Freie Universität Berlin
Klaus-Peter Löhr

3 Datenfluss

3.1 Spezifikation von Filtern

Spezifikation eines Filters `Filter`:

Abbbildung von Eingabefolgen auf Ausgabefolgen

❶ *Signatur:* getypte Ein/Ausgabe-Ports, z.B.

```
filter :: P->[X]->[Y] -> ([U], [V])
```

❷ *Semantik:*

```
filter p x y = .....
```

Termination von Filtern:

terminierend: Spezifikation fordert *endliche* E/A-Folgen
Beispiele: Unix `sort`, Haskell `sort, sum`

u.U. terminierend: E/A-Folgen *können unendlich* sein
Beispiele: Unix `grep`, Haskell `filter`

nichtterminierend: *Eingabe*folgen müssen unendlich sein
(d.h. Filter ist nicht auf Eingabe-Ende vorbereitet
- „EOF", `^D`, `[]` u.ä.)

3.2 Implementierung von Filtern

Die Programmiersprache ist beliebig, aber (2-2.3, S. 17):
kanalbezogene Kommunikationsmechanismen erforderlich

Es folgen *Beispiele* für diesen Spezialfall:

(`filter` der Einfachheit halber ohne weitere Parameter)

❶ **Occam:** (inspiriert von CSP [Hoare])

```
PROC filter(CHAN OF INT in1, in2, out1, out2)
  SEQ
    in1 ? mesvar                      Empfangen
    .....
    out2 ! message                    Senden
    ALT                               disjunktives Empfangen
      in1 ? mesvar
        out1 ! message
      guard & in2 ? mesvar            verzögertes Empfangen
        out2 ! message
:
```

- *synchrone* Kommunikation (ungepuffert!)
- höchstens je *ein* Sender und Empfänger je Kanal

❷ **Unix** (hier mit *C*):

```
// usage:  filter in2 out2
int main(int argc, char** argv) {
read(0, &message, count); ...
int c = open(argv[2], flags);
write(c, message, count); ...
}
```

Empfangen
(aktualisiere `out2`)
Senden

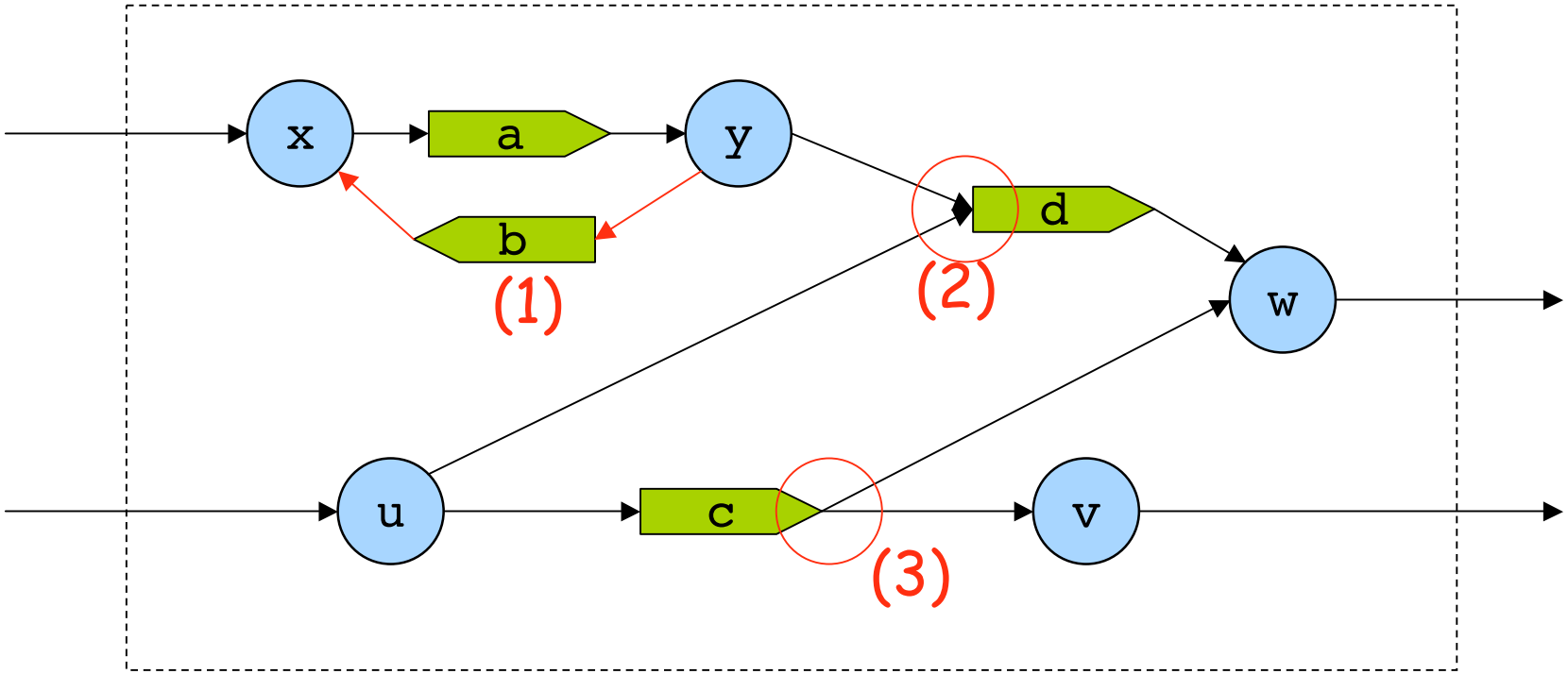
- beschränkt gepuffert
- Erstes Argument ist von `read/write` ist Portnummer
  = Nummer eines formalen Kanalparameters
  (hoffentlich richtig aktualisiert - *„file descriptor"*)
- disjunktives Empfangen/Senden mit `select`
- *keine Typisierung: Bytestrom!*

