Synthesis Without State Explosion

from concurrent processes to netlists

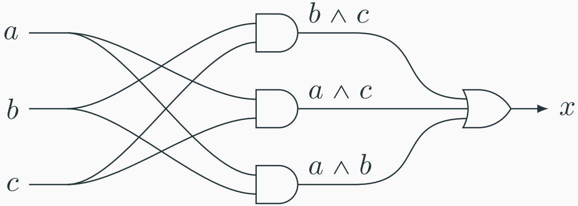
Dennis Furey

18 November 2019

Plumstead Publishing

A logic puzzle

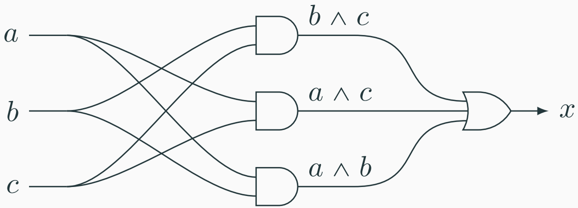
Build a three-way voting machine.



$$x \equiv (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

A logic puzzle

Build a three-way voting machine.

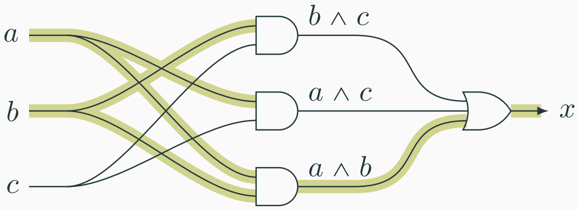


$$x \equiv (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

- presupposes a 4Φ (return to zero) protocol
- but x could drop prematurely

A logic puzzle

Build a three-way voting machine.

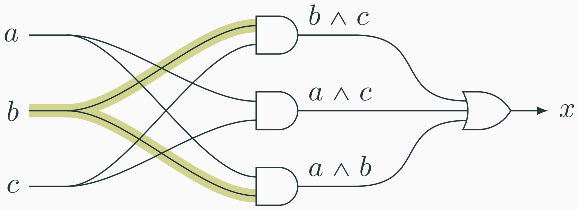


$$x \equiv (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

- presupposes a 4Φ (return to zero) protocol
- but x could drop prematurely

A logic puzzle

Build a three-way voting machine.

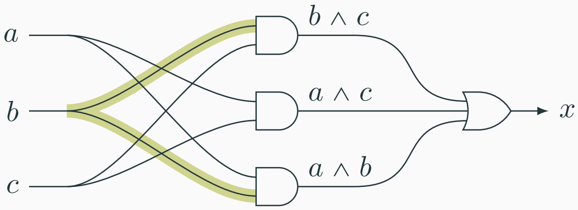


$$x \equiv (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

- presupposes a 4Φ (return to zero) protocol
- but x could drop prematurely

A logic puzzle

Build a three-way voting machine.

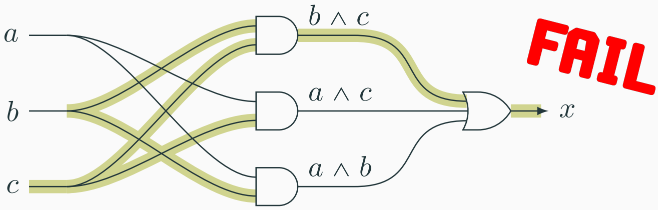


$$x \equiv (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

- presupposes a 4Φ (return to zero) protocol
- but x could drop prematurely

A logic puzzle

Build a three-way voting machine.



$$x \equiv (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

- presupposes a 4Φ (return to zero) protocol
- but x could drop prematurely

To maintain the output until both inputs drop, use a JOIN.

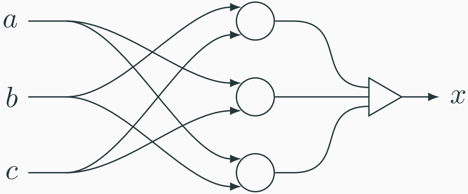
To maintain the output until both inputs drop, use a JOIN.

To maintain the output until both inputs drop, use a JOIN.

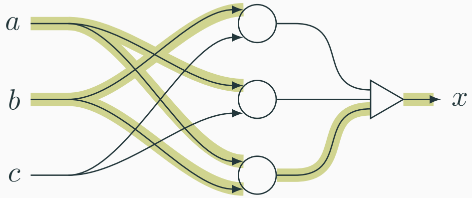
To maintain the output until both inputs drop, use a JOIN.

To maintain the output until both inputs drop, use a JOIN.

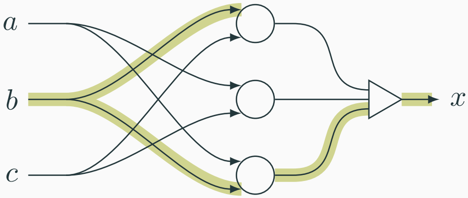
To maintain the output until both inputs drop, use a JOIN. It can be 2Φ or 4Φ with a MERGE instead of an OR.



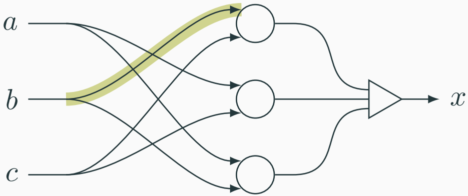
To maintain the output until both inputs drop, use a JOIN. It can be 2Φ or 4Φ with a MERGE instead of an OR.



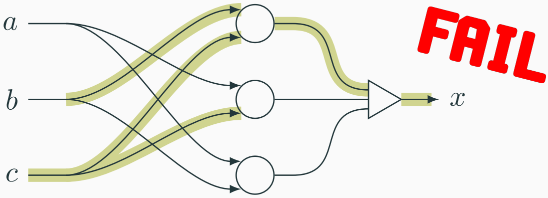
To maintain the output until both inputs drop, use a JOIN. It can be 2Φ or 4Φ with a MERGE instead of an OR.



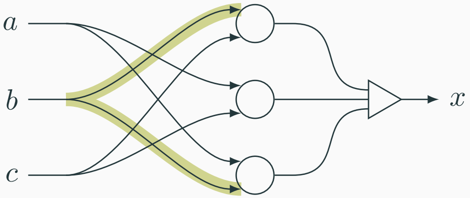
To maintain the output until both inputs drop, use a JOIN. It can be 2Φ or 4Φ with a MERGE instead of an OR.



To maintain the output until both inputs drop, use a JOIN. It can be 2Φ or 4Φ with a MERGE instead of an OR.



To maintain the output until both inputs drop, use a JOIN. It can be 2Φ or 4Φ with a MERGE instead of an OR.



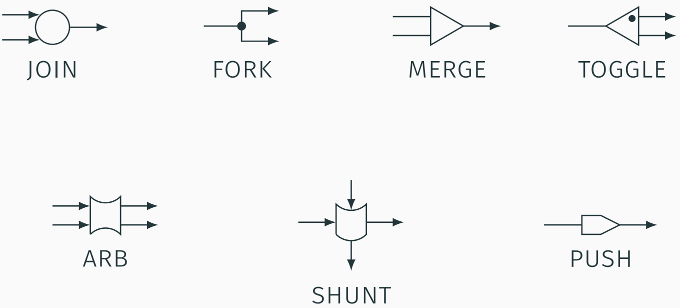
isochronic forks to the rescue?

General observations

- There is no way to fix these circuits without cheating.
- “QDI” designers insist on isochronic forks as a workaround.
- Logic gates might not be the best choice of primitives.
- Isochronic forks can be avoided with different primitives.

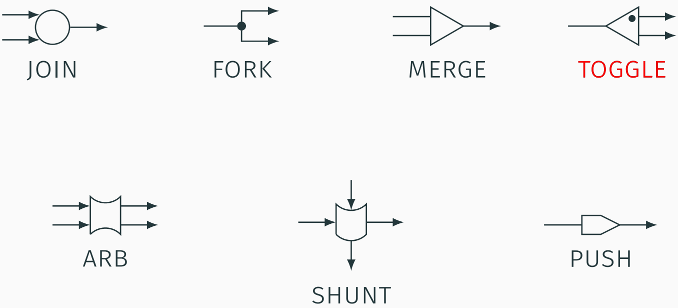
The gateless gate

Delay insensitive primitives



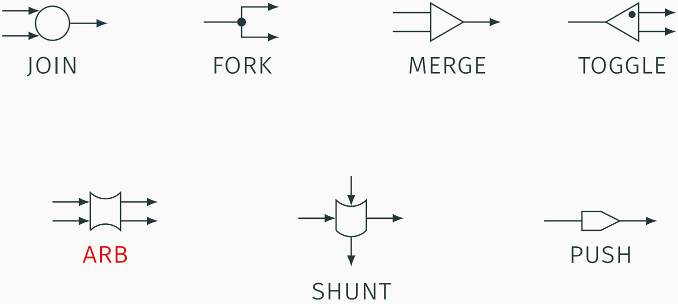
- only seven needed for universality
- no more than four terminals each

Delay insensitive primitives



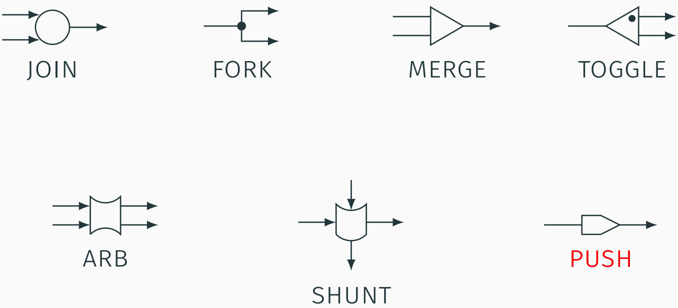
- only seven needed for universality
- no more than four terminals each

Delay insensitive primitives



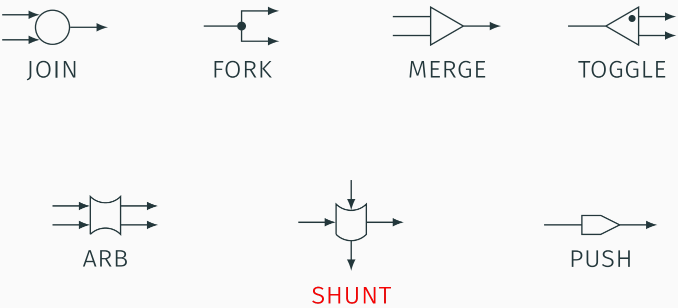
- only seven needed for universality
- no more than four terminals each

Delay insensitive primitives



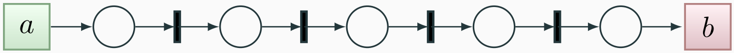
- only seven needed for universality
- no more than four terminals each

Delay insensitive primitives

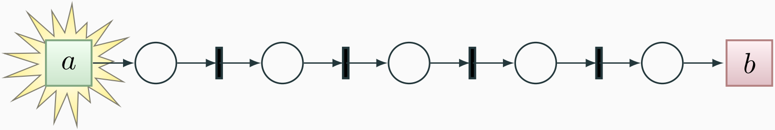


- only seven needed for universality
- no more than four terminals each

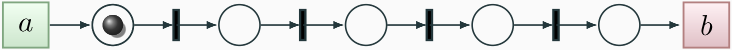
Petri nets



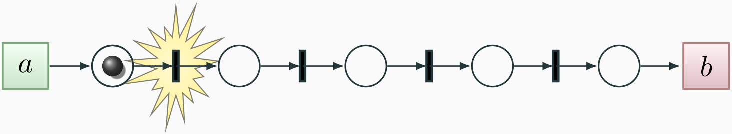
Petri nets



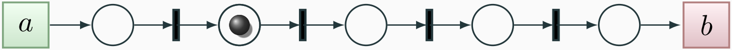
Petri nets



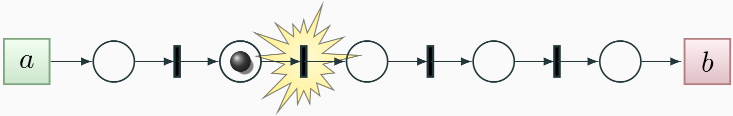
Petri nets



Petri nets

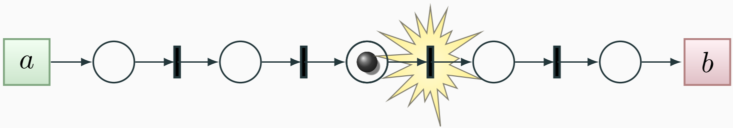


Petri nets

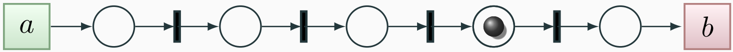


Petri nets

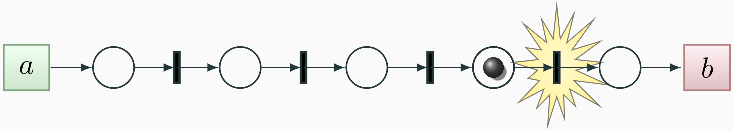
Petri nets



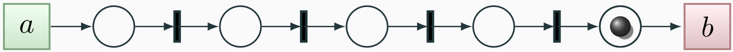
Petri nets



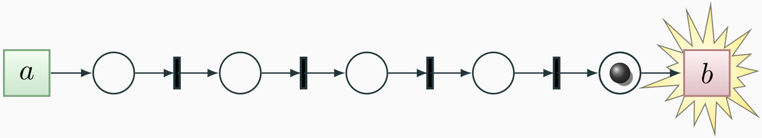
Petri nets



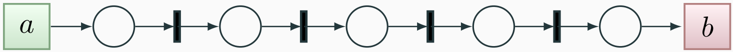
Petri nets



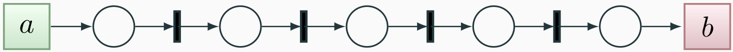
Petri nets



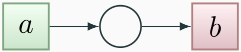
Petri nets



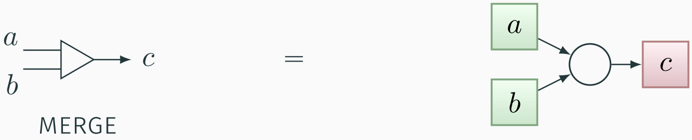
Petri nets



Local optimizations preserving observable behavior are a bonus.

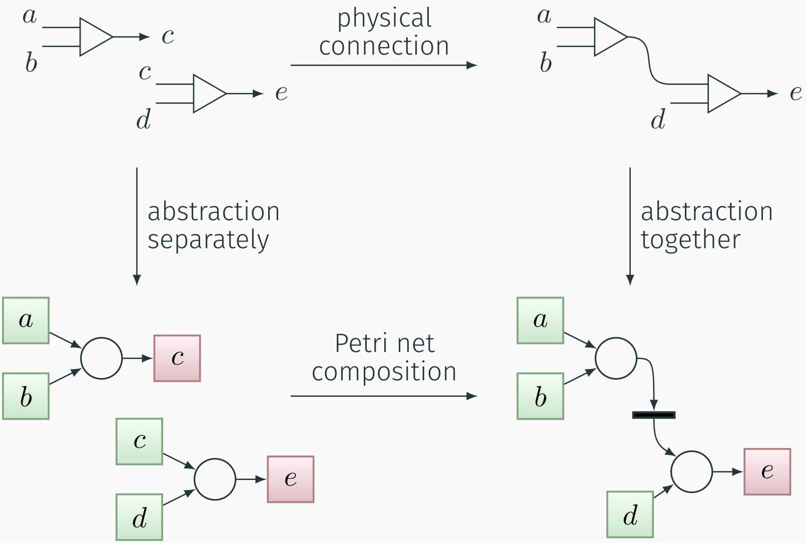


Petri net models of DI primitives



- an open input transition for each input terminal
- an open output transition for each output terminal
- internal places and transitions to constrain firing sequences

Compositionality

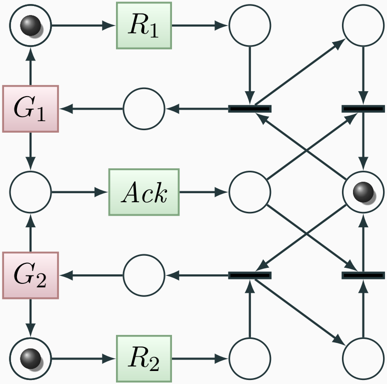


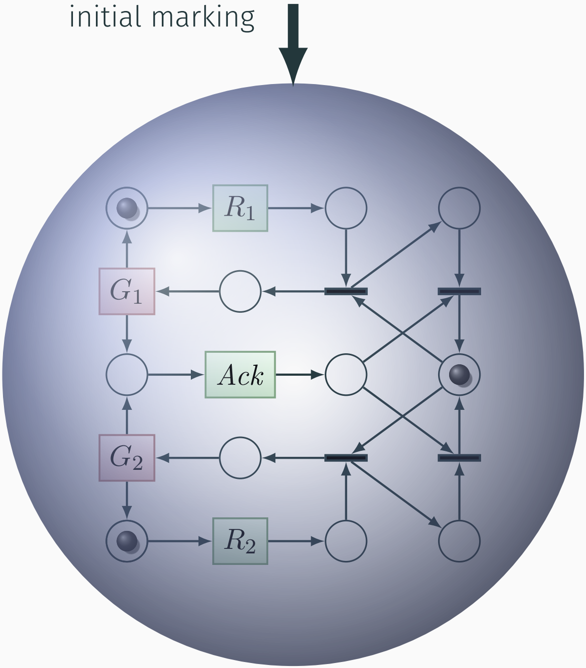
Benefits

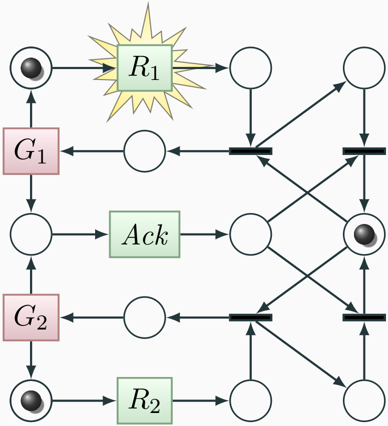
- Petri Net models combine just like components in a circuit.
- The Petri net model is proportional in size to the circuit.
- Things that look like they should be true *are* true !
- and more . . .

