Synthesis and Control: a Distributed Perspective

Anca Muscholl

LaBRI Bordeaux and TUM-IAS Munich

joint work with I. Walukiewicz

Distributed Runtime Verification, Bertinoro 2016

Motivation

Computing paradigms

High performance

multi-core processors

mobile systems, robots,…

Data: intrinsically distributed

cloud

Concurrent programs

Hard to write…

shared data dependencies

synchronization side-effects

… and to verify

corner-case errors, testing has low coverage

traditional exploration: too many interleavings

Motivation

Error-free programs are rare…

Error recovery: accept the reality

Monitoring = knowledge for recovery

Control = recovery from bad states

Motivation

Specific challenges for concurrent programs:

states are distributed

error recovery (strategy) must be distributed

Question: what is “distributed”?

This talk:

Mazurkiewicz trace theory:

A mathematical framework for
distributed monitoring and control

Outline

- Motivating examples
- Mazurkiewicz traces
- Distributed monitoring
- Distributed control
- Conclusions

Programs

Multi-threaded programs with shared variables:

r(T,x)	thread T reads variable x
w(T,x)	thread T writes variable x

Synchronization operations:

Locks or atomic read/writes, e.g. CAS = compare-and-swap

Questions

97% of concurrency bugs (not counting deadlocks):

Learning from mistakes - a comprehensive study on real world concurrency bug characteristics [Lu et al., ASPLOS'08]

Order violations: unintended order between threads

e.g. race conditions

Atomicity violations: interference from other threads

Races

unprotected access to shared data

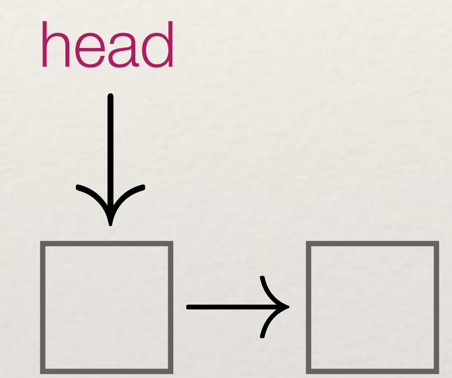
```
type list {int data; list *next}
list *head
```

Thread 1

```
1:  t1 = new(list);
2:  t1.data = 42;
3:  t1.next = head;
4:  acq(lock)
5:    head = t1
6:  rel(lock)
```

Thread 2

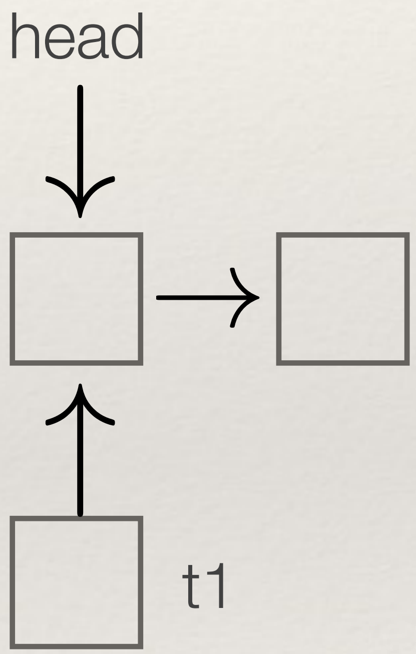
```
7:  t2 = head;
8:  acq(lock)
9:  head = head.next
10: rel(lock)
```



Races

type list {int data; list *next}
list *head

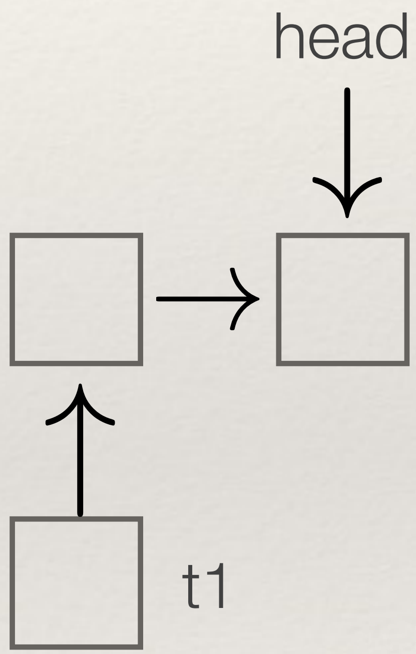
Thread 1	Thread 2
1: t1 = new(list);	7: t2 = head;
2: t1.data = 42;	8: acq(lock)
3: t1.next = head;	9: head = head.next
4: acq(lock)	10: rel(lock)
5: head = t1	
6: rel(lock)	



Races

type list {int data; list *next}
list *head

Thread 1	Thread 2
1: t1 = new(list);	7: t2 = head;
2: t1.data = 42;	8: acq(lock)
3: t1.next = head;	9: head = head.next
4: acq(lock)	10: rel(lock)
5: head = t1	
6: rel(lock)	



Races

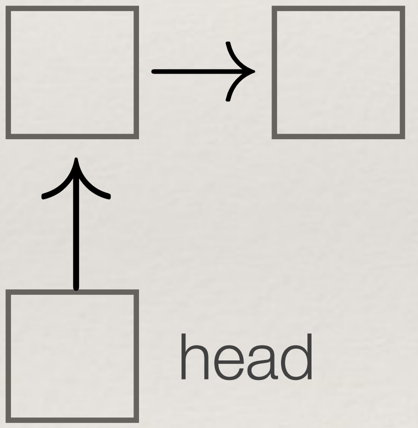
```
type list {int data; list *next}
list *head
```

Thread 1

```
1:  t1 = new(list);
2:  t1.data = 42;
3:  t1.next = head;
4:  acq(lock)
5:    head = t1
6:  rel(lock)
```

Thread 2

```
7:  t2 = head;
8:  acq(lock)
9:  head = head.next
10: rel(lock)
```



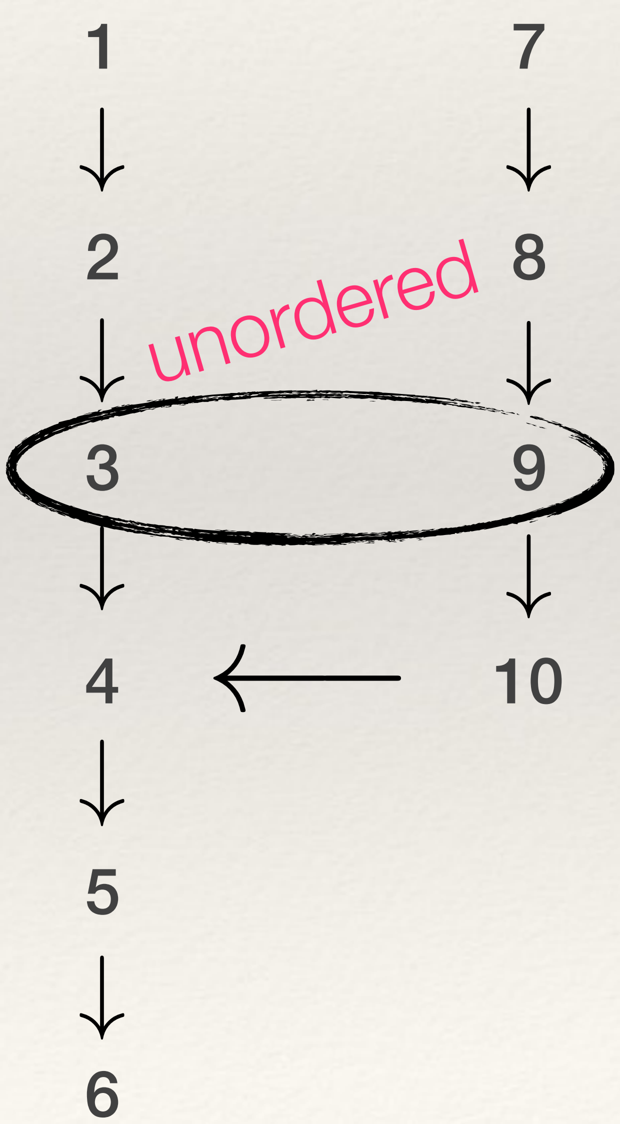
delete failed

What are races?

