Reo: a channel-based coordination model for component composition

(by Farhad Arbab)

presented by Patricia Derler
Software Systems Seminar
University of Salzburg
June 2007

Motivation

- Modular systems
  = components
  - components heavily depend on each other
  - fit into one specific application environment

- Component based systems
  = components + glue code
  - a lot of highly specific glue code
  - hard to maintain

- Reo
  = components + glue code components

Emphasize on what to do with components, not what they are or do.

Reo

Reo comes from the Greek word ***ρεω***

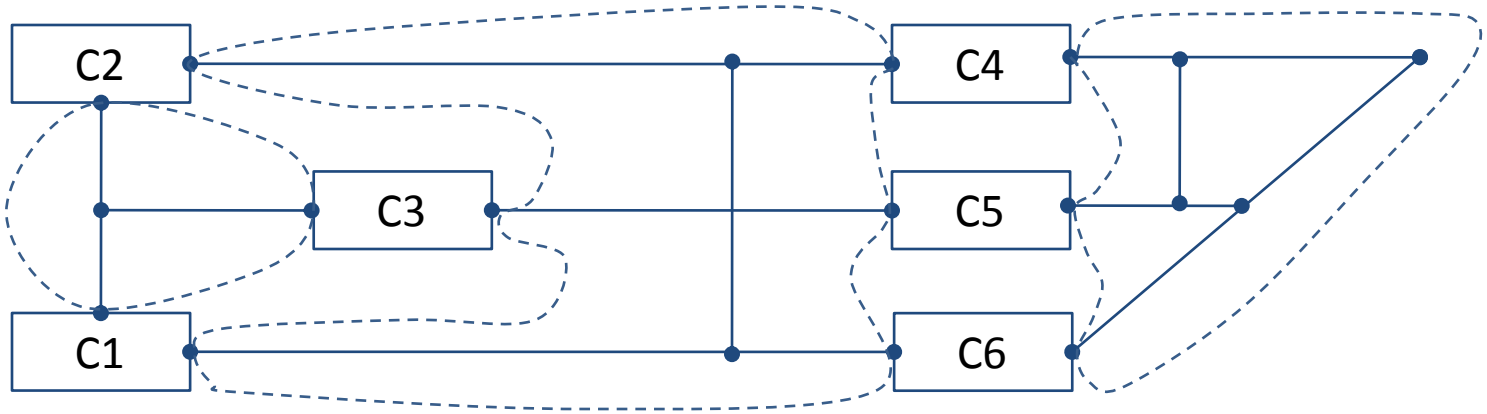
(pronounced ‘rhe-oh’, means ‘ flow‘ as water in streams and channels)

Reo is

- **glue code** in component based software systems
- a **coordination model** for **exogenous** coordination (= **orchestration**)
  of **entities = component instances**
  with complex **coordinators = connectors**
  - compositionally construction of **connectors**
  - simplest connector: **channels**
- glue code that only contains **interaction protocol** (e.g. ordering, timing, data dependency)

Coordinate Components without knowing what they are and what they do!

Example

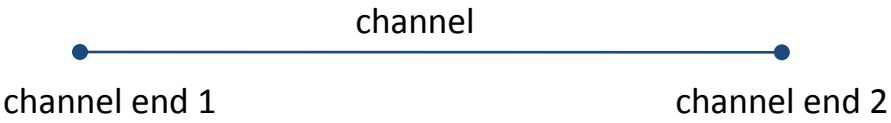


Component instance
- Black box
- non-empty set of **active entities** like process, agent, thread, actor
- Only means of communication is **I/O operation** on channels
- Executed on physical or logical **devices** in a location

Channel
Connecting two component instances

Connector
3-way connector
6-way connector

Channels



- **Point-to-point** medium of communication
- **Atomic** connector
- Directionless
- Used to transfer data using **input/output operations**
- Has exactly **two channel ends**

  Channel end:
  - Types
    - Sources: data enters into the channel
    - Sinks: data leaves the channel
  - Connected to at most one component instance
  - Can be used by any active entity inside the component instance