Analysis

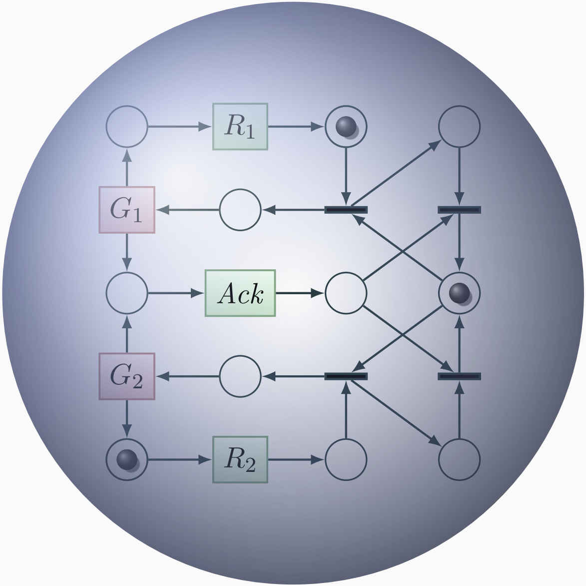
Markings



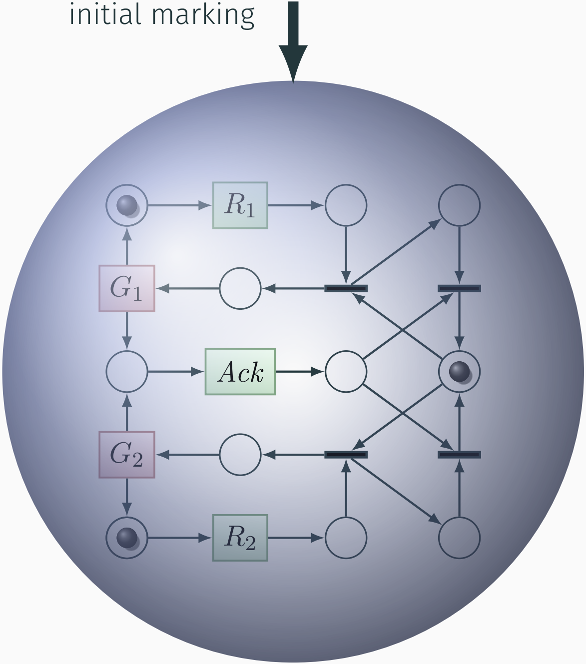

initial marking
R_1
G_1
Ack
G_2
R_2

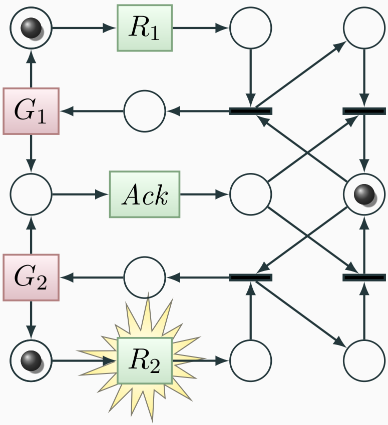

R_1
G_1
Ack
G_2
R_2

new marking



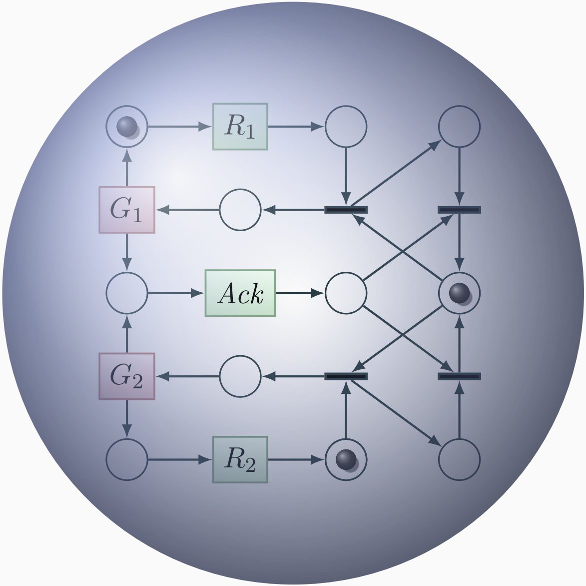
initial marking

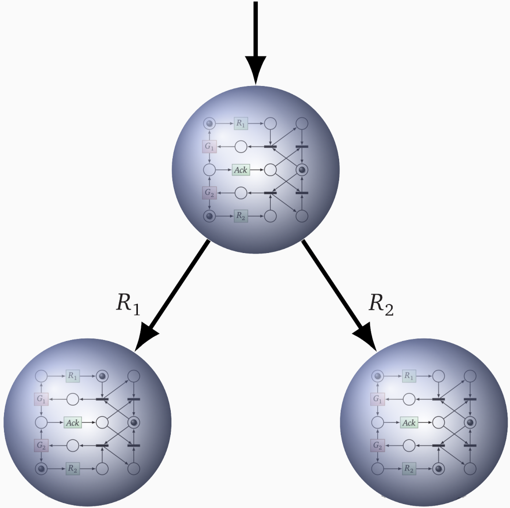


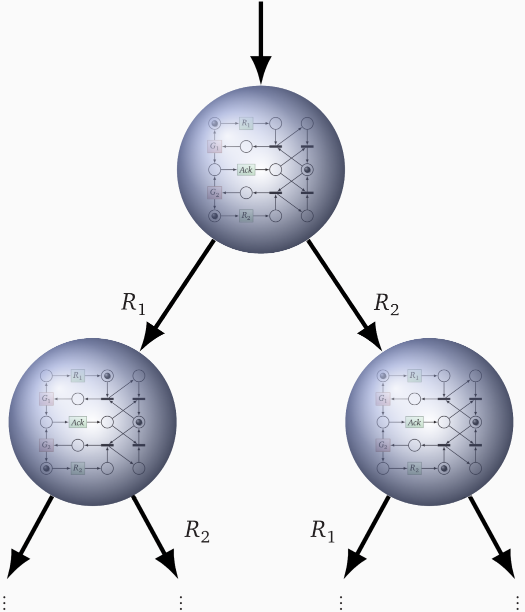


R_1
G_1
Ack
G_2
R_2

another new marking

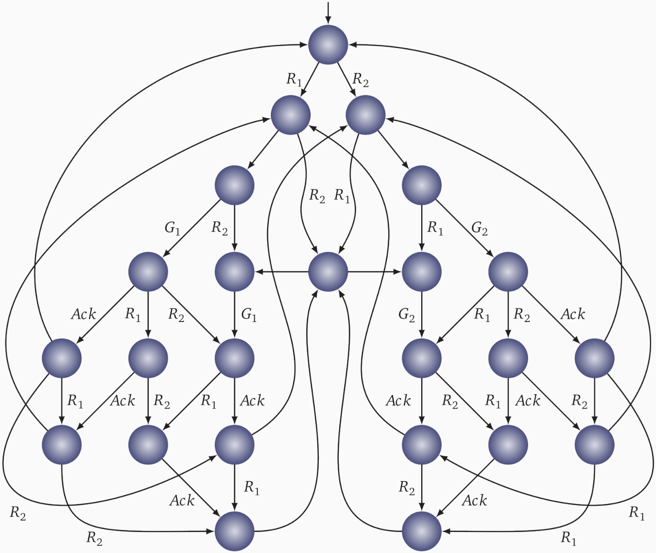



R1
R2
G1
Ack
G2

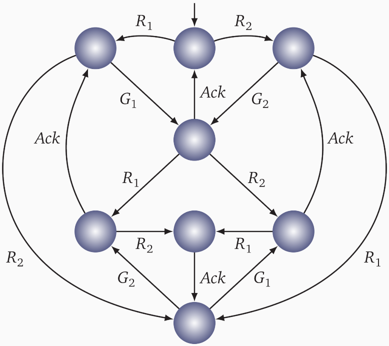


R_1
G_1
Ack
G_2
R_2
R_1
R_2
R_2
R_1

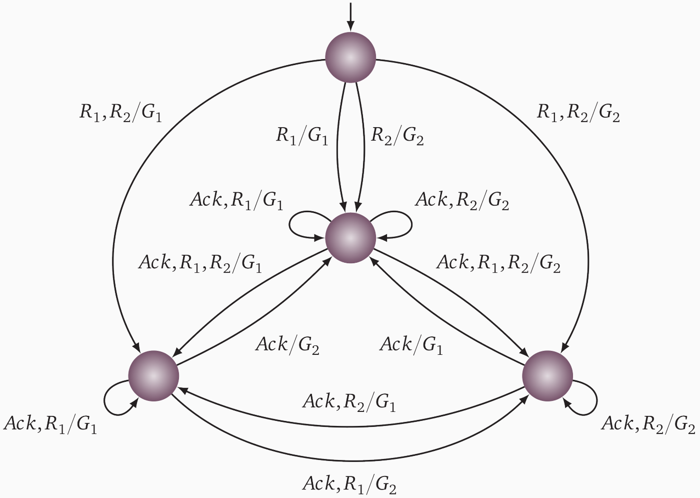
Reachability graphs

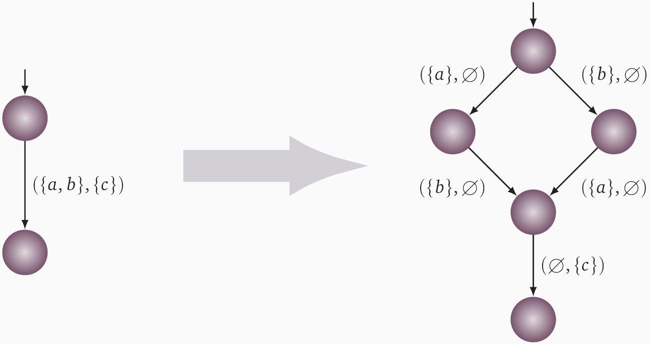


Reachability graphs

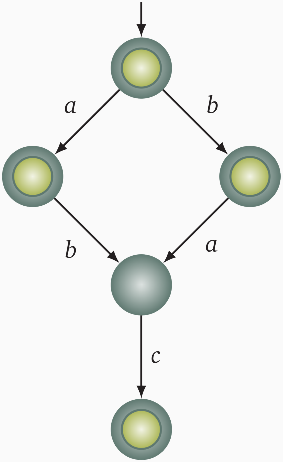


Transducers

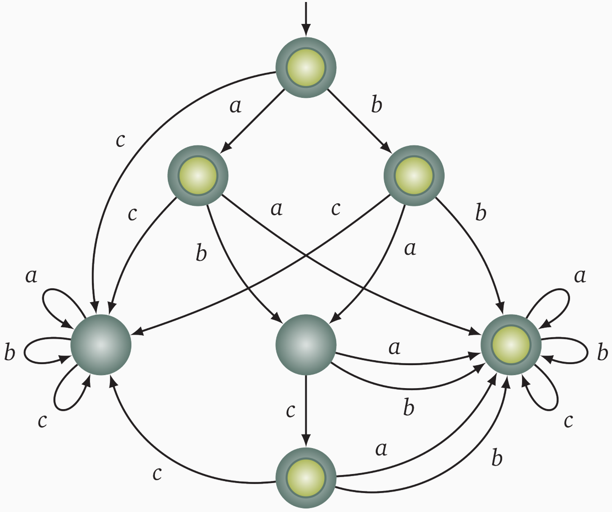




$(\{a,b\},\{c\})$
$(\{a\},\varnothing)$
$(\{b\},\varnothing)$
$(\{b\},\varnothing)$
$(\{a\},\varnothing)$
$(\varnothing,\{c\})$


a
b
b
a
c

Relational trace set recognizer



accepts quiescent and divergent traces only

Equivalence and refinement

For processes X and Y with identical alphabets and relational trace sets $[\![X]\!]$ and $[\![Y]\!]$

behavioral equivalence (processes are indistinguishable)

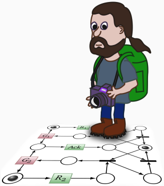
$$X \equiv Y \Leftrightarrow [\![X]\!] = [\![Y]\!]$$

refinement (Y is as good as X or better)

$$X \sqsubseteq Y \Leftrightarrow [\![X]\!] \supseteq [\![Y]\!]$$

A correct implementation refines its specification.

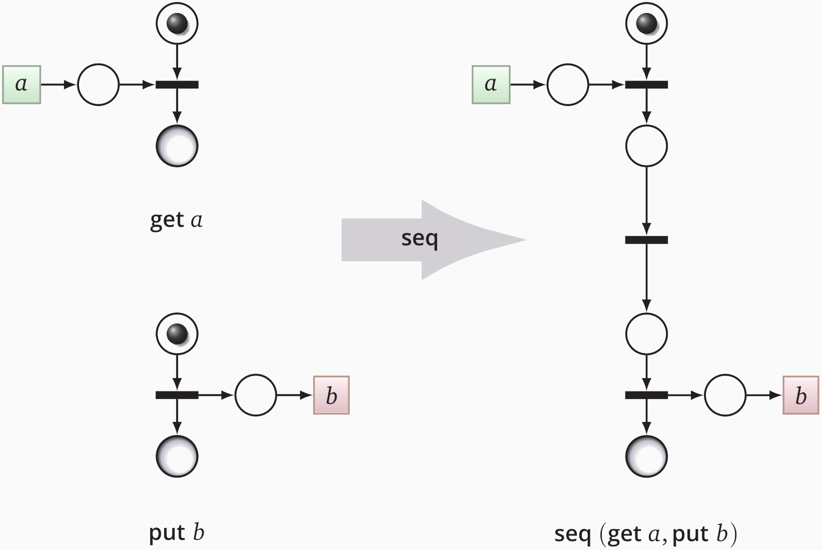
Specifications

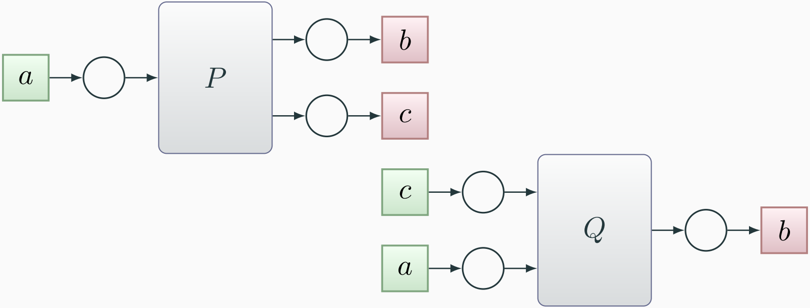


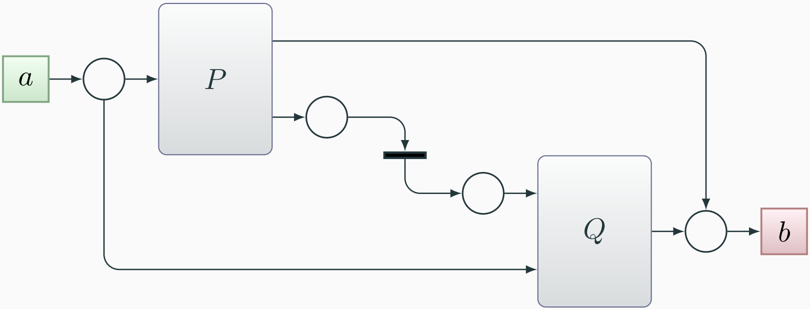
Process combinators

for terminals $\mathbb{T}$ and Petri-net-modeled DI processes $\mathbb{D}$

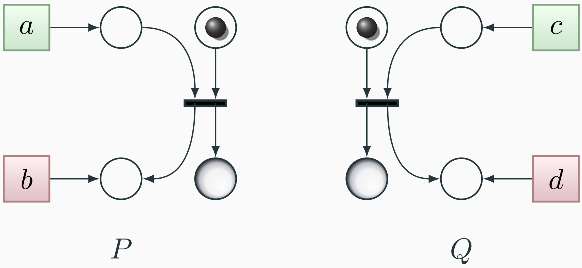
Type	Mnemonic	Description
$\mathbb{T} \to \mathbb{D}$	**get**	receive a signal
	put	send a signal
$(\mathbb{D} \times \mathbb{D}) \to \mathbb{D}$	**seq**	do one after the other
	par	do both concurrently
	alt	do either but not both
	env	do only what's needed to interact
$(\mathbb{D} \to \mathbb{D}) \to \mathbb{D}$	**fix**	act as the solution to a recurrence

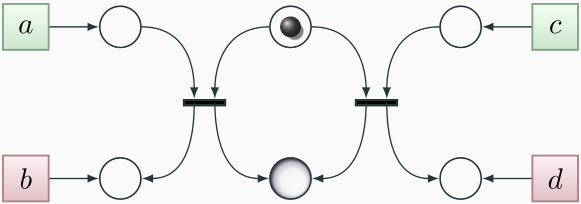

a
get a
seq
a
put b
b
b
seq (get a, put b)


a
P
b
c
c
a
Q
b



par (P, Q)


a
b
c
d
P
Q



alt (P, Q)

Process examples

repetition

$$loop = \lambda p.\ \mathsf{fix}\ \lambda f.\ \mathsf{seq}\ (p, f)$$

synchronization

$$j = loop\ \mathsf{seq}\ (\mathsf{par}\ (\mathsf{get}\ a, \mathsf{get}\ b), \mathsf{put}\ c)$$

4Φ handshake

$$h(a, b) = \mathsf{seq}\ (\mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b), \mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b))$$

arbitration

$$m = loop\ \mathsf{alt}\ (h(r_0, g_0), h(r_1, g_1))$$

Process examples

repetition

$$loop = \lambda p.\ \mathsf{fix}\ \lambda f.\ \mathsf{seq}\ (p, f)$$

synchronization

$$j = loop\ \mathsf{seq}\ (\mathsf{par}\ (\mathsf{get}\ a, \mathsf{get}\ b), \mathsf{put}\ c)$$

4Φ handshake

$$h(a, b) = \mathsf{seq}\ (\mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b), \mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b))$$

arbitration

$$m = loop\ \mathsf{alt}\ (h(r_0, g_0), h(r_1, g_1))$$

Process examples

repetition

$$loop = \lambda p.\ \mathsf{fix}\ \lambda f.\ \mathsf{seq}\ (p, f)$$

synchronization

$$j = loop\ \mathsf{seq}\ (\mathsf{par}\ (\mathsf{get}\ a, \mathsf{get}\ b), \mathsf{put}\ c)$$

4Φ handshake

$$h(a, b) = \mathsf{seq}\ (\mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b), \mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b))$$

arbitration

$$m = loop\ \mathsf{alt}\ (h(r_0, g_0), h(r_1, g_1))$$

Process examples

repetition

$$loop = \lambda p.\ \mathsf{fix}\ \lambda f.\ \mathsf{seq}\ (p, f)$$

synchronization

$$j = loop\ \mathsf{seq}\ (\mathsf{par}\ (\mathsf{get}\ a, \mathsf{get}\ b), \mathsf{put}\ c)$$

4Φ handshake

$$h(a, b) = \mathsf{seq}\ (\mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b), \mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b))$$

arbitration

$$m = loop\ \mathsf{alt}\ (h(r_0, g_0), h(r_1, g_1))$$

Process examples

repetition

$$loop = \lambda p.\ \mathsf{fix}\ \lambda f.\ \mathsf{seq}\ (p, f)$$

synchronization

$$j = loop\ \mathsf{seq}\ (\mathsf{par}\ (\mathsf{get}\ a, \mathsf{get}\ b), \mathsf{put}\ c)$$

4Φ handshake

$$h(a, b) = \mathsf{seq}\ (\mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b), \mathsf{seq}\ (\mathsf{get}\ a, \mathsf{put}\ b))$$

arbitration

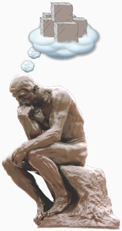
$$m = loop\ \mathsf{alt}\ (h(r_0, g_0), h(r_1, g_1))$$

nacking arbiter

$$
\begin{aligned}
n = \;& loop\ (\digamma\ \mathsf{alt})\ (\digamma\ \mathsf{seq})^{*} \langle \\
& \quad \langle \mathsf{get}\ r_0, \mathsf{put}\ g_0, \mathsf{get}\ r_0, \mathsf{put}\ g_0 \rangle, \\
& \quad \langle \mathsf{get}\ r_0, \mathsf{put}\ d_0 \rangle, \\
& \quad \langle \mathsf{get}\ r_1, \mathsf{put}\ g_1, \mathsf{get}\ r_1, \mathsf{put}\ g_1 \rangle, \\
& \quad \langle \mathsf{get}\ r_1, \mathsf{put}\ d_1 \rangle \rangle
\end{aligned}
$$

- 4Φ grant cycles
- 2Φ deny cycles
- functional programming operators (fold, map, lists)

Building blocks

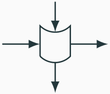


Shunt primitive

The SHUNT primitive has two modes of operation.

- normal mode

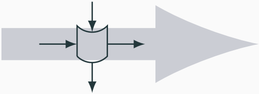
- shunting mode



Shunt primitive

The SHUNT primitive has two modes of operation.

- normal mode – can either
  - relay left to right and stay normal

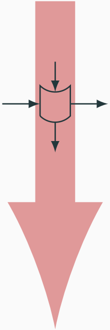


- shunting mode

Shunt primitive

The SHUNT primitive has two modes of operation.

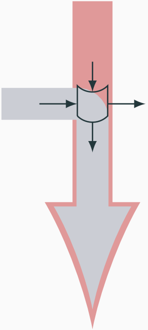
- normal mode – can either
  - relay left to right and stay normal
  - relay top to bottom and change modes
- shunting mode



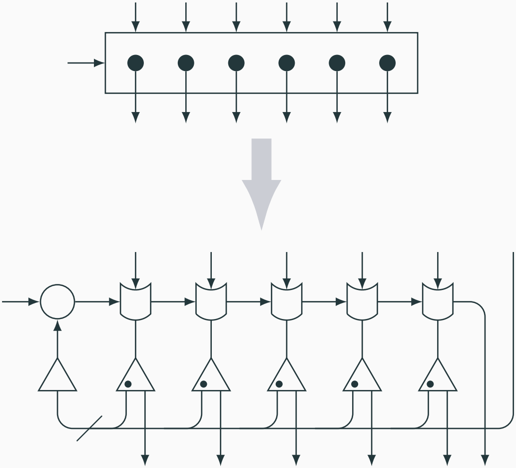
Shunt primitive

The SHUNT primitive has two modes of operation.

- normal mode – can either
  - relay left to right and stay normal
  - relay top to bottom and change modes
- shunting mode – must always
  - relay left to bottom exactly once
  - change back to normal mode

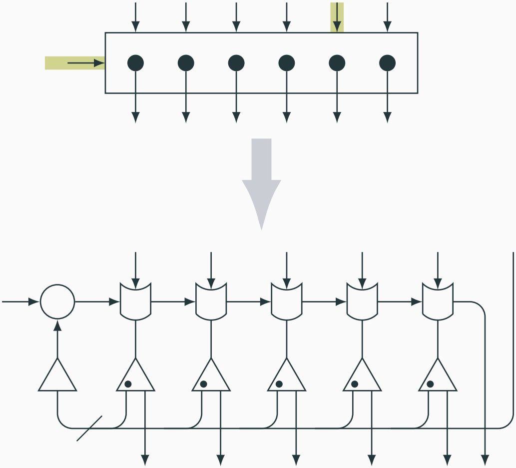


Decision waits



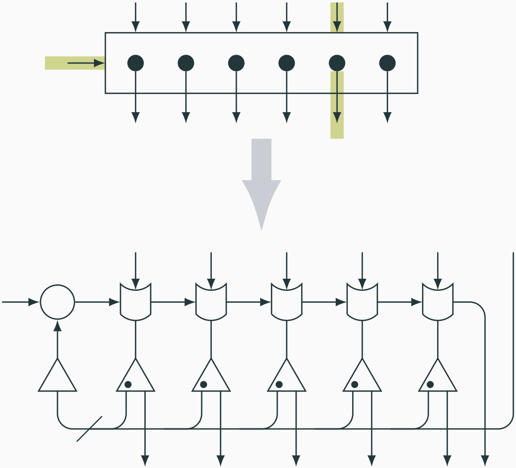
one row and multiple columns

Decision waits



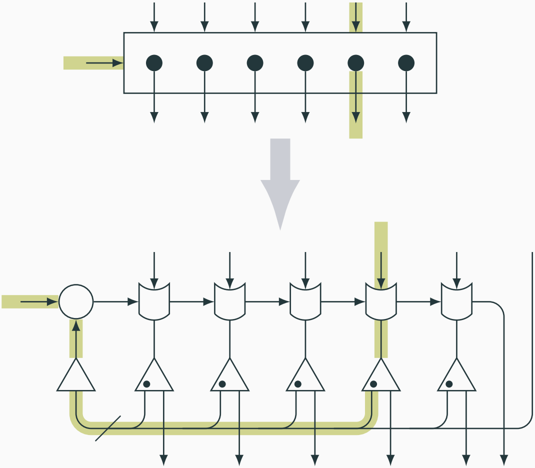
one row and multiple columns

Decision waits



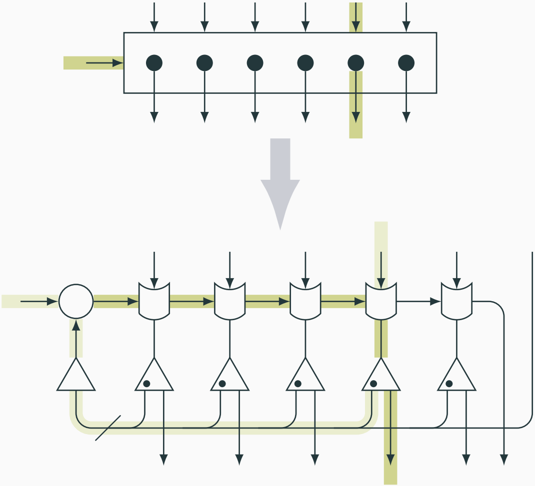
one row and multiple columns

Decision waits



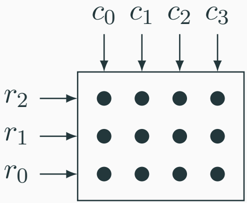
one row and multiple columns

Decision waits

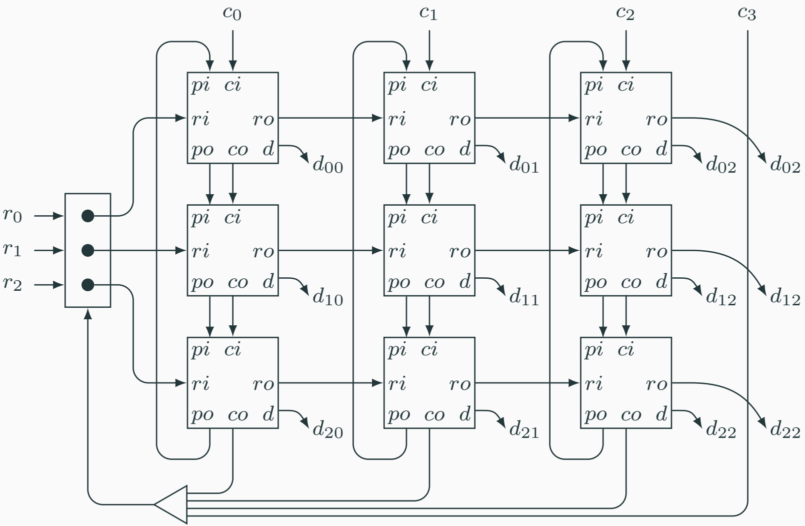


one row and multiple columns

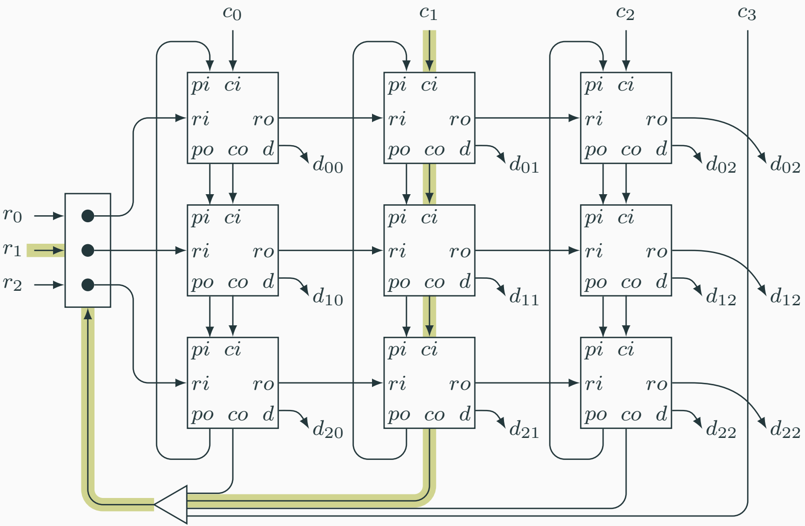
multiple rows and columns



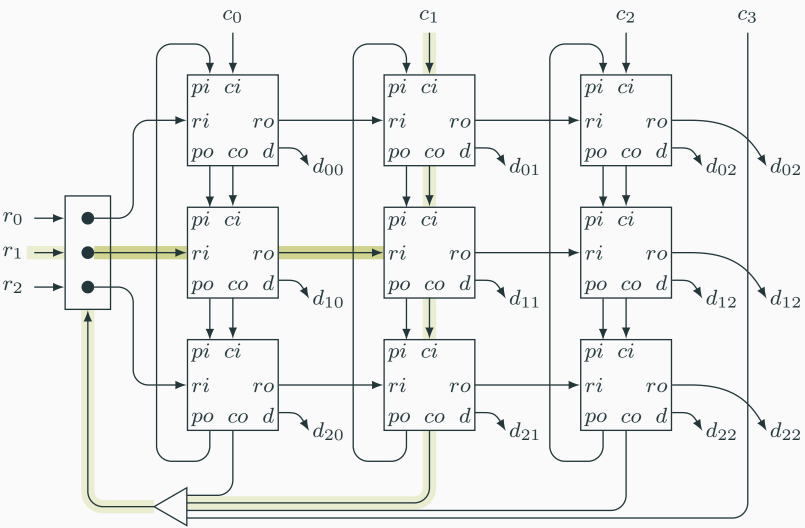
multiple rows and columns



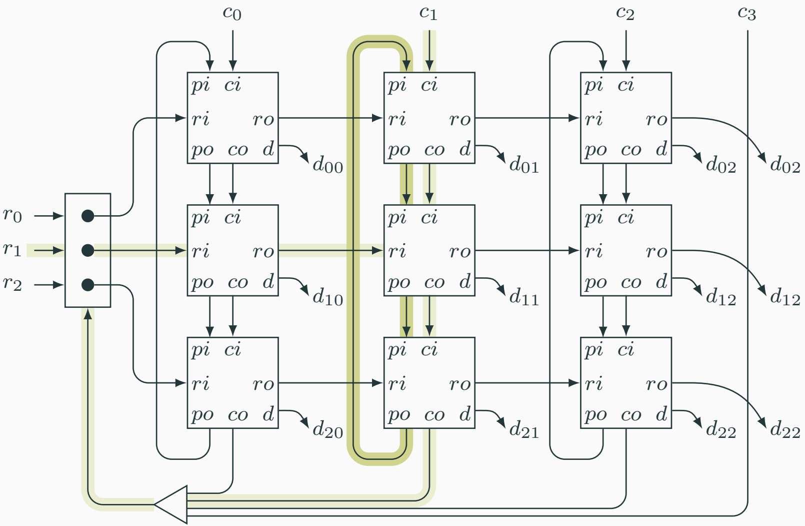
multiple rows and columns



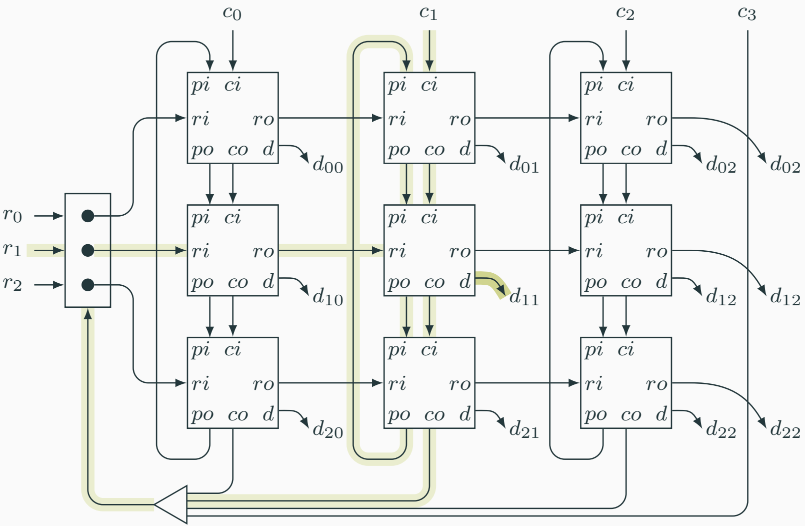
multiple rows and columns



multiple rows and columns

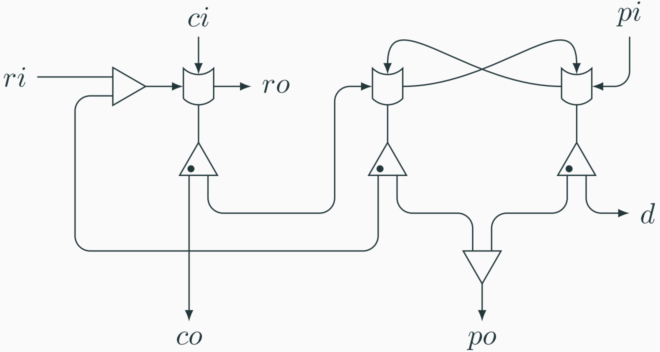


multiple rows and columns



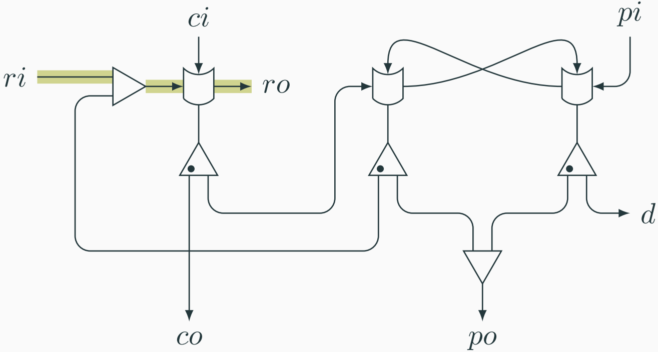
one cell, three possible handshakes

- $ri \to ro$
- $ci \to co \to pi \to po$
- $ci \to co \to ri \to po \to pi \to d$



