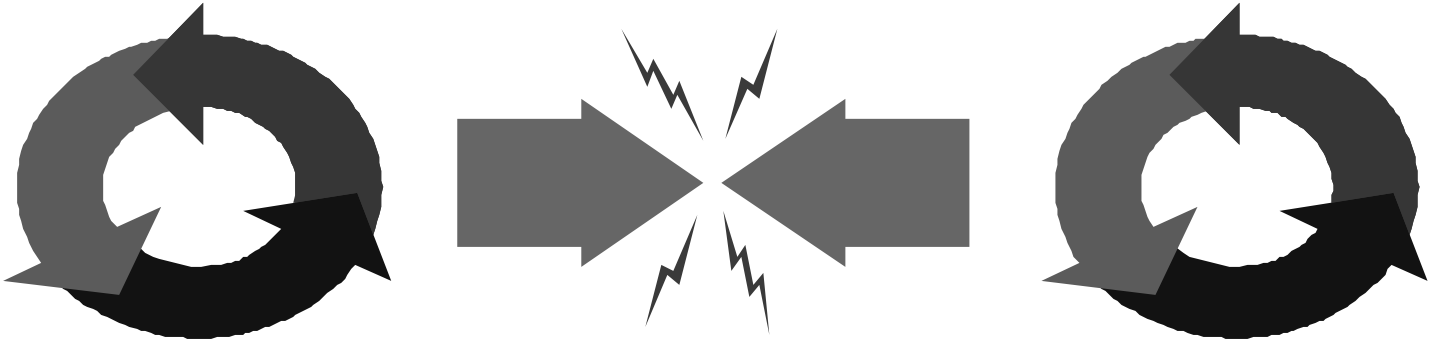
Sicherheit,

geteilte Objekte

und wechselseitiger Ausschluß



Übersicht

- Sicherheit, Lebendigkeit - Begriffsbestimmung
- Beispiel: Ornamental Garden
  - Interferenz von Prozessen, wechselseitiger Ausschluß
- FSP Modellierung und Analyse
  - Testen, Komposition mit Fehlerdetektionsprozeß, erschöpfende Suche
- Wechselseitiger Ausschluß in Java
- Patterns für sichere Objekte
  - unveränderliche Objekte (immutable objects)
  - voll synchronisierte Objekte (fully synchronized objects)
  - gekapselte Objekte (contained objects)

Sicherheits - und Lebendigkeitseigenschaften

- Sicherheit:
  - eine Eigenschaft, die zusichert, daß nichts „Böses“ passieren wird
  - „Böses“ meist „fehlerhaftes Verhalten“
  - falls eine Sicherheitseigenschaft verletzt wird, so ist der Effekt nicht mehr reparierbar
  - „things just start going wrong“
- Lebendigkeit:
  - eine Eigenschaft, die zusichert, daß etwas „Gutes“ irgendwann eintreten kann
  - umgekehrte Sichtweise, alternative Formulierung: es ist nícht möglich, eine Situation herbeizuführen, in der das „Gute“ für alle Zeit ausgeschlossen ist.
  - „things just stop running“
- viele Eigenschaften sind Mischformen dieser Extrema
- formale Behandlung: B. Alpern, F. Schneider: Defining Liveness, Information Processing Letters 21 (1985), 181-185

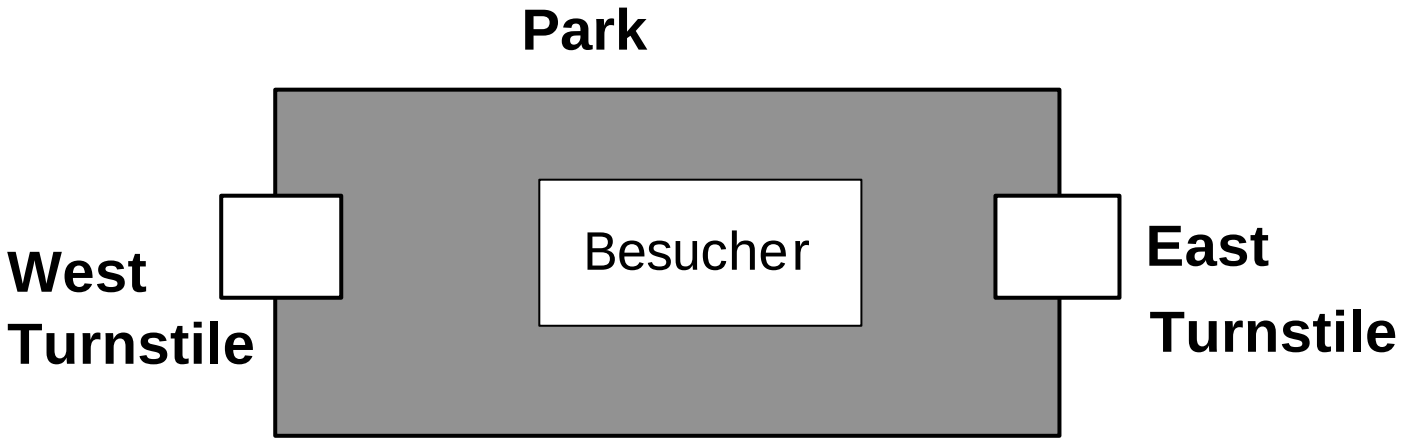
Sicherheits- und Lebendigkeitseigenschaften