Channel Types

Synchronous Channels:

Synchronous channel

Synchronous drain

Synchronous spout

Lossy synchronous channel

sink accepts data only when matching I/O operation exists

Filter channel

transfers only data items matching a pattern

Asynchronous Channels:

FIFO channel

Asynchronous drain

Asynchronous spout

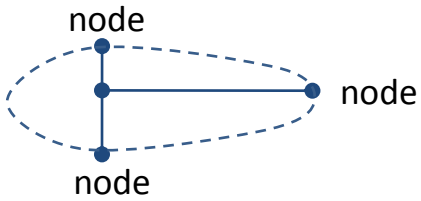
Lossy FIFOn channel

drop newest data when buffer is full

Shift FIFOn channel

drop oldest data when buffer is full

Connectors



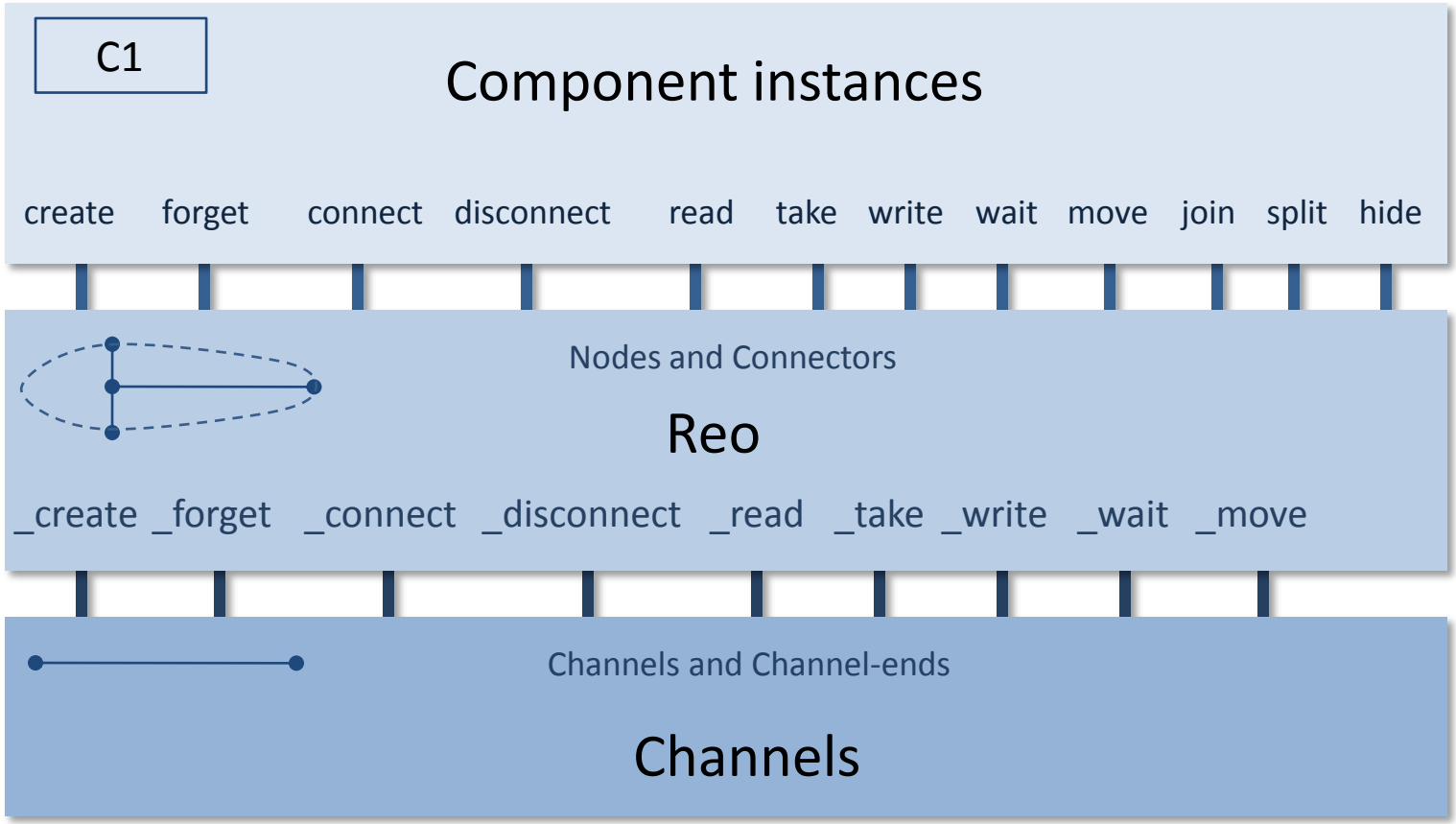
- Set of channel ends with connecting channels
- Directionless

