Freie Universität Berlin

Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf

Freie Universität Berlin

Überblick

Überblick

Freie Universität Berlin

- Interaktion über Nachrichten
  - Datenflussarchitekturen
  - Ereignisbasierte Systeme

Freie Universität Berlin

Interaktion über Nachrichten

Interaktion über Nachrichten

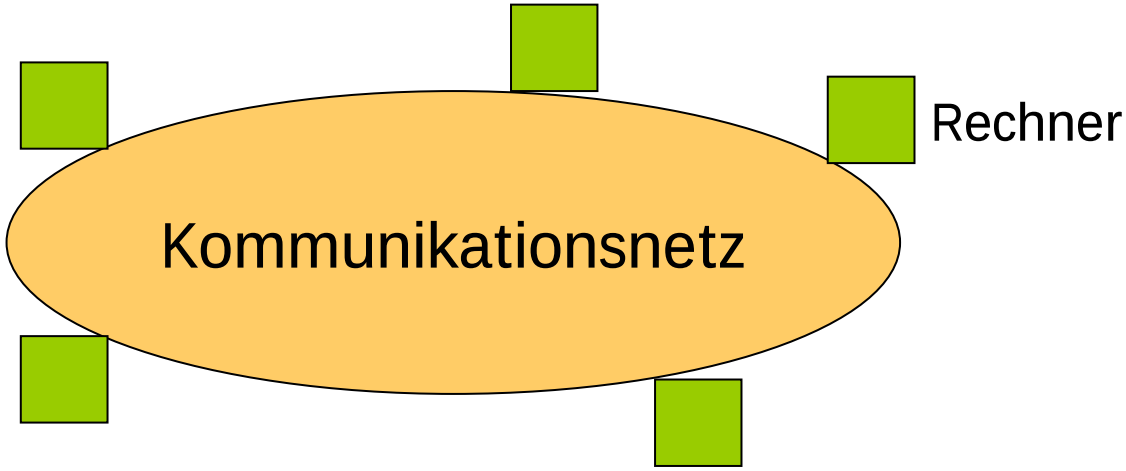
Freie Universität Berlin

- Bisherige Synchronisationsmittel nehmen gemeinsamen Speicher und gemeinsame Objekte an
- Aber: Es gibt auch Rechnerarchitekturen bei denen Prozesse *keinen gemeinsamen Speicher* haben, in dem sie gemeinsame Objekte unterbringen könnten
- Z.B.
  - Parallelrechner mit verteiltem Speicher
    (distributed-memory parallel computer)
    (auch „Mehrrechnersystem“ – *multi-computer*)
  - Rechnernetz *(computer network)*
    mit enger Kopplung *(cluster)* oder LAN oder WAN

Interaktion über Nachrichten

Freie Universität Berlin

- Prozesse in verschiedenen Rechnern müssen über Nachrichten *(messages)* interagieren, „kommunizieren“ (Nachrichtenaustausch, *message passing,* Interprozeßkommunikation, *inter-process communication, IPC*)
- Kommunikationsnetz *(communication network)* ist die HW/SW-Infrastruktur für die Nachrichtenübertragung



Nachrichteninteraktion in einem Rechner

Freie Universität Berlin

- **Aber:**
  Nachrichtenbasierte Interaktion kann auch *innerhalb eines Rechners* sinnvoll sein,
  z.B. zwischen Prozessen mit disjunkten Adressräumen

Verteiltes System

- **Def.:**
  Verteiltes System *(distributed system)* =
  System von Prozessoren, Prozessen, Threads, ...,
  die mangels gemeinsamen Speichers *nicht über Datenobjekte, sondern über Nachrichten* interagieren
  (Gegensatz: zentralisiertes System)
- **Beachte:**
  Die Klassifikation eines Systems als *verteilt* oder *zentralisiert* ist u.U. abhängig von der Abstraktionsebene, auf der das System betrachtet wird
  (Beispiel: System lokal kommunizierender Prozesse)

Kommunikationsoperationen

Freie Universität Berlin

- Prozesse kommunizieren unter Verwendung von Kommunikationsoperationen *(communication primitives)*
  - send
    Nachricht versenden (auch write, ...); die Nachrichtenquelle heißt Sender *(sender)* oder Produzent *(producer)*
  - recv
    Nachricht entgegennehmen (auch read, ...); die Nachrichtensenke heißt Empfänger *(receiver)*, Verbraucher *(consumer)*
  - in vielen möglichen Varianten (s.u.)

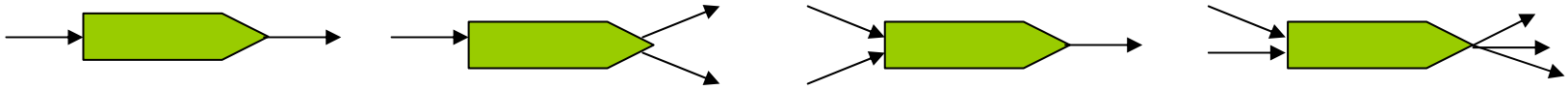
Freie Universität Berlin

- send/recv beziehen sich auf
  1. die Nachricht,
  2. den Empfänger/Sender – entweder *direkt* oder *indirekt* über einen Kanal *(channel)*, der sich wie ein Puffer (3←) verhält
     [auch bei „*direkt*“ gibt es einen verborgenen Kanal, der dem Empfänger zugeordnet ist und aus dem nur dieser liest *(mailbox)*]
- Asynchrone Nachrichtenübertragung
  wenn Pufferungskapazität > 0
- Synchrone Nachrichtenübertragung
  wenn Pufferungskapazität = 0

Arten der Zustellung

- Direkte Zustellung
  kann an *einen* oder mehrere Empfänger gehen:
  - Einfachsendung *(unicast)* oder
  - Rundsendung (*multicast* oder *broadcast*)

- Indirekte Zustellung über Kanal
  kann mit *einem* oder *mehreren* Sendern bzw. Empfängern verbunden sein:



Disjunktives Warten

Freie Universität Berlin

- Disjunktives Warten *(selective wait)* beim Empfangen von *mehreren Kanälen*:

```
SELECT {
  message = channelA.recv();
    ...
    break;
  OR   message = channelB.recv();
     ...
    break;
  OR   ...
}
```

- wartet, bis bei mindestens einem der Kanäle eine Nachricht anliegt, führt dann (nichtdeterministisch) eines der möglichen recv und die darauf folgenden Anweisungen aus.

Klassifizierung

Freie Universität Berlin

- Softwaretechnische Klassifizierung:
  1. Datenfluss-Architektur *(dataflow)* (➔6.1):
     Filter *(filter)* wandelt Eingabedaten in Ausgabedaten, *weiß nichts* von seiner Umgebung
  2. Dienst-Architektur *(client/server)*:
     Klienten *(clients)* beauftragen Dienstanbieter *(servers)*, *erwarten bestimmte* Funktionalität und Antwort
  3. Ereignisbasierte Systeme *(event-based systems)* (➔6.3):
     Abonnenten *(subscribers)* sind an bestimmten Ereignissen interessiert, über deren Eintreten sie von den auslösenden Ereignisquellen *(publishers)* benachrichtigt werden
  4. Verteilte Algorithmen *(distributed algorithms)*:
     gleichberechtigte Partner *(peers)* kooperieren, *setzen* vereinbartes Verhalten *voraus*

- Prozess arbeitet als Filter *(filter)*, d.h.
  - *empfängt* Daten, evtl. über verschiedene Kanäle,
  - *berechnet* daraus neue Daten,
  - *versendet* diese, evtl. über verschiedene Kanäle.

