Synchronizing the Asynchronous

Bernhard Kragl
IST Austria

Shaz Qadeer
Microsoft

Thomas A. Henzinger
IST Austria

Concurrency is Ubiquitous

Asynchronous

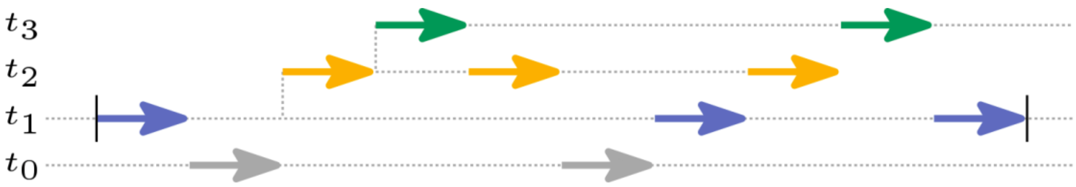
Concurrency is Ubiquitous

Asynchronous programs are hard to specify

assert Pre(Q)
call Q
assume Post(Q)

~~assert Pre(Q)~~
~~**async Q**~~
~~assume Post(Q)~~

Asynchronous programs are hard to verify



Structured program vs. Transition relation

```
            a: x := 0

b: acquire(l)    ||  acquire(l)
c: t1 := x       ||  t2 := x
d: t1 := t1+1    ||  t2 := t2+1
e: x  := t1      ||  x  := t2
f: release(l)    ||  release(l)

         g: assert x = 2
```

Init: $pc = pc_1 = pc_2 = a$

Next:

$pc = a \wedge pc' = pc_1' = pc_2' = b \wedge x' = 0 \wedge eq(l, t_1, t_2)$

$pc_1 = b \wedge pc_1' = c \wedge \neg l \wedge l' \wedge eq(pc, pc_2, x, t_1, t_2)$

$pc_1 = c \wedge pc_1' = d \wedge t_1' = x \wedge eq(pc, pc_2, l, x, t_2)$

$pc_1 = d \wedge pc_1' = e \wedge t_1' = t_1 + 1 \wedge eq(pc, pc_2, l, x, t_2)$

$pc_1 = e \wedge pc_1' = f \wedge x' = t_1 \wedge eq(pc, pc_2, l, t_1, t_2)$

$pc_1 = f \wedge pc_1' = g \wedge \neg l' \wedge eq(pc, pc_2, x, t_1, t_2)$

$pc_2 = b \wedge pc_2' = c \wedge \neg l \wedge l' \wedge eq(pc, pc_1, x, t_1, t_2)$

$pc_2 = c \wedge pc_2' = d \wedge t_2' = x \wedge eq(pc, pc_1, l, x, t_1)$

$pc_2 = d \wedge pc_2' = e \wedge t_2' = t_2 + 1 \wedge eq(pc, pc_1, l, x, t_1)$

$pc_2 = e \wedge pc_2' = f \wedge x' = t_2 \wedge eq(pc, pc_1, l, t_1, t_2)$

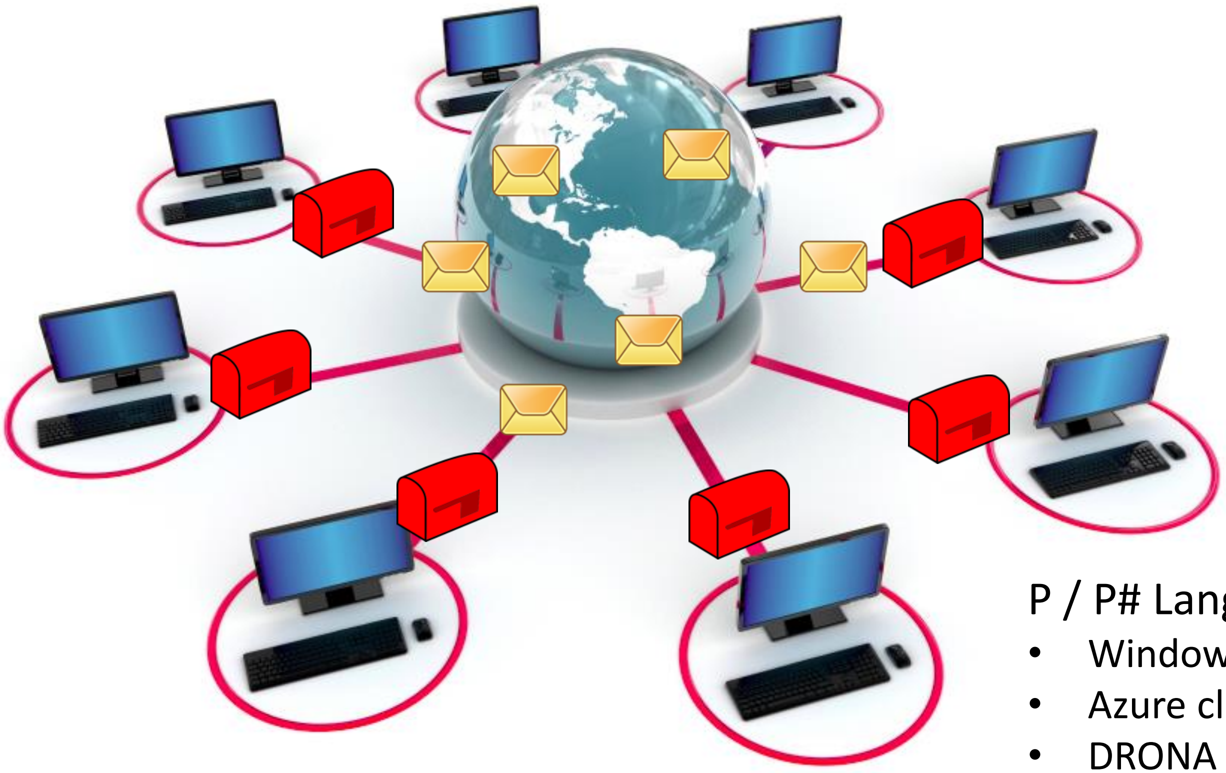
$pc_2 = f \wedge pc_2' = g \wedge \neg l' \wedge eq(pc, pc_1, x, t_1, t_2)$

$pc_1 = pc_2 = g \wedge pc' = g \wedge eq(pc_1, pc_2, l, x, t_1, t_2)$

Safe: $pc = g \Rightarrow x = 2$

Procedures and dynamic thread creation complicate transition relation further!