- Organized in a graph with nodes and edges
  - A node has 0 or more channel ends
  - Every channel end is at exactly one node
  - An edge between two nodes is a channel

  - Source node: all channel ends are sources
  - Sink node: all channel ends are sinks
  - Mixed node: source and sink channels coincide on the node

- Every channel is a (simple) connector

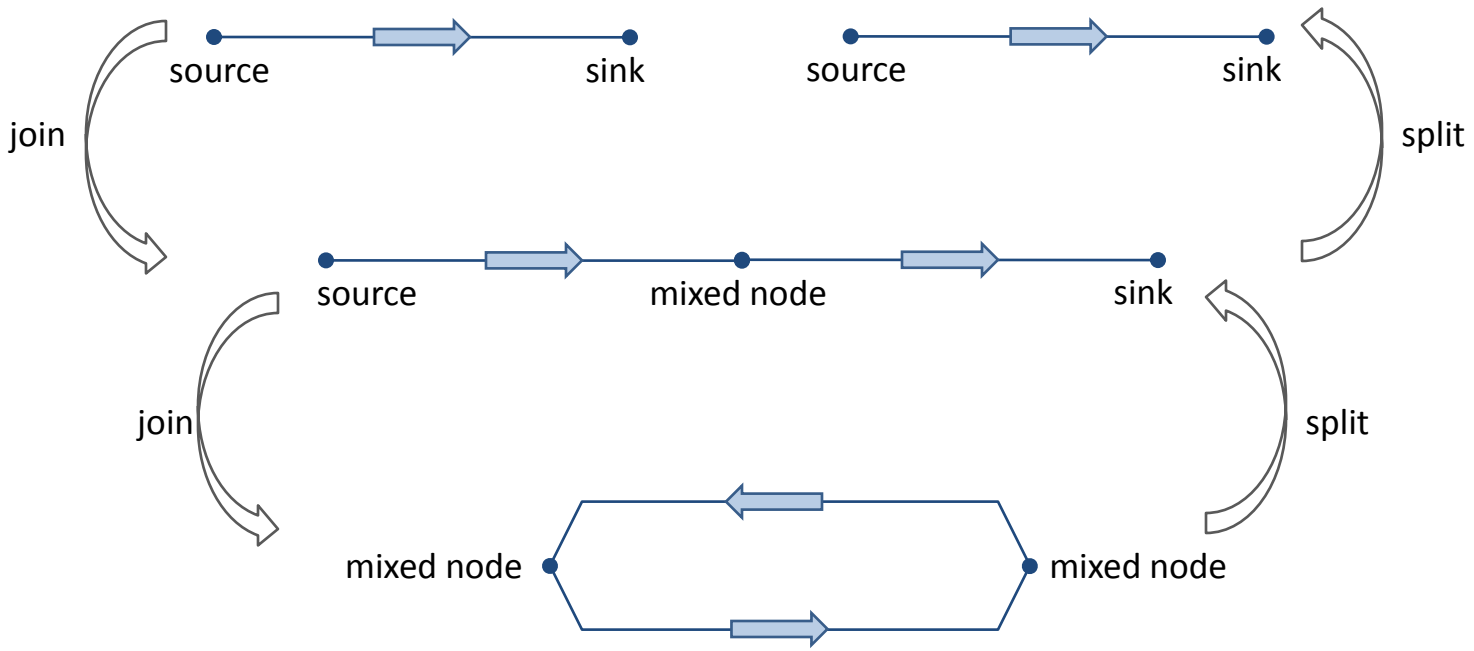
Reo Architecture



Node Operations

Join and Split operations

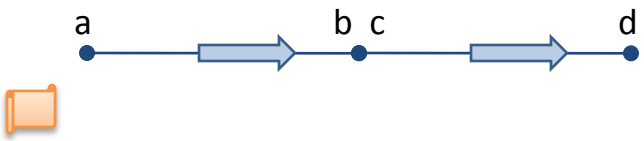
By joining the sink and the source ends of two channels, a new connector is created.



Channel Composition

- Flow-through connector

