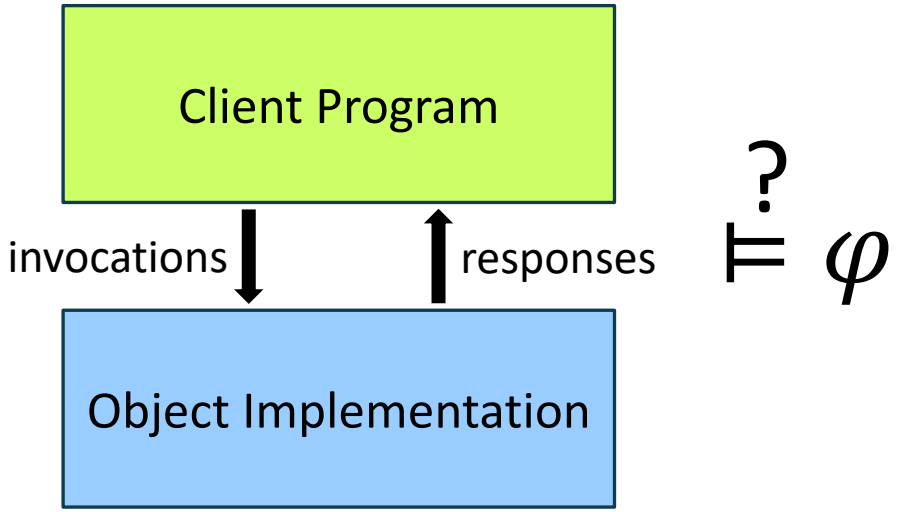
Hyperproperty-Preserving Register Specifications

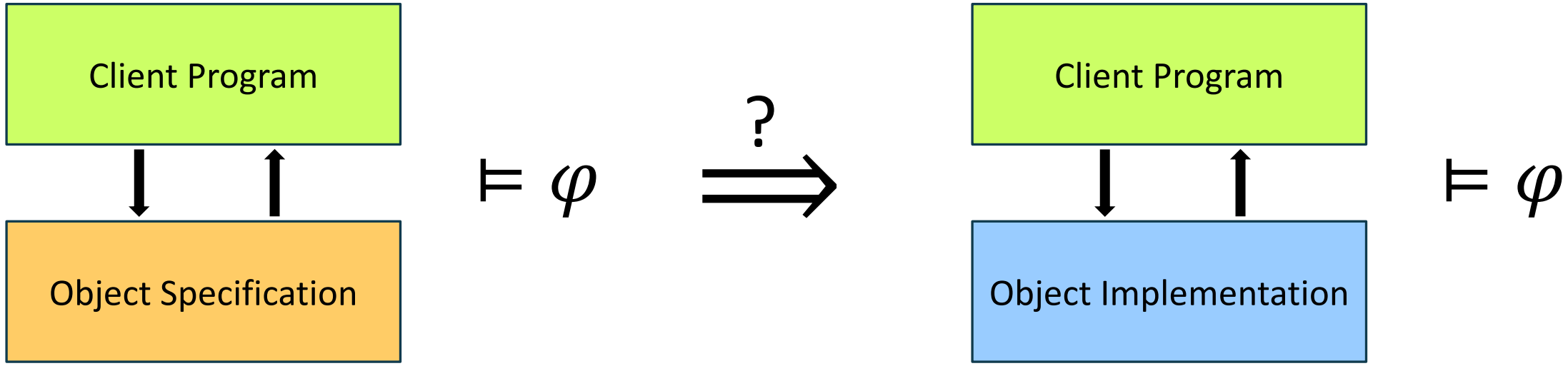
YOAV BEN SHIMON, ORI LAHAV, SHARON SHOHAM

