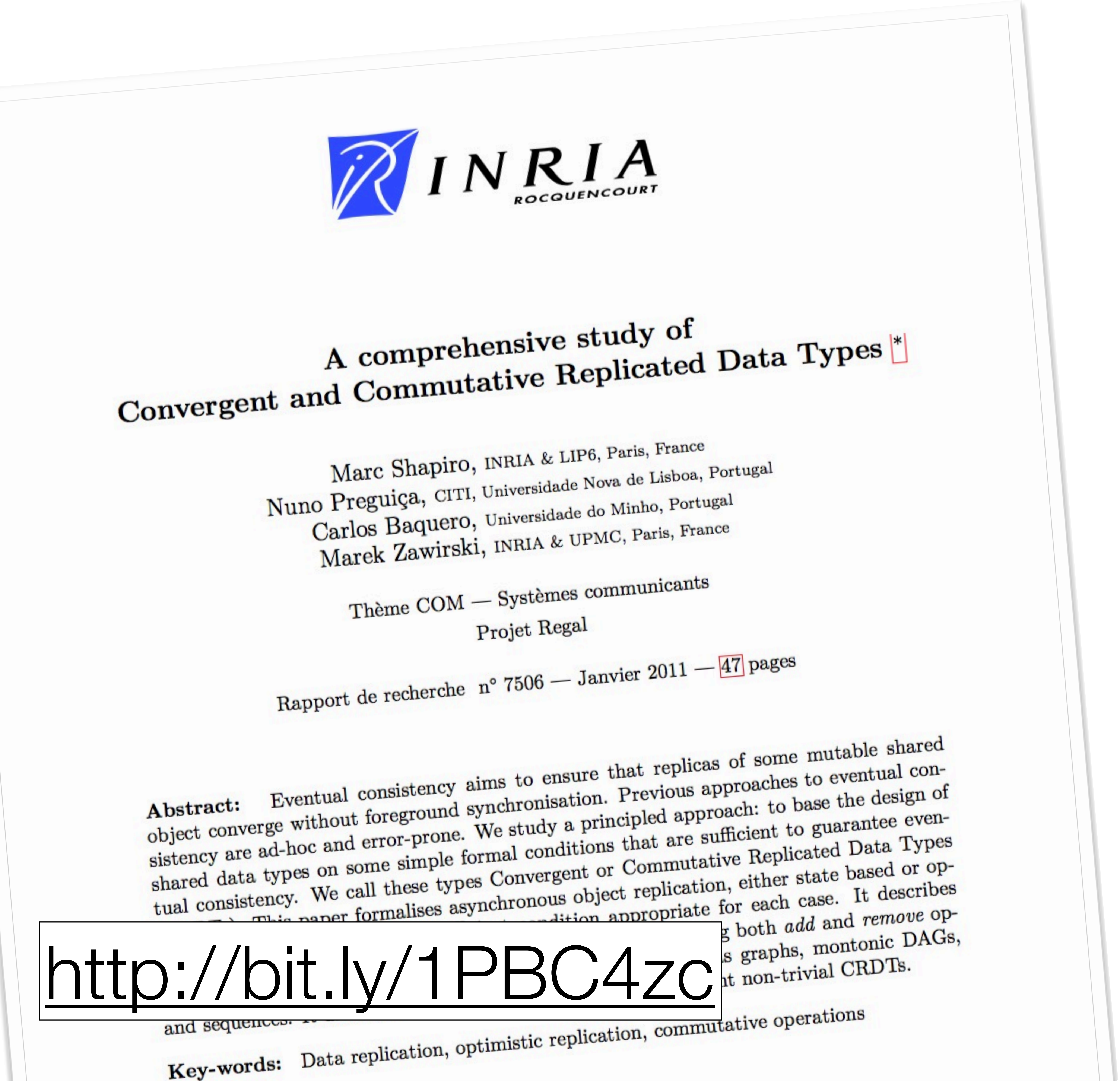
Verifying Implementations of CRDTs

Recommended Reading



INRIA ROCQUENCOURT

A comprehensive study of
Convergent and Commutative Replicated Data Types *

Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal
Marek Zawirski, INRIA & UPMC, Paris, France

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 7506 — Janvier 2011 — 47 pages

Abstract: Eventual consistency aims to ensure that replicas of some mutable shared object converge without foreground synchronisation. Previous approaches to eventual consistency are ad-hoc and error-prone. We study a principled approach: to base the design of shared data types on some simple formal conditions that are sufficient to guarantee eventual consistency. We call these types Convergent or Commutative Replicated Data Types ... This paper formalises asynchronous object replication, either state based or op- ... condition appropriate for each case. It describes ... both *add* and *remove* op- ... graphs, montonic DAGs, ... non-trivial CRDTs.
and sequences. ...

Key-words: Data replication, optimistic replication, commutative operations

http://bit.ly/1PBC4zc

Recommended Reading

INRIA
ROCQUENCOURT

A comprehensive study of Convergent and Commutative Replicated Data Types [*]

Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal
Marek Zawirski, INRIA & UPMC, Paris, France

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 7506 — Janvier 2011 — 47 pages

Abstract: Eventual consistency aims to ensure that replicas of some mutable shared object converge without foreground synchronisation. Previous approaches to eventual consistency are ad-hoc and error-prone. We study a principled approach: to base the design of shared data types on some simple formal conditions that are sufficient to guarantee eventual consistency. We call these types Convergent or Commutative Replicated Data Types ... This paper formalises asynchronous object replication, either state based or op- ... condition appropriate for each case. It describes ... both *add* and *remove* op- ... graphs, montonic DAGs, ... non-trivial CRDTs.
and sequences. ...

Key-words: Data replication, optimistic replication, commutative operations

http://bit.ly/1PBC4zc

'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems

Alexey Gotsman, IMDEA Software Institute, Spain
Hongseok Yang, University of Oxford, UK
Carla Ferreira, NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal
Mahsa Najafzadeh, Sorbonne Universités, Inria, UPMC Univ Paris 06, France
Marc Shapiro, Sorbonne Universités, Inria, UPMC Univ Paris 06, France

Abstract

Large-scale distributed systems often rely on replicated databases that allow a programmer to request different data consistency guarantees for different operations, and thereby control their performance. Using such databases is far from trivial: requesting stronger consistency in too many places may hurt performance, and requesting it in too few places may violate correctness. To help programmers in this task, we propose the first proof rule for establishing that a particular choice of consistency guarantees for various operations on a replicated database is enough to ensure the preservation ... Our rule is modular: it allows ...
tool. We present ...
illustrate its use on several examples.

http://bit.ly/2nM96mT

Categories and Subject Descriptors D.2.4 [*Software Engineering*] ... F.3.1 [*Logics and Meanings* ...

use. Ideally, we would like replicated databases to provide *strong consistency*, i.e., to behave as if a single centralised node handles all operations. However, achieving this ideal usually requires synchronisation among replicas, which slows down the database and even makes it unavailable if network connections between replicas fail [2, 24].

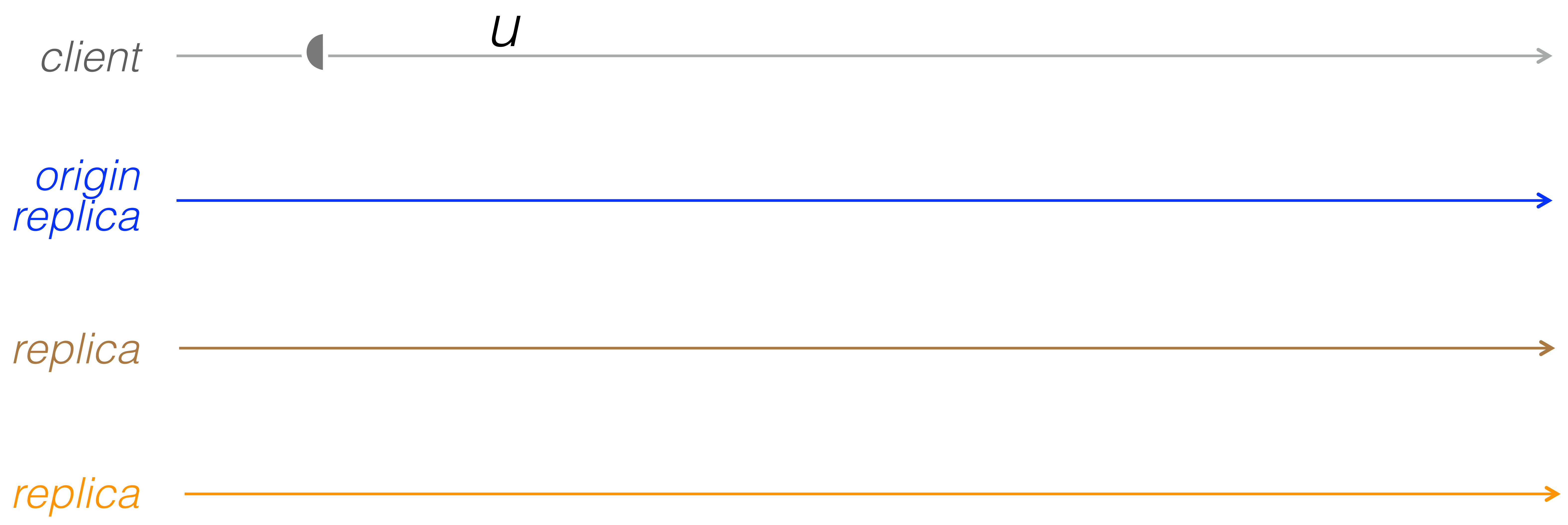
For this reason, modern replicated databases often eschew synchronisation completely; such databases are commonly dubbed *eventually consistent* [47]. In these databases, a replica performs an operation requested by a client locally without any synchronisation with other replicas and immediately returns to the client; the effect of the operation is propagated to the other replicas only *eventually*. ... lead to *anomalies*—behaviours deviating from ... One of them is illustrated in Figure 1(a). Here ... while connected to a replica r_1, and Bob, also ... sees the post and comments on it. After each of the two operations, r_1 sends a message to the other replicas in the system with the update performed by the user. If the messages with the updates by Alice and Bob arrive to a replica r_2 out of order, then Carol, connected to r_2, may end up seeing Bob's comment, ...

Disclaimer:

Slides kindly provided by Marc Shapiro

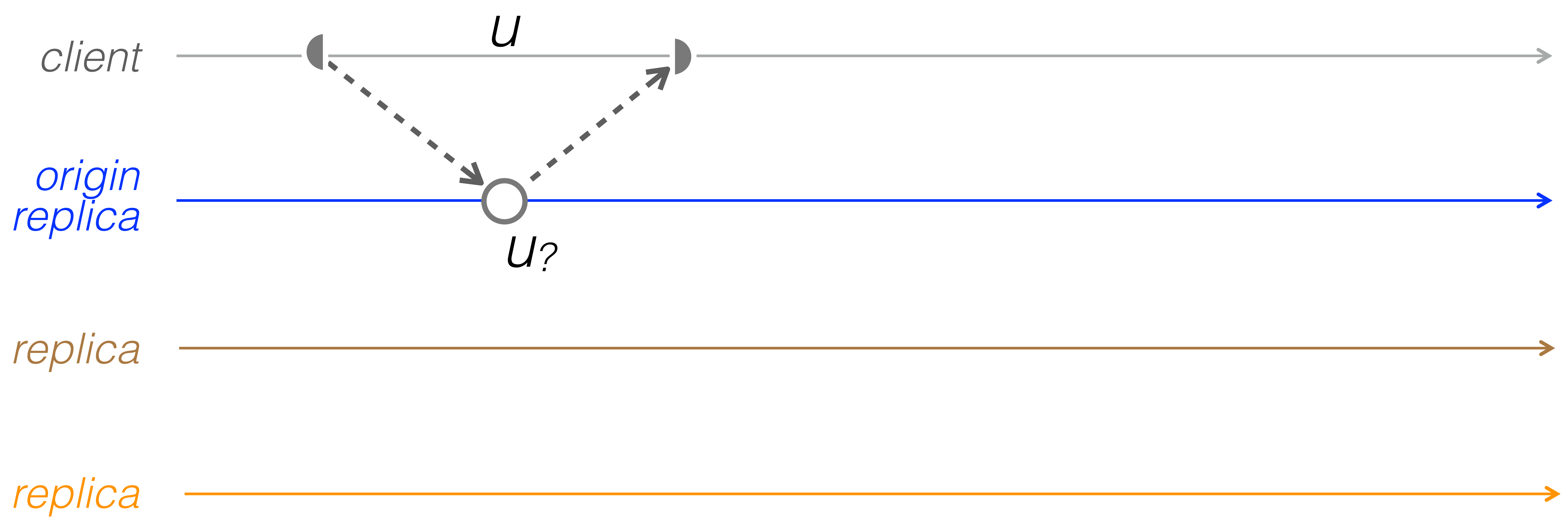
(all errors are mine)

Operation



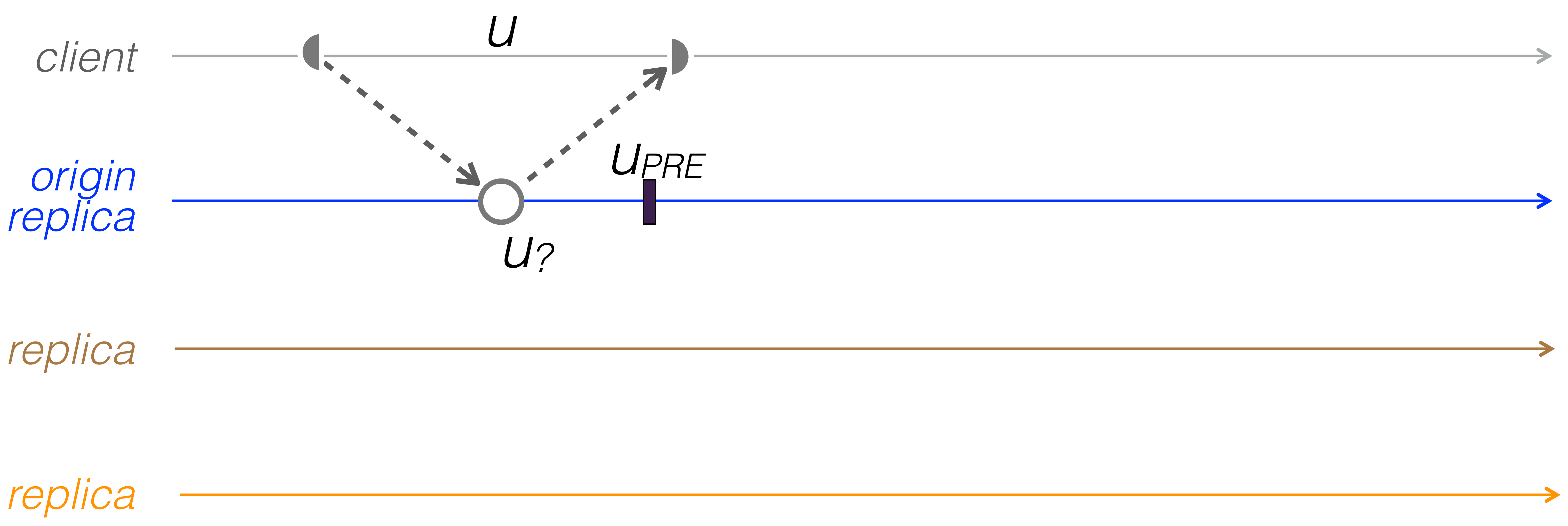
- *u: state ⤻ (retval, (state ⤻ state))*
- Prepare (@origin) $u_?$; deliver $u_!$
- Read one, write all (ROWA)
- Deferred-update replication (DUR)

Operation



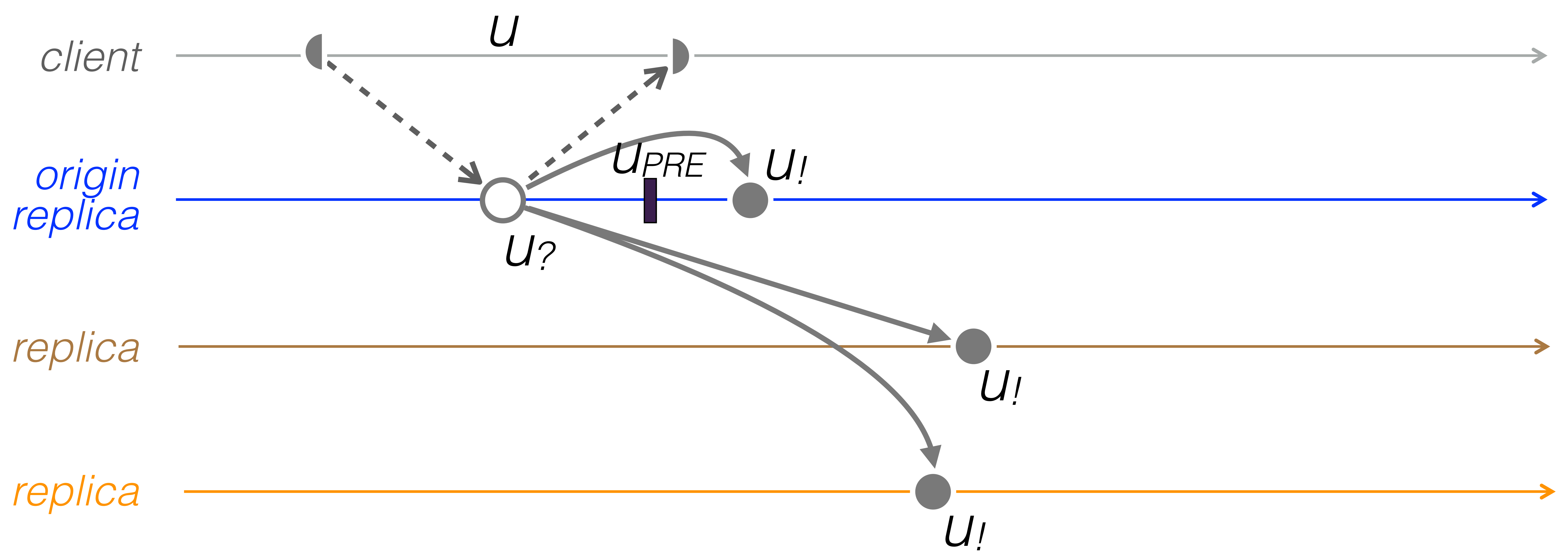
- *u: state ↝ (retval, (state ↝ state))*
- Prepare (@origin) $u_?$; deliver $u_!$
- Read one, write all (ROWA)
- Deferred-update replication (DUR)

