Under-Approximation for Scalable Bug Detection

Azalea Raad
Imperial College London

Iris Workshop
23 May 2023
State of the Art: *Correctness*

- Lots of work on *reasoning* for proving *correctness*
  - Prove the *absence of bugs*
  - *Over-approximate* reasoning
  - *Compositionality*
    - in *code* ⇒ reasoning about *incomplete components*
    - in *resources* accessed ⇒ spatial locality
  - *Scalability* to large teams and codebases
Hoare Logic (HL)

Hoare triples \( \{p\} \ C \ \{q\} \iff \post(C)p \subseteq q \)

For all states \( s \) in \( p \)
if running \( C \) on \( s \) terminates in \( s' \), then \( s' \) is in \( q \)
Hoare Logic (HL)

Hoare triples \(\{p\} C \{q\}\) \(iff\) \(\text{post}(C)p \subseteq q\)

\(q \text{ over-approximates } \text{post}(C)p\)
Hoare Logic (HL)

Hoare triples \( \{p\} \ C \ \{q\} \iff \text{post}(C)p \subseteq q \)

\( q \text{ over-approximates} \ \text{post}(C)p \)
“Don’t spam the developers!”
Incorrectness Logic: A Formal Foundation for Bug Catching
Part I.
Incorrectness Logic (IL)
&
Incorrectness Separation Logic (ISL)
For all states $s$ in $p$ if running $C$ on $s$ terminates in $s'$, then $s'$ is in $q$.

Hoare triples $\{p\} \ C \ \{q\}$ iff $\text{post}(C)p \subseteq q$
Incorrectness Logic (IL)

Hoare triples \( \{p\} C \{q\} \) iff \( \text{post}(C)p \subseteq q \)

For all states \( s \) in \( p \)
if running \( C \) on \( s \) terminates in \( s' \), then \( s' \) is in \( q \)

Incorrectness triples \([p] C [q]\) iff \( \text{post}(C)p \supseteq q \)

For all states \( s \) in \( q \)
s can be reached by running \( C \) on some \( s' \) in \( p \)
Incorrectness Logic (IL)

Hoare triples \{p\} C \{q\} \iff \text{post}(C)p \subseteq q

\text{q over-approximates post}(C)p

Incorrectness triples \[p\] C \[q\] \iff \text{post}(C)p \supseteq q

\text{q under-approximates post}(C)p
Incorrectness Logic (IL)

Hoare triples \( \{p\} \ C \ \{q\} \iff \text{post}(C)p \subseteq q \)
\( q \ \text{over-approximates} \ \text{post}(C)p \)

Incorrectness triples \( [p] \ C \ [q] \iff \text{post}(C)p \supseteq q \)
\( q \ \text{under-approximates} \ \text{post}(C)p \)
Incorrectness Logic (IL)

\[ [p] \ C \ [\varepsilon : q] \]

\( \varepsilon \): exit condition
- ok: normal execution
- er: erroneous execution

\[ [y=v] \ x:=y \ [ok: x=y=v] \quad [p] \ error( ) \ [er: p] \]
Incorrectness Logic (IL)

\[ [p] \ C [\varepsilon : q] \iff \ \text{post}(C, \varepsilon)p \supseteq q \]
Incorrectness Logic (IL)

\[ \vdash_B [p] C [\epsilon: q] \iff \text{post}(C, \epsilon)p \supseteq q \]

Equivalent Definition (reachability)

\[ \vdash_B [p] C [\epsilon: q] \iff \forall s \in q. \exists s' \in p. (s', s) \in [C]\epsilon \]
IL Proof Rules and Principles (Sequencing)

\[
\begin{align*}
[p] & \ C_1 \ [er: q] \\
[p] & \ C_1; \ C_2 \ [er: q]
\end{align*}
\]

- **Short-circuiting** semantics for errors
IL Proof Rules and Principles (Sequencing)

\[
\begin{align*}
[p] \quad & C_1 \quad [\text{er: } q] \\
\hline
[p] \quad & C_1 \quad ; \quad C_2 \quad [\text{er: } q]
\end{align*}
\]

\[
\begin{align*}
[p] \quad & C_1 \quad [\text{ok: } r] \quad [r] \quad C_2 \quad [\varepsilon: q] \\
\hline
[p] \quad & C_1 \quad ; \quad C_2 \quad [\varepsilon: q]
\end{align*}
\]

- Short-circuiting semantics for errors
IL Proof Rules and Principles (Branches)

\[
[p] \ C_i \ [\varepsilon: q] \quad \text{some} \ i \in \{1, 2\} \\
[p] \ C_1 + C_2 \ [\varepsilon: q]
\]

- **Drop paths/branches** (this is a **sound** under-approximation)
- **Scalable** bug detection!

\[
[p] \ C \ [\varepsilon: q] \quad \text{iff} \quad \forall \ s \in q. \ \exists \ s' \in p. \ (s', s) \in [C]_\varepsilon
\]
IL Proof Rules and Principles (Loops)

- **Bounded unrolling of loops** (this is a **sound** under-approximation)
- **Scalable** bug detection!

\[
\begin{align*}
[p] \ C^* \ \text{[ok: p]} & \quad \text{(Unroll-Zero)} \\
[p] \ C^*; \ C \ \epsilon: q & \quad \text{(Unroll-Many)}
\end{align*}
\]

\[
[p] \ C \ \epsilon: q \quad \text{iff} \quad \forall s \in q. \ \exists s' \in p. \ (s',s) \in [C] \epsilon
\]
Loop invariants are inherently over-approximate

Reason about loops under-approximately via sub-variants

∀ n ∈ ℤ. [p(n)] C [ok: p(n+1)] k ∈ ℤ

[p(0)] C* [ok: p(k)]

