Coordinating Multicore Computing

International School on Formal Methods for the Design of Computer, Communication, and Software Systems

(SFM-15:MP)

Bertinoro, Italy

Farhad Arbab

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

June 16, 2015

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.

Models of Concurrency

- Traditional models are action based
  - Petri nets
  - Work flow / Data flow
  - Process algebra / calculi
  - Actor models / Agents
  - …
- In prominent models, a system is composed from building blocks that represent actions/processes
- Interaction becomes an implicit side-effect
  - Makes coordination of interaction more difficult to
    - Specify
    - Verify
    - Manipulate
    - Reuse
- Composition of actions ***does not*** yield composition of interaction!

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**

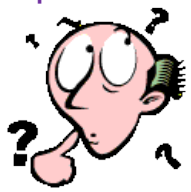
```
while (true) {
  sleep (4000);
  bufferSemaphore.acquire();
  if (buffer != EMPTY) {
    println(buffer);
    buffer = EMPTY;
  }
  bufferSemaphore.release();
}
```

- **Producers**

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

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



Process Algebraic Model

- **Shared entities**

  synchronization-points: g, r, b, d

- **Consumer**

  B := ?b(t) . print(t) . !d("done") . B

- **Producers**

  G := ?g(k) . genG(t) . !b(t) . ?d(j) . !r(k) . G

  R := ?r(k) . genR(t) . !b(t) . ?d(j) . !g(k) . R

- **Model**

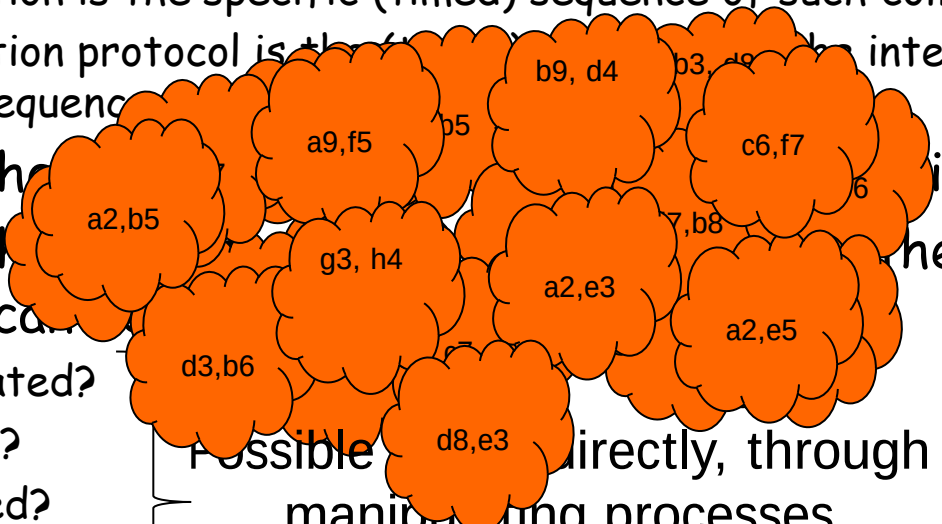
  G | R | B | !g("token")

- **What are the primitives and constructs in this model?**
  - **Shared names to synchronize communication:**
    - **g, r, b, d**
  - **Atomic actions:**
    - **Primitive actions defined by algebra:**
      - **?_(_), !_(_)**
    - **User-defined actions:**
      - **genG(_), genR(_), print(_)**
  - **Composition operators:**
    - **., |, +, :=, implied recursion**

- **A model is constructed by composing (atomic) actions into (more complex) actions.**
- **Primarily a model of actions/processes**
  - **Hence the name "process algebra"**
- **Where is interaction?**

Implicit Interaction

- Interaction (protocol) is implicit in action-based models of concurrency
- Interaction is a by-product of processes executing their actions
  - Action ai of process A collides with action bj of process B
  - Interaction is the specific (timed) sequence of such collisions in a run
  - Interaction protocol is th… intended collisions in such a sequenc…
- How can th… ifferentiated?
- How can t… he interaction protocol) ca…
  - Manipulated?
  - Verified?
  - Debugged?
  - Reused ?
  - …



Possible … irectly, through manipulating processes

Interaction protocols

- Specification, analysis, construction, and application of interaction protocols have been studied in concurrency theory for decades!
- Yet, we have not treated interaction protocols with the seriousness that they deserve!
  - Primarily, we have not considered protocols as first-class concepts.
  - We do not construct protocols by composing protocols!

Construction of artifacts

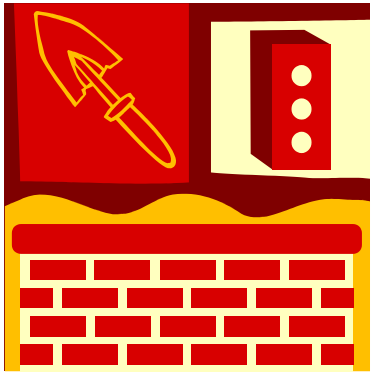
- Direct methods
  - The desired artifact is constructed by composing smaller pieces of that same artifact.
  - Artifact properties more likely to correspond compositionally to those of its parts.
  - Simpler specification, analysis, and construction.
- Indirect methods
  - The desired artifact is the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

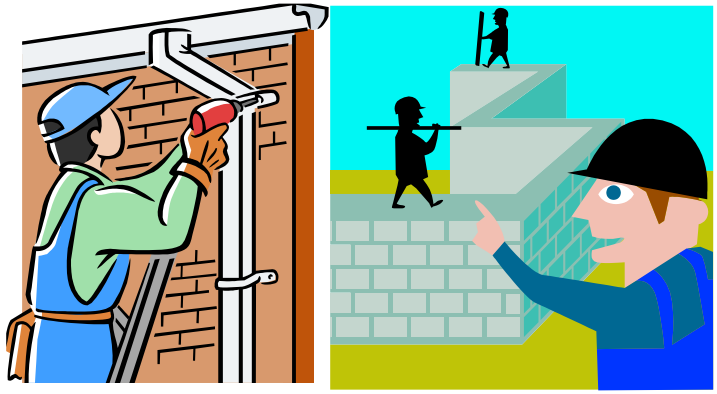
Desired Artifact

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).

Concurrent software

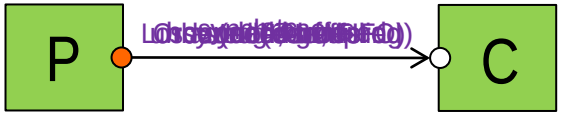
- Interaction protocol is the quintessential ingredient of concurrent software.
- The discourse in traditional models of concurrency consists of *processes/actors*, not *interaction*.
  - Petri nets
  - Work flow / Data flow
  - Process algebra / calculi; thread programming
  - Actor models; Agents; active objects
  - They model *things that interact*, not *interaction*!
- Interaction becomes an implicit side-effect
  - More difficult to specify, verify, manipulate, and/or reuse

What Are Protocols Made of?

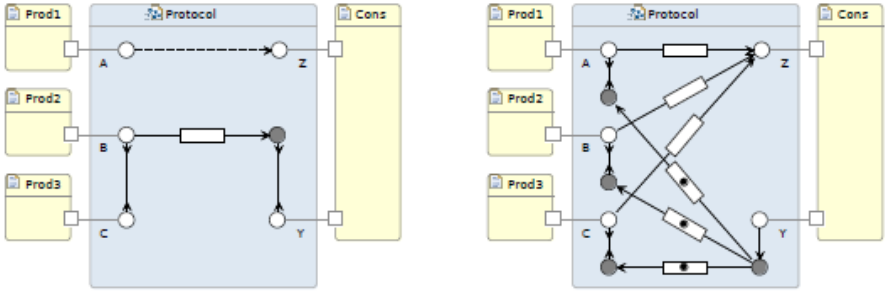
- Protocols express relationships among actions performed by actors that convey:
  - Synchrony / asynchrony
  - Atomicity
  - Ordering
  - Exclusion
  - Grouping
  - Selection
  - …
- How to formalize interaction explicitly?
  - **Constraints**

Interaction Based Concurrency

- Start with a set of primitive interactions as binary constraints
- Define (constraint) composition operators to combine interactions into more complex interactions
- black-box processes:
  - Offer no inter-process methods nor make such calls
  - Do not send/receive targeted messages
  - Communicate exclusively 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)
  - Composing processes with different connectors (that impose different protocols from outside) constructs different systems.



Reo



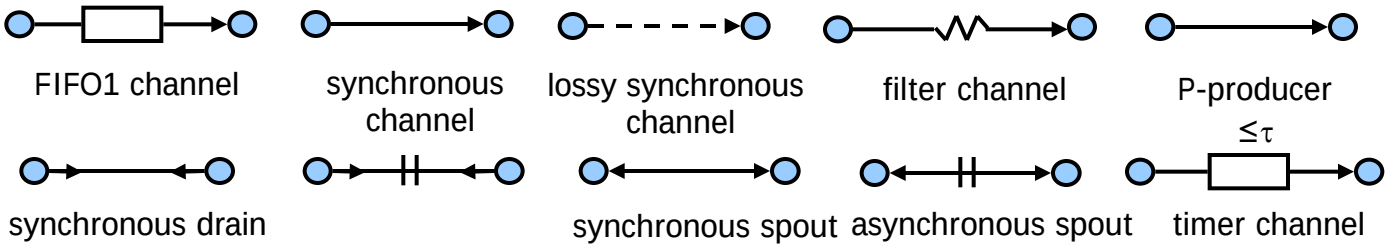
- Reo is a language for compositional construction of interaction protocols.
- Interaction is the only first-class concept in Reo:
  - Explicit constructs representing interaction
  - Composition operators over interaction constructs
- Protocols manifest as a connectors
- In its graphical syntax, connectors 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

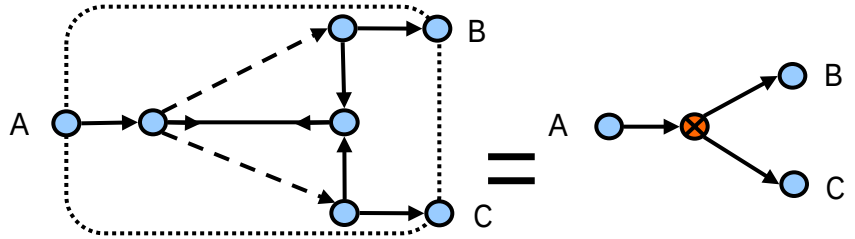
- F. Arbab "**Puff, The Magic Protocol**," Formal Modeling: Actors, Open Systems, Biological Systems 2011, SRI International, Menlo Park, California, November 3-4, 2011, Lecture Notes in Computer Science, Springer, vol. 7000, pp. 169-206, 2011.
- Farhad Arbab, "**Reo: A Channel-based Coordination Model for Component Composition**," *Mathematical Structures in Computer Science*, Cambridge University Press, Vol. 14, Issue 3, pp. 329-366, June 2004.

Channels

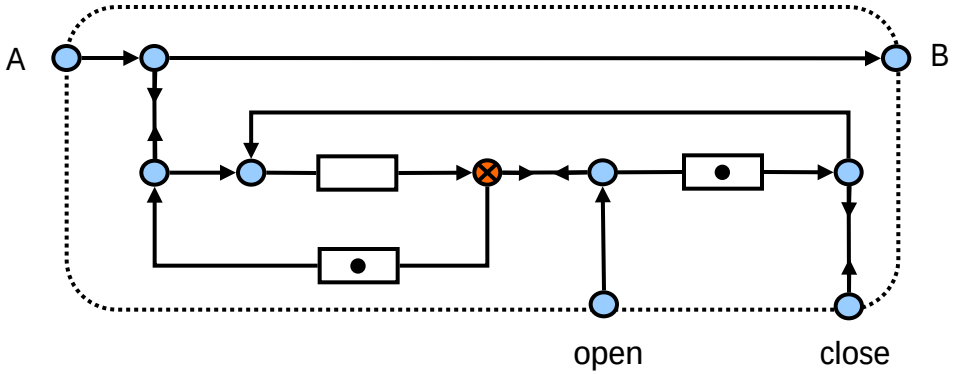
- Atomic connectors in Reo are called *channels*.
- Reo generalizes the common notion of channel.
- A channel is an abstract communication medium with:
  - exactly two ends; and
  - a constraint that relates (the flows of data at) its ends.
- Two types of channel ends
  - Source: data enters into the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

Reo Connectors





Exclusive choice (deffered XOR)

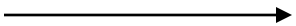


Valve connector:
controls flow from A to B

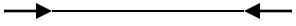
asynchronous drain

A Sample of Channels

- Synchronous channel
  - write/take



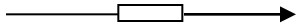
- Synchronous drain: two sources
  - write/write



- Synchronous spout: two sinks
  - take/take

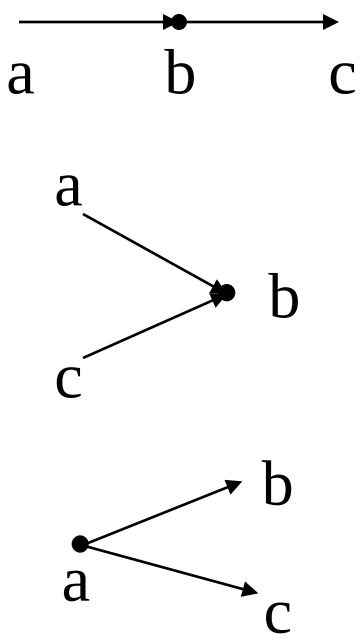
- Lossy synchronous channel

- Asynchronous FIFO1 channel
  - write/take



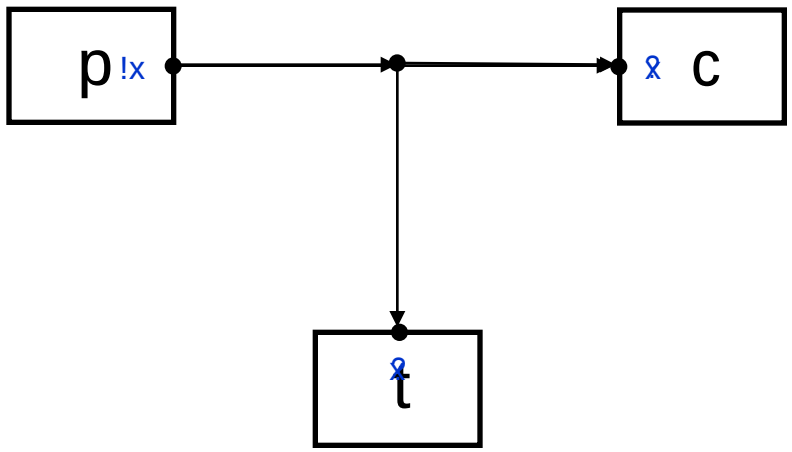
Join

- Mixed node
  - Atomic merge + replication
- Sink node
  - Non-deterministic merge
- Source node
  - Atomic replication



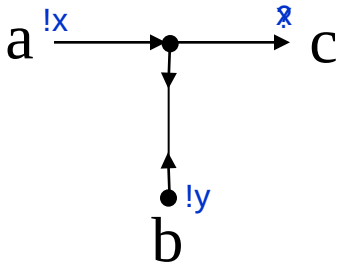
A Simple Composed System

- Read-cue synchronous flow-regulator



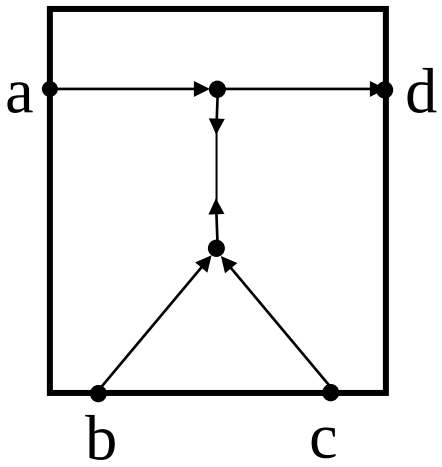
Flow regulator

- Write-cue synchronous flow-regulator



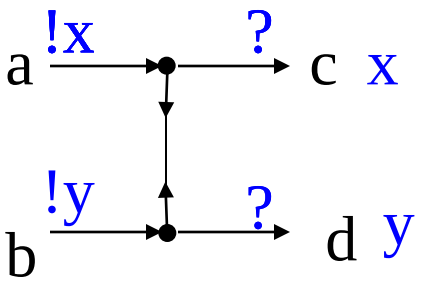
Take a through d when b or c

- 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*.