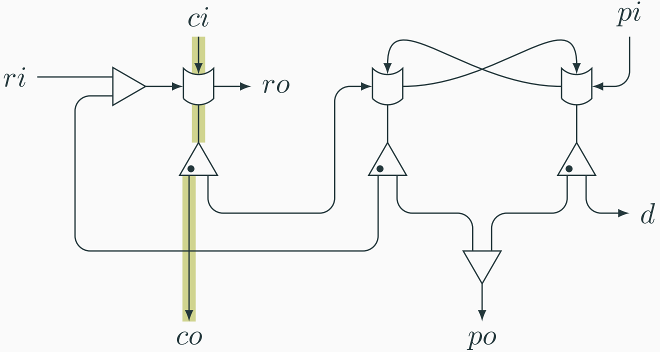
one cell, three possible handshakes

- $ri \to ro$
- $ci \to co \to pi \to po$
- $ci \to co \to ri \to po \to pi \to d$



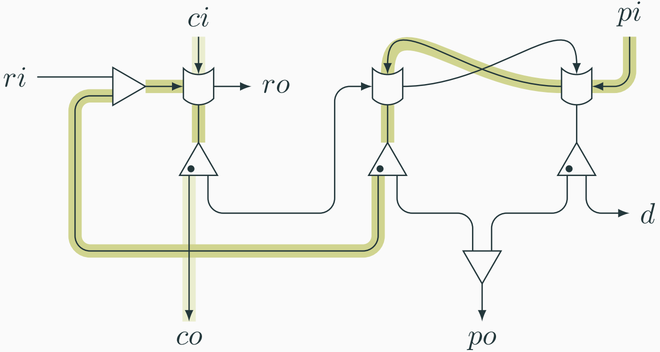
one cell, three possible handshakes

- $ri \to ro$
- $ci \to co \to pi \to po$
- $ci \to co \to ri \to po \to pi \to d$



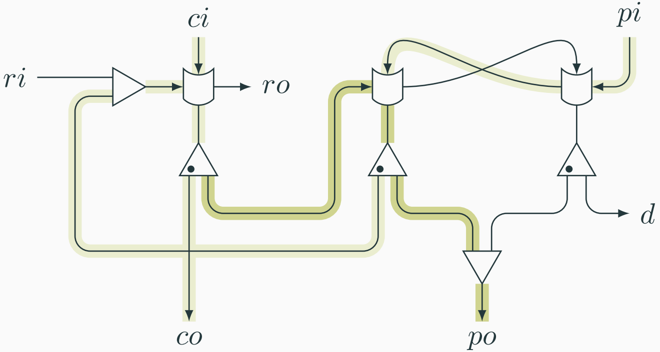
one cell, three possible handshakes

- $ri \to ro$
- $ci \to co \to pi \to po$
- $ci \to co \to ri \to po \to pi \to d$



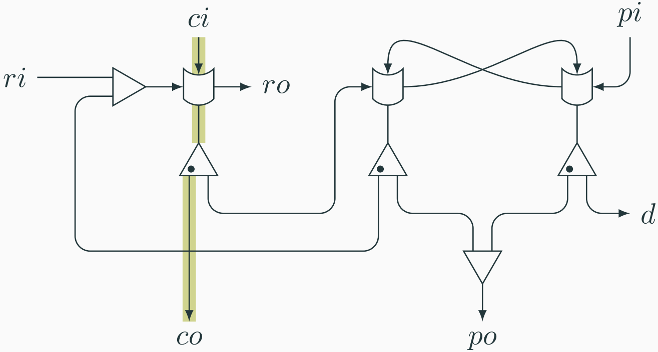
one cell, three possible handshakes

- $ri \to ro$
- $ci \to co \to pi \to po$
- $ci \to co \to ri \to po \to pi \to d$



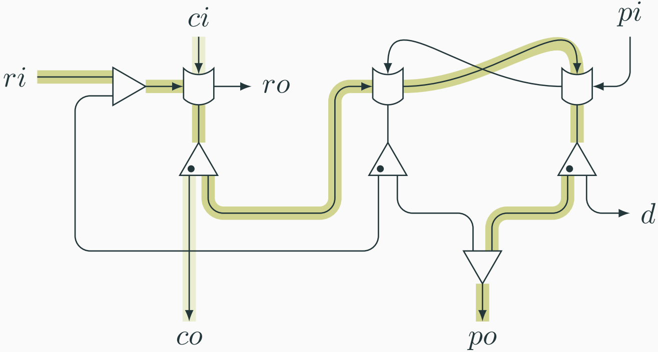
one cell, three possible handshakes

- $ri \to ro$
- $ci \to co \to pi \to po$
- $ci \to co \to ri \to po \to pi \to d$



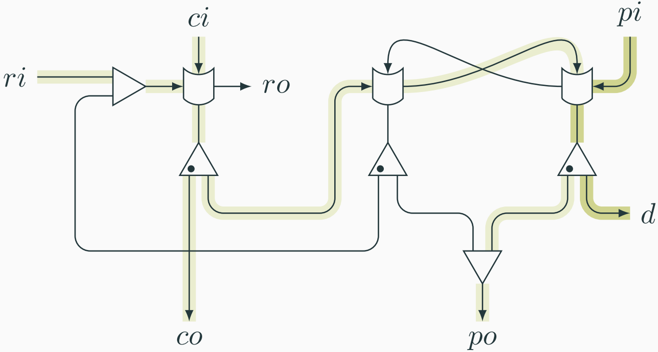
one cell, three possible handshakes

- $ri \to ro$
- $ci \to co \to pi \to po$
- $ci \to co \to ri \to po \to pi \to d$

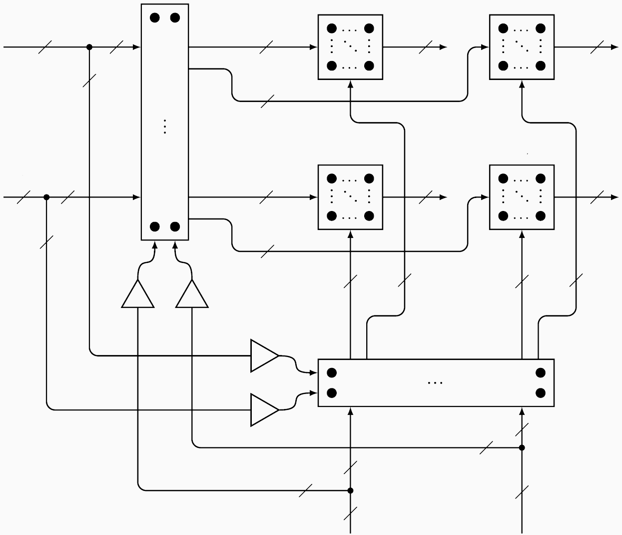


one cell, three possible handshakes

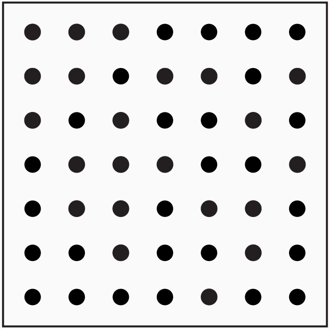
- $ri \to ro$
- $ci \to co \to pi \to po$
- $ci \to co \to ri \to po \to pi \to d$



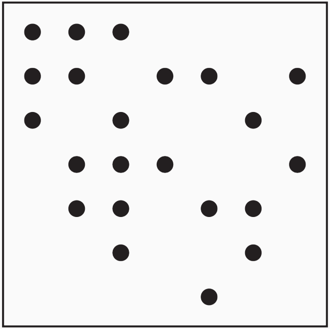
hierarchical decomposition



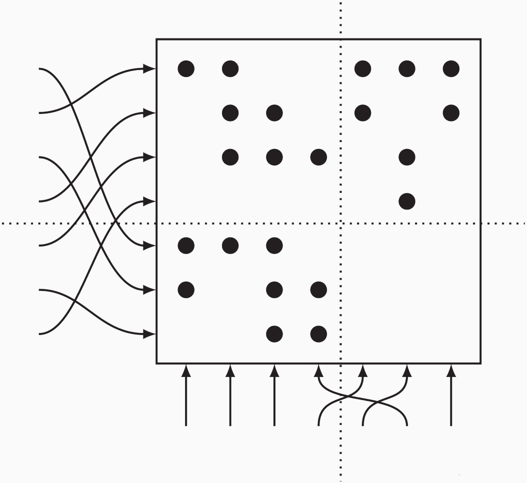
Sparse decision waits



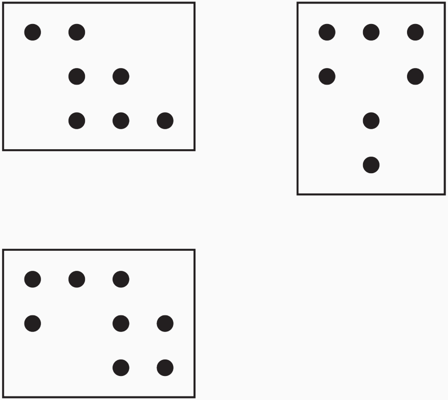
Sparse decision waits



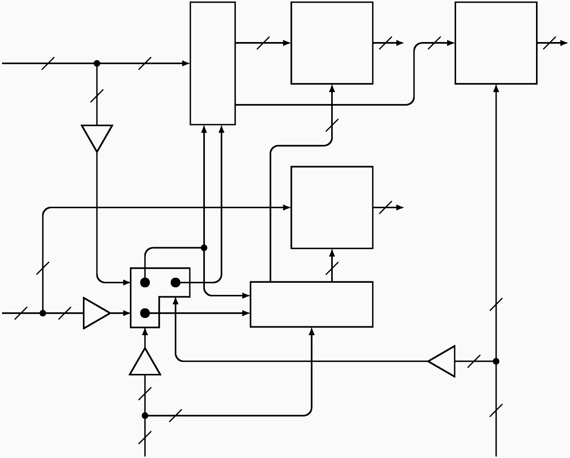
Sparse decision waits



Sparse decision waits

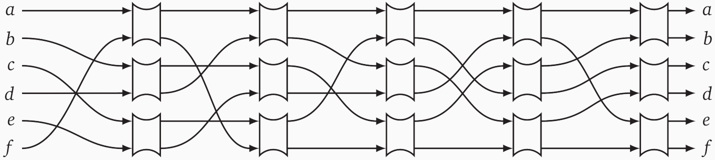


Sparse decision waits



Arbiters

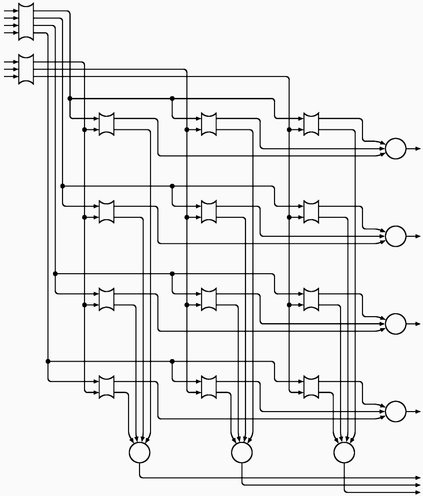
mesh



quadratic space, linear time

Arbiters

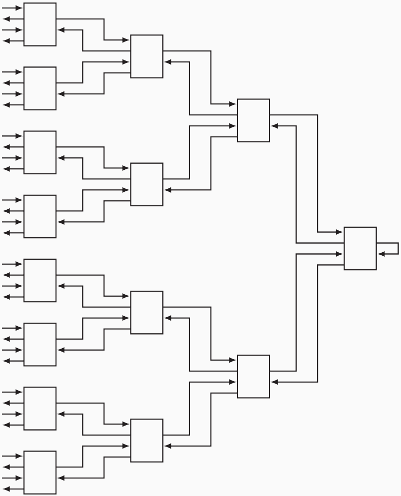
crossbar



quadratic space, $\log^2$ time

Arbiters

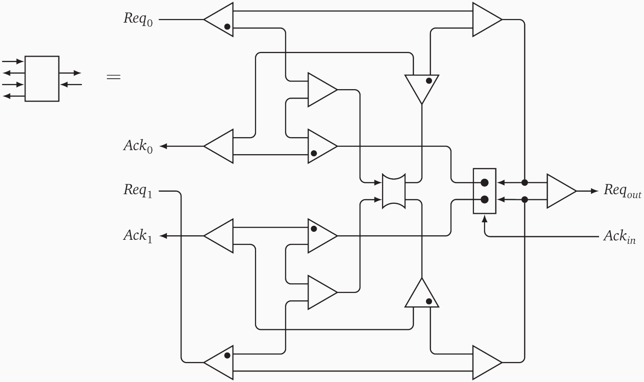
tree



linear space, logarithmic time

Arbiters

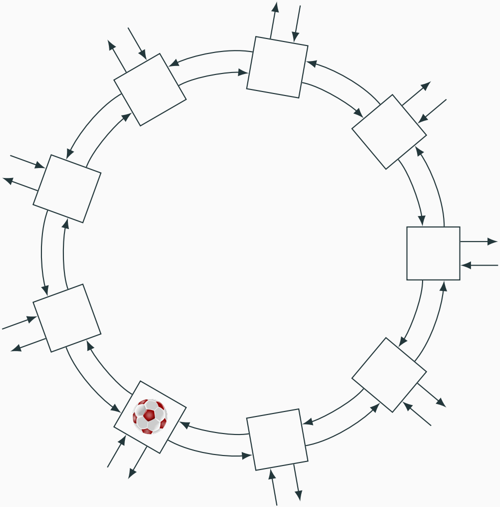
tree



linear space, logarithmic time

Arbiters

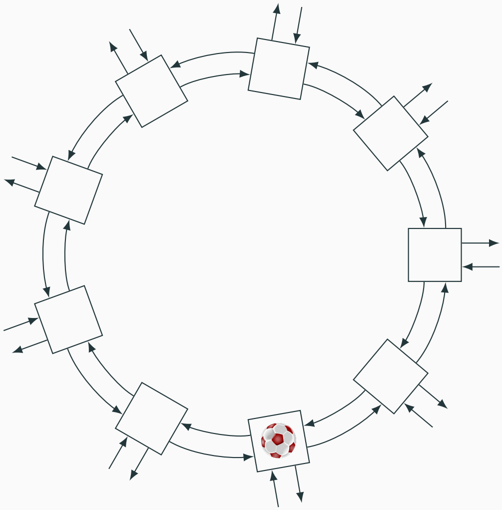
token ring



linear space, linear time (sort of)

Arbiters

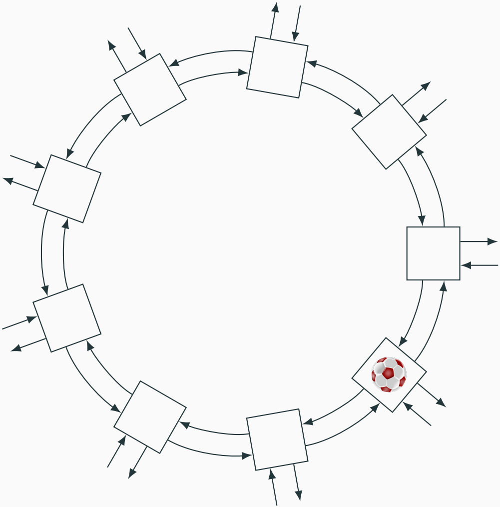
token ring



linear space, linear time (sort of)

Arbiters

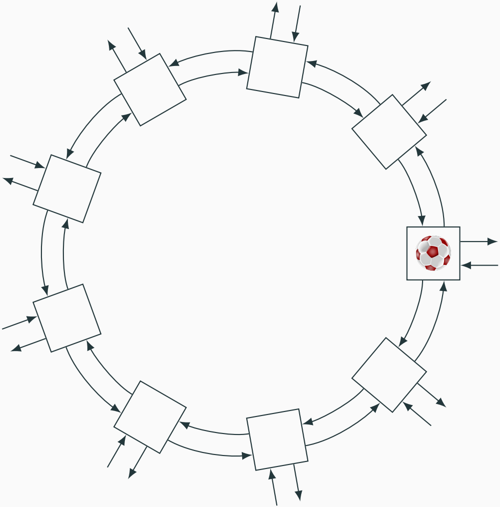
token ring



linear space, linear time (sort of)

Arbiters

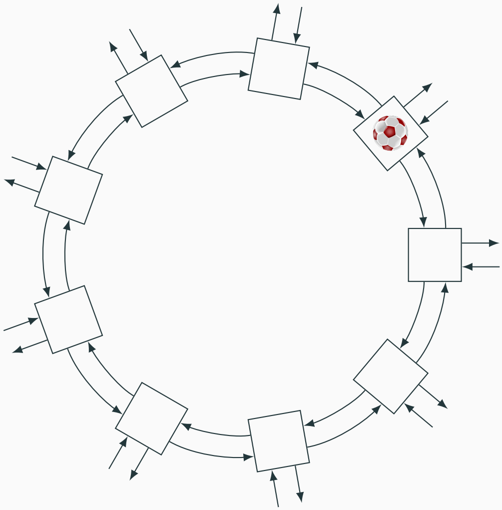
token ring



linear space, linear time (sort of)

Arbiters

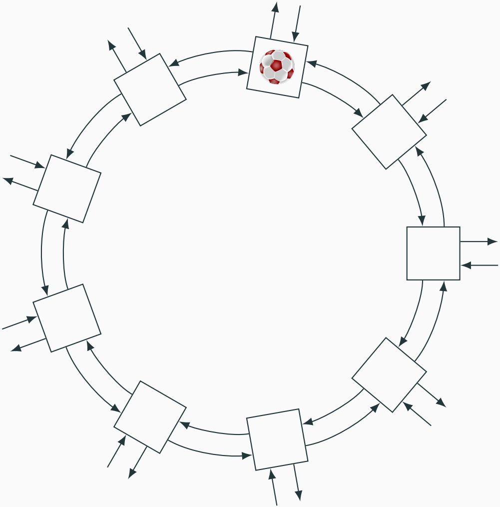
token ring



linear space, linear time (sort of)

Arbiters

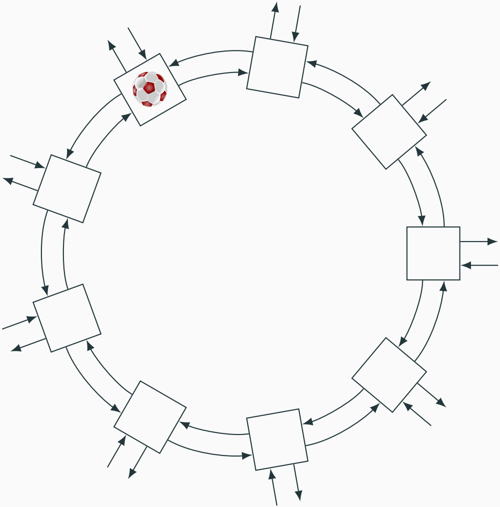
token ring



linear space, linear time (sort of)

Arbiters

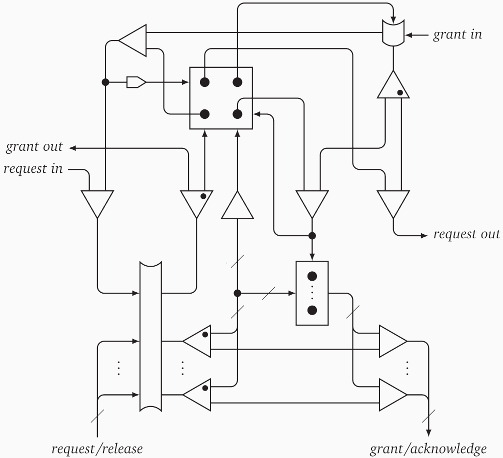
token ring



linear space, linear time (sort of)

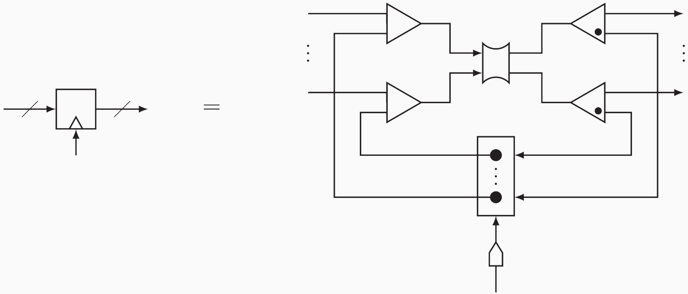
Arbiters

token ring



linear space, linear time (sort of)

Sequencers



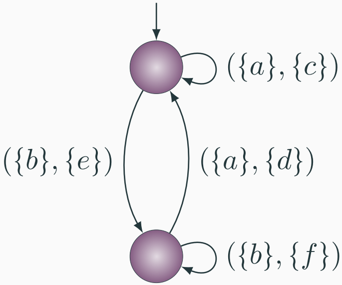
an arbiter with 2Φ ports and a shared acknowledgment terminal

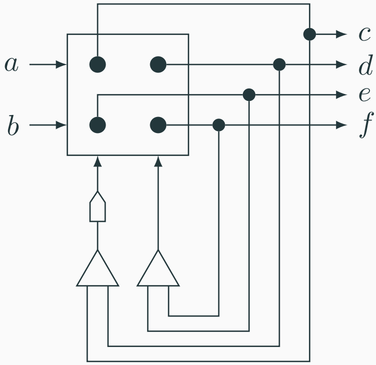
State Based Synthesis

A two-state solution

$$P_0 \equiv \mathsf{alt}\ \big((F\ \mathsf{seq})\,\langle \mathsf{get}\ a, \mathsf{put}\ c, P_0\rangle, (F\ \mathsf{seq})\,\langle \mathsf{get}\ b, \mathsf{put}\ e, P_1\rangle\big)$$

$$P_1 \equiv \mathsf{alt}\ \big((F\ \mathsf{seq})\,\langle \mathsf{get}\ b, \mathsf{put}\ f, P_1\rangle, (F\ \mathsf{seq})\,\langle \mathsf{get}\ a, \mathsf{put}\ d, P_0\rangle\big)$$

