# Vergleich der Nebenläufigkeitskonzepte von Ada, CHILL, Erlang und Java

**Diplomarbeit**

zur Erlangung des akademischen Grades Diplom-Informatiker


Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
Institut für Informatik
Lehrstuhl für Programmiersprachen und Compiler

eingereicht von Peter Brömel
geb. am 25.07.1973 in Eisenach

Betreuer: Prof. Dr. Jürgen Winkler

Jena, 12.04.1999

# Zusammenfassung

Ada, CHILL, Erlang und Java sind Programmiersprachen, die das Erstellen nebenläufiger Programme unterstützen. Jede Sprache benutzt hierbei ihren eigenen Ansatz. Um die Nebenläufigkeitskonzepte der Sprachen zu vergleichen, wurden Aufgabenstellungen gewählt, die in jeder der Sprachen realisiert werden mußten. Diese Aufgabenstellungen enthalten typische Probleme der Nebenläufigkeit, wie z.B. synchrone Kommunikation, koordinierter Zugriff auf gemeinsame Variablen und bedingte Synchronisation.

Ada erwies sich als am besten geeignet, um die vorliegenden Aufgaben zu lösen. Der Grund ist, daß Ada die größte Funktionalität bezüglich der Nebenläufigkeit bietet, während den anderen Sprachen einige Konstrukte, die für die Lösung der Aufgabenstellungen nützlich erscheinen, fehlen. So ist Ada die einzige Sprache, die ein Sprachmittel für die synchrone Kommunikation zwischen Akteuren zur Verfügung stellt. Außerdem sind die Sprachkonstrukte in Ada teilweise besser als die entsprechenden Konstrukte der anderen Sprachen. Das trifft vor allem auf das Problem des koordinierten Zugriffs auf gemeinsame Variablen in Zusammenhang mit bedingter Synchronisation zu (Leser-Schreiber-Problem, Verwaltung der Blockabschnitte bei der Kabinenbahn).

CHILL, Erlang und Java haben generell den Nachteil, daß nicht für jede Anforderung ein entsprechendes Konstrukt zur Verfügung steht. Zwar lassen sich die Aufgaben auch damit lösen, jedoch sind die entstandenen Lösungen teilweise umständlich und unübersichtlich.

# Inhaltsverzeichnis

# Vorwort

Im Wintersemester 1997/98 hielt Herr Prof. Dr. Jürgen Winkler erstmals die Vorlesung "Nebenläufige und parallele Programmierung". Durch diese Vorlesung wurde mein Interesse für die Nebenläufigkeit geweckt und deshalb beschloß ich, mich weitergehend mit diesem Gebiet zu beschäftigen. Als Ergebnis entstand — in Zusammenarbeit mit Herrn Frank Ecke — eine Studienarbeit [BE 98], die die Nebenläufigkeitskonzepte der Sprachen Ada, CHILL und Java miteinander vergleicht.

Die nun vorliegende Arbeit vertieft — unter Hinzunahme der Sprache Erlang — diesen Vergleich durch die Realisierung von typischen Aufgabenstellungen. Dazu werden in einem einführenden Kapitel zunächst grundlegende Definitionen der Kriterien gegeben, anhand derer später der Vergleich erfolgen soll. Jede der zu implementierenden Aufgabenstellungen wird innerhalb eines eigenen Kapitels behandelt. Ein solches Kapitel beginnt mit der genauen Beschreibung der Aufgabe. Danach werden Überlegungen zum Programmentwurf dargestellt. Der sich anschließende Programmtext wird — soweit das erforderlich ist — erläutert. Jedes Programm wird anhand der definierten Kriterien genau untersucht; am Ende eines jeden Kapitels werden die jeweiligen Lösungen gegenübergestellt und miteinander verglichen. Schließlich erfolgt in einem letzten Kapitel eine Gesamtzusammenfassung der Ergebnisse.

Ich möchte mich ganz herzlich bei Herrn Prof. Dr. Jürgen Winkler für seine zahlreichen Anregungen und Hinweise bedanken. Des weiteren gilt mein Dank den Herren Michael Hartmeier und Dr. Stefan Kauer, die die Lesbarkeit der Programmtexte beurteilten.

Außerdem danke ich der Siemens AG, die uns einen Rechner mit der VISION O.N.E CHILL Support Software zur Verfügung stellte und mir persönlich einen Aufenthalt in München — zum Erlernen des Umgangs mit dem CHILL System — ermöglichte. Insbesondere gilt hierbei mein Dank den Herren Thomas Burgard, Jonas Höf, Rainer von Mellenthin, Thomas Schiewer, Erwin Stambera und Richard Sturm, die mir bei verschiedenen Fragen und Problemen immer hilfreich zur Seite standen.

Jena, April 1999 Peter Brömel

# Kapitel 0

# Einleitung und allgemeine Definitionen

## 0.1 Einleitung

Das Ziel dieser Arbeit ist es, aufbauend auf [BE 98] den Vergleich der Nebenläufigkeitskonzepte von Ada, CHILL und Java — und zusätzlich noch Erlang — anhand folgender Aufgabenstellungen weiterzuführen:

1. Der nebenläufige Quicksort-Algorithmus
2. Rückmeldung bei asynchroner Kommunikation
3. Leser-Schreiber-Problem mit Priorität für Leser und Priorität für Schreiber
4. Kabinenbahn
5. Bakery-Algorithmus

Jedes dieser Probleme wird in einem gesonderten Kapitel behandelt. Innerhalb eines solchen Kapitels gehen wir nach folgenden Schritten vor:

- Aufgabenstellung
  Das eigentliche Problem wird hier genau spezifiziert.
- Programmentwurf
  Hier werden Überlegungen zur Lösung der Aufgabe in der jeweiligen Sprache gemacht.
- Programm
  Der jeweilige Quelltext wird aufgeführt und die wesentlichen Teile des Programms werden erläutert.
- Untersuchung der Programmlösung

## 0.2 Definitionen

Um für jedes Problem die verschiedenen Lösungen vergleichen zu können, sind Kriterien erforderlich, die eine objektive Einschätzung der Programme erlauben. Deshalb werden im Folgenden Definitionen und Erläuterungen der zu verwendenden Kriterien gegeben:

**Lesbarkeit** Lesbarkeit ist ein Maß, das angibt, wie leicht oder schwer es ist, ein als Quelltext gegebenes Programm zu verstehen. Die Lesbarkeit eines Programms hängt natürlich stark davon ab, wie vertraut man mit der Programmiersprache ist, in der das Programm formuliert wurde.

Um trotzdem möglichst objektiv zu bleiben, wählen wir folgende Vorgehensweise:

Es werden mehrere Testpersonen ausgewählt, die die vier Sprachen etwa gleich gut beherrschen. Diese Testpersonen beurteilen die Lesbarkeit der einzelnen Programme und vergeben für jedes Programm eine Wertung. Diese Wertung ist eine Zahl zwischen 1 und 5. Dabei bedeutet

1 = sehr gut lesbar
2 = gut lesbar
3 = durchschnittlich lesbar
4 = schlecht lesbar
5 = sehr schlecht lesbar.

**Programmgröße** Die Programmgröße messen wir in nloc (netto lines of code). Dabei steht eine Anweisung in einer Zeile; Kommentare und Leerzeilen werden nicht gezählt.

**Kommunikationsaufwand** Der Kommunikationsaufwand ist ein Maß für den Aufwand, der erforderlich ist, um durch explizites Blockieren und Reaktivieren von Akteuren Synchronisation zwischen diesen Akteuren zu erreichen. Dies wird gemessen über die dynamische Anzahl von Anweisungen, die dafür notwendig sind.

**Nebenläufigkeitsgrad** Sei $M = \{a_1, a_2, ..., a_n\}$ die Menge der Akteure eines Programms. Als Nebenläufigkeitsgrad definieren wir die durchschnittliche Anzahl von Akteuren aus $M$, die ausgeführt werden können. Wir unterscheiden zwei Arten von Nebenläufigkeitsgrad, den theoretischen und den tatsächlichen:

- Der theoretische Nebenläufigkeitsgrad ergibt sich nur aus dem Algorithmus, d.h. er ist unabhängig von Maschinenbeschränkungen.
- Der tatsächliche Nebenläufigkeitsgrad einer Programmlösung dagegen ist abhängig von den Beschränkungen der jeweiligen Sprache, in der das Programm formuliert wurde.

Somit stellt der theoretische Nebenläufigkeitsgrad eine obere Schranke für den tatsächlichen Nebenläufigkeitsgrad dar.

**Speedup** Der Speedup $S(p, n)$ berechnet sich nach der Vorschrift

$$S(p,n) = \frac{T(1,n)}{T(p,n)} = \frac{Rechenzeit\,des\,Algorithmus\,mit\,einem\,Prozessor}{Rechenzeit\,des\,Algorithmus\,mit\,p\,Prozessoren}$$

Der Speedup wird als Kriterium nur für den nebenläufigen Quicksort-Algorithmus eingesetzt, da es sich bei dieser Aufgabe um ein reines Parallelisierungsproblem handelt, bei dem auf disjunkten Daten gearbeitet wird.

Die nachfolgenden Kriterien Lebendigkeit, Verklemmung, Aushungern und faire Maschine werden nur für zyklische Programme verwendet. Ein zyklisches Programm P sei ein Programm, in dem gewisse Aktionen $ea_1, ea_2, ..., ea_n$ immer wieder — d.h. unendlich oft — ausgeführt werden sollen. Der Einfachheit halber wird im folgenden *eine* solche ausgezeichnete Aktion pro Akteur angenommen, d.h. P enthält n Akteure $a_1, a_2, ..., a_n$. Diese Aktionen werden als erwünschte Aktionen ($ea$) bezeichnet.

Endliche Ausführungen von P sind in diesem Zusammenhang uninteressant. Es wird daher angenommen, daß P nur unendliche Ausführungen hat. Dabei hat auch jeder der Akteure in P jeweils eine unendliche Ausführung, da jeder Akteur eine erwünschte Aktion enthält.
Es gibt nun zwei Arten von (unendlichen) Ausführungen:

- erwünschte: alle $ea_i$ kommen unendlich oft vor
- unerwünschte: mindestens ein $ea_i$ kommt nur endlich oft vor

Damit ist es nun möglich, die weiteren Kriterien zu definieren:

**Lebendigkeit** Eine Ausführung A von P ist lebendig in Bezug auf einen Akteur $a_i \Longleftrightarrow ea_i$ kommt unendlich oft in A vor. P ist lebendig in Bezug auf einen Akteur $a_i \Longleftrightarrow$ alle Ausführungen A von P sind lebendig in Bezug auf $a_i$. P ist lebendig $\Longleftrightarrow$ P ist lebendig in Bezug auf alle Akteure in P.

**Verklemmung** Für ein Programm P gibt es drei Stufen von Verklemmung:

1. verklemmungsfrei: es gibt nur erwünschte Ausführungen
2. teilweise verklemmt: es gibt sowohl erwünschte als auch unerwünschte Ausführungen
3. total verklemmt: es gibt nur unerwünschte Ausführungen

**Aushungern** Aushungern eines Akteurs $a_i$ in einer Ausführung A von P bedeutet, daß $ea_i$ nur endlich oft in A vorkommt.

**Faire Maschine** Eine Maschine ist fair, wenn sie garantiert, daß während jeder Ausführung A von P kein ausführungsbereiter Akteur $a_i$ unendlich oft übergangen wird.

Am Ende eines jeden Kapitels werden die Ergebnisse der Untersuchungen der einzelnen Sprachen noch einmal zusammengefaßt und vergleichend gegenübergestellt.

Des weiteren sind alle aufgeführten Programmtexte separat auf einer beiliegenden CD-ROM enthalten. Alle Programme wurden übersetzt und ausgeführt. Dabei wurden folgende Übersetzer verwendet:

- Ada: GNU Ada Translator, Version 3.10p
- Java: Java Development Kit, Version 1.1.5
- Erlang: Erlang System/OTP R4B, Version 4.7.3
- CHILL: VISION O.N.E. CHILL Support Software, Toolset TL13U

Eine Ausnahme gilt für CHILL. Da das zur Verfügung stehende System (VISION O.N.E.) einen Dialekt der Sprache CHILL unterstützt, konnten die hier abgedruckten CHILL96-Quellprogramme nicht übersetzt werden und somit sind syntaktische Fehler nicht auszuschließen.

Die auf der CD-ROM enthaltenen CHILL-Programme wurden in VISION O.N.E. CHILL formuliert. Im Anhang A sind diese Quellprogramme abgebildet.

# Kapitel 1

# Der nebenläufige Quicksort-Algorithmus

## 1.1 Aufgabenstellung

Die Aufgabe besteht darin, den Quicksort-Algorithmus in den vier Sprachen so zu formulieren, daß er nebenläufig ausgeführt werden kann.

Um das Problem besser zu verstehen, geben wir zunächst eine kurze Erläuterung des Quicksort-Algorithmus.

### 1.1.1 Der Quicksort-Algorithmus

Der Quicksort-Algorithmus ist ein Verfahren zur Sortierung einer Eingabefolge mit n Elementen. Sein Name begründet sich auf der Tatsache, daß Quicksort eines der schnellsten Sortierverfahren ist. Trotz einer Laufzeit von $\Theta(n^2)$ im schlechtesten Fall beträgt die durchschnittliche Laufzeit nur $\Theta(n \log n)$. Ein weiterer Vorteil dieses Verfahrens ist, daß während des Sortierens kein zusätzlicher Speicher benötigt wird (abgesehen von einer konstanten Zahl von Speicherplätzen für Tauschoperationen), d.h. Umordnungen finden innerhalb der Eingabefolge statt; am Ende des Algorithmus enthält die ursprüngliche Eingabefolge die sortierte Folge.

Quicksort basiert auf der Teile-und-Herrsche-Strategie. Wie in [Cor 94] dargestellt, läuft das Verfahren in folgenden Schritten ab:

**Teile:** Die Eingabefolge $A[p..r]$ wird derart in 2 nichtleere Teilfolgen $A[p..q]$ und $A[q+1..r]$ partitioniert, daß jedes Element aus $A[p..q]$ kleiner oder gleich jedem Element aus $A[q+1..r]$ ist. Der Index $q$ wird während der Partitionierungsprozedur berechnet.

**Herrsche:** Die 2 Teilfolgen $A[p..q]$ und $A[q+1..r]$ werden durch rekursive Aufrufe von Quicksort sortiert.

**Zusammensetzen:** Da die Teilfolgen sortiert sind (Sortieren findet innerhalb der Eingabefolge statt), ist kein weiteres Zusammensetzen der Teilfolgen notwendig; die ursprüngliche Eingabefolge $A[p..r]$ ist nun sortiert.

Der aus [Cor 94] entnommene Pseudocode des Quicksort-Algorithmus ist in Abbildung 1.1 gegeben.

```
QUICKSORT(A,p,r)
 1  if p < r
 2     then q ← PARTITION(A,p,r)
 3          QUICKSORT(A,p,q)
 4          QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
```

```
1  x ← A[p]
2  i ← p - 1
3  j ← r + 1
4  while TRUE
5     do repeat j ← j - 1 until A[j] ≤ x
6        repeat i ← i + 1 until A[i] ≥ x
7        if i < j
8          then vertausche A[i] ↔ A[j]
9          else return j
```

**Abbildung 1.1** Der Quicksort-Algorithmus

## 1.2 Der nebenläufige Quicksort-Algorithmus in Ada

### 1.2.1 Programmentwurf

Betrachtet man den in Abbildung 1.1 dargestellten Pseudocode, so ergibt sich die Idee, jeden der rekursiven Aufrufe mittels eines neuen Task auszuführen. Dazu ist ein Tasktyp innerhalb der Quicksort-Prozedur erforderlich. Da Tasks dynamisch gestartet werden sollen (immer dann, wenn ein neuer rekursiver Aufruf erfolgt), ist des weiteren ein Verweistyp auf den Tasktyp notwendig. Die Aufgabe der dynamisch gestarteten Tasks besteht darin, die Quicksort-Prozedur mit den neuen Parametern des rekursiven Aufrufs auszuführen. Dabei manipulieren alle Tasks dieselbe Array-Variable; der Zugriff erfolgt über einen Verweis.

### 1.2.2 Programm

```
package Quicksort is
   type Vektor is array (Integer range <>) of Integer;
   type Zeiger_Auf_Vektor is access Vektor;
   procedure Nebenläufiges_Quicksort (Folge : in Zeiger_Auf_Vektor;
                                      Von, Bis : in Integer);
end Quicksort;

package body Quicksort is
   procedure Partitionieren (Folge : in out Vektor;
                  Von, Bis : in Integer; Pivot_Index : out Integer) is
      Pivot_Element : Integer := Folge(Von);
      Unterer_Index : Integer := Von;
      Oberer_Index : Integer := Bis;
      Temp : Integer := 0;
   begin
      Bestimme_Index:
      loop
         while Folge(Oberer_Index) > Pivot_Element loop
            Oberer_Index := Oberer_Index - 1;
         end loop;
         while Folge(Unterer_Index) < Pivot_Element loop
            Unterer_Index := Unterer_Index + 1;
         end loop;

         if Unterer_Index < Oberer_Index then
```

```
            Temp := Folge(Unterer_Index);
            Folge(Unterer_Index) := Folge(Oberer_Index);
            Folge(Oberer_Index) := Temp;
         else
            exit Bestimme_Index;
         end if;
         Oberer_Index := Oberer_Index - 1;
         Unterer_Index := Unterer_Index + 1;
      end loop Bestimme_Index;
      Pivot_Index := Oberer_Index;
   end Partitionieren;

   task type Sortiere_Teilfolge is
      entry Sortieren (Folge : in Zeiger_Auf_Vektor; Von, Bis : in Integer);
   end Sortiere_Teilfolge;
   task body Sortiere_Teilfolge is
      Speicher : Zeiger_Auf_Vektor;
      Index1, Index2 : Integer;
   begin
      accept Sortieren (Folge : in Zeiger_Auf_Vektor; Von, Bis : in Integer)
      do
         Speicher := Folge;
         Index1 := Von;
         Index2 := Bis;
      end Sortieren;
      Nebenläufiges_Quicksort (Speicher, Index1, Index2);
   end Sortiere_Teilfolge;

   type Zeiger_Auf_Sortiere_Teilfolge is access Sortiere_Teilfolge;

   procedure Nebenläufiges_Quicksort (Folge : in Zeiger_Auf_Vektor;
                                      Von, Bis : in Integer) is
      Sortiere_Linke_Teilfolge,
      Sortiere_Rechte_Teilfolge : Zeiger_Auf_Sortiere_Teilfolge;
      Pivot_Index : Integer;
   begin
      if Von < Bis then
         Partitionieren (Folge.all, Von, Bis, Pivot_Index);
         Sortiere_Linke_Teilfolge := new Sortiere_Teilfolge;
         Sortiere_Linke_Teilfolge.Sortieren (Folge, Von, Pivot_Index);
         Sortiere_Rechte_Teilfolge := new Sortiere_Teilfolge;
         Sortiere_Rechte_Teilfolge.Sortieren (Folge, Pivot_Index+1, Bis);
      end if;
   end Nebenläufiges_Quicksort;
end Quicksort;
```

**Abbildung 1.2** Der nebenläufige Quicksort-Algorithmus in Ada

Der Rumpf der Prozedur `Nebenläufiges_Quicksort` entspricht der Prozedur `QUICKSORT` aus Abbildung 1.1. Dabei erfolgt jeder der beiden rekursiven Aufrufe immer in zwei Schritten:

1. Aktivieren eines neuen Task

2. Aufruf des entry `Sortieren` des aktivierten Tasks; dabei erhält `Sortieren` die Parameter für den rekursiven Aufruf.

Innerhalb des Rumpfes des neu gestarteten Tasks sieht der Ablauf folgendermaßen aus:

1. Im `accept` Rumpf des entry `Sortieren` werden die Parameter in lokale Variablen gespeichert.

2. Nach Beendigung des `accept` Rumpfes erfolgt der Aufruf von `Nebenläufiges_Quicksort` mit den lokalen Variablen als Parameter.

Der Grund, weshalb wir den Aufruf von `Nebenläufiges_Quicksort` nicht innerhalb des `accept` Rumpfes des entry `Sortieren` ausführen (um uns den Typ `Zeiger_Auf_Vektor` zu sparen), liegt im synchronen Ablauf des Rendezvous in Ada. Würden wir innerhalb des entry `Sortieren` die Prozedur `Nebenläufiges_Quicksort` aufrufen, so hätte dies zur Folge, daß jeder Task, der `Sortieren` aufruft, so lange blockiert wäre, bis die Anweisungsfolge im `accept` Rumpf des entry abgearbeitet ist. Tatsächlich wäre diese Lösung gar nicht nebenläufig, sondern rein sequentiell. Um das zu umgehen, beschränken wir die Anweisungsfolge im `accept` Rumpf auf die Übergabe der Parameter. Der eigentliche Aufruf findet erst nach Beendigung des `accept` Rumpfes statt.

In [Geh 91] ist eine andere Lösung beschrieben, die ohne Zeiger arbeitet und deshalb einen zweiten entry zum Rückschreiben des Ergebnisses benötigt.

### 1.2.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.2 vergeben.

#### Programmgröße

Das Programm hat eine Größe von 62 nloc.

#### Kommunikationsaufwand

Es ist kein Kommunikationsaufwand notwendig.

#### Speedup

Wir gehen davon aus, daß so viele Prozessoren zur Verfügung stehen, daß jeder Akteur auf einem eigenen Prozessor ausgeführt werden kann. Das bedeutet, es muß zunächst bestimmt werden, wieviel Akteure maximal zu einem Zeitpunkt ausgeführt werden können. Betrachten wir dazu Abbildung 1.3, in der eine mögliche Aufrufhierarchie dargestellt ist. Dabei ist die Anzahl der aktiven Tasks abhängig von der Tiefe, in der man sich befindet. Für die in Abbildung 1.3 dargestellte gleichmäßige Partitionierung beträgt $A_T(k)$ — die Anzahl der aktiven Tasks in der Aufruftiefe $k$ — $A_T(k) = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

Für eine andere Partitionierung kann das aber völlig anders aussehen. Um einen allgemeinen Speedup anzugeben, müßte man alle möglichen Partitionierungen betrachten und einen Mittelwert über diese bilden. Da das aber über das Ziel dieser Arbeit hinausgeht, werden wir uns auf ein Beispiel beschränken. Mit Hilfe eines Zufallsgenerators wurde eine Folge von 10 Zahlen erzeugt: $(63, 93, 67, 64, 13, 37, 61, 4, 3, 41)$. Für diese Folge zeigt Abbildung 1.4 die Aufrufhierarchie. Für die Bestimmung des Speedup gehen wir von folgenden Vereinbarungen aus:

- ein Zeitpunkt entspricht einer Tiefe im Baum; dabei ist $t_k$ derjenige Zeitpunkt, der der Tiefe k entspricht

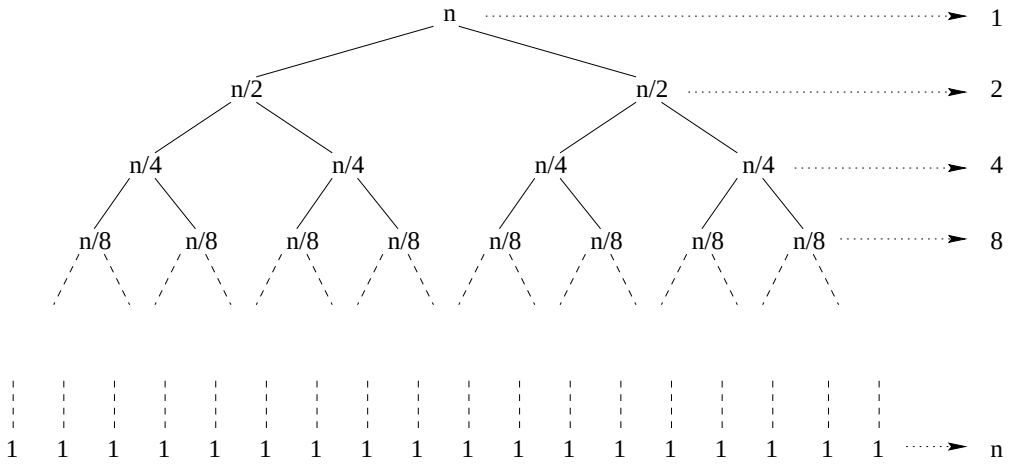


**Abbildung 1.3** Aufrufhierarchie bei gleichmäßiger Partitionierung

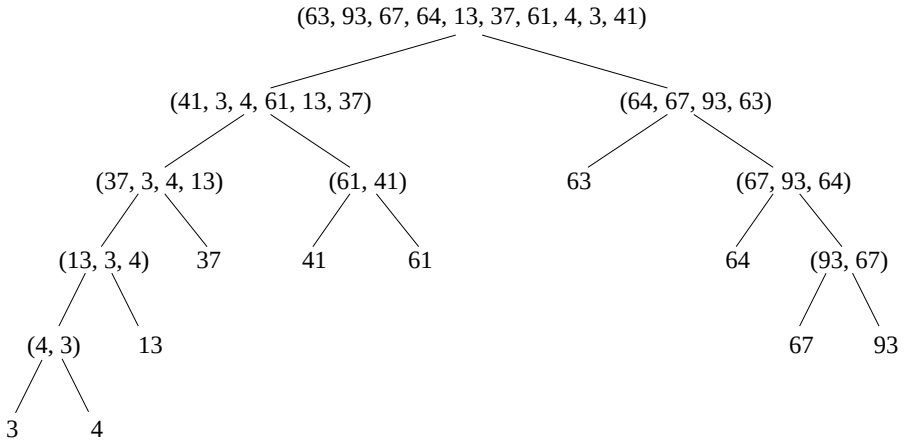


**Abbildung 1.4** Aufrufhierarchie der Beispielfolge

- jeder Knoten einer gegebenen Tiefe k repräsentiert einen Task, der zum Zeitpunkt $t_k$ aktiv ist
- ist ein Task zu einem Zeitpunkt $t_k$ aktiv, so sind auch alle seine Vorgänger zum Zeitpunkt $t_k$ aktiv
- ein Knoten, der zu einem Zeitpunkt $t_k$ eine Folge der Länge 1 enthält, ist zum Zeitpunkt $t_{k+1}$ nicht mehr aktiv

Damit ergibt sich die folgende Tabelle:

| Zeitpunkt | Anzahl aktiver Tasks |
|---|---|
| $t_0$ | 1 |
| $t_1$ | 3 |
| $t_2$ | 7 |
| $t_3$ | 12 |
| $t_4$ | 11 |
| $t_5$ | 7 |

Da maximal 12 Akteure gleichzeitig ausgeführt werden können, benötigen wir 12 Prozessoren. Mit den 6 Zeitpunkten $t_0..t_5$ gilt $T(12, 10) = 6$. Da der in Abbildung 1.4 dargestellte Baum 19 Knoten

hat, erhalten wir $T(1,10) = 19$. Damit ergibt sich für den Speedup folgender Wert:

$$S(12,10) = \frac{T(1,10)}{T(12,10)} = \frac{19}{6} = 3.17.$$

**Nebenläufigkeitsgrad**

Der Nebenläufigkeitsgrad beträgt $\frac{1+3+7+12+11+7}{6} = \frac{41}{6} = 6.83$. Das entspricht dem theoretischen Nebenläufigkeitsgrad für diese Ausführung.

## 1.3 Der nebenläufige Quicksort-Algorithmus in CHILL

### 1.3.1 Programmentwurf

Auch hier gehen wir wieder von dem Ansatz aus, für jeden der rekursiven Aufrufe einen neuen nebenläufigen Akteur zu starten. In diesem Fall wird das jeweils ein CHILL Prozeß sein. Ein solcher Prozeß wird mit den Parametern des rekursiven Aufrufs gestartet und ruft damit die eigentliche Quicksort-Prozedur auf. Dabei muß aber folgendes beachtet werden:

Ein Block, der einen Prozeß startet, wartet nicht am Ende des Blockes auf die Beendigung des gestarteten Prozesses. Das bedeutet, die Prozedur, die den nebenläufigen Quicksort-Algorithmus ausführt, könnte enden, obwohl das Feld noch von aktiven Prozessen modifiziert wird.

Dieses Problem lösen wir durch Setzen von Synchronisationspunkten mit Hilfe von Puffern.

### 1.3.2 Programm

```
quicksort:
MODULE
   GRANT nebenlaeufiges_quicksort;
   NEWMODE
      nachricht = INT(1:1),
      puffer_typ = BUFFER(1) nachricht,
      index = INT(1:100),
      feld_typ = ARRAY(index) INT;

   sortiere_teilfolge:
   PROCESS (puffer REF puffer_typ, folge REF feld_typ, von index, bis index);
      nebenlaeufiges_quicksort (folge, von, bis);
      SEND puffer-> (1);
   END sortiere_teilfolge;

   nebenlaeufiges_quicksort:
   PROC (folge REF feld_typ, von INT, bis INT);
      DCL puffer puffer_typ,
          temp nachricht,
          pivot_index INT;
      IF von < bis
         THEN
            partitionieren (folge->, von, bis, pivot_index);
            START sortiere_teilfolge (->puffer, folge, von, pivot_index);
            START sortiere_teilfolge (->puffer, folge, pivot_index+1, bis);
            RECEIVE (puffer IN temp);
```

```
            RECEIVE (puffer IN temp);
      FI;
   END nebenlaeufiges_quicksort;

   partitionieren:
   PROC (folge feld_typ INOUT, von INT, bis INT, pivot_index INT OUT);
      DCL pivot_element INT := folge(von),
          unterer_index INT := von,
          oberer_index INT := bis,
          temp INT := 0;
      bestimme_index:
      DO FOR EVER;
         DO WHILE folge(oberer_index) > pivot_element;
            oberer_index := oberer_index - 1;
         OD;
         DO WHILE folge(unterer_index) < pivot_element;
            unterer_index := unterer_index + 1;
         OD;
         IF unterer_index < oberer_index
            THEN
               temp := folge(unterer_index);
               folge(unterer_index) := folge(oberer_index);
               folge(oberer_index) := temp;
            ELSE
               pivot_index := oberer_index;
               EXIT bestimme_index;
         FI;
         unterer_index := unterer_index + 1;
         oberer_index := oberer_index - 1;
      OD;
   END partitionieren;
END quicksort;
```

**Abbildung 1.5** Der nebenläufige Quicksort-Algorithmus in CHILL

Der Puffer `puffer` dient der Synchronisation zwischen `nebenläufiges_quicksort` und den in dieser Prozedur gestarteten zwei Prozessen.

Ein rekursiver Aufruf innerhalb von `nebenlaeufiges_quicksort` ist in folgenden zwei Schritten organisiert:

1. Erzeugen und Starten zweier neuer Prozeßinstanzen von `sortiere_teilfolge`. Dabei werden die Parameter für den rekursiven Aufruf und eine Referenz auf `puffer` übergeben.

2. Empfangen von zwei Nachrichten aus `puffer`.

Innerhalb einer Prozeßinstanz erfolgt der Aufruf von `nebenlaeufiges_quicksort` mit den an die Instanz übergebenen Parametern. Danach wird eine Nachricht an `puffer` gesendet und damit das Ende der Prozeßinstanz angezeigt.

Diese explizite Synchronisation stellt sicher, daß eine Prozeßinstanz P erst enden kann, nachdem alle von P gestarteten Prozeßinstanzen beendet sind.

### 1.3.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.6 vergeben.

#### Programmgröße

Das Programm hat eine Größe von 54 nloc.

#### Kommunikationsaufwand

Es muß für jeden rekursiven Aufruf — also für jeden neu gestarteten Prozeß P — ein Synchronisationspunkt gesetzt werden. Dazu sind zwei Anweisungen notwendig, ein `SEND` am Ende von P und ein `RECEIVE` am Ende des Blockes, der P startet.

Somit ergibt sich für die Beispielfolge ein Kommunikationsaufwand von $19 * 2 = 38$ Anweisungen.

#### Speedup

Analog zur Ada-Lösung beträgt der Speedup für die Beispielfolge 3.17.

#### Nebenläufigkeitsgrad

Analog zur Ada-Lösung beträgt der Nebenläufigkeitsgrad für die Beispielfolge 6.83.

## 1.4 Der nebenläufige Quicksort-Algorithmus in Erlang

### 1.4.1 Programmentwurf

Auch in Erlang besteht unser Ansatz darin, jeden der rekursiven Aufrufe innerhalb eines neuen Akteurs zu starten. Hier muß aber die Parameterübergabe mittels Senden und Empfangen von Nachrichten erfolgen. Da nur Prozesse in der Lage sind, Nachrichten zu senden und zu empfangen, muß auch die Quicksort-Funktion als Prozeß gestartet werden. Das bewerkstelligen wir mittels einer zweiten Funktion. Das Partitionieren wird in einer dritten Funktion erledigt. Da es in Erlang keine Arrays gibt, benutzen wir statt dessen Listen. Dabei bedeutet Partitionieren, daß die Listen zerlegt und wieder neu zusammengesetzt werden.

### 1.4.2 Programm

```
-module(quicksort).
-export([start/1, nebenlaeufiges_quicksort/1]).

start (L) ->
   Pid = spawn(quicksort, nebenlaeufiges_quicksort, [L]),
   Pid ! self(),
   receive
      {Pid, Liste}  -> Liste
   end.

nebenlaeufiges_quicksort([]) ->
   receive
      Pid -> true
   end,
```

```
      Pid ! {self(), []};
   nebenlaeufiges_quicksort([Pivot|Rest]) ->
      receive
         Pid -> true
      end,
      {Linke_Teilfolge, Rechte_Teilfolge} = partitionieren(Pivot, Rest),
      Sortiere_Linke_Teilfolge =
            spawn(quicksort, nebenlaeufiges_quicksort, [Linke_Teilfolge]),
      Sortiere_Rechte_Teilfolge =
            spawn(quicksort, nebenlaeufiges_quicksort, [Rechte_Teilfolge]),
      Sortiere_Linke_Teilfolge ! self(),
      Sortiere_Rechte_Teilfolge ! self(),
      receive
         {Sortiere_Linke_Teilfolge, L1} -> true
      end,
      receive
         {Sortiere_Rechte_Teilfolge, L2} -> true
      end,
      Pid ! {self(), lists:append (L1, [Pivot|L2])}.

   partitionieren(Pivot, L) ->
      partitionieren(Pivot, L, [], []).
   partitionieren(Pivot, [], Linke_Teilfolge, Rechte_Teilfolge) ->
      {Linke_Teilfolge, Rechte_Teilfolge};
   partitionieren(Pivot, [H|T], Linke_Teilfolge, Rechte_Teilfolge)
   when H < Pivot ->
      partitionieren(Pivot, T, [H|Linke_Teilfolge], Rechte_Teilfolge);
   partitionieren(Pivot, [H|T], Linke_Teilfolge, Rechte_Teilfolge)
   when H >= Pivot ->
      partitionieren(Pivot, T, Linke_Teilfolge, [H|Rechte_Teilfolge]).
```

**Abbildung 1.6** Der nebenläufige Quicksort-Algorithmus in Erlang

Der Modul `quicksort` enthält die folgenden Funktionen: `start`, `nebenlaeufiges_quicksort` und `partitionieren`. Die Funktion `start` bekommt als Parameter eine Liste `L` und führt folgende Schritte aus:

1. Es wird ein neuer Prozeß P gestartet, der `nebenlaeufiges_quicksort(L)` ausführt. Die Identifikationsnummer von P wird in `Pid` gespeichert.

2. Dann sendet `start` seine eigene Identifikation an P. Das ist notwendig, damit P weiß, an wen er sein Ergebnis zurücksenden soll.

3. Danach wartet `start` auf eine Nachricht von P. Diese Nachricht enthält die sortierte Liste.

Abbildung 1.7 veranschaulicht diesen Ablauf.

In der Funktion `nebenlaeufiges_quicksort` wird als erstes auf eine Nachricht, die die Identifikation des rufenden Prozesses RP enthält, gewartet. Der weitere Ablauf hängt davon ab, ob die Liste L, die `nebenlaeufiges_quicksort` als Parameter bekommen hat, leer ist oder nicht:

1. Ist L die leere Liste, so wird an RP eine Nachricht, die aus der eigenen Identifikation und der leeren Liste besteht, gesendet. Abbildung 1.8 stellt das dar.

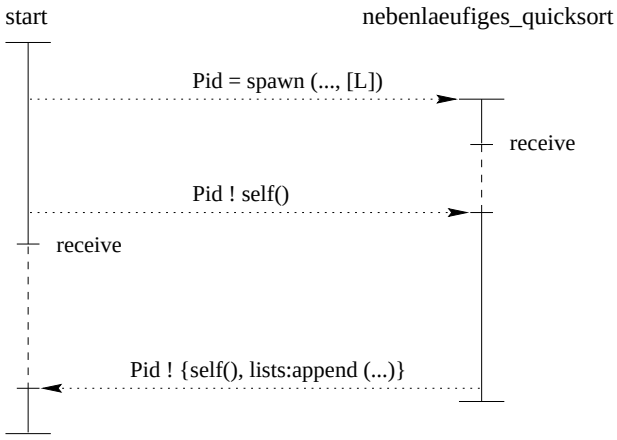


**Abbildung 1.7** Start des Programms

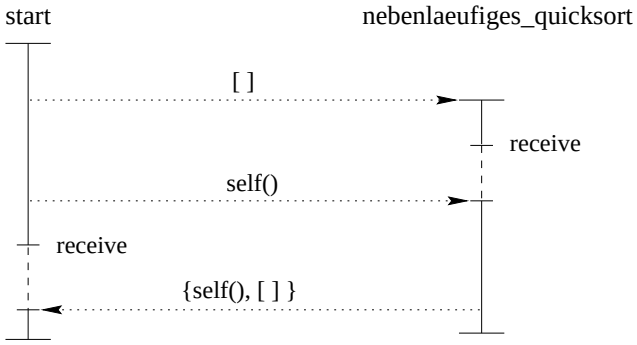


**Abbildung 1.8** Aufruf mit leerer Liste

2. Ist L nicht leer, wird `partitionieren` aufgerufen und liefert die zwei Listen `Linke_Teilfolge` und `Rechte_Teilfolge` zurück. Dann werden zwei Prozesse erzeugt, die die beiden rekursiven Aufrufe ausführen. Diesen Prozessen wird die eigene Identifikation gesendet und dann wird auf eine Nachricht von diesen Prozessen gewartet. Diese Nachricht enthält jeweils das Ergebnis des rekursiven Aufrufs, also eine sortierte Teilliste. Danach werden die beiden sortierten Teillisten mit dem Pivotelement zur sortierten Gesamtliste zusammengesetzt und an RP gesendet. Dieser Ablauf ist in Abbildung 1.9 dargestellt. Dabei wird auch deutlich, daß diese Lösung mit einem sehr hohen Kopieraufwand verbunden ist.

### 1.4.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.5 vergeben.

#### Programmgröße

Das Programm hat eine Größe von 37 nloc.

#### Kommunikationsaufwand

Es müssen für jeden rekursiven Aufruf — also für jeden neu gestarteten Prozeß P — zwei Nachrichten ausgetauscht werden; zum einen damit P die Identifikation desjenigen Prozesses S, der P startet, erfährt und zum anderen damit P an S das Ergebnis zurücksendet. Das heißt, für jeden Prozeß sind zwei Sendeoperationen (`!`) und zwei `receive` Anweisungen erforderlich.

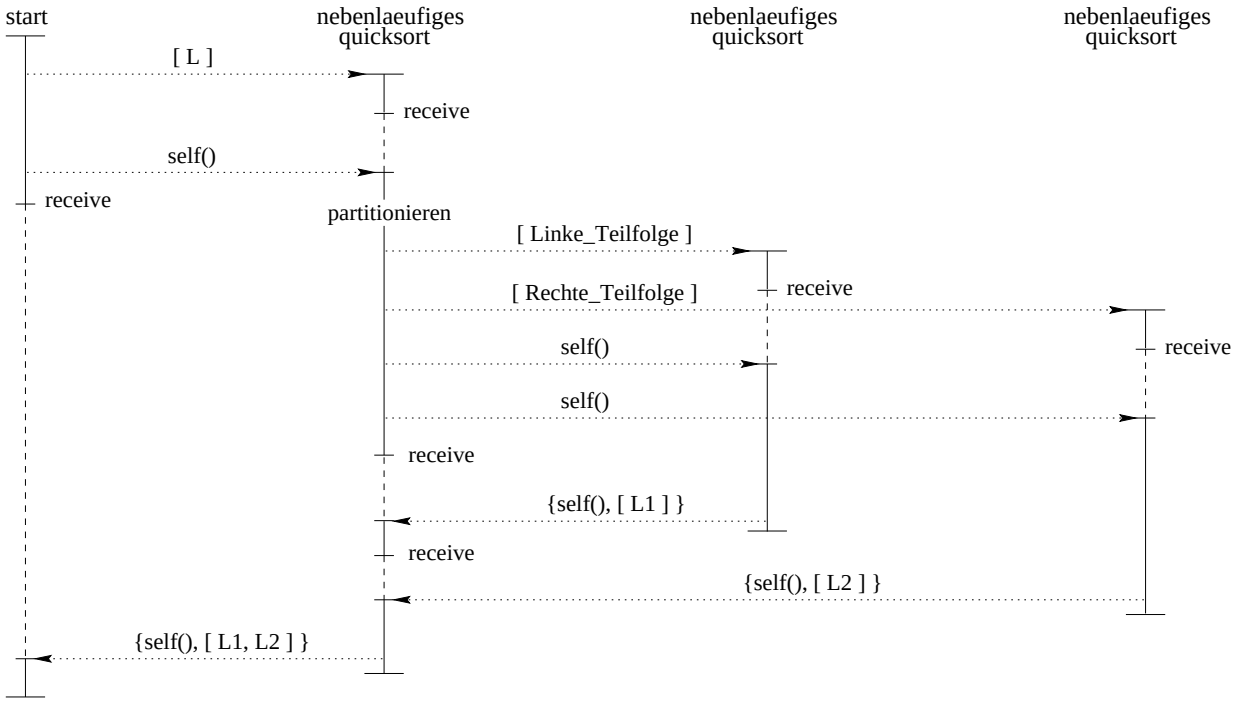


**Abbildung 1.9** Aufruf mit nichtleerer Liste

Somit beträgt der Kommunikationsaufwand für die Beispielfolge $4 * 19 = 76$ Anweisungen.

#### Speedup

Analog zu den beiden vorigen Lösungen beträgt der Speedup 3.17.

#### Nebenläufigkeitsgrad

Analog zu den vorigen Lösungen beträgt der Nebenläufigkeitsgrad 6.83.

## 1.5 Der nebenläufige Quicksort-Algorithmus in Java

### 1.5.1 Programmentwurf

Analog zu den anderen Lösungen werden wir auch in Java versuchen, jeden der rekursiven Aufrufe innerhalb eines neuen Akteurs — in diesem Fall ein Java Thread — zu starten. Da der Aktionsteil eines Java Thread nichts anderes ist als seine `run()` Methode, muß die `run()` Methode der zu kreierenden Thread-Klasse den Aufruf der Quicksort-Prozedur beinhalten.

### 1.5.2 Programm

```
class quicksort {
   private static int partitionieren (int folge[], int von, int bis) {
      int pivot_element = folge[von];
      int unterer_index = von;
      int oberer_index = bis;
      int temp = 0;
      while (true) {
```

```
            while (folge[oberer_index] > pivot_element)
               oberer_index--;
            while (folge[unterer_index] < pivot_element)
               unterer_index++;
            if (unterer_index < oberer_index) {
               temp = folge[unterer_index];
               folge[unterer_index] = folge[oberer_index];
               folge[oberer_index] = temp;
            }
            else
               break;
            oberer_index--;
            unterer_index++;
         }
         return oberer_index;
      }
      public static void nebenläufiges_quicksort(int folge[], int von, int bis) {
         if (von < bis) {
            int pivot_index = partitionieren (folge, von, bis);
            sortiere_teilfolge sortiere_linke_teilfolge =
                  new sortiere_teilfolge (folge, von, pivot_index);
            sortiere_linke_teilfolge.start();
            sortiere_teilfolge sortiere_rechte_teilfolge =
                  new sortiere_teilfolge (folge, pivot_index+1, bis);
            sortiere_rechte_teilfolge.start();
            try {
               sortiere_linke_teilfolge.join();
               sortiere_rechte_teilfolge.join();
            }
            catch (InterruptedException e) {}
         }
      }
   }
   class sortiere_teilfolge extends Thread {
      int folge[], von, bis;
      sortiere_teilfolge (int folge[], int von, int bis) {
         this.folge = folge;
         this.von = von;
         this.bis = bis;
      }
      public void run() {
         quicksort.nebenläufiges_quicksort (folge, von, bis);
      }
   }
```

**Abbildung 1.10** Der nebenläufige Quicksort-Algorithmus in Java

Die Methode `nebenläufiges_quicksort` implementiert `QUICKSORT` aus Abbildung 1.1. Die Besonderheit liegt dabei im rekursiven Aufruf, der in folgenden zwei Schritten abläuft:

1. Definieren eines Objektes der Klasse `sortiere_teilfolge` und Übergabe der Parameter (für den rekursiven Aufruf) mittels des Konstruktors der Klasse `sortiere_teilfolge`.

2. Starten des dem erzeugten Objekt zugehörigen Thread; d.h. Ausführen der Methode `run()` der Klasse `sortiere_teilfolge`.

Die Klasse `sortiere_teilfolge` ist von der Klasse `Thread` abgeleitet und somit gehört zu jedem Objekt der Klasse `sortiere_teilfolge` ein Java Thread, der die Methode `run()` ausführt. Die Methode `run()` enthält den rekursiven Aufruf zu `nebenläufiges_quicksort`.

Die beiden `join()` Anweisungen sind notwendig, da in Java ein Block, der einen Thread startet, nicht an seinem Ende auf die Beendigung des Threads wartet. Das heißt, ohne die beiden `join()` Anweisungen könnte `nebenläufiges_quicksort` enden, obwohl noch Threads aktiv sind, die das Feld verändern. Um das zu verhindern, zwingen wir `nebenläufiges_quicksort` mit Hilfe der beiden `join()` Anweisungen, auf das Ende der gestarteten Threads zu warten.

Ein anderer Lösungsansatz ist in Abbildung 1.11 gegeben. Im Unterschied zur Lösung aus Abbildung 1.10 wird nur noch eine Klasse verwendet, deren `run()` Methode der Prozedur `QUICKSORT` aus Abbildung 1.1 entspricht. Des weiteren wurde das Starten der neu erzeugten Threads in den Konstruktor verlegt.

```
class quicksort extends Thread {
   private int folge[], von, bis;
   quicksort (int folge[], int von, int bis) {
      this.folge = folge;
      this.von = von;
      this.bis = bis;
      this.start();
   }
   public void run() {
      if (von < bis) {
          int pivot_index = partitionieren (folge, von, bis);
          quicksort sortiere_linke_teilfolge =
                new quicksort (folge, von, pivot_index);
          quicksort sortiere_rechte_teilfolge =
                new quicksort (folge, pivot_index+1, bis);
          try {
              sortiere_linke_teilfolge.join();
              sortiere_rechte_teilfolge.join();
          }
          catch (InterruptedException e) {}
      }
   }
   private static int partitionieren (int folge[], int von, int bis) {
      int pivot_element = folge[von];
      int unterer_index = von;
      int oberer_index = bis;
      int temp = 0;
      while (true) {
         for (int i=oberer_index; i>=von; i--) {
            if (folge[i] <= pivot_element) {
               oberer_index = i;
               break;
            }
         }
         for (int i=unterer_index; i<=bis; i++) {
```

```
            if (folge[i] >= pivot_element) {
               unterer_index = i;
               break;
            }
         }
         if (unterer_index < oberer_index) {
            temp = folge[unterer_index];
            folge[unterer_index] = folge[oberer_index];
            folge[oberer_index] = temp;
         }
         else
            break;
         oberer_index--;
         unterer_index++;
      }
      return oberer_index;
   }
}
```

**Abbildung 1.11** Java-Lösung mit nur einer Klasse

### 1.5.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.5 vergeben.

#### Programmgröße

Das Programm hat eine Größe von 49 nloc bzw. 51 nloc.

#### Kommunikationsaufwand

Für jeden rekursiven Aufruf — also für jeden gestarteten Thread — ist eine `join()` Anweisung notwendig.

Somit beträgt der Kommunikationsaufwand für die Beispielfolge 19 Anweisungen.

#### Speedup

Analog zu den vorherigen Lösungen beträgt der Speedup 3.17.

#### Nebenläufigkeitsgrad

Analog zu den vorherigen Lösungen beträgt der Nebenläufigkeitsgrad 6.83.

## 1.6 Zusammenfassung und Vergleich

Der nebenläufige Quicksort-Algorithmus konnte in allen vier Sprachen ohne größere Schwierigkeiten implementiert werden. Die entstandenen Lösungen weisen Unterschiede in der Lesbarkeit, der Programmgröße und im Kommunikationsaufwand auf. Dabei sind die Unterschiede in der Lesbarkeit

gering, während die Unterschiede bezüglich der Programmgröße und des zusätzlichen Kommunikationsaufwandes signifikant sind. Das in Ada verwendete Warten am Blockende macht Kommunikationsaufwand überflüssig; dagegen muß in den anderen Sprachen das Warten am Blockende explizit erzwungen werden, was mit einem erhöhten Kommunikationsaufwand verbunden ist.

Tabelle 1.1 faßt die Ergebnisse noch einmal zusammen.

| **Verwendete Sprache** | Ada | CHILL | Erlang | Java |
|---|---|---|---|---|
| Lesbarkeit | 2.2 | 2.6 | 2.5 | 2.5 |
| Programmgröße | 62 nloc | 54 nloc | 37 nloc | 49 nloc / 51 nloc |
| Kommunikationsaufwand | - | 38 | 76 | 19 |
| Speedup | 3.17 | 3.17 | 3.17 | 3.17 |
| Nebenläufigkeitsgrad | 6.83 | 6.83 | 6.83 | 6.83 |

**Tabelle 1.1** Gegenüberstellung der verschiedenen Lösungen

Auf der beiliegenden CD-ROM sind die in diesem Kapitel abgebildeten Quelltexte im Verzeichnis `broemel\Quicksort` in folgenden Dateien enthalten:

Ada: `quicksort.ads`, `quicksort.adb`

CHILL: `quickspa.src`, `qsort00s.src`, `qproz00s.src`, `qsema00s.src`, `qtest00s.src`

Erlang: `quicksort.erl`

Java: `quicksort1.java`, `quicksort2.java`

# Kapitel 2

# Rückmeldung bei asynchroner Kommunikation

## 2.1 Aufgabenstellung

Wenn mehrere Akteure miteinander kommunizieren, so sind immer Synchronisationspunkte im Ablauf der Kommunikation notwendig, an denen Daten ausgetauscht werden können. Eine häufig genutzte Methode ist in Abbildung 2.1 dargestellt.