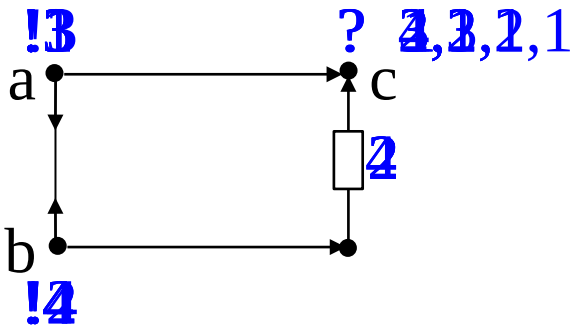
Flow Synchronization

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



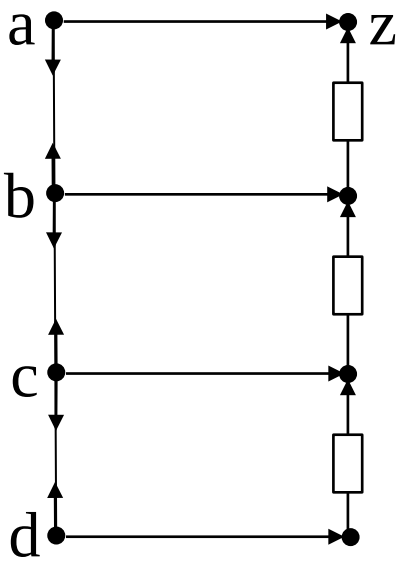
Alternator

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



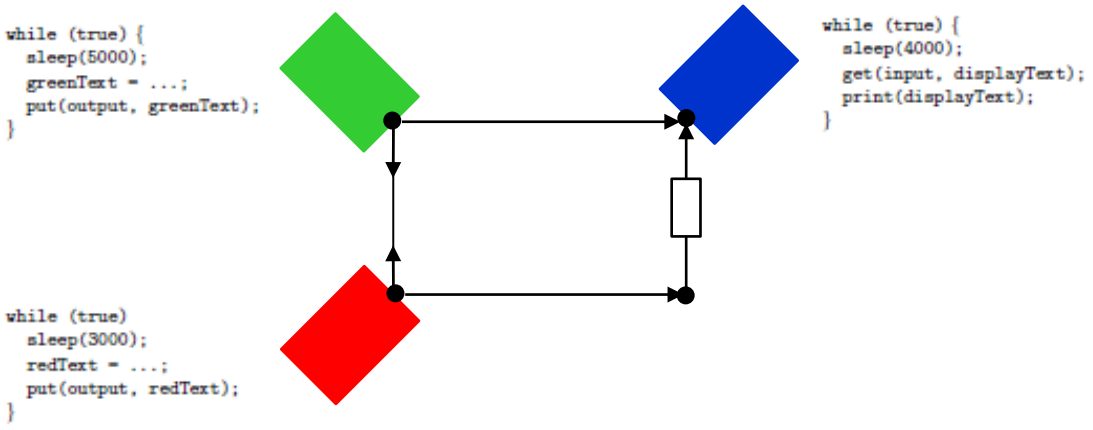
N-Alternator

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



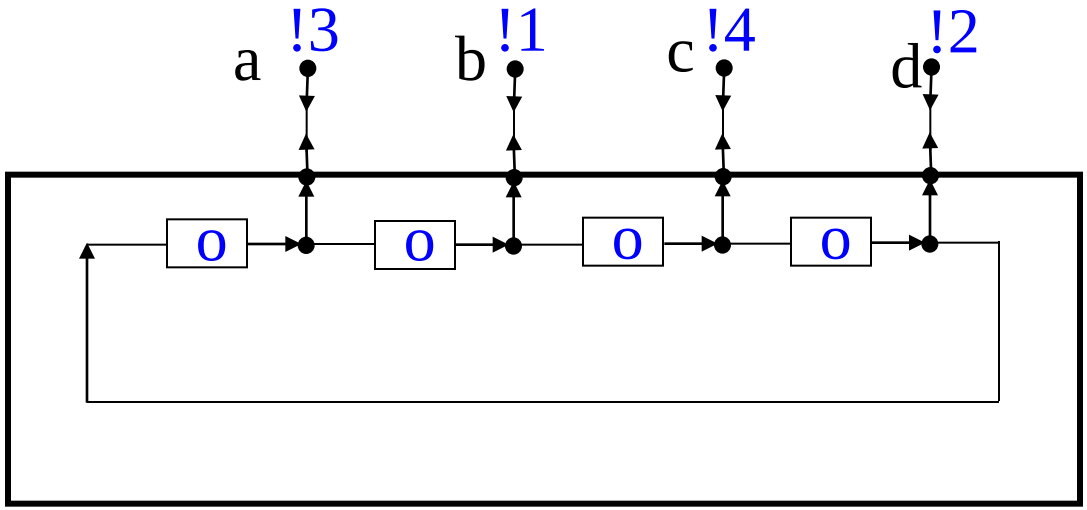
Alternating Producers

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



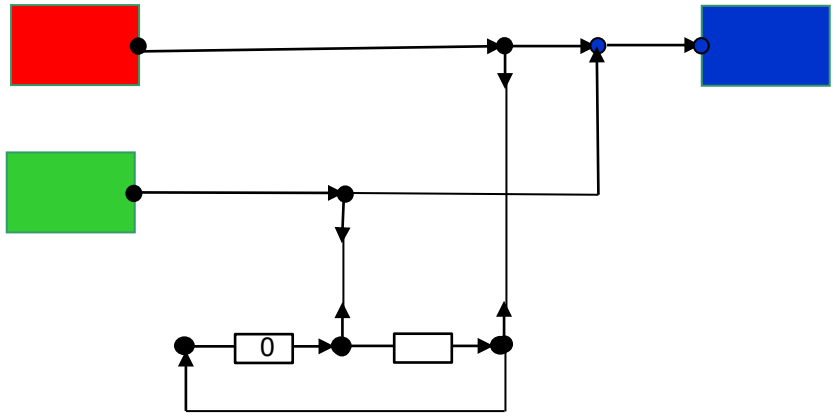
Sequencer

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



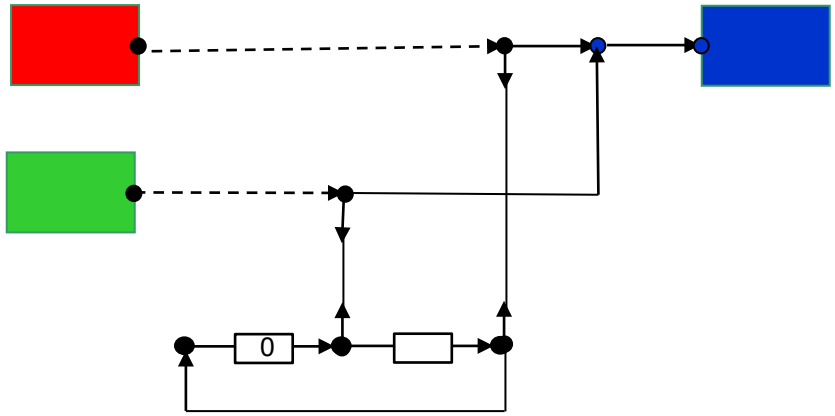
Sequenced blocking producers

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



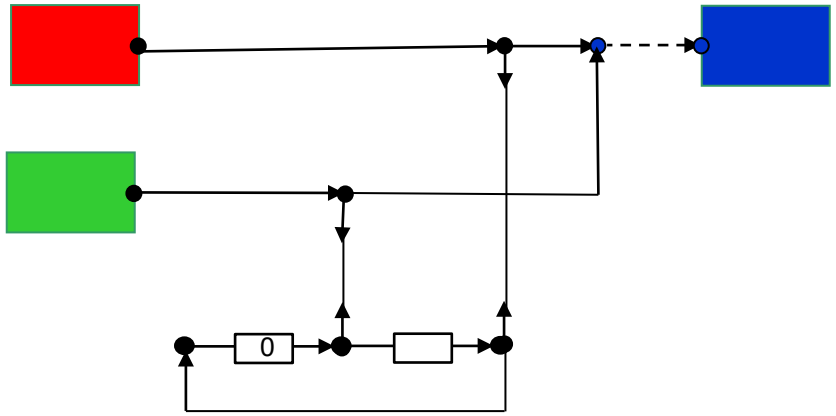
Sequenced non-blocking producers

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



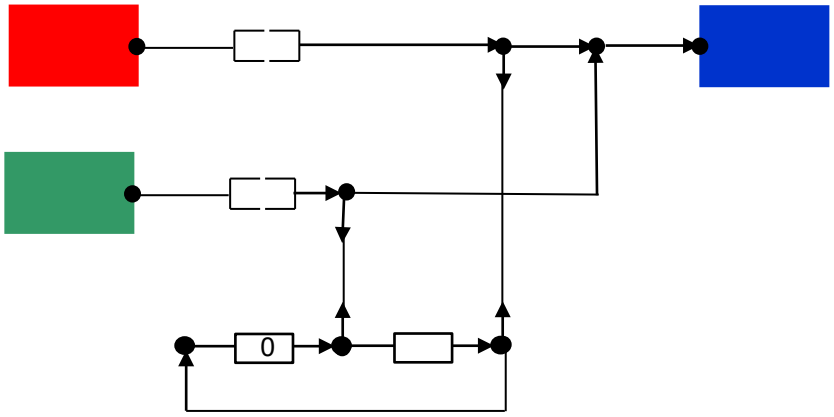
Sequenced non-blocking producers

- 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.

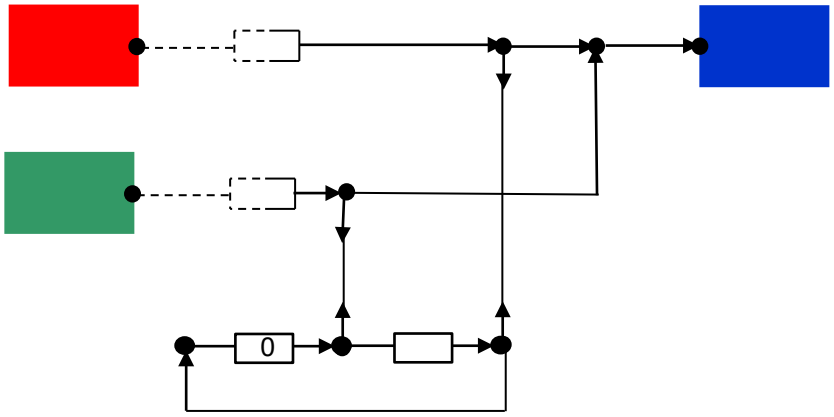


Overflow Lossy FIFO1

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

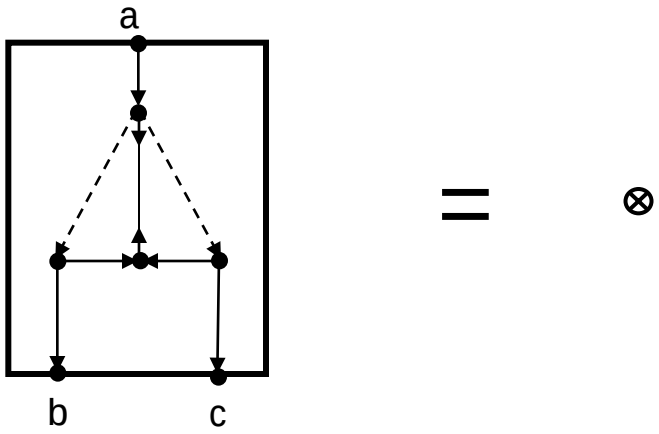
Sampled Producers

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



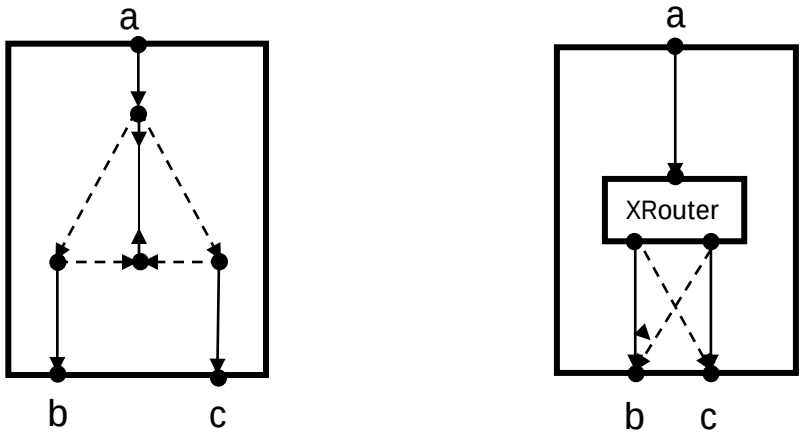
Exclusive Router

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



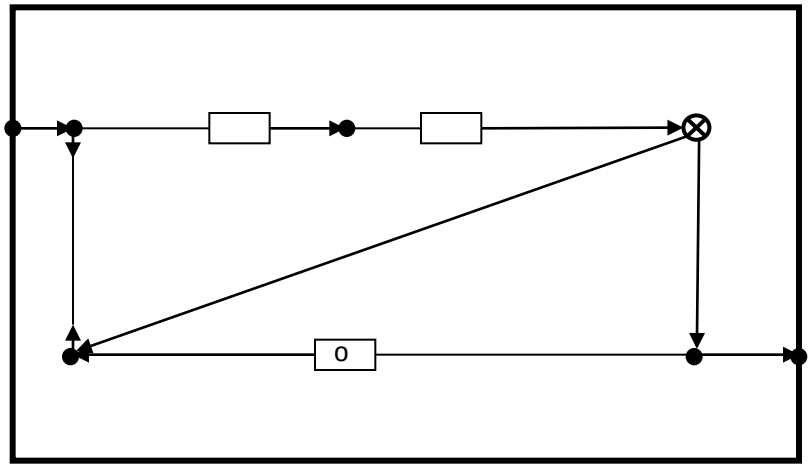
Inclusive Router

- A value written to *a* flows through to either *b* or *c*, or to both.