❸ Haskell:

```
filter :: [X]->[Y] -> ([U],[V])
filter x y = (u,v) where
                u = ...
                v = ...
```

- *synchrone* Kommunikation (ungepuffert),
  vermittelt durch *verzögerte Listen (lazy lists)* !
- „Senden" = Listenkopf zur Verfügung stellen
  „Empfangen" = Listenkopf übernehmen
- Argumente = Eingabefolgen
  anonymes Ergebnis = Tupel der Ausgabefolgen
- starke Typisierung

3.3 Komposition über Kanäle



„pipes-and-filters design pattern"

(1) *Rückkoppelung (Zyklus):* möglich, in der Praxis selten
(typisch: Datenflussmaschine [2.3, S. 14])

(2) *Verschmelzen:* *nichtdeterministisch*, oft akzeptabel
(typisch: gleichartige Daten aus mehreren Quellen)

(3) *Splitten:* *nichtdeterministisch*, selten akzeptabel
(typisch: Lastverteilung)

Beachte: Wenn alle Filter determiniert sind und jeder Kanal höchstens je einen Sender und Empfänger hat, ist das zusammengesetzte Programm determiniert. Es ist selbst wiederum als Filter einsetzbar.

Verklemmungsgefahr: steigt mit sinkender Pufferkapazität

Verklemmungsfreiheit liegt **genau dann** vor, wenn

- *unbeschränkte* Pufferung: Zyklenfreiheit
- *beschränkte* Pufferung: Kreisfreiheit*
- *keine* Pufferung: Kreisfreiheit*

* bei Ignorierung der Kantenrichtungen

Fairness: betrifft die Behandlung blockierender Kommunikationsoperationen beim *Verschmelzen* oder *Splitten*

- unbeschränkte Pufferung:
  nur Empfangen kann blockieren
- beschränkte Pufferung:
  auch Senden kann blockieren
- keine Pufferung:
  auch Senden kann blockieren

☛ In jedem Fall ist FCFS wünschenswert.

Beschreibung der Komposition:

❶ graphisch (wie auf S. 9)

❷ mit eigener Koordinations/Kompositionssprache

❸ mit Implementierungssprache der Komponenten

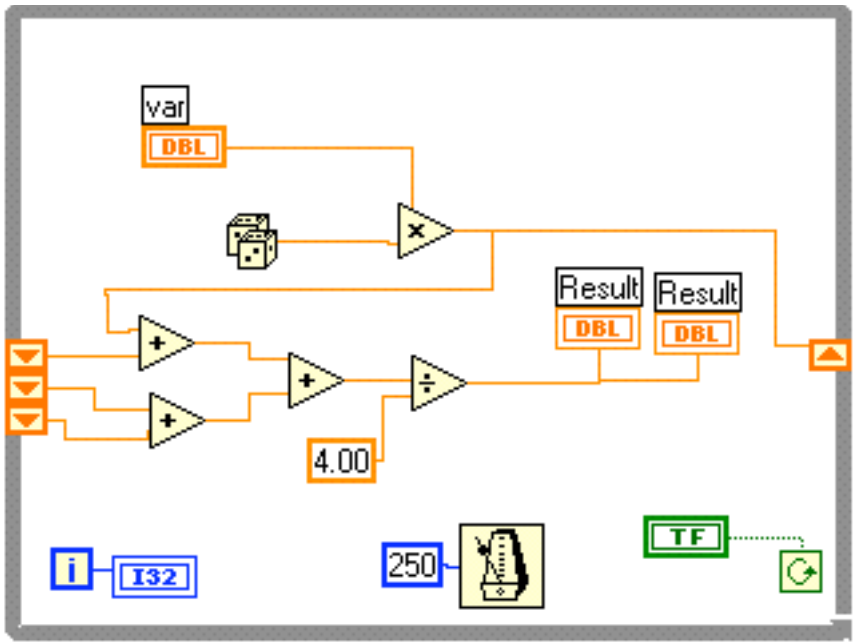
! In jedem Fall *formale Sprache* mit Syntax und Semantik !

