Automated Verification of Concurrent Programs
CountDownLatch Mechanism Case Study

Student: Nguyen Thanh Toan - A0154686R
Supervisor: Assoc. Prof. Chin Wei Ngan

National University of Singapore

November 3, 2017

Everything is multi-core

How to reason about concurrent programs?

Owicki-Gries approach

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 \wedge P_2\} C_1 \parallel C_2\ \{Q_1 \wedge Q_2\}}$$

if the proofs $\{P_1\}\ C_1\ \{Q_1\}$ and $\{P_2\}\ C_2\ \{Q_2\}$ are **interference free**.

Owicki-Gries approach

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 \wedge P_2\} C_1 \parallel C_2\ \{Q_1 \wedge Q_2\}}$$

if the proofs $\{P_1\}\ C_1\ \{Q_1\}$ and $\{P_2\}\ C_2\ \{Q_2\}$ are **interference free**.
Example:

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

Owicki-Gries approach

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 \wedge P_2\} C_1 \parallel C_2\ \{Q_1 \wedge Q_2\}}$$

if the proofs $\{P_1\}\ C_1\ \{Q_1\}$ and $\{P_2\}\ C_2\ \{Q_2\}$ are **interference free**.
Example:

- Program: $\{x = 0\}\ x := x + 1 \parallel x := x + 2\ \{x = 3\}$
- $P_1 : (x = 0 \vee x = 2)$ and $Q_1 : (x = 1 \vee x = 3)$
- $P_2 : (x = 0 \vee x = 1)$ and $Q_2 : (x = 2 \vee x = 3)$

Owicki-Gries approach

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 \wedge P_2\} C_1 \parallel C_2\ \{Q_1 \wedge Q_2\}}$$

if the proofs $\{P_1\}\ C_1\ \{Q_1\}$ and $\{P_2\}\ C_2\ \{Q_2\}$ are **interference free**.
Example:

- Program: $\{x = 0\}\ x := x + 1 \parallel x := x + 2\ \{x = 3\}$
- $P_1 : (x = 0 \vee x = 2)$ and $Q_1 : (x = 1 \vee x = 3)$
- $P_2 : (x = 0 \vee x = 1)$ and $Q_2 : (x = 2 \vee x = 3)$

Need to prove:

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

Owicki-Gries approach

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 \wedge P_2\} C_1 \parallel C_2\ \{Q_1 \wedge Q_2\}}$$

if the proofs $\{P_1\}\ C_1\ \{Q_1\}$ and $\{P_2\}\ C_2\ \{Q_2\}$ are **interference free**.
Example:

- Program: $\{x = 0\}\ x := x + 1 \parallel x := x + 2\ \{x = 3\}$
- $P_1 : (x = 0 \vee x = 2)$ and $Q_1 : (x = 1 \vee x = 3)$
- $P_2 : (x = 0 \vee x = 1)$ and $Q_2 : (x = 2 \vee x = 3)$

Need to prove:

- $\{P_1\}\ x := x + 1\ \{Q_1\}$
- $\{P_2\}\ x := x + 2\ \{Q_2\}$
- Interference freedom: every assertion used in the local verification is shown not invalidated by the execution of the other processes.

Interference free

- Program: $\{x = 0\}\ x := x + 1 \parallel x := x + 2\ \{x = 3\}$
- $P_1 : (x = 0 \vee x = 2)$ and $Q_1 : (x = 1 \vee x = 3)$
- $P_2 : (x = 0 \vee x = 1)$ and $Q_2 : (x = 2 \vee x = 3)$
- $\{P_1 \wedge P_2\}\ x := x + 2\ \{P_1\}$ and $\{P_2 \wedge P_1\}\ x := x + 1\ \{P_2\}$
- $\{Q_1 \wedge P_2\}\ x := x + 2\ \{Q_1\}$ and $\{Q_2 \wedge P_1\}\ x := x + 1\ \{Q_2\}$

Interference free

- Program: $\{x = 0\}\ x := x + 1 \parallel x := x + 2\ \{x = 3\}$
- $P_1 : (x = 0 \vee x = 2)$ and $Q_1 : (x = 1 \vee x = 3)$
- $P_2 : (x = 0 \vee x = 1)$ and $Q_2 : (x = 2 \vee x = 3)$
- $\{P_1 \wedge P_2\}\ x := x + 2\ \{P_1\}$ and $\{P_2 \wedge P_1\}\ x := x + 1\ \{P_2\}$
- $\{Q_1 \wedge P_2\}\ x := x + 2\ \{Q_1\}$ and $\{Q_2 \wedge P_1\}\ x := x + 1\ \{Q_2\}$
- $\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 0 \vee x = 2\}$
- $\{(x = 1 \vee x = 3) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 1 \vee x = 3\}$
- $\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 0 \vee x = 1\}$
- $\{(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 2 \vee x = 3\}$

Rely Guarantee Reasoning (P,R,G,Q)

- Precondition P and Postcondition Q.
- The rely condition R models all the atomic actions of the environment, describing the interference the program can tolerate from its environment.
- The guarantee condition G models the atomic actions of the program, and hence decribing the interference that it imposes on the other threads of the system.