- Sicherheit ist primäres Ziel
  - da fehlerhafte Berechnungen nutzlos bis gefährlich sind, sollte ein Programm im Zweifel eher abbrechen, als bei Fehlern fortfahren
- Lebendigkeit steht eher im Zusammenhang mit Performance, Verfügbarkeit und Durchführbarkeit von Berechnungen
- Thread Programmierung muß beides berücksichtigen
- Hauptziel bei Sicherheit
  - alle Objekte eines Systems müssen sich in einem konsistenten Zustand befinden
  - konsistent im Sinne von: Daten haben jeweils legale, sinnvolle Werte
- Blickpunkt liegt hier nicht auf allgemeiner Korrektheit eines Programms, sondern auf den Einflüssen des Interleavings in der Ausführung eines Programms mit mehreren Threads auf das berechnete Ergebnis
  - Interferenz von Prozessen, Konsistenz von Daten
- Problem tritt insbesondere bei geteilten Objekten auf

Interferenz am Beispiel Ornamental Garden

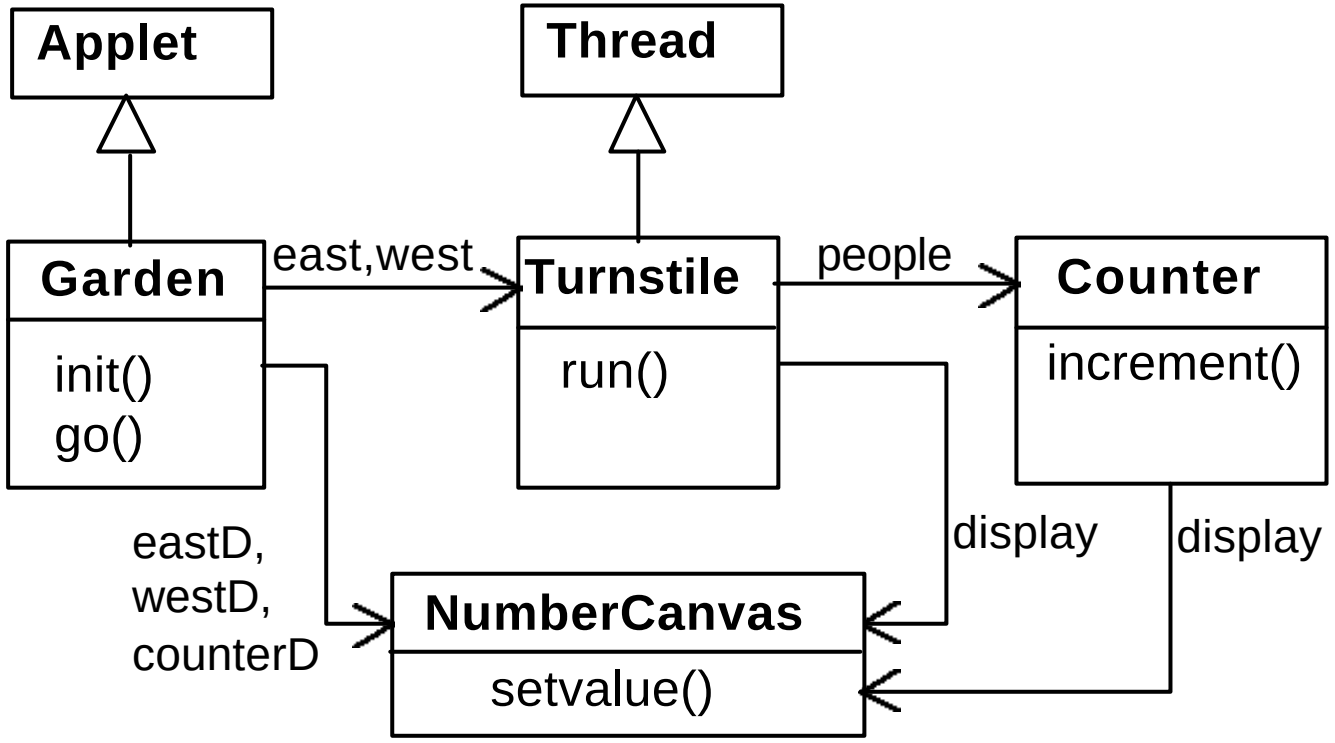
Ornamental Garden Problem:

Besucher betreten einen Park durch zwei Eingänge mit Drehkreuzen. Die Verwaltung möchte zu jedem Zeitpunkt über die aktuelle Anzahl Besucher informiert sein.



Das nebenläufige Programm besteht aus zwei Threads und einem geteilten Objekt zur Zählung der Besucher.

Ornamental Garden Programm - Klassendiagramm



Der **`Turnstile`** Thread simuliert periodische Ankünfte von Besuchern (1 je Sekunde) durch Warten und Aufruf der Methode **`increment()`** für das Counter Objekt.

