

Asynchronous Distributed Automata

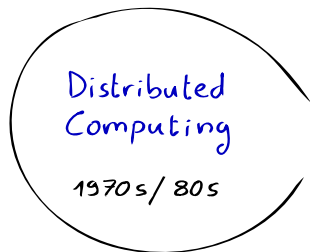
Fabian Reiter

Technical University of Munich

March 27, 2019

Larger context

Larger context



Larger context

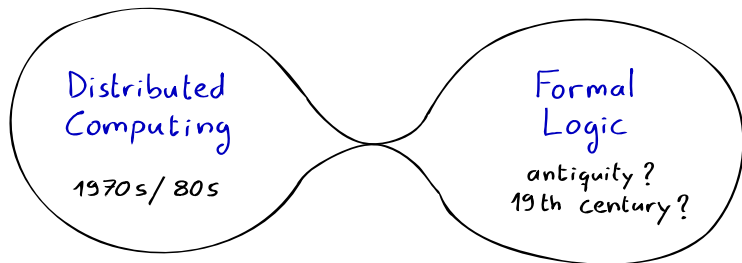
Distributed
Computing

1970s/80s

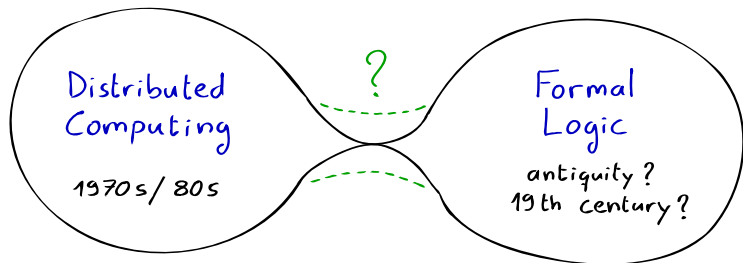
Formal
Logic

antiquity?
19th century?

Larger context

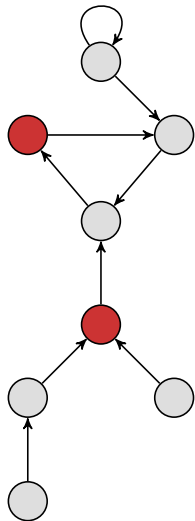


Larger context

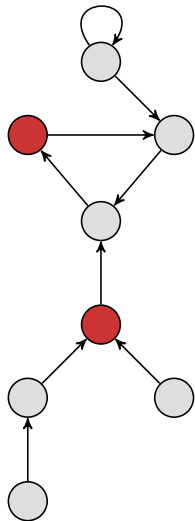


The backward μ -fragment

The backward μ -fragment

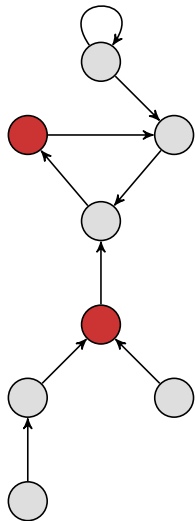


The backward μ -fragment



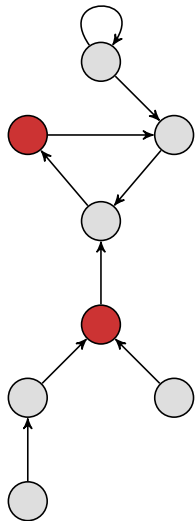
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \left(\quad \right)$$

The backward μ -fragment



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

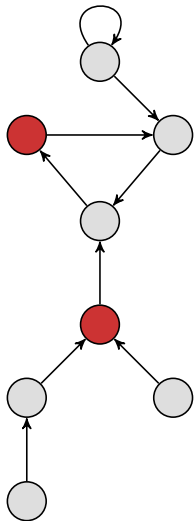
The backward μ -fragment



constant

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

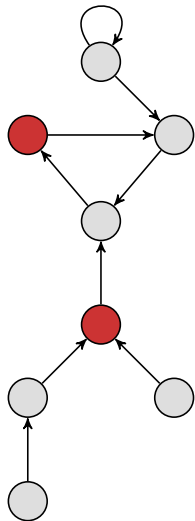
The backward μ -fragment



constant unnegated
 variable

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

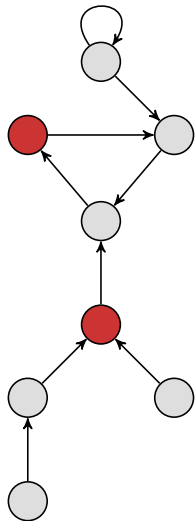
The backward μ -fragment



constant unnegated variable \exists incoming neighbor

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

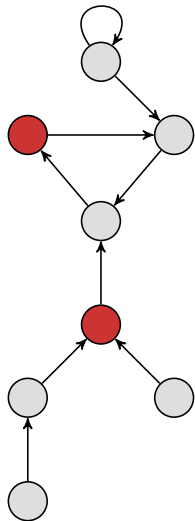
The backward μ -fragment



constant unnegated variable \exists incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{pmatrix}$$

The backward μ -fragment

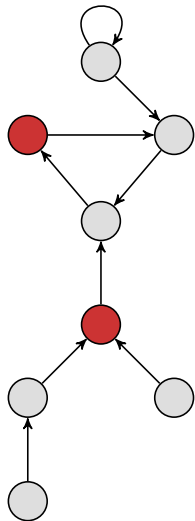


constant unnegated variable \exists incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{pmatrix}$$

\forall incoming neighbors

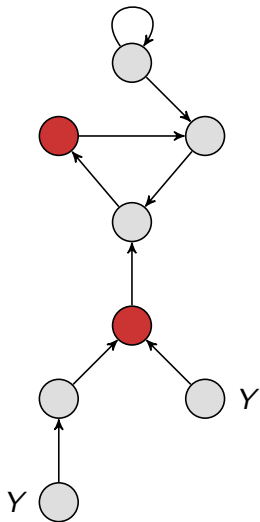
The backward μ -fragment



constant unnegated variable \exists incoming neighbor
 $\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{pmatrix}$
 \forall incoming neighbors

Compute the simultaneous least fixpoint.

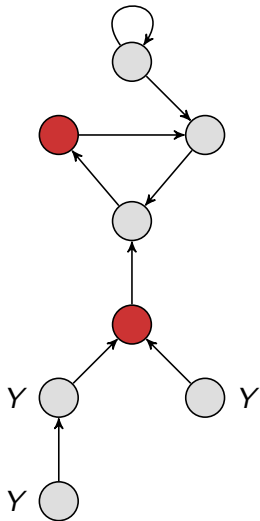
The backward μ -fragment



constant unnegated variable \exists incoming neighbor
 $\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{pmatrix}$
 \forall incoming neighbors

Compute the simultaneous least fixpoint.

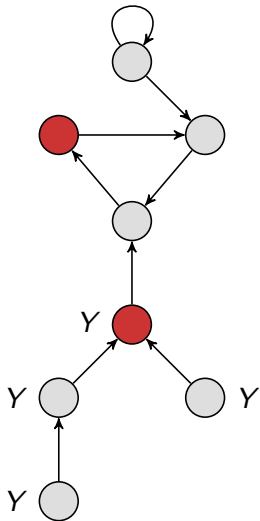
The backward μ -fragment



constant unnegated variable \exists incoming neighbor
 $\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{pmatrix}$
 \forall incoming neighbors

Compute the simultaneous least fixpoint.

The backward μ -fragment



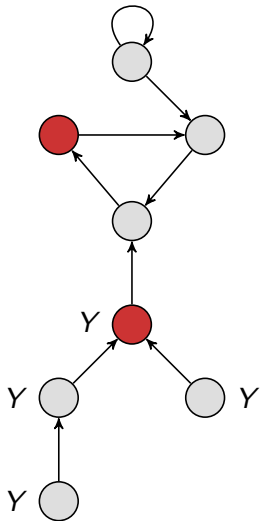
constant unnegated variable \exists incoming neighbor

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{pmatrix}$$

\forall incoming neighbors

Compute the simultaneous least fixpoint.

