Proving a Concurrent Program Correct by Demonstrating It Does Nothing

Bernhard Kragl
IST Austria

Shaz Qadeer
Microsoft

doi: 10.1007/978-3-319-96145-3_5

https://github.com/boogie-org/boogie

https://www.rise4fun.com/civl

Credits

Cormac Flanagan

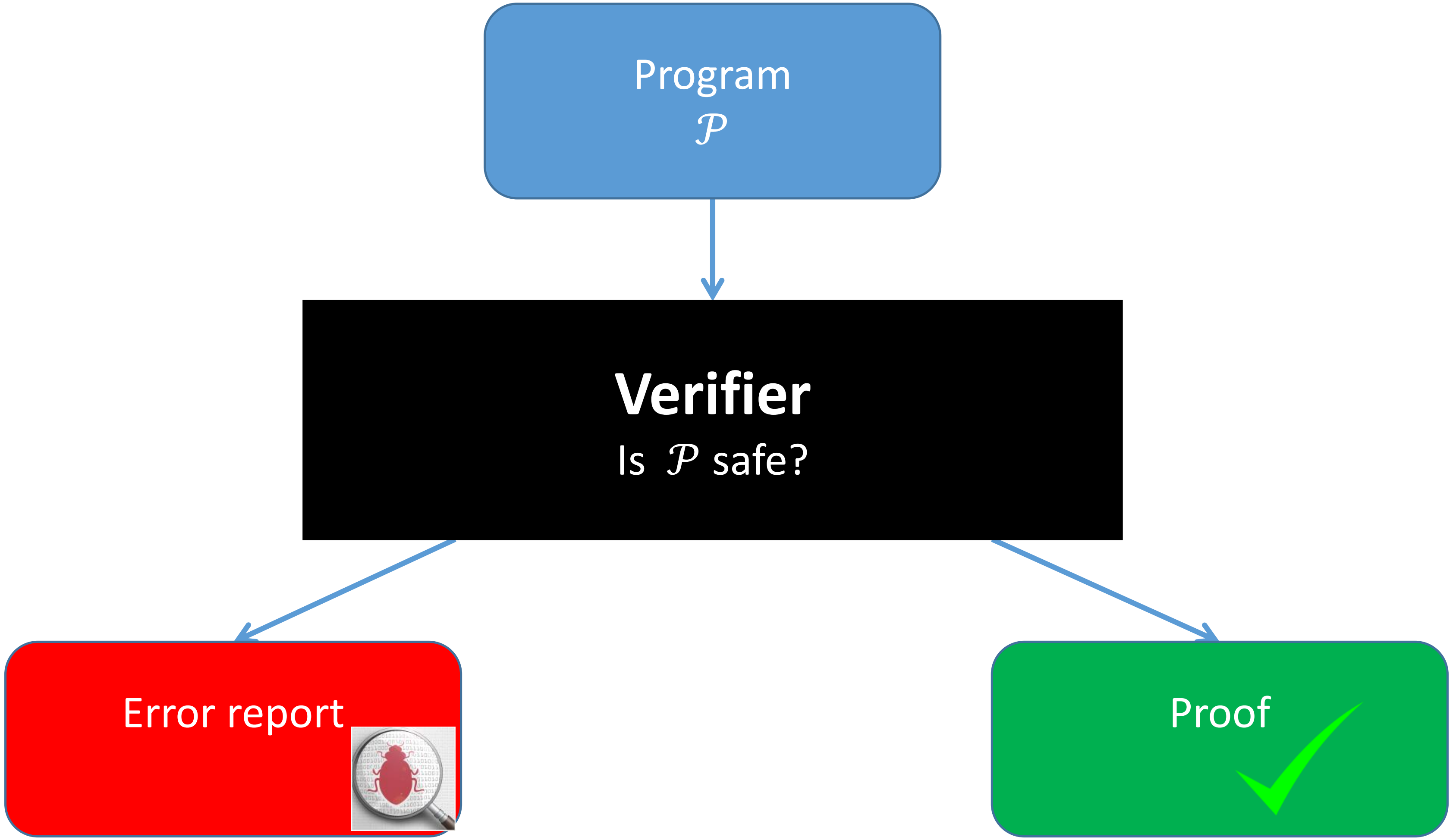
Stephen N. Freund

Serdar Tasiran

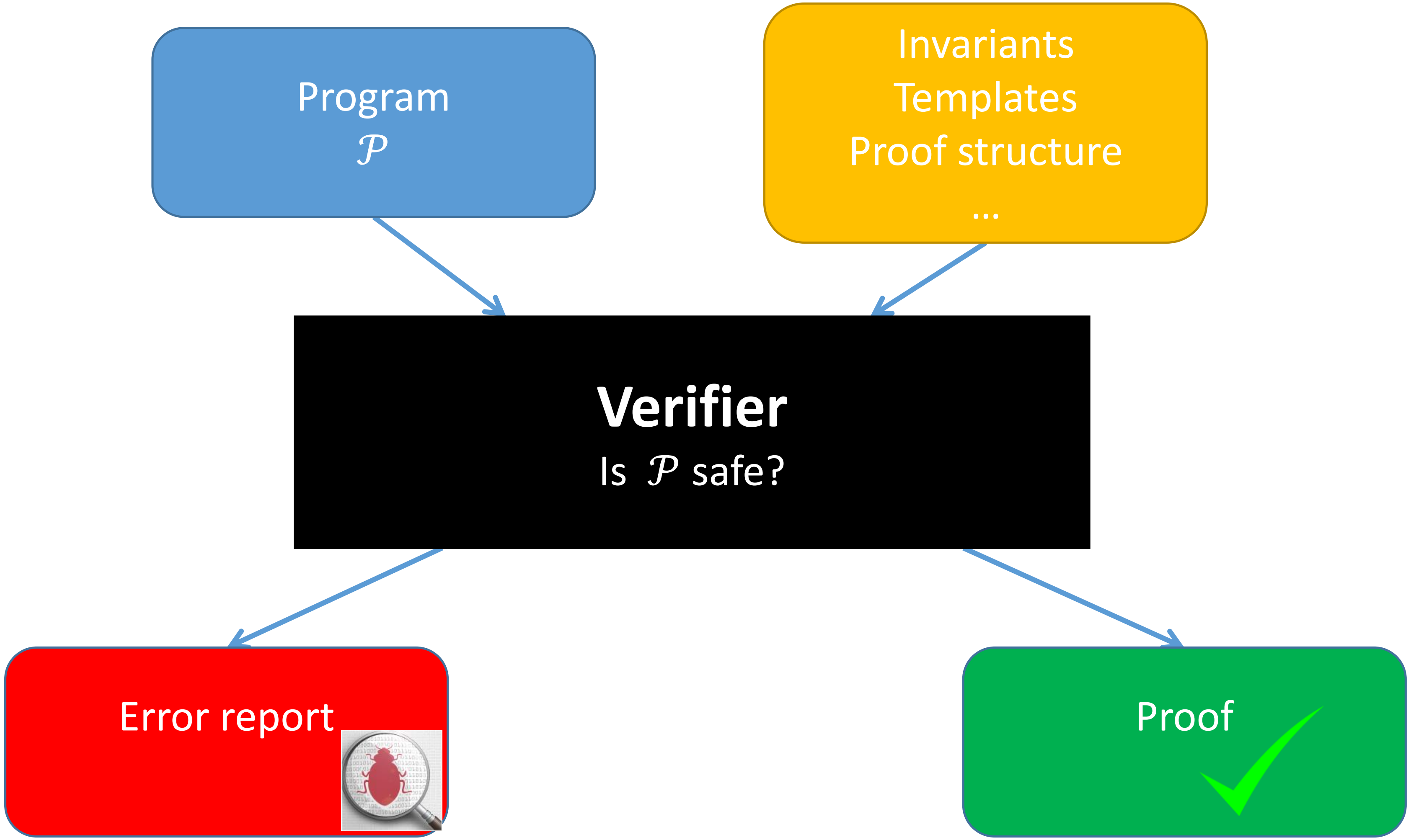
Tayfun Elmas

Chris Hawblitzel

Program verification



Program verification



Reasoning about transition systems

- Transition system $(Var, Init, Next, Safe)$

  Var (Variables)
  $Init$ (Initial state predicate over Var)
  $Next$ (Transition predicate over $Var \cup Var'$)
  $Safe$ (Safety predicate over Var)

- Inductive invariant Inv

  $Init \Rightarrow Inv$ (Initialization)
  $Inv \land Next \Rightarrow Inv'$ (Preservation)
  $Inv \Rightarrow Safe$ (Safety)

