Willkommen bei Verteilte Systeme!

Von Datenbanken
über Webdienste
bis zu p2p und Sensornetzen.

Heute: **Algorithmen und Zustand**.

Wer nichts garantiert, kann alles verteilen. Aber . . . ?

Wiederholung Vorlesung 2 (Zeit)

Reale Uhren:

- wall time vs. monotonic clocks
- Skew und Drift
- Synchronisieren: extern (Cristian, NTP), intern (Berkeley)

Logische Uhren:

- Lamport: Ein Zähler pro Knoten. *„Wenn es vorher war, dann ist der Zeitstempel kleiner.“*
- Vektor: N Zähler in jedem der N Knoten. Kausalität. *„Wenn der Zeitstempel kleiner ist, dann war es vorher.“*
  - sonst **vielleicht gleichzeitig**.

Ausschluss: Koordinator oder verteilt $\Rightarrow$ Zusätzliche Nachrichten.

Literatur

Distributed Systems - An Algorithmic Approach
– Sukumar Ghosh (2015).

Ablauf heute

- Warum?
- Wie? Representation und Fairness
- Richtigkeit 1: Sicherheit und Lebendigkeit

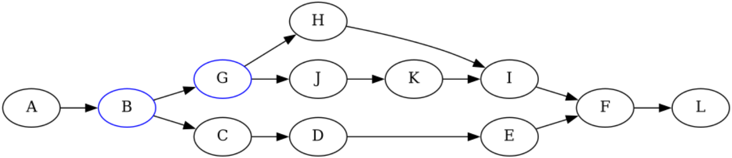
```
--- PAUSE ---
```

- Beispiel: Prozess-Farben
- Richtigkeit: Beweismethoden
- Zustand

Ziele heute I

- Sie verstehen, wieso in verteilten Algorithmen nicht einfach alle Möglichkeiten geprüft werden können.
- Sie verstehen, warum Richtigkeit aus Sicherheit und Lebendigkeit besteht.
- Sie kennen Definition über nichtdeterministische guarded commands.
- Sie verstehen Beweise über Invarianten und Rückführung auf bekannte Strukturen.
- Sie können erklären, wie ein Schnappschuss des Gesamtzustandes erstellt wird.
- Sie können erlären, wie Dijkstra-Scholten den Abschluss feststellt.

Verteilte Ausführung: Abfolgen von Zuständen

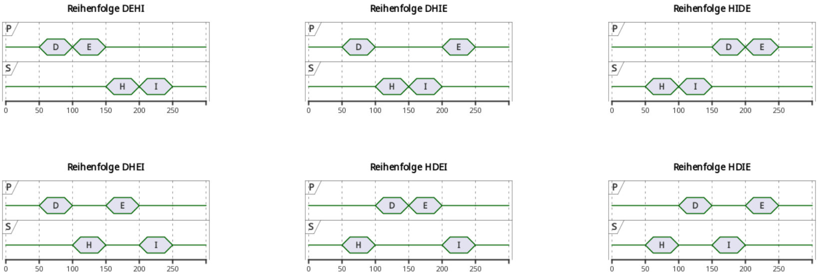


AB*CDE*FL oder AB*GHI*FL oder AB*GJKI*FL?

Alle möglichen Reihenfolgen prüfen?

$$N = \frac{(n \cdot m)!}{(m!)^n}; \text{n Prozesse, m Aktionen}^{1} \qquad (1)$$

Einfachster Fall:



[1] $n = 2, m = 2 \Rightarrow N = \frac{4!}{(2!)^2} = \frac{24}{4} = \mathbf{6}$; $n = 10, m = 4 \Rightarrow N > \mathbf{10^{34}}$

Kriterien statt Zustände

- ~~Alle Zustände prüfen~~
- Kriterien für alle Zustände beweisen

Repräsentation

Darstellung von verteilten Algorithmen.

Ziele:

- Sie verstehen choose-any und while-any.
- Sie können erklären, wie Fairness den Programmablauf ändern kann.

Notation für Programme

```
define : <program>
  choose-any
    <guard1>
      <statement1>
    <guard2>
      <statement2>
```

Kein Guard wahr: Abbruch (Fehler).

Notation für Programme

```
define : <program>
  choose-any
    <guard1>
      <statement1>
    <guard2>
      <statement2>
```

Kein Guard wahr: Abbruch (Fehler).

```
define : <program>
  define <variable> <value>
  while-any
    <guard1>
      <statement1>
    <guard2>
      <statement2>
```

Kein Guard wahr: Nichts (Ende).