Happens-before order [Lamport,1978]

type list {int data; list *next}
list *head

Thread 1	Thread 2
1: t1 = new(list);	7: t2 = head;
2: t1.data = 42;	8: acq(lock)
3: t1.next = head;	9: head = head.next
4: acq(lock)	10: rel(lock)
5: head = t1	
6: rel(lock)	



Outline

- Motivating examples
- Mazurkiewicz traces
- Monitoring
- Control
- Conclusions

Traces [Mazurkiewicz 1977]

Finite alphabet of actions Σ

Conflict relation $D \subseteq \Sigma \times \Sigma$ (symmetric)

Word		Trace
total order $\leq$	→	partial order $\prec^*$
$a_1 a_2 \cdots a_n$		$i \prec j$ iff $i < j$
$1 < 2 < \cdots < n$		and a_i, a_j in conflict

Traces

type list {int data; list *next}
list *head

Thread 1

```
1:  t1 = new(list);
2:  t1.data = 42;
3:  t1.next = head;
4:  acq(lock)
5:    head = t1
6:  rel(lock)
```

Thread 2

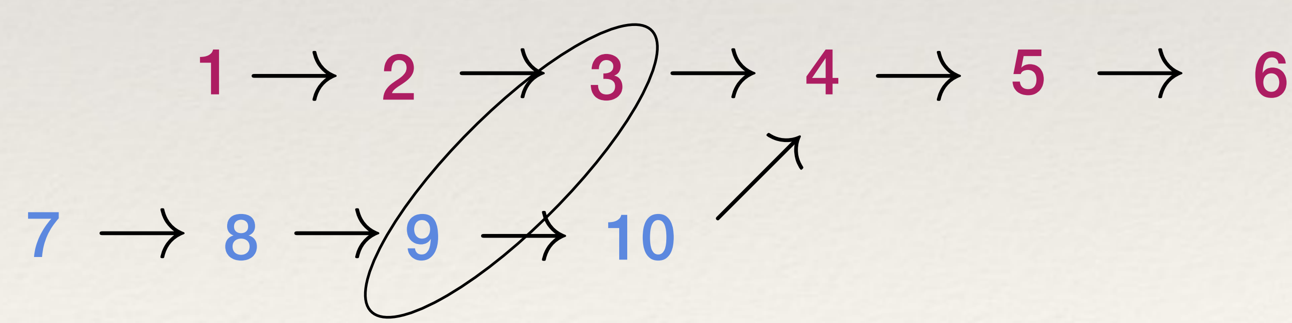
```
7:  t2 = head;
8:  acq(lock)
9:  head = head.next
10: rel(lock)
```

Word (total order)

1 2 3 7 8 9 10 4 5 6

1 2 7 8 9 3 10 4 5 6

Trace (partial order)



race

Conflict: same thread or same lock

Traces

Word (total order)

1 2 3 7 8 9 10 4 5 6

1 2 7 8 9 3 10 4 5 6

Trace (partial order)

$1 \to 2 \to 3 \to 4 \to 5 \to 6$

$7 \to 8 \to 9 \to 10 \nearrow$

Trace language:

all linearizations of a set of **trace** partial orders

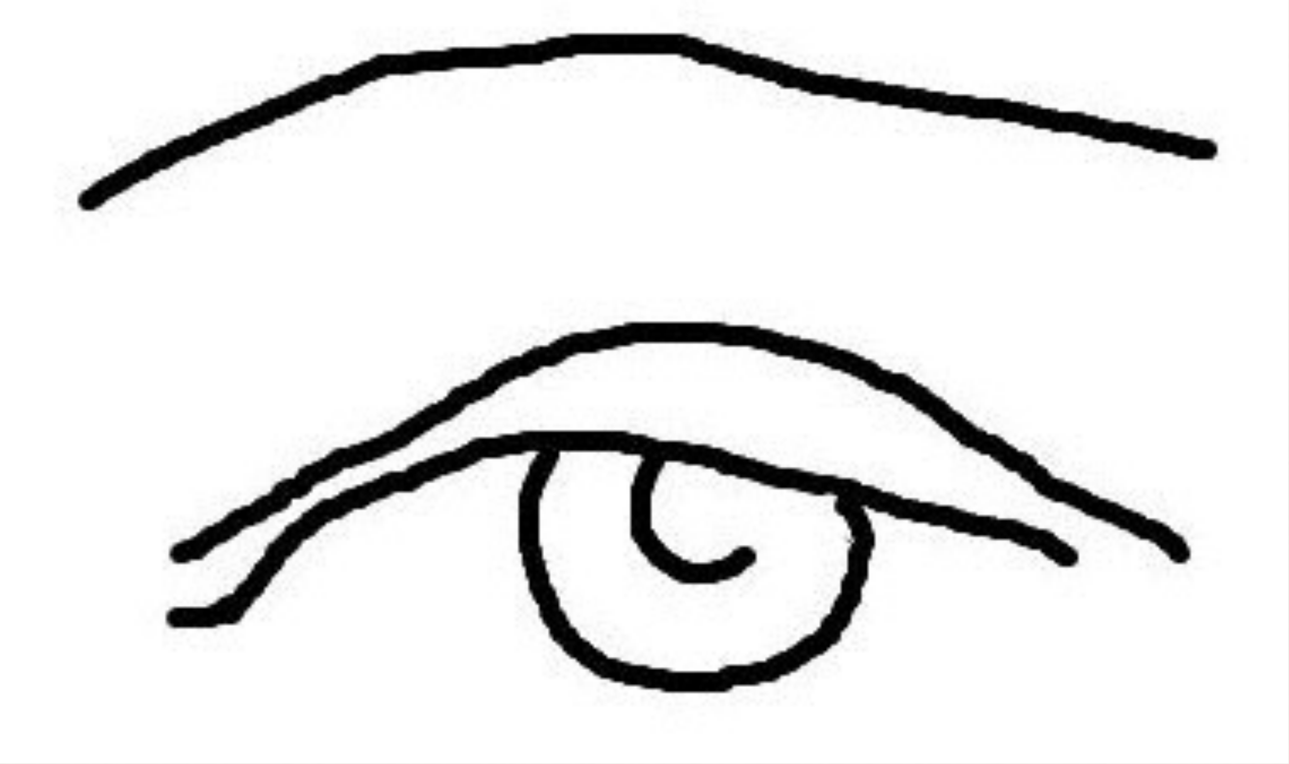
=

word language closed under $ab = ba$,

a, b not in conflict

Outline

- Motivating examples
- Mazurkiewicz traces
- Distributed monitoring
- Distributed control
- Conclusions



Monitors

- Synthesis of monitors

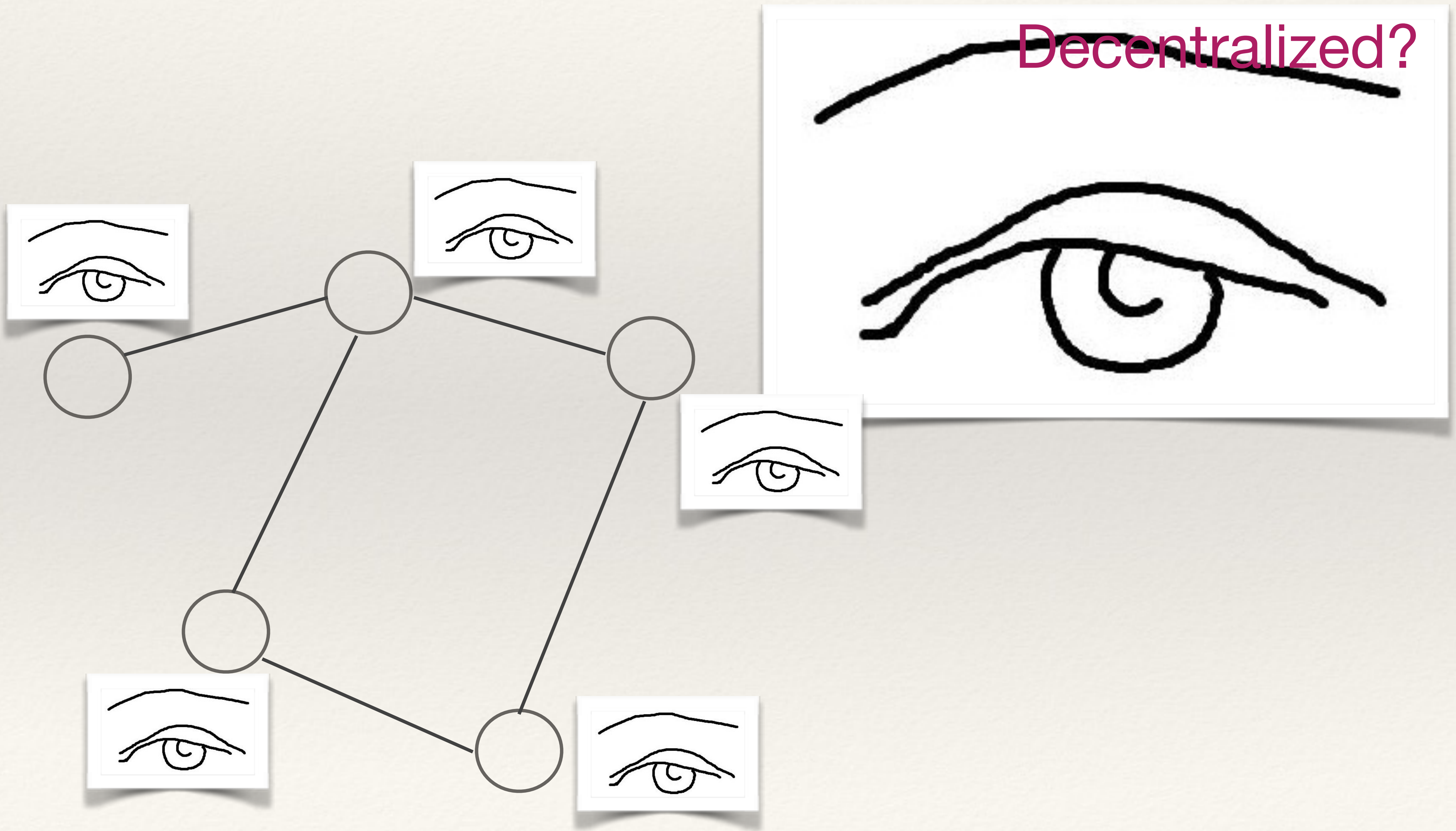
  turn a property into a monitor that should detect possible violations at runtime

