Asynchronous Distributed Automata

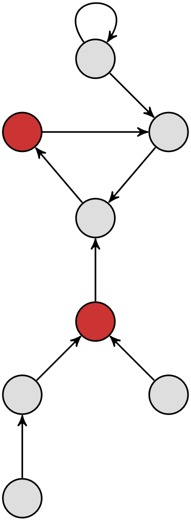
Fabian Reiter

IRIF, Université Paris Diderot

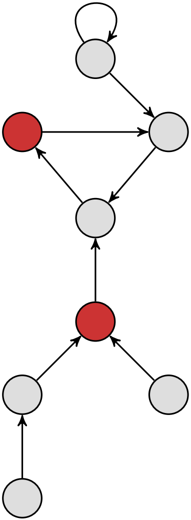
July 10, 2017

The backward μ-fragment

The backward μ-fragment

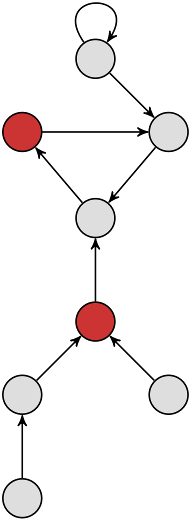


The backward μ-fragment



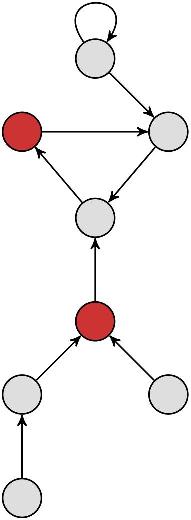
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \Bigg(\qquad\qquad \Bigg)$$

The backward μ-fragment



$$\mu\begin{pmatrix} X \\ Y \end{pmatrix}.\begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \end{pmatrix}$$

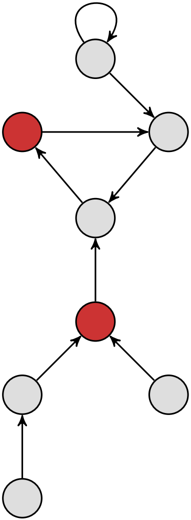
The backward μ-fragment



constant

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \\ \end{pmatrix}$$

The backward μ-fragment

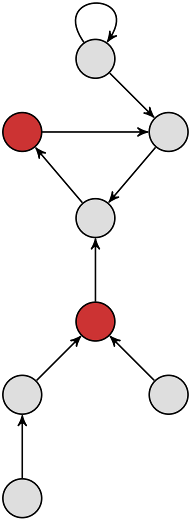


constant

unnegated variable

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \left((R \wedge Y) \vee \bar{\Diamond} X \right)$$

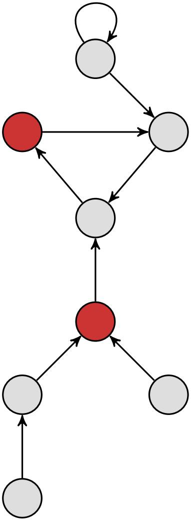
The backward μ-fragment



constant — unnegated variable — $\exists$ incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \\ \end{pmatrix}$$

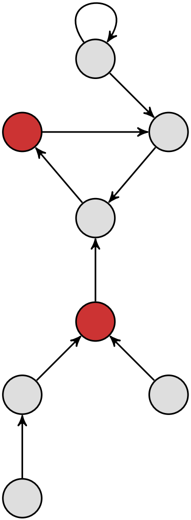
The backward μ-fragment



constant — unnegated variable — $\exists$ incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} ({\color{red}R} \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$

The backward μ-fragment



constant

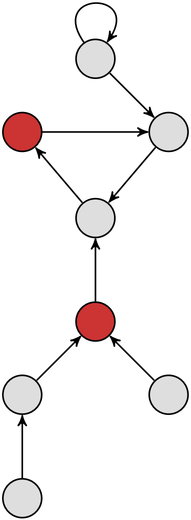
unnegated variable

$\exists$ incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box}\, Y \end{pmatrix}$$

$\forall$ incoming neighbors

The backward μ-fragment



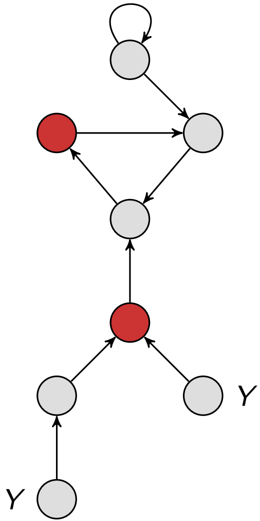
constant — unnegated variable — $\exists$ incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \land Y) \lor \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$

$\forall$ incoming neighbors

Compute the simultaneous least fixpoint.

The backward μ-fragment



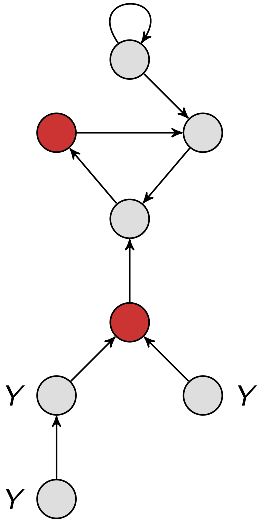
constant — unnegated variable — $\exists$ incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} ({\color{red}R} \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$

$\forall$ incoming neighbors

Compute the simultaneous least fixpoint.

The backward μ-fragment



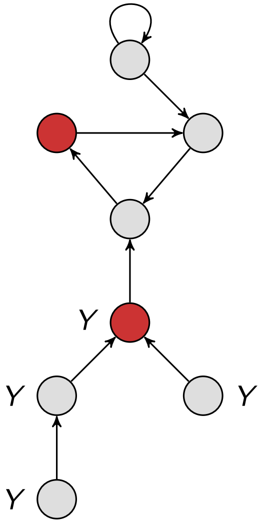
constant — unnegated variable — $\exists$ incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (\textcolor{red}{R} \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$

$\forall$ incoming neighbors

Compute the simultaneous least fixpoint.

The backward μ-fragment



constant

unnegated variable

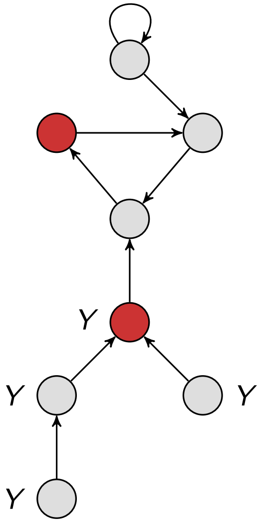
$\exists$ incoming neighbor

$$\mu\begin{pmatrix}X\\ Y\end{pmatrix}.\begin{pmatrix}({\color{red}R}\wedge Y)\vee \bar{\Diamond} X\\ \bar{\Box}\, Y\end{pmatrix}$$

$\forall$ incoming neighbors

Compute the simultaneous least fixpoint.

The backward μ-fragment



constant → R; unnegated variable → Y; $\exists$ incoming neighbor → $\bar{\Diamond}X$

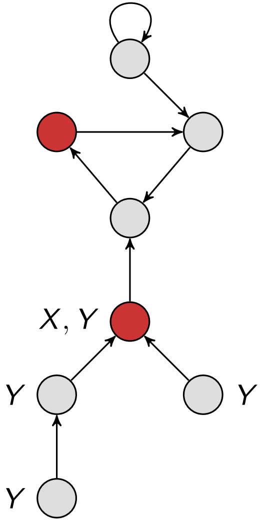
$$\mu\begin{pmatrix}X\\ Y\end{pmatrix}.\begin{pmatrix}({\color{red}R}\wedge Y)\vee\bar{\Diamond}X\\ \bar{\Box}\,Y\end{pmatrix}$$

$\forall$ incoming neighbors → $\bar{\Box}\,Y$

Compute the simultaneous least fixpoint.

Y: "Going backwards, we cannot reach any directed cycle (only dead-ends)."

The backward μ-fragment



constant → R; unnegated variable → Y; $\exists$ incoming neighbor → $\bar{\Diamond}$

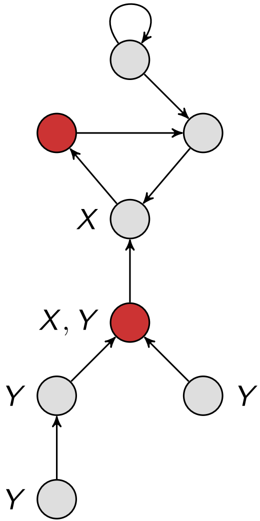
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box}\, Y \end{pmatrix}$$

$\forall$ incoming neighbors → $\bar{\Box}$

Compute the simultaneous least fixpoint.

Y: "Going backwards, we cannot reach any directed cycle (only dead-ends)."

The backward μ-fragment



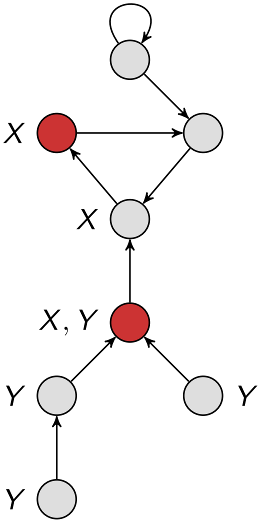
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \land Y) \lor \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$

constant: R — unnegated variable: Y — $\exists$ incoming neighbor: $\bar{\Diamond}$ — $\forall$ incoming neighbors: $\bar{\Box}$

Compute the simultaneous least fixpoint.