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
```

Procedures and dynamic thread creation complicate transition relation further!

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: $pc = g \Rightarrow x = 2$

Interference freedom and Owicki-Gries

$$\frac{\Psi_1 : \{P_1\}\ C_1\ \{Q_1\} \qquad \Psi_2 : \{P_2\}\ C_2\ \{Q_2\} \qquad \Psi_1, \Psi_2 \text{ interference free}}{\{P_1 \wedge P_2\}\ C_1 \parallel C_2\ \{Q_1 \wedge Q_2\}}$$

- Example: $\{x = 0\}\ x := x + 1 \parallel x := x + 2\ \{x = 3\}$

$\{x = 0\}$

$\{x = 0\}$ P_1: $\{x = 0 \vee x = 2\}$ $x := x + 1$ Q_1: $\{x = 1 \vee x = 3\}$	$\parallel$	$\{x = 0\}$ P_2: $\{x = 0 \vee x = 1\}$ $x := x + 2$ Q_2: $\{x = 2 \vee x = 3\}$

$\{Q_1 \wedge Q_2\}$
$\{x = 3\}$

Interference freedom:

$\{P_1 \wedge P_2\}\ x := x + 2\ \{P_1\}$ $\qquad$ $\{P_2 \wedge P_1\}\ x := x + 1\ \{P_2\}$

$\{Q_1 \wedge P_2\}\ x := x + 2\ \{Q_1\}$ $\qquad$ $\{Q_2 \wedge P_1\}\ x := x + 1\ \{Q_2\}$

Ghost variables

Need to refer to other thread's state

- local variables
- program counter

- Example: $\{x = 0\}\ x := x + 1 \parallel x := x + 1\ \{x = 2\}$

$$\{x = 0\}$$
$$[done_1 := false; done_2 := false]$$

$$P_1: \{\neg done_1 \land (\neg done_2 \Rightarrow x = 0) \land (done_2 \Rightarrow x = 1)\}$$
$$[x := x + 1;\ done_1 := true]$$
$$Q_1: \{done_1 \land (\neg done_2 \Rightarrow x = 1) \land (done_2 \Rightarrow x = 2)\}$$

$$\parallel$$

$$P_2: \{\neg done_2 \land (\neg done_1 \Rightarrow x = 0) \land (done_1 \Rightarrow x = 1)\}$$
$$[x := x + 1;\ done_2 := true]$$
$$Q_2: \{done_2 \land (\neg done_1 \Rightarrow x = 1) \land (done_1 \Rightarrow x = 2)\}$$

$$\{x = 2\}$$

Rely/Guarantee

Rely/Guarantee specifications $C \vDash (P, R, G, Q)$ for individual threads
and composition rule
allow for modular proofs of loosely-coupled systems.

$$\{x \geq 0\}$$
$$x \coloneqq x + 1 \parallel x \coloneqq x + 1 \qquad P = Q = (x \geq 0) \quad R = G = (x' \geq x)$$
$$\{x \geq 0\}$$

Multi-layered refinement proofs

$$\frac{P_1 \preccurlyeq P_2 \preccurlyeq \cdots \preccurlyeq P_{n-1} \preccurlyeq \overset{\text{[skip]}}{\overset{||}{P_n}} \qquad P_n \text{ is safe}}{P_1 \text{ is safe}}$$

Advantages of structured proofs:

Better for humans: easier to construct and maintain

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

Programs that do nothing cannot go wrong

Refinement is well-studied

- Logic
  - $P(x, x') \Rightarrow Q(x, x')$
- Labeled transition systems
  - Language containment
  - Simulation (forward, backward, upward, downward, diagonal, sideways, …)
  - Bisimulation (vanilla, mint, lavender, barbed, triangulated, complicated, …)
  - …

Refinement is difficult for programs

- Programs are complicated
  - Complex control and data
- Gap between program syntax and abstractions
- … especially for concurrent programs
- … especially for interactive proof construction

CIVL: Construct correct concurrent programs layer by layer

- Operates on program syntax
- Organizes proof as a sequence of program layers with increasingly coarse-grained atomic actions
- All layers and supporting invariants expressed together in one textual unit
- Automatically-generated verification conditions

```
procedure P(…) { S }

S1; S2

if (e) S1 else S2

while (e) S

call P

async call P

call P1 || P2

call A
```

Gated atomic actions [Elmas, Q, Tasiran 2009]

(Gate, Transition)

single-state predicate — two-state predicate

Command	**Gate**	**Transition**
x := x+y	$true$	$x' = x + y \wedge y' = y$
havoc x	$true$	$y' = y$
assert x<y	$x < y$	$x' = x \wedge y' = y$
assume x<y	$true$	$x < y \wedge x' = x \wedge y' = y$

Lock specification

var lock : ThreadID ∪ {nil}
Acquire(): [assume lock == nil; lock := tid]
Release(): [assert lock == tid; lock := nil]

- Unifies precondition and postcondition
- Primitive for modeling a (concrete or abstract) concurrent program

Operational semantics

- Program configuration $\left(g, \left\{(\ell, s) \cdot \vec{f}\right\} \uplus \mathcal{T}\right)$
- Transition relation $\Rightarrow$ between configurations (and failure configuration $\bot$)
- Safety: $\neg\exists g\ell\colon \big(g, (\ell, Main)\big) \Rightarrow^{*} \bot$

- $Good(P) = \left\{ g \mid \neg\exists\ell\colon \big(g, (\ell, Main)\big) \Rightarrow^{*} \bot \right\}$
- $Trans(P) = \left\{ (g, g') \mid \exists\ell\colon \big(g, (\ell, Main)\big) \Rightarrow^{*} (g', \emptyset) \right\}$

- $P_1 \preccurlyeq P_2$: (1) $Good(P_2) \subseteq Good(P_1)$ (2) $Good(P_2) \circ Trans(P_1) \subseteq Trans(P_2)$

P_2 preserves failures

P_2 preserves final states

```
var x;

IncrBy2() =
   [ x := x + 2 ]

call IncrBy2()
```

```
var x;

Incr() =
   [ x := x + 1 ]

Op() {
   call Incr()
}

call Op() || Op()
```

```
var lock;
var x;

Acquire() =
   [ assume lock == nil;
     lock := tid; ]

Release() =
   [ assert lock == tid;
     lock := nil; ]

Op() {
   var t;
   call Acquire();
   t := x;
   x := t + 1;
   call Release();
}

call Op() || Op()
```

```
var b;
var x;

Acquire() {
   while (true)
      if (CAS(b, 0, 1)) break;
}

Release() {
   b := 0;
}

Op() {
   var t;
   call Acquire();
   t := x;
   x := t + 1;
   call Release();
}

call Op() || Op()
```

```
const c >= 0;
var x;

call Main();
Main() {
    // Create c threads
    // each executing Incr
}
Incr() {
    acquire();
    assert x ≥ 0;
    x := x + 1;
    release();
}
```

➔

[assert x ≥ 0]

```
const c >= 0;
var x;