The backward μ -fragment



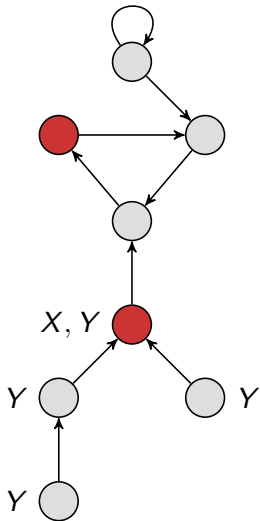
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{pmatrix}$$

constant \rightarrow R
 unnegated variable \rightarrow Y
 \exists incoming neighbor \rightarrow $\bar{\diamond} X$
 \forall incoming neighbors \rightarrow $\bar{\square} Y$

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

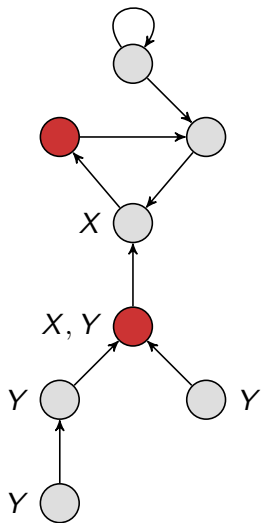
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \left((R \wedge Y) \vee \bar{\diamond} X \right)$$

\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

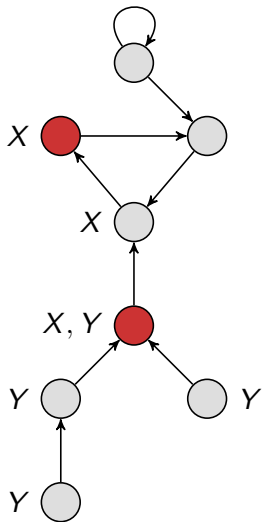
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \left((R \wedge Y) \vee \bar{\diamond} X \right)$$

\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

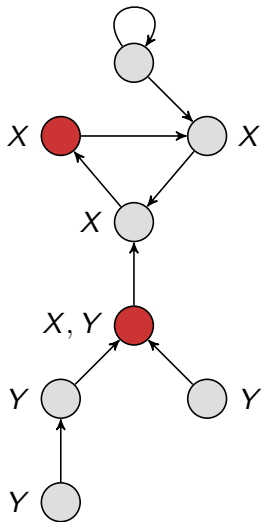
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \left((R \wedge Y) \vee \bar{\diamond} X \right)$$

\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

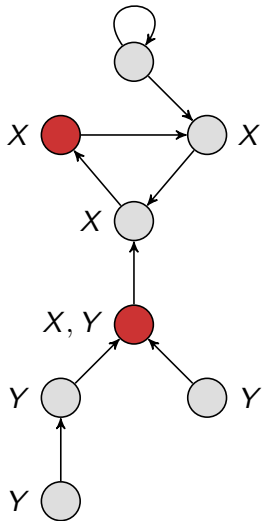
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \left((R \wedge Y) \vee \bar{\diamond} X \right)$$

\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

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

\forall incoming neighbors

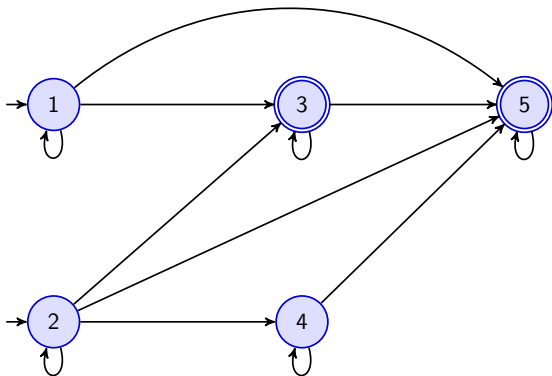
Compute the simultaneous least fixpoint.

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

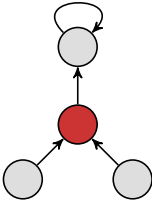
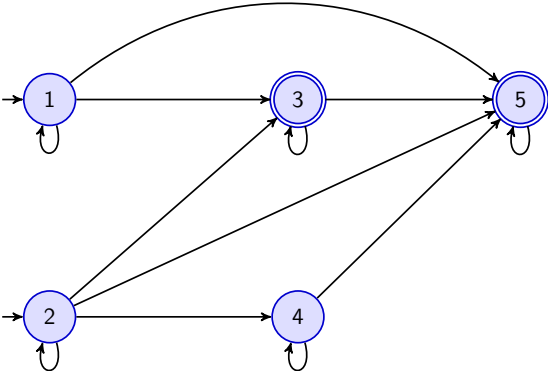
X : “Going backwards, we can reach a red node from which no directed cycle is reachable.”

Distributed automata

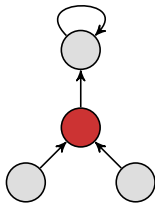
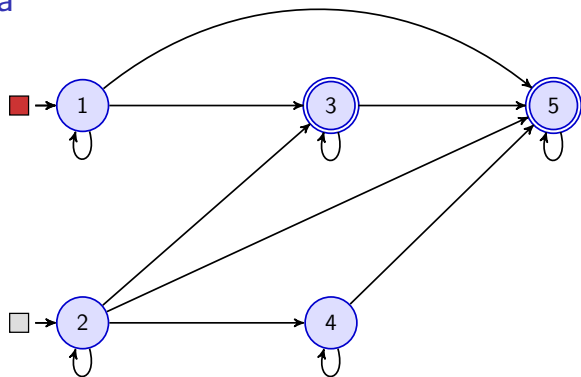
Distributed automata



Distributed automata



Distributed automata

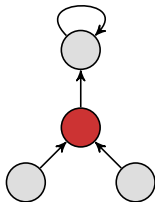
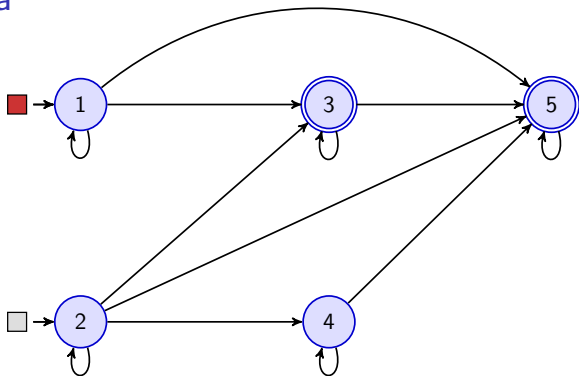


Distributed automata

Transition function:

$$\delta: Q \times 2^Q \rightarrow Q$$

(Q : set of states)

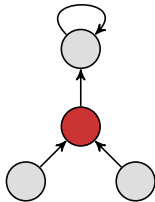
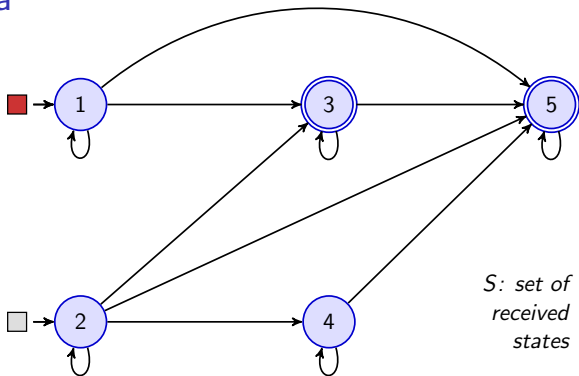


Distributed automata

Transition function:

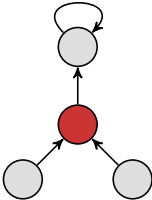
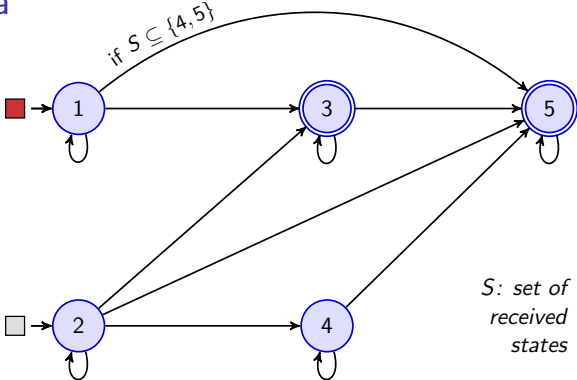
$$\delta: Q \times 2^Q \rightarrow Q$$

(Q : set of states)



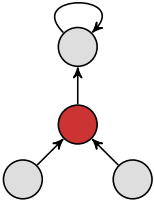
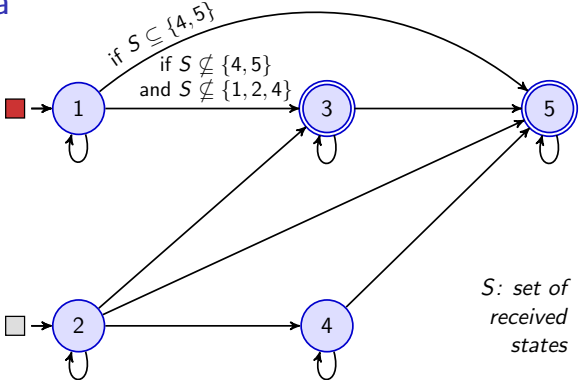
Distributed automata

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



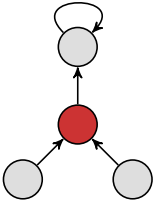
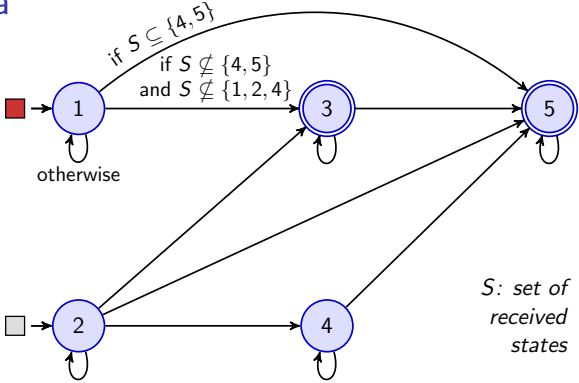
Distributed automata