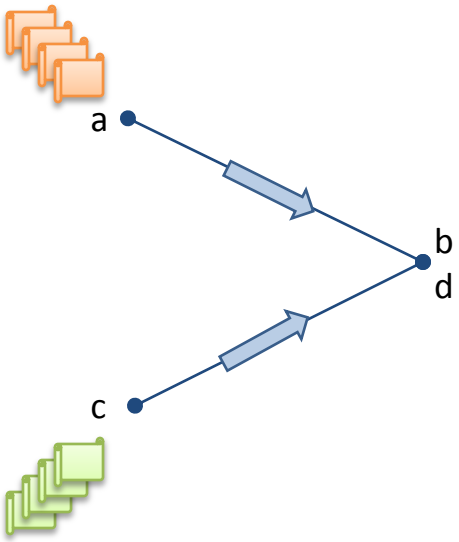
  data items flow through junction node



Channel Composition

- Merger

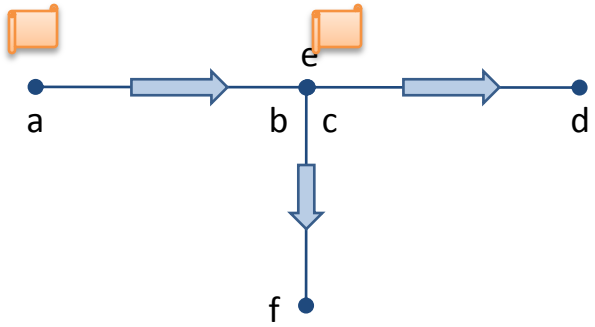
  b and d form a common node



non-deterministic merge of values from a and b

Channel Composition

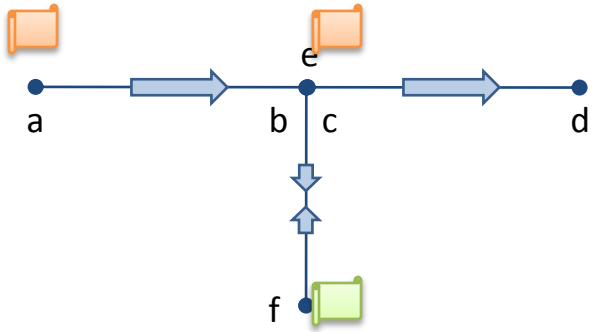
- Take-cue regulator



- can count or regulate data flow from ab to cd
- take operations on f regulate the flow

Channel Composition

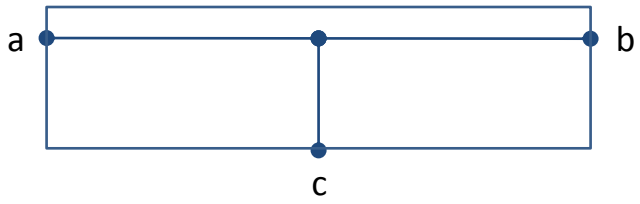
- Write-cue regulator



- write operations on f regulate the flow

Channel Composition

- Encapsulation and abstraction



The whole box is a ‘connector component’