Verkürzt

```
define : <program>
  while-any
    <guard1> : <statement1>
    <guard2> : <statement2>
```

Angelehnt an Dijkstras Guarded Command Language.[2]

choose-any = if, while-any = do

Ausprobieren:
https://hg.sr.ht/~arnebab/guarded-commands

[2] Ich nutze entgegen Dijkstras Vorstellungen ausführbaren Code, weil mir in Literatur zum Thema Fehler in dem entsprechenden Pseudocode aufgefallen sind. Dijkstras Notation produktiv: Promela language ⇒ SPIN

Strenge Notation

choose-any Äquivalent

```
if (...) {
  ...;
} else {
  throw new RuntimeException("undefined branch");
}
```

Standard if

```
define : if-else-ignore
  choose-any
    <guard1> : <statement1>
    <guard2> : <statement2>
    #t : skip
```

Beispiele

Nicht-Deterministisch

```
define : through-to-4
  define x 0
  while-any
    {x < 4}
      set! x {x + 1}
      display x
    {x = 3}
      set! x 0
      display x
  newline
```

1234 | *12301234* | *. . .* to copy

Beispiele

Nicht-Deterministisch

```
define : through-to-4
  define x 0
  while-any
    {x < 4}
      set! x {x + 1}
      display x
    {x = 3}
      set! x 0
      display x
  newline
```

1234 | 12301234 | ... to copy

Atomic

```
define : atomic-switch
  define a #t
  define flag #f
  while-any
    a
      set! flag #t
      set! flag #f
    : and flag a
      set! a #f
```

Endlosschleife to copy

Anwendung

```
define : euclidean a b
  while-any
    {a < b} : set! b {b - a}
    {b < a} : set! a {a - b}
  values a b
```

to copy

Größter gemeinsamer Teiler:

```
euclidean 999999 15678 .
;; => 117
```

Scheduler: Arten von Fairness

unbedingt fair Jeder Pfad wird irgendwann getestet[3]

stark fair Alle Pfade werden irgendwann getestet, deren Guard *unbegrenzt oft* wahr wird

schwach fair Alle Pfade werden irgendwann getestet, deren Guard wahr *bleibt*[4]

[3] Das ist der Normalfall, den wir ab jetzt ignorieren werden.

[4] Er wird nur auf zwei Arten wieder falsch, wenn er wahr war: sein Statement wird ausgeführt oder der Prozess terminiert.

Scheduler: Garantierte Fairness

- stark und schwach: geringere Garantien als bei sequenziellem Code
- ⇒ Mehr Freiheit für Netz-Implementierung
- ⇒ „günstigere“ Systeme

Fairness Beispiel

```
define : fair
  define b #t
  define x #f
  while-any
      b : set! x #t
      b : set! x #f
      x : set! b #f
      x : set! x : not x
to copy
```

Verhalten bei Fairness?

- stark
- schwach

Zusammenfassung

- Guarded actions
- Nicht deterministisch
- Fairness:
  - stark: Guard wird getestet wenn er beliebig oft wahr wird
  - schwach: Guard wird getestet, wenn er wahr bleibt

Richtigkeit

Garantien für verteilte Systeme.

In theoretischer Meteorologie werden die Grenzen und Ungenauigkeiten von Wettermodellen bewiesen, lange bevor sie implementiert werden.

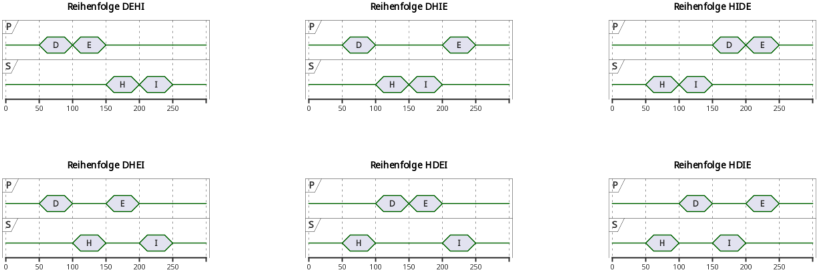
Um Versprechen von traditionellen p2p-Systemen für Systeme mit höheren Anforderungen an Verlässlichkeit zu realisieren, müssen wir beweisen, welche Garantien wir trotz reduzierter Koordination geben können.

Ziele für Richtigkeit

- Sie verstehen, warum in verteilten Systemen einfaches Testen schwerer ist
- Sie können die Kriterien Sicherheit (safety) und Lebendigkeit (liveness) beschreiben
- Sie erkennen den Einfluss von Fairness und Granularität.
- Sie verstehen Beweise über Invarianten.
- Sie verstehen Rückführung auf bekannte Strukturen.