Rely Guarantee Reasoning (P,R,G,Q)

- Precondition P and Postcondition Q.
- The rely condition R models all the atomic actions of the environment, describing the interference the program can tolerate from its environment.
- The guarantee condition G models the atomic actions of the program, and hence decribing the interference that it imposes on the other threads of the system.

$$\frac{C_1 \models (p_1, R_1, G_1, q_1) \qquad C_2 \models (p_2, R_2, G_2, q_2)}{C_1 \mid\mid C_2 \models (p_1 \wedge p_2, R_1 \wedge R_2, G_1 \vee G_2, q_1 \wedge q_2)}$$

Rely Guarantee Example

$$\frac{C_1 \models (p_1, R_1, G_1, q_1) \qquad C_2 \models (p_2, R_2, G_2, q_2)}{C_1 \parallel C_2 \models (p_1 \wedge p_2, R_1 \wedge R_2, G_1 \vee G_2, q_1 \wedge q_2)}$$

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

New form: $x := x + 1 \parallel x := x + 2 \quad \models (x = 0, x' = x, \mathit{True}, x' = 3)$

Rely Guarantee Example

$$\frac{C_1 \models (p_1, R_1, G_1, q_1) \qquad C_2 \models (p_2, R_2, G_2, q_2)}{C_1 \mid\mid C_2 \models (p_1 \wedge p_2, R_1 \wedge R_2, G_1 \vee G_2, q_1 \wedge q_2)}$$

Program: $\{x = 0\}\ x := x + 1 \mid\mid x := x + 2\ \{x = 3\}$

New form: $x := x + 1 \mid\mid x := x + 2 \quad \models (x = 0, x' = x, \mathit{True}, x' = 3)$

$$\begin{aligned} x := x + 1 \ \models\ & (x = 0 \vee x = 2, \\ & (x = 0 \wedge x' = 2) \vee (x = 1 \wedge x' = 3), \\ & (x = 0 \wedge x' = 1) \vee (x = 2 \wedge x' = 3), \\ & (x = 0 \wedge x' = 1) \vee (x = 2 \wedge x' = 3)) \end{aligned}$$

Rely Guarantee Example

$$\frac{C_1 \models (p_1, R_1, G_1, q_1) \qquad C_2 \models (p_2, R_2, G_2, q_2)}{C_1 \mid\mid C_2 \models (p_1 \wedge p_2, R_1 \wedge R_2, G_1 \vee G_2, q_1 \wedge q_2)}$$

Program: $\{x = 0\}\ x := x + 1 \mid\mid x := x + 2\ \{x = 3\}$

New form: $x := x + 1 \mid\mid x := x + 2 \quad \models (x = 0, x' = x, \mathit{True}, x' = 3)$

$$\begin{aligned} x := x + 1 \ \models\ & (x = 0 \vee x = 2, \\ & (x = 0 \wedge x' = 2) \vee (x = 1 \wedge x' = 3), \\ & (x = 0 \wedge x' = 1) \vee (x = 2 \wedge x' = 3), \\ & (x = 0 \wedge x' = 1) \vee (x = 2 \wedge x' = 3)) \end{aligned}$$

$$\begin{aligned} x := x + 2 \ \models\ & (x = 0 \vee x = 1, \\ & (x = 0 \wedge x' = 1) \vee (x = 2 \wedge x' = 3), \\ & (x = 0 \wedge x' = 2) \vee (x = 1 \wedge x' = 3), \\ & (x = 0 \wedge x' = 2) \vee (x = 1 \wedge x' = 3)) \end{aligned}$$

Concurrent Separation Logic - CSL

- Interfence-free concurrency

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \mid\mid C_2\ \{Q_1 * Q_2\}}$$

