

# **Parallelprogrammierung**

Vorlesung an der Hochschule Bremerhaven

Prof. Dr. Thomas Umland

Wintersemester 2024/2025

# Allgemeines

1. Vorlesungsmaterialien, insbesondere Folien und Aufgabenblätter stehen im pdf-Format auf dem ELLI-Server der Hochschule
2. Kontakt:
  - E-Mail: [Thomas.Umland@hs-bremerhaven.de](mailto:Thomas.Umland@hs-bremerhaven.de)
  - Raum Z2030
  - Telefon: (0471) 4823-406
  - Sprechzeiten: nach Vereinbarung

# Organisatorisches

- **Umfang:** 6 CP (4 SWS: 2 Std. Vorlesung + 2 Std. Übung)
- **Vorlesung:** Donnerstags, 08.00 – 09.30 Uhr, **M 200**  
**Übungen:** Donnerstags, 09.45 – 11.15 Uhr, **Z 2320**
- Es wird wöchentlich **Übungsaufgaben** geben
- **Pflicht:** Jeder stellt mind. drei Lösungen zu den Übungsaufgaben vor (von mind. drei versch. Aufgabenblättern)!
- Zur Erlangung eines **Leistungsnachweises** sind im Verlauf des Semester größere Aufgaben („Assignments“) zu lösen und termingerecht abzugeben

# Erwartungen

Was erwarten Sie von der Vorlesung?

Welche inhaltlichen Wünsche haben Sie?

- . . .
- . . .
- . . .

# Gliederung

## A Literaturhinweise

### 1 Einführung

#### 1.1 Begriffe und Motivation der Parallelverarbeitung

#### 1.2 Parallelrechnertypen

### 2 Erste Programme

#### 2.1 Starten von Prozessen in Java

#### 2.2 Gute Programmbeispiele?

### 3 Modellierung von Prozessen

- 3.1 Sequentielle Prozesse
- 3.2 Modellierung von Nebenläufigkeit
- 4 Nebenläufigkeit in Java
  - 4.1 Gegenseitiger Ausschluss in Java
  - 4.2 Monitore
  - 4.3 Semaphore
- 5 Eigenschaften paralleler Programme
  - 5.1 Verklemmungen
  - 5.2 Sicherheit und Lebendigkeit

## 6 Weitere Themen

6.1 Probleme mit gemeinsam benutzten Daten

6.2 „Veröffentlichung“ und „Entkommen“ von Objekten

6.3 Collections und Nebenläufigkeit

6.4 Ausgewähltes aus `java.util.concurrent`

6.5 Beispiel zur Thread-Safety

## 7 Thread-Prioritäten und Nebenläufigkeit

## 8 Synchronisation durch Austausch von Nachrichten

8.1 Nachrichtenaustausch mit Rendezvous-Konzept

## 8.2 Synchroner Nachrichtenaustausch in FSP

## 8.3 Umsetzung in Java – JCSP



# A Literaturhinweise

- [Car02] J. R. Carter, Java questions, *IEEE Computer*, (10) **35** (2002), 9.
- [Dij68] E. W. Dijkstra, Cooperating sequential processes. In: F. Genys (Ed.), *Programming Languages*, Academic Press, New York (1968) 43–112.
- [Goe06] B. Goetz, *Java Concurrency in Practice*, Addison-Wesley, Upper Saddle River, New Jersey (2006).
- [Han99] P. B. Hansen, Java's insecure parallelism, *ACM SIGPLAN Notices*, (4) **23** (1999), 38–45.

- [Hoa74] C. A. R. Hoare, Monitors: An operating system structuring concept, *Communications of the ACM*, (10) **17** (1974), 549–557.
- [Hoa78] C. A. R. Hoare, Communicating sequential processes, *Communications of the ACM*, (8) **21** (1978), 666–677.
- [Jon24] M. B. Jones, *What really happened on Mars?*  
<https://retis.sssup.it/~giorgio/mars/jones.html>  
(07.10.2024).
- [Lea00] D. Lea, *Concurrent Programming in Java – Design Principles and Patterns*, The Java Series, Addison-Wesley, Reading, Massachusetts (2000).

- [Lee06] E. A. Lee, The problem with threads, *IEEE Computer*, (5) **39** (2006), 33–42.
- [MK06] J. Magee, J. Kramer, *Concurrency – State Models and Java Programs*, John Wiley & Sons, West Sussex, 2. Auflage (2006).
- [Ora24a] Oracle, *Java Platform, Standard Edition & Java Development Kit, Version 17 API Specification*. <https://docs.oracle.com/en/java/javase/17/docs/api> (07.10.2024).
- [Ora24b] Oracle Corporation, *JEP 444: Virtual Threads*. <https://openjdk.org/jeps/444> (07.10.2024).

- [PM88] D. Pountain, D. May, *A Tutorial Introduction to Occam 2*, BSP Professional Books, London (1988).
- [Ros98] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall International, London (1998).
- [RR07] T. Rauber, G. Rünger, *Parallele Programmierung*, Springer-Verlag, Berlin, Heidelberg, 2. Auflage (2007).
- [San04] B. Sandén, Coping with java threads, *IEEE Computer*, (4) **37** (2004), 20–27.
- [Spr24] Spring by Pivotal Software, Inc., *Notes on Reactive Programming Part I: The Reactive Landscape*. <https://spring.io/blog/2016/06/07/notes->

on-reactive-programming-part-i-the-reactive-landscape (07.10.2024).

- [SRL90] L. Sha, R. Rajkumar, J. P. Lehoczky, Priority inheritance protocols: An approach to real-time synchronisation, *IEEE Transactions on Computers*, (9) **39** (1990), 1175–1185.
- [Ste88] R. Steinmetz, *Occam 2*, Hüthig, Heidelberg (1988).
- [TOP24] TOP500.org, *Top 500 Supercomputing Sites*. <https://www.top500.org/lists> (07.10.2024).
- [Uml94] T. Umland, Parallel sorting revisited, *Parallel Computing*, **20** (1994), 115–124.

- [Uml98] T. Umland, Parallel graph coloring using Java. In: P. H. Welch, A. W. P. Bakkers (Eds.), *Architectures, Languages and Patterns for Parallel and Distributed Applications*, Concurrent Systems Engineering Series, IOS Press, Amsterdam (1998) 211–218.
- [Uni24a] University of California, *SETI@home*. <https://setiathome.berkeley.edu> (07.10.2024).
- [Uni24b] University of Kent at Canterbury, *JCSP 1.1 Release Candidate 4 (rc4)*. <https://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4.jar> (07.10.2024).
- [UV92] T. Umland, R. Vollmar, *Transputerpraktikum*, B. G. Teubner, Stuttgart (1992).

- [WB98] P. H. Welch, A. W. P. Bakkers (Eds.), *Architectures, Languages and Patterns for Parallel and Distributed Applications*, Concurrent Systems Engineering Series, IOS Press, Amsterdam (1998).
- [WB24] P. Welch, N. Brown, *Communicating Sequential Processes for Java (JCSP)*. <https://www.cs.kent.ac.uk/projects/ofa/jcsp> (07.10.2024).
- [Wel98] P. H. Welch, Java threads in the light of occam/CSP. In: P. H. Welch, A. W. P. Bakkers (Eds.), *Architectures, Languages and Patterns for Parallel and Distributed Applications*, Concurrent Systems Engineering Series, IOS Press, Amsterdam (1998) 259–284.

[Wik24] Wikimedia Foundation, *JCSP*. <https://en.wikipedia.org/wiki/JCSP> (07.10.2024).



# 1 Einführung

- In natürlichen Systemen funktioniert **Parallelverarbeitung selbstverständlich!**
- Bei komplexen Programmsystemen treten dagegen immer wieder **Probleme** mit der „parallelen Denkweise“ auf:
  - **Beispiele:**
    - \* Versagen medizinischer Bestrahlungsgeräte (Therac-25) in den Jahren 1985-87 (vgl. [?])
    - \* Software des Mars Pathfinder 1997 [Jon24] (siehe Abschnitt ??)
    - \* Viele Probleme bleiben trotz Tests unerkannt!

- **Ursachen für die Probleme:** Unterschätzen der Komplexität, ungeeignete Programmiermodelle, ...
- **Warum?**
  - Die meisten Programmiersprachen arbeiten sequentiell
  - in der Ausbildung wird meistens nur sequentiell programmiert!
- **Schade, denn ...**
  - praktisch jeder Rechner enthält heute mehrere Prozessorkerne und/oder mehrere Prozessoren!
  - Parallelverarbeitung ist der „Normalzustand“ auf jedem Computer!

- **In dieser Vorlesung:**
  - Parallele Abläufe mithilfe eines einfachen Werkzeugs modellieren und dadurch besser verstehen
  - Modelle dann in *korrekte* Java-Programme umsetzen
  - Einfache parallele Algorithmen kennen und implementieren lernen
  - Bewusstsein für Fehlerquellen in parallelen Programmen schärfen und Lösungsmöglichkeiten aufzeigen

# 1.1 Begriffe und Motivation der Parallelverarbeitung

- Was heißt „sequentiell“, „parallel“, „nebenläufig“?
- Was ist ein Prozess?
- Wofür ist Parallelprogrammierung nützlich?

**Definition:** Die Ausführung von Anweisungen eines Programmes heißt *sequentiell*, wenn deren Abarbeitung nacheinander in deterministischer Reihenfolge geschieht.

**Definition:** Die Ausführung von Anweisungen eines Programmes heißt *parallel*, wenn es einen Zeitpunkt gibt, zu dem die Anweisungen gleichzeitig auf jeweils einem Prozessor ausgeführt werden.

**Frage:** Kann ein für die parallele Ausführung geschriebenes Programm auch auf einem einzelnen Prozessor ausgeführt werden?

**Definition:** Zwei Anweisungen heißen *nebenläufig* (engl. *concurrent*), wenn sie entweder gleichzeitig von zwei Prozessoren oder in beliebiger Reihenfolge sequentiell auf einem Prozessor ausgeführt werden können.

- „nebenläufig“ ist damit ein allgemeinerer Begriff als „parallel“,
- beide Begriffe werden auch synonym gebraucht (Bedeutung ist aus dem Zusammenhang erkennbar).

**Definition:** Unter einem *Prozess* wird eine „Folge atomarer – d. h. nicht teilbarer – Aktionen“ verstanden.

- Der Zustand jedes Prozesses wird zu jedem Zeitpunkt durch die Werte seiner Variablen beschrieben
- durch die Abarbeitung der einzelnen Aktionen entsteht eine Zustandsfolge

**Definition:** Ein *Prozessor* ist eine „Funktionseinheit“, die in der Lage ist, Prozesse auszuführen.

# Warum Parallelverarbeitung?

Es lassen sich vier **Hauptgründe** für den Einsatz von Parallelverarbeitung identifizieren:

## 1. **Beschleunigung einer Problemlösung**

- „Zeit pro Aufgabe“ reduzieren
- Voraussetzung: Die Aufgabe lässt sich gleichmäßig aufteilen
- Der Zeitgewinn durch Parallelverarbeitung ist unabhängig von der Technologie und den physikalischen Grenzen der Einzelprozessoren
- **Beispiel:** schnellere Wettervorhersage.



## 2. Erhöhung des Durchsatzes

- „Aufgaben pro Zeiteinheit“ erhöhen
- bei konstanter Zeit lassen sich größere Probleme lösen
- **Beispiel:** Wettervorhersage mit genaueren Modellen rechnen

### 3. **Natürlichere Beschreibung von Lösungen**

- Viele Probleme besitzen inhärente Parallelität
- sequentieller Ansatz verkompliziert oftmals die Lösung
- paralleler Ansatz ermöglicht evtl. „natürlichere“ Beschreibung/Dekomposition der Lösung
- **Beispiele:** graphische Benutzeroberflächen, natürliche/biologische Modelle (genetische Algorithmen, neuronale Netze, etc.)

## 4. Fehlertoleranz

- durch gleichzeitige Lösung einer Aufgabe auf verschiedener Hardware und/oder unterschiedlichen Algorithmen und anschließend „Voting“ können Fehler erkannt werden
- **Beispiel:** Reaktorsteuerung, Raumfahrt (z. B. 4-fache Redundanz in Space Shuttle) und andere zuverlässige Systeme; Ziel: zumeist Erkennen sogen. „Bit Flips“ aufgrund kosmischer Strahlung

# Exkurs „Parallele Algorithmen“

Zur Bewertung paralleler Algorithmen werden oft die beiden Größen **Beschleunigung** und **Effizienz** herangezogen:

**Beschleunigung (Speedup):** Seien  $n$  die Problemgröße,  $p$  die Anzahl verwendeter Prozessoren und  $t_n(p)$  die Ausführungszeit bei Einsatz von  $p$  Prozessoren, so bezeichnet

$$s_n(p) := \frac{t_n(1)}{t_n(p)}$$

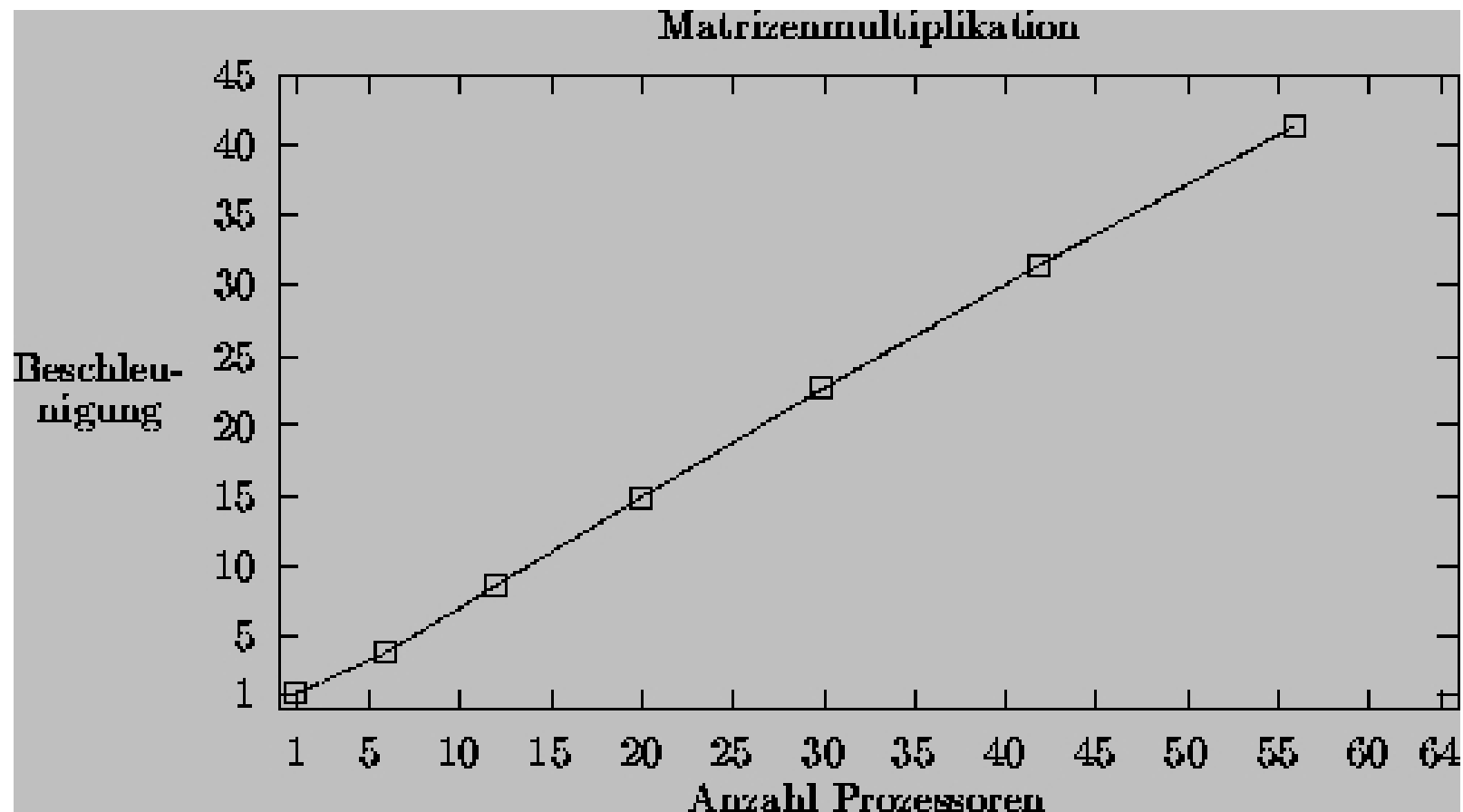
die mit dem parallelen Algorithmus erreichbare *Beschleunigung*.

**Effizienz:** Anhand der Beschleunigung  $s_n(p)$  und der dafür notwendigen Anzahl Prozessoren  $p$  wird die *Effizienz* definiert als

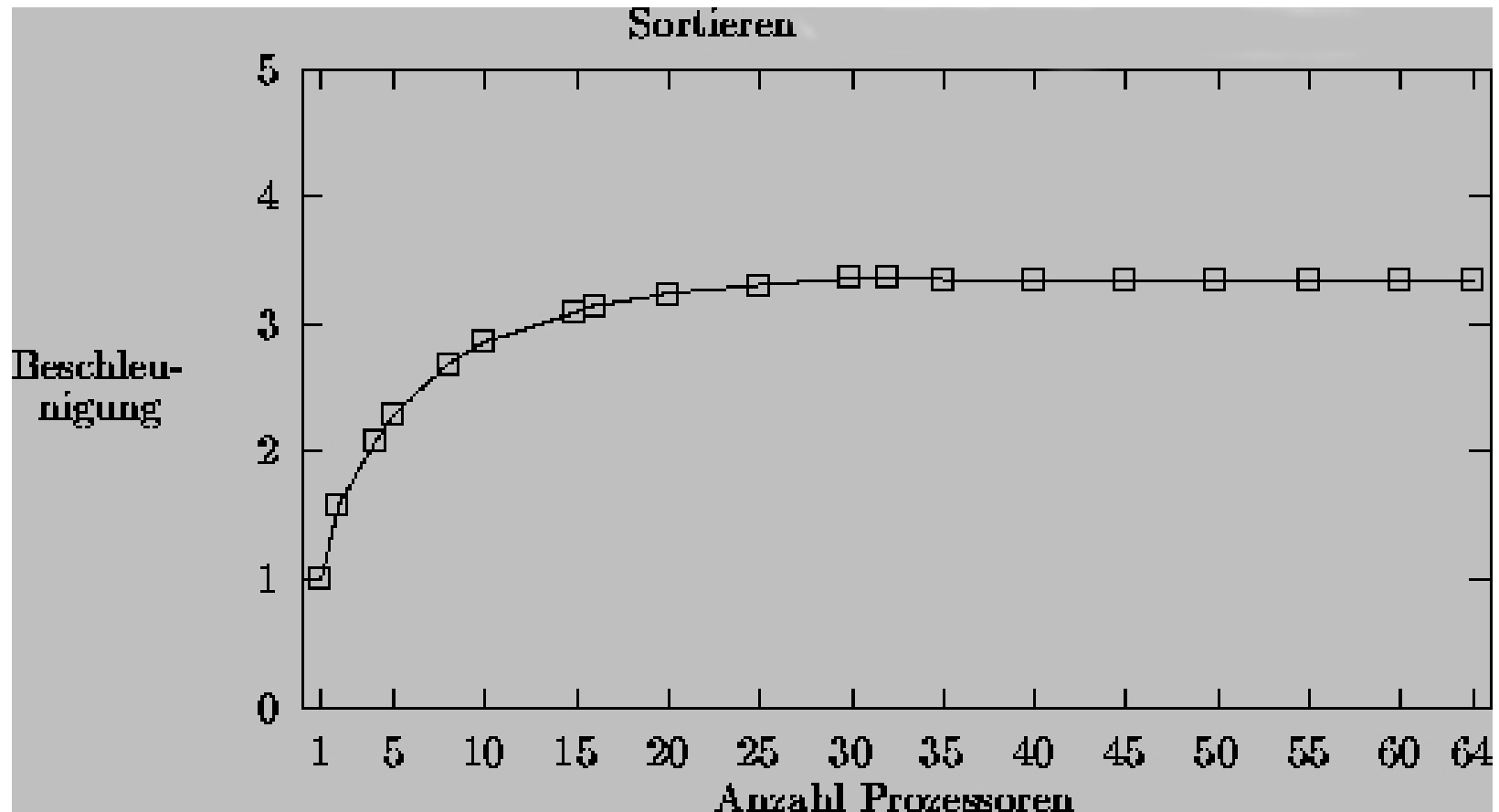
$$e_n(p) := \frac{s_n(p)}{p}.$$

# Beschleunigungsbeispiele paralleler Algorithmen

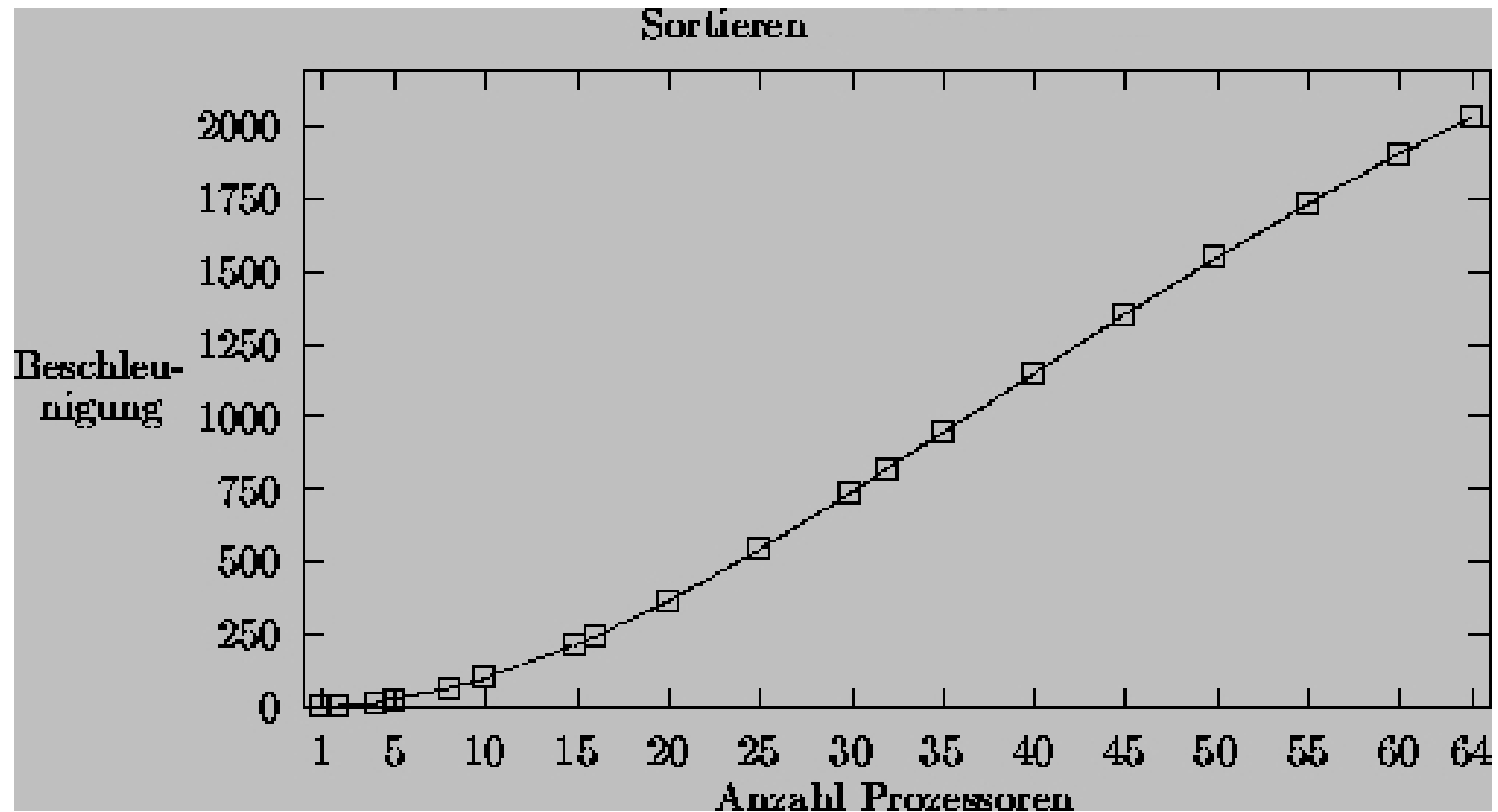
- Gute Beschleunigung:



- Realistische Beschleunigung:



- Superlineare Beschleunigung:





# Ein paralleler Sortieralgorithmus (1)

## Odd Even Sort:

Gegeben seien  $n$  zu sortierende Zahlen und  $n$  Prozessoren  $P_1, \dots, P_n$ . Jede Zahl wird auf genau einem der Prozessoren gespeichert.

Zum Sortieren werden die beiden folgenden Schritte abwechselnd insgesamt  $n$ -mal durchgeführt:

## Ein paralleler Sortieralgorithmus (2)

1. In einem ungeraden Schritt werden alle Prozessoren  $P_i$  ( $1 \leq i < n$ ) mit *ungeradem* Index aktiv. Sie vergleichen ihre gespeicherte Zahl  $z_i$  mit der des Prozessors  $P_{i+1}$ . Falls  $z_i > z_{i+1}$  ist, vertauschen die Prozessoren ihre Elemente, andernfalls bleiben die Elemente an ihren Plätzen.
2. Im folgenden geraden Schritt werden alle Prozessoren  $P_j$  ( $1 < j < n$ ) mit *geradem* Index aktiv, und sie vergleichen ihr eigenes Element  $z_j$  mit der des Prozessors  $P_{j+1}$ . Falls  $z_j > z_{j+1}$  ist, tauschen die Prozessoren wieder ihre Elemente aus, im anderen Fall ändert sich nichts.

# Ein paralleler Sortieralgorithmus (3)

- Wie arbeitet der Odd Even-Algorithmus die Zahlenfolge  $\{8, 3, 4, 6, 2, 1, 5, 7\}$  ab?
- Welche Struktur müssen die Eingabedaten aufweisen, damit alle  $n$  Schritte des Algorithmus notwendig sind?
- Wie lässt sich der Algorithmus modifizieren, damit die Anzahl der zu sortierenden Zahlen unabhängig von der Prozessoranzahl wird?

**Bemerkung:** Es gibt „bessere“ allerdings auch etwas kompliziertere parallele Sortieralgorithmen als Odd Even Sort (vgl. z. B. [[Uml94](#)]).

# Zusammenfassung (1)

- Ein **Prozess** in unserem Sinne ist „eine Folge von atomaren Anweisungen“
- Die Ausführung von Programmen lässt sich anhand der Begriffe „**sequentiell**“, „**parallel**“ und „**nebenläufig**“ näher beschreiben
- **Ziele des Einsatzes von Parallelverarbeitung** sind in der Regel die Beschleunigung der Ausführung, eine Erhöhung des Durchsatzes, das Erreichen von Fehlertoleranz oder die einfache Dekomposition von Problemlösungen
- **Odd Even Sort** als Beispiel für ein einfaches paralleles Sortierverfahren

## 1.2 Parallelrechnertypen

- Nach welchen Kriterien lassen sich Rechner unterscheiden?
- Welche Arten von Parallelrechnern lassen sich unterscheiden?
- Was sind deren Eigenschaften?
- Welche Art von Parallelrechnern lässt sich direkt für Java nutzen?

# Flynns Taxonomie

- Grobe Einteilung von Parallelrechnern nach der Art der „Verarbeitungsströme“
- unterscheidet einfachen (single) und mehrfachen (multiple) Befehls- bzw. Datenstrom

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

# SISD

Single Instruction Stream, Single Data Stream

- Zu jedem Zeitpunkt wird *eine* Instruktion auf *einem* Datum ausgeführt
- entspricht herkömmlichen sequentiellen Rechnern nach der von-Neumann-Architektur (z. B. PCs mit einem Kern)
- wenn Ein-, Ausgabe- oder Co-Prozessoren verwendet werden, liegt streng genommen kein „reiner“ SISD-Rechner mehr vor



# SIMD

Single Instruction Stream, Multiple Data Stream

- *Mehrere* Prozessoren arbeiten gleichzeitig auf verschiedenen Daten *denselben* Befehl ab
- werden auch als *synchrone* Multiprozessoren bezeichnet (z. B. Connection-Machine (CM-2), MasPar (MP) oder moderne Grafikkarten)
- bestehen in der Regel aus relativ einfachen sogenannten *Processing Elements (PEs)*
- Einfachheit der PEs wird durch eine sehr große Anzahl ausgeglichen (CM-2 mit 65.536, MP mit 16.384 PEs oder z. B. Nvidia RTX-4090 mit 16.384 CUDA Cores)

# MIMD

Multiple Instruction Stream, Multiple Data Stream

- *Mehrere* Prozessoren verarbeiten *unabhängig voneinander* jeweils ihre *eigenen* Daten
- in der Regel besitzen MIMD-Rechner identische Prozessoren oder Prozessorkerne
- je nach Bauart können die Prozessoren getrennte oder einen gemeinsamen Speicher besitzen
- **Beispiele:** Multiprozessor-PCs, Multiprozessor-Server/-Supercomputer<sup>1</sup>

---

<sup>1</sup> Für technische Details aktueller Supercomputer siehe [[TOP24](#)]

# MISD

Multiple Instruction Stream, Single Data Stream

- *Mehrere* Befehle werden gleichzeitig auf *einem* Datum ausgeführt
- z. B. fehlertolerante Systeme, in denen zu Kontrollzwecken jede Operation mehrfach auf unabhängigen Prozessoren ausgeführt wird
- wird in der Literatur oft nicht betrachtet

# Weitere Unterscheidung von MIMD-Rechnern

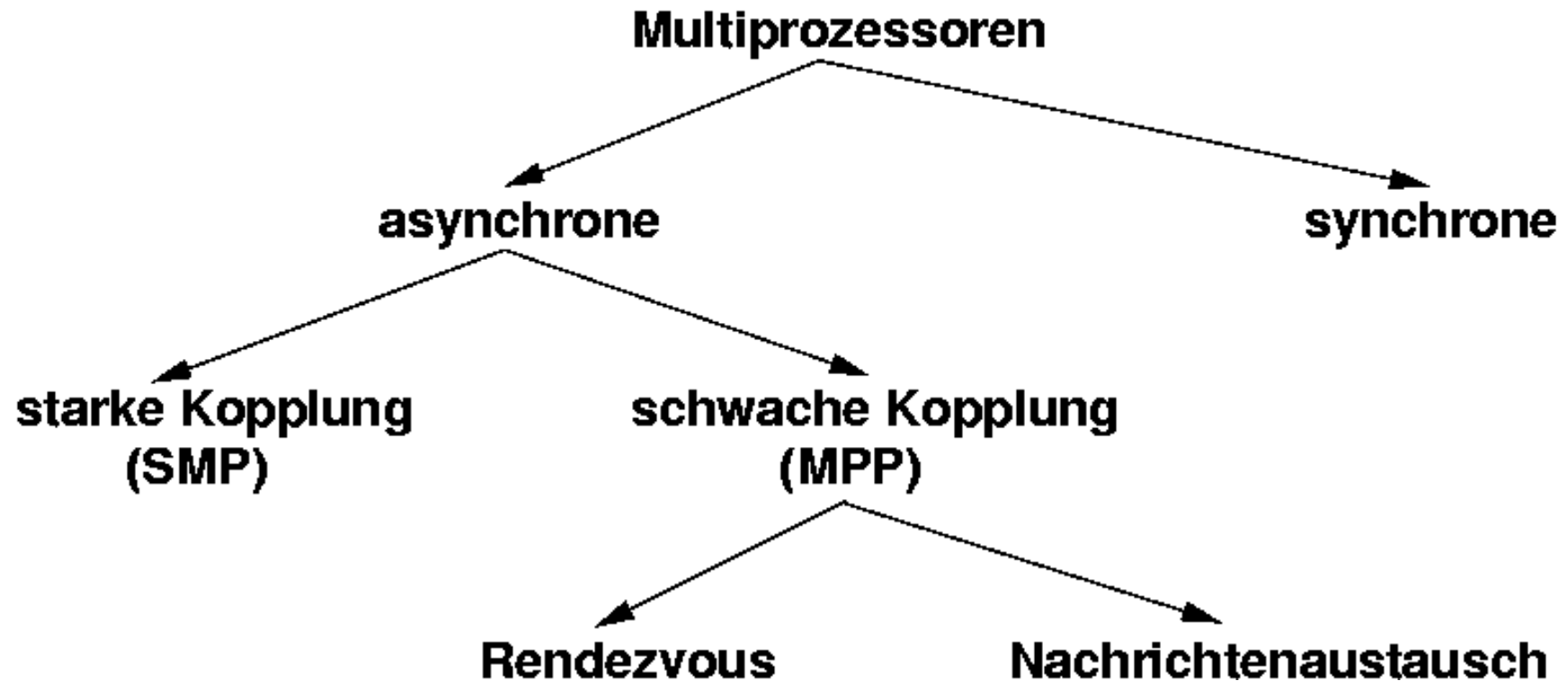
**SMP (symmetric multiprocessing)** SMP-Rechner bestehen aus mehreren, gleichartigen CPUs/Kernen, die alle auf denselben Speicher zugreifen:

- einfaches Programmiermodell
- bei „vielen“ Prozessoren ersetzt aufwändiger Crossbar den einfachen Systembus

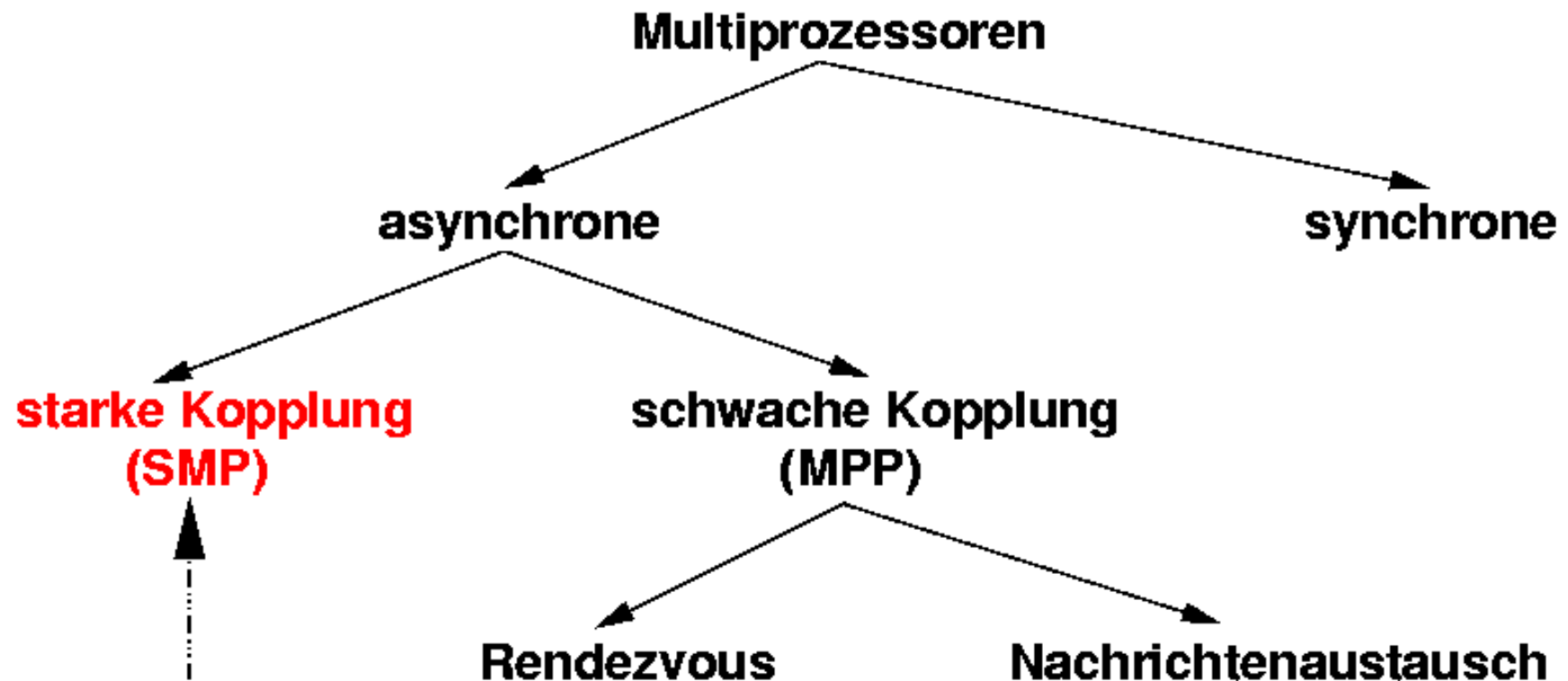
**MPP (massively parallel processing)** MPP-Rechner bestehen aus mehreren separaten CPUs, die jeweils ihren eigenen Speicher besitzen:

- kein Flaschenhals durch gemeinsamen Speicher
- jedoch Verbindungsnetzwerk erforderlich

# Einteilung von Multiprozessoren



# Einteilung von Multiprozessoren



Gegenstand dieser Vorlesung ist das **SMP-Modell** (bei  $\geq 2$  Prozessoren echte Parallelverarbeitung möglich, bei 1 Prozessor nur Multitasking).

