# Architectural design: the coordination perspective

Luís S. Barbosa

HASLab - INESC TEC
Universidade do Minho
Braga, Portugal

12 June, 2014

## Software architecture for reactive systems

There is no general-purpose, universally tailored, approach to architectural design of complex and reactive systems

Therefore, the course

- introduces different models for reactive systems
- discusses their architectural design
- with (reasonable) tool support for modelling and analysis

## Software architecture for reactive systems

- Introduction to software architecture
- Models and logics for reactive systems
  - Classical (non deterministic) (mCRL2)
  - Timed (with real time constraints) (Uppaal)
  - Probabilistic (PRISM)
  - Cyber-physical (KeYmaera)
- Architecture for reactive systems
  - An architectural description language (AADL)
  - Component-oriented architectural design
  - Coordination-oriented architectural design
    - Paradigm: The Reo exogenous coordination model
    - Extension: Probabilistic reactive systems (PRISM)
    - applied to coordination design with probabilistic requirements
  - Reconfigurable architectures
    - Paradigm: Specifications in hybridised logics (support in Hets)

## Composition in object-orientated software

- In OO the architecture is implicit: source code exposes class hierarchies but not the run-time interaction and configuration
- Objects are wired at a very low level and the description of the wiring patterns is distributed among them
- The semantics of method invocation is heavy and non-trivial:
    - The caller must know the callee and the method.
    - The callee must (pretend) to interpret the message.
    - The caller suspends while the callee (pretends to) perform the method and resumes when the callee returns a result.

## Composition in object-orientated software

The operations/methods provided by a class-interface impose a tight semantic binding which, at the inter-component level

- Weakens independence of components;
- Contributes to breaking of encapsulation;
- Tightens component inter-dependence.

## Composition in component-based software

- CBD retains the basic encapsulation of data and code principle to increase modularity
- ... but shifts the emphasis from class inheritance to object composition
- to avoid interference between inheritance and encapsulation and pave the way to a development methodology based on third-party assembly of components

## Composition in component-based software

- a palette of computational units (eg robust collections of objects) treated as black boxes
- and a canvas into which they can be dropped
- connections are established by drawing wires
- inter-component communication is through messages that invoke remote methods, typically given some suitable triggering condition on the source.

## Composition in coordination-based software

- a palette of computational units (eg robust collections of objects) treated as black boxes
- and a canvas into which they can be dropped
- connections are established by specific devices (with complex logic, memory, etc)
- inter-component communication becomes anonymous and externally coordinated

## Composition as coordination

### Example scenario

- Components: a bar-code scanner and a LCD panel
- bar-code scanner: single output port to communicate the product id
- LCD panel: single port to input text to be displayed
- Goal: build a system that allows to scan a product bar-code and have its name displayed on the LCD
- Problem: mismatch detected between the two components operation rates

## Composition as coordination

### Example scenario

What to do when the user starts scanning bar-codes at a pace that exceeds the rate at which the LCD can display the product names?

- Do we force the bar-code scanner to wait for when its output port is not busy to read another bar-code? (forced synchronisation)
- Do we buffer the excess data and display it on a read first, display first order?
- Do we disregard bar codes that are input while the LCD is busy displaying a previous product name?
- Do we combine the approaches 2, and 3 and provide a limited buffer where a finite amount of bar codes can be buffered while the LCD is busy displaying a product name?

## Composition as coordination

### Lesson learned

Building a system out of independent components does not simply amount to wiring properly their ports together.
Special glue code is necessary to coordinate their interactions.

### Coordination

- Endogenous: provide primitives that must be incorporated within a computation for its coordination
- Exogenous: ensure that the conceptual separation between computation and coordination is suitably respected

## Coordination

Carriero and Gelernter, 1986

Coordination is the process of building programs by gluing together active pieces

- distinguish computation from interaction
  (in massive parallel networks)
- focus on the emergent behaviour
- amenable to external, third-party control

Peter Wegner, 2000

Coordination is constrained interaction

## Composition in service-oriented software

'entails the need of managing an open-ended structure of autonomous components, possibly distributed and highly heterogeneous.
This means developing software components that are autonomous and can be interconnected with other components, software or otherwise, and managing the interconnections themselves as new components may be required to join in and others to be removed.' (Fiadeiro, 05)

