

Deadlock-Free Parallel Regions for Projected Workflows

Benedikt Bollig



Laboratoire
Méthodes
Formelles



école
normale
supérieure
paris—saclay

université
PARIS-SACLAY

EXPRESS/SOS 2026
September 5, 2026
Liverpool, UK



PROGRAMME
DE RECHERCHE
INTELLIGENCE
ARTIFICIELLE

Motivation

- A language to specify **multi-agent workflows**

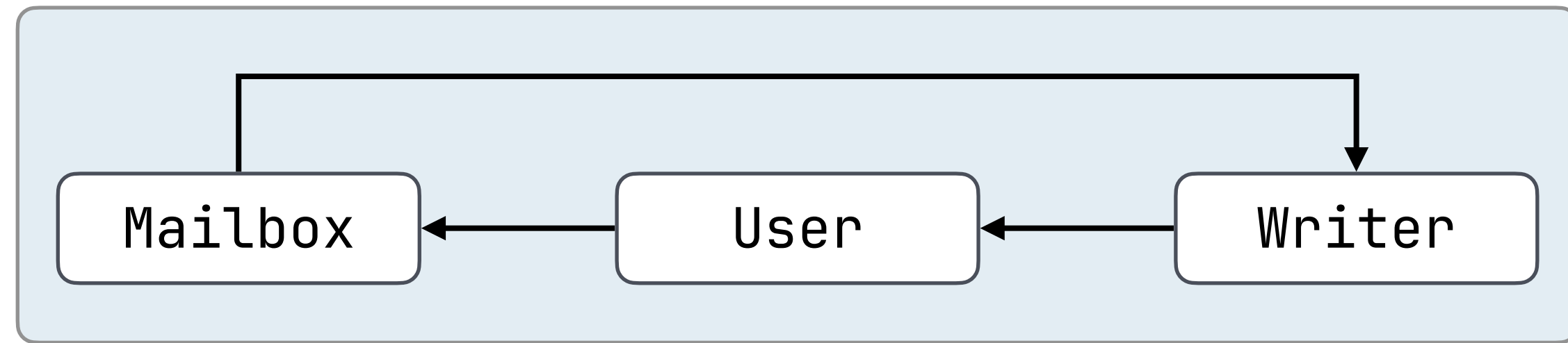
Motivation

- A language to specify **multi-agent workflows**
- Designed for large language models (LLMs) as both:
 - ▶ workflow participants (email writer, meeting scheduler)
 - ▶ workflow programmers (coding agent)

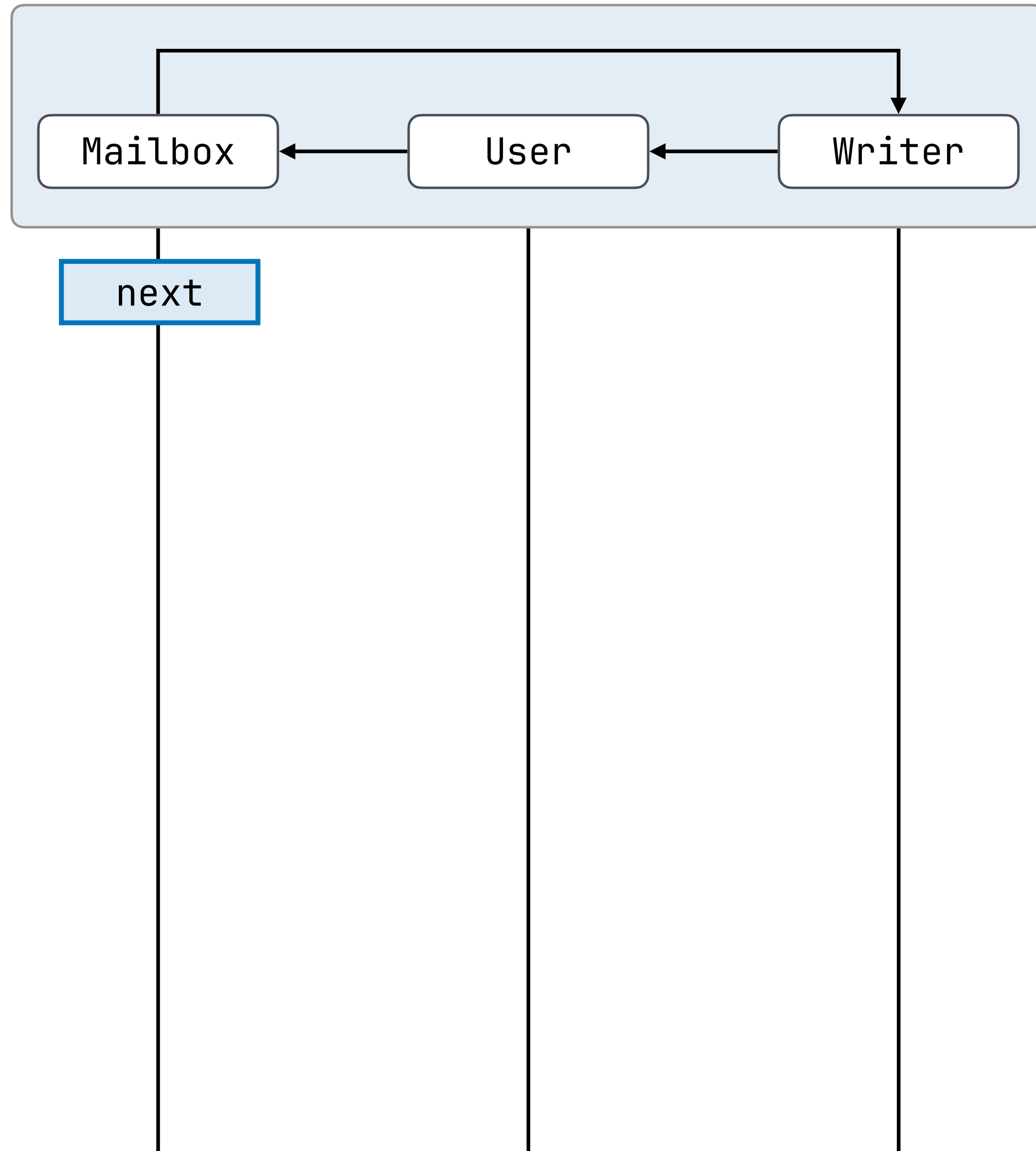
Motivation

- A language to specify **multi-agent workflows**
- Designed for large language models (LLMs) as both:
 - ▶ workflow participants (email writer, meeting scheduler)
 - ▶ workflow programmers (coding agent)
- Goal: add parallelism without losing deadlock-freedom

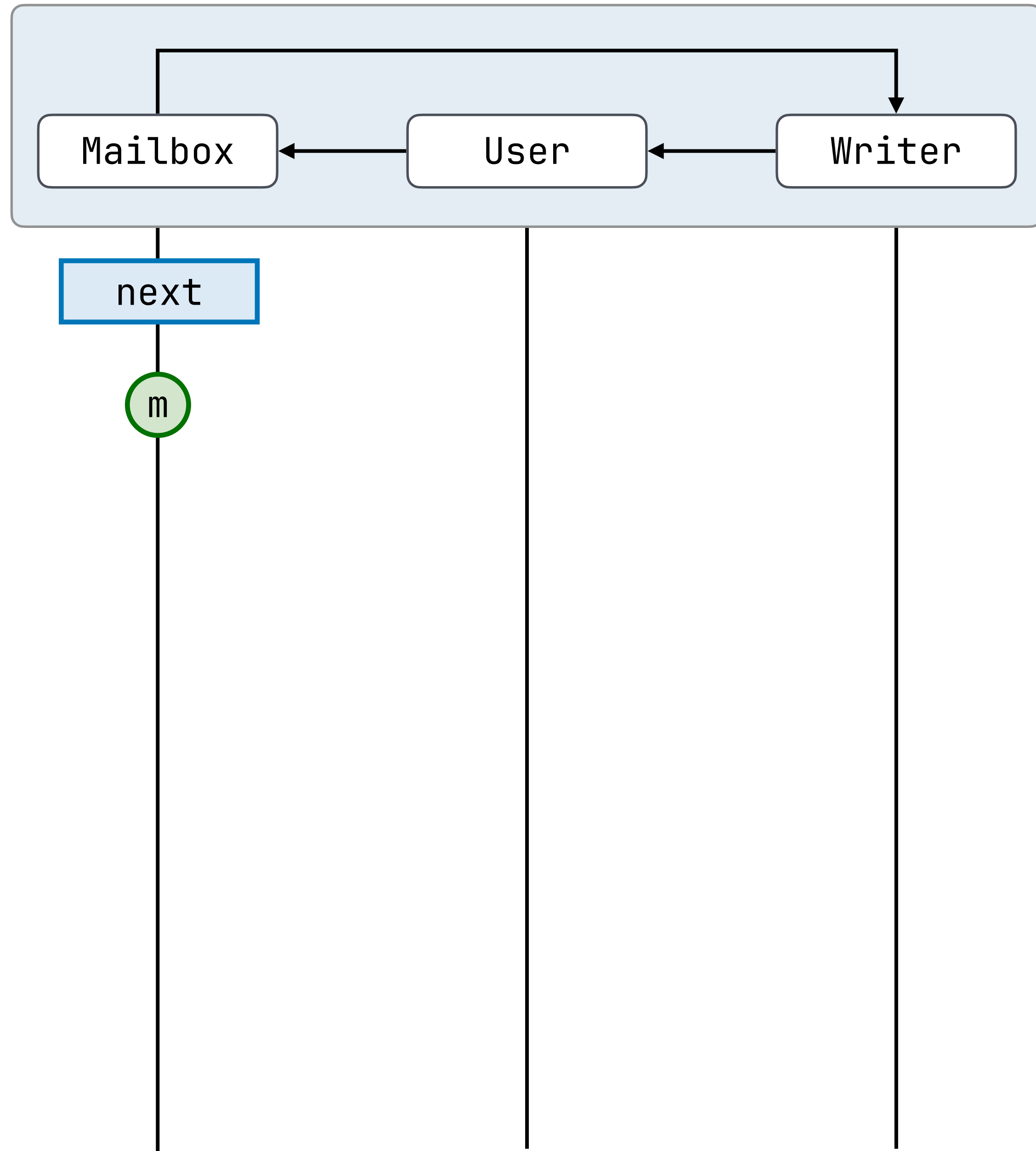
Multi-agent systems as distributed systems



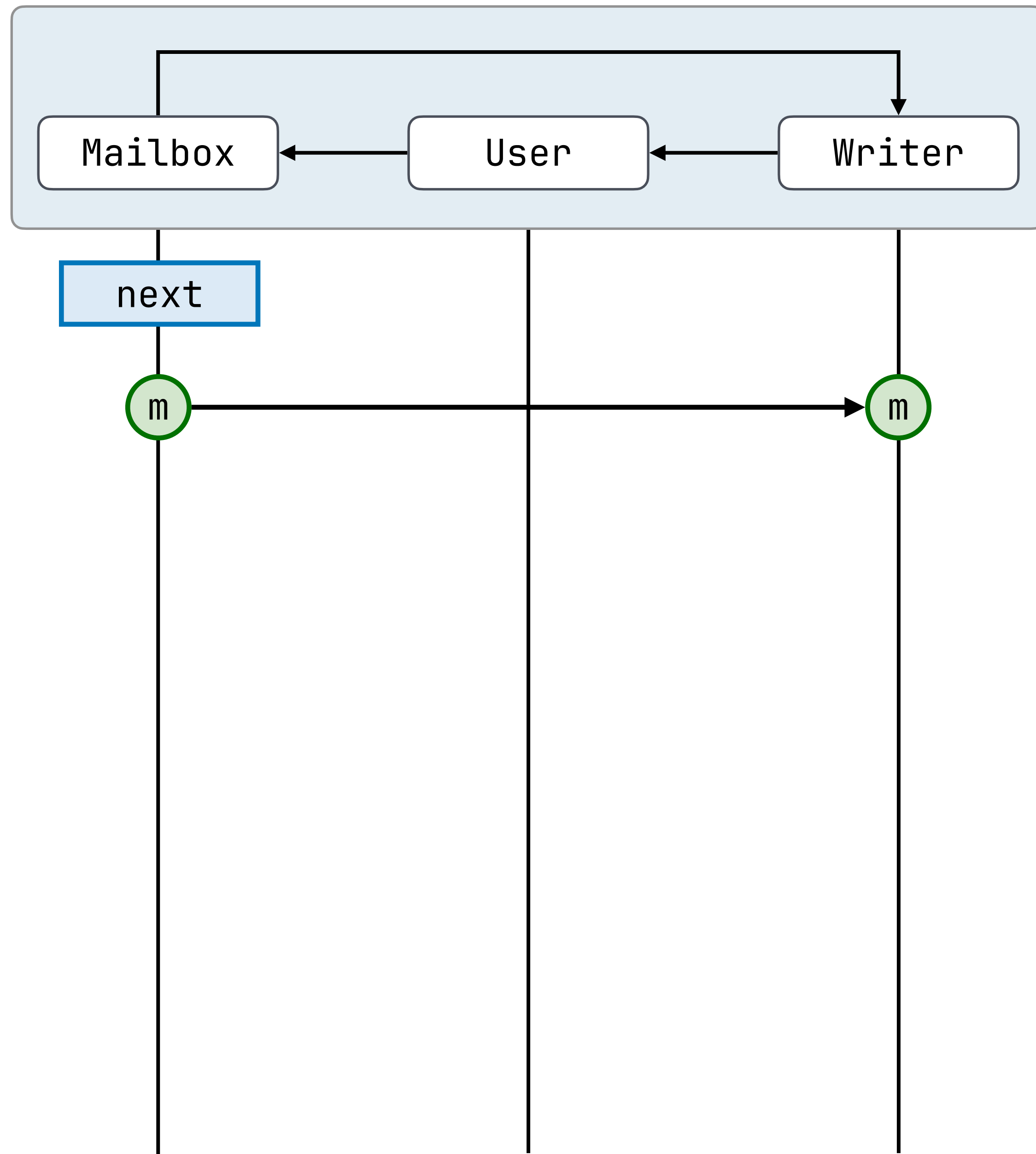
Multi-agent systems as distributed systems



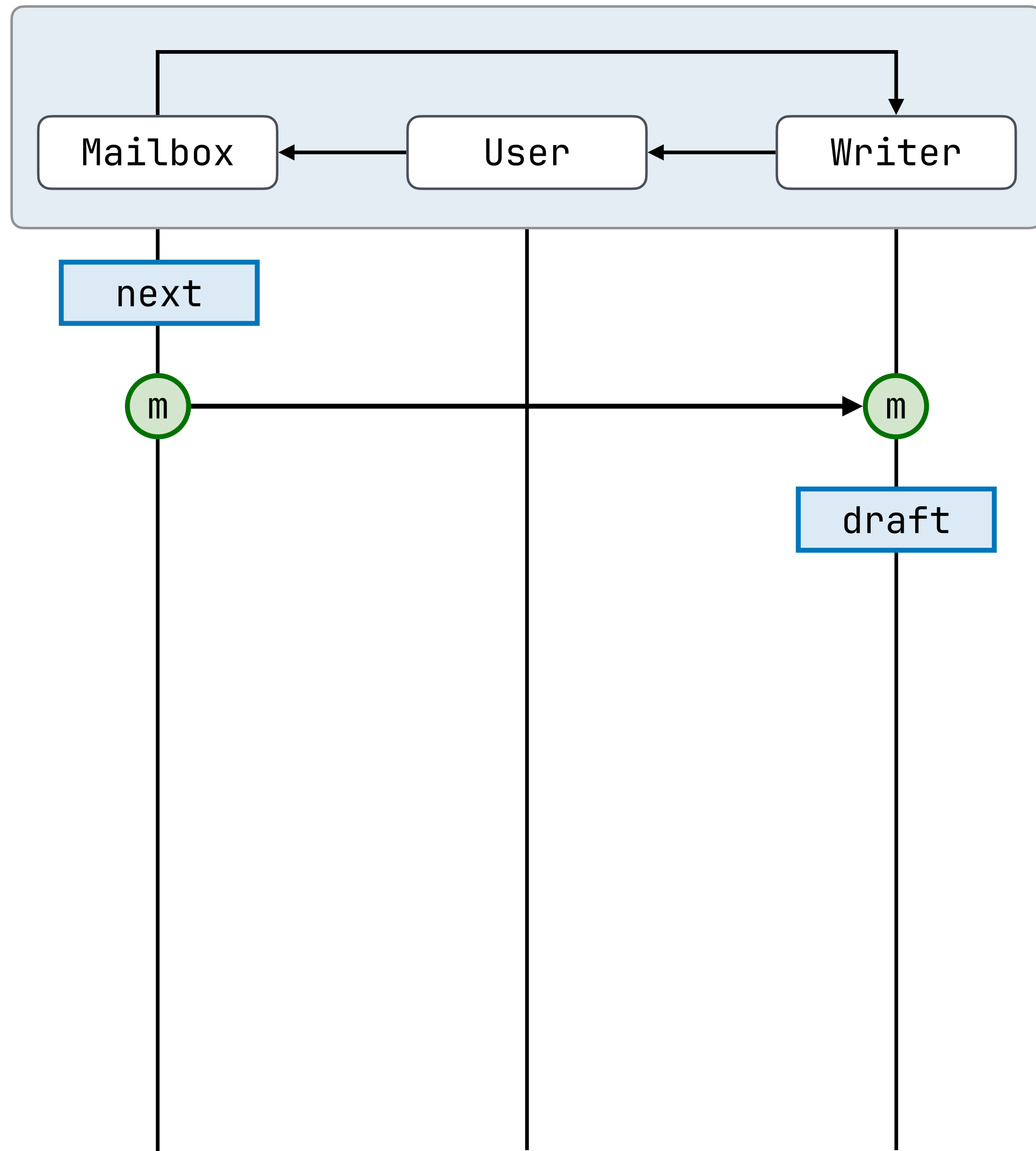
Multi-agent systems as distributed systems



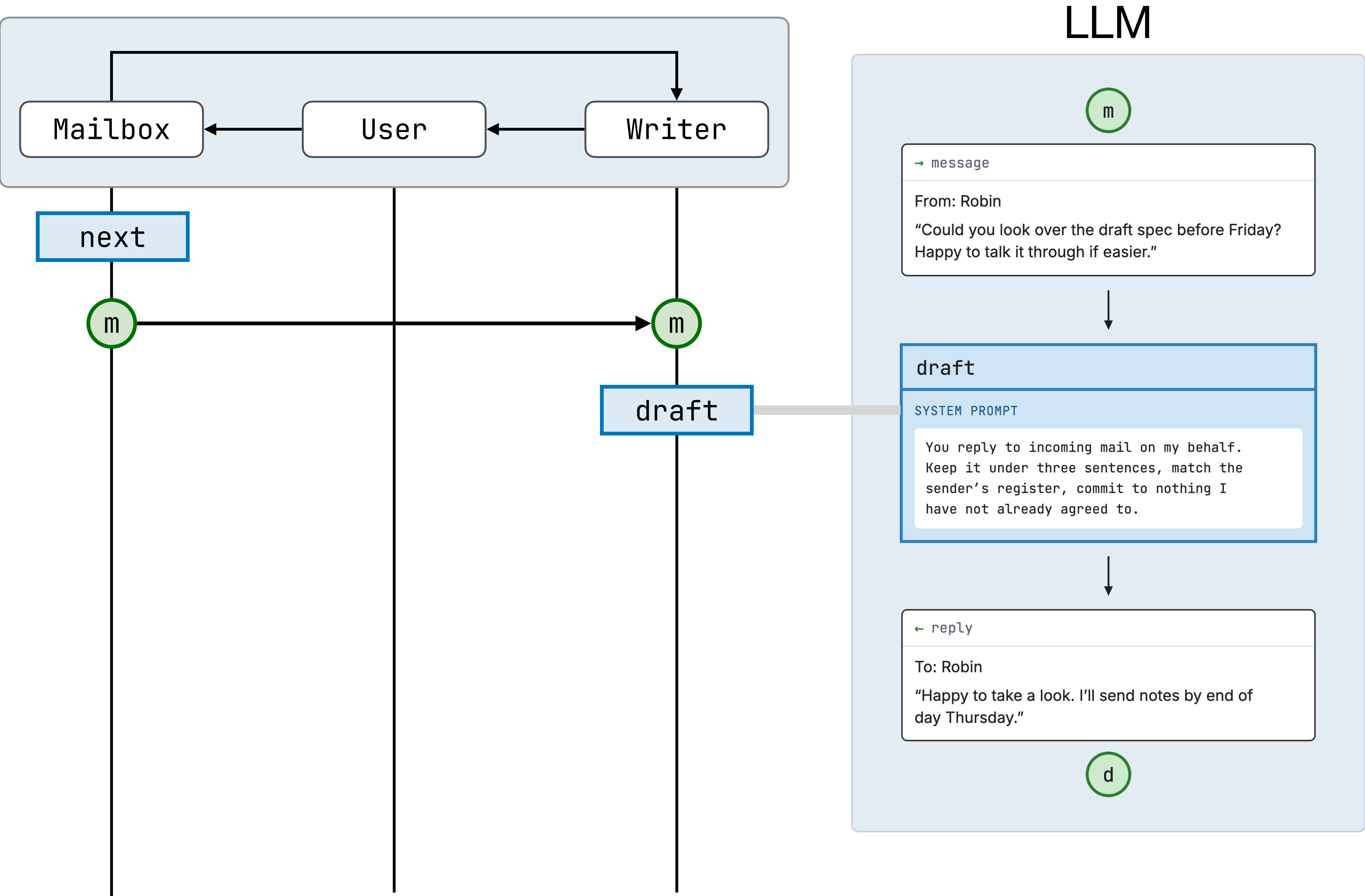
Multi-agent systems as distributed systems



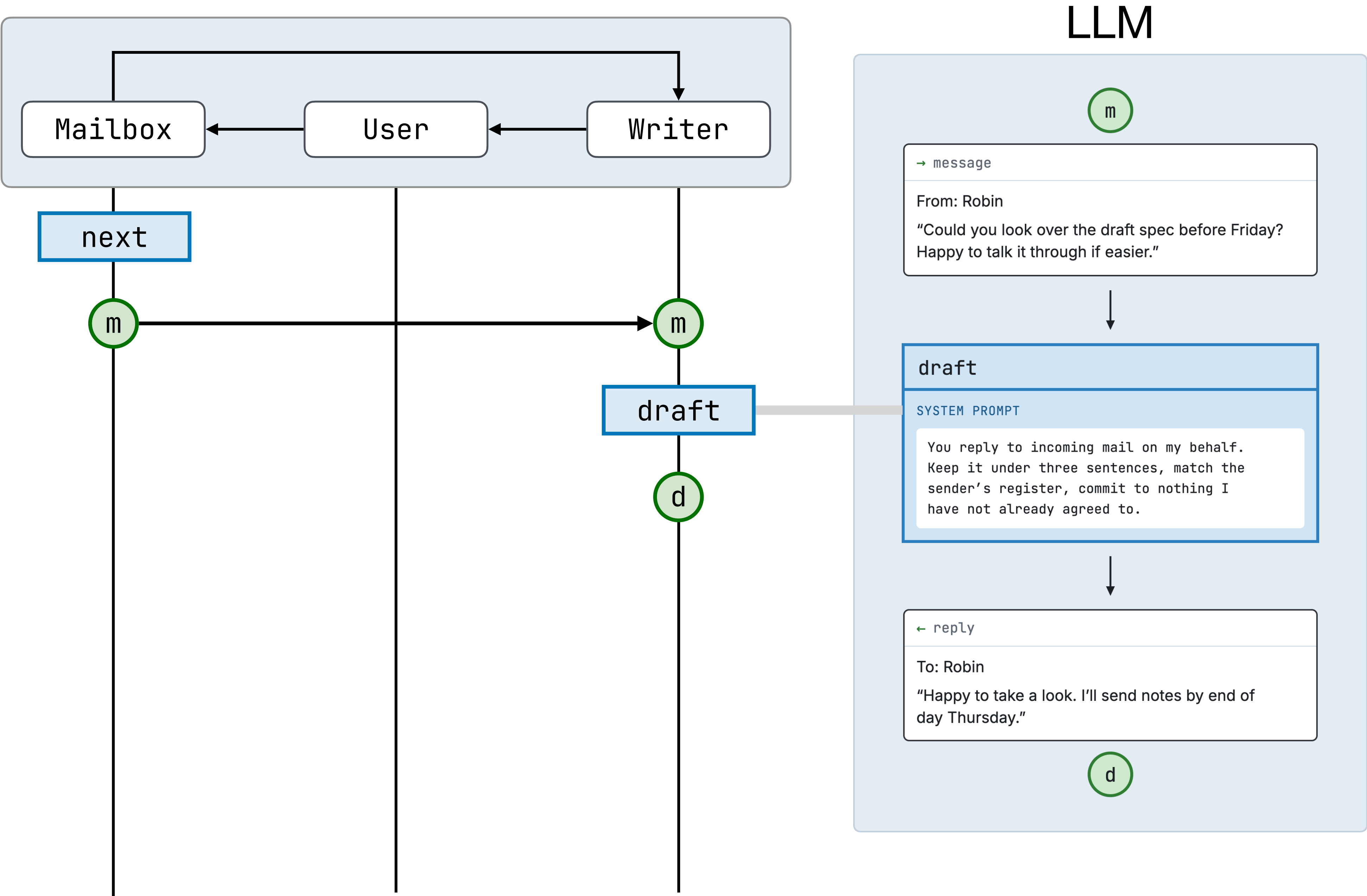
Multi-agent systems as distributed systems



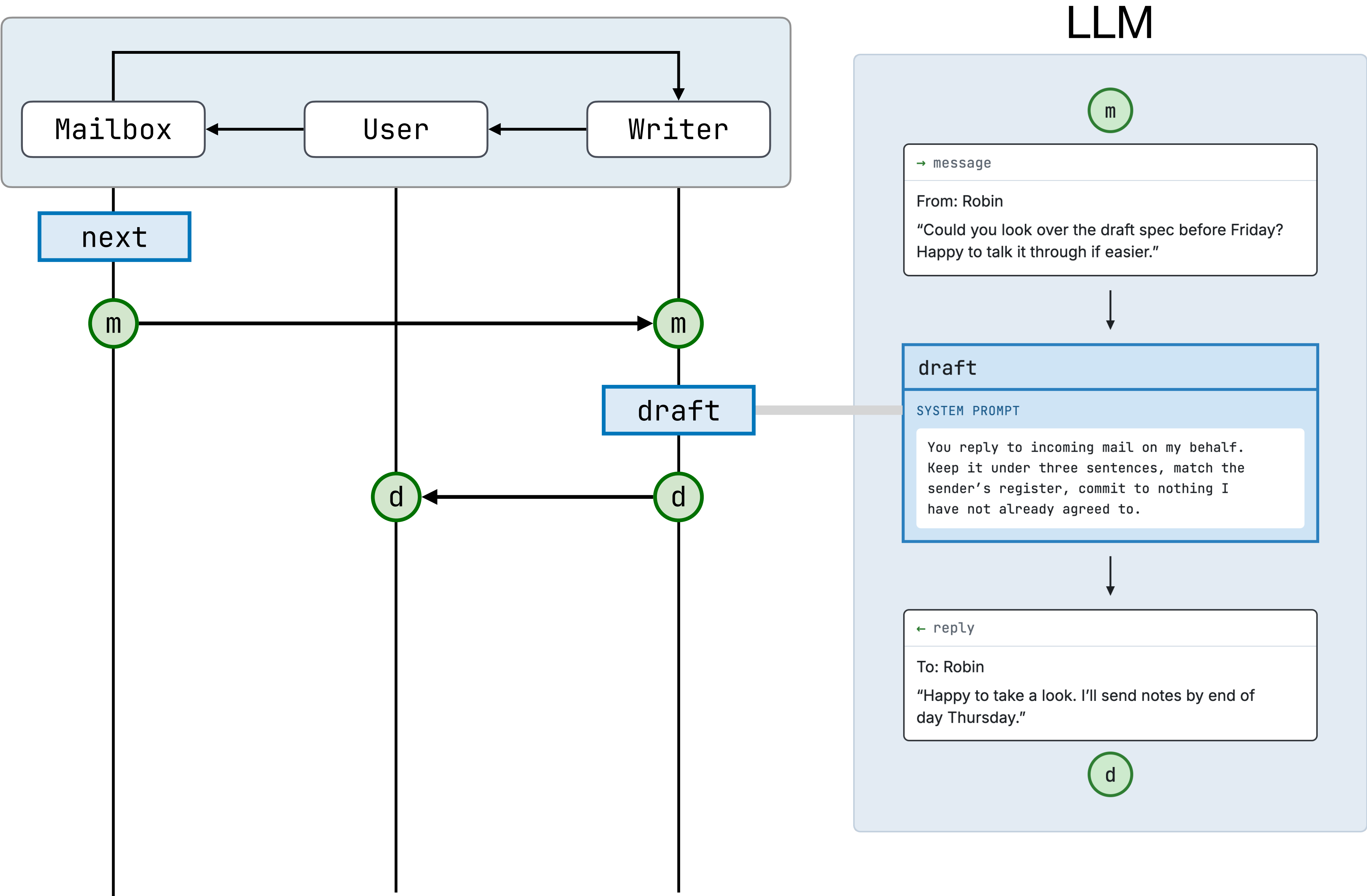
Multi-agent systems as distributed systems



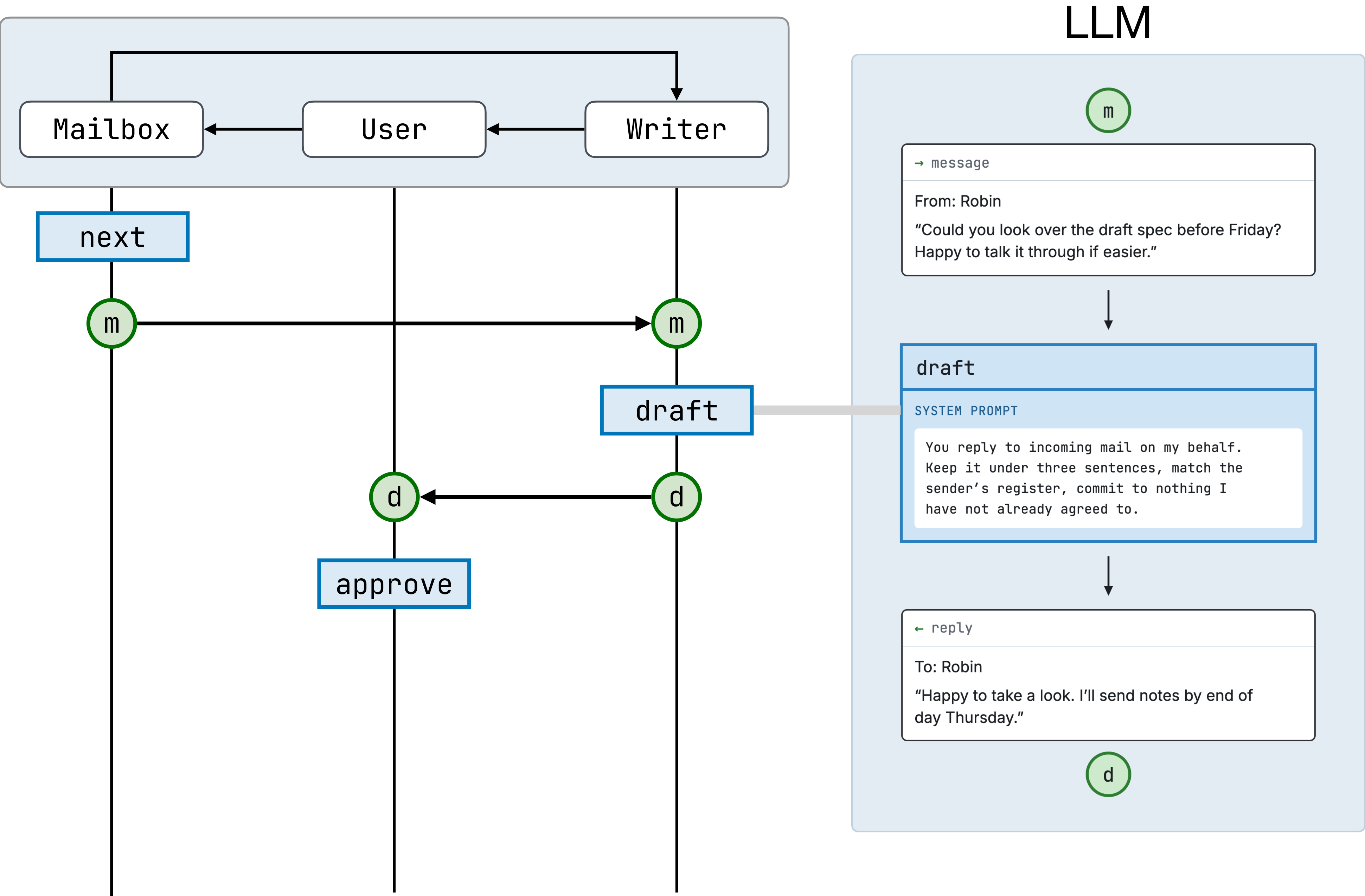
Multi-agent systems as distributed systems



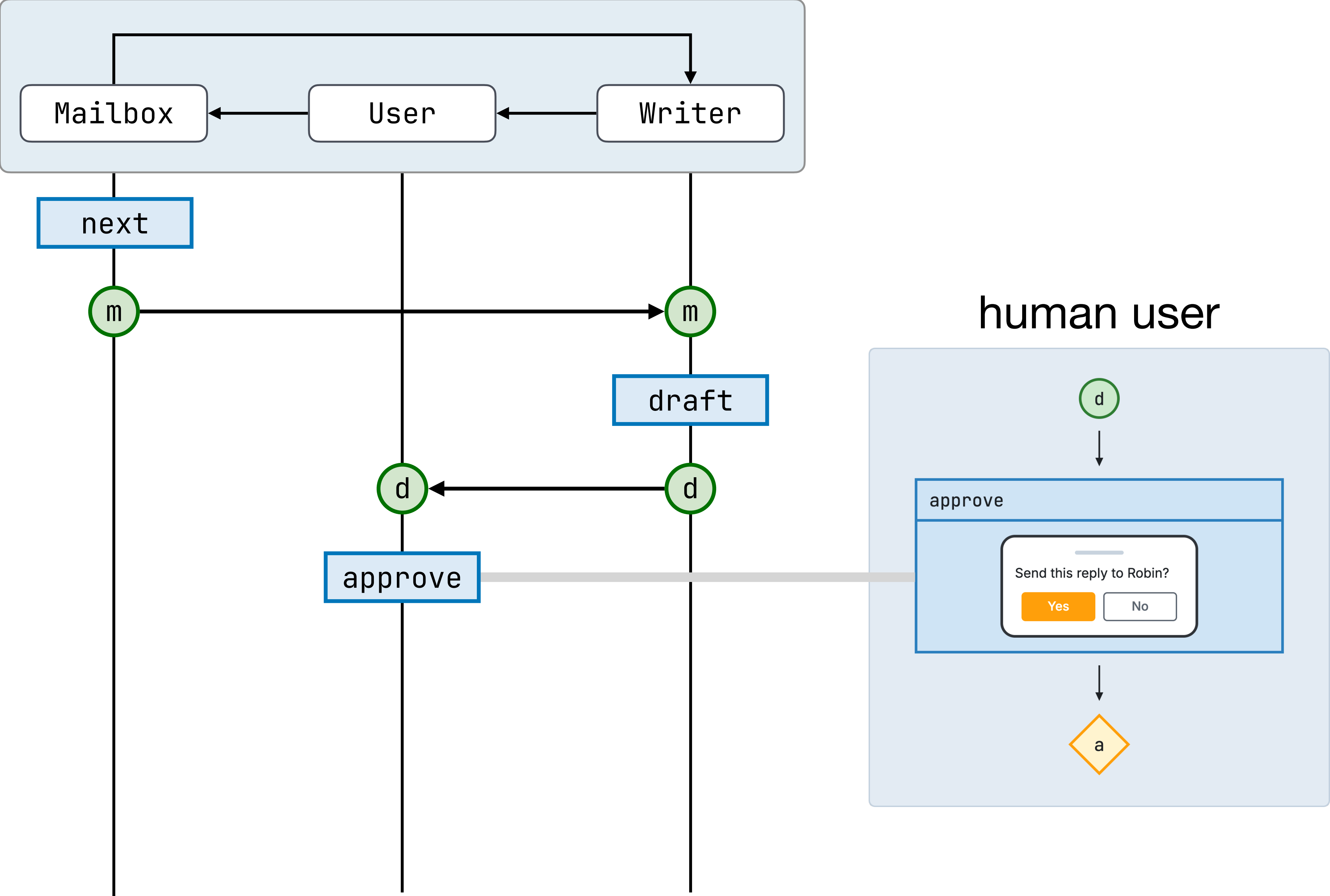
Multi-agent systems as distributed systems



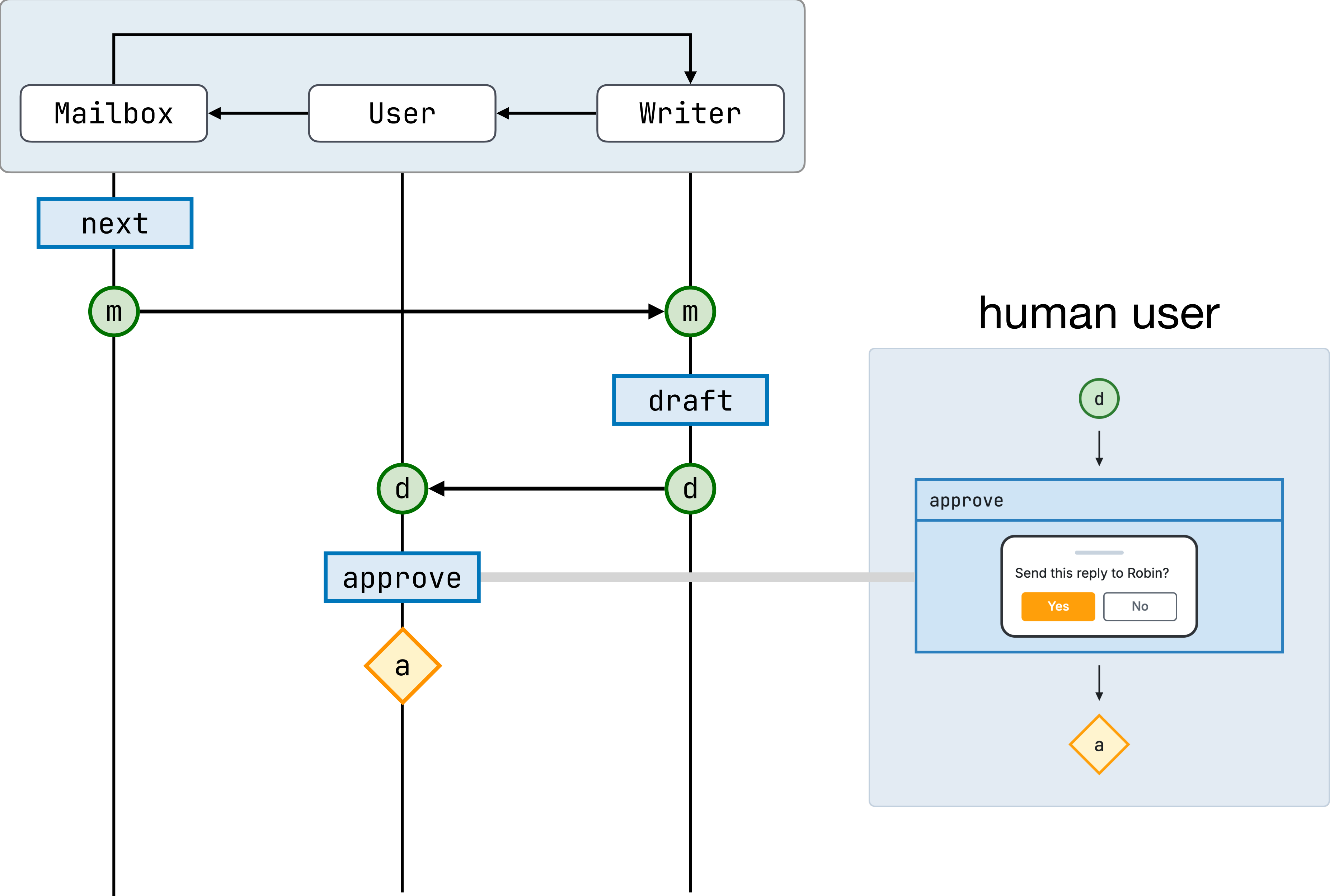
Multi-agent systems as distributed systems



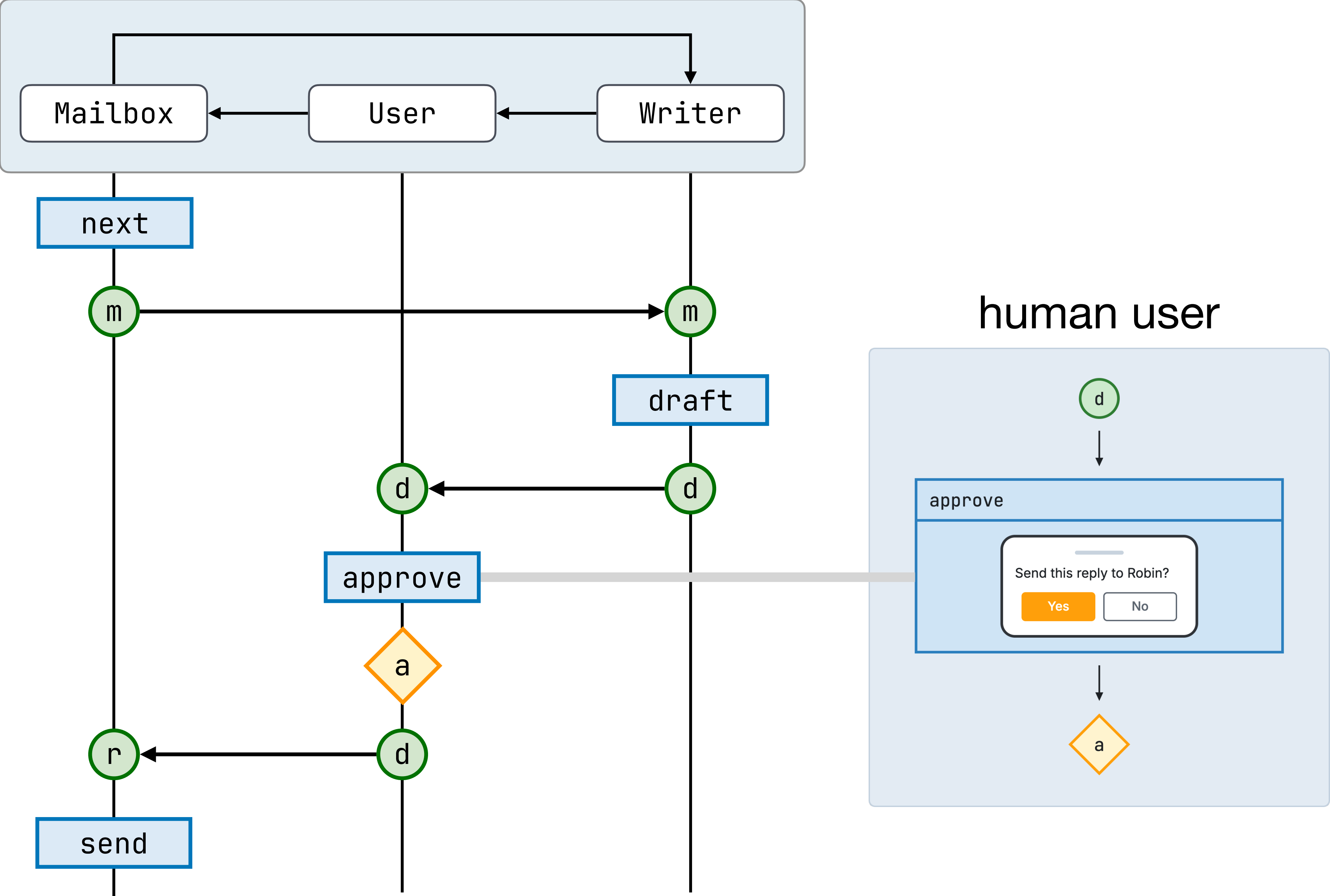
Multi-agent systems as distributed systems



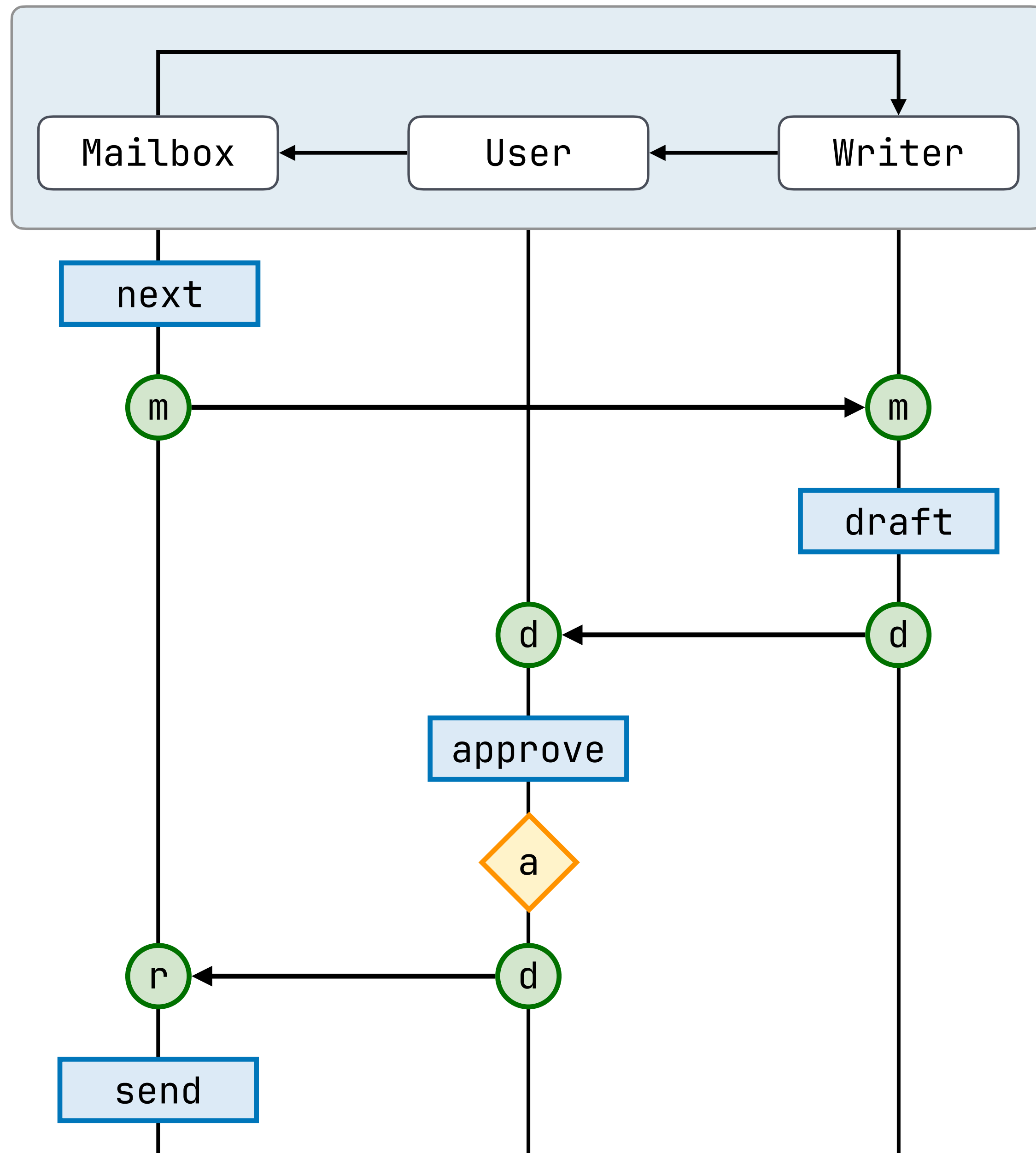
Multi-agent systems as distributed systems



Multi-agent systems as distributed systems



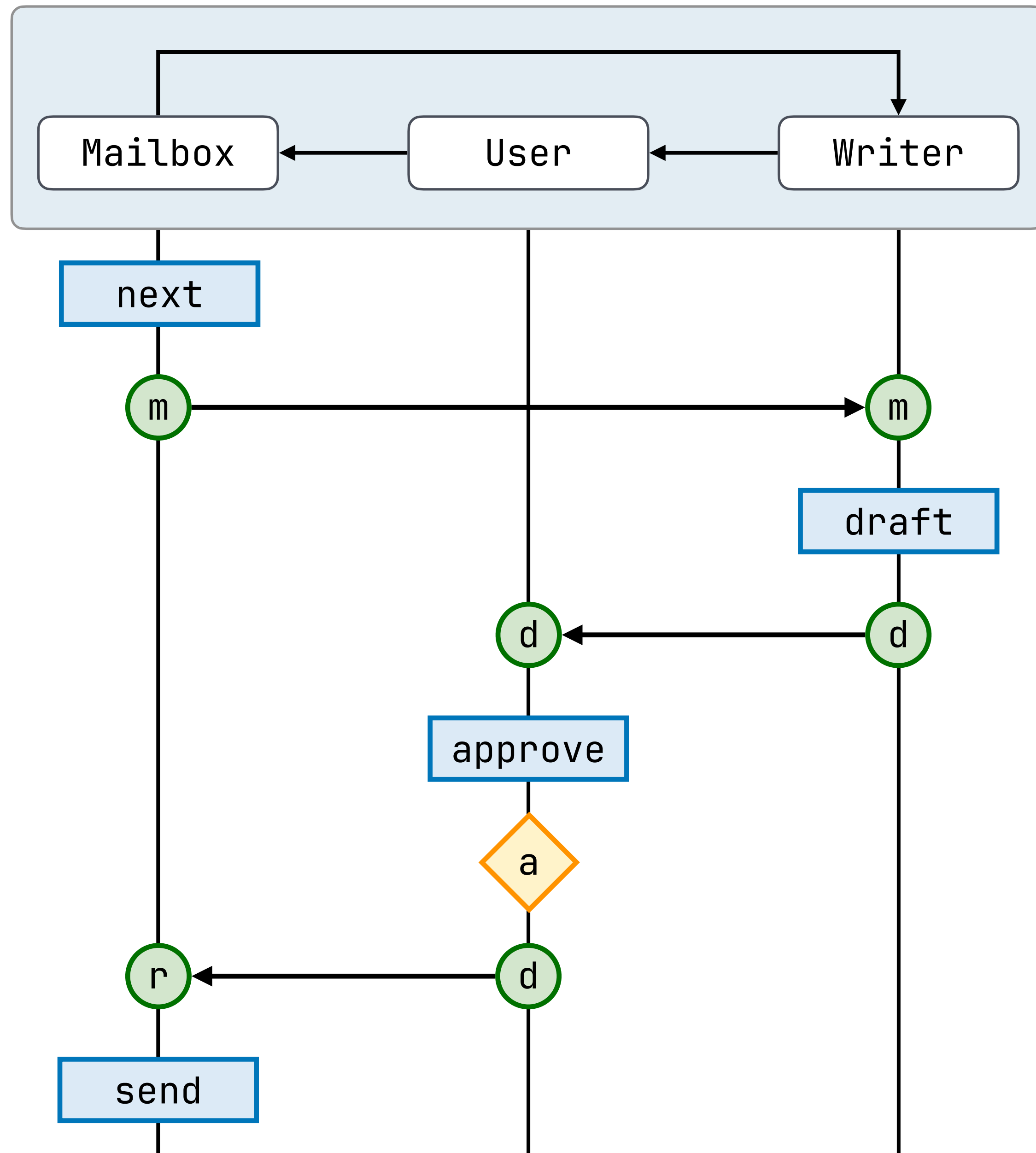
Multi-agent systems as distributed systems



@workflow

```
act Mailbox : m = next()
msg Mailbox(m) → Writer(m)
act Writer : d = draft(m)
msg Writer(d) → User(d)
act User : a = approve(d)
if a @ User then
    msg User(d) → Mailbox(r)
    act Mailbox : status = send(r)
else
    skip
```

Multi-agent systems as distributed systems

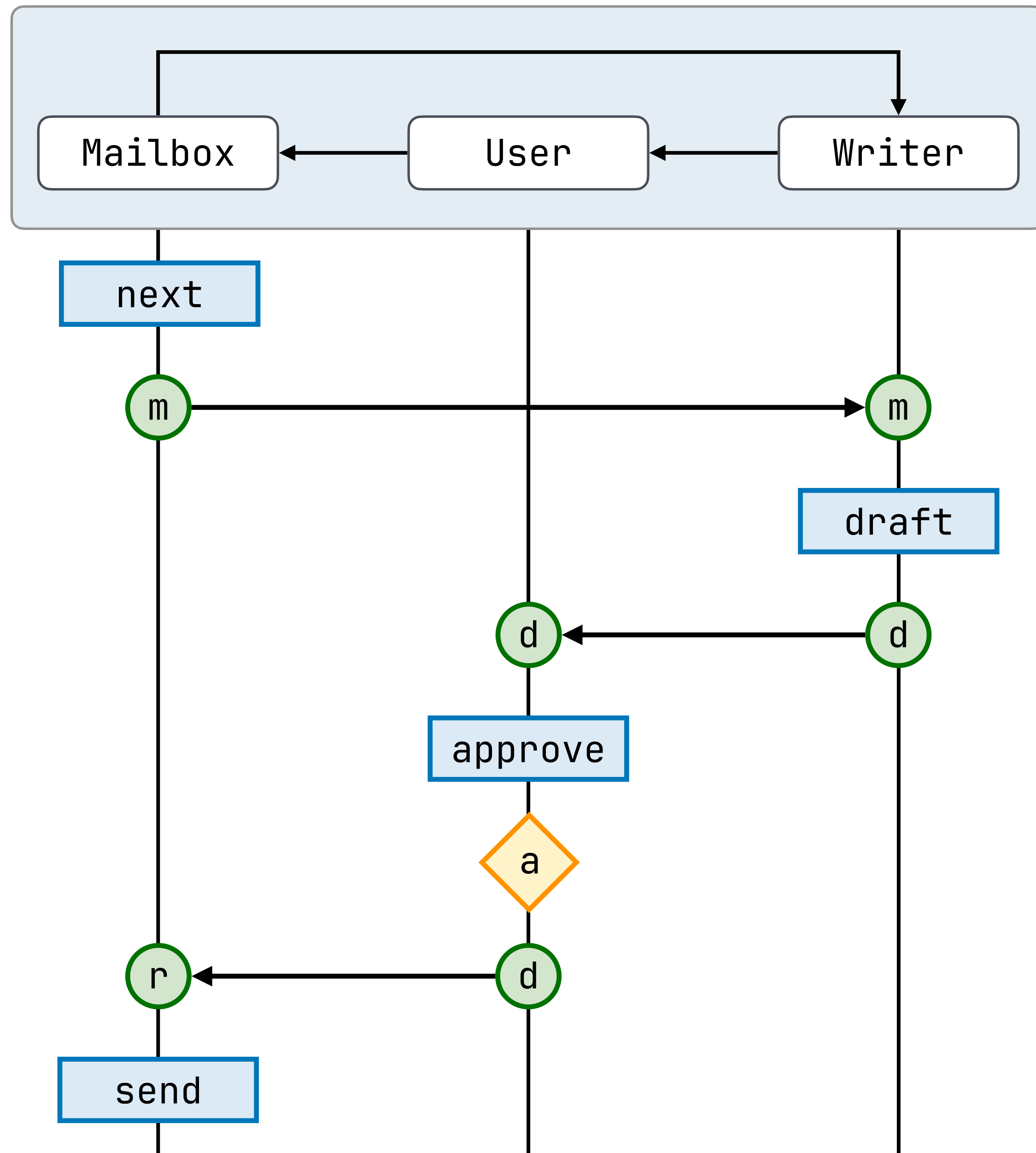


@workflow

```
act Mailbox : m = next()
msg Mailbox(m) → Writer(m)
act Writer : d = draft(m)
msg Writer(d) → User(d)
act User : a = approve(d)
if a @ User then
    msg User(d) → Mailbox(r)
    act Mailbox : status = send(r)
else
    skip
```

- Choreographic programming / global protocols
 - endpoint projection, deadlock-freedom
- [Carbone-Montesi '13; Honda-Yoshida-Carbone '16]

Multi-agent systems as distributed systems



```
@role(User)
```

```
recv User(d) ← Writer
```

```
act User : a = approve(d)
```

```
if a @ User then
```

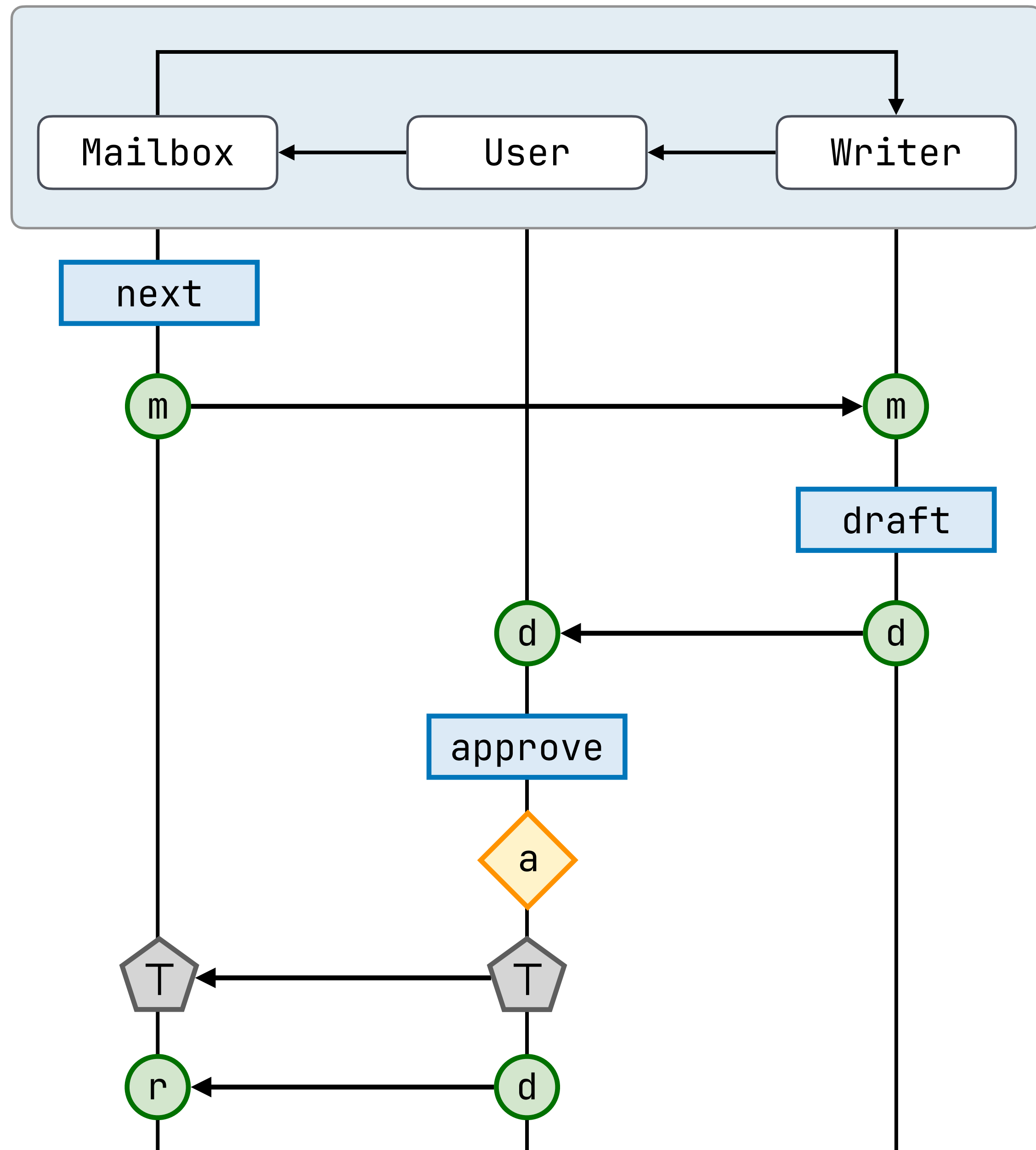
```
    send User(d) → Mailbox
```

```
else
```

```
    skip
```

- Choreographic programming / global protocols
 - endpoint projection, deadlock-freedom
- [Carbone-Montesi '13; Honda-Yoshida-Carbone '16]

Multi-agent systems as distributed systems



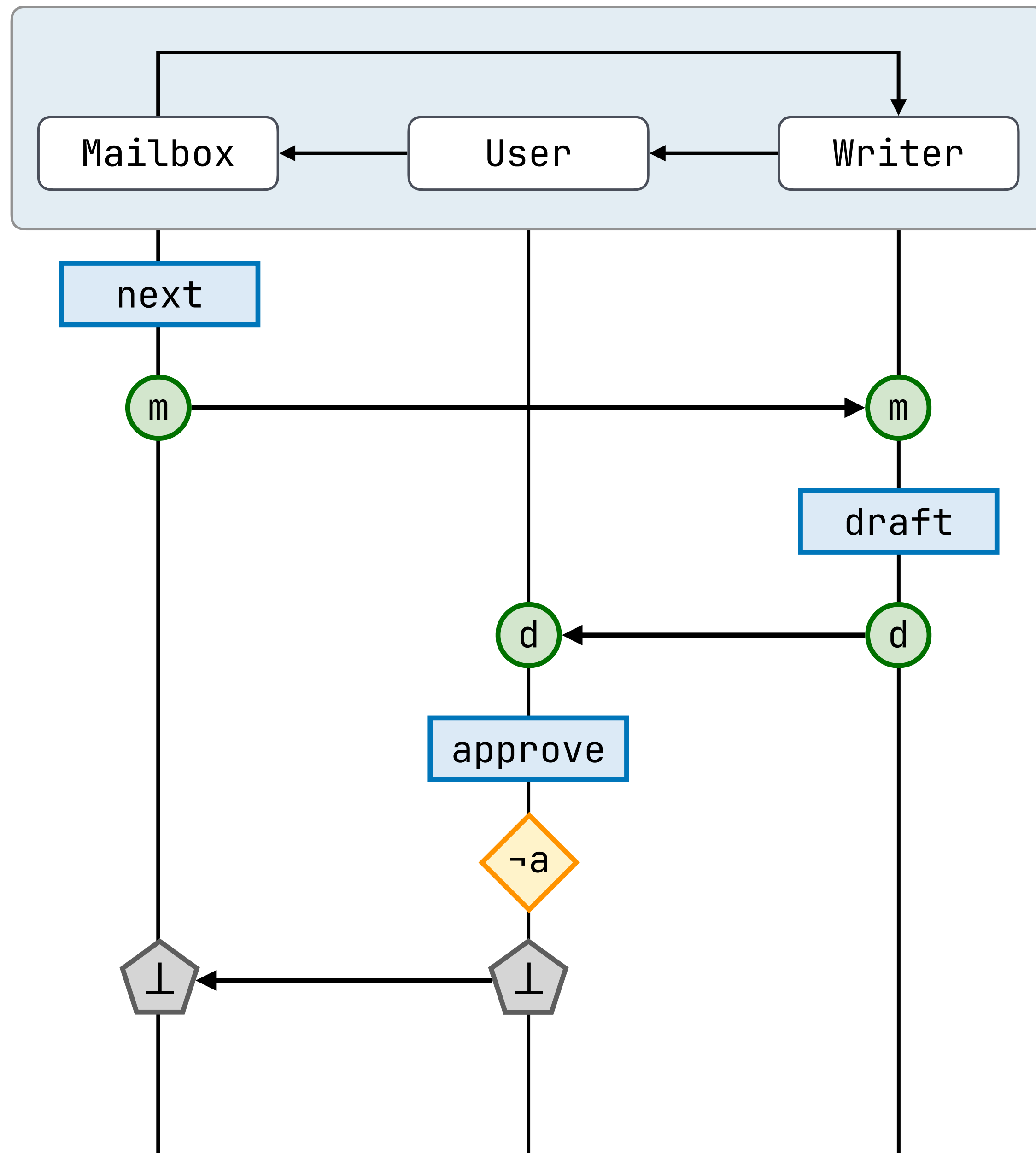
```
@role(User)
```

```
recv User(d) ← Writer
act User : a = approve(d)
if a @ User then
    send User(T) → Mailbox
    send User(d) → Mailbox
else
    send User(⊥) → Mailbox
```

- Choreographic programming / global protocols
 - ▶ endpoint projection, deadlock-freedom

[Carbone-Montesi '13; Honda-Yoshida-Carbone '16]
- This work (building on [Bollig-Függer-Nowak '26]):
 - ▶ realizability enforced by control broadcasts

Multi-agent systems as distributed systems



```
@role(User)
```

```
recv User(d) ← Writer
```

```
act User : a = approve(d)
```

```
if a @ User then
```

```
    send User(τ) → Mailbox
```

```
    send User(d) → Mailbox
```

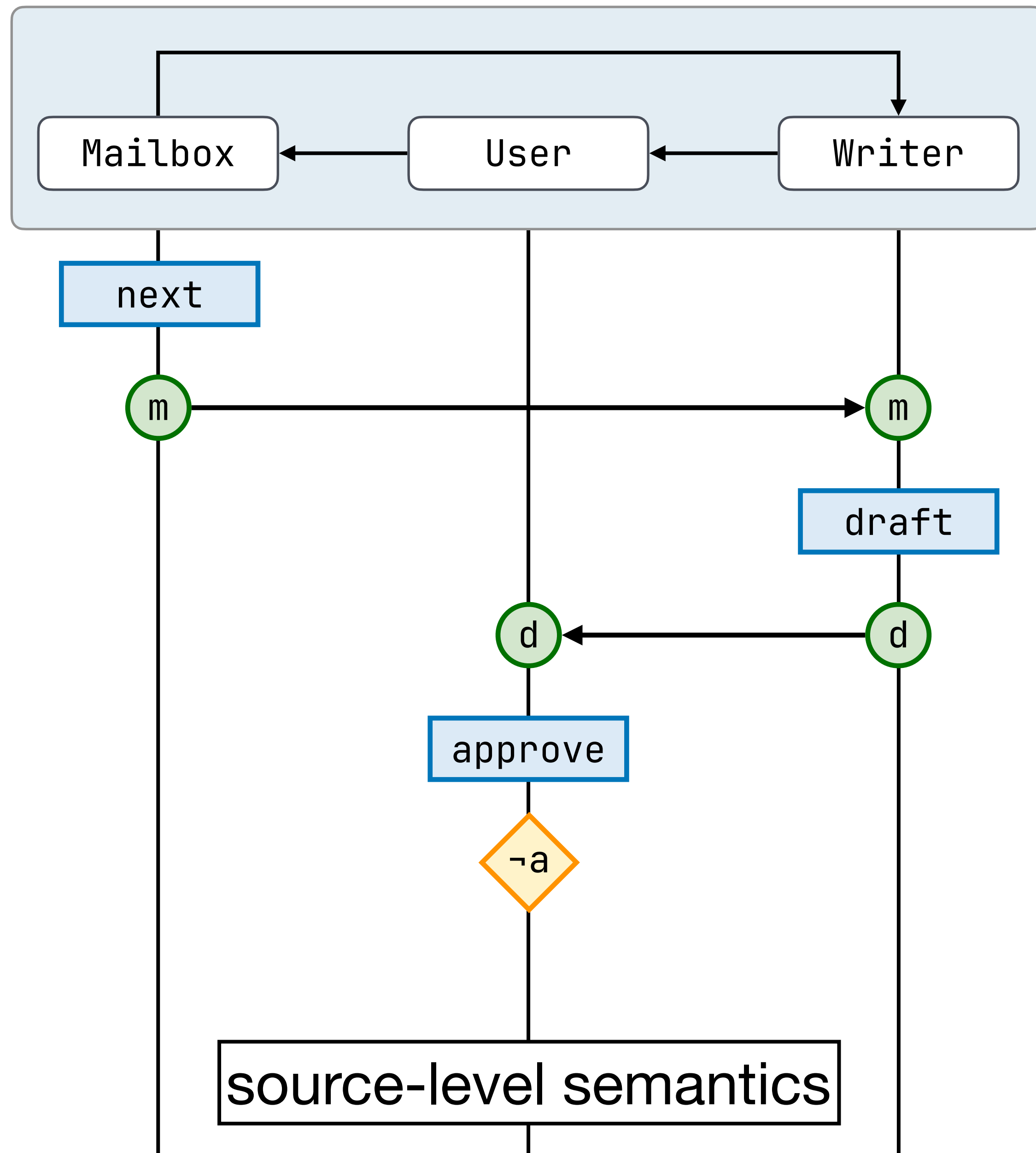
```
else
```

```
    send User(⊥) → Mailbox
```

- Choreographic programming / global protocols
 - ▶ endpoint projection, deadlock-freedom

[Carbone-Montesi '13; Honda-Yoshida-Carbone '16]
- This work (building on [Bollig-Függer-Nowak '26]):
 - ▶ realizability enforced by control broadcasts

Multi-agent systems as distributed systems



```
@role(User)
```

```
recv User(d) ← Writer
```

```
act User : a = approve(d)
```

```
if a @ User then
```

```
    send User(τ) → Mailbox
```

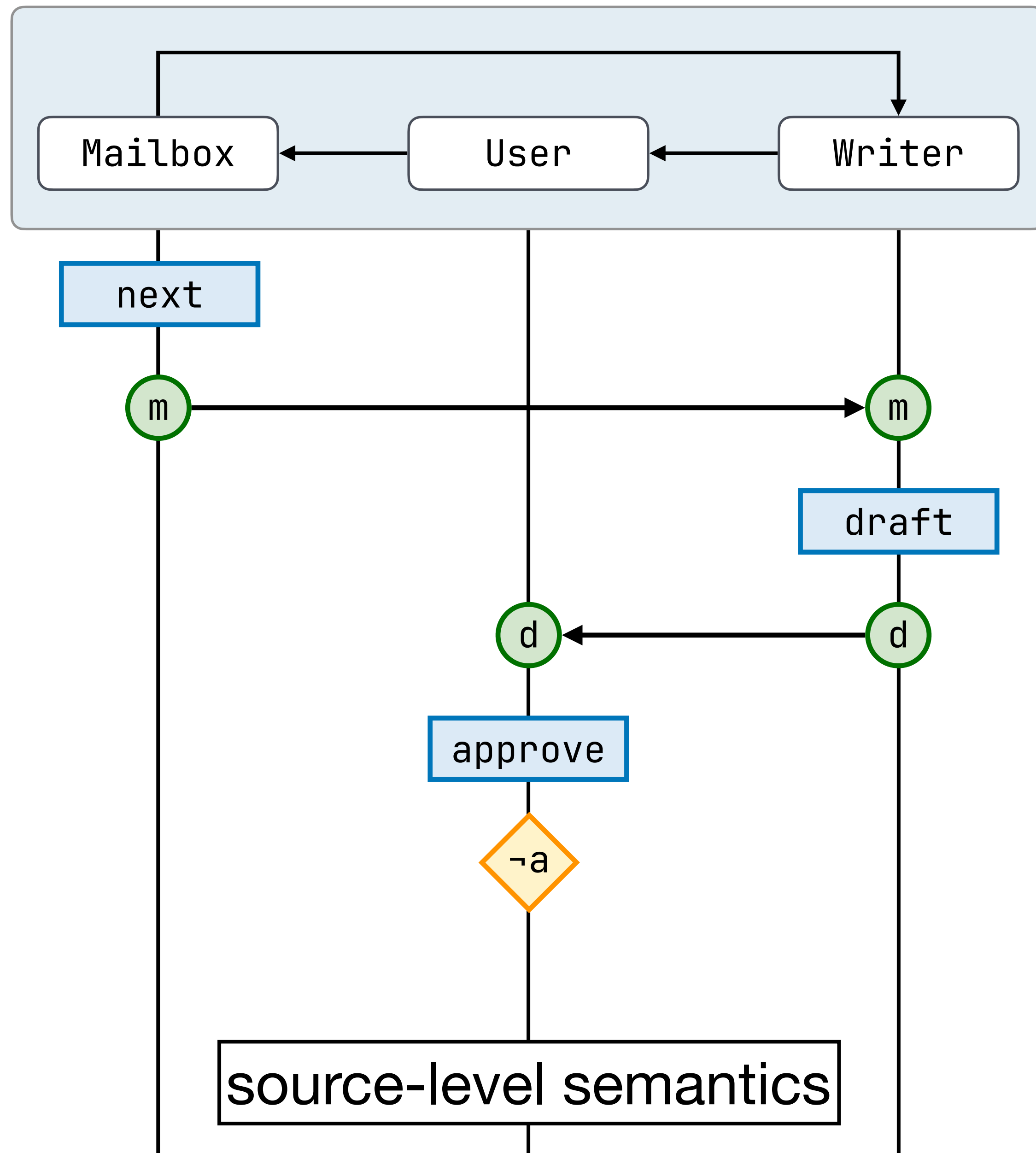
```
    send User(d) → Mailbox
```

```
else
```

```
    send User(⊥) → Mailbox
```

- Choreographic programming / global protocols
 - ▶ endpoint projection, deadlock-freedom
- This work (building on [Bollig-Függer-Nowak '26]):
 - ▶ realizability enforced by control broadcasts
 - ▶ MSC semantics [Katoen-Lambert '98]

Multi-agent systems as distributed systems



```
@role(User)
```

```
recv User(d) ← Writer
```

```
act User : a = approve(d)
```

```
if a @ User then
```

```
    send User(τ) → Mailbox
```

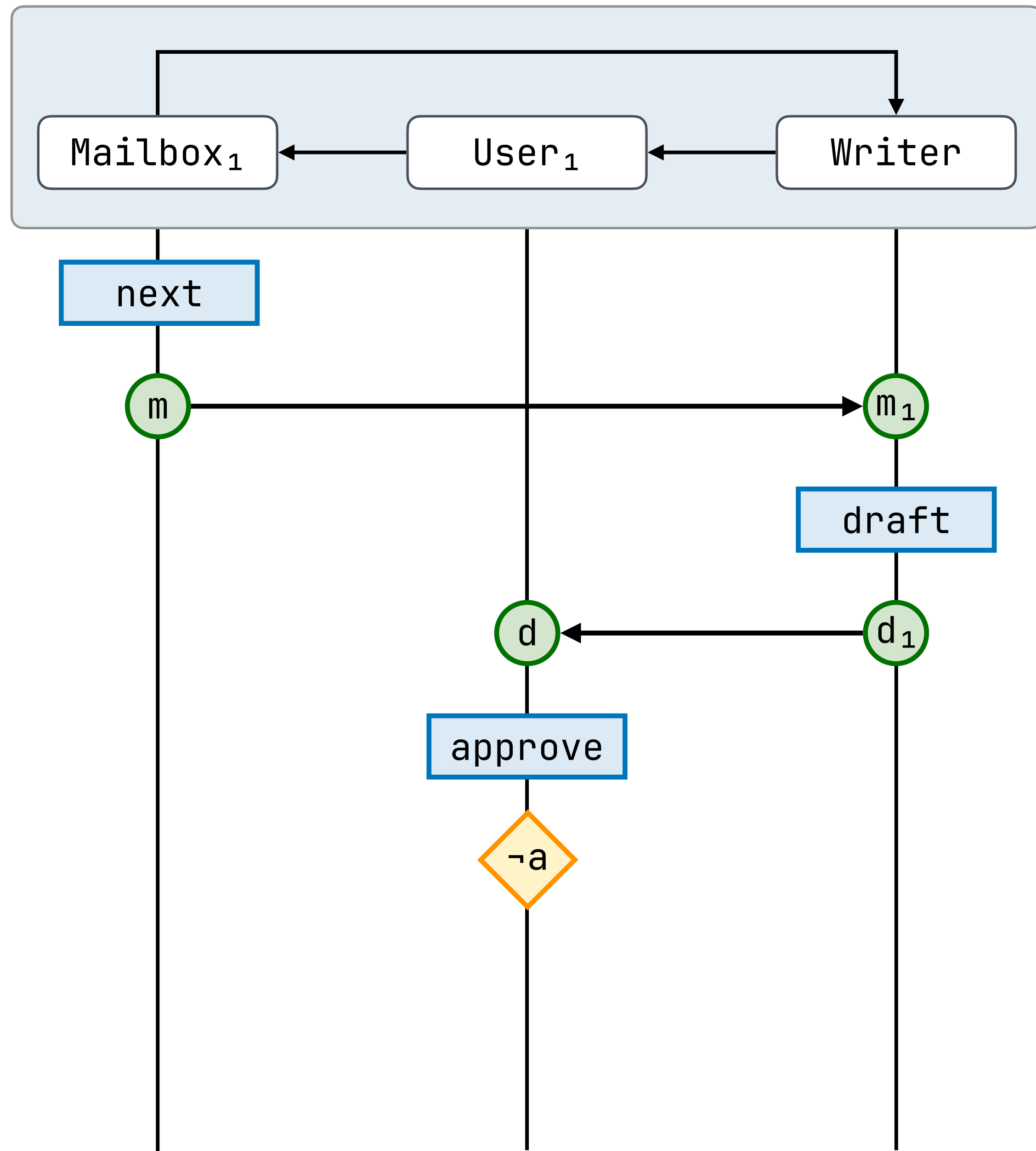
```
    send User(d) → Mailbox
```

```
else
```

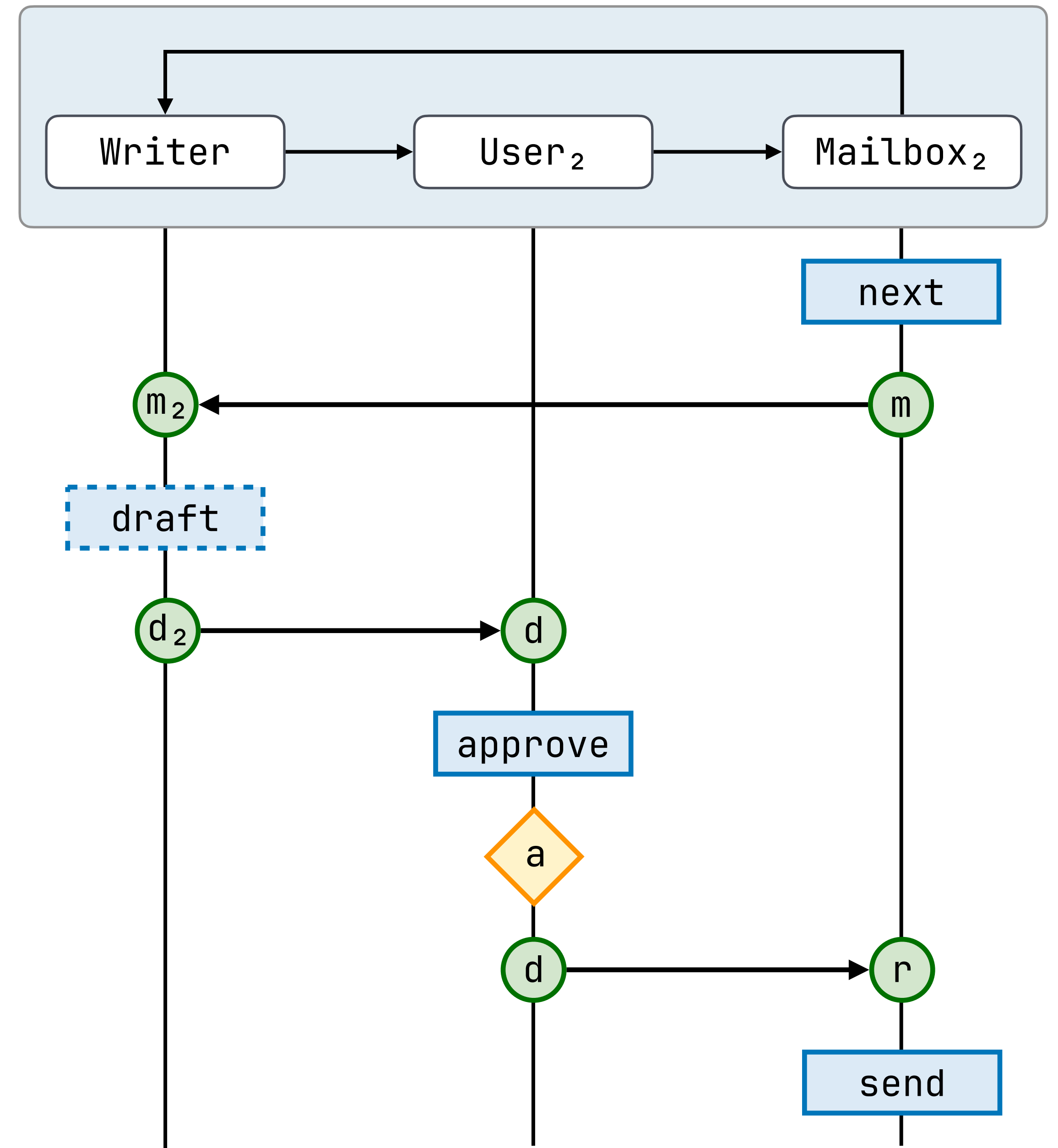
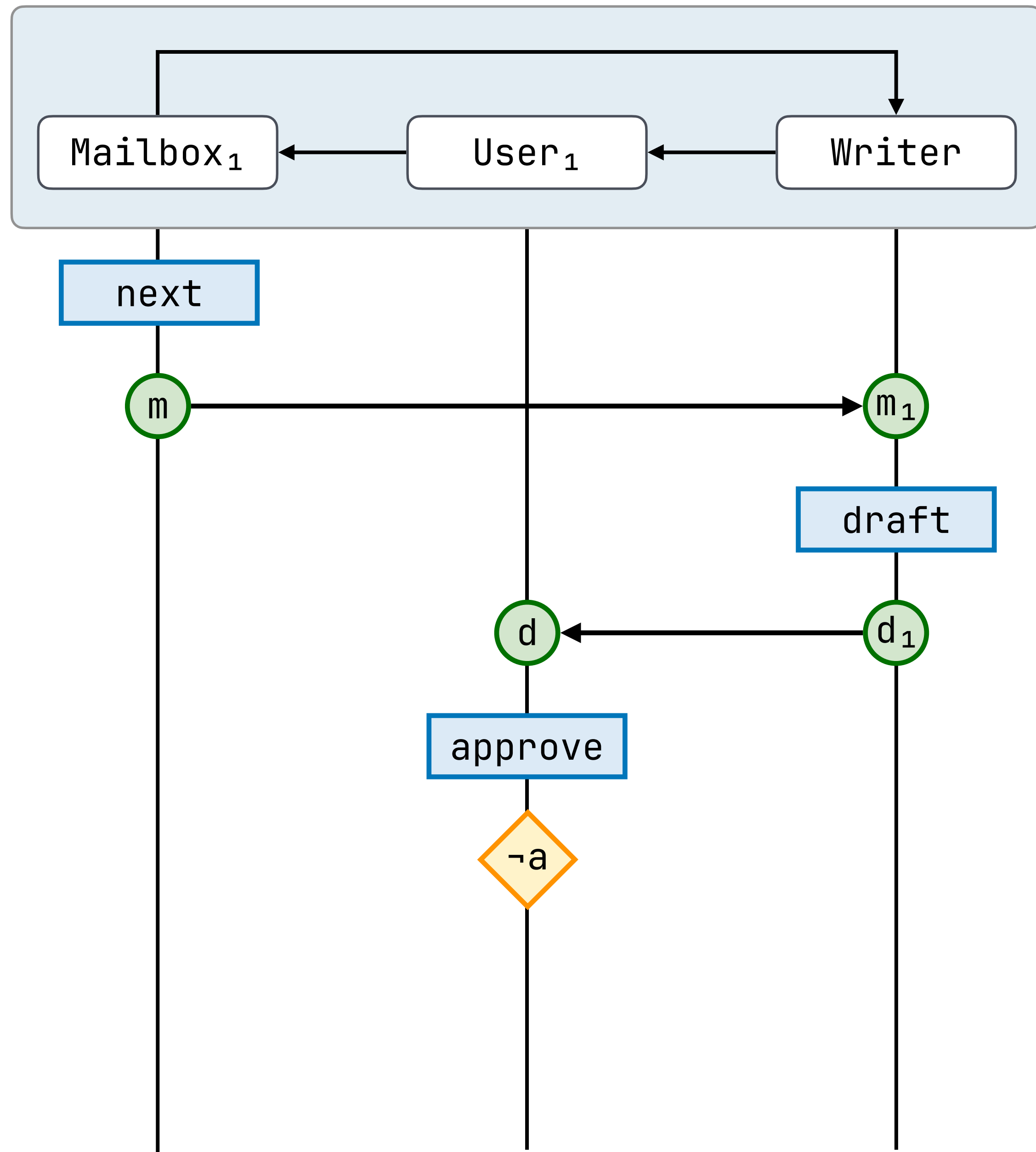
```
    send User(⊥) → Mailbox
```

- Choreographic programming / global protocols
 - ▶ endpoint projection, deadlock-freedom
- [Carbone-Montesi '13; Honda-Yoshida-Carbone '16]
- This work (building on [Bollig-Függer-Nowak '26]):
 - ▶ realizability enforced by control broadcasts
 - ▶ MSC semantics [Katoen-Lambert '98]
 - ▶ deadlock-free shared-lifeline parallelism

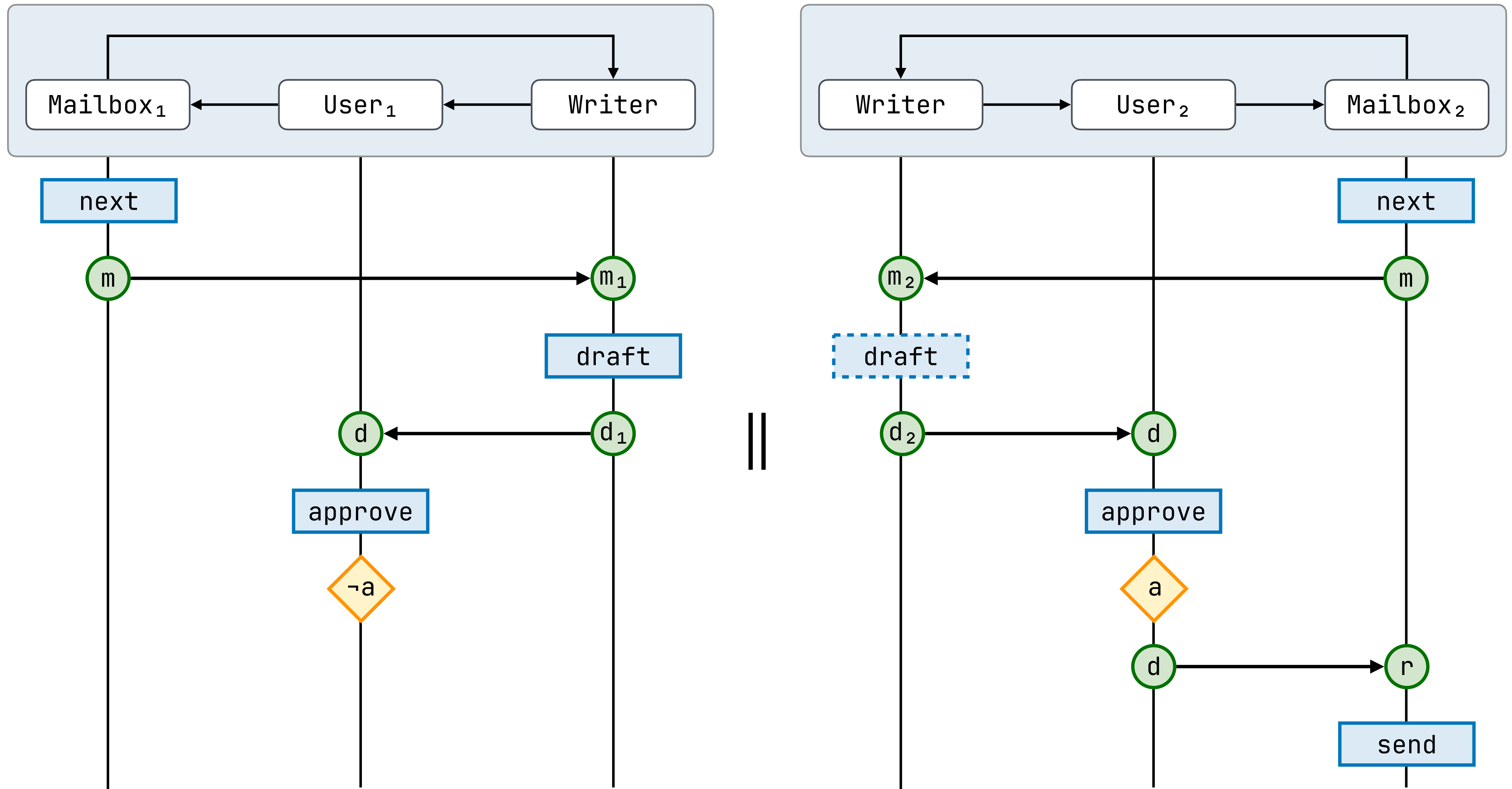
The parallel operator



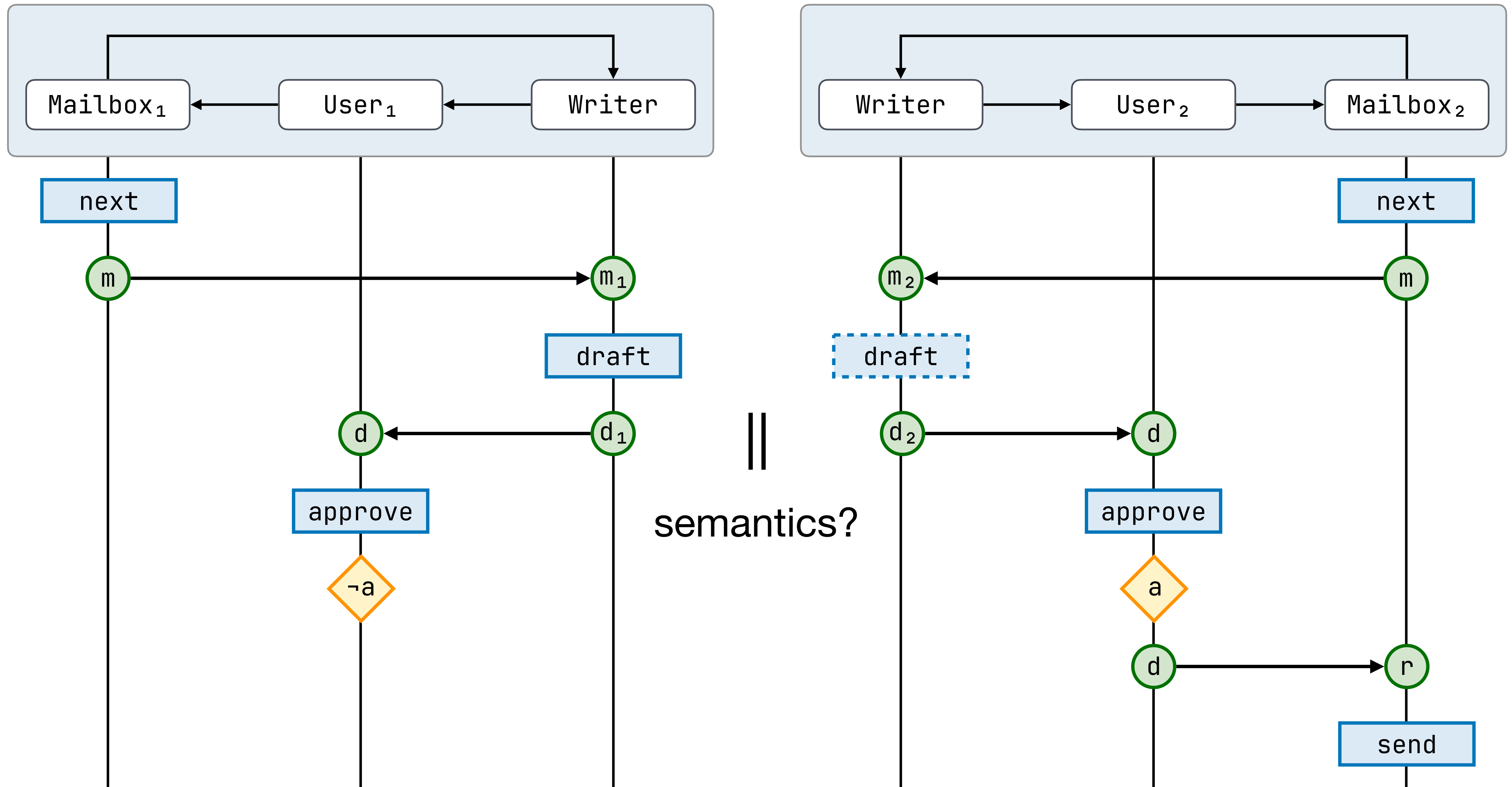
The parallel operator



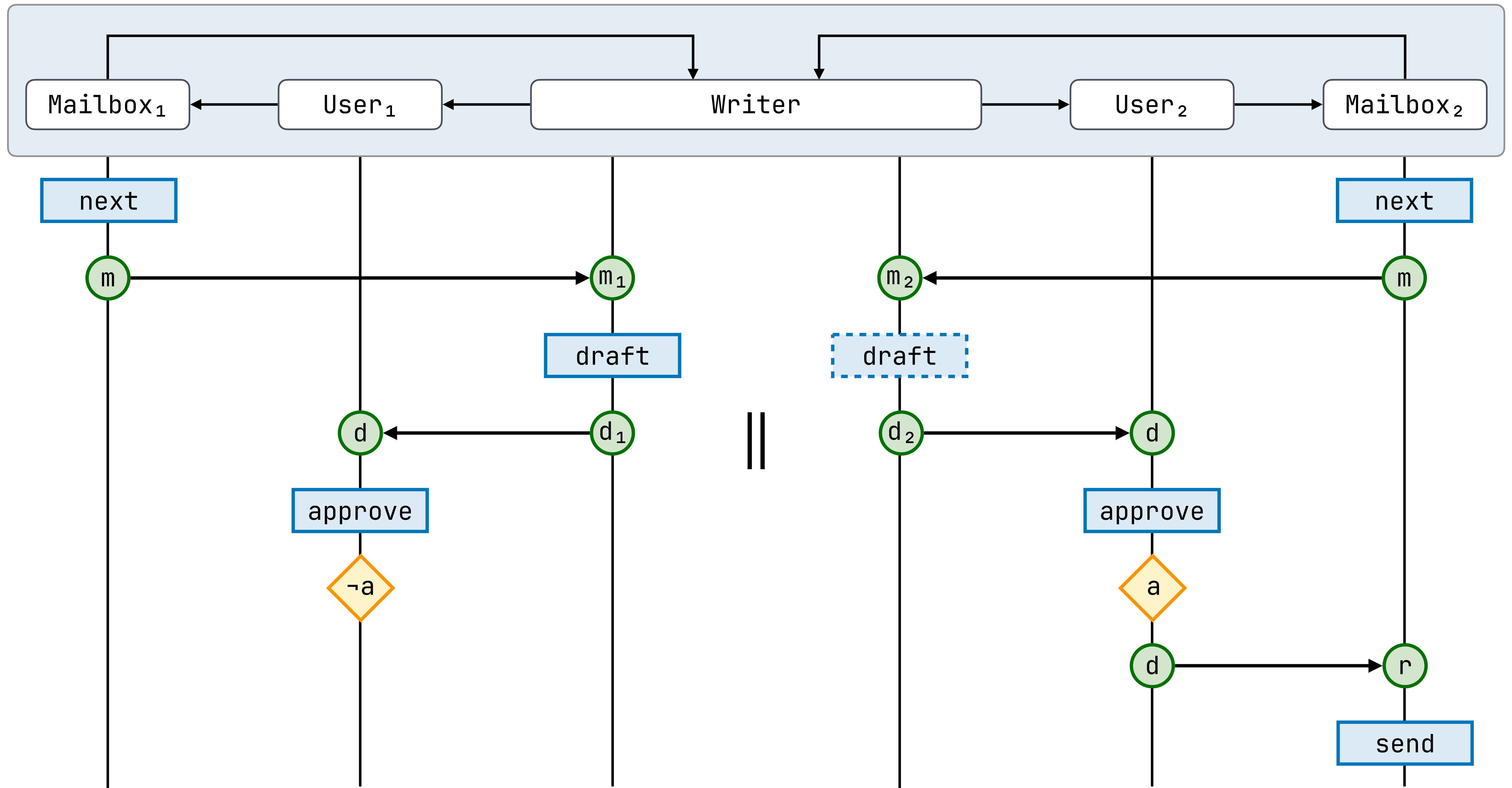
The parallel operator



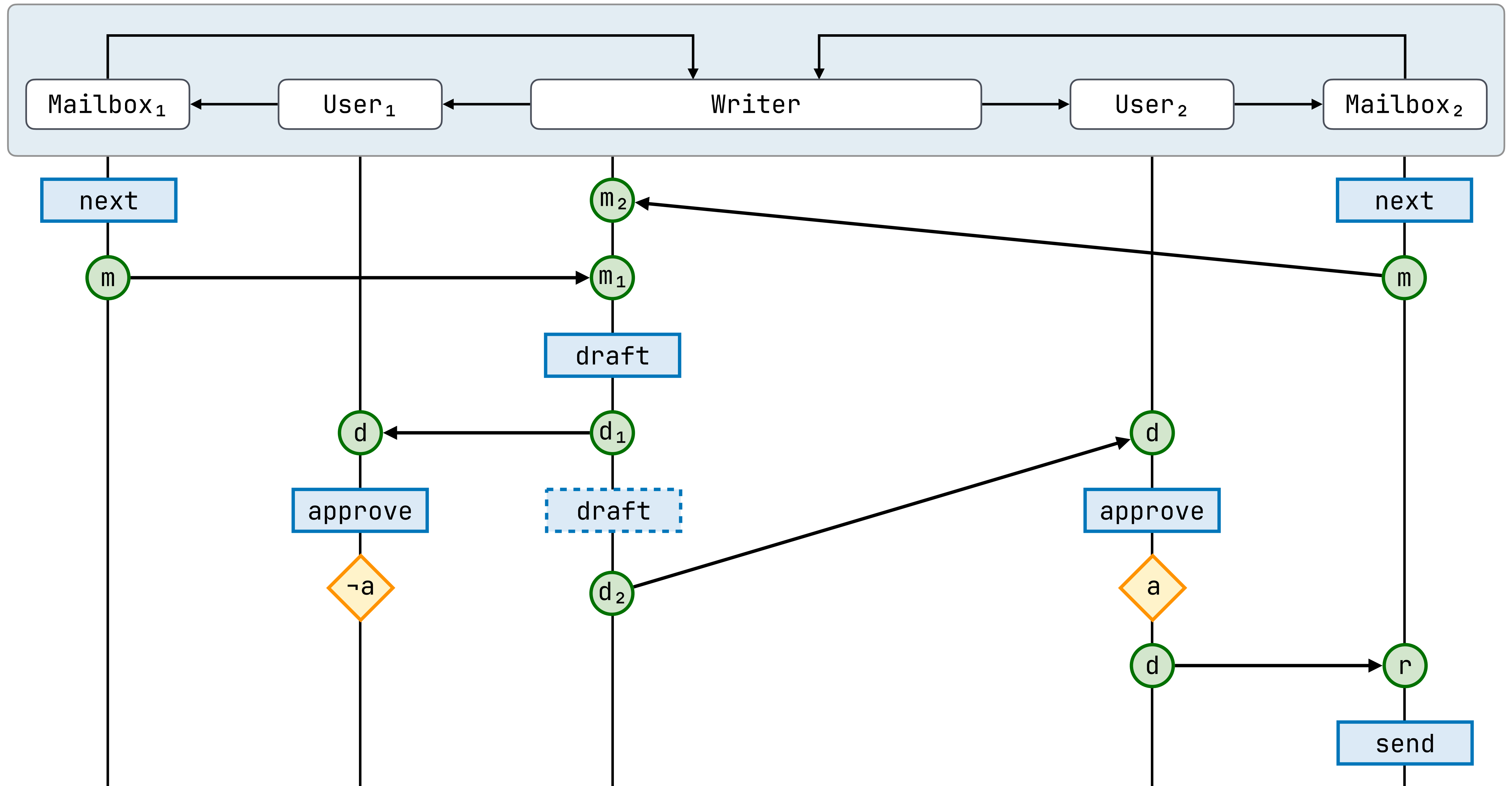
The parallel operator



The parallel operator



The parallel operator



The parallel operator

@workflow

```
act Mailbox1 : m = next()
msg Mailbox1(m) → Writer(m1)
act Writer : d1 = draft(m1)
msg Writer(d1) → User1(d)
act User1 : a = approve(d)

if a @ User1 then
  msg User1(d) → Mailbox1(r)
  act Mailbox1 : status = send(r)
else
  skip
```

```
act Mailbox2 : m = next()
msg Mailbox2(m) → Writer(m2)
act Writer : d2 = draft(m2)
msg Writer(d2) → User2(d)
act User2 : a = approve(d)

if a @ User2 then
  msg User2(d) → Mailbox2(r)
  act Mailbox2 : status = send(r)
else
  skip
```

The parallel operator

@workflow

```
act Mailbox1 : m = next()
msg Mailbox1(m) → Writer(m1)
act Writer : d1 = draft(m1)
msg Writer(d1) → User1(d)
act User1 : a = approve(d)

if a @ User1 then
  msg User1(d) → Mailbox1(r)
  act Mailbox1 : status = send(r)
else
  skip
```

||

```
act Mailbox2 : m = next()
msg Mailbox2(m) → Writer(m2)
act Writer : d2 = draft(m2)
msg Writer(d2) → User2(d)
act User2 : a = approve(d)

if a @ User2 then
  msg User2(d) → Mailbox2(r)
  act Mailbox2 : status = send(r)
else
  skip
```

The parallel operator: endpoint projection

@role(Writer)

```
recv Writer( $m_1$ )  $\leftarrow$  Mailbox $_1$   
act Writer :  $d_1$  = draft( $m_1$ )  
send Writer( $d_1$ )  $\rightarrow$  User $_1$ 
```

```
recv Writer( $m_2$ )  $\leftarrow$  Mailbox $_2$   
act Writer :  $d_2$  = draft( $m_2$ )  
send Writer( $d_2$ )  $\rightarrow$  User $_2$ 
```

The parallel operator: endpoint projection

@role(Writer)

```
recv Writer(m1) ← Mailbox1  
act Writer : d1 = draft(m1)  
send Writer(d1) → User1
```

$\parallel_{\text{loc}}$

```
recv Writer(m2) ← Mailbox2  
act Writer : d2 = draft(m2)  
send Writer(d2) → User2
```

The parallel operator: endpoint projection

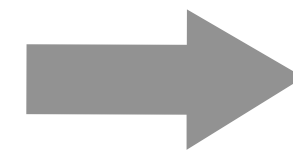
@role(Writer)

```
recv Writer(m1) ← Mailbox1[c1]  
act Writer : d1 = draft(m1)  
send Writer(d1) → User1[c1]
```

$\parallel_{\text{loc}}$

```
recv Writer(m2) ← Mailbox2[c2]  
act Writer : d2 = draft(m2)  
send Writer(d2) → User2[c2]
```

shared channel across branches



FIFO channels per branch

The parallel operator: endpoint projection

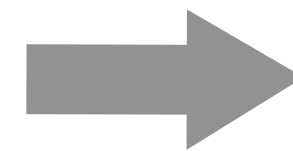
@role(Writer)

```
recv Writer(m1) ← Mailbox1[c1]  
act Writer : d1 = draft(m1)  
send Writer(d1) → User1[c1]
```

$\parallel_{\text{loc}}$

```
recv Writer(m2) ← Mailbox2[c2]  
act Writer : d2 = draft(m2)  
send Writer(d2) → User2[c2]
```

shared channel across branches



FIFO channels per branch

[Dalla Preda et al. '17]

shared agents, local $\parallel$, control messages, **synchronous**

[Guanciale-Tuosto '18]

shared agents, local product, **well-forkedness**

[Jongmans-Yoshida '20]

local $\parallel$, weak bisimulation, **synchronous**

[Kapus-Kolar '20]

async FIFO, shared agents, **static action separation**

[Edixhoven et al. '24]

async FIFO, shared agents, **well-channeledness**

The parallel operator: endpoint projection

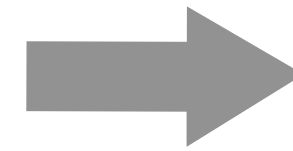
@role(Writer)

```
recv Writer(m1) ← Mailbox1[c1]  
act Writer : d1 = draft(m1)  
send Writer(d1) → User1[c1]
```

$\parallel_{\text{loc}}$

```
recv Writer(m2) ← Mailbox2[c2]  
act Writer : d2 = draft(m2)  
send Writer(d2) → User2[c2]
```

shared channel across branches



FIFO channels per branch

[Dalla Preda et al. '17]

shared agents, local $\parallel$, control messages, **synchronous**

[Guanciale-Tuosto '18]

shared agents, local product, **well-forkedness**

[Jongmans-Yoshida '20]

local $\parallel$, weak bisimulation, **synchronous**

[Kapus-Kolar '20]

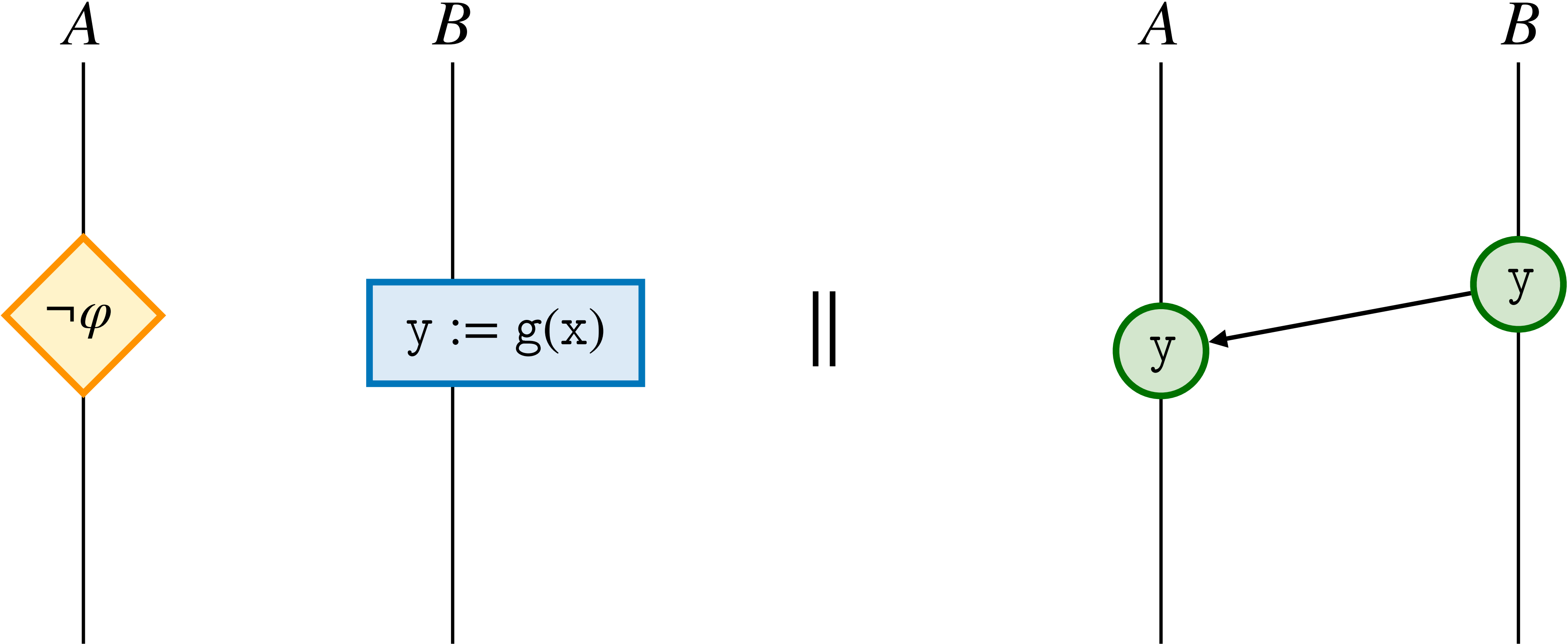
async FIFO, shared agents, **static action separation**

[Edixhoven et al. '24]

async FIFO, shared agents, **well-channeledness**

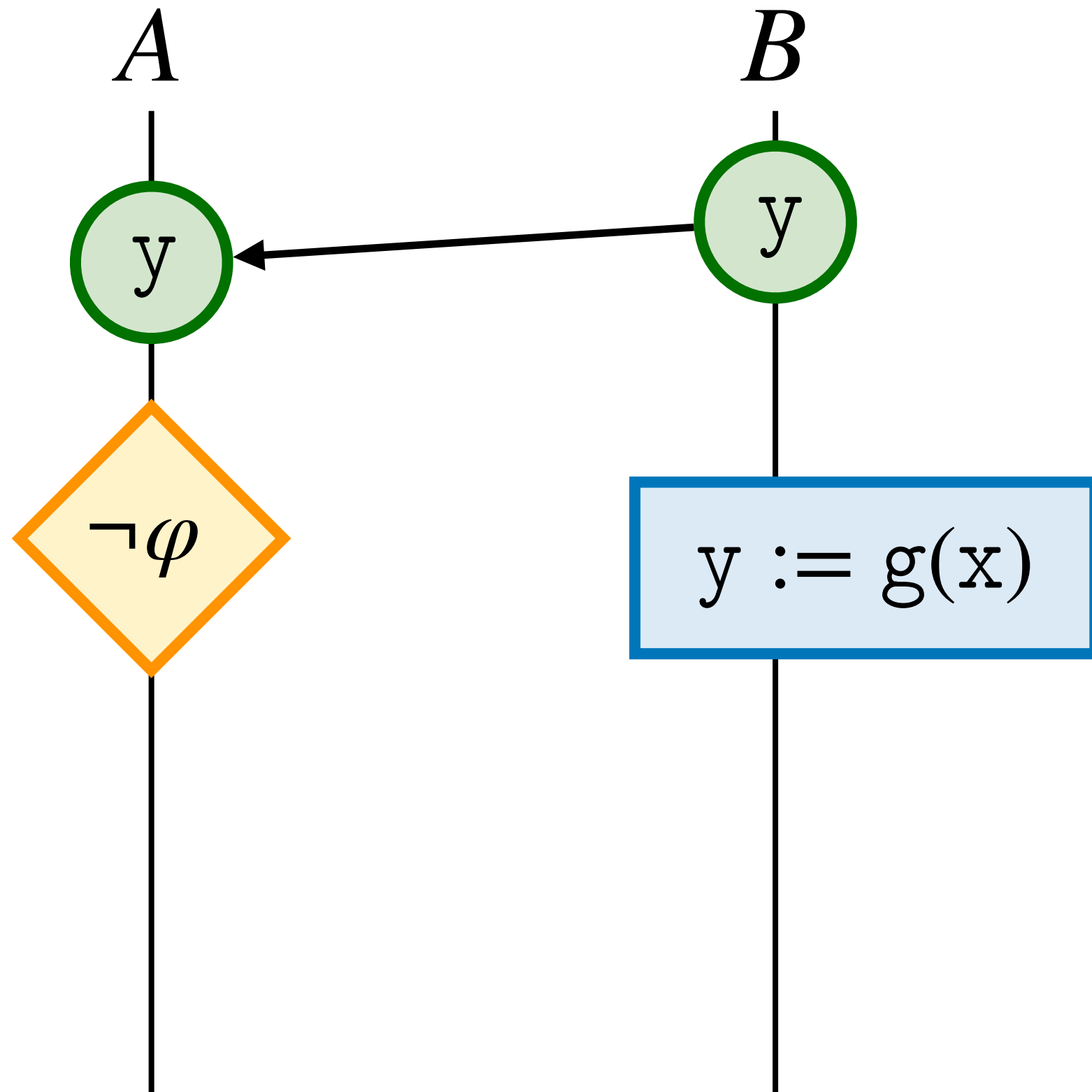
This work: shared agents, async FIFO, control messages, branch-local channels, exact MSC correspondence

Filtered shuffle semantics

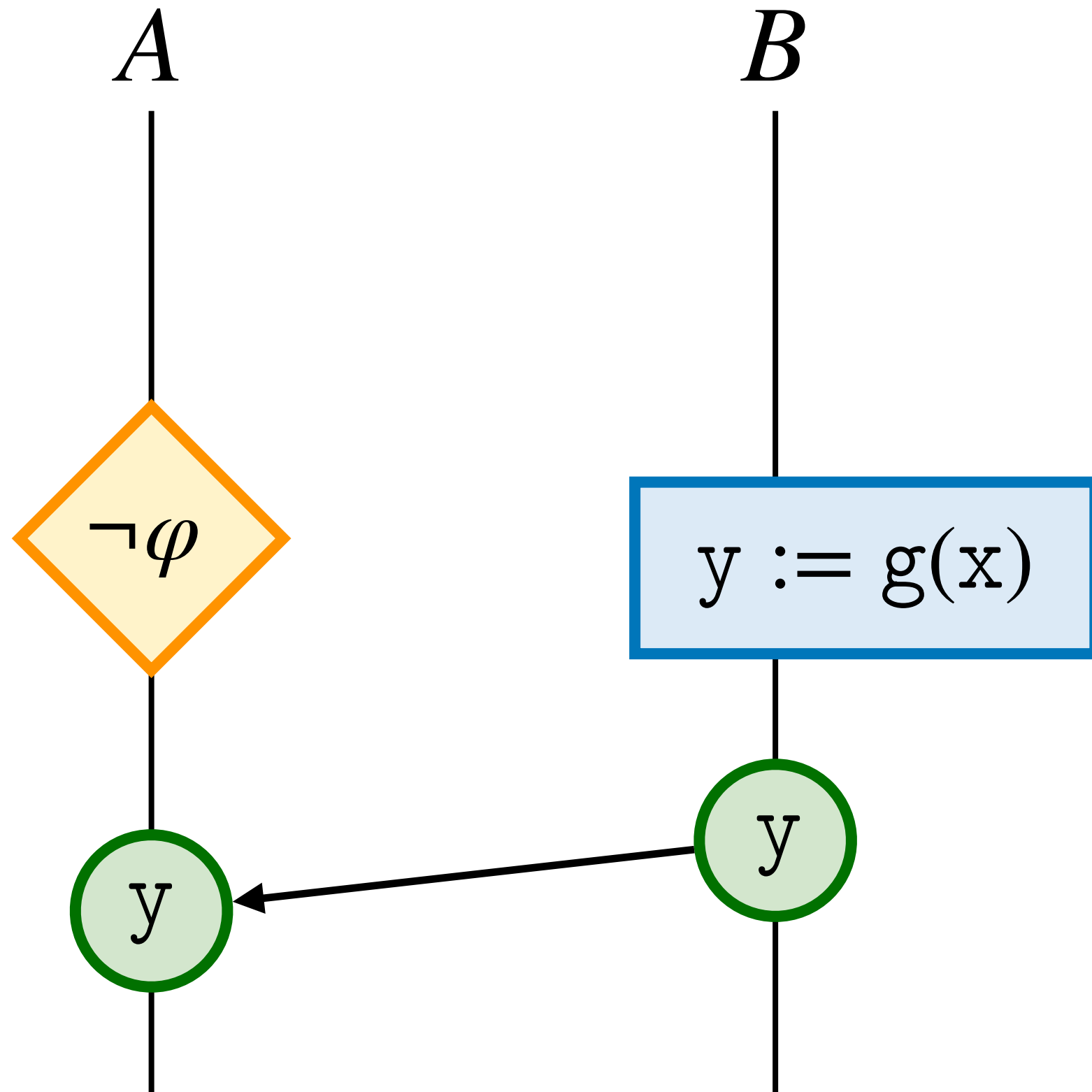


$$\left(\text{if } \varphi @ A \text{ then } B : y := f(x) \text{ else } B : y := g(x) \right) \quad || \quad B(y) \rightarrow A(y)$$

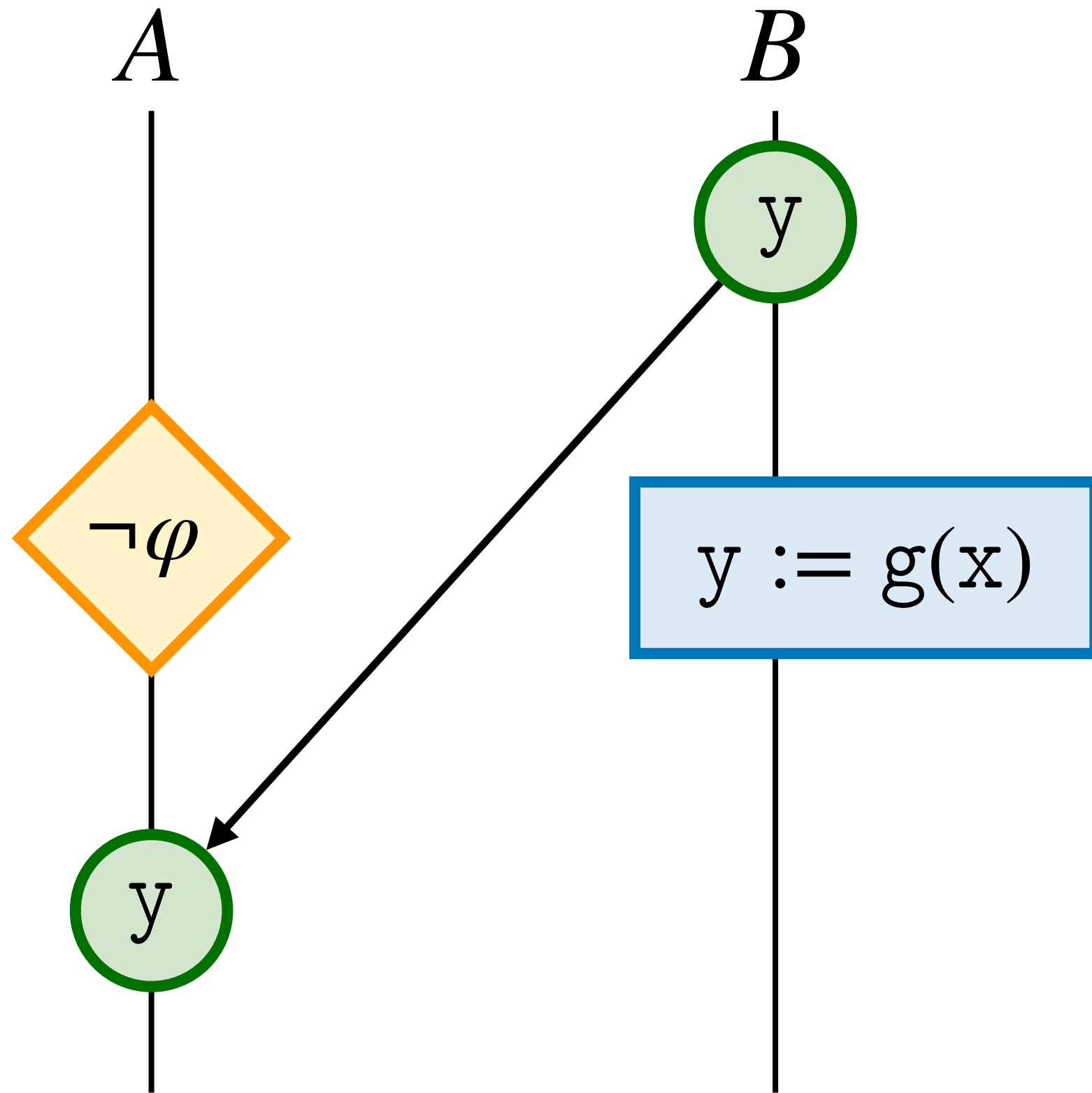
Filtered shuffle semantics



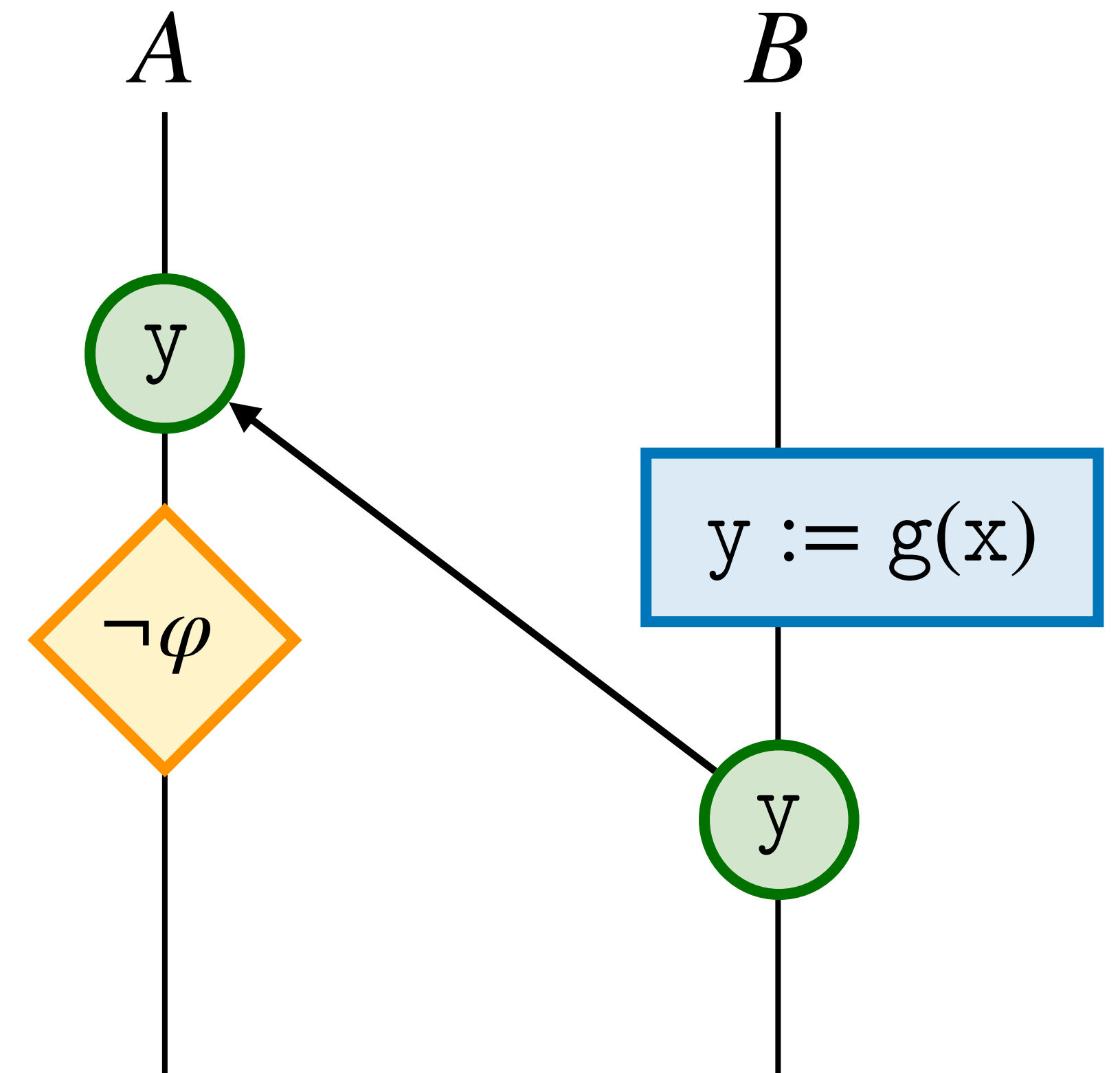
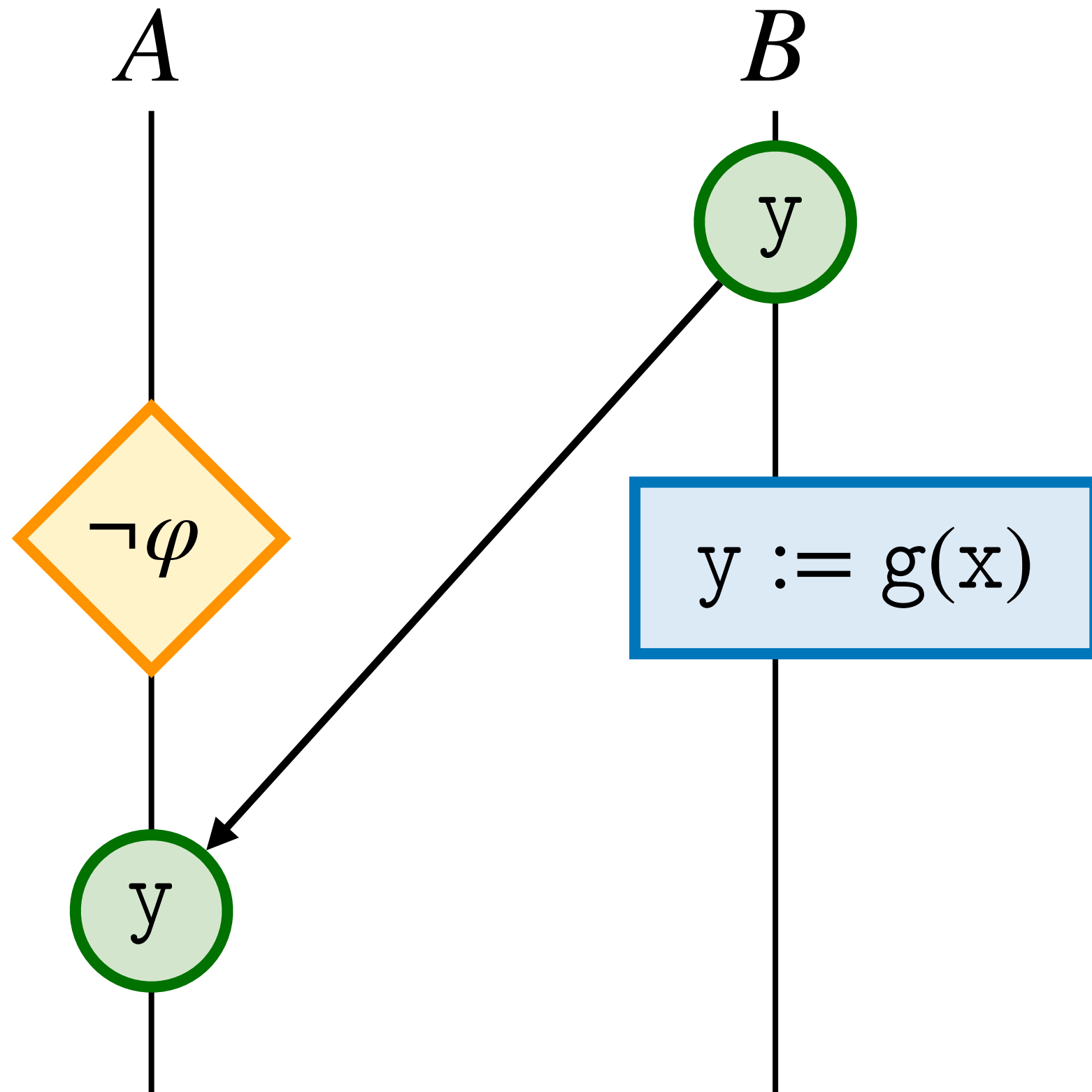
Filtered shuffle semantics



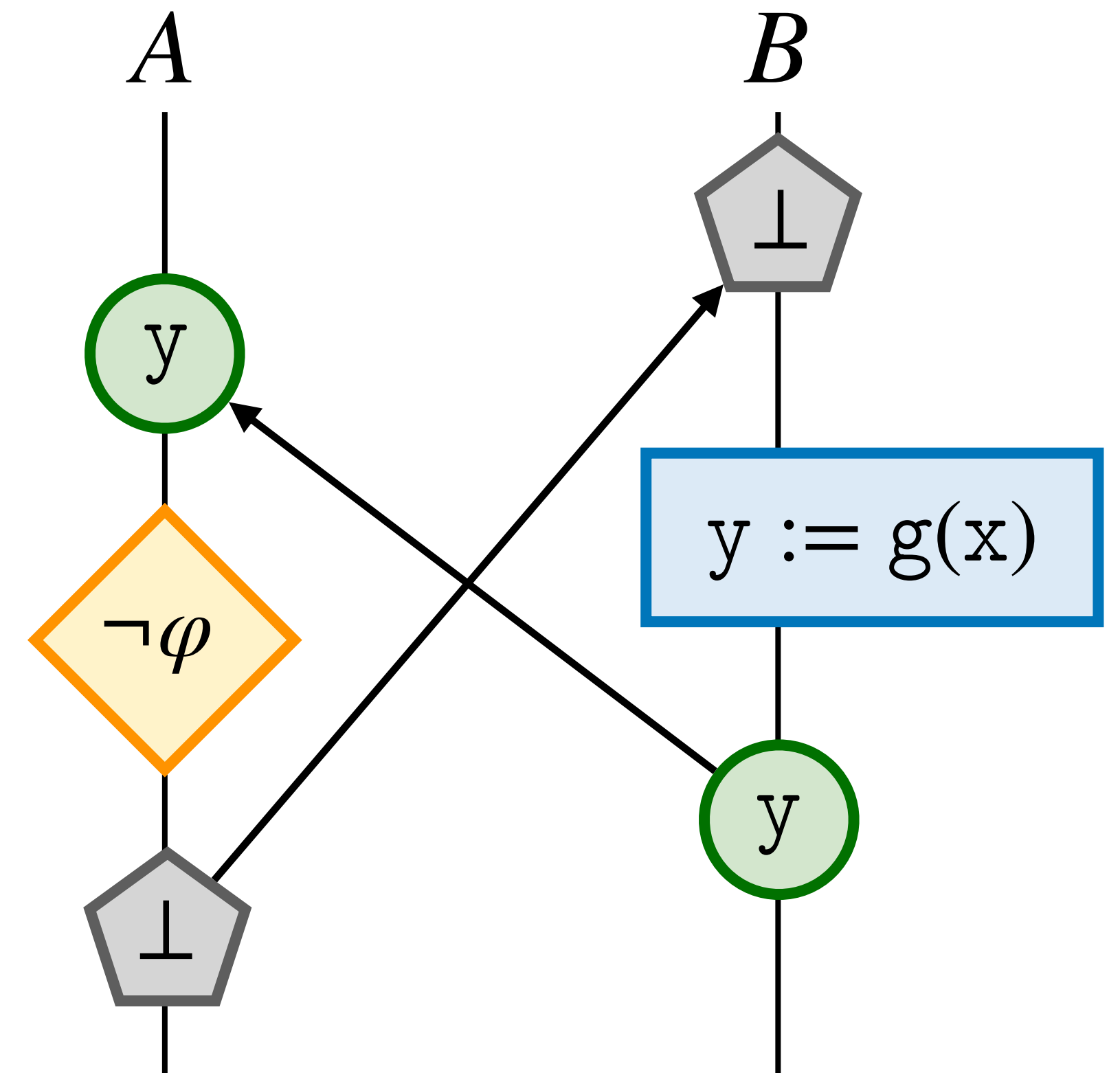
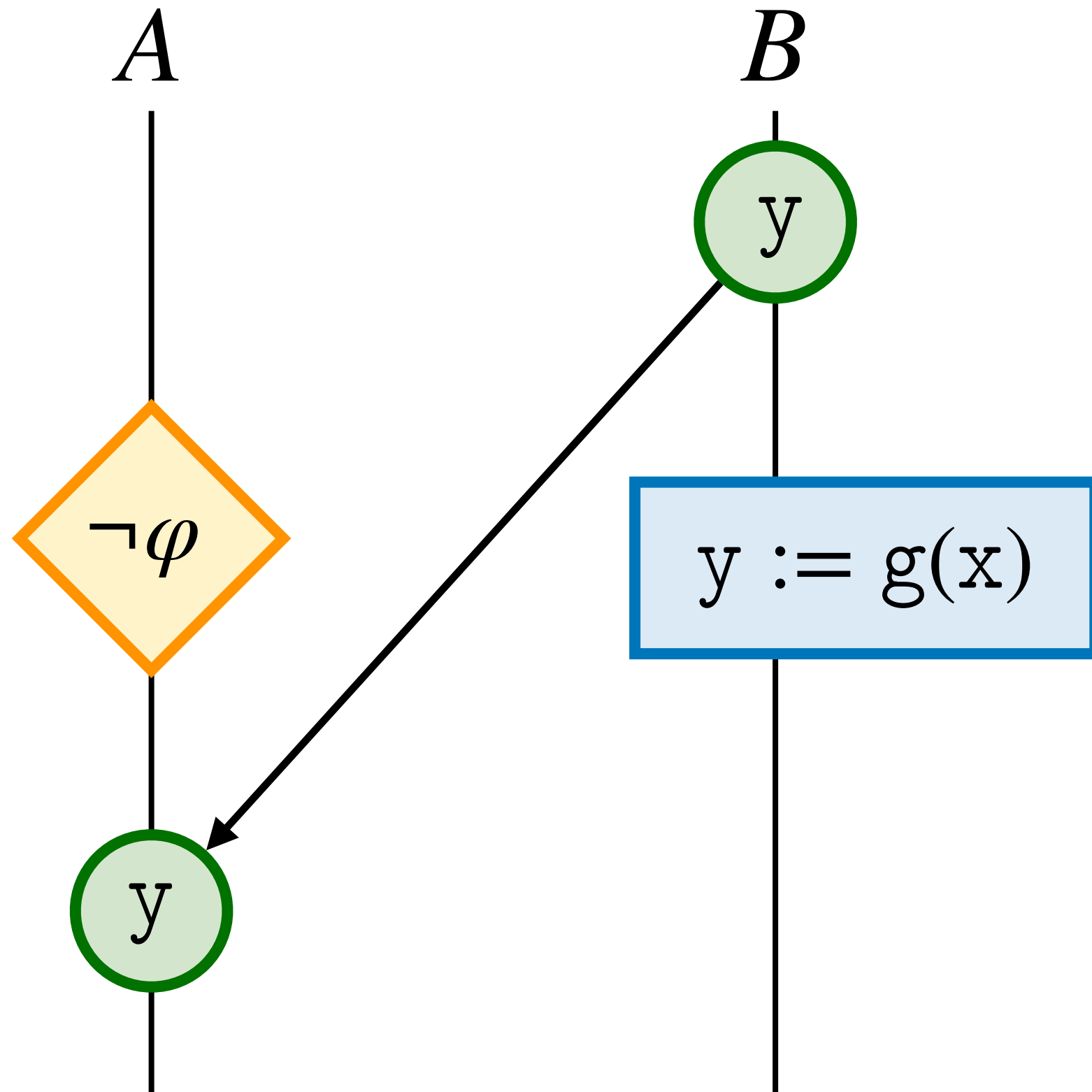
Filtered shuffle semantics



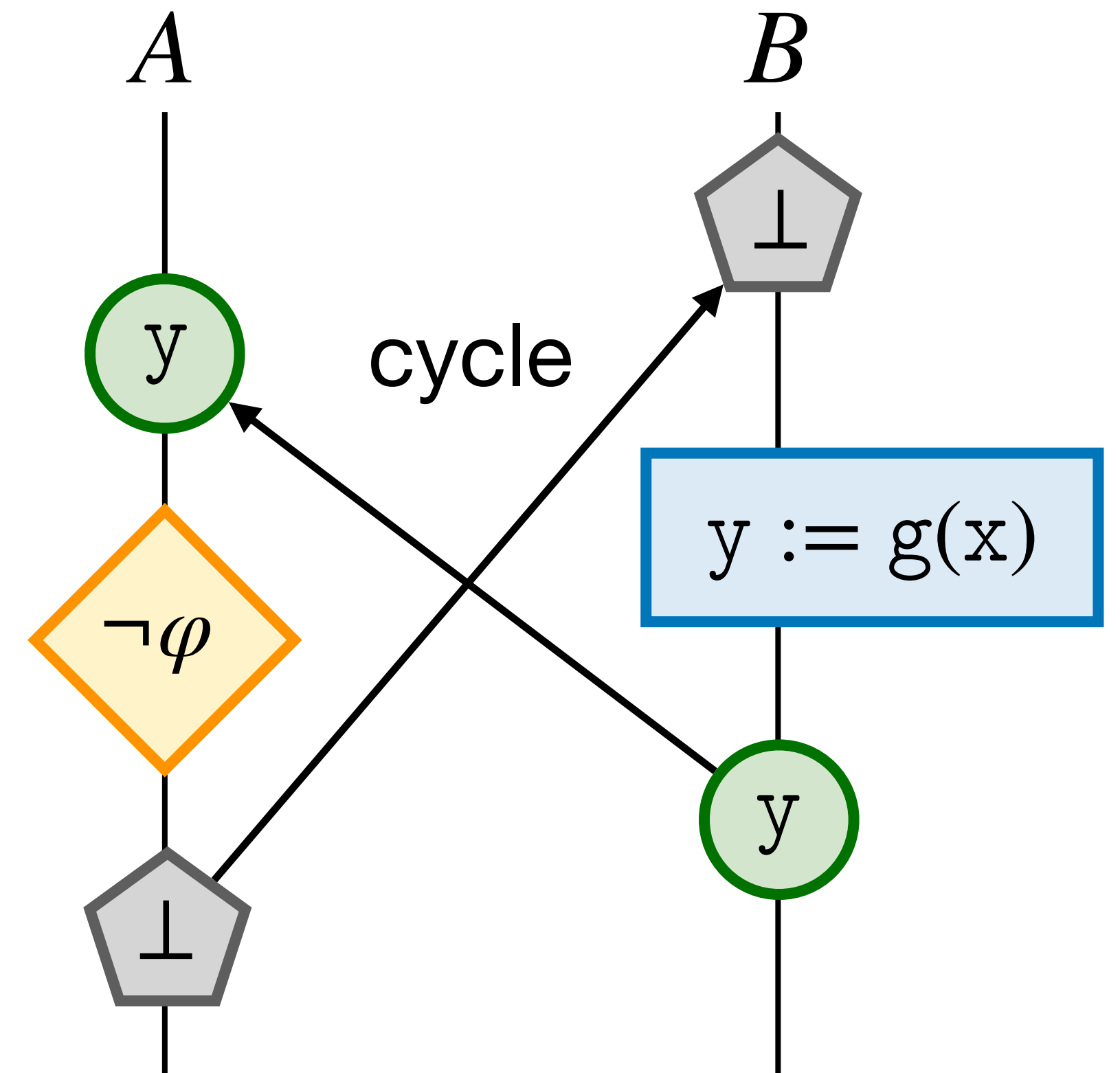
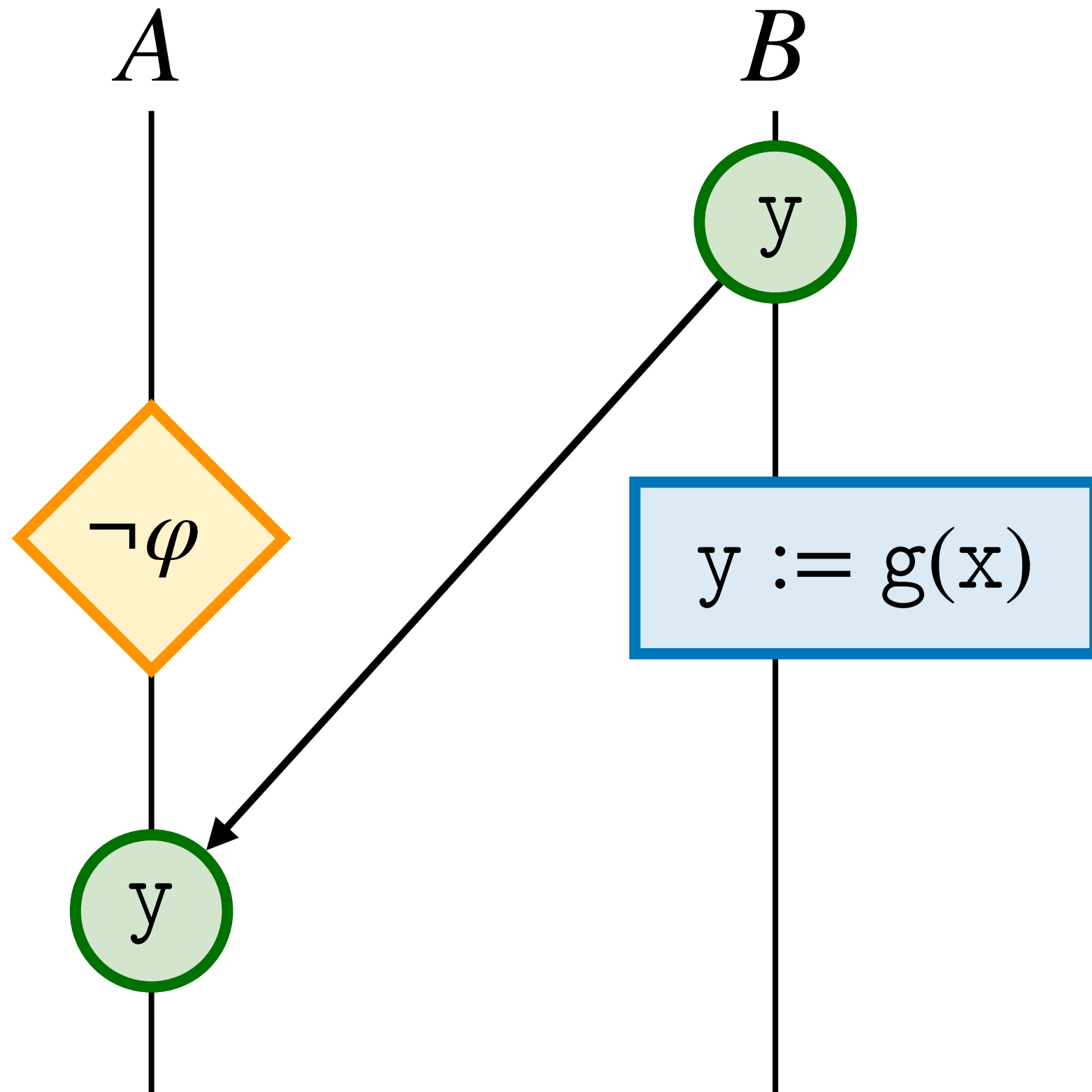
Filtered shuffle semantics



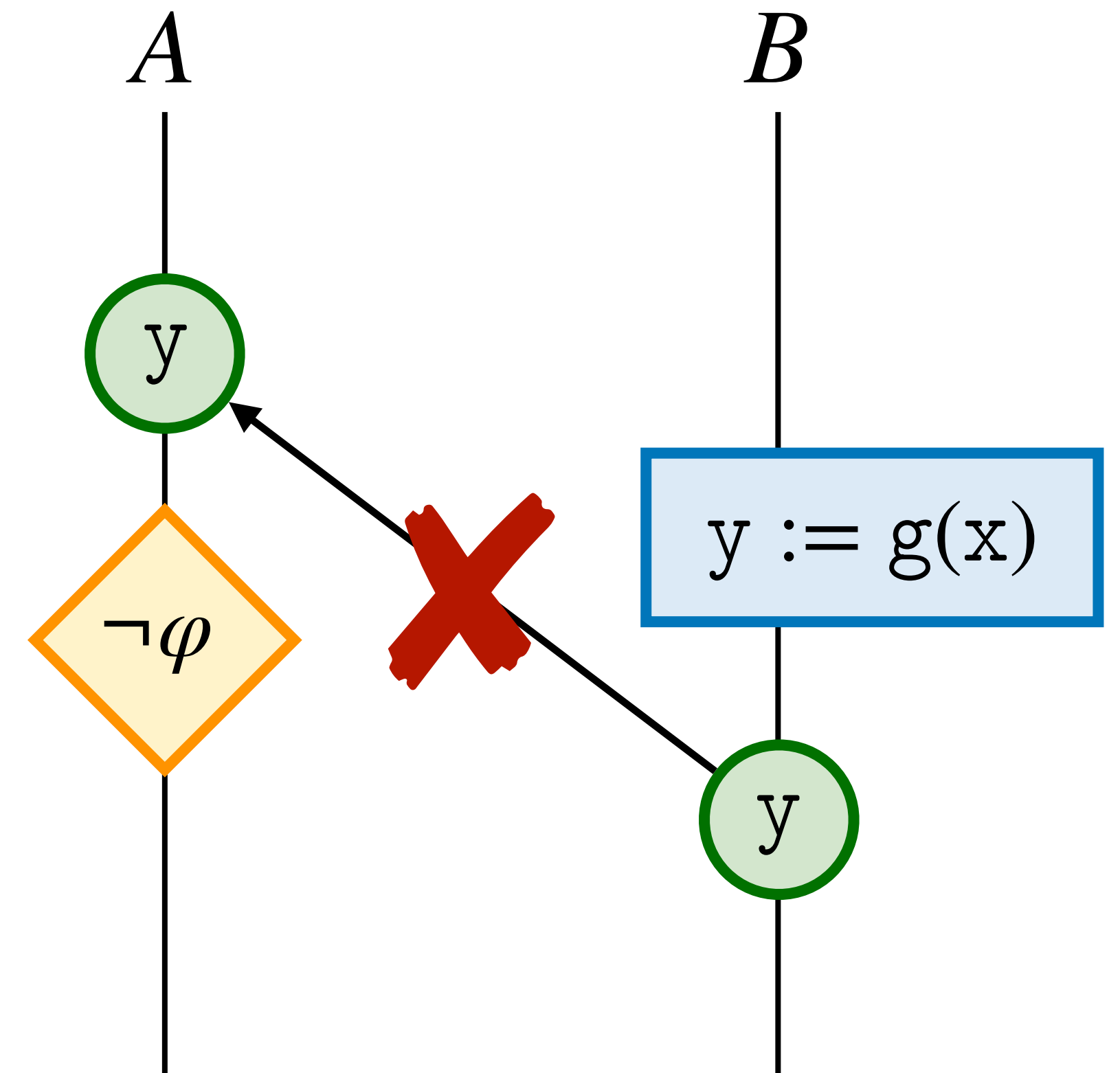
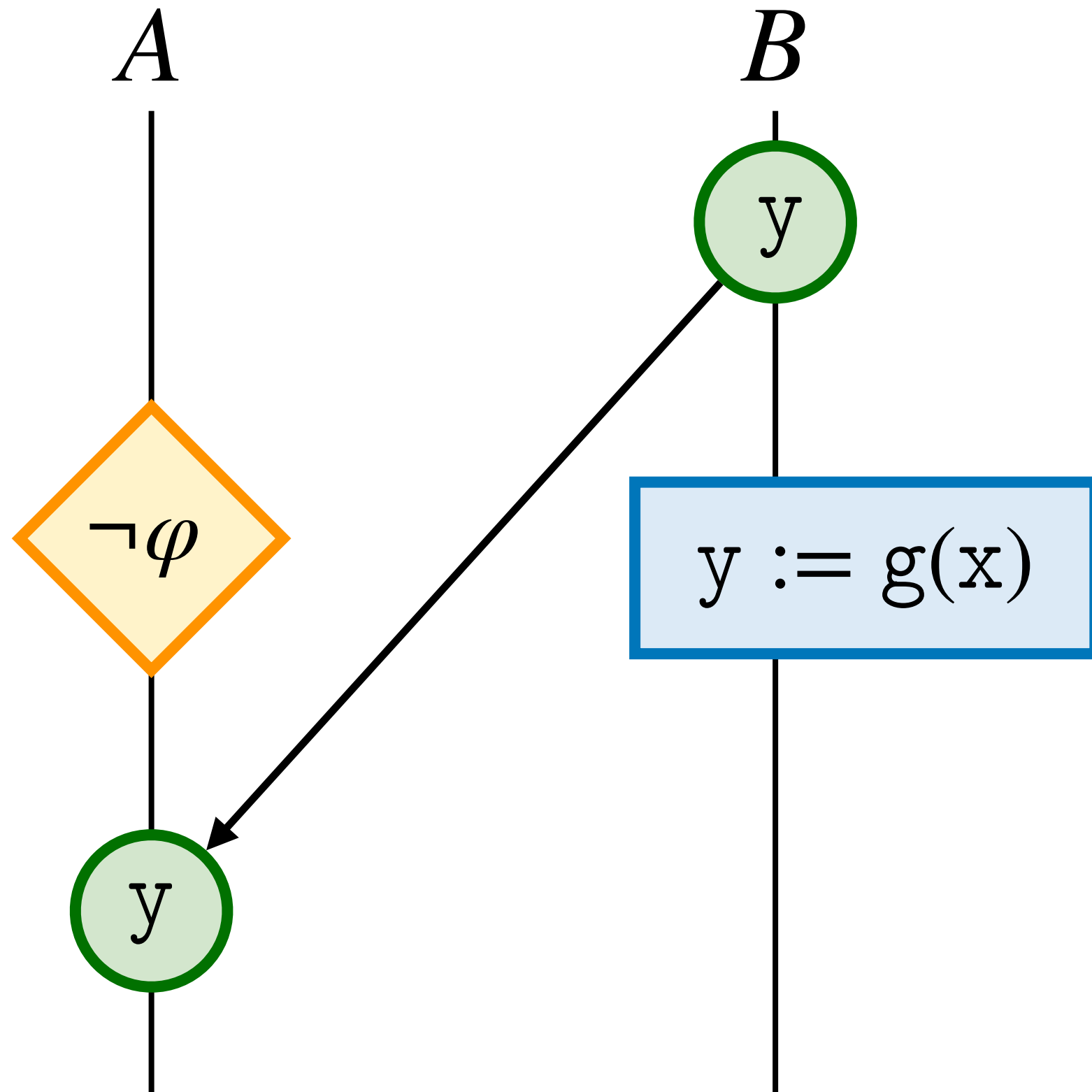
Filtered shuffle semantics



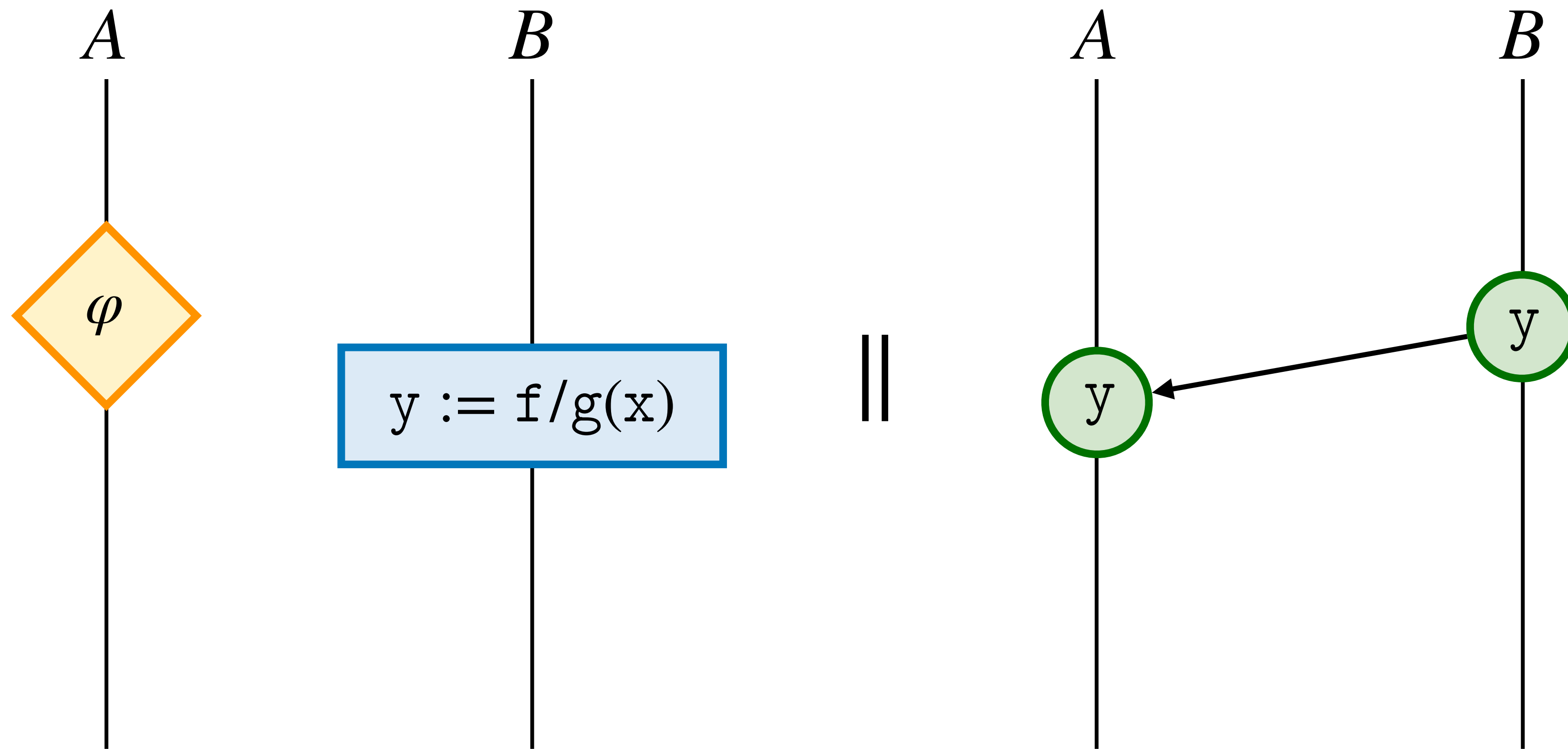
Filtered shuffle semantics



Filtered shuffle semantics

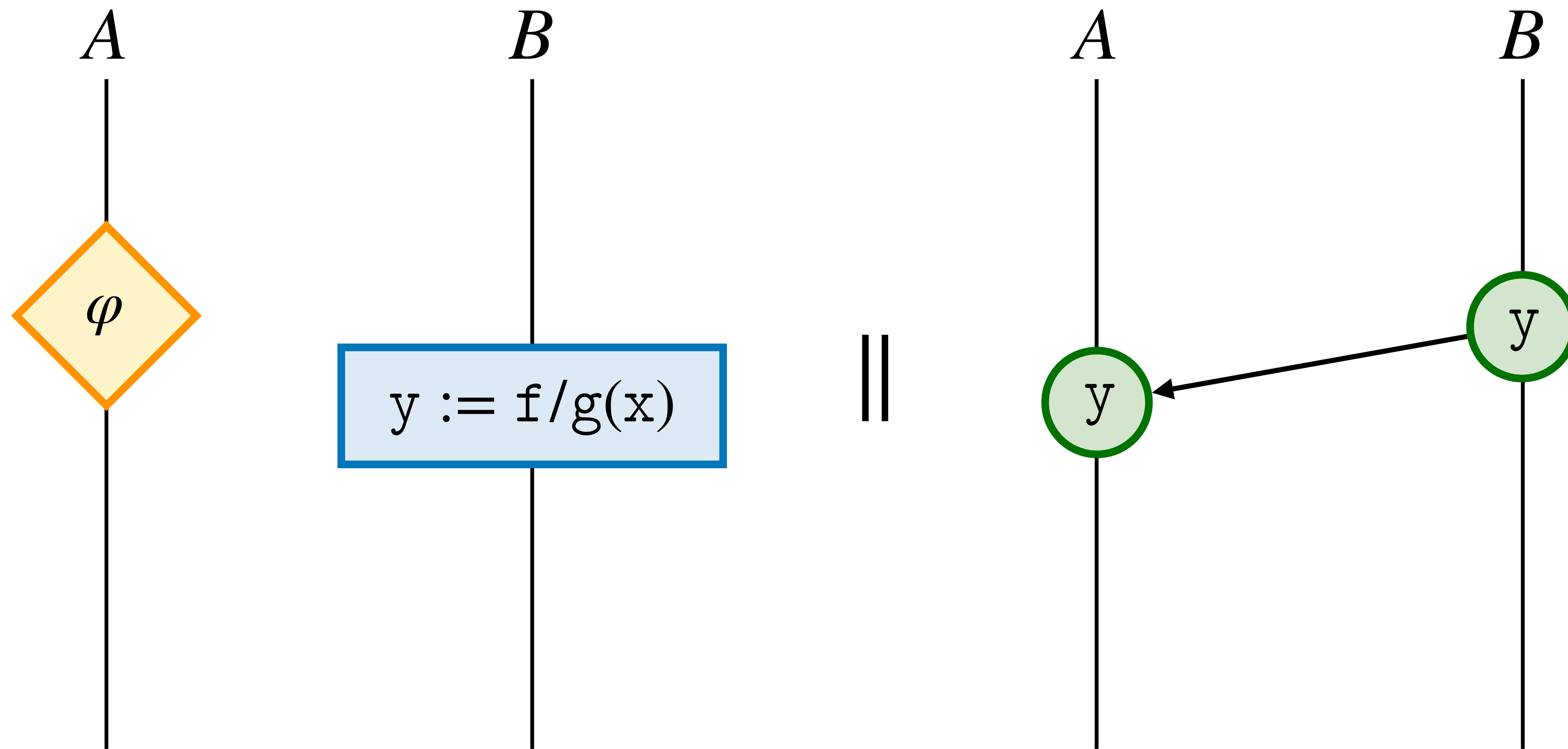


Filtered shuffle semantics

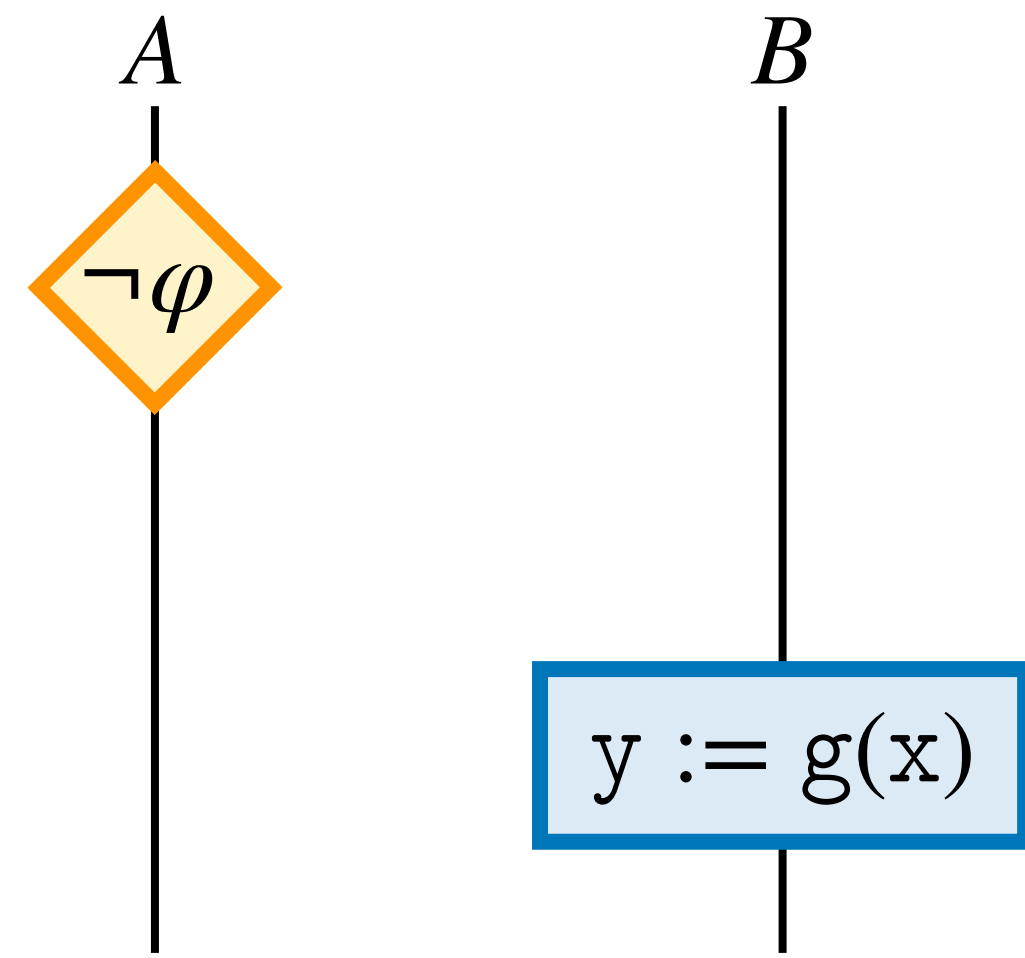
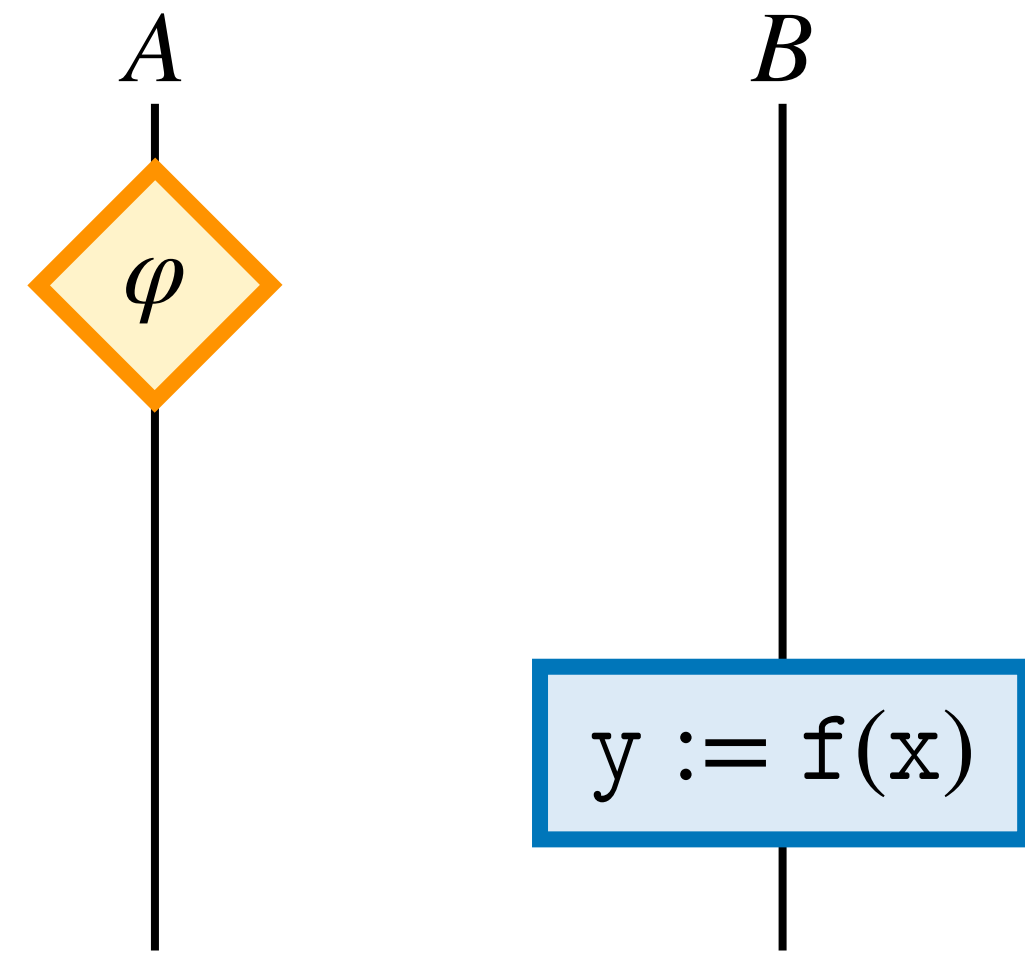


Filtered shuffle semantics

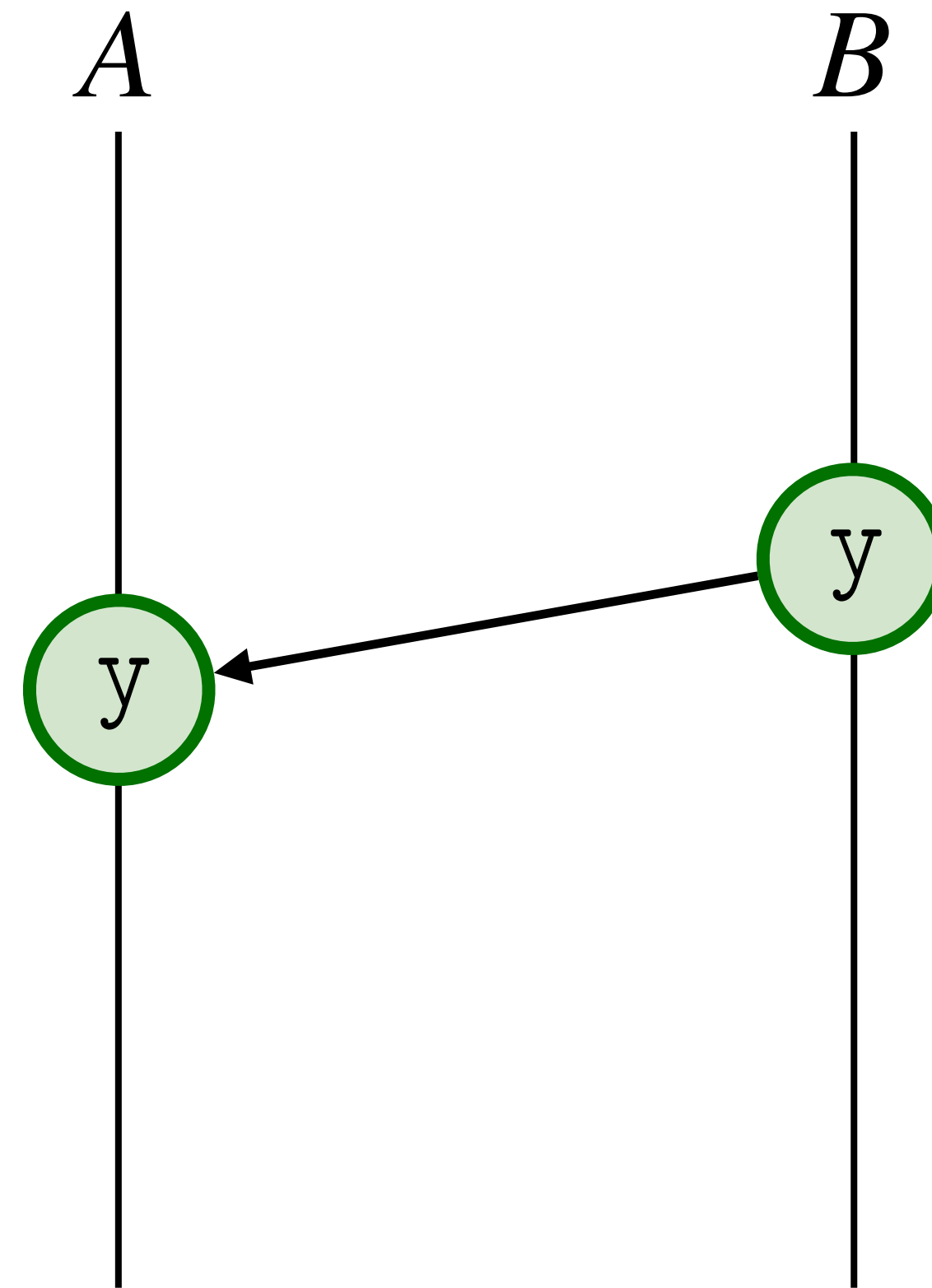
branches



Filtered shuffle semantics

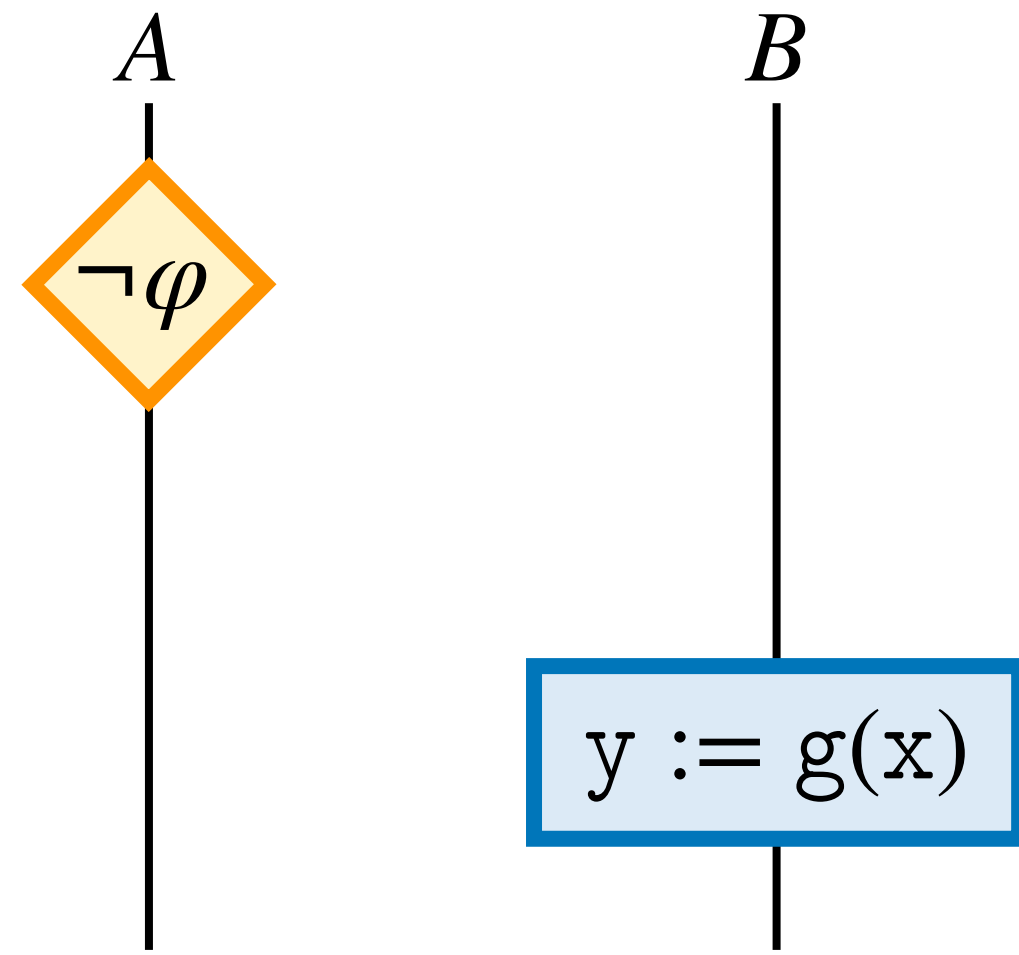
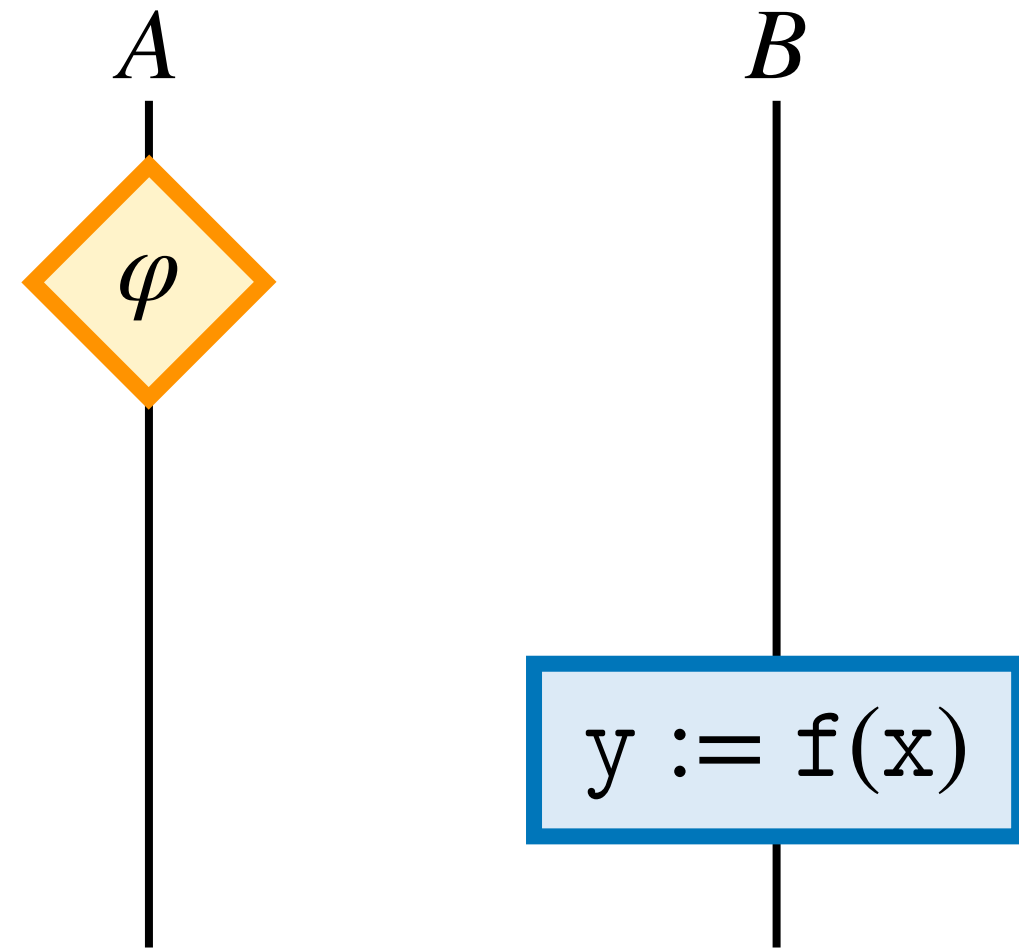


$\equiv$

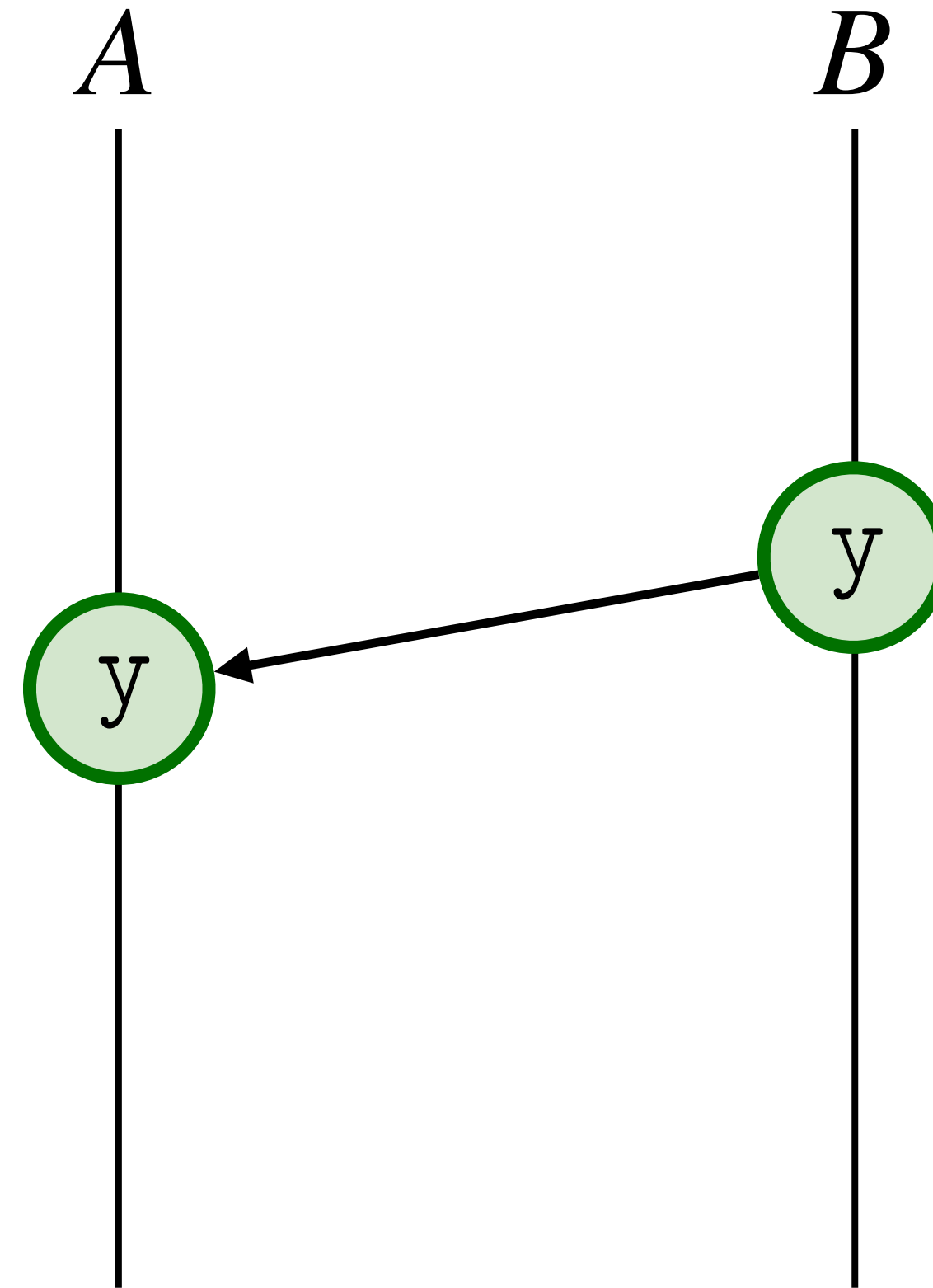


branches

Filtered shuffle semantics

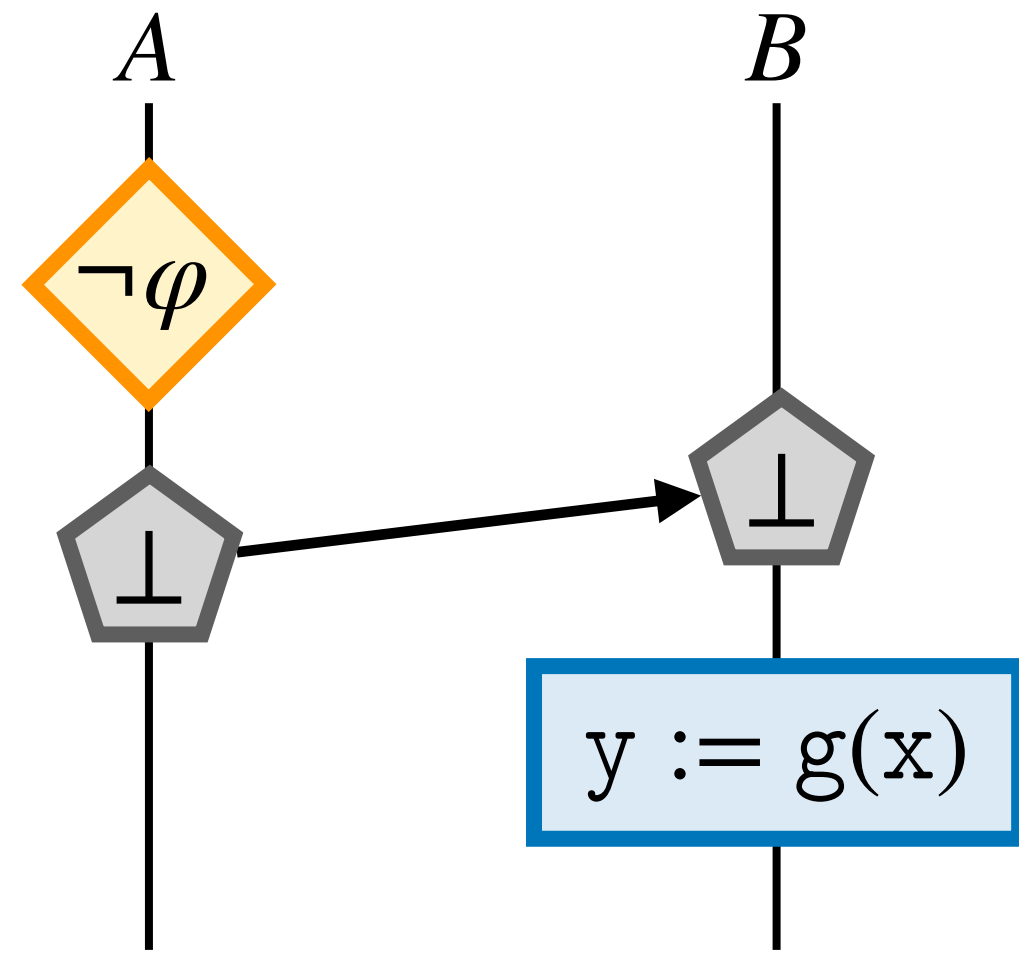
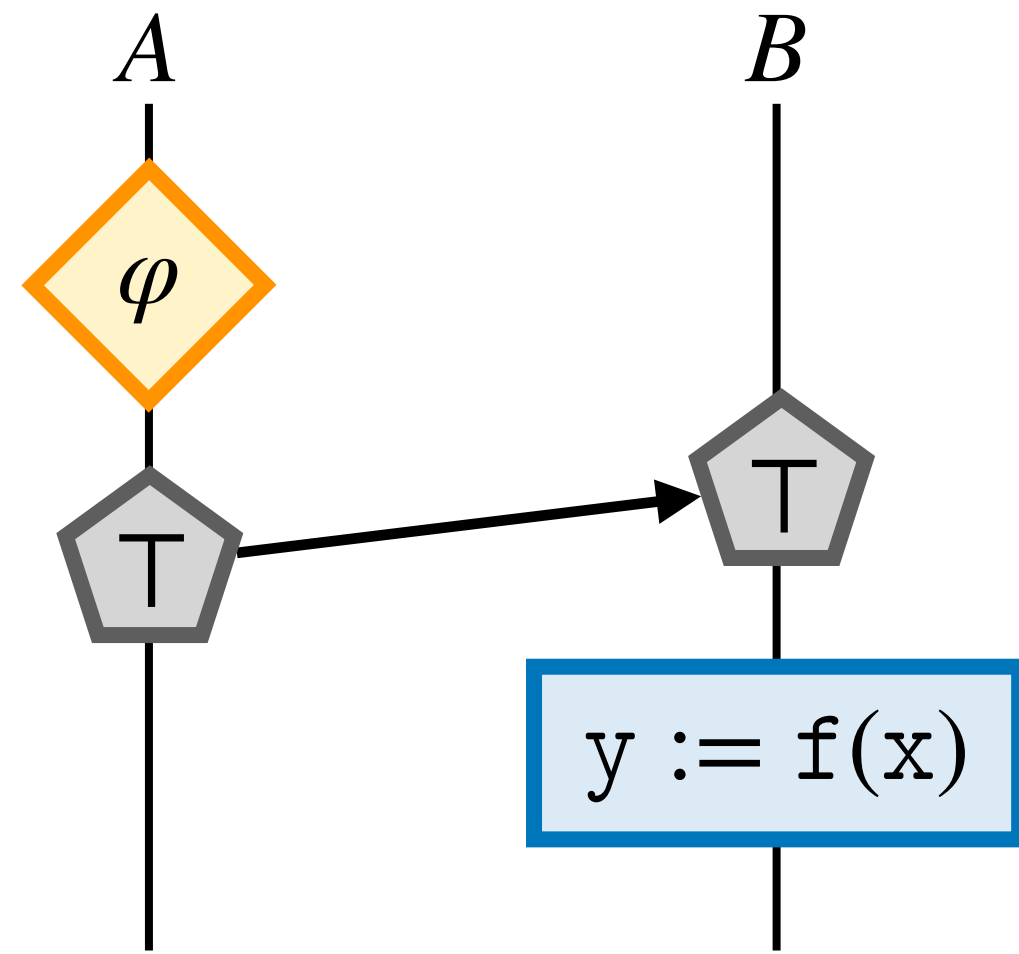


$\equiv$

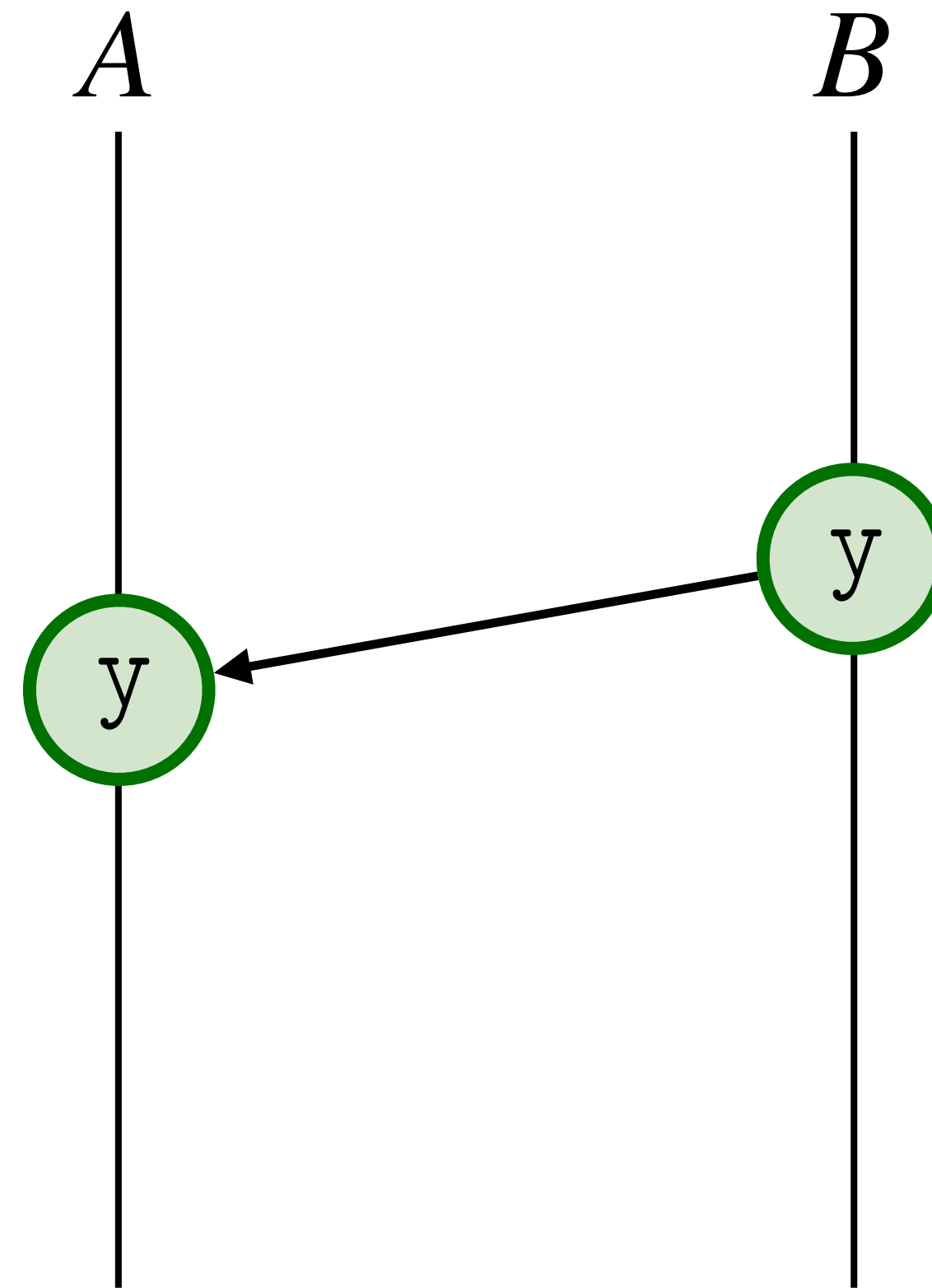


branches
↓
enriched semantics

Filtered shuffle semantics

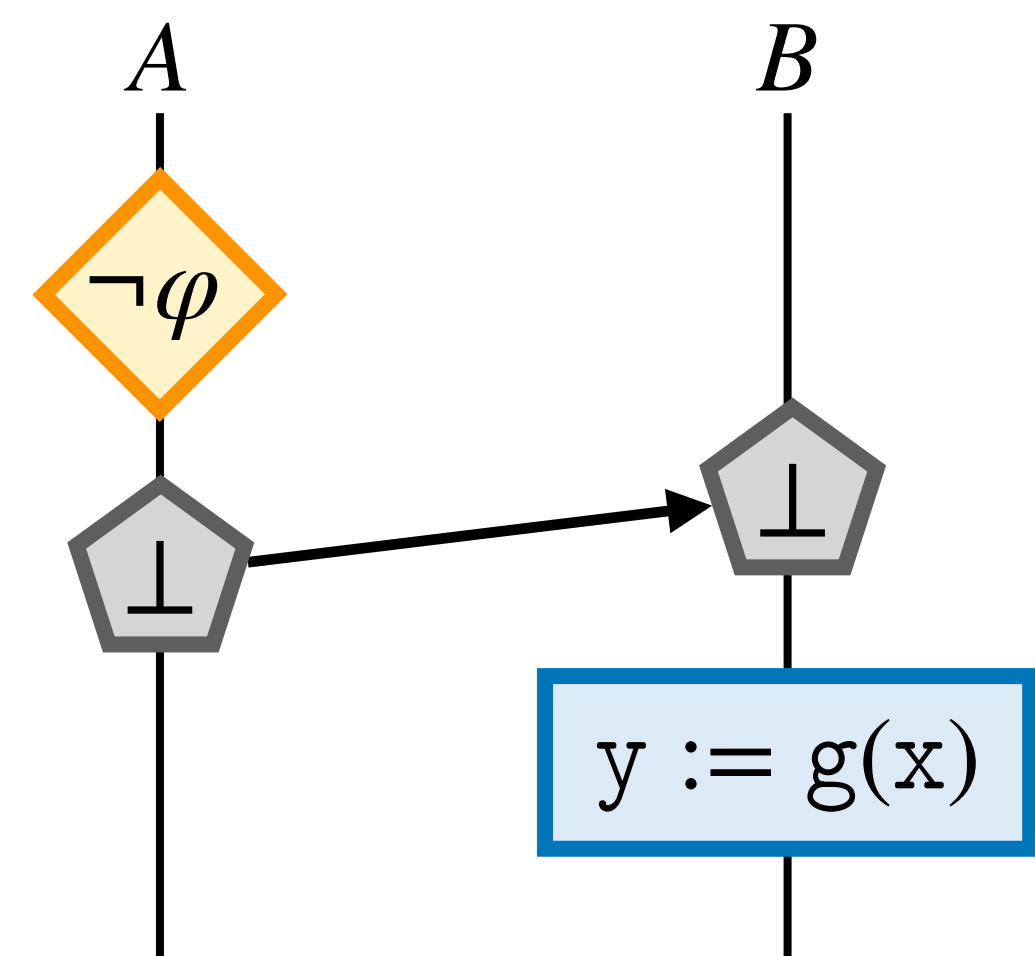
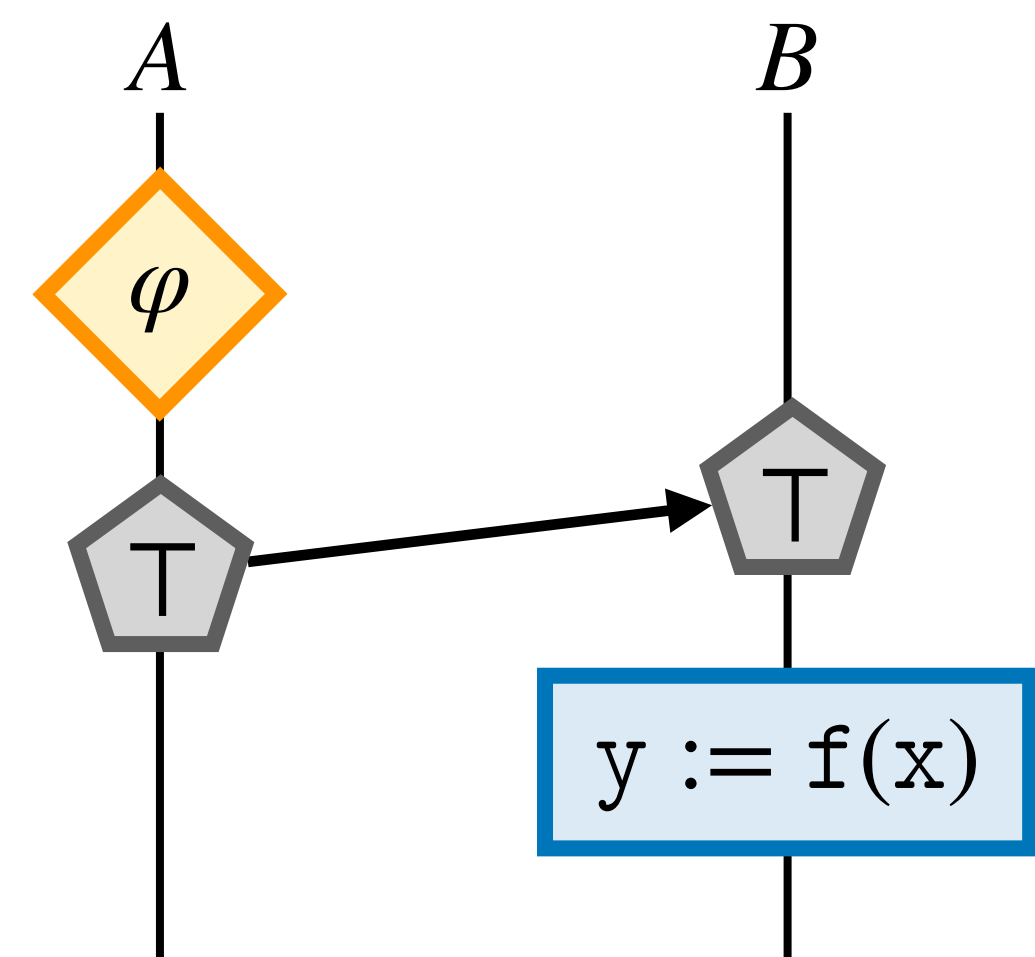


$\equiv$

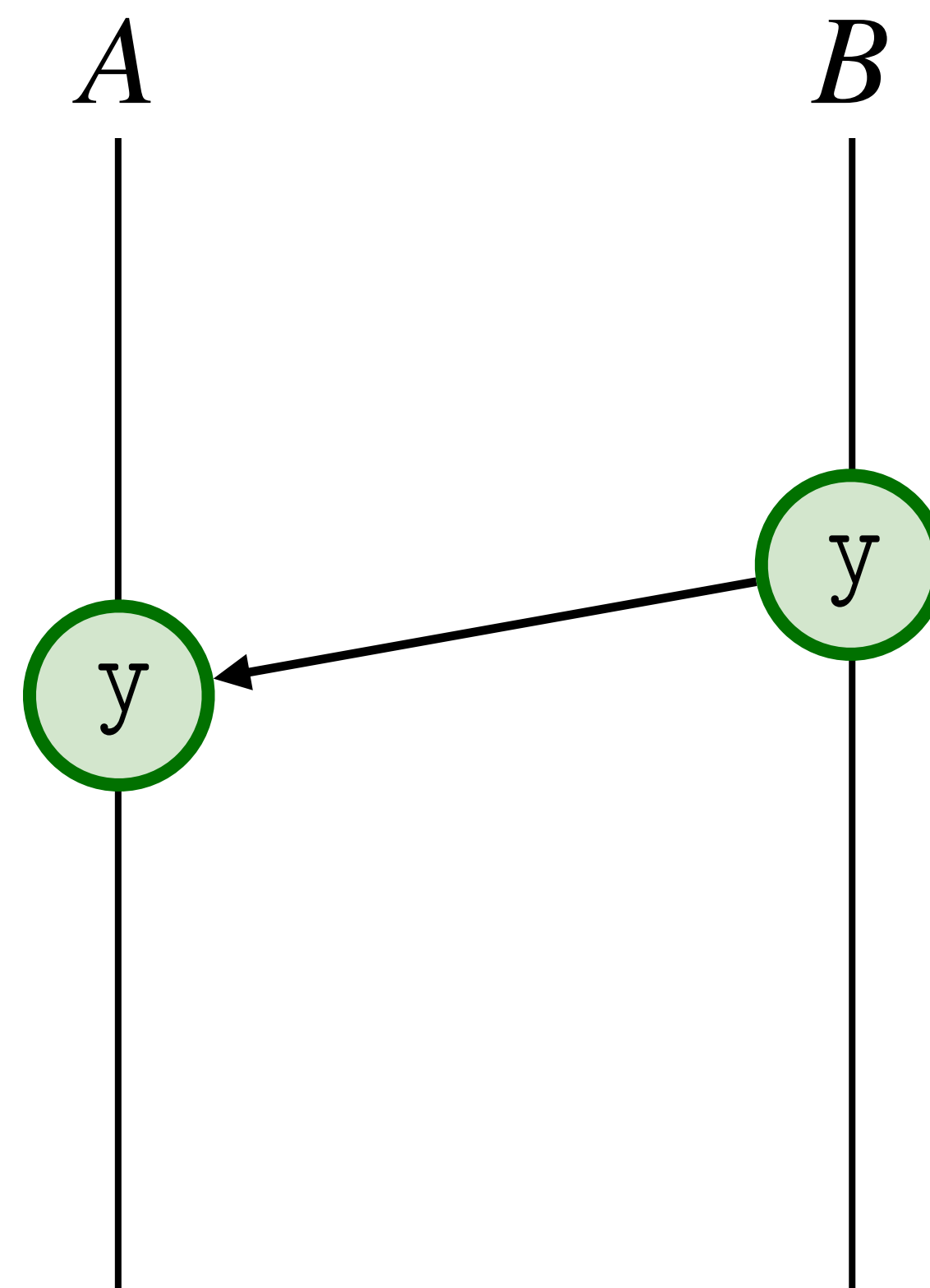


branches
↓
enriched semantics

Filtered shuffle semantics

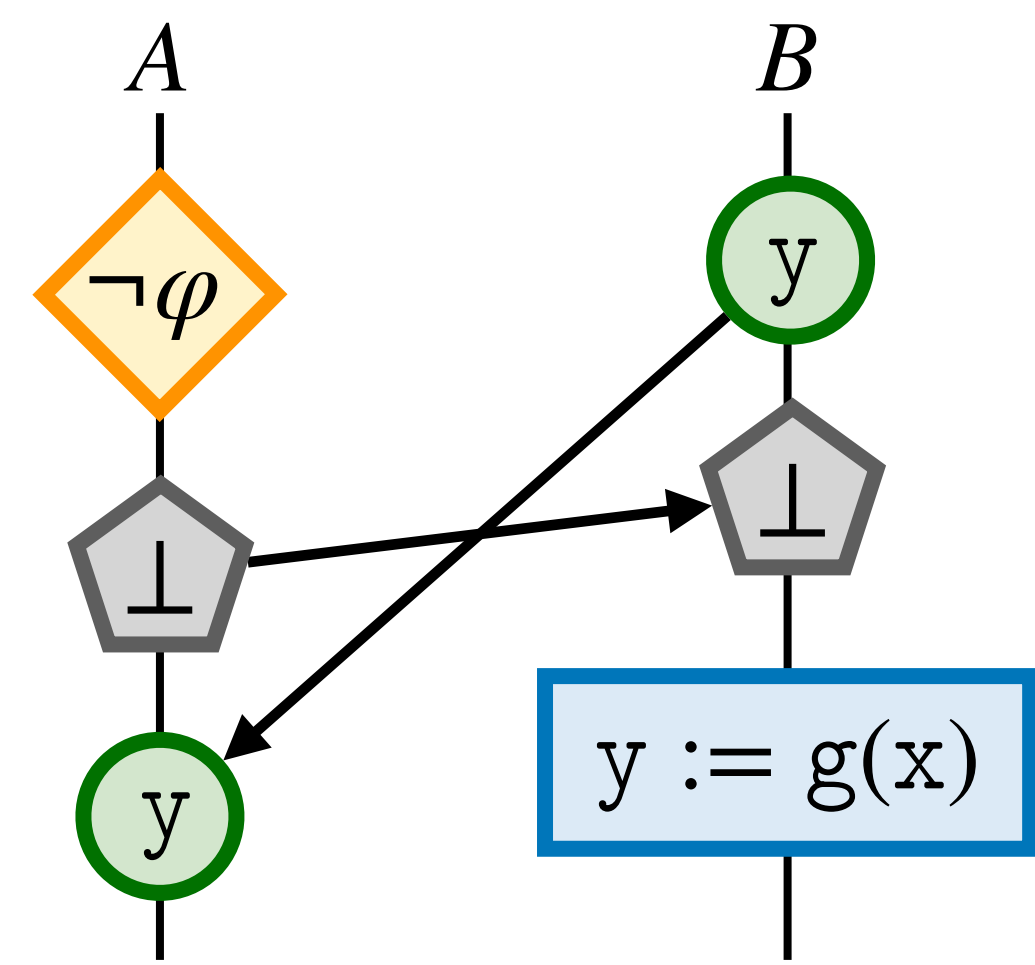
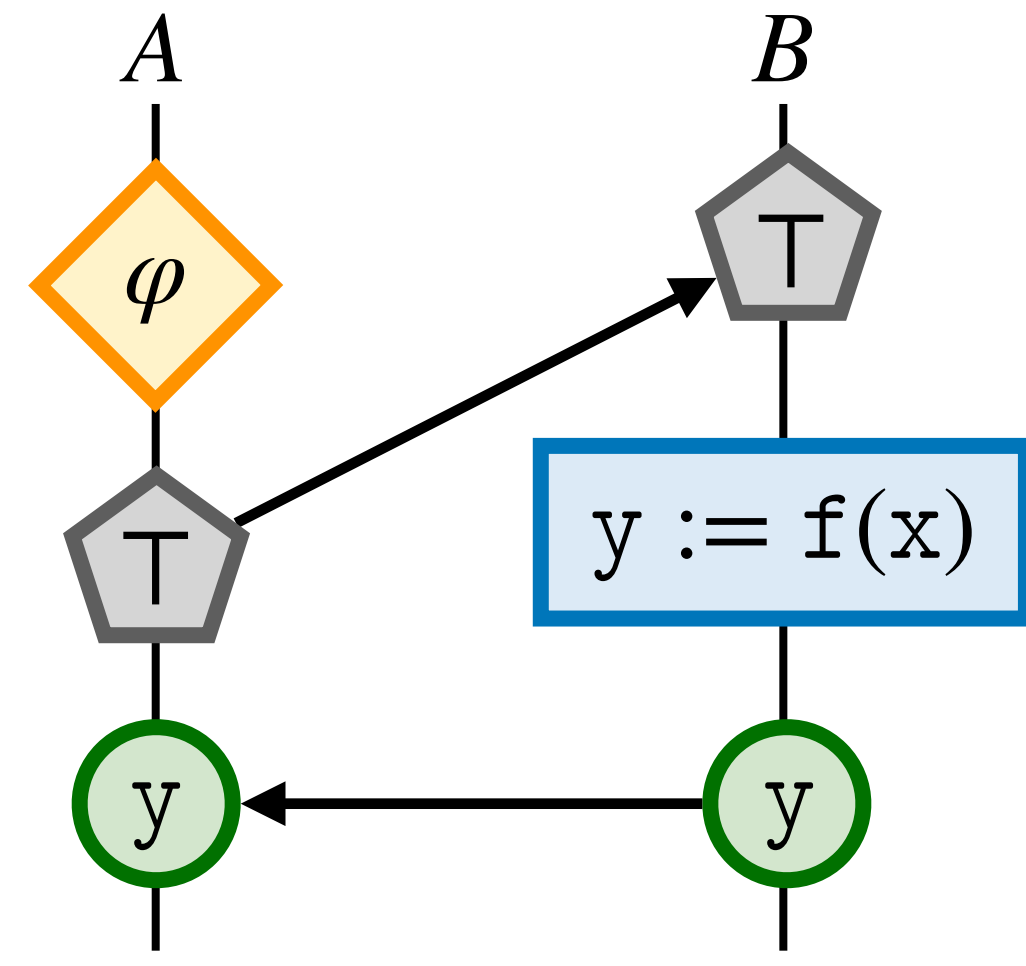
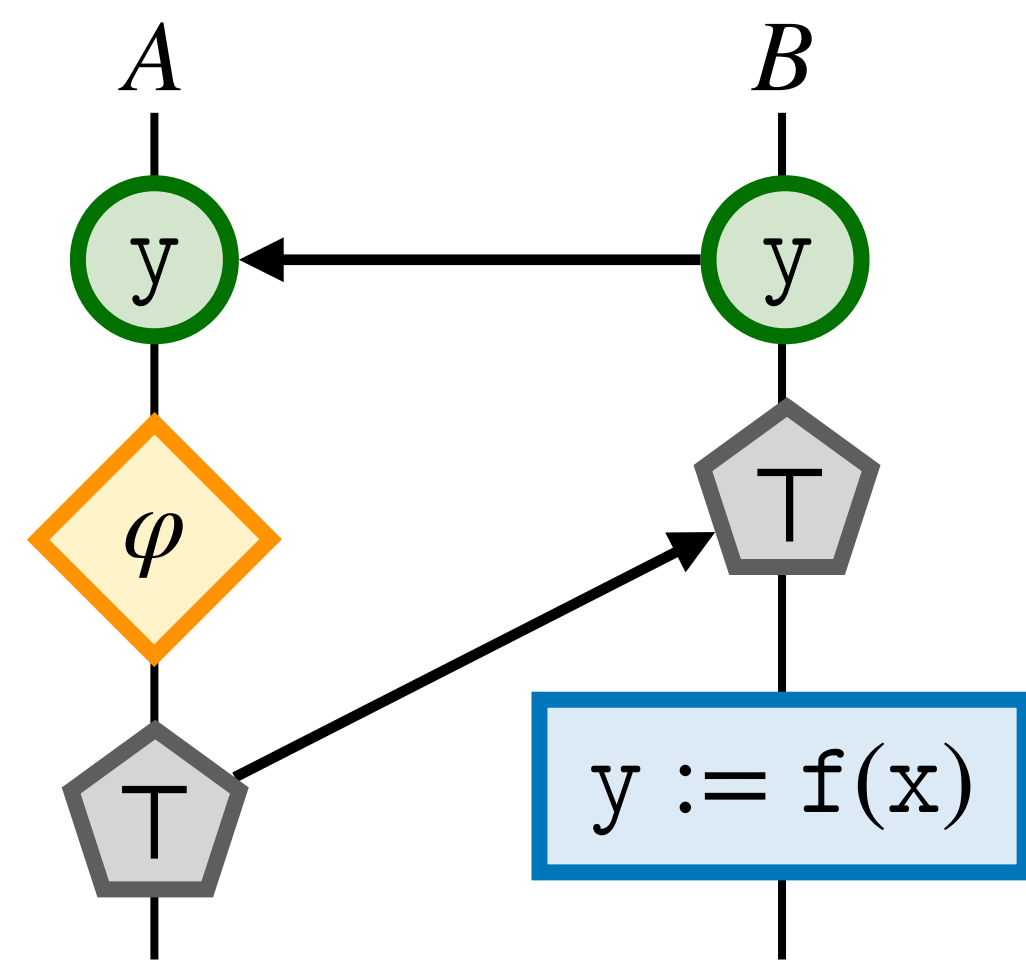
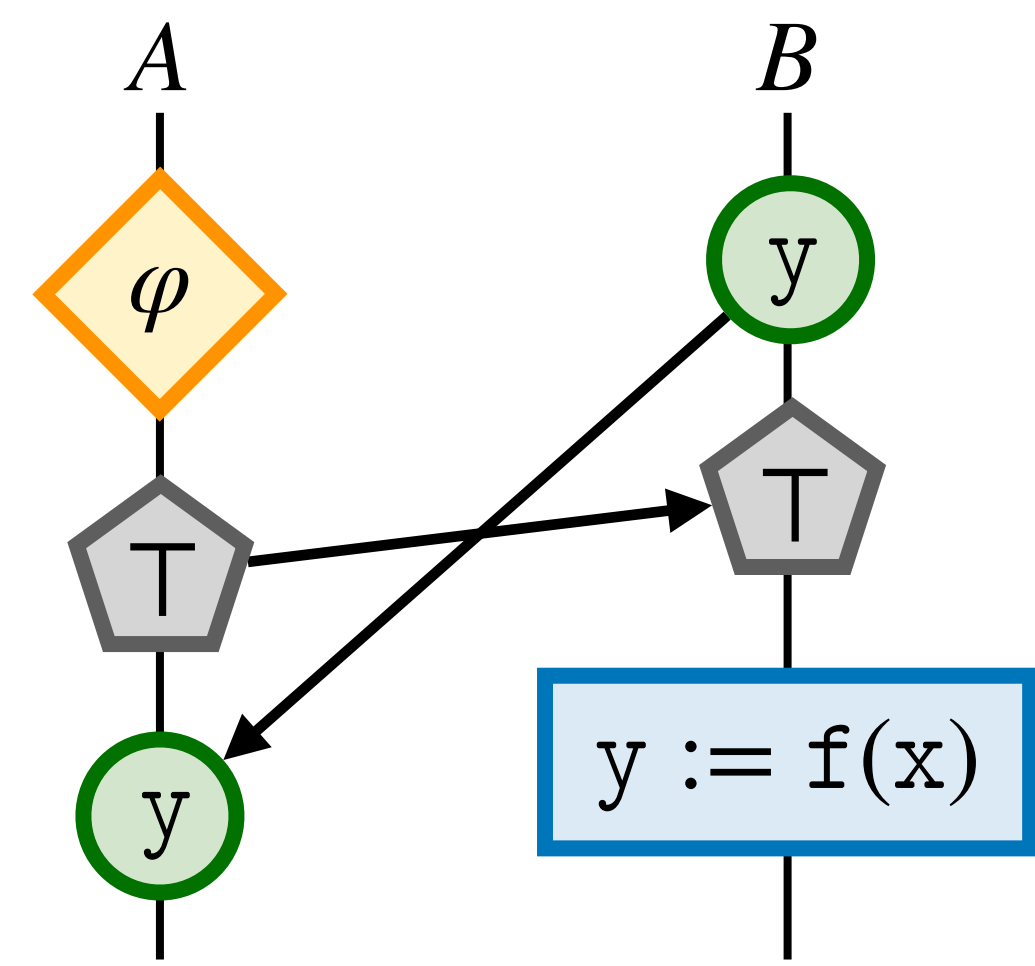


$\equiv$

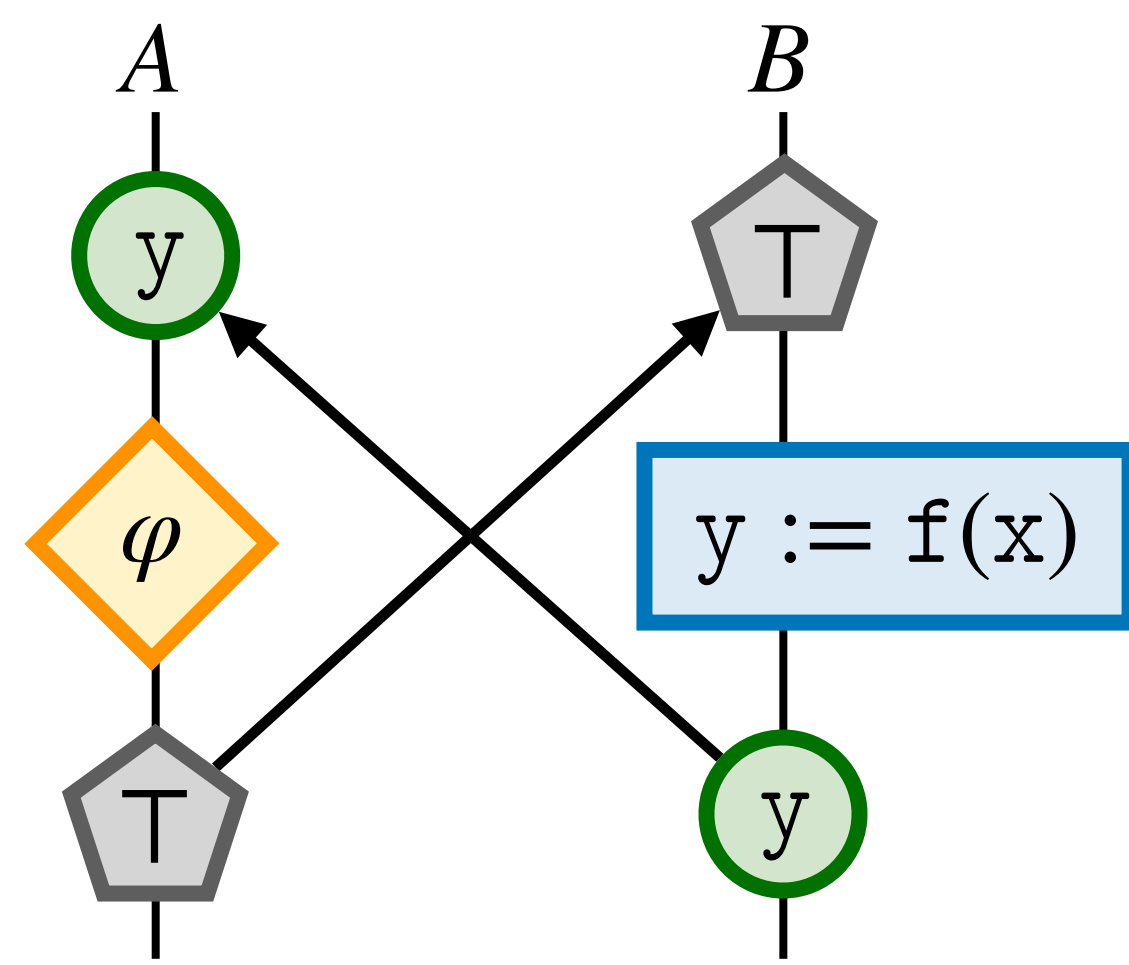


branches
↓
enriched semantics
↓
shuffle

Filtered shuffle semantics

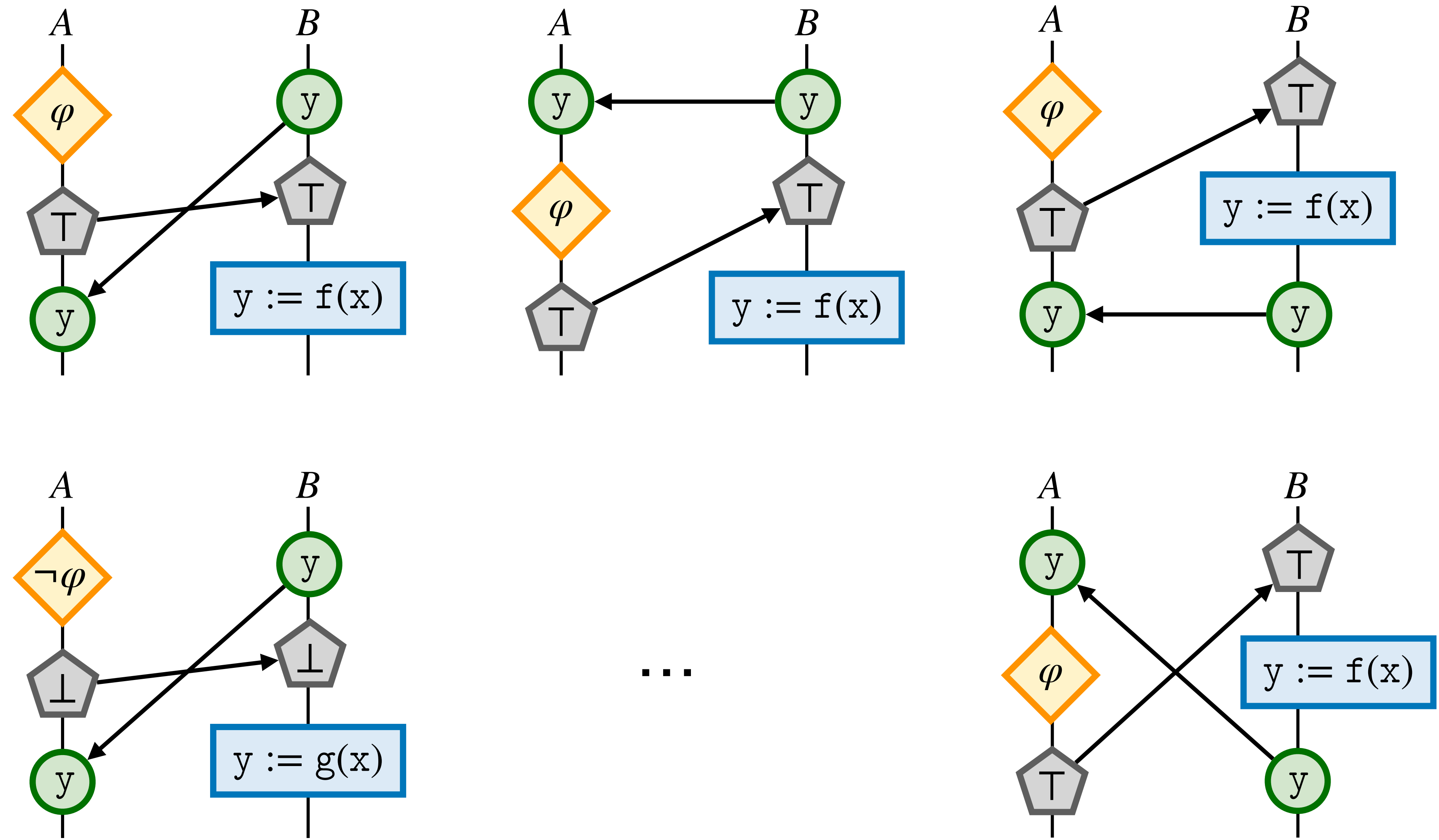


...



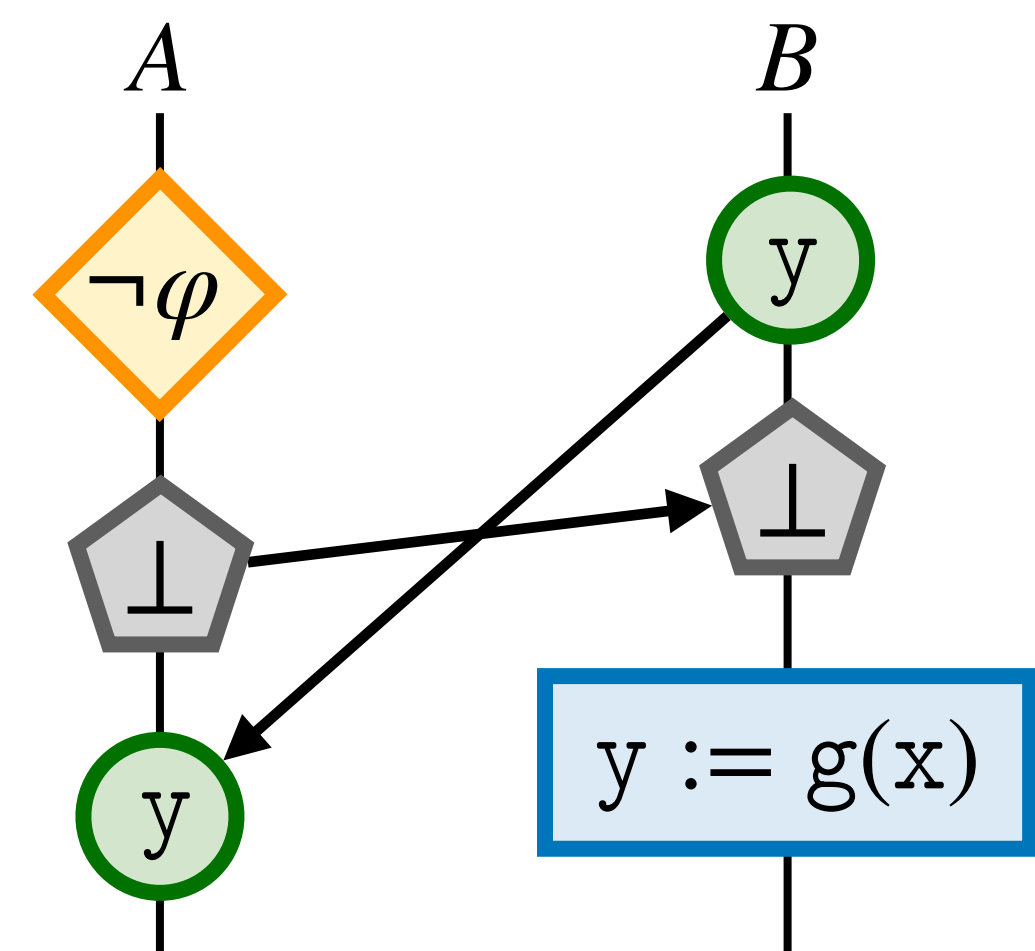
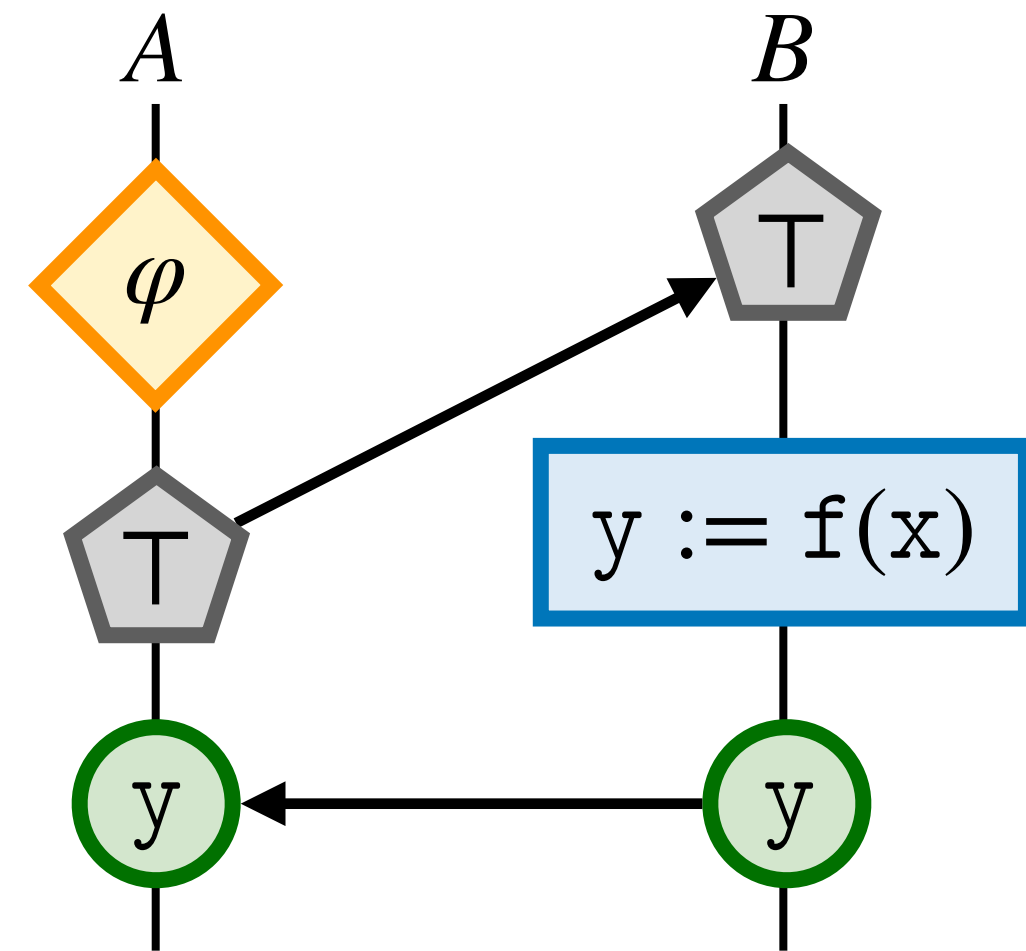
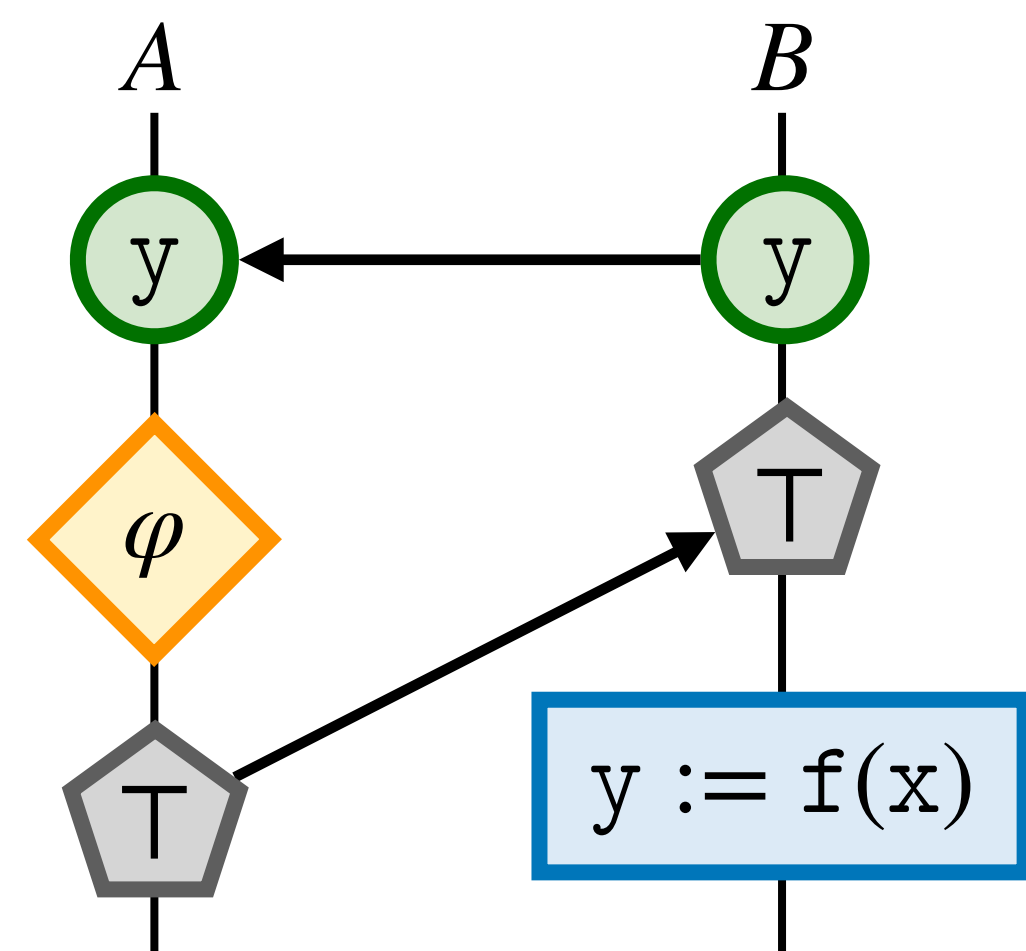
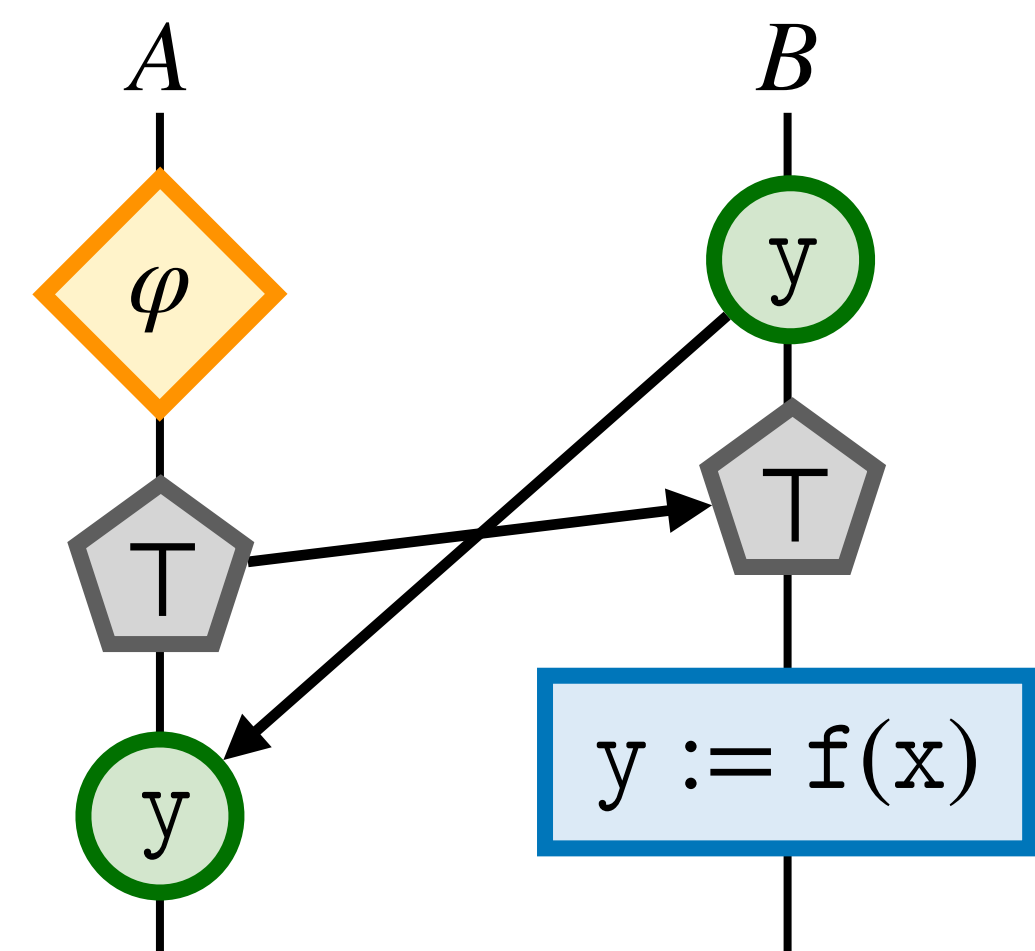
branches
↓
enriched semantics
↓
shuffle

Filtered shuffle semantics

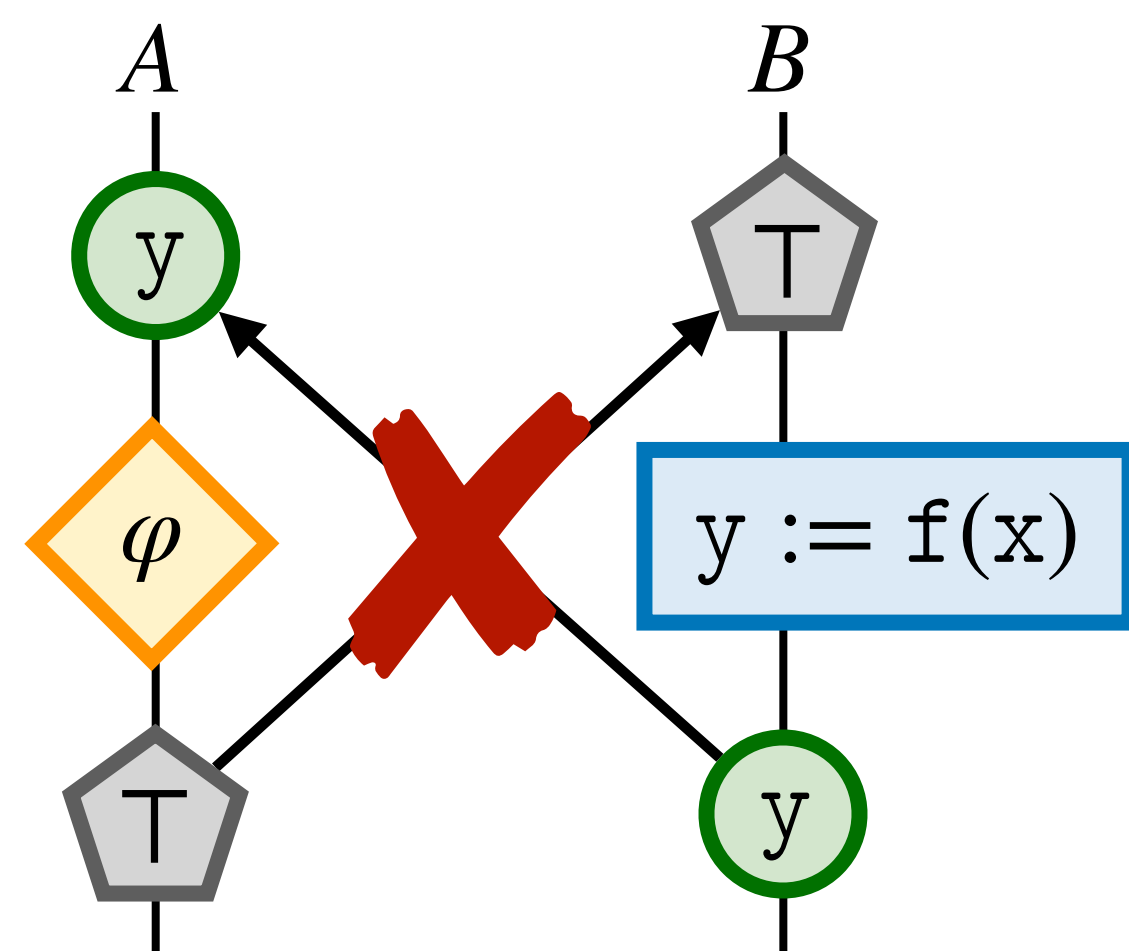


branches
↓
enriched semantics
↓
shuffle
↓
MSC filter

Filtered shuffle semantics

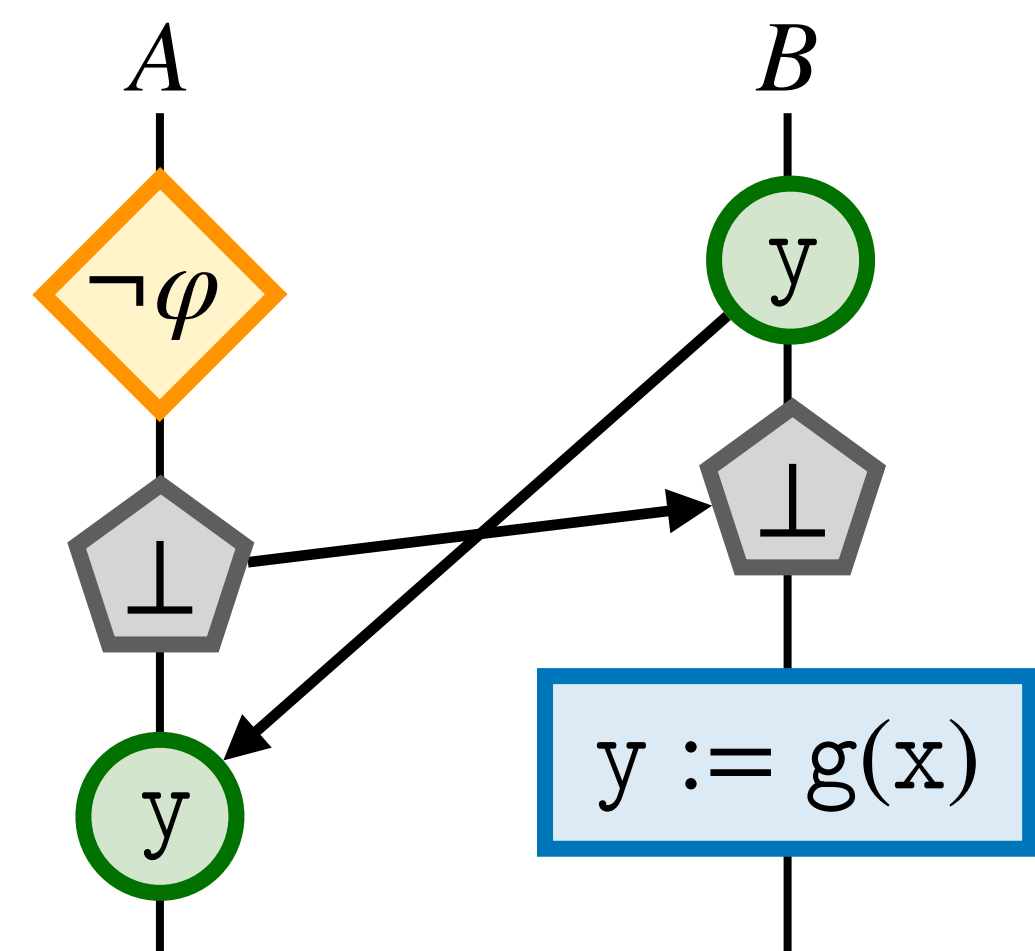
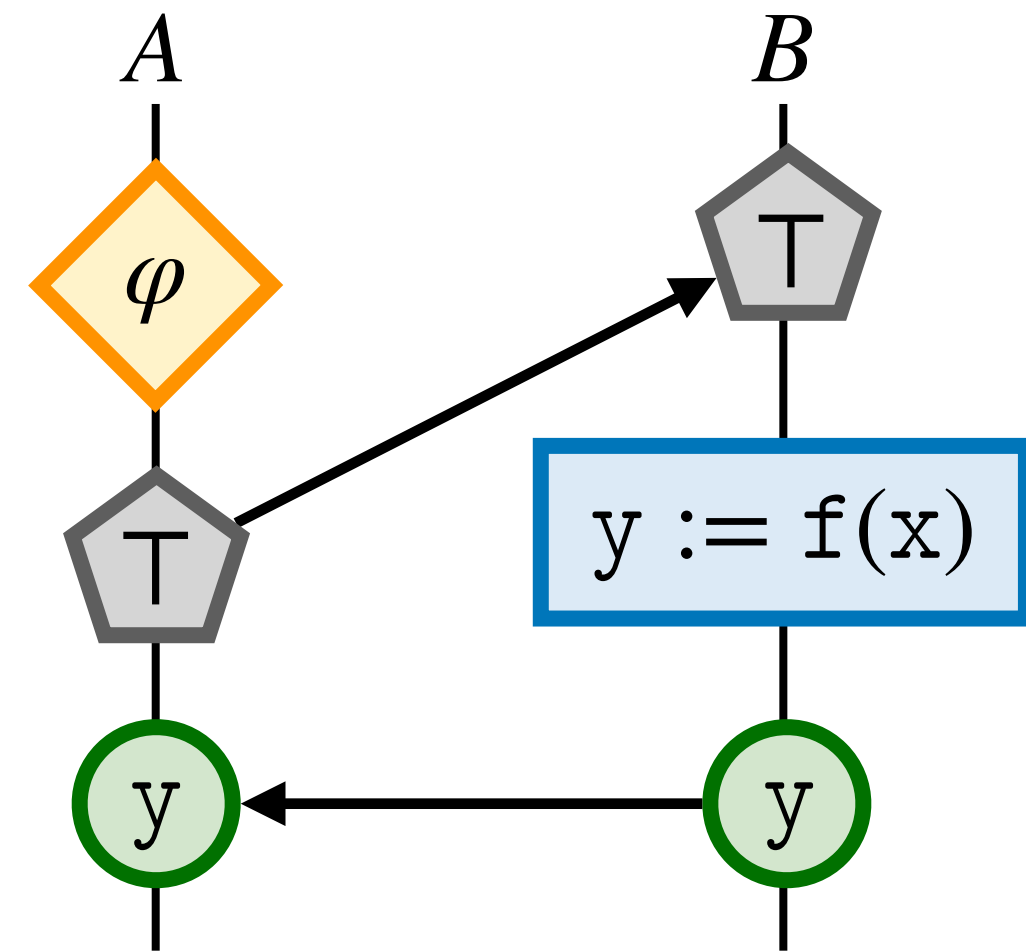
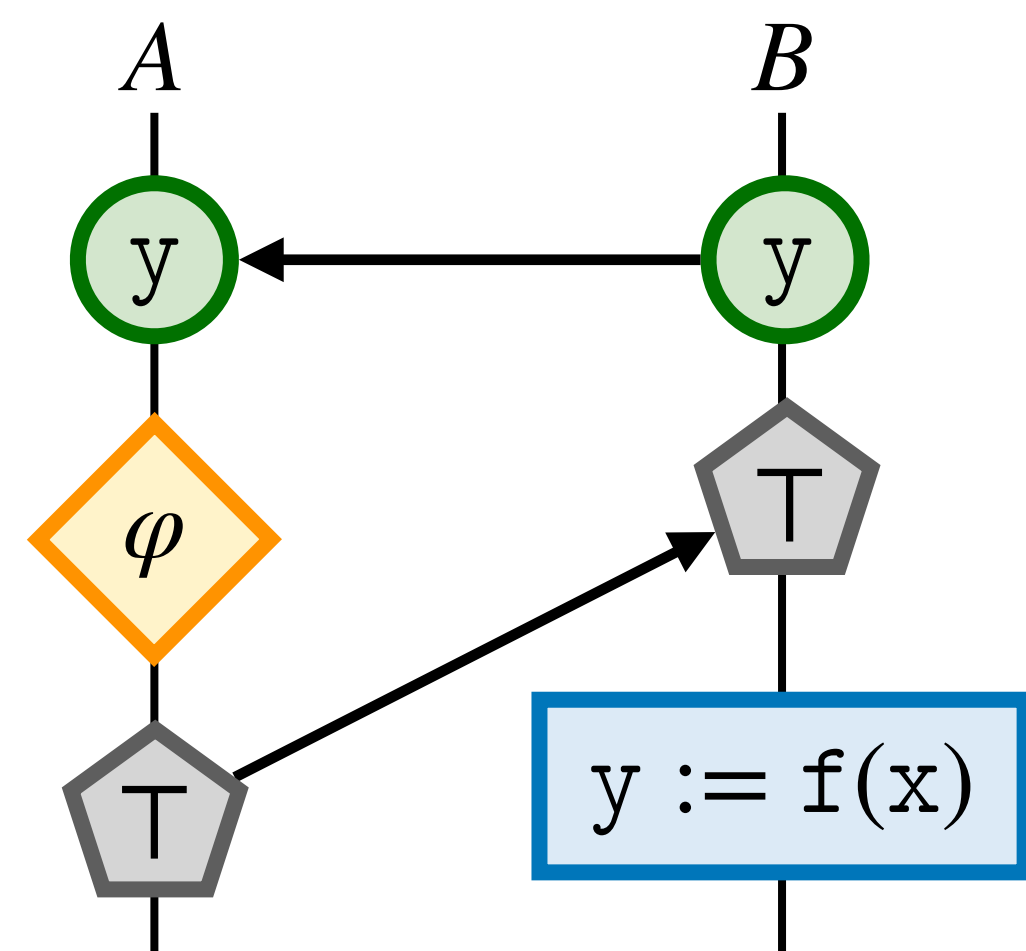
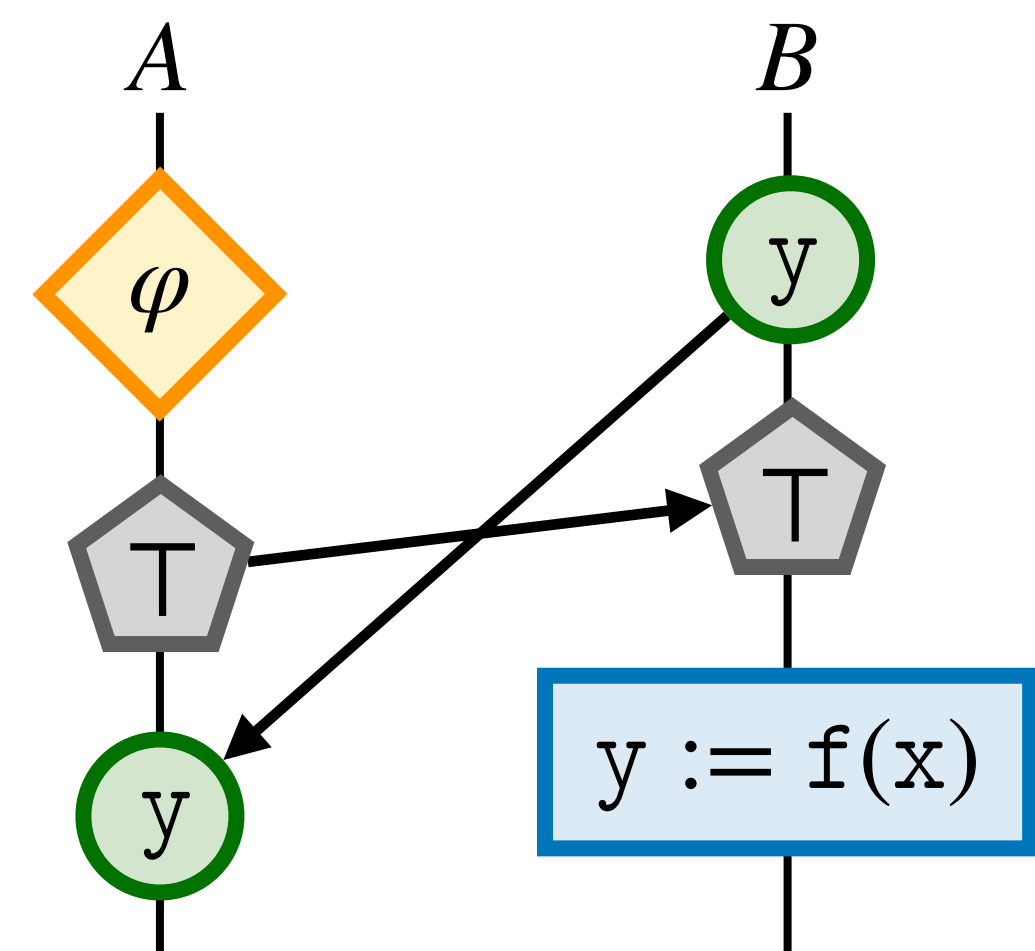


...

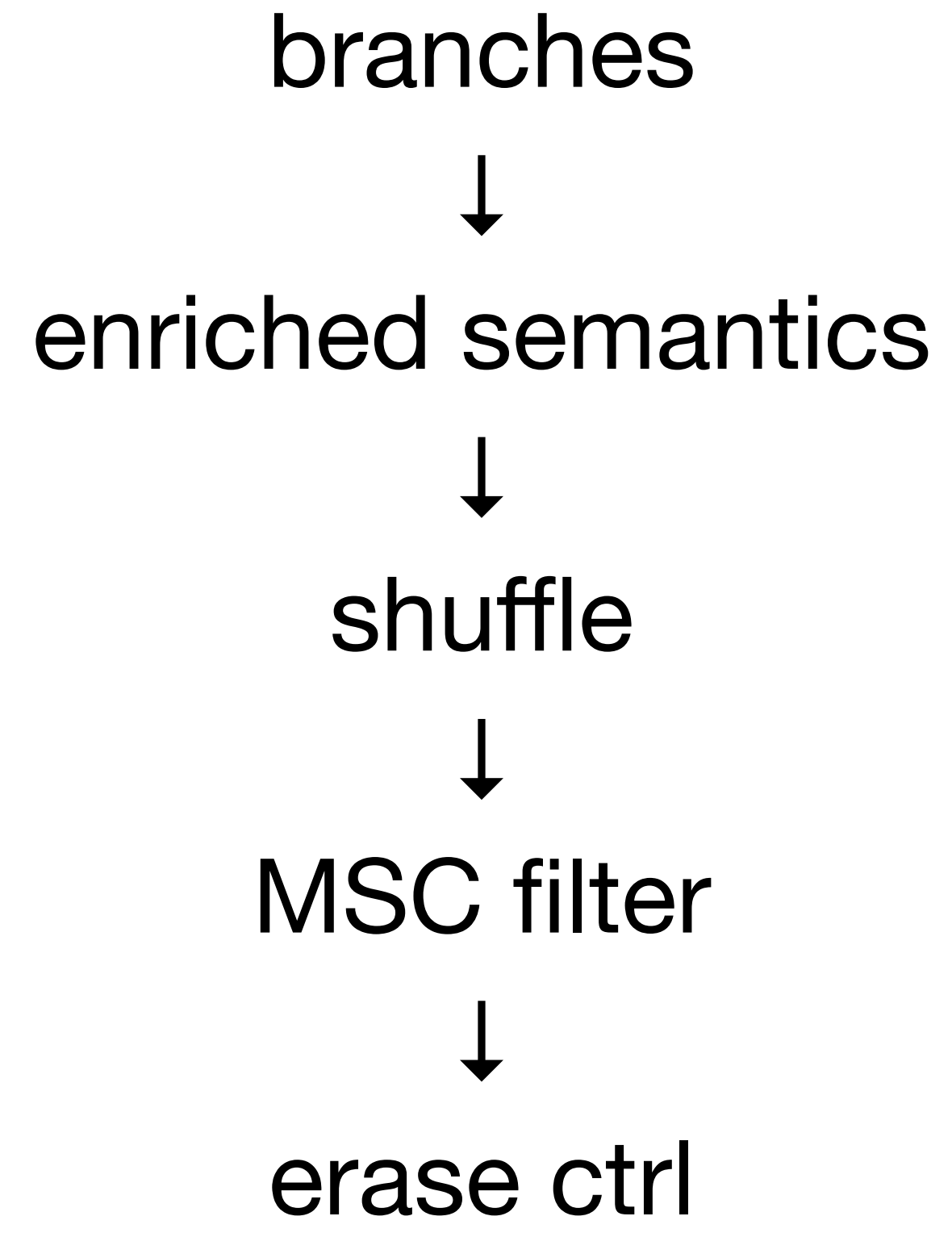


branches
↓
enriched semantics
↓
shuffle
↓
MSC filter

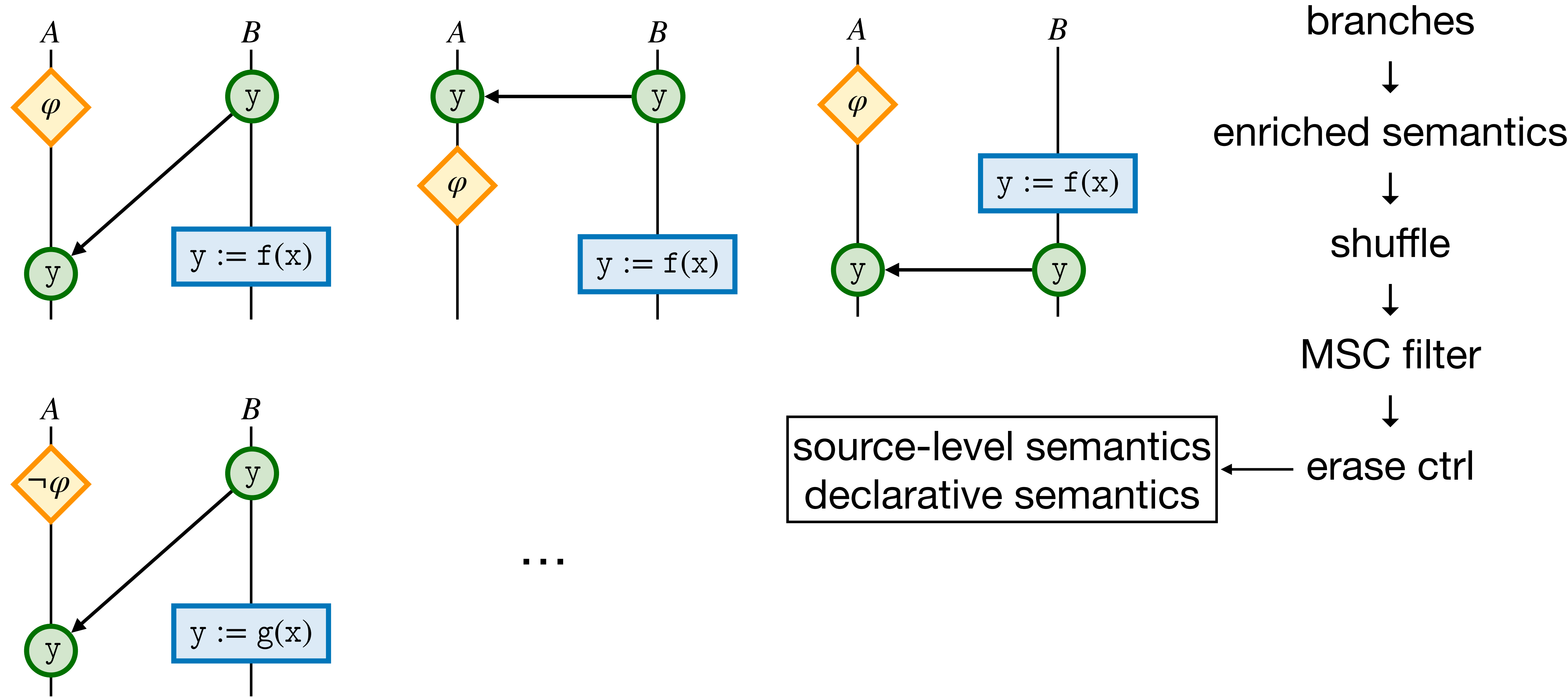
Filtered shuffle semantics



...



Filtered shuffle semantics



Correctness property

Theorem

Correctness:

Projected agents preserve source-level semantics.

Deadlock-freedom:

Every partial projected run can be completed.

Correctness property

Theorem

Correctness:

Projected agents preserve source-level semantics.

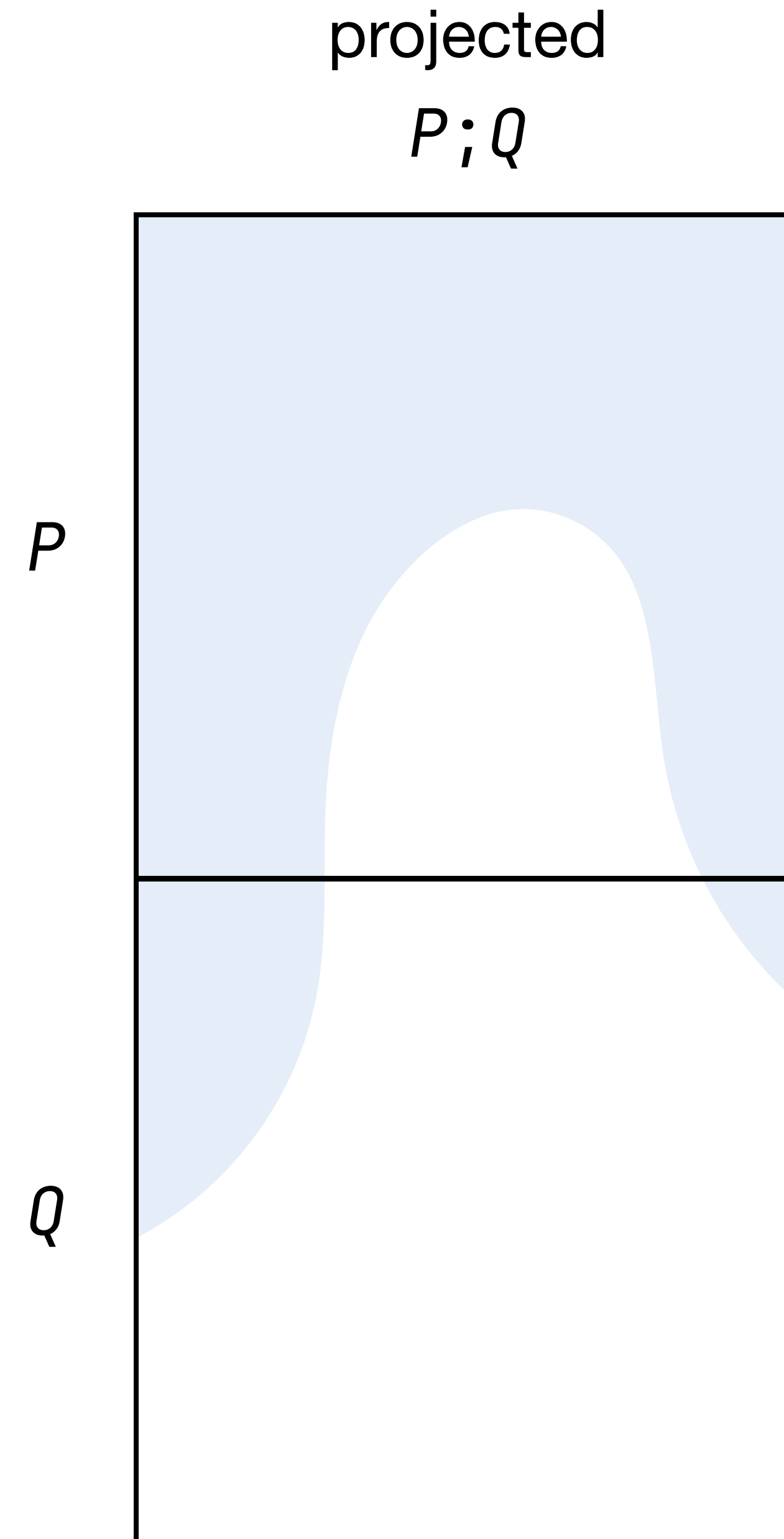
Deadlock-freedom:

Every partial projected run can be completed.

Proof idea: Zipper Lemma

For $P; Q$, projection is **compositional**.

Parallel case: restrict to branch traces, complete branchwise, reassemble (branch-local channels prevent cross-branch FIFO matching).



Correctness property

Theorem

Correctness:

Projected agents preserve source-level semantics.

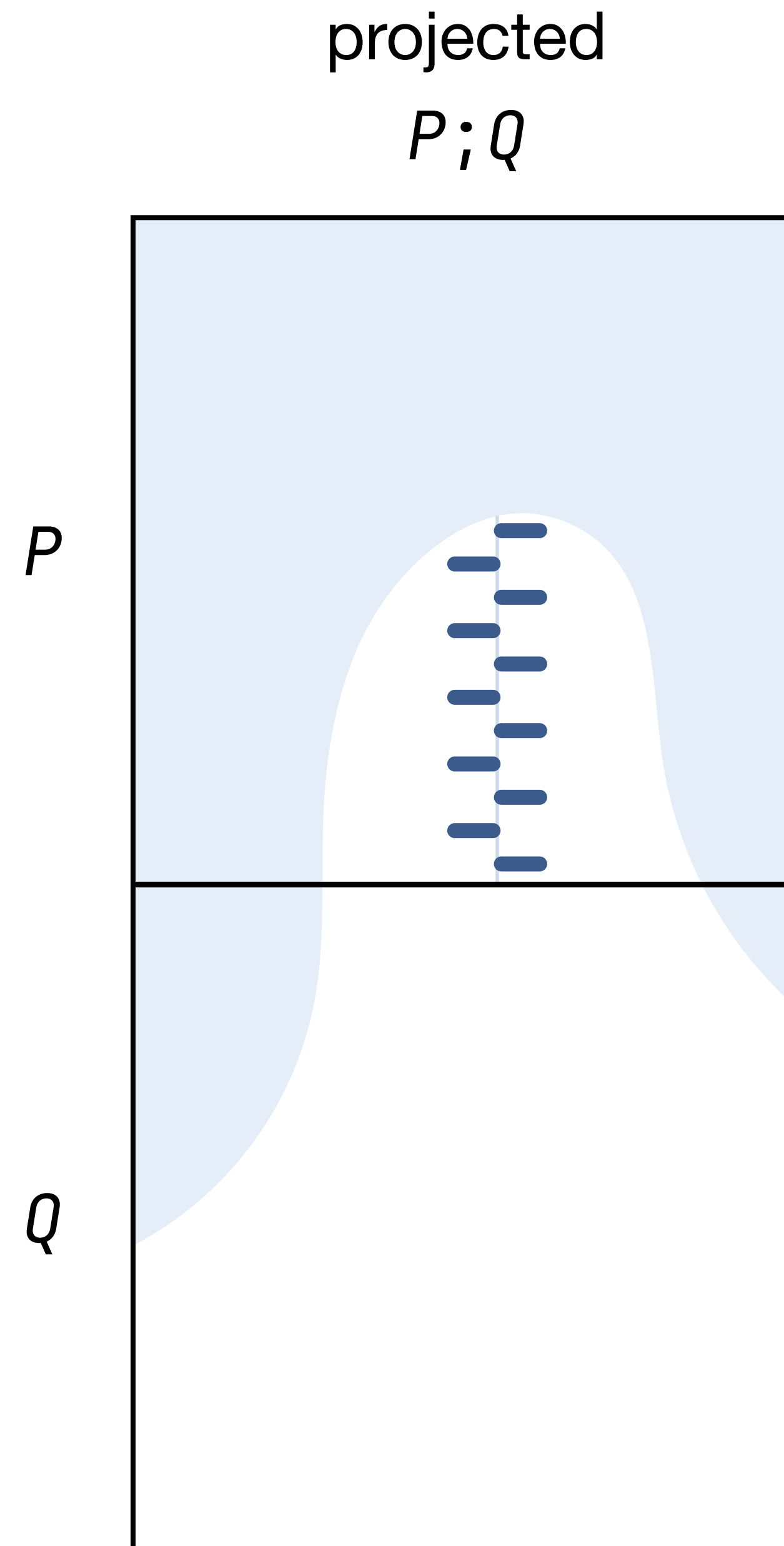
Deadlock-freedom:

Every partial projected run can be completed.

Proof idea: Zipper Lemma

For $P; Q$, projection is **compositional**.

Parallel case: restrict to branch traces, complete branchwise, reassemble (branch-local channels prevent cross-branch FIFO matching).



Future work and tool support

- Dynamic spawning and aggregation of subworkflows.
- Persistent memory for long-lived LLM agents.
- Learning from feedback across concurrent interactions.

Future work and tool support

- Dynamic spawning and aggregation of subworkflows.
- Persistent memory for long-lived LLM agents.
- Learning from feedback across concurrent interactions.

The logo for ZipperGen features a stylized 'Z' icon composed of a grid of small squares, followed by the text 'ipperGen' in a bold, sans-serif font.

`zippergen.io`

Python DSL for coordinating AI agents