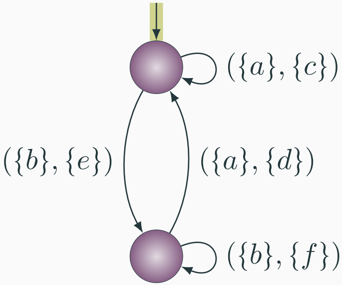


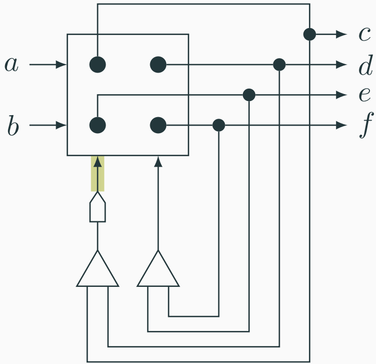


A two-state solution

$$P_0 \equiv \text{alt}\ \big((\digamma\ \text{seq})\,\langle \text{get}\ a, \text{put}\ c, P_0\rangle, (\digamma\ \text{seq})\,\langle \text{get}\ b, \text{put}\ e, P_1\rangle\big)$$

$$P_1 \equiv \text{alt}\ \big((\digamma\ \text{seq})\,\langle \text{get}\ b, \text{put}\ f, P_1\rangle, (\digamma\ \text{seq})\,\langle \text{get}\ a, \text{put}\ d, P_0\rangle\big)$$

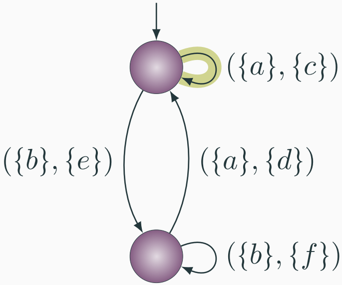


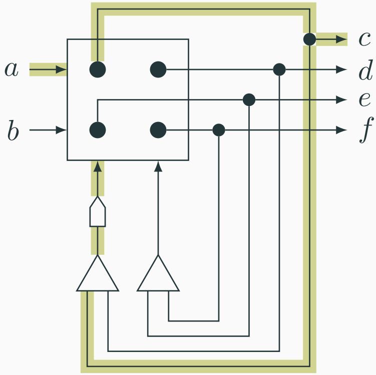


A two-state solution

$$P_0 \equiv \mathsf{alt}\ \big((F\ \mathsf{seq})\,\langle \mathsf{get}\ a, \mathsf{put}\ c, P_0\rangle, (F\ \mathsf{seq})\,\langle \mathsf{get}\ b, \mathsf{put}\ e, P_1\rangle\big)$$

$$P_1 \equiv \mathsf{alt}\ \big((F\ \mathsf{seq})\,\langle \mathsf{get}\ b, \mathsf{put}\ f, P_1\rangle, (F\ \mathsf{seq})\,\langle \mathsf{get}\ a, \mathsf{put}\ d, P_0\rangle\big)$$

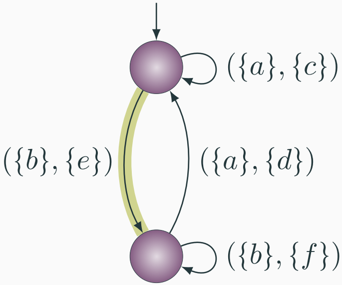


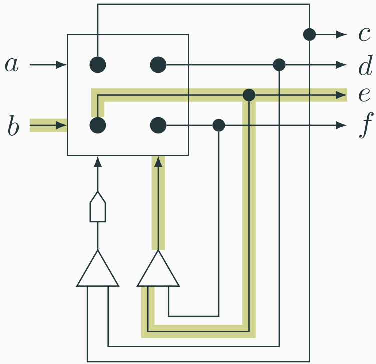


A two-state solution

$$P_0 \equiv \textsf{alt}\ \big((\digamma\ \textsf{seq}\,)\,\langle \textsf{get}\ a, \textsf{put}\ c, P_0\rangle, (\digamma\ \textsf{seq}\,)\,\langle \textsf{get}\ b, \textsf{put}\ e, P_1\rangle\big)$$

$$P_1 \equiv \textsf{alt}\ \big((\digamma\ \textsf{seq}\,)\,\langle \textsf{get}\ b, \textsf{put}\ f, P_1\rangle, (\digamma\ \textsf{seq}\,)\,\langle \textsf{get}\ a, \textsf{put}\ d, P_0\rangle\big)$$

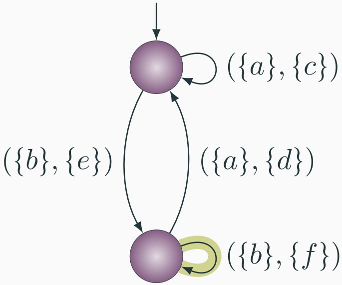


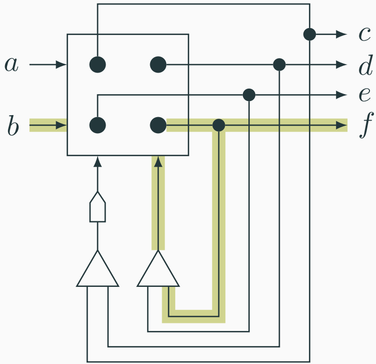


A two-state solution

$$P_0 \equiv \mathsf{alt}\ \big((\digamma\ \mathsf{seq})\,\langle \mathsf{get}\ a, \mathsf{put}\ c, P_0\rangle, (\digamma\ \mathsf{seq})\,\langle \mathsf{get}\ b, \mathsf{put}\ e, P_1\rangle\big)$$

$$P_1 \equiv \mathsf{alt}\ \big((\digamma\ \mathsf{seq})\,\langle \mathsf{get}\ b, \mathsf{put}\ f, P_1\rangle, (\digamma\ \mathsf{seq})\,\langle \mathsf{get}\ a, \mathsf{put}\ d, P_0\rangle\big)$$

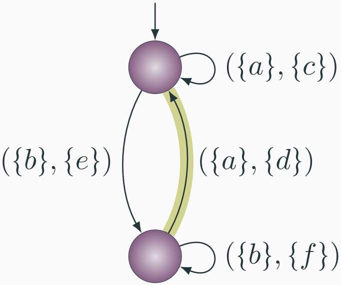


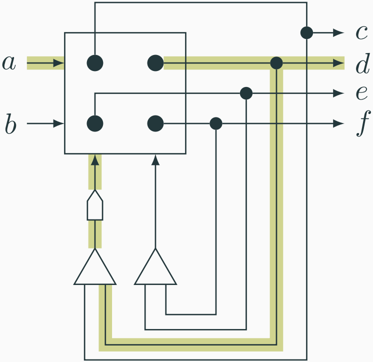


A two-state solution

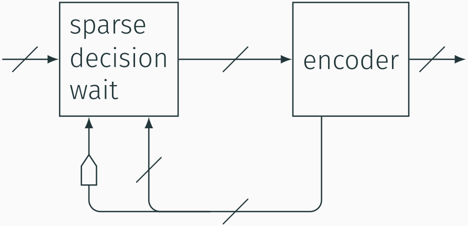
$$P_0 \equiv \mathsf{alt}\ \big((F\ \mathsf{seq})\,\langle \mathsf{get}\ a, \mathsf{put}\ c, P_0\rangle, (F\ \mathsf{seq})\,\langle \mathsf{get}\ b, \mathsf{put}\ e, P_1\rangle\big)$$

$$P_1 \equiv \mathsf{alt}\ \big((F\ \mathsf{seq})\,\langle \mathsf{get}\ b, \mathsf{put}\ f, P_1\rangle, (F\ \mathsf{seq})\,\langle \mathsf{get}\ a, \mathsf{put}\ d, P_0\rangle\big)$$

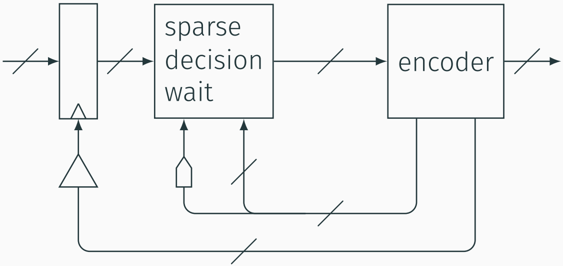




Further provisions

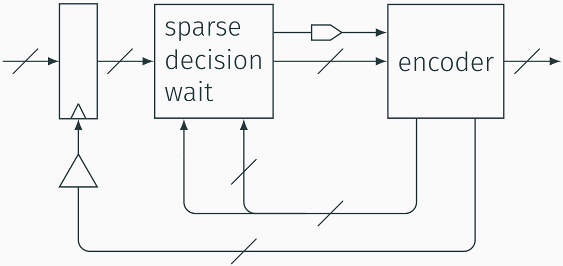


Further provisions



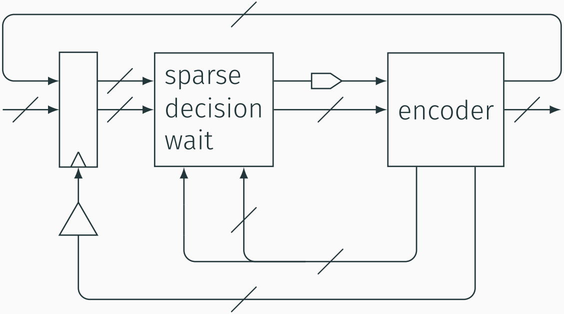
- concurrent inputs require at least one sequencer

Further provisions



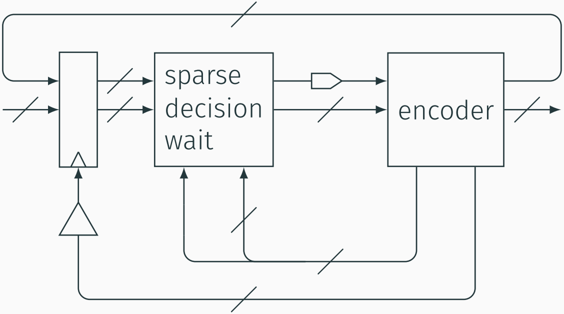
- concurrent inputs require at least one sequencer
- initial non-quiescence requires a PUSH in a different place

Further provisions



- concurrent inputs require at least one sequencer
- initial non-quiescence requires a PUSH in a different place
- non-deterministically concurrent inputs require feedback

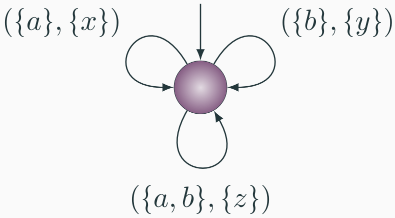
Further provisions

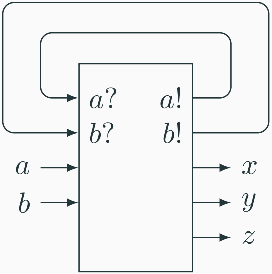


- concurrent inputs require at least one sequencer
- initial non-quiescence requires a PUSH in a different place
- non-deterministically concurrent inputs require feedback
- tons of optimizations possible

Non-deterministically concurrent inputs

When an input burst contains another from the same state

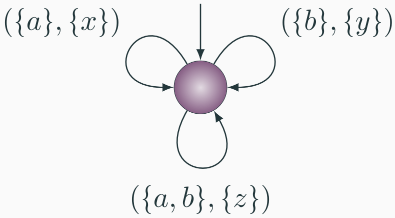


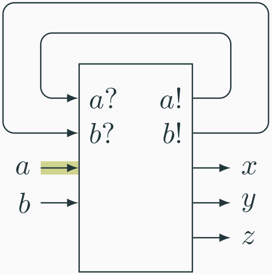


detect concurrency by a deliberate race condition !

Non-deterministically concurrent inputs

When an input burst contains another from the same state

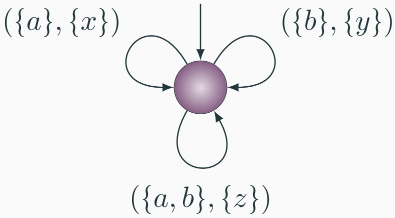


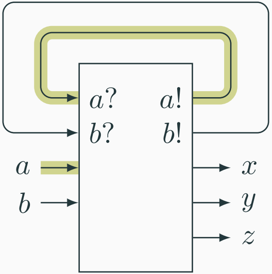


detect concurrency by a deliberate race condition !

Non-deterministically concurrent inputs

When an input burst contains another from the same state

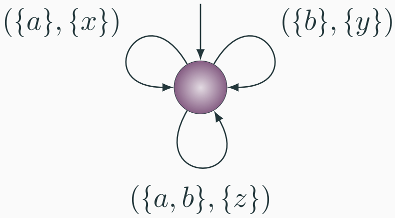


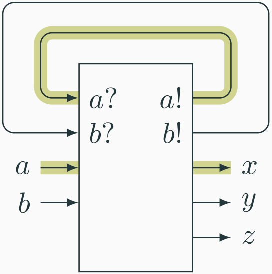


detect concurrency by a deliberate race condition !

Non-deterministically concurrent inputs

When an input burst contains another from the same state

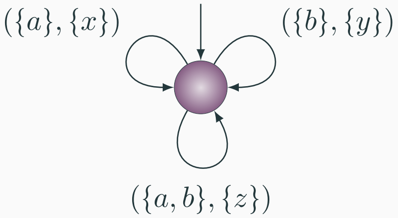


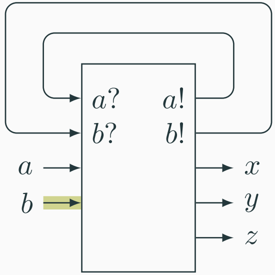


detect concurrency by a deliberate race condition !

Non-deterministically concurrent inputs

When an input burst contains another from the same state

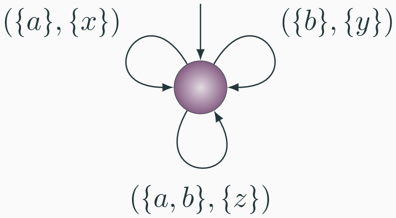


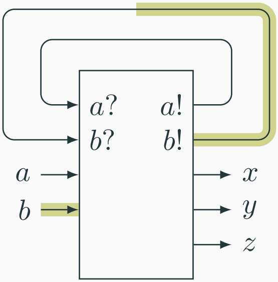


detect concurrency by a deliberate race condition !

Non-deterministically concurrent inputs

When an input burst contains another from the same state

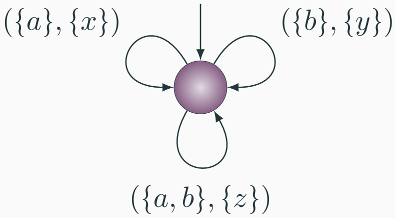


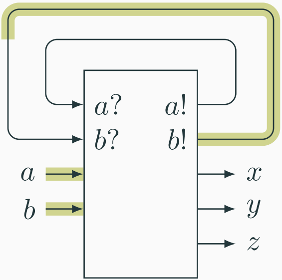


detect concurrency by a deliberate race condition !

Non-deterministically concurrent inputs

When an input burst contains another from the same state

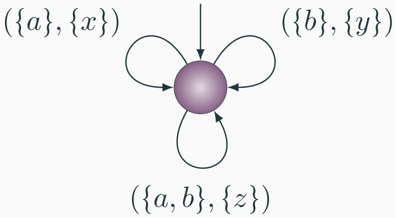


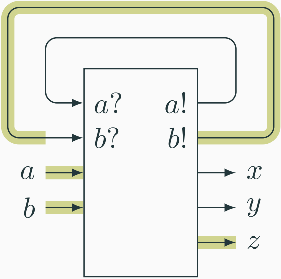


detect concurrency by a deliberate race condition !

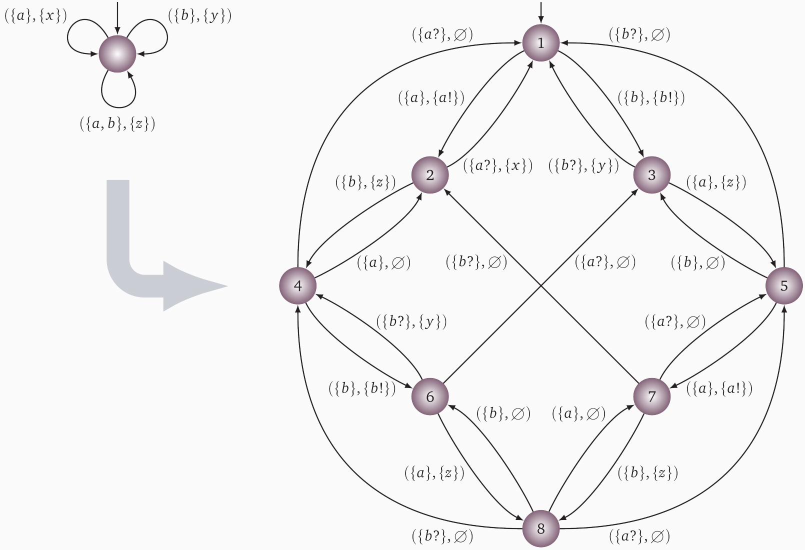
Non-deterministically concurrent inputs

When an input burst contains another from the same state

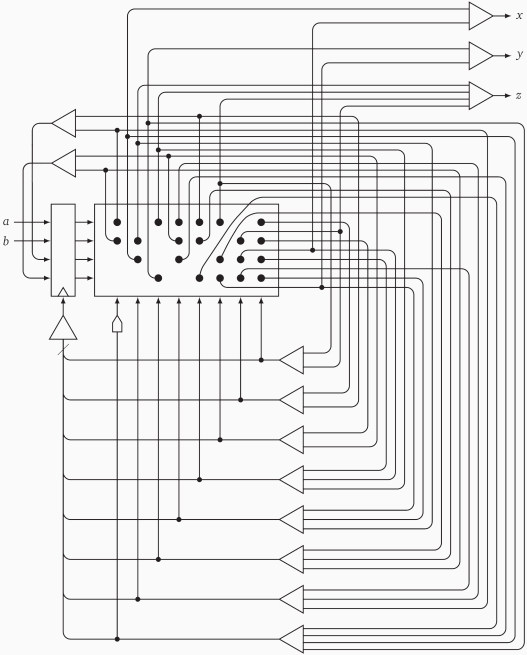




detect concurrency by a deliberate race condition !



({a}, {x})
({b}, {y})
({a, b}, {z})
1
2
3
4
5
6
7
8
({a?}, ∅)
({b?}, ∅)
({a}, {a!})
({b}, {b!})
({a?}, {x})
({b?}, {y})
({b}, {z})
({a}, {z})
({a}, ∅)
({b?}, ∅)
({a?}, ∅)
({b}, ∅)
({b?}, {y})
({a?}, ∅)
({b}, {b!})
({a}, {a!})
({b}, ∅)
({a}, ∅)
({a}, {z})
({b}, {z})
({b?}, ∅)
({a?}, ∅)



x
y
z
a
b

Limitations of state based synthesis

- exponential time and space to compute transducers
- feasibility only for small specifications
- results not always as good as manual

However, state based synthesis can serve as the base case of a recursive algorithm for direct mapping synthesis.

Direct Mapping Synthesis

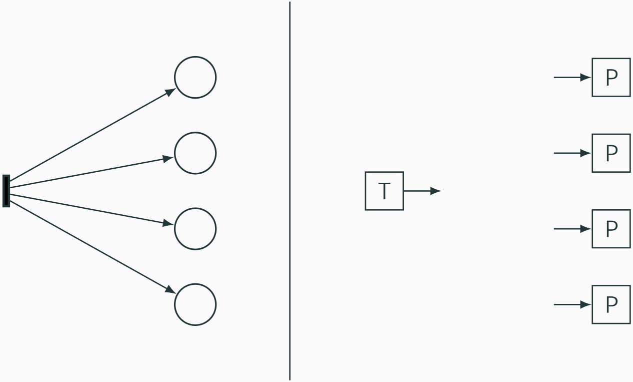
Bypassing state enumeration

Simulate token flow through a Petri net by signals in a circuit.
Use buffers for nodes, wires for arcs, and 4 kinds of interfaces.

- one transition to many places
- many transitions to one place
- many places to one transition
- one place to many transitions

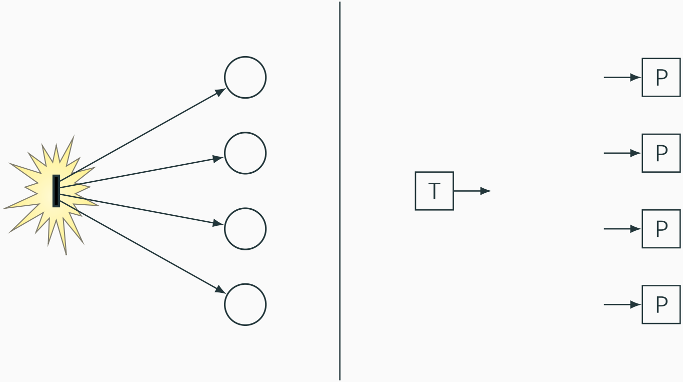
Four kinds of interfaces

one transition to many places



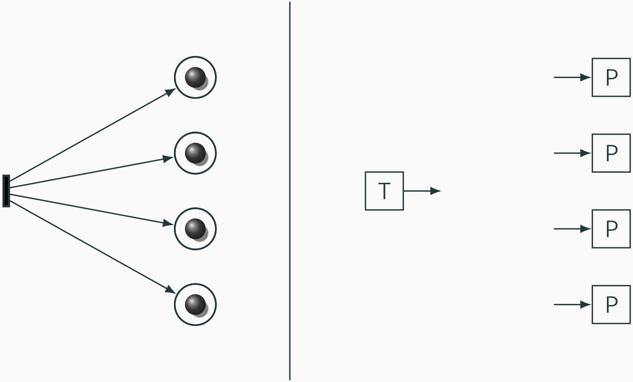
Four kinds of interfaces

one transition to many places



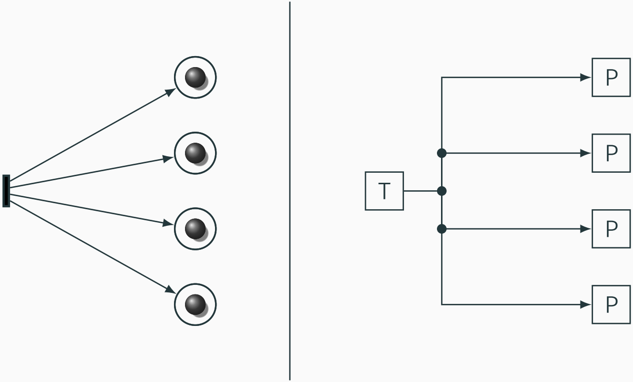
Four kinds of interfaces

one transition to many places



Four kinds of interfaces

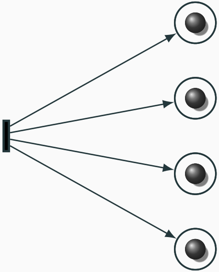
one transition to many places

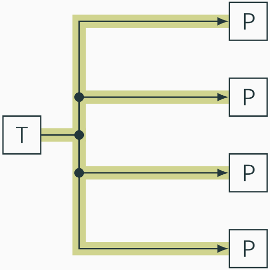


Use a FORK !

Four kinds of interfaces

one transition to many places

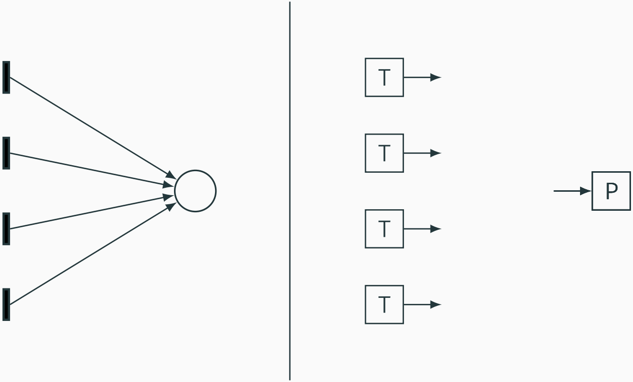




Use a FORK !

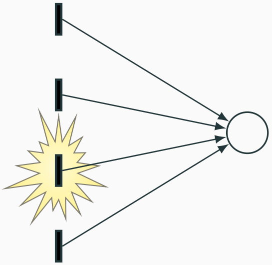
Four kinds of interfaces

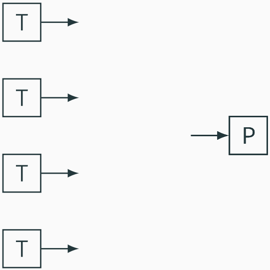
many transitions to one place



Four kinds of interfaces

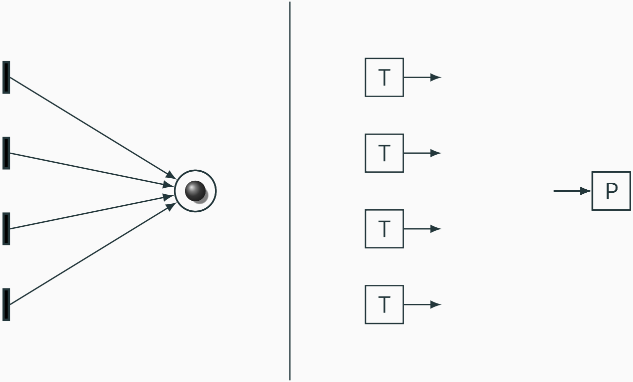
many transitions to one place





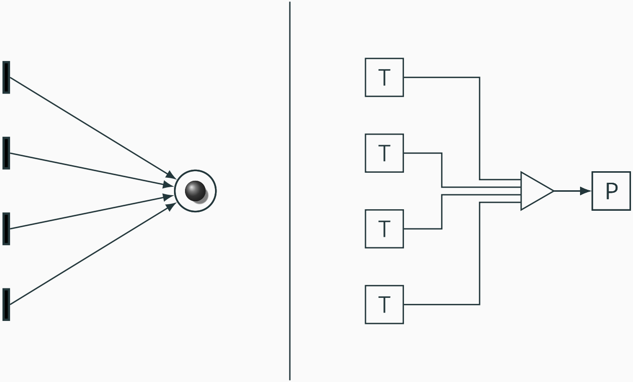
Four kinds of interfaces

many transitions to one place



Four kinds of interfaces

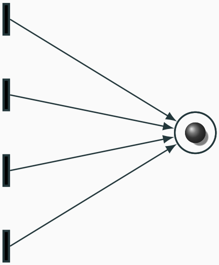
many transitions to one place

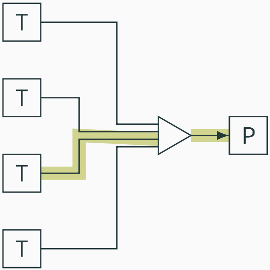


Use a MERGE !

