


# Interaction-centric Programming



Farhad Arbab

Center for Mathematics and Computer Science (CWI), Amsterdam  
Leiden Institute of Advanced Computer Science, Leiden University

23 May 2019

# Software Engineering



- Applying engineering discipline to construction of complex software intensive systems.
- A hallmark of all engineering disciplines is composition:
  - Construct more complex systems by composing simpler ones.
  - Derive properties of composed system as a composition of the properties of its constituents.

# Engineering of Complex Systems

- Engineering tackles complexity by:
  - Coping with it: **Practice of Engineering**
    - Methodologies
    - Standards, certification
    - Best practices
    - The art of engineering
  - Simplifying it: **Science behind engineering**
    - Deeper study of the foundational phenomena
    - Appropriate levels of abstraction
    - Formal, mathematical models

# Sources of Complexity

- Complexity inherent in task/algorithm/computation
  - Examples:
    - Computations/equations in quantum mechanics, astronomy, engineering, etc.
    - Bit-map to jpeg conversion, sorting, etc.
  - This type of complexity is not bewildering!
    - Good, intricate mathematical models have tamed the complexity.
- Complexity arising from composition of simple components
  - Example:
    - 4 components send messages to each other (12)
    - Each component can be in one of 4 computation states (256 system states)
    - Exchanges in the context of system state (3072 possibilities)
    - Asynchronous exchange: more to consider!
    - More than a single type of message: multiplicatively more to consider!
  - Bewildering complexity emerges out of **interaction**
  - Good formal models to tame this complexity?

# The way we program(med)

## Sequential

- Using progressively more abstract constructs
  - Machine code and assembly
  - Fortran, Cobol, Algol, PL/I, ...
  - Lisp, APL: functional abstraction
  - Rigorous type systems
  - Abstract data types
  - Objects & classes
  - Prolog: logic programming
  - Haskell: monads and monoids
- Higher-level abstractions
  - Simplify expressing intention
  - Facilitate reasoning and proofs
  - Produce more efficient executables (than hand-crafted code)

## Concurrent

- As tasks, processes, threads, etc., using primitives like
  - Locks & Mutex prehistoric
  - Semaphores (Dijkstra) 1962/1963
  - Monitors (Brinch Hansen & Hoare) 1973/'74
  - CSP (Hoare) 1978
  - $\pi$ -calculus (Milner) 1973-1980
  - Rendezvous (Ada) 1980
  - ACP (Bergstra & Klop) 1982
- Still use 40-50 year-old primitives!
- Lower-level abstractions
  - Complicate expressing intention
  - Hinder reasoning and proofs
  - Need top skills to get efficient executables (by hand-craft optimization)

# Agenda



- Affirm that there exists a better way to conceive of and express concurrency protocols using language constructs in higher-levels of abstraction.
- Introduce a concrete programming language that offers such constructs.

# Concurrent systems

- ❑ The discourse in traditional models of concurrency concerns *actions/processes/actors* and their composition, **not** *interaction*.
  - Petri nets
  - Work flow / Data flow
  - Process algebra / calculi; thread programming; shared memory
  - Actor models; Agents; active objects
  - They model *things that interact*, not *interaction!*
- ❑ Composition of **actions** *does not* yield composition of **interaction!**
- ❑ Interaction becomes an implicit side-effect
  - More difficult to specify, verify, manipulate, and/or reuse

# Producers and Consumer

- Construct an application consisting of:
  - A **Display** consumer process
  - A **Green** producer process
  - A **Red** producer process
- The **Display** consumer must display the contents made available alternately by the **Green** and the **Red** producers.



# Java-like Implementation

## □ Shared entities

```
private final Semaphore greenSemaphore = new Semaphore(1);
private final Semaphore redSemaphore = new Semaphore(0);
private final Semaphore bufferSemaphore = new Semaphore(1);
private String buffer = EMPTY;
```

## □ Consumer

```
while (true) {
    sleep (4000);
    bufferSemaphore.acquire();
    if (buffer != EMPTY) {
        println(buffer);
        buffer = EMPTY;
    }
    bufferSemaphore.release();
}
```

## □ Producers

```
while (true) {
    sleep (5000);
    greenText = ...
    greenSemaphore.acquire();
    bufferSemaphore.acquire();
    buffer = greenText;
    bufferSemaphore.release();
    redSemaphore.release();
}
```

• Where is green text computed?

• Where is red text computed?

• Where is text printed?

• Where is the protocol?

• What determines who goes first?

• What determines producers alternate?

• What provides buffer protection?

• Deadlocks?

• Live-locks?

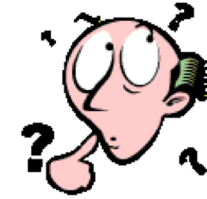
• ...

• Protocol becomes

• Implicit, nebulous, and intangible

• Difficult to reuse

```
while (true) {
    sleep (3000);
    redText = ...
    redSemaphore.acquire();
    bufferSemaphore.acquire();
    buffer = redText;
    bufferSemaphore.release();
    greenSemaphore.release();
}
```



# Process Algebras

- Calculus to contrive expressions of action compositions.
  - Composition operators, e.g.:  $.$ ,  $|$ ,  $+$ ,  $:=$ , implied recursion
- Abstract away the clutter of computation details.
- Enable reasoning through rules of an algebra.

Shared names:  $g, r, b, d$

Consumer:  $B := ?b(t) . \text{print}(t) . !d(\text{"done"}) . B$

Green producer:  $G := ?g(k) . \text{genG}(t) . !b(t) . ?d(j) . !r(k) . G$

Red producer:  $R := ?r(k) . \text{genR}(t) . !b(t) . ?d(j) . !g(k) . R$

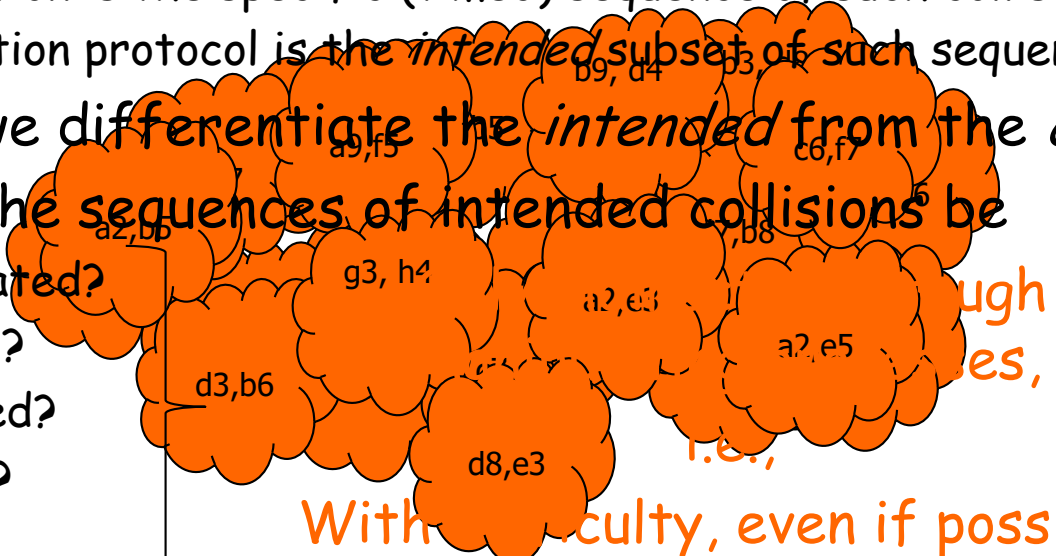
Model:  $G | R | B | !g(\text{"token"})$

- Composition of actions yields more complex actions!
  - Hence the name "process algebra"!
- Where is interaction?



Duh!

# Implicit Interaction

- ❑ Interaction (protocol) is implicit in action-based models of concurrency
  - ❑ Interaction is a by-product of processes executing their actions
    - Action  $a_i$  of process A collides with action  $b_j$  of process B
    - Interaction is the specific (timed) sequence of such collisions in a run
    - Interaction protocol is the intended subset of such sequences.
  - ❑ How can we differentiate the *intended* from the *coincidental*?
  - ❑ How can the sequences of intended collisions be
    - Manipulated?
    - Verified?
    - Debugged?
    - Reused ?
    - ...
- 
- ough  
es,  
i.e.,  
With difficulty, even if possible!

# Construction of artifacts



## □ Direct methods

- The desired artifact is constructed by composing smaller pieces of that same artifact.
- Artifact properties more likely to correspond compositionally to those of its parts.
- Simpler specification, analysis, and construction.

## □ Indirect methods

- The desired artifact is the by-product, side-effect, or indirect result of some other constructed product.
- Artifact properties less likely to relate compositionally to those of the ingredients in its construction.
- More complex specification, analysis, and construction.

# Direct construction



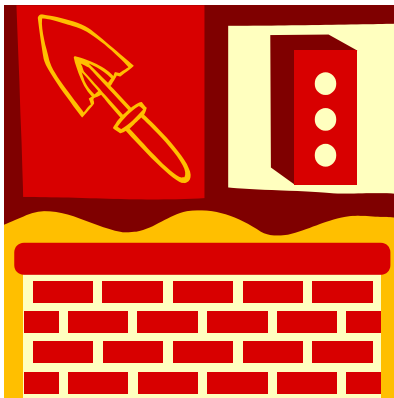
Desired Artifact



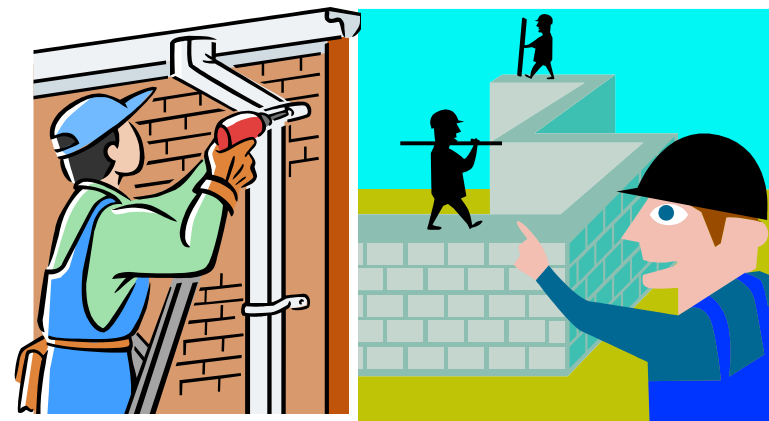
Specification



Analysis



Composition operator and primitives



Construction

# Direct construction



Desired Artifact

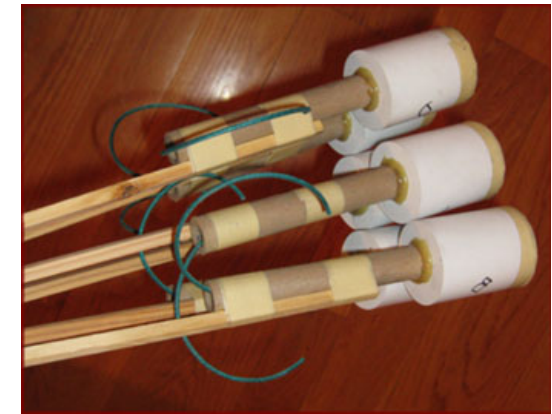


Composition operator and primitives

# Indirect construction



Desired Artifact



Constructed Artifact



Ingredients

# Sequential software



- We construct sequential programs ...
  - out of primitive “program fragments”
    - Constants, variables, etc.
  - That composition operators ...
    - Arithmetic, relational, assignment, etc.
  - Turn into more complex sequential programs ...
    - Statements
  - That other composition operators ...
    - Sequential composition, if-then-else, do-od, etc.
  - Turn into finished programs.
- High-level programming languages try to keep constructed artifacts (programs) “mistakably” close to desired artifacts (computations).



# Different views of interaction

- ❑ The interesting\* side of concurrency is *interaction*, not action!
- ❑ An action is a mere "half-interaction" in a binary interaction.
- ❑ An action is an **interaction-shard** in a multiparty interaction.
- ❑ ~~Alternative to algebra of interaction-shards?~~ **Alternative to algebra of interaction-shards?** than necessary when done through its shards
- ❑ Our failure to take **interaction** seriously as a first-class concept has made concurrency (simple interactions) programming more complex than necessary.
- ❑ ~~Unmanageable otherwise: increasingly the case~~ First-class concept.
  - Explicit construct to capture the concept
  - Composition operators, ideally, forming an algebra.
- ❑ Make action the implicit concept!



\*As in: intriguing, exciting, challenging, exacting, difficult, arduous, grueling, herculean, laborious, curse!

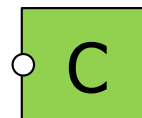
# Concurrency by interaction

- A concurrent system consists of *actors* that *interact*.
  - An actor may itself contain nested interacting actors.
  - An atomic actor performs a sequential computation.
- Specification of a concurrent system:
  - What does each actor do?
    - Specification of computation.
  - What are the permissible interactions amongst actors?
    - Specification of *interaction protocol* as a constraint on ordering of activities and exchanges of partial results amongst independently running actors.

# Interaction centric concurrency (1: actors)

- Specification of a concurrent system in terms of **actors** and their **interaction protocol**.
- **Actors** are **black-box** environment-agnostic processes:
  - Do not share memory
  - Contain no concurrency primitives (locks, semaphores, etc.)
  - Offer no inter-process methods nor make such calls
  - Do not send/receive targeted messages
  - Communicate exclusively by exchange of **values** through **blocking I/O** primitives that they perform only on their own **ports**:
    - `get(p, v)` or `get(p, v, t)`
    - `put(p, v)` or `put(p, v, t)`

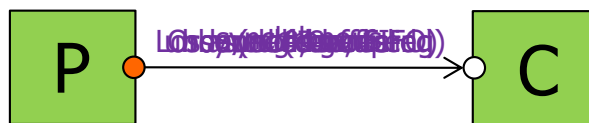
```
while (true)
  sleep(3000);
  redText = ...;
  put(output, redText);
}
```



```
while (true) {
  sleep(4000);
  get(input, displayText);
  print(displayText);
}
```

# Interaction centric concurrency (2: protocols)

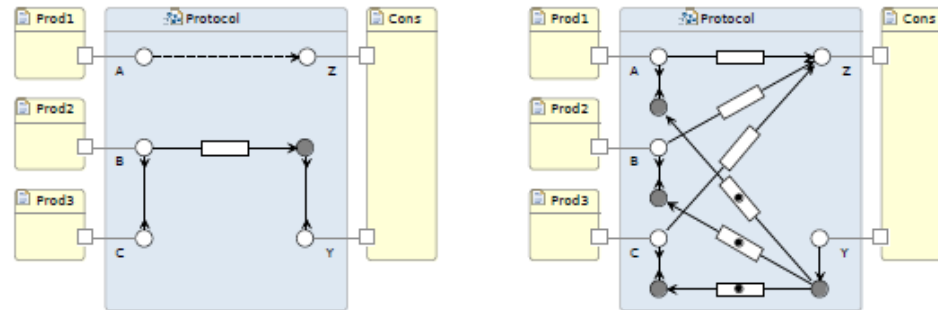
- ❑ **Interaction protocols** are connectors that *exogenously constrain* otherwise arbitrary interaction attempts by actors
- ❑ Composing same processes with different **connectors** yields different systems: **exogenous coordination**



- ❑ **Compositional specification of interaction protocols:**
  - Start with a set of primitive interactions as binary constraints
  - Define (constraint) composition operators to combine interactions into more complex interactions

- Farhad Arbab, Ivan Herman, and Per Spilling, "**Interaction Management of a Window Manager in Manifold**," Proceedings of the Fourth International Conference on Computing and Information, IEEE, Toronto, May 1992.
- Marcello Bonsangue, Farhad Arbab, Jaco de Bakker, Jan Rutten, Adriano Secutella, and Gianluigi Zavattaro, "**A Transition System Semantics for the Control-Driven Coordination Language Manifold**," *Theoretical Computer Science*, Elsevier, Vol. 240, No. 1, pp. 3-47, 2000.
- George A. Papadopoulos and Farhad Arbab, "Coordination Models and Languages," **Advances in Computers**, Vol. 46, Academic Press, 1998.

# Reo



- Reo is a language for compositional construction of interaction protocols.
  - Interaction is the only first-class concept in Reo:
    - Explicit constructs representing interaction
    - Composition operators over interaction constructs (set of interactions is closed under composition operators)
  - Protocols manifest as a connectors
  - In its graphical syntax, connectors are graphs
    - Data items flow through channels represented as edges
    - Boundary nodes permit (components to perform) I/O operations
  - Formal semantics given as ABT (and various other formalisms)
  - Tool support: draw, animate, verify, compile
- F. Arbab "Puff, The Magic Protocol," Formal Modeling: Actors, Open Systems, Biological Systems 2011, SRI International, Menlo Park, California, November 3-4, 2011, Lecture Notes in Computer Science, Springer, vol. 7000, pp. 169-206, 2011.
  - Farhad Arbab, "Reo: A Channel-based Coordination Model for Component Composition," *Mathematical Structures in Computer Science*, Cambridge University Press, Vol. 14, Issue 3, pp. 329-366, June 2004.

# Channels

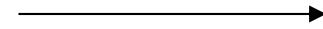


- ❑ Atomic connectors in Reo are called *channels*.
- ❑ Reo generalizes the common notion of channel.
- ❑ A *channel* is an abstract communication medium with:
  - exactly *two ends*; and
  - a *constraint* that relates (the flows of data at) its ends.
- ❑ Two types of channel ends
  - *Source*: data enters into the channel.
  - *Sink*: data leaves the channel.
- ❑ A channel can have two sources or two sinks.
- ❑ A channel represents a *primitive interaction*.

# A Sample of Channels

□ Synchronous channel

○ write/take



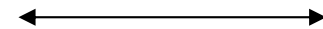
□ Synchronous drain: two sources

○ write/write

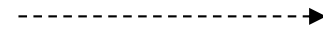


□ Synchronous spout: two sinks

○ take/take

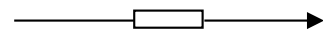


□ Lossy synchronous channel



□ Asynchronous FIFO1 channel

○ write/take



# Join

## □ Mixed node

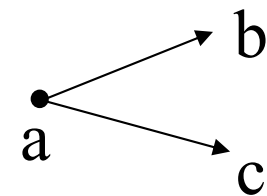
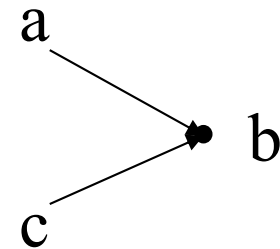
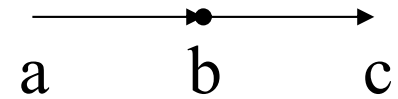
- Atomic merge + replication

## □ Sink node

- Non-deterministic merge

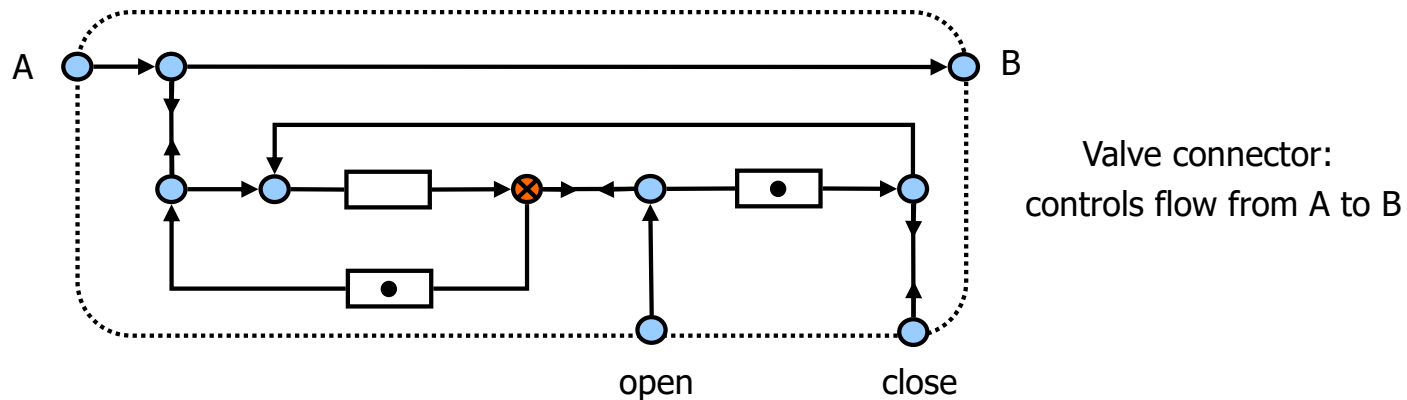
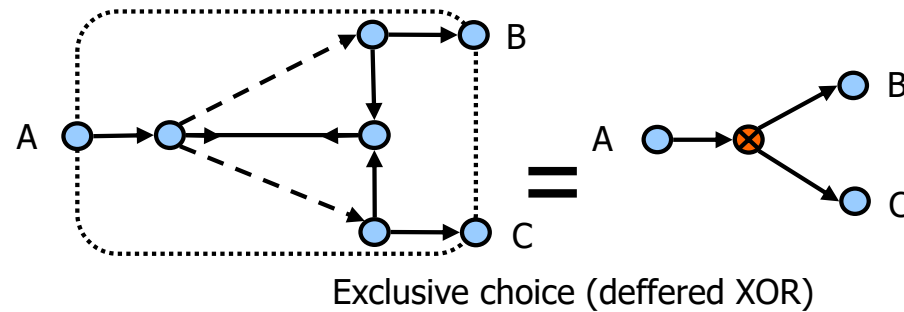
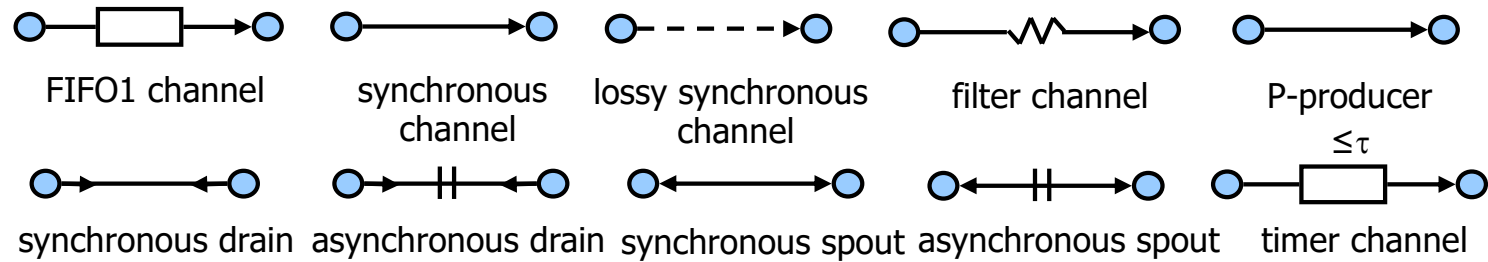
## □ Source node

- Atomic replication



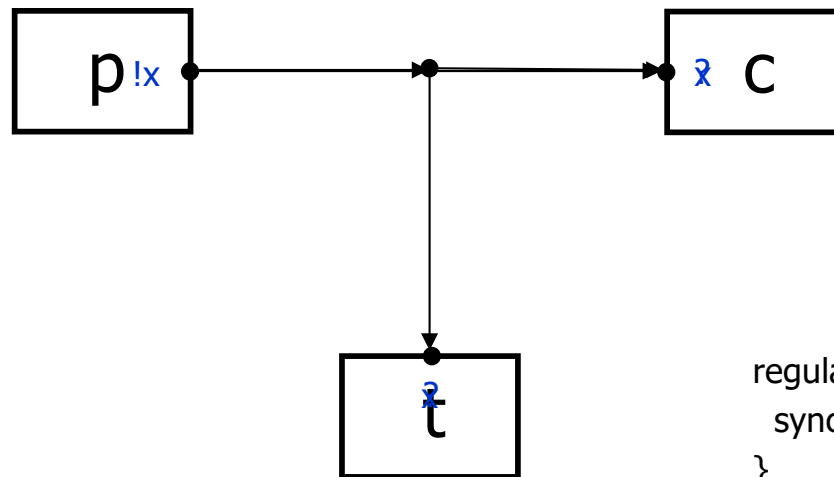


# Reo Connectors



# A Simple Composed System

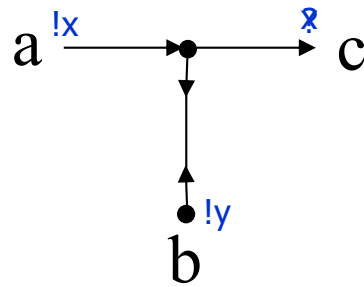
- Read-cue synchronous flow-regulator



```
regulatorwrr(a, b, c) {  
  sync(a, m) sync(m, b) sync(m, c)  
}
```

# Flow regulator

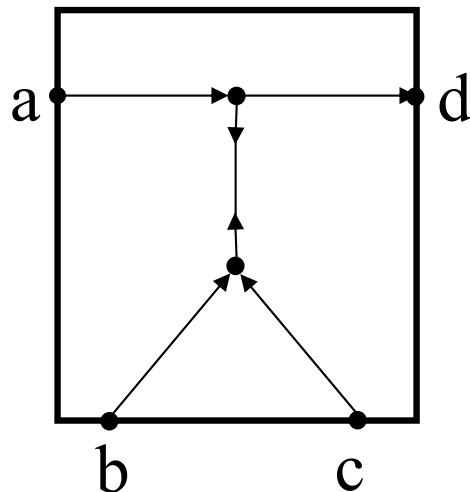
## □ Write-cue synchronous flow-regulator



```
regulatorwvr(a, b, c) {  
  sync(a, m) syncdrain(m, b) sync(m, c)  
}
```

# Take a through d when b or c

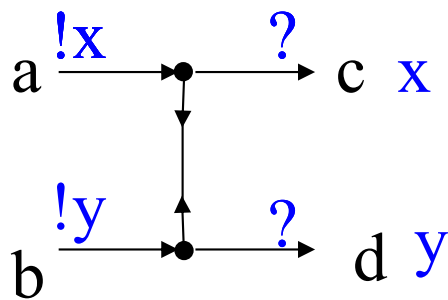
- We have 3 source nodes, a, b, and c, and a sink node, d. Design a Reo circuit for a protocol where:
  - A take from d succeeds only if there is a value written to b or c.
  - The values taken from d are elements of the stream  $a^*$ .



```
circ1(a, b, c, d) {  
  regulatorwvr(a, m, d) sync(b, m) sync(c, m)  
}
```

# Flow Synchronization

- The write/take operations on the pairs of channel ends a/c and b/d are synchronized.
- Barrier synchronization.



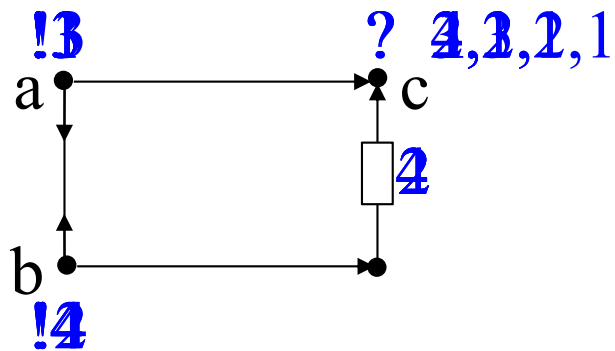
```

barrier(a[b,c], d[1..n]) {
  for (i=0; i<n; i++) {
    sync(x[i], z[i]) sync(z[i], y[i])
  }
}

```

# Alternator

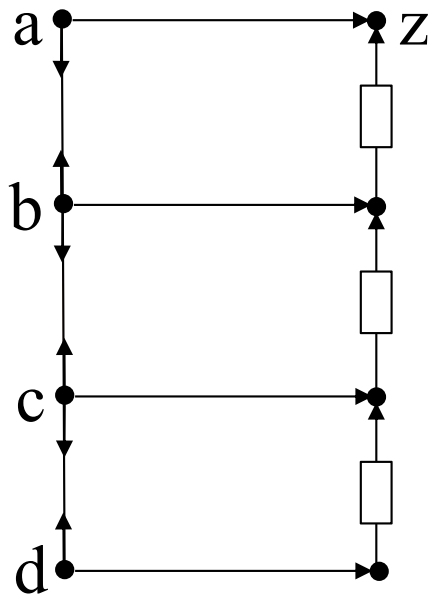
- Subsequent takes from c retrieve the elements of the stream
- Both a and b must be present before a pair can go through.



```
alternator(a, b, c) {  
  syncdrain(a, b) sync(b, x) fifo(x, c)  
  sync(a, c)  
}
```

# N-Alternator

- Subsequent takes from  $z$  retrieve the elements of the stream  $(abcd)^\omega$



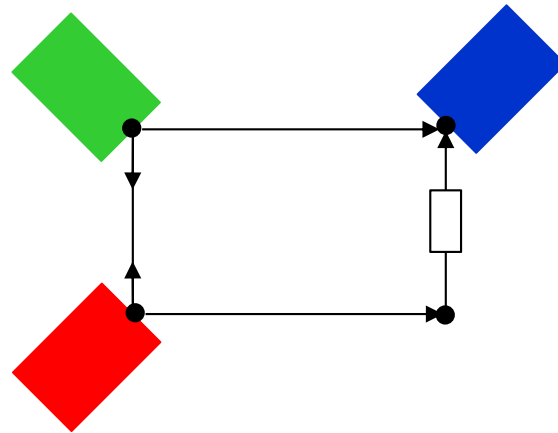
```
alternator(p[1..n], x[1]) {  
  for i = 2 .. n {syncdrain(p[i-1], p[i]) sync(p[i], x[i]) fifo(x[i], x[i-1])}  
  sync(p[1], x[1])  
}
```

# Alternating Producers

- We can use the alternator circuit to impose the protocol on the green and red producers of our example
  - From outside
  - Without their knowledge

```
while (true) {  
  sleep(5000);  
  greenText = ...;  
  put(output, greenText);  
}
```

```
while (true)  
  sleep(3000);  
  redText = ...;  
  put(output, redText);  
}
```



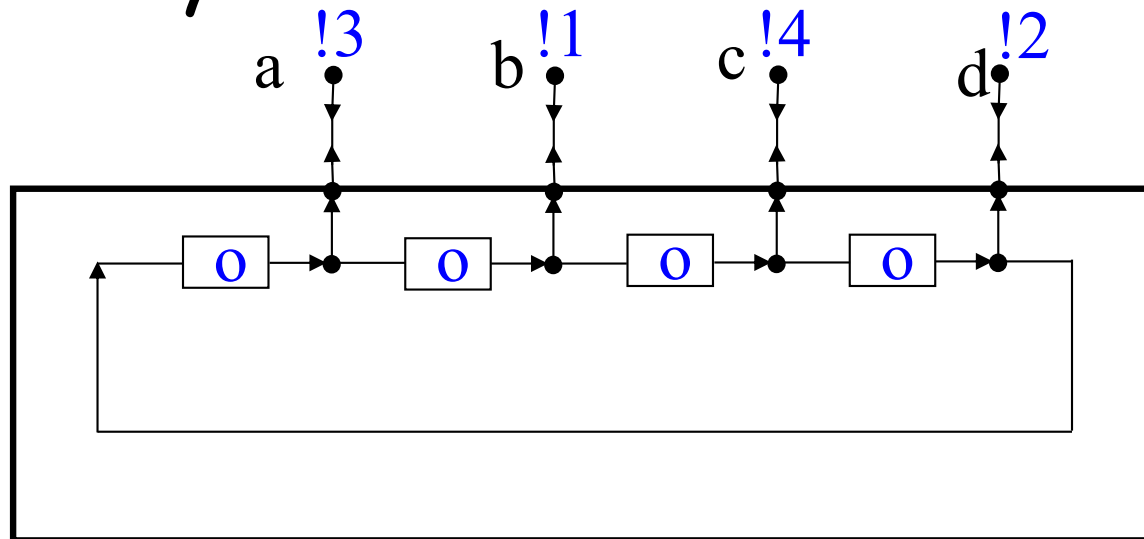
```
while (true) {  
  sleep(4000);  
  get(input, displayText);  
  print(displayText);  
}
```

```
main() {  
  green(a) red(b) blue(c) alternator(a, b, c)  
}
```



# Sequencer

- Writes to a, b, c, and d will succeed cyclically and in that order.

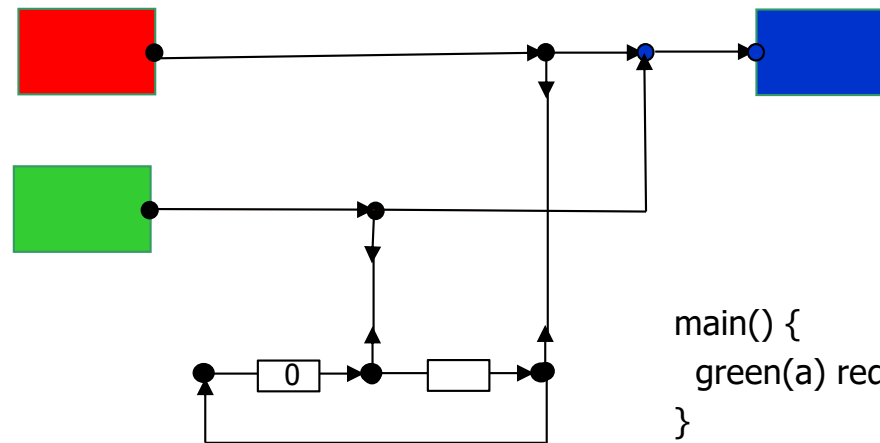


```
seqc(p[1..n]) {
  seqp(x[1..n]) for i = 1 .. n {syncdrain(x[i], p[i])}
}
```

```
seqp(p[1..n]) {
  for i = 1 .. n {if i = 1 {fifofull<0>(x[i], x[i+1])}
  else {fifo(x[i], x[i+1])}
  sync(x[i+1], p[i])}
  sync(x[n+1], x[1])
}
```

# Sequenced blocking producers

- A two-port sequencer and a few channels form the connector we need to compose and exogenously coordinate the green/red producers/consumer system.

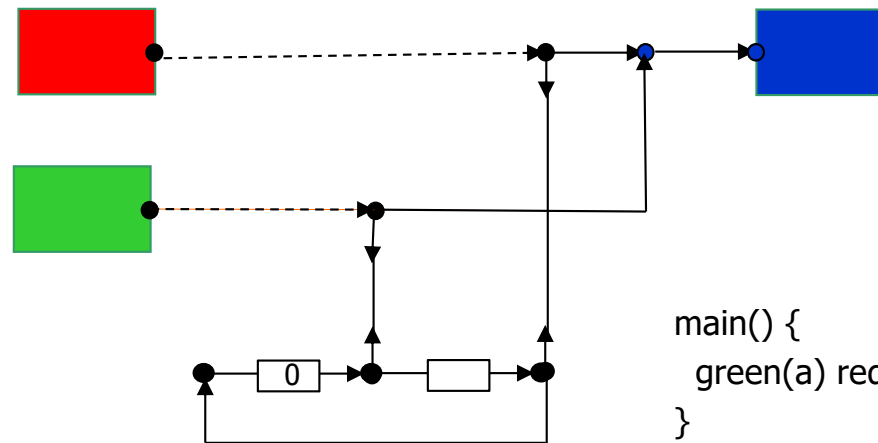


```
connector(a, b, c) {  
  seqc(x, y) sync(a, x) sync(b, y)  
  sync(m, c) sync(x, m) sync(y, m)  
}
```

```
main() {  
  green(a) red(b) blue(c) connector(a, b, c)  
}
```

# Sequenced non-blocking producers

- A two-port sequencer and a few channels form the connector we need to compose and exogenously coordinate the green/red producers/consumer system.