- interaction as a first-class citizen
- composition as exogenous coordination

## Composition in service-oriented software

- interacting components need not know each other.
  (cf traditionally communication is targeted, making the sender semantically dependent on (the scheme used to identify) the receiver)
- communication becomes anonymous: components exchange identifiable sequences of passive messages with the environment only
- therefore third parties can coordinate interactions between senders and receivers of their own choice

## Composition as coordination

### Components

- loci of computation
- are kept independent of each other and of their environment
- Components communicate with the environment only through read and write operations on the connector ends (or ports), possibly according some behavioural interface description.

## Composition as coordination

### Connectors

- act as interaction controllers: the glue code that makes components interact
  i.e., they coordinate the activities of individual components to ensure their proper interaction with one another to form a coherent system that behaves according to its requirements
- have no relevant role in the computation carried out by the overall system: they are component-independent and agnostic wrt the underlying computation model
- provide systems-independent interaction protocols
  (whereas components provide systems-specific functionality)
- ... built compositionally.
- but traditionally, glue code is the most rigid, component specific, special purpose software in component based systems!

## Reo

reo.project.cwi.nl/

- A compositional, connector-based coordination language for plugging together components in an exogenous discipline (from outside and without participants' knowledge);
- Primitive circuit-like connectors are composed to build complex coordination patterns
- Key concepts are synchrony ('happens together') and mutual exclusion;
- Connectors implement interaction protocols (dealing with aspects of concurrency, buffering, ordering, data flow and manipulation);

## Reo

reo.project.cwi.nl/

- Several formal semantics:
  - relations between timed streams (2002)
  - constraint automata (2004) and several variants
  - colours (2004) to capture context awareness
  - reo automata (2009) and intensional automata (2010)
  - ...
- Eclipse toolset available

## Reo connectors

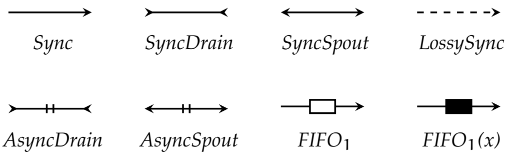
Characterized by

- a number of ends and a constraint which defines an interaction protocol through these ends

### Ends

- source end: through which data enters the connector
- sink end: through which data comes out of the connector

### Examples (channels)



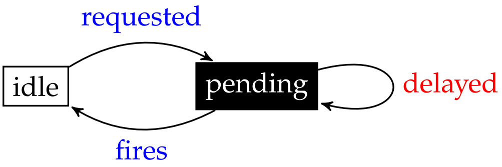
## Connector configurations

The configuration of a connector is the (abstract) structure that describes its global state:

- internal: describes the connector memory
- external: describes the environment in which the connector is currently being evaluated, i.e. the status of its ports

## Ports

- are the only medium for interacting with a connector (through io operations)
- a connector can have at most one component connected at each of its ports performing io requests
- upon the arrival of an io operation request at one of its ports, the connector decides whether the io operation can be fired or has to be delayed (becoming pending) because the interaction constraints that the connector imposes are not satisfiable in the present configuration.



## Connector memory

- Connectors with memory can store data in its buffer cells
- A buffer cell has two configurations: full or empty
- Connectors without memory cannot store data: any datum either flows through the connector to another port where it is output or it is lost

## Connector behaviour

- Dataflow behaviour is discrete in time: it can be observed and snapshots taken at a pace fast enough to obtain (at least) a snapshot as often as the configuration of the connector changes
- At each time unit the connector performs an evaluation step: it evaluates its configuration and according to its interaction constraints changes to another (possibly different) configuration
- A connector can fire multiple ports in the same evaluation step

## Connector behaviour

Synchronous dataflow behaviour

: synchronous means solely that a set of ports fire atomically, in a single indivisible step

Asynchronous dataflow behaviour

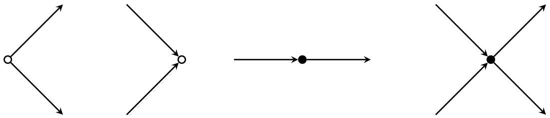
: mutual exclusion means that ports from different sets can never fire together