(Lastly, backwards-variant)
IL Proof Rules and Principles (Consequence)

\[
p' \subseteq p \quad [p'] \ C [\varepsilon : q'] \quad q' \supseteq q \quad (\text{Cons})
\]

- **Shrink** the post (e.g. drop disjuncts)
- **Scalable** bug detection!

\[
[p] \ C [\varepsilon : q] \quad \text{iff} \quad \forall s \in q. \ \exists s' \in p. \ (s', s) \in [C] \varepsilon
\]
IL Proof Rules and Principles (Consequence)

\[ p' \subseteq p \quad [p'] \ C [\varepsilon: q'] \quad q' \supseteq q \quad (\text{Cons}) \]

\[ [p] \ C [\varepsilon: q] \]

\[ [p] \ C [\varepsilon: q_1 \lor q_2] \]

\[ [p] \ C [\varepsilon: q_1] \]

- **Shrink** the post (e.g. drop disjuncts)
- **Scalable** bug detection!

\[ [p] \ C [\varepsilon: q] \iff \forall s \in q. \exists s' \in p. (s',s) \in [C]\varepsilon \]
IL Proof Rules and Principles (Consequence)

\[
\frac{p' \subseteq p}{[p] C [\varepsilon: q]} \quad q' \supseteq q \quad (\text{Cons})
\]

\[
\frac{p' \supseteq p}{\{p'\} C \{q'\}} \quad q' \subseteq q \quad (\text{HL-Cons})
\]

\[
\frac{\frac{[p] C [\varepsilon: q_1 \lor q_2]}{[p] C [\varepsilon: q_1]}}{[p] C [\varepsilon: q_1 \lor q_2]}
\]

- **Shrink** the post (e.g. drop disjuncts)
- **Scalable** bug detection!

\[
[p] C [\varepsilon: q] \iff \forall s \in q. \exists s' \in p. (s', s) \in [C]\varepsilon
\]
Incorrectness Logic: Summary

+ *Under-approximate* analogue of Hoare Logic

+ Formal foundation for *bug catching*

  – Global reasoning: *non-compositional* (as in original Hoare Logic)

  – Cannot target *memory safety bugs* (e.g. use-after-free)
Incorrectness Logic: Summary

+ Under-approximate analogue of Hoare Logic
+ Formal foundation for bug catching
  - Global reasoning: non-compositional (as in original Hoare Logic)
  - Cannot target memory safety bugs (e.g. use-after-free)

Solution

Incorrectness Separation Logic
Incorrectness Separation Logic (ISL)

**IL**

\[
[p] \ C \ [\varepsilon: q]
\]

**SL**

\[
\{p\} \ C \ \{q\}
\]

\[
\{p * r\} \ C \ \{q * r\}
\]

\[
x \mapsto - * x \mapsto - \Leftrightarrow false
\]

\[
x \mapsto v * emp \Leftrightarrow x \mapsto v
\]

**ISL**

\[
[p] \ C \ [\varepsilon: q]
\]

\[
[p * r] \ C \ [\varepsilon: q * r]
\]

\[
x \mapsto v * x \mapsto v' \Leftrightarrow false
\]

\[
x \mapsto v * emp \Leftrightarrow x \mapsto v
\]
ISL: Local Axioms

null-pointer-dereference error  double-free error
ISL: Local Axioms

\[ \boxed{\text{FREE}} \]

\[ [x \mapsto v] \text{ free}(x) \ [\text{ok: } x \mapsto v] \]

\[ [x = \text{null}] \text{ free}(x) \ [\text{er: } x = \text{null}] \]

\[ [x \mapsto v'] \ [x] := v \ [\text{ok: } x \mapsto v] \]

\[ [x = \text{null}] \ [x] := v \ [\text{er: } x = \text{null}] \]

\[ [x \mapsto v] \ [x] := v \ [\text{er: } x \mapsto v] \]

null-pointer-dereference error

double-free error
### ISL: Local Axioms

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( { x \mapsto v } \text{free}(x) )</td>
<td>( { x \mapsto v } \text{free}(x) ) is ok if ( x ) is free.</td>
<td>null-pointer-dereference error</td>
</tr>
<tr>
<td>( { x \mapsto v } \text{free}(x) )</td>
<td>( { x \mapsto v } \text{free}(x) ) is ok if ( x ) is free.</td>
<td>double-free error</td>
</tr>
<tr>
<td>( { x \mapsto v' } [x] \assign v )</td>
<td>( [x] \assign v ) is ok if ( x ) is free.</td>
<td></td>
</tr>
<tr>
<td>( { x \mapsto v } ) ( y \assign [x] )</td>
<td>( [x] \assign v ) and ( y = v ) is ok if ( x ) is free.</td>
<td></td>
</tr>
</tbody>
</table>

**Free Assignment (FREE):**
- \( \{ x \mapsto v \} \text{free}(x) \)
- \( \{ x \mapsto v \} \text{free}(x) \)

**Write (WRITE):**
- \( \{ x \mapsto v' \} [x] \assign v \)
- \( \{ x \mapsto v \} [x] \assign v \)

**Read (READ):**
- \( \{ x \mapsto v \} [x] \assign [x] \)
- \( \{ x \mapsto v \} [x] \assign [x] \)
ISL: Local Axioms

**FREE**

\[
[x \mapsto v] \text{ free}(x) \quad [\text{ok: } x \mapsto] \\
\text{null-pointer-dereference error}
\]

**WRITE**

\[
[x \mapsto v'] \quad [x] := v \quad [\text{ok: } x \mapsto v] \\
[x = \text{null}] \quad [x] := v \quad [\text{er: } x = \text{null}] \\
\text{double-free error}
\]

**READ**

\[
[x \mapsto v] \quad y := [x] \quad [\text{ok: } x \mapsto v \land y = v] \\
[x = \text{null}] \quad y := [x] \quad [\text{er: } x = \text{null}] \\
\text{null-pointer-dereference error}
\]

**ALLOC**

\[
[\text{emp}] \quad x := \text{alloc()} \quad [\text{ok: } \exists l. l \mapsto v \land x = l]
\]
ISL Summary

- Incorrectness **Separation Logic** (ISL)
  - IL + SL for **compositional bug catching**
  - **Under-approximate** analogue of SL
  - Targets **memory safety bugs** (e.g. use-after-free)

- Combining IL+SL: not straightforward
  - **invalid frame** rule!