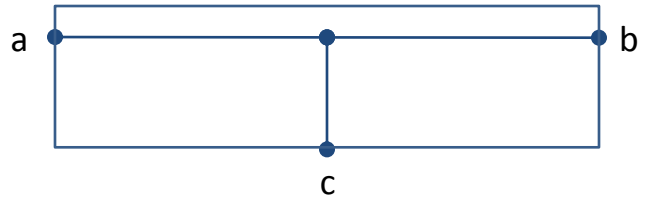
Channel Composition

- Generic write-cue regulator

```
WCRegulator(n)
   <a, x1> = create(Sync)
   <x2, b> = create(Sync)
   <x, y> = create(SyncDrain)
   connect(x1)
   connect(x2)
   join(x, x1)
   join(x1, x2)
   hide(x)
   c = <>
   for i = 1 to n do
      <u, w> = create(Sync)
      c = c ∘ (u)
      connect(w)
      join(y, w)
   done
   hide(y)
   return <a, b, c>
```

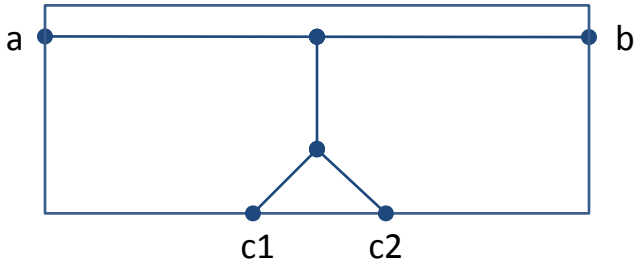
WCRegulator(1)

"a write to c enables transfer of a value from a to b"



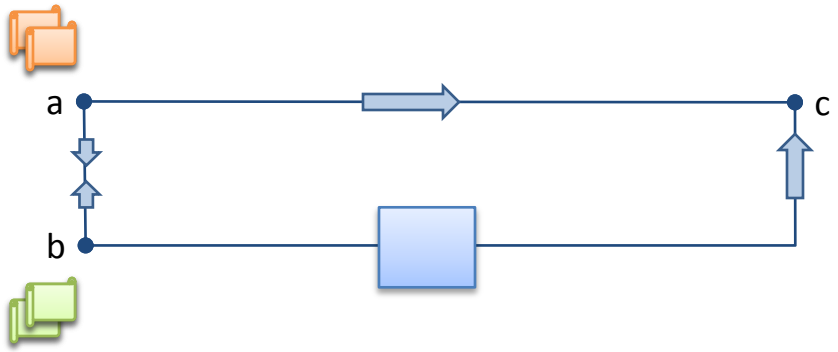
WCRegulator(2)

"a write to c1 or c2 enables transfer of a value from a to b"



Channel Composition

- Ordering



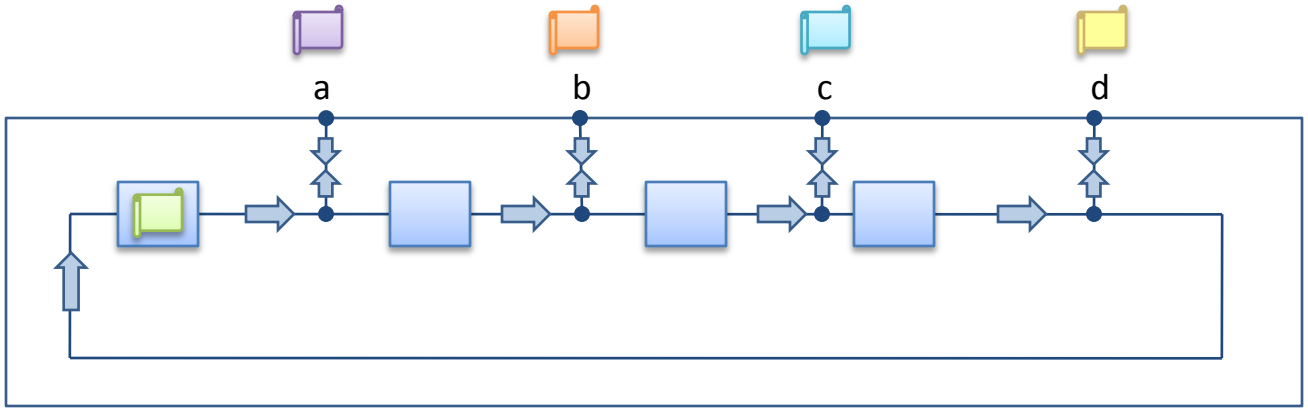
Flow of data items written to a and b is ordered
a1, b1, a2, b2, a3, b3, …