- Nicht notwendig, aber *typisch* ist:
  - Kommunikationspartner bzw. -kanäle sind *formale Parameter* des Prozess-Codes: Ports *(ports)*
  - Parameter ist *entweder* Eingabe- *oder* Ausgabekanal *(input port, output port)*

6.2. Datenfluss-Architekturen

Freie Universität Berlin

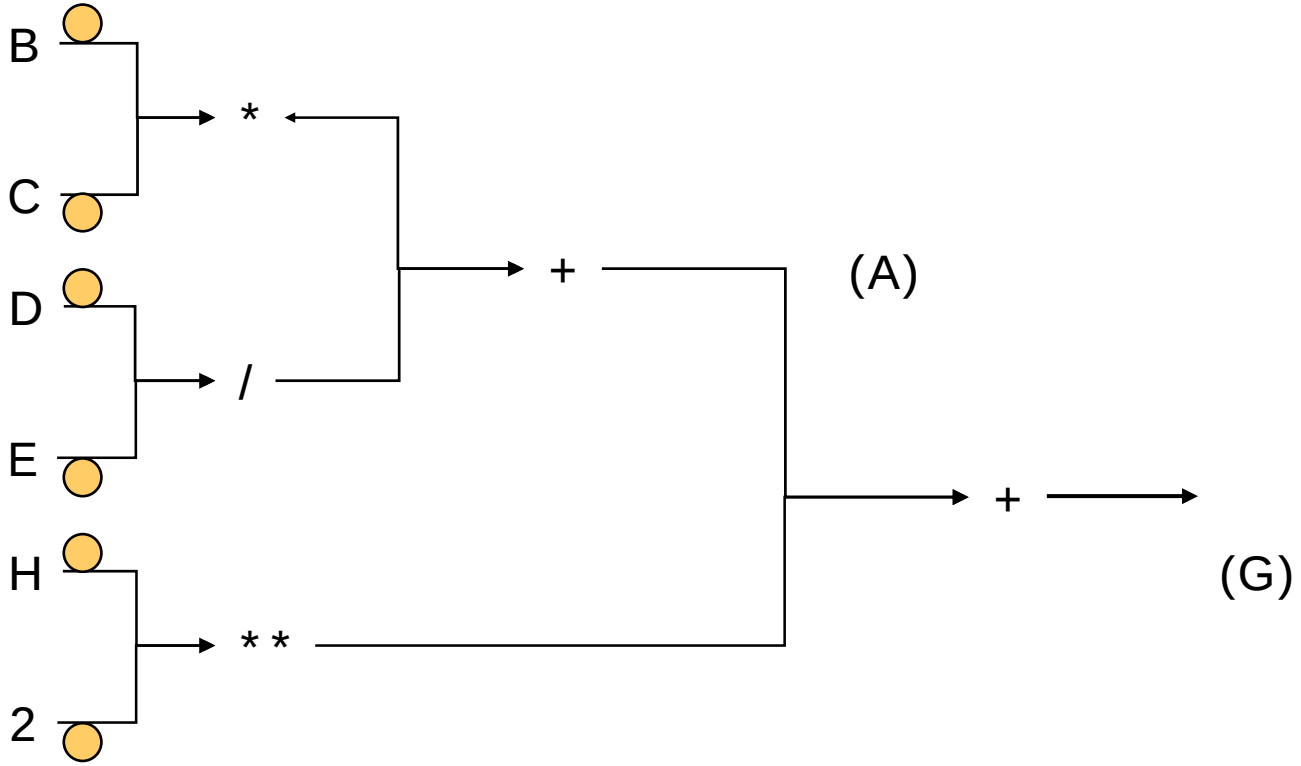
- Von-Neumann-Rechner:
  - Programm und Daten in einem Speicher
  - Programmzähler identifiziert nächste Operation die Speicher modifizert
  - Sequentiell, Fokus auf Ablauf der Berechnung
- Non-Von Architekturen:
  - Aufgabe des gemeinsamen Speichers Programm und Daten
  - Andere Ablaufmodelle
- Datenfluss-Architekturen:
  - Modell der Berechnung über ein Netz von Verarbeitungsknoten zu denen Daten fliessen
  - Vorliegen von Daten als Voraussetzung zur Berechnung

Berechnung nach Datenabhängigkeiten

- Berechnungsausschnitt:

  A := B*C + D/F;

  G := H**2 + A;

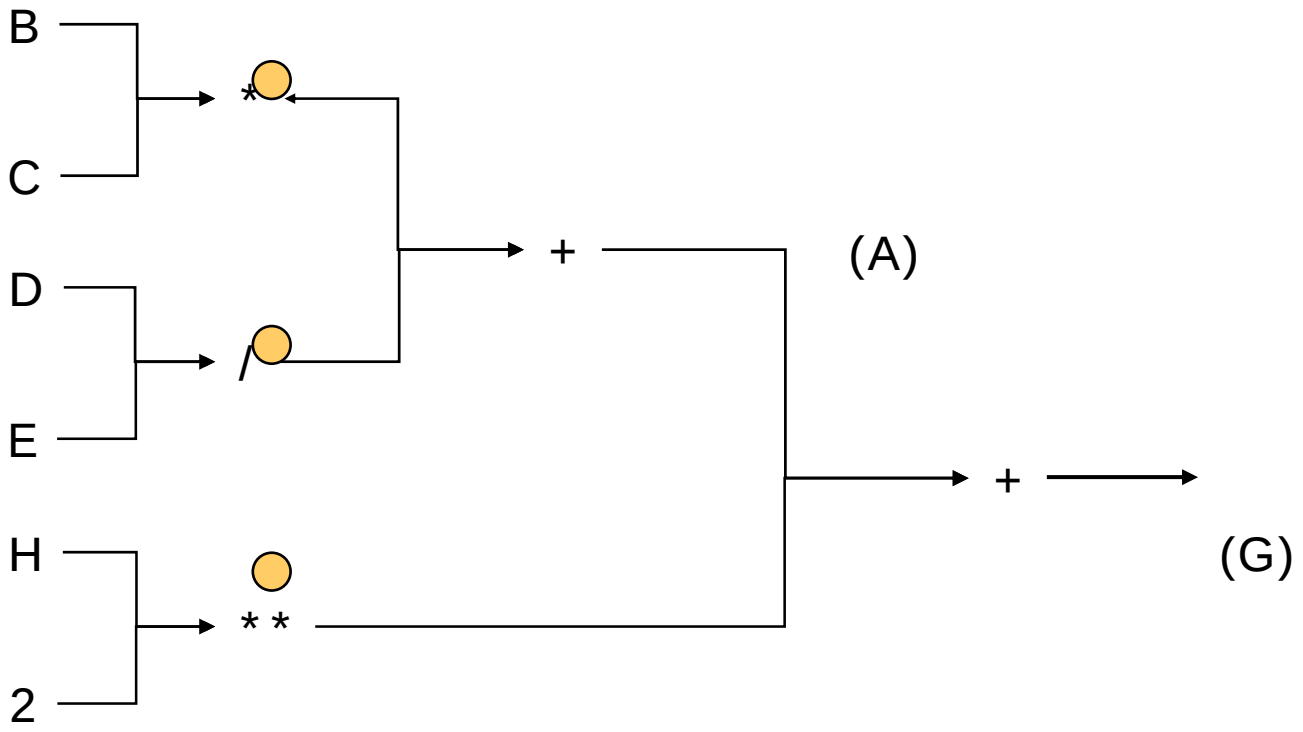


Berechnung nach Datenabhängigkeiten

- Berechnungsausschnitt:

  A := B*C + D/F;

  G := H**2 + A;



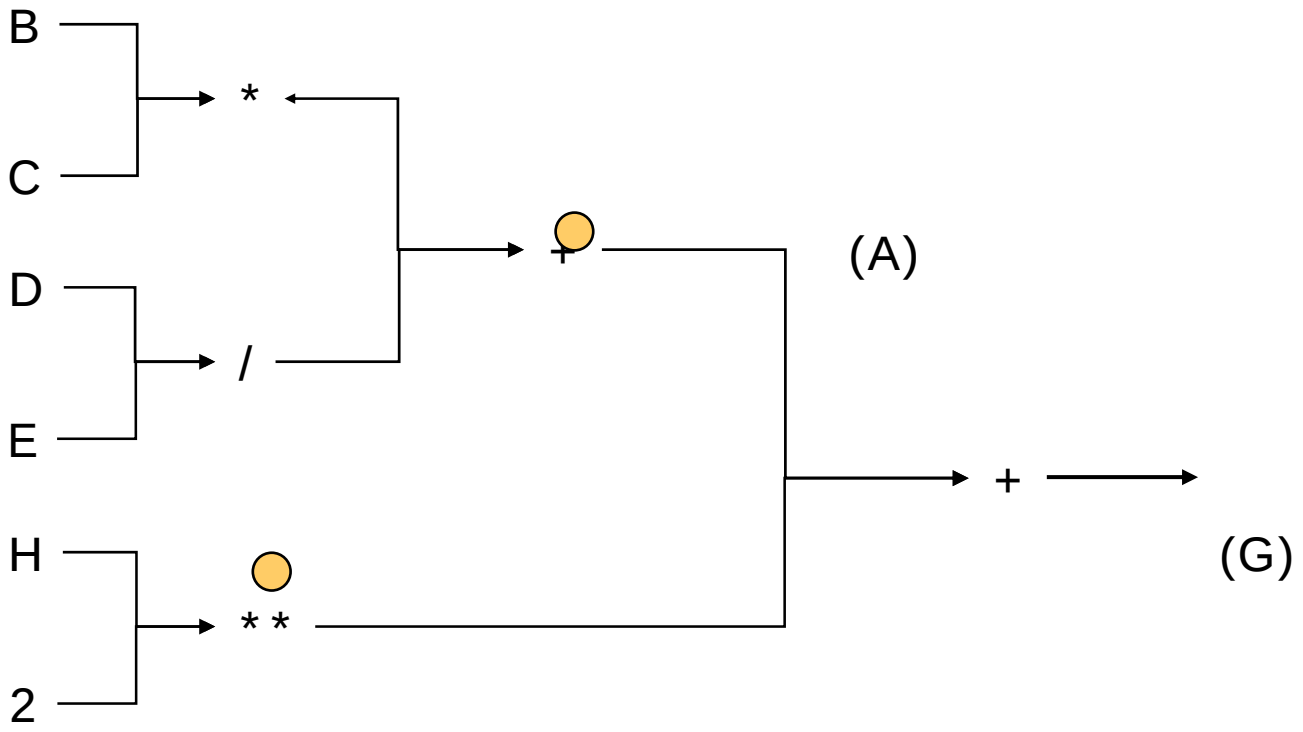
Berechnung nach Datenabhängigkeiten

Freie Universität Berlin

- Berechnungsausschnitt:

  A := B*C + D/F;

  G := H**2 + A;



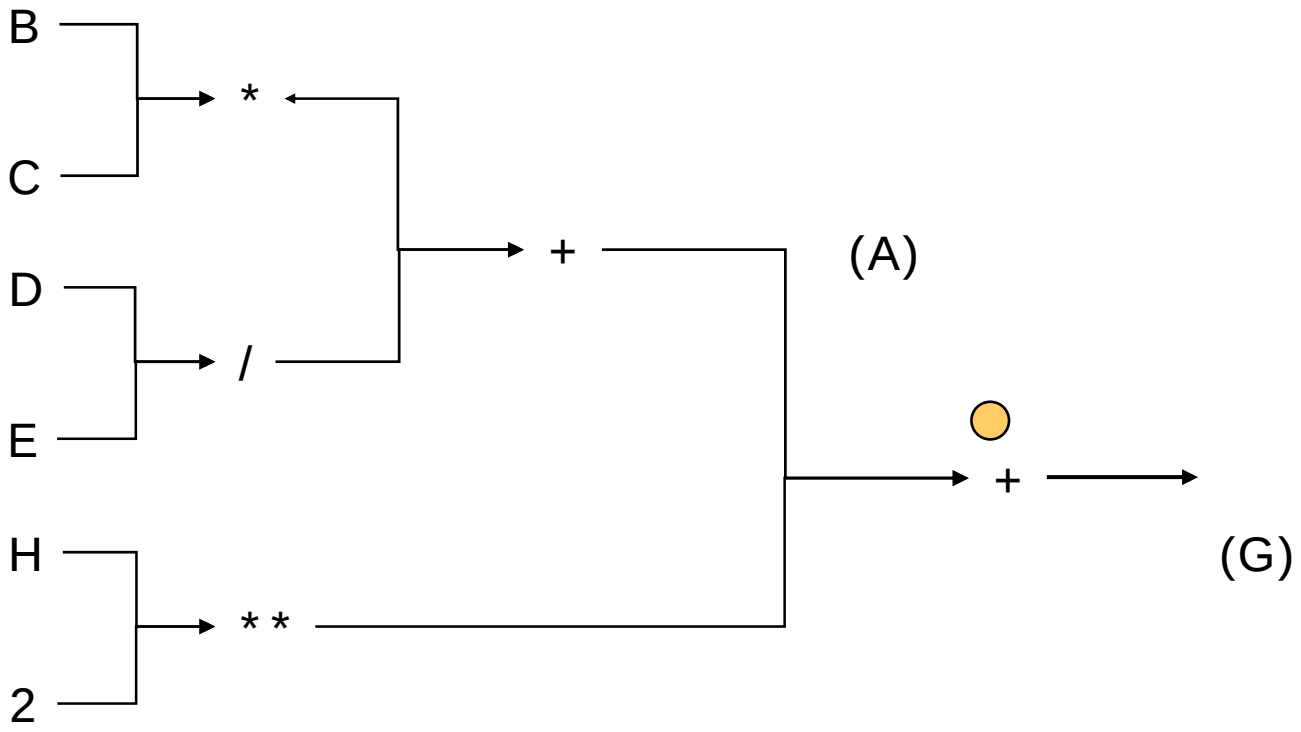
Berechnung nach Datenabhängigkeiten

Freie Universität Berlin

- Berechnungsausschnitt:

  A := B*C + D/F;

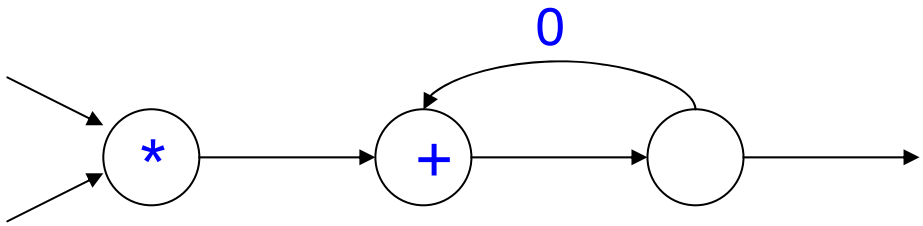
  G := H**2 + A;