TEL AVIV UNIVERSITY

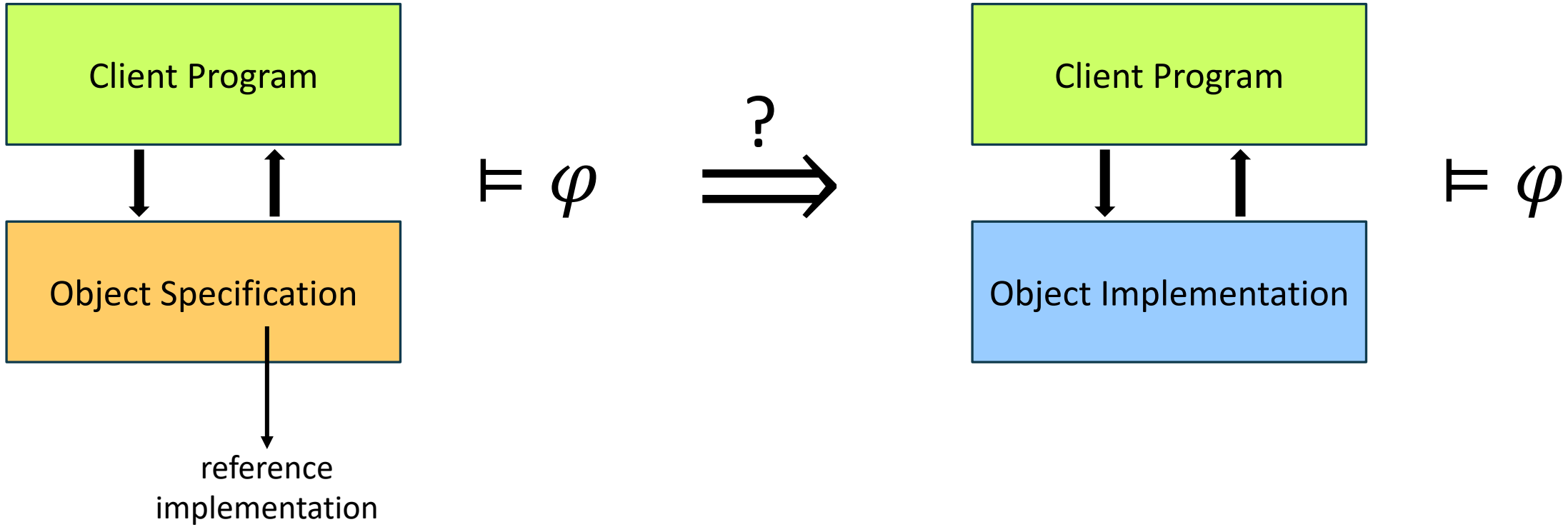
Verification via Abstraction



Verification via Abstraction

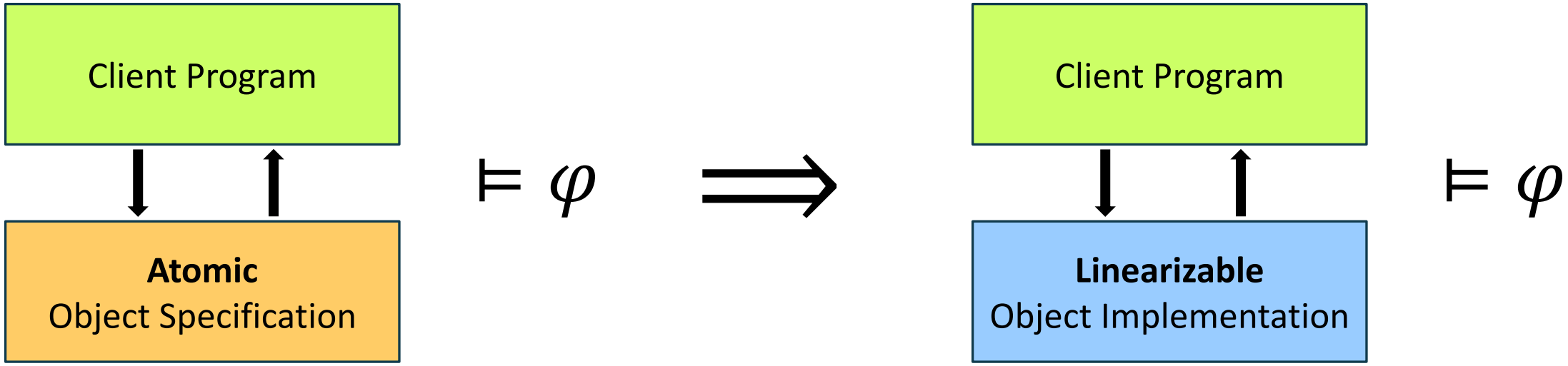


Verification via Abstraction



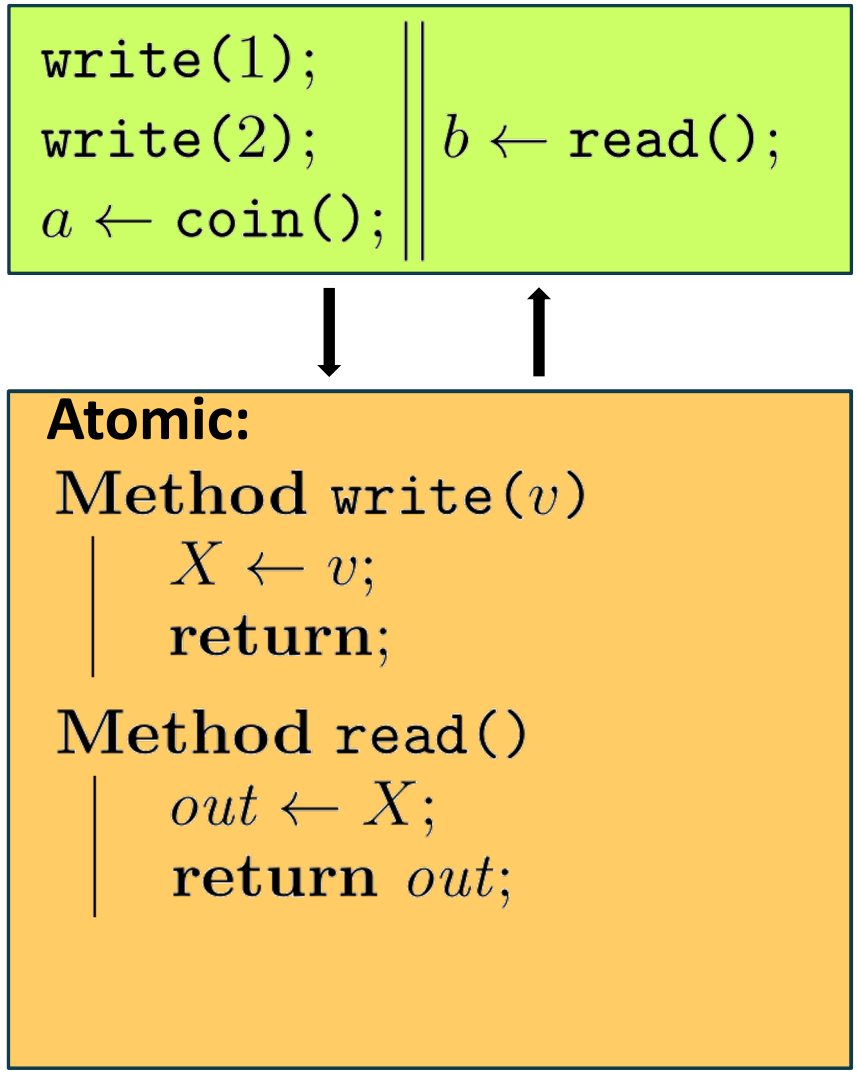
Abstraction via Linearizability

- If 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

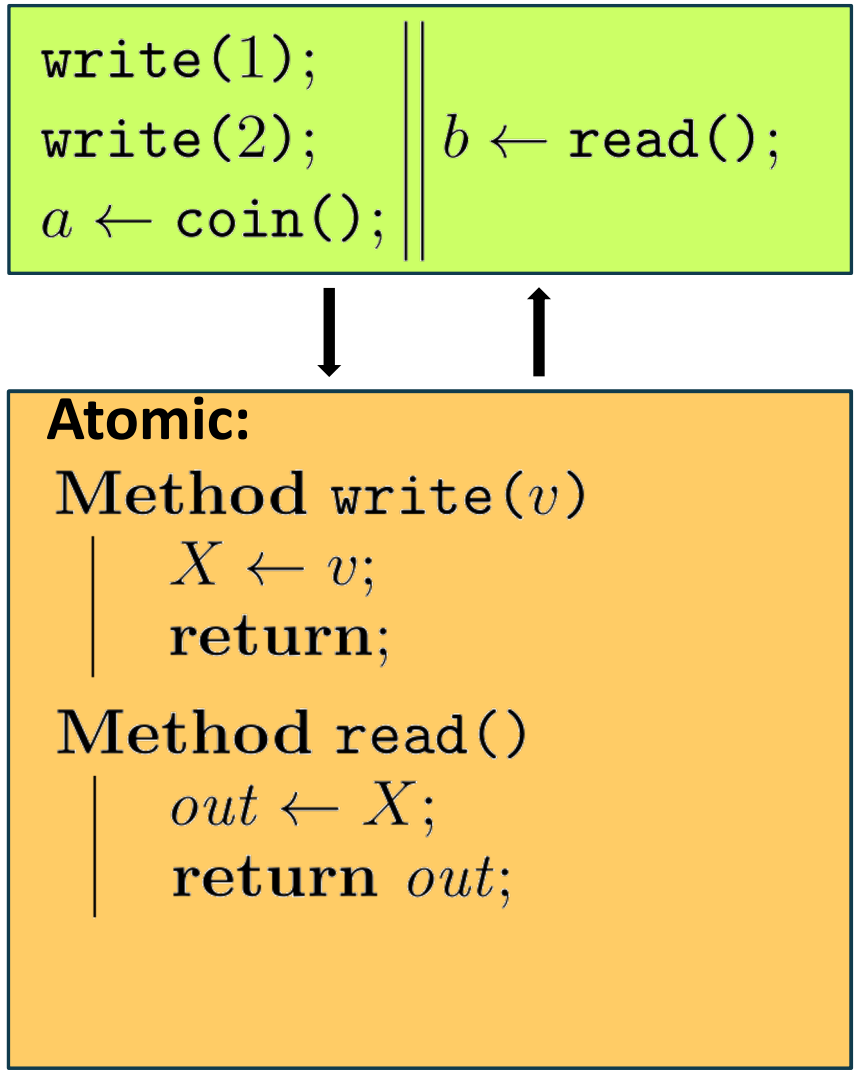


$\varphi =$ "$\Pr[a = b] = \frac{1}{2}$ for any **strong adversary**"

sees coin toss results

controls scheduling & non-determinism

Example: Program using Shared Register



$\models$

$\varphi =$ "$\Pr[a = b] = \frac{1}{2}$ for any **strong adversary**"

sees coin toss results