Concurrent Separation Logic - CSL

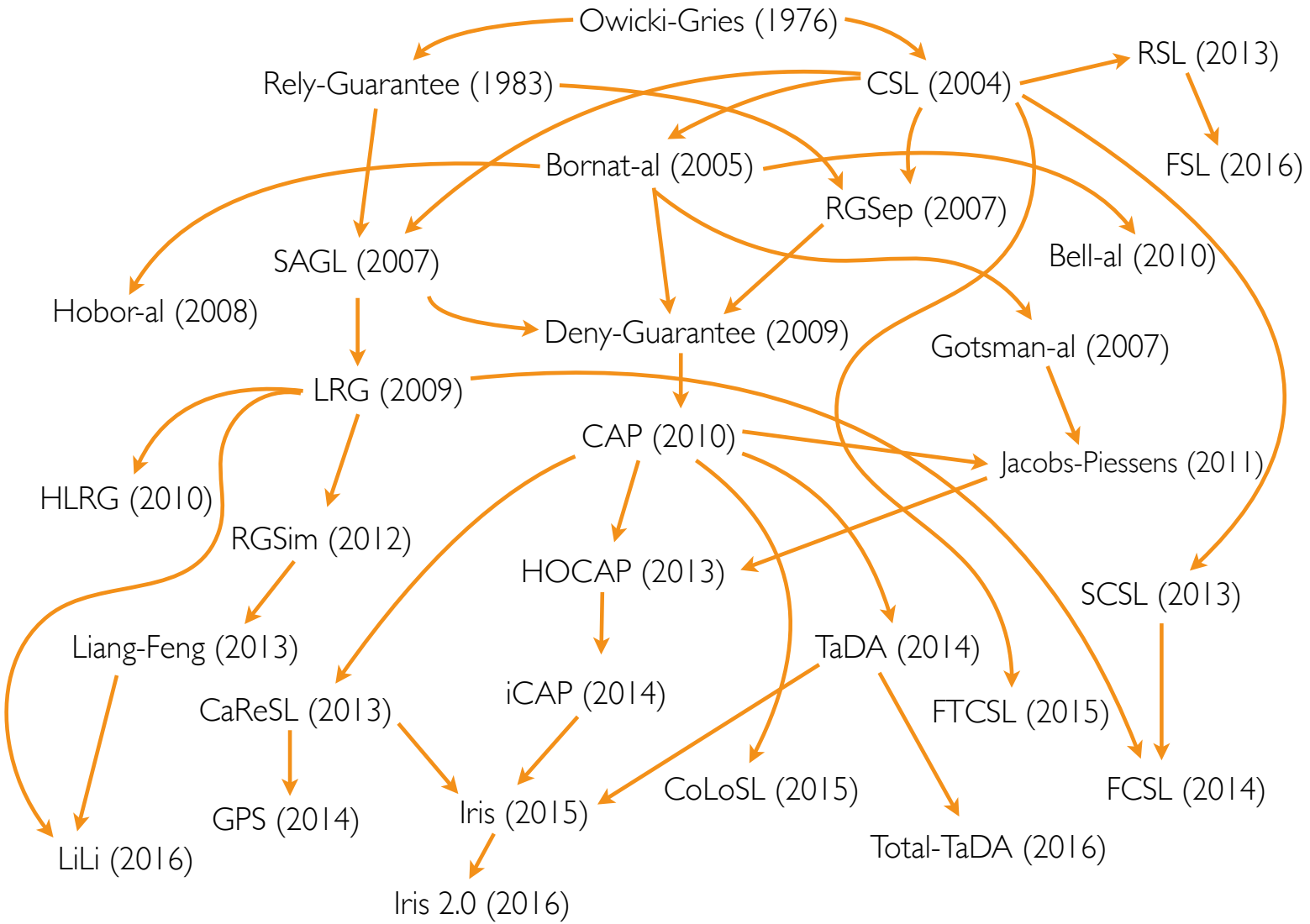
- Interfence-free concurrency

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

- A resource invariant RI_r is associated with each resource. Acquiring the resource imports the resource invariant in the local scope.

$$\frac{\{(P * RI_r) \wedge S\}\ C\ \{Q * RI_r\}}{\{P\}\ \mathit{with}\ r\ \mathit{when}\ S\ \mathit{do}\ C\ \{Q\}}$$

CSL tree


Owicki-Gries (1976)
Rely-Guarantee (1983)
CSL (2004)
RSL (2013)
FSL (2016)
Bornat-al (2005)
RGSep (2007)
Bell-al (2010)
SAGL (2007)
Hobor-al (2008)
Deny-Guarantee (2009)
Gotsman-al (2007)
LRG (2009)
CAP (2010)
Jacobs-Piessens (2011)
HLRG (2010)
RGSim (2012)
HOCAP (2013)
SCSL (2013)
Liang-Feng (2013)
TaDA (2014)
iCAP (2014)
CaReSL (2013)
FTCSL (2015)
CoLoSL (2015)
FCSL (2014)
GPS (2014)
Iris (2015)
Total-TaDA (2016)
LiLi (2016)
Iris 2.0 (2016)

Automated tools for CSL?

- Great successes of sequential separation logic tools, e.g. Infer of Facebook, and SLAyer of Microsoft.
- Only 2 CSL tools appear *this year*, Caper and Starling.

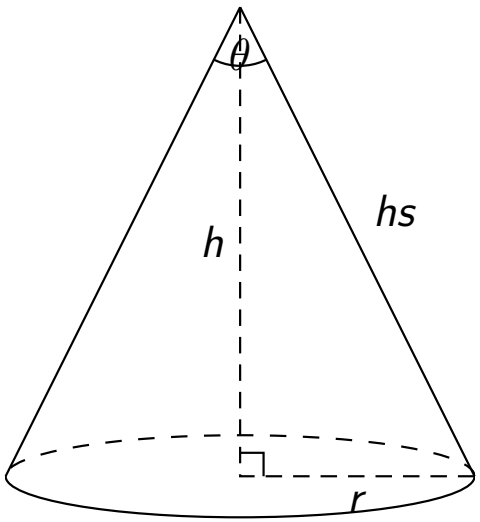
Automated tools for CSL?

- Great successes of sequential separation logic tools, e.g. Infer of Facebook, and SLAyer of Microsoft.
- Only 2 CSL tools appear *this year*, Caper and Starling.
- Our contributions:
  - The first formal verification for `CountDownLatch` by using abstract predicates.
  - A modular solution to the counter mechanism by combining a *thread-local* abstraction and the *global* view.
  - Ensure race-freedom and deadlock-freedom.
  - An automated prototype which can verify a library implemenation of `CountDownLatch`.