Alle möglichen Reihenfolgen prüfen?

$$N = \frac{(n \cdot m)!}{(m!)^n} \text{; n Prozesse, m Aktionen}^5 \qquad (2)$$



[5] $n = 2, m = 2 \Rightarrow N = \frac{4!}{(2!)^2} = \frac{24}{4} = \mathbf{6}$; $n = 10, m = 4 \Rightarrow N > \mathbf{10^{34}}$

Kriterien

- ~~Alle Zustände prüfen~~
- Kriterien für alle Zustände beweisen

Kriterien:

- Sicherheit (Safety)
- Lebendigkeit (Liveness)

Sicherheit (Safety)

Es passiert nie etwas Schlechtes.

- Die Temperatur steigt nie über 100°C
- Sendet nie in einen vollen Kanal
- Liest nie, während geschrieben wird
- Kein Verklemmen (Deadlock): Prüft guards
- Teilweise Richtigkeit (Partial correctness): Wenn das Programm endet, ist die Antwort richtig

Lebendigkeit (Liveness)

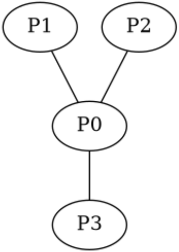
Irgendwann passiert etwas Gewünschtes.

- Fortschritt: Kein Verhungern / livelock $\rightarrow$ recursion step
- Fairness: Kommt eine Aktion irgendwann dran?
- Beendigung (termination): Das Programm wird enden

Richtigkeit = Teilweise Richtigkeit + Beendigung

(total correctness = partial correctness + termination)

Beispiel: Nachbarn mit unterschiedlichen Farben




```
define : colorme
  define P0 0
  define P1 0
  define P2 2
  define P3 2
  while-any
    : or {P0 = P1} {P0 = P2} {P0 = P3}
      set! P0 : modulo {P0 + 2} 4
    {P1 = P0}
      set! P1 : modulo {P1 + 2} 4
    {P2 = P0}
      set! P2 : modulo {P2 + 2} 4
    {P3 = P0}
      set! P3 : modulo {P3 + 2} 4
  values P0 P1 P2 P3
```

to copy

Beispiel korrekt?

Beispiel korrekt?

Teilweise Richtigkeit

Wenn alle Guards falsch sind, ist die Anforderung immer erfüllt. ✓

Guards:

- $P0 = P1$
- $P0 = P2$
- $P0 = P3$

Beispiel korrekt?

Teilweise Richtigkeit

Wenn alle Guards falsch sind, ist die Anforderung immer erfüllt. ✓

Guards:

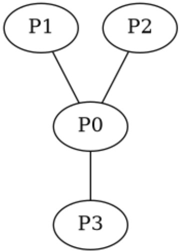
- P0 = P1
- P0 = P2
- P0 = P3

Beendigung

Bei Anfangszustand $P0 = P1 = 0$, $P2 = P3 = 2$ sind Endloschleifen **möglich**. ↯

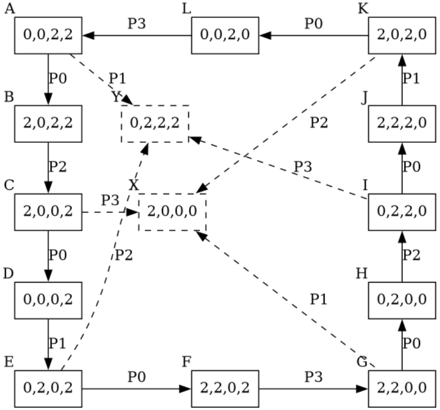
↯

Endlosschleife der Prozessfarben



Fairnessgarantien?

- stark
- schwach



Einfluss der Granularität

```
define : atomic-switch
  define a #t
  define flag #f
  while-any
    a
      set! flag #t
      set! flag #f
    : and flag a
      set! a #f
```

to copy

```
define : nonatomic-switch
  define a #t
  define flag #f
  while-any
    a : set! flag #t
    a : set! flag #f
    : and flag a
      set! a #f
```

to copy

Grenze: Reagierende System (open dynamic systems)

- Programme, die nicht enden sollen
- Reagieren auf die Umgebung

$\Rightarrow$ Nur Programm-Teile mit diesen Methoden beweisbar

Beweismethoden

- Asserting safety
- Liveness auf bekannte Fragen zurückführen
- Praxisverweise (Programmgestützte Beweise)

Asserting safety: Induktion mit Invarianten

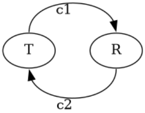
- Sicherheitsgarantie P
- Invariante I
- Initialzustand
- Prüfe alle möglichen Übergänge

Beispiel: Kommunizierende Prozesse

```
define c1 : channel 0
define c2 : channel 0

;; (empty? c1) : there is no message in the channel

;; programs for the processes
```