call Main();
Main() {
   x := 0;
   // Create c threads
   // each executing Incr
}
Incr() {
   acquire();
   assert x ≥ 0;
   x := x + 1;
   release();
}
```

➔ [ ]

Programs constructed with CIVL

- Concurrent garbage collector [Hawblitzel, Petrank, Q, Tasiran 2015]
- FastTrack2 race-detection algorithm [Flanagan, Freund, Wilcox 2018]
- Lock-protected memory atop TSO [Hawblitzel]
- Thread-local heap atop shared heap [Hawblitzel, Q]
- Two-phase commit [K, Q, Henzinger 2018]
- Work-stealing queue, Treiber stack, Ticket, …

Program layers in CIVL

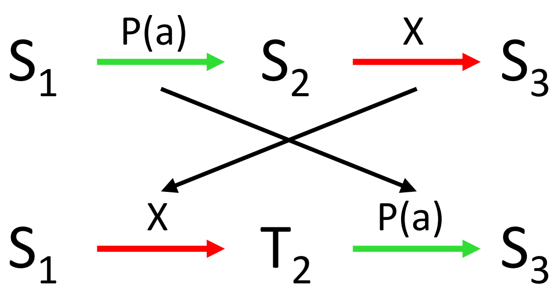
- A CIVL program denotes a sequence of concurrent programs (layers)
  - chained together by a refinement-preserving transformation

- Transformation between program layers combines
  - Atomization: Transform statement S into atomic block [S]
  - Summarization: Transform atomic block [S] into atomic action A
  - Abstraction: Replace atomic action A with atomic action B

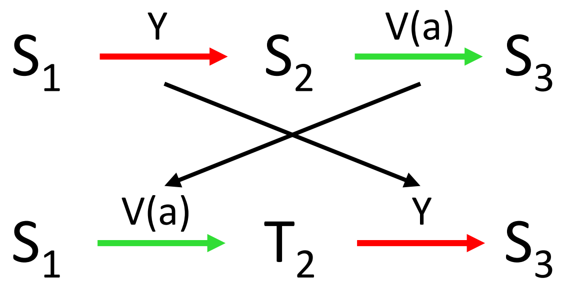
Right and left movers [Lipton 1975]

Integer a   “Semaphore”

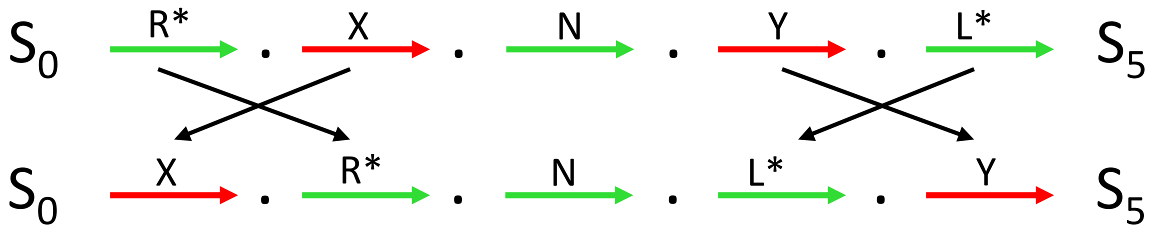
“wait”
P(a) = [assume a > 0; a := a - 1]
right mover (R)



“signal”
V(a) = [a := a + 1]
left mover (L)



Sequence R*;(N+ε); L* is atomic



Atomic actions can fail

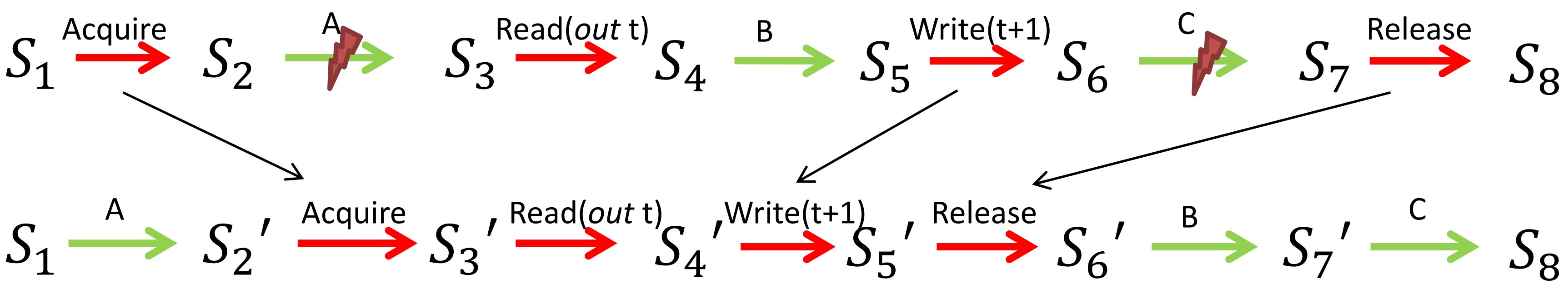
var x : int, lock : ThreadID ∪ {nil}

Acquire(): [assume lock == nil; lock := tid]

Release(): [assert lock == tid; lock := nil]

Read(*out* r): [assert lock == tid; r := x]

Write(v): [assert lock == tid; x := v]



Commutativity: R X → X R X L → L X

Forward preservation: R X⊥ → X⊥ X L⊥ → X⊥

Backward preservation: X⊥ → L X⊥

Nonblocking and Cooperation

[x := 0] || [x := x + 1] ^{N}
[assert x = 0] || [assume false] ~~^{L}~~

[x := 0] || [x := x + 1;
[assert x = 0] || assume false]

Left movers must be nonblocking

[x := 0] || [x := x + 1] ^{N}
[assert x = 0] || while (true) [skip] ^{L}

[x := 0] || [x := x + 1;
[assert x = 0] || while (true) skip]

Termination? Too strong.

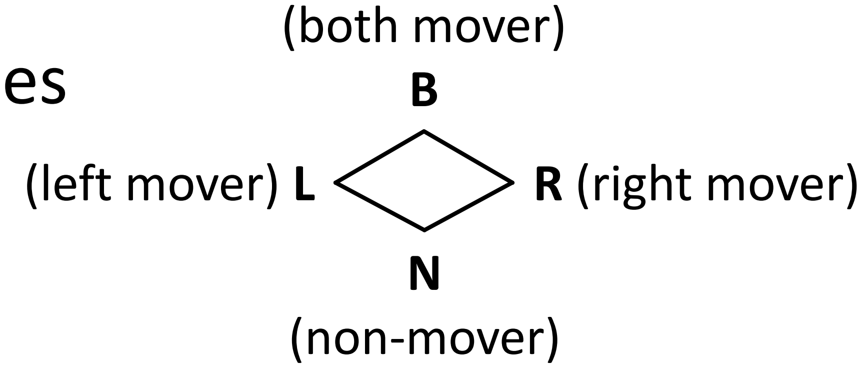
[x := 0] || [x := x + 1] ^{N}
[assert x = 0] || while (*) [skip] ^{L}

[x := 0] || [x := x + 1;
[assert x = 0] || while (*) skip]

Cooperation: always possible to terminate

Mover types in CIVL

1. Atomic actions are annotated with mover types



2. Induced logical commutativity conditions

Commutativity (G_1, T_1) is R or (G_2, T_2) is L

$\forall S_1 S_2 S_3 \exists S_2' : G_1(S_1) \wedge G_2(S_1) \wedge T_1(S_1, S_2) \wedge T_2(S_2, S_3) \Rightarrow T_2(S_1, S_2') \wedge T_1(S_2', S_3)$

Nonblocking (G, T) is L

$\forall S \exists S' : G(S) \Rightarrow T(S, S')$

Forward preservation (G_1, T_1) is R or (G_2, T_2) is L

$\forall S S' : G_1(S) \wedge G_2(S) \wedge T_1(S, S') \Rightarrow G_2(S')$

Backward preservation (G_2, T_2) is L

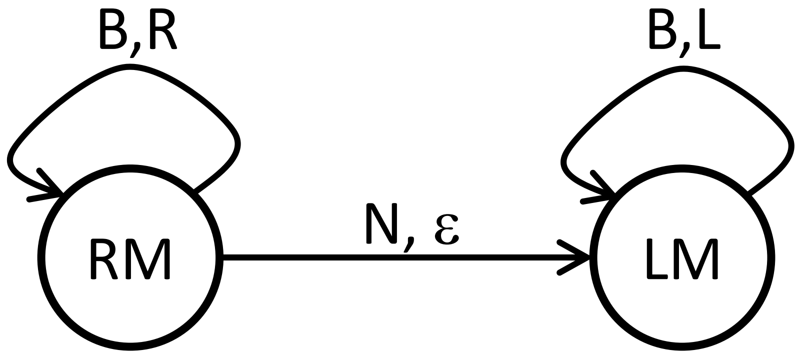
$\forall S S' : G_2(S) \wedge T_2(S, S') \wedge G_1(S') \Rightarrow G_1(S)$

3. Atomization of composite statements

Atomization (S → [S])

- We justified rearranging executions to create “atomic transactions”
- Goal: statically create atomic blocks with only rearrangeable executions

- Each path in S behaves like the automaton
  - Type system [Flanagan, Q 2003]
  - Simulation relation on labeled graphs [Hawblitzel, Petrank, Q, Tasiran 2015]

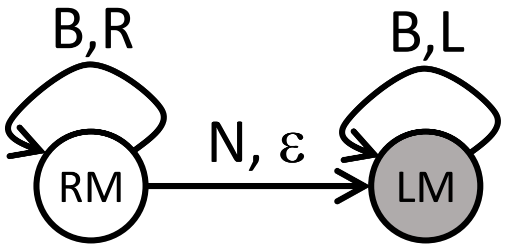


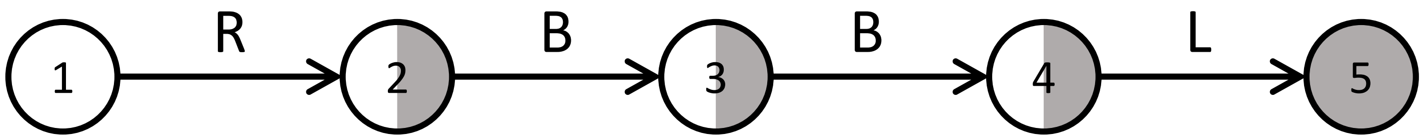
Example: Atomizing nonatomic increment

var x : int, lock : ThreadID ∪ {nil}

Acquire():	[assume lock == nil; lock := tid]	R
Release():	[assert lock == tid; lock := nil]	L
Read(*out* r):	[assert lock == tid; r := x]	B
Write(v):	[assert lock == tid; x := v]	B

```
proc Inc ()
    var t
    Acquire()
    Read(out t)
    Write(t + 1)
    Release()
```





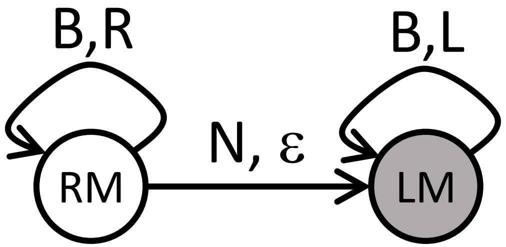
Simulation computation [Henzinger, Henzinger, Kopke 1995]

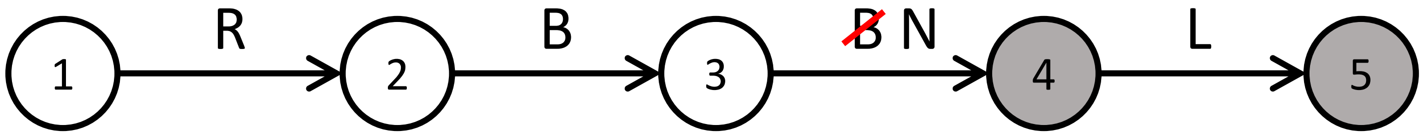
Example: Atomizing nonatomic increment

var x : int, lock : ThreadID ∪ {nil}

Acquire():	[assume lock == nil; lock := tid]	R
Release():	[assert lock == tid; lock := nil]	L
Read(*out* r):	[assert lock == tid; r := x]	B
Write(v):	[assert lock == tid; x := v]	~~B~~ N
Read2(*out* t):	[t := x]	N