# Zusammenfassung (1)

- Rechnerarchitekturen lassen sich nach **Flynns Taxonomie** unterscheiden,
- davon sind **SIMD** und **MIMD** Parallelrechnerarten,
- **MIMD**-Rechner können weiter in **MPP**- und **SMP**-Rechner unterschieden werden,
- Multitasking auf einem einzelnen Prozessor ist mit **SMP**-Ansatz vergleichbar.
- Java-Programmierung entspricht dem **SMP**-Ansatz (die Threads innerhalb einer „Java Virtual Machine“ nutzen einen gemeinsamen Speicherbereich),

## 2 Erste Programme

1. Wie werden nebenläufige Prozesse in Java gestartet?
2. Diskussion einiger Beispielprogramme



## 2.1 Starten von Prozessen in Java

Der Begriff „Prozess“ stammt ursprünglich aus dem Umfeld der Betriebssysteme:

**heavyweight Processes:** Betriebssystem-Prozesse mit jeweils eigenem Daten- und Programmspeicher (Beispiel: Innerhalb eines Rechners ausgeführte Programme)

**lightweight Processes** oder **threads of control:** Prozesse, die innerhalb eines Betriebssystem-Prozesses ausgeführt werden;

besitzen gemeinsamen Daten- und Programmspeicher aber getrennte lokale Variablen (z. B. : Java-(Platform)-Threads innerhalb einer Java Virtual Machine (JVM)).

# JVM und Threads

- Jede Instanz einer JVM wird in der Regel als *einzelner Betriebssystemprozess* ausgeführt
- Innerhalb jeder JVM können beliebig viele *Threads* („lightweight Processes“) ausgeführt werden; sie teilen sich alle Ressourcen der JVM (z. B. Speicher, geöffnete Dateien, . . .)
- Jedes Java-Programm wird in einer JVM gestartet und führt zunächst den sogenannten „main“-Thread aus; dieser kann weitere Threads starten
- Java-Threads werden in der Regel als Betriebssystem-Threads nebenläufig im Zeitscheibenverfahren oder echt parallel auf je einem Prozessor/-Kern ausgeführt<sup>2</sup>

---

<sup>2</sup> Ausnahme: Java Virtual Threads

# Starten von Threads

- Threads sind Java-Objekte,
- können *direkt erzeugt* werden entweder
  1. durch Erweitern der Klasse `Thread` (Vererbung)

```
public class MyThread extends Thread {  
    @Override  
    public void run() { ... }  
}
```

2. oder durch Implementieren des Interface `Runnable`

```
public class MyRun implements Runnable {  
    @Override  
    public void run() { ... }  
}
```

- werden explizit gestartet durch

```
MyThread mythread = new MyThread ();  
// oder: mythread = new Thread(new MyRun());  
mythread.start(); // ruft intern run() auf  
/* aufrufender und neuer Thread  
   laufen jetzt nebenlaeufig! */
```

- werden implizit gestartet durch Thread.Builder (Java 21):

```
MyThread mythread =  
    Thread.ofPlatform().start(new MyRun());  
/* aufrufender und neuer Thread  
   laufen jetzt nebenlaeufig! */
```

- der Zustand jedes Threads kann beeinflusst/abgefragt werden; z. B. durch `start()`, `sleep()`, `isAlive()`, `interrupt()`, `isInterrupted()`, ...

# Beispiel zum direkten Starten von Threads

Hilfsklasse Utils:

```
public class Utils {  
    public static void printThreadData(Integer n) {  
        Thread t = Thread.currentThread();  
        System.out.printf("instance of %s(%s): %s\n",  
            t.getClass().getSimpleName(), n, t.toString());  
    }  
  
    public static void doit(int n) {  
        printThreadData(n);  
        try {  
            Thread.sleep(1000); // *who* is sleeping?  
        } catch (Exception e) { }  
    }  
}
```

## Implementierung von Runnable:

```
public class MyRunnable implements Runnable {  
    private final int n;  
    public MyRunnable(int n) {  
        this.n = n;  
    }  
  
    @Override  
    public void run() {  
        Utils.doit(n);  
    }  
}
```

Erzeugen von Threads aus `MyRunnable` und ausführen →

```
public static void main(String... args) {
    Utils.printThreadData(null);

    // create named threads from MyRunnable
    Thread myFirst = new Thread(new MyRunnable(-1));
    Thread mySecond = new Thread(new MyRunnable(-2), "foo");
    myFirst.start(); mySecond.start();

    // await termination of both threads
    try { myFirst.join(); mySecond.join(); }
        catch (Exception e) { }

    // start some anonymous threads from MyRunnable
    for (int n = 1; n <= 10; n += 1) {
        (new Thread(new MyRunnable(n))).start();
    }
    System.out.println("* * *");
}
```

oder alternativ über Thread.Builder:

```
public static void main(String... args) {  
    Utils.printThreadData(null);  
  
    // create named threads via Thread.Builder and start them  
    Thread myFirst = Thread.ofPlatform()  
        .start(new MyRunnable(-1));  
    Thread mySecond = Thread.ofPlatform().name("foo")  
        .start(new MyRunnable(-2));  
  
    // await termination of both threads  
    try { myFirst.join(); mySecond.join(); }  
        catch (Exception e) { }  
  
    // start some anonymous threads from MyRunnable  
    for (int n = 1; n <= 10; n += 1) {  
        Thread.ofPlatform().start(new MyRunnable(n));  
    }  
    System.out.println("* * *");  
}
```



## Welche Ausgabe liefert die vorherige `main`-Methode?

```
instance of "Thread(null)": Thread[main,5,main]
instance of "Thread(-2)": Thread[foo,5,main]
instance of "Thread(-1)": Thread[Thread-0,5,main]
instance of "Thread(1)": Thread[Thread-1,5,main]
instance of "Thread(2)": Thread[Thread-2,5,main]
instance of "Thread(4)": Thread[Thread-4,5,main]
instance of "Thread(5)": Thread[Thread-5,5,main]
instance of "Thread(3)": Thread[Thread-3,5,main]
instance of "Thread(7)": Thread[Thread-7,5,main]
instance of "Thread(6)": Thread[Thread-6,5,main]
instance of "Thread(10)": Thread[Thread-10,5,main]
* * *
instance of "Thread(9)": Thread[Thread-9,5,main]
instance of "Thread(8)": Thread[Thread-8,5,main]
```

**Fazit:** Wie erwartet werden laufend neue Threads erzeugt und nebenläufig zum `main`-Thread ausgeführt.

# Zwei weitere Interfaces

**Frage:** Können Threads einen Ergebniswert zurückliefern?

**java.util.concurrent.Callable<V>:** Interface vergleichbar mit `Runnable`; liefert Wert vom Typ `V` zurück: statt `void run()` hier `V call()` implementieren; wird typischerweise in nebenläufigem Thread ausgeführt.

Aber wann ist die Berechnung fertig?

**java.util.concurrent.Future<V>:** Das Interface repräsentiert das Ergebnis einer asynchronen Berechnung; `boolean isDone()` fragt den Berechnungsstatus ab; `V get()` liefert das Berechnungsergebnis, wartet ggf. auf Ende der Berechnung (zur Verwendung, siehe z. B. [ExecutorService](#))

# Nachteile des direkten Erzeugens und Startens der Threads

Das *direkte* Erzeugen und Starten (s. o.) kann nachteilig sein, wenn z. B.

- „viele“ „kurzlebige“ Threads benötigt werden: Das Erzeugen zusammen mit dem Registrieren beim Betriebssystem ist „teuer“  
→ „alte“ Threads wiederverwenden!
- eine Kontrolle/Begrenzung der Gesamtanzahl der laufenden Threads notwendig ist  
→ Threads sofort erzeugen, aber ggf. erst später starten!
- ...

# Interfaces und Factories zum Erzeugen/Starten von Threads

**java.util.concurrent.Executor:** Interface enthält die Methode `public void execute(Runnable r);` „`r` wird irgendwann in der Zukunft ausgeführt“; z. B. in neuem Thread, in wiederverwendetem Thread, im aufrufenden Thread (näheres regelt die Implementierung)

**java.util.concurrent.ExecutorService:** Das Interface erweitert `Executor`; bietet `submit`-Methoden für `Runnable` und `Callable`, die ein `Future` zurückliefern; ermöglicht `close`, `shutdown` bzw. `shutdownNow` des Service; Instanzen können über Factory-Methoden der Klasse `Executors` erzeugt werden

**java.util.concurrent.Executors:** Die Klasse besitzt u.a. statische Factory-Methoden zum Erzeugen von `ExecutorService`-Instanzen mit bestimmten Eigenschaften:

- `newCachedThreadPool()`: Threads aus dem Pool werden wiederverwendet; bei Bedarf werden neue Threads erzeugt und zum Pool hinzugefügt
- `newFixedThreadPool(int nThreads)`: Pool enthält feste Anzahl von Threads; wenn alle Threads aktiv sind, warten neue Tasks in einer Warteschlange auf einen freien Thread; aufgrund eines Fehlers terminierte Threads werden durch einen neuen ersetzt

- `newSingleThreadExecutor()`: wie `newFixedThreadPool(1)`, aber nicht rekonfigurierbar (z. B. durch Cast zu `ThreadPoolExecutor` und anschließendem Aufruf von `setMaximumPoolSize`)
- `newScheduledThreadPool(int poolSize)`: liefert Implementierung eines `ScheduledExecutorService`, der Tasks mit einer bestimmten Verzögerung oder periodisch ausführen kann
- `newWorkStealingPool(int par)`: Pool enthält „ausreichend“ Threads, um Parallelität `par` sicherzustellen; kann mehrere Warteschlangen für Tasks enthalten (z. B. eine pro Thread), um Zugriffskonflikte zu vermeiden; ist eine Warteschlange leer, können Tasks von anderen Threads „gestohlen“ werden (Zugriffs-

strategie auf Warteschlangen ist typischerweise LIFO für den zugehörigen Thread und FIFO bei „Work Stealing“ aus einer anderen Warteschlange)

- `newVirtualThreadPoolExecutor()`: alle an diesen Executor übermittelten Tasks werden als Virtuelle Threads gestartet ([siehe später](#)); die Anzahl virtueller Threads ist nicht beschränkt

**java.util.concurrent.ForkJoinPool:** Klasse bietet speziellen ExecutorService für rekursive Berechnungsaufgaben (`ForkJoinTask<V>`) der „Work Stealing“ implementiert; die Anzahl der Threads ist beschränkt;

es gibt vordefinierten Standardpool:

```
public static ForkJoinPool commonPool();
```

spezielle Pools können über Konstruktoren erzeugt werden: `public ForkJoinPool(int parallelism);`

`execute(ForkJoinTask<?> task)` führt `task` asynchron im Pool aus;

`T invoke(ForkJoinTask<T> task)` wie `execute`, wartet jedoch auf das Berechnungsergebnis (identisch mit Sequenz `task.fork(); task.join()`)



**`java.util.concurrent.ForkJoinTask<V>`**: abstrakte Basisklasse für rekursive Berechnungen nach dem „Divide and Conquer“-Prinzip;

alle Berechnungen werden asynchron innerhalb eines `ForkJoinPool` durchgeführt; stellt eine auf hohen Durchsatz optimierte Form eines `Future` dar;

`ForkJoinTask<V> fork()` führt diesen Task in demselben Pool aus, wie der aktuell laufende Task;

`V join()` liefert das Berechnungsergebnis dieses Tasks, wartet ggf. auf das Ende der Berechnung;

`V invoke()` führt diesen Task aus und wartet ggf. auf das Ende der Berechnung;

**java.util.concurrent.RecursiveTask<V>:** von `ForkJoinTask` abgeleitete, direkt benutzbare Klasse; es ist lediglich die Methode `abstract V compute()` mit der Berechnungsvorschrift für den Ergebniswert zu implementieren

**java.util.concurrent.RecursiveAction:** von `RecursiveTask<Void>` abgeleitete Klasse für Rekursionen, die keinen Ergebniswert liefern (`abstract void compute()`)<sup>3</sup>

---

<sup>3</sup> Beispiele siehe z. B. in der API-Dokumentation der Klassen `RecursiveTask` und `RecursiveAction`.

# Beispiel: CachedThreadPool

```
public class CachedPool {  
    public static void runTest(  
        ExecutorService executorService, int timeOut) {  
        // submit some runnables to my executorService  
        for (int i = 0; i < 10; i += 1) {  
            executorService.submit(new MyRunnable(i));  
        }  
  
        // wait some time  
        try {Thread.sleep(timeOut);} catch (Exception e) {}  
  
        System.out.println("submitting more Runnables...");  
        // again execute some runnables in my executorService  
        for (int i = 0; i < 10; i += 1) {  
            executorService.execute(new MyRunnable(100 + i));  
        }  
    }  
}
```

```
public static void main(String... args) {  
    ExecutorService executorService =  
        Executors.newCachedThreadPool();  
  
    // print data of main thread  
    Utils.printThreadData(null);  
  
    runTest(executorService, 1500);  
    // runTest(executorService, 0);  
  
    Utils.shutdown(executorService);  
}  
}
```

## Fragen:

- Welche Ausgabe liefert die vorherige `main`-Methode?
- Welche Ausgabe liefert die vorherige `main`-Methode, wenn dort der Aufruf `runTest(executorService, 1500)` durch `runTest(executorService, 0)` ersetzt wird?
- Was ändert sich, wenn in der vorherigen `main`-Methode die Variable `executorService` mit `Executors.newFixedThreadPool(3)` initialisiert wird?

## Antworten: ...

# Spezialfall: Nebenläufige Streams

**Frage:** Was passiert beim Aufruf der folgenden Methode?

```
public static void runTest(IntStream stream) {  
    // submit some runnables to the stream in parallel  
    stream.parallel().forEach(Utils::doit);  
}
```

zum Beispiel durch

```
runTest(IntStream.range(0, 10));
```

**Antwort:** Die Stream-Pipeline wird *nebenläufig* im Standard-`ForkJoinPool` unter Zuhilfenahme des (angehaltenen) `main`-Thread ausgeführt!

# Verketteten asynchroner Aktionen

In „reaktiven Systemen“<sup>4</sup> soll oft nach Abschluss einer asynchronen Aktion automatisch eine weitere asynchrone Aktion gestartet werden!

**Lösung:** `java.util.concurrent.CompletableFuture`

```
Runnable task1 = () -> Utils.doit(1);  
Runnable task2 = () -> Utils.doit(2);  
Runnable task3 = () -> Utils.doit(3);
```

```
CompletableFuture.runAsync(task1).  
    thenRun(task2).thenRun(task3);
```

**Frage:** In welchen Threads werden die Tasks ausgeführt?

---

<sup>4</sup> Für eine Einführung siehe z. B. [[Spr24](#)]

# Varianten von `thenRun`

`java.util.concurrent.CompletableFuture` definiert diverse Varianten zum Starten asynchroner Aktionen!

**Frage:** Unterschiede zwischen ...

- `runAsync(Runnable r),`
- `runAsync(Runnable r, Executor e),`
- `thenRun(Runnable r),`
- `thenRunAsync(Runnable r)`
- `thenRunAsync(Runnable r, Executor e)`
- `supplyAsync(Supplier<U> s)`
- ...



# Java Virtual Threads – Vorüberlegungen

## Fakten:

- In Serveranwendungen lassen sich die einzelnen „Aufträge“ in der Regel unabhängig voneinander bearbeiten!  
→ ideal für Threads
- Aber: Serveraufträge enthalten meist **blockierende** I/O- oder Netzwerkaufrufe → Kontextswitch  
zur Auslastung der CPU sind „sehr viele“ Aufträge/Threads erforderlich ( $\approx 10^6$ )
- Aber: Erzeugen von (Platform-)Threads ist teuer ( $\approx ms$  pro Thread), Speicherbedarf pro Thread:  $\approx 2$  MByte
- Daher: Handhaben „sehr vieler“ (Platform-)Threads ist nicht praktikabel!

# Java Virtual Threads – Idee

- „leichtgewichtige Threads“, die nicht vom Betriebssystem, sondern von der JVM verwaltet werden.
- günstig zu erzeugen und im Überfluss vorhanden (kein „Pooling“ notwendig)<sup>5</sup>
- verwenden zur Ausführung klassische OS-Threads („Carrier“)
- belegen OS-Thread nur bei der Ausführung von Anweisungen; schneller Kontextswitch bei blockierenden Operationen
- Eigenschaften wie `java.lang.Threads`

→ Java Virtual Threads

<sup>5</sup> Idee vergleichbar mit virtuellem Speicher in Betriebssystemen

# Java Virtual Threads – noch mehr

- spielen ihre Vorteile nur aus, wenn sie blockierenden Code enthalten (nicht bei rechenintensiven Aufgaben)!
- (blockierender) imperativer Code ist einfacher zu lesen/warten als asynchroner, [reaktiver Ansatz](#) (keine unzusammenhängende Folge von Runnables/Lambdas)
- Verwaltung erfolgt über speziellen ForkJoinPool (FIFO-Strategie mit Work Stealing)
- Anzahl der Carrier-Threads (OS-Threads) in dem ForkJoinPool ist über Property einstellbar; Standard: Anzahl der vorhandenen Kerne
- für weitere Details siehe z. B. [[Ora24b](#)]

# Thread-Typen in Java

Name	OS Thread scheduling	Java-Version
„Platform Threads“	1 : 1	seit Java 1.0
„Green Threads“	$M : 1$	Java 1.0 – 1.2
„Virtual Threads“	$M : N$	seit Java 21

# Java Virtual Threads – Beispiel (1)

Start über `Thread.startVirtualThread`:

```
public class VirtualthreadsPerThreadBuilder {  
  
    public static void main(String[] args) {  
        final int noOfTasks = 10_000;  
  
        Thread[] allThreads = new Thread[noOfTasks];  
        IntStream.range(0, noOfTasks).forEach(i ->  
            allThreads[i] =  
                Thread.startVirtualThread(() -> Utils.doit(i)));  
  
        for (Thread t : allThreads) {  
            try { t.join(); } catch (Exception e) { }  
        }  
    }  
}
```

# Java Virtual Threads – Beispiel (2)

Start über Thread.Builder:

```
public class VirtualthreadsPerThreadBuilder {  
  
    public static void main(String[] args) {  
        final int noOfTasks = 10_000;  
  
        Thread.Builder threadBuilder = Thread.ofVirtual();  
        // Thread.Builder threadBuilder = Thread.ofPlatform();  
  
        Thread[] allThreads = new Thread[noOfTasks];  
        IntStream.range(0, noOfTasks).forEach(i -> {  
            String name = String.format("doit(%s)", i);  
            allThreads[i] =  
                threadBuilder.name(name).start(() -> Utils.doit(i));  
        });  
    }  
}
```

```
for (Thread t : allThreads) {  
    try {  
        t.join();  
    } catch (Exception e) { }  
}  
}  
}
```

# Java Virtual Threads – Beispiel (3)

Start über `ExecutorService` (vgl. [vorne](#)):

```
public class VirtualthreadsPerExecutorService {  
  
    public static void main(String[] args) {  
        final int noOfTasks = 10_000;  
  
        final ExecutorService executorService =  
            Executors.newVirtualThreadPerTaskExecutor();  
  
        IntStream.range(0, noOfTasks).forEach(i ->  
            executorService.submit(() -> Utils.doit(i));  
        );  
  
        executorService.close();  
    }  
}
```



## 2.2 Gute Programmbeispiele?

### Frage:

Können auch Probleme bei der Parallelprogrammierung auftreten?

Die Frage soll anhand von drei einfachen Programmbeispielen beantwortet werden.

# 1. Beispiel: StrangeCounter

Gegeben sei das folgende Java-Programm:

```
public class StrangeCounter {
    private final static int INCREMENTERS = 2;
    private final static int RUNS = 5;

    private static long counter = 0;

    protected static class Incrementer implements Runnable {
        private final CountdownLatch start, end;
        public Incrementer(CountDownLatch start,
                           CountdownLatch end) {
            this.start = start;
            this.end = end;
        }
    }
}
```

```
public void run() {  
    try {  
        start.await();  
        for (int i = 0; i < RUNS; i++) {  
            counter++;  
        }  
        end.countDown();  
    } catch (Exception e) { ... }  
}  
  
}  
  
public static void main(String[] args) {  
    CountdownLatch startLatch =  
        new CountdownLatch(1);  
    CountdownLatch endLatch =  
        new CountdownLatch(INCREMENTERS);  
}
```

```

Thread.Builder tb = Thread.ofPlatform();
// Thread.Builder tb = Thread.ofVirtual();
for (int i = 0; i < INCREMENTERS; i++) {
    tb.start(new Incrementer(startLatch, endLatch));
}

try {
    System.out.println("Starting with counter = "
                        + counter);
    startLatch.countDown();
    endLatch.await();
    long totalInc = RUNS * INCREMENTERS;
    System.out.println("Finished after " + totalInc
                        + " increments with counter = "
                        + counter);
} catch (Exception e) { ... }
}

```

# Fragen zu StrangeCounter

- Was macht das Programm? Arbeitet es korrekt?
- Ändert sich etwas, wenn z. B. `INCREMENTERS = 20` und `RUNS = 50` gesetzt werden? Was passiert hier?
- Was passiert, wenn der Wert `INCREMENTERS = 1` ist?
- Gibt es für `INCREMENTERS > 1` Unterschiede im Ergebnis, wenn für den Zähler der Typ `int` statt `long` verwendet wird? Warum?
- Gibt es Unterschiede im Ergebnis, wenn Sie die parallelen Prozesse statt über direkte Threads auf andere Arten starten (z. B. als `Runnable` über einen `Executor`, als `Callable` über einen `ExecutorService`, als parallele Streams, ...)?

## 2. Beispiel: „Webseiten-Zugriffsmerker“

**Aufgabe:** Ein in einen Webserver zu integrierendes Modul soll zu ausgewählten Webseiten das Datum des letzten Zugriffs speichern.

**Lösung:** Implementierung in Java unter Einsatz des Abstrakten Datentyps „Wörterbuch“ (`Map`-Interface).

Fasse den Pfadnamen jeder Seite als Objekt vom Typ `String` auf und assoziiere mit ihm das `Date`-Objekt des aktuellen Datums.

```
public class WebStat {  
    // map used to store the (key, value) pairs  
    private Map<String, Date> map;  
  
    public WebStat(Map<String, Date> map) {  
        // initialize local variable from constructor  
        this.map = map;  
        this.map.clear();  
    }  
  
    public void rememberLastAccess(String location) {  
        map.put(location, new Date());  
    }  
}
```

```
public class WebClientSimulator implements Runnable {  
    private final WebStat webStat;  
  
    private final int noOfPages;  
  
    private final int name;  
  
    private final CountDownLatch start, end;  
  
    public WebClientSimulator(WebStat webStat, int noOfPages,  
                               int no,  
                               CountDownLatch start,  
                               CountDownLatch end) {  
  
        this.webStat = webStat;  
        this.noOfPages = noOfPages;  
        this.name = no;  
        this.start = start;  
        this.end = end;  
    }  
}
```



```

public void run() {
    try {
        start.await();
        // visit all pages
        for (int i = 0; i < noOfPages; i++) {
            /* create location name from thread name
               and current loop index */
            String location =
                "/directory_accessed_by_" + name
                + "/page" + i + ".html";
            webStat.rememberLastAccess(location);
        }
        end.countDown()
    } catch (Exception e) { ... }
}
}

```

```
public class StrangeMap {  
  
    public StrangeMap(Map<String, Date> map, int noOfReaders,  
                      int noOfPagesPerReader) {  
        this.map = map;  
        this.noOfReaders = noOfReaders;  
        this.noOfPagesPerReader = noOfPagesPerReader;  
        this.start = new CountDownLatch(1);  
        this.end = new CountDownLatch(noOfReaders);  
        start();  
    }  
}
```

```

void start() {
    try {
        WebStat webStat = new WebStat(map);

        // create readers
        Thread.Builder tb = Thread.ofPlatform();
        for (int i = 0; i < noOfReaders; i++) {
            tb.start(new WebClientSimulator(webStat,
                noOfPagesPerReader, i, start, end));
        }

        // start readers
        start.countDown();
        // wait for termination of readers
        end.await();
    } catch (Exception e) { ... }
}

```

```
public static void main(String[] arg) {  
    Map<String, Date> map;  
    // map = new Hashtable<String, Date>(1);  
    map = new HashMap<String, Date>(1);  
    // map = Collections.synchronizedMap(  
    //         new HashMap<String, Date>(1));  
    new StrangeMap(map, 25, 1500);  
  
    /* check content of map now! */  
}  
}
```

## Fragen:

1. Wie lautet der Inhalt von `map` am Ende der `main`-Methode?

2. Was passiert, wenn viele Threads nebenläufig die Methode `rememberLastAccess` derselben `WebStat`-Instanz aufrufen?

**Achtung:** Manche Map-Implementierungen sind nicht „thread-safe“!

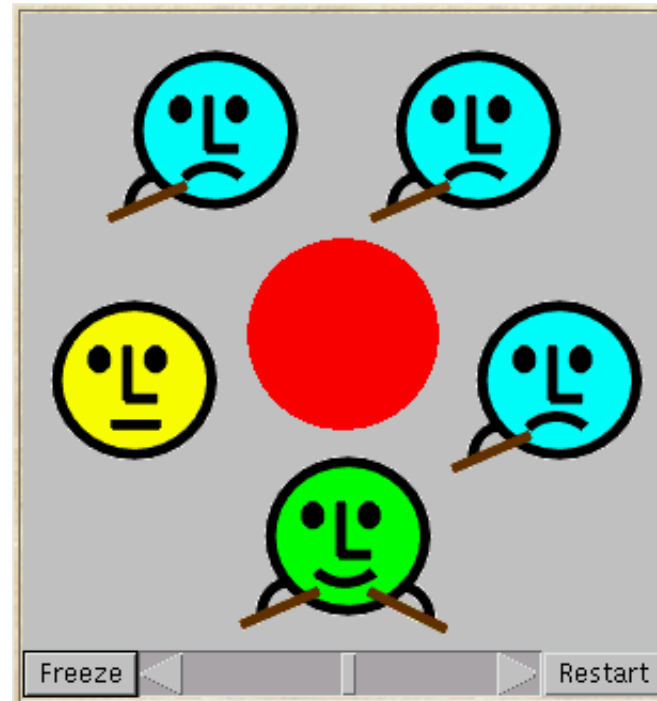
Die Speicheroperationen werden nicht atomar ausgeführt. Das führt zu „Müll“ im Speicher. Die Zugriffe auf gemeinsam benutzte Ressourcen müssen synchronisiert werden!

### 3. Beispiel: „Die dinierenden Philosophen“

Fünf Philosophen sitzen an einem runden Tisch. In der Mitte steht eine Schüssel mit Spaghetti. Die Philosophen sind genügsam, daher verzichten sie auf Teller und besitzen auch nur fünf Gabeln, die sie zwischen sich legen. Somit hat jeder Philosoph rechts und links von sich je eine Gabel.

Zum Essen benutzt jeder Philosoph allerdings immer beide neben ihm liegenden Gabeln, so dass er eventuell warten muss, wenn er Essen möchte und eine Gabel gerade durch seinen Nach-

barn verwendet wird.



**Frage:** Wie lässt sich das Leben der Philosophen („Denken“, „Essen“, „Denken“, „Essen“, ...) modellieren?

## „Lösung“:

- Modelliere den Philosophen als Prozess: „Denken“ → „linke Gabel nehmen“ → „rechte Gabel nehmen“ → „Essen“ → „linke Gabel ablegen“ → „rechte Gabel ablegen“ → „Denken“ → ...
- starte fünf dieser Prozesse parallel,
- was wird/kann passieren?

**Achtung:** Die Anforderung der Ressourcen geschieht nicht geordnet! Es können **Verklemmungen** (engl. deadlocks) auftreten!

# Zusammenfassung (2)

- Auswirkungen des „nachlässigen“ Einsatzes von Parallelismus wurden an Beispielen demonstriert.
- Reines Testen reicht bei nebenläufigen Prozessen nicht aus, da das Ergebnis nicht vorhersehbar von der zeitlichen „Verzahnung“ der nebenläufigen Prozesse abhängt!

**Fazit:** Die **Korrektheit** der Implementierung **muss** (formal) **bewiesen werden!**



# 3 Modellierung von Prozessen

- Wie können Prozesse modelliert werden?
- Wie lassen sich Prozesse in Java darstellen?

## 3.1 Sequentielle Prozesse

### Fakten:

- Jeder Prozess besteht aus einer Folge atomarer Aktionen
- er lässt sich zu jedem Zeitpunkt durch die Werte seiner Variablen (d. h. seinen Zustand) beschreiben
- jedes Ausführen einer Aktion führt eine Zustandstransformation durch

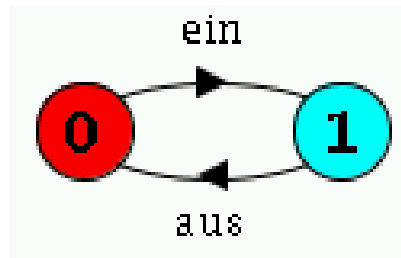
- bei endlicher Zustandsmenge lassen sich Prozesse graphisch als (gerichtete) Zustandsgraphen darstellen:
  - jeder Zustand entspricht einem Knoten,
  - jede Aktion entspricht einer Kante
- jeder (gerichtete) Zustandsgraph lässt sich als Prozess auffassen.

**Definition:** Solche Zustandsgraphen werden als „*LTS* (*Labeled Transition System*)“ bezeichnet.

Der Anfangszustand des Prozesses erhält immer die Bezeichnung 0.

# LTS-Beispiel: Prozess „Lichtschalter“

- Zustände des Prozesses:  $\{0, 1\}$ ,
- Aktionen des Lichtschalters: *ein*, *aus*,
- Zustandsgraph:



- Aktionsfolge:  $ein \rightarrow aus \rightarrow ein \rightarrow aus \rightarrow \dots$

**Bemerkung:** Selbst bei endlicher Zustandsmenge, muss das beschriebene Verhalten nicht endlich sein.

**Definition:** Eine durch die Ausführung eines Prozesses entstehende Aktionsfolge wird als *Trace* bezeichnet.

# Algebraische Beschreibung von Prozessen durch „FSP (Finite State Processes)“

- die graphische Darstellung wird bei einer „großen“ Anzahl von Zuständen unübersichtlich
- eine algebraische Notation ist für eine „griffige“ Beschreibung besser geeignet

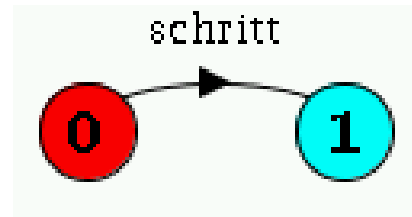
**Definition:** Ist  $x$  eine Aktion und  $P$  ein beliebiger Prozess, so wird durch den *Präfixoperator*  $(x \rightarrow P)$  ein neuer Prozess definiert, der zunächst die Aktion  $x$  ausführt und sich danach wie der Prozess  $P$  verhält.

**Bemerkung:** In FSP werden Aktionen mit einem *kleinen* und Prozesse mit einem *großen* Anfangsbuchstaben geschrieben.

**Definition:** *STOP* bezeichnet einen besonderen Prozess, von dem keine weiteren Aktionen ausgehen.

**Beispiel:** Ein Prozess, der nur eine Aktion ausführt und dann terminiert:

`EINSCHRITT = (schritt -> STOP) .`



**Bemerkung:** In FSP sind Rekursionen möglich.



# FSP-Beispiel: Prozess „Lichtschalter“

Aus dem Trace des Prozesses lässt sich eine FSP-Beschreibung ablesen:

$$\begin{array}{c} \overbrace{\hspace{15em}}^{AUS} \\ \text{ein} \rightarrow \underbrace{\text{aus} \rightarrow \overbrace{\text{ein} \rightarrow \text{aus} \rightarrow \text{ein} \rightarrow \text{aus} \rightarrow \dots}^{AUS}}_{EIN} \end{array}$$

SCHALTER = AUS,

AUS = (ein -> EIN),

EIN = (aus -> AUS) .

Eine andere Beschreibung entsteht durch Einsetzen der Beschreibung von EIN in der von AUS:

$$\text{SCHALTER} = \text{AUS},$$
$$\text{AUS} = (\text{ein} \rightarrow (\text{aus} \rightarrow \text{AUS})) .$$

Noch weiter vereinfacht, ergibt sich:

$$\text{SCHALTER} = (\text{ein} \rightarrow \text{aus} \rightarrow \text{SCHALTER}) .$$

Alle drei Beschreibungen von SCHALTER sind gleichwertig.

# LTSA Analyse-Tool (LTSA)

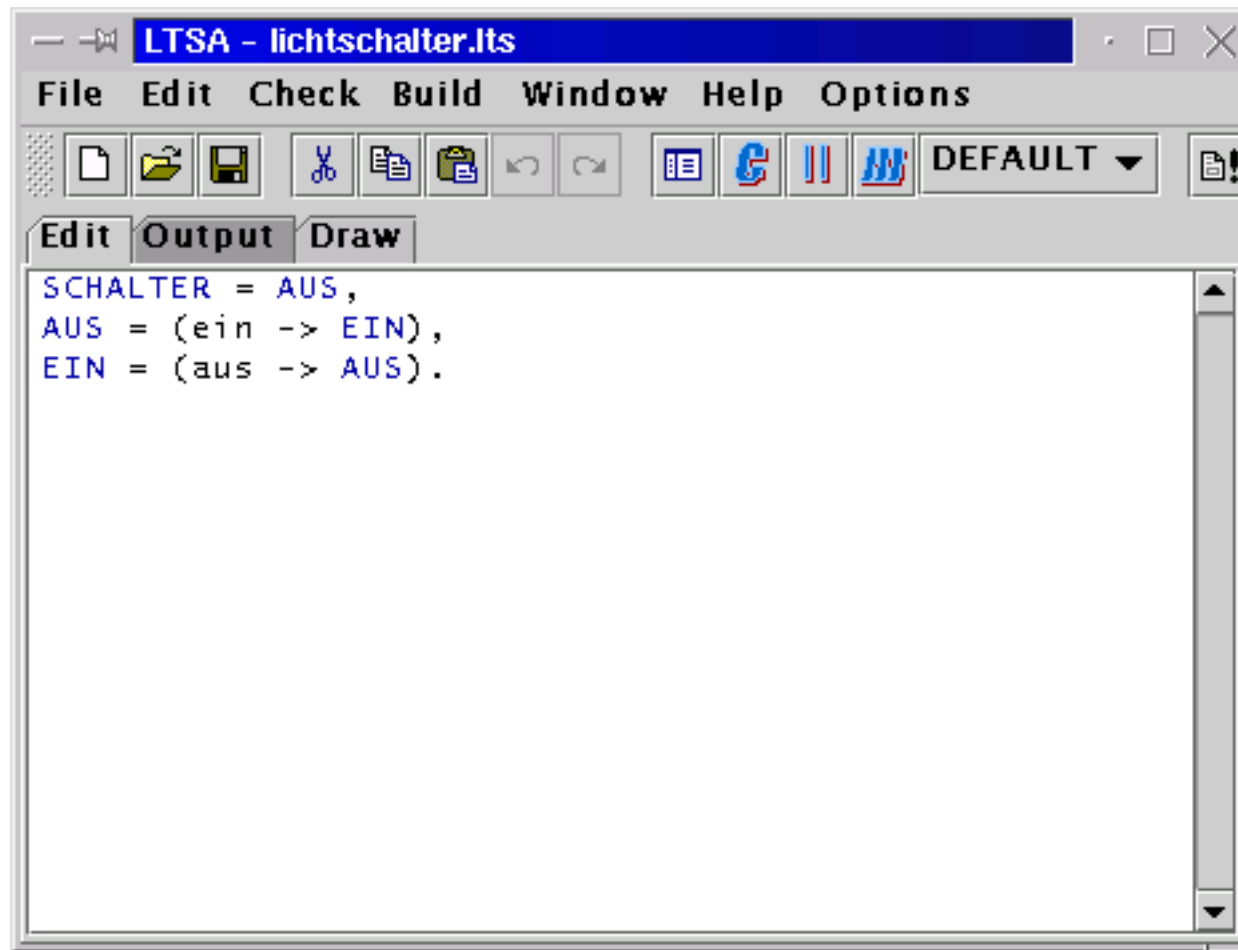
Die Analyse der Prozesse soll hier nicht „von Hand“ sondern mit Tool-Unterstützung durchgeführt werden.