Shared State in Message-Passing Programs



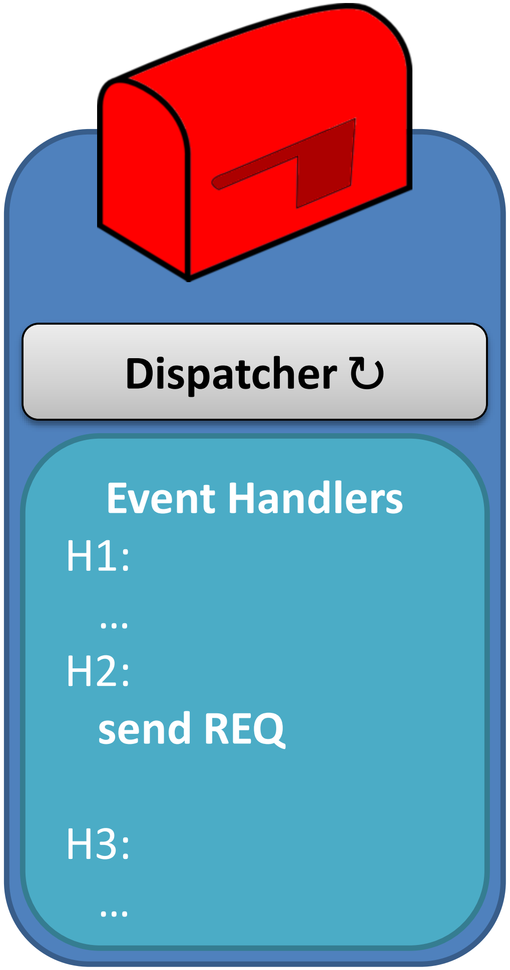
P / P# Language

- Windows 8 USB 3.0 driver
- Azure cloud services
- DRONA framework

Problem: Monolithic proofs do not scale

Question: How can structured proofs help?

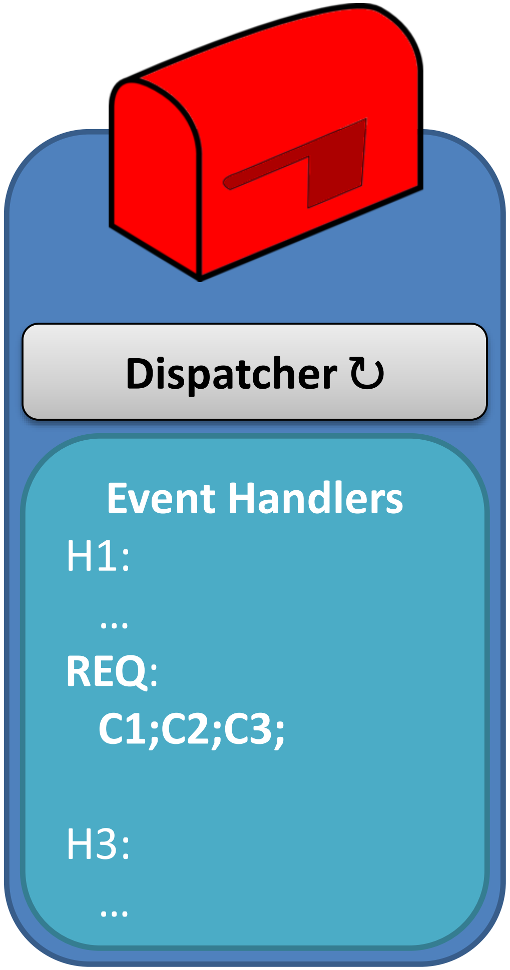
Idea: “Inlining of Event Handlers”



