The Civl Verifier

https://civl-verifier.github.io

Bernhard Kragl
Amazon Web Services and *IST Austria**

Shaz Qadeer
Facebook

*Work done while at IST Austria.

The transition system approach

$$
\begin{array}{l}
a\colon\ x := n \\
\begin{array}{lll}
b\colon & \text{acquire}(l) & \| \quad \text{acquire}(l) \\
c\colon & t_1 := x & \| \quad t_2 := x \\
d\colon & x := t_1 + 1 & \| \quad x := t_2 + 1 \\
e\colon & \text{release}(l) & \| \quad \text{release}(l)
\end{array} \\
f\colon\ \text{assert } x = n + 2
\end{array}
$$

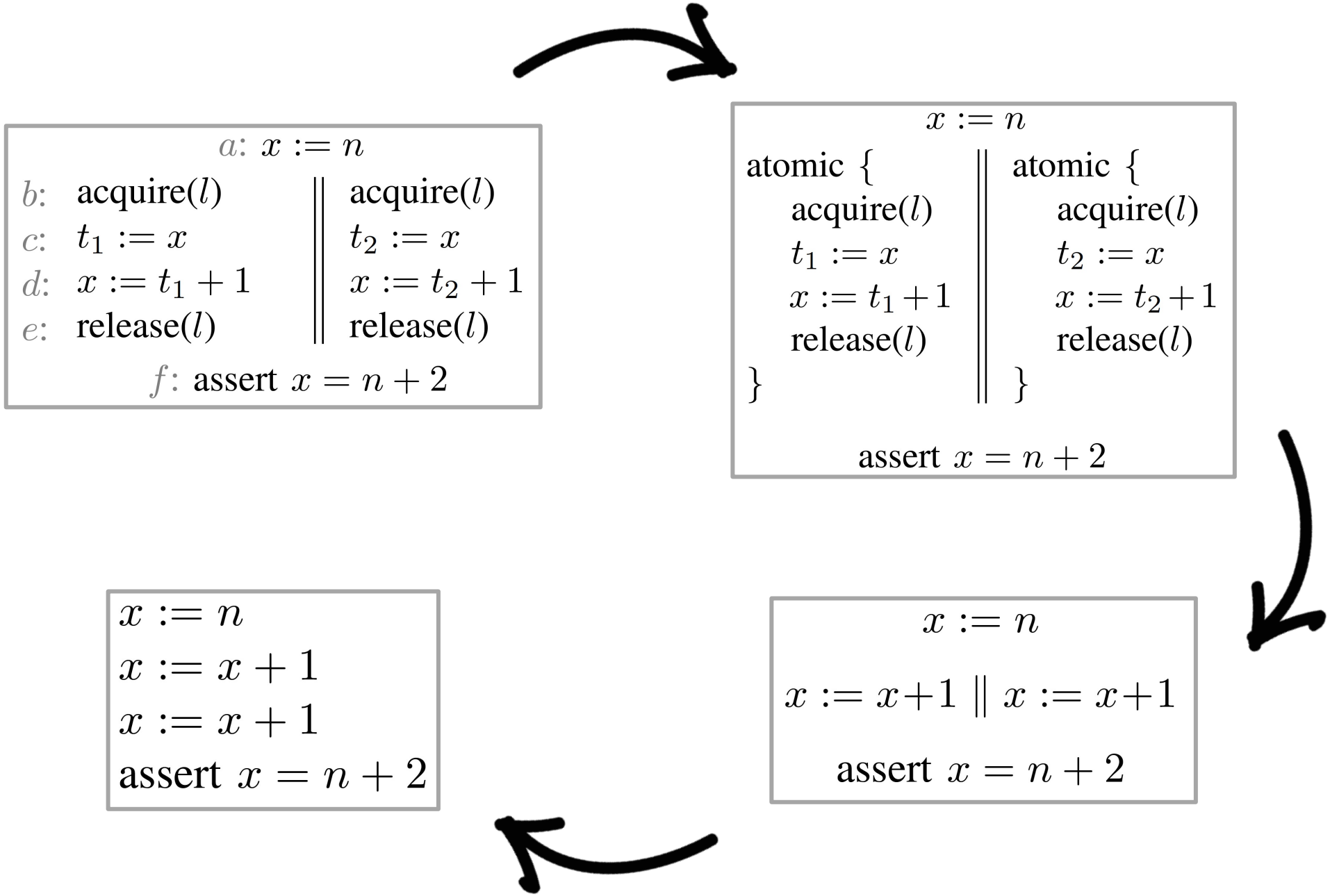
$$
\begin{array}{l}
\mathit{Init}\colon\ pc = pc_1 = pc_2 = a \\
\mathit{Next}\colon \\
\quad pc = a \wedge pc' = pc_1 = pc_2 = b \wedge x' = \\
\vee\ pc_1 = b \wedge pc_1' = c \wedge l = \bigcirc \wedge l' = ① \wedge eq(pc, pc_2, x, t_1, t_2) \\
\vee\ pc_1 = c \wedge pc_1' = d \wedge t_1' = x \wedge eq(pc, pc_2, l, x, t_2) \\
\vee\ pc_1 = d \wedge pc_1' = e \wedge x' = t_1 + 1 \wedge eq(pc, pc_2, l, t_1, t_2) \\
\vee\ pc_1 = e \wedge pc_1' = f \wedge l' = \bigcirc \wedge eq(pc, pc_2, x, t_1, t_2) \\
\vee\ pc_2 = b \wedge pc_2' = c \wedge l = \bigcirc \wedge l' = ② \wedge eq(pc, pc_1, x, t_1, t_2) \\
\vee\ pc_2 = c \wedge pc_2' = d \wedge t_2' = x \wedge eq(pc, pc_1, l, x, t_1) \\
\vee\ pc_2 = d \wedge pc_2' = e \wedge x' = t_2 + 1 \wedge eq(pc, pc_1, l, t_1, t_2) \\
\vee\ pc_2 = e \wedge pc_2' = f \wedge l' = \bigcirc \wedge eq(pc, pc_1, x, t_1, t_2) \\
\vee\ pc_1 = pc_2 = f \wedge pc' = f \wedge eq(pc_1, pc_2, l, x, t_1, t_2) \\
\mathit{Safe}\colon\ (pc = f \Rightarrow x = n + 2) \wedge \\
\qquad (pc_1 \in \{c, d, e\} \Rightarrow l = ①) \wedge (pc_2 \in \{c, d, e\} \Rightarrow l = ②)
\end{array}
$$