```
proc Inc ()
  var t
  Acquire()
  Read(out t)
  Write(t + 1)
  Release()
```





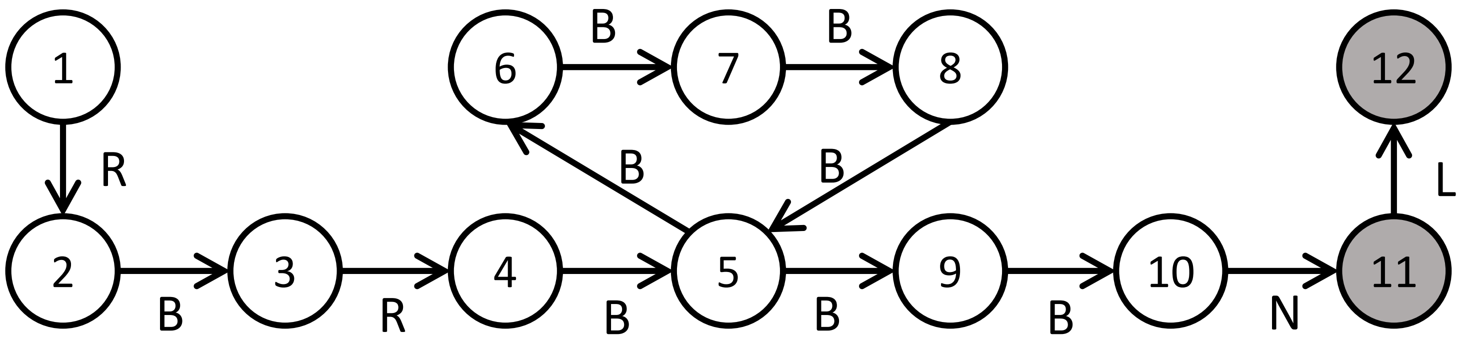
Simulation computation [Henzinger, Henzinger, Kopke 1995]

Example: Resizing an array

```
proc DoubleSize ()
    var t, B, v
    Acquire()
    GetLen(out t)
    B := Allocate(2*t)
    i := 0
    while (i < t)
        Read(i, out v)
        B[i] := v
        i := i + 1
    Switch(B)
    SetLen(2*t)
    Release()
```

var A : Array, len : Nat, lock : ThreadID ∪ {nil}

Acquire():	[assume lock == nil; lock := tid]	R
Release():	[assert lock == tid; lock := nil]	L
GetLen(*out* r):	[assert lock == tid; r := len]	B
GetLen2(*out* r):	[r := len]	N
SetLen (v):	[assert lock == tid; len := v]	N
Read(i, out r):	[assert lock == tid; r := A[i]]	B
Switch(B):	[assert lock == tid; A := B]	B

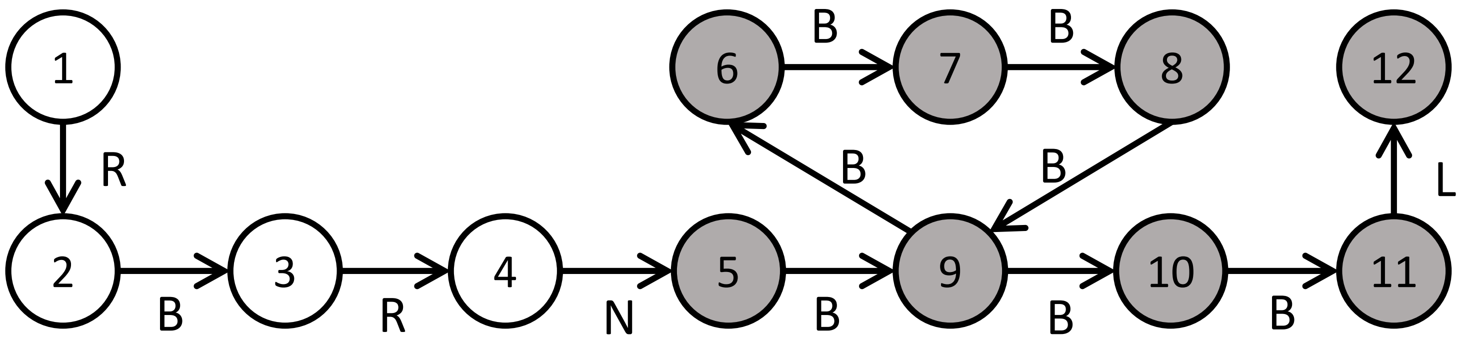


Example: Resizing an array

```
proc DoubleSize ()
    var t, B, v
    Acquire()
    GetLen(out t)
    B := Allocate(2*t)
    SetLen(2*t)
    i := 0
    while (i < t)
        Read(i, out v)
        B[i] := v
        i := i + 1
    Switch(B)
    Release()
```

var A : Array, len : Nat, lock : ThreadID ∪ {nil}

Acquire():	[assume lock == nil; lock := tid]	R
Release():	[assert lock == tid; lock := nil]	L
GetLen(*out* r):	[assert lock == tid; r := len]	B
GetLen2(*out* r):	[r := len]	N
SetLen (v):	[assert lock == tid; len := v]	N
Read(i, out r):	[assert lock == tid; r := A[i]]	B
Switch(B):	[assert lock == tid; A := B]	B



Summarization ([S]→ A)

Within an atomic block, sequential reasoning suffices to obtain an atomic action.

```
Acquire: [assume lock == nil; lock := tid;
Read:     assert lock == tid; t := x;
Write:    assert lock == tid; x := t + 1;
Release:  assert lock == tid; lock := nil    ]
```

⇩

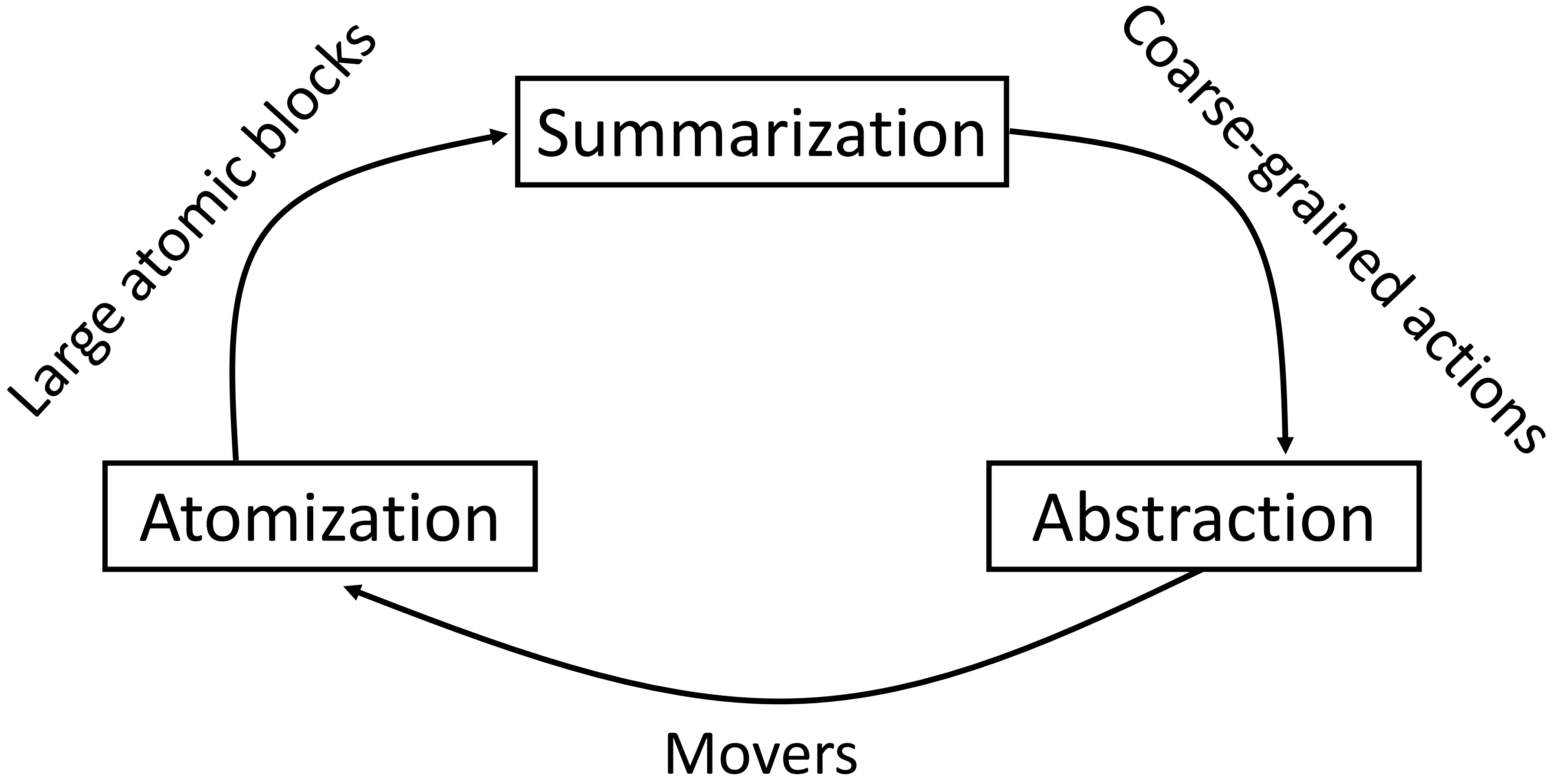
```
Inc: [assume lock == nil; x := x + 1]
```

Abstraction (A → B)

(G_1, A_1) refines (G_2, A_2)
iff
$G_2 \Rightarrow G_1$
$G_2 \bullet A_1 \Rightarrow A_2$

[g := g + 1]	refines	[assert 0 ≤ g; g := g + 1]
[g := g + 1]	refines	[var g_ = g; havoc g; assume g_ ≤ g]
[g := h]	refines	/* 0 ≤ h */ [havoc g; assume 0 ≤ g]

Atomization, summarization, and abstraction are symbiotic [Elmas, Q, Tasiran 2009]



Abstraction enables stronger mover types

Read and Write are conflicting (non-movers)