LTSA ...

- ... liest algebraische FSP-Beschreibung
- ... bietet einen einfachen Editor
- ... produziert eine Darstellung des Zustandsgraphen
- ... produziert Traces
- ...

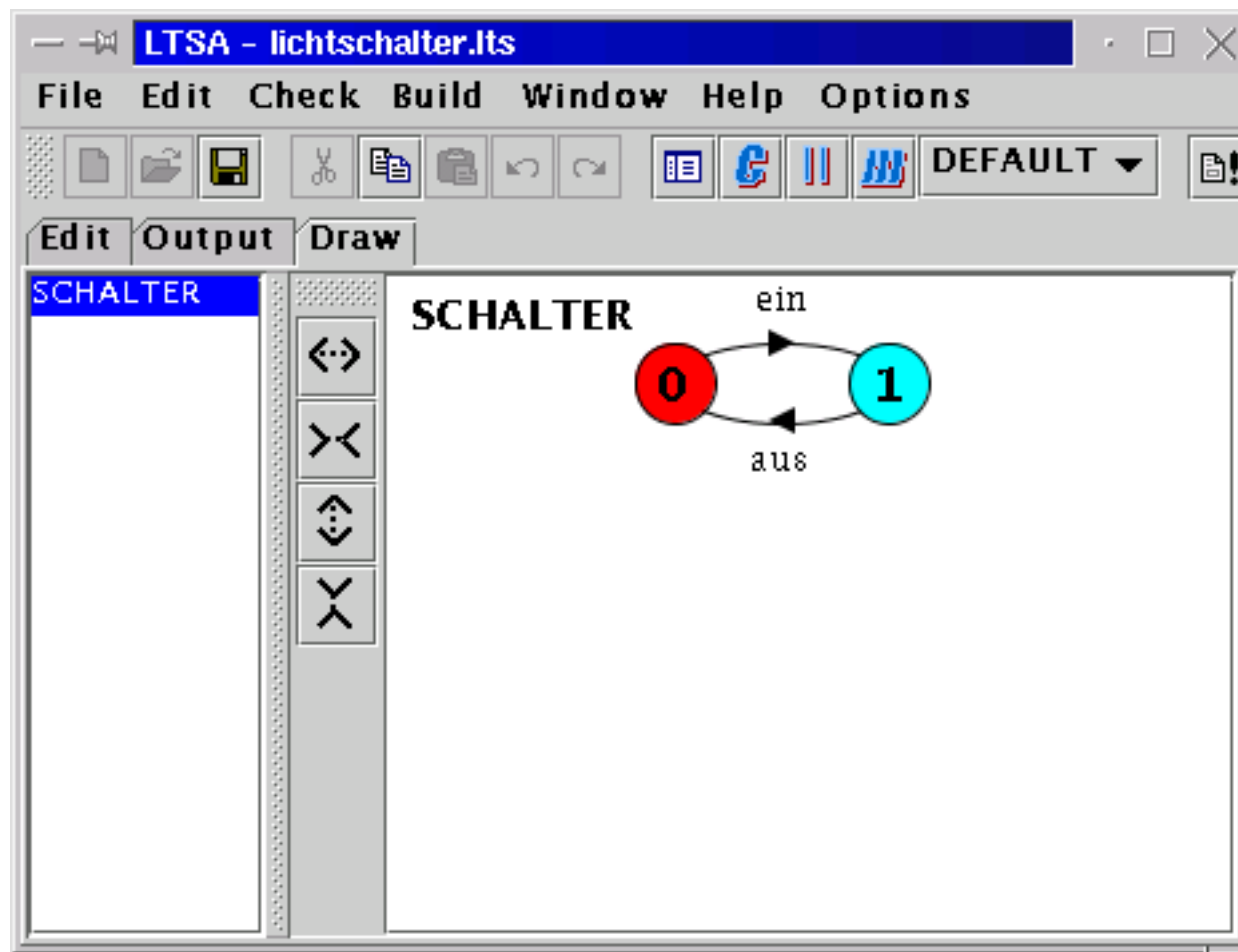
# Analyse des Prozesses „Lichtschalter“ (1)

Der Editor von LTSA:



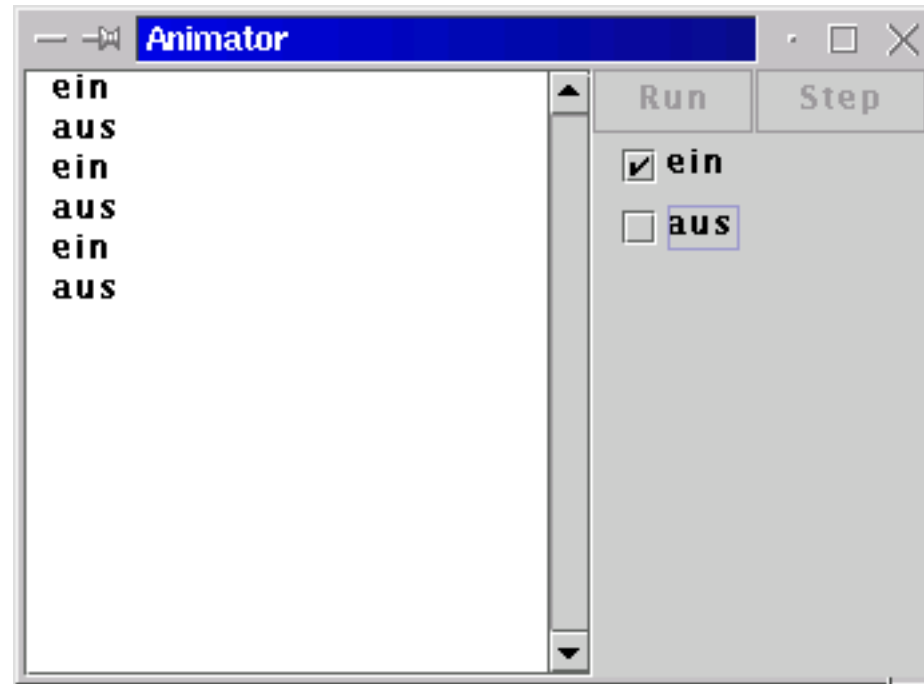
# Analyse des Prozesses „Lichtschalter“ (2)

LTSA-Ausgabe des Zustandsgraphen:



# Analyse des Prozesses „Lichtschalter“ (3)

Der LTSA-Animator liefert Traces:



# Auswahloperator

Alle bisherigen Prozesse lieferten genau einen Trace; es war keinerlei „Auswahl“ möglich.

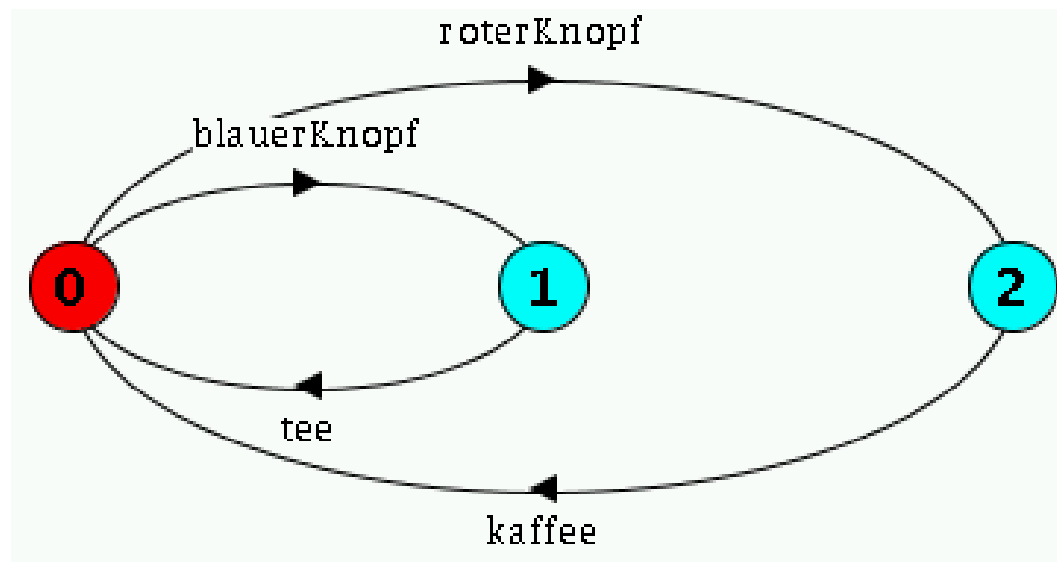
**Definition:** Seien  $x$  und  $y$  Aktionen sowie  $P$  und  $Q$  Prozesse, so beschreibt der *Auswahloperator*  $(x \rightarrow P \mid y \rightarrow Q)$  einen Prozess, der sich entweder wie  $(x \rightarrow P)$  oder wie  $(y \rightarrow Q)$  verhält.

# Beispiel: „Getränkeautomat“

Getränkeautomat als FSP-Beschreibung:

```
GETRAENKE = (roterKnopf -> kaffee -> GETRAENKE  
             | blauerKnopf -> tee -> GETRAENKE) .
```

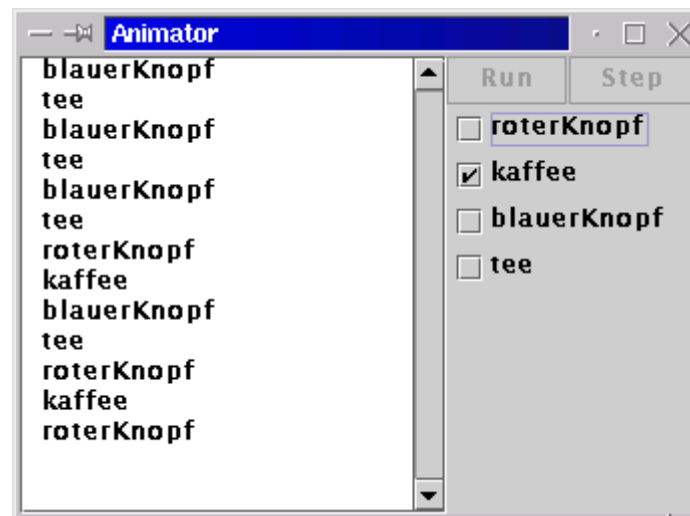
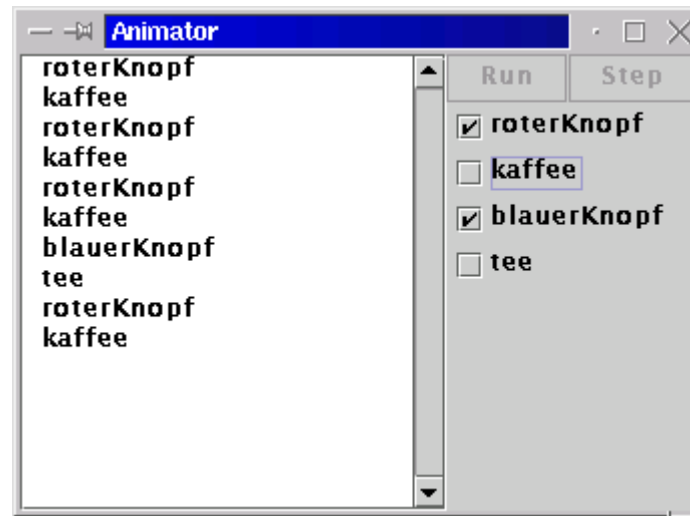
Getränkeautomat als Zustandsgraph:





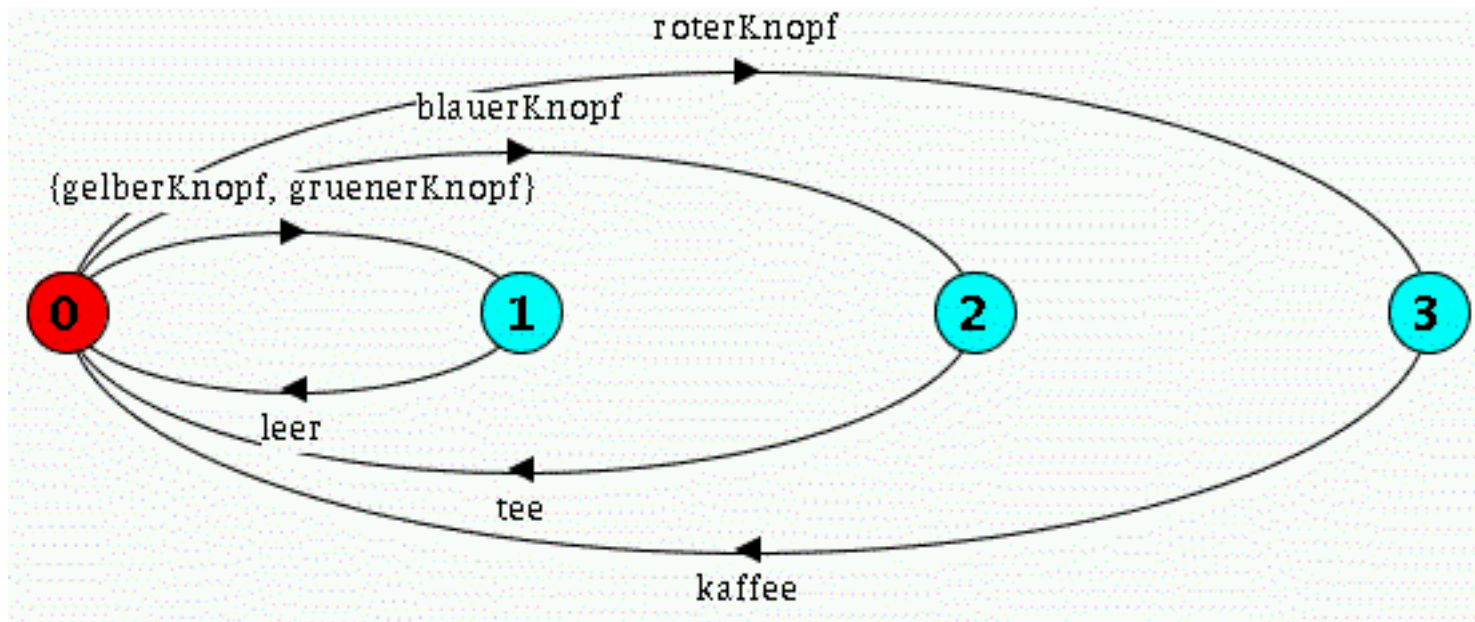
**Bemerkung:** Im Zustandsdiagramm wird nicht zwischen *Eingabe*- und *Ausgabe*aktionen unterschieden. Die Unterscheidung ergibt sich aus der Bedeutung d. h. aus dem Zusammenhang.

# Zwei Traces des Getränkeautomaten (erzeugt mit Hilfe des LTSA-Animator):



## Eine Auswahl mit mehr als zwei Alternativen:

```
GETRAENKE = (roterKnopf -> kaffee -> GETRAENKE  
            | blauerKnopf -> tee -> GETRAENKE  
            | gruenerKnopf -> LEER  
            | gelberKnopf -> LEER),  
LEER = (leer -> GETRAENKE).
```



# Spezialfälle der Alternative

Betrachte noch einmal das vorige Beispiel (Ausschnitt):

```
GETRAENKE = (gruenerKnopf -> LEER  
             | gelberKnopf -> LEER) ,  
...
```

In der **Auswahl** folgt auf zwei *unterschiedliche* Aktionen *derselbe* Prozess!

**Frage:** Lässt sich das auch „umdrehen“, d. h. auf *dieselbe* Aktion folgen *unterschiedliche* Prozesse?

```
GETRAENKE = (roterKnopf -> KAFFEE  
             | roterKnopf -> TEE) ,  
...
```

**Frage:** Was bedeutet das?

# Nichtdeterministische Auswahl

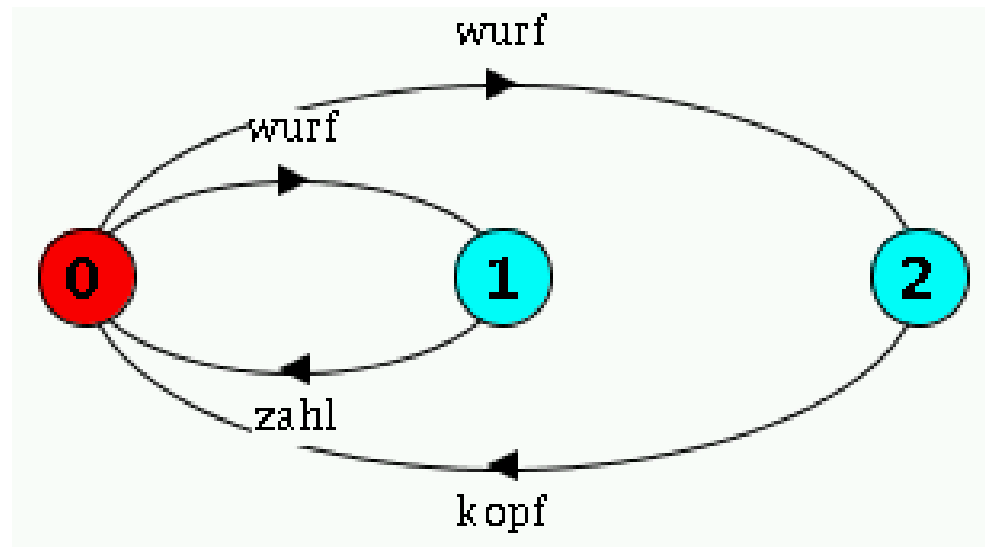
Bisher war in der Auswahl nach der ersten Aktion der folgende Prozess genau bestimmt (deterministische Ausführung).

**Definition:** Eine Auswahl  $(x \rightarrow P \mid x \rightarrow Q)$  heißt *nichtdeterministisch*, da sie sich nach Ausführen von  $x$  entweder wie  $P$  oder wie  $Q$  verhalten kann (das genaue Verhalten ist *nicht* bestimmt).

# Beispiel: „Münzwurf“

Die Folgeaktion (d. h. die Ausgabe) der Aktion `wurf` ist nicht vorherbestimmt.

```
MUENZE = ( wurf -> kopf -> MUENZE  
          | wurf -> zahl -> MUENZE ) .
```



# Indizierte Prozesse und Aktionen

Um die Verarbeitung verschiedener Eingabewerte modellieren zu können, können sowohl die Prozesse in der algebraischen Darstellung als auch die Aktionen im Zustandsgraphen indiziert werden.

**Bemerkung:** **Indices von Prozessen** im LTS entsprechen **Parametern** in Programmiersprachen!

**Bemerkung:** Indices haben immer einen endlichen Wertebereich (dadurch bleibt das Modell in FSP und LTS endlich).

# Beispiel: „indizierter Prozess“

Ein Prozess, der eine Zahl im Wertebereich von 0..3 speichern kann:

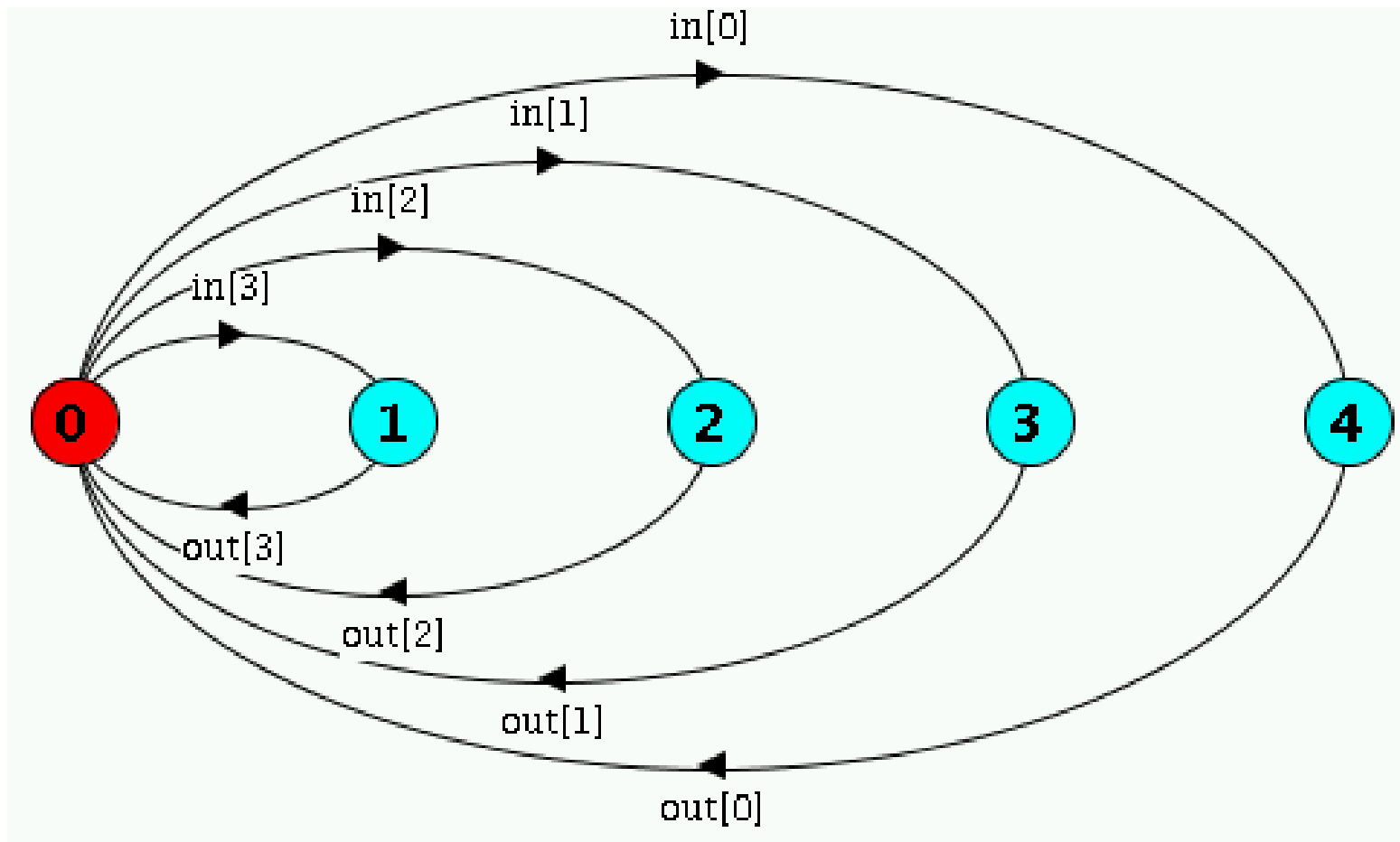
```
PUFFER = ( in[0] -> out[0] -> PUFFER
          | in[1] -> out[1] -> PUFFER
          | in[2] -> out[2] -> PUFFER
          | in[3] -> out[3] -> PUFFER) .
```

oder kürzer als:

```
range T = 0..3
PUFFER = ( in[i:T] -> out[i] -> PUFFER) .
```



und der Zustandsgraph für beide Beschreibungen:



# Beispiel: „doppelt indizierter Prozess“

Ein Prozess, der zwei Zahlen addieren kann:

```
const N = 1
```

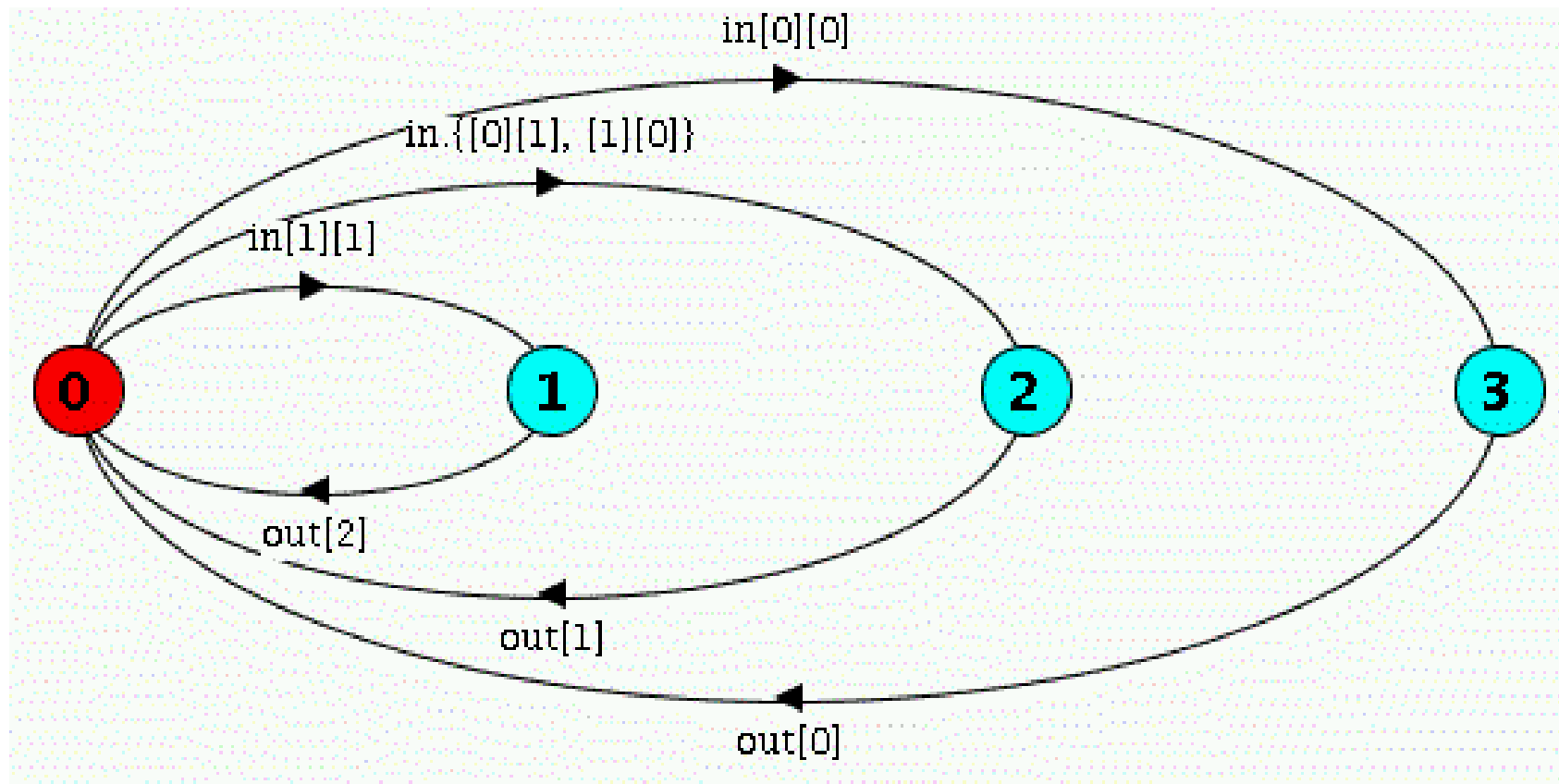
```
range T = 0..N
```

```
range R = 0..2*N
```

```
SUM          = (in[a:T][b:T] -> TOTAL[a+b]),
```

```
TOTAL[s:R] = (out[s] -> SUM) .
```

## Zustandsgraph des Summenprozesses:



# Parametrisierung von Prozessen

Prozesse können auch parametrisiert werden, so dass sie für allgemeine Werte definiert sind:

**Beispiel:** Puffer der Größe  $N$ :

$$\text{PUFFER}(N=3) = (\text{in}[i:0..N] \rightarrow \text{out}[i] \rightarrow \text{PUFFER}) .$$
$$\text{PUFFER}(N=10) = (\text{in}[i:0..N] \rightarrow \text{out}[i] \rightarrow \text{PUFFER}) .$$

# Bewachte Aktion

Bisher können noch keine Bedingungen (vergleichbar der if-Anweisung) formuliert werden.

**Definition:** Die Auswahl ( $\text{when } B \ x \rightarrow P \mid y \rightarrow Q$ ) beschreibt einen Prozess, der sich wie folgt verhält: Ist die Bedingung des *Wächters*  $B$  erfüllt, können entweder  $x$  oder  $y$  ausgeführt werden; ist  $B$  nicht erfüllt, kann als nächstes nur die Aktion  $y$  ausgeführt werden.

**Bemerkung:** Die Bedeutung bewachter Aktionen unterscheidet sich geringfügig von if-Anweisungen.

## **Beispiel:** Die Anweisung

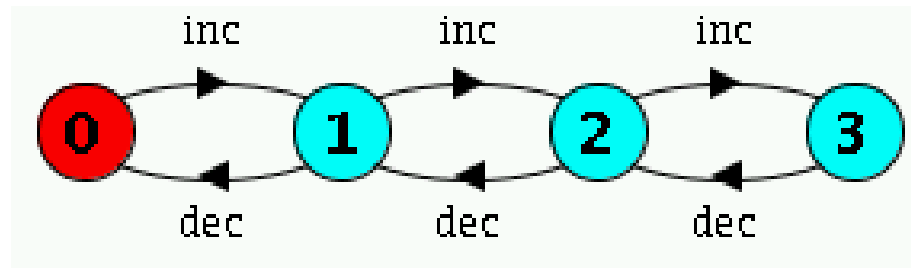
```
if (i==0) {  
    x(); P();  
} else {  
    y(); Q();  
}
```

ist folgendermaßen als bewachte Aktion zu formulieren:

```
(when (i==0) x -> P |  
when (i!=0) y -> Q) .
```

# Beispiel: „Zähler“

Ein Zähler kann mit den Aktionen *inc* und *dec* hoch- bzw. heruntergezählt werden.



$\text{ZAEHL}(N=3) = \text{ZAEHL}[0],$

$\text{ZAEHL}[i:0..N] = (\text{when } (i < N) \text{ inc} \rightarrow \text{ZAEHL}[i+1] \\ | \text{when } (i > 0) \text{ dec} \rightarrow \text{ZAEHL}[i-1]) .$

# Prozess-Alphabet

**Definition:** Unter dem *Alphabet* eines Prozesses wird die Menge der Aktionen, die er ausführen kann, verstanden. Das Alphabet eines Prozesses lässt sich durch die Angabe  $+ \{ \dots \}$  um eine beliebige Menge erweitern.

**Beispiel:** Der wie folgt definierte Prozess `WRITER`

```
WRITER = (write[1] -> write[3] -> WRITER)
        + {write[0..3]}.
```

besitzt das Alphabet `write[0..3]`.



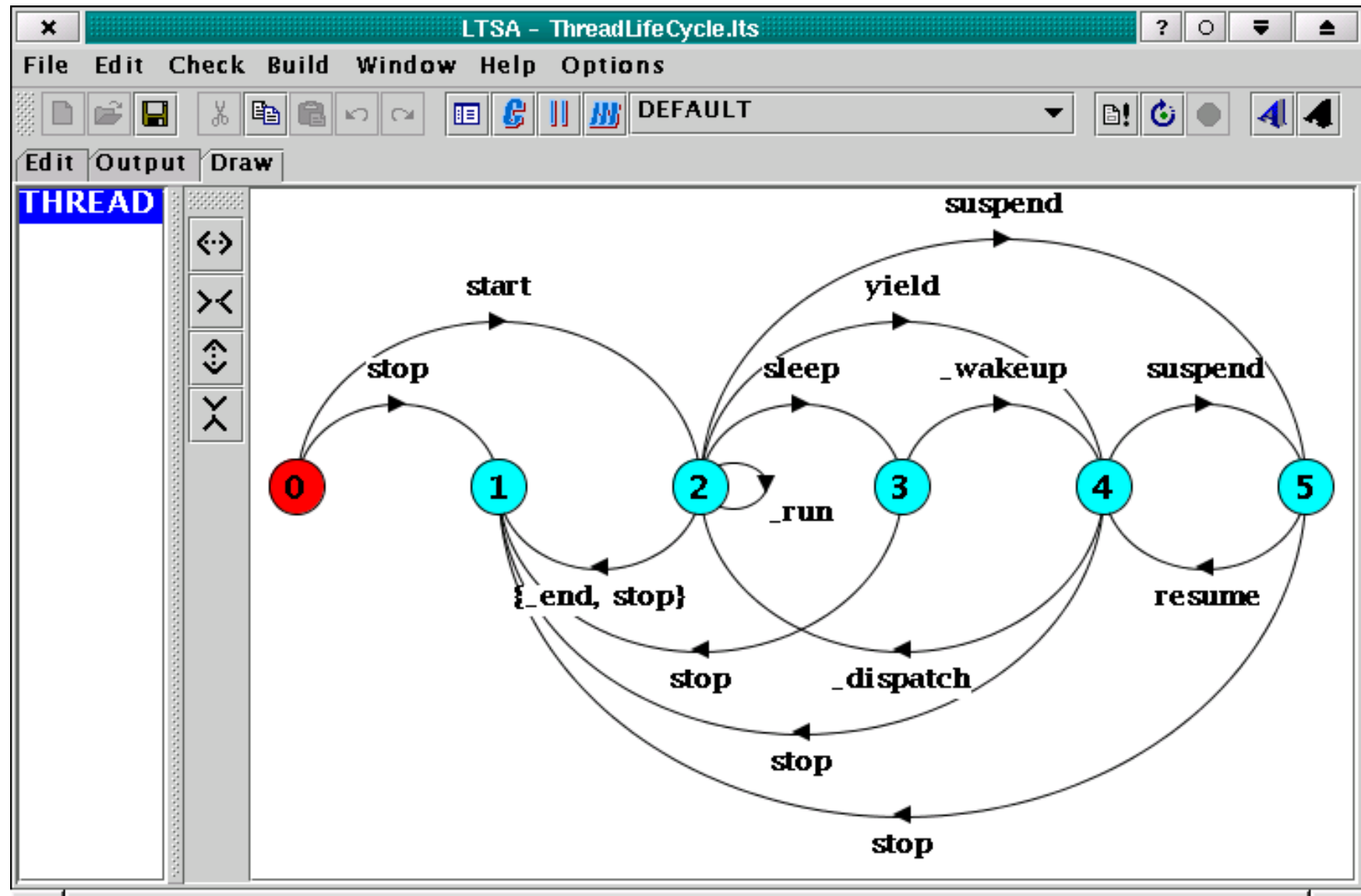
# Beispiel: Modellierung von Java-Prozessen

**Bemerkung:** Der Begriff „Prozess“ wird im Zusammenhang mit Modellen und der Begriff „Thread“ bei Java-Implementierungen von Prozessen gebraucht.

**Bemerkung:** Threads sind eine spezielle Art von Prozessen und können damit auch durch eine algebraische Beschreibung bzw. einen Zustandsgraphen modelliert werden.

# Lebenszyklus eines Java-Threads

```
THREAD    = CREATED,
CREATED   = (start -> RUNNING
             | stop  -> TERMINATED),
RUNNING   = (yield -> RUNNABLE
             | _run  -> RUNNING
             | {stop, _end} -> TERMINATED
             | suspend -> NON_RUNNABLE
             | sleep  -> SLEEPING),
RUNNABLE  = (suspend -> NON_RUNNABLE
             | _dispatch -> RUNNING
             | stop  -> TERMINATED),
NON_RUNNABLE = (resume -> RUNNABLE
               | stop  -> TERMINATED),
SLEEPING   = (_wakeUp -> RUNNABLE
             | stop  -> TERMINATED),
TERMINATED = STOP.
```



# Zusammenfassung (3)

- Ein Prozess kann entweder algebraisch durch eine FSP-Beschreibung oder grafisch durch einen Zustandsgraphen (LTS) modelliert werden,
- in FSP gibt es z. B. Präfix, Auswahl-, nichtdeterministische Auswahl- oder bewachte Aktionen,
- es wird nicht zwischen Eingabe- und Ausgabeaktionen unterschieden,
- Lebenszyklus eines Java-Threads ist auch als Prozess modellierbar.

## 3.2 Modellierung von Nebenläufigkeit

- Wie kann Nebenläufigkeit modelliert werden?
- Modellierung der Nebenläufigkeit soll unabhängig von der späteren Ausführung (Anzahl Prozessoren, Prozessorvergabestrategie, Prozessorgeschwindigkeit usw.) sein.

## Wiederholung:

- Modellierung ist „zeitlos“ (keine Annahme über Dauer der Aktionen bzw. Zustandsübergänge),
- jeder Prozess besteht aus einer Folge (atomarer) Aktionen

**Definition:** Seien  $P$  und  $Q$  beliebige Prozesse, dann wird durch die **parallele Komposition**  $||S = (P||Q)$  ein Prozess  $S$  definiert, der die nebenläufige Ausführung von  $P$  und  $Q$  beschreibt.