Dispatcher ↻

Event Handlers

```
H1:
  …
H2:
  send REQ

H3:
  …
```

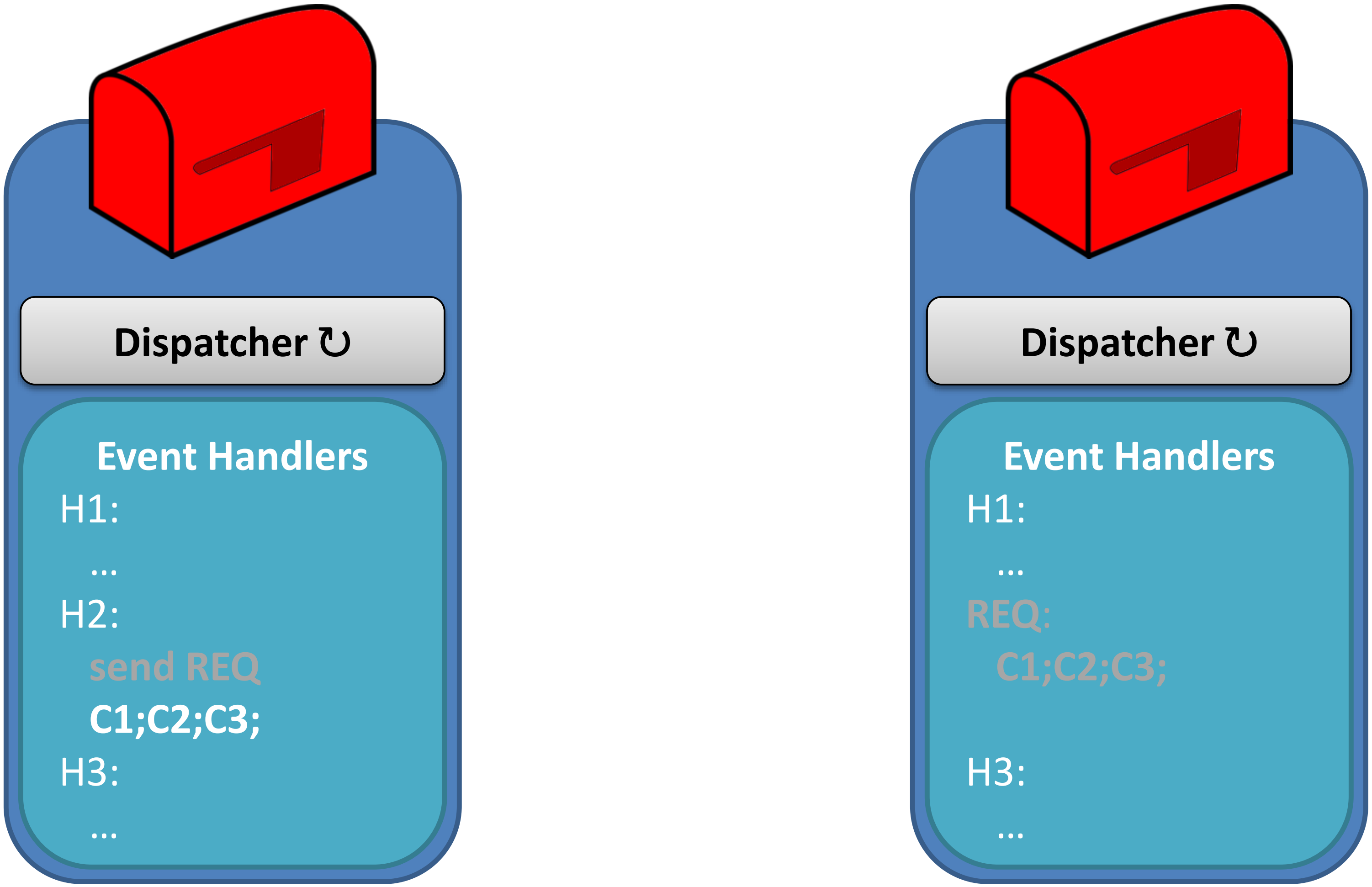


Dispatcher ↻

Event Handlers

```
H1:
  …
REQ:
  C1;C2;C3;

H3:
  …
```

Idea: “Inlining of Event Handlers”



