The paper in one slide

Problem: Automatic inference of preconditions

Define: What is a precondition?
- **Sufficient** precondition: if it holds, the function is correct
- **Necessary** precondition: if it does not hold, the function is definitely wrong

When automatic inference is considered, **only necessary preconditions make sense**

- Sufficient preconditions impose too large a burden to callers
- Necessary preconditions are easy to explain to users

Implementation in Clousot
- Precision improvements 9% to 21%
- Extremely low false positive ratio

Example

```csharp
int Example1(int x, object[] a)
{
    if (x >= 0)
    {
        return a.Length;
    }
    return -1;
}
```

- **Sufficient** precondition: `a != null`
  - Too strong for the caller
  - No runtime errors when `x < 0` and `a == null`

Clousot users complained about it “wrong preconditions”

Example

```csharp
void Example2(object[] a)
{
    Contract.Requires(a != null);
    for (var i = 0; i < a.Length; i++)
    {
        a[i] = F(a[i]);
        if (NonDet())
            return;
    }
}
```

- **Sufficient** precondition: `false`
  - It may fail, so eliminate all runs

- **Necessary** precondition: `0 < a.Length`
  - If `a.Length == 0` it will always fail

Necessary precondition is weaker than the weakest precondition!!!
Program semantics

Program traces: $T = G \cup B \cup I$
- $G = \text{good}$ traces, terminating in a good state
- $B = \text{bad}$ traces, terminating in an assertion violation

Assertions:
- Language-induced: division by zero, null pointers, buffer overrun …
- User-supplied annotations: assertions, preconditions, postconditions, object invariants
- $I = \text{infinite}$ traces, non-termination

Notation: $X(s)$ are the traces starting with $s$

Semantics

Necessary and sufficient

In $S \implies N$ we say that
- $S$ is a sufficient condition for $N$
- $N$ is a necessary condition for $S$

For a program $P$
- A condition $S$ is sufficient if its truth ensures that $P$ is correct
- A condition $N$ is necessary if its falsehood ensures $P$ is incorrect

Sufficient Preconditions
Weakest (liberal) preconditions

Provide sufficient preconditions guaranteeing partial correctness:

\[ \text{wlp}(P, \text{true})(s_0) \iff (B(s_0) = \varnothing) \]

Drawbacks of wlp for the automatic inference of preconditions:

1. With loops, there is no algorithm to compute \( \text{wlp}(P, \text{true}) \)
   Solution in deductive verification: Use loop invariant
2. Inferred preconditions are sufficient but not the weakest anymore
   Under-approximation of loops
3. Sufficient preconditions rule out good runs
   Callers should satisfy a too strong condition

Example

```csharp
int Sum(int[] xs)
{
    Contract.Requires(xs != null);
    int sum = 0;
    for (var i = 0; i < xs.Length; i++)
        sum += xs[i];
    Contract.Assert(sum >= 0);
    return sum;
}
```

Overflows are not an error
Ex. \( \text{Sum}([-2147483639, 2147483638, -1]) = 19 \)
In deductive verification, provide loop invariant
Which is the weakest precondition?
The method itself
Sufficient preconditions:

\[ \forall \ s. 0 \leq s[i] < \text{MaxInt}/xs \text{Length} \]
\[ \text{or } xs \text{ Length} = 3 \land xs[0] + xs[1] = 0 \land xs[2] \geq 0 \]
\[ \text{or } \ldots \]

Consequences

Sufficient preconditions impose too large a burden to the caller
They just ensure the correctness of the callee
Not practical in a realistic setting
Users complained about “wrong” preconditions
“Wrong preconditions” ≠ sufficient preconditions

Under-approximation of wlp

Formally, with loop invariants, we compute a sufficient condition \( S \):

\[ S(s_0) \implies \text{wlp}(P, \text{true})(s_0) \]

Which is equivalent to

\[ \lnot(s_0) \implies [S(s_0) \implies G(s_0) \neq \varnothing] \]

So that it may exists some initial state \( s \) such that

\[ \lnot s \land G(s) \neq \varnothing \]

i.e., \( s \) does not satisfy \( S \), but it does not lead to a bad state
**Necessary preconditions**

If the program terminates in a good state for $s_0$, then $N(s_0)$ should hold:

$$[I(s_0) = \mathord{\top}] \implies [G(s_0) \neq \mathord{\top} \implies N(s_0)]$$

Equivalently

$$[I(s_0) = \mathord{\top}] \implies [-N(s_0) \implies (G(s_0) = \mathord{\top} \land B(s_0) \neq \mathord{\top})]$$

i.e., if $N$ does not hold, either
- The program diverges, or
- The program reaches a bad state

