Cascade Decomposition of Asynchronous Zielonka Automata

Paul Gastin
LMF, ENS Paris-Saclay, IRL ReLaX

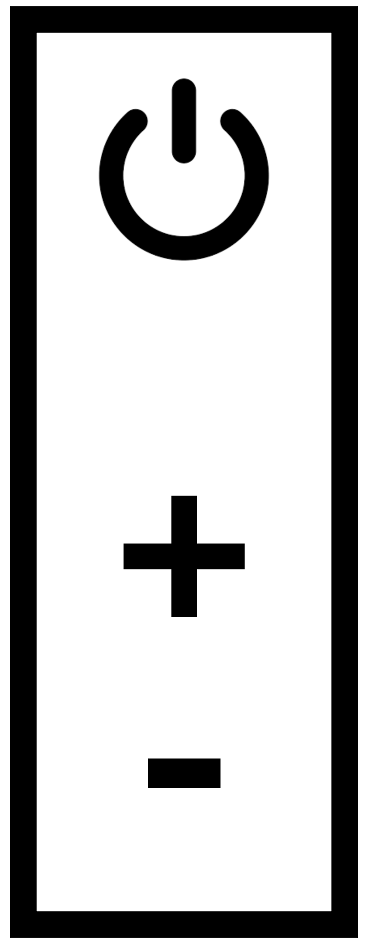
Joint Work with Bharat Adsul (IIT Bombay), Saptarshi Sarkar (IIT Bombay) and Pascal Weil (LaBRI, Univ. Bordeaux)

Based on CONCUR'20, LMCS'22, CONCUR'22 and work in progress

Outline

- Labelling functions, sequential transducers and cascade product
- Krohn-Rhodes theorem for aperiodic/regular word languages
- Model of concurrency: Mazurkiewicz traces and asynchronous Zielonka automata
- Asynchronous labelling functions, transducers and cascade product
- Propositional dynamic logic for traces
- Conclusion

Labelling function $\theta : \Sigma^* \rightarrow \Gamma^*$

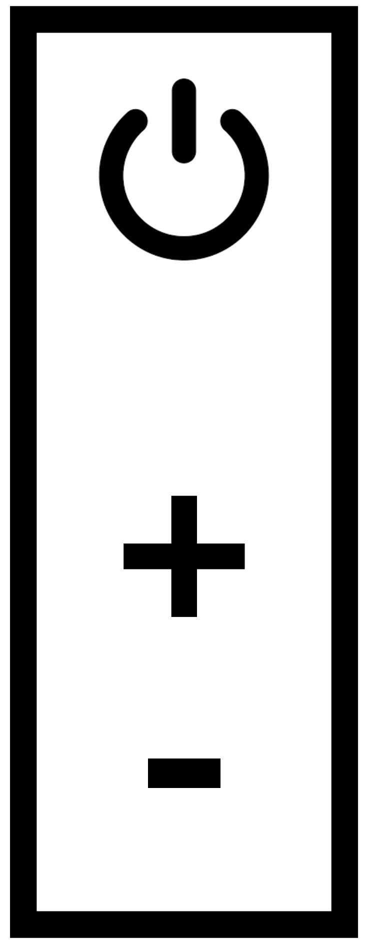


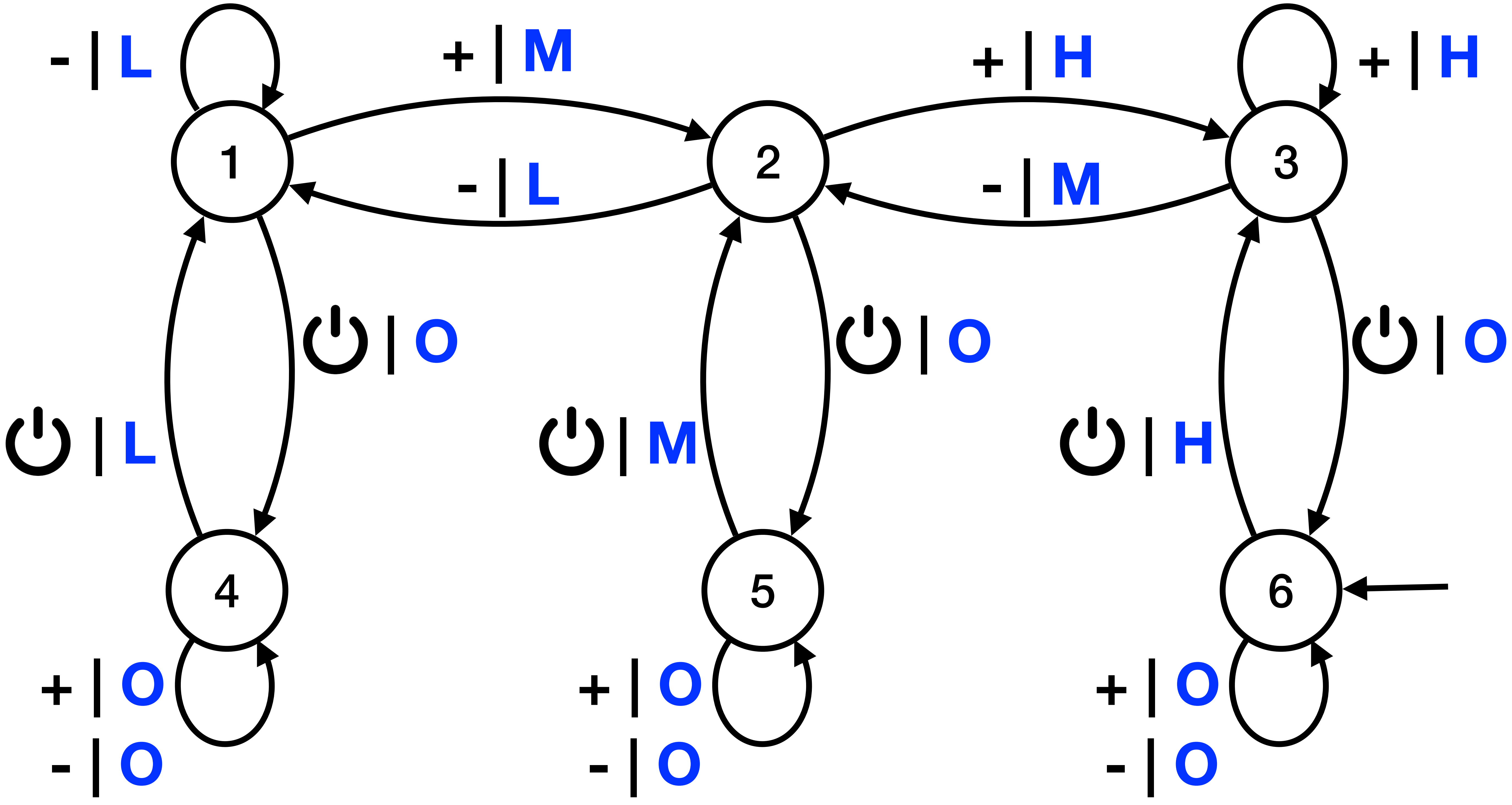
-	⏻	-	⏻	+	⏻	-	-	⏻	⏻	+	+	+	⏻
O	H	M	O	O	M	L	L	O	L	M	H	H	O

$\Sigma = \{$ ⏻ $, -, + \}$ and $\Gamma = \{\mathrm{O}, \mathrm{L}, \mathrm{M}, \mathrm{H}\}$

- ⏻ - ⏻ + ⏻ - - ⏻ ⏻ + + + ⏻

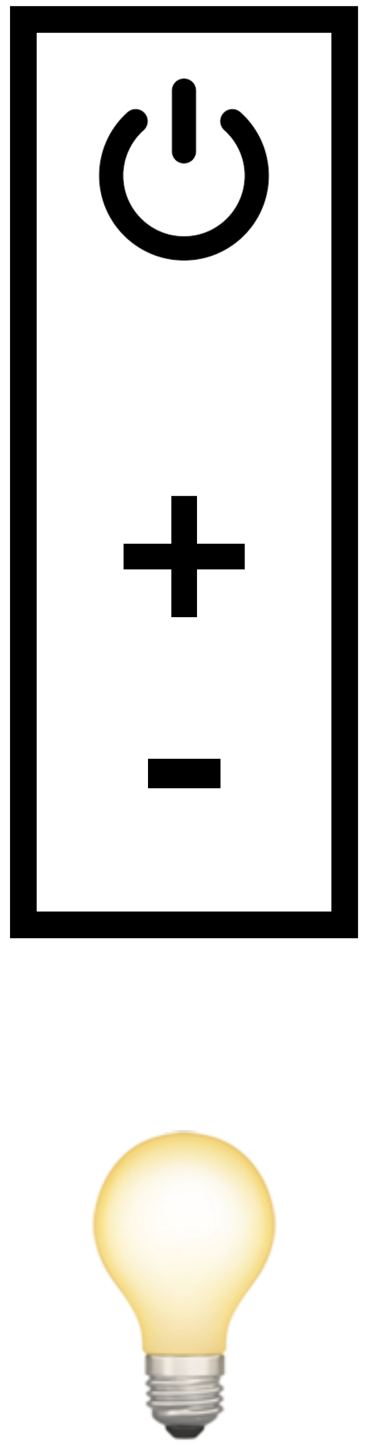
O H M O O M L L O L M H H O

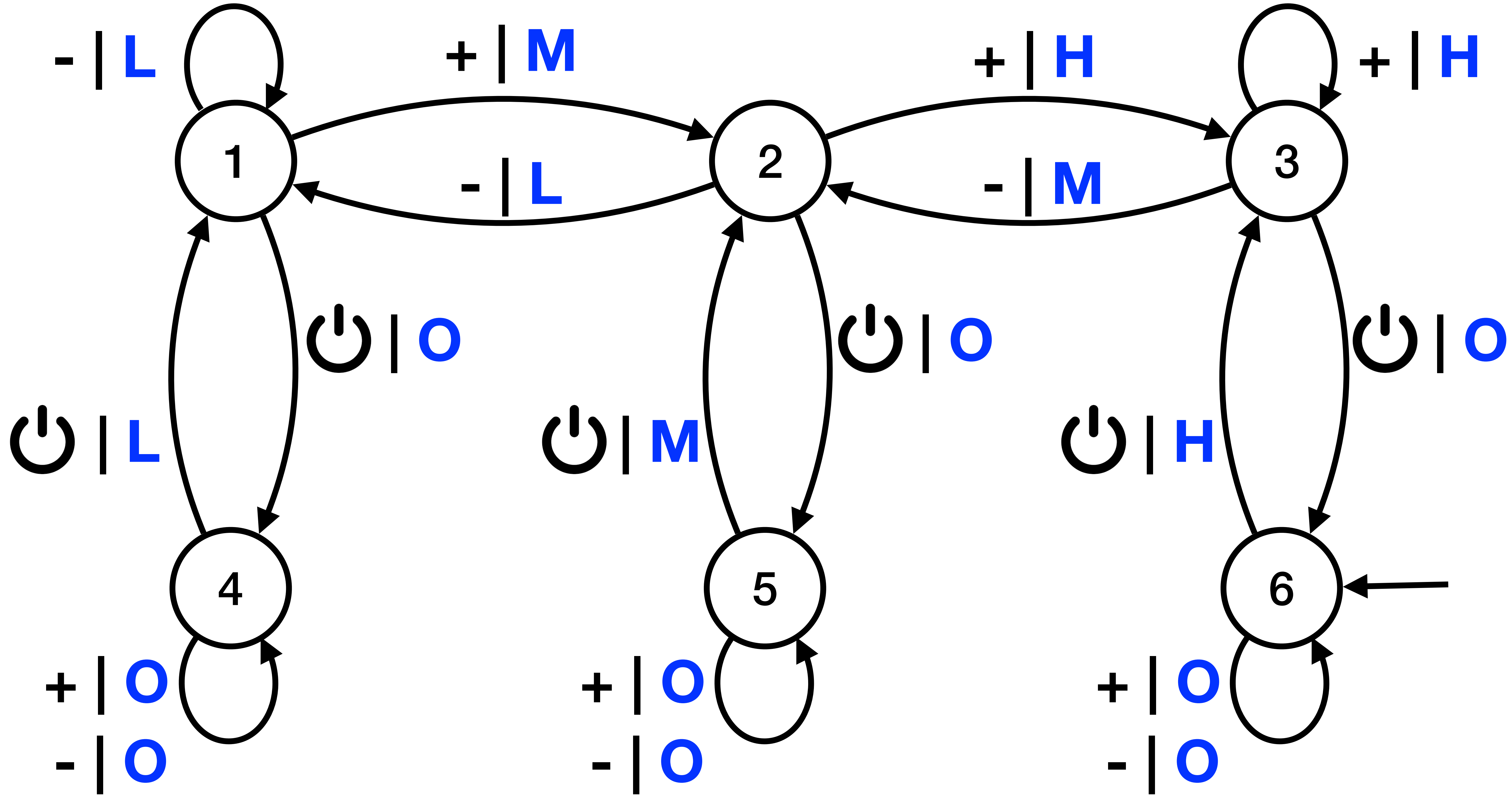




- ⏻ - ⏻ + ⏻ - - ⏻ ⏻ + + + ⏻

O H M O O M L L O L M H H O





letter-to-letter sequential transducer $\mathscr{T} = (Q, q_0, \Sigma, \delta, \Gamma, \mu)$

Composition - Cascade product

Composition - Cascade product

- Composition of labelling functions

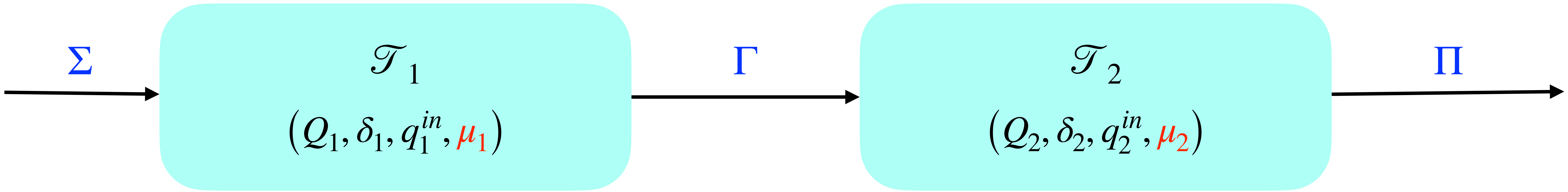
$$\Sigma^* \xrightarrow{\theta_1} \Gamma^* \xrightarrow{\theta_2} \Pi^*$$

Composition - Cascade product

- Composition of labelling functions

$$\Sigma^* \xrightarrow{\theta_1} \Gamma^* \xrightarrow{\theta_2} \Pi^*$$

- Cascade product of (letter-to-letter) sequential transducers



Composition - Cascade product

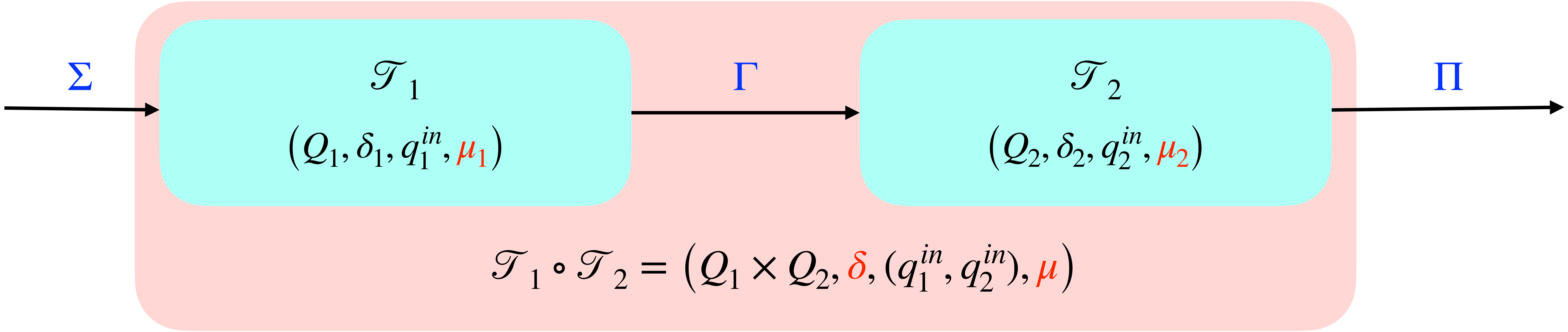
- Composition of labelling functions

$$\Sigma^* \xrightarrow{\theta_1} \Gamma^* \xrightarrow{\theta_2} \Pi^*$$

$$\delta((p,q),a) = \big(\delta_1(p,a), \delta_2(q, \mu_1(p,a))\big)$$

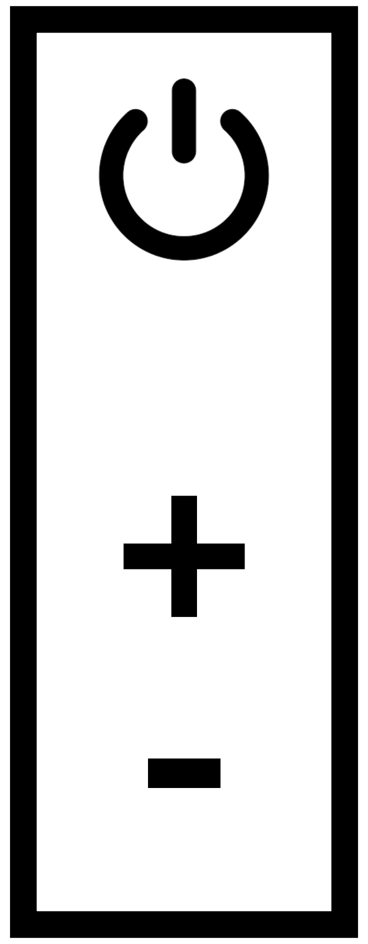
$$\mu((p,q),a) = \mu_2(q, \mu_1(p,a))$$

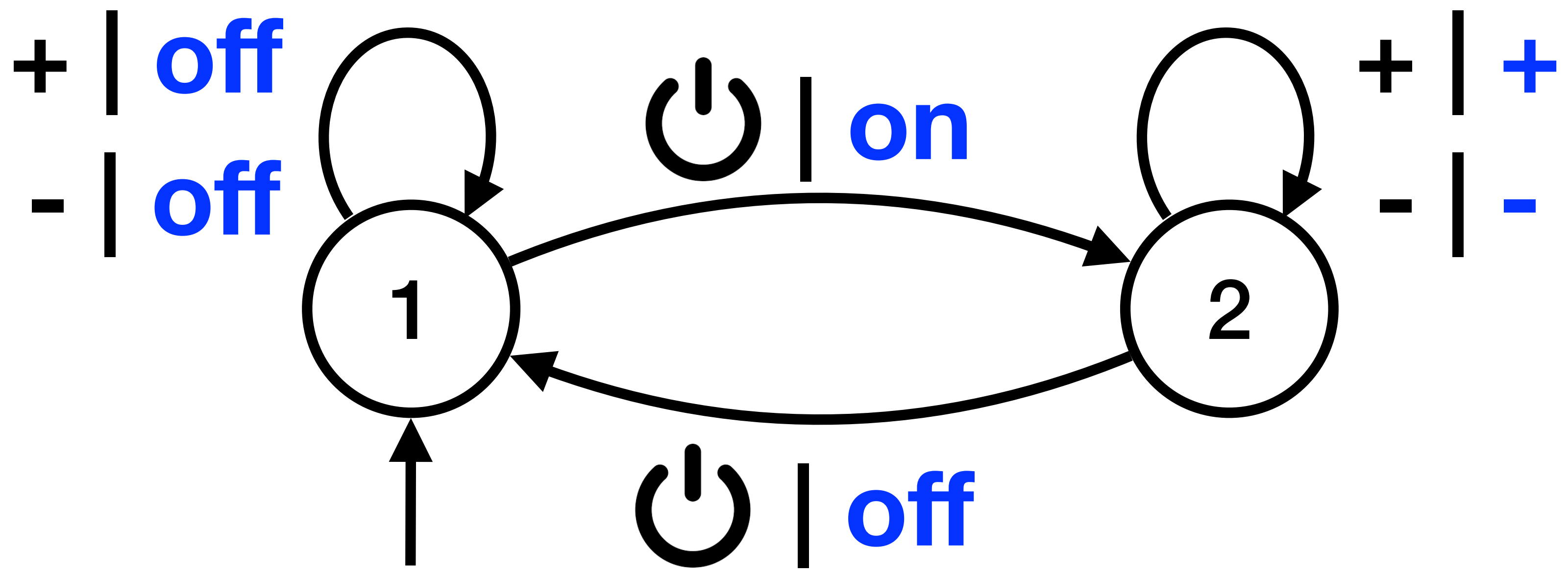
- Cascade product of (letter-to-letter) sequential transducers



- ⏻ - ⏻ + ⏻ - - ⏻ ⏻ + + + ⏻

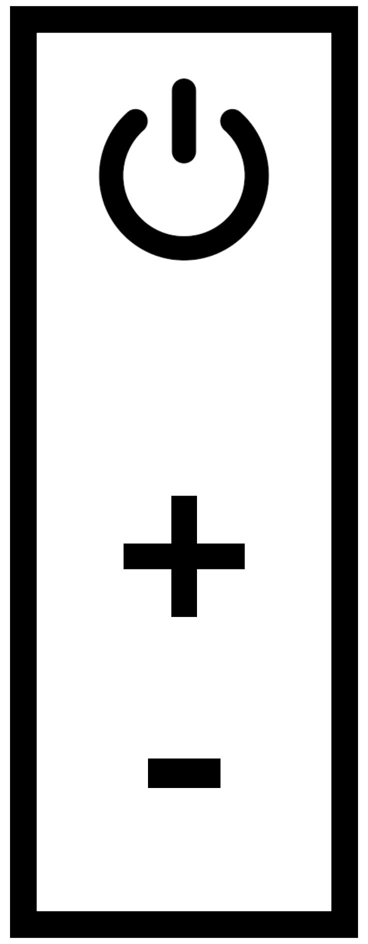
O H M O O M L L O L M H H O

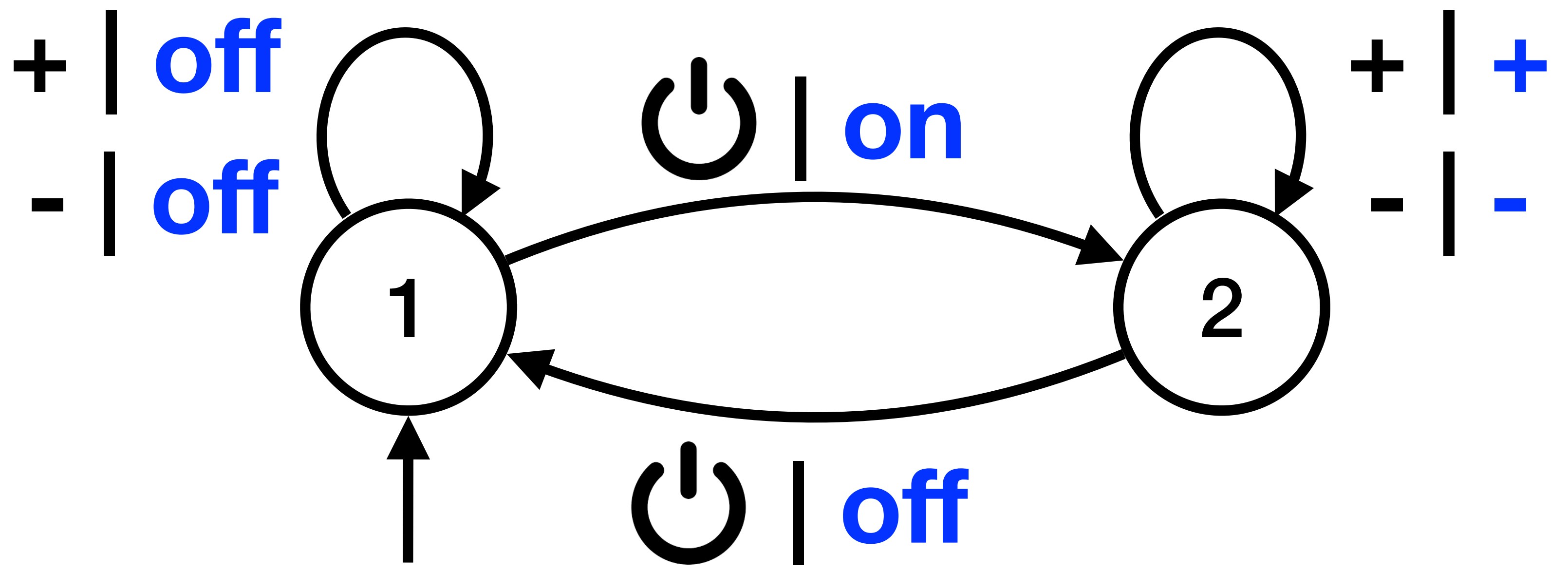


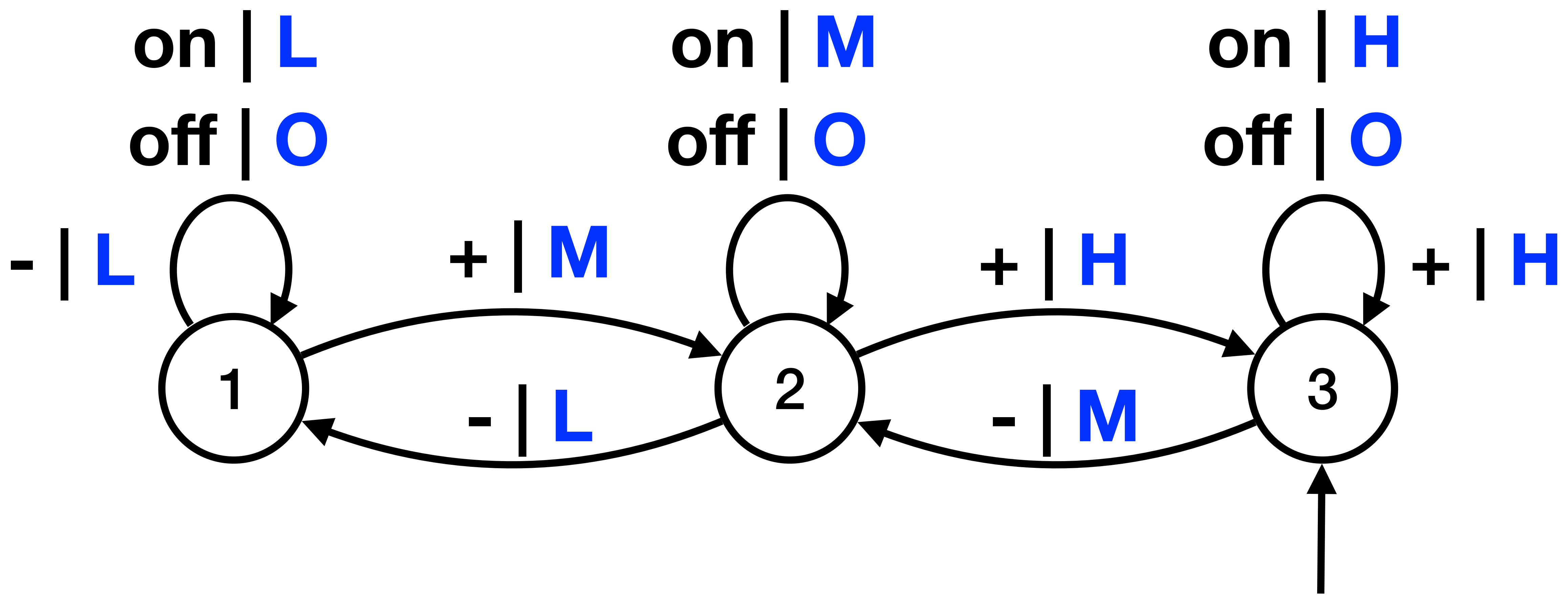


- ⏻ - ⏻ + ⏻ - - ⏻ ⏻ + + + ⏻

O H M O O M L L O L M H H O







Outline

- ✓ Labelling functions, sequential transducers and cascade product
- Krohn-Rhodes theorem for aperiodic/regular word languages
- Model of concurrency: Mazurkiewicz traces and asynchronous Zielonka automata
- Asynchronous labelling functions, transducers and cascade product
- Propositional dynamic logic for traces
- Conclusion

Krohn-Rhodes

Theorem

Any (letter-to-letter) sequential transducer $\mathcal{T}$ can be realised by a cascade product of reset or permutation transducers:

$$\mathcal{T} \equiv \mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Reset or Permutation is a property of the underlying input automaton:

Krohn-Rhodes

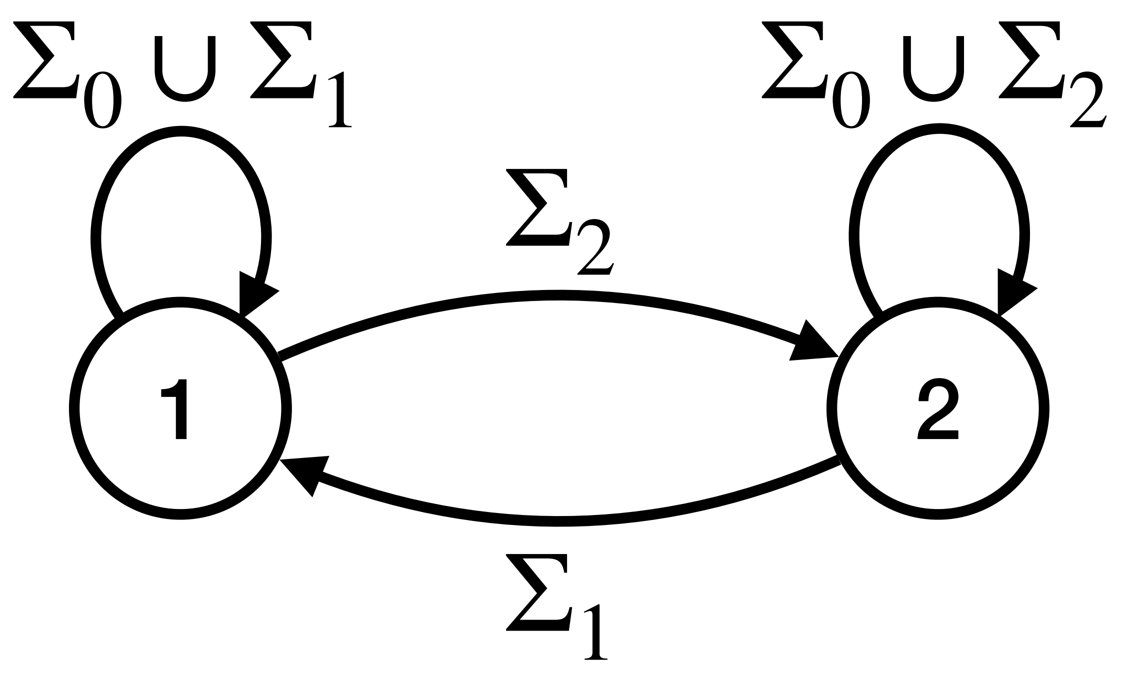
Theorem

Any (letter-to-letter) sequential transducer $\mathcal{T}$ can be realised by a cascade product of reset or permutation transducers:

$$\mathcal{T} \equiv \mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Reset or Permutation is a property of the underlying input automaton:

Reset - $\mathcal{U}_2$



Krohn-Rhodes

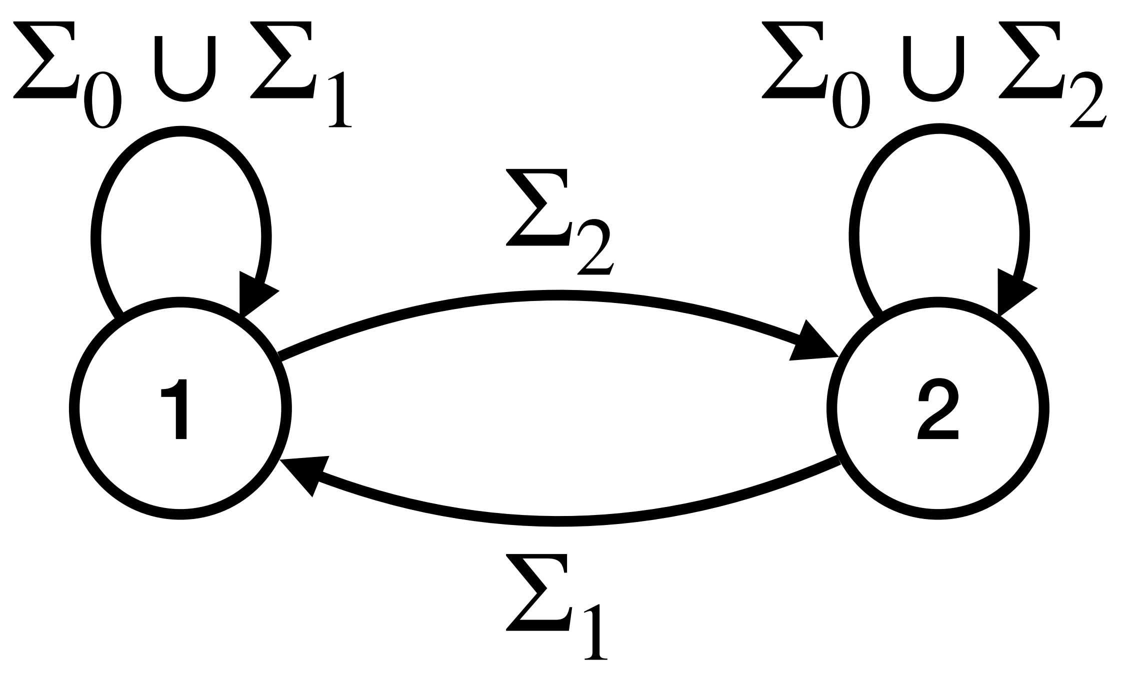
Theorem

Any (letter-to-letter) sequential transducer $\mathcal{T}$ can be realised by a cascade product of reset or permutation transducers:

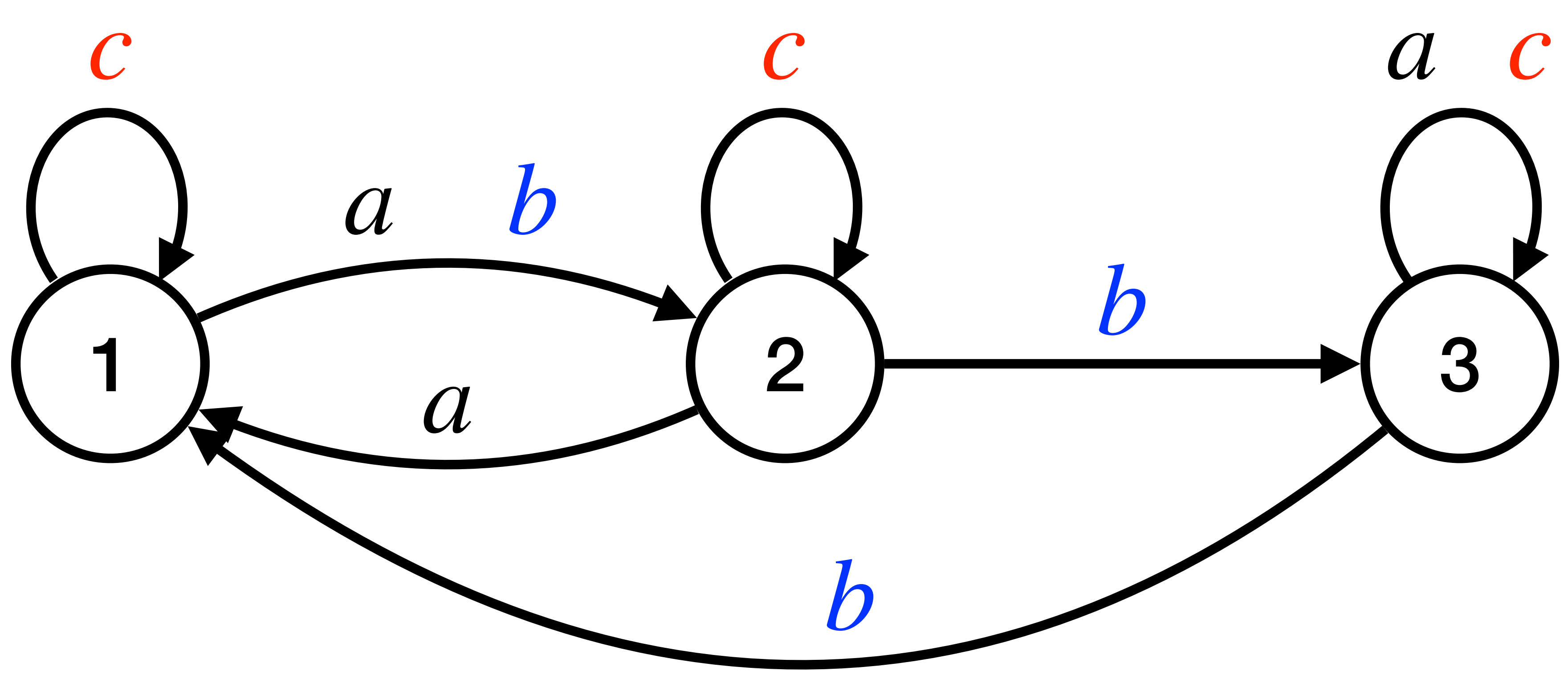
$$\mathcal{T} \equiv \mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Reset or Permutation is a property of the underlying input automaton:

Reset - $\mathcal{U}_2$

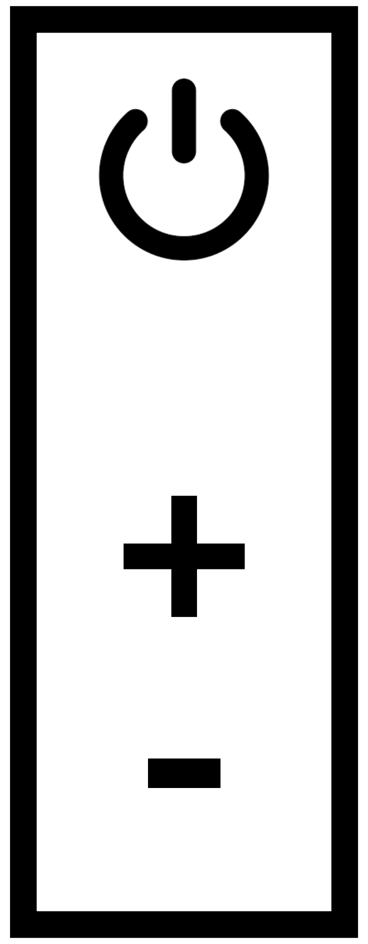


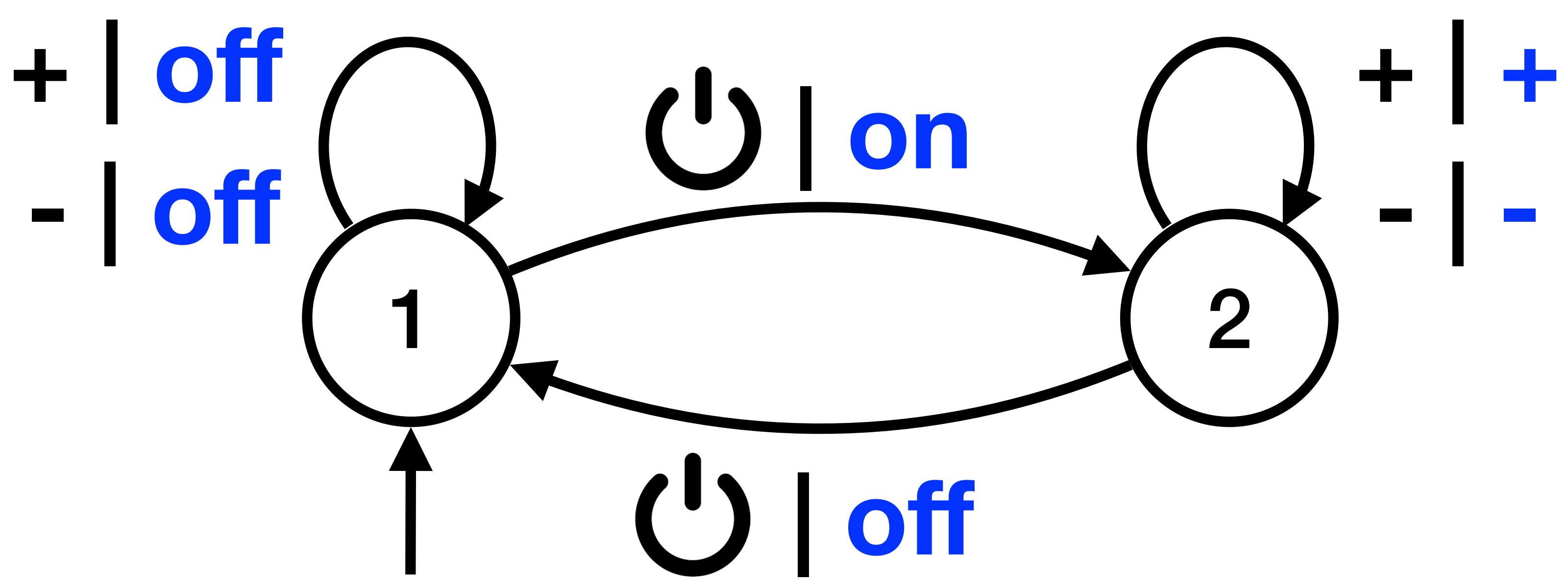
Permutation



- ⏻ - ⏻ + ⏻ - - ⏻ ⏻ + + + ⏻

O H M O O M L L O L M H H O

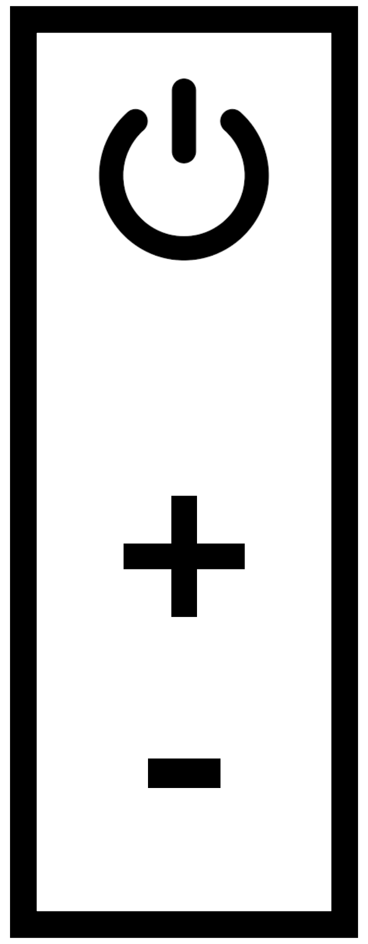


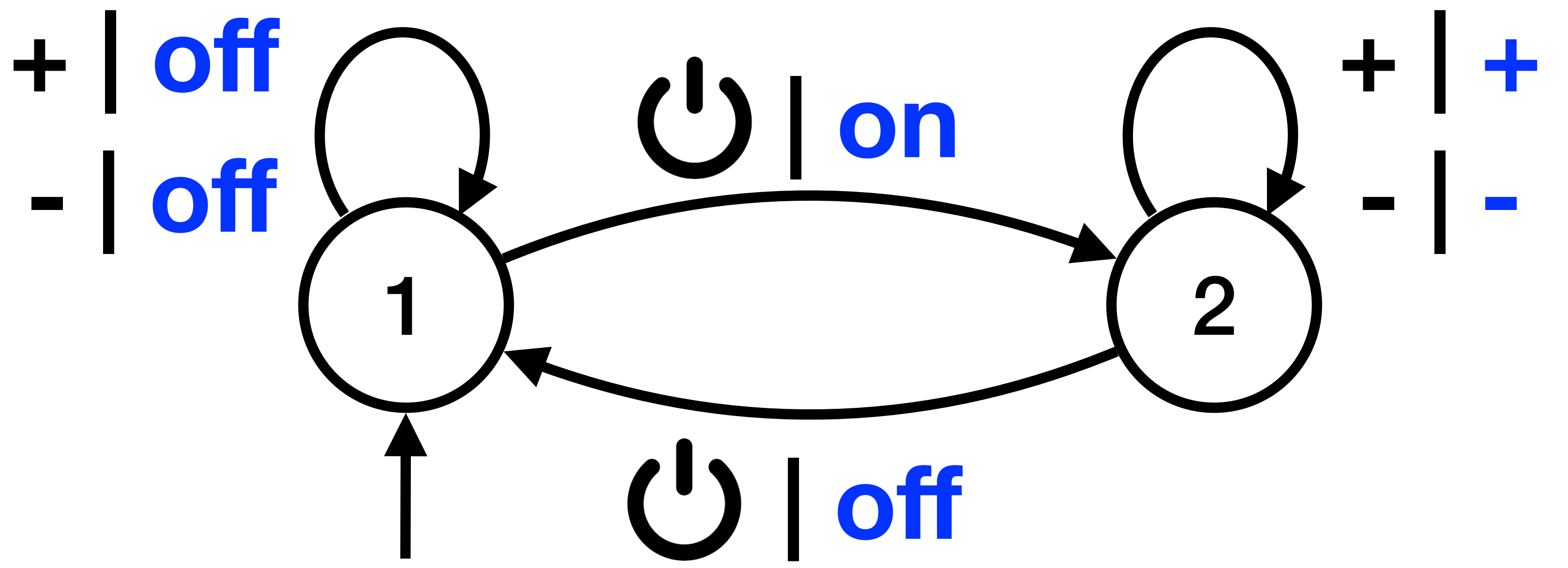


Permutation

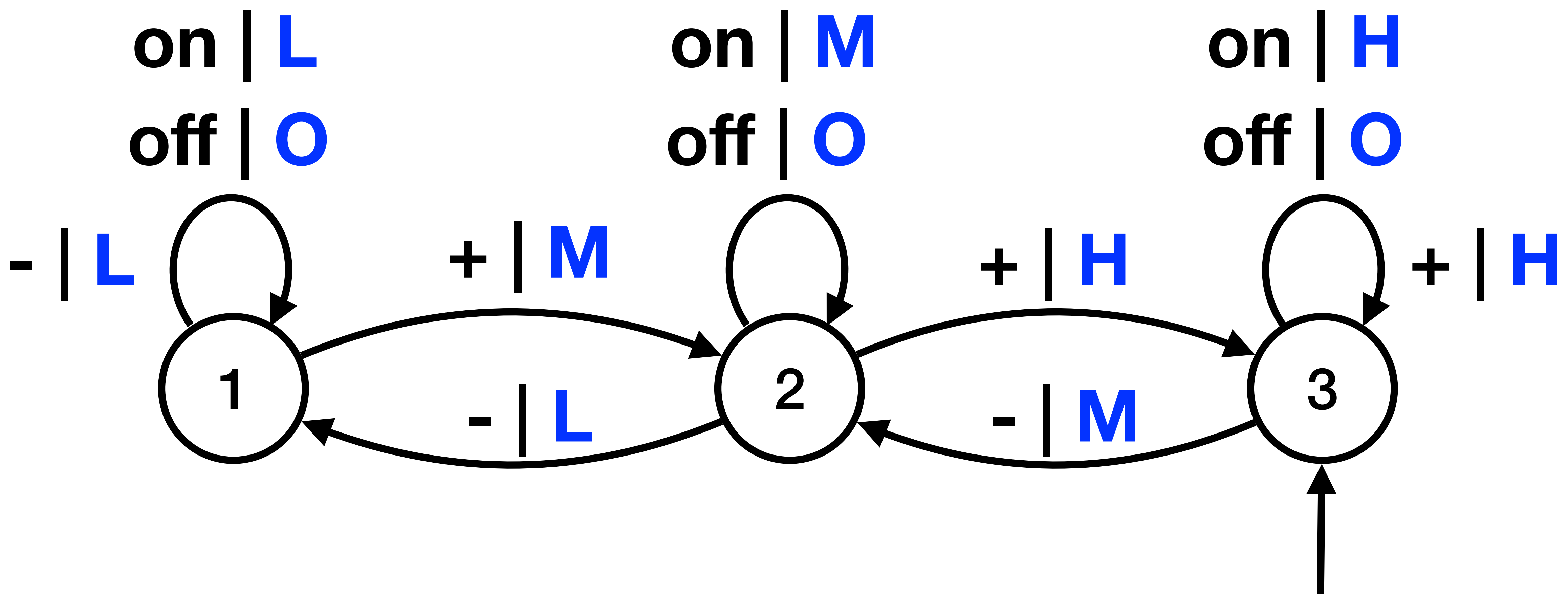
- ⏻ - ⏻ + ⏻ - - ⏻ ⏻ + + + ⏻

O H M O O M L L O L M H H O





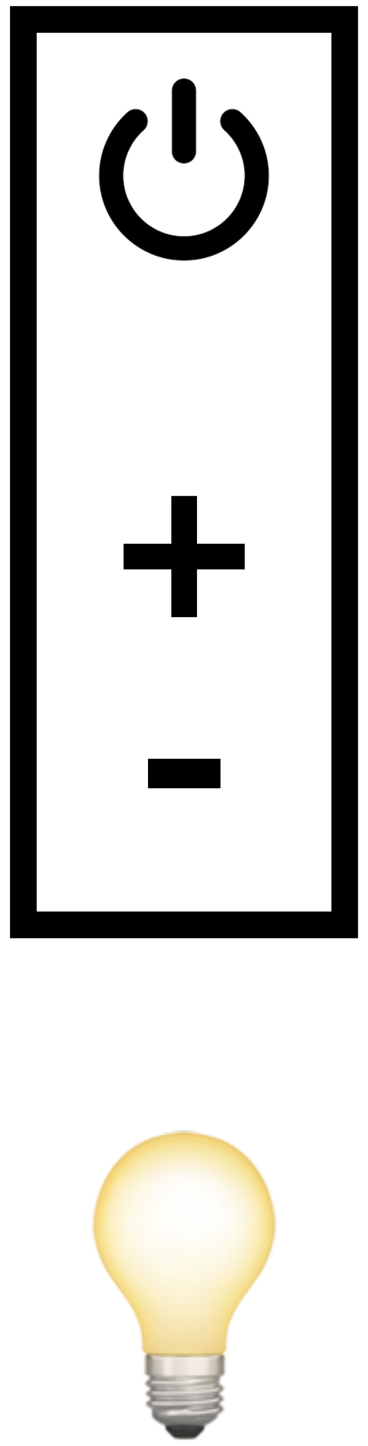
Permutation



Neither Reset nor Permutation

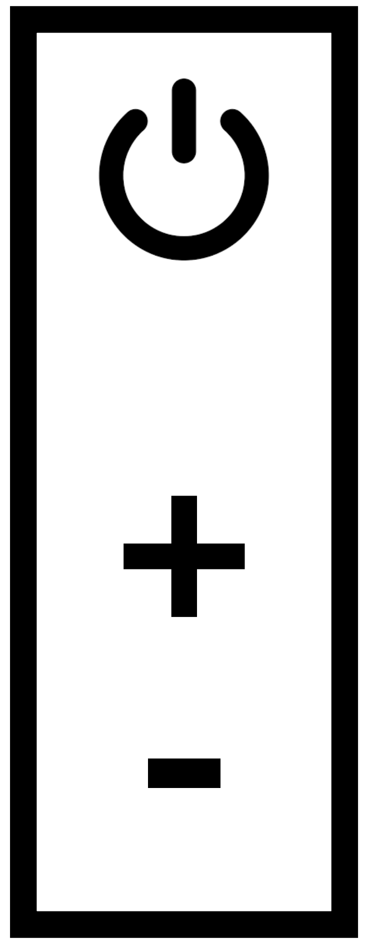
- ⏻ - ⏻ + ⏻ - + ⏻ ⏻ + - - ⏻

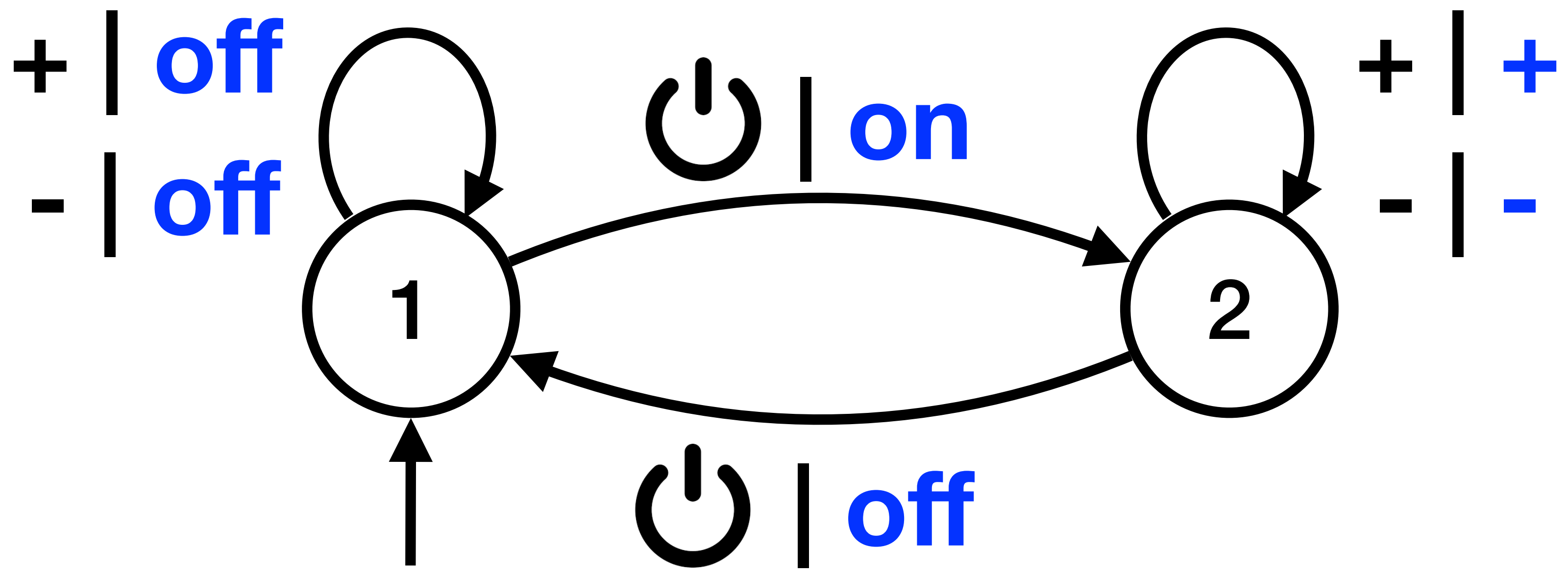
O H L O O L L H O H H L L O



- ⏻ - ⏻ + ⏻ - + ⏻ ⏻ + - - ⏻

O H L O O L L H O H H L L O

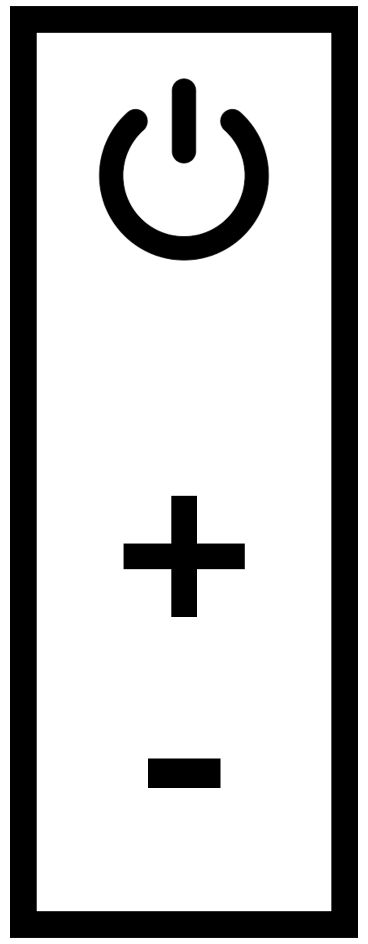


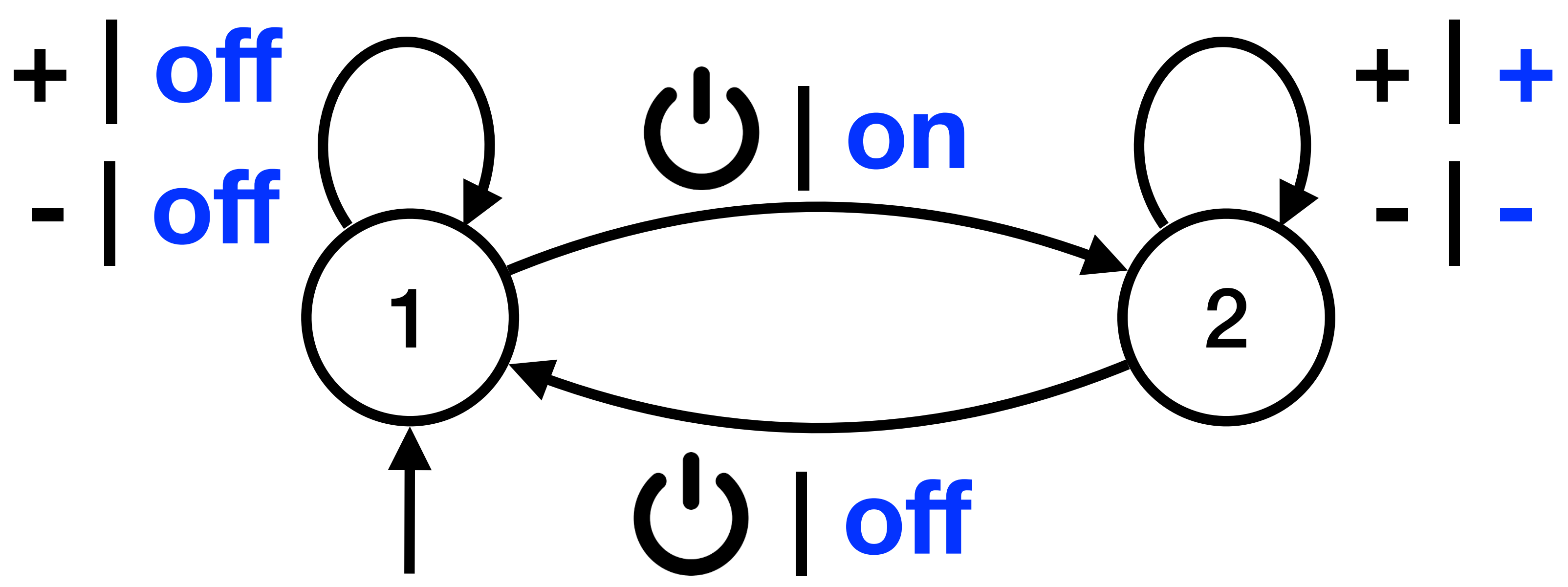


Permutation

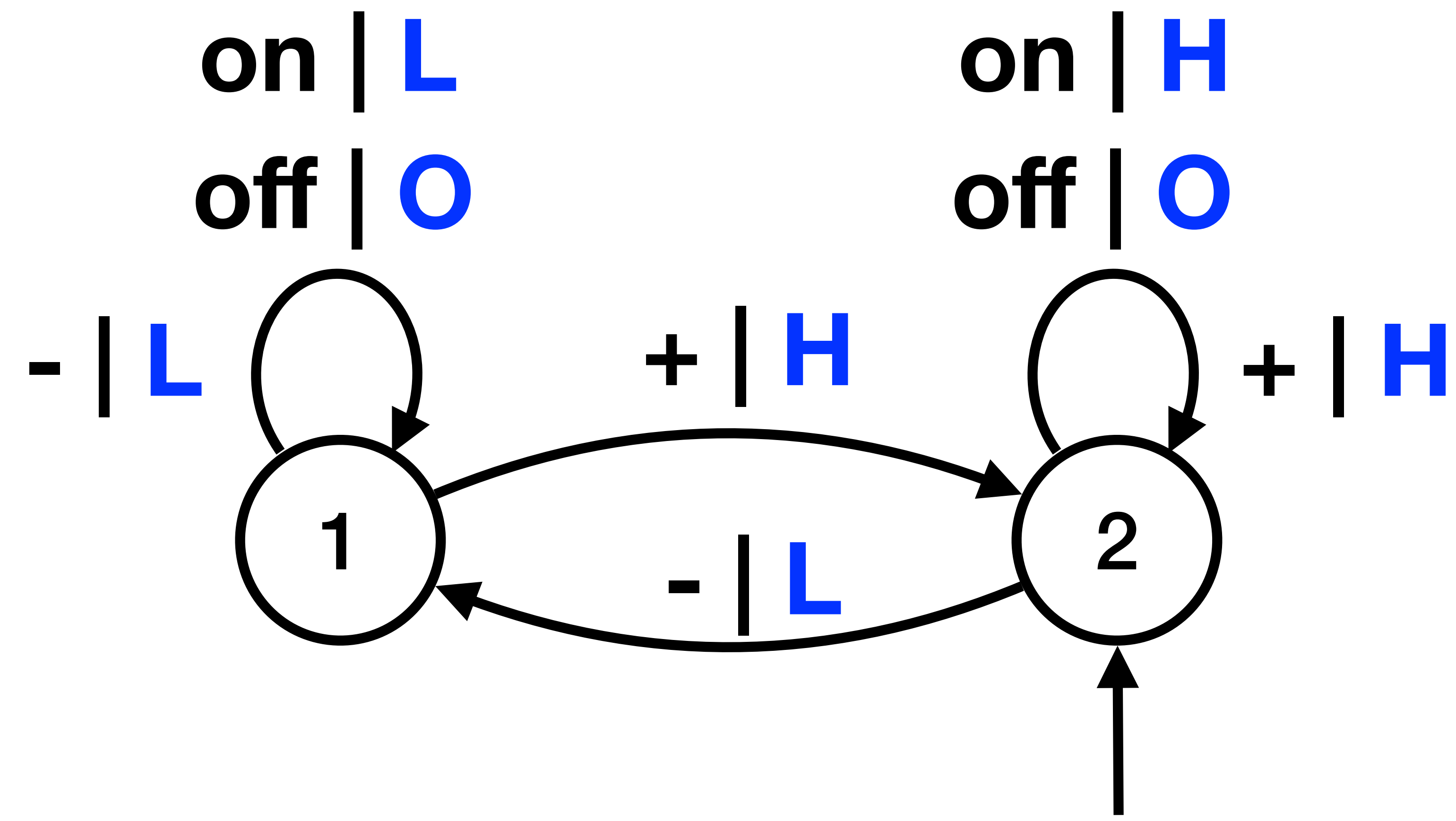
- ⏻ - ⏻ + ⏻ - + ⏻ ⏻ + - - ⏻

O H L O O L L H O H H L L O



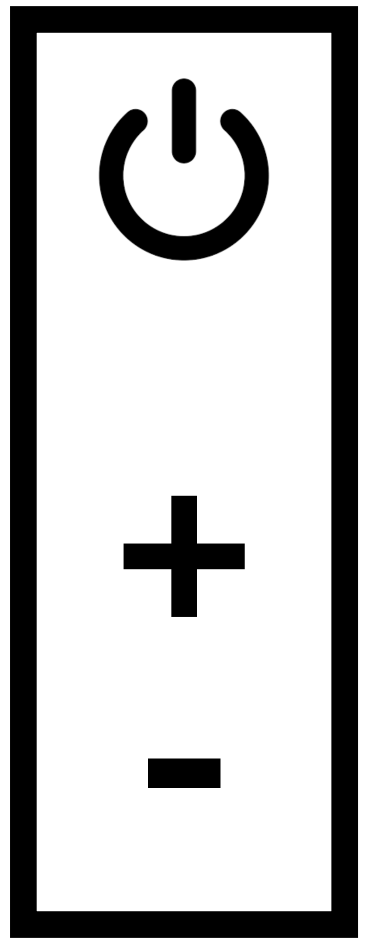


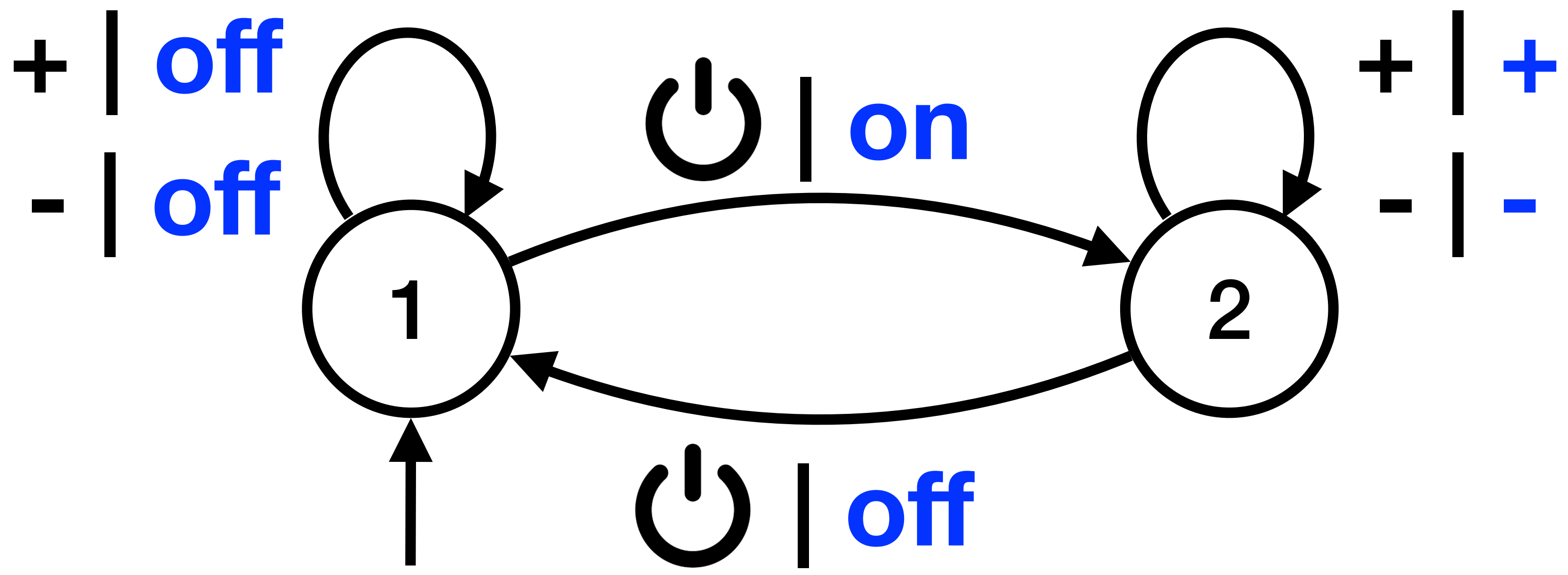
Permutation



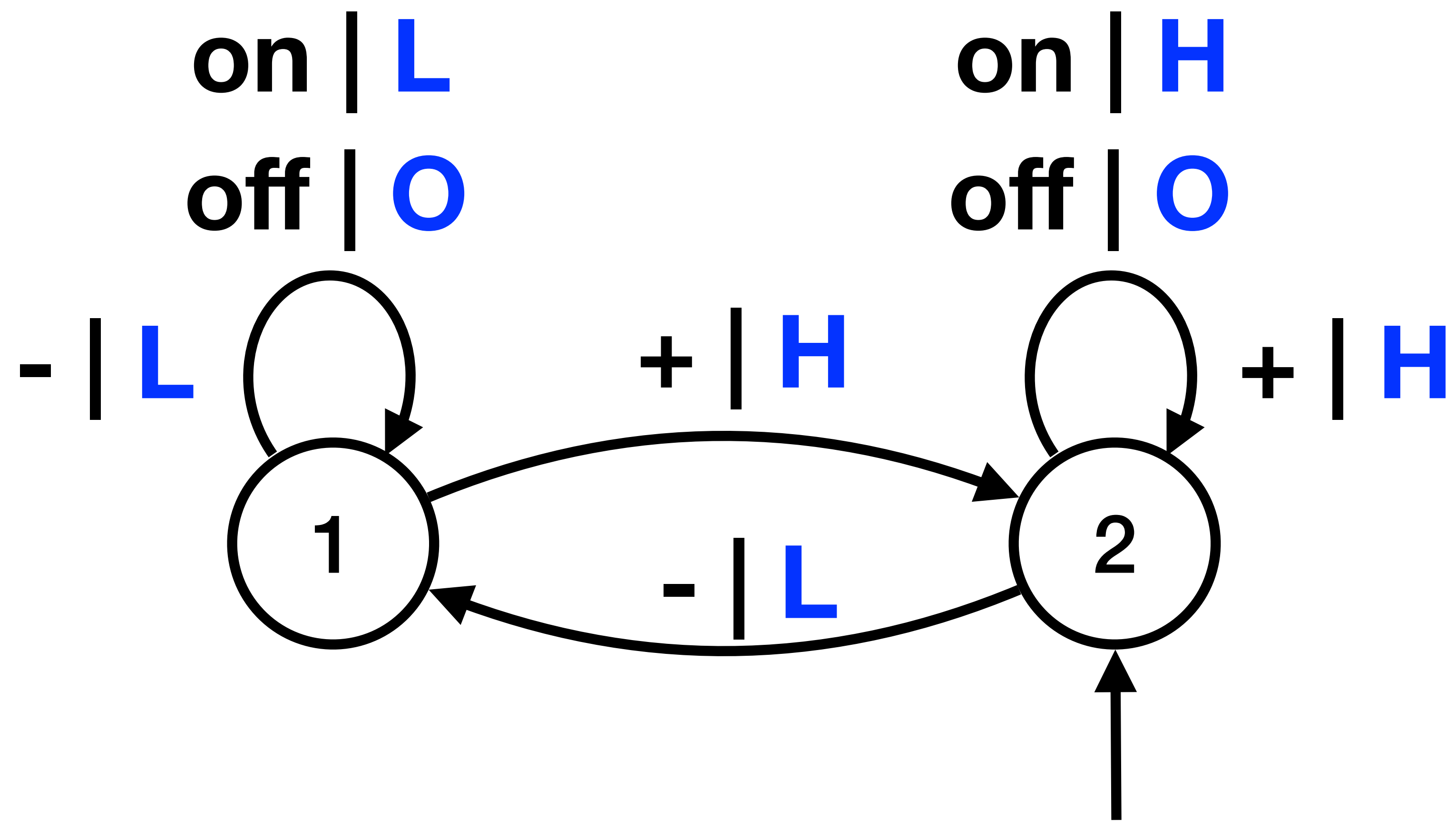
- ⏻ - ⏻ + ⏻ - + ⏻ ⏻ + - - ⏻

O H L O O L L H O H H L L O





Permutation

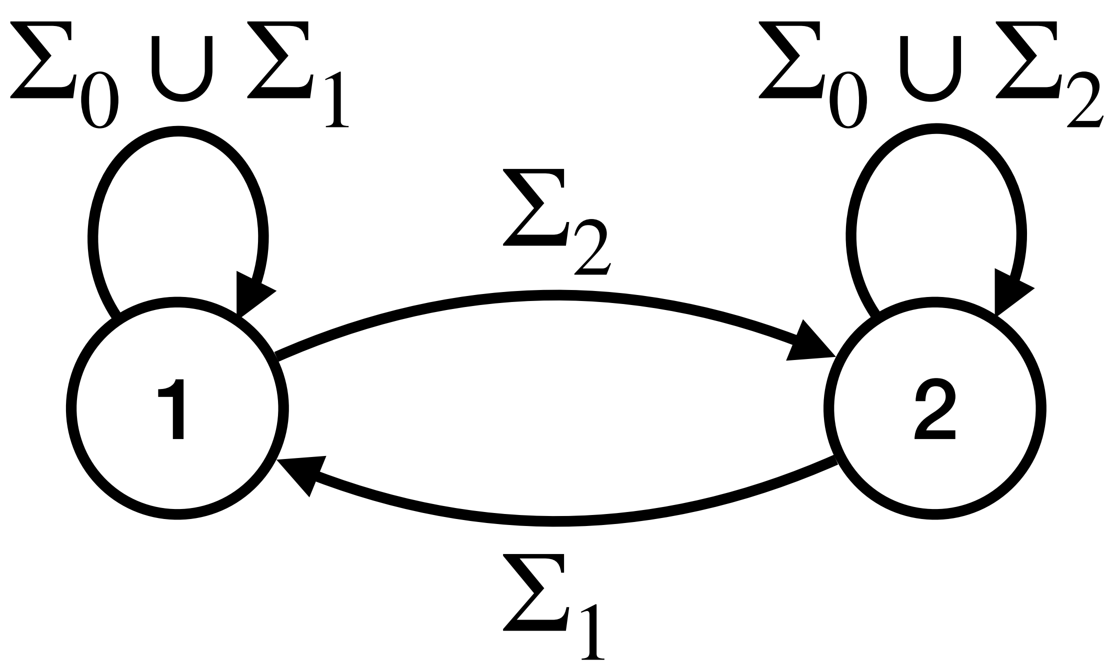


Reset

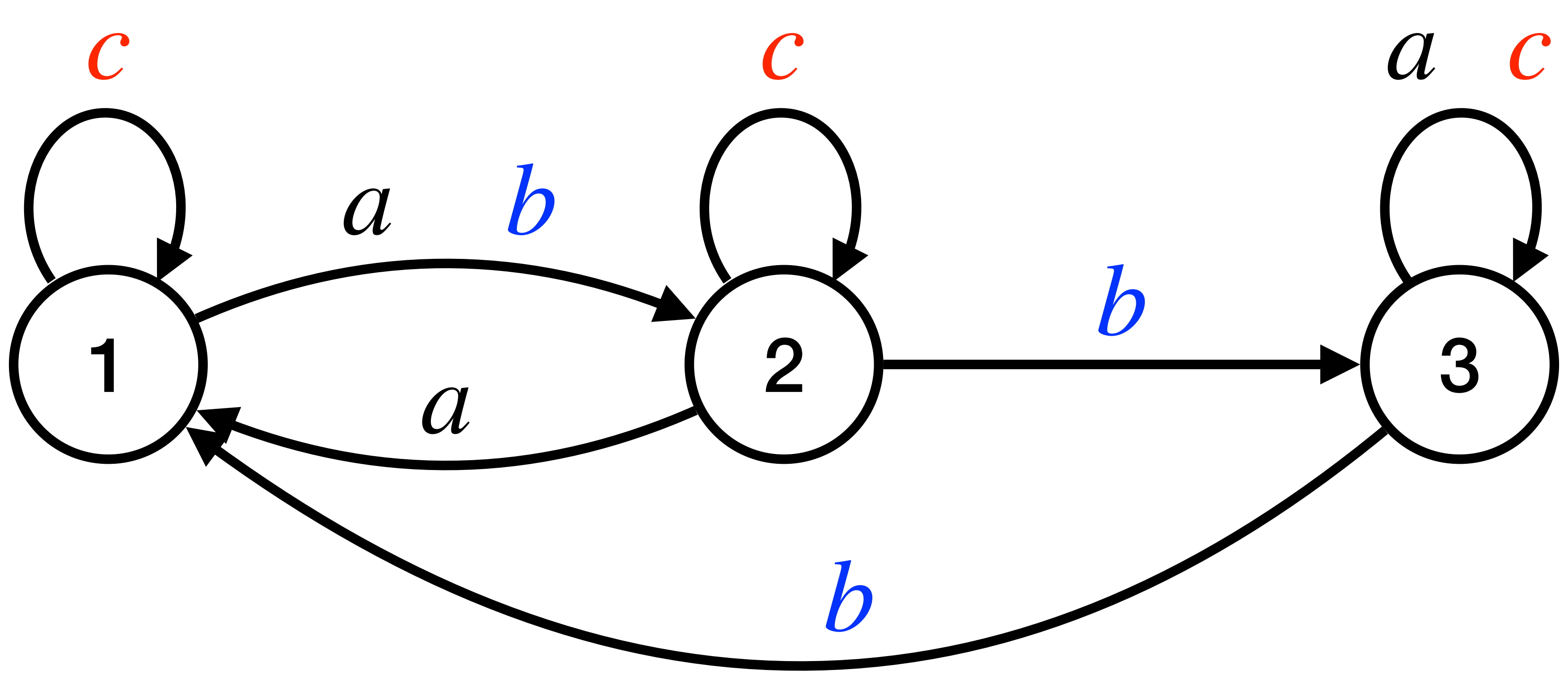
Krohn-Rhodes

Reset or Permutation automata:

Reset - $\mathscr{U}_2$



Permutation



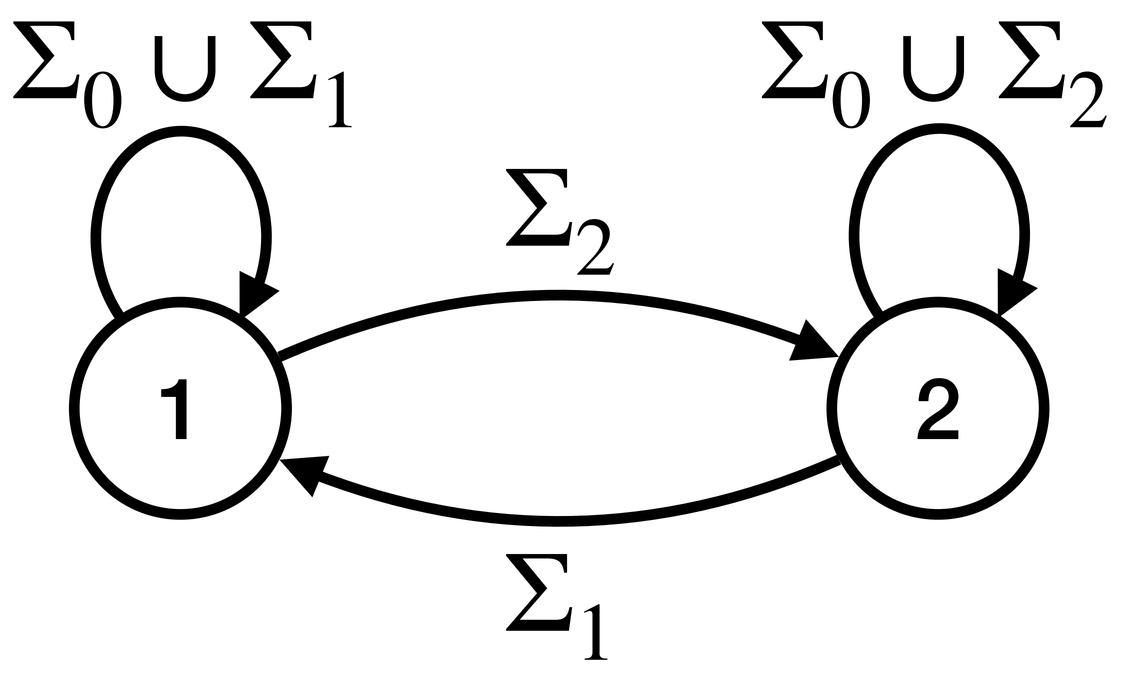
Krohn-Rhodes

Theorem

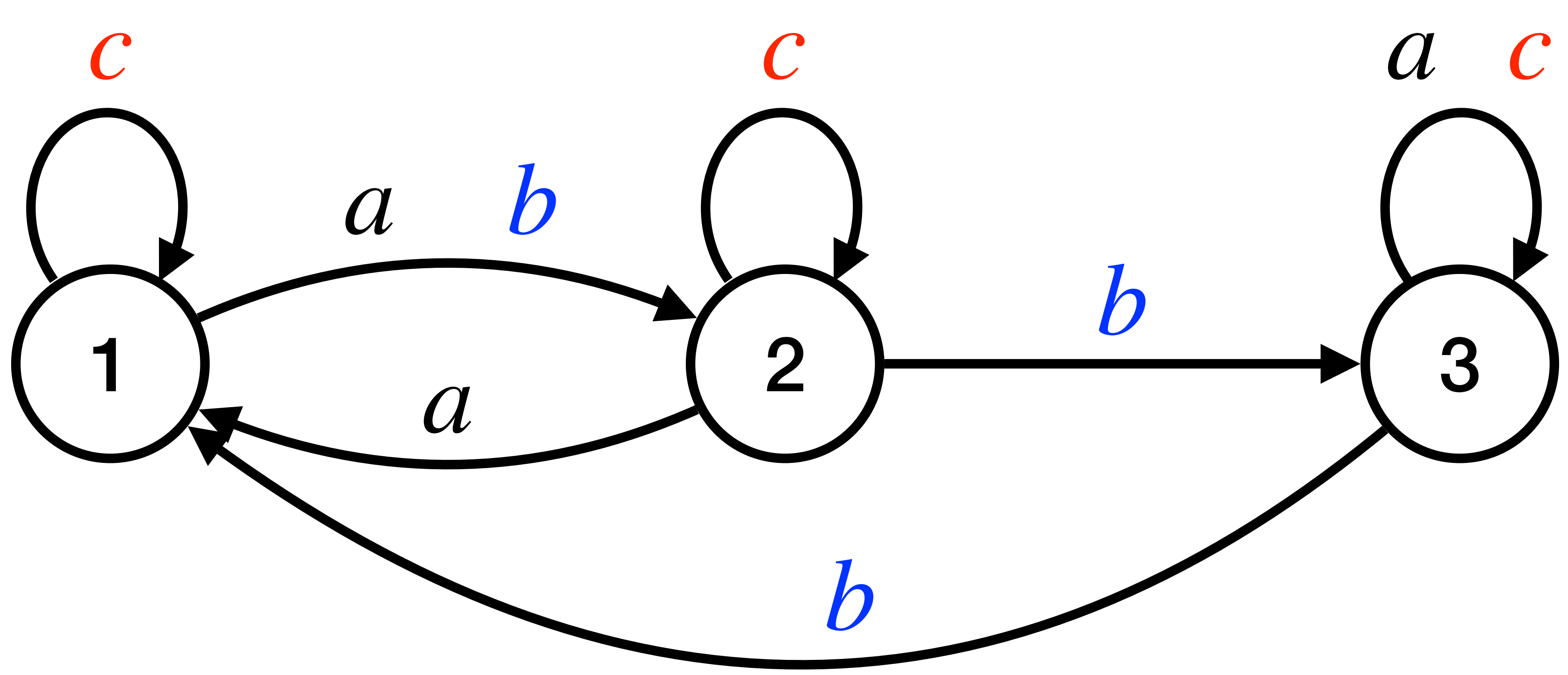
- Any regular language can be accepted by a cascade product of reset or permutation automata.

Reset or Permutation automata:

Reset - $\mathscr{U}_2$



Permutation



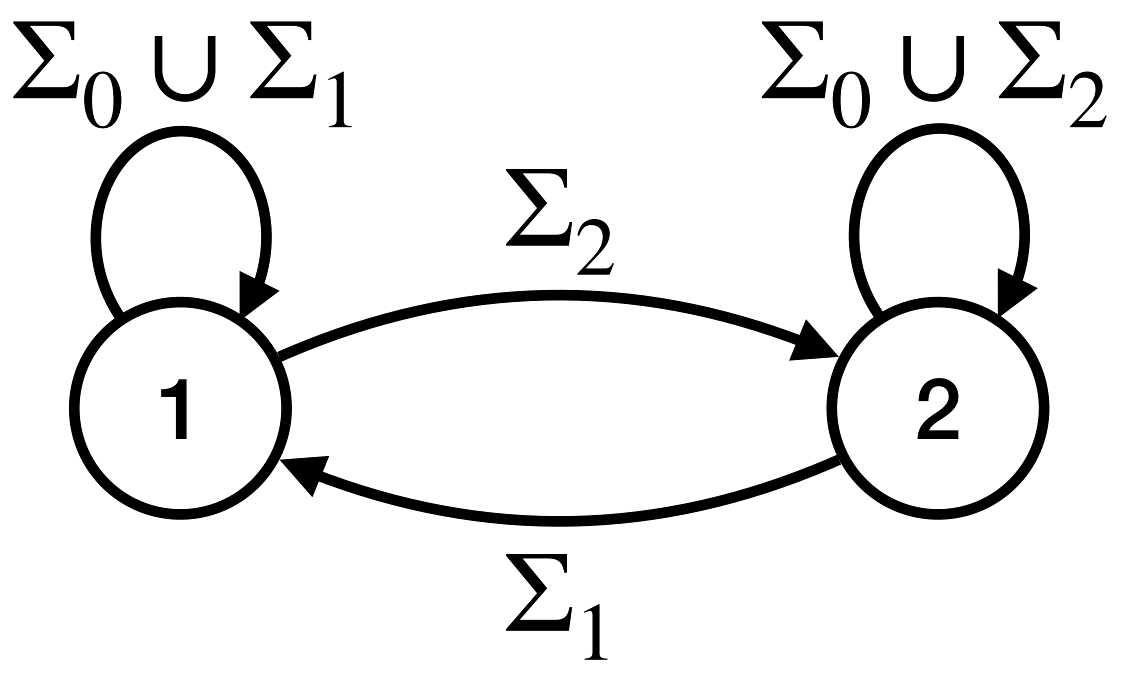
Krohn-Rhodes

Theorem

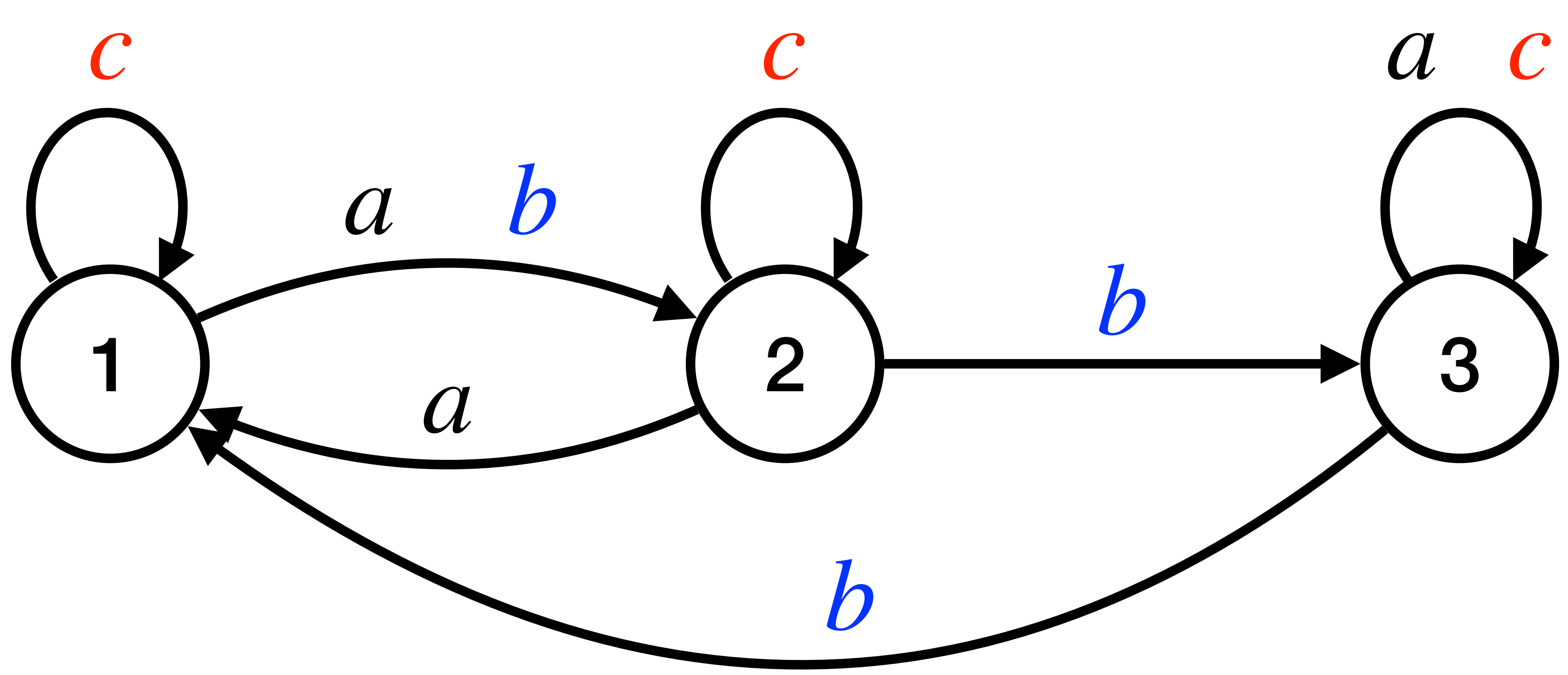
- Any regular language can be accepted by a cascade product of reset or permutation automata.
- Any aperiodic language can be accepted by a cascade product of reset automata.