3.3.1 ... mit graphischer Komposition

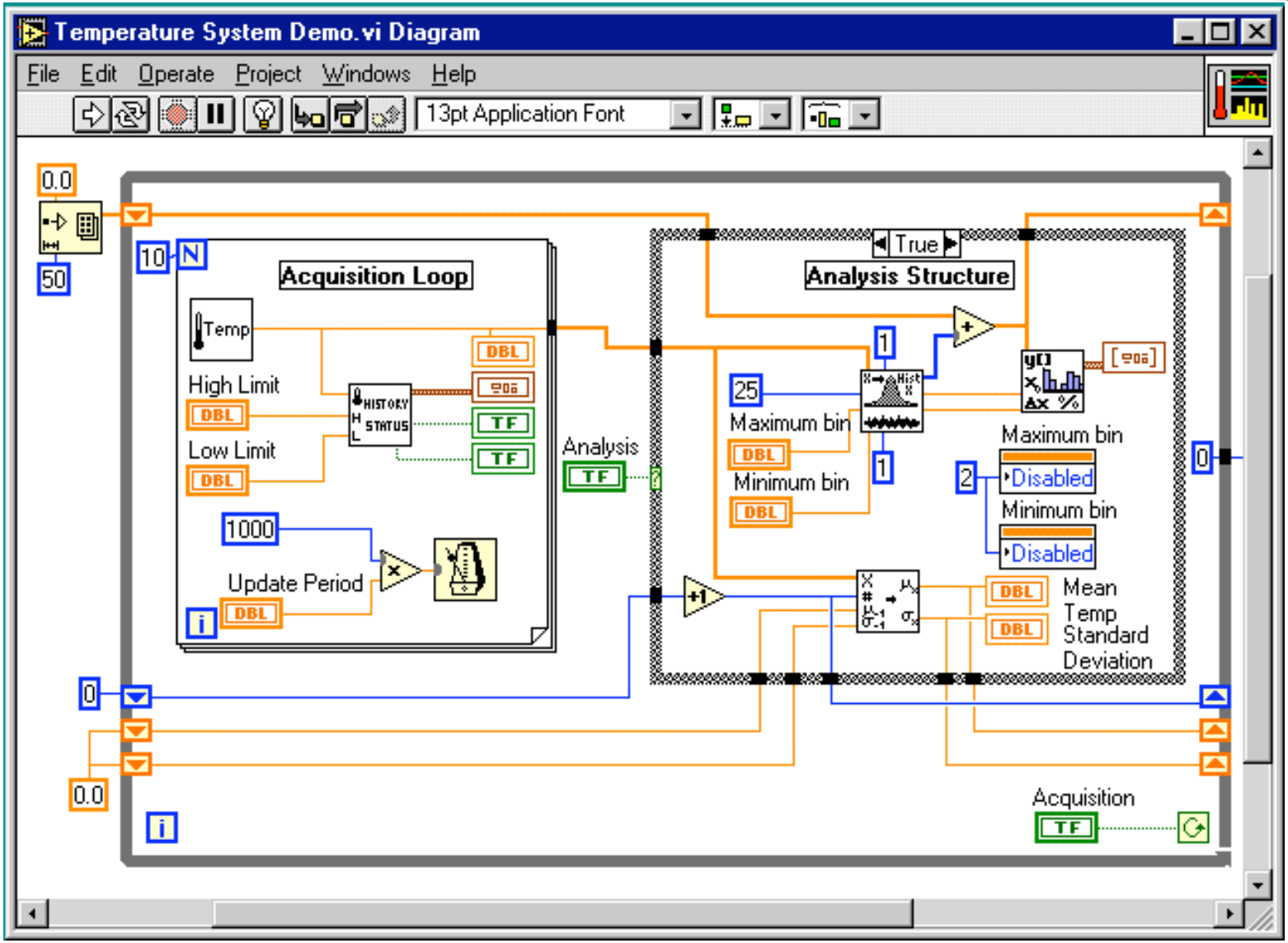
! Viele Ansätze in der Forschung - wenig praktische Systeme !