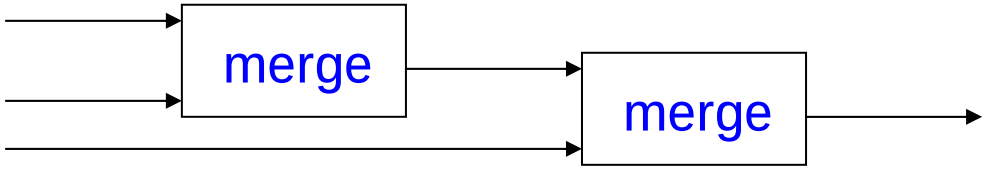
Datenflussgraph

- Datenflussgraph:
  Gerichteter Graph mit Filtern als Knoten und Kanälen als Kanten, d.h. genau 1 Sender/Empfänger je Kanal

- Beispiel 1: Partialsummen von Skalarprodukt:



- Beispiel 2: Verschmelzen sortierter Zahlenfolgen:



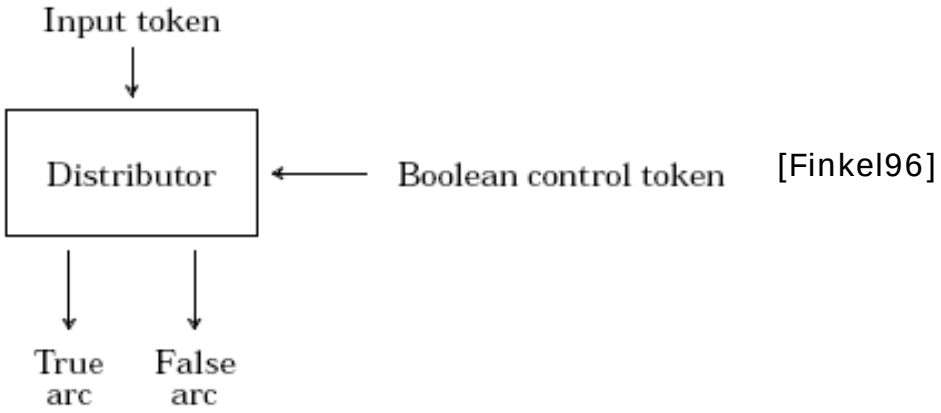
Datenflussgraph

- Datenflussgraph repräsentiert partielle Ordnung für die Ausführung
- Sequentielles Programm repräsentiert totale Ordnung

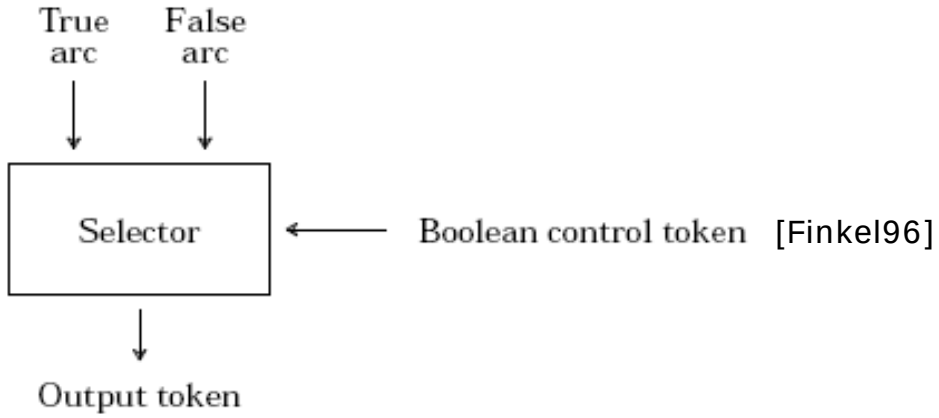
- Datenflussgraph ist inhärent nebenläufig
  - Lokale Bedingung: Ein Knoten kann „feuern“, wenn seine Eingabeparameter („Token“) vorliegen
  - Sehr feingranulare Nebenläufigkeit auf Operationsebene
- Funktionale Sicht: Kein Speicher!
  - Daher keine Seiteneffekte
  - Daher Potential zu Nebenläufigkeit:
    - Keine Konflikte möglich!

Datenflussgraph

- Programme haben Kontrollstrukturen
- Neben arithmetischen Knoten sind Kontrollknoten vorgesehen
  - Distributor



  - Selector



Programmieren mit Datenfluss-Graph

Freie Universität Berlin

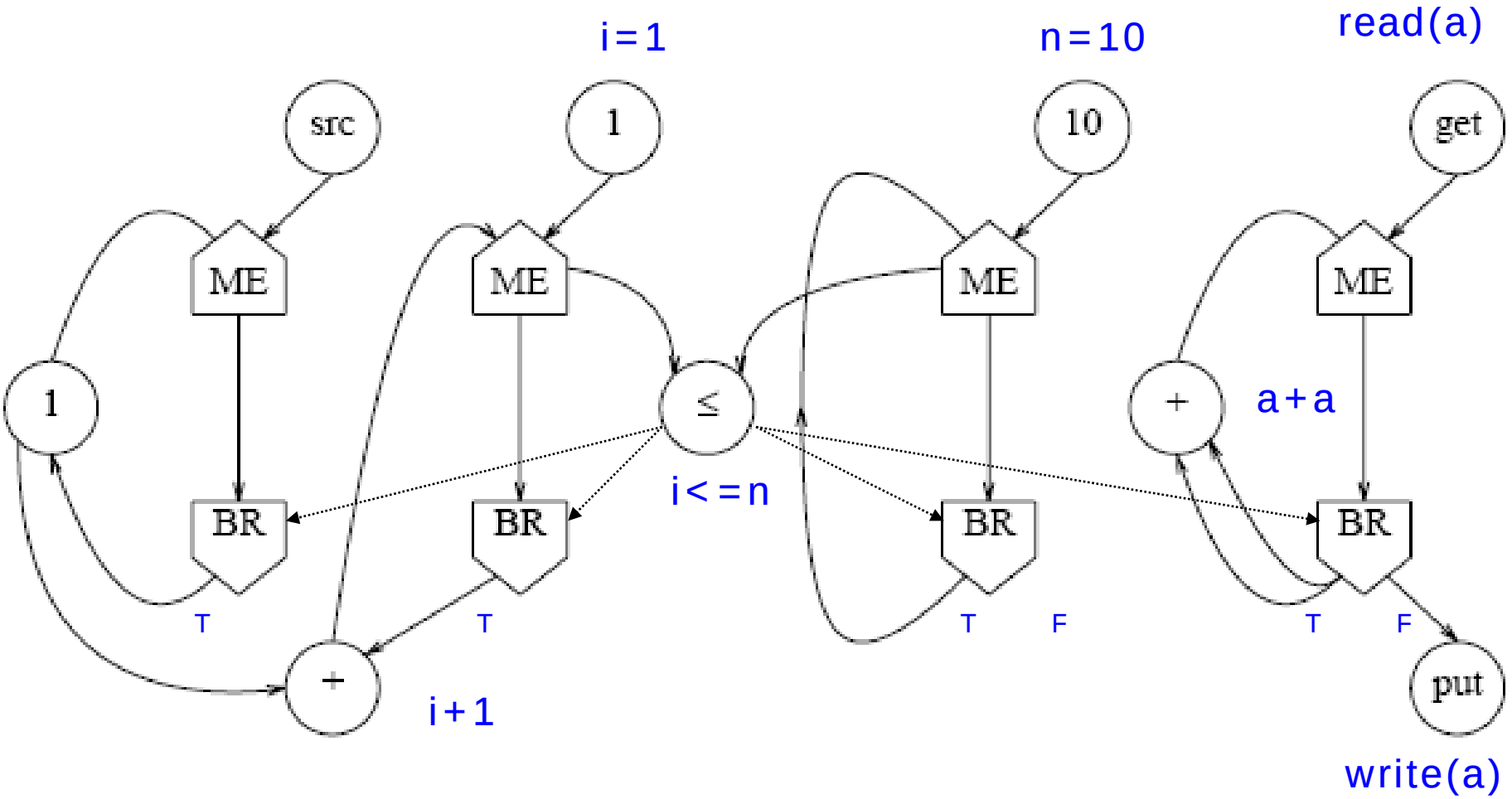
- *Einfache Zählschleife*

```
read (a);
n = 10;
for i:=1 to n
  a := a + a
end
write (a);
```

