# Atomare Objekte II

---

Claus Traulsen

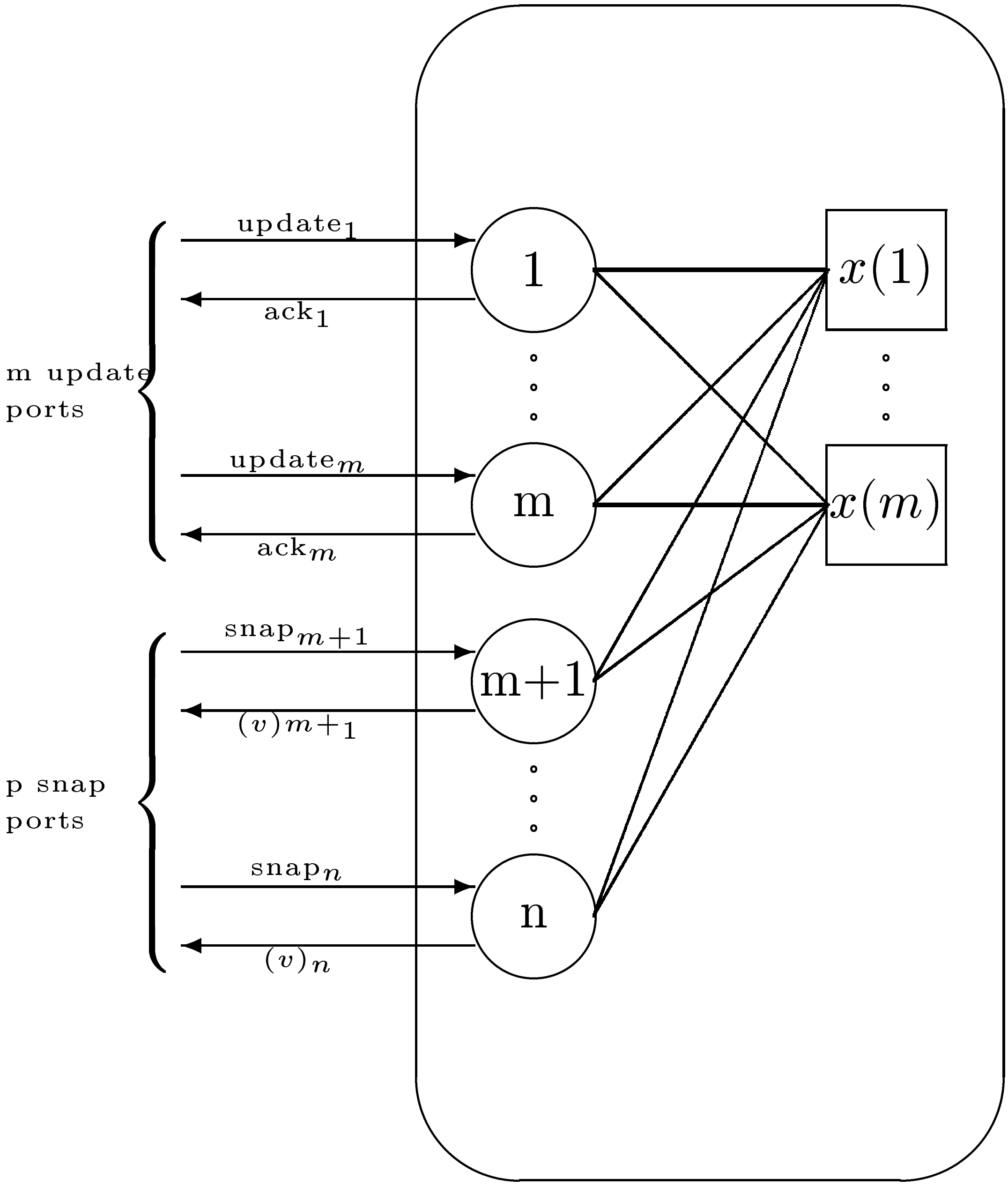
**Atomare Objekte I:** Definition und einfache Atomare Objekte

**jetzt:** Erstellen komplexer Variablentypen:

**snapshot** konsistentes, gleichzeitiges lesen mehrerer gemeinsamer Variablen

**read/write** konsistentes Lesen und Schreiben einer gemeinsamen Variablen durch mehrere Prozesse.
Algorithmen: VitanyiAwerbuch, Bloom