Four kinds of interfaces

many transitions to one place

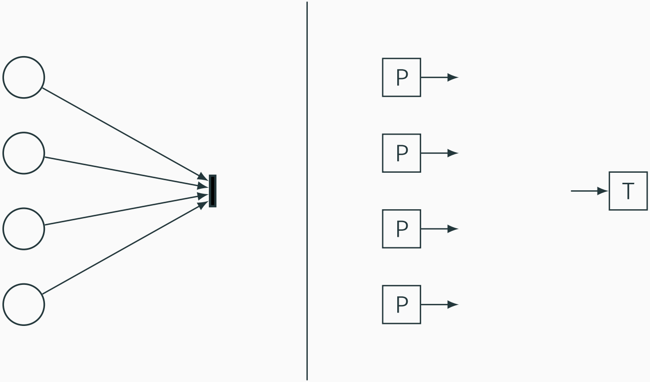




Use a MERGE !

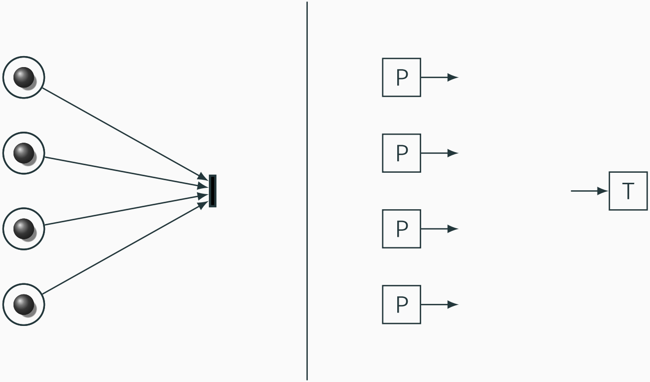
Four kinds of interfaces

many places to one transition



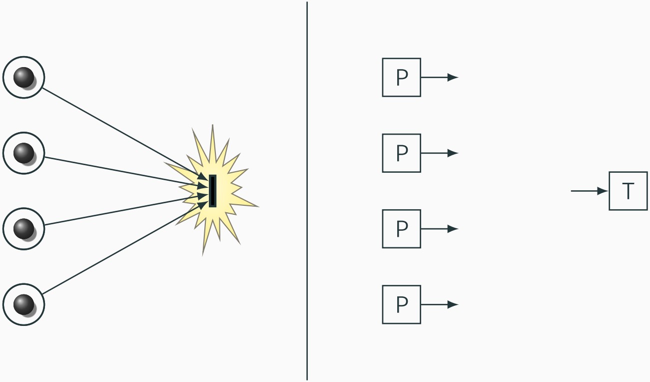
Four kinds of interfaces

many places to one transition



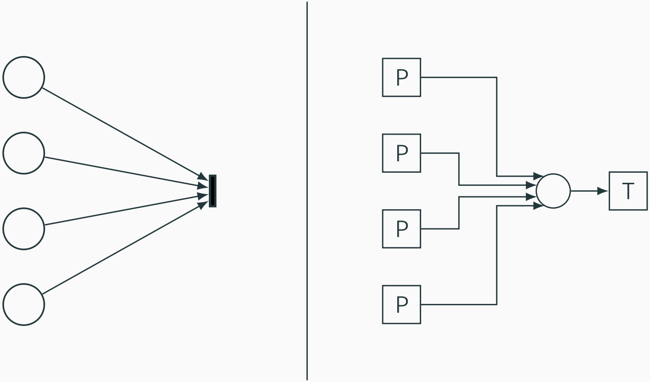
Four kinds of interfaces

many places to one transition



Four kinds of interfaces

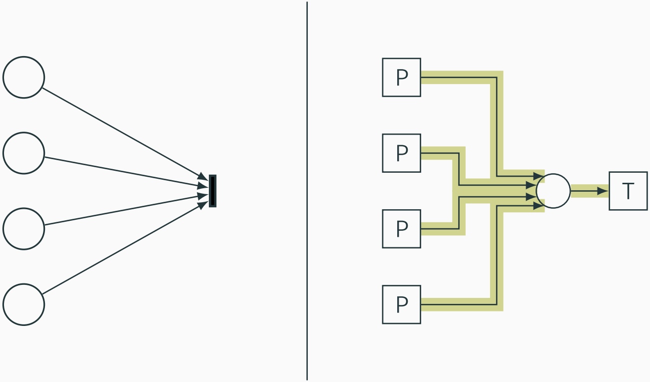
many places to one transition



Use a JOIN !

Four kinds of interfaces

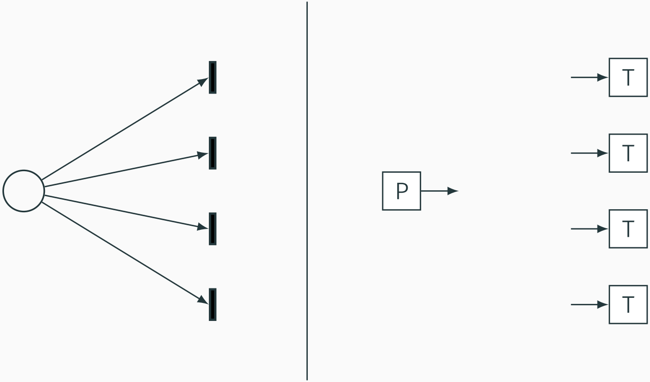
many places to one transition



Use a JOIN !

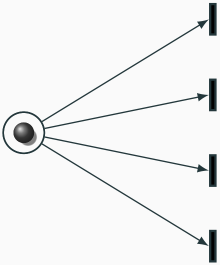
Four kinds of interfaces

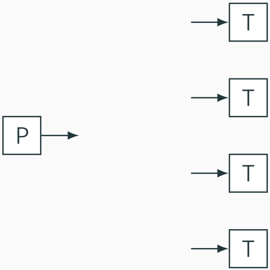
one place to many transitions



Four kinds of interfaces

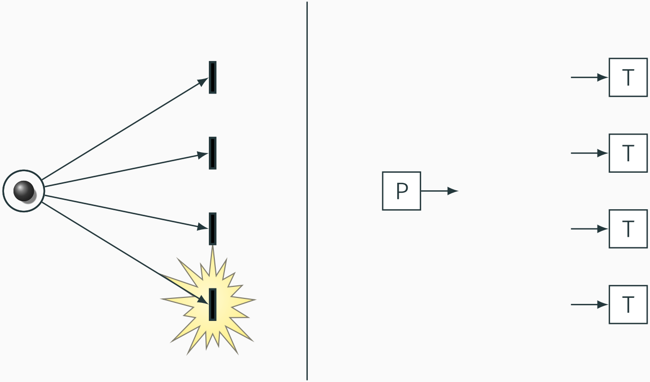
one place to many transitions





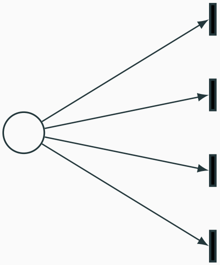
Four kinds of interfaces

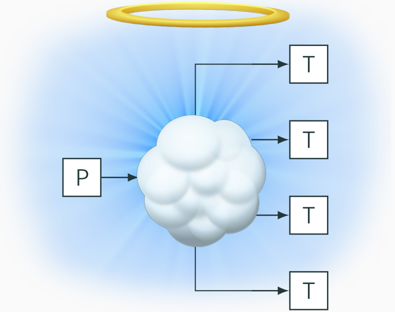
one place to many transitions



Four kinds of interfaces

one place to many transitions

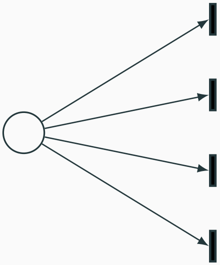


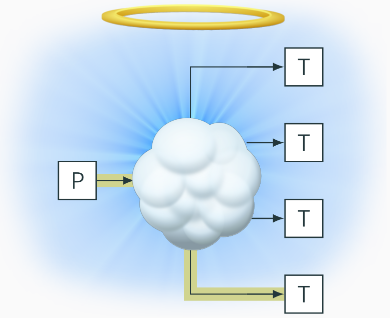


Use an angelic router !

Four kinds of interfaces

one place to many transitions





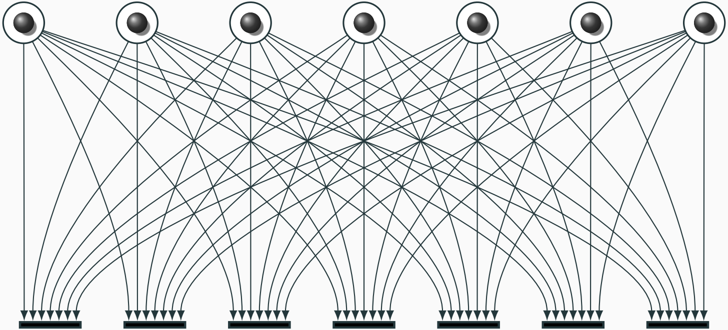
Use an angelic router !

Four kinds of interfaces

one place to many transitions

Use an angelic router !

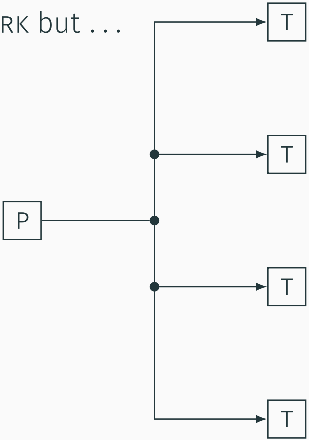
Routing for real



probability of a successfully negotiated token transfer: $\frac{1}{n^{n-1}}$

A feasible protocol

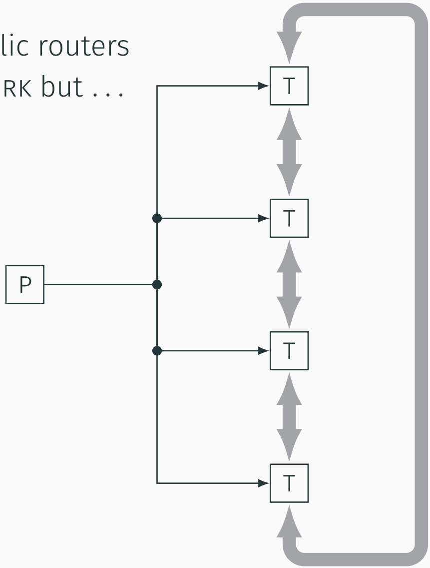
forget angelic routers
use a FORK but . . .



A feasible protocol

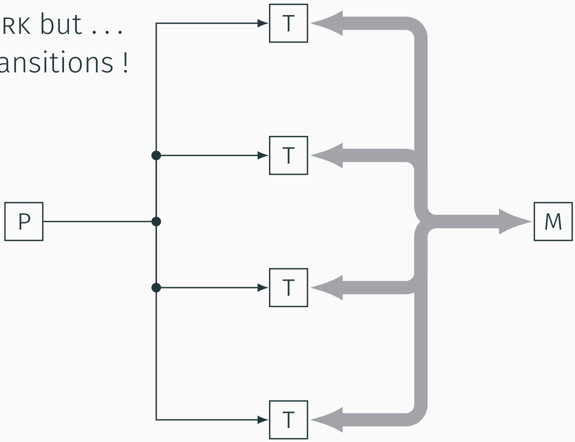
forget angelic routers

use a FORK but . . .



A feasible protocol

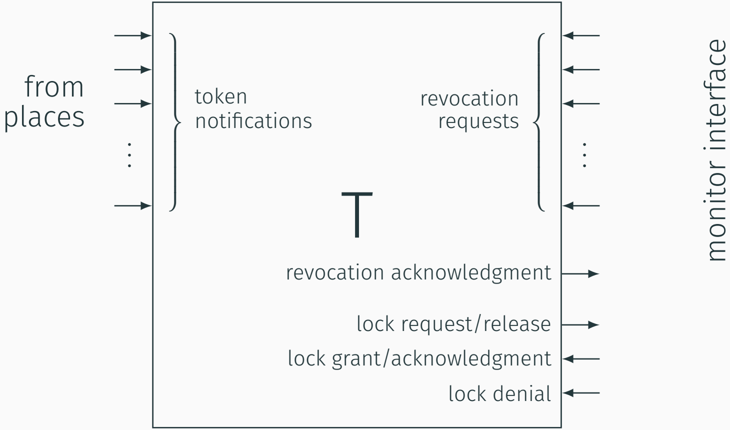
forget angelic routers
use a FORK but ...
arbitrate among transitions !



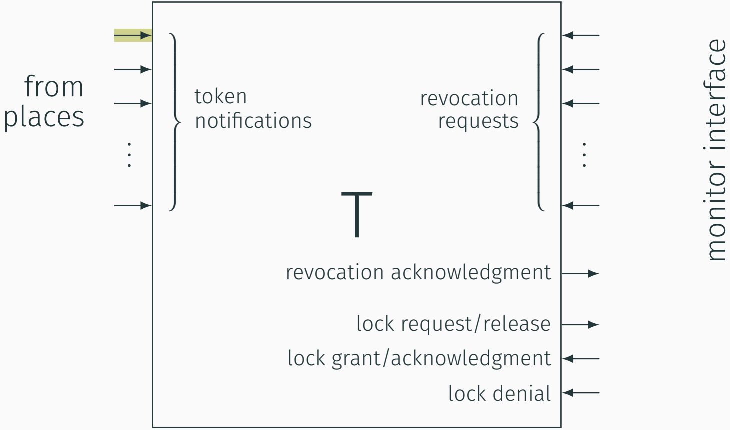
To-do list

- Precisely describe the transition $\Leftrightarrow$ monitor protcol.
- Design the transition circuit.
- Design the monitor circuit.