- From monitoring to control

  monitor information towards error recovery

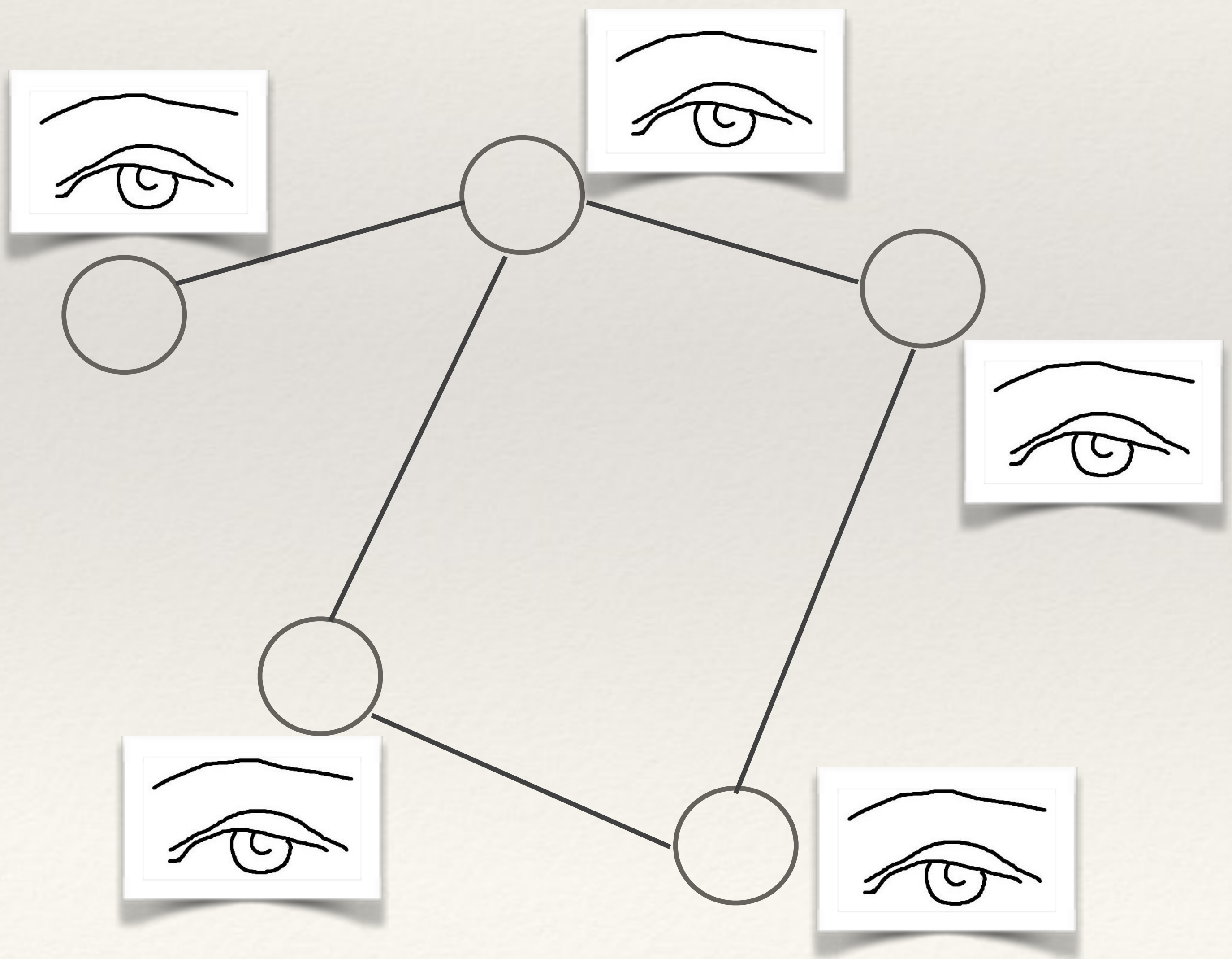
Monitoring

Centralized?



Monitoring

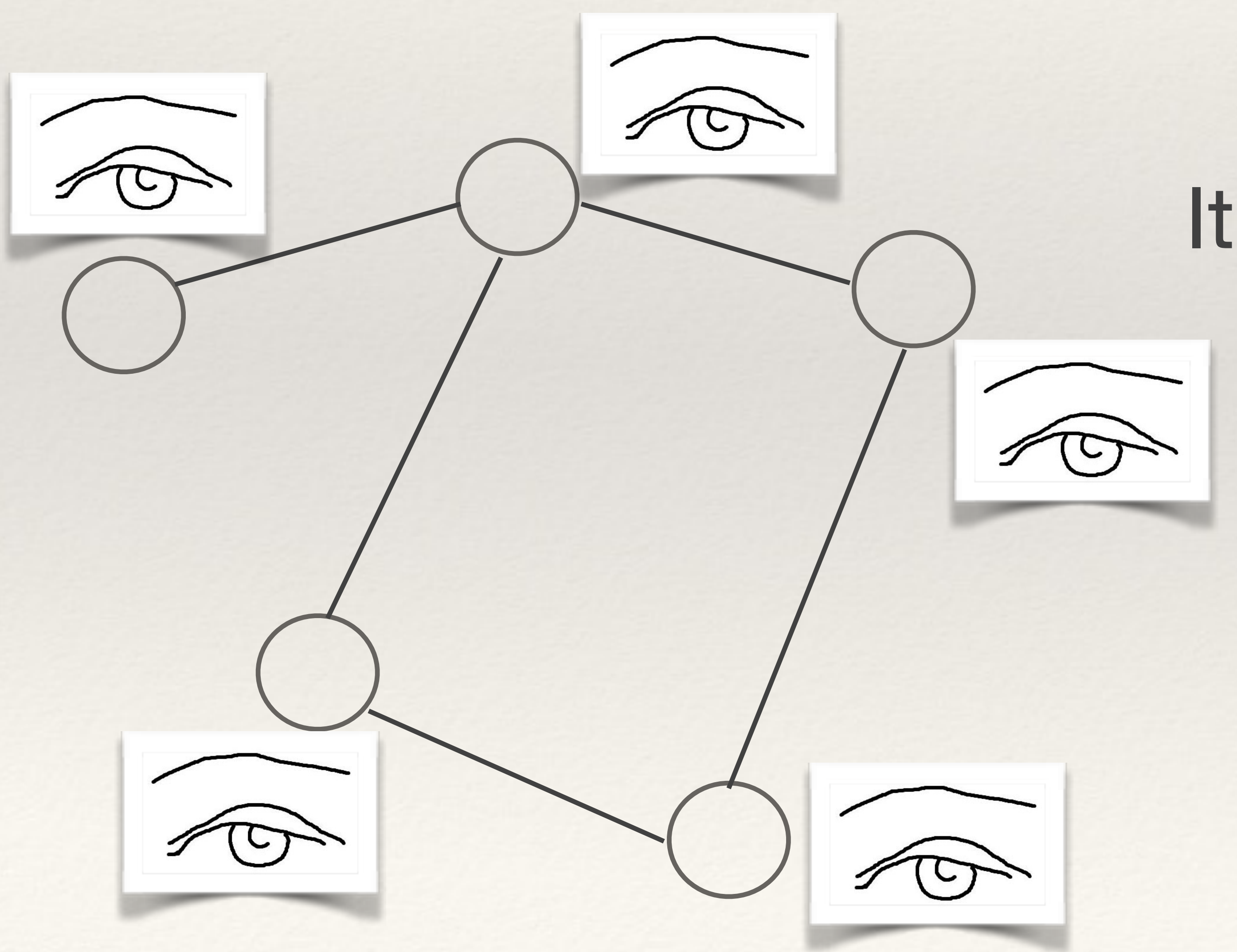
Decentralized!



no global synchronization required

localized failure detection and recovery: more efficient

Monitoring



Synchronization

between local monitors?