Ornamental Garden Programm - Code

`Counter` Objekt und **`Turnstile`** Threads werden durch Methode **`go()`** des Garden Applets erzeugt:

```
    private void go() {
      counter = new Counter(counterD);
      west = new Turnstile(westD,counter);
      east = new Turnstile(eastD,counter);
      west.start();
      east.start();
    }
```

`counterD`, **`westD`** und **`eastD`** sind Objekte aus **`NumberCanvas`**.

counter ist ein geteiltes Objekt, das beide Threads kennen.

Turnstile Class

```
class Turnstile extends Thread {
  NumberCanvas display;
  Counter people;

  Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

  public void run() {
    try{
      display.setvalue(0);
      for (int i=1;i<=Garden.MAX;i++){
        Thread.sleep(500); //0.5 s zwischen 2 Ankünften
        display.setvalue(i);
        people.increment();
      }
    } catch (InterruptedException e) {}
  }
}
```

Die run() Methode terminiert (und damit auch der Thread) nachdem **`Garden.MAX`** Besucher eingetreten sind.

Counter Class

```
class Counter {
  int value=0;
  NumberCanvas display;

  Counter(NumberCanvas n) {
    display=n;
    display.setvalue(value);
  }

  void increment() {
    int temp = value;     //read value
    ...
    value=temp+1;         //write value
    display.setvalue(value);
  }
}
```

Counter zählt Aufrufe und zeigt den jeweiligen Wert im Display an.

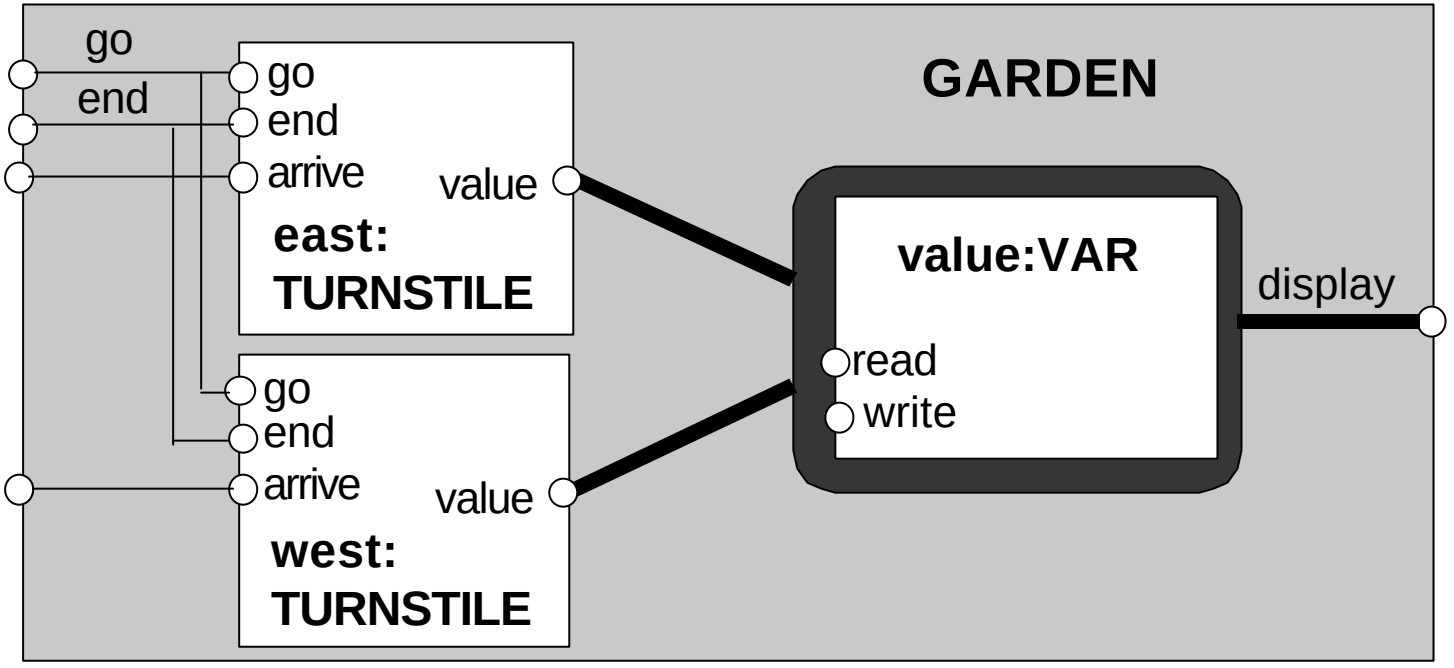
Achtung:
Hardware Interrupts und Unterbrechungen durch Betriebssystem Scheduler können jederzeit zwischen Befehlen auftreten.

Ornamental Garden Programm - Display

Nach vielen erfolgreichen Aufrufen tritt plötzlich und unerwartet folgender tragischer Fall auf:

Nachdem East und West Turnstile Threads jeweils 20 mal ein increment() durchgeführt haben, zeigt der Zähler nicht die Summe der beiden Turnstile Aufrufe! Einige Zählererhöhungen sind verloren gegangen, warum?

