Inductive Sequentialization of Asynchronous Programs

Bernhard Kragl
IST Austria

Constantin Enea
Université de Paris

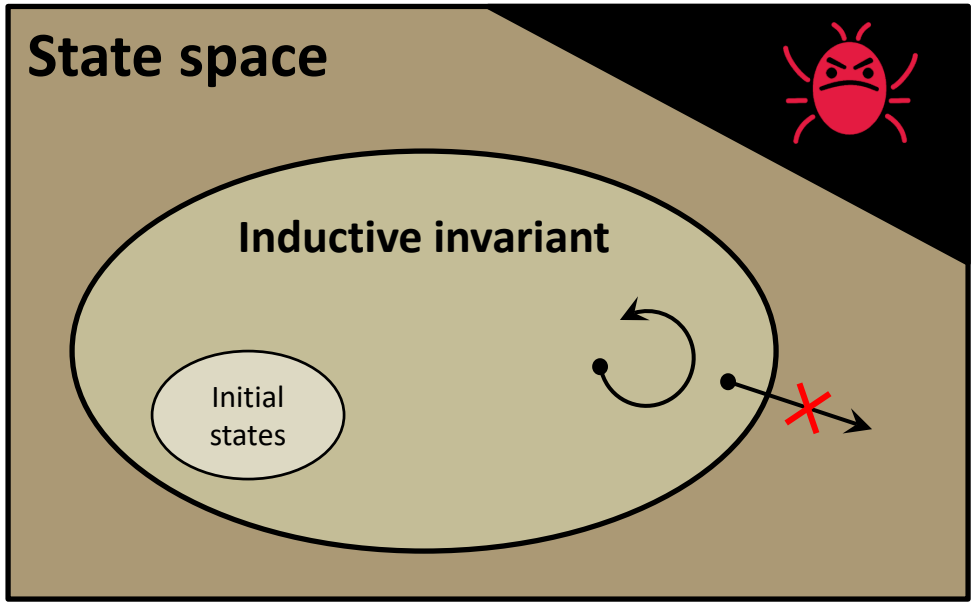
Thomas A. Henzinger
IST Austria

Suha Orhun Mutluergil
Université de Paris

Shaz Qadeer
Novi

Related work: IronFleet, Verdi, CertiKOS, QED, CIVL, CSPEC, Armada, Ivy, Pretend Synchrony, ATHOS, Disel, Iris/Aneris, …

The Invariant Challenge



The Invariant Challenge

Paxos consensus property

$$\forall n_1, n_2 : \text{node}, r_1, r_2 : \text{round}, v_1, v_2 : \text{value}.\ \mathit{decision}(n_1, r_1, v_1) \wedge \mathit{decision}(n_2, r_2, v_2) \rightarrow v_1 = v_2 \quad (4)$$

$$\forall r : \text{round}, v_1, v_2 : \text{value}.\ \mathit{propose\_msg}(r, v_1) \wedge \mathit{propose\_msg}(r, v_2) \rightarrow v_1 = v_2 \quad (5)$$

$$\forall n : \text{node}, r : \text{round}, v : \text{value}.\ \mathit{vote\_msg}(n, r, v) \rightarrow \mathit{propose\_msg}(r, v) \quad (6)$$

$$\forall r : \text{round}, v : \text{value}.\ (\exists n : \text{node}.\ \mathit{decision}(n, r, v)) \rightarrow \exists q : \text{quorum}. \forall n : \text{node}.\ \mathit{member}(n, q) \rightarrow \mathit{vote\_msg}(n, r, v) \quad (7)$$

$$\forall n : \text{node}, r, r' : \text{round}, v, v' : \text{value}.\ \mathit{join\_ack\_msg}(n, r, \bot, v) \wedge r' < r \rightarrow \neg \mathit{vote\_msg}(n, r', v') \quad (8)$$

$$\forall n : \text{node}, r, r' : \text{round}, v : \text{value}.\ \mathit{join\_ack\_msg}(n, r, r', v) \wedge r' \neq \bot \rightarrow r' < r \wedge \mathit{vote\_msg}(n, r', v) \quad (9)$$

$$\forall n : \text{node}, r, r', r'' : \text{round}\ v, v' : \text{value}.\ \mathit{join\_ack\_msg}(n, r, r', v) \wedge r' \neq \bot \wedge r' < r'' < r \rightarrow \neg \mathit{vote\_msg}(n, r'', v') \quad (10)$$

$$\forall n : \text{node}, v : \text{value}.\ \neg \mathit{vote\_msg}(n, \bot, v) \quad (11)$$

$$\forall r_1, r_2 : \text{round}\ v_1, v_2 : \text{value}, q : \text{quorum}.\ \mathit{propose\_msg}(r_2, v_2) \wedge r_1 < r_2 \wedge v_1 \neq v_2 \rightarrow \exists n : \text{node}, r', r'' : \text{round}, v : \text{value}.\ \mathit{member}(n, q) \wedge \neg \mathit{vote\_msg}(n, r_1, v_1) \wedge r' > r_1 \wedge \mathit{join\_ack\_msg}(n, r', r'', v) \quad (12)$$