- Fix: a **monotonic model** for frame preservation

- Recovering the **footprint property** for completeness

- ISL-based **analysis**
  - **No-false-positives theorem:** All bugs found are true bugs
Part II.
Pulse-X: ISL for Scalable Bug Detection
Pulse-X at a Glance

- **Automated** program analysis for **memory safety errors** (NPEs, UAFs) and **leaks**
- Underpinned by ISL (under-approximate) — **no false positives***
- **Inter-procedural** and **bi-abductive** — under-approximate analogue of Infer
- **Compositional** (begin-anywhere analysis) — important for CI
- Deployed at Meta

**Performance**: comparable to Infer, though merely an academic tool!

**Fix rate**: comparable or better than Infer!

- Three dimensional scalability
  - code size (large codebases)
  - people (large teams, CI)
  - speed (high frequency of code changes)
Compositional, Begin-Anywhere Analysis

- Analysis result of a program = analysis results of its parts + a method of combining them
Compositional, Begin-Anywhere Analysis

- **Analysis result** of a program = analysis results of its parts + a method of combining them

- **Parts:** Procedures

```plaintext
   f1
  /   
 /     
  f2
  |     |
  |     |
  f3   f4

   g1
  /   
 /     
  g2
  |     |
  |     |
  h    g4
  |     |
  |     |
g3----
```
Compositional, Begin-Anywhere Analysis

- **Analysis result** of a program = analysis results of its **parts** + a **method** of combining them

- **Parts:** Procedures

- **Method:** under-approximate bi-abduction
Compositional, Begin-Anywhere Analysis

- **Analysis result** of a program = analysis results of its **parts** + a **method** of combining them

  ➤ **Parts:** Procedures

  ➤ **Method:** under-approximate bi-abduction

  ➤ **Analysis result:** incorrectness triples (under-approximate specs)
Pulse-X Algorithm: Proof Search in ISL

- Analyse each procedure $f$ in isolation, find its **summary** (collection of ISL triples)
  - A **summary table** $T$, initially populated only with local (pre-defined) axioms
  - Use bi-abduction and $T$ to find the summary of $f$
  - Recursion: bounded unrolling
  - Extend $T$ with the summary of $f$

- Similar bi-abductive mechanism to Infer, but:
  - Can **soundly** drop execution paths/branches
  - Can **soundly** bound loop unrolling
Pulse-X: Null Pointer Dereference in OpenSSL

1. `int ssl_excert_prepend(...){`
2. `SSL_EXCERT *exc= app_malloc(sizeof(*exc), "prepend cert");`
3. `memset(exc, 0, sizeof(*exc));`
   ... calls CRYPTO_malloc (a malloc wrapper)
}

Pulse-X: Null Pointer Dereference in OpenSSL

1. `int ssl_excert_prepend(...){`
2. `SSL_EXCERT *exc= app_malloc(sizeof(*exc), "prepend cert");`
3. `memset(exc, 0, sizeof(*exc));`
   ...
}

null pointer dereference

calls CRYPTO_malloc (a malloc wrapper)

CRYPTO_malloc may return null!
Pulse-X: Null Pointer Dereference in OpenSSL

1. `int ssl_excert_prepend(...){`
2. `SSL_EXCERT *exc= app_malloc(sizeof(*exc), "prepend cert");`
3. `memset(exc, 0, sizeof(*exc));`
   ...
}

null pointer dereference

calls CRYPTO_malloc (a malloc wrapper)

CRYPTO_malloc may return null!

[emp] *exc= app_malloc(sz, ...) [ok: exc = null ]
  +
[exc = null ] memset(exc,-,-) [er: exc = null ]

[emp] ssl_excert_prepend(...) [er: exc = null ]
### Pulse-X: Null Pointer Dereference in OpenSSL

```c
... ... @@ -956,6 +956,0 @@ static int ssl_excert_prepend(SSL_EXCERT **pexc)
{
SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");

+ if (!exc) {
```

- **paulidale** 13 days ago  Contributor

  False positive, `app_malloc()` doesn’t return if the allocation fails.

- **lequangloc** 13 days ago  Author

  Our tool recognizes `app_malloc()` in `test/testutil/apps_mem.c` rather than the one in `apps/lib/apps.c` While the former doesn’t return if the allocation fails, the latter does. How do we know which one is actually called?

- **paulidale** 13 days ago  Contributor

  It would need to look at the link lines or build dependencies to figure out which sources were used.

  We should fix the one in `test/testutil/apps_mem.c`.
Pulse-X: Null Pointer Dereference in OpenSSL

Created pull request #15836 to commit the fix.
No False Positives: Report *All* Bugs Found?