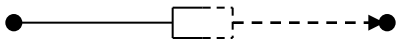


Shift Lossy FIFO1

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

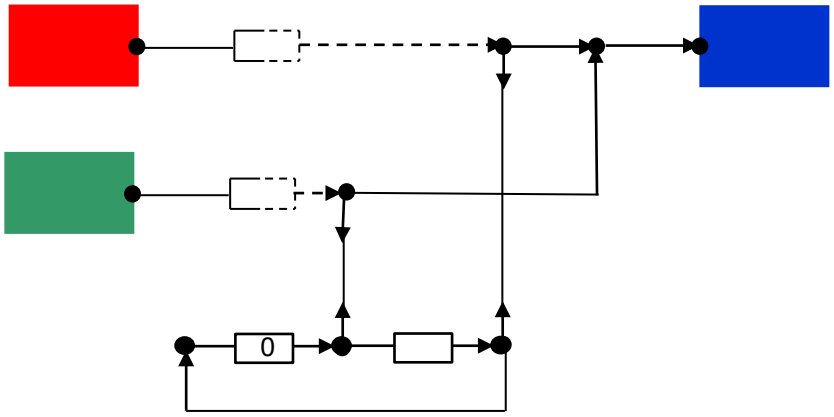


≡



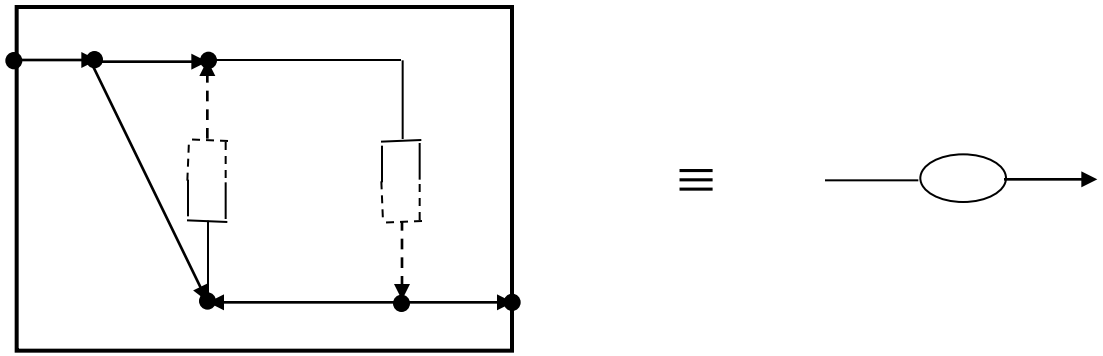
Sampled Producers

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



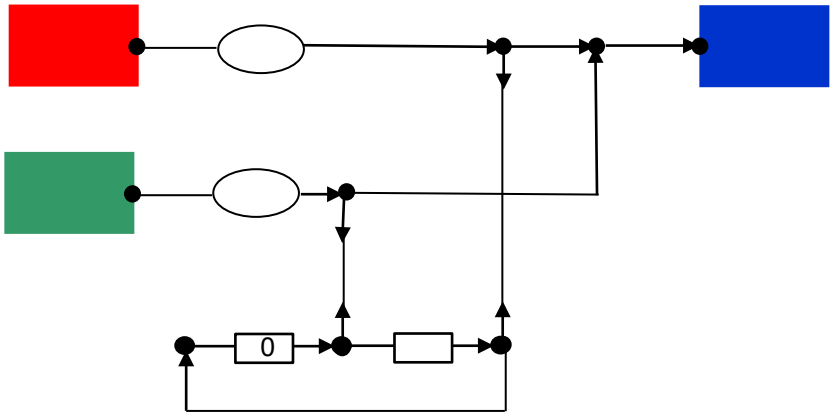
Variable

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



Buffered Producers

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



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();
}
```

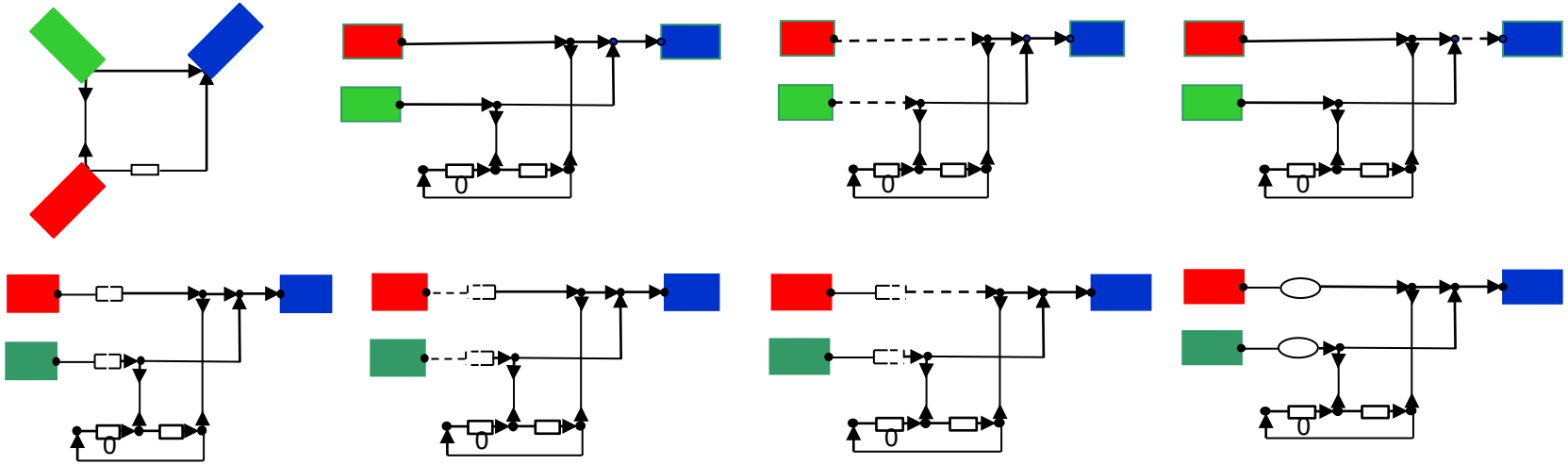
- Producers