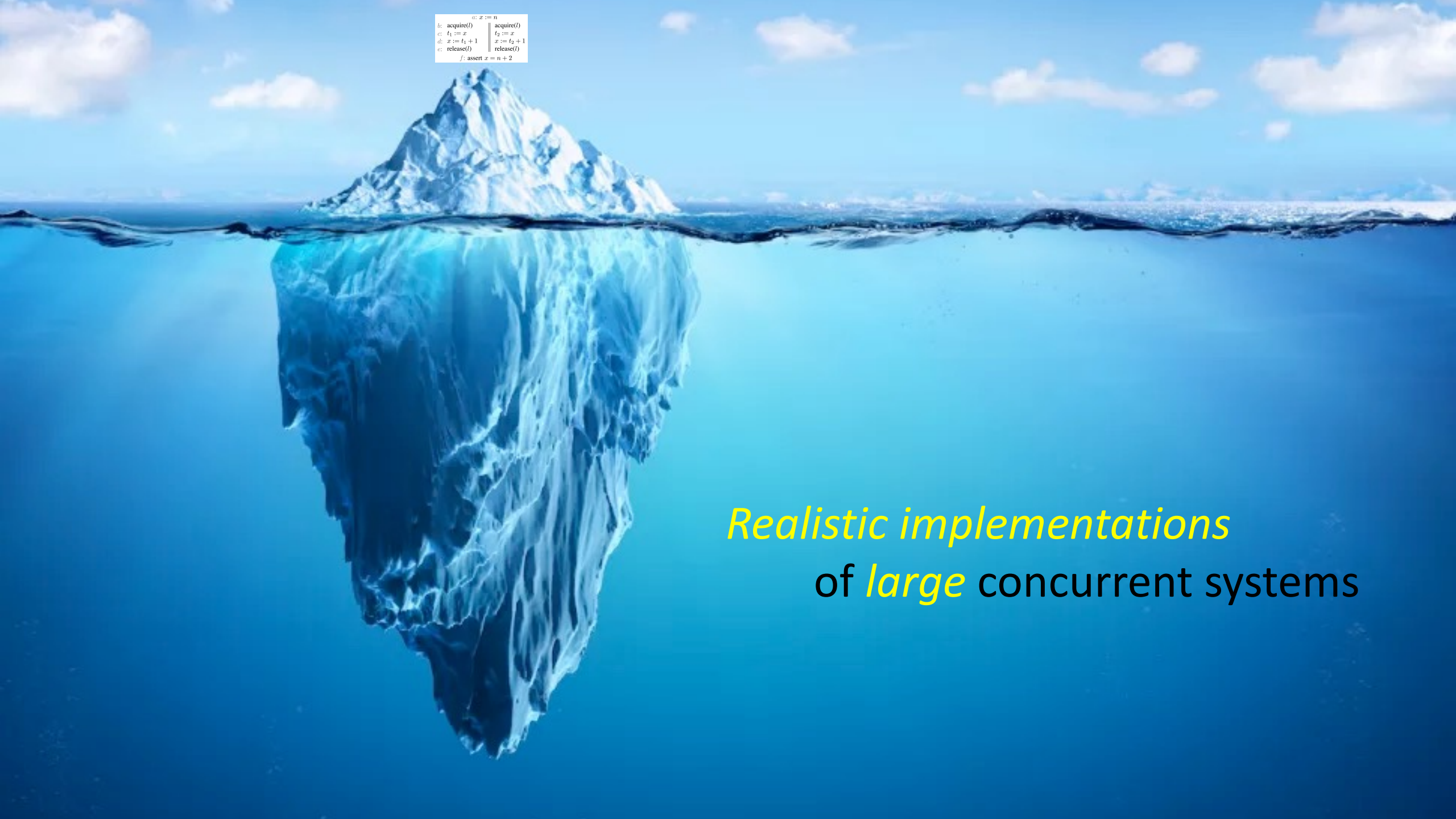
lost program structure

Invent inductive invariant Inv