Not quite…
Pulse-X: Bug Reporting

1. `void foo(int *x){
2.   *x = 42;
}

WRITE [x=null] *x = v [er: x=null]

[x=null] foo(x) [er: x=null]

Should we report this NPD?
Pulse-X: Bug Reporting

1. ```void foo(int *x){
   *x = 42;
}
```

2. ```WRITE [x=null] *x = v [er: x=null]```  

```[x=null] foo(x) [er: x=null]```  

```
Should we report this NPD?
```

Developer

```
“But I never call foo with null!”
```

Pulse-X

```
“Which bugs shall I report then?”
```
Pulse-X: Bug Reporting

```c
1. void foo(int *x) {
2.   *x = 42;
}
```

**Problem**
Must consider the **whole program** to decide whether to report

**Solution**
Manifest Errors

“But I never call foo with null!”     “Which bugs shall I report then?”
Pulse-X: *Manifest* Errors

- **Intuitively:** the error occurs for all input states
- **Formally:** $[p] C [er: q]$ is manifest iff:
  \[
  \forall s. \exists s'. (s, s') \in [C]_{er} \land s' \in (q * \text{true})
  \]
- **Algorithmically:** …
Pulse-X: Null Pointer Dereference in OpenSSL

1. `int ssl_excert_prepend(...){`
2. `SSL_EXCERT *exc= app_malloc(sizeof(*exc), "prepend cert");`
3. `memset(exc, 0, sizeof(*exc));`
   
   calls CRYPTO_malloc (a malloc wrapper)

   null pointer dereference

   CRYPTO_malloc may return null!

   [emp] ssl_excert_prepend(...) [er: exc = null ]
Pulse-X: Null Pointer Dereference in OpenSSL

1. int ssl_excert_prepend(...){
2.   SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");
3.   memset(exc, 0, sizeof(*exc));
   ...
}

[emp] ssl_excert_prepend(...) [er: exc = null ]

Manifest Error (all calls to ssl_excert_prepend can trigger the error)!
Pulse-X: *Latent* Errors

An error triple \([p] C [er: q]\) is *latent* iff it is not manifest.
1. `int chopup_args(ARGS *args,...)`{
   ...
2.   `if (args->count == 0 ) {`
3.       args->count=20;
4.   args->data= (char**)ssl_excert_prepend(...);
5.   }
5.   `for (i=0; i<args->count; i++) {`
6.       args->data[i]=NULL;
   }
   ...
1. `int chopup_args(ARGS *args, ...){
   ...
2.   if (args->count == 0) {
3.      args->count=20;
4.      args->data = (char**)ssl_excert_prepend(...);
5.   }
6.   for (i=0; i<args->count; i++) {
7.      args->data[i]=NULL;
8.   }
   ...
}`
1. int chopup_args(ARGS *args,...) {
    ...
2.    if (args->count == 0) {
3.        args->count=20;
4.        args->data = (char**)ssl_excert_prepend(...);
5.    }
6.    for (i=0; i<args->count; i++) {
    args->data[i]=NULL;
6.    }
   ...
}

Latent Error: 
only calls with args->count==0 can trigger the error

null pointer dereference
static int www_body(...) {
    ...
    io = BIO_new(BIO_f_buffer());
    ssl_bio BIO_new(BIO_f_ssl());
    ...
    BIO_push(io, ssl_bio);
    ...
    BIO_free_all(io);
    ...
    return ret;
}
Pulse-X: Memory Leak in OpenSSL

```c
static int www_body(...) {
    ...
    io = BIO_new(BIO_f_buffer());
    ssl_bio BIO_new(BIO_f_ssl());
    ...
    BIO_push(io, ssl_bio);
    ...
    BIO_free_all(io);
    ...
    return ret;
}

does nothing when io is null
```
static int www_body(...) {
    ...
    io = BIO_new(BIO_f_buffer());
    ssl_bio = BIO_new(BIO_f_ssl());
    ...
    BIO_push(io, ssl_bio);
    ...
    BIO_free_all(io);
    ...
    return ret;
}

does nothing when io is null

leaks ssl_bio
static int www_body(...)
{

... 
io = BIO_new(BIO_f_buffer());
ssl_bio BIO_new(BIO_f_ssl());
...
BIO_push(io, ssl_bio);
...
BIO_free_all(io);
...
return ret;
}

426 lines of complex code: 
i o manipulated by several procedures
and multiple loops

Pulse-X performs under-approximation
with bounded loop unrolling

does nothing when io is null

leaks ssl_bio
No-False Positives: Caveat

- Unknown procedures (e.g. where the code is unavailable) are treated as skip
- Incomplete arithmetic solver

Speed
(fast but simplistic) vs
Precision
(slow but accurate)

“Scientists seek perfection and are idealists. ... An engineer’s task is to not be idealistic. You need to be realistic as you have to compromise between conflicting interests.”
Pulse-X Summary

- Automated program analysis for detecting memory safety errors and leaks
- Manifest errors (underpinned by ISL): no false positives
- Compositional, scalable, begin-anywhere
Part III.

ISL Extensions:
Concurrent Incorrectness Separation Logic (CISL)
&
Concurrent Adversarial Separation Logic (CASL)
&
Incorrectness Non-Termination Logic (INTL)
Termination vs Non-Termination

- Showing **termination** is compatible with **correctness** frameworks:
  - **Every** trace of a given program must terminate
  - Inherently **over-approximate**

  \[
  \text{skip} + x := 1
  \]
Termination vs Non-Termination

❖ Showing termination is compatible with correctness frameworks:

→ Every trace of a given program must terminate
→ Inherently over-approximate

\[ \text{skip + x:=1} \]

❖ Showing non-termination compatible with incorrectness frameworks:

→ Some trace of a given program must not-terminate
→ Inherently under-approximate

\[ \text{skip + while(true)skip} \]
Incorrectness Non-Termination Logic (INTL)

❖ A framework for **detecting non-termination bugs**
❖ Supports **unstructured** constructs (goto), as well exceptions and breaks
❖ Reasons for non-termination:
  ➪ Infinite loops
  ➪ Infinite recursion
  ➪ Cyclic goto soups
INTL Divergence Proof Rules

[p] C [∞]

C has divergent traces starting from p
INTL Divergence Proof Rules

\[ [p] \ C \ [\infty] \]