Dispatcher ↻

Event Handlers

```
H1:
  …
H2:
  send REQ
  C1;C2;C3;
H3:
  …
```

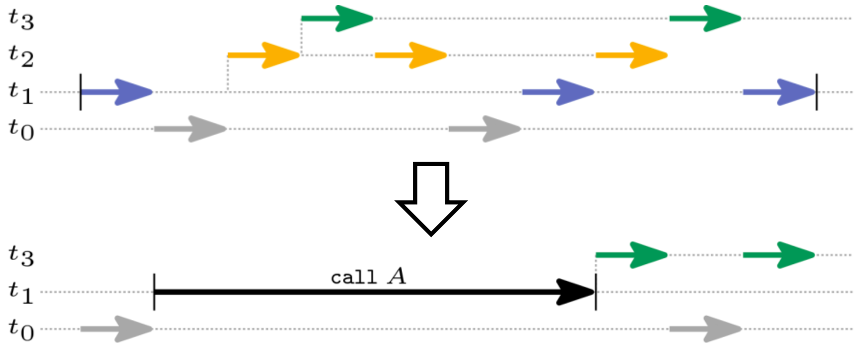
Dispatcher ↻

Event Handlers

```
H1:
  …
REQ:
  C1;C2;C3;

H3:
  …
```

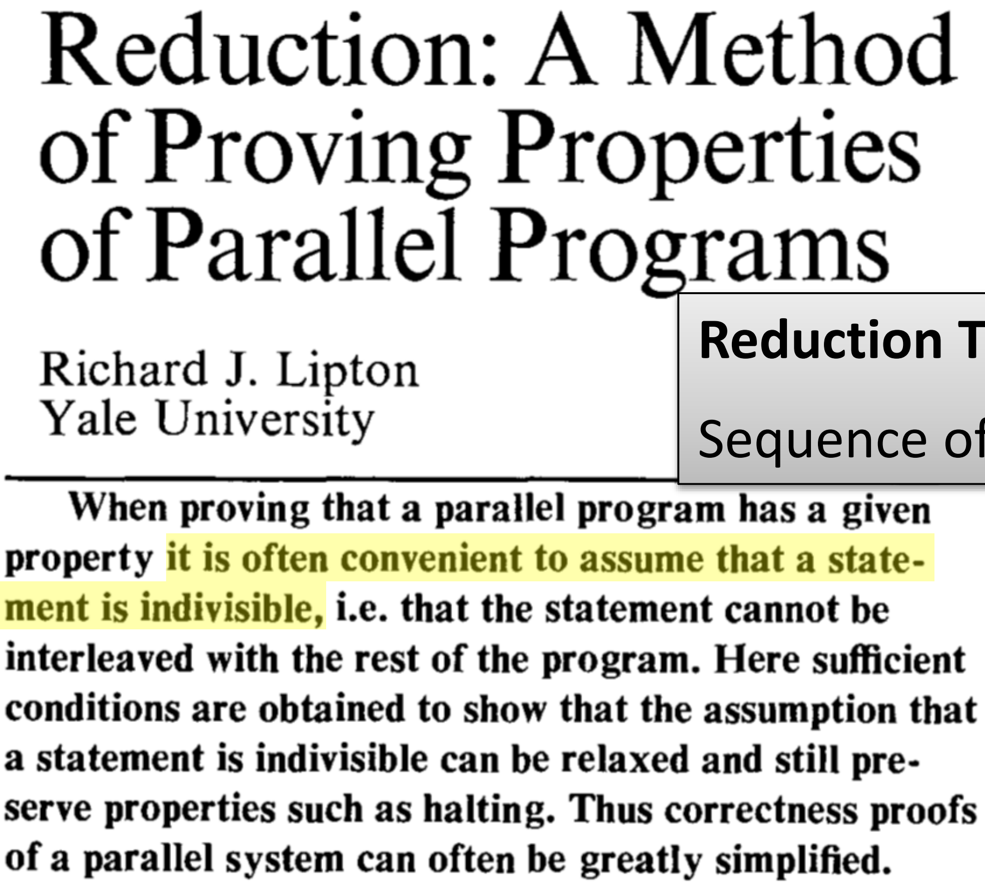
Our Contributions

Synchronization proof rule



Syntax-driven and structured proofs

$$\frac{P_1 \preccurlyeq P_2 \preccurlyeq \cdots \preccurlyeq P_{n-1} \preccurlyeq P_n \qquad P_n \text{ is safe}}{P_1 \text{ is safe}}$$



Reduction: A Method of Proving Properties of Parallel Programs

Richard J. Lipton
Yale University

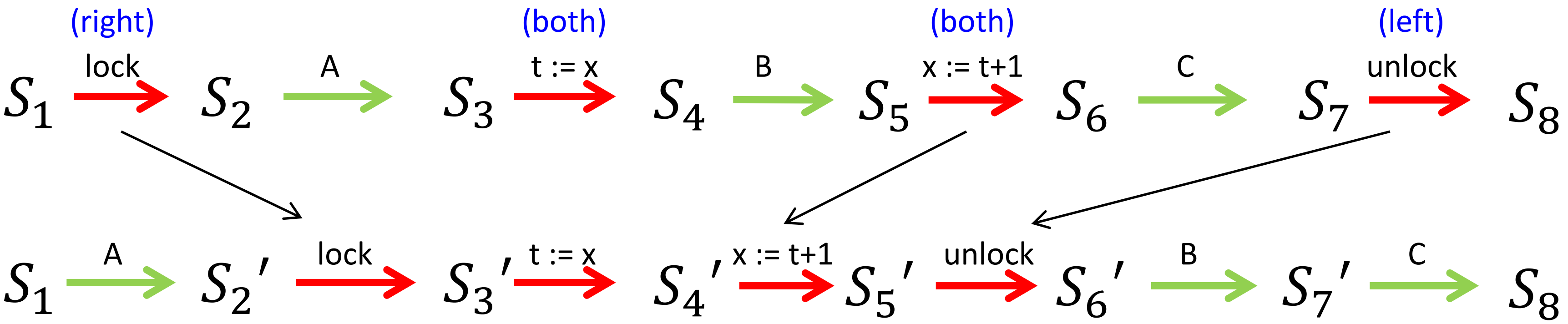
When proving that a parallel program has a given property it is often convenient to assume that a statement is indivisible, i.e. that the statement cannot be interleaved with the rest of the program. Here sufficient conditions are obtained to show that the assumption that a statement is indivisible can be relaxed and still preserve properties such as halting. Thus correctness proofs of a parallel system can often be greatly simplified.

Reduction Theorem

Sequence of (right)*(none)?(left)* is atomic.

Left/right movers

Commutativity



Lifting Reduction to Asynchronous Programs

Let Q be a procedure in program P

- Reduction

atomic action

$$\mathrm{Q} \rightsquigarrow [Q] \rightsquigarrow A$$

- Synchronization

$$Q \rightsquigarrow [sync(Q)] \rightsquigarrow A$$

contains asynchronous invocations

replaces asynchronous invocations with synchronous ones

Synchronization Example

```
global var x

proc Main(n):
  var i := 0
  while i < n:
    async [x := x + 1]
    async [x := x − 1]
    i := i + 1
```