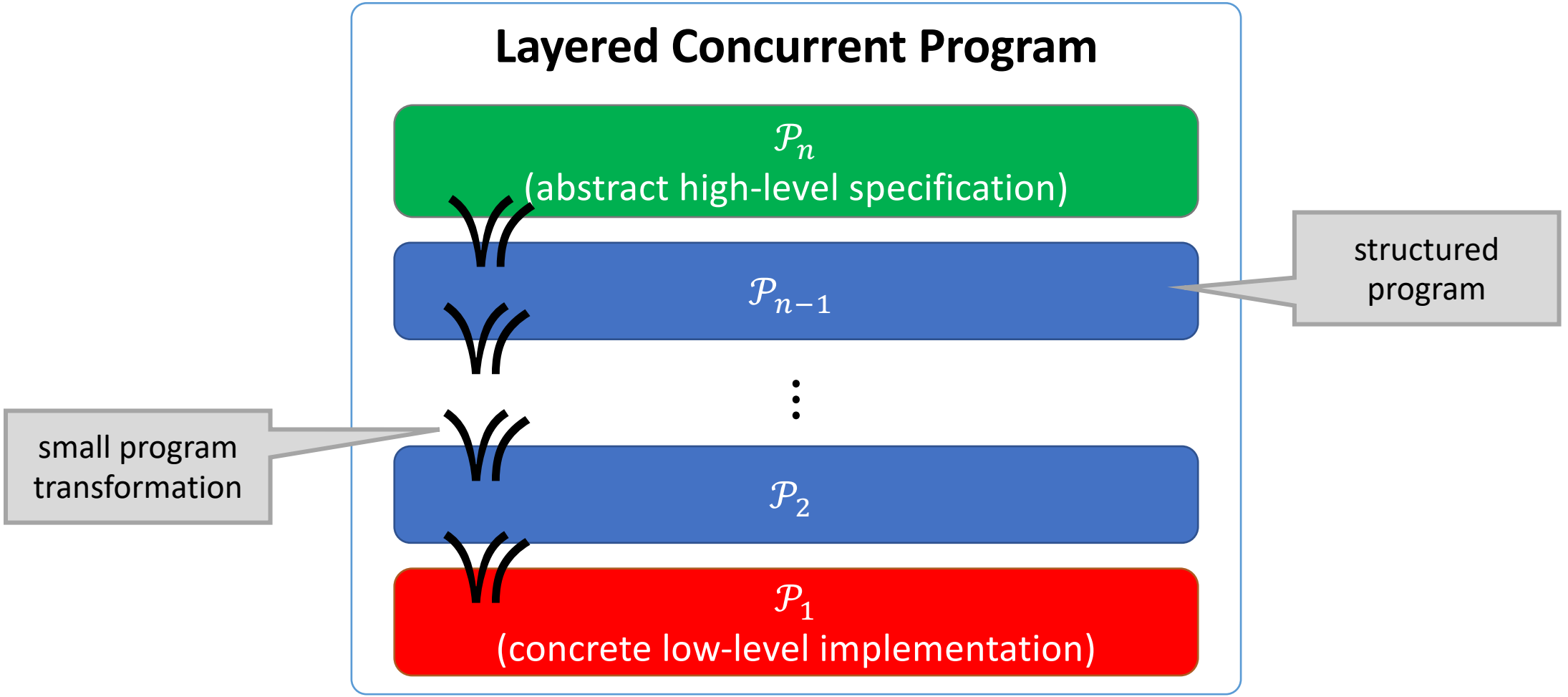
- $Init \Rightarrow Inv$
- $Inv \wedge Next \Rightarrow Inv'$
- $Inv \Rightarrow Safe$