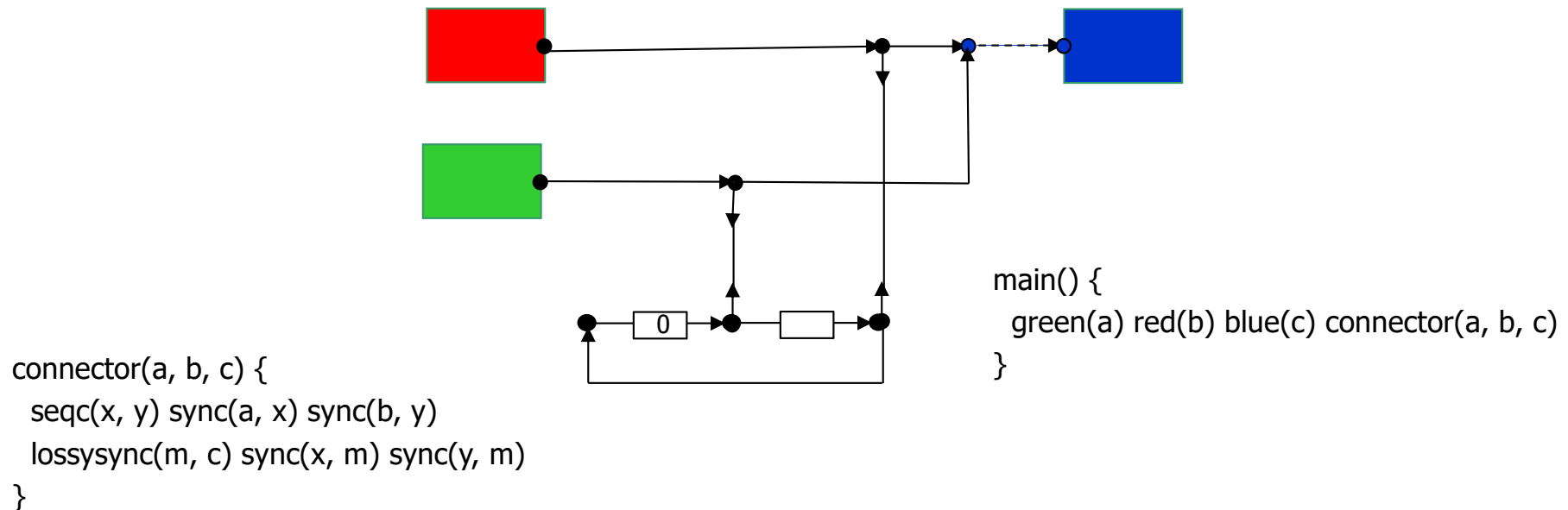


```
connector(a, b, c) {  
  seqc(x, y) lossysync(a, x) lossysync(b, y)  
  sync(m, c) sync(x, m) sync(y, m)  
}
```

```
main() {  
  green(a) red(b) blue(c) connector(a, b, c)  
}
```

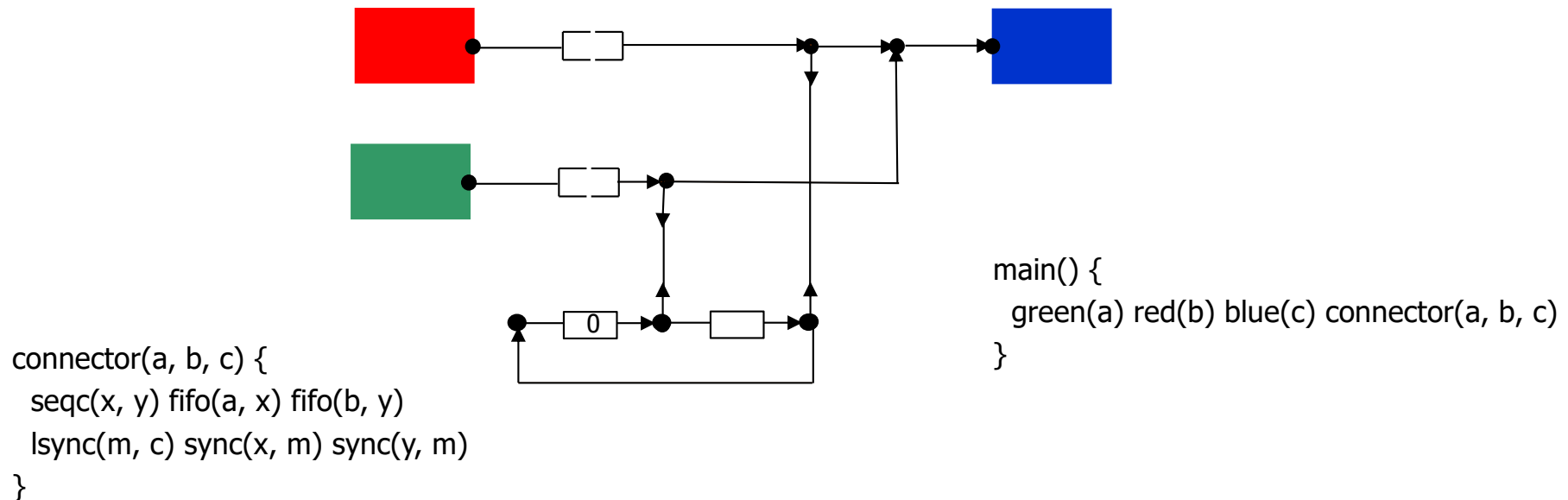
# Sequenced non-blocking producers

- What is the difference, if any, with the previous circuit?



# Buffered Producers

- Adding  $k > 0$  FIFO1 channels to the sequencer solution, buffers the actions of the producers and the consumer.



# Overflow Lossy FIFO1

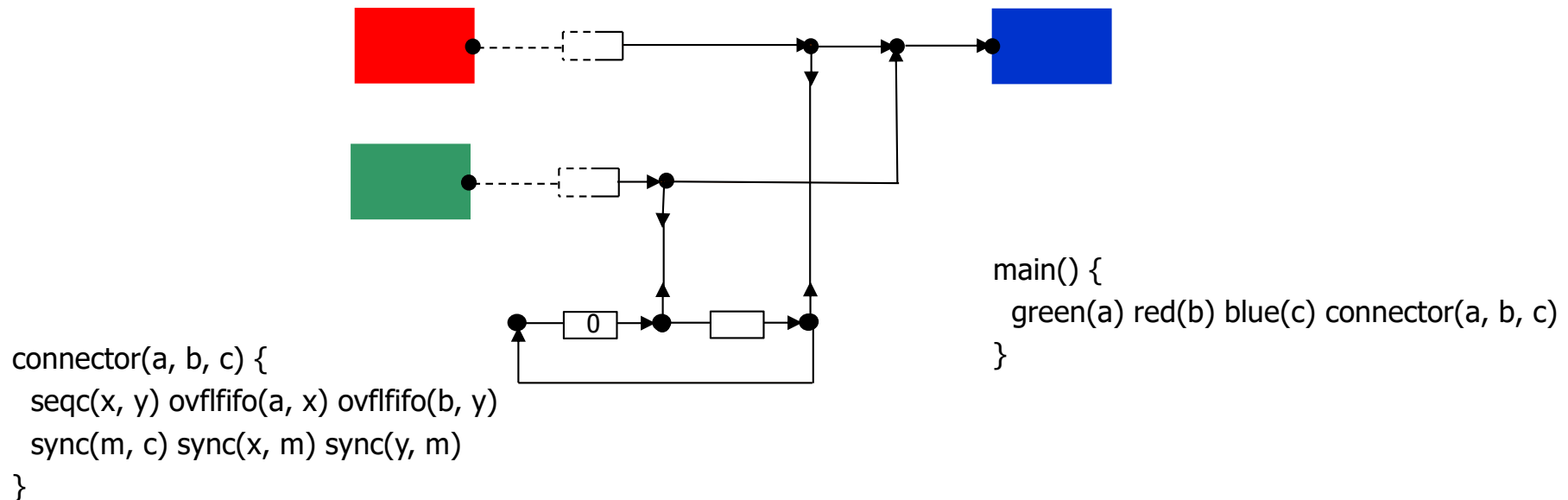
- A FIFO1 channel that accepts but loses new incoming values if its buffer is full.



```
ovlfifo(a, b) {  
  lossysync(a, m) fifo(m, b)  
}
```

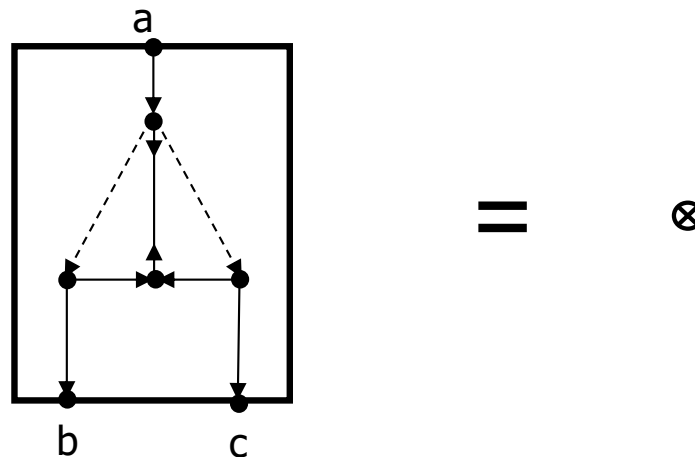
# Sampled Producers

- Adding Overflow-Lossy FIFO1 channels to the sequencer solution, buffers the actions of the producers and the consumer.



# Exclusive Router

- A value written to  $a$  flows through to either  $b$  or  $c$ , but never to both.

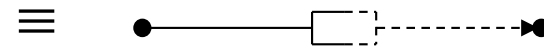
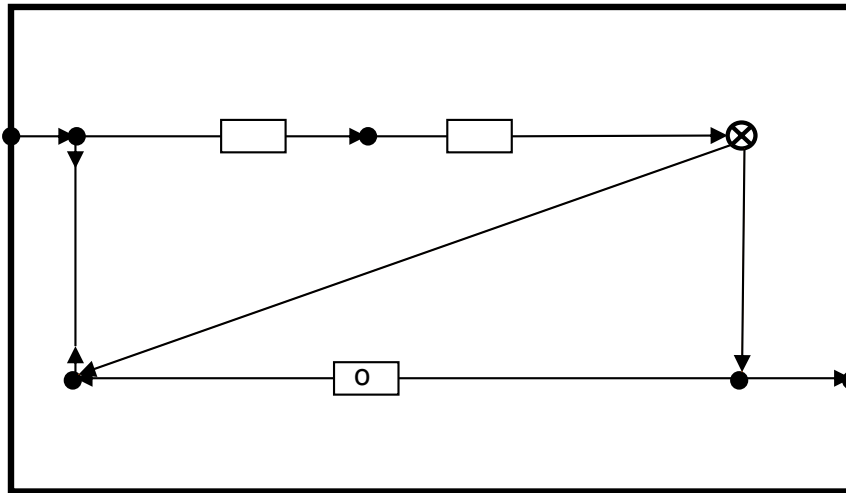


```
xrouter(in, out[1..n]) {  
  sync(in, s) syncdrain(s, m)  
  for i = 1 .. n {lossysync(s, x[i]) sync(x[i], m) sync(x[i], out[i])}  
}
```



# Shift Lossy FIFO

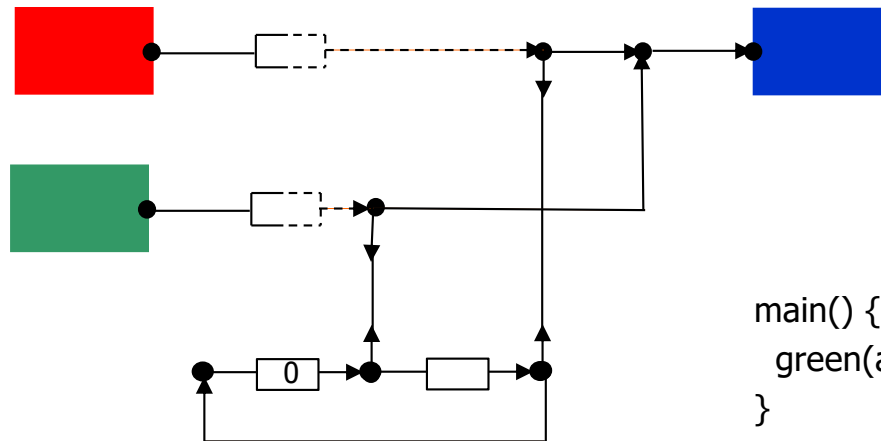
- A FIFO1 channel that loses its old buffer contents, if necessary, to make room for new incoming values.



```
shiftlossyfifo(in, out) {  
  sync(in, a) fifo(a, b) fifo(b, c) xrouter(c, d, e)  
  syncdrain(a, g) sync(d, f) sync(e, g) sync(f, out) fifofull<0>(f, g)  
}
```

# Sampled Producers

- Adding  $k > 0$  Shift-Lossy FIFO1 channels to the sequencer solution, buffers the actions of the producers and the consumer.

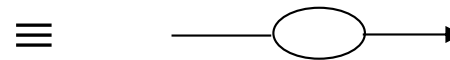
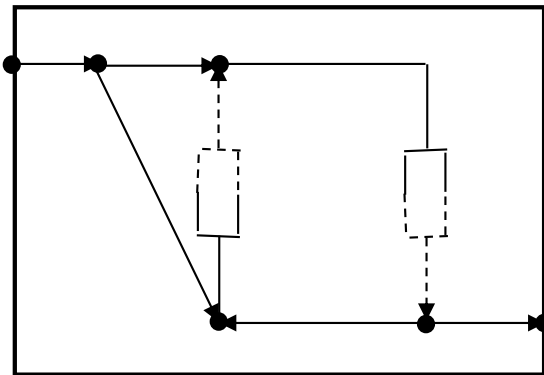


```
connector(a, b, c) {  
  seqc(x, y) shiftlossyfifo(a, x) shiftlossyfifo(b, y)  
  sync(m, c) sync(x, m) sync(y, m)  
}
```

```
main() {  
  green(a) red(b) blue(c) connector(a, b, c)  
}
```

# Variable

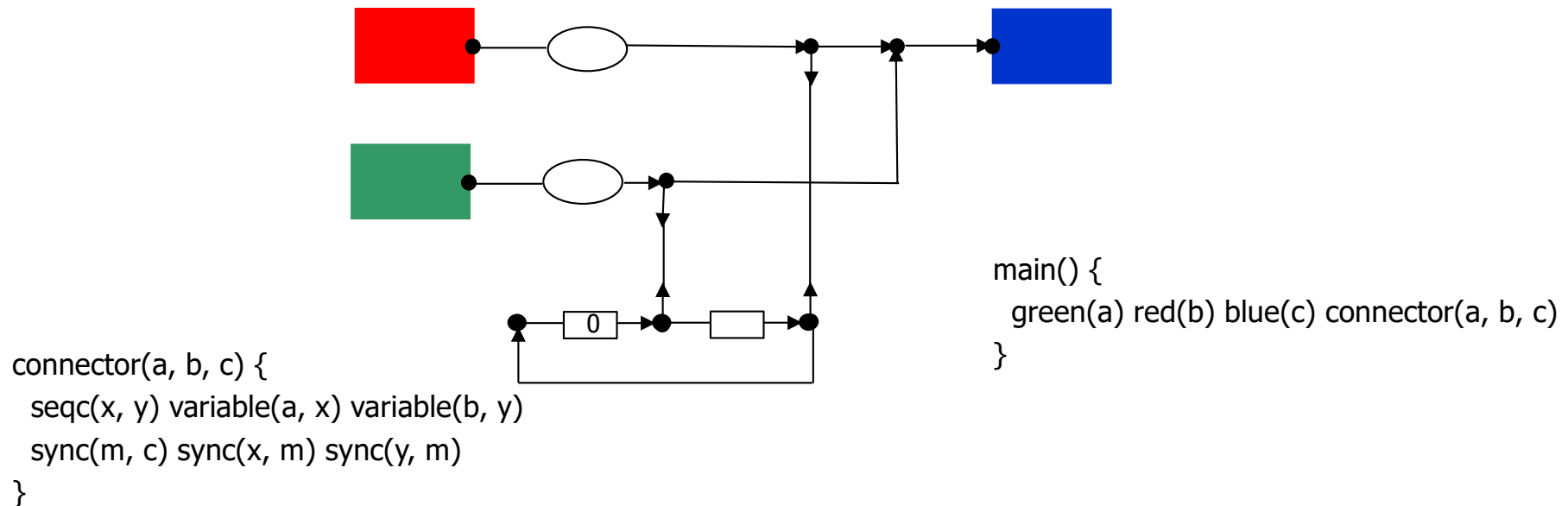
- Every input value is remembered and repeatedly reproduced as output, zero or more times, until it is replaced by the next input value.



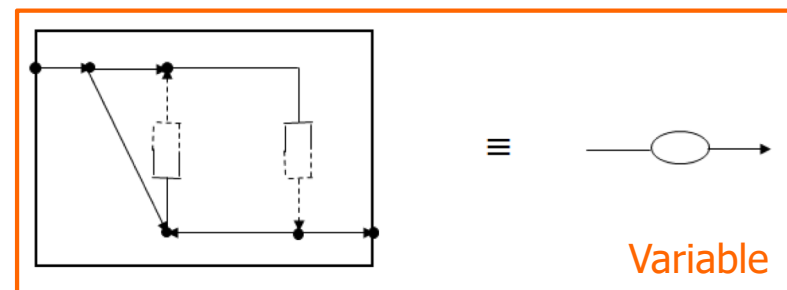
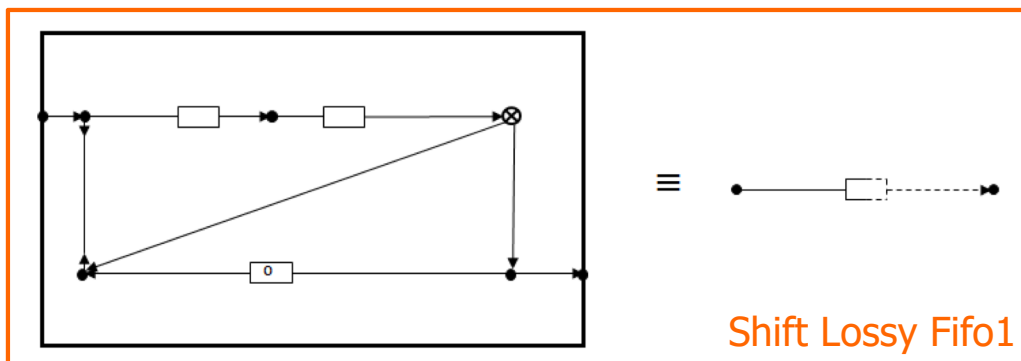
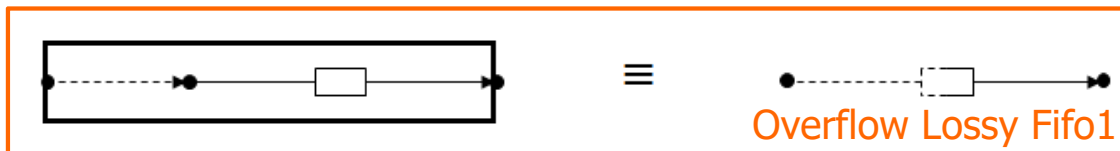
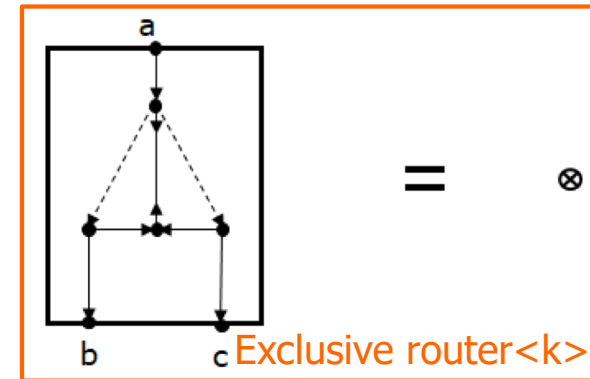
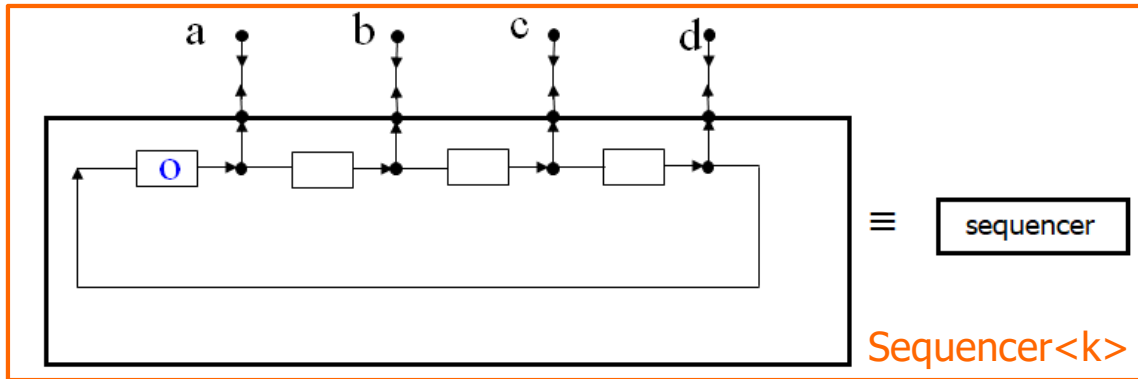
```
variable(a, b) {  
  sync(a, x) sync(x, y) shiftlossyfifo(y, z)  
  sync(z, b) sync(z, t) shiftlossyfifo(t, y)  
  sync(x, t)  
}
```

# Buffered Producers

- Adding variables to the sequencer solution, buffers the actions of the producers and the consumer.



# Library



# Java-like Implementation

## □ Shared entities

```
private final Semaphore greenSemaphore = new Semaphore(1);
private final Semaphore redSemaphore = new Semaphore(0);
private final Semaphore bufferSemaphore = new Semaphore(1);
private String buffer = EMPTY;
```

## □ Consumer

```
while (true) {
    sleep (4000);
    bufferSemaphore.acquire();
    if (buffer != EMPTY) {
        println(buffer);
        buffer = EMPTY;
    }
    bufferSemaphore.release();
}
```

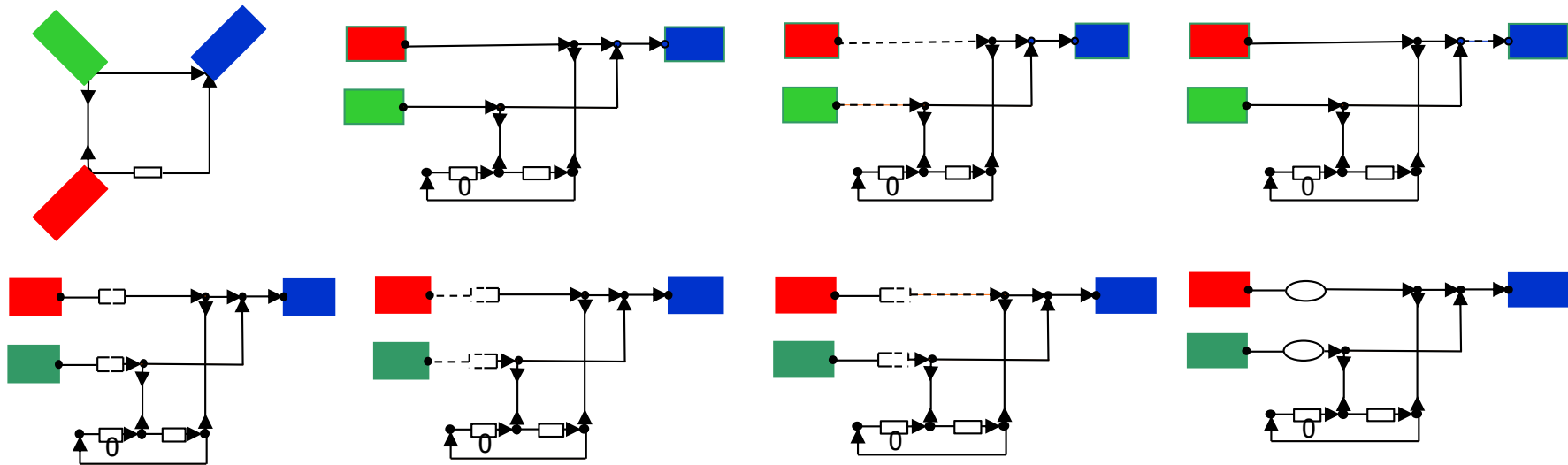
## □ Producers

```
while (true) {
    sleep (5000);
    greenText = ...
    greenSemaphore.acquire();
    bufferSemaphore.acquire();
    buffer = greenText;
    bufferSemaphore.release();
    redSemaphore.release();
}
```

```
while (true) {
    sleep (3000);
    redText = ...
    redSemaphore.acquire();
    bufferSemaphore.acquire();
    buffer = redText;
    bufferSemaphore.release();
    greenSemaphore.release();
}
```

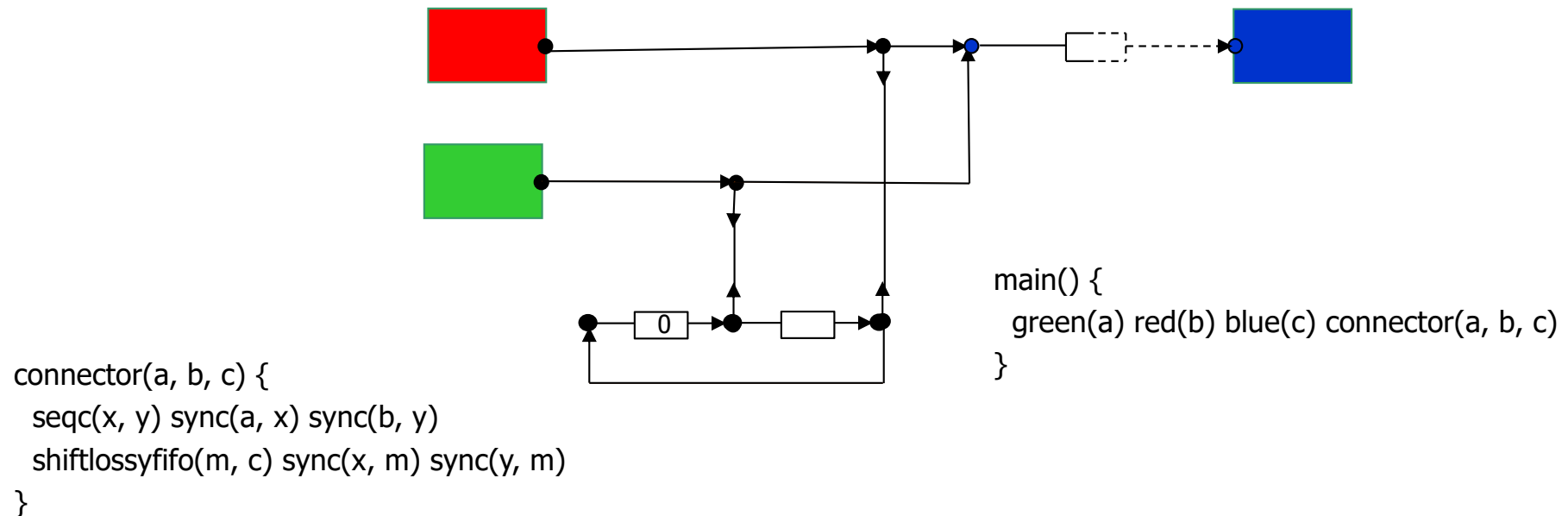
# Where is Waldo?

□ Which one of the protocols does the Java-like code actually implement?



# Over-writing Producers

- The protocol in the Java-like implementation corresponds to the following Reo circuit:

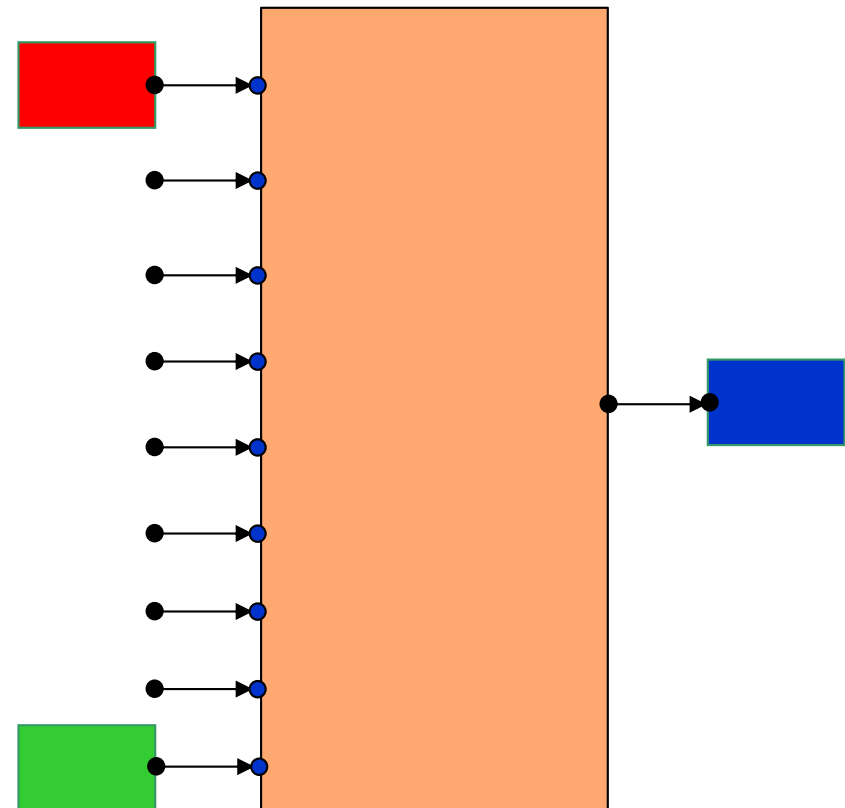




# Scaling up

## □ Scale up?

```
main() {  
  green(a[1]) ... red(a[n]) blue(b)  
  connector(a[1..n], b)  
}  
  
connector(a[1..n], b) {  
  seqc(x[n])  
  for i = 1 ..n {sync(a[i], x[i]) sync(x[i], m)}  
  sync(m, b)  
}
```

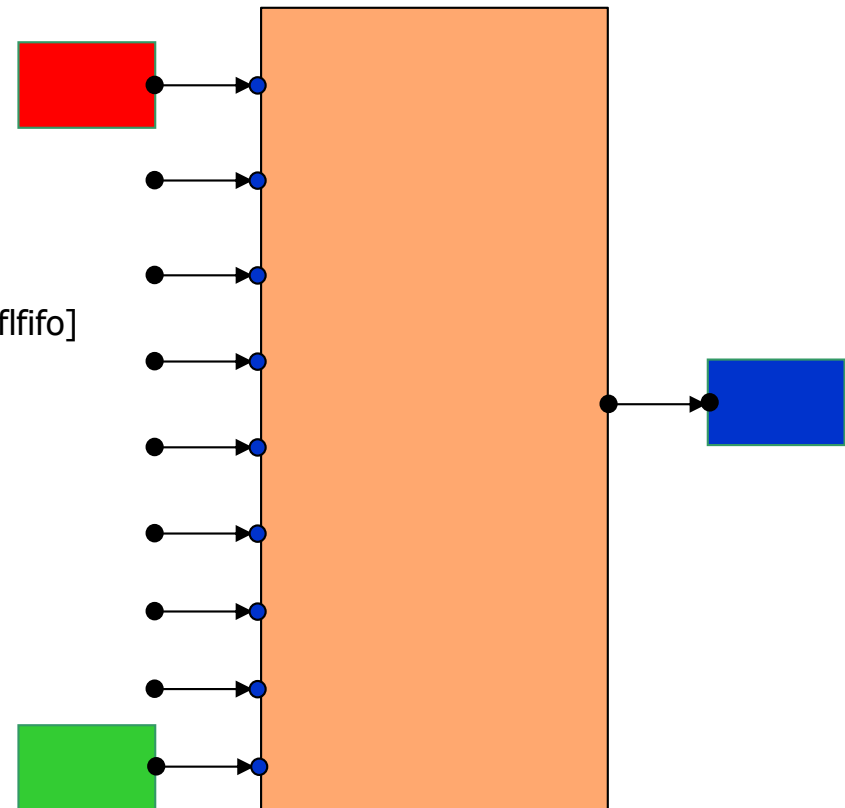


# Scale and combine

## □ Mix and match?

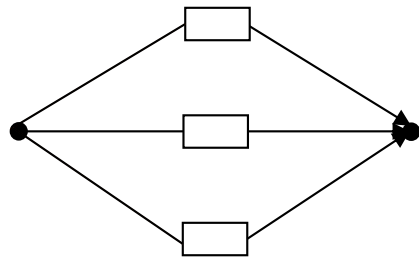
```
main() {  
  green(a[1] ... red(a[n]) blue(b)  
  ileg = [sync, lossysync, fifo, sync, variable, ..., shiftlossyfifo, ovlfifo]  
  connector<ileg[1..n], sync>(a[1..n], b)  
}
```

```
Connector<ileg[1..n](?, !), oleg(?, !)> (a[1..n], b) {  
  seqc(x[n])  
  for i = 1 ..n {ileg[i](a[i], x[i]) sync(x[i], m)}  
  oleg(m, b)  
}
```



# A $k$ -repeater

- A single fifo channel produces its input once as its output.
- Placing  $k$  fifo channels in parallel between a source and a sink node produces  $k$  copies of its input as its output.

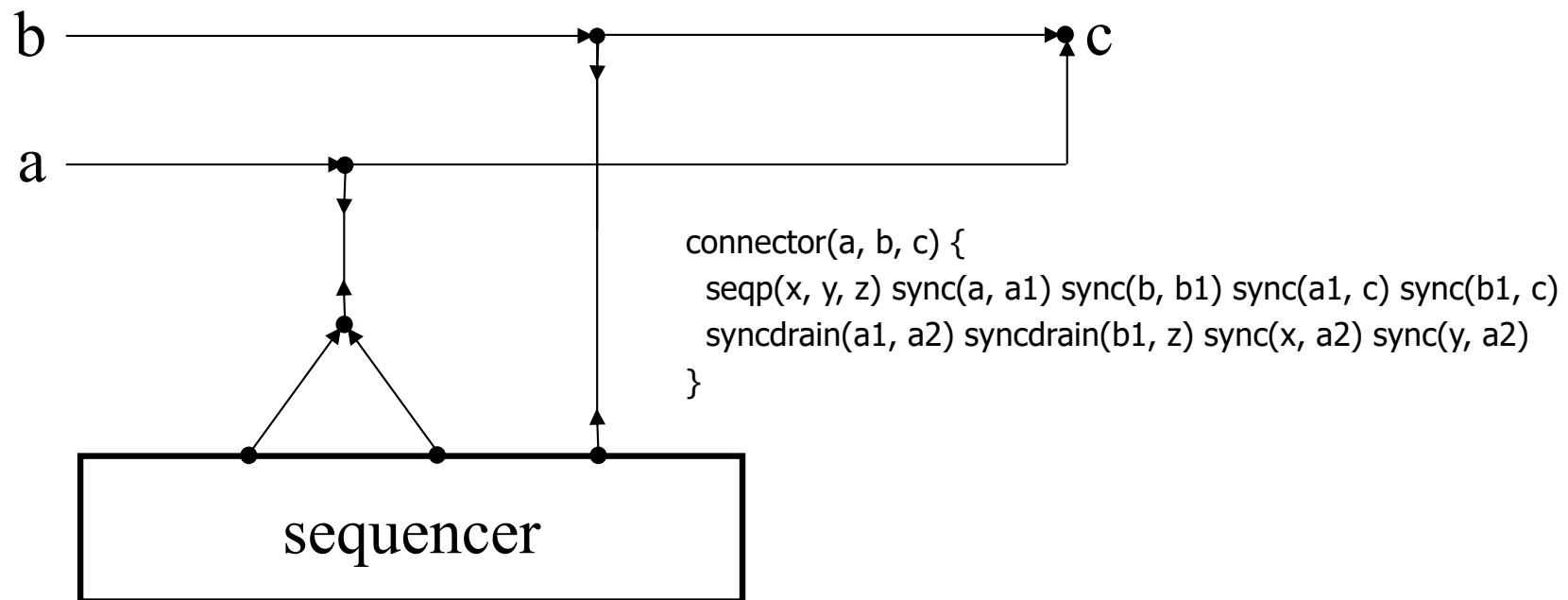


```
main(x?, y!) {  
  repeater<3>(x, y)  
}
```

```
repeater(k)(a, b) {  
  fifo(a, b).k {fifo(a, b)}  
}
```

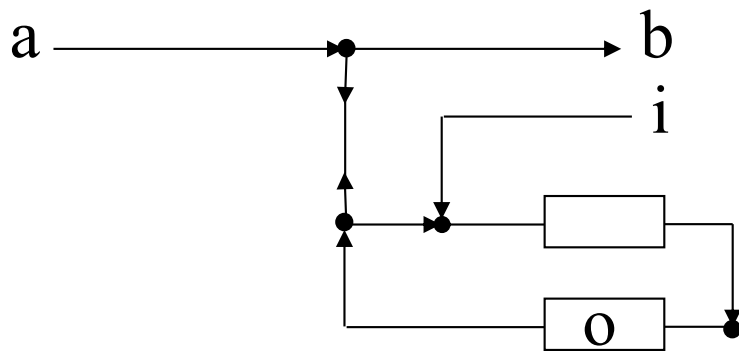
# Two for one

- ❑ Two source nodes, a and b, and a sink node.
- ❑ Output on c two from a and one from b.