Endbedingung?

```
define : T
  define t 5
  while-any
    {t > 0} ;; Aktion 1
      send-message-to c1
      set! t {t - 1}
    : not (empty? c2) ;; Aktion 2
      receive-message-from c2
      set! t {t + 1}
```

```
define : R
  define r 5
  while-any
    {r > 0} ;; Aktion 3
      send-message-to c2
      set! r {r - 1}
    : not (empty? c1) ;; A. 4
      receive-message-from c1
      set! r {r + 1}
```

Beweis durch Induktion

- Sicherheit P: Gesamtzahl Nachrichten in Kanälen ist $N \leq 10$.
- Invariante $I \equiv (t \geq 0) \wedge (r \geq 0) \wedge (c1 + t + c2 + r = 10)$
- Basis: c1 $=$ 0, c2=0, t=5, r=5 $\rightarrow N \leq 10$
- Schritt: I bleibt bei jeder möglichen Aktion erhalten

Aktion 1: Unverändert: (t+c1), c2, r. Da Guard {t $>$ 0}: $t \geq 0$ ✓

Aktion 2: Unverändert: (t+c2), c1, r. Da t nur steigt: $t \geq 0$ ✓

Aktion 3: Unverändert: (r+c2), c1, t. Da Guard {r $>$ 0}: $r \geq 0$ ✓

Aktion 4: Unverändert: (r+c1), c2, t. Da r nur steigt: $r \geq 0$ ✓

Liveness mit well-founded sets

Auf Bekanntes zurückführen (das WF)

$\rightarrow$ Eindeutige Abbildung f: S $\rightarrow$ WF.

Dabei muss gelten:

- Es gibt keine unendliche Folge mit $w1 \gg w2 \gg$ ... im WF.
- Beim Übergang s1 $\rightarrow$ s2 mit $w1 = f(s1), w2 = f(s2)$ ist $w1 \gg w2$.

f: Maßfunktion (measure function)[6]

$\gg$: Totalordnung (z.B. $>$ bei Ganzzahlen).

[6]Maßfunktion: gibt ein Element des WF zurück, z.B. eine Zahl.

Beispiel: Auf positive Ganzzahlen zurückführen

Es gibt keine unendliche Folge von positiven Ganzzahlen mit $w1 > w2 > ....$

Übergang s1 $\rightarrow$ s2 mit $f(s1) = n, f(s2) = n - 1 \rightarrow$ Terminiert.

Beispiel: Phasen von Uhren synchronisieren

Uhren mit $(x + 1) mod 3$. Fester Takt, aber Fehler möglich.

```
define N 20
define phases : make-list N 0
define : sync i
  choose-any
    : member (i+1%3 i) (neighbors i)
      i+2%3 i
    : not : member (i+1%3 i) (neighbors i)
      i+1%3 i
```

Hilfsfunktionen auf Folie 74.

Synchron

Algorithm:

```
choose-any
 : member (i+1%3 i) (neighbors i)
   i+2%3 i
 : not : member (i+1%3 i) (neighbors i)
   i+1%3 i
```

0000000000000000000
1111111111111111111
2222222222222222222
0000000000000000000

Hilfsfunktionen:

```
define : show-all
    for-each display phases
    newline
define : sync-all x
    set! phases : map sync : iota N
    show-all

set! phases : make-list N 0
show-all
for-each sync-all : iota 3
```

Gestört 1

00020000000000000000
11111111111111111111
22222222222222222222
00000000000000000000

Algorithm:

```
choose-any
 : member (i+1%3 i) (neighbors i)
   i+2%3 i
 : not : member (i+1%3 i) (neighbors i)
   i+1%3 i
```

Hilfsfunktionen:

```
set! phases : make-list N 0
list-set! phases 3 2

show-all
for-each sync-all : iota 3
```

Gestört 2

```
0001000000000000000000
1122211111111111111111
2000002222222222222222
1111111000000000000000
2222222211111111111111
0000000002222222222222
1111111111000000000000
2222222222211111111111
0000000000002222222222
1111111111111000000000
2222222222222211111111
```

```
set! phases : make-list N 0
list-set! phases 3 1
show-all
for-each sync-all : iota 10
```

Zufällig

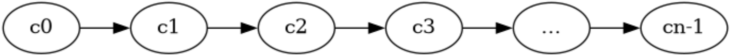
```
120002100111000002111
011111022222211111022
222222210000000222210
000000002111111100002
111111111022222221111
222222222221000000222
000000000002111111100
111111111111022222221
222222222222221000000
000000000000000211111
```

```
define : random-phase i
    inexact->exact
        floor {3 * (random:uniform)}

set! phases : make-list N 0
set! phases : map random-phase : iota N

show-all
for-each sync-all : iota 10
```

Beweisidee



Idee (ohne Beschränkung der Allgemeinheit):

- $1 \leftarrow 2$
- $2 \rightarrow 1$

Beobachtungen:

- $\rightarrow c_i$ - $\Rightarrow$ Pfeil verschiebt sich zu c_{i+1}. Kein $\rightarrow$ für c_0
- - $c_i \leftarrow \Rightarrow$ Pfeil verschiebt sich zu c_{i-1}. Kein $\leftarrow$ für c_{n-1}
- $\rightarrow c_i \leftarrow \Rightarrow$ Beide Pfeile verschwinden.