Y: "Going backwards, we cannot reach any directed cycle (only dead-ends)."

The backward μ-fragment



constant

unnegated variable

$\exists$ incoming neighbor

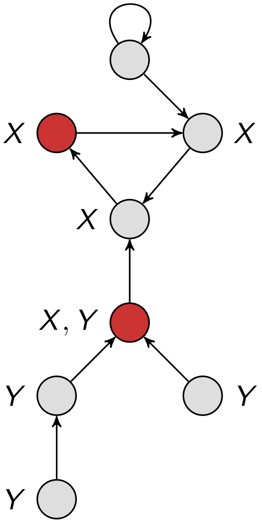
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box}\, Y \end{pmatrix}$$

$\forall$ incoming neighbors

Compute the simultaneous least fixpoint.

Y: "Going backwards, we cannot reach any directed cycle (only dead-ends)."

The backward μ-fragment



constant · unnegated variable · $\exists$ incoming neighbor

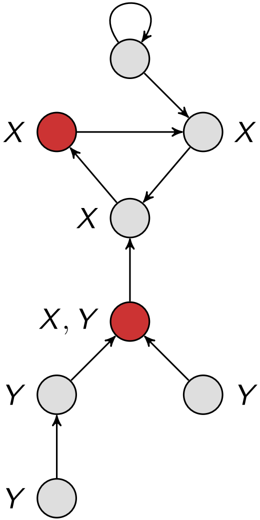
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box}\, Y \end{pmatrix}$$

$\forall$ incoming neighbors

Compute the simultaneous least fixpoint.

Y: "Going backwards, we cannot reach any directed cycle (only dead-ends)."

The backward μ-fragment



constant — unnegated variable — $\exists$ incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$

$\forall$ incoming neighbors

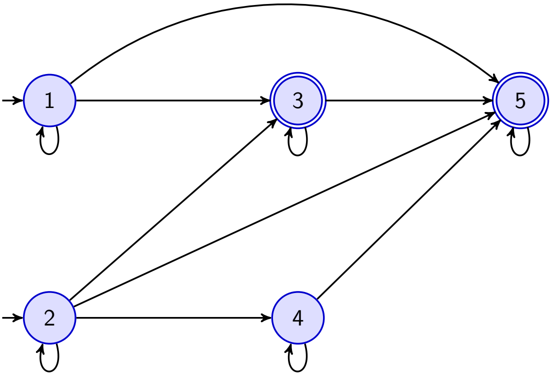
Compute the simultaneous least fixpoint.

Y: “Going backwards, we cannot reach any directed cycle (only dead-ends).”

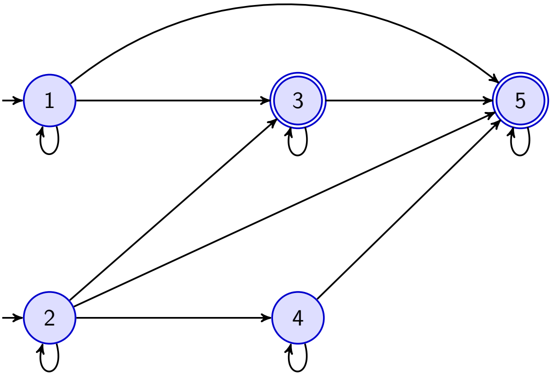
X: “Going backwards, we can reach a red node from which no directed cycle is reachable.”

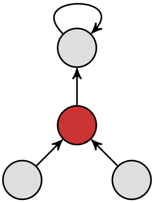
Distributed automata

Distributed automata

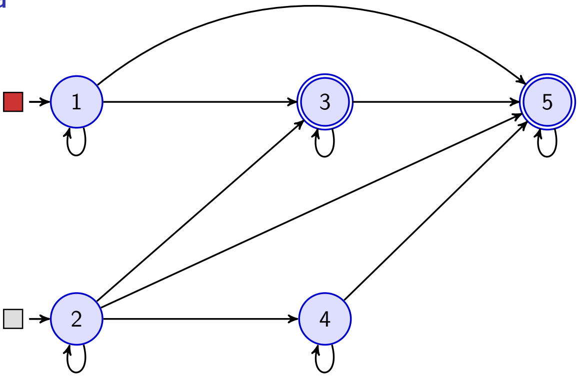


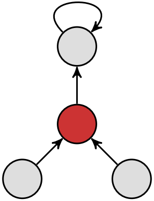
Distributed automata



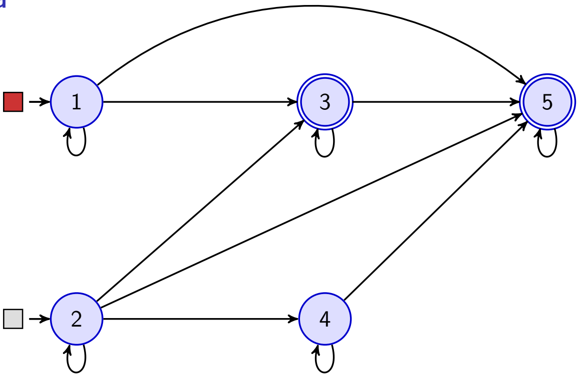


Distributed automata

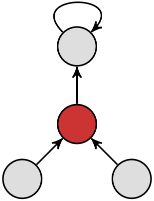




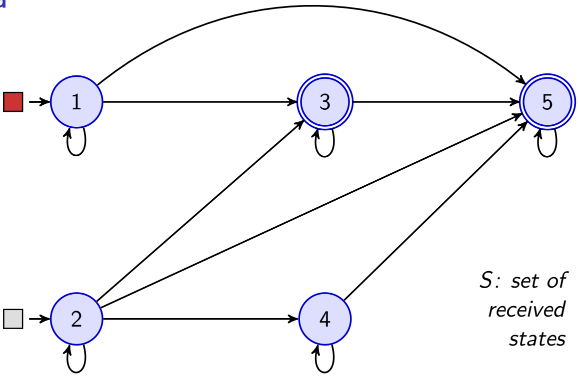
Distributed automata



Transition function:
$\delta\colon Q \times 2^Q \to Q$
(Q: set of states)

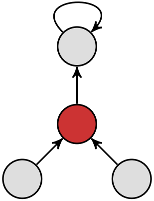


Distributed automata

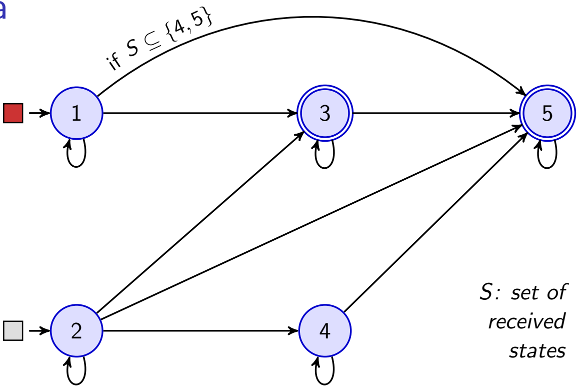


Transition function:
$\delta\colon Q \times 2^Q \to Q$
(Q: set of states)

S: set of received states

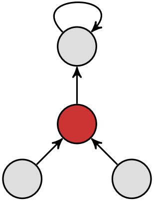


Distributed automata

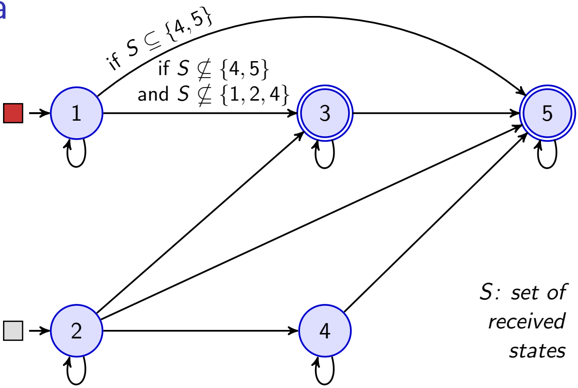


Transition function:
$\delta\colon Q \times 2^Q \to Q$
(Q: set of states)

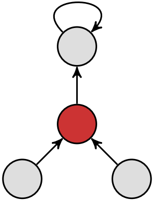
S: set of received states



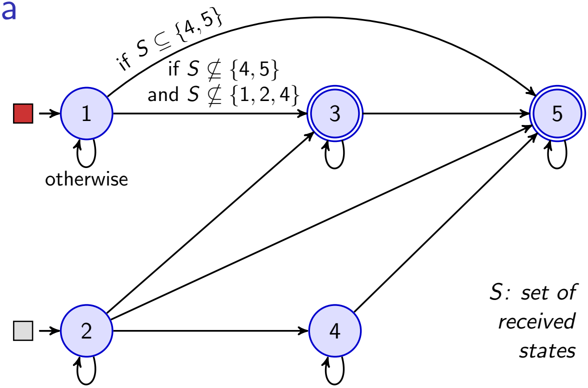
Distributed automata



Transition function:
$\delta\colon Q \times 2^Q \to Q$
(Q: set of states)



