On the Correctness of Optimistic Composable Data Structures

Ahmed Hassan, Sebastiano Peluso, Roberto Palmieri and Binoy Ravindran

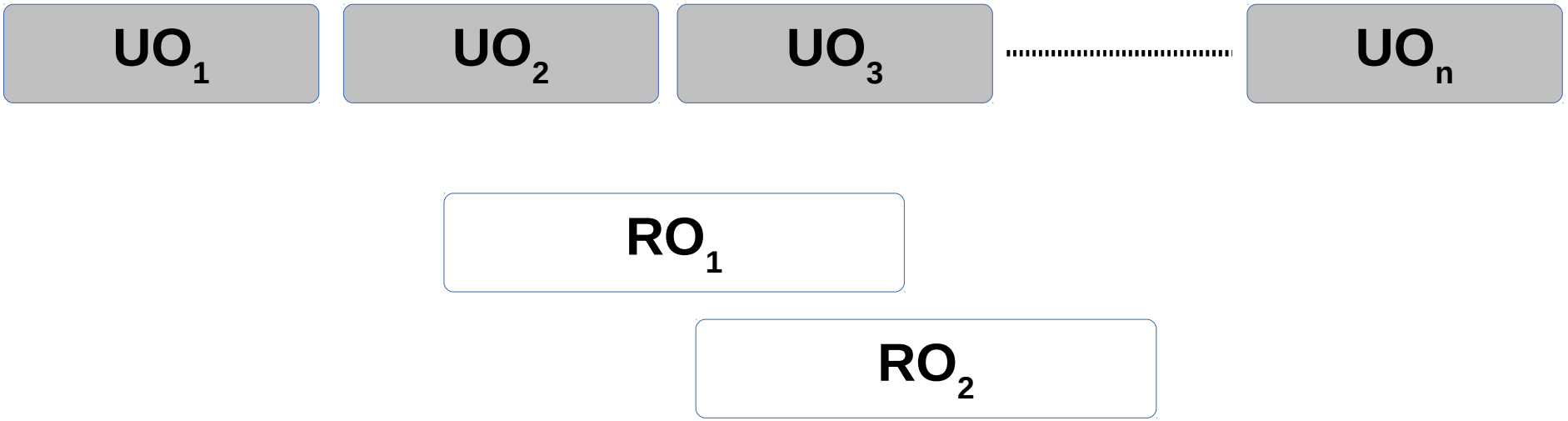
Systems Software Research Group
Virginia Tech

Concurrent Data Structures

- Different Designs and Implementations
  - Different ad-hoc approaches for proving correctness.

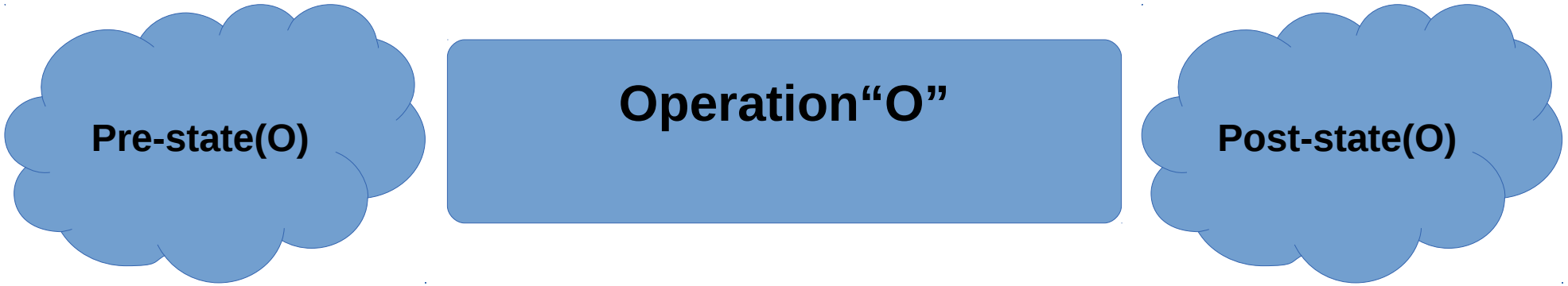
- Is there a unified model for concurrent data structures?
  - General enough.
  - Easy to use.

SWMR Model (Lev-Ari et. Al, DISC'14)



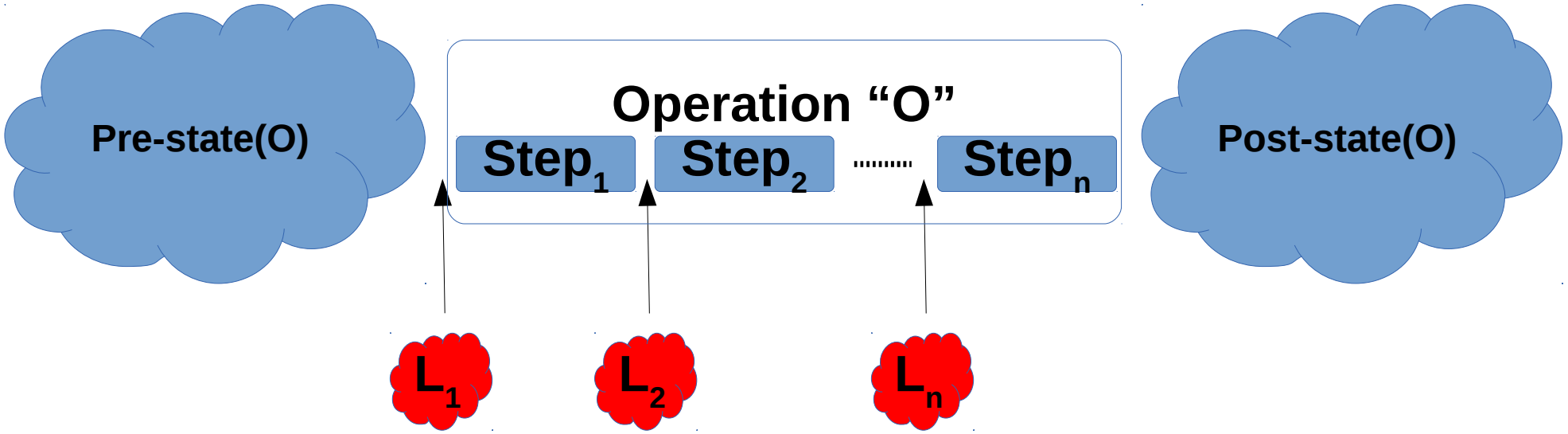
Shared States

- Data Structure is represented as a set of shared variables.
- The values of those variables is the shared state of the data structure.

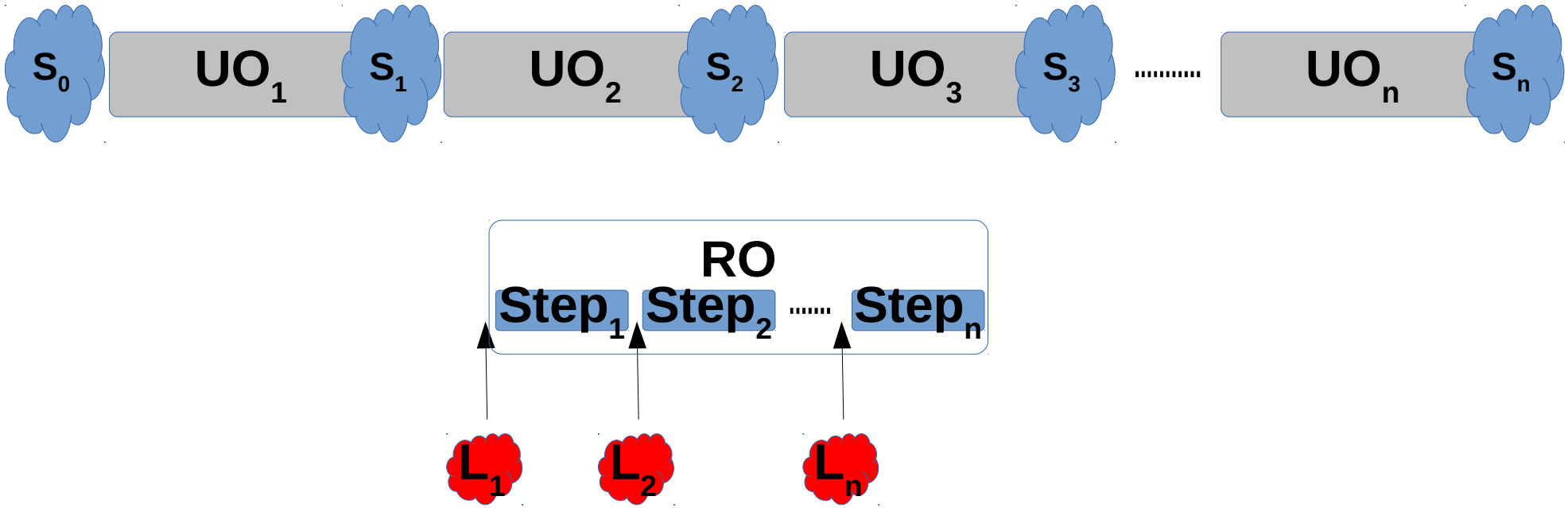


Local States

- Operation is represented as a set of steps.
- The values of the operation's local variables before any step is the local state of the step.