```
action Read(out r):          action Write (v):
  r := x                       x := v
```

⇩

```
action Read(out r):          action Write (v):
  assert lock == tid           assert lock == tid
  r := x                       x := v
```

Strengthening the gates satisfies commutativity

Inc is blocking

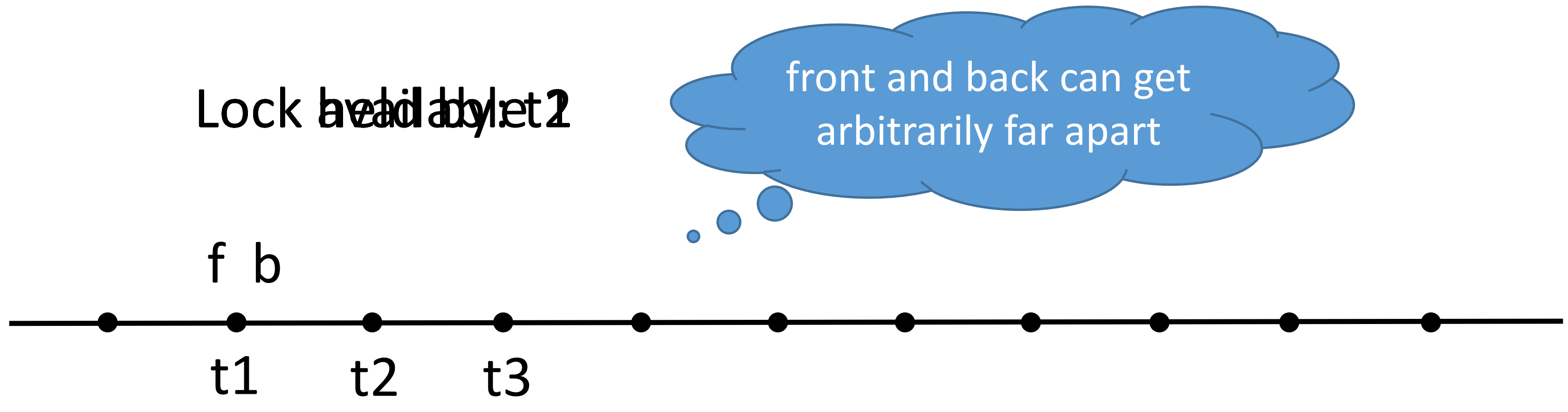
```
action Inc():
  assume lock == nil
  x := x + 1
```

⇩

```
action Inc():
  x := x + 1
```

Weakening the transition makes Inc nonblocking

Example: Ticket lock




```
var back
var front
```

```
Acquire() {
   var ticket
   [ ticket := back; back := back + 1 ]
   [ assume ticket == front ]
}
```

```
Release() {
   [ front := front + 1 ]
}
```

Example: Ticket lock

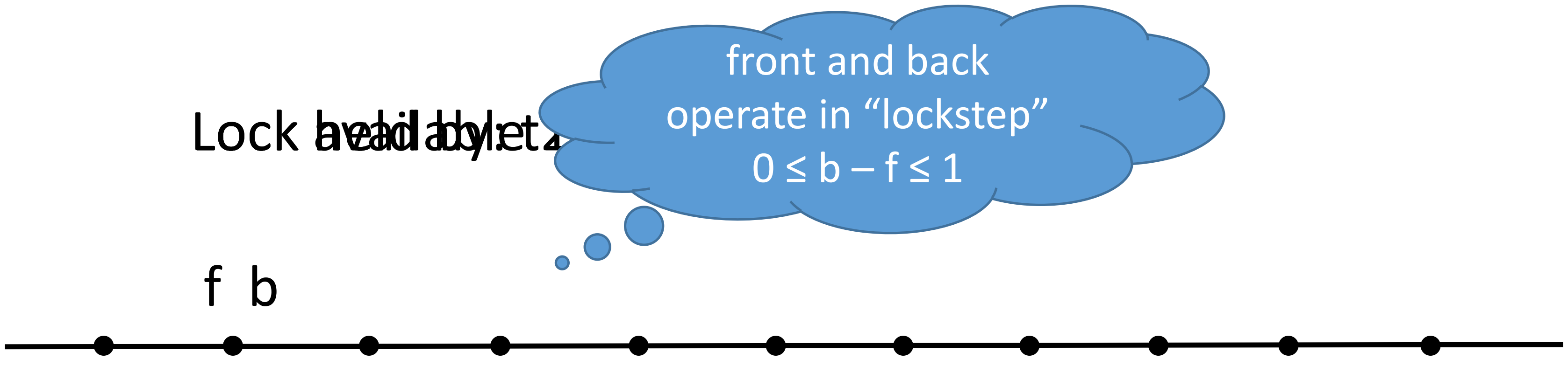
If we could treat
Acquire as atomic …

```
var back
var front
```

```
Acquire() {
  var ticket
  [ ticket := back; back := back + 1;
    assume ticket == front ]
}
```

```
Release() {
  [ front := front + 1 ]
}
```

Example: Ticket lock




```
var back
var front

Acquire() {
   var ticket
   [ assume front == back;
     back := back + 1 ]
}

Release() {
   [ front := front + 1 ]
}
```

```
var T                          Acquire() {                                                        Release() {
var front                        var ticket                                                         [ front := front + 1 ]
                                 [ havoc ticket; assume !T[ticket]; T[ticket] := true ]   R         }
                                 [ assume ticket == front ]   N
Invariant: T = (-∞, back)      }
```

```
var back                       Acquire() {                                                        Release() {
var front                        var ticket                                                         [ front := front + 1 ]
                                 [ ticket := back; back := back + 1 ]   N                           }
                                 [ assume ticket == front ]   N
                               }
```

```
var T
var front

Acquire() {
   var ticket
   [ havoc ticket; assume !T[ticket]; T[ticket] := true;
     assume ticket == front ]
}

Release() {
   [ front := front + 1 ]
}
```

```
var T
var front

Acquire() {
   var ticket
   [ havoc ticket; assume !T[ticket]; T[ticket] := true ]   R
   [ assume ticket == front ]   N
}

Release() {
   [ front := front + 1 ]
}
```

```
var back
var front

Acquire() {
   var ticket
   [ ticket := back; back := back + 1 ]   N
   [ assume ticket == front ]   N
}

Release() {
   [ front := front + 1 ]
}
```

```
var lock        Acquire() {                      Release() {
                  [ assume lock == nil;            [ assert lock == tid;
                    lock := tid ]                    lock := nil ]
                }                                }
```

Ticket lock has the same abstract spec as spinlock

Invariant: if lock == nil then T = (-∞, front) else T = (-∞, front]

```
var T           Acquire() {                                  Release() {
var front         [ assume !T[front]; T[front] := true ]       [ front := front + 1 ]
                }                                            }
```

```
var T           Acquire() {                                                  Release() {
var front         var ticket                                                   [ front := front + 1 ]
                  [ havoc ticket; assume !T[ticket]; T[ticket] := true ]  R  }
                  [ assume ticket == front ]  N
                }
```

```
var back        Acquire() {                                  Release() {
var front         var ticket                                   [ front := front + 1 ]
                  [ ticket := back; back := back + 1 ]  N    }
                  [ assume ticket == front ]  N
                }
```

Local reasoning is challenging

Read(*out* r): [assert lock == tid; r := x]
Write(v): [assert lock == tid; x := v]

$$lock == tid1 \land lock == tid2 \vDash [r := x;\ x := v] \Rightarrow [x := v;\ r := x]$$

Commutativity of Read and Write requires information about two tid variables in different scopes being distinct from each other

Patterns of concurrency control

- Exclusive access
  - thread identifier, lock-protected access, memory ownership, …
- Shared/exclusive access
  - barrier, read-shared memory access, vote collection, …

- Need to encode variety of patterns
- … without baking in each pattern

Our solution

1. Use linear typing and logical reasoning to establish global invariant

2. Exploit established invariant as a “free assumption” in verification conditions for commutativity and noninterference reasoning

Linear type system

1. Variables (global, local, parameters) have linearity annotations
2. Type system infers availability at every control location

```
// x available
// y unavailable
y := x
// x unavailable
// y available
```

```
proc P (lin p)

// x available
call P(x)
// x available
```

```
proc P (lin_in p)

// x available
call P(x)
// x unavailable
```

```
proc P (lin_out p)

// x unavailable
call P(x)
// x available
```

3. $\Gamma: Value \rightarrow 2^{\mathbb{N}}$ $\quad$ e.g.: Γ(tid) = {tid} $\quad$ Γ(tidSet) = tidSet
4. $Collect(c) = \left(\uplus_{x \in Lin \cap Glob} \Gamma(g(x))\right) \uplus \left(\uplus_{(x,\ell) \in Available(c)} \Gamma(\ell(x))\right)$
5. Invariant: $Collect(c)$ is a set

Exploiting the free assumption

Read(*linear* tid, *out* r): [assert lock == tid; r := x]
Write(*linear* tid, v): [assert lock == tid; x := v]

$$\text{lock} == \text{tid1} \land \text{lock} == \text{tid2} \vDash [r := x;\ x := v] \Rightarrow [x := v;\ r := x]$$ ✗

$$\text{IsSet}(\{\text{tid1}\} \uplus \{\text{tid2}\}) \land \text{lock} == \text{tid1} \land \text{lock} == \text{tid2} \vDash [r := x;\ x := v] \Rightarrow [x := v;\ r := x]$$ ✓

↓ simplifies to

$$\text{tid1} \neq \text{tid2}$$

Atomic actions must preserve invariant