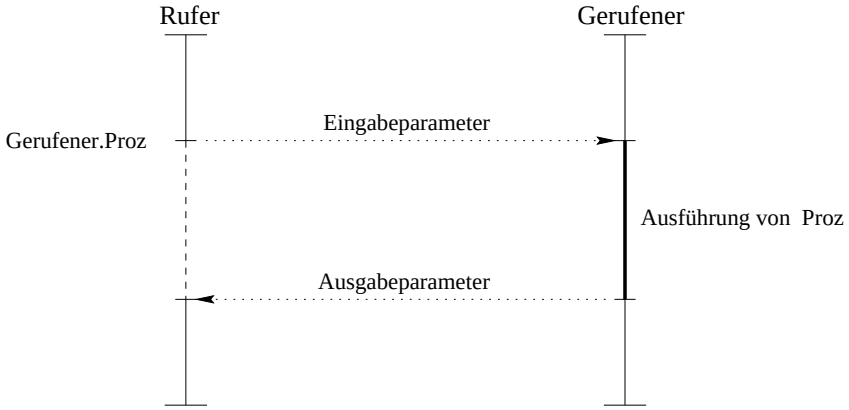


**Abbildung 2.1** Synchrone Kommunikation

Der Akteur Rufer, der die Ausführung der Prozedur Proz des Akteurs Gerufener anstoßen will, übergibt an einem Synchronisationspunkt dem Gerufenen die Eingabedaten und wartet daraufhin so lange, bis der Gerufene fertig ist und die Ausgabedaten übergibt. Nach einem solchen Schema läuft z.B. das Rendezvous in Ada ab. Der Nachteil dieser Vorgehensweise ist, daß der Rufer während der Ausführung von Proz blockiert ist.

Um diesen Nachteil zu umgehen, gibt es einen anderen Ansatz — die asynchrone Kommunikation. Wie in Abbildung 2.2 dargestellt, führt der Akteur Rufer seinen Aufruf Gerufener.Proz aus und arbeitet sofort weiter — ohne auf die Beendigung der Prozedur Proz zu warten. Der Vorteil dieses Vorgehens ist, daß Wartesituationen — wie sie beim synchronen Fall auftreten — vermieden werden. Beide Akteure können unabhängig voneinander weiterarbeiten. Wie in [BE 98] beschrieben, tritt diese Vorgehensweise in CHILL auf, wenn eine öffentliche Prozedur eines CHILL Task aufgerufen wird.

Dabei ergibt sich nun die folgende wichtige Frage: Wie kann der Rufer auf die Ergebnisse der vom Gerufenen ausgeführten Prozedur zugreifen?

Der Beantwortung dieser Frage werden wir uns in diesem Kapitel widmen. Da die direkte Kommunikation zwischen Akteuren in Ada synchron verläuft und in Java nicht möglich ist, werden

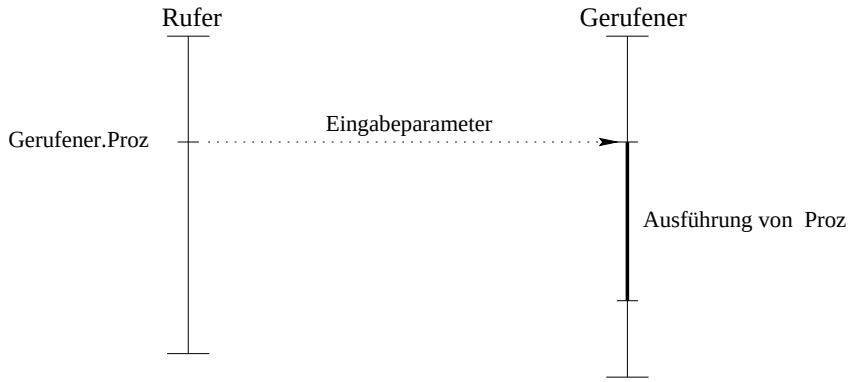


**Abbildung 2.2** Asynchrone Kommunikation

hier nur CHILL und Erlang betrachtet.

## 2.2 Rückmeldung bei asynchroner Kommunikation in CHILL

### 2.2.1 Programmentwurf

In CHILL lösen wir das Problem, indem der Rufer beim Aufruf der Prozedur Proz des Gerufenen einen zusätzlichen Parameter übergibt: Eine Referenz auf einen Puffer. Der Ablauf ist dann so, wie in Abbildung 2.3 dargestellt.

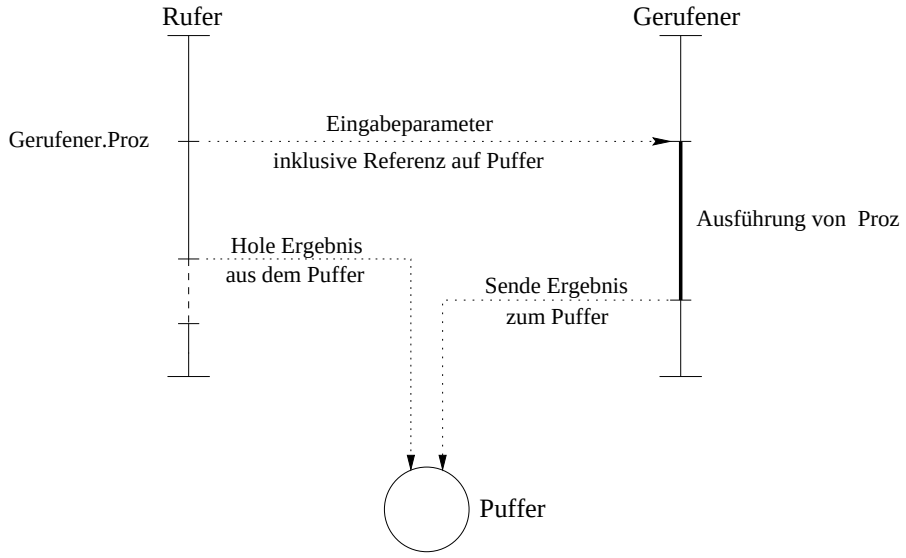


**Abbildung 2.3** Rückmeldung über Puffer

Nachdem der Rufer die Parameter (inklusive der Referenz auf den Puffer) übergeben hat, arbeitet er zunächst weiter. Sobald er aber an einer Stelle angelangt ist, an der er das Ergebnis von Proz benötigt, versucht er, dieses Ergebnis aus dem Puffer zu holen. Sollte der Puffer noch leer sein, so muß der Rufer warten, bis das Ergebnis an den Puffer gesendet worden ist. Das Senden des Ergebnisses zum Puffer erfolgt am Ende von Proz.

### 2.2.2 Programm

```
SYNMODE gerufener_task_typ = TASK SPEC
   SEIZE puffer_typ;
```

```
   GRANT proz;
   proz:
   PROC (puffer REF puffer_typ, argument INT);
END gerufener_task_typ;

SYNMODE gerufener_task_typ = TASK BODY
   proz:
   PROC (puffer REF puffer_typ, argument INT);
      DCL ergebnis INT;

      /* Ausfuehren einer Berechnung und
         Speichern der Ergebnisse in ergebnis */

      SEND puffer-> (ergebnis);
   END proz;
END gerufener_task_typ;

rufer:
MODULE
   GRANT puffer_typ;
   SEIZE gerufener_task_typ;
   NEWMODE puffer_typ = BUFFER (1) INT;
   DCL gerufener gerufener_task_typ,
       mein_puffer puffer_typ,
       ergebnis INT,
       argument INT;

   argument := 100;
   gerufener.proz (->mein_puffer, argument);

   /* Ausfuehren einer Folge von Anweisungen, bis irgendwann das
      Ergebnis von proz benoetigt wird */

   RECEIVE (mein_puffer IN ergebnis);
END rufer;
```

**Abbildung 2.4** Rückmeldung über einen Puffer

Der Ablauf erfolgt so, wie in Abbildung 2.3 dargestellt. Nach dem Aufruf der Prozedur `proz` wartet `rufer` nicht auf die Beendigung dieser Prozedur, sondern führt die nächsten Anweisungen aus (Der Einfachheit wegen haben wir das im Kommentar nur angedeutet). Auf der anderen Seite beginnt `gerufener` mit der Ausführung der Prozedur `proz`. Als letzte Anweisung innerhalb von `proz` wird das Ergebnis der durchgeführten Berechnung mittels einer `SEND` Anweisung in den Puffer `puffer` geschickt. Sobald `rufer` an einer Stelle angekommen ist, an der das Ergebnis von `proz` benötigt wird, führt `rufer` eine `RECEIVE` Anweisung aus, um sich den Wert aus dem Puffer zu holen. Sollte der Wert noch nicht im Puffer sein, wird `rufer` so lange blockiert, bis der Wert empfangen werden kann.

### 2.2.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.5 vergeben.

#### Programmgröße

Die Programmgröße beträgt 26 nloc.

#### Kommunikationsaufwand

Es ist ein `SEND` und ein `RECEIVE` für die Rückmeldung erforderlich. Das heißt, der Kommunikationsaufwand für eine Ausführung beträgt 2 Anweisungen.

## 2.3 Rückmeldung bei asynchroner Kommunikation in Erlang

### 2.3.1 Programmentwurf

In Erlang wird Rückmeldung bei asynchroner Kommunikation durch Senden und Empfangen von Nachrichten erreicht. Betrachten wir dazu Abbildung 2.5.

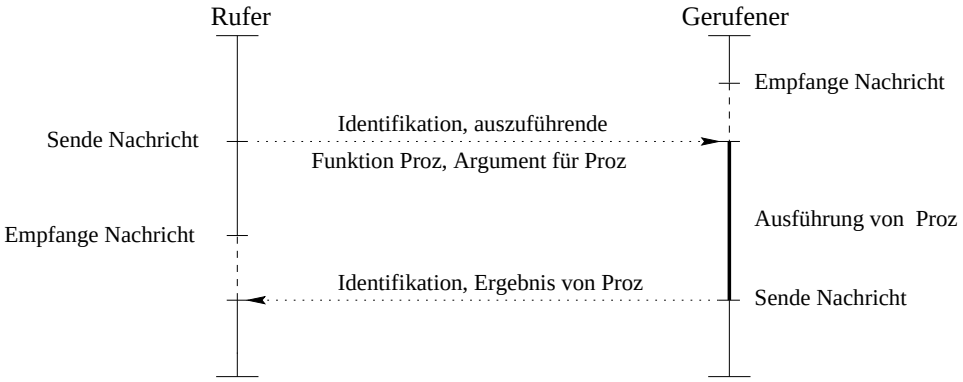


**Abbildung 2.5** Asynchrone Kommunikation in Erlang

Der Rufer sendet dem Gerufenen, der die Funktion Proz ausführen soll, eine Nachricht. Diese Nachricht enthält die Identifikation des Rufers (damit der Gerufene weiß, an wen er das Ergebnis zurückschicken soll), den Namen der auszuführenden Funktion Proz und die Parameter für den Aufruf von Proz. Nachdem der Rufer die Nachricht abgeschickt hat, arbeitet er normal weiter. Sobald er aber das Ergebnis von Proz benötigt, versucht er, eine Nachricht vom Gerufenen zu empfangen. Diese Nachricht vom Gerufenen besteht aus der Identifikation des Gerufenen (damit der Rufer weiß, von wem die Nachricht kommt) und dem Ergebnis von Proz. Wie man sieht, kommt es immer dann zu Wartesituationen, wenn ein Prozeß auf eine Nachricht des anderen Prozesses wartet.

### 2.3.2 Programm

```
-module (asynchrone_Kommunikation).
-export([start/0, rufer/0, gerufener/0]).

start () ->
    register (gerufener, spawn (asynchrone_Kommunikation, gerufener, [])),
```

```
    spawn (asynchrone_Kommunikation, rufer, []).

rufer () ->
    gerufener ! {self(), {proz, 10}},
    % Ausfuehren einer Folge von Anweisungen, bis
    % irgendwann das Ergebnis von proz benoetigt wird
    receive
        {gerufener, Ergebnis} ->
            Ergebnis
    end.

gerufener () ->
    receive
        {Von, {proz, Argument}} ->
            Ergebnis = proz (Argument),
            Von ! {gerufener, Ergebnis}
    end.

proz (Argument) ->
    2*Argument.
```

**Abbildung 2.6** Rückmeldung mittels Nachrichten in Erlang

Der Prozeß `rufer` sendet an den Prozeß `gerufener` eine Nachricht bestehend aus der eigenen Identifikation, der auszuführenden Funktion `proz` und dem Argument für `proz`. Nach dem Senden der Nachricht wartet `rufer` auf die Antwort von `gerufener`. Der Prozeß `gerufener` wartet auf eine Nachricht bestehend aus der Variablen `Von`, dem Atom `proz` und der Variablen `Argument`. Dabei wird `Von` als Identifikation eines Prozesses und `Argument` als Argument für die Funktion `proz` interpretiert. Nachdem `gerufener` die Nachricht empfangen hat, führt er `proz(Argument)` aus. Das Ergebnis von `proz(Argument)` wird an den Prozeß mit der Identifikation `Von` zurückgesendet.

### 2.3.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.5 vergeben.

#### Programmgröße

Die Programmgröße beträgt 19 nloc.

#### Kommunikationsaufwand

Es ist eine Sendeoperation (`!`) und ein `receive` für die Rückmeldung erforderlich. Das heißt, der Kommunikationsaufwand für eine Ausführung beträgt 2 Anweisungen.

## 2.4 Zusammenfassung und Vergleich

In beiden Sprachen konnte das Problem der Rückmeldung bei asynchroner Kommunikation ohne Schwierigkeiten gelöst werden. In CHILL wird die Rückmeldung über einen zwischengeschalteten Puffer erreicht (indirekte Kommunikation), während Erlang das Senden und Empfangen von Nachrichten (direkte Kommunikation) verwendet. Der Kommunikationsaufwand ist in beiden Sprachen

gleich, es sind jeweils zwei Anweisungen für die Rückmeldung erforderlich. Bezüglich der Lesbarkeit gibt es keine Unterschiede; die Programmgröße ist in Erlang geringer.

Tabelle 2.1 faßt die Ergebnisse noch einmal zusammen.

| **Verwendete Sprache** | CHILL | Erlang |
|---|---|---|
| Lesbarkeit | 2.5 | 2.5 |
| Programmgröße | 26 | 19 |
| Kommunikationsaufwand | 2 | 2 |

**Tabelle 2.1** Gegenüberstellung der verschiedenen Lösungen

Auf der beiliegenden CD-ROM ist der abgebildete Erlang-Quellcode im Verzeichnis `broemel\asynchrone_Kommunikation` in der Datei `asynchrone_kommunikation.erl` enthalten.

Da VISION O.N.E. keine Tasks gemäß CHILL96 unterstützt, können hier keine Quelldateien angegeben werden.

# Kapitel 3

# Das Leser-Schreiber-Problem

## 3.1 Das Leser-Schreiber-Problem mit Priorität für Leser

### 3.1.1 Aufgabenstellung

Gegeben sei eine Ressource R, auf die von beliebig vielen Akteuren lesend oder schreibend zugegriffen wird. Akteure, die R lesen, werden Leser genannt; Akteure, die R schreiben, heißen Schreiber. Der Zugriff auf R durch einen Akteur A erfolgt innerhalb der Lese- bzw. Schreiboperation. Für den Zugriff auf R gelten folgende Regeln:

1. Schreibzugriffe sind exklusiv. Während eine Schreiboperation ausgeführt wird, darf keine andere Operation — egal ob Lesen oder Schreiben — ausgeführt werden.

2. Lesezugriffe sind konkurrierend. Das heißt, wenn keine Schreiboperation ausgeführt wird, dürfen beliebig viele Leseoperationen ausgeführt werden.

3. Leser sollen so schnell wie möglich Zugriff zu R erhalten und dürfen dabei wartende Schreiber überholen. Wenn ein oder mehrere Leseoperationen ausgeführt werden, so muß ein ankommender Schreiber S warten. Sollte nun ein weiterer Leser L hinzukommen, so darf L seine Leseoperation ausführen. Damit wurde S von L überholt.

Es wird davon ausgegangen, daß kein Akteur während der Ausführung einer Lese- oder Schreiboperation anhält.

### 3.1.2 Das Leser-Schreiber-Problem mit Priorität für Leser in Ada

#### 3.1.2.1 Programmentwurf

Zur Lösung der Aufgabe wird ein protected object benutzt. Dabei kann aber keine protected function verwendet werden, weil damit die in der Aufgabenstellung formulierten Anforderungen nicht garantiert werden können. Statt dessen werden protected entry und protected procedure benutzt. Da diese aber jeweils exklusiv sind, kann das Lesen der Ressource nicht als protected entry oder protected procedure realisiert werden; denn Lesen soll laut Aufgabenstellung konkurrierend erfolgen können.

Die Lösung besteht in der Verwendung eines Zugangsprotokolls. Hierbei laufen Lesen und Schreiben immer in drei Schritten ab:

1. Belegen

2. Zugriff auf die Ressource

3. Freigeben

Das Belegen erfolgt jeweils mittels eines protected entry, da Belegen nur unter bestimmten Bedingungen — die in der entry barrier überprüft werden — erfolgen darf. Freigeben geschieht mittels protected procedure, da hierbei keine Bedingungen überprüft werden. Innerhalb des protected object werden folgende Größen verwaltet:

- ein Zähler, der angibt, wieviel Tasks gerade eine Leseoperation ausführen
- ein Zähler, der angibt, wieviel Tasks darauf warten, eine Leseoperation auszuführen
- eine Boolesche Variable, die anzeigt, ob ein Task gerade eine Schreiboperation ausführt

Diese Größen dienen der Koordinierung zwischen Lesern und Schreibern. Belegen zum Lesen ist nur möglich, wenn kein Task gerade eine Schreiboperation ausführt; Belegen zum Schreiben ist nur möglich, wenn kein Task gerade eine Lese- oder Schreiboperation ausführt und kein Task auf die Ausführung einer Leseoperation wartet.

#### 3.1.2.2 Programm

```
package Leser_Schreiber1 is
   procedure Lesen (Wert : out Integer);
   procedure Schreiben (Wert : in Integer);
private
   Ressource : Integer := 0;
end Leser_Schreiber1;

package body Leser_Schreiber1 is
   protected Ressource_Sperre is
      entry Start_Lesen;
      procedure Ende_Lesen;
      entry Start_Schreiben;
      procedure Ende_Schreiben;
   private
      Anzahl_Leser : Natural := 0;
      Schreiber_Aktiv : Boolean := False;
   end Ressource_Sperre;

   protected body Ressource_Sperre is
      entry Start_Lesen when not Schreiber_Aktiv is
      begin
         Anzahl_Leser := Anzahl_Leser + 1;
      end Start_Lesen;

      procedure Ende_Lesen is
      begin
         Anzahl_Leser := Anzahl_Leser - 1;
      end Ende_Lesen;

      entry Start_Schreiben when Anzahl_Leser = 0 and Start_Lesen'Count = 0
                                 and not Schreiber_Aktiv is
      begin
         Schreiber_Aktiv := True;
      end Start_Schreiben;
```

```
      procedure Ende_Schreiben is
      begin
         Schreiber_Aktiv := False;
      end Ende_Schreiben;
   end Ressource_Sperre;

   procedure Lesen (Wert : out Integer) is
   begin
      Ressource_Sperre.Start_Lesen;
      Wert := Ressource;
      Ressource_Sperre.Ende_Lesen;
   end Lesen;

   procedure Schreiben (Wert : in Integer) is
   begin
      Ressource_Sperre.Start_Schreiben;
      Ressource := Wert;
      Ressource_Sperre.Ende_Schreiben;
   end Schreiben;
end Leser_Schreiber1;
```

**Abbildung 3.1** Leser-Schreiber-Problem mit Priorität für Leser in Ada

Das protected object `Ressource_Sperre` realisiert das oben beschriebene Zugangsprotokoll. Das Belegen erfolgt mittels der protected entries `Start_Lesen` bzw. `Start_Schreiben` und das Freigeben mit Hilfe der protected procedures `Ende_Lesen` bzw. `Ende_Schreiben`. In `Ressource_Sperre` werden die Variablen `Anzahl_Leser` und `Schreiber_Aktiv` verwaltet, die angeben, wieviel Leseoperationen gerade ausgeführt werden, bzw. ob gerade eine Schreiboperation ausgeführt wird. Für die Anzahl der wartenden Leser ist keine Variable definiert worden, da dies über das `count` Attribut des entry `Start_Lesen` abgefragt werden kann.

#### 3.1.2.3 Untersuchung der Programmlösung

**Lesbarkeit**

Für die Lesbarkeit wurde eine 2 vergeben.

**Programmgröße**

Die Programmgröße beträgt 47 nloc.

**Kommunikationsaufwand**

Es ist kein Kommunikationsaufwand notwendig.

**Lebendigkeit**

Dazu muß man zwei Ebenen unterscheiden:

1. Das Verhalten der Tasks vor Erwerb des dem protected object zugehörigen Locks L.
2. Das Verhalten der Tasks nach Erwerb des dem protected object zugehörigen Locks L.

Ada legt für den Fall, daß mehrere Tasks um L konkurrieren, nicht fest, welcher Task L erhält. Damit kann es passieren, daß ein Task T bei diesem Wettbewerb unendlich oft übergangen wird. Das heißt, auf der 1. Ebene ist das Programm nicht lebendig.

Wenn dagegen T L erworben hat, gibt es beim Belegen folgende Möglichkeiten:

- T durchläuft das Belegen erfolgreich und kann damit auf die Ressource zugreifen.
- T wird blockiert, weil die zugehörige Anmeldebedingung nicht erfüllt ist. In diesem Fall wird T in die Warteschlange des entry eingereiht. Durch die Ordnung innerhalb der Warteschlange (siehe [BE 98]) wird garantiert, daß T nicht übergangen wird.

Beim Freigeben wird keine Bedingung überprüft und somit auch kein Task blockiert. Das bedeutet, auf der 2. Ebene ist das Programm lebendig.

Insgesamt gesehen ist das Programm nicht lebendig.

### Verklemmung

Totale Verklemmung kann nur vorkommen, wenn kein Akteur das Belegen zum Lesen oder Schreiben erfolgreich durchlaufen kann. Dies ist der Fall, wenn `Schreiber_Aktiv` dauerhaft den Wert `True` hat, d.h. wenn ein Schreiber während einer Schreiboperation anhält.

Partielle Verklemmung kann dann vorkommen, wenn mindestens ein Leser während einer Leseoperation anhält (damit wäre `Anzahl_Leser` nie Null und kein Schreiber könnte das Belegen erfolgreich durchlaufen).

Da wir aber davon ausgehen, daß kein Task während einer Lese- oder Schreiboperation anhält, ist das Programm verklemmungsfrei.

### Aushungern

Auf der 1. Ebene ist Aushungern möglich, da Akteure unendlich oft übergangen werden können. Auf der 2. Ebene wird durch die Mechanismen des protected entry (siehe [BE 98] und [In 95]) garantiert, daß kein Akteur übergangen wird.

### Faire Maschine

Für Ebene 1 ist eine faire Maschine erforderlich.

### Nebenläufigkeitsgrad

Für die Betrachtung des Nebenläufigkeitsgrades gehen wir — aus Gründen der Vereinfachung — von folgenden Vereinbarungen aus:

- Es sei $l$ die Anzahl der Leser und $s$ die Anzahl der Schreiber.
- Jeder Leser und jeder Schreiber wird genau einmal ausgeführt.
- Die Ausführung eines Akteurs dauere genau eine Zeiteinheit $t_i$.
- Die Leser werden alle gleichzeitig — d.h. während derselben Zeiteinheit — ausgeführt.

Der letzte Punkt scheint auf den ersten Blick zu restriktiv zu sein. Ohne diese Einschränkung müßten aber alle möglichen Ausführungen mit $l$ Lesern und $s$ Schreibern untersucht werden. Um zu zeigen, wie aufwendig das wäre, betrachten wir die folgenden Überlegungen:

Da die $s$ Schreiber exklusiv ausgeführt werden, kann man $s+1$ Zeitpunkte betrachten (vor dem ersten Schreiber, zwischen je zwei Schreibern, nach dem letzten Schreiber), an denen jeweils eine

Gruppe von Lesern ausgeführt werden kann. Gesucht ist also die Anzahl der Möglichkeiten, um $l$ Leser so auf $s+1$ Zeitpunkte zu verteilen, daß zu einem solchen Zeitpunkt mindestens Null und höchstens $l$ Leser ausgeführt werden. Das läßt sich formal beschreiben als die Anzahl der Lösungen für das folgende Gleichungssystem:

$$x_1 + x_2 + \cdots + x_{s+1} = l \ , \quad 0 \leq x_i \leq l.$$

Wie in [Gri 93] gezeigt, beträgt die Anzahl der Lösungen

$$\left( \begin{array}{c} s+1+l-1 \\ l \end{array} \right) = \left( \begin{array}{c} s+l \\ l \end{array} \right).$$

Es gilt

$$\left( \begin{array}{c} s+l \\ l \end{array} \right) = \frac{(s+l)(s+l-1)(s+l-2)\cdots(s+1)}{l(l-1)(l-2)\cdots 1} \geq \left(\frac{s+l}{l}\right)^l .$$

Für den Fall, daß die Anzahl der Leser gleich der Anzahl der Schreiber ist, ergeben sich bereits mehr als $2^l$ Ausführungen. Tatsächlich ist die Anzahl der möglichen Ausführungen noch viel höher, denn zwischen je zwei Schreibern können nacheinander beliebig viele — und nicht nur eine — Gruppen von Lesern ausgeführt werden. Die Untersuchung aller dieser Ausführungen würde aber, falls überhaupt möglich, den Rahmen dieser Arbeit sprengen. Aus diesem Grund wurde die Einschränkung, daß alle Leser gleichzeitig ausgeführt werden, getroffen.

Da alle Leser gleichzeitig ausgeführt werden, gibt es genau $s+1$ mögliche Ausführungen, denn es existieren $s+1$ Zeiteinheiten, in denen jeweils alle Leser ausgeführt werden können. Der theoretische Nebenläufigkeitsgrad beträgt $\frac{l+s}{s+1}$. Da Leser Priorität haben, werden die Leser — wenn sie während $t_i$ ausgeführt werden — erst während $t_{i-1}$ gestartet.

Zur besseren Veranschaulichung sind 2 Ausführungen in Abbildung 3.2 und Abbildung 3.3 dargestellt. Abbildung 3.2 zeigt den Fall, daß zuerst (während $t_1$) alle $l$ Leser ausgeführt werden und danach (während $t_2, \ldots, t_{s+1}$) je ein Schreiber. In Abbildung 3.3 wird zuerst (während $t_1$) ein Schreiber ausgeführt, danach (während $t_2$) alle $l$ Leser und anschließend (während $t_3, \ldots, t_{s+1}$) je ein Schreiber. Da Leser Priorität haben, werden die Leser bei dieser Ausführung während $t_1$ gestartet.

Die anderen Ausführungen ergeben sich analog. Der Nebenläufigkeitsgrad beträgt in jeder Ausführung $\frac{l+s}{s+1}$. Damit wurde der theoretische Nebenläufigkeitsgrad erreicht.

### 3.1.3 Das Leser-Schreiber-Problem mit Priorität für Leser in CHILL

#### 3.1.3.1 Programmentwurf

Zur Koordinierung des Zugriffs zur Ressource wird in CHILL die `REGION` verwendet, die dem klassischen Monitor entspricht. Da die kritischen Prozeduren einer `REGION` stets unter gegenseitigem Ausschluß ausgeführt werden, kann das Lesen von der Ressource nicht als kritische Prozedur realisiert werden; denn laut Aufgabenstellung soll Lesen konkurrierend erfolgen können.

Analog zur Ada-Lösung wird deshalb ein Zugangsprotokoll verwendet. Das Belegen und Freigeben erfolgt jeweils mittels einer kritischen Prozedur. Außerdem werden noch — wie in der Ada-Lösung — folgende Größen innerhalb der `REGION` verwaltet:

- ein Zähler, der angibt, wieviel Leseoperationen gerade ausgeführt werden
- ein Zähler, der angibt, wieviel Leser auf die Ausführung einer Leseoperation warten
- eine Boolesche Variable, die anzeigt, ob gerade eine Schreiboperation ausgeführt wird

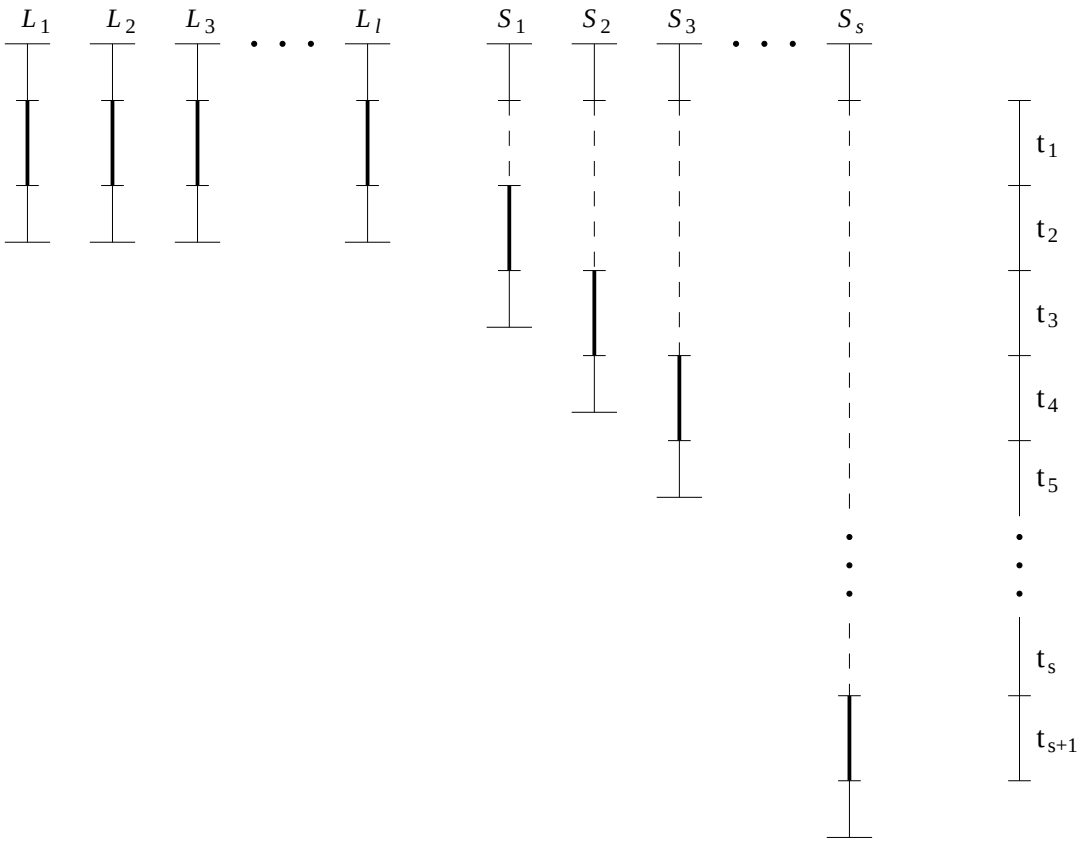


**Abbildung 3.2** Betrachtung des Nebenläufigkeitsgrades, Fall 1

#### 3.1.3.2 Programm

```
leser_schreiber1: REGION
   GRANT start_lesen, ende_lesen,
         start_schreiben, ende_schreiben;
   DCL anzahl_leser INT := 0,
       anzahl_wartende_leser INT := 0,
       schreiber_aktiv BOOL := FALSE,
       kein_schreiber, kein_akteur EVENT;

   start_lesen: PROC ();
      anzahl_wartende_leser := anzahl_wartende_leser + 1;
      DO WHILE schreiber_aktiv;
         DELAY kein_schreiber;
      OD;
      anzahl_wartende_leser := anzahl_wartende_leser - 1;
      anzahl_leser := anzahl_leser + 1;
   END start_lesen;

   ende_lesen: PROC ();
      anzahl_leser := anzahl_leser - 1;
      IF anzahl_leser = 0 AND anzahl_wartende_leser = 0
         THEN CONTINUE kein_akteur;
      FI;
   END ende_lesen;
```

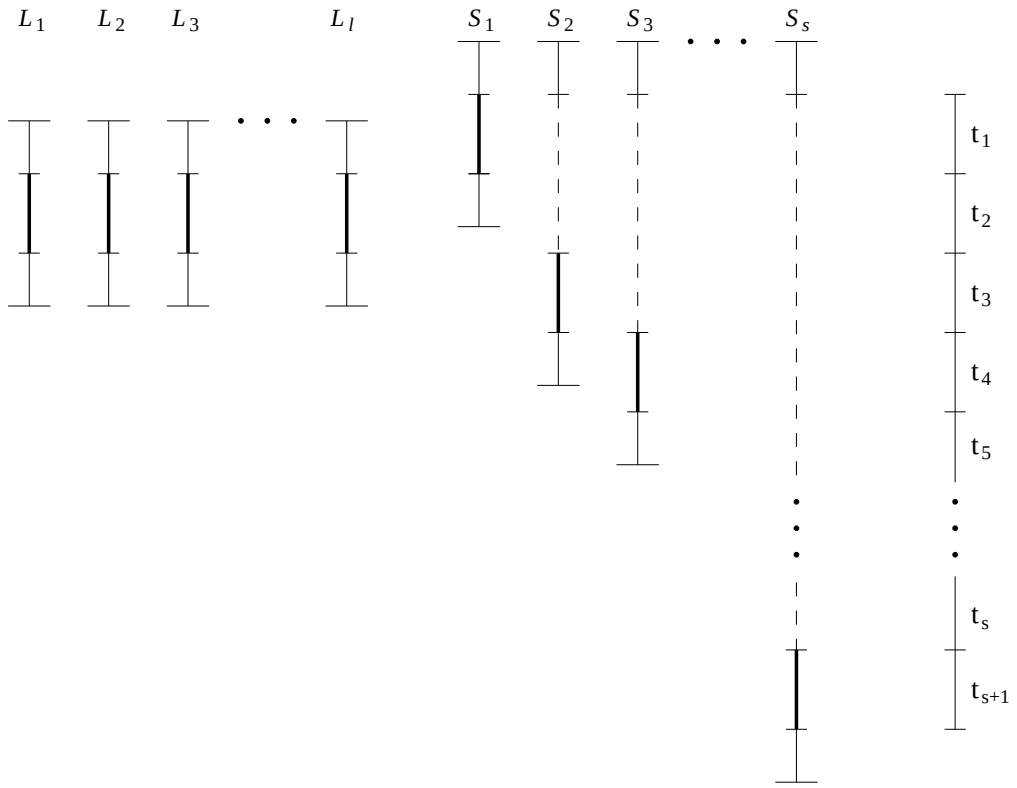


**Abbildung 3.3** Betrachtung des Nebenläufigkeitsgrades, Fall 2

```
   start_schreiben: PROC ();
      DO WHILE anzahl_leser > 0 OR anzahl_wartende_leser > 0 OR
      schreiber_aktiv;
         DELAY kein_akteur;
      OD;
      schreiber_aktiv := TRUE;
   END start_schreiben;

   ende_schreiben: PROC ();
      schreiber_aktiv := FALSE;
      IF anzahl_wartende_leser > 0
      THEN DO FOR i := 1 TO anzahl_wartende_leser;
              CONTINUE kein_schreiber;
           OD;
      ELSE CONTINUE kein_akteur;
      FI;
   END ende_schreiben;
END leser_schreiber1;

leser_schreiber: MODULE
   SEIZE start_lesen, ende_lesen, start_schreiben, ende_schreiben;
   GRANT lesen, schreiben;
   DCL ressource INT := 0;

   lesen: PROC (wert INT OUT);
```

```
      start_lesen();
      wert := ressource;
      ende_lesen();
   END lesen;

   schreiben: PROC (wert INT IN);
      start_schreiben();
      ressource := wert;
      ende_schreiben();
   END schreiben;
END leser_schreiber;
```

**Abbildung 3.4** Leser-Schreiber-Problem mit Priorität für Leser in CHILL

Die `REGION leser_schreiber1` enthält die kritischen Prozeduren, die das Belegen und Freigeben übernehmen, sowie die Variablen `anzahl_leser`, `anzahl_wartende_leser` und `schreiber_aktiv`, die angeben, wieviel Leseoperationen gerade ausgeführt werden, wieviel Leser auf Ausführung einer Leseoperation warten bzw. ob gerade eine Schreiboperation ausgeführt wird. Mit Hilfe der beiden `EVENT` Variablen werden Prozesse, die das Belegen nicht erfolgreich durchlaufen konnten — weil die Belegebedingung nicht erfüllt war —, blockiert und später wieder reaktiviert. Das Reaktivieren von Prozessen geschieht in folgenden Situationen:

- Ein Prozeß L führt `ende_lesen` aus und kein anderer Leser führt gerade eine Leseoperation aus bzw. wartet auf Ausführung einer Leseoperation. Das bedeutet, sobald die Ausführung von `ende_lesen` beendet ist, dürfte ein blockierter Schreiber $S_b$ mit seiner Ausführung fortfahren und wird deshalb reaktiviert. Sollte kein $S_b$ existieren, hat die `CONTINUE` Anweisung keinen Effekt.
- Ein Prozeß S führt `ende_schreiben` aus. Nach Beendigung von `ende_schreiben` dürften entweder alle blockierten Leser $L_b$ oder ein blockierter Schreiber $S_b$ mit ihrer Ausführung fortfahren. Da Leser bevorzugt werden sollen, wird zunächst überprüft, ob ein oder mehrere $L_b$ existieren; falls ja, so werden alle reaktiviert. Sollte kein $L_b$ existieren, so wird ein $S_b$ reaktiviert. Sollte kein $S_b$ existieren, hat die `CONTINUE` Anweisung keinen Effekt.

#### 3.1.3.3 Untersuchung der Programmlösung

**Lesbarkeit**

Für die Lesbarkeit wurde eine 2.5 vergeben.

**Programmgröße**

Das Programm hat eine Größe von 52 nloc.

**Kommunikationsaufwand**

Für den Kommunikationsaufwand beziehen wir uns auf die bei der Betrachtung des Nebenläufigkeitsgrades getroffenen Vereinbarungen. Das bedeutet, es müssen $s + 1$ Ausführungen betrachtet werden (da es $s + 1$ mögliche Zeiteinheiten gibt, in denen jeweils alle $l$ Leser ausgeführt werden).

Tabelle 3.1 zeigt die Ausführung, bei der die Leser während $t_1$ ausgeführt werden (Abbildung 3.2). Dabei werden alle Schreiber blockiert, die Leser werden alle ausgeführt und wecken am Ende von $t_1$ einen Schreiber auf. Dieser Schreiber wird während $t_2$ ausgeführt und weckt am Ende von $t_2$ einen Schreiber auf. Dieser Vorgang läuft so lange, bis alle Schreiber ausgeführt wurden. Der Kommunikationsaufwand beträgt hierbei $2s + 1$.

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | DELAY | CONTINUE | DELAY | CONTINUE |
| $t_1$ | $L_1, ..., L_l$ | - | 1 | s | - |
| $t_2$ | $S_1$ | - | - | - | 1 |
| $t_3$ | $S_2$ | - | - | - | 1 |
| $t_4$ | $S_3$ | - | - | - | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.1** Kommunikationsaufwand, Fall 1

Der Kommunikationsaufwand für den Fall, daß die Leser während $t_2$ ausgeführt werden (Abbildung 3.3), ist in Tabelle 3.2 dargestellt.

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | DELAY | CONTINUE | DELAY | CONTINUE |
| $t_1$ | $S_1$ | - | - | $s-1$ | 1 |
| $t_2$ | $L_1, ..., L_l$ | - | 1 | 1 | - |
| $t_3$ | $S_2$ | - | - | - | 1 |
| $t_4$ | $S_3$ | - | - | - | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.2** Kommunikationsaufwand, Fall 2

Da während $t_1$ ein Schreiber $S_1$ ausgeführt wird, werden alle anderen Schreiber blockiert. Am Ende von $t_1$ wird ein Schreiber aufgeweckt. Dieser reaktivierte Schreiber wird aber wieder blockiert, da die während $t_1$ gestarteten Leser während $t_2$ ausgeführt werden. Am Ende von $t_2$ wird ein Schreiber $S_2$ aufgeweckt und während $t_3$ ausgeführt. Am Ende von $t_3$ weckt $S_2$ einen blockierten Schreiber $S_3$ auf. Das Ausführen und Aufwecken der Schreiber wird so lange fortgeführt, bis alle Schreiber ausgeführt wurden. Der Kommunikationsaufwand beträgt in diesem Fall $2s + 1$.

Fall 3 — alle Leser werden während $t_3$ ausgeführt — ist in Tabelle 3.3 dargestellt. Der Kommunikationsaufwand beträgt auch hierbei $2s + 1$.

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | DELAY | CONTINUE | DELAY | CONTINUE |
| $t_1$ | $S_1$ | - | - | $s-1$ | 1 |
| $t_2$ | $S_2$ | - | - | - | 1 |
| $t_3$ | $L_1, ..., L_l$ | - | 1 | 1 | - |
| $t_4$ | $S_3$ | - | - | - | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.3** Kommunikationsaufwand, Fall 3

Wie man sich leicht überlegen kann, beträgt der Kommunikationsaufwand für die restlichen Fälle ebenfalls je $2s + 1$.

#### Lebendigkeit

Analog zu Ada unterscheiden wir zwei Ebenen.

Wenn mehrere Prozesse um das Lock einer `REGION` konkurrieren, ist nicht festgelegt, welcher Prozeß das Lock erhält. Damit ist es möglich, daß ein Prozeß bei diesem Wettbewerb unendlich oft übergangen wird. Das heißt, auf der 1. Ebene ist das Programm nicht lebendig.

Hat ein Prozeß P das Lock erworben, gibt es beim Belegen folgende Möglichkeiten:

- P durchläuft das Belegen erfolgreich und kann auf die Ressource zugreifen.
- P wird blockiert, weil die Anmeldebedingung nicht erfüllt ist. In diesem Fall kommt P in die Menge der blockierten Prozesse der zugehörigen `EVENT` Variable. Wenn ein Element dieser Menge reaktiviert wird, so ist nicht festgelegt, welches Element ausgewählt wird. Damit ist es möglich, daß P — wenn P nicht der einzige Prozeß in dieser Menge ist — beliebig oft übergangen wird.

Das bedeutet, daß das Programm auch bezüglich der 2. Ebene nicht lebendig ist.

#### Verklemmung

Analog zur Ada-Lösung tritt totale Verklemmung nur dann ein, wenn ein Schreiber während einer Schreiboperation anhält und partielle Verklemmung dann, wenn mindestens ein Leser während einer Leseoperation anhält.

Da wir davon ausgehen, daß kein Prozeß während einer Lese- oder Schreiboperation anhält, ist das Programm verklemmungsfrei.

#### Aushungern

Auf beiden Ebenen ist Aushungern von Prozessen möglich.

#### Faire Maschine

Für beide Ebenen ist eine faire Maschine erforderlich.

#### Nebenläufigkeitsgrad

Analog zur Ada-Lösung beträgt der Nebenläufigkeitsgrad $\frac{l+s}{s+1}$.

### 3.1.4 Das Leser-Schreiber-Problem mit Priorität für Leser in Erlang

#### 3.1.4.1 Programmentwurf

In Erlang existiert nur direkte Kommunikation zwischen Akteuren mittels Senden und Empfangen von Nachrichten. Wie in [AVWW 96] dargelegt, arbeitet das `receive` in Erlang selektiv, d.h. ein Prozeß kann zu einem Zeitpunkt nur eine `receive` Alternative ausführen. Dieser Mechanismus ist vergleichbar mit dem Rendezvous in Ada und genau wie dieses exklusiv.

Um trotzdem konkurrierendes Lesen zu erreichen, wird wieder ein Zugangsprotokoll verwendet. Die richtige Koordinierung zwischen Lesern und Schreibern erfolgt mittels eines Verwaltungsprozesses VP. Innerhalb von VP werden wieder drei Größen verwendet: die Anzahl der aktiven Leser, die Anzahl der wartenden Leser und die Anzahl der aktiven Schreiber.

Da Erlang eine funktionale Sprache ist, werden diese Größen als Parameter von VP verwaltet. Änderungen dieser Werte werden mittels eines neuen Aufrufs von VP mit geänderten Parametern erreicht.

#### 3.1.4.2 Programm

```
-module(leser_schreiber1).
-export([start/0, leser_schreiber1/3, lesen/0, schreiben/1]).

start() ->
   register(leser_schreiber, spawn(leser_schreiber1, leser_schreiber1,
            [0,0,0])).

lesen() ->
   anfrage_lesen(),
   start_lesen(),
   {ok, Binaerobjekt} = file:read_file("ressource"),
   Wert = binary_to_term(Binaerobjekt),
   ende_lesen(),
   Wert.

schreiben(Wert) ->
   start_schreiben(),
   file:write_file("ressource", term_to_binary(Wert)),
   ende_schreiben().

anfrage_lesen() ->
   leser_schreiber ! {self(), anfrage_lesen},
   receive
      anfrage_gestartet ->
         true
   end.

start_lesen() ->
   leser_schreiber ! {self(), start_lesen},
   receive
      lesen_gestartet ->
         true
   end.

ende_lesen() ->
   leser_schreiber ! {self(), ende_lesen},
   receive
      lesen_beendet ->
         true
   end.

start_schreiben() ->
   leser_schreiber ! {self(), start_schreiben},
   receive
      schreiben_gestartet ->
         true
   end.

ende_schreiben() ->
```

```
      leser_schreiber ! {self(), ende_schreiben},
      receive
         schreiben_beendet ->
            true
      end.

   leser_schreiber1 (Anzahl_Leser, Anzahl_Wartende_Leser, Anzahl_Schreiber) ->
      receive
         {PID, anfrage_lesen} ->
            PID ! anfrage_gestartet,
            leser_schreiber1 (Anzahl_Leser, Anzahl_Wartende_Leser+1,
                              Anzahl_Schreiber);

         {PID, start_lesen} when Anzahl_Schreiber==0 ->
            PID ! lesen_gestartet,
            leser_schreiber1 (Anzahl_Leser+1, Anzahl_Wartende_Leser-1,
                              Anzahl_Schreiber);

         {PID, ende_lesen} ->
            PID ! lesen_beendet,
            leser_schreiber1 (Anzahl_Leser-1, Anzahl_Wartende_Leser,
                              Anzahl_Schreiber);

         {PID, start_schreiben} when Anzahl_Leser==0, Anzahl_Wartende_Leser==0,
                                     Anzahl_Schreiber==0 ->
            PID ! schreiben_gestartet,
            leser_schreiber1 (Anzahl_Leser, Anzahl_Wartende_Leser,
                              Anzahl_Schreiber+1);

         {PID, ende_schreiben} ->
            PID ! schreiben_beendet,
            leser_schreiber1 (Anzahl_Leser, Anzahl_Wartende_Leser,
                              Anzahl_Schreiber-1)
      end.
```

**Abbildung 3.5** Leser-Schreiber-Problem mit Priorität für Leser in Erlang

Da in Erlang Variablen nur einmal mit einem Wert belegt werden können (und dann immer an diesen Wert gebunden sind), benutzen wir als Ressource eine Datei, von der gelesen bzw. auf die geschrieben wird.

Das Belegen zum Lesen erfolgt in zwei Schritten: `anfrage_lesen` und `start_lesen`.
Die Funktionen zum Belegen und Freigeben arbeiten alle nach dem gleichen Prinzip:

1. Es wird eine Nachricht N1 an `leser_schreiber1` gesendet und damit der Wunsch zum Belegen bzw. Freigeben geäußert.

2. Es wird auf eine Nachricht N2 von `leser_schreiber1` gewartet. N2 ist als Bestätigung dafür zu verstehen, daß das Belegen bzw. Freigeben ordnungsgemäß erfolgt ist.

Mit Hilfe der Guards in den `receive` Alternativen von `start_lesen` und `start_schreiben` wird sichergestellt, daß Belegen zum Lesen bzw. Schreiben nur erfolgen kann, wenn die zugehörige Bedingung erfüllt ist.

Das Ändern der Werte von `Anzahl_Leser`, `Anzahl_Wartende_Leser` bzw. `Anzahl_Schreiber` wird mittels eines neuen Aufrufs von `leser_schreiber1` (mit den neuen Werten) erreicht.

#### 3.1.4.3 Untersuchung der Programmlösung

**Lesbarkeit**

Für die Lesbarkeit wurde eine 3 vergeben.

**Programmgröße**

Das Programm hat eine Größe von 63 nloc.

**Kommunikationsaufwand**

In der Erlang-Lösung ist der Kommunikationsaufwand unabhängig von der Reihenfolge, in der Lese- und Schreiboperationen ausgeführt werden.

Für jeden Lesezugriff sind 6 Sendeoperationen (`!`) und 6 `receive`; für jeden Schreibzugriff 4 Sendeoperationen (`!`) und 4 `receive` erforderlich. Damit ergibt sich für eine endliche Ausführung mit $l$ Lesern und $s$ Schreibern — wobei jeder Akteur genau einmal seine Operation (Lesen oder Schreiben) ausführt — ein Kommunikationsaufwand von $12l + 8s$ Anweisungen.

**Lebendigkeit**

Da in Erlang nur direkte Kommunikation existiert, gibt es nur eine Ebene, die wir betrachten. Wie in [AVWW 96] beschrieben, gehört zu jedem Prozeß P eine Mailbox, in der alle an P gesendeten Nachrichten entsprechend dem FIFO-Protokoll verwaltet werden. Damit kann kein Prozeß übergangen werden. Das heißt, die Lösung ist lebendig.

**Verklemmung**

Analog zur Ada-Lösung tritt totale Verklemmung nur dann ein, wenn ein Schreiber während einer Schreiboperation anhält und partielle Verklemmung dann, wenn mindestens ein Leser während einer Leseoperation anhält.

Da wir davon ausgehen, daß kein Prozeß während einer Lese- oder Schreiboperation anhält, ist das Programm verklemmungsfrei.

**Aushungern**

Da kein Prozeß übergangen werden kann, ist Aushungern nicht möglich.

**Faire Maschine**

Es ist keine faire Maschine erforderlich.

**Nebenläufigkeitsgrad**

Analog zur Ada-Lösung beträgt der Nebenläufigkeitsgrad $\frac{l+s}{s+1}$.

### 3.1.5 Das Leser-Schreiber-Problem mit Priorität für Leser in Java

#### 3.1.5.1 Programmentwurf

Um den Zugriff auf gemeinsam benutzte Ressourcen zu koordinieren, werden in Java synchronisierte Unterprogramme oder synchronisierte Anweisungen verwendet. Da dieser Mechanismus aber exklusiv ist, verwenden wir wieder ein Zugangsprotokoll. Da das Belegen und Freigeben exklusiv erfolgen muß, werden dafür synchronisierte Prozeduren benutzt. Außerdem werden wieder drei Variablen verwaltet, die angeben, wieviel Leseoperationen gerade ausgeführt werden, wieviel Leser gerade auf Ausführung einer Leseoperation warten bzw. ob gerade eine Schreiboperation ausgeführt wird.

#### 3.1.5.2 Programm

```
class Leser_Schreiber1 {
   protected int anzahl_leser = 0;
   protected int anzahl_wartende_leser = 0;
   protected boolean schreiber_aktiv = false;
   protected int ressource = 0;

   public int lesen() {
      int temp;
      start_lesen();
      temp = ressource;
      ende_lesen();
      return temp;
   }

   public void schreiben (int wert) {
      start_schreiben();
      ressource = wert;
      ende_schreiben();
   }

   protected synchronized void start_lesen() {
      anzahl_wartende_leser++;
      while (schreiber_aktiv) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      anzahl_wartende_leser--;
      anzahl_leser++;
   }

   protected synchronized void ende_lesen() {
      anzahl_leser--;
      if (anzahl_leser == 0)
         notifyAll();
   }

   protected synchronized void start_schreiben() {
```

```
        while (anzahl_leser>0 || anzahl_wartende_leser>0 || schreiber_aktiv) {
           try {
              wait();
           }
           catch (InterruptedException e) {}
        }
        schreiber_aktiv = true;
     }

     protected synchronized void ende_schreiben() {
        schreiber_aktiv = false;
        notifyAll();
     }
  }
```

**Abbildung 3.6** Leser-Schreiber-Problem mit Priorität für Leser in Java

Die Java-Lösung ist der CHILL-Lösung ähnlich und deshalb sei auf die Erläuterungen des CHILL-Quellcode verwiesen. Im Unterschied zu CHILL gibt es in Java für jedes Objekt nur eine Menge — die sogenannte wait set — von blockierten Threads. Deshalb können nicht gezielt Leser *oder* Schreiber aufgeweckt werden; es müssen vielmehr *alle* Threads mittels `notifyAll()` reaktiviert werden.