Source: Padon et al., OOPSLA 2017

The Invariant Challenge

Paxos consensus property

$$\forall n_1, n_2 : \text{node}, r_1, r_2 : \text{round}, v_1, v_2 : \text{value}.\ \mathit{decision}(n_1, r_1, v_1) \wedge \mathit{decision}(n_2, r_2, v_2) \rightarrow v_1 = v_2 \quad (4)$$

$$\forall r : \text{round}, v_1, v_2 : \text{value}.\ \mathit{propose\_msg}(r, v_1) \wedge \mathit{propose\_msg}(r, v_2) \rightarrow v_1 = v_2 \quad (5)$$

$$\forall n : \text{node}, r : \text{round}, v : \text{value}.\ \mathit{vote\_msg}(n, r, v) \rightarrow \mathit{propose\_msg}(r, v) \quad (6)$$

$$\forall r : \text{round}, v : \text{value}.\ (\exists n : \text{node}.\ \mathit{decision}(n, r, v)) \rightarrow \exists q : \text{quorum}. \forall n : \text{node}.\ \mathit{member}(n, q) \rightarrow \mathit{vote\_msg}(n, r, v) \quad (7)$$

$$\forall n : \text{node}, r, r' : \text{round}, v, v' : \text{value}.\ \mathit{join\_ack\_msg}(n, r, \bot, v) \wedge r' < r \rightarrow \neg \mathit{vote\_msg}(n, r', v') \quad (8)$$

$$\forall n : \text{node}, r, r' : \text{round}, v : \text{value}.\ \mathit{join\_ack\_msg}(n, r, r', v) \wedge r' \neq \bot \rightarrow r' < r \wedge \mathit{vote\_msg}(n, r', v) \quad (9)$$

$$\forall n : \text{node}, r, r', r'' : \text{round}, v' : \text{value}.\ \mathit{join\_ack\_msg}(n, r, r', v) \wedge r' \neq \bot \wedge r'' < r \rightarrow \neg \mathit{vote\_msg}(n, r'', v') \quad (10)$$

$$\forall n : \text{node}, v : \text{value}.\ \neg \mathit{vote\_msg}(n, \bot, v) \quad (11)$$

$$\forall r_1, r_2 : \text{round}\ v_1, v_2 : \text{value}\ \ldots\text{quorum}.\ \mathit{propose\_msg}(r_2, v_2) \wedge r_1 < r_2 \wedge \ldots \rightarrow$$
$$\exists n : \text{node}, r', r'' : \ldots\text{round}, v : \text{value}.\ \mathit{member}(n, q) \wedge \neg \mathit{vote\_msg}(n, r_1, v_1) \wedge r' > r_1 \wedge \mathit{join\_ack\_msg}(n, r', r'', v) \quad (12)$$

Source: Padon et al., OOPSLA 2017

I like it when my concurrent programs execute sequentially.

StartRound(1) Join(1,1) Join(1,2) Propose(1) Vote(1,1,_) Vote(1,2,_) Conclude(1,_)

StartRound(2) Join(2,1) Join(2,2) Propose(2) Vote(2,1,_) Vote(2,2,_) Conclude(2,_)

StartRound(3) Join(3,1) Join(3,2) Propose(3) Vote(3,1,_) Vote(3,2,_) Conclude(3,_)

