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 $\mathcal{P}$

Verifier
Is $\mathcal{P}$ safe?

Error report

Proof
Program verification

- Program \( \mathcal{P} \)
- Invariants
  - Templates
  - Proof structure
- Verifier
  - Is \( \mathcal{P} \) safe?
- Error report
- Proof
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

Init: $pc = pc_1 = pc_2 = a$

Next:

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

$pc_1 = b \land pc_1' = c \land \overline{l} \land l' \land eq(pc, pc_2, x, t_1, t_2)$

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

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

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

$pc_1 = f \land pc_1' = g \land \overline{l'} \land eq(pc, pc_2, x, t_1, t_2)$

$pc_2 = b \land pc_2' = c \land \overline{l} \land l' \land eq(pc, pc_1, x, t_1, t_2)$

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

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

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

$pc_2 = f \land pc_2' = g \land \overline{l'} \land eq(pc, pc_1, x, t_1, t_2)$

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

Safe: $pc = g \Rightarrow x = 2$
Interference freedom and Owicki-Gries

\[ \Psi_1 : \{ P_1 \} C_1 \{ Q_1 \} \quad \Psi_2 : \{ P_2 \} C_2 \{ Q_2 \} \quad \Psi_1, \Psi_2 \text{ interference free} \]

\[ \{ P_1 \land P_2 \} C_1 \parallel C_2 \{ Q_1 \land Q_2 \} \]

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

\[
\begin{array}{c|c|c}
\hline
{ x = 0 } & \{ x = 0 \} & \text{Interference freedom:} \\
{ x = 0 } & \{ x = 0 \} & \{ P_1 \land P_2 \} x := x + 2 \{ P_1 \} \\
{ x = 1 \lor x = 3 } & { x = 1 \lor x = 1 } & \{ P_2 \land P_1 \} x := x + 1 \{ P_2 \} \\
{ x := x + 1 } & { x := x + 2 } & \{ Q_1 \land P_2 \} x := x + 2 \{ Q_1 \} \\
{ Q_1 : x = 1 \lor x = 3 } & { x = 2 \lor x = 3 } & \{ Q_2 \land P_1 \} x := x + 1 \{ Q_2 \} \\
\{ Q_1 \land Q_2 \} & \{ x = 3 \} & \\
\{ x = 3 \} & \\
\hline
\end{array}
\]
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\}

\[
\begin{align*}
\{x = 0\} \\
[\text{done}_1 := \text{false}; \text{done}_2 := \text{false}] \\

P_1: \{\neg \text{done}_1 \land (\neg \text{done}_2 \Rightarrow x = 0) \land (\text{done}_2 \Rightarrow x = 1)\} \\
[\text{x := x + 1}; \text{done}_1 := \text{true}] \\
Q_1: \{\text{done}_1 \land (\neg \text{done}_2 \Rightarrow x = 1) \land (\text{done}_2 \Rightarrow x = 2)\}
\end{align*}
\]

\[
\begin{align*}
\{x = 2\} \\

P_2: \{\neg \text{done}_2 \land (\neg \text{done}_1 \Rightarrow x = 0) \land (\text{done}_1 \Rightarrow x = 1)\} \\
[\text{x := x + 1}; \text{done}_2 := \text{true}] \\
Q_2: \{\text{done}_2 \land (\neg \text{done}_1 \Rightarrow x = 1) \land (\text{done}_1 \Rightarrow x = 2)\}
\end{align*}
\]
Rely/Guarantee

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

\[
\begin{align*}
\{x \geq 0\} \\
x := x + 1 \parallel x := x + 1 & \quad P = Q = (x \geq 0) \\
\{x \geq 0\} \\
R = G = (x' \geq x)
\end{align*}
\]
Multi-layered refinement proofs

\[ P_1 \Downarrow P_2 \Downarrow \cdots \Downarrow P_{n-1} \Downarrow P_n \]

\begin{align*}
\text{\textbf{\( P_n \) is safe}} \quad &\quad \text{\textbf{\( P_1 \) is safe}} \quad \text{[skip]} \\
\mid \mid & \mid \mid \\
\end{align*}

Advantages of structured proofs:

Better for humans: easier to construct and maintain
Better for computers: localized/small checks \(\rightarrow\) 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]

\[\text{(Gate, Transition)}\]

single-state predicate
two-state predicate

<table>
<thead>
<tr>
<th>Command</th>
<th>Gate</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := x+y</td>
<td>true</td>
<td>(x' = x + y \land y' = y)</td>
</tr>
<tr>
<td>havoc x</td>
<td>true</td>
<td>(y' = y)</td>
</tr>
<tr>
<td>assert x&lt;y</td>
<td>x &lt; y</td>
<td>(x' = x \land y' = y)</td>
</tr>
<tr>
<td>assume x&lt;y</td>
<td>true</td>
<td>(x &lt; y \land x' = x \land y' = y)</td>
</tr>
</tbody>
</table>

Lock specification

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

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

• Program configuration \((g, \{(\ell, s) \cdot \vec{f}\} \cup \mathcal{T})\)
• Transition relation \(\Rightarrow\) between configurations (and failure configuration \(\bot\))
• Safety: \(\neg \exists g\ell: (g, (\ell, \text{Main})) \Rightarrow^* \bot\)

• \(\text{Good}(P) = \{g \mid \neg \exists \ell: (g, (\ell, \text{Main})) \Rightarrow^* \bot\}\)
• \(\text{Trans}(P) = \{(g, g') \mid \exists \ell: (g, (\ell, \text{Main})) \Rightarrow^* (g', \emptyset)\}\)

• \(P_1 \preceq P_2:\)  
  1. \(\text{Good}(P_2) \subseteq \text{Good}(P_1)\)  
  2. \(\text{Good}(P_2) \circ \text{Trans}(P_1) \subseteq \text{Trans}(P_2)\)

  \(P_2\) preserves failures  \hspace{1cm}  \(P_2\) preserves final states
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()

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() {
}

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();
}
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”
\[ P(a) = \begin{cases} \text{assume } a > 0; & a := a - 1 \end{cases} \]
right mover (R)

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

Sequence \( R^*;(N+\varepsilon); L^* \) is atomic
Atomic actions can fail

var \( x \) : int, lock : ThreadID \( \cup \) \{nil\}

Acquire(): \[\text{assume lock} == \text{nil}; \ lock := \text{tid}\]
Release(): \[\text{assert lock} == \text{tid}; \ lock := \text{nil}\]
Read(out \( r \)): \[\text{assert lock} == \text{tid}; \ r := x\]
Write(v): \[\text{assert lock} == \text{tid}; \ x := v\]

Commutativity:
\[
\begin{align*}
R \ X & \rightarrow \ X \ R & \quad X \ L & \rightarrow \ L \ X \\
R \ X \perp & \rightarrow \ X \perp & \quad X \ L \perp & \rightarrow \ X \perp \\
X \perp & \rightarrow \ L \ X \perp
\end{align*}
\]

Forward preservation:
\[
\begin{align*}
R \ X \perp & \rightarrow \ X \perp & \quad X \ L \perp & \rightarrow \ X \perp \\
X \perp & \rightarrow \ L \ X \perp
\end{align*}
\]

Backward preservation:
\[
X \perp \rightarrow \ L \ X \perp
\]
Nonblocking and Cooperation

Left movers must be nonblocking

Termination? Too strong.

Cooperation: always possible to terminate
Mover types in CIVL

1. Atomic actions are annotated with mover types

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

\[
\forall S_1 S_2 S_3 \exists S_2': G_1(S_1) \land G_2(S_1) \land T_1(S_1, S_2) \land T_2(S_2, S_3) \Rightarrow T_2(S_1, S_2') \land T_1(S_2', S_3)
\]

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

\[
\forall S S': G_1(S) \land G_2(S) \land T_1(S, S') \Rightarrow G_2(S')
\]

Backward preservation \((G_2, T_2)\) is L

\[
\forall S S': G_2(S) \land T_2(S, S') \land G_1(S') \Rightarrow G_1(S)
\]

2. Induced logical commutativity conditions

3. Atomization of composite statements
 Atomization ($S \rightarrow [S]$)

- 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 U {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 ()

1. Acquire()
2. Read(out t)
3. Write(t + 1)
4. Release()
Example: Atomizing nonatomic increment

```plaintext
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
    1  Acquire()
    2  Read(out t)
    3  Write(t + 1)
    4  Release()
    5
```