Beweis

Kostenfunktion: D = d[0] + d[1] + . . . + d[n-1]

mit

$$
\begin{aligned}
d[i] &= 0 \quad \text{; - c -} && (3)\\
&= i+1 \text{ ; - c } \leftarrow && (4)\\
&= n-i \text{ ; } \rightarrow \text{ c -} && (5)\\
&= 1 \quad \text{; } \rightarrow \text{ c } \leftarrow && (6)
\end{aligned}
$$

Jeder Veränderung der Pfeile reduziert die Kosten um 1.

Die Zahl positiven Ganzzahlen kleiner D_0 ist endlich. QED.

Weiterführend

In der Praxis: Beweisprogramme wie Coq:
https://de.wikipedia.org/wiki/Coq_(Software)

- POPL Distinguished: Higher-Order Leak and Deadlock Free Locks — bewiesenes Typsystem für Lockfreie Paralellität: https://www.youtube.com/playlist?list=PLyrlk8Xaylp4ecKH5damlZ2FrhbauW9Fm

Tiefer einsteigen: The Little Prover (Friedman und Eastlund, 2015).

Aktuell: Konsistenz ohne Koordination (Hellerstein und Alvaro, 2019) → blog.

Zusammenfassung

- Alle Reihenfolgen nicht testbar: 10 Prozesse, 4 Aktionen $\rightarrow$ 10^{34} Möglichkeiten $\Rightarrow$ Kriterien für alle Zustände beweisen.
- Kriterien:
  - Safety: Teilweise Richtigkeit $\rightarrow$ Invariante für alle Übergänge
  - Liveness: Beendigung (terminiert) $\rightarrow$ Rückführung auf Bekanntes
- Einfluss von Fairness und Granularität

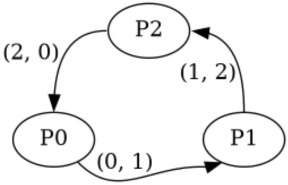
Globalen Zustand

Konsistenten Zustand zusammenfügen.

Ziele:

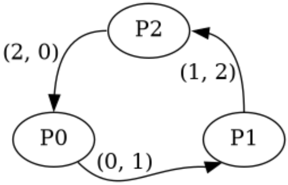
- Sie verstehen, welche Schwierigkeiten auftreten.
- Sie können einen konsistenten Schnitt von einem nicht konsistenten unterscheiden.
- Sie kennen Methoden, um verschiedene Arten von globalen Zuständen zu sammeln.

Beispiel: Token zählen



Es gibt 1 Token.
Wie viele Token gezählt?
(Möglichkeiten sammeln)

Beispiel: Token zählen



Es gibt 1 Token.
Wie viele Token gezählt?
(Möglichkeiten sammeln)

Möglichkeiten

1
- P0 (hat das Token) $\rightarrow n_0 = 1$.
- P1 (Token in (1,2)) $\rightarrow n_1 = 0$.
- P2 (Token in (2,0)) $\rightarrow n_2 = 0$.
- $\Rightarrow n_0 + n_1 + n_2 = 1$. ✓

2
- P0 (hat das Token) $\rightarrow n_0 = 1$.
- P1 (hat das Token) $\rightarrow n_1 = 1$.
- P2 (hat das Token) $\rightarrow n_2 = 1$.
- $\Rightarrow n_0 + n_1 + n_2 = 3$. $\lightning$

3
- P0 (Token in (0,1)) $\rightarrow n_0 = 0$.
- P1 (Token in (1,2)) $\rightarrow n_1 = 0$.
- P2 (Token in (2,0)) $\rightarrow n_2 = 0$.
- $\Rightarrow n_0 + n_1 + n_2 = 0$. $\lightning$

Globale Zustände

- Snapshot erstellen
- Informationen verbreiten (z.B. um Topologie zu erkunden)
- Abschluss feststellen

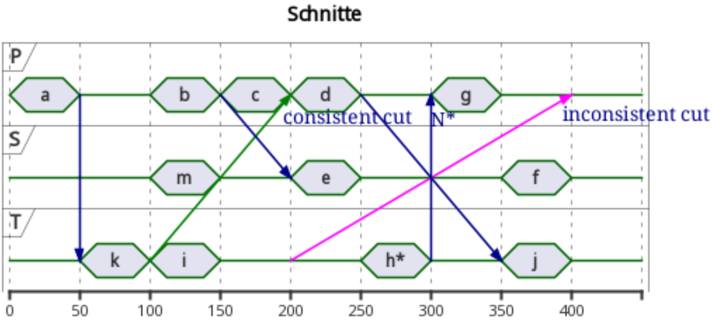
Snapshot

Ein in sich konsistenter Zustand.