…

Contributions

Inductive Sequentialization

eliminating concurrency from asynchronous programs

Refinement Methodology

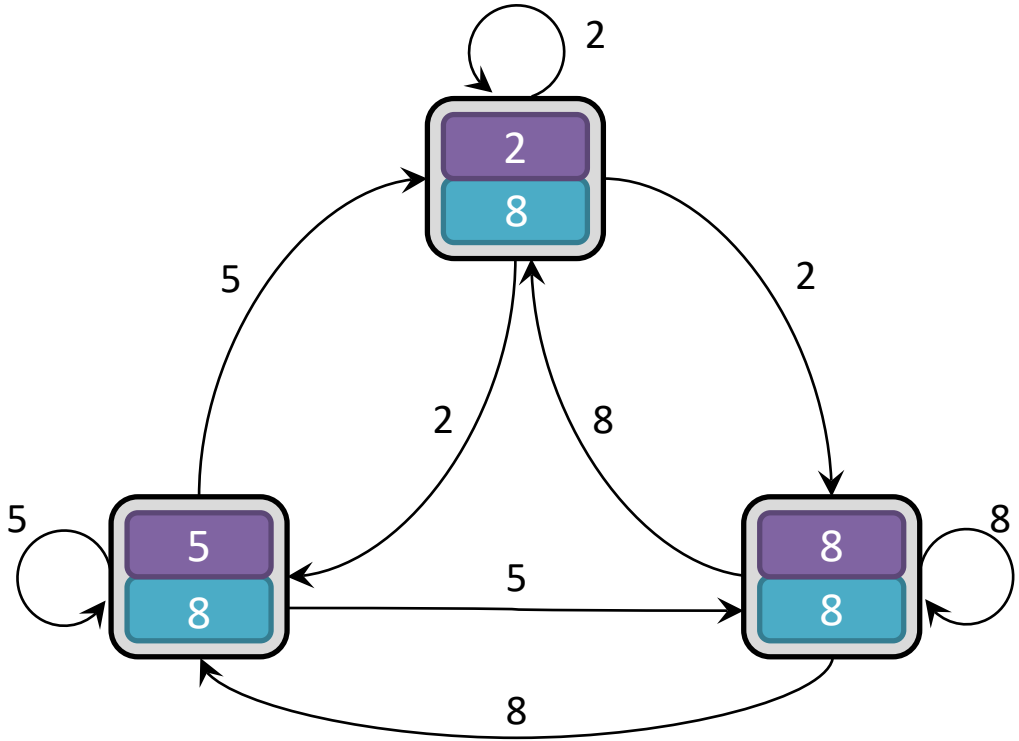
verified low-level implementations

Implementation & Evaluation

challenging case studies (including Paxos)

Broadcast Consensus

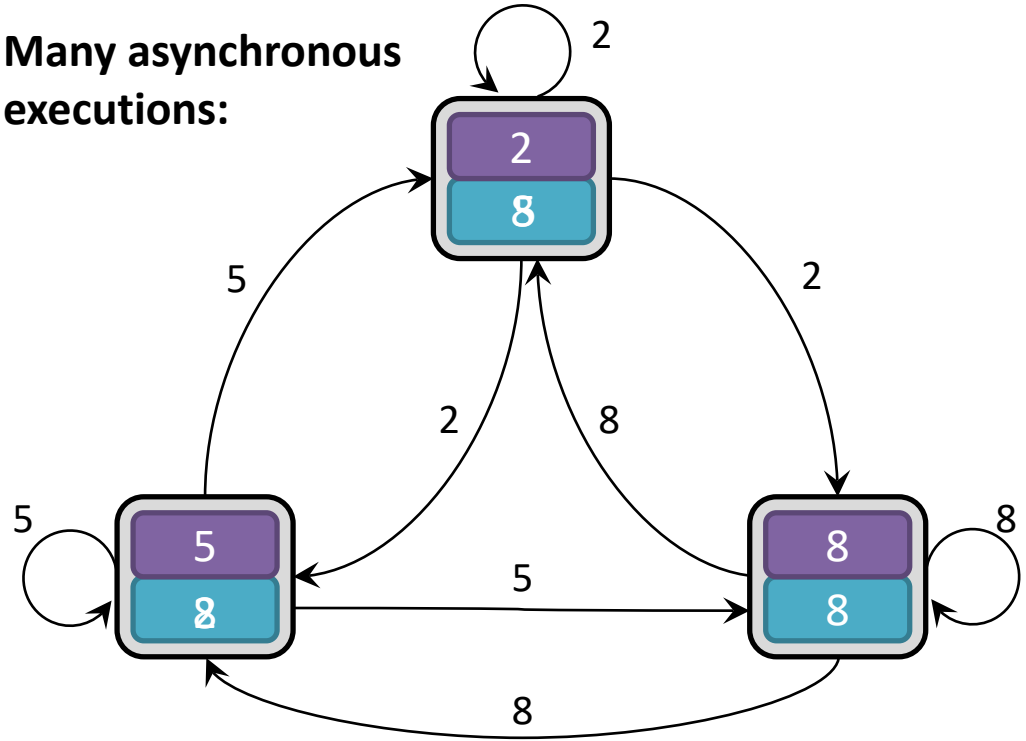
```
proc Main:
  for i in 1..n:
    async Broadcast(i)
    async Collect(i)

proc Broadcast(i):
  for j in 1..n:
    send value[i] CH[j]

proc Collect(i):
  decision[i] := -∞
  for j in 1..n:
    v := receive CH[i]
    if v > decision[i]:
      decision[i] := v
```



Broadcast Consensus

```
proc Main:
  for i in 1..n:
    async Broadcast(i)
    async Collect(i)

proc Broadcast(i):
  for j in 1..n:
    send value[i] CH[j]

proc Collect(i):
  decision[i] := -∞
  for j in 1..n:
    v := receive CH[i]
    if v > decision[i]:
      decision[i] := v
```

Many asynchronous executions:



Background

Mover Types

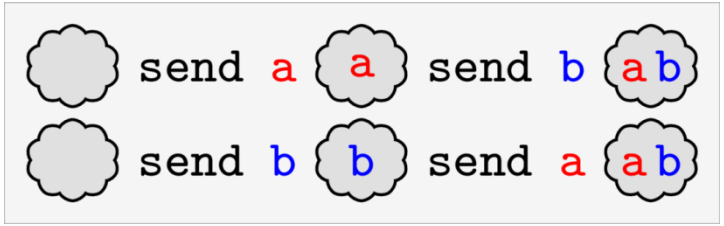
```
proc Main:
  for i in 1..n:
    async Broadcast(i)
    async Collect(i)

proc Broadcast(i):
  for j in 1..n:
    send value[i] CH[j]

proc Collect(i):
  decision[i] := -∞
  for j in 1..n:
    v := receive CH[i]
    if v > decision[i]:
      decision[i] := v
```