# Inhibitor

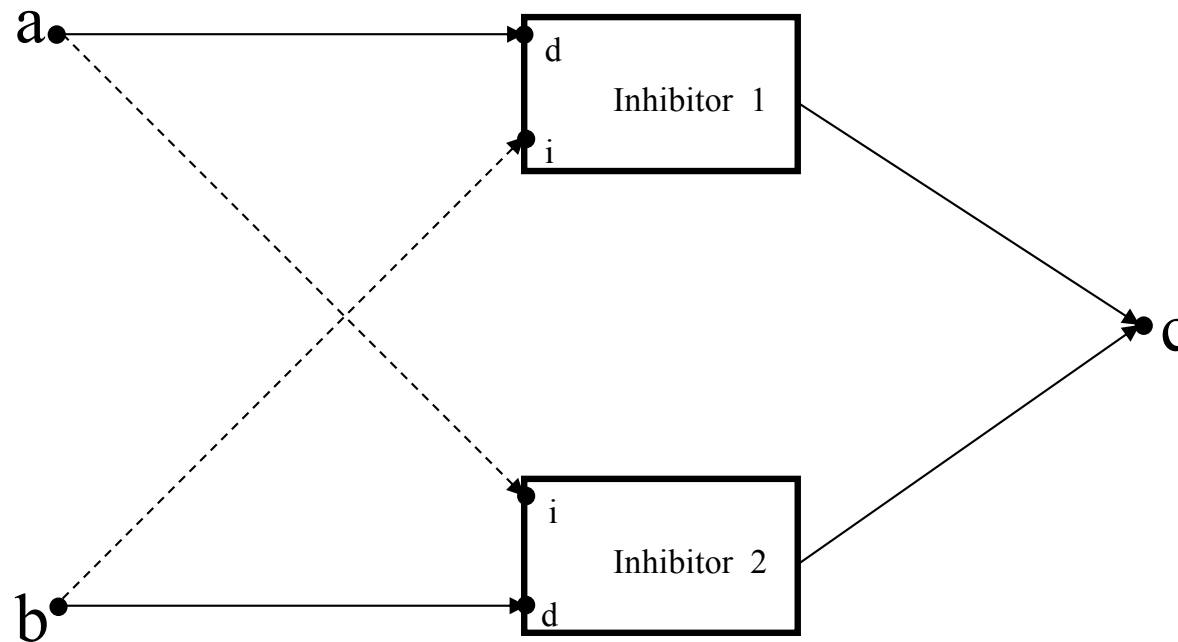
- All values flow from  $a$  to  $b$  until a value is written to  $i$ .
- A write to  $i$  *inhibits* (i.e., blocks) further writes to both  $d$  and  $i$ .



```
inhibitor(a, b, i) {  
  sync(a, c) sync(c, b) syncdrain(c, d)  
  sync(d, e) fifo(e, f) fifofull<0>(f, d)  
}
```

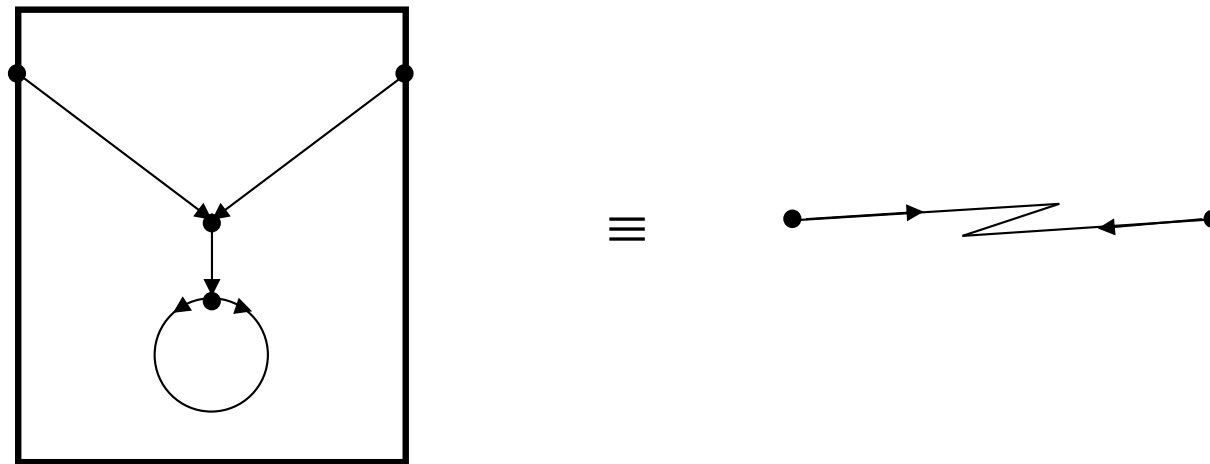
$$C = a^* \mid b^*$$

- The drain is asynchronous; dashed arrows show synchronous lossy channels; all other channels are synchronous.



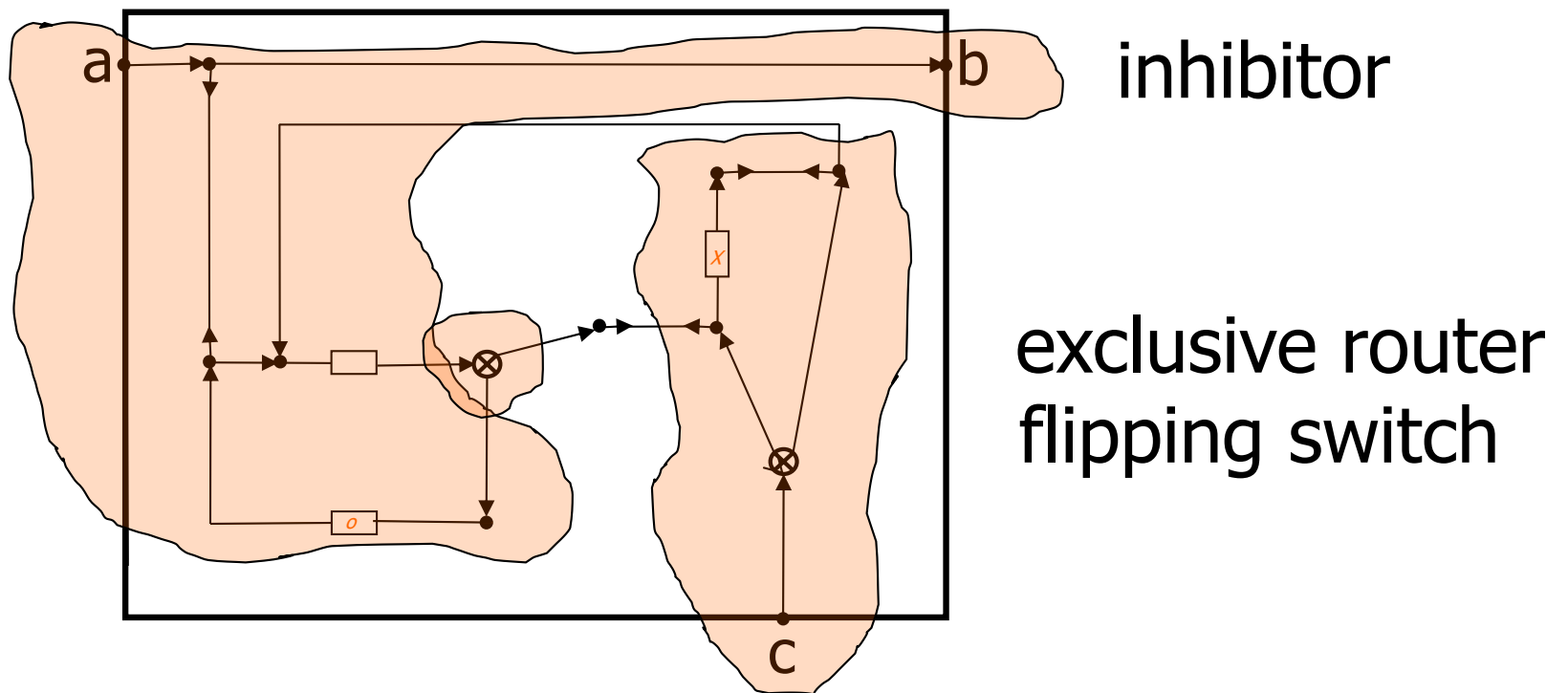
# Asynchronous Drain

- An AsyncDrain can be composed out of a SyncDrain and 3 (or 2) Sync channels.



# Valve (open)

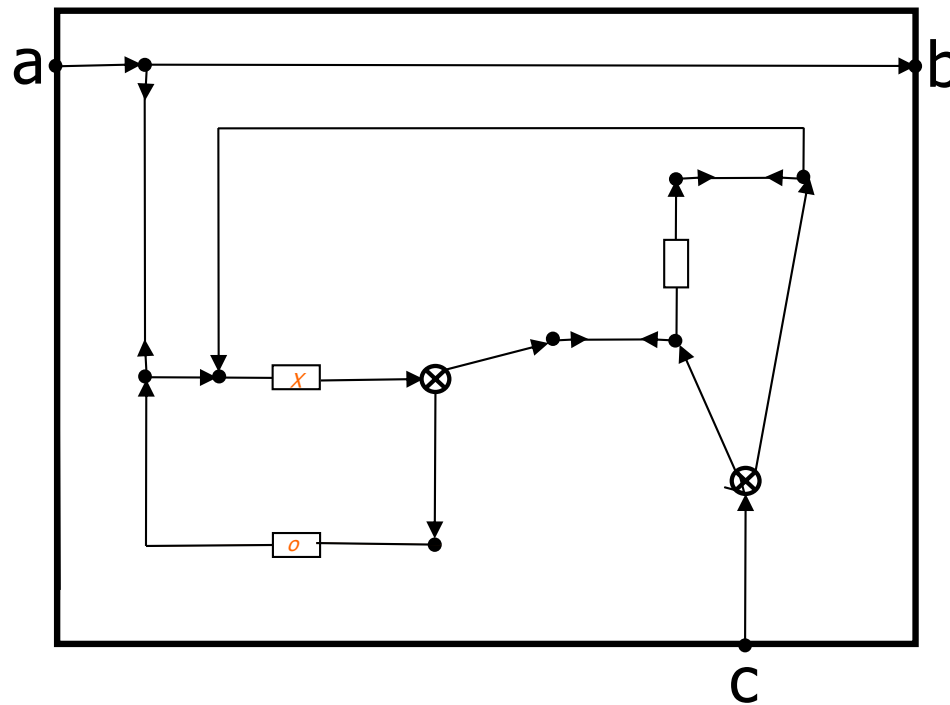
- A write to *c* closes the flow of data from *a* to *b*.



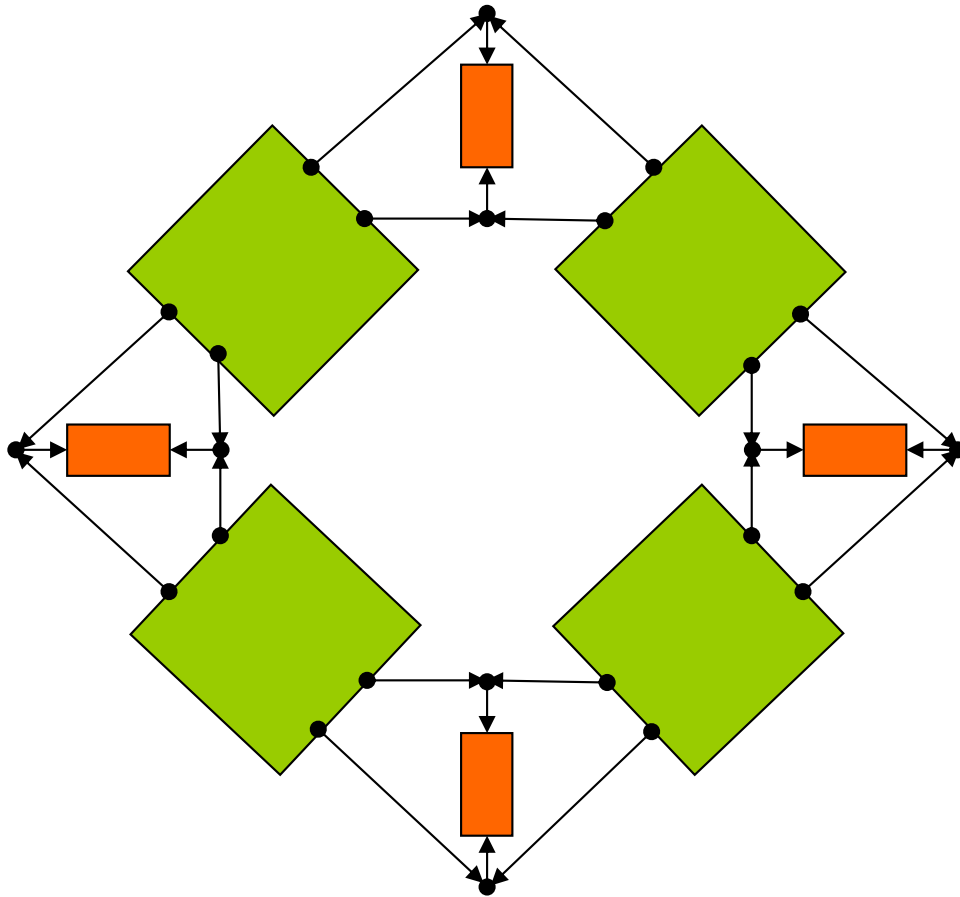


# Valve (closed)

- A write to *c* opens the flow of data from *a* to *b*.

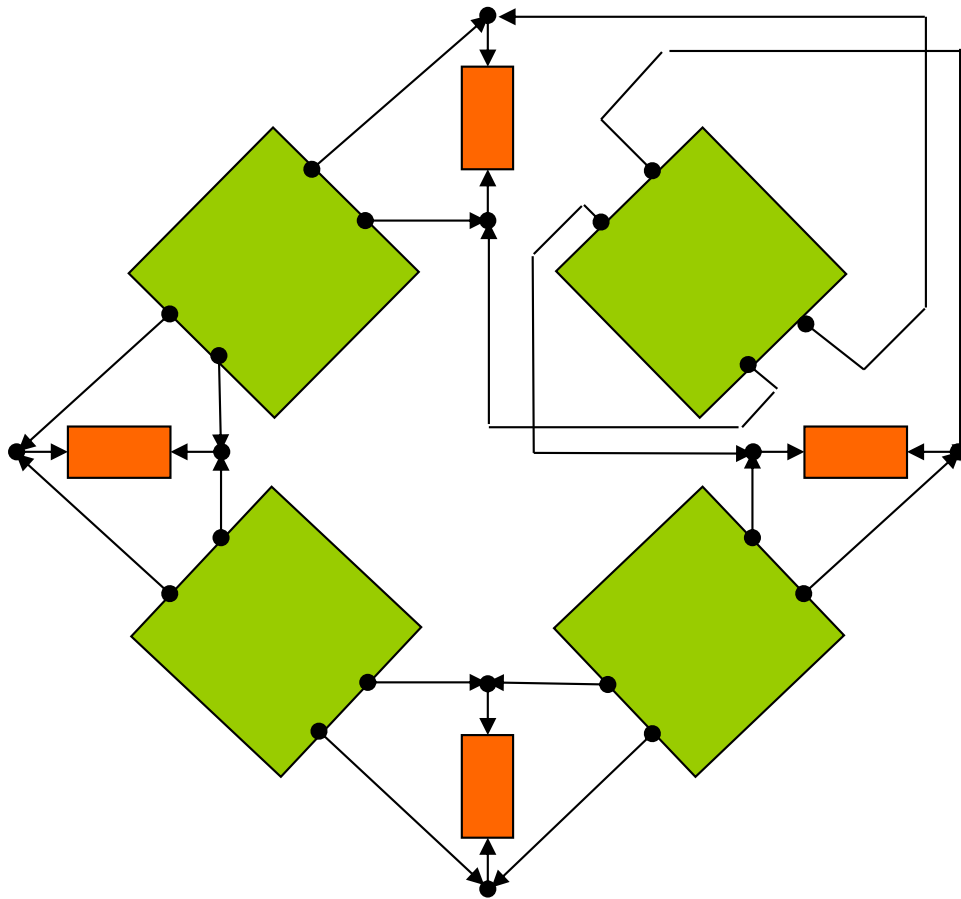


# Dining Philosophers (problem)



```
DinPhilProblem<n, phil(tr!, fr!, tl!, fl!), fork(t?, f?)>() {  
  for i = 1..n {  
    phil(tr[i], fr[i], tl[i], fl[i]) fork(t[i], f[i])  
    sync(tr[i], t[i]) sync(fr[i], f[i])  
    sync(tl[(i+1)%n], t[i]) sync(fl[(i+1)%n], f[i])  
  }  
}
```

# Dining Philosophers (solution)



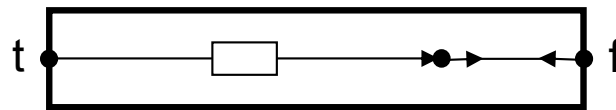
```

DinPhilSolution<n, phil(tr!, fr!, tl!, fl!), fork(t?, f?)>() {
  for i = 1..n {
    phil(tr[i], fr[i], tl[i], fl[i]) fork(t[i], f[i])
    if i == n {
      sync(tr[i], t[1]) sync(fr[i], f[1])
      sync(tl[n], t[i]) sync(fl[n], f[i])
    }
    else {
      sync(tr[i], t[i]) sync(fr[i], f[i])
      sync(tl[(i+1)%n], t[i]) sync(fl[(i+1)%n], f[i])
    }
  }
}

```

# Fork

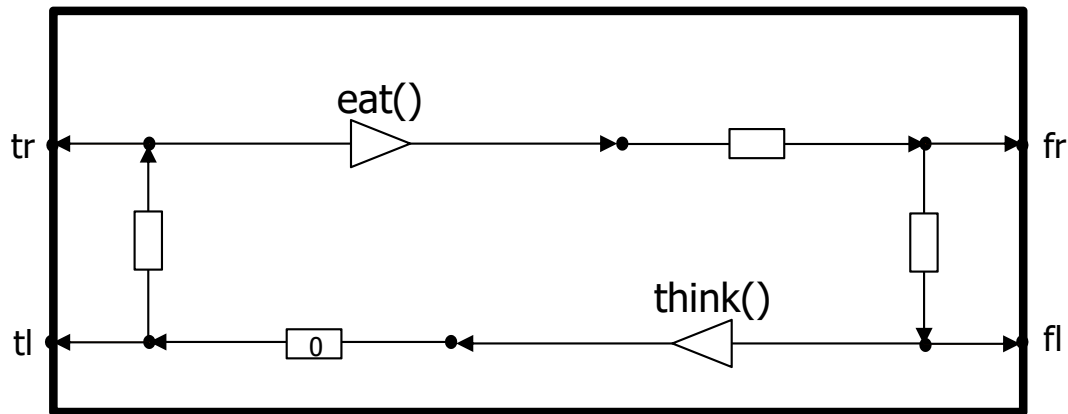
- The fork component used in the dining philosophers problem is a pure coordinator and can be constructed as a Reo connector circuit.



`fork(t?, f?) { fifo(t, z) syncdrain(z, f) }`

# Philosopher

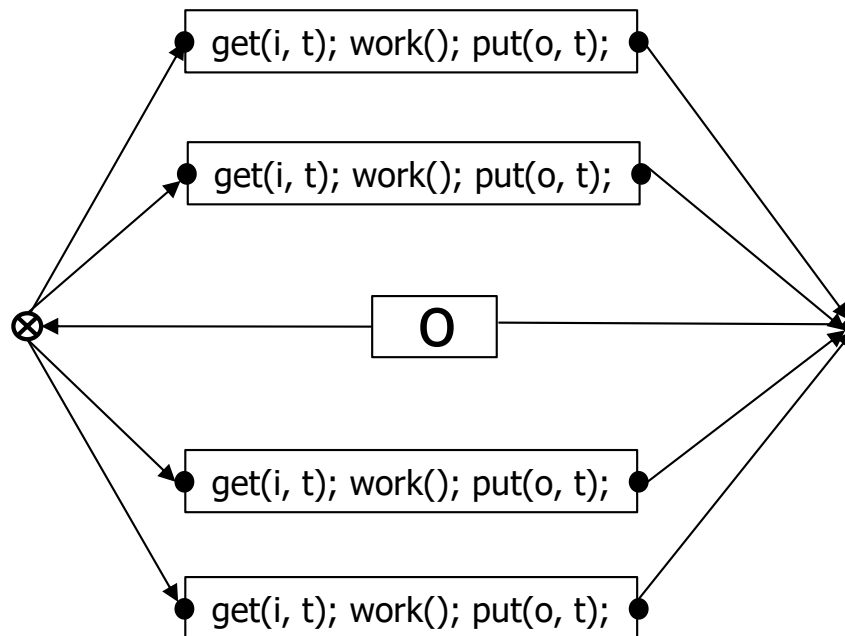
- Internal coordination of *think()* and *eat()* functions in a Philosopher.



```
Philosopher<think:(any:any), eat:(any:any)>(tr!, fr!, tl!, fl!) {  
  sync(a, tr) transformer<eat>(a, b) fifo(b, c) sync(c, fr) fifo(c, d)  
  sync(d, fl) transformer<think>(d, e) fifofull<0>(e, f) sync(f, tl) fifo(f, a)  
}
```

# Simple mutual exclusion (get-put)

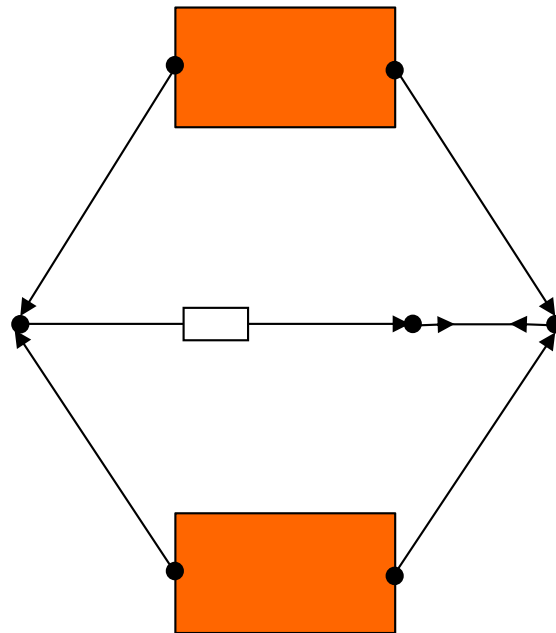
- Place a circuit to establish mutual exclusion between the following two components.



```
mutexgp(a[1..n]!, b[1..n]?) {  
  xrouter(y, a[1..n]) fifofull<0>(x, y)  
  for i = 1..n {sync(b[i], x) }  
}
```

# Simple mutual exclusion (put-put)

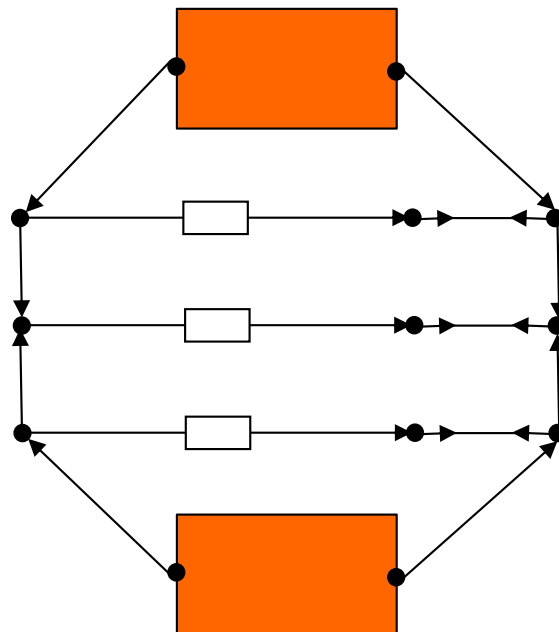
- Components are supposed to put a token on one port, announcing the start of their critical section, and put a token on another when they end.



```
mutexpp(a[1..n]?, b[1..n]?) {  
  fifo(x, z) syncdrain(z, y)  
  for i = 1..n {sync(a[i], x) sync(b[i], y)}  
}
```

# Fool-proof mutual exclusion

- ❑ Components are supposed to put a token on one port, announcing the start of their critical section, and put a token on another when they end.
- ❑ Components cannot be fully trusted to abide by this convention!



```
mutex(a[1..n]?, b[1..n]?) {  
  guard(x, y){fifo(x, z) syncdrain(z, y)}  
  guard(p, q)  
  for i = 1..n {guard(x[i], y[i])  
    sync(a[i], x[i]) sync(b[i], y[i])  
    sync(x[i], p) sync(y[i], q)  
  }  
}
```



# Concurrency in Reo

- Reo embodies a non-conventional model of concurrency:

- **Conventional**

- action based
- process as primitive
- imperative
- functional
- imperative programming
- protocol implicit in processes

- **Reo**

- interaction based
- Protocol as primitive
- declarative
- relational
- constraint programming
- Tangible explicit protocols

- Reo is more expressive than Petri nets, workflow, dataflow, Kahn networks, synchronous languages, and stream processing languages.

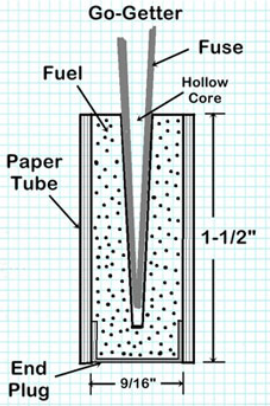
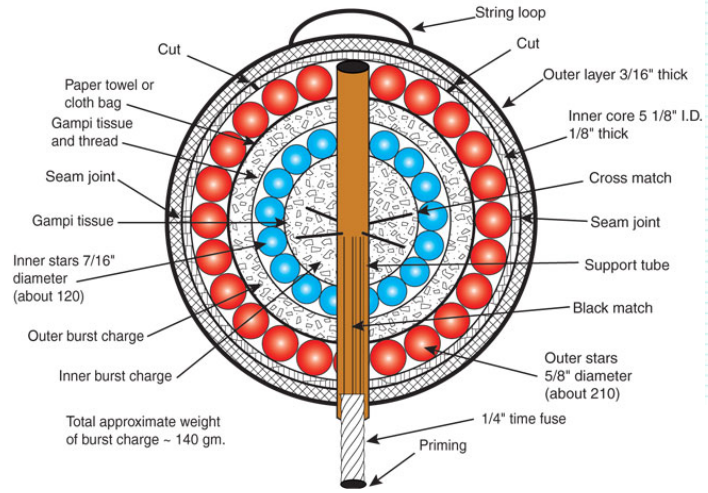
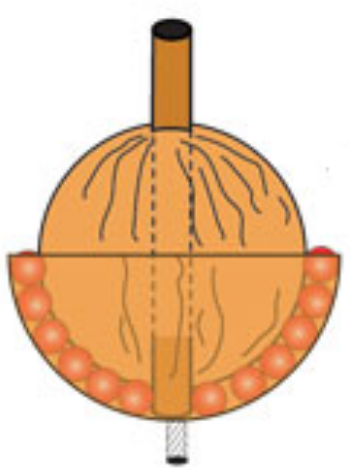
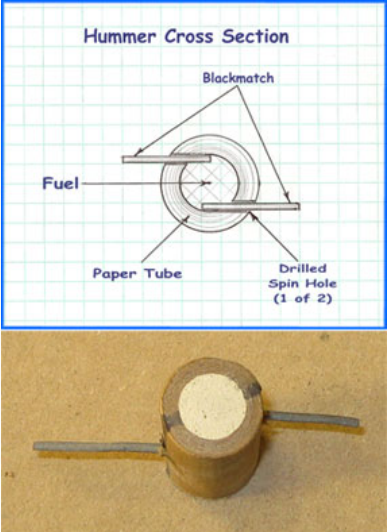
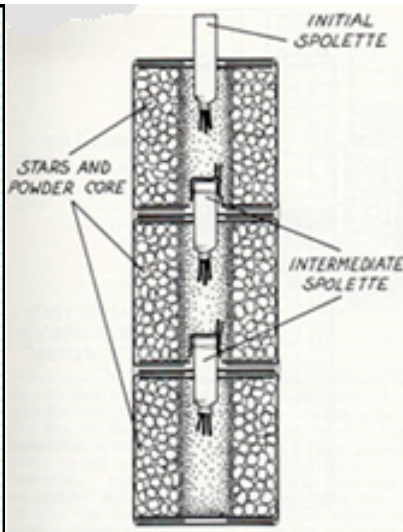
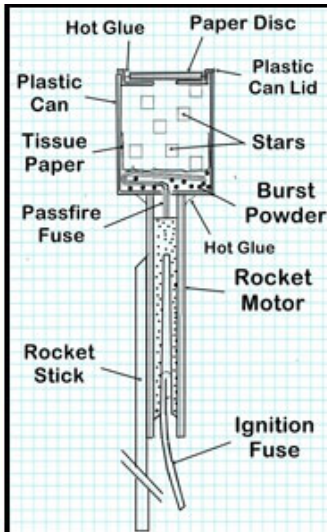
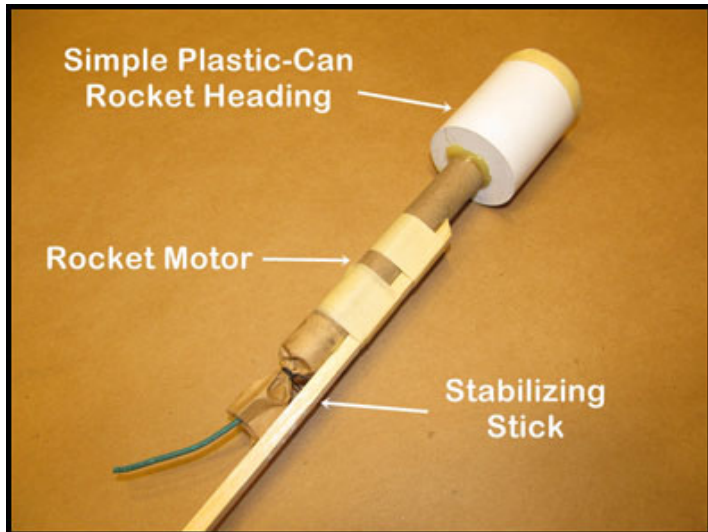
# Semantics



- Reo allows:
  - Arbitrary user-defined channels as primitives.
  - Arbitrary mix of synchrony and asynchrony.
  - Relational constraints between input and output.
- Reo is more expressive than, e.g., dataflow models, Kahn networks, workflow models, stream processing models, Petri nets, and synchronous languages.
- Formal semantics:
  - Coalgebraic semantics based on timed-data streams.
  - Constraint automata.
  - SOS semantics (in Maude).
  - Constraint propagation (connector coloring scheme).
  - Intuitionistic linear logic

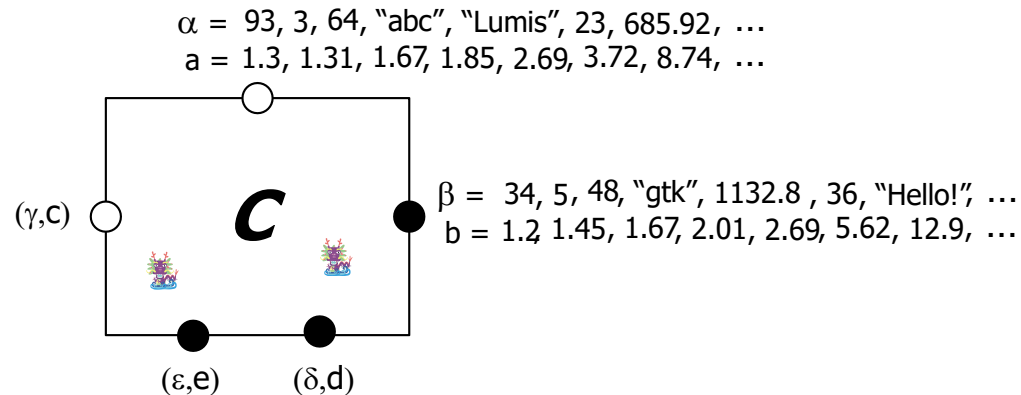
• Sung-Shik T.Q. Jongmans and Farhad Arbab, "Overview of Thirty Semantic Formalisms for Reo," *Scientific Annals of Computer Science*, vol. 12, Issue 1, pp. 201-251, 2012.

# Candipresitt iconas transition action



Ingredients

# Component behavior



**Timed-data-streams (TDS):**  $(\alpha, a), (\beta, b), (\gamma, c), (\delta, d), (\epsilon, e)$

**Abstract Behavior Type (ABT):** Relation over TDSs:

$$C = ((\alpha, a), (\gamma, c); (\beta, b), (\delta, d), (\epsilon, e))$$

- F. Arbab "**Abstract Behavior Types: A foundation model for components and their composition**," International Symposium on Formal Methods for Components and Objects, (FMCO 2002), November 5-8, 2002, Leiden, The Netherlands, F. S. de Boer and M. M. Bonsangue and S. Graf and W.-P. de Roever (eds.), LNCS 2852, pp. 33-70, September 2003.
- F. Arbab and J.J.M.M. Rutten, "**A coinductive calculus of component connectors**," post Proc. of the 16th International Workshop on Algebraic Development Techniques (WADT 2002), M. Wirsing, D. Pattinson and R. Hennicker (eds.), LNCS 2755, pp. 35-56, 2003.
- J.J.M.M. Rutten, "**Component Connectors**," In Prakash Panangaden and Franck van Breugel, editors, Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems, volume 23 of CRM, pages 73-87, AMS, 2004.

# Timed-Data-Streams

□ A *timed-data-stream* is a twin pair of infinite streams,  $\langle \alpha, a \rangle$ , where :

○ Data stream  $\alpha$

- Elements of  $\alpha$  are uninterpreted data items

○ Time stream  $a$

- Elements of  $a$  are non-negative real numbers
- Time elapses incrementally:  $\forall i \geq 0, a(i) < a(i + 1)$
- Finite steps in any interval:  $\forall N, \exists i: a(i) > N$

○ Data item  $\alpha(i)$  is observed at time  $a(i)$ .