C has divergent traces starting from p
INTL Divergence Proof Rules (Sequencing)

\[
\begin{array}{c}
\text{[p]} \quad C_1 \quad \infty \\
\hline
\text{[p]} \quad C_1; \quad C_2 \quad \infty
\end{array}
\]
INTL Proof Rules and Principles

INTL Proof Rules

= 
(Under-Approximate) IL/ISL Proof Rules 
+
Divergence (Non-Termination) Rules
INTL Divergence Proof Rules (Sequencing)

\[
\frac{[p] C_1 \ [\infty]}{[p] C_1; \ C_2 \ [\infty]}
\]

\[
\frac{\vdash_B [p] C_1 \ [\text{ok: } q]}{[q] C_2 \ [\infty]}
\]

\[
\vdash_B [p] C_1; \ C_2 \ [\infty]
\]
INTL Divergence Proof Rules (Branches)

\[
\begin{array}{c}
[p] C_i [\infty] \quad \text{some } i \in \{1, 2\}
\end{array}
\]

\[
[p] C_1 + C_2 [\infty]
\]

- Drop paths/branches (this is a **sound** under-approximation)
- **Scalable** bug detection!
INTL Divergence Proof Rules (Loops — first attempt)

\[ [q] \ C; \ C^* \ [\infty] \]

\[ [p] \ C^* \ [\infty] \]
INTL Divergence Proof Rules (Loops — first attempt)

\[
\frac{[q] \ C; \ C^* [\infty]}{[p] \ C^* [\infty]}
\]

\[
\frac{[p] \ C_1 [\infty]}{[p] \ C_1; \ C_2 [\infty]}
\]
INTL Divergence Proof Rules (Loops — first attempt)

\[
\begin{align*}
[q] & \quad C; C^* \quad [\infty] \\
[p] & \quad C^* \quad [\infty] \\
\hline
[p] & \quad C \quad [\infty] \\
[p] & \quad C^* \quad [\infty] \quad \text{(derived)}
\end{align*}
\]

\[
[p] \quad C_1 \quad [\infty] \\
[p] \quad C_1; C_2 \quad [\infty]
\]
INTL Divergence Proof Rules (Loops — first attempt)

\[
\frac{[q] \ C; \ C^* \ [\infty]}{[p] \ C^* \ [\infty]}
\]

\[
\frac{[p] \ C \ [\infty]}{[p] \ C^* \ [\infty]}
\] (derived)

\[
\frac{\vdash_B [p] \ C \ [\text{ok: } p]}{[p] \ C^* \ [\infty]}
\]
INTL Divergence Proof Rules (Loops — first attempt)

\[
\begin{align*}
[q] & \quad C; C^* [\infty] \\
\hline
[p] & \quad C^* [\infty]
\end{align*}
\]

\[
\begin{align*}
[p] & \quad C [\infty] \\
\hline
[p] & \quad C^* [\infty]
\end{align*}
\]

(derived)

\[
\begin{align*}
\vdash_B & \quad [p] \quad C [ok: p] \\
\hline
[p] & \quad C^* [\infty]
\end{align*}
\]
INTL Divergence Proof Rules (While Loops — first attempt)

\[ [p \land b] \text{ while}(b) \ C \ [\infty] \]

**while** (b) C \equiv (assume(b); C)^*; assume(!b)
INTL Divergence Proof Rules (While Loops — first attempt)

\[ [p \land b] \text{(assume}(b); C)^*; \text{assume}(\neg b) \quad [\infty] \]

\[ [p \land b] \text{while}(b) C \quad [\infty] \]

\textbf{while} (b) C \equiv (\text{assume}(b); C)^*; \text{assume}(\neg b)
INTL Divergence Proof Rules (While Loops — first attempt)

\[
[p \land b] \ (\text{assume}(b); \ C)^* ; \ \text{assume}(\neg b) \ [\infty] \\
\frac{[p \land b] \ \text{while}(b) \ C \ [\infty]}{[p] \ C_1 \ [\infty] \quad [p] \ C_1 ; \ C_2 \ [\infty]}
\]

\textbf{while} (b) \ C \equiv (\text{assume}(b); \ C)^* ; \ \text{assume}(\neg b)
INTL Divergence Proof Rules (While Loops — first attempt)

\[
\begin{align*}
[p \land b] & \ (\text{assume}(b); \ C)^*; [\infty] \\
[p \land b] & \ (\text{assume}(b); \ C)^*; \ \text{assume}(\lnot b) \ [\infty] \\
\hline \\
[p \land b] & \ \text{while}(b) \ C \ [\infty]
\end{align*}
\]

\[
\begin{align*}
[p] & \ C_1 \ [\infty] \\
[p] & \ C_1; \ C_2 \ [\infty]
\end{align*}
\]

\textbf{while} (b) C \ \equiv \ (\text{assume}(b); \ C)^*; \ \text{assume}(\lnot b)
INTL Divergence Proof Rules (While Loops — first attempt)

\[
\begin{align*}
\frac{[p \land b] \ (\text{assume}(b); \ C)^*; [\infty]}{[p \land b] \ (\text{assume}(b); \ C)^*; \ \text{assume}(!b) \ [\infty]} \\
\hline
[p \land b] \ \text{while}(b) \ C \ [\infty]
\end{align*}
\]

\[
\frac{\vdash_b [p] \ C \ [\text{ok: } p]}{[p] \ C^* \ [\infty]}
\]
INTL Divergence Proof Rules (While Loops — first attempt)

\[
\frac{
\vdash_B [p \land b] \text{ assume}(b); \ C \ [\text{ok: } p \land b]
}{
[p \land b] (\text{assume}(b); \ C)^*;[\infty]
\}
\]

\[
\frac{
[p \land b] (\text{assume}(b); \ C)^*; \text{ assume}(!b) \ [\infty]
}{
[p \land b] \text{ while}(b) \ C \ [\infty]
\}
\]