Operation



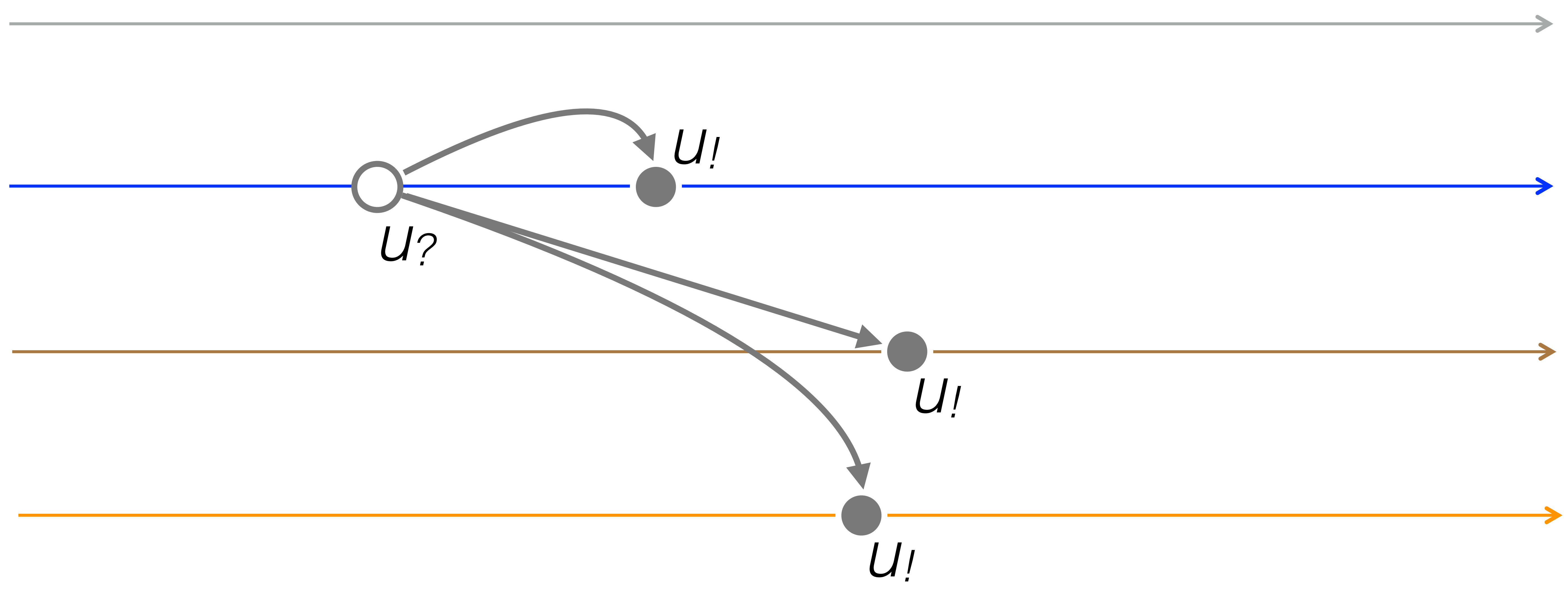
- *u: state ⤻ (retval, (state ⤻ state))*
- Prepare (@origin) $u_?$; deliver $u_!$
- Read one, write all (ROWA)
- Deferred-update replication (DUR)

Operation



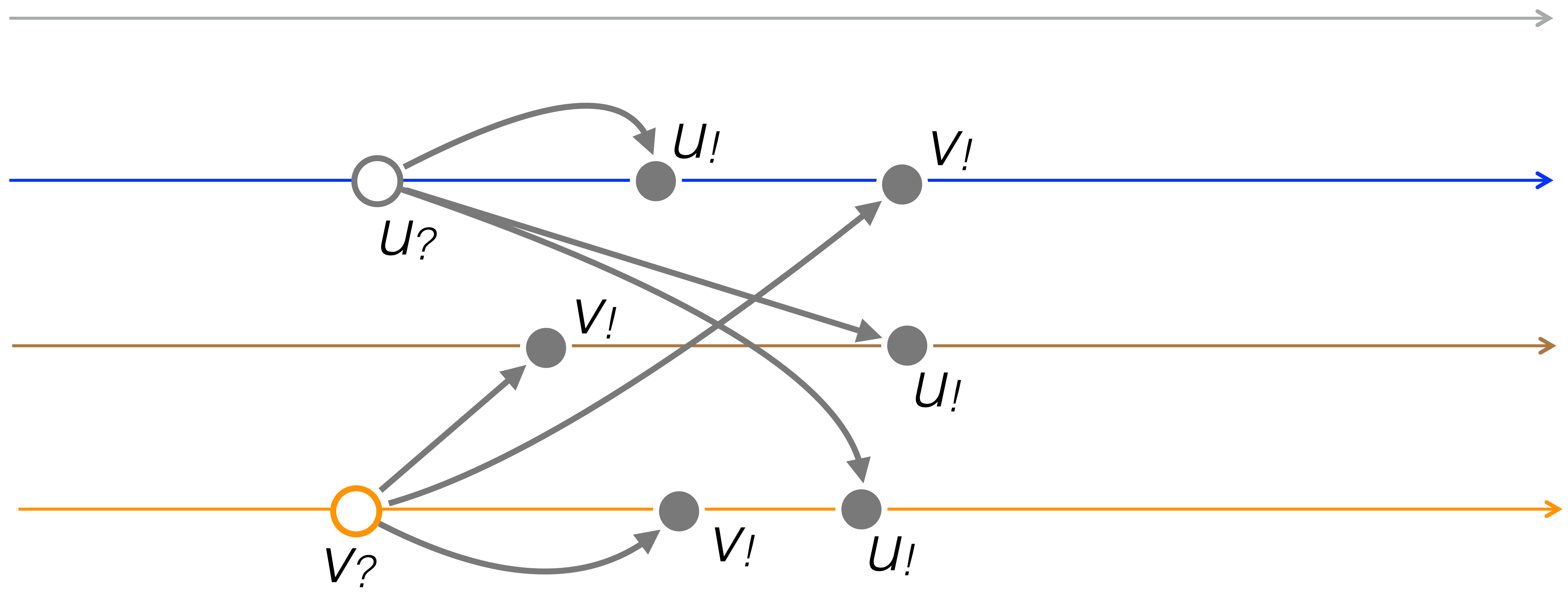
- *u: state ⤻ (retval, (state ⤻ state))*
- Prepare (@origin) $u_?$; deliver $u_!$
- Read one, write all (ROWA)
- Deferred-update replication (DUR)

Concurrency



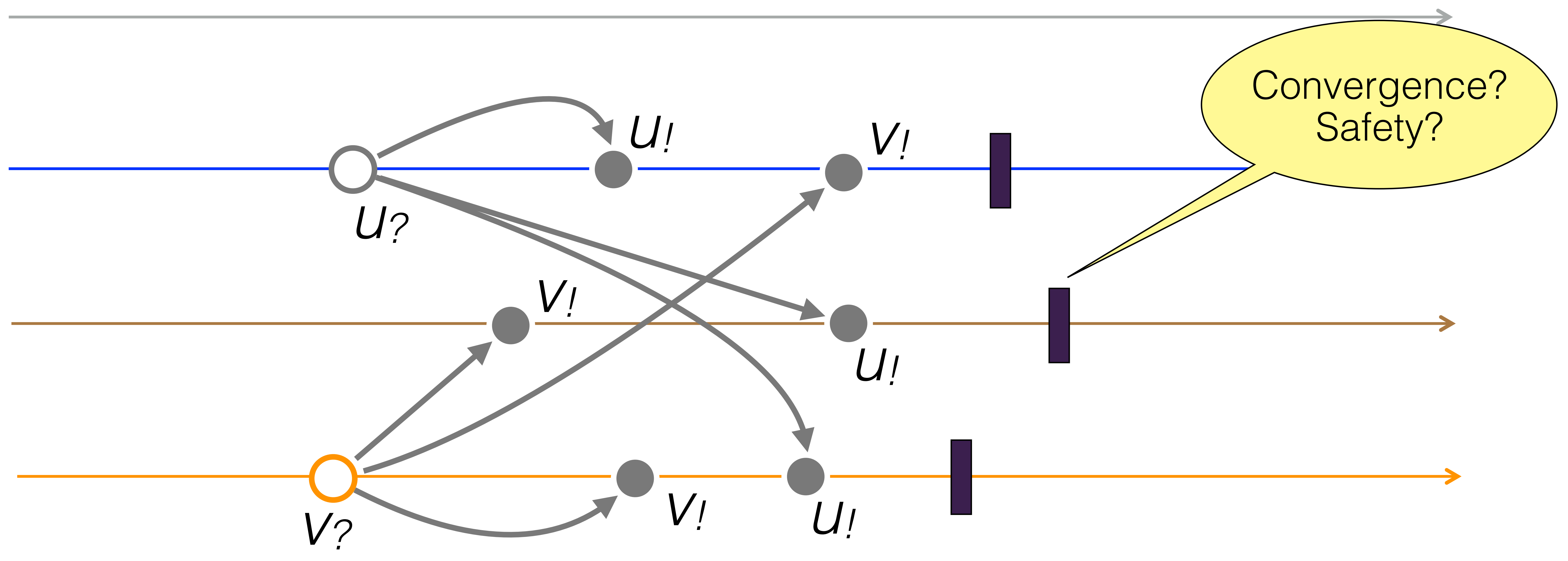
- Concurrent, Multi-master
- Strong: total order, identical state
- Weak: concurrent, interleaving, no global state

Concurrency



- Concurrent, Multi-master
- Strong: total order, identical state
- Weak: concurrent, interleaving, no global state

Concurrency



- Concurrent, Multi-master
- Strong: total order, identical state
- Weak: concurrent, interleaving, no global state

Anomalies of concurrent updates

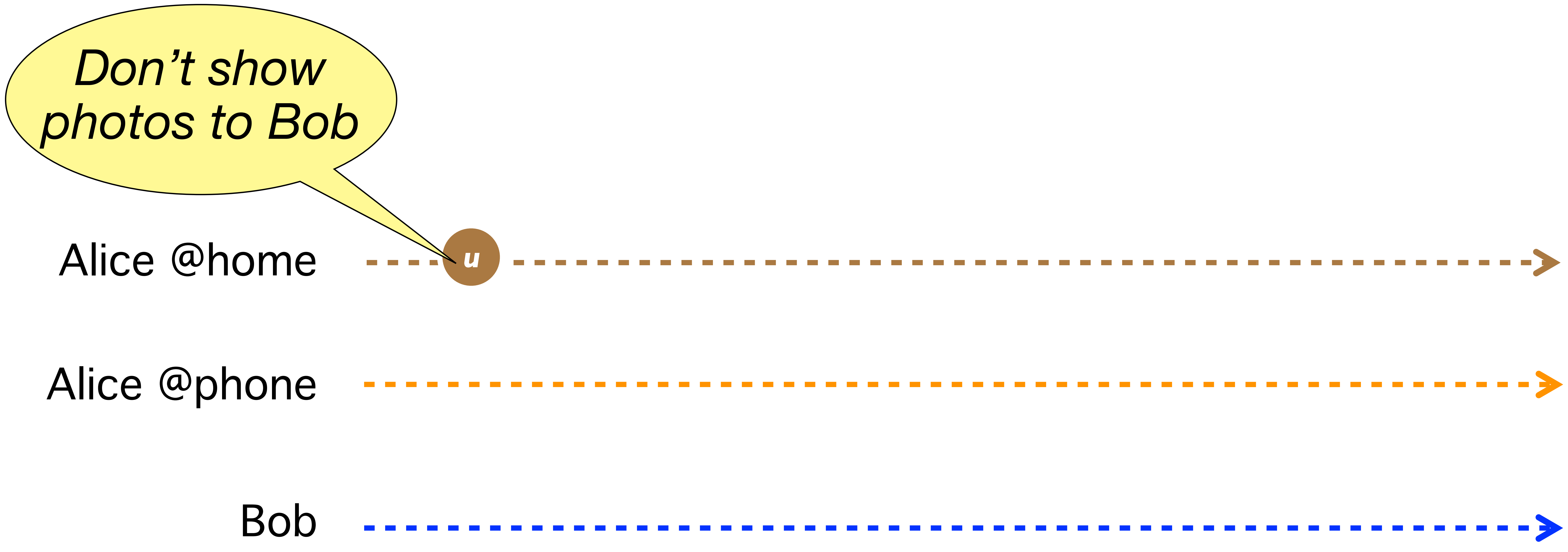
- Bank:
  - σ_{init} = 100€
  - Alice: *credit(20)* = { σ ≔ 120 }
  - Bob: *debit (60)* = { σ ≔ 40 }
  - σ = ???

Anomalies of concurrent updates

- Bank:
  - σ_{init} = 100€
  - Alice: *credit(20)* = { σ ≔ 120 }
  - Bob: *debit (60)* = { σ ≔ 40 }
  - σ = ???

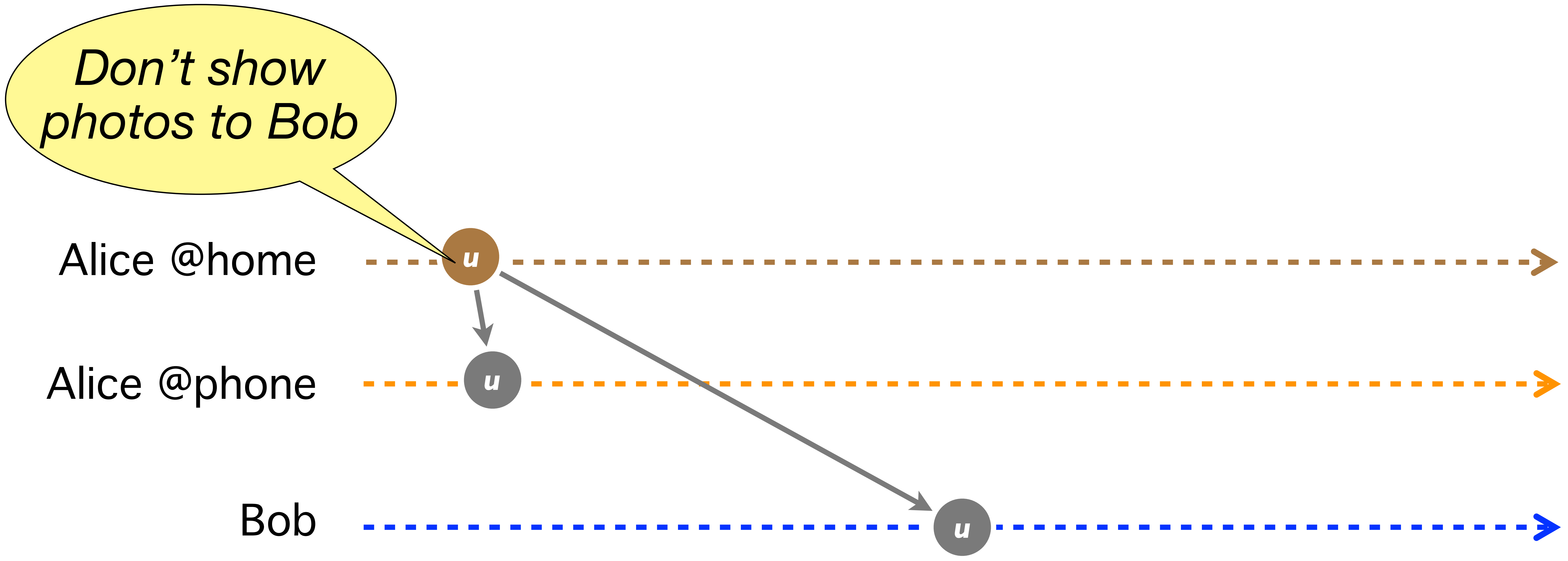
- File system:
  - σ_{init} = “/“
  - Alice: *mkdir (“/foo”); mkdir (“/foo/bar”)*
  - Bob: receives *mkdir (“/foo/bar”)*
  - σ = ???

Eventual Consistency



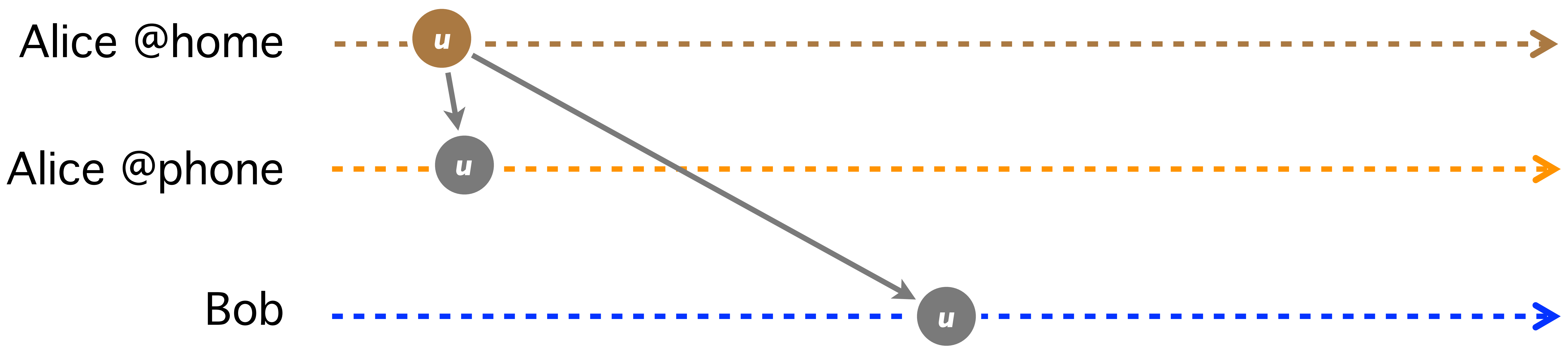
- *access (Bob, photo)* $\Longrightarrow$ *ACL (Bob, photo)*
- *v* observed effects of *u* $\Longrightarrow$ *v* should be delivered after *u*
- *Available: doesn't slow down sender*

Eventual Consistency



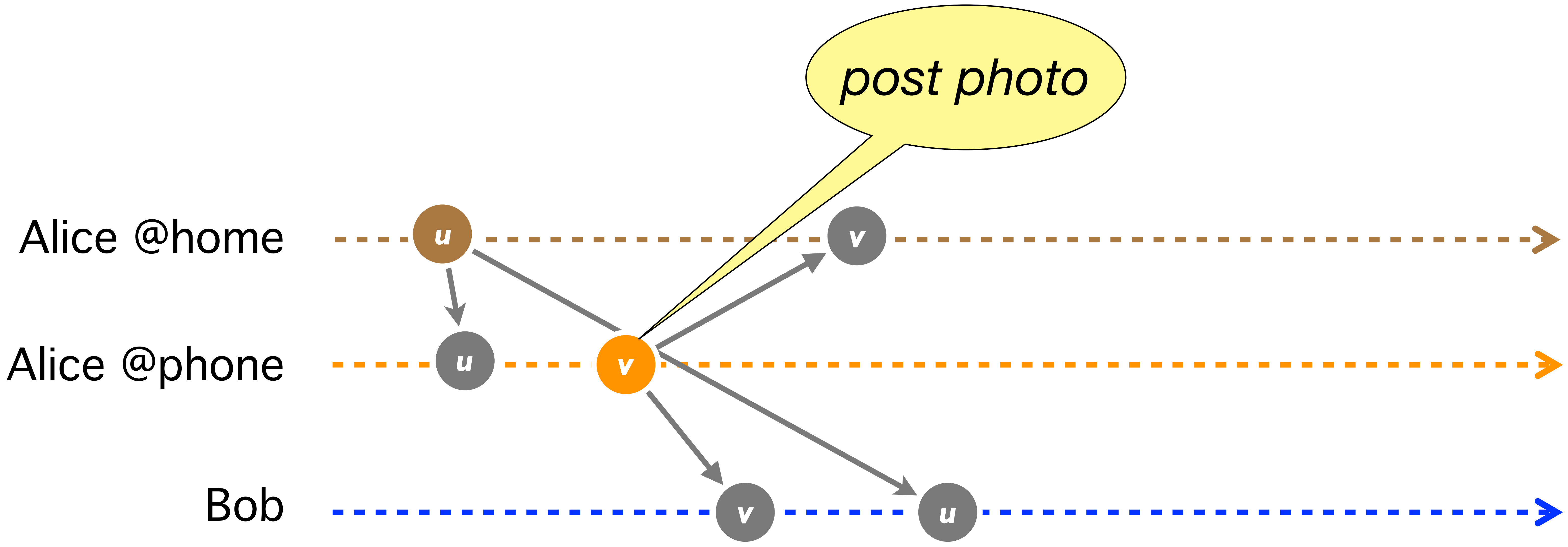
- *access (Bob, photo)* $\Longrightarrow$ *ACL (Bob, photo)*
- *v* observed effects of *u* $\Longrightarrow$ *v* should be delivered after *u*
- *Available: doesn't slow down sender*