□ Based on *Stream Calculus* by Jan Rutten

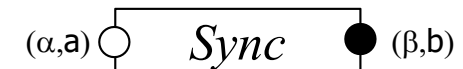
- F Arbab and JJMM Rutten, "A coinductive calculus of component connectors," Recent Trends in Algebraic Development Techniques, LNCS 2755, pp. 34-55, 2003.
- JJMM Rutten, "A coinductive calculus of streams," Mathematical Structures in Computer Science 15 (01), 93-147, 2005.
- JJMM Rutten, "Behavioural differential equations: a coinductive calculus of streams, automata, and power series," Theoretical Computer Science 308 (1), 1-53, 2003.



# Component examples

□ Synchronously passes its input as its output:

○  $\text{Sync}(\langle \alpha, a \rangle; \langle \beta, b \rangle) \equiv \alpha = \beta, a = b$



□ An infinite FIFO:

○  $\text{FIFO}(\langle \alpha, a \rangle; \langle \beta, b \rangle) \equiv \alpha = \beta, a < b$



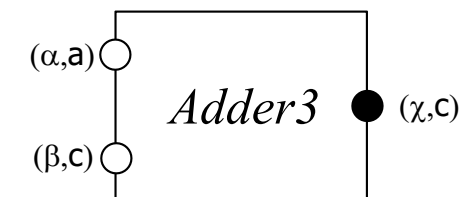
□ A FIFO1:

○  $\text{FIFO1}(\langle \alpha, a \rangle; \langle \beta, b \rangle) \equiv \alpha = \beta, a < b < a'$



□ An adder:

$$\begin{aligned} \text{Adder3}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv \\ \gamma(0) = \alpha(0) + \beta(0) \wedge \\ a(0) < b(0) < c(0) < a(1) \wedge \\ \text{Adder3}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). \end{aligned}$$



# Channels: binary components

- Synchronously passes its input as its output:

- $\text{Sync}(\langle \alpha, a \rangle; \langle \beta, b \rangle) \equiv \alpha = \beta, a = b$



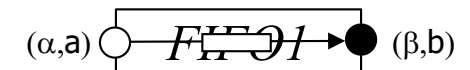
- An infinite FIFO:

- $\text{FIFO}(\langle \alpha, a \rangle; \langle \beta, b \rangle) \equiv \alpha = \beta, a < b$



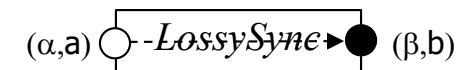
- A FIFO1:

- $\text{FIFO1}(\langle \alpha, a \rangle; \langle \beta, b \rangle) \equiv \alpha = \beta, a < b < a'$



- A lossy synchronous channel:

$$\text{LossySync}(\langle \alpha, a \rangle; \langle \beta, b \rangle) \equiv \begin{cases} \text{LossySync}(\langle \alpha', a' \rangle; \langle \beta, b \rangle) & \text{if } a(0) < b(0) \\ \alpha(0) = \beta(0), \text{LossySync}(\langle \alpha', a' \rangle; \langle \beta', b' \rangle) & \text{if } a(0) = b(0) \end{cases}$$



- A Synchronous drain:

- $\text{SyncDrain}(\langle \alpha, a \rangle, \langle \beta, b \rangle; ) \equiv a = b$



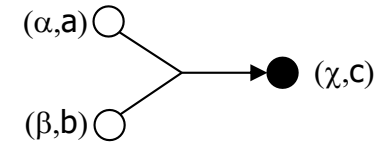
- A Synchronous spout:

- $\text{SyncSpout}(\langle \alpha, a \rangle, \langle \beta, b \rangle) \equiv a = b$



# Behavior of Reo Nodes

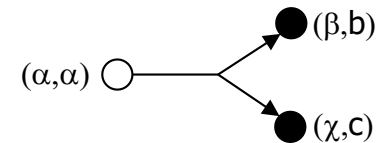
## □ Nondeterministic binary merge:



$$M(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv \begin{cases} \gamma(0) = \alpha(0), c(0) = a(0), M(\langle \alpha', a' \rangle, \langle \beta, b \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) < b(0) \\ \gamma(0) = \beta(0), c(0) = b(0), M(\langle \alpha, a \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) > b(0) \end{cases}$$

## □ Binary replicator:

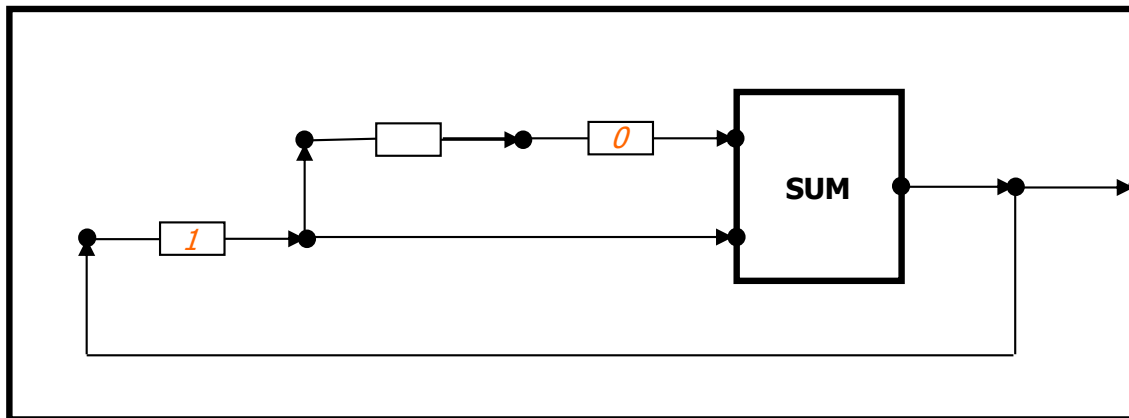
$$R(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \alpha = \beta = \gamma, a = b = c$$





# Fibonacci Series

- This circuit produces the Fibonacci series using an adder component.



```
Fibonacci<sum(a?,b?,c!)>(out) {  
  sync(c, d) sync(d, e) sync(d, out)  
  fifofull<1>(e, f) sync(f, b)  
  sync(f, g) fifo(g, h) fifofull<0>(h, a)  
}
```

- The timed-data-streams semantics allows us to prove its correctness.

# Some possible adders (1)

$$\begin{aligned} \text{Adder1}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\ \gamma(0) &= \alpha(0) + \beta(0) \wedge \\ \exists t : \max(a(0), b(0)) < t < \min(a(1), b(1)) &\wedge c(0) = t \wedge \\ \text{Adder1}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). & \end{aligned}$$

- Arbitrary input order; produces an output after each pair of input, some time before the next input.

$$\begin{aligned} \text{Adder2}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\ \gamma(0) &= \alpha(0) + \beta(0) \wedge \\ c(0) &= \max(a(0), b(0)) \wedge \\ \text{Adder2}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). & \end{aligned}$$

- Arbitrary input order; produces an output at the same time as the last of each input pair.

$$\begin{aligned} \text{Adder3}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\ \gamma(0) &= \alpha(0) + \beta(0) \wedge \\ a(0) < b(0) < c(0) < a(1) &\wedge \\ \text{Adder3}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). & \end{aligned}$$

- Ordered input; produces an output after each pair of input, some time before the next input.

# Some possible adders (2)

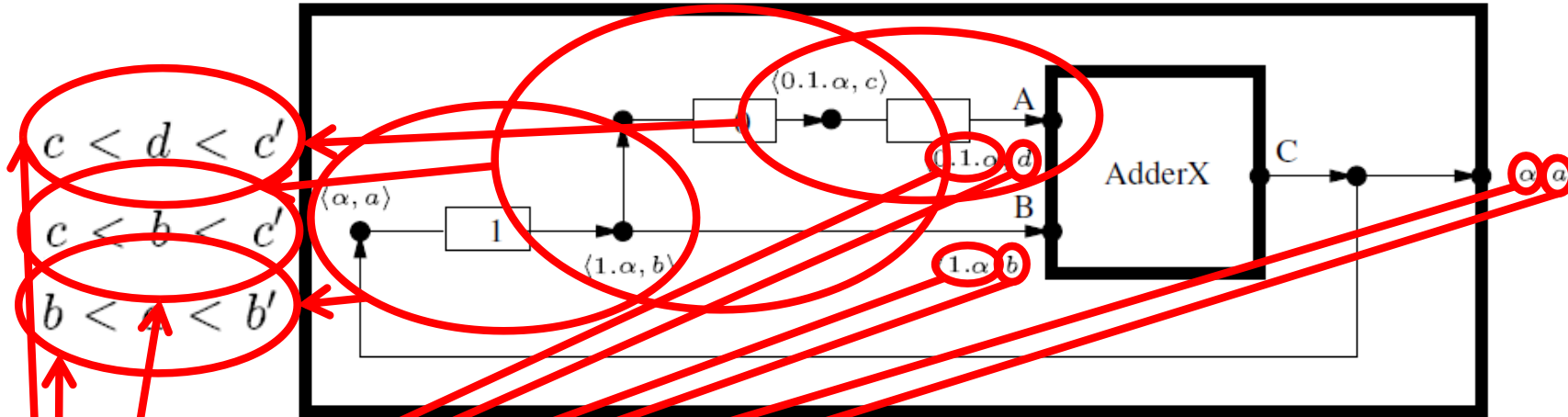
$$\begin{aligned} \text{Adder4}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\ \gamma(0) &= \alpha(0) + \beta(0) \wedge \\ a &= b = c \wedge \\ \text{Adder4}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). \end{aligned}$$

- Synchronous adder: reads a pair and outputs their sum all at the same time (atomically).

$$\begin{aligned} \text{Adder5}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\ \gamma(0) &= \alpha(0) + \beta(0) \wedge \\ c(0) &= \min(a(1), b(1)) \wedge \\ \text{Adder5}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). \end{aligned}$$

- Arbitrary input order; produces an output at the same time as the first of the next input pair.

# Fibonacci correctness proof (1)



$c < d < c'$   
 $c < b < c'$   
 $b < a < b'$

$Adder3(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv$   
 $\gamma(0) = \alpha(0) + \beta(0) \wedge$   
 $a(0) < b(0) < c(0) < a(1) \wedge$   
 $Adder3(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle).$

$\alpha = 0.1.\alpha + 1.\alpha$

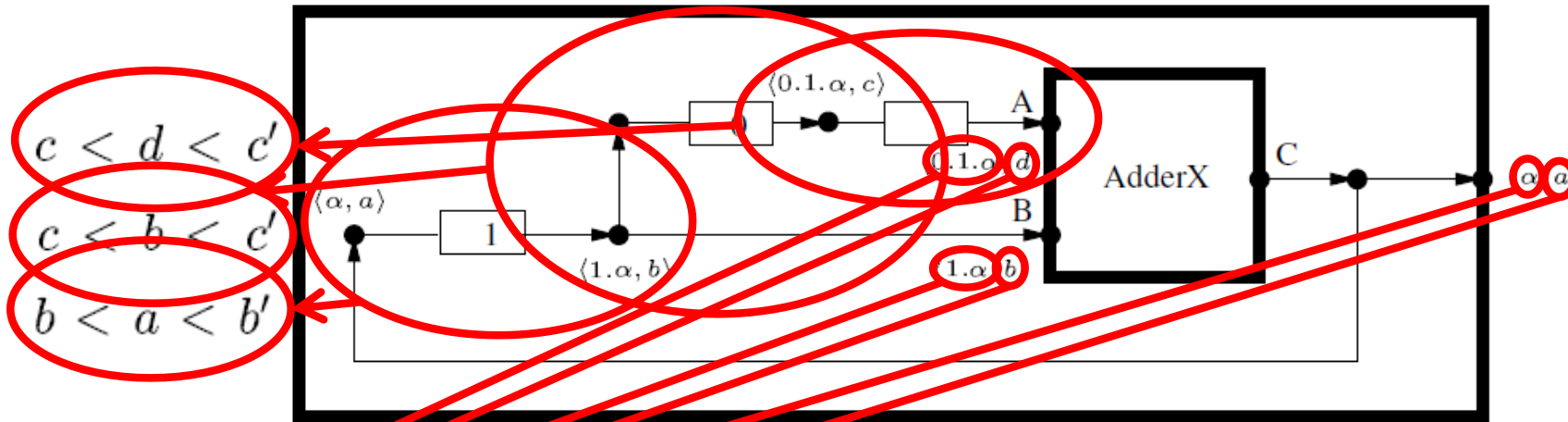
$\alpha(0) = 0 + 1 = 1$   
 $\alpha(1) = 1 + \alpha(0) = 1 + 1 = 2$   
 $\alpha(2) = \alpha(0) + \alpha(1) = 1 + 2 = 3$   
 $\alpha(3) = \alpha(1) + \alpha(2) = 2 + 3 = 5$   
 $\vdots$

$d < b < a < d'$   
 $d < b$   
 $d' < b'$   
 $d < b < a < d' < b'$

Real numbers  $c$  and  $c'$  always exist to satisfy the timing equations.

**Verified!**

# Fibonacci correctness proof (2)



$c < d < c'$   
 $c < b < c'$   
 $b < a < b'$

$Adder4(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv$

$\gamma(0) = \alpha(0) + \beta(0) \wedge$

$a = b = c \wedge$

$Adder4(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle).$

$\alpha = 0.1.\alpha + 1.\alpha$

$\alpha(0) = 0 + 1 = 1$

$\alpha(1) = 1 + \alpha(0) = 1 + 1 = 2$

$\alpha(2) = \alpha(0) + \alpha(1) = 1 + 2 = 3$

$\alpha(3) = \alpha(1) + \alpha(2) = 2 + 3 = 5$

$\vdots$

$d=b=a$

The timing equations  $b=a$   
and  $b < a$  have no solution!

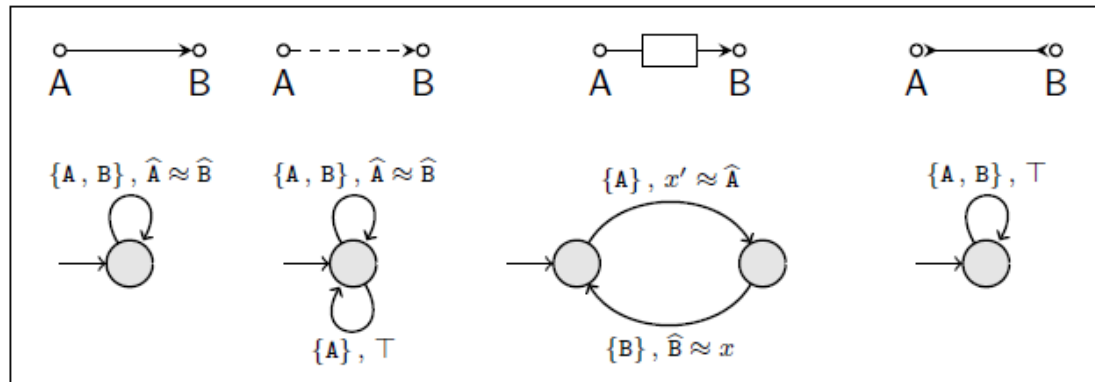
*No behavior!*

# Constraint automata

- Finite-state automata where a transition has a pair of constraints as its label:
  - (Synchronization-constraint, Data-constraint)
- Introduced to capture operational semantics of Reo



CA of typical Reo primitives:



- F. Arbab, C. Baier, J.J.M.M. Rutten, and M. Sirjani, "Modeling Component Connectors in Reo by Constraint Automata," Proc. International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2003), CONCUR 2003, Marseille, France, September 2003, Electronic Notes in Theoretical Computer Science, 97.22, Elsevier Science, July 2004.
- C. Baier, M. Sirjani, F. Arbab, and J.J.M.M. Rutten, "Modeling Component Connectors in Reo by Constraint Automata," Science of Computer Programming, Elsevier, Vol. 61, Issue 2, pp. 75-113, July 2006.
- F. Arbab, C. Baier, F.S. de Boer, and J.J.M.M. Rutten, "Models and Temporal Logical Specifications for Timed Component Connectors," International Journal on Software and Systems Modeling, pp. 59-82, Vol. 6, No. 1, March 2007, Springer.

# Product Constraint Automata

**Definition 4.1** [*Product-automaton*] The product-automaton of the two constraint automata  $\mathcal{A}_1 = (Q_1, \mathcal{N}_{ames_1}, \longrightarrow_1, Q_{0,1})$  and  $\mathcal{A}_2 = (Q_2, \mathcal{N}_{ames_2}, \longrightarrow_2, Q_{0,2})$ , is:

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}_{ames_1} \cup \mathcal{N}_{ames_2}, \longrightarrow, Q_{0,1} \times Q_{0,2})$$

where  $\longrightarrow$  is defined by the following rules:

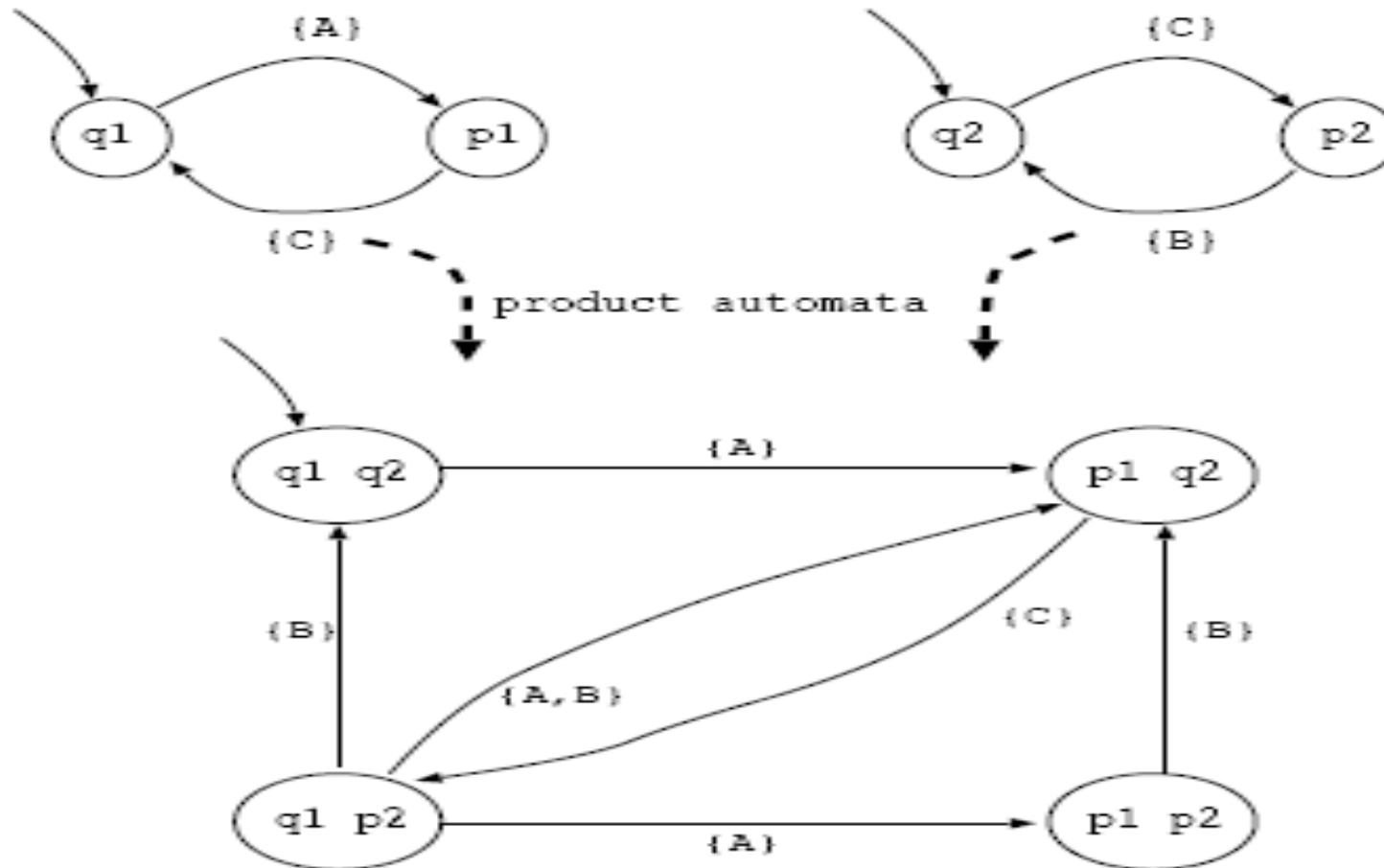
$$\frac{q_1 \xrightarrow{N_1, g_1}_1 p_1, \quad q_2 \xrightarrow{N_2, g_2}_2 p_2, \quad N_1 \cap \mathcal{N}_{ames_2} = N_2 \cap \mathcal{N}_{ames_1}}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle}$$

and

$$\frac{q_1 \xrightarrow{N, g}_1 p_1, \quad N \cap \mathcal{N}_{ames_2} = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N, g} \langle p_1, q_2 \rangle}$$

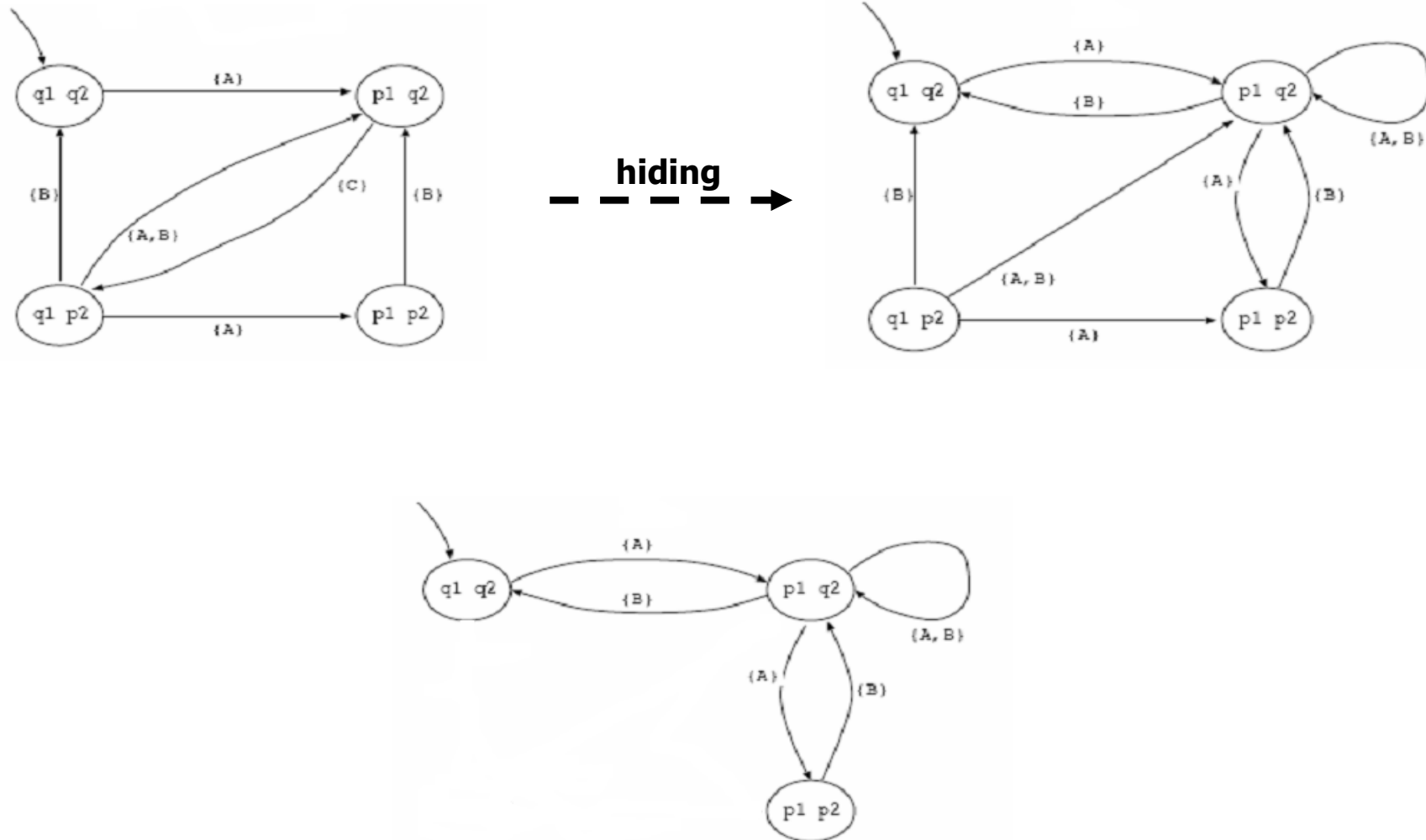
and latter's symmetric rule.  $\square$

# Product of 2 FIFO1 Automata



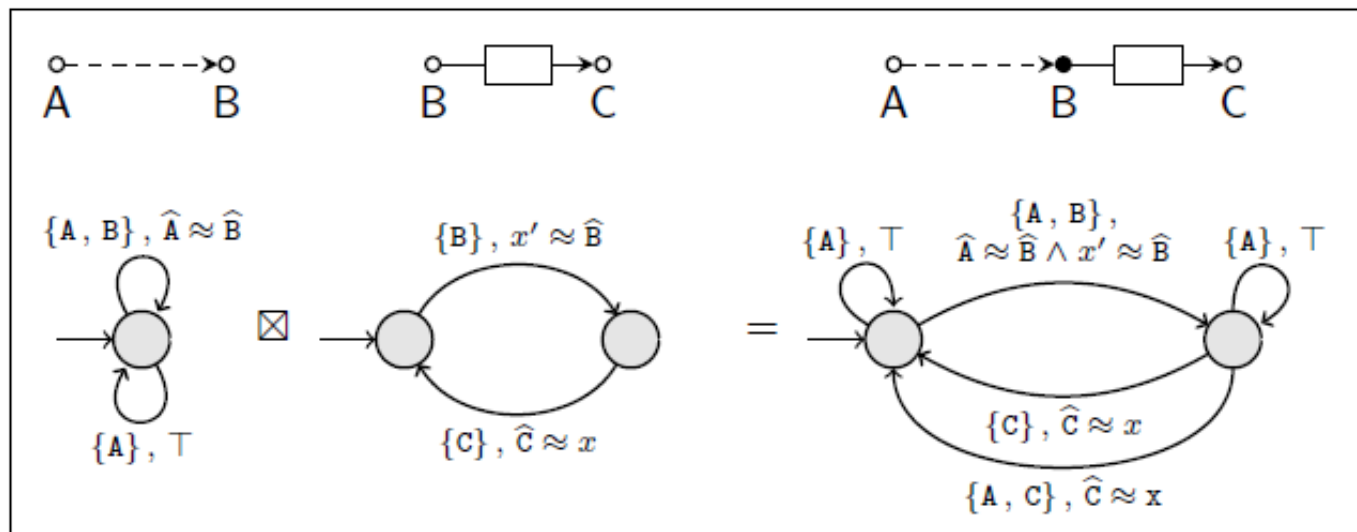


# Hiding of Node C



# CA of a connector

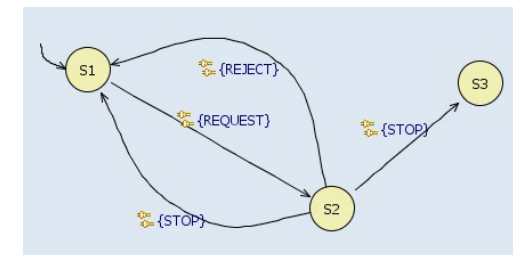
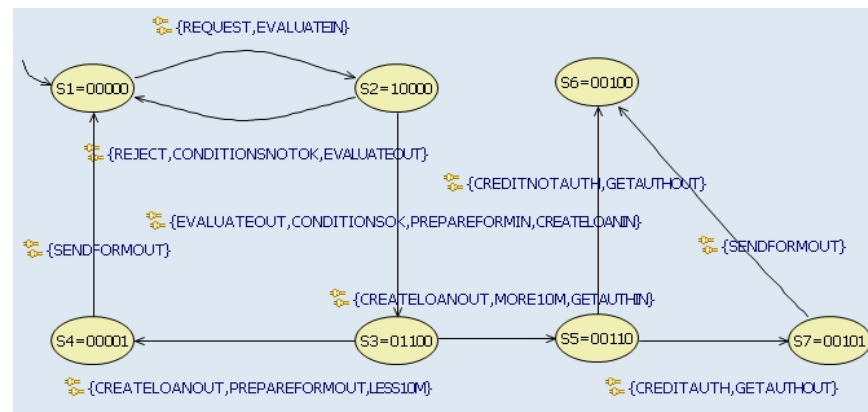
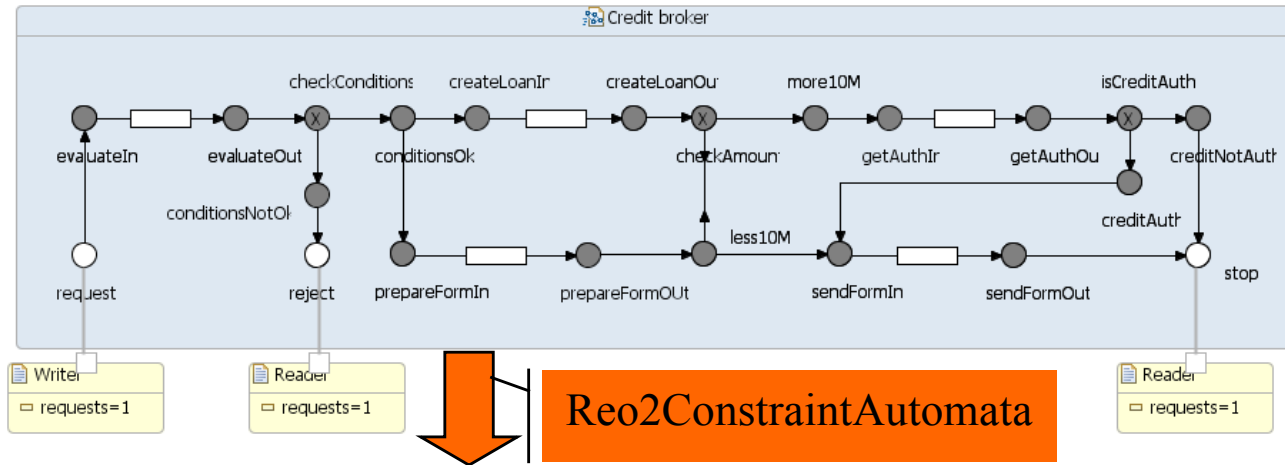
- The CA semantics of a connector is composed from the CA of its constituents via a synchronous product operator.



# Vereofy Model Checker

- Symbolic model checker for Reo:
  - Based on constraint automata
  - Developed at the University of Dresden
  - LTL and CTL-like logic for property specification
- Modal formulae
  - Branching time temporal logic:
    - $AG[EX[true]]$
    - check for deadlocks
  - Linear temporal logics:
    - $G(request \rightarrow F(reject \cup sendFormOut))$
    - check that admissible states *reject* or *sendFormOut* are reached
- <http://www.veroeffy.de>

# Verification with Vereofy



## □ Modal formulae

- Branching time temporal logic:  $AG[EX[true]]$  - check for deadlocks
- Linear temporal logics:  $G(request \rightarrow F(reject \cup sendFormOut))$  - check that admissible states *reject* or *sendFormOut* are reached

# Context Sensitive Behavior

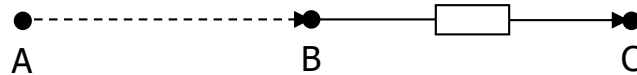
- ❑ Certain channels may have context-sensitive behavior.
- ❑ Nodes must respect and propagate such context information.

write -----> take

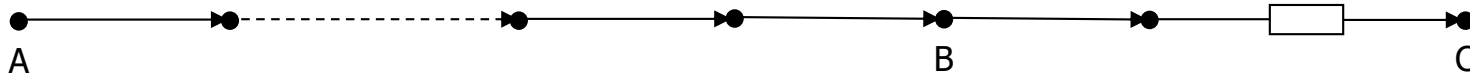
Write and take, data is, data must be transferred

# Effect on Node Behavior

- Node B must make sure that the first write to A is never lost.



- Even in this case



# Other Automata Models



- ❑ The pure CA cannot capture context sensitivity directly.
- ❑ Two alternatives
  - Extensions to CA are necessary:
    - Intentional Constraint Automata
    - Context sensitive CA
    - Reo automata (ready ports, not-ready ports, firing ports)
  - Encode context sensitivity on top of CA

# Distributed Semantics

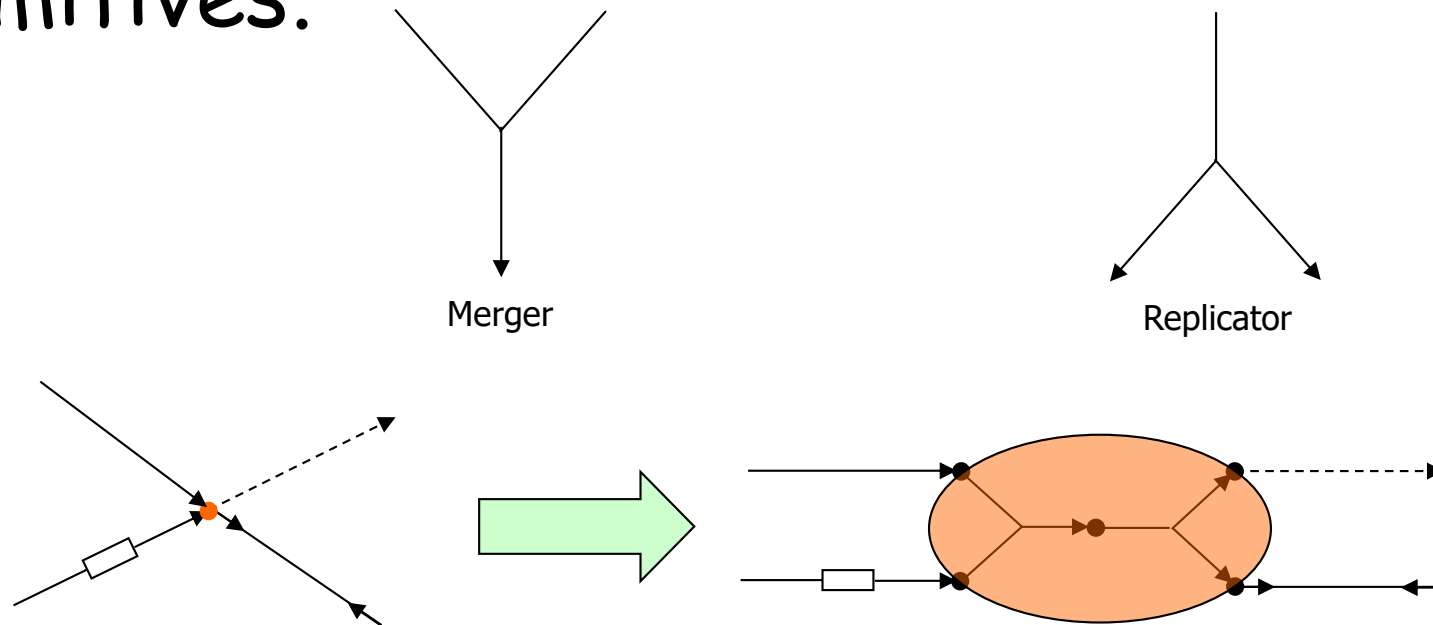


- ❑ Automata models capture the global behavior of a Reo circuit
- ❑ Reo primitives (must) act locally
  - Need a model to allow global behavior of a circuit emerge as a consensus of the possible local behavior alternatives of its primitives.
  - Primitives that coincide on a node must agree on a common behavior
    - Primitives constrain each other's behavior alternatives
    - Viable global behavior can be found through constraint solving.