\[ \text{while } (b) \ C \equiv (\text{assume}(b); \ C)^*; \text{ assume}(!b) \]
INTL Divergence Proof Rules (While Loops — first attempt)

\[ \vdash_B [p \land b] \text{ assume}(b); C \ [\text{ok: } p \land b] \]
\[ [p \land b] (\text{assume}(b); C)^*; [\infty] \]
\[ [p \land b] \text{ while}(b) C \ [\infty] \]

while \ (b) C \equiv (\text{assume}(b); C)^*; \text{ assume}(!b)
while (b) C ≡ (assume(b); C)*; assume(!b)
INTL Divergence Proof Rules (While Loops — first attempt)

\[ \vdash_b [p \land b] C \quad [\text{ok: } p \land b] \]

\[ [p \land b] \text{ while}(b) C \quad [\infty] \]

while \( (b) C \equiv (\text{assume}(b); C)^*; \text{assume(!b)} \)
\[ \begin{align*} &\vdash_B [p \land b] \ C \ [\text{ok: } p \land b] \\ \text{[p \land b] while}(b) \ C \ [\infty] \end{align*} \]
INTL Divergence Proof Rules (Loops — first attempt)

Program $\text{while}(x > 0) \ x--$ always terminates. But…

\[
\vdash_B [p \land b] \ C \; [\text{ok: } p \land b] \\
[p] \ \text{while}(b) \ C \; [\infty]
\]
INTL Divergence Proof Rules (Loops — first attempt)

Program \( \text{while}(x > 0) \ x-- \) always terminates. But…

\[
\begin{bmatrix}
\ x > 0 \ \\
\text{while}(x > 0) \ x--
\end{bmatrix} \quad [\infty]
\]

\( \vdash_B [p \land b] \ C \ [\text{ok: } p \land b] \)

\[
[ p ] \ \text{while}(b) \ C \ [\infty]
\]
INTL Divergence Proof Rules (Loops — first attempt)

Program \( \text{while}(x > 0) \ x-- \) always terminates. But…

\[ \vdash_B [x > 0] \ x-- \ [\text{ok: } x > 0] \]

\[ \vdash_B [p \land b] \ C \ [\text{ok: } p \land b] \]

\[ [p] \ \text{while}(b) \ C \ [\infty] \]
INTL Divergence Proof Rules (Loops — first attempt)

Program while\((x > 0)\ x--\) always terminates. But…

\[
\vdash_B [x > 0] \ x-- \ [\text{ok: } x > 0]
\]

\[
[ x > 0 ] \ \text{while}(x > 0) \ x-- \ [\infty]
\]

\[
\vdash_B [p \land b] \ C \ [\text{ok: } p \land b]
\]

\[
[p] \ \text{while}(b) \ C \ [\infty]
\]

\[
\vdash_B [p] \ C \ [\varepsilon: q]
\]

iff

\[
\forall s \in q. \ \exists s' \in p. \ (s',s) \in [C]_\varepsilon
\]
Problem

- Premise: p reached by executing C on some p
- I.e. in the **backward** direction
- Can construct a *backward infinite* trace

\[\vdash_B \, [p] \, C \, [\text{ok: p}] \]

\[\vdash [p] \, C^* \, [\infty]\]
Problem

- Premise: $p$ reached by executing $C$ on some $p$
- I.e. in the **backward** direction
- Can construct a *backward infinite* trace
- We need a *forward infinite* trace

\[
\Gamma_B \vdash [p] \ C \ [\text{ok: } p] \\
\hline
[p] \ C^* \ [\infty]
\]
Problem

- Premise: p reached by executing C on some p
- I.e. in the **backward** direction
- Can construct a **backward** infinite trace
- We need a **forward** infinite trace

Solution

**Forward Under-Approximate Triples**
Forward Under-Approximate (FUX) Triples

\[ \vdash_F [p] C [\varepsilon: q] \quad \text{iff} \quad \forall s \in p. \exists s' \in q. (s, s') \in [C]\varepsilon \]
Forward Under-Approximate (FUX) Triples

\[ \vdash_F [p] C [\varepsilon: q] \iff \forall s \in p. \exists s' \in q. (s, s') \in [C]\varepsilon \]

\[ \vdash_F [p] C [\text{ok: p}] \]

\[ [p] C^* [\infty] \]
Forward Under-Approximate (FUX) Triples

\[ \vdash_F [p] C [\varepsilon: q] \iff \forall s \in p. \exists s' \in q. (s, s') \in [C]\varepsilon \]

\[ \vdash_F [p] C [\text{ok: } p] \quad \frac{[p] C^* [\infty]}{[p] C^* [\infty]} \]
FUX is **Under-Approximate!**

\[ \vdash_F [p] C [\varepsilon : q] \iff \forall s \in p. \exists s' \in q. (s, s') \in [C]\varepsilon \]
FUX is **Under-Approximate!**

\[ \vdash_F [p] C \ [\varepsilon \colon q] \quad \text{iff} \quad \forall s \in p. \ \exists s' \in q. \ (s, s') \in [C] \varepsilon \]

\[ \vdash_F [p] C_1 \ [\text{er} \colon q] \quad \vdash_F [p] C_1 \land C_2 \ [\text{er} \colon q] \]

\[ \vdash_F [p_1] C \ [\varepsilon \colon q_1] \quad \vdash_F [p_2] C \ [\varepsilon \colon q_2] \quad \vdash_F [p_1 \lor p_2] C \ [\varepsilon \colon q_1 \lor q_2] \]

\[ \vdash_F [p] C_1 \lor C_2 \ [\varepsilon \colon q_1 \lor q_2] \]

\[ \vdash_F [p] C_i \ [\varepsilon \colon q] \quad \text{some} \ i \in \{1, 2\} \]

\[ \vdash_F [p] C^*; C \ [\varepsilon \colon q] \]