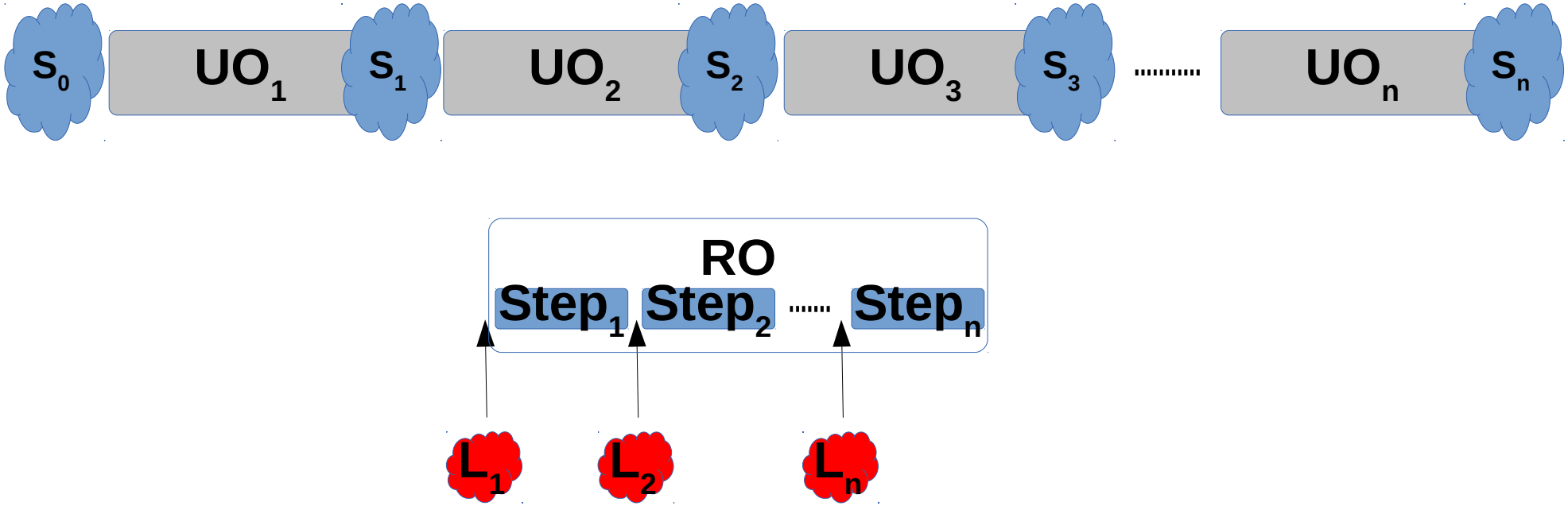


SWMR Scenario

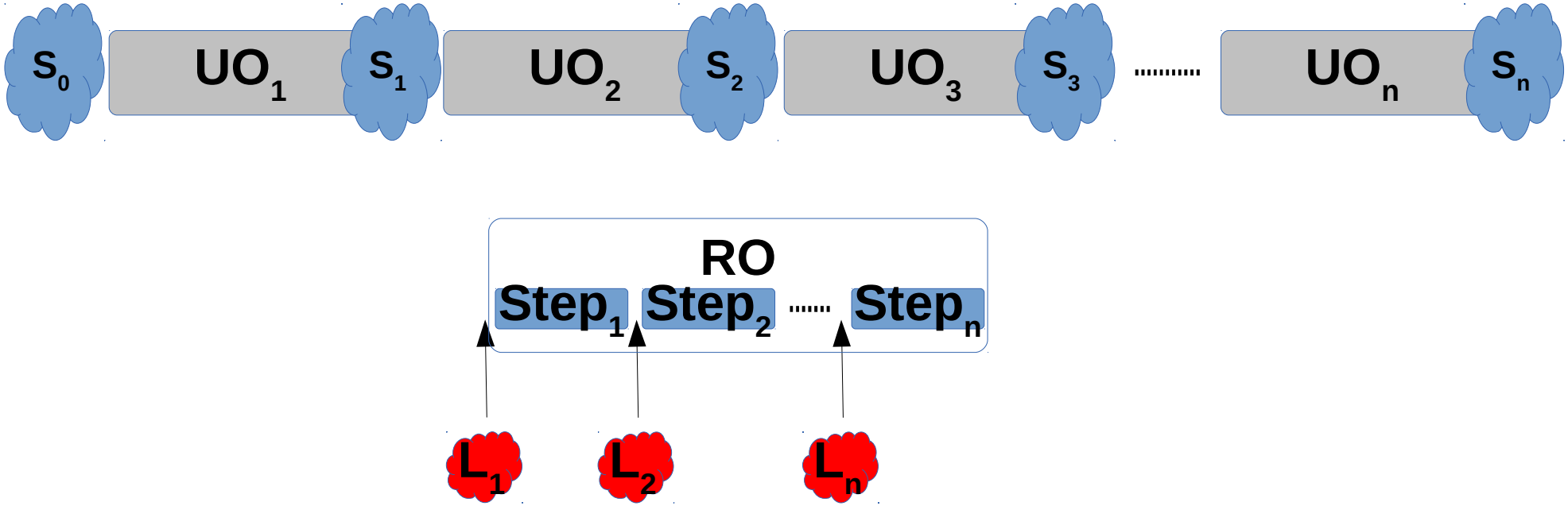


Validity

Validity

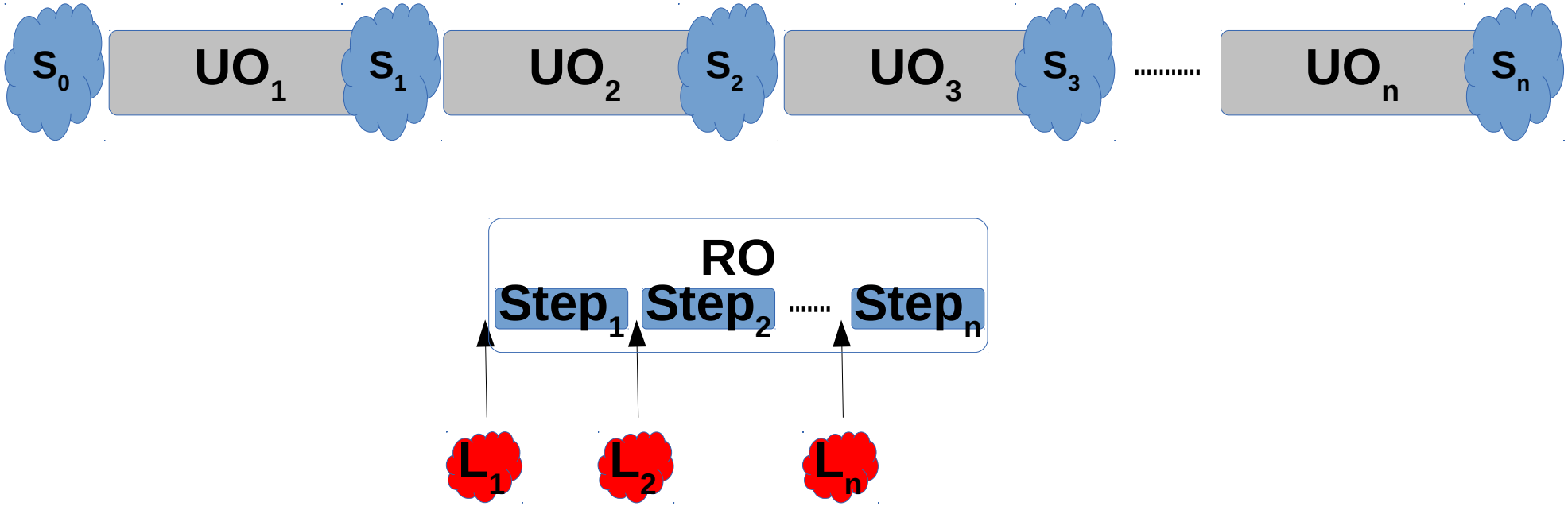


Validity



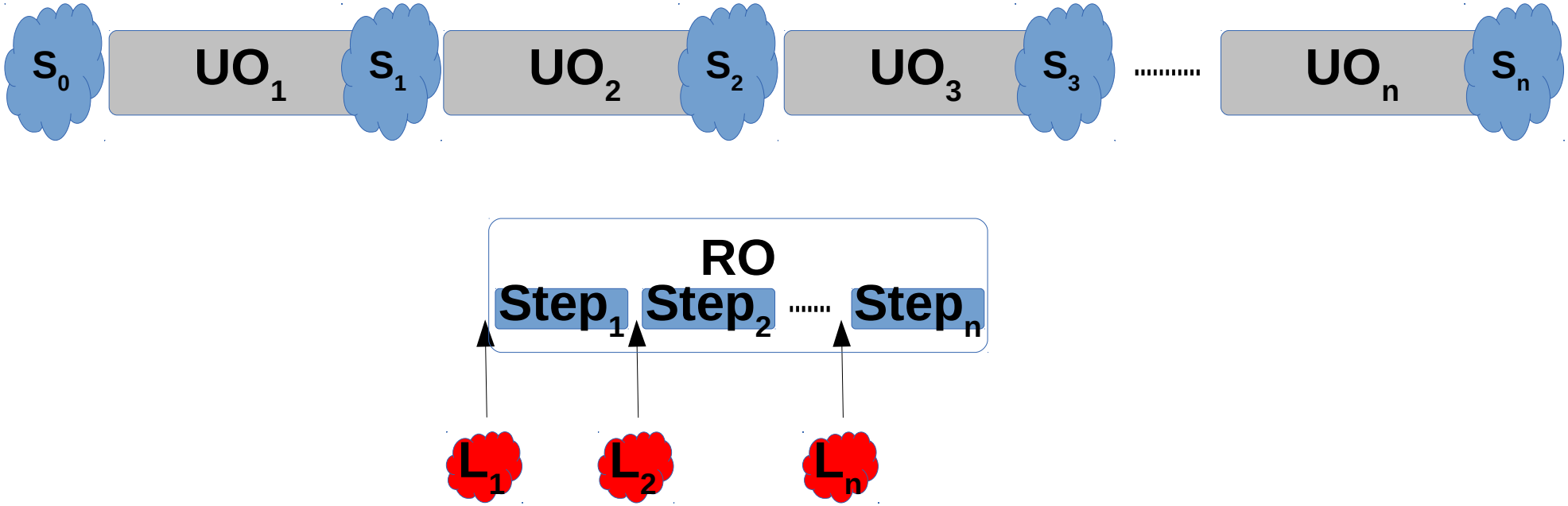
- All S_i are sequentially reachable, so all UO_i are valid.