- Ein konsistenter Snapshot ermöglicht z.B. einen roll-back.
- Alle Knoten anzuhalten ist üblicherweise zu teuer.
- Nachträglich empfangene Nachrichten müssen gespeichert werden.

Beispiel: Token zählen.

Bedingung für Snapshot: Konsistente Schnitte



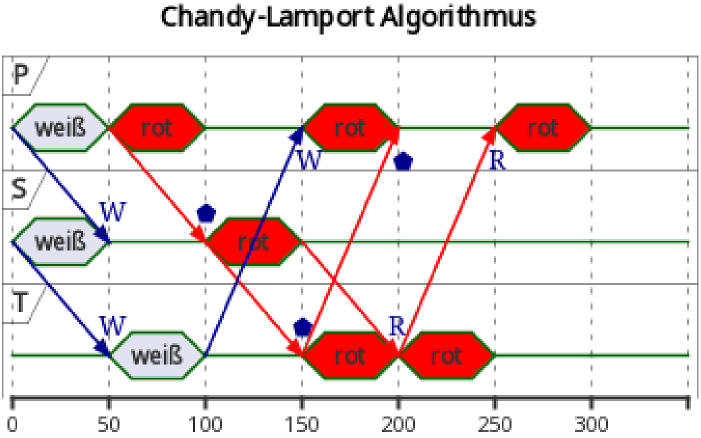
Konsistent: Enthält alle logischen Vorgänger.

Inkonsistent: Nach roll-back würde g N von h* erneut erhalten.*

Chandy-Lamport Algorithmus

- Initiator wird rot, speichert eigenen Zustand, sended Marker an alle Ausgänge.
- Erhalt des Markers: wird rot, speichert eigenen Zustand, sended Marker an alle Ausgänge.
- Alle roten speichern empfangene weiße Nachrichten.
- Ende: Alle sind rot, jeder hat über jeden Eingang einen Marker erhalten und über jeden Ausgang einen verschickt.
- Danach: Daten einsammeln.

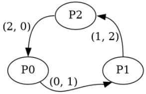
Chandy-Lamport, Beispiel



Chandy-Lamport, Beweisidee

- Kein weißer Prozess erhält je eine rote Nachricht.
- Braucht FIFO-Kanäle! (z.B. TCP)
- Rote Zustände, die zeitlich vor weißen liegen, können vertauscht werden.

Beispiel: Token richtig zählen



- P0 hat Token geschickt, wird rot, speichert: $n_0 = 0, sent_0 = 1, received_0 = 0$, sendet Marker.
- P1 leitet Token weiter, erhält Marker, wird rot, speichert: $n_1 = 0, sent_1 = 1, received_1 = 1$, sendet Marker.
- P2 leitet Token weiter, erhält Marker, wird rot, speichert: $n_2 = 0, sent_1 = 1, received_1 = 1$, sendet Marker.
- P0 erhält Token, erhält dann den Marker. Algorithmus abgeschlossen.

$$(n_0+n_1+n_2)+(sent_0-received_0)+(sent_1-received_1)+(sent_2-received_2) = 1 \tag{7}$$

Abschluss feststellen: Dijkstra-Scholten

- Initiator sendet Signal an alle Verbundenen.
- Empfänger sendet Signal an alle Folgenden, sendet Ack, wenn
  - Berechnung terminiert, und
  - alle Folgenden Ack geschickt haben
- Wenn Initiator so viele Acks wie Signale erhalten hat, ist die Berechnung terminiert.

Zusammenfassung

- Token zählen ist nicht-trivial
- Konsistente Schnitte müssen alle logisch früheren Daten enthalten
- Chandy-Lamport sendet Farbmarker
- Broadcast
- Abschluss feststellen: Dijkstra-Scholten

Auf dass Sie furchtlos Garantien geben können!

$\ddot\smile$

Notation für Daten

```
define-record-type <message>
    message a b c ;; Konstruktor
    . message? ;; Test
    a message-a ;; Getter
    b message-b
    c message-c
```

copy-paste Programme

Kopierbare Versionen der Programmschnipsel.