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



Distributed automata

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

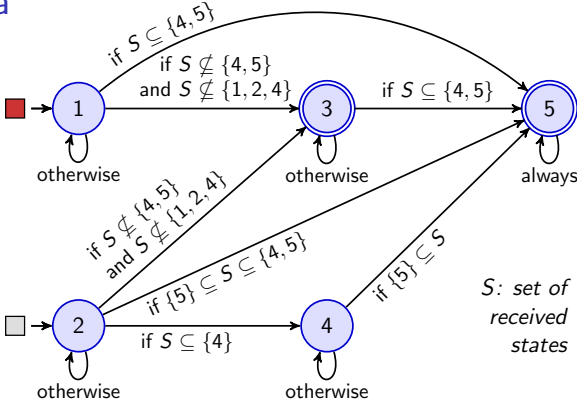


Distributed automata

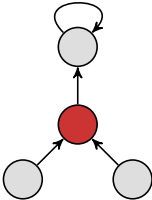
Transition function:

$$\delta: Q \times 2^Q \rightarrow Q$$

(Q : set of states)



S : set of received states

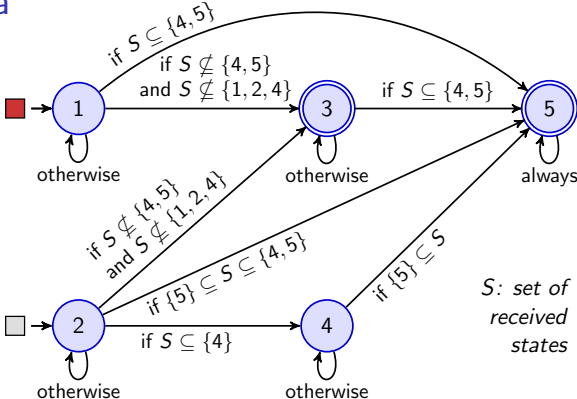


Distributed automata

Transition function:

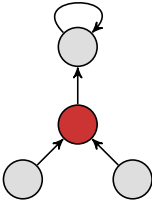
$$\delta: Q \times 2^Q \rightarrow Q$$

(Q : set of states)



S : set of received states

Synchronous run:

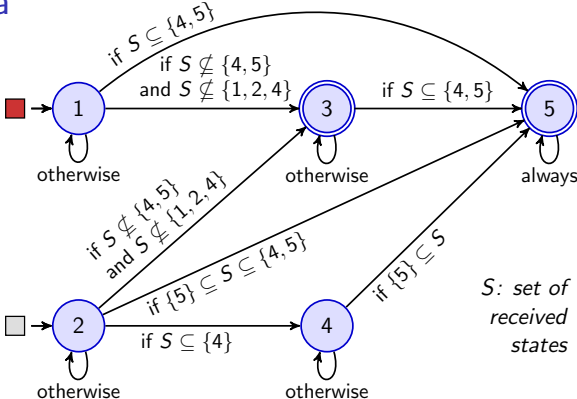


Distributed automata

Transition function:

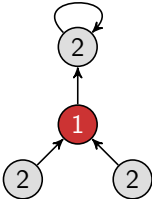
$$\delta: Q \times 2^Q \rightarrow Q$$

(Q : set of states)



S : set of received states

Synchronous run:

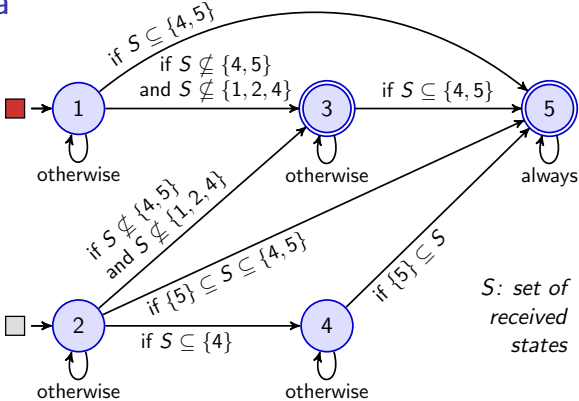


Distributed automata

Transition function:

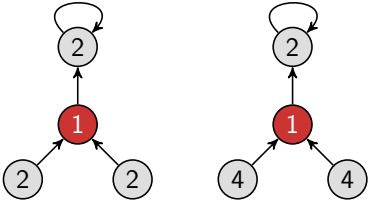
$$\delta: Q \times 2^Q \rightarrow Q$$

(Q : set of states)

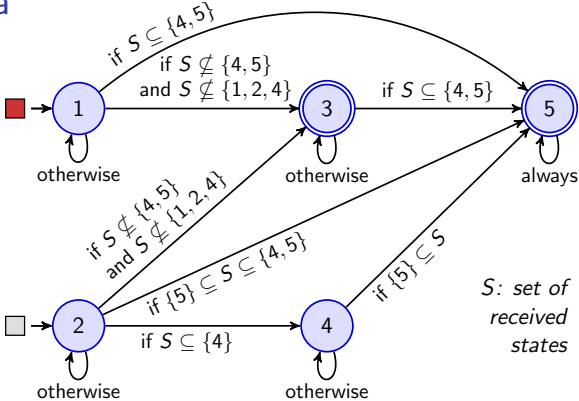


S : set of received states

Synchronous run:



Distributed automata



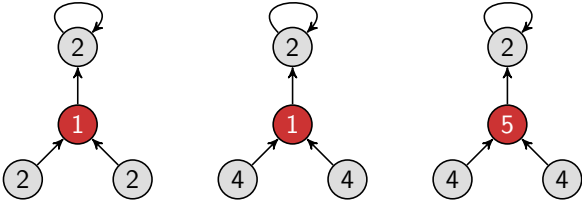
Transition function:

$$\delta: Q \times 2^Q \rightarrow Q$$

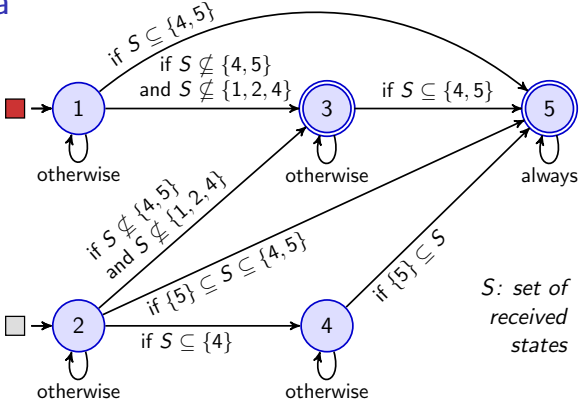
(Q : set of states)

S : set of received states

Synchronous run:



Distributed automata

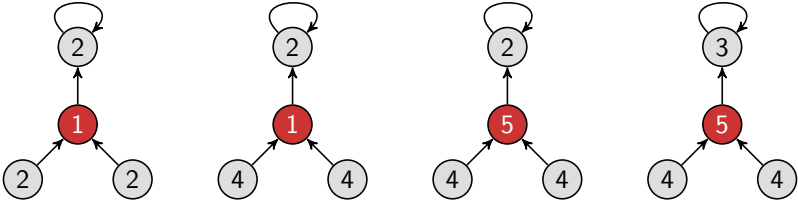


Transition function:

$$\delta: Q \times 2^Q \rightarrow Q$$

(Q : set of states)

Synchronous run:



Powerset construction

Powerset construction

(Antti Kuusisto, 2013)

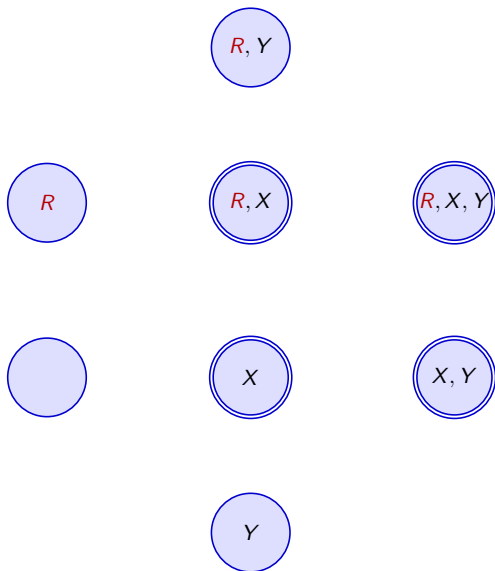
Powerset construction

$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{pmatrix}$$

(Antti Kuusisto, 2013)

Powerset construction

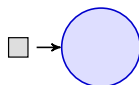
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \left(\begin{pmatrix} R \wedge Y \\ \bar{\square} Y \end{pmatrix} \vee \bar{\diamond} X \right)$$



(Antti Kuusisto, 2013)

Powerset construction

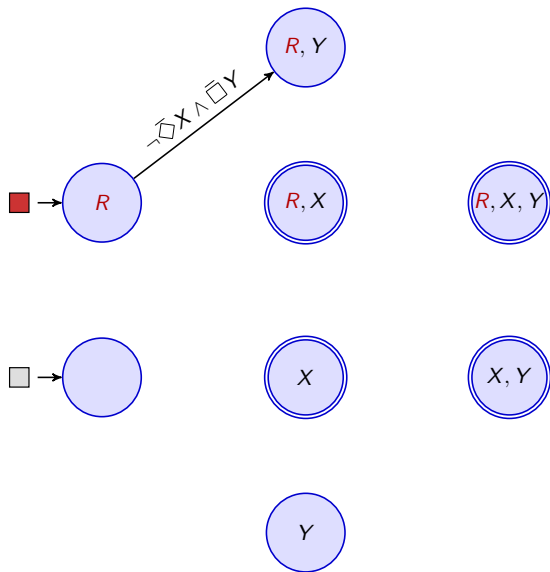
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \left(\begin{array}{l} (R \wedge Y) \vee \bar{\diamond} X \\ \bar{\square} Y \end{array} \right)$$