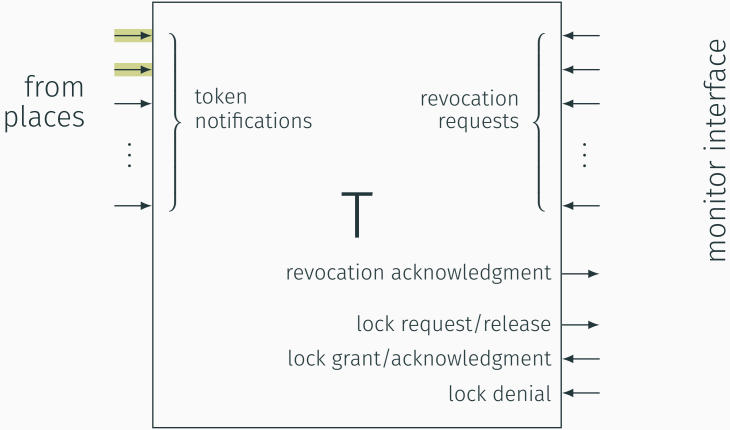
Transition-side protocol



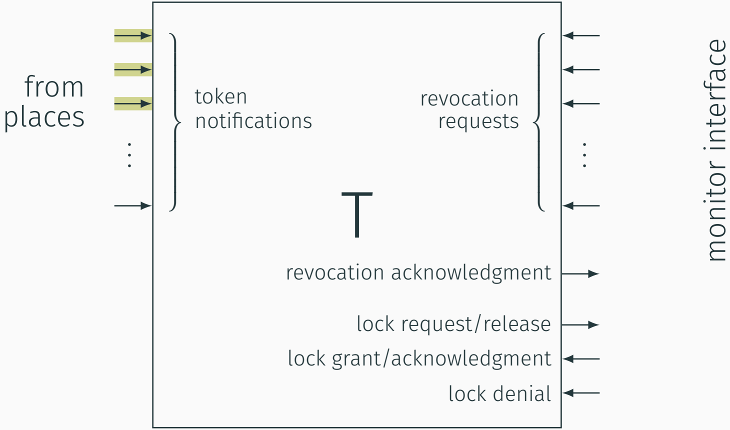
Transition-side protocol



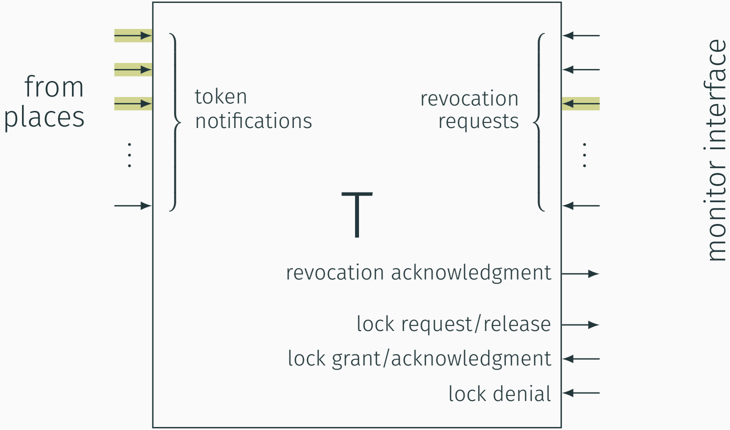
Transition-side protocol



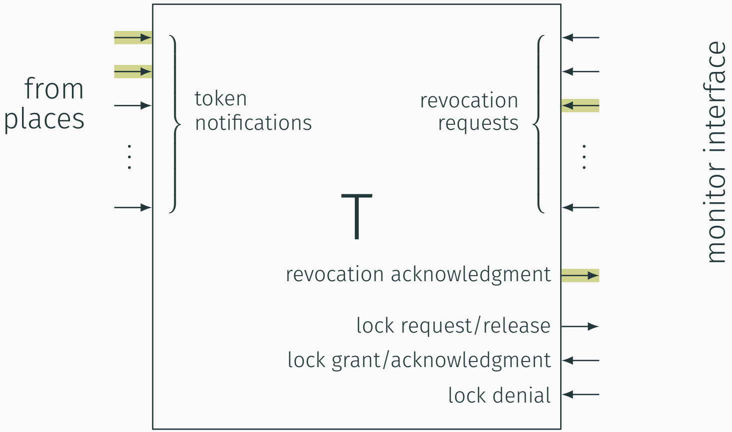
Transition-side protocol



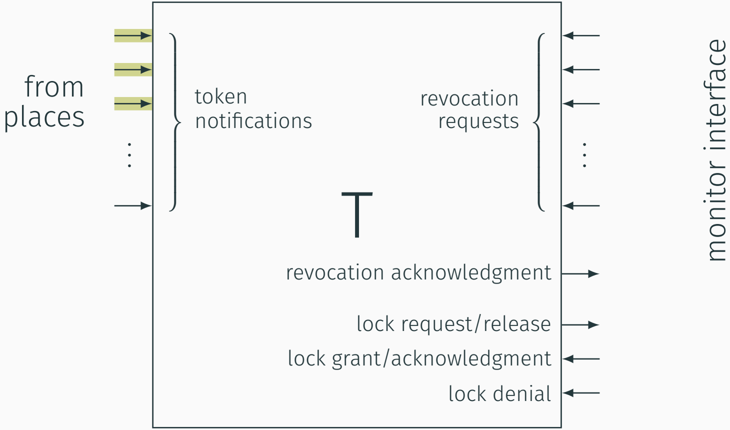
Transition-side protocol



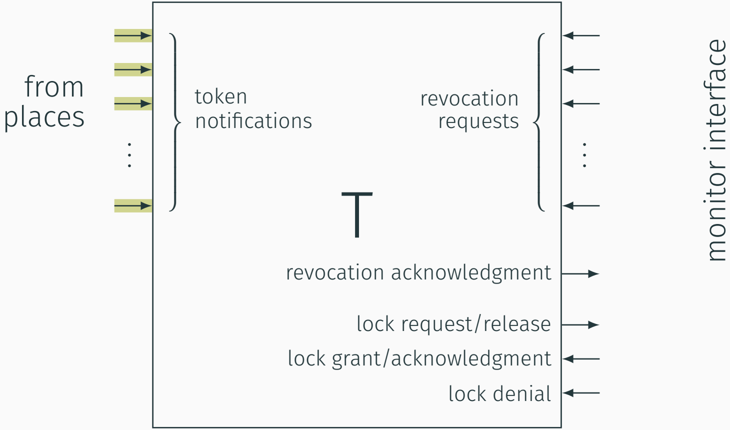
Transition-side protocol



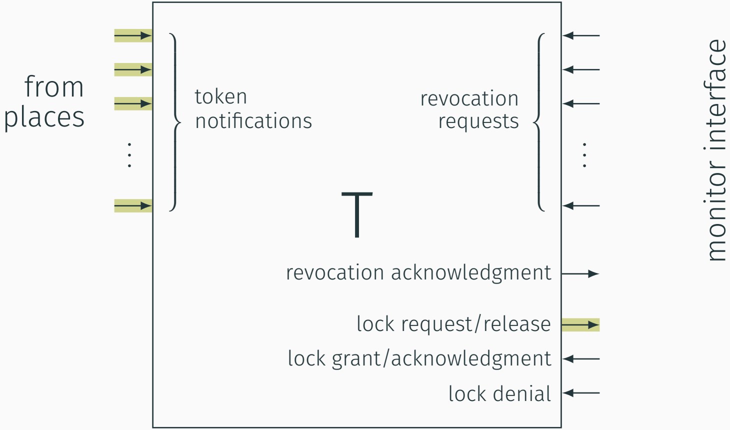
Transition-side protocol



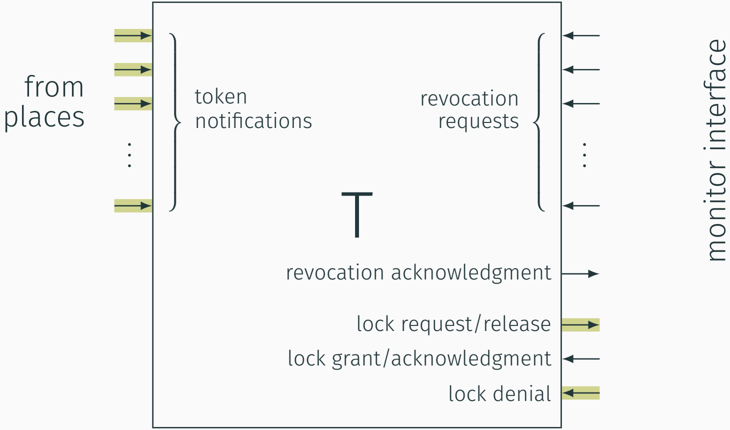
Transition-side protocol



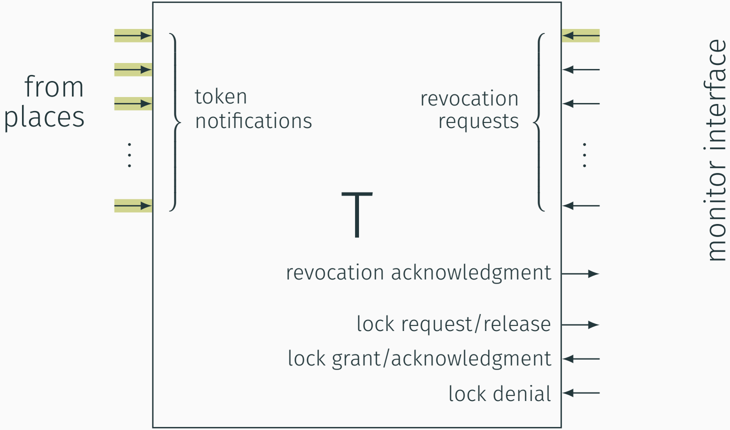
Transition-side protocol



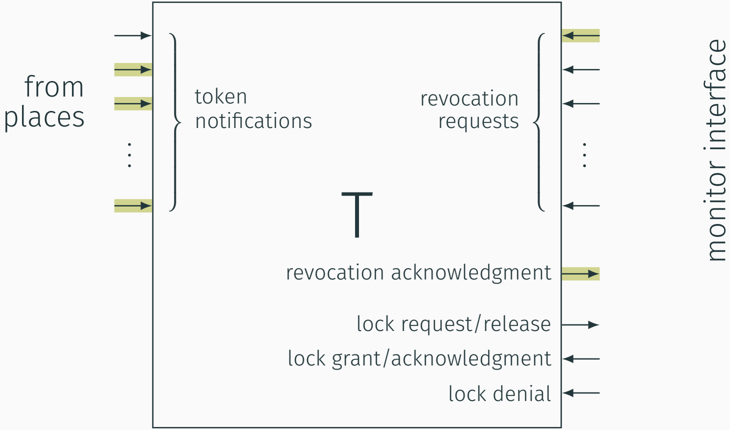
Transition-side protocol



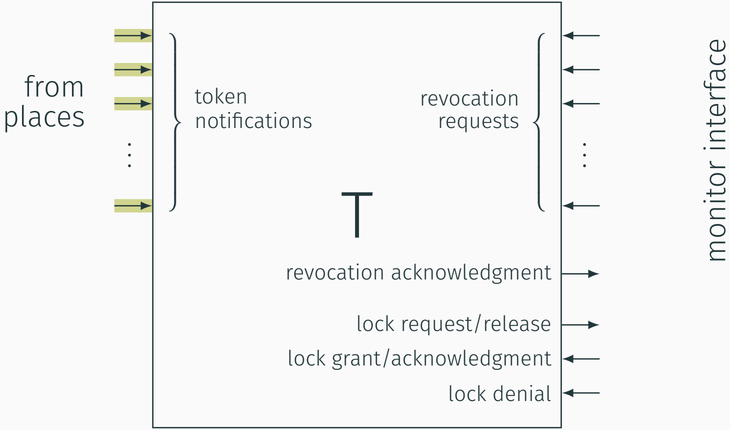
Transition-side protocol



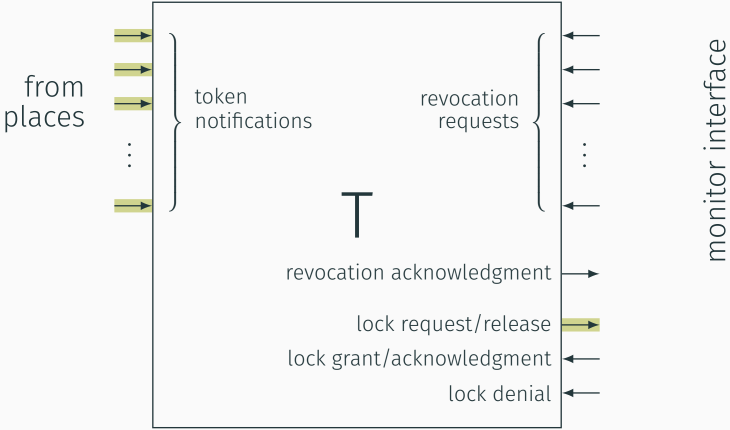
Transition-side protocol



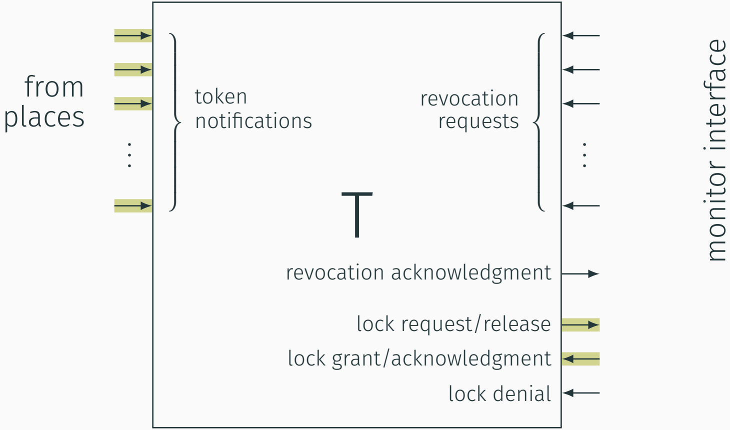
Transition-side protocol



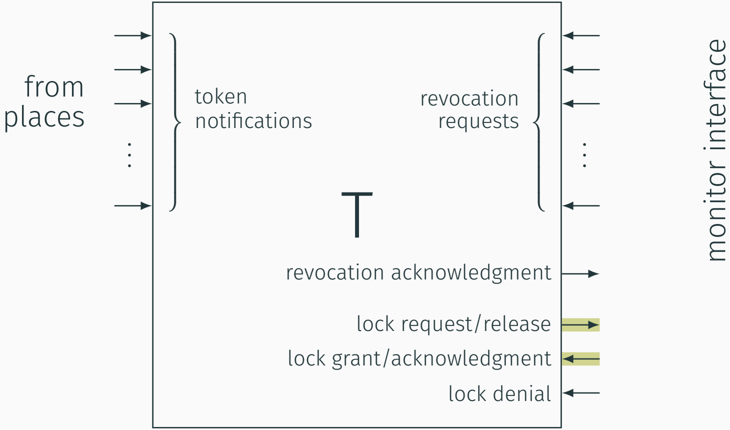
Transition-side protocol



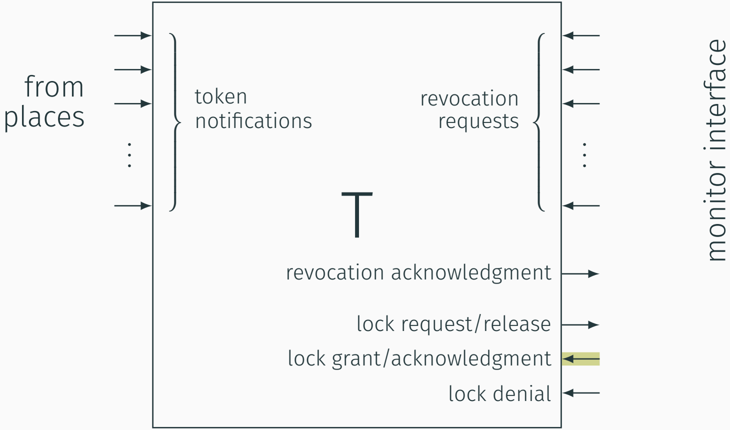
Transition-side protocol



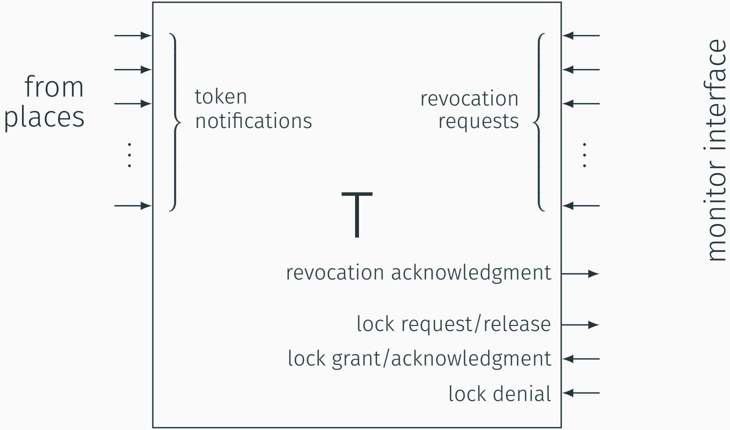
Transition-side protocol



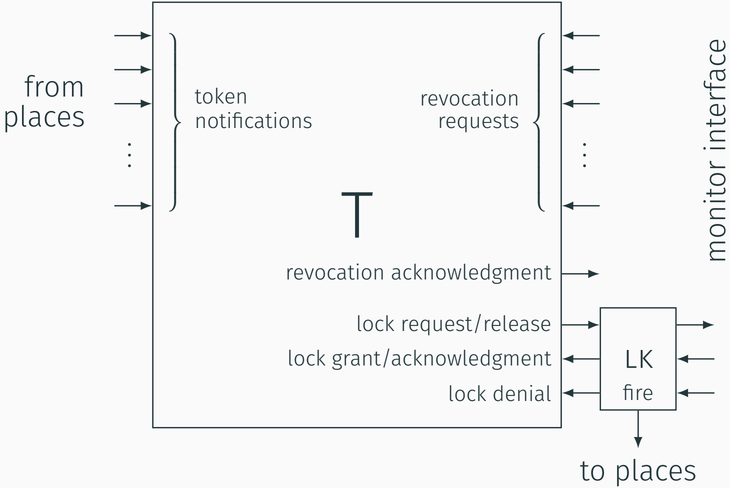
Transition-side protocol



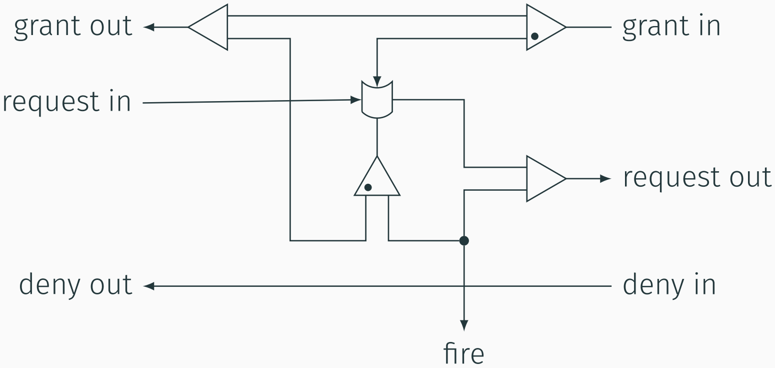
Transition-side protocol



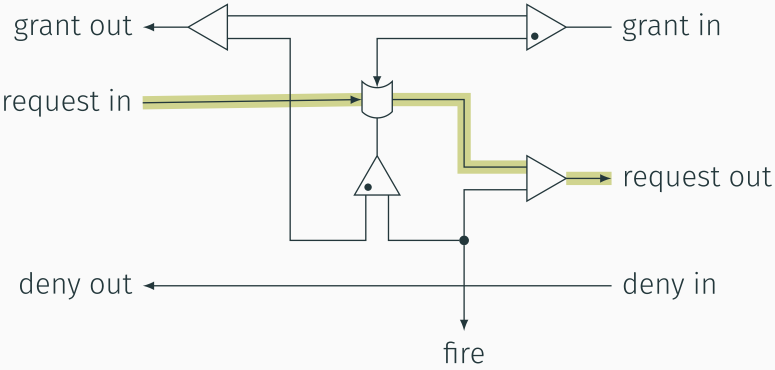
Transition-side protocol



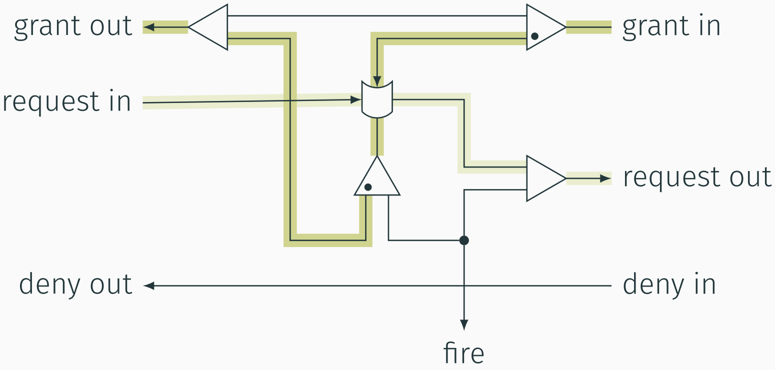
Lock negotiation channel



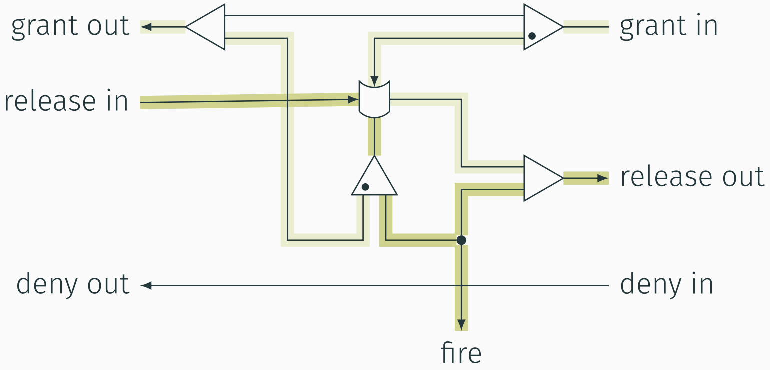
Lock negotiation channel



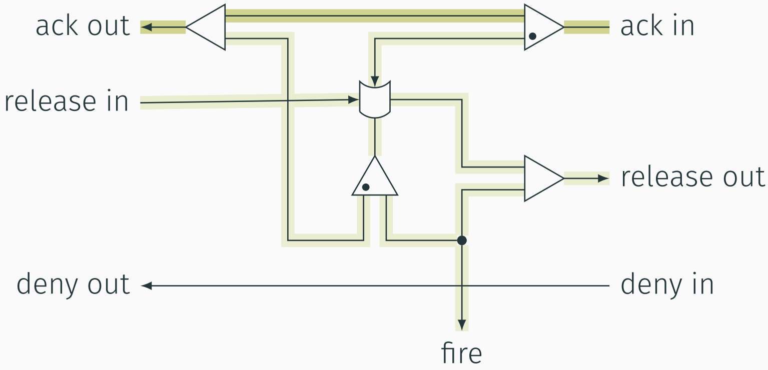
Lock negotiation channel



Lock negotiation channel



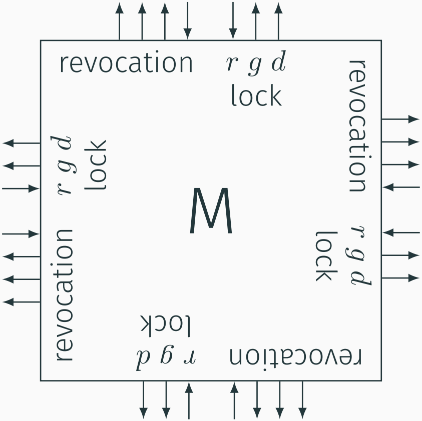
Lock negotiation channel



Monitor-side protocol

lock requests can be

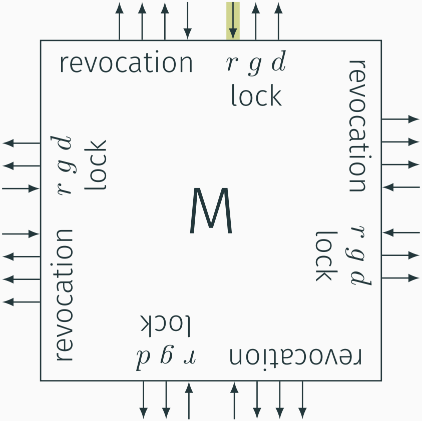
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

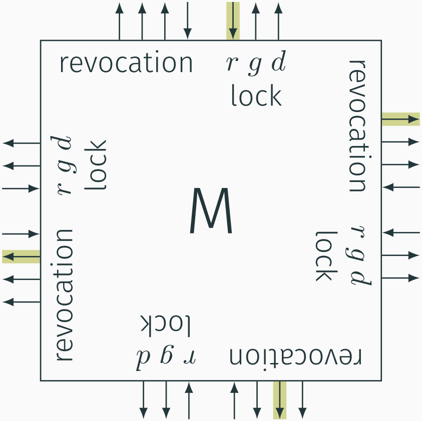
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

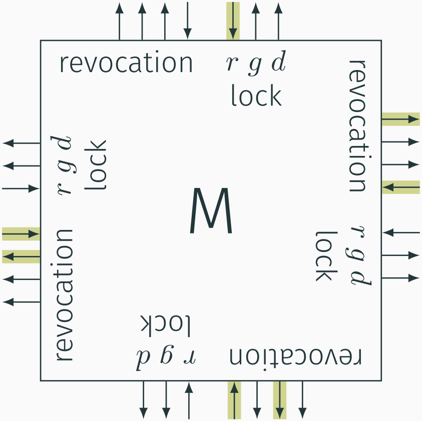
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

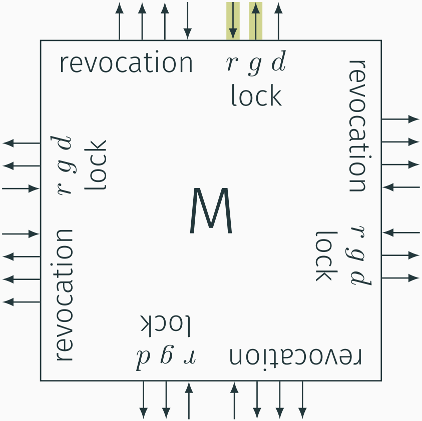
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

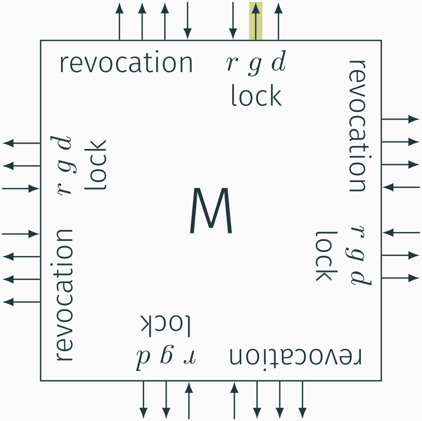
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

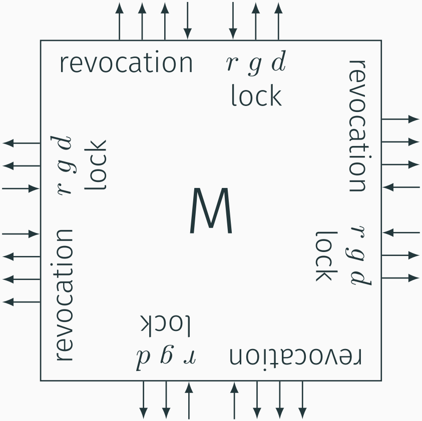
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

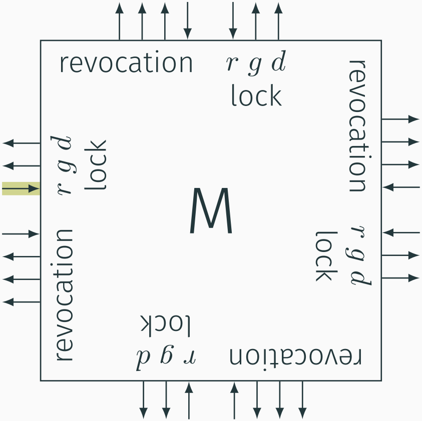
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

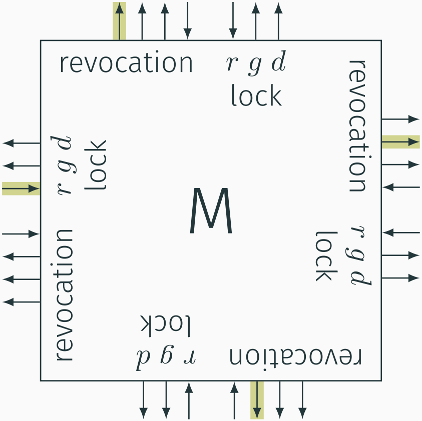
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

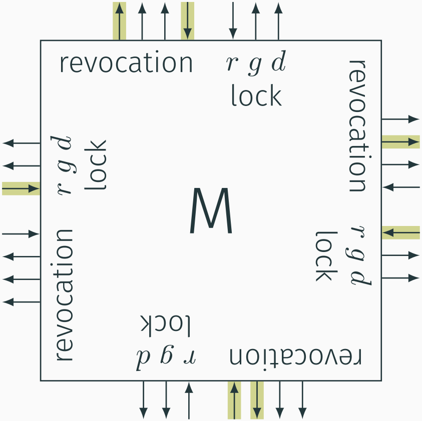
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

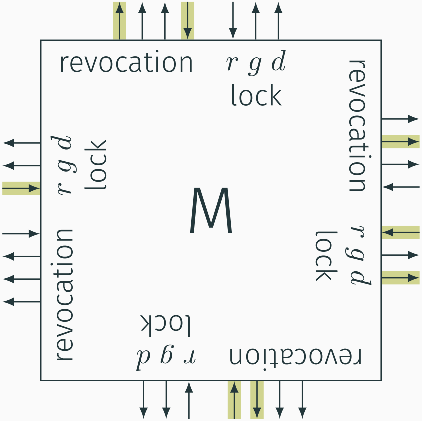
- grantable
- deniable
- blockable



Monitor-side protocol

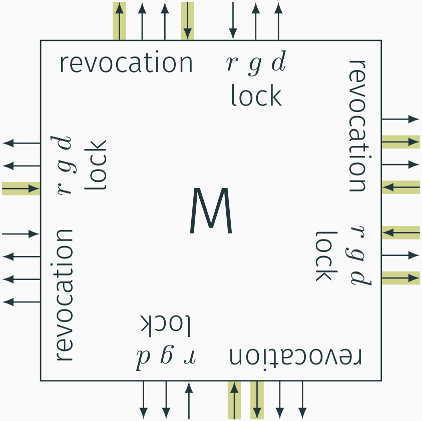
lock requests can be

- grantable
- deniable
- blockable



lock requests can be

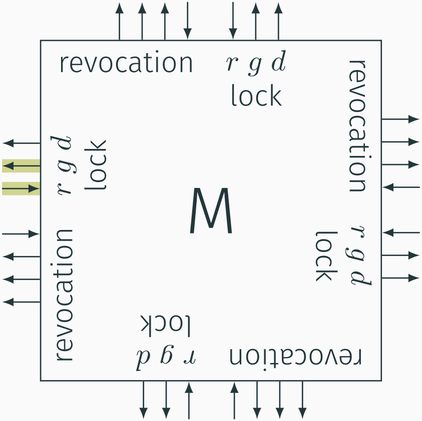
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

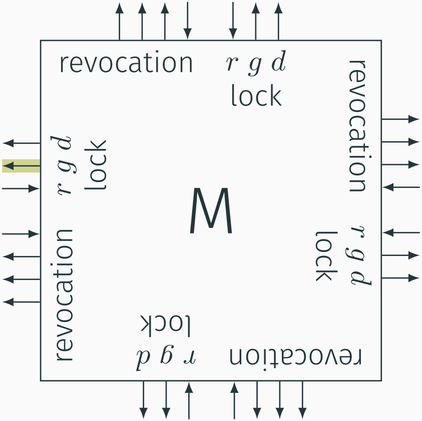
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

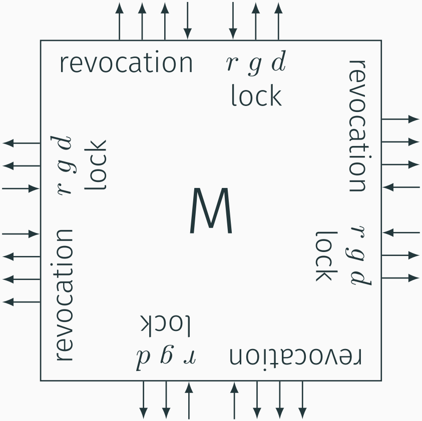
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

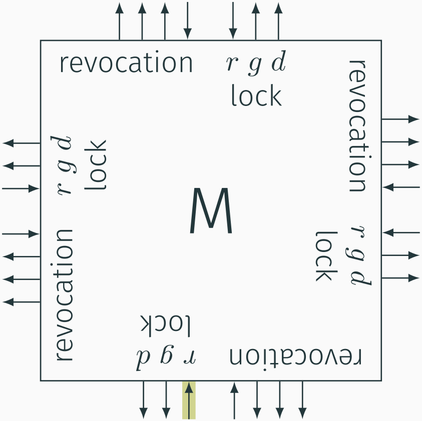
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

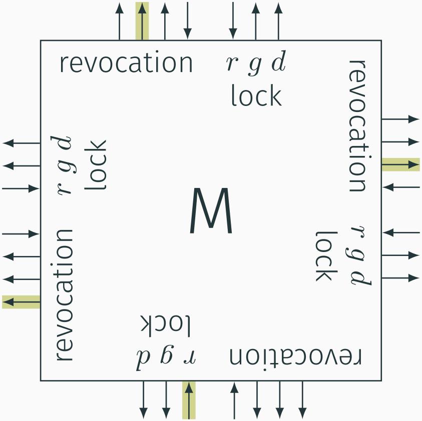
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

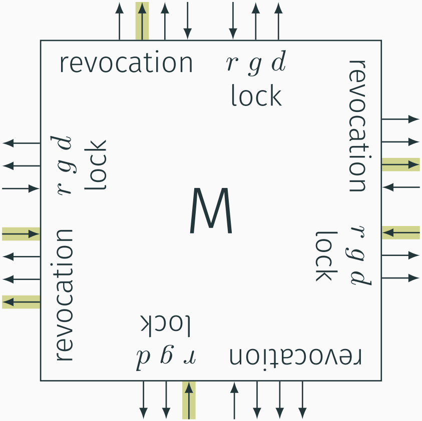
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

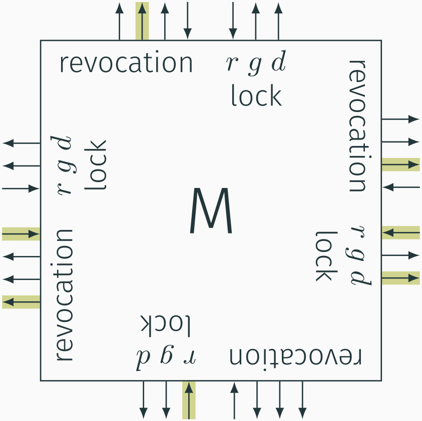
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