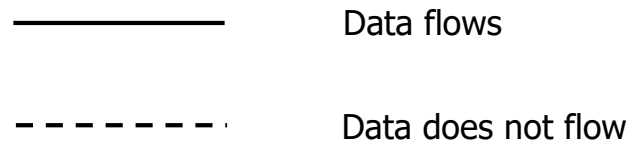
# Node Expansion

- Explicitly represent the merge and replicate behavior of nodes as (builtin) primitives.



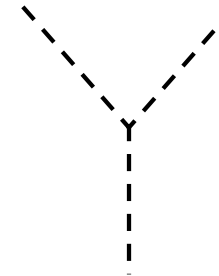
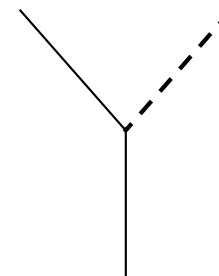
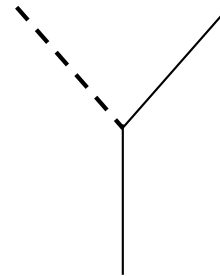
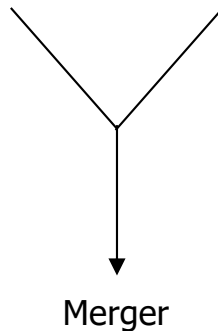
# Coloring Semantics

- A model for the semantics of Reo
  - Preserves circuit topology.
  - Allows an open set of primitives.
  - Composes behavior alternatives of primitives.
  - Suitable for distributed implementation.
- We use (initially two) different colors to represent alternative forms of (dataflow) behavior of primitives.



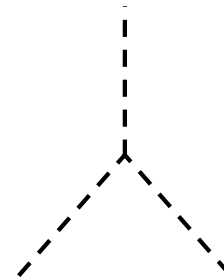
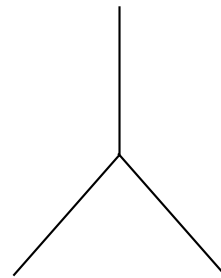
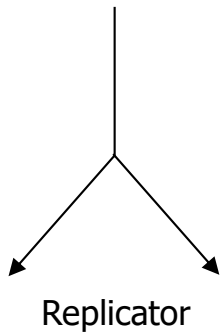
# Merger (2-color)

- Alternative forms of dataflow behavior of merger in the 2-color scheme.



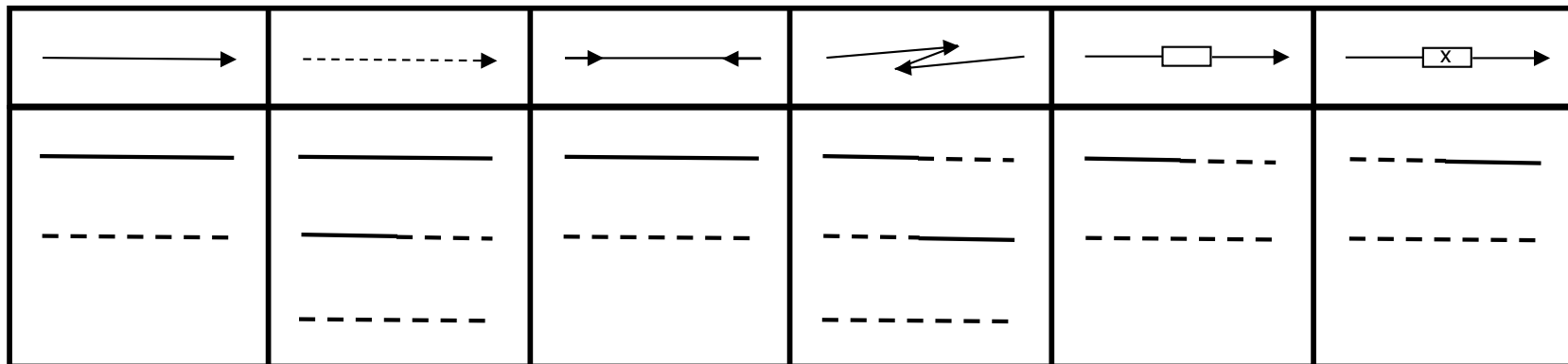
# Replicator (2-color)

- Alternative forms of dataflow behavior of replicator in the 2-color scheme.



# 2-color Scheme

- Alternative forms of dataflow behavior of a typical set of channels.

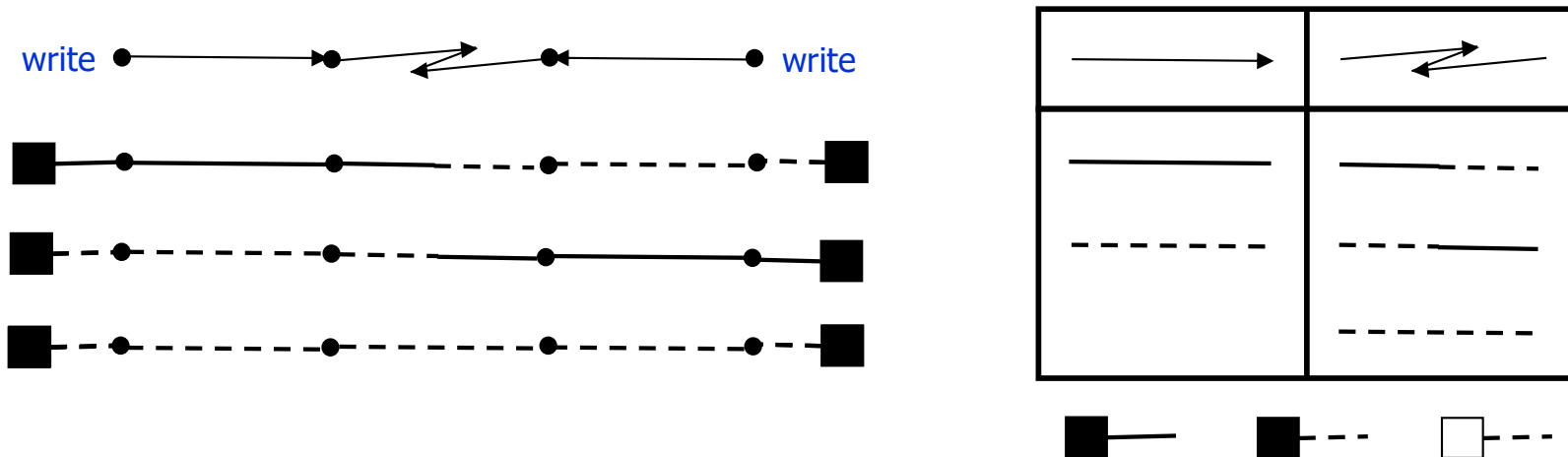


- Representing I/O operations at boundary nodes:



# Circuit Coloring

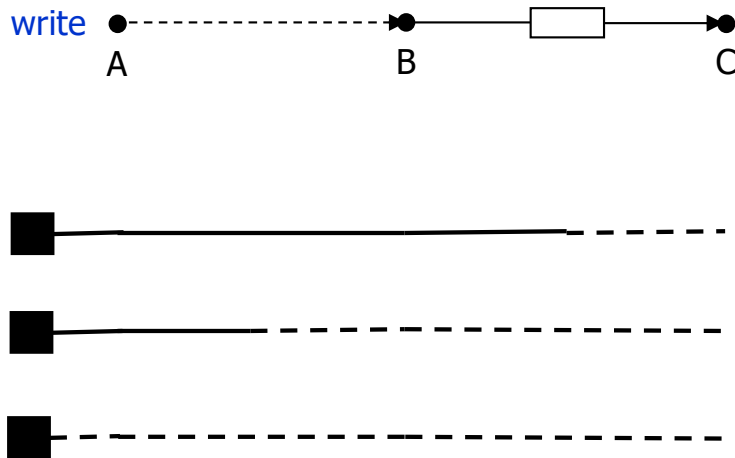
- Nodes must match the colors of their coincident channel ends.



- Total no-flow alternative always exists.
  - Annoyance: unbridled non-determinism can always choose it.

# Lack of Context Awareness

- The 2-color scheme does not support context-sensitivity.



# 3-color Scheme

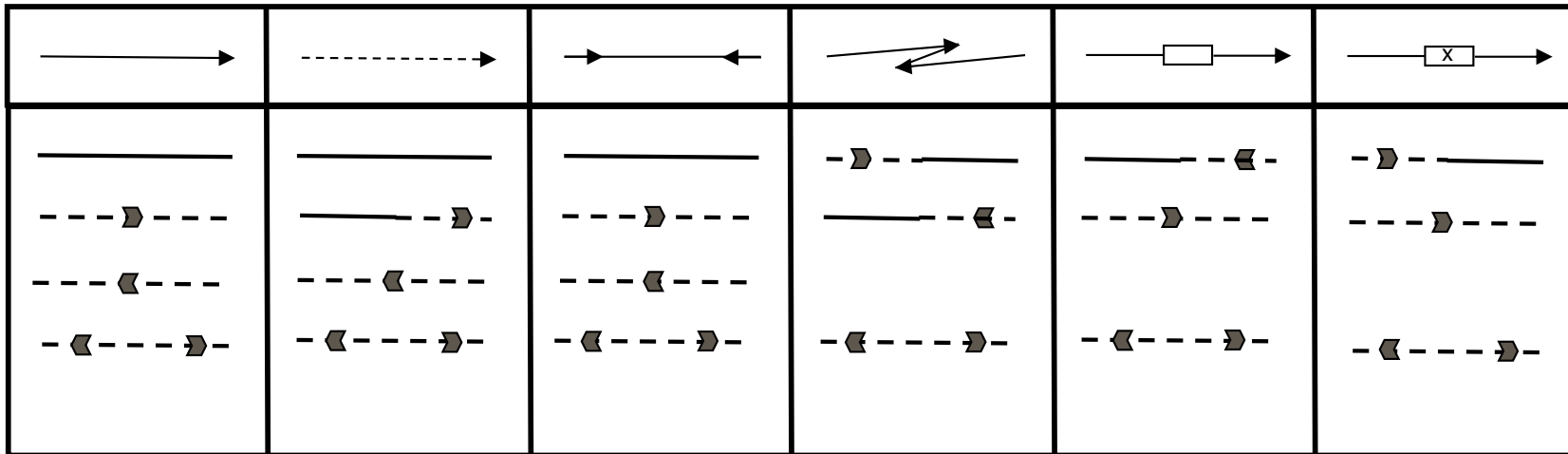


- Two different reasons for no-flow:
  - Unavailability:
    - A (place-holder for a) data item does not exist.
  - Exclusion:
    - The state of the channel refuses to use it.
- Adorn no-flow with one of two markers to show its cause.



# 3-color Scheme

- Distinguish between the two possible causes of no-flow:
  - Non-availability: inbound chevron
  - Exclusion: outbound chevron
  - The chevron points to the reason for no-flow

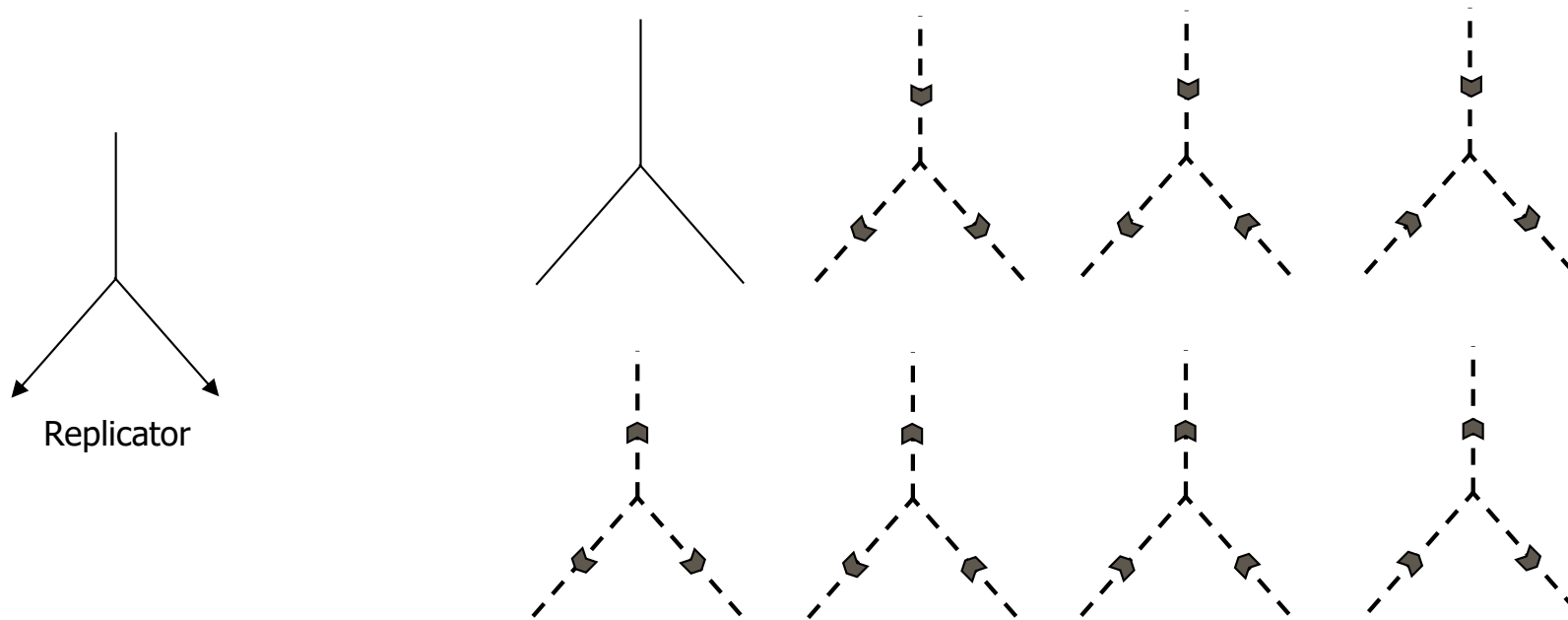


- Representing I/O operations at boundary nodes:



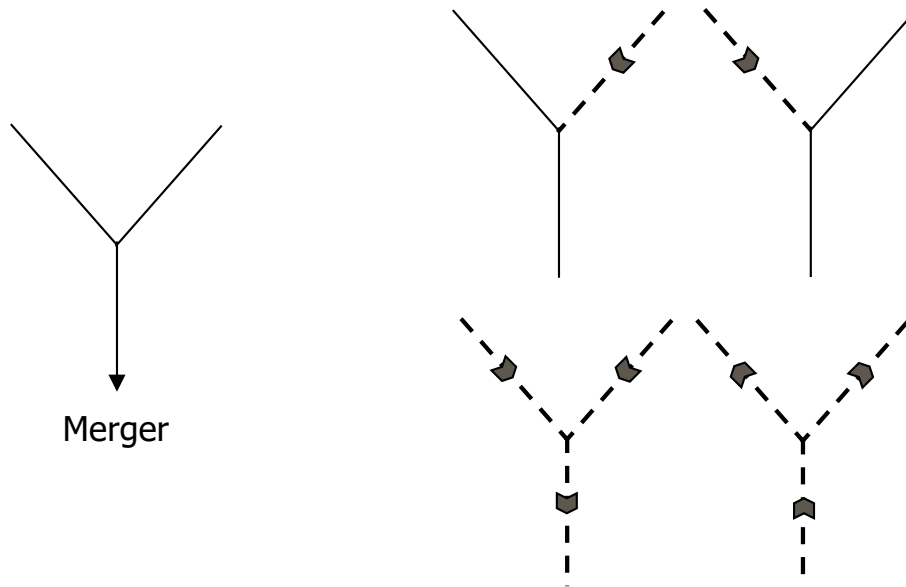
# Replicator (3-color)

- Alternative forms of dataflow behavior of replicator in the 3-color scheme.



# Merger (3-color)

- Alternative forms of dataflow behavior of merger in the 3-color scheme.



# General Rules for 3-color Primitives (1)

- In sensible primitives:
  - A no-flow behavior alternative with exclusion on all of its ends is not allowed.

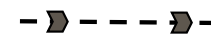
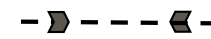


# General Rules for 3-color Primitives (2)

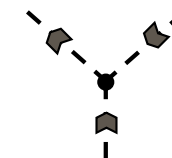
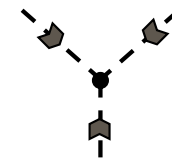
## □ In sensible primitives:

- The existence of a behavior alternative with an exclusion no-flow on one of its ends implies that the primitive tolerates non-availability no-flow on that same end.

- If this is present
- Then this must be implied as well
- If this is present

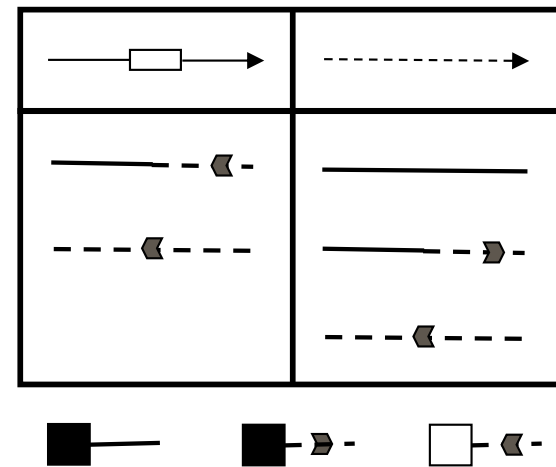
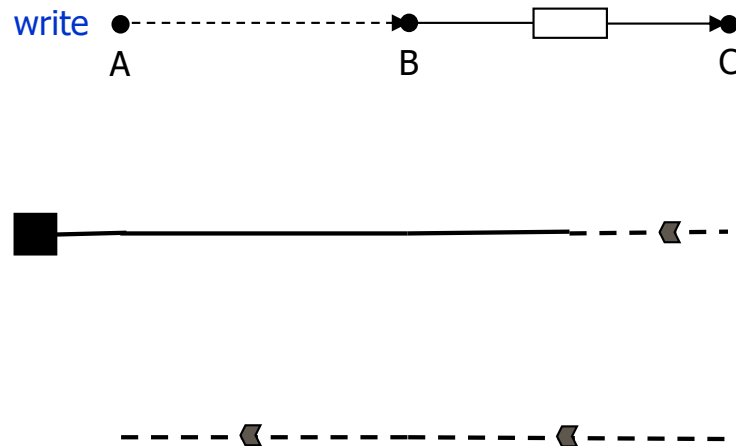


- Then these must be implied as well



# Context Awareness

- The 3-color scheme supports context-sensitivity.



- It works even when Sync channels are inserted at B!

# Extensible Coordination Tools



- ❑ A set of Eclipse plug-ins provide the ECT visual programming environment.
- ❑ Protocols can be designed by composing Reo circuits in a graphical editor.
- ❑ The Reo circuit can be **animated** in ECT.
- ❑ ECT can automatically generate the CA for a Reo circuit.
- ❑ Model-checkers integrated in ECT can be used to verify the correctness properties of a protocol using its CA.
- ❑ ECT can generate executable (Java/C) code from a CA as a single sequential thread.

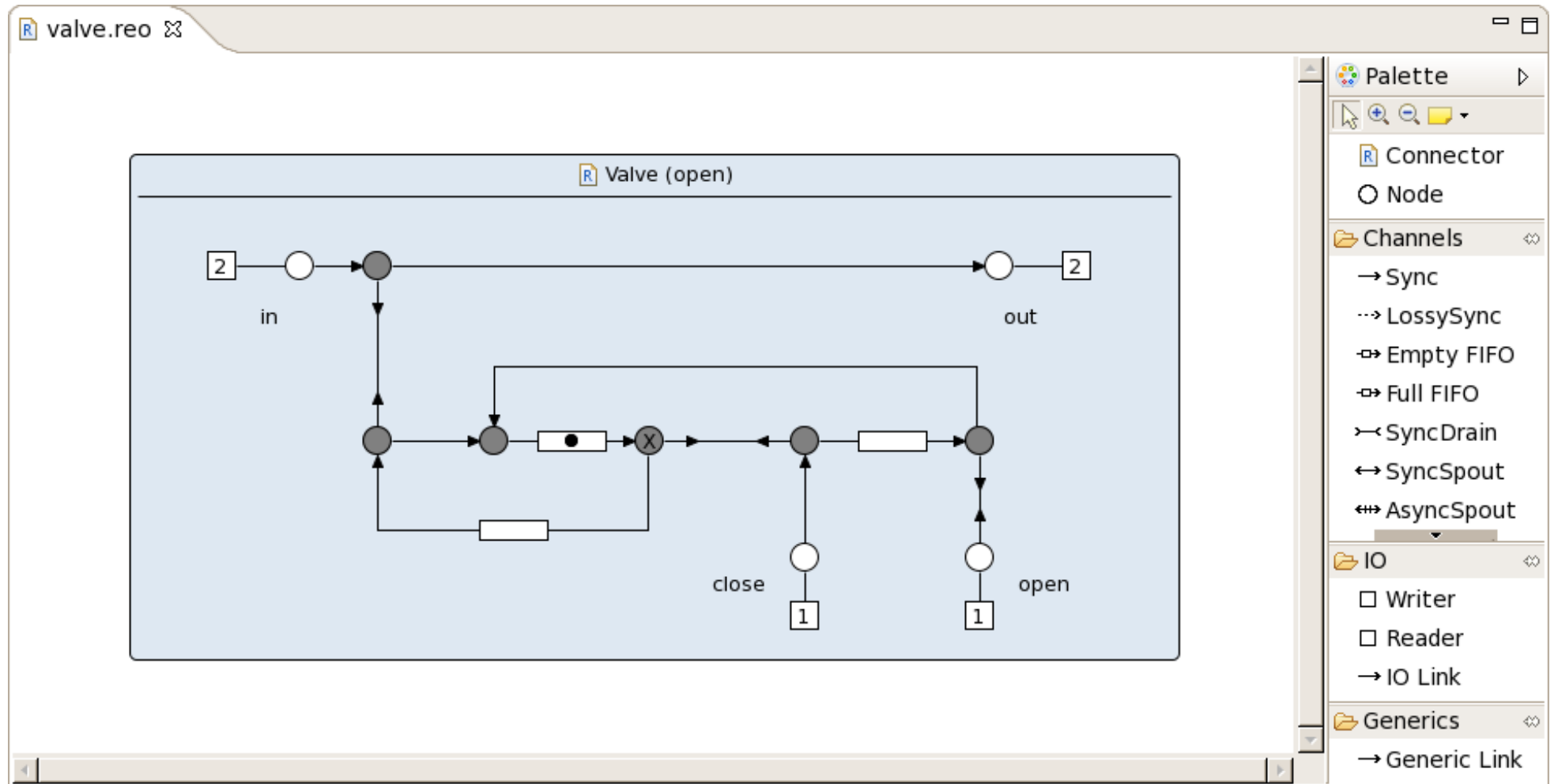
<http://reo.project.cwi.nl>

# Tool support

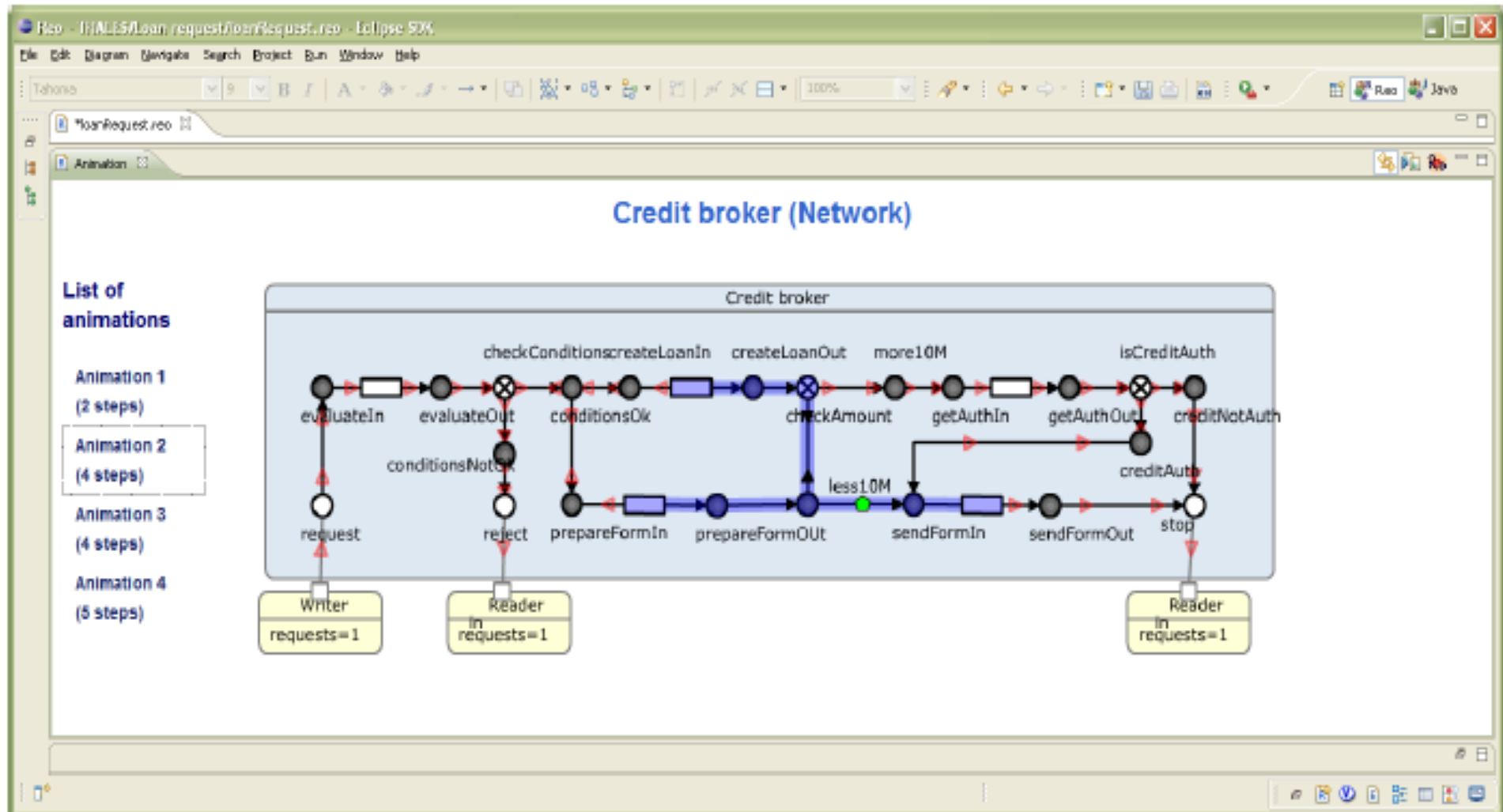
Tool	Description
Reo graphical editor	Drag and drop editing of Reo circuits
Reo animation plug-in	Flash animation of data-flow in Reo circuits
Extensible Automata editor and tools	Graphical editor and other automata tools
Reo to constraint automata converter	Conversion of Reo to Constraint Automata
Verification tools	<ul style="list-style-type: none"><li>•Vereofy model checker (<a href="http://www.vereofy.de">www.vereofy.de</a>)</li><li>•mCRL model checking</li><li>•Bounded model checking of Timed Constraint Automata</li></ul>
Java code generation plug-in	State machine based coordinator code (Java, C, and CA interpreter for Tomcat servlets)
Distributed Reo middleware	Distributed Reo code generated in Scala (Actor-based Java)
(UML / BPMN / BPEL) GMT to Reo converter	Automatic translation of UML SD / BPMN / BPEL to Reo
Reo Services platform	Web service wrappers and Mash-ups
Markov chain generator	Compositional QoS model based on Reo Analysis using, e.g., probabilistic symbolic model checker Prism ( <a href="http://www.prismmodelchecker.org">http://www.prismmodelchecker.org</a> )
Algebraic Graph Transformation	Dynamic reconfiguration of Reo circuits



# Snapshot of Reo Editor



# Reo Animation Tool

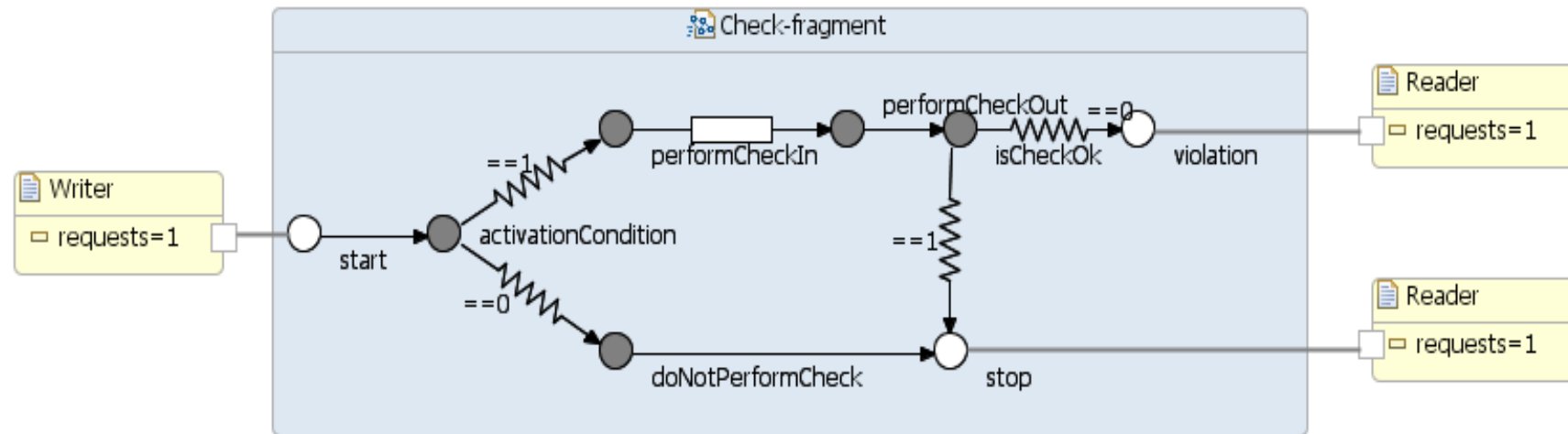


# Model Checking



- ❑ Constraint automata are used for model checking of Reo circuits
- ❑ Model checker for Reo built in Dresden:
  - Symbolic model, LTL, and CTL-like logic for specification
  - Can also verify properties such as deadlock-freeness and behavioral equivalence
- ❑ SAT-based bounded model checking of Timed Constraint Automata
- ❑ Translation of Reo to mCRL for model checking

# Data-Dependent Control-Flow



## □ Input parameters:

- Activation condition
  - **Data:** b: Boolean
  - **Filter condition:** b==true, b==false
- Check condition
  - **Data:** x, y: Real; (e.g., credit amount, maximal amount)
  - **Filter condition:** x < y

## □ Problems:

- Data constraint specification language is needed
- Properties that include conditions:
  - $G [(b \ \& \ !(x < y)) \rightarrow F \ \text{violation}]$

# Verification with mCRL2

- mCRL2 behavioral specification language and associated toolset developed at TU Eindhoven
  - <http://www.mcrl2.org>
  - Based on the Algebra of Communicating Processes (ACP)
  - Extended with data and time
  - Expressive property specification language ( $\mu$  calculus)
  - Abstract data types, functional language ( $\lambda$  calculus)
- Automated mapping from Reo to mCRL2
  - N. Kokash, E. d. V., C. Krause, Data-aware Design and Verification of Service Compositions with Reo and mCRL2, in: ACM Symposium on Applied Computing, 2010

# Data flow analysis with mCRL2

The image shows the Eclipse IDE with a project named 'loanRequest.reo'. The main editor displays a process algebra specification for a loan request system. The specification includes components like 'Writer', 'Reader', and 'loanRequest' with various transitions and guards. A callout box points to this specification with the text 'Generated process algebra specification'.

Below the specification, the 'Specification' panel shows the following code:

```

Options:  With components  With data  With colours  Intensional end
Traversal:  none  depth-first  breadth-first
Output: sort
Data = struct d1(e1 : DataWriter1) ?isData
DataWriter1 = struct request(amount: Pos,
DataWriter3 = struct Alice;
DataFIFO = struct empty | full(e : Data);

act
Approved, Approved', Approved'', Approved0, Approved0', Approve
Authorized0'', CheckClientProfileIn, CheckClientProfileIn', Che
CheckClientProfileOut0'', Denied, Denied', Denied'', Denied0, D
IsSalarySufficient0'', IsSalarySufficient1, IsSalarySufficient1
M'', N, N', N'', ProcessRequestIn, ProcessRequestIn', ProcessRe

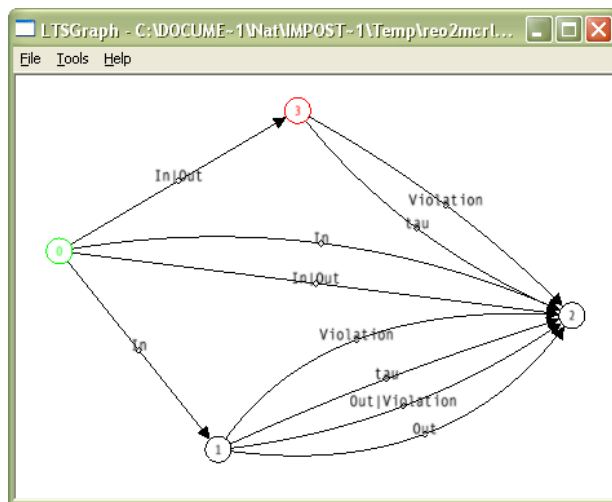
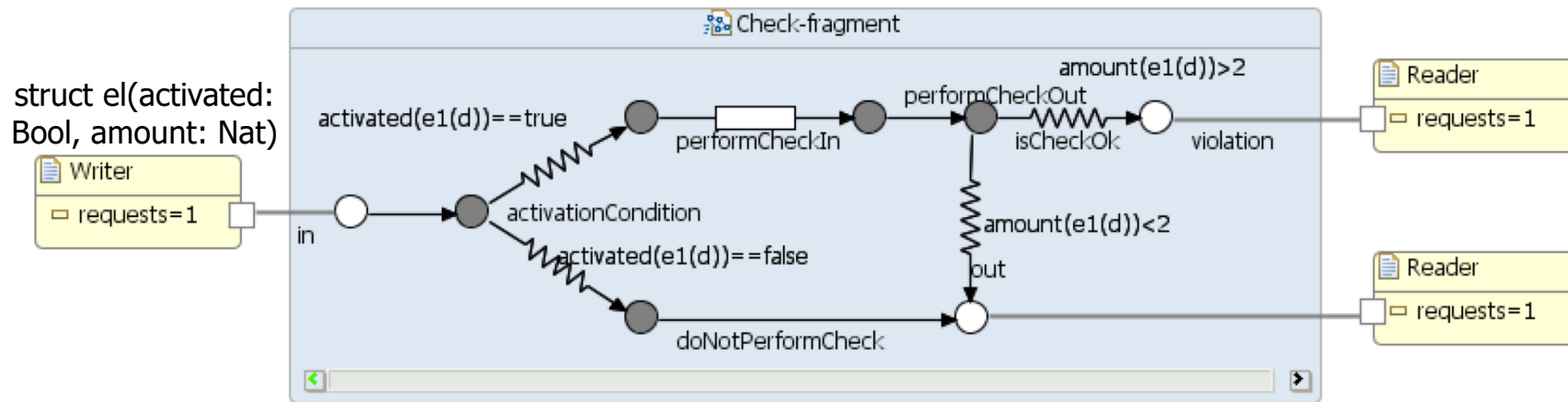
```

The 'Formula' field contains the property: `[true*]<true>true`. Buttons for 'Check' and 'Show LTS' are visible.