complicated invariants


a: x := n
b: acquire(l) || acquire(l)
c: t1 := x || t2 := x
d: x := t1 + 1 || x := t2 + 1
e: release(l) || release(l)
f: assert x = n + 2
x := n
atomic { acquire(l) t1 := x x := t1 + 1 release(l) } || atomic { acquire(l) t2 := x x := t2 + 1 release(l) }
assert x = n + 2
x := n
x := x + 1 || x := x + 1
assert x = n + 2
x := n
x := x + 1
x := x + 1
assert x = n + 2


a: $x := n$
b: acquire(l) || acquire(l)
c: $t_1 := x$ || $t_2 := x$
d: $x := t_1 + 1$ || $x := t_2 + 1$
e: release(l) || release(l)
f: assert $x = n + 2$
Realistic implementations
of *large* concurrent systems

Civl: *layered refinement* over *structured concurrent programs*



Tactic 1: Creating atomic blocks

User declares mover types

Right mover

Left/Right mover

Left mover

$$
\begin{array}{l}
a\colon\ x := n \\
\hline
b\colon\ \text{acquire}(l) \quad \| \quad \text{acquire}(l) \\
c\colon\ t_1 := x \quad \| \quad t_2 := x \\
d\colon\ x := t_1 + 1 \quad \| \quad x := t_2 + 1 \\
e\colon\ \text{release}(l) \quad \| \quad \text{release}(l) \\
\hline
f\colon\ \text{assert } x = n + 2
\end{array}
$$

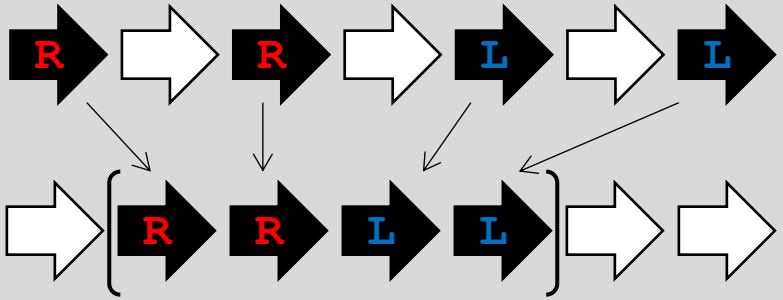
$$
\begin{array}{l}
x := n \\
\hline
\text{atomic } \{ \quad \| \quad \text{atomic } \{ \\
\quad \text{acquire}(l) \quad \| \quad \quad \text{acquire}(l) \\
\quad t_1 := x \quad \| \quad \quad t_2 := x \\
\quad x := t_1 + 1 \quad \| \quad \quad x := t_2 + 1 \\
\quad \text{release}(l) \quad \| \quad \quad \text{release}(l) \\
\} \quad \| \quad \} \\
\hline
\text{assert } x = n + 2
\end{array}
$$

Civl applies reduction

- Pairwise commutativity checks
- Mover sequence check

Lipton's reduction:

Sequences of R*L* are atomic.



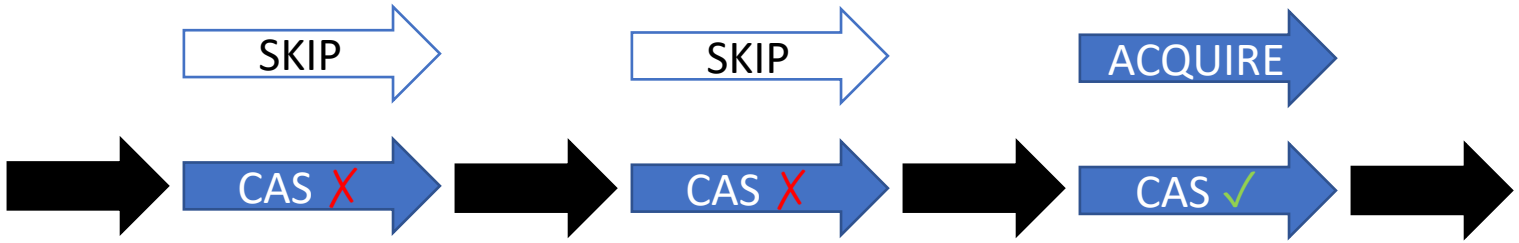
Tactic 2: Creating atomic actions

```
var b: bool

proc Acquire()
   // yield
   while true
      // yield
      exec s := CAS(b, false, true)
      if (s)
         break
   // yield
```

```
action ACQUIRE()
   assume b == false
   b := true
```

Every execution of Acquire behaves like “SKIP*; ACQUIRE; SKIP*”.