c = (ab)*

Remember: A value can only be written from a to c if a value on b is available

Channel Composition

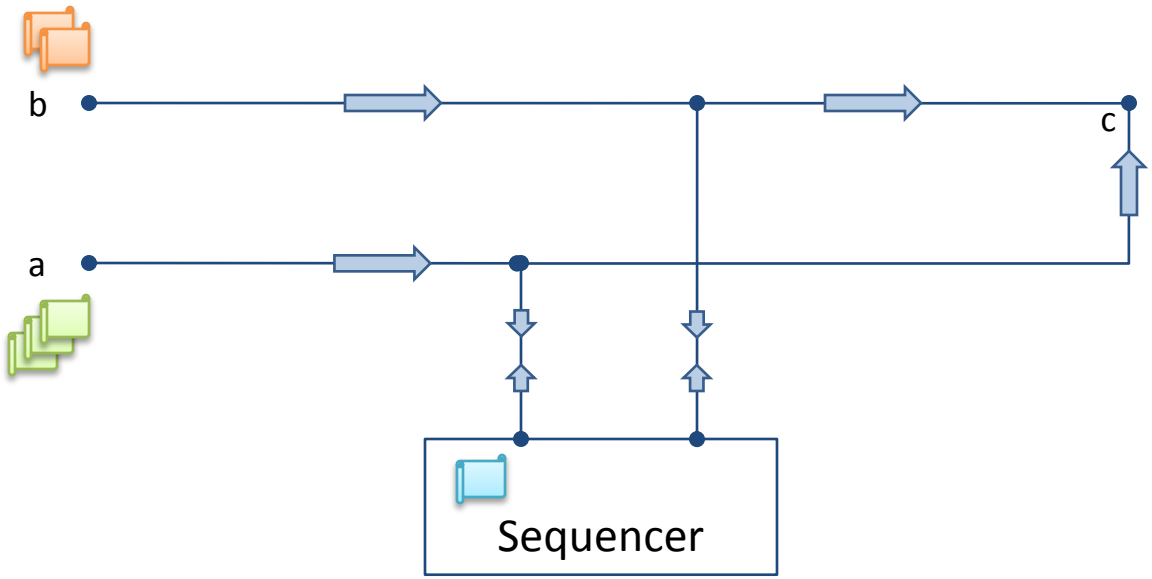
- Sequencer



- Take data out only in strict left-right order
- Generic Sequencer Controller: add or remove channels

