Parameterized Verification of Distributed Shared-Memory Systems

Nicolas Waldburger

Journée D4, 05/10/23

From algorithms to automata-based models

Peterson's mutual exclusion algorithm:

For process $i \in \{0,1\}$:

while true:
do non-critical things ;
$\text{flag}_i = \text{true}$; $\text{turn} \coloneqq 1 - i$;
wait until $(\text{flag}_{1-i} == \text{false}$ *or* $\text{turn} == i)$
do critical things;
$\text{flag}_\text{i} = \text{false}$;

Correctness = the processes are not in their critical section simultaneously

Algorithm from
Peterson, G.: *Myths about the Mutuel Exclusion Problem.* Information Processing Letters, 1981

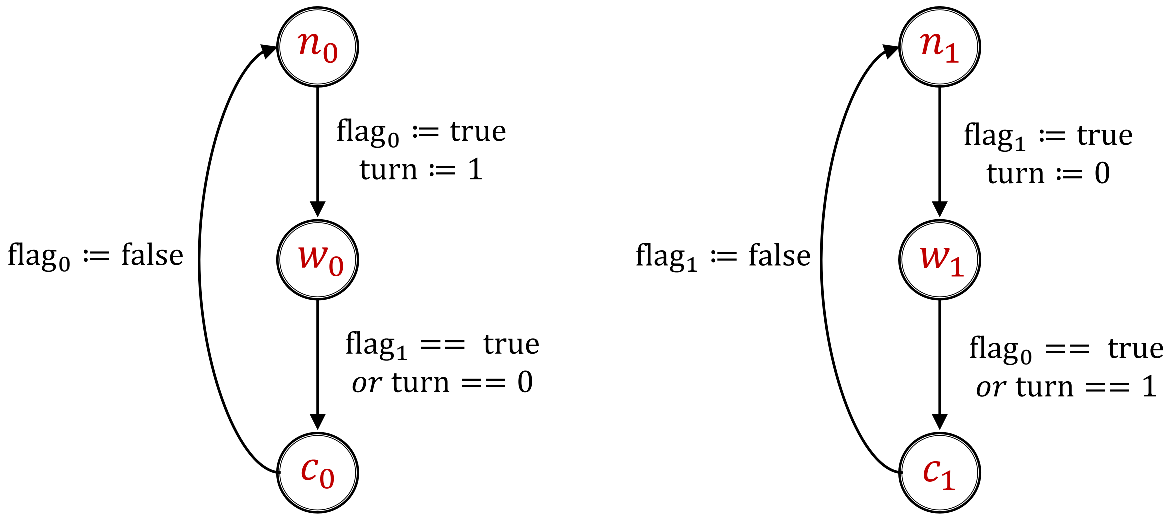
From algorithms to automata-based models

Peterson's mutual exclusion algorithm:

For process $i \in \{0,1\}$:

while true:
n_i do non-critical things ;
flag$_i$ = true ; turn $:= 1 - i$;
w_i **wait until** (flag$_{1-i}$ == false *or* turn == i)
c_i do critical things;
flag$_\mathrm{i}$ = false ;

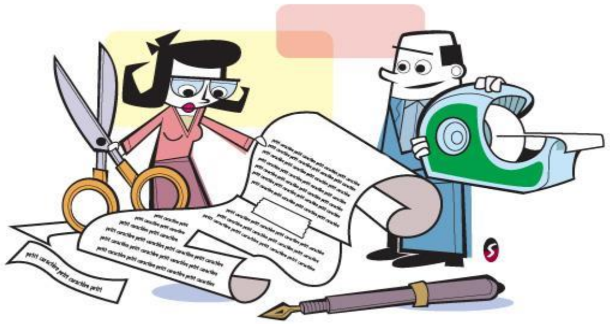
Automata-based model



Correctness = the processes are not in their critical section simultaneously
= c_0 and c_1 cannot be covered simultaneously

Algorithm from
Peterson, G.: *Myths about the Mutuel Exclusion Problem.* Information Processing Letters, 1981

Model Checking

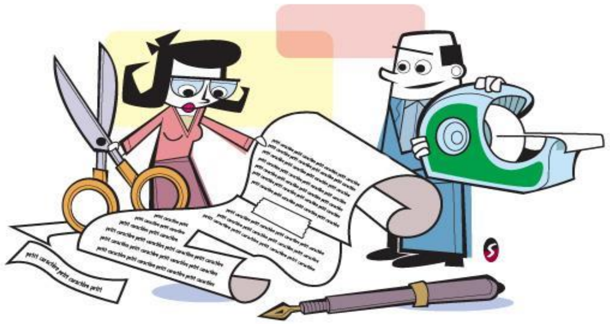
Does  satisfy  ?

distributed system

requirement

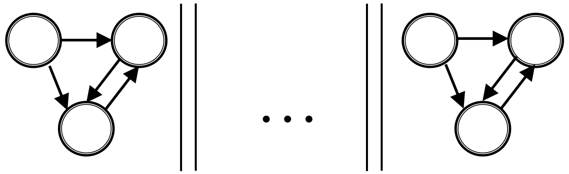
Many thanks to Nathalie Bertrand for this slide

Model Checking

Does  satisfy  ?

distributed system

requirement

 $\models$ $\mathbf{AG}(\neg c_0 \vee \neg c_1)$?

model

model-checking algorithm

property

Many thanks to Nathalie Bertrand for this slide

Issues with traditional Model Checking

“**Traditional**” model checking: describe behavior of each process separately

⇒ fix number of processes beforehand

Issues with traditional Model Checking

“**Traditional**” model checking: describe behavior of each process separately
⇒ fix number of processes beforehand

- Scalability issue when the size of the system is large

Issues with traditional Model Checking

“**Traditional**” model checking: describe behavior of each process separately
⇒ fix number of processes beforehand

- Scalability issue when the size of the system is large

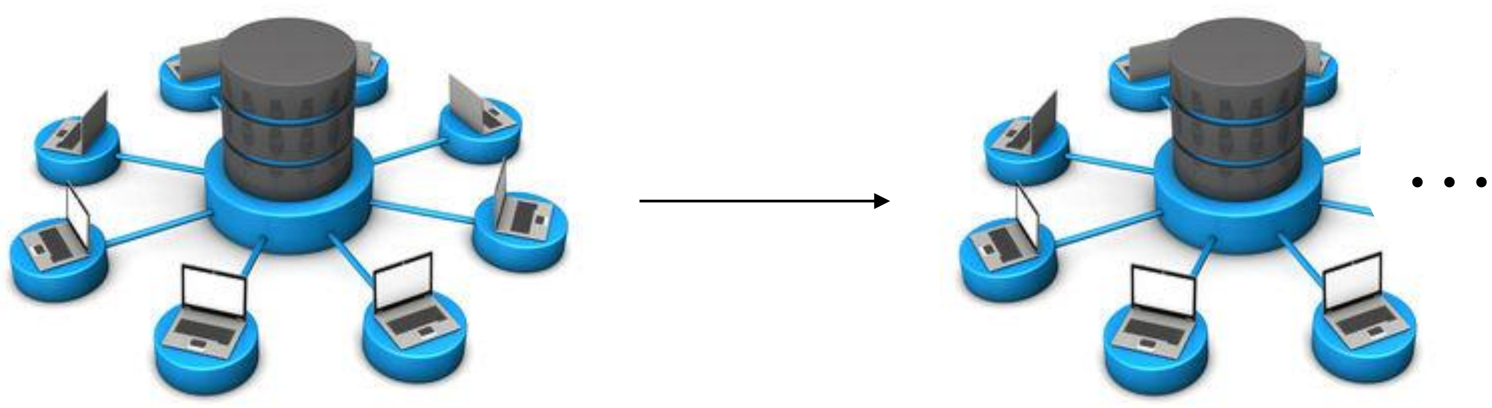
- What if I don’t know the number of agents beforehand ?

Issues with traditional Model Checking

"**Traditional**" model checking: describe behavior of each process separately
⇒ fix number of processes beforehand

- Scalability issue when the size of the system is large
- What if I don't know the number of agents beforehand ?
- Often undecidable problems…

Parameterized Verification



- Parameterized system = the number of participants is not fixed in advance
- System must be correct for any number of participants

  → New techniques than can be more efficient on large systems !

Many possible models

Many possible models

- We could say that all processes are identical, or that there is one leader and all others are followers

Many possible models

- We could say that all processes are identical, or that there is one leader and all others are followers
- How much computing power for a given process? Finite-state machines, pushdown machines, access to private variables…

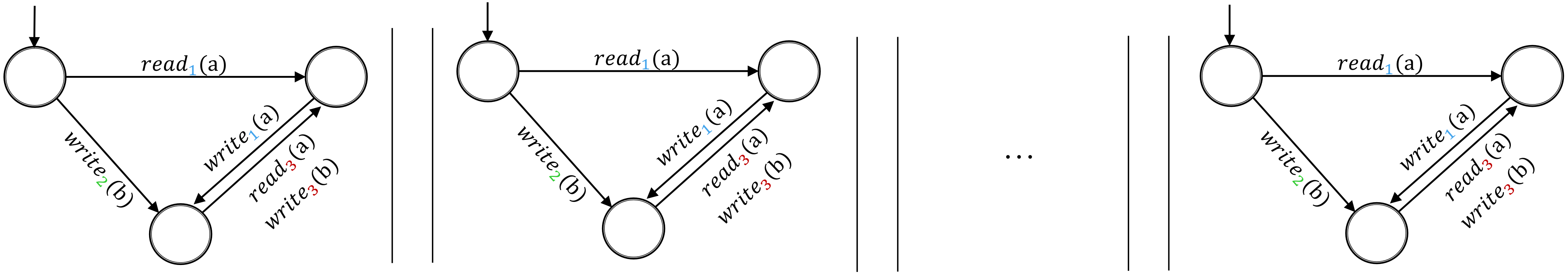
Many possible models

- We could say that all processes are identical, or that there is one leader and all others are followers
- How much computing power for a given process? Finite-state machines, pushdown machines, access to private variables…
- Means of communication:

Communication primitive	*Rendez-vous*	*Broadcast*	*Shared memory*	…
				
	two processes must synchronize	*a process sends a messages to its neighbors*	*a process reads from the shared memory or writes to the shared memory*	

Let’s focus on shared-memory systems

From now on, all processes are identical and described by a simple finite-state machine (no stack, no private memory…) where transitions interact with the shared memory.

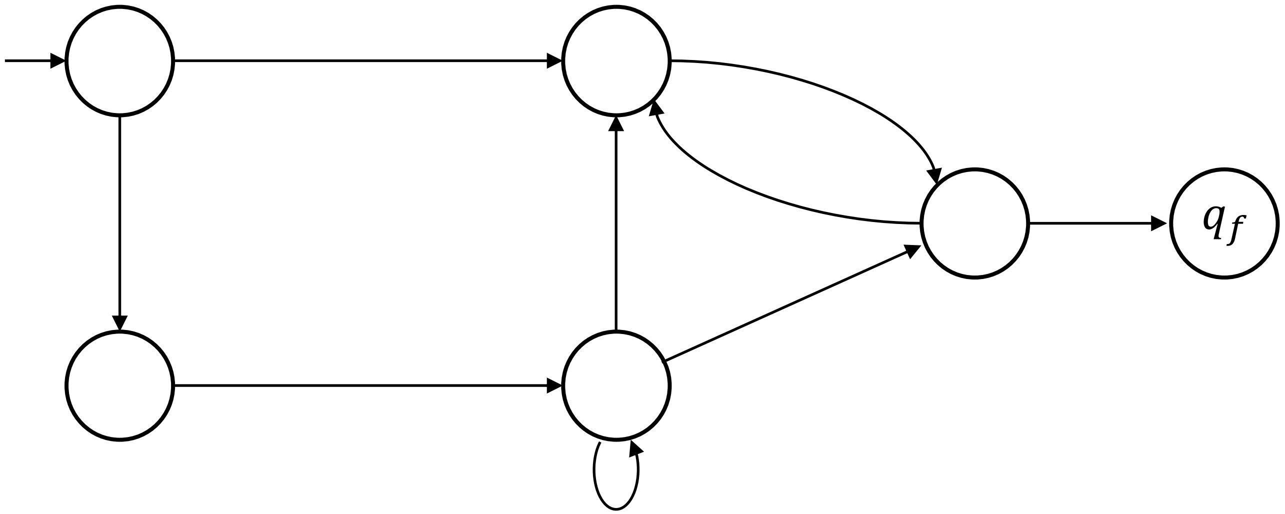


Shared memory

a	b	a
1	**2**	**3**

A basic problem: coverability

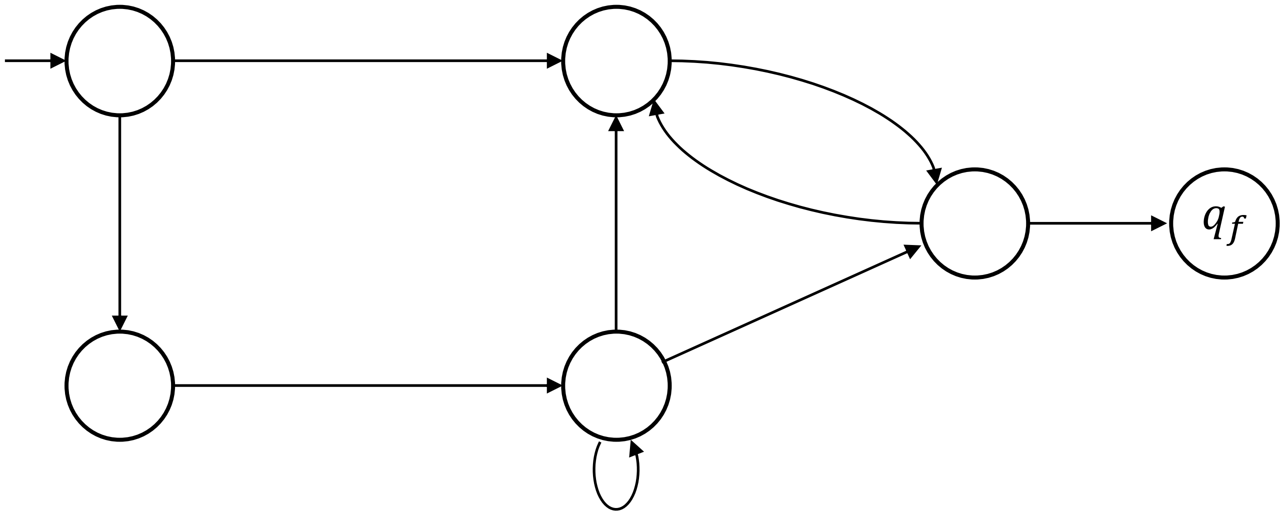
Coverability problem: *Input*: A protocol P (= an automaton) with an error state q_f.
Question: Does there exists a number of processes n and an execution of the system with n processes where one of them gets to q_f?



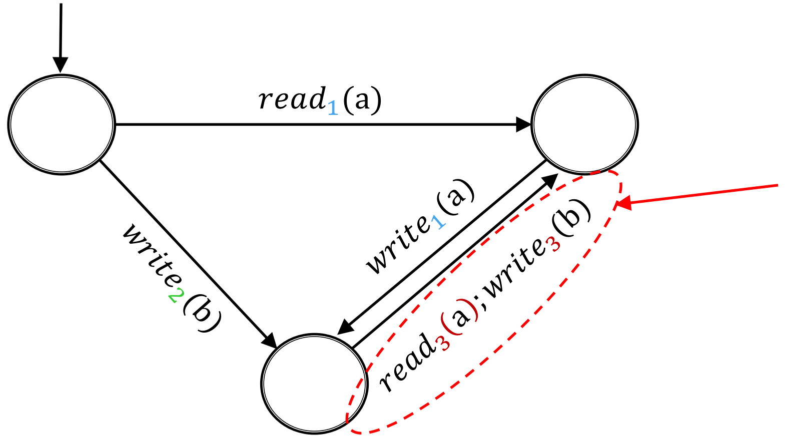
A basic problem: coverability

Coverability problem: *Input*: A protocol P (= an automaton) with an error state q_f.
Question: Does there exists a number of processes n and an execution of the system with n processes where one of them gets to q_f?