Tactic 3: Changing state representation

```
var b: bool

proc Acquire()
  // yield
  while true
    // yield
    exec s := CAS(b, false, true)
    if (s)
      break
  // yield
```

```
var l: Option<ThreadId>

action ACQUIRE(tid)
  assume l == None
  l := Some(tid)
```

Every execution of Acquire behaves like “SKIP*; ACQUIRE; SKIP*”.

Tactic 3: Changing state representation

```
var b: bool

proc Acquire(tid)
  // yield
  while true
    // yield
    exec s := CAS(b, false, true)
    if (s)
      intro [l := Some(tid)]
      break
  // yield
```

```
var l: Option<ThreadId>

action ACQUIRE(tid)
  assume l == None
  l := Some(tid)
```

Every execution of Acquire behaves like “SKIP*; ACQUIRE; SKIP*”.

Tactic 4: Synchronizing asynchrony

```
var x: nat

action ADD(i: nat)
   x := x + i

action ASYNC_SUM(n: nat)
   for i in 1..n
      async ADD(i)

action SUM(n: nat)
   x := x + (n * (n+1)) div 2
```

```
inv SUM_INV(n: nat)
   var j: nat
   assume 0 <= j && j <= n
   x := x + (j * (j+1)) div 2
   for i in j+1..n
      async ADD(i)
```

Asynchronous executions

1: ADD(1) ADD(2) ADD(3) ADD(4) ADD(5) ADD(6) ⋯
2: ADD(2) ADD(5) ADD(1) ADD(4) ADD(6) ADD(3) ⋯
3: ADD(6) ADD(4) ADD(5) ADD(1) ADD(3) ADD(2) ⋯
4: ADD(4) ADD(1) ADD(6) ADD(3) ADD(2) ADD(5) ⋯
⋮

Works for practical protocols:
Paxos, Two-phase commit, Chang-Roberts, …

Foundation: Invariants & Permissions

Yield invariants

named

parameterized

invariant yield_x(i: int)

x ≥ i

procedure double_inc()
requires yield_x(x)
[x := x + 1]
call yield_x(x)
[x := x + 1]
ensures yield_x(**old**(x) + 2)

Linear permissions

disjointness invariants “for free”

procedure Mutator(i: Tid)
…
call MutatorInv(i)
[barrierSet := barrierSet - {i}]
…

invariant MutatorInv(i: Tid)
i ∈ barrierSet

{MutatorInv(i) ∧ MutatorInv(j)}
barrierSet := barrierSet - {i}
{MutatorInv(j)}
✗

Foundation: Invariants & Permissions

Yield invariants

named

parameterized

invariant yield_x(i: int)

x ≥ i

procedure double_inc()
requires yield_x(x)
[x := x + 1]
call yield_x(x)
[x := x + 1]
ensures yield_x(**old**(x) + 2)

Linear permissions

disjointness invariants “for free”

procedure Mutator(*linear* i: Tid)
…
call MutatorInv(i)
[barrierSet := barrierSet - {i}]
…

invariant MutatorInv(*linear* i: Tid)
i ∈ barrierSet

{MutatorInv(i) ∧ MutatorInv(j) ∧ i ≠ j}
barrierSet := barrierSet - {i}
{MutatorInv(j)} ✓

Layered Concurrent Programs

- Write $\mathcal{P}_1 \preccurlyeq \cdots \preccurlyeq \mathcal{P}_n$ as a single $\mathcal{LP}$

- "data layering"
  which variables live on which layer
- "control layering"
  which operations live on which layer

- Express only the changes from one layer to the next

Civl Syntax

Conservative extension of Boogie

Boogie provides
- Imperative language
- Specification features

Civl adds
- async, par, yield statements
- other syntactic extensions: *attributes* {:attr e1, e2, …}

Global variable

```
var l: Option Tid;
```

```
var l: Option Tid;

                    Mover type
procedure {:right}
AcquireSpec(tid: Tid)
modifies l;
{
    assume l == None();
    l := Some(tid);
}

                    Mover type
procedure {:left}
ReleaseSpec(tid: Tid)
modifies l;
{
    assert l == Some(tid);
    l := None();
}
```

```
var l: Option Tid;

procedure {:right}
AcquireSpec(tid: Tid)
modifies l;
{
    assume l == None();
    l := Some(tid);
}

procedure {:left}
ReleaseSpec(tid: Tid)
modifies l;
{
    assert l == Some(tid);
    l := None();
}
```

```
var b: bool;

procedure {:yields}
    {:refines "AcquireSpec"}
Acquire(tid: Tid)
{
    var s: bool;

    while (true)
    {
        call s := CAS_b(false, true);
        if (s) {
            break;
        }
    }
}
```