Context-dependent behaviour

: allows a connector to propagate information about pending io operations on its ports: dataflow behaviour may depend on the presence or absence of pending operations

## Connector composition

Connectors are composed by conjoining their ends to form nodes with multiple ends



## Connector composition

### Nodes

- source node: superposes only source ends and atomically copies incoming data items to all of its outgoing source ends
- sink node: superposes only sink ends and acts as a non-deterministic merger, randomly choosing a data item from one of the sink ends for delivery
- mixed node: combines both acting as pumping station by atomically consuming a data item from one sink end and replicating it to all source ends ($1 : n$ synchronization)

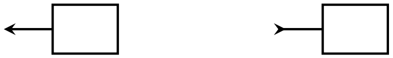
### Note: synchrony propagates through connectors

... because nodes do not perform any buffering

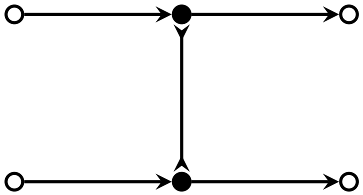
## Components

- active (computational) entities with a fixed interface that consists of a number of source and sink ends
- often (but not necessarily) interpreted as black boxes, i.e., no assumptions about their behavior
- actually, for analysis it is often beneficial to take into account the behavior of components (e.g. to detect potential deadlocks or to validate temporal properties) — may be annotated with a specification that reflects its behaviour

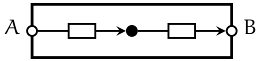
### Write and Take operations

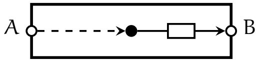


## The synchronisation barrier

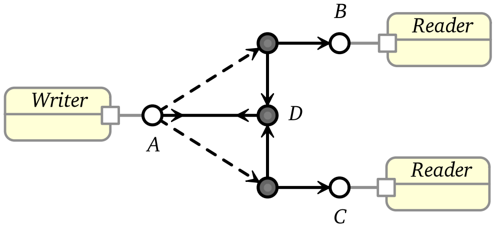


## Fifo vs LossySync



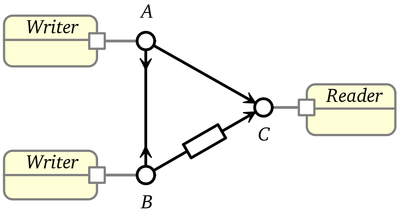


## The exclusive router



- routes data items synchronously from the source to exactly one of the two sinks;
- if both of them are ready to accept data, the choice of where the data item goes is made non-deterministically (merge goes without a priority)

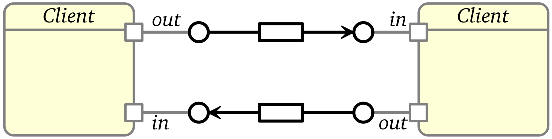
## The alternator



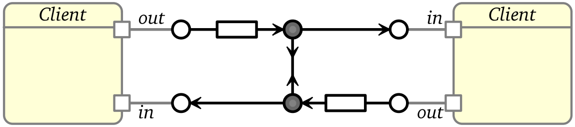
- enforces an ordered output of the data items provided by the two sources
- inputs synchronized through a synchronous drain
- the FIFO1 stores the data item and makes it available in the next execution step; and guarantees alternation (why?)

## Messenger patterns

### Messages exchanged through two buffered channels

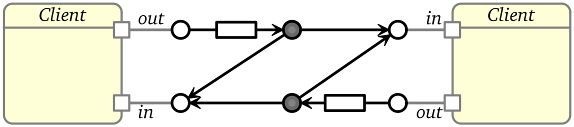


### Message retrievals are synchronized



## Messenger patterns

### Messenger with automatic acknowledgments



Clients get, as an acknowledgment, a copy of their own message when the other client has successfully received it

## Timed data streams

### Time streams

constrained streams over (positive) real numbers, representing moments in time such that

- strictly increasing: $a(i) < a(i+1)$

### Timed data stream

pair $\langle \alpha, a \rangle$ consisting of a data stream $\alpha$ and a time stream $a$, with the interpretation that for for all $i \in \boldsymbol{N}$, the input/output of data item $\alpha(i)$ occurs at time $a(i)$