controls scheduling & non-determinism

Example: Program using Shared Register

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

Double-load:

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

→ Linearizable

$\varphi =$ "$\Pr[a = b] = \frac{1}{2}$ for any **strong adversary**"

sees coin toss results

controls scheduling & non-determinism

Example: Program using Shared Register

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

Double-load:

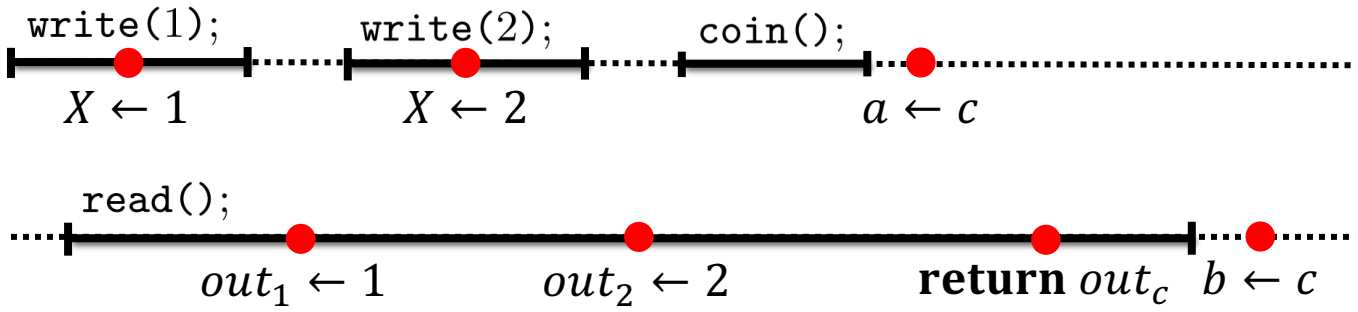
```
Method write(v)
    X ← v;
    return;
Method read()
    out1 ← X;
    out2 ← X;
    if * then return out1;
    else return out2;
```