Simulation computation [Henzinger, Henzinger, Kopke 1995]
Example: Resizing an array

```
proc DoubleSize ()
    var t, B, v
1    Acquire()
2    GetLen(out t)
3    B := Allocate(2*t)
4    i := 0
5    while (i < t)
6        Read(i, out v)
7        B[i] := v
8        i := i + 1
9    Switch(B)
10   SetLen(2*t)
11   Release()
```

```
var A : Array, len : Nat, lock : ThreadID U {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

```
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

proc DoubleSize ()
  var t, B, v
  1 Acquire()
  2 GetLen(out t)
  3 B := Allocate(2*t)
  4 SetLen(2*t)
  5 i := 0
  6 while (i < t)
  7   Read(i, out v)
  8   B[i] := v
  9   i := i + 1
 10 Switch(B)
 11 Release()
```
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 $\rightarrow$ 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 $[\text{assert } 0 \leq g; \ g := g + 1]$

$[g := g + 1]$ refines $[\text{var } g_\_ = g; \ \text{havoc } g; \ \text{assume } g_\_ \leq g]$

$[g := h]$ refines $/* \ 0 \leq h */ [\text{havoc } g; \ \text{assume } 0 \leq g]$
Atomization, summarization, and abstraction are symbiotic [Elmas, Q, Tasiran 2009]
Abstraction enables stronger mover types

Read and Write are conflicting (non-movers)

\[
\text{action Read}(\text{out } r):
\begin{align*}
  r & := x \\
\end{align*}
\]

\[
\text{action Write}(v):
\begin{align*}
  x & := v \\
\end{align*}
\]

Strengthening the gates satisfies commutativity

Inc is blocking

\[
\text{action Inc}():
\begin{align*}
  \text{assume lock} & = \text{tid} \\
  x & := x + 1 \\
\end{align*}
\]

Weakening the transition makes Inc nonblocking
Example: Ticket lock

```
var back
deb 

var front

det
var ticket
[

[ ticket := back; back := back + 1 ]
[ assume ticket == front ]
]

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

Lock held by: t1

Lock held by: t2

Lock held by: t3

front and back can get arbitrarily far apart

var back

var front

Acquire() {

var ticket
[

[ ticket := back; back := back + 1 ]
[ assume ticket == front ]
]
```
Example: Ticket lock