## Timed data streams

Formally,

$$TDS \;=\; \{\langle \alpha, a\rangle \in Data^{\omega} \times \boldsymbol{R}_{+}^{\omega} | \forall_{i \geq 0} \cdot a(i) < a(i{+}1) \text{ and } \lim_{i \to \infty} a(i) = \infty\}$$

### Notes

- A timed data stream is associated to each connector port
- No distinction between input and output actions

## Timed data streams

### Connectors

are relations over timed data streams:

$$\begin{aligned}
\langle \alpha, a\rangle \, [\![\mathsf{Sync}]\!] \, \langle \beta, b\rangle \ &\Leftrightarrow \ \langle \alpha, a\rangle = \langle \beta, b\rangle \\
\langle \alpha, a\rangle \, [\![\mathsf{FIFO}]\!] \, \langle \beta, b\rangle \ &\Leftrightarrow \ \alpha = \beta \ \wedge \ a < b \\
\langle \alpha, a\rangle \, [\![\mathsf{FIFO}_1]\!] \, \langle \beta, b\rangle \ &\Leftrightarrow \ \alpha = \beta \ \wedge \ a < b < a'
\end{aligned}$$

- coalgebraic semantics [Arbab, Rutten, 2002; Arbab 2003] with incipient calculus
- cannot capture context-awareness

## Constraint automata

Automata labelled by

- a data constraint which represents a set of data assignments to port names

$$g \quad ::= \quad \text{true} \;\mid\; d_A = v \;\mid\; g_1 \vee g_2 \;\mid\; \neg g$$

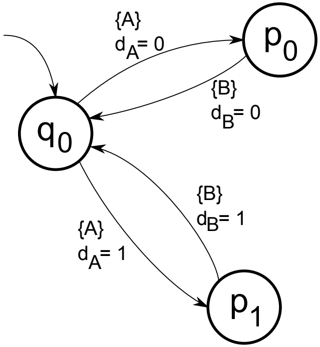
  Note: other constraints, as
  $d_A = d_B \overset{\text{abv}}{=} \bigvee_{d \in Data}(d_A = d \wedge d_B = d)$ are derived.

- a name set which represents the set of port names at which io can occur

States represent the configurations of the corresponding connector, while transitions encode its maximally-parallel stepwise behavior.

## Constraint automata

### Example: Fifo1



## Constraint automata

### Definition

$A = \langle S, N, \rightarrow, S_0 \rangle$

- $S$ is a set of states
- $S_0 \subseteq S$ is the set of initial states
- $N$ is a (finite) set of (port) names
- $\rightarrow$: $S \times \mathcal{P}N \times DC \times S$ such that $s \xrightarrow{P,g} s'$ iff
    1. $P \neq \emptyset$
    2. $g \in DC(P, Data)$
       ($DC(P, Data)$ is the set of data constraints over $Data$ and $P$)

## Constraint automata

### Intuition

$$s \xrightarrow{P,g} s'$$

means that

in configuration $s$ ports in $P$ can perform io operations which meet guard $g$ and lead to $s'$

Conditions

1. $P \neq \emptyset$: transitions fire only if data occurs at a (set of) ports
2. $g \in DC(P, Data)$: behaviour depends only on observed data (not on future evolution)

## Constraint automata

### Intuition

| labelled transition system | (model) reactive system |
|---|---|
| constraint automaton | (model) coordination connector |

Moreover

- act as acceptors for timed stream tuples $t \in TDS^N$
- ... just as finite (infinite) automata accept finite (infinite) words
- but ... there are no final states: accepting runs are always infinite
- the state space may be infinite if modelling a connector with unbounded memory
- as expected: for any constraint automaton there exists a language-equivalent deterministic constraint automaton

## Constraint automata

### Acceptors for timed streams

Given $A$ and $t \in TDS^N$ as its input, find out whether $t$ describes a possible data flow of $A$

- $A$ starts in one of its initial states and waits until data items occur at some of its io ports
- Data occurring at a subset of ports triggers checking the guard
- ... choose a transition with a validated guard
- ... if no data constraint is fulfilled then $A$ rejects $t$