Through-to-4

```
define : through-to-4
___ define x 0
___ while-any
_______ {x < 4}
___________ set! x {x + 1}
___________ display x
_______ {x = 3}
___________ set! x 0
___________ display x
___ newline
```

Folie

atomic

```
define : atomic-switch
___ define a #t
___ define flag #f
___ while-any
_______ a
___________ set! flag #t
___________ set! flag #f
_______ : and flag a
___________ set! a #f
```

Folie

Euclidean

```
define : euclidean a b
___ while-any
________ {a < b} : set! b {b - a}
________ {b < a} : set! a {a - b}
___ values a b
```

Folie

Fairness

```
define : fair
_ define b #t
_ define x #f
_ while-any
_____ b : set! x #t
_____ b : set! x #f
_____ x : set! b #f
_____ x : set! x : not x
```

Folie

colorme

```
define : colorme
___ define P0 0
___ define P1 0
___ define P2 2
___ define P3 2
___ while-any
_______ : or {P0 = P1} {P0 = P2} {P0 = P3}
_________ set! P0 : modulo {P0 + 2} 4
_______ {P1 = P0}
_________ set! P1 : modulo {P1 + 2} 4
_______ {P2 = P0}
_________ set! P2 : modulo {P2 + 2} 4
_______ {P3 = P0}
_________ set! P3 : modulo {P3 + 2} 4
___ values P0 P1 P2 P3
```

Folie

atomic switch

```
define : atomic-switch
___ define a #t
___ define flag #f
___ while-any
_______ a
___________ set! flag #t
___________ set! flag #f
_______ : and flag a
___________ set! a #f
```

Folie

non-atomic switch

```
define : nonatomic-switch
___ define a #t
___ define flag #f
___ while-any
_______ a : set! flag #t
_______ a : set! flag #f
_______ : and flag a
____________ set! a #f
```

Folie

while-any/deterministic

```
define-syntax-rule : while-any guarded ...
 while #t
  cond guarded ...
   else
    break
```

choose-any/correct basics

```
import : ice-9 pretty-print
         srfi :1 lists
         srfi srfi-9
         only (srfi :26) cut
         prefix (fibers channels) fibers:
         prefix (fibers) fibers:
;; fibers 1.0 and 1.1 compat
false-if-exception
    import : fibers internal
false-if-exception
    import : fibers scheduler

random-state-from-platform
```

choose-any/correct (delayed evaluation)

```
define-syntax wrap-all-in-lambda
 lambda : x
  syntax-case x (SEPARATOR)
   : _ (done ...) SEPARATOR
    #` begin (list done ...)
   : _ (done ...) SEPARATOR (guard action ...) guarded ...
    #` wrap-all-in-lambda
     . (done ... (cons (lambda() guard) (lambda() action ...)))
     . SEPARATOR guarded ...
   : _ (guard action ...) guarded ...
    #` wrap-all-in-lambda
     . ((cons (lambda () guard) (lambda() action ...)))
     . SEPARATOR guarded ...
   : _
     . '()
```

choose-any/correct I

```
define : shuffle items
  sort items : λ (x y) {(random:uniform) < 0.5}

define : choose-any/internal guards
     let loop : : guards : shuffle guards
       when : not : null? guards
         let : : guard : car guards
           if ((car guard)) ;; gets and calls the lambda
              : cdr guard ;; gets and calls the lambda
              loop : cdr guards
```

choose-any/correct II

```
define-syntax-rule : choose-any guarded ...
   choose-any/internal
     wrap-all-in-lambda guarded ...
```

while-any/correct

```
define : while-any/internal guards
  while #t
   let loop : : guards : shuffle guards
     when : null? guards
       break
     let : : guard : car guards
       if : (car guard) ;; gets and calls the lambda
          : cdr guard ;; gets and calls the lambda
          loop : cdr guards

define-syntax-rule : while-any guarded ...
  while-any/internal
    wrap-all-in-lambda guarded ...
```

channel tools I

```
define-record-type <channel>
  channel message-count
  . channel?
  message-count
    . channel-message-count
    . channel-message-count-set!
```

channel tools II

```
define : send-message-to chan
  channel-message-count-set! chan
    + 1 : channel-message-count chan

define : receive-message-from chan
  channel-message-count-set! chan
    + -1 : channel-message-count chan

define : empty? chan
  equal? 0 : channel-message-count chan
```

Phase helpers

```
define (phase i) : list-ref phases i
define (i+1%3 i) : modulo {(phase i) + 1} 3
define (i+2%3 i) : modulo {(phase i) + 2} 3
define : neighbors i
    take : drop phases (max 0 {i - 1})
         min 3 {N - {i - 1}} {i + 2}
define : random-phase i
    inexact->exact : floor : * 3 : random:uniform .
```