Parameterized problem: if answer is no then the system is safe for every value of n

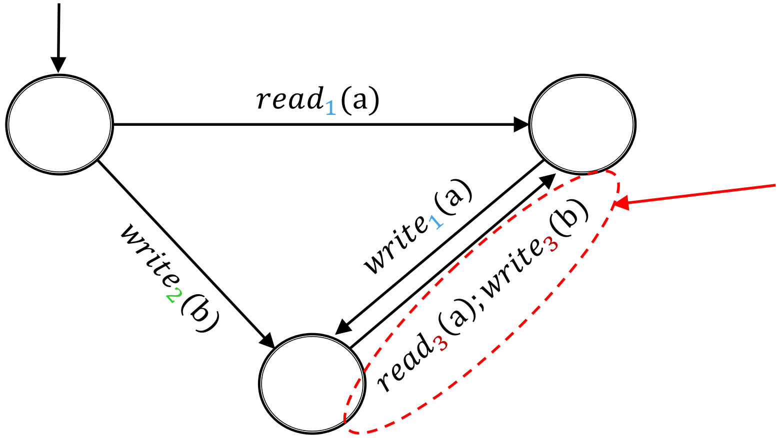


Atomic combinations are a bad idea

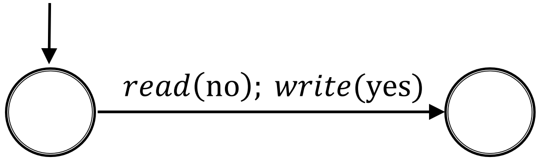


atomic read-write combination: a process can perform a read then a write and no one else can act in between

Atomic combinations are a bad idea

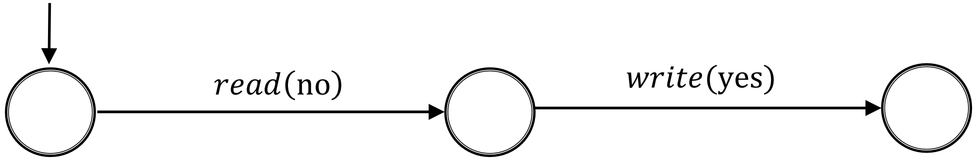


atomic read-write combination: a process can perform a read then a write and no one else can act in between



atomic

$\neq$



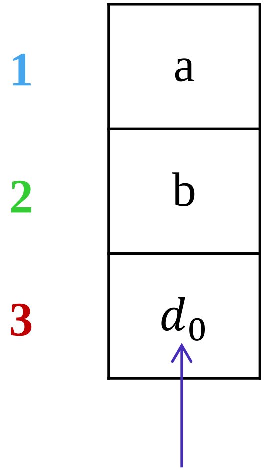
non-atomic

Atomic combinations allow for *leader election*: (too) powerful model

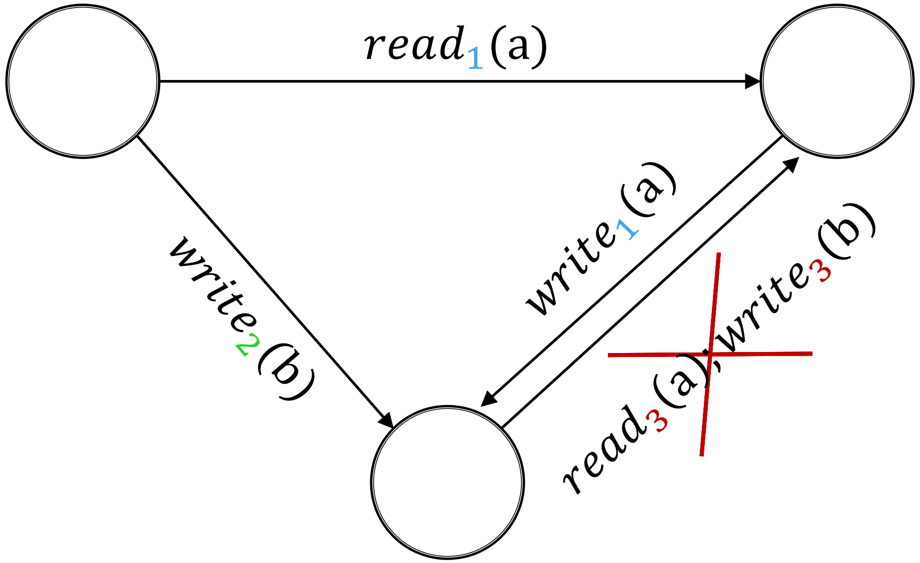
In fact, as expressive as *Petri Nets*: coverability is EXPSPACE-complete… → let's forbid atomic combinations

The model we obtain

Finite number of shared registers, each register has a value from finite set of symbols Σ



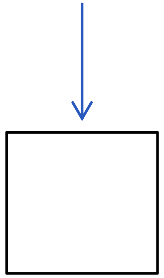
Initial value in the registers

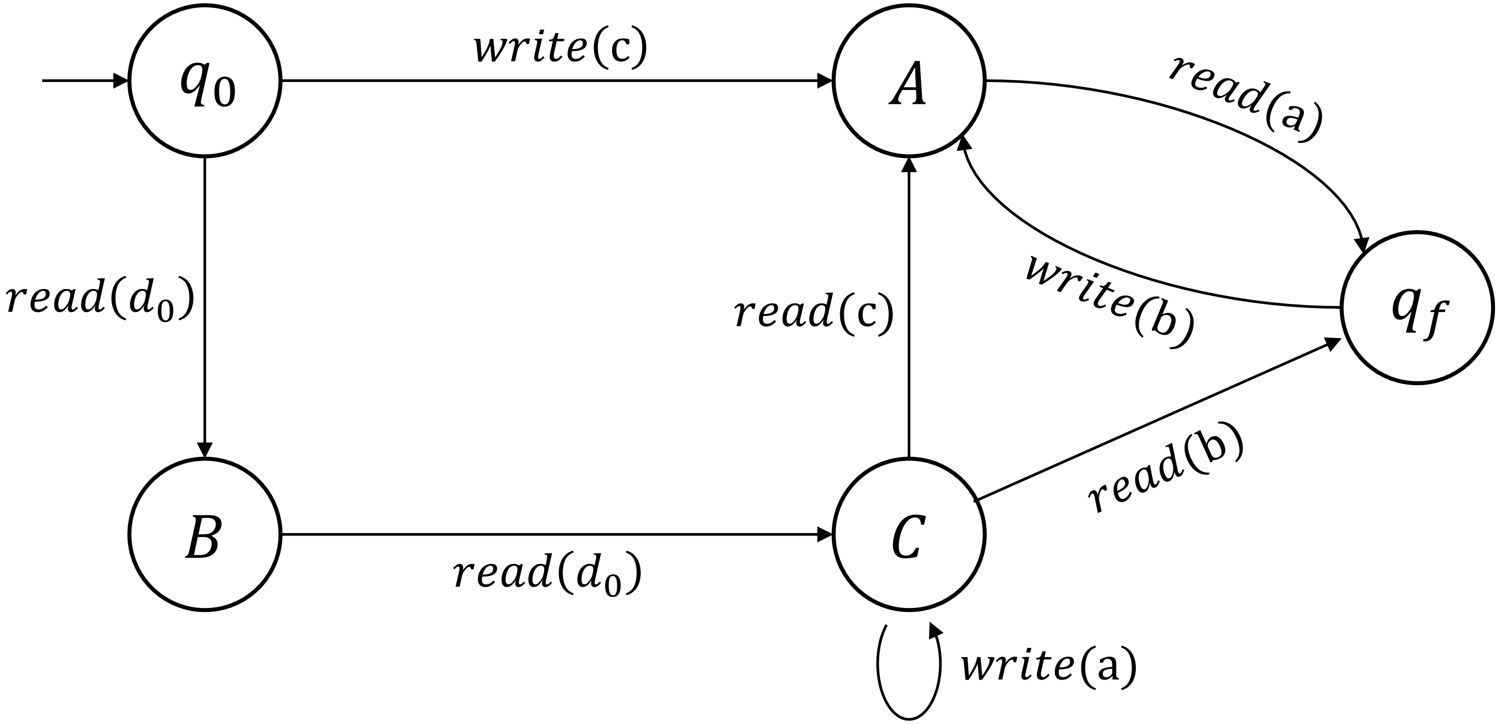


No atomic read/write combinations

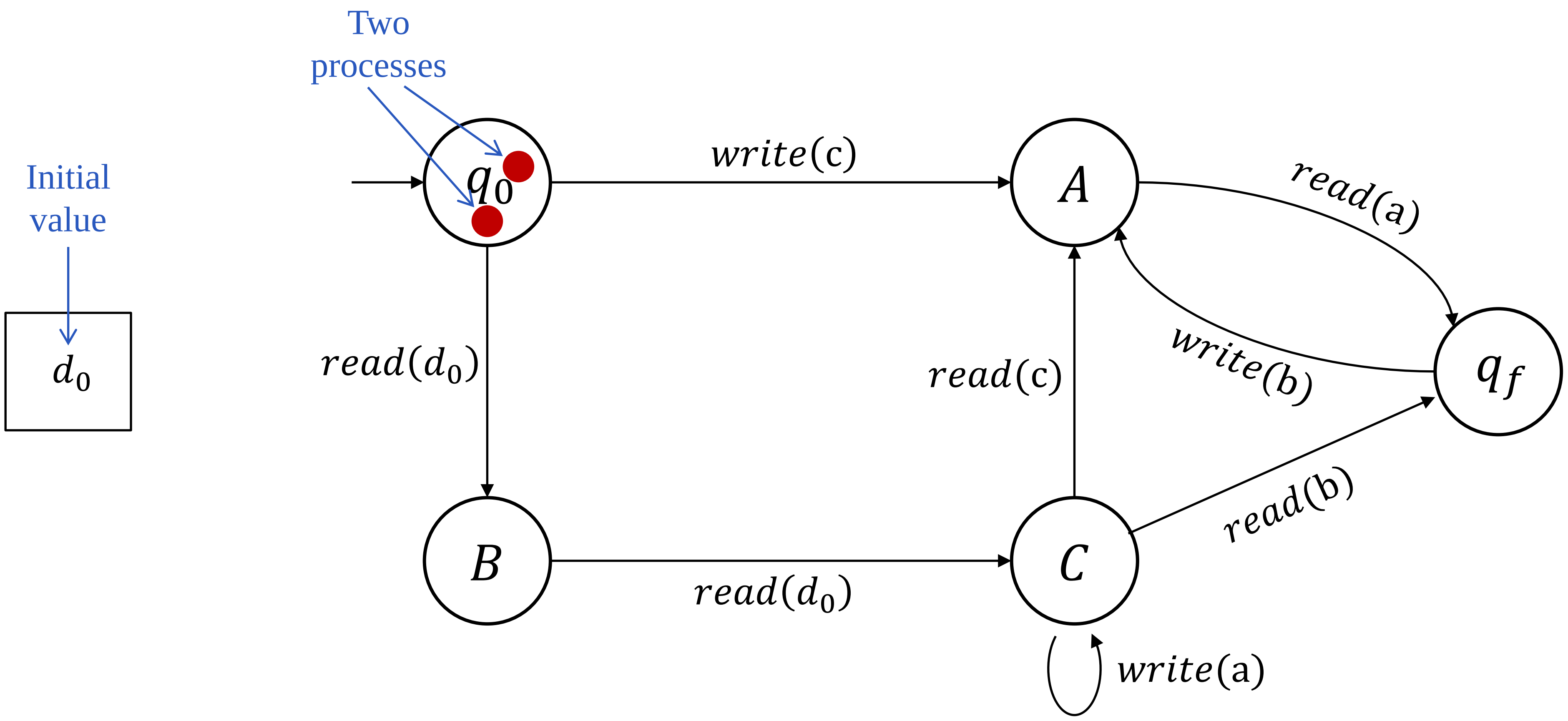
Model inspired by: Esparza, J., Ganty, P., Majumdar, R.: *Parameterized verification of asynchronous shared-memory systems*. Journal of the ACM, 2016