It depends on the architecture

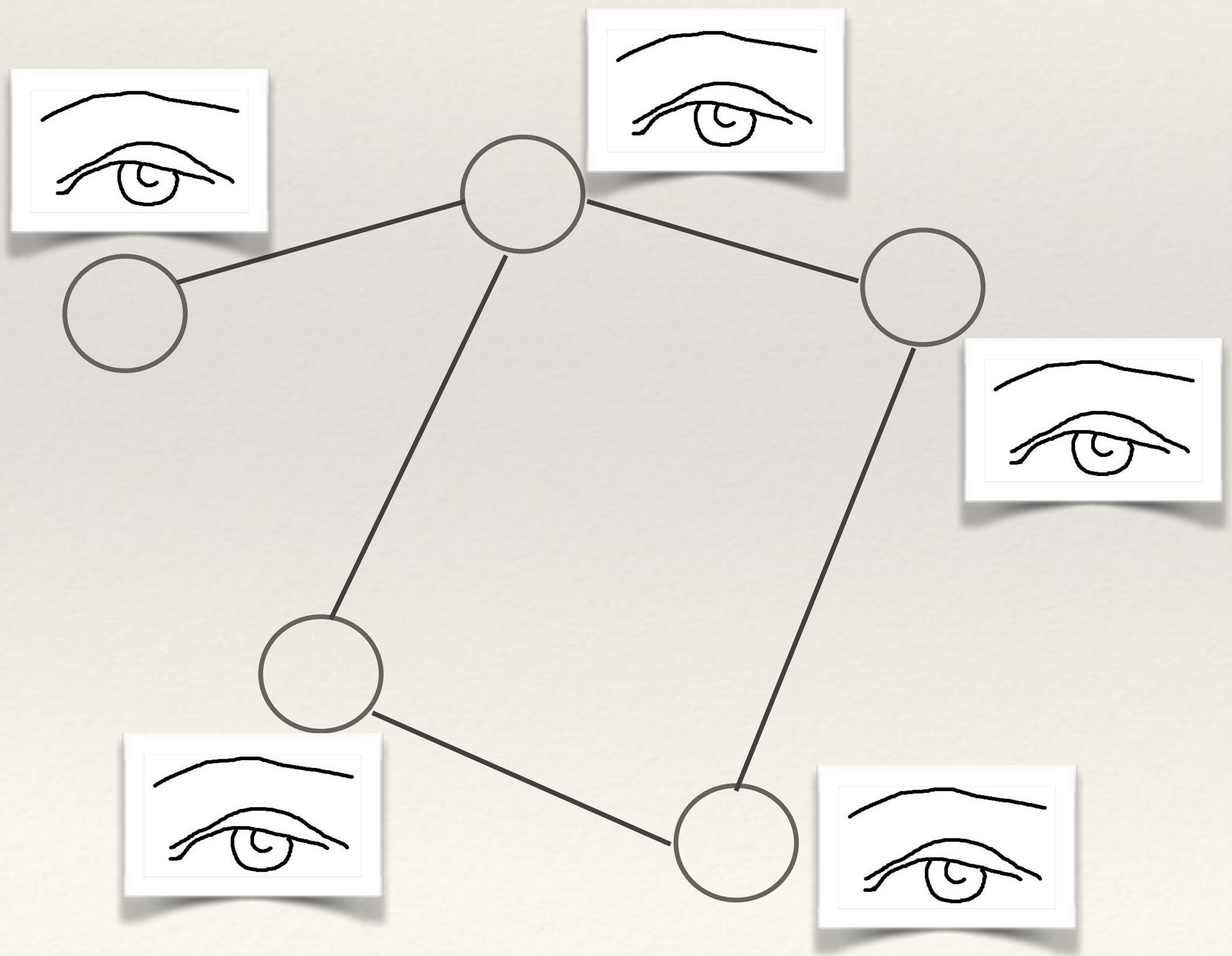
- communication channels
- shared memory

Monitors with shared memory: Zielonka automata

Zielonka???

Zielonka automata [Zielonka 1989]

monitor = finite-state automaton



synchronization:

shared actions +

exchange of information

Example

compare-and-swap CAS (T,x,old,new)

T is a thread, x a shared variable

if the value of x is old, then replace it by new;