Traces of x: 0 1 2 1 0 -1 -2 -1 0 … 0
0 1 2 3 4 3 2 3 2 … 0
…

⇩

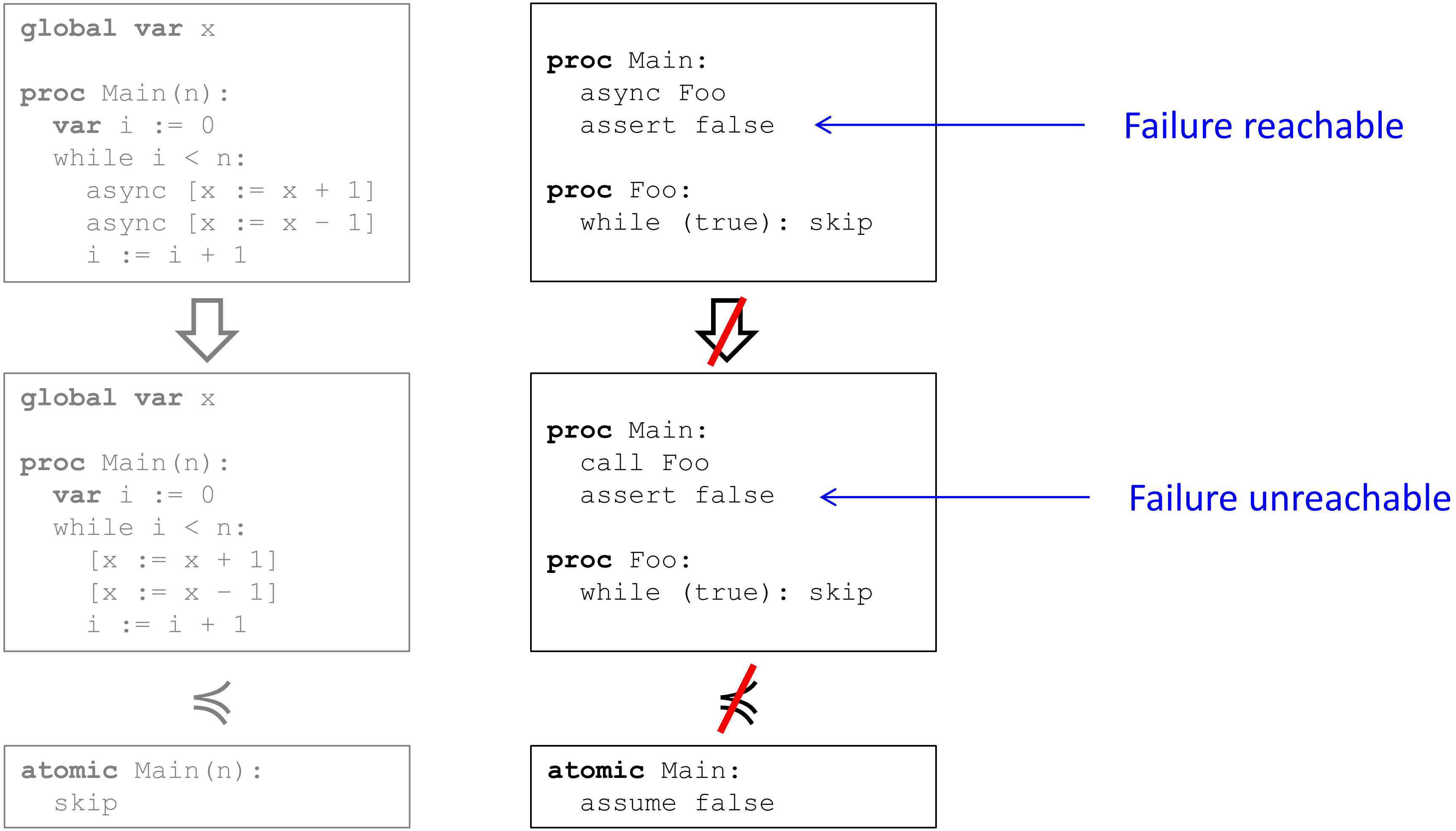
```
global var x

proc Main(n):
  var i := 0
  while i < n:
    [x := x + 1]
    [x := x − 1]
    i := i + 1
```

Trace of x: 0 1 0 1 0 1 0 … 0

≼

```
atomic Main(n):
  skip
```

Termination?



```
global var x

proc Main(n):
  var i := 0
  while i < n:
    async [x := x + 1]
    async [x := x - 1]
    i := i + 1
```

⇩

```
global var x

proc Main(n):
  var i := 0
  while i < n:
    [x := x + 1]
    [x := x - 1]
    i := i + 1
```

≼

```
atomic Main(n):
  skip
```

```
proc Main:
  async Foo
  assert false

proc Foo:
  while (true): skip
```

Failure reachable

$\not\Downarrow$

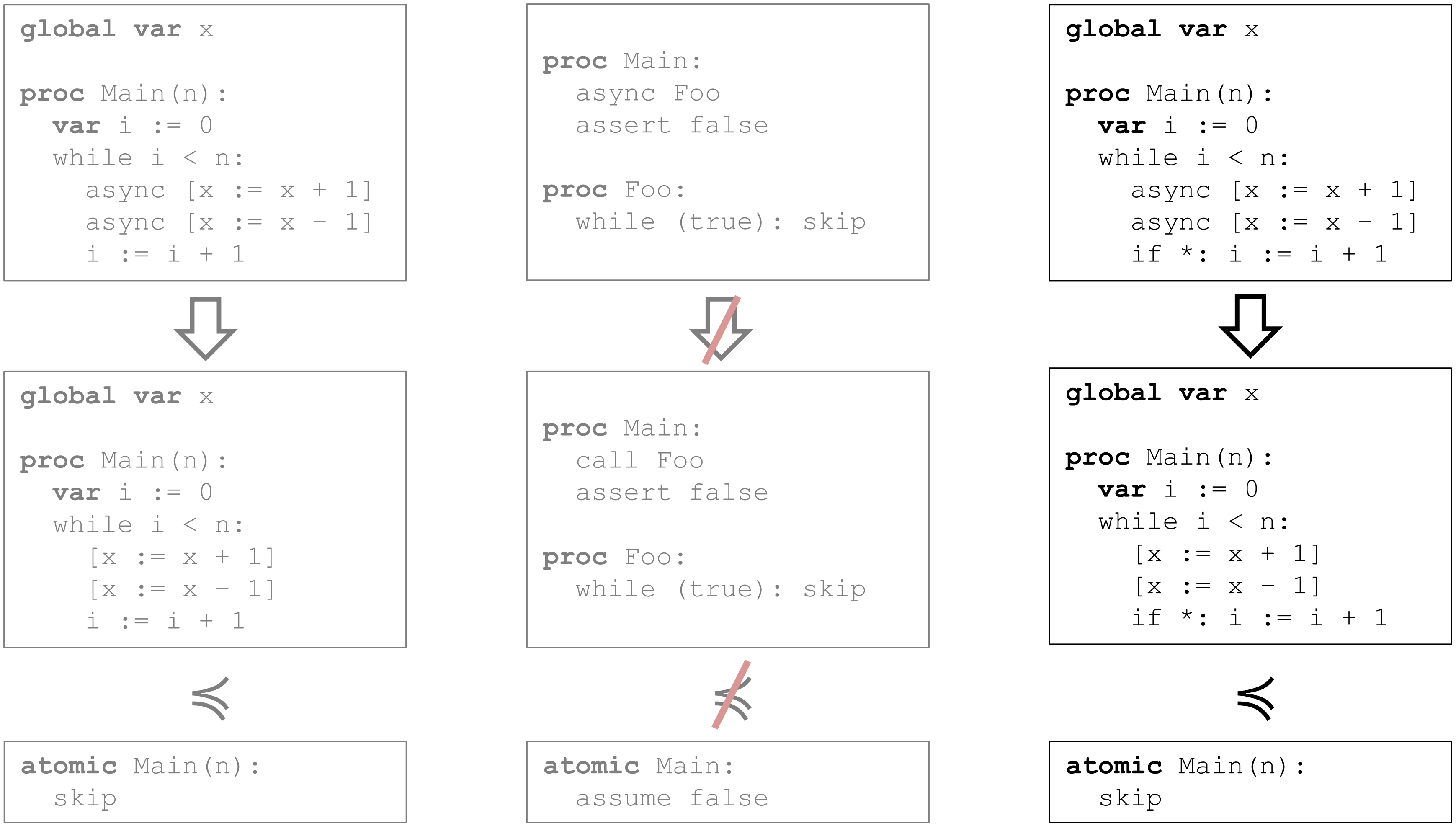
```
proc Main:
  call Foo
  assert false

proc Foo:
  while (true): skip
```