(Antti Kuusisto, 2013)

Powerset construction

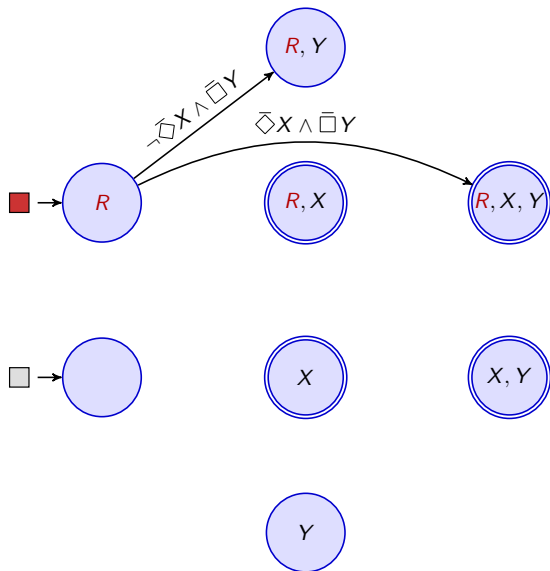
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

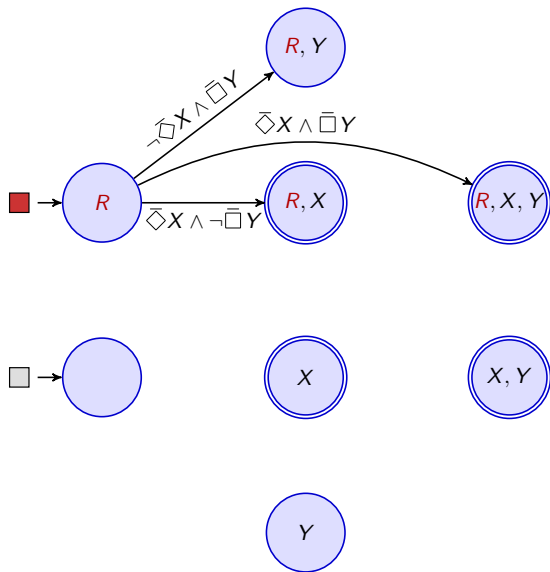
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

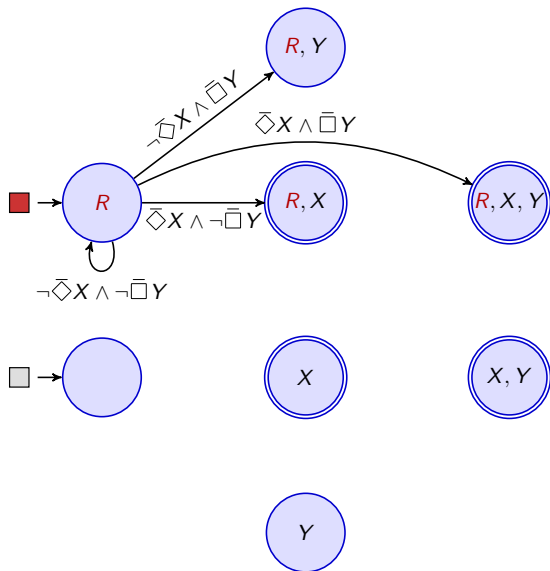
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

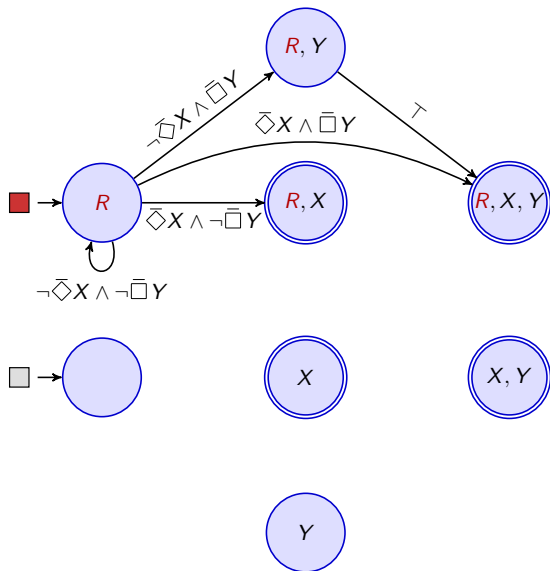
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

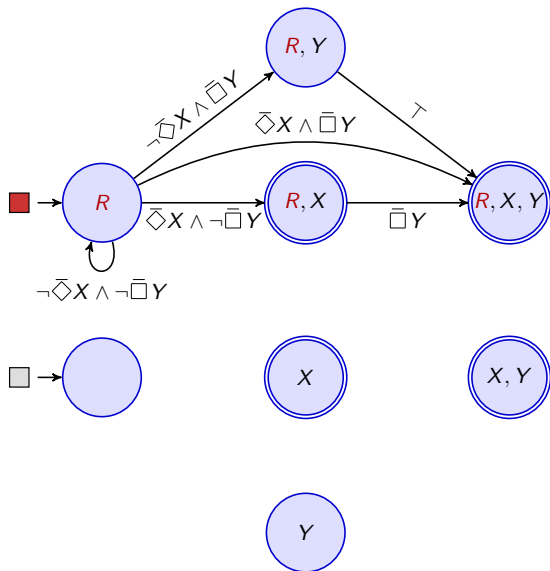
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

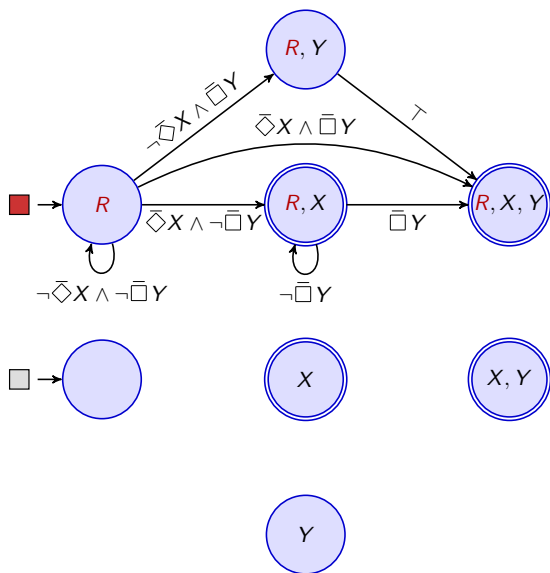
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

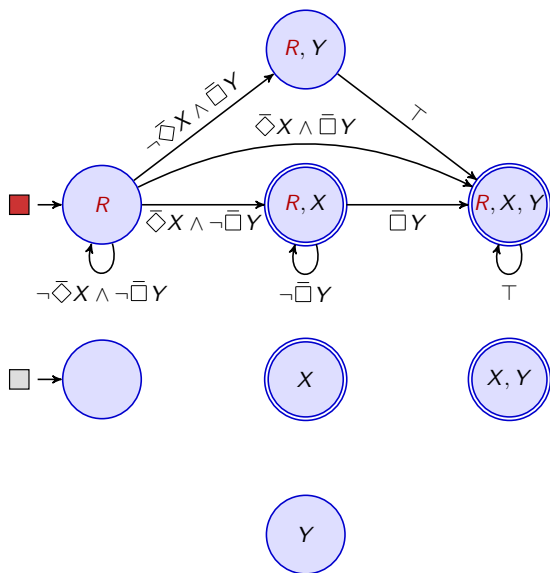
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

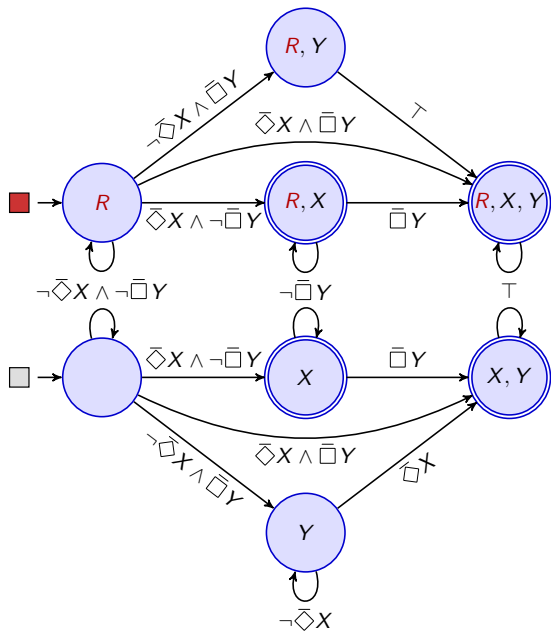
$$\mu \begin{pmatrix} X \\ Y \end{pmatrix} \cdot \begin{pmatrix} (R \wedge Y) \vee \bar{\Diamond} X \\ \bar{\Box} Y \end{pmatrix}$$



(Antti Kuusisto, 2013)