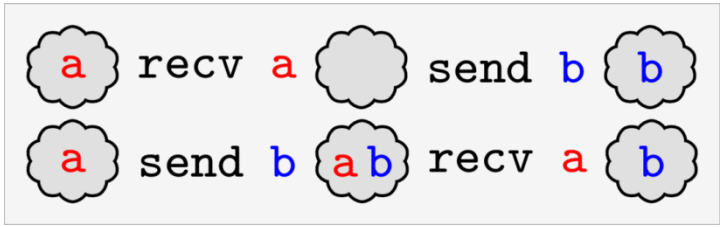
Background

Mover Types

```
proc Main:
  for i in 1..n:
    async Broadcast(i)      L
    async Collect(i)        L

proc Broadcast(i):
  for j in 1..n:
    send value[i] CH[j]     L

proc Collect(i):
  decision[i] := -∞         R
  for j in 1..n:
    v := receive CH[i]      R
    if v > decision[i]:     R
      decision[i] := v      R
```

send is a **left mover**

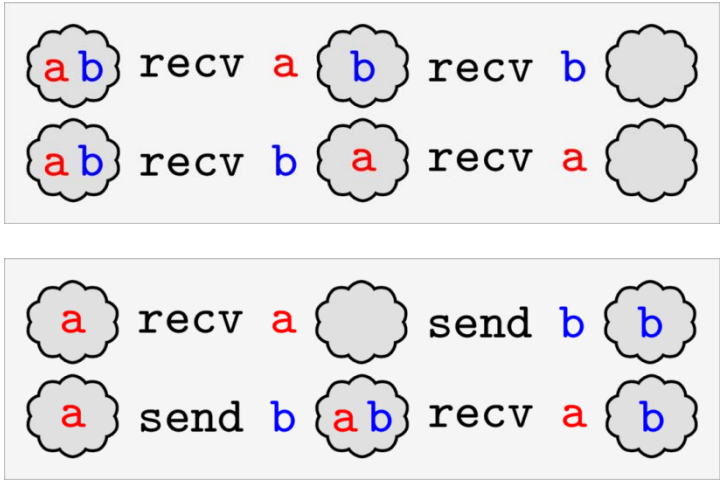




Background

Mover Types

```
proc Main:
  for i in 1..n:
    async Broadcast(i)      L
    async Collect(i)        L

proc Broadcast(i):
  for j in 1..n:
    send value[i] CH[j]     L

proc Collect(i):
  decision[i] := -∞         R
  for j in 1..n:
    v := receive CH[i]      R
    if v > decision[i]:     R
      decision[i] := v      R
```

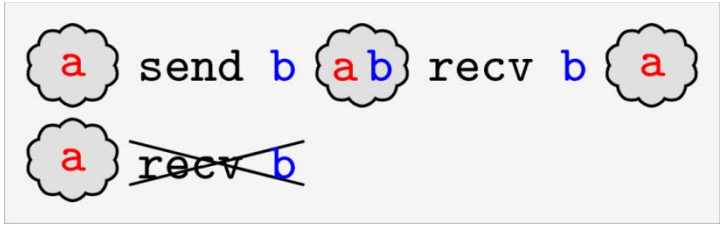


receive is a **right mover**

Background

Mover Types

```
proc Main:
  for i in 1..n:
    async Broadcast(i)    L
    async Collect(i)      L

proc Broadcast(i):
  for j in 1..n:
    send value[i] CH[j]   L

proc Collect(i):
  decision[i] := -∞       R
  for j in 1..n:
    v := receive CH[i]    R
    if v > decision[i]:   R
      decision[i] := v    R
```



Background

Mover Types

```
proc Main:
  for i in 1..n:
    async Broadcast(i)      L
    async Collect(i)        L

proc Broadcast(i):
  for j in 1..n:
    send value[i] CH[j]     L

proc Collect(i):
  decision[i] := -∞         R
  for j in 1..n:
    v := receive CH[i]      R
    if v > decision[i]:     R
      decision[i] := v      R
```

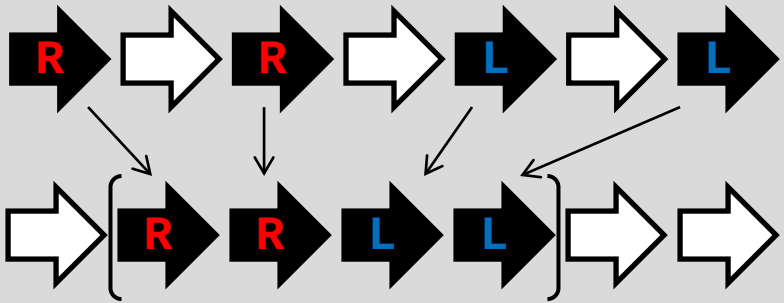
	left mover	right mover
send	✓	
receive		✓
async	✓	
thread-local read/write	✓	✓

Background

Mover Types

```
proc Main:
  for i in 1..n:
    async Broadcast(i)    L
    async Collect(i)      L

proc Broadcast(i):
  for j in 1..n:
    send value[i] CH[j]   L

proc Collect(i):
  decision[i] := -∞       R
  for j in 1..n:
    v := receive CH[i]    R
    if v > decision[i]:   R
      decision[i] := v    R
```

Lipton's Reduction:
Sequences of **R*L*** are atomic.