A small example

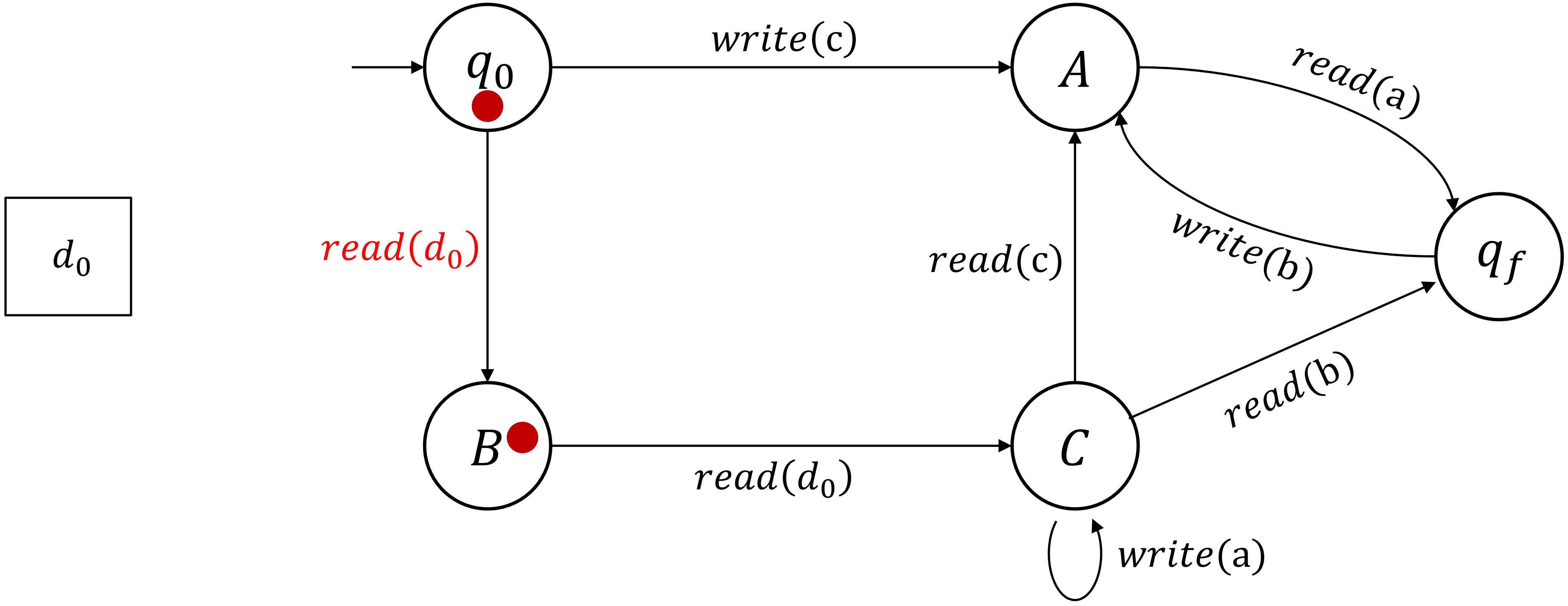




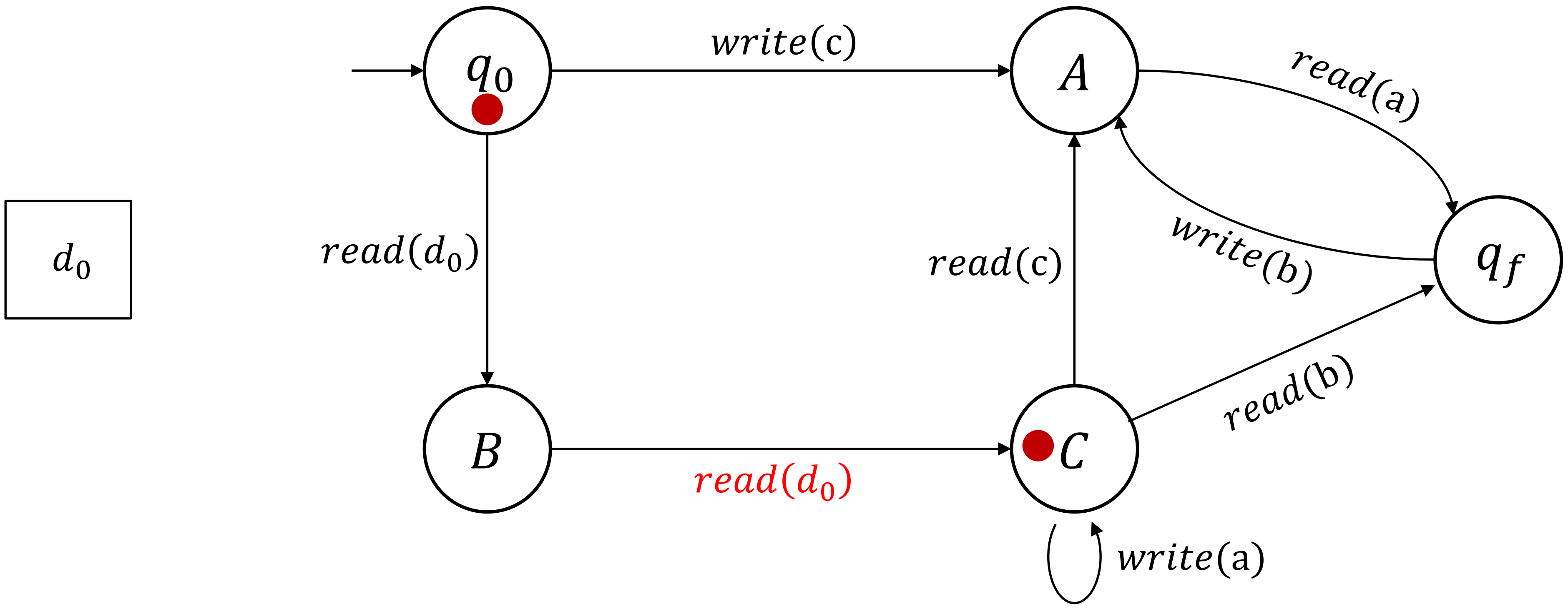
A small example



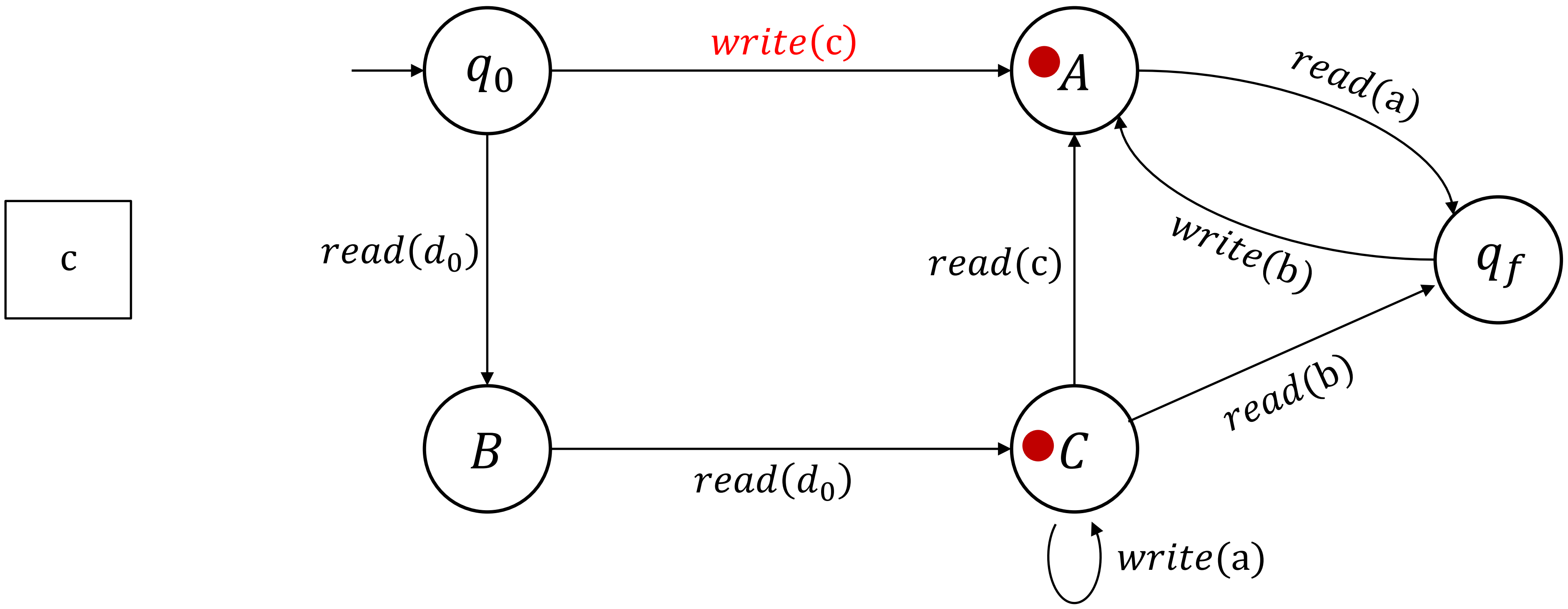
A small example



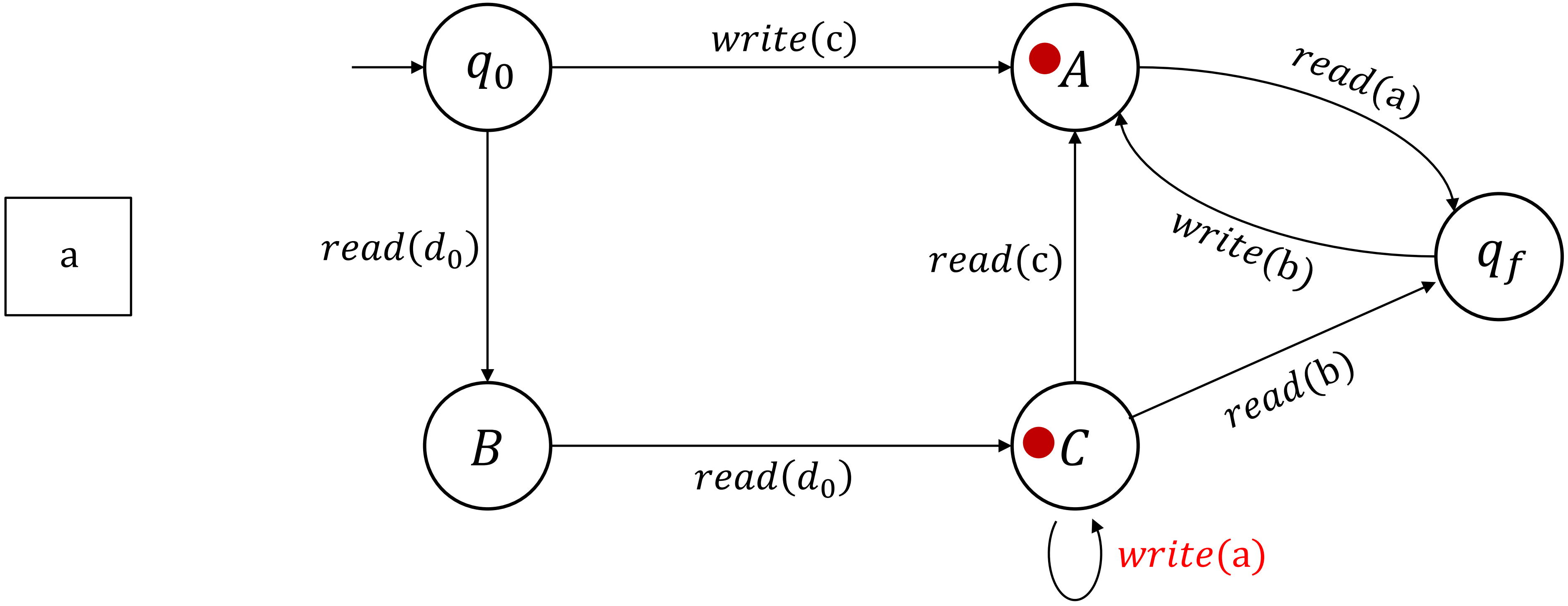
A small example



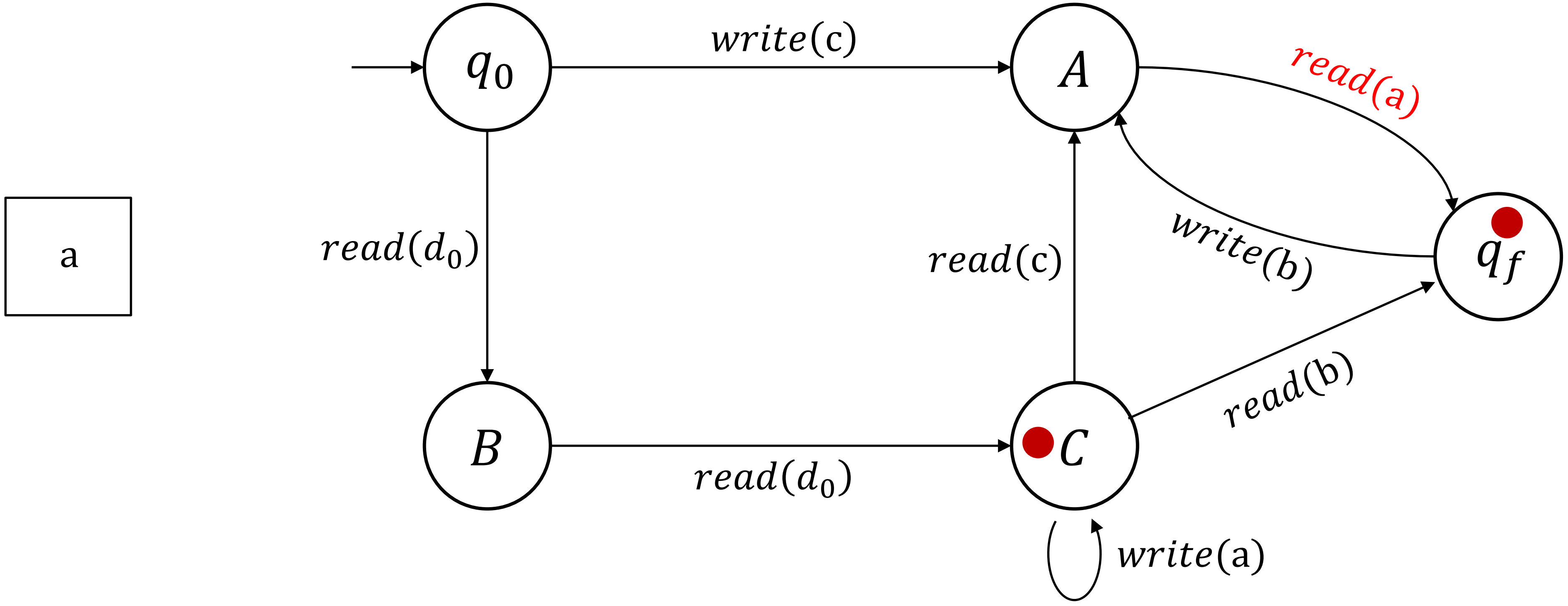
A small example



A small example



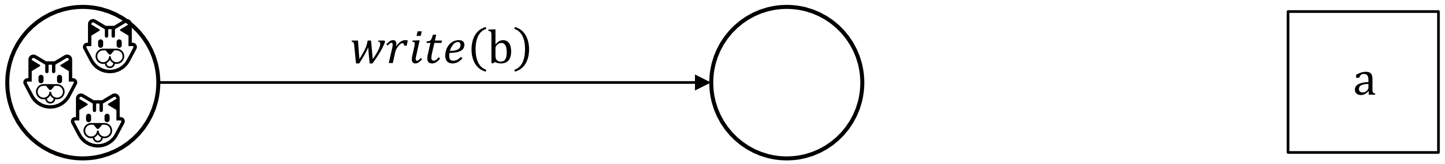
A small example



q_f is covered ✓

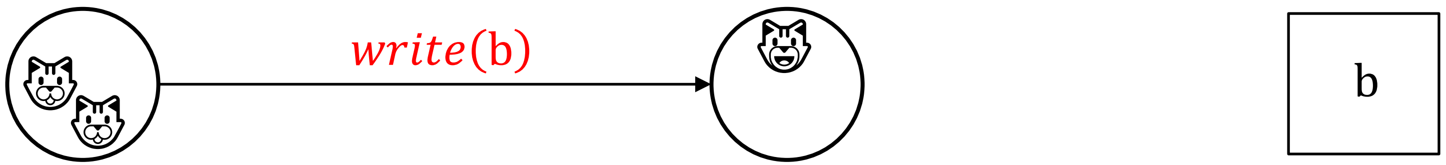
Cloning processes

A process may "copy" the behavior of another process on the same state.



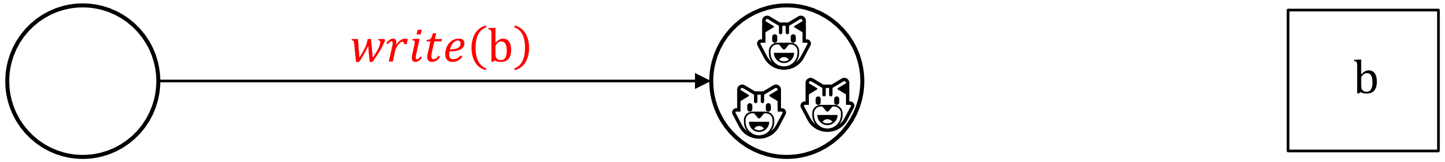
Cloning processes

A process may "copy" the behavior of another process on the same state.



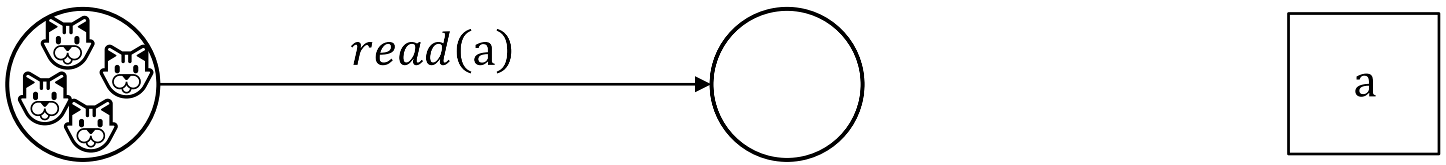
Cloning processes

A process may “copy” the behavior of another process on the same state.



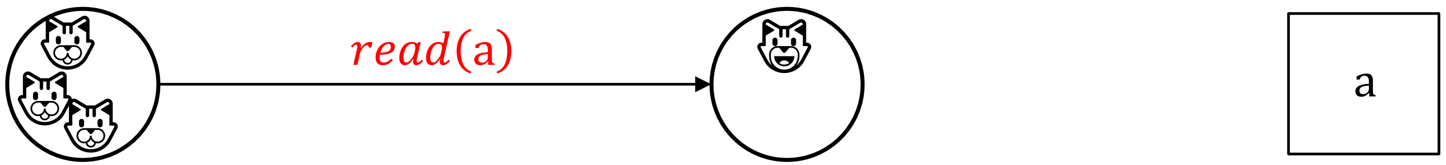
Cloning processes

A process may "copy" the behavior of another process on the same state.



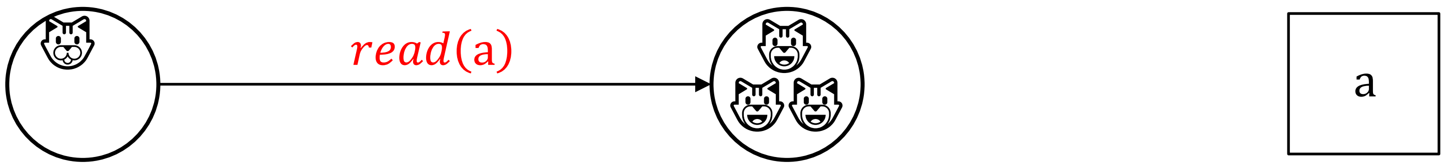
Cloning processes

A process may "copy" the behavior of another process on the same state.