```
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();
}
```

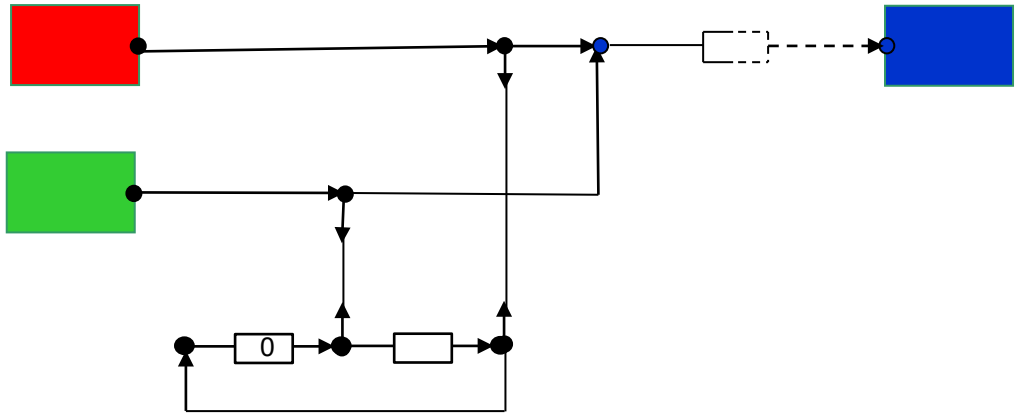
Where is Waldo?

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



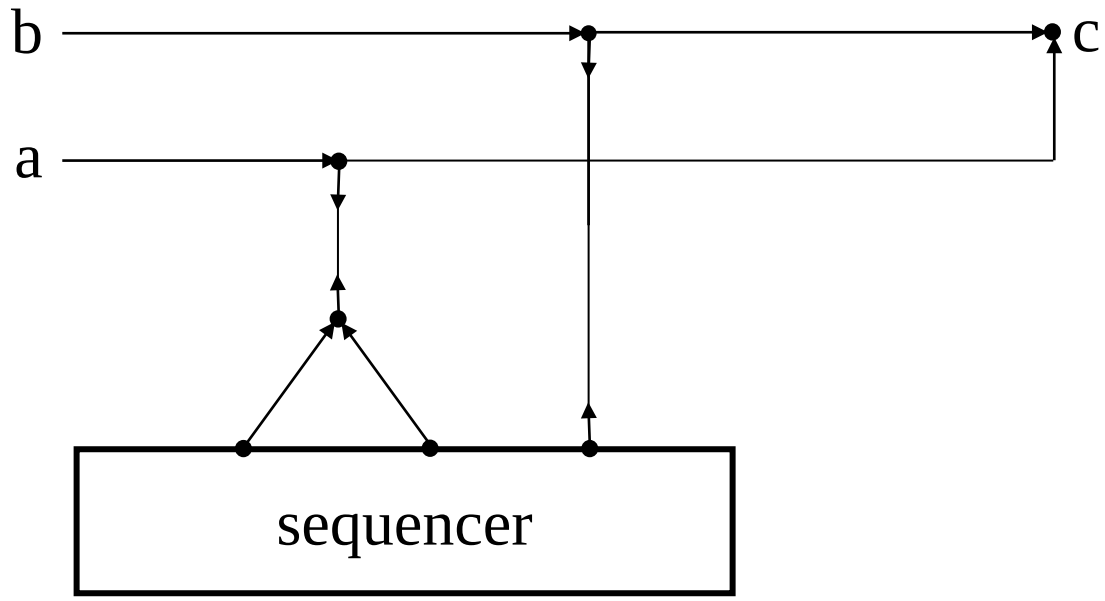
Over-writing Producers

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



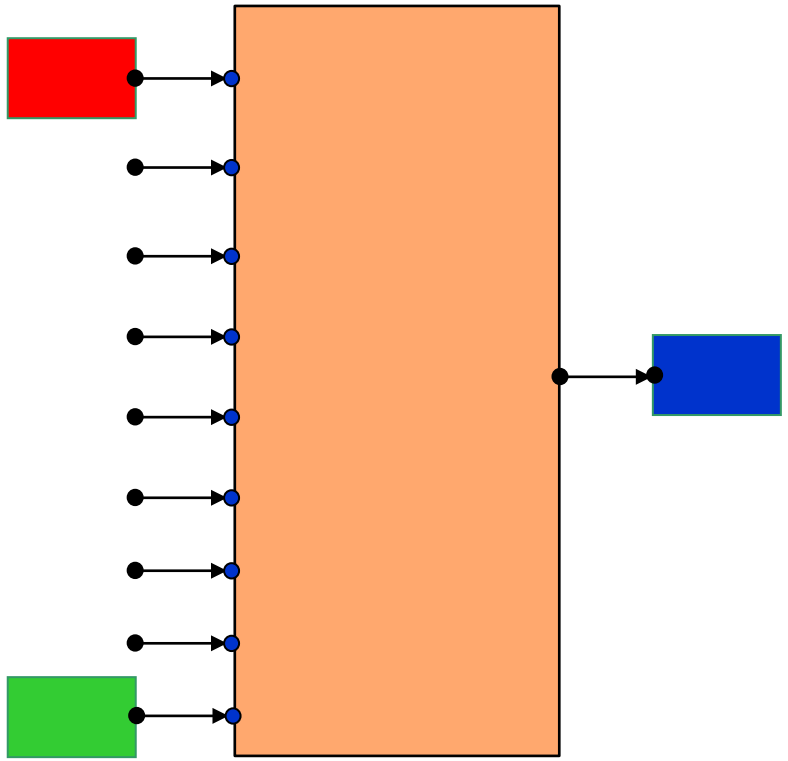
Two for one

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



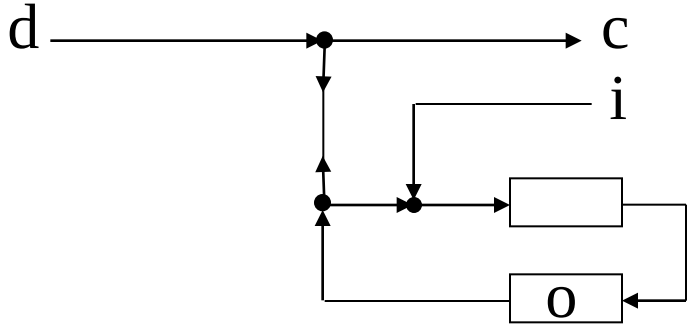
Scaling up

- Scale up?
- Mix and match?



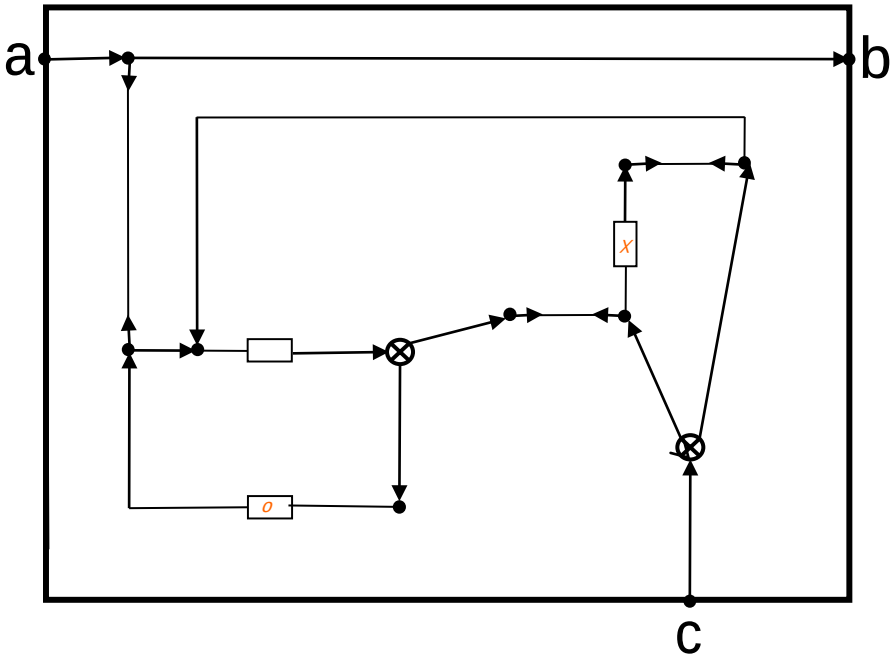
Inhibitor

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



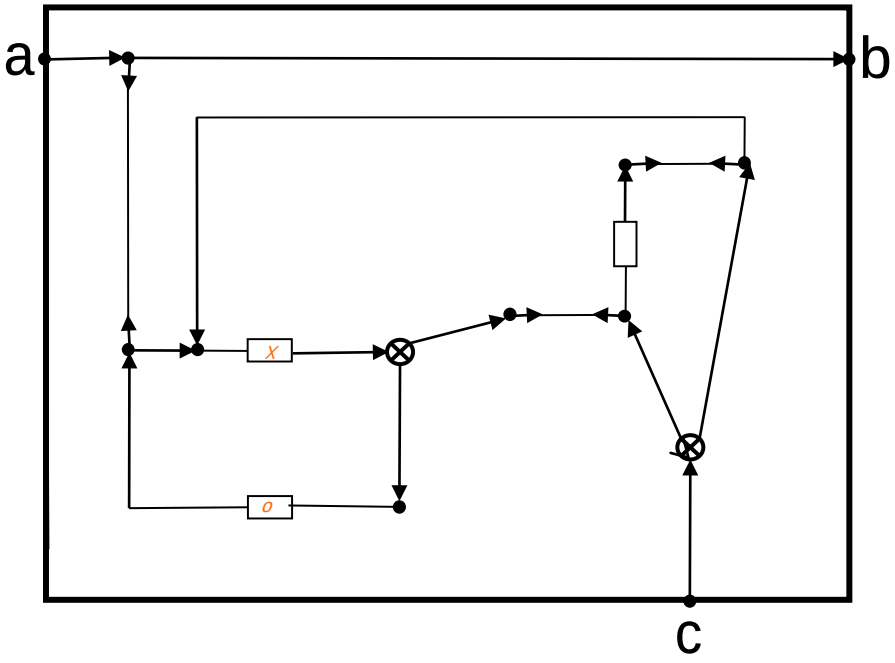
Valve (open)

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

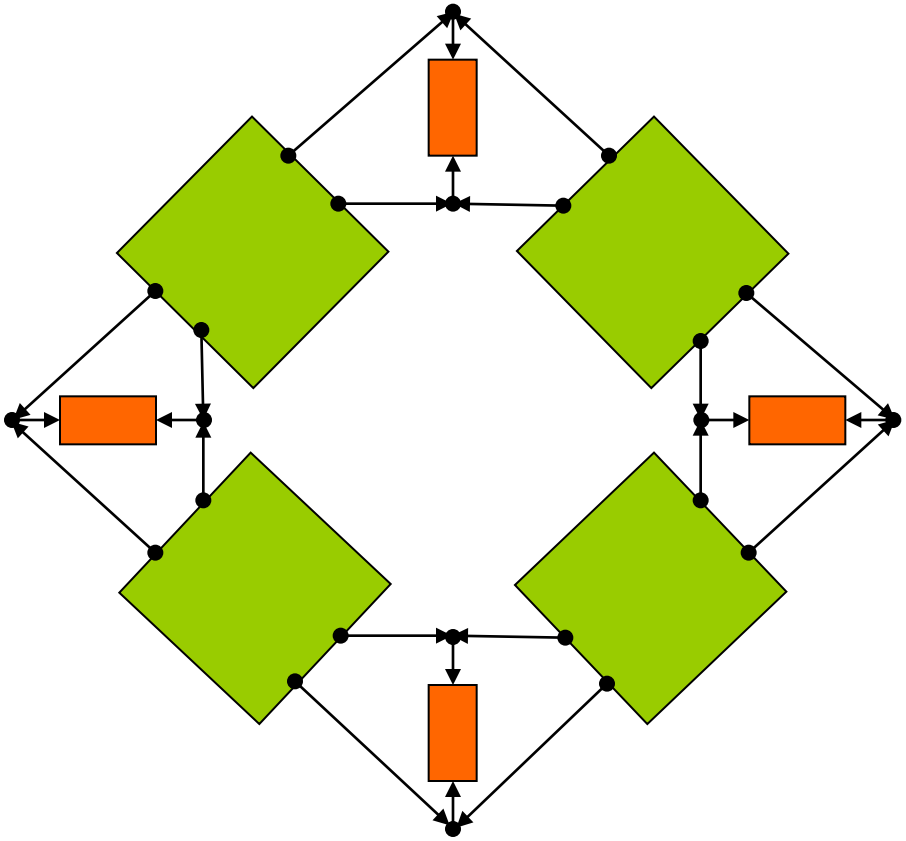


Valve (closed)

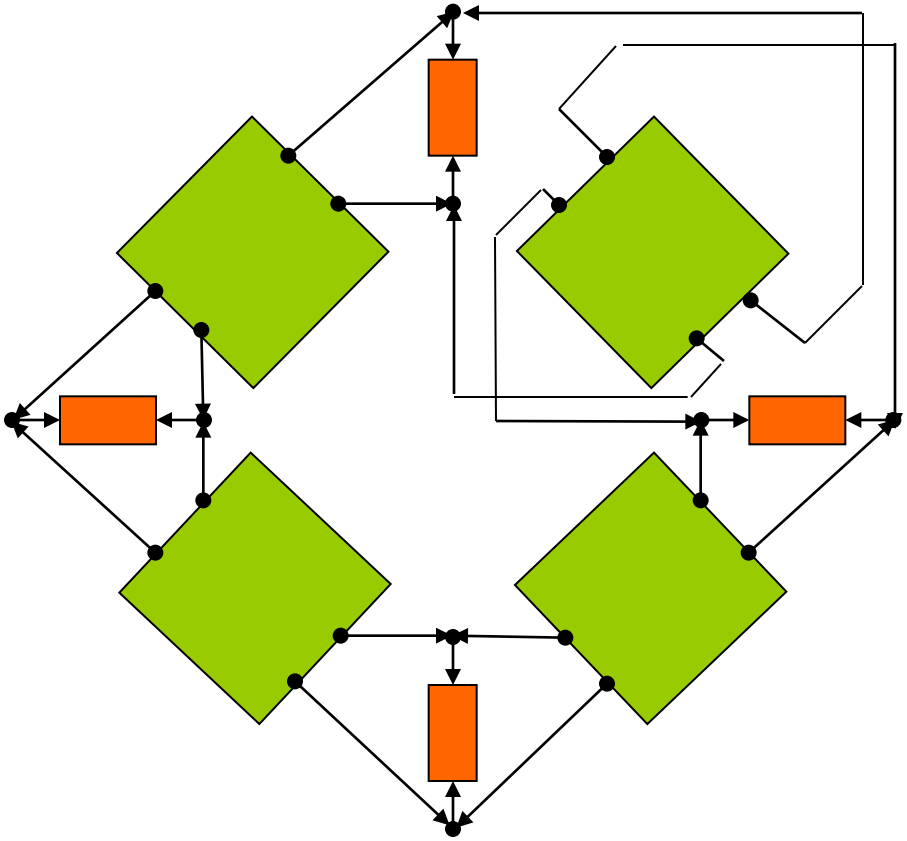
- A write to *c* opens the flow of data from *a* to *b*.



Dining Philosophers (problem)

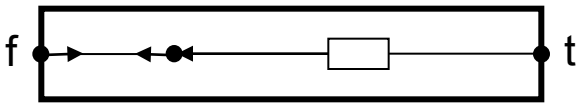


Dining Philosophers (solution)



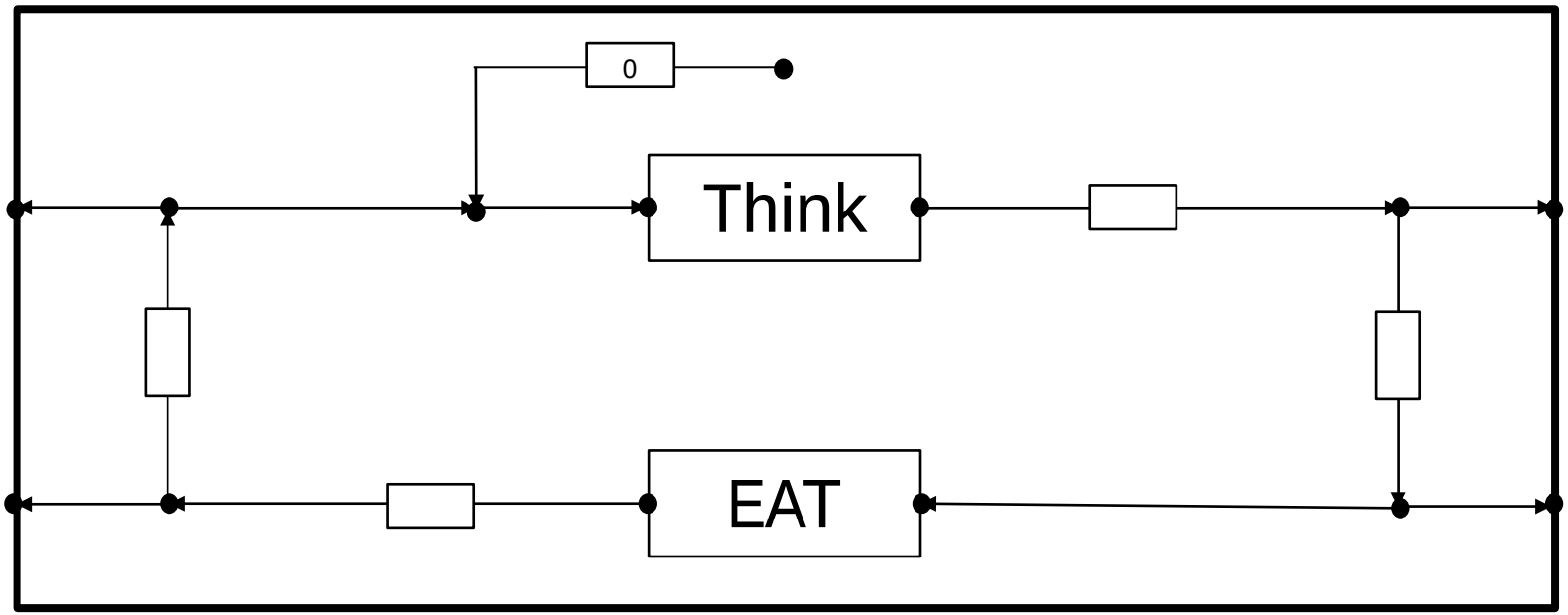
Fork

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



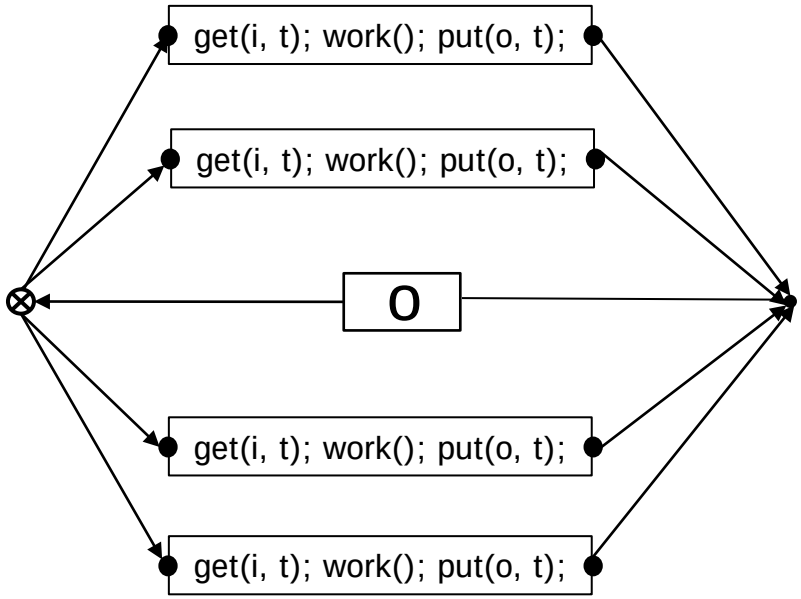
Philosopher

- Internal coordination of Think and Eat functions in a Philosopher.



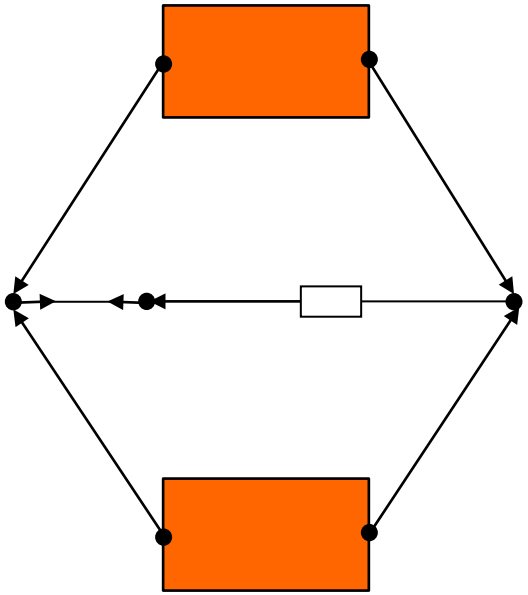
Simple mutual exclusion (get-put)

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



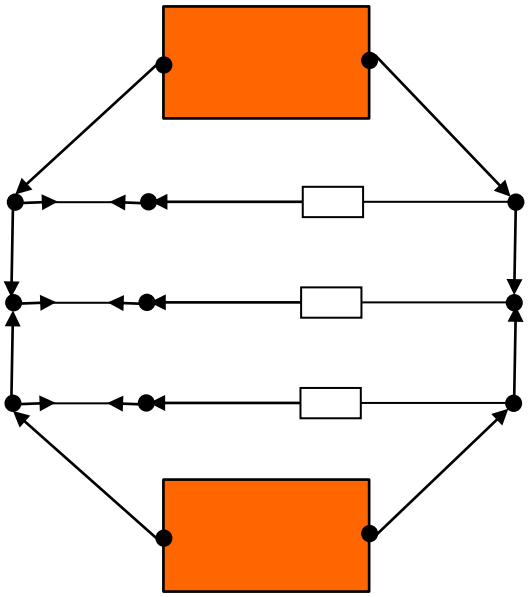
Simple mutual exclusion (put-put)

- 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.



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!



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, and dataflow models.

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.

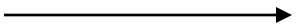
Timed-Data-Streams

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

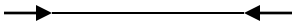
A Sample of Channels

- Synchronous channel



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

- Synchronous drain: two sources

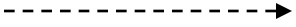


  - write/write $\text{SyncDrain}(\langle \alpha, a \rangle, \langle \beta, b \rangle;) \equiv a = b$