Powerset construction

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

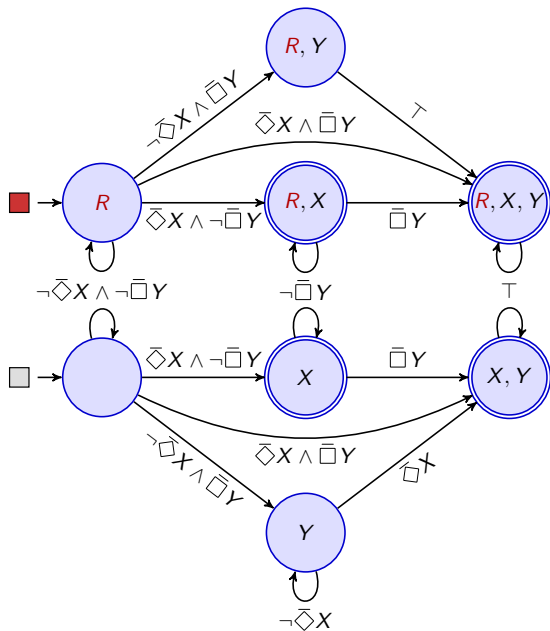


(Antti Kuusisto, 2013)

Powerset construction

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

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



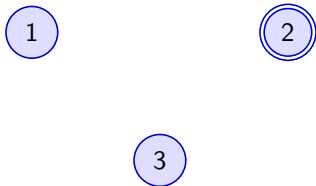
(Antti Kuusisto, 2013)

Synchrony is too powerful

Synchrony is too powerful

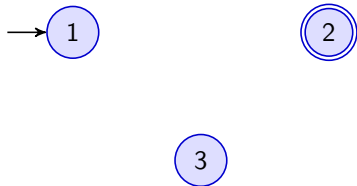
(Antti Kuusisto, 2013)

Synchrony is too powerful



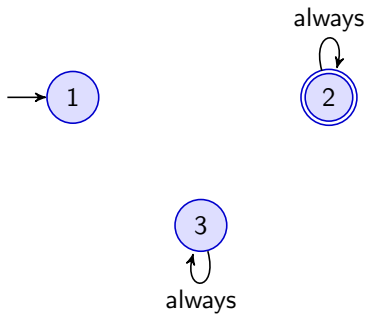
(Antti Kuusisto, 2013)

Synchrony is too powerful



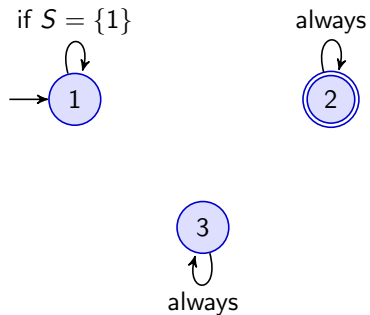
(Antti Kuusisto, 2013)

Synchrony is too powerful



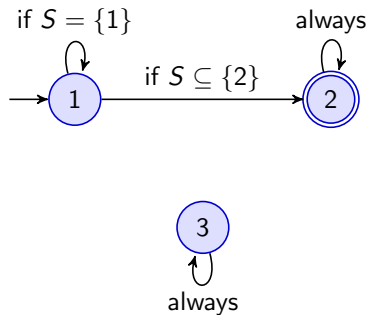
(Antti Kuusisto, 2013)

Synchrony is too powerful



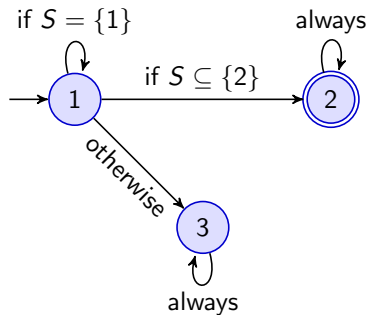
(Antti Kuusisto, 2013)

Synchrony is too powerful



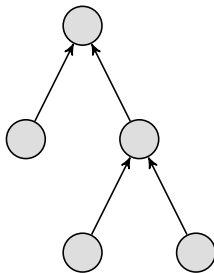
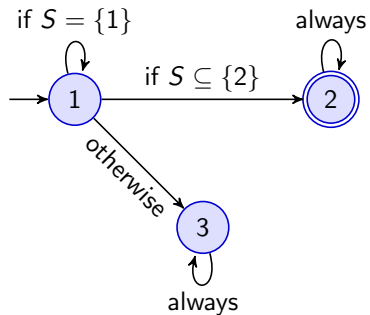
(Antti Kuusisto, 2013)

Synchrony is too powerful



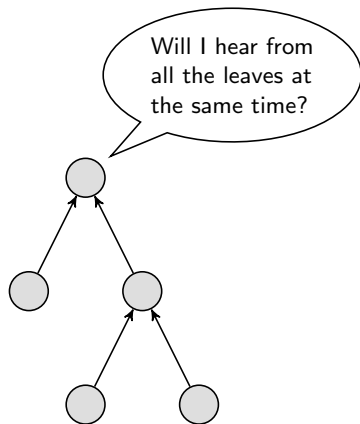
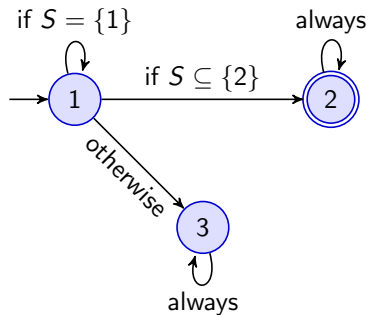
(Antti Kuusisto, 2013)

Synchrony is too powerful



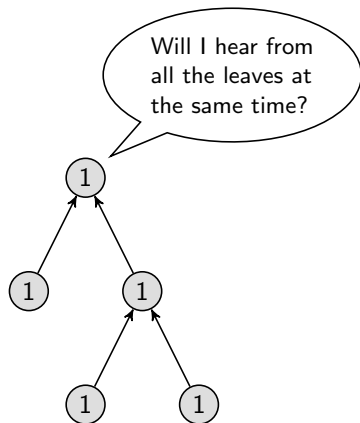
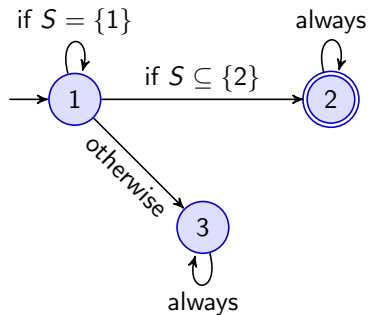
(Antti Kuusisto, 2013)

Synchrony is too powerful



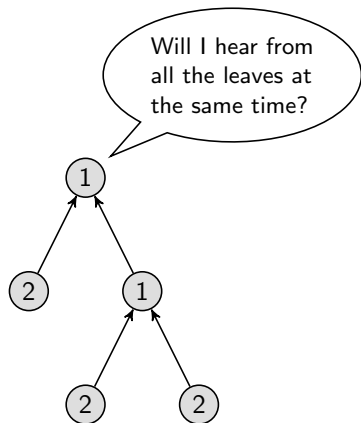
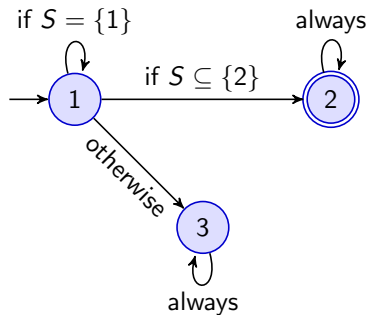
(Antti Kuusisto, 2013)

Synchrony is too powerful



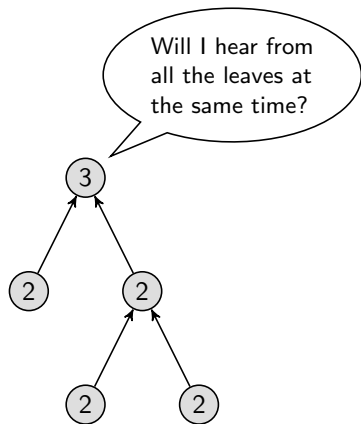
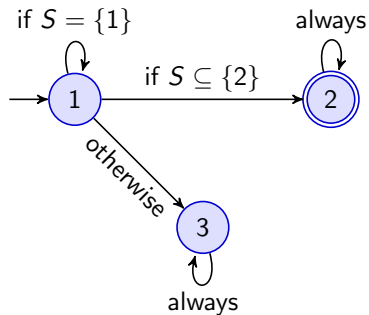
(Antti Kuusisto, 2013)

Synchrony is too powerful



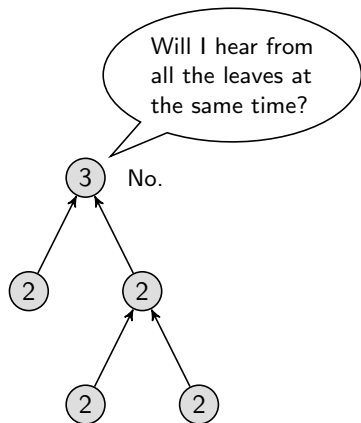
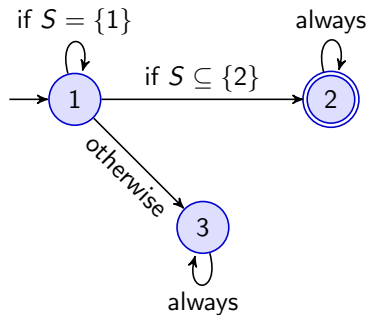
(Antti Kuusisto, 2013)