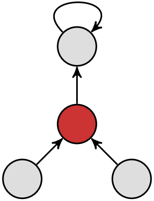
Distributed automata



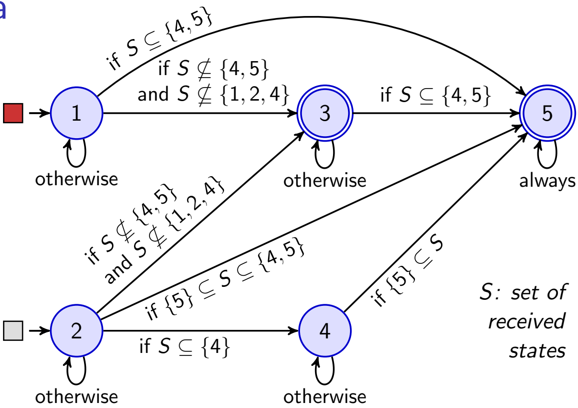
Transition function:

$\delta\colon Q \times 2^Q \to Q$

(Q: set of states)



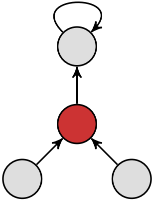
Distributed automata



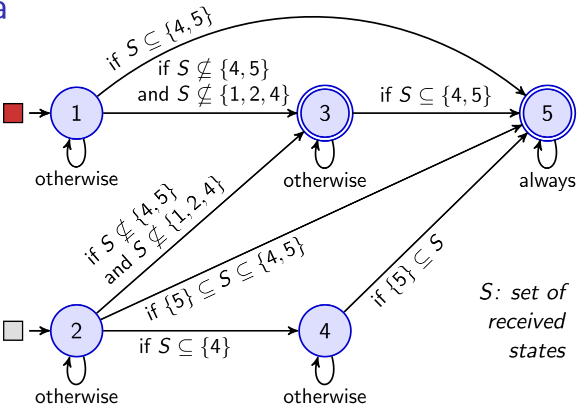
Transition function:

$\delta\colon Q \times 2^Q \to Q$

(Q: set of states)



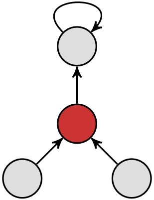
Distributed automata



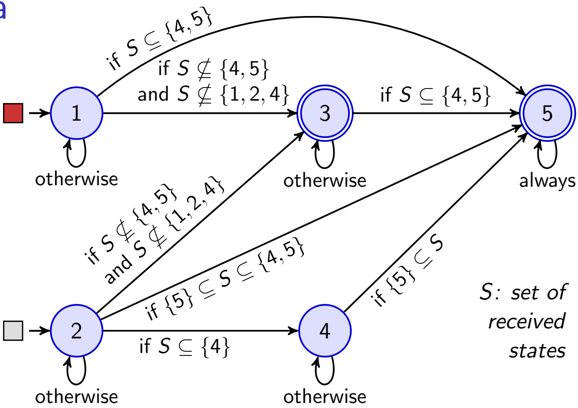
Transition function:
$\delta\colon Q \times 2^Q \to Q$
(Q: set of states)

S: set of received states

Synchronous run:



Distributed automata

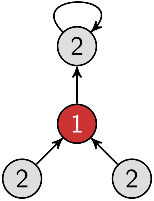


Transition function:

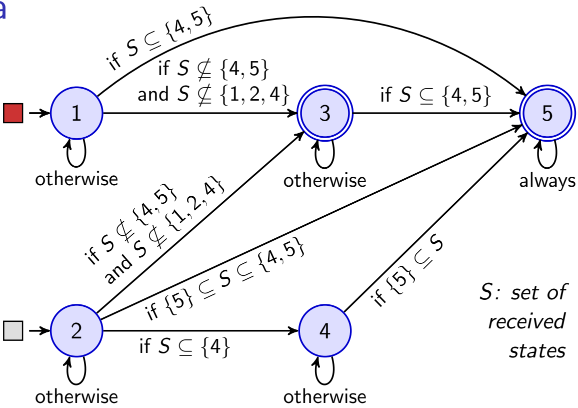
$\delta\colon Q \times 2^Q \to Q$

(Q: set of states)

Synchronous run:



Distributed automata

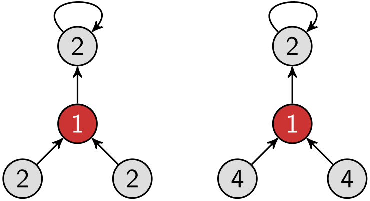


Transition function:

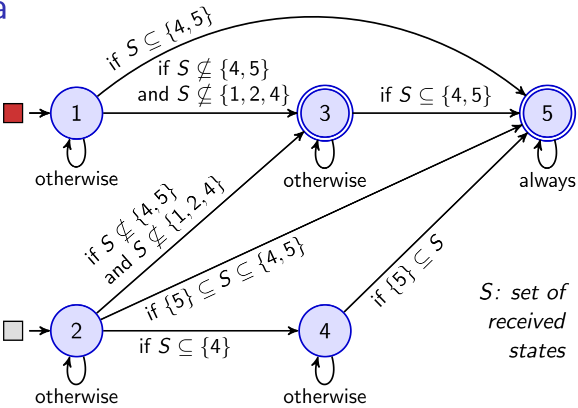
$\delta \colon Q \times 2^Q \to Q$

(Q: set of states)

Synchronous run:



Distributed automata

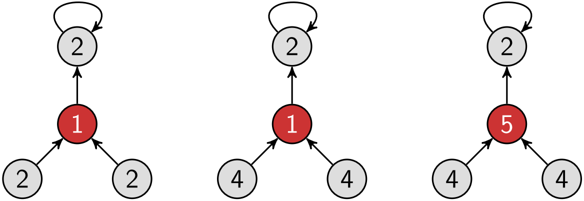


Transition function:

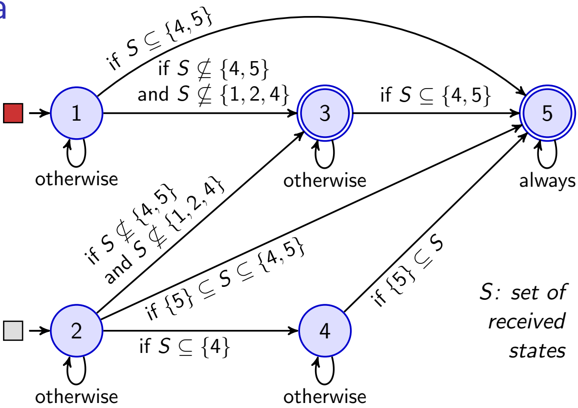
$\delta\colon Q \times 2^Q \to Q$

(Q: set of states)

Synchronous run:



Distributed automata

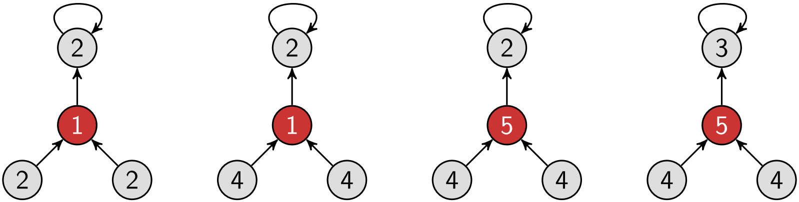


Transition function:

$\delta\colon Q \times 2^Q \to Q$

(Q: set of states)

Synchronous run:



Powerset construction

Powerset construction

(Antti Kuusisto, 2013)

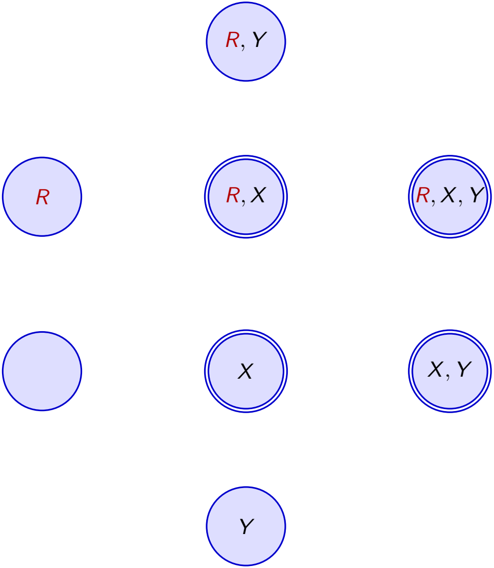
Powerset construction

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$

(Antti Kuusisto, 2013)