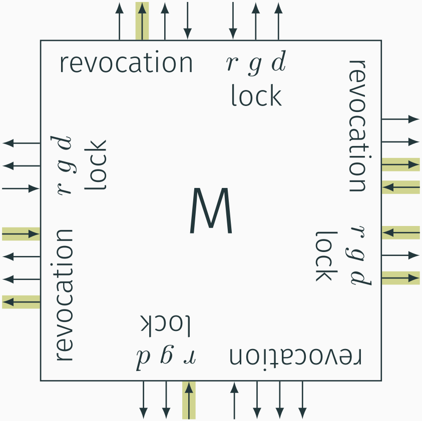
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

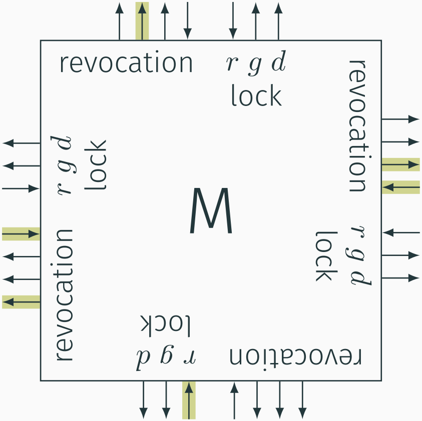
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

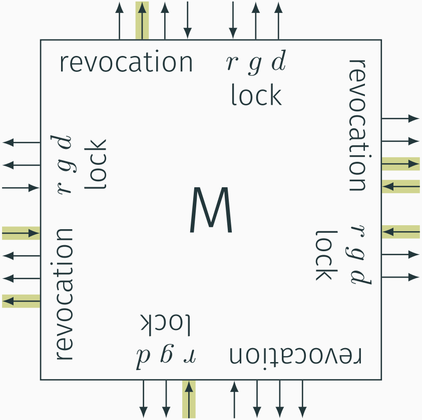
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

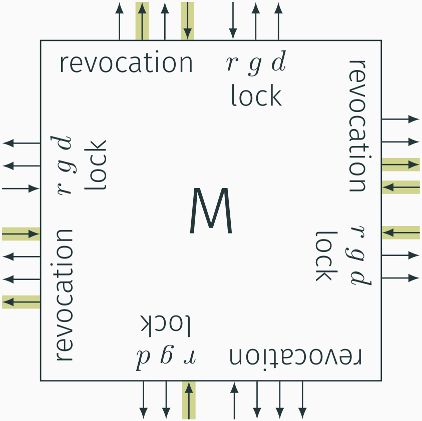
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

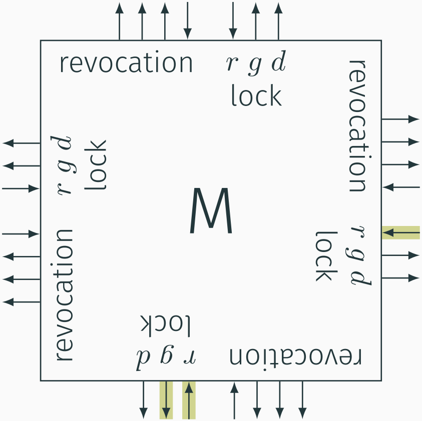
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

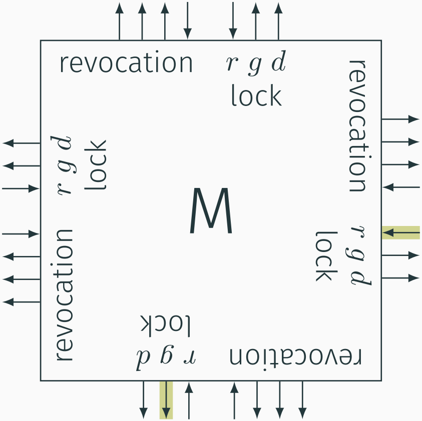
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

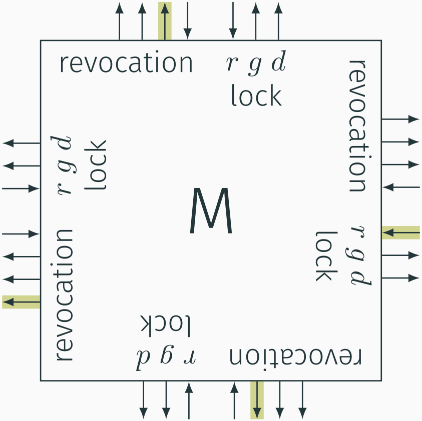
- grantable
- deniable
- blockable



Monitor-side protocol

lock requests can be

- grantable
- deniable
- blockable



To-do list

- ✓ Precisely describe the transition $\Leftrightarrow$ monitor protcol.
- Design the transition circuit.
- Design the monitor circuit.

To-do list

- ✓ Precisely describe the transition ⇔ monitor protcol.
- ~~Design the transition circuit.~~
- Design the monitor circuit.

Monitor process derivation

Snapshot the monitor's progress in a structure

$$\begin{matrix} d & b \\ w & g \end{matrix}$$

consisting of sets d, b, w, and g of transitions such that

- lock requests from d are deniable
- lock requests from b are blockable
- revocation acknowledgments are awaited from w
- locks will be granted to g after acknowledgments.

Monitor process derivation

Snapshot the monitor's progress in a structure

$$\begin{array}{cc} d & b \\ w & g \end{array}$$

consisting of sets d, b, w, and g of transitions such that

- lock requests from d are deniable
- lock requests from b are blockable
- revocation acknowledgments are awaited from w
- locks will be granted to g after acknowledgments.

Monitor process derivation

Snapshot the monitor's progress in a structure

$$\begin{bmatrix} d & b \\ w & g \end{bmatrix}$$

consisting of sets d, b, w, and g of transitions such that

- lock requests from d are deniable
- lock requests from b are blockable
- revocation acknowledgments are awaited from w
- locks will be granted to g after acknowledgments.

Monitor process derivation

Snapshot the monitor's progress in a structure

$$\begin{array}{cc} d & b \\ w & g \end{array}$$

consisting of sets d, b, w, and g of transitions such that

- lock requests from d are deniable
- lock requests from b are blockable
- revocation acknowledgments are awaited from w
- locks will be granted to g after acknowledgments.

Monitor process derivation

Snapshot the monitor's progress in a structure

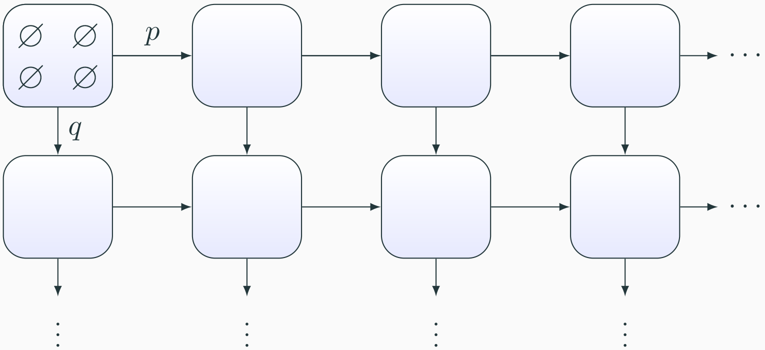
$$\begin{array}{cc} d & b \\ w & g \end{array}$$

consisting of sets d, b, w, and g of transitions such that

- lock requests from d are deniable
- lock requests from b are blockable
- revocation acknowledgments are awaited from w
- locks will be granted to g after acknowledgments.

Monitor process derivation

Build a graph of them starting from all four sets empty



and edges labeled by process expression snippets.

Monitor process derivation

For each transition i, let

- $c_i =$ the set of transitions sharing preset places with i
- $r_i =$ the set of revocation requests issued on behalf of i
- $l_{i0}, l_{i1}, l_{i2} =$ the lock request, grant, and deny signals for i
- $h_i =$ the revocation acknowledgment signal from i.

Then each vertex tells us how to find its neighbors !

Monitor process derivation

For each transition i, let

- c_i = the set of transitions sharing preset places with i
- r_i = the set of revocation requests issued on behalf of i
- l_{i0}, l_{i1}, l_{i2} = the lock request, grant, and deny signals for i
- h_i = the revocation acknowledgment signal from i.

Then each vertex tells us how to find its neighbors !

Monitor process derivation

For each transition i, let

- c_i = the set of transitions sharing preset places with i
- r_i = the set of revocation requests issued on behalf of i
- l_{i0}, l_{i1}, l_{i2} = the lock request, grant, and deny signals for i
- h_i = the revocation acknowledgment signal from i.

Then each vertex tells us how to find its neighbors !

Monitor process derivation

For each transition i, let

- c_i = the set of transitions sharing preset places with i
- r_i = the set of revocation requests issued on behalf of i
- l_{i0}, l_{i1}, l_{i2} = the lock request, grant, and deny signals for i
- h_i = the revocation acknowledgment signal from i.

Then each vertex tells us how to find its neighbors !

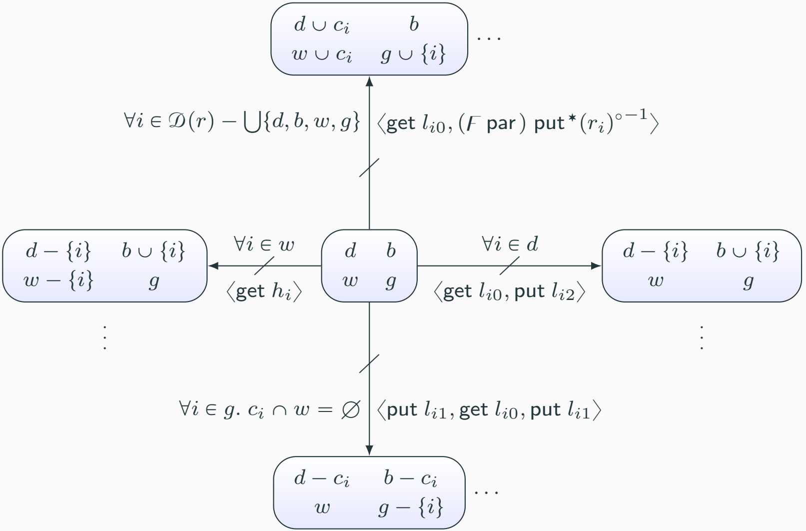
Monitor process derivation

For each transition i, let

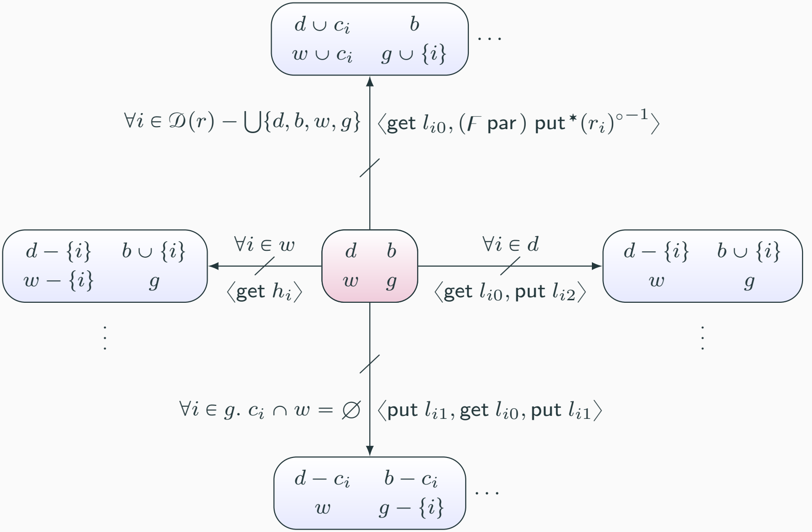
- c_i = the set of transitions sharing preset places with i
- r_i = the set of revocation requests issued on behalf of i
- l_{i0}, l_{i1}, l_{i2} = the lock request, grant, and deny signals for i
- h_i = the revocation acknowledgment signal from i.

Then each vertex tells us how to find its neighbors !

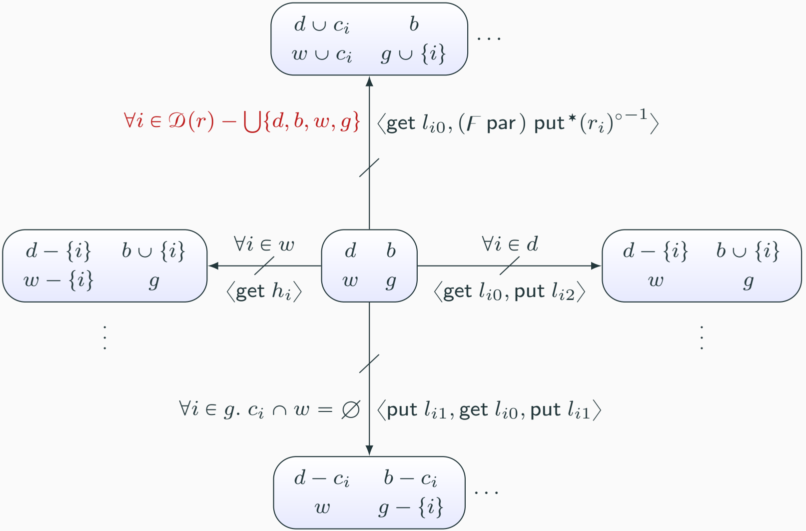
Monitor process derivation



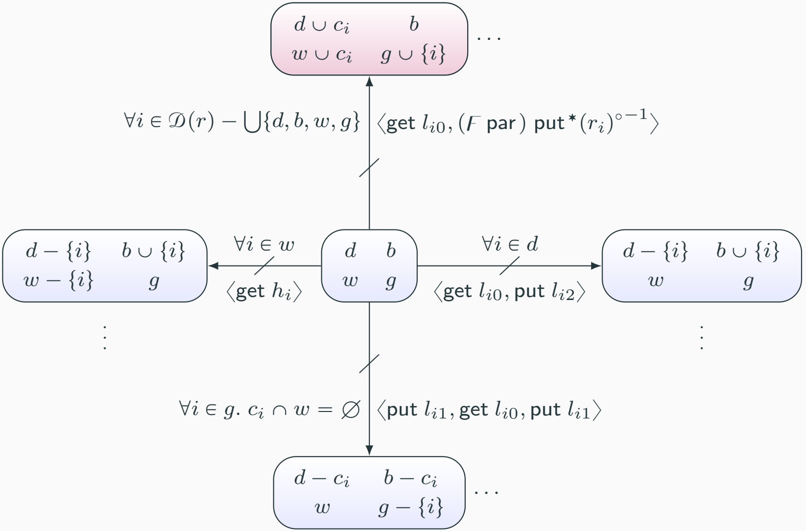
Monitor process derivation



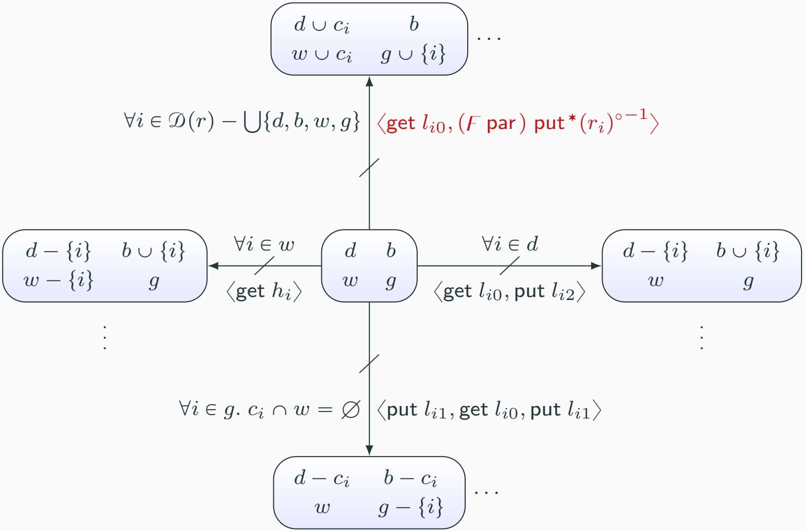
Monitor process derivation



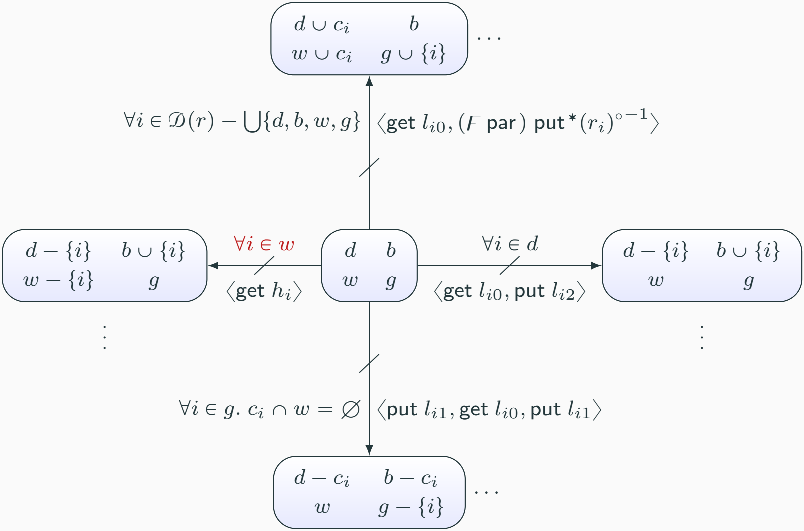
Monitor process derivation



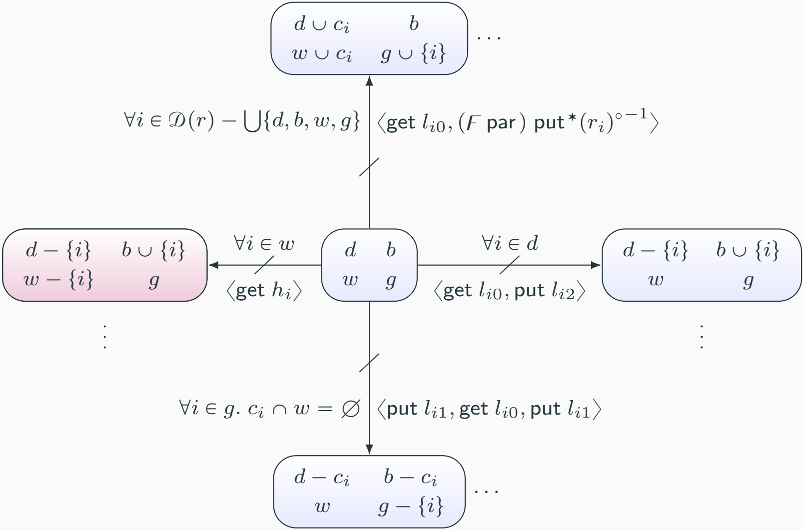
Monitor process derivation



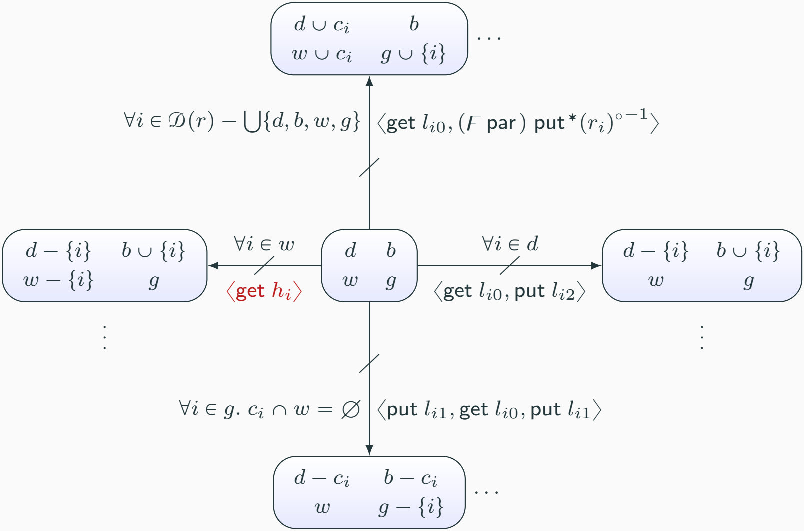
Monitor process derivation



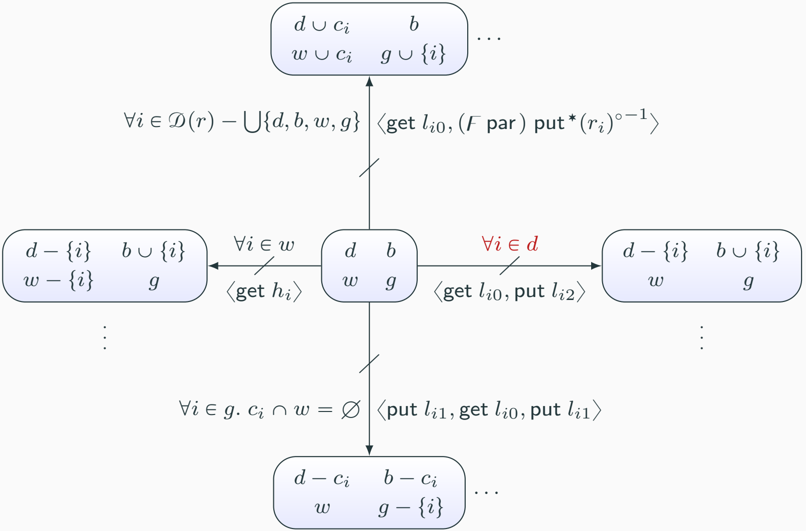
Monitor process derivation



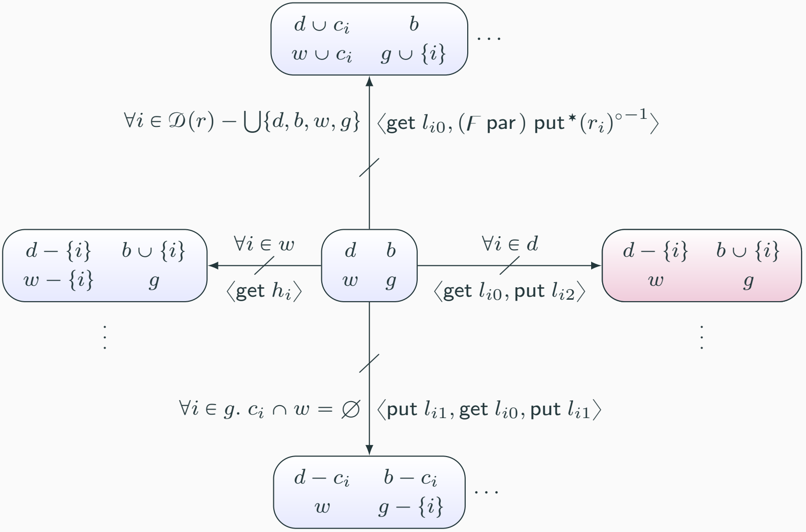
Monitor process derivation



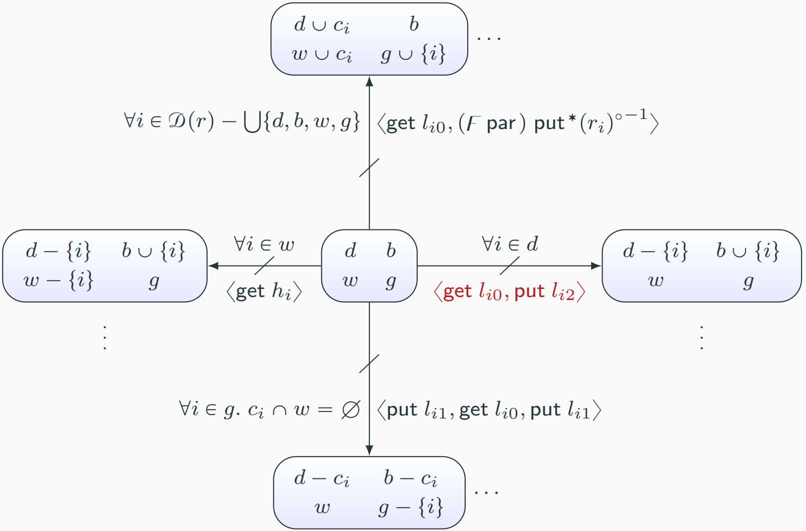
Monitor process derivation



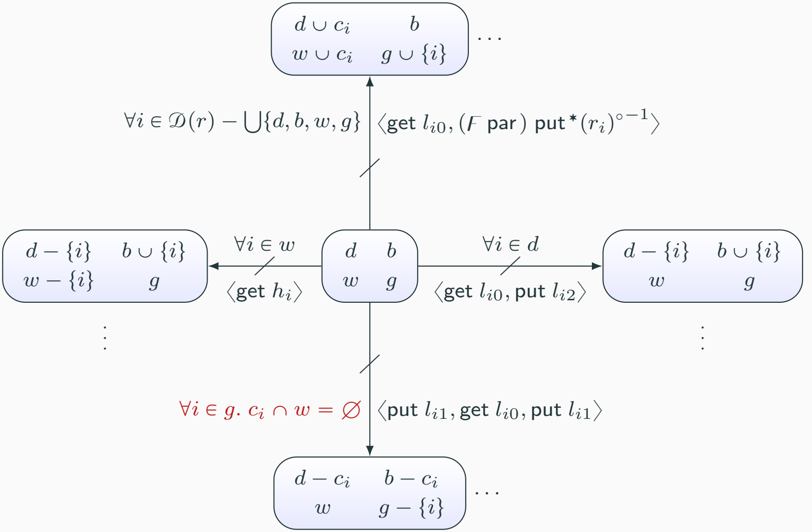
Monitor process derivation



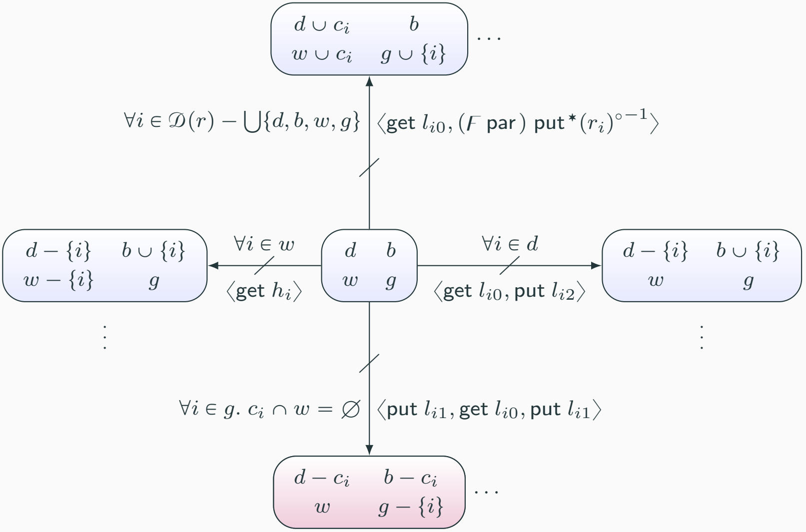
Monitor process derivation



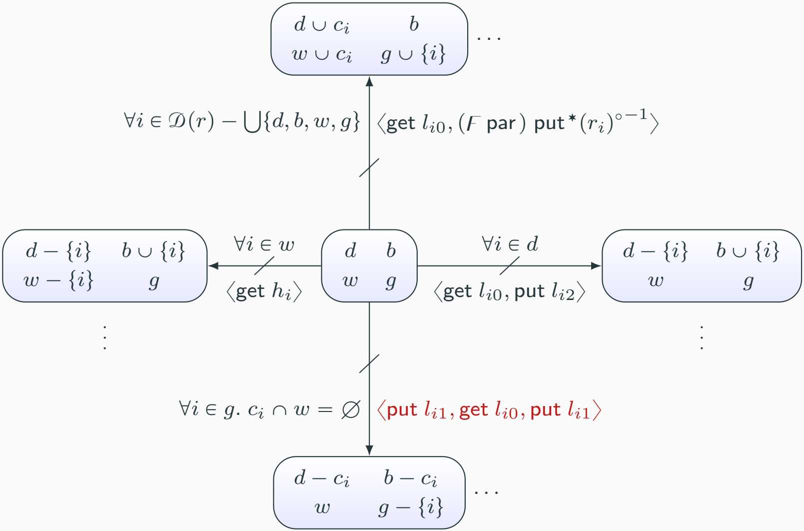
Monitor process derivation



Monitor process derivation



Monitor process derivation



Monitor process derivation

Let G denote the set of all pairs (m, e) in which

- m is a vertex $\langle d, b, w, g \rangle \in \mathcal{D}(G)$
- e is the adjacency set of all pairs (p, n) with
    - a vertex n adjacent to m
    - the list p of processes along the edge from m to n