**Bemerkung:**

- Die parallele Komposition kann wieder durch einen Zustandsgraphen ausgedrückt werden,
- die **Traces** der parallelen Komposition  $(P||Q)$  bestehen aus sämtlichen Verzahnungsmöglichkeiten der Traces von  $P$  und  $Q$ .

# Beispiel: „Parallele Komposition“

Definition der nebenläufigen Prozesse Jucken und Reden:

Jucken = (**kratzen**  $\rightarrow$  STOP) .

Reden = (**denken**  $\rightarrow$  **reden**  $\rightarrow$  STOP) .

||JuckenReden = (Jucken || Reden) .

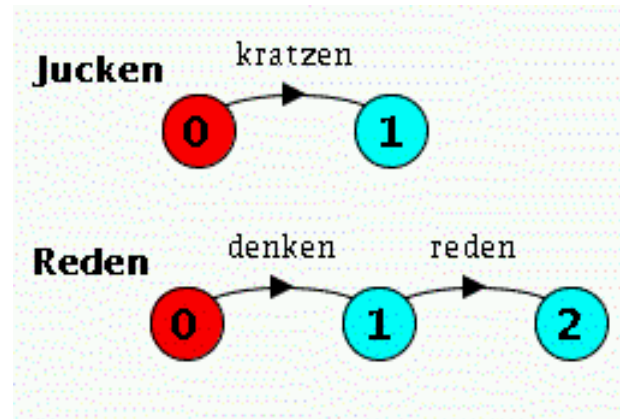
Traces von JuckenReden:

1. **denken**  $\rightarrow$  **reden**  $\rightarrow$  **kratzen**
2. **kratzen**  $\rightarrow$  **denken**  $\rightarrow$  **reden**
3. **denken**  $\rightarrow$  **kratzen**  $\rightarrow$  **reden**

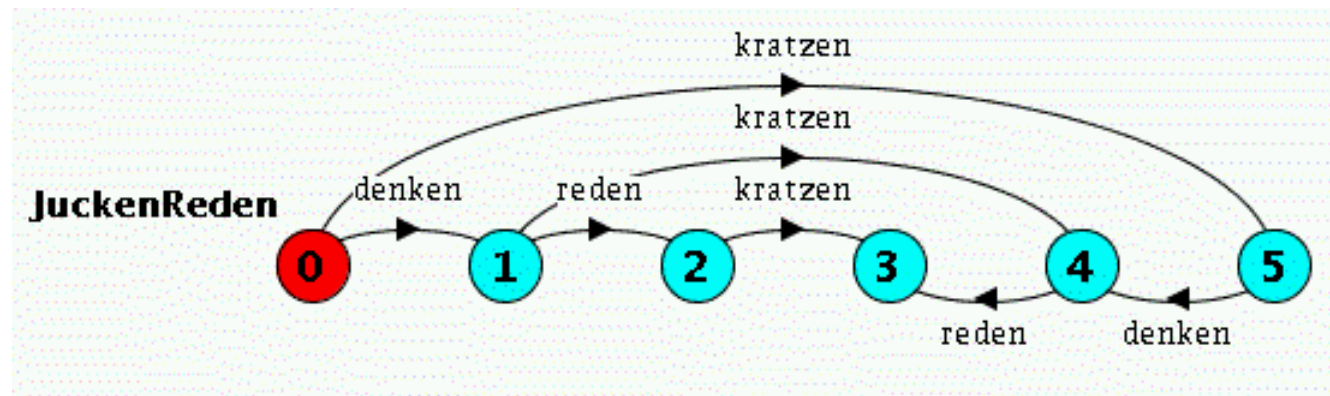


**Bemerkung:** In den Traces der parallelen Komposition ist die Reihenfolge der Aktionen jedes Einzelprozesses eingehalten.

## Zugehörige Zustandsgraphen:



Wie sieht der Zustandsgraph von JuckenReden aus?



# Variante (1)

Wiederholte Ausführung durch rekursive Prozessdefinition:

$\text{Jucken2} = (\text{kratzen} \rightarrow \text{Jucken2}) .$

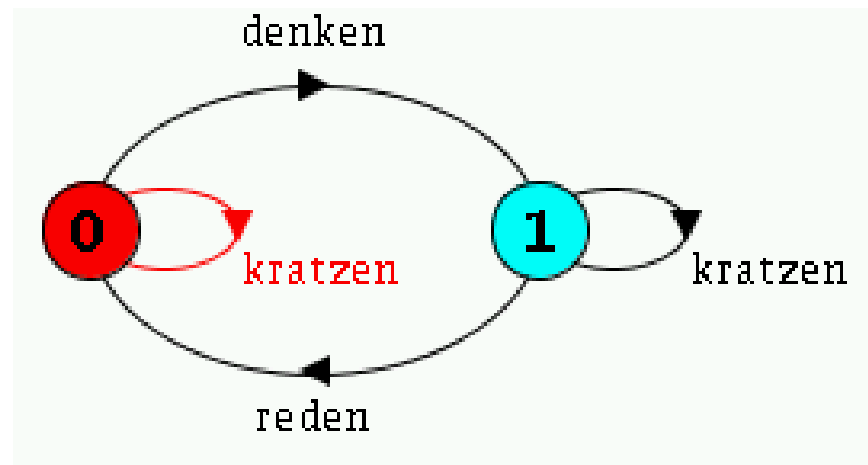
$\text{Reden2} = (\text{denken} \rightarrow \text{reden} \rightarrow \text{Reden2}) .$

$|| \text{JuckenReden2} = (\text{Jucken2} || \text{Reden2}) .$

Traces von  $\text{JuckenReden2}$ :

- $\text{denken} \rightarrow \text{reden} \rightarrow \text{kratzen} \rightarrow \text{kratzen} \rightarrow \text{denken} \rightarrow \dots$
- $\text{denken} \rightarrow \text{reden} \rightarrow \text{denken} \rightarrow \text{kratzen} \rightarrow \text{reden} \rightarrow \dots$
- $\dots$

## Zustandsgraph von JuckenReden2:



## Variante (2)

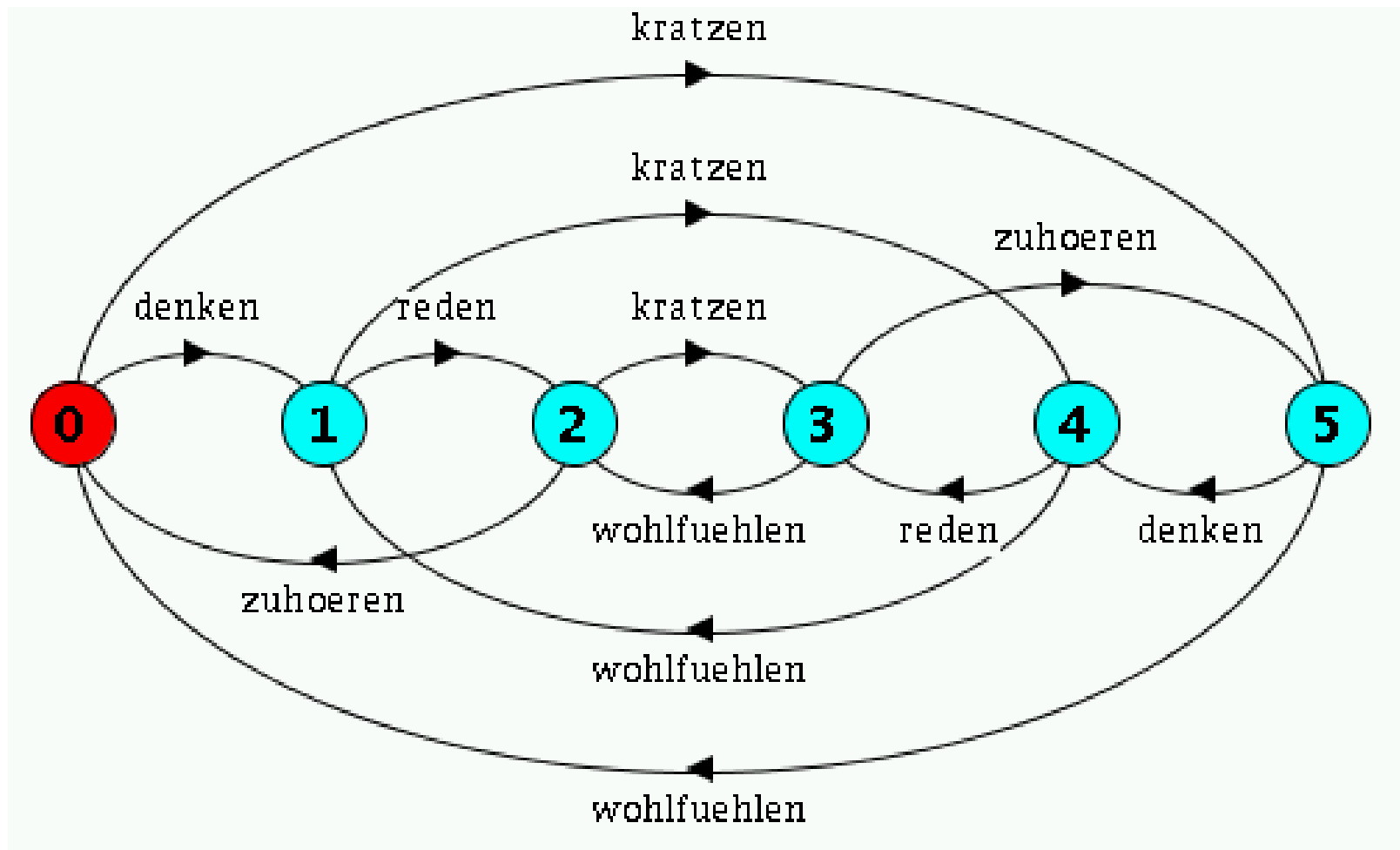
Mehr Aktionen:

```
Jucken3 = (kratzen -> wohlfuehlen -> Jucken3) .  
Reden3 = (denken -> reden -> zuhoeren -> Reden3) .  
||JuckenReden3 = (Jucken3 || Reden3) .
```

Traces von JuckenReden3:

?

## Zustandsgraph von JuckenReden3:



**Bemerkung:** Die parallele Komposition ist:

- kommutativ, d. h.  $(P||Q) = (Q||P)$ ,
- assoziativ, d. h.  $(P||(Q||R)) = ((P||Q)||R) = (P||Q||R)$ .

# Gemeinsame Aktionen

- Bisher liefen die Komponenten einer parallelen Komposition völlig unabhängig von einander,
- sie haben sich nicht beeinflusst (disjunkte Alphabete).

**Definition:** Besitzen die Alphabete der Prozesse einer parallelen Komposition eine nichtleere Schnittmenge, so werden die Elemente dieser Schnittmenge als **gemeinsame Aktionen** bezeichnet.



## **Bemerkung:**

- Gemeinsame Aktionen müssen synchron in allen beteiligten Prozessen ausgeführt,
- alle anderen Aktionen können beliebig verzahnt ausgeführt werden.

# Beispiel: „Gemeinsame Aktionen“

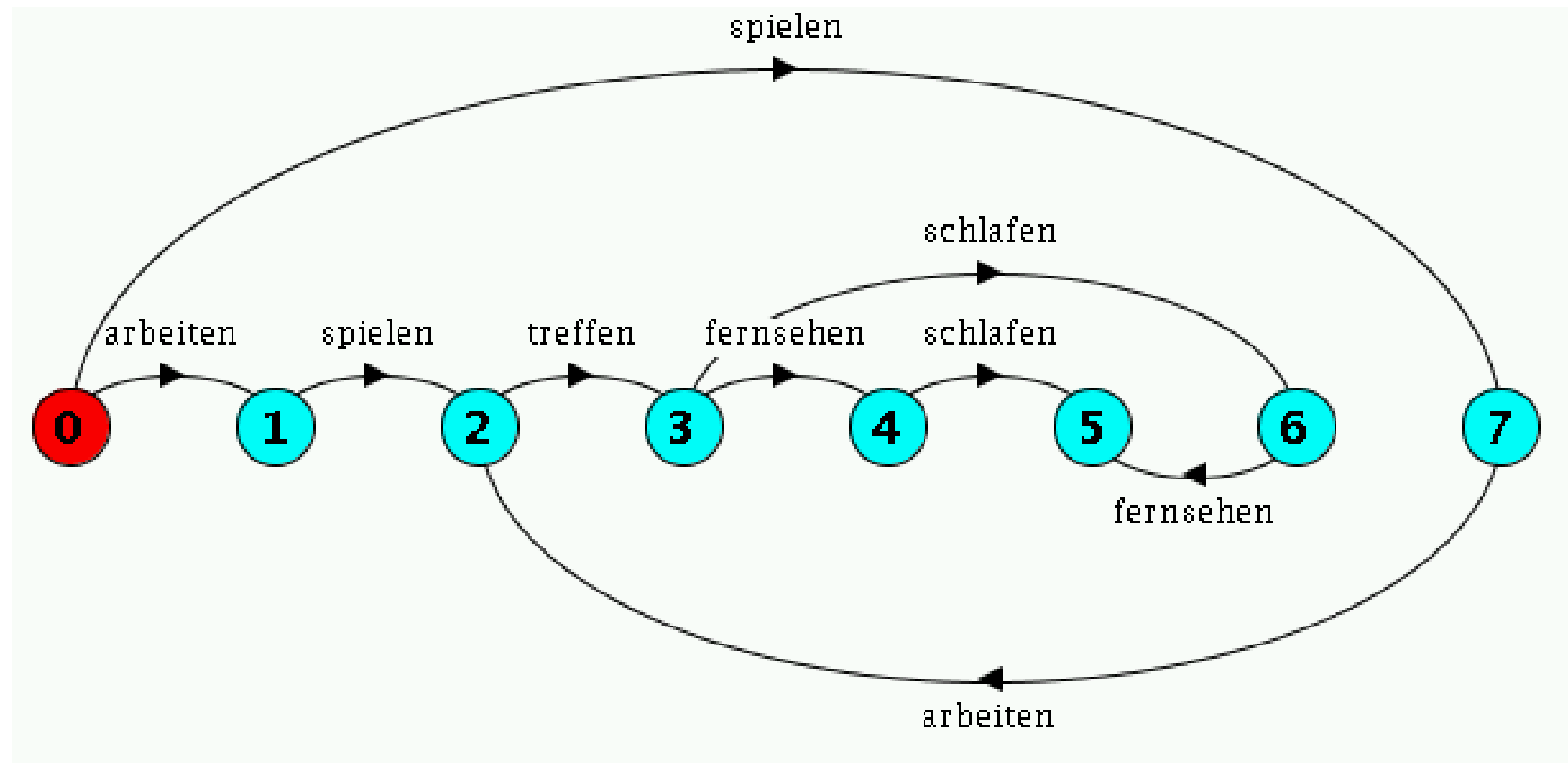
Definition der nebenläufigen Prozesse Vater und Sohn:

```
Vater = (arbeiten -> treffen -> fernsehen -> STOP) .  
Sohn  = (spielen -> treffen -> schlafen -> STOP) .  
||Tagesablauf = (Sohn || Vater) .
```

Traces von Tagesablauf:

1. arbeiten → spielen → treffen → fernsehen → schlafen
2. arbeiten → spielen → treffen → schlafen → fernsehen
3. spielen → arbeiten → treffen → fernsehen → schlafen
4. spielen → arbeiten → treffen → schlafen → fernsehen

## Zustandsgraph von Tagesablauf:



# „Produktions- und Verbrauchsprozess (1)“

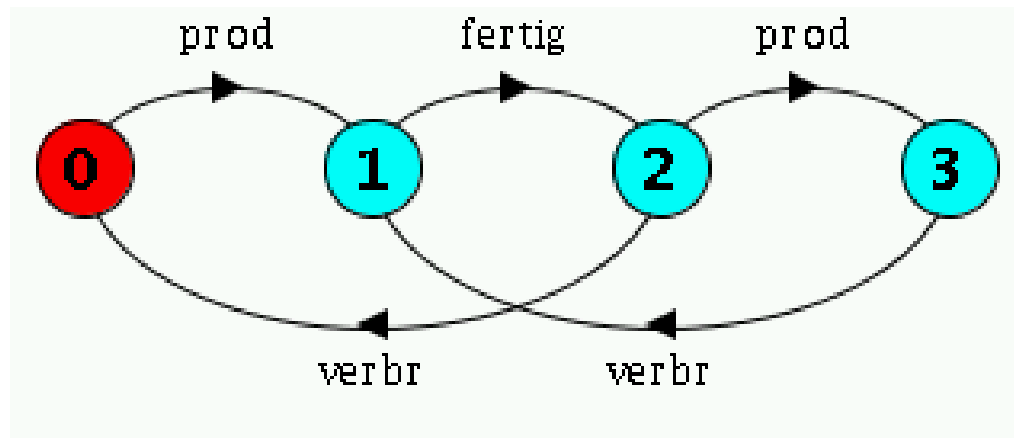
Prozessbeschreibung:

```
Prod = (prod -> fertig -> Prod) .
```

```
Verbr = (fertig -> verbr -> Verbr) .
```

```
||Konsum = (Prod || Verbr) .
```

Zustandsgraph:



## Prozessbeschreibung:

```
Prod = (prod -> fertig -> Prod) .  
Verbr = (fertig -> verbr -> Verbr) .  
||Konsum = (Prod || Verbr) .
```

## Traces:

- prod → fertig → verbr → prod → fertig → verbr → ...
- prod → fertig → prod → verbr → fertig → verbr → ...

## Eigenschaft dieser Implementierung:

Nachdem ein Teil produziert wurde, kann der Verbrauch des ersten Teils und die Produktion des zweiten Teils *gleichzeitig* erfolgen!

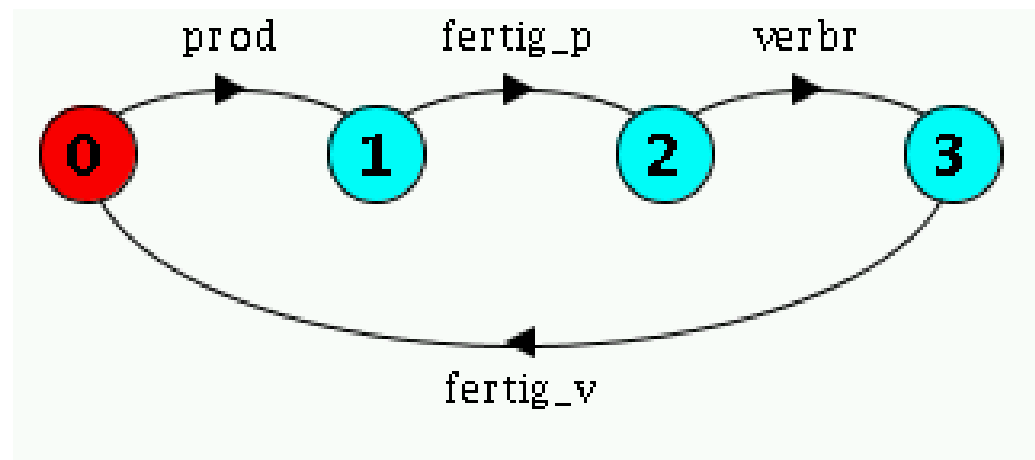
**Frage:** Wie kann die Produktion so lange angehalten werden, bis das Teil verbraucht wurde?

# „Produktions- und Verbrauchsprozess (2)“

Prozessbeschreibung:

```
Prod2 = (prod -> fertig_p -> fertig_v -> Prod2).  
Verbr2 = (fertig_p -> verbr -> fertig_v -> Verbr2).  
||Konsum2 = (Prod2 || Verbr2).
```

Zustandsgraph:



## Prozessbeschreibung:

```
Prod2 = (prod -> fertig_p -> fertig_v -> Prod2) .  
Verbr2 = (fertig_p -> verbr -> fertig_v -> Verbr2) .  
||Konsum2 = (Prod2 || Verbr2) .
```

## Trace:

- $\text{prod} \rightarrow \text{fertig\_p} \rightarrow \text{verbr} \rightarrow \text{fertig\_v} \rightarrow \text{prod} \rightarrow \text{fertig\_p} \rightarrow \dots$

## Eigenschaft dieser Implementierung:

Die Produktion eines neuen Teils kann erst dann beginnen, wenn das vorher produzierte Teil verbraucht wurde.

**Bemerkung:** Diese Art der Interaktion, in der eine Aktion durch eine zweite bestätigt wird, nennt man **Handshake**.



**Bemerkung:** Es können auch mehr als zwei Prozesse an einer gemeinsamen Aktion beteiligt sein (siehe nächstes Beispiel ...).

# „Fertigungssystem“

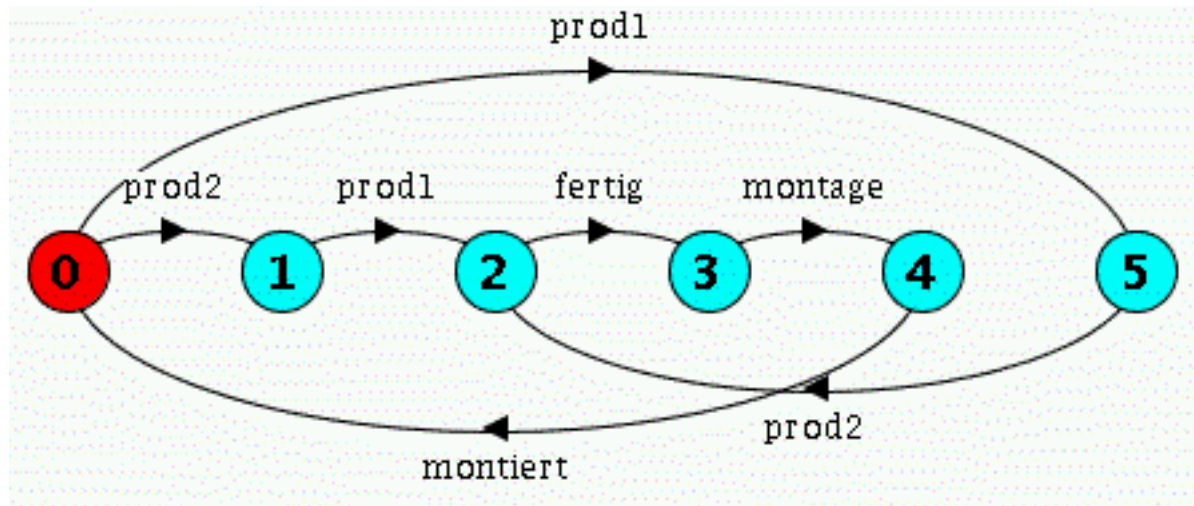
Fertigungssystem bestehend aus zwei Produktionsprozessen und einem Montageprozess:

`Prod1 = (prod1 -> fertig -> montiert -> Prod1).`

`Prod2 = (prod2 -> fertig -> montiert -> Prod2).`

`Montage = (fertig -> montage -> montiert -> Montage).`

`||Fertigung = (Prod1 || Prod2 || Montage).`



Anwendung des Assoziativgesetzes liefert eine andere, gleichwertige Darstellung:

```
Prod1 = (prod1 -> fertig -> montiert -> Prod1).
```

```
Prod2 = (prod2 -> fertig -> montiert -> Prod2).
```

```
||Produktion = (Prod1 || Prod2).
```

```
Montage = (fertig -> montage -> montiert -> Montage).
```

```
||Fertigung = (Produktion || Montage).
```

# Label-Präfix

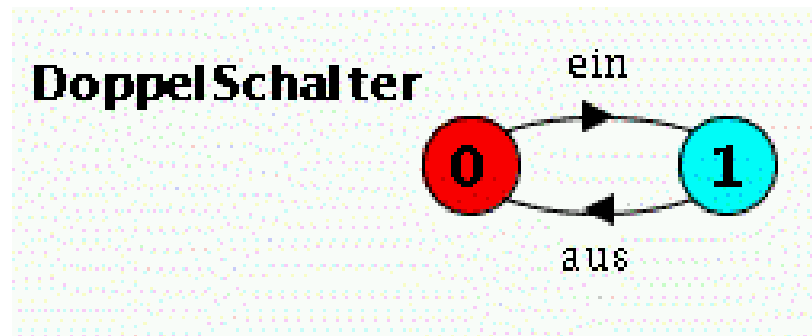
**Beispiel:** Modelliere ein System bestehend aus zwei unabhängigen Ein-/Ausschaltern der Art:

Schalter = (ein -> aus -> Schalter) .

# Doppelschalter: 1. Versuch

```
Schalter = (ein -> aus -> Schalter).  
||DoppelSchalter = (Schalter || Schalter).
```

liefert:



erfüllt die Anforderung nicht, weil die gleich benannten Aktionen beider Schalter nicht unterschieden werden können!

Soll die Definition eines Prozesses in einer parallelen Komposition mehrfach verwendet werden, müssen die Aktionen zwischen den Prozessen unterscheidbar sein.

**Definition:** Sei  $P$  ein beliebiger Prozess, so wird durch  $a : P$  die Bezeichnung jeder Aktion (d. h. jedes „Label“) im Alphabet von  $P$  mit dem Präfix  $a$ . versehen.

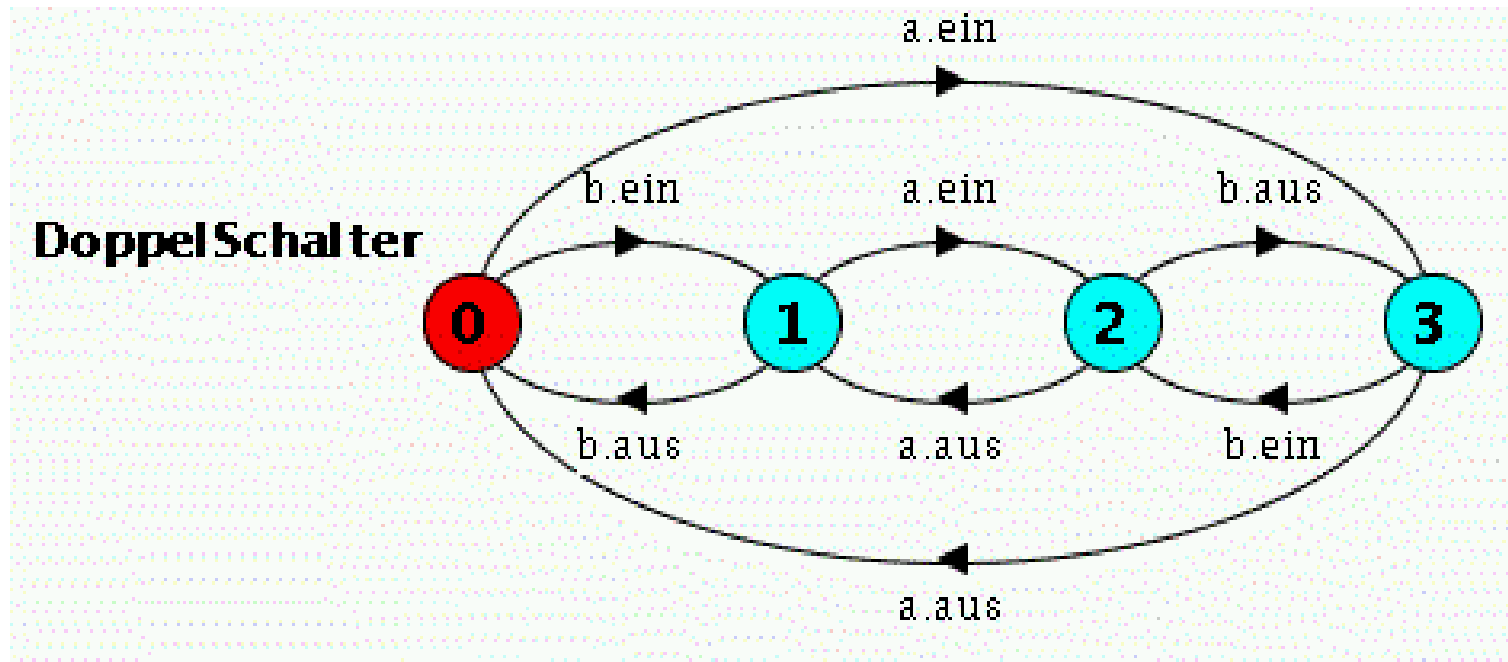
**Beispiel:** Ist  $P$  ein Prozess mit dem Alphabet  $A = \{eins, zwei, drei\}$ , so liefert  $a : P$  einen äquivalenten Prozess mit dem Alphabet  $a.A = \{a.eins, a.zwei, a.drei\}$ .

# Doppelschalter: 2. Versuch

Schalter = (ein -> aus -> Schalter).

||DoppelSchalter = (a:Schalter || b:Schalter).

liefert:

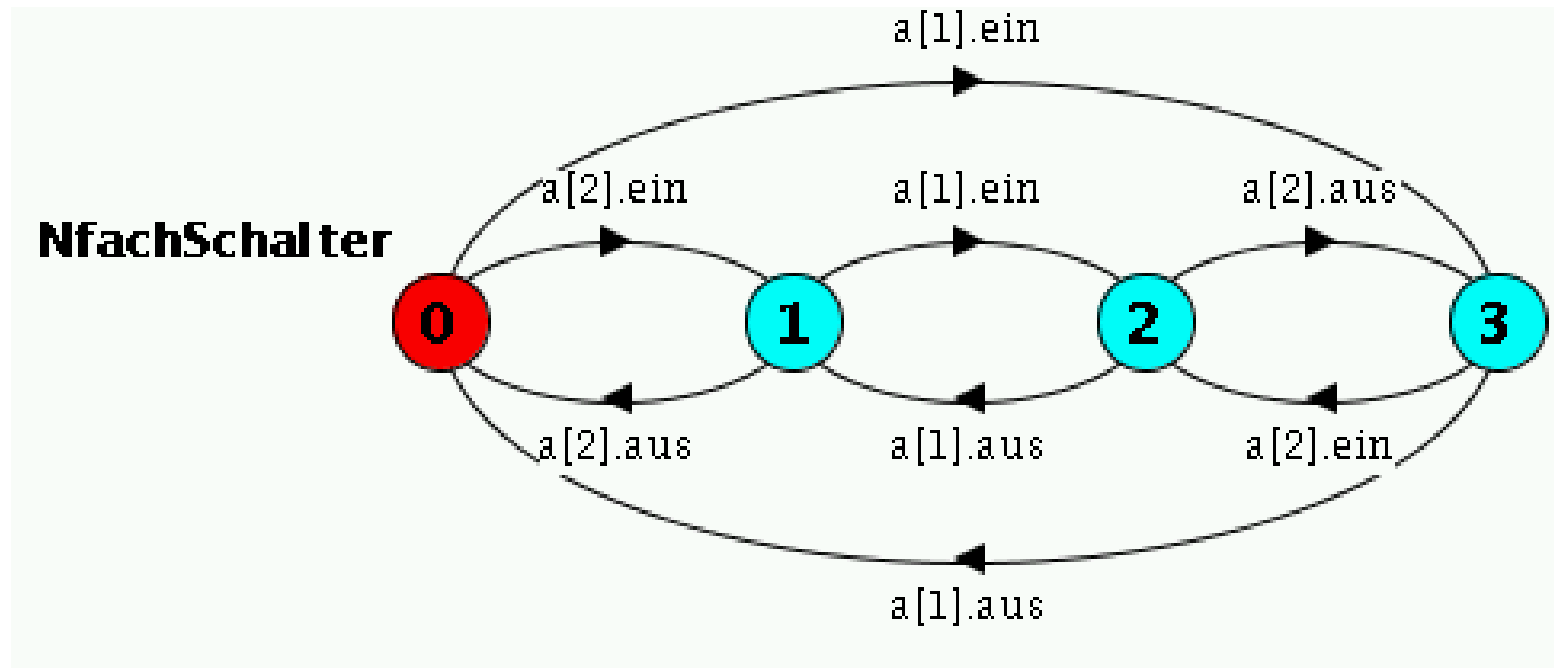


# Variante: N-fach-Schalter

Schalter = (ein -> aus -> Schalter) .

|| NfachSchalter(N=5) = (a[i:1..N]:Schalter) .

wobei  $a[i:1..N]:\text{Schalter}$  eine Kurzform für  
 $\text{forall } [i:1..N] \ a[i]:\text{Schalter}$  ist.

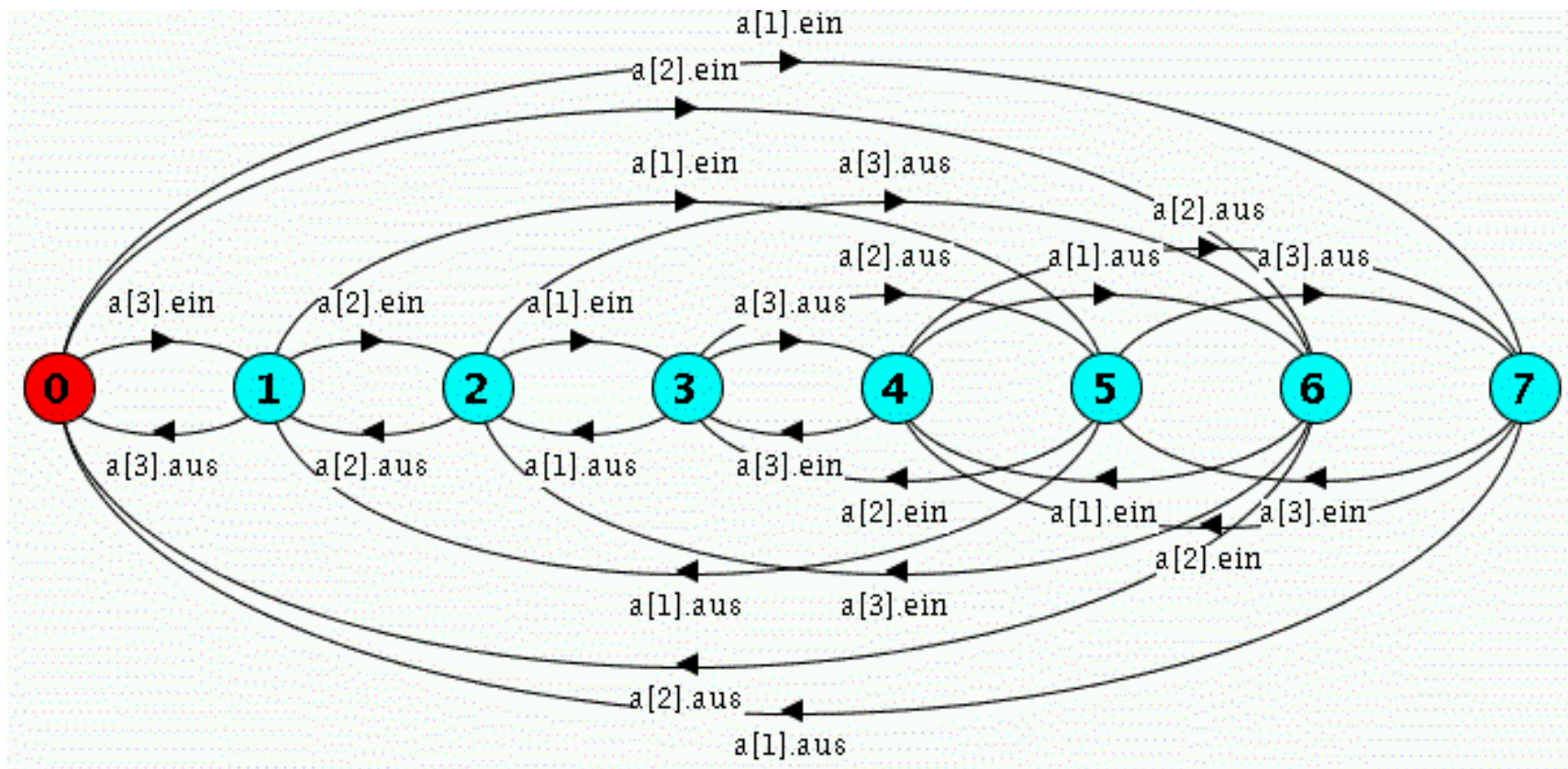




# Zur Kontrolle: 3-fach-Schalter

Schalter = (ein  $\rightarrow$  aus  $\rightarrow$  Schalter).

$||N\text{fachSchalter}(N=3) = (a[i:1..N]:\text{Schalter}).$



Es ist außerdem möglich, Prozesse mit einer Menge von Präfixen zu versehen:

**Definition:** Sei  $P$  ein beliebiger Prozess, so wird durch  $\{a_1, \dots, a_n\} :: P$  ein Prozess definiert, in dem jeder Bezeichner einer Aktion (Label)  $l$  im Alphabet von  $P$  durch die Menge  $\{a_1.l, \dots, a_n.l\}$  und jede Aktion  $(y \rightarrow Q)$  durch die Menge  $(\{a_1.y, \dots, a_n.y\} \rightarrow Q)$  ersetzt wird.

