Atomare Objekte I

Heiko Schmidt

Atomare Objekte I

- Einführung in gemeinsame Variablen, Kapitel 9 aus [Lyn96]
- Definition und elementare Eigenschaften atomarer Objekte, Kapitel 13.1 aus [Lyn96]
- Ein erstes Beispiel eines atomaren Objekts, Kapitel 13.2 aus [Lyn96]

Atomare Objekte II

- Weitere Implementierungen atomarer Objekte, Kapitel 13.3 und 13.4 aus [Lyn96]

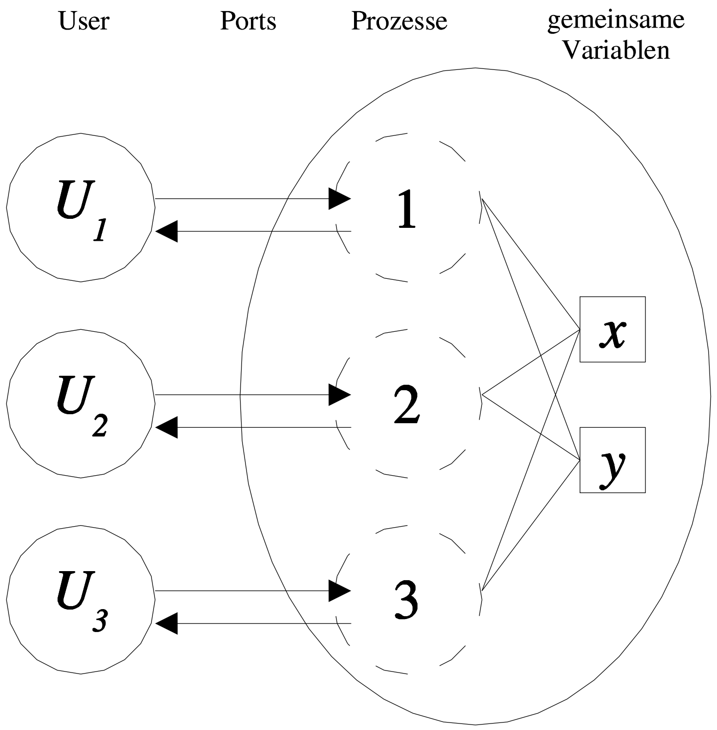
[Lyn96] Nancy Lynch, *Distributed Algorithms*, Kaufmann Publishers, 1996

Einleitung

- Gemeinsame Variablen werden durch atomare Objekte ersetzt.
- Im Gegensatz zu gemeinsamen Variablen kann auf atomare Objekte nebenläufig zugegriffen werden; Aufruf und Antwort sind zeitlich getrennt.
- Operationen auf atomaren Objekten sind **nicht** atomar. Das atomare Objekt tut aber so als ob.
- Atomare Objekte können komplizierte Variablentypen mittels einfacherer Typen realisieren.

$\leadsto$ zunächst Einführung in gemeinsame Variablen

Shared Memory Model



Das gesamte *Shared Memory System* wird als ein I/O-Automaton A modelliert.

- **Zustände:** Zustände aller Prozesse sowie Werte der Variablen
- **Transitionen:** Jede Transition genau einem Prozess zugeordnet, Zugriff auf eine Variable erlaubt

Variablentypen

Gemeinsame Variablen werden vollständig charakterisiert über ihren Typ *(variable type)*. Dieser besteht aus:

- einem Wertebereich V,
- einem Startwert $v_0 \in V$,
- einer Menge von Aufrufen *(invocations)*,
- einer Menge von Antworten *(responses)*,
- einer Funktion

  $$f : invocations \times V \to responses \times V,$$

  die jeder Kombination aus Aufruf und altem Wert $v_{old} \in V$ eine Antwort und einen neuen Wert $v_{new} \in V$ zuweist.