Ornamental Garden: FSP Modell



Der Prozeß **VAR** modelliert Lese- und Schreibzugriffe auf den geteilten Zähler value.

Increment() intern in **`TURNSTILE`** modelliert, weil Aktivierung von Java Methoden nicht atomar.

Interleaving bei Lese- und Schreibzugriffen

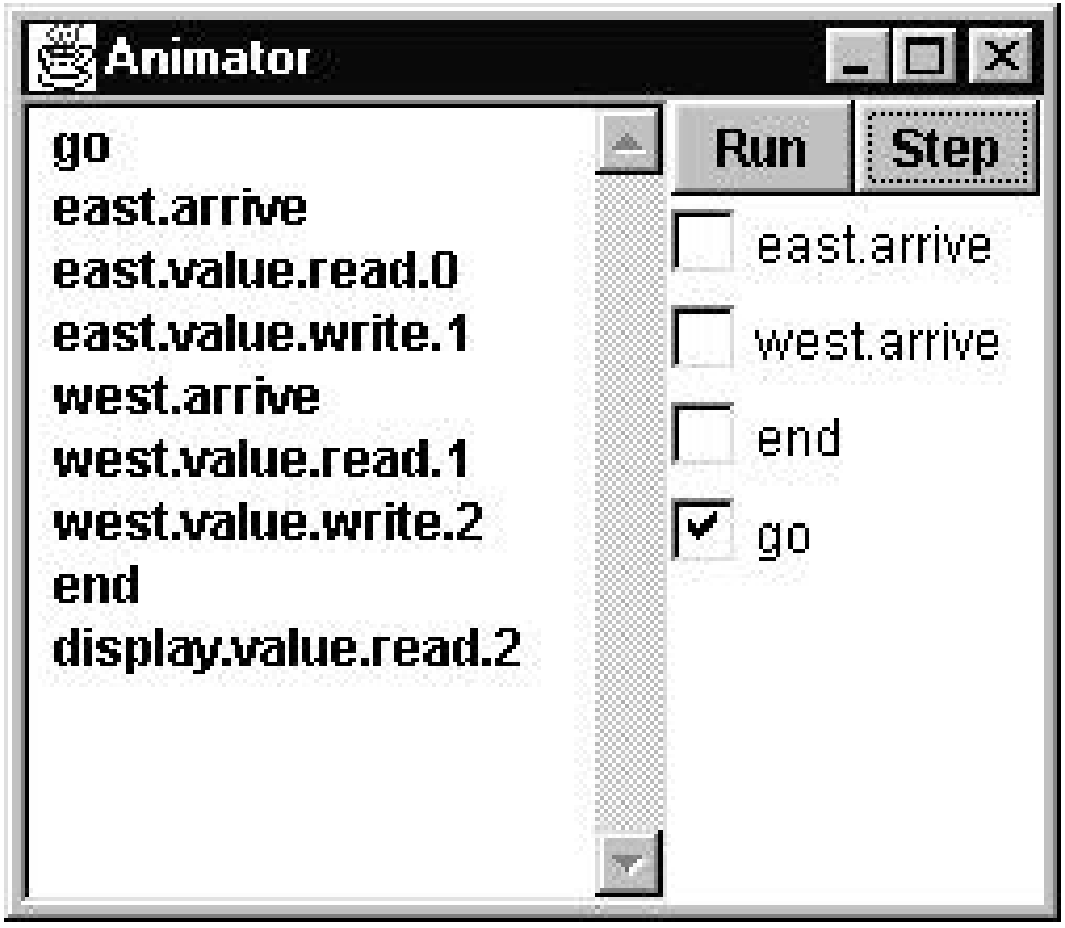
Ornamental Garden: FSP Modell

```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

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

TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
             |end   -> TURNSTILE),
INCREMENT = (value.read[x:T]
              -> value.write[x+1]->RUN
             )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || { east,west,display} ::value:VAR)
            /{ go /{ east,west} .go,
              end/{ east,west} .end} .
```

Alphabet des Prozesses **VAR** explizit als **set** Konstante, **VarAlpha**.

Alphabet von **TURNSTILE** um **VarAlpha** erweitert damit keine freien Aktionen in **VAR** auftreten, dh alle Aktionen in **VAR** sind von **TURNSTILE kontrolliert**.

Fehlersuche mittels Testen - Animation



Testen eines Szenario - erzeuge einen Trace.

Ist dieser Trace korrekt?