Validity



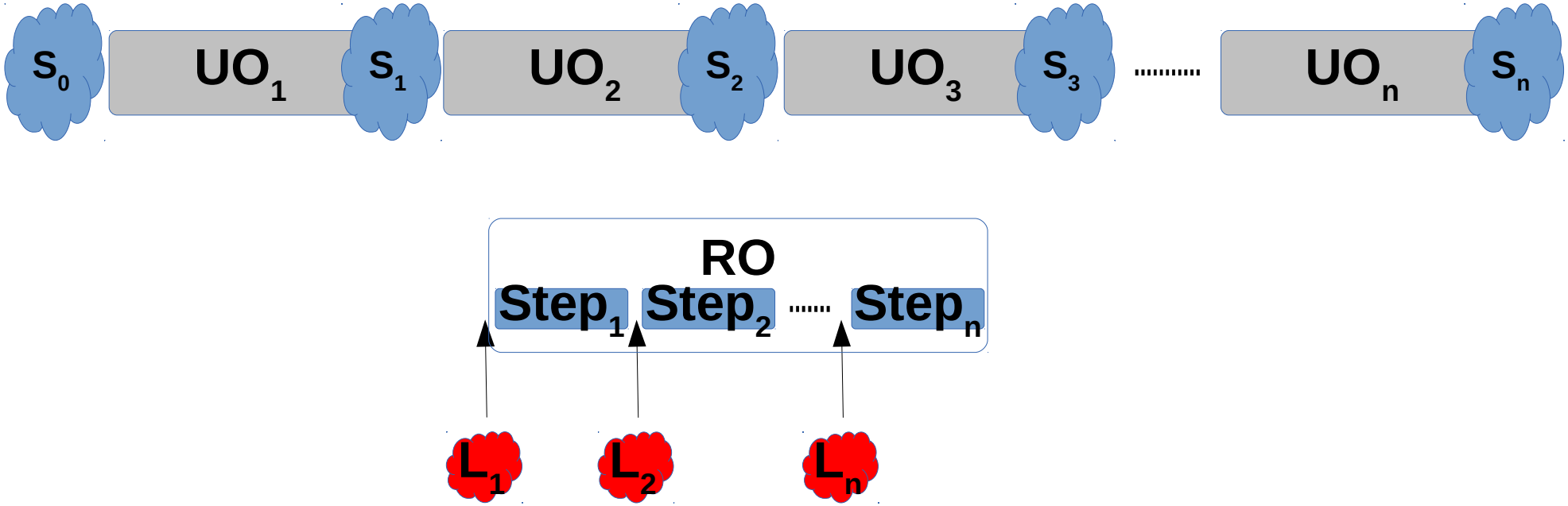
- All S_i are sequentially reachable, so all UO_i are valid.
- $Step_i$ in RO is valid if there is S_j such that a sequential execution of RO starting from S_j reaches L_i.

Validity



- All S_i are sequentially reachable, so all UO_i are valid.
- $Step_j$ in RO is valid if there is S_i such that a sequential execution of RO starting from S_i reaches L_j.

Validity



- All S_i are sequentially reachable, so all UO_i are valid.
- $Step_j$ in RO is valid if there is S_i such that a sequential execution of RO starting from S_i reaches L_j.

↓

- $Step_j$ in RO is valid if there is a "base point" where the "base condition" of $step_j$ holds.

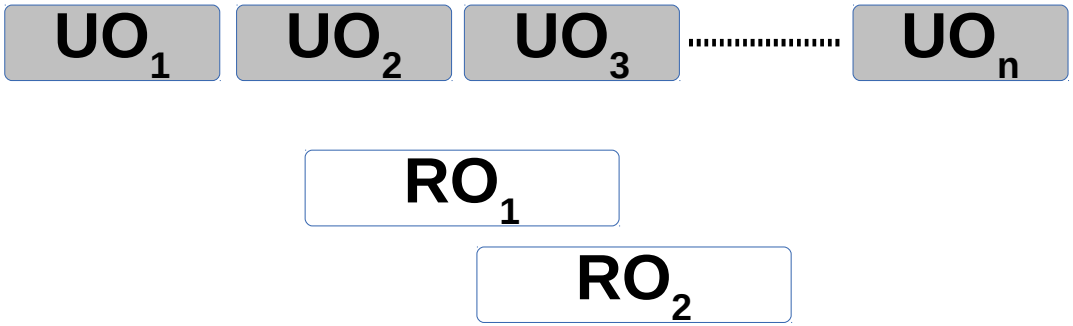
Validity

- How to prove validity for any data structure.
  - Identify the base conditions for each step in each operation (it is sufficient to do so only for steps that access the shared memory).
  - Prove that in any concurrent execution, every step has a base point that satisfies its base condition.

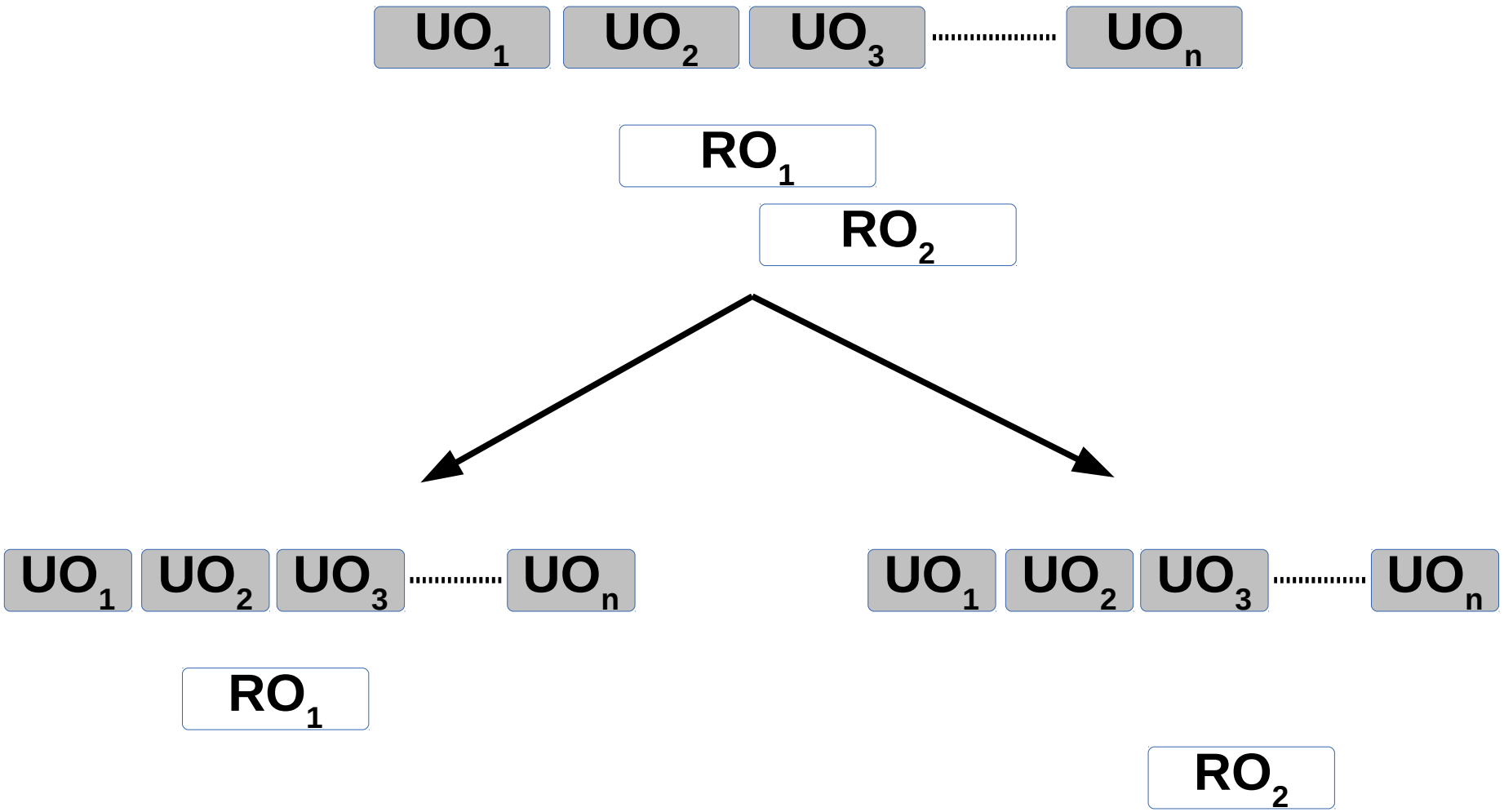
Validity

- How to prove validity for any data structure.
  - Identify the base conditions for each step in each operation (it is sufficient to do so only for steps that access the shared memory).
  - Prove that in any concurrent execution, every step has a base point that satisfies its base condition.