Beispiel 1: Der Variablentyp "Read/Write"

- Wertebereich V beliebig
- Startwert $v_0 \in V$ beliebig
- $invocations = \{read\} \cup \{write(v); v \in V\}$
- $responses = \{return(v); v \in V\} \cup \{ack\}$
- $f(write(v_{new}), v_{old}) = (ack, v_{new})$
  $f(read, v) = (return(v), v)$

Beispiel 2: Der Variablentyp "Read-Modify-Write"

Wir verpacken die nicht-atomare Funktionalität

- Prozess liest v_{old}
- Prozess macht eine Modifikation $v_{new} := h(v_{old})$ und ändert evtl. seinen eigenen Zustand auf Basis von v_{old}
- Prozess schreibt v_{new}

in einen Variablentypen:

- Wertebereich V und Startwert $v_0 \in V$ beliebig
- Als *invocation* wird die Modifikationsfunktion h übergeben
- Als *response* wird der **alte** Wert v_{old} zurückgegeben
- $f(h, v_{old}) = (v_{old}, h(v_{old}))$

Ausführungen von Variablentypen

Eine Ausführung *(execution)* eines Variablentypen ist eine Sequenz

$$v_0, a_1, b_1, v_1, a_2, b_2, v_2, ...$$

mit

- v_0 ist Startwert
- $a_k \in invocations$
- $b_k \in responses$
- $f(a_{k+1}, v_k) = (b_{k+1}, v_{k+1})$

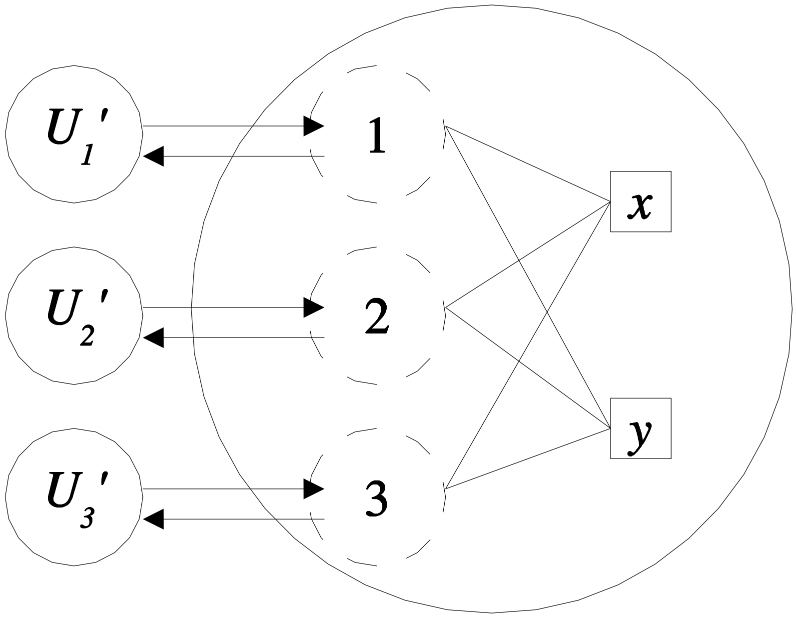
z.B.: $0, read, return(0), 0, write(8), ack, 8$

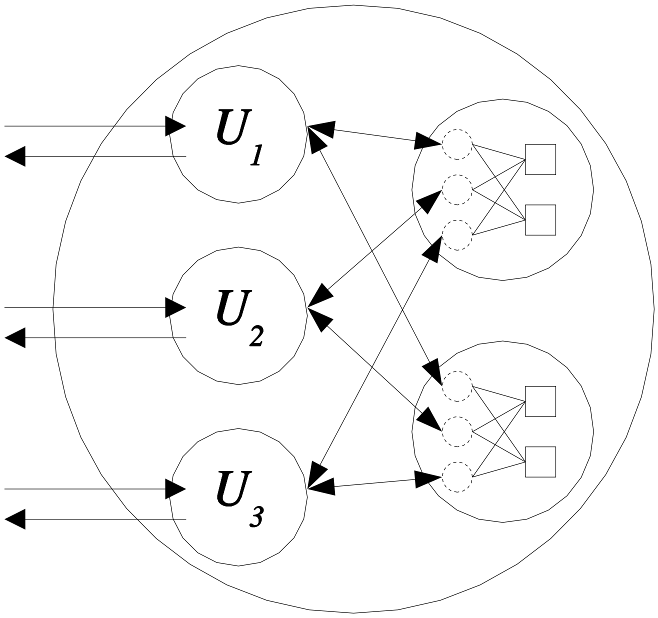
Ein *trace* ist eine *execution*, bei der die v_k weggelassen werden:

$$a_1, b_1, a_2, b_2, a_3, b_3, ...$$

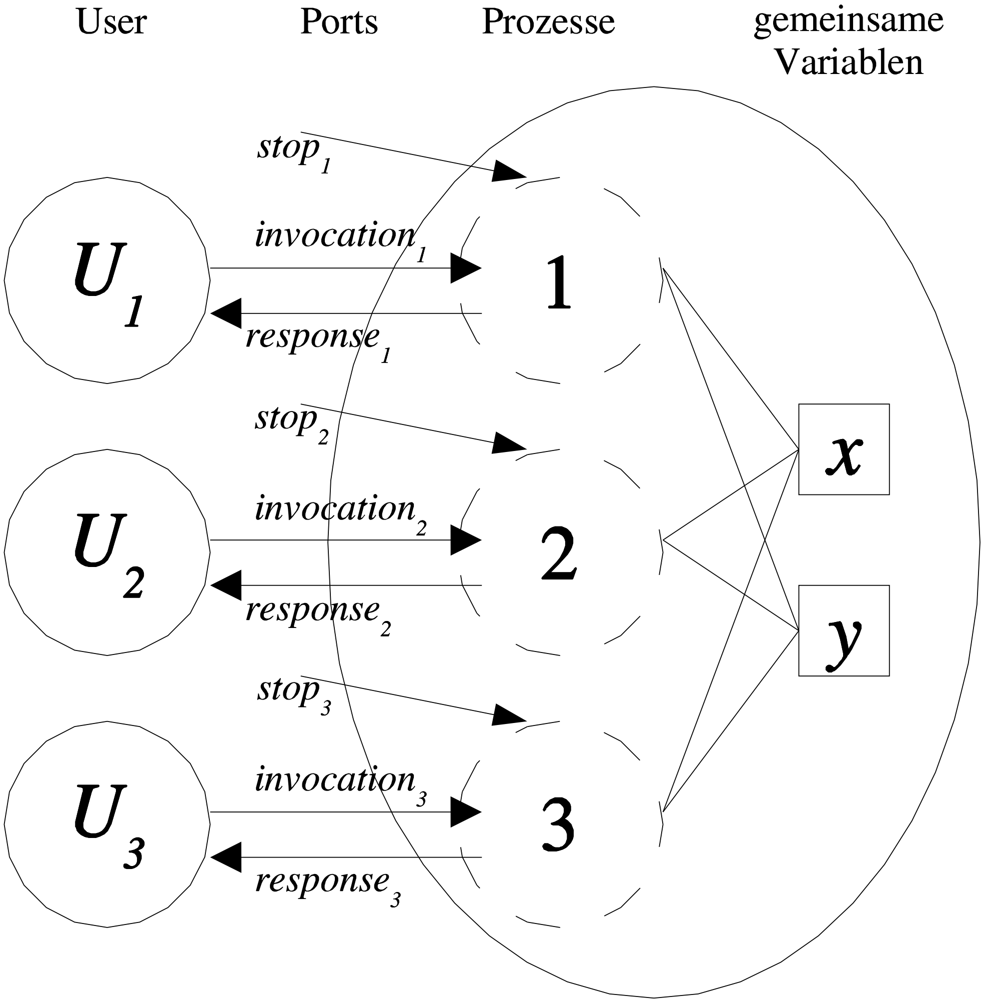
z.B.: $read, return(0), write(8), ack$

Von *Shared Memory Systems* **zu atomaren Objekten**





Aufbau atomarer Objekte



(für einen Variablentypen $\mathcal{T}$)

Definition atomarer Objekte

Ein I/O-Automaton A mit einer wie eben vorgestellten externen Schnittstelle für einen Variablentypen $\mathcal{T}$ nennen wir **atomares Objekt** vom Typ $\mathcal{T}$, wenn es für jede Menge von *wohlgeformten* User-Automata U folgende Bedingungen erfüllt:

- $A \times U$ ist *wohlgeformt*, d.h. Aufrufe und Antworten wechseln sich auf jedem Port ab und es wird mit einem Aufruf begonnen,
- in jeder fairen fehlerfreien Ausführung von $A \times U$ hat jeder Aufruf eine zugehörige Antwort *(failure-free termination)*,
- jede Ausführung von $A \times U$ erfüllt die *Atomizitätsbedingung*.