Cloning processes

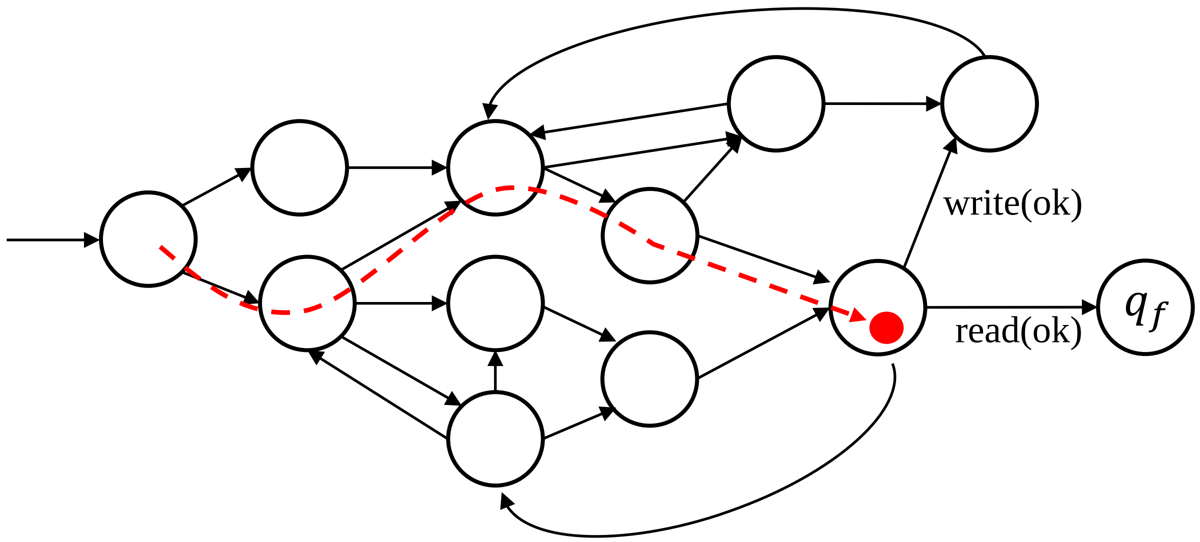
A process may "copy" the behavior of another process on the same state.



Cloning processes

A process may "copy" the behavior of another process on the same state.

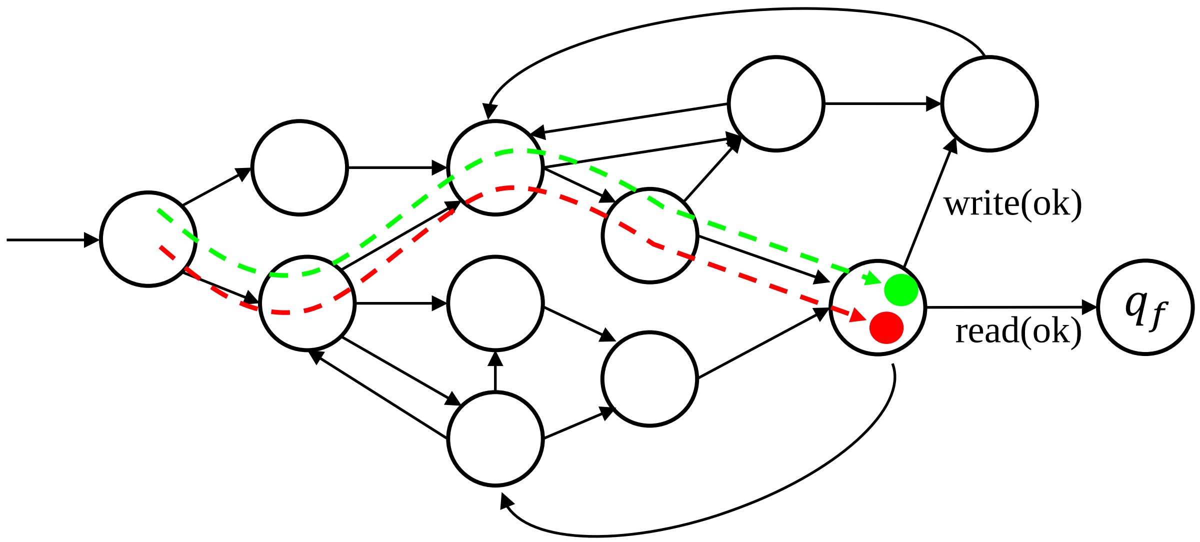
Copycat property: Where we can have one process, we can have many processes.



Cloning processes

A process may “copy” the behavior of another process on the same state.

Copycat property: Where we can have one process, we can have many processes.

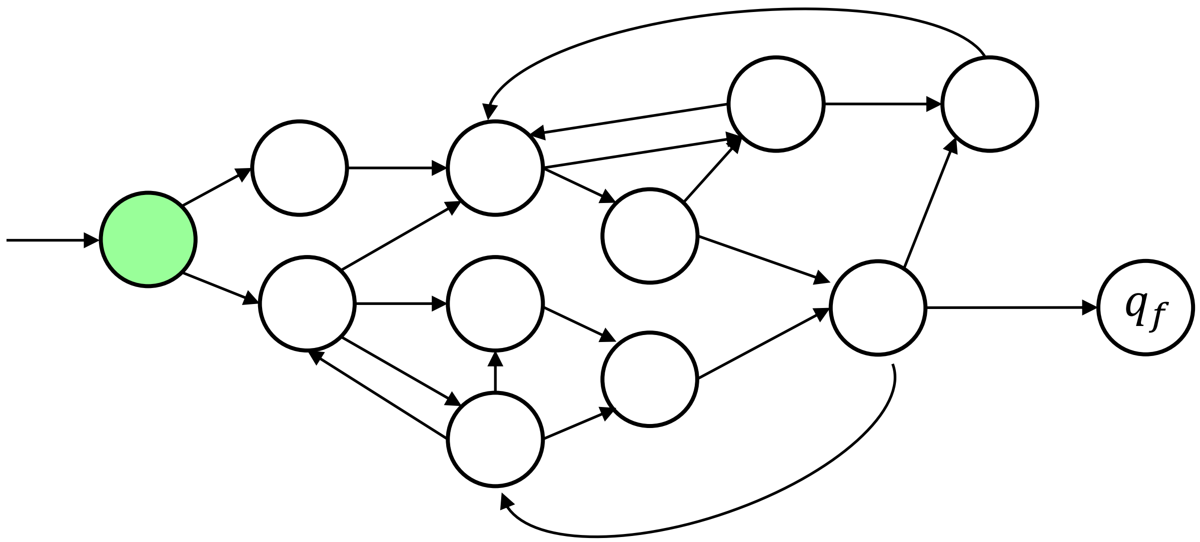


Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.

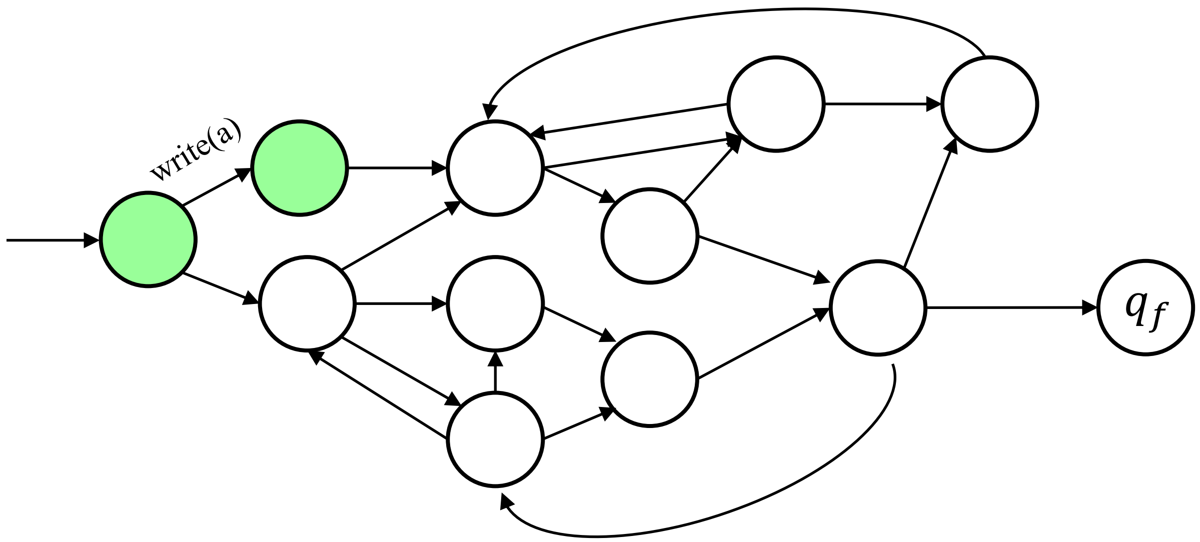
Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.



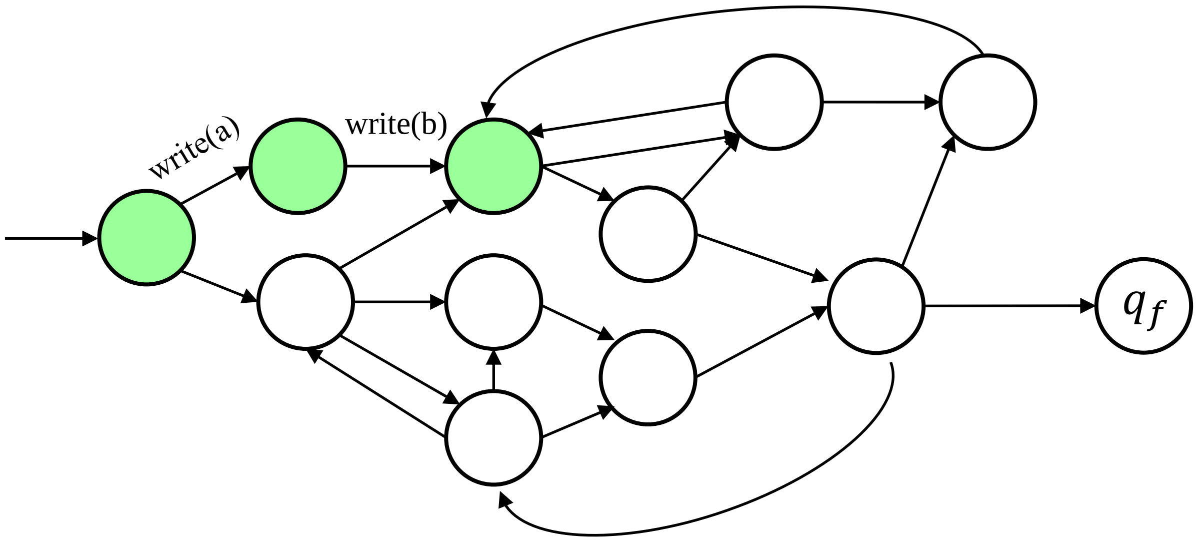
Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.



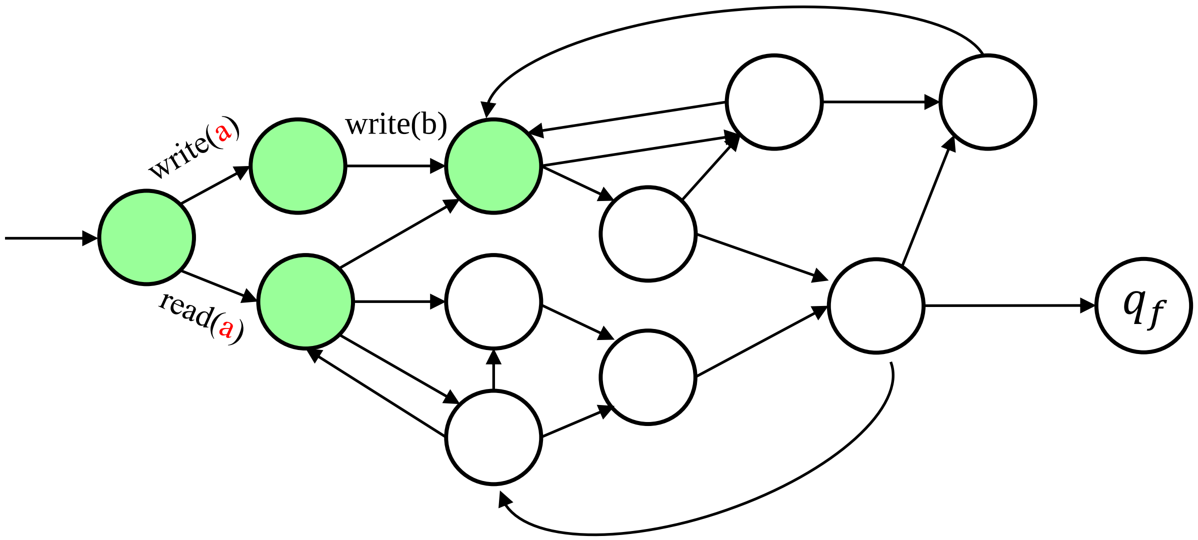
Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.



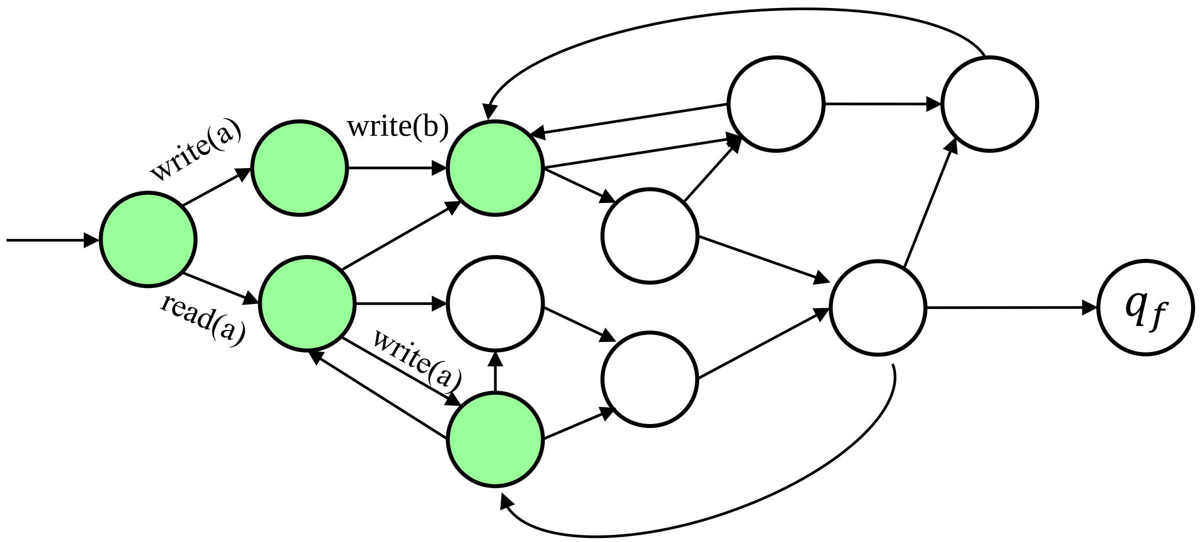
Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.



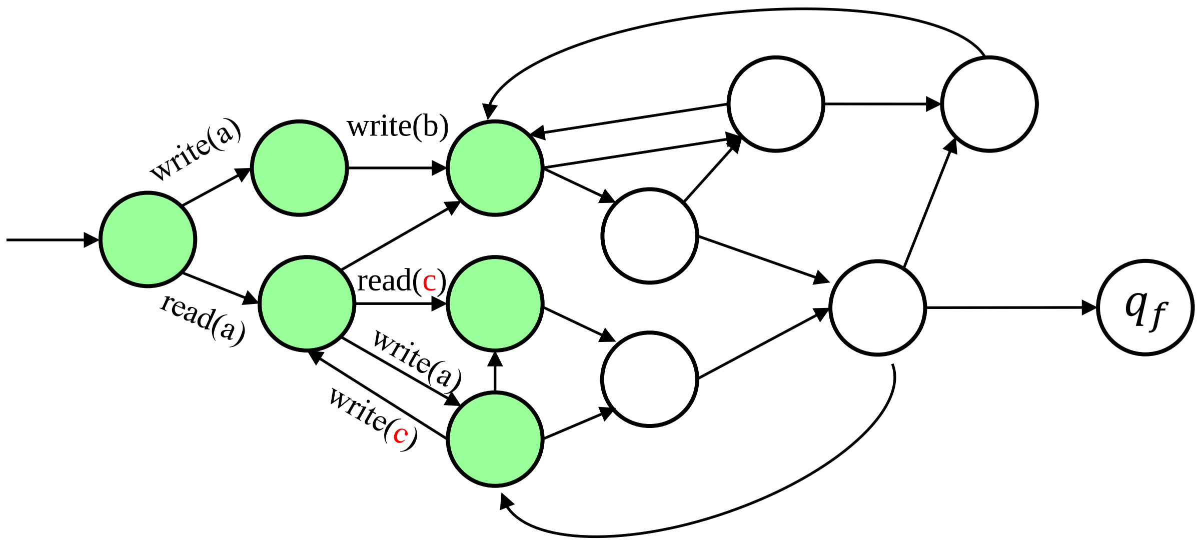
Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.