Failure unreachable

⋠

```
atomic Main:
  assume false
```

~~Termination?~~ Cooperation!



```
global var x

proc Main(n):
  var i := 0
  while i < n:
    async [x := x + 1]
    async [x := x - 1]
    i := i + 1
```

$\Downarrow$

```
global var x

proc Main(n):
  var i := 0
  while i < n:
    [x := x + 1]
    [x := x - 1]
    i := i + 1
```

$\preccurlyeq$

```
atomic Main(n):
  skip
```

```
proc Main:
  async Foo
  assert false

proc Foo:
  while (true): skip
```

$\cancel{\Downarrow}$

```
proc Main:
  call Foo
  assert false

proc Foo:
  while (true): skip
```

$\cancel{\preccurlyeq}$

```
atomic Main:
  assume false
```

```
global var x

proc Main(n):
  var i := 0
  while i < n:
    async [x := x + 1]
    async [x := x - 1]
    if *: i := i + 1
```

$\Downarrow$

```
global var x

proc Main(n):
  var i := 0
  while i < n:
    [x := x + 1]
    [x := x - 1]
    if *: i := i + 1
```

$\preccurlyeq$

```
atomic Main(n):
  skip
```

Pending Asynchronous Calls

$$Q \rightsquigarrow [sync(Q)] \rightsquigarrow A$$

contains asynchronous invocations

replaces asynchronous invocations with synchronous ones

Example: Lock Service

```
global var lock : nat?

proc Acquire(tid : nat)
  s := false
  while (!s)
    call s := CAS(lock,NIL,tid)
  async Callback(tid)
```

Pending Asynchronous Calls

$$Q \rightsquigarrow [sync(Q)] \rightsquigarrow A$$

contains asynchronous invocations

replaces **SOME** asynchronous invocations with synchronous ones

contains pending asyncs

Example: Lock Service

```
global var lock : nat?

proc Acquire(tid : nat)
  s := false
  while (!s)
    call s := CAS(lock,NIL,tid)
  async Callback(tid)
```

⇨

```
global var lock : nat?

atomic ACQUIRE(tid : nat)
  assume lock == NIL
  lock := tid
  async Callback(tid)
```

Example: Lock Service

Server

```
proc Acquire(tid: nat)
  s := false
  while (!s)
    call s := CAS(lock,NIL,tid)
  async Callback(tid)

proc Release(tid: nat)
  lock := nil
```

By synchronization

```
atomic ACQUIRE(tid: nat)
  assume lock == NIL
  lock := tid
  async Callback(tid)

left RELEASE(tid: nat)
  assert lock == tid
  lock := nil
```

Client

```
proc Callback(tid: nat)
  t := x
  x := t + 1
  async Release(tid)
```

By synchronization

```
left CALLBACK(tid: nat)
  assert lock == tid
  x := x + 1
  lock := nil
```

By async elimination

```
atomic ACQUIRE'(tid: nat)
  assume lock == NIL
  lock := tid
  x := x + 1
  lock := nil
```