Die Atomizitätsbedingung

Eine Ausführung α erfüllt die **Atomizitätsbedingung** für einen Variablentypen $\mathcal{T}$ genau dann, wenn es möglich ist,

- für jede abgeschlossene Operation π einen Serialisierungspunkt $*_\pi$ zwischen Aufruf und Antwort platzieren,
- eine Teilmenge Φ der nicht-abgeschlossenen Operationen zu wählen,
- für jedes $\pi \in \Phi$ eine Antwort auszuwählen
- und für jedes $\pi \in \Phi$ einen Serialisierungspunkt $*_\pi$ irgendwo nach dem Aufruf zu platzieren,

so dass man einen Trace von $\mathcal{T}$ erhält, indem man Aufrufe und Antworten auf ihre Serialisierungspunkte zusammenschiebt und ggf. eine ausgewählte Antwort generiert sowie alle Aufrufe von Operationen $\pi \notin \Phi$ entfernt.

Eine hinreichende Bedingung für Atomizität

Falls

- $A \times U$ wohlgeformt ist und *failure-free termination* garantiert
- und jede Ausführung **ohne unvollständige Operationen** die Atomizitätsbedingung erfüllt,

so gilt die Atomizitätsbedingung auch für **alle** Ausführungen.

Fehlertoleranz

Einige strengere Fehlertoleranz-Bedingungen: Wir wollen garantieren, dass jeder Aufruf auf einem fehlerfreien Port eine zugehörige Antwort hat...

- in jeder fairen, fehlerfreien Ausführung *(failure-free termination)*.
- in jeder fairen Ausführung *(wait-free termination)*.
- in jeder fairen Ausführung, in der höchstens f Prozesse stoppen, $0 \leq f \leq n$ *(f-failure termination)*.
- in jeder fairen Ausführung, in der nur Prozesse mit Indizes aus I stoppen, $I \subseteq \{1, ..., n\}$ *(I-failure termination)*.

Das kanonische atomare Objekt C

- Wir konstruieren ein atomares Objekt C des Typs $\mathcal{T}$ und benutzen darin gemeinsame Variablen des Typs $\mathcal{T}$.
- C garantiert *wait-free termination.*
- C hat keinen praktischen Nutzen, ist aber von theoretischem Interesse, da man mit Hilfe von C zeigen kann, dass andere I/O-Automata atomare Objekte sind und *wait-free termination* garantieren.

Das kanonische atomare Objekt C

Signature:

Input:

a_i's as in the given external interface

$stop_i, 1 \leq i \leq n$

Output:

b_i's as in the given external interface

Internal:

$perform(a)_i, a_i$ in the external interface, $1 \leq i \leq n$

$dummy_i, 1 \leq i \leq n$

States:

val, a value in V, initially v_0

inv-buffer, a set of pairs (i, a), for a_i in the ext. interface

resp-buffer, set of pairs (i, b), for b_i in the ext. interface

stopped $\subseteq \{1, ..., n\}$, initially empty

Das kanonische atomare Objekt C

Transitions:

a_i

Effect:

$$inv\text{-}buffer := inv\text{-}buffer \cup \{(i,a)\}$$

$perform_i$

Precondition:

$$(i,a) \in inv\text{-}buffer$$

Effect:

$$inv\text{-}buffer := inv\text{-}buffer - \{(i,a)\}$$
$$(b, val) := f(a, val)$$
$$resp\text{-}buffer := resp\text{-}buffer \cup \{(i,b)\}$$

b_i

Precondition:

$$(i,b) \in resp\text{-}buffer$$

Effect:

$$resp\text{-}buffer := resp\text{-}buffer - \{(i,b)\}$$

Das kanonische atomare Objekt C

Transitions (cont'd):
$stop_i$
Effect:
$stopped := stopped \cup \{i\}$

$dummy_i$
Precondition:
$i \in stopped$
Effect:
none

Tasks:
for every i:
$\{perform(a)_i : a_i$ is an input $\} \cup$
$\{b_i : b_i$ is an output $\} \cup \{dummy_i\}$

C **ist ein atomares Objekt mit** *wait-free termination*

- **Wohlgeformtheit**
- **Wait-free termination:** Sei α faire Ausführung von $C \times U$ ohne Fehler auf Port $i \Rightarrow dummy_i$ nie "enabled".
  Fairness $\Rightarrow$ Jeder Aufruf auf Port i löst ein $perform_i$ und schließlich eine Antwort aus.
- **Atomizität:** Sei α Ausführung von $C \times U$.
  Sei $\Phi := \{\pi : \pi$ unvollständige Operation, aber *performed* $\}$.
  Platziere $*_\pi$ an die Stellen der *performs*.
  $\forall \pi \in \Phi$ wähle als Antwort die von $perform$ berechnete.