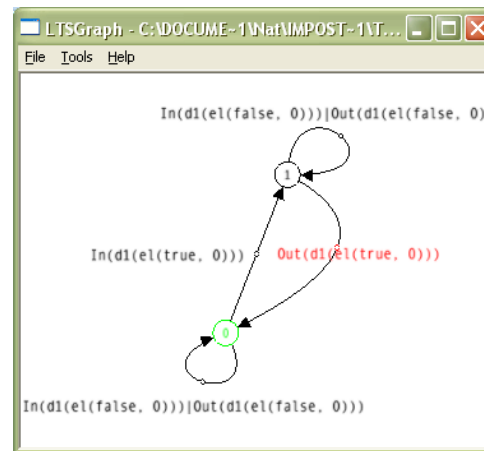
To the right, the 'LTSGraph' window displays a transition system with five states (0, 1, 2, 3, 4) and transitions labeled with actions like 'authorized(d2(Alice))' and 'approved(Tuple2(d1(request(10000, 3000, 1)), d2(Alice)))'. A callout box points to this graph with the text 'Show labeled transition system'.

At the bottom right, another callout box points to the 'Check' button with the text 'Specify and check formal property'.

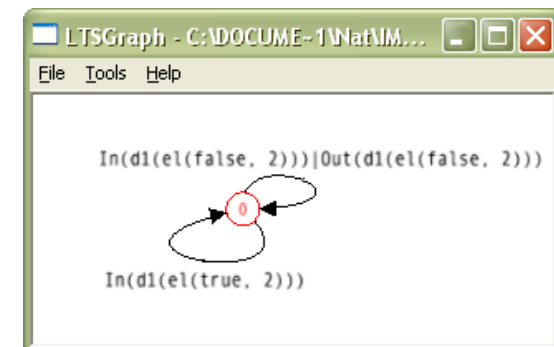
# Data Dependent Control Flow



No data



(amount(d) < 1)



(amount(d) == 2)

# Process verification tools: summary



## □ Vereofy:

### ○ Advantages:

- Developed for Reo and Constraint Automata
- Visualization of counterexamples

### ○ Disadvantages:

- No support for abstract data types
- Global domain for all components
- Primitive data constraint specification language (for filter channels)

## □ mCRL2

### ○ Advantages:

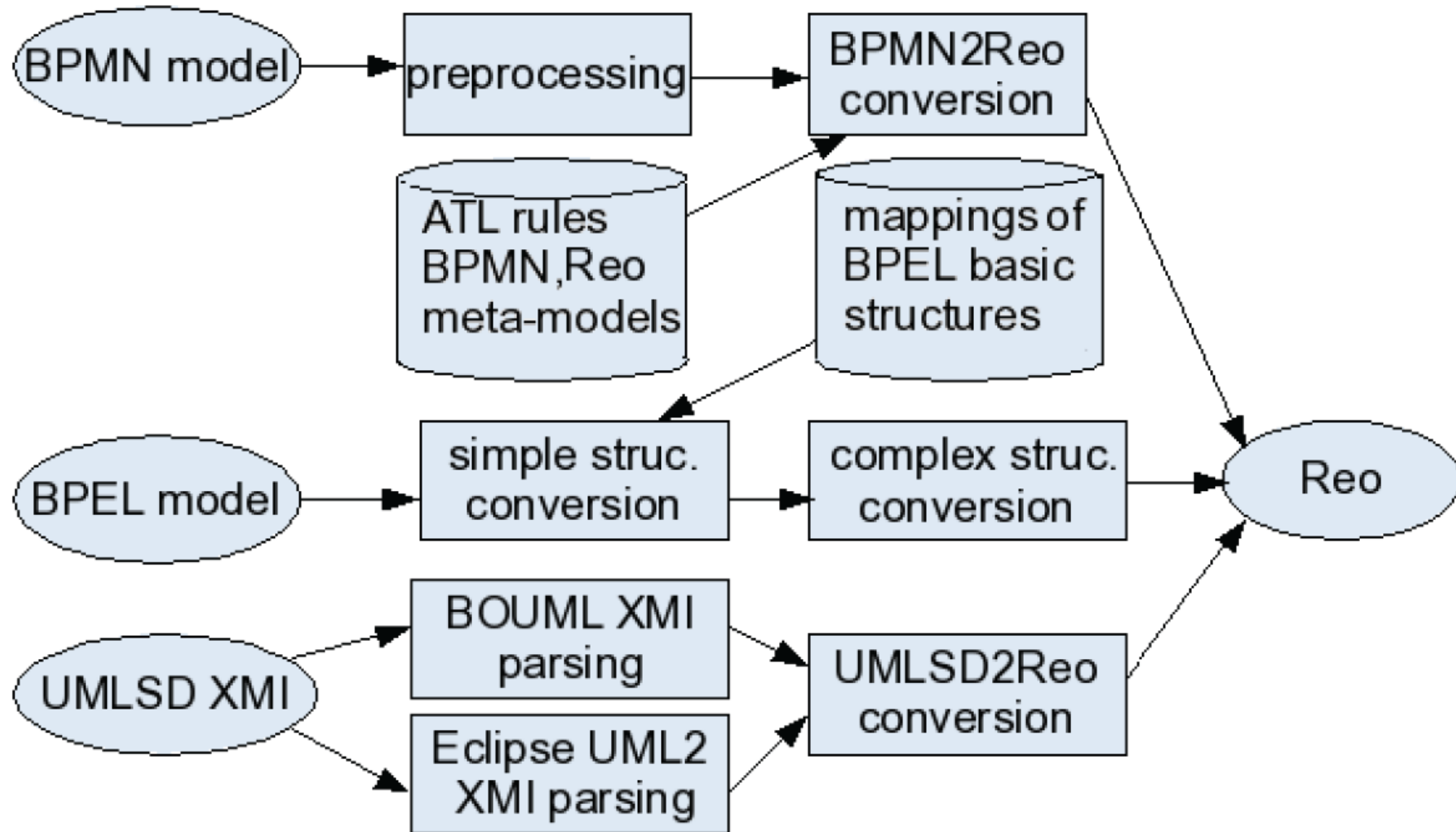
- Support abstract data types including lists and sets
- Allows the definition of functions
- Very rich property specification format (mu-calculus)

### ○ Disadvantages:

- Hard to extract counterexamples
- For infinite domains model checker often does not terminate (problems with algorithms for formulae rewriting)



# Architecture of ECT Converters



# Building an application in ECT

The screenshot illustrates the steps for building an application in ECT within the Eclipse IDE:

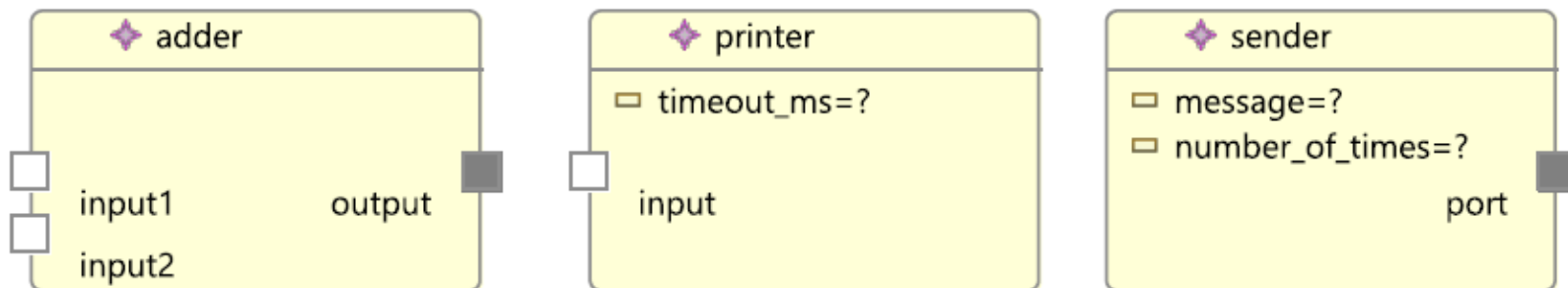
1. A diagram is shown with nodes A, B, and C, and a 'syncer' component.
2. A context menu is open over the diagram, with 'Add Source Code' selected.
3. A file explorer window shows the selection of 'test.h' (C/C++ Header) in the 'components' directory.
4. A 'Selection Needed' dialog box is open, showing a list of functions to include: sender, printer, signaller, and adder. The first three are selected.
5. Another file explorer window shows the selection of 'test.lib' (Object File Library) in the 'components' directory.

# Import computation code

- ❑ Drag and drop computation code written in C onto the canvas in the ECT to create components.
- ❑ Contents of source or header file:

```
void adder(REOPortIn* input1, REOPortIn* input2, REOPortOut* output);  
void printer(REOPortIn* input, int timeout_ms);  
void sender(REOPortOut* port, char* message, int number_of_times);
```

- ❑ Created components



# Building an application in ECT

The screenshot displays the Eclipse IDE interface for building a Reo application. The main workspace shows a Reo diagram with the following components and connections:

- sender** (yellow box): Contains properties `string=?` and `max_iters=10`. It has a `port` on its right side.
- syncer** (blue box): A central component with three nodes labeled `A`, `B`, and `C`. Node `A` is connected to the `port` of the `sender`. Node `B` is connected to the `port` of the `printer`. Node `C` is connected to the `port` of the `signaller`.
- printer** (yellow box): Contains the property `timeout=1000`. It has a `port` on its left side.
- signaller** (yellow box): Contains the property `max_iters=10`. It has a `port` on its right side.

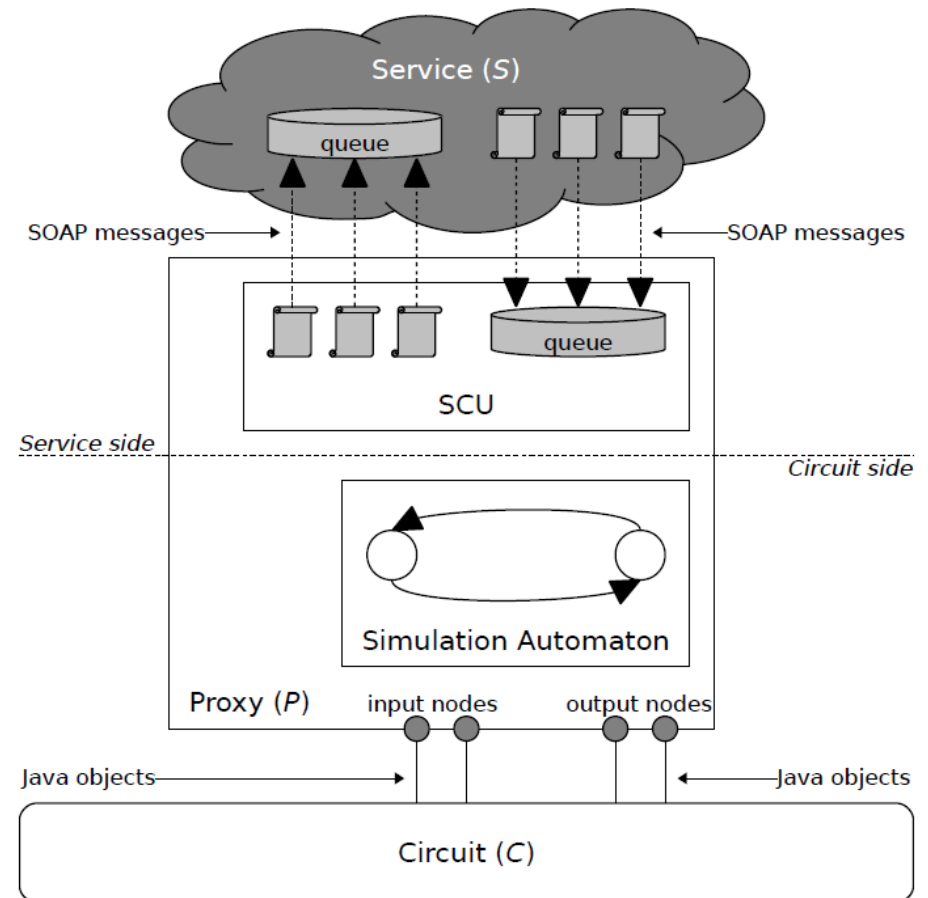
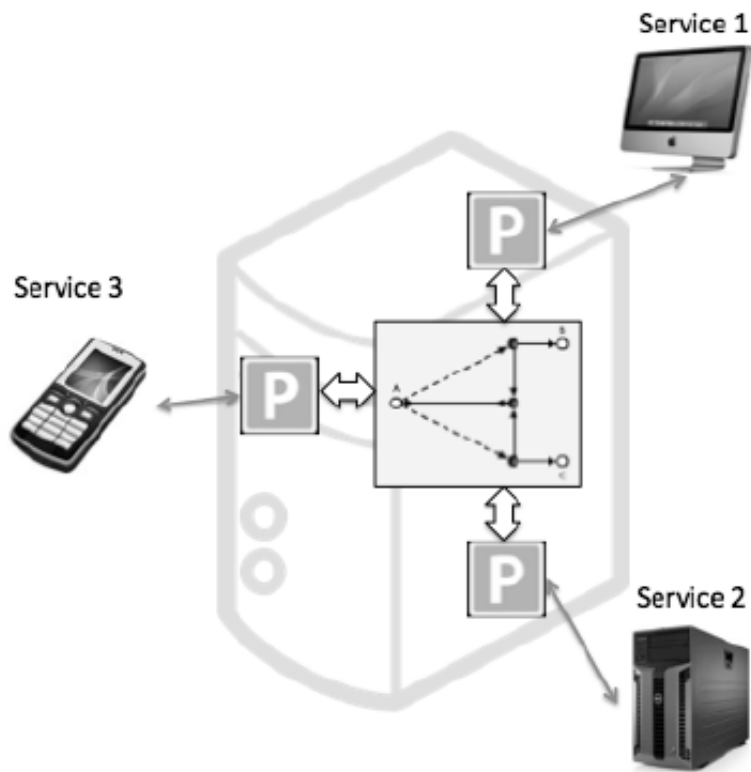
The **Core Generator** settings panel at the bottom is configured as follows:

- Connector:** `From editor`
- Destination:** `To C code (file system)`
- Output Path:** `C:\Users\Mathijs\Desktop\export_eclipse`
- Generate** button is visible.

Red handwritten numbers are overlaid on the image to highlight specific elements:

- 6:** Points to the `signaller` component.
- 7:** Points to the `printer` component.
- 8:** Points to the `Destination` dropdown menu.
- 9:** Points to the output path text field.
- 10:** Points to the `Generate` button.

# Service proxies



# Proxy generation

The screenshot displays the Eclipse Platform interface for the Proxy Generator tool. The main window is titled "Java - Connectors/service-orchestration-scenario.reo - Eclipse Platform". The "Proxy Generator" tool is active, showing the following settings:

- Technology: WSDL
- WSDL file: <http://localhost:8080/IncService?wsdl>
- Service name: IncService
- Simulation automaton: From file system
- WSDL file path: <local:\services\src\service\IncService.wsdl>
- Destination: To new project in workspace

Buttons at the bottom include "Draw", "Generate Proxy", and "Generate Orchestration".

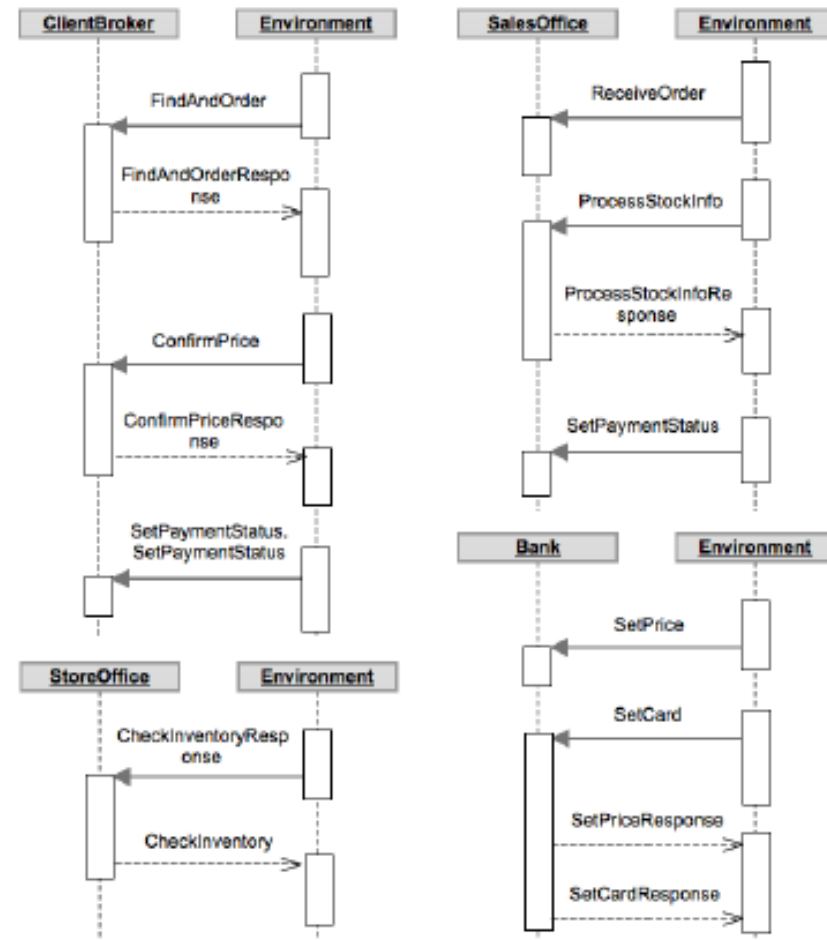
The main diagram area shows a BPMN diagram titled "OrchestratorCl...". It features two service interfaces:

- IncService**: Includes operations `SetInc`, `SetInc`, `Inc`, and `Inc` with response `IncResponse`.
- CalcService**: Includes operations `CalcPrimeFactors` with responses `CalcPrimeFactorsSoapIn` and `CalcPrimeFactorsSoapOut`.

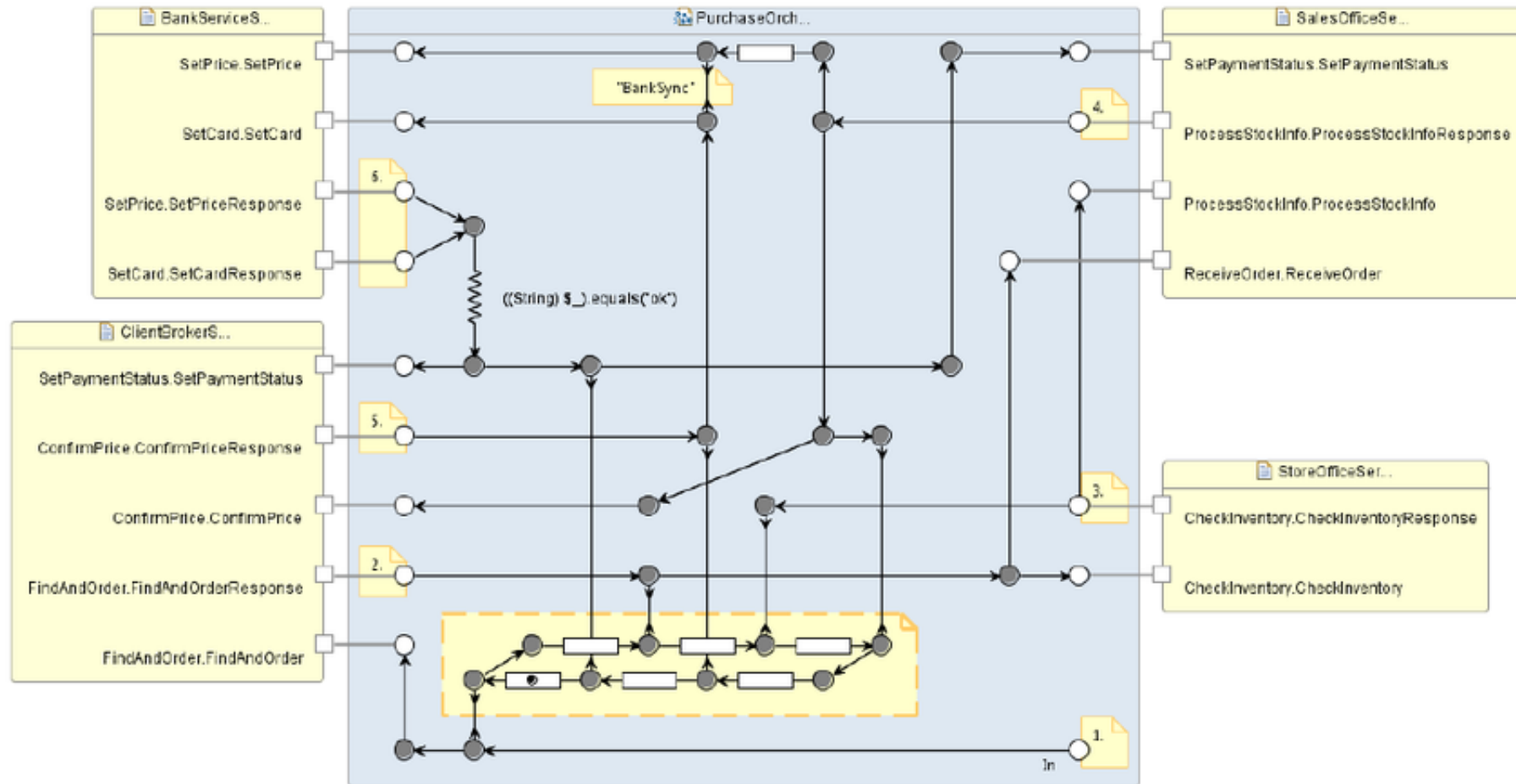
The diagram includes a start event labeled "Inc", a parallel gateway, and a task labeled "N". A task labeled "Res" is connected to a complex gateway with the following conditions: `(((String)$_.split(" ").length==1` and `(((String)$_.split(" ").length>1`. The diagram also shows a "Chann..." palette and an "I/O" palette on the right side.

# Service behavior specification

- ❑ A WSDL file describes the syntax of messages accepted by a service.
- ❑ The behavior of a (stateful) service is given by a CA.
- ❑ Instead of a CA, service behavior can be specified as UML sequence diagrams.
- ❑ In principle, any sufficiently complete formal specification of the behavior of a service is acceptable.
- ❑ ECT tools use WSDL and behavior specifications of a service to automatically generate its simulation automaton and its proxy.

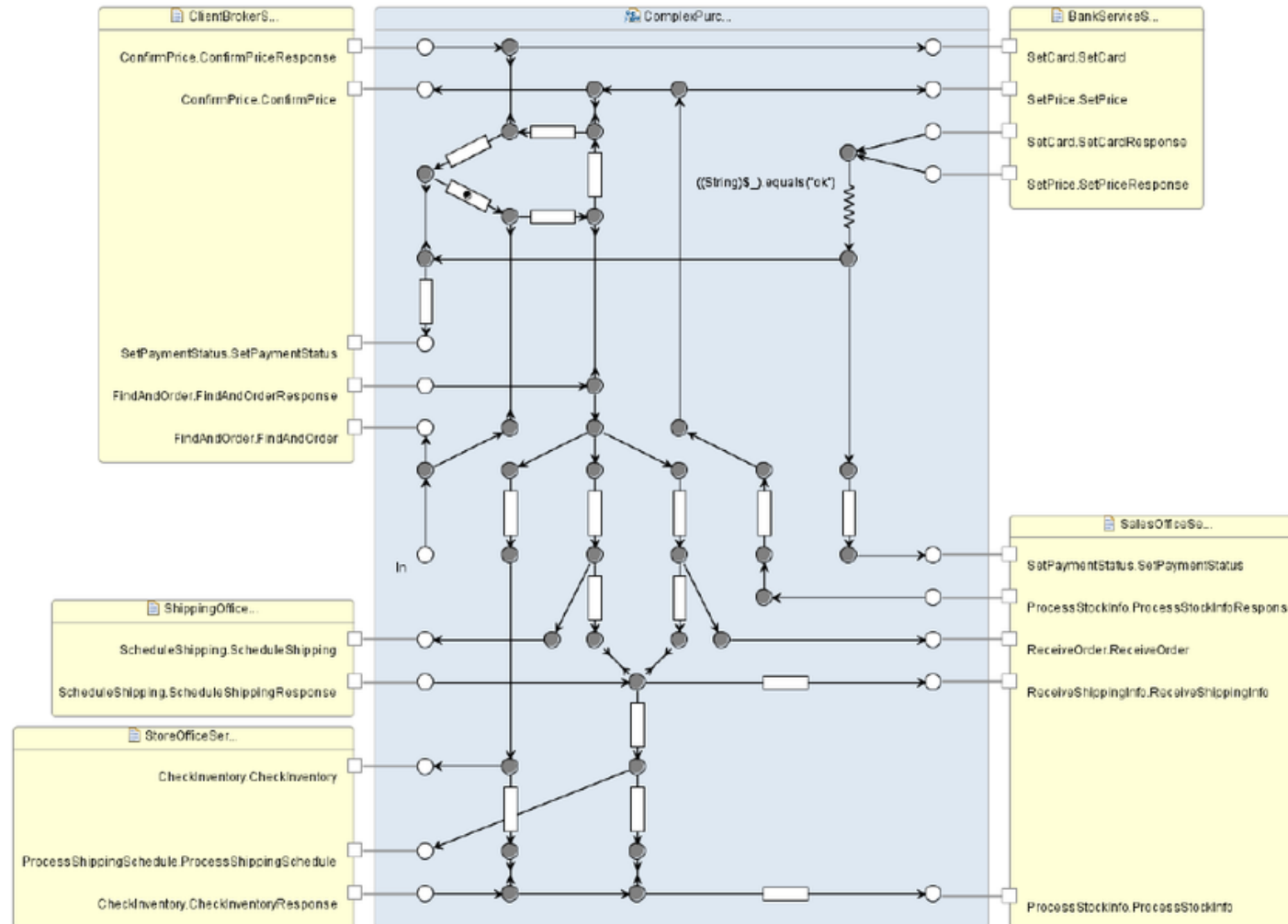


# A simple purchase scenario





# An advanced purchase scenario



# Executable code generation

## □ Reo makes interaction explicit and tangible, allowing

- Specification
- Composition
- Analysis
- Verification
- Reuse

Of interaction protocols

## □ Efficient executable code directly from Reo models?

- Performance comparable to hand-crafted optimized code.
- Choreography of Web services
- Coordinated composition of distributed components
- Concurrent applications on multi-core platforms

## □ Use Constraint Automata

# Centralized vs. distributed

- Centralized implementation of circuit with  $n$  primitives:
  - A single coordinator/protocol process: state machine of *the* protocol CA
  - Poor scalability (at compile- and run-time)
  - Minimal concurrency
  - All synchronization resolved in CA product at compile-time
    - Low run-time overhead
- Distributed implementation of circuit with  $n$  primitives:
  - Every primitive runs as a separate state machine:  $n$  processes
  - Excellent scalability (at compile- and run-time)
  - Maximal concurrency
  - Must resolve all synchronization through consensus at run-time
    - High run-time overhead
- Hybrid implementation of circuit with  $n$  primitives:
  - Start from distributed and remove useless concurrency, moving toward centralized
  - A total of  $1 \leq m \leq n$  state machines running as separate processes
  - Best of both worlds!

# Useful vs. useless concurrency

## □ Useful concurrency:

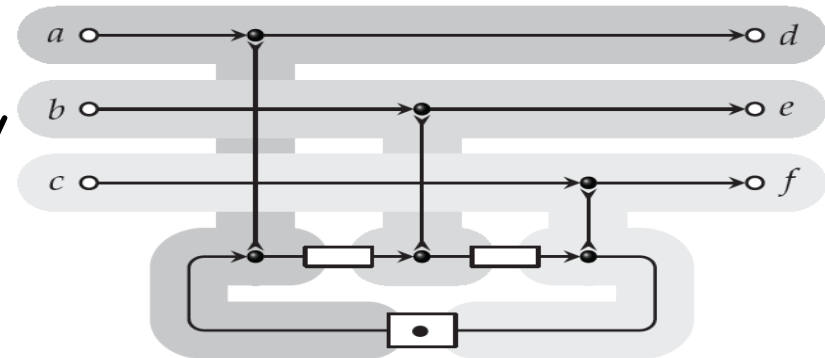
- A unit can make a move independently of another
- Units require only cheap run-time consensus involving only local communication

## □ Useless concurrency

- A unit must reach consensus about its global behavior before it can make a move.
- Units require expensive run-time consensus involving non-local communication

# Compiling Reo onto multi-core

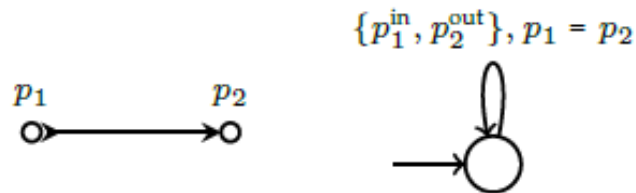
- ❑ Splits a Reo circuit into synchronous islands.
- ❑ Compiles each island into a constraint automaton.
- ❑ Maps asynchronous regions (FIFOs) into passive shared memory.
- ❑ Each island runs as a separate state machine thread concurrently with computation threads.



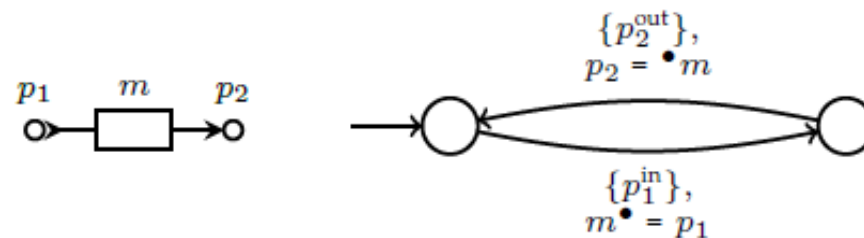
- Sung-Shik T.Q. Jongmans and Farhad Arbab, "Can High Throughput Atone for High Latency in Compiler-Generated Protocol Code?," LNCS, FSEN 2015, April 22-24, 2015, Tehran, Iran.
- Sung-Shik T. Q. Jongmans and Farhad Arbab, "Toward Sequentializing Overparallelized Protocol Code," ICE 2014: pp. 38-44.
- Sung-Shik T. Q. Jongmans, Sean Halle and Farhad Arbab, "Automata-Based Optimization of Interaction Protocols for Scalable Multicore Platforms," the 16th International Conference on Coordination Models and Languages ([Coordination 2014](#)), June 3-6, 2014, Berlin, Germany, LNCS 8459, pp 65-82.
- Sung-Shik T. Q. Jongmans and Farhad Arbab, "Global Consensus through Local Synchronization," *Advances in Service-Oriented and Cloud Computing Communications in Computer and Information Science*, Vol. 393, pp 174-188, 2013.
- Sung-Shik T.Q. Jongmans, Sean Halle and Farhad Arbab, "Reo: A Dataflow Inspired Language for Multicore," Data-Flow Execution Models for Extreme Scale Computing (DFM 2013), Edinburgh, Scotland, September 8, 2013.

# A FIFO by any other name ...

- ❑ All channels in Reo are user defined!
- ❑ What is it about a FIFO that enables partitioning of a circuit into synchronous regions?
  - Automata transitions that can fire involving disjoint subsets of ports
- ❑ Transition in Sync requires consensus of both ports



- ❑ Transitions in FIFO can fire by checking local conditions



# Global vs local product

## □ Global product

- Generally unattainable even at compile time!

## □ Consider the $n$ CA for the $n$ circuit primitives

- Form  $1 \leq m \leq n$  groups

- CA inside a group cannot make transitions independently of each other
  - Useless concurrency
- CA in different groups can make transitions independently of each other
  - Useful concurrency

- Cheap compile-time determination of dependency
- Cheap run-time check for local agreement

# Syntactic subtraction

- ❑ Hiding internal nodes is important optimization
  - It simplifies observable behavior
- ❑ The end-to end observable behavior of a series of Sync channels is identical to that of a single Sync
- ❑ Standard hiding on *CA* yields data constraints that logically hide internal nodes, but do not eliminate them
  - $hide(\{a = b, b = c, c = d, d = e, \dots, y = z\}, \{b, c, d, e, \dots, y\})$
  - $\exists b \exists c \exists d \dots \exists y a = b, b = c, c = d, d = e, \dots, y = z$
  - Checking this constraint is still unnecessarily expensive!
- ❑ Syntactic subtraction eliminates existentially quantified variables and produces a syntactically simplified, logically equivalent data constraint
  - $a = z$



# Commandification



- ❑ Firing a *CA* transition requires checking whether or not its data constraint is satisfied.
- ❑ In general, this requires constraint solving at run-time
- ❑ Using a general purpose constraint solver is expensive!
- ❑ Generate an imperative program to verify the constraint, instead.
- ❑ This is possible for a reasonably expressive data constraint language.

# Queue inference

- Combine single transitions of a normal *CA* into *multi-transitions* of a *multi-CA*
- Three *CA* transitions with synchronization constraints:
  - $\{A, D\}, \{B, D\},$  and  $\{C, D\}$
  - become a single *multi-transition* with  $\{(A \wedge D) \vee (B \wedge D) \vee (C \wedge D)\}$
  - Which simplifies to  $\{(A \vee B \vee C) \wedge D\}$
- Four transitions with synchronization constraints:
  - $\{A, C, E\}, \{A, D, E\}, \{B, C, E\}, \{B, D, E\}$
  - become a single *multi-transition*  $\{(A \wedge C \wedge E) \vee (A \wedge D \wedge E) \vee (B \wedge C \wedge E) \vee (B \wedge D \wedge E)\}$
  - Which simplifies to  $\{(A \vee B) \wedge (C \vee D) \wedge E\}$
- A queue efficiently implements the run-time check for every *v*-group

# Reachable states product

- ❑ Standard CA product operates state by state
  - Intermediate results can grow exponentially, even when the final product is linearly small.
  - Unreachable states remain until the very end
- ❑ Transition by transition product of CA:
  - Starts from the initial state
  - Follows through to reachable states, only
  - Never visits unreachable states

# Benchmark

- Protocol:  $k$ -tuple

- Applications:

- barrier synchronization
- Join part of fork/join
- Etc.