Synchrony is too powerful



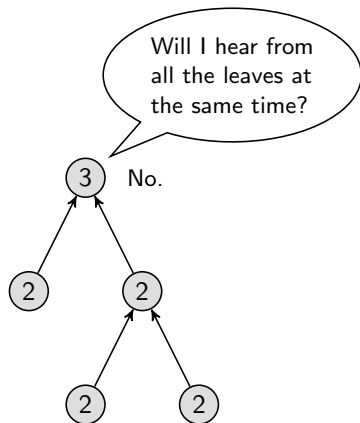
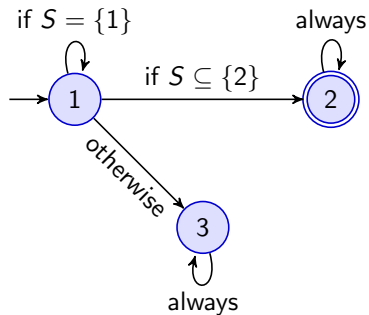
(Antti Kuusisto, 2013)

Synchrony is too powerful



(Antti Kuusisto, 2013)

Synchrony is too powerful

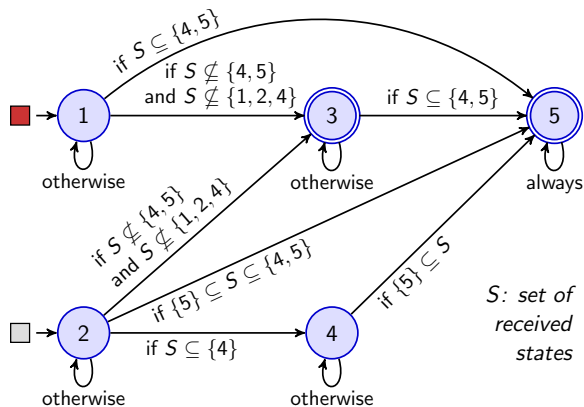


Not even expressible in **monadic second-order logic** (MSO)!

(Antti Kuusisto, 2013)

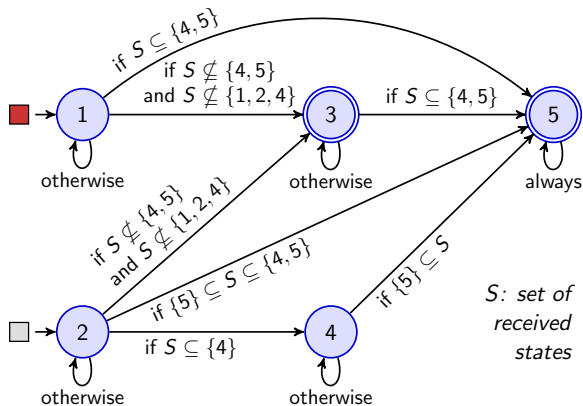
Asynchronous run

Asynchronous run



Asynchronous run

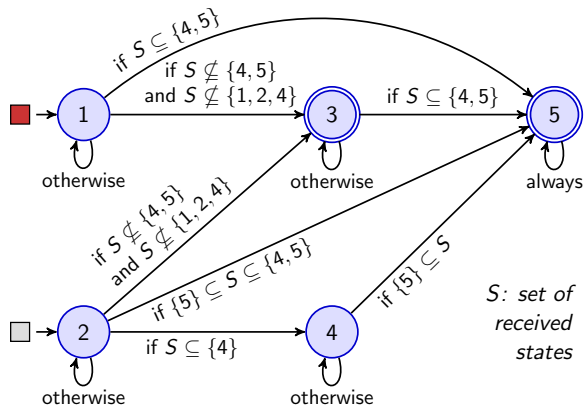
Nodes may sleep,
miss messages.



Asynchronous run

Nodes may sleep,
miss messages.

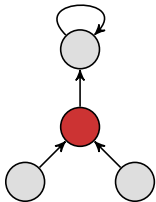
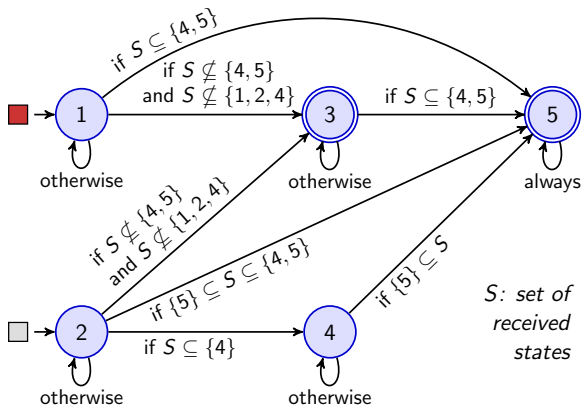
Messages may be
delayed (FIFO).



Asynchronous run

Nodes may sleep,
miss messages.

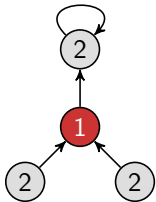
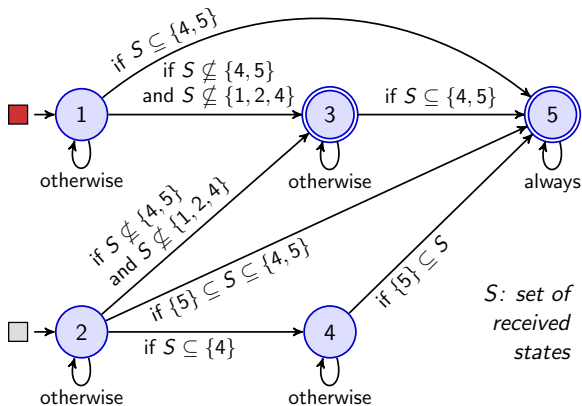
Messages may be
delayed (FIFO).



Asynchronous run

Nodes may sleep,
miss messages.

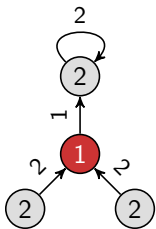
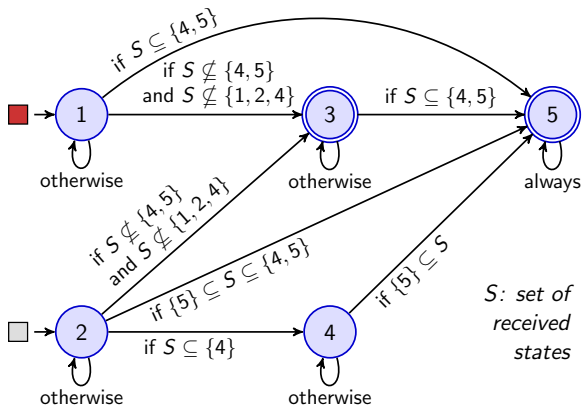
Messages may be
delayed (FIFO).



Asynchronous run

Nodes may sleep,
miss messages.

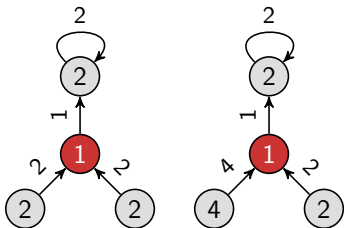
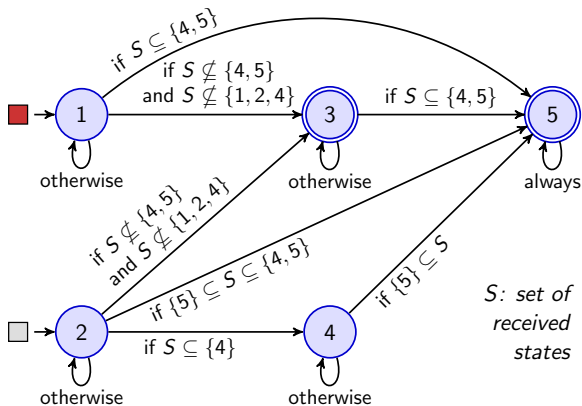
Messages may be
delayed (FIFO).



Asynchronous run

Nodes may sleep,
miss messages.

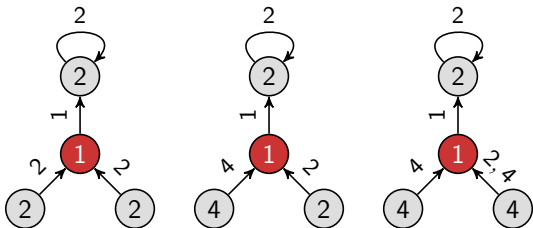
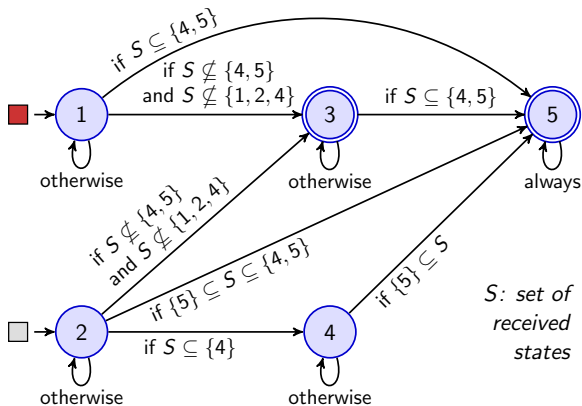
Messages may be delayed (FIFO).



Asynchronous run

Nodes may sleep,
miss messages.