```plaintext
var back
var front

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

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

If we could treat
Acquire as atomic ...
Example: Ticket lock

```
var back
def front

Release() {
  front := front + 1
}

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

Lock head by: t1
Lock head by: t2
Lock available

Example: Ticket lock

front and back operate in “lockstep”
0 ≤ b − f ≤ 1
Acquire() {
    var ticket
    ticket := back; back := back + 1
    assume ticket == front
}

var back
var front

Invariant: T = (-∞, back)

Release() {
    front := front + 1
}

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

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

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

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

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

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

Release() {
    [ front := front + 1 ]
}
var lock
Acquire() {
    [ assume lock == nil;
    lock := tid ]
}
Release() {
    [ assert lock == tid;
    lock := nil ]
}
Invariant: if lock == nil then T = (-∞, front) else T = (-∞, front)

def Ticket lock has the same abstract spec as spinlock

var T
var front
Acquire() {
    [ assume !T[front]; T[front] := true ]
}
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 ]
}
Local reasoning is challenging

Read\((\text{out } r)\): \[\text{assert lock }= \text{tid}; r := x\]
Write\((v)\): \[\text{assert lock }= \text{tid}; x := v\]

\[
\text{lock }= \text{tid}_1 \land \text{lock }= \text{tid}_2 \models [r := x; x := v] \Rightarrow [x := v; r := x]
\]

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

\[
\begin{align*}
\text{proc } P (\text{lin } p) & \quad \text{proc } P (\text{lin}_\text{in } p) & \quad \text{proc } P (\text{lin}_\text{out } p) \\
// x \text{ available} & \quad // x \text{ available} & \quad // x \text{ available} \\
// y \text{ unavailable} & \quad // x \text{ available} & \quad // x \text{ unavailable} \\
y := x & \quad \text{call } P(x) & \quad \text{call } P(x) \\
// x \text{ unavailable} & \quad // x \text{ available} & \quad \text{call } P(x) \\
// y \text{ available} & \quad // x \text{ available} & \quad // x \text{ available}
\end{align*}
\]

3. \( \Gamma : \text{Value} \rightarrow 2^\mathbb{N} \) \quad e.g.: \( \Gamma(\text{tid}) = \{\text{tid}\} \) \quad \( \Gamma(\text{tidSet}) = \text{tidSet} \)

4. \( \text{Collect}(c) = \left( \bigcup_{x \in \text{Lin} \cap \text{Glob}} \Gamma(g(x)) \right) \cup \left( \bigcup_{(x, \ell) \in \text{Available}(c)} \Gamma(\ell(x)) \right) \)

5. Invariant: \( \text{Collect}(c) \) is a set
Exploiting the free assumption

Read(\textit{linear tid, out r}): [assert lock == tid; r := x]
Write(\textit{linear tid, v}): [assert lock == tid; x := v]

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

\[
\text{IsSet}({\text{tid1}} \cup \{\text{tid2}\}) \land \text{lock} = \text{tid1} \land \text{lock} = \text{tid2} \models [r := x; x := v] \Rightarrow [x := v; r := x]
\]

simplifies to
\[
\text{tid1} \neq \text{tid2}
\]
Atomic actions must preserve invariant

\[
\Gamma(\text{slots'}) \cup \Gamma(\text{tid'}) \subseteq \Gamma(\text{slots})
\]
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

- $\mathcal{P}_1, ..., \mathcal{P}_{h+1}$ are concurrent programs
  - $\mathcal{P}_i$ refines $\mathcal{P}_{i+1}$ for all $i \in [1, h]$

- $\mathcal{C}_1, ..., \mathcal{C}_h$ 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 $\mathcal{P}_1, ..., \mathcal{P}_{h+1}$ and the key insight of $\mathcal{C}_1, ..., \mathcal{C}_h$ in a single layered concurrent program $\mathcal{LP}$
- Generate $\mathcal{C}_1, ..., \mathcal{C}_h$ automatically from $\mathcal{LP}$
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
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)
  icall 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

```plaintext
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)
```

```plaintext
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
```

```plaintext
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

\[
\begin{align*}
\text{var} & \quad b@[0,1] : \text{bool} \\
\text{var} & \quad \text{lock}@[1,2] : \text{nat} \\
\text{var} & \quad \text{linear slots}@[1,2] : \text{set<nat>} \\
\text{var} & \quad \text{pos}@[1,1] : \text{nat} \\
\end{align*}
\]

call Main()

\begin{itemize}
\item \textbf{proc} Main@2()
  \textbf{refines} SKIP
  \begin{itemize}
  \item while (\text{\ast})
  \item async call Worker()
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{left proc} Worker@2()
  \textbf{refines} SKIP
  \begin{itemize}
  \item call \text{Enter}(\text{tid})
  \item call \text{Leave}(\text{tid})
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{right proc} Enter@1(\text{linear tid}@1 : \text{nat})
  \textbf{refines} \text{ACQUIRE}
  \begin{itemize}
  \item \textbf{var} success@0 : \text{bool}
  \item while (true)
  \item call success := \text{CAS}()
  \item if (success)
  \item \text{icall iSetLock}(\text{tid})
  \item break
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{left proc} Leave@1(\text{linear tid}@1 : \text{nat})
  \textbf{refines} \text{RELEASE}
  \begin{itemize}
  \item call \text{RESET}()
  \item \text{icall iSetLock}(\text{nil})
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{iaction} iSetLock@1(v : \text{nat})
  \text{lock} := v
\end{itemize}
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 | iIncr@1() (linear tid : nat) |
| assert | InvAlloc        |
| 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

```plaintext
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)
```

```plaintext
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
```

```plaintext
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
```

```plaintext
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
```
A chain of concurrent programs

\[ \mathcal{P}_1 \rightarrow \tilde{\mathcal{P}}_1 \rightarrow \mathcal{P}_2 \rightarrow \cdots \rightarrow \mathcal{P}_i \rightarrow \tilde{\mathcal{P}}_i \rightarrow \mathcal{P}_{i+1} \rightarrow \cdots \rightarrow \mathcal{P}_h \rightarrow \tilde{\mathcal{P}}_h \rightarrow \mathcal{P}_{h+1} \]