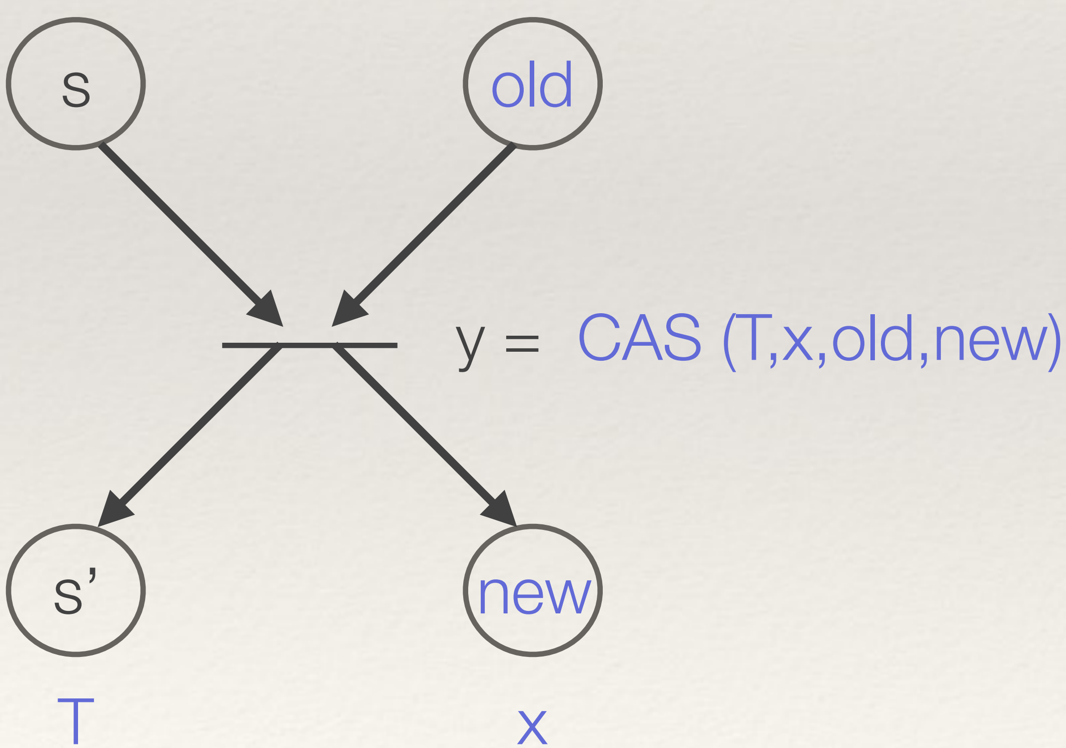
otherwise do nothing

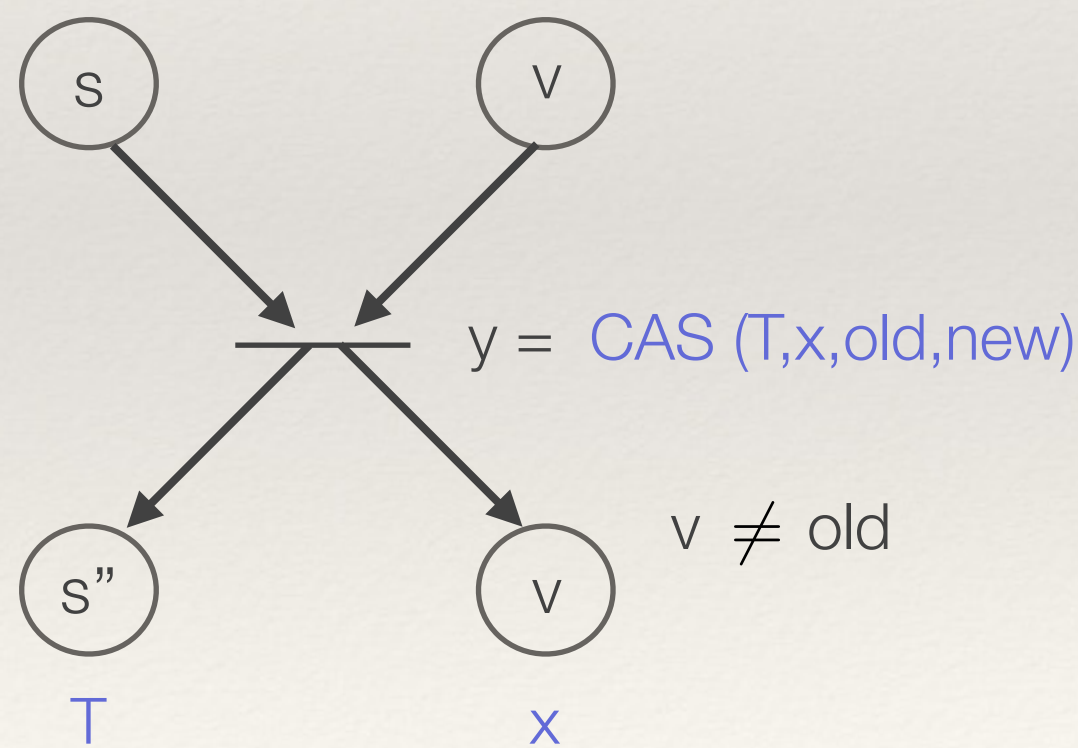
returns 1 if the value was replaced, 0 otherwise

Example

compare-and-swap CAS (T,x,old,new)

> if the value of x is old, then replace it by new; otherwise do nothing





Zielonka automaton

- fixed set of processes $\mathcal{P}$
- one finite-state automaton per process
- each shared action a synchronizes a fixed set of processes:

$$\mathrm{dom}(a) \subseteq \mathcal{P}$$

Upon executing a , processes in $\mathrm{dom}(a)$ exchange information about their states.

Zielonka's theorem

Construction of deterministic Zielonka automaton for every regular trace language.

[Zielonka, 1989]

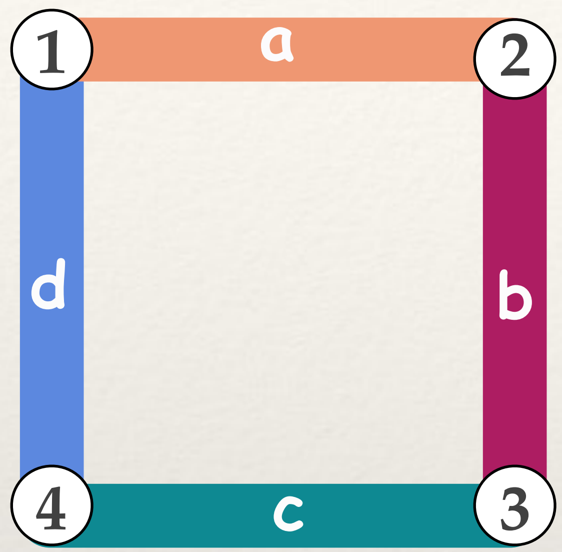
Crux: finite gossiping

Complexity: from several exponentials down to

From a DFA of size **s**, constructs equivalent Zielonka automaton on **p** processes with $4^{p^4} \cdot s^{p^2}$ states.

[Genest-Gimbert-M.-Walukiewicz, 2010]

Zielonka construction



$$\mathrm{dom}(a) = \{1, 2\}$$

$$\mathrm{dom}(b) = \{2, 3\}$$

$$\mathrm{dom}(c) = \{3, 4\}$$

$$\mathrm{dom}(d) = \{4, 1\}$$

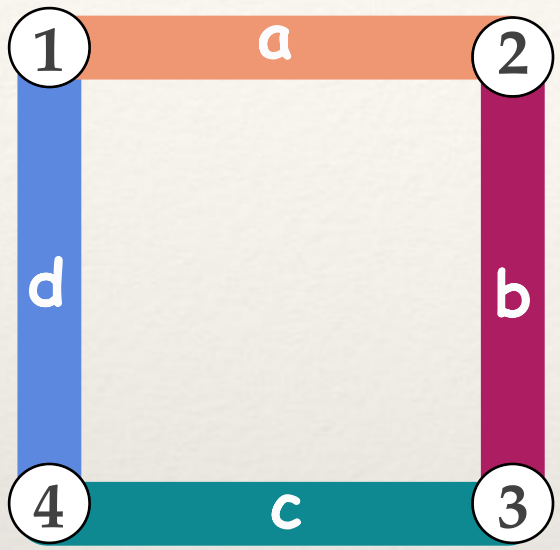
$$[(a + c)(b + d)]^*$$

$$a \longrightarrow d \longrightarrow c \longrightarrow d \longrightarrow a \longrightarrow b$$

$$a \searrow {\color{red} d}$$

not allowed

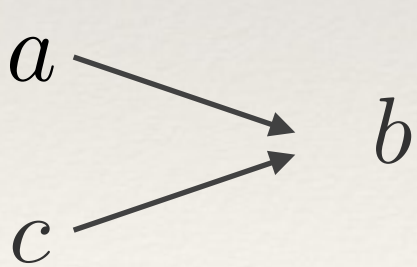
Zielonka construction



$$[(a+c)(b+d)]^*$$

What should each process remember?

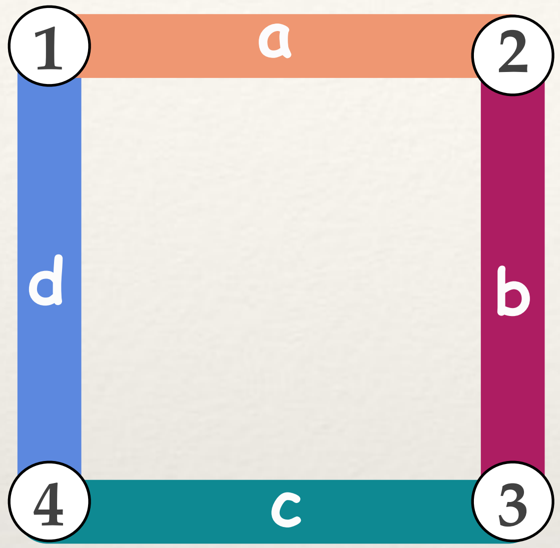
Each process remembers its last action: not sufficient



process 2: b (after a)

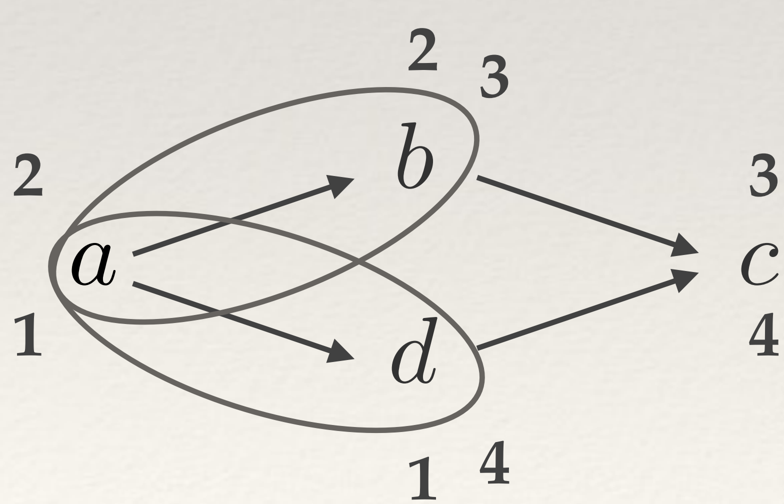
process 3: b (after c)