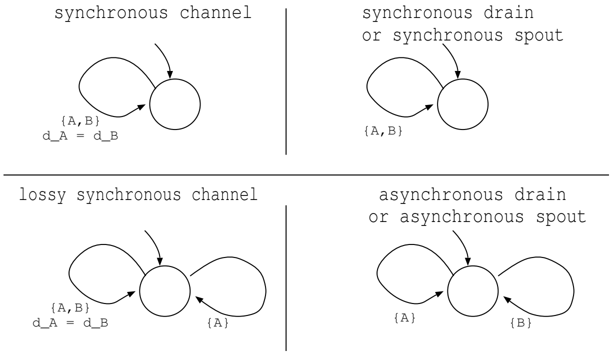
Accepted language is composed by all input streams that have at least one non-rejecting run in $A$

## Constraint automata as a semantics for Reo

- cannot capture context-awareness [Baier, Sirjani, Arbab, Rutten 2006], but forms the basis for more elaborated models (eg, Reo automata)
- captures all behavior alternatives of a connector; useful to generate a state-machine implementing the connector's behavior
- basis for several tools, including the model checker Vereofy [Kluppelholz, Baier 2007]

## Constraint automata as a semantics for Reo

### Examples

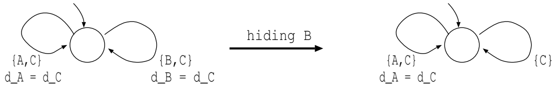


## Constraint automata as a semantics for Reo

### Connector construction

Connector operators are modelled by typical automata constructions

- join
- hide: hiding a node means that its data flow is no longer externally observable:

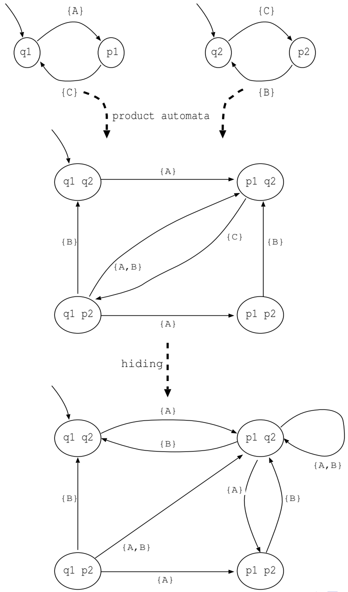


## Constraint automata as a semantics for Reo

### Connector construction

A 2-bounded FIFO obtained from two 1-bounded FIFO channels via product and hiding (assume $Data = \{d\}$ for simplicity)

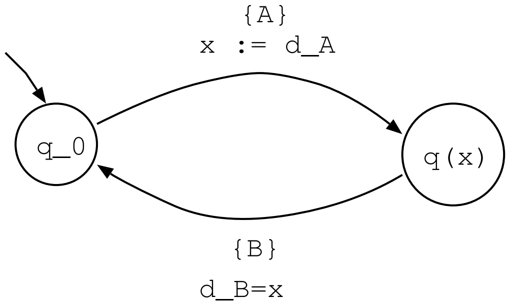
## Constraint automata as a semantics for Reo



## Parametrized constraint automata

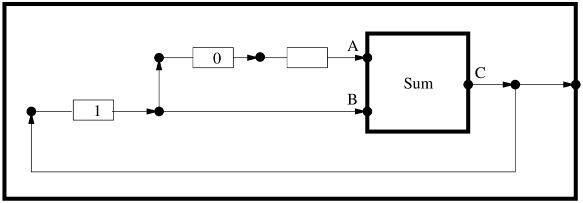
States are parametric on data values ... therefore capturing complex constraint automata emerging form data-dependencies