# Beispiel: „Gegenseitiger Ausschluss“

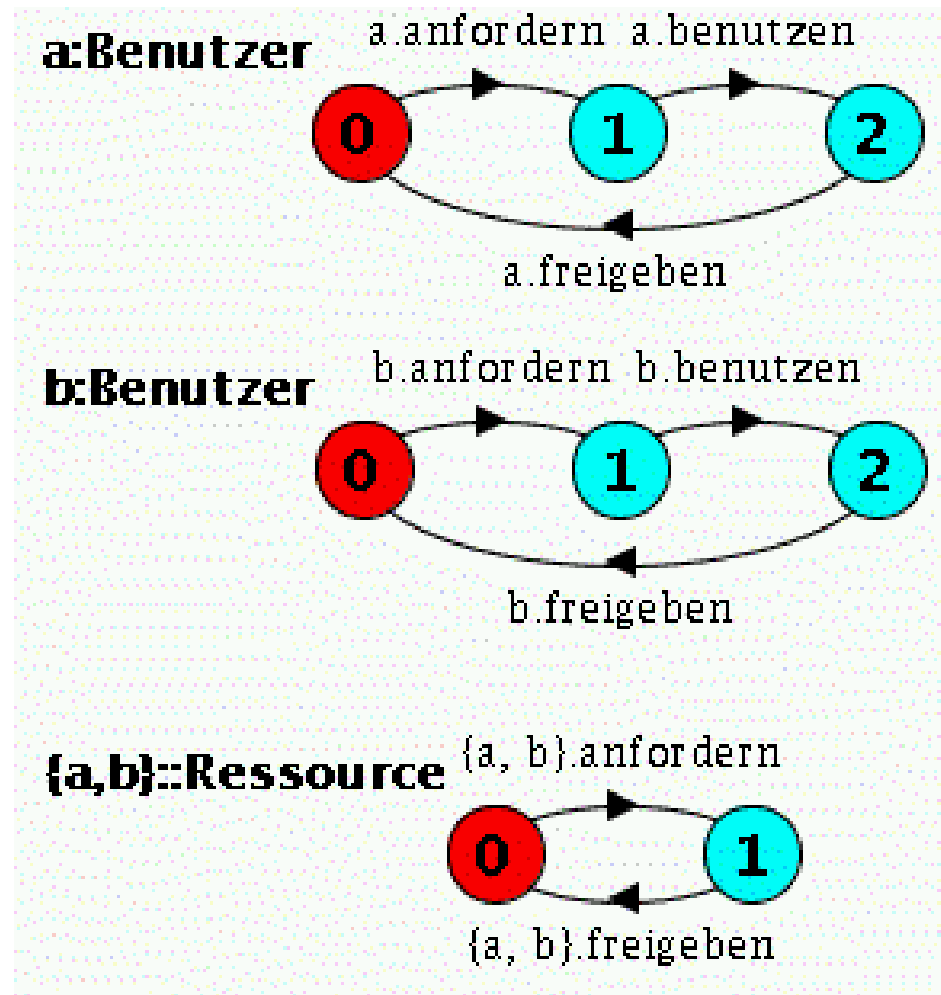
Zwei Benutzer teilen sich eine Ressource; zu jeder Zeit hat maximal ein Benutzer die Kontrolle über die Ressource.

```
Benutzer = (anfordern -> benutzen -> freigeben  
           -> Benutzer) .
```

```
Ressource = (anfordern -> freigeben -> Ressource) .
```

```
||Ausschluss = (a:Benutzer || b: Benutzer  
               || {a,b}::Ressource) .
```

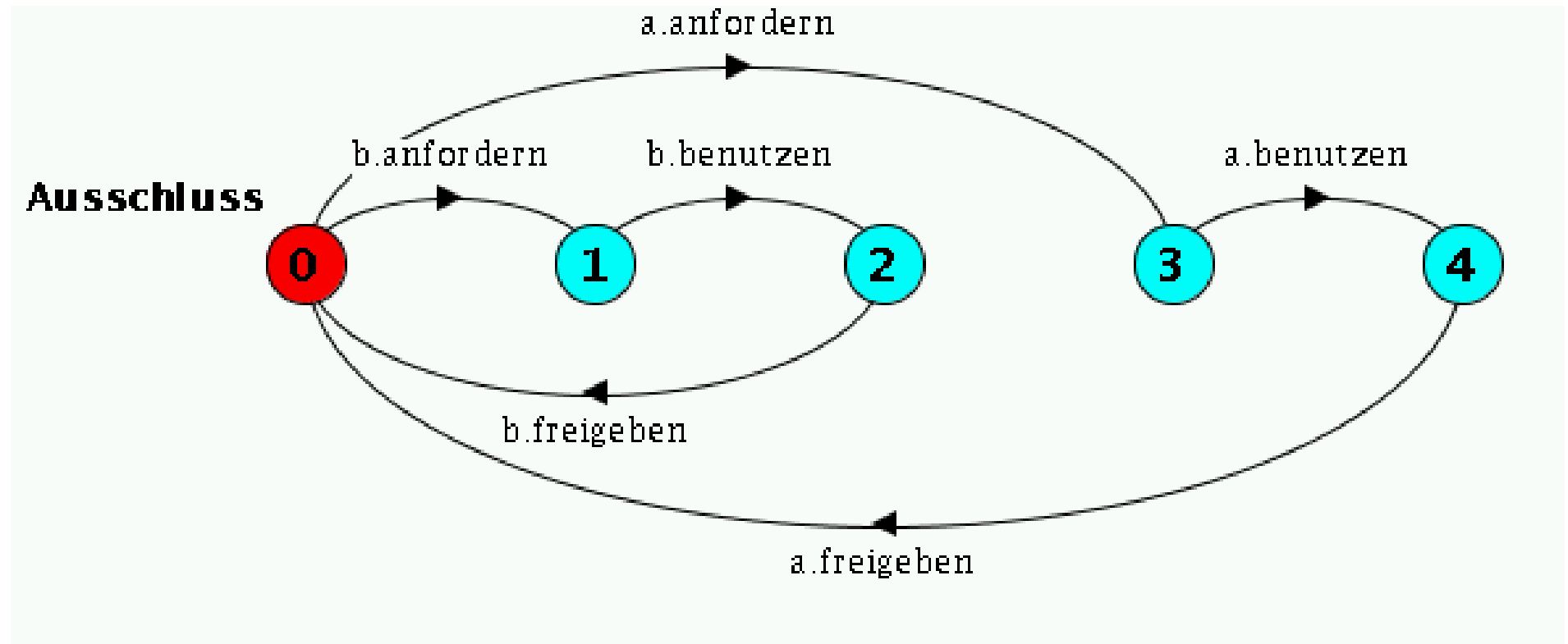
Ausschluss ist die parallele Komposition aus drei nebenläufigen Prozessen:



- Eine Ressource darf nur von dem Benutzer freigegeben werden, der sie auch angefordert hat;  
im Prozess  $\{a, b\} :: \text{Ressource}$  ist auch folgender Trace erlaubt:  $a.\text{anfordern} \rightarrow b.\text{freigeben} \rightarrow \dots$

**Frage:** Arbeitet der Gesamtprozess trotzdem korrekt?  
Warum?

**Antwort:** Betrachte das Zustandsdiagramm:



# Client-/Server-Prozess: 1. Versuch

**Aufgabe:** Es ist das Verhalten einer verteilten Client-/Server-Architektur (jeweils ein Client und ein Server) zu modellieren.

```
Client = (call -> wait -> continue -> Client).  
Server = (request -> service -> reply -> Server).  
||ClientServer = (Client || Server).
```

**Falsch:** `Client` und `Server` besitzen disjunkte Alphabete, laufen also unabhängig voneinander.

**Frage:** Wie lassen sich die Prozesse kombinieren, *ohne* deren Definition zu ändern?

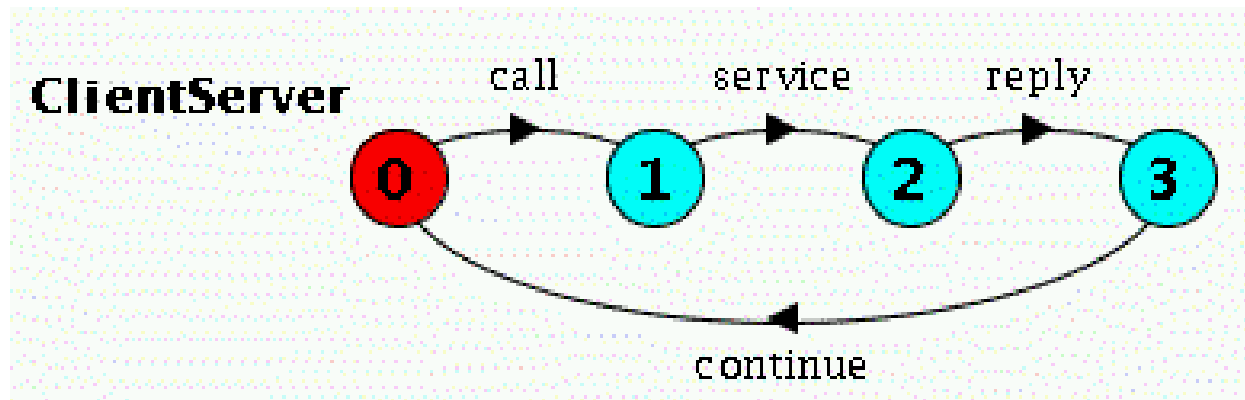
# „Relabelling“

**Definition:** Sei  $P$  ein beliebiger Prozess, so wird durch  $P/\{neu_1/alt_1, \dots, neu_n/alt_n\}$  ein **Relabelling** der Aktionen von  $P$  durchgeführt, in dem in  $P$  die alten Bezeichner  $alt_1, \dots, alt_n$  durch die neuen  $neu_1, \dots, neu_n$  ersetzt werden.



# Client-/Server-Prozess: 2. Versuch

```
Client = (call -> wait -> continue -> Client).  
Server = (request -> service -> reply -> Server).  
||ClientServer = (Client || Server)  
                / {call/request, reply/wait}.
```

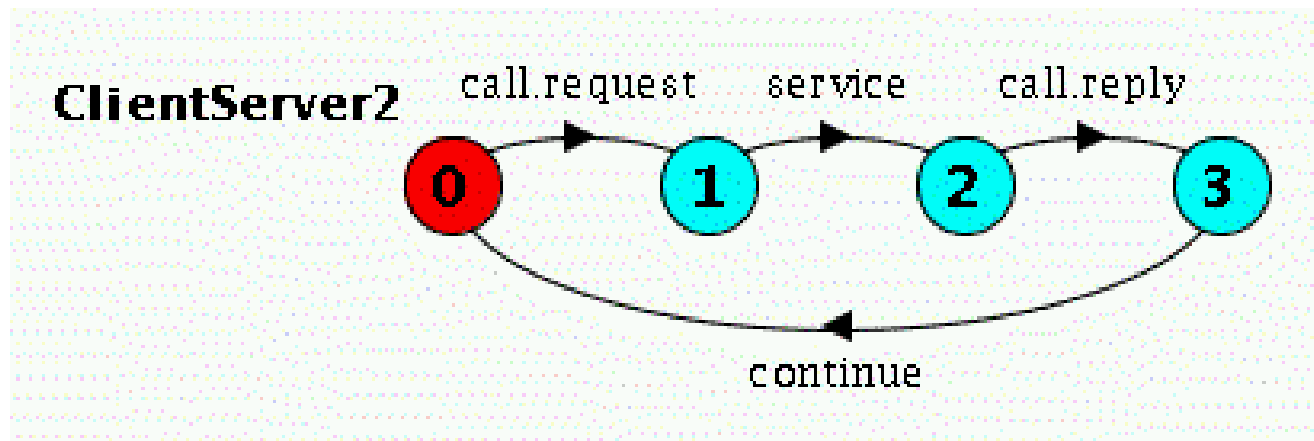


# Client-/Server-Prozess (Variante)

```
Client2 = (call.request -> call.reply -> continue  
          -> Client2) .
```

```
Server2 = (accept.request -> service -> accept.reply  
          -> Server2) .
```

```
||ClientServer2 = (Client2 || Server2)  
                  / {call/accept}.
```



# Aufgabe

Wie lässt sich eines der vorigen Client-/Server-Modelle so modifizieren, dass mehr als ein Client den Dienst des Servers abrufen kann?

# Verborgene Aktionen

**Definition:** Sei  $P$  ein beliebiger Prozess, so wird durch  $P \setminus \{a_1, \dots, a_n\}$  ein Prozess definiert, in dem die Aktionen  $\{a_1, \dots, a_n\}$  aus dem Alphabet von  $P$  verborgen werden („\“ heißt **Hiding-Operator**). Verborgene Aktionen werden mit  $\tau$  markiert und können *nicht* von nebenläufigen Prozessen gemeinsam benutzt werden.

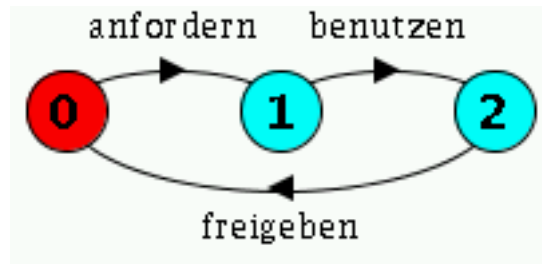
**Bemerkung:** Sinn und Zweck des Verbergens von Aktionen:

- Entfernen von Zuständen, die außerhalb des Prozesses nicht betrachtet werden sollen,
- Reduktion der Komplexität (Zustandsanzahl) in zusammengesetzten Prozessen.

**Definition:** Sei  $P$  ein beliebiger Prozess, so wird durch  $P@{a_1, \dots, a_n}$  ein Prozess definiert, in dem alle Aktionen bis auf  $\{a_1, \dots, a_n\}$  verborgen werden (**Interface-Operator**).

**Beispiel:** Sei folgender Prozess gegeben:

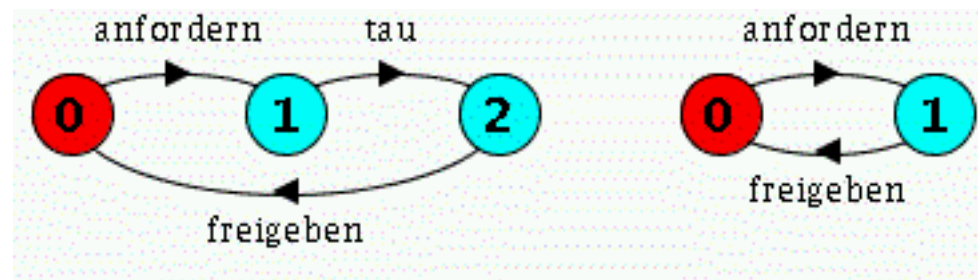
$\text{User} = (\text{anfordern} \rightarrow \text{benutzen} \rightarrow \text{freigeben} \rightarrow \text{User}).$



So sind die wie folgt definierten Prozesse identisch:

$\text{User} \setminus \{\text{benutzen}\}$

$\text{User} @ \{\text{anfordern}, \text{freigeben}\}$

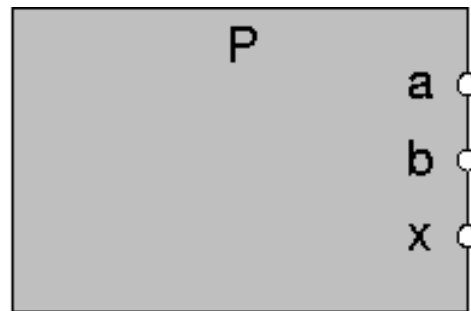


# Strukturdiagramme

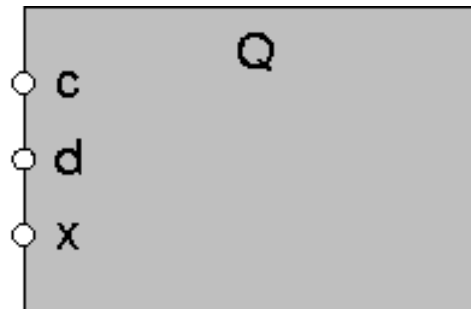
- Durch **Zustandsdiagramme** lässt sich das *dynamische* Verhalten eines Prozesses beschreiben.
- Im **Zustandsdiagramm** einer parallelen Komposition geht die Struktur der Komponenten verloren.
- Durch **Strukturdiagramme** lässt sich die *Struktur* einer parallelen Komposition beschreiben.

# Strukturdiagramme (1)

Prozess  $P$  mit Alphabet  $\{a, b, x\}$ :



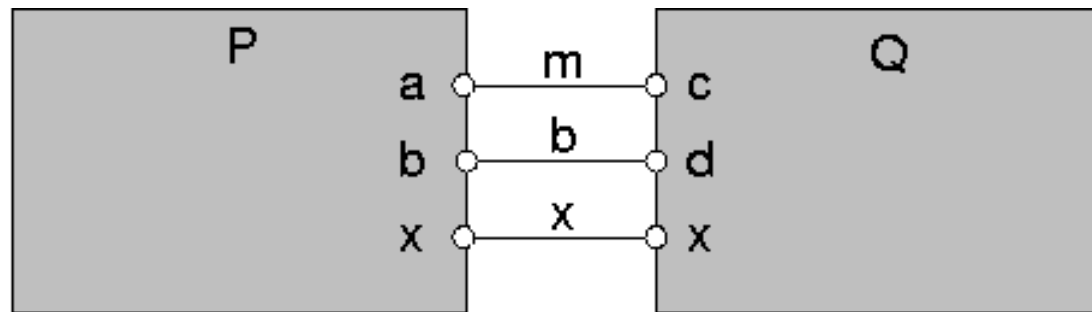
Prozess  $Q$  mit Alphabet  $\{c, d, x\}$ :





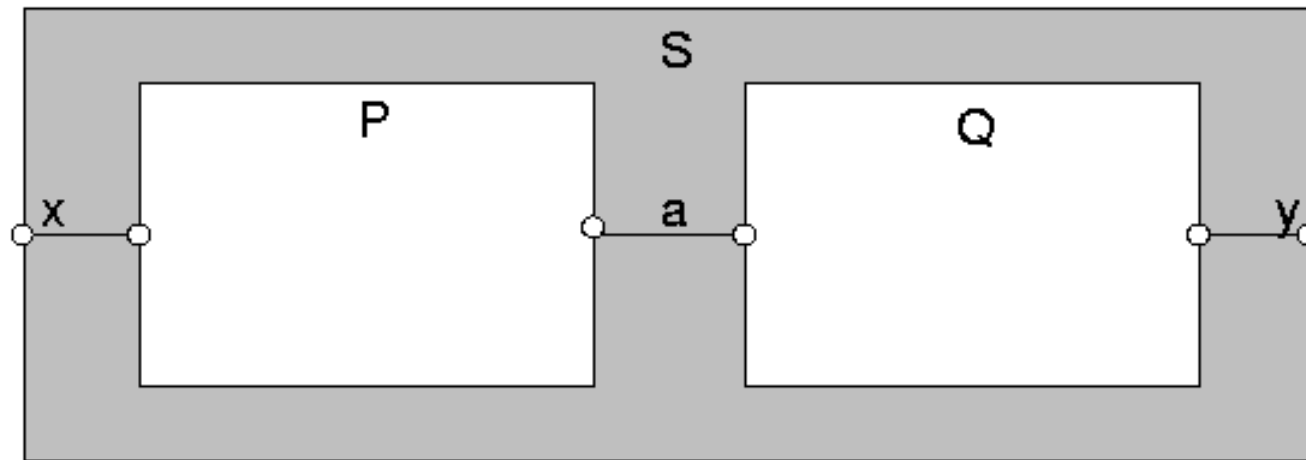
## Strukturdiagramme (2)

parallele Komposition  $(P||Q)/\{m/a, m/c, b/d\}$ :



# Strukturdiagramme (3)

parallele Komposition  $||S = (P||Q)@ \{x, y\}.$



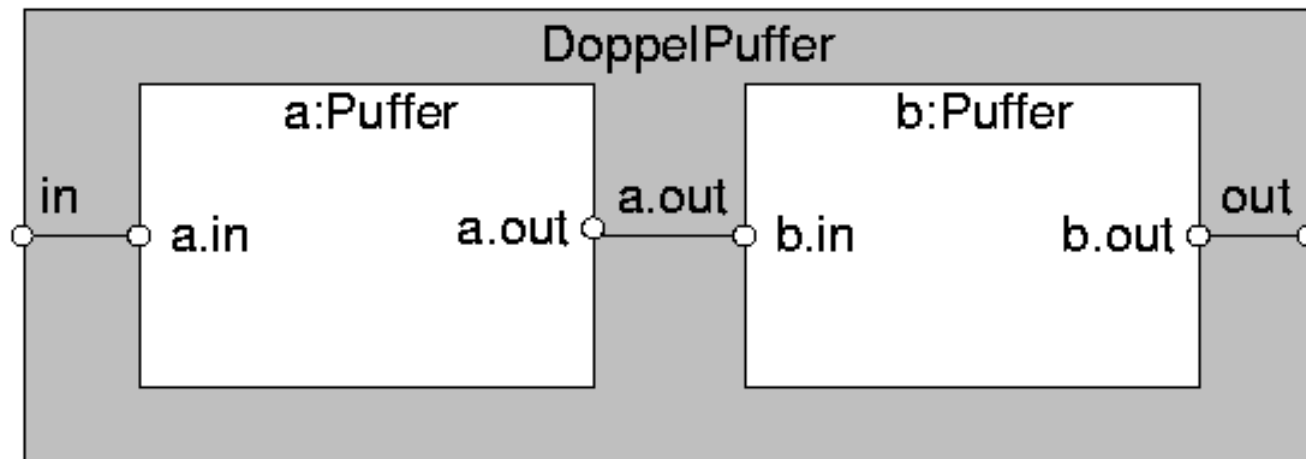
# Strukturdiagramme (4)

Doppelpuffer:

```
range T=0..3
```

```
Puffer = (in[i:T] -> out[i] -> Puffer).
```

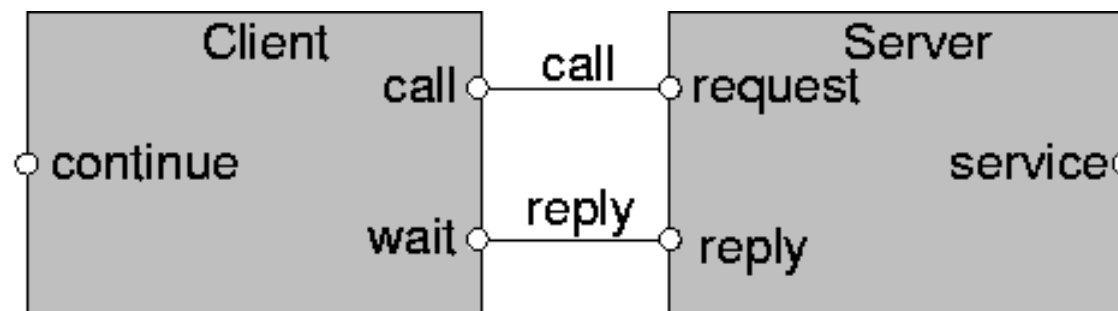
```
||DoppelPuffer = (a:Puffer || b:Puffer)  
                / {a.out/b.in, in/a.in, out/b.out}.
```



# Strukturdiagramme (5)

## Client-/Server-Prozess:

```
Client = (call -> wait -> continue -> Client).  
Server = (request -> service -> reply -> Server).  
||ClientServer = (Client || Server)  
                /{call/request, reply/wait}.
```



# Zusammenfassung (3)

- parallele Komposition von Prozessen,
- gemeinsame Aktionen einer parallelen Komposition,
- Handshake
- Label-Präfix, Relabelling, verborgene Aktionen
- Strukturdiagramme

# 4 Nebenläufigkeit in Java

## Bisher:

- Nebenläufigkeit wurde bisher nur im Modell betrachtet (parallele Komposition in FSP),
- Interaktion zwischen nebenläufigen Prozessen erfolgte durch gemeinsame (unteilbare) Aktionen,
- Traces der parallelen Komposition bestehen aus allen möglichen Kombinationen der nebenläufigen und nicht gemeinsamen Aktionen.

## Jetzt:

- Formulieren von Nebenläufigkeit in Java,
- Fragen:
  1. Was entspricht der parallelen Komposition?
  2. Was entspricht den unteilbaren gemeinsamen Aktionen?

# Beispiel: „Schlossgarten“

Folgende Aufgabenstellung soll in Java durch nebenläufige Prozesse/Threads realisiert werden:

- Ein Schlossgarten kann durch eines von zwei Toren betreten werden
- die Tore sind unabhängig voneinander (jedes Tor wird als Thread realisiert; Vereinfachung: Durch jedes Tor kommen genau  $N$  Besucher)
- es gibt einen zentralen Zähler: Er wird durch die Tore hochgezählt, sobald ein Besucher ein Tor durchquert
- jedes Tor besitzt auch einen lokalen Zähler



# Zählerimplementierung: 1. Versuch (a)

Wie könnte die Implementierung aussehen?

```
public class Counter {  
    private int value; // aktueller Zaehlerstand  
    public Counter() { value = 0; }  
  
    public void incrementValue() {  
        int temp = value + 1;  
        value = temp;  
        // Kurzform "value++" wird intern auch so aufgelöst  
    }  
  
    public int getValue() {return value;}  
}
```

**Bemerkung:** Die Anweisungen nebenläufiger Prozesse können beliebig verzahnt ablaufen!

**Bemerkung:** Die Methode `incrementValue()` ist **nicht unteilbar**, d.h. es muss an jeder Stelle mit einer Unterbrechung gerechnet werden (Prozesswechsel, Warten auf Speicherzugriff, ...).

**Frage:** Was kann dadurch passieren? → siehe **Eingangsbeispiel**, bzw. später

**Modifikation:** Die Wahrscheinlichkeit einer Unterbrechung innerhalb von `incrementValue()` soll durch eine entsprechende Erweiterung des Codes erhöht werden; die sonstige Logik der Implementierung wird durch diese Erweiterung nicht verändert!

**Idee:** Mit einer Wahrscheinlichkeit  $> 0$  gibt der Thread den Prozessor ab oder „legt sich schlafen“:

Aufruf von `Thread.yield()` oder `Thread.sleep(n)`

# Zählerimplementierung: 1. Versuch (b)

```
public class Counter {  
    private int value; // aktueller Zaehlerstand  
    public Counter() { this.value = 0;}  
  
    public void incrementValue() {  
        int temp = value + 1;  
        { // eventuell Prozesswechsel vornehmen  
            try {  
                if (Math.random() < 0.5)  
                    Thread.yield(); //Thread.sleep(1);  
            } catch (Exception e) {}  
        }  
        value = temp;  
    }  
  
    public int getValue() {return value;}  
}
```

# Implementierung des Tores

```
public class Gate extends Thread {  
    private Counter globalCounter;  
    private Counter localCounter;  
  
    public Gate (String name, Counter globalCounter) {  
        // Konstruktor der Oberklasse Thread aufrufen  
        super("Tor \" + name + "\");  
        // Zaehler zuweisen  
        this.globalCounter = globalCounter;  
        this.localCounter = new Counter();  
    }  
  
    public int getLocalValue() {  
        return localCounter.getValue();  
    }  
}
```

```
public void run() {  
    System.out.println (getName() + ": oeffne...");  
    for (int i = 0; i < MAX_VISITORS; i++) {  
        localCounter.incrementValue();  
        globalCounter.incrementValue();  
    }  
    System.out.println (getName() + ": schliesse.");  
}  
}
```

# Mainmethode

```
public static void main (String[] args) {  
    Counter firstCounter = new Counter();  
    Gate east = new Gate("Osten", firstCounter);  
    Gate west = new Gate("Westen", firstCounter);  
    east.start();  
    west.start();  
    // jetzt laufen drei Threads!  
  
    // auf Beendigung der Gate-Threads warten  
    try {  
        east.join();  
        west.join();  
    } catch (Exception e) {  
        System.out.println(">>> Exception: " + e);  
    }  
}
```

```
// Zaehlerstaende ausgeben
System.out.println();
System.out.println(east.getName() + ": lokal " +
    east.getLocalValue() + " Besucher gezaehlt.");
System.out.println(west.getName() + ": lokal " +
    west.getLocalValue() + " Besucher gezaehlt.");

System.out.println("Zaehler: global " +
    firstCounter.getValue() + " Besucher gezaehlt.");
}
```



## Ausgabe:

```
> java Garden  
Tor "Osten": oeffne...  
Tor "Westen": oeffne...  
Tor "Osten": schliesse.  
Tor "Westen": schliesse.  
  
Tor "Osten": lokal 20 Besucher gezaehlt.  
Tor "Westen": lokal 20 Besucher gezaehlt.  
Zaehler: global 28 Besucher gezaehlt.
```

**Fakt:** Offensichtlich gehen bei dieser Implementierung der Methode `incrementValue()` Zählimpulse verloren!

**Warum?**

## Antwort:

Methodenaufrufe in Java sind nicht atomar, d.h. die Ausführung kann *jederzeit* unterbrochen und zu einem späteren Zeitpunkt fortgesetzt werden. Zwischenzeitlich kann ein anderer Prozess den Wert des gemeinsamen Zählers verändern.

Ein möglicher Trace, der zu einem falschen Zählerstand führt (`value` innerhalb des **gemeinsamen Zählers** habe den Wert  $n$ ):

1. „Tor Osten“ führt `incrementValue()` aus, liest `value` und weist seiner lokalen Variablen `temp` den Wert  $n + 1$  zu.
2. „Tor Osten“ führt `yield()` bzw. `sleep()` aus.

3. „Tor Westen“ führt `incrementValue()` aus, liest `value` und weist seiner lokalen Variablen `temp` den Wert  $n + 1$  zu.
4. „Tor Westen“ weist `value` den Wert seiner lokalen Variablen `temp` zu ( $n + 1$ ) und beendet den Aufruf von `incrementValue()`.
5. „Tor Osten“ weist `value` den Wert seiner lokalen Variablen `temp` zu (ebenfalls  $n + 1$ ) und beendet den Aufruf von `incrementValue()`.

Innerhalb des Zählers besitzt `value` jetzt den Wert  $n + 1$ , obwohl **zwei Mal** `incrementValue()` aufgerufen wurde!

Es sind auch Traces möglich, die zu korrekten Zählerständen führen!

**Bezeichnung:** Nicht vorhersagbare Zustände, die durch den unkontrollierten Zugriff auf eine gemeinsame Variable entstehen können, werden als „**Interferenz**“ oder „**Race Hazard**“ bezeichnet.

**Frage:** Wie lassen sich Interferenzen oder Race Hazards vermeiden?

**Antwort:** Stelle sicher, dass nur ein Prozess zurzeit eine Anweisungsfolge ausführt, die eine gemeinsame Variable manipuliert (**gegenseitiger Ausschluss**). Diese Anweisungsfolge kann dann als unteilbar (**atomar**) angesehen werden.

**Frage:** Wie geht das in Java?

## 4.1 Gegenseitiger Ausschluss in Java

In Java lässt sich **gegenseitiger Ausschluss** mit dem Schlüsselwort `synchronized` realisieren:

- Jeder Instanz eines Java-Objektes ist ein sogenannter „Lock“ (dt. „Schloss“ oder „Riegel“) zugeordnet,
- durch `synchronized (obj) {Anweisungsblock}` wird sichergestellt, dass das Laufzeitsystem vor der Ausführung des Anweisungsblocks prüft, ob der Lock des Objektes `obj` frei ist,
- ist der Lock frei, wird der Lock gesperrt (das Objekt für andere Threads „verriegelt“), der Anweisungsblock ausgeführt und anschließend der Lock wieder freigegeben,

- ist der Lock von `obj` schon durch einen anderen Thread gesperrt, wird der prüfende Thread solange blockiert, bis der Lock wieder frei ist („Ausschluss-Wartemenge“),
- es können mehrere Threads auf die Freigabe eines Locks warten; in diesem Fall wird beim Freiwerden ein Thread *nichtdeterministisch* ausgewählt, der als nächster den Lock sperren darf.

# Gegenseitiger Ausschluss bei Methoden

Der Methodendeklaration kann das Schlüsselwort `synchronized` vorangestellt werden:

```
public synchronized void foo() {  
    ... Methodenrumpf  
}
```

Von den mit `synchronized` gekennzeichneten Methoden eines Objektes kann jeweils nur *eine* zurzeit ausgeführt werden, d. h. der Lock des Objektes wird hier implizit verwendet!

entspricht: `synchronized (this) {Methodenrumpf}`

# Bemerkung

Auch **Klassen**methoden können als `synchronized` gekennzeichnet werden:

```
public static synchronized void foo() { ... }
```

**Frage:** Wo ist hier der zugehörige Lock angesiedelt?

**Antwort:** Von der Java-VM wird beim Laden jeder Klasse automatisch ein zugehöriges `Class`-Objekt angelegt<sup>6</sup>.

Mit dieser Instanz wird der Lock von Klassenmethoden assoziiert!

---

<sup>6</sup> Solche Objekte können nicht vom Benutzer erzeugt werden!



## Zählerimplementierung: 2. Versuch

Die Klasse `Counter` wird durch eine neue Klasse `GoodCounter` erweitert, in der die fehleranfällige Methode `incrementValue()` durch eine synchronisierte Methode überschrieben wird:

```
class GoodCounter extends Counter {  
    public synchronized void incrementValue() {  
        super.incrementValue();  
    }  
    public synchronized int getValue() {  
        return super.getValue();  
    }  
}
```

## Dieser Zähler liefert die korrekte Ausgabe:

```
> java Garden  
Tor "Osten": oeffne...  
Tor "Westen": oeffne...  
Tor "Osten": schliesse.  
Tor "Westen": schliesse.  
  
Tor "Osten": lokal 20 Besucher gezaehlt.  
Tor "Westen": lokal 20 Besucher gezaehlt.  
Guter Zaehler: global 40 Besucher gezaehlt.
```

# Rekursive Locks

```
class Counter {  
    private int value;  
    ...  
    public synchronized void increment() {value+=1;}  
  
    public synchronized void increment(int n) {  
        if (n > 0) {  
            increment();    // einmal hochzaehlen  
            increment(n-1); // rekursiv noch (n-1)-mal  
        }  
    }  
}
```

Hält ein Thread einen Lock, kann er ohne blockiert zu werden, direkt oder indirekt andere Methoden/Anweisungen ausführen, die *denselben* Lock erfordern.

# Modellierung des Schlossgartens in FSP

## Ziele:

- Die falsche und die korrekte Java-Implementierung des globalen Schlossgartenzählers in FSP modellieren
- durch Einsatz des Analysetools beliebige Traces durchspielen oder
- automatisch die Abwesenheit von Interferenzen feststellen oder
- automatisch einen Trace finden, der zu Race Hazards im globalen Zähler führt

# Im Modell verwendete Bezeichner

- `Counter`: modelliert den globalen Zähler
- `Gate`: modelliert ein Tor einschließlich des Zugriffs auf den globalen Zähler
- `Display`: Anzeige bzw. Abruf des globalen Zählerstandes
- `Garden`: parallele Komposition bestehend aus zwei Toren, dem globalen Zähler und der Anzeige.