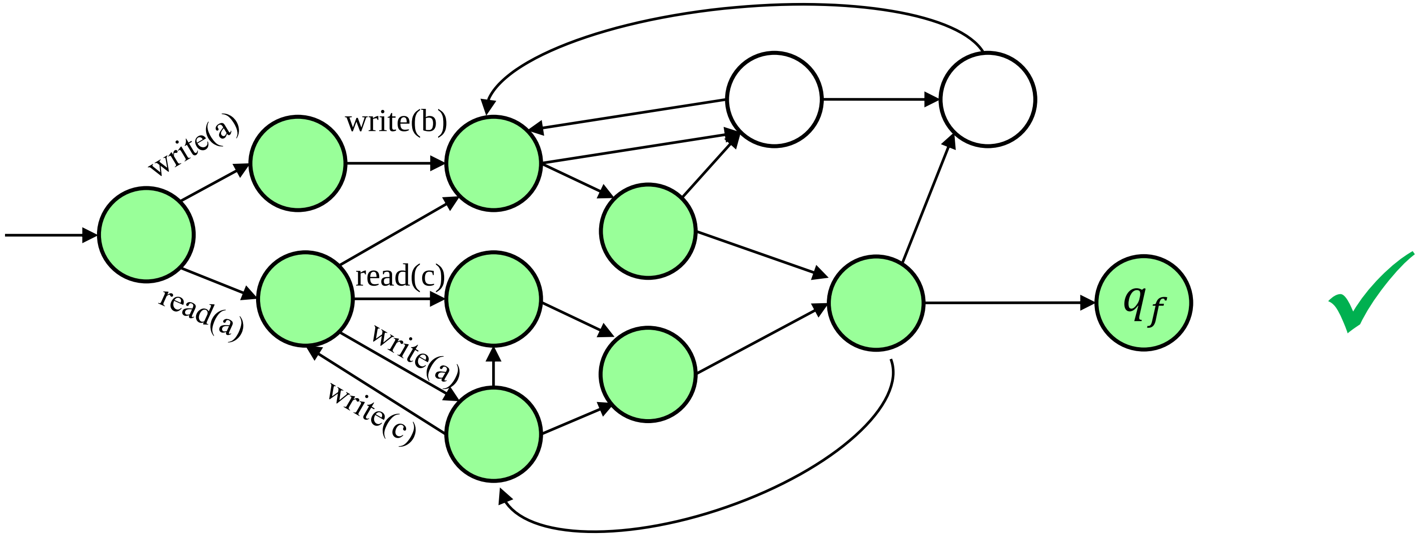
Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.



Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.



Complexity of COVER

COVER is decidable in polynomial time if d_0 cannot be read (= no initialization) using a simple saturation algorithm that computes all coverable states.

COVER is in **PTIME** in this case, by contrast it is:

- **NP**-complete if $read(d_0)$ transitions are allowed,
- **PSPACE**-complete if n is given as input.

Parameterized problem is much easier !

A too simple model?

In this model, many parameterized questions are between **PTIME** and **NP**.

A too simple model?

In this model, many parameterized questions are between **PTIME** and **NP**.

However, the model is limited; many shared-memory algorithms require more expressiveness !
One such example: *round-based algorithms*.

Our current model

Round-based algorithms

Round-based algorithms

We want to model round-based distributed algorithms[2,3,4] that look like this:

Asynchronous rounds

Each round has its own set of shared registers
→ unbounded memory !

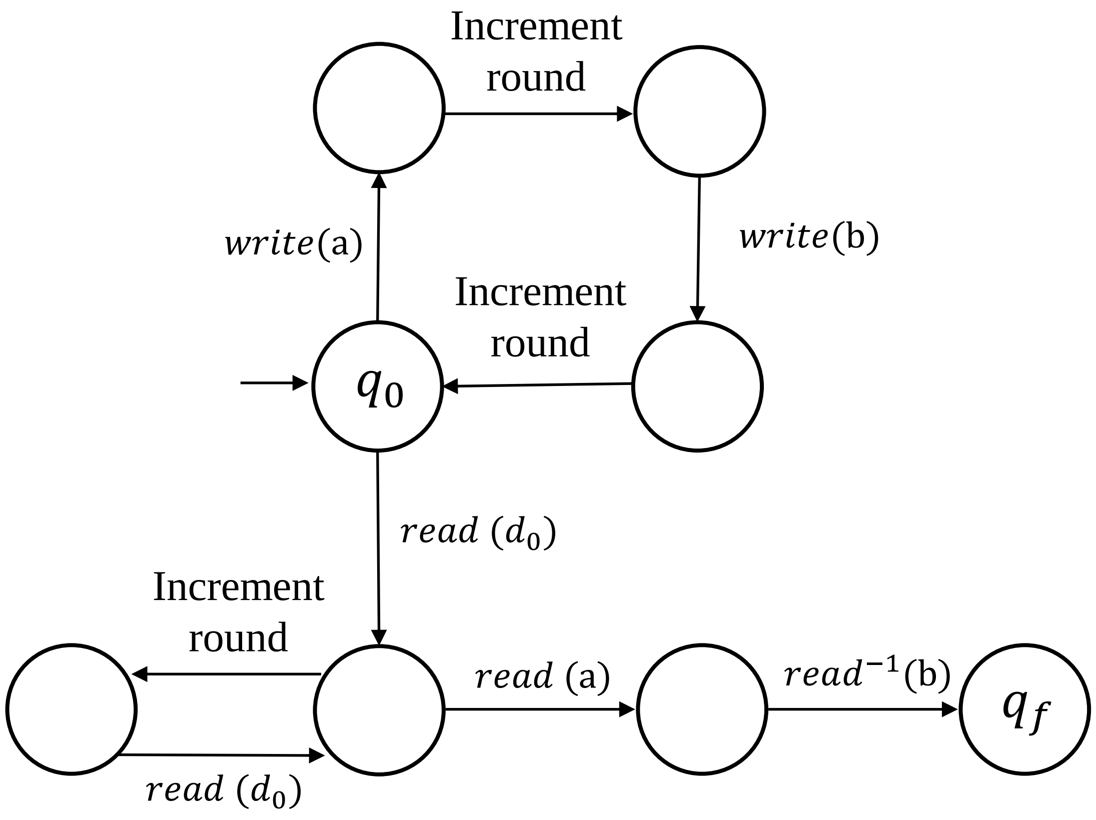
read and write to registers
of nearby rounds only

for $k = 0$ **to** ∞ **do**
- read register 0 of round k ;
- **if** *read value is a* **then** … **else** … ;
- write to register 1 of round k ;
- …
- read register 0 of round $k-1$;
- …

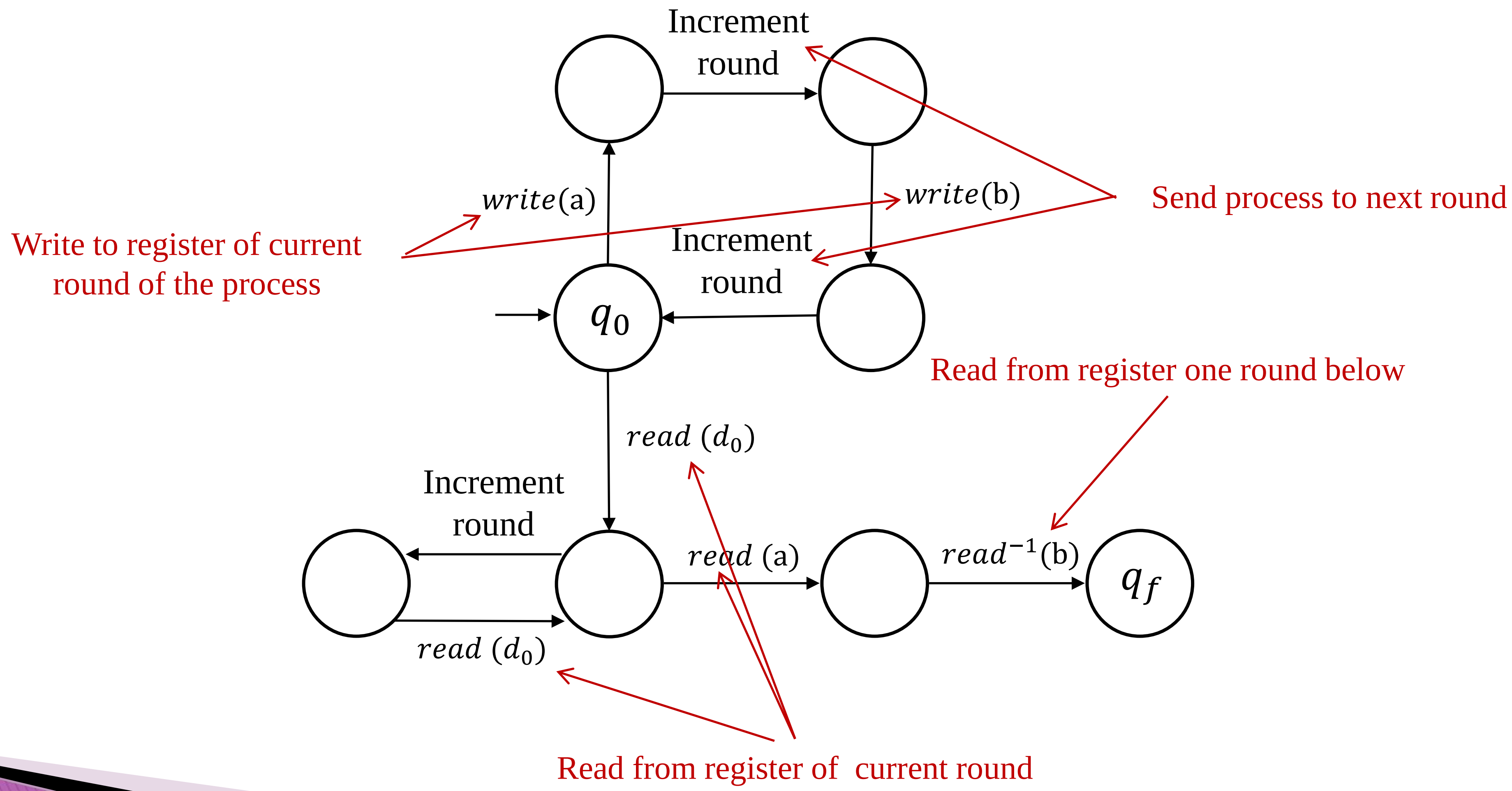
end

2. Aspnes, J.: *Fast deterministic consensus in a noisy environment.* Journal of Algorithms, 2002
3. Guerraoui, R., Ruppert, E.: *Anonymous and fault-tolerant shared-memory computing.* Distrib. Comput., 2007
4. Raynal, M., Stainer, J.: *A Simple Asynchronous Shared Memory Consensus Algorithm Based on Omega and Closing Sets. CISIS, 2012*

An example

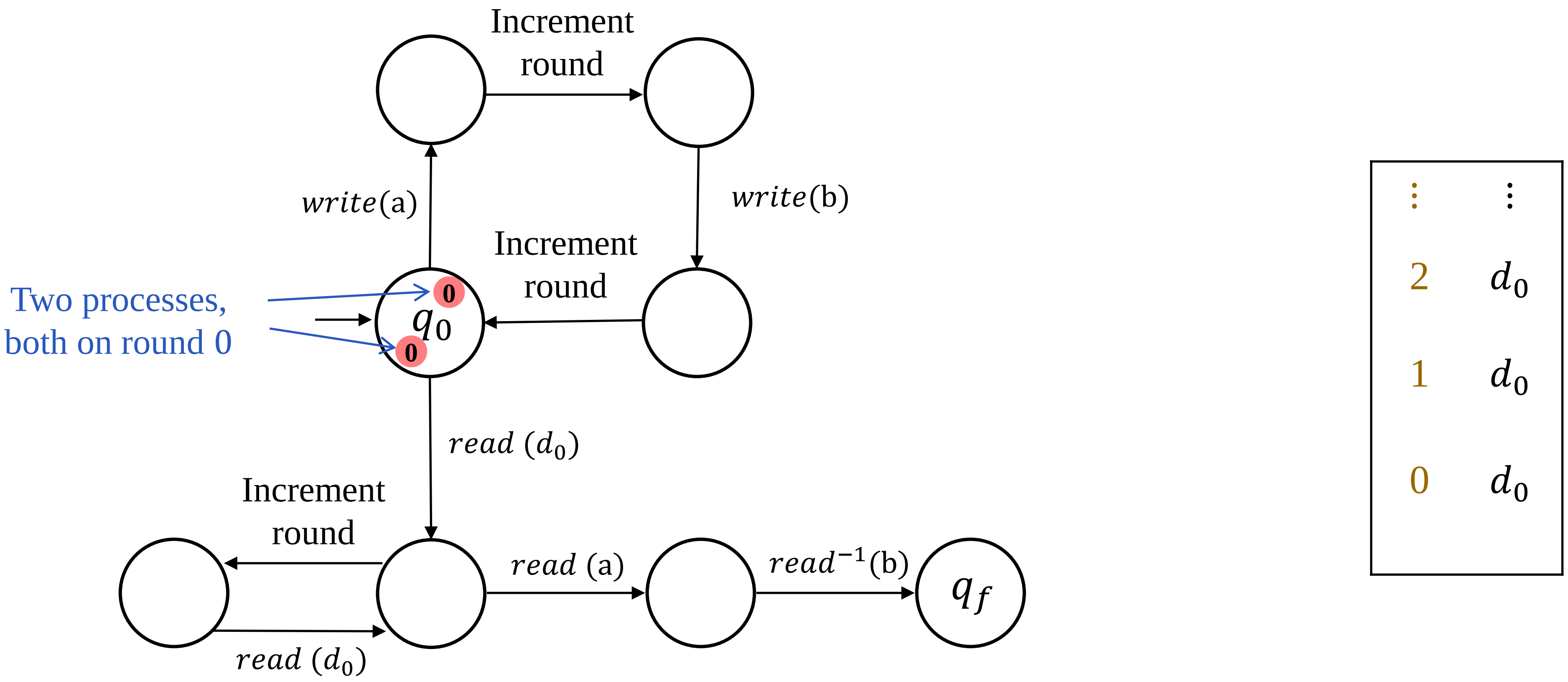

Increment round
$write(\text{a})$
$write(\text{b})$
Increment round
q_0
$read\ (d_0)$
Increment round
$read\ (\text{a})$
$read^{-1}(\text{b})$
q_f
$read\ (d_0)$