```
var lock : nat?
var linear slots : set<nat>

call Main()

proc Main
  while (*)
    async Worker()

proc Worker()
  var linear tid : nat
  call tid := ALLOC()
  call ACQUIRE(tid)
  // critical section
  call RELEASE(tid)

right ALLOC() : (linear tid: nat)
  assume tid ∈ slots
  slots := slots – tid

right ACQUIRE(linear tid: nat)
  assume lock == NIL
  lock := tid

left RELEASE(linear tid: nat)
  assert lock == tid
  lock := NIL
```

$$\Gamma(\text{slots}') \uplus \Gamma(\text{tid}') \subseteq \Gamma(\text{slots})$$

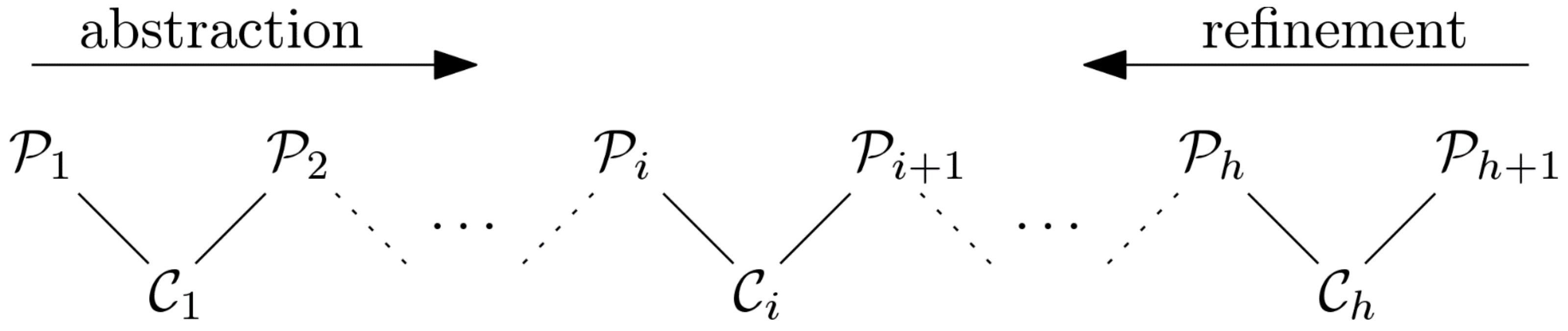
Patterns of concurrency control

- Exclusive access
  - thread identifier, lock-protected access, memory ownership, …
- Shared/exclusive access
  - barrier, read-shared memory access, vote collection, …

- Need to encode variety of patterns
- … without baking in each pattern

- All patterns mentioned above are encodable by a suitable choice for Γ

A chain of concurrent programs



- $\mathcal{P}_1, \ldots, \mathcal{P}_{h+1}$ are concurrent programs
  - $\mathcal{P}_i$ refines $\mathcal{P}_{i+1}$ for all $i \in [1, h]$
- $\mathcal{C}_1, \ldots, \mathcal{C}_h$ are concurrent checker programs
  - safety of $\mathcal{C}_i$ justifies $\mathcal{C}_i$ refines $\mathcal{C}_{i+1}$ for all $i \in [1, h]$
- Goal
  - Express $\mathcal{P}_1, \ldots, \mathcal{P}_{h+1}$ and the key insight of $\mathcal{C}_1, \ldots, \mathcal{C}_h$ in a single layered concurrent program $\mathcal{LP}$
  - Generate $\mathcal{C}_1, \ldots, \mathcal{C}_h$ automatically from $\mathcal{LP}$

1

```
var b : bool


call Main()

proc Main
  while (*)
    async Worker()

proc Worker()

  call Alloc()
  call Enter()
  // critical section
  call Leave()

proc Alloc() : ()
  skip


proc Enter()
  var success : bool
  while (true)
    call success := CAS()
    if (success) break

proc Leave()
  call RESET()

atomic CAS() : (s: bool)
  if (b) s := false
  else   s, b := true, true

atomic RESET()
  assert b
  b := false
```

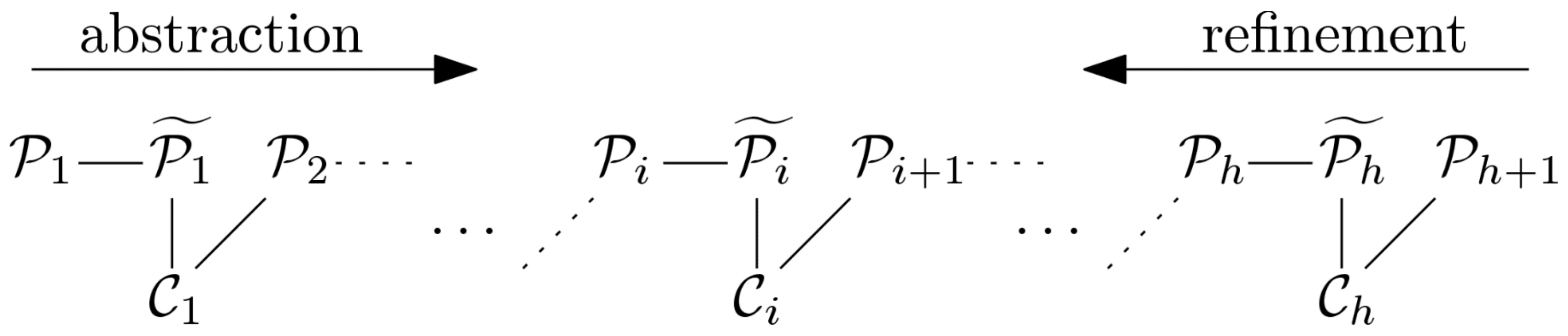
2

```
var lock : nat?
var linear slots : set<nat>

call Main()

proc Main
  while (*)
    async Worker()

proc Worker()
  var linear tid : nat
  call tid := ALLOC()
  call ACQUIRE(tid)
  // critical section
  call RELEASE(tid)

right ALLOC() : (linear tid: nat)
  assume tid ∈ slots
  slots := slots – tid

right ACQUIRE(linear tid: nat)
  assume lock == NIL
  lock := tid



left RELEASE(linear tid: nat)
  assert lock == tid
  lock := NIL
```

3

```
call SKIP()

both SKIP()
  skip
```

A chain of concurrent programs



- $\mathcal{P}_1, \ldots, \mathcal{P}_{h+1}$ are concurrent programs
  - $\mathcal{P}_i$ refines $\mathcal{P}_{i+1}$ for all $i \in [1, h]$
- $\mathcal{C}_1, \ldots, \mathcal{C}_h$ are concurrent checker programs
  - safety of $\mathcal{C}_i$ justifies $\mathcal{P}_i$ refines $\mathcal{P}_{i+1}$ for all $i \in [1, h]$
- $\mathcal{C}_i$ is constructed in two steps
  - (optionally) add computation to $\mathcal{P}_i$ to get $\tilde{\mathcal{P}}_i$
  - instrument $\tilde{\mathcal{P}}_i$ to obtain $\mathcal{C}_i$

```
var b : bool

proc Main
  while (*)
    async Worker()

proc Worker()

  call Alloc()
  call Enter()
  // critical section
  call Leave()

proc Alloc() : ()


proc Enter()
  var success : bool
  while (true)
    call success := CAS()
    if (success)

      break

proc Leave()
  call RESET()


atomic CAS() : (s: bool)
  if (b) s := false
  else   s, b := true, true

atomic RESET()
  assert b
  b := false
```

```
var lock : nat?
var linear slots : set<nat>
var pos : nat

predicate InvAlloc
  slots = [pos, ∞)

iaction iIncr() : (linear tid : nat)
  assert InvAlloc
  tid := pos
  pos := pos + 1
  slots := slots − tid

iaction iSetLock(v: nat)
  lock := v
```

```
var b : bool

proc Main
  while (*)
    async Worker()

proc Worker()
  var linear tid: nat
  call tid := Alloc()
  call Enter(tid)
  // critical section
  call Leave(tid)

proc Alloc() : (linear tid: int)
  icall tid := iIncr()

proc Enter(linear tid: int)
  var success : bool
  while (true)
    call success := CAS()
    if (success)
      icall iSetLock(tid)
      break

proc Leave(linear tid: int)
  call RESET()
  icall iSetLock(nil)

atomic CAS() : (s: bool)
  if (b) s := false
  else   s, b := true, true

atomic RESET()
  assert b
  b := false
```

Layered concurrent program

```
var b@[0,1] : bool
var lock@[1,2] : nat?
var linear slots@[1,2] : set<nat>
var pos@[1,1] : nat

call Main()

proc Main@2()
refines SKIP
  while (*)
    async call Worker()

left proc Worker@2()
refines SKIP
  var linear tid@1 : nat
  call tid := Alloc()
  call Enter(tid)
  call Leave(tid)
```

```
right ACQUIRE@[2,2](linear tid : nat)
  assume lock == 0
  lock := tid

left RELEASE@[2,2](linear tid : nat)
  assert lock == tid
  lock := 0

proc Enter@1(linear tid@1: nat)
refines ACQUIRE
  var success@0 : bool
  while (true)
    call success := Cas()
    if (success)
      icall iSetLock(tid)
      break

proc Leave@1(linear tid@1 : nat)
refines RELEASE
  call Reset()
  icall iSetLock(nil)

iaction iSetLock@1(v: nat?)
  lock := v
```

```
right ALLOC@[2,2]() : (linear tid : nat)
  assume tid ∈ slots
  slots := slots - tid

proc Alloc@1() : (linear tid@1 : nat)
refines ALLOC
  icall tid := iIncr()

predicate InvAlloc
  slots = [pos, ∞)

iaction iIncr@1() : (linear tid : nat)
  assert InvAlloc
  tid := pos
  pos := pos + 1
  slots := slots – tid
```