# Schlossgartenmodell

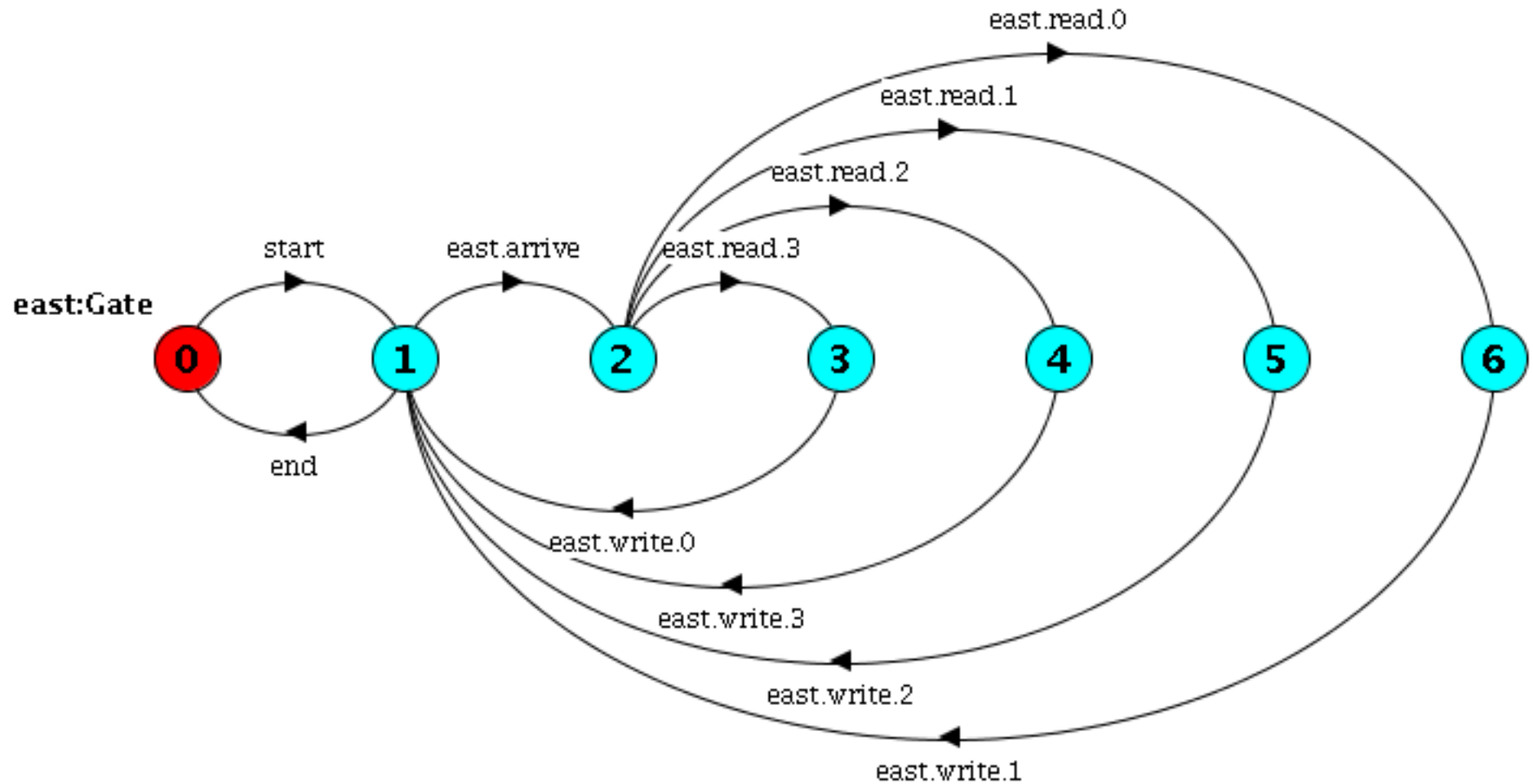
```
const N = 3          // maximale Anzahl Besucher
range T = 0..N       // Wertebereich fuer Zaehler (zyklisch)

Counter            = Counter[0],
Counter[u:T]       = (read[u] -> Counter[u]
                      | write[v:T] -> Counter[v]).

Gate               = (start -> Run),
Run                = (arrive -> Increment
                      | end   -> Gate),
Increment          = (read[x:T] -> write[(x+1)%(N+1)] -> Run).
Display            = (read[T] -> Display)
                  + {write[T]}.

||Garden = (east:Gate || west:Gate || display:Display
           || {east, west, display}::Counter)
           / { start / {east, west}.start,
              end / {east, west}.end }.
```

# Zustandsgraph des Ost-Tores



# Gültige Traces des Schlossgartens



The Animator window displays a sequence of events in a list. The events are: start, east.arrive, east.read.0, east.write.1, west.arrive, west.read.1, west.write.2, end, and display.read.2. The window has a blue header bar with a red 'X' icon and the title 'Animator'.

```
start  
east.arrive  
east.read.0  
east.write.1  
west.arrive  
west.read.1  
west.write.2  
end  
display.read.2
```

richtiges Ergebnis



The Animator window displays a sequence of events in a list. The events are: start, east.arrive, east.read.0, west.arrive, west.read.0, west.write.1, east.write.1, end, and display.read.1. The window has a blue header bar with a red 'X' icon and the title 'Animator'.

```
start  
east.arrive  
east.read.0  
west.arrive  
west.read.0  
west.write.1  
east.write.1  
end  
display.read.1
```

falsches Ergebnis



# Bemerkung

**Frage:** Was passiert, wenn im Prozess `Display` die Erweiterung des Alphabets um

`+ write[T]`

weggelassen wird?

**Hinweis:** Betrachte im Gesamtprozess `Garden` die Ausführbarkeit der Aktionen `display.write.{N}` (z. B. unter Verwendung des Animators).

# Automatisches Auffinden von Race Hazards

- Das Modell wird um einen `Test`-Prozess erweitert, der nach Ablauf der `end`-Aktion den Zählerstand prüft, und im Fehlerfall den `ERROR`-Zustand annimmt.
- Das mögliche Erreichen des `ERROR`-Zustands kann durch das Analysetool getestet werden.

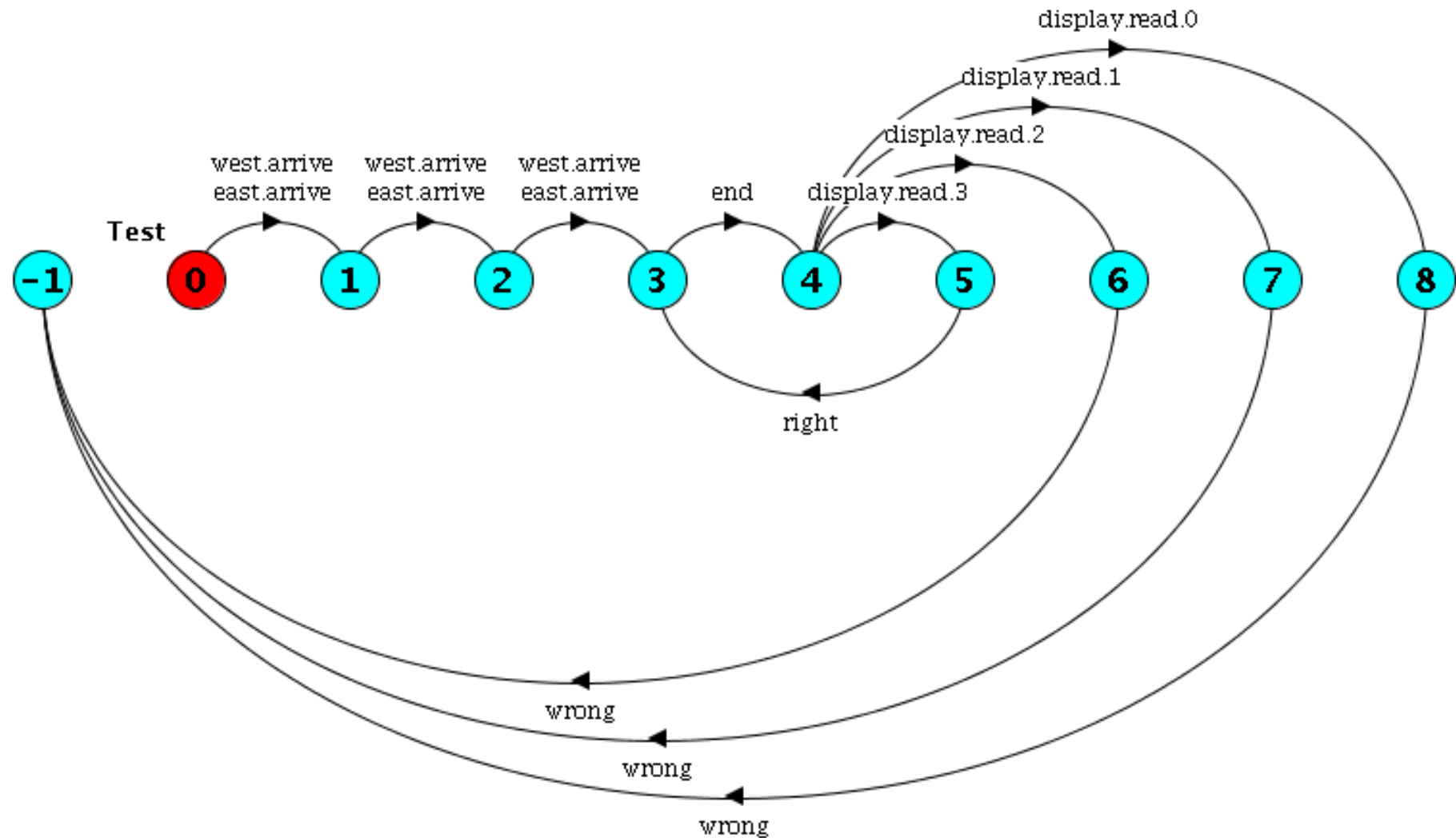
# Schlossgartenmodell + autom. Test

...

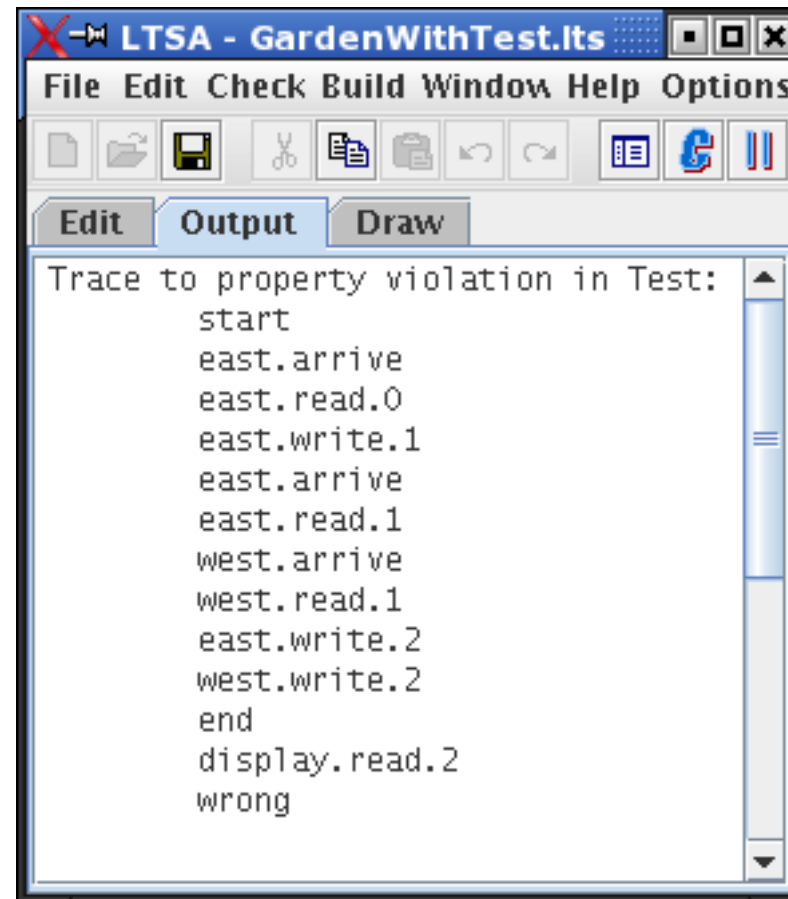
```
Test          = Test[0],    // zaehle explizit mit!
Test[v:T]     = (when (v < N)
                  {east.arrive,west.arrive} -> Test[v+1]
                  | when (v == N) end -> Check[v]),
Check[v:T]    = (display.read[u:T] ->
                  (when (u == v) right -> Test[v]
                   |when (u != v) wrong -> ERROR)
                  ).

||Testgarden = (Garden || Test).
```

# Zustandsgraph des Test-Prozesses ( $N = 3$ )



# Trace zu einem Race Hazard



# Überlegungen zum korrekten Modell

**Frage:** Wie sieht das Modell zum korrekten Java-Programm aus bzw. wie wird der durch `synchronized` erzwungene Lock-Mechanismus modelliert?

**Antwort:** vgl. [voriges FSP-Beispiel zum gegenseitigen Ausschluss!](#)

→ Überall, wo auf den globalen Zähler zugegriffen wird – auch beim Lesen(!) –, wird der Zugriff über gegenseitigen Ausschluss geregelt.

# Korrektes Schlossgartenmodell

```
const N = 3
range T = 0..N // Wertebereich fuer Zaehler (zyklisch)

Counter      = Counter[0],
Counter[u:T] = (read[u] -> Counter[u]
                | write[v:T] -> Counter[v]).

Lock = (acquire -> release -> Lock).
||LockedCounter = (Lock || Counter).

Gate      = (start -> Run),
Run       = (arrive -> Increment
            | end   -> Gate),
Increment = (acquire -> read[x:T] -> write[(x+1) % (N+1)]
            -> release -> Run).
```

```
Display = (acquire -> read[T] -> release -> Display)
          + {write[T]}.
```

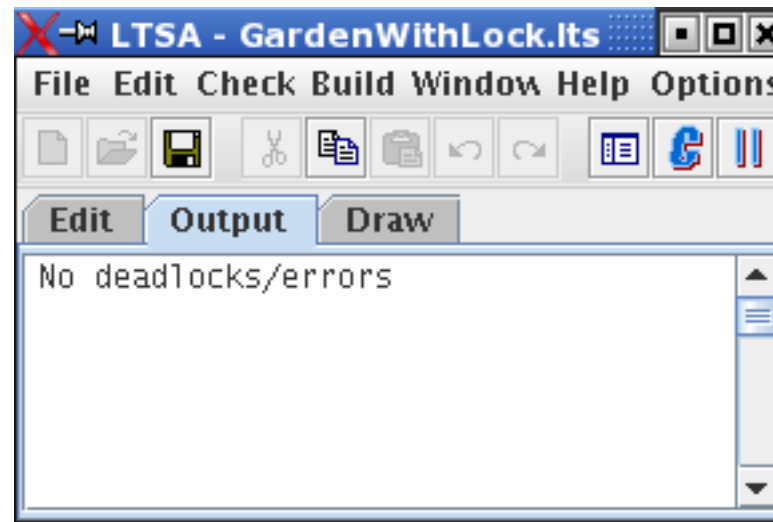
```
||Garden = (east:Gate || west:Gate || display:Display
           || {east, west, display}::LockedCounter)
           / { start / {east, west}.start,
              end / {east, west}.end }.
```

```
Test      = Test[0],
Test[v:T]  = (when (v < N)
              {east.arrive, west.arrive} -> Test[v+1]
              | when (v == N) end -> Check[v]),
Check[v:T] = (display.acquire -> display.read[u:T]
              -> display.release
              -> (when (u == v) right -> Test[v]
                  |when (u != v) wrong -> ERROR) ).
```

```
||Testgarden = (Garden || Test).
```



# Korrektheitsnachweis mit Analysetool



# Fazit

- Race Hazards werden durch nicht vom Programmierer steuerbare Einflüsse verursacht (besondere zeitliche Konstellationen)
- Race Hazards lassen sich nicht durch Testen finden
- Race Hazards lassen sich auch nicht durch Debugger aufspüren (Debugger sind für „statische“ Fehler gedacht und beeinflussen zusätzlich den zeitlichen Ablauf)
- die Abwesenheit von Race Hazards lässt sich nur durch formale Methoden („Model Checker“) nachweisen
- Tipp: Einfache Programm-Konstrukte sind in der Regel auch einfacher verifizierbar als komplizierte.

## 4.2 Monitore

- Monitore sind ein allgemeines Synchronisationskonzept für nebenläufige Prozesse,
- Monitore wurden 1974 von Hoare vorgestellt [[Hoa74](#)],
- interne Daten eines Monitors dürfen nicht direkt sondern nur über Methoden (Prozeduren) verändert werden,
- nur *ein* Prozess zurzeit darf eine Methode eines Monitors ausführen,
- die Ausführung kann an eine Bedingung geknüpft sein,
- eine „übergeordnete Instanz“ (Laufzeit-, Betriebssystem, ...) regelt, *welcher* Prozess eine Methode ausführen darf.

# Monitore in Java

- das Monitorkonzept ist Bestandteil von Java,
- die „übergeordnete Instanz“ wird durch das Java-Laufzeitsystem realisiert (JVM, Locks),
- implementiert werden Monitore in Java durch Klassen,
  - die nur verborgene (`private`) Daten enthalten und
  - deren öffentliche (`public`) Methoden alle als `synchronized` deklariert sind.

**Beispiele:** Implementierung von `GoodCounter` bzw. rekursives Beispiel von `Counter` im Schlossgarten.

# Beispiel: Parkplatz

- Der Parkplatz hat  $N$  Stellplätze,
- wenn Plätze frei sind, darf die Aktion **parken** über die **Zufahrt** ausgeführt werden,
- wenn Plätze belegt sind, darf die Aktion **wegfahren** über die **Ausfahrt** ausgeführt werden.

Wie sieht eine Modellierung der beteiligten Prozesse aus?

# Parkplatzmodellierung in LTS (1)

```
/* Parkplatz */
```

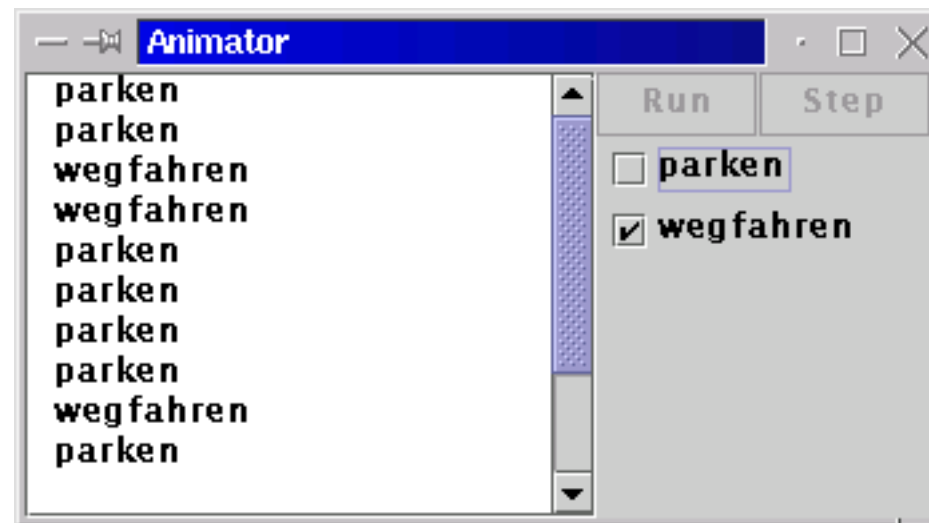
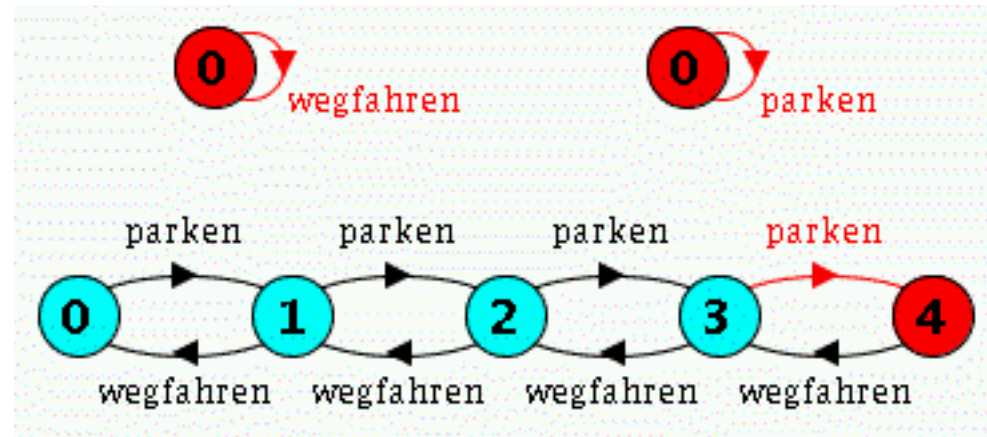
```
ParkKontrolle(N=3) = FreiePlaetze[N],  
FreiePlaetze[i:0..N] = (when (i > 0) parken ->  
                        FreiePlaetze[i-1]  
                        | when (i < N) wegfahren ->  
                        FreiePlaetze[i+1]) .
```

```
Zufahrt = (parken -> Zufahrt) .
```

```
Ausfahrt = (wegfahren -> Ausfahrt) .
```

```
||Parkplatz = (Zufahrt || Ausfahrt || ParkKontrolle(4)) .
```

# Parkplatzmodellierung in LTS (2)



# Parkplatzimplementierung in Java (1)

```
class ParkKontrolle {  
    private int freiePlaetze = N;  
  
    public ParkKontrolle() {}  
  
    public synchronized void parken() {  
        freiePlaetze--;  
    }  
  
    public synchronized void wegfahren() {  
        freiePlaetze++;  
    }  
}
```

Monitor ist o.k., aber was ist mit den bewachten Aktionen?



```
ParkKontrolle(N=3) = FreiePlaetze[N],  
FreiePlaetze[i:0..N] = (when (i > 0) parken ->  
                        FreiePlaetze[i-1]  
                        | when (i < N) wegfahren ->  
                        FreiePlaetze[i+1]) .
```

- Parken ist nur erlaubt, wenn noch freie Plätze vorhanden sind!
- Wegfahren ist nur möglich, wenn mindestens ein Platz belegt ist!

Wie kann das in Java implementiert werden?

# Parkplatzimplementierung in Java (2)

```
class ParkKontrolle {  
    private int plaetze = N;  
    ...  
    public synchronized boolean erfolgreichGeparkt() {  
        if (plaetze == 0) {  
            return false; // Parkplatz ist voll  
        } else {  
            plaetze--;  
            return true;  // Parken war moeglich  
        }  
    }  
    ...  
}
```

```

class Zufahrt extends Thread {
    private ParkKontrolle parkKontrolle;
    private int n; // Anzahl einfahrende Fahrzeuge
    ...

    public void run() {
        for (int i = 0; i < n; i++) {
            // i-tes Auto parken
            boolean geparkt = false;
            // solange versuchen, bis es geklappt hat
            while (!geparkt) {
                geparkt = parkKontrolle.erfolgreichGeparkt();
            }
        }
    }
}

```

o.k., aber „**aktives Warten**“<sup>7</sup> verbraucht unnötig Prozessorzeit!

<sup>7</sup> Andere Bezeichnungen: „busy waiting“ oder „spinning“

**Frage:** Wie kann innerhalb des Monitors *passiv* auf das Erfüllen einer Bedingung gewartet werden?

**Antwort:** Jedem Java-Objekt (`java.lang.Object`) ist neben dem **Lock** und der **Ausschluss-Wartemenge** eine „**Ereignis-Wartemenge**“ zugeordnet, in der Threads auf das Eintreten von Ereignissen warten können:

- während des Aufenthalts in der Ereignis-Wartemenge sind sie „**nicht lauffähig**“, so dass ihnen nicht der Prozessor zugeteilt werden kann,
- sie müssen durch einen anderen, laufenden Thread aus der Ereignis-Wartemenge „befreit“ werden,
- die Manipulation der Ereignis-Wartemenge ist durch die Methoden `wait`, `notify` und `notifyAll` möglich.

Die folgenden Methoden können nur aus dem zum Objekt gehörenden Monitor aufgerufen werden!

1. `public final void wait()`: suspendiert den aufrufenden Thread; der Thread verlässt den Monitor und wird in die entsprechende Ereignis-Wartemenge überführt.
2. `public final void notify()`: entfernt einen *beliebigen* Thread aus der Ereignis-Wartemenge und versetzt ihn in den Status „**lauffähig**“ (d. h. er wartet jetzt wieder in der Ausschluss-Wartemenge).
3. `public final void notifyAll()`: entfernt *alle* Threads aus der Ereignis-Wartemenge und versetzt sie in den Status „**lauffähig**“ (d. h. alle warten jetzt wieder in der Ausschluss-Wartemenge).

**Bemerkung:** Ein mit `notify()` oder `notifyAll()` „aufgeweckter“ Thread setzt seine Abarbeitung direkt nach dem zuletzt ausgeführten `wait()` fort.

**Bemerkung:** *Welcher* Thread nach der Ausführung von `notify()` oder `notifyAll()` als nächstes über den Lock Zugang zum Monitor erhält, ist unbestimmt!

**Bemerkung:** Werden die Methoden `wait()`, `notify()` oder `notifyAll()` von einem Thread aufgerufen, der *keine* Kontrolle über den Monitor des Objektes besitzt, so wird eine `IllegalMonitorStateException` geworfen!

# Realisierung bewachter (atomarer) Aktionen in Java

## FSP:

Zustand1 = (when *bedingung* aktion -> Zustand2) ,

## Java:

```
public synchronized void aktion ()  
    throws InterruptedException {  
    while (! bedingung) { wait(); }  
    ...    // Variablen aendern (Zustand2)  
    notifyAll();  
}
```

# Umsetzungsregeln für bewachte Aktionen

- Jede bewachte Aktion wird innerhalb eines Monitors als `synchronized` Methode implementiert
- solange die negierte Bedingung des Wächters nicht erfüllt ist, wird der Monitor mit `wait()` wieder verlassen
- Zustandsänderungen innerhalb des Java-Monitors werden eventuellen Prozessen in der Ereignis-Wartemenge mit `notifyAll()` signalisiert
- nur in begründeten *Ausnahmefällen* darf `notify()` statt `notifyAll()` verwendet werden!



## Parkplatzimplementierung in Java (3)

```
class ParkKontrolle {  
    private int plaetze = N;  
    ...  
    public synchronized void parken()  
        throws InterruptedException {  
        while (! (plaetze > 0)) { wait(); }  
        plaetze--;  
        notifyAll();  
    }  
  
    public synchronized void wegfahren()  
        throws InterruptedException {  
        while (! (plaetze < N)) { wait(); }  
        plaetze++;  
        notifyAll();  
    }  
}
```

# Übung: Bewerten Sie das folgende Beispiel

```
public class Test {  
    private Object o = new Object();  
    private int i = 0, j = 0; // gemeinsame benutzte Var.  
  
    public synchronized void foo()  
        throws InterruptedException {  
        while (i != 42) { wait(); }  
        i = j;  
    }  
    public void bar() {  
        synchronized(o) {  
            j = 7;  
            synchronized(this) { i = 42; }  
            notifyAll();  
        }  
    }  
}
```

# Beispiel: Beschränkte Puffer (Bounded Buffer)

**Beschränkte Puffer** werden oft eingesetzt, um ...

- kleine Datenmengen zwischenspeichern (meist als FI-FO realisiert) oder
- den Datenstrom zwischen Produzent und Konsument zu „glätten“ (z. B. weil die Verarbeitungszeit im Konsumenten variiert).
- Produzent und Konsument können gleichzeitig schreibend bzw. lesend auf den Puffer zugreifen.

# Modellierung beschränkter Puffer

Da die Daten im Puffer nur gespeichert, aber nicht verarbeitet/verändert werden, braucht nur die Logik des Puffers und des Zugriffs (ohne die Daten) modelliert zu werden!

Modellierung eines „Bounded Buffer“ in FSP:

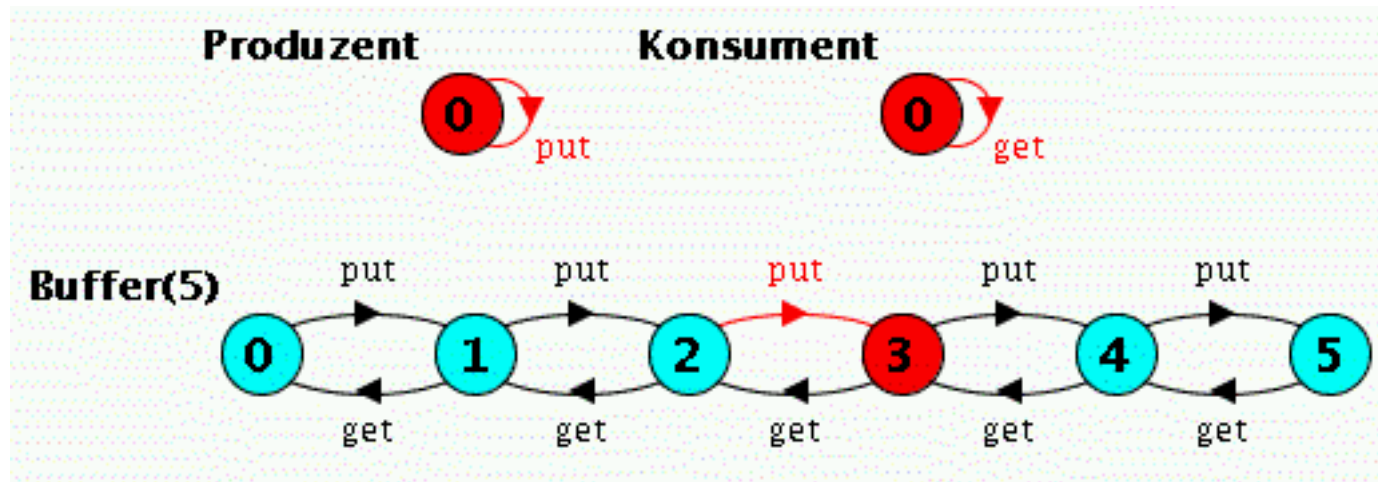
```
Buffer(N=5) = Speicher[0],  
Speicher[i:0..N] = ( when (i<N) put -> Speicher[i+1]  
                    | when (i>0) get -> Speicher[i-1]) .
```

```
Produzent = (put -> Produzent) .
```

```
Konsument = (get -> Konsument) .
```

```
||BoundedBuffer = (Produzent || Buffer(5) || Konsument) .
```

# Zustandsdiagramm beschränkter Puffer



**Bemerkung:** Das Zustandsdiagramm des beschränkten Puffers hat starke Ähnlichkeit mit der [Modellierung des Parkplatzes](#).

**Bemerkung:** Aber, hier ist die Ausgabereihenfolge wesentlich (zwar nicht modelliert, aber bei der Implementierung zu beachten)!

# Implementierung beschränkter Puffer in Java

```
public class BoundedBuffer<E> {  
    private final E[] speicher;  
    private int rein = 0,    // nächster Schreibindex  
               raus = 0,    // nächster Leseindex  
               count = 0;   // Belegungszähler  
    private final int size;  
  
    public BoundedBuffer(int size) {  
        this.size = size;  
        speicher = (E[]) new Object[size];  
    }  
}
```

```

public synchronized void put(E object)
    throws InterruptedException {
    while (!(count < size)) wait(); // count == size
    speicher[rein] = object;
    count++;    rein = (rein + 1) % size;
    notifyAll();
}

public synchronized E get()
    throws InterruptedException {
    while (!(count > 0)) wait(); // count == 0)
    final E object = speicher[raus];
    speicher[raus] = null;
    count--;    raus = (raus + 1) % size;
    notifyAll();
    return object;
}
}

```

# Aufgabe

Ändert sich das Verhalten der vorgestellten **Implementierung von `BoundedBuffer`**, wenn dort abweichend von den Umsetzungsregeln `notifyAll()` durch `notify()` ersetzt wird?

Was kann z. B. beim Zusammenspiel mit einem Konsumenten- und zwei Produzenten-Threads und einem `BoundedBuffer` der Größe 1 passieren?

Lässt sich die Beobachtung auf Puffer beliebiger Größe verallgemeinern?



## 4.3 Semaphore

- **Semaphore** stellen ein *einfaches* Synchronisationskonzept für nebenläufige Prozesse dar („Semaphor: Signalmast“)
- Semaphore wurden 1968 von Dijkstra eingeführt [[Dij68](#)]
- bei Semaphoren gibt es im Unterschied zu **Monitoren** *keine zentrale Instanz*, die die zusammengehörenden kritischen Regionen einer Anwendung überwacht

- ein Semaphor kann als Variable mit ganzzahligem, nicht-negativem Wertebereich aufgefasst werden
- nach der Initialisierung der Variablen können nur die Operationen `down` oder `up` ausgeführt werden (im Original hießen die Operationen  $P$  und  $V$ ):
  - `up` ist eine *atomare* Operation, die den Wert des Semaphors um eins hochzählt
  - `down` ist eine *atomare* Operation, die den Wert des Semaphors um eins herunterzählt, falls der Wert vorher  $> 0$  ist und andernfalls blockiert
- auch mit Semaphoren kann der **gegenseitige Ausschluss** realisiert werden<sup>8</sup>

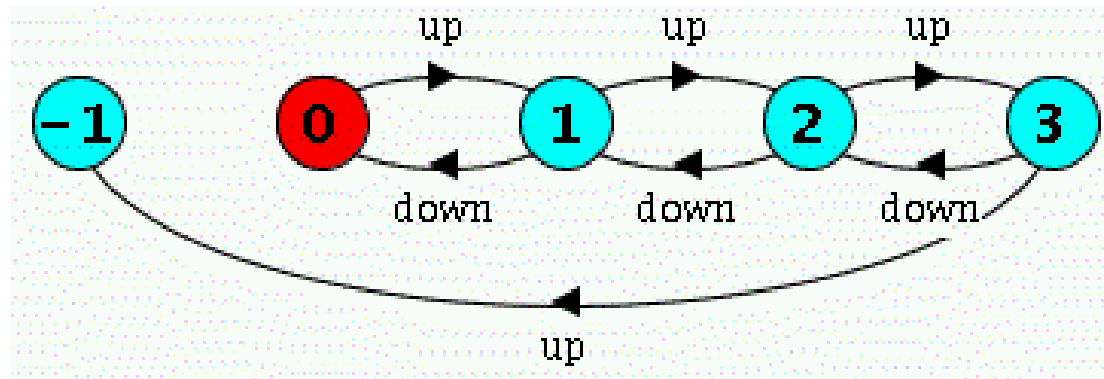
---

<sup>8</sup> Für diesen Zweck werden sie auch als „Mutex“ bezeichnet.