Motivation Example

```
c = create_latch(2);
```

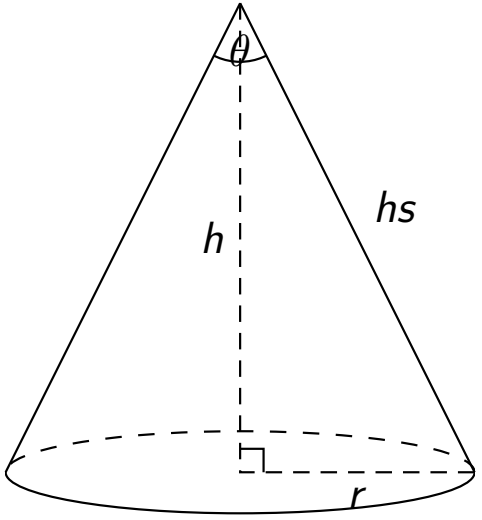
# T_1	# T_2	# T_3
h=hs*cosine(θ/2);	r=hs*cosine(θ/2);	
countDown(c);	countDown(c);	await(c);
		V=(1/3)*π*power(r,2)*h;

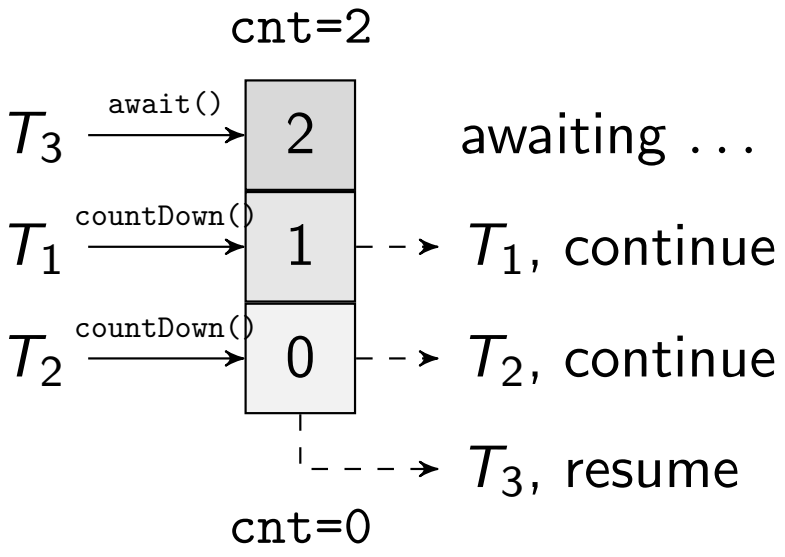


Motivation Example

```
c = create_latch(2);
```

# T_1	# T_2	# T_3
h=hs*cosine(θ/2);	r=hs*cosine(θ/2);	
countDown(c);	countDown(c);	await(c);
		V=(1/3)*π*power(r,2)*h;





Motivation Example

- Resources can be split and sent to multiple receiving threads.

```
c = create_latch(1);
```

$$\left(\begin{array}{l} \texttt{\# send P*Q} \\ \texttt{countDown(c);} \\ \texttt{...} \end{array} \;\middle\|\; \begin{array}{l} \texttt{await(c);} \\ \texttt{\# receive P} \\ \texttt{...} \end{array} \;\middle\|\; \begin{array}{l} \texttt{await(c);} \\ \texttt{\# receive Q} \\ \texttt{...} \end{array}\right)$$

Motivation Example

- Model the barrier synchronization.

```
c = create_latch(2);
```

```
(  ...                          ||  ...
   # owns P                        # owns Q
   countDown(c); await(c);      ||  countDown(c); await(c);
   # owns Q                        # owns P
   ...                          ||  ...                    )
```

Resource Predicates

```
CountDownLatch create_latch(n) with P
   requires n>0
   ensures LatchIn(res,P)*LatchOut(res,P)*CNT(res,n);
   requires n=0
   ensures CNT(res,-1);
```

Resource Predicates

```
void countDown(CountDownLatch i)
  requires LatchIn(i,P)*P*CNT(i,n)∧n>0
  ensures CNT(i,n-1);
  requires CNT(i,-1)
  ensures CNT(i,-1);
```

Resource Predicates

```
void countDown(CountDownLatch i)
  requires LatchIn(i,P)*P*CNT(i,n)∧n>0
  ensures CNT(i,n-1);
  requires CNT(i,-1)
  ensures CNT(i,-1);


void await(CountDownLatch i)
  requires LatchOut(i,P)*CNT(i,0)
  ensures P*CNT(i,-1);
  requires CNT(i,-1)
  ensures CNT(i,-1);
```

Splitting Lemmas

$$[\underline{\textsc{split}-1}] : \texttt{LatchOut}(\texttt{i}, \texttt{P*Q}) \longrightarrow \texttt{LatchOut}(\texttt{i}, \texttt{P})\texttt{*}\texttt{LatchOut}(\texttt{i}, \texttt{Q})$$

$$[\underline{\textsc{split}-2}] : \texttt{LatchIn}(\texttt{i}, \texttt{P*Q}) \longrightarrow \texttt{LatchIn}(\texttt{i}, \texttt{P})\texttt{*}\texttt{LatchIn}(\texttt{i}, \texttt{Q})$$

$$[\underline{\textsc{split}-3}] : \texttt{CNT}(\texttt{c}, \texttt{n}) \wedge \texttt{n1}\,\texttt{,n2} \geq 0 \wedge \texttt{n} = \texttt{n1+n2} \longrightarrow \texttt{CNT}(\texttt{c}, \texttt{n1})\texttt{*}\texttt{CNT}(\texttt{c}, \texttt{n2})$$