Reset or Permutation automata:

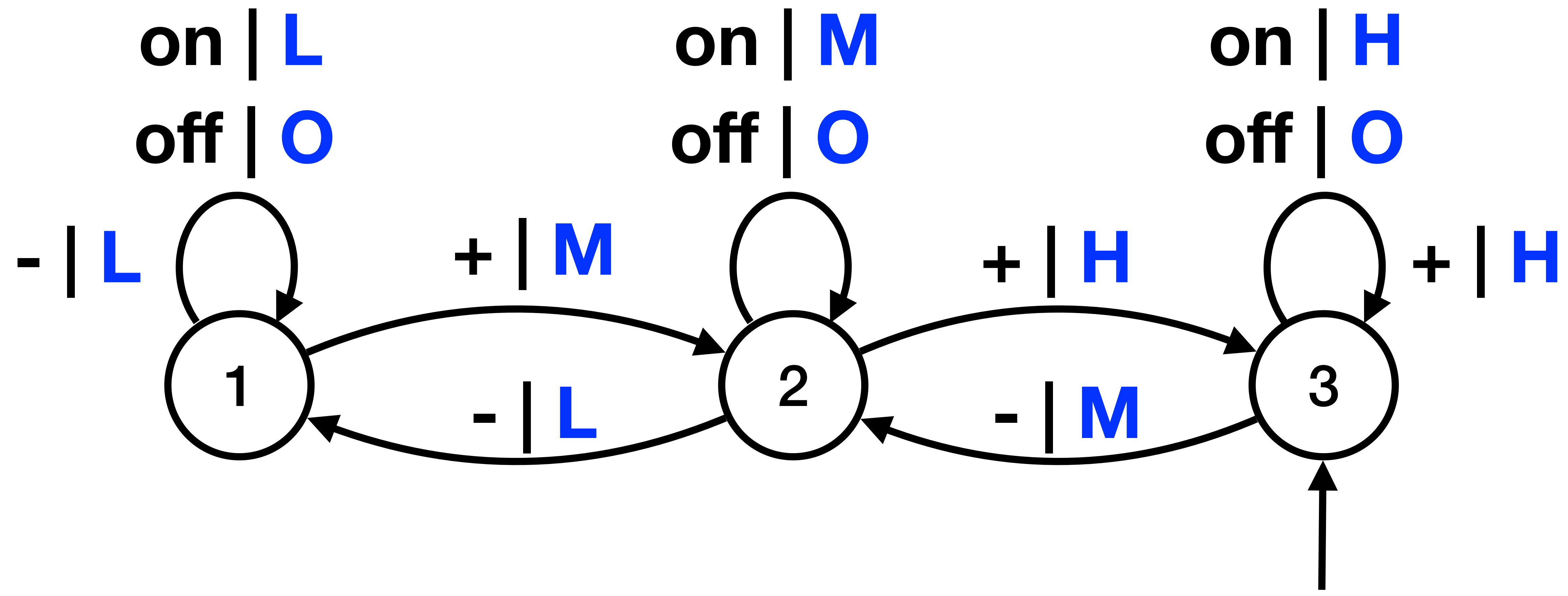
Reset - $\mathscr{U}_2$



Permutation



KR proof for aperiodic



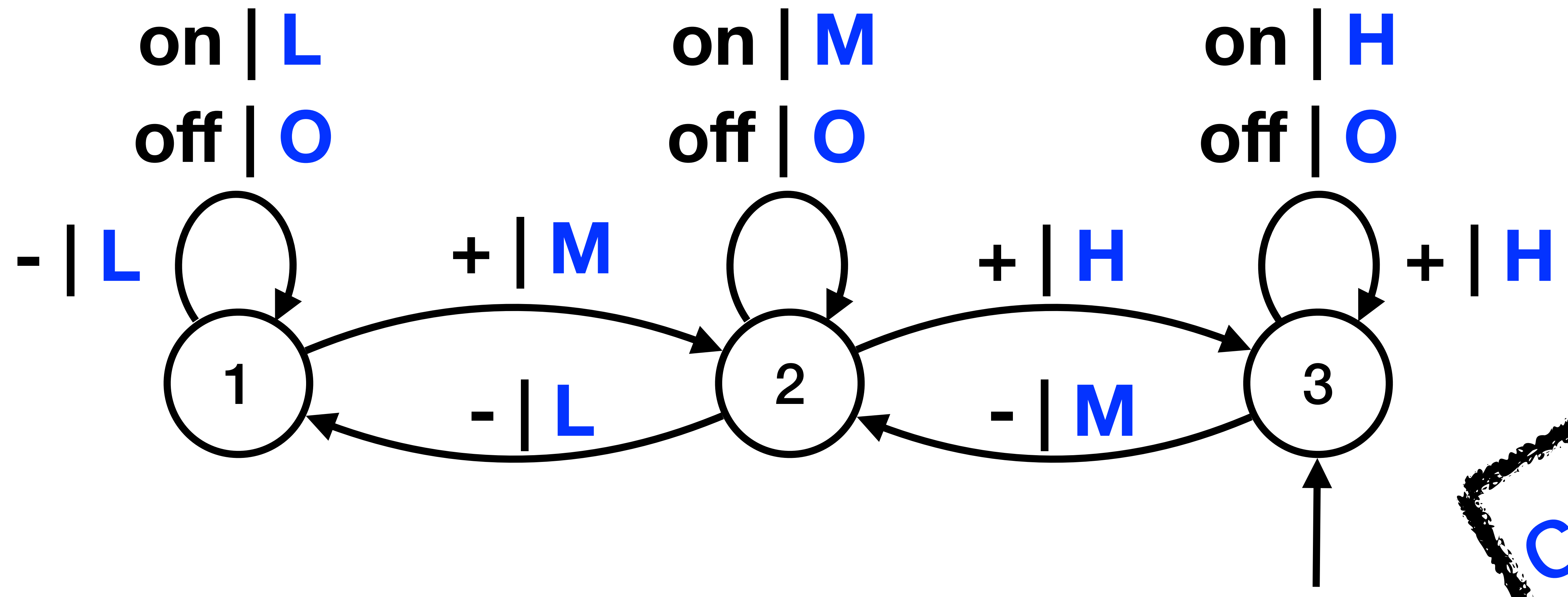
- If we ignore on and off

State_1 = $\{ + , - \}^* - - (+ -)^*$

- With $A = \{\text{on}, \text{off}, + , - \}$ and $B = \{\text{on}, \text{off}\}$

State_1 = $A^* - B^* - B^* (+ B^* - B^*)^*$

KR proof for aperiodic



cascade product of reset transducers

- If we ignore on and off

State_1 = $\{ + , - \}^* - - (+ -)^*$

- With $A = \{\text{on}, \text{off}, + , - \}$ and $B = \{\text{on}, \text{off}\}$

State_1 = $A^* - B^* - B^* (+ B^* - B^*)^*$

KR proof for aperiodic

Theorem (Kamp)

Aperiodic = Past Temporal Logic

- If we ignore on and off

State_1 = $\{ + , - \}^* - - (+ -)^*$

KR proof for aperiodic

Theorem (Kamp)

Aperiodic = Past Temporal Logic

- If we ignore on and off

State_1 = $\{ + , - \}^* - - (+ -)^*$

$$- \wedge \Big((Y -) \; \vee \; (+ \to Y -) \; SS \; (- \wedge Y -) \Big)$$

KR proof for aperiodic

Theorem (Kamp)

Aperiodic = Past Temporal Logic

Each PastLTL formula φ defines a boolean labelling function

$$\theta_\varphi : \Sigma^* \to \{0,1\}$$

Each position is labelled with the truth value of φ at this position.

KR proof for aperiodic

Theorem (Kamp)

Aperiodic = Past Temporal Logic

Each PastLTL formula φ defines a boolean labelling function

$$\theta_\varphi : \Sigma^* \to \{0,1\}$$

Each position is labelled with the truth value of φ at this position.

Example $\varphi = \mathsf{Y}\, a$

a b b b a a b a b b a
0 1 0 0 0 1 1 0 1 0 0

KR proof for aperiodic

Theorem (Kamp)

Aperiodic = Past Temporal Logic

Each PastLTL formula φ defines a boolean labelling function

$$\theta_\varphi : \Sigma^* \to \{0,1\}$$

Each position is labelled with the truth value of φ at this position.

Example $\varphi = \mathsf{Y}\, a$

a b b b a a b a b b a
0 1 0 0 0 1 1 0 1 0 0

Example $\varphi = a \;\mathsf{SS}\; b$

a b b a a c c b a a c
0 0 1 1 1 1 0 0 1 1 1

KR proof for aperiodic

Theorem (Kamp)

Aperiodic = Past Temporal Logic

Each PastLTL formula φ defines a boolean labelling function

$$\theta_\varphi : \Sigma^* \to \{0,1\}$$

Each position is labelled with the truth value of φ at this position.

KR proof for aperiodic

Theorem (Kamp)

Aperiodic = Past Temporal Logic

Each PastLTL formula φ defines a boolean labelling function

$$\theta_\varphi : \Sigma^* \to \{0,1\}$$

Each position is labelled with the truth value of φ at this position.

Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathcal{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

KR proof for aperiodic

Theorem (Kamp)

Aperiodic = Past Temporal Logic

Each PastLTL formula φ defines a boolean labelling function

$$\theta_\varphi : \Sigma^* \to \{0,1\}$$

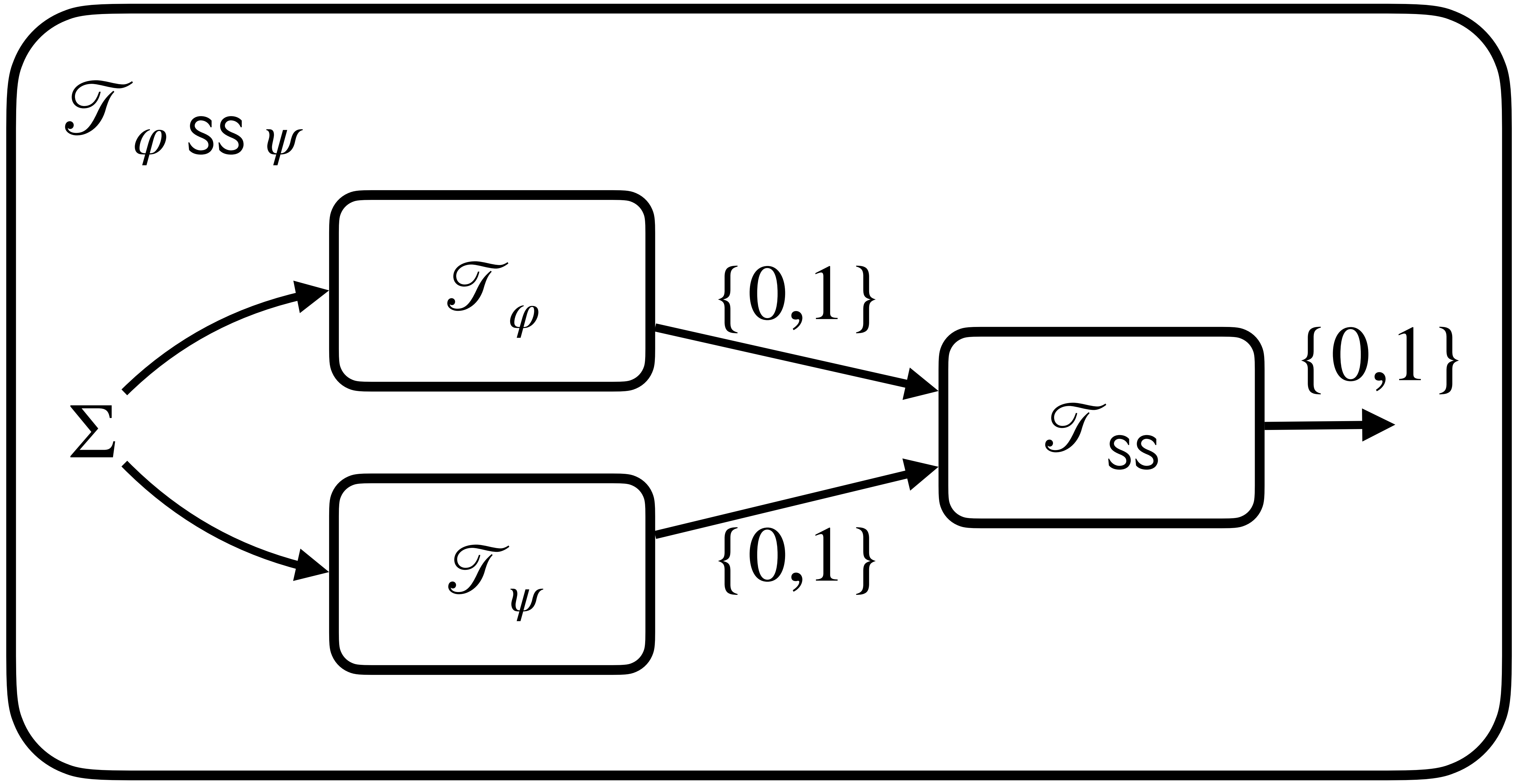
Each position is labelled with the truth value of φ at this position.

Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathscr{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

PastLTL: boolean connectives and strict-since

$$\mathsf{Y}\,\psi = \bot\ \mathsf{SS}\ \psi \quad \text{and} \quad \varphi\ \mathsf{S}\ \psi = \psi \vee (\varphi \wedge (\varphi\ \mathsf{SS}\ \psi))$$

KR proof for aperiodic

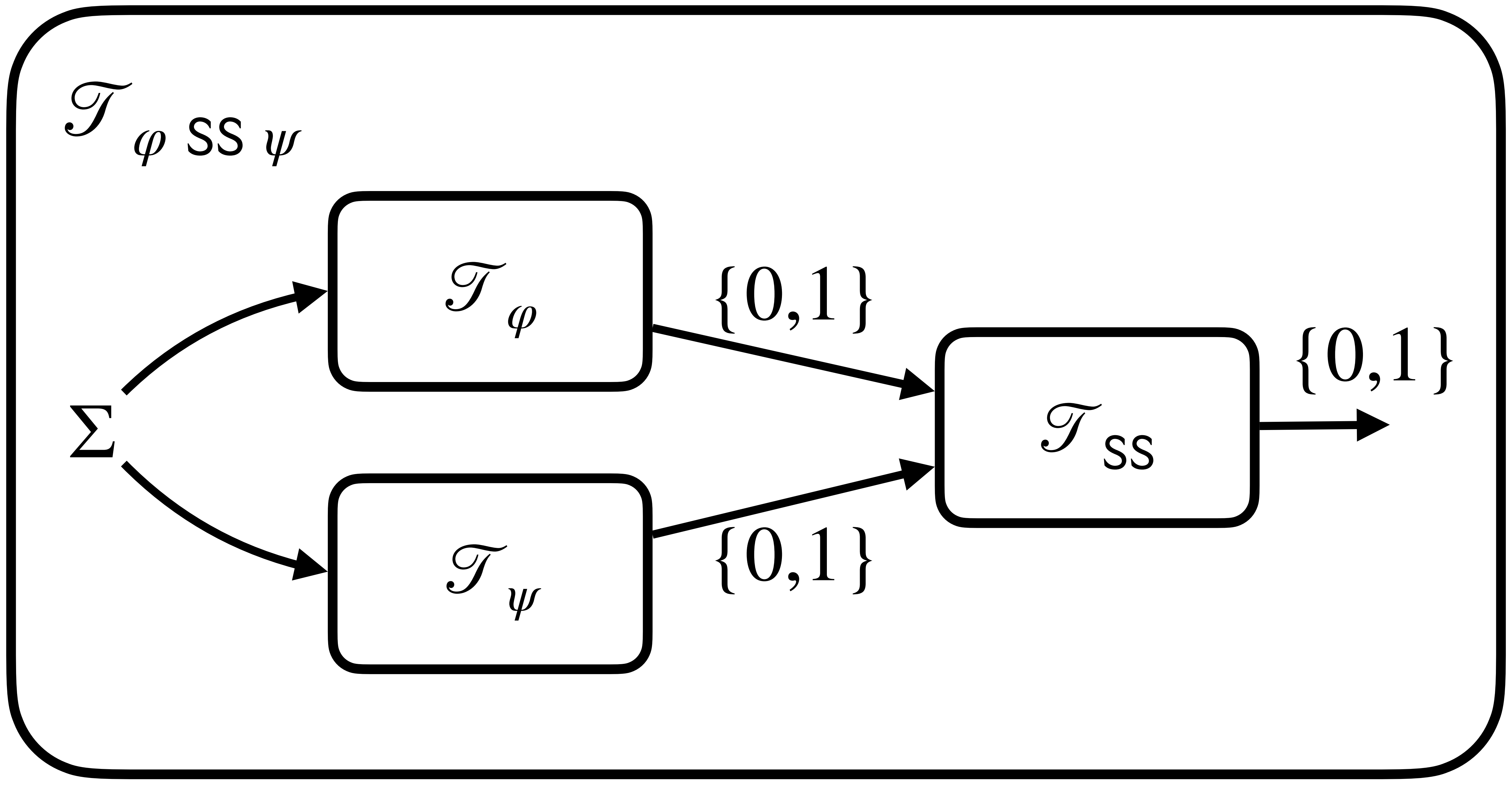


Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathcal{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

PastLTL: boolean connectives and strict-since

$$\text{Y}\,\psi = \bot \text{ SS } \psi \quad \text{and} \quad \varphi \text{ S } \psi = \psi \vee (\varphi \wedge (\varphi \text{ SS } \psi))$$

KR proof for aperiodic



Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathscr{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

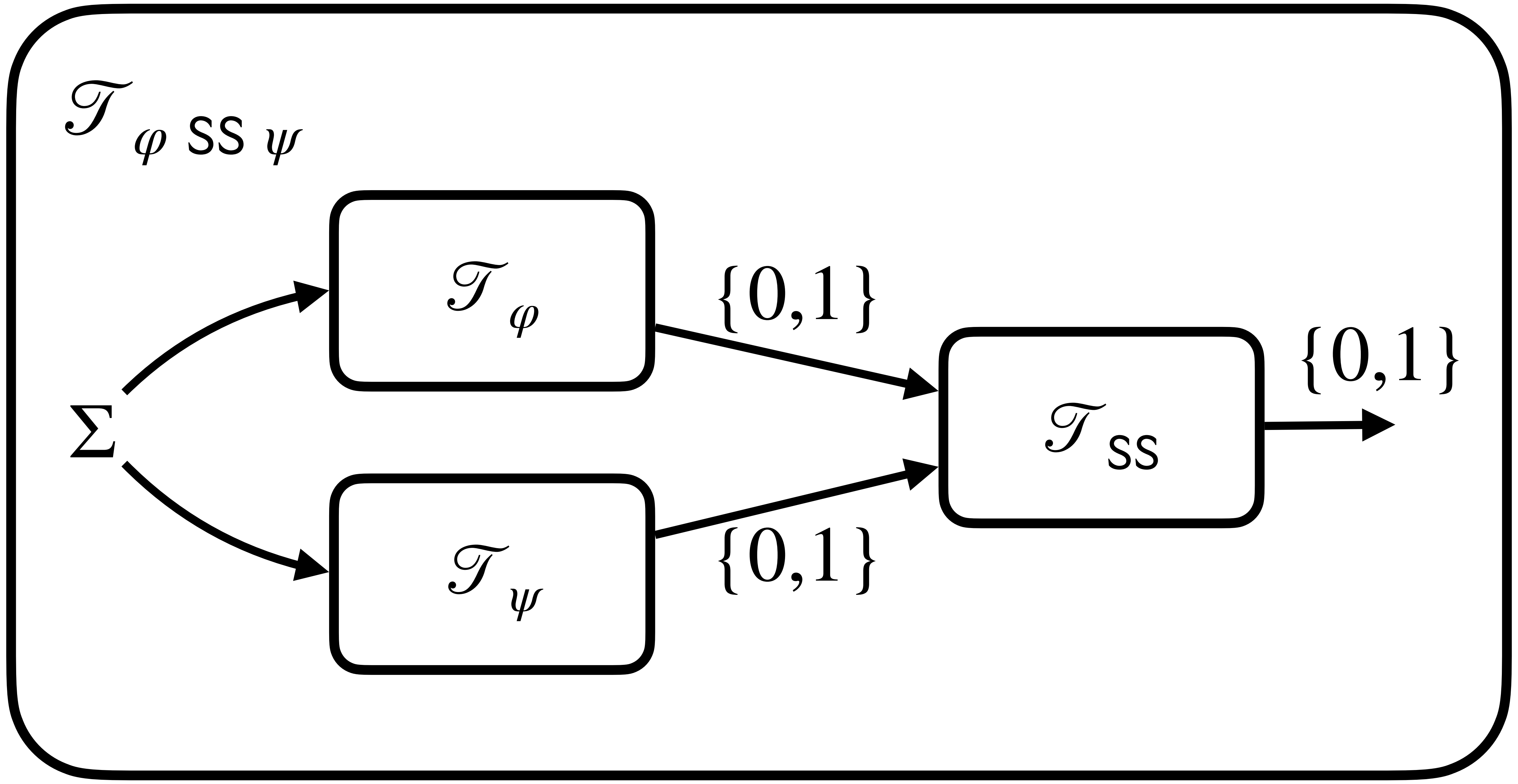
Example $\varphi = a$ SS b

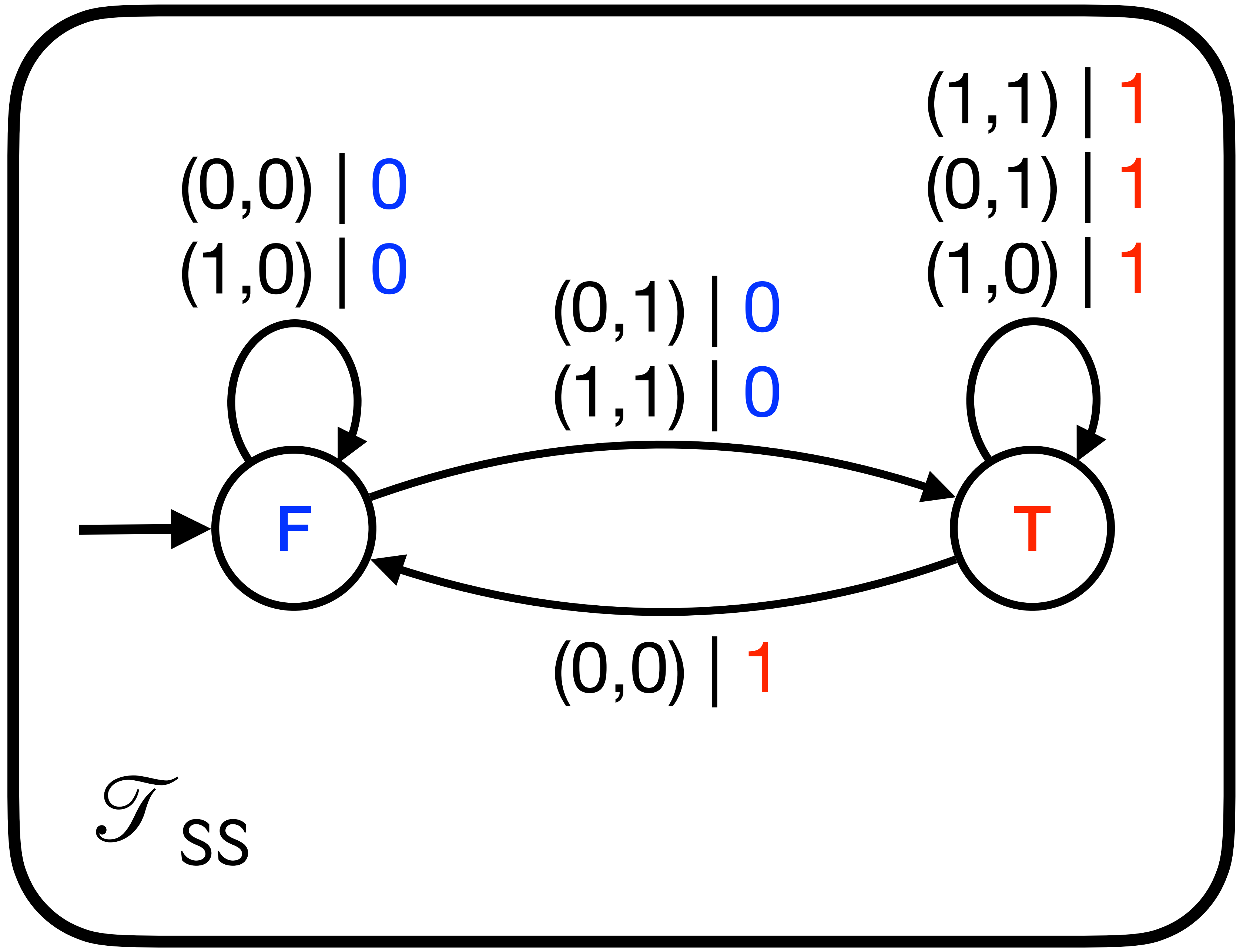
a b b a a c c b a a c
0 0 1 1 1 1 0 0 1 1 1

PastLTL: boolean connectives and strict-since

$\text{Y}\,\psi = \bot \text{ SS } \psi$ and $\varphi \text{ S } \psi = \psi \vee (\varphi \wedge (\varphi \text{ SS } \psi))$

KR proof for aperiodic





Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathscr{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

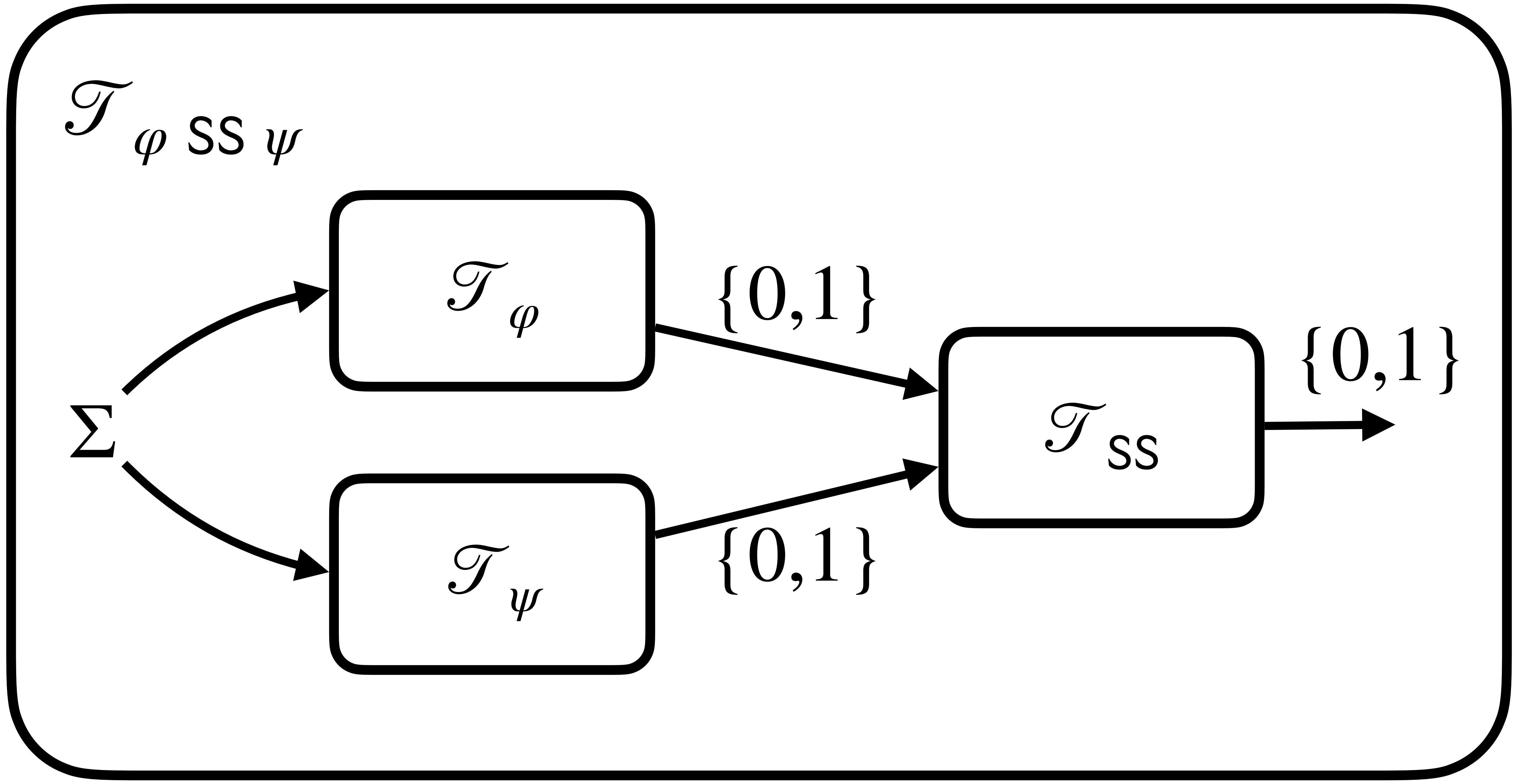
Example $\varphi = a$ SS b

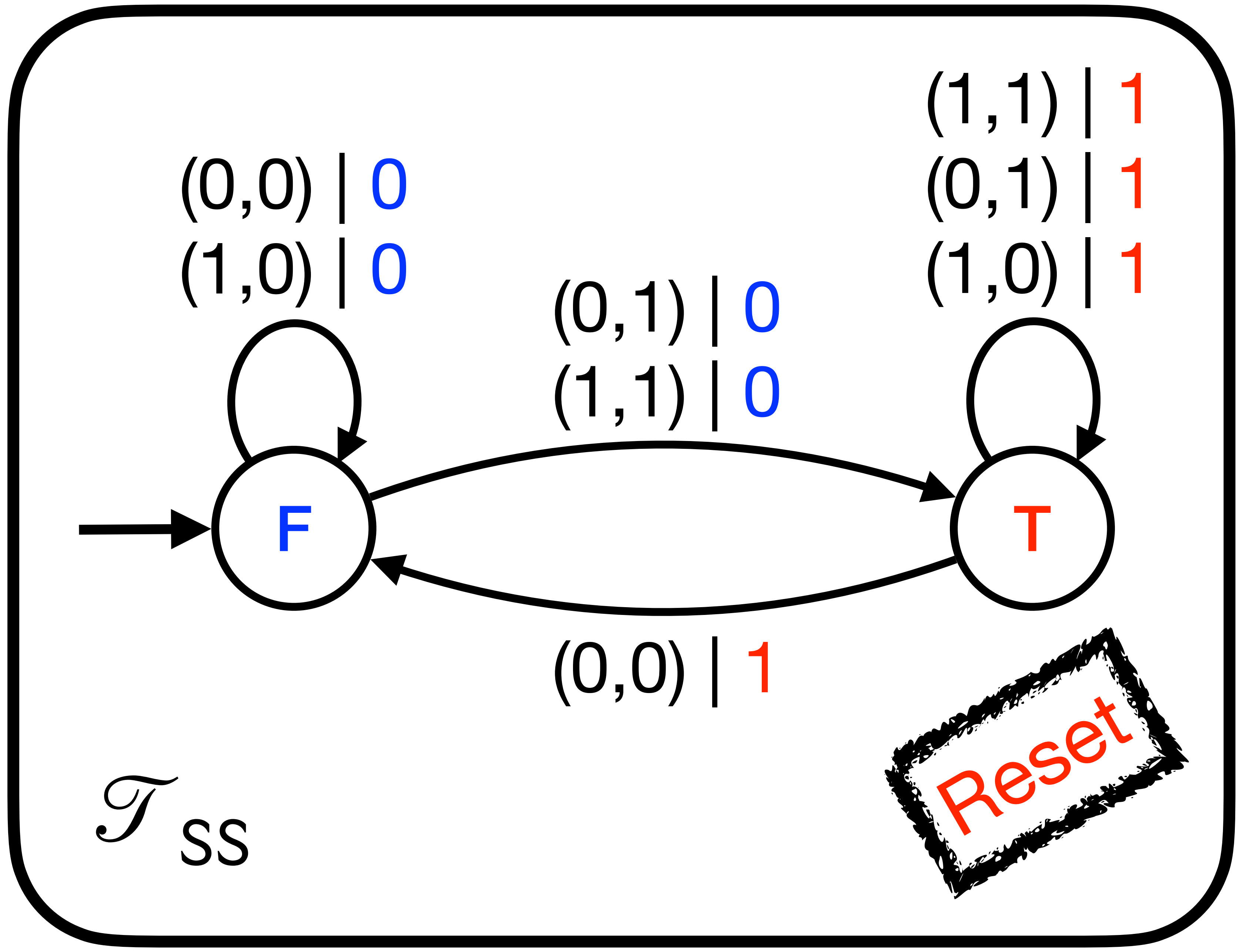
a	b	b	a	a	c	c	b	a	a	c
0	0	1	1	1	1	0	0	1	1	1

PastLTL: boolean connectives and strict-since

$\mathsf{Y}\,\psi = \bot \text{ SS } \psi$ and $\varphi \text{ S } \psi = \psi \vee (\varphi \wedge (\varphi \text{ SS } \psi))$

KR proof for aperiodic





Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathscr{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

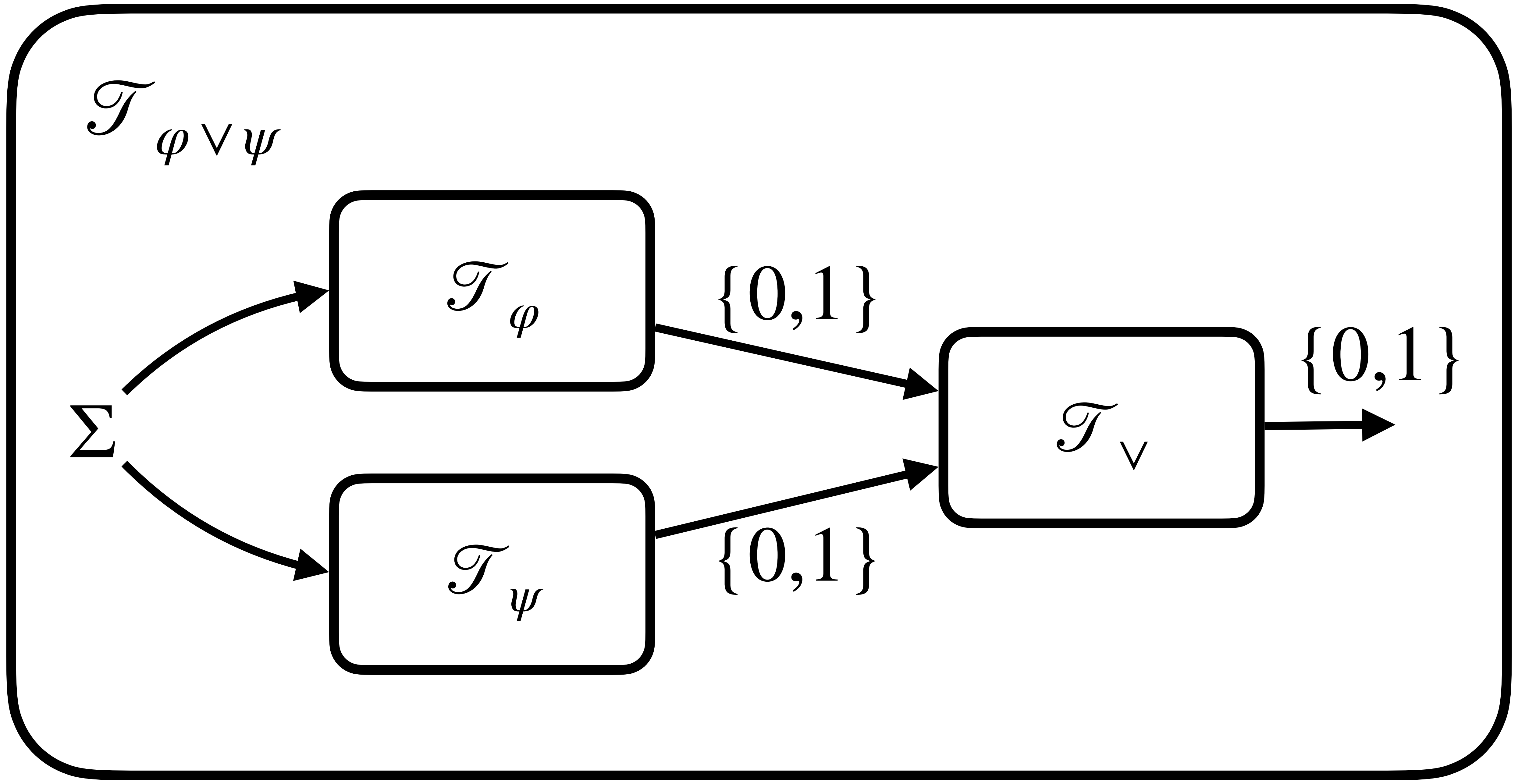
Example $\varphi = a$ SS b

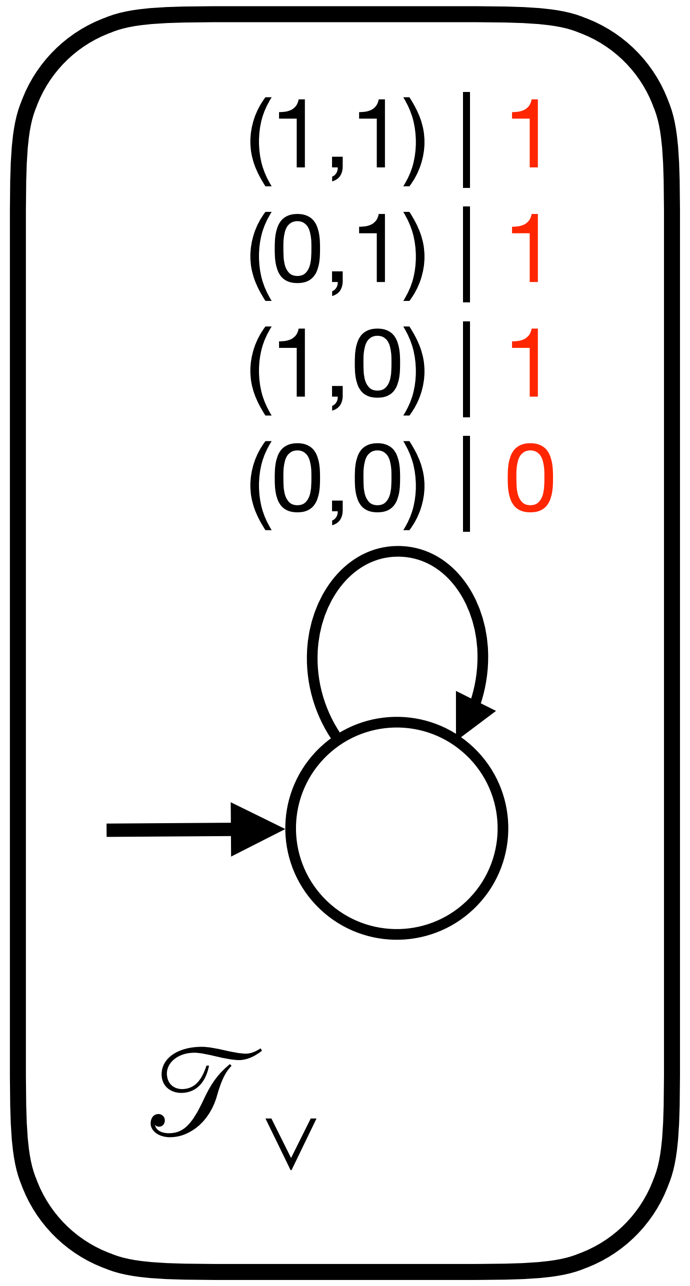
a b b a a c c b a a c
0 0 1 1 1 1 0 0 1 1 1

PastLTL: boolean connectives and strict-since

$\mathsf{Y}\,\psi = \bot\ \mathsf{SS}\ \psi$ and $\varphi\ \mathsf{S}\ \psi = \psi \vee (\varphi \wedge (\varphi\ \mathsf{SS}\ \psi))$