- Synchronous spout: two sinks

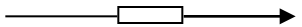
  - take/take $\text{SyncSpout}(; \langle \alpha, a \rangle, \langle \beta, b \rangle) \equiv a = b$

- Lossy synchronous channel



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

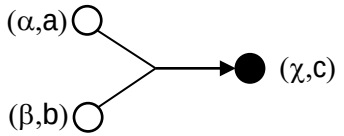
- Asynchronous FIFO1 channel



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

Behavior of Reo Nodes

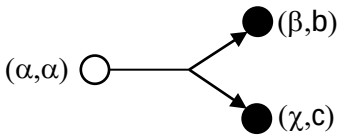
- Nondeterministic binary merge:



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

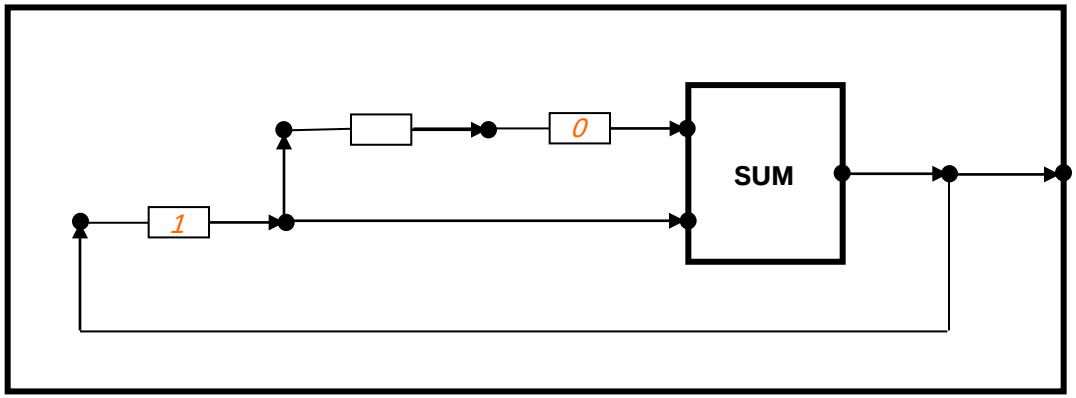
- Binary replicator:

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



Fibonacci Series

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



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

Some possible adders (1)

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

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

$$Adder2(\langle \alpha, a\rangle, \langle \beta, b\rangle; \langle \gamma, c\rangle) \equiv$$
$$\gamma(0) = \alpha(0) + \beta(0) \wedge$$
$$c(0) = max(a(0), b(0)) \wedge$$
$$Adder2(\langle \alpha', a'\rangle, \langle \beta', b'\rangle; \langle \gamma', c'\rangle).$$

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

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

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

Some possible adders (2)

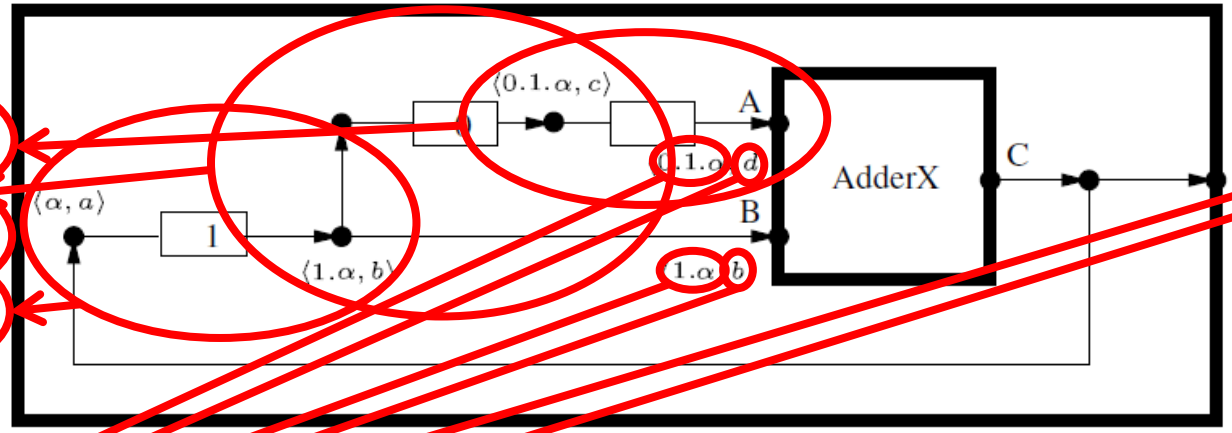
$$Adder4(\langle \alpha, a\rangle, \langle \beta, b\rangle; \langle \gamma, c\rangle) \equiv$$
$$\gamma(0) = \alpha(0) + \beta(0) \wedge$$
$$a = b = c \wedge$$
$$Adder4(\langle \alpha', a'\rangle, \langle \beta', b'\rangle; \langle \gamma', c'\rangle).$$

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

$$Adder5(\langle \alpha, a\rangle, \langle \beta, b\rangle; \langle \gamma, c\rangle) \equiv$$
$$\gamma(0) = \alpha(0) + \beta(0) \wedge$$
$$c(0) = min(a(1), b(1)) \wedge$$
$$Adder5(\langle \alpha', a'\rangle, \langle \beta', b'\rangle; \langle \gamma', c'\rangle).$$