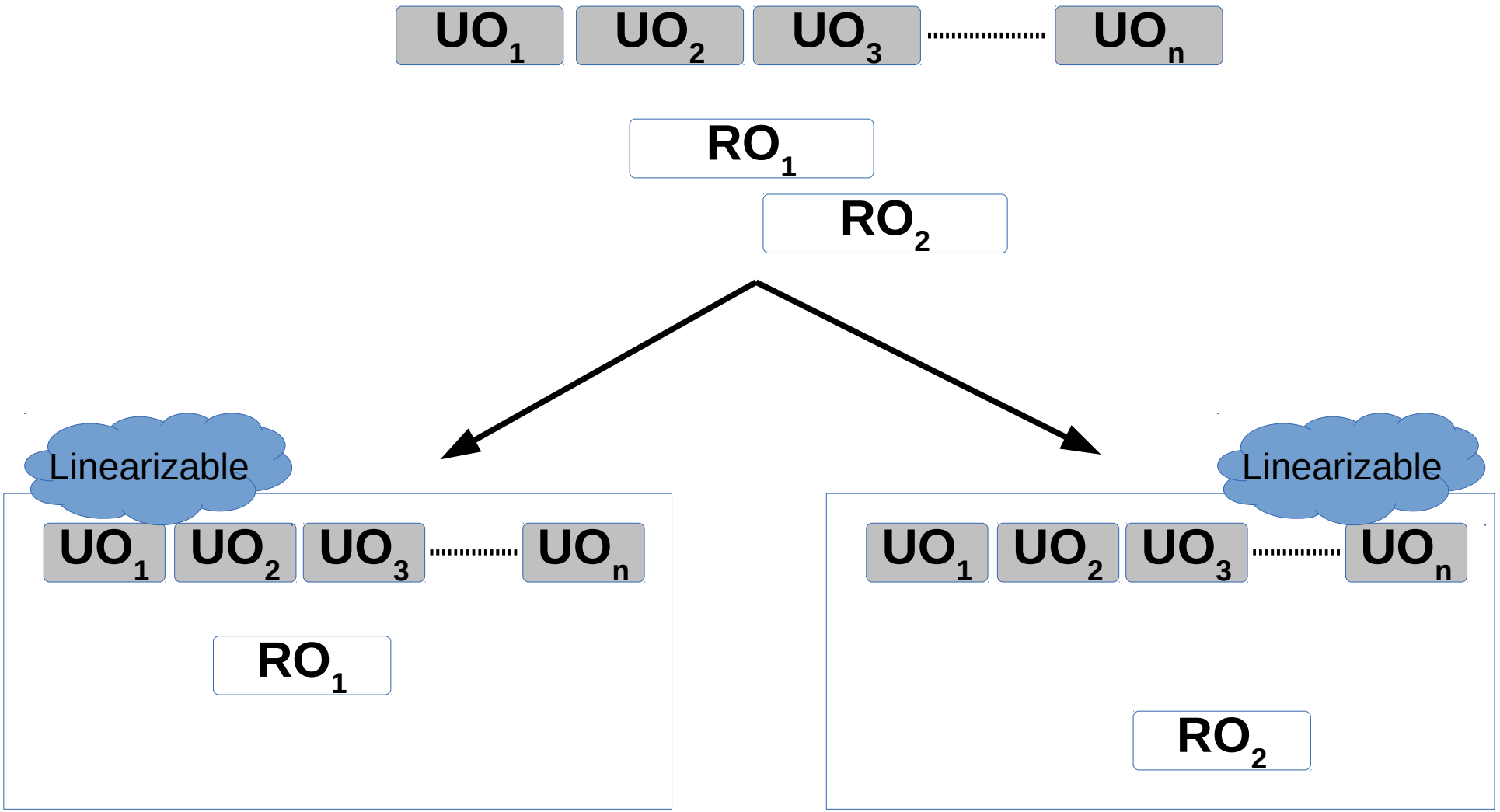
Regularity



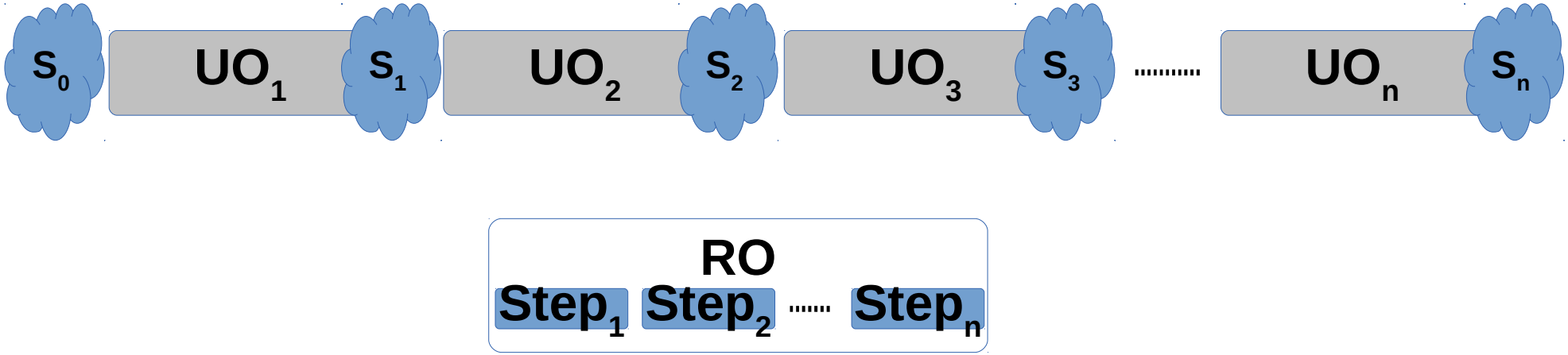
Regularity



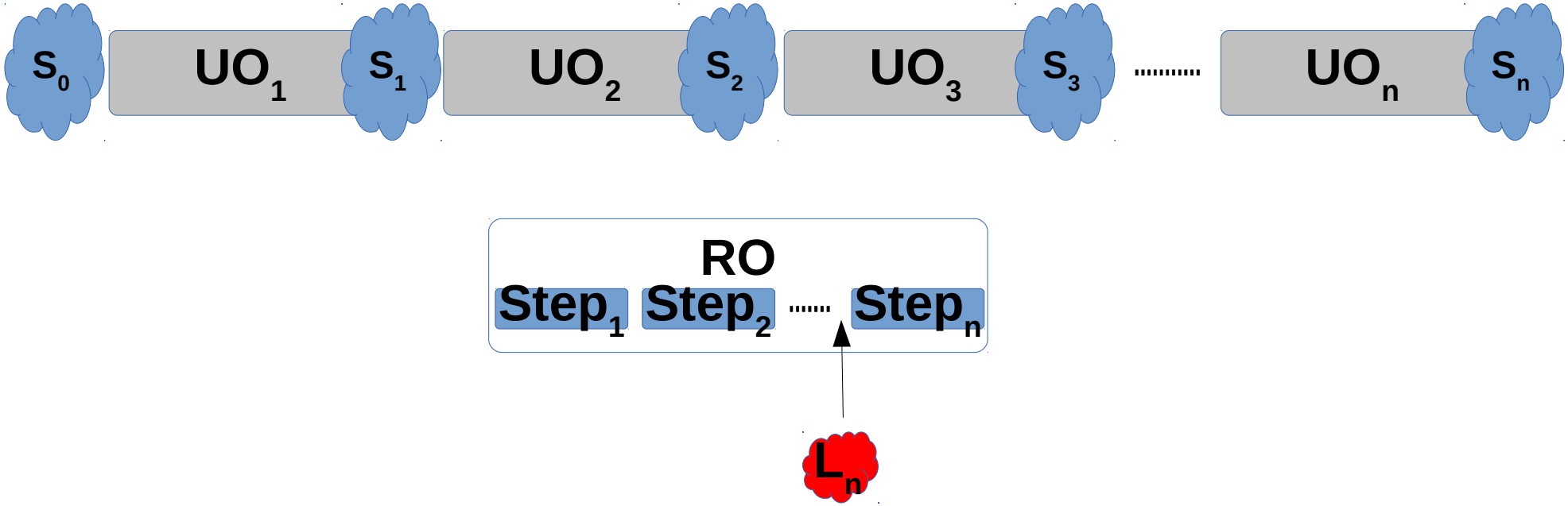
Regularity


UO1
UO2
UO3
UOn
RO1
RO2
Linearizable
UO1
UO2
UO3
UOn
RO1
Linearizable
UO1
UO2
UO3
UOn
RO2

Regularity



Regularity



- Acceptable base points for RO's return step are only S_1, S_2, S_3.
  - Observes either the last update or a concurrent update.