- Transformiert:

```
read (a);
n := 10;
i := 1;
while (i <= n)
  a := a + a;
  i := i + 1;
end
write (a);
```

Programmieren mit Datenfluss-Graph



6.2.1 Deklarative Formulierung

Freie Universität Berlin

- in hypothetischer Datenflusssprache (dataflow language)
  - Kanäle benennen
  - Prozesse deklarieren, auf Kanäle bezogen (mehrere Sender/Empfänger je Kanal möglich!)

- Es wird vorausgesetzt, daß Elementarbausteine wie *, +, merge, ... fertig vorliegen (als Sprachelemente oder in Bibliothek)

- Reale Sprache
  - Val
  - Sisal
  - ...

Beispiel in val: Mittelwert und Standardabweichung

Freie Universität Berlin

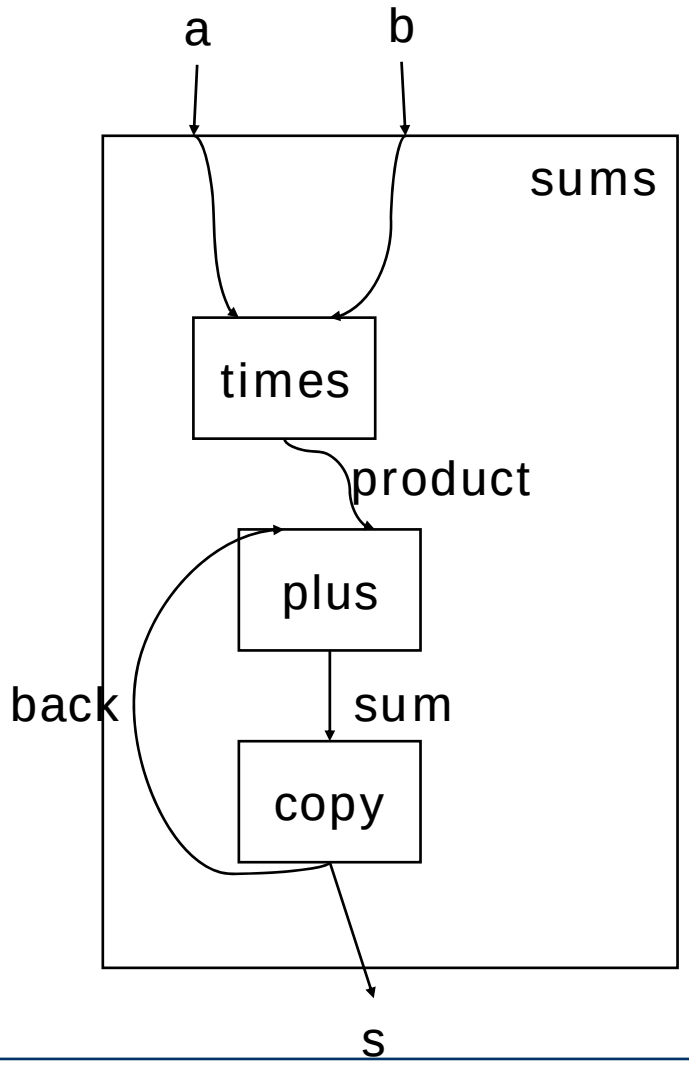
```
function stats(X, Y, Z: real) : real, real;
  let
    Mean : real := (X+Y+Z)/3;
    SD : real := sqrt((X - Mean)**2 +
                      (Y - Mean)**2 + (Z - Mean)**2)/3;
  in
    Mean, SD
  end
end
```

Beispiel Partialsummen

```
sums(in  chan float a,
     in chan float b,
     out chan float s)

   channels
        float product,
        float sum,
        float back(0)

   filters
        times(a,b,product),
        plus(product,back,sum),
        copy(sum,back,s)
end sums.
```



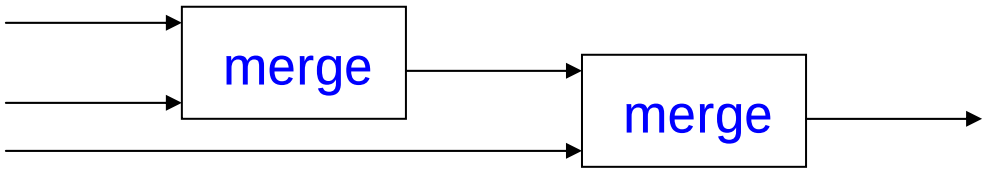
Beispiel Verschmelzen:

Freie Universität Berlin

```
merge3(in  chan int in1, in  chan int in2, in  chan int in3,
          out chan int res)

   channels
         int ch

   filters
         merge(in1,in2,ch),
         merge(ch,in3,res)
end merge3.
```



6.2.2 Imperative Formulierung

- Kanaltyp repräsentiert durch generische Schnittstelle

```
interface Channel<Message> {
  void send(Message m);
  Message recv();
}
```

- Prozesse erzeugen, die mit Kanälen parametrisiert sind (mehrere Sender/Empfänger je Kanal möglich!)

Beispiel

- **Beispiel** (s.o.): Verschmelzen dreier Zahlenfolgen

```
void merge3(Channel<Integer> in1, Channel<Integer> in2,
            Channel<Integer> in3, Channel<Integer> out) {
  Channel<Integer> ch = new ChannelImp();
  FORK merge(in1, in2, ch);
  FORK merge(ch,  in3, out);
}
```

- mit FORK wie in 2.2.2 und merge wie folgt:

Beispiel/2

```
void merge(Channel<Integer> in1, Channel<Integer> in2,
           Channel<Integer> out) {

   int one = (in1.recv()).intValue();
   int two = (in2.recv()).intValue();

   while(true) {
        if(one < two) {
             out.send(new Integer(one));
             one = (in1.recv()).intValue();
        } else {
             out.send(new Integer(two));
             two = (in2.recv()).intValue();
        }
    }
}
```

- arbeitet *nichtterminierend* auf unendlichen Eingabefolgen!
- (Übung: endliche Folgen mit null als Ende-Markierung)

Kanäle und Semaphore sind gleich mächtig

Freie Universität Berlin

1. Kanäle können mit Semaphoren implementiert werden (3←):

```
class Buffer<T> implements Channel<T> {...}
```

2. Semaphore können mit Kanälen implementiert werden:

```
class ChannelSema implements Semaphore {
  Channel<Object> c = new Ch<Object>();
  public void P() {c.recv();}
  public void V() {c.send(null);}
  public ChanSema(int init){
     for(int i=0; i<init; i++) V();
  }
}
```