Powerset construction

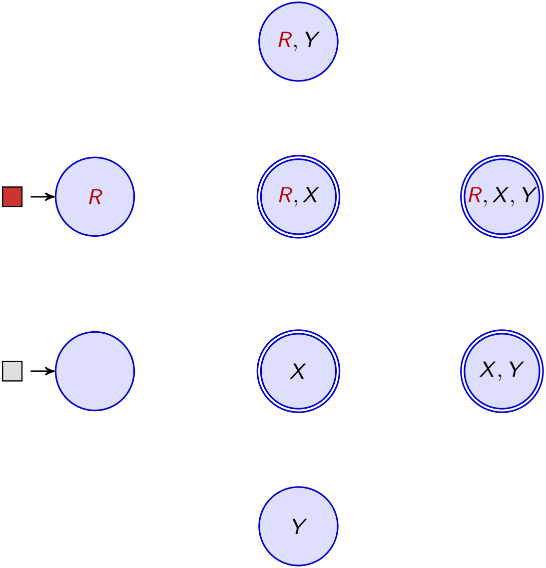
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

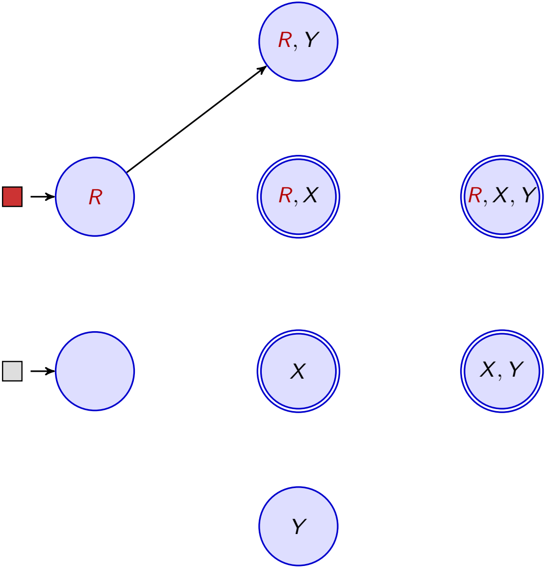
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

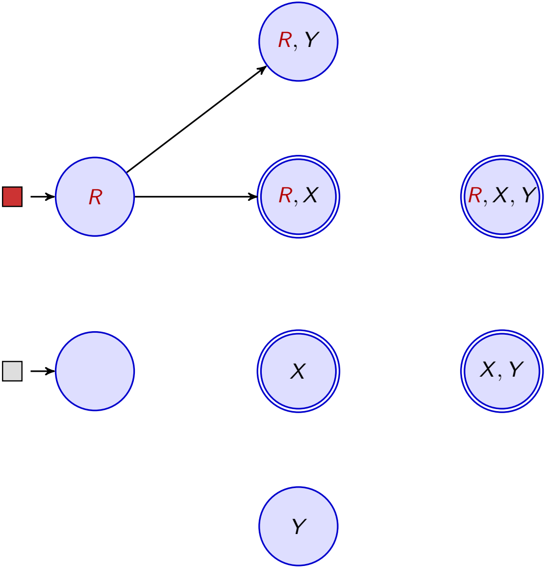
$$\mu\begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

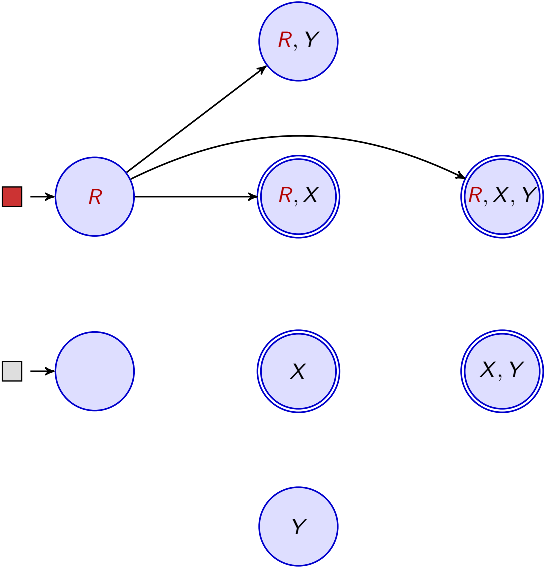
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

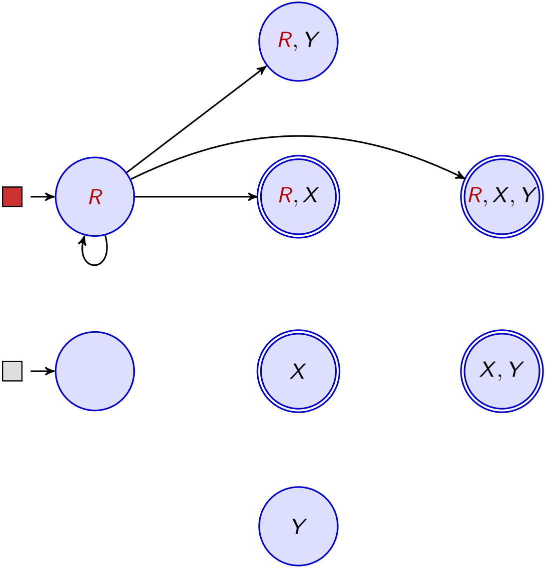
$$\mu\begin{pmatrix}X\\Y\end{pmatrix}.\begin{pmatrix}(R \wedge Y) \vee \bar{\Diamond} X\\ \bar{\Box} Y\end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

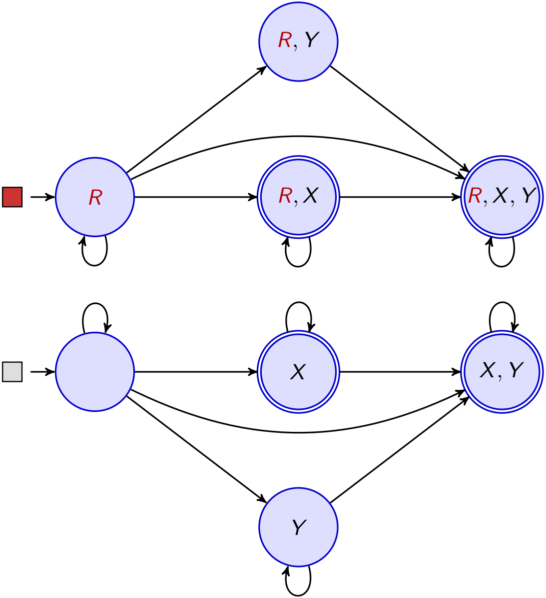
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

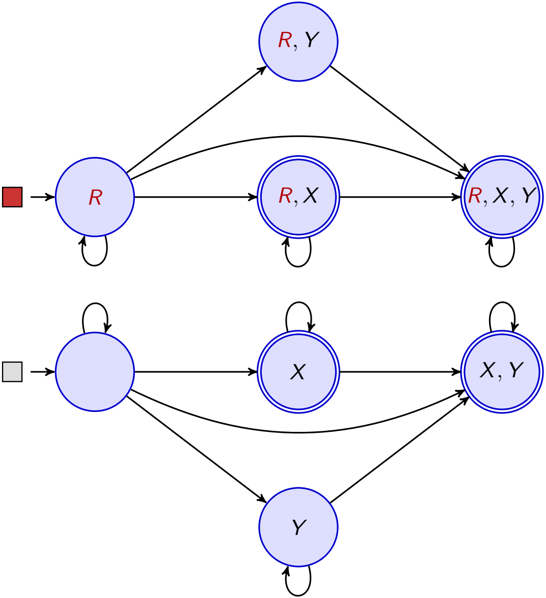
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} . \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



Quasi-acyclic diagram (self-loops are allowed).

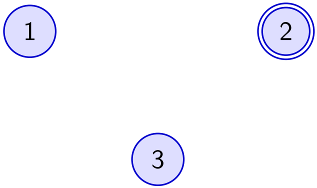
(Antti Kuusisto, 2013)

Synchrony is too powerful

Synchrony is too powerful

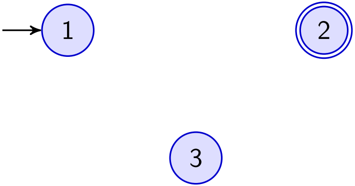
(Antti Kuusisto, 2013)

Synchrony is too powerful



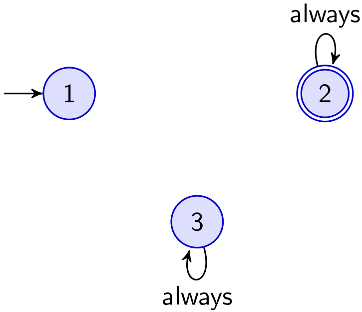
(Antti Kuusisto, 2013)

Synchrony is too powerful



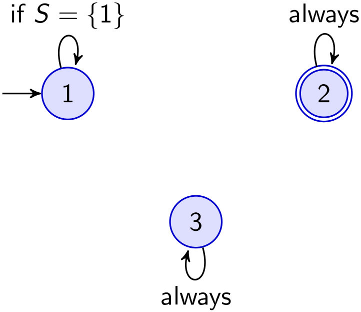
(Antti Kuusisto, 2013)