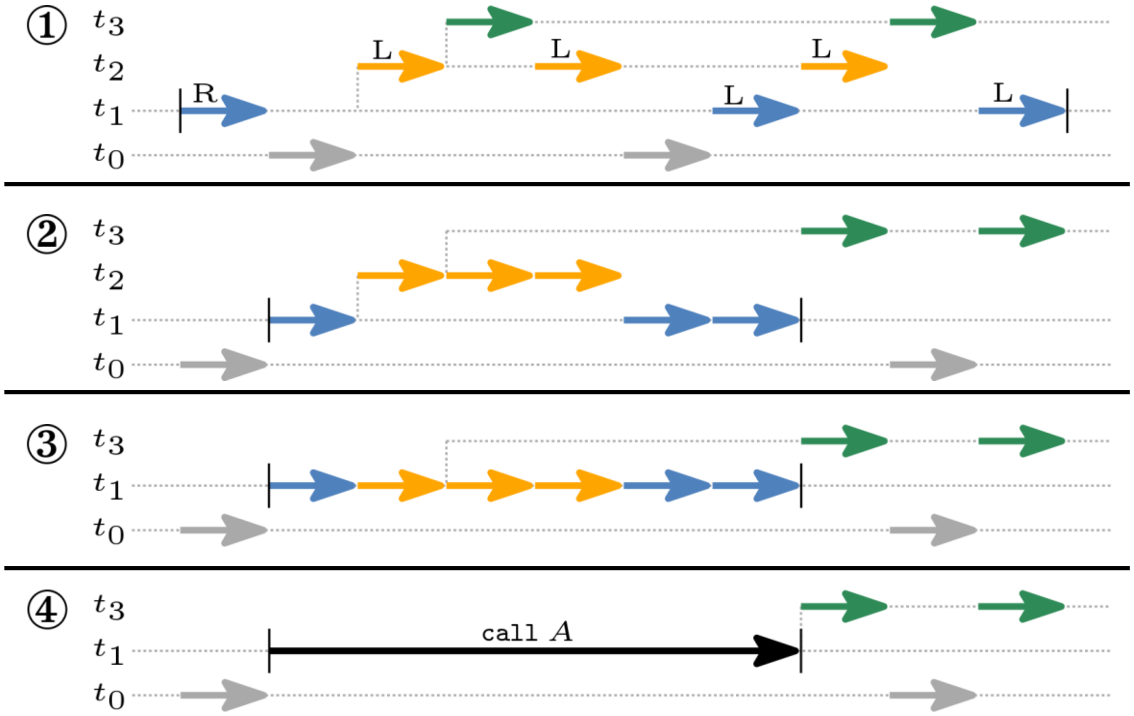
By abstraction

```
atomic ACQUIRE''(tid: nat)
  x := x + 1
```

Synchronizing Asynchrony I

Synchronization transforms procedure Q into atomic action A

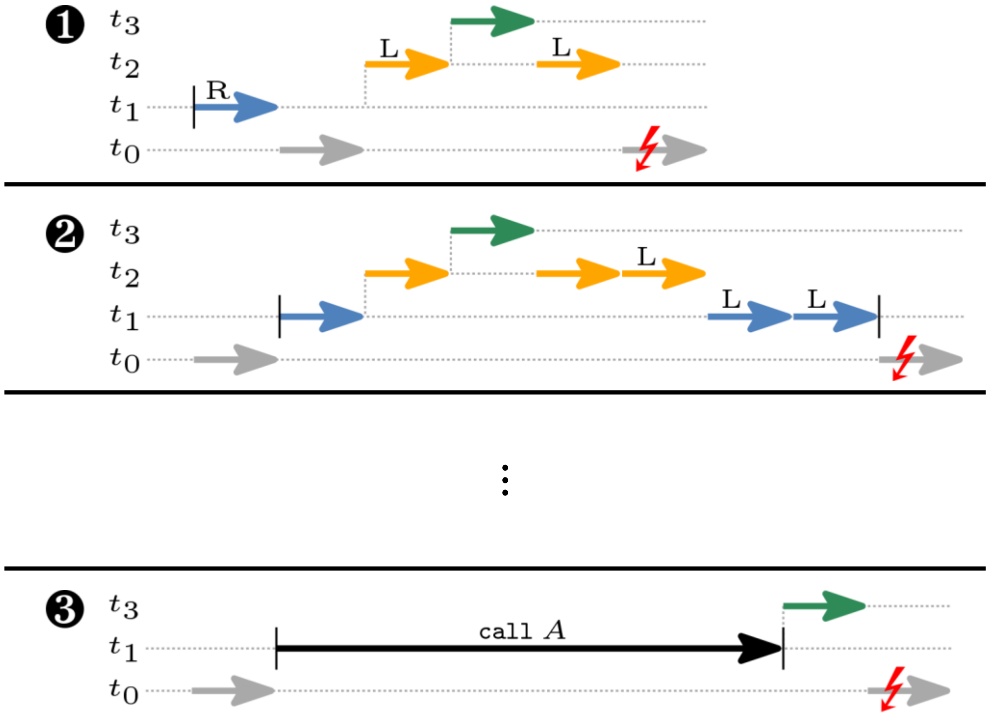
Atomicity : (1) execution steps of Q match (right)*(none)?(left)*
(2) execution steps in asynchronous threads of Q match (left)*



Synchronizing Asynchrony II

Synchronization transforms procedure Q into atomic action A

Cooperation: partial sequential executions of Q must have some terminating extension



Multi-layered Refinement Proofs

$$\frac{P_1 \preccurlyeq P_2 \preccurlyeq \cdots \preccurlyeq P_{n-1} \preccurlyeq P_n \qquad P_n \text{ is safe}}{P_1 \text{ is safe}}$$

Advantages of structured proofs:

Better for humans: easier to construct and maintain

Better for computers: localized/small checks → easier to automate

Layered Programs [Hawblitzl, Petrank, Qadeer, Tasiran 2015] [K, Qadeer 2018]