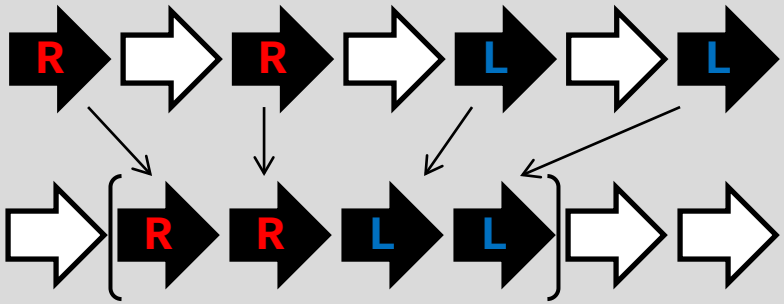


Background

Mover Types

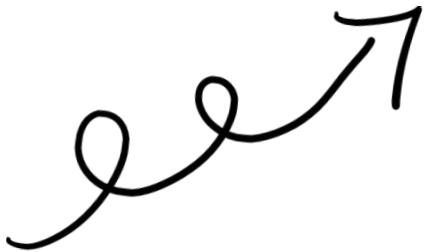
```
action Main:
⎛ for i in 1..n:          ⎞
⎜   async Broadcast(i)    ⎟ L
⎝   async Collect(i)      ⎠ L

action Broadcast(i):
⎛ for j in 1..n:          ⎞
⎝   send value[i] CH[j]   ⎠ L

action Collect(i):
⎛ decision[i] := -∞       ⎞ R
⎜ for j in 1..n:          ⎟
⎜   v := receive CH[i]    ⎟ R
⎜   if v > decision[i]:   ⎟ R
⎝     decision[i] := v    ⎠ R
```

Lipton's Reduction:
Sequences of **R*L*** are atomic.



Inductive Sequentialization

```
action Main:
( for i in 1..n:
    async Broadcast(i)
    async Collect(i) )

action Broadcast(i):                L
( for j in 1..n:
    send value[i] CH[j] )

action Collect(i):                  R
( decision[i] := -∞
  for j in 1..n:
    v := receive CH[i]
    if v > decision[i]:
      decision[i] := v )
```

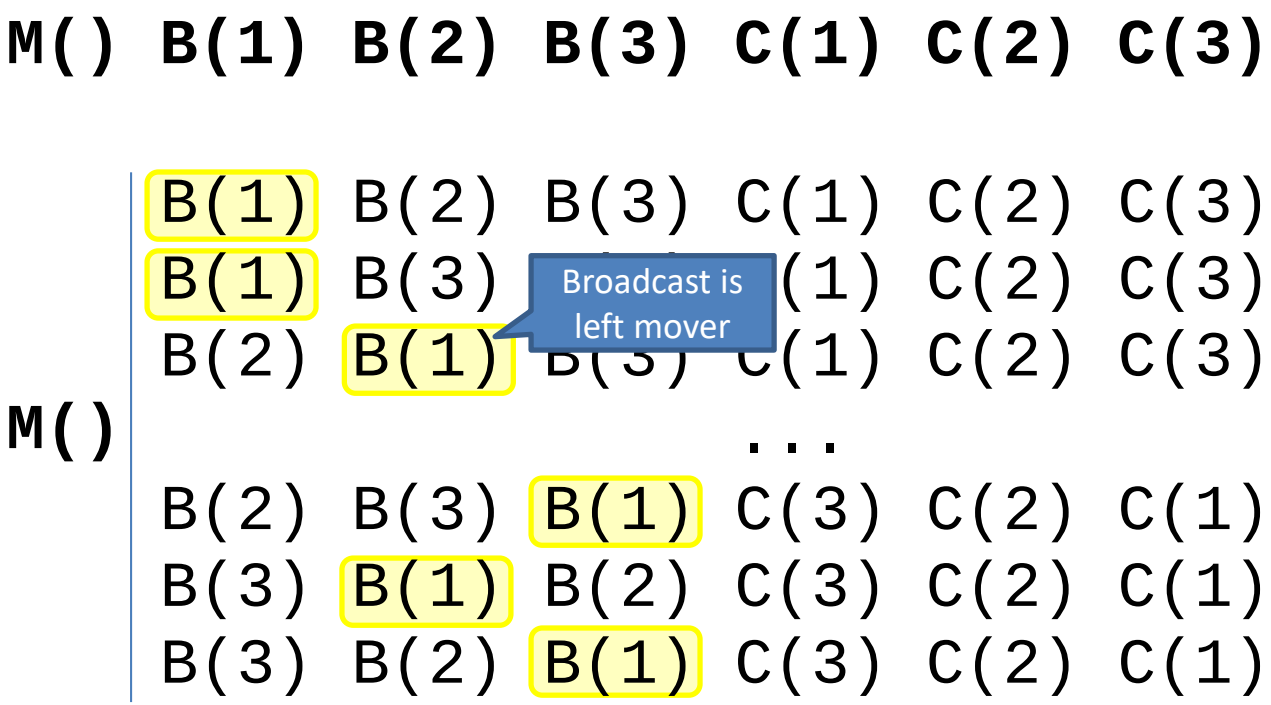


```
action SequentialMain:
  for i in 1..n:
    call Broadcast(i)
  for i in 1..n:
    call Collect(i)
```