# Snapshot: externe Schnittstelle




schreiben von Variable $i$ nur über Port $i$

# Snapshot: Variablentyp

Gegeben seien m Variablen eines Typs $W$ mit Startwert $w_0$. Der zugehörige Snapshot Variablentyp hat:

**Wertebereich:** $V = W^m$

**Startwert:** $v_0 = (w_0, \ldots, w_0) \in W^m$

**invocations:** *snap*

$update(i, w)$, $w \in W$

**responses:** return($v$), $v \in W^m$

*ack*

**Funktion f** :

$$f(snap, v) = (\mathrm{return}(v), v)$$

$$f(update(i, w), (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m))$$
$$= (ack, (x_1, \ldots, x_{i-1}, w, x_{i+1}, \ldots, x_m))$$

## Snapshot: Beobachtung 1

1. jedes update hat UID und schreibt dies in die Variable
2. wir lesen zweimal *alle* Variablen
3. kein UID wurde verändert
   $\Rightarrow$ der gelesene Zustand gilt vom Ende 1. Lesen bis Anfang 2. Lesen

$\Rightarrow$ einfacher Algorithmus

## Snapshot: Beobachtung 2

1. jedes update hat UID und schreibt dies in die Variable
2. wir lesen alle Variablen mehrmals
3. existiert *eine* Variable, die von 4 verschiedenen updates geschrieben wurde $\Rightarrow$ 3. update vollständig in der Zeit des Lesens enthalten.

## Snapshot: Daten

Wir benutzen m 1-writer/n-reader read/write Variablen $x(i)$ mit den Feldern:

- $x(i).w \in W$, Startwert $w_0$, der aktuelle Wert der Variable
- $x(i).view \in W^m$, Startwert $v_0$, der Wert *aller* Variablen bei, letzten Schreiben von $x(i)$.
- $x(i).tag \in \mathbb{N}$, Startwert 0, UID der letzten update-Operation, die $x(i)$ geschrieben hat.

# Snapshot: Algorithmus

**update**$(w)_i$

read $k := x(i).tag$

read $v := snap(i)$

write $x(i).w = w$

$x(i).tag = k + 1$

$x(i).view = v$

output $ack$

**snap**$_i$

for $k = 1, \ldots$

read $s_{k,1} := x(1)$

$\vdots$

read $s_{k,m} := x(m)$

if $\exists k_1, k_2 \forall i : s_{k_1,i}.tag = s_{k_2,i}.tag$ then

output $(s_{k_1,1}.w, \ldots, s_{k_1,m}).w$

if $\exists i : s_{k_1,i}.tag, \ldots, s_{k_4,i}.tag$ different then

output $s_{k_3,i}.view$

## Snapshot: atomares Objekt?

- Wohlgeformtheit
- wait-free termination:
  snap höchsten $(3m + 1)m$ lesende Zugriffe
- Atomizität: Serialisierungspunkte
  1. update: zu den Zeitpunkt, an dem geschrieben wird
  2. snap mit 2 gleichen Werten: irgendwo zwischen Ende des ersten und dem Anfang des zweiten Lesens
  3. snap mit 4 verschiedenen Werten: zu dem Zeitpunkt, an dem der Serialisierungspunkt für die snap-Operation der dritten update Operation gesetzt wurde.

## Snapshot: beschränkter Algorithmus

Können die tags beschränkt werden?