Express $P_1, \cdots, P_n$ (and their connection) as single entity

Lock Service (Layered Program)

```
// Global variables

var lock@[1,3] : nat?
var x    @[1,4] : int
```

```
// Client

left CALLBACK@[3,3](tid: nat)
   assert lock == tid
   x := x + 1
   lock := NIL

proc Callback@2(tid: nat)
refines CALLBACK
   var t: int
   call t := READ(tid)
   call WRITE(tid, t+1)
   async Release(tid)
```

```
// Server

atomic ACQUIRE@[2,3](tid: nat)
   assume lock == NIL
   lock := tid
   async Callback(tid)


left RELEASE@[2,2](tid: nat)
   assert lock == tid
   lock := NIL

proc Acquire@1(tid: nat)
refines ACQUIRE
   var s: bool
   s := false
   while (!s) call s := CAS(NIL, tid)
   async Callback(tid)

proc Release@1(tid: nat)
refines RELEASE
   call RESET()
```

```
// Primitive atomic actions

atomic CAS@[1,1](old, new: nat?)
returns (s: bool)
   if (lock == old)
      lock := new
      s := true
   else
      s := false

atomic RESET@[1,1]()
   lock := NIL

both READ@[1,2](tid: nat)
returns (v: int)
   assert lock == tid
   v := x

both WRITE@[1,2](tid: nat, v: int)
   assert lock == tid
   x := v
```

Lock Service (Layer 1)

```
// Global variables

var lock@[1,3] : nat?
var x     @[1,4] : int
```

```
// Client

left CALLBACK@[3,3](tid: nat)
   assert lock == tid
   x := x + 1
   lock := NIL

proc Callback@2(tid: nat)
refines CALLBACK
   var t: int
   call t := READ(tid)
   call WRITE(tid, t+1)
   async Release(tid)
```

```
// Server

atomic ACQUIRE@[2,3](tid: nat)
   assume lock == NIL
   lock := tid
   async Callback(tid)


left RELEASE@[2,2](tid: nat)
   assert lock == tid
   lock := NIL

proc Acquire@1(tid: nat)
refines ACQUIRE
   var s: bool
   s := false
   while (!s) call s := CAS(NIL, tid)
   async Callback(tid)

proc Release@1(tid: nat)
refines RELEASE
   call RESET()
```

```
// Primitive atomic actions

atomic CAS@[1,1](old, new: nat?)
returns (s: bool)
   if (lock == old)
      lock := new
      s := true
   else
      s := false

atomic RESET@[1,1]()
   lock := NIL

both READ@[1,2](tid: nat)
returns (v: int)
   assert lock == tid
   v := x

both WRITE@[1,2](tid: nat, v: int)
   assert lock == tid
   x := v
```

Lock Service (Layer 2)

```
// Global variables

var lock@[1,3] : nat?
var x    @[1,4] : int
```

```
// Client

left CALLBACK@[3,3](tid: nat)
   assert lock == tid
   x := x + 1
   lock := NIL

proc Callback@2(tid: nat)
refines CALLBACK
   var t: int
   call t := READ(tid)
   call WRITE(tid, t+1)
   async RELEASE(tid)
```

```
// Server

atomic ACQUIRE@[2,3](tid: nat)
   assume lock == NIL
   lock := tid
   async Callback(tid)


left RELEASE@[2,2](tid: nat)
   assert lock == tid
   lock := NIL

proc Acquire@1(tid: nat)
refines ACQUIRE
   var s: bool
   s := false
   while (!s) call s := CAS(NIL, tid)
   async Callback(tid)

proc Release@1(tid: nat)
refines RELEASE
   call RESET()
```

```
// Primitive atomic actions

atomic CAS@[1,1](old, new: nat?)
returns (s: bool)
   if (lock == old)
      lock := new
      s := true
   else
      s := false

atomic RESET@[1,1]()
   lock := NIL

both READ@[1,2](tid: nat)
returns (v: int)
   assert lock == tid
   v := x

both WRITE@[1,2](tid: nat, v: int)
   assert lock == tid
   x := v
```

Lock Service (Layer 3)

```
// Global variables

var lock@[1,3] : nat?
var x    @[1,4] : int
```

```
// Client

left CALLBACK@[3,3](tid: nat)
   assert lock == tid
   x := x + 1
   lock := NIL

proc Callback@2(tid: nat)
refines CALLBACK
   var t: int
   call t := READ(tid)
   call WRITE(tid, t+1)
   async Release(tid)
```

```
// Server

atomic ACQUIRE@[2,3](tid: nat)
   assume lock == NIL
   lock := tid
   x := x + 1
   lock := NIL

left RELEASE@[2,2](tid: nat)
   assert lock == tid
   lock := NIL

proc Acquire@1(tid: nat)
refines ACQUIRE
   var s: bool
   s := false
   while (!s) call s := CAS(NIL, tid)
   async Callback(tid)

proc Release@1(tid: nat)
refines RELEASE
   call RESET()
```

```
// Primitive atomic actions

atomic CAS@[1,1](old, new: nat?)
returns (s: bool)
   if (lock == old)
      lock := new
      s := true
   else
      s := false

atomic RESET@[1,1]()
   lock := NIL

both READ@[1,2](tid: nat)
returns (v: int)
   assert lock == tid
   v := x

both WRITE@[1,2](tid: nat, v: int)
   assert lock == tid
   x := v
```

CIVL (Boogie Extension)

github.com/boogie-org/boogie

rise4fun.com/civl

Programmer Input

- Layer annotations
- Atomic action specs
- Mover types
- Supporting invariants

CIVL

- Commutativity checking
- *Atomicity checking*
- Refinement checking
- *Cooperation checking*

Details in the paper!

Case studies:

- Lock service
- Two-phase commit (2PC) protocol
- Task distribution service

Shared memory:

- Concurrent garbage collector [Hawblitzel et. al; CAV’15]
- FastTrack2 race-detection algorithm [Flanagan, Freund, Wilcox; PPoPP’18]
- Weak memory (TSO) programs [Bouajjani, Enea, Mutluergil, Tasiran; CAV’18]

Conclusion

- Synchronization Proof Rule
  - Coarse-grained atomic action from (potentially unbounded) asynchronous computations
  - Pending asynchronous calls

- Multi-layered Refinement
  - Structured Proofs
  - Simpler Invariants