An example

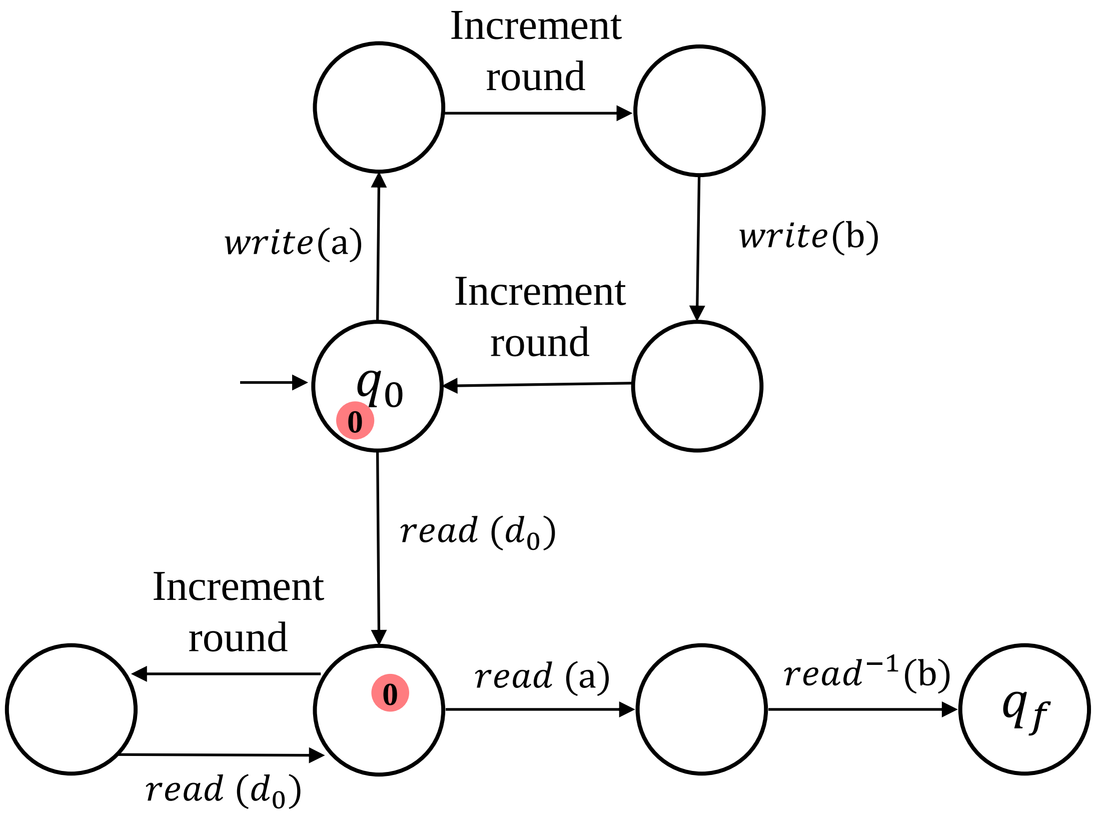


Increment
round
write(a)
write(b)
Send process to next round
Write to register of current
round of the process
Increment
round
q_0
Read from register one round below
read (d_0)
Increment
round
read (a)
$read^{-1}$(b)
q_f
read (d_0)
Read from register of current round

An example

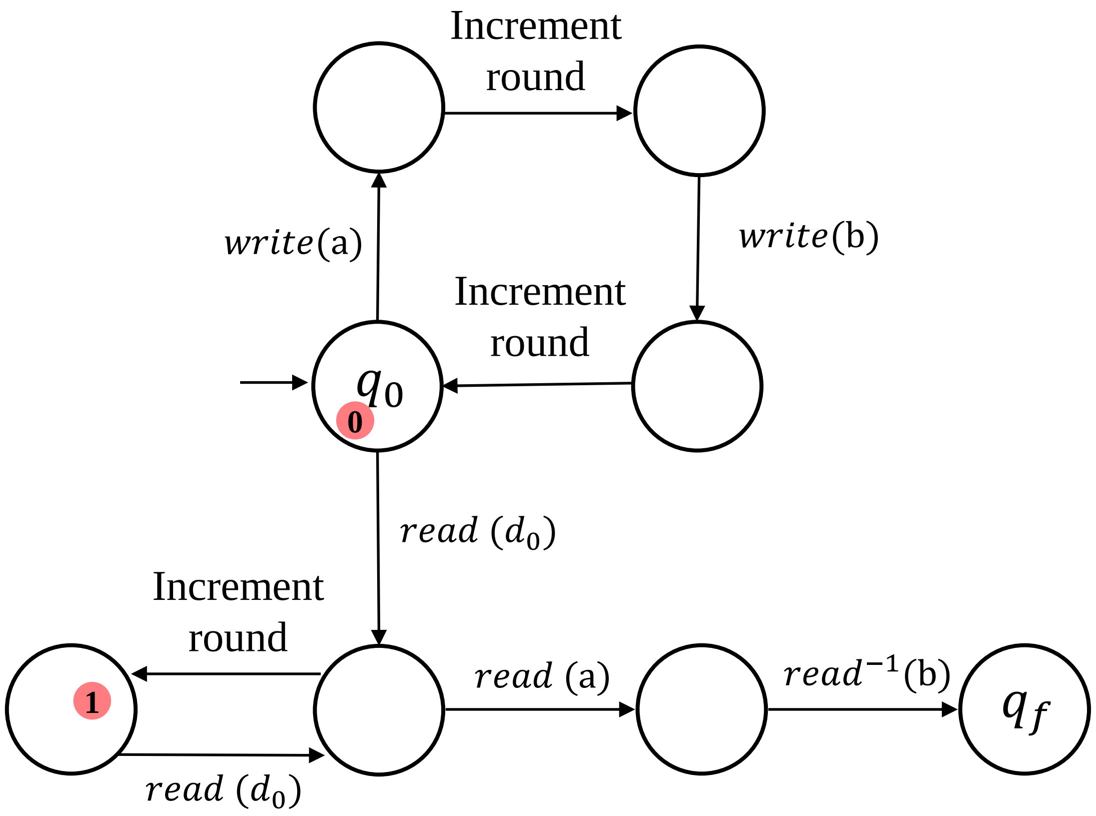

Increment
round
write(a)
write(b)
Increment
round
Two processes,
both on round 0
0
q_0
0
read (d_0)
Increment
round
read (a)
$read^{-1}$(b)
q_f
read (d_0)
⋮ ⋮
2 d_0
1 d_0
0 d_0

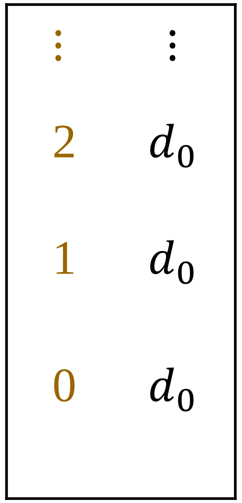
An example



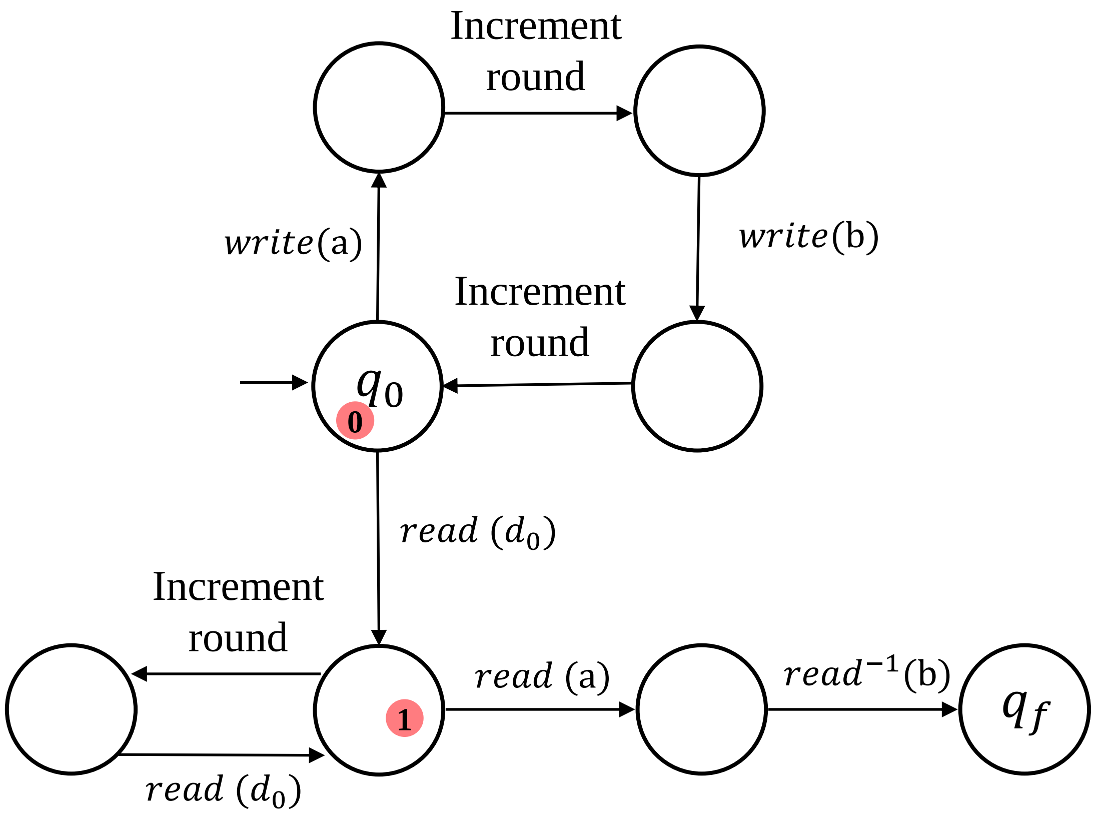
⋮	⋮
2	d_0
1	d_0
0	d_0

An example


Increment round
write(a)
write(b)
Increment round
q_0
0
read (d_0)
Increment round
1
read (a)
$read^{-1}$(b)
q_f
read (d_0)

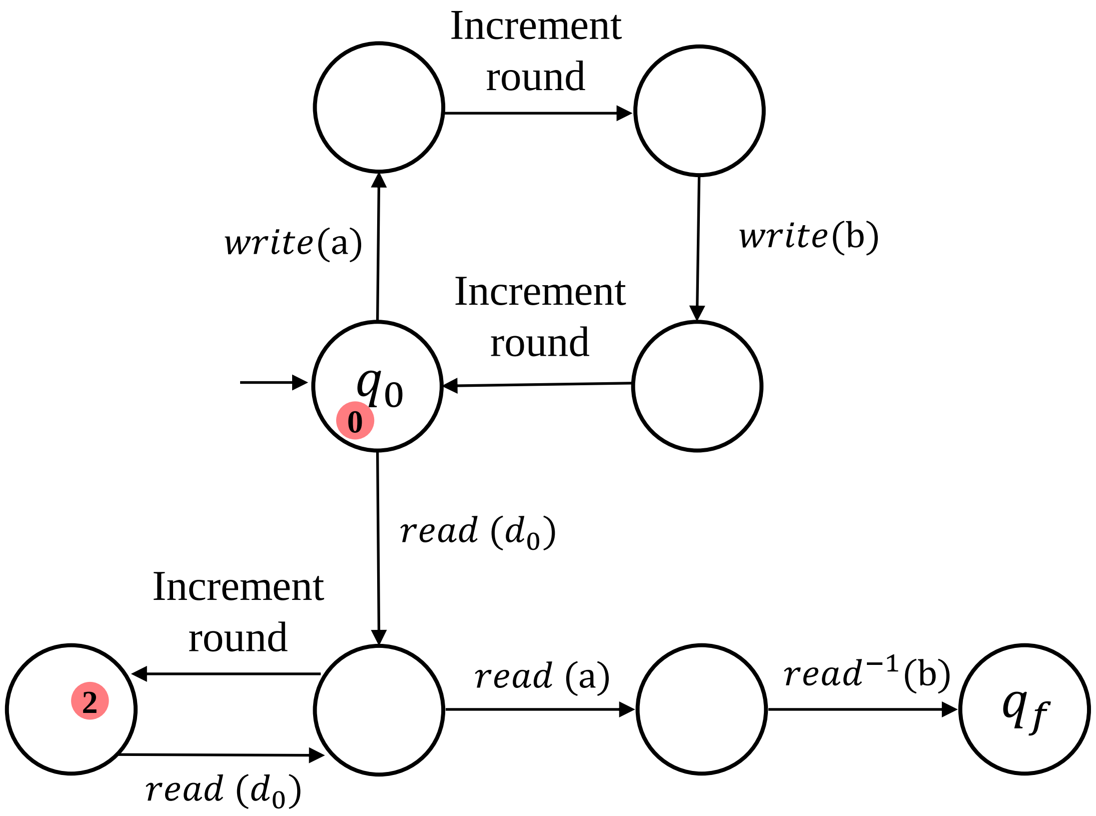

⋮ ⋮
2 d_0
1 d_0
0 d_0

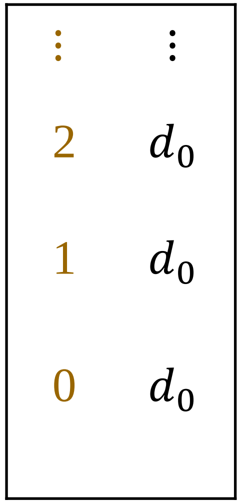
An example



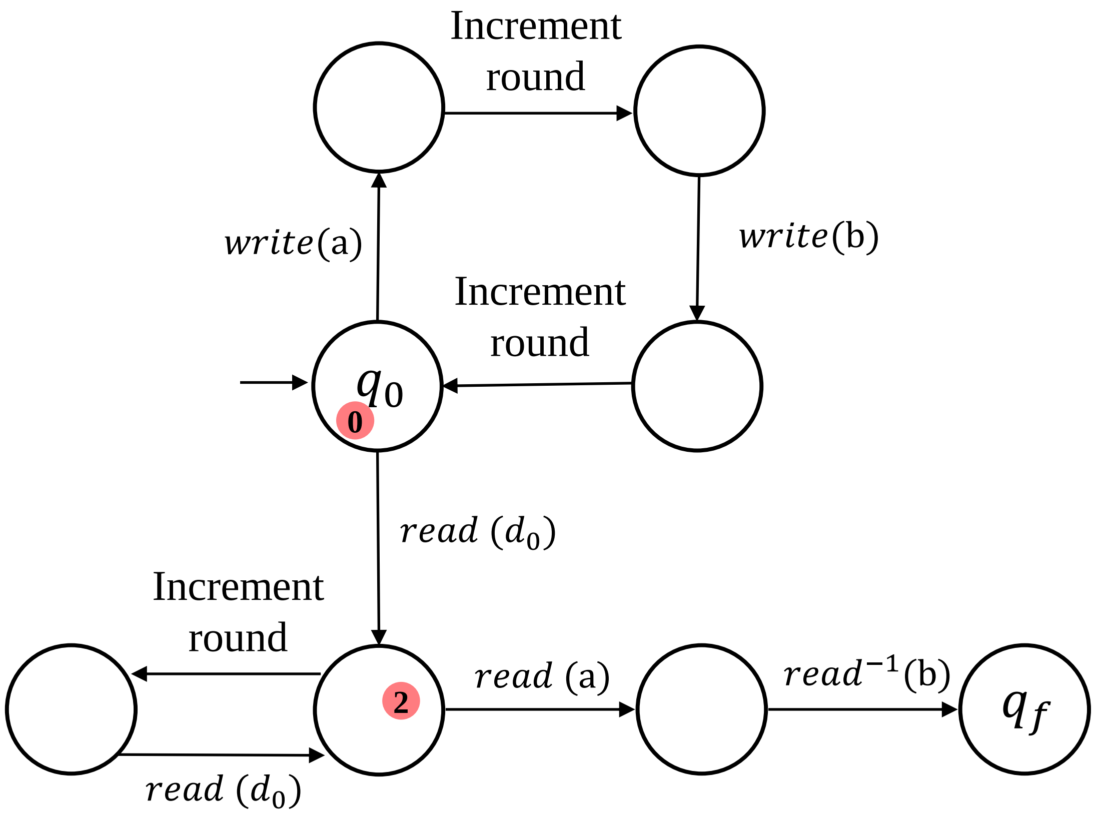
⋮	⋮
2	d_0
1	d_0
0	d_0

An example

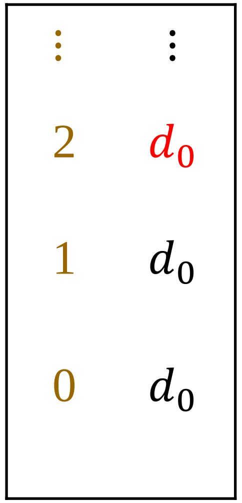

Increment round
write(a)
write(b)
Increment round
q_0
0
read (d_0)
Increment round
2
read (a)
read^{-1}(b)
q_f
read (d_0)