Splitting Lemmas

$$[\text{SPLIT}-1] : \texttt{LatchOut(i, P*Q)} \longrightarrow \texttt{LatchOut(i, P)*LatchOut(i, Q)}$$
$$[\text{SPLIT}-2] : \texttt{LatchIn(i, P*Q)} \longrightarrow \texttt{LatchIn(i, P)*LatchIn(i, Q)}$$
$$[\text{SPLIT}-3] : \texttt{CNT(c, n)} \wedge \texttt{n1, n2} \geq 0 \wedge \texttt{n} = \texttt{n1+n2} \longrightarrow \texttt{CNT(c, n1)*CNT(c, n2)}$$

```
c = create_latch(2) with P*Q;
# LatchOut(c, P*Q)*LatchIn(c, P*Q)*CNT(c, 2)
# LatchOut(c, P*Q)*LatchIn(c, P)*LatchIn(c, Q)*CNT(c, 0)*CNT(c, 1)*CNT(c, 1)
```

...	...create P...	...create Q...
# LatchOut(c, P*Q)*CNT(c, 0)	# P*LatchIn(c, P)*CNT(c, 1)	# Q*LatchIn(c, Q)*CNT(c, 1)
await(c);	countDown(c);	countDown(c);
# P*Q*CNT(c, -1)	# CNT(c, 0)	# CNT(c, 0)
...use P*Q...	...	...

Race Condition

```
c = create_latch(1) with P*Q;
```

$$\left(\begin{array}{l|l|l} \cdots & \cdots\texttt{create P}\cdots & \cdots\texttt{create Q}\cdots \\ \texttt{await(c);} & \texttt{countDown(c);} & \texttt{skip();} \\ \cdots\texttt{use P*Q}\cdots & \cdots & \cdots \end{array}\right)$$

- The first thread uses `Q` while it's not ready for use.

Race Error

[**ERR−1**]: `LatchIn(c,P)*CNT(c,-1)` $\longrightarrow$ `RACE-ERROR`

Race Error

[ERR−1]: LatchIn(c, P)*CNT(c,-1) ⟶ RACE-ERROR

```
                    c = create_latch(1) with P*Q;
                # LatchOut(c, P*Q)*LatchIn(c, P*Q)*CNT(c, 1)
        # LatchOut(c, P*Q)*LatchIn(c, P)*LatchIn(c, Q)*CNT(c, 0)*CNT(c, 1)*CNT(c, 0)
```

···	···create P···	···create Q···
# LatchOut(c, P*Q)*CNT(c, 0)	# P*LatchIn(c, P)*CNT(c, 1)	# Q*LatchIn(c, Q)*CNT(c, 0)
await(c);	countDown(c);	skip();
# P*Q*CNT(c, -1)	# CNT(c, 0)	# Q*LatchIn(c, Q)*CNT(c, 0)
···use P*Q···	···	···

```
# P*Q*CNT(c, -1) *CNT(c, 0) *Q*LatchIn(c, Q)*CNT(c, 0)
            # P*Q*CNT(c, -1) *Q*LatchIn(c, Q)
        # RACE-ERROR detected by [ERR−1]
```

Race Error

```
c = create_latch(1) with P*Q;
# LatchOut(c, P*Q)*LatchIn(c, P*Q)*CNT(c, 1)
# LatchOut(c, P*Q)*LatchIn(c, P)*LatchIn(c, Q)*CNT(c, 0)*CNT(c, 1)*CNT(c, 0)
```

...	...create P...	...create Q...
# LatchOut(c, P*Q)*CNT(c, 0)	# P*LatchIn(c, P)*CNT(c, 1)	# Q*LatchIn(c, Q)*CNT(c, 0)
await(c);	countDown(c);	countDown(c);
# P*Q*CNT(c, -1)	# CNT(c, 0)	# RACE-ERROR

- The precondition of `countDown` cannot be `CNT(c,0)`

Race Error

```
c = create_latch(2) with P*Q;
# LatchOut(c, P*Q)*LatchIn(c, P*Q)*CNT(c, 2)
# LatchOut(c, P*Q)*LatchIn(c, P)*LatchIn(c, Q)*CNT(c, 0)*CNT(c, 1)*CNT(c, 1)
```

···	···create P···	···create Q···
# LatchOut(c, P*Q)*CNT(c, 0)	# P*LatchIn(c, P)*CNT(c, 1)	# Q*LatchIn(c, Q)*CNT(c, 1)
await(c);	countDown(c);	countDown(c);
# P*Q*CNT(c, -1)	# CNT(c, 0)	# CNT(c, 0)
···use P*Q···	···	···

```
# P*Q*CNT(c, -1) *CNT(c, 0) *CNT(c, 0)
# P*Q*CNT(c, -1) *CNT(c, 0)
# P*Q*CNT(c, -1)
```