\[ \vdash_F [p] C^* \ [\text{ok} \colon p] \]
FUX is **Under-Approximate!**

\[ \vdash_F [p] C [\varepsilon: q] \iff \forall s \in p. \exists s' \in q. (s, s') \in [C]_\varepsilon \]

\[ \vdash_{BF} [p] C_1 [\text{er}: q] \]
\[ \vdash_{BF} [p] C_1; C_2 [\text{er}: q] \]

\[ \vdash_{BF} [p] C_1 [\text{ok}: r] \quad \vdash_{BF} [r] C_2 [\varepsilon: q] \]
\[ \vdash_{BF} [p] C_1; C_2 [\varepsilon: q] \]

\[ \vdash_{BF} [p_1] C [\varepsilon: q_1] \quad \vdash_{BF} [p_2] C [\varepsilon: q_2] \]
\[ \vdash_{BF} [p_1 \lor p_2] C [\varepsilon: q_1 \lor q_2] \]

\[ \vdash_{BF} [p_1] C_1 [\varepsilon: q_1] \quad \vdash_{BF} [p_2] C_2 [\varepsilon: q_2] \quad \text{some } i \in \{1, 2\} \]
\[ \vdash_{BF} [p] C_1 + C_2 [\varepsilon: q] \]

\[ \vdash_{BF} [p] C^*; C [\varepsilon: q] \]
\[ \vdash_{BF} [p] C^* [\varepsilon: q] \]

\[ \vdash_{BF} [p] C^* [\text{ok: } p] \]
FUX is Under-Approximate!

Q: What is the difference between FUX and BUX reasoning?

A: Rule of Consequence
BUX vs. FUX

(ConsB)

\[ p' \subseteq p \quad \vdash_B [p'] C [\epsilon : q'] \quad q' \supseteq q \]

\[ \vdash_B [p] C [\epsilon : q] \]

\[ \vdash_B [p] C [\epsilon : q] \iff \forall s \in q. \exists s' \in p. (s',s) \in [C]\epsilon \]
BUX vs. FUX

(ConsB)

\[ p' \subseteq p \quad \vdash_B [p'] C [\varepsilon: q'] \quad q' \supseteq q \]

\[ \vdash_B [p] C [\varepsilon: q] \]

(ConsF)

\[ p' \supseteq p \quad \vdash_F [p'] C [\varepsilon: q'] \quad q' \subseteq q \]

\[ \vdash_F [p] C [\varepsilon: q] \]

\[ \vdash_B [p] C [\varepsilon: q] \quad \text{iff} \]
\[ \forall s \in q. \exists s' \in p. (s',s) \in [C]_\varepsilon \]

\[ \vdash_F [p] C [\varepsilon: q] \quad \text{iff} \]
\[ \forall s \in p. \exists s' \in q. (s,s') \in [C]_\varepsilon \]
(ConsB)

\[
p' \subseteq p \quad \vdash_B [p'] \ C \ [\varepsilon: q'] \quad q' \supseteq q
\]

\[
\vdash_B [p] \ C \ [\varepsilon: q]
\]

(ConsF)

\[
p' \supseteq p \quad \vdash_F [p'] \ C \ [\varepsilon: q'] \quad q' \subseteq q
\]

\[
\vdash_F [p] \ C \ [\varepsilon: q]
\]

\[
\vdash_B [p] \ C \ [\varepsilon: q] \quad \text{iff} \quad \forall s \in q. \exists s' \in p. \ (s',s) \in [C]_{\varepsilon}
\]

\[
\vdash_F [p] \ C \ [\varepsilon: q] \quad \text{iff} \quad \forall s \in p. \exists s' \in q. \ (s,s') \in [C]_{\varepsilon}
\]

\[
\vdash_B [p] \ C \ [\varepsilon: q_1 \lor q_2]
\]

\[
\vdash_B [p] \ C \ [\varepsilon: q_1]
\]

Shrink the post
BUX vs. FUX

(ConsB)

\[
p' \subseteq p \quad \vdash_B [p'] C [\varepsilon : q'] \
q' \supseteq q \\
\vdash_B [p] C [\varepsilon : q]
\]

\[
\vdash_B [p] C [\varepsilon : q] \iff \forall s \in q. \exists s' \in p. (s',s) \in [C]\varepsilon
\]

Shrink the post

(ConsF)

\[
p' \supseteq p \quad \vdash_F [p'] C [\varepsilon : q'] \
q' \subseteq q \\
\vdash_F [p] C [\varepsilon : q]
\]

\[
\vdash_F [p] C [\varepsilon : q] \iff \forall s \in p. \exists s' \in q. (s,s') \in [C]\varepsilon
\]

Shrink the pre
Problem

Want to use existing **UX tools** (e.g. Pulse) based on BUX

How to **practically reconcile BUX & FUX**?
When are Disj and ConsB used in BUX?

\[ \vdash_{BF} [p_1] C [\varepsilon: q_1] \quad \vdash_{BF} [p_2] C [\varepsilon: q_2] \]

\[ \vdash_{BF} [p_1 \lor p_2] C [\varepsilon: q_1 \lor q_2] \]

- **Disj on paper**: to combine multiple triples
- **ConsB on paper**: to weaken pre or strengthen post
When are $\text{Disj}$ and $\text{ConsB}$ used in BUX?

