Refinement for Structured Concurrent Programs

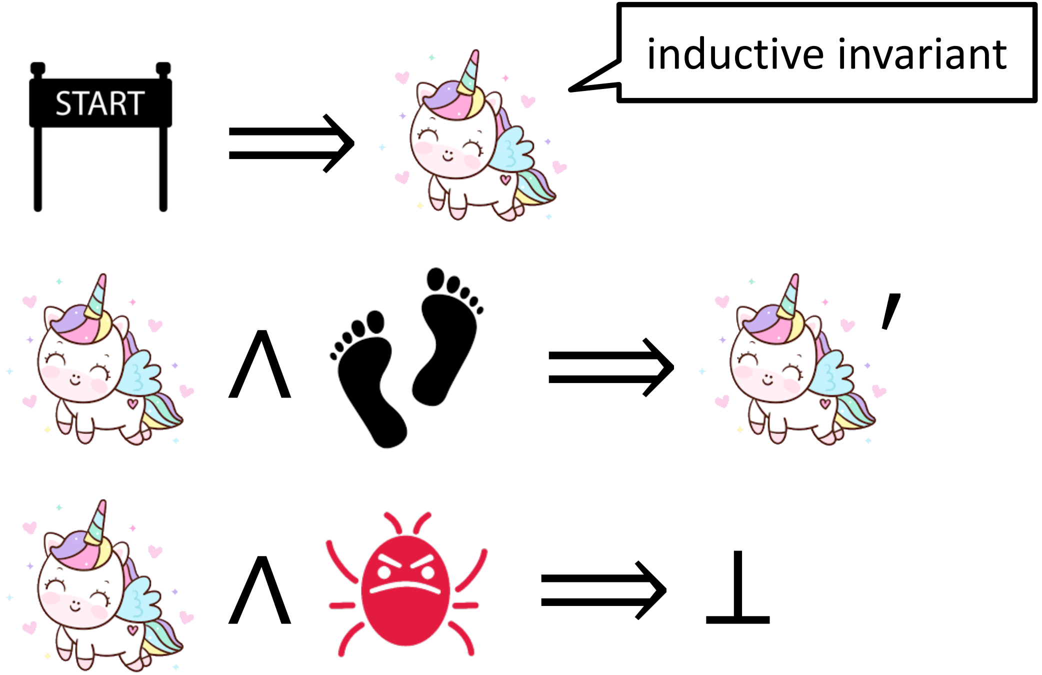
Bernhard Kragl
IST Austria

Shaz Qadeer
Novi

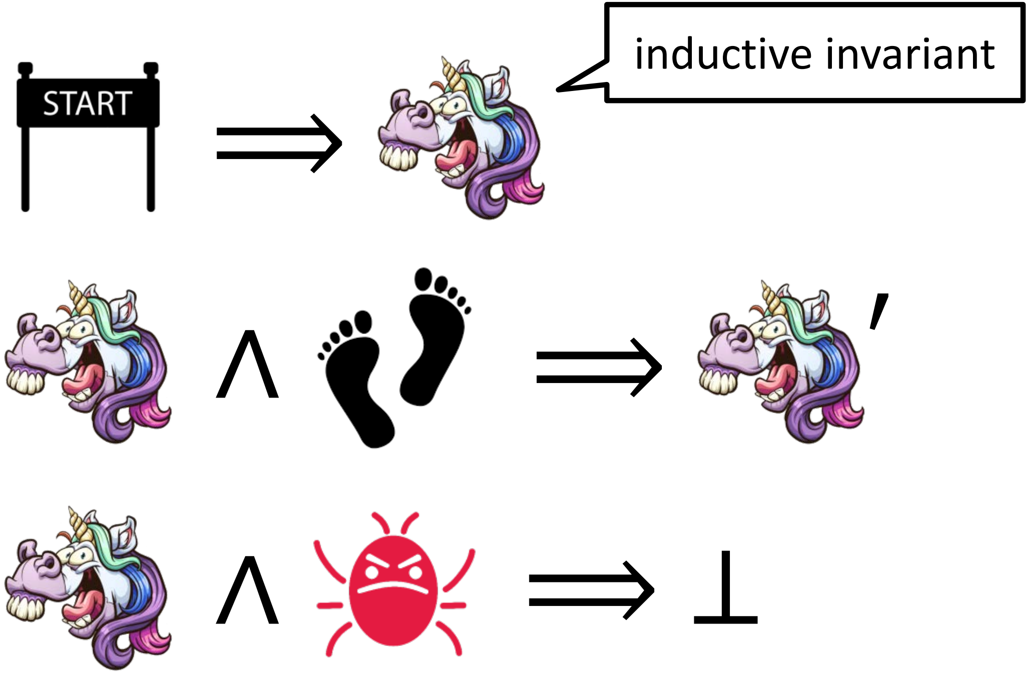
Thomas A. Henzinger
IST Austria

Deductive Safety Verification

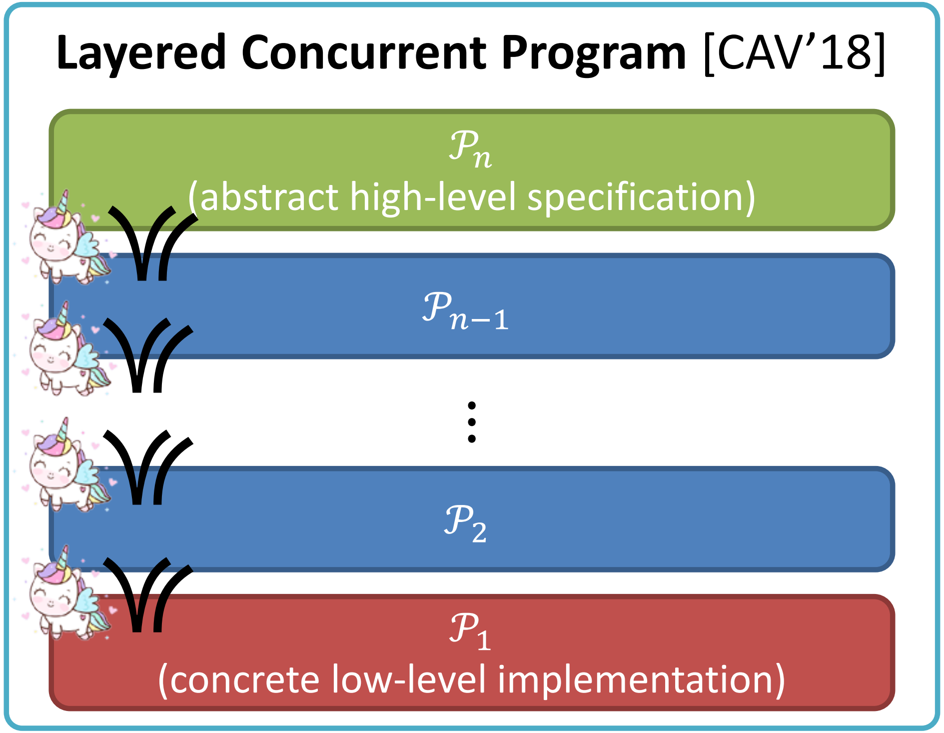
The Invariant Challenge



The Invariant Challenge



CIVL: Layered Refinement over Structured Concurrent Programs



Procedures

proc P(…) { S }

S_1 ; S_2 **if** * **then** S_1 **else** S_2 **exec** A

call P **call** P_1 || P_2 **async** P …

Gated atomic actions

CAS_x(old, new)
returns (s)
 s := (x = old)
 if s **then** x := new

RELEASE(tid)
 assert l = Some(tid)
 l := None

Contributions

General Refinement Proof Rule

elimination of special cases

Yield Invariants

named, parameterized, interference-free invariants

Linear Permission System

disjointness invariants “for free”

```
var x: int

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

hidden local

---

introduced local

introduced local

```
var x: int
var l: Option<Tid>

proc Inc(linear tid)
  exec ACQUIRE(tid)
  exec t := READ(tid)
  exec WRITE(tid, t+1)
  exec RELEASE(tid)

