Reasoning About Concurrent Data Types

Constantin Enea

IRIF, Université de Paris, CNRS, IUF

European Research Council

Established by the European Commission

(Sequential) Data Types

Abstractions to simplify the manipulation of high-quantity data: objects (instances) + operations

- **Queue** = **enqueue**(value) + **dequeue**( ) => value
- Stack, Set, Key-value map, …

Specifications of data types:

- API documentation

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
E	**element**() Retrieves, but does not remove, the head of this queue.
boolean	**offer**(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E	**peek**() Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or returns `null` if this queue is empty.
E	**remove**() Retrieves and removes the head of this queue.

(Sequential) Data Types

Abstractions to simplify the manipulation of high-quantity data: objects (instances) + operations

- **Queue** = **enqueue**(value) + **dequeue**( ) => value
- Stack, Set, Key-value map, …

Specifications of data types:

- API documentation
- Pre/Post conditions in Hoare logic (Abstract Data Types):
  - { Seq } **enqueue**(value) { Seq :: value }
  - { value :: Seq } **dequeue**() => value { Seq }

Concurrent Data Types

Basic blocks of software that needs to process data in parallel

Concurrent Data Types: operations can be invoked in parallel from different threads or sites in a network

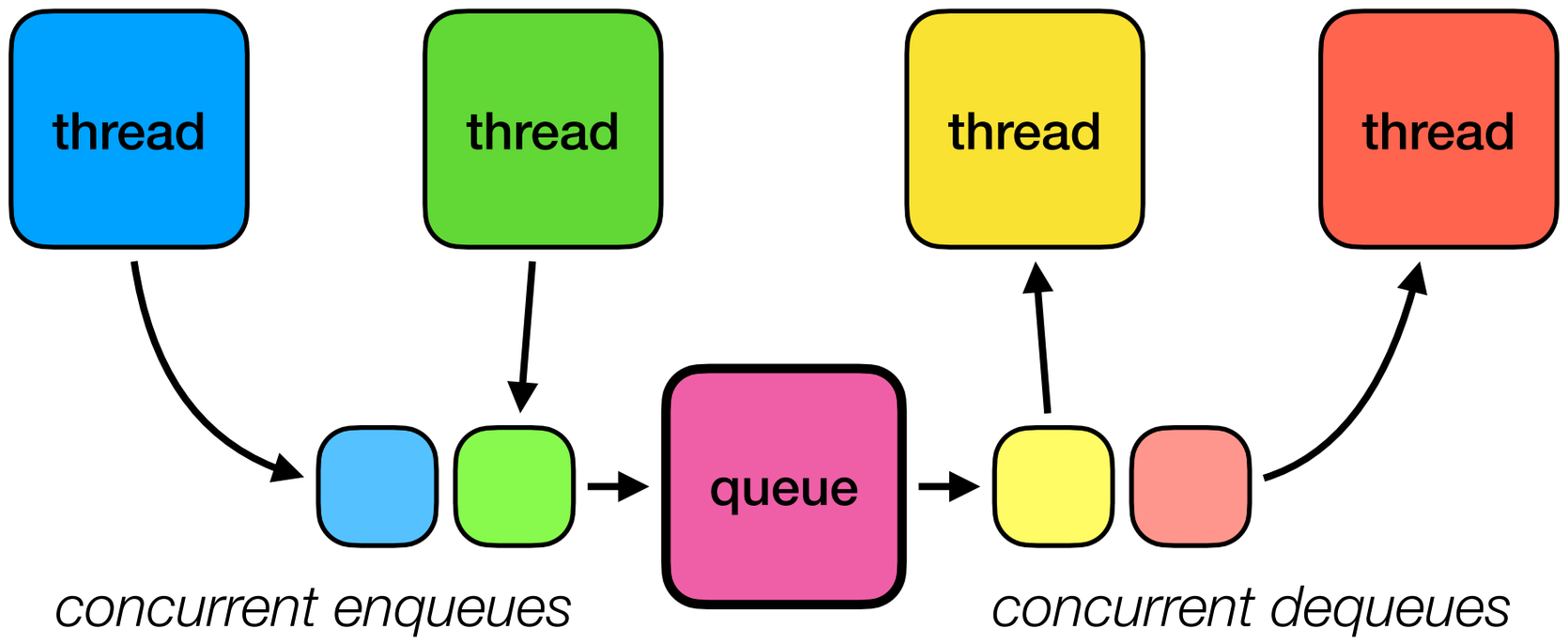
Support high-frequency parallel accesses to high-quantity data

Deployed over a shared-memory or a network

Formal specification and verification ?

Concurrent Objects

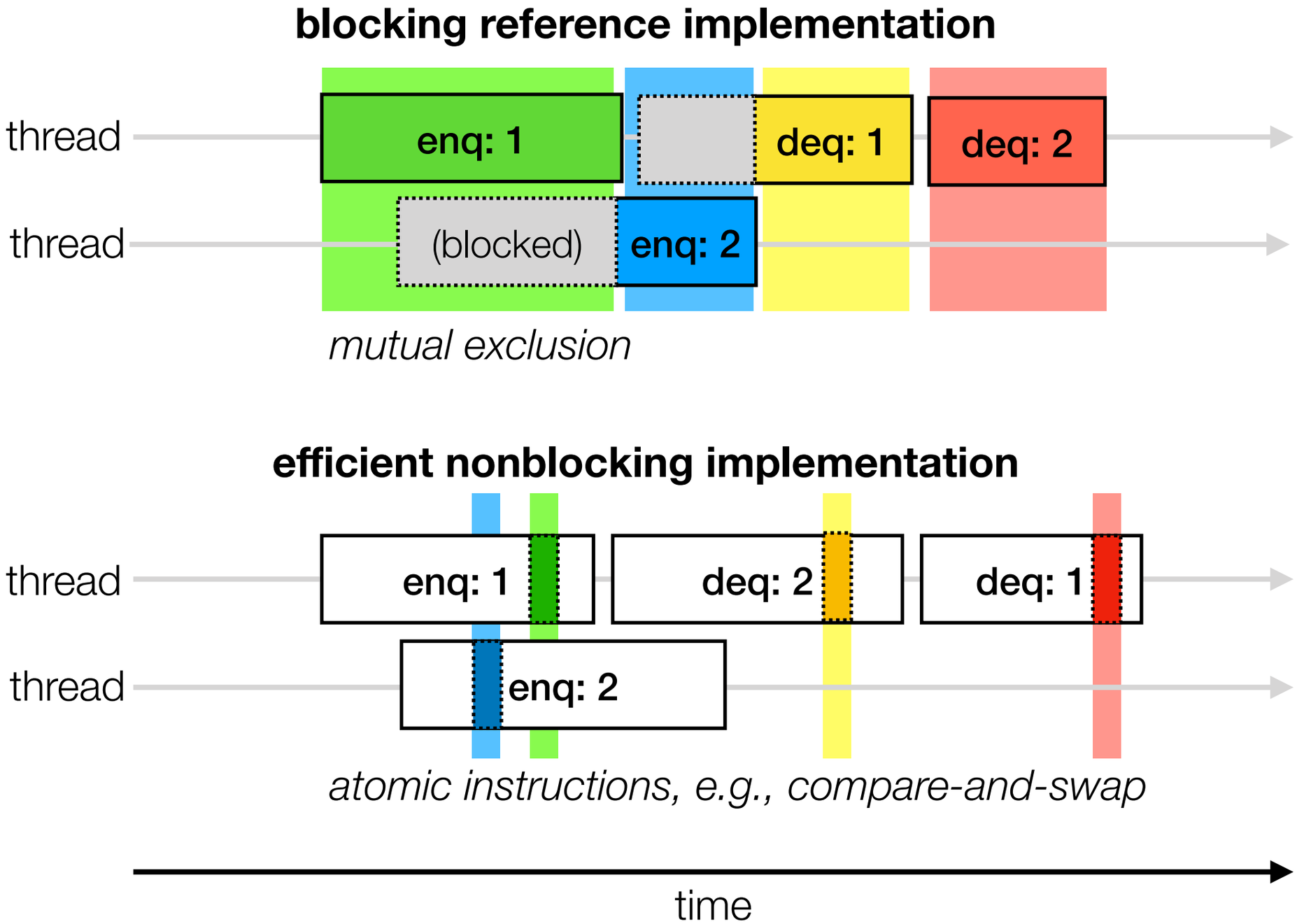
Multi-threaded programming



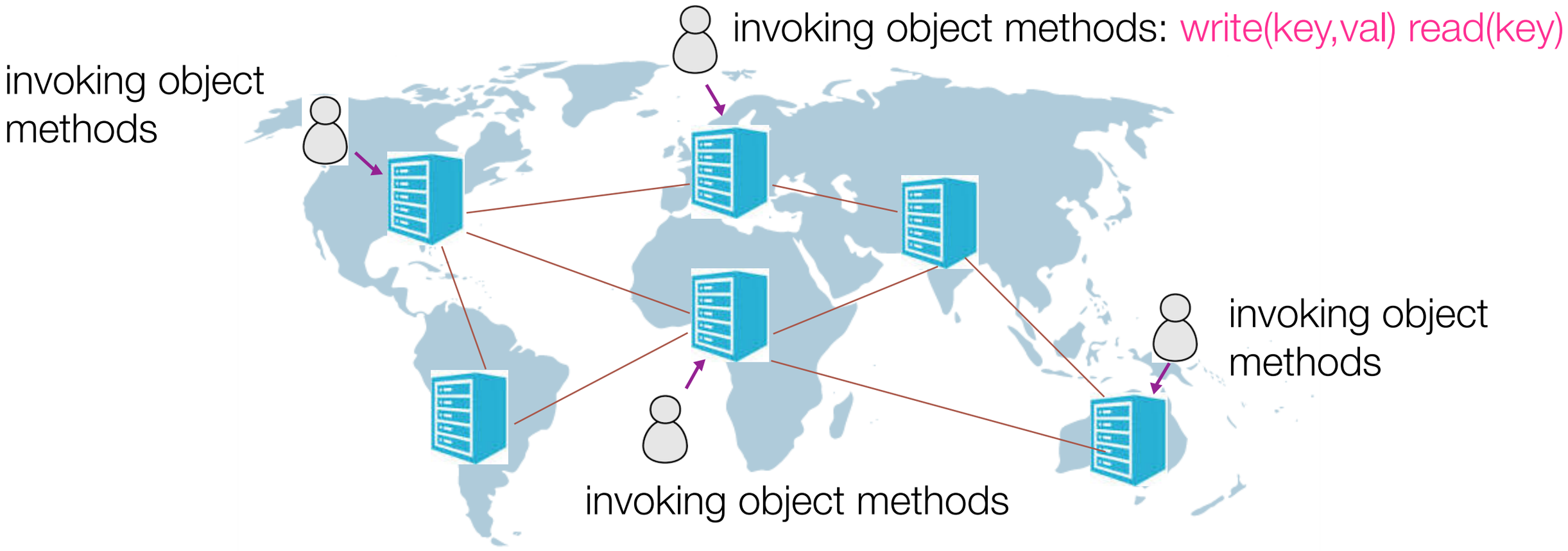
e.g. Java Development Kit SE

dozens of objects, including queues, maps, sets, lists, locks, atomic integers, …

Lock-Free Implementations



Replicated Objects (NoSQL)



To support failures, the state of the object is **replicated**

For availability, replicas may store different versions: **weak consistency**

CAP theorem: No replicated object is strongly **C**onsistent, highly-**A**vailable, and **P**artition-tolerant

Instances: key-value stores (Amazon Dynamo,Cosmos DB), CRDTs

Concurrent Data Types

Consistency model + Data type semantics

Strong Consistency:

- sequential consistency: interleaving semantics
- linearizability: interleavings are consistent with real-time

Weak Consistency:

- eventual consistency: eventual propagation of updates
- session guarantees: consistency w.r.t. previous operations in the same session
- causal consistency: causally-related operations observed in the same order

Data Type Semantics:

- the effect of an operation (return value) in function of a past history of operations
- features of the sequential semantics + conflict resolution (replication)

This Talk

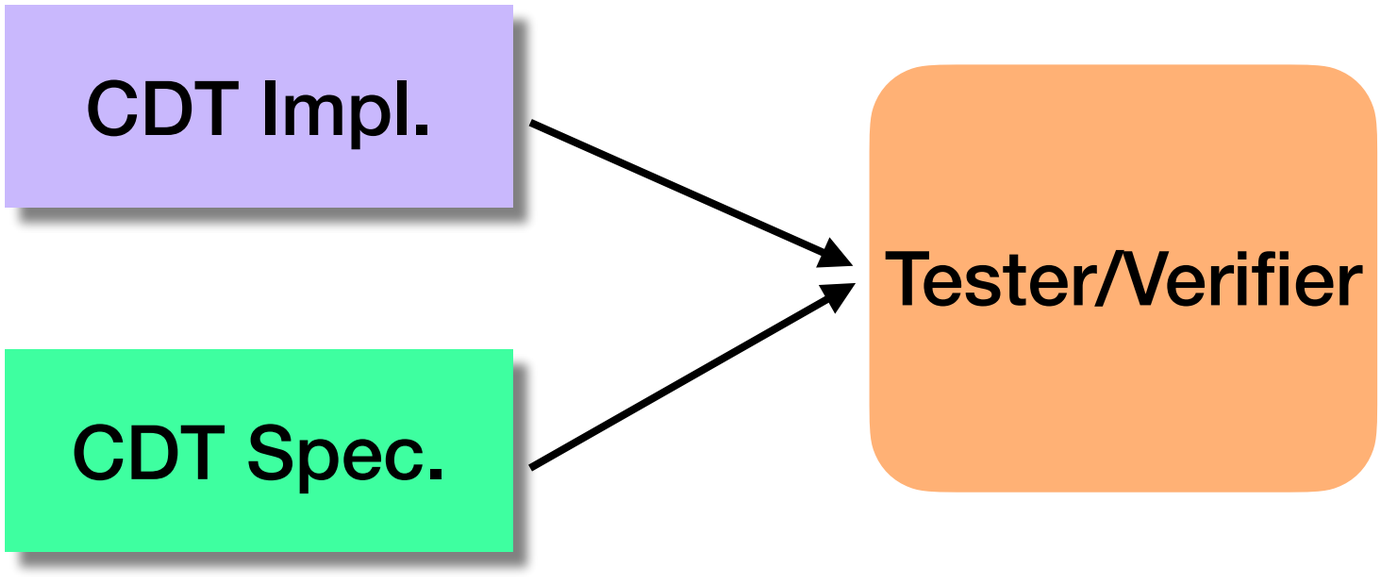
Investigating the complexity of **testing** and **verification** for CDTs

Testing is NP-complete [Gibbons.et.al.'97]: polynomial-time restrictions

Static Verification is undecidable [Alur et al.'96, ESOP'13, POPL'17]:

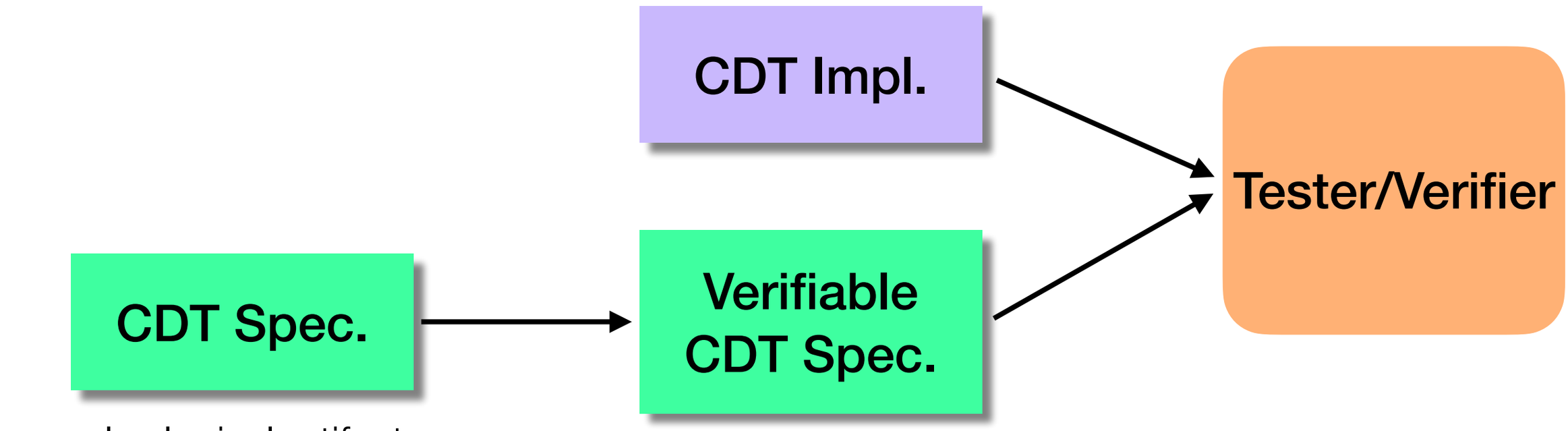
reductions to assertion-checking, decidable classes (complexity)

Approach: Verifiable Specs



complex logical artifacts:
e.g., second order quantification
over relations

Approach: Verifiable Specs



complex logical artifacts:
e.g., second order quantification over relations

e.g., acyclicity constraints
finite-state automata
reference implementations

Applied to classes of specifications and assuming mild constraints on implementations (data independence)

Approach: Verifiable Specs

Instantiations:

1. Concurrent collections (queues, stacks, etc) [ICALP'15, PLDI'15, POPL'18]
2. Replicated key-value stores [POPL'17]

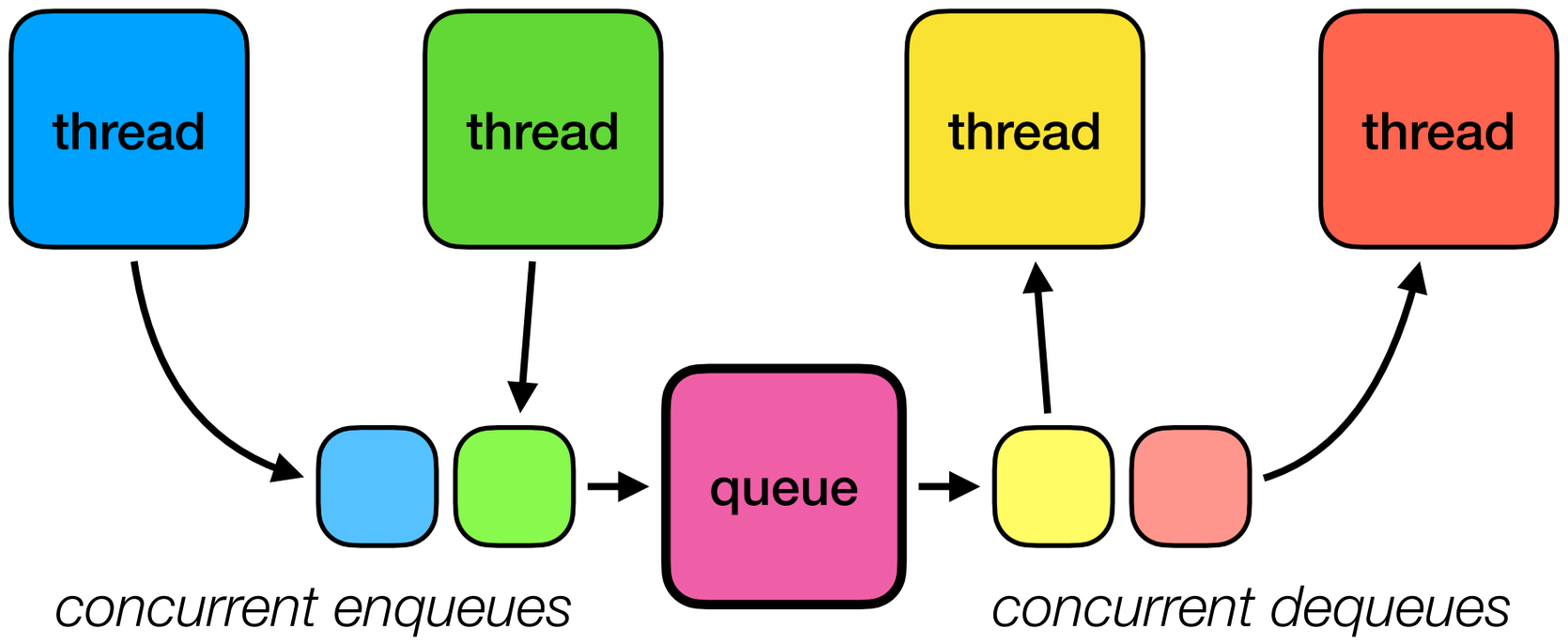
Approach: Verifiable Specs

Instantiations:

1. Concurrent collections (queues, stacks, etc) [ICALP'15, PLDI'15, POPL'18]
2. Replicated key-value stores [POPL'17]

Concurrent Objects

Multi-threaded programming



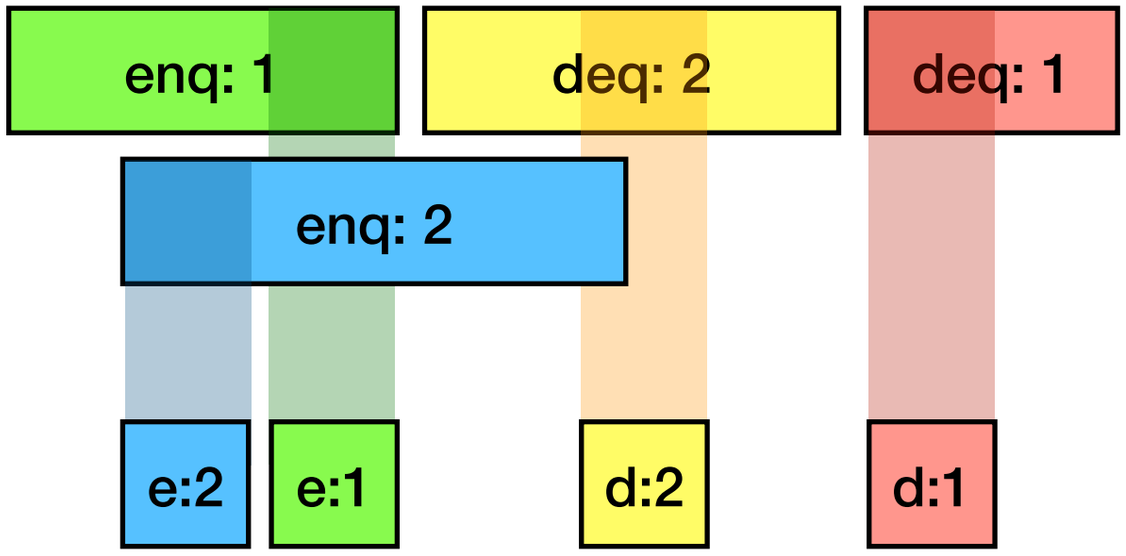
e.g. Java Development Kit SE

dozens of objects, including queues, maps, sets, lists, locks, atomic integers, …

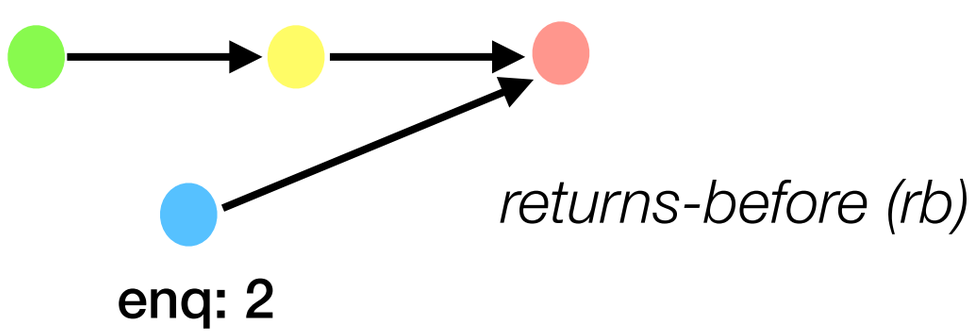
Linearizability [Herlihy&Wing'90]

Effects of each invocation appear to occur instantaneously

Execution history

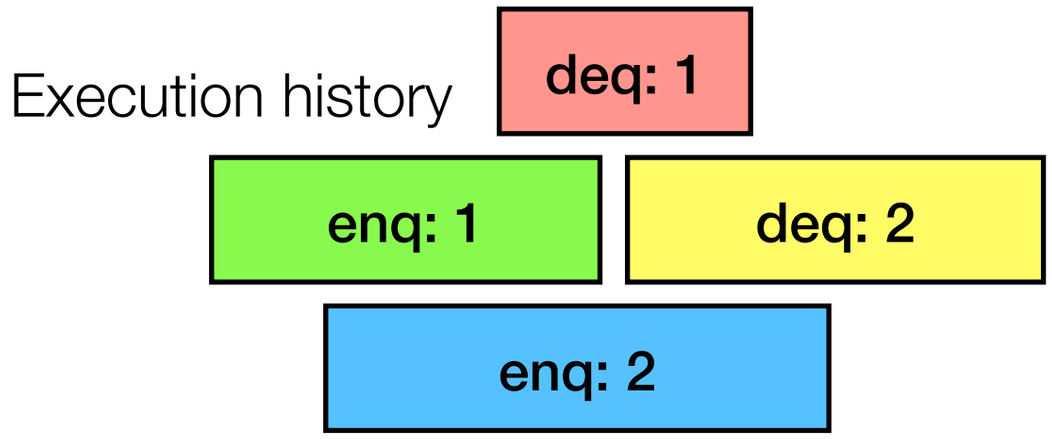


Linearization admitted by Queue ADT

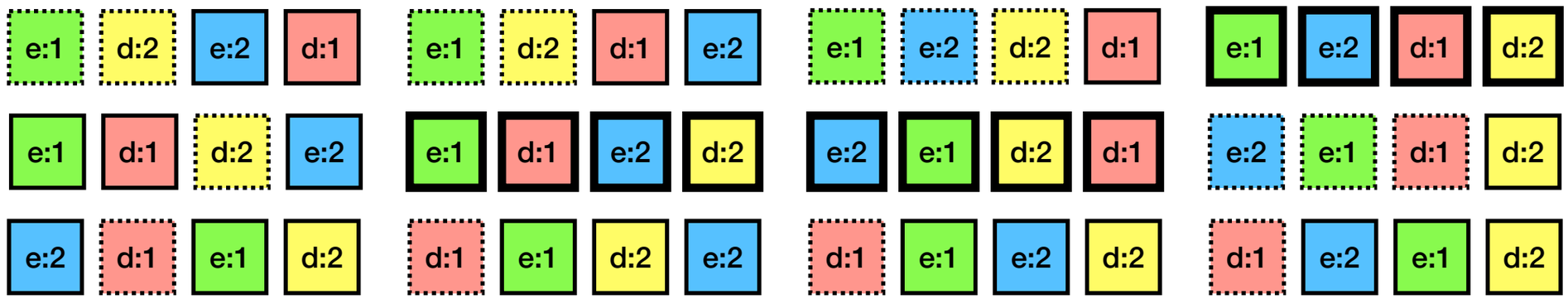


$\exists$ lin. rb $\subseteq$ lin $\wedge$ lin $\in$ Queue ADT

Complexity of Linearizability



Exponentially many linearizations to consider



Theorem [Gibbons.et.al.'97]

Checking linearizability for a fixed execution is NP-hard

Theorem [ESOP 2013]

Checking linearizability of a finite-state impl. is undecidable

Concurrent Queues [Henzinger et al'13] [ICALP'15]

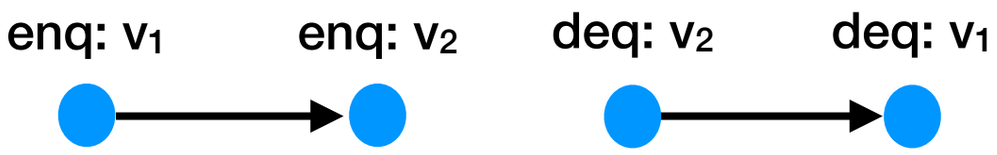
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"



Concurrent Queues [Henzinger et al'13] [ICALP'15]

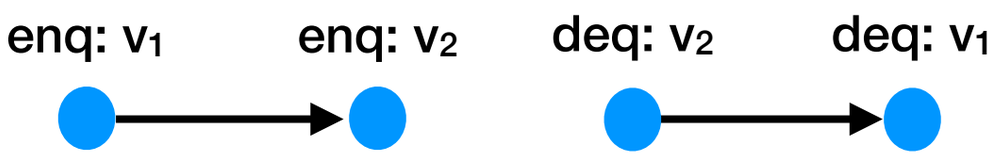
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

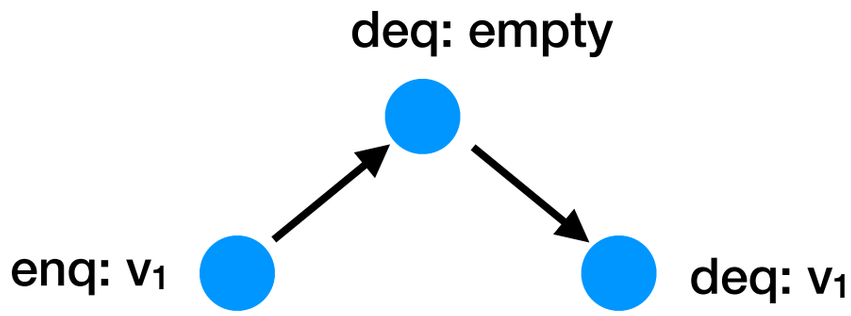
"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"



"Dequeue wrongfully returns empty"



Concurrent Queues [Henzinger et al'13] [ICALP'15]

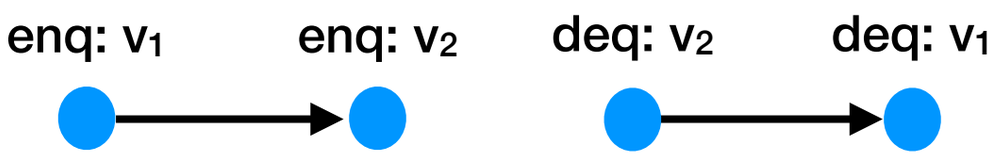
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

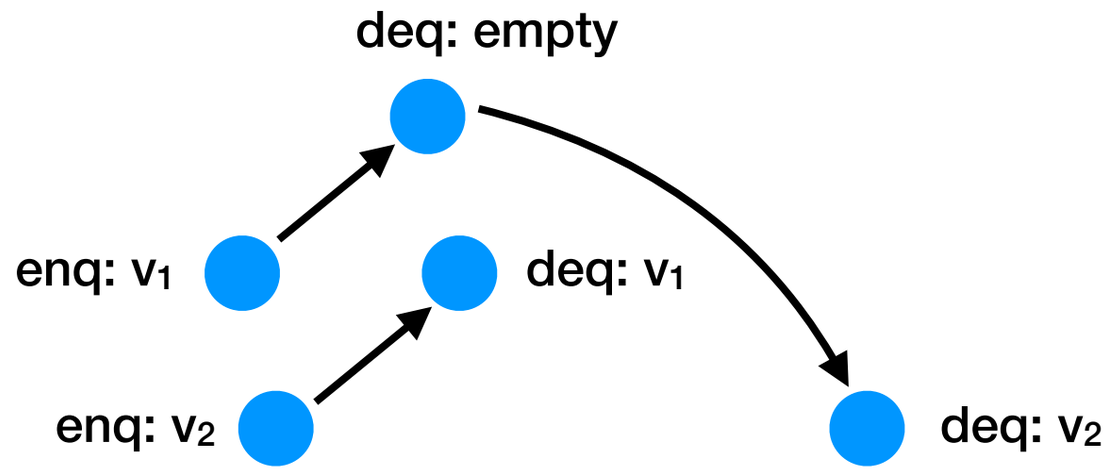
"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"



"Dequeue wrongfully returns empty"



Concurrent Queues [Henzinger et al'13] [ICALP'15]

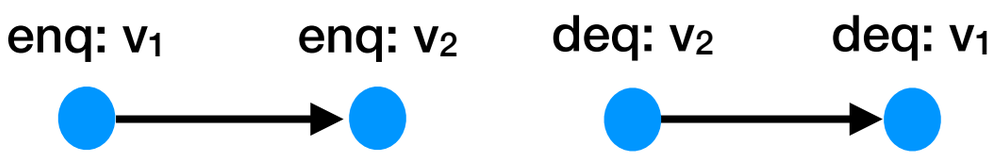
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

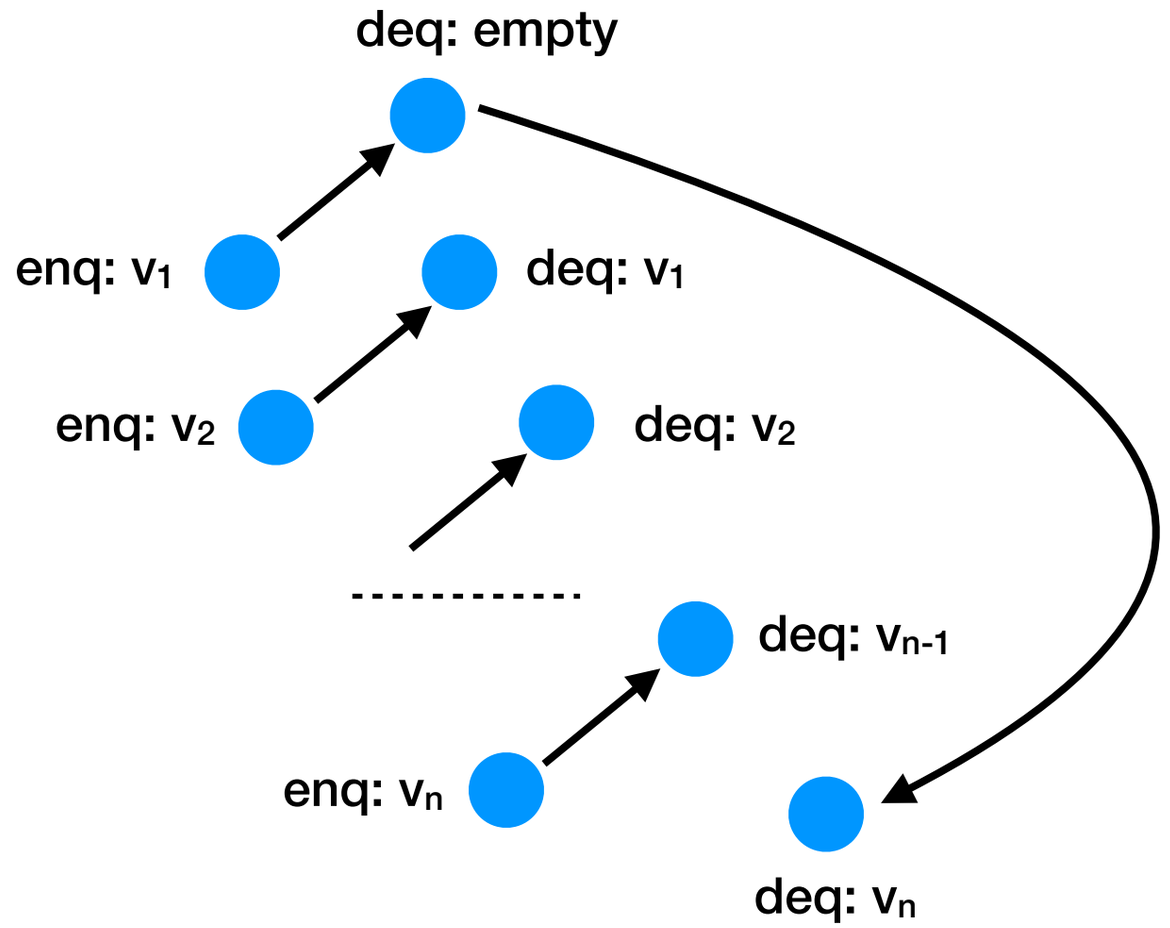
"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"



"Dequeue wrongfully returns empty"



Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

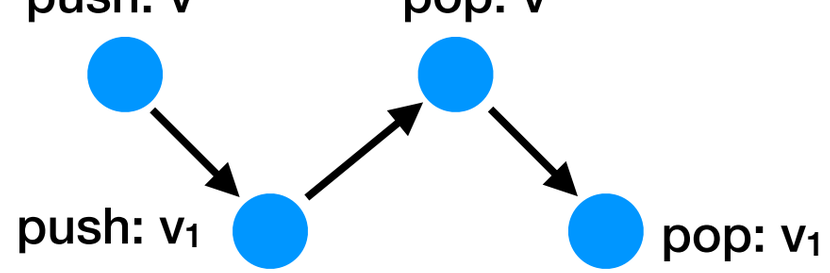
"Pop doesn't return the top of the stack"

"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

"Pop wrongfully returns empty"



Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

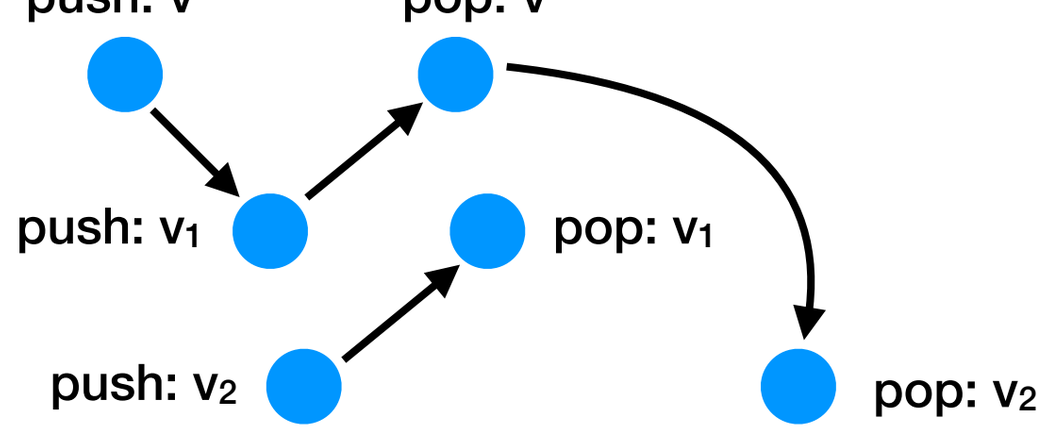
"Pop doesn't return the top of the stack"

"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

"Pop wrongfully returns empty"



Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

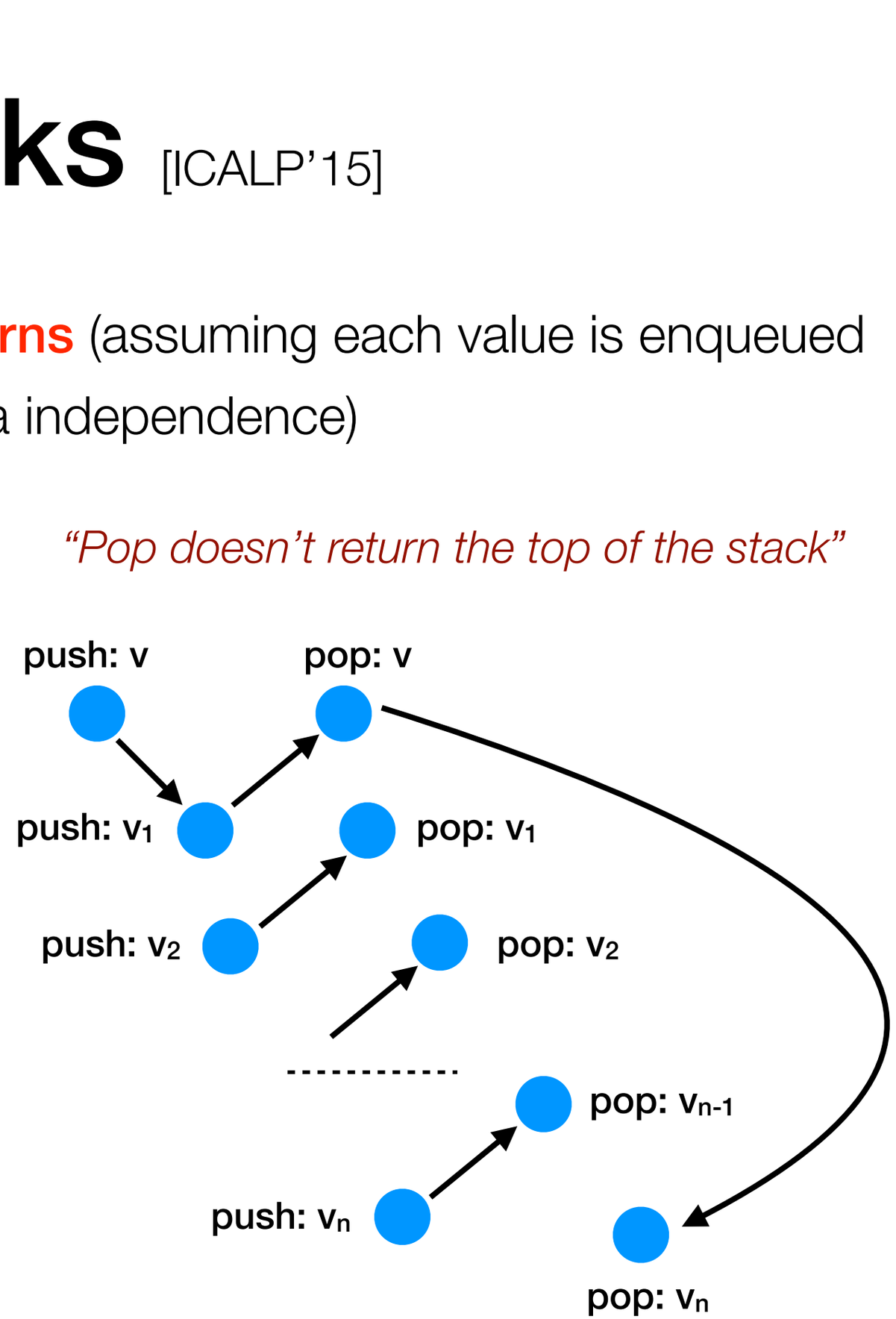
"Pop doesn't return the top of the stack"

"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

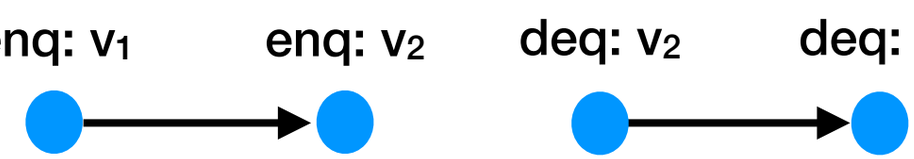
"Pop wrongfully returns empty"



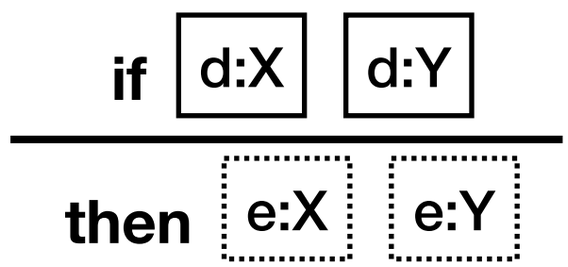
Bad Patterns - Testing

Using **bad patterns** to define inference rules for monotonic deductive inference (DATALOG)

Bad pattern

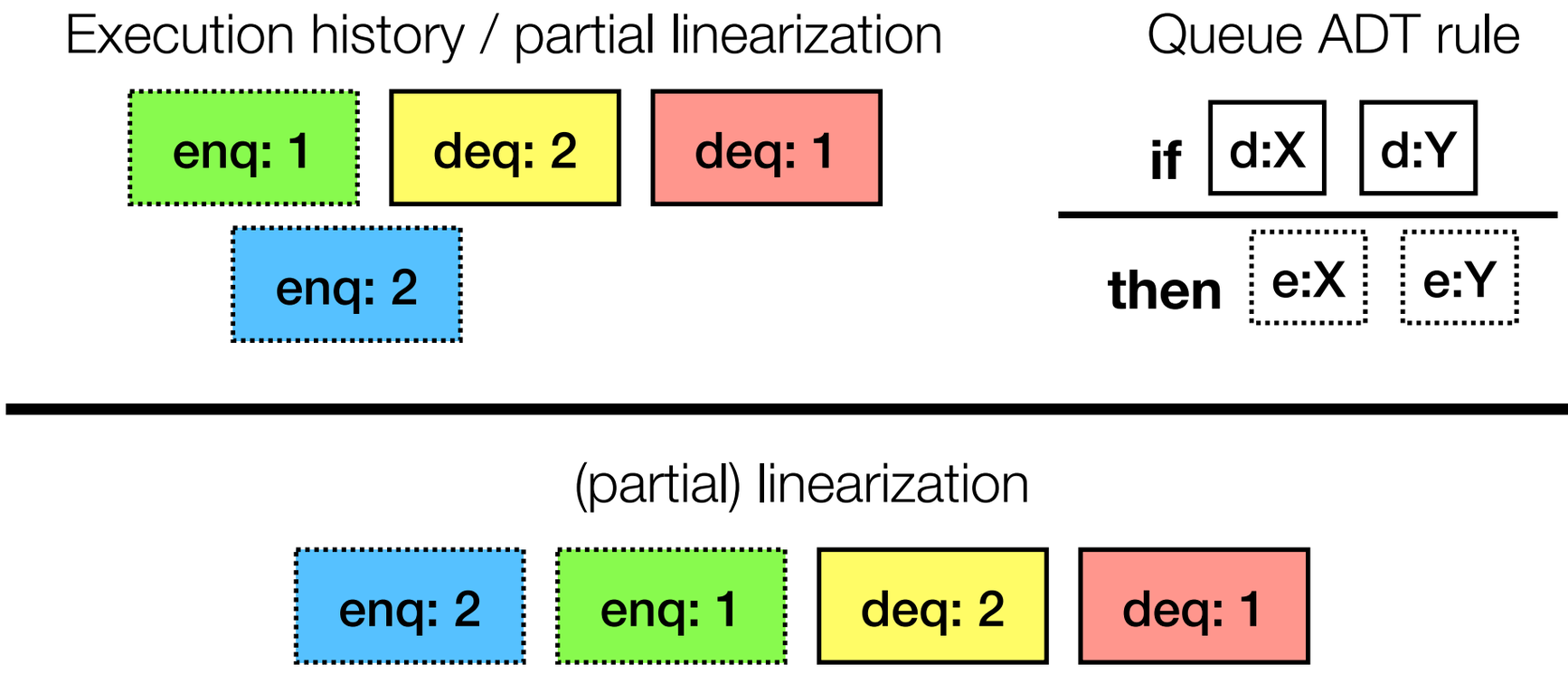


Queue ADT rule



Bad Patterns - Testing

Using **bad patterns** to define inference rules for monotonic deductive inference (DATALOG)



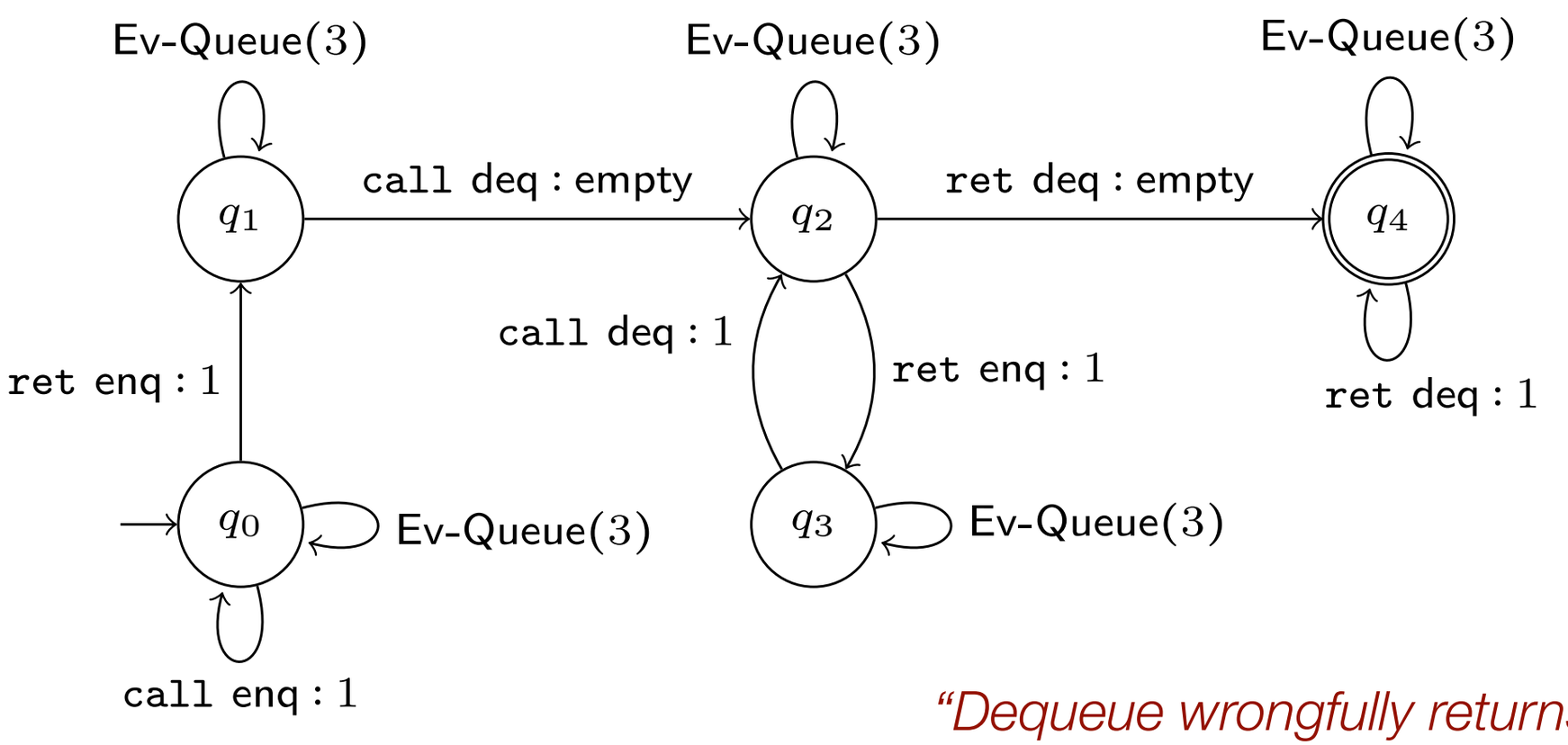
Theorem [PLDI'15, POPL'18]

Checking linearizability of an execution is polynomial-time for *collection types*

Bad Patterns - Verif. [ICALP'15]

Presence of **bad patterns** can be checked using **finite-state automata**

A **fixed** number of values suffices to expose all violations (data independence)



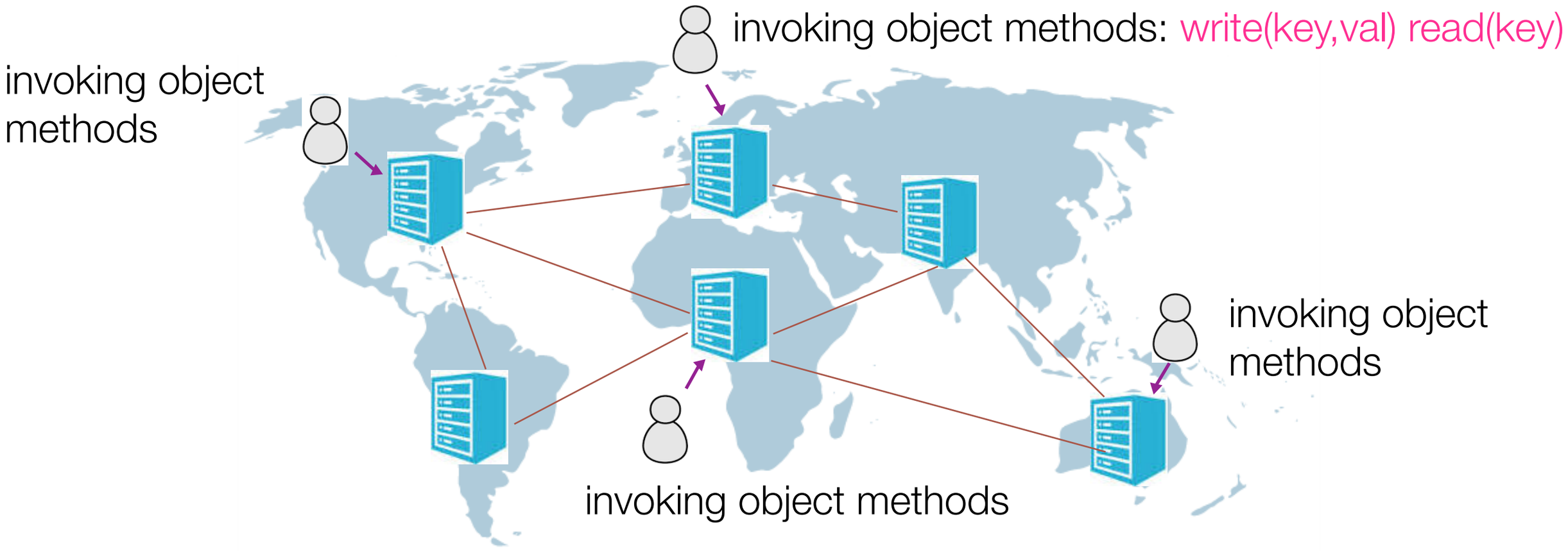
"Dequeue wrongfully returns empty"

Approach: Verifiable Specs

Instantiations:

1. Concurrent collections (queues, stacks, etc) [ICALP'15, PLDI'15, POPL'18]
2. Replicated key-value stores [POPL'17]

Replicated Objects (NoSQL)



To support failures, the state of the object is **replicated**

For availability, replicas may store different versions: **weak consistency**

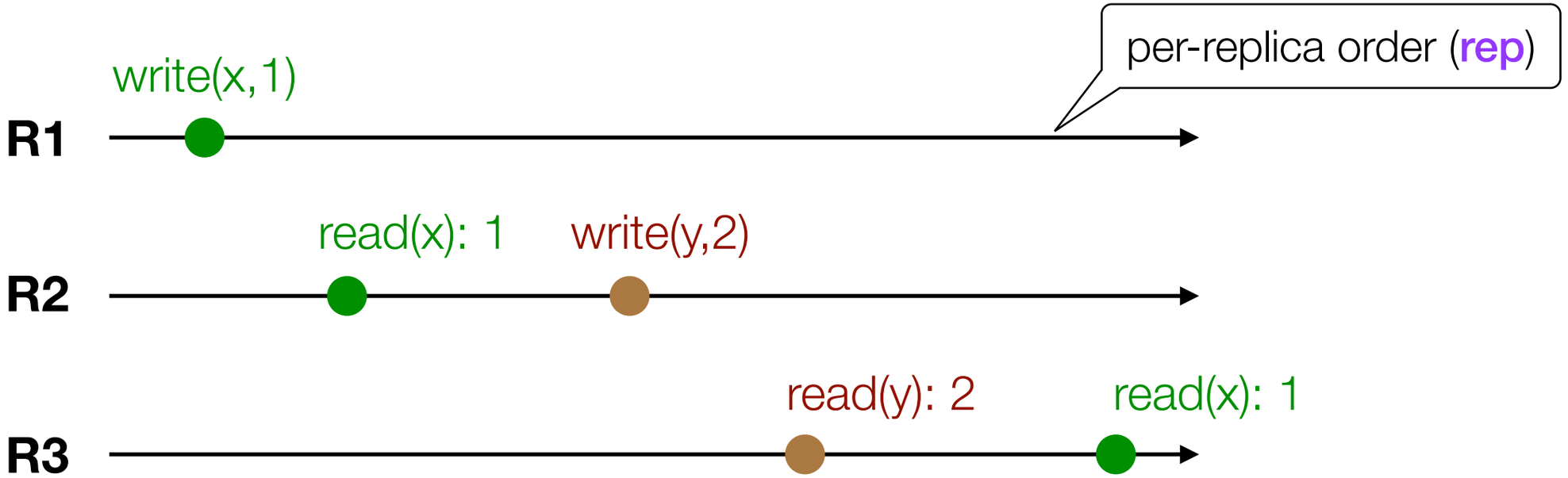
CAP theorem: No replicated object is strongly **C**onsistent, highly-**A**vailable, and **P**artition-tolerant

Instances: key-value stores (Amazon Dynamo,Cosmos DB), CRDTs

Weak Consistency Specs

[Burckhardt et al.'14]

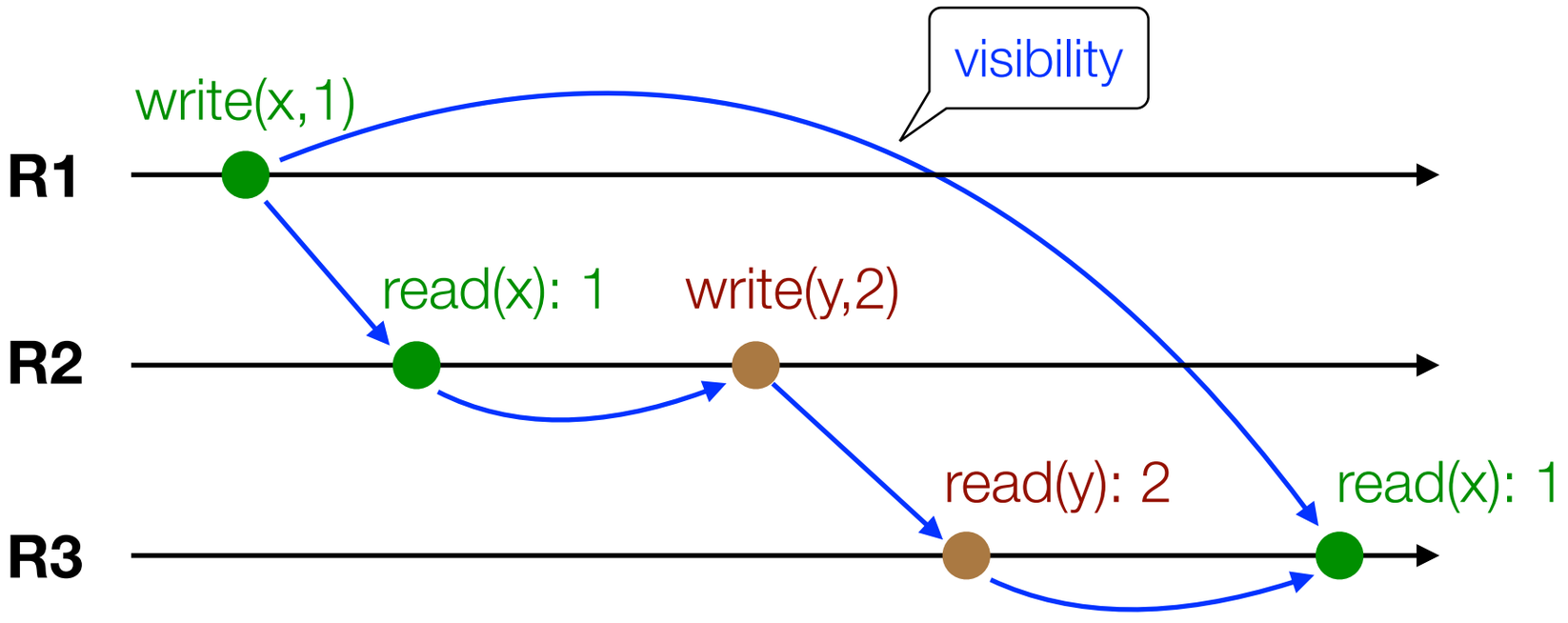
Specifying executions **as observed by a client**: operations + per-replica order (**rep**) (without internal events)



Weak Consistency Specs

[Burckhardt et al.'14]

Specifying executions **as observed by a client**: operations + per-replica order (**rep**) (without internal events)



Visibility relation: acyclic relation between ops. that extends per-replica order

Exec ⊧ **ConsistencyModel** iff

∃ **vis**. ∀ op. ``**return value** is consistent with the set of visible ops"

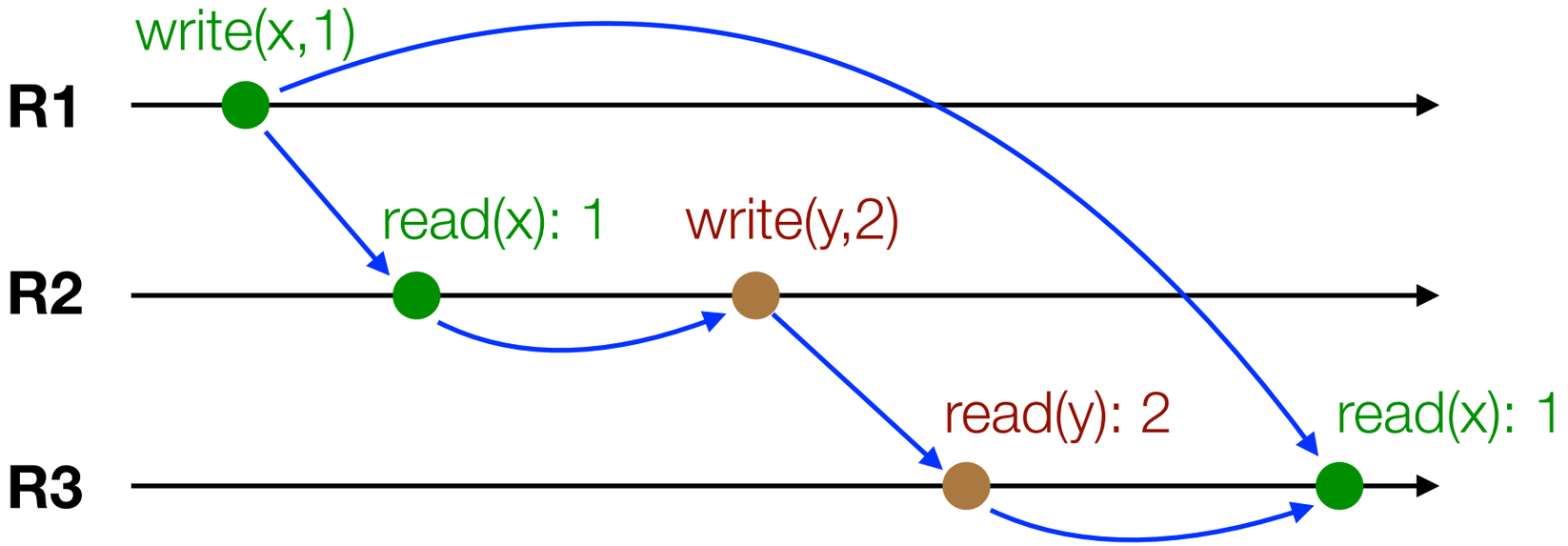
∧ **vis** ⊇ **rep** ∧ ordering properties

(ignoring the arbitration relation used to model conflict-resolution with timestamps)

Causal Consistency (CC)

[Lamport'78]

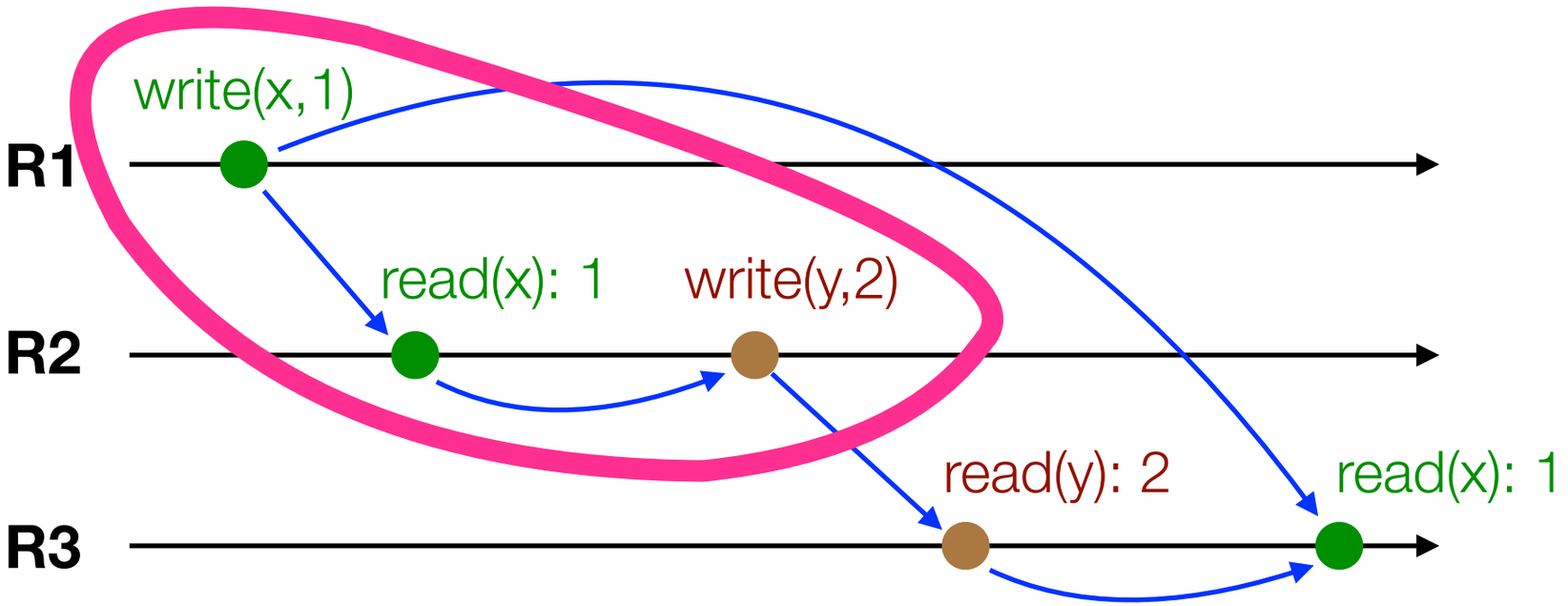
If an update is visible, then **all** the updates is **dependent** on should be also visible



Causal Consistency (CC)

[Lamport'78]

If an update is visible, then **all** the updates is **dependent** on should be also visible

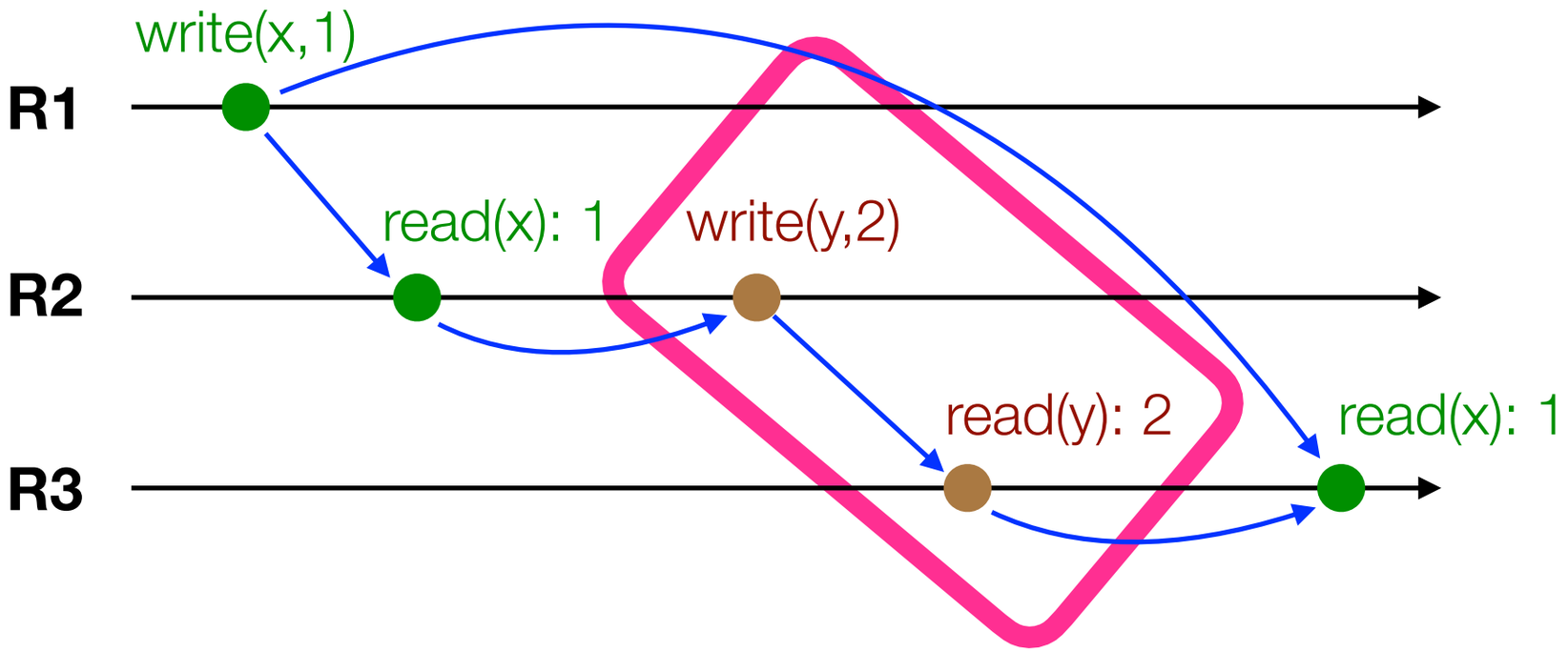


write(y,2) depends on write(x,1)

Causal Consistency (CC)

[Lamport'78]

If an update is visible, then **all** the updates is **dependent** on should be also visible



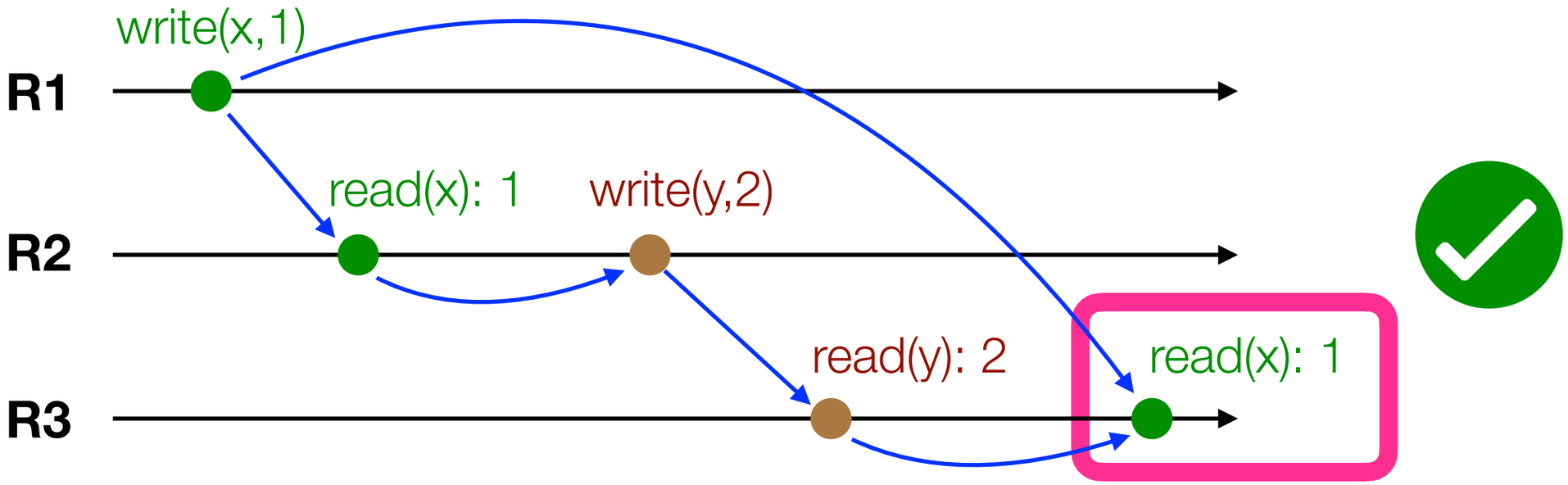
write(y,2) depends on write(x,1)

write(y,2) visible to **R3**

Causal Consistency (CC)

[Lamport'78]

If an update is visible, then **all** the updates is **dependent** on should be also visible



write(y,2) depends on write(x,1)

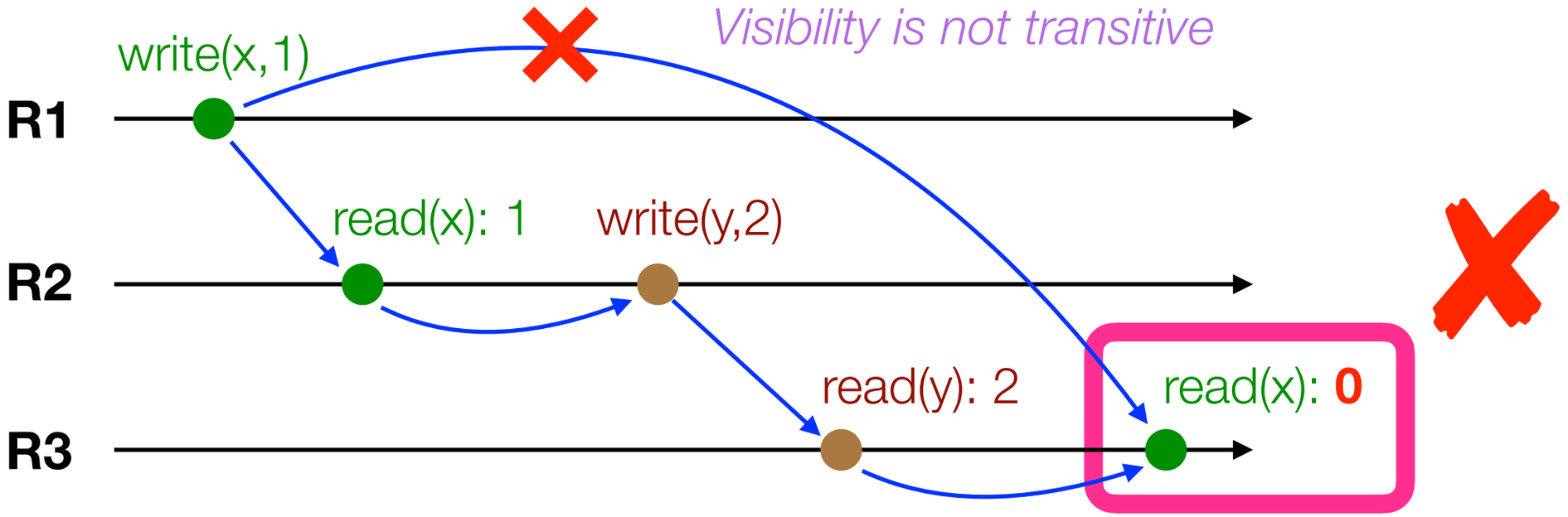
write(y,2) visible to **R3**

⇒ write(x,1) is also visible to **R3**

Causal Consistency (CC)

[Lamport'78]

If an update is visible, then **all** the updates is **dependent** on should be also visible



write(y,2) depends on write(x,1)

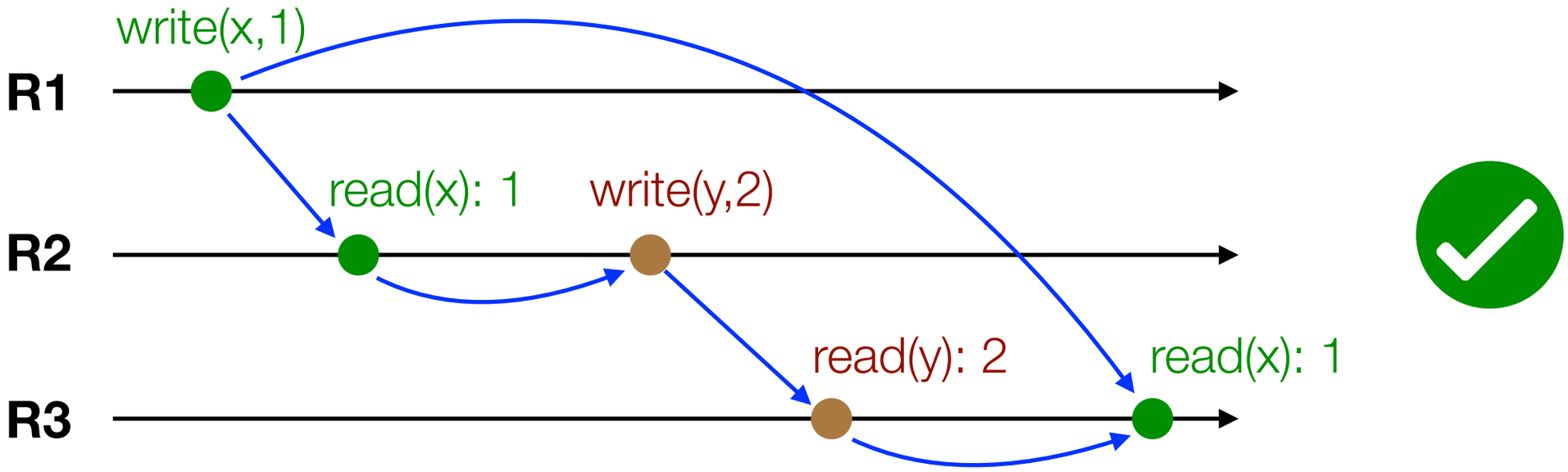
write(y,2) visible to **R3**

⇒ write(x,1) is ~~also visible~~ to **R3**

Causal Consistency (CC)

[Lamport'78]

If an update is visible, then **all** the updates is **dependent** on should be also visible



Exec ⊧ **CC** iff

∃ **vis**. ∀ op. ``**return value** is consistent with the set of visible ops"

∧ **vis** ⊇ **rep** ∧ **vis** is transitive

Causal Consistency (CC)

[Lamport'78]

If an update is visible, then **all** the updates is **dependent** on should be also visible

An instance of weak consistency: "anomalies" ≠ interleavings of operations

Empty inter-replica visibility

Possible interleavings: write(x,1); write(y,2); read(y): 2; read(x): 1

write(x,1); read(y): 0; write(y,2); read(x):1

…

(at least one read returns a non-zero value)

Checking CC: Th. Limits

Exec ⊧ CC iff

∃ **vis**. ∀ op. ``**return value** is consistent with the set of visible ops"

∧ **vis** ⊇ **rep** ∧ **vis** is transitive

Theorem. (Testing)

Checking CC for a single execution is NP-complete

Theorem. (Static verification)

Checking CC for a finite-state implementation and regular specification is undecidable

Showing that checking CC is more difficult than assertion checking.

What happens for **key-value stores** ?

CC: Key-Value Stores

∃ **vis**. ∀ read. ``**return value** → a maximal visible write”
∧ **vis** ⊇ **rep** ∧ **vis** is transitive

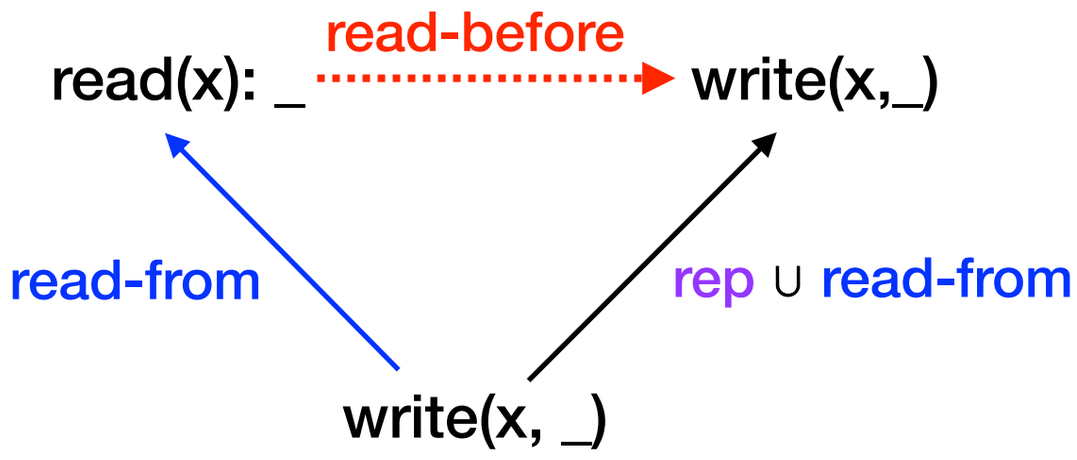
} original CC specification

is equivalent to

∃ **read-from**. **rep** ∪ **read-from** ∪ **read-before** acyclic } red. to assertion checking

read-from: relating each read to a write with the same variable/value

read-before: relating each read to all writes that overwrote the read value:



CC: Key-Value Stores

∃ **vis**. ∀ read. ``**return value** → a maximal visible write"
∧ **vis** ⊇ **rep** ∧ **vis** is transitive

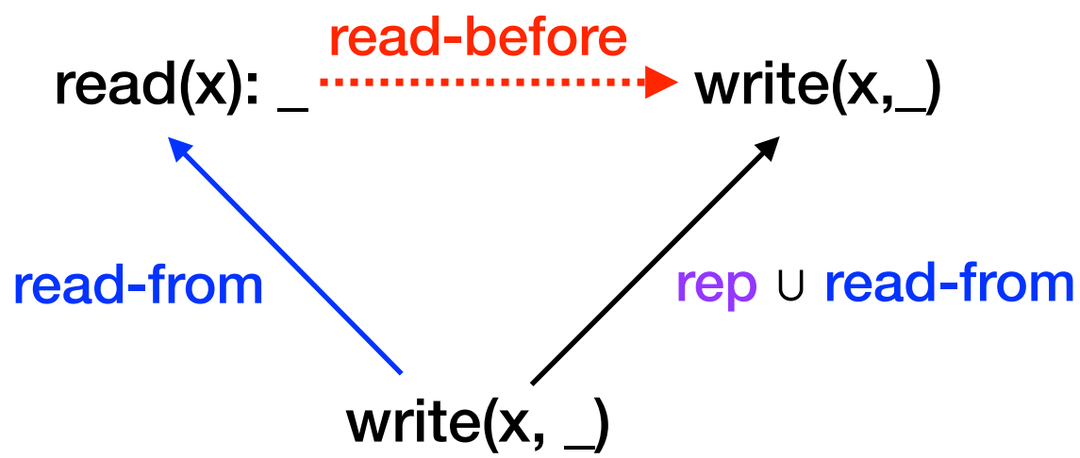
} original CC specification

is equivalent to

∃ **read-from**. **rep** ∪ **read-from** ∪ **read-before** acyclic } red. to assertion checking

read-from: relating each read to a write with the same variable/value

read-before: relating each read to all writes that overwrote the read value:



CC: Key-Value Stores

∃ **vis**. ∀ read. ``**return value** → a maximal visible write”
∧ **vis** ⊇ **rep** ∧ **vis** is transitive

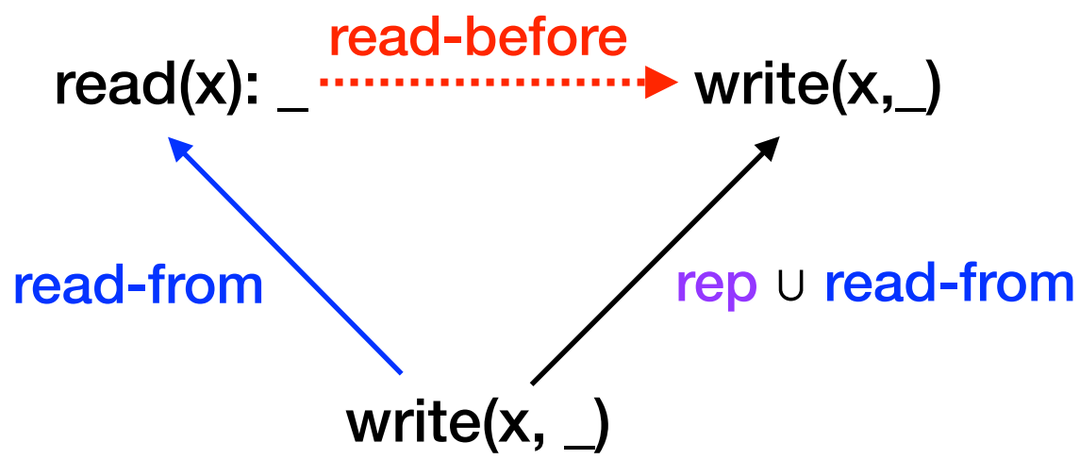
} original CC specification

is equivalent to

∃ **read-from**. **rep** ∪ **read-from** ∪ **read-before** acyclic } red. to assertion checking

read-from: relating each read to a write with the same variable/value

read-before: relating each read to all writes that overwrote the read value:



CC: Key-Value Stores

∃ **read-from**. **rep** ∪ **read-from** ∪ **read-before** is acyclic

read-from: relating each read to a write with the same variable/value

read-before: relating each read to a write that overwrites the read value

Each value is written once (data independence) => **fixed read-from**

CC: Key-Value Stores

∃ **read-from**. **rep** ∪ **read-from** ∪ **read-before** is acyclic

read-from: relating each read to a write with the same variable/value

read-before: relating each read to a write that overwrites the read value

Each value is written once (data independence) => **fixed read-from**

Theorem. (Testing)

Checking CC for key-value store differentiated executions is polynomial

Theorem. (Static verification)

Checking CC for **data independent** finite-state key-value stores is decidable

(finite-state monitors to detect “cyclic” executions)

CC: Key-Value Stores

∃ **read-from**. **rep** ∪ **read-from** ∪ **read-before** is acyclic

read-from: relating each read to a write with the same variable/value

read-before: relating each read to a write that overwrites the read value

Each value is written once (data independence) => **fixed read-from**

Theorem. (Testing)

Checking CC for key-value store differentiated executions is polynomial

Theorem. (Static verification)

Checking CC for **data independent** finite-state key-value stores is decidable

(finite-state monitors to detect “cyclic” executions)

Extended to transactions in DBCOP [OOPSLA’19]

Summary

Using verifiable specifications for

Testing: polynomial-time under reasonable assumptions

- effective tools: Java objects, key-value stores

Verification: techniques that rely on the existing technology for automation

- proof of concept implementations on paradigmatic case studies

Consistency: eventual → causal → linearizability, transactions

Data Types: key-value maps, queues, stacks, sets, CRDTs

Perspectives

CDT specifications: uniform construction of “verifiable” specifications

Root causing:

- understand the reasons behind a violation
- identify important events that invalidate an input-output behavior

Quantitative consistency: tradeoff consistency vs. performance

Programming on top of weak consistency:

- weak consistency (e.g., causal) admits “anomalies”
- understanding the impact of these anomalies on ensuring a certain functionality