- Three implementations

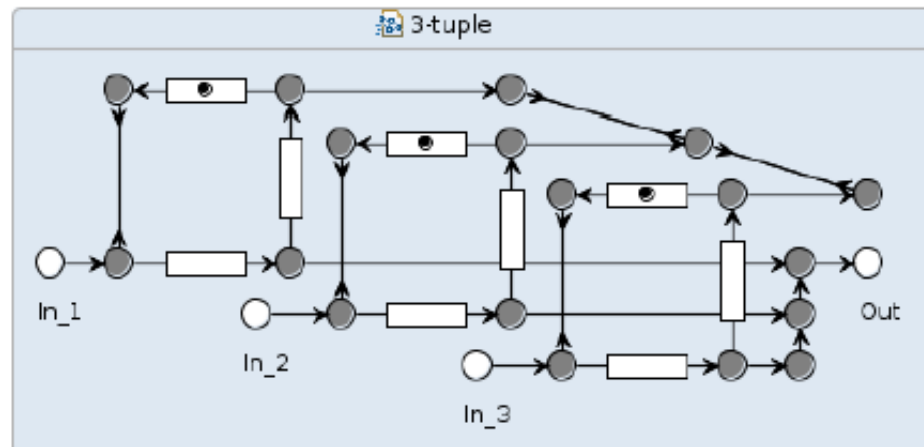
- Reo

- Pthreads conditional variables

- Straight-forward

- Pthreads queue

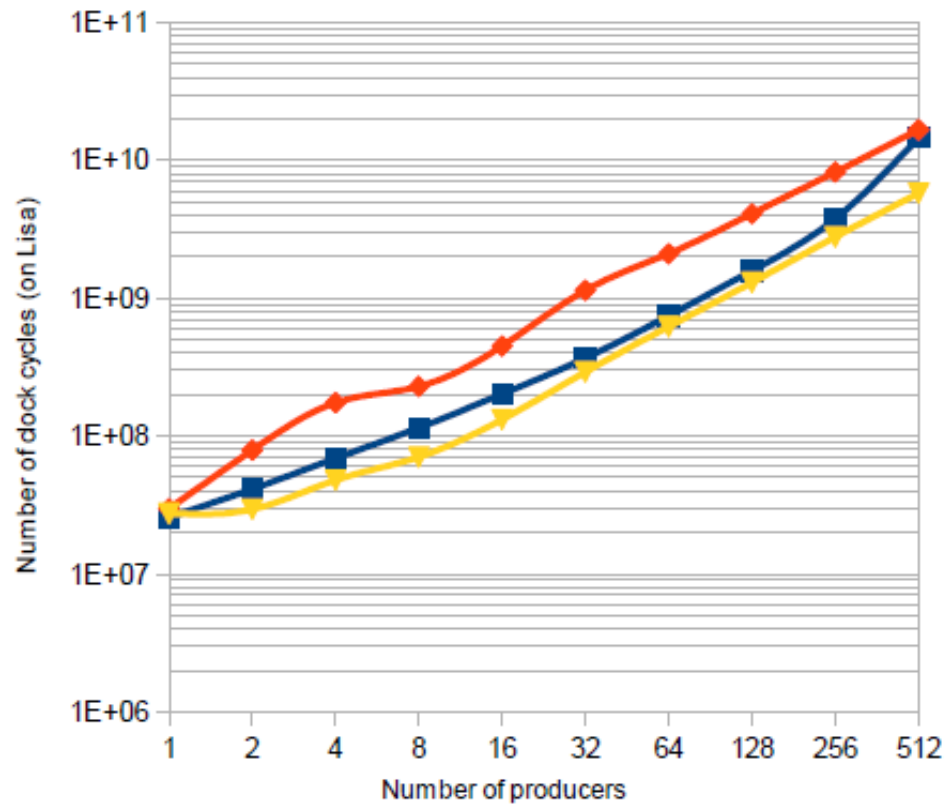
- Application-specific optimization



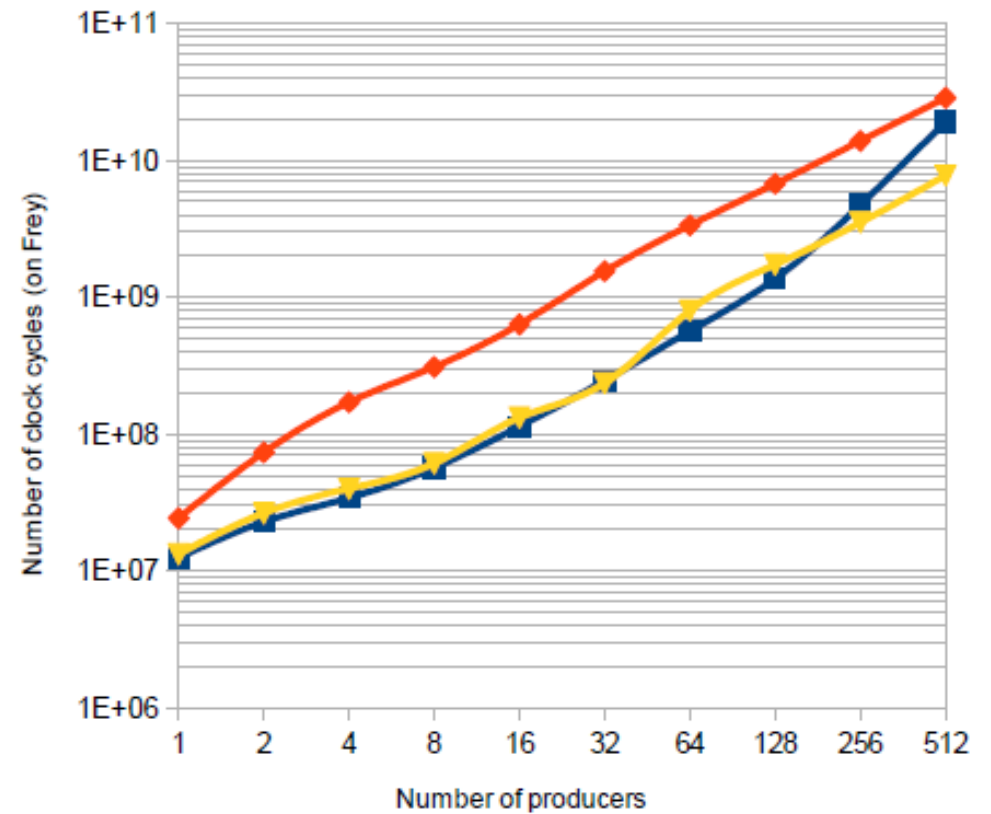
# Performance

■ Reo    ◆ Pthreads-conds    ▲ Pthreads-queue

(a) Legend



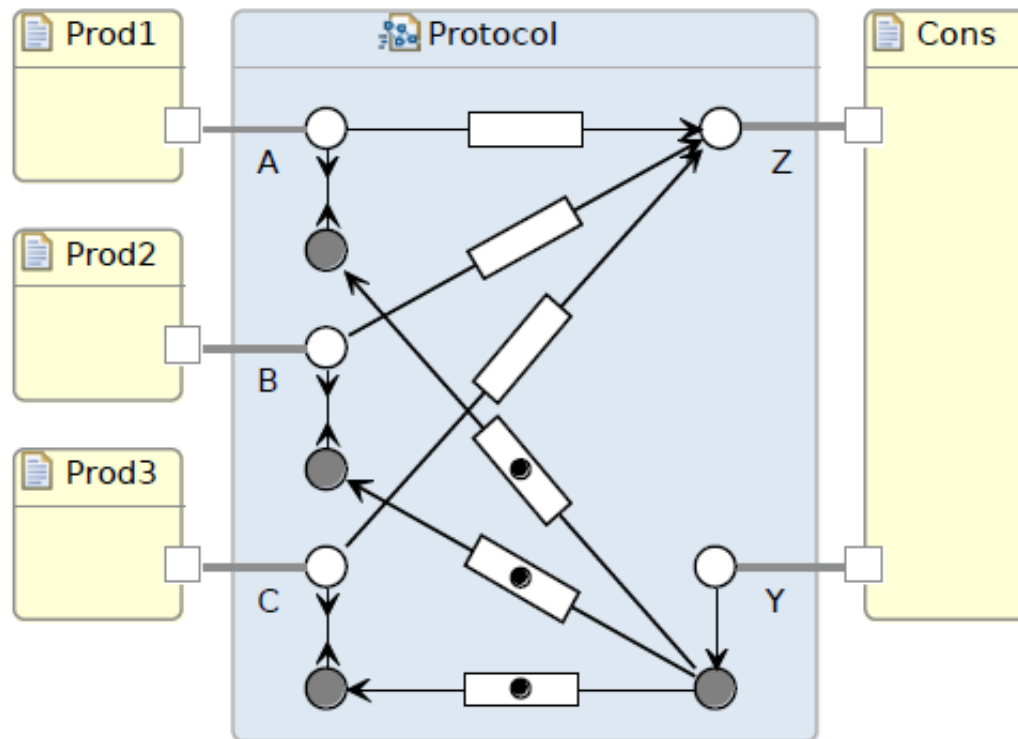
(b) Lisa



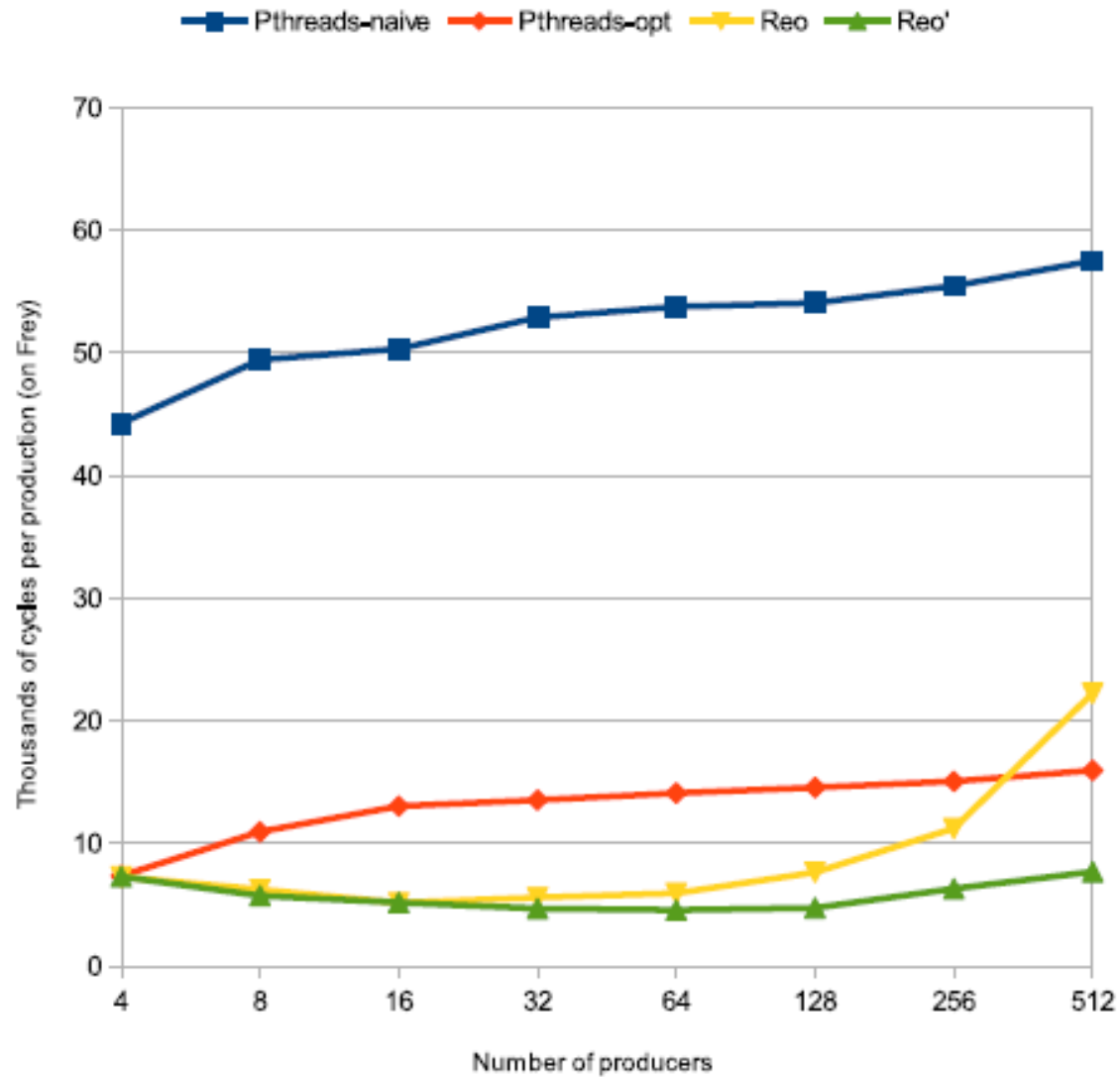
(c) Frey

# Asynchronous bundle merge

- In each cycle, the Consumer receives a bundle of  $n$  items, each produced by one of the producers.



# Compiling Reo



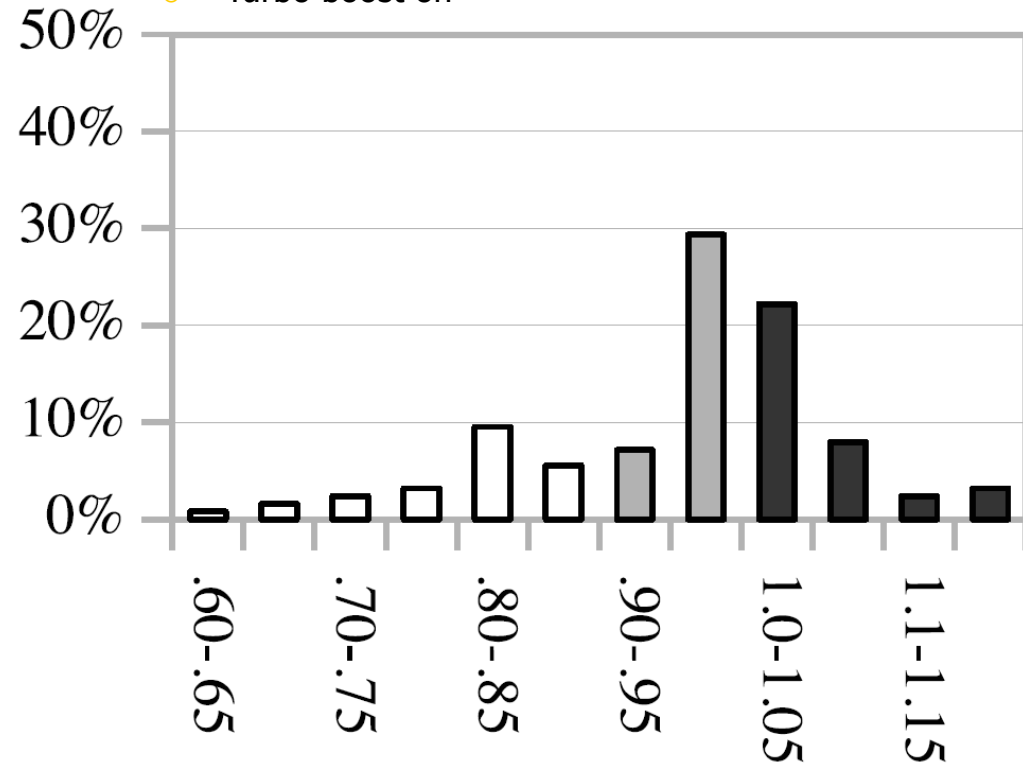
# NASA benchmarks

## Java version of NASA Parallel Benchmarks (NPB)

- 84 full programs
- Reo circuits reused for same protocols in different cases
- Each case ran 5 times
- In 37% of cases generated code no worse than 10% slower
- In 38% of cases generated code is up to 20% faster
- In 25% of cases generated code is between 10% to 40% slower

*Optimization opportunities!*

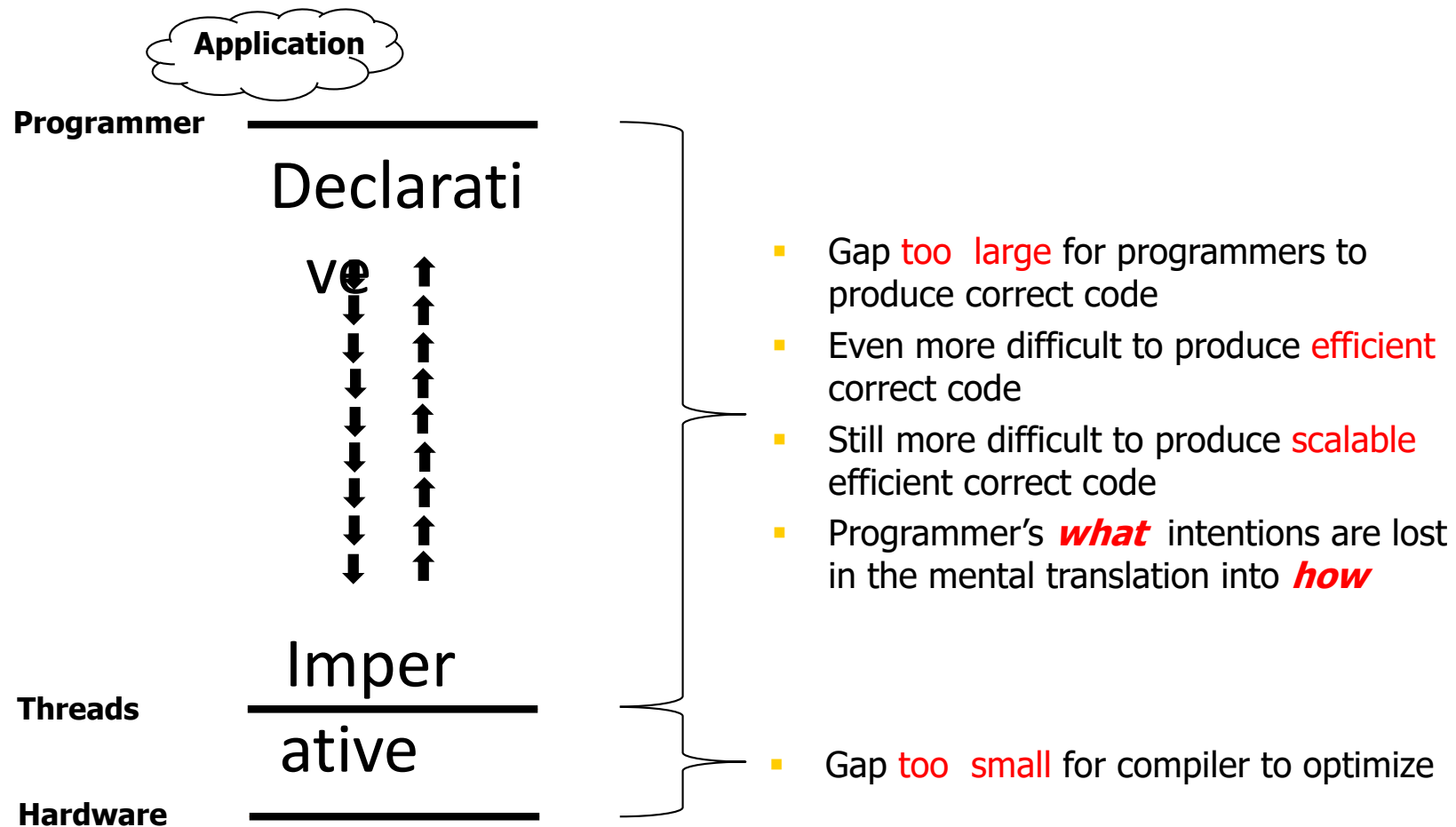
- 24-cores, 2 Intel E5-2690V3 processors in 2 sockets
- Static clock frequency
  - Hyper-threading off
  - Turbo boost off



- Sung-Shik T.Q. Jongmans "Automata-theoretic protocol programming," PhD thesis, Leiden University, 2016, <http://hdl.handle.net/1887/38223>.

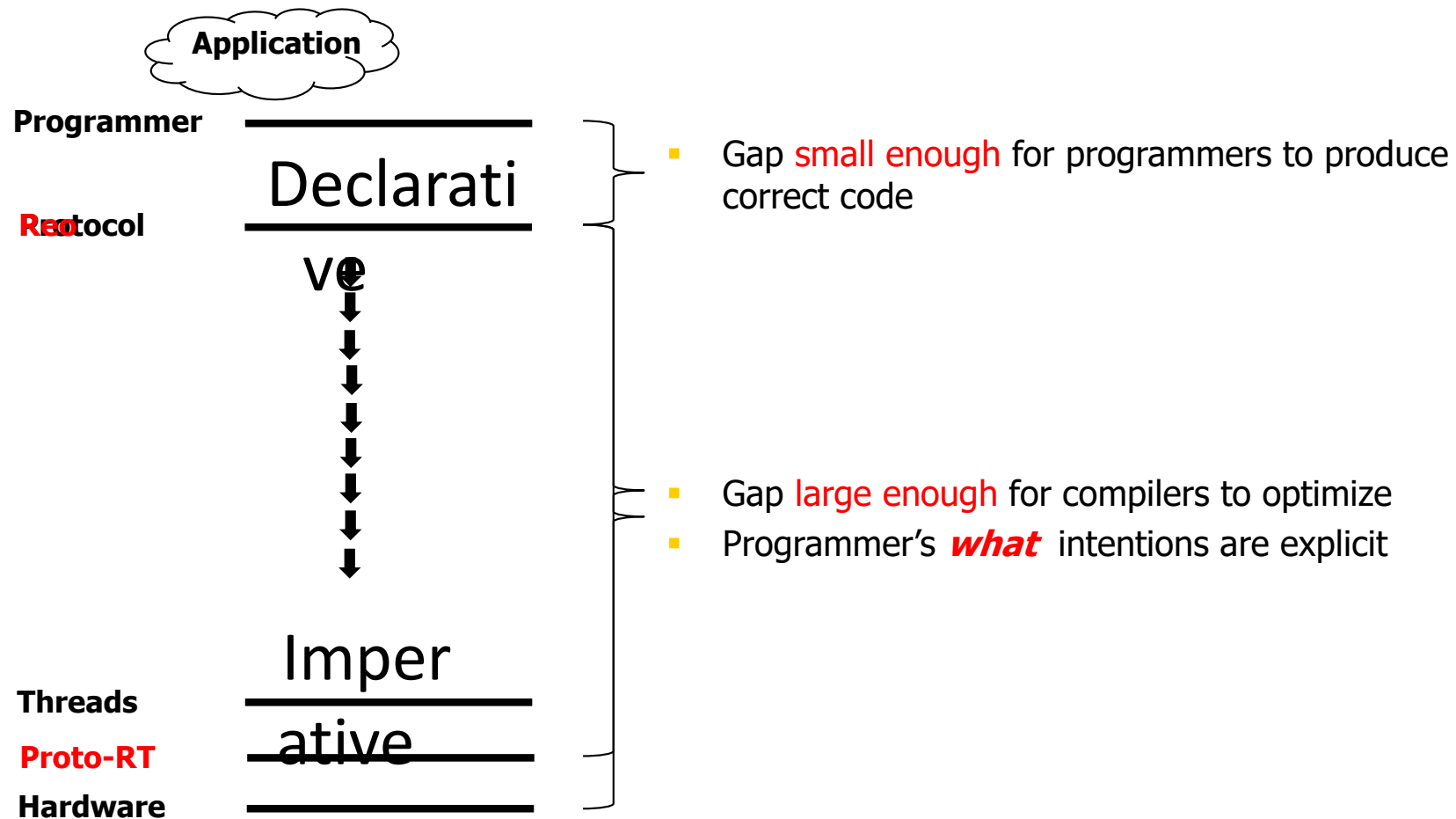


# Typical Concurrent Programming



- Gap **too large** for programmers to produce correct code
  - Even more difficult to produce **efficient** correct code
  - Still more difficult to produce **scalable** efficient correct code
  - Programmer's **what** intentions are lost in the mental translation into **how**
- 
- Gap **too small** for compiler to optimize

# Better Concurrent Programming



# A 1-out-of-n protocol

- Output 1 item out of items from n input ports & repeat.
  - Which item?
    - Any one?
    - The first/last arriving one?
    - A specific one? Which one? In temporal order? In structural order?
  - How to handle excess input from the same source in a cycle?
    - Delay it for next cycle?
    - Lose it?
  - When should output become available?
    - As soon as available?
    - At the end of a cycle?
  - When does a cycle end?
    - After one input from each source?
    - Once the output is taken?
- Generalize 1-out-of-n to k-out-of-n

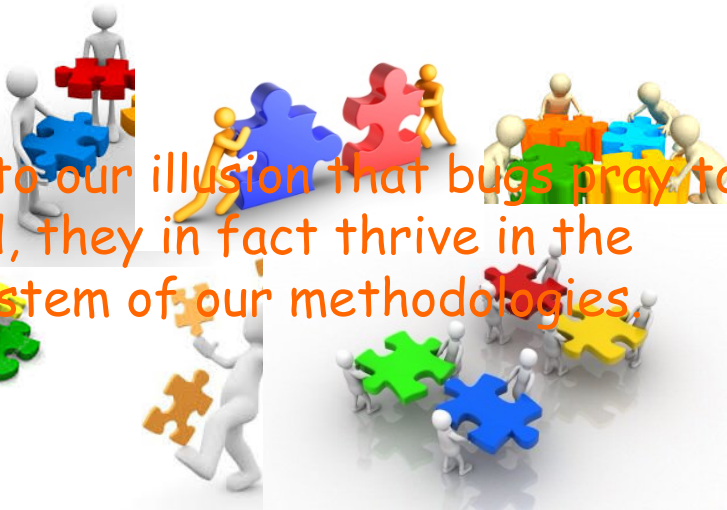
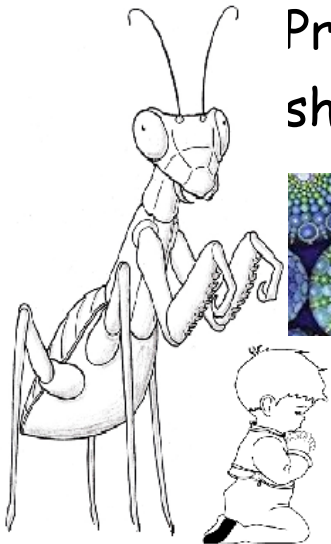
# Protocol programming example

- A 1-out-of-n protocol:
  - Output 1 item out of items from n input ports & repeat.

# Protocol programming

- Action-centric programming of an interaction:
  - Smash interaction on the solid rock of **action**
  - Let each process/thread pick up some interaction-shards.
  - Pray that:
    - No shards go missing or get lost

Processes will independently pick up and flip over just the right shard at the right time to reconstitute the original interaction.



Contrary to our illusion that bugs pray to spread, they in fact thrive in the ecosystem of our methodologies



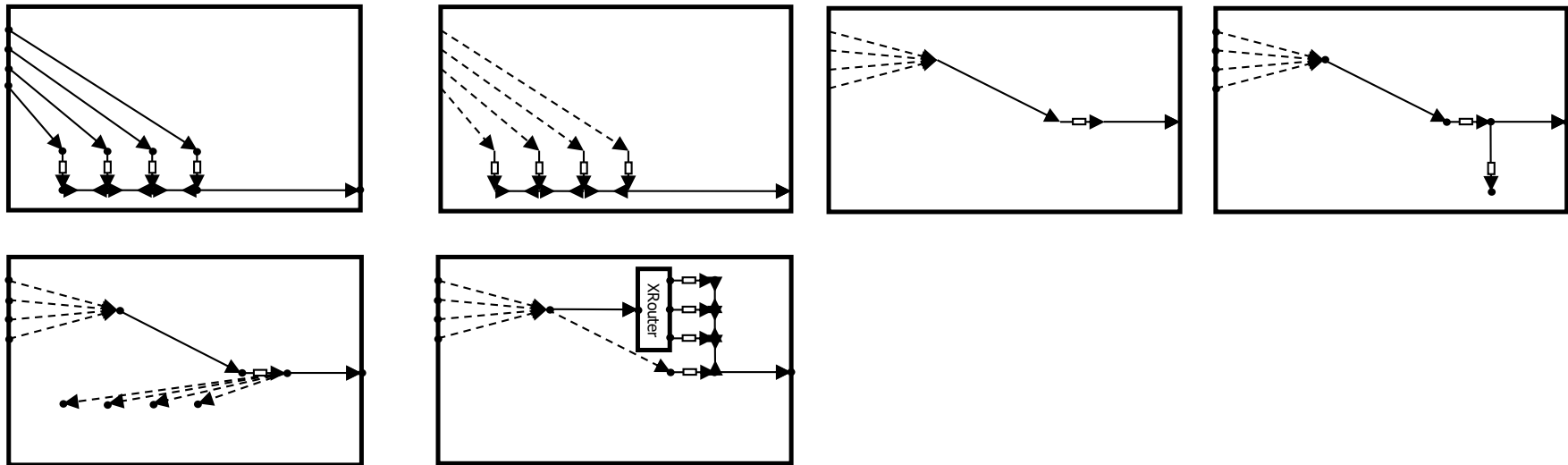
# Protocol programming

- Interaction-centric programming of an interaction:
  - Separation of concerns.
    - Nature: ultimate machinery employing separation of concerns.
  - Nature manifests magnificently complex forms and behaviors by bridling simple unintelligent actions of independent actors, ignorant of those emerging patterns, with superimposition of easy constraints.
  - Consider the protocol as manifestation of a constraint.
  - Decompose the constraint into simple, down to easy constraints.
  - Superimpose constraints through mathematical composition of relations.



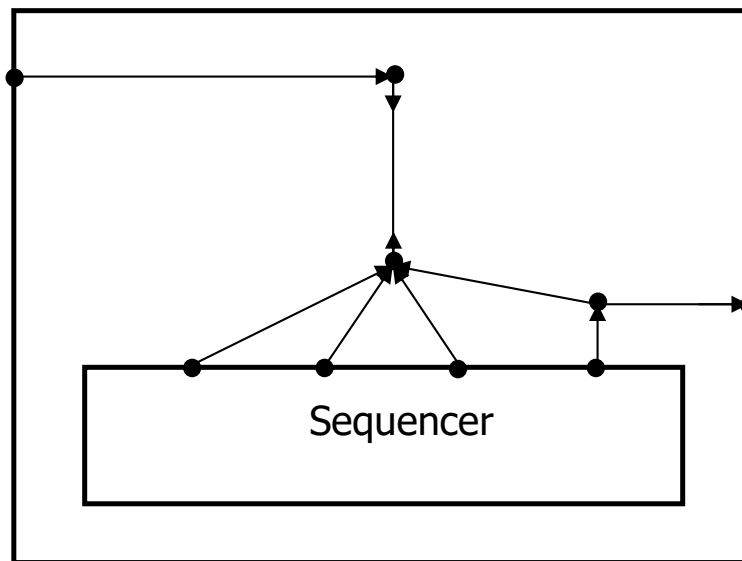
# Interaction programming

- Decompose a protocol into simpler protocols.
- Compose the original protocol by superimposition of simpler protocols.
- Some simple sub-protocols for k-out-of-n:



# n-Counter

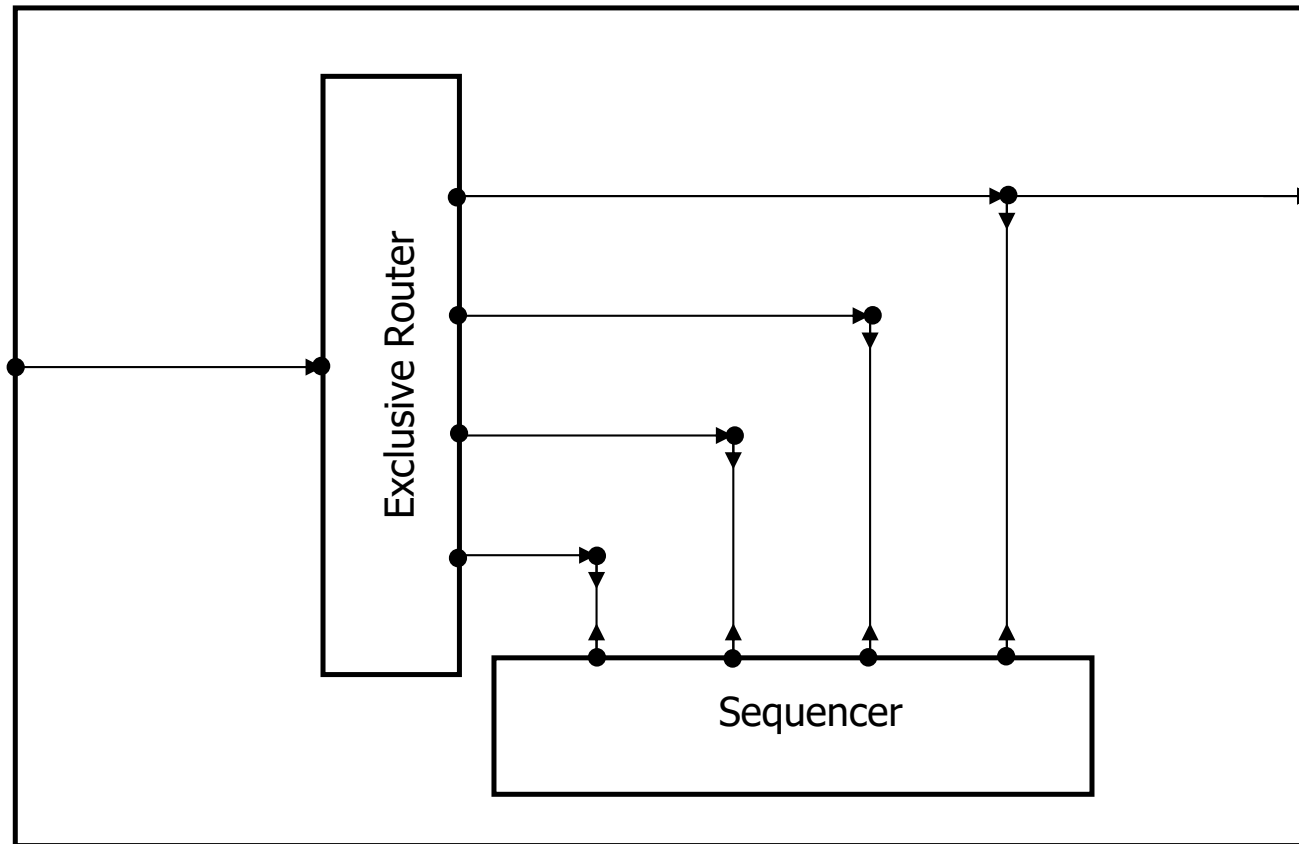
- Makes a token available on its output upon the availability of every 4<sup>th</sup> input data item.





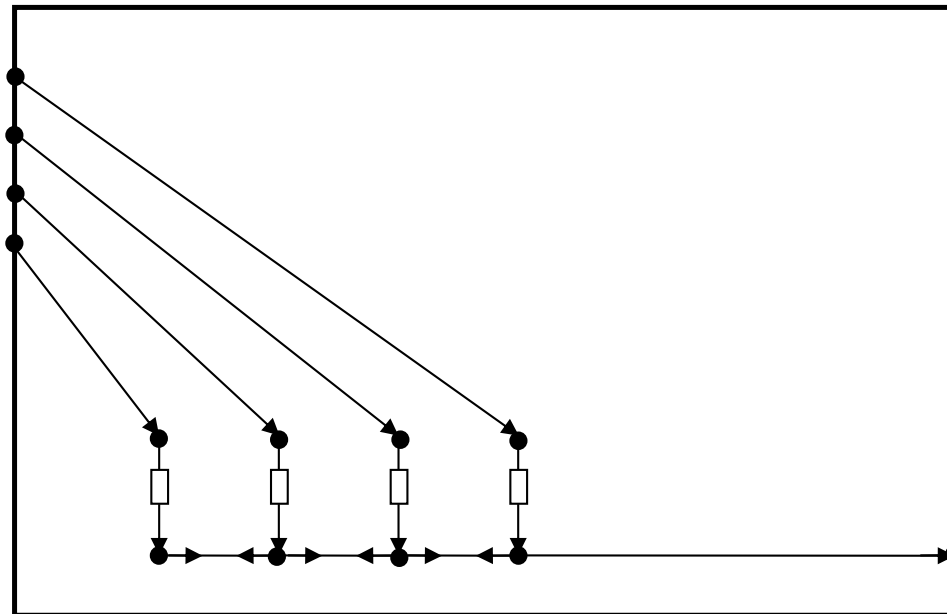
# $n^{\text{th}}$ -Filter

- Passes every 4<sup>th</sup> input item.



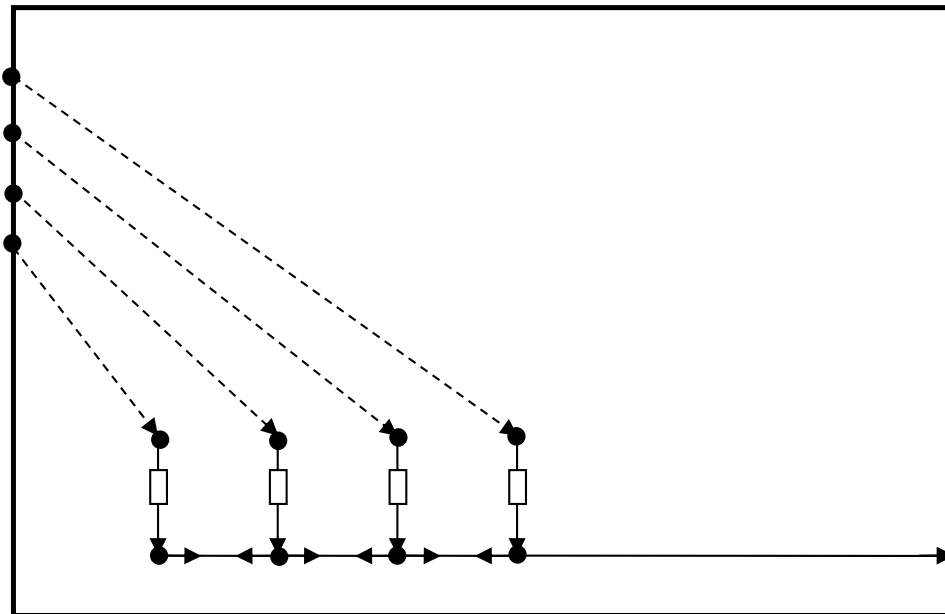
# Sparing Delayed 1-out-of- $n$

- Outputs **one** of the  $n$  input values in each cycle.
- Output is **delayed** until the end of a cycle.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are **spared** for the next cycle.