- 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'

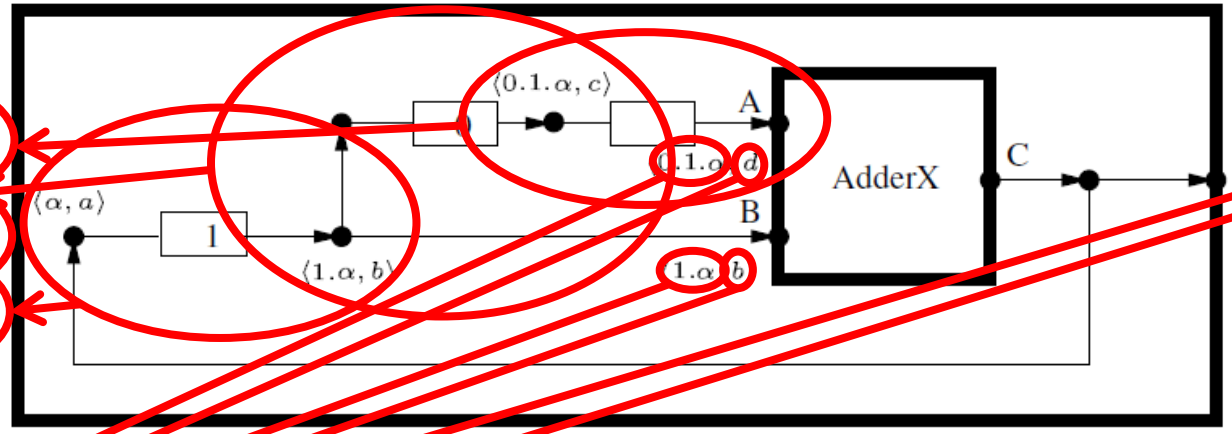
d<b<a<d'<b'

Consistent!

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

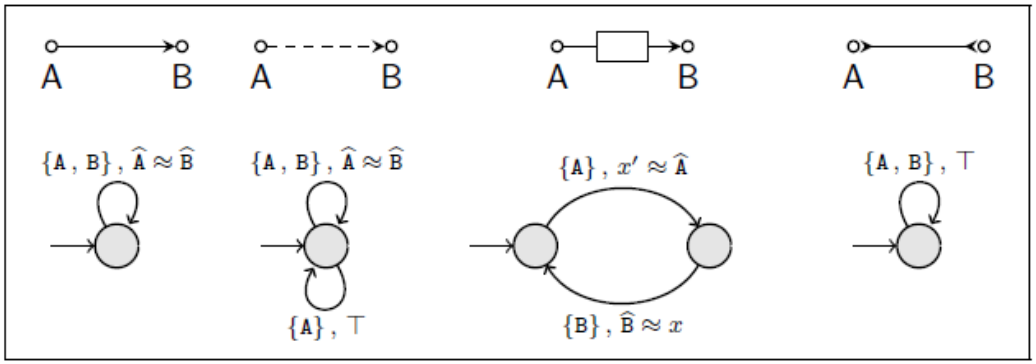
The timing equations b=a and b<a have no solution!

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:

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

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

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

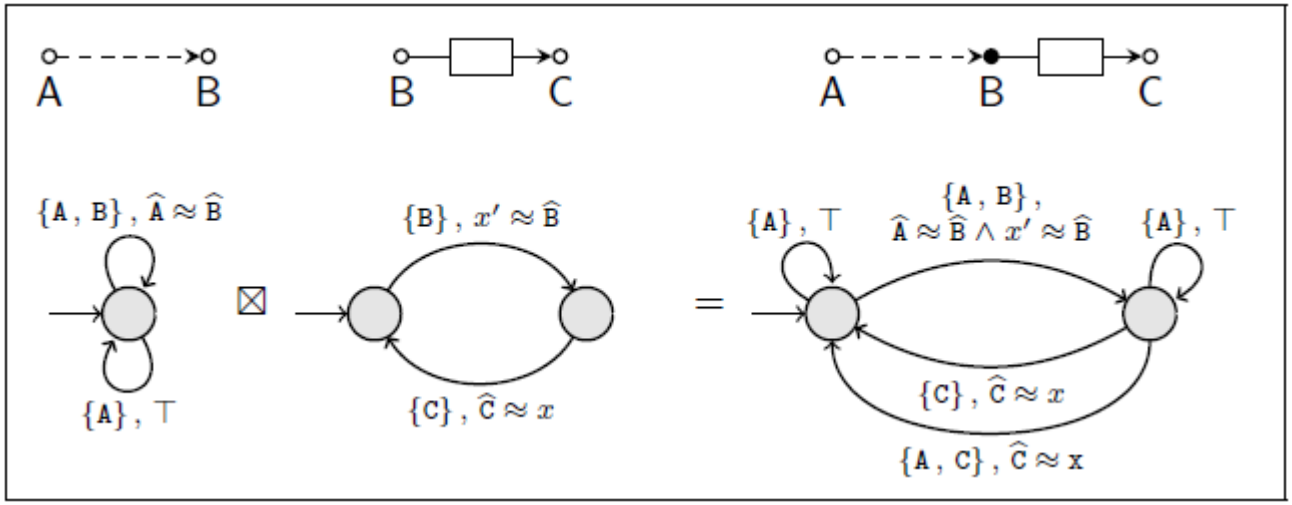
and

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

and latter's symmetric rule. □

CA of a connector

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



Extensible Coordination Tools

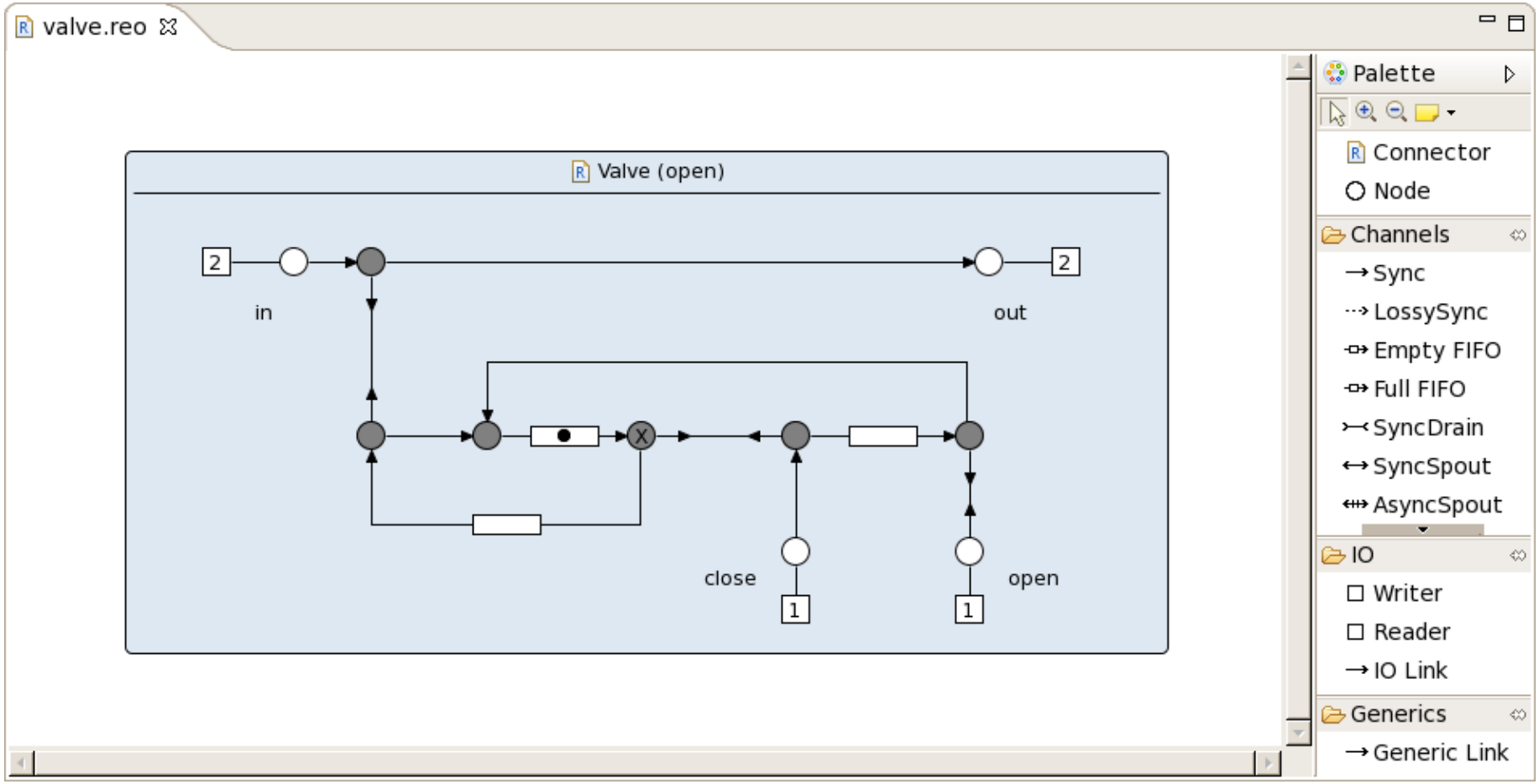
- A set of Eclipse plug-ins provide the ECT visual programming environment.
- Protocols can be designed by composing Reo circuits in a graphical editor.
- The Reo circuit can be animated in ECT.
- ECT can automatically generate the CA for a Reo circuit.
- Model-checkers integrated in ECT can be used 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

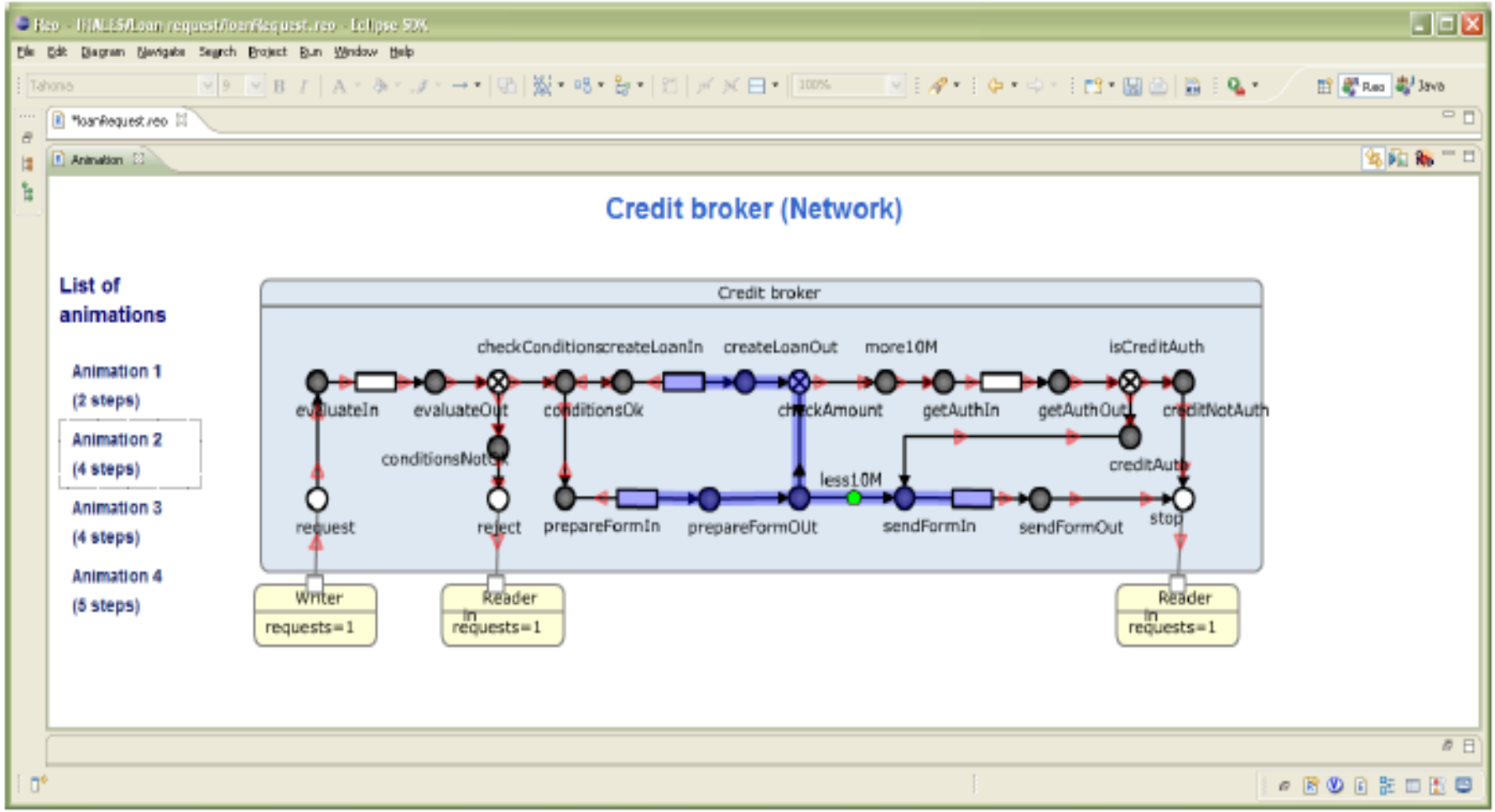
Tool support

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	•Vereofy model checker (www.vereofy.de) •mCRL model checking •Bounded model checking of Timed Constraint Automata
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 (http://www.prismmodelchecker.org)
Algebraic Graph Transformation	Dynamic reconfiguration of Reo circuits

Snapshot of Reo Editor



Reo Animation Tool



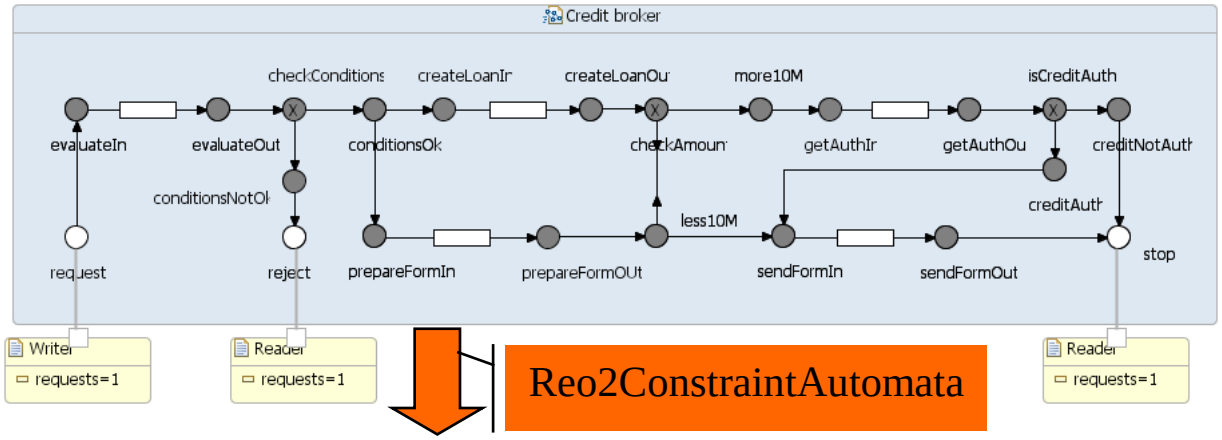
Model Checking

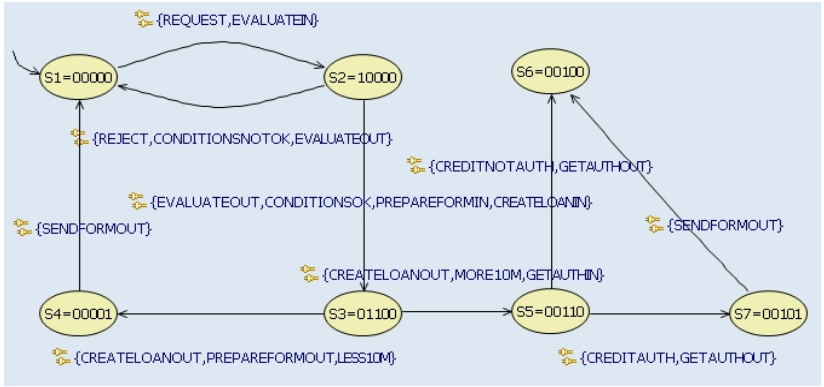
- Constraint automata are used for model checking of Reo circuits
- Model checker for Reo built in Dresden:
  - Symbolic model, LTL, and CTL-like logic for specification
  - Can also verify properties such as deadlock-freeness and behavioral equivalence
- SAT-based bounded model checking of Timed Constraint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

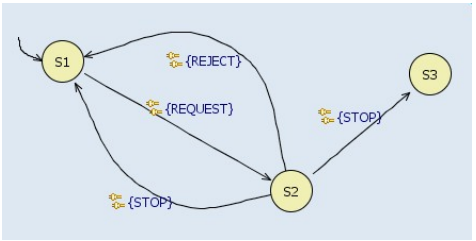
Vereofy Model Checker