# Modellierung von Semaphoren

Um Semaphore besser analysieren zu können, werden sie zunächst in FSP modelliert:

```
const Max = 3
range Int = 0..Max
Semaphor(N=0) = Sema[N],
Sema[i:Int]   = ( up -> Sema[i+1]
                  | when (i>0) down -> Sema[i-1] ) .
```



# Gegenseitiger Ausschluss mit Semaphoren

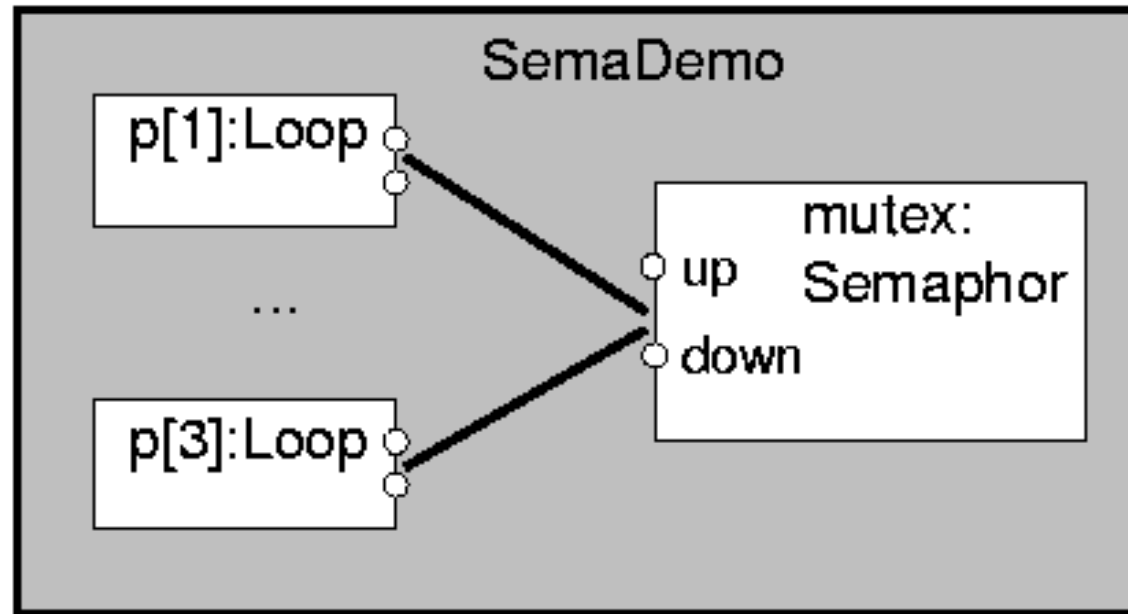
Nur ein Prozess zurzeit darf einen sogenannten **kritischen Bereich** betreten:

```
const Max = 3
range Int = 0..Max
Semaphor(N=0) = Sema[N],
Sema[i:Int]   = ( up -> Sema[i+1]
                  | when (i>0) down -> Sema[i-1]) .

Loop = (mutex.down -> critical -> mutex.up -> Loop) .

|| SemaDemo = (p[1..3]:Loop
               || p[1..3]::mutex:Semaphor(1)) .
```

# Strukturdiagramm

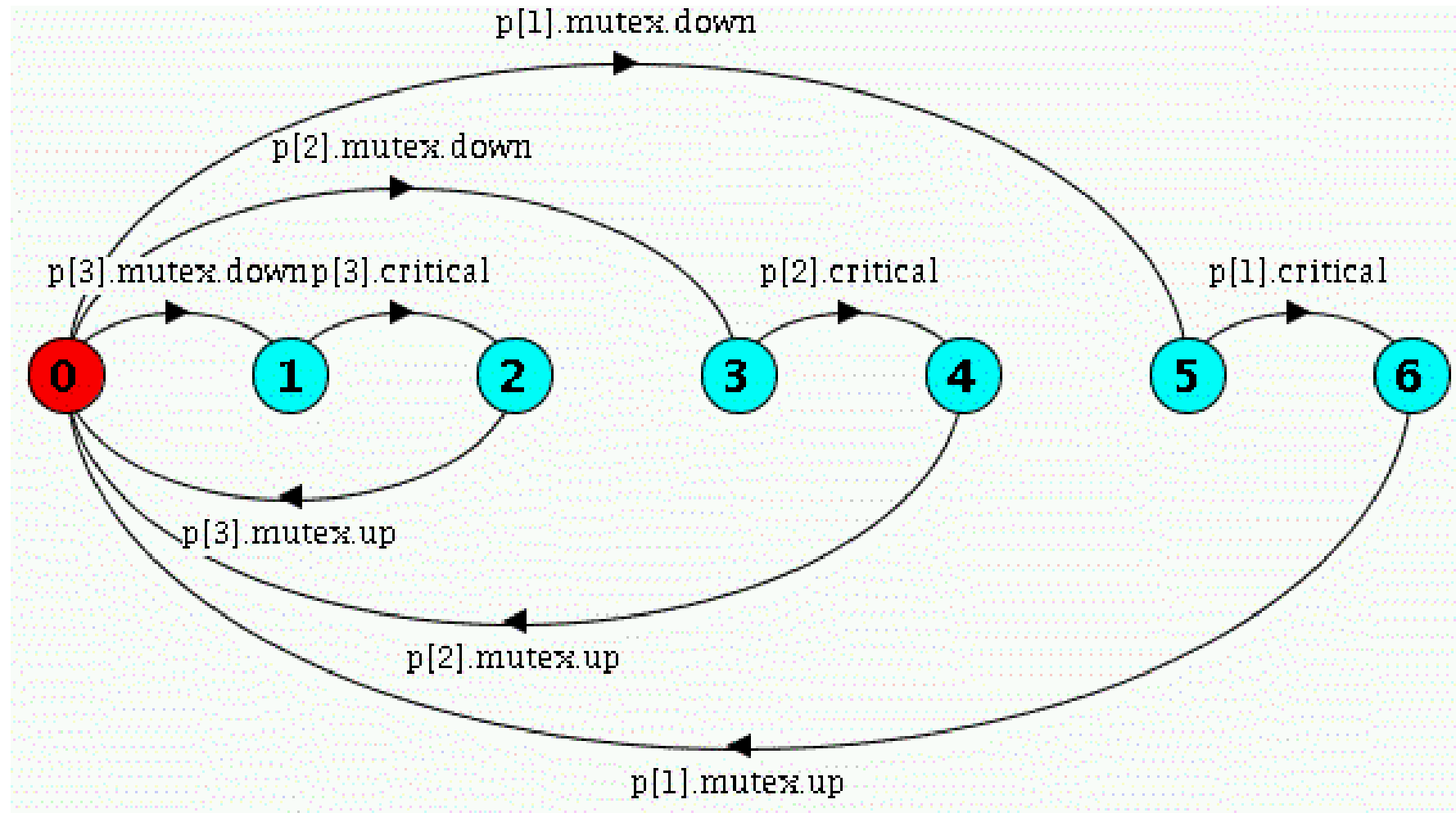


# Zeitlicher Ablauf

1. der Semaphore wird mit „1“ initialisiert (d. h. nur ein Prozess darf sich zurzeit im kritischen Bereich aufhalten),
2. der erste Prozess, der den kritischen Bereich betreten möchte, führt `mutex.down` aus, dadurch werden die anderen Prozesse am Betreten gehindert,
3. nach Verlassen des kritischen Bereiches muss der Semaphore durch `mutex.up` explizit wieder freigegeben werden.

Korrekt angewendet, wird der Wertebereich des Semaphors nicht verletzt (mit LTSA verifiziert).

# Zustandsgraph



# Semaphore in Java

Semaphore lassen sich mithilfe eines Monitors einfach nachbilden:<sup>9</sup>

- jeder Semaphor besitzt einen internen Zähler (`private Variable`),
- der Zähler wird durch unteilbare Aktionen `up` und `down` verändert (Verwendung von `synchronized`)

Die Implementierung von Semaphoren in Java folgt damit direkt aus dem **Modell** mithilfe der **Umsetzungsregeln für bewachte Aktionen**:

---

<sup>9</sup> In der Praxis verfährt man jedoch meist umgekehrt und implementiert die „high level“ Monitore mithilfe der „low level“ Semaphore



```

public class Semaphor {
    private int wert;
    public Semaphor(int initial){
        this.wert = initial;    // Semaphor initialisieren
    }
    public synchronized void up() {
        // FSP: up -> Sema[i+1]
        // kein Wächter, daher hier auch kein wait
        wert++;
        notifyAll();
    }
    public synchronized void down()
        throws InterruptedException {
        // FSP: when (i>0) down -> Sema[i-1]
        while (!(wert > 0)) { wait(); }
        wert--;
        notifyAll();
    }
}

```

# Semaphore in Java ab Version 1.5

Ab der Version 1.5 sind Semaphore in Java auch direkt implementiert!

Klasse `java.util.concurrent.Semaphore`:

- Konstruktor `Semaphore(int permits)` erzeugt Semaphore mit Initialwert `permits`
- Methode `public void acquire()` implementiert die down-Operation
- Methode `public void release()` implementiert up-Operation
- weitere Methode siehe API ...

# Beispiel (1): Schlossgarten mit Semaphor

- Zentrale Stelle der Implementierung ist die Instanz der Klasse `Counter`, die von allen nebenläufigen Threads gemeinsam benutzt wird.
- Beim Zugriff auf den Zähler gegenseitigen Ausschluss sicherstellen („kritischer Bereich“).

```
class Counter { // ist kein Monitor mehr!
    private int value; // aktueller Zaehlerstand

    // nur ein Prozess darf den Zähler manipulieren
    private Semaphor semaphor = new Semaphor(1);

    // Instanziieren findet nur von einem Thread
    // statt, ist also unkritisch.
    public Counter() { value = 0; }
```

```
// Gegenseitiger Ausschluss für das Hochzählen
public void incrementValue() { // nicht synchronized!
    semaphor.down();
    // kritischer Bereich
    int temp = value + 1;
    value = temp;
    semaphor.up();
}

// Gegenseitiger Ausschluss auch für das Lesen
public int getValue() { // nicht synchronized!
    semaphor.down();
    // kritischer Bereich
    int temp = value;
    semaphor.up();
    return temp;
}
}
```

## Beispiel (2): Parkplatz mit Semaphor

- Zentrale Stelle ist die Implementierung der **Parkkontrolle**, die von allen Threads gemeinsam benutzt wird.
- Zusätzlich zum gegenseitigen Ausschluss müssen **Bedingungen** für das Parken bzw. Wegfahren eingehalten werden.

```
class ParkKontrolleSemaphor01 { // ist kein Monitor mehr!  
    // Zähler für die freien Plätze  
    private int plaetze = N;  
  
    // nur ein Prozess darf auf die Verwaltung zugreifen  
    private Semaphor semaphor = new Semaphor(1);  
  
    ...
```

```

public void parken() { // nicht synchronized!
    boolean keinPlatzBekommen = true;
    while (keinPlatzBekommen) {
        // kritischen Bereich betreten
        semaphor.down();
        if (plaetze <= 0) {
            // kritischen Bereich temporär verlassen...
            semaphor.up();
        } else { keinPlatzBekommen = false; }
    }
    plaetze--;
    // kritischen Bereich wieder verlassen
    semaphor.up();
}
...
}

```

o.k., aber wieder aktives Warten!

# Beispiel (3): Parkplatz mit zwei Semaphoren

Statt aktiv zu warten, *zwei* Semaphore verwenden:

1. `freiePlaetze` verwaltet die freien Plätze,
2. `belegtePlaetze` verwaltet die belegten Plätze.

```
class ParkKontrolleSemaphor02 { // ist kein Monitor mehr!
    private final Semaphor freiePlaetze,
                          belegtePlaetze;

    public ParkKontrolleSemaphor02 () {
        freiePlaetze = new Semaphor(N);
        belegtePlaetze = new Semaphor(0);
    }
}
```

```
public void parken() { // nicht mehr synchronized!  
    freiePlaetze.down();  
    belegtePlaetze.up();  
}  
  
public void wegfahren() { // nicht mehr synchronized!  
    belegtePlaetze.down();  
    freiePlaetze.up();  
}  
}
```

Implementierung o.k.? Hoffentlich (hat nämlich mit der **ursprünglichen Modellierung** in FSP nichts mehr gemeinsam!).

Bei der Verwendung von Semaphoren ist eine Modellierung *ohne Monitor*, d. h. ohne zentrale/synchronisierende Parkkontrolle, notwendig! Wie muss die Modellierung aussehen?



# Parkplatzmodellierung mit Semaphoren in FSP

- Die Klasse `ParkKontrolle` wird in FSP nicht mehr als eigener Prozess modelliert, da ihre Methoden nicht mehr `synchronized` sind und daher unabhängig voneinander aufgerufen werden können,
- stattdessen werden zwei Semaphore zur Synchronisation verwendet.

```
// übernehme bekannte Semaphor-Modellierung
const Max = 5
range Int = 0..Max
Semaphor(N=0) = Sema[N],
Sema[i:Int]   = ( up -> Sema[i+1]
                  | when (i>0) down -> Sema[i-1]) .

Zufahrt = (freiePlaetze.down -> belegtePlaetze.up ->
           Zufahrt) .

Ausfahrt = (belegtePlaetze.down -> freiePlaetze.up ->
            Ausfahrt) .

||Parkplatz = (Zufahrt || Ausfahrt
               || freiePlaetze:Semaphor(5)
               || belegtePlaetze:Semaphor(0)) .
```

**Aufgabe:** Korrekte Funktionsweise mit LTSA überprüfen!

**Aufgabe:** Was passiert, wenn im vorigen Programm `ParkKontrolleSemaphor02` die Methoden `parken` und `wegfahren` trotz des Semaphors als `synchronized` gekennzeichnet werden?

**Lösungsidee:** Modelliere die Funktion und überprüfe die Funktionsweise mit LTSA („Nested Monitors“)!

# Bemerkungen

1. **Semaphor:** Maßnahmen zum gegenseitigen Ausschluss (zur Atomarität der Aktionen im kritischen Bereich) sind über den Quellcode bzw. das Modell verteilt:
  - Gemeinsame `down`- und `up`-Aktionen im Modell
  - Zugriff auf `down`- und `up`-Methoden in der Implementierung
2. **Monitor:** Gegenseitiger Ausschluss (Atomarität der Aktionen im kritischen Bereich) allein im Monitor gelöst:
  - Modellierung durch Auswahl mit bewachten Aktionen
  - Implementierungsdetails des Monitors sind von Außen nicht sichtbar

# Beispiel (4): **Bounded Buffer** mit Semaphoren

Idee?

Im Gegensatz zur **Parkplatzimplementierung** mit Semaphoren müssen im Puffer auch Daten gespeichert werden!

Das erfordert Maßnahmen zum gegenseitigen Ausschluss beim Zugriff auf den Speicher!

# Bounded Buffer mit Semaphoren: 1. Ansatz

Verwende zwei Semaphore zum Zählen der freien Plätze (wie beim Parkplatz) und einen Monitor zum gegenseitigen Ausschluss für den Speicherzugriff:

```
public class BoundedBufferSemaphore01<E> {
    private final E[] speicher;
    private int vorne = 0, hinten = 0, count = 0;
    private final int size;
    private final Semaphore freie, belegte;

    public BoundedBufferSemaphore01(int size) {
        this.size = size;
        speicher = (E[]) new Object[size];
        freie = new Semaphore(size);
        belegte = new Semaphore(0);
    }
}
```

```

public synchronized void put(E object)
    throws InterruptedException {
    freie.acquire(); // freie.down()
    speicher[vorne] = object;
    count++; vorne = (vorne + 1) % size;
    belegte.release(); // belegte.up()
}

```

```

public synchronized E get()
    throws InterruptedException {
    belegte.acquire(); // belegte.down()
    E object = speicher[hinten];
    speicher[hinten] = null;
    count--; hinten = (hinten + 1) % size;
    freie.release(); // freie.up()
    return object;
}
}

```

**Bewertung:** Wie bei der [Aufgabe zum Parkplatz](#) können auch hier „Blockaden“ des Monitors auftreten, weil die Semaphore innerhalb des Monitors verwendet werden („Nested Monitors“)!

**Abhilfe:** Keinen Monitor verwenden, sondern die Anweisungen zum Zugriff auf den Datenspeicher über einen separaten `synchronized-Block` sichern!

```
public void put(E object) throws InterruptedException {
    freie.acquire(); // freie.down()
    synchronized (this) {
        speicher[vorne] = object;
        count++; vorne = (vorne + 1) % size;
    }
    belegte.release(); // belegte.up()
}
```



```
public E get() throws InterruptedException {
    belegte.acquire(); // belegte.down()
    E object;
    synchronized (this) {
        object = speicher[hinten];
        speicher[hinten] = null;
        count--; hinten = (hinten + 1) % size;
    }
    freie.release(); // freie.up()
    return object;
}
```

# Bemerkungen

- Die Verwendung von Semaphoren ist extrem fehleranfällig, da der *Programmierer* sicherstellen muss, dass die `up`- und `down`-Operationen immer paarweise und in der richtigen Reihenfolge verwendet werden
- Es ist Programmierdisziplin erforderlich, um die `up`- und `down`-Operationen der Semaphore möglichst zentral im Code zu halten!
- Deshalb sollten Semaphore nur dann verwendet werden, wenn keine höheren Synchronisationskonzepte (z. B. Monitore) zur Verfügung stehen
- In Java sollten immer Monitore statt Semaphore verwendet werden!

# Zusammenfassung (4)

- Nebenläufige Prozesse werden in Java durch Starten mehrerer Threads realisiert,
- gegenseitiger Ausschluss wird durch sogen. „**Locks**“ unter Verwendung des `synchronized` Schlüsselwortes erreicht,
- neben den Locks und der **Ausschluss-Wartemenge** besitzen Monitore eine **Ereignis-Wartemenge**, in der Threads passiv auf das Eintreten von Ereignissen warten,
- Java-Programme lassen sich durch FSP-Modelle modellieren und das Modell mit Hilfe des Analysetool auf mögliche **Interferenzen** (**Race Hazards**) überprüfen,

- Semaphore sind ein „low level“ Synchronisationskonzept,
- Monitore sind ein „high level“ Synchronisationskonzept,
- die Verwendung von Semaphoren ist extrem fehleranfällig, da eine zentrale Stelle zur Synchronisation fehlt:
  - Verwechseln oder Vergessen der „verstreuten“ `up`- und `down`-Operationen leicht möglich,
- wenn Vorhanden, sollten *Monitore* statt Semaphore eingesetzt werden!

# 5 Eigenschaften paralleler Programme

## Fragen:

- „Deadlocks“, was sind das und unter welchen Bedingungen treten sie auf?
- „Sicherheit“, was ist das und wann ist sie gewährleistet?
- „Lebendigkeit“, was ist das und wann ist sie gewährleistet?

## 5.1 Verklemmungen

**Definition:** Wenn in einem nebenläufigen System alle Prozesse des Systems blockiert sind, d. h. keine Aktion mehr ausgeführt werden kann, spricht man von einer *Verklemmungen* oder einem *Deadlock*.

# Beispiel: Deadlock

Zwei nebenläufige Prozesse  $P$  und  $Q$  wollen gleichzeitig einen Scanner und einen Drucker benutzen:

- $P$  fordert zuerst den Drucker und dann den Scanner an:

```
P = (printer.get -> scanner.get -> useP ->  
     printer.put -> scanner.put -> P) .
```

- $Q$  fordert zuerst den Scanner und dann den Drucker an:

```
Q = (scanner.get -> printer.get -> useQ ->  
     scanner.put -> printer.put -> Q) .
```

# Modellierung des Gesamtsystems

```
P = (printer.get -> scanner.get -> useP ->
     printer.put -> scanner.put -> P) .
```

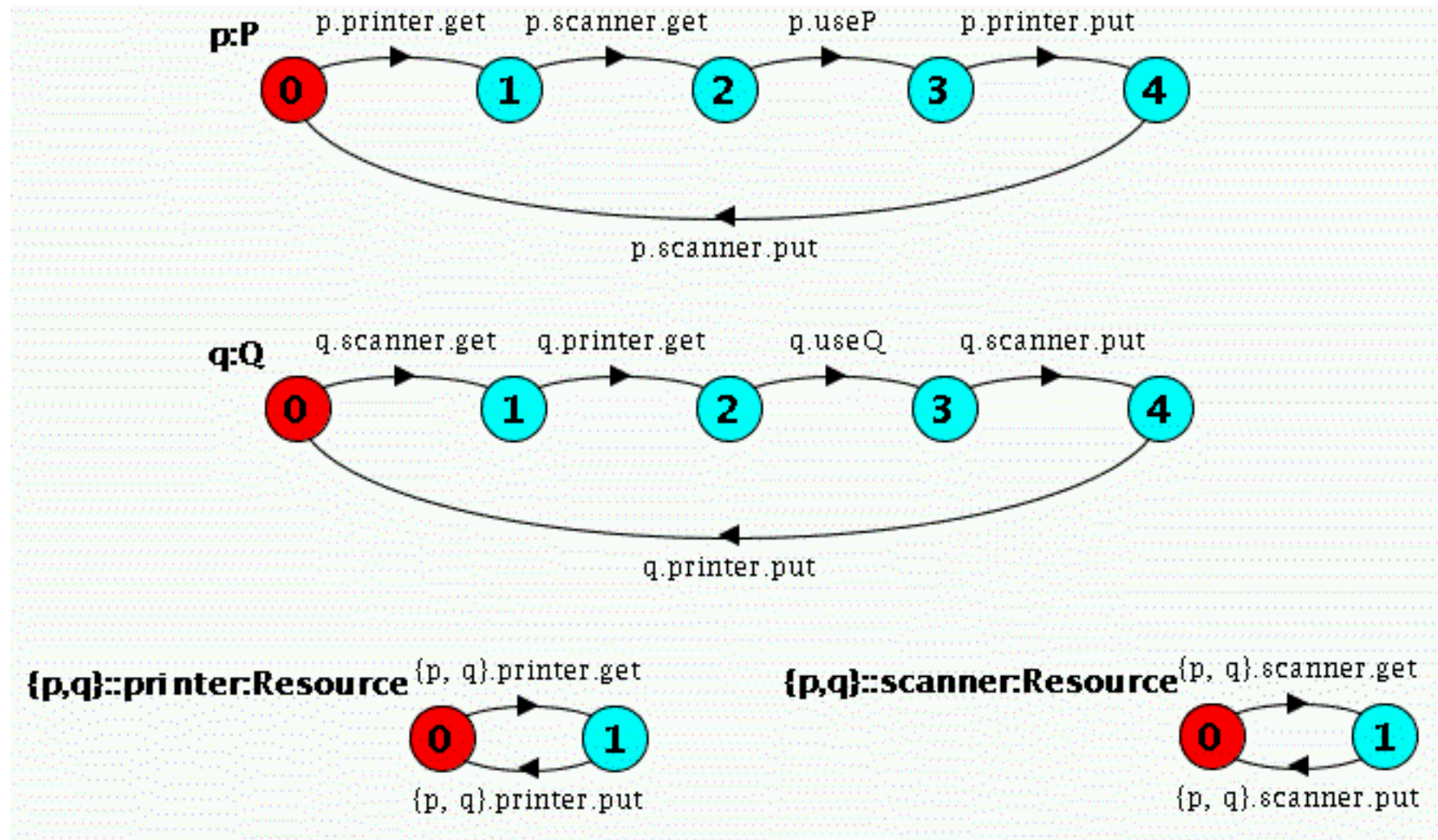
```
Q = (scanner.get -> printer.get -> useQ ->
     scanner.put -> printer.put -> Q) .
```

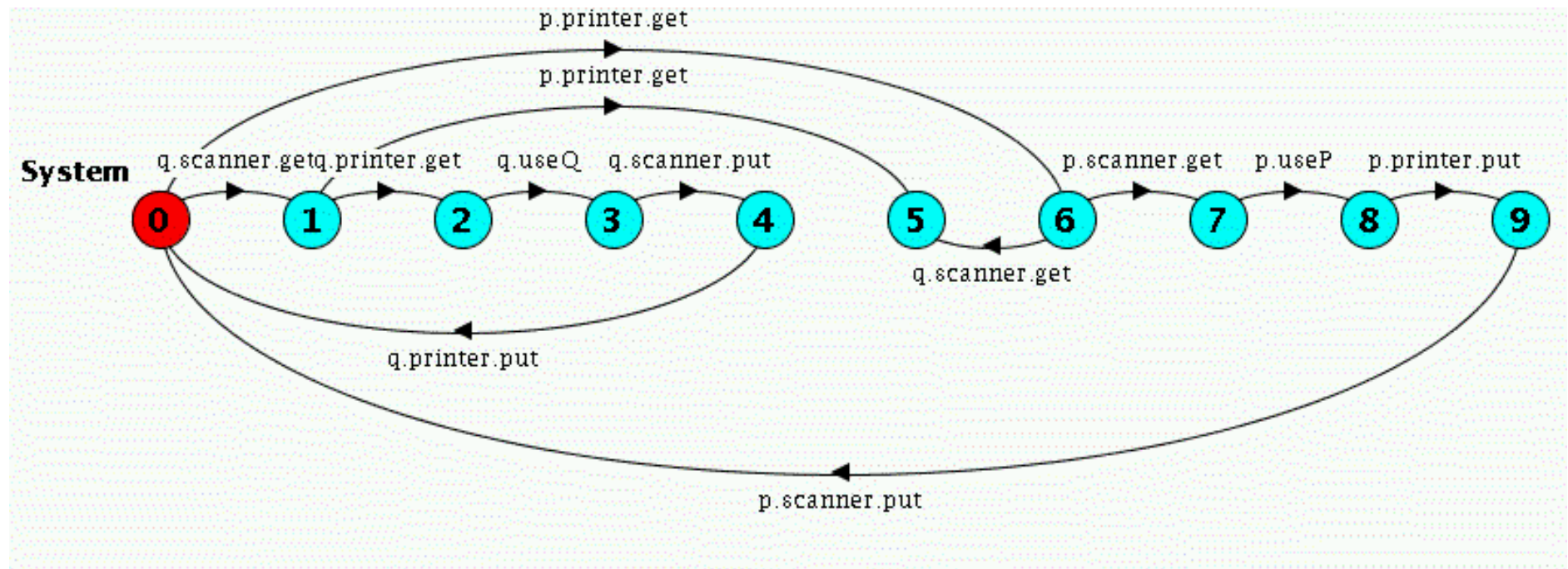
```
Resource = (get -> put -> Resource) .
```

```
||System = (p:P || q:Q ||
            {p, q}::printer:Resource ||
            {p, q}::scanner:Resource ) .
```



# Zustandsdiagramm des Gesamtsystems



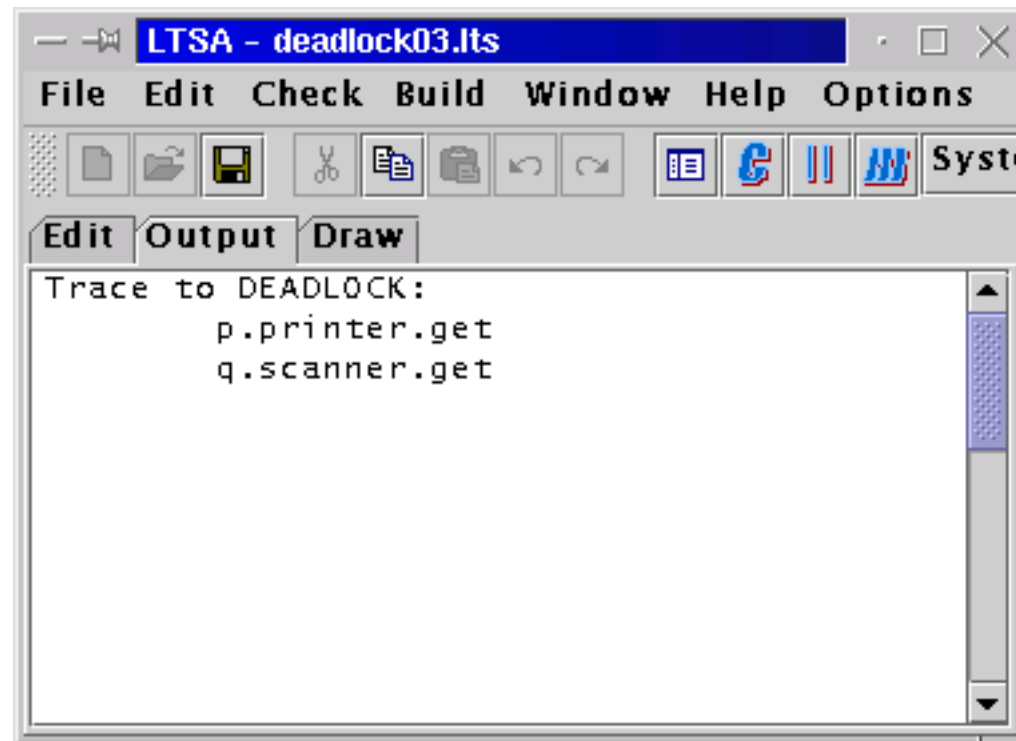


**Frage:** Was fällt auf? Was passiert im Zustand „5“?

# Analyse des Systems

Wie lassen sich potentielle Deadlocks finden?

Von LTSA suchen lassen!



# Deadlock-Bedingungen

Es gibt vier notwendige und hinreichende Bedingungen für das Auftreten von Deadlocks:

1. **Exklusive Benutzung von Ressourcen** (gegenseitiger Ausschluss).
2. **Inkrementelle Anforderung von Ressourcen** (Belegen von Ressourcen während auf weitere Ressourcen gewartet wird).
3. **Nichtunterbrechbarkeit** (keine temporäre Rückgabe von Ressourcen).
4. **Zyklische Abhängigkeit** (ein Prozess hält Ressourcen, auf die ein anderer Prozess wartet).

# Strategien zum Umgang mit Deadlocks

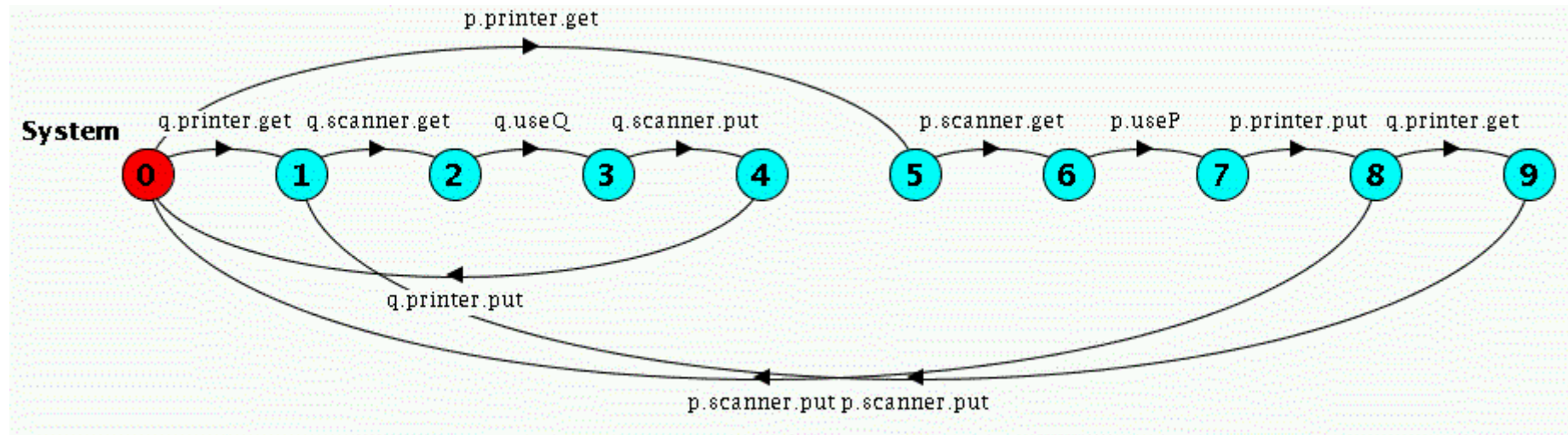
1. **Erkennen und Beseitigen:** Terminieren eines in einen Deadlock verwickelten Prozesses und Freigabe der von ihm belegten Ressourcen (Beseitigen von **Deadlock-Bedingung 3**).
2. **Vermeiden:** Sicherstellen, dass (mindestens) eine der vier notwendigen **Deadlock-Bedingungen** nie erfüllt sein wird.

# Beispiel: Vermeiden von Deadlocks

1. Feste Sperr-Reihenfolge, Ordnen der Ressourcenanforderungen (**Deadlock-Bedingung 4**):

$P = (\text{printer.get} \rightarrow \text{scanner.get} \rightarrow \text{useP} \rightarrow \text{printer.put} \rightarrow \text{scanner.put} \rightarrow P) .$

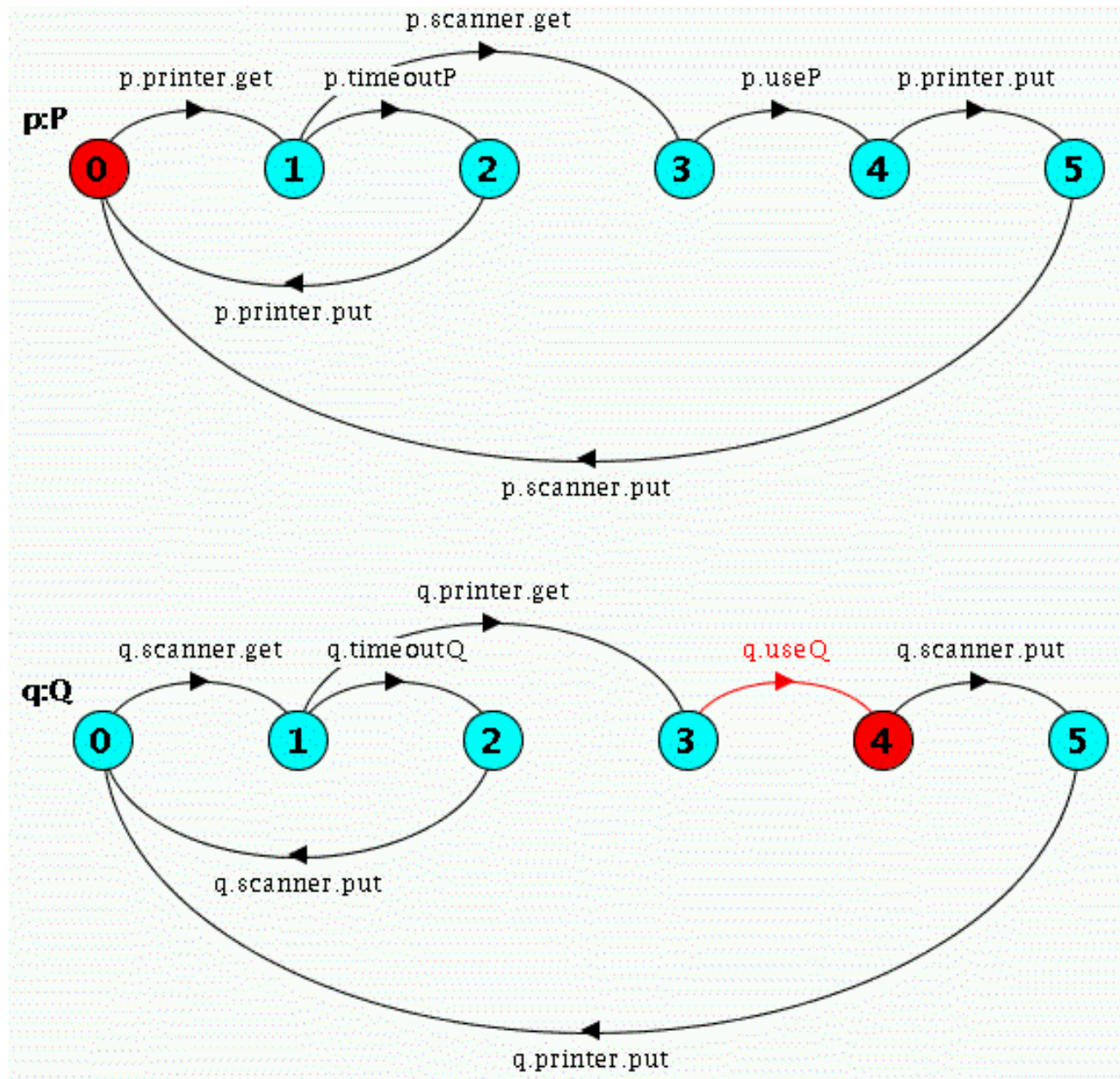
$Q = (\text{printer.get} \rightarrow \text{scanner.get} \rightarrow \text{useQ} \rightarrow \text{scanner.put} \rightarrow \text{printer.put} \rightarrow Q) .$



## 2. Rückgabe von Ressourcen, „Backoff-Strategie“ (Deadlock-Bedingung 3):

```
P = (printer.get -> GetScanner),  
GetScanner = (scanner.get -> useP ->  
              printer.put -> scanner.put -> P  
              | timeoutP -> printer.put -> P) .
```

```
Q = (scanner.get -> GetPrinter),  
GetPrinter = (printer.get -> useQ ->  
              scanner.put -> printer.put -> Q  
              | timeoutQ -> scanner.put -> Q) .
```





# Beispiel: „Dinierende Philosophen“

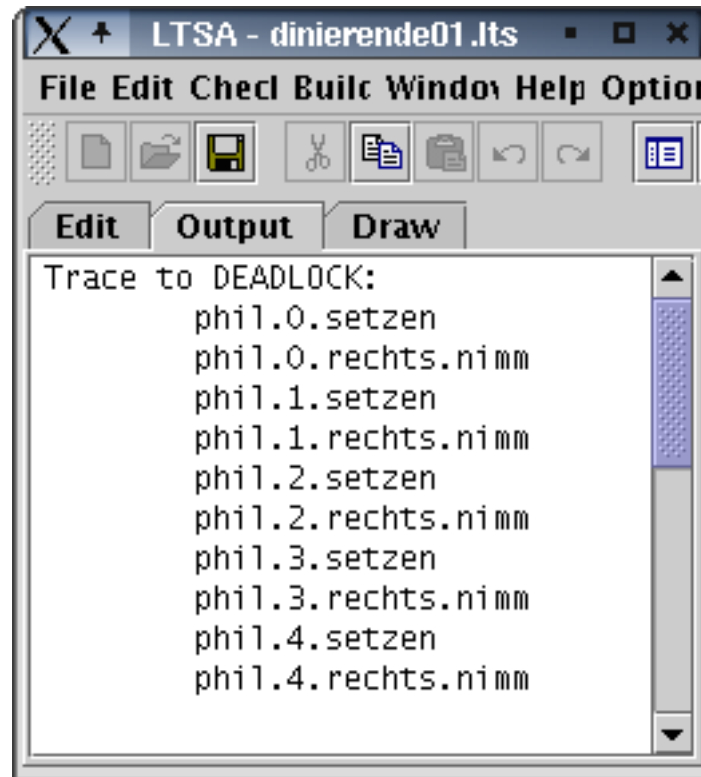
**Modellierung in FSP:** Was soll als Prozess modelliert werden?

```
GABEL = (aufnehmen -> ablegen -> GABEL) .
```

```
PHIL = (setzen -> rechts.aufnehmen -> links.aufnehmen  
        -> essen -> links.ablegen -> rechts.ablegen  
        -> aufstehen -> PHIL) .
```

```
||DINNER(N=5) = forall [i:0..N-1]  
    (phil[i]:PHIL ||  
     {phil[i].links, phil[((i-1)+N)%N].rechts}::GABEL) .
```

Analyse mit LTSA liefert den Hinweis auf einen möglichen  
Deadlock:



## Implementierung in Java: Was wird als Thread und was als Monitor implementiert?

- jeden Philosophen als eigenen Thread implementieren,
- jede Gabel als Monitor implementieren,
- die Modellierung der Gabel vorher noch überarbeiten, damit die Umsetzungsregeln für Monitore anwendbar sind.

