Modelling Reversible Systems

and way back: 12 years of reversibility

NiRvAna kickoff meeting Fano

Claudio Antares Mezzina

Reversibility: Historical Reasons

Landaurer Principle (IBM) 1961

“any logically irreversible manipulation of information, such as the erasure of a bit or the merging of two computation paths, must be accompanied by a corresponding entropy increase in non-information-bearing degrees of freedom of the information-processing apparatus or its environment”

- A so-called logically reversible computation, in which no information is erased, may in principle be carried out without releasing any heat.
- This has led to considerable interest in the study of reversible computing.

Reversible Computing: History

Bennet 1973: reversible Turing machine

- A Turing machine with 3 tapes: input tape, output tape and history tape
- Theorem: For every standard one-tape Turing machine S, there exist a three-tape reversible, deterministic Turing machine R that has the same functionality as S

Reversible Circuits

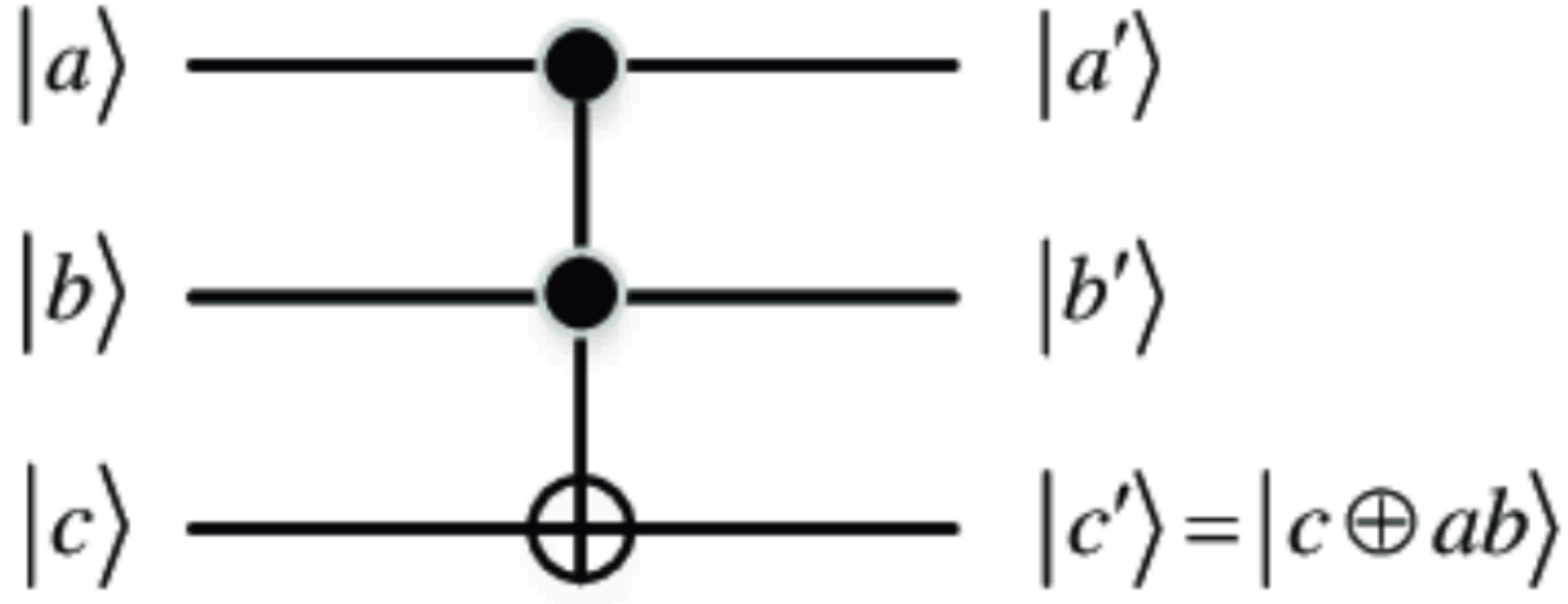
To implement reversible computation, estimate its cost, and to judge its limits, it can be formalized in terms of gate-level circuits.

Toffoli gate 1980: a 3-input invariant gate

- It preserves two of the 3 input
- Replaces the third by
- With c=0 we get the AND port $c \oplus (a \wedge b)$
- With c=1 we get the NAND port
- With a = 1 OR b = 1 we get the XOR
- It is an universal gate

Toffoli gate and quantum circuits

Inputs			Ouputs		
a	b	c	a'	b'	c'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0



Reversible Computation on the hype

https://spectrum.ieee.org/the-future-of-computing-depends-on-making-it-reversible

IEEE Spectrum

Type to search

Explore by topic

FEATURE | COMPUTING

THE FUTURE OF COMPUTING DEPENDS ON MAKING IT REVERSIBLE

It’s time to embrace reversible computing, which could offer dramatic improvements in energy efficiency

Aside Circuits

Reversibility or reversible behaviour can be found in other fields

- System biology
- Transaction / Checkpoint Rollback Schema / Failure handling primitives
- Reversible Debugging
- Record/Replay (reproducibility of system behaviour)
- Quantum computing

Reversible computation

- We can image two directions of computations: forward and backward
- Which action we undo first?
    - In a sequential setting simple: we undo a computation starting from the last action (backtracking)
    - In a concurrent/distributed system?
        - No concept of last action
        - No global clock

Reversibility in Concurrent System

A good approximation is causal consistent reversibility

Causal consistent reversibility relates reversibility and causality

It allows to consider as last action any action which as no consequences:

in a concurrent system, any action can be undone provided that all of its consequences, if any, are undone beforehand.

Reversibility in Concurrent System

Modelling

- Reversible Process Algebras
- Reversible Petri nets
- Reversible Event Structure

Reversibility in Concurrent System

Calculi

Reversible Communicating System (RCCS) Danos&Krivine

- Use of explicit memories to keep track of past events
- Suitable for complex languages (e.g., scales with pi-calculus, Erlang)

CCS with communication keys (CCSK) Phillips&Ulidowski

- History information directly recorderded into the term
- Use of keys to keep track of synchronisations
- Suitable for CCS-like languages with LTSs

Example

$$a.P + b.Q \xrightarrow{a} P$$

After the computation, we loose information about

- The performed action a
- The other branch b.Q

RCCS

$$m \triangleright (a.P + b.Q) \xrightarrow{a[i]} \langle a, b.Q, i \rangle \cdot m \triangleright P \xrightarrow{a[i]} m \triangleright (a.P + b.Q)$$

Memory monitoring the process

Information about the previous state

CCSK

$$\underline{a.P + b.Q} \xrightarrow{a[i]} a[i].P + b.Q \xrightarrow{a[i]} a.P + b.Q$$

No need of extra memories

History information directly in the term

Results

Acta Informatica (2021) 58:1–34
https://doi.org/10.1007/s00236-019-00346-6

ORIGINAL ARTICLE

Check for updates

Static versus dynamic reversibility in CCS

Ivan Lanese[2,3] · Doriana Medić[1,2,3] · Claudio Antares Mezzina[4]

Received: 13 June 2018 / Accepted: 22 October 2019 / Published online: 7 November 2019

Abstract
The notion of reversible computing is attracting interest because of its applications in diverse fields, in particular the study of programming abstractions for fault tolerant systems. Most computational models are not naturally reversible since computation causes loss of information, and history information must be stored to enable reversibility. In the literature, two approaches to reverse the CCS process calculus exist, differing on how history information is kept. Reversible CCS (RCCS), proposed by Danos and Krivine, exploits dedicated stacks of memories attached to each thread. CCS with Keys (CCSK), proposed by Phillips and Ulidowski, makes CCS operators static so that computation does not cause information loss. In this paper we show that RCCS and CCSK are equivalent in terms of LTS isomorphism.

The two approaches are equivalent

Cross-Fertilization results

More expressive power: pi calculus

Reversible Higher order Pi calculus

$$a\langle P\rangle \parallel a(X).Q \rightarrow Q\{^{P}/_{X}\}$$

Message content is a process

Substitution is not bijective

Reversible Higher Order Pi: rhoPI

- Unique identifier per process
- “Dumping” the previous state

$$\kappa_1 : a\langle P\rangle \parallel \kappa_2 : a(X).Q \rightarrow \nu k\,(k : Q\{^P/_X\} \parallel [\kappa_1 : a\langle P\rangle \parallel \kappa_2 : a(X).Q; k]$$

Process identifiers identifiers

Memory / Snapshot

RhoPi rules

$$\text{(R.Fw)} \quad (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q) \twoheadrightarrow \nu k.\,(k : Q\{^P/_X\}) \mid [(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q); k]$$

$$\text{(R.Bw)} \quad (k : P) \mid [M; k] \rightsquigarrow M$$

Controlling reversibility

- So far we have seen uncontrolled reversibility
- Each step can be undone
- Rules free to be triggered
- We want to enable reversibility as a reaction to a failure
- Reversible steps should be triggered by a specific command (e.g., a rollback)

Controlling Reversibility in rhoPi

- We want an operator which is able to bring the system before the happening of an event
- E.g., we want to undo an event along with its computational history
- We use a specific rollback operator

Rollback operator

Communication rule as before

$$\text{(H.Com)} \quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \rhd_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \rhd_\gamma Q) \twoheadrightarrow \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(H.Start)} \quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^\bullet$$

Marks the snapshot we want to restore

$$\text{(H.Roll)} \quad \frac{N \blacktriangleright k \qquad \mathtt{complete}(N \mid [\mu; k])}{N \mid [\mu; k]^\bullet \rightsquigarrow \mu \mid N \lightning_k}$$

Part of the system which is caused by k

Rollback operator: implementation

- The previous semantics uses a big atomic step to undo an entire computational history
- Works as a High Level specification
- We have implemented a low-level semantics (based on message) which is bisimilar to the HL one

Reversible Debuggers

Reverse debugging is the ability of a debugger to stop after a failure in a program has been observed and go back into the history of the execution to uncover the reason for the failure [Jakob Engblom, S4D 2012]

Implications:

- Ability to execute a program both in forward and backward way
- Reproduce or keep track of the past of an execution

Reversible debuggers

- GDB version 7.0 (September 2009) supports reversibility
  - step -> reverse-step, next -> reverse-next
- UndoDB improves GDB history bookkeeping
- Mozilla RR, Microsoft Intellitrace and many more

Reversible Debuggers: state of the art

Non-deterministi replay

The execution is replayed non deterministically from the start (or from a previous checkpoint) till the desired point.

Deterministic replay/reverse-execute debugging

A log of the scheduling among threads is kept and then actions are reversed or replayed accordingly.

Causal Consistent Rev Deb

Actions are reversed respecting the causes

- Only actions that have caused no successive actions can be undone
- Concurrent actions can be reverted in any order
- Dependent action are reverted starting from the consequences

Benefit

The programmer can easily individuate and undo the actions that caused a given misbehaviour.

CareDeb: Fase2014

Table 1. CaReDeb main commands

control	**forth (f) t**	(forward execution of one step of thread `t`)
	run	(runs the program)
	rollvariable (rv) id	(causal-consistent undo of the creation of variable id)
	rollsend (rs) id n	(causal-consistent undo of last n send to port `id`)
	rollreceive (rr) id n	(causal-consistent undo of last n receive from port `id`)
	rollthread (rt) t	(causal-consistent undo of the creation of thread `t`)
	roll (r) t n	(causal-consistent undo of n steps of thread `t`)
	back (b) t	(backward execution of one step of thread `t` (if possible))
explore	**list (l)**	(displays all the available threads)
	store (s)	(displays all the ids contained in the store)
	print (p) id	(shows the state of a thread, channel, or variable)
	history (h) id	(shows thread/channel computational history)

CauDEr

A Causal-Consistent Reversible Debugger for Erlang.

Erlang/OTP 23.0 test passing License MIT

This tool is still under development

Core Erlang version

In 2020, we decided to rewrite CauDEr to work directly with Erlang instead of Core Erlang. The main reasons for this change where simplicity, user-friendliness and breaking changes introduced in newer version of Erlang/OTP.

Back to system modelling: biology

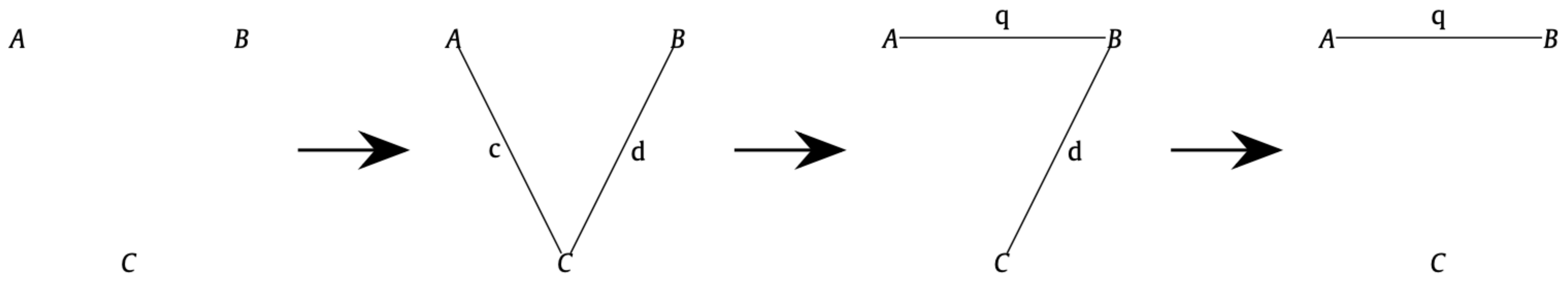


Fig. 1. A catalytic reaction (borrowed from [8]).