⋮ ⋮
2 d_0
1 d_0
0 d_0

An example

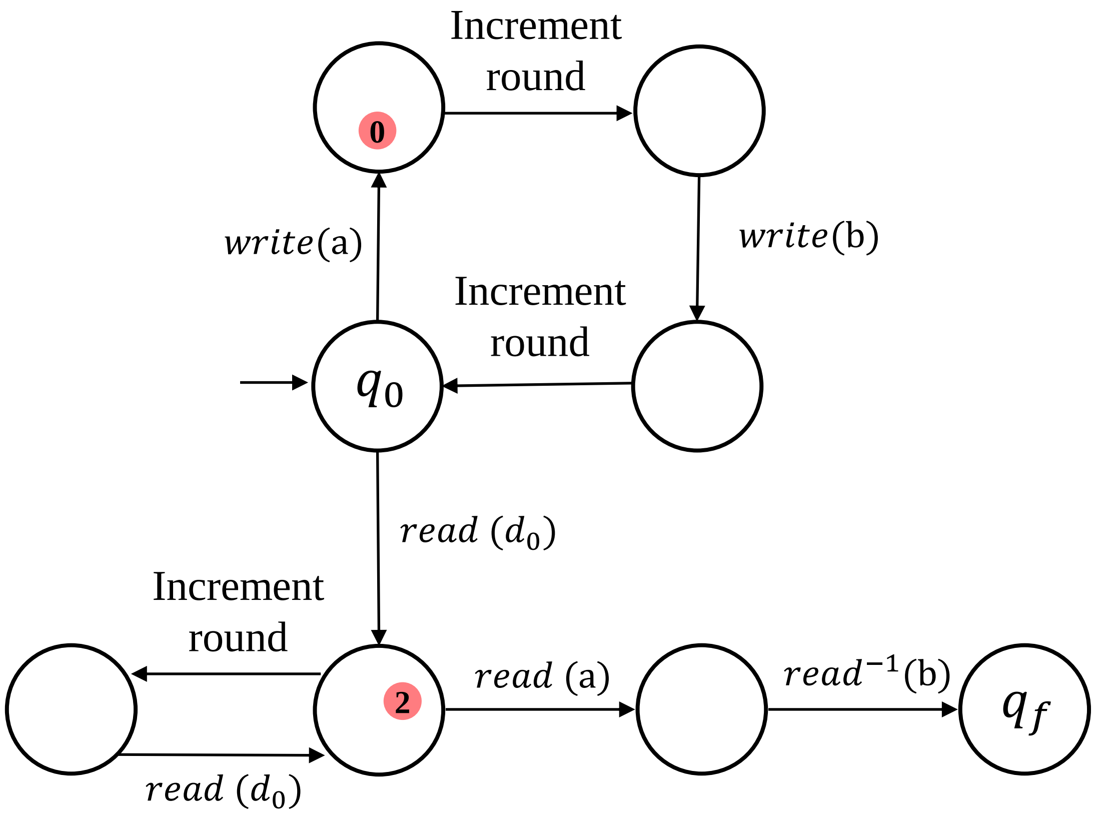


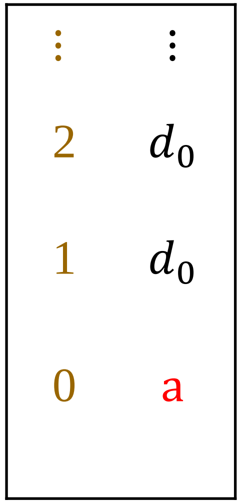
Increment
round
write(a)
write(b)
Increment
round
q_0
0
read (d_0)
Increment
round
2
read (a)
read^{-1}(b)
q_f
read (d_0)



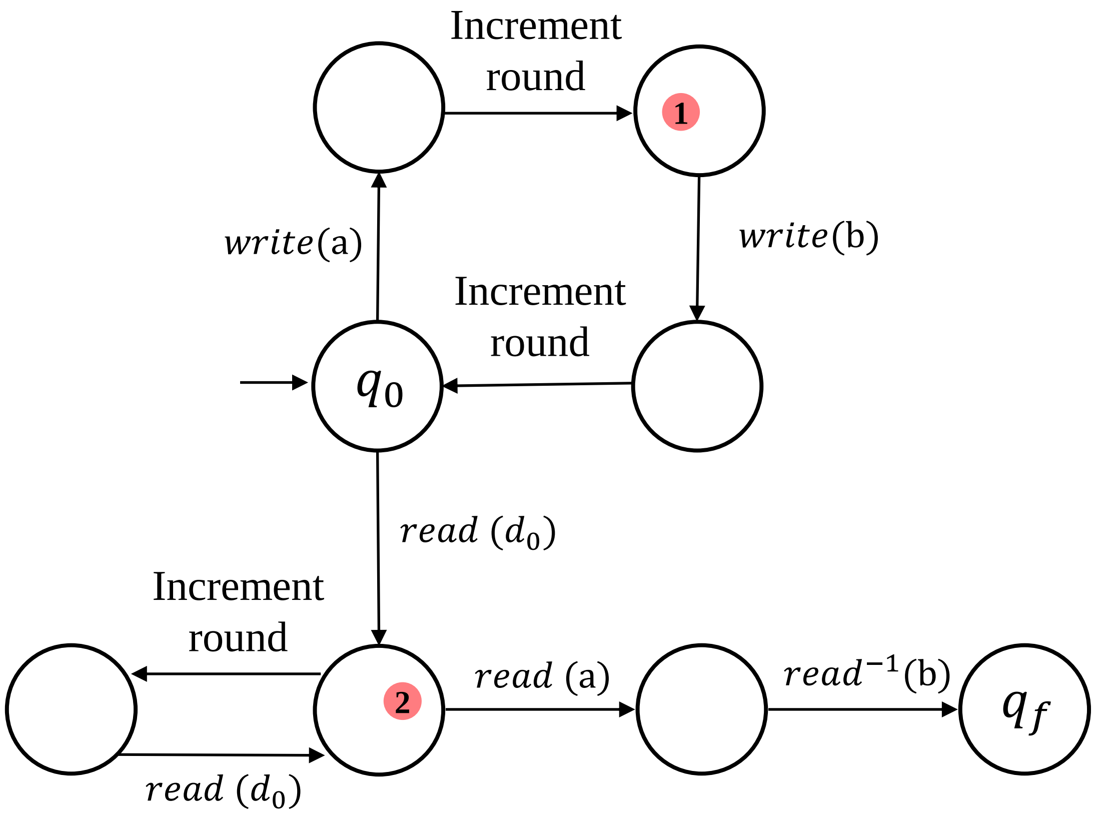
⋮ ⋮
2 d_0
1 d_0
0 d_0

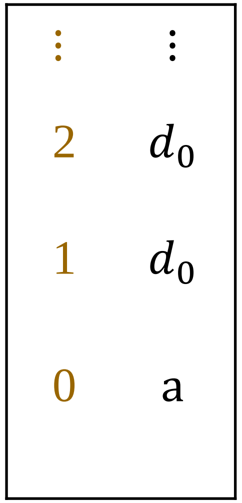
An example


Increment round
0
write(a)
write(b)
Increment round
q_0
read (d_0)
Increment round
2
read (a)
$read^{-1}$(b)
q_f
read (d_0)

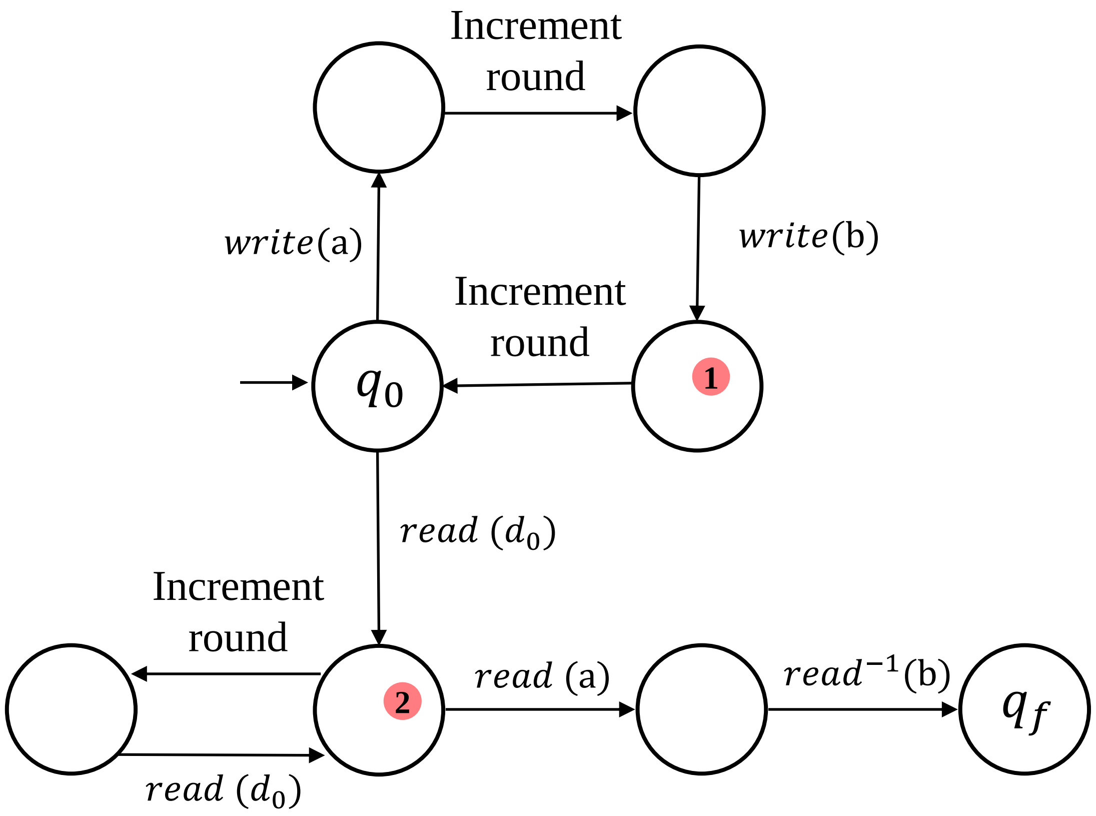

⋮ ⋮
2 d_0
1 d_0
0 a

An example


Increment
round
1
write(a)
write(b)
Increment
round
q_0
read (d_0)
Increment
round
2
read (a)
$read^{-1}$(b)
q_f
read (d_0)