Als Sprachkonzept

Freie Universität Berlin

- Ohne Objektorientierung, Kommunikation als Sprachkonzept:
- z.B.
  - **CSP** (Hoare 1978)
  - **Occam** (Inmos Ltd. 1983, für *Transputer*)
  - **Pascal-FC** (Burns/Davies 1993, für Ausbildung)

- Senden: ChannelExpr **!** MessageExpr
- Empfangen: ChannelExpr **?** MessageVar

- Nebenläufigkeitsanweisung:

```
PAR
  process1
  process2
  ...
```

Beispiel Produzent und Verbraucher

Freie Universität Berlin

- Einfaches Beispiel Produzent und Verbraucher:

```
CHAN ch:
PAR
  REAL64 x:
  SEQ
        produce x
        ch ! x
        continue
  REAL64 y:
  SEQ
        prepare
        ch ? y
        consume y
```

- Ein interaktives Programm mit einfacher, textbasierter Benutzer-Interaktion

  - liest von **Standard-Eingabe** *(standard input)* – hinter der sich die Tastatur verbirgt
    (in Java durch Benutzung von System.in),

  - schreibt nach **Standard-Ausgabe** *(standard output)* – hinter der sich der Bildschirm verbirgt
    (in Java durch Benutzung von System.out)

- Typische Systemaufrufe *(system calls)* für Ein/Ausgabe:
  - read (portNumber, bufferAddress, byteCount)
  - write(portNumber, bufferAddress, byteCount)

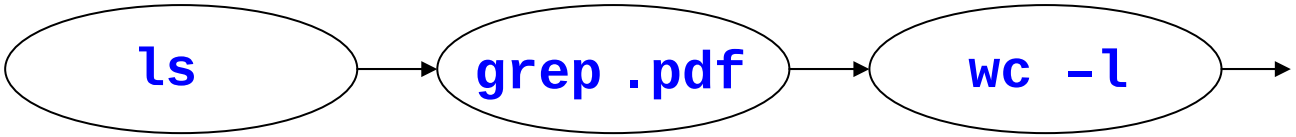
- In Unix:
  - Lesen von Standard-Eingabe: read(0,b,c)
  - Schreiben nach Standard-Ausgabe: write(1,b,c)

- 0, 1 sind *implizite formale Parameter des Programms*, die bei Programmstart *mit bestimmten Kanälen aktualisiert* werden !
- read entspricht recv, write entspricht send

Pipeline

- Pipeline, in der Unix-Befehlssprache (Shell) formuliert, ist einfaches, *lineares* Datenflusssystem in deklarativer Formulierung (2.1.2←)

- Beispiel (liefert Anzahl der PDF-Dateien im aktuellen Verzeichnis):
  ls | grep .pdf | wc –l



- Tiefere Bedeutung von | :
  - Trennzeichen zwischen 2 Prozessen,
  - Kanal, der den Ausgabe-Port 1 des Senders mit dem Eingabe-Port 0 des Empfängers verbindet
    (nur 1 Sender/Empfänger je Kanal möglich!)

Pipeline

- Als Kanal fungiert eine Pipe („Rohr“), die mit einem speziellen Systemaufruf eingerichtet werden kann.

- Befehlsinterpretierer richtet die benötigten Pipes ein und startet die Prozesse mit entsprechend aktualisierten Ports.
  (➔ VL *Betriebssysteme*)

- Datenflussarchitekturen werden daher häufig auch als *pipe-and-filter architectures* bezeichnet.

Pipes in Java

Freie Universität Berlin

- In java.io sind Pipes definiert
  - PipedReader
  - PipedWriter
  - PipedInputStream
  - PipedOutputStream
- Threads können so miteinander verbunden werden und sich über die entsprechenden Ströme koordinieren
- Die beiden Endstücke werden über eine connect-Methode (oder gleich im Konstruktoraufruf) verbunden

Beispiel

Freie Universität Berlin

```
import java.io.*;
import java.util.*;

public class Pipe {

    static Random rand = new Random();
    // Hilfsmethode – Schreiben einer Zeichenkette in eine Pipe
    public static void writePipe(String s, PipedWriter pw) {
      try {
        for (int i=0; i<s.length(); i++) {
          pw.write(s.charAt(i));
          System.out.println("Wrote "+s.charAt(i));
          Thread.sleep(rand.nextInt(500));
        }
      } catch (Exception e){}
    }
```

Beispiel

Freie Universität Berlin

```
public static void main(String[] arg) {
    final PipedWriter pw = new PipedWriter();
    final PipedReader pr = new PipedReader();
     try {
      pr.connect(pw);
      (new Thread(new Runnable() {          // erster Schreib-Prozess
            public void run() {writePipe("Hallo von mir",pw);}})).start();

      (new Thread(new Runnable() {          // Lese-Prozess
            public void run() {
                  while (true) {
                     try {
                           System.out.println("Read "+(char)pr.read());
                     } catch (Exception e){}
                  }
            }})).start();
      (new Thread(new Runnable() {          // zweiter Schreib-Prozess
            public void run() {writePipe("Auch von mir!",pw);}})).start();
     } catch (Exception e) {}
   }
}
```

6.2.4 Funktionale Formulierung

- Zur Erinnerung:

- Bei Auswertung eines Funktionsaufrufs
  Argumentübergabe

  - entweder als Wert
    (sofortige Auswertung, *eager evaluation, applicative order evaluation, data-driven*)
  - oder als Ausdruck
    (verzögerte Auswertung, *lazy evaluation, normal order evaluation, demand-driven*)

Funktionale Formulierung

- *Haskell* praktiziert *verzögerte Auswertung* und erlaubt damit
  - nichtstrikte Funktionen
  - Verschränkung von Aufbau/Verarbeitung von Listen
  - unendliche Listen

- Verzögert ausgewertete Listen heißen auch Ströme *(streams)*

- und die damit arbeitenden Funktionen stromverarbeitende Funktionen *(stream-processing functions)*

Funktionale Formulierung

Freie Universität Berlin

- Beispiel 1: Zahlenfolge durch Quadratwurzeln ersetzen
```
roots :: Floating a => [a] -> [a]
roots = map sqrt
```

- oder mit expliziter Rekursion:
```
roots(h:t) = sqrt h : (roots t)
```

- Anwendung z.B.
```
roots [3, 4, 5/0, 6]
```

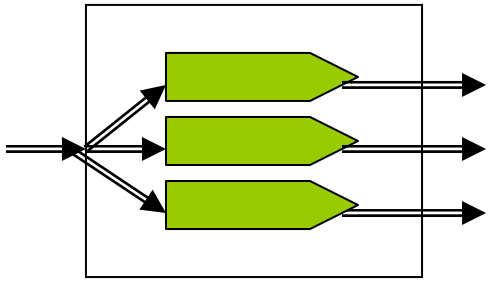
  liefert
```
[1.73205, 2.0,    ...
```
(Fehlermeldung)

- Ereignis *(event)*:
  - eine **Ereignis-Quelle** *(event source, publisher)* generiert Benachrichtigung *(event notification)*,
  - an der i. a. *mehrere* **Ereignis-Senken** *(event sinks, subscribers)* interessiert sind.
- Z.B.
  - „Telekom-Aktie fällt unter 5!“
  - „Druck steigt über 4,5!“
- Entkopplung:
  - Für die **Quelle** ist es *irrelevant*, *wer* auf ein Ereignis *wie* reagiert.
  - Die **Senken** sind an ganz *bestimmten* Ereignissen interessiert.