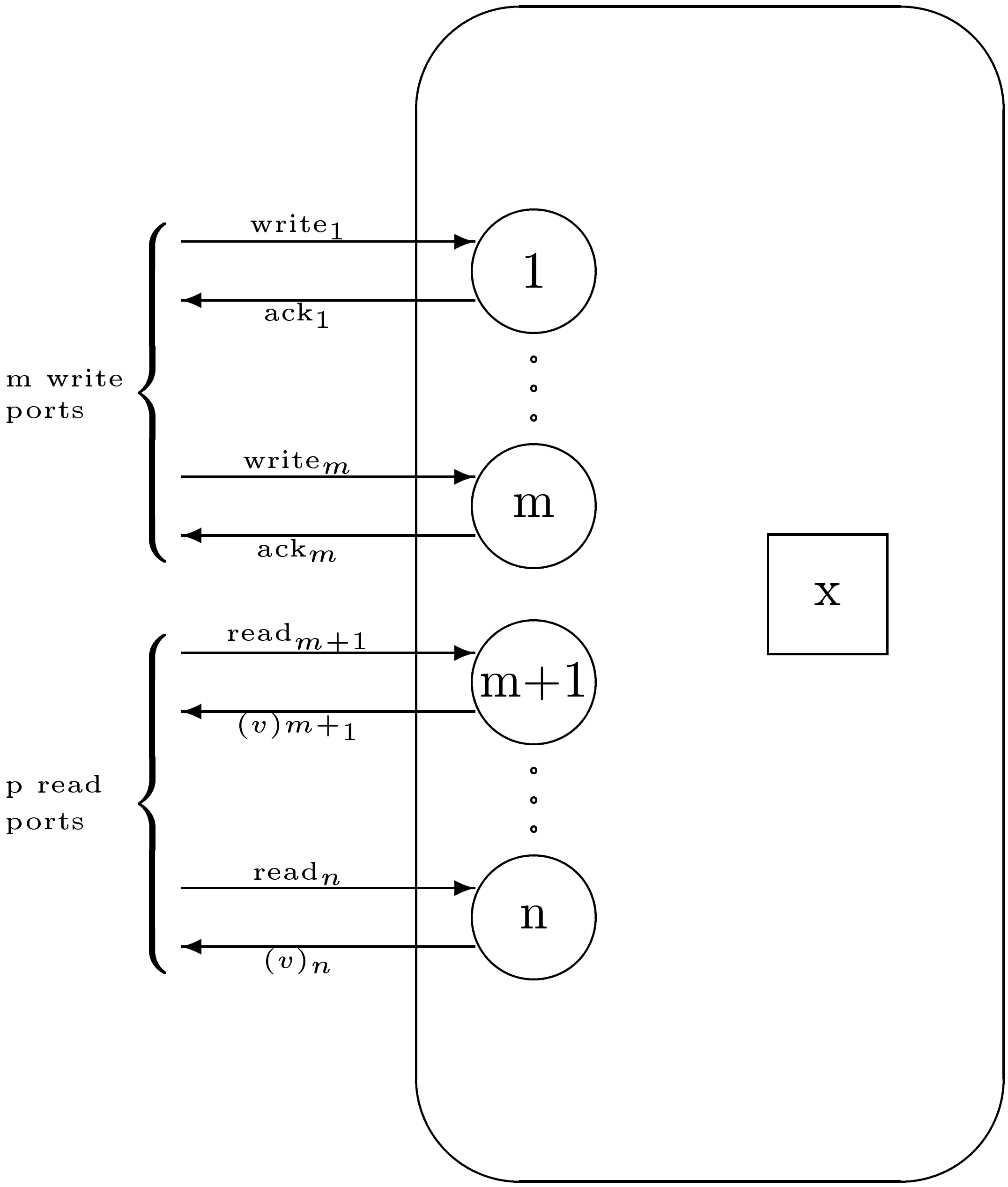
Ja, wir ersetzen die tags durch handshake bits.
speicher statt $tag$:

- zu jedem Prozess: $ack \in \{0,1\}^m$, 1 bit für jede Variable
- zu jeder Variable: 1 toggle-bit und $comm \in \{0,1\}^n$, 1 bit für jeden Prozess