KR proof for aperiodic

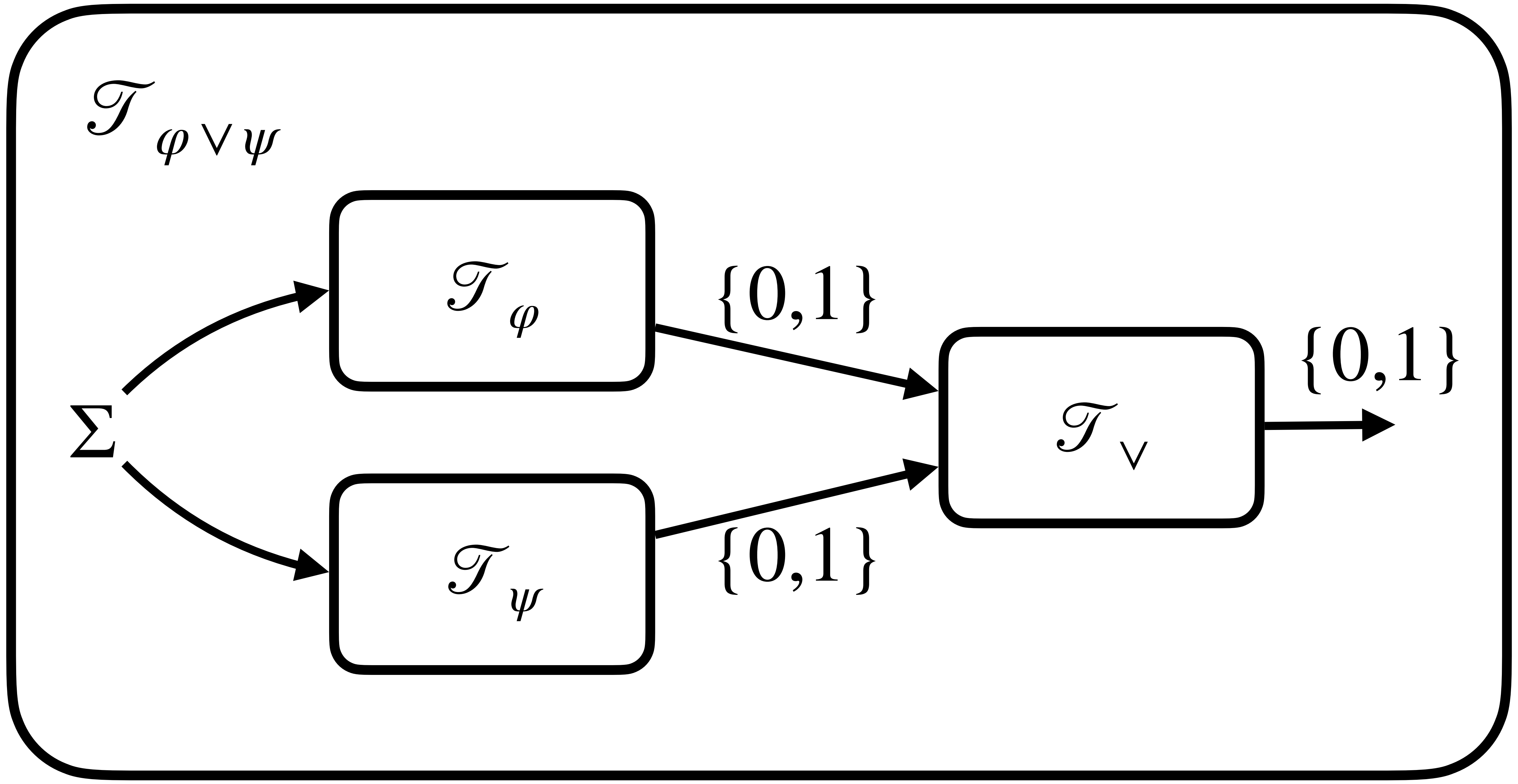


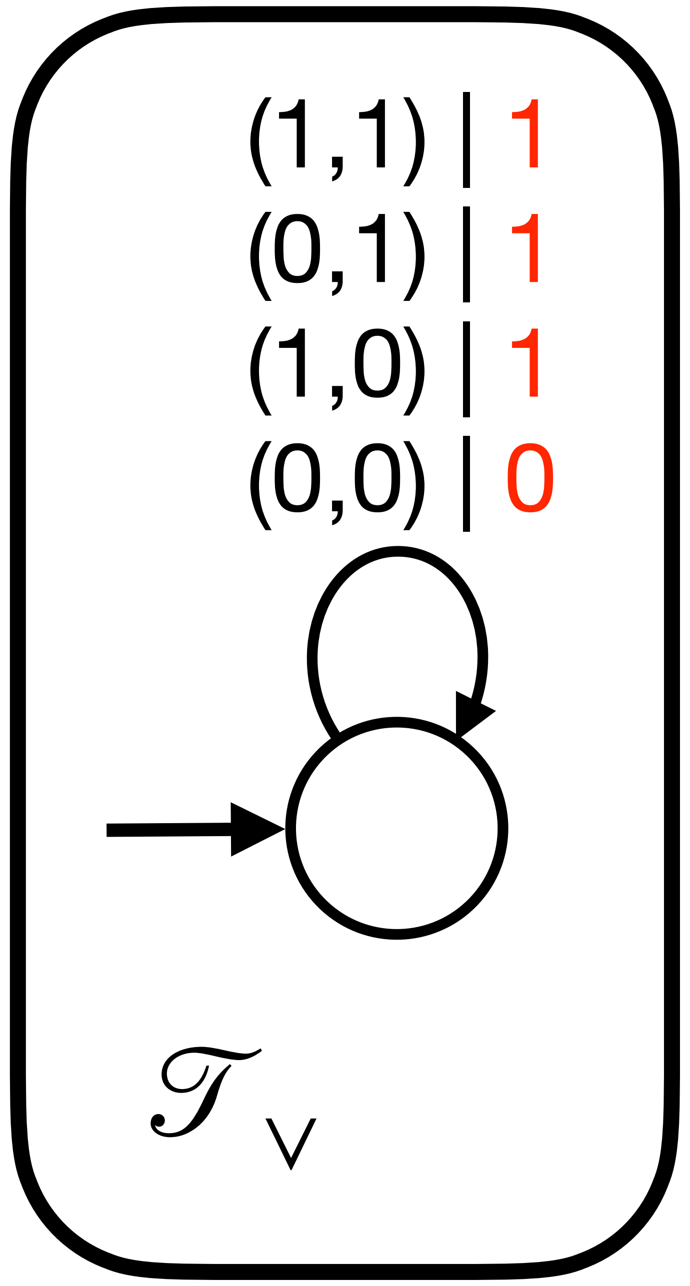


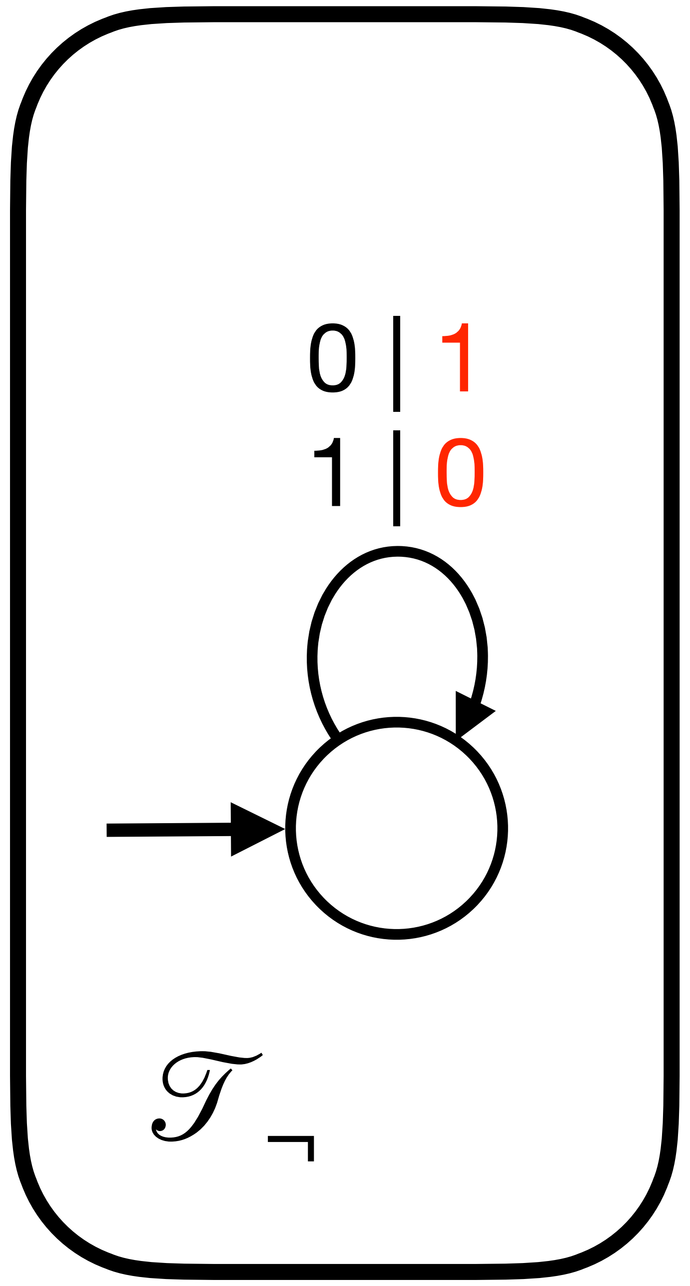
Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathscr{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

PastLTL: boolean connectives and strict-since

KR proof for aperiodic



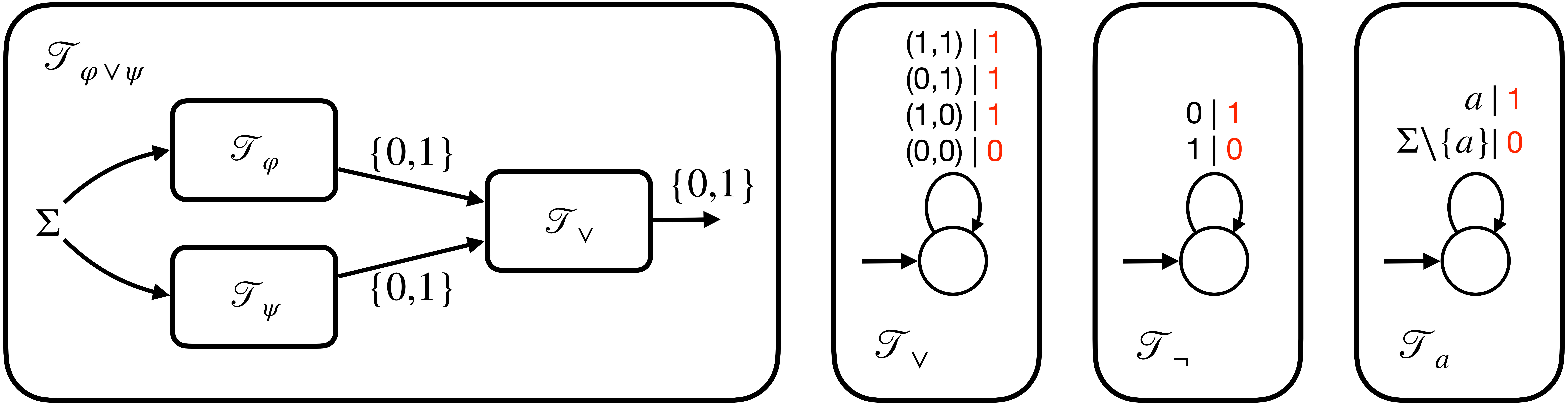




Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathscr{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

PastLTL: boolean connectives and strict-since

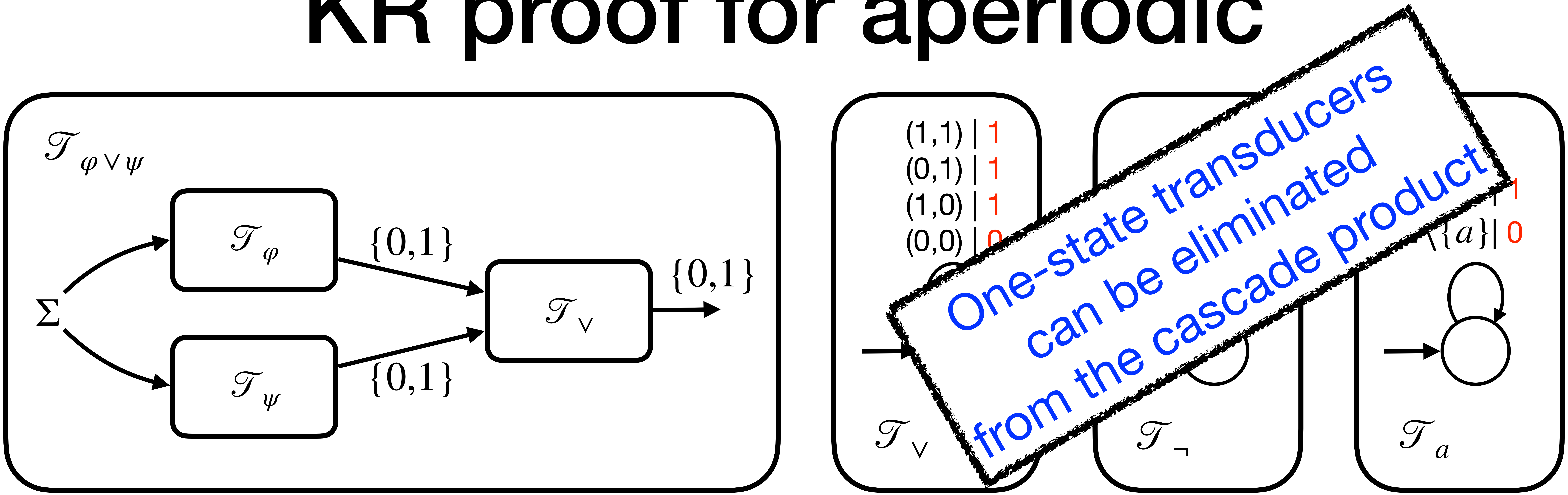
KR proof for aperiodic



Given a PastLTL formula φ, we will implement θ_φ with a transducer $\mathscr{T}_\varphi$ constructed inductively as a cascade product of reset transducers.

PastLTL: boolean connectives and strict-since

KR proof for aperiodic



Given a PastLTL formula φ, we will implement θ_{φ} with a transducer $\mathcal{T}_{\varphi}$ constructed inductively as a cascade product of reset transducers.

PastLTL: boolean connectives and strict-since

Outline

- ✅ Labelling functions, sequential transducers and cascade product
- ✅ Krohn-Rhodes theorem for aperiodic/regular word languages
- Model of concurrency: Mazurkiewicz traces and asynchronous Zielonka automata
- Asynchronous labelling functions, transducers and cascade product
- Propositional dynamic logic for traces
- Conclusion

Mazurkiewicz Traces

Architecture

$\mathscr{P} = \{1,2,3\}$

$\Sigma = \{a, b, c, d, e\}$

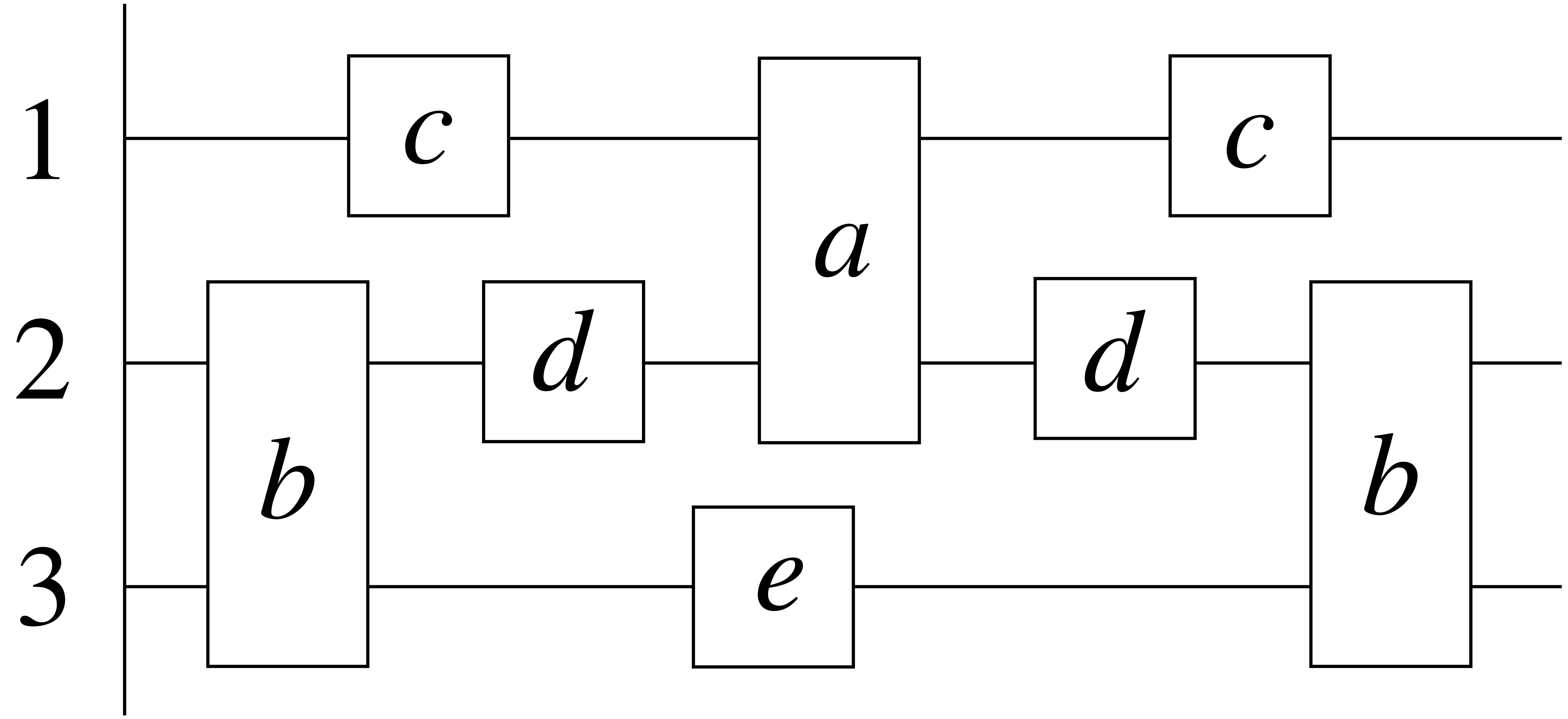
$\text{loc}(a) = \{1,2\}$

$\text{loc}(b) = \{2,3\}$

$\text{loc}(c) = \{1\}$

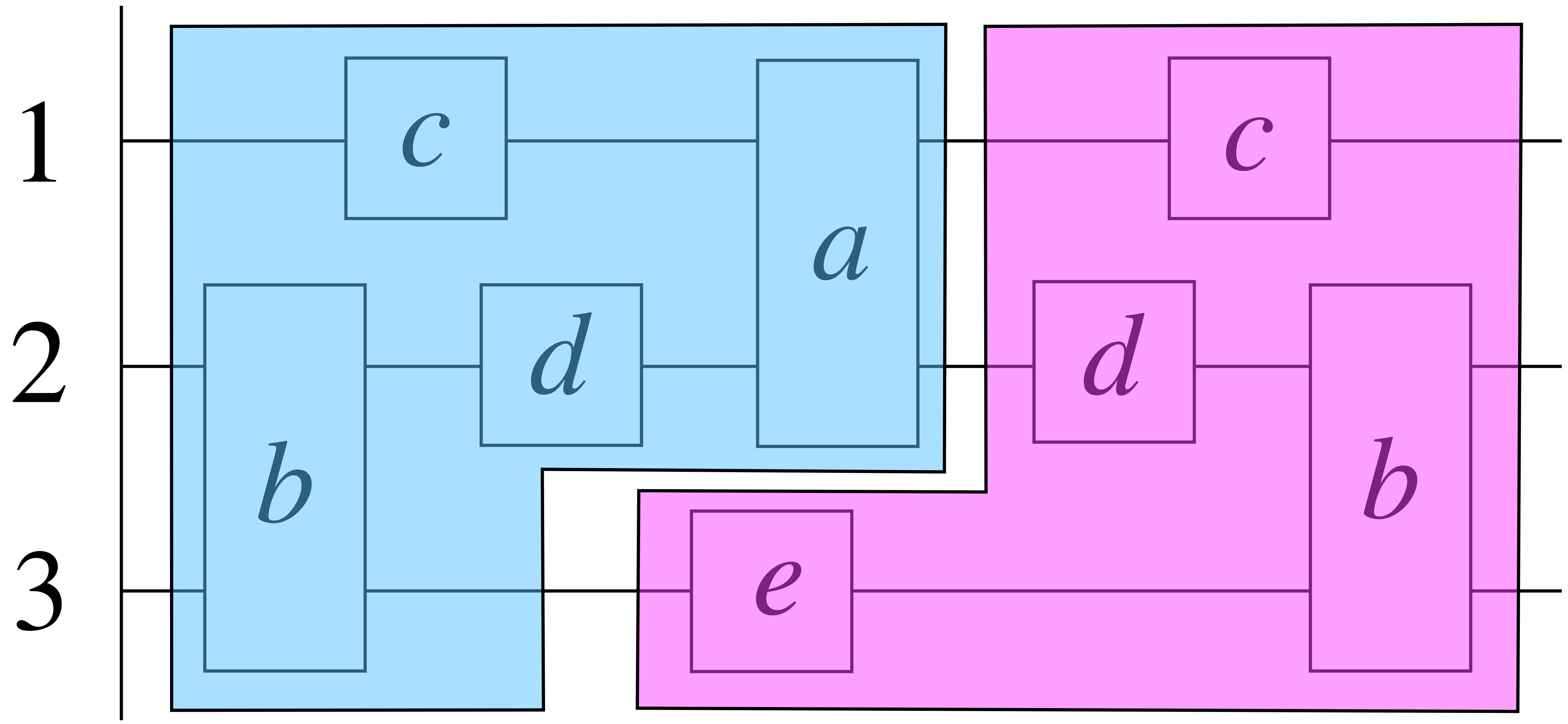
$\text{loc}(d) = \{2\}$

$\text{loc}(e) = \{3\}$

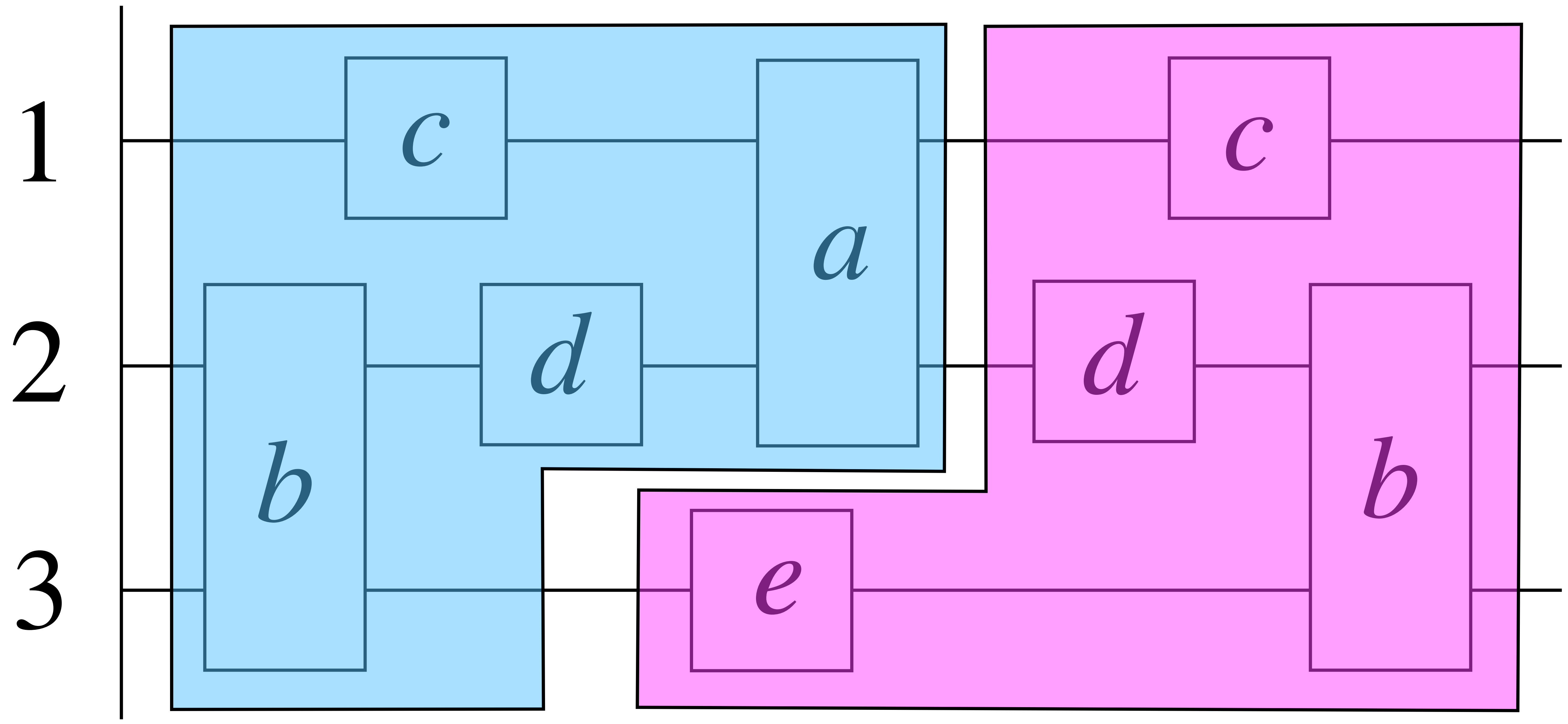


- *set of traces denoted* $Tr(\Sigma, \mathscr{P}, \text{loc})$ *or simply* $Tr(\Sigma)$
- *Trace Language:* $L \subseteq Tr(\Sigma)$

Concatenation, Independence, Commutation, Monoid

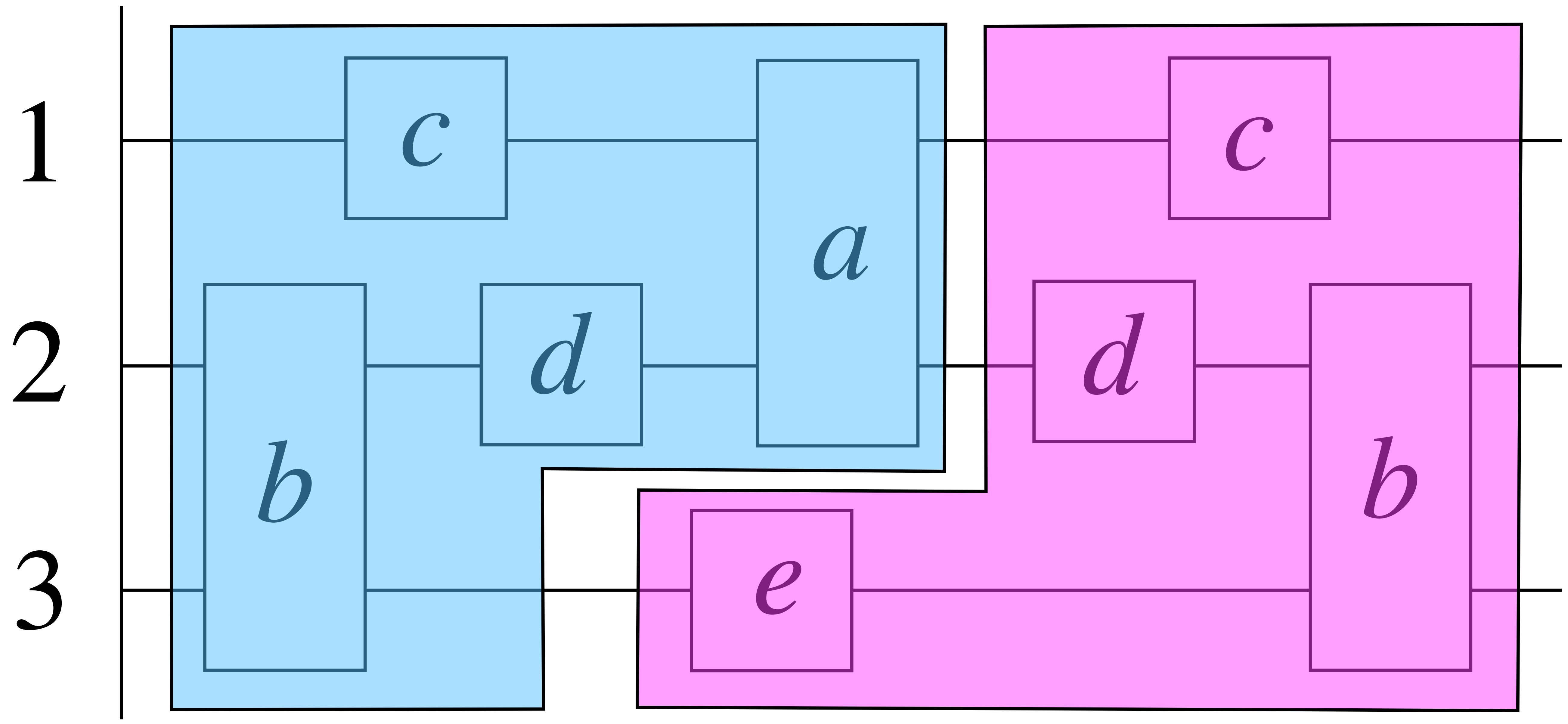


Concatenation, Independence, Commutation, Monoid



- $Tr(\Sigma)$ *with trace concatenation is a* *monoid*

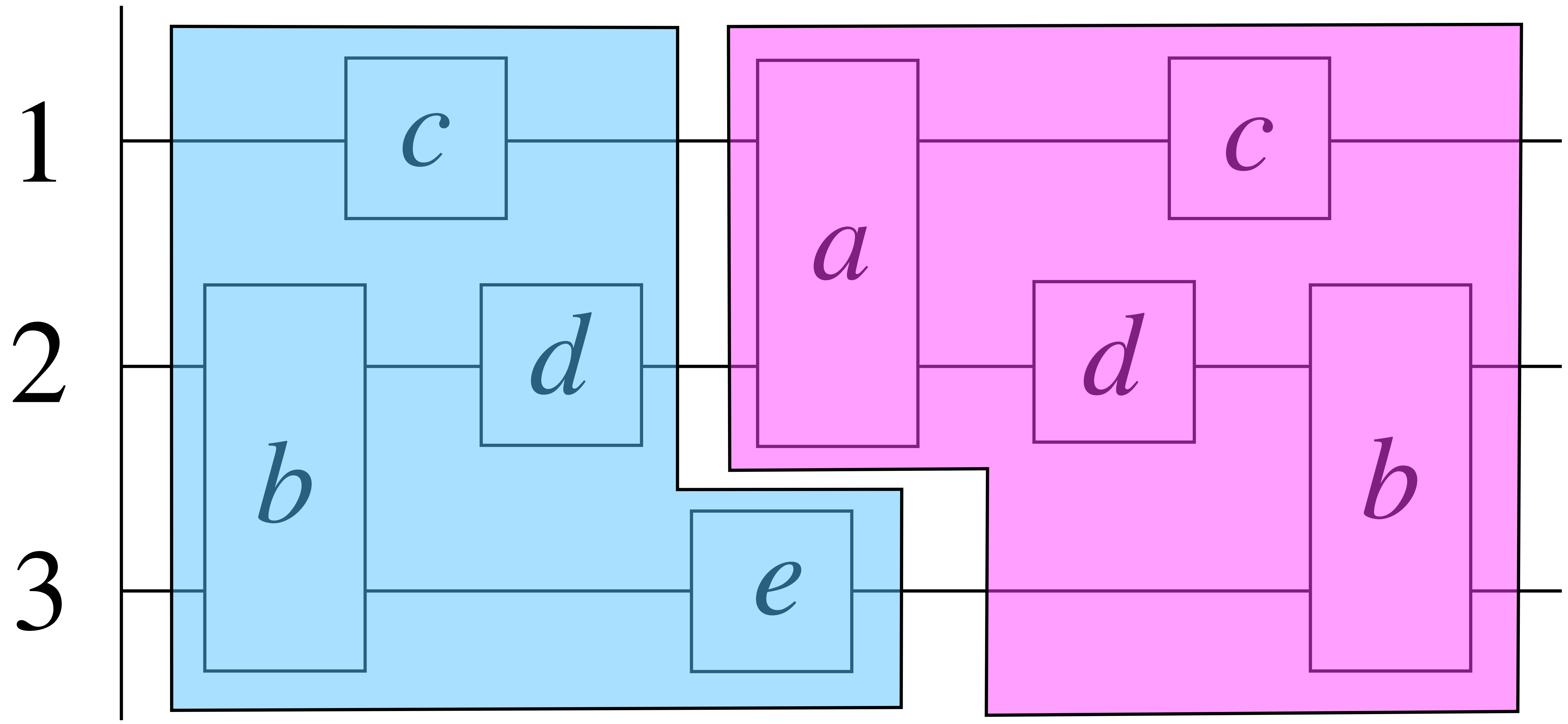
Concatenation, Independence, Commutation, Monoid



- $x \; I \; y$ iff $\mathrm{loc}(x) \cap \mathrm{loc}(y) = \varnothing$
- $a \; I \; e$
- $a \cdot e = e \cdot a$

- *$Tr(\Sigma)$ with trace concatenation is a monoid*
- *Free partially commutative monoid*

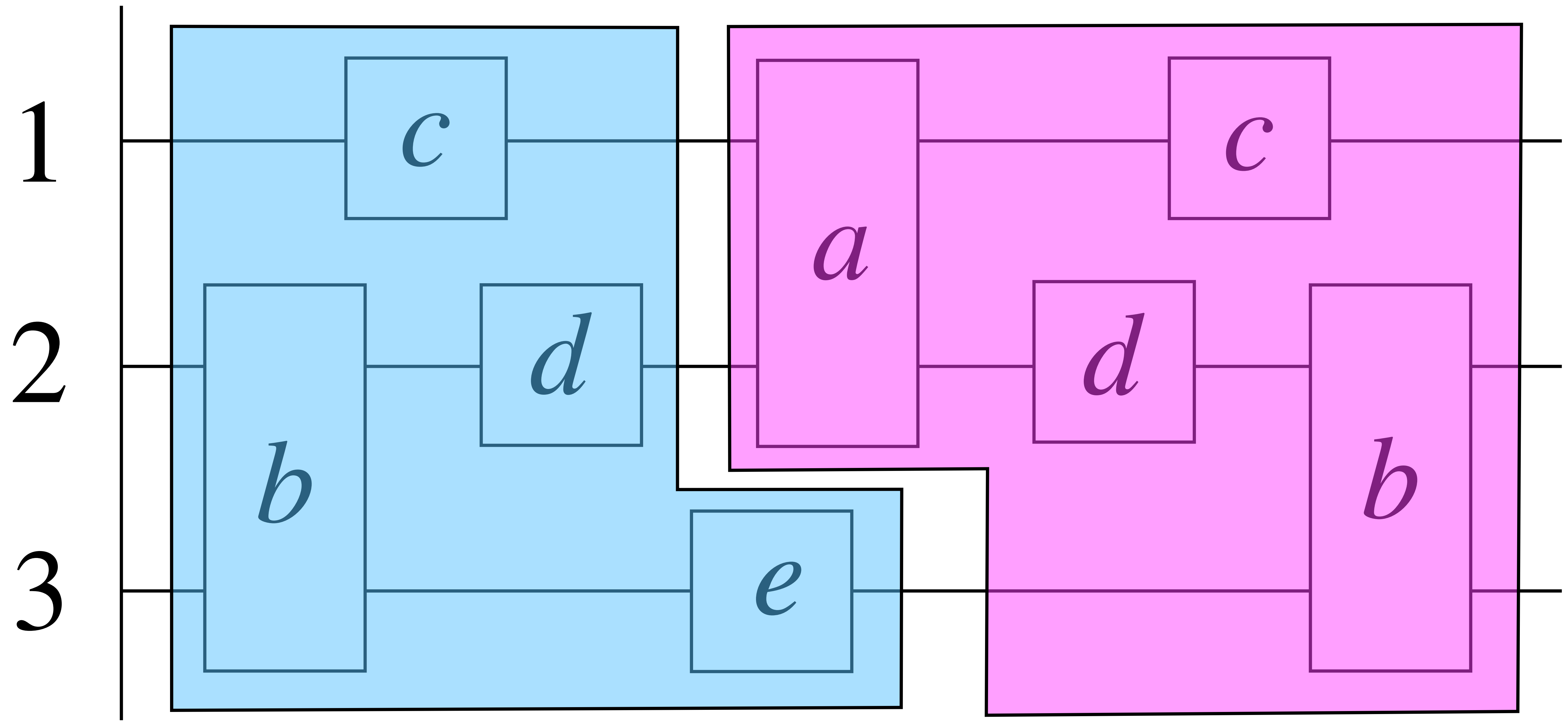
Concatenation, Independence, Commutation, Monoid



- $x \; I \; y$ iff $\mathrm{loc}(x) \cap \mathrm{loc}(y) = \emptyset$
- $a \; I \; e$
- $a \cdot e = e \cdot a$

- *$Tr(\Sigma)$ with trace concatenation is a monoid*
- *Free partially commutative monoid*

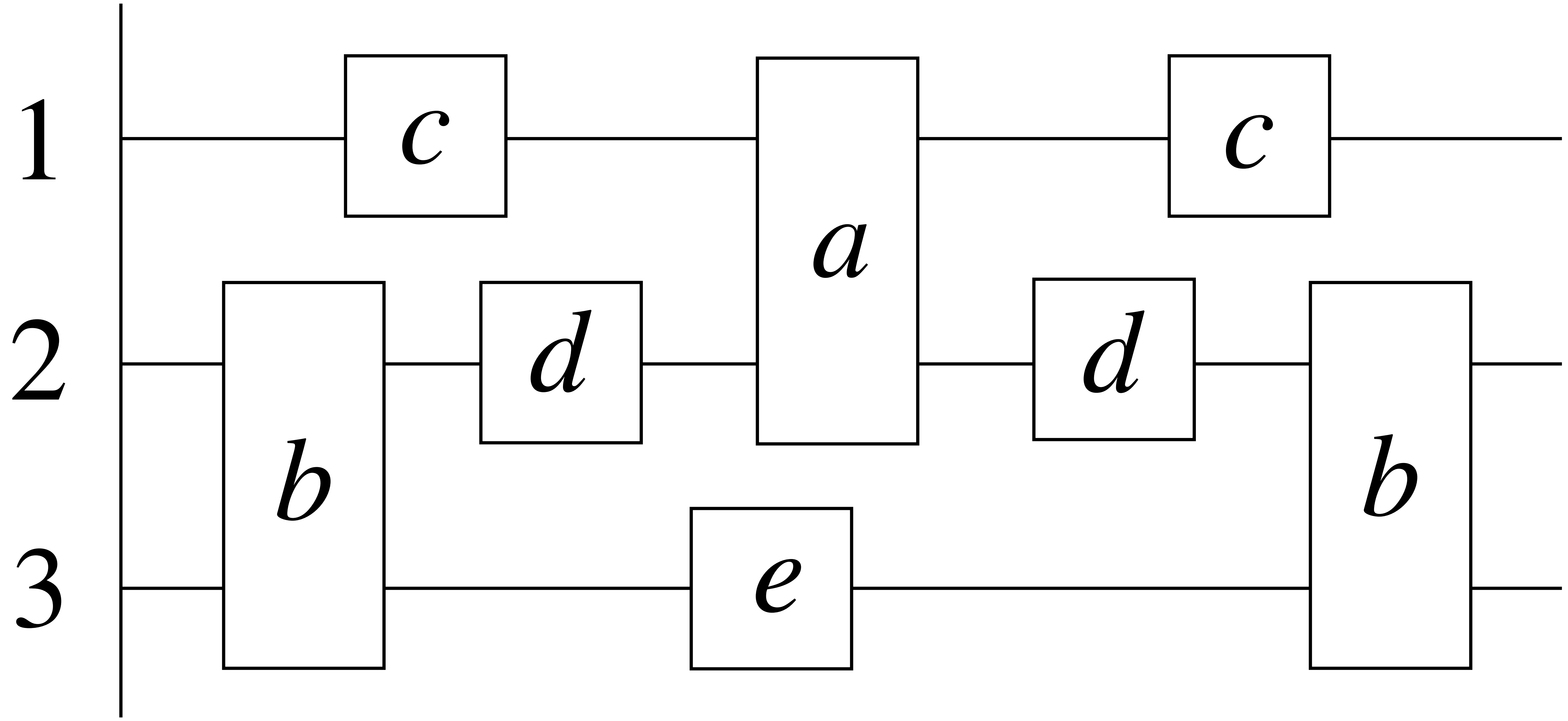
Concatenation, Independence, Commutation, Monoid



- $x\ I\ y$ iff $\mathrm{loc}(x) \cap \mathrm{loc}(y) = \varnothing$
- $a\ I\ e$
- $a \cdot e = e \cdot a$

- $Tr(\Sigma)$ *with trace concatenation is a monoid*
- *Free partially commutative monoid*
- $L \subseteq Tr(\Sigma)$ *is regular if* $L = \eta^{-1}(\eta(L))$ *for some morphism* $\eta : Tr(\Sigma) \to M$ *(finite monoid)*

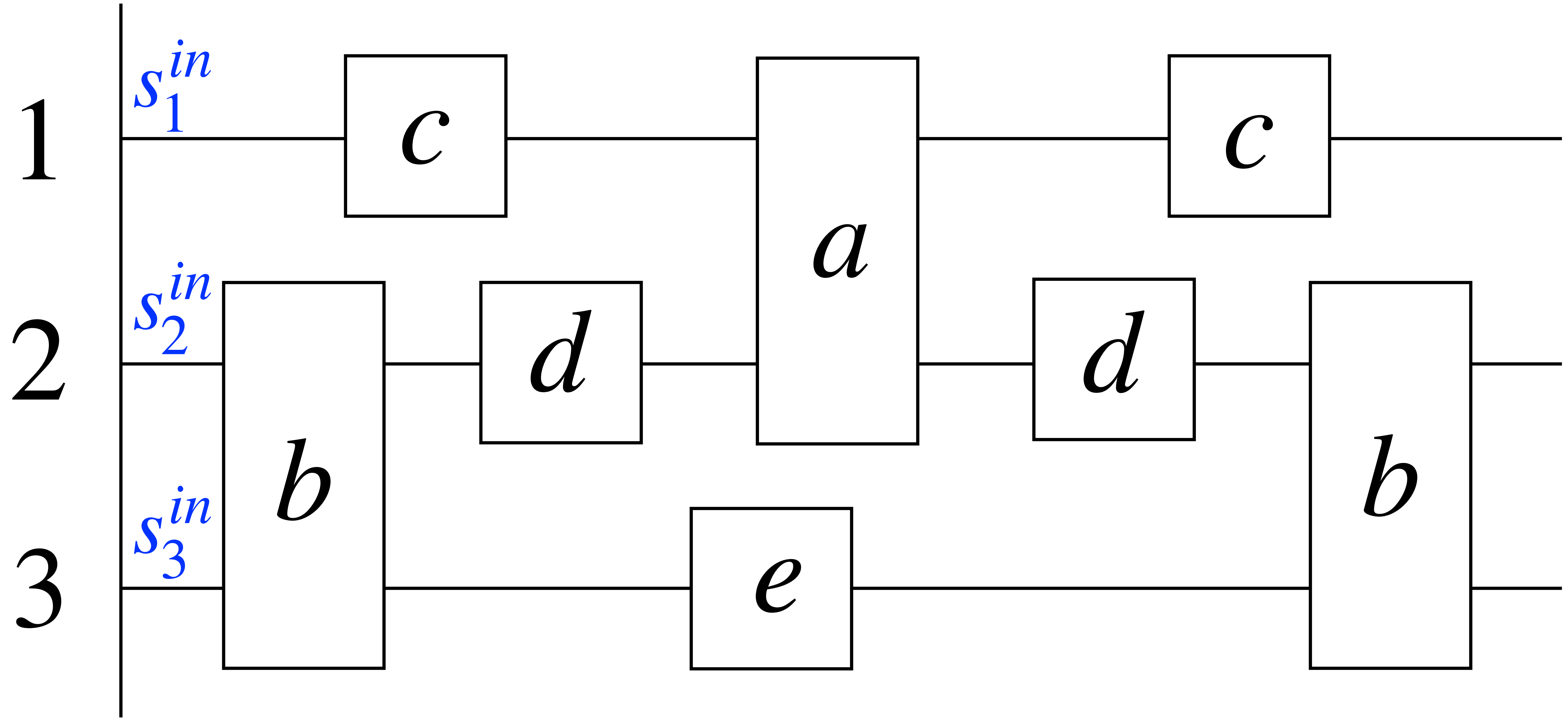
Asynchronous automata (Zielonka)



$\mathscr{A} = \left(\{S_i\}_{i\in\mathscr{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}, F\right)$

- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

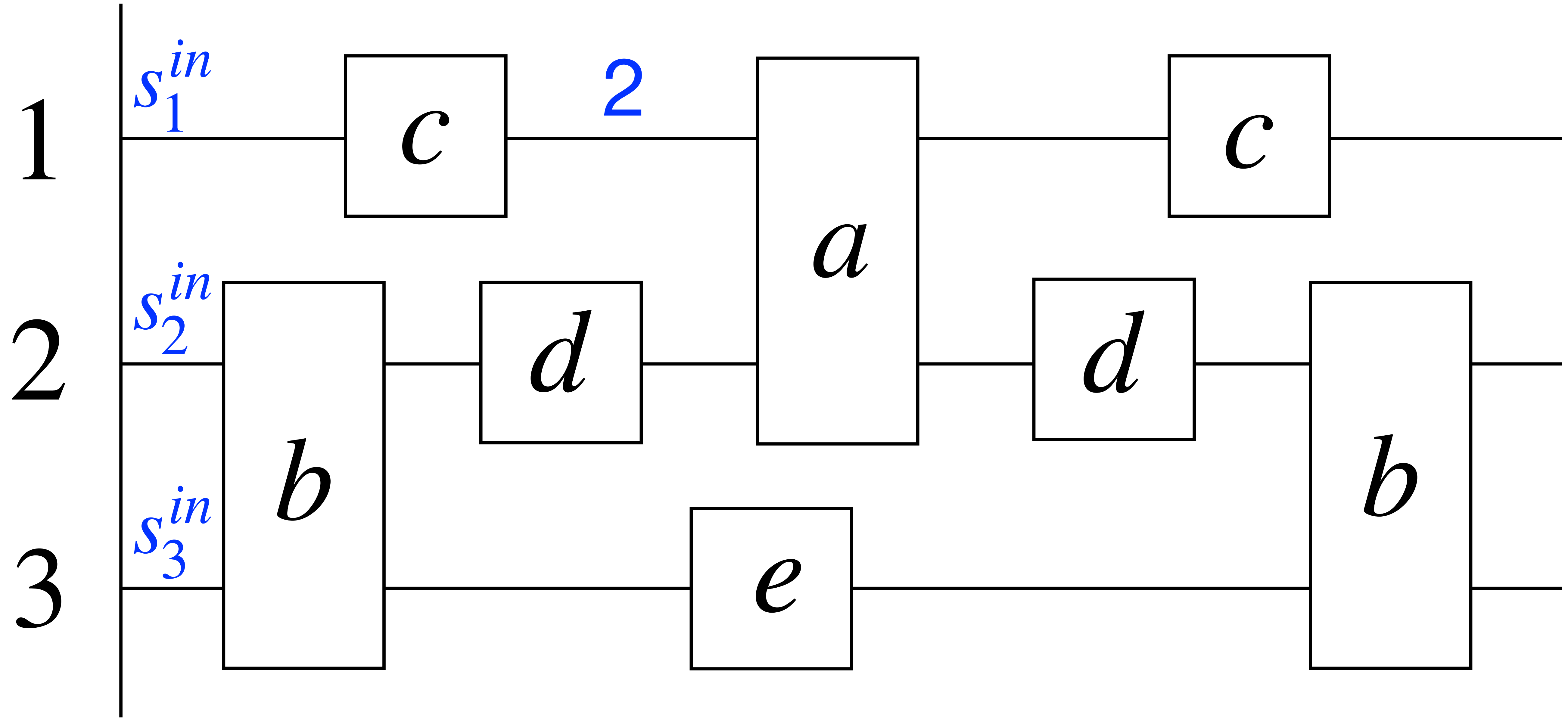
Asynchronous automata (Zielonka)



$$\mathscr{A} = \left(\{S_i\}_{i\in\mathscr{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}, F\right)$$

- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

Asynchronous automata (Zielonka)

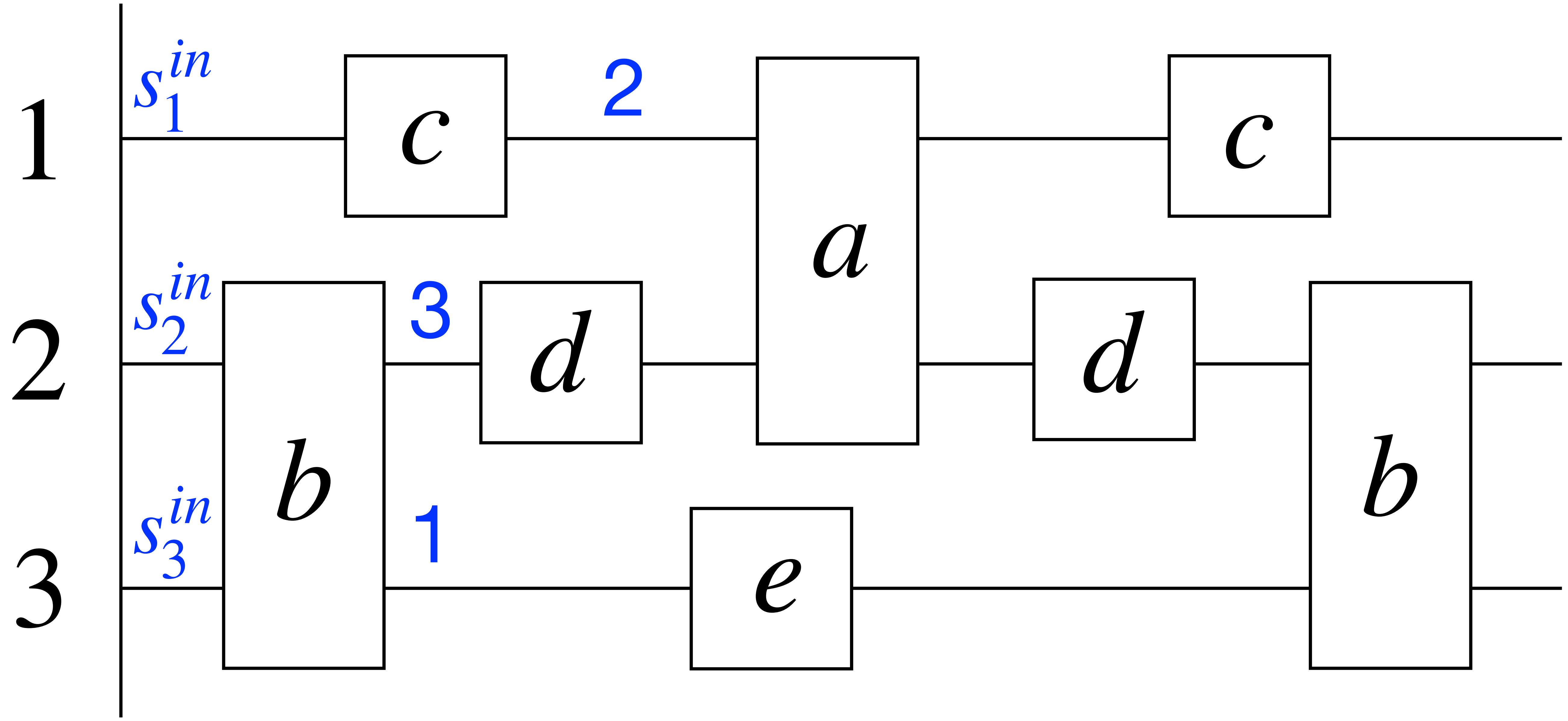


$$\mathscr{A} = \left(\{S_i\}_{i\in\mathscr{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}, F\right)$$

- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

$$\delta_c : S_1 \to S_1$$

Asynchronous automata (Zielonka)



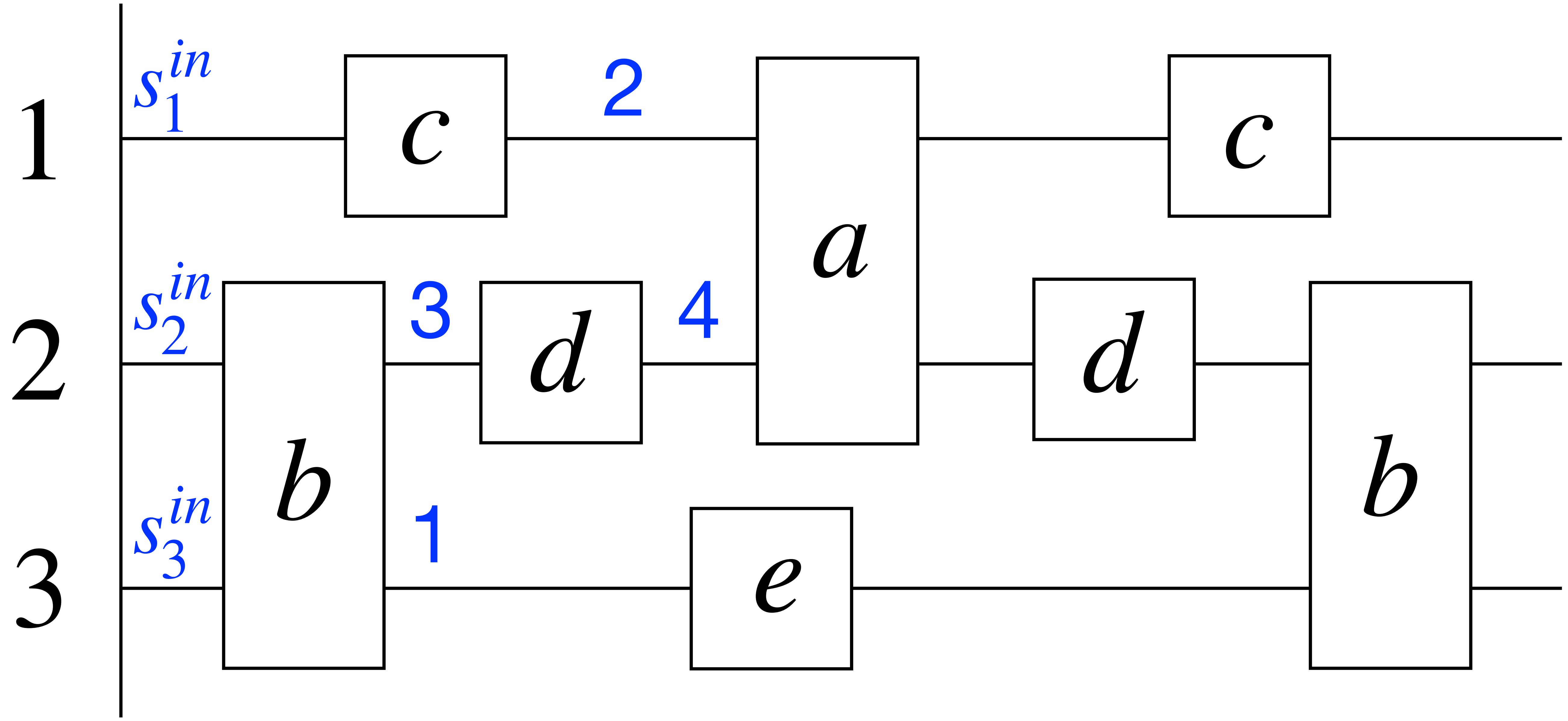
$\mathscr{A} = \left(\{S_i\}_{i \in \mathscr{P}}, \{\delta_a\}_{a \in \Sigma}, s^{in}, F\right)$

- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

$$\delta_c : S_1 \to S_1$$

$$\delta_b : S_2 \times S_3 \to S_2 \times S_3$$

Asynchronous automata (Zielonka)



$\mathscr{A} = \left(\{S_i\}_{i\in\mathscr{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}, F\right)$

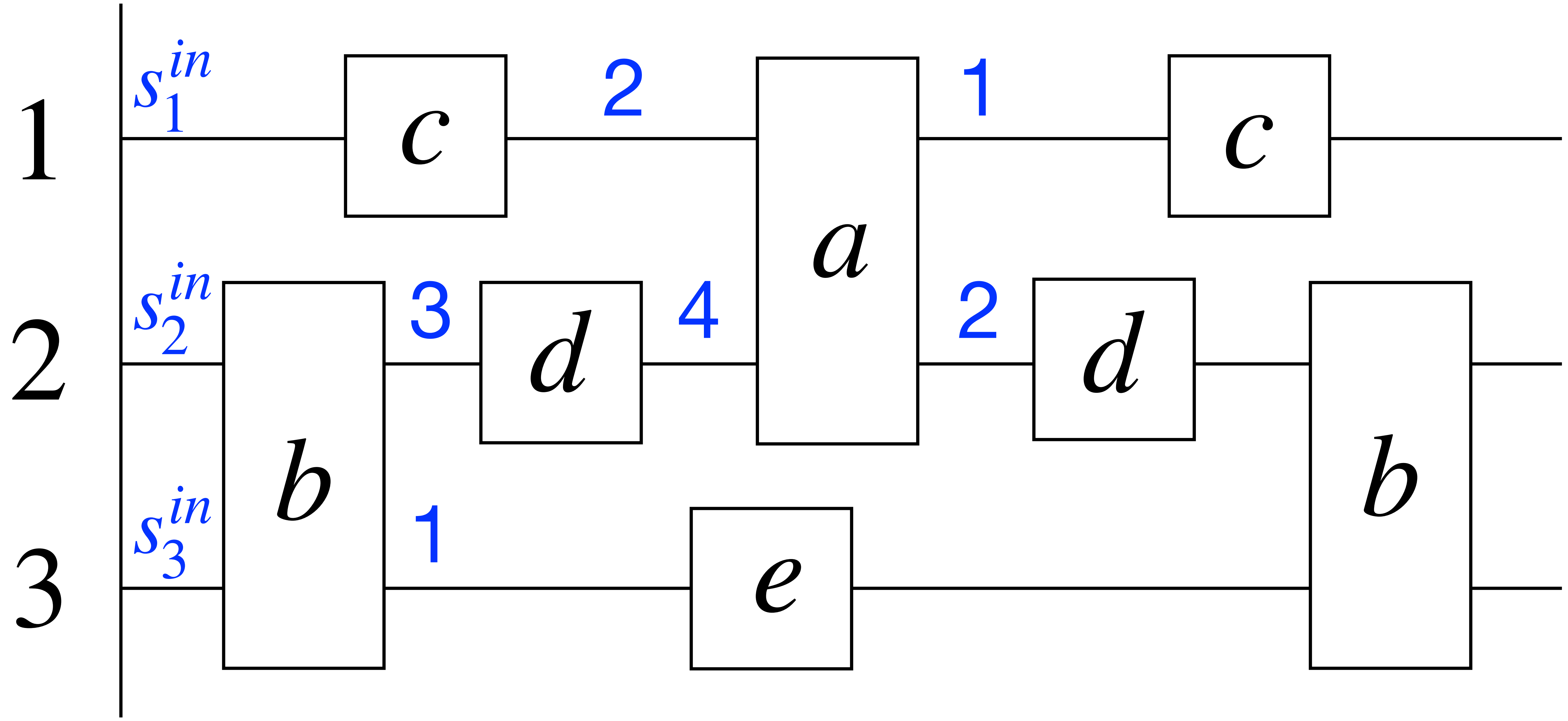
- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

$\delta_c : S_1 \to S_1$

$\delta_b : S_2 \times S_3 \to S_2 \times S_3$

$\delta_d : S_2 \to S_2$

Asynchronous automata (Zielonka)



$\mathscr{A} = \left(\{S_i\}_{i\in\mathscr{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}, F\right)$

- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

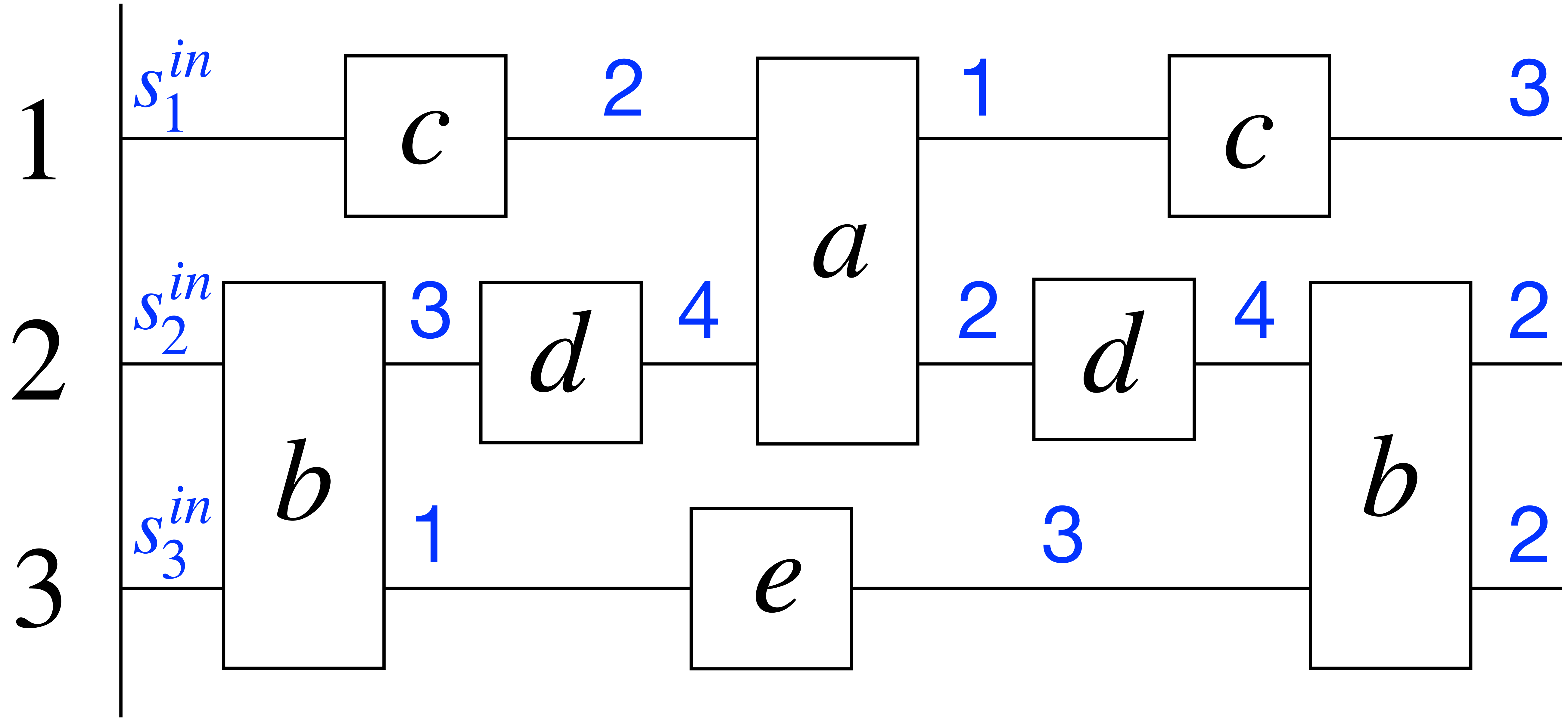
$$\delta_c : S_1 \to S_1$$

$$\delta_b : S_2 \times S_3 \to S_2 \times S_3$$

$$\delta_d : S_2 \to S_2$$

$$\delta_a : S_1 \times S_2 \to S_1 \times S_2$$

Asynchronous automata (Zielonka)

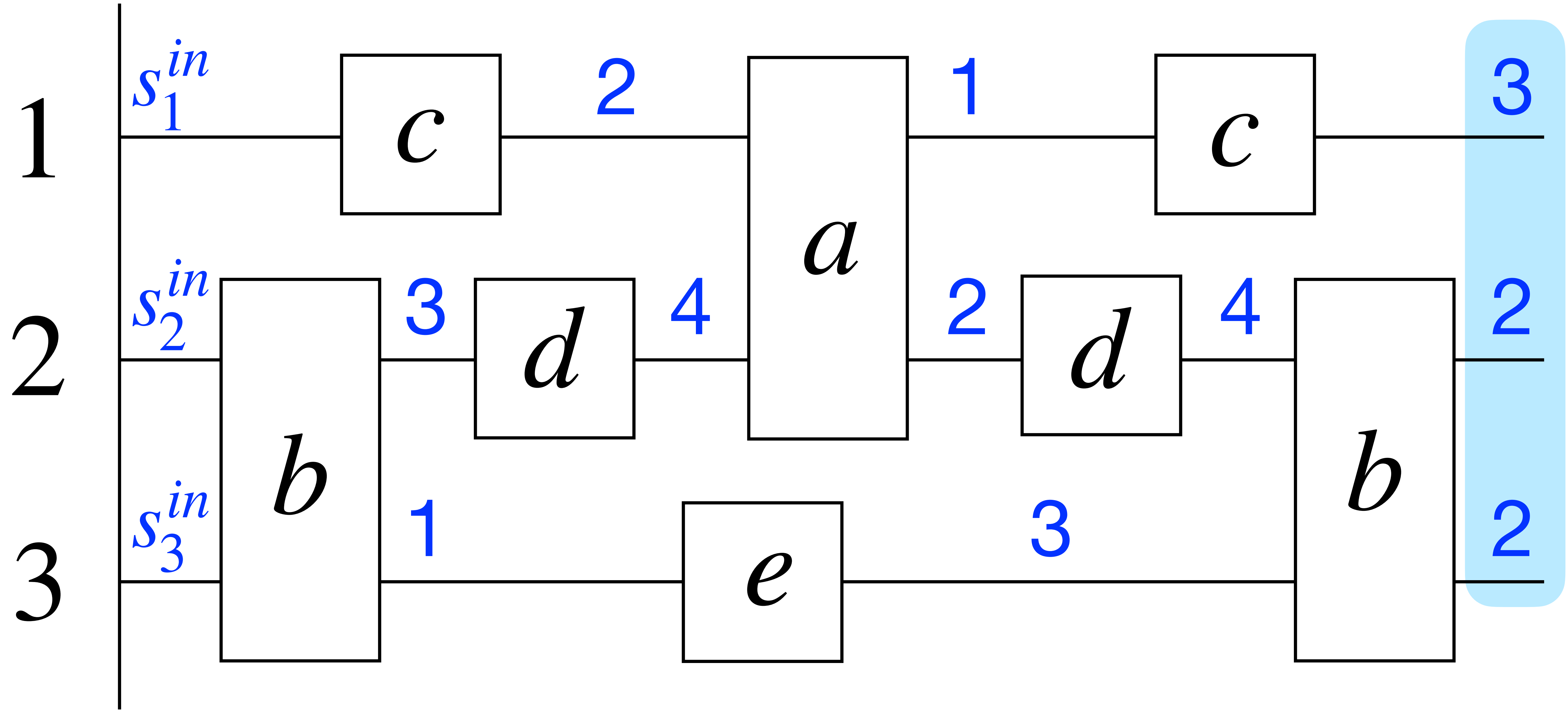


$\mathscr{A} = \left(\{S_i\}_{i \in \mathscr{P}}, \{\delta_a\}_{a \in \Sigma}, s^{in}, F\right)$

- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

$$\delta_c : S_1 \to S_1$$
$$\delta_b : S_2 \times S_3 \to S_2 \times S_3$$
$$\delta_d : S_2 \to S_2$$
$$\delta_a : S_1 \times S_2 \to S_1 \times S_2$$
$$\delta_e : S_3 \to S_3$$

Asynchronous automata (Zielonka)



$$\mathscr{A} = \left(\{S_i\}_{i\in\mathscr{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}, F\right)$$

- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

$$\delta_c : S_1 \to S_1$$

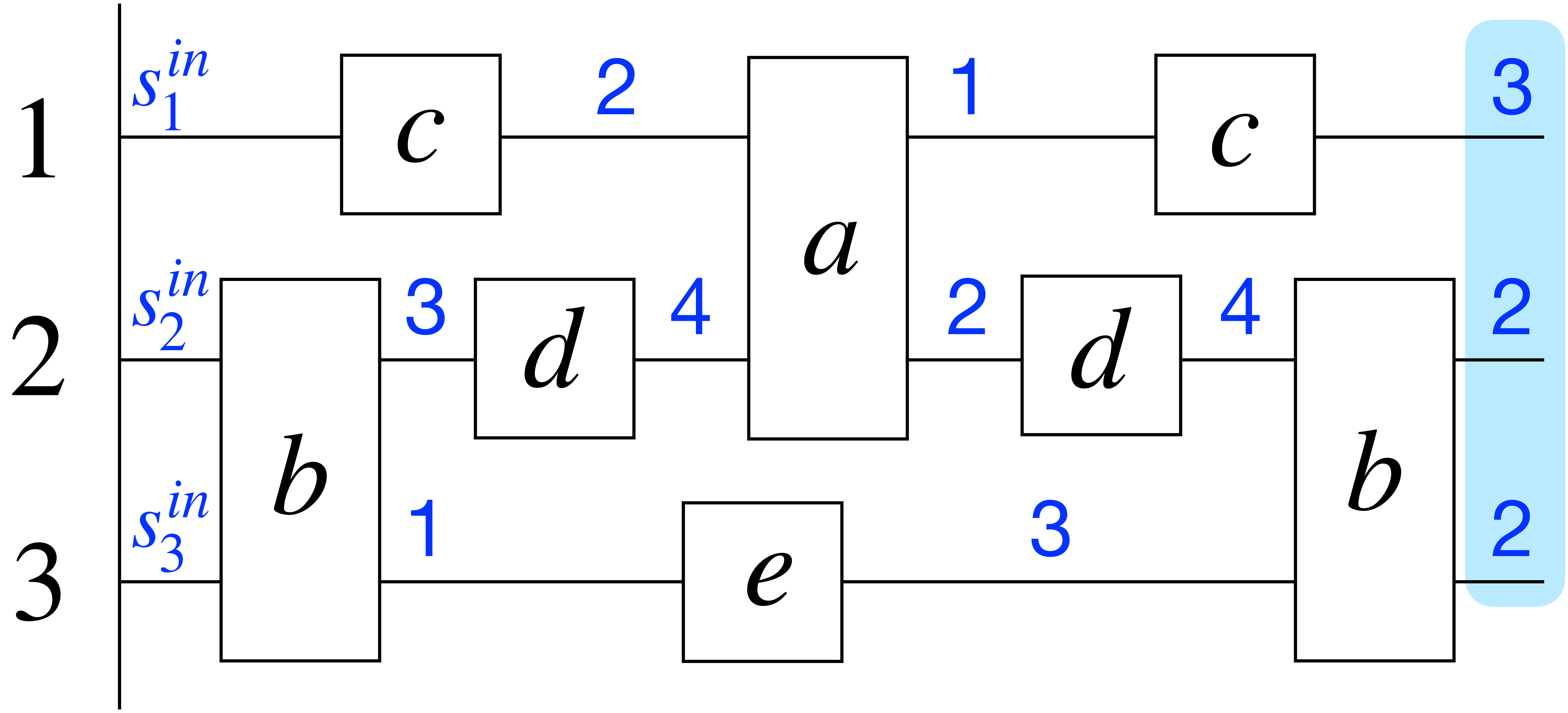
$$\delta_b : S_2 \times S_3 \to S_2 \times S_3$$

$$\delta_d : S_2 \to S_2$$

$$\delta_a : S_1 \times S_2 \to S_1 \times S_2$$

$$\delta_e : S_3 \to S_3$$

Asynchronous automata (Zielonka)



$\mathscr{A} = \left(\{S_i\}_{i\in\mathscr{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}, F\right)$

- S_i local states for process i
- δ_a transition function for action a
- $s^{in} = (s_1^{in}, s_2^{in}, s_3^{in})$ global initial state
- F global accepting states

$\delta_c : S_1 \to S_1$

$\delta_b : S_2 \times S_3 \to S_2 \times S_3$

$\delta_d : S_2 \to S_2$

$\delta_a : S_1 \times S_2 \to S_1 \times S_2$

$\delta_e : S_3 \to S_3$

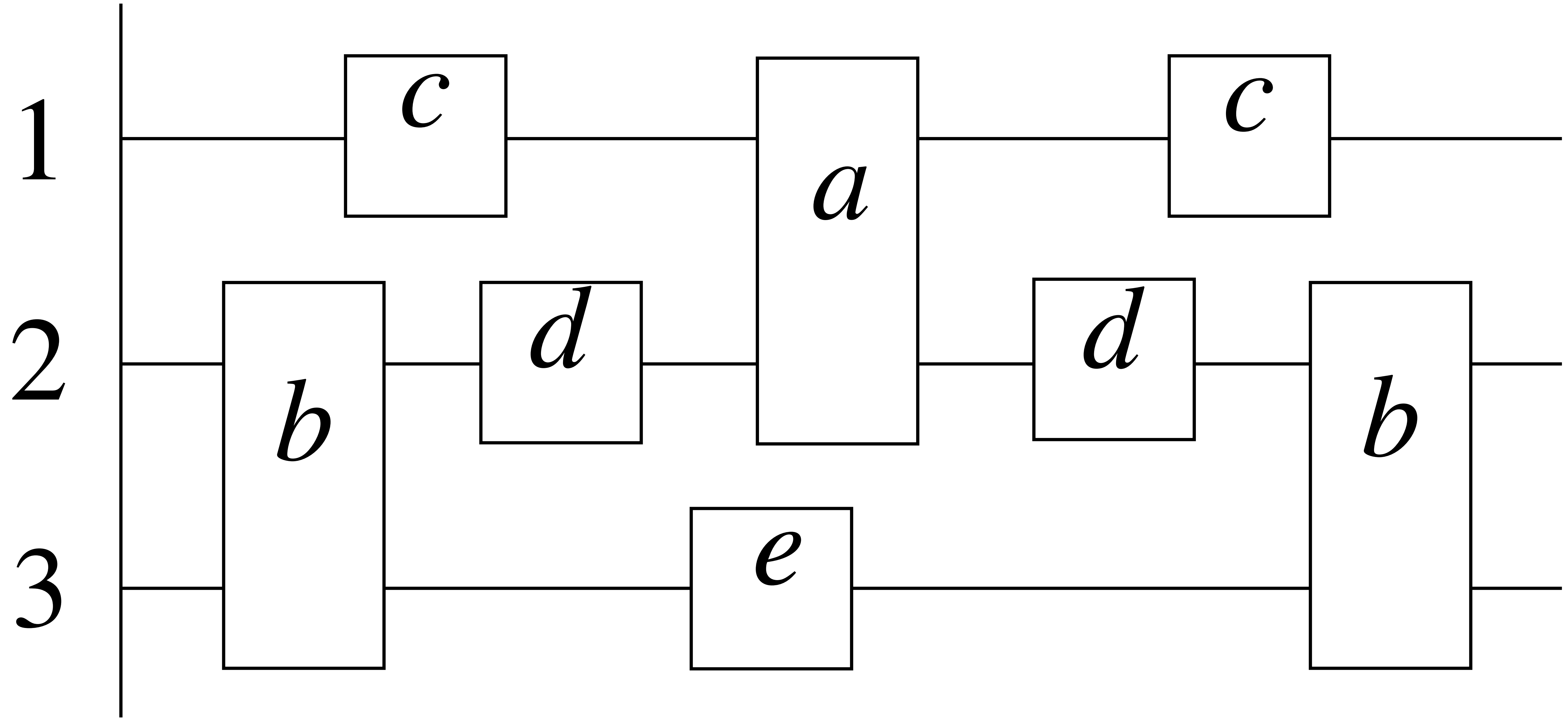
Theorem (Zielonka, 1987)

Asynchronous Automata = Regular Trace Languages

Outline

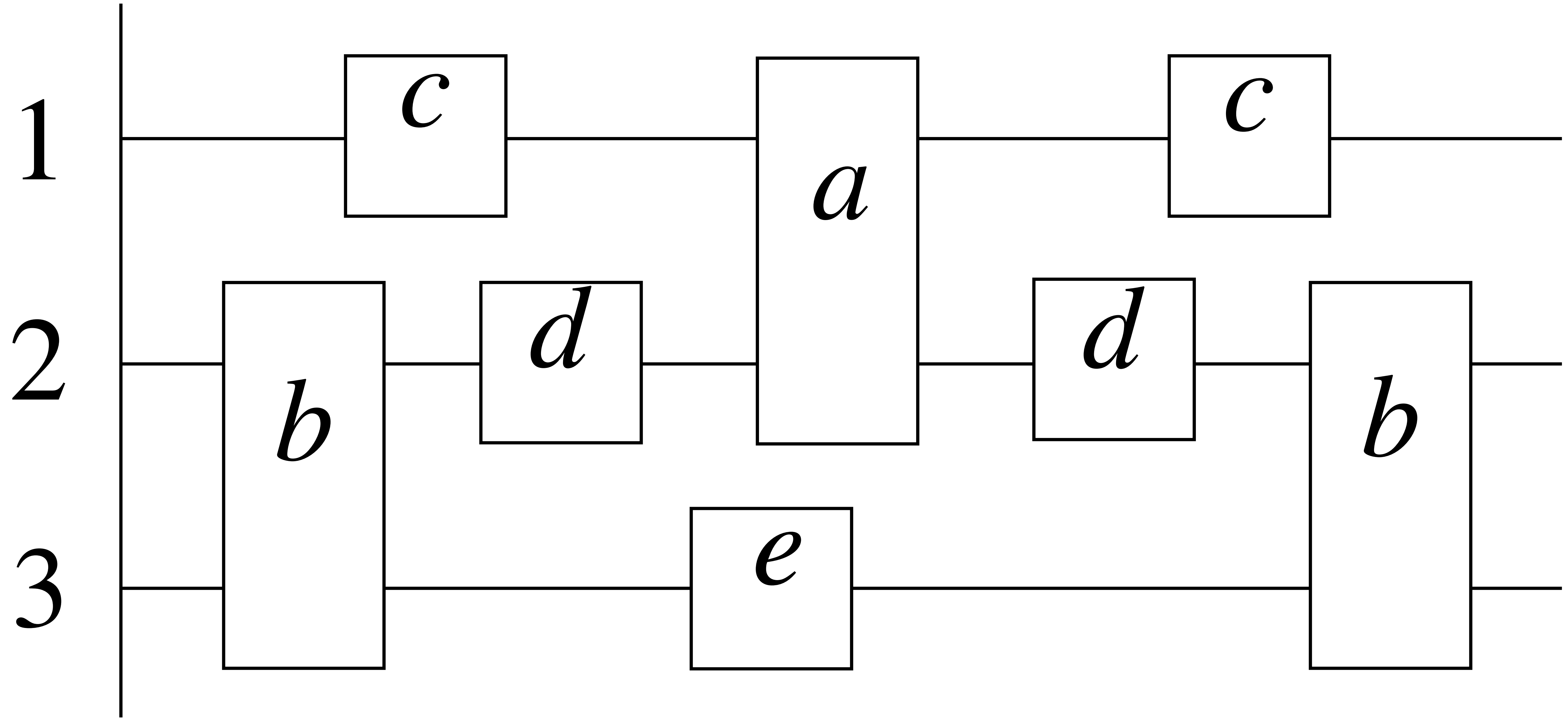
- ✓ Labelling functions, sequential transducers and cascade product
- ✓ Krohn-Rhodes theorem for aperiodic/regular word languages
- ✓ Model of concurrency: Mazurkiewicz traces and asynchronous Zielonka automata
- Asynchronous labelling functions, transducers and cascade product
- Propositional dynamic logic for traces
- Conclusion

Labelling function



$\theta : Tr(\Sigma) \rightarrow Tr(\Sigma \times \Gamma)$

Labelling function



$\theta : Tr(\Sigma) \to Tr(\Sigma \times \Gamma)$

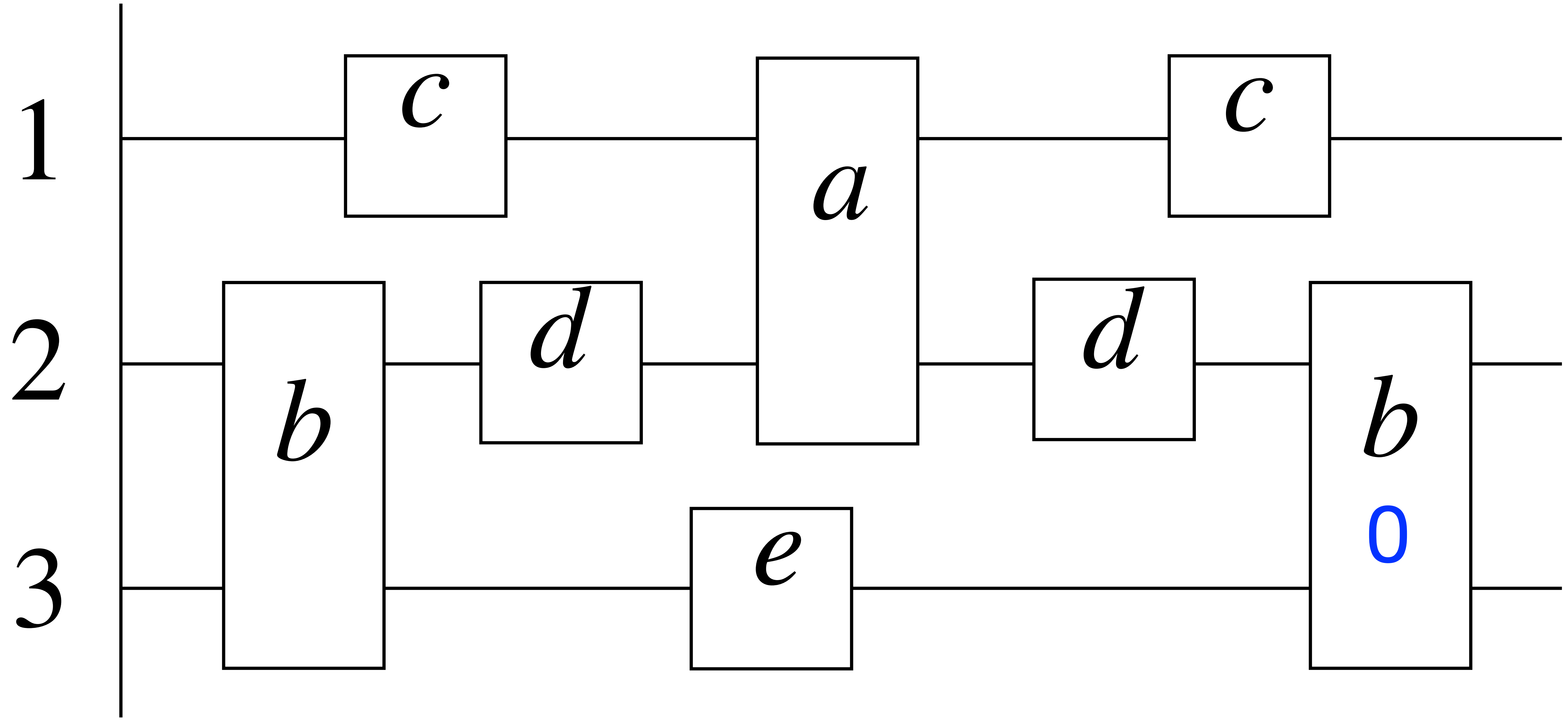
$\Gamma = \{0,1\}$

$Y_3 \leq Y_2$: In the strict past,

the last event on process 3 is below

the last event on process 2

Labelling function



$$\theta : Tr(\Sigma) \rightarrow Tr(\Sigma \times \Gamma)$$

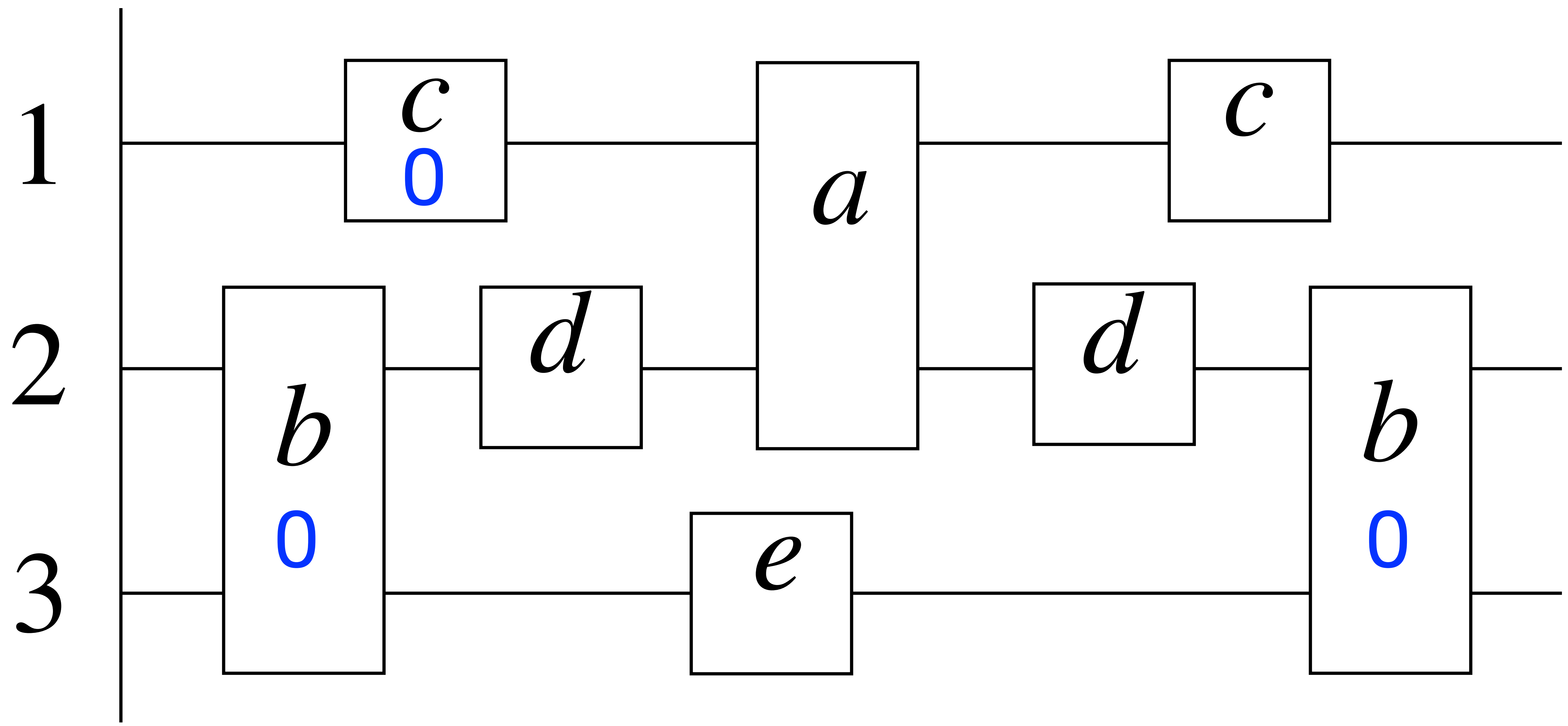
$\Gamma = \{0,1\}$

$Y_3 \leq Y_2$: In the strict past,

the last event on process 3 is below

the last event on process 2

Labelling function

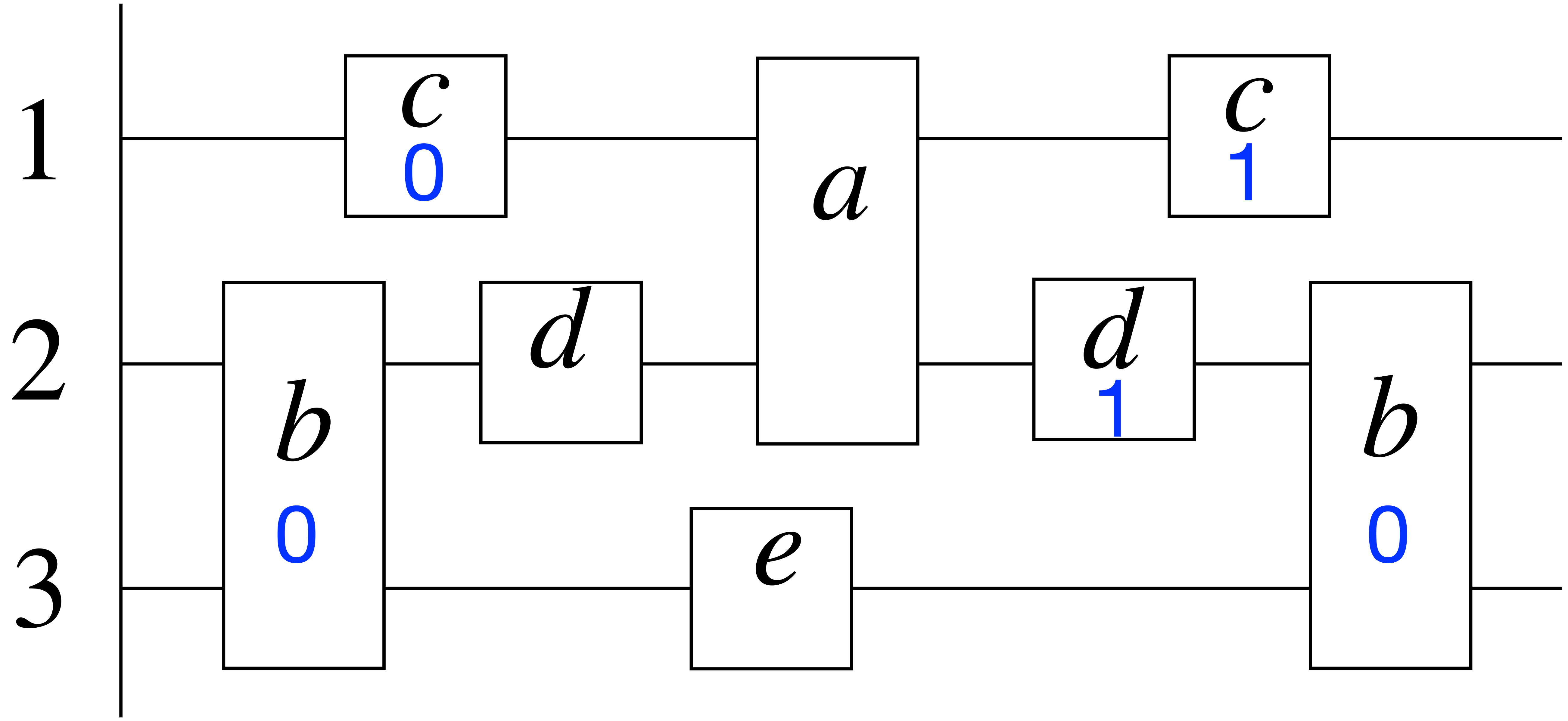


$\theta : Tr(\Sigma) \to Tr(\Sigma \times \Gamma)$

$\Gamma = \{0,1\}$

$Y_3 \leq Y_2$: In the strict past, the last event on process 3 is below the last event on process 2