$\not\models$

$\varphi =$ "$\Pr[a = b] = \frac{1}{2}$ for any **strong adversary**"

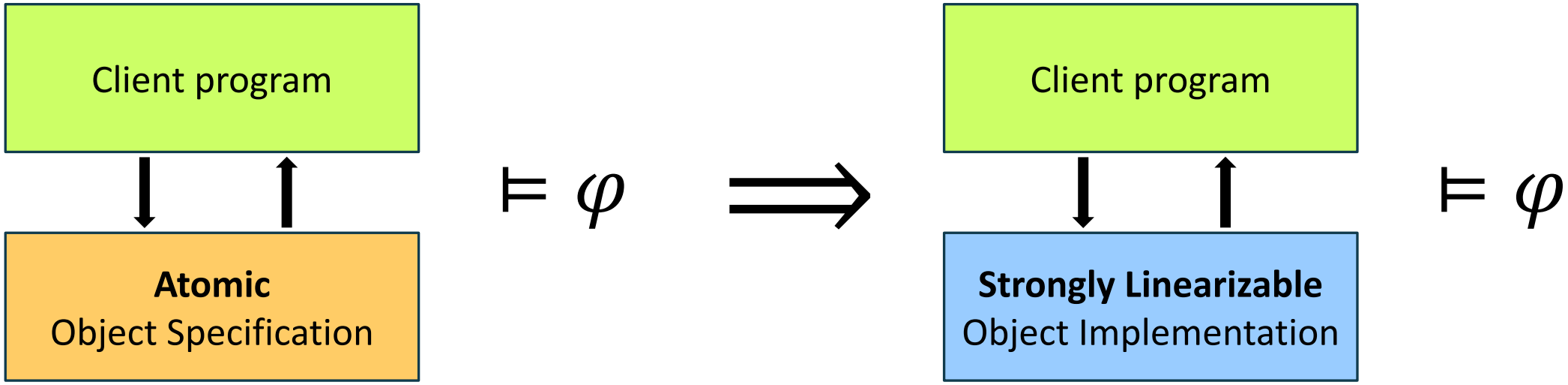
sees coin toss results

controls scheduling & non-determinism



Hyperproperty Preservation via Strong Linearizability

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



[Golab, Higham, Woelfel '11]

- Indeed, the *double-load register implementation* is not strongly linearizable

Strong Linearizability is Rarely Achievable

- Various impossibility results for strongly linearizable implementations
- Example: Crash-resilient lock-free message passing register implementation
  - No strongly linearizable implementation exists
  - In particular, ABD [Attiya, Bar-Noy, Dolev '95] is not strongly linearizable

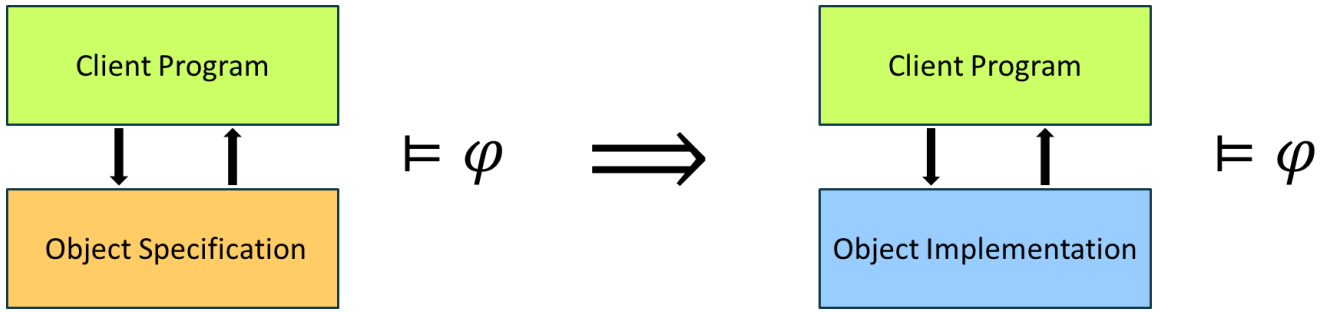
Strong Linearizability is Rarely Achievable

- Various impossibility results for strongly linearizable implementations
- Example: Crash-resilient lock-free message passing register implementation
  - No strongly linearizable implementation exists
  - In particular, ABD [Attiya, Bar-Noy, Dolev '95] is not strongly linearizable

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

Our Contributions

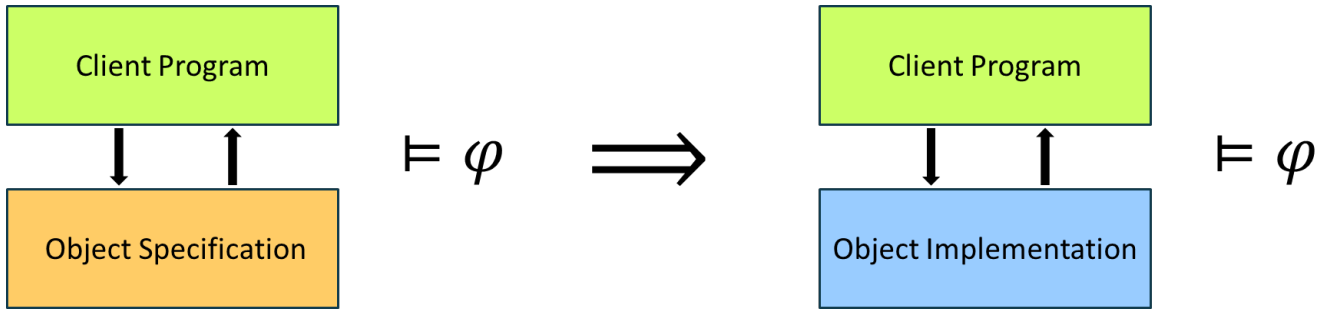
- Simple shared memory register specifications
  - In the form of (non-atomic) reference-implementations
  - Enable reasoning about hyperproperties of clients that use non-strongly linearizable implementations