Beispiel **LabVIEW**:

Graphische Datenflußsprache für Laborautomatisierung -
aber mehr graphische *Programmierung* als *Komposition*



(Quelle: http://www.upscale.utoronto.ca/PVB/LabView.html)



3.3.2 ... mit Koordinationssprache

Populär sind **Skriptsprachen**, die von den **Betriebssystem**-Mechanismen zur Interprozesskommunikation Gebrauch machen.

Beispiel **Unix**:

- *Pipes* und *Named Pipes*
- beschränkt gepuffert
- nach FCFS bedient
- Senden blockiert, bis Empfänger vorhanden
- Senden verursacht Ausnahme *„Broken Pipe"*, wenn kein Empfänger mehr vorhanden

Und die Unix **Shell**:

Verteiltes Programm *deklarativ* mittels Operator **&** :

```
x & y & z & x & ...
```

Pipe-Operator **|** an Stelle von **&** :

vereinbart *anonymen Kanal (= Pipe)* zwischen den beiden Operanden, der die Ports 1 bzw. 0 im linken bzw. rechten Operanden aktualisiert

Named Pipes

ermöglichen beliebige Datenfluss-Graphen:

Named Pipes

ermöglichen beliebige Datenfluss-Graphen:

1. Weitere *Kanäle* imperativ als *Named Pipes* bereitstellen:
   z.B. `mkfifo a b c d ...`

2. Die *Filter* mit diesen Kanälen passend parametrisieren:
   z.B. `x <a b >d ... & ...`
   Achtung: ob `b` als Ein- oder Ausgabekanal oder als sonstiger „normaler" Parameter fungiert, ist der Spezifikation von `x` zu entnehmen.

3. Die *Named Pipes* wieder abräumen:
   `rm a b c ...`

Implizite Named Pipes für die Standard-E/A-Ports:

. . . <(*commands*) . . .

Der Standard-Eingabe-Port dieses Befehls wird über eine ad-hoc erzeugte, namentlich nicht bekannte *Named Pipe* mit dem Standard-Ausgabe-Port von *commands* verbunden.

. . . >(*commands*) . . .

Der Standard-Ausgabe-Port dieses Befehls wird über eine ad-hoc erzeugte, namentlich nicht bekannte *Named Pipe* mit dem Standard-Eingabe-Port von *commands* verbunden.

(Dies ist eine Fähigkeit der *bash Shell !)*

Benennung, Parametrisierung, Ausführung von Skripten:

- *Formale Parameter* `$1`, `$2`, `...` (ohne Vereinbarung),
  z.B. für 2.3, S. 15:

```
mkfifo x
merge $1 $2 x  &  merge $3 x $4
rm x
```

- *Benennung* durch Name der das Skript enthaltenden Datei

- *Als Programm deklarieren* durch Erteilung des x-Rechts

- *Ausführung* mit aktuellen Parametern wie normales Programm,
  z.B. `merge3 in1 in2 in3 out` ,
  endet mit Beendigung des textuell letzten Prozesses

Die beteiligten Filter können in beliebigen Sprachen implementiert sein (die die Unix-E/A unterstützen).

Ein Skript kann wiederum als Filter fungieren.

Das Beispiel von S. 8:

```
# usage:  page8 <in2 in1 out1 >out2
mkfifo a b c d
   x  a b    <$1     \
&  y <a b  >d        \
&  u      c d       \
&  v     <c         \
&  w      c d   >$2
rm a b c d
```

Übungsfrage: welche Annahmen über die E/A-Ports der beteiligten Programme wurden hier gemacht?

3.3.3 ... mit textueller Programmiersprache

... in der auch die Komponenten geschrieben sind, z.B.
Occam, Haskell, ..., Datenflußsprachen [W.M. Johnston et al.]