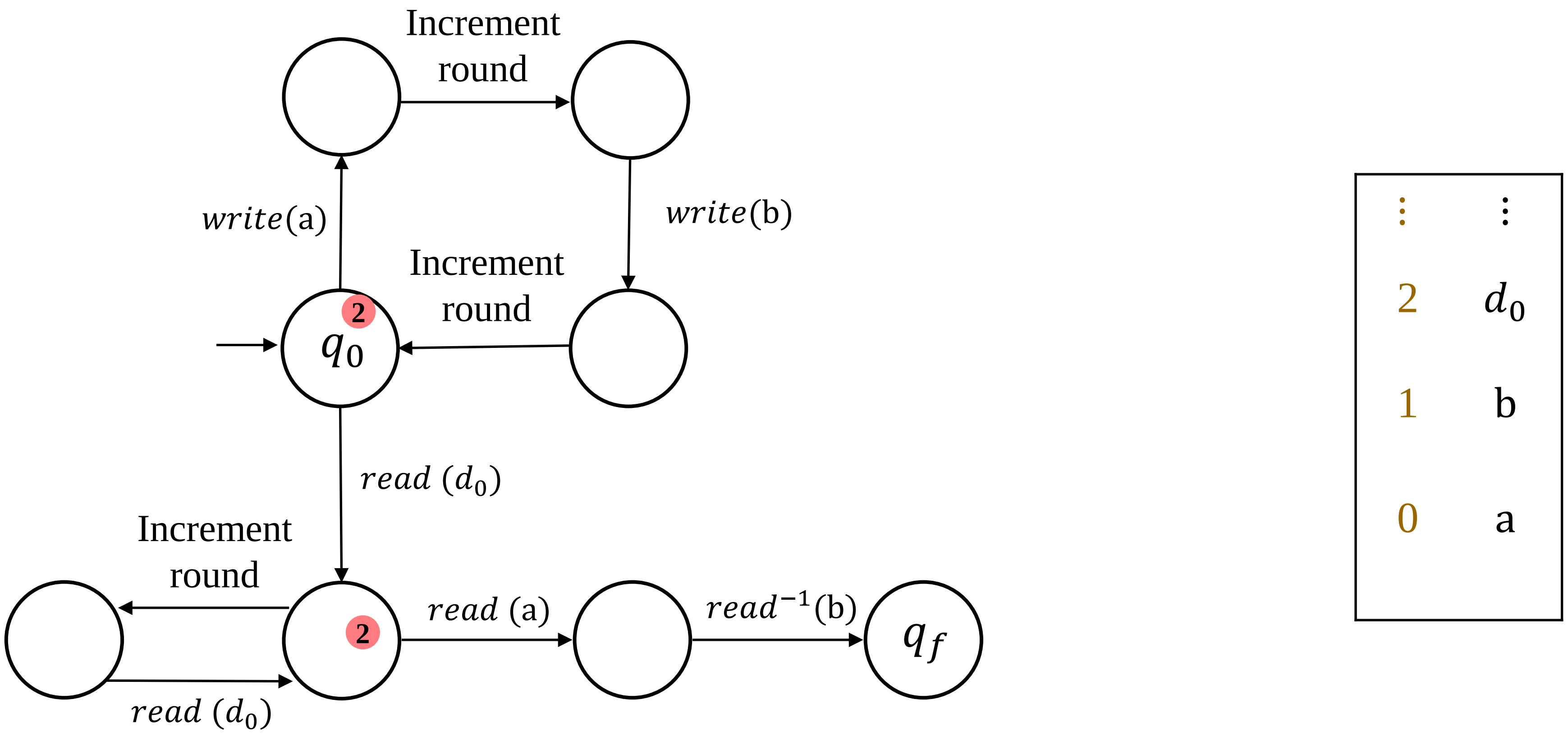

⋮ ⋮
2 d_0
1 d_0
0 a

An example



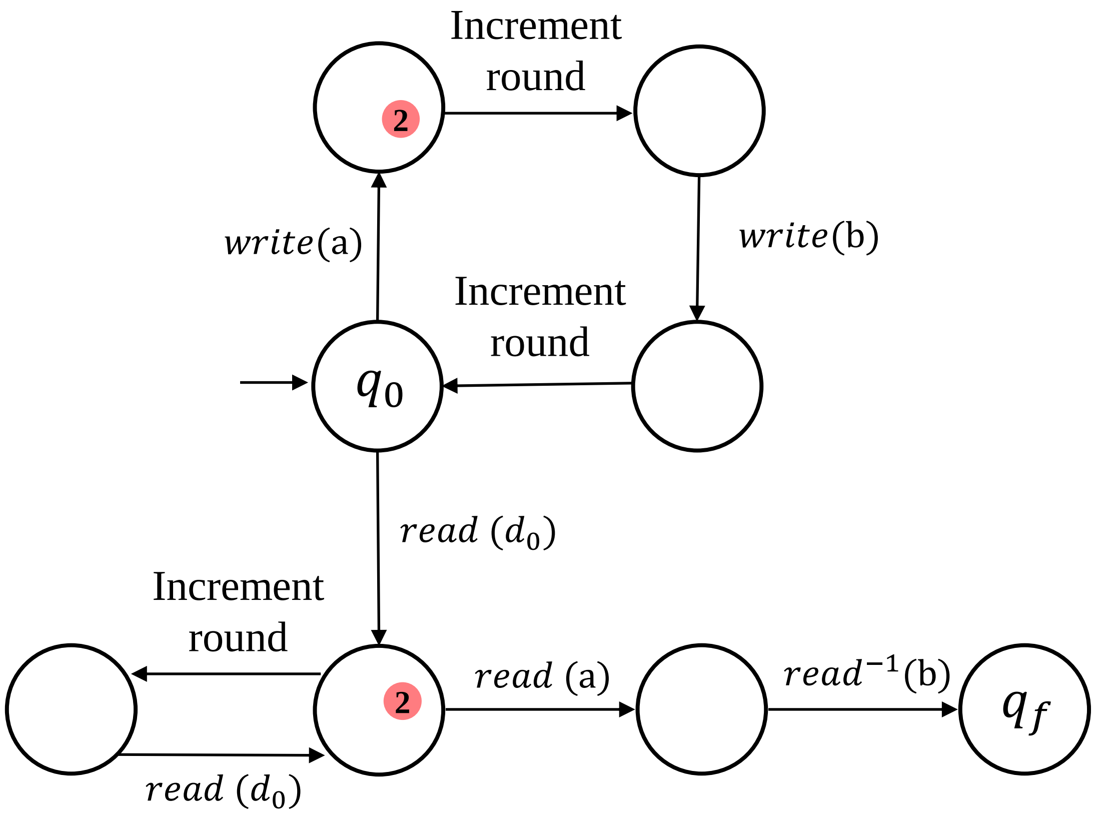
⋮	⋮
2	d_0
1	b
0	a

An example



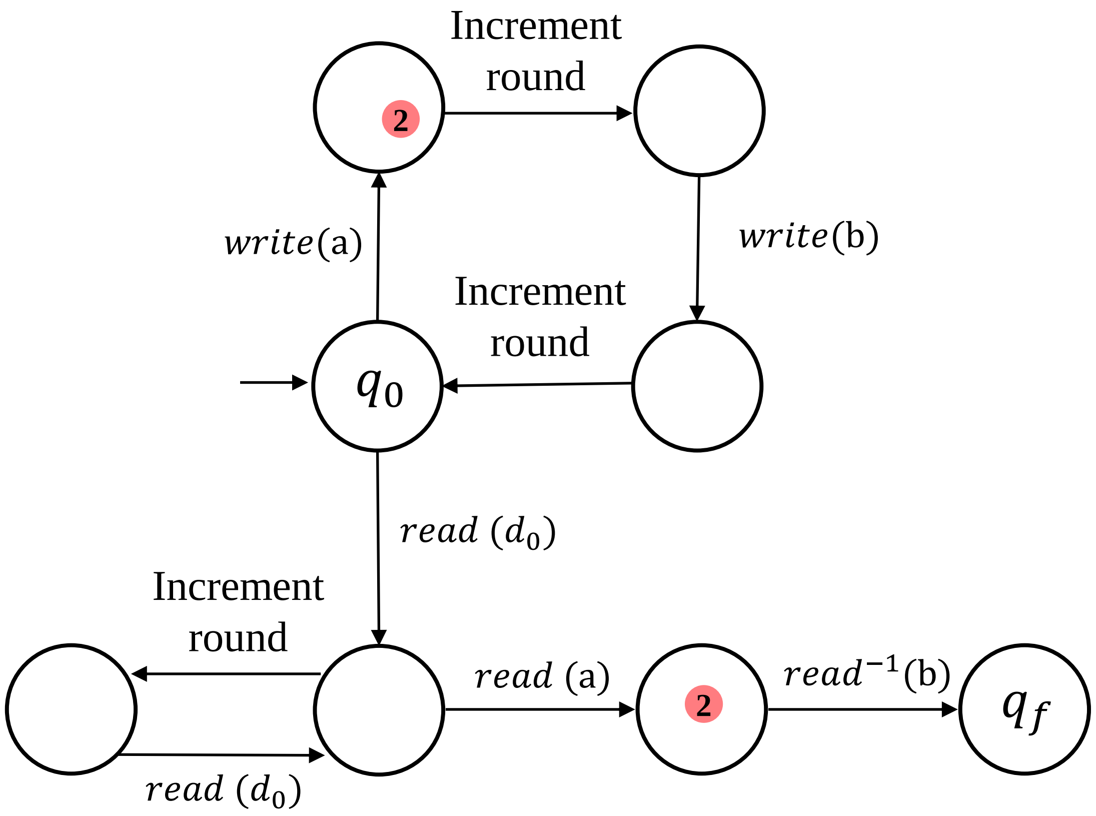
Increment round
write(a)
write(b)
Increment round
2
q_0
read (d_0)
Increment round
2
read (a)
$read^{-1}$(b)
q_f
read (d_0)
⋮ ⋮
2 d_0
1 b
0 a

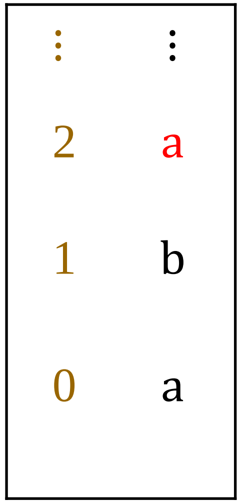
An example



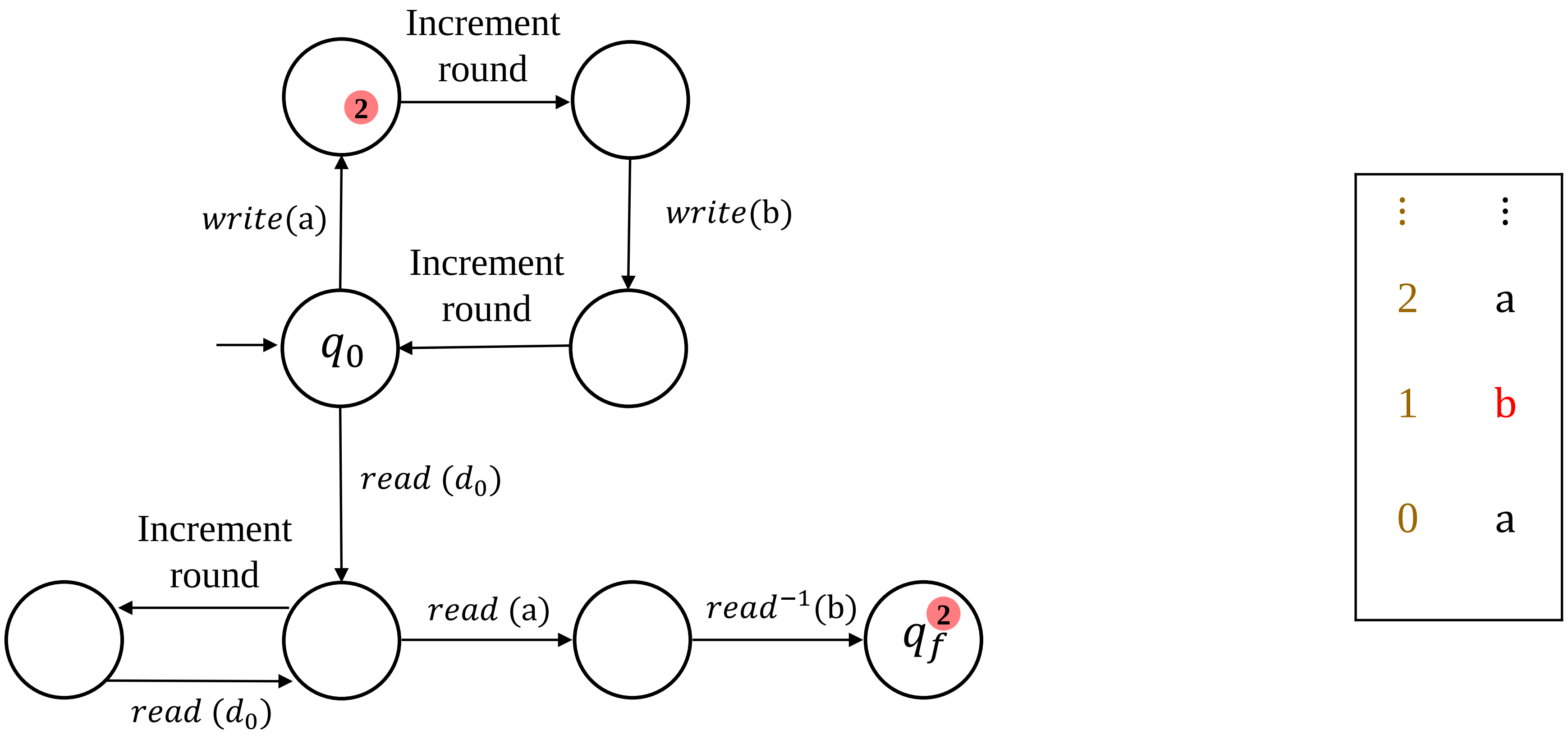
⋮	⋮
2	a
1	b
0	a

An example

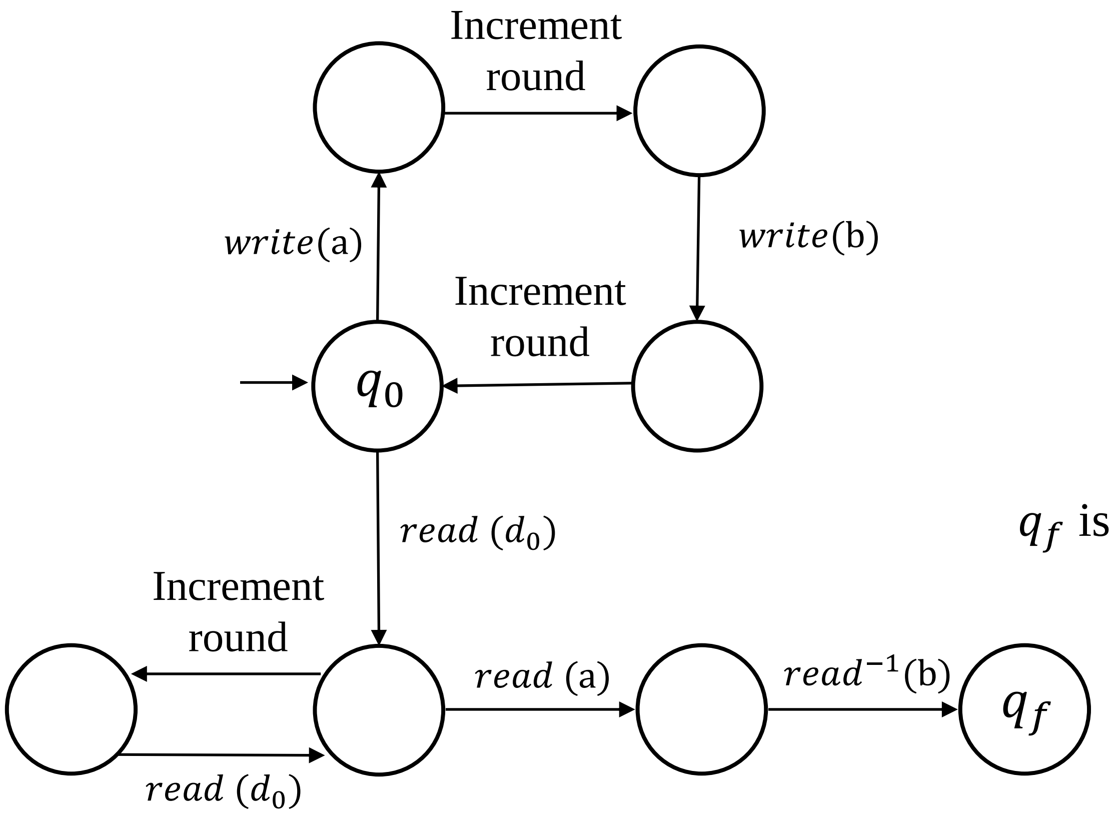

Increment
round
2
write(a)
write(b)
Increment
round
q_0
read (d_0)
Increment
round
read (a)
2
$read^{-1}$(b)
q_f
read (d_0)


⋮ ⋮
2 a
1 b
0 a

An example

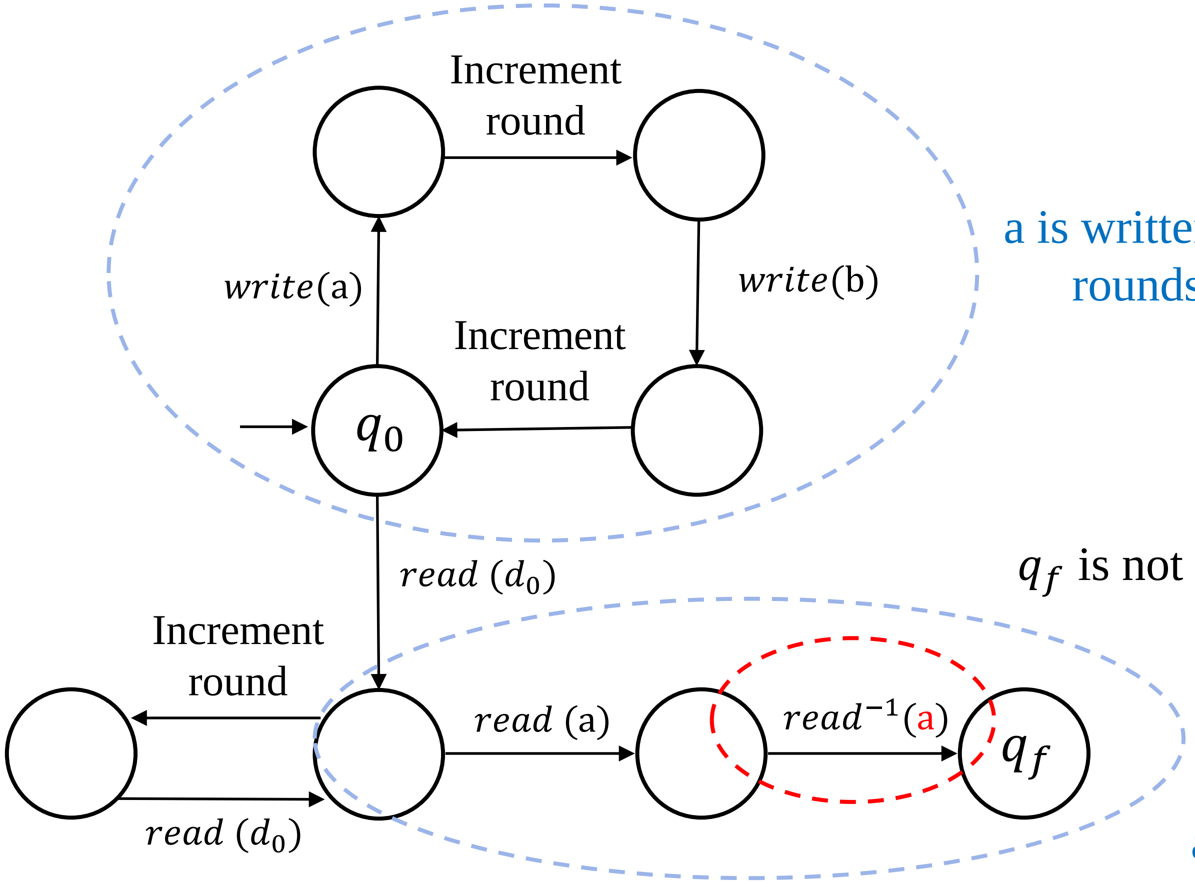

Increment
round
2
write(a)
write(b)
Increment
round
q_0
read (d_0)
Increment
round
read (a)
read^{-1}(b)
2
q_f
read (d_0)
⋮ ⋮
2 a
1 b
0 a

An example



q_f is coverable ✓

An example



a is written to even rounds only

q_f is not coverable

a must be read two rounds in a row

Complexity result

Theorem[5]: COVER is PSPACE-complete.

5. Bertrand, N., Markey, N., Sankur, O., W.: *Parameterized safety verification of round-based shared-memory systems*. ICALP, 2022

Conclusion

General aim : automated methods for verification of distributed systems using model checking.

Parameterized verification:

- Systems of arbitrary number of participants
- If algorithm says yes, then the system is correct regardless of the number of participants
- Efficient techniques thanks to copycat properties

In this talk:

- Simple model for shared-memory systems with finite memory
- More complex model for round-based systems

Conclusion

General aim : automated methods for verification of distributed systems using model checking.

Parameterized verification:

- Systems of arbitrary number of participants
- If algorithm says yes, then the system is correct regardless of the number of participants
- Efficient techniques thanks to copycat properties

In this talk:

- Simple model for shared-memory systems with finite memory
- More complex model for round-based systems

Thanks for your attention ! Any questions?