(Ergänzung)

Erzeugung von Abläufen durch Ableitungsregeln für Transitionsrelation

Beispiel: parallele Komposition

sei $P \neq \Pi, Q \neq \Pi \qquad a \in A$ Π ist ERROR Prozeß

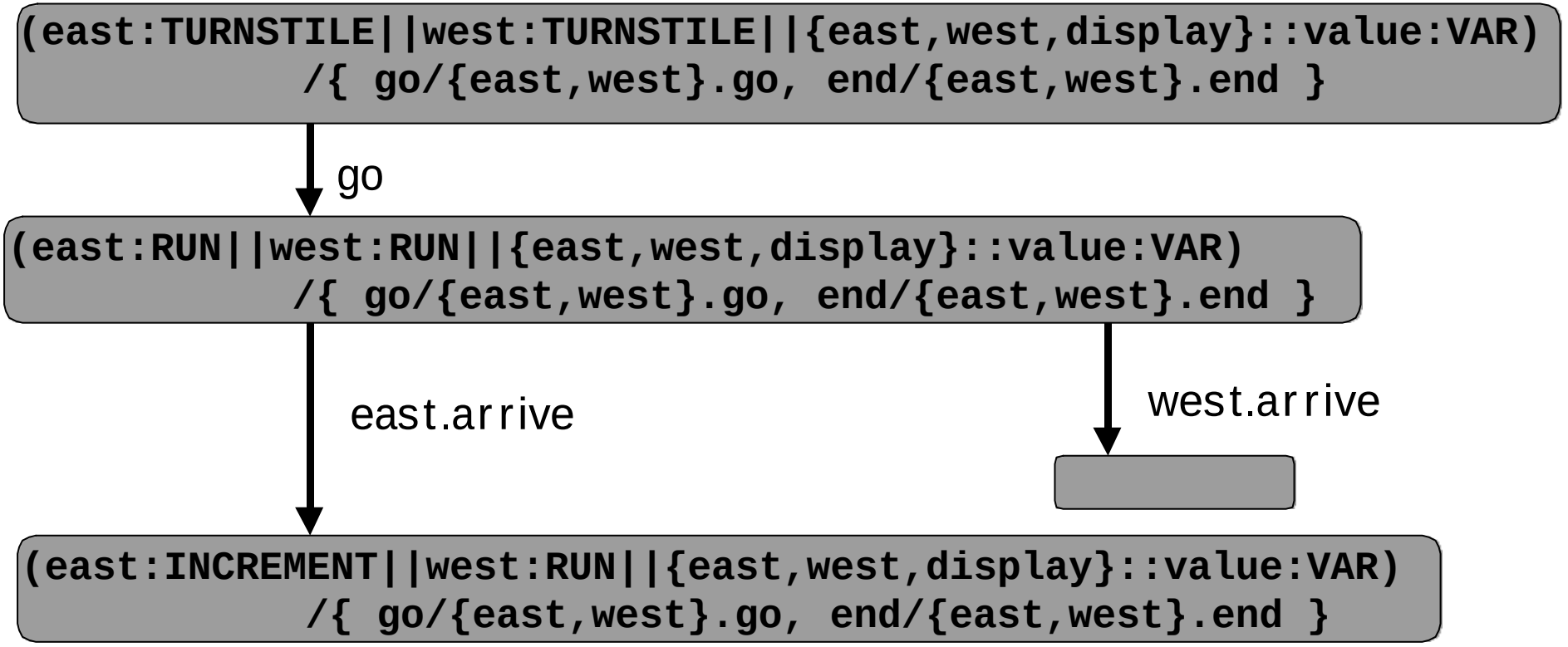
Für P||Q wird Δ durch folgende Regeln erzeugt:

$$\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} a \notin aQ \qquad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'} a \notin aP$$

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'} a \neq t$$

(Ergänzung)

Beispielberechnung zur Transitionsrelation



Nachfolgerberechnung durch textuelle Manipulation der Prozeßbeschreibung textuelle Ersetzung gemäß Regeln für ->,|,|| und Rekursion.

Fehlersuche mittels erschöpfender Suche

Das Modell wird mit einem TEST Prozeß komponiert, der die Summe der Ankünfte mit dem angezeigten Wert vergleicht:

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

Wie **STOP, nur ERROR** ist ein vordefinierter FSP Prozeß mit fester Zustandsnummer -1 im LTS.

Algorithmus zur erschöpfenden Suche nach ERROR Zuständen in einem LTS

- s ist initialer Zustand, $S = N = \{s\}$
- while ($N \neq \{\}$) do
  - entferne s aus N
  - teste s auf ERROR (z.B. keine Nachfolger), ggfs -> STOP
  - for all $(s, x, s') \in \Delta$
    - if $s' \notin S$
    - then $S = S \cup \{s'\}; N = N \cup \{s'\}$
  - od
- od

Resultat: Menge aller erreichbaren Zustände S, bzw Erreichbarkeit ERROR
Wesentliche Anforderungen: a) Nachfolgerelation, b) Test auf Gleichheit.

Datenstruktur für N: bei DFS Stack, bei BFS Queue

Crux: Platzbedarf, eff. Such-/Einfügeoperation bei großem S und N.

Ornamental Garden Modell - Fehlersuche

`||TESTGARDEN = (GARDEN || TEST).`

Mit ***LTSA*** erschöpfende Suche nach **ERROR**.

```
Trace zum Fehlerfall in TEST:
   go
   east.arrive
   east.value.read.0
   west.arrive
   west.value.read.0
   east.value.write.1
   west.value.write.1
   end
   display.value.read.1
   wrong
```

LTSA produziert den kürzesten Pfad zum **ERROR Zustand.**
BFS oder DFS?