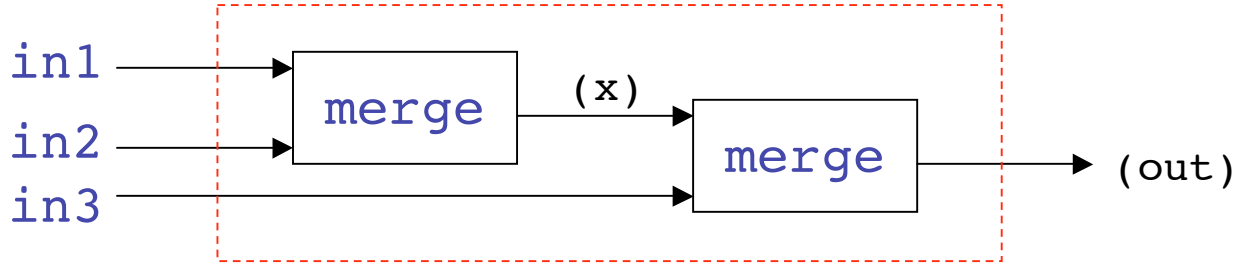
Occam (S. 5): Beispiel Verschmelzung wie in 2.3, S. 15:

```
PROC merge3(CHAN OF INT in1, in2, in3, out)
  CHAN OF INT x :
  PAR
    merge(in1,in2,x)
    merge(in3,x,out)
:
```

Haskell (S. 8): *„stream processing"*

Beispiel Verschmelzung wie in 2.3, S. 15:

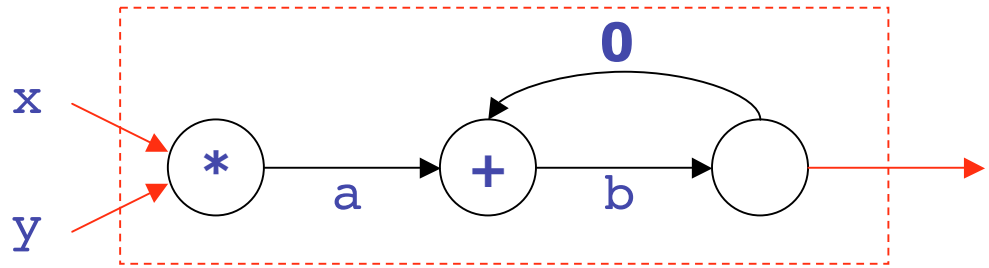
```
merge3 :: Num a => [a]->[a]->[a] -> [a]
merge3 in1 in2 in3 =
       merge (merge in1 in2) in3
```



Keine echte Nichtsequentialität in Haskell!
Aber: *Communicationg Haskell Processes* [CHP]

Beispiel Partialsummen von Skalarprodukt wie in 2.3, S. 14:

```
sumprod x y = b  where
        a = zipWith(*) x y
        b = zipWith(+) a (0:b)
```



Generell: *Zyklus* im Graphen entspricht *Rekursion* im Text

Haskell-Literatur ist reiche Quelle komplexer Beispiele

Haskell-Filter in *Shell*-Skripten

❶ `interact :: (String -> String) -> IO ()`

macht `interact f` zum Filter mit *Standard-E/A !*

Beispiel: Datei `lower.hs` enthält

```
main = interact (map toLower)
```

Datei `check.sh` enthält (vgl. Aufgabe 2.2a)

```
runhugs lower | java Check
```

Damit

```
$ check.sh <text
```

❷ E/A mit beliebigen Dateien → *Named Pipes* verwendbar

Zusammenfassung

- *Spezifikation* der Filter
- *Implementierung* der Filter mit Programmiersprache
- *Komposition* der Filter über Kanäle
  - mit graphischer Sprache
  - mit Koordinations/Kompositionssprache
  - mit Programmiersprache
- determiniert, wenn kein Splitten/Verschmelzen
- aber Verklemmungsgefahr
- komponiertes System ist wiederum Filter!

Quellen

C.A.R Hoare: Communicating Sequential Processes. Prentice-Hall 1985
http://www.usingcsp.com/

J.P. Morrison: *Flow-Based Programming.* Morrison Enterprises 2010 (2. ed.).
ISBN 0-442-1771-5

W.M. Johnston et al.: Advances in Dataflow Programming Languages.
ACM Computing Surveys 36.1, March 2004

(Fortsetzung:)

Quellen (Fortsetzung)

Occam:	www.wotug.org www.wotug.org/occam/documentation/oc21refman.pdf
Unix/Linux...:	pubs.opengroup.org/onlinepubs/9699919799/ www.FreeBSD.org/ www.linuxmanpages.com/ dell9.ma.utexas.edu/cgi-bin/man-cgi?pipe+7
Haskell:	www.haskell.org/haskellwiki/Haskell en.wikibooks.org/wiki/Haskell www.haskell.org/haskellwiki/Concurrency
CHP:	www.cs.kent.ac.uk/projects/ofa/chp/
LabVIEW:	www.ni.com/gettingstarted/labviewbasics/d/