Single *sequential* execution:

M() B(1) B(2) B(3) C(1) C(2) C(3)

Sequentialization Idea



Sequentialization Idea

M() B(1) B(2) B(3) C(1) C(2) C(3)

M() B(1)

B(2) B(3) C(1) C(2) C(3)
B(3) B(2) C(1) C(2) C(3)
B(2) B(3) C(1) C(3) C(2)
...
B(3) B(2) C(3) C(1) C(2)
B(2) B(3) C(3) C(2) C(1)
B(3) B(2) C(3) C(2) C(1)

sequential prefix

Sequentialization Idea

M() B(1) B(2) B(3) C(1) C(2) C(3)

M() B(1) B(2)

B(3) C(1) C(2) C(3)
B(3) C(1) C(3) C(2)
B(3) C(2) C(1) C(3)
B(3) C(2) C(3) C(1)
B(3) C(3) C(1) C(2)
B(3) C(3) C(2) C(1)

sequential prefix

Sequentialization Idea

M() B(1) B(2) B(3) C(1) C(2) C(3)

M() B(1) B(2) B(3)

C(1) C(2) C(3)
C(1) C(
C(2) C(1) C(3)
C(2) C(3) C(1)
C(3) C(1) C(2)
C(3) C(2) C(1)

Collect is not left mover

sequential prefix

Sequentialization Idea

M() B(1) B(2) B(3) C(1) C(2) C(3)

Abstraction

```
action CollectAbs(i):
  assert ∀j.Broadcast(j) ∉ Ω
  ...
```

M() B(1) B(2) B(3)

C(1) C(2) C(3)
C(1) C(2) C(3)
C(2) C(1) C(3)
C(2) C(3) C(1)
C(3) C(1) C(2)
C(3) C(2) C(1)

CollectAbs is left mover

sequential prefix

Sequentialization Idea

M() B(1) B(2) B(3) C(1) C(2) C(3)

Abstraction

```
action CollectAbs(i):
  assert ∀j.Broadcast(j) ∉ Ω
  ...
```

M() B(1) B(2) B(3) C(1)

C(2) C(3)
C(3) C(2)

sequential prefix

Sequentialization Idea

M() B(1) B(2) B(3) C(1) C(2) C(3)

Abstraction

```
action CollectAbs(i):
  assert ∀j.Broadcast(j) ∉ Ω
  ...
```

M() B(1) B(2) B(3) C(1) C(2) C(3)

sequential prefix

Sequentialization Idea

M() B(1) B(2) B(3) C(1) C(2) C(3)

M() B(1) B(2) B(3) C(1) C(2) C(3)

complete sequentialization

Inductive Sequentialization

Concurrent program

```
action Main:
  for i in 1..n:
    async Broadcast(i)
    async Collect(i)

action Broadcast(i):
  ...

action Collect(i):
  ...
```