- \( \mathcal{P}_1, \ldots, \mathcal{P}_{h+1} \) 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

```plaintext
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

A and skip are disjoint

\[
\begin{align*}
\text{pc}_0 &= \text{false} \\
\text{assert } g_i &\neq g_{i+1} \Rightarrow \neg \text{pc}_i \land A(g_i, g_{i+1}) \\
\text{pc}_{i+1} &= \text{pc}_i \lor g_i \neq g_{i+1} \\
\text{assert } \text{pc}_n
\end{align*}
\]

In general

\[
\begin{align*}
\text{pc}_0 &= \text{false} \\
\text{assert } g_i &\neq g_{i+1} \Rightarrow \neg \text{pc}_i \land A(g_i, g_{i+1}) \\
\text{pc}_{i+1} &= \text{pc}_i \lor g_i \neq g_{i+1} \\
\text{done}_0 &= \text{false} \\
\text{done}_{i+1} &= \text{done}_i \lor A(g_i, g_{i+1}) \\
\text{assert } \text{done}_n
\end{align*}
\]
**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

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

- How do we verify concurrent checker programs $C_1, \ldots, C_h$  
  - 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

async P
if *
call P
assume false
call P1 || P2
check noninterference
if *
call P1
assume false
elsiif *
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, ..., g_m\}
LocalVar = \{l_1, ..., l_n\}

S ::= x := e | assume e | assert e | S ; S | S \boxplus 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) \cdot Trans(S1) \subseteq Trans(S2)

S is nonblocking iff
- Good(S) \subseteq \exists G’. Trans(G, G’)
Calculating Good and Trans

\begin{align*}
wp(x := e, \phi) &= \phi[x/e] \\
wp(\text{assume } e, \phi) &= e \Rightarrow \phi \\
wp(\text{assert } e, \phi) &= e \land \phi \\
wp(S_1 ; S_2, \phi) &= wp(S_1, wp(S_2, \phi)) \\
wp(S_1 \sqcup S_2, \phi) &= wp(S_1, \phi) \land wp(S_2, \phi)
\end{align*}

\begin{align*}
tr(x := e, \phi) &= \phi[x/e] \\
tr(\text{assume } e, \phi) &= e \land \phi \\
tr(\text{assert } e, \phi) &= e \land \phi \\
tr(S_1 ; S_2, \phi) &= tr(S_1, tr(S_2, \phi)) \\
tr(S_1 \sqcup S_2, \phi) &= tr(S_1, \phi) \lor tr(S_2, \phi)
\end{align*}

Good(S) = \forall l_1, \ldots, l_n. wp(S, true) \\
Trans(S) = \exists l_1, \ldots, l_n. tr(S, g_1 = g'_1 \land \ldots \land g_m = g'_m)

<table>
<thead>
<tr>
<th>S</th>
<th>Good(S)</th>
<th>Trans(S)</th>
</tr>
</thead>
</table>
| $l := g + 1$
  $g := l$ | true | $g + 1 = g'$ |
| assume $g \leq l$
  $g := l$ | true | $\exists l. g \leq l \land l = g'$ |
| assume $g \leq l$
  $g := l$
  assert $0 \leq g$ | $\forall l. g \leq l \Rightarrow 0 \leq l$ | $\exists l. g \leq l \land 0 \leq l \land l = g'$ |
Quantifiers are a problem

Is $\varphi \Rightarrow \psi$ valid?

- SMT solvers become unpredictable
- Universal quantifier in $\varphi$ is a problem
- Existential quantifier in $\psi$ 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 $\varphi$ into $\varphi_1 \lor \varphi_2$
- 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 $\varphi_1(E_1(y), y) \lor \varphi_2(E_2(y), y)$

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

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

Look for equalities in path condition:
$x = e$
$e' = A[e := x] \Rightarrow x = e'[e]$
...

...
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

\[
0 < N \quad A \subseteq [1,N] \quad B \subseteq [1,N] \quad B \subseteq A \quad |B| = N \quad |A| = N
\]