Labelling function

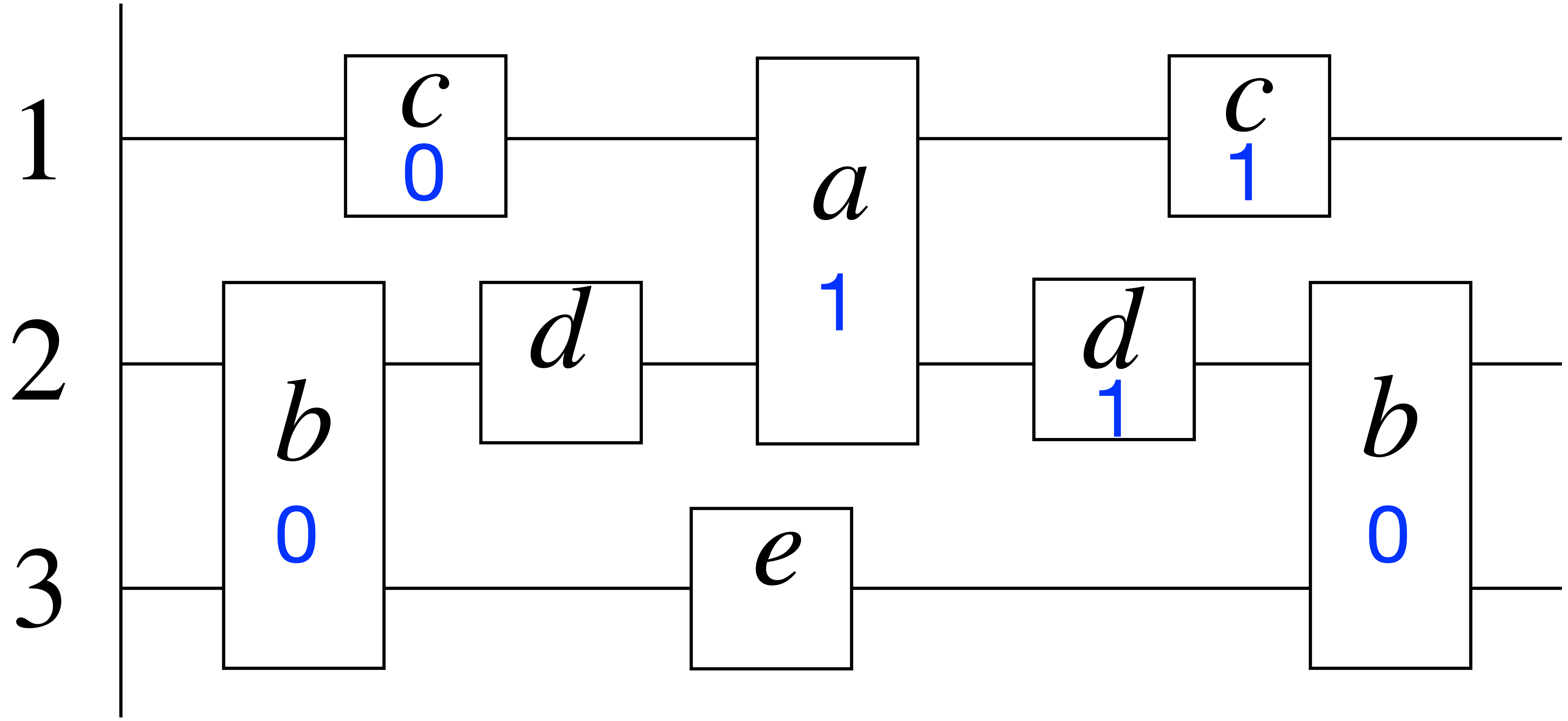


$\theta: Tr(\Sigma) \to Tr(\Sigma \times \Gamma)$

$\Gamma = \{0,1\}$

$Y_3 \leq Y_2$: In the strict past, the last event on process 3 is below the last event on process 2

Labelling function



$\theta: Tr(\Sigma) \to Tr(\Sigma \times \Gamma)$

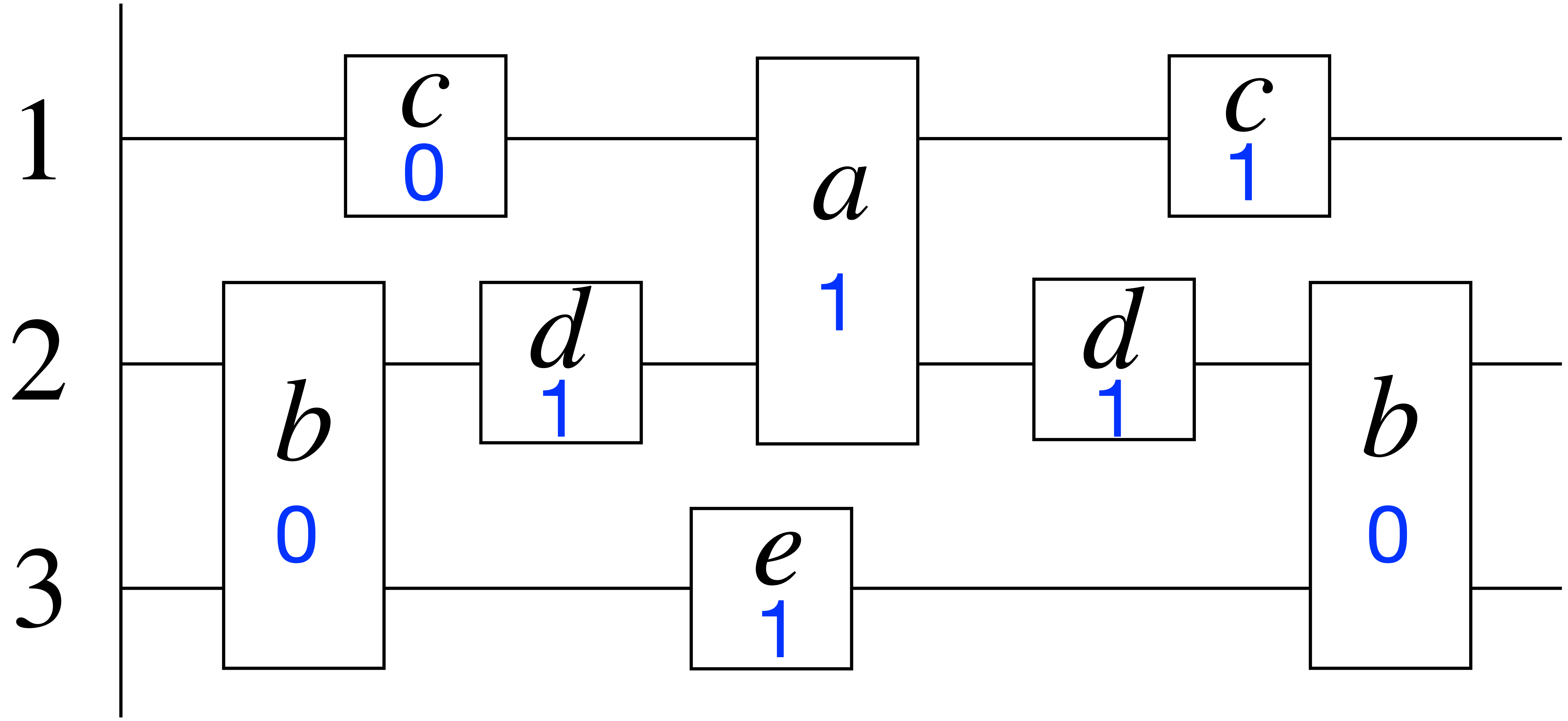
$\Gamma = \{0,1\}$

$Y_3 \leq Y_2$: In the strict past,

the last event on process 3 is below

the last event on process 2

Labelling function



$\theta : Tr(\Sigma) \rightarrow Tr(\Sigma \times \Gamma)$

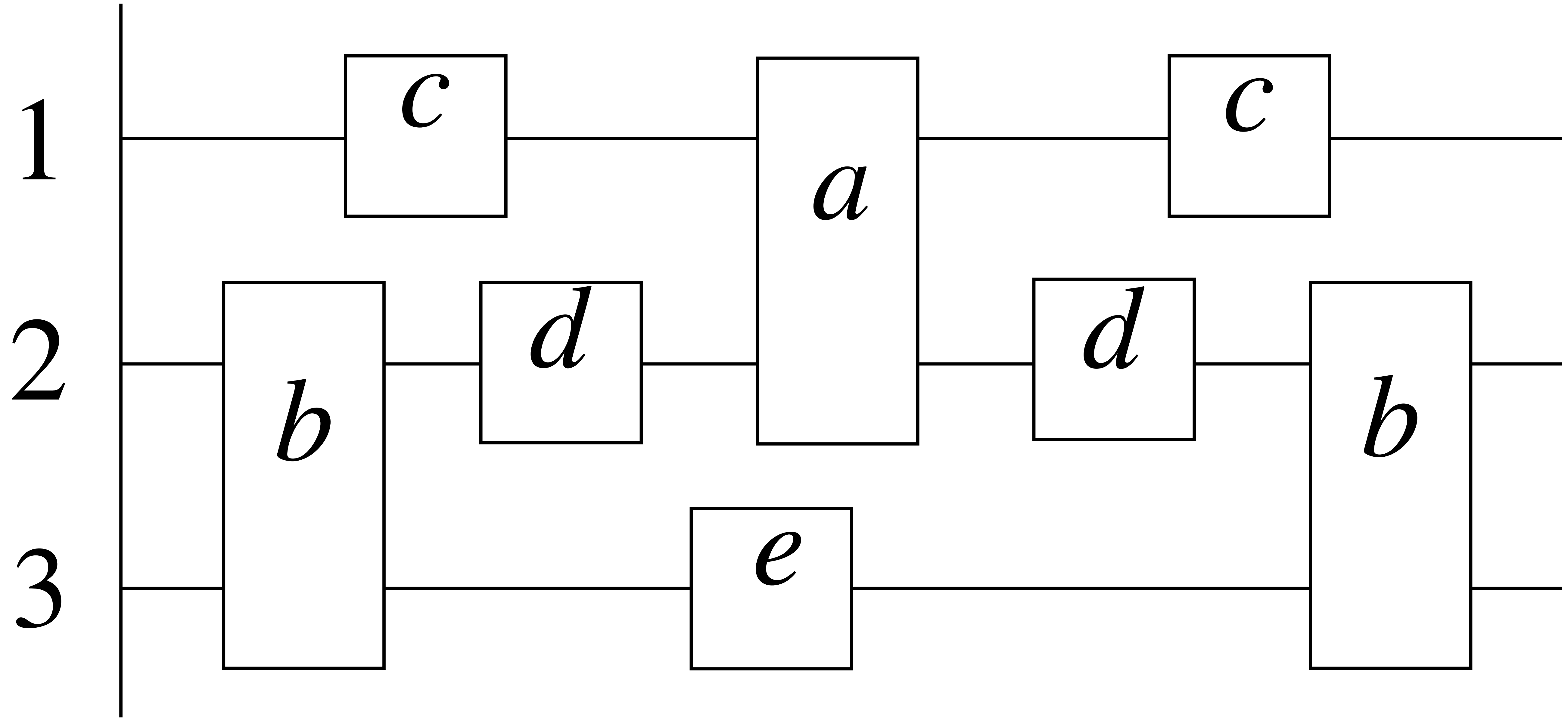
$\Gamma = \{0,1\}$

$Y_3 \leq Y_2$: In the strict past,

the last event on process 3 is below

the last event on process 2

Asynchronous Labelling function



$\theta : Tr(\Sigma) \rightarrow Tr(\Sigma \times \Gamma)$

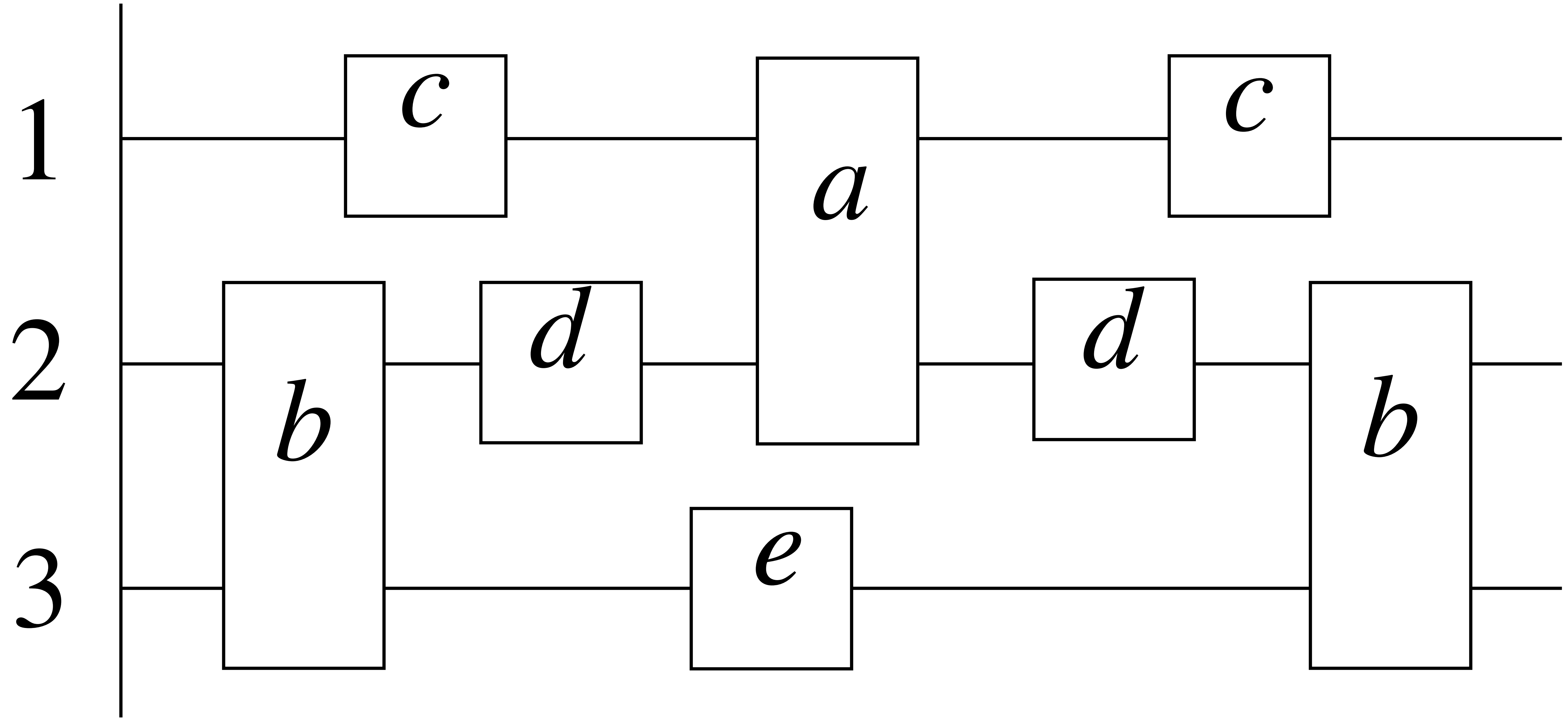
$\Gamma = \{0,1\}$ and $Y_3 \leq Y_2$

In the strict past,

the last event on process 3 is below

the last event on process 2

Asynchronous Labelling function



$\theta : Tr(\Sigma) \rightarrow Tr(\Sigma \times \Gamma)$

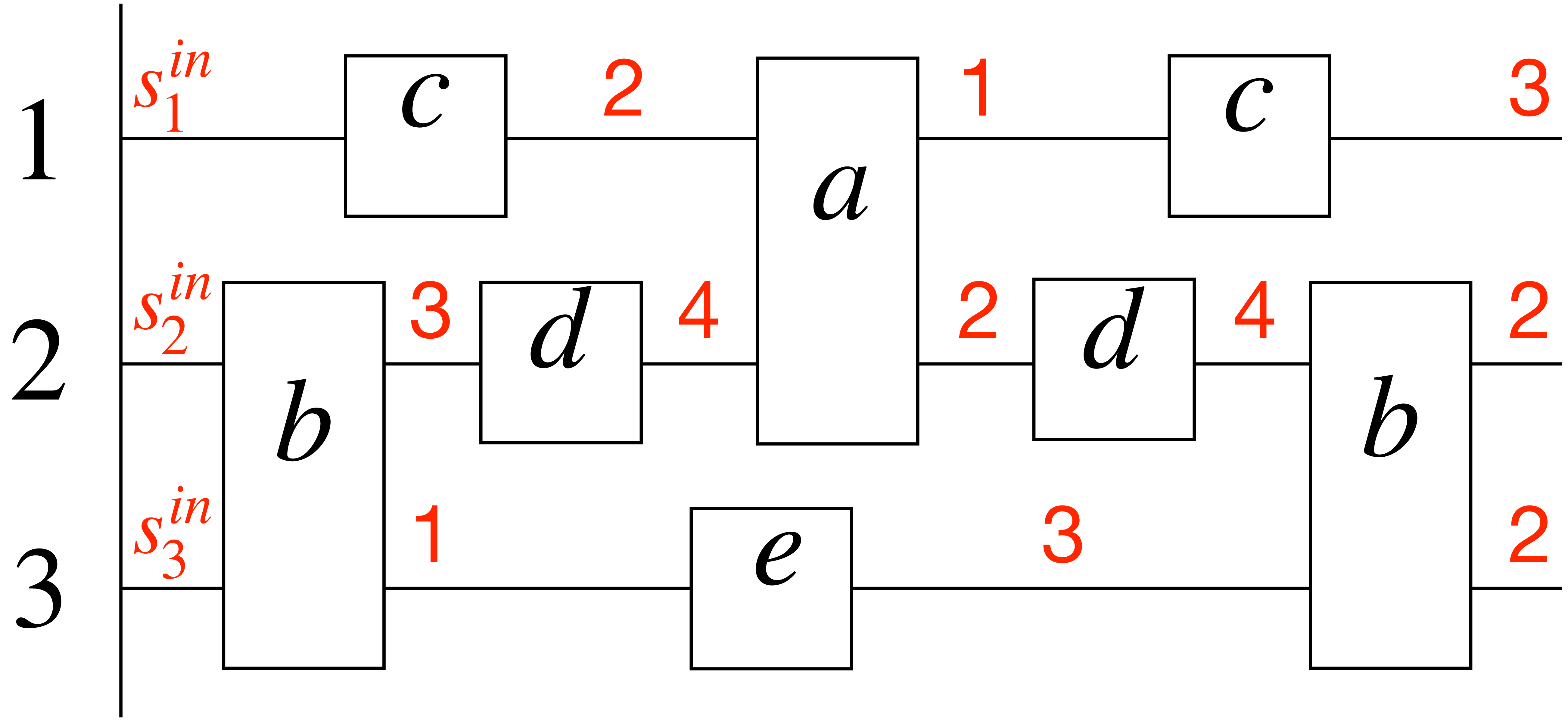
$\Gamma = \{0,1\}$ and $Y_3 \leq Y_2$

In the strict past,
the last event on process 3 is below
the last event on process 2

Asynchronous (letter-to-letter) transducer $\mathcal{T} = \left(\mathcal{A}, \{\mu_a\}_{a\in\Sigma}\right)$

- $\mathcal{A} = \left(\{S_i\}_{i\in\mathcal{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}\right)$

Asynchronous Labelling function



$\theta: Tr(\Sigma) \to Tr(\Sigma \times \Gamma)$

$\Gamma = \{0,1\}$ and $Y_3 \leq Y_2$

In the strict past,

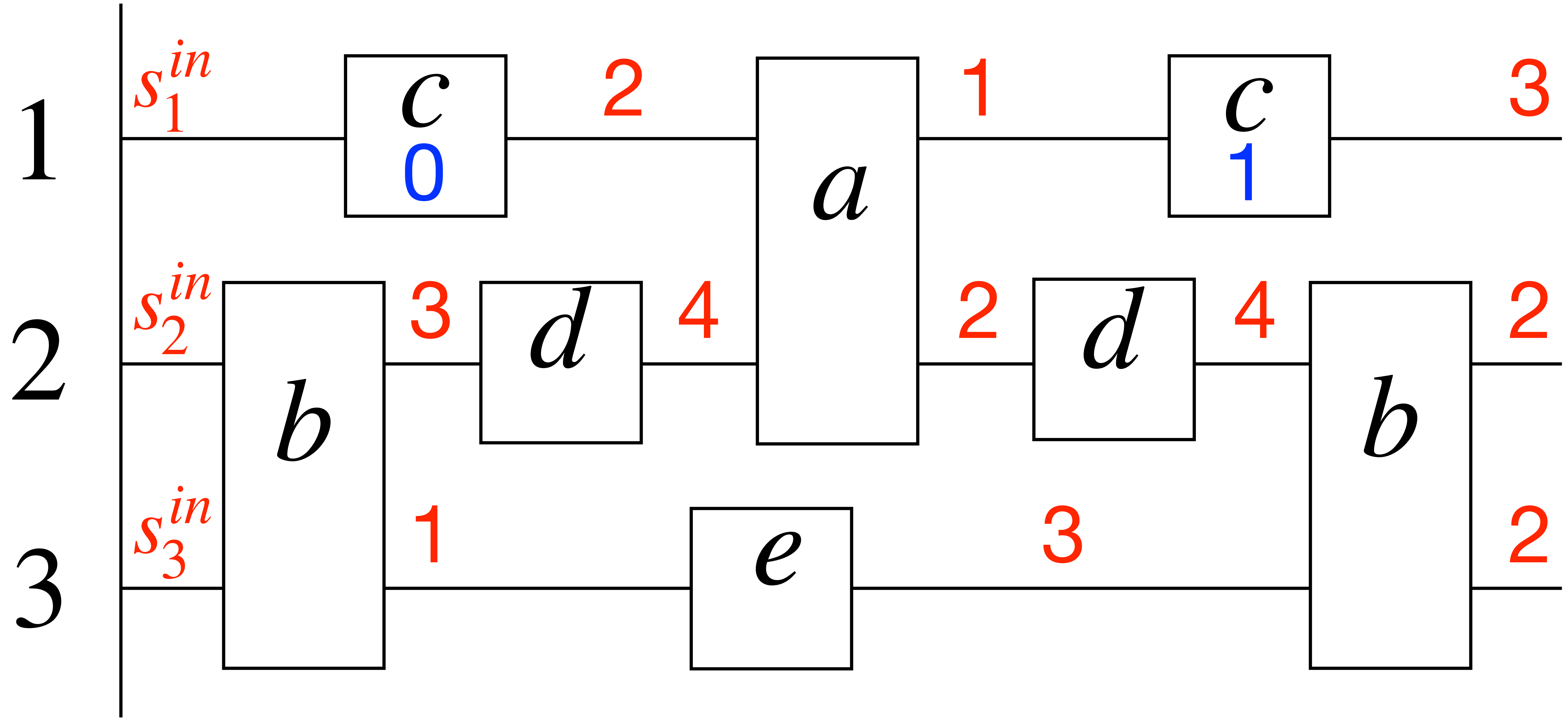
the last event on process 3 is below

the last event on process 2

Asynchronous (letter-to-letter) transducer $\mathcal{T} = (\mathcal{A}, \{\mu_a\}_{a\in\Sigma})$

- $\mathcal{A} = (\{S_i\}_{i\in\mathcal{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in})$

Asynchronous Labelling function



$\theta \colon Tr(\Sigma) \to Tr(\Sigma \times \Gamma)$

$\Gamma = \{0,1\}$ and $Y_3 \leq Y_2$

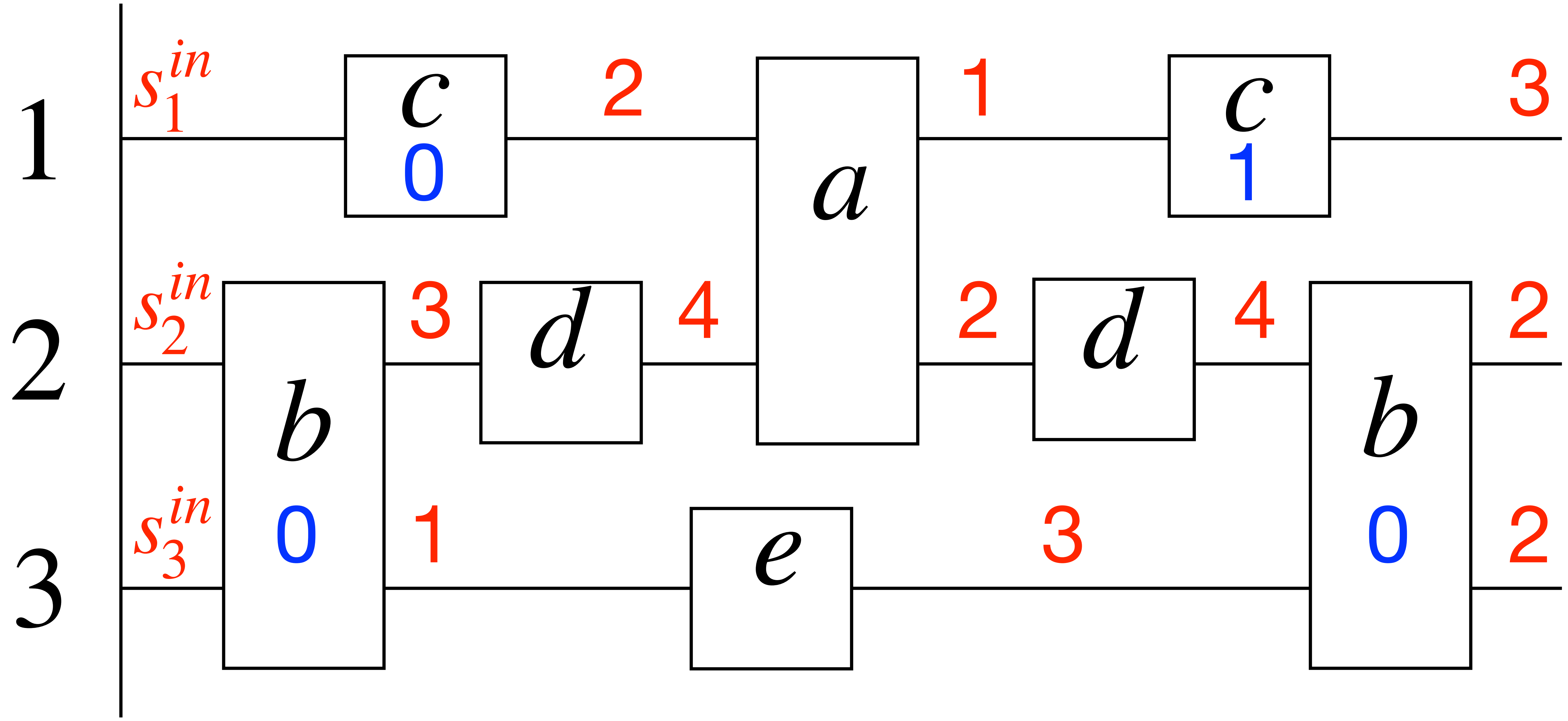
In the strict past,
the last event on process 3 is below
the last event on process 2

$\mu_c \colon S_1 \to \Gamma$

Asynchronous (letter-to-letter) transducer $\mathcal{T} = \left(\mathcal{A}, \{\mu_a\}_{a \in \Sigma}\right)$

- $\mathcal{A} = \left(\{S_i\}_{i \in \mathcal{P}}, \{\delta_a\}_{a \in \Sigma}, s^{in}\right)$
- $\mu_a \colon S_a \to \Gamma$

Asynchronous Labelling function



$\theta: Tr(\Sigma) \to Tr(\Sigma \times \Gamma)$

$\Gamma = \{0,1\}$ and $Y_3 \leq Y_2$

In the strict past,

the last event on process 3 is below

the last event on process 2

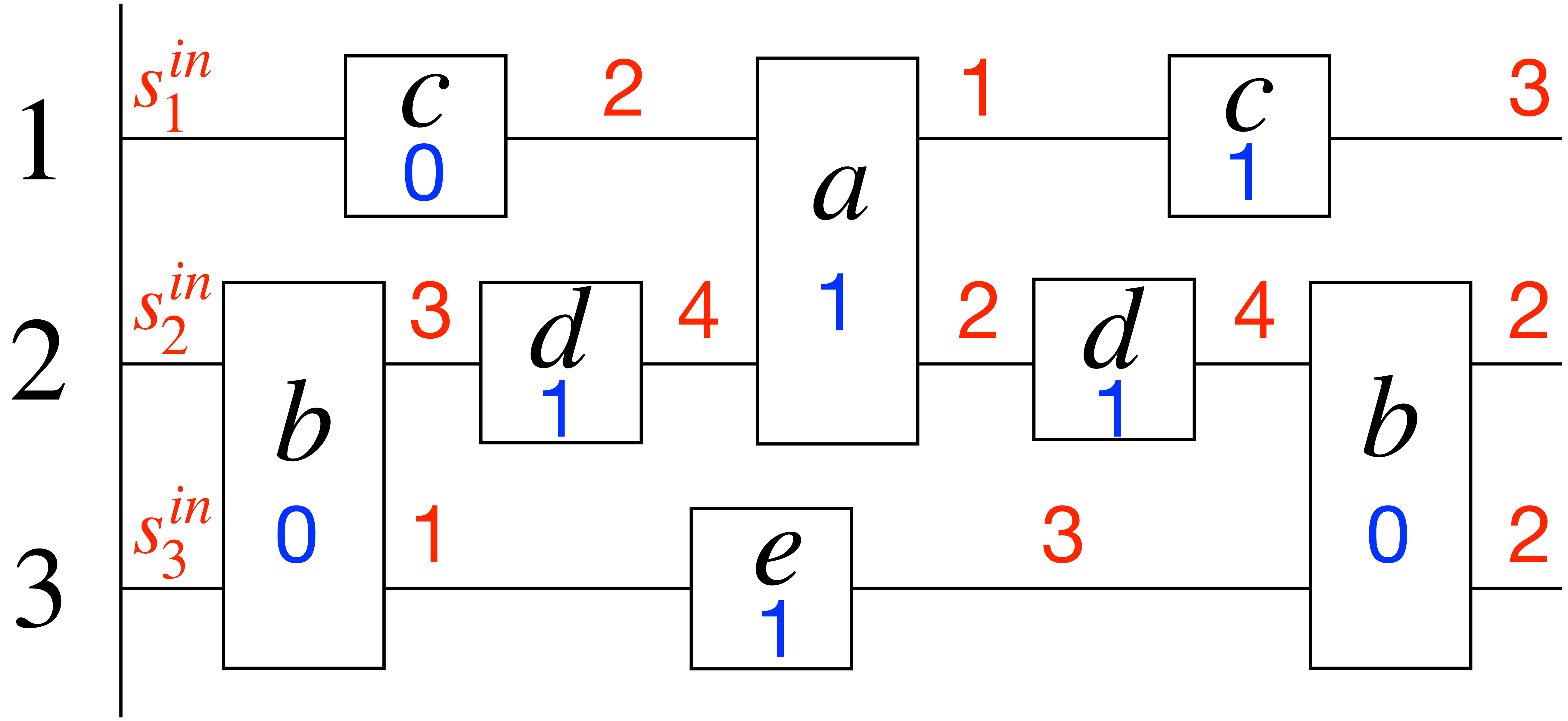
$\mu_c: S_1 \to \Gamma$

$\mu_b: S_2 \times S_3 \to \Gamma$

Asynchronous (letter-to-letter) transducer $\mathcal{T} = \left(\mathcal{A}, \{\mu_a\}_{a\in\Sigma}\right)$

- $\mathcal{A} = \left(\{S_i\}_{i\in\mathcal{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}\right)$
- $\mu_a: S_a \to \Gamma$

Asynchronous Labelling function



$\theta : Tr(\Sigma) \rightarrow Tr(\Sigma \times \Gamma)$

$\Gamma = \{0,1\}$ and $Y_3 \leq Y_2$

In the strict past,

the last event on process 3 is below

the last event on process 2

Asynchronous (letter-to-letter) transducer $\mathcal{T} = \left(\mathcal{A}, \{\mu_a\}_{a\in\Sigma}\right)$

- $\mathcal{A} = \left(\{S_i\}_{i\in\mathcal{P}}, \{\delta_a\}_{a\in\Sigma}, s^{in}\right)$
- $\mu_a : S_a \rightarrow \Gamma$

$\mu_c : S_1 \rightarrow \Gamma$

$\mu_b : S_2 \times S_3 \rightarrow \Gamma$

$\mu_d : S_2 \rightarrow \Gamma$

$\mu_a : S_1 \times S_2 \rightarrow \Gamma$

$\mu_e : S_3 \rightarrow \Gamma$

Composition - Cascade product

Composition - Cascade product

- Composition of labelling functions

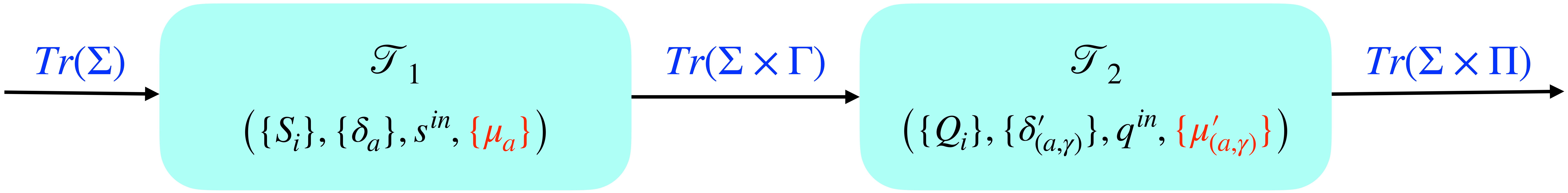
$$Tr(\Sigma) \xrightarrow{\theta_1} Tr(\Sigma \times \Gamma) \xrightarrow{\theta_2} Tr(\Sigma \times \Pi)$$

Composition - Cascade product

- Composition of labelling functions

$$Tr(\Sigma) \xrightarrow{\theta_1} Tr(\Sigma \times \Gamma) \xrightarrow{\theta_2} Tr(\Sigma \times \Pi)$$

- Cascade product of asynchronous (letter-to-letter) transducers



Composition - Cascade product

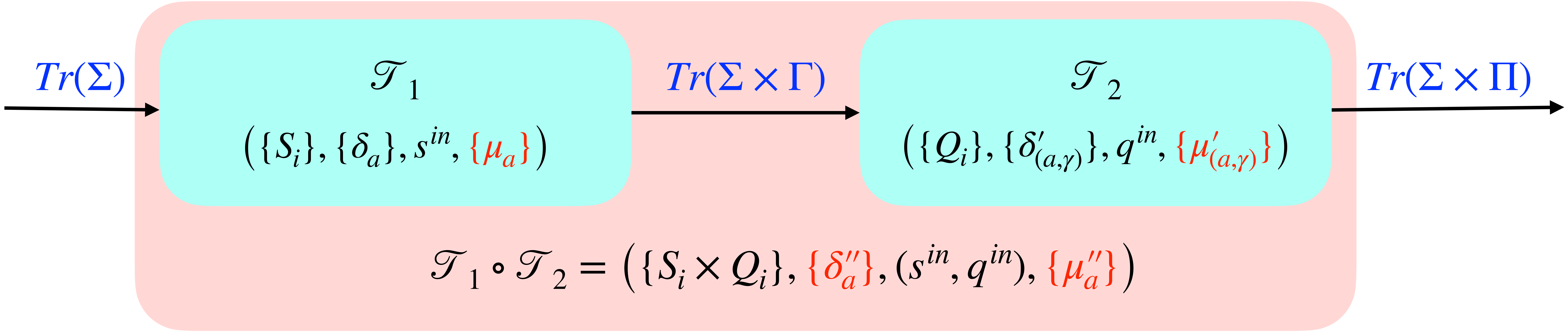
- Composition of labelling functions

$$Tr(\Sigma) \xrightarrow{\theta_1} Tr(\Sigma \times \Gamma) \xrightarrow{\theta_2} Tr(\Sigma \times \Pi)$$

$$\delta''_a(s,q) = \left(\delta_a(s), \delta'_{(a,\mu_a(s))}(q)\right)$$

$$\mu''_a(s,q) = \mu'_{(a,\mu_a(s))}(q)$$

- Cascade product of asynchronous (letter-to-letter) transducers



Cascade Decomposition

Theorem (main)

Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Cascade Decomposition

Theorem (main)

Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

$\mathcal{T} = \left(\{S_i\}_{i \in \mathcal{P}}, \{\delta_a\}, s^{in}, \{\mu_a\}\right)$ is local if at most one process is non-trivial $(|S_k| \neq 1)$

Cascade Decomposition

Theorem (main)

Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

$\mathcal{T} = \left(\{S_i\}_{i \in \mathcal{P}}, \{\delta_a\}, s^{in}, \{\mu_a\}\right)$ is local if at most one process is non-trivial $(|S_k| \neq 1)$

Corollary: Zielonka's theorem

Asynchronous Automata = Regular Trace Languages

Cascade Decomposition

Theorem (main)

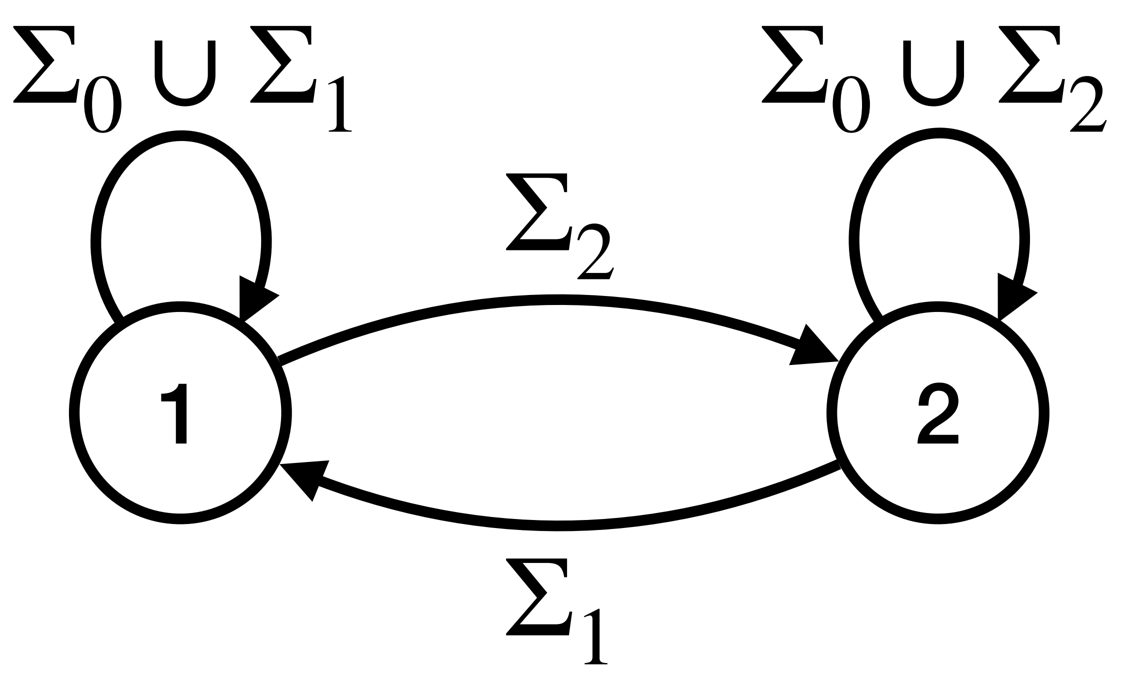
Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

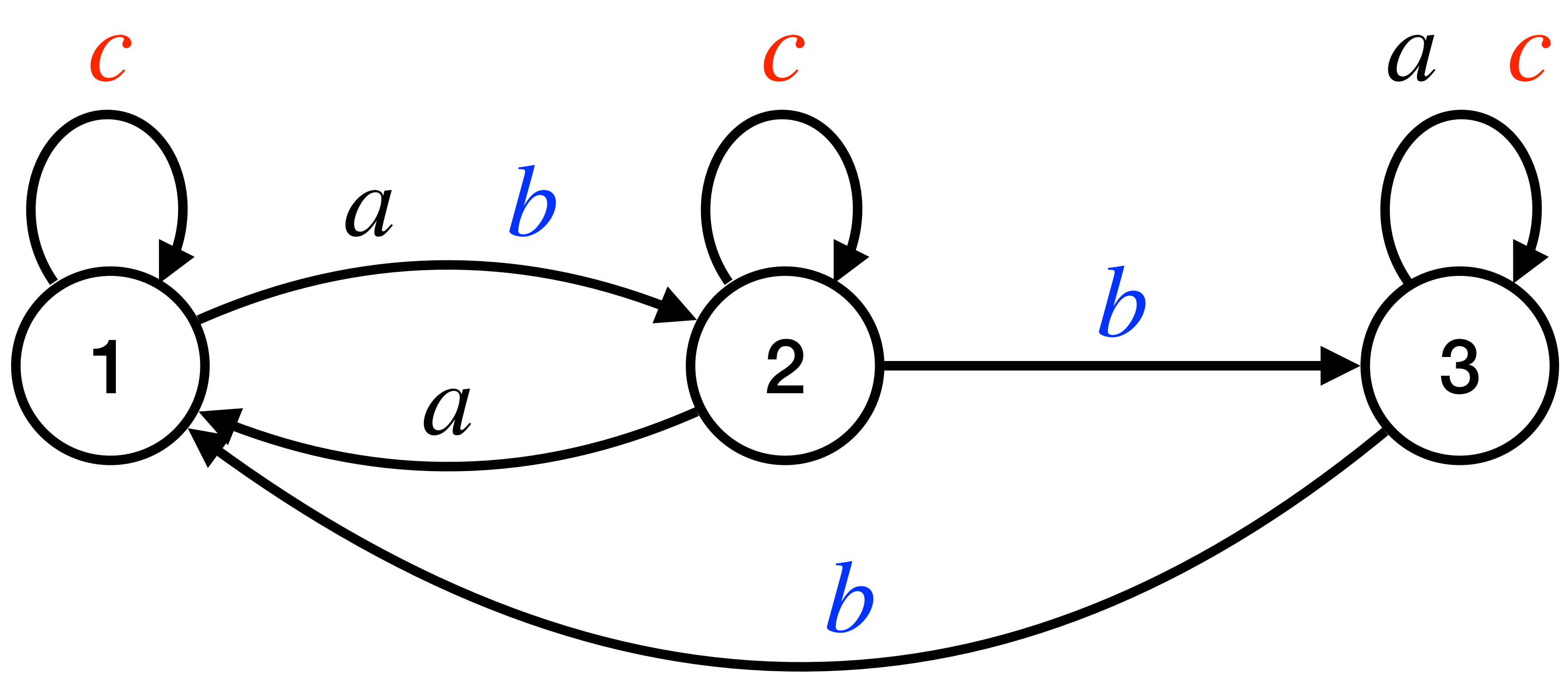
Bonus: Using Krohn-Rhodes theorem

Each local asynchronous transducer $\mathcal{T}$ can be chosen to be (on its non-trivial component)