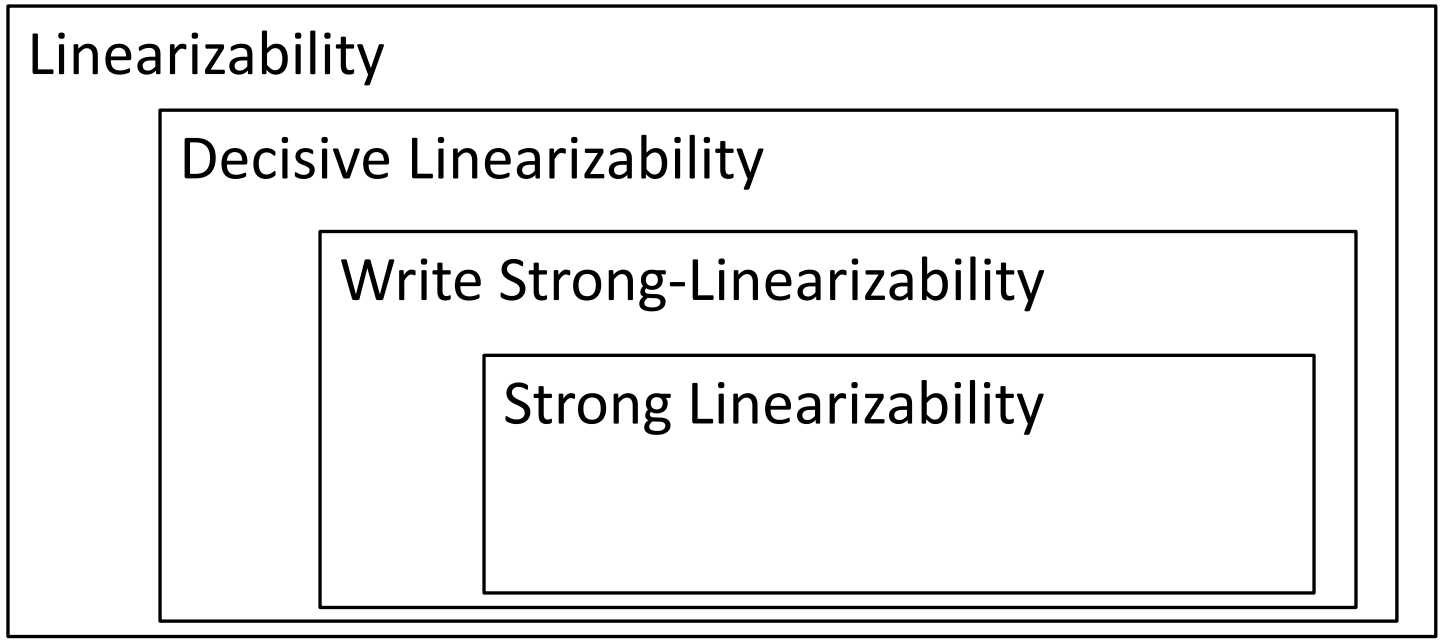


Our Contributions

- 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

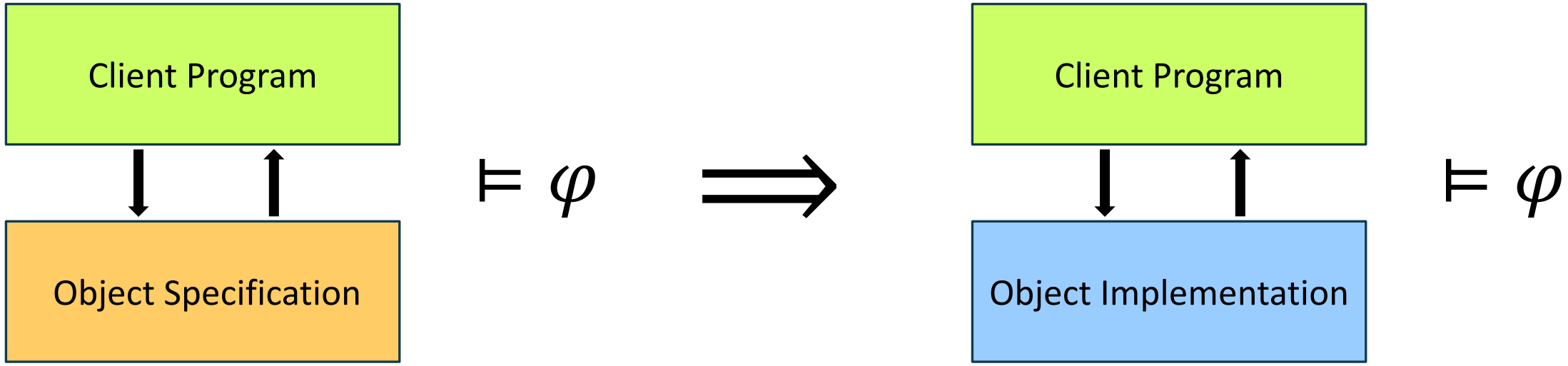
Novel linearizability class





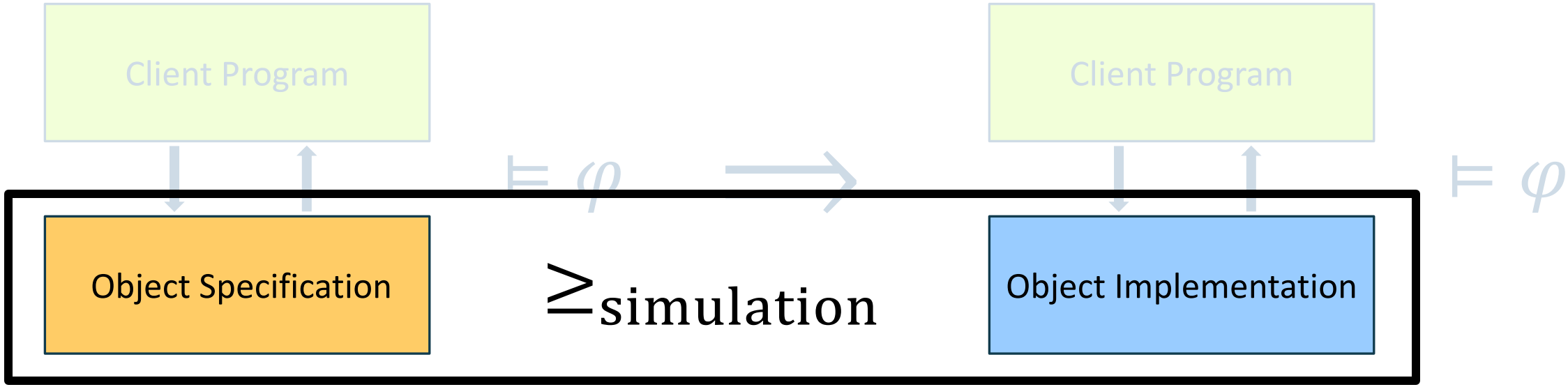
Hyperproperty Preservation via Simulation

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



Hyperproperty Preservation via Simulation

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



Complete Implementations

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

Complete Implementations

- $\mathcal{C}$ - class of implementations
- An implementation I is
  - **$\mathcal{C}$-hard** if $I' \leq_{\text{simulation}} I$ for all $I' \in \mathcal{C}$
  - **$\mathcal{C}$-complete** if additionally, $I \in \mathcal{C}$
- 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 $I' \in \mathcal{C}$
  - **$\mathcal{C}$-complete** if additionally, $I \in \mathcal{C}$