Invariant action

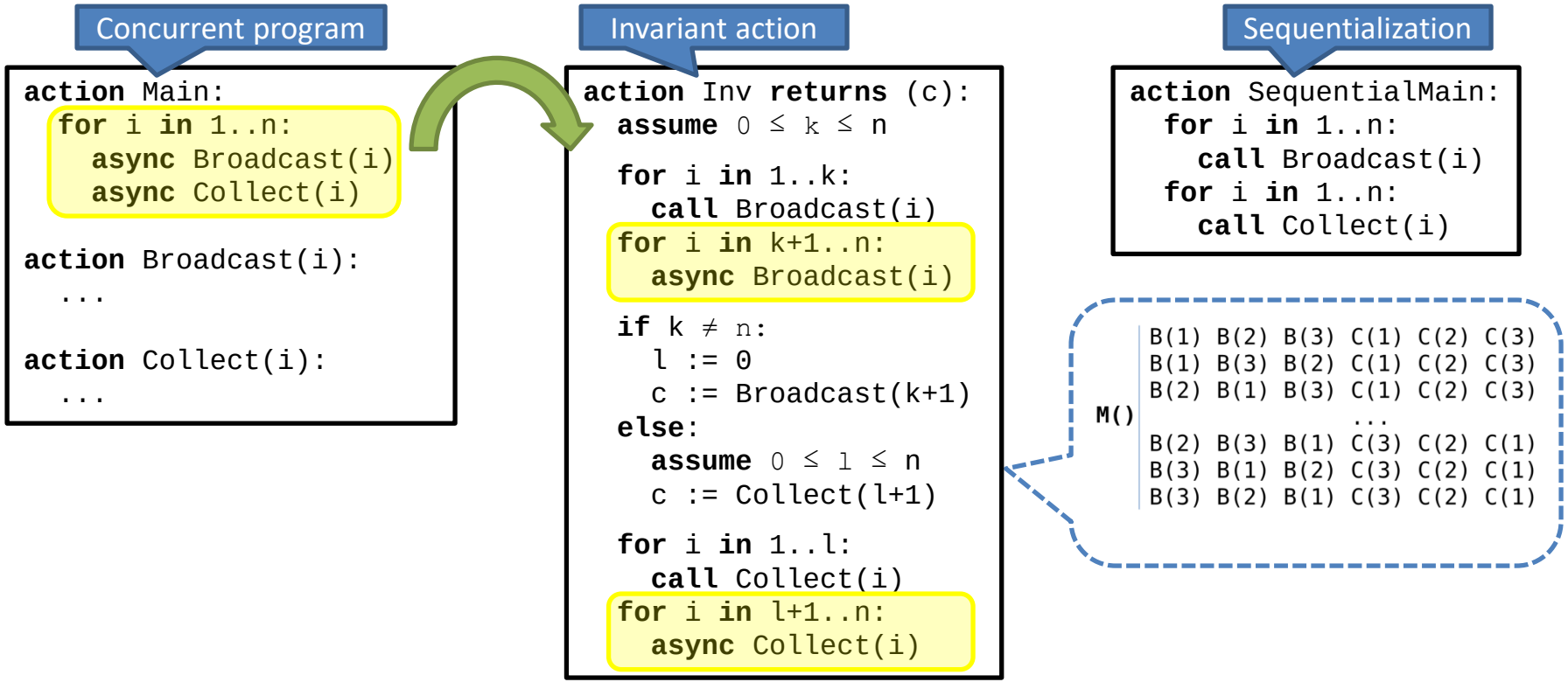
```
action Inv returns (c):
  assume 0 ≤ k ≤ n

  for i in 1..k:
    call Broadcast(i)
  for i in k+1..n:
    async Broadcast(i)

  if k ≠ n:
    l := 0
    c := Broadcast(k+1)
  else:
    assume 0 ≤ l ≤ n
    c := Collect(l+1)

  for i in 1..l:
    call Collect(i)
  for i in l+1..n:
    async Collect(i)
```

Sequentialization

```
action SequentialMain:
  for i in 1..n:
    call Broadcast(i)
  for i in 1..n:
    call Collect(i)
```

Partial sequentializations

Inductive Sequentialization



Inductive Sequentialization

Concurrent program

```
action Main:
  for i in 1..n:
    async Broadcast(i)
    async Collect(i)

action Broadcast(i):
  ...

action Collect(i):
  ...
```

Abstraction

```
action CollectAbs(i):
  assert ∀j.Broadcast(j) ∉ Ω
  ...
```

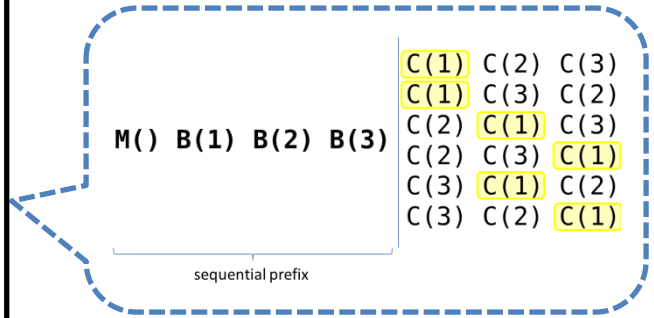
Invariant action

Choice

```
action Inv returns (c):
  assume 0 ≤ k ≤ n

  for i in 1..k:
    call Broadcast(i)
  for i in k+1..n:
    async Broadcast(i)

  if k ≠ n:
    l := 0
    c := Broadcast(k+1)
  else:
    assume 0 ≤ l ≤ n
    c := Collect(l+1)

  for i in 1..l:
    call Collect(i)
  for i in l+1..n:
    async Collect(i)
```

Sequentialization

```
action SequentialMain:
  for i in 1..n:
    call Broadcast(i)
  for i in 1..n:
    call Collect(i)
```



Inductive Sequentialization

Concurrent program

```
action Main:
  for i in 1..n:
    async Broadcast(i)
    async Collect(i)

action Broadcast(i):
  ...

action Collect(i):
  ...
```