Reset - $\mathcal{U}_2$



Permutation



Cascade Decomposition

Theorem (main)

Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Cascade Decomposition

Theorem (main)

Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Proof sketch:

Cascade Decomposition

Theorem (main)

Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Proof sketch:

- Design a local and past propositional dynamic logic (locPastPDL)
  - State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle \pi \rangle \varphi$
  - Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Cascade Decomposition

Theorem (main)

Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathscr{T}_1 \circ \mathscr{T}_2 \circ \cdots \circ \mathscr{T}_n$$

Proof sketch:

- Design a local and past propositional dynamic logic (locPastPDL)
  - State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
  - Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- Prove that event formulas are expressively complete wrt regular past predicates (difficult)

Cascade Decomposition

Theorem (main)

Any asynchronous labelling function can be realised by a cascade product of local asynchronous transducers:

$$\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Proof sketch:

- Design a local and past propositional dynamic logic (locPastPDL)
  - State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
  - Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
- Prove that event formulas are expressively complete wrt regular past predicates (difficult)
- For each event formula φ, construct by structural induction a cascade product of local asynchronous transducers computing its labelling function θ_φ (easier)

Outline

- ✅ Labelling functions, sequential transducers and cascade product
- ✅ Krohn-Rhodes theorem for aperiodic/regular word languages
- ✅ Model of concurrency: Mazurkiewicz traces and asynchronous Zielonka automata
- ✅ Asynchronous labelling functions, transducers and cascade product
- Propositional dynamic logic for traces
- Conclusion

Propositional Dynamic Logic

Propositional Dynamic Logic

- First introduced to reason about programs (Fischer, Ladner 1979)
  - State formulas $\varphi ::= p \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
  - Program expressions $\pi ::= \varphi? \mid x := e \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Propositional Dynamic Logic

- First introduced to reason about programs (Fischer, Ladner 1979)
  - State formulas $\varphi ::= p \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
  - Program expressions $\pi ::= \varphi? \mid x := e \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
    - If φ then π_1 else π_2 $(\varphi? \cdot \pi_1) + (\neg\varphi? \cdot \pi_2)$

Propositional Dynamic Logic

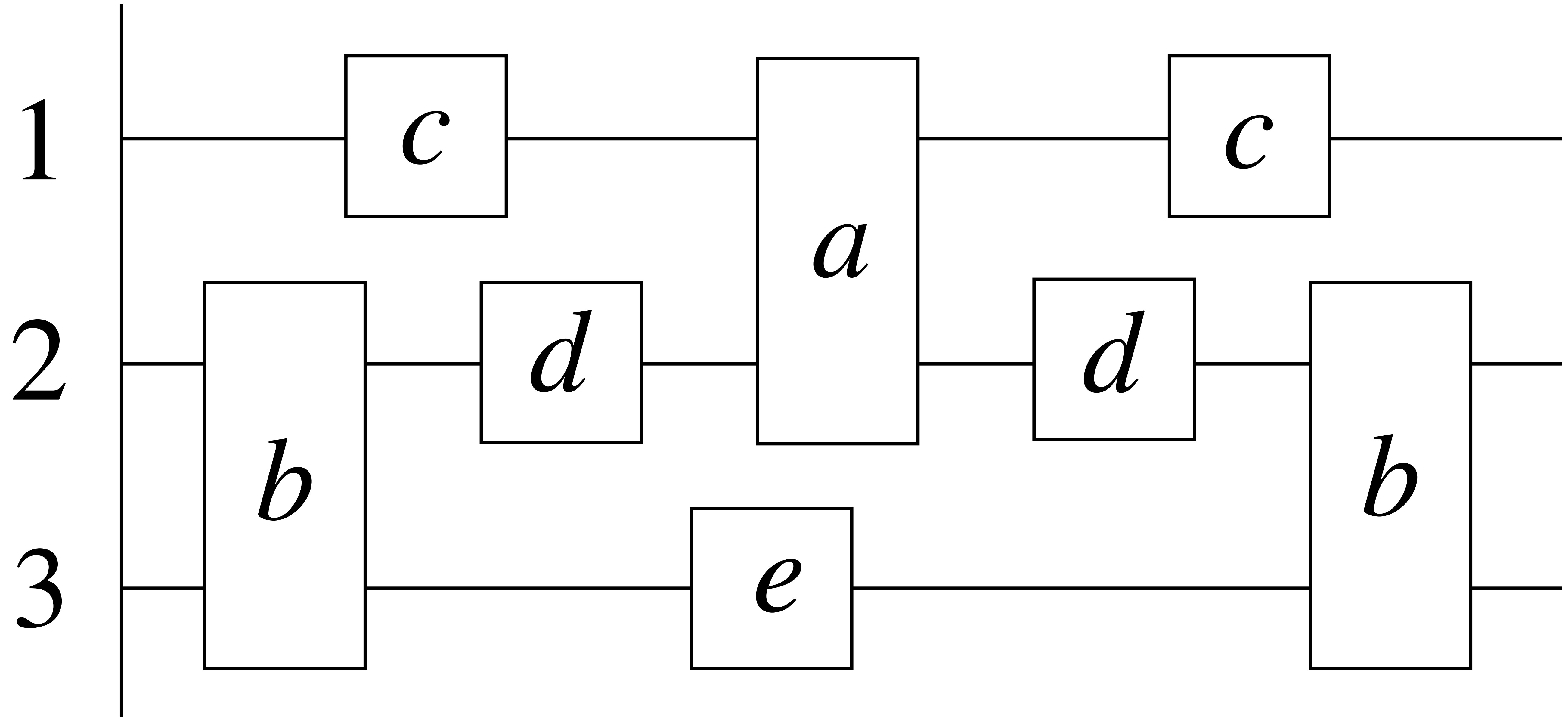
- First introduced to reason about programs (Fischer, Ladner 1979)
  - State formulas $\varphi ::= p \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
  - Program expressions $\pi ::= \varphi? \mid x := e \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
    - If φ then π_1 else π_2 $(\varphi? \cdot \pi_1) + (\neg\varphi? \cdot \pi_2)$
    - While φ do π_1 od; π_2 $(\varphi? \cdot \pi_1)^* \cdot \neg\varphi? \cdot \pi_2$

Propositional Dynamic Logic

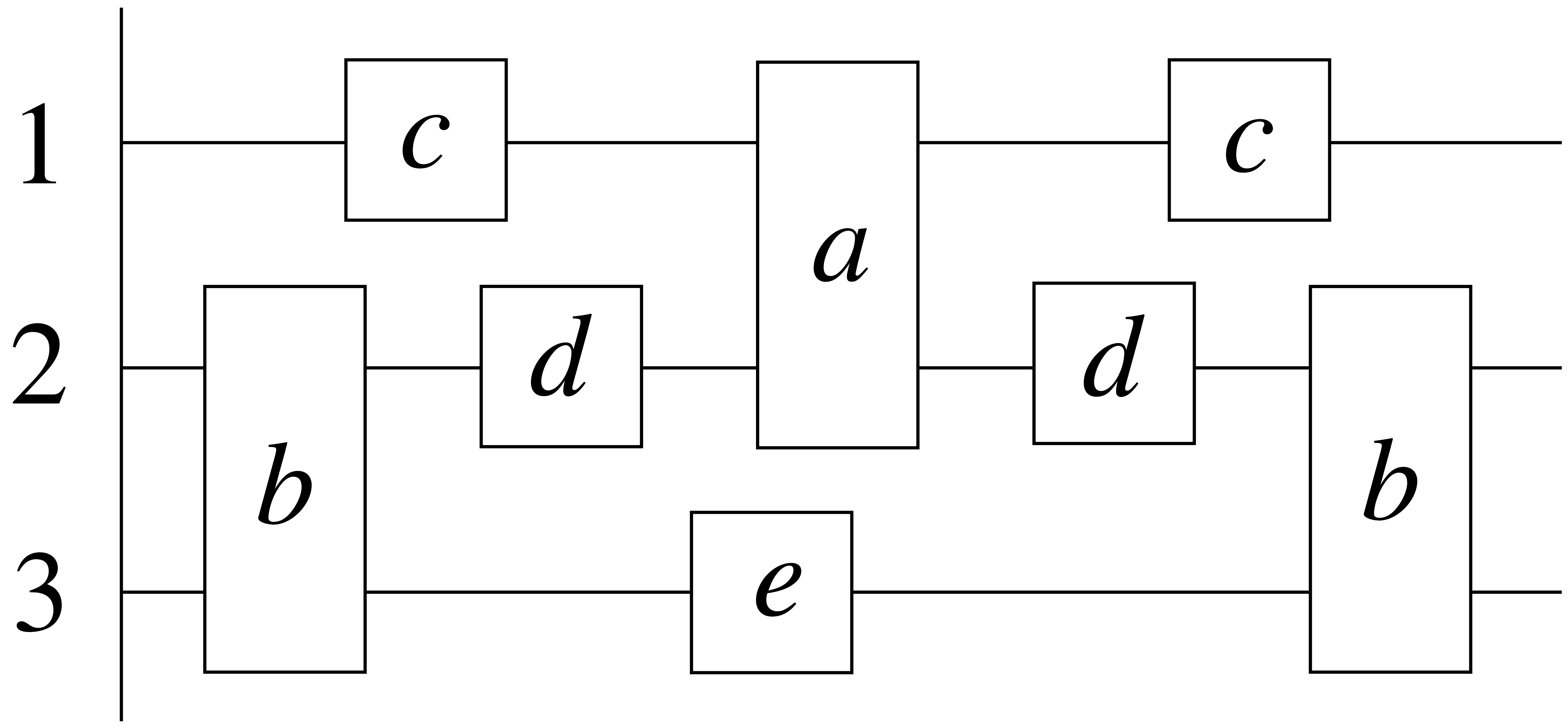
- First introduced to reason about programs (Fischer, Ladner 1979)
  - State formulas $\varphi ::= p \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
  - Program expressions $\pi ::= \varphi? \mid x := e \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$
    - If φ then π_1 else π_2 $(\varphi? \cdot \pi_1) + (\neg\varphi? \cdot \pi_2)$
    - While φ do π_1 od; π_2 $(\varphi? \cdot \pi_1)^* \cdot \neg\varphi? \cdot \pi_2$
- Interpretation over words: Linear Dynamic Logic (Giacomo, Vardi 2013)

Regular word languages = MSO definable = LDL definable

Past PDL for Traces

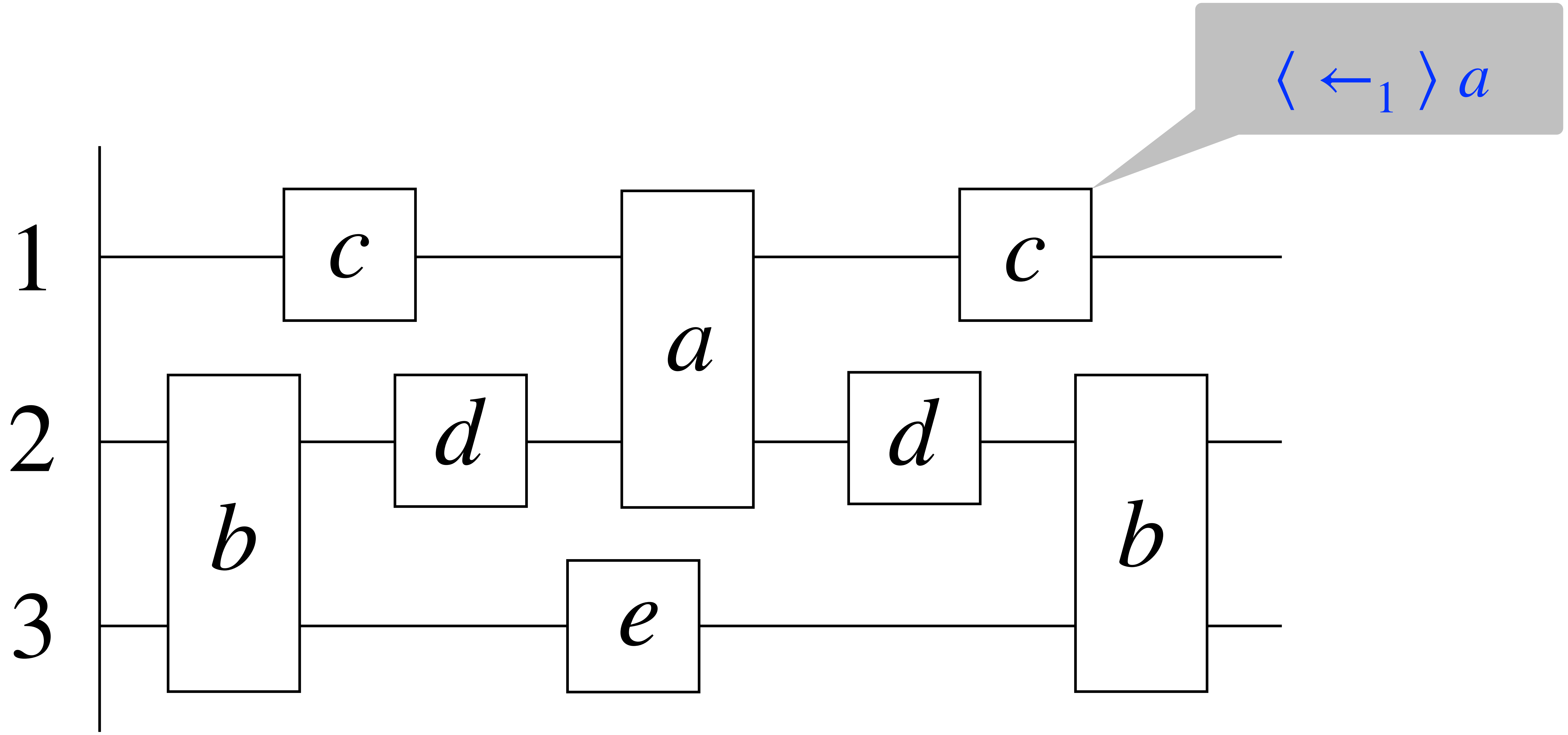


Past PDL for Traces



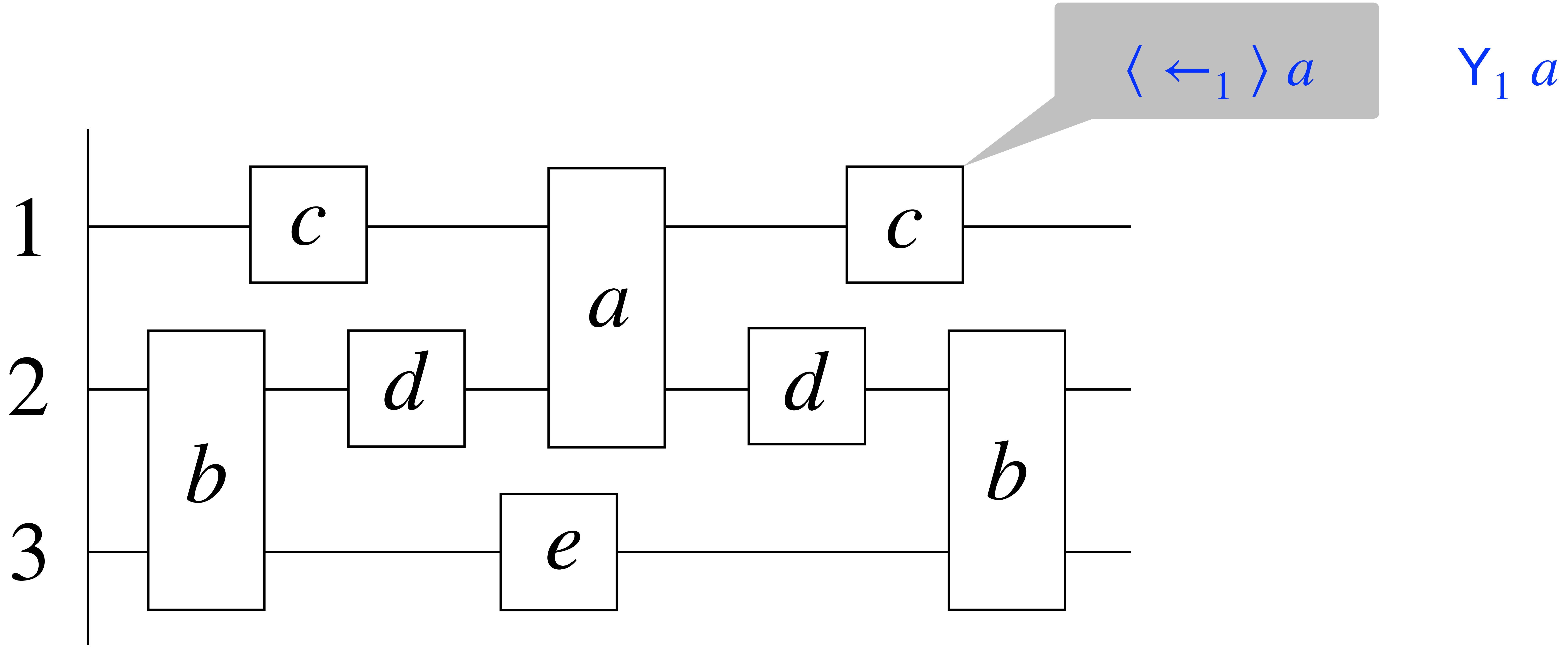
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past PDL for Traces



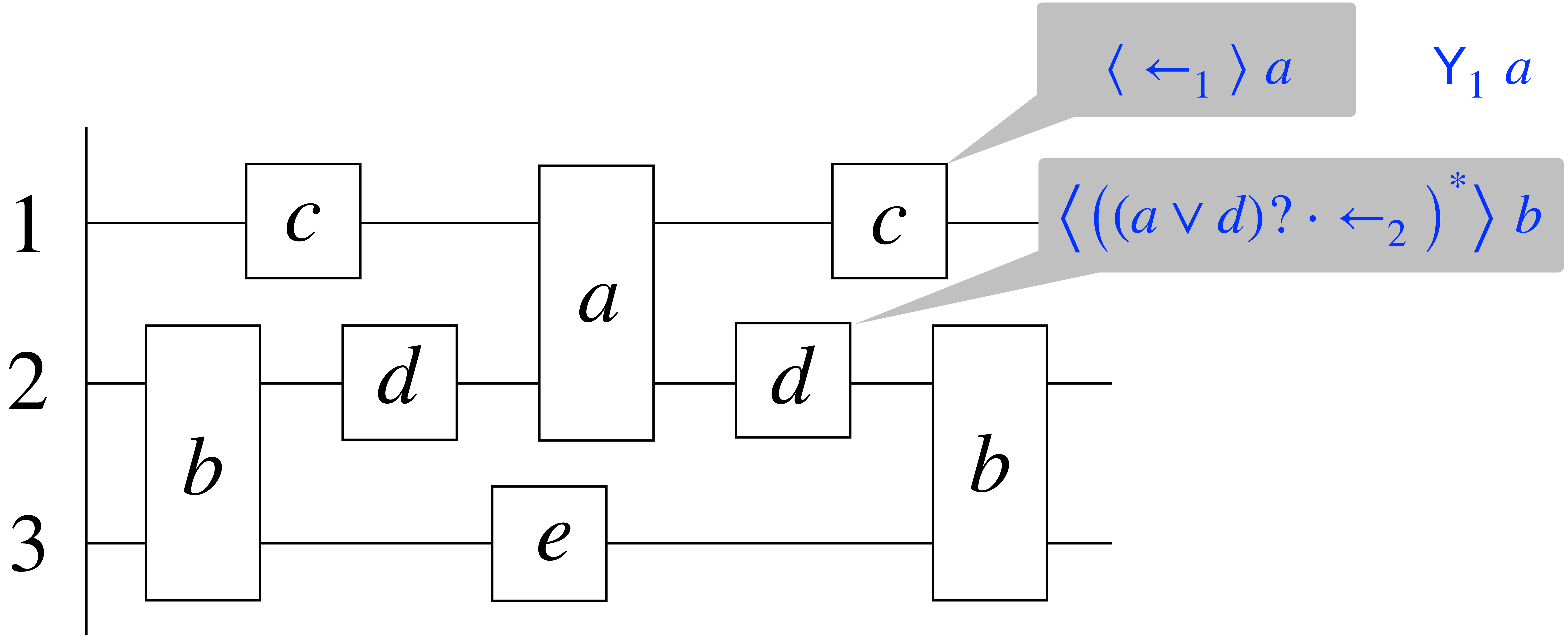
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past PDL for Traces



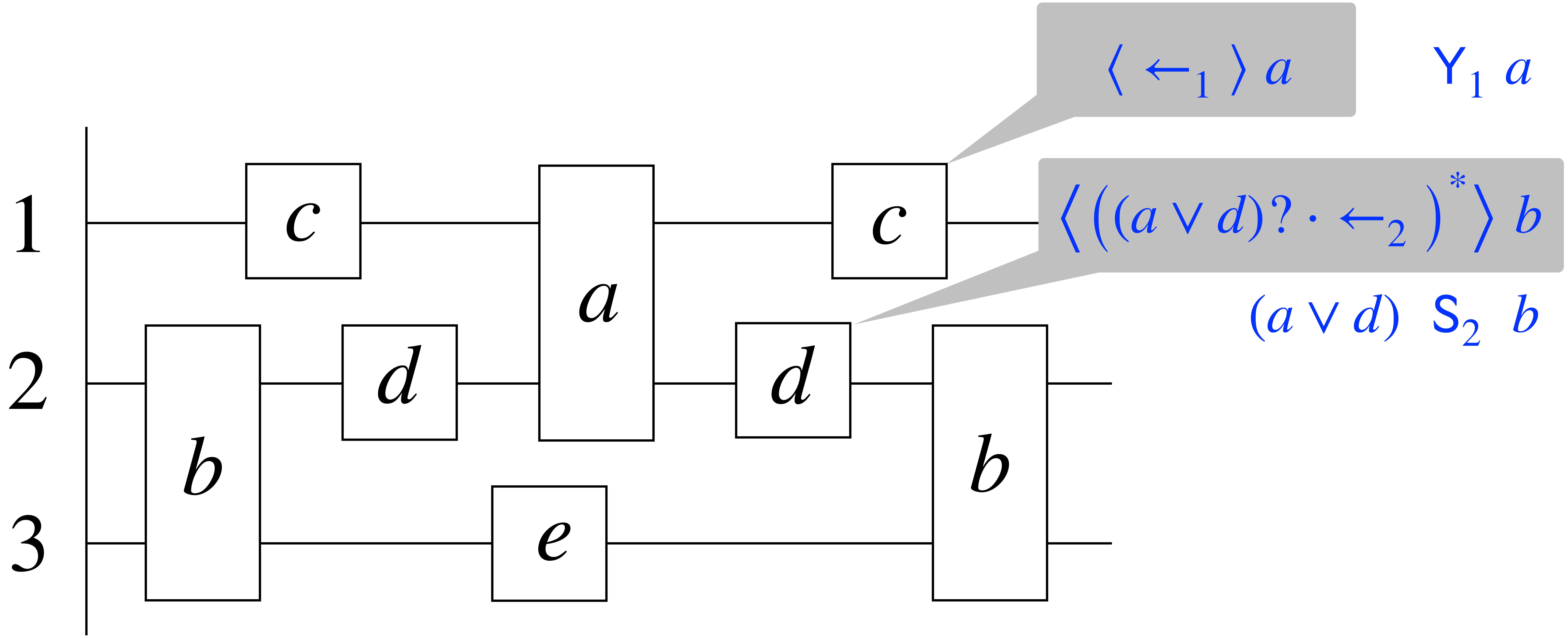
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past PDL for Traces



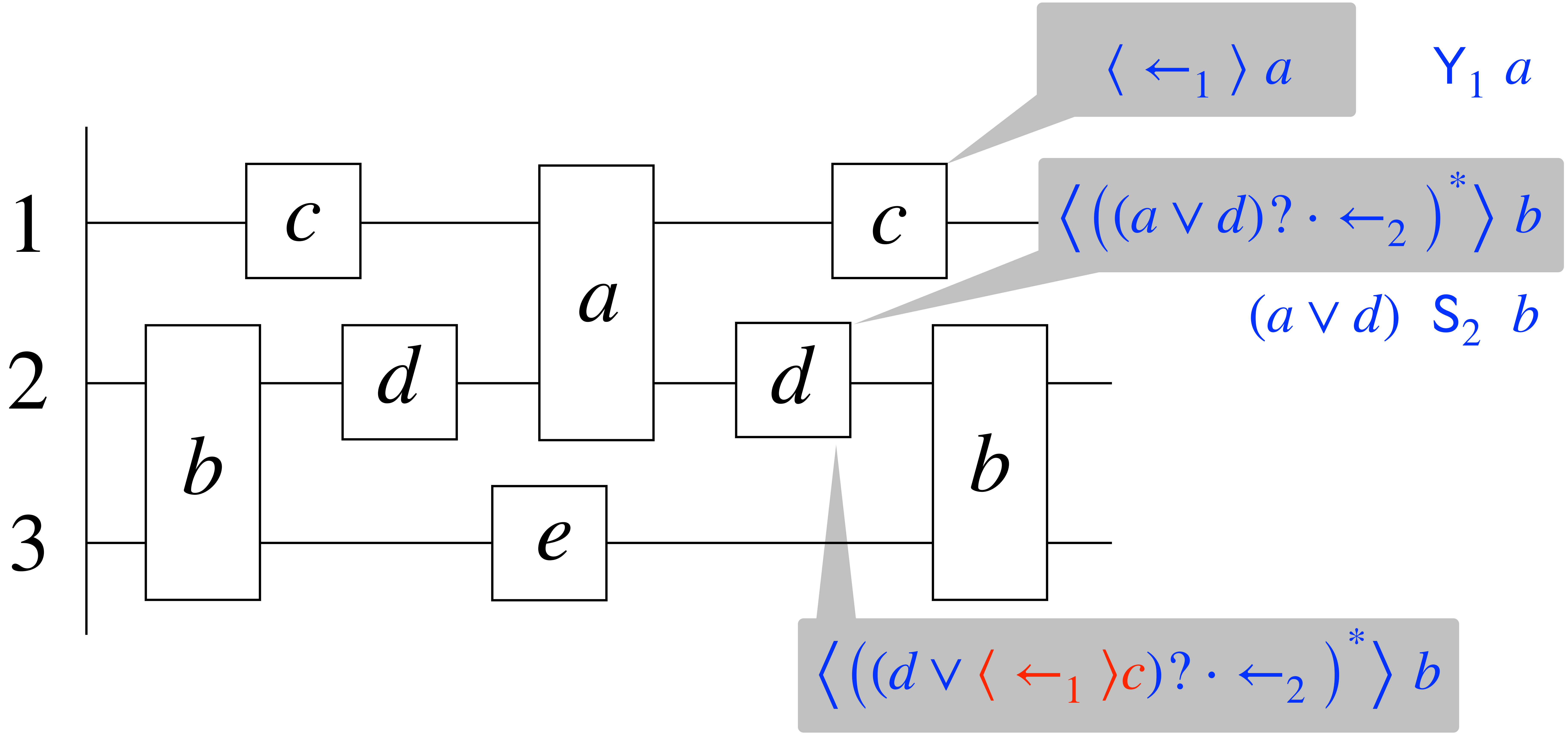
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past PDL for Traces



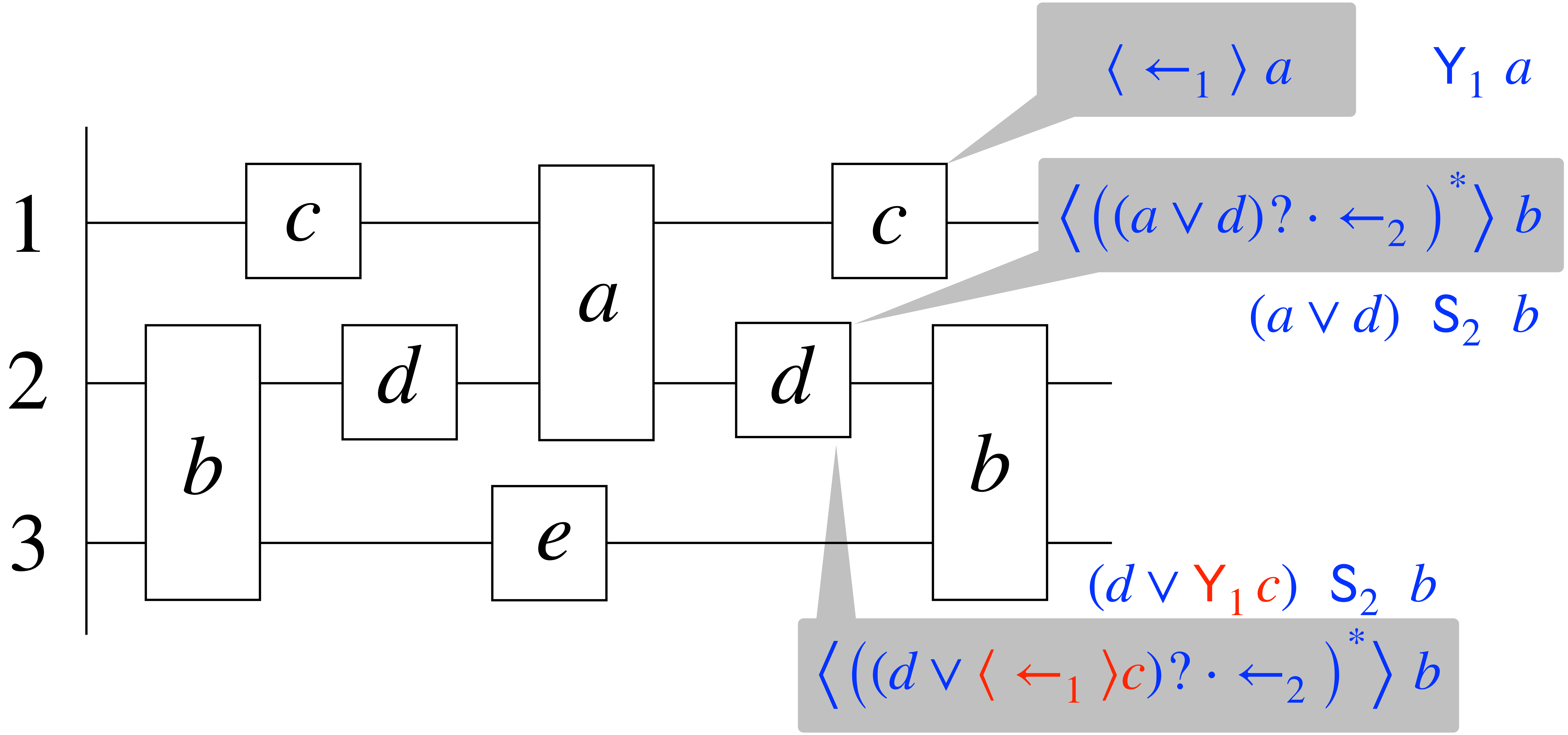
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past PDL for Traces



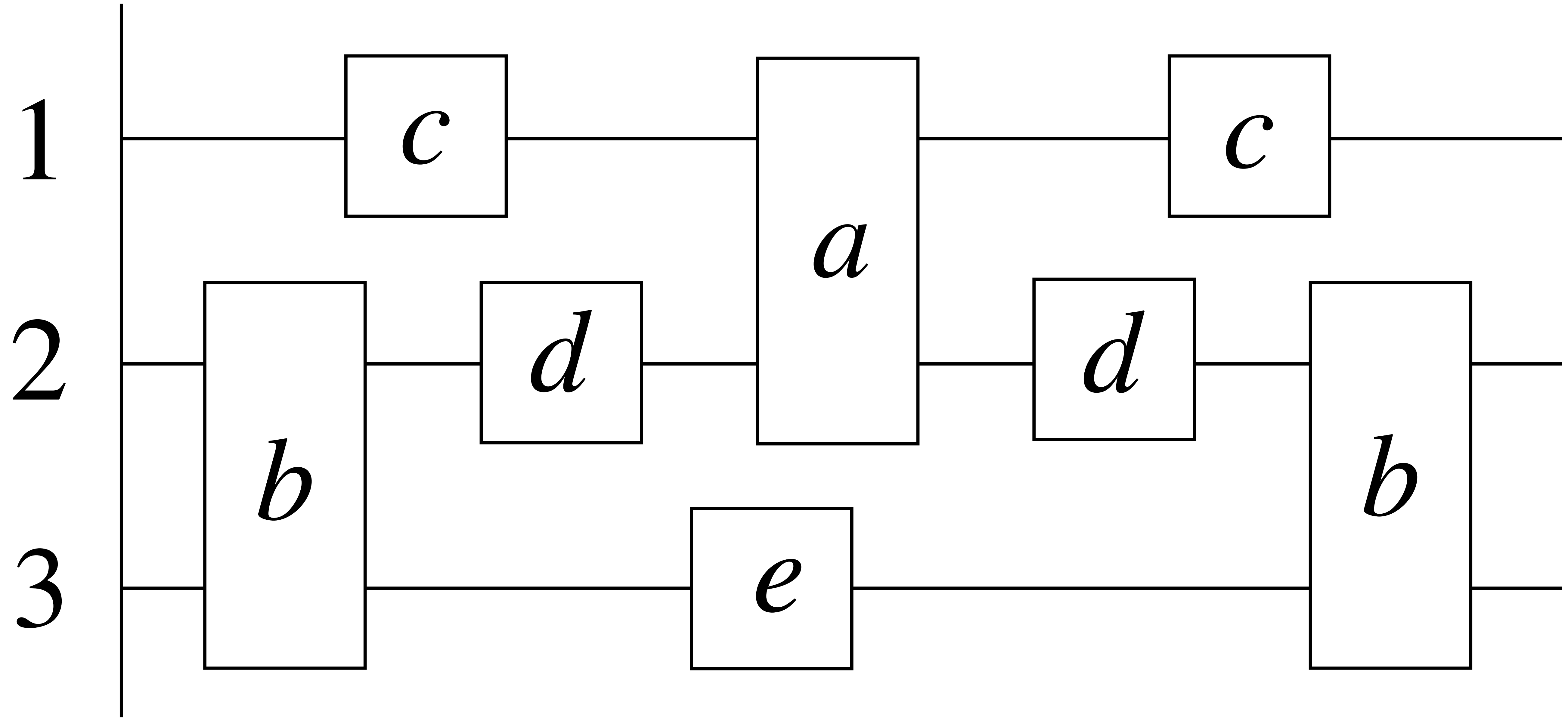
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle \pi \rangle \varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past PDL for Traces



- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

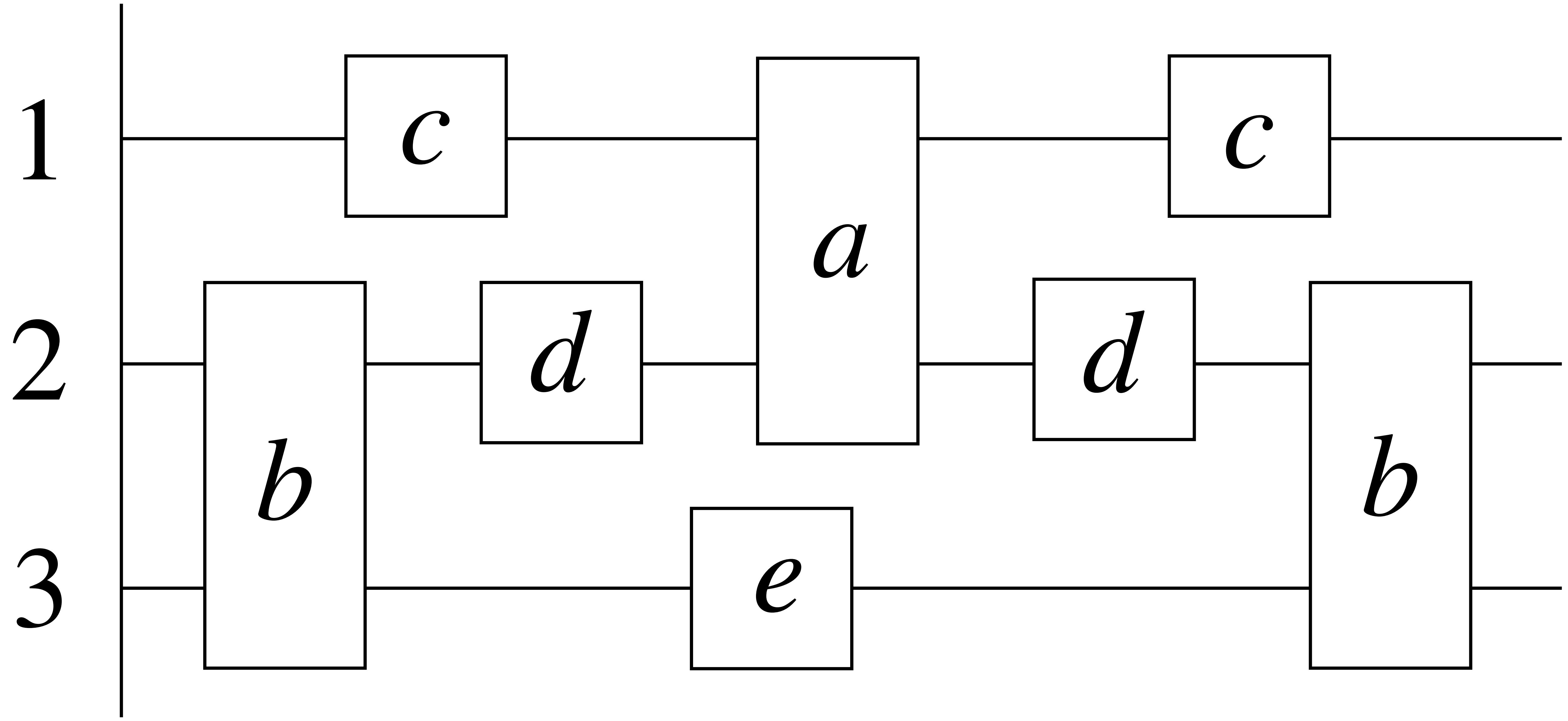
Past PDL for Traces



- Sentences / Trace formulas $\Phi ::= \mathsf{EM}_i\ \varphi \mid \Phi \vee \Phi \mid \neg\Phi$
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past PDL for Traces

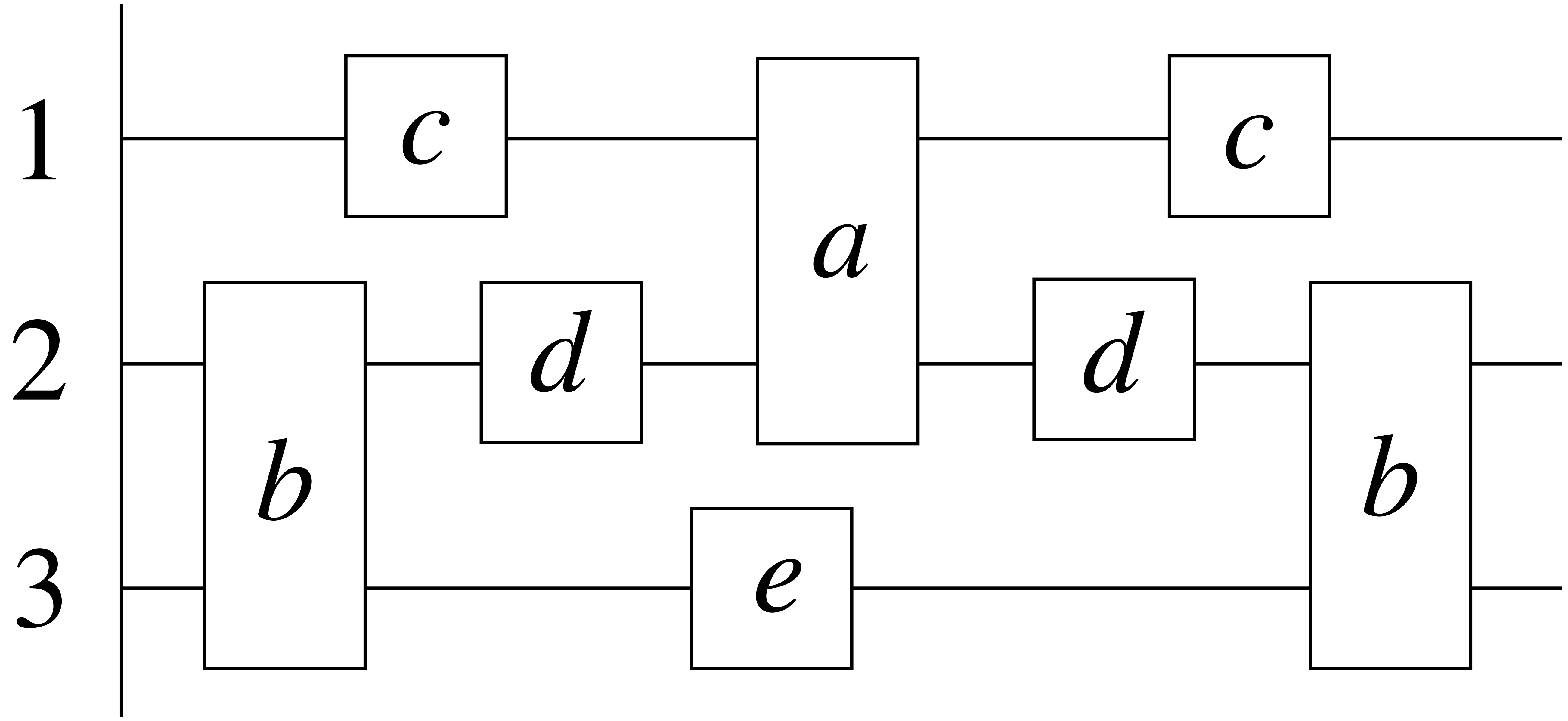
$T \models \mathsf{EM}_1\, c$



- Sentences / Trace formulas $\Phi ::= \mathsf{EM}_i\, \varphi \mid \Phi \vee \Phi \mid \neg\Phi$
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Past PDL for Traces

$T \models \mathsf{EM}_1\, c \wedge \mathsf{EM}_3 \Big(b \wedge \langle \leftarrow_2 \cdot ((a \vee d)? \cdot \leftarrow_2)^* \rangle b\Big)$



- Sentences / Trace formulas $\quad \Phi ::= \mathsf{EM}_i\, \varphi \mid \Phi \vee \Phi \mid \neg \Phi$
- State/Event formulas $\quad \varphi ::= a \mid \varphi \vee \varphi \mid \neg \varphi \mid \langle \pi \rangle \varphi$
- Program/Path expressions $\quad \pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