#### 3.1.5.3 Untersuchung der Programmlösung

**Lesbarkeit**

Für die Lesbarkeit wurde eine 2 vergeben.

**Programmgröße**

Das Programm hat eine Größe von 47 nloc.

**Kommunikationsaufwand**

Für den Kommunikationsaufwand beziehen wir uns auf die bei der Betrachtung des Nebenläufigkeitsgrades getroffenen Vereinbarungen.

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | `wait` | `notifyAll` | `wait` | `notifyAll` |
| $t_1$ | $L_1, \ldots, L_l$ | - | 1 | $s$ | - |
| $t_2$ | $S_1$ | - | - | $s-1$ | 1 |
| $t_3$ | $S_2$ | - | - | $s-2$ | 1 |
| $t_4$ | $S_3$ | - | - | $s-3$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | 1 | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.4** Kommunikationsaufwand, Fall 1

Tabelle 3.4 zeigt den Kommunikationsaufwand für den Fall 1 (alle $l$ Leser werden während $t_1$ ausgeführt). Dabei werden alle $s$ Schreiber blockiert, die Leser werden ausgeführt und wecken am Ende von $t_1$ alle Schreiber auf. Der Schreiber, der den Wettbewerb um das Lock gewinnt, wird

während $t_2$ ausgeführt und weckt am Ende von $t_2$ die $s-1$ blockierten Schreiber auf. Dieser Vorgang läuft so lange, bis alle Schreiber ausgeführt worden sind.

Der Kommunikationsaufwand beträgt für diesen Fall

$$1 + s + \sum_{i=1}^{s} i = 1 + s + \frac{s^2 + s}{2} = 1 + \frac{s^2 + 3s}{2}.$$

Der Kommunikationsaufwand für den Fall 2 (die Leser werden während $t_2$ ausgeführt) ist in Tabelle 3.5 dargestellt.

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | `wait` | `notifyAll` | `wait` | `notifyAll` |
| $t_1$ | $S_1$ | - | - | $s-1$ | 1 |
| $t_2$ | $L_1, ..., L_l$ | - | 1 | $s-1$ | - |
| $t_3$ | $S_2$ | - | - | $s-2$ | 1 |
| $t_4$ | $S_3$ | - | - | $s-3$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | 1 | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.5** Kommunikationsaufwand, Fall 2

Der Kommunikationsaufwand beträgt in diesem Fall

$$1 + s - 1 + s + \sum_{i=1}^{s-1} i = 2s + \frac{s^2 - s}{2} = \frac{s^2 + 3s}{2}.$$

Tabelle 3.6 stellt Fall 3 dar (die Leser werden während $t_3$ ausgeführt). Der Kommunikationsaufwand beträgt hierbei

$$1 + s - 2 + s + \sum_{i=1}^{s-1} i = 2s - 1 + \frac{s^2 - s}{2} = \frac{s^2 + 3s}{2} - 1.$$

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | `wait` | `notifyAll` | `wait` | `notifyAll` |
| $t_1$ | $S_1$ | - | - | $s-1$ | 1 |
| $t_2$ | $S_2$ | - | - | $s-2$ | 1 |
| $t_3$ | $L_1, ..., L_l$ | - | 1 | $s-2$ | - |
| $t_4$ | $S_3$ | - | - | $s-3$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | 1 | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.6** Kommunikationsaufwand, Fall 3

Wie man leicht sieht, beträgt der Kommunikationsaufwand für den Fall $k = 1, ..., s+1$

$$1 + s - (k-1) + s + \sum_{i=1}^{s-1} i = 2s + 2 - k + \frac{s^2 - s}{2} = 2 - k + \frac{s^2 + 3s}{2}.$$

Man erhält nun insgesamt — gemittelt über alle $s+1$ Fälle — folgenden Kommunikationsaufwand:

$$\frac{1}{s+1}\left(\sum_{i=1}^{s+1}\left(2-i+\frac{s^2+3s}{2}\right)\right)=\frac{1}{s+1}\left(\frac{(s+1)(s^2+3s)}{2}+1-\sum_{i=1}^{s-1}i\right)=$$

$$\frac{s^2+3s}{2}+\frac{1}{s+1}\left(1-\frac{s^2-s}{2}\right)=\frac{s^2+3s}{2}+\frac{2-s^2+s}{2(s+1)}=$$

$$\frac{s^2+3s}{2}+\frac{(2-s)(s+1)}{2(s+1)}=\frac{s^2+2s+2}{2}.$$

### Lebendigkeit

Analog zu Ada und CHILL unterscheiden wir wieder zwei Ebenen.

Wenn mehrere Threads um das Lock eines Objektes konkurrieren, ist nicht festgelegt, welcher Thread das Lock erhält. Damit ist es möglich, daß ein Thread bei diesem Wettbewerb unendlich oft übergangen wird. Das heißt, auf der 1. Ebene ist das Programm nicht lebendig.

Hat ein Thread T das Lock erworben, gibt es beim Belegen folgende Möglichkeiten:

- T durchläuft das Belegen erfolgreich und kann auf die Ressource zugreifen
- T wird blockiert, weil die Anmeldebedingung nicht erfüllt ist. In diesem Fall kommt T in die Wartemenge des Objektes. Mittels `notifyAll` werden alle Threads dieser Menge reaktiviert und werden Teil der Menge von Threads, die um das Lock des Objektes konkurrieren. Damit ist die Situation so, wie in der 1. Ebene.

Das bedeutet, daß das Programm auch bezüglich der 2. Ebene nicht lebendig ist.

### Verklemmung

Analog zur Ada-Lösung tritt totale Verklemmung nur dann ein, wenn ein Schreiber während einer Schreiboperation anhält und partielle Verklemmung dann, wenn mindestens ein Leser während einer Leseoperation anhält.

Da wir davon ausgehen, daß kein Thread während einer Lese- oder Schreiboperation anhält, ist das Programm verklemmungsfrei.

### Aushungern

Auf beiden Ebenen ist Aushungern möglich.

### Faire Maschine

Es ist für beide Ebenen eine faire Maschine erforderlich.

### Nebenläufigkeitsgrad

Analog zur Ada-Lösung beträgt der Nebenläufigkeitsgrad $\frac{l+s}{s+1}$.

| **Verwendete Sprache** | Ada | CHILL | Erlang | Java |
|---|---|---|---|---|
| **Lesbarkeit** | 2 | 2.5 | 3 | 2 |
| **Programmgröße** | 47 nloc | 52 nloc | 63 nloc | 47 nloc |
| **Kommunikationsaufwand** | - | $2s+1$ | $12l+8s$ | $\frac{s^2+2s+2}{2}$ |
| **Lebendigkeit** | | | | |
| auf Ebene 1 | nein | nein | - | nein |
| auf Ebene 2 | ja | nein | ja | nein |
| **Verklemmung** | nein | nein | nein | nein |
| **Aushungern** | | | | |
| auf Ebene 1 | ja | ja | - | ja |
| auf Ebene 2 | nein | ja | nein | ja |
| **Erfordernis von fairer Maschine** | | | | |
| auf Ebene 1 | ja | ja | - | ja |
| auf Ebene 2 | nein | ja | nein | ja |
| **Nebenläufigkeitsgrad** | $\frac{l+s}{s+1}$ | $\frac{l+s}{s+1}$ | $\frac{l+s}{s+1}$ | $\frac{l+s}{s+1}$ |

**Tabelle 3.7** Zusammenfassung und Gegenüberstellung der Ergebnisse

### 3.1.6 Zusammenfassung und Vergleich

Das Leser-Schreiber-Problem mit Priorität für Leser konnte in allen Sprachen gelöst werden. Für alle Lösungen wurde dabei ein Zugangsprotokoll eingesetzt. Die erhaltenen Lösungen weisen teilweise erhebliche Unterschiede auf.

Am besten ließ sich das Problem in Ada lösen. Mit Hilfe der protected entries ist es möglich, das Belegen zum Lesen und Schreiben — das nur erfolgen darf, wenn die zugehörige Bedingung erfüllt ist — auf sehr einfache Weise zu implementieren. Dabei enthält die entry barrier die Anmeldebedingung, während das eigentliche Belegen — also das Manipulieren der innerhalb des protected object definierten Variablen — im entry body enthalten ist. Das hat folgende Vorteile:

- Durch die Trennung von Belegebedingung und eigentlichem Belegen wird die Übersichtlichkeit und Lesbarkeit erhöht.
- Durch die Ordnung innerhalb der Warteschlange eines jeden entry wird verhindert, daß innerhalb der Warteschlange Tasks übergangen werden.
- Durch die Mechanismen beim Ausführen von protected entry und protected procedure wird garantiert, daß erst alle Tasks in der Warteschlange eines protected entry E abgearbeitet werden, bevor ein Task außerhalb der Warteschlange E ausführen kann.

Die letzten beiden Punkte zusammen garantieren, daß Tasks innerhalb der Warteschlange eines protected entry nicht verhungern können.

CHILL hat den Nachteil, daß die kritischen Prozeduren einer `REGION` keine bedingte Ausführung erlauben. Damit ist es notwendig, innerhalb der kritischen Prozeduren die Anmeldebedingung zu überprüfen und die ausführenden Prozesse explizit zu blockieren und zu reaktivieren. Neben dem erhöhten Kommunikationsaufwand ergibt sich hier vor allem eine größere Fehleranfälligkeit. Da es innerhalb der (bezüglich einer `EVENT`-Variable) blockierten Prozesse keine Ordnung gibt, ist Aushungern von blockierten Prozessen möglich. Außerdem ist es notwendig, das Abfragen der Bedingung

und die zugehörige `DELAY`-Anweisung innerhalb einer Schleife auszuführen, da zwischen dem Reaktivieren eines Prozesses P und dem Wiedererlangen des der `REGION` zugehörigen Locks durch P ein anderer Prozeß die Bedingung wieder geändert haben könnte.

Java hat die gleichen Nachteile wie CHILL; als weiteres Problem kommt aber noch hinzu, daß es zu jedem Objekt nur eine Wartemenge von blockierten Prozessen gibt. Damit kann beim Reaktivieren von Threads nicht mehr zwischen Lesern und Schreibern unterschieden werden (wie dies in CHILL durch die `EVENT`-Variablen möglich ist). Das bedeutet, beim Freigeben müssen immer alle blockierten Threads (mittels eines Aufrufs von `notifyAll`) reaktiviert werden, was zu einem erheblichen Kommunikationsaufwand (quadratisch in Bezug auf die Anzahl der Schreiber) führt.

Erlang besitzt keine Objekte mit koordiniertem Zugriff; Kommunikation zwischen Prozessen ist nur auf direktem Weg — über das Senden und Empfangen von Nachrichten — möglich. Damit muß für die richtige Koordinierung ein Prozeß — also ein aktives Kommunikationsobjekt — eingesetzt werden. Hierbei erweist sich als vorteilhaft, daß — mittels der Guards — ein bedingtes Empfangen von Nachrichten möglich ist. Die Ordnung innerhalb der Mailbox und die Vorgehensweise beim Empfangen von Nachrichten (siehe [AVWW 96]) gewährleistet, daß Aushungern nicht auftreten kann. Als Nachteil stellt sich das asynchrone Senden von Nachrichten in Erlang heraus. Um die Leser und Schreiber während des Belegens und Freigebens zu blockieren, müssen zusätzliche Synchronisationspunkte eingebaut werden. Dies erhöht den Kommunikationsaufwand und die Programmgröße; Übersichtlichkeit und Lesbarkeit werden verringert.

Ada, CHILL und Java legen beim Wettbewerb der Akteure um das Lock des passiven Kommunikationsobjektes keine Reihenfolge fest. Somit kann bei diesem Wettbewerb (auf Ebene 1) Aushungern von Akteuren auftreten und es ist hier jeweils eine faire Maschine erforderlich. Damit ist die Erlang-Lösung als einzige lebendig.

Bezüglich Verklemmung und Nebenläufigkeitsgrad traten keine Unterschiede in den verschiedenen Lösungen auf. Der theoretische Nebenläufigkeitsgrad wurde in allen Lösungen erreicht.

Eine Zusammenfassung der Ergebnisse ist in Tabelle 3.7 gegeben.

Auf der beiliegenden CD-ROM sind die in diesem Kapitel abgebildeten Quelltexte im Verzeichnis `broemel\Leser_Schreiber` in folgenden Dateien enthalten:

| | |
|---|---|
| Ada: | `leser_schreiber1.ads`, `leser_schreiber1.adb` |
| CHILL: | `ls1spu0a.src`, `ls1reg0s.src`, `ls1pro0s.src`, `leser01s.src`, `schrei1s.src`, `ls1tests.src` |
| Erlang: | `leser_schreiber1.erl` |
| Java: | `leser_schreiber1.java` |

## 3.2 Das Leser-Schreiber-Problem mit Priorität für Schreiber

### 3.2.1 Aufgabenstellung

Gegeben sei eine Ressource R, auf die von beliebig vielen Akteuren lesend oder schreibend zugegriffen wird. Akteure, die R lesen, werden Leser genannt; Akteure, die R schreiben, heißen Schreiber. Der Zugriff zu R durch einen Akteur A erfolgt innerhalb der Lese- bzw. Schreiboperation. Für den Zugriff zu R gelten folgende Regeln:

1. Schreibzugriffe sind exklusiv. Während eine Schreiboperation ausgeführt wird, darf keine andere Lese- oder Schreiboperation ausgeführt werden.

2. Lesezugriffe sind konkurrierend. Das heißt, wenn keine Schreiboperation ausgeführt wird, dürfen beliebig viele Leseoperationen ausgeführt werden.

3. Schreiber sollen so schnell wie möglich Zugriff zu R erhalten. Wenn ein Schreiber S eine Schreiboperation ausführen möchte, so dürfen alle Leser, die gerade eine Leseoperation ausführen, ihre Arbeit beenden. Alle Leser, die zeitlich nach S ankommen, dürfen erst dann ihre Leseoperation ausführen, wenn S mit der Ausführung seiner Schreiboperation fertig ist.

Es wird davon ausgegangen, daß kein Akteur während einer Lese- oder Schreiboperation anhält.

### 3.2.2 Das Leser-Schreiber-Problem mit Priorität für Schreiber in Ada

#### 3.2.2.1 Programmentwurf

Die Vorgehensweise ist analog zum Leser-Schreiber-Problem mit Priorität für Leser, d.h. es wird wieder ein protected object verwendet, um das Zugangsprotokoll zu realisieren. Für die richtige Koordinierung zwischen Lesern und Schreibern wird aber nun die Anzahl der wartenden Schreiber (statt der Anzahl der wartenden Leser) benötigt. Belegen zum Lesen kann nur erfolgen, wenn keine Schreiboperation ausgeführt wird und kein Schreiber auf Ausführung einer Schreiboperation wartet. Belegen zum Schreiben kann nur erfolgen, wenn kein anderer Akteur gerade eine Lese- oder Schreiboperation ausführt.

#### 3.2.2.2 Programm

```
package Leser_Schreiber2 is
   procedure Lesen (Wert : out Integer);
   procedure Schreiben (Wert : in Integer);
private
   Ressource : Integer := 0;
end Leser_Schreiber2;
package body Leser_Schreiber2 is
   protected Ressource_Sperre is
      entry Start_Lesen;
      procedure Ende_Lesen;
      entry Start_Schreiben;
      procedure Ende_Schreiben;
   private
      Anzahl_Leser : Natural := 0;
      Schreiber_Aktiv : Boolean := False;
   end Ressource_Sperre;

   protected body Ressource_Sperre is
      entry Start_Lesen when not Schreiber_Aktiv and
                             Start_Schreiben'Count = 0 is
      begin
         Anzahl_Leser := Anzahl_Leser + 1;
      end Start_Lesen;

      procedure Ende_Lesen is
      begin
         Anzahl_Leser := Anzahl_Leser - 1;
      end Ende_Lesen;

      entry Start_Schreiben when Anzahl_Leser = 0 and not Schreiber_Aktiv is
      begin
```

```
            Schreiber_Aktiv := True;
         end Start_Schreiben;

         procedure Ende_Schreiben is
         begin
            Schreiber_Aktiv := False;
         end Ende_Schreiben;
      end Ressource_Sperre;

      procedure Lesen (Wert : out Integer) is
      begin
         Ressource_Sperre.Start_Lesen;
         Wert := Ressource;
         Ressource_Sperre.Ende_Lesen;
      end Lesen;

      procedure Schreiben (Wert : in Integer) is
      begin
         Ressource_Sperre.Start_Schreiben;
         Ressource := Wert;
         Ressource_Sperre.Ende_Schreiben;
      end Schreiben;
   end Leser_Schreiber2;
```

**Abbildung 3.7** Leser-Schreiber-Problem mit Priorität für Schreiber in Ada

Die Wirkungsweise ist analog der Lösung mit Priorität für Leser. Der Unterschied besteht in den geänderten Bedingungen in den entry barriers. Damit wird die Bevorzugung der Schreiber gewährleistet.

#### 3.2.2.3 Untersuchung der Programmlösung

**Lesbarkeit**

Für die Lesbarkeit wurde eine 2 vergeben.

**Programmgröße**

Die Programmgröße beträgt 47 nloc.

**Kommunikationsaufwand**

Es ist kein Kommunikationsaufwand erforderlich.

**Lebendigkeit**

Es gilt die gleiche Argumentation wie bei der Lösung mit Priorität für Leser. Das heißt, das Programm ist bezüglich der 1. Ebene nicht lebendig und bezüglich der 2. Ebene lebendig.

### Verklemmung

Analog zur Lösung mit Priorität für Leser tritt totale Verklemmung nur dann ein, wenn ein Schreiber während einer Schreiboperation anhält und partielle Verklemmung dann, wenn mindestens ein Leser während einer Leseoperation anhält.

Da wir davon ausgehen, daß kein Akteur während einer Lese- oder Schreiboperation anhält, ist das Programm verklemmungsfrei.

### Aushungern

Auf der 1. Ebene ist Aushungern möglich, da Akteure unendlich oft übergangen werden können. Auf der 2. Ebene wird durch die Mechanismen des protected entry garantiert, daß kein Akteur übergangen wird.

### Faire Maschine

Für Ebene 1 ist eine faire Maschine erforderlich.

### Nebenläufigkeitsgrad

Für die Betrachtung des Nebenläufigkeitsgrades gehen wir von den Vereinbarungen aus, die beim Leser-Schreiber-Problem mit Priorität für Leser getroffen wurden. Das bedeutet, es gibt $s+1$ mögliche Ausführungen, da es $s+1$ Zeitspannen gibt, in denen jeweils alle $l$ Leser ausgeführt werden.

Es muß jedoch folgendes beachtet werden: Da Schreiber Priorität haben, können die Leser nur dann während einer Zeitspanne $t_i$ ausgeführt werden, wenn kein Schreiber gerade seine Operation ausführen will. Aus diesem Grund treffen wir für den Ablauf der $s+1$ Ausführungen zusätzlich die folgenden Vereinbarungen:

- Die Ausführung $i = 1, ..., s+1$ sei die Ausführung, bei der die $l$ Leser während der Zeitspanne $t_i$ ausgeführt werden.
- Zu Beginn der Ausführung $i$ (noch vor Beginn von $t_1$) werden alle Leser $L_1, ..., L_l$ und die ersten $i-1$ Schreiber $S_1, ..., S_{i-1}$ gestartet. Zu Beginn von Ausführung 1 wird kein Schreiber gestartet.
- Während der Zeitspannen $t_1, ..., t_{i-1}$ werden die Schreiber $S_1, ..., S_{i-1}$ ausgeführt.
- Während der Zeitspanne $t_i$ werden die Leser $L_1, ..., L_l$ ausgeführt und die Schreiber $S_i, ..., S_s$ gestartet.
- Während der Zeitspannen $t_{i+1}, ..., t_{s+1}$ werden die Schreiber $S_i, ..., S_s$ ausgeführt.

Der theoretische Nebenläufigkeitsgrad beträgt $\frac{l+s}{s+1}$.

Zur besseren Veranschaulichung ist in Abbildung 3.8 die Ausführung 1 und in Abbildung 3.9 die Ausführung 2 dargestellt.

Der Nebenläufigkeitsgrad beträgt für jede Ausführung $\frac{l+s}{s+1}$, d.h. der theoretische Nebenläufigkeitsgrad wurde erreicht.

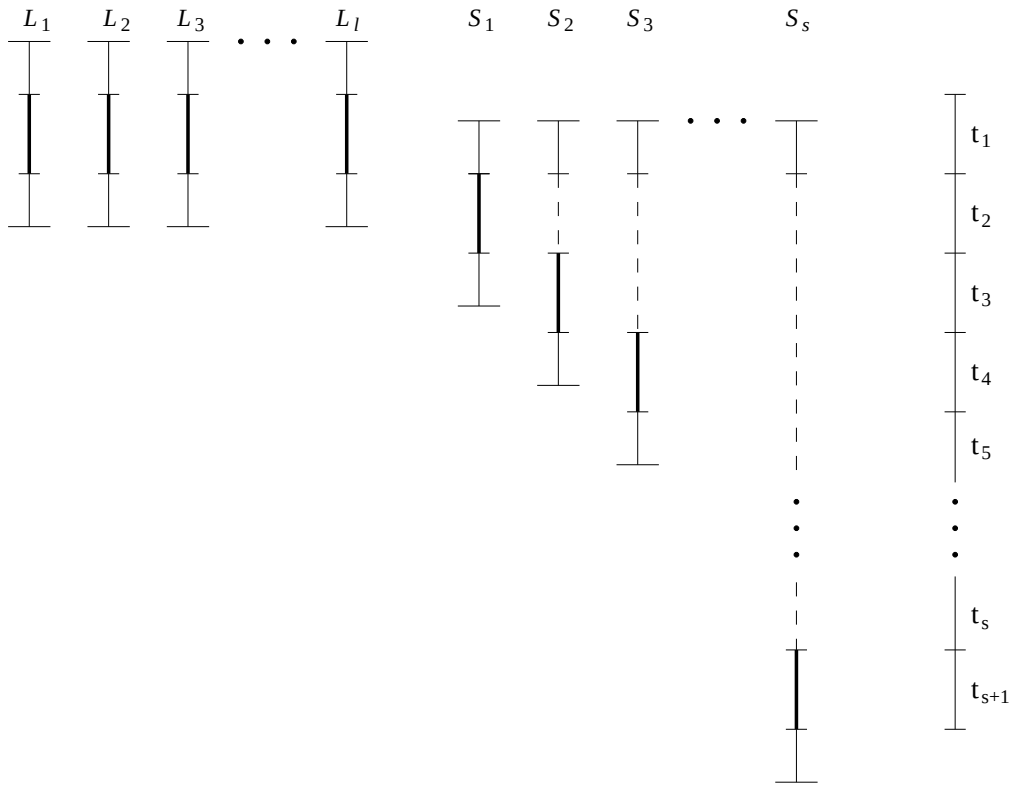


**Abbildung 3.8** Betrachtung des Nebenläufigkeitsgrades, Ausführung 1

### 3.2.3 Das Leser-Schreiber-Problem mit Priorität für Schreiber in CHILL

#### 3.2.3.1 Programmentwurf

Die Vorgehensweise ist analog zum Leser-Schreiber-Problem mit Priorität für Leser, d.h. es wird wieder eine `REGION` verwendet, um das Zugangsprotokoll zu realisieren. Für die richtige Koordinierung zwischen Lesern und Schreibern wird aber nun zusätzlich noch die Anzahl der wartenden Schreiber innerhalb der `REGION` verwaltet.

#### 3.2.3.2 Programm

```
leser_schreiber2: REGION
   GRANT start_lesen, ende_lesen, start_schreiben, ende_schreiben;
   DCL anzahl_leser INT := 0,
       anzahl_wartende_leser INT := 0,
       anzahl_wartende_schreiber INT := 0,
       schreiber_aktiv BOOL := FALSE,
       kein_schreiber, kein_akteur EVENT;

   start_lesen: PROC ();
      anzahl_wartende_leser := anzahl_wartende_leser + 1;
      DO WHILE schreiber_aktiv OR anzahl_wartende_schreiber > 0;
         DELAY kein_schreiber;
      OD;
      anzahl_wartende_leser := anzahl_wartende_leser - 1;
      anzahl_leser := anzahl_leser + 1;
   END start_lesen;
```

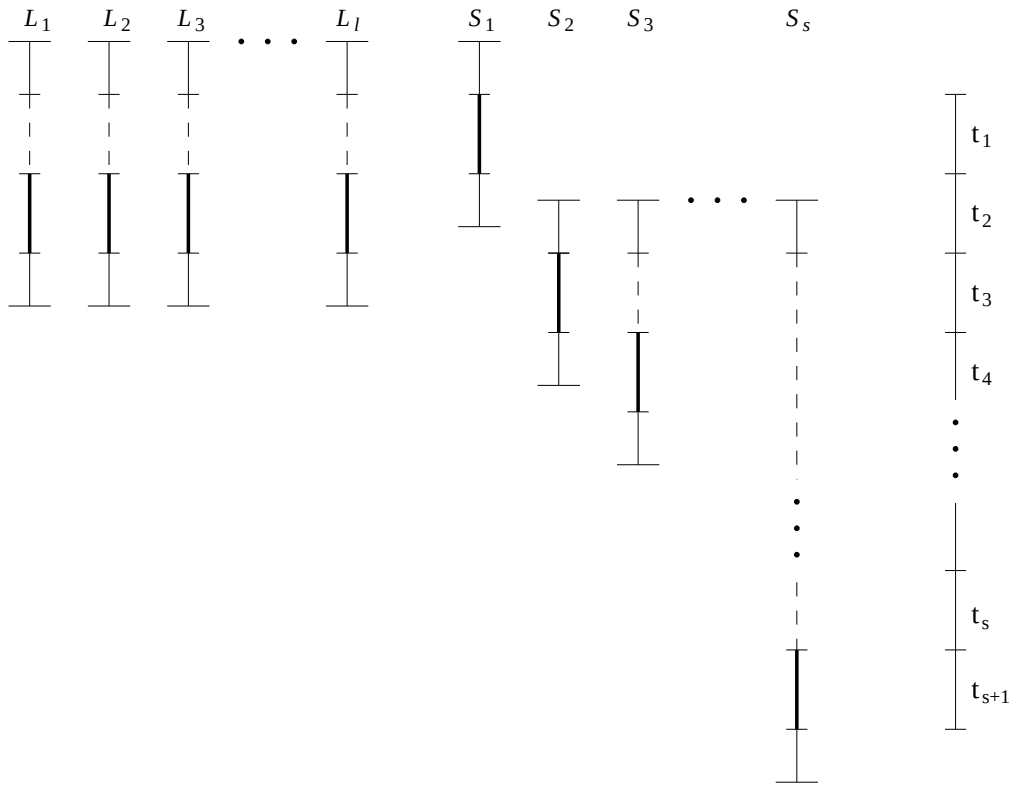


**Abbildung 3.9** Betrachtung des Nebenläufigkeitsgrades, Ausführung 2

```
   ende_lesen: PROC ();
      anzahl_leser := anzahl_leser - 1;
      IF anzahl_leser = 0 AND anzahl_wartende_schreiber > 0
         THEN CONTINUE kein_akteur;
      FI;
   END ende_lesen;

   start_schreiben: PROC ();
      anzahl_wartende_schreiber := anzahl_wartende_schreiber + 1;
      DO WHILE anzahl_leser > 0 OR schreiber_aktiv;
         DELAY kein_akteur;
      OD;
      anzahl_wartende_schreiber := anzahl_wartende_schreiber - 1;
      schreiber_aktiv := TRUE;
   END start_schreiben;

   ende_schreiben: PROC ();
      schreiber_aktiv := FALSE;
      IF anzahl_wartende_schreiber > 0
      THEN CONTINUE kein_akteur;
      ELSE DO FOR i := 1 TO anzahl_wartende_leser;
              CONTINUE kein_schreiber;
           OD;
      FI;
   END ende_schreiben;
END leser_schreiber2;
```

```
leser_schreiber: MODULE
   SEIZE start_lesen, ende_lesen, start_schreiben, ende_schreiben;
   GRANT lesen, schreiben;
   DCL ressource INT := 0;

   lesen: PROC (wert INT OUT);
      start_lesen();
      wert := ressource;
      ende_lesen();
   END lesen;

   schreiben: PROC (wert INT IN);
      start_schreiben();
      ressource := wert;
      ende_schreiben();
   END schreiben;
END leser_schreiber;
```

**Abbildung 3.10** Leser-Schreiber-Problem mit Priorität für Schreiber in CHILL

Die Wirkungsweise ist analog der Lösung mit Priorität für Leser. Der Unterschied besteht in den geänderten Bedingungen in den kritischen Prozeduren, um Bevorzugung der Schreiber zu gewährleisten.

#### 3.2.3.3 Untersuchung der Programmlösung

**Lesbarkeit**

Für die Lesbarkeit wurde eine 2.5 vergeben.

**Programmgröße**

Das Programm hat eine Größe von 54 nloc.

**Kommunikationsaufwand**

Wir beziehen uns dabei auf die beim Nebenläufigkeitsgrad betrachteten Ausführungen.

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | DELAY | CONTINUE | DELAY | CONTINUE |
| $t_1$ | $L_1, \ldots, L_l$ | - | 1 | - | - |
| $t_2$ | $S_1$ | - | - | $s-1$ | 1 |
| $t_3$ | $S_2$ | - | - | - | 1 |
| $t_4$ | $S_3$ | - | - | - | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | - | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | - |

**Tabelle 3.8** Kommunikationsaufwand, Ausführung 1

Ausführung 1 (Abbildung 3.8) ist in Tabelle 3.8 dargestellt. Der Kommunikationsaufwand beträgt dabei $1 + s - 1 + s - 1 = 2s - 1$.

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | DELAY | CONTINUE | DELAY | CONTINUE |
| $t_1$ | $S_1$ | $l$ | - | - | $l$ |
| $t_2$ | $L_1, ..., L_l$ | - | 1 | - | - |
| $t_3$ | $S_2$ | - | - | $s-2$ | 1 |
| $t_4$ | $S_3$ | - | - | - | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | - | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | - |

**Tabelle 3.9** Kommunikationsaufwand, Ausführung 2

Ausführung 2 (Abbildung 3.9) wird in Tabelle 3.9 gezeigt. In diesem Fall beträgt der Kommunikationsaufwand

$$2l + 1 + s - 2 + s - 2 = 2l + 2s - 3 = 2(l + s) - 3.$$

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | DELAY | CONTINUE | DELAY | CONTINUE |
| $t_1$ | $S_1$ | $l$ | - | 1 | 1 |
| $t_2$ | $S_2$ | - | - | - | $l$ |
| $t_3$ | $L_1, ..., L_l$ | - | 1 | - | - |
| $t_4$ | $S_3$ | - | - | $s-3$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | - | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | - |

**Tabelle 3.10** Kommunikationsaufwand, Ausführung 3

Ausführung 3 wird in Tabelle 3.10 dargestellt. Für diese Ausführung beträgt der Kommunikationsaufwand

$$2l + 2 + s - 3 + s - 2 = 2l + 2s - 3 = 2(l + s) - 3.$$

Wie man sich leicht überlegen kann, beträgt der Kommunikationsaufwand für die Ausführung $i = 2, ..., s + 1$ immer $2(l + s) - 3$.

Damit gilt für den durchschnittlichen Kommunikationsaufwand

$$\frac{1}{s+1}\left(2s - 1 + \sum_{i=2}^{s+1}(2(l+s) - 3)\right) = \frac{2s - 1 + 2s^2 + 2ls - 3s}{s+1} = \frac{2s^2 + 2sl - s - 1}{s+1} = \frac{2(s^2 + sl)}{s+1} - 1.$$

### Lebendigkeit

Analog zur Lösung mit Priorität für Leser ist das Programm bezüglich beider Ebenen nicht lebendig.

### Verklemmung

Analog zur Lösung mit Priorität für Leser tritt totale Verklemmung nur dann ein, wenn ein Schreiber während einer Schreiboperation anhält und partielle Verklemmung dann, wenn mindestens ein Leser während einer Leseoperation anhält.

Da wir davon ausgehen, daß kein Prozeß während einer Lese- oder Schreiboperation anhält, ist das Programm verklemmungsfrei.

#### Aushungern

Auf beiden Ebenen ist Aushungern von Prozessen möglich.

#### Faire Maschine

Auf beiden Ebenen ist eine faire Maschine erforderlich.

#### Nebenläufigkeitsgrad

Analog zur Ada-Lösung beträgt der Nebenläufigkeitsgrad $\frac{l+s}{s+1}$.

### 3.2.4 Das Leser-Schreiber-Problem mit Priorität für Schreiber in Erlang

#### 3.2.4.1 Programmentwurf

Die Vorgehensweise ist analog zum Leser-Schreiber-Problem mit Priorität für Leser, d.h. es wird wieder ein Verwaltungsprozeß VP verwendet, um das Zugangsprotokoll zu realisieren. Für die richtige Koordinierung zwischen Lesern und Schreibern wird aber nun die Anzahl der wartenden Schreiber (statt der Anzahl der wartenden Leser) als Parameter von VP verwaltet.

#### 3.2.4.2 Programm

```
-module(leser_schreiber2).
-export([start/0, leser_schreiber2/3, lesen/0, schreiben/1]).

start() ->
   register(leser_schreiber, spawn(leser_schreiber2, leser_schreiber2,
            [0, 0, 0])).
lesen() ->
   start_lesen(),
   {ok, Binaerobjekt} = file:read_file("ressource"),
   Wert = binary_to_term(Binaerobjekt),
   ende_lesen(),
   Wert.

schreiben(Wert) ->
   anfrage_schreiben(),
   start_schreiben(),
   file:write_file("ressource", term_to_binary(Wert)),
   ende_schreiben().

start_lesen() ->
   leser_schreiber ! {self(), start_lesen},
   receive
      lesen_gestartet ->
         true
   end.
```

```
ende_lesen() ->
   leser_schreiber ! {self(), ende_lesen},
   receive
      lesen_beendet ->
         true
   end.

anfrage_schreiben() ->
   leser_schreiber ! {self(), anfrage_schreiben},
   receive
      anfrage_gestartet ->
         true
   end.

start_schreiben() ->
   leser_schreiber ! {self(), start_schreiben},
   receive
      schreiben_gestartet ->
         true
   end.

ende_schreiben() ->
   leser_schreiber ! {self(), ende_schreiben},
   receive
      schreiben_beendet ->
         true
   end.

leser_schreiber2 (Anzahl_Leser, Anzahl_Wartende_Schreiber, Anzahl_Schreiber) ->
   receive
      {PID, start_lesen} when Anzahl_Schreiber==0,
                        Anzahl_Wartende_Schreiber==0 ->
         PID ! lesen_gestartet,
         leser_schreiber2 (Anzahl_Leser+1, Anzahl_Wartende_Schreiber,
                           Anzahl_Schreiber);

      {PID, ende_lesen} ->
         PID ! lesen_beendet,
         leser_schreiber2 (Anzahl_Leser-1, Anzahl_Wartende_Schreiber,
                           Anzahl_Schreiber);

      {PID, anfrage_schreiben} ->
         PID ! anfrage_gestartet,
         leser_schreiber2 (Anzahl_Leser, Anzahl_Wartende_Schreiber+1,
                           Anzahl_Schreiber);

      {PID, start_schreiben} when Anzahl_Leser==0, Anzahl_Schreiber==0 ->
         PID ! schreiben_gestartet,
         leser_schreiber2 (Anzahl_Leser, Anzahl_Wartende_Schreiber-1,
                           Anzahl_Schreiber+1);
```

```
      {PID, ende_schreiben} ->
         PID ! schreiben_beendet,
         leser_schreiber2 (Anzahl_Leser, Anzahl_Wartende_Schreiber,
                           Anzahl_Schreiber-1)
end.
```

**Abbildung 3.11** Leser-Schreiber-Problem mit Priorität für Schreiber in Erlang

Die Wirkungsweise ist analog der Lösung mit Priorität für Leser. Durch die geänderten Bedingungen in den Guards wird die Bevorzugung der Schreiber gewährleistet.

#### 3.2.4.3 Untersuchung der Programmlösung

**Lesbarkeit**

Für die Lesbarkeit wurde eine 3 vergeben.

**Programmgröße**

Das Programm hat eine Größe von 63 nloc.

**Kommunikationsaufwand**

Analog zur Lösung mit Priorität für Leser beträgt der Kommunikationsaufwand $12l + 8s$ Anweisungen.

**Lebendigkeit**

Analog zur Lösung mit Priorität für Leser gilt auch hier, daß das Programm lebendig ist.

**Verklemmung**

Analog zur Lösung mit Priorität für Leser tritt totale Verklemmung nur dann ein, wenn ein Schreiber während einer Schreiboperation anhält und partielle Verklemmung dann, wenn mindestens ein Leser während einer Leseoperation anhält.

Da wir davon ausgehen, daß kein Prozeß während einer Lese- oder Schreiboperation anhält, ist das Programm verklemmungsfrei.

**Aushungern**

Da kein Prozeß übergangen werden kann, ist Aushungern nicht möglich.

**Faire Maschine**

Es ist keine faire Maschine erforderlich.

**Nebenläufigkeitsgrad**

Analog zur Ada-Lösung beträgt der Nebenläufigkeitsgrad $\frac{l+s}{s+1}$.

### 3.2.5 Das Leser-Schreiber-Problem mit Priorität für Schreiber in Java

#### 3.2.5.1 Programmentwurf

Die Vorgehensweise ist analog zur Lösung mit Priorität für Leser, jedoch wird statt der Anzahl der wartenden Leser nun die Anzahl der wartenden Schreiber verwaltet.

#### 3.2.5.2 Programm

```
class Leser_Schreiber2 {
   protected int anzahl_leser = 0;
   protected int anzahl_wartende_schreiber = 0;
   protected boolean schreiber_aktiv = false;
   protected int ressource = 0;

   public int lesen() {
      int temp;
      start_lesen();
      temp = ressource;
      ende_lesen();
      return temp;
   }

   public void schreiben (int wert) {
      start_schreiben();
      ressource = wert;
      ende_schreiben();
   }

   protected synchronized void start_lesen() {
      while (schreiber_aktiv || anzahl_wartende_schreiber>0) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      anzahl_leser++;
   }

   protected synchronized void ende_lesen() {
      anzahl_leser--;
      if (anzahl_leser == 0)
         notifyAll();
   }

   protected synchronized void start_schreiben() {
      anzahl_wartende_schreiber++;
      while (anzahl_leser>0 || schreiber_aktiv) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
```

```
        }
        anzahl_wartende_schreiber--;
        schreiber_aktiv = true;
    }

    protected synchronized void ende_schreiben() {
        schreiber_aktiv = false;
        notifyAll();
    }
}
```

**Abbildung 3.12** Leser-Schreiber-Problem mit Priorität für Schreiber in Java

Die Wirkungsweise ist analog der Lösung mit Priorität für Leser. Der Unterschied besteht in den geänderten Bedingungen in den synchronisierten Prozeduren, um Bevorzugung der Schreiber zu gewährleisten.

#### 3.2.5.3 Untersuchung der Programmlösung

**Lesbarkeit**

Für die Lesbarkeit wurde eine 2 vergeben.

**Programmgröße**

Das Programm hat eine Größe von 47 nloc.

**Kommunikationsaufwand**

Wir beziehen uns dabei auf die beim Nebenläufigkeitsgrad betrachteten Ausführungen.

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitspanne | ausgeführt? | `wait` | `notifyAll` | `wait` | `notifyAll` |
| $t_1$ | $L_1, \ldots, L_l$ | - | 1 | - | - |
| $t_2$ | $S_1$ | - | - | $s-1$ | 1 |
| $t_3$ | $S_2$ | - | - | $s-2$ | 1 |
| $t_4$ | $S_3$ | - | - | $s-3$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | 1 | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.11** Kommunikationsaufwand, Fall 1

Der Kommunikationsaufwand für Ausführung 1 (Abbildung 3.8) ist in Tabelle 3.11 dargestellt. Der Kommunikationsaufwand beträgt hierbei

$$1 + s + \sum_{i=1}^{s-1} i = 1 + \sum_{i=1}^{s} i = 1 + \frac{s^2 + s}{2}.$$

Ausführung 2 (Abbildung 3.9) ist in Tabelle 3.12 dargestellt. Dabei beträgt der Kommunikationsaufwand

$$l + 1 + s + \sum_{i=1}^{s-2} i = l + 1 + s + \frac{(s-1)(s-2)}{2} = l + 1 + \frac{s^2 - s + 2}{2}.$$

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitpunkt | ausgeführt? | `wait` | `notifyAll` | `wait` | `notifyAll` |
| $t_1$ | $S_1$ | $l$ | - | - | 1 |
| $t_2$ | $L_1, ..., L_l$ | - | 1 | - | - |
| $t_3$ | $S_2$ | - | - | $s-2$ | 1 |
| $t_4$ | $S_3$ | - | - | $s-3$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | 1 | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.12** Kommunikationsaufwand, Fall 2

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitpunkt | ausgeführt? | `wait` | `notifyAll` | `wait` | `notifyAll` |
| $t_1$ | $S_1$ | $l$ | - | 1 | 1 |
| $t_2$ | $S_2$ | $l$ | - | - | 1 |
| $t_3$ | $L_1, ..., L_l$ | - | 1 | - | - |
| $t_4$ | $S_3$ | - | - | $s-3$ | 1 |
| $t_5$ | $S_4$ | - | - | $s-4$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_s$ | $S_{s-1}$ | - | - | 1 | 1 |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.13** Kommunikationsaufwand, Fall 3

Ausführung 3 ist in Tabelle 3.13 gezeigt. Hierbei ist der Kommunikationsaufwand

$$2l + 1 + s + 1 + \sum_{i=1}^{s-3} i = 2l + 2 + s + \frac{(s-3)(s-2)}{2} = 2l + 2 + \frac{s^2 - 3s + 6}{2}.$$

Der Kommunikationsaufwand für die Ausführung $i = 3, ..., s-1$ beträgt

$$(i-1)l + 1 + s + \sum_{j=1}^{i-2} j + \sum_{j=1}^{s-i} j.$$

Dabei bezeichnet die erste Summe die Anzahl der von den Schreibern $S_1, ..., S_{i-1}$ ausgeführten `wait`, während die zweite Summe die Anzahl der `wait` angibt, die von den Schreibern $S_i, ..., S_s$ ausgeführt werden. Für die Ausführungen 1 und 2 hat die erste Summe den Wert Null; für die Ausführungen $s$ und $s+1$ hat die zweite Summe den Wert Null.

Die Ausführungen $s$ und $s+1$ sind in den Tabellen 3.14 und 3.15 dargestellt.
Der Kommunikationsaufwand der Ausführung $s$ ist

$$(s-1)l + 1 + s + \sum_{i=1}^{s-2} i = (s-1)l + 1 + s + \frac{(s-2)(s-1)}{2} = (s-1)l + 1 + \frac{s^2 - s + 2}{2}.$$

Für die Ausführung $s+1$ ergibt sich ein Kommunikationsaufwand von

$$sl + 1 + s + \sum_{i=1}^{s-1} i = sl + 1 + s + \frac{(s-1)s}{2} = sl + 1 + \frac{s^2 + s}{2}.$$

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitpunkt | ausgeführt? | `wait` | `notifyAll` | `wait` | `notifyAll` |
| $t_1$ | $S_1$ | $l$ | - | $s-2$ | 1 |
| $t_2$ | $S_2$ | $l$ | - | $s-3$ | 1 |
| $t_3$ | $S_3$ | $l$ | - | $s-4$ | 1 |
| $t_4$ | $S_4$ | $l$ | - | $s-5$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_{s-1}$ | $S_{s-1}$ | $l$ | - | - | 1 |
| $t_s$ | $L_1, \ldots, L_l$ | - | 1 | - | - |
| $t_{s+1}$ | $S_s$ | - | - | - | 1 |

**Tabelle 3.14** Kommunikationsaufwand, Fall $s$

| | Wer wird | Leser | | Schreiber | |
|---|---|---|---|---|---|
| Zeitpunkt | ausgeführt? | `wait` | `notifyAll` | `wait` | `notifyAll` |
| $t_1$ | $S_1$ | $l$ | - | $s-1$ | 1 |
| $t_2$ | $S_2$ | $l$ | - | $s-2$ | 1 |
| $t_3$ | $S_3$ | $l$ | - | $s-3$ | 1 |
| $t_4$ | $S_4$ | $l$ | - | $s-4$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $t_{s-1}$ | $S_{s-1}$ | $l$ | | 1 | 1 |
| $t_s$ | $S_s$ | $l$ | - | - | 1 |
| $t_{s+1}$ | $L_1, \ldots, L_l$ | - | 1 | - | - |

**Tabelle 3.15** Kommunikationsaufwand, Fall $s+1$

Für den durchschnittlichen Kommunikationsaufwand gilt nun

$$\frac{1}{s+1}\left(1+\frac{s^2+s}{2}+l+1+\frac{s^2-s+2}{2}+\sum_{i=3}^{s-1}\left((i-1)\,l+1+s\right)+\right.$$
$$\left.\sum_{i=3}^{s-1}\left(\sum_{j=1}^{i-2}j+\sum_{j=1}^{s-i}j\right)+(s-1)l+1+\frac{s^2-s+2}{2}+sl+1+\frac{s^2+s}{2}\right)=$$

$$\frac{1}{s+1}\left(2+l+\frac{2s^2+2}{2}+l\sum_{i=2}^{s-2}i+(s-3)(1+s)+\sum_{i=3}^{s-1}\sum_{j=1}^{i-2}j+\sum_{i=3}^{s-1}\sum_{j=1}^{s-i}j+(2s-1)l+2+\frac{2s^2+2}{2}\right)=$$

$$\frac{1}{s+1}\left(2sl+4+2s^2+2+l\frac{(s-2)(s-1)-2}{2}+s^2-2s-3+2\sum_{i=1}^{s-3}\sum_{j=1}^{i}j\right)=$$

$$\frac{1}{s+1}\left(3s^2+2sl-2s+3+\frac{l}{2}(s^2-3s)+2\frac{(s-3)(s-2)(s-1)}{6}\right)=$$

$$\frac{1}{s+1}\left(3s^2+\frac{l}{2}s^2+\frac{l}{2}s-2s+3+\frac{s^3-6s^2+11s-6}{3}\right)=$$

$$\frac{1}{s+1}\left(\frac{1}{3}s^3+s^2+\frac{l}{2}(s^2+s)+\frac{5}{3}s+1\right)=$$

$$\frac{1}{s+1}\left(\frac{1}{3}s^2(s+1)+\frac{2}{3}s(s+1)+\frac{l}{2}s(s+1)+s+1\right)=$$

$$\frac{1}{3}s^2+\frac{2}{3}s+\frac{l}{2}s+1.$$

#### Lebendigkeit

Analog zur Lösung mit Priorität für Leser ist das Programm bezüglich beider Ebenen nicht lebendig.

#### Verklemmung

Analog zur Lösung mit Priorität für Leser tritt totale Verklemmung nur dann ein, wenn ein Schreiber während einer Schreiboperation anhält und partielle Verklemmung dann, wenn mindestens ein Leser während einer Leseoperation anhält.

Da wir davon ausgehen, daß kein Thread während einer Lese- oder Schreiboperation anhält, ist das Programm verklemmungsfrei.

#### Aushungern

Auf beiden Ebenen ist Aushungern möglich.

#### Faire Maschine

Für beide Ebenen ist eine faire Maschine erforderlich.

#### Nebenläufigkeitsgrad

Analog zur Ada-Lösung beträgt der Nebenläufigkeitsgrad $\frac{l+s}{s+1}$.

### 3.2.6 Zusammenfassung und Vergleich

Analog zum Leser-Schreiber-Problem mit Priorität für Leser erweist sich Ada's protected object als am besten geeignet für die Lösung des Leser-Schreiber-Problems mit Priorität für Schreiber. Da ansonsten bezüglich der Vor- und Nachteile der einzelnen Sprachen dieselbe Argumentation gilt wie beim Leser-Schreiber-Problem mit Priorität für Leser, sei auf die dortige Zusammenfassung verwiesen.

Tabelle 3.16 faßt die Lösungen zum Leser-Schreiber-Problem mit Priorität für Schreiber noch einmal zusammen.

Auf der beiliegenden CD-ROM sind die in diesem Kapitel abgebildeten Quelltexte im Verzeichnis `broemel\Leser_Schreiber` in folgenden Dateien enthalten:

| | |
|---|---|
| Ada: | `leser_schreiber2.ads`, `leser_schreiber2.adb` |
| CHILL: | `ls2spu0a.src`, `ls2reg0s.src`, `ls2pro0s.src`, `leser02s.src`, `schrei2s.src`, `ls2tests.src` |
| Erlang: | `leser_schreiber2.erl` |
| Java: | `leser_schreiber2.java` |

| **Verwendete Sprache** | Ada | CHILL | Erlang | Java |
|---|---|---|---|---|
| **Lesbarkeit** | 2 | 2.5 | 3 | 2 |
| **Programmgröße** | 47 nloc | 52 nloc | 63 nloc | 47 nloc |
| **Kommunikationsaufwand** | - | $\frac{2(s^2+sl)}{s+1} - 1$ | $12l + 8s$ | $\frac{1}{3}s^2 + \frac{2}{3}s + \frac{l}{2}s + 1$ |
| **Lebendigkeit** | | | | |
| auf Ebene 1 | nein | nein | - | nein |
| auf Ebene 2 | ja | nein | ja | nein |
| **Verklemmung** | nein | nein | nein | nein |
| **Aushungern** | | | | |
| auf Ebene 1 | ja | ja | - | ja |
| auf Ebene 2 | nein | ja | nein | ja |
| **Erfordernis von fairer Maschine** | | | | |
| auf Ebene 1 | ja | ja | - | ja |
| auf Ebene 2 | nein | ja | nein | ja |
| **Nebenläufigkeitsgrad** | $\frac{l+s}{s+1}$ | $\frac{l+s}{s+1}$ | $\frac{l+s}{s+1}$ | $\frac{l+s}{s+1}$ |

**Tabelle 3.16** Zusammenfassung und Gegenüberstellung der Ergebnisse

# Kapitel 4

# Die Kabinenbahn

## 4.1 Aufgabenstellung

In [HH 94: 421] ist eine Ada83-Implementierung eines Steuerungssystems für eine Kabinenbahnlinie dargestellt. Dieses Steuerungssystem soll in Ada95, CHILL, Erlang und Java implementiert und die einzelnen Lösungen dann miteinander verglichen werden.

Zunächst geben wir noch einmal die genaue Aufgabenstellung:

Gegeben ist eine Folge von Bahnhöfen, die durch eine Kabinenbahnlinie miteinander verbunden sind. Benachbarte Bahnhöfe sind durch eine zweigleisige Strecke mit je einem Gleis für den Verkehr in eine Richtung verbunden. Mit Hilfe von Kehrschleifen kann eine Kabine von einem Gleis auf das andere wechseln. Das bedeutet, die Gleise bilden einen Kreis, der in einer Richtung — entgegen dem Uhrzeigersinn — durchfahren wird. Abbildung 4.1 zeigt die Strecke mit 5 Bahnhöfen.