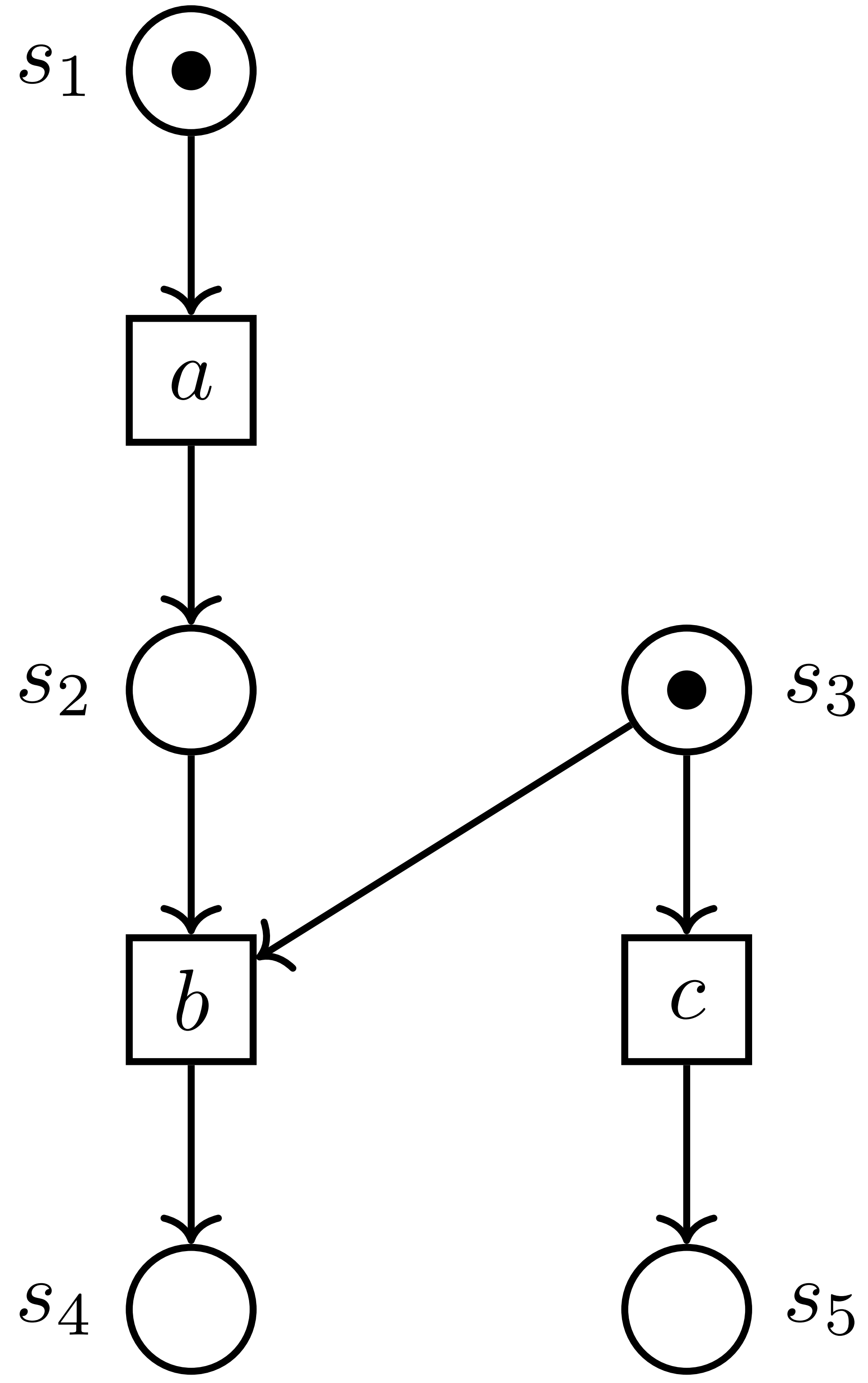
Sometimes causes are not respected: out-of-causal-order reversibility

A further step back

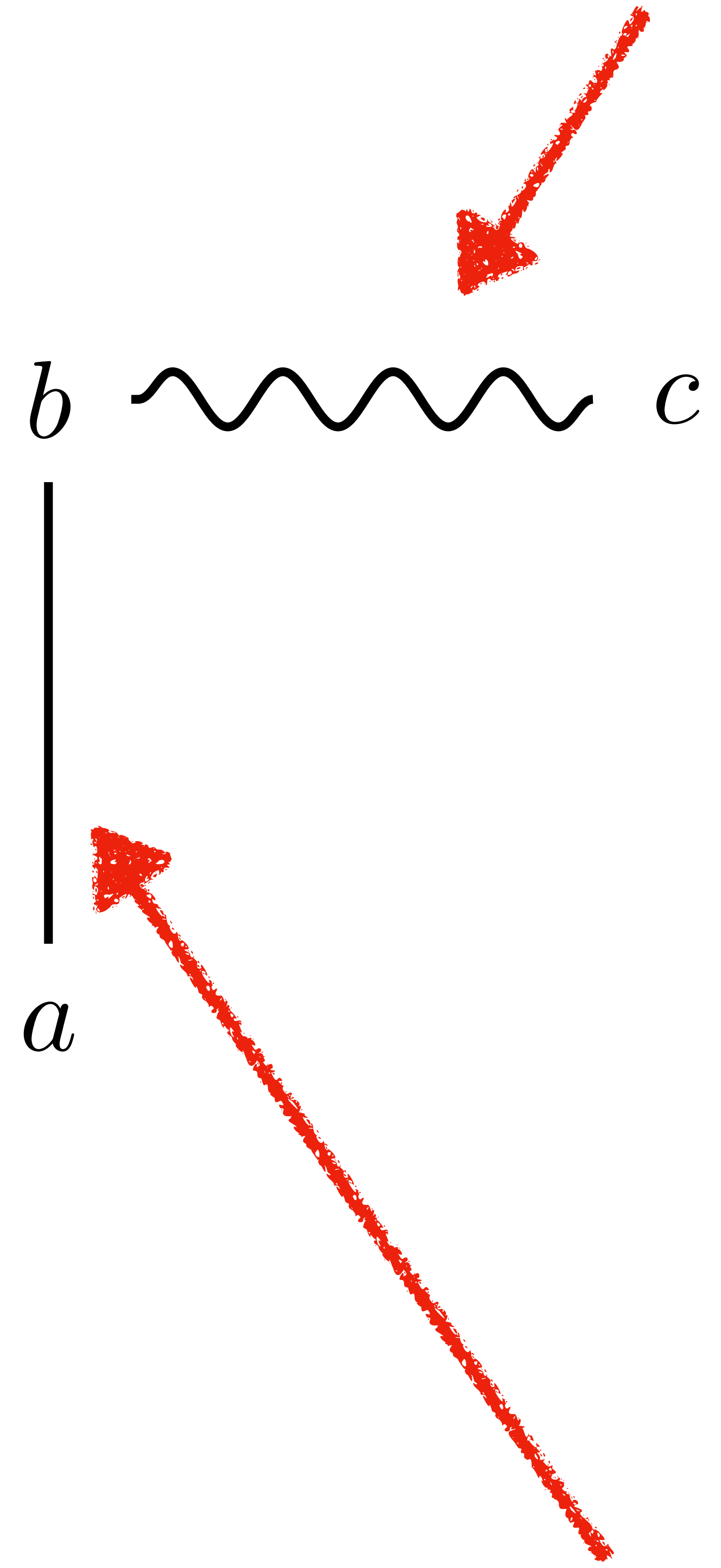
Two well-known models to describe concurrent systems:

- Event structures
  - Event occurrences and constraints on events
  - Denotational view of a system
- Petri nets
  - Consumption / production of data from repositories
  - Places, tokens, transitions
  - Operational view of a system

Example



b and c are in conflict



b causally depends on a

$$
\begin{tikzcd}
\emptyset \arrow[r] \arrow[d] \arrow[dr] & \{a\} \arrow[r] \arrow[d] & \{a, b\} \\
\{c\} \arrow[r] & \{a, c\} &
\end{tikzcd}
$$

Since b and c are in conflict there is no configuration containing both

If b is present in a configuration then also a is present

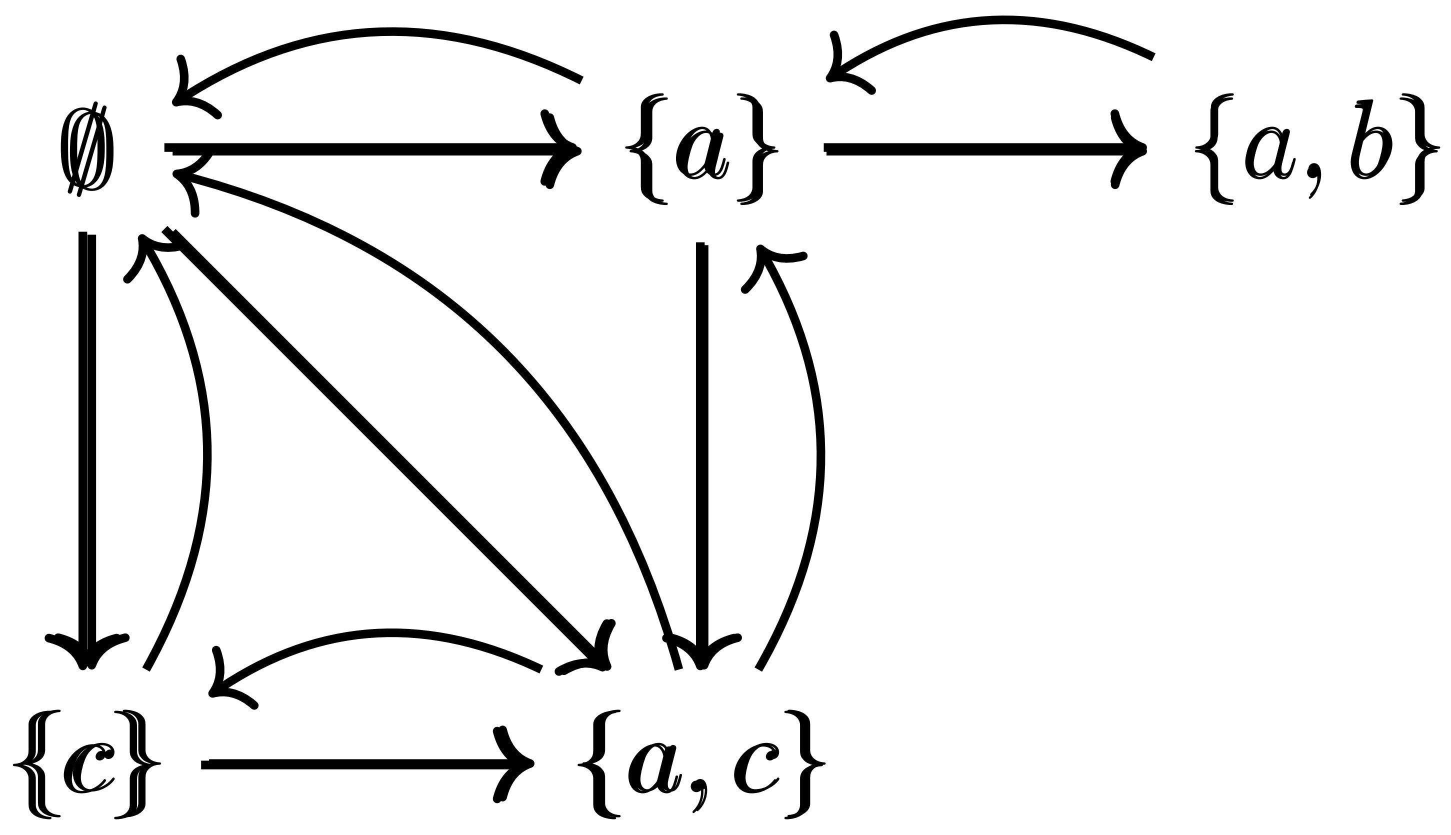
A further step back 2/2

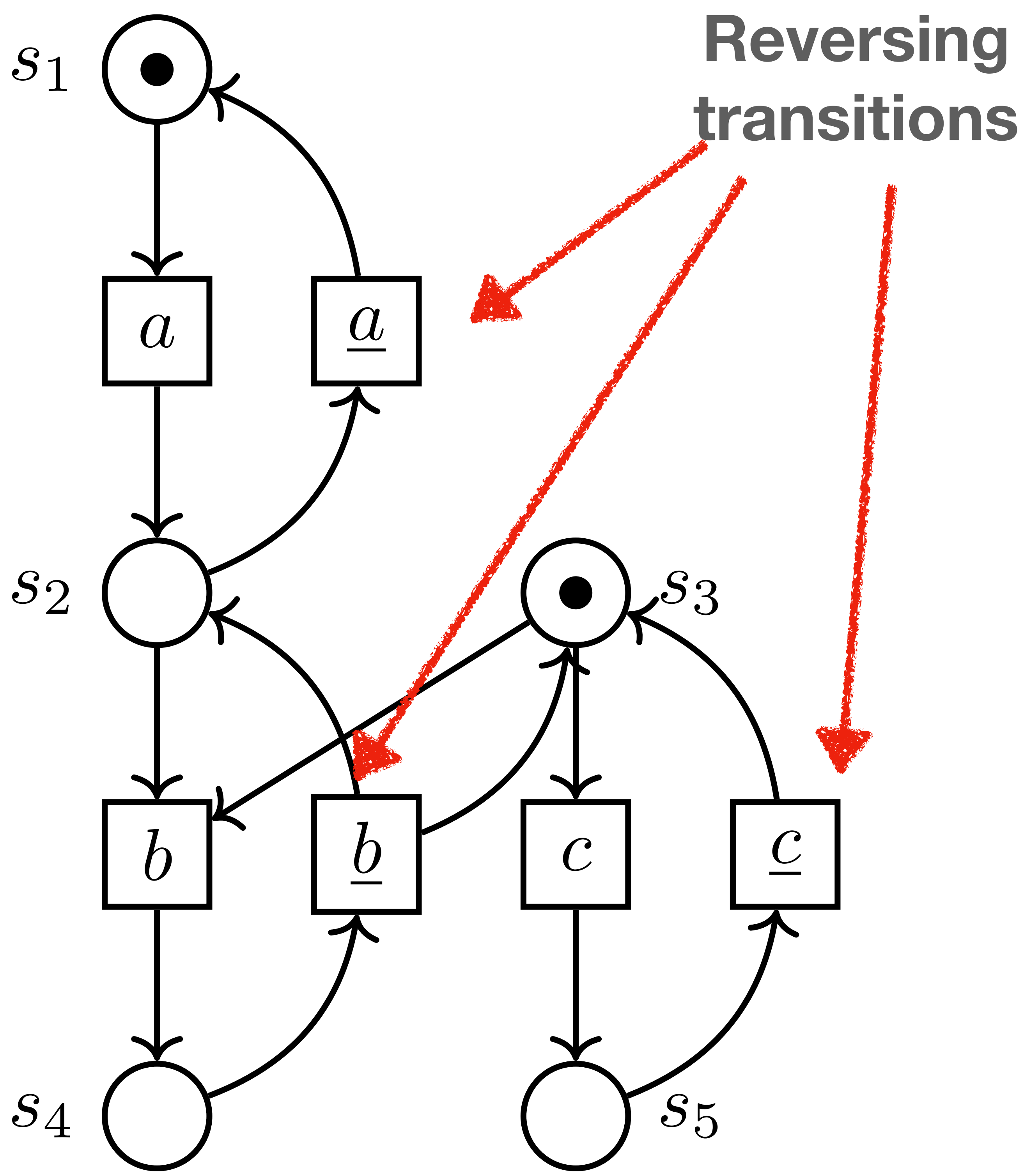
- A seminal work of Winskel showed a relation between Occurrence Nets (ON) and Prime Event Structure (PES)
- A lot of effort connecting guises of event structure with their nets counterpart
- Lately PESs have been extended to account for reversible computing
  - accomodate the undoing of executed actions by removing events from configurations
  - accounts for different kinds of reversibility: backtracking, causal-respecting (transactions / checkpoint rollback) and out-of-order (biochemical reactions)

Reversibility on Nets (so far)

- Melgratti, Mezzina & Ulidowski proposed a causal reversible semantics for 1-safe petri nets by exploiting the natural unfolding in ON
    - from ON reversible ON (RON) are derived
- Psara & Philippou proposed a new model of PT able to capture all the three kinds of reversibility
    - uses ad-hoc tokens to keep track of the path
    - uses extra information (histories) to store the scheduling

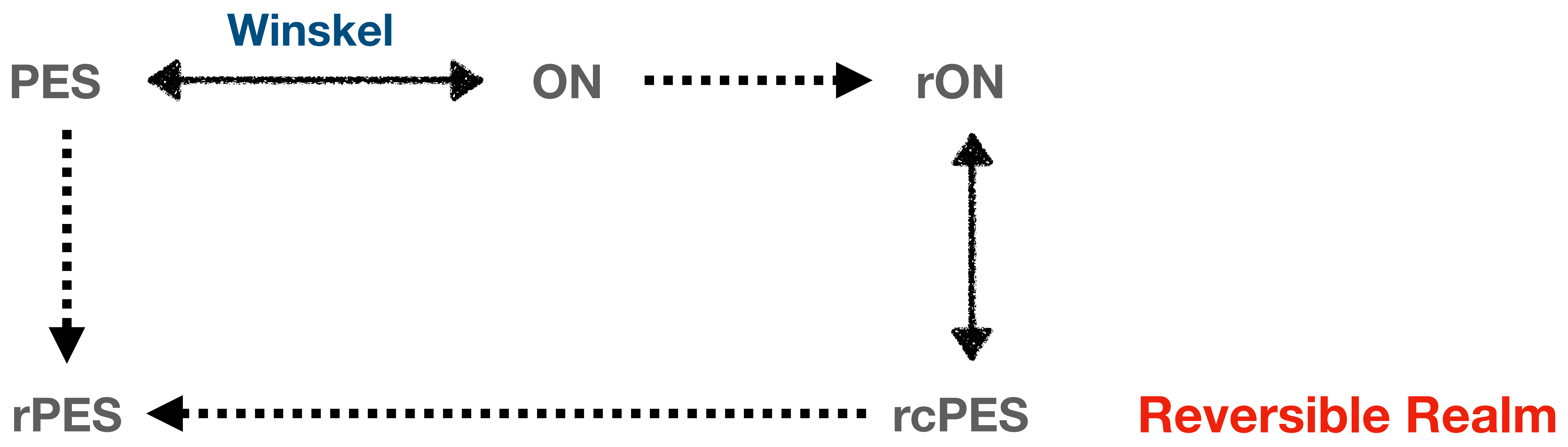
Background: Simple idea





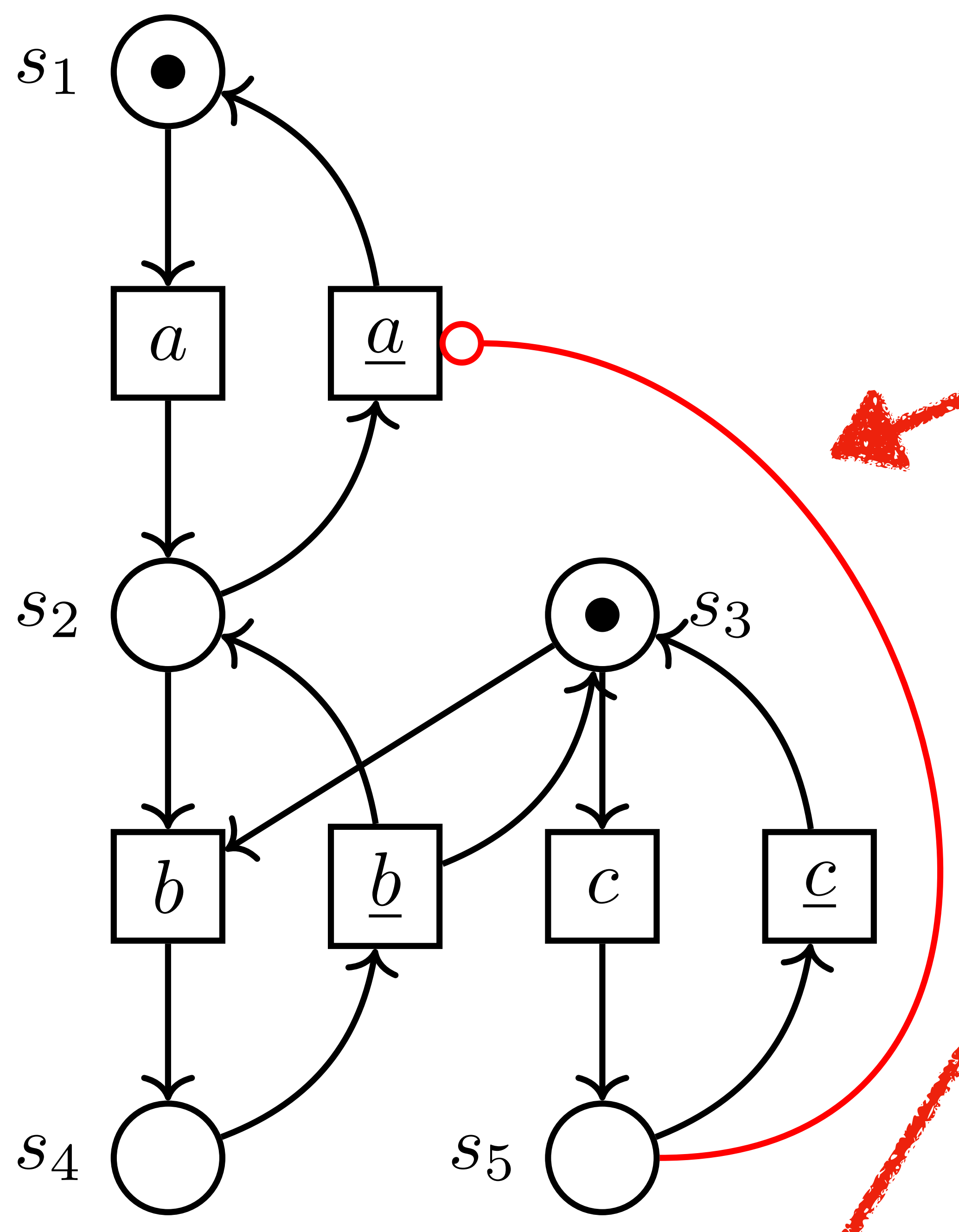
- This simple idea works just for causal order reversibility
- rPES are more expressive as they use *prevention* and *reverse causality* operators

So far



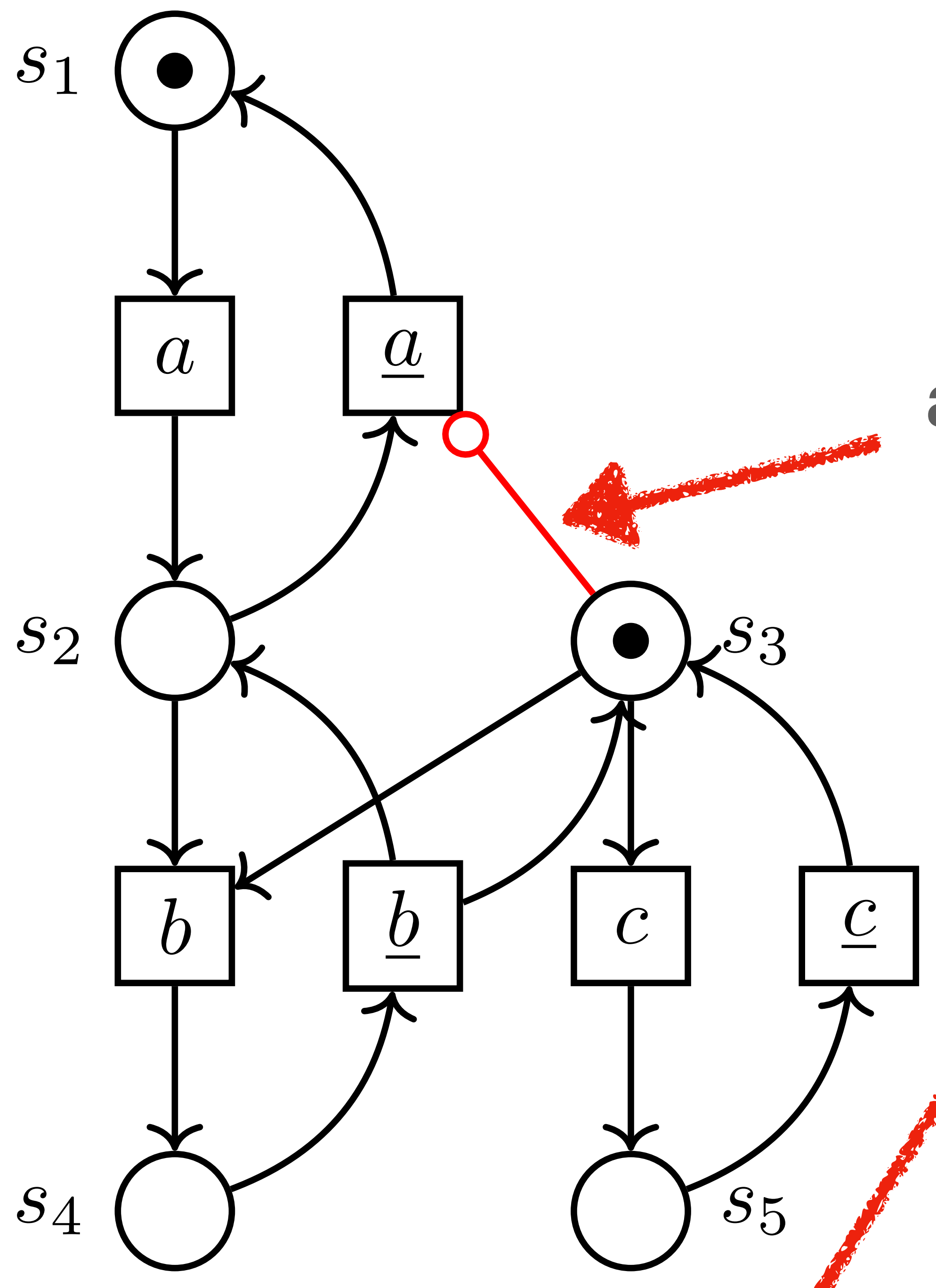
- Melgratti et al. shown a correspondence between RON and causal RPES (e.g., RPES with just causal reversibility)
- Failed in the general case

Prevention and Reverse Causality



(a) $c \triangleright \underline{a}$

c **prevents**
the undoing of a



(b) $c \prec \underline{a}$

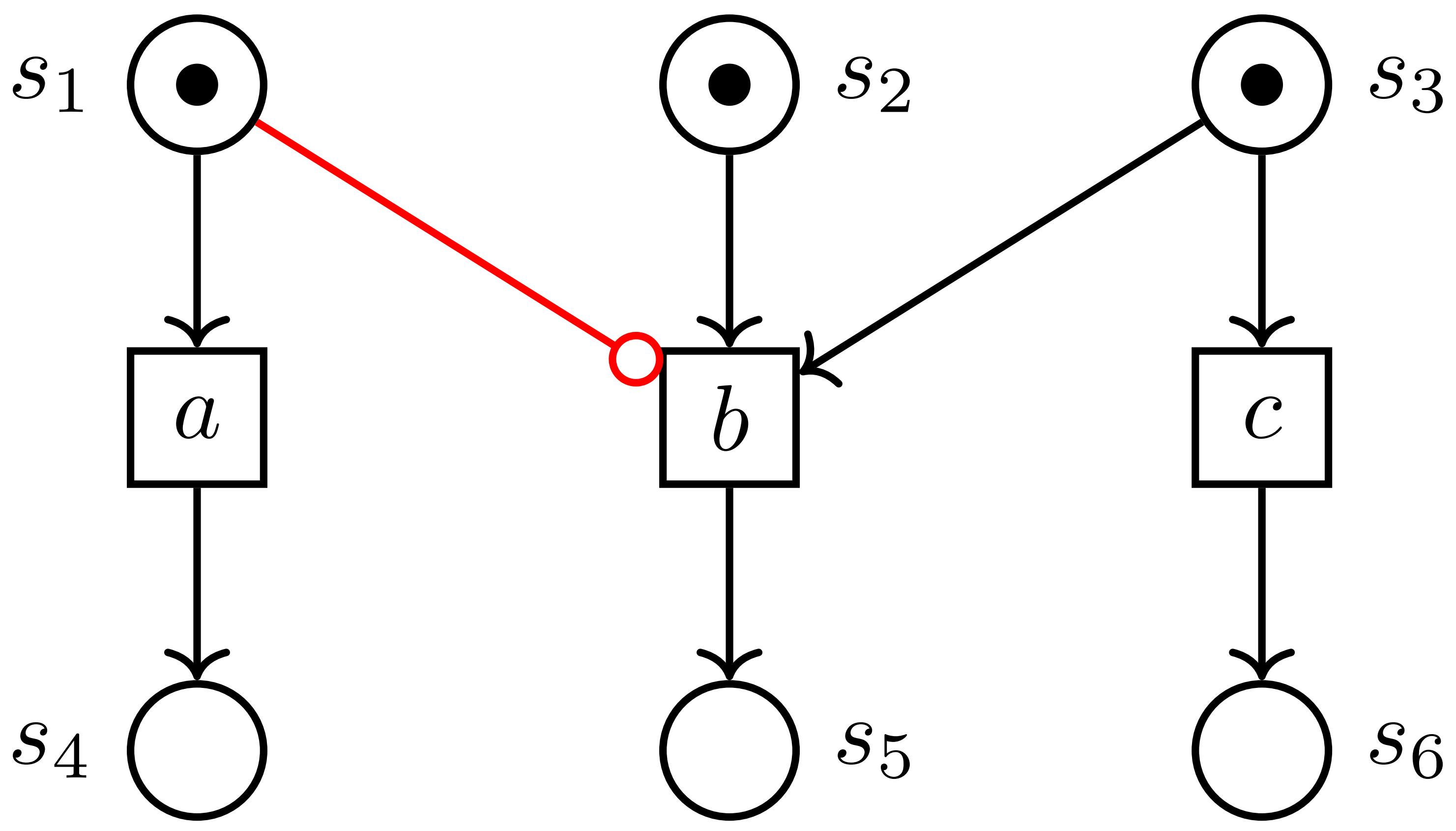
a can be undone if c **happens**

Two questions

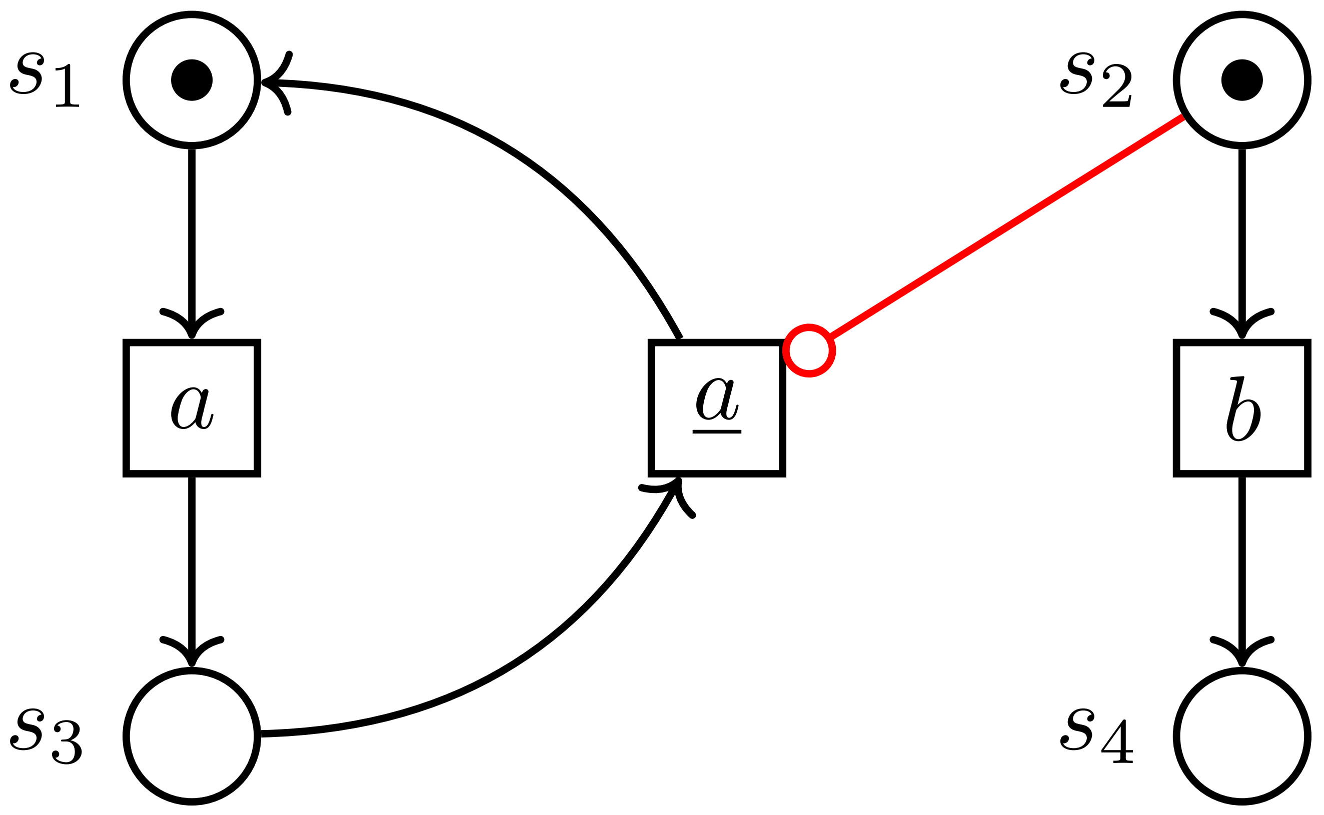
- Which kind of net can ben associated with an rPES?
- Can we do it by relying on standard notion of Petri nets?

Answer

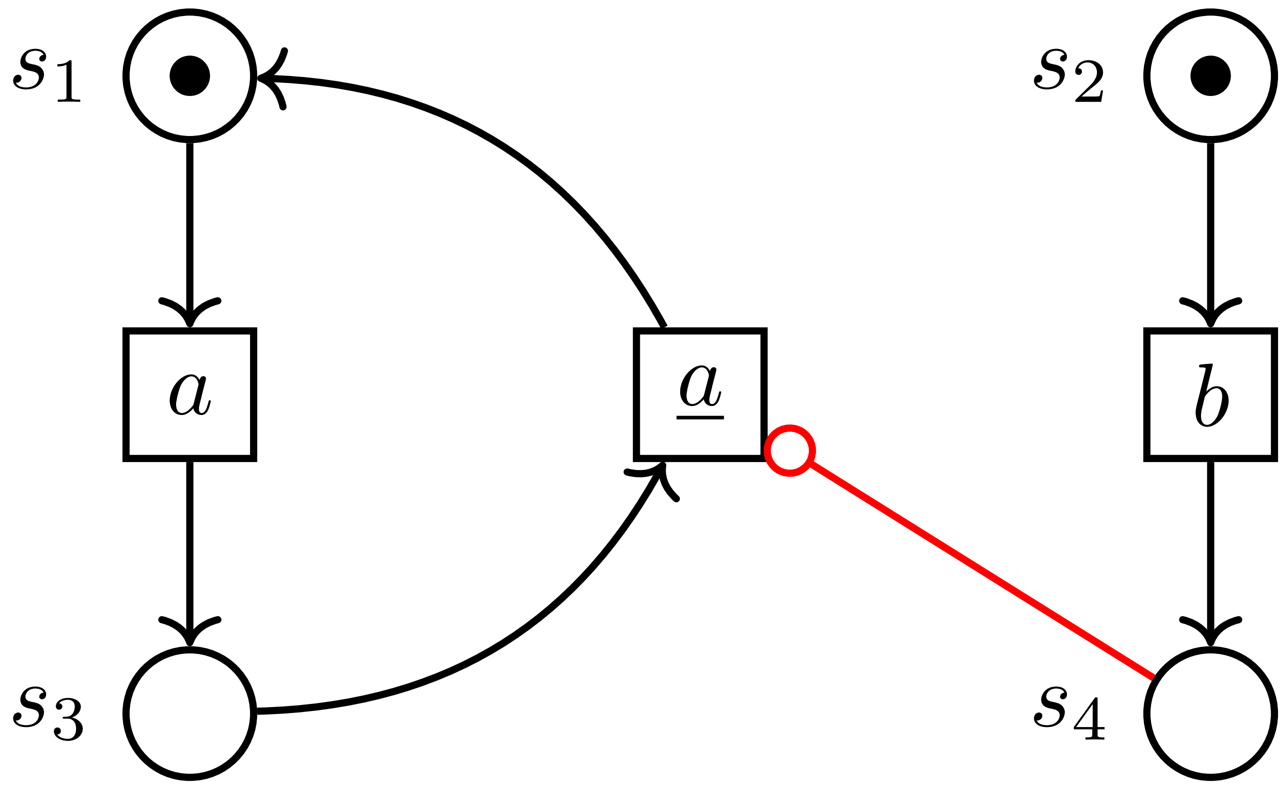
Inhibitor arcs can be used to model **causality**, but also more complex relations such as **reverse causation** and **prevention**



$a < b \quad b\#c$



$b \prec \underline{a}$



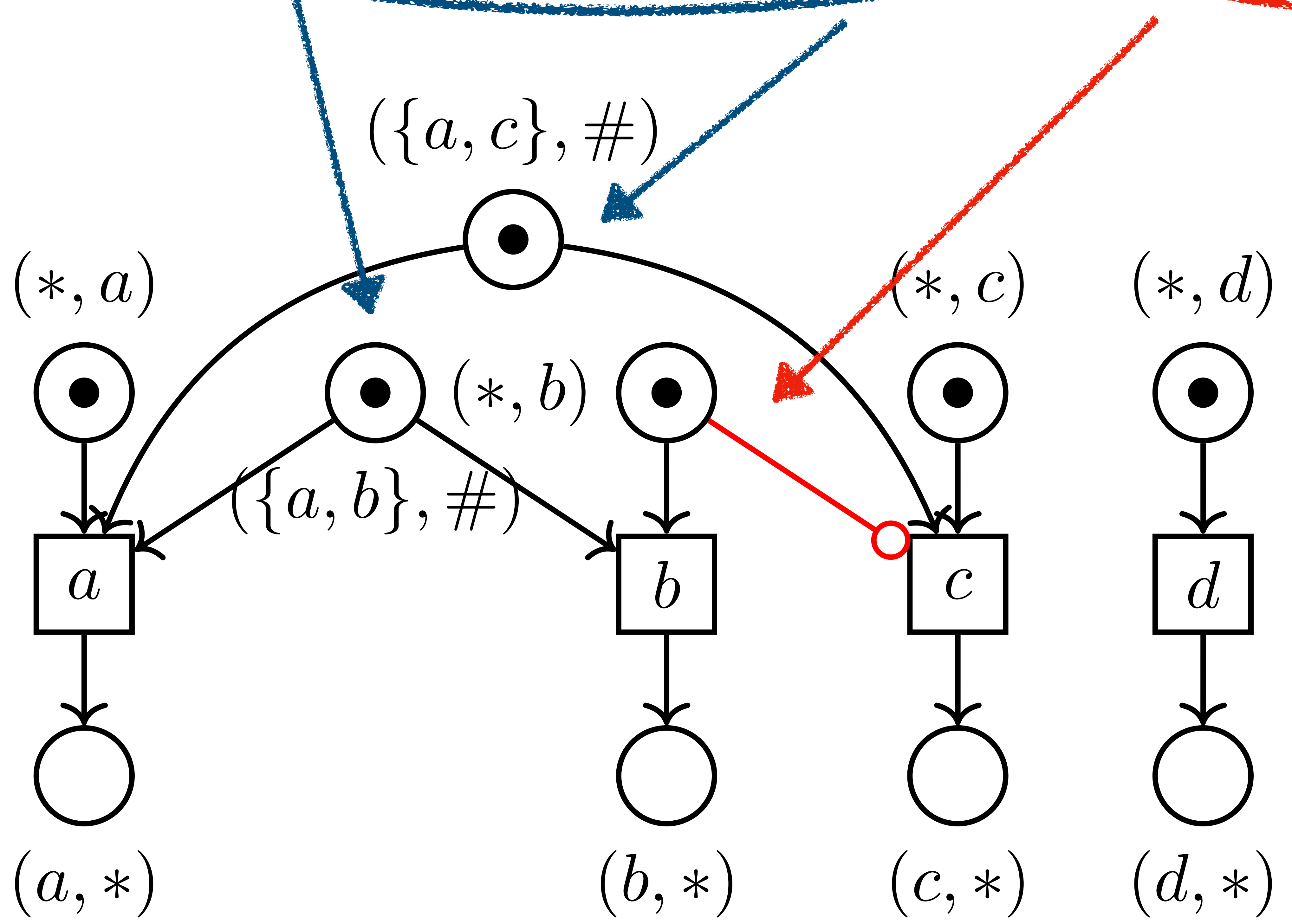
$b \triangleright \underline{a}$

Roadmap

- We first introduce causal nets (**CN**), where causality is modelled by inhibitor arcs instead of the classic flow relation
- We show that **CN** are the right model for **PES**
- We show that **rCN** are the right model for **rPES**
- We show that **ON** can be modelled into **CN**

Causal Nets and PES

$E = \{a, b, c, d\}$ $\quad \# = \{(a,b),(b,a),(a,c),(c,a)\}$ $\quad < = \{(b,c)\}$



rCN and rPES

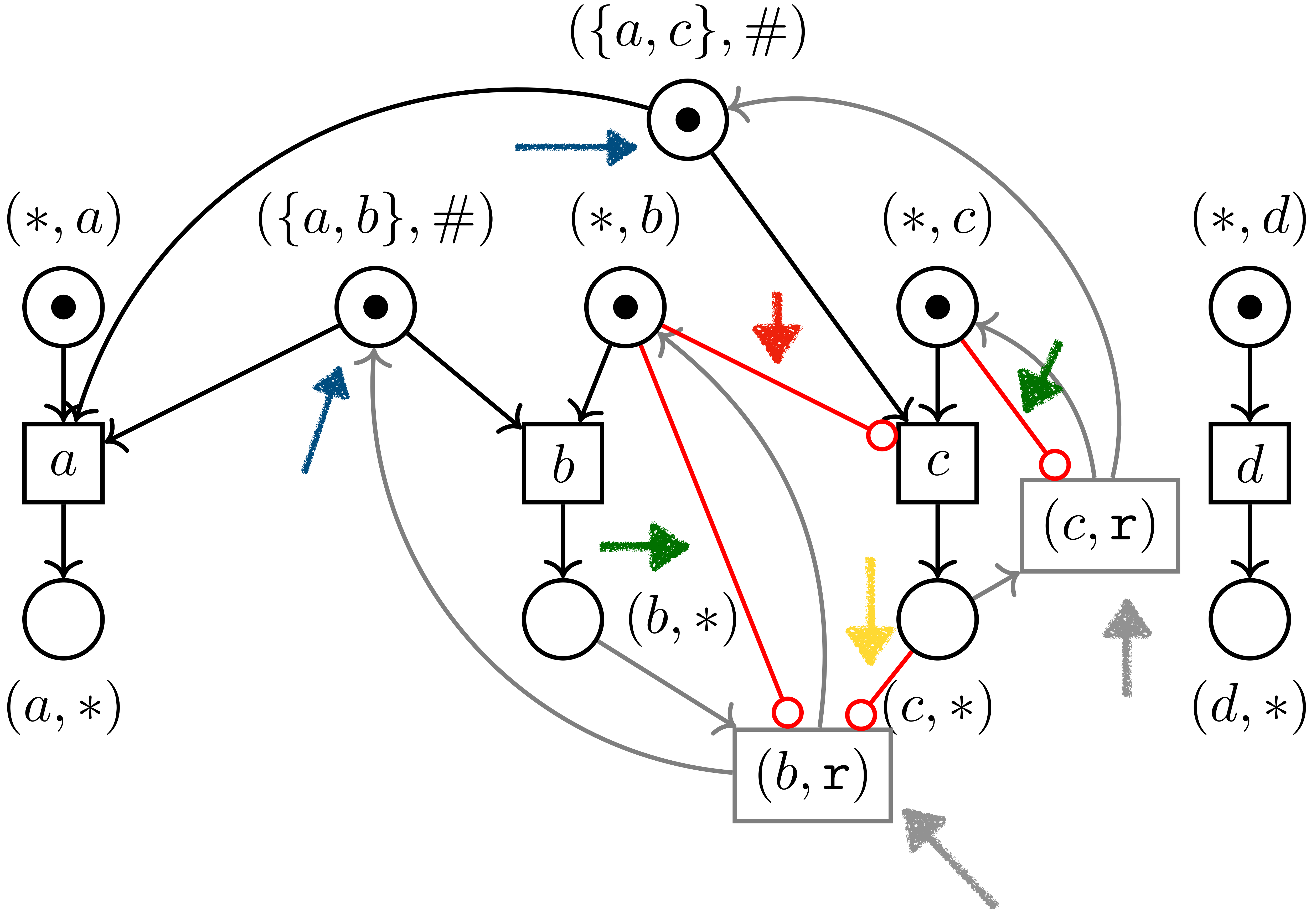
reversibe events

conflict

$$E = \{a, b, c, d\} \qquad U = \{b, c\}$$

causality $\quad < \, = \{(b, c)\} \qquad \# = \{(a, b), (b, a), (a, c), (c, a)\}$

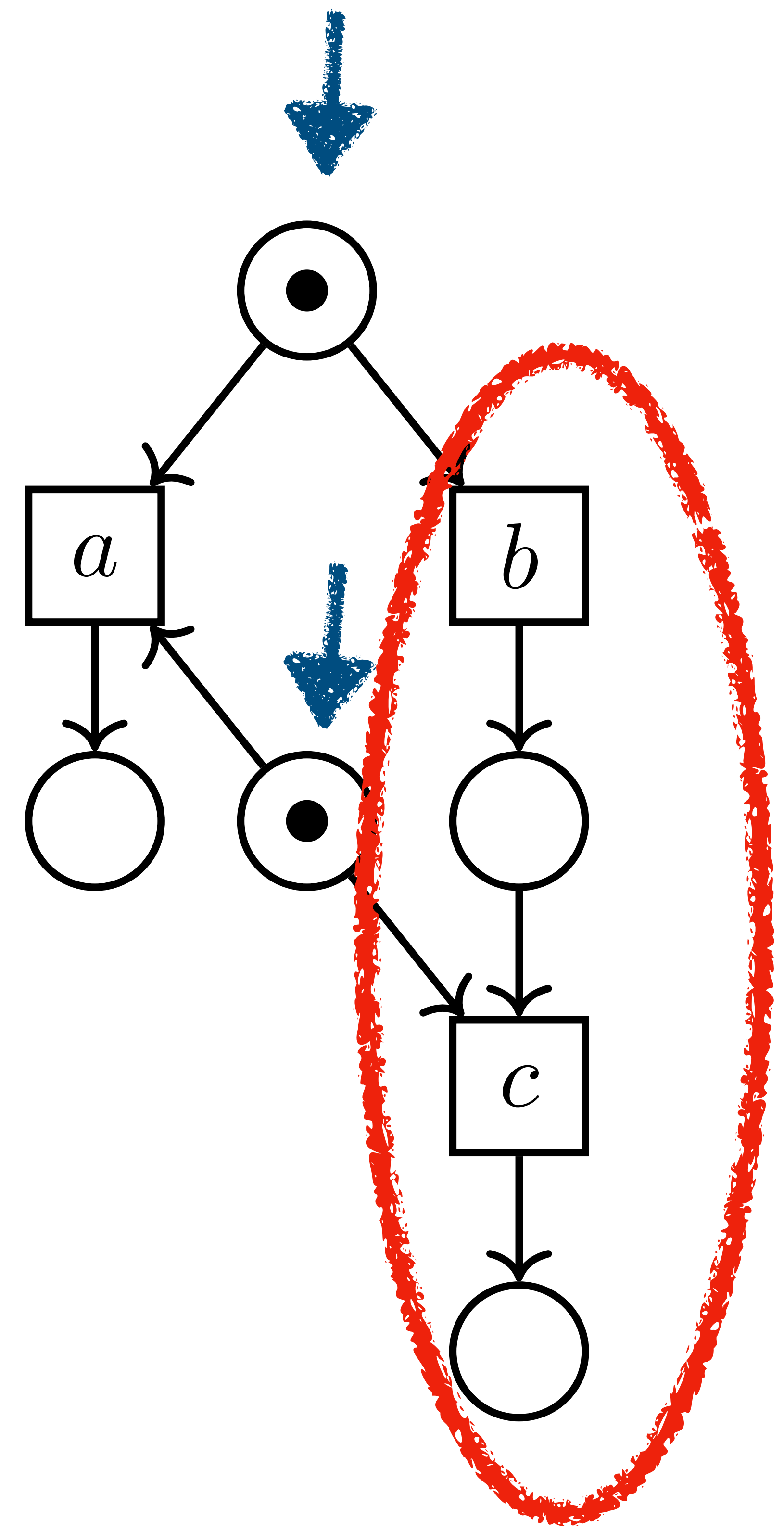
reverse causation $\quad \prec \, = \{(b, \underline{b}), (c, \underline{c})\} \qquad \triangleright = \{(c, \underline{b})\}$ Prevention



Results

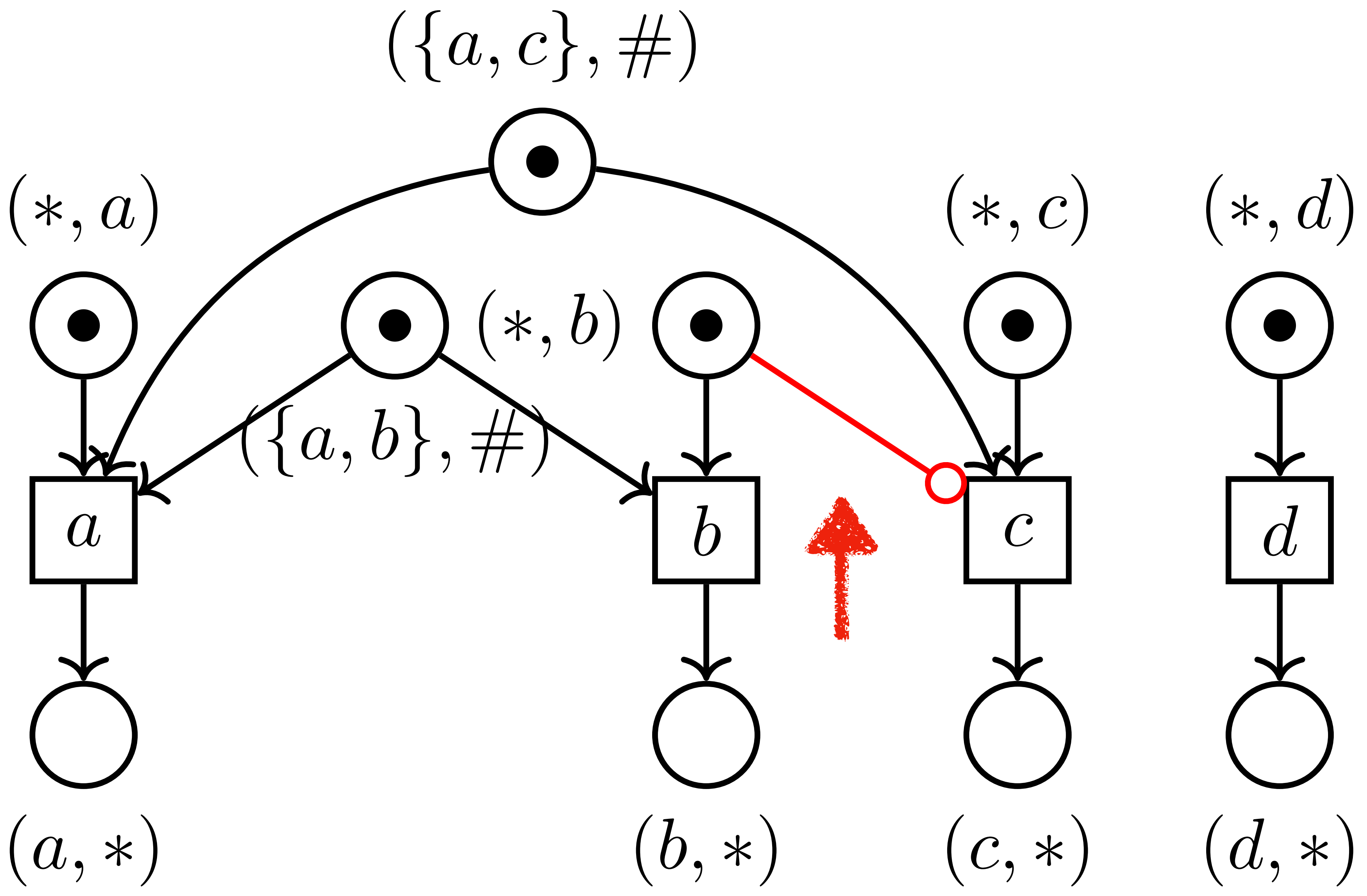
- (Reversible) Causal Nets are an operationally counterpart of (reversible) Prime Event Structures
- The key idea is that inhibitor arcs can model all the operator of (reversible) Prime Event Structures

From ON to CN and vice versa



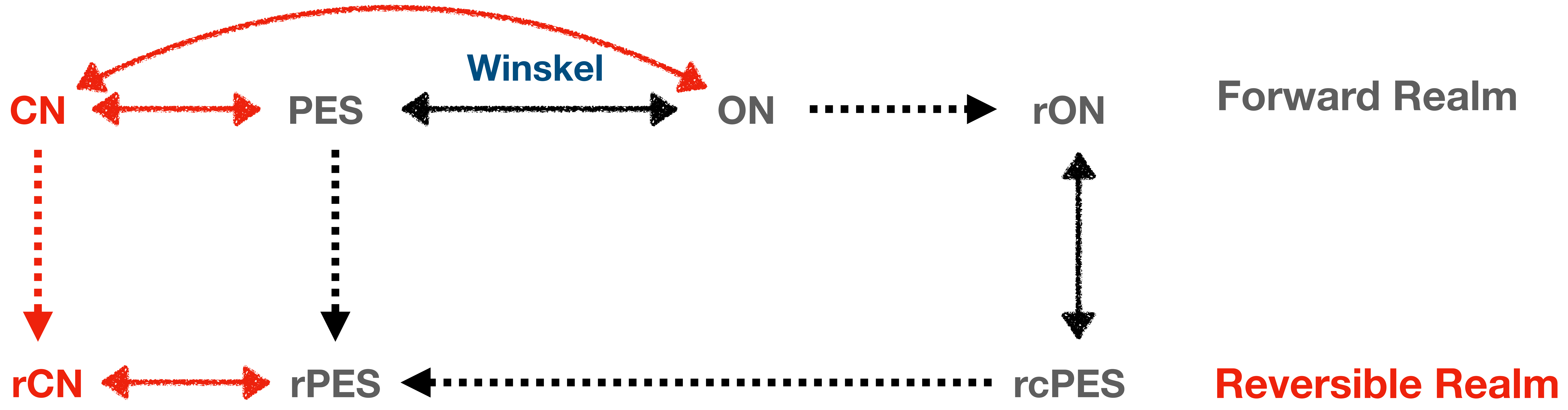
a # b
a # c

a < c



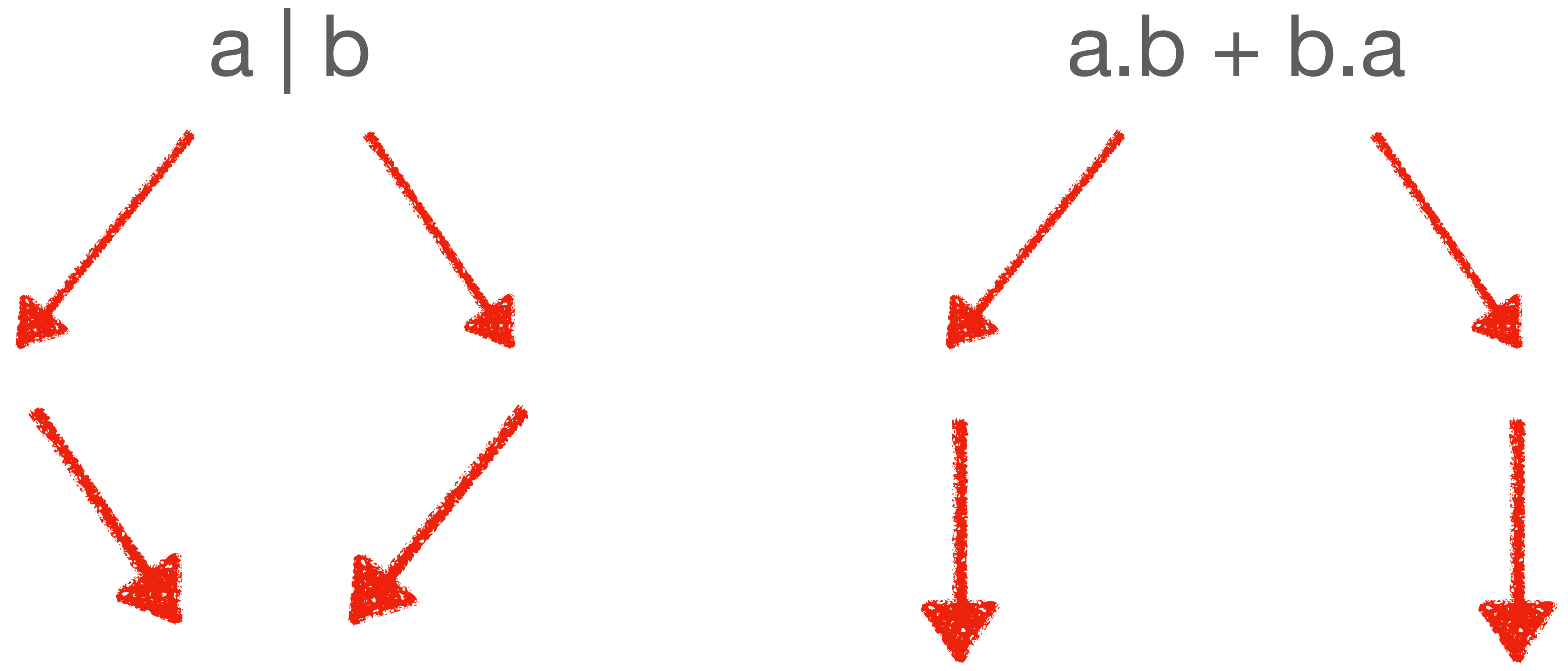
Causality is modelled directly via inhibitor arcs, not through the flow relation

Results Graphically



Toward a truly semantics for RCCS

- Back in the past there has been a lot of effort to give to CCS a true concurrent semantics
- CCS semantics is given in terms of an interleaved one



The two processes (and traces)

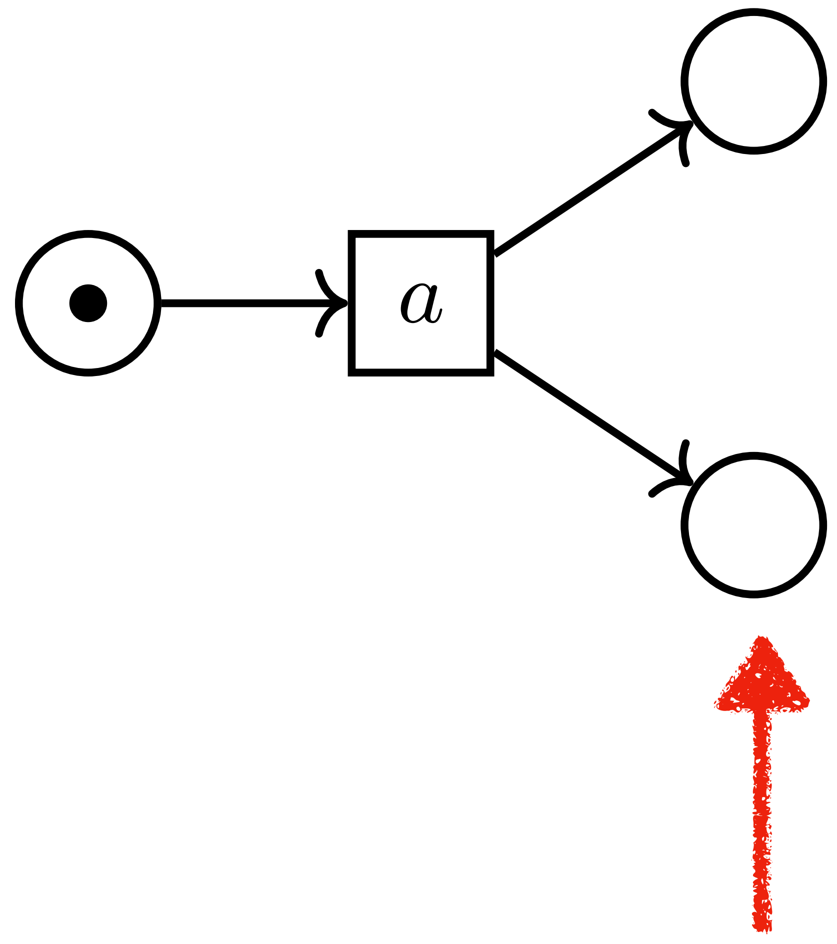
are deemed equivalent in CCS

Background

- Different works have given an truly concurrent semantics of CCS in
  - Occurrence Nets
  - Event Structures
  - Prime Event Structures
- What about reversible CCS?
- Two different flavours of reversible CCS: RCCS and CCSK but both are in the interleaved semantics
  - An interpretation of (controlled) CCSK in (reversible) bundle event structures [Graversen, Phillips, Yoshida 2021]

From CCS to Petri net: example

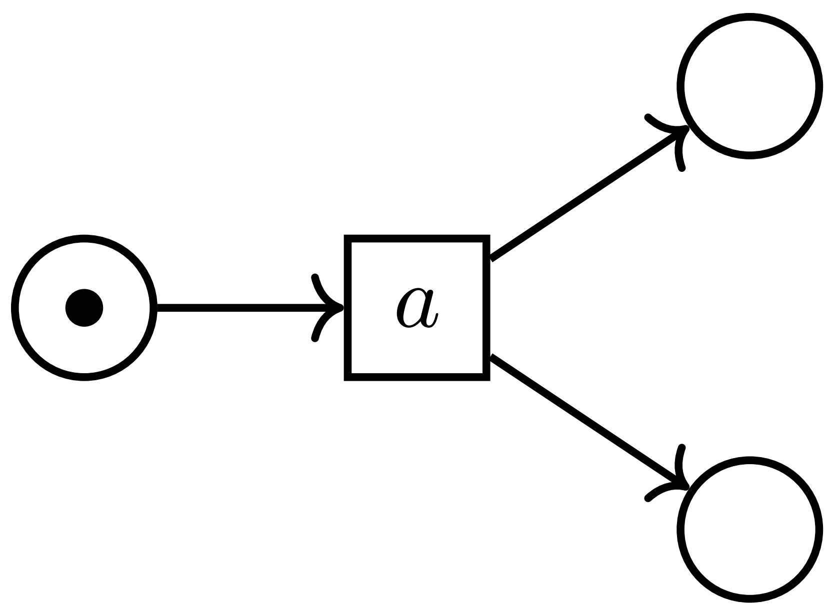
- The simple process a is encoded as a Petri net with one transition named a



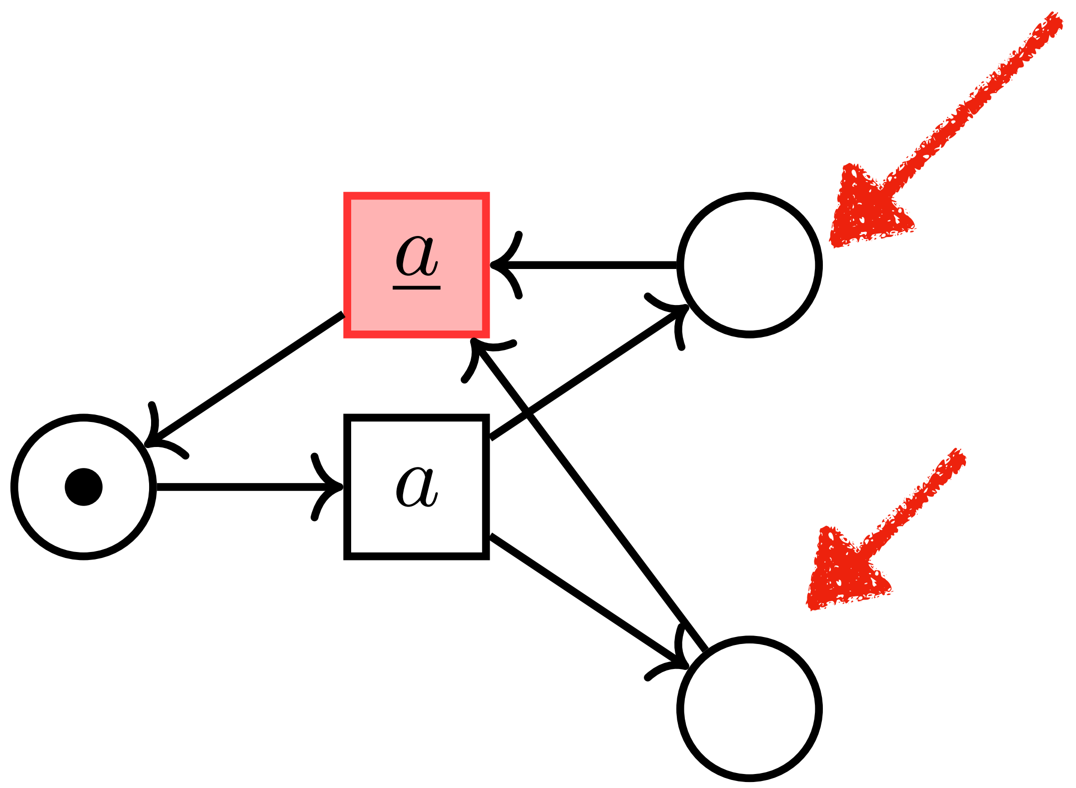
Redundant place in the post-set of a

How to reverse?

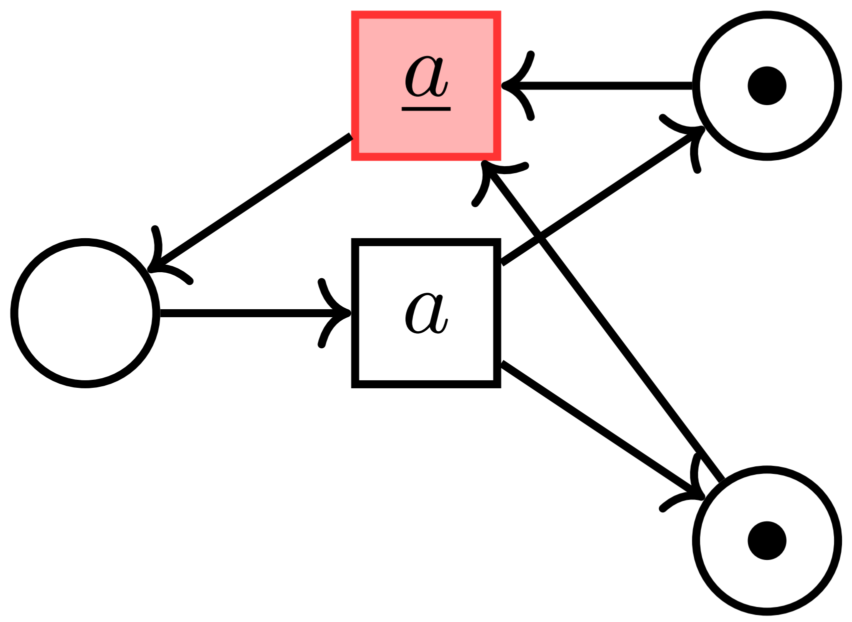
The preset of **a** are the postset of **a**



$$a$$



$$\langle\rangle \triangleright a$$



$$\langle a, *, 0\rangle \triangleright \mathbf{0}$$

- For each transition we create an exact inverse of it
- What changes from $\langle\rangle \triangleright a$ to $\langle a, *, 0\rangle \triangleright \mathbf{0}$ in terms of nets is the **marking**

A very simple idea

- The encoding of a CCS term into a net already bears all the information needed for reversing it
- This contrasts with RCCS memories (e.g., the need to add memories to remember)
- An initial RCCS term (e.g., with empty memory) and all its derivate have the same net, what changes is the marking
- Markings correspond to RCCS memories
- This simple observation gives an almost straightforward true concurrent representation of RCCS terms

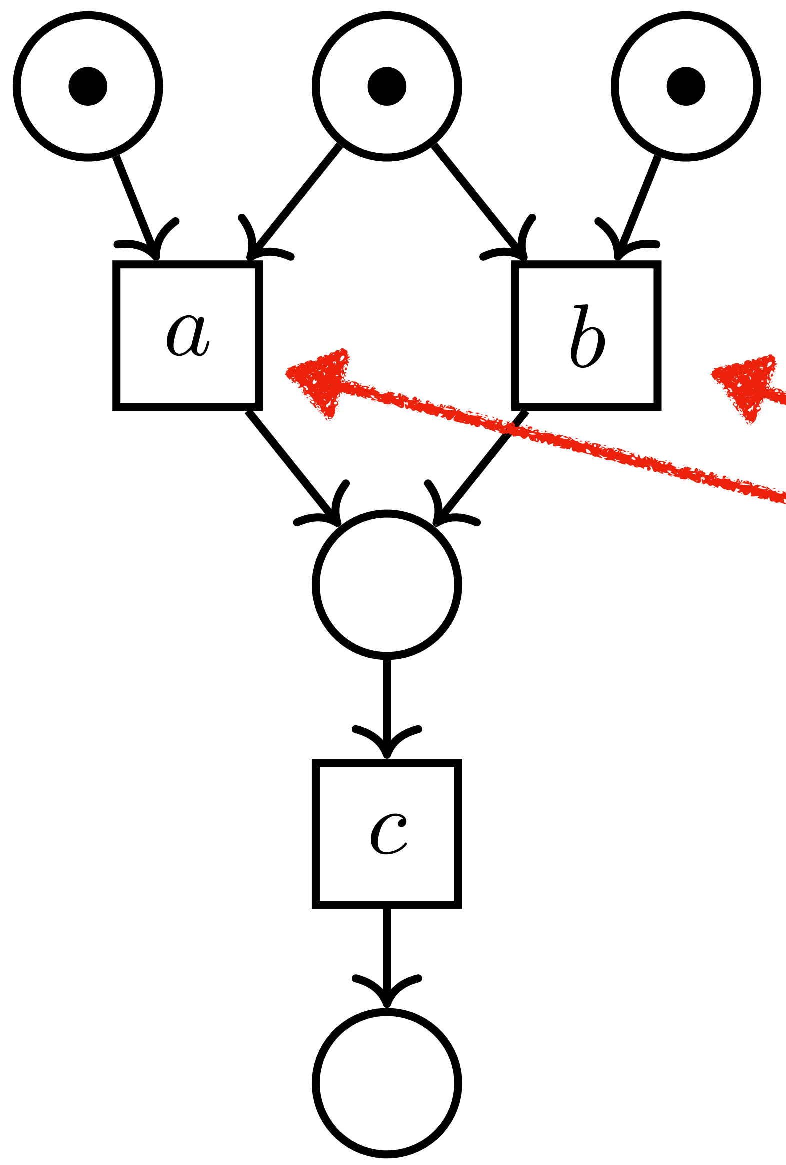
Method

- Starting from the observation that the encoding of a CCS process into a net bears all the needed information
- We modify the encoding of **finite** CCS processes into unravel nets [Boudol,Castellani94]
- We show how unravel nets can be made causal-consistent reversible

Unravel nets (in a nutshel)

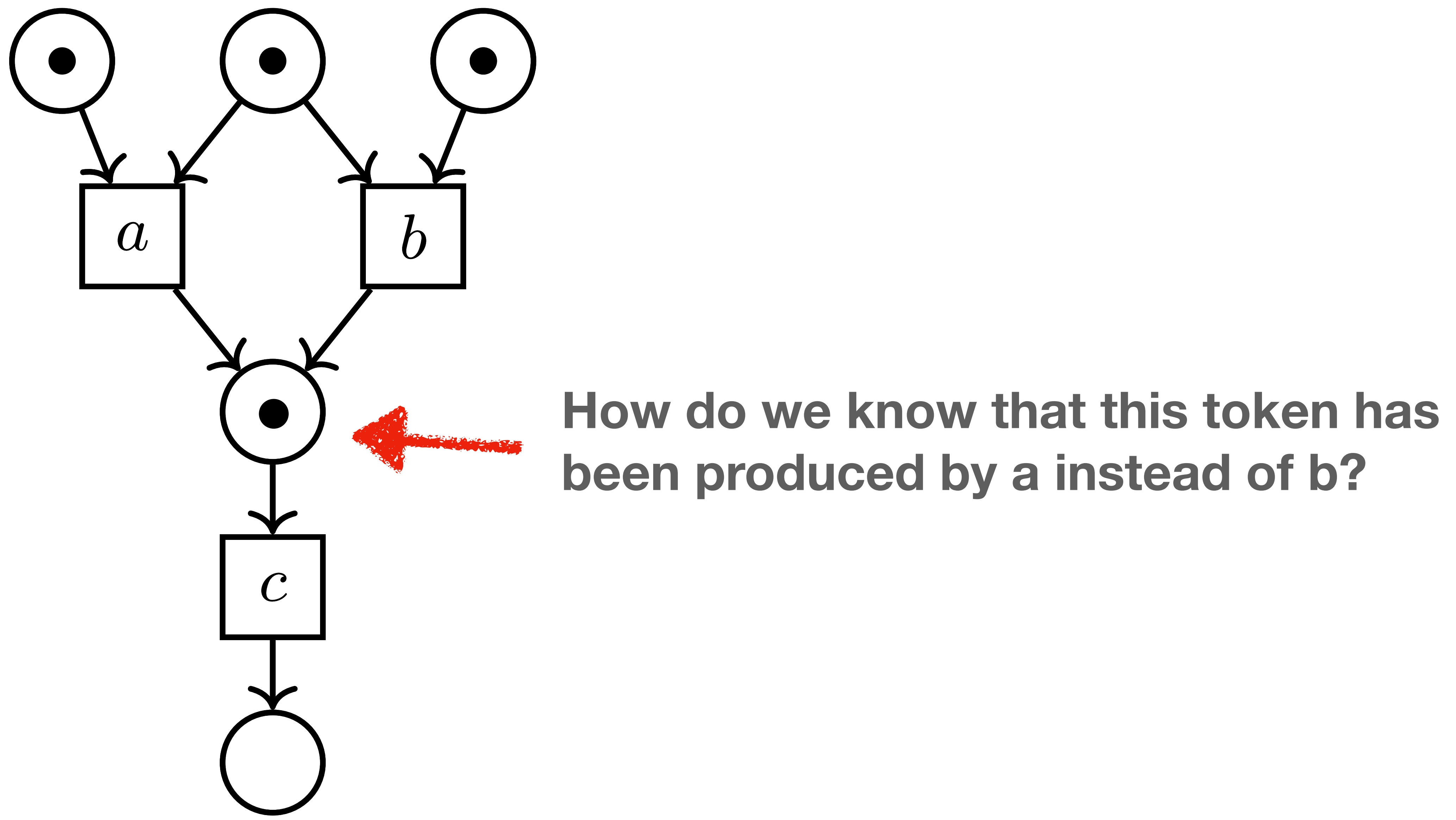
- Unravel net are (1)-safe nets
  - Each place can hold at most one token per time
- If a place has two incoming transition then these transitions are in conflict

Unravel net: example



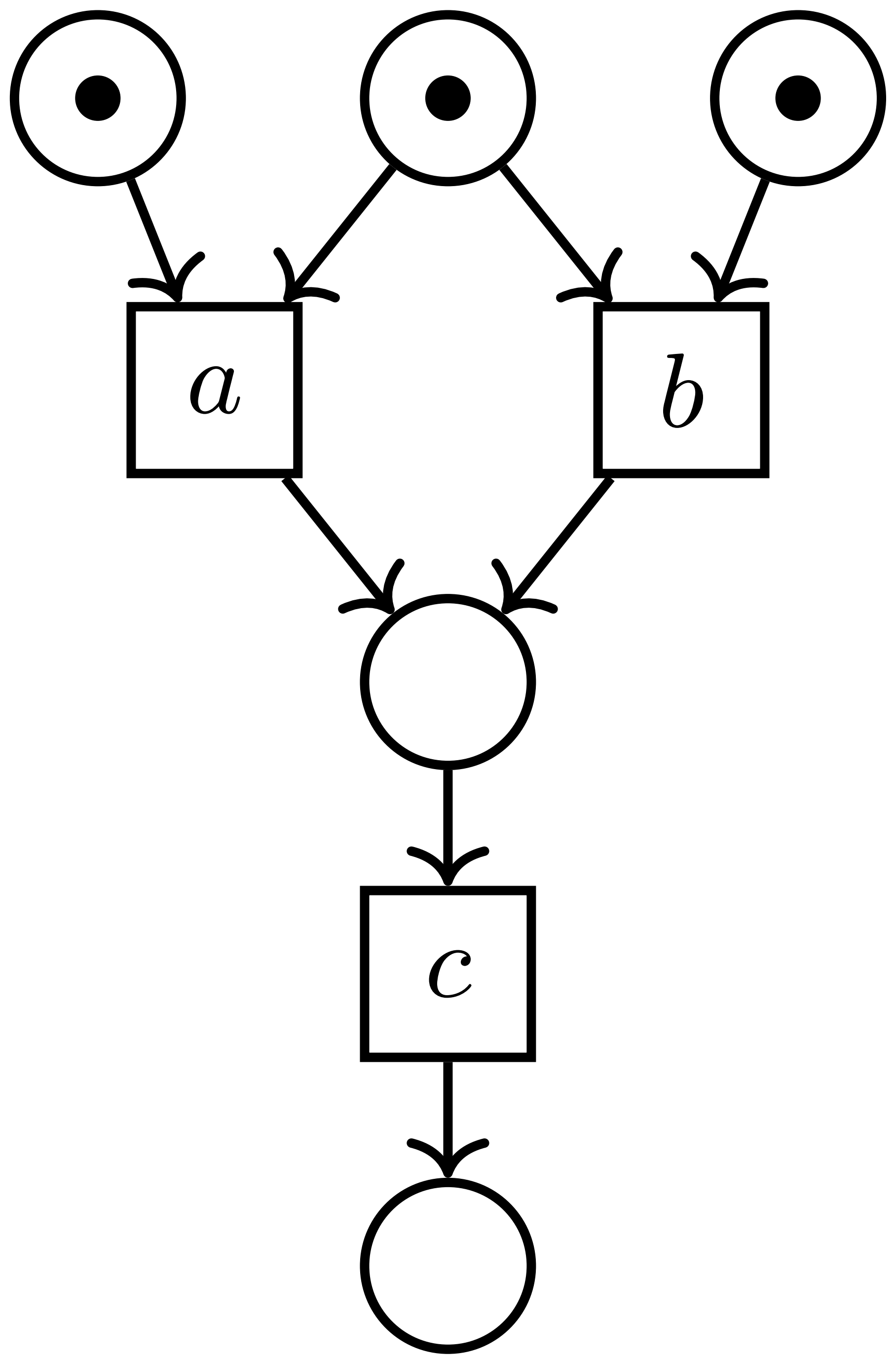
a and b are in conflict

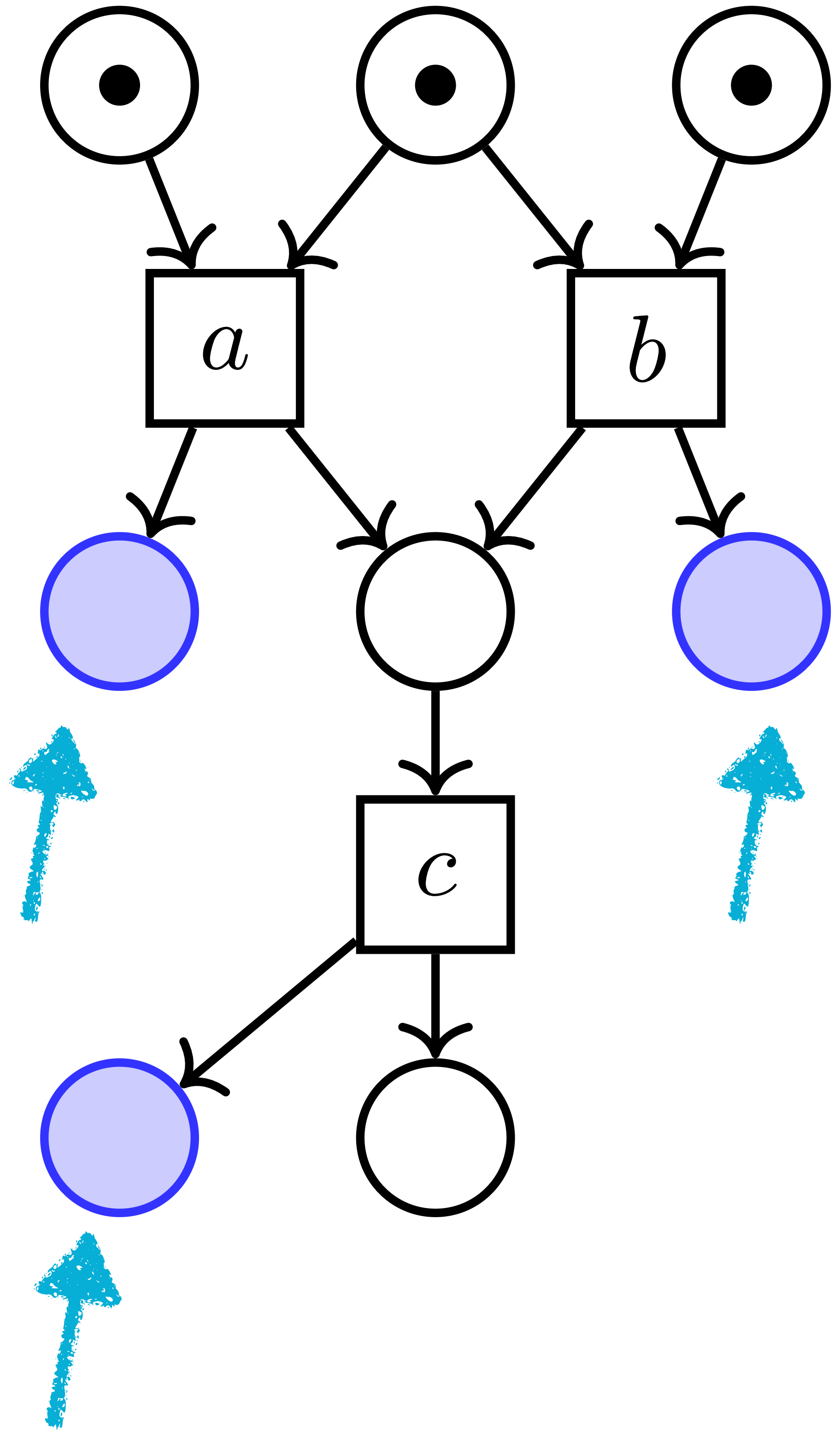
Unravel net: how to reverse?



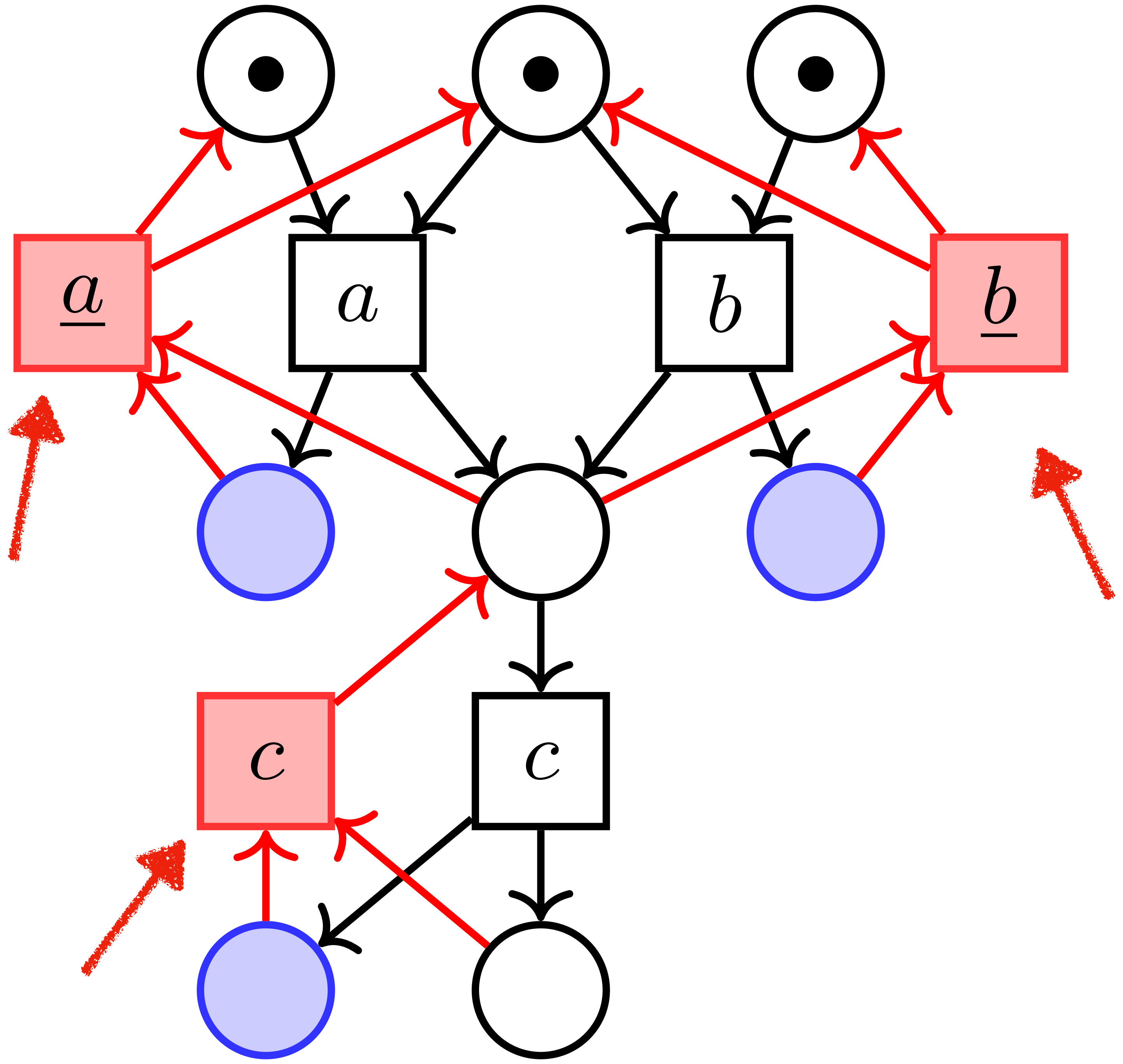
- The idea is then to add extra-place signalling the execution of a transition
  - We call these unravel net complete UN
- These extra place preserve the original semantics of urnavel nets

Reversible Unravel Net





Extra places to remember computation
(they can be seen as communication keys)



Symmetric reversible transitions

Encoding: nil and prefix

Definition 8. *The net* $\mathcal{N}(\mathbf{0}) = \langle \{\mathbf{0}\}, \emptyset, \emptyset, \{\mathbf{0}\} \rangle$ *is the net associated to* $\mathbf{0}$ *and it is called* zero.

Definition 9. *Let* P *a CCS process and* $\mathcal{N}(P) = \langle S_P, T_P, F_P, \mathsf{m}_P \rangle$ *be the associated net. Then* $\mathcal{N}(\alpha.P)$ *is the net* $\langle S_{\alpha.P}, T_{\alpha.P}, F_{\alpha.P}, \mathsf{m}_{\alpha.P} \rangle$ *where*

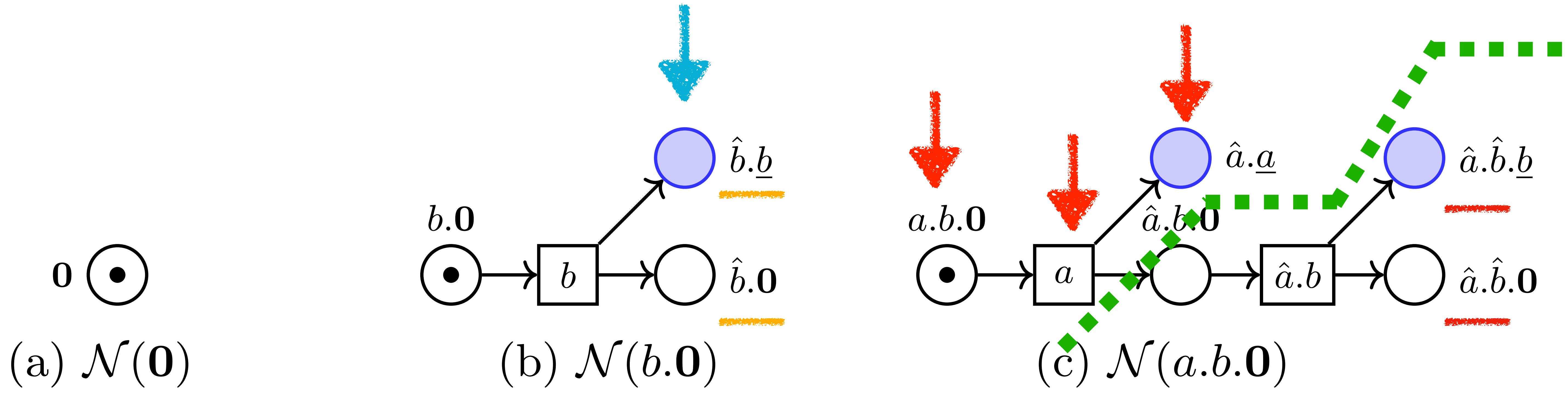
$$
\begin{aligned}
S_{\alpha.P} &= \{\alpha.P, \hat{\alpha}.\underline{\alpha}\} \cup \boxed{\hat{\alpha}}.S_P \\
T_{\alpha.P} &= \{\alpha\} \cup \boxed{\hat{\alpha}}.T_p \\
F_{\alpha.P} &= \{(\alpha.P, \alpha), (\alpha, \hat{\alpha}.\underline{\alpha})\} \cup \{(\alpha, \hat{\alpha}.b) \mid b \in \mathsf{m}_{0P}\} \cup \hat{\alpha}.F_P \\
\mathsf{m}_{\alpha.P} &= \{\alpha.P\}
\end{aligned}
$$

Two places per prefix
One transition per prefix

Decorates all the places/transitions with $\hat{\alpha}$ indicating that α is their past is

Extra place to remember α
e.g. similar to a communication key

Encoding: nil and prefix examples



Net corresponding to b.0

Parts corresponding to prefix a

Key place to remember b has been done

^ is used to indicate a past transition

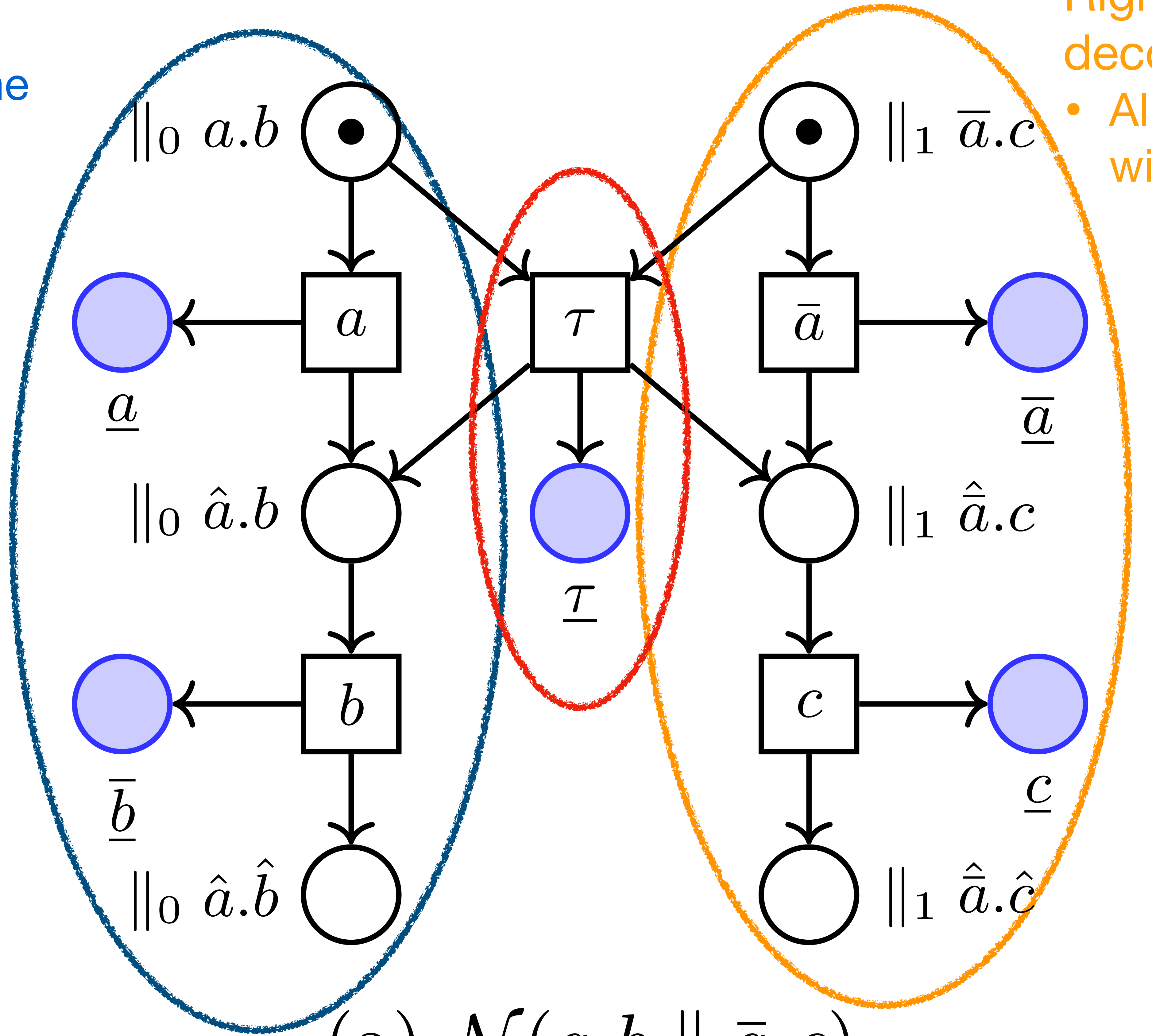
Encoding: parallel example

Left part of the parallel is the process decorated with ||0 to indicate it is the left one

- All the transitions/places are decorated with ||0

Right part of the parallel is the process decorated with ||1 to indicate it is the right one

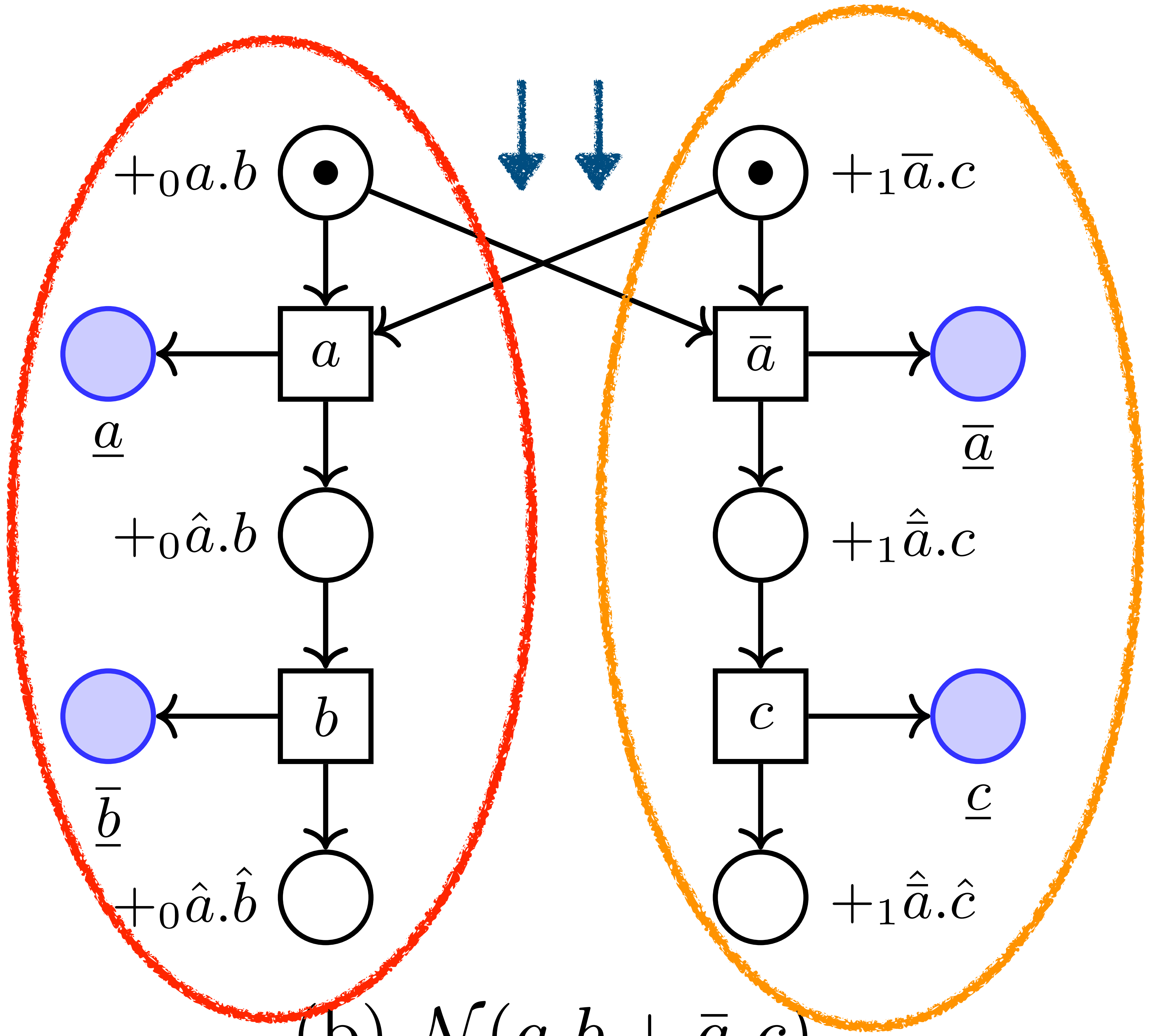
- All the transitions/places are decorated with ||1



(a) $\mathcal{N}(a.b \parallel \bar{a}.c)$

We add all the possible synchronisations/tau moves

Encoding: choice example



(b) $\mathcal{N}(a.b + \overline{a}.c)$

Similarly to the encoding of || we distinguish left +0 part of the process from the right one +1

Mutual exclusion/conflict of initial transitions

Reversible Unravel nets

Proposition 1. *Let* $N = \langle S, T, F, \mathsf{m}\rangle$ *be a complete unravel net and* $U \subseteq T$ *the set of transitions to be reversed. Define* $\overleftarrow{N}^U = \langle S', T', U', F', \mathsf{m}'\rangle$ *where* $S = S'$, $U' = U \times \{\mathtt{r}\}$, $T' = (T \times \{\mathtt{f}\}) \cup U'$,

$$\begin{aligned} F' = &\{(s,(t,\mathtt{f})) \mid (s,t) \in F\} \cup \{((t,\mathtt{f}),s) \mid (t,s) \in F\} \cup \\ &\underline{\{(s,(t,\mathtt{r})) \mid (t,s) \in F\} \cup \{((t,\mathtt{r}),s) \mid (s,t) \in F\}} \end{aligned}$$

For each forward transition we add an exact inverse one

Results

Ancestor of R

Marking derived from the memory of R

Definition 14. *Let R be an RCCS term with $\rho(R) = P$. Then $\overleftarrow{\mathcal{N}(R)}$ is the net $\langle S, T, F, \mu(R)\rangle$ where $\mathcal{N}(P) = \langle S, T, F, \mathsf{m}\rangle$.*

Proposition 3. *Let R be an RCCS term with $\rho(R) = P$. Then $\overleftarrow{\mathcal{N}(R)}$ is a reversible unravel net.*

What changes from N(R) and N(P) is the marking

Theorem 1. *Let P be a finite CCS process, then $\langle\rangle \triangleright P \sim \overleftarrow{\mathcal{N}(P)}$.*

Proof sketch. It is sufficient to show that

$$\mathcal{R} = \{(R, \langle S, T, F, \mu(R)\rangle) \mid \rho(R) = P,\ \overleftarrow{\mathcal{N}(P)} = \langle S, T, F, m\rangle\}$$

is a bisimulation.

Reversibility in blockchains?

Revert in Solidity

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Insufficient balance for transfer. Needed `required` but only
/// `available` available.
/// @param available balance available.
/// @param required requested amount to transfer.
error InsufficientBalance(uint256 available, uint256 required);

contract TestToken {
    mapping(address => uint) balance;
    function transfer(address to, uint256 amount) public {
        if (amount > balance[msg.sender])
            revert InsufficientBalance({
                available: balance[msg.sender],
                required: amount
            });
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
    // ...
}
```

Revert and Ethereum

From stack overflow

“Revert op code” means any EVM (that is the virtual machine that execute your code or the code of the applications you use on the Ethereum network) instruction that give to it the command to erase and nullify the last elaborations, made by the current task, “reverting” the blockchain status to that before your code run.

That is if you make some operations on the blockchain (mint, transfer, read, write, etc) but a revert op code is encountered, all is erased and the blockchain remain that it was before you tried to change (by your code).