Synchrony is too powerful



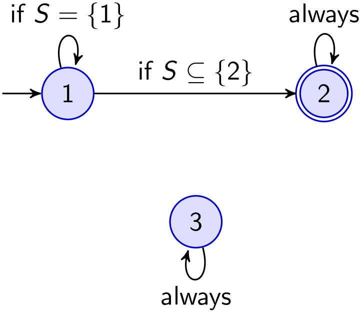
(Antti Kuusisto, 2013)

Synchrony is too powerful



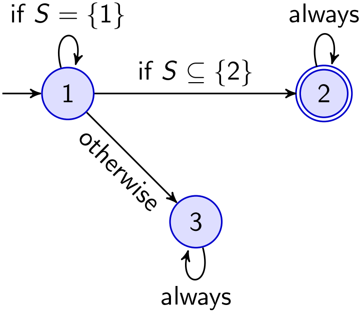
(Antti Kuusisto, 2013)

Synchrony is too powerful



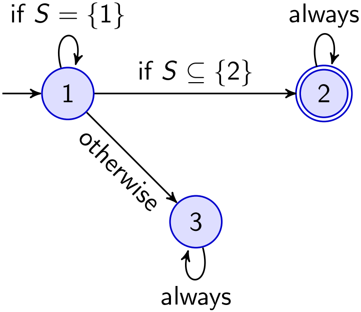
(Antti Kuusisto, 2013)

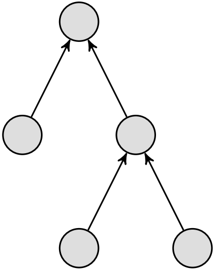
Synchrony is too powerful



(Antti Kuusisto, 2013)

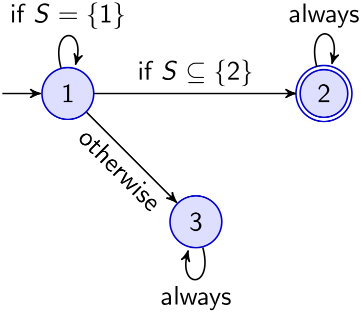
Synchrony is too powerful

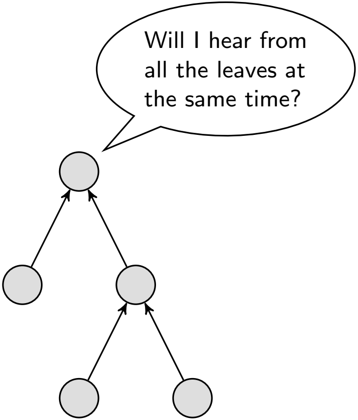




(Antti Kuusisto, 2013)

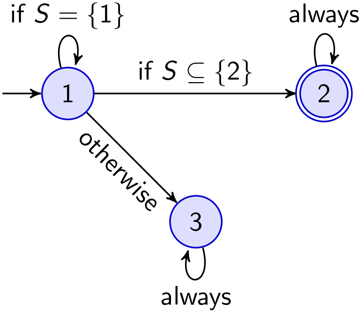
Synchrony is too powerful

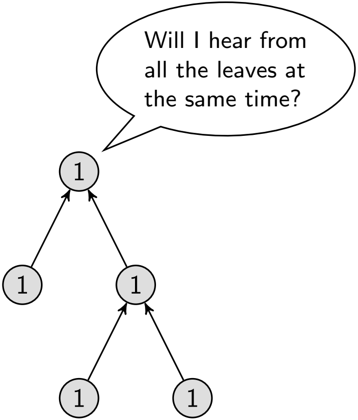




(Antti Kuusisto, 2013)

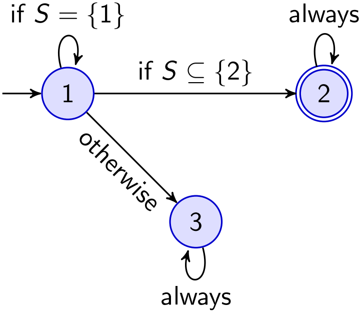
Synchrony is too powerful

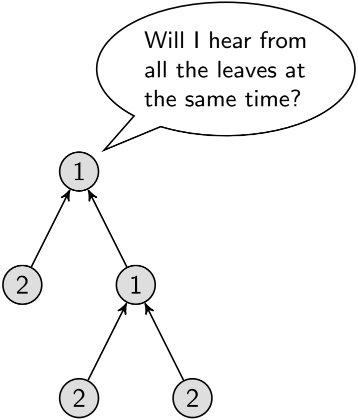




(Antti Kuusisto, 2013)

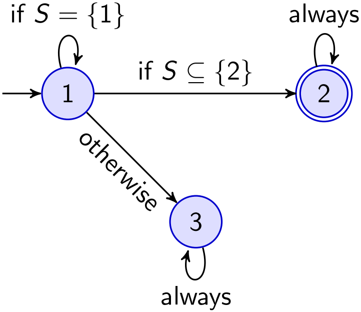
Synchrony is too powerful

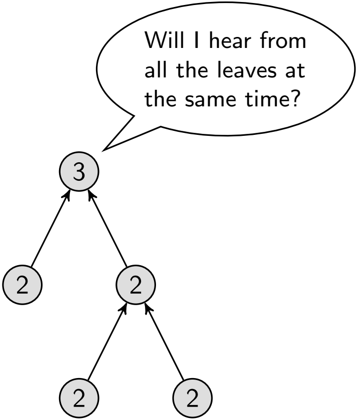




(Antti Kuusisto, 2013)

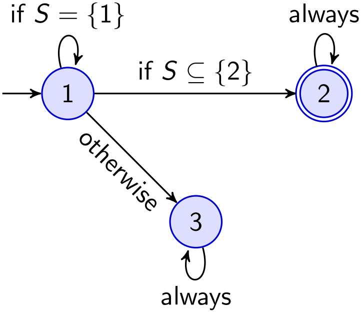
Synchrony is too powerful

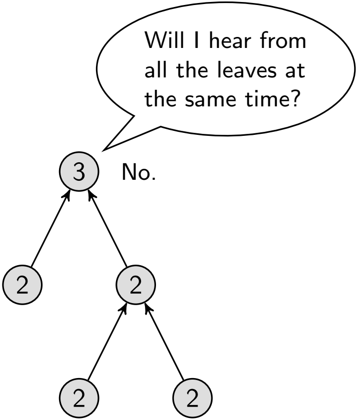




(Antti Kuusisto, 2013)

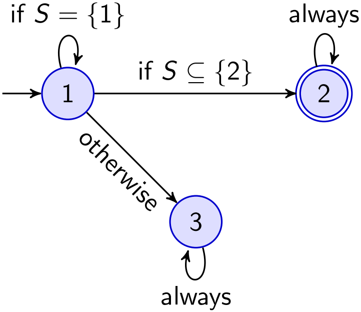
Synchrony is too powerful

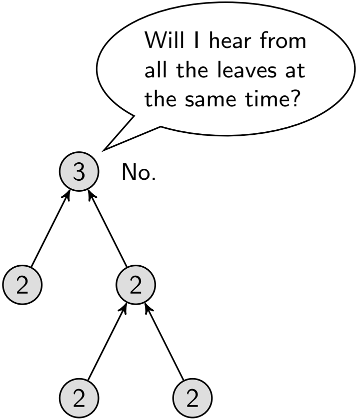




(Antti Kuusisto, 2013)

Synchrony is too powerful



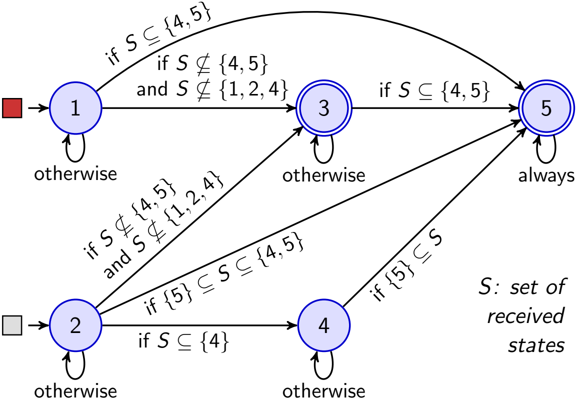


Not even expressible in monadic second-order logic (MSO)!

(Antti Kuusisto, 2013)

Asynchronous run

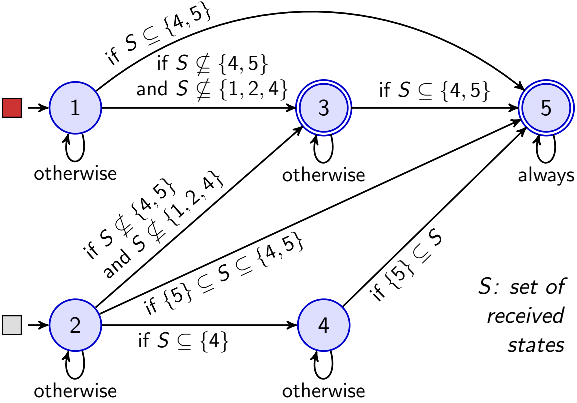
Asynchronous run



S: set of received states

Asynchronous run

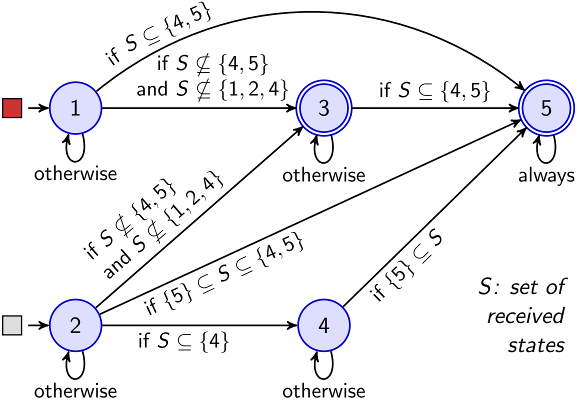
Nodes may sleep,
miss messages.