Counter class
(kompletter Code zur Provokation des Problems)

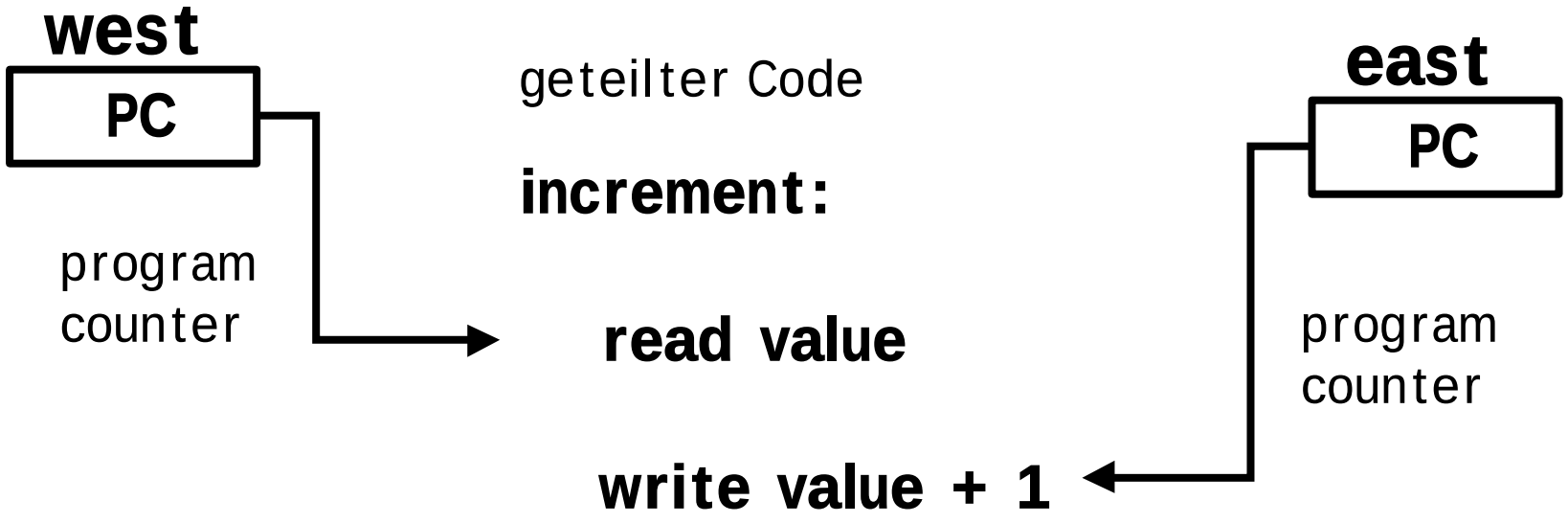
```
class Counter {
  int value=0;
  NumberCanvas display;

  Counter(NumberCanvas n) {
    display=n;
    display.setvalue(value);
  }

  void increment() {
    int temp = value;     //read value
    Simulate.HWinterrupt();
    value=temp+1;         //write value
    display.setvalue(value);
  }
}
```

Hardware Interrupts können jederzeit auftreten.

Der **`counter`** simulatiert einen HW Interrupt innerhalb von **`increment()`** zwischen Lese- und Schreibzugriffen auf die geteilte Variable **`value`**. Die Interrupt-Methode ruft zufallsgesteuert **`Thread.yield()`** auf, um einen Threadwechsel zu stimulieren.

Aktivierung nebenläufiger Methoden

Die Aktivierung von Java Methoden ist nicht atomar- Threads East und West können gleichzeitig den Code für die Inkrement Methode ausführen und zwar für ein und dieselbe Instanz des Counters!



Interferenz und wechselseitiger Ausschluß

Ein destruktives Update, das durch beliebiges Interleaving aus Lese- und Schreiboperationen erzeugt wird, heißt Interferenz.

Fehler durch Interferenz sind schwer zu lokalisieren. Generelle (aber auch sehr restriktive) Lösung : Methoden dürfen nur im wechselseitigen Ausschluß auf geteilte Objekte zugreifen. Wechselseitiger Ausschluß kann durch atomare Aktionen modelliert werden.

(klassisches Thema in Vorlesung Betriebssysteme)

Wechselseitiger Ausschluß in Java

Das Schlüsselwort **`synchronized`** in Java bewirkt einen wechselseitigen Ausschluß bzgl der Methoden eines Objektes.

Korrektur für **`COUNTER`** Class durch Vererbung und synchronized Attribut für die Methode increment():

```
class SynchronizedCounter extends Counter {

  SynchronizedCounter(NumberCanvas n)
     {super(n);}

   synchronized void increment() {
        super.increment();
   }
}
```

Java: synchronized

Exklusiver Zugriff auf ein Objekt kann auch lokal durch Access to an **`synchronized`** hergestellt werden:

```
synchronized (object) { statements }
```

Eine weniger elegante Korrektur des Beispiels wäre die Veränderung der **`Turnstile.run()`** Methode:

```
synchronized(counter) {counter.increment();}
```

Warum ist dies "nicht elegant" ?