- 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 $I' \in \mathcal{C}$
  - **$\mathcal{C}$-complete** if additionally, $I \in \mathcal{C}$
- 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]

- Includes all single-writer register implementations
  - Specifically, single-writer ABD
- The "Write Strong Register" is complete:

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;

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

- Includes all single-writer register implementations
  - Specifically, single-writer ABD
- The "Write Strong Register" is complete:

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, can force $a = b$ (Proof in the paper)

Example: Multiple Writers

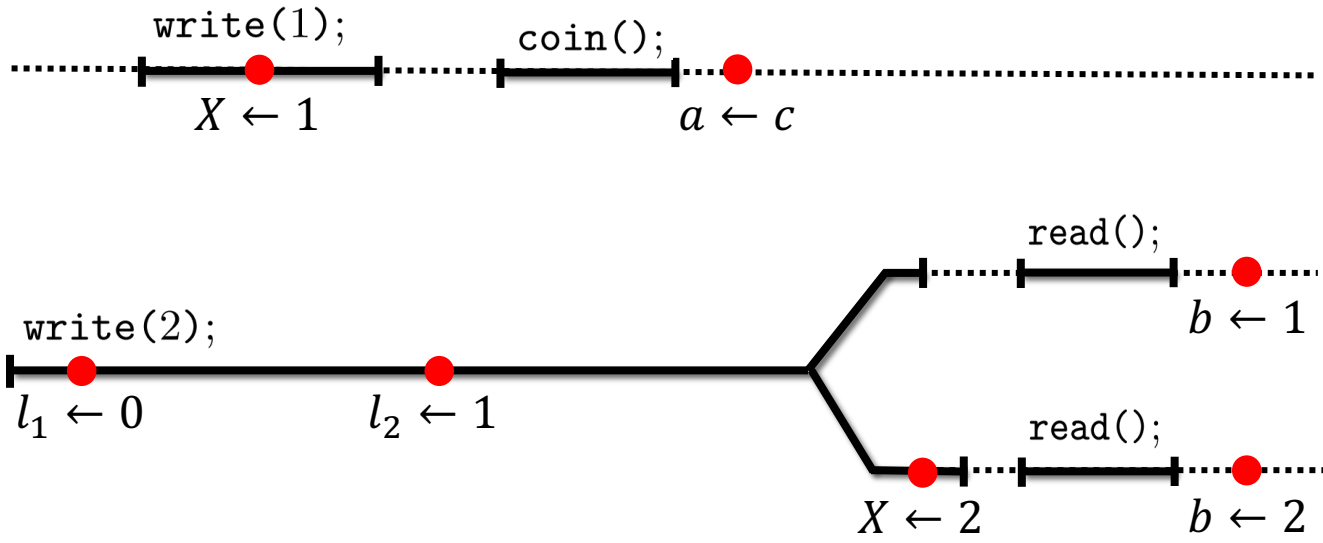
```
write(1);          ||  write(2);
a ← coin();        ||
barrier();         ||  barrier();
                   ||  b ← read();
```