- 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* → **F** (*reject* ∪ *sendFormOut*))
    - check that admissible states *reject* or *sendFormOut* are reached
- http://www.vereofy.de

Verification with Vereofy

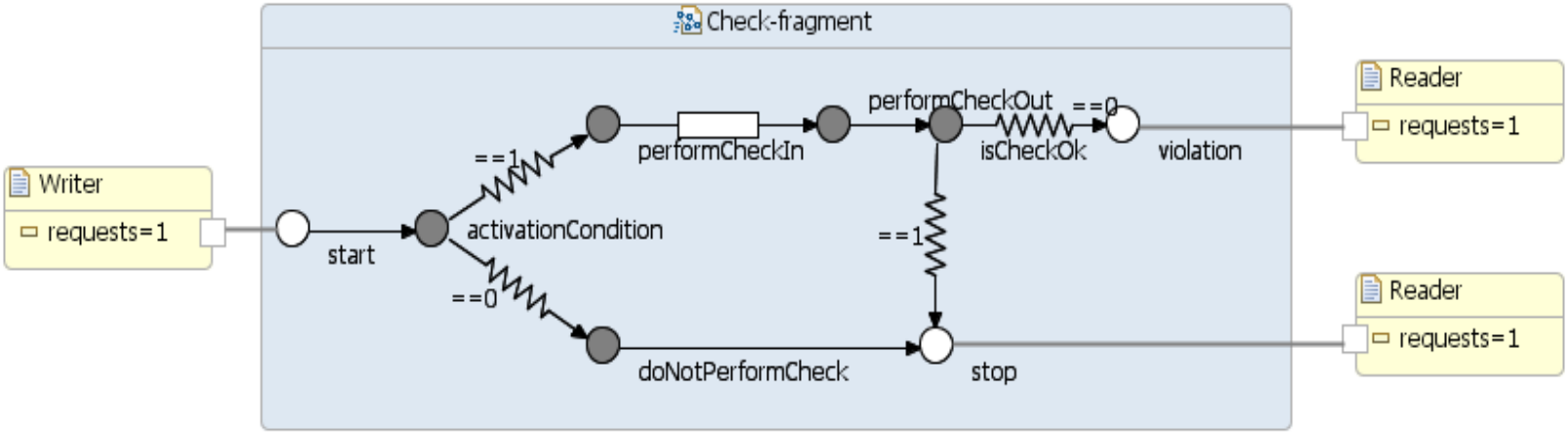






- Modal formulae
  - Branching time temporal logic: *AG*[*EX*[*true*]] – check for deadlocks
  - Linear temporal logics: **G**(*request* → **F** (*reject* ∪ *sendFormOut*)) – check that admissible states *reject* or *sendFormOut* are reached

Data-Dependent Control-Flow

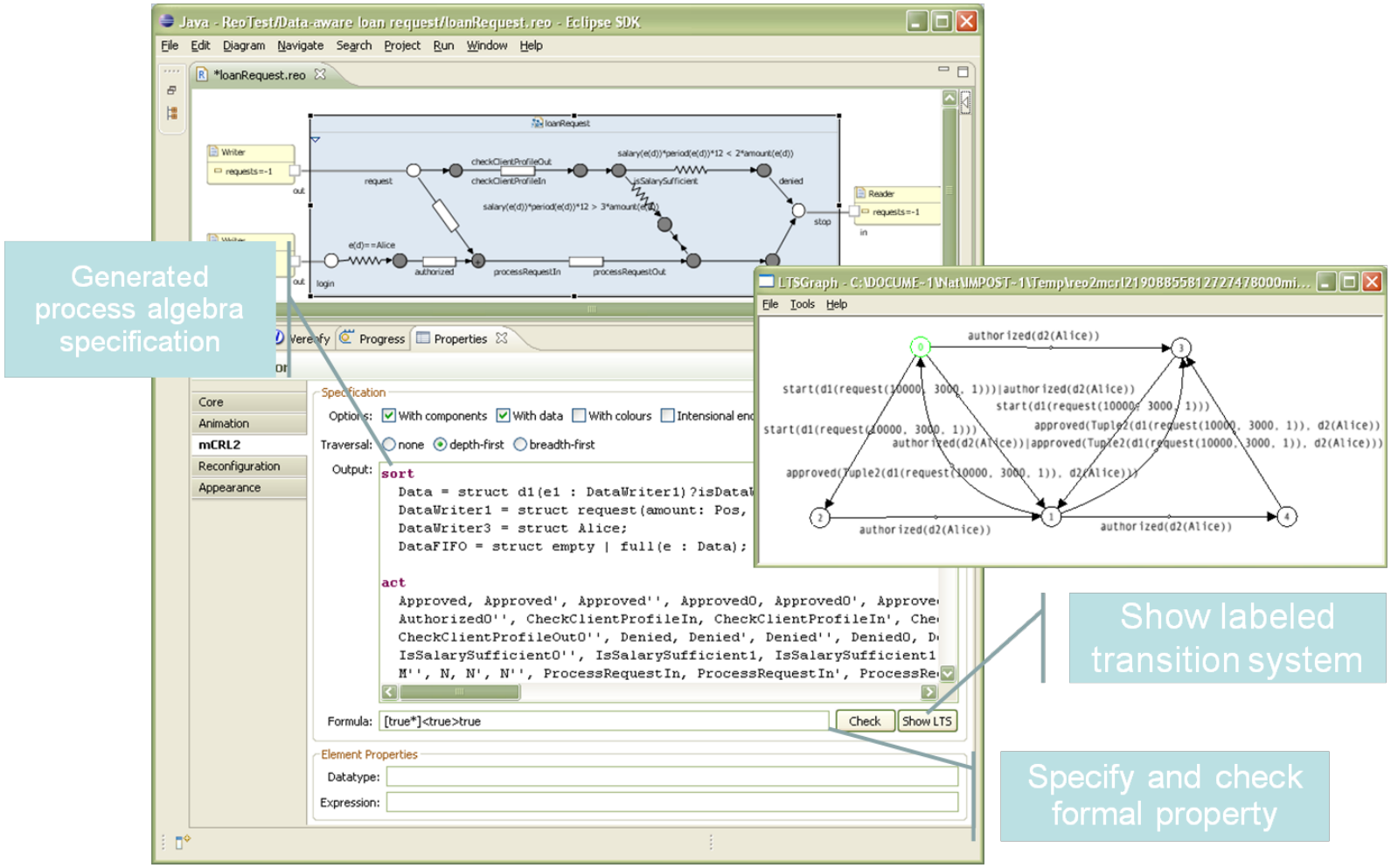


- Input parameters:
  - Activation condition
    - **Data:** b: Boolean
    - **Filter condition:** b==true, b==false
  - Check condition
    - **Data:** x, y: Real; (e.g., credit amount, maximal amount)
    - **Filter condition:** x < y
- Problems:
  - Data constraint specification language is needed
  - Properties that include conditions:
    - **G** [(b & !(x < y)) → **F** 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 (μ calculus)
  - Abstract data types, functional language (λ 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



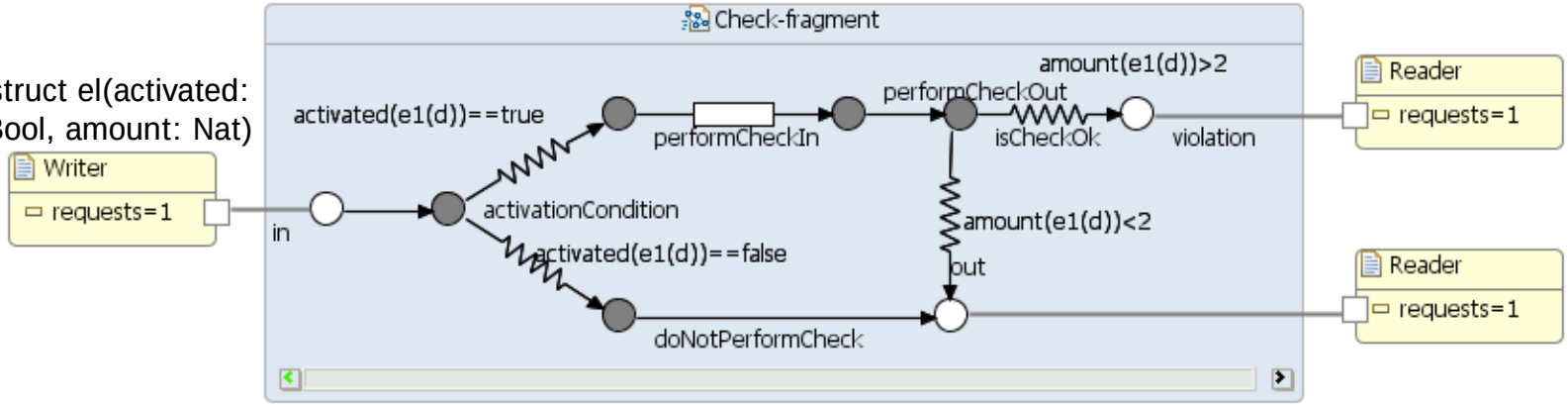
Generated process algebra specification

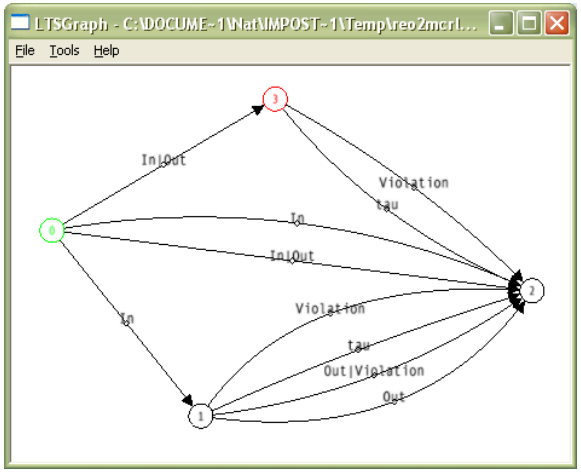
Show labeled transition system

Specify and check formal property

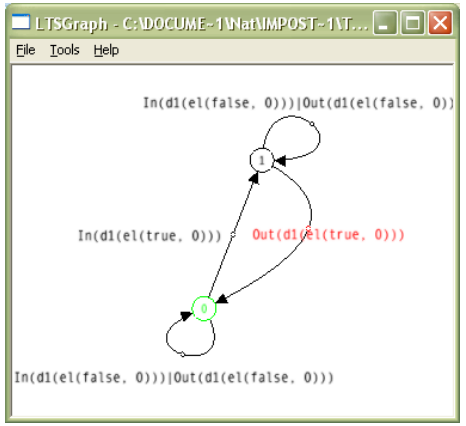
Data Dependent Control Flow

struct el(activated: Bool, amount: Nat)

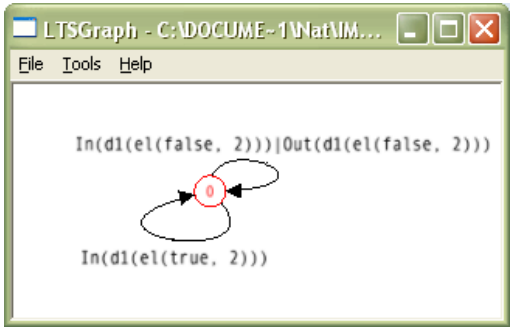




No data



(amount(d)<1)

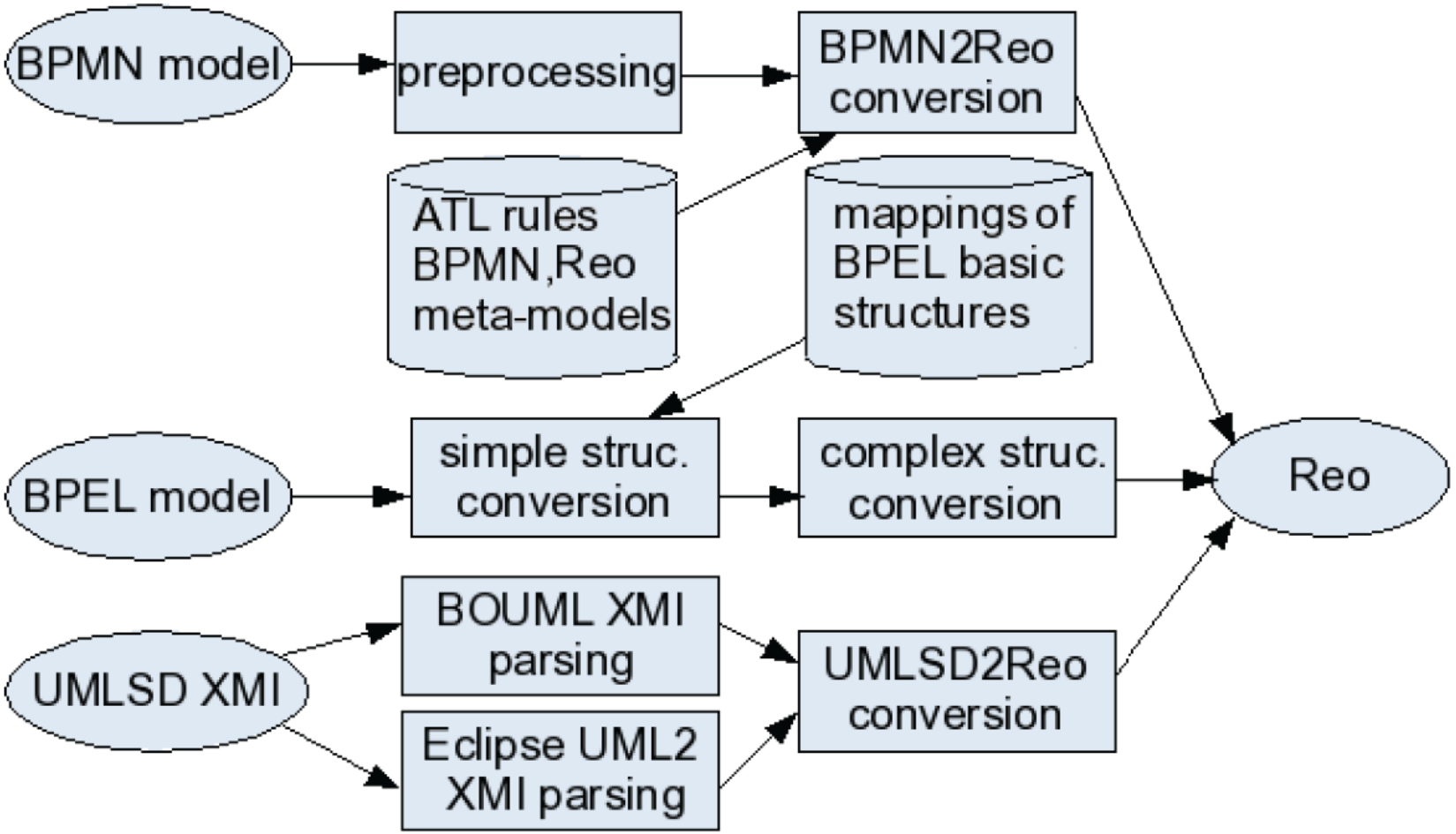


(amount(d)==2)

Process verification tools: summary

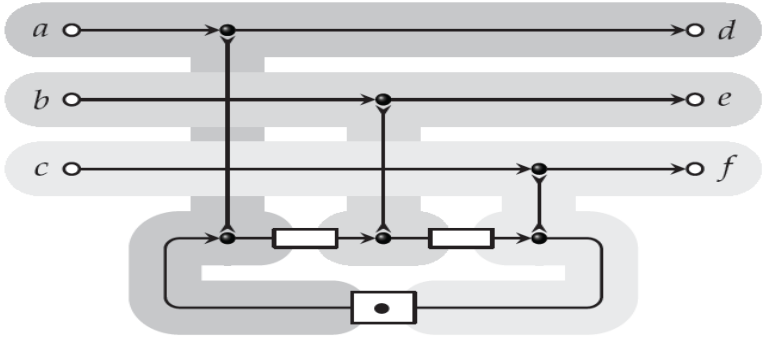
- Vereofy:
  - Advantages:
    - Developed for Reo and Constraint Automata
    - Visualization of counterexamples
  - Disadvantages:
    - No support for abstract data types
    - Global domain for all components
    - Primitive data constraint specification language (for filter channels)
- mCRL2
  - 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



Compiling Reo onto multi-core

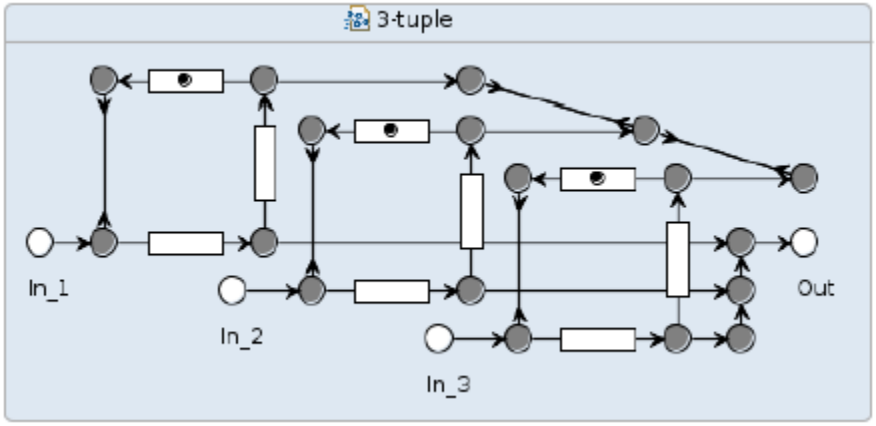
- Splits a Reo circuit into synchronous islands.