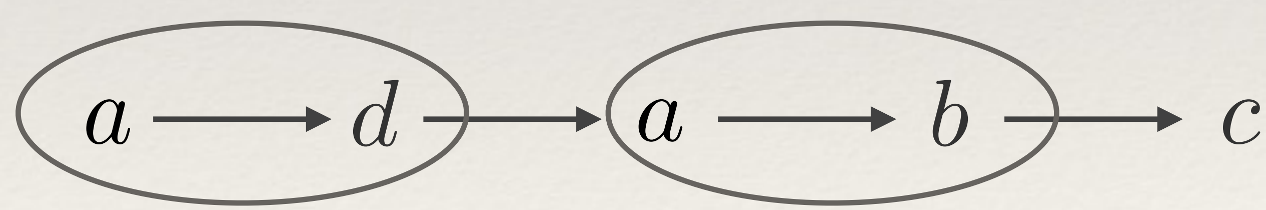
Zielonka construction



$$[(a+c)(b+d)]^*$$

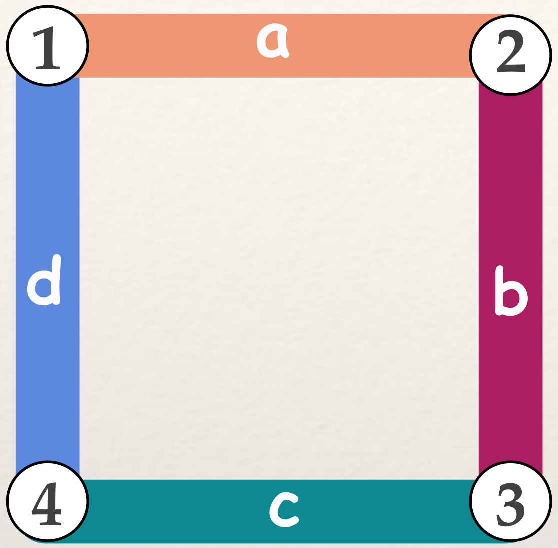
Each process remembers its last action and the last action of its synchronization partner.





Not sufficient.

Gossip



$$[(a+c)(b+d)]^*$$

Solution here: each process keeps the order of the last occurrences of all symbols.

… as complicated as …

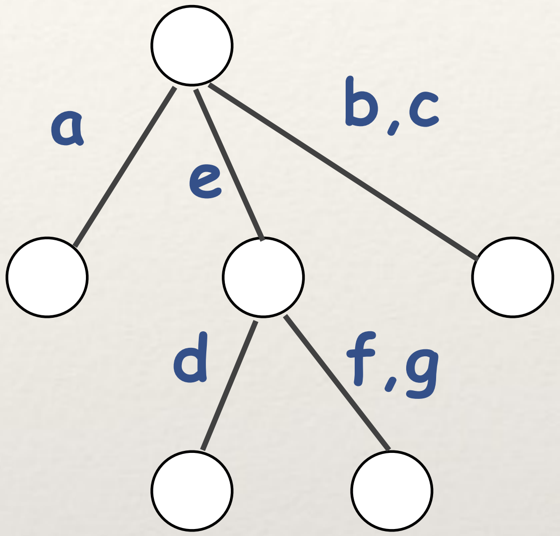
General solution: each process keeps latest information about other processes (gossip).

Gossip = kind of vector clock algorithm, but finite.

Zielonka going practical

Construction of deterministic Zielonka automata for regular trace languages: quadratic-time when the communication between processes is acyclic.

[Krishna-M., 2013]

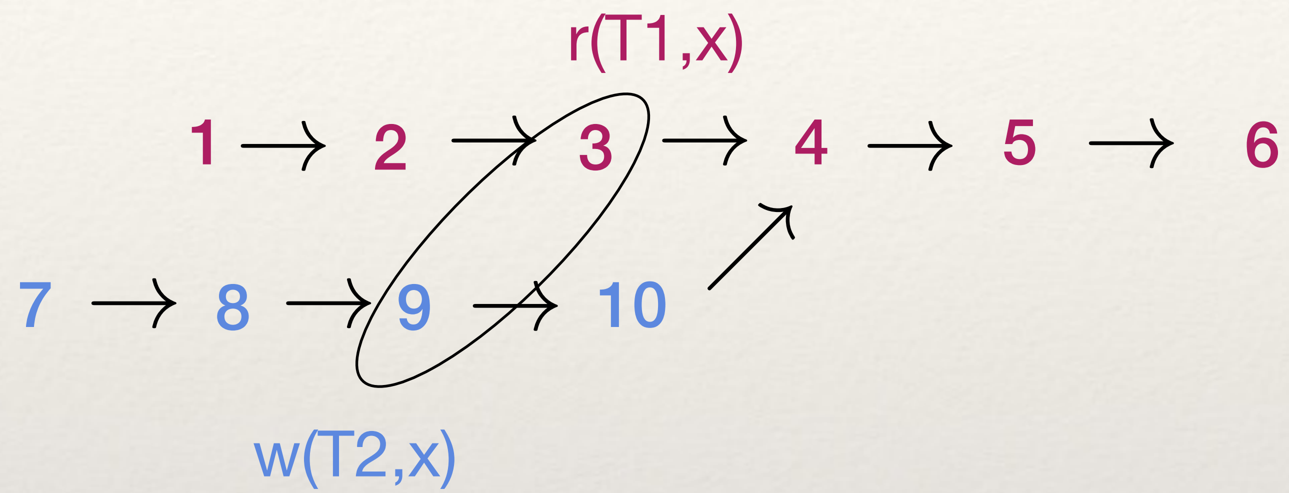


Is quadratic-time good enough?

Not if we start with very large DFA.

Zielonka going practical

Example: races



Apply Zielonka to properties stated by patterns.

Pattern =
DAG with vertices corresponding to monitored actions

Zielonka going practical

Processes: thread1, thread2, lock

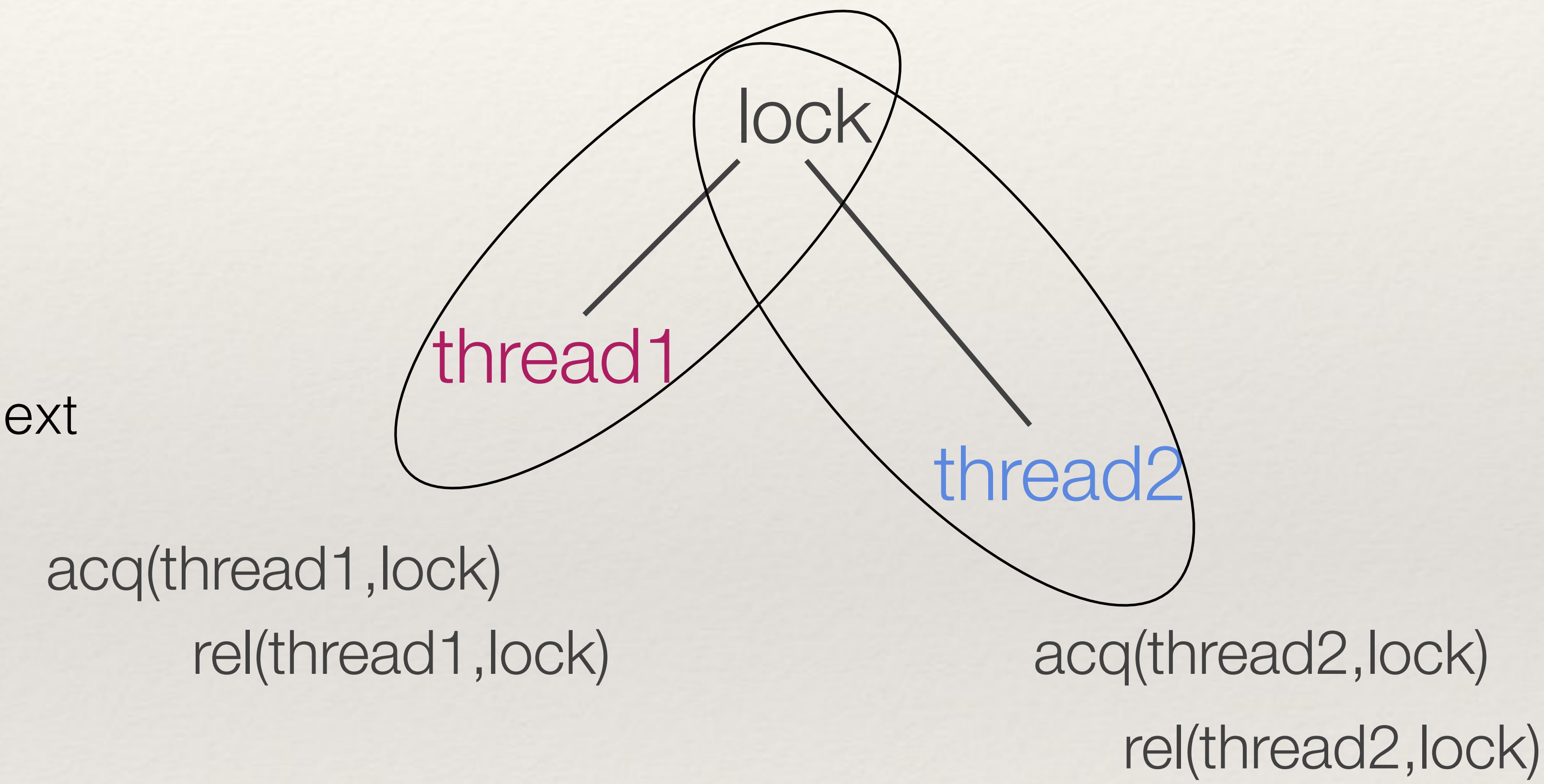
```
type list {int data; list *next}
list *head
```

Thread 1

```
1:  t1 = new(list);
2:  t1.data = 42;
3:  t1.next = head;
4:  acq(lock)
5:    head = t1
6:  rel(lock)
```

Thread 2

```
7:  t2 = head;
8:  acq(lock)
9:  head = head.next
10: rel(lock)
```



Efficient race monitoring if communication is acyclic.