Yielding procedure

Atomicity spec

Layer range

```
var {:layer 1, 2} l: Option Tid;
```

Layer range

```
procedure {:right} {:layer 2, 2}
AcquireSpec(tid: Tid)
modifies l;
{
  assume l == None();
  l := Some(tid);
}
```

Layer range

```
procedure {:left} {:layer 2, 2}
ReleaseSpec(tid: Tid)
modifies l;
{
  assert l == Some(tid);
  l := None();
}
```

Layer range

```
var {:layer 0, 1} b: bool;
```

Disappearing layer

Introduction layer

```
procedure {:yields} {:layer 1}
  {:refine "c"}
Acquire({:layer 1} tid: Tid)
{
  var s: bool;

  while (true)
    invariant {:layer 1}{:yields} true;
  {
    call s := CAS_b(false, true);
    if (s) {
      break;
    }
  }
}
```

```
var {:layer 1, 2} l: Option Tid;

procedure {:right} {:layer 2, 2}
AcquireSpec(tid: Tid)
modifies l;
{
  assume l == None();
  l := Some(tid);
}

procedure {:left} {:layer 2, 2}
ReleaseSpec(tid: Tid)
modifies l;
{
  assert l == Some(tid);
  l := None();
}
```

```
var {:layer 0, 1} b: bool;

procedure {:yields} {:layer 1}
  {:refines "AcquireSpec"}
Acquire({:layer 1} tid: Tid)
{
  var s: bool;

  while (true)
    invariant {:layer 1}{:yields} true;
  {
    call s := CAS_b(false, true);
    if (s) {
      call set_l(Some(tid));
      break;
    }
  }
}
```

Introduction action

Introduction action

```
procedure {:intro} {:layer 1}
set_l(v: Option Tid)
modifies l;
{ l := v; }
```

```
var {:layer 1, 2} l: Option Tid;

procedure {:right} {:layer 2, 2}
AcquireSpec(tid: Tid)
modifies l;
{
  assume l == None();
  l := Some(tid);
}

procedure {:left} {:layer 2, 2}
ReleaseSpec(tid: Tid)
modifies l;
{
  assert l == Some(tid);
  l := None();
}
```

```
var {:layer 0, 1} b: bool;

procedure {:yields} {:layer 1}
  {:r                uireSpec"}
  {:yield_preserves "LockInv"}
Acquire({:layer 1} tid: Tid)
{
  var s: bool;

  while (true)
    invariar                  :yields}
      {:yield_loop "LockInv"} true;
  {
    call s := CAS_b(false, true);
    if (s) {
      call set_l(Some(tid));
      break;
    }
  }
}
```

Yield invariant

Yield invariant

```
procedure {:intro} {:layer 1}
set_l(v: Option Tid)
modifies l;
{ l := v; }

procedure {:yield_invariant} {:layer 1}
LockInv();
requires b <==> (l != None());
```

```
var {:layer 1, 2} l: Option Tid;

procedure {:right} {:layer 1, 2}
AcquireSpec({:linear "tid"} tid: Tid)
modifies l;
{
  assume l == None();
  l := Some(tid);
}

procedure {:left} {:layer 1, 2}
ReleaseSpec({:linear "tid"} tid: Tid)
modifies l;
{
  assert l == Some(tid);
  l := None();
}
```

Linear variable

Linear variable

```
var {:layer 0, 1} b: bool;

procedure {:yields} {:layer 1}
  {:refines "AcquireSpec"}
  {:yield_preserves "tid"}
Acquire({:layer 1}{:linear "tid"} tid: Tid)
{
  var s: bool;

  while (true)
    invariant {:layer 1}{:yields}
      {:yield_loop "LockInv"} true;
  {
    call s := CAS_b(false, true);
    if (s) {
      call set_l(Some(tid));
      break;
    }
  }
}
```