Race Error

$$\texttt{c = create\_latch(2) with P*Q;}$$
$$\texttt{\# LatchOut(c, P*Q)*LatchIn(c, P*Q)*CNT(c, 2)}$$
$$\texttt{\# LatchOut(c, P*Q)*LatchIn(c, P)*LatchIn(c, Q)*CNT(c, 0)*CNT(c, 1)*CNT(c, 1)}$$

$$\left(\begin{array}{l|l|l}
\cdots & \cdots\texttt{create P}\cdots & \cdots\texttt{create Q}\cdots \\
\texttt{\# LatchOut(c, P*Q)*CNT(c, 0)} & \texttt{\# P*LatchIn(c, P)*CNT(c, 1)} & \texttt{\# Q*LatchIn(c, Q)*CNT(c, 1)} \\
\texttt{await(c);} & \texttt{countDown(c);} & \texttt{countDown(c);} \\
\texttt{\# P*Q*CNT(c, -1)} & \texttt{\# CNT(c, 0)} & \texttt{\# CNT(c, 0)} \\
\cdots\texttt{use P*Q}\cdots & \cdots & \cdots
\end{array}\right)$$

$$\texttt{\# P*Q*CNT(c, -1) *CNT(c, 0) *CNT(c, 0)}$$
$$\texttt{\# P*Q*CNT(c, -1) *CNT(c, 0)}$$
$$\texttt{\# P*Q*CNT(c, -1)}$$

$$[\textsc{norm}-1] : \texttt{CNT(c, n)*CNT(c, -1)} \wedge \texttt{n} \leq 0 \longrightarrow \texttt{CNT(c, -1)}$$
$$[\textsc{norm}-2] : \texttt{CNT(c, n1)*CNT(c, n2)} \wedge \texttt{n=n1+n2} \wedge \texttt{n1, n2} \geq 0 \longrightarrow \texttt{CNT(c, n)}$$

Deadlock

```
c = create_latch(2);

( countDown(c); || await(c); );
```

Deadlock

$$\texttt{c = create\_latch(2);}$$

$$\big(\ \texttt{countDown(c);}\ \ \big\|\ \ \texttt{await(c);}\big)\ \texttt{;}$$

[**ERR−2**]: $\texttt{CNT(c,a)*CNT(c,-1)}\wedge\texttt{a>0} \longrightarrow \texttt{DEADLOCK-ERROR}$

Deadlock

```
c = create_latch(2);

( countDown(c); || await(c); );
```

[ERR−2]: CNT(c,a)*CNT(c,-1)∧a>0 ⟶ DEADLOCK-ERROR

```
c = create_latch(2);
# CNT(c,2) ⟶ CNT(c,2)*CNT(c,0)

( # CNT(c,2)          || # CNT(c,0)
  countDown(c);       || await(c);
  # CNT(c,1)          || # CNT(c,-1) );

# CNT(c,1)*CNT(c,-1)
# DEADLOCK-ERROR detected by [ERR−2]
```

Inter-latch Deadlock

```
c1 = create_latch(1); c2 = create_latch(1);
```

$$\left(\begin{array}{l} \texttt{await(c2);} \\ \texttt{countDown(c1);} \end{array} \right\| \left. \begin{array}{l} \texttt{await(c1);} \\ \texttt{countDown(c2);} \end{array} \right)$$

Inter-latch Deadlock

```
c1 = create_latch(1); c2 = create_latch(1);
```

$$\left(\begin{array}{l} \texttt{await(c2);} \\ \texttt{countDown(c1);} \end{array} \;\middle\|\; \begin{array}{l} \texttt{await(c1);} \\ \texttt{countDown(c2);} \end{array} \right)$$

$[\underline{\textbf{WAIT}-\textbf{1}}]$: $\texttt{WAIT(S)} \wedge \neg \texttt{isCyclic(S)} \longrightarrow \texttt{WAIT}(\{\})$

$[\underline{\textbf{WAIT}-\textbf{2}}]$: $\texttt{CNT}(c_1\texttt{,a}) * \texttt{CNT}(c_2\texttt{,-1}) * \texttt{WAIT(S)} \wedge \texttt{a>0} \longrightarrow$
$\texttt{CNT}(c_1\texttt{,a}) * \texttt{CNT}(c_2\texttt{,-1}) \wedge \texttt{a>0} * \texttt{WAIT}(\texttt{S} \cup \{c_2 \rightarrow c_1\})$

$[\underline{\textbf{WAIT}-\textbf{3}}]$: $\texttt{WAIT}(S_1) * \texttt{WAIT}(S_2) \longleftrightarrow \texttt{WAIT}((S_1 \cup S_2))$

$[\underline{\textbf{ERR}-\textbf{3}}]$: $\texttt{WAIT(S)} \wedge \texttt{isCyclic(S)} \longrightarrow \texttt{DEADLOCK-ERROR}$

