Bilateral Proofs of Concurrent Programs

Jayadev Misra

Department of Computer Science
University of Texas at Austin

WG 2.3, Istanbul
March 23, 2015
This talk is about:

- Verification of concurrent programs.
- With concurrent programs of full generality.
- With emphasis on specification and their composition.
A simple Example: Podelski et. al., POPL 2015

Given global integer variable $g$ and local variables $x_i$ of thread $i$

$$x_0 := g; \quad g := g + x_0 \quad \cdots x_i := g; \quad g := g + x_i \quad \cdots$$

Show that if $g$ is positive initially, it remains positive.
A proof in my theory

\{g > 0\}
\ x_i := g;
\{g > 0 \land x_i > 0\}
g := g + x_i
\{g > 0\}

Claim: Proof is complete.

Observation: Construct an annotation of the program in which every assertion is of the form \( p \land I \), \( p \) is local to the program point and \( I \) is any fixed predicate.

Then the annotation is valid.
Epoch-making developments in Verification

- Inductive assertions, by Floyd and Hoare.
- Non-interference, by Owicki and Gries.
- Rely-Guarantee, Cliff Jones.
From assertions to Properties: Unity

- Simplify program structure: $\text{loop } \langle g \rightarrow s \rangle \mid \text{loop } \langle g' \rightarrow s' \rangle \mid \cdots$

- Each $\langle g \rightarrow s \rangle$ is a guarded action.

- Prove program properties, not assertions at program points:
  - If $g$ is initially positive, it stays positive.
  - A resource is never granted unless requested.
  - A request for a resource is eventually granted.

- Specification of a component is a set of properties.

- Specifications compose.
Goal of the current work

- Extend Unity to apply to arbitrary concurrent programs.
- Extend rely-guarantee to prove both safety and progress properties.
- Do it all effectively within a single framework.
Commutative Associative Fold of a bag

*put* and *get* are atomic operations on bag $s$.

*put* is non-blocking, *get* blocking.

$$f_1 = \text{get}(x); \text{get}(y); \text{put}(x \oplus y)$$

$$f_k = f_1 \parallel f_{k-1}$$

Show that with $n$ items in $s$ initially:

- the execution of $f_{n-1}$ terminates, and

- leaves $s$ with one item, the fold of all the original items.

Another definition:

$$f_1 = (\text{get}(x) \parallel \text{get}(y)); \text{put}(x \oplus y)$$
Commutative Associative Fold of a bag

*put* and *get* are atomic operations on bag $s$.

*put* is non-blocking, *get* blocking.

$$f_1 = get(x); get(y); put(x \oplus y)$$

$$f_k = f_1 \parallel f_{k-1}$$

Show that with $n$ items in $s$ initially:

- the execution of $f_{n-1}$ terminates, and
- leaves $s$ with one item, the fold of all the original items.

Another definition:

$$f_1 = (get(x) \parallel get(y)); put(x \oplus y)$$
Commutative Associative Fold of a bag

*put* and *get* are atomic operations on bag *s*.

*put* is non-blocking, *get* blocking.

\[
f_1 = get(x); get(y); put(x \oplus y)
\]

\[
f_k = f_1 \parallel f_{k-1}
\]

Show that with *n* items in *s* initially:

- the execution of \( f_{n-1} \) terminates, and

- leaves *s* with one item, the fold of all the original items.

Another definition:

\[
f_1 = (get(x) \parallel get(y)); put(x \oplus y)
\]
Observations about the problem

- Desired: Respect the recursive program structure in proof.

- The result does not hold for $f_n$. There is deadlock.

- Interplay between sequential and concurrent aspects.

- Entire code is not available.
What we need

• Specification $spec_k$ of $f_k$, $k \geq 1$.

• Show from its code that $f_1$ satisfies $spec_1$.

• Show that $spec_k$ can be deduced from $spec_1 \parallel spec_{k-1}$.

• Show that the required properties can be deduced from $spec_{n-1}$.
Summary of the Theory

- Programs with arbitrary interleaving of sequential and concurrent.
- Construct assertions and program properties simultaneously.
- Properties are created from assertions.
- Assertions are strengthened using properties; bilateral proofs.
- Properties are also deduced compositionally.
- Both safety and progress properties considered.
Program Model

A component is one of:

- **Action**: Uninterruptible, terminating code, e.g.: \( x := x + 1 \), put, get.

- **Sequencer**: Combines components using sequential constructs, e.g.:
  \( s; t, \text{ if } b \text{ then } s \text{ else } t, \text{ while } b \text{ do } s. \)

- **Fork**: \( f \| g, f \text{ and } g \text{ are components.} \)
  \( f \| g \| h = (f \| g) \| h = f \| (g \| h) \)

**Execution**:

- **Sequential components** follow their execution rules.

- **Fork**: start all components simultaneously.
  
  Terminates when they all do.
Program Model

A component is one of:

- **Action**: Uninterruptible, terminating code, e.g.: \( x := x + 1 \), *put*, *get*.

- **Sequencer**: Combines components using sequential constructs, e.g.:
  \[
  s; t, \text{ if } b \text{ then } s \text{ else } t, \text{ while } b \text{ do } s.
  \]

- **Fork**: \( f \parallel g \), \( f \) and \( g \) are components.
  
  \[
  f \parallel g \parallel h = (f \parallel g) \parallel h = f \parallel (g \parallel h)
  \]

Execution:

- Sequential components follow their execution rules.

- Fork: start all components simultaneously.
  Terminates when they all do.
For component $f$, predicates $I$ and $E$, and sets of predicates $P$ and $Q$:

- a specification is: $\{I \mid P\} \ f \ \{Q \mid E\}$.

- Call this an augmented assertion.

- Proof rules for augmented assertions. Derived from regular proof rules.
Meaning of \( \{I \mid P\} \ f \ \{Q \mid E\} \)

- If program \( f \) is started in an \( I \)-state, its execution either terminates in an \( E \)-state or never terminates.

- If the environment preserves every predicate in \( P \), the predicates in \( Q \) are preserved by \( f \).

Notes:

- Predicates in \( P \) and \( Q \) need not be stable in either the environment or \( f \).

- Sequential \( \{I\} \ f \ \{E\} \) is: \( \{I \mid \{\text{ALL}\}\} \ f \ \{\\} \mid E\} \).

- \( \{\mid P\} \ f \ \{Q \mid\} \) is: \( \{\text{true} \mid P\} \ f \ \{Q \mid \text{true}\} \).

- Closed Execution has \( \text{ALL} \) for \( P \).
Meaning of $\{I \mid P\} \ f \ \{Q \mid E\}$

- If program $f$ is started in an $I$-state, its execution either terminates in an $E$-state or never terminates.

- If the environment preserves every predicate in $P$, the predicates in $Q$ are preserved by $f$.

Notes:

- Predicates in $P$ and $Q$ need not be stable in either the environment or $f$.

- Sequential $\{I\} f \ \{E\}$ is: $\{I \mid \{ALL\}\} \ f \ \{\{} \mid E\}$.

- $\{\mid P\} f \ \{Q \mid\}$ is: $\{true \mid P\} \ f \ \{Q \mid true\}$.

- Closed Execution has $ALL$ for $P$. 
Technical Contributions

- \((I, P)\) annotation of a program.
- Proof rules for augmented assertions, Jones-style.
- Extensions of \(Q\) to include general (Unity-style) properties.
- Proof rules for properties, Unity-style.