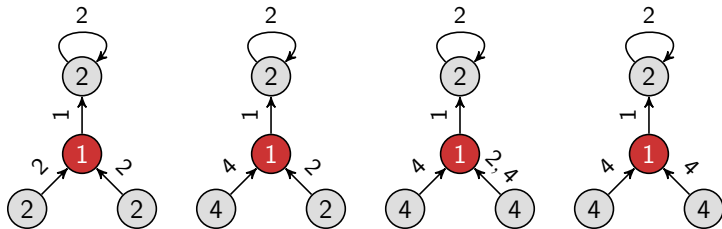
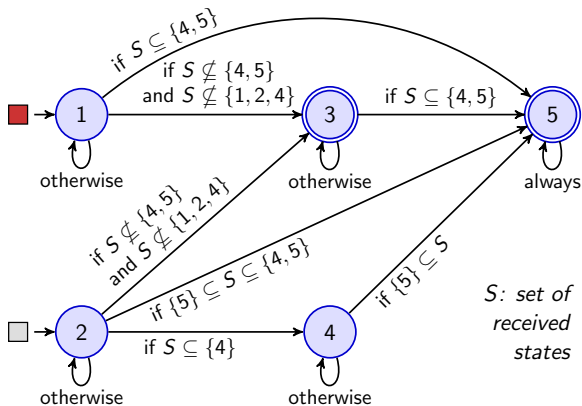
Messages may be
delayed (FIFO).



Asynchronous run

Nodes may sleep, miss messages.

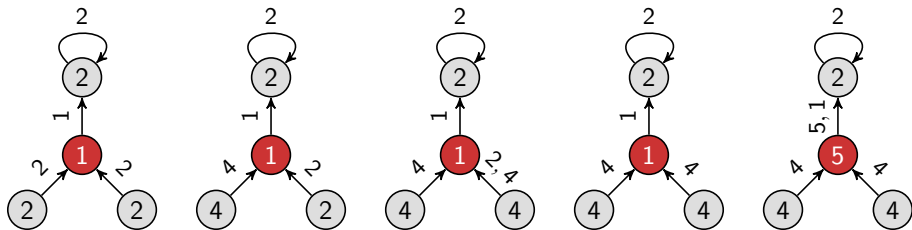
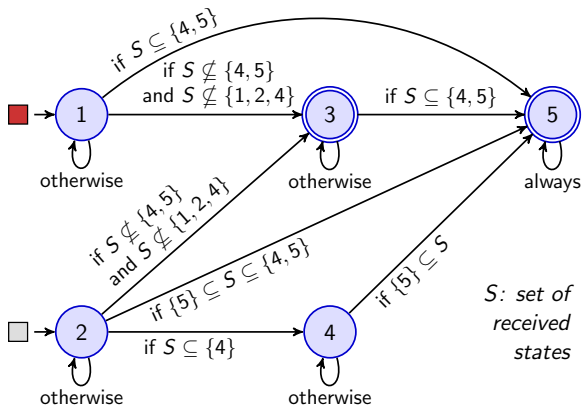
Messages may be delayed (FIFO).



Asynchronous run

Nodes may sleep,
miss messages.

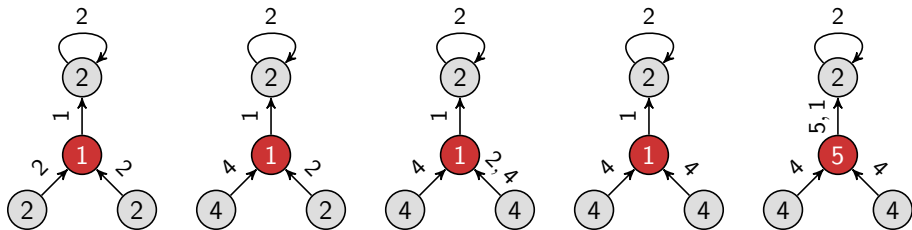
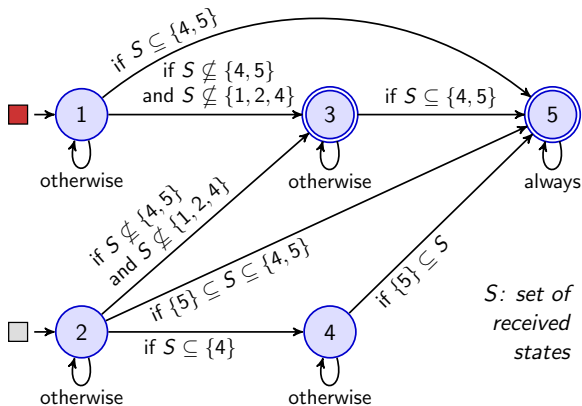
Messages may be
delayed (FIFO).



Asynchronous run

Nodes may sleep,
miss messages.

Messages may be delayed (FIFO).



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).

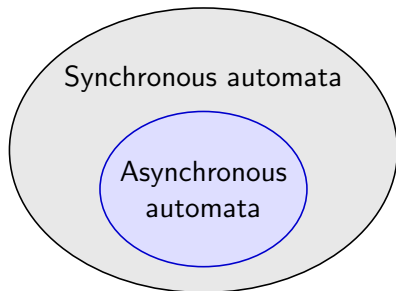
Synchronous automata



Asynchronous automata

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

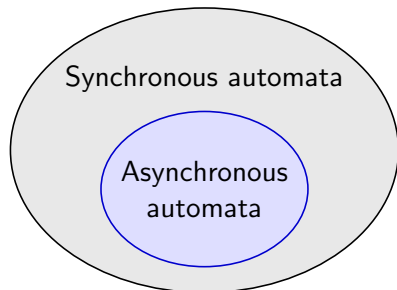
An automaton is **asynchronous** if its acceptance behavior is independent of the adversary (on all finite digraphs).



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?

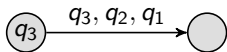
A little bonus

A little bonus

Lossless asynchrony
(weaker adversary):

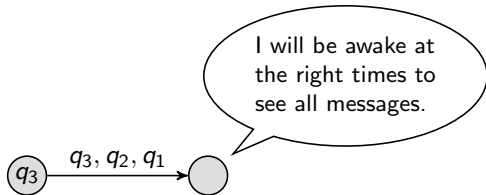
A little bonus

Lossless asynchrony
(weaker adversary):



A little bonus

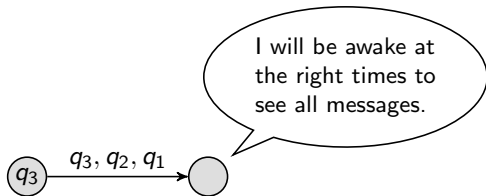
Lossless asynchrony
(weaker adversary):



A little bonus

Lossless asynchrony
(weaker adversary):

synchronous



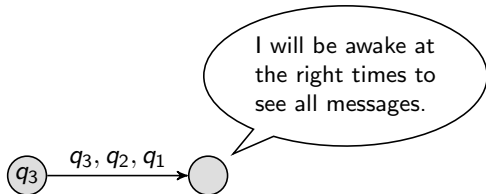
A little bonus

Lossless asynchrony
(weaker adversary):

synchronous

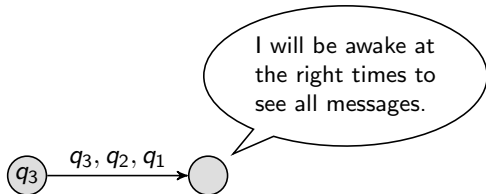
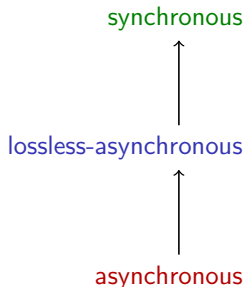


lossless-asynchronous



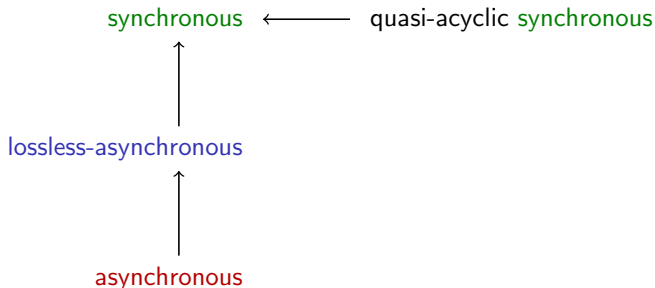
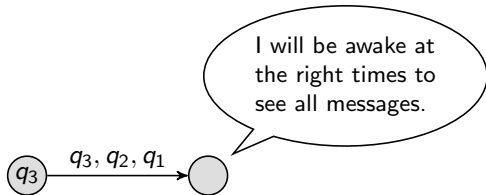
A little bonus

Lossless asynchrony
(weaker adversary):



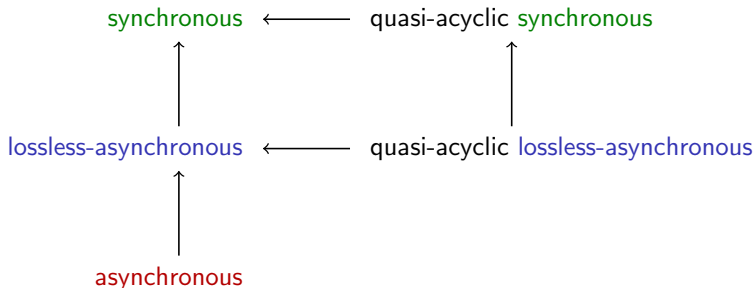
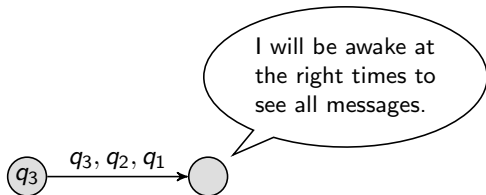
A little bonus