Abstraction

```
action CollectAbs(i):
  assert ∀j.Broadcast(j) ∉ Ω
  ...
```

Invariant action

Choice

```
action Inv returns (c):
  assume 0 ≤ k ≤ n
  for i in 1..k:
    call Broadcast(i)
  for i in k+1..n:
    async Broadcast(i)
  if k ≠ n:
    l := 0
    c := Broadcast(k+1)
  else:
    assume 0 ≤ l ≤ n
    c := Collect(l+1)
  for i in 1..l:
    call Collect(i)
  for i in l+1..n:
    async Collect(i)
```

Sequentialization

```
action SequentialMain:
  for i in 1..n:
    call Broadcast(i)
  for i in 1..n:
    call Collect(i)
```

M() B(1) B(2) B(3) C(1) C(2) C(3)

complete sequentialization

Implementation

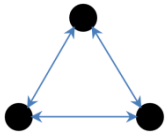
CIVL verifier

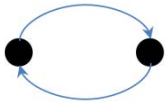
(extension of Boogie)

github.com/boogie-org/boogie

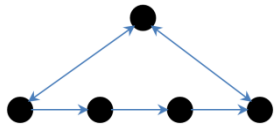
✓ automated ✓ fast ✓ interactive

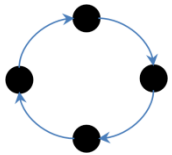
Case Studies

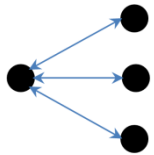

Broadcast consensus

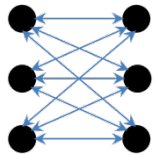

Ping-Pong


Producer-Consumer

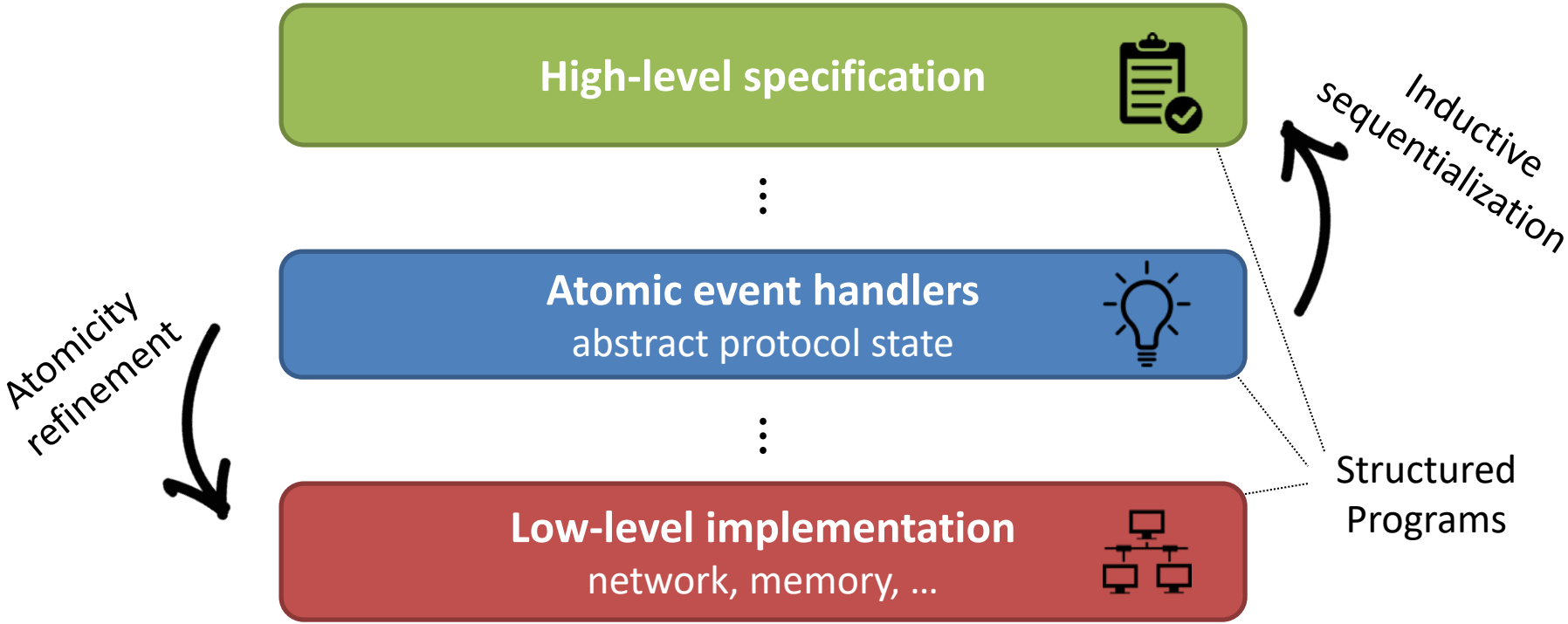

N-Buyer


Chang-Roberts


Two-phase commit


Paxos

The CIVL Methodology



The CIVL Methodology