$\vdash_{BF} [p_1] C [\epsilon: q_1] \vdash_{BF} [p_2] C [\epsilon: q_2]$  
$\vdash_{BF} [p_1 \lor p_2] C [\epsilon: q_1 \lor q_2]$  

- **Disj on paper**: to combine multiple triples  
- **ConsB on paper**: to weaken pre or strengthen post  
- **Disj in Pulse**: rarely used; pre-post correspondence tracked (distinct summaries)
When are \textbf{Disj} and \textbf{ConsB} used in BUX?

\[
\begin{align*}
\vdash_{BF} [p_1] C [\varepsilon: q_1] & \quad \vdash_{BF} [p_2] C [\varepsilon: q_2] \\
\vdash_{BF} [p_1 \lor p_2] C [\varepsilon: q_1 \lor q_2] & \quad \vdash_B [p] C [\varepsilon: q_1 \lor q_2] \\
& \quad \vdash_B [p] C [\varepsilon: q_1]
\end{align*}
\]

- **Disj on paper**: to combine multiple triples
- **ConsB on paper**: to weaken pre or strengthen post
- **Disj in Pulse**: rarely used; pre-post correspondence tracked (distinct summaries)
- **ConsB in Pulse**: mainly to drop disjuncts (i.e. forget summaries)
Indexed Disjuncts

\[
P, Q \in \mathbb{N} \rightarrow \mathcal{P}(\text{States})
\]

\[Q \equiv \bigvee_{i \in \text{dom}(Q)} q_i\]
Indexed Disjuncts

\[ P, Q \in \mathbb{N} \to \mathcal{P}(\text{States}) \quad Q \equiv \bigvee_{i \in \text{dom}(Q)} q_i \]

\[ \vdash \triangleright [P] C [\varepsilon : Q] \text{ iff } \text{dom}(P) = \text{dom}(Q) \land \]

\[ \forall i \in \text{dom}(P). \vdash \triangleright [P(i)] C [\varepsilon : Q(i)] \]
Unified BUX/FUX Framework

\[ \vdash_{BF} [p_1] C [\varepsilon: q_1] \quad \vdash_{BF} [p_2] C [\varepsilon: q_2] \]

\[ \vdash_{BF} [p_1 \lor p_2] C [\varepsilon: q_1 \lor q_2] \]

\[ \vdash_{BF} [P_1] C [\varepsilon: Q_1] \quad \vdash_{BF} [P_2] C [\varepsilon: Q_2] \]

\[ \vdash_{BF} [P_1 \uplus P_2] C [\varepsilon: Q_1 \uplus Q_2] \]
Unified BUX/FUX Framework

\[
\begin{align*}
\vdash_{BF} [p_1] C [\varepsilon: q_1] & \quad \vdash_{BF} [p_2] C [\varepsilon: q_2] \\
\implies & \\
\vdash_{BF} [p_1 \lor p_2] C [\varepsilon: q_1 \lor q_2]
\end{align*}
\]

(ConsB)
\[
\begin{align*}
p' \subseteq p & \quad \vdash_{B} [p'] C [\varepsilon: q'] \\
& \quad q' \subseteq q \\
\implies & \\
\vdash_{B} [p] C [\varepsilon: q]
\end{align*}
\]

(ConsF)
\[
\begin{align*}
p' \supseteq p & \quad \vdash_{F} [p'] C [\varepsilon: q'] \\
& \quad q' \subseteq q \\
\implies & \\
\vdash_{F} [p] C [\varepsilon: q]
\end{align*}
\]

\[
\begin{align*}
\vdash_{BF} [P_1] C [\varepsilon: Q_1] & \quad \vdash_{BF} [P_2] C [\varepsilon: Q_2] \\
\implies & \\
\vdash_{BF} [P_1 \cup P_2] C [\varepsilon: Q_1 \cup Q_2]
\end{align*}
\]

\[
\begin{align*}
\vdash_{BF} [P] C [\varepsilon: Q] & \quad I \subseteq \text{dom}(P) \\
\implies & \\
\vdash_{BF} [P \downarrow I] C [\varepsilon: Q \downarrow I]
\end{align*}
\]
Unified BUX/FUX Framework

Can use Pulse **as is!**

☞ **Extend** Pulse w. **divergence rules**
Theorem 1.

\[ \vdash_B [p] \ C \ [\varepsilon: q] \land \minpre(p, C, q) \Rightarrow \vdash_F [p] \ C \ [\varepsilon: q] \]
Theorem 1.

$$\vdash_B [p] C [\varepsilon : q] \land \text{minpre}(p, C, q) \Rightarrow \vdash_F [p] C [\varepsilon : q]$$

where $$\text{minpre}(p, C, q) \iff \forall p'. \vdash_B [p'] C [\varepsilon : q] \Rightarrow p' \not\in p$$
Theorem 1.
\[ \vdash_B \lbrack p \rbrack C \lbrack \varepsilon : q \rbrack \land \text{minpre}(p, C, q) \Rightarrow \vdash_F \lbrack p \rbrack C \lbrack \varepsilon : q \rbrack \]

where \[ \text{minpre}(p, C, q) \iff \forall p'. \vdash_B \lbrack p' \rbrack C \lbrack \varepsilon : q \rbrack \Rightarrow p' \not\in p \]

Theorem 2.
\[ \vdash_F \lbrack p \rbrack C \lbrack \varepsilon : q \rbrack \land \text{minpost}(p, C, q) \Rightarrow \vdash_B \lbrack p \rbrack C \lbrack \varepsilon : q \rbrack \]

where \[ \text{minpost}(p, C, q) \iff \forall q'. \vdash_F \lbrack p \rbrack C \lbrack \varepsilon : q' \rbrack \Rightarrow q' \not\in q \]
The goal is to find bugs!

“Most program analysis & verification research seems confused about the ultimate goal of software defect detection. The main practical usefulness of such techniques is the ability to find bugs, not to report that no bugs have been found.”

Patrice Godefroid, 2005
The goal is to find bugs!

“Most program analysis & verification research seems confused about the ultimate goal of software defect detection. The main practical usefulness of such techniques is the ability to find bugs, not to report that no bugs have been found.”

Patrice Godefroid, 2005

The soundness of bugs is what matters!

Thank You for Listening!

azalea@imperial.ac.uk

SoundAndComplete.org

@azalearaad