**snap:** setze $ack$ auf entsprechenden Wert in $comm$

**update:** ändere toggle und sezte $comm$ jeweils verschieden zu $ack$

# Read/Write Objekt: externe Schnittstelle




$$n = p + m$$

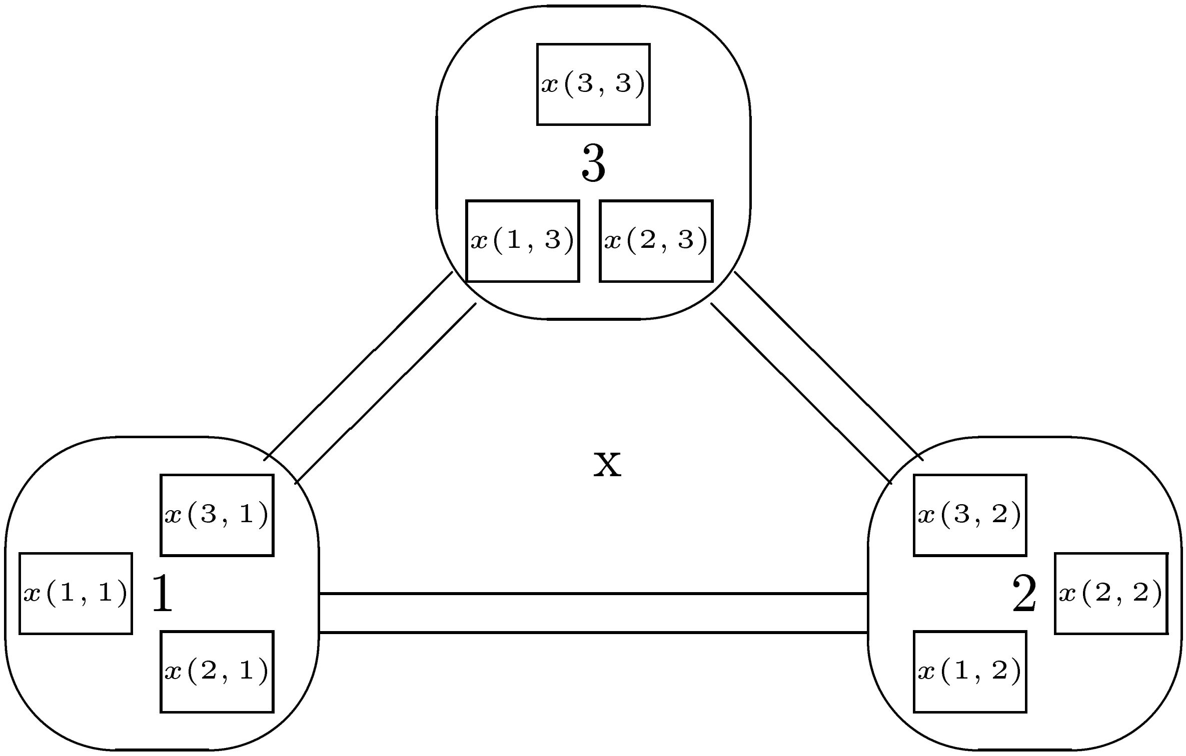
# Atomizität von Read/Write-Objekten

Sei $\beta$ eine Sequenz externer Aktionen eines read/write Objektes. $\beta$ sei für jeden Port $i$ wohlgeformt und enthalte nur abgeschlossene Operationen. Sei $\Pi$ die Menge aller Operationen in $\beta$. Existiere weiterhin eine irreflexive partielle Ordnung $\prec$ auf $\Pi$ mit den folgenden Eigenschaften:

1. Falls response($\pi$) vor invocation($\phi$) auftritt, so gilt *nicht* $\phi \prec \pi$.
2. Falls $\pi$ eine WRITE Operation ist, so gilt für jedes andere $\phi \in \Pi$ $\pi \prec \phi$ oder $\phi \prec \pi$.
3. Jedes READ $\pi$ liefert den Wert zurück, der von dem letzten WRITE $\phi$ mit $\phi \prec \pi$ geschrieben wurde, bzw. $v_0$, falls es kein solches WRITE gibt.

Dann erfüllt $\beta$ die Atomizitätsbedingung.

# VitanyiAwerbuch: Idee




jeder Prozess

- hat $n$ 1-writer/1-reader read/write Variablen
- kann seine eigenen Variablen schreiben
- kann von jedem Prozess eine Variable lesen

$$
\begin{array}{lcccccc}
 & & & \overbrace{\hphantom{x(1,m)}}^{\text{m schreibt}} & & & \\
 & x(1,1) & \cdots & x(1,m) & x(1,m+1) & \cdots & x(1,n) \\
 & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\text{m liest } \{ & x(m,1) & \cdots & x(m,m) & x(m,m+1) & \cdots & x(m,n) \\
 & x(m+1,1) & \cdots & x(m+1,m) & x(m+1,m+1) & \cdots & x(m+1,n) \\
 & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
 & x(n,1) & \cdots & x(n,m) & x(n,m+1) & \cdots & x(n,n)
\end{array}
$$

$x(i,j)$ wird von Prozess $i$ gelesen und von Prozess $j$ geschrieben
jedes $x(i,j)$ hat die Felder:

- $val \in V$, der neueste bekannte Wert
- $tag \in \mathbb{N}$, eindeutige Bestimmung der WRITE-Operation
- $index \in \{0,\ldots,m\}$, der Ursprung des aktuellen Wert