```
atomic CAS@[1,1]() : (s: bool)
  if (b) s := false
  else   s, b := true, true

atomic RESET@[1,1]()
  assert b
  b := false

proc Cas@0() : (success@0 : bool)
refines CAS

proc Reset@0()
refines RESET
```

Layered concurrent program

Layer 1

```
var b@[0,1] : bool
var lock@[1,2] : nat?
var linear slots@[1,2] : set<nat>
var pos@[1,1] : nat

call Main()

proc Main@2()
refines SKIP
  while (*)
    async call Worker()

left proc Worker@2()
refines SKIP
  var linear tid@1 : nat
  call tid := Alloc()
  call Enter(tid)
  call Leave(tid)
```

```
right ACQUIRE@[2,2](linear tid : nat)
  assume lock == 0
  lock := tid

left RELEASE@[2,2](linear tid : nat)
  assert lock == tid
  lock := 0

proc Enter@1(linear tid@1: nat)
refines ACQUIRE
  var success@0 : bool
  while (true)
    call success := CAS()
    if (success)
      icall iSetLock(tid)
      break

proc Leave@1(linear tid@1 : nat)
refines RELEASE
  call RESET()
  icall iSetLock(nil)

iaction iSetLock@1(v: nat?)
  lock := v
```

```
right ALLOC@[2,2]() : (linear tid : nat)
  assume tid ∈ slots
  slots := slots - tid

proc Alloc@1() : (linear tid@1 : nat)
refines ALLOC
  icall tid := iIncr()

predicate InvAlloc
  slots = [pos, ∞)

iaction iIncr@1() : (linear tid : nat)
  assert InvAlloc
  tid := pos
  pos := pos + 1
  slots := slots – tid
```

```
atomic CAS@[1,1]() : (s: bool)
  if (b) s := false
  else   s, b := true, true

atomic RESET@[1,1]()
  assert b
  b := false

proc Cas@0() : (success@0 : bool)
refines CAS

proc Reset@0()
refines RESET
```

Layered concurrent program

Layer 2

```
var b@[0,1] : bool
var lock@[1,2] : nat?
var linear slots@[1,2] : set<nat>
var pos@[1,1] : nat

call Main()

proc Main@2()
refines SKIP
  while (*)
    async call Worker()

left proc Worker@2()
refines SKIP
  var linear tid@1 : nat
  call tid := ALLOC()
  call ACQUIRE(tid)
  call RELEASE(tid)
```

```
right ACQUIRE@[2,2](linear tid : nat)
  assume lock == 0
  lock := tid

left RELEASE@[2,2](linear tid : nat)
  assert lock == tid
  lock := 0

proc Enter@1(linear tid@1: nat)
refines ACQUIRE
  var success@0 : bool
  while (true)
    call success := Cas()
    if (success)
      icall iSetLock(tid)
      break

proc Leave@1(linear tid@1 : nat)
refines RELEASE
  call Reset()
  icall iSetLock(nil)

iaction iSetLock@1(v: nat?)
  lock := v
```

```
right ALLOC@[2,2]() : (linear tid : nat)
  assume tid ∈ slots
  slots := slots - tid

proc Alloc@1() : (linear tid@1 : nat)
refines ALLOC
  icall tid := iIncr()

predicate InvAlloc
  slots = [pos, ∞)

iaction iIncr@1() : (linear tid : nat)
  assert InvAlloc
  tid := pos
  pos := pos + 1
  slots := slots – tid
```

```
atomic CAS@[1,1]() : (s: bool)
  if (b) s := false
  else   s, b := true, true

atomic RESET@[1,1]()
  assert b
  b := false

proc Cas@0() : (success@0 : bool)
refines CAS

proc Reset@0()
refines RESET
```

Layered concurrent program

Layer 3

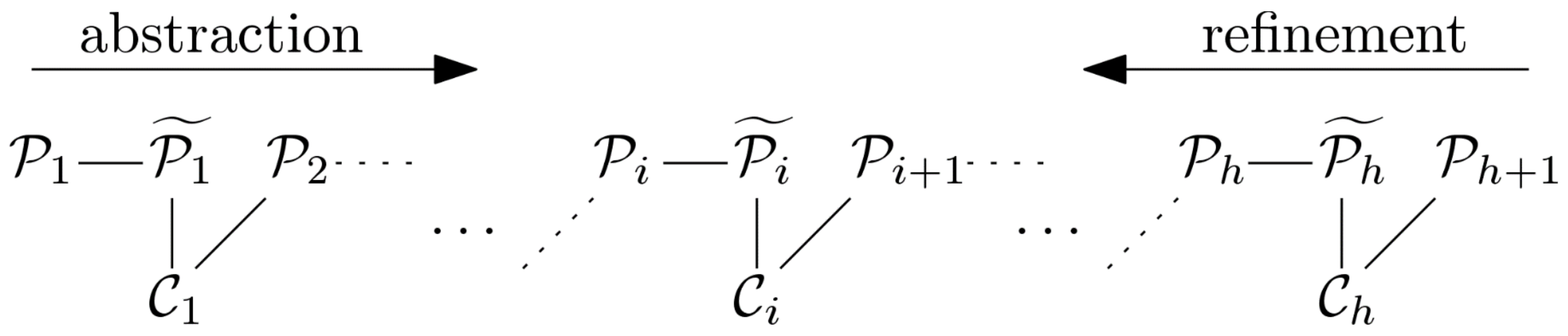
```
var b@[0,1] : bool
var lock@[1,2] : nat?
var linear slots@[1,2] : set<nat>
var pos@[1,1] : nat

call SKIP()

proc Main@2()
refines SKIP
  while (*)
    async call Worker()

left proc Worker@2()
refines SKIP
  var linear tid@1 : nat
  call tid := Alloc()
  call Enter(tid)
  call Leave(tid)
```

```
right ACQUIRE@[2,2](linear tid : nat)
  assume lock == 0
  lock := tid

left RELEASE@[2,2](linear tid : nat)
  assert lock == tid
  lock := 0

proc Enter@1(linear tid@1: nat)
refines ACQUIRE
  var success@0 : bool
  while (true)
    call success := Cas()
    if (success)
      icall iSetLock(tid)
      break

proc Leave@1(linear tid@1 : nat)
refines RELEASE
  call Reset()
  icall iSetLock(nil)

iaction iSetLock@1(v: nat?)
  lock := v
```

```
right ALLOC@[2,2]() : (linear tid : nat)
  assume tid ∈ slots
  slots := slots - tid

proc Alloc@1() : (linear tid@1 : nat)
refines ALLOC
  icall tid := iIncr()

predicate InvAlloc
  slots = [pos, ∞)

iaction iIncr@1() : (linear tid : nat)
  assert InvAlloc
  tid := pos
  pos := pos + 1
  slots := slots – tid
```

```
atomic CAS@[1,1]() : (s: bool)
  if (b) s := false
  else   s, b := true, true

atomic RESET@[1,1]()
  assert b
  b := false

proc Cas@0() : (success@0 : bool)
refines CAS

proc Reset@0()
refines RESET
```

A chain of concurrent programs



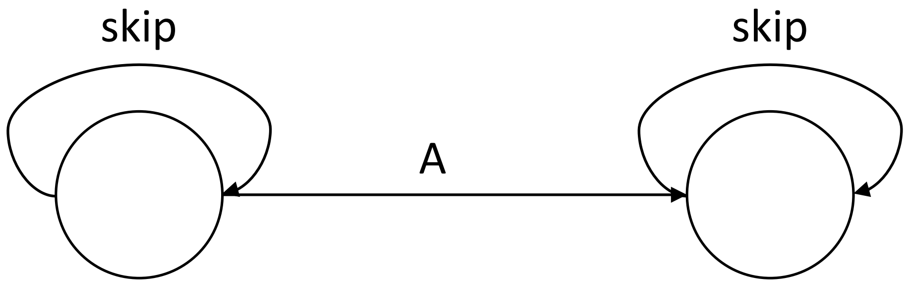
- $\mathcal{P}_1, \ldots, \mathcal{P}_{h+1}$ are concurrent programs
  - $\mathcal{P}_i$ refines $\mathcal{P}_{i+1}$ for all $i \in [1, h]$
- $\mathcal{C}_1, \ldots, \mathcal{C}_h$ are concurrent checker programs
  - safety of $\mathcal{C}_i$ justifies $\mathcal{P}_i$ refines $\mathcal{P}_{i+1}$ for all $i \in [1, h]$
- $\mathcal{C}_i$ is constructed in two steps
  - (optionally) add computation to $\mathcal{P}_i$ to get $\tilde{\mathcal{P}}_i$
  - instrument $\tilde{\mathcal{P}}_i$ to obtain $\mathcal{C}_i$

Making interference explicit

```
proc Leave(linear tid)
refines RELEASE
  yield
  call RESET()
  icall iSetLock(nil)
  yield
```