Lossless asynchrony
(weaker adversary):



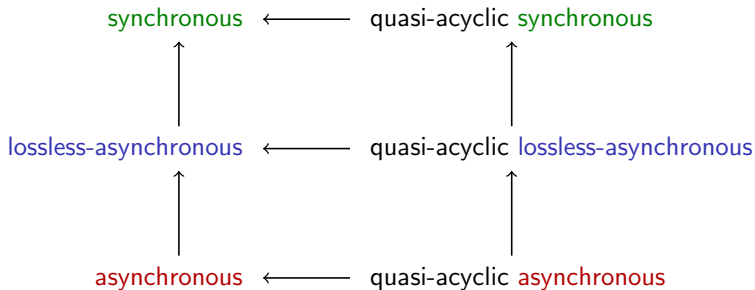
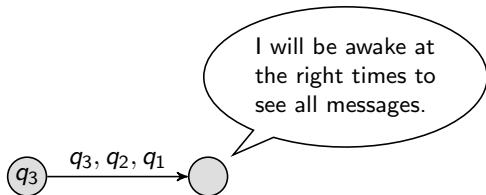
A little bonus

Lossless asynchrony
(weaker adversary):



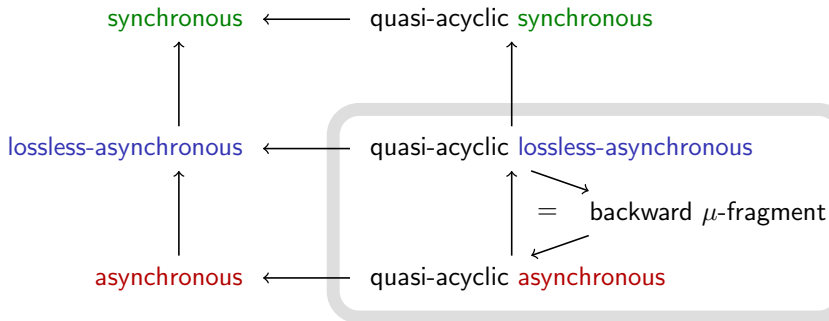
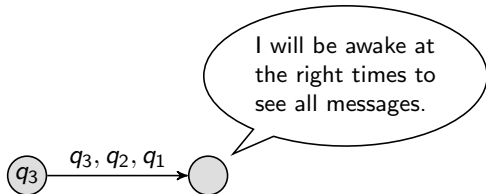
A little bonus

Lossless asynchrony
(weaker adversary):



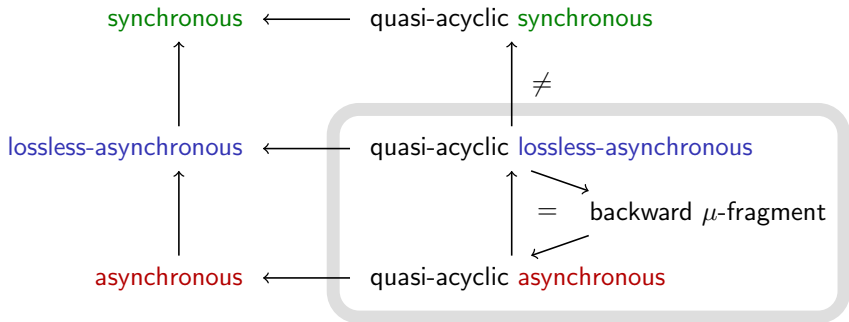
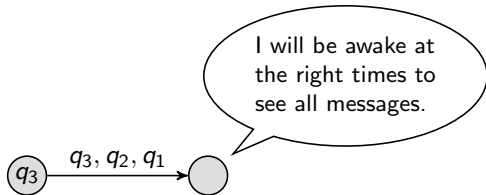
A little bonus

Lossless asynchrony
(weaker adversary):



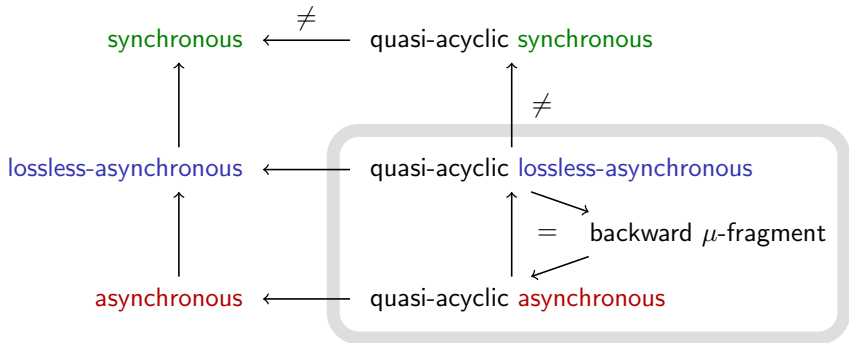
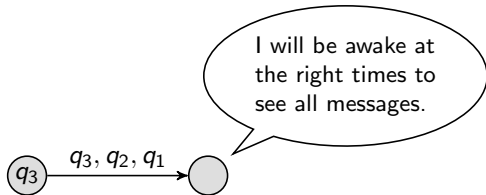
A little bonus

Lossless asynchrony
(weaker adversary):



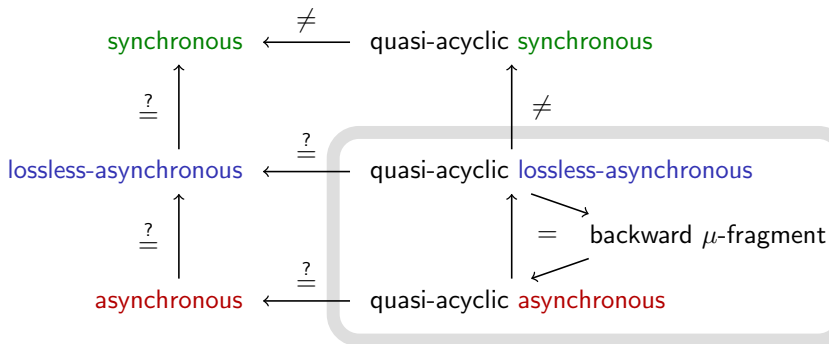
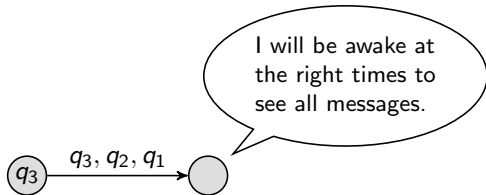
A little bonus

Lossless asynchrony
(weaker adversary):



A little bonus

Lossless asynchrony
(weaker adversary):



Thank you!

Asynchronous run (formal version)

Asynchronous run (formal version)

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

Asynchronous run (formal version)

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

Asynchronous run (formal version)

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

Asynchronous run (formal version)

- ▶ *Labeled digraph*: $G = (V, E, \lambda)$, where $\lambda: V \rightarrow \text{Labels}$.
- ▶ *Distributed automaton*: $A = (Q, \delta_0, \delta, F)$,
where $\delta_0: \text{Labels} \rightarrow Q$, $\delta: Q \times 2^Q \rightarrow Q$ and $F \subseteq Q$.
- ▶ *Timing of G*: $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t: V \cup E \rightarrow \{0, 1\}$,
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 $\rho_t: V \cup E \rightarrow Q^+$ with $\rho_t(V) \subseteq Q$, such that

Asynchronous run (formal version)

- ▶ *Labeled digraph*: $G = (V, E, \lambda)$, where $\lambda: V \rightarrow \text{Labels}$.
- ▶ *Distributed automaton*: $A = (Q, \delta_0, \delta, F)$,
where $\delta_0: \text{Labels} \rightarrow Q$, $\delta: Q \times 2^Q \rightarrow Q$ and $F \subseteq Q$.
- ▶ *Timing of G*: $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t: V \cup E \rightarrow \{0, 1\}$,
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 $\rho_t: V \cup E \rightarrow Q^+$ 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 $\lambda: V \rightarrow \text{Labels}$.
- ▶ *Distributed automaton*: $A = (Q, \delta_0, \delta, F)$,
where $\delta_0: \text{Labels} \rightarrow Q$, $\delta: Q \times 2^Q \rightarrow Q$ and $F \subseteq Q$.
- ▶ *Timing of G*: $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t: V \cup E \rightarrow \{0, 1\}$,
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 $\rho_t: V \cup E \rightarrow Q^+$ with $\rho_t(V) \subseteq Q$, such that

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

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

Asynchronous run (formal version)

- ▶ *Labeled digraph*: $G = (V, E, \lambda)$, where $\lambda: V \rightarrow \text{Labels}$.
- ▶ *Distributed automaton*: $A = (Q, \delta_0, \delta, F)$,
where $\delta_0: \text{Labels} \rightarrow Q$, $\delta: Q \times 2^Q \rightarrow Q$ and $F \subseteq Q$.
- ▶ *Timing of G*: $\tau = (\tau_1, \tau_2, \tau_3, \dots)$, where $\tau_t: V \cup E \rightarrow \{0, 1\}$,
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 $\rho_t: V \cup E \rightarrow Q^+$ with $\rho_t(V) \subseteq Q$, such that

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

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

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