# Modifizierte Gabel

## Originalmodellierung:

```
GABEL = (aufnehmen -> ablegen -> GABEL) .
```

## Monitortaugliche Modellierung:

```
GABEL = GABEL[0] ,  
GABEL[b:0..1] = ( when (!b) aufnehmen -> GABEL[1]  
                  | when  (b) ablegen  -> GABEL[0]) .
```

Sind die Modellierungen wirklich gleichwertig?

```
class Gabel {
    protected boolean istBelegt;
    protected final int nummer;

    ... Konstruktor

    public synchronized void aufnehmen()
        throws InterruptedException {
        while (!!istBelegt) wait();
        istBelegt = true;
        notifyAll(); // weglassen?
    }

    public synchronized void ablegen()
        throws InterruptedException {
        while (!istBelegt) wait(); // weglassen?
        istBelegt = false;
        notifyAll();
    }
}
```

```
public int getNummer() {  
    return nummer;  
}  
}
```

```

class Philosoph extends Thread {
    ...
    public Philosoph (int nummer, Gabel linkeGabel,
                     Gabel rechteGabel) {...}

    public void run() {
        while (true) {
            /* Gefahr eines Deadlocks! */
            nimmLinkeGabel();
            nimmRechteGabel();

            System.out.println(getName() + ": esse :-)");

            legeLinkeGabel();
            legeRechteGabel();
        }
    }
}

```

**Bemerkung:** Da diese Java-Implementierung von einem Deadlock-behafteten Modell abgeleitet ist, besteht auch hier die Möglichkeit eines Deadlocks!

Lassen sich in der Praxis Deadlocks beobachten?

**Frage:** Durch welche Maßnahme kann ein Deadlock ausgeschlossen werden?



# „Dinierende Philosophen“ :

## Deadlock-Vermeidung

Beispiele zur Deadlock-Vermeidung (Ausschließen einer notwendigen **Deadlock-Bedingung**):

1. Gabeln nummerieren und immer die mit der kleineren Nummer zuerst anfordern
2. Philosophen nummerieren: „gerade“ Philosophen nehmen linke Gabel zuerst, „ungerade“ die rechte
3. nach „Timeout“ die belegte Gabel zurückgeben
4. ein „Butler“ führt maximal vier Philosophen gleichzeitig an den Tisch
5. ...

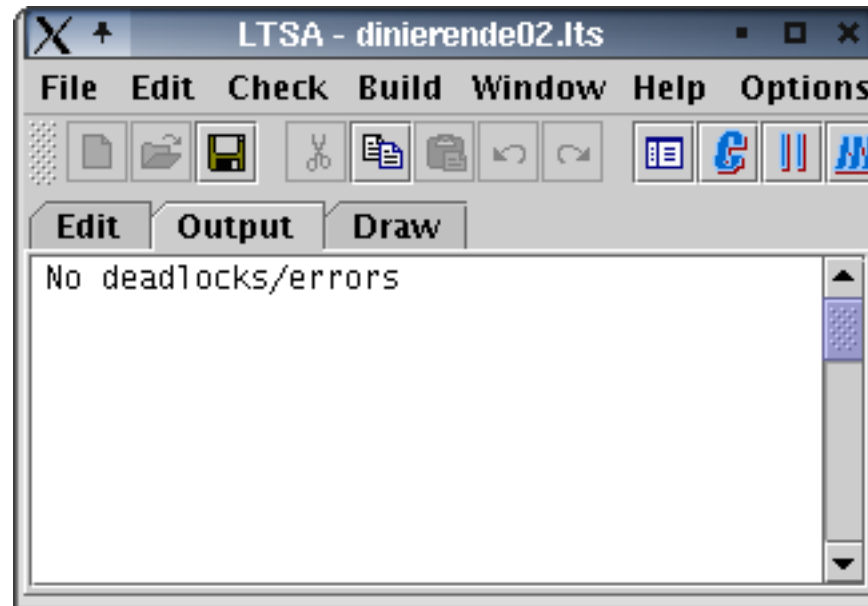
## Deadlock-freie Modellierung in FSP (Variante 2):

```
GABEL = (aufnehmen -> ablegen -> GABEL) .
```

```
PHIL(I=0) = (  
    when (I%2==0) setzen  
        -> links.aufnehmen -> rechts.aufnehmen  
        -> essen -> links.ablegen -> rechts.ablegen  
        -> aufstehen -> PHIL  
    | when (I%2==1) setzen  
        -> rechts.aufnehmen -> links.aufnehmen  
        -> essen -> links.ablegen -> rechts.ablegen  
        -> aufstehen -> PHIL) .
```

```
||DINNER(N=5) = forall [i:0..N-1]  
    (phil[i]:PHIL(i) ||  
    {phil[i].links, phil[((i-1)+N)%N].rechts}::GABEL) .
```

Analyse mit LTSA liefert wirklich die gewünschte Verklemmungsfreiheit:



# Aufgabe

1. Implementieren Sie die „**Dinierenden Philosophen**“ in Java. Wählen Sie zuerst einen naiven Ansatz, in dem Verklemmungen auftreten können und zeigen Sie diese bei der Programmausführung.
2. Erweitern Sie Ihr Programm in drei Varianten, so dass in jeder Variante jeweils eine andere der hinreichenden und notwendigen **Deadlock-Bedingungen 2–4** ausgeschaltet wird.

## 5.2 Sicherheit und Lebendigkeit

Für parallele Programme werden zwei wesentliche Eigenschaften gefordert:

**Sicherheit:** „Nichts Schlimmes wird passieren!“

**Lebendigkeit:** „Etwas Gutes wird (irgendwann) passieren!“

# Sicherheit

Sicherheitseigenschaft sequentieller Programme:

- **partielle Korrektheit**: „Wenn das Programm terminiert, dann ist das Ergebnis korrekt!“
- **totale Korrektheit**: „Das Programm terminiert immer und partielle Korrektheit ist sichergestellt!“

Schon bekannte Sicherheitseigenschaften nebenläufiger Programme:

- Abwesenheit von Interferenzen (Race Hazards),
- Abwesenheit von Verklemmungen (Deadlocks).

**Frage:** Wie können Sicherheitseigenschaften nachgewiesen werden?

**Antwort:** Durch formalen Nachweis! Model Checker (z. B. LTSA) verwenden und nicht gewünschte Zustände automatisch suchen lassen.

# Prüfen von Sicherheitseigenschaften

**Sicherheit:** „Nichts Schlimmes wird passieren!“

**Frage:** Was bedeutet das im Zustandsmodell?

**Antwort:** Eine Verletzung einer Sicherheitseigenschaft liegt vor, wenn „schlimme/unerwünschte“ Zustände erreicht werden können:

- Senken im Zustandsgraphen (entsprechen Deadlocks), Beispiel: **Ressourcenanforderung**
- explizite Fehlerzustände („ERROR“ in FSP bzw. „-1“ in LTSA), Beispiel: **unzulässige up-Operation eines Semaphors**
- ...



# Sicherheitsprüfung in LTSA

**Definition:** Durch `property P` wird in FSP eine Sicherheitseigenschaft definiert, die nur die Traces des Prozesses `P` akzeptiert.

Alle anderen möglichen Traces mit Aktionen aus dem Alphabet von `P` werden durch das System auf den Fehlerzustand `ERROR` geführt.

# Beispiel: Verwendung eines Semaphors (1)

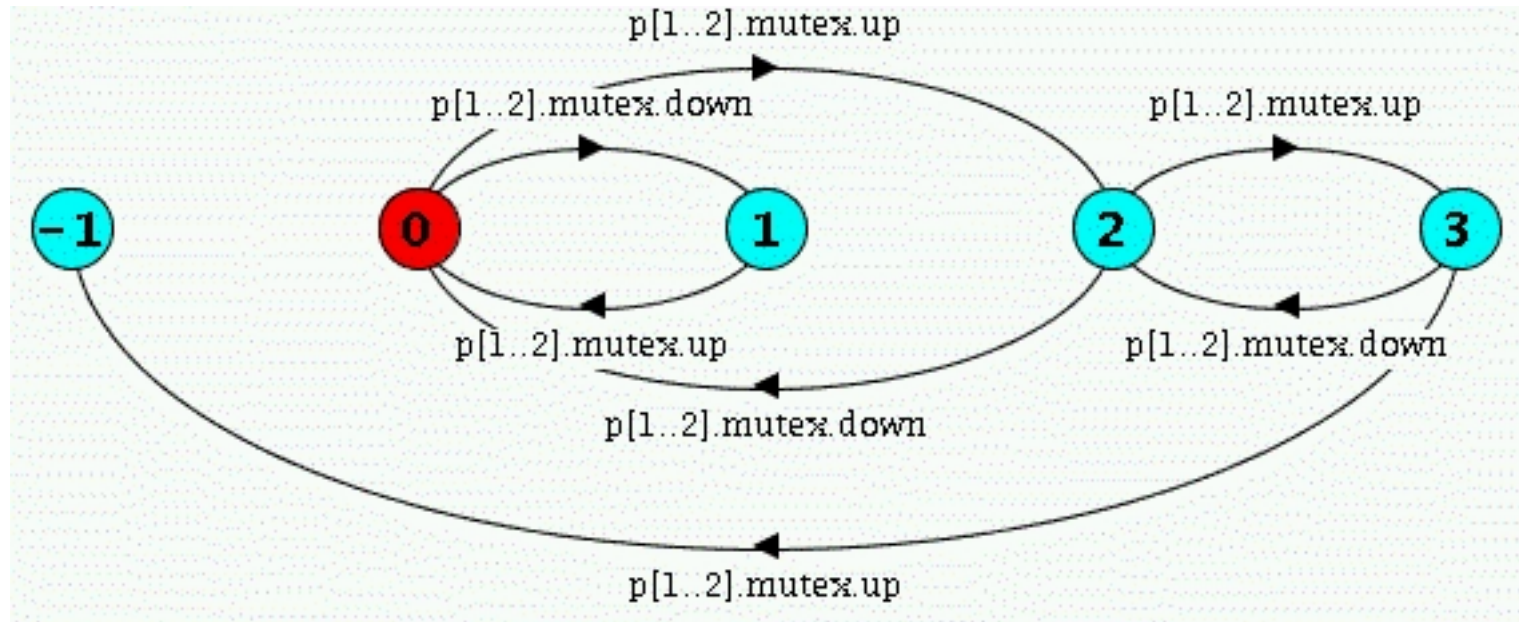
Gegenseitiger Ausschluss *ohne* Definition einer Sicherheitseigenschaft:

```
const N      = 2
const Max    = 3
range Int    = 0..Max
Semaphor(N=0) = Sema[N],
Sema[i:Int]   = ( up -> Sema[i+1]
                  | when (i>0) down -> Sema[i-1]) .

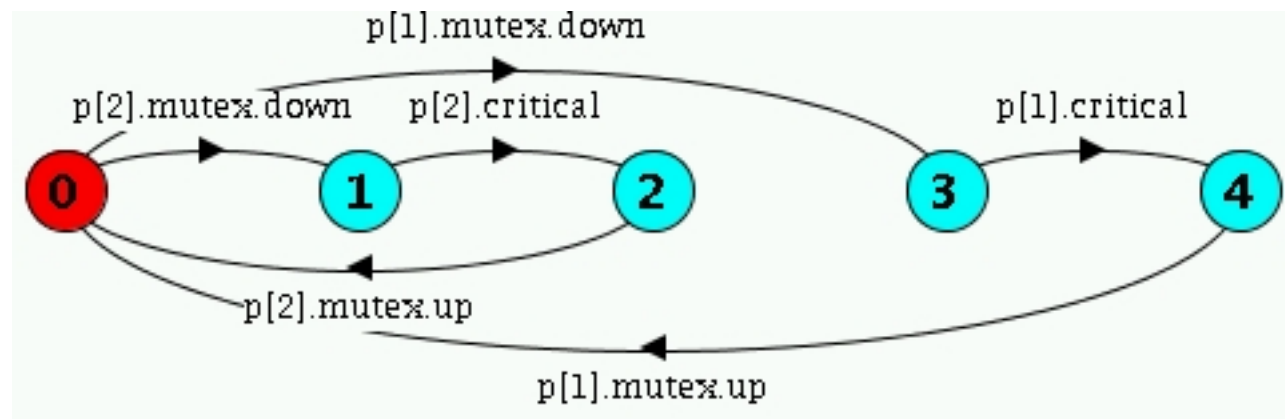
Loop = (mutex.down -> critical -> mutex.up -> Loop) .

/* ohne Sicherheitseigenschaft */
|| SemaDemo = (p[1..N]:Loop
               || p[1..N]::mutex:Semaphor(1)) .
```

$p[1..N] :: \text{mutex} : \text{Semaphor}(1) :$



|| SemaDemo :



## Fragen:

Ist wirklich sichergestellt, dass ...

- ... nur ein Prozess den kritischen Bereich betritt (keine zwei Prozesse führen `down` aus)?
- ... jeder Prozess nach `down` auch `up` ausführt?
- ... nur der Prozess die Aktion `up` aufruft, der zuvor `down` ausgeführt hat?

# Verwendung eines Semaphors (2)

**Sicherheitseigenschaft:** „Erst down dann up ausführen“

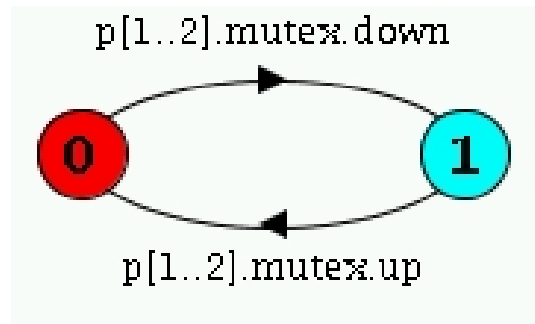
```
const N      = 2
const Max    = 3
range Int    = 0..Max
Semaphor(N=0) = Sema[N],
Sema[i:Int]   = ( up -> Sema[i+1]
                  | when (i>0) down -> Sema[i-1]) .
```

```
Loop = (mutex.down -> critical -> mutex.up -> Loop) .
```

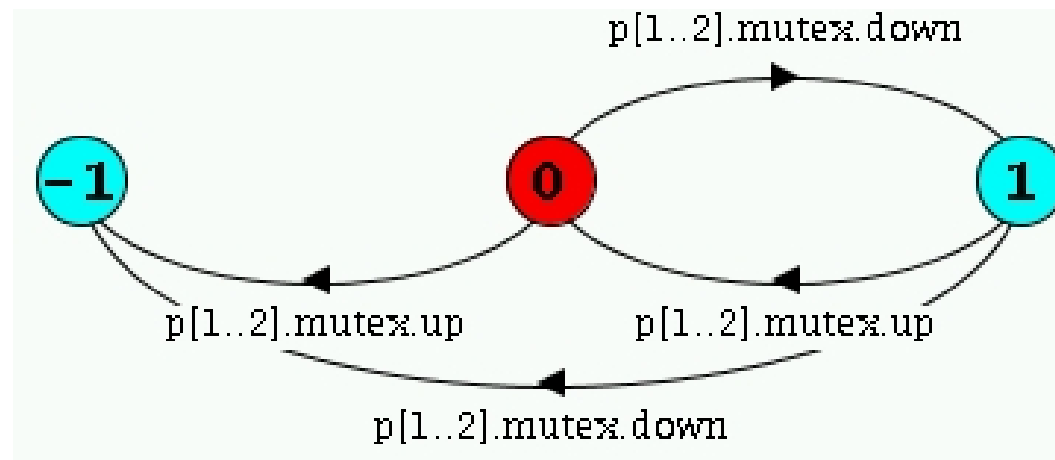
```
property OK = (mutex.down -> mutex.up -> OK) .
```

```
|| SemaDemo = (p[1..N]:Loop || p[1..N]::mutex:Semaphor(1)
               || p[1..N]::OK) .
```

Prozess  $p[1..N] :: \text{OK}$  *ohne* property-Schlüsselwort:



Prozess  $p[1..N] :: \text{OK}$  *mit* property-Schlüsselwort:



**Bemerkung:** Durch die vorher definierte Property ist auch sichergestellt, dass nur ein Prozess den kritischen Bereich betritt, also keine zwei Prozesse direkt nacheinander `down` ausführen (vgl. [Zustandsgraph](#)).

# Verwendung eines Semaphors (3)

**Sicherheitseigenschaft:** „Erst down dann up ausführen und Prozessreihenfolge beachten“:

```
const N      = 2
const Max    = 3
range Int    = 0..Max
Semaphor(N=0) = Sema[N],
Sema[i:Int]   = ( up -> Sema[i+1]
                  | when (i>0) down -> Sema[i-1]) .
```

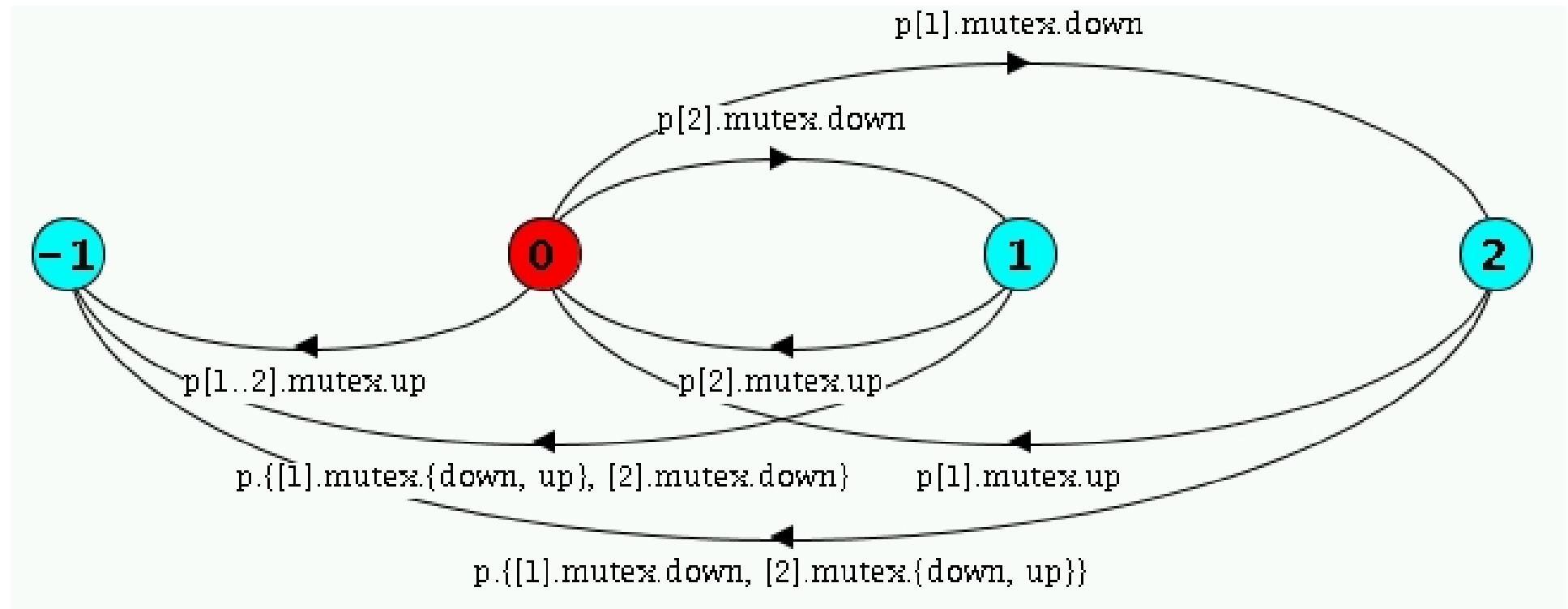
```
Loop = (mutex.down -> critical -> mutex.up -> Loop) .
```

```
property OK = (p[i:1..N].mutex.down -> p[i].mutex.up -> OK) .
```

```
||SemaDemo = (p[1..N]:Loop || p[1..N]::mutex:Semaphor(1)
              || OK) .
```

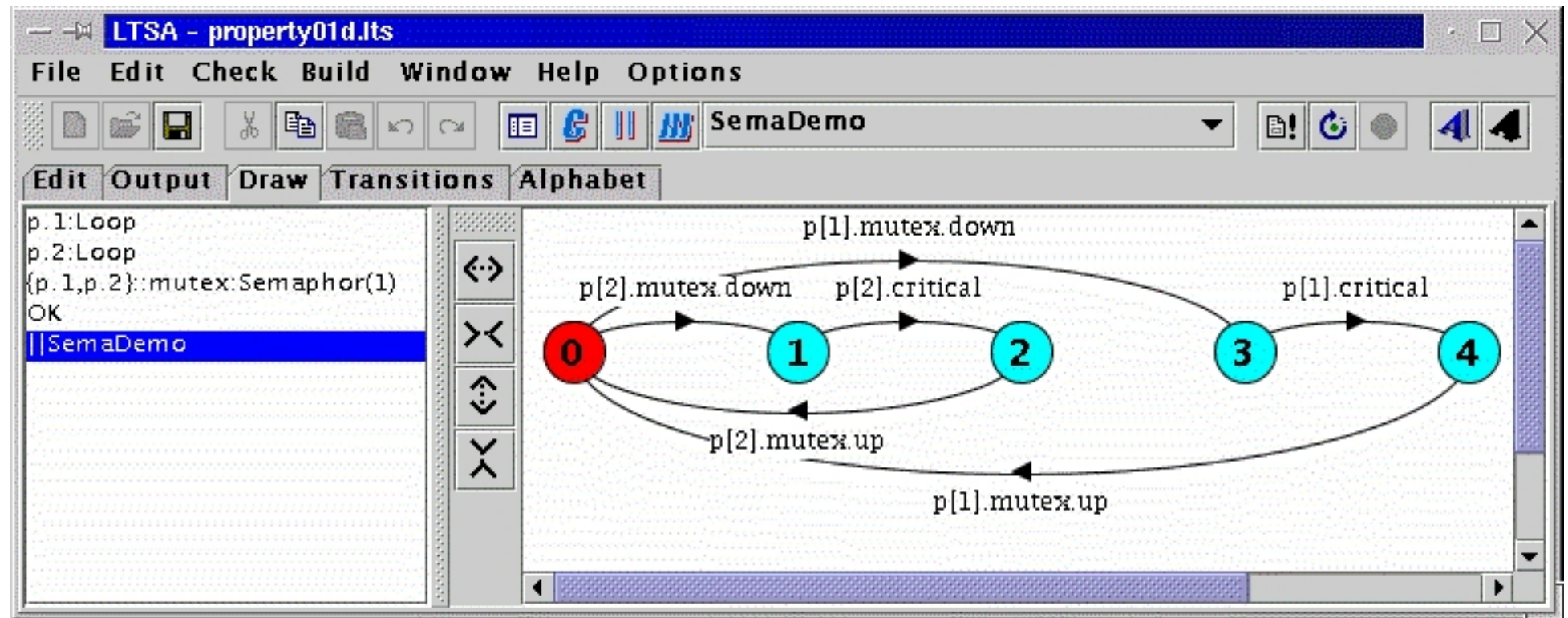


# Zustandsgraph der Sicherheitseigenschaft

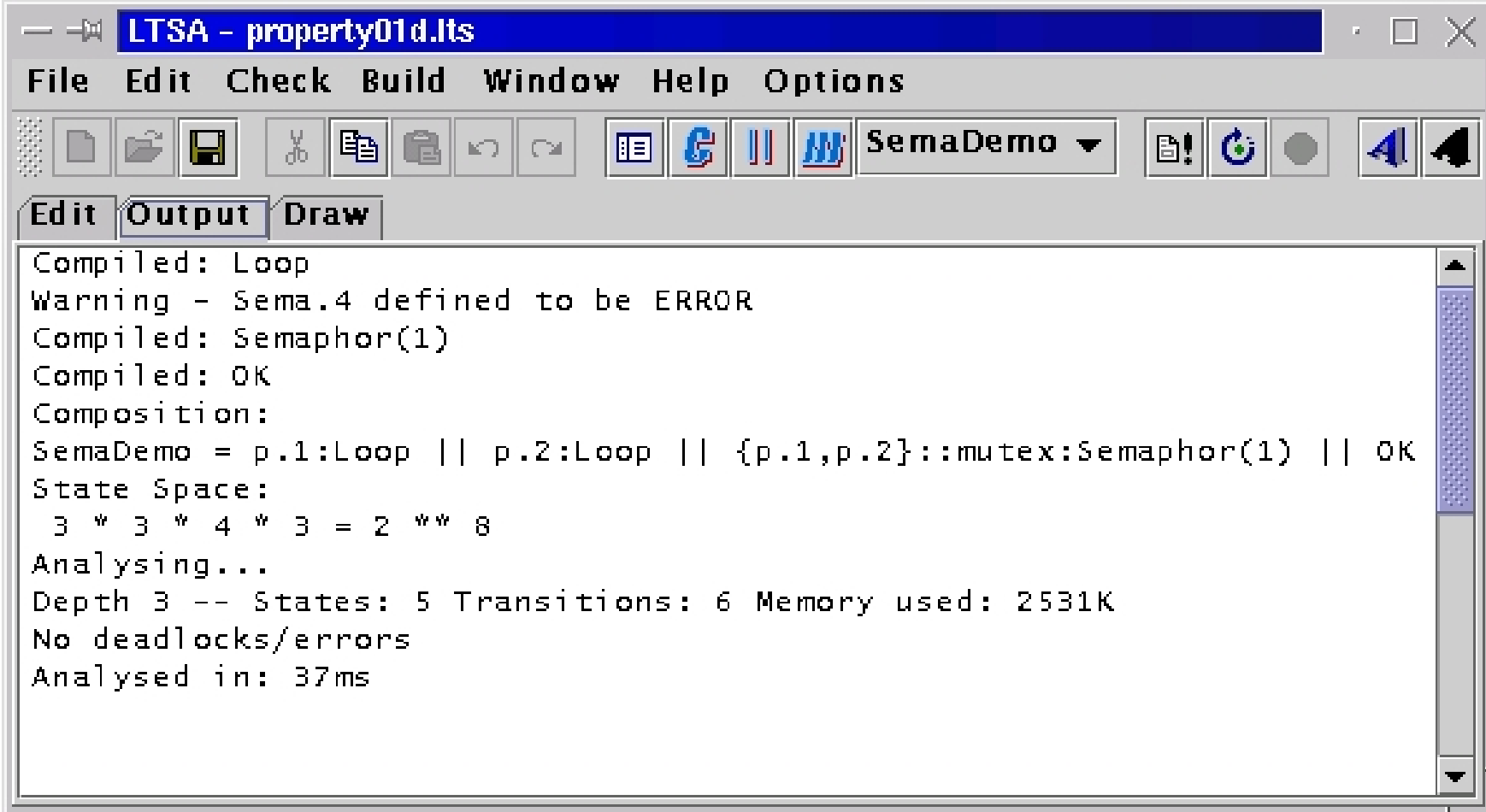


up- und down-Aktionen in der „falschen“ Reihenfolge oder von „falschen“ Prozessen werden explizit auf einen Fehlerzustand geführt.

Der Zustandsgraph des Gesamtprozesses SemaDemo zeigt keinen Unterschied zur **Originalfassung** ohne Sicherheitseigenschaft:



Eine Überprüfung mit LTSA meldet keine Verletzung der Sicherheitseigenschaft:



The screenshot shows the LTSA application window with the title bar 'LTSA - property01d.lts'. The menu bar includes 'File', 'Edit', 'Check', 'Build', 'Window', 'Help', and 'Options'. The toolbar contains various icons for file operations and analysis. The 'Output' tab is selected, displaying the following text:

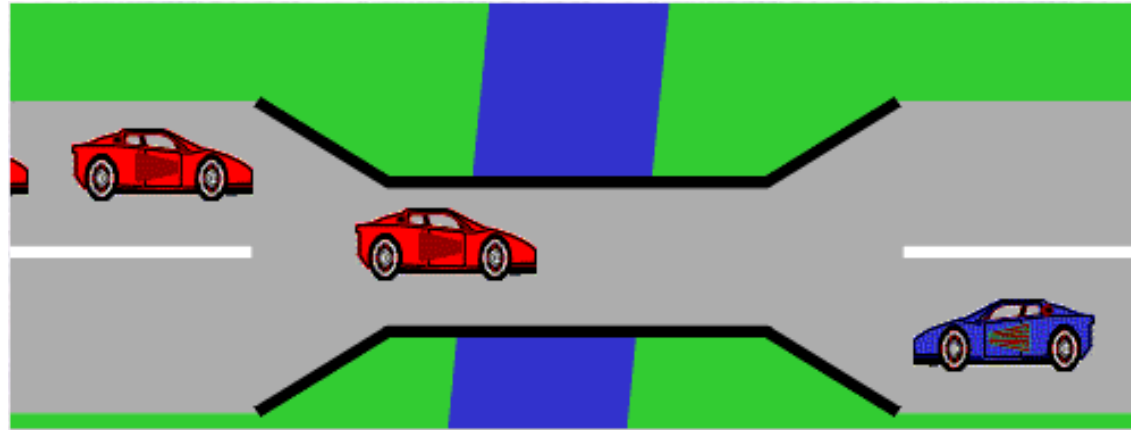
```
Compiled: Loop
Warning - Sema.4 defined to be ERROR
Compiled: Semaphor(1)
Compiled: OK
Composition:
SemaDemo = p.1:Loop || p.2:Loop || {p.1,p.2}::mutex:Semaphor(1) || OK
State Space:
  3 * 3 * 4 * 3 = 2 ** 8
Analysing...
Depth 3 -- States: 5 Transitions: 6 Memory used: 2531K
No deadlocks/errors
Analysed in: 37ms
```

# Modifikationen

1. Initialisierung des Semaphors mit `Semaphor(2)`.
2. Änderung des Wächters innerhalb von `Sema` auf `when (i >= 0)`.
3. ...

Wie wirken sich die Änderungen auf die Sicherheitseigenschaft aus?

# Beispiel: „Einspurige Brücke“



Über einen Fluss führt eine einspurige Brücke. Wenn Fahrzeuge in einer Richtung die Brücke passieren, muss der Gegenverkehr warten.

Zur Vereinfachung: „Rote“ Fahrzeuge fahren von „links nach rechts“, „blaue“ von „rechts nach links“.

Eine Verletzung der Sicherheitseigenschaft tritt auf, wenn zwei Fahrzeuge die Brücke gleichzeitig in entgegen gesetzter Richtung befahren.<sup>10</sup>

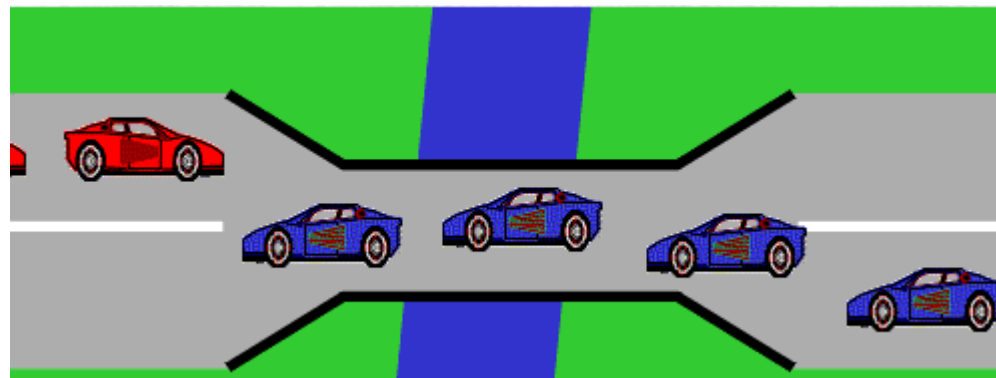
<sup>10</sup> Achtung: Auf dem Bild herrscht Linksverkehr!

# „Einspurige Brücke“ – 1. Ansatz

**Idee:** Regele die Zufahrt über **gegenseitigen Ausschluss**:  
Nur ein Fahrzeug zurzeit darf die Brücke befahren.

Lösung ist o.k., aber bei viel Verkehr wird die Brücke schlecht ausgenutzt.

**Abhilfe:** Die Brücke von mehreren Fahrzeugen gleichzeitig *in derselben* Richtung befahren lassen!



## „Einspurige Brücke“ – 2. Ansatz

**Idee:** Verallgemeinerung des gegenseitigen Ausschlusses.

Gegenseitiger Ausschluss gilt nur zwischen „roten“ und „blauen“ Fahrzeugen. Fahrzeuge derselben Farbe dürfen die Brücke gleichzeitig befahren.

**Umsetzung:** Lösung zunächst in FSP modellieren und dann in Java umsetzen.

```
const N = 3          // Anzahl Autos einer Farbe
range T = 0..N       // Wertebereich des Zaehlers
range ID = 1..N      // Ids der Autos
```

```
Auto=(rauf -> runter -> Auto) .
```

```
Bruecke=Bruecke[0][0], // [rote Autos][blaue Autos]
Bruecke[r:T][b:T]=( when (b==0)
                    rot[ID].rauf -> Bruecke[r+1][b]
                    | rot[ID].runter -> Bruecke[r-1][b]
                    | when (r==0)
                    blau[ID].rauf -> Bruecke[r][b+1]
                    | blau[ID].runter -> Bruecke[r][b-1] ) .
```

```
||Konvoi=([ID]:Auto) .
```

```
||Autos=(rot:Konvoi || blau:Konvoi) .
```

```
||EinspurigesBefahren=(Autos || Bruecke) .
```



# Testen des Modells

- Testen des Modells mit LTSA: Funktioniert der gegenseitige Ausschluss?
- Ja, aber Fahrzeuge derselben Farbe können sich auf der Brücke überholen!

**Abhilfe:** Definiere zwei Prozesse, die die Ein- und Ausfahrtreihenfolge der Fahrzeuge erzwingen.

# Erweiterung: Kein Überholen auf der Brücke

...

```
ReiheRauf=DarfRauf[1],  
DarfRauf[i:ID]=([i].rauf -> DarfRauf[i%N+1]).
```

```
ReiheRunter=DarfRunter[1],  
DarfRunter[i:ID]=([i].runter -> DarfRunter[i%N+1]).
```

```
||Konvoi=([ID]:Auto || ReiheRauf || ReiheRunter).
```

...

Testen in LTSA: o.k.

**Frage:** Kann es jetzt wirklich zu keinen Kollisionen auf der Brücke kommen? → Sicherheitseigenschaften definieren!

# Definition der Sicherheitseigenschaft

„Wenn sich rote Fahrzeuge auf der Brücke befinden, dürfen nur rote Fahrzeuge auf die Brücke fahren.

Wenn sich blaue Fahrzeuge auf der Brücke befinden, dürfen nur blaue Fahrzeuge auf die Brücke fahren.

Wenn die Brücke leer ist, darf entweder ein rotes oder blaues Fahrzeug auf die Brücke fahren.“

Realisiert in FSP als `property MUTEX_ROTBLAU`

# Sicherheitseigenschaft in FSP

...

```
property MUTEX_ROTBLAU=( rot[ID].rauf -> ROT[1]
                        | blau[ID].rauf -> BLAU[1] ),
ROT[i:ID]=( rot[ID].rauf -> ROT[i+1]
            | when (i==1) rot[ID].runter -> MUTEX_ROTBLAU
            | when (i>1) rot[ID].runter -> ROT[i-1] ),
BLAU[i:ID]=( blau[ID].rauf -> BLAU[i+1]
            | when (i==1) blau[ID].runter -> MUTEX_ROTBLAU
            | when (i>1) blau[ID].runter -> BLAU[i-1] ).
```

...

```
||EinspurigesBefahren=(Autos || Bruecke || MUTEX_ROTBLAU) .
```

# Überprüfen der Sicherheitseigenschaft

Definiere zusätzlich zur Kontrolle einen Prozess `Crash`, in dem der gegenseitige Ausschluss durch `Bruecke` nicht gewährleistet ist:

```
|| Crash = (Autos || MUTEX_ROTBLAU) .
```

Mit LTSA die Sicherheitseigenschaft für die Prozesse `EinspurigesBefahren` und `Crash` überprüfen ...

# Umsetzung in Java

Ableiten aus der Modellierung von **Bruecke**:

```
class SafeBridge {  
  
    // number of red or blue cars on the bridge  
    private int nred  = 0;  
    private int nblue = 0;  
  
    /*  
        Monitor invariant:  
        =====  
        nred >= 0 and nblue >= 0 and  
        not (nred > 0 and nblue > 0)  
    */  
}
```

```
synchronized void redEnter()  
    throws InterruptedException {  
    while (!nblue==0) { wait(); }  
    ++nred;  
    // notifyAll(); // forgotten or o. k.?  
}
```

```
synchronized void redExit() {  
    --nred;  
    // notifyAll(); // next lines instead is o. k.?  
    if (nred==0) {  
        notifyAll();  
    }  
}
```

```
synchronized void blueEnter() ...  
synchronized void blueExit() ...  
}
```

# Testen der Implementierungen . . .