Example: 1 bounded FIFO



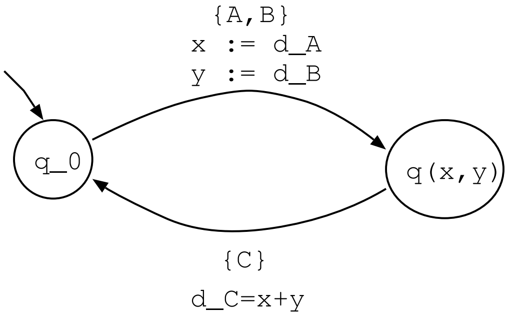
## Parametrized constraint automata

### Example: Fibonacci generator



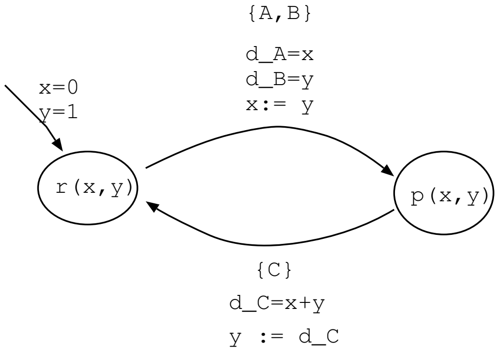
## Parametrized constraint automata

### Example: Fibonacci generator (Sum)



## Parametrized constraint automata

### Example: Fibonacci generator (complete)



## Constraint automata: bisimulation

### Definition

A bisimulation on a constraint automata $A = \langle S, N, \rightarrow, S_0 \rangle$ is an equivalence relation $R$ on $S$ such that for all pairs $\langle s, s' \rangle$, all $R$-induced equivalence classes $P \in S/R$ and every $Ns \subseteq N$,

$$dc(s, Ns, P) \; = \; dc(s', Ns, P)$$

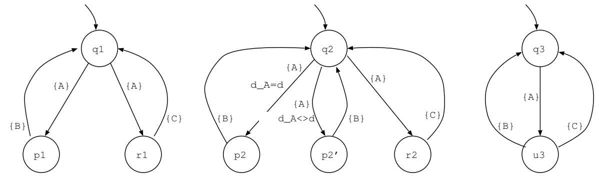
where

$$dc(s, Ns, P) \; = \; \bigvee \{ g \mid s \xrightarrow{Ns,g} s' \text{ for some } s' \in P \; \}$$

i.e., the weakest data constraint ensuring a $Ns$-transition from $s$ to a state in $P$.

## Constraint automata: bisimulation

### Example



The equivalence $R$ induced by partition

$$S/R = \{\{q_1, q_2\}, \{q_3\}, \{p_1, p_2, p_2'\}, \{r_1, r_2\}, \{u_3\}\}$$

is a bisimulation. Why?

## Constraint automata: bisimulation

### Bisimilarity

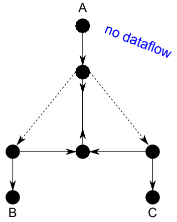
As usual, two sates are bisimilar if contained in a bisimulation.

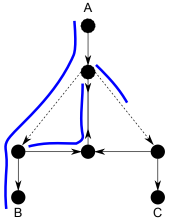
### Theorem

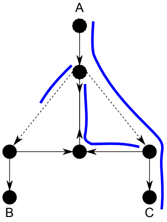
Bisimilarity is strictly finer than language equivalence (TDS acceptance), but for deterministic automata for which they coincide.

## Coulorings

Based on the set of all of dataflow alternatives of the connector, represented by different colours meaning data flowing and no data flowing

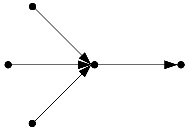






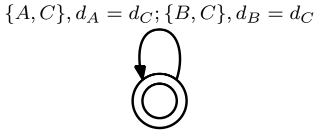
## Merger

### Reo circuit



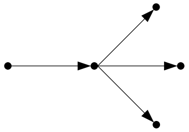
### Semantics

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



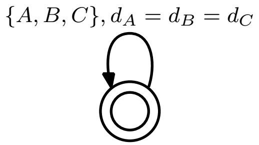
## Replicator

### Reo circuit



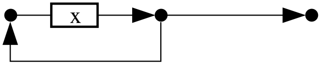
### Semantics

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



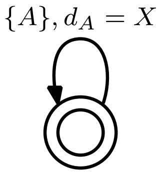
## Feedback loop

### Reo circuit



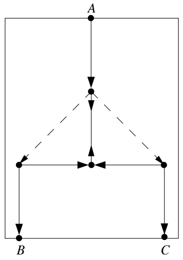
### Semantics

$$FL_X(\langle \alpha, a \rangle) \equiv \alpha(0) = X \ \wedge \ FL_X(\langle \alpha', a' \rangle)$$