Channel Composition

- Utility of sequencer I

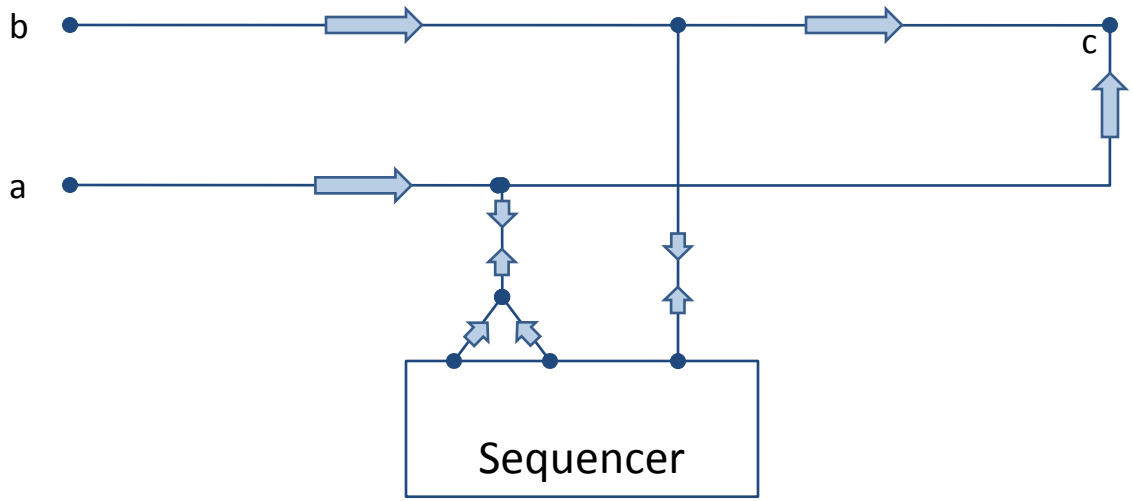


c = (ab)*

Write to a succeeds without availablity of value in b

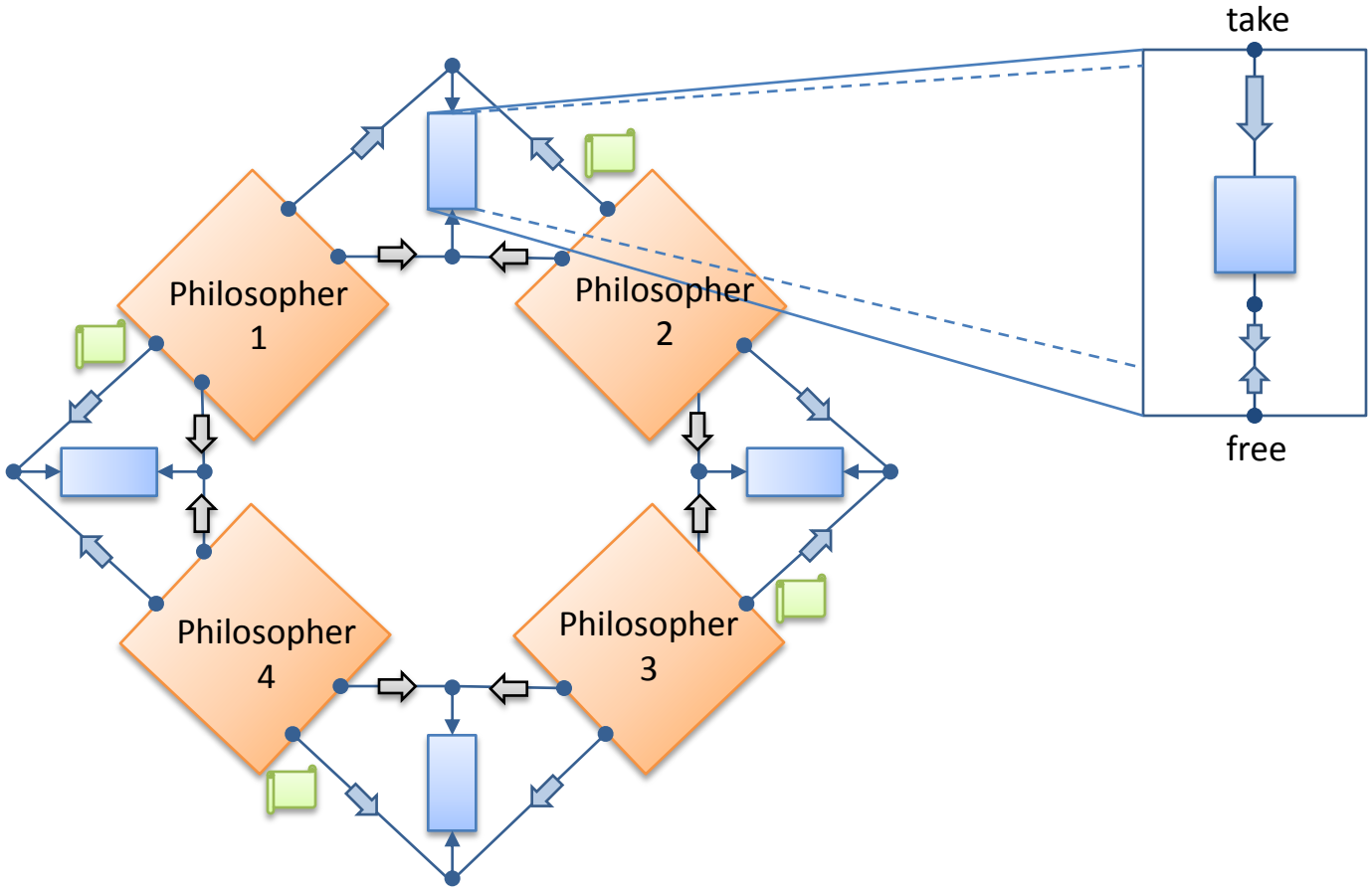
Channel Composition

- Utility of sequencer II

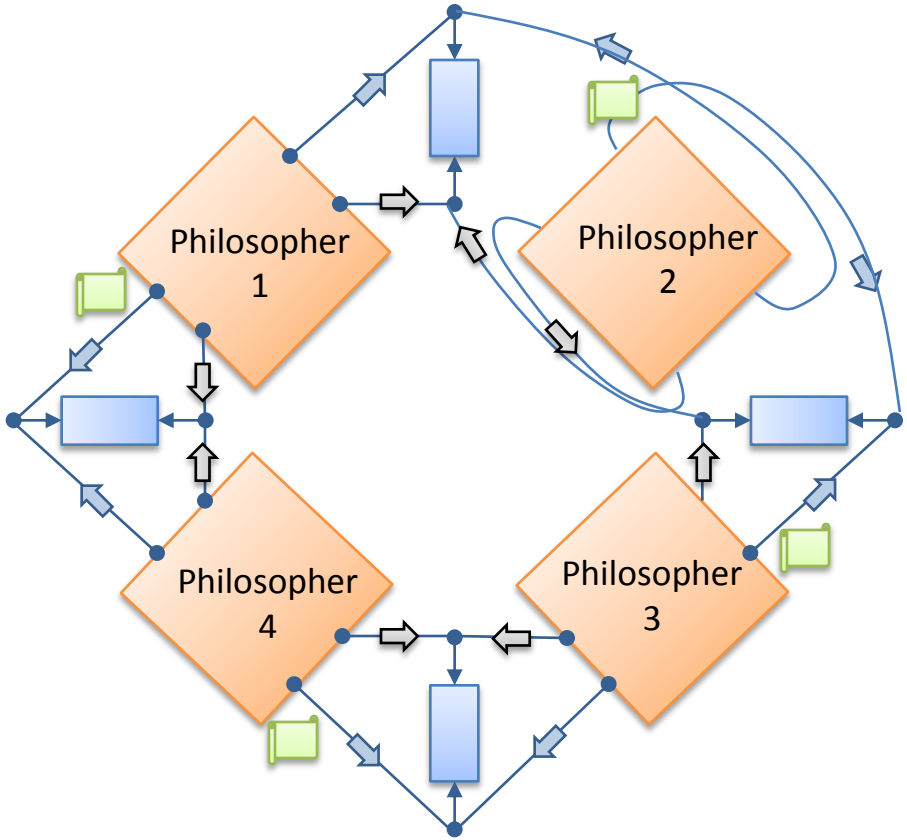


c = (aab)*

Dining Philosophers



Dining Philosophers



Summary

- Reo is a powerful expressive **coordination language**.
- Means for coordination are **connectors**.
- **Complex connectors** are built out of simpler ones.
- Connectors **don't know about entities** that use them.
- Concept is intuitive because of the **relation to physical data flows.**
- Concept allows **visual programming**.
- Topology of connectors is **dynamic** and **mobile**.
- **Secure implementation possible** (shared data space everybody can look, with channels not)