Eventual Consistency



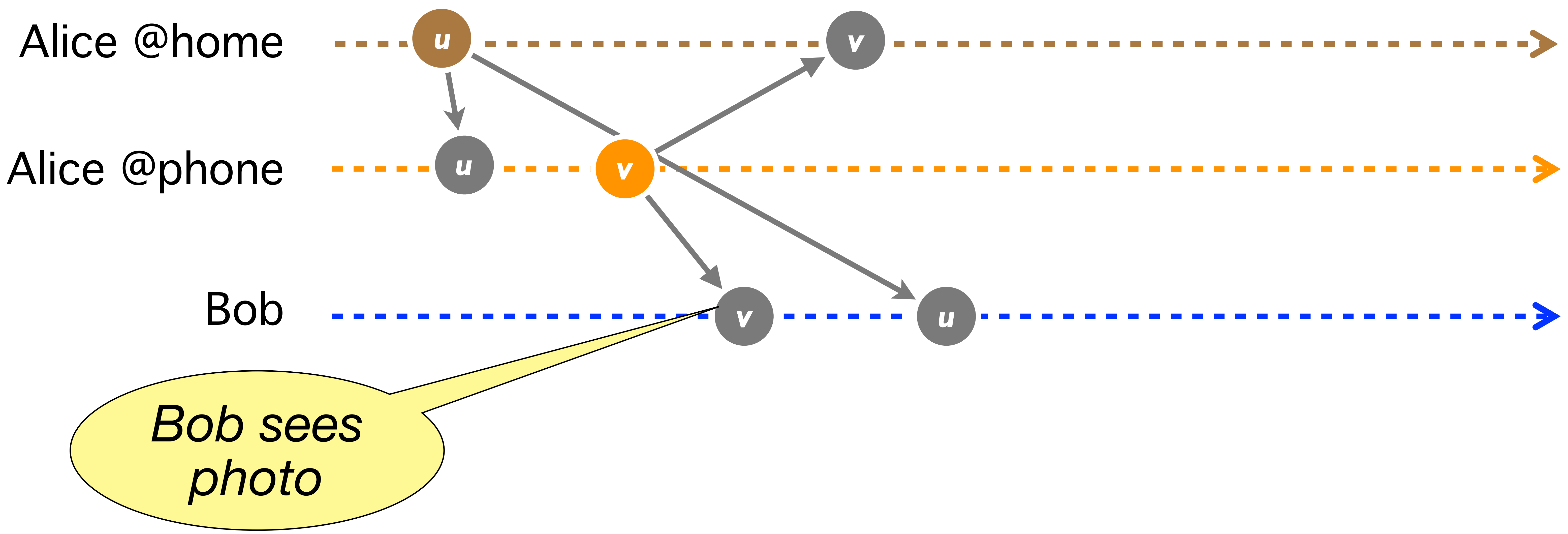
- *access (Bob, photo)* $\Longrightarrow$ *ACL (Bob, photo)*
- *v* observed effects of *u* $\Longrightarrow$ *v* should be delivered after *u*
- *Available: doesn't slow down sender*

Eventual Consistency



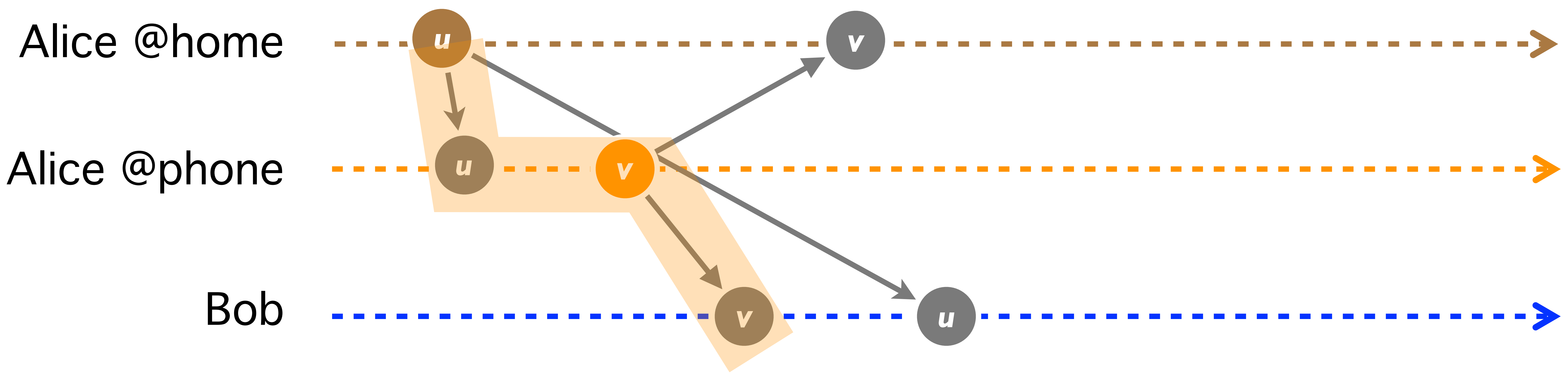
- *access (Bob, photo)* $\Longrightarrow$ *ACL (Bob, photo)*
- *v* observed effects of *u* $\Longrightarrow$ *v* should be delivered after *u*
- *Available: doesn't slow down sender*

Eventual Consistency



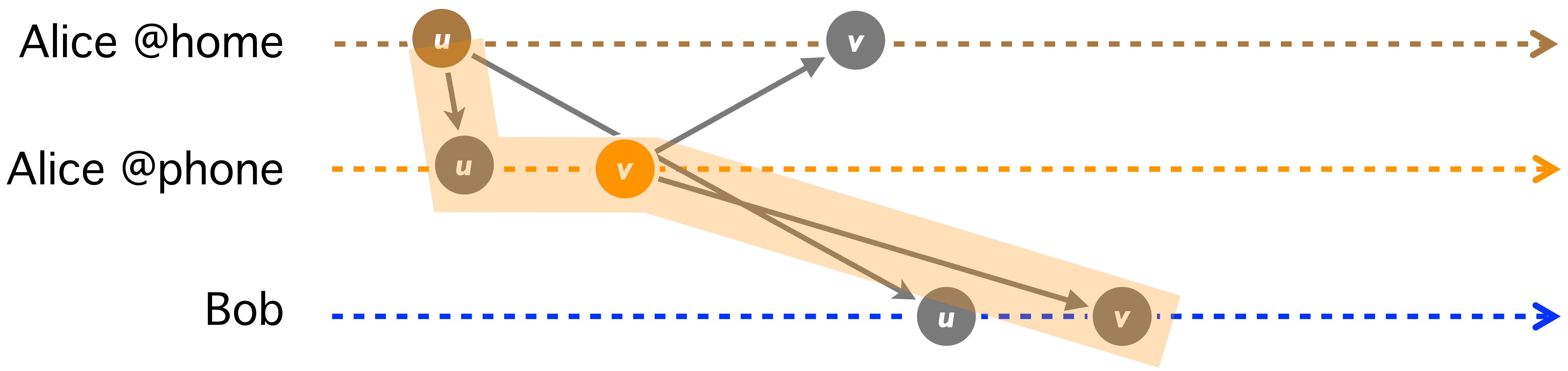
- *access (Bob, photo)* $\Longrightarrow$ *ACL (Bob, photo)*
- *v* observed effects of *u* $\Longrightarrow$ *v* should be delivered after *u*
- *Available: doesn't slow down sender*

Eventual Consistency



- *access (Bob, photo)* $\Longrightarrow$ *ACL (Bob, photo)*
- *v* observed effects of *u* $\Longrightarrow$ *v* should be delivered after *u*
- *Available: doesn't slow down sender*

(1) Causal consistency



- *access (Bob, photo)* $\Longrightarrow$ *ACL (Bob, photo)*
- *v* observed effects of *u* $\Longrightarrow$ *v* should be delivered after *u*
- *Available: doesn't slow down sender*

(2) Conflict-free Replicated Data Types (CRDTs)

- Data type
  - Encapsulates state
- Replicated
  - At multiple nodes
- Available
  - Update my replica without coordination
  - Convergence guaranteed by design
  - Decentralized, peer-to-peer

Commute $\Longrightarrow$ Converge

- Bank account:
  - ***credit(amt)***$_!$ = { *local_balance += amt* }
  - ***debit(amt)***$_!$ = { *local_balance –= amt* }
  - ***interest()***$_!$ = { *local_balance += origin_balance**.05 }
- File system:
  - *write(f)*$_!$ = { *local_f* ⊔ *f* }

CRDT design concept

- Backward-compatible with sequential datatype
- Commute $\Longrightarrow$ concurrent is same
  - *add(e); rm(f) = rm(f); add(e)* ≜ *add(e) || rm (f)*
- Otherwise, *concurrency semantics*
  - Example: *add(e) || rm (e)*
  - Deterministic, similar to sequential
    - ≈ *rm(e);add(e)* or ≈ *add(e); rm(e)*
  - Merge, don't lose updates
  - Result doesn't depend on order received
  - Stable preconditions

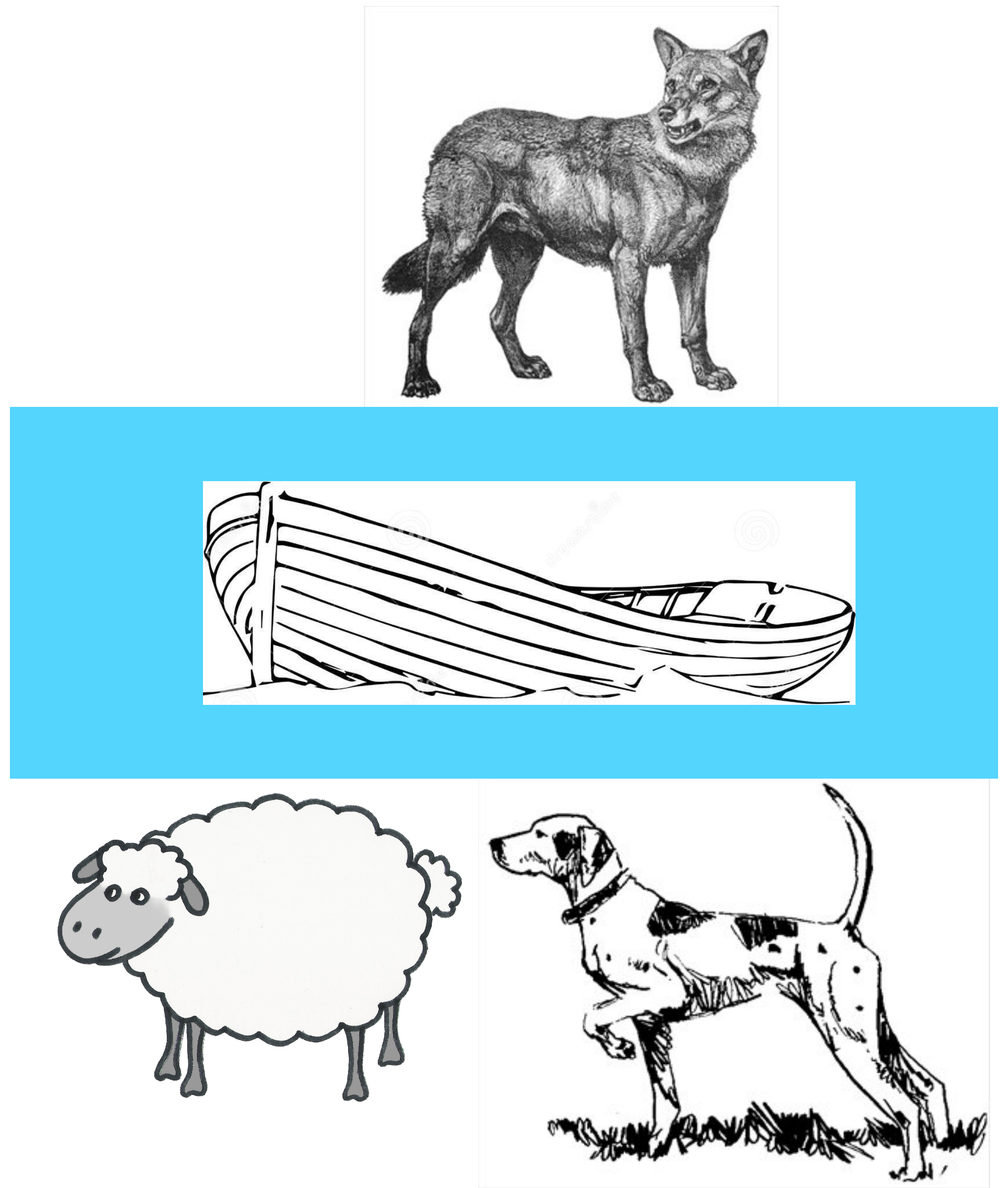
CRDT design concept

- Backward-compatible with sequential datatype
- Commute $\implies$ concurrent is same
  - *add(e); rm(f) = rm(f); add(e)* ≜ *add(e) || rm (f)*
- Otherwise, *concurrency semantics*
  - Example: *add(e) || rm (e)*
  - Deterministic, similar to sequential
    - ≈ *rm(e);add(e)* or ≈ *add(e); rm(e)*
  - Merge, don't lose updates
  - Result doesn't depend on order received
  - Stable preconditions

CRDT design concept

- Backward-compatible with sequential datatype
- Commute $\implies$ concurrent is same
  - *add(e); rm(f) = rm(f); add(e)* $\triangleq$ *add(e) || rm (f)*
- Otherwise, *concurrency semantics*
  - Example: *add(e) || rm (e)*
  - Deterministic, similar to sequential
    - $\approx$ *rm(e);add(e)* or $\approx$ *add(e); rm(e)*
  - Merge, don’t lose updates
  - Result doesn't depend on order received
  - Stable preconditions

Application invariants



- $South \uplus Boat \uplus North = \{ sheep, dog, wolf \}$
- $carryNorth(S) \Longrightarrow 1 \leq |S| \leq 2$
- $carrySouth(S) \Longrightarrow 1 \leq |S| \leq 2$
- $\forall S \in \{South, Boat, North\} : sheep \in S \wedge wolf \in S \Longrightarrow dog \in S$
- *Hard to tease invariants out*
  - Silent invariants

Seq. consistency examples

- Bank account
  - *deposit(amt), withdraw(amt), accrueInterest(amt)*
  - Invariant: *“balance ≥ 0”*
  - { *amt ≤ balance ∧ Inv* } *withdraw(amt)* { *Inv* }

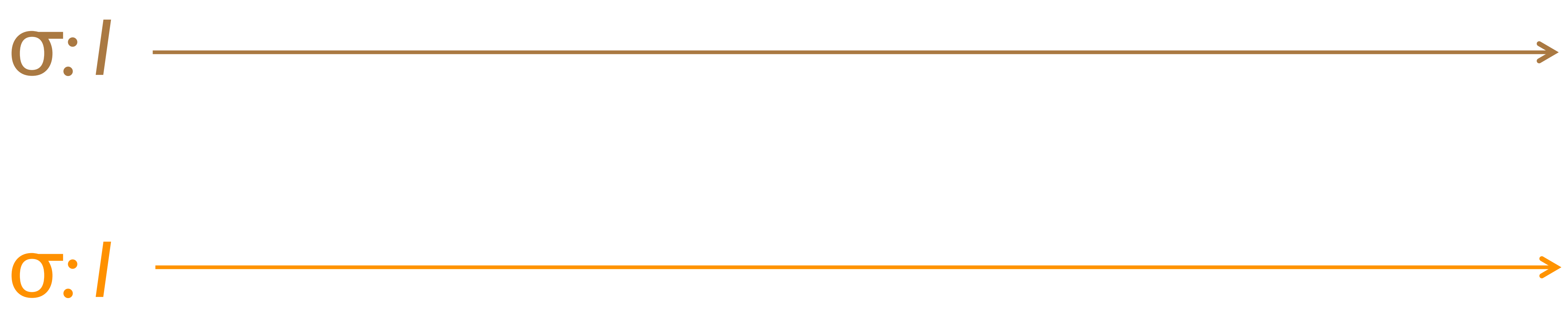
Seq. consistency examples

- Bank account
  - *deposit(amt), withdraw(amt), accrueInterest(amt)*
  - Invariant: *“balance ≥ 0”*
  - { $amt \leq balance \wedge Inv$ } *withdraw(amt)* { Inv }

- File system
  - *mkdir, rmdir, create, write, rm, ls*, etc.
  - Invariant: Tree
  - { Tree $\wedge \neg$ *x/…/y* } *mv(x,y)* { Tree }

Just-Right Consistency

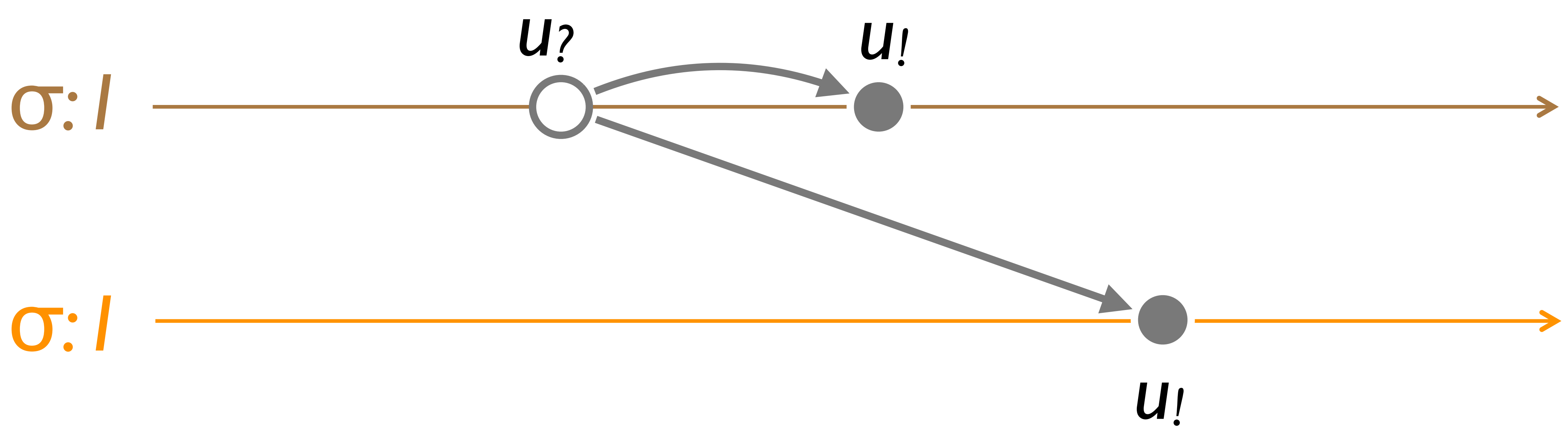
- CRDT geo-replicated database
  - Lots of internal parallelism
  - Transactional, causal consistency by default
- Specification of application updates, invariant
  - CISE: do all state transitions preserve the invariant?
  - If not, fix: adjust
    - either specification
    - or synchronisation
  - Repeat until safe
- App / synch co-design: Minimal synchronisation



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

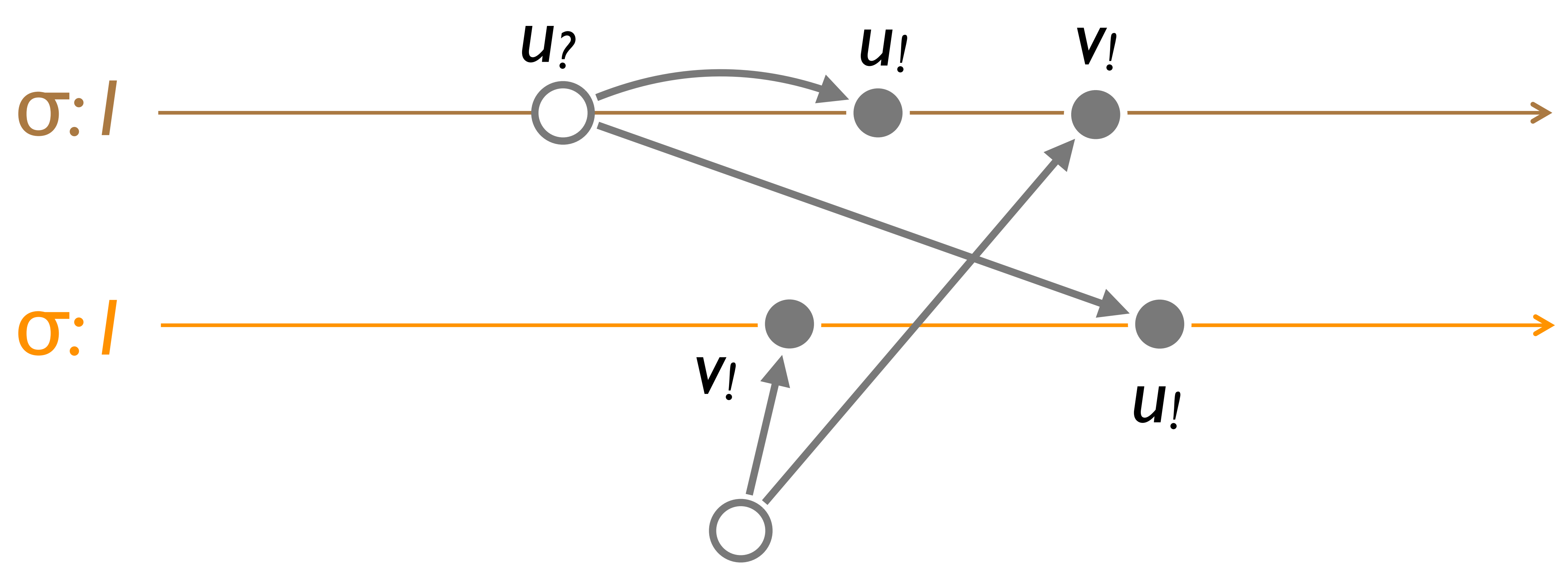
Converge? Invariant OK?



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

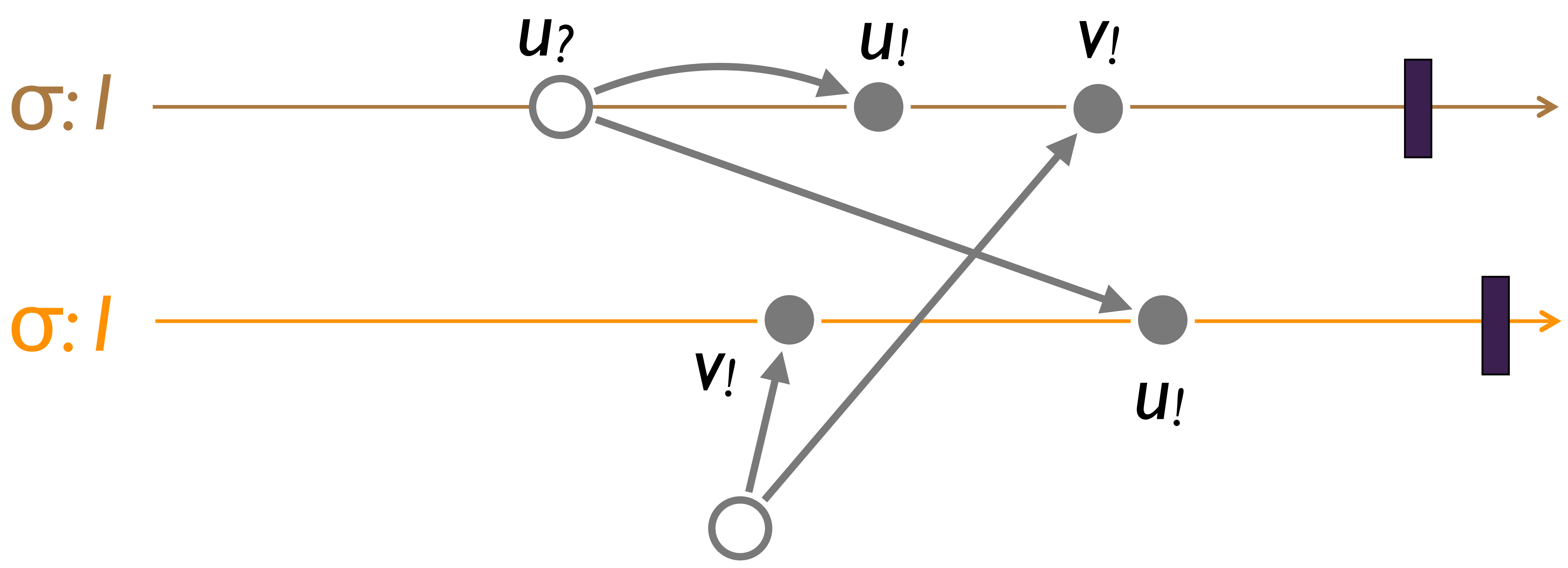
Converge? Invariant OK?



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

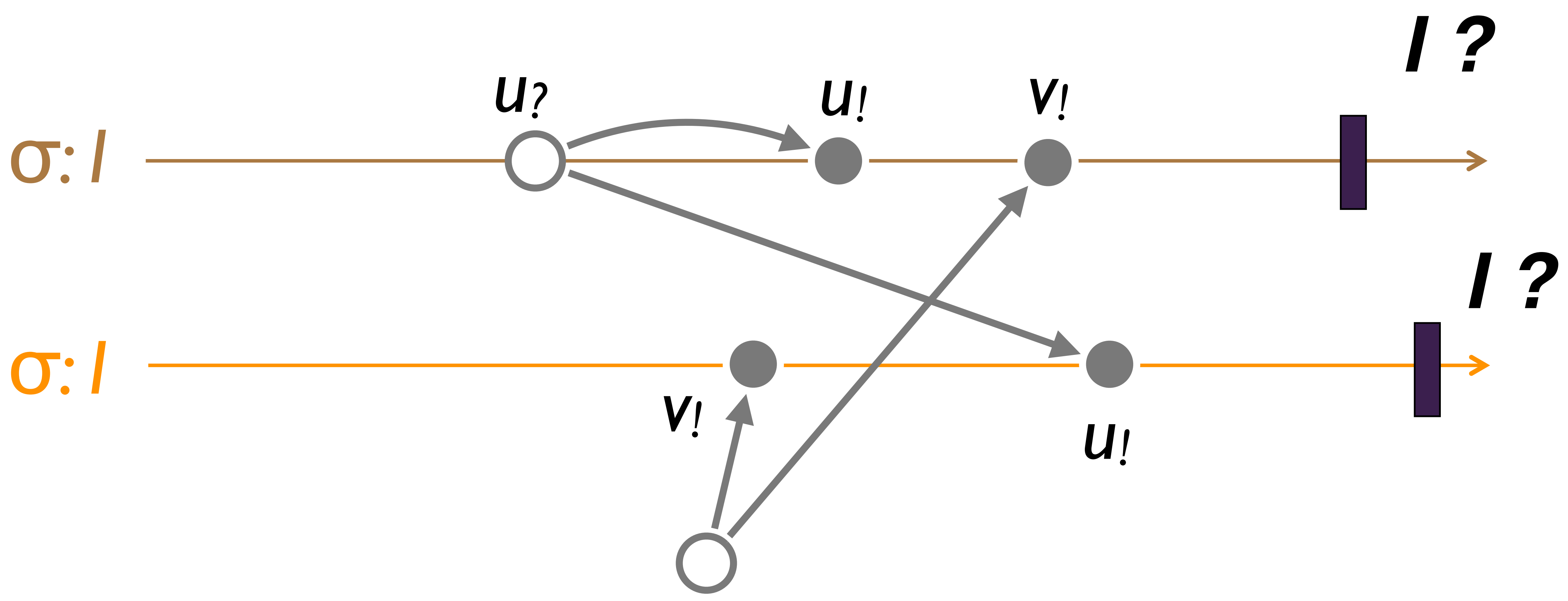
Converge? Invariant OK?



Asynchronous, replicated updates

- State σ
- Invariant *I*
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

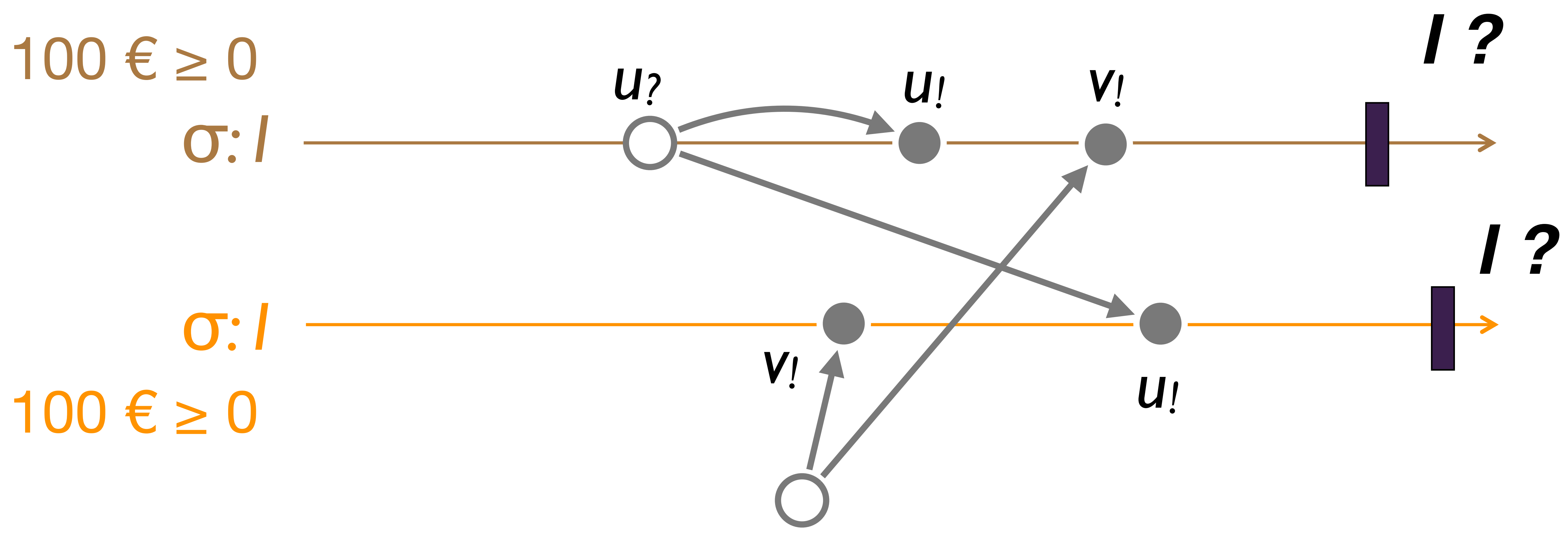
Converge? Invariant OK?



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

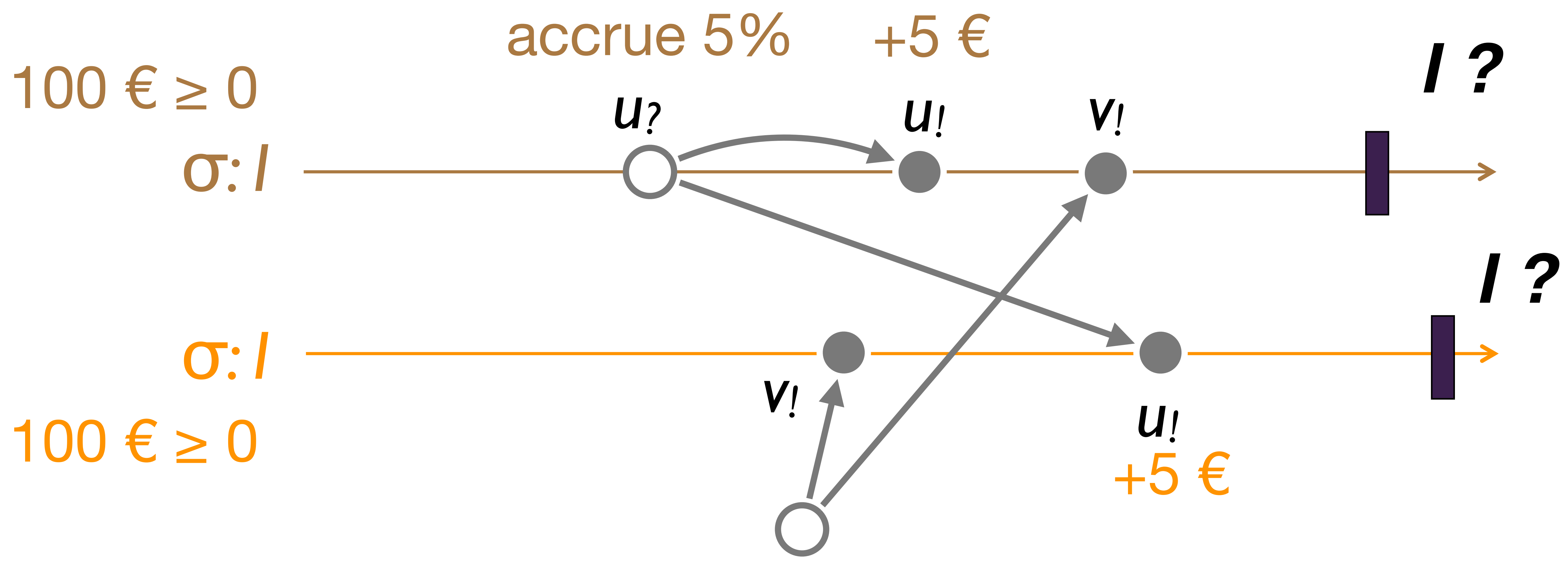
Converge? Invariant OK?



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

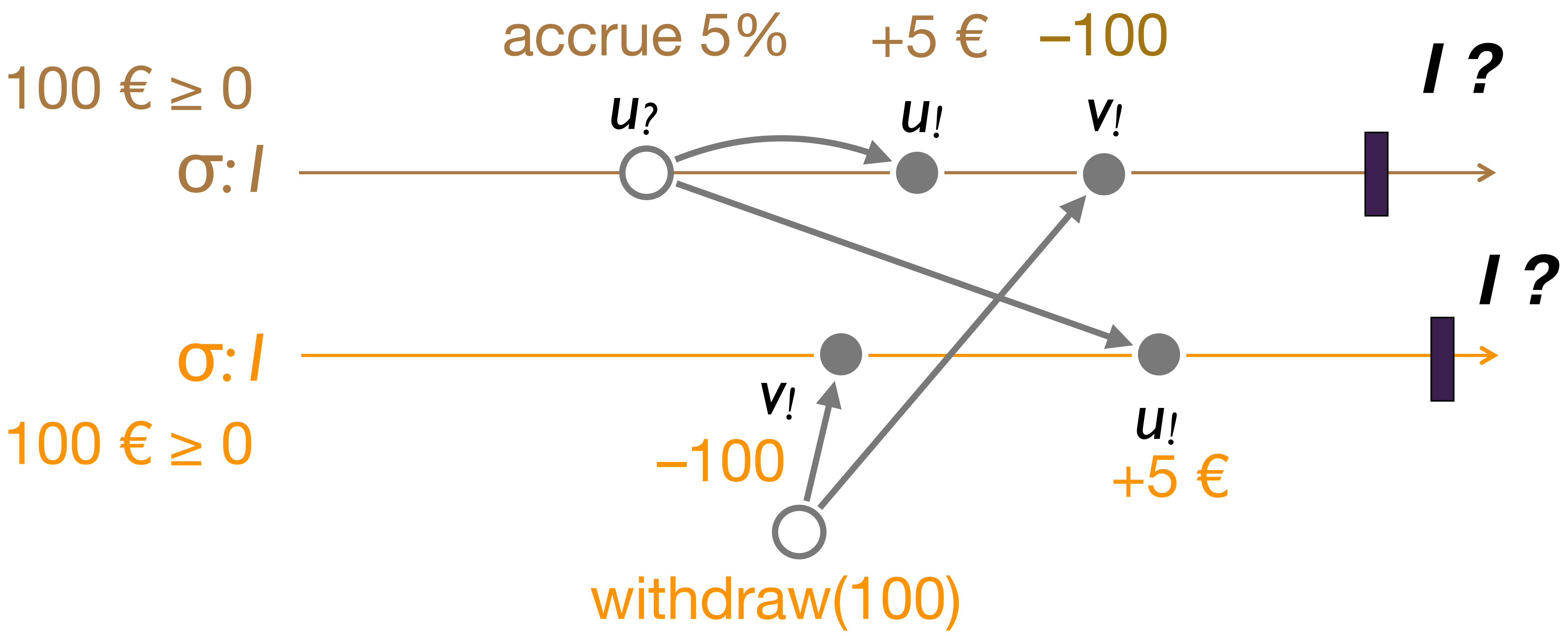
Converge? Invariant OK?



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

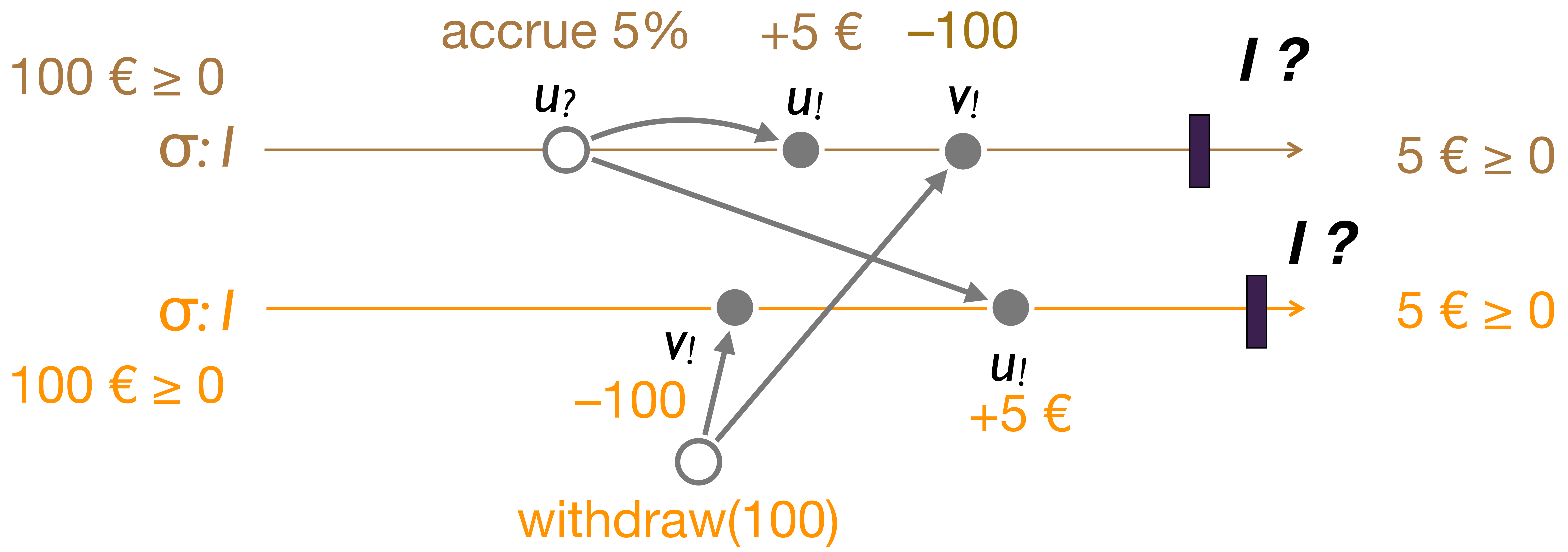
Converge? Invariant OK?



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

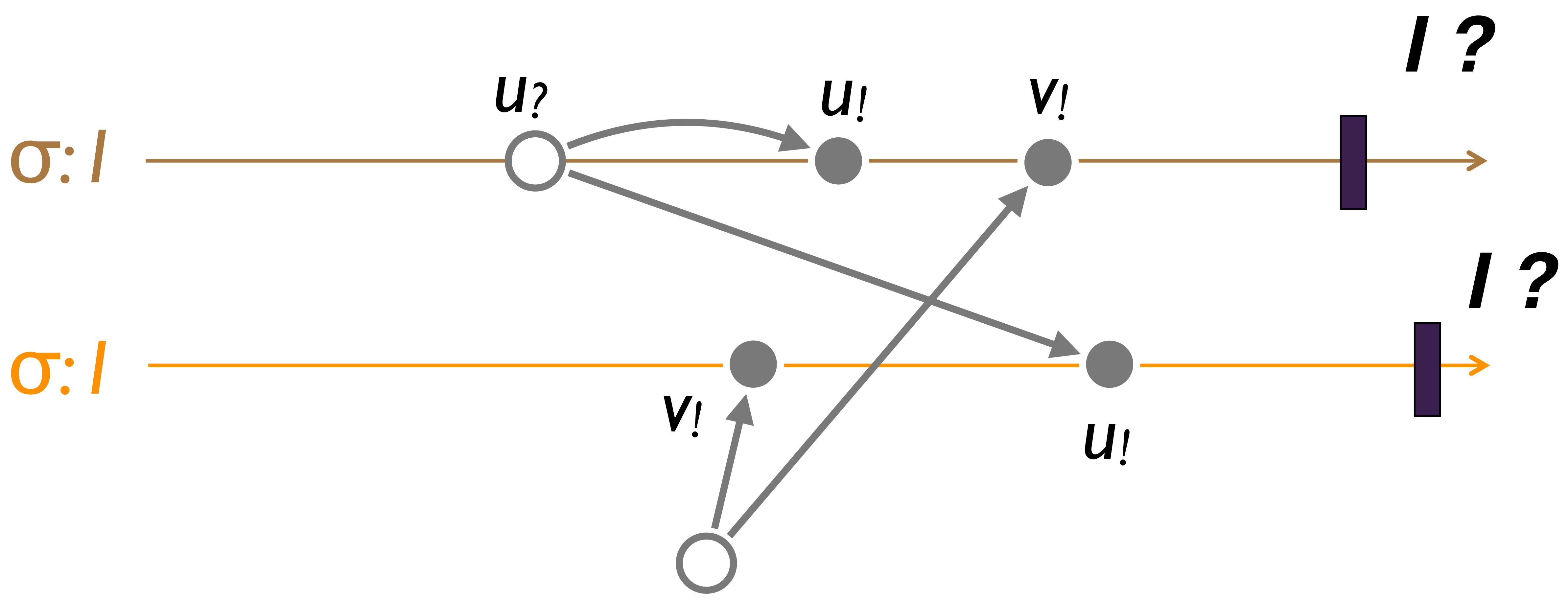
Converge? Invariant OK?



Asynchronous, replicated updates

- State σ
- Invariant I
- Prepare: read one, generate effector
- Update all, deferred: deliver effector

Converge? Invariant OK?



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

2: Convergence

- Concurrent effectors commute

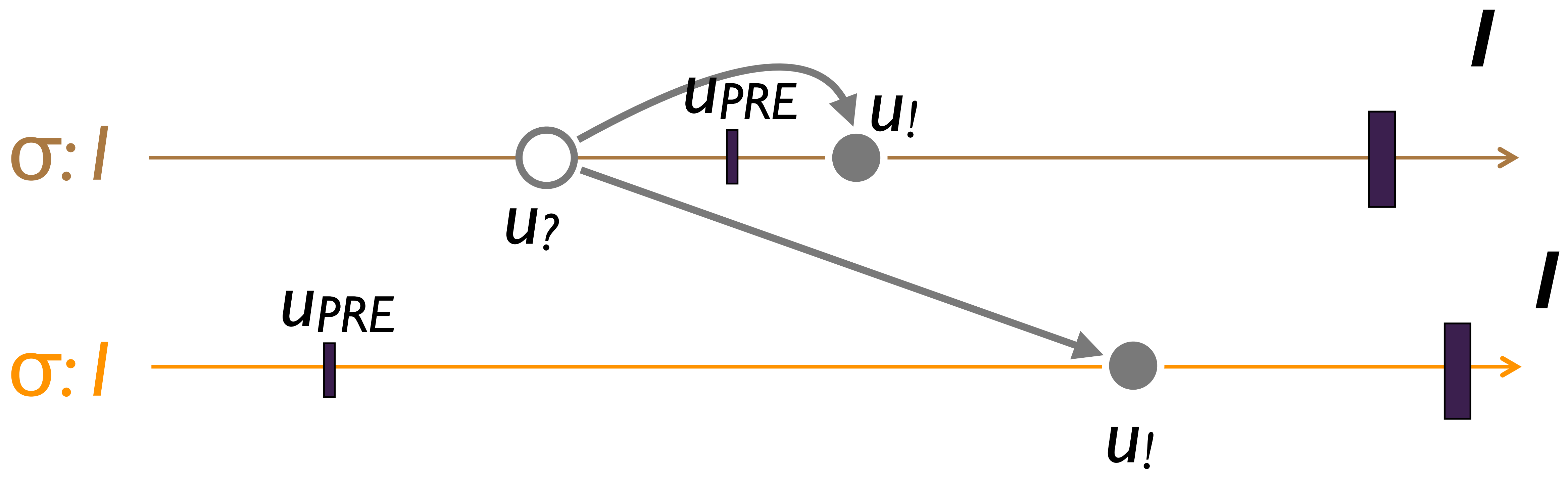
3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed

Simple example: bank account

- Operations: ***deposit(amount), withdraw(amount)***
- Invariant: *balance ≥ 0*
  - Start with weak specification
  - Rule 1 ⟶ strengthen precondition for withdraw
  - Rule 2: OK
  - Rule 3 ⟶ ***withdraw*** || ***withdraw*** unsafe
    - fixed with concurrency control



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

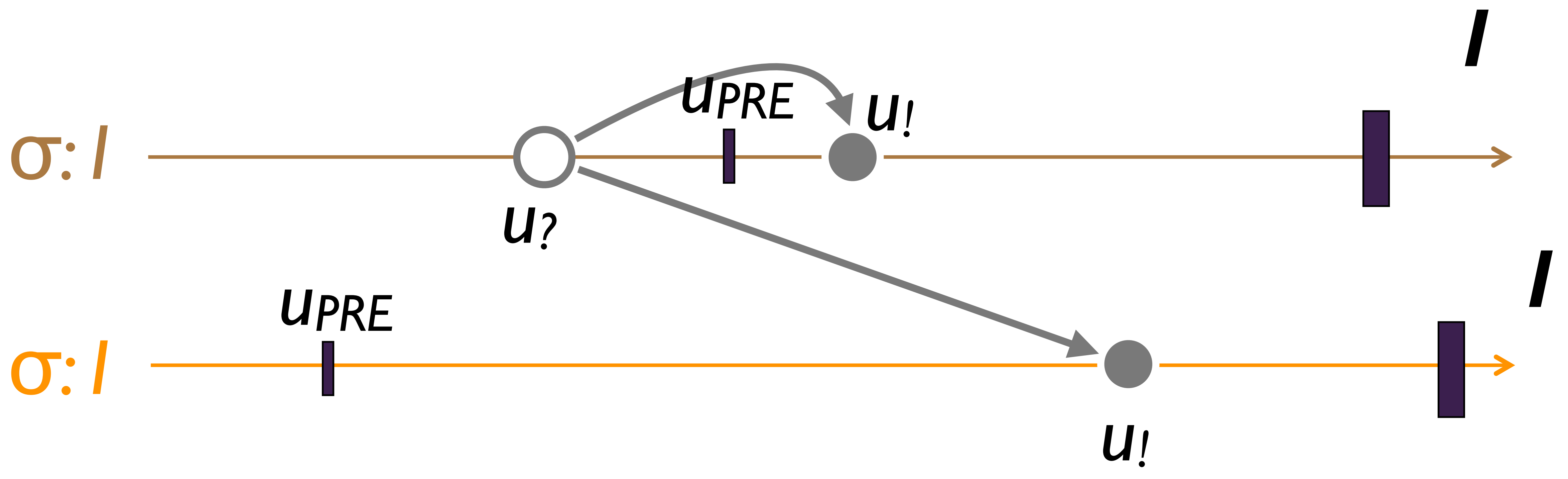
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

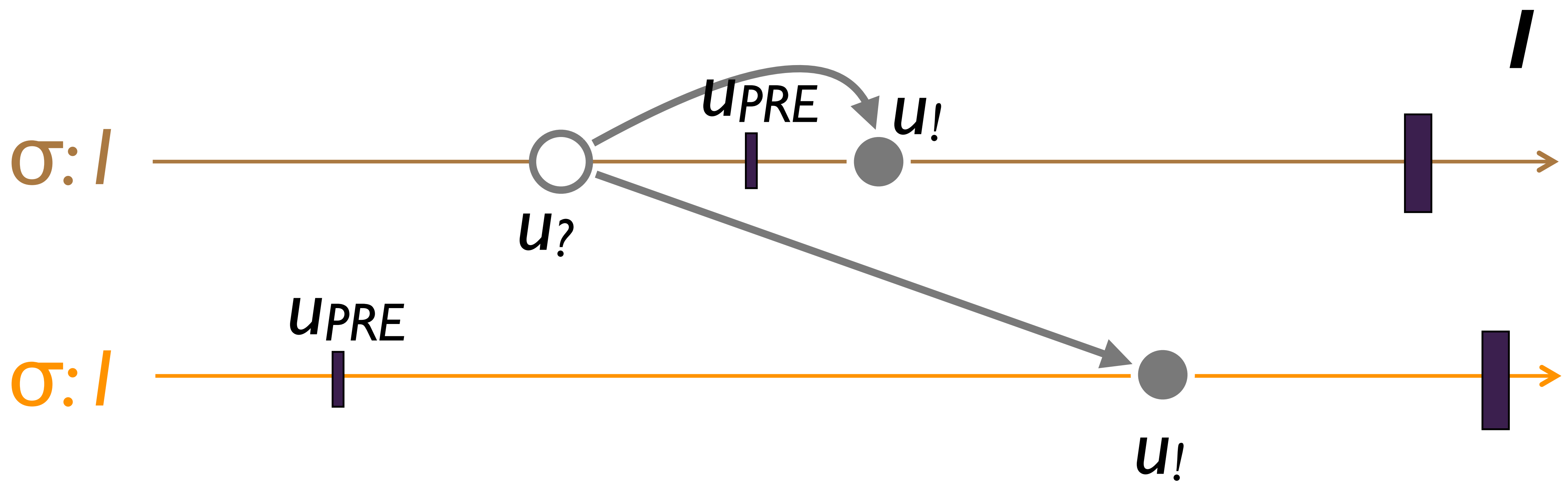
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

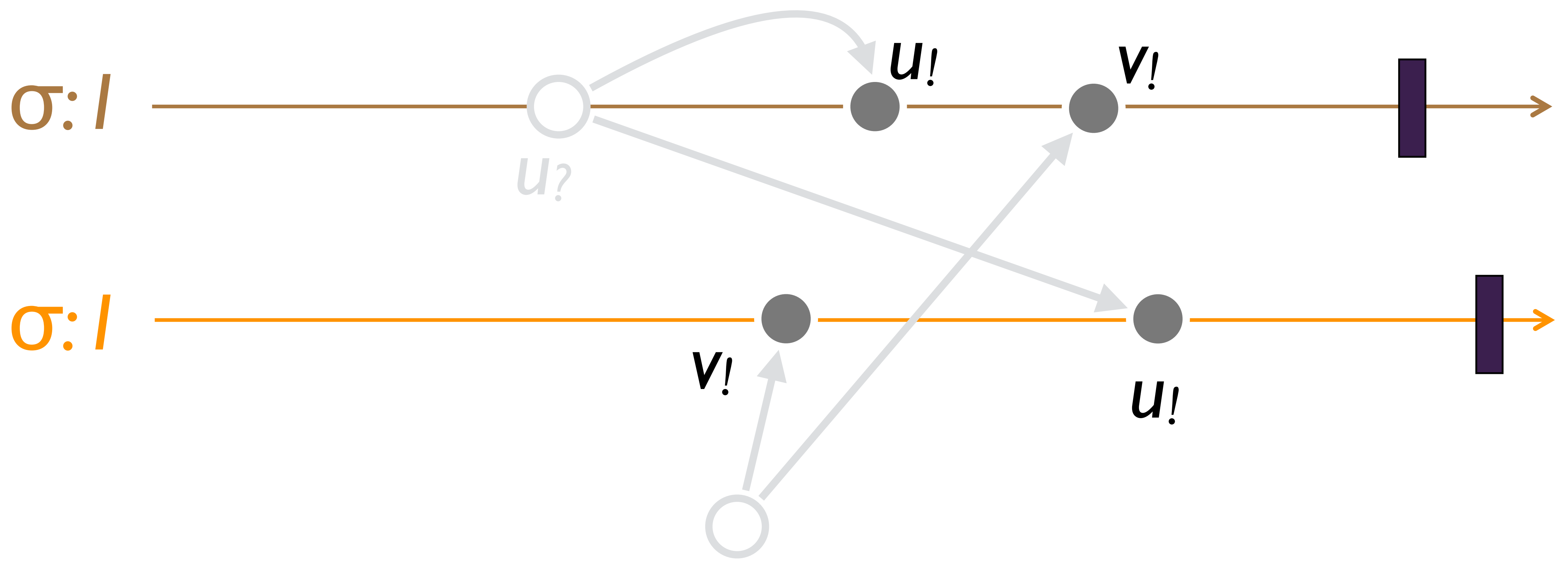
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

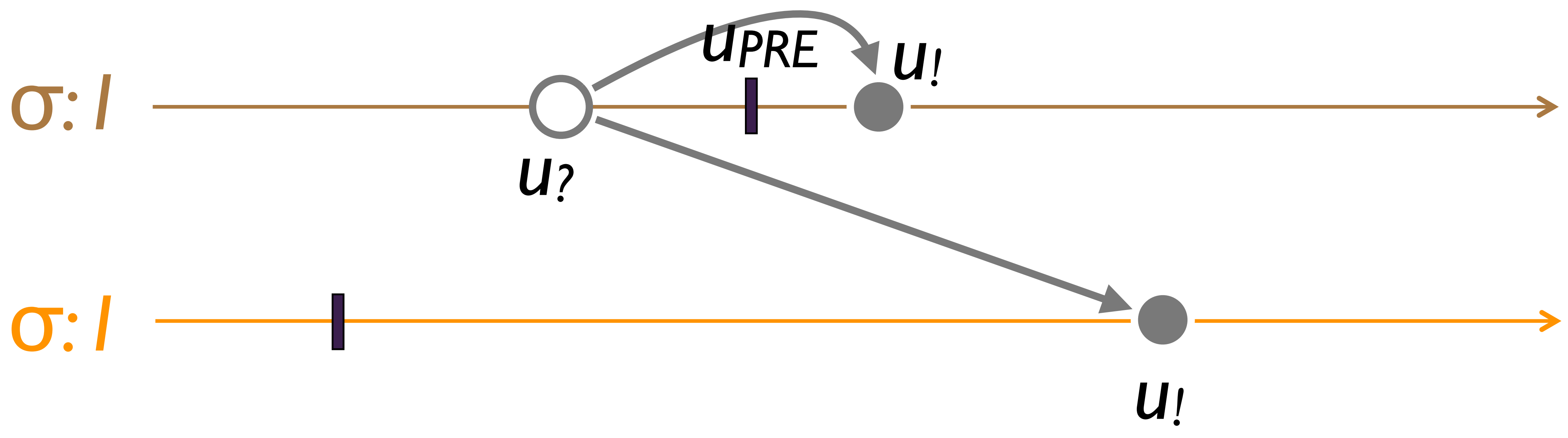
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

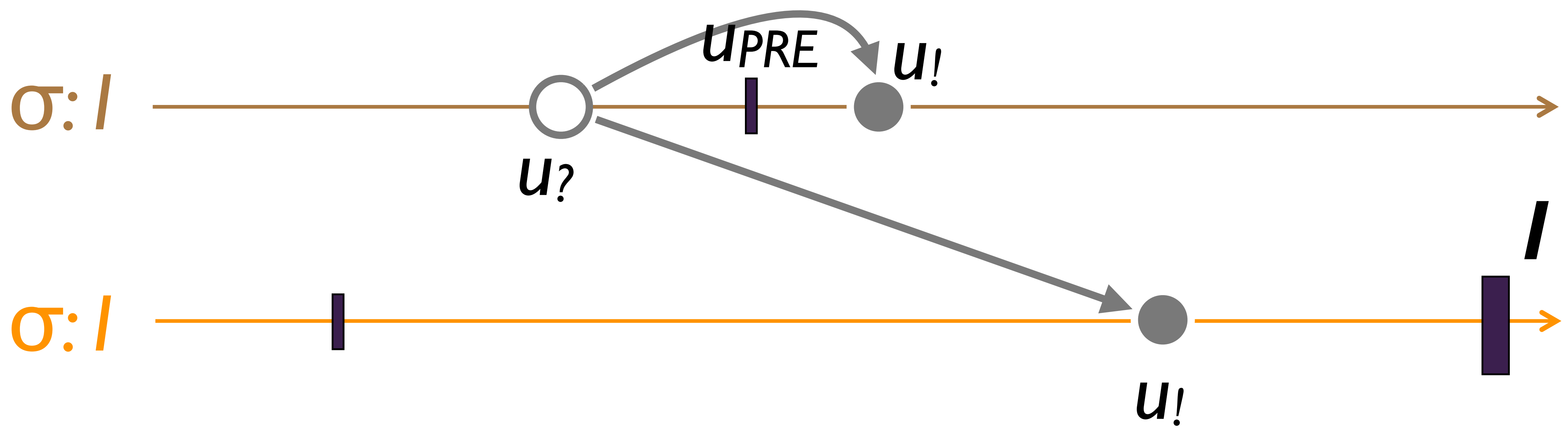
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

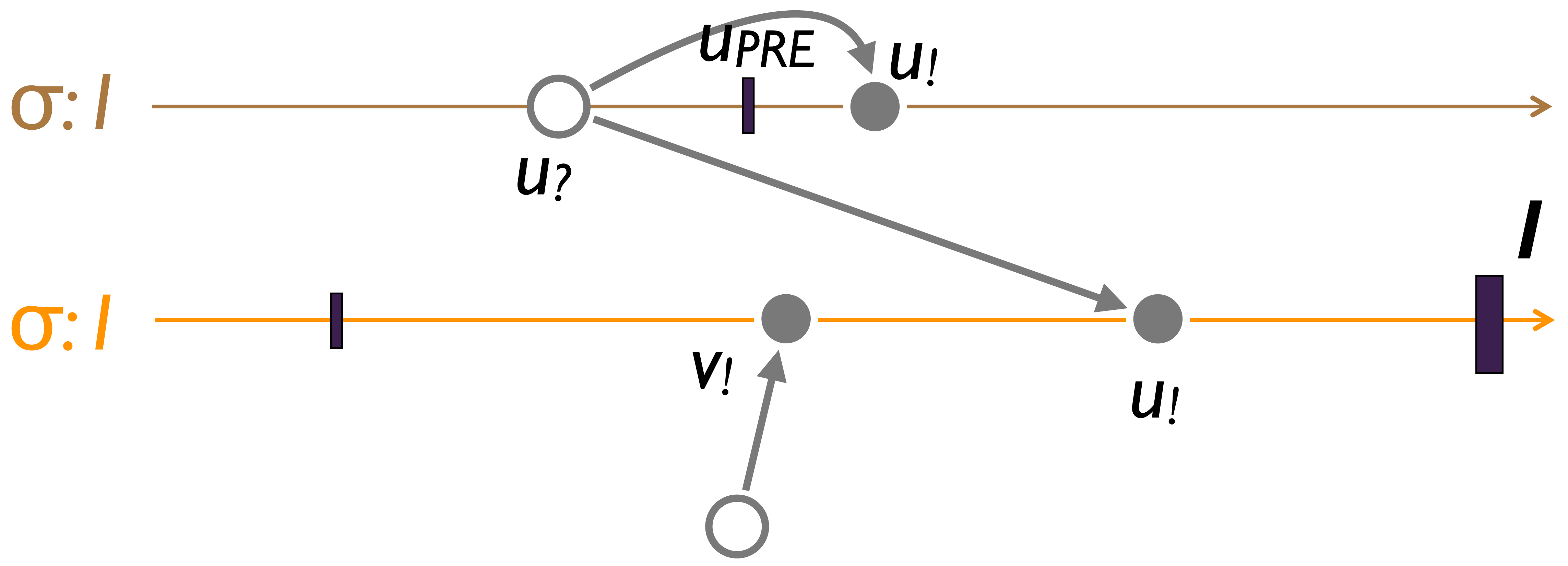
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

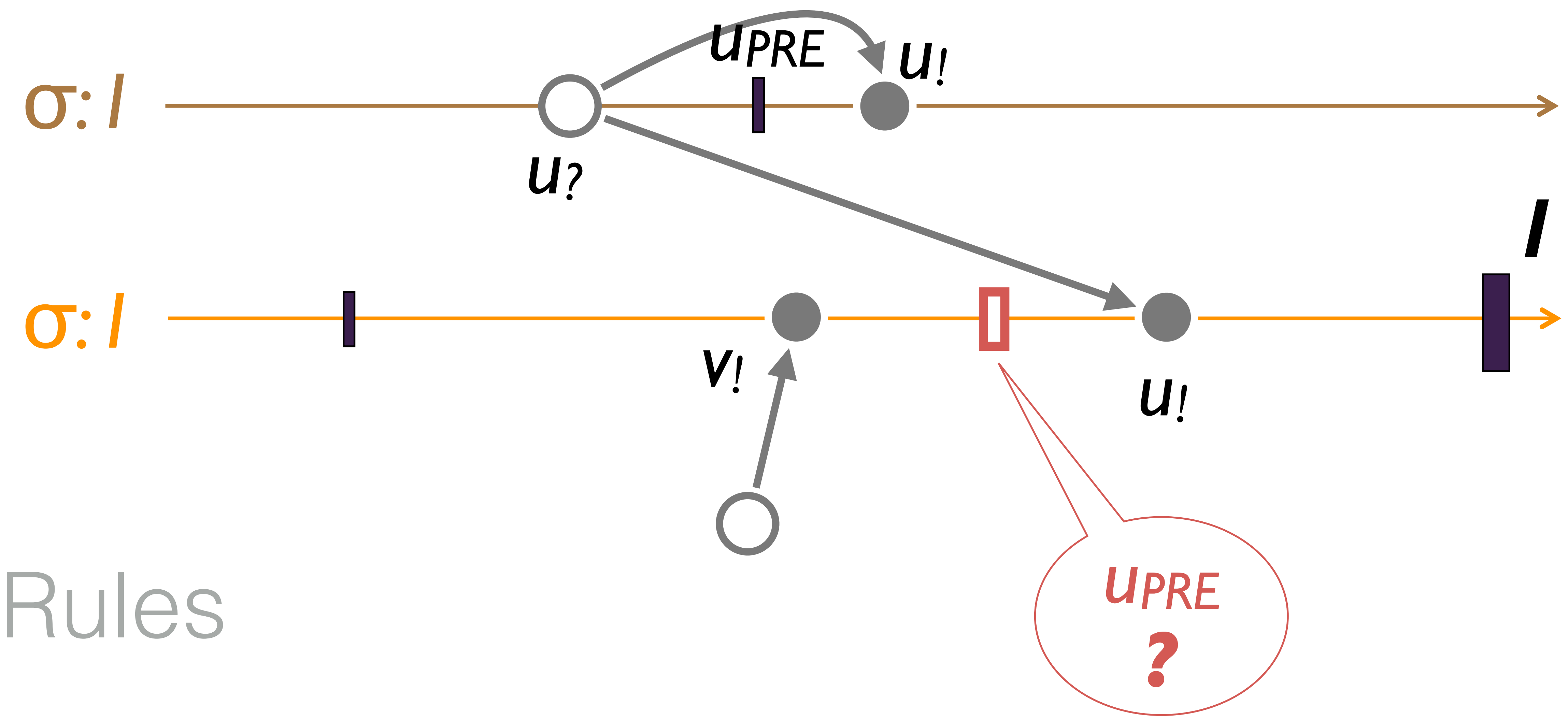
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



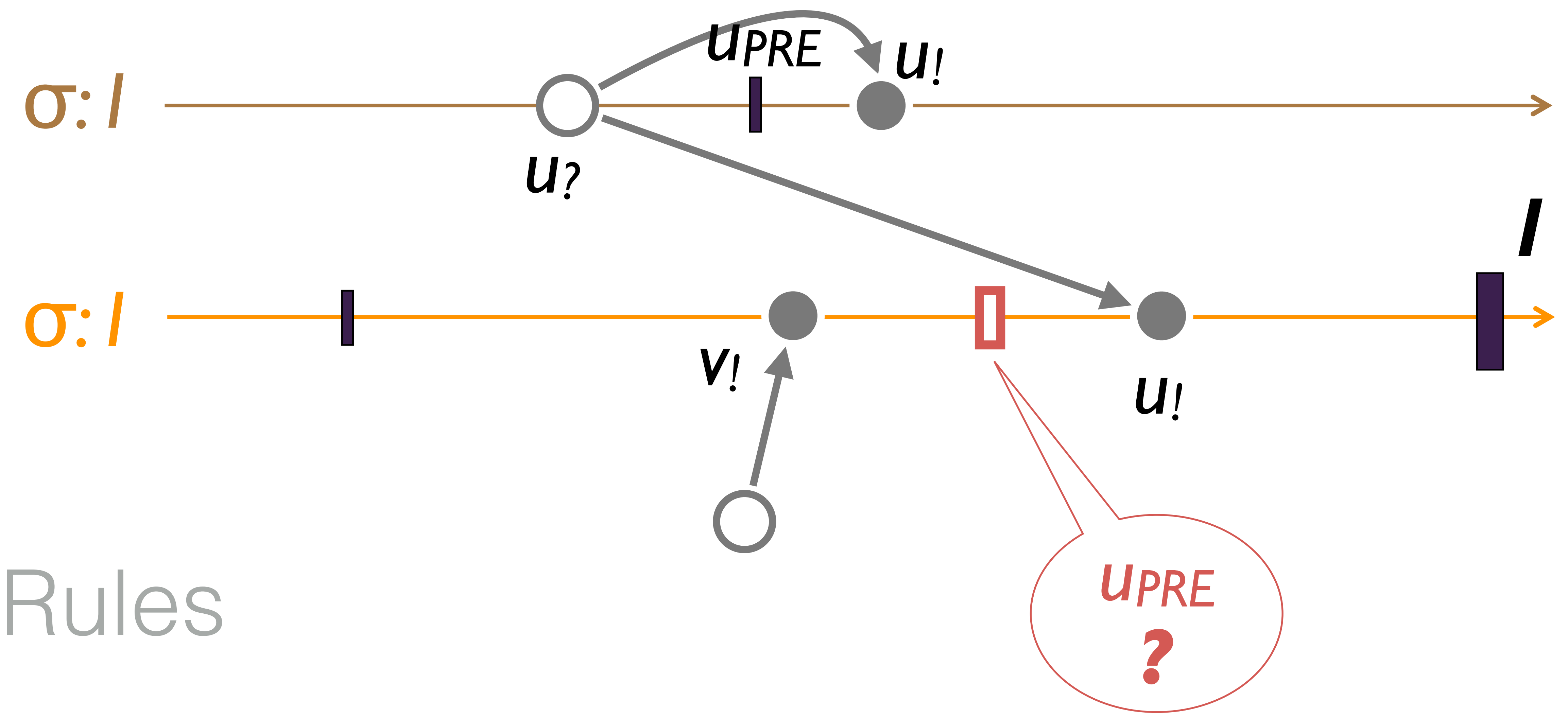
CISE Rules

1: Sequential correctness
- Any single operation maintains the invariant

2: Convergence
- Concurrent effectors commute

3: Precondition Stability
- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

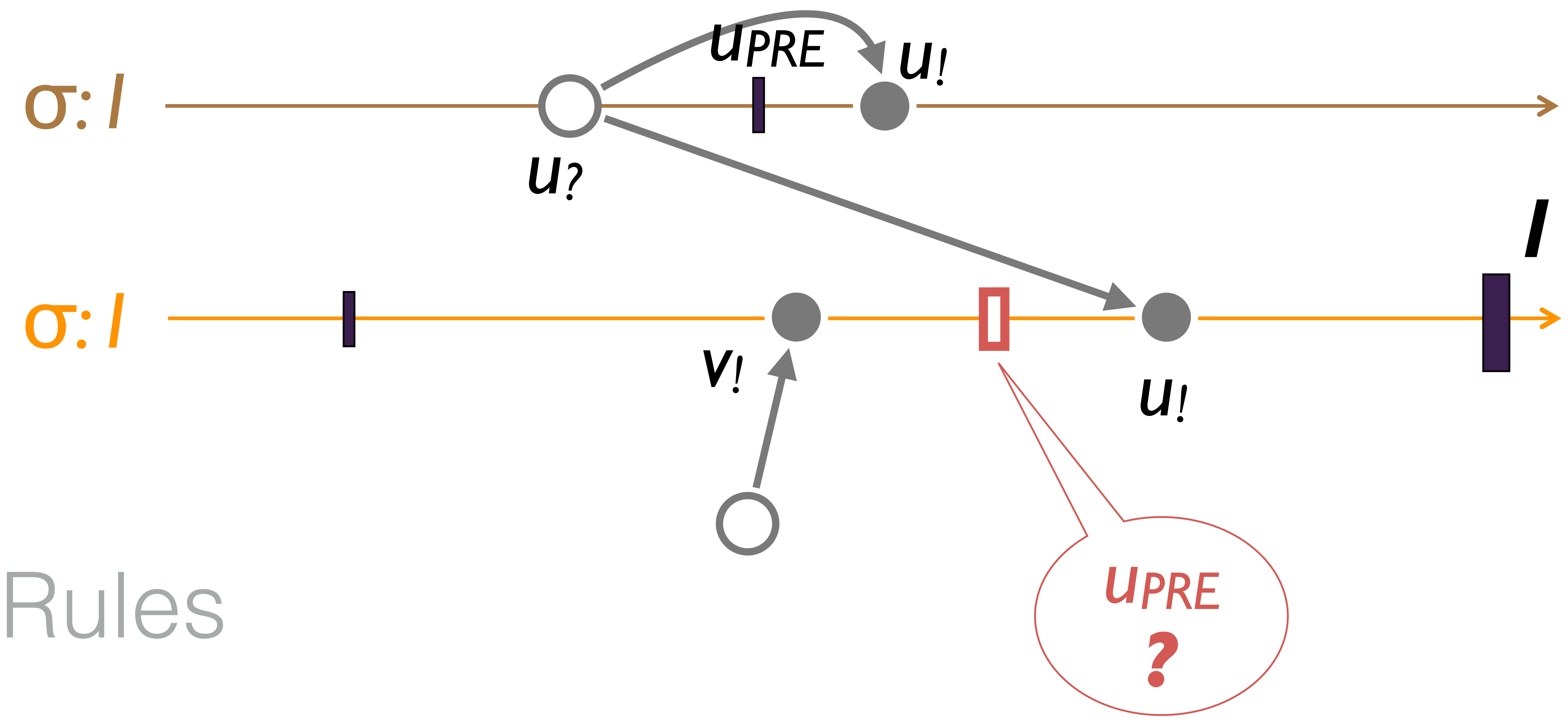
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

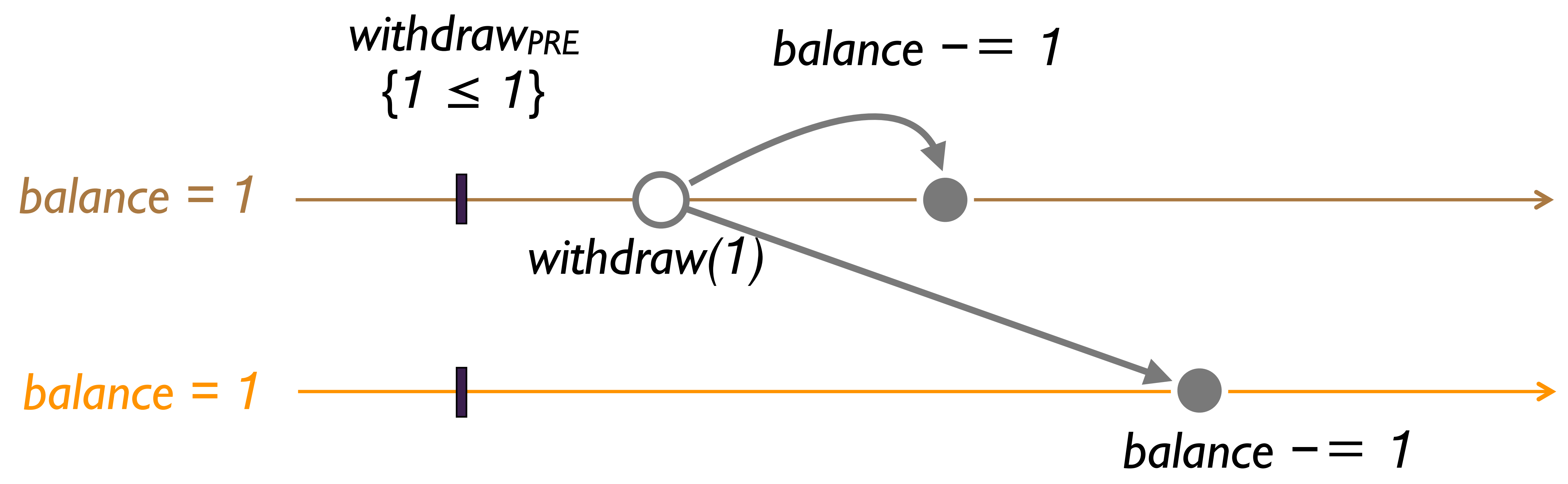
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



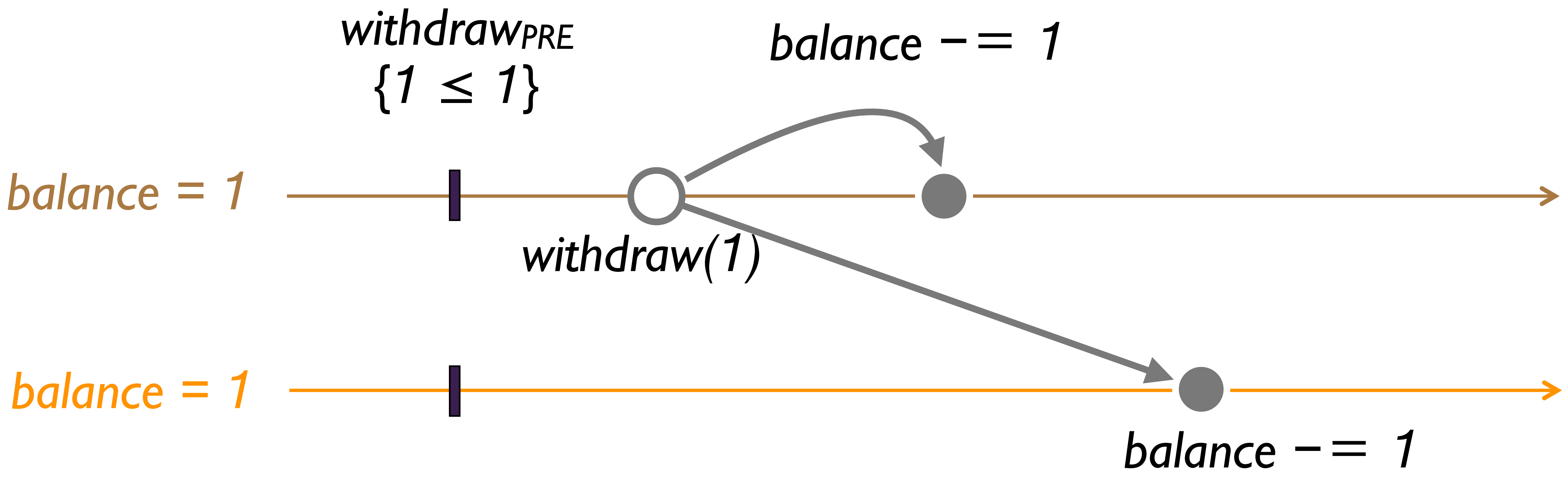
CISE Rules

1: Sequential correctness
- Any single operation maintains the invariant

2: Convergence
- Concurrent effectors commute

3: Precondition Stability
- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



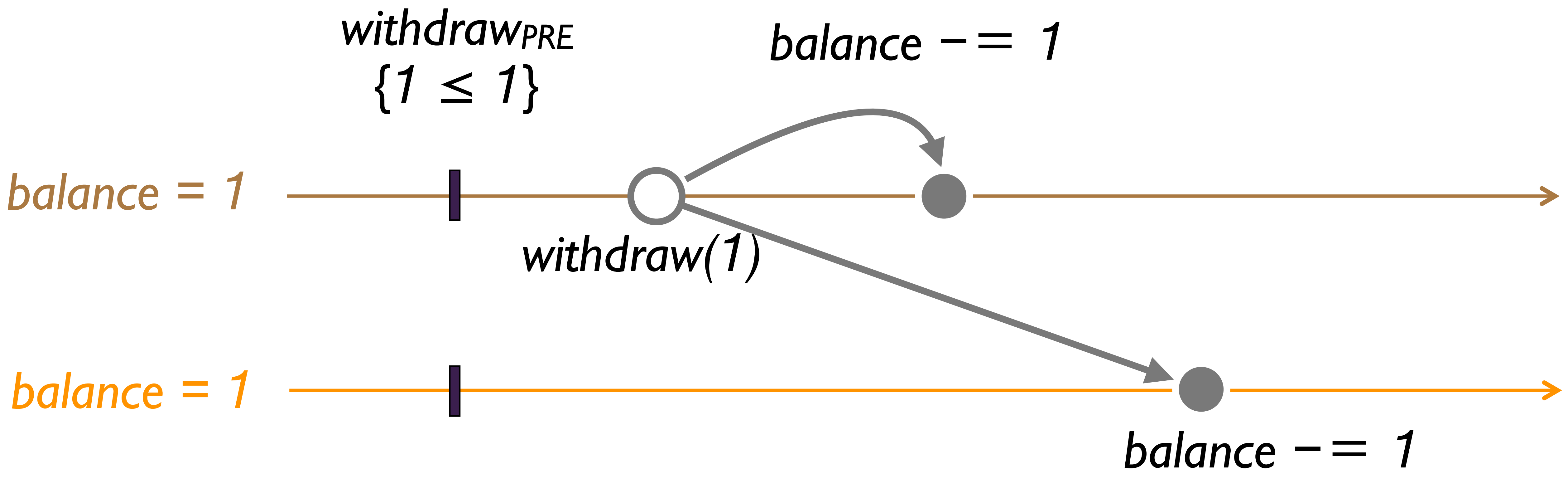
CISE Rules

1: Sequential correctness
- Any single operation maintains the invariant

2: Convergence
- Concurrent effectors commute

3: Precondition Stability
- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



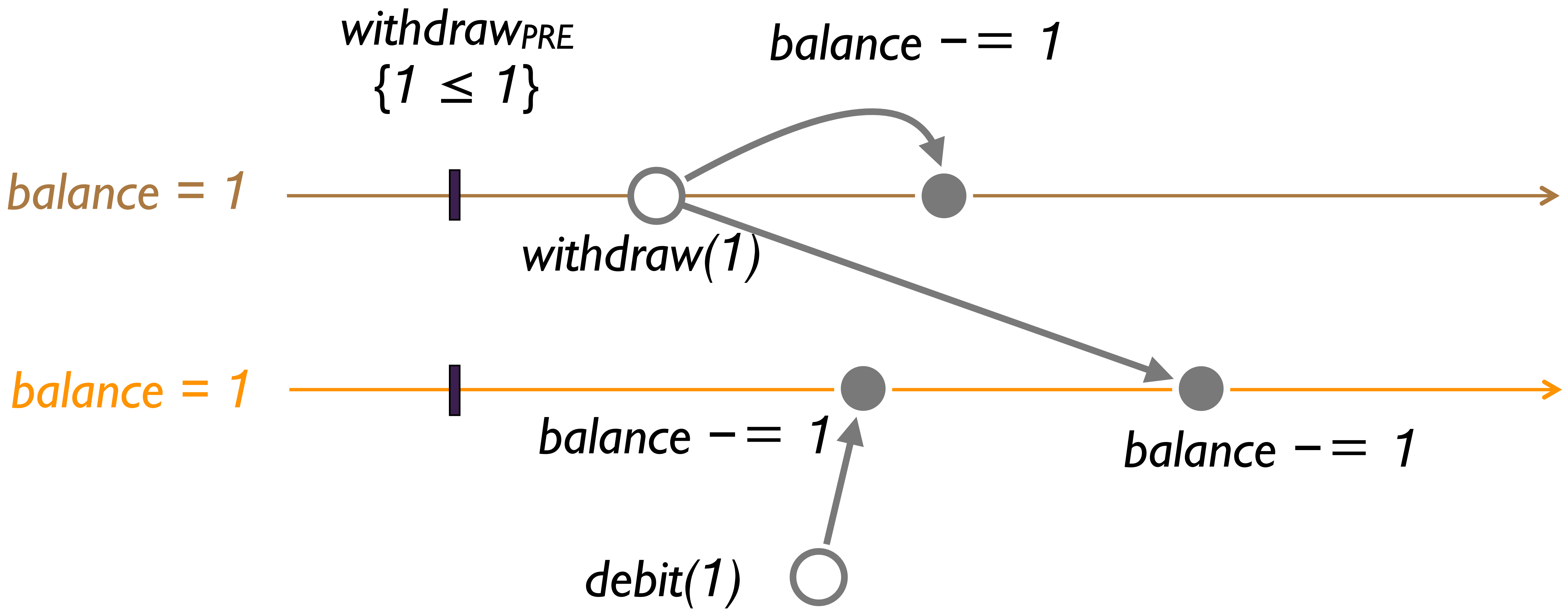
CISE Rules

1: Sequential correctness
- Any single operation maintains the invariant

2: Convergence
- Concurrent effectors commute

3: Precondition Stability
- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

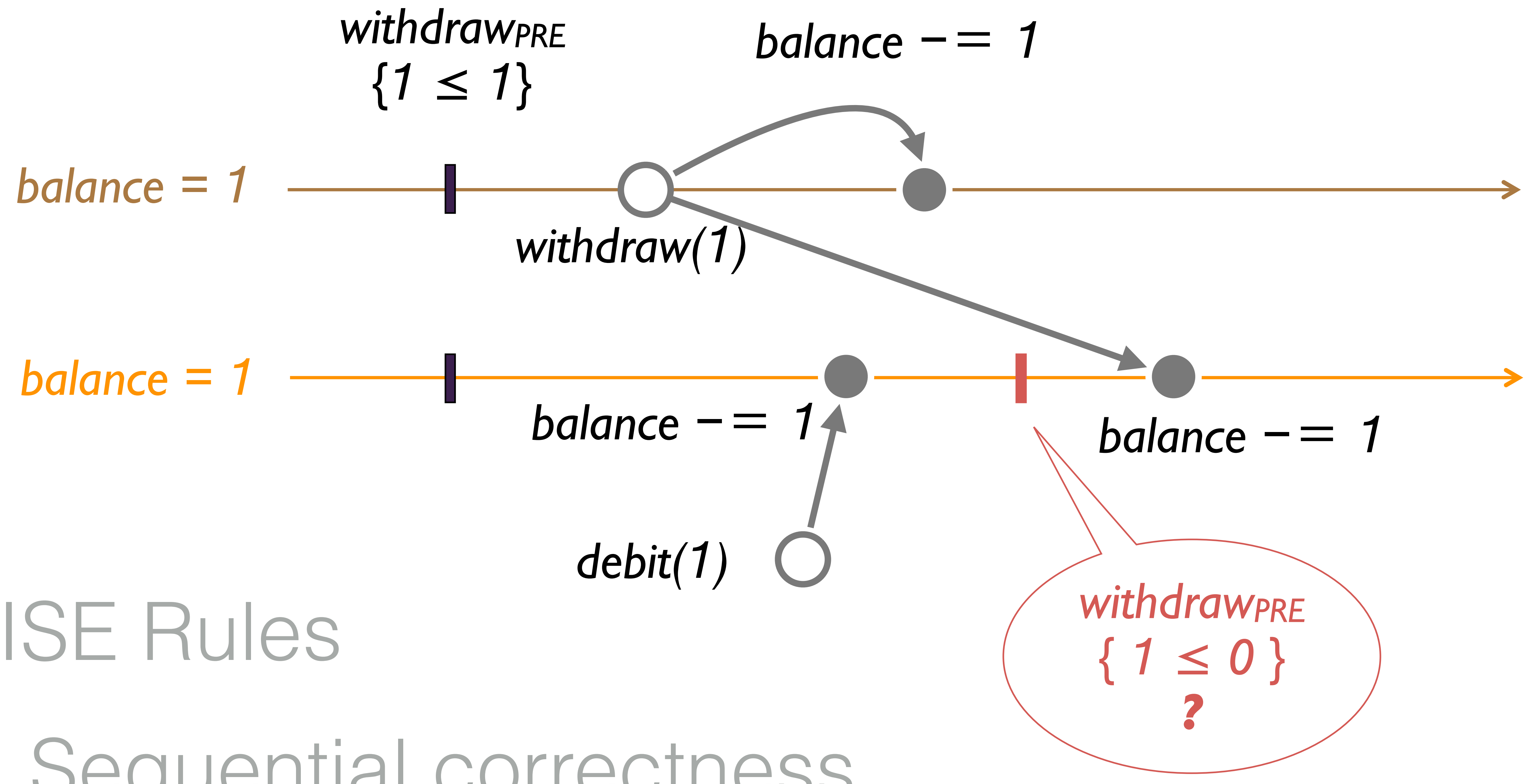
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

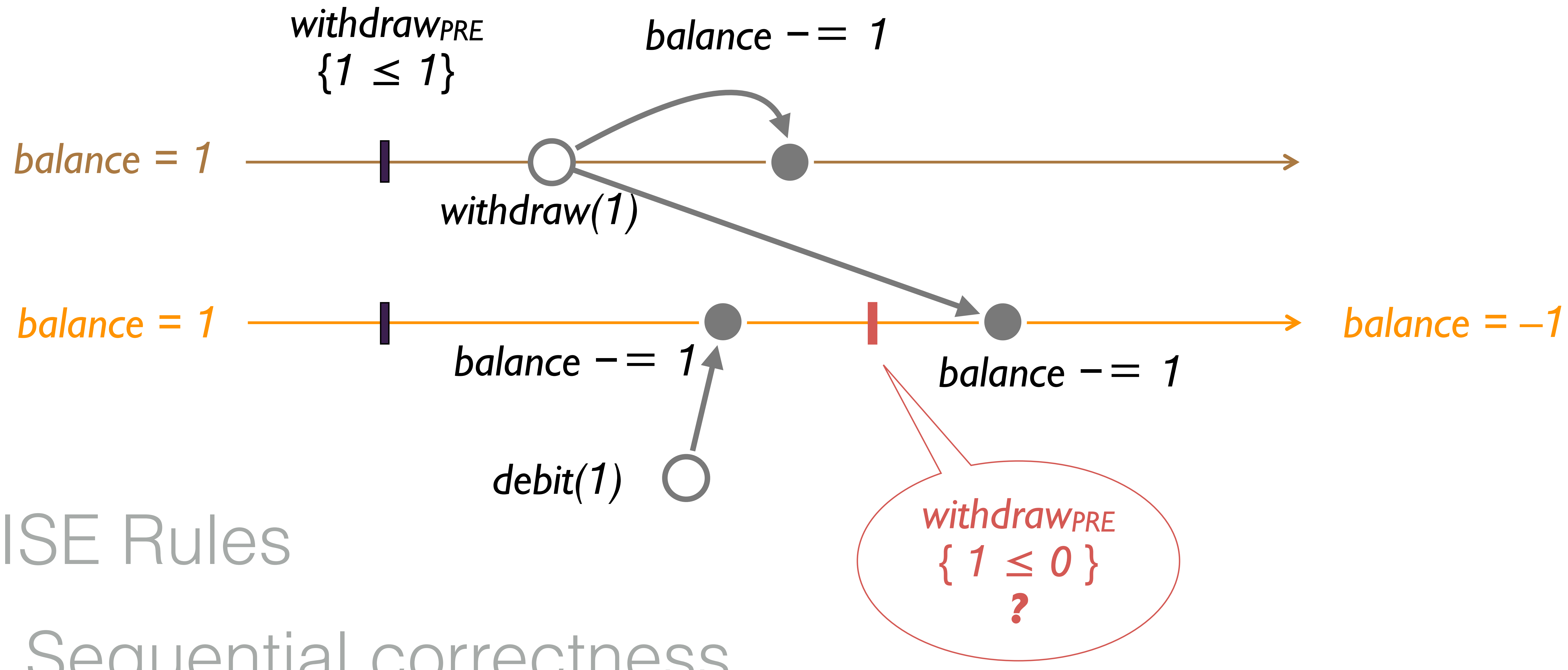
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

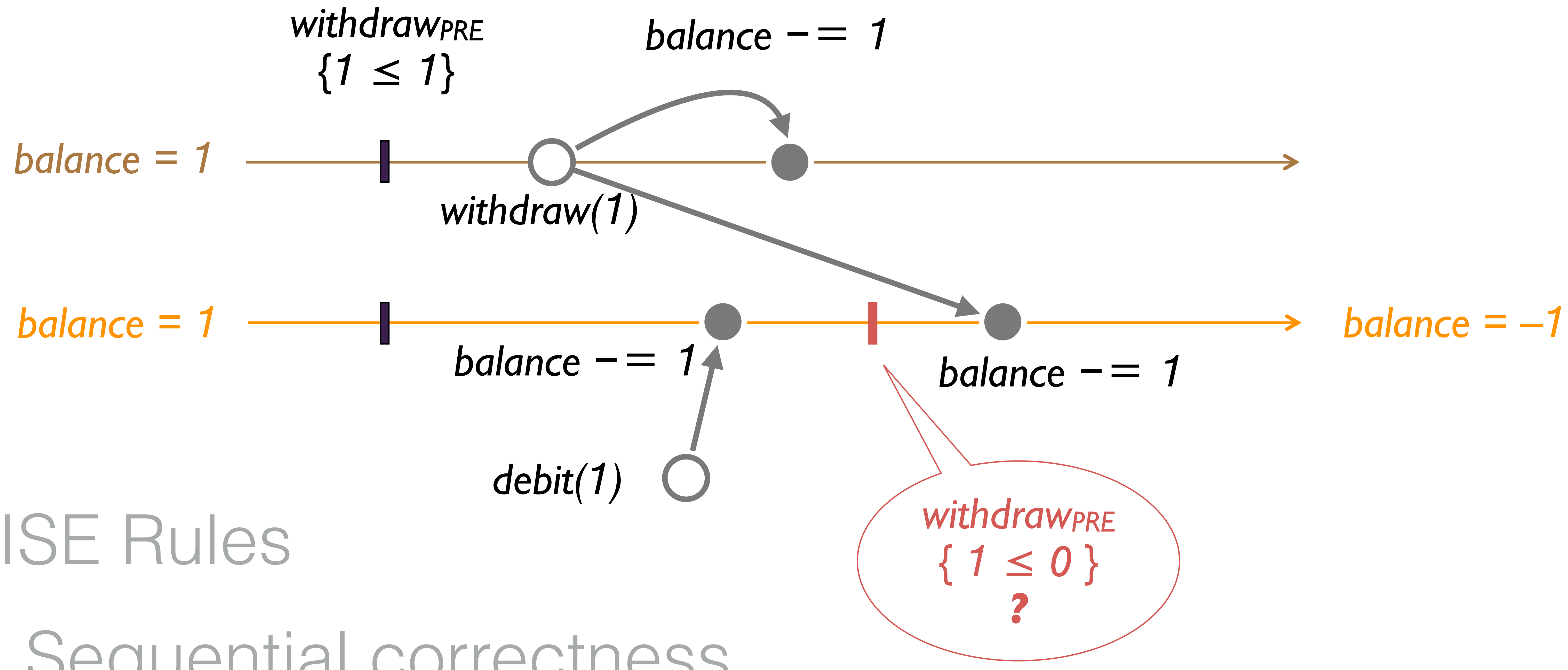
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

Fix: concurrency control

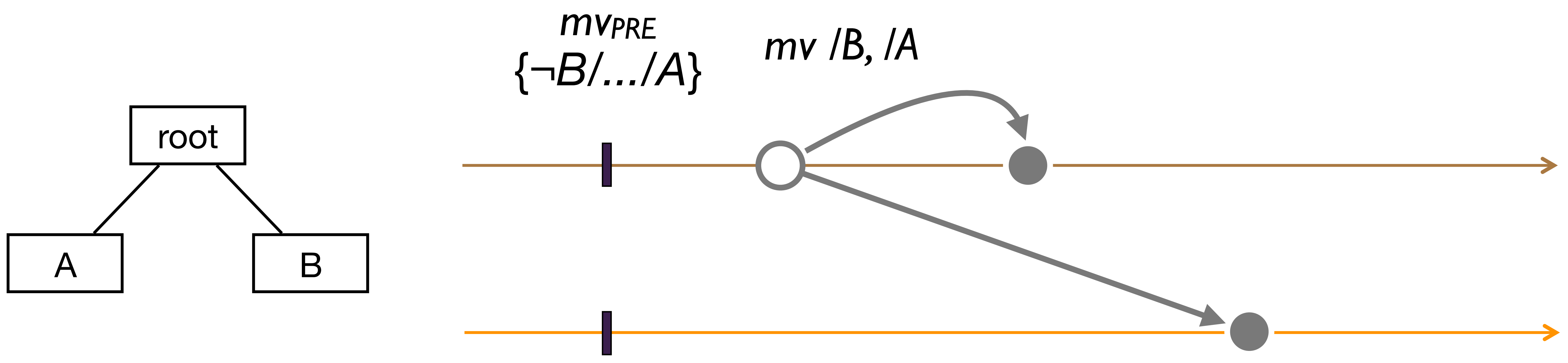
If satisfied: invariant is guaranteed

Advanced example: file system

- Operations: *mkdir, rmdir, mv, write*, etc.
- Invariant: Tree
  - Rule 1 ⟶ precondition on *mv*
    "May not **move** *node under self"*
  - Rule 2 ⟶ Use CRDTs for *write* || *write*
  - Rule 3 ⟶ *mv* || *mv* precondition unstable

Advanced example: file system

- Operations: ***mkdir***, ***rmdir***, *mv*, ***update***, etc.
- Invariant: Tree
    - Rule 1 ⟶ precondition on ***mv***
      *"May not **move** node under self"*
    - Rule 2 ⟶ Use CRDTs for ***update*** || ***update***
    - Rule 3 ⟶ *mv* || *mv* precondition unstable



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

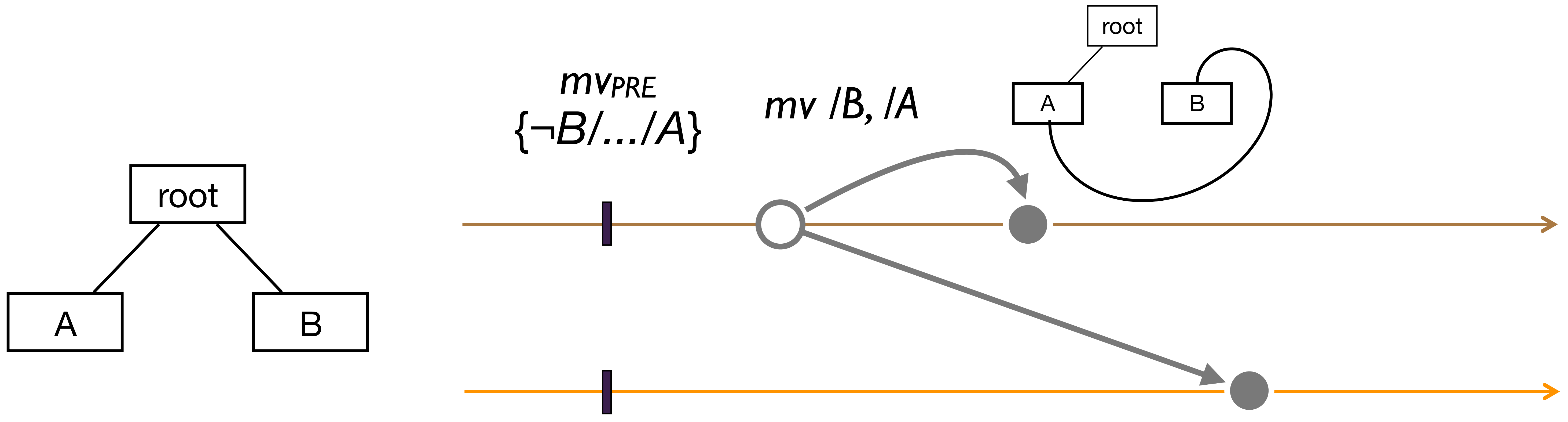
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

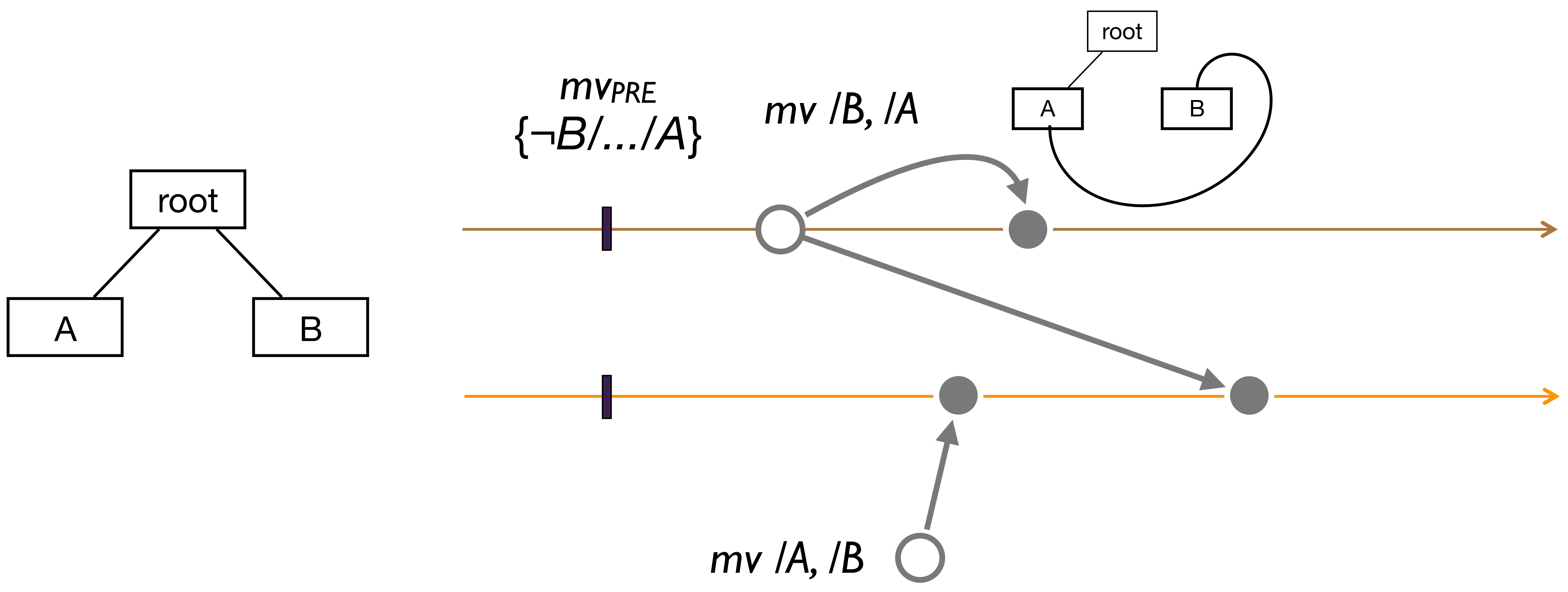
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

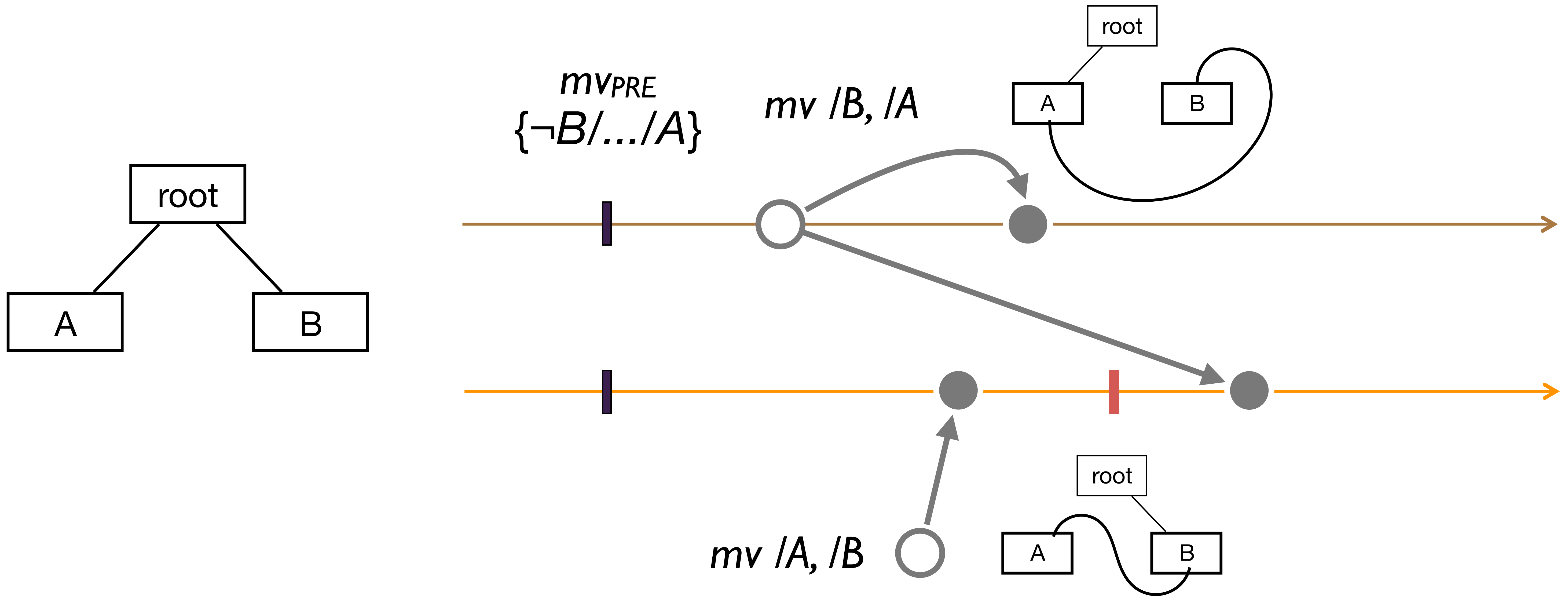
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

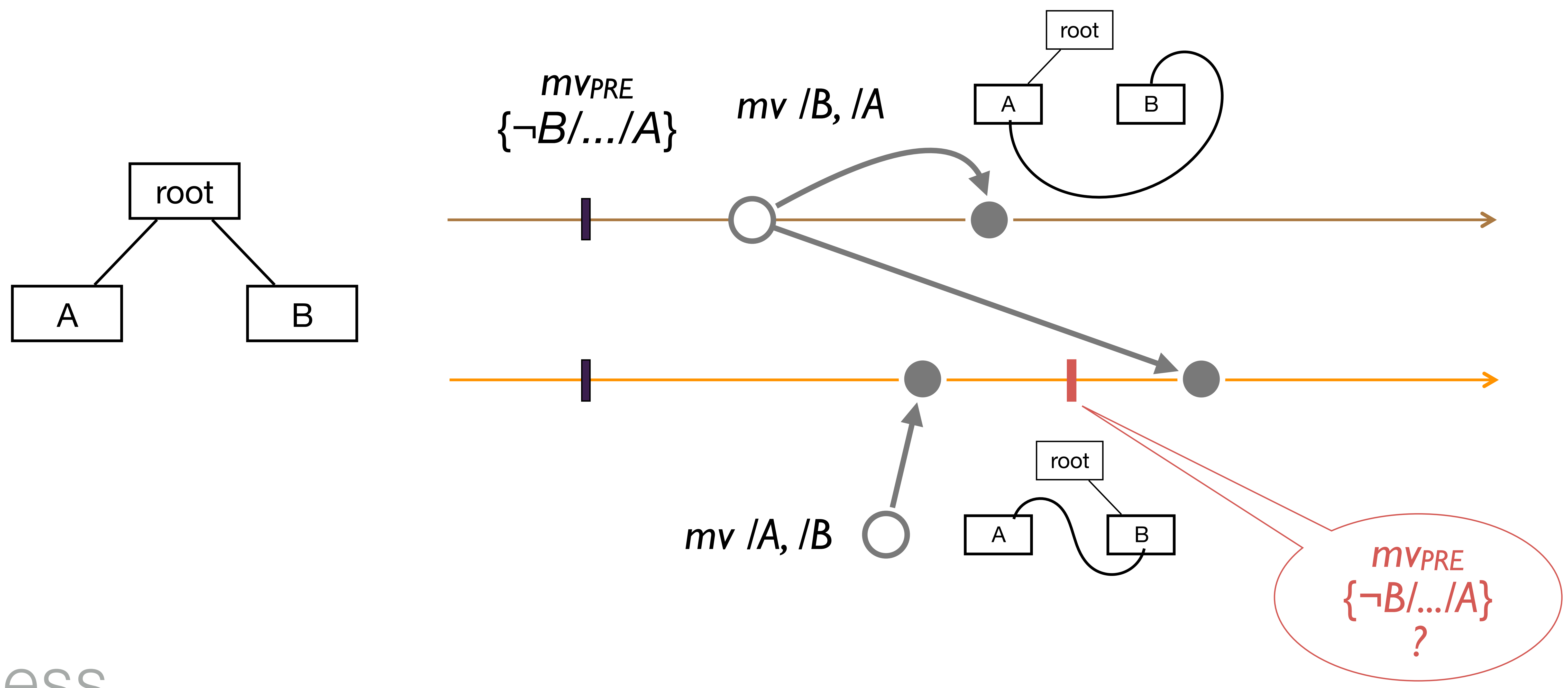
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

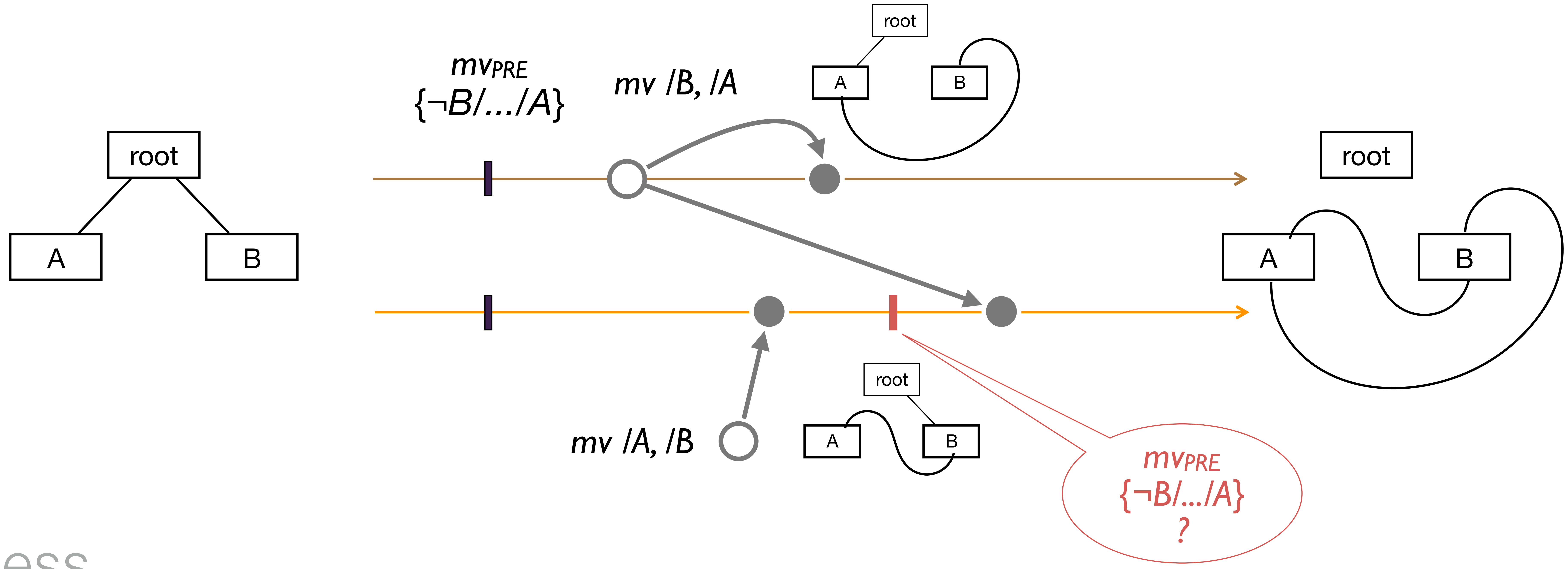
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invariant

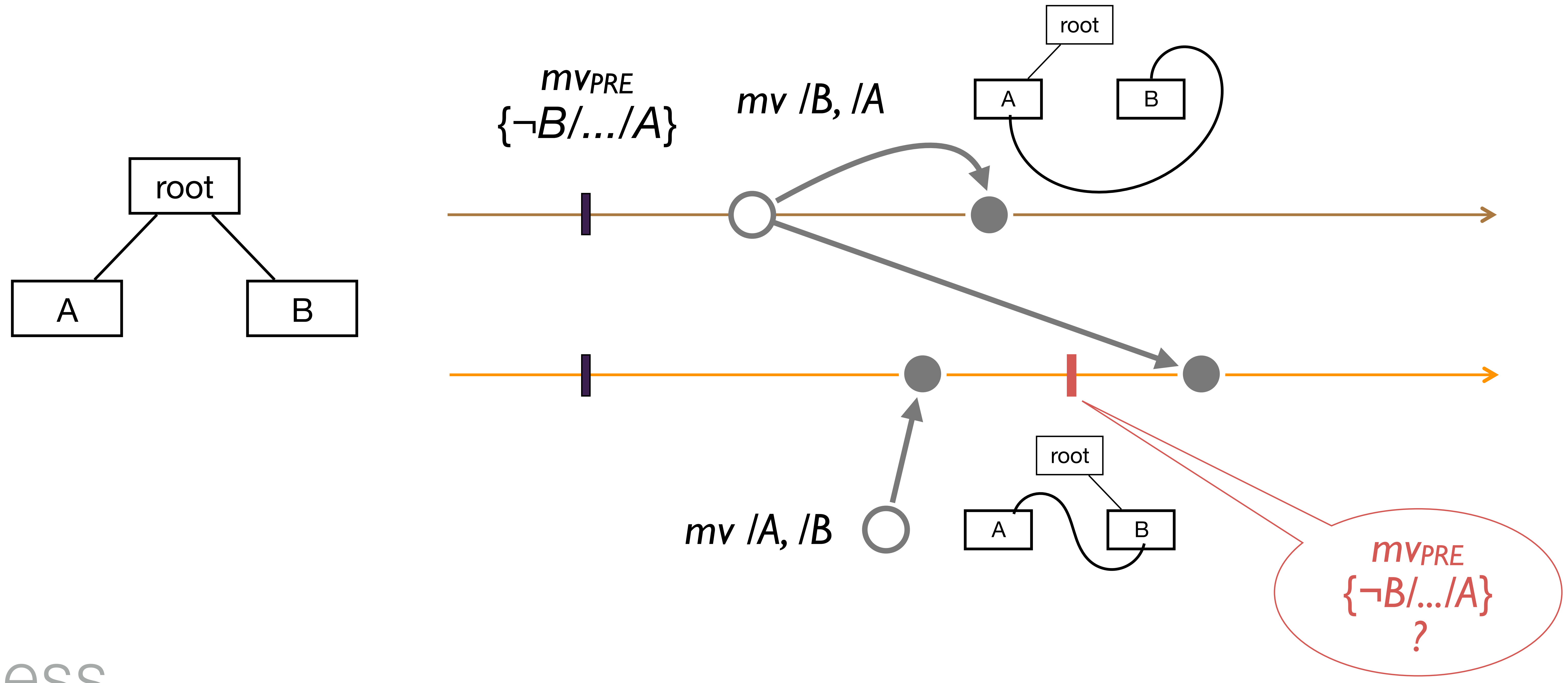
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness

- Any single operation maintains the invaria[nt]

Fix: concurrency control

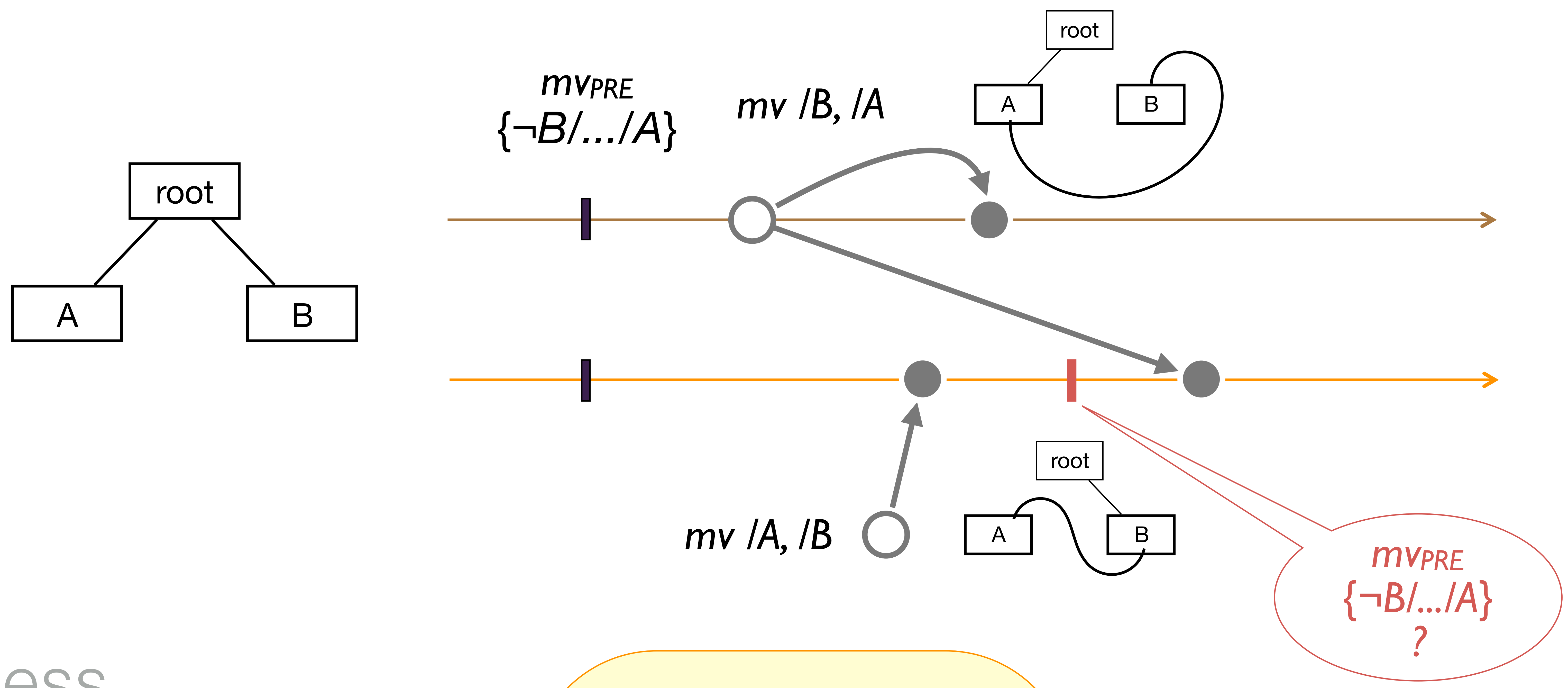
2: Convergence

- Concurrent effectors commute

3: Precondition Stability

- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed



CISE Rules

1: Sequential correctness
- Any single operation maintains the i

2: Convergence
- Concurrent effectors commute

3: Precondition Stability
- Every precondition is stable under every concurrent operation

If satisfied: invariant is guaranteed

You can have your cake and eat it too

CISE: The tool

Version of the tool (CEC) by Sreeja Nair

Related Problems

- Going beyond single invariants
  - Verify Pre/Post conditions of client programs
- State-Based implementations of CRDTs
- Composition of CRDTs
- … and much more :-)

The END