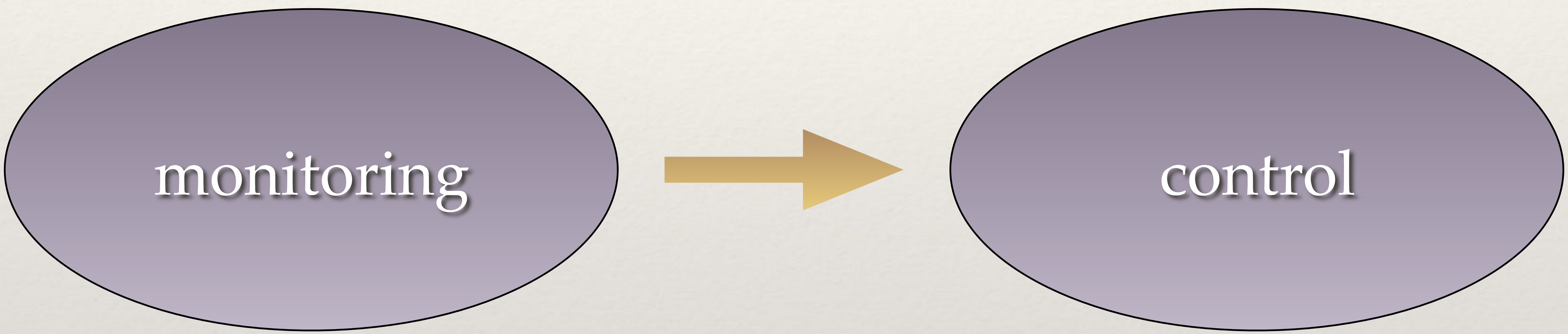
Monitoring: summary

- Zielonka's construction: general-purpose solution for distributed runtime monitoring with finite memory.
- The general construction requires polynomial memory on each process.
- Lightweight construction for acyclic architectures.
- Efficiency depends on what we start with. (Boolean combinations of) patterns are better than DFA.
- Zielonka's construction: also used for other kinds of synchronization, e.g. messages.

Outline

- Motivating examples
- Mazurkiewicz traces
- Distributed monitoring
- Distributed control
  - Pnueli-Rosner
  - Ramadge-Wonham
- Conclusion

Error recovery: control



error detection

restrict program execution

prevent error propagation

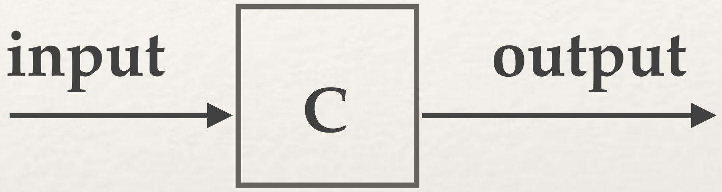
…

Control: basics

Church (1957)

Given: specification $S \subseteq \{0,1\}^\omega \times \{0,1\}^\omega$

relating inputs/outputs



C: device reacting continuously

Compute **C** that satisfies S

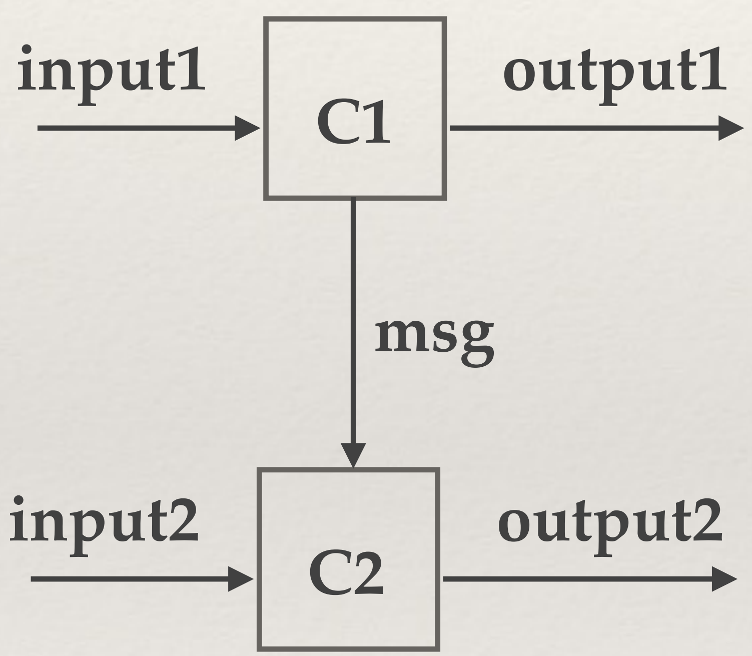
function $C : \{0,1\}^* \to \{0,1\}$

for all $w \in \{0,1\}^\omega : \quad (w, C(w)) \in S$

[Rabin, 1972] If S is omega-regular the existence of a controller can be decided. If "yes", then there exists a finite-state controller **C**.

Distributed control

[Pnueli-Rosner, 1990] Distributed reactive systems are hard to synthesize.



synchronous behavior

input1	0	0	1	1
msg		0	0	1
input2	1	0	0	0
output1		1	1	0
output2		1	1	0

Given an architecture and an omega-regular specification S, construct controllers C1, C2,… such that all controlled behaviors belong to S.

Pnueli-Rosner

Distributed synthesis is in general undecidable. It is decidable iff the architecture is a pipeline (non-elementary complexity).

[Pnueli-Rosner, 1990]

Undecidability comes from partial information:

[Peterson-Reif,1979] Multi-person alternation

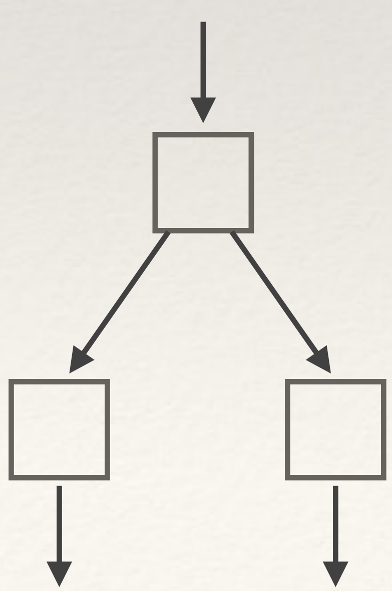
Basic problem: Specification may talk about inputs that are not seen by some process.

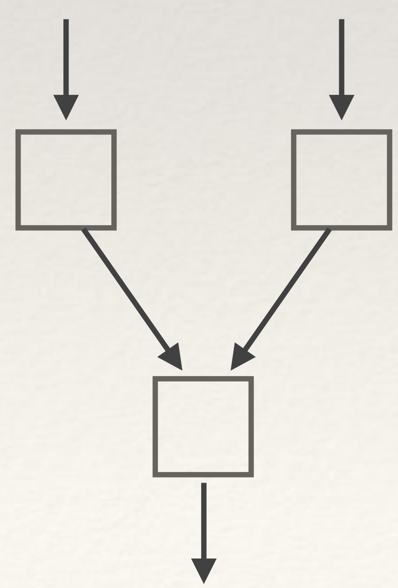
Pnueli-Rosner

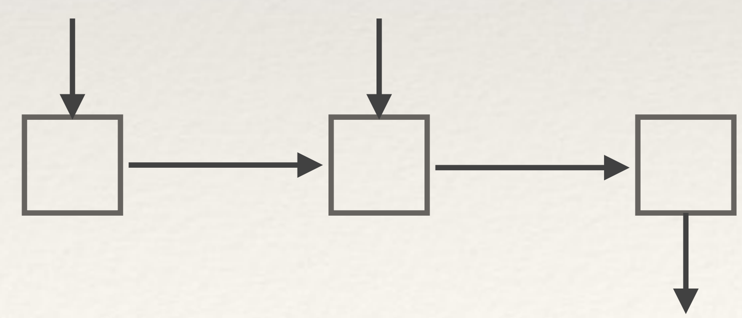
Undecidability: a problem with specifications?