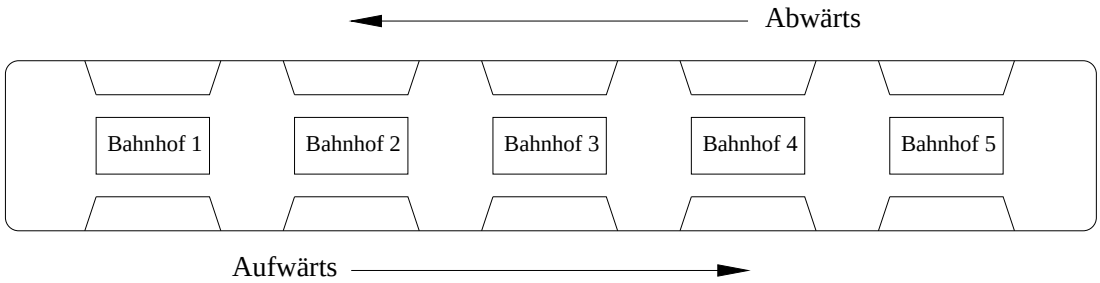


**Abbildung 4.1** Strecke der Kabinenbahn

In jedem Bahnhof gibt es für eine Kabine die Möglichkeit, den Bahnhof ohne Halt zu passieren oder an den Bahnsteig heranzufahren, was durch je eine Weiche vor und hinter dem Bahnhof ermöglicht wird. Während eine Kabine K im Bahnhof steht, können andere Kabinen an K auf der geraden Strecke vorbeifahren. In jedem Bahnhof kann maximal eine Kabine in jeder Fahrtrichtung stehen. Abbildung 4.2 zeigt einen Bahnhof.

Jeder Bahnhof besitzt einen Fahrkartenautomaten, an dem die Fahrgäste ihre Fahrkarten kaufen. Eine Gruppe von Passagieren mit dem gleichen Fahrziel löst zusammen eine Fahrkarte und wird in einer Kabine befördert. Fahrten verlaufen grundsätzlich auf dem kürzesten Weg, das bedeutet, es gibt keine Fahrten mit Passagieren durch die Kehrschleifen.

Kabinen können nur eine gewisse Zahl von Passagieren aufnehmen. Hat eine Kabine noch freie Plätze, so kann sie an einem Bahnhof anhalten, um weitere Fahrgäste aufzunehmen. Wenn eine Kabine keinen Fahrauftrag besitzt, so wartet sie in einem Bahnhof.

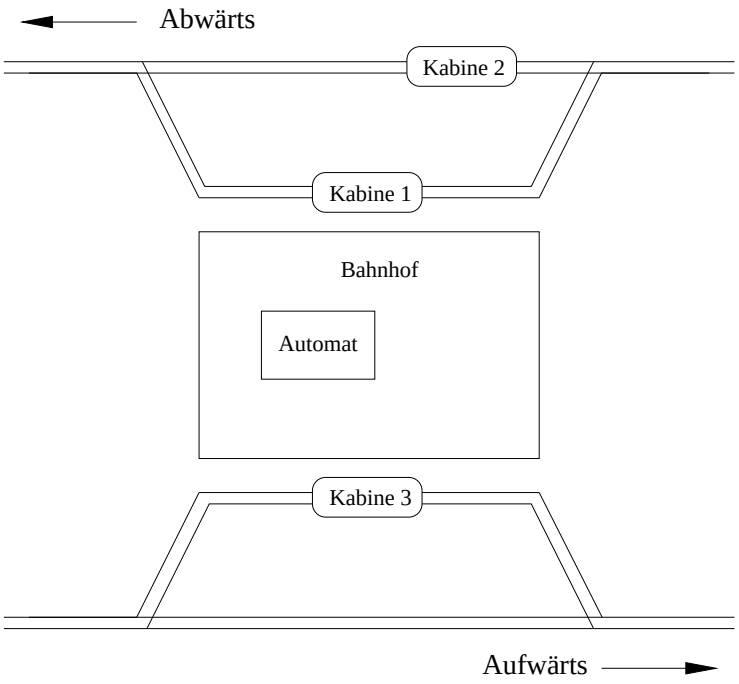


**Abbildung 4.2** Darstellung eines Bahnhofs

## 4.2 Die Kabinenbahn in Ada

### 4.2.1 Programmentwurf

Analog zur Ada83-Lösung wird jede Kabine von einem eigenen Prozeß gesteuert. Für jeden Fahrkartenautomaten wird ebenfalls ein Prozeß verwendet. Außerdem wird jede Gruppe von Passagieren als eigenständiger Prozeß dargestellt.

Ein besonders wichtiger Bestandteil der Kabinenbahnsteuerung ist die Kontrolle der Kabinenbewegung, die verhindert, daß Kabinen zusammenstoßen. Dazu wird das Verfahren der Blocksicherung eingesetzt: Die Fahrstrecke wird in Blockabschnitte eingeteilt, in denen sich maximal eine fahrende Kabine befinden darf. In unserem Fall beginnt bzw. endet ein Blockabschnitt immer in der Mitte eines Gleises zwischen zwei Bahnhöfen. Das bedeutet, daß zu jedem Bahnhof zwei Blockabschnitte — einer für jede Fahrtrichtung — existieren. Abbildung 4.3 zeigt die Blockabschnitte für die Strecke aus Abbildung 4.1. Dabei wird jeder Blockabschnitt durch ein Tupel der Form (Richtung, Bahnhof) bezeichnet.

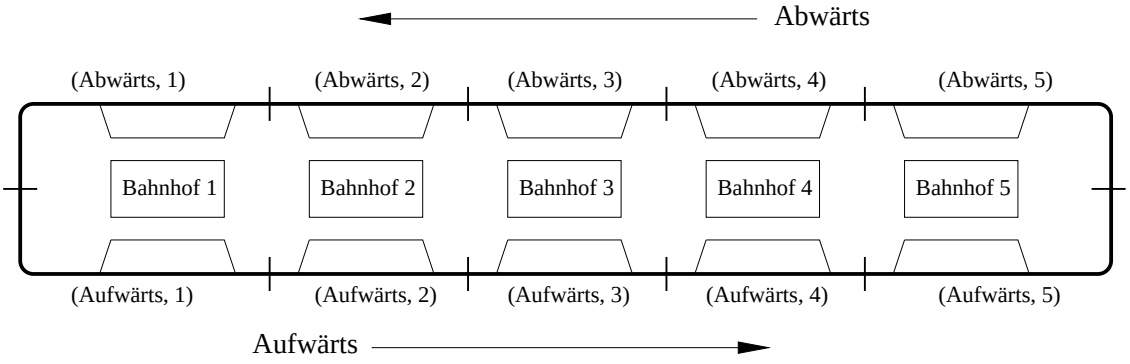


**Abbildung 4.3** Unterteilung der Fahrstrecke in Blockabschnitte

Im Gegensatz zur Ada83-Implementierung werden wir für die Verwaltung der Blockabschnit-

te keine Tasks sondern protected objects einsetzen; d.h. jeder Blockabschnitt wird innerhalb eines protected object verwaltet. Außerdem bietet Ada95 die Möglichkeit, Task Typen zu parametrisieren. Wir nutzen dies zur Initialisierung der Tasks und ersparen uns somit ein Rendezvous zur Initialisierung wie es in der Ada83-Implementierung notwendig ist.

### 4.2.2 Programm

```
with Calendar;
use  Calendar;

generic
   Max_Station, Max_Kabine, Max_Platz, Max_Passagier : Positive;
package Kabinenbahn is
   procedure Start;
end Kabinenbahn;


package body Kabinenbahn is
   type Richtungen is (Aufwärts, Abwärts);
   subtype Stationen is Integer range 1 .. Max_Station;
   subtype Kabinen is Integer range 1 .. Max_Kabine;
   subtype Kabinen0 is Integer range 0 .. Max_Kabine;
   subtype Plätze is Integer range 1 .. Max_Platz;
   subtype Plätze0 is Integer range 0 .. Max_Platz;
   subtype Passagiere is Integer range 1 .. Max_Passagier;
   subtype Passagiere0 is Integer range 0 .. Max_Passagier;

   type Bahnsteige is record
      Richtung : Richtungen;
      Station  : Stationen;
   end record;

   type Wünsche is record
      Von, Nach : Bahnsteige;
      Zahl : Plätze0;
      Passagier : Passagiere0;
   end record;

   package Streckenbestimmung is
      procedure Nachfolger (Bahnsteig : in out Bahnsteige);
      procedure Vorgänger (Bahnsteig : in out Bahnsteige);
   end Streckenbestimmung;
   use Streckenbestimmung;

   --Spezifikation der Tasktypen und des protected type
   task type Kabinensteuerung (Kabine : Kabinen; Richtung : Richtungen;
                               Station : Stationen) is
      entry Fahrtwunsch (Wunsch : in Wünsche; Ok : out Boolean);
   end Kabinensteuerung;

   protected type Abschnittskontrolle (Kabine : Kabinen0) is
```

```
   entry Strecke_Ein (Kabine : in Kabinen);
   entry Bahnhof_Ein (Kabine : in Kabinen);
   entry Strecke_Frei;
   procedure Strecke_Aus;
   procedure Bahnhof_Aus;
   function Kabine_Auf_Strecke return Kabinen0;
   function Kabine_Im_Bahnhof return Kabinen0;
private
  Imbahnhof  : Kabinen0 := Kabine;
  Aufstrecke : Kabinen0 := 0;
  Ausfahrabsicht : Boolean := False;
end Abschnittskontrolle;

task type Kartenverkauf (Station : Stationen) is
   entry Karte_Kaufen (Ziel : in Stationen; Zahl : in Plätze;
                       Passagier : in Passagiere; Ok : out Boolean);
end Kartenverkauf;

task type Taxifahren (Passagier : Passagiere) is
   entry Einsteigen;
   entry Aussteigen;
end Taxifahren;

--Definition der Zeigertypen auf Tasktypen bzw. protected type
type Zeiger_Auf_Kabinensteuerung is access Kabinensteuerung;
type Zeiger_Auf_Abschnittskontrolle is access Abschnittskontrolle;
type Zeiger_Auf_Kartenverkauf is access Kartenverkauf;
type Zeiger_Auf_Taxifahren is access Taxifahren;

--Definition von Tasks bzw. protected objects
Kabinentaxi : array (Kabinen) of Zeiger_Auf_Kabinensteuerung;
Abschnitt : array (Richtungen, Stationen) of Zeiger_Auf_Abschnittskontrolle;
Automat : array (Stationen) of Zeiger_Auf_Kartenverkauf;
Fahrgast : array (Passagiere) of Zeiger_Auf_Taxifahren;

--Initialisierungsprozeduren
procedure Automaten_Initialisieren;
procedure Passagiere_Initialisieren;
procedure Kabinen_Verteilen;

--Start sorgt für den Start des Systems
procedure Start is
begin
   Kabinen_Verteilen;
   Automaten_Initialisieren;
   Passagiere_Initialisieren;
end;

--Implementierung des Pakets Streckenbestimmung
package body Streckenbestimmung is
   function Differenz (Richtung : in Richtungen) return Integer is
```

```
   begin
      if Richtung = Aufwärts then
         return 1;
      else
         return -1;
      end if;
   end Differenz;

   procedure Nachfolger (Bahnsteig : in out Bahnsteige) is
   begin
      if Bahnsteig.Richtung = Aufwärts and Bahnsteig.Station = Max_Station
      then
         Bahnsteig.Richtung := Abwärts;
      elsif
        Bahnsteig.Richtung = Abwärts and Bahnsteig.Station = 1
      then
         Bahnsteig.Richtung := Aufwärts;
      else
         Bahnsteig.Station := Bahnsteig.Station +
           Differenz (Bahnsteig.Richtung);
      end if;
   end Nachfolger;

   procedure Vorgänger (Bahnsteig : in out Bahnsteige) is
   begin
      if Bahnsteig.Richtung = Abwärts and Bahnsteig.Station = Max_Station
      then
         Bahnsteig.Richtung := Aufwärts;
      elsif
        Bahnsteig.Richtung = Aufwärts and Bahnsteig.Station = 1
      then
         Bahnsteig.Richtung := Abwärts;
      else
         Bahnsteig.Station := Bahnsteig.Station -
           Differenz (Bahnsteig.Richtung);
      end if;
   end Vorgänger;
end Streckenbestimmung;

--Diese Prozedur wird sowohl in Kabinensteuerung als auch in Kartenverkauf
--aufgerufen und muß deshalb global zu diesen sein
procedure Kabine_Fragen (Kabine : in Kabinen0;
                         Wunsch : in Wünsche; Ok : out Boolean) is
   Anfragezeit : constant Duration := 5.0;
begin
   Ok := False;
   if Kabine > 0 then
      select
         Kabinentaxi(Kabine).Fahrtwunsch (Wunsch, Ok);
      or
         delay Anfragezeit;
```

```
      end select;
   end if;
end Kabine_Fragen;


--Hier folgen die Implementierungen der Task Typen
task body Kabinensteuerung is
   type Personen is array (Plätze) of Passagiere;
   type Fahrgastwechsel is record
      Zahl : Plätze0 := 0;
      Passagier : Personen;
   end record;

   type Bahnsteigdaten is record
      Halten : Boolean := False;
      Aussteiger, Einsteiger : Fahrgastwechsel;
      Abfahrer : Plätze0;
   end record;

   type Halbzyklusdaten is array (Stationen) of Bahnsteigdaten;
   type Zyklusdaten is array (Richtungen) of Halbzyklusdaten;

   Zyklus : Zyklusdaten;
   Nächster_Halbzyklus : Halbzyklusdaten;
   Diese_Kabine : Kabinen;
   Hier : Bahnsteige;

   procedure Aussteigen_Lassen is
      Anzahl : Plätze0;
   begin
      Anzahl := Zyklus (Hier.Richtung)(Hier.Station).Aussteiger.Zahl;
      for I in 1 .. Anzahl loop  --Rendezvous mit Fahrgästen
         Fahrgast (Zyklus(Hier.Richtung)(Hier.Station).
                     Aussteiger.Passagier(I)).Aussteigen;
      end loop;
      Zyklus (Hier.Richtung)(Hier.Station).Aussteiger.Zahl := 0;
   end Aussteigen_Lassen;

   procedure Einsteigen_Lassen is
      Anzahl : Plätze0;
   begin
      Anzahl := Zyklus (Hier.Richtung)(Hier.Station).Einsteiger.Zahl;
      for I in 1 .. Anzahl loop  --Rendezvous mit Fahrgästen
         Fahrgast (Zyklus(Hier.Richtung)(Hier.Station).
                     Einsteiger.Passagier(I)).Einsteigen;
      end loop;
      Zyklus (Hier.Richtung)(Hier.Station).Einsteiger.Zahl := 0;
   end Einsteigen_Lassen;

   function Weiterfahren (Hier : in Bahnsteige; Zyklus : in Zyklusdaten;
              Nächster_Halbzyklus : in Halbzyklusdaten) return Boolean is
```

```
begin
   for I in Stationen loop
      for J in Richtungen loop
         if I /= Hier.Station or J /= Hier.Richtung then
            if Zyklus(J)(I).Halten then
               return True;
            end if;
         end if;
      end loop;
      if Nächster_Halbzyklus(I).Halten then
         return True;
      end if;
   end loop;
   return False;
end Weiterfahren;

function Beförderbar (Wunsch : in Wünsche; Daten : in Halbzyklusdaten)
      return Boolean is
   Bahnsteig : Bahnsteige := Wunsch.Von;
begin
   while Bahnsteig /= Wunsch.Nach loop
      if Daten(Bahnsteig.Station).Abfahrer + Wunsch.Zahl > Max_Platz
      then
         return False;
      end if;
      Nachfolger(Bahnsteig);
   end loop;
   return True;
end Beförderbar;

procedure Befördern (Wunsch : in Wünsche;
                     Daten : in out Halbzyklusdaten) is
   Bahnsteig : Bahnsteige := Wunsch.Von;
   procedure Fahrgastwechsel_Registrieren
                (Wechsel : in out Fahrgastwechsel) is
   begin
      for I in 1 .. Wunsch.Zahl loop
         Wechsel.Passagier(Wechsel.Zahl+I) := Wunsch.Passagier;
      end loop;
      Wechsel.Zahl := Wechsel.Zahl + Wunsch.Zahl;
   end Fahrgastwechsel_Registrieren;
begin
   --Einstiegspunkt;
   Daten(Bahnsteig.Station).Halten := True;
   Fahrgastwechsel_Registrieren(Daten(Bahnsteig.Station).Einsteiger);
   --gesamte Fahrstrecke
   while Bahnsteig /= Wunsch.Nach loop
      Daten(Bahnsteig.Station).Abfahrer :=
                Daten(Bahnsteig.Station).Abfahrer + Wunsch.Zahl;
      Nachfolger(Bahnsteig);
   end loop;
```

```
   --Ausstiegspunkt
   Daten(Bahnsteig.Station).Halten := True;
   Fahrgastwechsel_Registrieren(Daten(Bahnsteig.Station).Aussteiger);
end Befördern;

procedure Wunsch_Registrieren (Wunsch : in Wünsche; Ok : out Boolean) is
   function Anderer_Zyklus (Bahnsteig, Hier : in Bahnsteige)
            return Boolean is
   begin
      if Bahnsteig.Richtung /= Hier.Richtung then
         return False;
      elsif Bahnsteig.Station = Hier.Station then
         return not Zyklus(Hier.Richtung)(Hier.Station).Halten;
      elsif Bahnsteig.Richtung = Aufwärts then
         return Bahnsteig.Station < Hier.Station;
      else
         return Bahnsteig.Station > Hier.Station;
      end if;
   end Anderer_Zyklus;
begin
   Ok := False;
   if Wunsch.Zahl = 0 then
      --Leerfahrten sind immer erlaubt
      Zyklus(Wunsch.Nach.Richtung)(Wunsch.Nach.Station).Halten := True;
      Ok := True;
   else
      --Start- und Zielrichtung sind immer gleich
      if Anderer_Zyklus(Wunsch.Von, Hier) then
         --Fahrtwunsch betrifft Nächster_Halbzyklus
         if Beförderbar(Wunsch, Nächster_Halbzyklus) then
            Befördern(Wunsch, Nächster_Halbzyklus);
            Ok := True;
         end if;
         else
            --Fahrtwunsch betrifft diesen Zyklus
            if Beförderbar(Wunsch, Zyklus(Wunsch.Von.Richtung)) then
               Befördern (Wunsch, Zyklus(Wunsch.Von.Richtung));
               Ok := True;
            end if;
      end if;
   end if;
end Wunsch_Registrieren;

procedure In_Den_Nächsten_Abschnitt is
   Dort : Bahnsteige := Hier;

   procedure Richtungswechsel is
   begin
      Zyklus(Hier.Richtung) := Nächster_Halbzyklus;
      for I in Stationen loop
         Nächster_Halbzyklus(I).Halten := False;
```

```
      Nächster_Halbzyklus(I).Abfahrer := 0;
      Nächster_Halbzyklus(I).Einsteiger.Zahl := 0;
      Nächster_Halbzyklus(I).Aussteiger.Zahl := 0;
   end loop;
end Richtungswechsel;

procedure Strecke_Prüfen is
   Anfragezeit : constant Duration := 30.0;
begin
   Strecke_Besetzt:
   loop
      select
         Abschnitt(Hier.Richtung, Hier.Station).Strecke_Frei;
         exit Strecke_Besetzt;
      or
         delay Anfragezeit;
      end select;
   end loop Strecke_Besetzt;
end Strecke_Prüfen;

procedure Bahnhof_Einfahren is
   Wiederholzeit : constant Duration := 90.0;
   Wunsch : Wünsche;
   Ok : Boolean;

   procedure Leerfahrtwunsch_Zusammenstellen is
      Bahnsteig : Bahnsteige;
   begin
      Wunsch.Von := Dort;
      Bahnsteig := Dort;     --blockierter Bahnsteig
      Nachfolger(Bahnsteig); --Bahnsteig dahinter
      Wunsch.Nach := Bahnsteig;
      Wunsch.Zahl := 0;
      Wunsch.Passagier := 0;
   end Leerfahrtwunsch_Zusammenstellen;

begin
   Einfahrerlaubnis_Bekommen:
   loop
      select
         Abschnitt(Dort.Richtung, Dort.Station).
                                  Bahnhof_Ein(Diese_Kabine);
         exit Einfahrerlaubnis_Bekommen;
      or
         delay Wiederholzeit;
         --Kabine blockiert Bahnhof
         Leerfahrtwunsch_Zusammenstellen;
         Kabine_Fragen (Abschnitt(Dort.Richtung, Dort.Station).
                        Kabine_Im_Bahnhof, Wunsch, Ok);
      end select;
   end loop Einfahrerlaubnis_Bekommen;
```

```
      end Bahnhof_Einfahren;

   begin  --In_Den_Nächsten_Abschnitt
      Nachfolger(Dort);
      if Zyklus(Hier.Richtung)(Hier.Station).Halten then
         Strecke_Prüfen; --Kollision vermeiden
      end if;
      --Einfahrerlaubnis für den kommenden Abschnitt
      if Zyklus(Dort.Richtung)(Dort.Station).Halten then
         Bahnhof_Einfahren;
      else
         Abschnitt(Dort.Richtung, Dort.Station).Strecke_Ein(Diese_Kabine);
      end if;
      --Ausfahrerlaubnis für den aktuellen Abschnitt
      if Zyklus(Hier.Richtung)(Hier.Station).Halten then
         Zyklus(Hier.Richtung)(Hier.Station).Halten := False;
         Abschnitt(Hier.Richtung, Hier.Station).Bahnhof_Aus;
      else
         Abschnitt(Hier.Richtung, Hier.Station).Strecke_Aus;
      end if;
      Zyklus(Hier.Richtung)(Hier.Station).Abfahrer := 0;
      if Hier.Richtung /= Dort.Richtung then
         Richtungswechsel;
      end if;
      Hier := Dort;
   end In_Den_Nächsten_Abschnitt;

begin  --task body Kabinensteuerung
   --Beginn Initialisieren
   Diese_Kabine := Kabine;
   Hier.Richtung := Richtung;
   Hier.Station := Station;
   Zyklus (Hier.Richtung)(Hier.Station).Halten := True;
   --Ende Initialisieren
   loop
      if Zyklus (Hier.Richtung)(Hier.Station).Halten then
         declare
            Haltezeit : constant Duration := 90.0;
            Resthaltezeit : Duration;
            Ausfahrzeitpunkt : Time;
            Losfahren, Erfolg : Boolean;
         begin
            Ausfahrzeitpunkt := Clock + Haltezeit;
            Aussteigen_Lassen;
            Losfahren := Weiterfahren (Hier, Zyklus, Nächster_Halbzyklus);
            Fahrtwünsche_Im_Bahnhof:
            loop
               Resthaltezeit := Ausfahrzeitpunkt - Clock;
               select
                  accept Fahrtwunsch (Wunsch : in Wünsche;
                                      Ok : out Boolean) do
```

```
                        Wunsch_Registrieren (Wunsch, Erfolg);
                        Ok := Erfolg;
                     end Fahrtwunsch;
                     if Erfolg then
                        Losfahren := True;
                     end if;
                     exit Fahrtwünsche_Im_Bahnhof
                          when Losfahren and Resthaltezeit < 0.0;
                  or
                     delay Resthaltezeit;
                     exit Fahrtwünsche_Im_Bahnhof when Losfahren;
                  end select;
               end loop Fahrtwünsche_Im_Bahnhof;
               Einsteigen_Lassen;
            end;
         else
            declare
               Fahrzeit : constant Duration := 30.0;
               Restfahrzeit : Duration;
               Ausfahrzeitpunkt : Time;
            begin
               Ausfahrzeitpunkt := Clock + Fahrzeit;
               Fahrtwünsche_Auf_Strecke:
               loop
                  Restfahrzeit := Ausfahrzeitpunkt - Clock;
                  select
                     accept Fahrtwunsch (Wunsch : in Wünsche;
                                         Ok : out Boolean) do
                        Wunsch_Registrieren (Wunsch, Ok);
                     end Fahrtwunsch;
                     exit Fahrtwünsche_Auf_Strecke when Restfahrzeit < 0.0;
                  or
                     delay Restfahrzeit;
                     exit Fahrtwünsche_Auf_Strecke;
                  end select;
               end loop Fahrtwünsche_Auf_Strecke;
            end;
         end if;
         In_Den_Nächsten_Abschnitt;
      end loop;
   end Kabinensteuerung;

   protected body Abschnittskontrolle is
      entry Strecke_Ein (Kabine : in Kabinen) when not Ausfahrabsicht and
             Aufstrecke = 0 is
      begin
         Aufstrecke := Kabine;
         --Weiche in Richtung Strecke stellen
      end Strecke_Ein;

      entry Bahnhof_Ein (Kabine : in Kabinen) when Imbahnhof = 0 is
```

```
      begin
         Imbahnhof := Kabine;
      end Bahnhof_Ein;

      entry Strecke_Frei when Aufstrecke = 0 is
      begin
         Ausfahrabsicht := True;
      end Strecke_Frei;

      procedure Strecke_Aus is
      begin
         Aufstrecke := 0;
      end Strecke_Aus;

      procedure Bahnhof_Aus is
      begin
         Imbahnhof := 0;
         Ausfahrabsicht := False;
      end Bahnhof_Aus;

      function Kabine_Auf_Strecke return Kabinen0 is
      begin
         return Aufstrecke;
      end Kabine_Auf_Strecke;

      function Kabine_Im_Bahnhof return Kabinen0 is
      begin
         return Imbahnhof;
      end Kabine_Im_Bahnhof;
   end Abschnittskontrolle;

   task body Kartenverkauf is
      Diese_Station : Stationen := Station;
      Wunsch : Wünsche;

      procedure Fahrtwunsch_Zusammenstellen (Ziel : in Stationen;
                     Zahl : in Plätze; Passagier : in Passagiere) is
         Richtung : Richtungen;
      begin
         if Diese_Station < Ziel then
            Richtung := Aufwärts;
         else
            Richtung := Abwärts;
         end if;
         Wunsch.Von.Richtung := Richtung;
         Wunsch.Von.Station := Diese_Station;
         Wunsch.Nach.Richtung := Richtung;
         Wunsch.Nach.Station := Ziel;
         Wunsch.Zahl := Zahl;
         Wunsch.Passagier := Passagier;
      end Fahrtwunsch_Zusammenstellen;
```

```
   procedure Kabine_Suchen (Ok : out Boolean) is
      Bahnsteig : Bahnsteige;
      Erfüllt : Boolean;
   begin
      Bahnsteig := Wunsch.Von;
      Kabine_Suchen:
      loop
         Kabine_Fragen (Abschnitt(Bahnsteig.Richtung, Bahnsteig.Station).
                        Kabine_Im_Bahnhof, Wunsch, Erfüllt);
         if not Erfüllt then
            Vorgänger(Bahnsteig);
            Kabine_Fragen (Abschnitt(Bahnsteig.Richtung, Bahnsteig.Station).
                           Kabine_Auf_Strecke, Wunsch, Erfüllt);
         end if;
         exit Kabine_Suchen when Erfüllt or Bahnsteig = Wunsch.Von;
      end loop Kabine_Suchen;
      Ok := Erfüllt;
   end Kabine_Suchen;

begin  --task body Kartenverkauf
   loop
      accept Karte_Kaufen (Ziel : in Stationen; Zahl : in Plätze;
                           Passagier : in Passagiere; Ok : out Boolean) do
         if Diese_Station = Ziel then
            Ok := False;
         else
            Fahrtwunsch_Zusammenstellen (Ziel, Zahl, Passagier);
            Kabine_Suchen (Ok);
         end if;
      end Karte_Kaufen;
   end loop;
end Kartenverkauf;

task body Taxifahren is
   Von, Nach : Stationen;
   Zahl : Plätze;
   Dieser_Passagier : Passagiere := Passagier;
   Ok : Boolean;
begin
   loop
      --Hier müßte die Eingabe der Werte für Von, Nach und Plätze erfolgen
      Automat(Von).Karte_Kaufen(Nach, Zahl, Dieser_Passagier, Ok);
      if Ok then
         --Einsteigen der Passagiere
         for I in 1 .. Zahl loop
            accept Einsteigen;
         end loop;
         --Aussteigen der Passagiere
         for I in 1 .. Zahl loop
            accept Aussteigen;
```

```
            end loop;
         else
            --Fahrtwunsch ist nicht erfüllbar
            null;
         end if;
      end loop;
   end Taxifahren;

   --Implementierung der Initialisierungsprozeduren
   procedure Automaten_Initialisieren is
   begin
      for I in Stationen loop
         Automat(I) := new Kartenverkauf(I);
      end loop;
   end Automaten_Initialisieren;

   procedure Passagiere_Initialisieren is
   begin
      for I in Passagiere loop
         Fahrgast(I) := new Taxifahren(I);
      end loop;
   end Passagiere_Initialisieren;

   procedure Kabinen_Verteilen is
      Bahnsteig, Anfang : Bahnsteige;
      Kabine : Kabinen0 := 0;
   begin
      Anfang.Richtung := Aufwärts;
      Anfang.Station := 1;
      Bahnsteig := Anfang;
      Bahnhöfe_Besetzen:
      loop  --es gibt mindestens einen Bahnsteig mehr als Kabinen
        if Kabine < Max_Kabine then
           Kabine := Kabine + 1;
           Kabinentaxi(Kabine) := new Kabinensteuerung
                 (Kabine, Bahnsteig.Richtung, Bahnsteig.Station);
           Abschnitt (Bahnsteig.Richtung, Bahnsteig.Station) := new
                 Abschnittskontrolle (Kabine);
        else --keine Kabine auf diesem Bahnsteig
           Abschnitt (Bahnsteig.Richtung, Bahnsteig.Station) := new
                 Abschnittskontrolle (0);
        end if;
        Nachfolger (Bahnsteig);
        exit Bahnhöfe_Besetzen when Bahnsteig = Anfang;
      end loop Bahnhöfe_Besetzen;
   end Kabinen_Verteilen;

end Kabinenbahn;
```

**Abbildung 4.4** Die Kabinenbahn in Ada95

Da in [HH 94: 421] eine ausführliche Erläuterung der Ada83-Lösung gegeben wird, beschränken wir uns bei der Erläuterung des Programmtextes auf die Elemente, die geändert wurden.

Zunächst muß auf einen Fehler in der Ada83-Lösung hingewiesen werden: Die Spezifikation des Paketes `Kabinenbahn` darf nicht leer sein, sondern muß mindestens die Definition einer Komponente enthalten. Deshalb wird die Prozedur `Start` als Komponente dieses Paketes spezifiziert.

Da es nicht möglich ist, Felder von parametrisierten Typen (ohne Vorbesetzung der Diskriminante) zu definieren, verwenden wir Verweistypen auf parametrisierte Typen — in diesem Fall Tasktypen und protected type. Die Angabe der Diskriminante erfolgt dann bei der Erzeugung der Objekte; dies geschieht in den Initialisierungsprozeduren.

Die wichtigste Änderung betrifft die Verwaltung der Blockabschnitte. Da ein Blockabschnitt eine kritische Ressource darstellt, ergab sich der Ansatz, dafür ein protected object — statt eines Task — einzusetzen. Dabei wurden `Strecke_Ein`, `Bahnhof_Ein` und `Strecke_Frei` jeweils als protected entry realisiert (da ihre Ausführung an eine Bedingung geknüpft ist), während `Strecke_Aus` und `Bahnhof_Aus` als protected procedure implementiert wurden. Außerdem enthält das protected object noch die protected functions `Kabine_Auf_Strecke` und `Kabine_Im_Bahnhof` statt der entries `FRAGEN_AUF_STRECKE` bzw. `FRAGEN_IM_BAHNHOF` aus der Ada83-Lösung. Der Grund ist folgender: Die entries `FRAGEN_AUF_STRECKE` bzw. `FRAGEN_IM_BAHNHOF` des Task `ABSCHNITT(I)` aus der Ada83-Lösung dienen dazu, an die Kabine K, die sich auf der Strecke oder im Bahnhof des Blockabschnitts `I` befindet, einen Fahrtwunsch zu richten. Das bedeutet, innerhalb des accept-Rumpfes von `FRAGEN_AUF_STRECKE` bzw. `FRAGEN_IM_BAHNHOF` erfolgt (mittels `KABINE_FRAGEN`) ein Aufruf des entry `FAHRTWUNSCH` von `KABINENTAXI(K)`. Da der Aufruf eines entry aber eine sogenannte "potentially blocking operation" darstellt, darf der Aufruf von `Fahrtwunsch` in unserem Fall nicht innerhalb des protected object erfolgen. Deshalb gehen wir folgendermaßen vor:

Das protected object `Abschnitt(I)` enthält die protected functions `Kabine_Auf_Strecke` und `Kabine_Im_Bahnhof`, die die Nummer derjenigen Kabine K zurückliefern, die gerade auf der Strecke bzw. im Bahnhof des Blockabschnitts `I` ist. Jeder Task, der den entry `Fahrtwunsch` von K aufrufen will, holt sich mittels des Aufrufs einer der beiden Funktionen die Nummer von K und führt den Aufruf des entry (mittels `Kabine_Fragen`) selber aus. Da sowohl Kabinen mit Kabinen als auch Automaten mit Kabinen kommunizieren, muß `Kabine_Fragen` nun global definiert werden.

### 4.2.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.5 vergeben.

#### Programmgröße

Die Programmgröße beträgt 491 nloc.

#### Kommunikationsaufwand

Es ist kein Kommunikationsaufwand notwendig.

#### Lebendigkeit

Bei der Lebendigkeit muß die Frage untersucht werden, ob jede Gruppe von Passagieren, die von Bahnhof A nach Bahnhof B will, auch irgendwann auf Bahnhof B ankommt. Betrachten wir dazu Abbildung 4.5, in der ein typischer Ablauf dargestellt ist.

Nachdem eine Gruppe P von Passagieren die Eingabedaten (aktueller Bahnhof, Zielbahnhof, Größe der Gruppe) bestimmt hat, beantragt P ein Rendezvous mit dem Automaten A des aktuellen Bahnhofs. Damit wird P so lange blockiert, bis das Rendezvous mit A abgearbeitet ist, d.h. bis

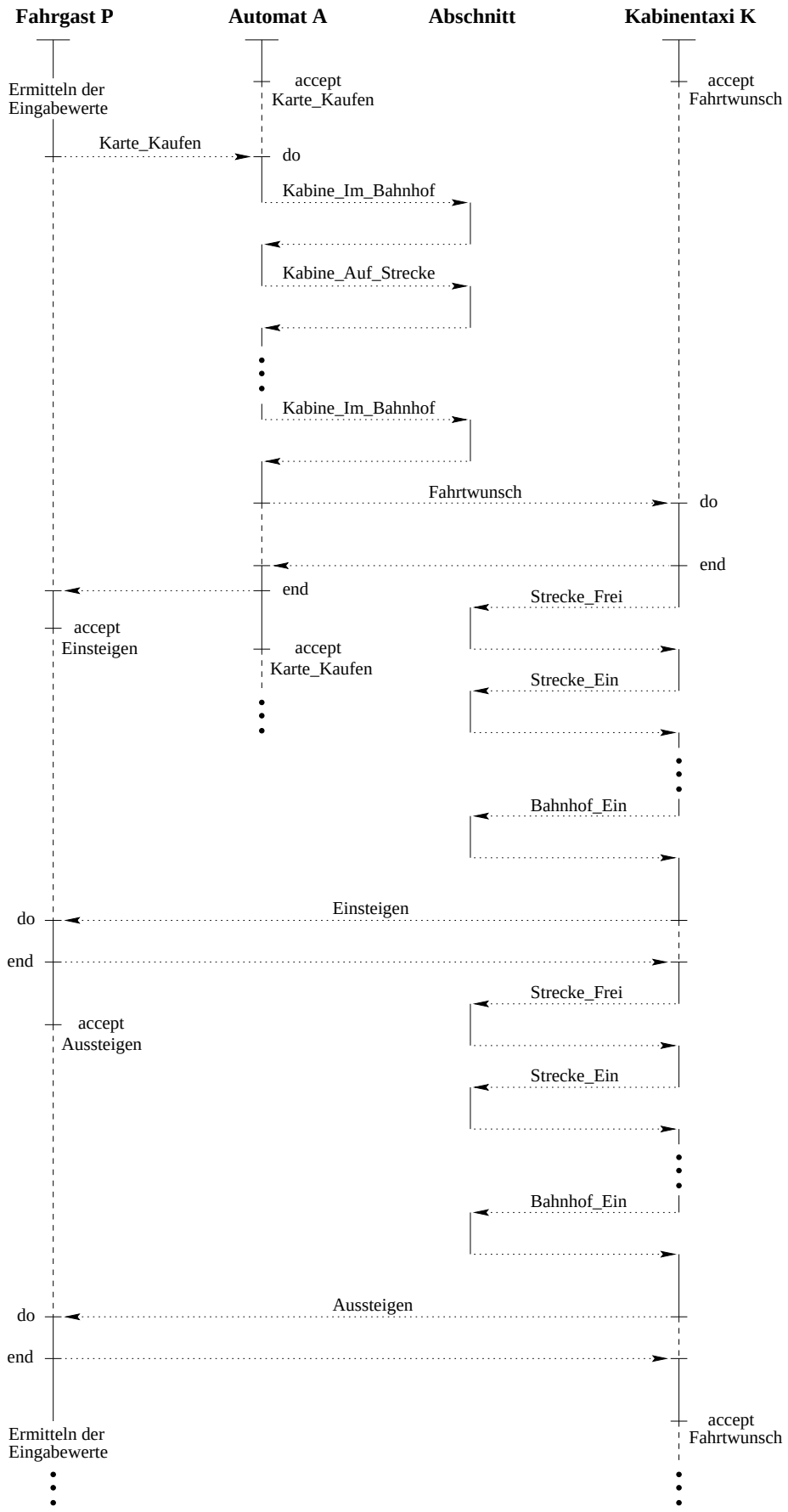


**Abbildung 4.5** Beförderung eines Passagiers

der `accept`-Rumpf von A beendet ist. Innerhalb des `accept`-Rumpfes von A wird eine Kabine zur Beförderung von P gesucht. Dazu muß nacheinander jeder Abschnitt gefragt werden, ob er eine Kabine enthält. Sobald eine Kabine K in einem Abschnitt gefunden wurde, wird (innerhalb des `accept`-Rumpfes von A) ein Rendezvous mit K beantragt, um K den Fahrtwunsch von P zu übermitteln. Damit wird A blockiert. Wir gehen der Einfachheit wegen davon aus, daß K noch genügend freie Plätze hat, um P zu befördern; d.h. K nimmt den Fahrtwunsch an. Nach Ende des Rendezvous mit K wird A reaktiviert und beendet seinen `accept`-Rumpf. Damit wird P reaktiviert und steht dann an seinem `accept`-Rumpf `Einsteigen`; d.h. P muß nun warten, bis von K ein Anruf zu `Einsteigen` kommt. Falls K — wie in Abbildung 4.5 — nicht in dem Bahnhof ist, in dem sich P befindet, muß K zunächst bis zu diesem Bahnhof fahren. Dazu muß K die protected entries und protected procedures derjenigen Abschnitte, die K passieren möchte, ausführen. Sobald K im aktuellen Bahnhof von P angekommen ist, ruft K den entry `Einsteigen` von P auf und wird so lange blockiert, bis P seinen `accept`-Rumpf ausgeführt hat. Danach steht P am `accept`-Rumpf seines entry `Aussteigen` und muß so lange warten, bis K diesen entry aufruft. K beginnt nun mit der Beförderung von P, d.h. K fährt zum Zielbahnhof von P und muß dabei wieder die protected entries bzw. protected procedures der zu passierenden Abschnitte ausführen. Nachdem K im Zielbahnhof von P angekommen ist, ruft K den entry `Aussteigen` von P auf und wird wieder so lange blockiert, bis P den `accept`-Rumpf von `Aussteigen` ausgeführt hat. Danach beginnt der ganze Ablauf von vorne.

Wir müssen nun diejenigen Stellen im Ablauf suchen, an denen es zu Wartesituationen kommt bzw. an denen Akteure übergangen werden können. Wartesituationen können auftreten beim

1. Aufruf eines Task-entry
2. Aufruf einer Operation eines protected object.

Im ersten Fall werden Tasks, die nicht unmittelbar bedient werden können, in die Warteschlange des entry eingereiht. Die Ordnung innerhalb der Warteschlange stellt sicher, daß solche Tasks nicht übergangen werden können. Das heißt, im ersten Fall ist das Programm lebendig.

Im zweiten Fall müssen wieder — wie im Kapitel 3 — zwei Ebenen unterschieden werden. Auf der ersten Ebene (beim Wettbewerb der Tasks um das dem protected object zugehörige Lock) kann es passieren, daß einzelne Tasks ständig übergangen werden. Auf der zweiten Ebene wird durch die Mechanismen des protected entry Lebendigkeit garantiert.

Das bedeutet, daß das Programm bezüglich der ersten Ebene nicht lebendig, bezüglich der zweiten Ebene aber lebendig ist.

### Verklemmung

Zu Verklemmungen kann es nur kommen, wenn Akteure anhalten. Da wir das aber ausschließen, kann Verklemmung nicht auftreten.

### Aushungern

Auf der ersten Ebene ist Aushungern von Akteuren möglich. Stellen wir uns dazu folgendes Szenario vor:

Gegeben sei eine Kabine K, die im Bahnhof $B_A$ des Abschnitt A steht und auf Fahrtwünsche wartet. Alle anderen Kabinen transportieren ständig Passagiere von Bahnhöfen $B_{A-m}$, die vor A liegen, zu Bahnhöfen $B_{A+n}$, die hinter A liegen. Das bedeutet, diese Kabinen durchfahren alle die Strecke des Abschnitt A. Wenn K nun irgendwann losfahren will (weil ein Fahrauftrag vorliegt und die Verweilzeit im Bahnhof abgelaufen ist), so ruft K den protected entry `Strecke_Frei` des zu A gehörenden protected object auf, um seinen Wunsch zum Losfahren anzuzeigen und Kollisionen zu vermeiden. Da die anderen Kabinen aber immer A durchfahren und somit den protected entry

`Strecke_Ein` bzw. die protected procedure `Strecke_Aus` ausführen, kann es passieren, daß K nie das Lock des protected object erlangen kann und somit nicht den Bahnhof verlassen kann. Damit ist K verhungert.

Auf der zweiten Ebene ist — aufgrund der Mechanismen des protected entry — Aushungern nicht möglich.

### Faire Maschine

Für die erste Ebene ist eine faire Maschine erforderlich.

### Nebenläufigkeitsgrad

Für die Betrachtung des Nebenläufigkeitsgrades wird zur Vereinfachung eine spezielle Ausführung für die Beförderung eines Passagiers betrachtet. Abbildung 4.6 stellt dies dar. Hierbei ist zusätzlich noch eine globale Zeitskala für den Gesamtablauf eingeführt worden. Dabei dauert der Gesamtablauf 25 Zeitspannen; es wird angenommen, daß jede Zeitspanne $t_i$ gleich lang ist.

Aus der graphischen Darstellung kann man nun Tabelle 4.1 ableiten. Damit läßt sich der Nebenläufigkeitsgrad leicht ausrechnen; er beträgt $\frac{28}{25} = 1.12$.

| Zeitspanne | Fahrgast | Automat | Kabinentaxi | aktive Tasks |
|---|---|---|---|---|
| $t_1$ | aktiv | blockiert | blockiert | 1 |
| $t_2$ | blockiert | aktiv | blockiert | 1 |
| $t_3$ | blockiert | aktiv | blockiert | 1 |
| $t_4$ | blockiert | aktiv | blockiert | 1 |
| $t_5$ | blockiert | aktiv | blockiert | 1 |
| $t_6$ | blockiert | aktiv | blockiert | 1 |
| $t_7$ | blockiert | aktiv | blockiert | 1 |
| $t_8$ | blockiert | aktiv | blockiert | 1 |
| $t_9$ | blockiert | blockiert | aktiv | 1 |
| $t_{10}$ | blockiert | aktiv | aktiv | 2 |
| $t_{11}$ | aktiv | aktiv | aktiv | 3 |
| $t_{12}$ | blockiert | blockiert | aktiv | 1 |
| $t_{13}$ | blockiert | blockiert | aktiv | 1 |
| $t_{14}$ | blockiert | blockiert | aktiv | 1 |
| $t_{15}$ | blockiert | blockiert | aktiv | 1 |
| $t_{16}$ | blockiert | blockiert | aktiv | 1 |
| $t_{17}$ | aktiv | blockiert | blockiert | 1 |
| $t_{18}$ | blockiert | blockiert | aktiv | 1 |
| $t_{19}$ | blockiert | blockiert | aktiv | 1 |
| $t_{20}$ | blockiert | blockiert | aktiv | 1 |
| $t_{21}$ | blockiert | blockiert | aktiv | 1 |
| $t_{22}$ | blockiert | blockiert | aktiv | 1 |
| $t_{23}$ | blockiert | blockiert | aktiv | 1 |
| $t_{24}$ | blockiert | blockiert | aktiv | 1 |
| $t_{25}$ | aktiv | blockiert | blockiert | 1 |
| Summe der aktiven Tasks | | | | 28 |

**Tabelle 4.1** Untersuchung des Nebenläufigkeitsgrades

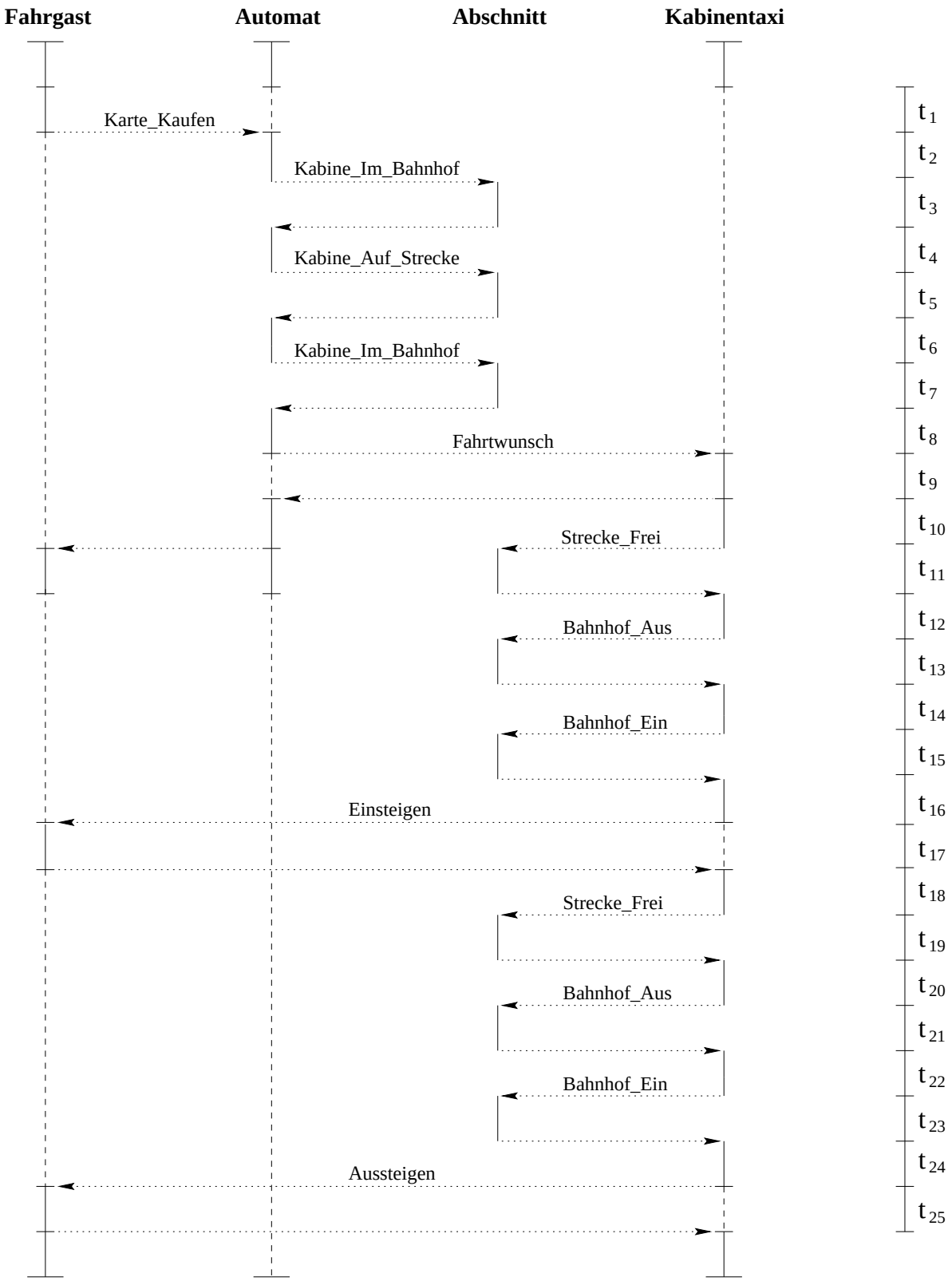


**Abbildung 4.6** Betrachtung des Nebenläufigkeitsgrades

## 4.3 Die Kabinenbahn in CHILL

### 4.3.1 Programmentwurf

In CHILL erfolgt die Kommunikation zwischen den Prozessen über zwischengeschaltete Puffer. Für die Verwaltung der Blockabschnitte werden `REGION`s eingesetzt; das heißt, jeder Blockabschnitt wird innerhalb einer `REGION` verwaltet.