Um den wechselseitigen Ausschluß für den Zugriff auf ein Objekt zu sichern, sollten **alle Methoden des Objektes** synchronisiert werden.

Modelierung eines wechselseitigen Ausschlusses

Wir definieren einen **LOCK** Prozeß, komponieren in mit Prozeß **VAR** und modifizieren das Alphabet:

```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

set VarAlpha = {value.{read[T],write[T],
                       acquire, release}}
```

Wir ergänzen **TURNSTILE** um die Belegung/Freigabe des Locks:

```
TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
            |end   -> TURNSTILE),
INCREMENT = (value.acquire
             -> value.read[x:T]->value.write[x+1]
             -> value.release->RUN
            )+VarAlpha.
```

Wechselseitiger Ausschluß im Beispielproblem

Java kann mit jedem Objekt einen ***lock*** assoziieren (***synchronized***). Bei Aufruf einer Methode wird zusätzlicher Code zur Belegung (Aquire, vor Ausführung) und Freigabe (Release nach Ausführung und vor Return) des Locks durchlaufen. Nebenläufige Prozesse müssen ggfs bei Belegung zunächst auf Freigabe des Locks warten.

Verbessertes Ornamental Garden Modell - Fehlersuche

Animation eines

Beispiel Traces

```
go
east.arrive
east.value.acquire
east.value.read.0
east.value.write.1
east.value.release
west.arrive
west.value.acquire
west.value.read.1
west.value.write.2
west.value.release
end
display.value.read.2
right
```

Mittels TEST und ***LTSA*** erschöpfende Suche:

Ist TEST erfüllt?

Zwischenfazit

- Problem durch Interferenz erkannt
- durch wechselseitigen Ausschluß mittels synchronized behandelt
- Grundsätzliche Lösung: alles synchronisieren ?
  - Verhindert parallelität
  - erzeugt vermutlich Deadlocks
- Patterns nach Lea (Kap. 2, Safety)
  - unveränderliche Objekte
    - Idee: Zustandsänderungen vermeiden
  - voll synchronisierte Objekte
    - Idee: dynamisch exklusiven Zugriff sicherstellen
  - gekapselte Objekte
    - Idee: strukturell exklusiven Zugriff sicherstellen

Unveränderliche Objekte

- Grundidee: Variable eines Objektes werden nur bei Erzeugung gesetzt, jedoch von keiner Methode geändert.
- Für diesen Spezialfall existiert kein Problem durch Interferenz
- Dafür sehr begrenzter Einsatzbereich
  - Objekte als Instanzen einfacher Datentypen für bestimmte Werte, z.B. Farben (java.awt.Color), Zahlen, Zeichenketten
  - unterschiedliche Klassen je nach Nutzung: java.lang.String unveränderlich, aber java.lang.StringBuffer ist veränderbar
  - falls Erzeugung von unterschiedlichen Varianten/Versionen/Zustände durch Erzeugung neuer Objekte als partielle Kopien relativ selten/preiswert ist im Vergleich zur Synchronisation von Zustandsänderungen
  - falls mehrere Objekte gleiche Werte oder Funktionalitäten repräsentieren und es nicht darauf ankommt, welcher Repräsentant verwendet wird
  - einfache Hilfsklassen

Zustandslose Methoden

- Zustandslose Methoden von veränderlichen Objekten können wie Methoden unveränderlicher Objekte genutzt werden, z.B.

```
class NumericalOps { static int plus(int a, int b) { return(a+b) ; }}
public class Sorter {      // nur Auszug
  private CollatingRule rule_ ; // unveränderlich
  public Sorter() {
    rule_ = new DefaultCollatingRule() ;
  }
  public String[] sort(String[] array) {
     String[] copy = new String[array.length];
     // array in copy übertragen und copy sortieren
     return copy ;
    }
}
```

- Sorter erzeugt lokale Kopie und gibt Referenz darauf zurück, dies ist nur solange sicher, wie jede Kopie nur genau einmal genutzt wird.

Voll synchronisierte Objekte

- Sicherheit in Java wird je Objekt, je Klasse durch Lock Mechanismus unterstützt
- Jedes Objekt muß sich selbst schützen, mittels synchronized an seinen Methoden, dadurch werden Methoden atomar und Abarbeitung sequentiell
- Ausnahme: Konstruktoren sind nicht synchronized - problematisch nur in Sonderfällen, z.B. innerhalb des Konstruktors wird Thread mit this erzeugt
- Sonderfall: Klassen mit veränderlichen static Variablen erfordern zusätzlich Nutzung des Klassen-Lock-Mechanismus, d.h. block synchronization über das Klassen Objekt: synchronized (getClass()) { ... Krit. Abschnitt ... }
- Abgeschwächte Fassung: partielle Synchronisation
  - falls Objekt veränderliche und unveränderliche Attribute enthält, müssen Methoden nicht als Ganzes synchronisiert sein
  - falls möglich: innerhalb einer Methode zuerst Zugriffe auf veränderliche Variable durchführen und mittels synchronized (this) schützen und anschließend Zugriffe auf unveränderliche Variable durchführen
  - Grundidee: kritische Abschnitte bilden und nur lokal schützen, evtl auch durch synchronisierte, interne Hilfsmethoden