Example

Function *remove(n)*
  $p \leftarrow \perp$
  $next \leftarrow \boldsymbol{read}(head.next)$
  while $next \neq n$
    $p \leftarrow next$
    $next \leftarrow \boldsymbol{read}(p.next)$
  $\boldsymbol{write}(p.next,\ n.next)$

Function *insertLast(n)*
  $last \leftarrow readLast()$
  $\boldsymbol{write}(last.next,\ n)$

Base conditions:

$\Phi_1 : true$

$\Phi_2 : head \overset{*}{\Rightarrow} n$

$\Phi_3 : head \overset{*}{\Rightarrow} n$

Function *readLast()*
  $n \leftarrow \perp$
  $next \leftarrow \boldsymbol{read}(head.next)$
  while $next \neq \perp$
    $n \leftarrow next$
    $next \leftarrow \boldsymbol{read}(n.next)$
  $\boldsymbol{return}(n)$

Example

Function *remove(n)*
  $p \leftarrow \perp$
  $next \leftarrow \boldsymbol{read}(head.next)$
  while $next \neq n$
    $p \leftarrow next$
    $next \leftarrow \boldsymbol{read}(p.next)$
  $\boldsymbol{write}(p.next,\ n.next)$

Function *insertLast(n)*
  $last \leftarrow readLast()$
  $\boldsymbol{write}(last.next,\ n)$

Base conditions:

$\Phi_1 : true$

$\Phi_2 : head \overset{*}{\Rightarrow} n$

$\Phi_3 : head \overset{*}{\Rightarrow} n$

Function *readLast()*
  $n \leftarrow \perp$
  $next \leftarrow \boldsymbol{read}(head.next)$
  while $next \neq \perp$
    $n \leftarrow next$
    $next \leftarrow \boldsymbol{read}(n.next)$
  $\boldsymbol{return}(n)$

Example

Function *remove(n)*
  $p \leftarrow \perp$
  $next \leftarrow \boldsymbol{read}(head.next)$
  while $next \neq n$
    $p \leftarrow next$
    $next \leftarrow \boldsymbol{read}(p.next)$
  $\boldsymbol{write}(p.next,\ n.next)$

Function *insertLast(n)*
  $last \leftarrow readLast()$
  $\boldsymbol{write}(last.next,\ n)$

Base conditions:

$\Phi_1 : true$

$\Phi_2 : head \overset{*}{\Rightarrow} n$

$\Phi_3 : head \overset{*}{\Rightarrow} n$

Function *readLast()*
  $n \leftarrow \perp$
  $next \leftarrow \boldsymbol{read}(head.next)$
  while $next \neq \perp$
    $n \leftarrow next$
    $next \leftarrow \boldsymbol{read}(n.next)$
  $\boldsymbol{return}(n)$

Where is the Problem?

It covers only single-writer designs

It does not cover composable designs

Can we cover a wider set?

Optimistic Composable Data Structures

Optimistic Data Structures

Optimistic Data Structures

Concurrent Operation (add, remove, contains, ...)

Optimistic Data Structures

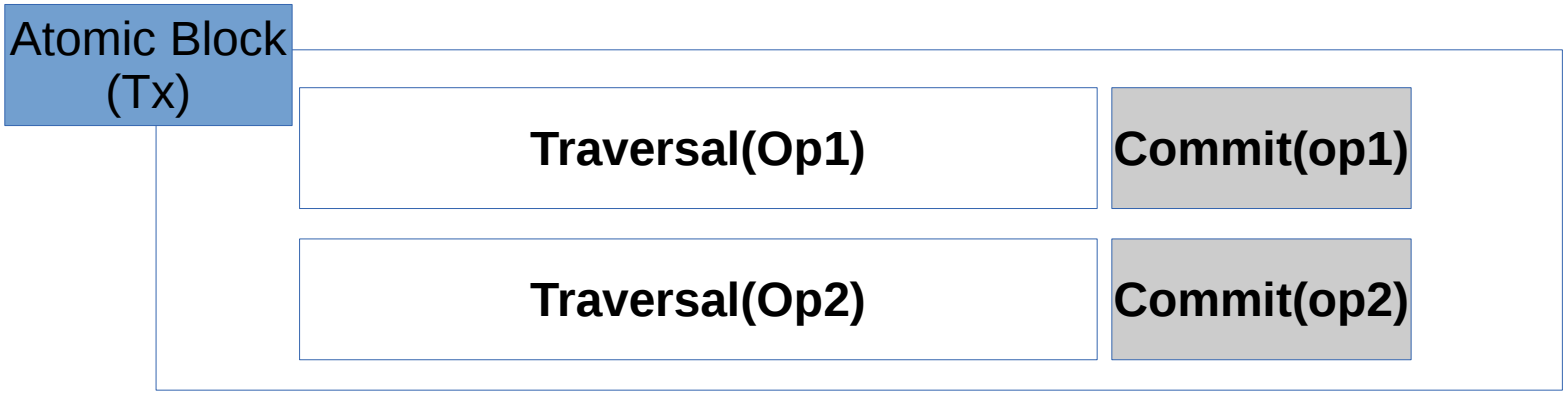
Concurrent Operation (add, remove, contains, ...)

Traversal
(long - unmonitored)

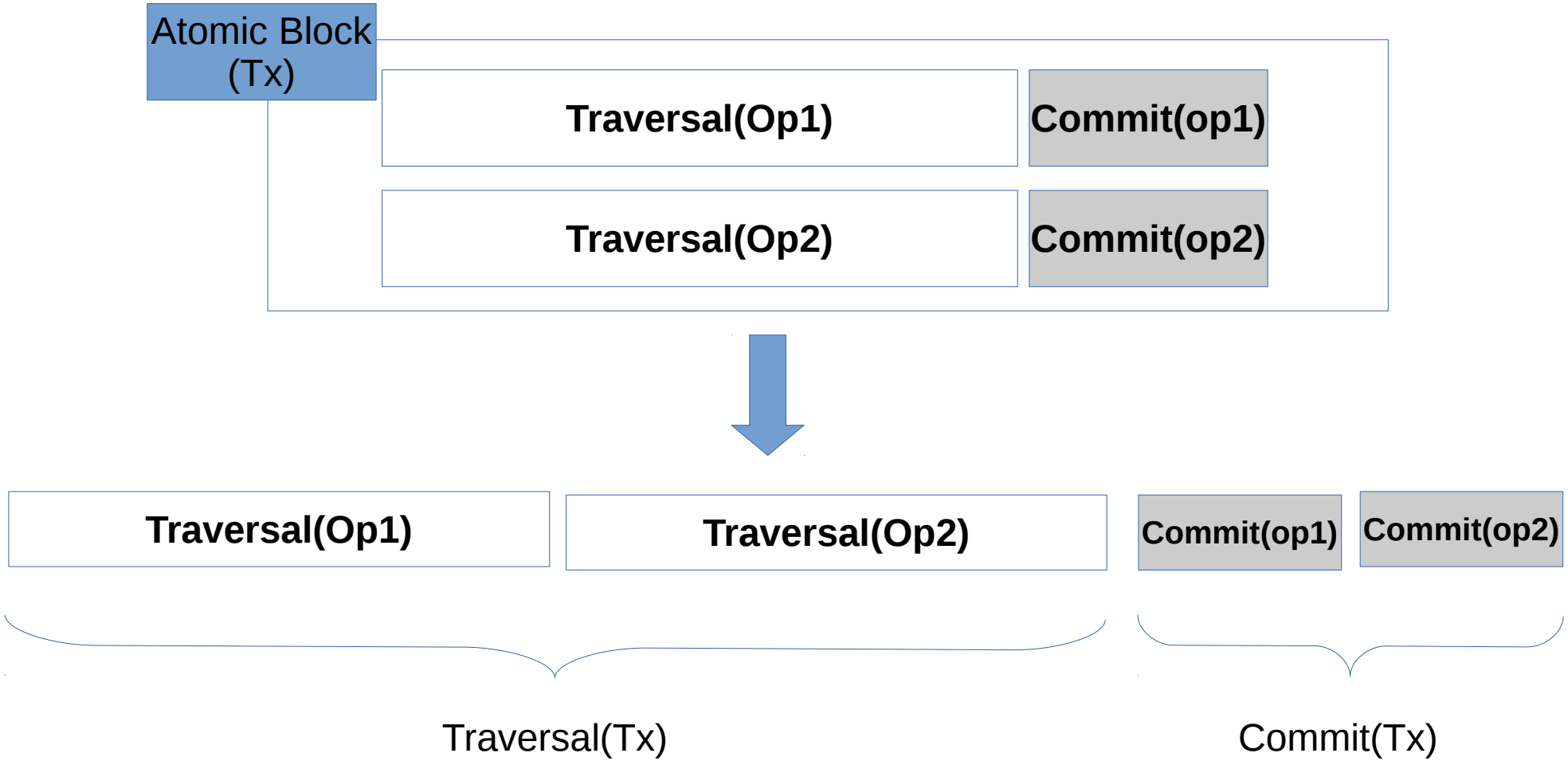
Commit
(short - monitored)