Zustands-Broadcast all-to-all I

```
define : broadcast init out in
  define V init
  define W '()
  define inqueue '()
  pretty-print V
  while-any
    : not : equal? V W ;; Schritt 1
      send-to-all out : lset-difference equal? V W
      set! W V
      pretty-print W
    : check-in-has-input!? inqueue in ;; Schritt 2
      set! V : apply lset-union equal? V inqueue
      set! inqueue '()
```

Zustands-Broadcast all-to-all II

```
        pretty-print V
    pretty-print V
    . V

define : send-to-all channels value
  for-each : cut fibers:put-message <> value
           . channels
define : receive-from-all channels
  map fibers:get-message channels
define-syntax-rule : check-in-has-input!? inqueue in
  begin
    set! inqueue : receive-from-all in
    : λ _ : not : every empty? inqueue
define : make-buffered-channel
```

Zustands-Broadcast all-to-all III

```
  define chan-in : fibers:make-channel
  define chan-out : fibers:make-channel
  fibers:

define N 3
define init-values : map list : iota N
;; connect every channel to every other channel
define out-channels
  map (λ _ '()) : iota N
define in-channels
  map (λ _ '()) : iota N
let loop : (N N)
  when : not : zero? N
    for-each
```

Zustands-Broadcast all-to-all IV

```
λ (n)
  let-values : ((chan-in chan-out) (fibers:make-chann
    list-set! out-channels {N - 1}
       cons chan : list-ref out-channels {N - 1}
    list-set! in-channels n
       cons chan : list-ref in-channels n
  let : (chan (fibers:make-channel))
    list-set! in-channels {N - 1}
       cons chan : list-ref in-channels {N - 1}
    list-set! out-channels n
       cons chan : list-ref out-channels n
iota {N - 1}
loop {N - 1}
```

Zustands-Broadcast all-to-all V

```
fibers:run-fibers
  λ _
    map
      λ (init out in)
        fibers:spawn-fiber
          λ _
             broadcast init out in
      . init-values out-channels in-channels
  . #:drain? #t
```

Auf strongly connected graph: Jeder Knoten in Richtung der Kanten („in Pfeilrichtung“) erreichbar.

Zustands-Broadcast terminiert

Wertungsfunktion:

$$Y = (V_0, V_1, ..., V_{n-1}, c_0, c_1, ..., c_{m-1}) \tag{8}$$

c Kanalinhalt

V Zustand

In Schritt 1 wächst c.

In Schritt 2 wächst V.

Terminiert, weiß aber nicht, wann.

Logikprogrammierung

Automatisierte Beweise durch Rückführung auf bewiesene Axiome.

- Trivial
  - $\{P\}$ skip $\{P\}$
- Variablenersetzung
  - $\{Q[x \leftarrow E]\}$ x:= E $\{Q\}$
- Minimalableitung:
  - $\{?\}$ x:=1 $\{x=1\}$;
  - $? = (1 = 1) = \mathit{true}$
  - $\{\text{true}\}$ x:= 1 $\{x = 1\}$
- Ebenso:
  - $\{?\}$ x:= 100 $\{x=0\}$
  - $? = (100 = 0) = \mathit{false}$
  - $\{\text{false}\}$ x:= 1 $\{x = 1\}$

Kein Beweis der Terminierung $\Rightarrow$ Safety, nicht Liveness.
Äquivalent zu „Wenn alle Guards false sind, ist der Zustand richtig“.

Prädikatumformung (predicate transformers)

$$wp(S, \mathit{false}) = \mathit{false} \tag{9}$$

- S: Programm
- wp(S, Zielzustand) = Bedingung
- Kein Programm kann false erfüllen

$$wp(\text{while-any}, Q) = \exists k \geq 0 : H_k(Q) \tag{10}$$

- k: Schritte
- $H_k(Q)$: Alle Zustände, die nach k Schritten terminieren.

Beispiel für Prädikatumformung

```
define : toss
  define x 'egal
  choose-any
    #t : set! x 0
    #t : set! x 1
```

$$wp(toss, x = 0) = false \quad (11)$$
$$wp(toss, x = 1) = false \quad (12)$$
$$wp(toss, x = 0 \vee x = 1) = true \quad (13)$$

Verweise I

Friedman, D. P. und Eastlund, C. (2015). *The Little Prover*. MIT Press, ISBN: 978-0262527958.

Ghosh, S. (2015). *Distributed Systems - An Algorithmic Approach*. Computer & Information Science. Chapman & Hall/CRC, 2 edition, ISBN: 978-1466552975.

Hellerstein, J. M. und Alvaro, P. (2019). Keeping CALM: when distributed consistency is easy. *CoRR*, abs/1901.01930, http://arxiv.org/abs/1901.01930.

Bilder: