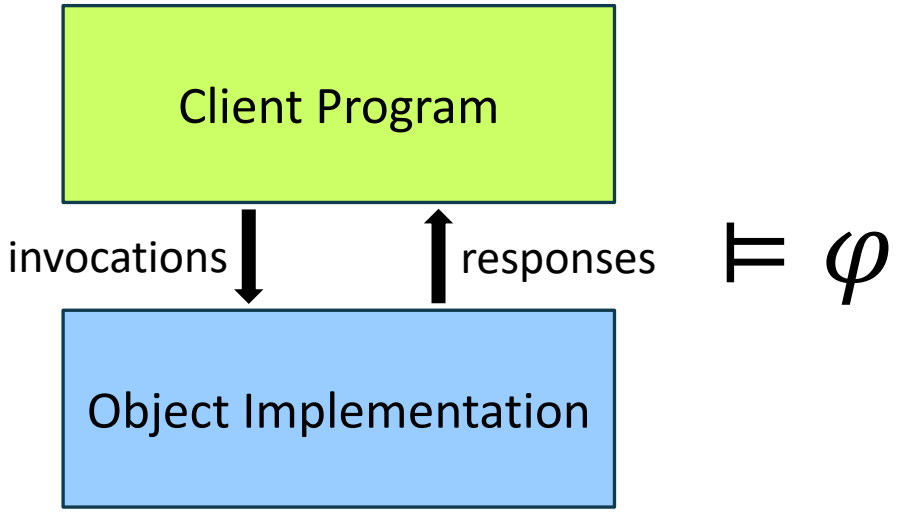
Hyperproperty-Preserving Register Specifications

ORI LAHAV

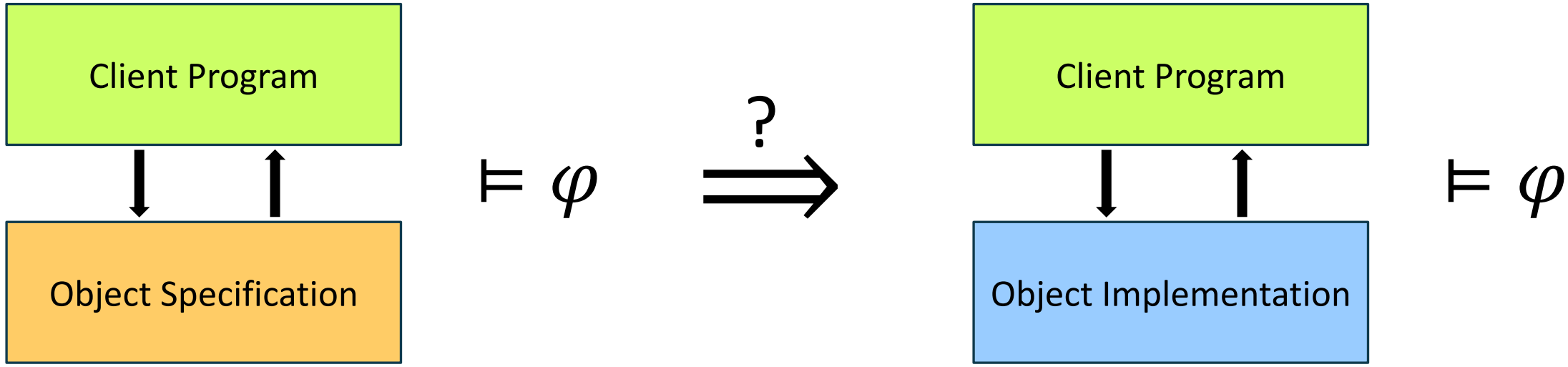
JOINT WORK WITH: YOAV BEN SHIMON AND SHARON SHOHAM

(PRESENTED IN DISC'24)

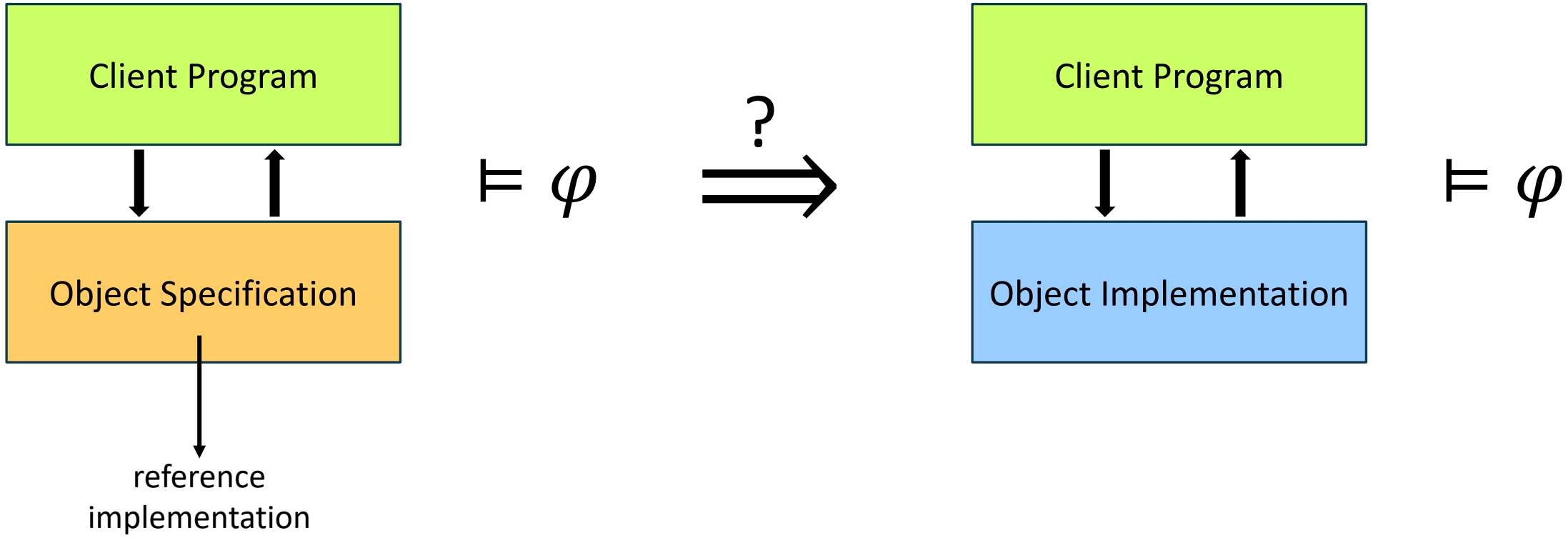
Verification via Abstraction



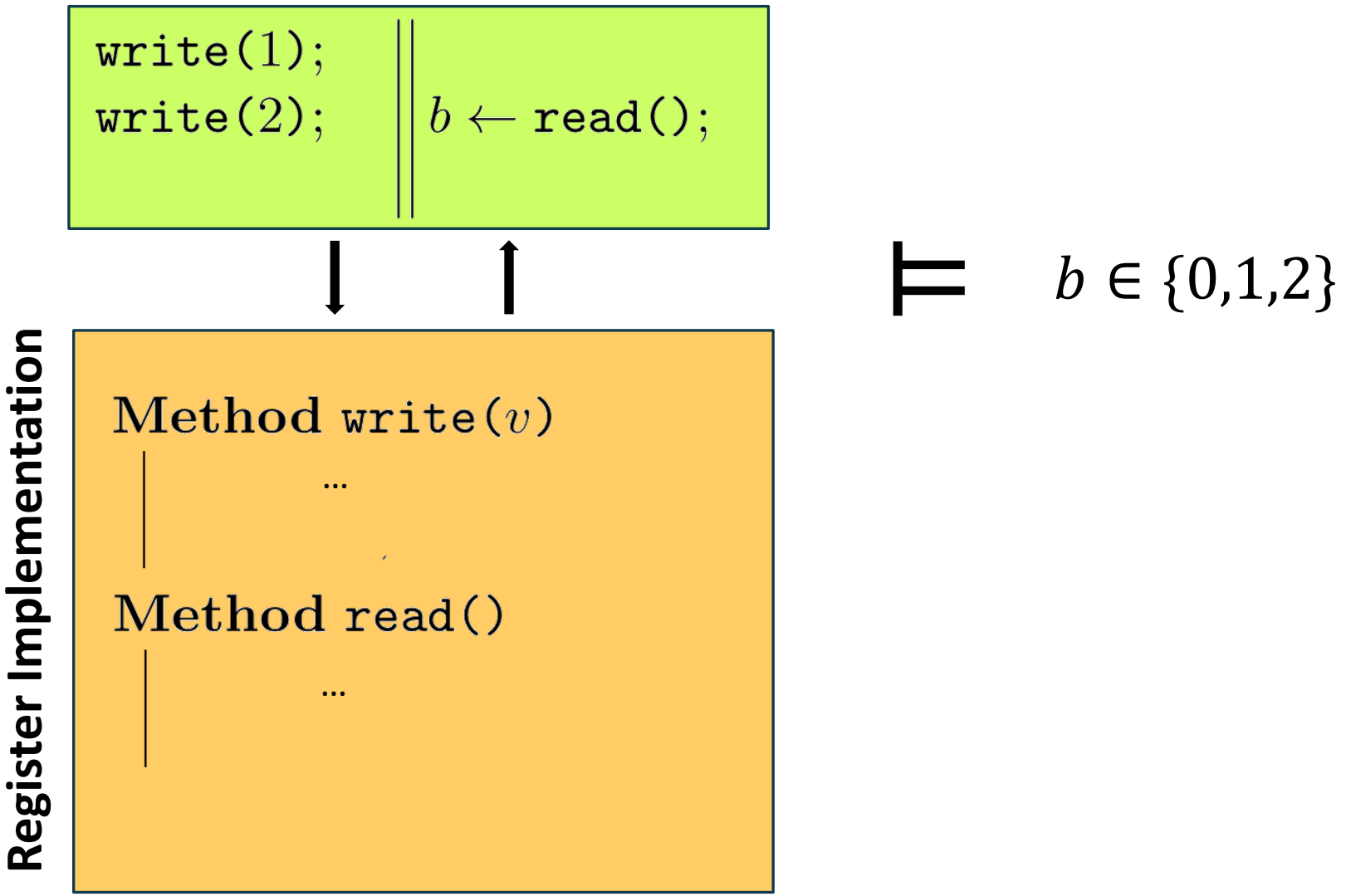
Verification via Abstraction



Verification via Abstraction

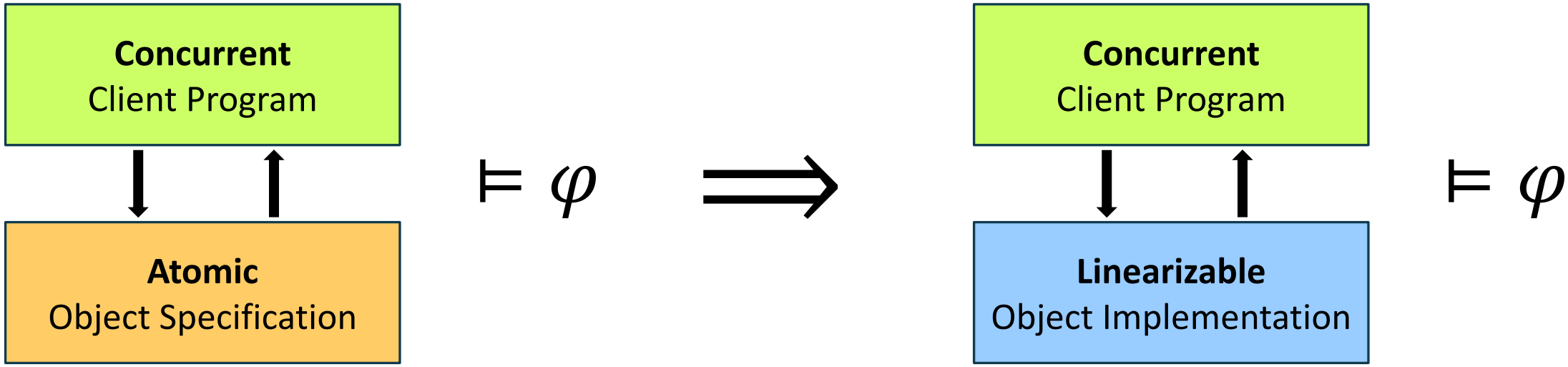


Example: Program using Shared Register

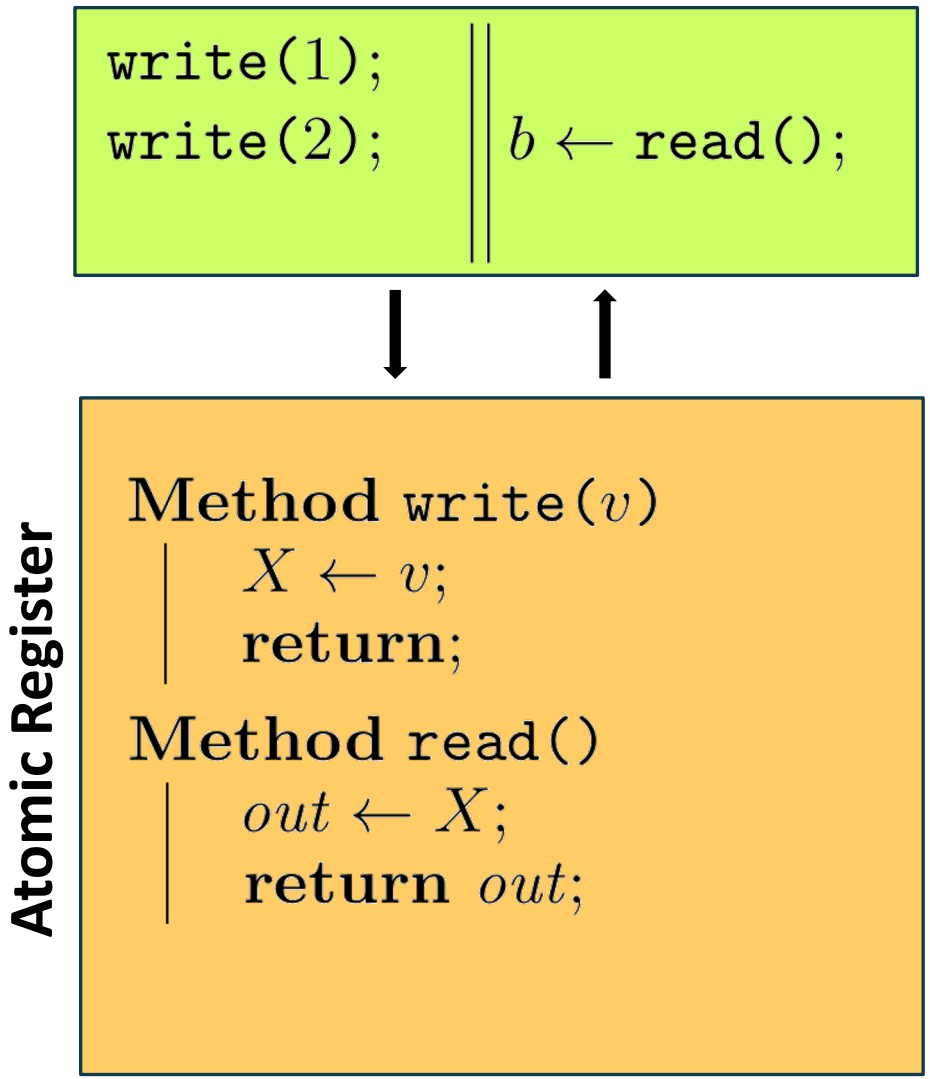


$\models \quad b \in \{0,1,2\}$

Abstraction via Linearizability [Filipović, O'Hearn, Rinetzky, Yang '10]



Example: Program using Shared Register



$\models \quad b \in \{0,1,2\}$

Since it holds for the atomic register, it will hold for any other linearizable implementation.

Linearizability [Herlihy, Wing '90]

- $e \sqsubseteq s$: concurrent execution e *linearized* by sequential history s

- An implementation I is ***linearizable*** (wrt a sequential specification Seq) if there exists a mapping L: executions(I) → Seq such that $\forall e.\, e \sqsubseteq L(e)$

ABD [Attiya, Bar-Noy, Dolev '95]

Method `read()`
  $\langle v, ts\rangle \leftarrow$`query()`;
  `update`(v, ts);
  return v;

Method `write`(v)
  $\langle \_, \langle t, \_\rangle\rangle \leftarrow$`query()`;
  `update`$(v, \langle t+1,$`my_process_id()`$\rangle)$;
  return;

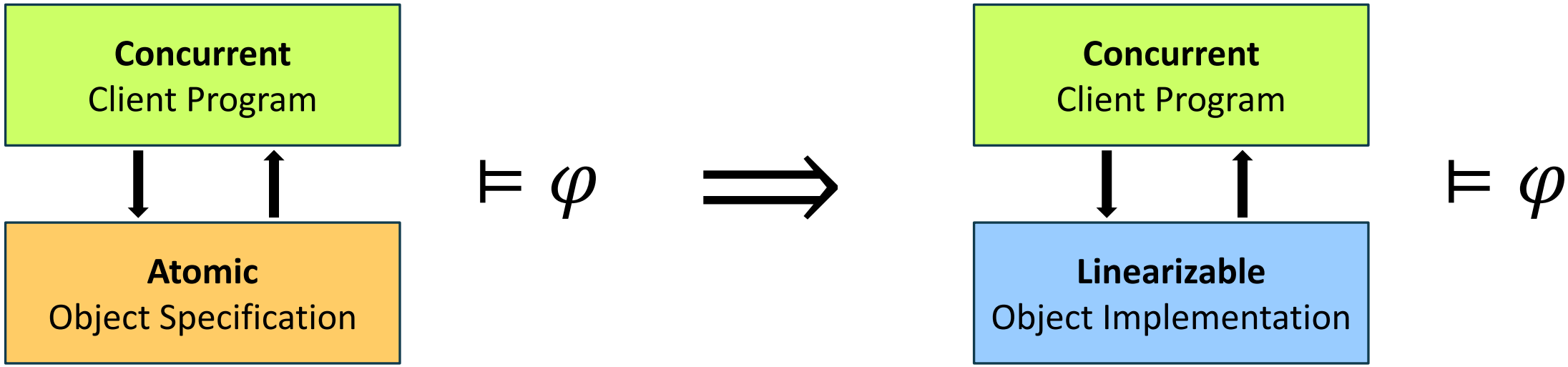
Function `update`(v, ts)
  broadcast $m =$ `update`(v, ts);
  wait until $|Replies(m)| > N/2$;
  return;

Function `query()`
  broadcast $m =$ `query`;
  wait until $|Replies(m)| > N/2$;
  $Q \leftarrow$**pick** $Q \subseteq Replies(m)$ *s.t.* $|Q| > N/2$;
  return $\max Q$;

Background activity by process p:
when $m \in Broadcasts$ **received**
  if $m =$ `query` **then**
    reply $\langle v_p, ts_p\rangle$ **to** m;
  if $m =$ `update`(v, ts) **then**
    $\langle v_p, ts_p\rangle \leftarrow \max\{\langle v, ts\rangle, \langle v_p, ts_p\rangle\}$;
    reply *"ack"* **to** m;

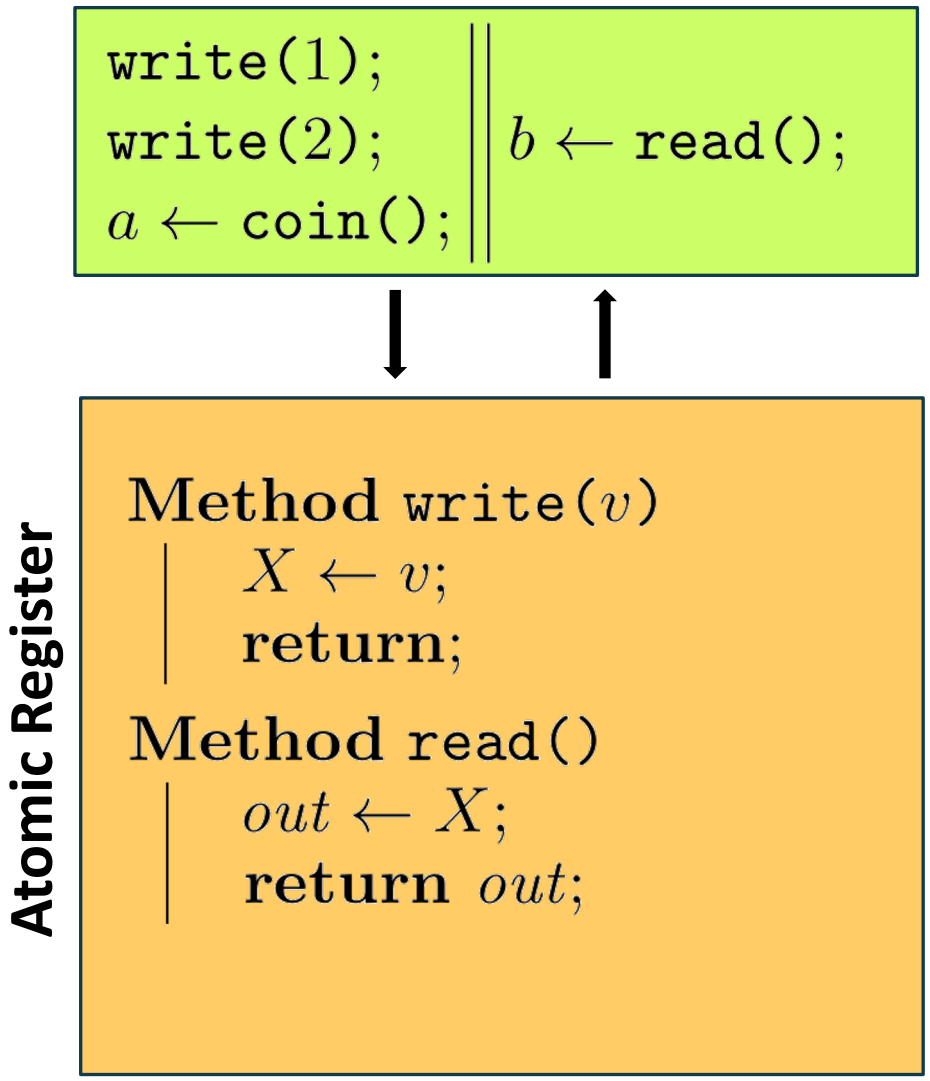
- Message passing implementation of a register
- Tolerates process crashes as long as less than half of the processes crash
- Linearizable

Abstraction via Linearizability [Filipović, O'Hearn, Rinetzky, Yang '10]



- Works when φ is a **trace property** (e.g., bad state not reachable)
- Does not work for **hyperproperties!** [Golab, Higham, Woelfel '11] [Attiya, Enea '19]

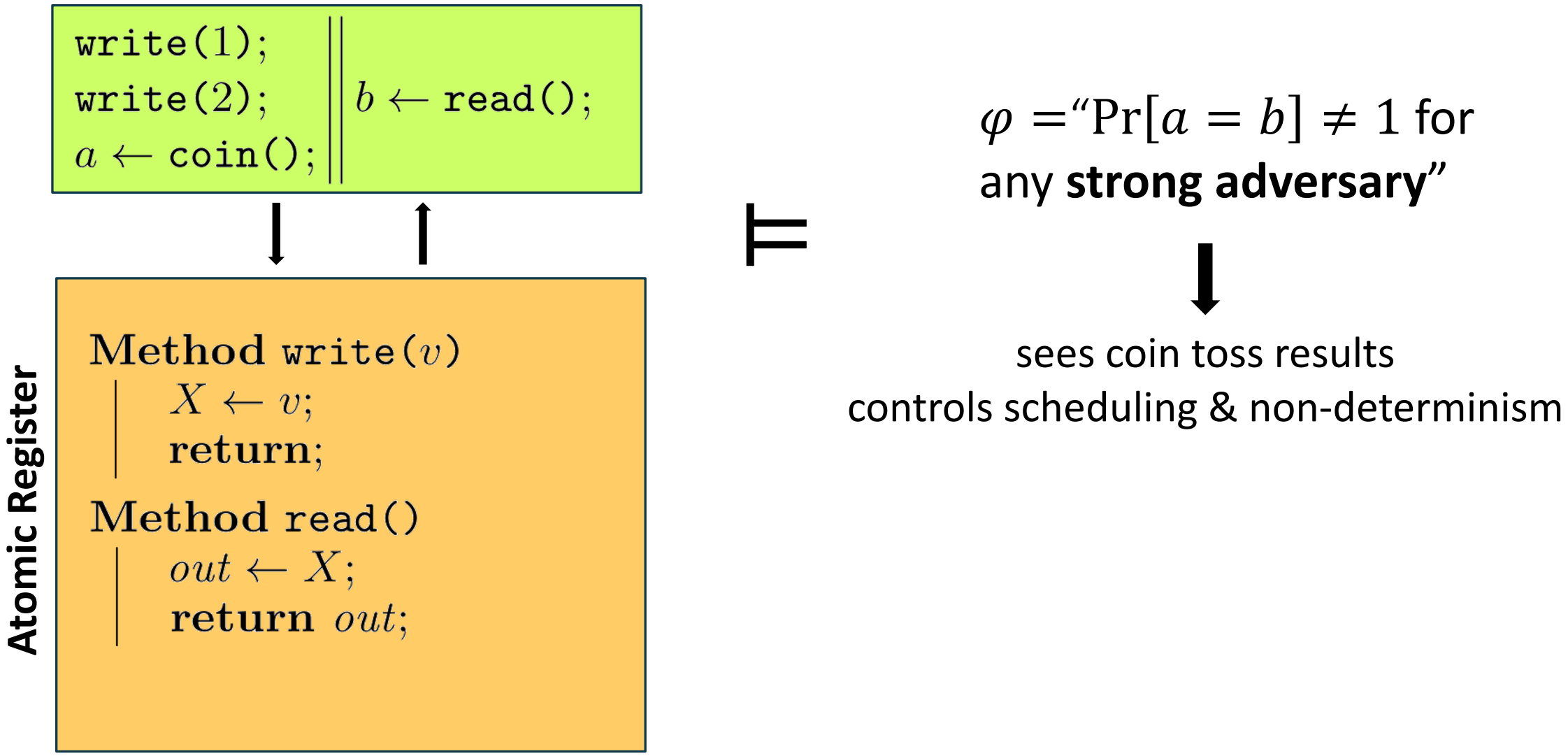
Example: Program using Shared Register



$\varphi =$ "$\Pr[a = b] \neq 1$ for any **strong adversary**"

sees coin toss results
controls scheduling & non-determinism

Example: Program using Shared Register

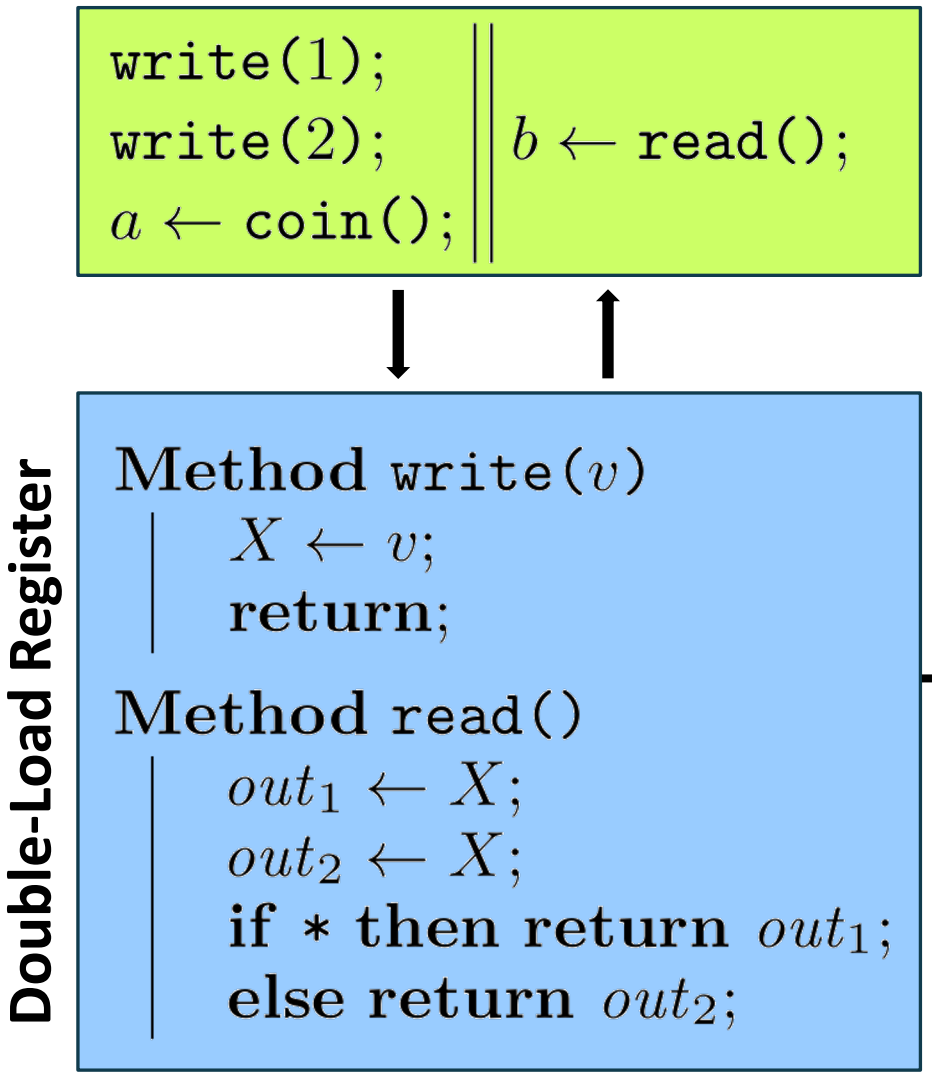


$\models$

$\varphi =$ "$\Pr[a = b] \neq 1$ for any **strong adversary**"

sees coin toss results
controls scheduling & non-determinism

Example: Program using Shared Register



$\varphi =$ "$\Pr[a = b] \neq 1$ for any **strong adversary**"

sees coin toss results
controls scheduling & non-determinism

→ linearizable

Example: Program using Shared Register

```
write(1);
write(2);
a ← coin();
```
`||`
```
b ← read();
```

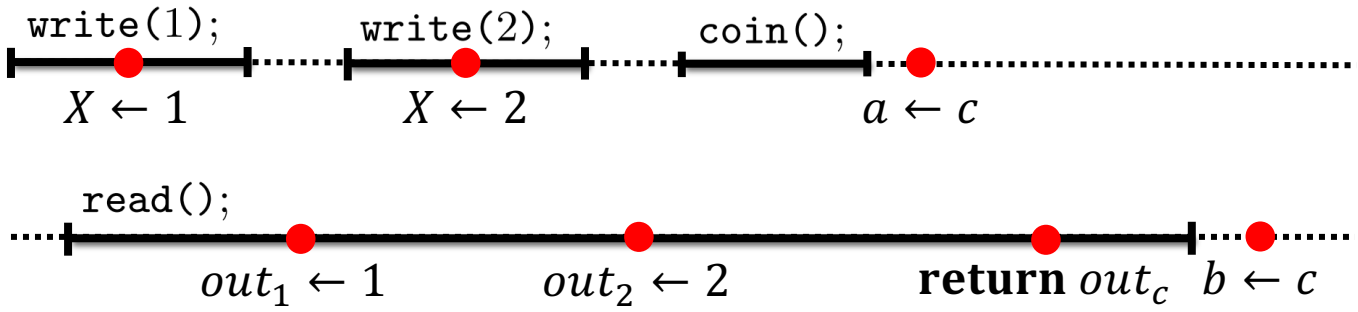
Double-Load Register

```
Method write(v)
    X ← v;
    return;
Method read()
    out1 ← X;
    out2 ← X;
    if * then return out1;
    else return out2;
```

$\not\models$

φ = "$\Pr[a = b] \neq 1$ for any **strong adversary**"

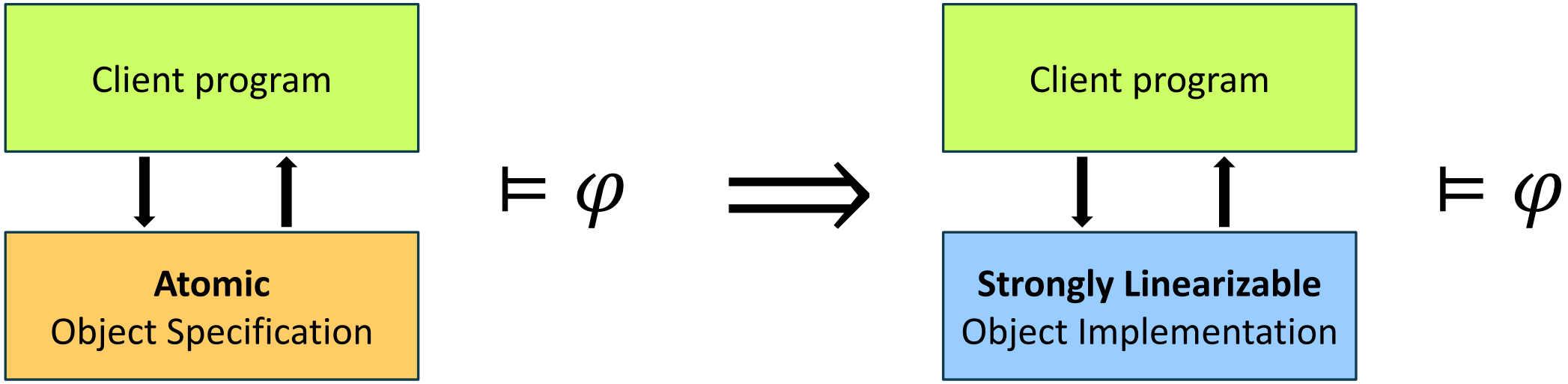
sees coin toss results
controls scheduling & non-determinism



Hyperproperty Preservation via Strong Linearizability

- If φ is a **property of sets** of traces generated by **strong adversaries**:

"hyperproperty"



[Golab, Higham, Woelfel '11]

Strong Linearizability [Golab, Higham, Woelfel '11]

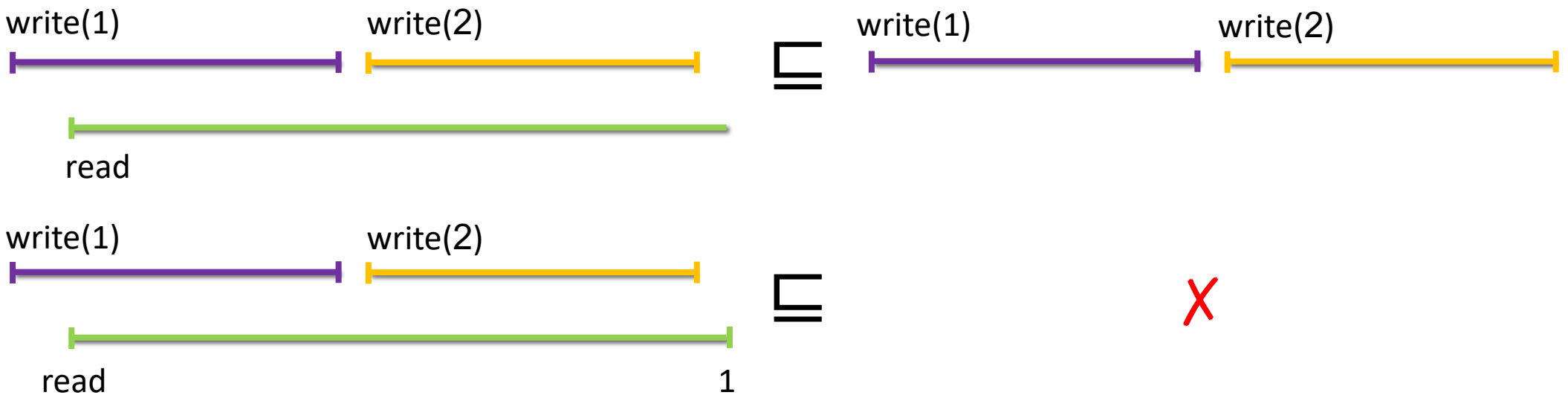
- An implementation I is:
  - **Linearizable** if there exists a mapping L: executions$(I) \rightarrow$ Seq s.t. $\forall e.\, e \sqsubseteq L(e)$
  - **Strongly Linearizable** if $e_1 \leq_{\text{prefix}} e_2 \Longrightarrow L(e_1) \leq_{\text{prefix}} L(e_2)$

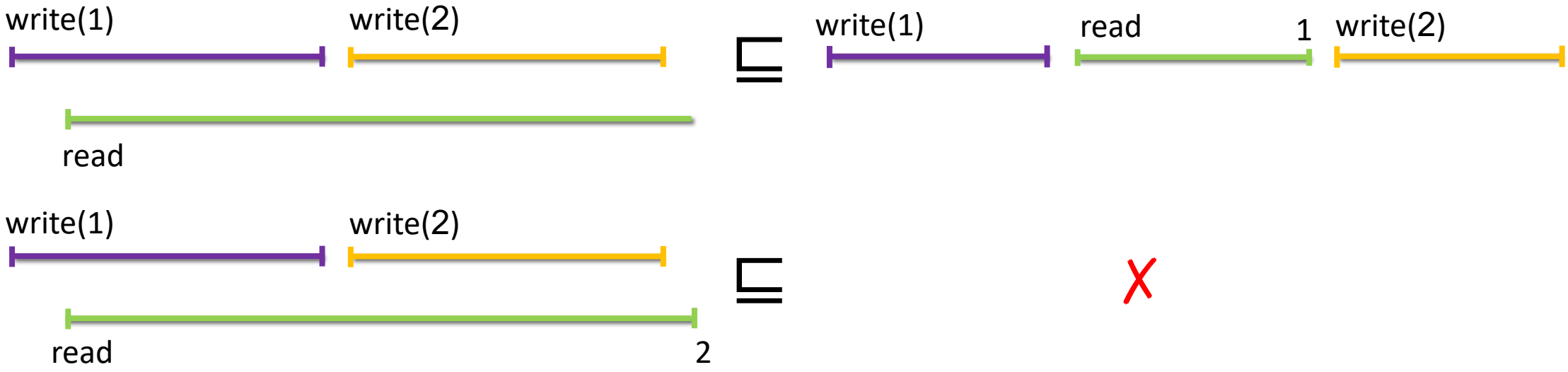
$L($  $) =$ 

$L($  $) =$

	Linearizability	Strong Linearizability
	✓	✓
	✓	✗
	✓	✗

Double-Load Register is not Strongly Linearizable





Strong Linearizability is Rarely Achievable

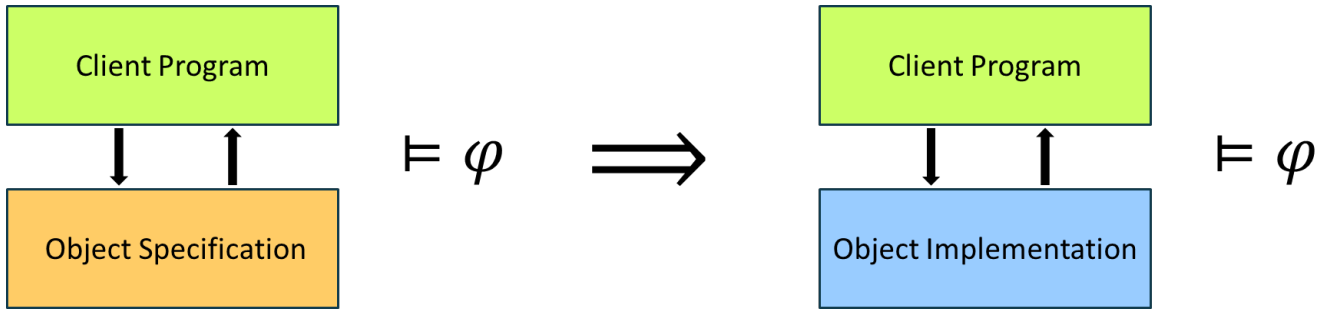
- Various impossibility results for strongly linearizable implementations
- Example: Crash-resilient message passing register implementation
  - No strongly linearizable implementation exists
  - In particular, ABD is not strongly linearizable

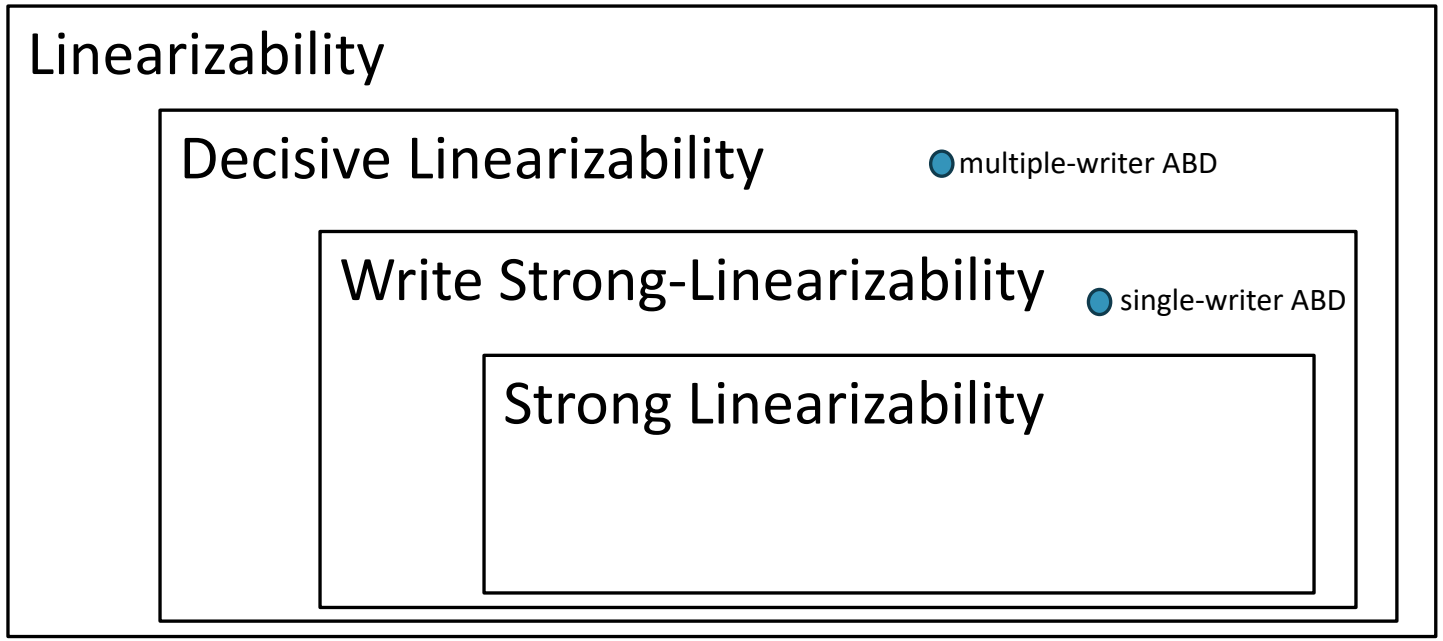
How to reason about hyperproperties of clients that use non-strongly linearizable implementations, such as ABD?

Our Contributions

- Simple shared memory register specifications
  - In the form of (non-atomic) reference-implementations
  - Enable reasoning about hyperproperties of clients that use ***non***-strongly linearizable implementations
  - "Complete" for a range of linearizability classes, including:
    - Write strong-linearizability [Hadzilacos, Hu, Toueg '21]
    - **Decisive** linearizability

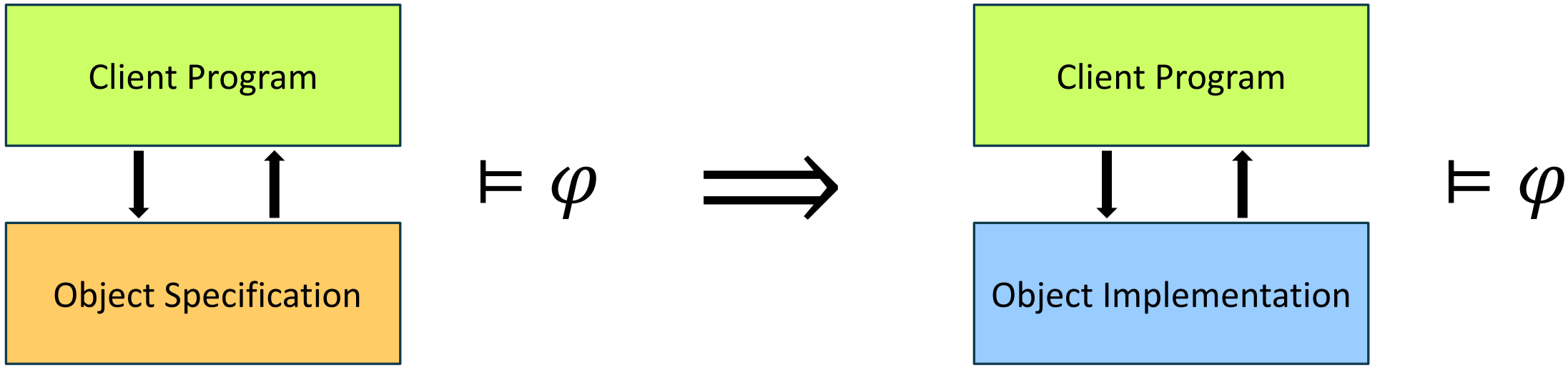
Novel linearizability class





Key idea: Hyperproperty Preservation via Simulation

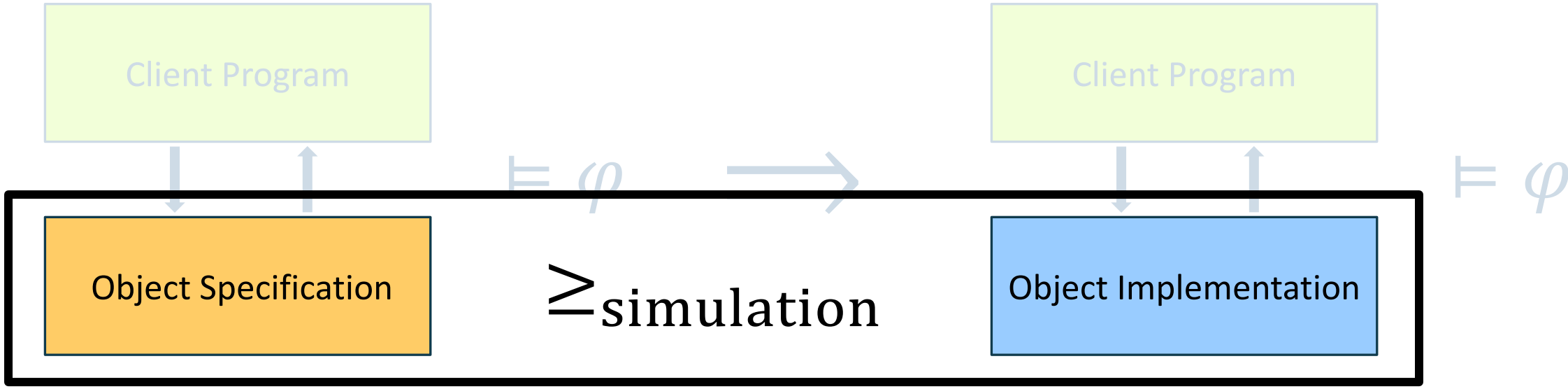
- Preservation of hyperproperties ≡ forward simulation [Attiya, Enea '19]
  (for a strong adversary)



Key idea: Hyperproperty Preservation via Simulation

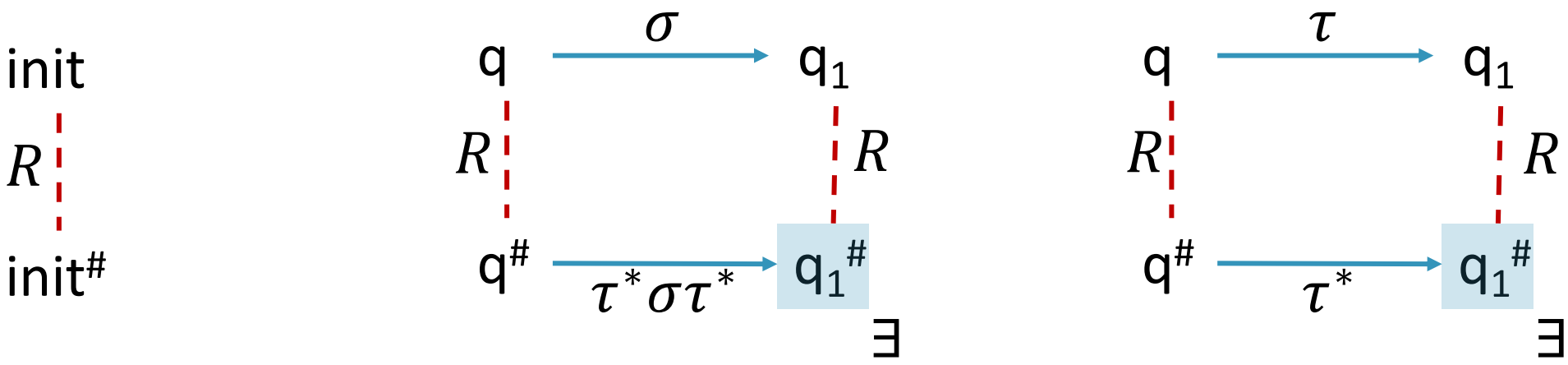
- Preservation of hyperproperties ≡ forward simulation [Attiya, Enea '19]

  (for a strong adversary)



Forward Simulation

- A relation R relating states of the implementation to states of the specification satisfying the following conditions:



Double Load Register vs. Atomic Register

Double-Load Register

Method `write`(v)
  $X \leftarrow v$;
  return;

Method `read`$()$
  $out_1 \leftarrow X$;
  $out_2 \leftarrow X$;
  if $*$ **then return** out_1;
  else return out_2;

$\not\preceq_{\text{simulation}}$

Atomic Register

Method `write`(v)
  $X \leftarrow v$;
  return;

Method `read`$()$
  $out \leftarrow X$;
  return out;

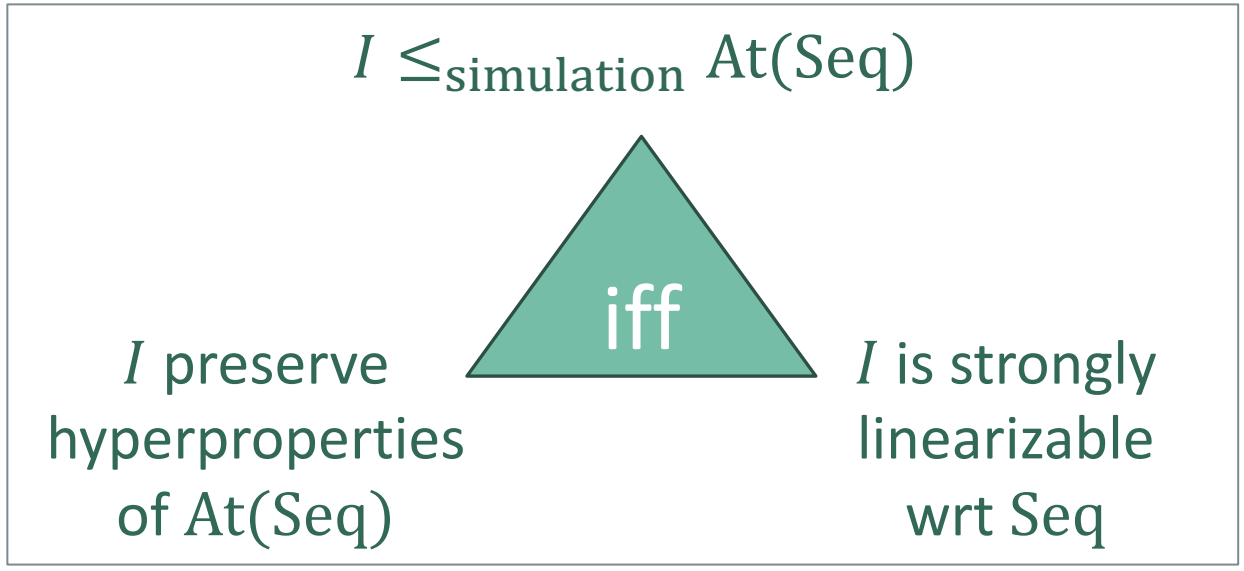
Strong linearizability ≡ forward simulation

(Side note: linearizability ≡ trace inclusion)

An implementation I is strongly linearizable wrt a sequential specification Seq

iff

$I \leq_{\text{simulation}} \text{At(Seq)}$ where At(Seq) is an atomic implementation of Seq



Complete Implementations

- $\mathcal{C}$ - class of implementations
- An implementation I is **$\mathcal{C}$-hard** if $I' \leq_{\text{simulation}} I$ for all $I' \in \mathcal{C}$
- An implementation I is **$\mathcal{C}$- complete** if it is $\mathcal{C}$-hard and $I \in \mathcal{C}$
- Example: Atomic implementation is complete for the class of strongly linearizable implementations

- Problem reformulation: devise *simple* complete implementations for (non-strong) linearizability classes

Complete Implementations

- $\mathcal{C}$ - class of implementations
- An implementation I is **$\mathcal{C}$-hard** if $I' \leq_{\text{simulation}} I$ for all $I' \in \mathcal{C}$
- An implementation I is **$\mathcal{C}$- complete** if it is $\mathcal{C}$-hard and $I \in \mathcal{C}$
- Example: Atomic implementation is complete for the class of strongly linearizable implementations

- Problem reformulation: devise *simple* complete implementations for (non-strong) linearizability classes
- Focus on registers

$\mathcal{C}$ = Write Strong Linearizability [Hadzilacos, Hu, Toueg '21]

- A condition *stronger than linearizability* but *weaker than strong linearizability*.
- Write strongly-linearizable **MWMR** registers are implementable from **SWMR** registers, unlike for strong linearizability.
- Any linearizable implementation of **SWMR** registers is write strongly-linearizable (including single-writer ABD)

$\mathcal{C}$ = Write Strong Linearizability [Hadzilacos, Hu, Toueg '21]

- An implementation I is:
  - **Linearizable** if there exists a mapping L: executions$(I) \to$ Seq s.t. $\forall e.\, e \sqsubseteq L(e)$
  - **Write Strongly Linearizable** if $e_1 \leq_{\text{prefix}} e_2 \implies L(e_1)|_W \leq_{\text{prefix}} L(e_2)|_W$
  - **Strongly Linearizable** if $e_1 \leq_{\text{prefix}} e_2 \implies L(e_1) \leq_{\text{prefix}} L(e_2)$
- Example: the "Double Load Register" is write strongly linearizable
- *How to reason about write strongly linearizable implementations?*

"Some randomized algorithms that fail to terminate with linearizable registers, work with write strongly-linearizable ones…"

A complete implementation for this class

Write Strong Register

Method `write`(v)

$X \leftarrow v$;

return;

Method `read`$()$

$\mathcal{V} \leftarrow \{X\}$;

do

$\langle \mathcal{V}_{\text{prev}}, \mathcal{V} \rangle \leftarrow \langle \mathcal{V}, \mathcal{V} \cup \{X\} \rangle$;

while $\mathcal{V} \neq \mathcal{V}_{\text{prev}}$;

$out \leftarrow$ **pick** $v \in \mathcal{V}$;

return out;

A complete implementation for this class

Write Strong Register

Method `write`(v)

$X \leftarrow v$;

return;

Method `read`$()$

$\mathcal{V} \leftarrow \{X\}$;

do

$\langle \mathcal{V}_{\text{prev}}, \mathcal{V} \rangle \leftarrow \langle \mathcal{V}, \mathcal{V} \cup \{X\} \rangle$;

while $\mathcal{V} \neq \mathcal{V}_{\text{prev}}$;

$out \leftarrow$ **pick** $v \in \mathcal{V}$;

return out;

- Captures hyperproperties of single-writer ABD

A complete implementation for this class

Write Strong Register

Method `write`(v)

$X \leftarrow v$;

return;

Method `read`$()$

$\mathcal{V} \leftarrow \{X\}$;

do

$\langle \mathcal{V}_{\text{prev}}, \mathcal{V} \rangle \leftarrow \langle \mathcal{V}, \mathcal{V} \cup \{X\} \rangle$;

while $\mathcal{V} \neq \mathcal{V}_{\text{prev}}$;

$out \leftarrow$ **pick** $v \in \mathcal{V}$;

return out;

- Captures hyperproperties of single-writer ABD
- *What about multi-writer ABD?*

Example: Multiple Writers

```
write(1);          ||  write(2);
a ← coin();        ||
barrier();         ||  barrier();
                   ||  b ← read();
```

- When using multi-writer ABD, a strong adversary can force $a = b$
- Shared memory implementation capturing this behavior: **Try-not-to-store** register

Example: Multiple Writers

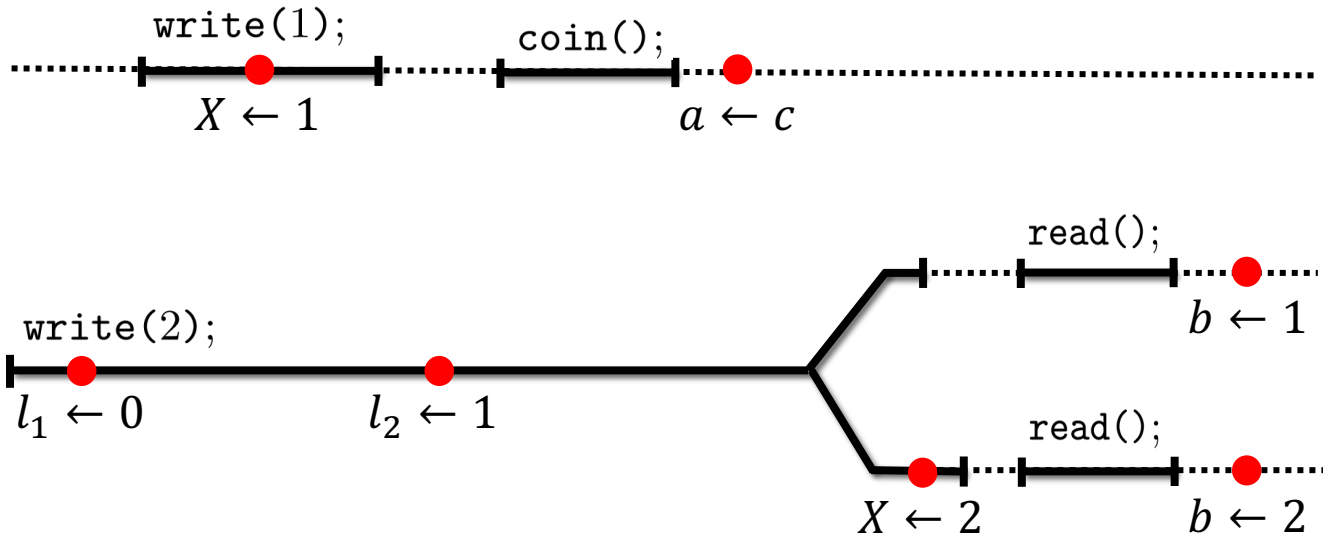
```
write(1);          ||  write(2);
a ← coin();        ||
barrier();         ||  barrier();
                   ||  b ← read();
```

- When using multi-writer ABD, a strong adversary can force $a = b$
- Shared memory implementation capturing this behavior: **Try-not-to-store** register

Try-not-to-store Register

```
Method write(v)
    l1 ← X;
    l2 ← X;
    if * then
        if l1 ≠ l2 then
            return;
    X ← v;
    return;

Method read()
    out ← X;
    return out;
```



Reference Implementation for Multi-Writer ABD

- Combines ideas from the "Write-strong" and "Try-not-to-store" registers
  - Comparing version numbers instead of register values
  - Communicating "overwritten" writes to concurrent reads

Decisive Register

Method `write`(v)

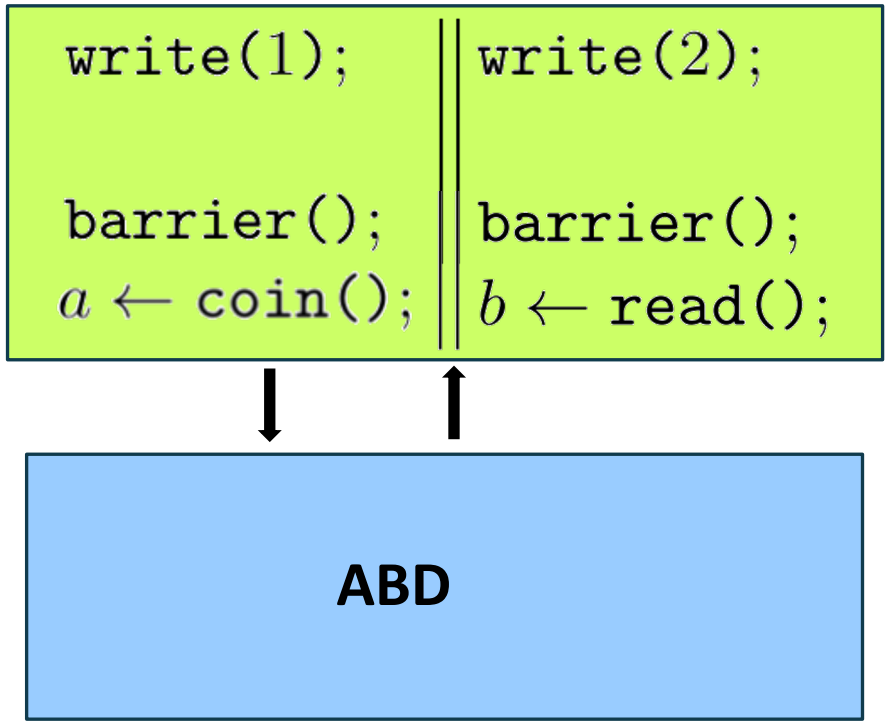
```
await L = 0 do s ← Ver;
if * then
    await L = 0 do ⟨X, Ver⟩ ← ⟨v, Ver + 1⟩;
else
    await L = 0 ∧ Ver > s do
        ⟨L, tmp, X, Ver⟩ ← ⟨1, X, v, Ver − 1⟩;
    ⟨L, X, Ver⟩ ← ⟨0, tmp, Ver + 1⟩;
return;
```

Method `read()`

```
await L = 0 do ⟨s, V⟩ ← ⟨Ver, {X}⟩;
do
    atomic
        V_prev ← V;
        if Ver ≥ s then V ← V ∪ {X};
while V ≠ V_prev;
out ← pick v ∈ V;
return out;
```

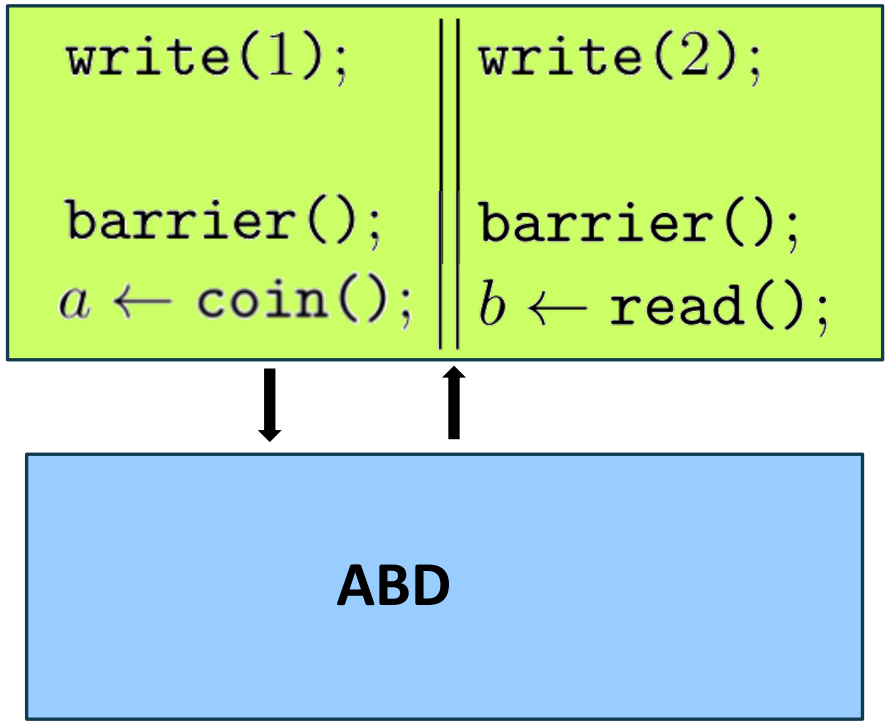
- Captures hyperproperties of multi-writer ABD

Simple Application



We can use our “Decisive Register” to conclude that a strong adversary **cannot** force $a = b$

Simple Application



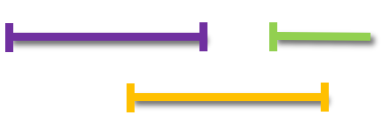
We can use our “Decisive Register” to conclude that a strong adversary **cannot** force $a = b$

- *Applications beyond ABD?*

Decisive linearizability: a new class for which the decisive register is complete

Decisive Linearizability

- An implementation I is:
  - **Linearizable** if there exists a mapping L: executions$(I) \rightarrow$ Seq s.t. $\forall e.\, e \sqsubseteq L(e)$
  - **Decisively Linearizable** if $e_1 \leq_{\text{prefix}} e_2 \implies L(e_1) \leq_{\text{subsequence}} L(e_2)$
  - **Strongly Linearizable** if $e_1 \leq_{\text{prefix}} e_2 \implies L(e_1) \leq_{\text{prefix}} L(e_2)$

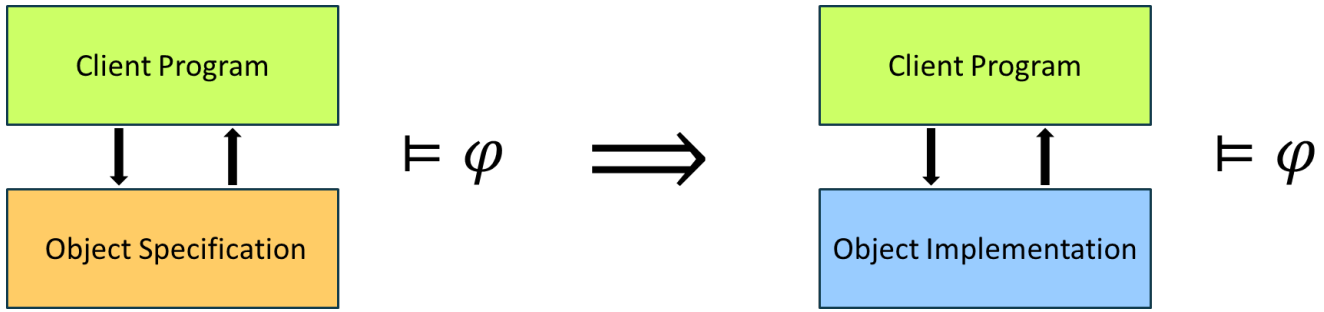
$L($  $) =$ 

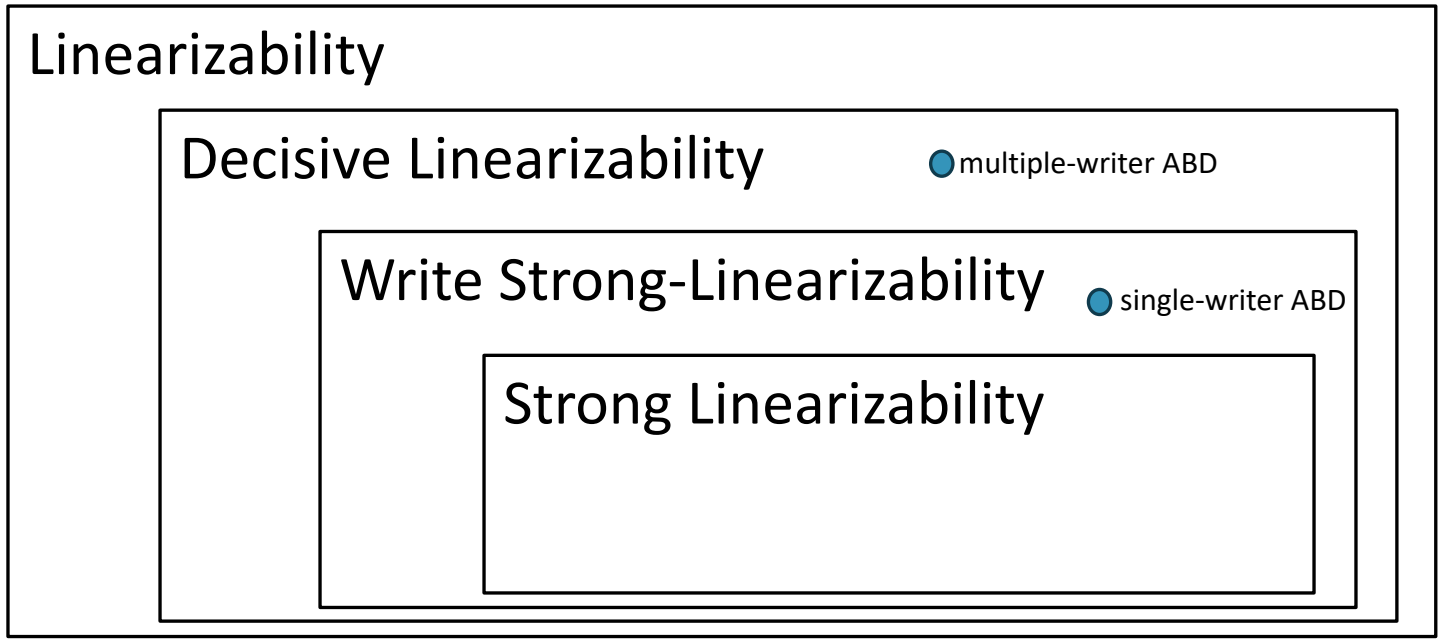
$L($  $) =$

	Lin.	Decisive	Strong
	✓	✓	✓
	✓	✓	✗
	✓	✗	✗

Conclusion

- Simple shared memory register specifications:
  - Write strong register
  - Decisive register
  - General construction (in the paper)
- Enable reasoning about hyperproperties of clients that use implementations satisfying different linearizability criteria
- New linearizability class: ***decisive linearizability***
  - Applicable beyond registers





Ongoing and Future Work

- Compositionality of decisive linearizability
- Using complete objects for automatic linearizability verification
- Going beyond registers (consensus protocols?)
- Weaker adversaries?
- Preserving *certain* hyperproperties

Ongoing and Future Work

- Compositionality of decisive linearizability
- Using complete objects for automatic linearizability verification
- Going beyond registers (consensus protocols?)
- Weaker adversaries?
- Preserving *certain* hyperproperties

Thank You!