- When using multi-writer ABD, can force $a = b$ (Proof in the paper)
- Shared memory implementation capturing this behavior: **Try-not-to-store** register

Try-not-to-store:

```
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
  - Details in the paper
- Captures hyperproperties of multi-writer ABD

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
  - Details in the paper
- Captures hyperproperties of multi-writer ABD

- Next: decisive linearizability, a new class of linearizable implementations for which this implementation is complete

Decisive Linearizability

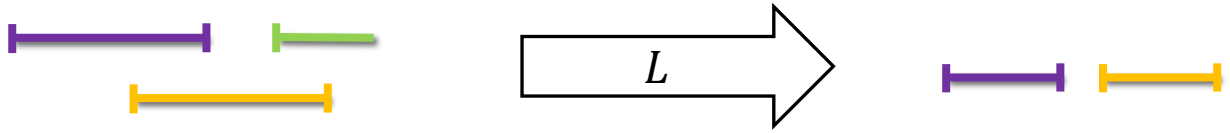
- $e \sqsubseteq s$: execution e linearized by sequential history s

Decisive Linearizability

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

Decisive Linearizability

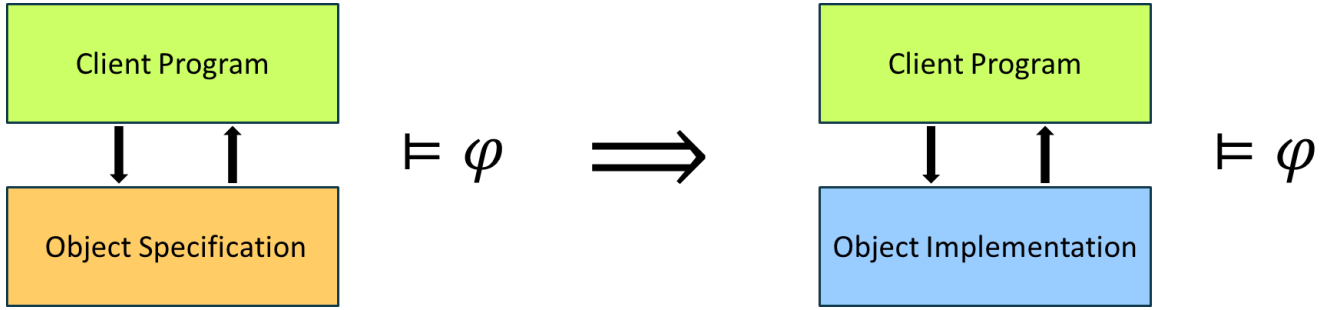
- $e \sqsubseteq s$: execution e linearized by sequential history s
- An implementation I is:
  - **Linearizable** if there exists a mapping L: executions$(I) \rightarrow$ Seq s.t. $\forall e.\, e \sqsubseteq L(e)$
  - **Decisively Linearizable** if $e_1 \leq_{\text{prefix}} e_2 \Longrightarrow L(e_1)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

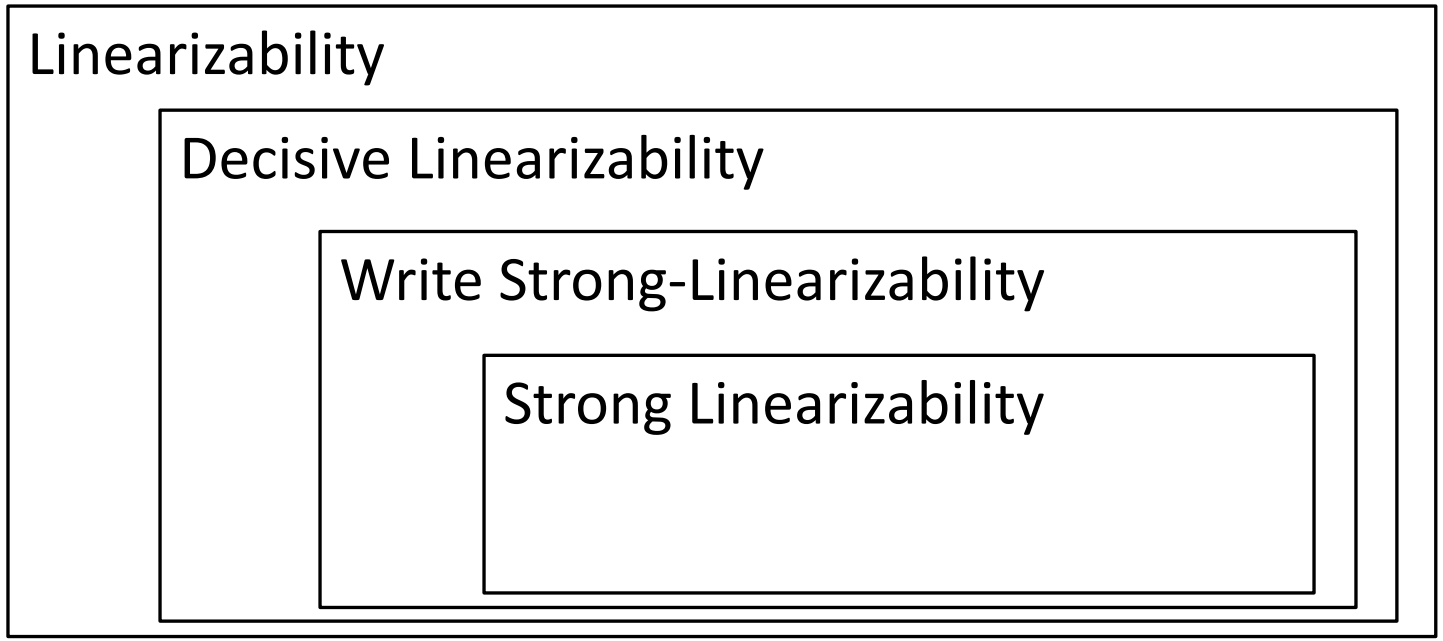


	Lin.	Decisive	Strong
	✓	✓	✓
	✓	✓	✗
	✓	✗	✗

Conclusion

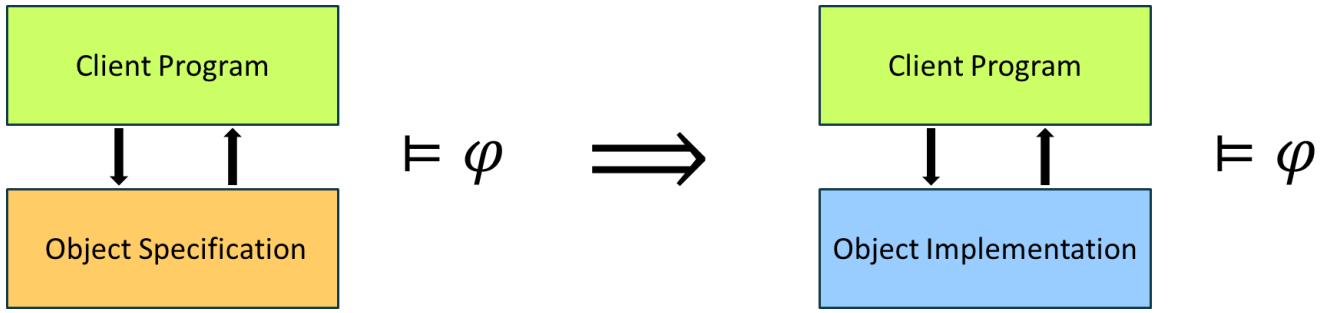
- Simple shared memory register specifications:
  - Write strong register
  - Decisive register
  - General construction (in the paper)
- Enable reasoning about hyperproperties of clients that use implementations satisfying certain linearizability criteria
- New linearizability class: decisive linearizability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

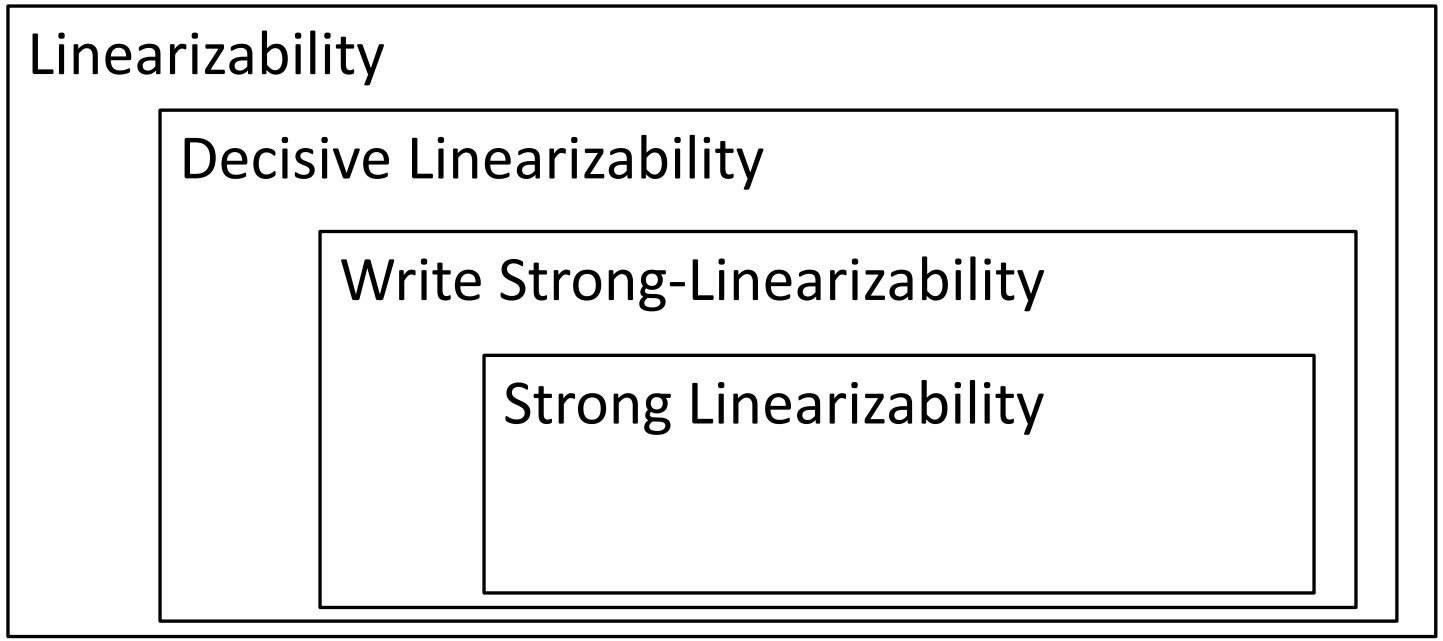




Conclusion

- 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 certain linearizability criteria
- New linearizability class: decisive linearizability
  - Applicable beyond registers





Thank You!