Composable Data Structures

Composable Data Structures



Composable Data Structures

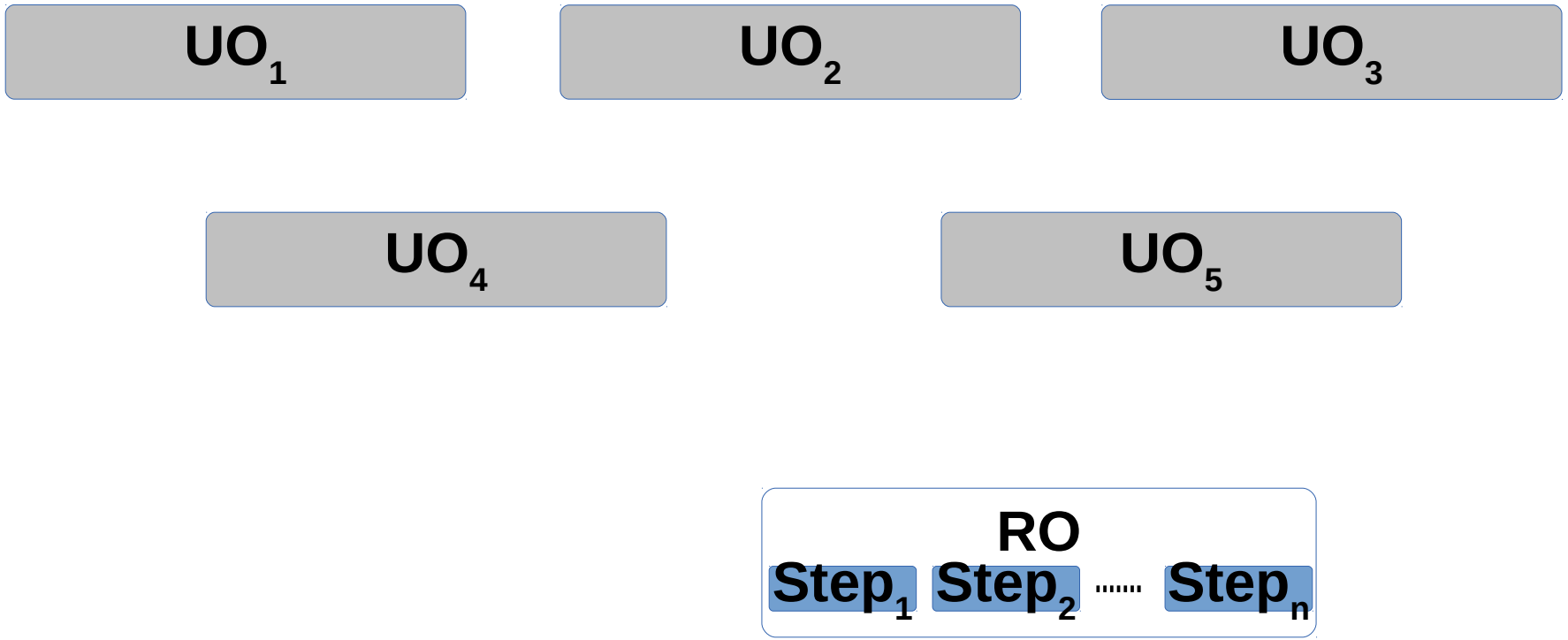


Our Models

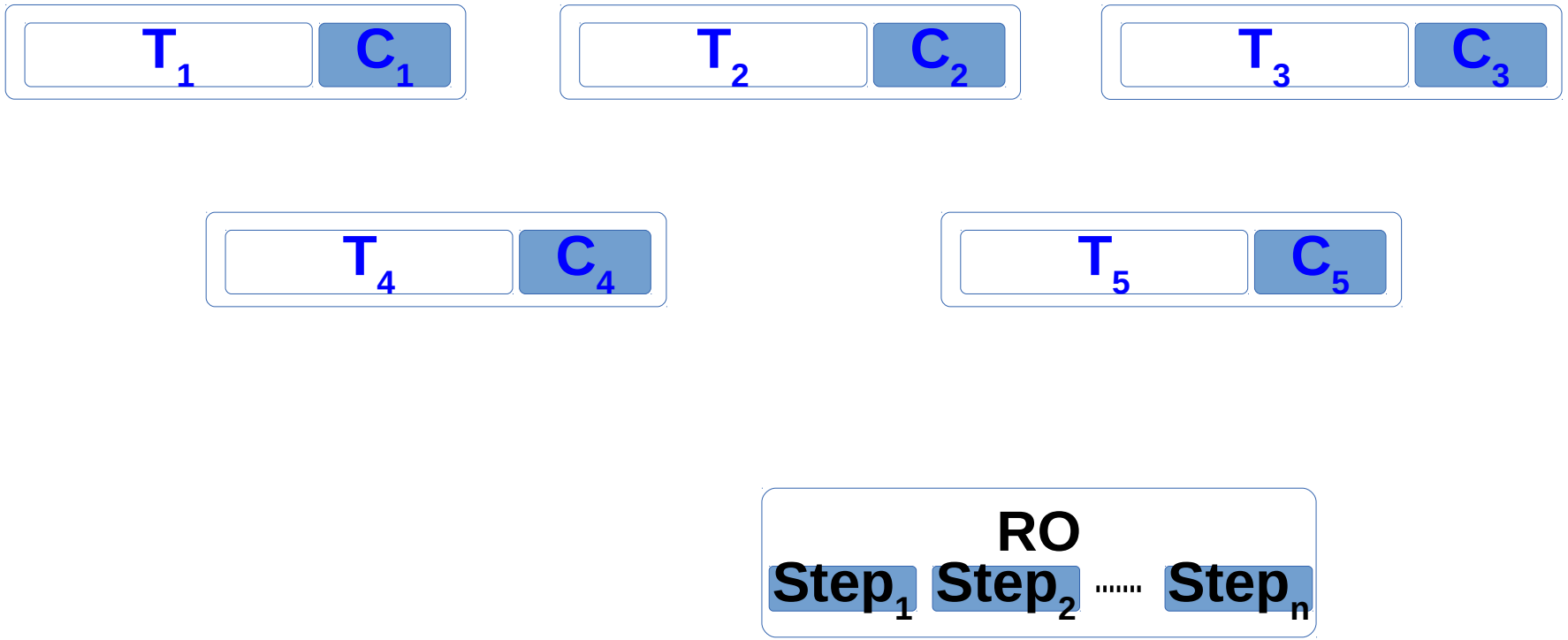
Single Writer Commit (SWC)

Composable SWC (C-SWC)