C in Beweisen verwenden

Wir können C benutzen, um zu zeigen, dass ein anderes Automaton ein atomares Objekt ist, welches *wait-free termination* garantiert:

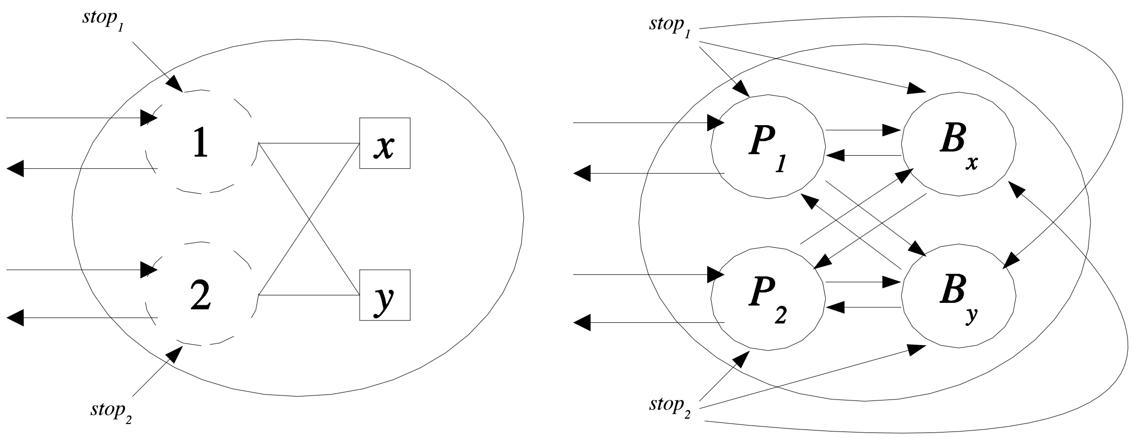
Sei A ein I/O-Automaton mit derselben externen Schnittstelle wie C. Dann sind äquivalent:

- $fairtraces(A \times U) \subseteq fairtraces(C \times U)$ für alle U.
- A ist ein atomares Objekt, welches *wait-free termination* garantiert.

Ersetzung gemeinsamer Variablen durch atomare Objekte

- Erstelle ein System mit gemeinsamen Variablen.
- Ersetze gemeinsame Variablen durch beliebige atomare Objekte desselben Variablentyps.
- Das resultierende System soll von außen nicht unterscheidbar vom ursprünglichen System sein.

Aus A mach $Trans(A)$



- Ersetze Prozess i durch I/O-Automaton P_i, welches im Zeitraum zwischen Aufruf und Antwort wartet.

- Ersetze gemeinsame Variable v durch das atomare Objekt B_v.

- $stop_i$-Botschaften werden an P_i und alle B_v gesendet.

Theorem

Unter der Voraussetzung, dass die P_i keine Inputs von U bekommen, während sie auf die Antwort eines atomaren Objekts warten, gilt folgendes Theorem:

Sei α eine Ausführung von $Trans(A) \times U$. Dann gibt es eine Ausführung α' von $A \times U$, so dass gilt:

- α und α' sind nicht unterscheidbar.
- Für jedes i gibt es ein $stop_i$ in α genau dann, wenn es eines in α' gibt.

Beispiel: Atomares Objekt des Typs Read-Modify-Write

Wir konstruieren ein atomares Objekt vom Typ Read-Modify-Write, welches intern nur eine einfache Read/Write-Variable verwendet.

- Der Wert wird in einer Read/Write-Variable gespeichert.
- Read-Modify-Write-Operationen werden sequentiell realisiert:
  - Lies Wert aus Variable
  - Modifiziere den Wert
  - Schreib Wert in Variable
- Verwende einen Algorithmus für gegenseitigen Ausschluss.

Dieses atomare Objekt garantiert "nur" *failure-free termination.*