Local specifications: conjunction of requirements on each process.

For local specifications, the only decidable architecture is basically the pipeline. [Madhusudan-Thiagarajan, 2001]







undecidable

Pnueli-Rosner : summary

- Simple extension of the Church setting
- Very few cases are decidable
- Even local specifications do not help
- Unsolved problem: define specifications that are compatible with the architecture
- Open: decidable architectures when controllers can add information to messages

[Kupferman-Vardi, 2001] Synthesizing distributed systems

[Finkbeiner-Schewe, 2005] Uniform distributed synthesis

[Gastin-Sznajder, 2013] External specifications

Outline

- Motivating examples
- Mazurkiewicz traces
- Distributed monitoring
- Distributed control
  - Pnueli-Rosner
  - Ramadge-Wonham
- Conclusion

Distributed control: a different approach

Approach based on

- Zielonka automata
- Ramadge-Wonham control setting

[Ramadge-Wonham,1989] The control of discrete-event systems

- Given: Plant (automaton) ***A*** with controllable (system) actions and uncontrollable (environment) actions, and specification ***S***.
- Compute controller ***C*** such that ***A*** x ***C*** satisfies ***S***.
  C must allow all uncontrollable actions.

Zielonka automata

- fixed set of processes $\mathcal{P}$
- one finite-state automaton per process
- each shared action a is located on a fixed set of processes

$$\mathrm{dom}(a) \subseteq \mathcal{P}$$

Upon executing a , processes in $\mathrm{dom}(a)$ exchange information about their states.

Distributed control

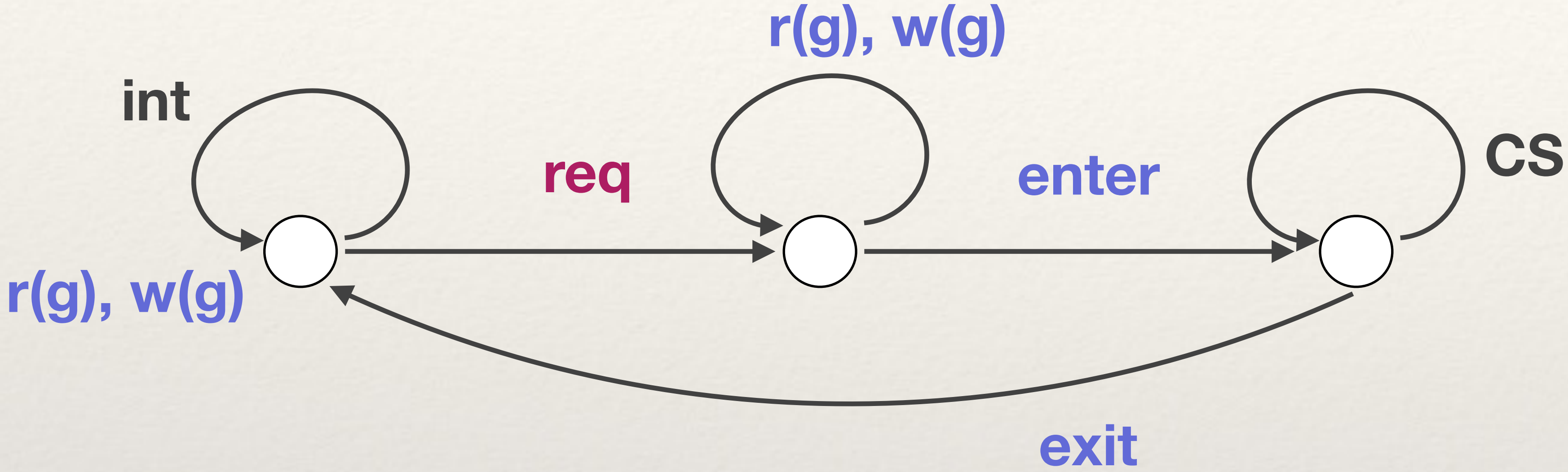
- Given: Zielonka automaton ***A*** with controllable (system) actions and uncontrollable (environment) actions, and specification ***S***.

- Compute another Zielonka automaton (controller) ***C*** such that ***A*** x ***C*** satisfies ***S***.

C must allow all uncontrollable actions.

Controllers are local and exchange information in order to satisfy *S*.

Distributed control and synthesis



mutual exclusion

requests **req**: uncontrollable actions

mutual exclusion algorithm: distributed controller for n+1 processes

n processes + 1 process for the shared variable

Control & Zielonka automata

Setting:

- A controller for each process.
- Controller of process p can disallow some of p's controllable actions.
- Control decisions for p depend on information in the causal past of p, i.e., including communication with other controllers.

Decidability status is open. No hidden information.

Control & Zielonka automata

Some partial decidability results:

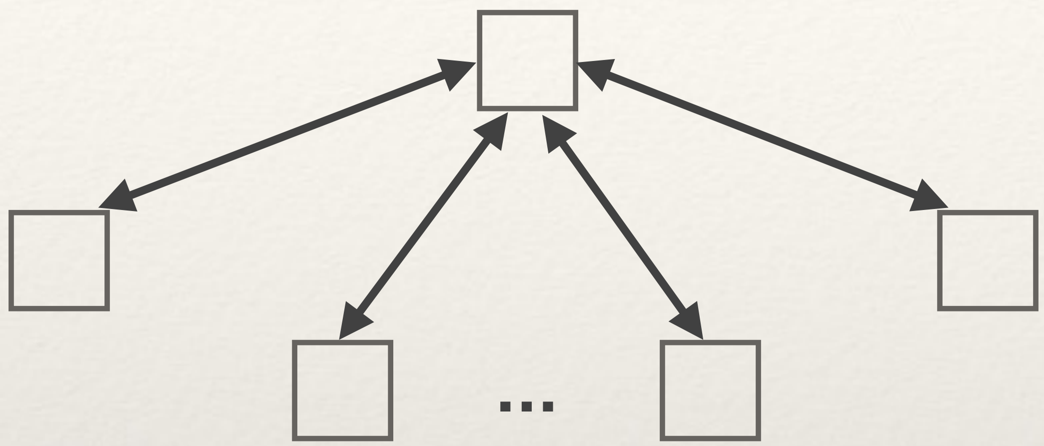
[Madhusudan-Thiagarajan-Yang, 2005] Decidability for restricted Zielonka automata: every process misses only bounded knowledge. MSO specifications.

[Gastin-Lerman-Zeitoun, 2004] Decidability for restricted automata: series-parallel systems. Reachability specifications.

[Genest-Gimbert-M-Walukiewicz, 2013] Decidability for acyclic process communication. Local reachability (blocking). Complexity is Tower(2,n)-complete.

[M-Walukiewicz, 2014] Decidability for acyclic process communication. Local parity specifications. Complexity is Tower(2,n)-complete.

Control & Zielonka automata



Control is decidable (EXPTIME-complete).

This architecture is undecidable in the Pnueli-Rosner setting.

Control & Zielonka automata

Why is control for Zielonka automata still open?

- Control for Zielonka automata reduces to satisfiability of monadic second-order (MSO) formula over the event structure of the automaton.
- These event structures can be rather complicated (grids), so MSO satisfiability is undecidable in general.

Control & Zielonka automata

words, trees…

…traces, event structures

Control & Zielonka automata: summary

- Simple extension of the Ramadge-Wonham control setting
- More cases are decidable: acyclic communication, local parity specifications
- Open: cyclic architectures
- Unsolved: understanding event structures of Zielonka automata

Conclusion

Error recovery for concurrent programs requires distributed monitoring and control

- Mazurkiewicz traces: automata-theoretic reasoning about concurrent programs is in reach.
- Distributed monitoring: synthesis of monitors through Zielonka construction.
- Distributed control: synthesis of controllers that can alter behavior.

Outlook

- Synthesis of distributed monitors using Lamport's vector clocks?
- Synthesis of distributed monitors if the set of (active) processes in the system is unknown?
- Our monitor/synthesis algorithm is conservative (no additional synchronizations). If we allow additional synchronizations, how should they be quantified?

Thank you!