### 4.3.2 Programm

```
kabinenbahn:
GENERIC
   SEIZE positive;
   SYN max_station, max_kabine, max_platz, max_passagier = positive;
MODULE
   GRANT start;
   SYNMODE richtungen  = SET (aufwaerts, abwaerts),
           stationen   = INT (1:max_station),
           kabinen     = INT (1:max_kabine),
           kabinen0    = INT (0:max_kabine),
           plaetze     = INT (1:max_platz),
           plaetze0    = INT (0:max_platz),
           passagiere  = INT (1:max_passagier),
           passagiere0 = INT (0:max_passagier);
   SYNMODE bahnsteige = STRUCT (richtung richtungen, station stationen),
           wuensche   = STRUCT (von, nach bahnsteige, zahl plaetze0,
                                passagier passagiere0);
   SYNMODE ergebnis_puffer = BUFFER (1) BOOL,
           synch_puffer = BUFFER (0) BOOL;
   SYNMODE fahrtwunsch_daten = STRUCT (wunsch wuensche,
                                       zurueck REF ergebnis_puffer),
           karte_kaufen_daten = STRUCT (ziel stationen, zahl plaetze,
                                        passagier passagiere,
                                        zurueck REF ergebnis_puffer);
   SYNMODE fahrtwunsch_puffer = BUFFER (0) fahrtwunsch_daten,
           karte_kaufen_puffer = BUFFER (0) karte_kaufen_daten,

   /* Definition der globalen Rendezvouspuffer */
   DCL fahrtwunsch ARRAY (kabinen) fahrtwunsch_puffer,
       karte_kaufen ARRAY (stationen) karte_kaufen_puffer,
       einsteigen ARRAY (passagiere) synch_puffer,
       aussteigen ARRAY (passagiere) synch_puffer;
   /* Definition der REGIONs für die Verwaltung der Blockabschnitte */
   DCL abschnitt ARRAY (richtungen, stationen) abschnittskontrolle;

   /* Prozedur zum Starten des Systems */
   start: PROC ();
      automaten_initialisieren();
      passagiere_initialisieren();
      kabinen_verteilen();
   END start;
```

```
automaten_initialisieren: PROC ();
   DO FOR i := 1 TO max_station;
      START kartenverkauf(i);
   OD;
END automaten_initialisieren;

passagiere_initialisieren: PROC ();
   DO FOR i := 1 TO max_passagier;
      START taxifahren(i);
   OD;
END passagiere_initialisieren;

kabinen_verteilen: PROC ();
   DCL bahnsteig, anfang bahnsteige,
       kabine kabinen0 := 0;
   anfang.richtung := aufwaerts;
   anfang.station := 1;
   bahnsteig := anfang;
   bahnhoefe_besetzen:
   DO FOR EVER;
      IF kabine < max_kabine THEN
         kabine := kabine + 1;
         START kabinensteuerung(kabine, bahnsteig);
         abschnitt(bahnsteig.richtung, bahnsteig.station).
            initialisieren(kabine);
      ELSE
         abschnitt(bahnsteig.richtung,bahnsteig.station).
            initialisieren(0);
      FI;
      nachfolger(bahnsteig);
      IF bahnsteig = anfang THEN
         EXIT bahnhoefe_besetzen;
      FI;
   OD;
END kabinen_verteilen;

/* Prozeduren zur Streckenbestimmung*/
differenz: PROC (richtung richtungen) RETURNS (INT);
   IF richtung = aufwaerts THEN
      RETURN 1;
   ELSE
      RETURN -1;
   FI;
END differenz;

nachfolger: PROC (bahnsteig bahnsteige INOUT);
   IF bahnsteig.richtung = aufwaerts AND
         bahnsteig.station = max_station THEN
      bahnsteig.richtung := abwaerts;
   ELSIF bahnsteig.richtung = abwaerts AND bahnsteig.station = 1 THEN
      bahnsteig.richtung := aufwaerts;
```

```
   ELSE
      bahnsteig.station := bahnsteig.station +
                              differenz(bahnsteig.richtung);
   FI;
END nachfolger;

vorgaenger: PROC (bahnsteig bahnsteige INOUT);
   IF bahnsteig.richtung = abwaerts AND
         bahnsteig.station = max_station THEN
      bahnsteig.richtung := aufwaerts;
   ELSIF bahnsteig.richtung = aufwaerts AND bahnsteig.station = 1 THEN
      bahnsteig.richtung := abwaerts;
   ELSE
      bahnsteig.station := bahnsteig.station -
                              differenz(bahnsteig.richtung);
   FI;
END vorgaenger;


/* Definition des REGIONmode für die Blockabschnitte */
SYNMODE abschnittskontrolle = REGION SPEC
   GRANT initialisieren, strecke_ein, bahnhof_ein,
         strecke_frei, strecke_aus, bahnhof_aus,
         kabine_auf_strecke, kabine_im_bahnhof;

   initialisieren: PROC(kabine kabinen0) END;
   strecke_ein: PROC(kabine kabinen) END;
   bahnhof_ein: PROC(kabine kabinen) END;
   strecke_frei: PROC() END;
   strecke_aus: PROC() END;
   bahnhof_aus: PROC() END;
   kabine_auf_strecke: PROC() RETURNS (kabinen0) END;
   kabine_im_bahnhof: PROC() RETURNS (kabinen0) END;
END abschnittskontrolle;

SYNMODE abschnittskontrolle = REGION BODY
   DCL aufstrecke, imbahnhof kabinen0 := 0,
      ausfahrabsicht BOOL := FALSE,
      freie_strecke, freier_bahnhof EVENT;

   initialisieren: PROC(kabine kabinen0);
      imbahnhof := kabine;
   END initialisieren;

   strecke_ein: PROC(kabine kabinen);
      DO WHILE ausfahrabsicht OR aufstrecke > 0;
         DELAY freie_strecke;
      OD;
      aufstrecke := kabine;
   END strecke_ein;
```

```
   bahnhof_ein: PROC(kabine kabinen);
      DO WHILE imbahnhof > 0;
         DELAY freier_bahnhof;
      OD;
      imbahnhof := kabine;
   END bahnhof_ein;

   strecke_frei: PROC();
      DO WHILE aufstrecke > 0;
         DELAY freie_strecke;
      OD;
      ausfahrabsicht := TRUE;
   END strecke_frei;

   strecke_aus: PROC();
      aufstrecke := 0;
      CONTINUE freie_strecke;
      CONTINUE freie_strecke;
      OD;
   END strecke_aus;

   bahnhof_aus: PROC();
      imbahnhof := 0;
      ausfahrabsicht := FALSE;
      CONTINUE freier_bahnhof;
   END bahnhof_aus;

   kabine_auf_strecke: PROC() RETURNS (kabinen0);
      RETURN aufstrecke;
   END kabine_auf_strecke;

   kabine_im_bahnhof: PROC() RETURNS (kabinen0);
      RETURN imbahnhof;
   END kabine_im_bahnhof;
END abschnittskontrolle;

/* Definition der Prozesstypen */

/* Prozesstyp fuer Simulation der Kabinensteuerung */
kabinensteuerung: PROCESS (diese_kabine kabinen, bahnsteig bahnsteige);
   SYNMODE personen = ARRAY (plaetze) passagiere,
           fahrgastwechsel = STRUCT (zahl plaetze0, passagier personen),
           bahnsteigdaten  = STRUCT (halten BOOL,
                                     aussteiger fahrgastwechsel,
                                     einsteiger fahrgastwechsel,
                                     abfahrer plaetze0),
           halbzyklusdaten = ARRAY (stationen) bahnsteigdaten,
           zyklusdaten     = ARRAY (richtungen) halbzyklusdaten;
   DCL zyklus zyklusdaten,
       naechster_halbzyklus halbzyklusdaten,
       hier bahnsteige;
```

```
vorgehen_im_bahnhof: PROC ();
   SYN haltezeit DURATION := SECS(90);
   DCL resthaltezeit DURATION,
       ausfahrzeitpunkt TIME,
       losfahren, erfolg BOOL,
       meine_daten fahrtwunsch_daten;
   ausfahrzeitpunkt := ABSTIME() + haltezeit;
   aussteigen_lassen();
   losfahren := weiterfahren(hier, zyklus, naechster_halbzyklus);
   fahrtwuensche_im_bahnhof:
   DO FOR EVER;
      resthaltezeit := ausfahrzeitpunkt - ABSTIME();
      AFTER resthaltezeit IN
         RECEIVE (fahrtwunsch(diese_kabine) IN meine_daten);
      TIMEOUT
         IF losfahren THEN
            EXIT fahrtwuensche_im_bahnhof;
         ELSE
            RECEIVE (fahrtwunsch(diese_kabine) IN meine_daten);
         FI;
      END;
      wunsch_registrieren(wunsch, erfolg);
      SEND meine_daten.zurueck-> (erfolg);
      IF erfolg THEN
         losfahren := TRUE;
      FI;
   OD fahrtwuensche_im_bahnhof;
   einsteigen_lassen();
END;

aussteigen_lassen: PROC ();
   DCL anzahl plaetze0;
   anzahl := zyklus(hier.richtung)(hier.station).aussteiger.zahl;
   DO FOR i := 1 TO anzahl;
      SEND aussteigen(zyklus(hier.richtung)(hier.station).
                      aussteiger.passagier(i)) (TRUE);
   OD;
   zyklus(hier.richtung)(hier.station).aussteiger.zahl := 0;
END aussteigen_lassen;

einsteigen_lassen: PROC ();
   DCL anzahl plaetze0;
   anzahl := zyklus(hier.richtung)(hier.station).einsteiger.zahl;
   DO FOR i := 1 TO anzahl;
      SEND einsteigen(zyklus(hier.richtung)(hier.station).
                      einsteiger.passagier(i)) (TRUE);
   OD;
   zyklus(hier.richtung)(hier.station).einsteiger.zahl := 0;
END einsteigen_lassen;
```

```
weiterfahren: PROC (hier bahnsteige, zyklus zyklusdaten,
                    naechster_halbzyklus halbzyklusdaten) RETURNS BOOL;
   DO FOR i IN stationen;
      DO FOR j IN richtungen;
         IF i /= hier.station OR j /= hier.richtung THEN
            IF zyklus(j)(i).halten THEN
               RETURN TRUE;
            FI;
         FI;
      OD;
      IF naechster_halbzyklus(i).halten THEN
         RETURN TRUE;
      FI;
   OD;
   RETURN FALSE;
END weiterfahren;

vorgehen_auf_strecke: PROC ();
   SYN fahrzeit DURATION := SECS(30);
   DCL restfahrzeit DURATION,
       ausfahrzeitpunkt TIME,
       meine_daten fahrtwunsch_daten,
       erfolg BOOL;
   ausfahrzeitpunkt := ABSTIME() + fahrzeit;
   fahrtwuensche_auf_strecke:
   DO FOR EVER;
      restfahrzeit := ausfahrzeitpunkt - ABSTIME();
      AFTER restfahrzeit IN
         RECEIVE (fahrtwunsch(diese_kabine) IN meine_daten);
      TIMEOUT
         EXIT fahrtwuensche_auf_strecke;
      END;
      wunsch_registrieren(meine_daten.wunsch, erfolg);
      SEND meine_daten.zurueck-> (erfolg);
      IF restfahrzeit < 0 THEN
         EXIT fahrtwuensche_auf_strecke;
      FI;
   OD fahrtwuensche_auf_strecke;
END vorgehen_auf_strecke;

wunsch_registrieren: PROC (wunsch wuensche, ok BOOL OUT);
   anderer_zyklus: PROC (bahnsteig, hier bahnsteige) RETURNS (BOOL);
      IF bahnsteig.richtung /= hier.richtung THEN
         RETURN FALSE;
      ELSIF bahnsteig.station = hier.station THEN
         RETURN NOT zyklus(hier.richtung)(hier.station).halten;
      ELSIF bahnsteig.richtung = aufwaerts THEN
         RETURN bahnsteig.station < hier.station
      ELSE
         RETURN bahnsteig.station > hier.station;
      FI;
```

```
   END anderer_zyklus;

   ok := FALSE;
   IF wunsch.zahl = 0 THEN
      /* Leerfahrten sind immer erlaubt */
      zyklus(wunsch.nach.richtung)(wunsch.nach.station).halten := TRUE;
      ok := TRUE;
   ELSE
      /* Start- und Zielrichtung sind immer gleich */
      IF anderer_zyklus(wunsch.von, hier) THEN
         /* Fahrtwunsch betrifft naechster_halbzyklus */
         IF befoerderbar (wunsch, naechster_halbzyklus) THEN
            befoerdern(wunsch, naechster_halbzyklus);
            ok := TRUE;
         FI;
      ELSE
         /* Fahrtwunsch betrifft diesen zyklus */
         IF befoerderbar(wunsch, zyklus(wunsch.von.richtung)) THEN
            befoerdern(wunsch, zyklus(wunsch.von.richtung));
            ok := TRUE;
         FI;
      FI;
   FI;
END wunsch_registrieren;

befoerderbar: PROC (wunsch wuensche, daten halbzyklusdaten)
              RETURNS (BOOL);
   DCL bahnsteig bahnsteige;
   bahnsteig := wunsch.von;
   DO WHILE bahnsteig /= wunsch.nach;
      IF daten(bahnsteig.station).abfahrer + wunsch.zahl > max_platz
      THEN RETURN FALSE;
      FI;
      nachfolger(bahnsteig);
   OD;
   RETURN TRUE;
END befoerderbar;

befoerdern: PROC (wunsch wuensche, daten halbzyklusdaten INOUT);
   fahrgastwechsel_registrieren: PROC (wechsel fahrgastwechsel INOUT);
      DO FOR i := 1 TO wunsch.zahl;
         wechsel.passagier(wechsel.zahl + i) := wunsch.passagier;
      OD;
      wechsel.zahl := wechsel.zahl + wunsch.zahl;
   END fahrgastwechsel_registrieren;

   DCL bahnsteig bahnsteige,
       platz plaetze0;
   bahnsteig := wunsch.von;
   /* Einstiegspunkt */
   daten(bahnsteig.station).halten := TRUE;
```

```
      fahrgastwechsel_registrieren(daten(bahnsteig.station).einsteiger);
      /* gesamte Fahrstrecke */
      DO WHILE bahnsteig /= wunsch.nach;
         daten(bahnsteig.station).abfahrer :=
            daten(bahnsteig.station).abfahrer + wunsch.zahl;
         nachfolger(bahnsteig);
      OD;
      /* Ausstiegspunkt */
      daten(bahnsteig.station).halten := TRUE;
      fahrgastwechsel_registrieren(daten(bahnsteig.station).aussteiger);
   END befoerdern;

   in_den_naechsten_abschnitt: PROC ();
      richtungswechsel: PROC ();
         zyklus(hier.richtung) := naechster_halbzyklus;
         DO FOR i IN stationen;
            naechster_halbzyklus(i).halten := FALSE;
            naechster_halbzyklus(i).abfahrer := 0;
            naechster_halbzyklus(i).einsteiger.zahl := 0;
            naechster_halbzyklus(i).aussteiger.zahl := 0;
         OD;
      END richtungswechsel;

      strecke_pruefen: PROC ();
         SYN anfragezeit DURATION := SECS(30);
         strecke_besetzt:
         DO FOR EVER;
            AFTER anfragezeit IN
               abschnitt(hier.richtung, hier.station).strecke_frei();
               EXIT strecke_besetzt;
            TIMEOUT
            END;
         OD strecke_besetzt;
      END strecke_pruefen;

      bahnhof_einfahren: PROC();
         leerfahrtwunsch_zusammenstellen: PROC ()
            DCL bahnsteig bahnsteige;
            wunsch.von := dort;
            bahnsteig  := dort;
            nachfolger(bahnsteig);
            wunsch.nach := bahnsteig;
            wunsch.zahl := 0;
            wunsch.passagier := 0;
         END leerfahrtwunsch_zusammenstellen;

         SYN wiederholzeit DURATION := SECS(90);
         DCL wunsch wuensche,
             ok BOOL,
             kabine kabinen;
         einfahrerlaubnis_bekommen:
```

```
      DO FOR EVER;
         AFTER wiederholzeit IN
            abschnitt(dort.richtung, dort.station).
                                    bahnhof_ein(diese_kabine);
            EXIT einfahrerlaubnis_bekommen;
         TIMEOUT
            leerfahrtwunsch_zusammenstellen();
            kabine := abschnitt(dort.richtung, dort.station).
                                              kabine_im_bahnhof();
            kabine_fragen(kabine, wunsch, ok);
         END;
      OD einfahrerlaubnis_bekommen;
   END bahnhof_einfahren;

   /* Beginn in_den_naechsten_abschnitt */
   DCL dort bahnsteige;
   dort := hier;
   nachfolger(dort);
   IF zyklus(hier.richtung)(hier.station).halten THEN
      strecke_pruefen();
   FI;
   /* Einfahrerlaubnis fuer kommenden Abschnitt */
   IF zyklus(dort.richtung)(dort.station).halten THEN
      bahnhof_einfahren();
   ELSE
      abschnitt(dort.richtung, dort.station).strecke_ein(diese_kabine);
   FI;
   /* Ausfahrmeldung fuer aktuellen Abschnitt */
   IF zyklus(hier.richtung)(hier.station).halten THEN
      zyklus(hier.richtung)(hier.station).halten := FALSE;
      abschnitt(hier.richtung, hier.station).bahnhof_aus();
   ELSE
      abschnitt(hier.richtung, hier.station).strecke_aus();
   FI;
   zyklus(hier.richtung)(hier.station).abfahrer := 0;
   IF hier.richtung /= dort.richtung THEN
      richtungswechsel();
   FI;
   hier := dort;
END in_den_naechsten_abschnitt;

/* Beginn kabinensteuerung */
hier := bahnsteig;
zyklus := [ (aufwaerts, abwaerts): [ (1:max_station):
            [.halten: FALSE, .aussteiger: [0, (1:max_passagier): 1],
             .einsteiger: [0, (1:max_passagier): 1],
             .abfahrer: 0]]];
naechster_halbzyklus := [(1:max_station):
            [.halten: FALSE, .aussteiger: [0, (1:max_passagier): 1],
             .einsteiger: [0, (1:max_passagier): 1],
             .abfahrer: 0]];
```

```
      zyklus(hier.richtung)(hier.station).halten := TRUE;
      DO FOR EVER;
         IF zyklus(hier.richtung)(hier.station).halten THEN
            vorgehen_im_bahnhof();
         ELSE
            vorgehen_auf_strecke();
         FI;
         in_den_naechsten_abschnitt();
      OD;
   END kabinensteuerung;

   /* Prozesstyp fuer Simulation der Automaten fuer den Kartenverkauf
   kartenverkauf: PROCESS (diese_station stationen);
      DCL meine_daten karte_kaufen_daten,
          mein_puffer ergebnis_puffer,
          wunsch wuensche,
          ok BOOL;
      fahrtwunsch_zusammenstellen:
      PROC (ziel stationen, zahl plaetze, passagier passagiere);
         DCL richtung richtungen;
         IF diese_station < ziel THEN
            richtung := aufwaerts;
         ELSE
            richtung := abwaerts;
         FI;
         wunsch.von.richtung := richtung;
         wunsch.von.station := diese_station;
         wunsch.nach.richtung := richtung;
         wunsch.nach.station := ziel;
         wunsch.zahl := zahl;
         wunsch.passagier := passagier;
      END fahrtwunsch_zusammenstellen;

      kabine_suchen : PROC (ok BOOL OUT);
         DCL bahnsteig bahnsteige,
             erfuellt BOOL,
             kabine kabinen,
             mein_puffer ergebnis_puffer;
         bahnsteig := wunsch.von;
         kabine_suchen:
         DO FOR EVER;
            kabine := abschnitt(bahnsteig.richtung, bahnsteig.station).
                           kabine_im_bahnhof();
            kabine_fragen(kabine, wunsch, erfuellt);
            IF NOT erfuellt THEN
               vorgaenger(bahnsteig);
               kabine := abschnitt(bahnsteig.richtung, bahnsteig.station).
                           kabine_auf_strecke();
               kabine_fragen(kabine, wunsch, erfuellt);
            FI;
            IF erfuellt OR bahnsteig = wunsch.von THEN
```

```
            EXIT kabine_suchen;
         FI;
      OD kabine_suchen;
      ok := erfuellt;
   END kabine_suchen;

   /* Beginn kartenverkauf */
   DO FOR EVER;
      RECEIVE (karte_kaufen IN meine_daten);
      IF diese_station = meine_daten.ziel THEN
         ok := FALSE;
      ELSE
         fahrtwunsch_zusammenstellen(meine_daten.ziel, meine_daten.zahl,
                                     meine_daten.passagier);
         kabine_suchen(ok);
      FI;
      SEND meine_daten.zurueck-> (ok);
   OD;
END kartenverkauf;

/* Prozesstyp fuer Simulation der Passagiere */
taxifahren: PROCESS (dieser_passagier passagiere);
   DCL von, nach stationen,
       zahl plaetze,
       ok BOOL,
       mein_puffer ergebnis_puffer,
       meine_daten karte_kaufen_daten;
   DO FOR EVER;
      /* ... Bestimmen der Werte fuer von, nach, zahl */
      meine_daten := [nach, zahl, dieser_passagier, ->mein_puffer];
      SEND karte_kaufen(von) (meine_daten);
      RECEIVE (mein_puffer IN ok);
      IF ok THEN
         /* Einsteigen */
         DO FOR i := 1 TO zahl;
            RECEIVE (einsteigen(dieser_passagier) IN ok);
         OD;
         /* Fahren */
         /* Aussteigen */
         DO FOR i := 1 TO zahl;
            RECEIVE (aussteigen(dieser_passagier) IN ok);
         OD;
      ELSE
         /* Fahrtwunsch nicht erfuellbar */
      FI;
   OD;
END taxifahren;

/* Prozedur kabine_fragen wird sowohl in kabinensteuerung als auch
   in kartenverkauf benoetigt */
kabine_fragen: PROC (kabine kabinen, wunsch wuensche, ok BOOL OUT);
```

```
      DCL mein_puffer ergebnis_puffer;
      SYN anfragezeit DURATION := SECS(5);
      AFTER anfragezeit IN
         SEND fahrtwunsch(kabine) ([wunsch, ->mein_puffer]);
      TIMEOUT
         ok := FALSE;
         RETURN;
      END;
      RECEIVE (mein_puffer IN ok);
   END kabine_fragen;
```

**Abbildung 4.7** Die Kabinenbahn in CHILL

Für jedes Rendezvous wird ein globaler Puffer definiert. Ein solcher Rendezvous-Puffer hat immer die Länge 0. Das hat folgenden Grund: Sowohl beim Senden, als auch beim Empfangen soll es möglich sein, Timeouts anzugeben. Wenn die Länge des Puffers aber größer als 0 ist, wird ein Prozeß, der in einen Puffer (der nicht voll ist) sendet, nicht blockiert — unabhängig davon, ob der Empfängerprozeß den Wert empfangen kann oder nicht. Damit ist es aber nicht mehr möglich, ein Timeout anzugeben. Das Problem läßt sich mit einem Puffer der Länge 0 lösen, denn ein solcher Puffer ist immer voll und somit wird ein Prozeß, der in einen solchen Puffer sendet, so lange blockiert, bis ein anderer Prozeß ein `RECEIVE` auf diesem Puffer ausführt. Auf diese Weise kann auch beim Senden ein Timeout angegeben werden.

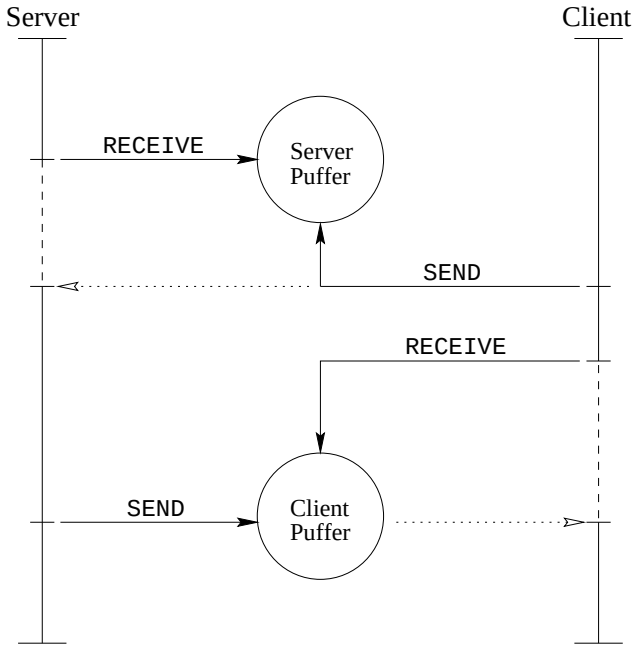


**Abbildung 4.8** Kommunikation über Puffer

Die Kommunikation zwischen einem Server-Prozeß (der ein Rendezvous anbietet) und einem Client-Prozeß (der ein Rendezvous beantragt) ist in Abbildung 4.8 dargestellt. Der Server zeigt durch das Ausführen eines `RECEIVE` bezüglich des ihm zugehörigen Puffers seine Bereitschaft zum Rendezvous an. Sollte noch kein Prozeß Daten in den Puffer gesendet haben, muß der Server warten. Ein Client, der eine Anfrage an den Server stellt, sendet seine Daten — inklusive einer Referenz auf seinen eigenen Puffer — in den Puffer des Servers. Damit wird der Server reaktiviert und kann mit der Bearbeitung der Anfrage beginnen. Da der Client auf das Ergebnis der Anfrage warten muß, führt er als nächstes ein `RECEIVE` bezüglich seines eigenen Puffers aus. Damit ist er so lange

blockiert, bis der Server die Anfrage bearbeitet und die Ergebnisdaten in den Puffer des Clients gesendet hat.

Das heißt, eine synchrone Kommunikation — also ein Rendezvous — wird durch 2 `SEND` und 2 `RECEIVE` erreicht.

Für das Einsteigen und Aussteigen der Passagiere reicht ein `SEND` und ein `RECEIVE` bezüglich eines Puffers mit Länge 0 (`synch_puffer`) aus. Der Grund ist, daß hierbei keine Daten übergeben werden, sondern nur eine kurzzeitige Synchronisation zwischen einer Kabine und einem Passagier erfolgt.

Die Verwaltung der Blockabschnitte erfolgt mit Hilfe von `REGION`s. Im zugehörigen `REGION` Typ `abschnittskontrolle` wird mittels der beiden `EVENT` Variablen `freie_strecke` und `freier_bahnhof` und der für `EVENT` Variablen definierten Operationen `DELAY` und `CONTINUE` für die richtige Koordination zwischen den Prozessen gesorgt. Dabei erfolgt in der Prozedur `strecke_aus` ein zweimaliges `CONTINUE`. Das ist notwendig, weil es zwei Möglichkeiten für eine Kabine $K$ gibt, darauf warten zu müssen, daß eine Strecke $S$ frei wird:

1. $K$ steht im zugehörigen Bahnhof $B_S$ und will aus diesem ausfahren
2. $K$ steht auf der Vorgängerstrecke $S_V$ und will in $S$ einfahren

Das bedeutet, es könnten zwei Kabinen bezüglich der `EVENT` Variable `freie_strecke` blockiert sein. Deshalb sind zwei `CONTINUE`-Anweisungen erforderlich.

### 4.3.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 3 vergeben.

#### Programmgröße

Die Programmgröße beträgt 466 nloc.

#### Kommunikationsaufwand

Bei der Betrachtung des Kommunikationsaufwandes beziehen wir uns auf die Ausführung aus Abbildung 4.6. Dabei finden 4 Rendezvous statt: `karte_kaufen`, `fahrtwunsch`, `einsteigen` und `aussteigen`. Für die beiden ersten Rendezvous sind jeweils 2 `SEND` und 2 `RECEIVE` erforderlich; für die anderen beiden Rendezvous jeweils ein `SEND` und ein `RECEIVE`. Das ergibt 12 Anweisungen. Außerdem werden folgende Prozeduren der Abschnittsverwaltung aufgerufen: 2 mal `strecke_frei` (je 1 `DELAY`), 2 mal `bahnhof_aus` (je 1 `CONTINUE`), 2 mal `bahnhof_ein` (je 1 `DELAY`), 2 mal `kabine_im_bahnhof` und einmal `kabine_auf_strecke`. Das ergibt für die Abschnittsverwaltung 6 Anweisungen.

Damit beträgt der Kommunikationsaufwand für die betrachtete Ausführung 18 Anweisungen.

#### Lebendigkeit

Da die Kommunikation zwischen den Prozessen über zwischengeschaltete Puffer erfolgt, ist das Programm nicht lebendig.

Wenn beispielsweise eine Gruppe P von Passagieren am Automat A des aktuellen Bahnhofs eine Karte kaufen will, so muß P für den Fall, daß gerade eine andere Passagiergruppe eine Karte an A kauft, warten. Das bedeutet, der zugehörige Prozeß $P_P$ wird blockiert und kommt in die Menge der blockierten Sendeprozesse des Puffers `karte_kaufen(A)`. Da es in dieser Menge keine Ordnung gibt, kann es passieren, daß $P_P$ beim Reaktivieren eines Prozesses aus dieser Menge (aufgrund eines `RECEIVE` bezüglich `karte_kaufen(A)`) immer übergangen wird. Damit wäre $P_P$ verhungert.

Für die Verwaltung der Blockabschnitte werden `REGION`s benutzt. Da es beim Wettbewerb der Prozesse um das Lock der `REGION` keine Ordnung oder Bevorzugung gibt, kann es passieren, daß Prozesse bei diesem Wettbewerb beliebig oft übergangen werden. Damit ist das Programm auf der 1. Ebene nicht lebendig. Hat ein Prozeß P das Lock bekommen, kann es passieren, daß P mittels `DELAY` blockiert wird, weil eine bestimmte Bedingung, die innerhalb der `REGION` abgefragt wird, nicht erfüllt ist. In diesem Fall muß P das Lock wieder abgeben. Wenn P reaktiviert wird, muß er zunächst wieder mit den anderen Prozessen um das Lock konkurrieren. Das bedeutet, das Programm ist auch auf der 2. Ebene nicht lebendig.

#### Verklemmung

Zu Verklemmungen kann es nur kommen, wenn Akteure anhalten. Da wir das aber ausschließen, kann Verklemmung nicht auftreten.

#### Aushungern

Aushungern von Prozessen ist auf beiden Ebenen möglich.

#### Faire Maschine

Für beide Ebenen ist eine faire Maschine erforderlich.

#### Nebenläufigkeitsgrad

Da mit Hilfe von Puffern die Kommunikation zwischen den Prozessen synchron abläuft und außerdem die Verwaltung der Blockabschnitte innerhalb von passiven Kommunikationsobjekten (`REGION`s) erfolgt, ergibt sich der gleiche Ablauf wie in Abbildung 4.6. Damit beträgt der Nebenläufigkeitsgrad — wie in der Ada-Lösung — 1.12.

## 4.4 Die Kabinenbahn in Erlang

### 4.4.1 Programmentwurf

Prozeßkommunikation läuft in Erlang auf direktem Weg über das asynchrone Senden und Empfangen von Nachrichten ab. Der für die Lösung der Aufgabe erforderliche synchrone Ablauf der Kommunikation läßt sich — wie in Kapitel 2 beschrieben — durch 2 Sendeoperationen (`!`) und 2 `receive` erreichen.

Da keine passiven Kommunikationsobjekte in Erlang existieren, muß jeder Blockabschnitt innerhalb eines eigenen Prozesses verwaltet werden.

Da Erlang als funktionale Sprache keine globalen Variablen hat, müssen alle Größen, die in einer Funktion verwendet werden, als Parameter an diese Funktion übergeben werden. Wenn aber — wie in unserem Fall — bestimmte Daten (z.B. Anzahl der Passagiere, Anzahl der Kabinen) von fast allen Funktionen benötigt werden, führt dies zu aufgeblähten Parameterlisten in den Funktionen und damit zu verringerter Übersichtlichkeit und Lesbarkeit. Um das zu umgehen, gibt es in Erlang das Konzept des sogenannten process dictionary. Jeder Erlang Prozeß besitzt ein solches process dictionary, in das — mittels der Funktion `put(Key, Value)` — Werte geschrieben und von dem — mittels `get(Key)` — Werte gelesen werden können. Damit können alle Funktionen eines Prozesses auf die Daten dieses Prozesses zugreifen.

### 4.4.2 Programm

```
-module(streckenbestimmung).
-export([vorgaenger/1, nachfolger/1]).

differenz (aufwaerts) ->
   1;
differenz (abwaerts) ->
   -1.

nachfolger ({Richtung, Station}) ->
   Max_Station = get(max_station),
   if Richtung == aufwaerts, Station == Max_Station ->
         {abwaerts, Station};
      Richtung == abwaerts, Station == 1 ->
         {aufwaerts, Station};
      true ->
         {Richtung, Station+differenz(Richtung)}
   end.

vorgaenger ({Richtung, Station}) ->
   Max_Station = get(max_station),
   if Richtung == abwaerts, Station == Max_Station ->
         {aufwaerts, Station};
      Richtung == aufwaerts, Station == 1 ->
         {abwaerts, Station};
      true ->
         {Richtung, Station-differenz(Richtung)}
   end.

-module(kabinensteuerung).
-export([kabinensteuerung/6]).
-import(kartenverkauf, [position/2, kabine_fragen/2]).
-import(streckenbestimmung, [nachfolger/1]).

kabinensteuerung (Zyklus, Naechster_Halbzyklus, Diese_Kabine, Hier,
                  Max_Platz, Max_Station) ->
   Nummer = position(Hier, Zyklus),
   Anfangszyklus = setelement(Nummer, Zyklus, {true, [], [], 0}),
   receive
      {Kabinen, Abschnitte, Passagiere} ->
         put(zyklus, Anfangszyklus),
         put(naechster_halbzyklus, Naechster_Halbzyklus),
         put(diese_kabine, Diese_Kabine),
         put(hier, Hier),
         put(max_platz, Max_Platz),
         put(max_station, Max_Station),
         put(kabinen, Kabinen),
         put(abschnitte, Abschnitte),
         put(passagiere, Passagiere),
         kabinensteuerung1()
```

```
   end.

kabinensteuerung1 () ->
   Nummer = position(get(hier), get(zyklus)),
   Halten = element(1, element(Nummer, get(zyklus))),
   if Halten == true ->
         vorgehen_im_bahnhof();
      Halten == false ->
         vorgehen_auf_strecke()
   end,
   in_den_naechsten_abschnitt(),
   kabinensteuerung1 ().

vorgehen_auf_strecke() ->
   Fahrzeit = 30000,
   Ausfahrzeitpunkt = aktuelle_zeit() + Fahrzeit,
   fahrtwuensche_auf_strecke(Ausfahrzeitpunkt).

fahrtwuensche_auf_strecke (Ausfahrzeitpunkt) ->
   Restzeit = Ausfahrzeitpunkt - aktuelle_zeit(),
   %bei Timeout darf kein negatives Argument verwendet werden
   if Restzeit < 0 ->
         Restfahrzeit = 0;
      Restzeit >= 0 ->
         Restfahrzeit = Restzeit
   end,
   receive
      {Pid, fahrtwunsch, Wunsch} ->
         Ok = wunsch_registrieren(Wunsch),
         Pid ! {self(), Ok},
         if Restfahrzeit == 0 ->
               true;
            Restfahrzeit >  0 ->
               fahrtwuensche_auf_strecke(Ausfahrzeitpunkt)
         end
   after
      Restfahrzeit ->
         true
   end.

wunsch_registrieren (Wunsch) when element(3, Wunsch) == 0 ->
   Nummer = position(element(2, Wunsch), get(zyklus)),
   Wunsch_Nach = setelement(1, element(Nummer, get(zyklus)), true),
   Zyklus = setelement(Nummer, get(zyklus), Wunsch_Nach),
   put(zyklus, Zyklus),
   true;

wunsch_registrieren (Wunsch) ->
   Anderer_Zyklus = anderer_zyklus(element(1, Wunsch), get(hier)),
   if Anderer_Zyklus == true ->
         Befoerderbar = befoerderbar(Wunsch, get(naechster_halbzyklus)),
```

```
         if Befoerderbar == true ->
               Halbzyklus = befoerdern(Wunsch, get(naechster_halbzyklus)),
               put(naechster_halbzyklus, Halbzyklus),
               true;
            Befoerderbar == false ->
               false
         end;
      Anderer_Zyklus == false ->
         Richtung = element(1, element(1, Wunsch)),
         Halbzyklus = halbzyklus(Richtung, get(zyklus)),
         Befoerderbar = befoerderbar(Wunsch, Halbzyklus),
         if Befoerderbar == true ->
               Neuer_Halzyklus = befoerdern(Wunsch, Halbzyklus),
               Neuer_Zyklus = zyklus_ergaenzen
                         (Richtung, get(zyklus), Neuer_Halzyklus),
               put(zyklus, Neuer_Zyklus),
               true;
            Befoerderbar == false ->
               false
         end
   end.

halbzyklus (aufwaerts, Zyklus)  ->
   list_to_tuple(halbzyklus(aufwaerts, 1, Zyklus));
halbzyklus (abwaerts, Zyklus) ->
   list_to_tuple(halbzyklus(abwaerts, round(size(Zyklus)/2)+1, Zyklus)).
halbzyklus (aufwaerts, Bis, Zyklus) when Bis < size(Zyklus)/2 ->
   lists:append([element(Bis, Zyklus)], halbzyklus(aufwaerts, Bis+1, Zyklus));
halbzyklus (aufwaerts, Bis, Zyklus) when Bis == size(Zyklus)/2 ->
   [element(Bis, Zyklus)];
halbzyklus (abwaerts, Bis, Zyklus) when  Bis < size(Zyklus) ->
   lists:append([element(Bis, Zyklus)], halbzyklus(abwaerts, Bis+1, Zyklus));
halbzyklus (abwaerts, Bis, Zyklus) when  Bis == size(Zyklus) ->
   [element(Bis, Zyklus)].

zyklus_ergaenzen (aufwaerts, Zyklus, Halber_Zyklus) ->
   list_to_tuple(lists:append(tuple_to_list(Halber_Zyklus),
                              tuple_to_list(halbzyklus(abwaerts, Zyklus))));
zyklus_ergaenzen (abwaerts, Zyklus, Halber_Zyklus) ->
   list_to_tuple(lists:append(tuple_to_list(halbzyklus(aufwaerts, Zyklus)),
                              tuple_to_list(Halber_Zyklus))).

anderer_zyklus (Bahnsteig, Hier)
 when element(1, Bahnsteig) /= element(1, Hier) ->
   false;
anderer_zyklus (Bahnsteig, Hier)
 when element(2, Bahnsteig) == element(2, Hier) ->
   Nummer = position (Hier, get(zyklus)),
   Halten = element (1, element(Nummer, get(zyklus))),
   if Halten == true ->
         false;
```

```
      Halten == false ->
         true
   end;
anderer_zyklus (Bahnsteig, Hier)
 when element(1, Bahnsteig) == aufwaerts ->
   if element(2, Bahnsteig) < element(2, Hier) ->
         true;
      element(2, Bahnsteig) > element(2, Hier) ->
         false
   end;
anderer_zyklus (Bahnsteig, Hier) ->
   if element(2, Bahnsteig) > element(2, Hier) ->
         true;
      element(2, Bahnsteig) < element(2, Hier) ->
         false
   end.

befoerderbar (Wunsch, Daten) ->
   Bahnsteig = element(1, Wunsch),
   befoerderbar(Bahnsteig, Wunsch, Daten).
befoerderbar (Bahnsteig, Wunsch, Daten)
 when Bahnsteig == element(2, Wunsch) ->
   true;
befoerderbar (Bahnsteig, Wunsch, Daten)
 when Bahnsteig /= element(2, Wunsch) ->
   Max_Platz = get(max_platz),
   Station = element(2, Bahnsteig),
   Abfahrer = element(4, element(Station, Daten)),
   if Abfahrer + element(3, Wunsch) > Max_Platz ->
         false;
      Abfahrer + element(3, Wunsch) =< Max_Platz ->
         befoerderbar(nachfolger(Bahnsteig), Wunsch, Daten)
   end.


befoerdern (Wunsch, Daten) ->
   Bahnsteig_Von = element(1, Wunsch),
   % Einstiegspunkt
   Station_Von = element(2, Bahnsteig_Von),
   Bahnsteig_Daten = setelement(1, element(Station_Von, Daten), true),
   Fahrgastwechsel =
     fahrgastwechsel_registrieren(element(3, Bahnsteig_Daten), Wunsch),
   Bahnsteig_Daten2 = setelement(3, Bahnsteig_Daten, Fahrgastwechsel),
   Daten2 = setelement(Station_Von, Daten, Bahnsteig_Daten2),
   % gesamte Fahrstrecke
   Daten3 = befoerdern_fahrstrecke(Bahnsteig_Von, Daten2, Wunsch),
   % Ausstiegspunkt
   Bahnsteig_Nach = element(2, Wunsch),
   Station_Nach = element(2, Bahnsteig_Nach),
   Bahnsteig_Daten3 = setelement(1, element(Station_Nach, Daten), true),
   Fahrgastwechsel2 =
```

```
     fahrgastwechsel_registrieren(element(2, Bahnsteig_Daten3), Wunsch),
   Bahnsteig_Daten4 = setelement(2, Bahnsteig_Daten3, Fahrgastwechsel2),
   setelement(Station_Nach, Daten3, Bahnsteig_Daten4).

befoerdern_fahrstrecke (Bahnsteig, Daten, Wunsch)
 when Bahnsteig /= element(2, Wunsch) ->
   Station = element(2, Bahnsteig),
   Abfahrer = element(4, element(Station, Daten)),
   Bahnsteig_Daten = setelement(4, element(Station, Daten),
                                Abfahrer + element(3, Wunsch)),
   Daten_Neu = setelement(Station, Daten, Bahnsteig_Daten),
   befoerdern_fahrstrecke (nachfolger(Bahnsteig), Daten_Neu, Wunsch);
befoerdern_fahrstrecke (Bahnsteig, Daten, Wunsch)
 when Bahnsteig == element(2, Wunsch) ->
   Daten.

fahrgastwechsel_registrieren (Wechsel, Wunsch) ->
   if element(3, Wunsch) == 0 ->
         Wechsel;
      element(3, Wunsch) >  0 ->
         registrieren(1, Wechsel, Wunsch)
   end.

registrieren (Bis, Wechsel, Wunsch) when Bis < element(3, Wunsch) ->
   Neuer_Wechsel = lists:append(Wechsel, [element(4, Wunsch)]),
   registrieren(Bis+1, Neuer_Wechsel, Wunsch);
registrieren (Bis, Wechsel, Wunsch) when Bis == element(3, Wunsch) ->
   lists:append(Wechsel, [element(4, Wunsch)]).

vorgehen_im_bahnhof() ->
   Haltezeit = 90000,
   Ausfahrzeitpunkt = aktuelle_zeit() + Haltezeit,
   aussteigen_lassen(),
   Losfahren =
       weiterfahren (get(hier), get(zyklus), get(naechster_halbzyklus)),
   fahrtwuensche_im_bahnhof (Ausfahrzeitpunkt, Losfahren),
   einsteigen_lassen().

fahrtwuensche_im_bahnhof (Ausfahrzeitpunkt, Losfahren) ->
   Restzeit = Ausfahrzeitpunkt - aktuelle_zeit(),
   % bei Timeout darf kein negatives Argument verwendet werden
   if Restzeit < 0 ->
         Resthaltezeit = 0;
      Restzeit >= 0 ->
         Resthaltezeit = Restzeit
   end,
   receive
      {Pid, fahrtwunsch, Wunsch} ->
         Ok = wunsch_registrieren(Wunsch),
         Pid ! {self(), Ok},
         if Ok == true, Resthaltezeit == 0 ->
```

```
                true;
             Ok == true, Resthaltezeit > 0 ->
                fahrtwuensche_im_bahnhof(Ausfahrzeitpunkt, Ok);
             Ok == false ->
                fahrtwuensche_im_bahnhof(Ausfahrzeitpunkt, Losfahren)
          end
    after
       Resthaltezeit ->
          if Losfahren == true ->
                true;
             Losfahren == false ->
                fahrtwuensche_im_bahnhof(Ausfahrzeitpunkt, Losfahren)
          end
    end.

aussteigen_lassen () ->
    Hier = get(hier),
    Zyklus = get(zyklus),
    Bahnsteigdaten = element(position(Hier, Zyklus), Zyklus),
    Aussteiger = element(2, Bahnsteigdaten),
    rendezvous_mit_aussteigern(Aussteiger),
    Bahnsteigdaten_Neu = setelement(2, Bahnsteigdaten, []),
    Zyklus_Neu =
       setelement(position(Hier, Zyklus), Zyklus, Bahnsteigdaten_Neu),
    put(zyklus, Zyklus_Neu).

rendezvous_mit_aussteigern ([])  ->
    true;
rendezvous_mit_aussteigern ([Kopf|Schwanz])  ->
    Pid = element(Kopf, get(passagiere)),
    Pid ! {self(), aussteigen},
    receive
       ausgestiegen ->
          true
    end,
    rendezvous_mit_aussteigern (Schwanz).

einsteigen_lassen () ->
    Zyklus = get(zyklus),
    Hier = get(hier),
    Bahnsteigdaten = element(position(Hier, Zyklus), Zyklus),
    Einsteiger = element(3, Bahnsteigdaten),
    rendezvous_mit_einsteigern(Einsteiger),
    Bahnsteigdaten_Neu = setelement(3, Bahnsteigdaten, []),
    Zyklus_Neu = setelement(position(Hier, Zyklus), Zyklus, Bahnsteigdaten_Neu),
    put(zyklus, Zyklus_Neu).

rendezvous_mit_einsteigern ([])  ->
    true;
rendezvous_mit_einsteigern ([Kopf|Schwanz])  ->
    Pid = element(Kopf, get(passagiere)),
```

```
   Pid ! {self(), einsteigen},
   receive
      eingestiegen ->
         true
   end,
   rendezvous_mit_einsteigern(Schwanz).

weiterfahren (Hier, Zyklus, Naechster_Halbzyklus) ->
   weiterfahren (Hier, 1, Zyklus, Naechster_Halbzyklus).
weiterfahren (Hier, Position, Zyklus, Naechster_Halbzyklus)
 when Position =< size(Naechster_Halbzyklus) ->
   Halten = element(1, element(Position, Naechster_Halbzyklus)),
   if Halten == true ->
         true;
      Halten == false ->
         weiterfahren (Hier, Position+1, Zyklus, Naechster_Halbzyklus)
   end;
weiterfahren (Hier, Position, Zyklus, Naechster_Halbzyklus) ->
   Halten = element(1, element(Position, Zyklus)),
   Hier_Position = position(Hier, Zyklus),
   if Halten == true, Position /= Hier_Position->
         true;
      Position == size(Zyklus) ->
         false;
      true ->
         weiterfahren (Hier, Position+1, Zyklus, Naechster_Halbzyklus)
   end.

in_den_naechsten_abschnitt() ->
   Hier = get(hier),
   Diese_Kabine = get(diese_kabine),
   Zyklus = get(zyklus),
   Naechster_Halbzyklus = get (naechster_halbzyklus),
   Abschnitte = get(abschnitte),
   Dort = nachfolger(Hier),
   Position_Hier = position(Hier, Zyklus),
   Bahnsteigdaten_Hier = element(Position_Hier, Zyklus),
   Halten_Hier = element(1, Bahnsteigdaten_Hier),
   if Halten_Hier == true ->
         strecke_pruefen(Hier);
      Halten_Hier == false ->
         true
   end,
   % Einfahrerlaubnis fuer kommenden Abschnitt
   Position_Dort = position(Dort, Zyklus),
   Bahnsteigdaten_Dort = element(Position_Dort, Zyklus),
   Halten_Dort = element(1, Bahnsteigdaten_Dort),
   if Halten_Dort == true ->
         bahnhof_einfahren(Dort);
      Halten_Dort == false ->
         Abschnitt_Pid = element(Position_Dort, Abschnitte),
```

```
            Abschnitt_Pid ! {self(), strecke_ein, Diese_Kabine},
            receive
               {Abschnitt_Pid, strecke_ein} ->
                  true
            end
      end,
      % Ausfahrmeldung fuer aktuellen Abschnitt
      if Halten_Hier == true ->
            Bahnsteigdaten_Hier_Neu = setelement(1, Bahnsteigdaten_Hier, false),
            Zyklus_Neu =
               setelement(Position_Hier, Zyklus, Bahnsteigdaten_Hier_Neu),
            put(zyklus, Zyklus_Neu),
            Abschnitt_Pid2 = element(Position_Hier, Abschnitte),
            Abschnitt_Pid2 ! {self(), bahnhof_aus},
            receive
               {Abschnitt_Pid2, bahnhof_aus} ->
                  true
            end;
         Halten_Hier == false ->
            Abschnitt_Pid2 = element(Position_Hier, Abschnitte),
            Abschnitt_Pid2 ! {self(), strecke_aus},
            receive
               {Abschnitt_Pid2, strecke_aus} ->
                  true
            end,
            Bahnsteigdaten_Hier_Neu = Bahnsteigdaten_Hier
      end,
      Bahnsteigdaten = setelement(4, Bahnsteigdaten_Hier_Neu, 0),
      Neuer_Zyklus = setelement(Position_Hier, get(zyklus), Bahnsteigdaten),
      put(zyklus, Neuer_Zyklus),
      if element(1, Hier) /= element(1, Dort) ->
            richtungswechsel ();
         true ->
            true
      end,
      put(hier, Dort).

richtungswechsel () ->
      Hier = get(hier),
      Zyklus = get(zyklus),
      Naechster_Halbzyklus = get(naechster_halbzyklus),
      Zyklus_Neu =
         zyklus_ergaenzen(element(1, Hier), Zyklus, Naechster_Halbzyklus),
      put(zyklus, Zyklus_Neu),
      Halbzyklus_Neu = halbzyklus_initialisieren(Naechster_Halbzyklus),
      put(naechster_halbzyklus, Halbzyklus_Neu).

halbzyklus_initialisieren (Naechster_Halbzyklus) ->
      halbzyklus_initialisieren (Naechster_Halbzyklus,
                                 size(Naechster_Halbzyklus)).
```

```
halbzyklus_initialisieren (Naechster_Halbzyklus, 1) ->
   setelement(1, Naechster_Halbzyklus, {false, [], [], 0});
halbzyklus_initialisieren (Naechster_Halbzyklus, N) ->
   Naechster_Halbzyklus2 =
      halbzyklus_initialisieren(Naechster_Halbzyklus, N - 1),
   setelement(N, Naechster_Halbzyklus2, {false, [], [], 0}).

strecke_pruefen (Hier) ->
   Abschnitte = get(abschnitte),
   Abschnitt_Pid = element(position(Hier, Abschnitte), Abschnitte),
   Abschnitt_Pid ! {self(), strecke_frei},
   receive
      {Abschnitt_Pid, strecke_frei} ->
         true
   end.

bahnhof_einfahren (Dort) ->
   Abschnitte = get(abschnitte),
   Diese_Kabine = get(diese_kabine),
   Kabinen = get(kabinen),
   Wiederholzeit = 90000,
   Abschnitt_Pid = element(position(Dort, Abschnitte), Abschnitte),
   Abschnitt_Pid ! {self(), bahnhof_ein, Diese_Kabine},
   receive
      {Abschnitt_Pid, bahnhof_ein} ->
         true
   after
      Wiederholzeit ->
         Wunsch = {Dort, nachfolger(Dort), 0, 0},
         Abschnitt_Dort_Pid = element(position(Dort, Abschnitte), Abschnitte),
         Abschnitt_Dort_Pid ! {self(), fragen_im_bahnhof},
         receive
            {Abschnitt_Dort_Pid, fragen_im_bahnhof, Kabine} ->
               kabine_fragen(Kabine, Wunsch),
               receive
                  {Abschnitt_Pid, bahnhof_ein} ->
                     true
               end
         end
   end.

aktuelle_zeit() -> %in Millisekunden
   round(1000 * element(2, now()) + element(3, now()) / 1000).


-module(abschnittskontrolle).
-export([abschnittskontrolle/3]).

abschnittskontrolle (Aufstrecke, Imbahnhof, Ausfahrabsicht) ->
   receive
      {Pid, strecke_ein, Kabine} when Ausfahrabsicht == false,
```

```
                                        Aufstrecke == 0 ->
            Pid ! {self(), strecke_ein},
            abschnittskontrolle(Kabine, Imbahnhof, Ausfahrabsicht);

         {Pid, bahnhof_ein, Kabine} when Imbahnhof == 0 ->
            Pid ! {self(), bahnhof_ein},
            abschnittskontrolle(Aufstrecke, Kabine, Ausfahrabsicht);

         {Pid, strecke_frei} when Aufstrecke == 0 ->
            Pid ! {self(), strecke_frei},
            abschnittskontrolle(Aufstrecke, Imbahnhof, true);

         {Pid, strecke_aus} ->
            Pid ! {self(), strecke_aus},
            abschnittskontrolle(0, Imbahnhof, Ausfahrabsicht);

         {Pid, bahnhof_aus} ->
            Pid ! {self(), bahnhof_aus},
            abschnittskontrolle(Aufstrecke, 0, false);

         {Pid, fragen_auf_strecke} ->
            Pid ! {self(), fragen_auf_strecke, Aufstrecke},
            abschnittskontrolle(Aufstrecke, Imbahnhof, Ausfahrabsicht);

         {Pid, fragen_im_bahnhof} ->
            Pid ! {self(), fragen_im_bahnhof, Imbahnhof},
            abschnittskontrolle(Aufstrecke, Imbahnhof, Ausfahrabsicht)
      end.


-module(kartenverkauf).
-export([kartenverkauf/1, kabine_fragen/2, position/2]).
-import(streckenbestimmung, [vorgaenger/1]).
kartenverkauf (Diese_Station) ->
   receive
      {Abschnitte, Kabinen, Max_Station} ->
         put(abschnitte, Abschnitte),
         put(kabinen, Kabinen),
         put(max_station, Max_Station),
         kartenverkauf1(Diese_Station)
   end.

kartenverkauf1 (Diese_Station) ->
   receive
      {Pid, karte_kaufen, Ziel, Zahl, Passagier} ->
            if Diese_Station == Ziel ->
                  Pid ! {self(), false};
               Diese_Station /= Ziel ->
                  Wunsch = fahrtwunsch_zusammenstellen
                              (Diese_Station, Ziel, Zahl, Passagier),
                  Ok =  kabine_suchen(Wunsch),
```

```
                Pid ! {self(), Ok}
            end
   end,
   kartenverkauf1(Diese_Station).

fahrtwunsch_zusammenstellen (Diese_Station, Ziel, Zahl, Passagier) ->
   if Diese_Station < Ziel ->
         {{aufwaerts, Diese_Station}, {aufwaerts, Ziel}, Zahl, Passagier};
      Diese_Station > Ziel ->
         {{abwaerts, Diese_Station}, {abwaerts, Ziel}, Zahl, Passagier}
   end.

kabine_suchen (Wunsch) ->
   Bahnsteig = element(1, Wunsch),
   kabine_suchen (Bahnsteig, Wunsch).

kabine_suchen (Bahnsteig, Wunsch) ->
   Abschnitte = get(abschnitte),
   Kabinen = get(kabinen),
   Abschnitt = element(position(Bahnsteig, Abschnitte), Abschnitte),
   Abschnitt ! {self(), fragen_im_bahnhof},
   receive
      {Abschnitt, fragen_im_bahnhof, Kabine} ->
         Ok = kabine_fragen(Kabine, Wunsch),
         if Ok == true ->
               true;
            Ok == false ->
               Vorgaenger_Bahnsteig = vorgaenger(Bahnsteig),
               Vorgaenger_Abschnitt = element
                  (position(Vorgaenger_Bahnsteig, Abschnitte), Abschnitte),
               Vorgaenger_Abschnitt ! {self(), fragen_auf_strecke},
               receive
                  {Vorgaenger_Abschnitt, fragen_auf_strecke, Vor_Kabine} ->
                     Ok2 = kabine_fragen(Vor_Kabine, Wunsch),
                     if Ok2 == true ->
                           true;
                        Vorgaenger_Bahnsteig == element(1, Wunsch) ->
                           false;
                        true ->
                           kabine_suchen(Vorgaenger_Bahnsteig, Wunsch)
                     end
               end
         end
   end.

kabine_fragen (Kabine, Wunsch) ->
   if Kabine == 0 ->
         false;
      Kabine  > 0 ->
         Pid = element(Kabine, get(kabinen)),
         Pid ! {self(), fahrtwunsch, Wunsch},
```

```
         receive
            {Pid, Ok} ->
               Ok
         end
   end.

position (Bahnsteig, Zyklus) ->
   {Richtung, Station} = Bahnsteig,
   if Richtung == aufwaerts ->
         Station;
      Richtung == abwaerts ->
         size(Zyklus) + 1 - Station
   end.


-module(taxifahren).
-export([taxifahren/1]).

taxifahren (Passagier) ->
   receive
      Automaten ->
         taxifahren1(Automaten, Passagier)
   end.

taxifahren1 (Automaten, Passagier) ->
   %Bestimmen der Werte fuer Von, Nach, Zahl
   Von  = 1,
   Nach = 4,
   Zahl = 2,

   Automat = element(Von, Automaten),
   Automat ! {self(), karte_kaufen, Nach, Zahl, Passagier},
   receive
      {Automat, false} -> %Fahrtwunsch nicht erfuellbar
         true;
      {Automat, true}  -> %Karte wurde gekauft
         einsteigen(Zahl),
         aussteigen(Zahl)
   end,
   taxifahren1(Automaten, Passagier).

einsteigen (1) ->
   receive
      {Pid, einsteigen} ->
         Pid ! eingestiegen
   end;

einsteigen (Zahl) when Zahl > 1 ->
   einsteigen(1),
   einsteigen(Zahl-1).
```

```
aussteigen (1) ->
   receive
      {Pid, aussteigen} ->
         Pid ! ausgestiegen
   end;

aussteigen (Zahl) when Zahl > 1 ->
   aussteigen(1),
   aussteigen(Zahl-1).



-module(initialisieren).
-export([start/4]).
-import(kabinensteuerung, [kabinensteuerung/6]).
-import(kartenverkauf, [kartenverkauf/1]).
-import(taxifahren, [taxifahren/1]).
-import(abschnittskontrolle, [abschnittskontrolle/3]).
-import(streckenbestimmung, [nachfolger/1]).

start (Max_Station, Max_Kabine, Max_Platz, Max_Passagier) ->
   put(max_station, Max_Station),
   {Kabinen_Liste, Abschnitt_Liste} =
      kabinen_verteilen(Max_Kabine, Max_Station, Max_Platz),
   Passagier_Liste = passagiere_starten(Max_Passagier),
   Automaten_Liste = automaten_initialisieren(Max_Station, Abschnitt_Liste,
                                              Kabinen_Liste),
   kabinen_initialisieren(Kabinen_Liste, Abschnitt_Liste, Passagier_Liste),
   passagiere_initialisieren(Passagier_Liste, Automaten_Liste).

kabinen_verteilen (Max_Kabine, Max_Station, Max_Platz) ->
   Anfangskabine = 1,
   Anfangsbahnsteig = {aufwaerts, 1},
   Zyklus = zyklus_erzeugen(2*Max_Station),
   Naechster_Halbzyklus = zyklus_erzeugen(Max_Station),
   kabinen_verteilen(Anfangskabine, [], Max_Kabine, Max_Platz,
           Anfangsbahnsteig, [], Max_Station, Zyklus, Naechster_Halbzyklus).

kabinen_verteilen (Kabine, Kabinen, Max_Kabine, Max_Platz, Bahnsteig,
                   Abschnitte, Max_Station, Zyklus, Naechster_Halbzyklus)
 when Kabine =< Max_Kabine ->
   Kabine_Pid = spawn(kabinensteuerung, kabinensteuerung, [Zyklus,
      Naechster_Halbzyklus, Kabine, Bahnsteig, Max_Platz, Max_Station]),
   Abschnitt_Pid = spawn(abschnittskontrolle, abschnittskontrolle,
                         [0, Kabine, false]),
   Kabinen_Liste = lists:append(Kabinen, [Kabine_Pid]),
   Abschnitt_Liste = lists:append(Abschnitte, [Abschnitt_Pid]),
   if Kabine < Max_Kabine ->
         kabinen_verteilen(Kabine + 1, Kabinen_Liste, Max_Kabine, Max_Platz,
                         nachfolger(Bahnsteig), Abschnitt_Liste, Max_Station,
                         Zyklus, Naechster_Halbzyklus);
```

```
      Kabine == Max_Kabine ->
         kabinen_verteilen(Kabinen_Liste, nachfolger(Bahnsteig),
                           Abschnitt_Liste, Max_Station)
   end.

kabinen_verteilen (Kabinen, Bahnsteig, Abschnitte, Max_Station) ->
   Abschnitt_Pid = spawn(abschnittskontrolle, abschnittskontrolle,
                         [0, 0, false]),
   Abschnitt_Liste = lists:append(Abschnitte, [Abschnitt_Pid]),
   if element(1,Bahnsteig) == abwaerts, element(2,Bahnsteig) == 1 ->
         {list_to_tuple(Kabinen), list_to_tuple(Abschnitt_Liste)};
      true ->
         kabinen_verteilen(Kabinen, nachfolger(Bahnsteig),
                           Abschnitt_Liste, Max_Station)
   end.

automaten_initialisieren (Max_Station, Abschnitte, Kabinen) ->
   automaten_initialisieren([],Max_Station,Abschnitte,Kabinen, Max_Station).

automaten_initialisieren
      (Automaten, Station, Abschnitte, Kabinen, Max_Station) ->
   Automat = spawn(kartenverkauf, kartenverkauf, [Station]),
   Automat ! {Abschnitte, Kabinen, Max_Station},
   Automaten_Liste = lists:append([Automat], Automaten),
   if Station > 1 ->
         automaten_initialisieren (Automaten_Liste, Station-1,
                                   Abschnitte, Kabinen, Max_Station);
      Station == 1 ->
         list_to_tuple(Automaten_Liste)
   end.

passagiere_starten (Max_Passagier) ->
   passagiere_starten ([], Max_Passagier).

passagiere_starten (Passagiere, Passagier) ->
   Passagier_Pid = spawn(taxifahren, taxifahren, [Passagier]),
   Passagier_Liste = lists:append([Passagier_Pid], Passagiere),
   if Passagier > 1 ->
         passagiere_starten(Passagier_Liste, Passagier-1);
      Passagier == 1 ->
         list_to_tuple(Passagier_Liste)
   end.

passagiere_initialisieren (Passagiere, Automaten) ->
   Passagier_Liste = tuple_to_list(Passagiere),
   passagiere_initialisieren1(Passagier_Liste, Automaten).

passagiere_initialisieren1 (Passagiere, Automaten) ->
   if Passagiere == [] ->
         true;
      Passagiere /= [] ->
```

```
            hd(Passagiere) ! Automaten,
            passagiere_initialisieren1 (tl(Passagiere), Automaten)
      end.

kabinen_initialisieren (Kabinen, Abschnitte, Passagiere) ->
      Kabinen_Liste = tuple_to_list (Kabinen),
      kabinen_initialisieren1(Kabinen_Liste, Kabinen, Abschnitte, Passagiere).

kabinen_initialisieren1 (Liste, Kabinen, Abschnitte, Passagiere) ->
      if Liste == [] ->
            true;
         Liste /= [] ->
            hd(Liste) ! {Kabinen, Abschnitte, Passagiere},
            kabinen_initialisieren1(tl(Liste), Kabinen, Abschnitte, Passagiere)
      end.

zyklus_erzeugen (Max_Station) ->
      list_to_tuple(zyklus_erzeugen1(Max_Station)).

zyklus_erzeugen1 (1) ->
      [{false, [], [], 0}];
zyklus_erzeugen1 (Max_Station) ->
      lists:append(zyklus_erzeugen1(1),zyklus_erzeugen1(Max_Station-1)).
```