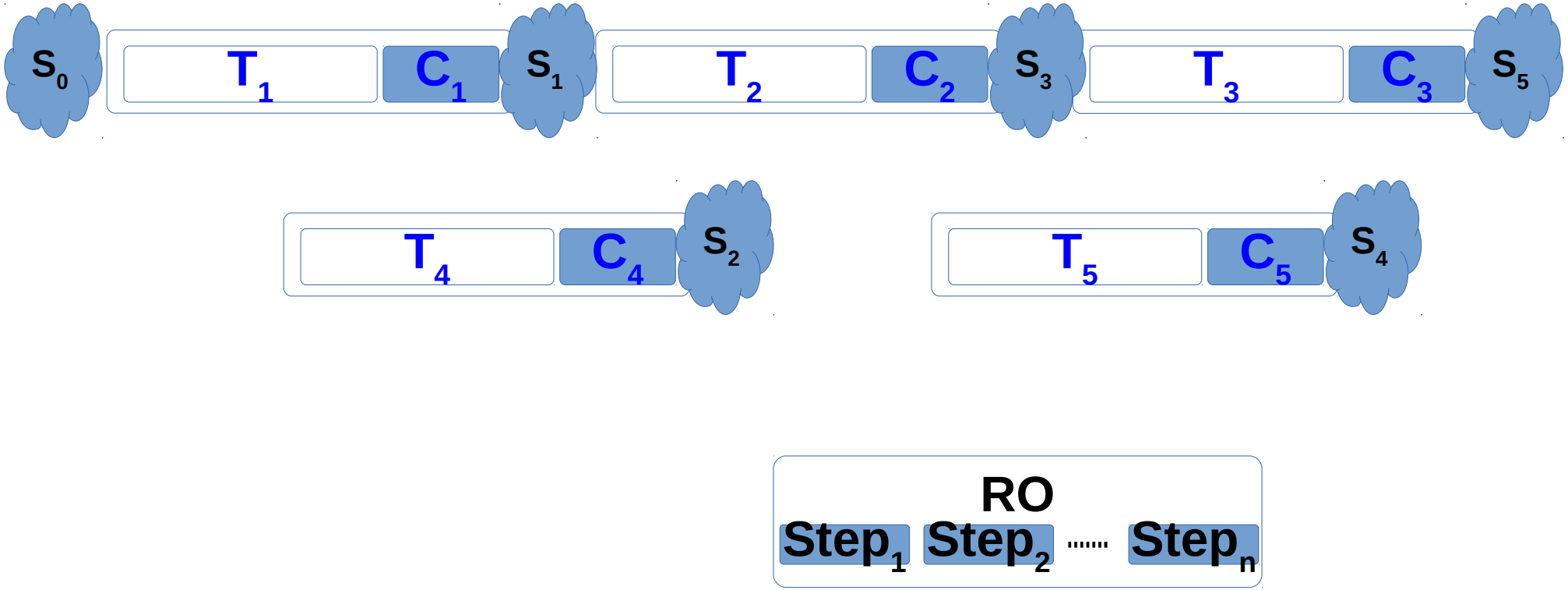
SWC Model



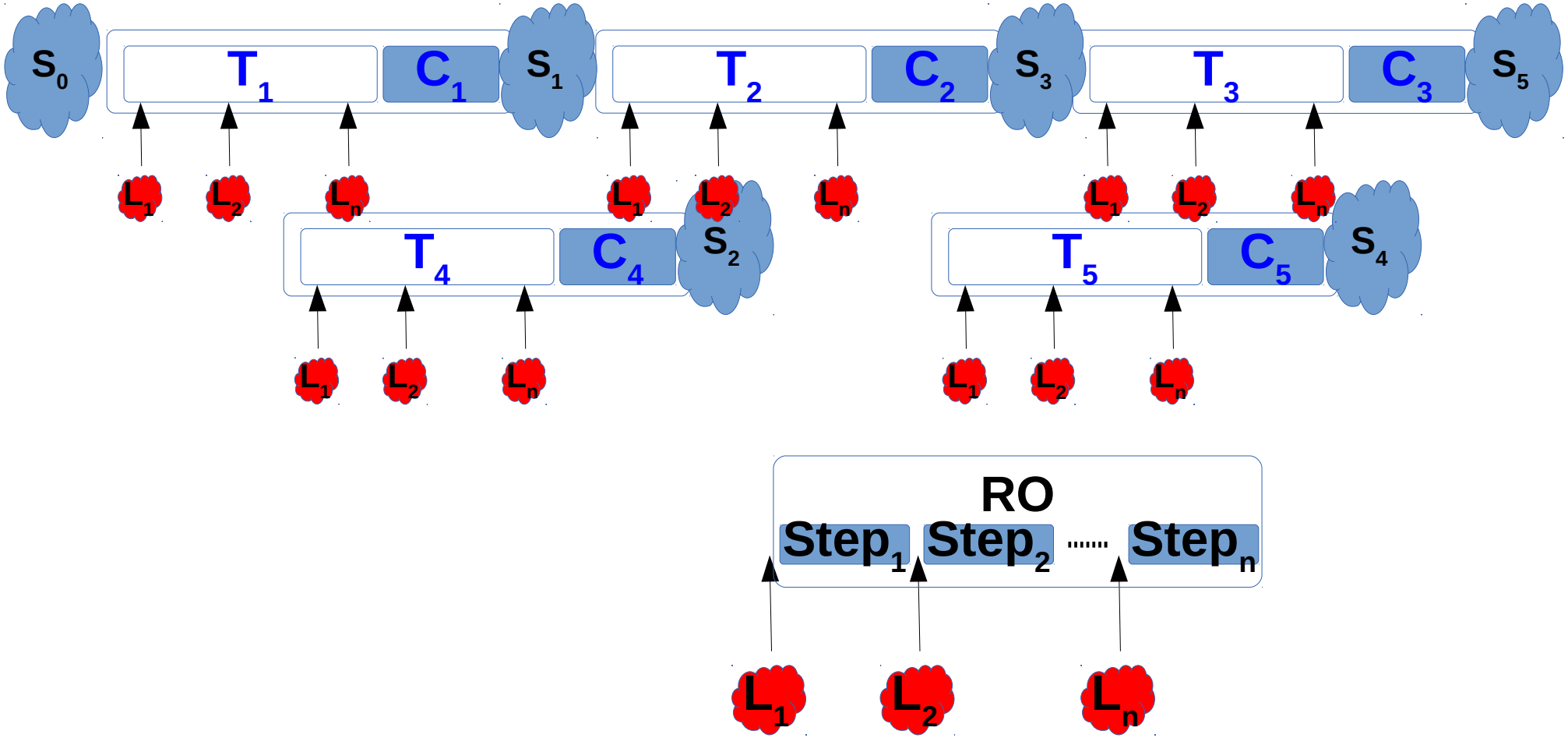
SWC Model



SWC Model



SWC Model


S0
T1
C1
S1
T2
C2
S3
T3
C3
S5
L1
L2
Ln
T4
C4
S2
T5
C5
S4
RO
Step1
Step2
.....
Stepn

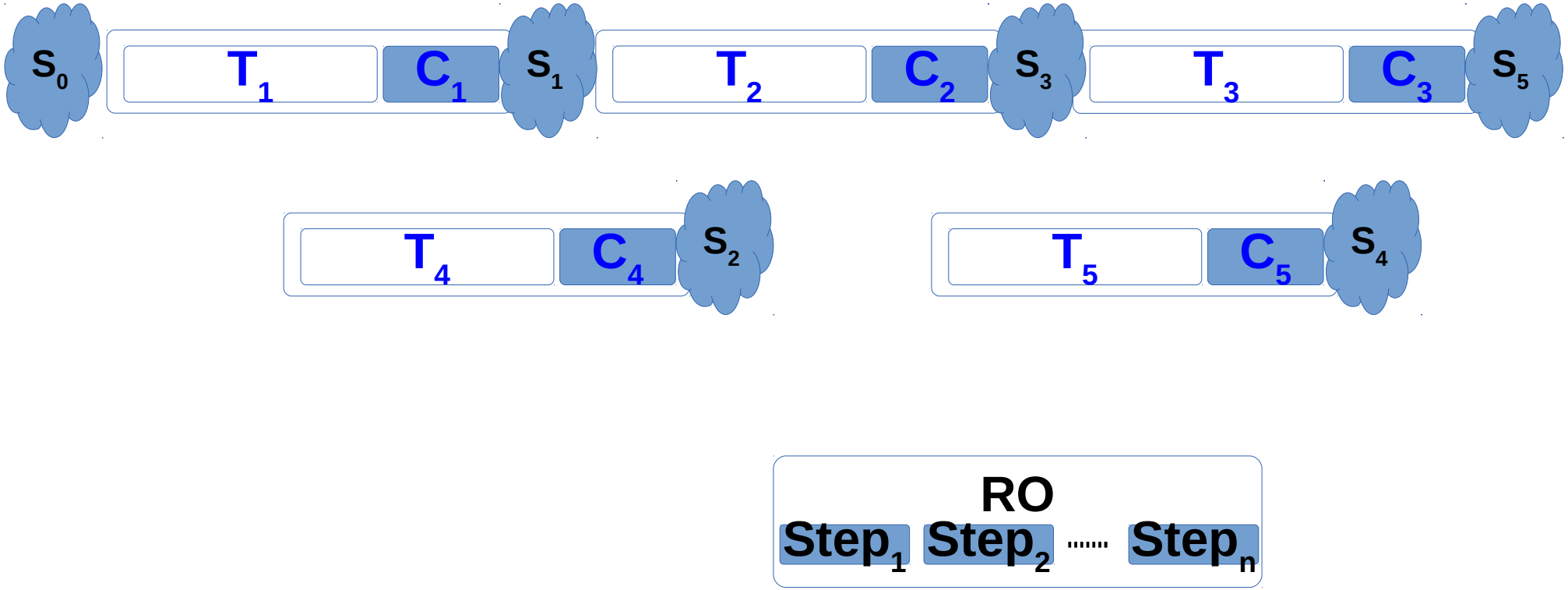
Even More...

- Do we really need single commit at a time:
  - NO!!!
  - It is enough to execute commit phases atomically with *single lock atomicity (SLA)* guarantees.
  - More practical alternatives:
    - HTM (e.g. Intel TSX).
    - STM (e.g. NOrec “the SLA version”).

Validity

- Guarantee that all S_i's are sequentially reachable.
  - Comes for free in the SWMR!
  - In SWC: Prove that S_{i-1} is the base point of the first step of S_i's commit phase.
- Watch your step
  - In RO AND the traversal phase of all UO's

Regularity



- Acceptable base points for RO's return step are only S_2, S_3, S_4.
  - Observes either the last commit or a concurrent commit.