Voll synchronisierte Objekte

- Besonderheiten
  - wird von einer synchronisierten Methode aus eine nicht-synchronisierte Methode desselben Objektes aufgerufen, so wird der Lock nicht freigegeben
  - sind nicht alle Methoden einer Klasse synchronisiert, so können nicht-synchronisierte Methoden parallel zu einer einzigen synchronisierten Methode abgearbeitet werden
- Idee der partiellen Synchronisation ist es, wechselseitigen Ausschluß nur auf kritische Abschnitte zu begrenzen
  - Beispiel: lineare Liste, deren Daten modifiziert, deren Struktur jedoch nur über Konstruktoren der Elemente modifiziert wird
  - Achtung: Aufteilung in 3 Abschnitte innerhalb 1 Methode geht i.a. schief
    - synchronized: Variable des Objektes lesen
    - nicht synchronisiert: Berechnungen mit Daten durchführen
    - synchronized: Variable des Objektes aktualisieren
  - Achtung: bei Vererbung können wesentliche Annahmen für partielle Synchronisation aufgehoben werden,

Gekapselte Objekte

- Idee: exklusiven Zugriff strukturell durch Vermeidung von geteilten Variablen sicherstellen
- „piggybacked“ Synchronisation
- eingekapselte Objekte verlassen sich auf Synchronisation des umschließenden Objektes
- Sichtweise
  - umschließende Objekte sind „Eigentümer“ der gekapselten Objekte
  - gekapselte Objekte sind „physikalischer“ Bestandteil des umschließenden Objektes
- Einfachste Variante: fixed containment
  - umschließendes Objekt erzeugt gekapseltes Objekt innerhalb des Konstruktors, verwaltet Referenz auf interner Variable, verändert Referenz nicht und gibt Referenz keinesfalls an andere (externe) Objekte bekannt

Gekapselte Objekte

- Erweiterte Variante: managed ownership
  - Grundidee: exklusive Ressource wird jeweils nur von 1 Objekt genutzt, aber kann von Objekt zu Objekt weitergereicht werden
  - alternative Bezeichnungen: tokes, batons, linear objects, capabilities
  - Politik:
    - falls Objekt im Besitz der Ressource, kann es damit agiert werden
    - falls Objekt Ressource besitzt, ist kein anderes Objekt im Besitz der Ressource
    - falls Objekt Ressource an anderes Objekt weitergibt, ist es nicht länger im Besitz der Ressource
    - falls Objekt Ressource zerstört, dann kann kein anderes Objekt Ressource jemals wieder nutzen
  - erfordert Festlegung von Operationen
  - erfordert Konventionen zur Nutzung und deren Durchsetzung

Gekapselte Objekte: managed ownership

- Interpretationen
  - Eigentümer = Host — Eigentümer = Adapter
  - Ressource = Helper — Ressource = Adaptee
- Operationen
  - aquire: Anlegen, Erzeugen, erstmalig Beschaffen
  - forget: Aufräumen und Vergessen
  - give,put: Referenz weiterreichen und beim Eigentümer zerstören
  - take: Referenz anfordern/wegnehmen
  - exchange: Austausch von 2 Ressourcen zwischen 2 Eigentümern
  - duplicate/clone: erzeugen und weiterreichen einer duplizierten Ressource auf Anforderung
- Konventionen zur Durchsetzung der Politik
  - Grundproblem: Zuweisung bei Referenzen p = q beläßt ebenfalls Wert von q.
  - Konvention: Reinitialisierung nach entsprechener Operation vereinbaren

Konventionen in managed ownership

- 3 Arten: Nutzungskonventionen für Eigentümer, Konventionen für Methoden der Ressource, Konventionen für Ressource Kontroll- oder Verwaltungsklassen
- Objekte haben 4 Wege um Kenntnis von einer Ressource zu erlangen
  - Ressource war Parameter des Konstruktors oder wurde zur Initalisierung verwendet
  - Ressource wurde durch Objekt erzeugt
  - anderes Objekt übermittelt Referenz als Nachricht oder Rückgabewert
  - Referenz konnte einer public Variable entnommen werden
- Ziel der Konventionen ist es, diese Zugangswege zu schließen
- weitere Aspekte
  - Verwaltung von Ressourcenpools
  - Finalization, Überlagerung von Object.finalize

Zusammenfassung

- Problem durch Interferenz erkannt
- durch wechselseitigen Ausschluß mittels synchronized behandelt
- Grundsätzliche Lösung: alles synchronisieren ?
  - Verhindert Parallelität
  - erzeugt vermutlich Deadlocks
- Patterns nach Lea (Kap. 2, Safety)
  - unveränderliche Objekte
    - Idee: Zustandsänderungen vermeiden
    - zustandslose Methoden
  - voll synchronisierte Objekte
    - Idee: dynamisch exklusiven Zugriff sicherstellen
    - abgeschwächt: partielle Synchronisation
  - gekapselte Objekte
    - Idee: strukturell exklusiven Zugriff sicherstellen
    - fixed containment, Ressourcen, managed ownership