- Compiles each island into a constraint automaton.
- Maps asynchronous regions (FIFOs) into passive shared memory.
- Each island runs as a separate state machine thread concurrently with computation threads.



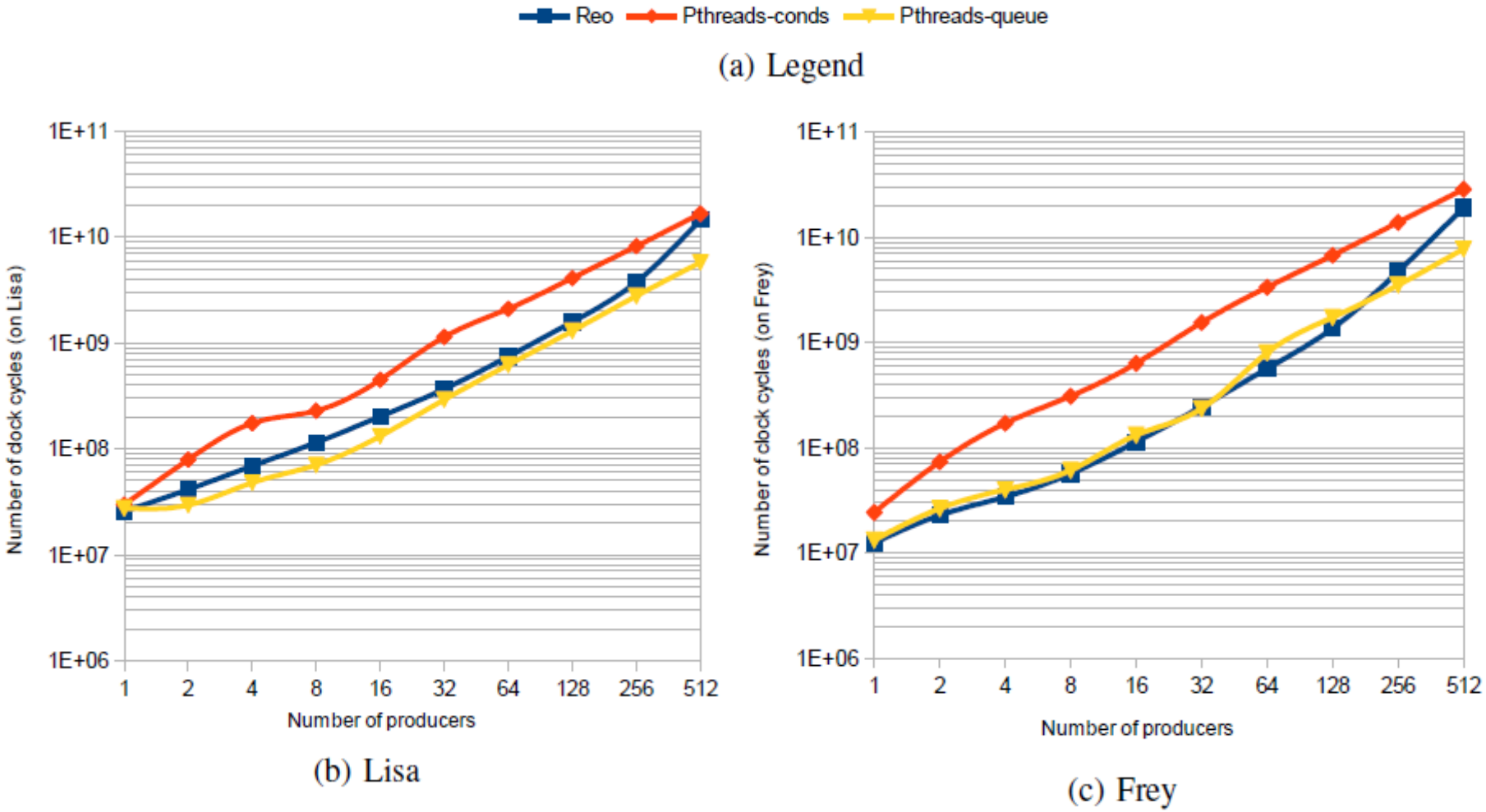
- Sung-Shik T.Q. Jongmans and Farhad Arbab, "**Can High Throughput Atone for High Latency in Compiler-Generated Protocol Code?**," LNCS, FSEN 2015, April 22-24, 2015, Tehran, Iran.
- Sung-Shik T. Q. Jongmans and Farhad Arbab, "**Toward Sequentializing Overparallelized Protocol Code**," ICE 2014: pp. 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.

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

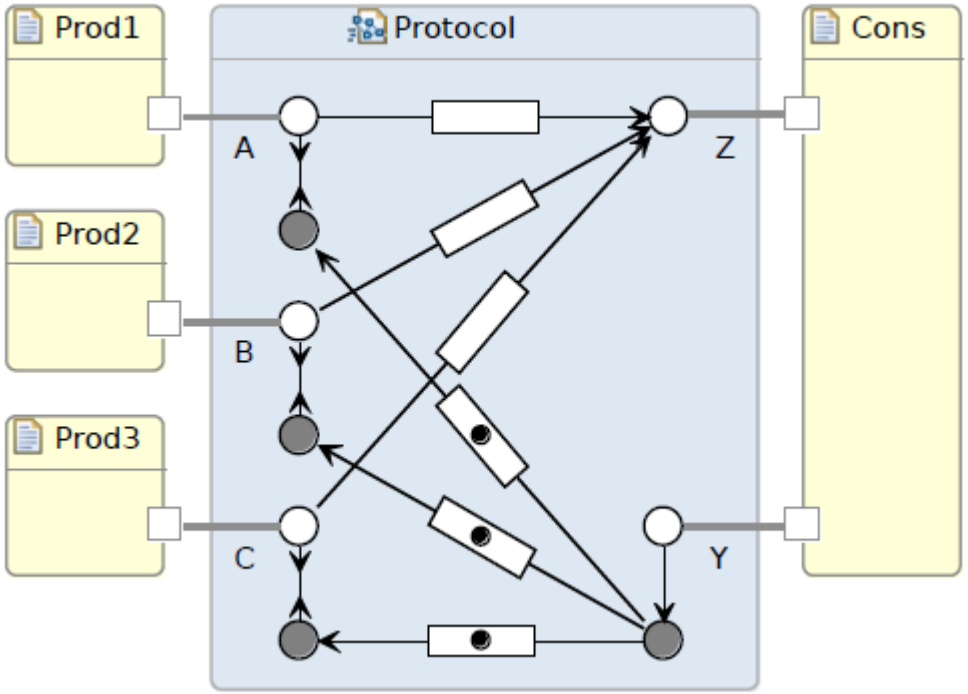


Performance

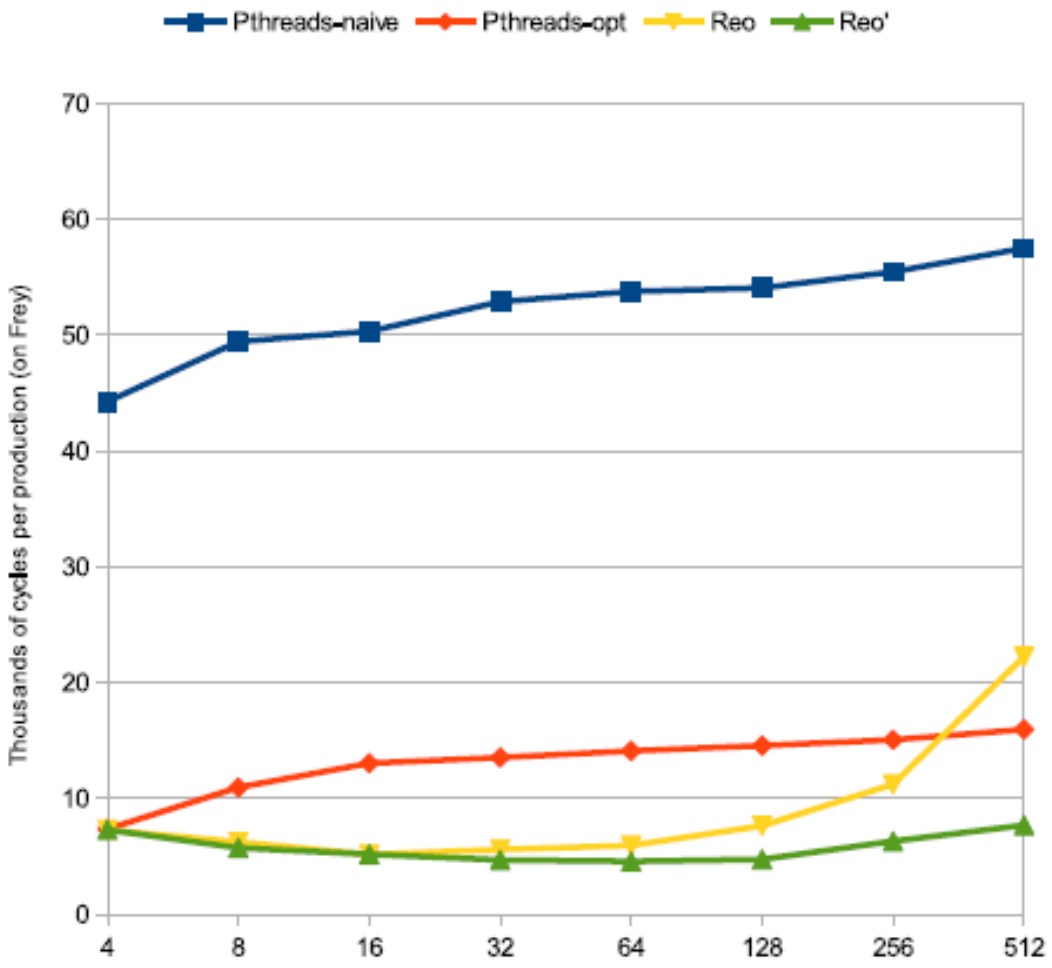


Asynchronous bundle merge

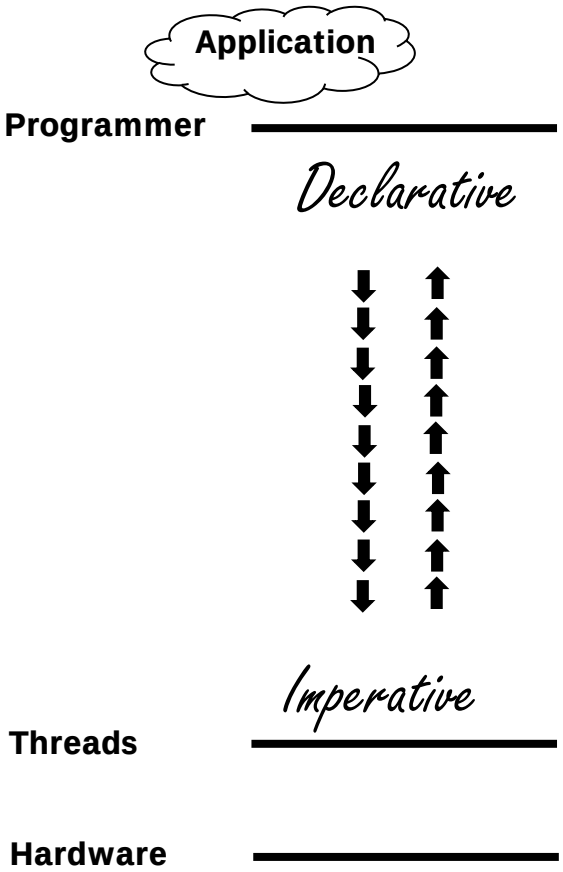
- In each cycle, the Consumer receives a bundle of *n* items, each produced by one of the producers.



Compiling Reo



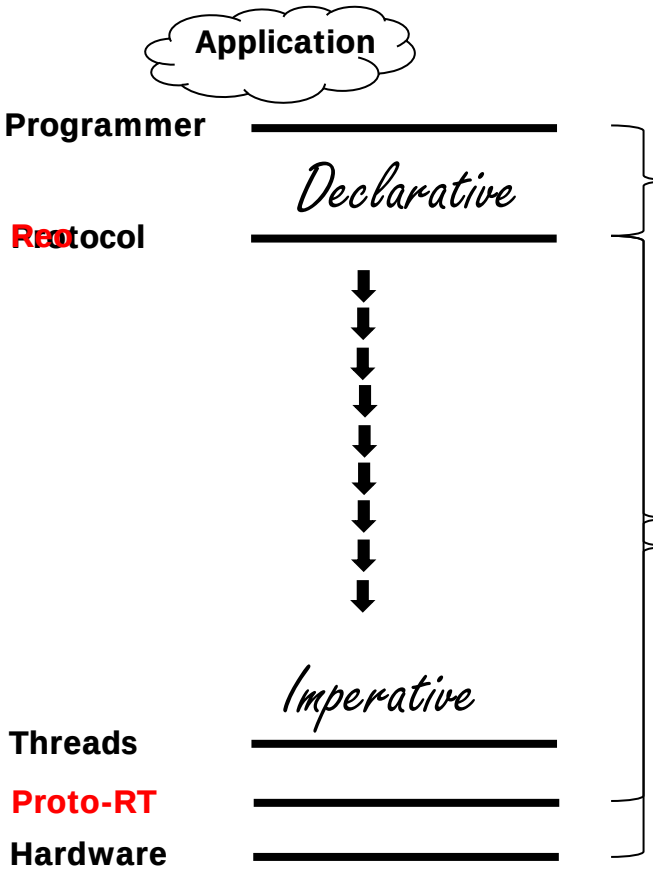
Typical Concurrent Programming



- 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



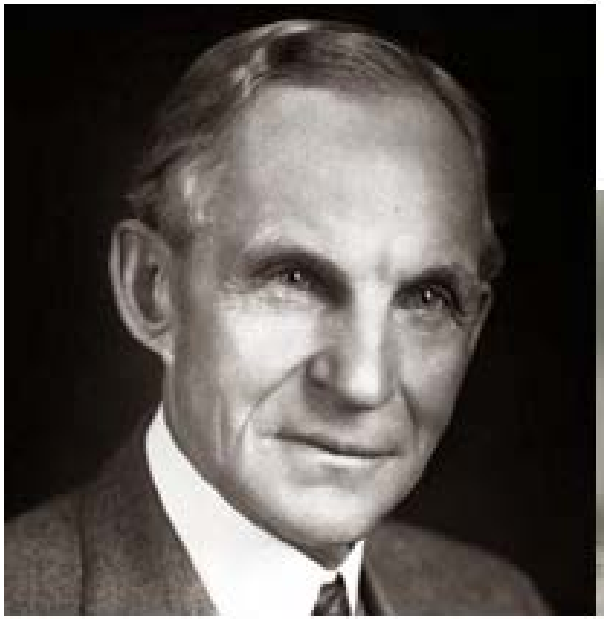
- Gap small enough for programmers to produce correct code
- Gap large enough for compilers to optimize
- Programmer's ***what*** intentions are explicit

Conclusion

- Interaction = Constraint = Relation
- Reo is a simple, rich, versatile, and surprisingly expressive language for compositional construction of pure interaction protocols.
  - Strict separation of computation and concurrency concerns.
  - A model where interaction is (the only) first-class concept.
  - Free combination of synchrony, exclusion, and asynchrony.
  - Exogenous interaction /coordination.
  - Direct composition of protocols:
    - synchrony and exclusion propagate through composition.
  - Verbatim reuse of protocols.
- Model-driven development:
  - from specification, analysis, and verification to executable.

http://reo.project.cwi.nl

Observation



"If I had asked people what they wanted, they would have said faster horses."

Henry Ford