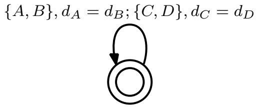
## Exclusive router

### Reo circuit



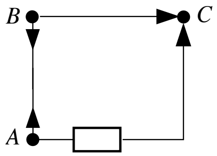
### Semantics

$$
\begin{array}{l}
ExR(\langle\alpha, a\rangle; \langle\beta, b\rangle, \langle\gamma, c\rangle) \equiv \\
\quad b(0) \neq c(0) \;\wedge \\
\quad \begin{cases} \alpha(0) = \beta(0) \;\wedge\; a(0) = b(0) \;\wedge\; ExR(\langle\alpha', a'\rangle, \langle\beta', b'\rangle, \langle\gamma, c\rangle) & \text{if } b(0) < c(0) \\ \alpha(0) = \gamma(0) \;\wedge\; a(0) = c(0) \;\wedge\; ExR(\langle\alpha', a'\rangle, \langle\beta, b\rangle, \langle\gamma', c'\rangle) & \text{if } c(0) < b(0) \end{cases}
\end{array}
$$



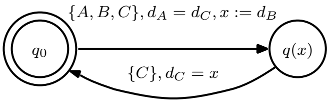
# Ordering

## Reo circuit



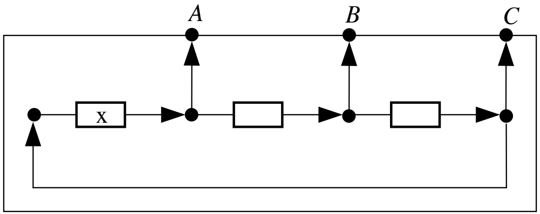
## Semantics

$$
\begin{aligned}
&OC(\langle \alpha, a\rangle, \langle \beta, b\rangle; \langle \gamma, c\rangle) \equiv \\
&\quad \alpha(0) = \gamma(0) \ \wedge \ \beta(0) = \gamma(1) \ \wedge \ a(0) = b(0) = c(0) \ \wedge \ a(1) = b(1) > c(1) \ \wedge \\
&\quad OC(\langle \alpha', a'\rangle, \langle \beta', b'\rangle; \langle \gamma'', c''\rangle)
\end{aligned}
$$



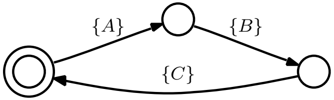
# Sequencer

## Reo circuit



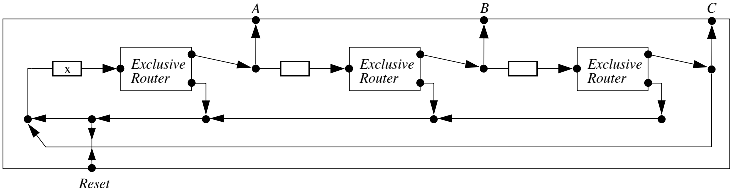
## Semantics

$$SQ(;\langle\alpha,a\rangle,\langle\beta,b\rangle,\langle\gamma,c\rangle) \quad\equiv\quad a<b<c<a'$$

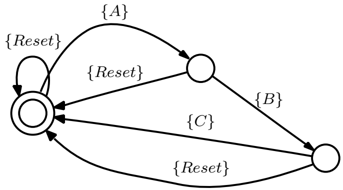


# Sequencer with reset

## Reo circuit

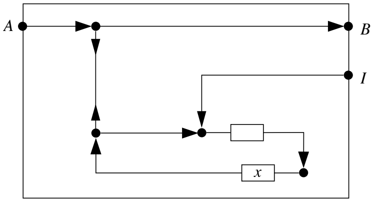


## Semantics



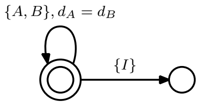
## Inhibitor

### Reo circuit



### Semantics

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



## Concluding

- tools ... & case studies
- several semantic models ... & incipient calculus
- extensions: timed, stochastic, QoS annotated