**Abbildung 4.9** Die Kabinenbahn in Erlang

Damit die Prozesse miteinander kommunizieren können, erhält jeder Prozeß — mit Ausnahme der Prozesse für die Verwaltung der Blockabschnitte — zu Beginn die Listen der PIDs derjenigen Prozesse, mit denen Nachrichten ausgetauscht werden sollen.

Als Datenstrukturen bietet Erlang lediglich Listen und Tupel. Somit mußten die in den Prozessen zur Kabinensteuerung verwalteten Daten `zyklus` und `naechster_halbzyklus` mittels Tupel und Listen implementiert werden. Dabei wurden folgende Vereinbarungen getroffen:

- Der Typ `Fahrgastwechsel` wird als Liste dargestellt. Die einzelnen Elemente der Liste geben jeweils die Passagiergruppe an, die Länge der Liste entspricht der Anzahl der Passagiere.
- Der Typ `Bahnsteigdaten` wird als Tupel mit 4 Elementen dargestellt.
- Der Typ `Halbzyklusdaten` wird als Tupel von `Bahnsteigdaten` dargestellt. Die Länge eines solchen Tupels entspricht der Anzahl der Bahnhöfe.
- Der Typ `Zyklusdaten` wird als Tupel von `Bahnsteigdaten` dargestellt. Die Länge eines solchen Tupels entspricht der Anzahl der Bahnsteige (=2∗Bahnhöfe).

Die Manipulation dieser Daten erweist sich in Erlang als sehr umständlich. Komponenten eines Tupels lassen sich nur über ihre Nummer im Tupel ansprechen (`element(N, Tupel)`). Um ein Tupel zu manipulieren, benutzt man die Funktion `setelement(Index, Tupel, Value)`. Diese Funktion liefert ein Tupel zurück, das eine Kopie des Arguments `Tupel` ist, wobei aber das Element mit der Nummer `Index` durch das Argument `Value` ersetzt wurde.

Da wir geschachtelte Tupel benutzen, wird diese Art des Zugriffs sehr umständlich. Außerdem wurden für die Verwaltung der beiden Variablen `Zyklus` und `Naechster_Halbzyklus` noch 2 Funktionen definiert:

- `halbzyklus(Richtung, Zyklus)` liefert — in Abhängigkeit von Richtung — die erste (wenn `Richtung` gleich `aufwaerts`) oder die zweite (wenn `Richtung` gleich `abwaerts`) Hälfte von `Zyklus` zurück.

- `zyklus_ergaenzen(Richtung, Zyklus, Halber_Zyklus)` liefert einen Zyklus zurück, der zusammengesetzt ist aus `Halber_Zyklus` und der zweiten Hälfte von `Zyklus` (bei Richtung `aufwaerts`) oder aus der ersten Hälfte von `Zyklus` und `Halber_Zyklus` (Richtung `abwaerts`).

Die Kommunikation der Prozesse untereinander erfolgt mittels Senden und Empfangen von Nachrichten. Der erforderliche synchrone Ablauf wird jeweils durch 2 Sendeoperationen (`!`) und 2 `receive` erreicht.

Ein Problem trat bei den Timeouts auf. Die Aufgabenstellung erfordert 2 Arten von Timeout:

1. Timeout auf Seiten des Servers: Wenn ein Server innerhalb einer gewissen Zeitspanne keine Nachricht von einem Client bekommen hat, so soll der Wartevorgang abgebrochen und eine alternative Anweisungsfolge ausgeführt werden. Diese Art von Timeout läßt sich mittels des `after` bei der `receive` Anweisung angeben.

2. Timeout auf Seiten des Client: Wenn ein Client eine Anfrage an einen Server gestellt hat, die dieser nicht sofort bearbeiten kann, so soll der Client nur eine gewisse Zeitspanne warten und — wenn die Anfrage nach Ablauf dieser Zeitspanne noch nicht angenommen wurde — die Anfrage zurückziehen. Das Problem in Erlang ist, daß das Senden von Nachrichten asynchron erfolgt. Nachdem der Client eine Anfrage (Nachricht) an den Server geschickt hat, gibt es keine Möglichkeit, diese Nachricht wieder zurückzuziehen; die Nachricht steht dann in der Mailbox des Servers. Somit läßt sich diese Art von Timeout nicht implementieren.

Das bedeutet, es sind Änderungen dort erforderlich, wo mit Timeout auf Seiten des Client gewartet werden soll. Dies betrifft 3 Funktionen:

1. `strecke_pruefen` In [HH 94] wird hier nach Ablauf des Timeouts die Anfrage so lange wiederholt, bis der Server das Rendezvous eingehen kann. Das bedeutet für die Erlang-Lösung, daß nach dem Senden der Nachricht auf die Antwort gewartet wird — und zwar ohne Timeout.

2. `bahnhof_einfahren` In [HH 94] wird hier nach Ablauf des Timeouts ein Leerfahrtwunsch zusammengestellt und an die Kabine, die den Bahnhof blockiert, geschickt. Danach wird die Anfrage wiederholt. Für die Erlang-Lösung heißt das, daß nach Absenden der Nachricht `bahnhof_ein` zunächst mit Timeout auf die Antwort gewartet wird. Sollte das Timeout ablaufen, wird ein Leerfahrtwunsch zusammengestellt und an die Kabine, die den Bahnhof blockiert, geschickt. Danach kann aber die Anfrage nicht wiederholt werden, da die Nachricht ja bereits abgeschickt wurde. Statt dessen wird nun — ohne Timeout — auf die Antwort (zu `bahnhof_ein`) gewartet.

3. `kabine_fragen` In [HH 94] wird hier mit Timeout gewartet, um Verklemmungen zu vermeiden. Verklemmungen können auftreten, da ein Prozeß der Abschnittssicherung ein Rendezvous mit einer Kabine beantragt und andererseits eine Kabine auch ein Rendezvous mit der Abschnittssicherung. Wir lösen dieses Problem in Erlang, indem das Rendezvous mit einer Kabine nicht im Prozeß der Abschnittssicherung erfolgt, sondern im Prozeß des Fahrkartenverkaufs. Das bedeutet, im Prozeß des Fahrkartenverkaufs wird zunächst ein Rendezvous mit der Abschnittssicherung eingegangen, um die Nummer der Kabine zu erfahren. Danach beantragt der Prozeß des Fahrkartenverkaufs ein Rendezvous mit dieser Kabine. Da nun das Beantragen eines Rendezvous nur noch in einer Richtung erfolgen kann, ist die Gefahr der Verklemmung gebannt.

### 4.4.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 3 vergeben.

#### Programmgröße

Die Programmgröße beträgt 627 nloc.

#### Kommunikationsaufwand

Bei der Betrachtung des Kommunikationsaufwandes beziehen wir uns auf die in Abbildung 4.10 dargestellte Ausführung. Dabei finden 13 Rendezvous statt. Jedes Rendezvous erfordert genau 2 Sendeoperationen (`!`) und 2 `receive`. Somit ergibt sich ein Kommunikationsaufwand von $13 * (2 + 2) = 52$ Anweisungen.

#### Lebendigkeit

Zu jedem Prozeß P in Erlang gehört eine Mailbox, in der alle an P gesendeten Nachrichten in der Reihenfolge ihrer Ankunft gespeichert werden. Wenn P ein `receive` ausführt, so wird die erste in der Mailbox befindliche Nachricht — also die, die als erste angekommen ist — mit den verschiedenen `receive`-Alternativen verglichen und geprüft, ob sie empfangen werden kann. Wenn ja, so wird die Nachricht aus der Mailbox entfernt; wenn nein, so wird die zweite Nachricht überprüft, dann die dritte usw.

Diese Vorgehensweise garantiert, daß keine Nachricht in der Mailbox von einer nach ihr ankommenden Nachricht übergangen werden kann. Das bedeutet, das Programm ist lebendig.

#### Verklemmung

Verklemmung kann nur auftreten, wenn Prozesse in ihrer Ausführung anhalten. Da wir das aber ausschließen, ist das Programm verklemmungsfrei.

#### Aushungern

Da Nachrichten in der Mailbox eines Prozesses nicht übergangen werden können, ist Aushungern nicht möglich.

#### Faire Maschine

Es ist keine faire Maschine erforderlich.

#### Nebenläufigkeitsgrad

Für die Betrachtung des Nebenläufigkeitsgrades erweitern wir die Ausführung aus Abbildung 4.10 um eine globale Zeitskala (Abbildung 4.11). Zur Vereinfachung nehmen wir an, daß jede der 25 Zeitspannen gleich lang ist. Aus der graphischen Darstellung läßt sich Tabelle 4.2 ableiten. Damit ergibt sich ein Nebenläufigkeitsgrad von $\frac{28}{25} = 1.12$.

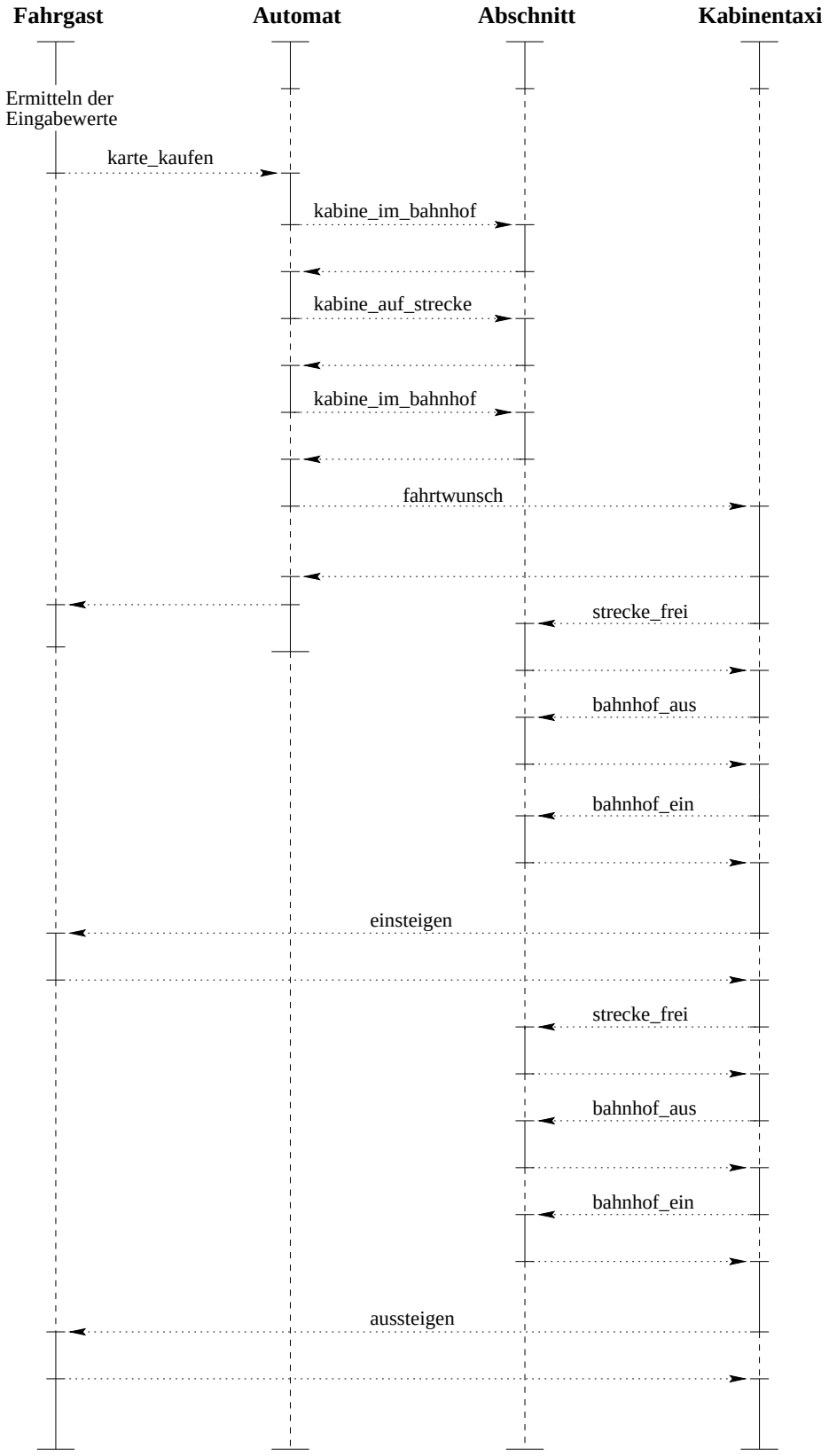


**Abbildung 4.10** Betrachtung des Kommunikationsaufwandes

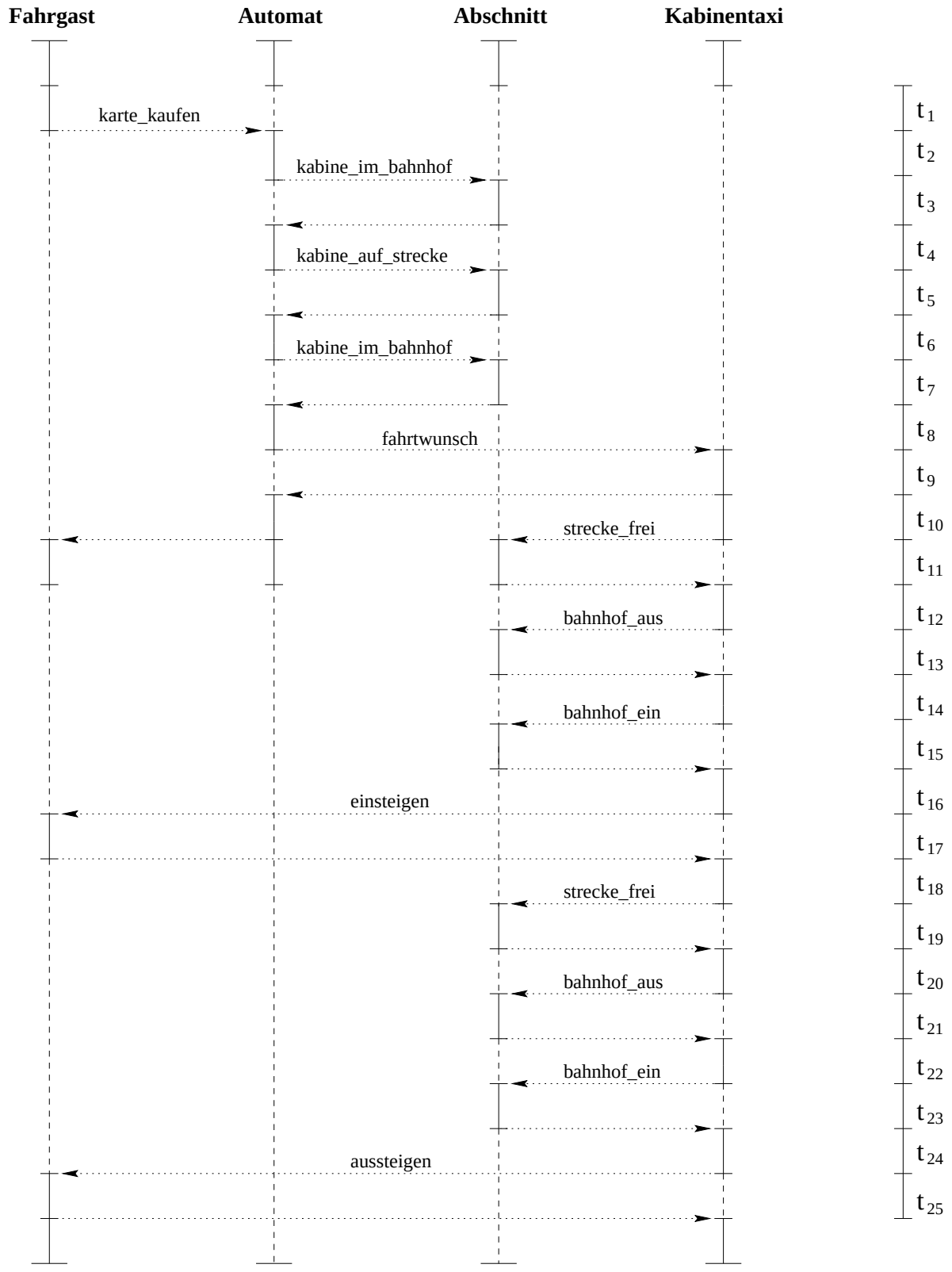


**Abbildung 4.11** Betrachtung des Nebenläufigkeitsgrades

| Zeitspanne | Fahrgast | Automat | Abschnittsverwaltung | Kabinentaxi | aktive Tasks |
|---|---|---|---|---|---|
| $t_1$ | aktiv | blockiert | blockiert | blockiert | 1 |
| $t_2$ | blockiert | aktiv | blockiert | blockiert | 1 |
| $t_3$ | blockiert | blockiert | aktiv | blockiert | 1 |
| $t_4$ | blockiert | aktiv | blockiert | blockiert | 1 |
| $t_5$ | blockiert | blockiert | aktiv | blockiert | 1 |
| $t_6$ | blockiert | aktiv | blockiert | blockiert | 1 |
| $t_7$ | blockiert | blockiert | aktiv | blockiert | 1 |
| $t_8$ | blockiert | aktiv | blockiert | blockiert | 1 |
| $t_9$ | blockiert | blockiert | blockiert | aktiv | 1 |
| $t_{10}$ | blockiert | aktiv | blockiert | aktiv | 2 |
| $t_{11}$ | aktiv | aktiv | aktiv | blockiert | 3 |
| $t_{12}$ | blockiert | blockiert | blockiert | aktiv | 1 |
| $t_{13}$ | blockiert | blockiert | aktiv | blockiert | 1 |
| $t_{14}$ | blockiert | blockiert | blockiert | aktiv | 1 |
| $t_{15}$ | blockiert | blockiert | aktiv | blockiert | 1 |
| $t_{16}$ | blockiert | blockiert | blockiert | aktiv | 1 |
| $t_{17}$ | aktiv | blockiert | blockiert | blockiert | 1 |
| $t_{18}$ | blockiert | blockiert | blockiert | aktiv | 1 |
| $t_{19}$ | blockiert | blockiert | aktiv | blockiert | 1 |
| $t_{20}$ | blockiert | blockiert | blockiert | aktiv | 1 |
| $t_{21}$ | blockiert | blockiert | aktiv | blockiert | 1 |
| $t_{22}$ | blockiert | blockiert | blockiert | aktiv | 1 |
| $t_{23}$ | blockiert | blockiert | aktiv | blockiert | 1 |
| $t_{24}$ | blockiert | blockiert | blockiert | aktiv | 1 |
| $t_{25}$ | aktiv | blockiert | blockiert | blockiert | 1 |
| Summe der aktiven Tasks | | | | | 28 |

**Tabelle 4.2** Untersuchung des Nebenläufigkeitsgrades

## 4.5 Die Kabinenbahn in Java

### 4.5.1 Programmentwurf

Im Gegensatz zu Ada gibt es in Java keine direkte Kommunikation zwischen Akteuren. Deshalb muß hier die Kommunikation zwischen den einzelnen Threads über zwischengeschaltete passive Kommunikationsobjekte erfolgen. Dabei müssen die Kommunikationsobjekte folgende Funktionen erfüllen:

1. Die Daten, die zwischen den Threads ausgetauscht werden sollen, müssen verwaltet und der Zugriff zu ihnen richtig koordiniert werden.
2. Die Kommunikation muß synchron — in Form eines Rendezvous — erfolgen; d.h. ein Rufer muß nach Ausführen seines Aufrufs so lange warten, bis der Gerufene den angeforderten Dienst ausgeführt und eventuelle Ergebnisdaten zurückgeschickt hat.
3. Timeouts müssen realisiert werden.

### 4.5.2 Programm

```
public class kabinenbahn {

   int max_station, max_kabine, max_platz, max_passagier;
   static final int aufwärts = 0, abwärts = 1, richtungen = 2;
   // Objekt zur Verwaltung der Blockabschnitte
   Abschnittskontrolle[][] abschnitt;

   // Rendezvous Objekte:
   // Objekt zur Simulation des Rendezvous einsteigen
   rendezvous_ohne_daten[] einsteigen;
   // Objekt zur Simulation des Rendezvous aussteigen
   rendezvous_ohne_daten[] aussteigen;
   // Objekt zur Simulation des Rendezvous fahrtwunsch
   rendezvous_mit_daten_und_timeout[] fahrtwunsch;
   // Objekt zur Simulation des Rendezvous karte_kaufen
   rendezvous_mit_daten[] karte_kaufen;

   // Definition der Klasse bahnsteige
   class bahnsteige implements Cloneable {
      int richtung;
      int station;

      protected Object clone() {
         bahnsteige kopie = new bahnsteige();
         kopie.richtung = this.richtung;
         kopie.station = this.station;
         return kopie;
      }
      protected boolean equals (bahnsteige b) {
         if (this.richtung == b.richtung && this.station == b.station)
            return true;
         else
            return false;
```

```
   }
}
// Definition der Klasse wuensche
class wuensche implements Cloneable {
   bahnsteige von  = new bahnsteige();
   bahnsteige nach = new bahnsteige();
   int zahl;
   int passagier;

   protected Object clone() {
      wuensche kopie = new wuensche();
      kopie.von = (bahnsteige) this.von.clone();
      kopie.nach = (bahnsteige) this.nach.clone();
      kopie.zahl = this.zahl;
      kopie.passagier = this.passagier;
      return kopie;
   }
}

// Konstruktor
kabinenbahn (int  max_station, int max_kabine, int max_platz,
             int max_passagier) {
   this.max_station   = max_station;
   this.max_kabine    = max_kabine;
   this.max_platz     = max_platz;
   this.max_passagier = max_passagier;

   // Erzeugen und Initialisieren der Rendezvous-Objekte
   initialisieren_rendezvous_objekte();
   // Erzeugen und Starten der Threads, Erzeugen und Initialisieren
   // von abschnitt, sowie Verteilen der Kabinen auf Bahnhöfe
   kabinen_verteilen();
   passagiere_initialisieren();
   automaten_initialisieren();
}

void initialisieren_rendezvous_objekte () {
   einsteigen = new rendezvous_ohne_daten[max_passagier];
   aussteigen = new rendezvous_ohne_daten[max_passagier];
   for (int i=0; i<max_passagier; i++) {
      einsteigen[i] = new rendezvous_ohne_daten();
      aussteigen[i] = new rendezvous_ohne_daten();
   }
   fahrtwunsch = new rendezvous_mit_daten_und_timeout[max_kabine];
   for (int i=0; i<max_kabine; i++)
      fahrtwunsch[i] = new rendezvous_mit_daten_und_timeout();
   karte_kaufen = new rendezvous_mit_daten[max_station];
   for (int i=0; i<max_station; i++)
      karte_kaufen[i] = new rendezvous_mit_daten();
}
```

```
void kabinen_verteilen () {
   bahnsteige bahnsteig = new bahnsteige();
   bahnsteige anfang = new bahnsteige();
   int kabine = -1;
   // Ereugen des Feldes der Abschnittsobjekte
   abschnitt = new Abschnittskontrolle[richtungen][max_station];
   bahnsteig.richtung = anfang.richtung = aufwärts;
   bahnsteig.station  = anfang.station = 0;

   bahnhöfe_besetzen:
   while (true) {  // es gibt mindestens einen Bahnsteig mehr als Kabinen
      if (kabine < max_kabine-1) {
         kabine++;
         new Kabinensteuerung(kabine, bahnsteig).start();
         abschnitt[bahnsteig.richtung][bahnsteig.station] =
               new Abschnittskontrolle(kabine);
      }
      else
         abschnitt[bahnsteig.richtung][bahnsteig.station] =
               new Abschnittskontrolle(-1);
      bahnsteig = nachfolger(bahnsteig);
      if (bahnsteig.equals(anfang))
         break bahnhöfe_besetzen;
   }
}

void automaten_initialisieren() {
   for (int i=0; i<max_station; i++)
      new Kartenverkauf(i).start();
}

void passagiere_initialisieren() {
   for (int i=0; i<max_passagier; i++)
      new Taxifahren(i).start();
}

// Funktionen zur Streckenbestimmung
protected int differenz (int richtung) {
   if (richtung == aufwärts)
      return 1;
   else
      return -1;
}
bahnsteige nachfolger (bahnsteige bahnsteig) {
   bahnsteige temp = (bahnsteige) bahnsteig.clone();
   if (bahnsteig.richtung == aufwärts &&
        bahnsteig.station == max_station-1)
      temp.richtung = abwärts;
   else
      if (bahnsteig.richtung == abwärts && bahnsteig.station == 0)
         temp.richtung = aufwärts;
```

```
      else
         temp.station = bahnsteig.station + differenz(bahnsteig.richtung);
   return temp;
}
bahnsteige vorgänger (bahnsteige bahnsteig) {
   bahnsteige temp = (bahnsteige) bahnsteig.clone();
   if (bahnsteig.richtung == abwärts && bahnsteig.station == max_station-1)
      temp.richtung = aufwärts;
   else
      if (bahnsteig.richtung == aufwärts && bahnsteig.station == 0)
         temp.richtung = abwärts;
      else
         temp.station = bahnsteig.station -
            differenz(bahnsteig.richtung);
   return temp;
}

// Klasse zur Simulation eines Rendezvous ohne Datenübergabe
// und ohne Timeout
class rendezvous_ohne_daten {
   protected boolean anfrage_liegt_vor = false;
   protected boolean anfrage_wird_bearbeitet = false;
   protected boolean anfrage_fertig_bearbeitet = false;

   public synchronized void warte_auf_anfrage() {
      while (!anfrage_liegt_vor) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      anfrage_liegt_vor = false;
   }
   public synchronized void starte_anfrage() {
      while (anfrage_wird_bearbeitet) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      anfrage_liegt_vor = true;
      anfrage_wird_bearbeitet = true;
      notifyAll();
   }
   public synchronized void ende_anfrage() {
      while (!anfrage_fertig_bearbeitet) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
```

```
      anfrage_wird_bearbeitet = false;
      anfrage_fertig_bearbeitet = false;
      notifyAll();
   }
   public synchronized void anfrage_abgearbeitet() {
      anfrage_fertig_bearbeitet = true;
      notifyAll();
   }
}

// Klasse zur Simulation eines Rendezvous mit Datenübergabe
// und ohne Timeout
class rendezvous_mit_daten {
   protected boolean anfrage_liegt_vor = false;
   protected boolean anfrage_wird_bearbeitet = false;
   protected boolean anfrage_fertig_bearbeitet = false;
   protected karte_kaufen_daten datum;

   public synchronized karte_kaufen_daten warte_auf_anfrage() {
      while (!anfrage_liegt_vor) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      anfrage_liegt_vor = false;
      return (karte_kaufen_daten) datum.clone();
   }
   public synchronized void starte_anfrage(karte_kaufen_daten datum) {
      while (anfrage_wird_bearbeitet) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      anfrage_liegt_vor = true;
      anfrage_wird_bearbeitet = true;
      this.datum = (karte_kaufen_daten) datum.clone();
      notifyAll();
   }
   public synchronized karte_kaufen_daten ende_anfrage() {
      while (!anfrage_fertig_bearbeitet) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      anfrage_wird_bearbeitet = false;
      anfrage_fertig_bearbeitet = false;
      notifyAll();
      return (karte_kaufen_daten) datum.clone();
```

```
   }
   public synchronized void anfrage_abgearbeitet
                         (karte_kaufen_daten datum) {
      anfrage_fertig_bearbeitet = true;
      this.datum = (karte_kaufen_daten) datum.clone();
      notifyAll();
   }
}

// Definition der Ausnahmeklasse für Timeout
class Timeout extends Exception {}

// Klasse zur Simulation eines Rendezvous mit Datenübergabe
// und mit Timeout

class rendezvous_mit_daten_und_timeout {
   protected boolean anfrage_liegt_vor = false;
   protected boolean anfrage_wird_bearbeitet = false;
   protected boolean anfrage_fertig_bearbeitet = false;
   protected fahrtwunsch_daten datum;

   public synchronized fahrtwunsch_daten warte_auf_anfrage(long zeitspanne)
           throws Timeout {
      final long erster_aufruf = System.currentTimeMillis();
      while (!anfrage_liegt_vor) {
         try {
            if (System.currentTimeMillis()-erster_aufruf >= zeitspanne)
               throw new Timeout();
            wait(zeitspanne-(System.currentTimeMillis()-erster_aufruf));
         }
         catch (InterruptedException e) {}
      }
      anfrage_liegt_vor = false;
      return (fahrtwunsch_daten) datum.clone();
   }
   public synchronized void starte_anfrage
         (fahrtwunsch_daten datum, long zeitspanne) throws Timeout {
      final long erster_aufruf = System.currentTimeMillis();
      while (anfrage_wird_bearbeitet) {
         try {
            if (System.currentTimeMillis()-erster_aufruf >= zeitspanne)
               throw new Timeout();
            wait(zeitspanne-(System.currentTimeMillis()-erster_aufruf));
         }
         catch (InterruptedException e) {}
      }
      anfrage_liegt_vor = true;
      anfrage_wird_bearbeitet = true;
      this.datum = (fahrtwunsch_daten) datum.clone();
      notifyAll();
   }
```

```
   public synchronized fahrtwunsch_daten ende_anfrage() {
      while (!anfrage_fertig_bearbeitet) {
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      anfrage_wird_bearbeitet = false;
      anfrage_fertig_bearbeitet = false;
      notifyAll();
      return (fahrtwunsch_daten) datum.clone();
   }
   public synchronized void anfrage_abgearbeitet
                                    (fahrtwunsch_daten datum) {
      anfrage_fertig_bearbeitet = true;
      this.datum = (fahrtwunsch_daten) datum.clone();
      notifyAll();
   }
}

// Definition der Klassen für die Pufferdaten
// der Rendezvous-Klassen
class fahrtwunsch_daten {
   wuensche wunsch;
   boolean ok;

   fahrtwunsch_daten (wuensche wunsch, boolean ok) {
      this.wunsch = (wuensche) wunsch.clone();
      this.ok = ok;
   }
   protected Object clone() {
      fahrtwunsch_daten kopie =
            new fahrtwunsch_daten(this.wunsch, this.ok);
      return kopie;
   }
}
//Daten für Rendezvous karte_kaufen
class karte_kaufen_daten {
   int ziel;
   int zahl;
   int passagier;
   boolean ok;

   karte_kaufen_daten (int ziel, int zahl, int passagier, boolean ok) {
      this.ziel = ziel;
      this.zahl = zahl;
      this.passagier = passagier;
      this.ok = ok;
   }
   protected Object clone() {
      karte_kaufen_daten kopie = new karte_kaufen_daten
```

```
            (this.ziel, this.zahl, this.passagier, this.ok);
         return kopie;
      }
   }

   // Definition der Threadklasse Kabinensteuerung

   class Kabinensteuerung extends Thread {
      Kabinensteuerung (int kabine, bahnsteige bahnsteig) {
         diese_kabine = kabine;
         hier = (bahnsteige) bahnsteig.clone();
         // Erzeugen der Objekte der Array-Elemente von
         // nächster_halbzyklus und zyklus
         for (int i=0; i<max_station; i++)
            nächster_halbzyklus[i] = new bahnsteigdaten();
         for (int i=0; i<richtungen; i++)
            for (int j=0; j<max_station; j++)
               zyklus[i][j] = new bahnsteigdaten();
         // am Anfang steht die Kabine in einem Bahnhof
         zyklus[hier.richtung][hier.station].halten = true;
      }

      class fahrgastwechsel implements Cloneable {
         int zahl = 0;
         int[] passagier = new int[max_platz];

         protected Object clone() {
            fahrgastwechsel kopie = new fahrgastwechsel();
            kopie.zahl = this.zahl;
            kopie.passagier = (int[]) this.passagier.clone();
            return kopie;
         }
      }

      class bahnsteigdaten implements Cloneable {
         boolean halten = false;
         fahrgastwechsel aussteiger = new fahrgastwechsel();
         fahrgastwechsel einsteiger = new fahrgastwechsel();
         int abfahrer = 0;

         protected Object clone() {
            bahnsteigdaten kopie = new bahnsteigdaten();
            kopie.halten = this.halten;
            kopie.aussteiger = (fahrgastwechsel) this.aussteiger.clone();
            kopie.einsteiger = (fahrgastwechsel) this.einsteiger.clone();
            kopie.abfahrer = this.abfahrer;
            return kopie;
         }
      }

      bahnsteigdaten[] nächster_halbzyklus =
```

```
            new bahnsteigdaten[max_station];
bahnsteigdaten[][] zyklus =
            new bahnsteigdaten[richtungen][max_station];
int diese_kabine;
bahnsteige hier;

boolean wunsch_registrieren (wuensche wunsch) {
   boolean ok = false;
   if (wunsch.zahl == 0) {
      zyklus[wunsch.nach.richtung][wunsch.nach.station].halten = true;
      ok = true;
   }
   else {
      if (anderer_zyklus(wunsch.von, hier)) {
         if (beförderbar(wunsch, nächster_halbzyklus)) {
            befördern(wunsch, nächster_halbzyklus);
            ok = true;
         }
      }
      else {
         if (beförderbar(wunsch, zyklus[wunsch.von.richtung])) {
            befördern(wunsch, zyklus[wunsch.von.richtung]);
            ok = true;
         }
      }
   }
   return ok;
}
boolean anderer_zyklus (bahnsteige bahnsteig, bahnsteige hier) {
   if (bahnsteig.richtung != hier.richtung)
      return false;
   else {
      if (bahnsteig.station == hier.station)
         return !(zyklus[hier.richtung][hier.station].halten);
      else {
         if (bahnsteig.richtung == aufwärts)
            return bahnsteig.station < hier.station;
         else
            return bahnsteig.station > hier.station;
      }
   }
}
boolean beförderbar (wuensche wunsch, bahnsteigdaten[] daten) {
   bahnsteige bahnsteig = (bahnsteige) wunsch.von.clone();
   while (!bahnsteig.equals(wunsch.nach)) {
      if (daten[bahnsteig.station].abfahrer + wunsch.zahl > max_platz)
         return false;
      bahnsteig = nachfolger(bahnsteig);
   }
   return true;
}
```

```
void befördern (wuensche wunsch, bahnsteigdaten[] daten) {
   bahnsteige bahnsteig = (bahnsteige) wunsch.von.clone();
   int platz;
   daten[bahnsteig.station].halten = true;
   fahrgastwechsel_registrieren(wunsch,
                  daten[bahnsteig.station].einsteiger);
   while (!bahnsteig.equals(wunsch.nach)) {
      daten[bahnsteig.station].abfahrer += wunsch.zahl;
      bahnsteig = nachfolger(bahnsteig);
   }
   daten[bahnsteig.station].halten = true;
   fahrgastwechsel_registrieren(wunsch,
                  daten[bahnsteig.station].aussteiger);
}
void fahrgastwechsel_registrieren (wuensche wunsch,
      fahrgastwechsel wechsel) {
   for (int i=1; i<=wunsch.zahl; i++)
      wechsel.passagier[wechsel.zahl-1+i] = wunsch.passagier;
   wechsel.zahl += wunsch.zahl;
}
void aussteigen_lassen() {
   int anzahl = zyklus[hier.richtung][hier.station].aussteiger.zahl;
   for (int i=0; i<anzahl; i++) {
   //Rendezvous mit Fahrgast
   aussteigen[zyklus[hier.richtung][hier.station].
                  aussteiger.passagier[i]].starte_anfrage();
   aussteigen[zyklus[hier.richtung][hier.station].
                  aussteiger.passagier[i]].ende_anfrage();
   }
   zyklus[hier.richtung][hier.station].aussteiger.zahl = 0;
}
void einsteigen_lassen() {
   int anzahl = zyklus[hier.richtung][hier.station].einsteiger.zahl;
   for (int i=0; i<anzahl; i++) {
      // Rendezvous mit Fahrgast
      einsteigen[zyklus[hier.richtung][hier.station].
              einsteiger.passagier[i]].starte_anfrage();
      einsteigen[zyklus[hier.richtung][hier.station].
              einsteiger.passagier[i]].ende_anfrage();
   }
   zyklus[hier.richtung][hier.station].einsteiger.zahl = 0;
}
boolean weiterfahren (bahnsteige hier, bahnsteigdaten[][] zyklus,
                     bahnsteigdaten[] nächster_halbzyklus) {
   for (int i=0; i<max_station; i++) {
      for (int j=0; j<richtungen; j++) {
         if ((i != hier.station) || (j != hier.richtung)) {
            if (zyklus[j][i].halten)
               return true;
         }
      }
```

```
      if (nächster_halbzyklus[i].halten)
         return true;
   }
   return false;
}
void in_den_nächsten_abschnitt () {
   bahnsteige dort = nachfolger(hier);

   if (zyklus[hier.richtung][hier.station].halten)
      strecke_prüfen(); // Kollision vermeiden
   // Einfahrerlaubnis für den kommenden Abschnitt
   if (zyklus[dort.richtung][dort.station].halten)
      bahnhof_einfahren(dort);
   else
      abschnitt[dort.richtung][dort.station].strecke_ein(diese_kabine);
   // Ausfahrerlaubnis für den aktuellen Abschnitt
   if (zyklus[hier.richtung][hier.station].halten) {
      zyklus[hier.richtung][hier.station].halten = false;
      abschnitt[hier.richtung][hier.station].bahnhof_aus();
   }
   else
      abschnitt[hier.richtung][hier.station].strecke_aus();
   zyklus[hier.richtung][hier.station].abfahrer = 0;
   if (hier.richtung != dort.richtung)
      richtungswechsel();
   hier = (bahnsteige) dort.clone();
}
void richtungswechsel () {
   for (int i=0; i<max_station; i++)
      zyklus[hier.richtung][i] =
               (bahnsteigdaten) nächster_halbzyklus[i].clone();
   for (int i=0; i<max_station; i++) {
      nächster_halbzyklus[i].halten = false;
      nächster_halbzyklus[i].abfahrer = 0;
      nächster_halbzyklus[i].einsteiger.zahl = 0;
      nächster_halbzyklus[i].aussteiger.zahl = 0;
   }
}
void strecke_prüfen () {
   final int anfragezeit = 30000; //Millisekunden
   strecke_besetzt:
   while (true) {
      try {
         abschnitt[hier.richtung][hier.station].
                        strecke_frei(anfragezeit);
      }
      catch (Timeout t) {
         continue strecke_besetzt;
      }
      break strecke_besetzt;
   }
```

```
}
void bahnhof_einfahren(bahnsteige dort) {
   final int wiederholzeit = 90000; //Millisekunden
   wuensche wunsch = new wuensche();
   boolean ok;
   einfahrerlaubnis_bekommen:
   while (true) {
      try {
         abschnitt[dort.richtung][dort.station].
                  bahnhof_ein(diese_kabine, wiederholzeit);
      }
      catch (Timeout t) {
         leerfahrtwunsch_zusammenstellen(wunsch, dort);
         ok = kabine_fragen (abschnitt[dort.richtung][dort.station].
                              kabine_im_bahnhof(), wunsch);
         continue einfahrerlaubnis_bekommen;
      }
      break einfahrerlaubnis_bekommen;
   }
}
void leerfahrtwunsch_zusammenstellen (wuensche wunsch, bahnsteige dort) {
   wunsch.von = (bahnsteige) dort.clone();
   wunsch.nach = nachfolger(dort);
   wunsch.zahl = 0;
   wunsch.passagier = 0;
}
void vorgehen_auf_strecke() {
   final int fahrzeit = 30000; //Millisekunden
   long restfahrzeit, ausfahrzeitpunkt;
   fahrtwunsch_daten daten;
   ausfahrzeitpunkt = System.currentTimeMillis() + fahrzeit;
   fahrtwünsche_auf_strecke:
   while (true) {
      restfahrzeit = ausfahrzeitpunkt - System.currentTimeMillis();
      if (restfahrzeit < 0)
         break fahrtwünsche_auf_strecke;
      // Beginn Rendezvous
      try {
         daten =
            fahrtwunsch[diese_kabine].warte_auf_anfrage(restfahrzeit);
      }
      catch (Timeout t) {
         break fahrtwünsche_auf_strecke;
      }
      daten.ok = wunsch_registrieren(daten.wunsch);
      fahrtwunsch[diese_kabine].anfrage_abgearbeitet(daten);
      // Ende Rendezvous
   } //fahrtwünsche_auf_strecke
}
public void vorgehen_im_bahnhof () {
   final int haltezeit = 90000; //Millisekunden
```

```
      long resthaltezeit, ausfahrzeitpunkt;
      boolean losfahren;
      fahrtwunsch_daten daten;
      ausfahrzeitpunkt = System.currentTimeMillis() + haltezeit;
      aussteigen_lassen();
      losfahren = weiterfahren (hier, zyklus, nächster_halbzyklus);
      fahrtwünsche_im_bahnhof:
      while (true) {
         resthaltezeit = ausfahrzeitpunkt - System.currentTimeMillis();
         if (losfahren && (resthaltezeit < 0))
            break fahrtwünsche_im_bahnhof;
         // Beginn Rendezvous
         try {
            daten =
               fahrtwunsch[diese_kabine].warte_auf_anfrage(resthaltezeit);
         }
         catch (Timeout t) {
            if (losfahren)
               break fahrtwünsche_im_bahnhof;
            else
               continue fahrtwünsche_im_bahnhof;
         }
         daten.ok = wunsch_registrieren(daten.wunsch);
         fahrtwunsch[diese_kabine].anfrage_abgearbeitet(daten);
         // Ende Rendezvous
         if (daten.ok)
            losfahren = true;
      }
      einsteigen_lassen();
   }

   public void run() {
      while (true) {
         if (zyklus[hier.richtung][hier.station].halten)
            vorgehen_im_bahnhof();
         else
            vorgehen_auf_strecke();
         in_den_nächsten_abschnitt();
      }
   }
}

// Definition der Klasse Abschnittskontrolle zur Verwaltung
// der Streckenabschnitte

class Abschnittskontrolle {
   int aufstrecke, imbahnhof;
   boolean ausfahrabsicht = false;

   Abschnittskontrolle (int kabine) {
      aufstrecke = -1;
```

```
   imbahnhof = kabine;
}
public synchronized void strecke_ein (int kabine) {
   while (ausfahrabsicht || aufstrecke > -1) {
      try {
         wait();
      }
      catch (InterruptedException e) {}
   }
   aufstrecke = kabine;
}
public synchronized void bahnhof_ein (int kabine, long zeitspanne)
      throws Timeout {
   final long erster_aufruf = System.currentTimeMillis();
   while (imbahnhof > -1) {
      try {
         if (System.currentTimeMillis()-erster_aufruf >= zeitspanne)
            throw new Timeout();
         wait (zeitspanne-(System.currentTimeMillis()-erster_aufruf));
      }
      catch (InterruptedException e) {}
   }
   imbahnhof = kabine;
}
public synchronized void strecke_frei (long zeitspanne) throws Timeout {
   final long erster_aufruf = System.currentTimeMillis();
   while (aufstrecke > -1) {
      try {
         if (System.currentTimeMillis()-erster_aufruf >= zeitspanne)
            throw new Timeout();
         wait (zeitspanne-(System.currentTimeMillis()-erster_aufruf));
      }
      catch (InterruptedException e) {}
   }
   ausfahrabsicht = true;
}
public synchronized void strecke_aus () {
   aufstrecke = -1;
   notifyAll();
}
public synchronized void bahnhof_aus () {
   imbahnhof = -1;
   ausfahrabsicht = false;
   notifyAll();
}
public synchronized int kabine_auf_strecke () {
   return aufstrecke;
}
public synchronized int kabine_im_bahnhof () {
   return imbahnhof;
}
```

```
}

// Die Funktion kabine_fragen führt ein Rendezvous mit einer Kabine aus
// Da diese Funktion sowohl aus der Klasse Kartenverkauf als auch aus
// der Klasse Kabinensteuerung aufgerufen wird, muß sie global zu
// diesen sein.

boolean kabine_fragen (int kabine, wuensche wunsch) {
   boolean ok = false;
   fahrtwunsch_daten daten = new fahrtwunsch_daten (wunsch, ok);
   final int anfragezeit = 5000; //Millisekunden

   if (kabine > -1) {
      // Beginn Rendezvous
      try {
         fahrtwunsch[kabine].starte_anfrage (daten, anfragezeit);
      }
      catch (Timeout t) {
         return false;
      }
      // Rendezvous läuft
      daten = fahrtwunsch[kabine].ende_anfrage ();
      // Ende Rendezvous
   }
   return daten.ok;
}

// Definition der Klasse Kartenverkauf, die die Fahrkartenautomaten
// simuliert

class Kartenverkauf extends Thread {

   Kartenverkauf (int station) {
      diese_station = station;
      wunsch = new wuensche();
   }
   int diese_station;
   wuensche wunsch;

   void fahrtwunsch_zusammenstellen (int ziel, int zahl, int passagier) {
      int richtung;
      if (diese_station < ziel)
         richtung = aufwärts;
      else
         richtung = abwärts;
      wunsch.von.richtung = richtung;
      wunsch.von.station = diese_station;
      wunsch.nach.richtung = richtung;
      wunsch.nach.station = ziel;
      wunsch.zahl = zahl;
      wunsch.passagier = passagier;
```

```
   }
   boolean kabine_suchen () {
      bahnsteige bahnsteig = (bahnsteige) wunsch.von.clone();
      boolean erfüllt = false;

      kabine_suchen:
      while (true) {
         erfüllt = kabine_fragen (abschnitt[bahnsteig.richtung]
                     [bahnsteig.station].kabine_im_bahnhof(), wunsch);
         if (!erfüllt) {
            bahnsteig = vorgänger(bahnsteig);
            erfüllt = kabine_fragen (abschnitt[bahnsteig.richtung]
                        [bahnsteig.station].kabine_auf_strecke(), wunsch);
         }
         if (erfüllt || bahnsteig.equals (wunsch.von))
            break kabine_suchen;
      } // kabine_suchen
      return erfüllt;
   }

   public void run () {
      while (true) {
         karte_kaufen_daten daten;
         // Beginn Rendezvous
         daten = karte_kaufen[diese_station].warte_auf_anfrage();
         if (diese_station == daten.ziel)
            daten.ok = false;
         else {
            fahrtwunsch_zusammenstellen
                    (daten.ziel, daten.zahl, daten.passagier);
            daten.ok = kabine_suchen();
         }
         karte_kaufen[diese_station].anfrage_abgearbeitet (daten);
         // Ende Rendezvous
      }
   }
}

// Definition der Klasse Taxifahren, die die Passagiere simuliert

class Taxifahren extends Thread {
   int von, nach, zahl, dieser_passagier;
   boolean ok;

   Taxifahren (int passagier) {
      dieser_passagier = passagier;
   }

   public void run () {
      karte_kaufen_daten daten;
      while (true) {
```

```
                //Bestimmen der Werte für von, nach, zahl
                daten = new karte_kaufen_daten (nach, zahl, dieser_passagier, ok);
                // Beginn Rendezvous
                karte_kaufen[von].starte_anfrage (daten);
                // Rendezvous
                daten = karte_kaufen[von].ende_anfrage ();
                // Ende Rendezvous
                if (daten.ok) {
                   for (int i=0; i<zahl; i++) {
                      // Beginn Rendezvous Einsteigen
                      einsteigen[dieser_passagier].warte_auf_anfrage();
                      einsteigen[dieser_passagier].anfrage_abgearbeitet();
                      // Ende Rendezvous Einsteigen
                   }
                   for (int i=0; i<zahl; i++) {
                      // Beginn Rendezvous Aussteigen
                      aussteigen[dieser_passagier].warte_auf_anfrage();
                      aussteigen[dieser_passagier].anfrage_abgearbeitet();
                   }
                }
                else
                   // Fahrtwunsch nicht erfüllbar
                   ;
             }
          }
       }
    }
```

**Abbildung 4.12** Die Kabinenbahn in Java

Bei der Beschreibung des Programmtextes beschränken wir uns auf die Elemente, die die Kommunikation zwischen den Threads betreffen.