Freie Universität Berlin

- Verschiedenartige Ereignisse werden durch verschiedene Ereigniskanäle *(event channels)* repräsentiert, die auch gemäß den Ereignisnachrichten *getypt* sein können

Quelle



Senken

- *nachrichtenbasierter* Ereigniskanal (hier im Beispiel mit 3 Senken)
- Aufrufbasierte Variante:
  asynchroner Prozeduraufruf!

- Prozess
  - erklärt sich *dynamisch* zur Senke für einen Ereigniskanal – „abonniert das Ereignis“ *(subscription)*
  - und kann das Abonnement später wieder „kündigen“ *(cancellation).*

- Senke für Ereignis(se) kann gleichzeitig Quelle anderer Ereignis(se) sein.

- Ereignisbasiertes System *(event-based system)* = Prozessnetz aus Ereignisquellen und -senken

6.3.1 Ereignisorientierte Programmierung

Freie Universität Berlin

- Typischer *aufrufbasierter* Ereigniskanal:

```
interface Event<Notification> {
  void signal(Notification n);
    // forks all subscribers, passing n
  void subscribe(Handler<Notification> h);
    // subscribes event handler h
  void cancel(Handler<Notification> h);
    // cancels subscription
}

interface Handler<Notification> {
  void handle(Notification n);
    // handles n
}
```

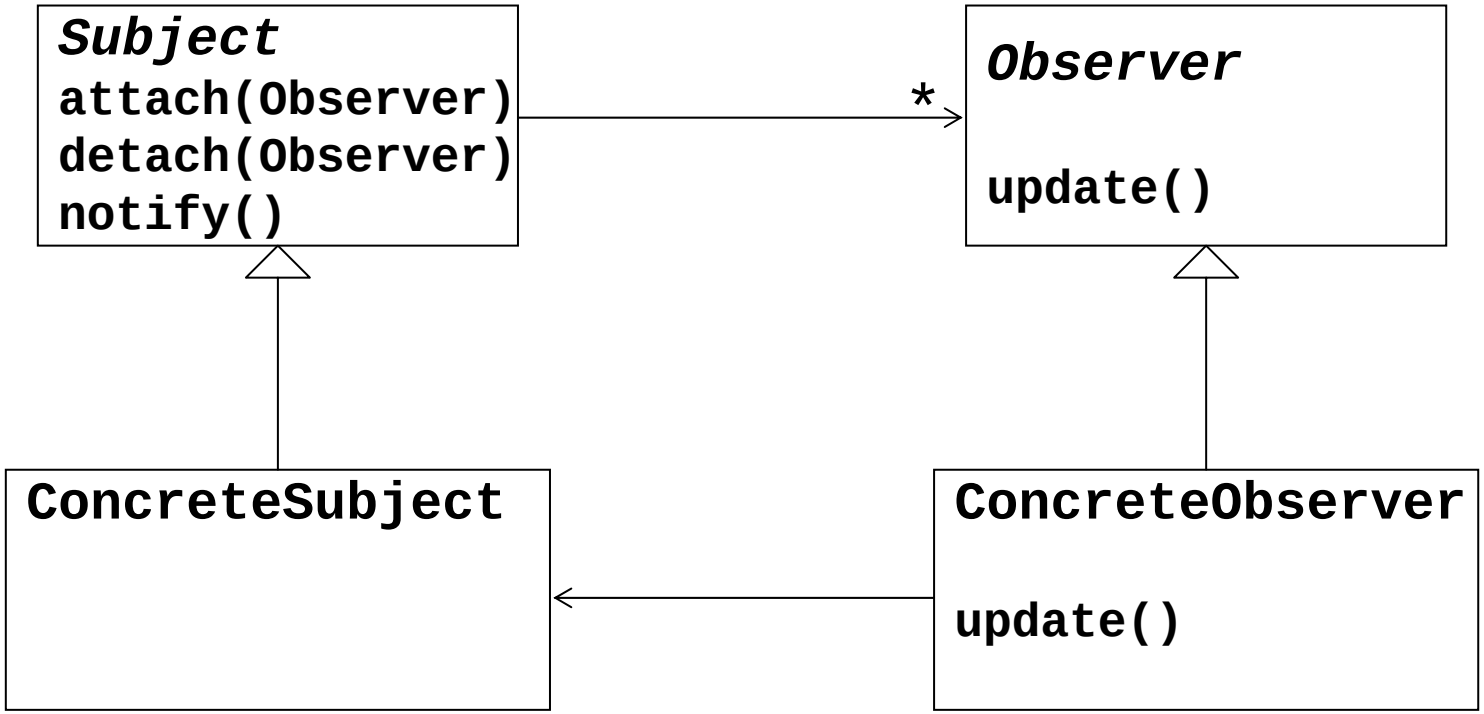
- Ereignisquelle für ein Ereignis lowPressure:

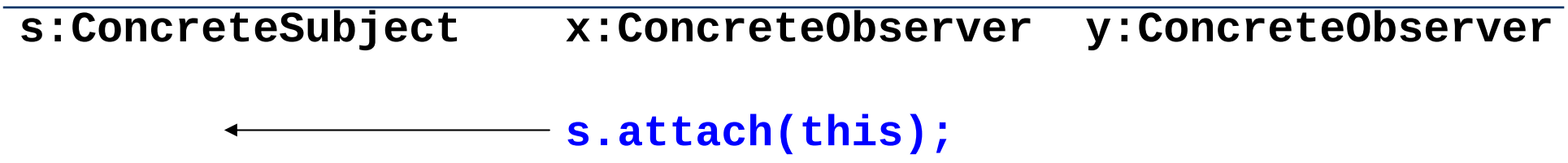
```
...
lowPressure.signal(pressure);
...
```

- Behandlung des Ereignisses:

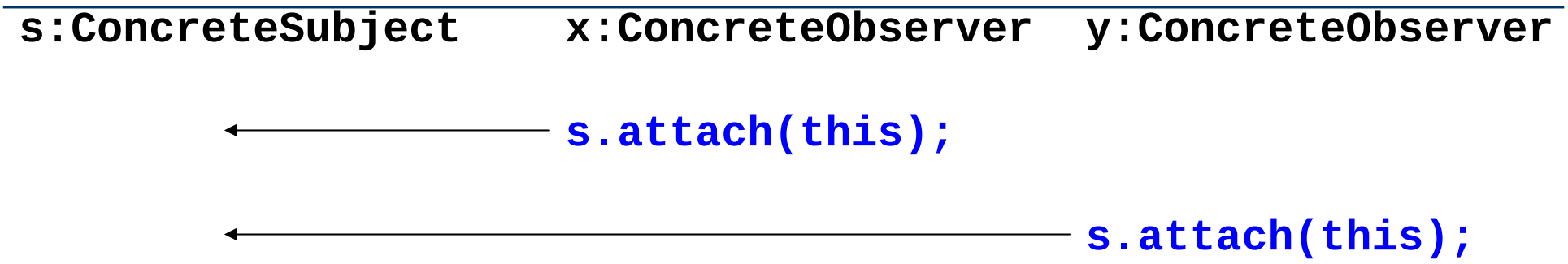
```
...
Handler h = new Handler<Float>(){
        void handle(Float press){ ..hier.. }};
...
lowPressure.subscribe(h);
...
...
lowPressure.cancel(h);
...
```

- *(observer pattern)* ist ein objektorientiertes Entwurfsmuster *(design pattern)*, das als *sequentielle* Variante der ereignisbasierten Interaktion betrachtet werden kann:

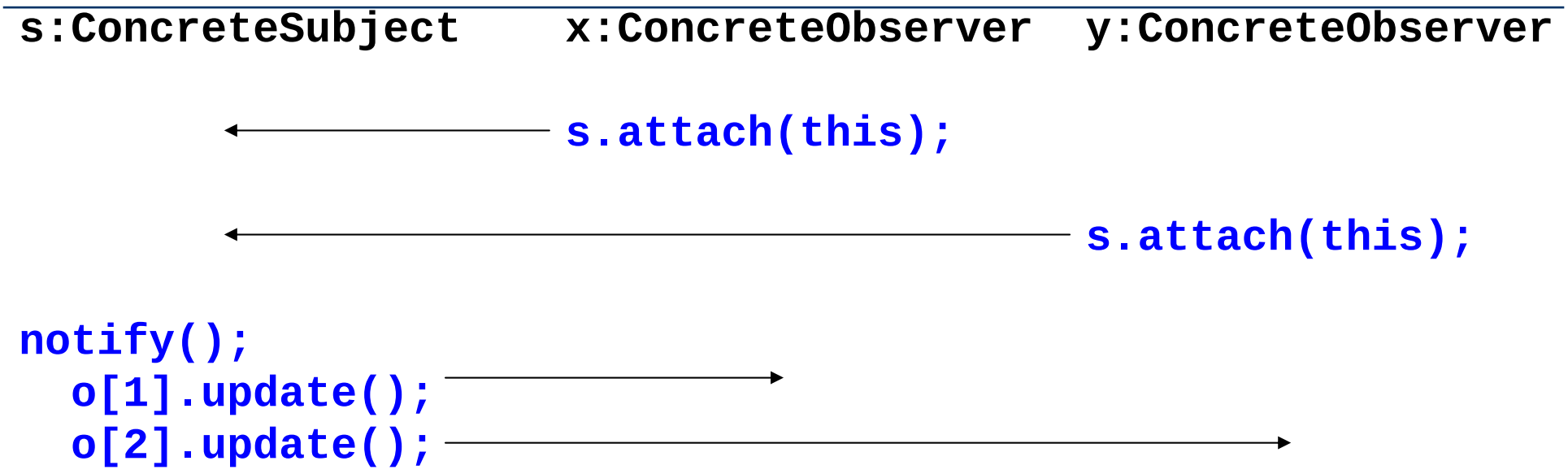



s:ConcreteSubject
x:ConcreteObserver
y:ConcreteObserver
s.attach(this);

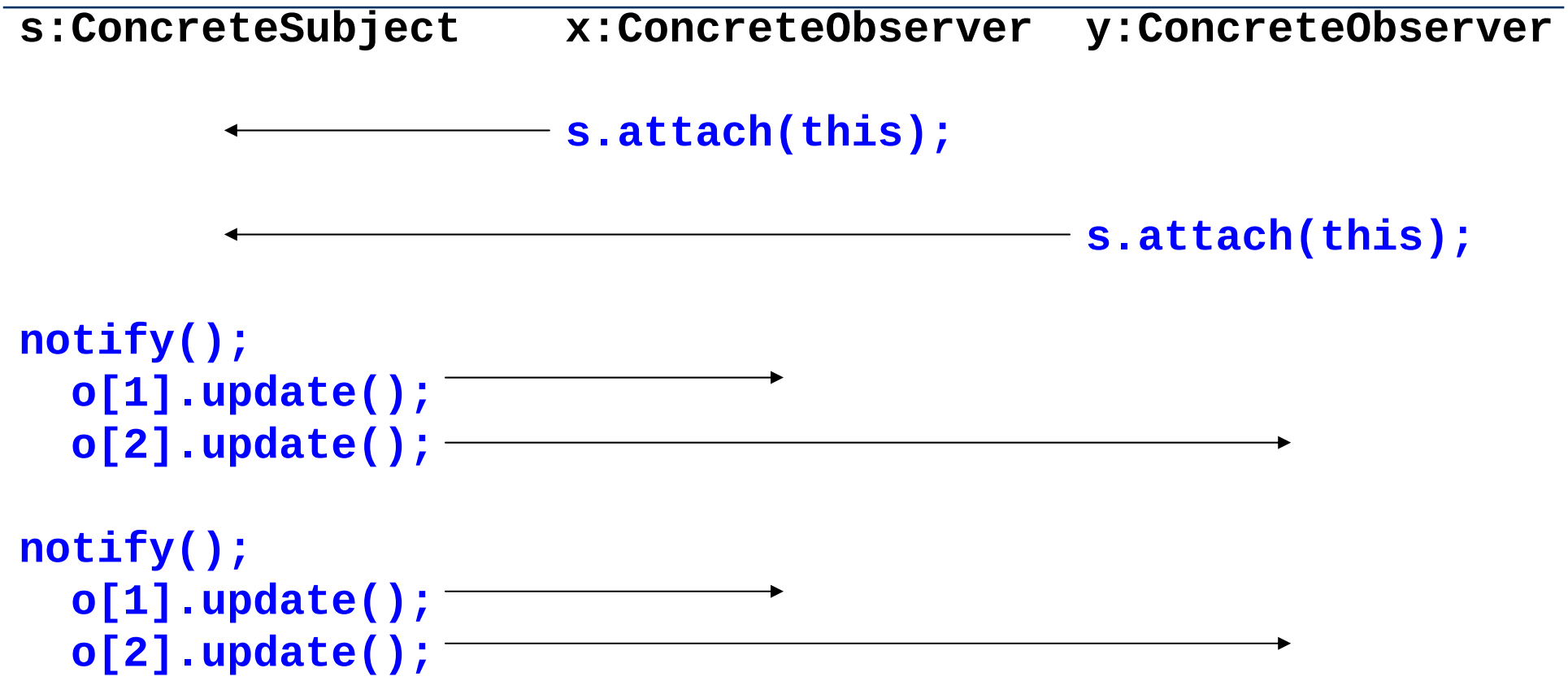
Das Beobachter-Muster



Das Beobachter-Muster



Das Beobachter-Muster



Analogie: *Ereigniskanal:* `event.subscribe(handler)`

Beobachter-Muster: `subject.attach(observer)`

- Unterstützung durch java.util:

```
interface Observer {
  void update(Observable o, Object notification);
}

class Observable {
  public synchronized void addObserver(Observer o);
  public synchronized void deleteObserver(Observer o);
  public void notifyObservers(Object notification);
  ...
}
```

- (für GUI-Ereignisse: Paket java.awt.event u.a.) orientiert sich am Beobachter-Muster:
  - *event listener* = Beobachter (= Ereignissenke)
  - *event source* = Beobachteter (= Ereignisquelle)
  - *event type* + *event source* (= Ereigniskanal/typ) (*event type* in AWT per Namenskonvention)
  - *event object* = Benachrichtigung

Das Ereignis-Modell von Java

Freie Universität Berlin

- Abonnieren eines Ereignisses: *event type* X + *event source*

- Ereigniskanal: `event.subscribe(handler);`
- Beobachter-Muster: `subject.attach(observer);`
- Java AWT: `source.addXListener(anXListener);`
- z.B.

```
button.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e) {
      Button b = e.getSource();…
    }
  }
);
```

- *Ereignisquelle* ist hier ein Button-Objekt:
  das Ereignis wird durch Klicken auf die entsprechende Taste der GUI ausgelöst.

Zusammenfassung

Zusammenfassung

Freie Universität Berlin

- Interaktion über Nachrichten
  - Datenflussarchitekturen
  - Ereignisbasierte Systeme