(Local) Past-PDL for Traces

(Local) Past-PDL for Traces

Theorem 2

- Sentences locPastPDL = PastPDL = Regular Trace Languages

(Local) Past-PDL for Traces

Theorem 2

- Sentences locPastPDL = PastPDL = Regular Trace Languages
- Event formulas locPastPDL = PastPDL = Regular Past Predicates

(Local) Past-PDL for Traces

Theorem 2

- Sentences locPastPDL = PastPDL = Regular Trace Languages
- Event formulas locPastPDL = PastPDL = Regular Past Predicates

PastPDL

$\subseteq$ easy

MSO

(Local) Past-PDL for Traces

Theorem 2

- Sentences locPastPDL = PastPDL = Regular Trace Languages
- Event formulas locPastPDL = PastPDL = Regular Past Predicates

PastPDL

⊆ easy

MSO

⊆ known

Morphisms

(Local) Past-PDL for Traces

Theorem 2

- Sentences — locPastPDL = PastPDL = Regular Trace Languages
- Event formulas — locPastPDL = PastPDL = Regular Past Predicates

PastPDL

$\subseteq$ easy

MSO

$\subseteq$ known

Morphisms

$\subseteq$ difficult

locPastPDL

(Local) Past-PDL for Traces

Theorem 2

- Sentences locPastPDL = PastPDL = Regular Trace Languages
- Event formulas locPastPDL = PastPDL = Regular Past Predicates

PastPDL

⊆ easy

MSO

⊆ known

Morphisms

⊆ difficult

locPastPDL

- Let $\eta\colon Tr(\Sigma) \to M$ be a morphism to a finite monoid

(Local) Past-PDL for Traces

Theorem 2

- Sentences locPastPDL = PastPDL = Regular Trace Languages
- Event formulas locPastPDL = PastPDL = Regular Past Predicates

PastPDL

$\subseteq$ easy

MSO

$\subseteq$ known

Morphisms

$\subseteq$ difficult

locPastPDL

- Let $\eta\colon Tr(\Sigma) \to M$ be a morphism to a finite monoid
- For each $m \in M$, we construct a locPastPDL event formula $\varphi^{(m)}$ such that,

  if T is a prime trace (i.e., having a single maximal event),

$$\eta(T) = m \text{ if and only if } T, \max(T) \models \varphi^{(m)}$$

Induction on the number of processes

(Local) Past-PDL for Traces

Theorem 2

- Sentences   locPastPDL = PastPDL = Regular Trace Languages
- Event formulas   locPastPDL = PastPDL = Regular Past Predicates

PastPDL

⊆ easy

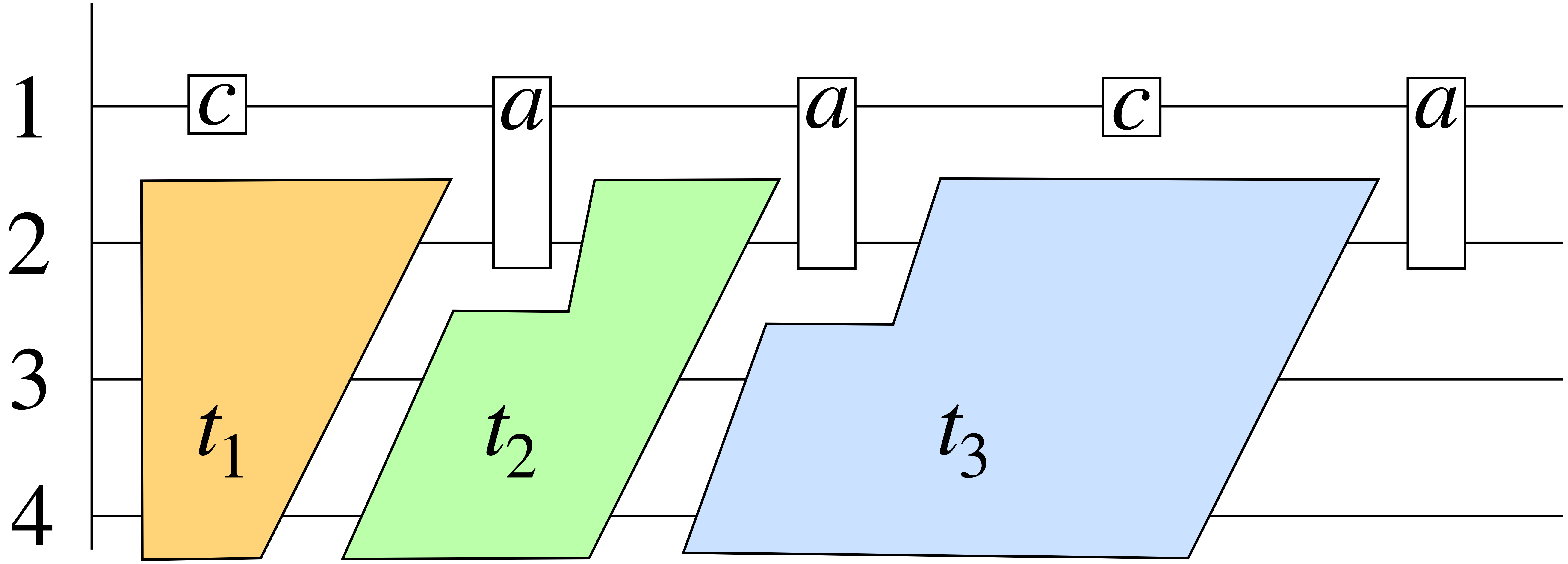
MSO

⊆ known

Morphisms

⊆ difficult

locPastPDL



(Local) Past-PDL for Traces

Theorem 2

- Sentences — locPastPDL = PastPDL = Regular Trace Languages
- Event formulas — locPastPDL = PastPDL = Regular Past Predicates

PastPDL

I∩ easy

MSO

I∩ known

Morphisms

I∩ difficult

locPastPDL

(Local) Past-PDL for Traces

Theorem 2

- Sentences   locPastPDL = PastPDL = Regular Trace Languages
- Event formulas   locPastPDL = PastPDL = Regular Past Predicates

PastPDL

$\subseteq$ easy

MSO

$\subseteq$ known

Morphisms

$\subseteq$ difficult

locPastPDL

- Let $\eta: Tr(\Sigma) \to M$ be a morphism to a finite monoid
- For each $m \in M$, we construct a locPastPDL event formula $\varphi^{(m)}$ such that,

  if T is a prime trace (i.e., having a single maximal event),

$$\eta(T) = m \text{ if and only if } T, \max(T) \models \varphi^{(m)}$$

Induction on the number of processes

(Local) Past-PDL for Traces

Theorem 2

- Sentences — locPastPDL = PastPDL = Regular Trace Languages
- Event formulas — locPastPDL = PastPDL = Regular Past Predicates

PastPDL

⊆ easy

MSO

⊆ known

Morphisms

⊆ difficult

locPastPDL

- Let $\eta\colon Tr(\Sigma) \to M$ be a morphism to a finite monoid
- For each $m \in M$, we construct a locPastPDL event formula $\varphi^{(m)}$ such that,

  if T is a prime trace (i.e., having a single maximal event),

  $$\eta(T) = m \text{ if and only if } T, \max(T) \models \varphi^{(m)}$$

  Induction on the number of processes
- For each $m \in M$, we construct a sentence $\Phi^{(m)}$ which defines $\eta^{-1}(m)$

  decompose an arbitrary (non prime) trace into a product of prime traces.

(Local) Past-PDL for Traces

Theorem 2

- Sentences locPastPDL = PastPDL = Regular Trace Languages
- Event formulas locPastPDL = PastPDL = Regular Past Predicates

PastPDL

⫅ easy

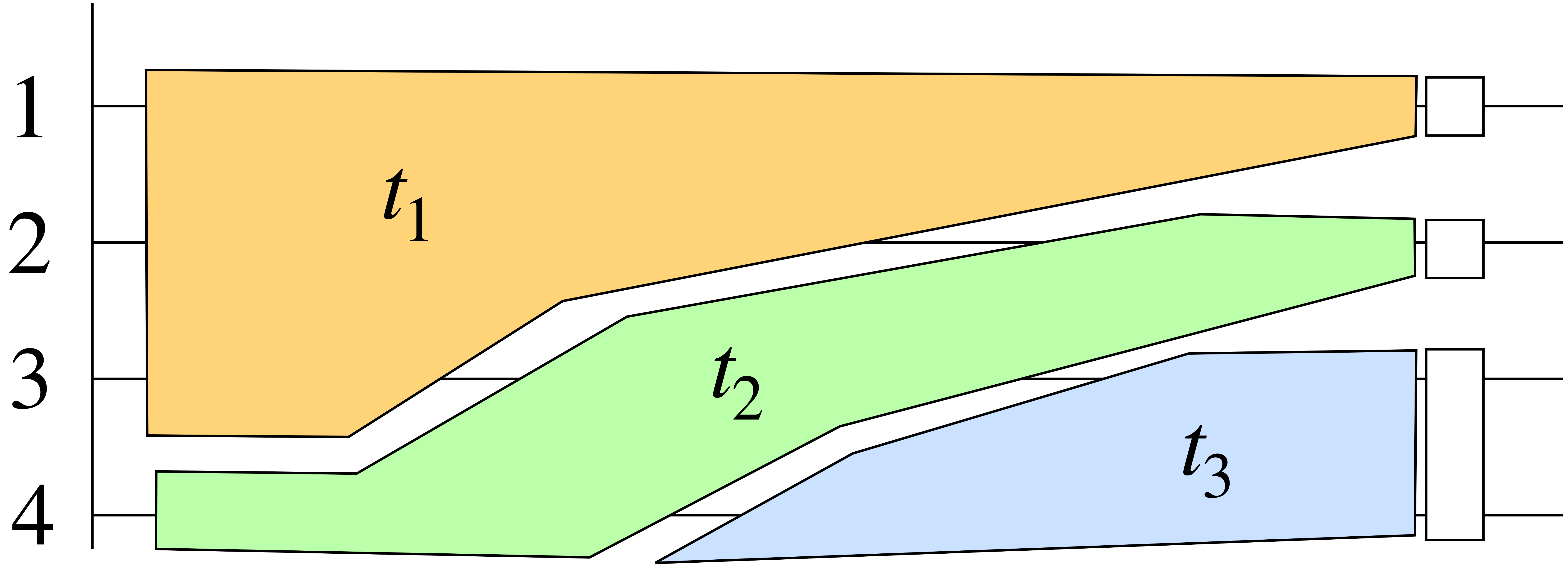
MSO

⫅ known

Morphisms

⫅ difficult

locPastPDL



(Local) Past-PDL for Traces

Theorem 2

- Sentences   locPastPDL = PastPDL = Regular Trace Languages
- Event formulas   locPastPDL = PastPDL = Regular Past Predicates

PastPDL

$\subseteq$ easy

MSO

$\subseteq$ known

Morphisms

$\subseteq$ difficult

locPastPDL

- Let $\eta\colon Tr(\Sigma) \to M$ be a morphism to a finite monoid
- For each $m \in M$, we construct a locPastPDL event formula $\varphi^{(m)}$ such that,

  if T is a prime trace (i.e., having a single maximal event),

  $$\eta(T) = m \text{ if and only if } T, \max(T) \models \varphi^{(m)}$$

  Induction on the number of processes
- For each $m \in M$, we construct a sentence $\Phi^{(m)}$ which defines $\eta^{-1}(m)$

  decompose an arbitrary (non prime) trace into a product of prime traces.

(Local) Past-PDL for Traces

Theorem 2

- Sentences — locPastPDL = PastPDL = Regular Trace Languages
- Event formulas — locPastPDL = PastPDL = Regular Past Predicates

PastPDL

⊆ easy

MSO

⊆ known

Morphisms

⊆ difficult

locPastPDL

- Let $\eta : Tr(\Sigma) \to M$ be a morphism to a finite monoid
- For each $m \in M$, we construct a locPastPDL event fo… …at,

  if T is a prime trace (i.e., having a single maxi…

  $$\eta(T) = m \text{ if and o… } \varphi^{(m)}$$

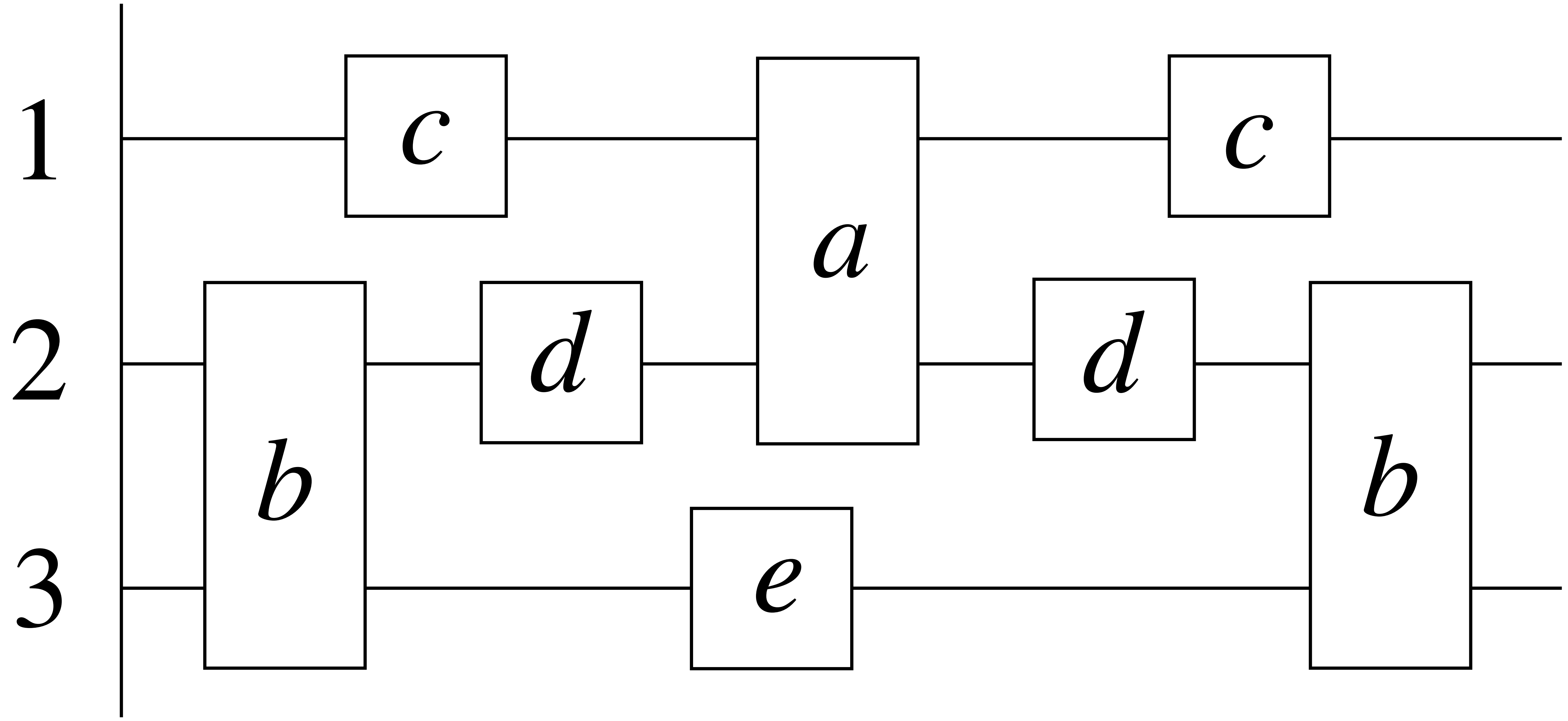
  Induction on the number of pro…
- For each $m \in M$, we constr… $\Phi^{(m)}$ which defines $\eta^{-1}(m)$

  decompose an arbitrary (non p… trace into a product of prime traces.

Decompositions depend crucially on

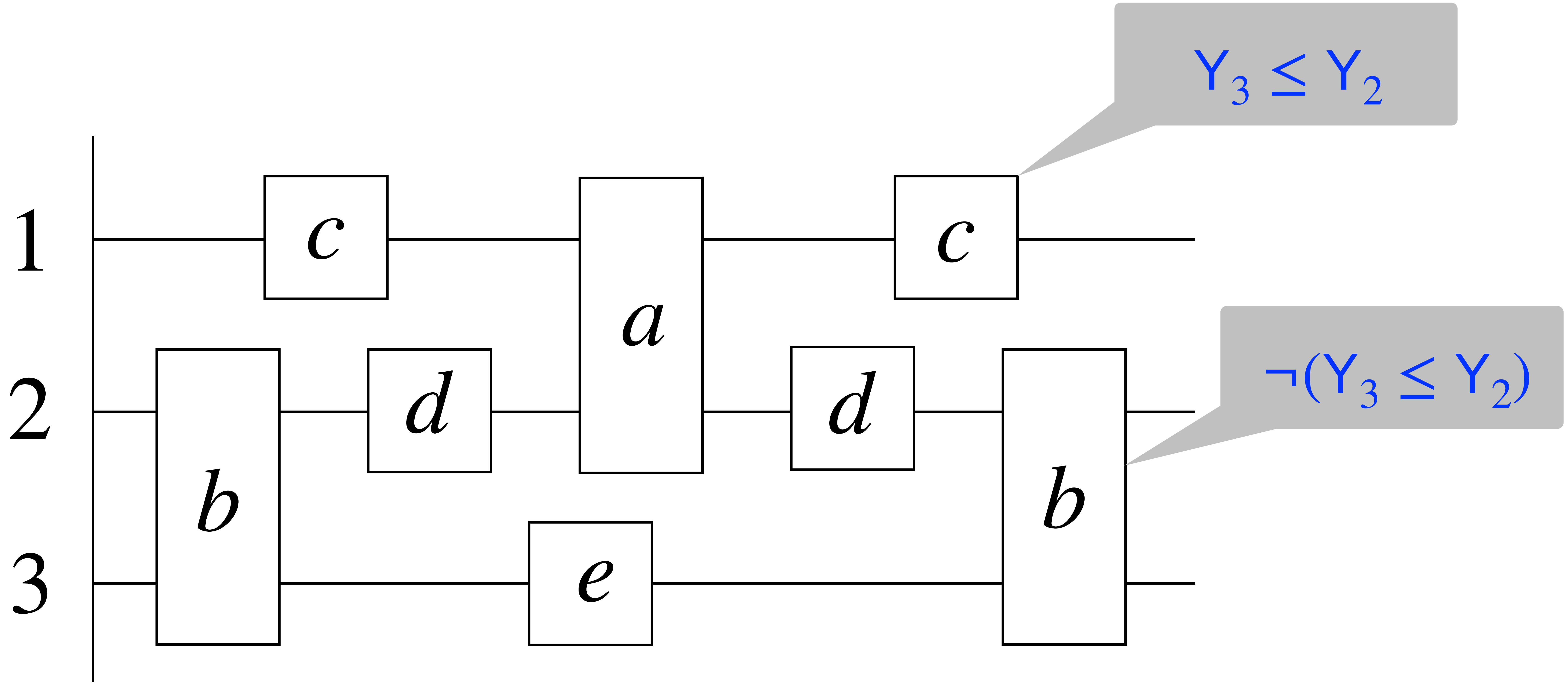
$Y_i \leq Y_j$, $Y_{i,k} \leq Y_j$ and $L_i \leq L_j$, $L_{i,k} \leq L_j$

Extended locPastPDL for Traces



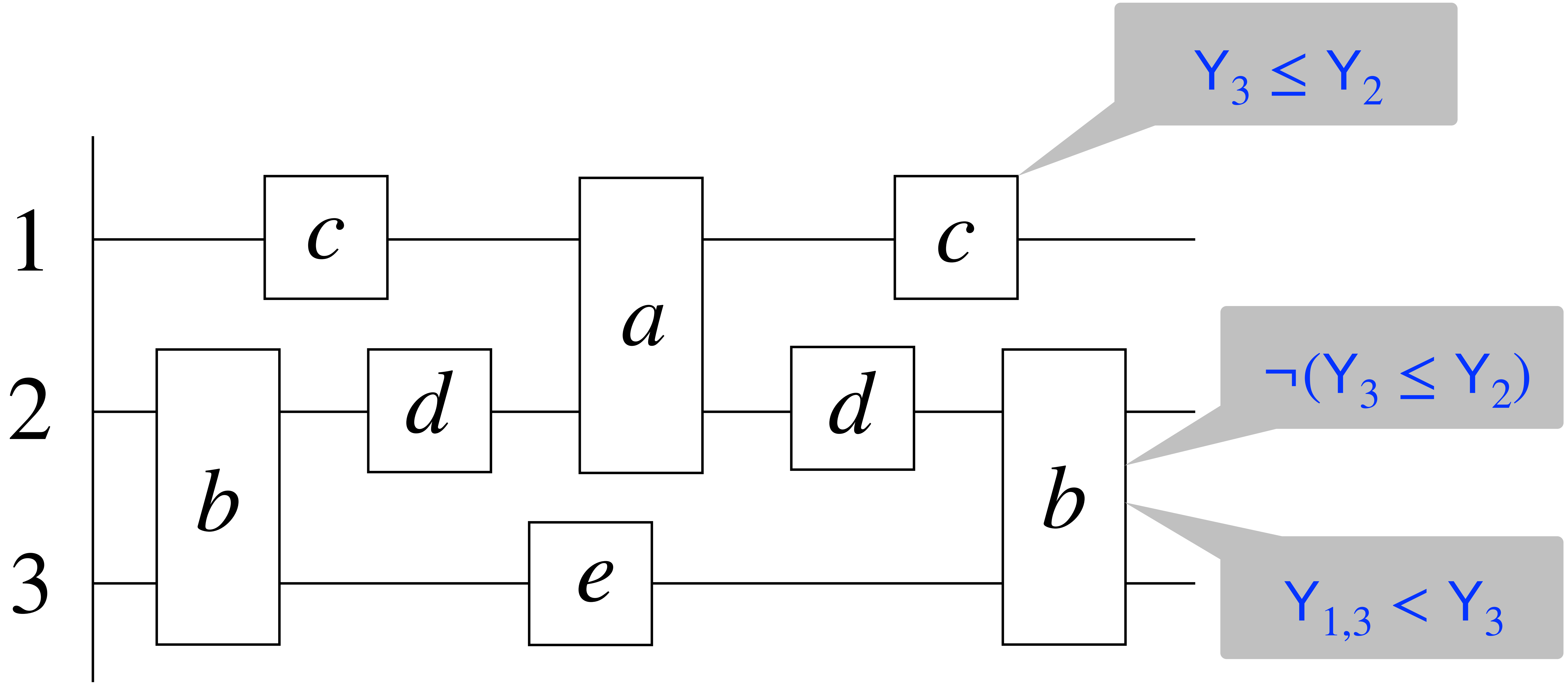
- Trace formulas / Sentences $\Phi ::= \mathsf{EM}_i\ \varphi \mid \Phi \vee \Phi \mid \neg\Phi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,k} \leq \mathsf{L}_j$
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,k} \leq \mathsf{Y}_j$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Extended locPastPDL for Traces



- Trace formulas / Sentences $\Phi ::= \mathsf{EM}_i\ \varphi \mid \Phi \vee \Phi \mid \neg\Phi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,k} \leq \mathsf{L}_j$
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,k} \leq \mathsf{Y}_j$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

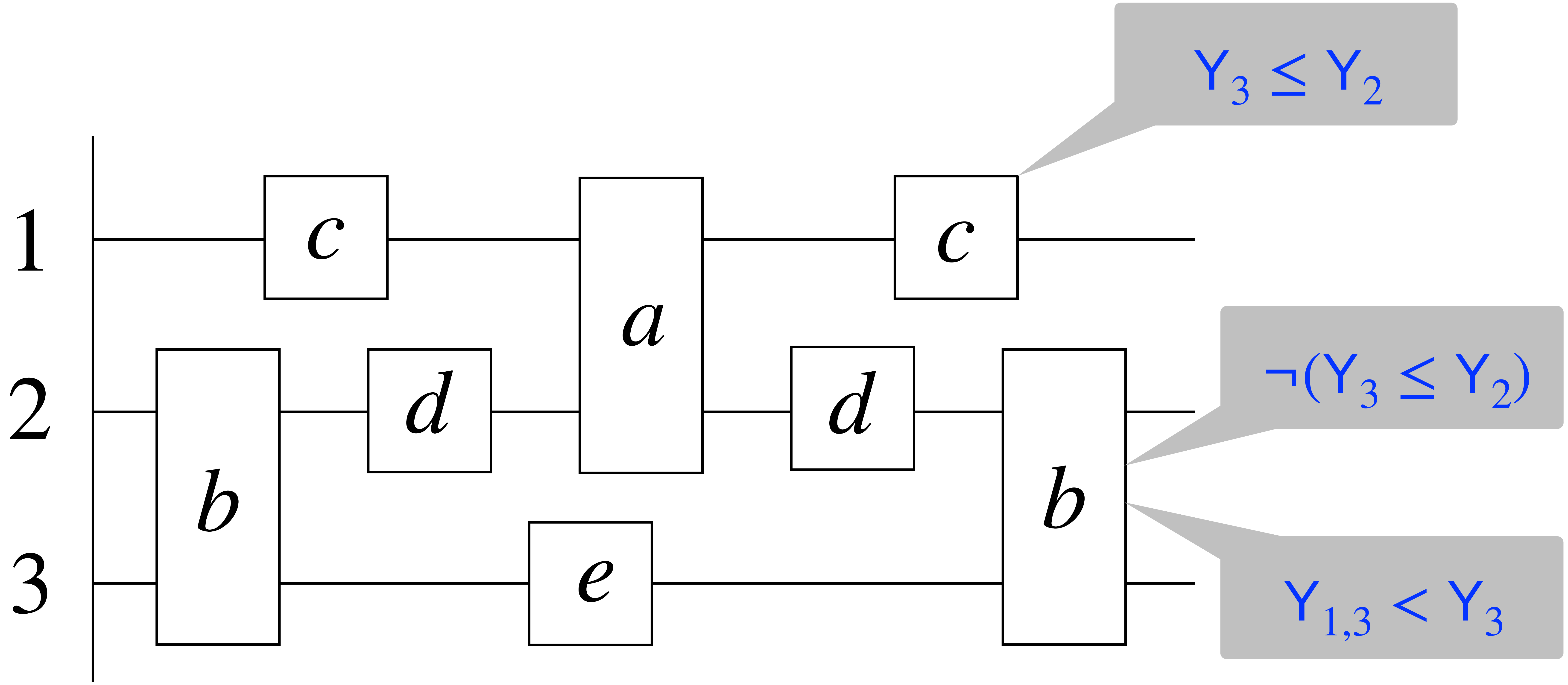
Extended locPastPDL for Traces



- Trace formulas / Sentences $\Phi ::= \mathsf{EM}_i\, \varphi \mid \Phi \vee \Phi \mid \neg\Phi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,k} \leq \mathsf{L}_j$
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,k} \leq \mathsf{Y}_j$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Extended locPastPDL for Traces

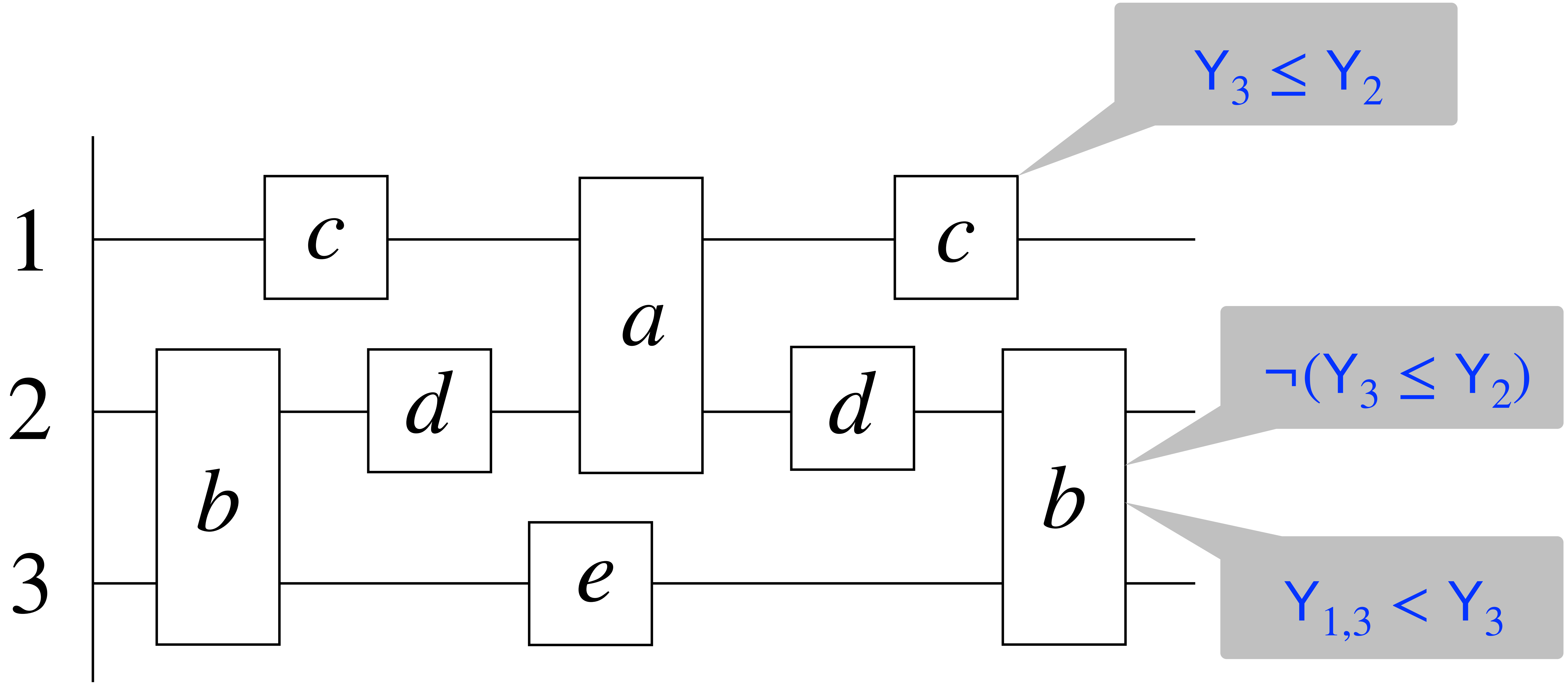
- $T \models \neg(\mathsf{L}_1 \leq \mathsf{L}_3)$



- Trace formulas / Sentences $\Phi ::= \mathsf{EM}_i\ \varphi \mid \Phi \vee \Phi \mid \neg\Phi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,k} \leq \mathsf{L}_j$
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,k} \leq \mathsf{Y}_j$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Extended locPastPDL for Traces

- $T \models \neg(L_1 \leq L_3)$
- $T \models L_{1,2} < L_3$



- Trace formulas / Sentences $\Phi ::= EM_i\ \varphi \mid \Phi \vee \Phi \mid \neg\Phi \mid L_i \leq L_j \mid L_{i,k} \leq L_j$
- State/Event formulas $\varphi ::= a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi \mid Y_i \leq Y_j \mid Y_{i,k} \leq Y_j$
- Program/Path expressions $\pi ::= \varphi? \mid \leftarrow_i \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$

Extended locPastPDL for Traces

Extended locPastPDL for Traces

Theorem (Adsul, Gastin, Sarkar, Weil — CONCUR'22)

- Extended locPastPDL is expressively complete for regular trace languages

Extended locPastPDL for Traces

Theorem (Adsul, Gastin, Sarkar, Weil — CONCUR'22)

- Extended locPastPDL is expressively complete for regular trace languages

Theorem (Mukund-Sohoni 1997)

There is an asynchronous letter-to-letter transducer $\mathscr{G}$ which computes the truth values of the constants from

$$\mathscr{Y} = \{\mathsf{Y}_i \leq \mathsf{Y}_j \, , \mathsf{Y}_{i,k} \leq \mathsf{Y}_j \ \mid \ i, j, k \in \mathscr{P}\}.$$

Extended locPastPDL for Traces

Theorem (Adsul, Gastin, Sarkar, Weil — CONCUR'22)

- Extended locPastPDL is expressively complete for regular trace languages
- Any regular trace language is accepted by a cascade product of the gossip transducer followed by a sequence of local asynchronous transducers:

$$\mathcal{G} \circ \mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$$

Theorem (Mukund-Sohoni 1997)

There is an asynchronous letter-to-letter transducer $\mathcal{G}$ which computes the truth values of the constants from

$$\mathcal{Y} = \{\mathsf{Y}_i \leq \mathsf{Y}_j\,, \mathsf{Y}_{i,k} \leq \mathsf{Y}_j \mid i, j, k \in \mathcal{P}\}.$$

Extended locPastPDL for Traces

Theorem (Adsul, Gastin, Sarkar, Weil — CONCUR'22)

- Extended locPastPDL is expressively complete for regular trace languag[e]
- Any regular trace language is accepted by a cascade product of th[e] transducer followed by a sequence of local asynchronous tra[...]

$$\mathscr{G} \circ \mathscr{T}_1 \circ \mathscr{T}_2 \circ \cdots \circ \mathscr{T}_n$$

Extended locPastPDL = locPastPDL

Theorem (Mukund-Sohoni 1997)

There is an asynchronous letter-to-letter trans[...] which computes the truth values of the constants from

$$\mathscr{Y} = \{\mathsf{Y}_i \leq \mathsf{Y}_j, \mathsf{Y}_{i,k} \leq \mathsf{Y}_j \mid i,j,k \in \mathscr{P}\}.$$

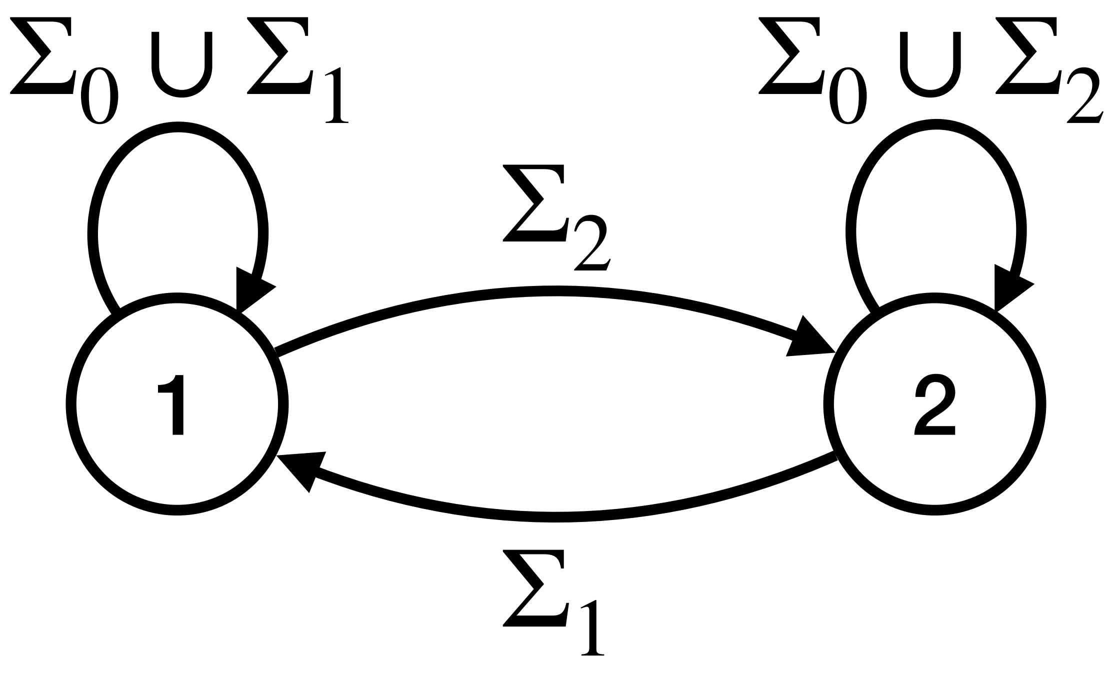
Aperiodic = FO-definable

Theorem [Adsul, Gastin, Sarkar, Weil — Concur'20, LMCS'22]

Any aperiodic (FO) trace language is accepted by a cascade product of the gossip transducer followed by a sequence of local reset transducers:

$$\mathcal{G} \circ \mathcal{U}_2 \circ \mathcal{U}_2 \circ \cdots \circ \mathcal{U}_2$$

Reset - $\mathcal{U}_2$



Aperiodic = FO-definable

Theorem [Adsul, Gastin, Sarkar, Weil — Concur'20, LMCS'22]

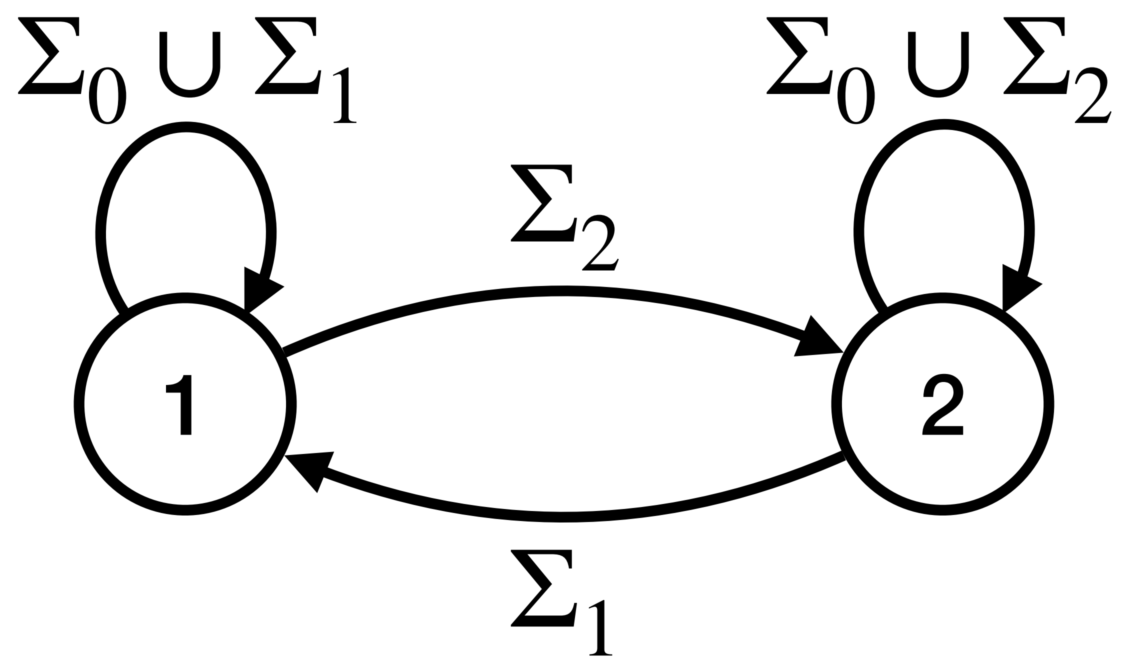
Any aperiodic (FO) trace language is accepted by a cascade product of the gossip transducer followed by a sequence of local reset transducers:

$$\mathcal{G} \circ \mathcal{U}_2 \circ \mathcal{U}_2 \circ \cdots \circ \mathcal{U}_2$$

Direct proof (Not using Krohn-Rhodes theorem)

based on a past temporal logic $\mathsf{LTL}\left(\mathsf{Y}_i \leq \mathsf{Y}_j, \mathsf{S}_i\right)$ proved expressively complete for FO

Reset - $\mathcal{U}_2$



Outline

- ✅ Labelling functions, sequential transducers and cascade product
- ✅ Krohn-Rhodes theorem for aperiodic/regular word languages
- ✅ Model of concurrency: Mazurkiewicz traces and asynchronous Zielonka automata
- ✅ Asynchronous labelling functions, transducers and cascade product
- ✅ Propositional dynamic logic for traces
- Conclusion

Conclusion

Conclusion

Main results

- (Local) (past) Propositional dynamic logic expressively complete for regular trace languages
  - Specification language: natural, easy, expressive, good complexity

Conclusion

Main results

- (Local) (past) Propositional dynamic logic expressively complete for regular trace languages
  - Specification language: natural, easy, expressive, good complexity
- Cascade decomposition using simple & local asynchronous automata/transducers
  - Allows inductive reasoning on automata

Conclusion

Main results

- (Local) (past) Propositional dynamic logic expressively complete for regular trace languages
  - ▷ Specification language: natural, easy, expressive, good complexity
- Cascade decomposition using simple & local asynchronous automata/transducers
  - ▷ Allows inductive reasoning on automata
- $\mathcal{U}_2 \circ \cdots \circ \mathcal{U}_2 = \mathsf{locTL}(\mathsf{SS}_i) \subseteq$ Aperiodic = FO = $\mathsf{locTL}(\mathsf{SS}_i, \mathsf{Y}_i \leq \mathsf{Y}_j) = \mathcal{G} \circ \mathcal{U}_2 \circ \cdots \circ \mathcal{U}_2$

Conclusion

Main results

- (Local) (past) Propositional dynamic logic expressively complete for regular trace languages
  - ▷ Specification language: natural, easy, expressive, good complexity
- Cascade decomposition using simple & local asynchronous automata/transducers
  - ▷ Allows inductive reasoning on automata
- $\mathscr{U}_2 \circ \cdots \circ \mathscr{U}_2 = \mathsf{locTL}(\mathsf{SS}_i) \subseteq$ Aperiodic = FO = $\mathsf{locTL}(\mathsf{SS}_i, \mathsf{Y}_i \leq \mathsf{Y}_j) = \mathscr{G} \circ \mathscr{U}_2 \circ \cdots \circ \mathscr{U}_2$
  - ▷ Equality for acyclic architectures (communication graph).

Conclusion

Main results

- (Local) (past) Propositional dynamic logic expressively complete for regular trace languages
  - Specification language: natural, easy, expressive, good complexity
- Cascade decomposition using simple & local asynchronous automata/transducers
  - Allows inductive reasoning on automata
- $\mathcal{U}_2 \circ \cdots \circ \mathcal{U}_2 = \mathsf{locTL}(\mathsf{SS}_i) \subseteq$ Aperiodic = FO = $\mathsf{locTL}(\mathsf{SS}_i, \mathsf{Y}_i \leq \mathsf{Y}_j) = \mathcal{G} \circ \mathcal{U}_2 \circ \cdots \circ \mathcal{U}_2$
  - Equality for acyclic architectures (communication graph).
  - Inclusion strict in general: gossip is (past) first-order definable, but cannot be computed with an aperiodic asynchronous transducer, hence also with a cascade of $\mathcal{U}_2$.

Conclusion

Main results

- (Local) (past) Propositional dynamic logic expressively complete for regular trace languages
  - Specification language: natural, easy, expressive, good complexity
- Cascade decomposition using simple & local asynchronous automata/transducers
  - Allows inductive reasoning on automata
- $\mathcal{U}_2 \circ \cdots \circ \mathcal{U}_2 = \mathrm{locTL}(\mathrm{SS}_i) \subseteq$ Aperiodic = FO = $\mathrm{locTL}(\mathrm{SS}_i, \mathrm{Y}_i \leq \mathrm{Y}_j) = \mathcal{G} \circ \mathcal{U}_2 \circ \cdots \circ \mathcal{U}_2$
  - Equality for acyclic architectures (communication graph).
  - Inclusion strict in general: gossip is (past) first-order definable, but cannot be computed with an aperiodic asynchronous transducer, hence also with a cascade of $\mathcal{U}_2$.

Open problem

- Generalisation to other structures, eg, Message sequence charts & Message passing automata?

Thank you for your attention!

Outline

- ✓ Labelling functions, sequential transducers and cascade product
- ✓ Krohn-Rhodes theorem for aperiodic/regular word languages
- ✓ Model of concurrency: Mazurkiewicz traces and asynchronous Zielonka automata
- ✓ Asynchronous labelling functions, transducers and cascade product
- ✓ Propositional dynamic logic for traces
- ✓ Conclusion