Die Aufgabenstellung verlangt, daß die Kommunikation zwischen den Threads synchron — also in Form eines Rendezvous — abläuft.

Da in Java keine direkte Kommunikation zwischen Akteuren möglich ist, muß für jedes Rendezvous der Ada-Lösung ein passives Kommunikationsobjekt — wir bezeichnen es im Folgenden als Rendezvousobjekt — erzeugt werden. Es gibt drei Arten von Rendezvousobjekten:

1. Rendezvousobjekte ohne Datenübergabe. Die Klasse `rendezvous_ohne_daten` stellt diese Objekte zur Verfügung.

2. Rendezvousobjekte mit Datenübergabe und ohne Timeout. Die Klasse `rendezvous_mit_daten` stellt diese Objekte zur Verfügung.

3. Rendezvousobjekte mit Datenübergabe und mit Timeout. Diese Objekte werden von der Klasse `rendezvous_mit_daten_und_timeout` zur Verfügung gestellt.

Diese Klassen arbeiten alle nach dem gleichen Prinzip. Ein solches Rendezvousobjekt stellt immer 4 Methoden zur Verfügung, 2 für den Client und 2 für den Server. Betrachten wir dazu Abbildung 4.13. Der Server `Automat` zeigt mittels eines Aufrufs von `rend_objekt.warte_auf_anfrage` seine Bereitschaft zum Rendezvous. Sollte noch keine Anfrage eines Clients vorliegen, wird der Server blockiert. Von Seiten des Clients `Fahrgast` erfolgt die Anfrage zum Rendezvous mittels eines Aufrufs

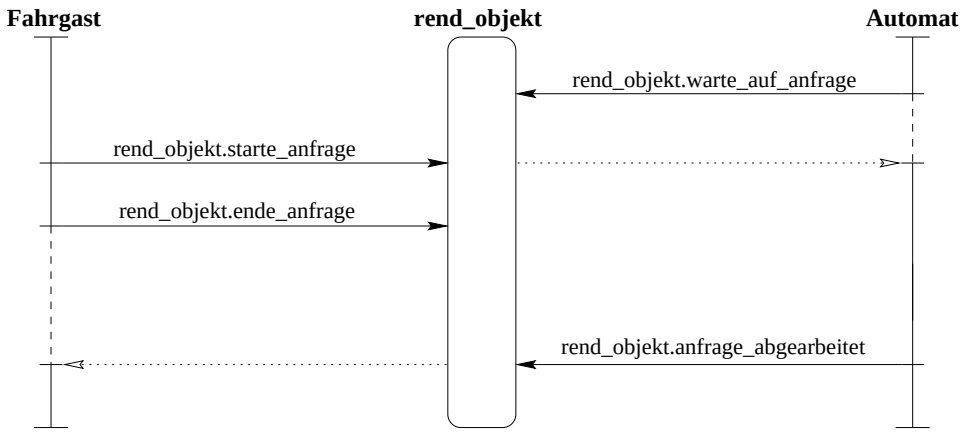


**Abbildung 4.13** Synchronisation über ein zwischengeschaltetes Objekt

von `rend_objekt.starte_anfrage`. Damit wird der Server `Automat` aufgeweckt und beginnt mit der Ausführung des angeforderten Dienstes. Ruft der Client nun `rend_objekt.ende_anfrage` auf, wird er so lange blockiert, bis das Rendezvous beendet ist. Das Ende des Rendezvous zeigt der Server durch Aufruf von `rend_objekt.anfrage_abgearbeitet` an. Das bewirkt, daß der Client aufgeweckt wird.

Die richtige Synchronisation der kommunizierenden Threads wird innerhalb des Rendezvousobjektes durch Anwendung der Methoden `wait` und `notifyAll` realisiert. Damit wird erreicht, daß die Methoden eines Rendezvousobjektes `rend_objekt` immer in der Reihenfolge

1. `rend_objekt.starte_anfrage`
2. `rend_objekt.warte_auf_anfrage`
3. `rend_objekt.anfrage_abgearbeitet`
4. `rend_objekt.ende_anfrage`

ausgeführt werden; auch wenn die Aufrufreihenfolge anders ist.

Diese Reihenfolge ist notwendig, um beim Rendezvous Daten in beide Richtungen übergeben zu können; denn im allgemeinen übergibt der Client dem Server Daten, der Server führt Operationen auf diesen Daten aus und der Client bekommt die Ergebnisse dieser Operationen.

In unserem Fall haben `rendezvous_mit_daten` und `rendezvous_mit_daten_und_timeout` eine Variable `datum` zur Speicherung der übergebenen Daten. Damit ergibt sich (gemäß obiger Reihenfolge) folgender Datenfluß zwischen Client und Server:

1. Der Client schreibt seine Daten in die Variable `datum`.
2. Der Server liest die Variable `datum`. Damit hat der Server die Eingabedaten des Client bekommen und kann die angeforderte Operation ausführen.
3. Der Server schreibt die Ergebnisse der ausgeführten Operation in die Variable `datum`.
4. Der Client liest die Variable `datum`. Damit hat der Client die Ausgabedaten des Server bekommen und das Rendezvous ist beendet.

Die Klasse `rendezvous_mit_daten_und_timeout` realisiert außerdem ein Timeout für den Client und für den Server — in den Methoden `starte_anfrage` und `warte_auf_anfrage`. Timeouts können in Java als Argument `arg` der Methode `wait(arg)` angegeben werden. Das bedeutet, ein Thread,

der `wait(arg)` ausführt, muß nicht auf das `notify` bzw. `notifyAll` eines anderen Threads warten, sondern wird nach Ablauf der Zeitspanne `arg` aufgeweckt.

Dabei gibt es aber zwei Probleme:

1. Man kann einem wieder erweckten Thread nicht ansehen, ob er durch den Ablauf der Zeitspanne `arg` oder durch das `notify` bzw. `notifyAll` eines anderen Thread aufgeweckt wurde.

2. Da die Wartebedingung in einer Schleife abgefragt wird, könnte es passieren, daß ein nach Ablauf seines Timeouts aufgeweckter Thread sofort wieder blockiert wird, da die Wartebedingung nicht geändert wurde.

Wir lösen diese Probleme durch folgende Vorgehensweise:

- Zu Beginn von `starte_anfrage` und `warte_auf_anfrage` wird die aktuelle Zeit abgefragt und in der Konstanten `erster_aufruf` gespeichert.

- Innerhalb der Schleife wird unmittelbar vor dem Aufruf von `wait(arg)` überprüft, ob die Wartezeit abgelaufen ist. Falls ja, wird die Ausnahme `Timeout` ausgelöst. Falls nein, wird `wait(arg)` aufgerufen, wobei `arg` die verbleibende Restzeit ist.

Eine nach Ablauf eines Timeout ausgelöste Ausnahme `Timeout` wird innerhalb von `starte_anfrage` und `warte_auf_anfrage` nicht behandelt. Das bedeutet, daß diese Ausnahme an den Aufrufer weitergeleitet wird und dieser damit die Möglichkeit hat, auf den Ablauf des Timeout zu reagieren.

Die Verwaltung der Streckenabschnitte erfolgt mittels synchronisierter Methoden in der Klasse `Abschnittskontrolle`. Da die Ausführung einiger dieser Methoden an eine Bedingung geknüpft ist, ist auch hier ein explizites Blockieren und Aufwecken von Threads (mit Hilfe von `wait` und `notifyAll`) erforderlich.

### 4.5.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 3 vergeben.

#### Programmgröße

Die Programmgröße beträgt 732 nloc.

#### Kommunikationsaufwand

Bei der Betrachtung des Kommunikationsaufwandes beziehen wir uns auf die in Abbildung 4.6 dargestellte Ausführung. Dabei wird 4 mal über ein Rendezvousobjekt kommuniziert; 9 mal wird eine Methode der Abschnittsverwaltung ausgeführt.

Für jedes Rendezvous ist ein Kommunikationsaufwand von 6 Anweisungen erforderlich. Der Kommunikationsaufwand für die Methoden der Abschnittsverwaltung beträgt insgesamt 6 Anweisungen. Damit ergibt sich für den dargestellten Ablauf ein Kommunikationsaufwand von $4*6+6 = 30$ Anweisungen.

### Lebendigkeit

Da die Kommunikation zwischen den Threads immer mittels synchronisierter Methoden erfolgt, ist die Lösung auf beiden Ebenen nicht lebendig.

Wenn beispielsweise eine Gruppe P von Passagieren am Automat A des aktuellen Bahnhofs eine Karte kaufen möchte, so muß P für den Fall, daß gerade eine andere Gruppe eine Karte kauft, warten. Das bedeutet, der zugehörige Thread $T_P$ wird blockiert und kommt in die Wartemenge des dem Automaten A zugehörigen Rendezvousobjektes $RO_A$. Da es keine Bevorzugung beim Kampf um das Lock eines Objektes gibt, ist es möglich, daß $T_P$ — nachdem er durch das `notifyAll` eines anderen Passagiers aufgeweckt wurde — bei diesem Wettbewerb unendlich oft übergangen wird. Damit ist das Programm bezüglich Ebene 1 nicht lebendig.

Für den Fall, daß $T_P$ das Lock von $RO_A$ bekommen hat, ist nicht garantiert, daß $T_P$ auch tatsächlich ausgeführt werden kann, denn das Überprüfen der Wartebedingungen erfolgt innerhalb einer synchronisierten Methode, also nach Erlangung des Locks. Sollte $T_P$ also feststellen, daß er nicht ausgeführt werden kann (weil gerade ein anderer Passagier eine Karte kauft), so muß $T_P$ das Lock wieder abgeben, wird blockiert und kommt in die Wartemenge von $RO_A$. Damit ist das Programm auch bezüglich Ebene 2 nicht lebendig.

### Verklemmung

Verklemmung tritt nur auf, wenn Threads in ihrer Ausführung anhalten. Da wir das aber ausschließen, tritt Verklemmung nicht auf.

### Aushungern

Aushungern auf Ebene 1 tritt auf, wenn ein Thread beim Kampf um das Lock eines Rendezvousobjektes oder des Objektes zur Abschnittssicherung unendlich oft übergangen wird. Da ein Thread nach Erlangen des Locks wieder blockiert werden kann (weil die zugehörige Ausführungsbedingung nicht erfüllt ist), kann auch auf Ebene 2 Aushungern auftreten.

### Faire Maschine

Für beide Ebenen ist eine faire Maschine erforderlich.

### Nebenläufigkeitsgrad

Da durch die Rendezvousobjekte die Kommunikation zwischen den Threads synchron abläuft und außerdem die Verwaltung der Blockabschnitte innerhalb von passiven Objekten erfolgt, ergibt sich der gleiche Ablauf wie in Abbildung 4.6. Somit beträgt der Nebenläufigkeitsgrad 1.12.

## 4.6 Zusammenfassung und Vergleich

Die Kabinenbahn ließ sich in allen Sprachen ohne allzu große Schwierigkeiten implementieren. Die Lösungen weisen jedoch teilweise erhebliche Unterschiede auf.

Am besten ließ sich das Problem in Ada lösen. Dabei erwies es sich als großer Vorteil, daß Ada sowohl die direkte Kommunikation, als auch die indirekte Kommunikation unterstützt. Für die direkte Kommunikation zwischen den Tasks wurde das Ada-Rendezvous eingesetzt; für das Verwalten der kritischen Ressource Blockabschnitt jeweils ein protected object. Dabei ist Aushungern nur auf der ersten Ebene möglich. Ein weiterer Vorteil von Ada ist, daß Timeouts sowohl für den Client als auch für den Server existieren. Die Ada-Lösung benötigt als einzige keinen Kommunikationsaufwand.

CHILL besitzt zwar das Konzept der CHILL-Tasks zur direkten Kommunikation, jedoch eignen sich CHILL-Tasks — wie in [BE 98] dargestellt — zur Realisierung von reinen Servern, die nur bei Bedarf, das heißt bei Aufruf durch einen Client, gewisse Aktionen ausführen. Bezogen auf die Kabinenbahn lassen sich CHILL-Tasks jedoch nicht einsetzen.

Statt dessen wurden Prozesse benutzt; die Kommunikation zwischen diesen Prozessen erfolgte mit Hilfe von Puffern. Um auch für das Senden zum Puffer ein Timeout angeben zu können, wurde als Puffergröße 0 gewählt.

Für die Verwaltung der Blockabschnitte wurden `REGION`s benutzt. Damit ist die CHILL-Lösung auf beiden Ebenen nicht lebendig; außerdem ist — um eine bedingte Ausführung der kritischen Prozeduren zu erreichen — zusätzlicher Kommunikationsaufwand erforderlich.

Erlang unterstützt nur direkte Kommunikation zwischen Prozessen mittels asynchronen Senden und Empfangen von Nachrichten. Somit wurden hier die Blockabschnitte in Prozessen verwaltet. Ein synchroner Ablauf der Kommunikation wurde durch je 2 Sendeoperationen (`!`) und 2 `receive` erreicht, was allerdings den Kommunikationsaufwand erhöhte.

Ein Problem trat bei den Timeouts auf. Da das Senden von Nachrichten asynchron erfolgt, ist es nicht möglich, dabei ein Timeout anzugeben. Somit war es erforderlich, in einigen Funktionen Änderungen vorzunehmen. Das Erlang-Programm ist als einziges lebendig.

Den meisten Aufwand bei der Erstellung des Erlang-Programms bereitete die Verwaltung der Daten. Dabei erwies sich das Fehlen komplexerer Datenstrukturen als gravierender Nachteil. Statt dessen mußte mit geschachtelten Tupeln gearbeitet werden. Außerdem war es notwendig, zusätzliche Funktionen für die Manipulation der Daten zu definieren, was letztlich zu einem Anwachsen der Programmgröße führte.

| **Verwendete Sprache** | Ada | CHILL | Erlang | Java |
|---|---|---|---|---|
| **Lesbarkeit** | 2.5 | 3 | 3 | 3 |
| **Programmgröße** | 491 | 466 | 627 | 732 |
| **Kommunikationsaufwand** | - | 18 | 52 | 30 |
| **Lebendigkeit** | | | | |
| auf Ebene 1 | nein | nein | - | nein |
| auf Ebene 2 | ja | nein | ja | nein |
| **Verklemmung** | nein | nein | nein | nein |
| **Aushungern** | | | | |
| auf Ebene 1 | ja | ja | - | ja |
| auf Ebene 2 | nein | ja | nein | ja |
| **Erfordernis von fairer Maschine** | | | | |
| auf Ebene 1 | ja | ja | - | ja |
| auf Ebene 2 | nein | ja | nein | ja |
| **Nebenläufigkeitsgrad** | 1.12 | 1.12 | 1.12 | 1.12 |

**Tabelle 4.3** Zusammenfassung und Gegenüberstellung der Ergebnisse

Da Java keine direkte Kommunikation zwischen Threads unterstützt, mußten passive Kommunikationsobjekte erstellt werden, mit deren Hilfe eine synchrone Kommunikation zwischen den Threads möglich ist. Hierfür wurden synchronisierte Methoden benutzt; die bedingte Ausführung dieser synchronisierten Methoden wurde mittels `wait` und `notifyAll` erreicht. Die Verwaltung der Blockabschnitte erfolgte ebenfalls mittels synchronisierter Methoden; bedingte Ausführung wurde auch hier mit `wait` und `notifyAll` erreicht. Die somit entstandene Lösung ist auf beiden Ebenen nicht lebendig; zusätzlicher Kommunikationsaufwand ist erforderlich.

In Java sind Variablen eines Objekttyps stets Referenzen auf Objekte. Da es aber nicht möglich ist, solche Referenzen explizit zu dereferenzieren — um auf das Objekt als Ganzes zugreifen zu können — war es bei einigen Klassen erforderlich, die Methoden `clone()` und `equals()` zu reimplementieren. Das führte allerdings zum Anwachsen der Programmgröße.

Tabelle 4.3 faßt die Ergebnisse noch einmal zusammen.

Auf der beiliegenden CD-ROM sind die abgebildeten Quellen im Verzeichnis `broemel\Kabinenbahn` in folgenden Dateien enthalten:

| | |
|---|---|
| Ada: | `kabinenbahn.ads`, `kabinenbahn.adb` |
| Erlang: | `abschnittskontrolle.erl`, `kartenverkauf.erl`, `kabinensteuerung.erl`, `taxifahren.erl`, `streckenbestimmung.erl`, `initialisieren.erl` |
| Java: | `kabinenbahn.java` |

Da das zur Verfügung stehende VISION O.N.E. System Timeouts und Echtzeitbezug nicht unterstützt, konnte keine Lösung in VISION O.N.E. CHILL angegeben werden.

# Kapitel 5

# Der Bakery-Algorithmus

## 5.1 Aufgabenstellung

Der in [Ben 90] dargestellte Bakery-Algorithmus soll in Ada95, CHILL, Erlang und Java implementiert und die einzelnen Lösungen miteinander verglichen werden. Allerdings gehen wir dabei nicht von dem in [Ben 90] benutzten abgeschwächten Modell bezüglich des Zugriffs zu gemeinsamen Speicherbereichen aus. Wir verlangen dagegen, daß Schreibzugriffe auf gemeinsame Variablen stets exklusiv erfolgen; Lesezugriffe dürfen (müssen aber nicht) konkurrierend erfolgen. Damit ergibt sich der in Abbildung 5.1 dargestellte Pseudo-Code für den Bakery-Algorithmus.

```
Wähle_Nummer : array (1 .. N) of Integer := (others => 0);
Nummer : array (1 .. N) of Integer := (others => 0);

task body Pi is
   I : constant Integer := ...; -- Task Id
begin
   loop
      Nicht_Kritischer_Abschnitt_I;
      <<Wähle_Nummer(I) := 1>>;
      <<Nummer(I) := 1 + max(Nummer)>>;
      <<Wähle_Nummer(I) := 0>>;
      for J in 1 .. N loop
         if J /= I then
            loop
               exit when <Wähle_Nummer(J) = 0>;
            end loop;
            loop
               exit when <Nummer(J) = 0 or Nummer(I) < Nummer(J) or
                          (Nummer(I) = Nummer(J) and I < J)>;
            end loop;
         end if;
      end loop;
      Kritischer_Abschnitt_I;
      <<Nummer(I) := 0>>;
   end loop;
end Pi;
```

**Abbildung 5.1** Der Bakery-Algorithmus

Die spitzen Klammern in Abbildung 5.1 kennzeichnen Aktionen, bei denen ein Zugriff auf gemeinsame Variablen erfolgt. Schreibzugriffe werden durch << >> gekennzeichnet, Lesezugriffe durch < >.

Der Bakery-Algorithmus arbeitet nach folgendem Prinzip: Jeder Prozeß, der seinen kritischen Abschnitt betreten möchte, muß vorher eine Nummer ziehen. Diese Nummer ist größer als alle aktuell vergebenen Nummern. Ein Prozeß darf seinen kritischen Abschnitt erst dann betreten, wenn seine Nummer kleiner ist als die Nummern aller anderen Prozesse. Für den Fall, daß zwei Prozesse die gleiche niedrigste Nummer haben, darf der Prozeß mit der niedrigeren Prozeß-Id zuerst seinen kritischen Abschnitt betreten.

Das Array `Wähle_Nummer` zeigt an, welcher Prozeß gerade eine Nummer wählt; d.h. wenn der Wert von `Wähle_Nummer(I)` Eins ist, so signalisiert dies, daß der Prozeß `I` gerade eine Nummer wählt.

Im Array `Nummer` werden die gezogenen Nummern der einzelnen Prozesse gespeichert.

## 5.2 Der Bakery-Algorithmus in Ada

### 5.2.1 Programmentwurf

Die Verwaltung der globalen Variablen `Wähle_Nummer`, `Nummer` und `Max_Nummer` (die größte aktuell vergebene Nummer) erfolgt innerhalb eines protected object; der Zugriff zu diesen Variablen geschieht mittels der Operationen des protected object.

### 5.2.2 Programm

```
generic
   Anzahl : Positive;
package Bakery is
   procedure Start;
end Bakery;

package body Bakery is
   subtype Task_Nummer is Positive range 1 .. Anzahl;
   type Nummern_Feld is array (Task_Nummer) of Natural;

   protected Bakery_Objekt is
      procedure Beginn_Ziehe_Nummer(I : Task_Nummer);
      procedure Ende_Ziehe_Nummer(I : Task_Nummer);
      procedure Ziehe_Nummer(I : Task_Nummer);
      procedure Setze_Nummer_Zurück(I : Task_Nummer);
      function Zieht_Nummer(I : Task_Nummer) return Boolean;
      function Hat_Vorrang_Vor(I, J : Task_Nummer) return Boolean;
   private
      Wähle_Nummer : Nummern_Feld := (others => 0);
      Nummer : Nummern_Feld := (others => 0);
      Max_Nummer : Natural := 0;
   end Bakery_Objekt;

   protected body Bakery_Objekt is
      procedure Beginn_Ziehe_Nummer (I : Task_Nummer) is
      begin
         Wähle_Nummer(I) := 1;
      end Beginn_Ziehe_Nummer;
```

```
   procedure Ende_Ziehe_Nummer (I : Task_Nummer) is
   begin
      Wähle_Nummer(I) := 0;
   end Ende_Ziehe_Nummer;

   procedure Ziehe_Nummer (I : Task_Nummer) is
   begin
      Nummer(I) := 1 + Max_Nummer;
      Max_Nummer := Nummer(I);
   end Ziehe_Nummer;

   procedure Setze_Nummer_Zurück(I : Task_Nummer) is
   begin
      if Nummer(I) = Max_Nummer then
         --Maximum muß neu bestimmt werden
         Max_Nummer := 0;
         for J in 1 .. Anzahl loop
            if I /= J and Nummer(J) > Max_Nummer then
               Max_Nummer := Nummer(J);
            end if;
         end loop;
      end if;
      Nummer(I) := 0;
   end setze_nummer_zurück;

   function Zieht_Nummer(I : Task_Nummer) return Boolean is
   begin
      return Wähle_Nummer(I) = 1;
   end Zieht_Nummer;

   function Hat_Vorrang_Vor(I, J : Task_Nummer) return Boolean is
   begin
      return Nummer(J) = 0 or Nummer(I) < Nummer(J) or
        (Nummer(I) = Nummer(J) and I < J);
   end Hat_Vorrang_Vor;
end Bakery_Objekt;

task type Prozeß (I : Task_Nummer);
task body Prozeß is
   procedure Nicht_Kritischer_Abschnitt (I : Task_Nummer) is
   begin
      --Anweisungen des nicht kritischen Abschnitts des Task I
      null;
   end Nicht_Kritischer_Abschnitt;
   procedure Kritischer_Abschnitt (I : Task_Nummer) is
   begin
      --Anweisungen des kritischen Abschnitts des Task I
      null;
   end Kritischer_Abschnitt;
begin
```

```
      loop
         Nicht_Kritischer_Abschnitt(I);
         Bakery_Objekt.Beginn_Ziehe_Nummer(I);
         Bakery_Objekt.Ziehe_Nummer(I);
         Bakery_Objekt.Ende_Ziehe_Nummer(I);
         for J in 1 .. Anzahl loop
            if J /= I then
               loop
                  exit when not Bakery_Objekt.Zieht_Nummer(J);
               end loop;
               loop
                  exit when Bakery_Objekt.Hat_Vorrang_Vor(I, J);
               end loop;
            end if;
         end loop;
         Kritischer_Abschnitt(I);
         Bakery_Objekt.Setze_Nummer_Zurück(I);
      end loop;
   end Prozeß;

   procedure Start is
      type Zeiger_Auf_Prozeß is access Prozeß;
      Prozeß_Feld : array(1 .. Anzahl) of Zeiger_Auf_Prozeß;
   begin
      for I in 1 .. Anzahl loop
         Prozeß_Feld(I) := new Prozeß(I);
      end loop;
   end Start;
end Bakery;
```

**Abbildung 5.2** Der Bakery-Algorithmus in Ada

Das protected object `Bakery_Objekt` verwaltet die globalen Variablen und stellt die Operationen für den Zugriff auf die globalen Variablen zur Verfügung. Da die Ausführung dieser Operationen nicht von einer Bedingung abhängt, wurden keine protected entries, sondern protected procedures und protected functions benutzt. Der Einsatz von protected functions erlaubt konkurrierenden Lesezugriff zu den globalen Variablen.

### 5.2.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2 vergeben.

#### Programmgröße

Die Programmgröße beträgt 94 nloc.

#### Kommunikationsaufwand

Es ist kein Kommunikationsaufwand erforderlich.

### Lebendigkeit

Da es beim Wettbewerb der Tasks um das dem protected object zugehörige Lock keine Ordnung gibt, ist das Programm auf der 1. Ebene nicht lebendig.

Die Ausführung der Operationen des protected object ist nicht an eine Bedingung geknüpft; somit kann ein Task nach Erlangen des Locks die gewünschte Operation immer ausführen. Das heißt, auf der 2. Ebene ist das Programm lebendig.

### Verklemmung

Verklemmung kann in folgenden Situationen eintreten:

1. Ein Task hält während der Ausführung einer Operation des protected object an.
2. Die Ada-Maschine ist unfair; d.h. es existieren Tasks, die beim Wettbewerb um das Lock des protected object unendlich oft übergangen werden.

Da wir davon ausgehen, daß kein Task während seiner Ausführung anhält, entfällt die erste Möglichkeit.

Ist die Ada-Maschine unfair, so kann das bei bestimmten Ausführungen zur Verklemmung führen. Tabelle 5.1 zeigt eine solche Ausführung. Dabei wird angenommen, daß Task1 beim Wettbewerb um das Lock stets Vorrang gegenüber Task2 hat. Ein " — " in der Ausführung von Task2 zeigt an, daß Task2 blockiert ist, weil er auf das Lock des protected object wartet.

| Task1 | Task2 |
|---|---|
| `Nicht_Kritischer_Abschnitt(1)` | `Nicht_Kritischer_Abschnitt(2)` |
| `Beginn_Ziehe_Nummer(1)` | — |
| `Ziehe_Nummer(1)` | — |
| `Ende_Ziehe_Nummer(1)` | — |
| `J /= I --false` | `Beginn_Ziehe_Nummer(2)` |
| `J /= I --true` | `Ziehe_Nummer(2)` |
| `Zieht_Nummer(2) --true` | — |
| `Zieht_Nummer(2) --true` | — |
| `Zieht_Nummer(2) --true` | — |
| ⋮ | ⋮ |

**Tabelle 5.1** Verklemmung bei unfairer Maschine

### Aushungern

Da es beim Wettbewerb der Tasks um das Lock des protected object keine Ordnung gibt, kann es passieren, daß ein Task bei diesem Wettbewerb unendlich oft übergangen wird. Das bedeutet, Aushungern ist auf Ebene 1 möglich.

### Faire Maschine

Für Ebene 1 ist eine faire Maschine erforderlich.

### Nebenläufigkeitsgrad

Für die Betrachtung des Nebenläufigkeitsgrades benutzen wir eine Ausführung mit 2 Akteuren, wobei jeder Akteur genau einmal ausgeführt wird.

Aufgrund der großen Anzahl von möglichen Interleavings werden wir — im Gegensatz zu Kapitel 1, 3 und 4 — diesmal den maximalen Nebenläufigkeitsgrad betrachten. Dazu wird zunächst der maximale theoretische Nebenläufigkeitsgrad für 2 Akteure bestimmt und danach wird untersucht, ob dieser maximale theoretische Nebenläufigkeitsgrad erreicht werden kann.

Wie in Tabelle 5.2 zu sehen ist, führt jeder Task 12 Operationen aus. Damit werden insgesamt 24 Operationen ausgeführt. (`Zieht_Nummer` und `Hat_Vorrang_Vor` können zwar von einem Task mehrfach ausgeführt werden; da dies aber einem "busy waiting" entspricht, werden `Zieht_Nummer` und `Hat_Vorrang_Vor` nur einmal gezählt.)

| Schritt | Task1 | Task2 |
|---|---|---|
| 1 | `Nicht_Kritischer_Abschnitt(1)` | `Nicht_Kritischer_Abschnitt(2)` |
| 2 | `Beginn_Ziehe_Nummer(1)` | `Beginn_Ziehe_Nummer(2)` |
| 3 | `Ziehe_Nummer(1)` | `Ziehe_Nummer(2)` |
| 4 | `Ende_Ziehe_Nummer(1)` | `Ende_Ziehe_Nummer(2)` |
| 5 | `J /= I --false` | `J /= I --true` |
| 6 | `J /= I --true` | `Zieht_Nummer(1)` |
| 7 | `Zieht_Nummer(2)` | `exit` |
| 8 | `exit` | `Hat_Vorrang_Vor(2, 1)` |
| 9 | `Hat_Vorrang_Vor(1, 2)` | `exit` |
| 10 | `exit` | `J /= I --false` |
| 11 | `Kritischer_Abschnitt(1)` | `Kritischer_Abschnitt(2)` |
| 12 | `Setze_Nummer_Zurück(1)` | `Setze_Nummer_Zurück(2)` |

**Tabelle 5.2** Betrachtung des theoretischen Nebenläufigkeitsgrades

Für die Bestimmung des maximalen theoretischen Nebenläufigkeitsgrades benötigen wir noch die Anzahl der Zeiteinheiten, die für die 24 Operationen notwendig sind. Ein Schritt aus Tabelle 5.2 benötige eine Zeiteinheit. Die Anzahl der notwendigen Zeiteinheiten läßt sich aus folgenden Überlegungen bestimmen:

1. Da jeweils nur ein Prozeß in seinem kritischen Abschnitt sein darf, liefert `Hat_Vorrang_Vor` des Prozesses mit der höheren gezogenen Nummer erst dann `true`, nachdem der Prozeß mit der niedrigeren gezogenen Nummer `Setze_Nummer_Zurück` ausgeführt hat. Damit sind mindestens 4 (Task2 hat die niedrigere gezogene Nummer) bzw. 5 (Task1 hat die niedrigere gezogene Nummer) zusätzliche Zeiteinheiten notwendig. Sei Task2 der Task mit der niedrigeren gezogenen Nummer, d.h. es sind mindestens 16 Zeiteinheiten notwendig.

2. Damit `Zieht_Nummer(1)` den Wert `false` hat (und somit Task2 nicht warten muß), gibt es 2 Möglichkeiten:

   (a) Task1 hat `Ende_Ziehe_Nummer(1)` bereits ausgeführt. In diesem Fall sind — aufgrund der Exklusivität der Schritte 2, 3 und 4 — mindestens 2 weitere Zeiteinheiten erforderlich. Das ergibt 18 Zeiteinheiten für diesen Fall.

   (b) Task1 hat `Beginn_Ziehe_Nummer(1)` noch nicht ausgeführt. In diesem Fall könnte Task1 seinen Schritt 2 frühestens zum gleichen Zeitpunkt ausführen, wie Task2 seinen Schritt 7. Damit wäre Task1 in seiner Ausführung 5 Zeiteinheiten hinter Task2 zurück, d.h. die Gesamtausführung würde 17 Zeiteinheiten dauern. Da aber Schritt 3 von Task1 und Schritt 8 von Task2 nicht gleichzeitig ausgeführt werden können, ist eine weitere Zeiteinheit notwendig. Das bedeutet, es sind auch in diesem Fall mindestens 18 Zeiteinheiten notwendig.

Damit sind insgesamt 18 Zeiteinheiten für die Ausführung der 24 Operationen erforderlich. Der maximale theoretische Nebenläufigkeitsgrad beträgt somit $\frac{24}{18} = \frac{4}{3} = 1.33$.

| Zeiteinheit | Task1 | Task2 |
|---|---|---|
| $t_1$ | `Nicht_Kritischer_Abschnitt(1)` | `Nicht_Kritischer_Abschnitt(2)` |
| $t_2$ | — | `Beginn_Ziehe_Nummer(2)` |
| $t_3$ | `Beginn_Ziehe_Nummer(1)` | — |
| $t_4$ | — | `Ziehe_Nummer(2)` |
| $t_5$ | `Ziehe_Nummer(1)` | — |
| $t_6$ | — | `Ende_Ziehe_Nummer(2)` |
| $t_7$ | `Ende_Ziehe_Nummer(1)` | `J /= I --true` |
| $t_8$ | `J /= I --false` | `Zieht_Nummer(1) --false` |
| $t_9$ | `J /= I --true` | `exit` |
| $t_{10}$ | `Zieht_Nummer(2) --false` | `Hat_Vorrang_Vor(2, 1) --true` |
| $t_{11}$ | `exit` | `exit` |
| $t_{12}$ | `Hat_Vorrang_Vor(1, 2) --false` | `J /= I --false` |
| $t_{13}$ | `Hat_Vorrang_Vor(1, 2) --false` | `Kritischer_Abschnitt(2)` |
| $t_{14}$ | — | `Setze_Nummer_Zurück(2)` |
| $t_{15}$ | `Hat_Vorrang_Vor(1, 2) --true` | |
| $t_{16}$ | `exit` | |
| $t_{17}$ | `Kritischer_Abschnitt(1)` | |
| $t_{18}$ | `Setze_Nummer_Zurück(1)` | |

**Tabelle 5.3** Betrachtung des tatsächlichen Nebenläufigkeitsgrades

Zur Bestimmung des tatsächlich erreichten maximalen Nebenläufigkeitsgrades betrachten wir Tabelle 5.3. Die Ausführung aus Tabelle 5.3 benötigt 18 Zeiteinheiten. Damit beträgt der Nebenläufigkeitsgrad für diese Ausführung $\frac{24}{18} = \frac{4}{3} = 1.33$. Das bedeutet, der maximale theoretische Nebenläufigkeitsgrad wurde erreicht.

## 5.3 Der Bakery-Algorithmus in CHILL

### 5.3.1 Programmentwurf

Die Verwaltung der globalen Variablen `Wähle_Nummer`, `Nummer` und `Max_Nummer` erfolgt innerhalb einer `REGION`; der Zugriff zu diesen Variablen geschieht mit Hilfe der kritischen Prozeduren der `REGION`. Damit wird die Atomarität aller Zugriffe auf die globalen Variablen sichergestellt.

### 5.3.2 Programm

```
bakery: GENERIC
   SEIZE positive, natural;
   SYN anzahl = positive;
MODULE
   GRANT start;
   SYNMODE bakery_objekt_mode = REGION SPEC
      GRANT beginn_ziehe_nummer, ende_ziehe_nummer, ziehe_nummer,
            setze_nummer_zurueck, zieht_nummer, hat_vorrang_vor;
      SYNMODE task_nummer = INT(1:anzahl);
      DCL nummer, waehle_nummer ARRAY(task_nummer) natural := [(ELSE):0];
```

```
   DCL max_nummer natural := 0;

   beginn_ziehe_nummer: PROC (i task_nummer);
   ende_ziehe_nummer: PROC (i task_nummer);
   ziehe_nummer: PROC (i task_nummer);
   setze_nummer_zurueck: PROC (i task_nummer);
   zieht_nummer: PROC(i task_nummer) RETURNS (BOOL);
   hat_vorrang_vor: PROC (i, j task_nummer) RETURNS (BOOL);
END bakery_objekt_mode;
SYNMODE bakery_objekt_mode = REGION BODY
   beginn_ziehe_nummer: PROC (i task_nummer);
      waehle_nummer(i) := 1;
   END beginn_ziehe_nummer;

   ende_ziehe_nummer: PROC (i task_nummer);
      waehle_nummer(i) := 0;
   END ende_ziehe_nummer;

   ziehe_nummer: PROC (i task_nummer);
      nummer(i) := 1 + max_nummer;
      max_nummer := nummer(i);
   END ziehe_nummer;

   setze_nummer_zurueck: PROC (i task_nummer);
      IF nummer(i) = max_nummer THEN
         /* max_nummer muss neu bestimmt werden */
         max_nummer := 0;
         DO FOR j IN task_nummer;
            IF nummer(j) > max_nummer AND j /= i THEN
               max_nummer := nummer(j);
            FI;
         OD;
      FI;
      nummer(i) := 0;
   END setze_nummer_zurueck;

   zieht_nummer: PROC (i task_nummer) RETURNS (BOOL);
      RETURN waehle_nummer(i) = 1;
   END zieht_nummer;

   hat_vorrang_vor: PROC (i, j task_nummer) RETURNS (BOOL);
      RETURN nummer(j) = 0 OR nummer(i) < nummer(j) OR
            (nummer(i) = nummer(j) AND i < j);
   END hat_vorrang_vor;
END bakery_objekt_mode;

/* Definition des REGION-Objektes */
DCL bakery_objekt bakery_objekt_mode;

/* Definition des Prozesstyps */
prozess: PROCESS (i task_nummer);
```

```
      nicht_kritischer_abschnitt: PROC (i task_nummer);
         /* Anweisungen des nicht kritischen Abschnitts des Prozesses i */
         ;
      END nicht_kritischer_abschnitt;
      kritischer_abschnitt: PROC (i task_nummer);
         /* Anweisungen des kritischen Abschnitts des Prozesses i */
         ;
      END kritischer_abschnitt;

      DO FOR EVER;
         nicht_kritischer_abschnitt(i);
         bakery_objekt.beginn_ziehe_nummer(i);
         bakery_objekt.ziehe_nummer(i);
         bakery_objekt.ende_ziehe_nummer(i);
         DO FOR j := 1 TO anzahl;
            IF j /= i THEN
               DO FOR EVER;
                  IF NOT bakery_objekt.zieht_nummer(j) THEN
                     EXIT;
                  FI;
               OD;
               DO FOR EVER;
                  IF bakery_objekt.hat_vorrang_vor(i, j) THEN
                     EXIT;
                  FI;
               OD;
            FI;
         OD;
         kritischer_abschnitt(i);
         bakery_objekt.setze_nummer_zurueck(i);
      OD;
   END prozess;

   start: PROC ();
      DO FOR i := 1 TO anzahl;
         START prozess(i);
      OD;
   END start;
END bakery;
```

**Abbildung 5.3** Der Bakery-Algorithmus in CHILL

### 5.3.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.5 vergeben.

#### Programmgröße

Die Programmgröße beträgt 84 nloc.

#### Kommunikationsaufwand

Es ist kein Kommunikationsaufwand erforderlich.

#### Lebendigkeit

Da es beim Wettbewerb der Prozesse um das Lock der `REGION` keine Ordnung gibt, ist das Programm auf der 1. Ebene nicht lebendig.

Die Ausführung der Operationen der `REGION` ist nicht an eine Bedingung geknüpft; somit kann ein Prozeß nach Erlangen des Locks die gewünschte Operation immer ausführen. Das heißt, auf der 2. Ebene ist das Programm lebendig.

#### Verklemmung

Ist die CHILL-Maschine unfair, so kann — analog zur Ada-Lösung — bei bestimmten Ausführungen Verklemmung auftreten. Eine solche Ausführung ist in Tabelle 5.1 dargestellt.

#### Aushungern

Da es beim Wettbewerb der Prozesse um das Lock der `REGION` keine Ordnung gibt, kann es passieren, daß ein Prozeß bei diesem Wettbewerb unendlich oft übergangen wird. Das bedeutet, Aushungern ist auf Ebene 1 möglich.

#### Faire Maschine

Für Ebene 1 ist eine faire Maschine erforderlich.

#### Nebenläufigkeitsgrad

Tabelle 5.4 zeigt eine Ausführung, die 18 Zeiteinheiten benötigt und somit einen Nebenläufigkeitsgrad von $\frac{24}{18} = \frac{4}{3} = 1.33$ besitzt. Damit wurde auch in CHILL der maximale theoretische Nebenläufigkeitsgrad erreicht.

## 5.4 Der Bakery-Algorithmus in Erlang

### 5.4.1 Programmentwurf

Da Erlang keine passiven Kommunikationsobjekte besitzt, muß die Verwaltung der globalen Variablen in einem Prozeß erfolgen. Der Zugriff wird über das Senden und Empfangen von Nachrichten geregelt.

### 5.4.2 Programm

```
-module(bakery).
-export([bakery/1, start/1, prozess/2]).

start(Anzahl) ->
   register(bakery_objekt, spawn(bakery, bakery, [Anzahl])),
   starte_prozesse(Anzahl).

starte_prozesse(Anzahl) ->
   starte_prozesse(1, Anzahl).
```

| Zeiteinheit | Prozeß1 | Prozeß2 |
|---|---|---|
| $t_1$ | `nicht_kritischer_abschnitt(1)` | `nicht_kritischer_abschnitt(2)` |
| $t_2$ | — | `beginn_ziehe_nummer(2)` |
| $t_3$ | `beginn_ziehe_nummer(1)` | — |
| $t_4$ | — | `ziehe_nummer(2)` |
| $t_5$ | `ziehe_nummer(1)` | — |
| $t_6$ | — | `ende_ziehe_nummer(2)` |
| $t_7$ | `ende_ziehe_nummer(1)` | `j /= i /*TRUE*/` |
| $t_8$ | `j /= i /*FALSE*/` | `zieht_nummer(1) /*FALSE*/` |
| $t_9$ | `j /= i /*TRUE*/` | `EXIT` |
| $t_{10}$ | — | `hat_vorrang_vor(2, 1) /*TRUE*/` |
| $t_{11}$ | `zieht_nummer(2) /*FALSE*/` | `EXIT` |
| $t_{12}$ | `EXIT` | `j /= i /*FALSE*/` |
| $t_{13}$ | `hat_vorrang_vor(1, 2) /*FALSE*/` | `kritischer_abschnitt(2)` |
| $t_{14}$ | — | `setze_nummer_zurueck(2)` |
| $t_{15}$ | `hat_vorrang_vor(1, 2) /*TRUE*/` | |
| $t_{16}$ | `EXIT` | |
| $t_{17}$ | `kritischer_abschnitt(1)` | |
| $t_{18}$ | `setze_nummer_zurueck(1)` | |

**Tabelle 5.4** Betrachtung des Nebenläufigkeitsgrades

```
starte_prozesse(I, Anzahl) ->
   spawn(bakery, prozess, [I, Anzahl]),
   if
      I == Anzahl ->
         true;
      I  < Anzahl ->
         starte_prozesse(I+1, Anzahl)
   end.

bakery(Anzahl) ->
   Nummer = list_to_tuple(erzeuge_anfangsliste(Anzahl)),
   Waehle_Nummer = Nummer,
   Max_Nummer = 0,
   bakery1(Nummer, Waehle_Nummer, Max_Nummer).

erzeuge_anfangsliste(Anzahl) ->
   if Anzahl == 1 ->
         [0];
      Anzahl >  1 ->
         lists:append([0], erzeuge_anfangsliste(Anzahl-1))
   end.

beginn_ziehe_nummer(I) ->
   bakery_objekt ! {self(), beginn_ziehe_nummer, I},
   receive
      {bakery_objekt, ok} ->
         true
   end.
```

```
ende_ziehe_nummer(I) ->
   bakery_objekt ! {self(), ende_ziehe_nummer, I},
   receive
      {bakery_objekt, ok} ->
         true
   end.

ziehe_nummer(I) ->
   bakery_objekt ! {self(), ziehe_nummer, I},
   receive
      {bakery_objekt, ok} ->
         true
   end.

setze_nummer_zurueck(I) ->
   bakery_objekt ! {self(), setze_nummer_zurueck, I},
   receive
      {bakery_objekt, ok} ->
         true
   end.

zieht_nummer(I) ->
   bakery_objekt ! {self(), zieht_nummer, I},
   receive
      {bakery_objekt, Ergebnis} ->
         Ergebnis
   end.

hat_vorrang_vor(I, J) ->
   bakery_objekt ! {self(), hat_vorrang_vor, I, J},
   receive
      {bakery_objekt, Ergebnis} ->
         Ergebnis
   end.

bakery1(Nummer, Waehle_Nummer, Max_Nummer) ->
   receive
      {Pid, beginn_ziehe_nummer, I} ->
         Waehle_Nummer_Neu = setelement(I, Waehle_Nummer, 1),
         Pid ! {bakery_objekt, ok},
         bakery1(Nummer, Waehle_Nummer_Neu, Max_Nummer);
      {Pid, ende_ziehe_nummer, I} ->
         Waehle_Nummer_Neu = setelement(I, Waehle_Nummer, 0),
         Pid ! {bakery_objekt, ok},
         bakery1(Nummer, Waehle_Nummer_Neu, Max_Nummer);
      {Pid, ziehe_nummer, I} ->
         Nummer_Neu = setelement(I, Nummer, Max_Nummer + 1),
         Pid ! {bakery_objekt, ok},
         bakery1(Nummer_Neu, Waehle_Nummer, Max_Nummer + 1);
      {Pid, setze_nummer_zurueck, I} ->
```

```
            Nummer_Neu = setelement(I, Nummer, 0),
            Pid ! {bakery_objekt, ok},
            if
               element(I, Nummer) == Max_Nummer ->
                  %Max_Nummer muss neu bestimmt werden
                  Max_Nummer_Neu = lists:max(tuple_to_list(Nummer_Neu)),
                  bakery1(Nummer_Neu, Waehle_Nummer, Max_Nummer_Neu);
               element(I, Nummer) /= Max_Nummer ->
                  bakery1(Nummer_Neu, Waehle_Nummer, Max_Nummer)
            end;
         {Pid, zieht_nummer, I} ->
            if
               element(I, Waehle_Nummer) == 0 ->
                  Pid ! {bakery_objekt, false};
               element(I, Waehle_Nummer) == 1 ->
                  Pid ! {bakery_objekt, true}
            end,
            bakery1(Nummer, Waehle_Nummer, Max_Nummer);
         {Pid, hat_vorrang_vor, I, J} ->
            if
               element(J, Nummer) == 0 ->
                  Pid ! {bakery_objekt, true};
               element(I, Nummer) < element(J, Nummer) ->
                  Pid ! {bakery_objekt, true};
               element(I, Nummer) == element(J, Nummer), I < J ->
                  Pid ! {bakery_objekt, true};
               true ->
                  Pid ! {bakery_objekt, false}
            end,
            bakery1(Nummer, Waehle_Nummer, Max_Nummer)
      end.

nicht_kritischer_abschnitt(I) ->
      % Anweisungen fuer den nicht kritischen Abschnitt von prozess I
      true.

kritischer_abschnitt(I) ->
      % Anweisungen fuer den kritischen Abschnitt von prozess I
      true.

prozess(I, Anzahl) ->
      nicht_kritischer_abschnitt(I),
      beginn_ziehe_nummer(I),
      ziehe_nummer(I),
      ende_ziehe_nummer(I),
      protokoll(I, Anzahl),
      kritischer_abschnitt(I),
      setze_nummer_zurueck(I),
      prozess(I, Anzahl).

protokoll(I, Anzahl) ->
```

```
   protokoll(I, 1, Anzahl).

protokoll(I, J, Anzahl) ->
   if
      I /= J ->
         test1(J),
         test2(I, J);
      I == J ->
         true
   end,
   if
      J == Anzahl ->
         true;
      J <  Anzahl ->
         protokoll(I, J+1, Anzahl)
   end.

test1(J) ->
   % benutzerdefinierte Funktionen duerfen nicht im Guard des if benutzt
   % werden; deshalb erfolgt Zuweisung an Variable Zieht_Nummer,
   % diese Variable wird dann im Guard des if ueberprueft
   Zieht_Nummer = zieht_nummer(J),
   if
      Zieht_Nummer == false ->
         true;
      Zieht_Nummer == true ->
         test1(J)
   end.

test2(I, J) ->
   Hat_Vorrang_Vor = hat_vorrang_vor(I, J),
   if
      Hat_Vorrang_Vor == true ->
         true;
      Hat_Vorrang_Vor == false ->
         test2(I, J)
   end.
```

**Abbildung 5.4** Der Bakery-Algorithmus in Erlang

Die Funktion `bakery1` verwaltet `Nummer`, `Waehle_Nummer` und `Max_Nummer` als Parameter; die Änderung der Werte dieser Größen erfolgt durch einen neuen Aufruf von `bakery1` mit den geänderten Parametern. Da in Erlang keine Arrays existieren, wurden `Nummer` und `Waehle_Nummer` als Tupel implementiert.