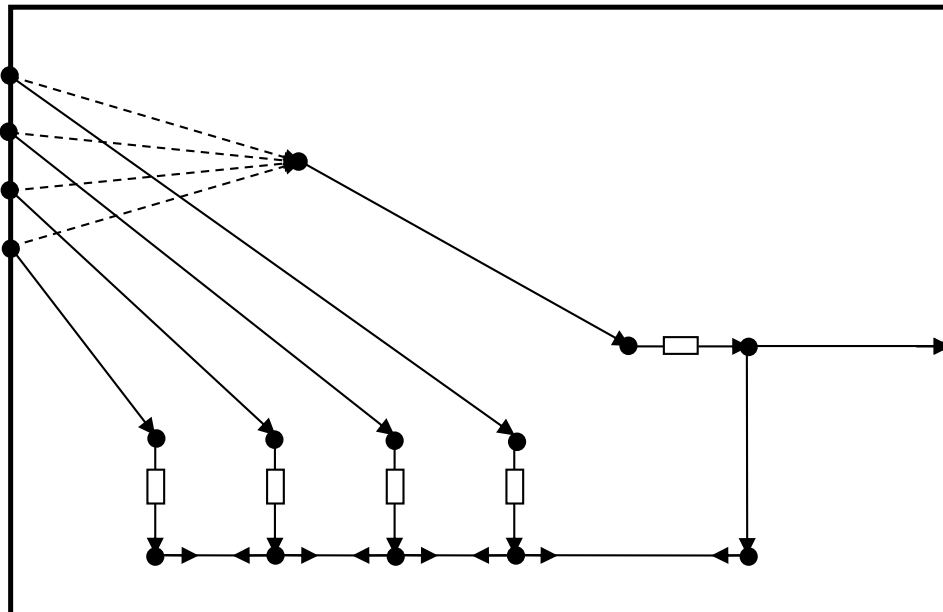
# Lossy Delayed 1-out-of- $n$

- Outputs **one** of the  $n$  input values in each cycle.
- Output is **delayed** until the end of a cycle.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are **lost** as they arrive.



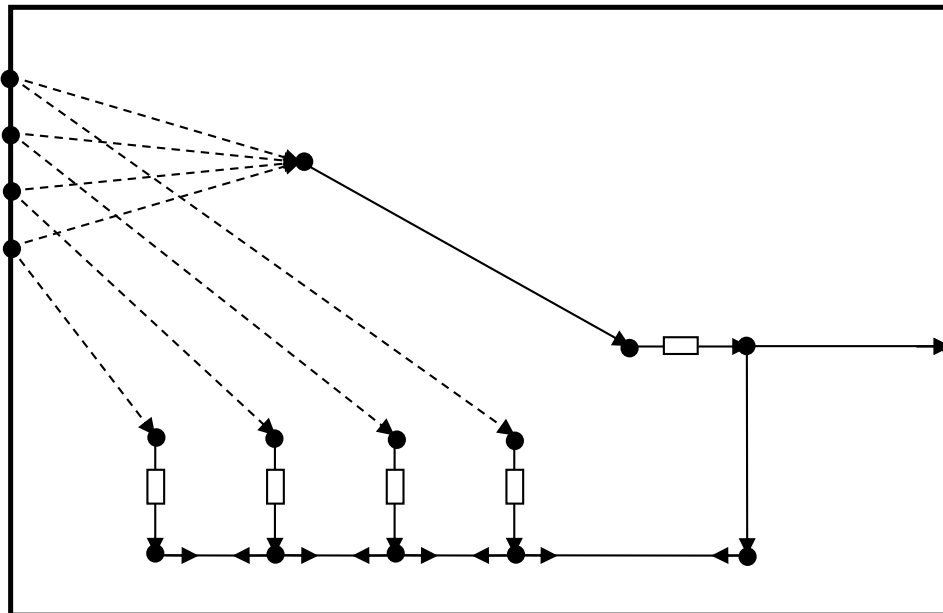
# Sparing Delayed 1<sup>st</sup> Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is **delayed** until the end of each cycle.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are **spared** for the next cycle.



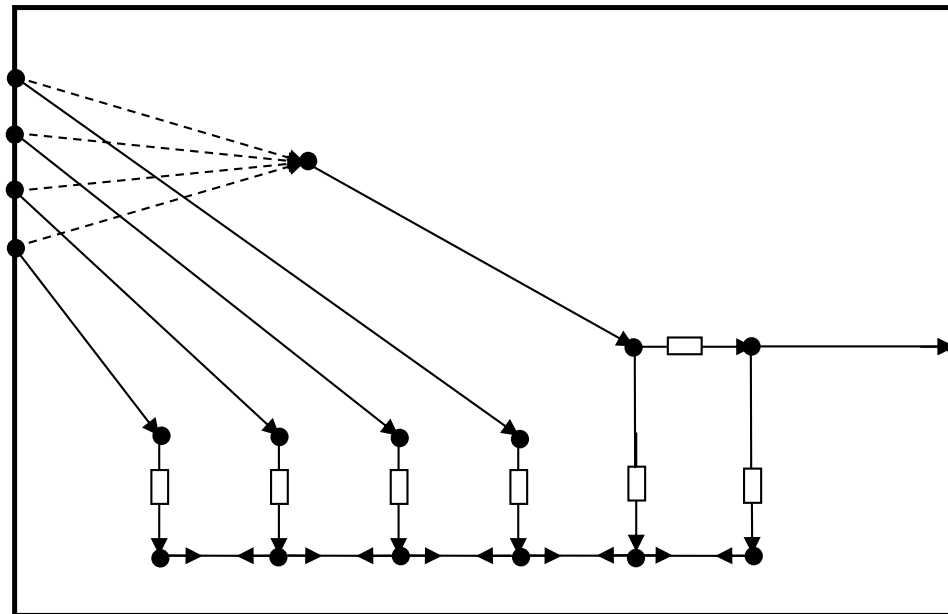
# Lossy Delayed 1<sup>st</sup> Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is **delayed** until the end of each cycle.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are **lost** as they arrive.



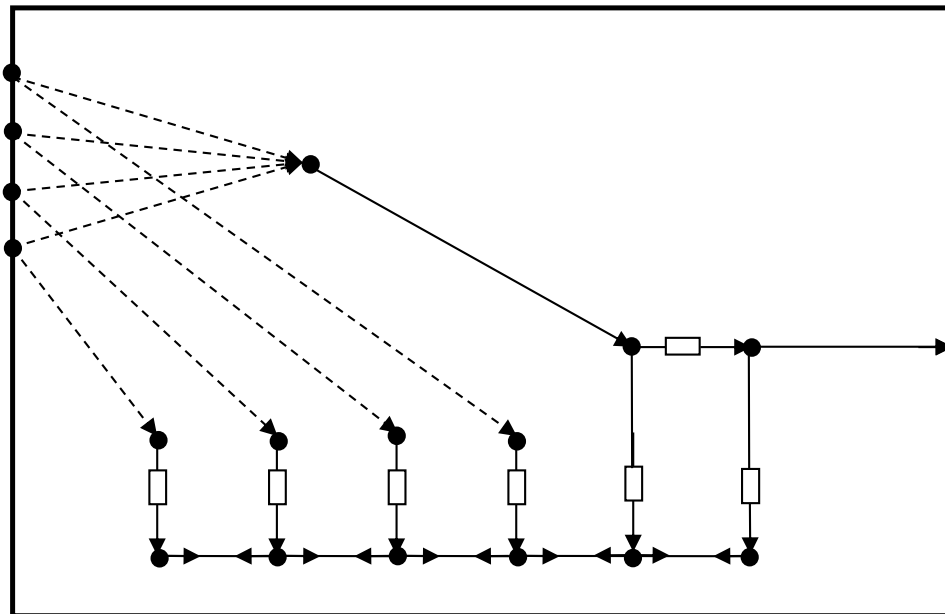
# Sparing Prompt 1<sup>st</sup> Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is possible **prompt** after the first input arrives.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are **spared** for the next cycle.



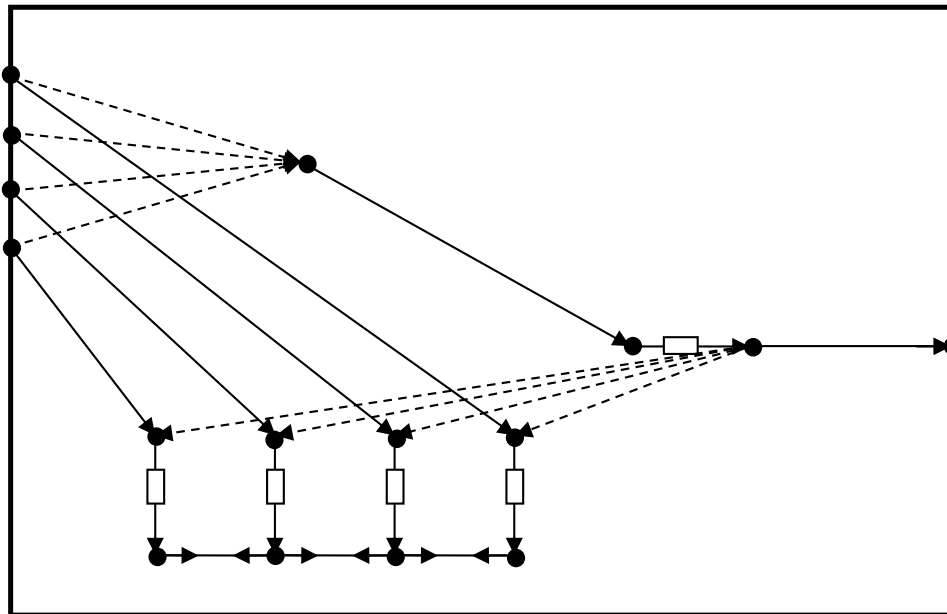
# Lossy Prompt 1<sup>st</sup> Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is possible **prompt** after the first input arrives.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are **lost** as they arrive.



# Sparing Forced 1<sup>st</sup> Out of $n$

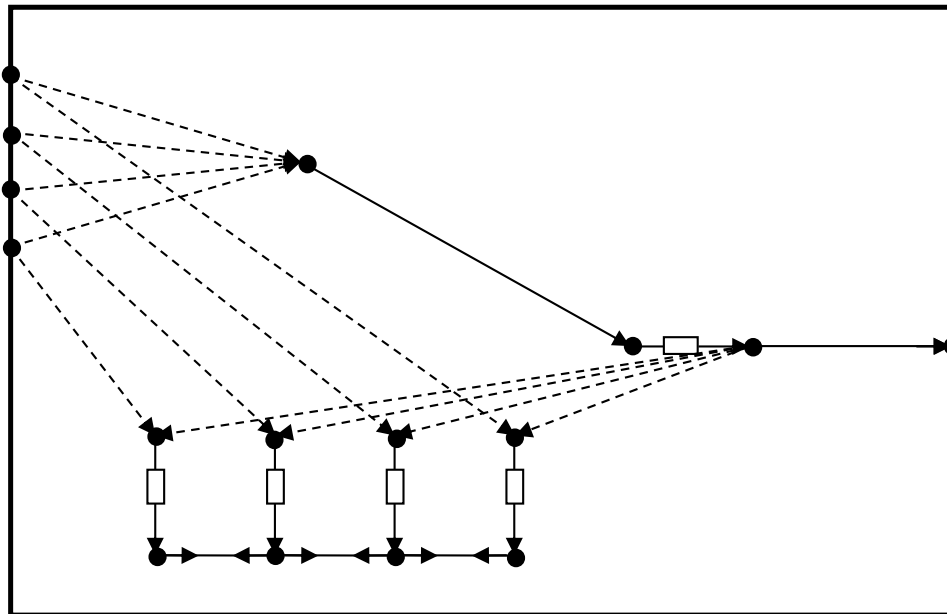
- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle is **forced** to end after a value is taken from the output.
- Extra input values of a node are **spared** for the next cycle.





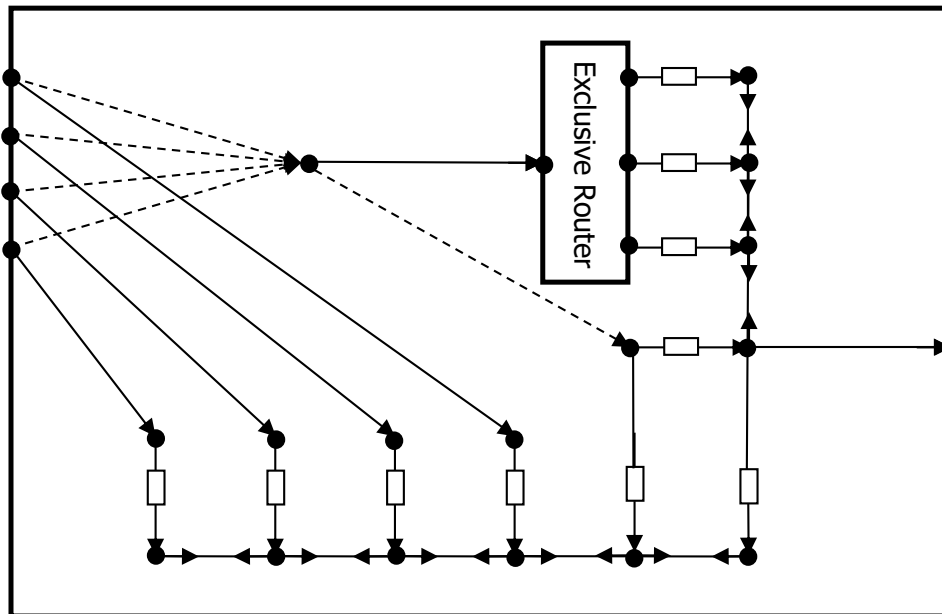
# Lossy Forced 1<sup>st</sup> Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle is **forced** to end after a value is taken from the output.
- Extra input values of a node are **lost** as they arrive.



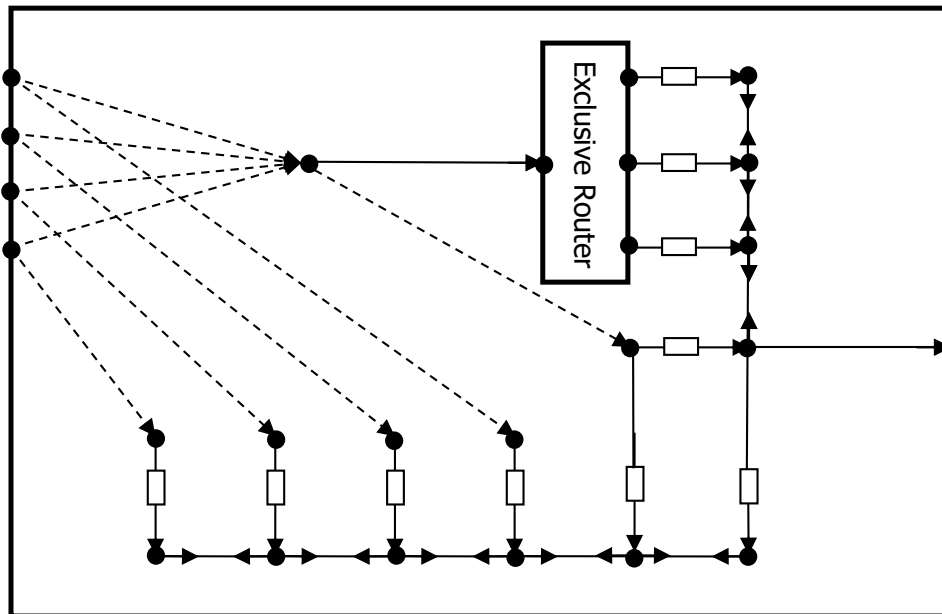
# Sparing Prompt $m$ Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is possible **prompt** after the first  $m$  input values arrive.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are **spared** for the next cycle.



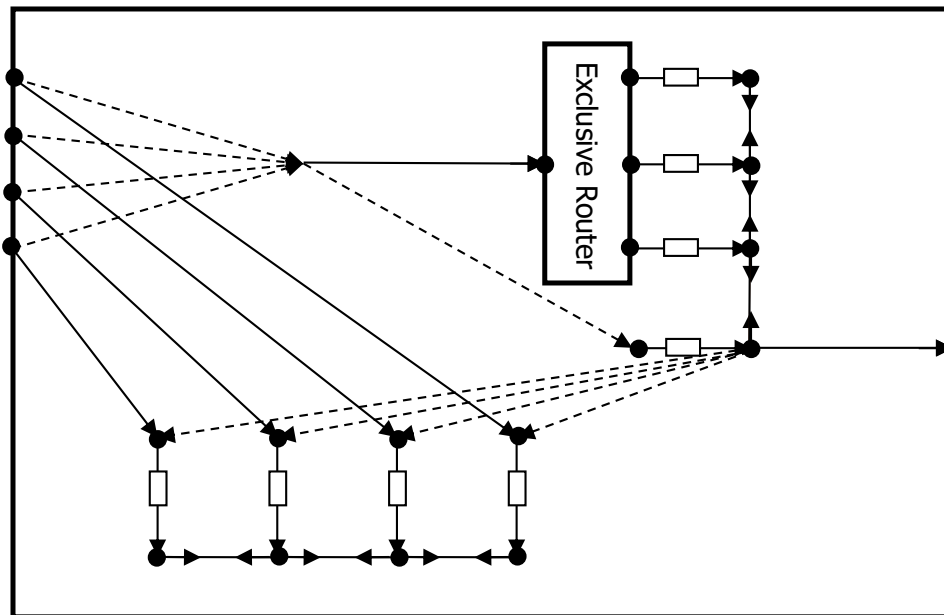
# Lossy Prompt $m$ Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is possible **prompt** after the first  $m$  input values arrive.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are **lost** as they arrive.



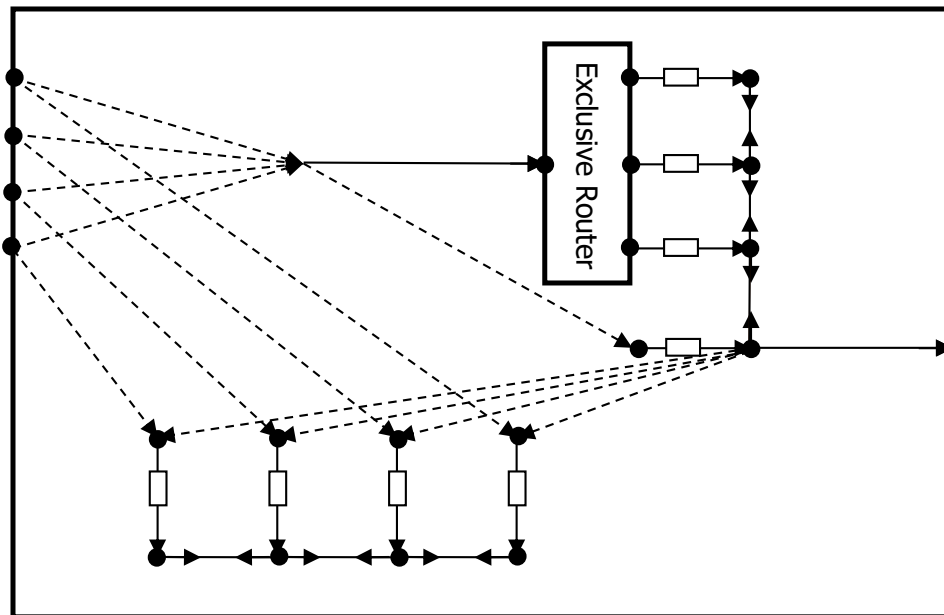
# Sparing Forced $m$ Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle is **forced** to end after a value is taken from the output.
- Extra input values of a node are **spared** for the next cycle.



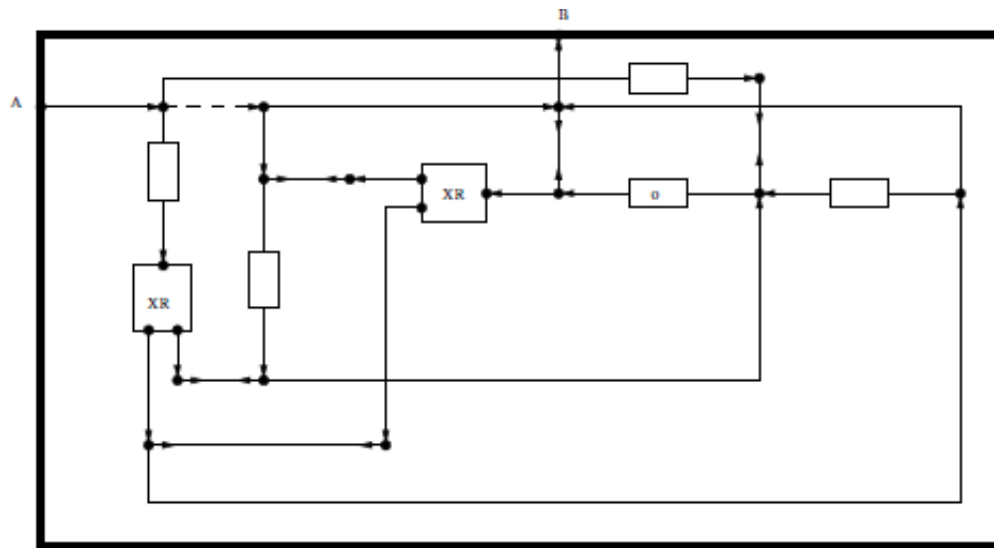
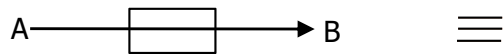
# Lossy Forced $m$ Out of $n$

- Outputs only the **first** of the  $n$  arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle is **forced** to end after a value is taken from the output.
- Extra input values of a node are **lost** as they arrive.

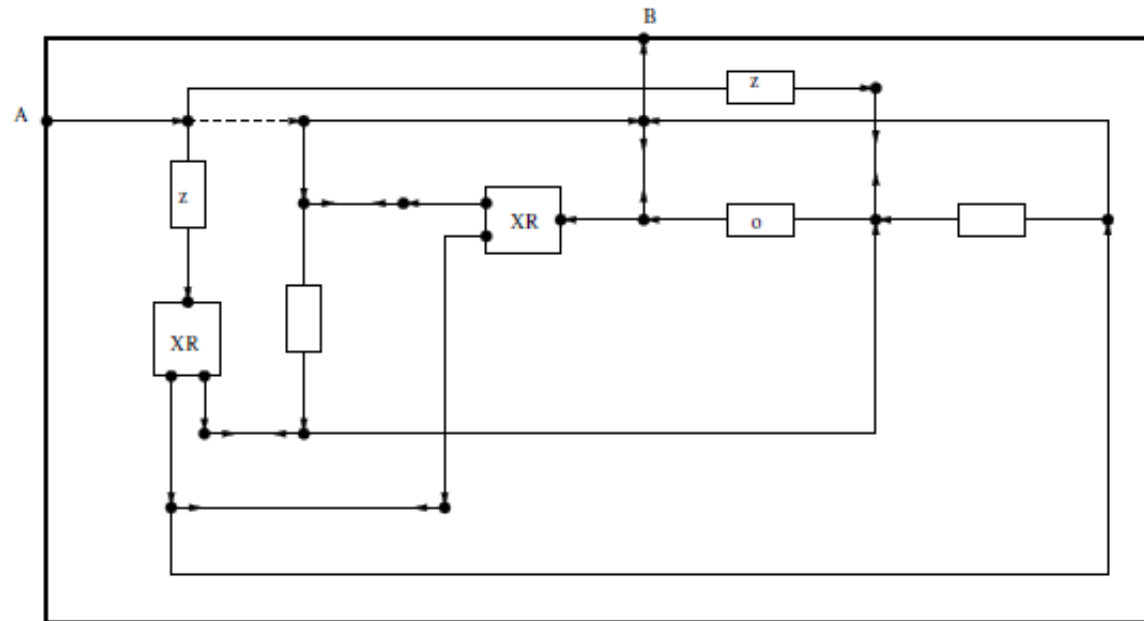


# Synchronous FIFO1

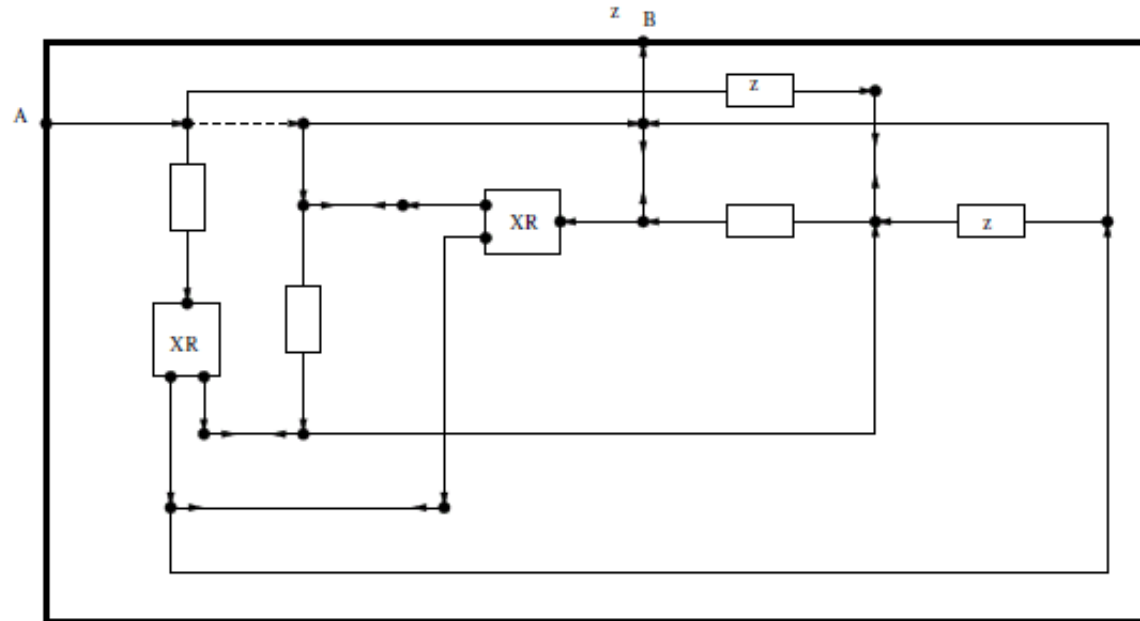
- ❑ Combines the behavior of Sync and FIFO1.
- ❑ Behaves as a FIFO1, except that if the buffer is empty, and
  - a take is pending on B,
  - the value written to A synchronously goes to B and leaves the buffer empty.



# Write $z$ to SyncFIFO1

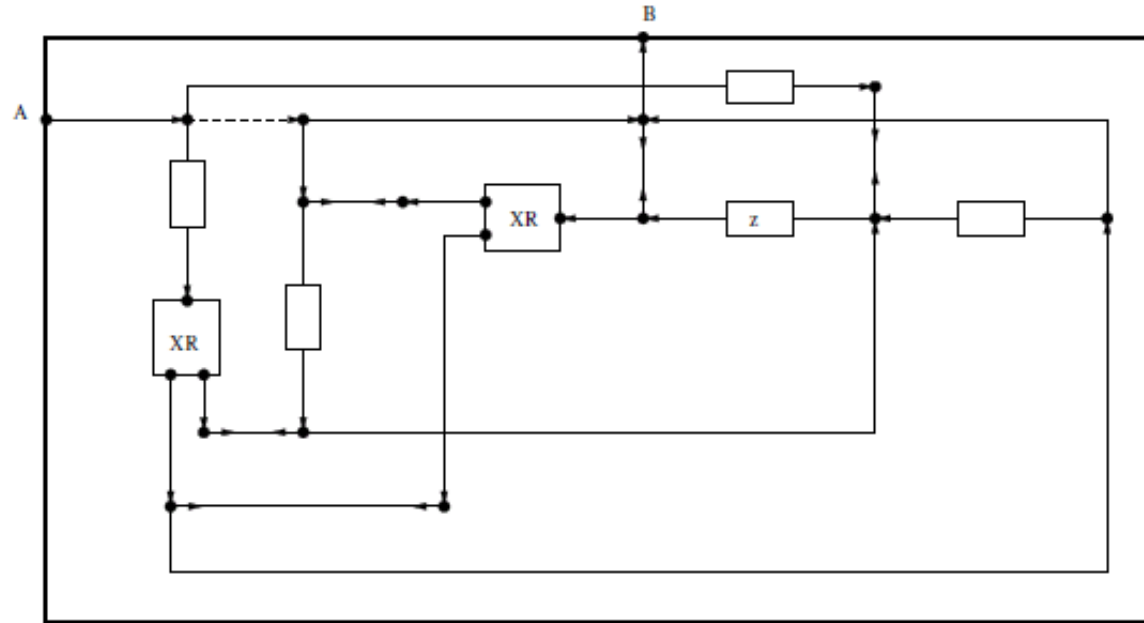


# Take $z$ from SyncFIFO1

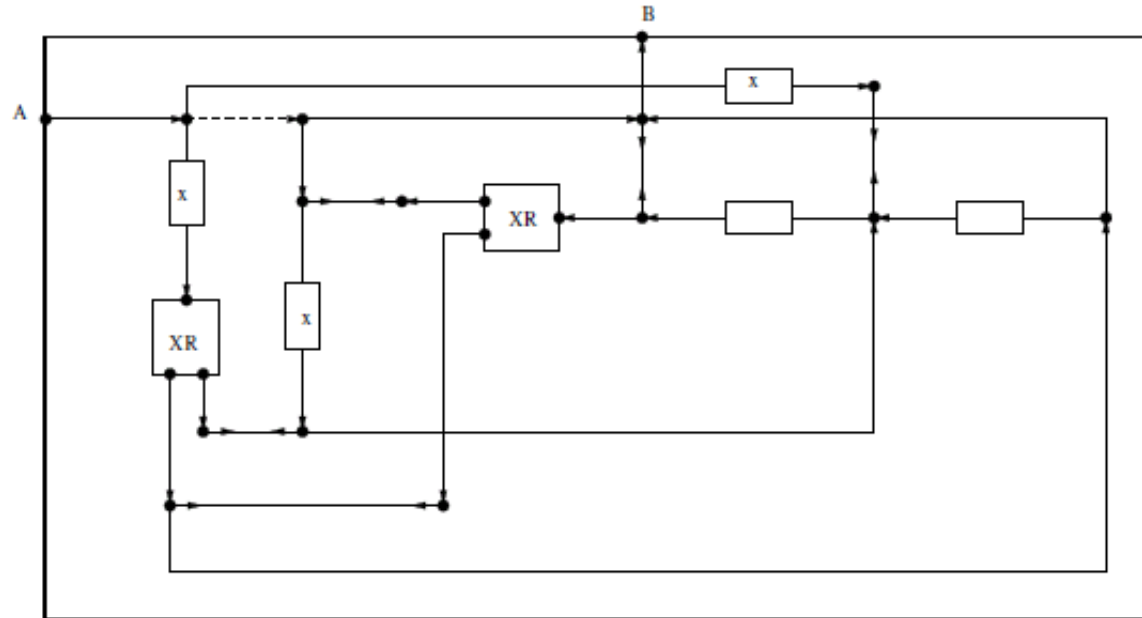




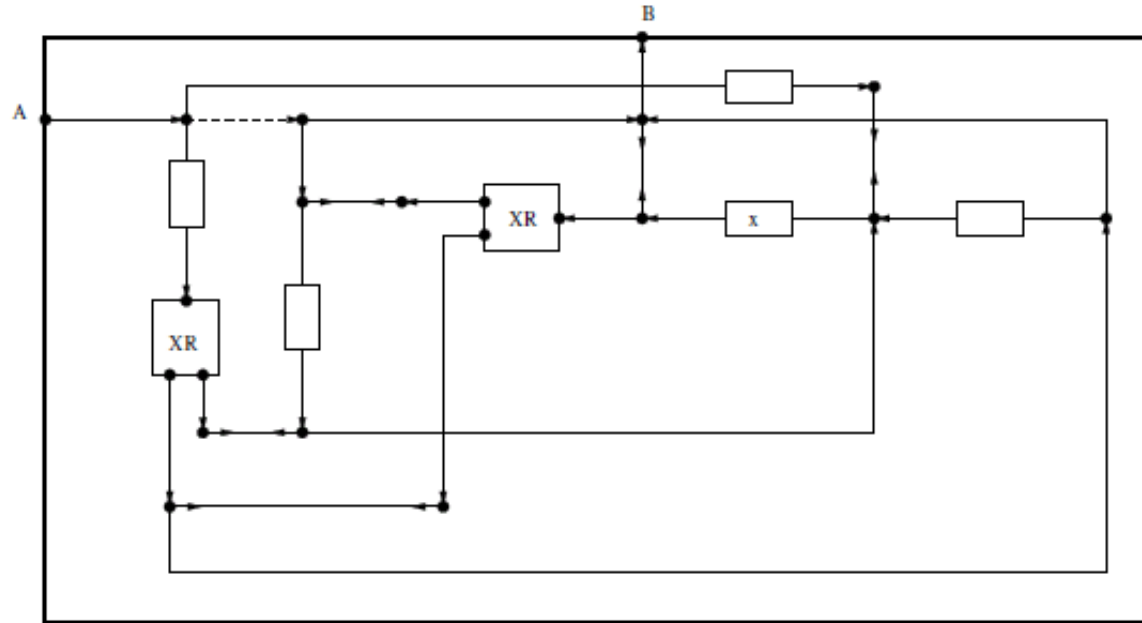
# SyncFIFO1 Resets Itself



# Write After a Pending Take

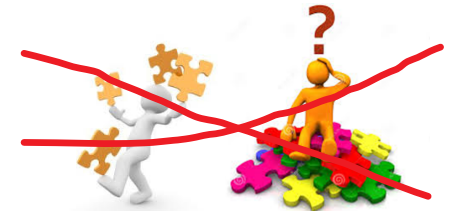


# After Synchronous Write/Take



# What are you doing the rest of your life?

- ❑ As the exponentially complex aspect of concurrency, interaction protocols become simpler to construct, validate, compose, and reuse as first-class entities.
- ❑ Interaction-centric programming needs programming constructs for:
  - Explicit formal representation
  - Direct composition
- ❑ Reo is a simple, rich, versatile, and surprisingly expressive language for compositional construction of pure interaction protocols.
  - Treats interaction as (the only) first-class concept.
  - Free combination of synchrony, exclusion, and asynchrony.
  - Offers direct composition and verbatim reuse of protocols.



<http://reo.project.cwi.nl>

# Conclusions



- ❑ As the exponentially complex aspect of concurrency, interaction protocols become simpler to construct, validate, compose, and reuse as first-class entities.
- ❑ Interaction-centric programming needs programming constructs for:
  - Explicit formal representation
  - Direct composition
- ❑ Reo is a simple, rich, versatile, and surprisingly expressive language for compositional construction of pure interaction protocols.
  - Treats interaction as (the only) first-class concept.
  - Free combination of synchrony, exclusion, and asynchrony.
  - Offers direct composition and verbatim reuse of protocols.

<http://reo.project.cwi.nl>