Asynchronous run

Nodes may sleep, miss messages.

Messages may be delayed (FIFO).

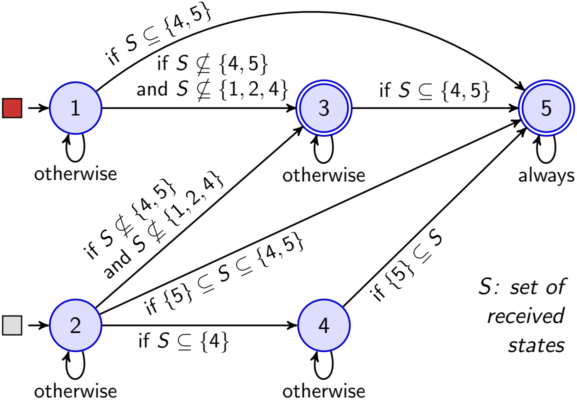


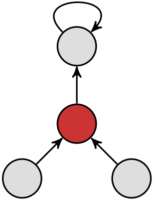
S: set of received states

Asynchronous run

Nodes may sleep, miss messages.

Messages may be delayed (FIFO).

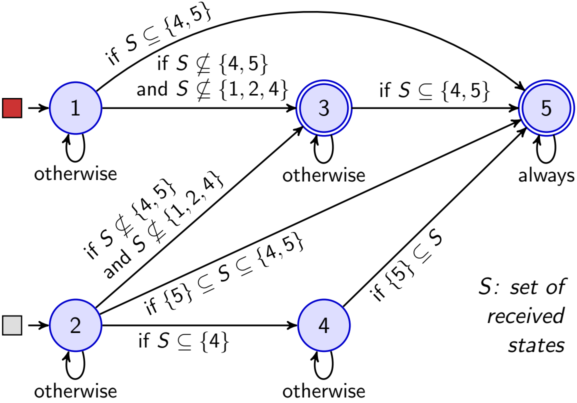


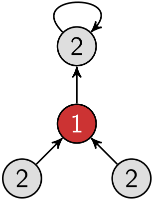


Asynchronous run

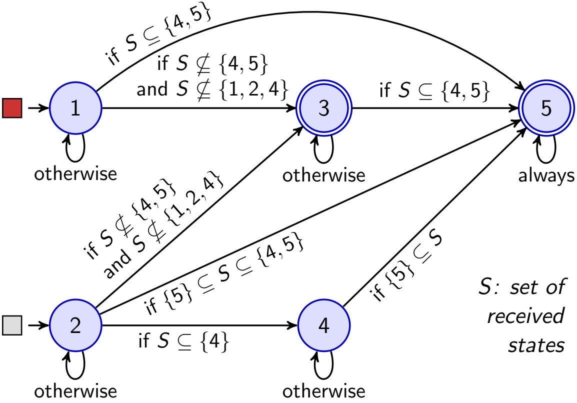
Nodes may sleep, miss messages.

Messages may be delayed (FIFO).





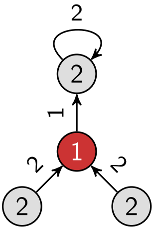
Asynchronous run



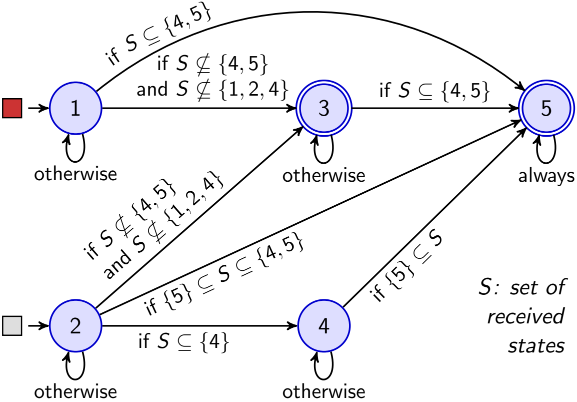
Nodes may sleep, miss messages.

Messages may be delayed (FIFO).

S: set of received states



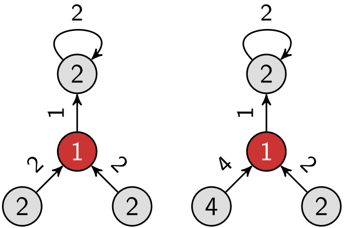
Asynchronous run



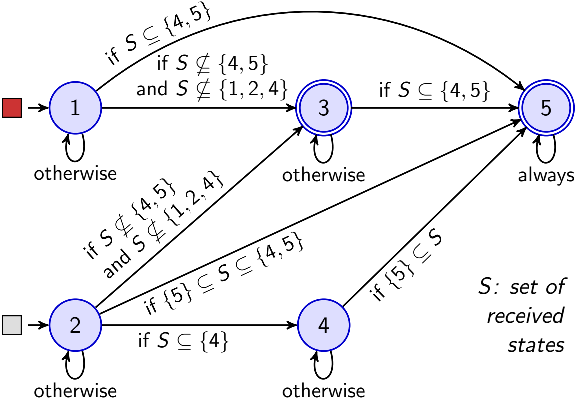
Nodes may sleep, miss messages.

Messages may be delayed (FIFO).

S: set of received states

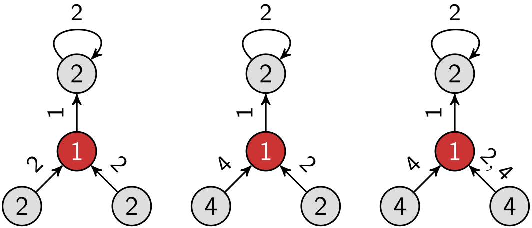


Asynchronous run



Nodes may sleep, miss messages.

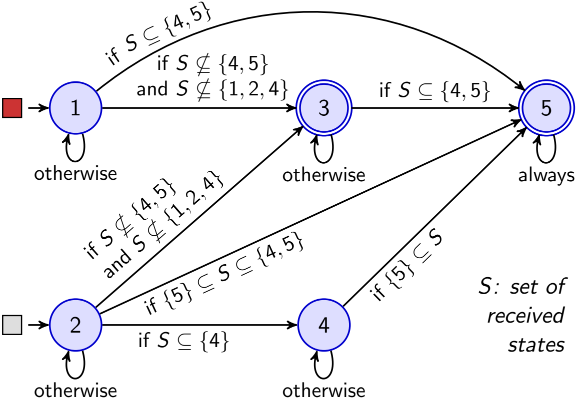
Messages may be delayed (FIFO).



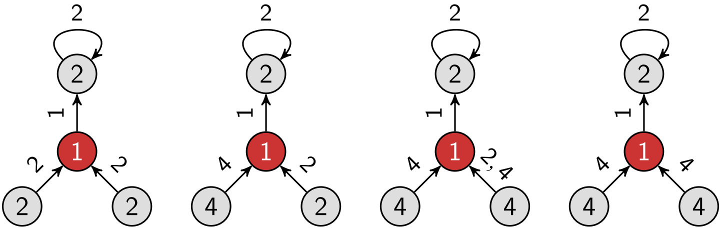
Asynchronous run

Nodes may sleep, miss messages.

Messages may be delayed (FIFO).



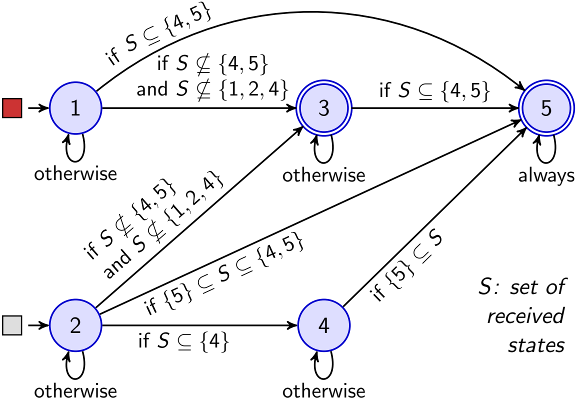
S: set of received states



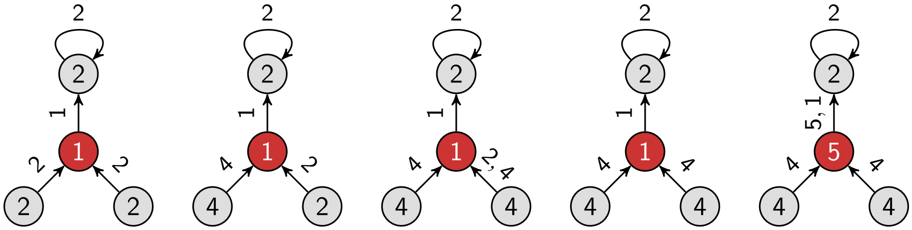
Asynchronous run

Nodes may sleep, miss messages.