Der Zugriff zu den von `bakery1` verwalteten Variablen wird über Schnittstellenfunktionen geregelt; diese kommunizieren mit `bakery1` durch asynchrones Senden und Empfangen von Nachrichten. Ein synchroner Ablauf dieser Kommunikation wird durch Senden und Empfangen von jeweils 2 Nachrichten erreicht.

### 5.4.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 3 vergeben.

#### Programmgröße

Die Programmgröße beträgt 152 nloc.

#### Kommunikationsaufwand

Wir beziehen uns hierbei auf die Ausführung aus Tabelle 5.5, die auch Grundlage bei der Betrachtung des Nebenläufigkeitsgrades ist. Dabei erfolgen 13 Aufrufe einer Schnittstellenfunktion. Da

| Zeiteinheit | Prozeß1 | Prozeß2 |
|---|---|---|
| $t_1$ | `nicht_kritischer_abschnitt(1)` | `nicht_kritischer_abschnitt(2)` |
| $t_2$ | — | `beginn_ziehe_nummer(2)` |
| $t_3$ | `beginn_ziehe_nummer(1)` | — |
| $t_4$ | — | `ziehe_nummer(2)` |
| $t_5$ | `ziehe_nummer(1)` | — |
| $t_6$ | — | `ende_ziehe_nummer(2)` |
| $t_7$ | `ende_ziehe_nummer(1)` | `j /= i` `%true` |
| $t_8$ | `j /= i` `%false` | `zieht_nummer(1)` `%false` |
| $t_9$ | `j /= i` `%true` | `true` |
| $t_{10}$ | — | `hat_vorrang_vor(2, 1)` `%true` |
| $t_{11}$ | `zieht_nummer(2)` `%false` | `true` |
| $t_{12}$ | `true` | `j /= i` `%false` |
| $t_{13}$ | `hat_vorrang_vor(1, 2)` `%false` | `kritischer_abschnitt(2)` |
| $t_{14}$ | — | `setze_nummer_zurueck(2)` |
| $t_{15}$ | `hat_vorrang_vor(1, 2)` `%true` | |
| $t_{16}$ | `true` | |
| $t_{17}$ | `kritischer_abschnitt(1)` | |
| $t_{18}$ | `setze_nummer_zurueck(1)` | |

**Tabelle 5.5** Betrachtung des Nebenläufigkeitsgrades

für jeden solchen Aufruf 2 Sendeoperationen (`!`) und 2 `receive` erforderlich sind, ergibt sich ein Kommunikationsaufwand von $13 * 4 = 52$ Anweisungen.

#### Lebendigkeit

Da die Nachrichten in der Mailbox eines Prozesses in der Reihenfolge ihrer Ankunft abgelegt und verarbeitet werden, kann kein Prozeß übergangen werden. Das heißt, die Lösung ist lebendig.

#### Verklemmung

Verklemmung kann nur auftreten, wenn Prozesse in ihrer Ausführung anhalten. Da wir das aber ausschließen, ist die Lösung verklemmungsfrei.

#### Aushungern

Da Prozesse nicht übergangen werden können, ist Aushungern nicht möglich.

#### Faire Maschine

Es ist keine faire Maschine erforderlich.

#### Nebenläufigkeitsgrad

Für die in Tabelle 5.5 dargestellte Ausführung werden 18 Zeiteinheiten benötigt; damit beträgt der Nebenläufigkeitsgrad $\frac{24}{18} = \frac{4}{3} = 1.33$. Das bedeutet, der maximale theoretische Nebenläufigkeitsgrad wurde erreicht.

## 5.5 Der Bakery-Algorithmus in Java

### 5.5.1 Programmentwurf

In Java werden die globalen Variablen als private Variablen in einer Klasse definiert. Der Zugriff erfolgt mittels synchronisierter Methoden; das stellt die Atomarität der Zugriffe sicher.

### 5.5.2 Programm

```
class bakery {
   bakery (int anzahl) {
      bakery_klasse bakery_objekt = new bakery_klasse(anzahl);
      prozess[] prozess_feld = new prozess[anzahl];
      for (int i=0; i<anzahl; i++) {
         prozess_feld[i] = new prozess(i, bakery_objekt, anzahl);
         prozess_feld[i].start();
      }
   }
   class bakery_klasse {
      bakery_klasse (int anzahl) {
         nummer = new int[anzahl];
         waehle_nummer = new int[anzahl];
         max_nummer = 0;
         for (int i=0; i<anzahl; i++) {
            nummer[i] = 0;
            waehle_nummer[i] = 0;
         }
      }
      private int[] nummer;
      private int[] waehle_nummer;
      private int max_nummer;
      public synchronized void beginn_ziehe_nummer (int i) {
         waehle_nummer[i] = 1;
      }
      public synchronized void ende_ziehe_nummer (int i) {
         waehle_nummer[i] = 0;
      }
      public synchronized void ziehe_nummer (int i) {
         nummer[i] = 1 + max_nummer;
         max_nummer = nummer[i];
      }
```

```
   public synchronized void setze_nummer_zurueck (int i) {
      if (nummer[i] == max_nummer) {
         /* max_nummer muss neu bestimmt werden */
         max_nummer = 0;
         for (int j=0; j<nummer.length; j++)
            if (j != i && nummer[j] > max_nummer)
               max_nummer = nummer[j];
      }
      nummer[i] = 0;
   }
   public synchronized boolean zieht_nummer (int i) {
      return (waehle_nummer[i] == 1);
   }
   public synchronized boolean hat_vorrang_vor (int i, int j) {
      return (nummer[j] == 0 || nummer[i] < nummer[j] ||
             (nummer[i] == nummer[j] && i < j));
   }
}
class prozess extends Thread {
   prozess(int i, bakery_klasse bakery_objekt, int anzahl) {
      this.i = i;
      this.bakery_objekt = bakery_objekt;
      this.anzahl = anzahl;
   }
   private int i;
   private bakery_klasse bakery_objekt;
   private int anzahl;

   public void run() {
      while (true) {
         nicht_kritischer_abschnitt(i);
         bakery_objekt.beginn_ziehe_nummer(i);
         bakery_objekt.ziehe_nummer(i);
         bakery_objekt.ende_ziehe_nummer(i);
         for (int j=0; j<anzahl; j++) {
            if (j != i) {
               while (true)
                  if (!bakery_objekt.zieht_nummer(j))
                     break;
               while (true)
                  if (bakery_objekt.hat_vorrang_vor(i, j))
                     break;
            }
         }
         kritischer_abschnitt(i);
         bakery_objekt.setze_nummer_zurueck(i);
      }
   }
   private void nicht_kritischer_abschnitt (int i) {
      /* Anweisungen des nicht kritischen Abschnitts von prozess i */
      ;
```

```
        }
        private void kritischer_abschnitt (int i) {
           /* Anweisungen des kritischen Abschnitts von prozess i */
           ;
        }
     }
  }
```

**Abbildung 5.5** Der Bakery-Algorithmus in Java

### 5.5.3 Untersuchung der Programmlösung

#### Lesbarkeit

Für die Lesbarkeit wurde eine 2.5 vergeben.

#### Programmgröße

Die Programmgröße beträgt 86 nloc.

#### Kommunikationsaufwand

Es ist kein Kommunikationsaufwand erforderlich.

#### Lebendigkeit

Da es beim Wettbewerb der Threads um das Lock des Objektes keine Ordnung gibt, kann es passieren, daß Threads dabei unendlich oft übergangen werden. Das heißt, auf der 1. Ebene ist das Programm nicht lebendig.

Da die Ausführung der synchronisierten Methoden nicht an eine Bedingung geknüpft ist, führt ein Thread nach Erlangen des Locks die jeweilige Methode immer aus. Das heißt, auf der 2. Ebene ist das Programm lebendig.

#### Verklemmung

Ist die Java-Maschine unfair, so kann — analog zu den Lösungen in Ada und CHILL — bei bestimmten Ausführungen Verklemmung auftreten. Eine solche Ausführung ist in Tabelle 5.1 dargestellt.

#### Aushungern

Da es beim Wettbewerb der Threads um das Lock des Objektes keine Ordnung gibt, kann es passieren, daß ein Thread bei diesem Wettbewerb unendlich oft übergangen wird. Das bedeutet, Aushungern ist auf Ebene 1 möglich.

#### Faire Maschine

Für Ebene 1 ist eine faire Maschine erforderlich.

#### Nebenläufigkeitsgrad

Wir beziehen uns auf den Ablauf aus Tabelle 5.4. Dabei beträgt der Nebenläufigkeitsgrad $\frac{24}{18} = \frac{4}{3} = 1.33$. Das bedeutet, der maximale theoretische Nebenläufigkeitsgrad wurde auch in Java erreicht.

## 5.6 Zusammenfassung und Vergleich

In allen vier Sprachen ließ sich der Bakery-Algorithmus ohne Schwierigkeiten implementieren.

Die Ada-Lösung hat den Vorteil, daß mit Hilfe der Operationen eines protected object der Zugriff zu den gemeinsamen Variablen unterteilt werden kann in exklusiven Schreibzugriff und konkurrierenden Lesezugriff.

Die Lösungen in CHILL und Java unterscheiden sich kaum voneinander. Das liegt vor allem daran, daß keine bedingte Synchronisation vorlag; d.h. die Ausführung der exklusiven Operationen war nicht von einer Bedingung abhängig und somit war es nicht erforderlich, Akteure explizit zu blockieren bzw. zu reaktivieren. Somit ist auch kein Kommunikationsaufwand erforderlich.

Da in Ada, CHILL und Java keine Ordnung beim Wettbewerb der Akteure um das Lock des passiven Kommunikationsobjektes existiert, ist Aushungern auf Ebene 1 möglich. Außerdem kann es dadurch bei bestimmten Ausführungen zur Verklemmung kommen.

Die Erlang-Lösung ist als einzige lebendig und verklemmungsfrei. Allerdings sind für jeden Zugriff auf die geschützten Daten 2 Sendeoperationen (!) und 2 `receive` erforderlich. Das führt zu einem erhöhten Kommunikationsaufwand und zum Anwachsen der Programmgröße.

Der maximale theoretische Nebenläufigkeitsgrad wurde in allen Sprachen erreicht.

Tabelle 5.6 faßt die Ergebnisse noch einmal zusammen.

Auf der beiliegenden CD-ROM sind die abgebildeten Quellen im Verzeichnis `broemel\Bakery` in folgenden Dateien enthalten:

| | |
|---|---|
| Ada: | `bakery.ads`, `bakery.adb` |
| CHILL: | `bakery0a.src, bak_regs.src, bak_pros.src, bak_tsts.src` |
| Erlang: | `bakery.erl` |
| Java: | `bakery.java` |

| **Verwendete Sprache** | Ada | CHILL | Erlang | Java |
|---|---|---|---|---|
| **Lesbarkeit** | 2 | 2.5 | 3 | 2.5 |
| **Programmgröße** | 94 | 84 | 152 | 86 |
| **Kommunikationsaufwand** | - | - | 52 | - |
| **Lebendigkeit** | | | | |
| auf Ebene 1 | nein | nein | - | nein |
| auf Ebene 2 | ja | ja | ja | ja |
| **Verklemmung** | ja | ja | nein | ja |
| **Aushungern** | | | | |
| auf Ebene 1 | ja | ja | - | ja |
| auf Ebene 2 | nein | nein | nein | nein |
| **Erfordernis von fairer Maschine** | | | | |
| auf Ebene 1 | ja | ja | - | ja |
| auf Ebene 2 | nein | nein | nein | nein |
| **Nebenläufigkeitsgrad** | 1.33 | 1.33 | 1.33 | 1.33 |

**Tabelle 5.6** Zusammenfassung und Gegenüberstellung der Ergebnisse

# Kapitel 6

# Zusammenfassung

Das Ziel dieser Arbeit war es, die Nebenläufigkeitskonzepte der Sprachen Ada, CHILL, Erlang und Java zu vergleichen — und zwar anhand von konkreten Aufgabenstellungen. Aus den betrachteten Aufgaben lassen sich folgende Anforderungen an die Nebenläufigkeitskonzepte der Sprachen ableiten:

- Synchrone Kommunikation
- Koordinierter Zugriff auf gemeinsame Variablen
- Bedingte Synchronisation
- Echtzeitbezug
- Zeitüberwachung (Timeout)

Diese Anforderungen werden von den Sprachen in sehr unterschiedlicher Weise erfüllt.

## 6.1 Ada

Für die synchrone Kommunikation zwischen Tasks stellt Ada das Rendezvous zur Verfügung. Koordinierter Zugriff auf gemeinsame Variablen erfolgt durch die Verwendung von protected objects. Die protected objects erlauben die Unterteilung der Zugriffsart in konkurrierenden Lesezugriff (protected function) und exklusiven Schreib– und Lesezugriff (protected procedure und protected entry).

Bedingte Synchronisation wird mit Hilfe der protected entries erreicht; diese haben folgende Vorteile:

1. Durch die Trennung von Bedingung (entry barrier) und zugehörigen Anweisungen (entry body) werden Übersichtlichkeit und Lesbarkeit erhöht.
2. Das Blockieren und Reaktivieren von Tasks geschieht automatisch.
3. Durch die Ordnung innerhalb der Warteschlange eines protected entry und die Mechanismen bei der Ausführung einer protected action (siehe [Int 95]) wird Lebendigkeit auf der zweiten Ebene garantiert.

Echtzeitbezug erfolgt mittels der Funktion `clock` aus dem Paket `Ada.Calendar`. Für höhere Anforderungen an die gemessene Zeit (z.B. Genauigkeit) steht noch das Paket `Ada.Real_Time` zur Verfügung.

Timeouts sind — beim Ada-Rendezvous — sowohl auf Seiten des Servers als auch auf Seiten des Clients möglich. Für den Aufruf eines protected entry läßt sich ebenfalls ein Timeout angeben.

## 6.2 CHILL

CHILL besitzt kein Konstrukt für die synchrone Kommunikation zwischen Akteuren. Sowohl die direkte Kommunikation über CHILL-Tasks als auch die indirekte Kommunikation über Puffer läuft asynchron ab. Ein synchroner Ablauf der Kommunikation wurde durch zweifache asynchrone Kommunikation (über Puffer) erreicht.

Für den koordinierten Zugriff zu gemeinsamen Variablen stellt CHILL das Konstrukt der `REGION` zur Verfügung. Dabei handelt es sich um einen klassischen Monitor; alle Operationen laufen unter gegenseitigem Ausschluß ab.

Bedingte Synchronisation der kritischen Prozeduren einer `REGION` wird mit Hilfe von `EVENT`-Variablen und den auf `EVENT`-Variablen definierten Operationen `DELAY` und `CONTINUE` realisiert. Das bedeutet, Blockieren und Reaktivieren von Prozessen geschieht explizit ("von Hand"). Die zugehörigen Bedingungen werden innerhalb der kritischen Prozeduren überprüft; d.h. Bedingung und zugehörige Anweisungen sind nicht klar voneinander getrennt. Lebendigkeit wird weder auf der ersten noch auf der zweiten Ebene garantiert.

Für den Echtzeitbezug steht die Prozedur `ABSTIME` zur Verfügung. Timeouts lassen sich in CHILL für beliebige Anweisungsfolgen angeben.

## 6.3 Erlang

In Erlang ist Kommunikation zwischen Prozessen nur auf direktem Weg — durch Senden und Empfangen von Nachrichten — möglich. Da das Senden asynchron erfolgt, muß der synchrone Ablauf der Kommunikation durch zweifaches asynchrones Senden und Empfangen erzwungen werden. Das führt allerdings zu einem erhöhten Kommunikationsaufwand.

Für den koordinierten Zugriff auf gemeinsame Variablen existiert kein separates Sprachkonstrukt; koordinierter Zugriff wird durch Einsatz eines Verwaltungsprozesses VP erreicht. Da Erlang eine funktionale Sprache ist, werden die zu schützenden Daten als Parameter von VP verwaltet. Prozesse greifen auf die gemeinsamen Daten zu, indem sie Nachrichten an VP schicken und auf die entsprechenden Antworten warten. Das `receive` arbeitet selektiv; durch das Angeben einer Bedingung vor einer `receive`-Alternative läßt sich bedingte Synchronisation implementieren. Diese Art des koordinierten Zugriffs ist lebendig.

Echtzeitbezug geschieht mit Hilfe der Funktion `now()`. Timeouts lassen sich beim Empfangen von Nachrichten angeben. Das Senden hingegen ist asynchron; somit kann hier kein Timeout angegeben werden.

## 6.4 Java

Kommunikation zwischen Threads kann in Java nur indirekt über den Aufruf von synchronisierten Methoden bzw. synchronisierten Anweisungen erfolgen. Diese Art der Kommunikation ist zunächst asynchron; ein synchroner Ablauf läßt sich durch explizites Blockieren und Reaktivieren der Threads erreichen. Für diese bedingte Synchronisation benutzt man in Java die Methoden `wait` und `notify`/`notifyAll`. Dabei ist — ähnlich wie in CHILL — keine strikte Trennung zwischen Bedingung und zugehörigen Anweisungen möglich. Lebendigkeit wird weder auf der ersten noch auf der zweiten Ebene garantiert. Der Einsatz von `notifyAll`, der notwendig ist, wenn Threads bezüglich mehrerer Bedingungen blockiert werden, führt zu einem erhöhten Kommunikationsaufwand.

Echtzeitbezug ist mittels der Methode `System.currentTimeMillis` möglich; Timeouts lassen sich als Argument `arg` der Methode `wait(arg)` angeben.

## 6.5 Fazit

Insgesamt gesehen läßt sich feststellen, daß Ada am besten zur Lösung der betrachteten Aufgabenstellungen geeignet ist. Das hat zwei wesentliche Gründe:

1. Ada stellt als einzige der betrachteten Sprachen für jede der genannten Anforderungen an die Nebenläufigkeitskonzepte ein geeignetes Konstrukt zur Verfügung.

2. Die Ada-Konstruktionen sind teilweise besser als die entsprechenden Konstrukte in den anderen Sprachen. Das trifft vor allem für die protected objects zu.

CHILL, Erlang und Java haben generell den Nachteil, daß nicht für jede Anforderung ein entsprechendes Konstrukt zur Verfügung steht. Zwar lassen sich die Aufgaben auch damit lösen, jedoch sind die entstandenen Lösungen teilweise umständlich und unübersichtlich.

Abbildung 6.1 faßt die Ergebnisse noch einmal zusammen.

| Sprachkonstrukte | | Verwendete Sprache | | | |
|---|---|---|---|---|---|
| | | Ada | CHILL | Erlang | Java |
| **Synchrone Kommunikation** | | ja | nein | nein | nein |
| **Koordinierter Zugriff auf gemeinsame Variablen** | | ja | ja | nein | ja |
| | Exklusiver Schreibzugriff | ja | ja | - | ja |
| | Konkurrierender Lesezugriff | ja | nein | - | nein |
| **Bedingte Synchronisation** | | ja | ja | ja | ja |
| | Automatisches Blockieren und Reaktivieren | ja | nein | ja | nein |
| | Trennung von Bedingung und zugehörigen Anweisungen | ja | nein | ja | nein |
| | Lebendigkeit der beteiligten Akteure | ja [1)] | nein | ja | nein |
| **Echtzeitbezug** | | ja | ja | ja | ja |
| **Timeout** | | ja | ja | ja | ja |
| | bei direkter Kommunikation | ja | nein | ja [2)] | nein |
| | bei koordiniertem Zugriff auf gemeinsame Variable | ja | ja | - | ja |

[1)] nur auf Ebene 2
[2)] nur auf Seiten des Empfängers

**Abbildung 6.1** Die von den Sprachen zur Verfügung gestellten Sprachkonstrukte

# Anhang A

# Die Quelltexte in VISION O.N.E. CHILL

## A.1 Der nebenläufige Quicksort-Algorithmus

### A.1.1 SPU

```
<> SPU_CONTROL_HEADER
   BUDGET_INFOS :
   BEGIN
      HEAP_BUDGET           = 25,
      TIMER_PERIODIC_BUDGET = 10,
      TIMER_ONE_OFF_BUDGET  = 10
   END
<>
QUICKSPA : MODULE <> SPU <>
   ONSPMKOS : MODULE REMOTE <> REUSED <>  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   ONSPTKOS : MODULE REMOTE <> REUSED <>  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   QSORTOOS : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   QPROZOOS : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   QTESTOOS : MODULE REMOTE  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
```

```
   QSEMA00S : MODULE REMOTE  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
END QUICKSPA;
```

### A.1.2 Partitionieren und Sortieren

```
QSORT00S : MODULE
   SEIZE <> GLOBAL <>
      PMMSG01M_PM_MESSAGE;
   SEIZE
      SORTIERE_TEILFOLGE,
      WAIT;
   GRANT
      START_PUFFER,     /* Puffertyp fuer START-Anweisung*/
      FELD_TYP,
      NEBENLAEUFIGES_QUICKSORT;
   SYNMODE
      START_PUFFER = BUFFER(20) PMMSG01M_PM_MESSAGE,
      FELD_TYP = ARRAY (1:50) INT;
   DCL
      PUFFER START_PUFFER,
      PUFFER_ADRESSE REF START_PUFFER INIT := ADDR(PUFFER),
      PID INT INIT := 0;

   PARTITIONIEREN : PROC (FOLGE FELD_TYP LOC, VON INT, BIS INT,
                          PIVOT_INDEX INT LOC);
      DCL
         PIVOT_ELEMENT, UNTERER_INDEX, OBERER_INDEX, TEMP INT;
      PIVOT_ELEMENT := FOLGE (VON);
      UNTERER_INDEX := VON;
      OBERER_INDEX := BIS;
      TEMP := 0;
      BESTIMME_INDEX:
      DO FOR EVER;
         DO WHILE FOLGE (OBERER_INDEX) > PIVOT_ELEMENT;
            OBERER_INDEX := OBERER_INDEX -1;
         OD;
         DO WHILE FOLGE (UNTERER_INDEX) < PIVOT_ELEMENT;
            UNTERER_INDEX := UNTERER_INDEX + 1;
         OD;
         IF UNTERER_INDEX < OBERER_INDEX
            THEN
               TEMP := FOLGE (UNTERER_INDEX);
               FOLGE (UNTERER_INDEX) := FOLGE (OBERER_INDEX);
               FOLGE (OBERER_INDEX) := TEMP;
            ELSE
               EXIT BESTIMME_INDEX;
         FI;
```

```
         UNTERER_INDEX := UNTERER_INDEX + 1;
         OBERER_INDEX := OBERER_INDEX -1;
      OD BESTIMME_INDEX;
      PIVOT_INDEX := OBERER_INDEX;
   END PARTITIONIEREN;

   NEBENLAEUFIGES_QUICKSORT : PROC (FOLGE REF FELD_TYP, VON INT, BIS INT);
      DCL
         PIVOT_INDEX INT;
      PID := PID + 1;
      IF VON < BIS THEN
         PARTITIONIEREN (FOLGE->, VON, BIS, PIVOT_INDEX);
         START SORTIERE_TEILFOLGE (PUFFER_ADRESSE, PID, FOLGE,
                                   VON, PIVOT_INDEX);
         START SORTIERE_TEILFOLGE (PUFFER_ADRESSE, PID, FOLGE,
                                   PIVOT_INDEX+1, BIS);
         WAIT();
         WAIT();
      FI;
   END NEBENLAEUFIGES_QUICKSORT;
   ;
END QSORT00S;
```

### A.1.3 Prozeßdefinition

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = SORTIERE_TEILFOLGE,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN,
      PRIVILEGES    = ONSP21M_PRIV_PWS (:REGION_PRIV:),
      INCARNATION   = 500,
      STACKSIZE     = 32
   END
<>
QPROZ00S : MODULE
   SEIZE <> GLOBAL <>
      SCLMM50_PROCESS_CLASS,
      ONSP20M_PRIV_SET,
      ONSP21M_PRIV_PWS;
   SEIZE
      START_PUFFER,
      FELD_TYP,
      NEBENLAEUFIGES_QUICKSORT,
      SIGNAL;
   GRANT
      SORTIERE_TEILFOLGE;

   SORTIERE_TEILFOLGE : PROCESS (START_INFO REF START_PUFFER, PID INT,
                                 FOLGE REF FELD_TYP, VON INT, BIS INT);
      NEBENLAEUFIGES_QUICKSORT (FOLGE, VON, BIS);
```

```
      SIGNAL();
   END SORTIERE_TEILFOLGE;
   ;
END QPROZ00S;
```

### A.1.4 Semaphore zum Erzwingen des Wartens am Blockende

```
QSEMA00S : REGION
   GRANT
      WAIT,
      SIGNAL;
   DCL COUNTER INT INIT := 0;
   DCL COUNTER_ZERO EVENT;

   WAIT : PROC ();
      DO WHILE COUNTER = 0;
         DELAY COUNTER_ZERO;
      OD;
      COUNTER := COUNTER - 1;
   END WAIT;

   SIGNAL : PROC ();
      COUNTER := COUNTER + 1;
      CONTINUE COUNTER_ZERO;
   END SIGNAL;
;
END QSEMA00S;
```

### A.1.5 Testprogramm

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = HAUPTPROGRAMM,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN
   END
<>
QTEST00S : MODULE
   SEIZE <> GLOBAL <>
      SCLMM50_PROCESS_CLASS;
   SEIZE
      START_PUFFER,
      FELD_TYP,
      NEBENLAEUFIGES_QUICKSORT;
   GRANT
      HAUPTPROGRAMM;
   SYNMODE
      MEIN_PUFFER_MODE = BUFFER(1) INT;
   DCL
      FELD FELD_TYP INIT := (:1, 4, 8, 3, 56, 9, -34, 6, 0, 10,
                             3, 4, 7, -23, 89, 16, 9, -23, -12, 456,
```

```
                              9, -2, -3, -4, -5, -6, -7, -8, -9, -10:);
      ZEIGER_AUF_FELD REF FELD_TYP,
      PUFFER MEIN_PUFFER_MODE;

   HAUPTPROGRAMM : PROCESS (START_INFO REF START_PUFFER, PID INT);
      DCL ZEIT, ERGEBNIS INT;
      ZEIGER_AUF_FELD := ADDR (FELD);
      NEBENLAEUFIGES_QUICKSORT (ZEIGER_AUF_FELD, 1, 30);
   END HAUPTPROGRAMM;
;
END QTEST00S;
```

## A.2 Das Leser-Schreiber-Problem mit Priorität für Leser

### A.2.1 SPU

```
<> SPU_CONTROL_HEADER
   BUDGET_INFOS :
   BEGIN
      HEAP_BUDGET           = 25,
      TIMER_PERIODIC_BUDGET = 10,
      TIMER_ONE_OFF_BUDGET  = 10
   END
<>
LS1SPU0A : MODULE <> SPU <>
   ONSPMK0S : MODULE REMOTE <> REUSED <>  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   ONSPTK0S : MODULE REMOTE <> REUSED <>  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   LS1REG0S : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   LS1PRO0S : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   LESER01S : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   SCHREI1S : MODULE REMOTE <> PRODUCTION_PARAMETER
```

```
                BEGIN
                   SOURCE_TYPE = C,
                   TARGET_TYPE = F
                END <>;
   LS1TESTS : MODULE REMOTE <> PRODUCTION_PARAMETER
                BEGIN
                   SOURCE_TYPE = C,
                   TARGET_TYPE = F
                END <>;
END LS1SPU0A;
```

### A.2.2 REGION-Modul

```
LS1REG0S: REGION
   GRANT
      START_LESEN,
      ENDE_LESEN,
      START_SCHREIBEN,
      ENDE_SCHREIBEN;
   DCL
      ANZAHL_LESER INT INIT := 0,
      ANZAHL_WARTENDE_LESER INT INIT := 0,
      SCHREIBER_AKTIV BOOL INIT := FALSE,
      KEIN_SCHREIBER, KEIN_AKTEUR EVENT;

   START_LESEN: PROC();
      ANZAHL_WARTENDE_LESER := ANZAHL_WARTENDE_LESER + 1;
      DO WHILE SCHREIBER_AKTIV;
         DELAY KEIN_SCHREIBER;
      OD;
      ANZAHL_WARTENDE_LESER := ANZAHL_WARTENDE_LESER - 1;
      ANZAHL_LESER := ANZAHL_LESER + 1;
   END START_LESEN;

   ENDE_LESEN: PROC();
      ANZAHL_LESER := ANZAHL_LESER - 1;
      IF ANZAHL_LESER = 0 AND ANZAHL_WARTENDE_LESER = 0
         THEN CONTINUE KEIN_AKTEUR;
      FI;
   END ENDE_LESEN;

   START_SCHREIBEN: PROC();
      DO WHILE ANZAHL_LESER > 0 OR ANZAHL_WARTENDE_LESER > 0 OR SCHREIBER_AKTIV;
         DELAY KEIN_AKTEUR;
      OD;
      SCHREIBER_AKTIV := TRUE;
   END START_SCHREIBEN;

   ENDE_SCHREIBEN: PROC();
      DCL I INT;
      SCHREIBER_AKTIV := FALSE;
```

```
      IF ANZAHL_WARTENDE_LESER > 0
         THEN DO FOR I := 1 TO ANZAHL_WARTENDE_LESER;
                 CONTINUE KEIN_SCHREIBER;
              OD;
         ELSE CONTINUE KEIN_AKTEUR;
      FI;
   END ENDE_SCHREIBEN;
;
END LS1REG0S;
```

### A.2.3 Zugangsprotokoll

```
LS1PRO0S: MODULE
   SEIZE
      START_LESEN,
      ENDE_LESEN,
      START_SCHREIBEN,
      ENDE_SCHREIBEN;
   GRANT
      LS1_LESEN,
      LS1_SCHREIBEN;
   DCL
      RESSOURCE INT INIT := 0;

   LS1_LESEN: PROC (WERT INT LOC);
      START_LESEN();
      WERT := RESSOURCE;
      ENDE_LESEN();
   END LS1_LESEN;

   LS1_SCHREIBEN: PROC (WERT INT);
      START_SCHREIBEN();
      RESSOURCE := WERT;
      ENDE_SCHREIBEN();
   END LS1_SCHREIBEN;
;
END LS1PRO0S;
```

### A.2.4 Leser-Prozeß

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = LESER,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN,
      PRIVILEGES    = ONSP21M_PRIV_PWS (:REGION_PRIV:),
      START_BY_OS   = NO,
      INCARNATION   = 20
   END
<>
LESER01S : MODULE
```

```
   SEIZE <> GLOBAL <>
      PMMSG01M_PM_MESSAGE, SCLMM50_PROCESS_CLASS,
      ONSP20M_PRIV_SET, ONSP21M_PRIV_PWS;
   SEIZE
      LS1_LESEN,
      START_PUFFER;
   GRANT
      LESER;

   LESER: PROCESS (START_INFO REF START_PUFFER, PID INT);
      DCL WERT, I INT;
      DO FOR I:=1 TO 10;
         LS1_LESEN (WERT);
      OD;
   END LESER;
;
END LESER01S;
```

### A.2.5 Schreiber-Prozeß

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = SCHREIBER,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN,
      PRIVILEGES    = ONSP21M_PRIV_PWS (:REGION_PRIV:),
      START_BY_OS   = NO,
      INCARNATION   = 20
   END
<>
SCHREI1S : MODULE
   SEIZE <> GLOBAL <>
      PMMSG01M_PM_MESSAGE,
      SCLMM50_PROCESS_CLASS,
      ONSP20M_PRIV_SET,
      ONSP21M_PRIV_PWS;
   SEIZE
      LS1_SCHREIBEN,
      START_PUFFER;
   GRANT
      SCHREIBER;

   SCHREIBER: PROCESS (START_INFO REF START_PUFFER, PID INT);
      DCL I INT;
      DO FOR I:=1 TO 10;
         LS1_SCHREIBEN (I);
      OD;
   END SCHREIBER;
   ;
END SCHREI1S;
```

### A.2.6 Testprogramm

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = HAUPTPROGRAMM,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN
   END
<>
LS1TESTS : MODULE
   SEIZE <> GLOBAL <>
      PMMSG01M_PM_MESSAGE,
      SCLMM50_PROCESS_CLASS;
   SEIZE
      LESER,
      SCHREIBER;
   GRANT
      HAUPTPROGRAMM,
      START_PUFFER;
   SYNMODE
      START_PUFFER = BUFFER(20) PMMSG01M_PM_MESSAGE;
   DCL
      PUFFER START_PUFFER,
      PUFFER_ADR REF START_PUFFER;

   HAUPTPROGRAMM : PROCESS (START_INFO REF START_PUFFER, PID INT);
      PUFFER_ADR := ADDR (PUFFER);
      START SCHREIBER (PUFFER_ADR, 1);
      START LESER (PUFFER_ADR, 2);
   END HAUPTPROGRAMM;
   ;
END LS1TESTS;
```

## A.3 Das Leser-Schreiber-Problem mit Priorität für Schreiber

### A.3.1 SPU

```
<> SPU_CONTROL_HEADER
   BUDGET_INFOS :
   BEGIN
      HEAP_BUDGET           = 25,
      TIMER_PERIODIC_BUDGET = 10,
      TIMER_ONE_OFF_BUDGET  = 10
   END
<>
LS2SPU0A : MODULE <> SPU <>
   ONSPMK0S : MODULE REMOTE <> REUSED <>  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
```

```
   ONSPTK0S : MODULE REMOTE <> REUSED <>  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   LS2REG0S : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   LS2PRO0S : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   LESER02S : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   SCHREI2S : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   LS2TEST5 : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
END LS2SPU0A;
```

### A.3.2 REGION-Modul

```
LS2REG0S: REGION
GRANT
   START_LESEN,
   ENDE_LESEN,
   START_SCHREIBEN,
   ENDE_SCHREIBEN;
DCL
   ANZAHL_LESER INT INIT := 0,
   ANZAHL_WARTENDE_SCHREIBER INT INIT := 0,
   ANZAHL_WARTENDE_LESER INT INIT := 0,
   SCHREIBER_AKTIV BOOL INIT := FALSE,
   KEIN_SCHREIBER, KEIN_AKTEUR EVENT;

   START_LESEN: PROC();
      ANZAHL_WARTENDE_LESER := ANZAHL_WARTENDE_LESER + 1;
      DO WHILE SCHREIBER_AKTIV OR ANZAHL_WARTENDE_SCHREIBER > 0;
         DELAY KEIN_SCHREIBER;
```

```
      OD;
      ANZAHL_WARTENDE_LESER := ANZAHL_WARTENDE_LESER - 1;
      ANZAHL_LESER := ANZAHL_LESER + 1;
   END START_LESEN;

   ENDE_LESEN: PROC();
      ANZAHL_LESER := ANZAHL_LESER - 1;
      IF ANZAHL_LESER = 0 AND ANZAHL_WARTENDE_SCHREIBER > 0
         THEN CONTINUE KEIN_AKTEUR;
      FI;
   END ENDE_LESEN;

   START_SCHREIBEN: PROC();
      ANZAHL_WARTENDE_SCHREIBER := ANZAHL_WARTENDE_SCHREIBER + 1;
      DO WHILE ANZAHL_LESER > 0 OR SCHREIBER_AKTIV;
         DELAY KEIN_AKTEUR;
      OD;
      ANZAHL_WARTENDE_SCHREIBER := ANZAHL_WARTENDE_SCHREIBER - 1;
      SCHREIBER_AKTIV := TRUE;
   END START_SCHREIBEN;

   ENDE_SCHREIBEN: PROC();
      DCL I INT;
      SCHREIBER_AKTIV := FALSE;
      IF ANZAHL_WARTENDE_SCHREIBER = 0
         THEN DO FOR I := 1 TO ANZAHL_WARTENDE_LESER;
                 CONTINUE KEIN_SCHREIBER;
              OD;
         ELSE CONTINUE KEIN_AKTEUR;
      FI;
   END ENDE_SCHREIBEN;
;
END LS2REGOS;
```

### A.3.3 Zugangsprotokoll

```
LS2PROOS: MODULE
   SEIZE
      START_LESEN,
      ENDE_LESEN,
      START_SCHREIBEN,
      ENDE_SCHREIBEN;
   GRANT
      LS2_LESEN,
      LS2_SCHREIBEN;
   DCL
      RESSOURCE INT INIT := 0;

   LS2_LESEN: PROC (WERT INT LOC);
      START_LESEN();
      WERT := RESSOURCE;
```

```
      ENDE_LESEN();
   END LS2_LESEN;

   LS2_SCHREIBEN: PROC (WERT INT);
      START_SCHREIBEN();
      RESSOURCE := WERT;
      ENDE_SCHREIBEN();
   END LS2_SCHREIBEN;
   ;
END LS2PR00S;
```

### A.3.4 Leser-Prozeß

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = LESER,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN,
      PRIVILEGES    = ONSP21M_PRIV_PWS (:REGION_PRIV:),
      START_BY_OS   = NO,
      INCARNATION   = 20
   END
<>
LESER02S : MODULE
   SEIZE <> GLOBAL <>
      PMMSG01M_PM_MESSAGE,
      SCLMM50_PROCESS_CLASS,
      ONSP20M_PRIV_SET,
      ONSP21M_PRIV_PWS;
   SEIZE
      LS2_LESEN,
      START_PUFFER;
   GRANT
      LESER;

   LESER: PROCESS (START_INFO REF START_PUFFER, PID INT);
      DCL WERT, I INT;
      DO FOR I:=1 TO 10;
         LS2_LESEN (WERT);
      OD;
   END LESER;
;
END LESER02S;
```

### A.3.5 Schreiber-Prozeß

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = SCHREIBER,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN,
```

```
      PRIVILEGES    = ONSP21M_PRIV_PWS (:REGION_PRIV:),
      START_BY_OS   = NO,
      INCARNATION   = 20
   END
<>
SCHREI2S : MODULE
   SEIZE <> GLOBAL <>
      PMMSG01M_PM_MESSAGE,
      SCLMM50_PROCESS_CLASS,
      ONSP20M_PRIV_SET,
      ONSP21M_PRIV_PWS;
   SEIZE
      LS2_SCHREIBEN,
      START_PUFFER;
   GRANT
      SCHREIBER;

   SCHREIBER: PROCESS (START_INFO REF START_PUFFER, PID INT);
      DCL I INT;
      DO FOR I:=1 TO 10;
         LS2_SCHREIBEN (I);
      OD;
   END SCHREIBER;
;
END SCHREI2S;
```

### A.3.6 Testprogramm

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = HAUPTPROGRAMM,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN
   END
<>
LS2TESTS : MODULE
   SEIZE <> GLOBAL <>
      PMMSG01M_PM_MESSAGE,
      SCLMM50_PROCESS_CLASS;
   SEIZE
      LESER,
      SCHREIBER;
   GRANT
      HAUPTPROGRAMM,
      START_PUFFER;
   SYNMODE
      START_PUFFER = BUFFER(20) PMMSG01M_PM_MESSAGE;
   DCL
      PUFFER START_PUFFER,
      PUFFER_ADR REF START_PUFFER;
```

```
   HAUPTPROGRAMM : PROCESS (START_INFO REF START_PUFFER, PID INT);
      PUFFER_ADR := ADDR (PUFFER);
      START SCHREIBER (PUFFER_ADR, 1);
      START LESER (PUFFER_ADR, 2);
   END HAUPTPROGRAMM;
;
END LS2TESTS;
```

## A.4 Der Bakery-Algorithmus

### A.4.1 SPU

```
<> SPU_CONTROL_HEADER
   BUDGET_INFOS :
   BEGIN
      HEAP_BUDGET           = 25,
      TIMER_PERIODIC_BUDGET = 10,
      TIMER_ONE_OFF_BUDGET  = 10
   END
<>
BAKERY0A : MODULE <> SPU <>
   ONSPMK0S : MODULE REMOTE <> REUSED <>  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   ONSPTK0S : MODULE REMOTE <> REUSED <>  <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   BAK_REGS : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   BAK_PR0S : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
   BAK_TSTS : MODULE REMOTE <> PRODUCTION_PARAMETER
              BEGIN
                 SOURCE_TYPE = C,
                 TARGET_TYPE = F
              END <>;
END BAKERY0A;
```

### A.4.2 REGION-Modul

```
BAK_REGS : REGION
   GRANT
      BEGINN_ZIEHE_NUMMER,
      ENDE_ZIEHE_NUMMER,
      ZIEHE_NUMMER,
      SETZE_NUMMER_ZURUECK,
      ZIEHT_NUMMER,
      HAT_VORRANG_VOR,
      PROZ_NUMMER,
      ANZAHL_PROZESSE;
   SYN
      ANZAHL_PROZESSE = 10;
   SYNMODE
      PROZ_NUMMER = RANGE(1:ANZAHL_PROZESSE);
   DCL
      NUMMER, WAEHLE_NUMMER ARRAY(PROZ_NUMMER) INT INIT
                                        := (:(1:ANZAHL_PROZESSE):0:),
      MAX_NUMMER INT INIT := 0;

   BEGINN_ZIEHE_NUMMER: PROC (I PROZ_NUMMER);
      WAEHLE_NUMMER(I) := 1;
   END BEGINN_ZIEHE_NUMMER;

   ENDE_ZIEHE_NUMMER: PROC (I PROZ_NUMMER);
      WAEHLE_NUMMER(I) := 0;
   END ENDE_ZIEHE_NUMMER;

   ZIEHE_NUMMER: PROC (I PROZ_NUMMER);
      NUMMER(I) := 1 + MAX_NUMMER;
   END ZIEHE_NUMMER;

   SETZE_NUMMER_ZURUECK: PROC (I PROZ_NUMMER);
      DCL J INT;
      IF NUMMER(I) = MAX_NUMMER THEN
         /* MAX_NUMMER MUSS NEU BESTIMMT WERDEN */
         MAX_NUMMER := 0;
         DO FOR J := 1 TO ANZAHL_PROZESSE;
            IF NUMMER(J) > MAX_NUMMER AND J /= I THEN
               MAX_NUMMER := NUMMER(J);
            FI;
         OD;
      FI;
      NUMMER(I) := 0;
   END SETZE_NUMMER_ZURUECK;

   ZIEHT_NUMMER: PROC (I PROZ_NUMMER) RETURNS (BOOL);
      RESULT WAEHLE_NUMMER(I) = 1;
   END ZIEHT_NUMMER;
```

```
   HAT_VORRANG_VOR: PROC (I, J PROZ_NUMMER) RETURNS (BOOL);
      RESULT NUMMER(J) = 0 OR NUMMER(I) < NUMMER(J) OR
            (NUMMER(I) = NUMMER(J) AND I < J);
   END HAT_VORRANG_VOR;
   ;
END BAK_REGS;
```

### A.4.3 Prozeßdefinition

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = BAKERY_PROZESS,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN,
      PRIVILEGES    = ONSP21M_PRIV_PWS (:REGION_PRIV:),
      INCARNATION   = 50,
      STACKSIZE     = 64
   END
<>
BAK_PROS : MODULE
   SEIZE <> GLOBAL <>
      SCLMM50_PROCESS_CLASS,
      ONSP20M_PRIV_SET,
      ONSP21M_PRIV_PWS;
   SEIZE
      BEGINN_ZIEHE_NUMMER,
      ENDE_ZIEHE_NUMMER,
      ZIEHE_NUMMER,
      SETZE_NUMMER_ZURUECK,
      ZIEHT_NUMMER,
      HAT_VORRANG_VOR,
      START_PUFFER,
      PROZ_NUMMER,
      ANZAHL_PROZESSE;
   GRANT
      BAKERY_PROZESS;

   BAKERY_PROZESS: PROCESS (START_INFO REF START_PUFFER,
                            PID INT, I PROZ_NUMMER);
      DCL J INT;

      NICHT_KRITISCHER_ABSCHNITT: PROC (I PROZ_NUMMER);
         /* ANWEISUNGEN DES NICHT KRITISCHEN ABSCHNITTS DES PROZESSES I */
         ;
      END NICHT_KRITISCHER_ABSCHNITT;

      KRITISCHER_ABSCHNITT: PROC (I PROZ_NUMMER);
         /* ANWEISUNGEN DES KRITISCHEN ABSCHNITTS DES PROZESSES I*/
         ;
      END KRITISCHER_ABSCHNITT;
```

```
      DO FOR EVER;
         NICHT_KRITISCHER_ABSCHNITT(I);
         BEGINN_ZIEHE_NUMMER(I);
         ZIEHE_NUMMER(I);
         ENDE_ZIEHE_NUMMER(I);
         DO FOR J := 1 TO ANZAHL_PROZESSE;
            IF J /= I THEN
               L1: DO FOR EVER;
                  IF NOT ZIEHT_NUMMER(J) THEN
                     EXIT L1;
                  FI;
               OD L1;
               L2: DO FOR EVER;
                  IF HAT_VORRANG_VOR(I, J) THEN
                     EXIT L2;
                  FI;
               OD L2;
            FI;
         OD;
         KRITISCHER_ABSCHNITT(I);
         SETZE_NUMMER_ZURUECK(I);
      OD;
   END BAKERY_PROZESS;
;
END BAK_PROS;
```

### A.4.4 Start des Programms

```
<> MODULE_DESCRIPTION_HEADER
   PROCESS_TYPE_INFOS:
   BEGIN
      PROCESS_NAME  = START_BAKERY,
      PROCESS_CLASS = SC_PC_MAINTENANCE_OWN
   END
<>
BAK_TSTS : MODULE
   SEIZE <> GLOBAL <>
      SCLMM50_PROCESS_CLASS,
      PMMSG01M_PM_MESSAGE;
   SEIZE
      ANZAHL_PROZESSE,
      BAKERY_PROZESS;
   GRANT
      START_BAKERY,
      START_PUFFER;
   SYNMODE
      START_PUFFER = BUFFER(10) PMMSG01M_PM_MESSAGE;
   DCL
      PUFFER START_PUFFER,
      PUFFER_ADRESSE REF START_PUFFER INIT := ADDR(PUFFER);
```

```
   START_BAKERY : PROCESS ();
      DCL I INT;
      DO FOR I := 1 TO ANZAHL_PROZESSE;
         START BAKERY_PROZESS (PUFFER_ADRESSE, I, I);
      OD;
   END START_BAKERY;
;
END BAK_TSTS;
```

# Literaturverzeichnis


[AVWW 96] J. Armstrong, R. Virding, C. Wikström, M. Williams: *Concurrent Programming in ERLANG*, Prentice Hall, 2. Auflage, 1996

[BE 98] P. Brömel, F. Ecke: *Description and Comparison of the Concurrency Concepts of Ada, CHILL, and Java*, Studienarbeit, Friedrich-Schiller-Universität Jena, Institut für Informatik, Lehrstuhl für Programmiersprachen und Compiler, 1998

[Ben 90] M. Ben-Ari: *Principles of Concurrent and Distributed Programming*, Prentice Hall, 1990

[BuWe 98] A. Burns, A. Wellings: *Concurrency in Ada*, Cambridge University Press, 2. Auflage, 1998

[Cor 94] T. Cormen, C. Leiserson, R. Rivest: *Introduction to Algorithms*, MIT Press, 14. Auflage, 1994

[Geh 91] N. Gehani: *Ada: Concurrent Programming*, Silicon Press, 2. Auflage, 1991

[GJS 96] J. Gosling, B. Joy, G. Steele: *The Java$^{TM}$ Language Specification*, Addison-Wesley, 1996

[Gri 93] Ralph P. Grimaldi: *Discrete and Combinatorial Mathematics: An Applied Introduction*, Addison-Wesley, 1993

[Int 95] Intermetrics, Inc.: *Ada Reference Manual - Language and Standard Libraries*, ISO/IEC 8652:1995

[Lea 97] D. Lea: *Concurrent Programming in Java$^{TM}$ – Design Principles and Patterns*, Addison-Wesley, 1997

[Z.200 96] Recommendation Z.200 (11/96) – CCITT High Level Language (CHILL)


### Literatur zu VISION O.N.E. CHILL


[CHILL] Description P30308-A2737-B000-02-7618
Support Software
CHILL Language and Compiler

[Chapter 1] Guiding Instruction P30308-A2821-A100-01-7635
CHILL OS Kernel User Manual
Chapter 1: Introduction and Overview

[Chapter 3] Guiding Instruction P30308-A2821-A300-06-7635
CHILL OS Kernel User Manual
Chapter 3: Process Management and Scheduling

[Chapter 4] Guiding Instruction P30308-A2821-A400-06-7635
User Manual MP Operating System
Chapter 4: Service Management

[Chapter 5] Guiding Instruction P30308-A2821-A500-11-7635
User Manual VOCOS Operating System on MP
Chapter 5: Interprocess Communication

[Chapter 6] Guiding Instruction P30308-A2821-A600-07-7635
User Manual MP Operating System
Chapter 6: Time Management

[Chapter 8] Guiding Instruction P30308-A2821-A800-04-7635
CHILL OS Kernel User Manual
Chapter 8: Concurrency and Transaction Control

[Chapter 11] Guiding Instruction P30308-A2821-B200-04-7635
CHILL User Manual
Chapter 11: VCPUs, Process Classes and Process Priorities

[Kurs 6142] A30181-X6142D-X100-02-0035
EWSD CHILL Support Software
Kurs 6142, Version 02, 03/1998

[Kurs 6712] A30181-X1208D-X200-07-35
EWSD Programmierpsrache CHILL
Kurs 6712, Version 02, 03/1995

[SIT Bsp] Beschreibung P30308-A7887-A000-01-0018
SITOP3 Überblick und Beispiel

[SIT WB] Operating Instructions P30308-A7887-A000-05-7619
SITOP3 Workbench
Simulation Test Tools (SWS/4, SAP2/3)

# Erklärung

Ich erkläre hiermit, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, 12.04.1999 Peter Brömel