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

action RELEASE(linear tid)
  assert l = Some(tid)
  l := None

action READ(linear tid)
returns (v)
  assert l = Some(tid)
  v := x

action WRITE(linear tid, v)
  assert l = Some(tid)
  x := v
```

introduced global

hidden global

rewritten calls

right mover

left mover

both mover

both mover

```
var x: int
var b: bool

proc Inc()
  call Acquire()
  call t := Read()
  call Write(t+1)
  call Release()

proc Acquire()
  exec s := CAS_b(false, true)
  if (¬s) then call Acquire()

proc Release()
  exec [b := false]

proc Read()
returns (v)
  exec [v := x]

proc Write(v)
  exec [x := v]
```

hidden global

procedure call

Modular Refinement Checking

Challenge 1: Matching States

```
proc Acquire()
    exec s := CAS_b(false, true)
    if (¬s)
        call Acquire()
```

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

Modular Refinement Checking

Challenge 1: Matching States

proc Acquire(*linear* tid)
  exec s := CAS_b(false, true)
  if (¬s)
    call Acquire()
  else
    [l := Some(tid)]

introduction action

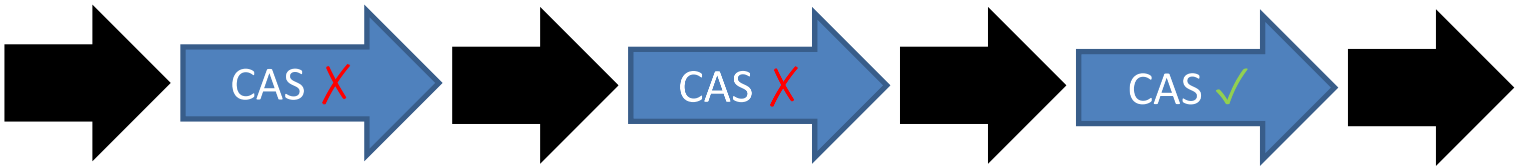
action ACQUIRE(*linear* tid)
  assume l = None
  l := Some(tid)

Modular Refinement Checking

Challenge 2: Matching Executions

```
proc Acquire(linear tid)
    exec s := CAS_b(false, true)
    if (¬s)
        call Acquire()
    else
        [l := Some(tid)]
```

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

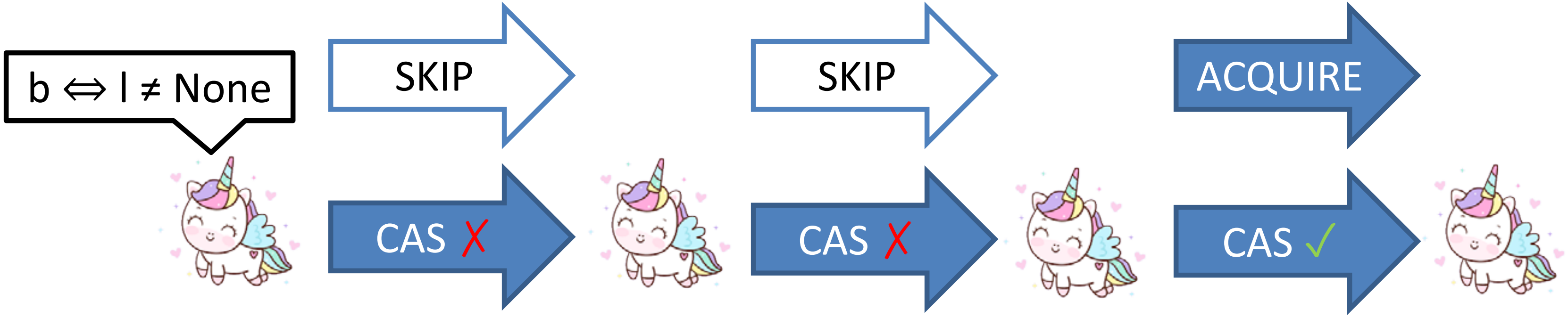


Modular Refinement Checking

Challenge 2: Matching Executions

proc Acquire(*linear* tid)
  exec s := CAS_b(false, true)
  if (¬s)
    call Acquire()
  else
    [l := Some(tid)]

action ACQUIRE(*linear* tid)
  assume l = None
  l := Some(tid)



Yield Invariants

named parameterized

invariant yield_x(i: int)

$x \geq i$

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

assert $x_0 \geq x_0$ // before call

// yield at entry
$x_1 \geq x_0$
$x_2 = x_1 + 1$
assert $x_2 \geq x_2$
// yield between increments
$x_3 \geq x_2$
$x_4 = x_3 + 1$
assert $x_4 \geq x_0 + 2$
// yield at exit

$x_5 \geq x_0 + 2$ // after return

Noninterference & Linearity