Messages may be delayed (FIFO).



S: set of received states



Asynchronous automata

Asynchronous automata

A malicious adversary can choose the timing, subject to fairness constraints.

Asynchronous automata

A malicious adversary can choose the timing, subject to fairness constraints.

Asynchronous automata

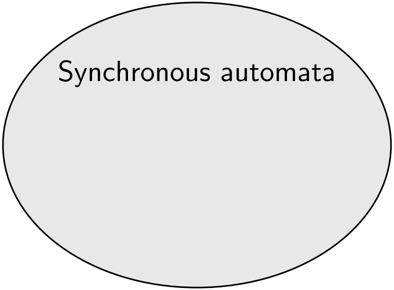
A malicious adversary can choose the timing, subject to fairness constraints.

An automaton is asynchronous if its acceptance behavior is independent of the adversary (on all finite digraphs).

Asynchronous automata

A malicious adversary can choose the timing, subject to fairness constraints.

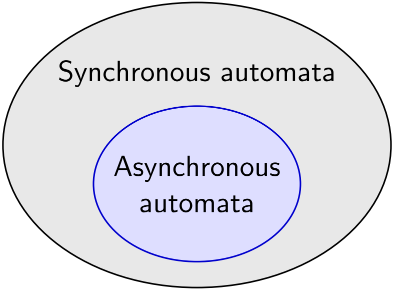
An automaton is asynchronous if its acceptance behavior is independent of the adversary (on all finite digraphs).



Asynchronous automata

A malicious adversary can choose the timing, subject to fairness constraints.

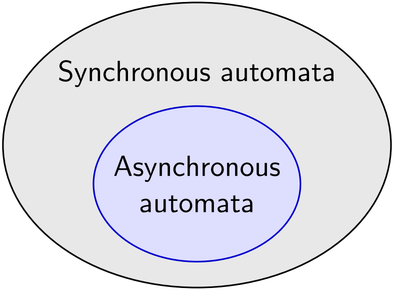
An automaton is asynchronous if its acceptance behavior is independent of the adversary (on all finite digraphs).



Asynchronous automata

A malicious adversary can choose the timing, subject to fairness constraints.

An automaton is asynchronous if its acceptance behavior is independent of the adversary (on all finite digraphs).



Asynchrony is an additional semantic property.

Main result

Theorem

On finite digraphs, the backward μ-fragment is effectively equivalent to quasi-acyclic asynchronous automata.

Main result

Theorem

On finite digraphs, the backward μ-fragment is effectively equivalent to quasi-acyclic asynchronous automata.

Open question: Is quasi-acyclicity really necessary?

Thank you!

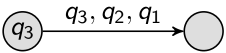
A little bonus

A little bonus

Lossless asynchrony
(weaker adversary):

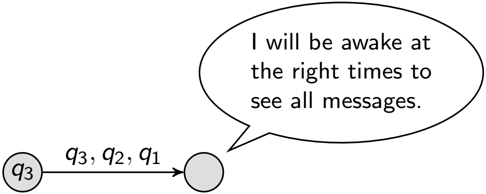
A little bonus

Lossless asynchrony
(weaker adversary):



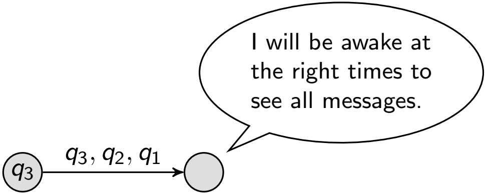
A little bonus

Lossless asynchrony (weaker adversary):



A little bonus

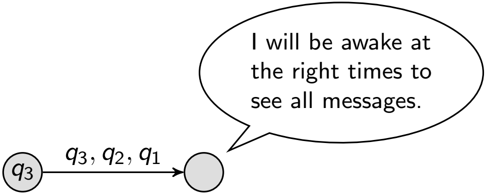
Lossless asynchrony
(weaker adversary):



synchronous

A little bonus

Lossless asynchrony
(weaker adversary):



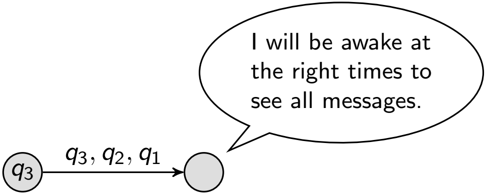
synchronous

↑

lossless-asynchronous

A little bonus

Lossless asynchrony
(weaker adversary):



synchronous

↑

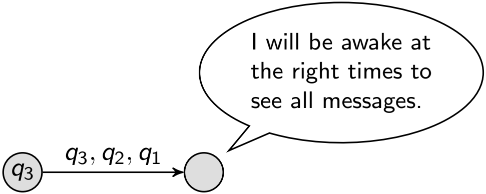
lossless-asynchronous

↑

asynchronous

A little bonus

Lossless asynchrony
(weaker adversary):



synchronous ⟵ quasi-acyclic synchronous

↑

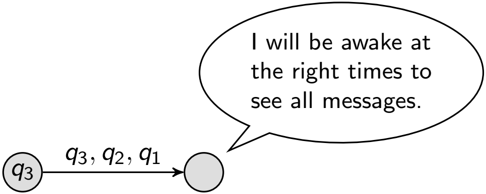
lossless-asynchronous

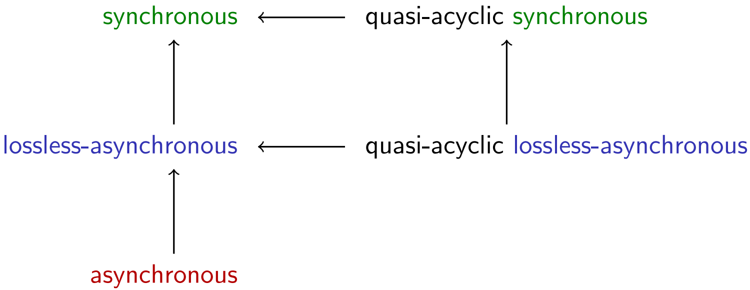
↑

asynchronous

A little bonus

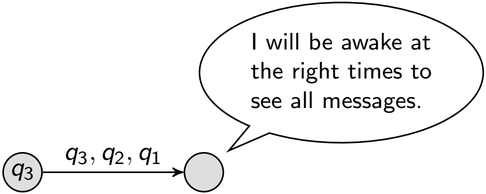
Lossless asynchrony
(weaker adversary):

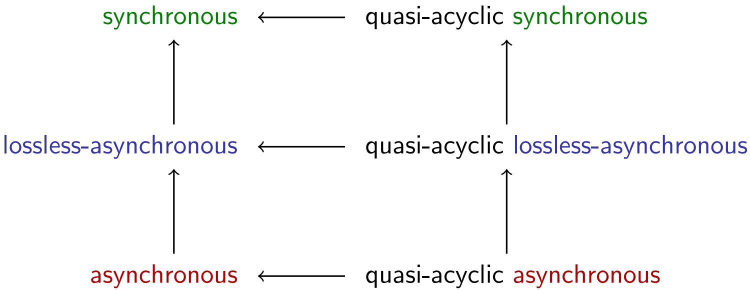




A little bonus

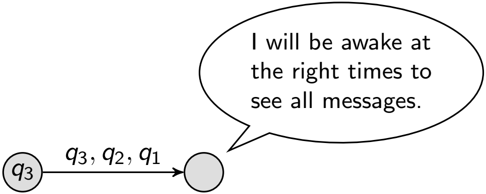
Lossless asynchrony
(weaker adversary):

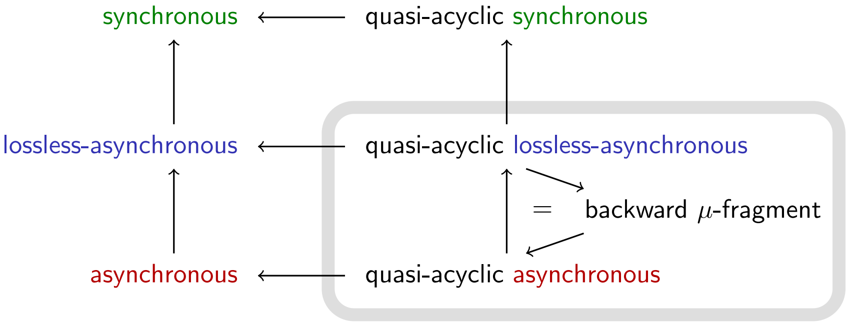




A little bonus

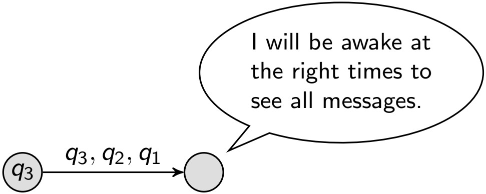
Lossless asynchrony
(weaker adversary):