```
proc Enter(linear tid)
refines ACQUIRE
  yield
  while (true)
    call success := CAS()
    if (success)
      icall iSetLock(tid)
      break;
    yield
  yield
```

Refinement checking



$g_0 \longrightarrow g_1 \longrightarrow g_2 \longrightarrow \bullet\bullet\bullet \longrightarrow g_n$

A and skip are disjoint

pc_0 = false
assert $g_i \neq g_{i+1} \Rightarrow \neg pc_i \wedge A(g_i, g_{i+1})$
$pc_{i+1} = pc_i \vee g_i \neq g_{i+1}$
assert pc_n

In general

pc_0 = false
assert $g_i \neq g_{i+1} \Rightarrow \neg pc_i \wedge A(g_i, g_{i+1})$
$pc_{i+1} = pc_i \vee g_i \neq g_{i+1}$

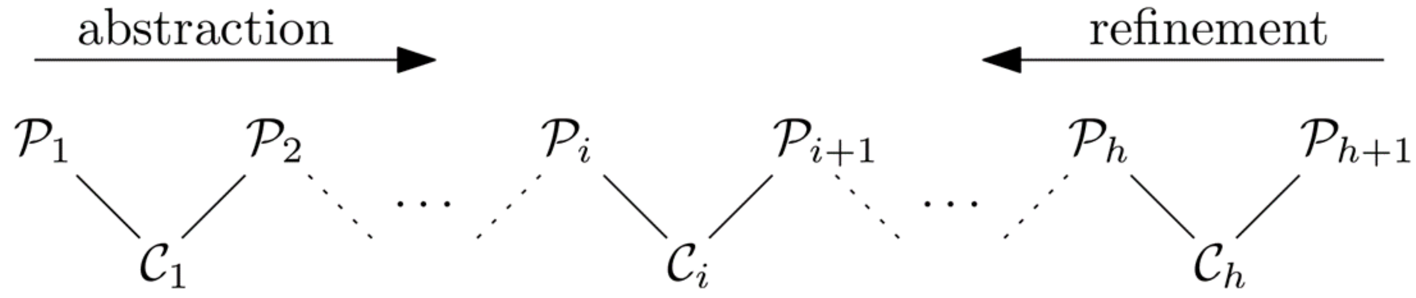
$done_0$ = false
$done_{i+1} = done_i \vee A(g_i, g_{i+1})$
assert $done_n$

```
macro *CHANGED* is !(lock == _lock && slots == _slots)
macro *RELEASE* is lock == nil && slots == _slots
macro *ACQUIRE* is _lock == nil && lock == tid && slots == _slots
```

```
proc Leave(linear tid)
  var _lock, _slots, pc, done
  pc, done := false, false
  yield
  _lock, _slots := lock, slots
  assume pc || lock == tid

  call RESET()
  icall iSetLock(nil)

  assert *CHANGED* ==> (!pc && *RELEASE*)
  pc := pc || *CHANGED*
  done := done || *RELEASE*
  yield
  assert done
```

```
proc Enter(linear tid)
  var success, _lock, _slots, pc, done
  pc, done := false, false
  yield
  _lock, _slots := lock, slots
  assume pc || true

  while (true)
    call success := CAS()
    if (success)
      icall iSetLock(tid)
      break;

    assert *CHANGED* ==> (!pc && *ACQUIRE*)
    pc := pc || *CHANGED*
    done := done || *ACQUIRE*
    yield
    _lock, _slots := lock, slots
    assume pc || true

  assert *CHANGED* ==> (!pc && *ACQUIRE*)
  pc := pc || *CHANGED*
  done := done || *ACQUIRE*
  yield
  assert done
```

So far …



- How do we verify concurrent checker programs $\mathcal{C}_1, \ldots, \mathcal{C}_h$
  - Pick your favorite concurrent verifier
- CIVL implements the Owicki-Gries method in two steps
  - compile away interference using invariants attached to yield statements
  - leverage sequential verification-condition generation

Compiling interference away

yield I

```
assert I
check noninterference
havoc globals
assume I
update snapshot
```

call P

```
check noninterference
call P
update snapshot
```

async P

```
if *
  call P
  assume false
```

call P1 || P2

```
check noninterference
if *
  call P1
  assume false
elsif *
  call P2
  assume false
havoc call targets
havoc globals
assume post(P1) ∧ post(P2)
update snapshot
```

check noninterference

assert $\forall$locals. I_1(locals, snapshot) => I_1(locals, globals)
assert $\forall$locals. I_2(locals, snapshot) => I_2(locals, globals)
…

New verification problems introduced by CIVL

- CIVL expresses gated atomic actions as an atomic code block
- Does atomic block A refine atomic block B?
  - In checker program
  - In commutativity checking
- Is atomic block A nonblocking?
  - checking a left mover

Atomic block

GlobalVar = $\{g_1, \ldots, g_m\}$
LocalVar = $\{l_1, \ldots, l_n\}$

S ::= x := e | assume e | assert e | S ; S | S ■ S

Good(S) = $\{ G \mid \neg\exists L. (G \cdot L, S) \Rightarrow^* \bot \}$
Trans(S) = $\{ (G, G') \mid \exists L,L'. (G \cdot L, S) \Rightarrow^* (G' \cdot L', \varepsilon) \}$

S1 refines S2 iff
- Good(S2) $\subseteq$ Good(S1)
- Good(S2)•Trans(S1) $\subseteq$ Trans(S2)

S is nonblocking iff
- Good(S) $\subseteq \exists G'$. Trans(G, G')

Calculating Good and Trans

$wp(x := e, \varphi) = \varphi[x/e]$
$wp(\text{assume } e, \varphi) = e \Rightarrow \varphi$
$wp(\text{assert } e, \varphi) = e \wedge \varphi$
$wp(S1 ; S2, \varphi) = wp(S1, wp(S2, \varphi))$
$wp(S1 \blacksquare S2, \varphi) = wp(S1, \varphi) \wedge wp(S2, \varphi)$

$Good(S) = \forall\, l_1, \ldots, l_n.\ wp(S, true)$

$tr(x := e, \varphi) = \varphi[x/e]$
$tr(\text{assume } e, \varphi) = e \wedge \varphi$
$tr(\text{assert } e, \varphi) = e \wedge \varphi$
$tr(S1 ; S2, \varphi) = tr(S1, tr(S2, \varphi))$
$tr(S1 \blacksquare S2, \varphi) = tr(S1, \varphi) \vee tr(S2, \varphi)$

$Trans(S) = \exists l_1, \ldots, l_n.\ tr(S, g_1 = g_1' \wedge \ldots \wedge g_m = g_m')$

S	Good(S)	Trans(S)
$l := g + 1$ $g := l$	true	$g + 1 = g'$
assume $g \leq l$ $g := l$	true	$\exists\, l.\ g \leq l \wedge l = g'$
assume $g \leq l$ $g := l$ assert $0 \leq g$	$\forall\, l.\ g \leq l \Rightarrow 0 \leq l$	$\exists\, l.\ g \leq l \wedge 0 \leq l \wedge l = g'$

Quantifiers are a problem

Is $\varphi \Rightarrow \psi$ valid?

- SMT solvers become unpredictable
- Universal quantifier in φ is a problem
- Existential quantifier in ψ is a problem

Heuristics for eliminating quantifiers

Eliminate x from $\exists x.\ \varphi(x, y)$:
- find $E(y)$ such that $\varphi(x, y) \Rightarrow x = E(y)$ is valid
- $\exists x.\ \varphi(x, y)$ is equivalent to $\varphi(E(y), y)$

Eliminate x from $\exists x.\ \varphi(x, y)$:
- split φ into $\varphi_1 \vee \varphi_2$
- find $E_1(y)$ and $E_2(y)$ such that $\varphi_1(x, y) \Rightarrow x = E_1(y)$ and $\varphi_2(x, y) \Rightarrow x = E_2(y)$
- $\exists x.\ \varphi(x, y)$ is equivalent to $\varphi_1(E_1(y), y) \vee \varphi_2(E_2(y), y)$

Look for equalities in path condition:
$x = e$
$e' = A[e := x]$ ➔ $x = e'[e]$
…

Eliminate x from $\forall x.\ \varphi(x, y)$:
- find $E(y)$ such that $\varphi(x, y) \vee x = E(y)$ is valid
- $\forall x.\ \varphi(x, y)$ is equivalent to $\varphi(E(y), y)$

Eliminate x from $\forall x.\ \varphi(x, y)$:
- split φ into $\varphi_1 \wedge \varphi_2$
- find $E_1(y)$ and $E_2(y)$ such that $\varphi_1(x, y) \vee x = E_1(y)$ and $\varphi_2(x, y) \vee x = E_2(y)$
- $\forall x.\ \varphi(x, y)$ is equivalent to $\varphi_1(E_1(y), y) \wedge \varphi_2(E_2(y), y)$

CIVL in relation to …

- Floyd-Hoare (rely-guarantee, concurrent separation logic, …)
  - CIVL departs from the orthodoxy of pre/post-conditions
  - CIVL is less modular but more flexible
- Model checking (aka automatic verification of decidable abstractions)
  - CIVL addresses programmer-computer interaction
  - CIVL is less automated but more general
- Types and process algebra
  - CIVL is less automated but more expressive

Unsolved problems

- Concurrent programming language
  - Compiles to CIVL for verification
  - Generates executable code
- Modularity
  - Minimize cross-module interference checks
- Other (more automated) techniques for verifying checker programs
- Better PL and IDE support for understanding layers
- Better decision procedures

$$\frac{0 < N \quad A \subseteq [1,N] \quad B \subseteq [1,N] \quad B \subseteq A \quad |B| == N}{|A| == N}$$