Linear variable

```
procedure {:intro} {:layer 1}
set_l(v: Option Tid)
modifies l;
{ l := v; }

procedure {:yield_invariant} {:layer 1}
LockInv();
requires b <==> (l != None());
```

```
var {:layer 1, 2} l: Option Tid;

procedure {:right} {:layer 2, 2}
AcquireSpec({:linear "tid"} tid: Tid)
modifies l;
{
  assume l == None();
  l := Some(tid);
}

procedure {:left} {:layer 2, 2}
ReleaseSpec({:linear "tid"} tid: Tid)
modifies l;
{
  assert l == Some(tid);
  l := None();
}
```

```
var {:layer 0, 1} b: bool;

procedure {:yields} {:layer 1}
  {:refines "AcquireSpec"}
  {:yield_preserves "LockInv"}
Acquire({:layer 1}{:linear "tid"} tid: Tid)
{
  var s: bool;

  while (true)
    invariant {:layer 1}{:yields}
      {:yield_loop "LockInv"} true;
  {
    call s := CAS_b(false, true);
    if (s) {
      call set_l(Some(tid));
      break;
    }
  }
}
```

```
procedure {:intro} {:layer 1}
set_l(v: Option Tid)
modifies l;
{ l := v; }

procedure {:yield_invariant} {:layer 1}
LockInv();
requires b <==> (l != None());

procedure {:yields} {:layer 2}
  {:refines "ClientSpec"}
  {:yield_preserves "LockInv"}
Client({:layer 1,2} {:hide}
       {:linear "tid"} tid: Tid)
{
  call Acquire(tid);
  ...
  call Release(tid);
}
```

Two major case studies

Concurrent garbage collector

6 Actions to allocate, read/write fields, check equality

5 Atomic root scan

4 “Gray objects” stack ⇒ set

3 Collector-mutator coordination for phase changes

2 Actions used in barrier synchronization

1 Locks and read/write access

0 Atomic CPU operations

2031 LOC

Paxos

High-level specification
Consensus in a single atomic step

Atomic event handlers
abstract protocol state

Low-level implementation
network, memory, …

1116 LOC

Experience

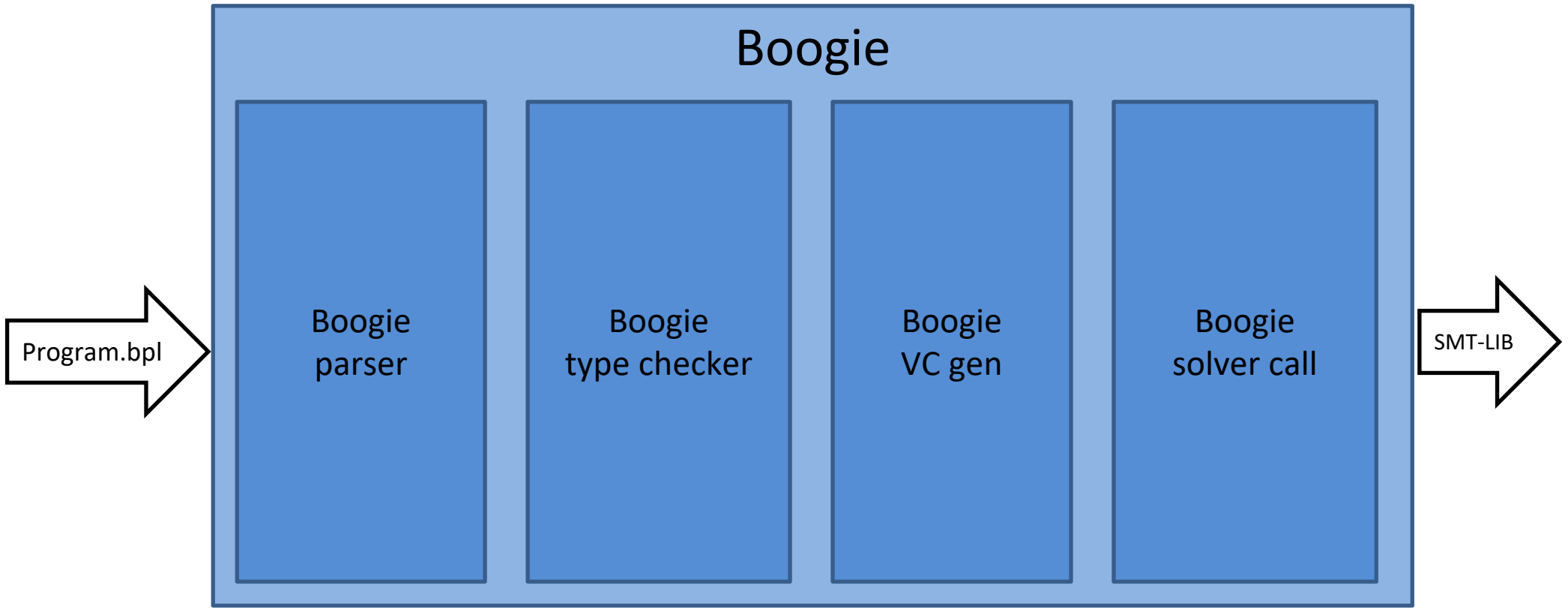
Ours

- Concurrent garbage collector [Hawblitzel et. al; CAV'15]
- Paxos [Kragl et. al; PLDI'20]
- Two-phase commit [Kragl et. al; CONCUR'18]
- Lock-protected memory atop TSO, thread-local heap atop shared heap, work-stealing queue, Treiber stack, ticket lock, …