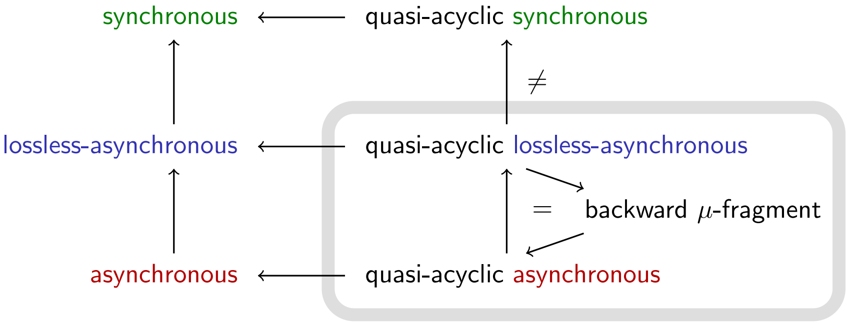




A little bonus

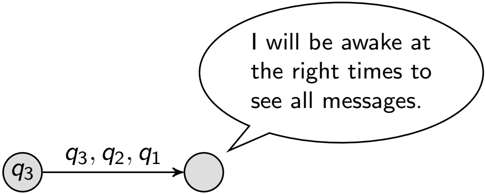
Lossless asynchrony
(weaker adversary):

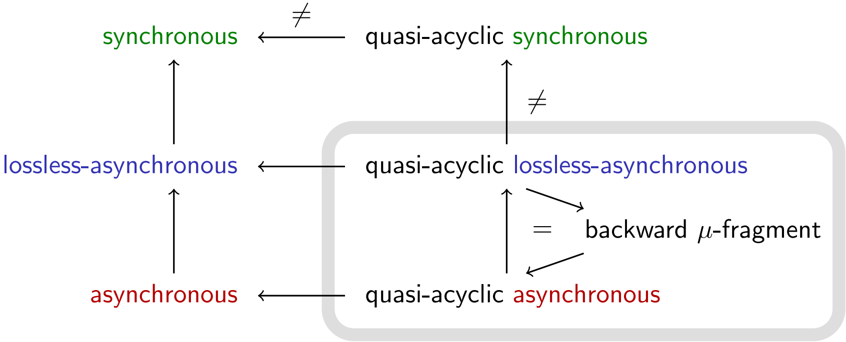




A little bonus

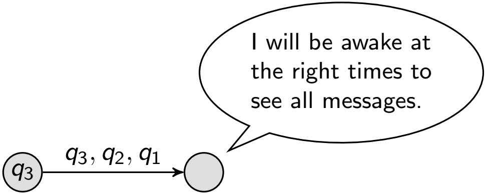
Lossless asynchrony (weaker adversary):

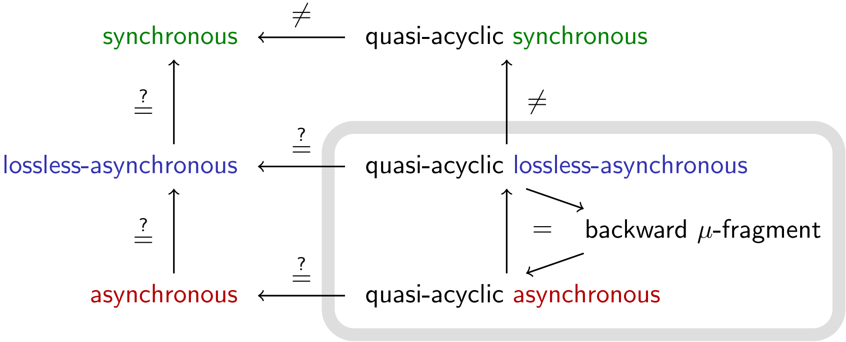




A little bonus

Lossless asynchrony
(weaker adversary):





Asynchronous run (formal version)

Asynchronous run (formal version)

- *Labeled digraph:* $G = (V, E, \lambda)$, where $\lambda \colon V \to \text{Labels}$.

Asynchronous run (formal version)

- *Labeled digraph:* $G = (V, E, \lambda)$, where $\lambda\colon V \to \text{Labels}$.
- *Distributed automaton:* $A = (Q, \delta_0, \delta, F)$, where $\delta_0\colon \text{Labels} \to Q$, $\delta\colon Q \times 2^Q \to Q$ and $F \subseteq Q$.

Return

Asynchronous run (formal version)

- *Labeled digraph:* $G = (V, E, \lambda)$, where $\lambda\colon V \to \text{Labels}$.
- *Distributed automaton:* $A = (Q, \delta_0, \delta, F)$, where $\delta_0\colon \text{Labels} \to Q$, $\delta\colon Q \times 2^Q \to Q$ and $F \subseteq Q$.
- *Timing of G:* $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t\colon V \cup E \to \{0, 1\}$, such that 1 is assigned infinitely often to every node and edge.

Return

Asynchronous run (formal version)

- *Labeled digraph:* $G = (V, E, \lambda)$, where $\lambda\colon V \to \text{Labels}$.
- *Distributed automaton:* $A = (Q, \delta_0, \delta, F)$,
  where $\delta_0\colon \text{Labels} \to Q$, $\delta\colon Q \times 2^Q \to Q$ and $F \subseteq Q$.
- *Timing of G:* $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t\colon V \cup E \to \{0, 1\}$,
  such that 1 is assigned infinitely often to every node and edge.
- *Asynchronous run of A on G timed by τ:* $\rho = (\rho_0, \rho_1, \rho_2, \dots)$,
  where $\rho_t\colon V \cup E \to Q^+$ with $\rho_t(V) \subseteq Q$, such that

Return

Asynchronous run (formal version)

- *Labeled digraph:* $G = (V, E, \lambda)$, where $\lambda\colon V \to \text{Labels}$.
- *Distributed automaton:* $A = (Q, \delta_0, \delta, F)$,
  where $\delta_0\colon \text{Labels} \to Q$, $\delta\colon Q \times 2^Q \to Q$ and $F \subseteq Q$.
- *Timing of G:* $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t\colon V \cup E \to \{0, 1\}$,
  such that 1 is assigned infinitely often to every node and edge.
- *Asynchronous run of A on G timed by* τ: $\rho = (\rho_0, \rho_1, \rho_2, \dots)$,
  where $\rho_t\colon V \cup E \to Q^+$ with $\rho_t(V) \subseteq Q$, such that

  $$\rho_0(v) = \rho_0(vw) = \delta_0(\lambda(v)),$$

Asynchronous run (formal version)

- *Labeled digraph:* $G = (V, E, \lambda)$, where $\lambda\colon V \to \text{Labels}$.
- *Distributed automaton:* $A = (Q, \delta_0, \delta, F)$,
  where $\delta_0\colon \text{Labels} \to Q$, $\delta\colon Q \times 2^Q \to Q$ and $F \subseteq Q$.
- *Timing of G:* $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t\colon V \cup E \to \{0, 1\}$,
  such that 1 is assigned infinitely often to every node and edge.
- *Asynchronous run of A on G timed by τ:* $\rho = (\rho_0, \rho_1, \rho_2, \dots)$,
  where $\rho_t\colon V \cup E \to Q^+$ with $\rho_t(V) \subseteq Q$, such that

$$\rho_0(v) = \rho_0(vw) = \delta_0(\lambda(v)),$$

$$\rho_{t+1}(v) = \begin{cases} \rho_t(v) & \text{if } \tau_{t+1}(v) = 0, \\ \delta\big(\rho_t(v), \{\rho_t(uv).\text{first} \mid uv \in E\}\big) & \text{if } \tau_{t+1}(v) = 1, \end{cases}$$

Asynchronous run (formal version)

- *Labeled digraph:* $G = (V, E, \lambda)$, where $\lambda \colon V \to \text{Labels}$.
- *Distributed automaton:* $A = (Q, \delta_0, \delta, F)$,
  where $\delta_0 \colon \text{Labels} \to Q$, $\delta \colon Q \times 2^Q \to Q$ and $F \subseteq Q$.
- *Timing* of G: $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t \colon V \cup E \to \{0, 1\}$,
  such that 1 is assigned infinitely often to every node and edge.
- *Asynchronous run* of A on G timed by τ: $\rho = (\rho_0, \rho_1, \rho_2, \dots)$,
  where $\rho_t \colon V \cup E \to Q^+$ with $\rho_t(V) \subseteq Q$, such that

$$\rho_0(v) = \rho_0(vw) = \delta_0(\lambda(v)),$$

$$\rho_{t+1}(v) = \begin{cases} \rho_t(v) & \text{if } \tau_{t+1}(v) = 0, \\ \delta\big(\rho_t(v), \{\rho_t(uv).\text{first} \mid uv \in E\}\big) & \text{if } \tau_{t+1}(v) = 1, \end{cases}$$

$$\rho_{t+1}(vw) = \begin{cases} \rho_t(vw).\text{pushlast}(\rho_{t+1}(v)) & \text{if } \tau_{t+1}(vw) = 0, \\ \rho_t(vw).\text{pushlast}(\rho_{t+1}(v)).\text{popfirst} & \text{if } \tau_{t+1}(vw) = 1. \end{cases}$$

Return