Inter-latch Deadlock

```
c1 = create_latch(1); c2 = create_latch(1);
   # WAIT{} ∗ CNT(c1,1) ∗ CNT(c2,1) ⟹
# CNT(c1,1) * CNT(c2,0) * CNT(c2,1) * CNT(c1,0)
```

```
(  # CNT(c1,1)*CNT(c2,0)     ||  # CNT(c2,1)*CNT(c1,0)
   await(c2);                ||  await(c1);
   # CNT(c1,1)*CNT(c2,-1)    ||  # CNT(c2,1)*CNT(c1,-1)
   # ∗ WAIT({c2 → c1})       ||   # ∗ WAIT({c1 → c2})
   countDown(c1);            ||  countDown(c2);
   # CNT(c1,0)*CNT(c2,-1)    ||  # CNT(c2,0)*CNT(c1,-1)
   # ∗ WAIT({c2 → c1})       ||   # ∗ WAIT({c1 → c2})  )
```

```
#   CNT(c1,-1)*CNT(c2,-1) * WAIT({c2 → c1, c1 → c2})
      # DEADLOCK-ERROR detected by [ERR-3]
```

Implementation

Let's watch the demo

Conclusions

- Formal verification of `CountDownLatch` mechanism
- Detect deadlock and race cases
- Implement an automated verifier for `CountDownLatch` programs

Future work

- Verify fundamental concurrent algorithms: locks, etc.
- Compare with state-of-the-art tools Caper and Starling
- Build on the recent work of “tree share” model
- Build on recent logics such as Views, Iris, and LiLi

Thank you for your attention!

Questions ?

Interpretations for Abstract Predicates

$$\texttt{LatchOut(i,P)} \;\widehat{=}\; \boxed{\texttt{i}\mapsto\texttt{0}} \longrightarrow \texttt{P}$$

$$\texttt{LatchIn(i,P)} \;\widehat{=}\; (\texttt{P} \mathrel{-\!\!\circledast} \texttt{emp}) * [\texttt{DEC}]* \boxed{\texttt{i}\mapsto\texttt{m}} \wedge \texttt{m>0}$$

$$\texttt{DEC}: \quad \boxed{\texttt{i}\mapsto\texttt{n}} \wedge \texttt{n>0} \quad \rightsquigarrow \quad \boxed{\texttt{i}\mapsto\texttt{n-1}}$$

Interpretations for Abstract Predicates

$$\texttt{LatchOut(i,P)} \;\widehat{=}\; \boxed{\texttt{i}\mapsto\texttt{0}} \longrightarrow \texttt{P}$$

$$\texttt{LatchIn(i,P)} \;\widehat{=}\; (\texttt{P} \text{—}\circledast \texttt{emp}) * [\texttt{DEC}]* \boxed{\texttt{i}\mapsto\texttt{m}} \wedge \texttt{m>0}$$

$$\texttt{DEC}: \boxed{\texttt{i}\mapsto\texttt{n}} \wedge \texttt{n>0} \rightsquigarrow \boxed{\texttt{i}\mapsto\texttt{n-1}}$$

$$\{\!|\texttt{i}\mapsto\texttt{n}|\!\} \longrightarrow \boxed{\texttt{i}\mapsto\texttt{m}} \wedge \texttt{m}\geq\texttt{n}\geq\texttt{0}$$

$$\{\!|\texttt{i}\mapsto\texttt{n}|\!\} \wedge \texttt{a,b}\geq\texttt{0} \wedge \texttt{n=a+b} \longleftrightarrow \{\!|\texttt{i}\mapsto\texttt{a}|\!\} * \{\!|\texttt{i}\mapsto\texttt{b}|\!\}$$

$$\boxed{\texttt{i}\mapsto\texttt{a}} * \boxed{\texttt{i}\mapsto\texttt{b}} \longleftrightarrow \boxed{\texttt{i}\mapsto\texttt{a}} \wedge \texttt{a=b}$$

Interpretations for Abstract Predicates

$$\texttt{LatchOut(i,P)} \;\widehat{=}\; \boxed{\texttt{i}\mapsto\texttt{0}} \longrightarrow \texttt{P}$$

$$\texttt{LatchIn(i,P)} \;\widehat{=}\; (\texttt{P}\text{—}\circledast\texttt{emp}) * [\texttt{DEC}]* \boxed{\texttt{i}\mapsto\texttt{m}} \wedge \texttt{m>0}$$

$$\texttt{DEC}: \boxed{\texttt{i}\mapsto\texttt{n}} \wedge \texttt{n>0} \rightsquigarrow \boxed{\texttt{i}\mapsto\texttt{n-1}}$$

$$\{\!|\texttt{i}\mapsto\texttt{n}|\!\} \longrightarrow \boxed{\texttt{i}\mapsto\texttt{m}} \wedge \texttt{m}\geq\texttt{n}\geq\texttt{0}$$

$$\{\!|\texttt{i}\mapsto\texttt{n}|\!\} \wedge \texttt{a,b}\geq\texttt{0} \wedge \texttt{n=a+b} \longleftrightarrow \{\!|\texttt{i}\mapsto\texttt{a}|\!\} * \{\!|\texttt{i}\mapsto\texttt{b}|\!\}$$

$$\boxed{\texttt{i}\mapsto\texttt{a}} * \boxed{\texttt{i}\mapsto\texttt{b}} \longleftrightarrow \boxed{\texttt{i}\mapsto\texttt{a}} \wedge \texttt{a=b}$$

$$\texttt{CNT(i,n)} \;\widehat{=}\; \{\!|\texttt{i}\mapsto\texttt{n}|\!\} \wedge \texttt{n}\geq\texttt{0} \vee \boxed{\texttt{i}\mapsto\texttt{0}} \wedge \texttt{n=-1}$$

