für Zustände) und eine Art von Pfadausdrücken (Ereignisausdrücken) gesteuert wird (s. [St& 85]).
Die parallele Ausführung von Anweisungen, die dieselbe Variable definieren bzw. definieren und referenzieren, ist nur eine Form einer Datenflußanomalie (vgl. Kap. 12.2 für sequentielle Systeme). Frühe Ansätze zur Erkennung von **Datenflußanomalien** beziehen sich auf einfache Synchronisationsmechanismen, die andere Prozesse komplett anstoßen oder auf ihr Ende warten (s. [Br& 79], [TaO 80]). Für eine Art paralleles FORTRAN gibt McDowell ein Analyseverfahren an (s. [McH 89]). Für Systeme mit Rendezvous-Synchronisation präsentieren Long/Clarke Verfahren zur Datenflußanalyse (s. [LoC 91]).
Von Tai stammen die Vorschläge und Konzepte zur **dynamischen Analyse** (zum **Testen**) von nebenläufigen Programmen (s. [Tai 85]). Die Verwendung und Manipulation von Pfadausdrücken bei der Testdatenauswahl auf Spezifikationsbasis basiert auf den Arbeiten von Seehusen und Heerdt/Vesper (s. [See 87], [HeV 88]). Auf die Probleme bei der spezifikationsorientierten Testdatenauswahl hat z. B. Schippers hingewiesen (s. [Sch 88a]). Taylor/Kelly stellen die symbolische Ausführung als Technik zur implementationsorientierten Testdatenauswahl vor (s. [TaK 86], S. 167); von ihnen stammen auch die Überdeckungskriterien für nebenläufige Programme.
Einführungen in (bzw. Überblicke über) die Programmierung, Verifikation, Validierung und Analyse von **Echtzeitsystemen** geben Schaufelberger et al., Quirk und Zöbel (s. [SSW 85], [Qui 85], [Zöb 87]). Die besonderen Probleme (aber auch Vorteile) von Echtzeit-, verteilten und eingebetteten Systemen beschreiben Ghezzi et al. im Kapitel 2.3 von [GJM 91]. Auf Probleme beim Testen von Echtzeitsystemen weisen Bologna/Leveson hin (s. [BoL 86]). Eine Architektur zum nahezu interferenzfreien Testen von Echtzeit- und verteilten Systemen stellen Fang/Chang vor (s. [FaC 85]).
Die Darstellung und Überprüfung des Verhaltens von **eingebetteten Systemen** mit reellwertigen Variablen (wie z. B. Zeit, Druck, Temperatur) durch „hybride" Automaten schlagen Alur et al. vor (s. [AHH 96]). Einen Modularisierungsansatz, bei dem kommunizierende **zeitbehaftete Petri-Netze** verwendet werden, stellen Bucci/Vicario in [BuV 95] vor. Modelle, Sprachen und Techniken zur Beschreibung von Echtzeitanforderungen finden sich bei [BoL 86] und [KeG 92]. Eine logische Notation für Zeit präsentieren Halbwachs et al. (s. [HLR 92]). Die **temporale Logik** mit den Operatoren *Futur*, *Past* (und weiteren) stammt von Felder/Morzenti (s. [FMM 94]). Genaueres zum Einsatz objektorientierter Techniken findet sich bei Morzenti/SanPietro (s. [MoS 91]). Hinweise zum **Integrations- und Systemtest** von Echtzeitsystemen gab Head schon 1964 (s. [Hea 64]). Eine Studie über Fehler in Echtzeitsystemen wurde von Perry/Stieg durchgeführt (s. [PeS 93]).
Die Darstellung der partiellen zeitlichen Ordnung von Ereignissen in **verteilten Systemen** durch eine **Präzedenzrelation** stammt von Chen/Yeh (s. [ChY 83]). Die neuen Probleme bei verteilten Systemen beschreibt Haban (s. [Hab 86]). Das Problem des Beobachtungseffekts bei nebenläufigen Systemen untersucht Gait (s. [Gai 86]), eine Lösung mit Hilfe eines Mechanismus zur Aufzeichnung der In-

terprozeßkommunikation und eines daraus konstruierten Zeitabhängigkeitsgraphen (*timing graph*) geben Gordon/Finkel an (s. [GoF 86]). Eine Lösung des **Testreproduktionsproblems** durch lokale **Monitore** präsentiert Fidge (s. [Fid 89], S. 184 f.). Vorschläge zur Lösung des Informationsmengenproblems machen Garcia-Molina et al. (s. [GGK 84], S. 214 f.). Die **Testkriterien** für verteilte Systeme stammen von Chang et al. (s. [CSW 90], S. 114 f.).
Überblicke über das Testen und Korrigieren nebenläufiger Programme und über die formale Spezifikation und Verifikation von Echtzeitsystemen geben McDowell/Helmbold und Ghezzi et al. (s. [McH 89], [GFB 93]). Spezielle Probleme beim Testen und Verifizieren von **Protokollen** behandelt der Tagungsband [DeS 95].