$tagpair \triangleq (tag, index)$

## VitanyiAwerbuch: Ablauf

**write:** schreibt alle Variablen mit neuem maximalen tag und eigenem index

**read:** liest alle Variablen, schreibt den neuesten Wert (den mit dem höchsten tag) einschliesslich index in alle eigenen Variablen und liefert ihn zurück

# VitanyiAwerbuch: WRITE$(v)_i$

**WRITE**$(v)_i$

for $j = 1, \ldots, n$

read $x(i,j)$

$k := \max_{1 \le j \le n} x(i,j).tag$

for $i = 1, \ldots, n$

write $x(i,j).val = v$

$x(i,j).tag = k+1$

$x(i,j).index = i$

output $ack_i$

**READ**$_i$

for $j = 1, \ldots, n$

read $x(i,j)$

let $x(i,k)$ be variable with maximal tagpair

for $j = 1, \ldots, n$

write $x(j,i).val = x(i,k).val$

$x(j,i).tag = x(i,k).tag$

$x(j,i).index = x(i,k).index$

output $(x(i,k).val)_i$

## VitanyiAwerbuch: atomares Objekt?

- Wohlgeformtheit
- wait-free termination:
  jeweils n lesende- und n schreibende
  Zugriffe
- Atomizität: partielle Ordnung

# VitanyiAwerbuch: partielle Ordnung

tagpair($\pi$)=$(tag, index)$ geschrieben von $\pi$

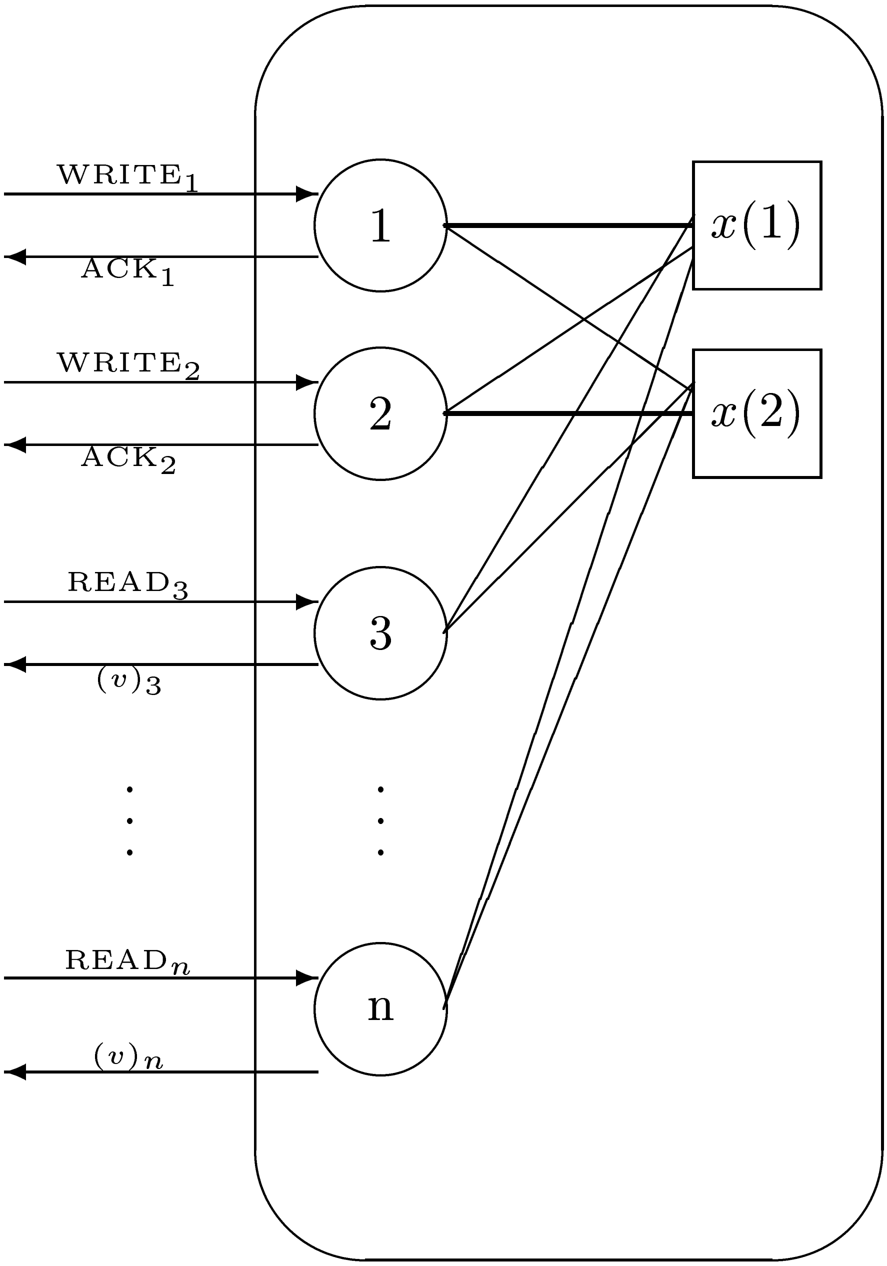
$\pi \prec \phi$ wenn:

1. tagpair($\pi$) $<$ tagpair($\phi$)
2. tagpair($\pi$) $=$ tagpair($\phi$), $\pi$ ist ein WRITE und $\phi$ ist READ

Beachte:

- tagpairs sind monoton steigend
- verschiedene WRITE-Operationen haben verschiedene tagpairs

# Bloom: externe Schnittstelle



# Bloom: Idee

**VitanyiAwerbuch:** 1-writer/1-reader read/write objects

**Bloom:** 1-writer/p+1-reader read/write objects, p Anzahl Leser

- mächtigere Variablen $\Rightarrow$ nur eine Variable pro Schreiber
- eigener tag = fremder tag + 1 ergibt bereits eindeutigen tag, deshalb kein Index-Feld

## IntegerBloom: Datenstruktur

Jede Variable $x(i)$ hat die Felder:

1. $val \in V$, Startwert $v_0$
2. $tag \in \mathbb{N}$, Startwert 0 für $i = 1$, 1 für $i = 2$.

$\bar{i} = \left\{ \begin{smallmatrix} 1 & \text{if} & i=2 \\ 2 & \text{if} & i=1 \end{smallmatrix} \right.$

# IntegerBloom: Algorithmus

**WRITE**$(v)_i$

read $t := x(\bar{i}).tag$

write $x(i).val := v$

$x(i).tag := t + 1$ output $\text{ack}_i$

**READ**$_i$

read $t_1 := x(1).tag$

read $t_2 := x(2).tag$

if $|t_1 - t_2| \leq 1$ then

if $t_1 > t_2$ output read$(x(1).val)$

if $t_2 > t_1$ output read$(x(2).val)$

else if $|t_1 - t_2| > 1$

output(read$(x(k).val)$) $//k \in \{1, 2\}$

## IntegerBloom: atomares Objekt?

- Wohlgeformtheit
- wait-free termination
  read: 4 lesende Zugriffe
  write: 1 lesender, 1 schreibender Zugriff
- Atomizität: partielle Ordnung

# IntegerBloom: Partielle Ordnung

Seien $\pi, \phi$ Operationen.
$\pi \prec \phi$ wenn:

1. beides sind WRITE-Operationen an verschiedenen Ports und $tag(\pi) < tag(\phi)$
2. beides sind WRITE-Operationen am selben Port und $\pi$ tritt vor $\phi$ auf.
3. $\phi$ ist eine READ-Operationen und liest den von $\pi$ geschrieben Wert.
4. $\pi$ ist eine READ-Operation, die einen Wert liest, den eine WRITE-Operation $\psi$ mit $\psi \prec \phi$ geschrieben hat.

# Bloom: beschränkter Algorithmus

Kann die Größe des tags beschränkt werden?

Ja. Die tags werden nur benötigt, um zu erkennen, welche Variable zuletzt geschrieben wurde.Es gilt:
tag bei Prozess 1: gerade
tag bei Prozess 2: ungerade

$\Rightarrow$ letzter Schreiber:
Prozess 1: vorletzten bits sind verschieden
Prozess 2: vorletzten bits sind gleich

# Bloom

**WRITE**$(v)_i$

read $b := x(\bar{i}).tag$

write $x(i).val := v$

$x(i).tag := (b + i) \mod 2$

output $ack_i$

//process 1 makes tags unequal

//process 2 makes tags equal

**READ**$_i$ read $b := (x(1).tag + x(2).tag) \mod 2$

if $b = 1$ then output read$(x(1).val)_i$

if $b = 0$ then output read$(x(2).val)_i$