**Strongest (liberal) necessary precondition:**

$$\text{snp}(\text{P, true})(s_0) \not\equiv [G(s_0) \neq \mathord{\top}]$$

**Strongest necessary preconditions**

$$\text{snp}(\text{P, true})(s_0) \not\equiv [G(s_0) \neq \mathord{\top} \land B(s_0) \neq \mathord{\top}] = [G(s_0) \neq \mathord{\top}]$$

**Comparison, ignoring non-termination**

<table>
<thead>
<tr>
<th>Weakest sufficient preconditions</th>
<th>Strongest necessary preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(s_0)$</td>
<td>$G(s_0)$</td>
</tr>
<tr>
<td>$G(s_0)$</td>
<td>$N(s_0)$</td>
</tr>
<tr>
<td>$N(s_0)$</td>
<td>$B(s_0)$</td>
</tr>
<tr>
<td>$B(s_0)$</td>
<td>$\not\equiv$</td>
</tr>
<tr>
<td>$\not\equiv$ true</td>
<td>$\not\equiv$ true</td>
</tr>
<tr>
<td>$\not\equiv$ false</td>
<td>$\not\equiv$ false</td>
</tr>
</tbody>
</table>

**Approximation of necessary conditions**

Static analyses to infer an error condition $E$ such that

$$E(s_0) \implies [G(s_0) = \mathord{\top} \land B(s_0) \neq \mathord{\top}]$$

i.e., $E$ is sufficient to guarantee the presence of definite errors or non-termination

$E$ is an **under-approximation** of the error semantics

The negation, $\neg E = N$ is weaker than the strongest (liberal) necessary precondition:

$$G(s_0) \neq \mathord{\top} \lor B(s_0) = \mathord{\top} \implies \neg E(s_0)$$
Main Algorithm

Iterate until stabilization
For each method m
    Analyze m using the underlying static analysis
    Collect proof obligations A
    Use the analysis to prove the assertions in A
Let W \subseteq A be the set of warnings
If W \neq \emptyset then
    Infer necessary preconditions for assertions in W
    Simplify the inferred preconditions
    Propagate the necessary preconditions to the callers of m

Static analyses for the inference

- **All-Paths** precondition analysis
  - Hoists unmodified assertions to the code entry
- **Conditional-path** precondition analysis
  - Hoist assertions by taking into account assignments and tests
  - Use dual-widening for loops
    - Dual-widening under-approximates its arguments
- **Quantified** precondition analysis
  - Deal with unbounded data structures

Examples

```java
int FirstOccurrence(int[] a) { int i = 0; while (a[i] != 3) i++; return i; }
```

All-paths infers
- a != null

Conditional-paths also infers
- a.Length > 0 \land \{a[0] != 3 \implies a.Length >1 \}
- a.Length > 0 \land \{a[0] != 3 \implies a.Length >1 \}

Quantified infers
- \exists j \in [0, a.Length]. a[j] == 3

Details in the paper
Simplification

We can infer many preconditions for a given method. **Simplification** allows reducing them.

Key to scalability:
- Pretty print preconditions for the user

Simplification is a set of **rewriting rules** to iterate to fixpoint.

Examples:
- $P, [b \Rightarrow a], [-b \Rightarrow a] \rightarrow P, [true \Rightarrow a]$
- $P, [true \Rightarrow a] \rightarrow P, a$

---

Implementation

---

Code Contracts static checker

**Clousot/cccheck** static analyzer for .NET:
- Downloaded more than 80,000 times
- Use preconditions/postconditions to reason on method calls
- Suggest and propagates inferred preconditions and postconditions

Users **complained** about sufficient preconditions.

Starting point for this work

---

User experience

---

![User interface screenshot](image.png)
Experimental results

Un-annotated code (.NET base libraries)
- All paths analysis
  - Infer 18,643 preconditions
    - Simplification removes >32%
- Conditional path analysis
  - Infers 28,623 preconditions
    - Simplification removes >24%

Similar results for partially annotated code (Facebook C# SDK)
- Conditional path analysis is more precise but up to 4x slower than all-paths analysis
- Because of inferred disjunctions

Precision

- Number of inferred preconditions is not a good measure
  - We are interested in the precision, i.e., fewer methods with warnings
    - Precision gain is between 9% (framework libraries) and 21% (Facebook C# SDK)
  - Missing preconditions public surface are errors
  - The library does not defend against "bad inputs"
  - Onmscorlib, the core library of .Net, we found 129 new bugs
  - Only **one** false positive
    - Because of exception handling in Clousot

Conclusions

Sic transit gloria mundi

- The violation of a necessary precondition guarantees a **definite error**
- When automatically inferring preconditions, **only necessary preconditions make sense**
  - Sufficient preconditions are **too strict for callers**
- Advantages
  - Easy to **explain** to the users
  - Provide **chain leading** to errors
  - **No false positives**
- Implemented, and used in a widely downloaded tool (Clousot/cccheck)