```
var barrierSet: Set<Tid>

action EnterBarrier(i: Tid)
  …
  barrierSet := barrierSet + {i}

action ExitBarrier(i: Tid)
  assert i ∈ barrierSet
  …
  barrierSet := barrierSet - {i}
```

```
procedure Mutator(i: Tid)
  …
  exec EnterBarrier(i)
  call MutatorInv(i)
  exec ExitBarrier(i)
  // access memory here
```

```
invariant MutatorInv(i: Tid)
  i ∈ barrierSet
```

{MutatorInv(i) ∧ MutatorInv(j)}
ExitBarrier(i)
{MutatorInv(j)} ✗

Noninterference & Linearity

```
var barrierSet: Set<Tid>

action EnterBarrier(i: Tid)
    ...
    barrierSet := barrierSet + {i}

action ExitBarrier(i: Tid)
    assert i ∈ barrierSet
    ...
    barrierSet := barrierSet - {i}
```

```
procedure Mutator(linear i: Tid)
    ...
    exec EnterBarrier(i)
    call MutatorInv(i)
    exec ExitBarrier(i)
    // access memory here

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

{MutatorInv(i) ∧ MutatorInv(j) ∧ i ≠ j}
ExitBarrier(i)
{MutatorInv(j)} ✓

Noninterference & Linearity

{Left(i) | i ∈ barrierSet}

```
var linear barrierSet: Set<Tid>

action EnterBarrier(linear_in i: Tid)
returns (linear_out p: Perm)
    …
    barrierSet := barrierSet + {i}
    p := Right(i)

action ExitBarrier
(linear_in p: Perm, linear_out i: Tid)
    assert p = Right(i) ∧ i ∈ barrierSet
    …
    barrierSet := barrierSet - {i}
```

{Left(i), Right(i)}

```
procedure Mutator(linear i: Tid)
    …
    exec p := EnterBarrier(i)
    call MutatorInv(p, i)
    exec ExitBarrier(p, i)
    // access memory here
```

{Right(i)}

```
invariant MutatorInv(linear p: Perm, i: Tid)
    p = Right(i) ∧ i ∈ barrierSet
```

{MutatorInv(p, i) ∧ MutatorInv(q, j) ∧ p ≠ q}
ExitBarrier(p, i)
{MutatorInv(q, j)} ✓

Benefits of Yield Invariants

Ported 30 existing examples to yield invariants

Proof Simplification

Reuse factor of up to 13

Performance

Artificial parametric example: $n^2 \rightarrow n$

VerifiedFT Race Detector: 10 sec → 5 sec

Garbage Collector: 60 sec → 10 sec

The CIVL Verifier

Extension of Boogie

github.com/boogie-org/boogie

Layered concurrent program →
Sequential Boogie program →
SMT verification conditions

Examples

• Garbage collector [Hawblitzel et. al; CAV'15] • VerifiedFT [Flanagan et. al; PPoPP'18] • Weak memory (TSO) programs [Bouajjani, et. al; CAV'18] • Chase-Lev deque [Mutluergil & Tasiran; Computing '19] • Weakly-consistent data structures [Krishna et. al; ESOP'20] • Two-phase commit [Kragl et. al; CONCUR'18] • Paxos [Kragl et. al; PLDI'20]

Related Work

Refinement

TLA+, Event-B,

CertiKOS/CCAL, CSPEC, IronFleet, Armada, …

Concurrency Verifiers

Chalice, VCC, VeriFast, VerCors, Viper, Verdi, …

Software Model Checking

Blast, Threader, Weaver, …