Example

```
1:  procedure READLAST
2:      last ←⊥
3:      next ← read(head.next)                    ▷ φ1 : true
4:      while next ≠⊥ do
5:          last ← next
6:          next ← read(last.next)                ▷ φ2 : head ⇒* last
7:      return(last)                              ▷ φ3 : head ⇒* last
8:  end procedure

9:  procedure INSERTLAST(n)
10:     last ←⊥
11:     next ← read(head.next)                    ▷ φ4 : true
12:     while next ≠⊥ do
13:         last ← next
14:         next ← read(last.next)                ▷ φ5 : head ⇒* last
15:     lockAcquire(gl)
                                                  ▷ φ6 : head ⇒* last
16:     if read(last.next) ≠⊥ then
17:         lockRelease(gl)
18:         go to 10
19:     write(last.next, n)
20:     lockRelease(gl)
21: end procedure
```

Example

```
1:  procedure READLAST
2:      last ← ⊥
3:      next ← read(head.next)                    ▷ φ1 : true
4:      while next ≠ ⊥ do
5:          last ← next
6:          next ← read(last.next)                ▷ φ2 : head ⇒* last
7:      return(last)                              ▷ φ3 : head ⇒* last
8:  end procedure

9:  procedure INSERTLAST(n)
10:     last ← ⊥
11:     next ← read(head.next)                    ▷ φ4 : true
12:     while next ≠ ⊥ do
13:         last ← next
14:         next ← read(last.next)                ▷ φ5 : head ⇒* last
15:     lockAcquire(gl)
                                                  ▷ φ6 : head ⇒* last
16:     if read(last.next) ≠ ⊥ then
17:         lockRelease(gl)
18:         go to 10
19:     write(last.next, n)
20:     lockRelease(gl)
21: end procedure
```

Example

```
1:  procedure READLAST
2:      last ← ⊥
3:      next ← read(head.next)
4:      while next ≠ ⊥ do
5:          last ← next
6:          next ← read(last.next)
7:      return(last)
8:  end procedure

9:  procedure INSERTLAST(n)
10:     last ← ⊥
11:     next ← read(head.next)
12:     while next ≠ ⊥ do
13:         last ← next
14:         next ← read(last.next)
15:     lockAcquire(gl)

16:     if read(last.next) ≠ ⊥ then
17:         lockRelease(gl)
18:         go to 10
19:     write(last.next, n)
20:     lockRelease(gl)
21: end procedure
```

$\triangleright\ \phi_1 : true$

$\triangleright\ \phi_2 : head \overset{*}{\Rightarrow} last$

$\triangleright\ \phi_3 : head \overset{*}{\Rightarrow} last$

$\triangleright\ \phi_4 : true$

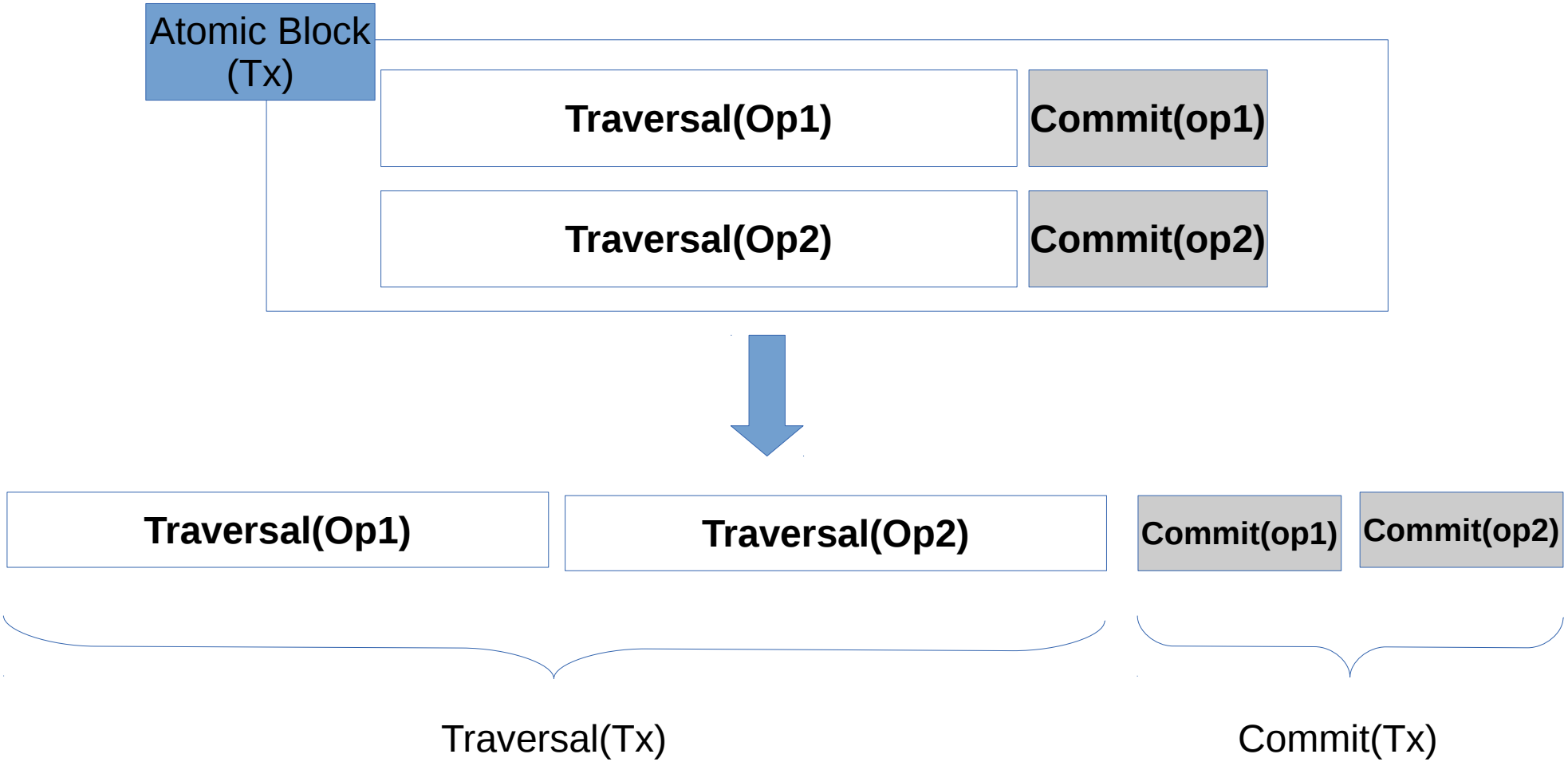
$\triangleright\ \phi_5 : head \overset{*}{\Rightarrow} last$

$\triangleright\ \phi_6 : head \overset{*}{\Rightarrow} last$

Composable SWC Model (C-SWC)

```
procedure ATOMIC:T1
    x = 5
    if readLast() ≠ x then
        insertLast(x)
    if readLast() ≠ x then
        ... // illegal execution
end procedure
```

Composable SWC Model (C-SWC)

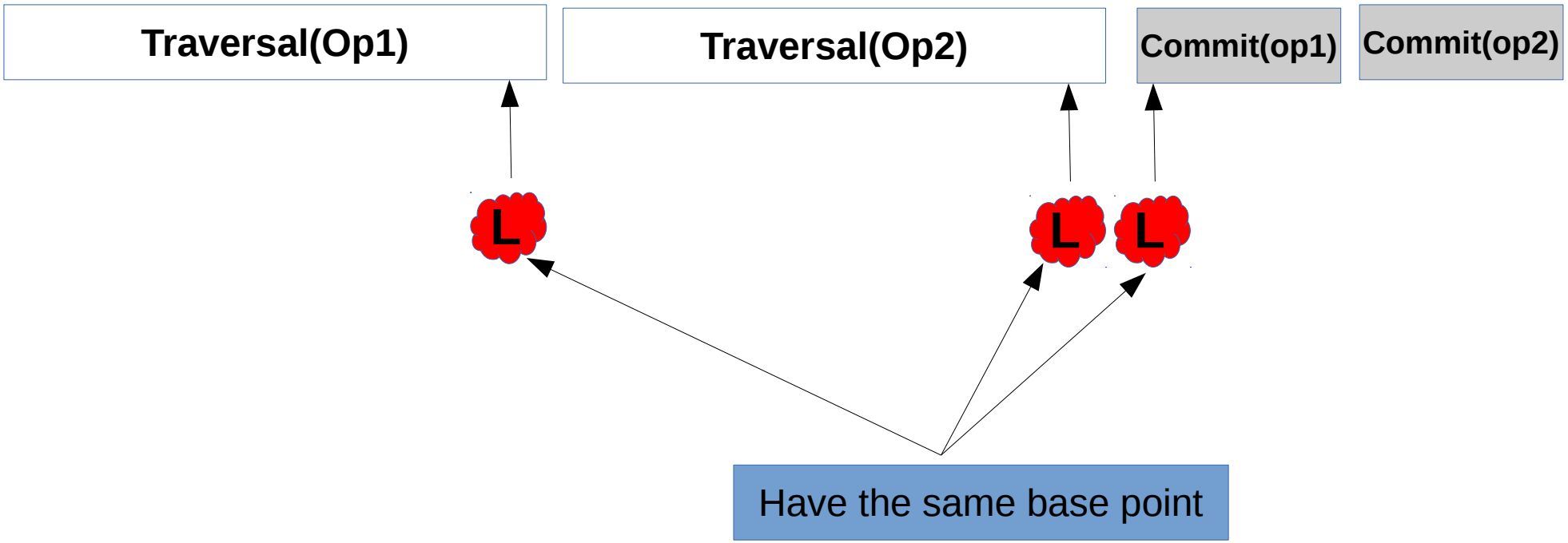


What is remaining?

- Internal Consistency.
  - The commit phase of each operation reflects what the operation observed in its traversal.
  - The shared state of an operation is visible to subsequent operations in the same transaction.

How to prove internal consistency?

How to prove internal consistency?



Related Work

- SWMR Model (Lev-Ari et. al, DISC'14)
  - SWC is a superset.
- MWMR Model (Shao et. Al, SIAM'11)
  - Lattice of consistency levels.
  - SWC corresponds to MWReg.
- LS-Linearizability (Gramoli et. Al, PODC'12)
  - Local serializability instead of validity.
  - Linearizability instead of regularity.
  - SWC is less conservative.

Conclusion

- SWMR model is the first step towards a general modeling of concurrent data structures, but it only covers:
  - Single writer designs.
  - Concurrent (non-composable) designs.

- SWC Model: allows multiple writers with SLA-based commit phases.

- C-SWC: extends SWC to allow operations composition.

Thanks!

Questions?