Core Language

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \overline{\mathit{data}}\ \ \overline{\texttt{proc}} \\
\texttt{datat} & ::= & \textbf{data}\ \texttt{C}\ \{\overline{t\ \ f}\} \\
\texttt{proc} & ::= & t\ \ \mathit{pn}(\overline{t\ \ v})\ \ \overline{\mathit{spec}}\{\ e\ \} \\
\texttt{spec} & ::= & \texttt{requires}\ \ \Phi_{pr}\ \ \texttt{ensure}\ \ \Phi \mathit{po}; \\
\texttt{t} & ::= & \texttt{void} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{CountDownLatch} \mid \texttt{C} \\
\texttt{e} & ::= & \texttt{v} \mid \texttt{v.f} \mid \texttt{k} \mid \texttt{new C}(\overline{v}) \mid e_1\,;e_2 \mid e_1 \,||\, e_2 \mid \texttt{<e>} \\
 & & \textbf{create\_latch}(\texttt{n})\ \ \textbf{with}\ \kappa \wedge \pi \mid \textbf{coutnDown}(n) \\
 & & \textbf{await}(v) \mid \mathit{pn}(\overline{v}) \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \ldots
\end{array}
$$

Core Language Example

```
data CDL{}
data cell{int val;}
pred_prim LatchIn{-%P@Split }<>
pred_prim LatchOut{+%P@Split }<>
pred_prim CNT<n:int>
  inv n >= (-1)

lemma "split" self::CNT<n> & a>=0 & b>=0 & n=a+b
    -> self::CNT<a> * self::CNT<b>;
// Normalization lemmas
lemma_prop "idemp-CNT" self::CNT<a> * self::CNT<(-1)>
    -> self::CNT<(-1)>;
lemma_prop "combine-CNT" self::CNT<a> * self::CNT<b> & a,b>=0
    -> self::CNT<a+b>;
```

Core Language Example

```
void main()
  requires emp  ensures emp;
{
  cell h, r;
  int v;
  CDL c = create_latch(2) with h'::cell<1> * r'::cell<2> * @full[h, r];
  par h, r, v, c@L
  {
    case h, c@L c'::LatchIn- h'::cell<1> * @full[h]<> * c'::CNT<(1)> ->
      h = new cell(1);
      countDown(c);
||
    case r, c@L c'::LatchIn- r'::cell<2> * @full[r]<> * c'::CNT<(1)> ->
      r = new cell(2);
      countDown(c);
||
    case v, c@L c'::LatchOut+ h'::cell<1> * r'::cell<2> * @full[h, r]<> * c'::CNT<0> ->
      await(c);
      v = h.val + r.val;
  }
  assert h'::cell<1> * r'::cell<2> & v' = 3;
}
```

Core Specification Language

FA Pred.	*rpred*	::=	$\texttt{pred R(self}, \bar{\texttt{V}}, \bar{\texttt{v}}) \; [\equiv \; \Phi] \; [\textbf{inv} \; \pi]$
Action	*act*	::=	$\texttt{action I} \; \equiv \; \overline{\iota_1 \wedge \pi_1 \rightsquigarrow \iota_2 \wedge \pi_2}$
Disj. formula	Φ	::=	$\bigvee(\exists \, \bar{\texttt{v}} \cdot \eta * \kappa \wedge \pi)$
Non-Resource	η	::=	$[\texttt{I}]_\xi \mid \texttt{WAIT}(\overline{v_1 \rightarrow v_2})_\xi \mid \boxed{\texttt{v} \mapsto \texttt{C}(\bar{\texttt{v}})} \mid \eta_1 * \eta_2$
Sep. formula	κ	::=	$\iota \mid \texttt{V} \mid \texttt{R}(\texttt{v}, \overline{\Phi_{\texttt{f}}}, \bar{\texttt{v}}) \mid \kappa_1 * \kappa_2$
Simple heap	ι	::=	$\texttt{emp} \mid \texttt{v} \mapsto \texttt{C}(\bar{\texttt{v}}) \mid \{\!\mid \texttt{v} \mapsto \texttt{C}(\bar{\texttt{v}}) \mid\!\} \mid \iota_1 * \iota_2$
Perms	ξ	::=	$\epsilon \mid 1$
Pure formula	π	::=	$\alpha \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg \pi \mid \exists \texttt{v} \cdot \pi \mid \forall \texttt{v} \cdot \pi$
Arith. formula	α	::=	$\alpha_1^t = \alpha_2^t \mid \alpha_1^t \neq \alpha_2^t \mid \alpha_1^t < \alpha_2^t \mid \alpha_1^t \leq \alpha_2^t$
Arith. term	α^t	::=	$\texttt{k} \mid \texttt{v} \mid \texttt{k} \times \alpha^t \mid \alpha_1^t + \alpha_2^t \mid -\alpha^t$

$k \in$ *integer constants* $\quad \texttt{v} \in$ *variables*, $\bar{\texttt{v}} \equiv \texttt{v}_1, .., \texttt{v}_n \quad \texttt{C} \in$ *data names*

$\texttt{V} \in$ *resource variables* $\quad \texttt{R} \in$ *resource pred. names* $\quad \epsilon \in (0, 1]$