Monitor process derivation

Let G denote the set of all pairs (m, e) in which

- m is a vertex $\langle d, b, w, g \rangle \in \mathcal{D}(G)$
- e is the adjacency set of all pairs (p, n) with
  - a vertex n adjacent to m
  - the list p of processes along the edge from m to n

Monitor process derivation

Let G denote the set of all pairs (m, e) in which

- m is a vertex $\langle d, b, w, g \rangle \in \mathcal{D}(G)$
- e is the adjacency set of all pairs (p, n) with
    - a vertex n adjacent to m
    - the list p of processes along the edge from m to n

Monitor process derivation

Let G denote the set of all pairs (m, e) in which

- m is a vertex $\langle d, b, w, g \rangle \in \mathcal{D}(G)$
- e is the adjacency set of all pairs (p, n) with
    - a vertex n adjacent to m
    - the list p of processes along the edge from m to n

Monitor process derivation

Let G denote the set of all pairs (m, e) in which

- m is a vertex $\langle d, b, w, g\rangle \in \mathscr{D}(G)$
- e is the adjacency set of all pairs (p, n) with
  - a vertex n adjacent to m
  - the list p of processes along the edge from m to n

Recurrences

Let $\hat{m} \in \{0 \ldots |G| - 1\}$ denote the lexicographic ordinal of a vertex m with respect to $\mathscr{D}(G)$. Then each vertex determines

$$(m, e)$$

Recurrences

Let $\hat{m} \in \{0 \ldots |G| - 1\}$ denote the lexicographic ordinal of a vertex m with respect to $\mathscr{D}(G)$. Then each vertex determines

$$(m, \{(p, n) \ldots (q, r)\})$$

Let $\hat{m} \in \{0 \ldots |G| - 1\}$ denote the lexicographic ordinal of a vertex m with respect to $\mathscr{D}(G)$. Then each vertex determines

$$
\begin{aligned}
&(m, \{ \\
&\quad (\langle p_0 \ldots p_{|p|-1} \rangle, n), \\
&\quad\quad \vdots \\
&\quad (\langle q_0 \ldots q_{|q|-1} \rangle, r)\})
\end{aligned}
$$

Recurrences

Let $\hat{m} \in \{0 \ldots |G| - 1\}$ denote the lexicographic ordinal of a vertex m with respect to $\mathcal{D}(G)$. Then each vertex determines

$$
\begin{aligned}
H_{\hat{m}} \quad = \quad \lambda P.\, (F\ \mathsf{alt}\,) \langle & \\
& (F\ \mathsf{seq}\,) \langle p_0 \ldots p_{|p|-1}, P_{\hat{n}} \rangle, \\
& \qquad \vdots \\
& (F\ \mathsf{seq}\,) \langle q_0 \ldots q_{|q|-1}, P_{\hat{r}} \rangle \rangle
\end{aligned}
$$

Recurrences

Let $\hat{m} \in \{0 \ldots |G| - 1\}$ denote the lexicographic ordinal of a vertex m with respect to $\mathcal{D}(G)$. Then each vertex determines

$$
\begin{aligned}
H_{\hat{m}} \quad = \quad \lambda P.\, (\digamma\ \mathsf{alt})\, \langle \\
(\digamma\ \mathsf{seq})\, (p \shortparallel \langle P_{\hat{n}} \rangle), \\
\vdots \\
(\digamma\ \mathsf{seq})\, (q \shortparallel \langle P_{\hat{r}} \rangle) \rangle
\end{aligned}
$$

Recurrences

Let $\hat{m} \in \{0 \ldots |G| - 1\}$ denote the lexicographic ordinal of a vertex m with respect to $\mathscr{D}(G)$. Then each vertex determines

$$H_{\hat{m}} \quad = \quad \lambda P.\ (\digamma\ \mathsf{alt})\left(\bigcup_{(p,n)\,\in\, e} \{(\digamma\,\mathsf{seq})\ (p \shortparallel \langle P_{\hat{n}} \rangle)\}\right)^{\circ -1}$$

Recurrences

Let $\hat{m} \in \{0 \ldots |G| - 1\}$ denote the lexicographic ordinal of a vertex m with respect to $\mathscr{D}(G)$. Then each vertex determines

$$H_{\hat{m}} \quad = \quad \lambda P.\ (\digamma\ \mathsf{alt}\,)\left(\bigcup_{(p,n)\,\in\, e} \{(\digamma\,\mathsf{seq}\,)\ (p \shortparallel \langle P_{\hat{n}} \rangle)\}\right)^{\circ -1}$$

in a system $H \in (\mathbb{D}^{|G|} \to \mathbb{D})^{|G|}$ of recurrences solvable for a fixed point $X \in \mathbb{D}^{|G|}$ such that for all $i < |H|$

$$X_i \equiv H_i\ X.$$

Recurrences

Let $\hat{m} \in \{0 \ldots |G| - 1\}$ denote the lexicographic ordinal of a vertex m with respect to $\mathscr{D}(G)$. Then each vertex determines

$$H_{\hat{m}} \quad = \quad \lambda P.\ (\digamma\ \mathsf{alt})\left(\bigcup_{(p,n)\,\in\, e}\{(\digamma\ \mathsf{seq})\ (p \shortparallel \langle P_{\hat{n}}\rangle)\}\right)^{\circ-1}$$

in a system $H \in (\mathbb{D}^{|G|} \to \mathbb{D})^{|G|}$ of recurrences solvable for a fixed point $X \in \mathbb{D}^{|G|}$ such that for all $i < |H|$

$$X_i \equiv H_i\ X.$$

The monitor process (to be synthesized) is the initial term X_0.

Solving for a fixed point

For a list of recurrences $H = \langle H_0 \ldots H_{|H|-1} \rangle \in (\mathbb{D}^* \to \mathbb{D})^*$ derive

$$H \circlearrowleft i = \langle H_i \ldots H_{|H|-1} \rangle \mathbin{+\!\!+} \langle H_1 \ldots H_{i-1} \rangle$$

from H by deleting H_0 and rolling H_i to the head. Let

$$H' = \langle H \circlearrowleft 1, H \circlearrowleft 2, \ldots, H \circlearrowleft |H| - 1 \rangle$$

denote the list of values of $H \circlearrowleft i$ for $0 < i < |H|$. Then

$$\dot{\Upsilon}(H) = \begin{cases} \text{fix } \lambda p.\ H_0\langle p \rangle & \text{if } |H| = 1 \\ \text{fix } \lambda p.\ H_0(p : \dot{\Upsilon}\ (\lambda h.\ h \circ \lambda q.\ p : q)^\star\ H') & \text{otherwise} \end{cases}$$

yields the first process in the fixed point of H. The recursion terminates because $|H'| = |H \circlearrowleft i| = |H| - 1$ holds.

The end game

$$\textsc{Dms}(x) = \begin{cases} \textsc{Sbs}\ x & \text{if } \|x\| < K_s \\ (\Omega\ \textsc{Dms})\ \mho\ x & \text{otherwise} \end{cases}$$

- $\textsc{Dms}$ recursively synthesizes a circuit by direct mapping.
- $\textsc{Sbs}\ x$ is the state based synthetic form of a process x.
- $\|x\|$ is a proxy for the cost of computing $\textsc{Sbs}\ x$.
- K_s is the (freely adjustable!) cost budget.
- $\mho\ x$ is the process x decomposed into smaller parts.
- $(\Omega\ f)\ y$ is the combined effect of f on each part of y.

The end game

$$\textsc{Dms}(x) = \begin{cases} \textsc{Sbs}\ x & \text{if } \|x\| < K_s \\ (\Omega\ \textsc{Dms})\ \mho\ x & \text{otherwise} \end{cases}$$

- DMS recursively synthesizes a circuit by direct mapping.
- SBS x is the state based synthetic form of a process x.
- $\|x\|$ is a proxy for the cost of computing SBS x.
- K_s is the (freely adjustable!) cost budget.
- $\mho\ x$ is the process x decomposed into smaller parts.
- $(\Omega\ f)\ y$ is the combined effect of f on each part of y.

The end game

$$\textsc{Dms}(x) = \begin{cases} \textsc{Sbs}\ x & \text{if } \|x\| < K_s \\ (\Omega\ \textsc{Dms})\ \mho\ x & \text{otherwise} \end{cases}$$

- DMS recursively synthesizes a circuit by direct mapping.
- SBS x is the state based synthetic form of a process x.
- $\|x\|$ is a proxy for the cost of computing SBS x.
- K_s is the (freely adjustable!) cost budget.
- $\mho\ x$ is the process x decomposed into smaller parts.
- $(\Omega\ f)\ y$ is the combined effect of f on each part of y.

The end game

$$\textsc{Dms}(x) = \begin{cases} \textsc{Sbs}\ x & \text{if } \|x\| < K_s \\ (\Omega\ \textsc{Dms})\ \mho\ x & \text{otherwise} \end{cases}$$

- $\textsc{Dms}$ recursively synthesizes a circuit by direct mapping.
- $\textsc{Sbs}\, x$ is the state based synthetic form of a process x.
- $\|x\|$ is a proxy for the cost of computing $\textsc{Sbs}\ x$.
- K_s is the (freely adjustable!) cost budget.
- $\mho\ x$ is the process x decomposed into smaller parts.
- $(\Omega\ f)\ y$ is the combined effect of f on each part of y.

The end game

$$\textsc{Dms}(x) = \begin{cases} \textsc{Sbs}\ x & \text{if } \|x\| < K_s \\ (\Omega\ \textsc{Dms})\ \mho\ x & \text{otherwise} \end{cases}$$

- Dms recursively synthesizes a circuit by direct mapping.
- Sbs x is the state based synthetic form of a process x.
- $\|x\|$ is a proxy for the cost of computing Sbs x.
- K_s is the (freely adjustable!) cost budget.
- $\mho\ x$ is the process x decomposed into smaller parts.
- $(\Omega\ f)\ y$ is the combined effect of f on each part of y.

The end game

$$\textsc{Dms}(x) = \begin{cases} \textsc{Sbs}\ x & \text{if } \|x\| < K_s \\ (\Omega\ \textsc{Dms})\ \mho\ x & \text{otherwise} \end{cases}$$

- Dms recursively synthesizes a circuit by direct mapping.
- Sbs x is the state based synthetic form of a process x.
- $\|x\|$ is a proxy for the cost of computing Sbs x.
- K_s is the (freely adjustable!) cost budget.
- $\mho\ x$ is the process x decomposed into smaller parts.
- $(\Omega\ f)\ y$ is the combined effect of f on each part of y.

The end game

$$\textsc{Dms}(x) = \begin{cases} \textsc{Sbs}\ x & \text{if } \|x\| < K_s \\ (\Omega\ \textsc{Dms})\ \mho\ x & \text{otherwise} \end{cases}$$

- DMS recursively synthesizes a circuit by direct mapping.
- SBS x is the state based synthetic form of a process x.
- $\|x\|$ is a proxy for the cost of computing SBS x.
- K_s is the (freely adjustable!) cost budget.
- $\mho\ x$ is the process x decomposed into smaller parts.
- $(\Omega\ f)\ y$ is the combined effect of f on each part of y.

The end game

$$\textsc{Dms}(x) = \begin{cases} \textsc{Sbs}\ x & \text{if } \|x\| < K_s \\ (\Omega\ \textsc{Dms})\ \mho\ x & \text{otherwise} \end{cases}$$

- Dms recursively synthesizes a circuit by direct mapping.
- Sbs x is the state based synthetic form of a process x.
- $\|x\|$ is a proxy for the cost of computing Sbs x.
- K_s is the (freely adjustable!) cost budget.
- $\mho\ x$ is the process x decomposed into smaller parts.
- $(\Omega\ f)\ y$ is the combined effect of f on each part of y.

For example, let $\|x\|$ equal the number of places in the Petri net model of x with $K_s = 20$ for a cutoff under 2^{20} markings.

References

https://www.delayinsensitive.com

– full details on everything in this presentation

https://www.cs.upc.edu/~jordicf/gavina/BIB/files/lcpn04_synth.pdf

– on direct mapping synthesis

https://csl.yale.edu/~rajit/ps/aer.pdf

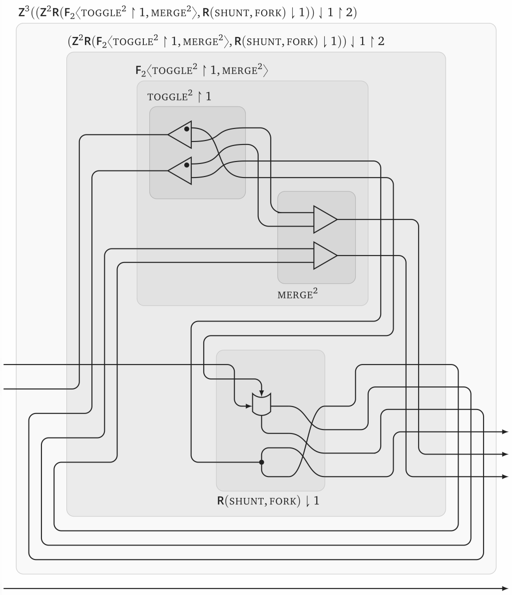
– on token rings and trees for neural networks

http://ccr.sigcomm.org/archive/1995/jan95/ccr-9501-nagle84.pdf

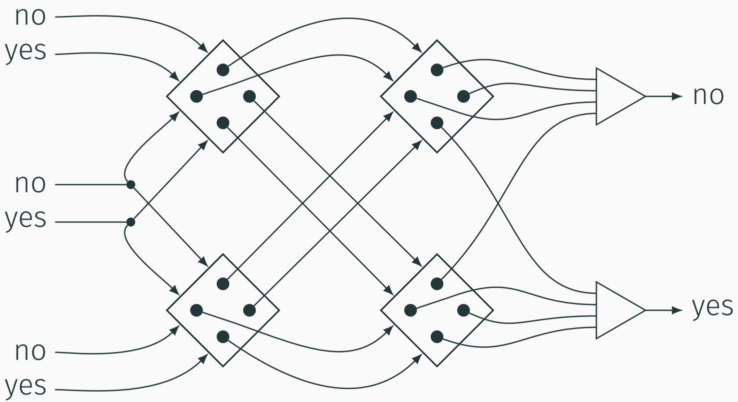
– on the small packet problem

Appendix

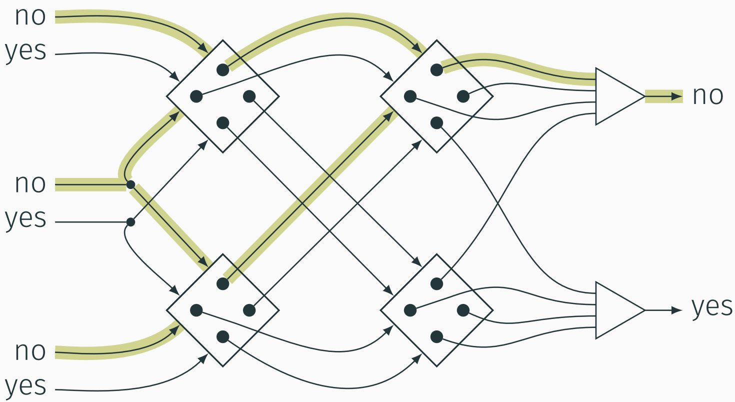
Network combining forms



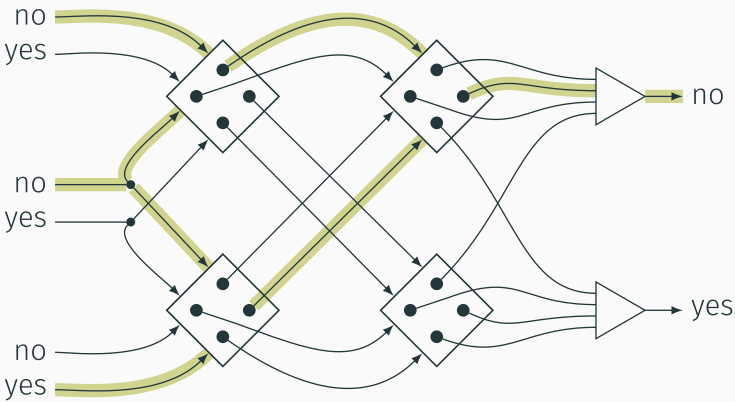
A delay insensitive three-way voting machine



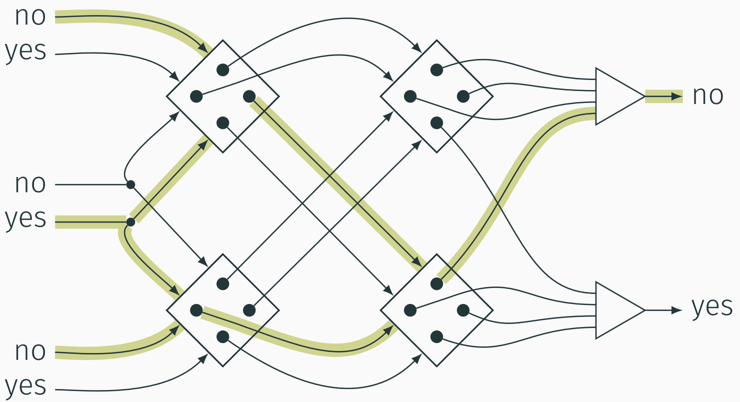
A delay insensitive three-way voting machine



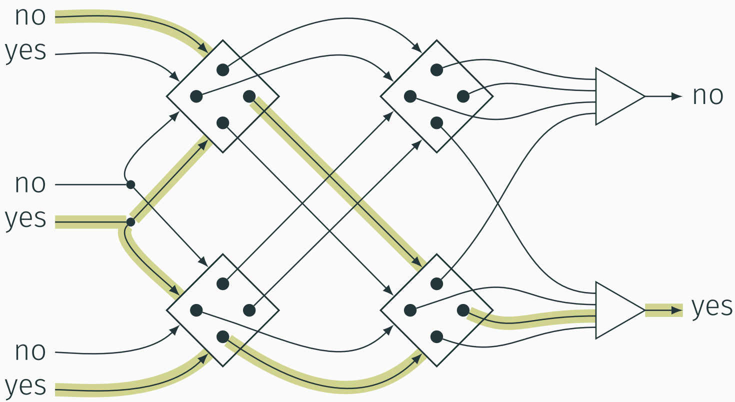
A delay insensitive three-way voting machine



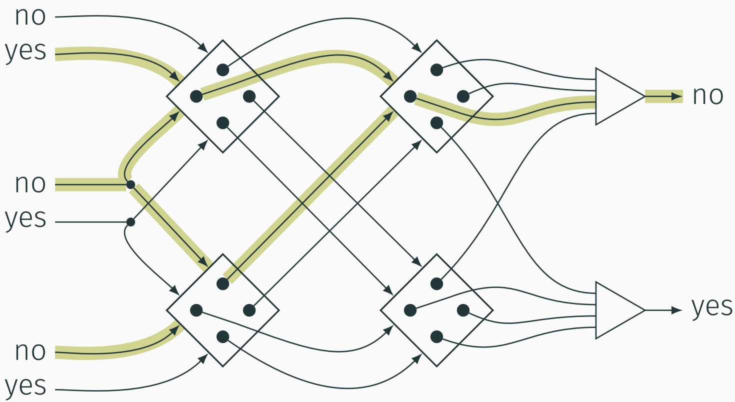
A delay insensitive three-way voting machine



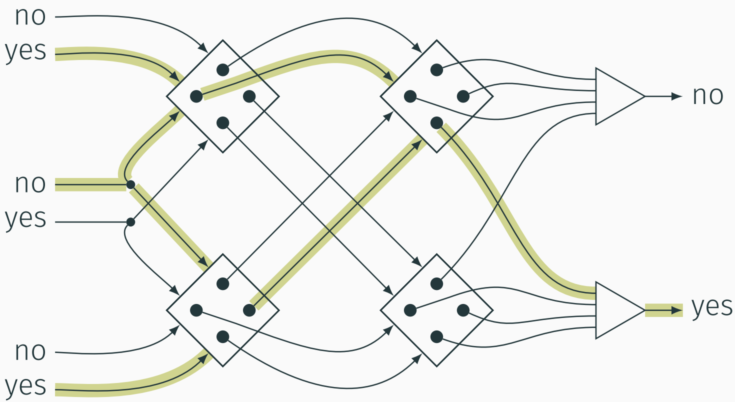
A delay insensitive three-way voting machine



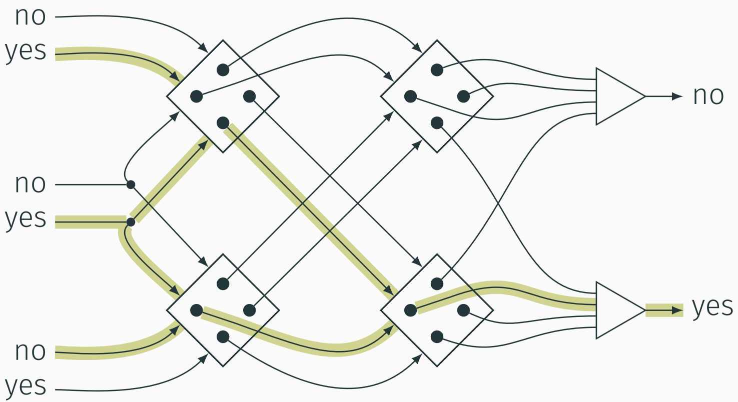
A delay insensitive three-way voting machine



A delay insensitive three-way voting machine



A delay insensitive three-way voting machine



A delay insensitive three-way voting machine