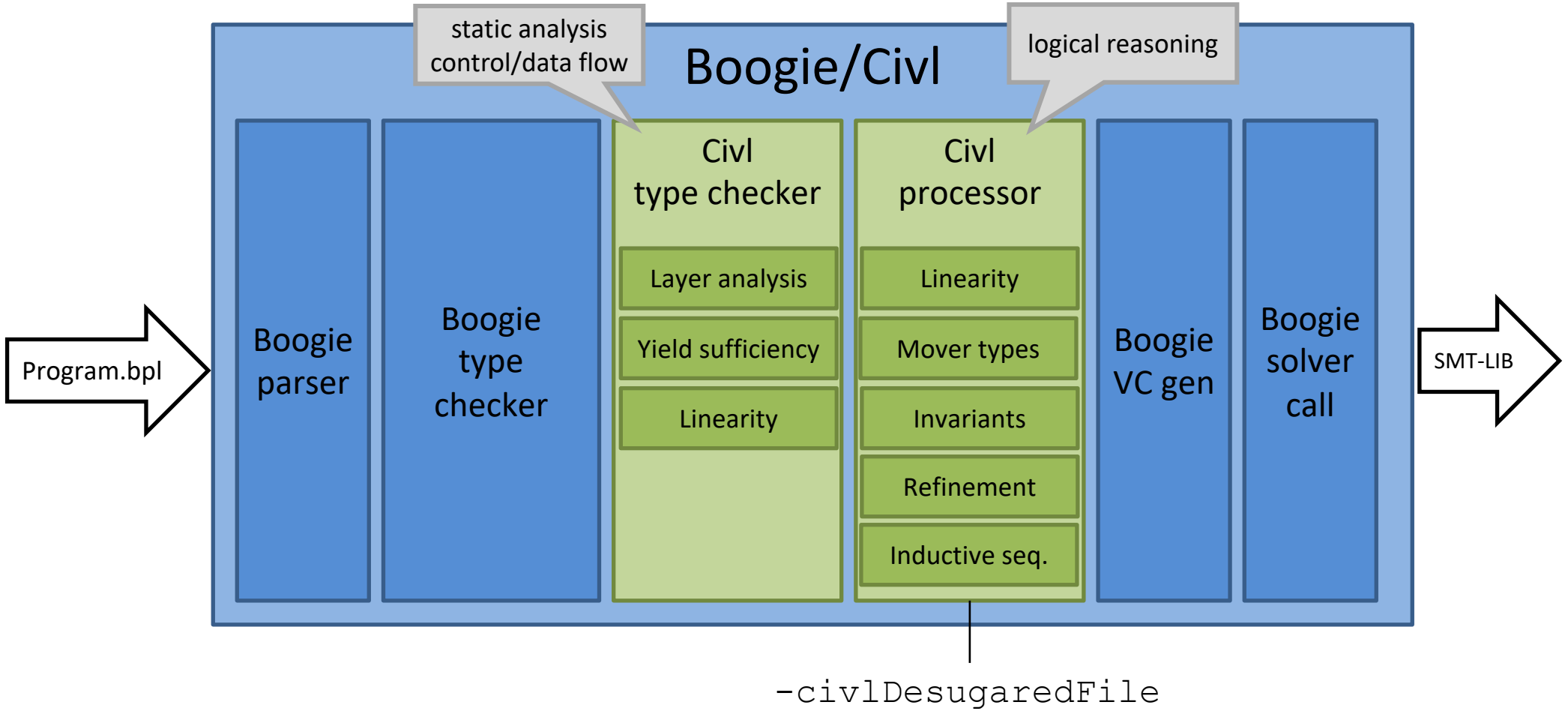
Others

- VerifiedFT (FastTrack data-race detector) [Flanagan et. al; PPoPP'18]
- Chase-Lev deque [Mutluergil & Tasiran; Computing '19]
- Java weakly-consistent data structures [Krishna et. al; ESOP'20]
- Weak memory (TSO) programs [Bouajjani, et. al; CAV'18]

Implementation



Implementation



Give Civl a try!

https://civl-verifier.github.io

Source code
Documentation
Examples
Papers
Talks
…