Bilateral Proofs of Concurrent Programs

Jayadev Misra

Department of Computer Science
University of Texas at Austin

WG 2.3, Istanbul
March 23–27, 2015

A Hoare-style Proof Rule

$$\frac{\{I\}\ s\ \{I'\},\ \{I'\}\ t\ \{E\}}{\{I\}\ s;t\ \{E\}}$$

- A proof rule is a composition rule for specifications.
- The proof rules suggest constructing hierarchical proofs, from codes and/or specifications.
- Users need only program specification, not code.

A Hoare-style Proof Rule

$$\frac{\{I\}\ s\ \{I'\},\ \{I'\}\ t\ \{E\}}{\{I\}\ s;t\ \{E\}}$$

- A proof rule is a composition rule for specifications.
- The proof rules suggest constructing hierarchical proofs, from codes and/or specifications.
- Users need only program specification, not code.

Concurrent Program Proofs

- Shambles, generally.
- Bright spot is model checking.
- Model checking is not sufficient.

A very difficult program to prove

$$\{x = 0\}$$

$$x := x + 1 \;\|\; x := x + 2$$

$$\{x = 3\}$$

Owicki's Thesis

- Construct annotation of each sequential component.

$$\{x = 0\}$$

$$(\{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\}$$

$$\| \ \{x = 0 \vee x = 1\}\ x := x + 2\ \{x = 2 \vee x = 3\})$$

$$\{(x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)\}$$

$$\{x = 3\}$$

- Show that the proofs don't interfere, e.g.,

$$\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\}\ \ x := x + 2\ \ \{x = 0 \vee x = 2\}$$

Owicki's Thesis

- Construct annotation of each sequential component.

$$\{x = 0\}$$

$$(\ \{x = 0 \lor x = 2\}\ x := x + 1\ \{x = 1 \lor x = 3\}$$

$$\| \ \{x = 0 \lor x = 1\}\ x := x + 2\ \{x = 2 \lor x = 3\})$$

$$\{(x = 1 \lor x = 3) \land (x = 2 \lor x = 3)\}$$

$$\{x = 3\}$$

- Show that the proofs don't interfere, e.g.,

$$\{(x = 0 \lor x = 2) \land (x = 0 \lor x = 1)\}\ \ x := x + 2\ \ \{x = 0 \lor x = 2\}$$

Assessment

- First real proof technique for concurrent programs.
- Works well for small tightly-coupled components.
- Not scalable.
- Needs program code.
- No notion of a specification.

Rely-Guarantee of Cliff Jones

- Replace non-interference proofs by checks against stable predicates.
- First scalable proof technique for concurrent programs.
- Notion of specification and composition.
- Limited to safety properties.

Unity by Chandy and Misra

- Simplify program structure: $\mathit{loop}\ \langle g \quad \longrightarrow \ s \rangle \ [\!] \ \mathit{loop}\ \langle g' \quad \longrightarrow \ s' \rangle \ [\!] \ \cdots$

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

- Prove program properties, not assertions at program points:
  - A resource is never granted unless requested.
  - A request for a resource is eventually granted.

- Specification is a set of properties.
  Stable predicates are properties.

- Composition rules for specification are given.

Implementations

- Some succeses: Telephony, Control systems
- Model checkers:
  UV (Markus Kaltenbach, UT),
  Murϕ (David Dill, Stanford),
  Siemens (Jorge Cuellar),
  SAL
- Implementations in other logics:
  Boyer-Moore prover, Larch, HOL, Coq, Isabelle/ZF
  DisCo (based on Unity) in PVS
  CommUNITY workbench

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 initially in s:

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

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 initially in s:

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

Observations about the problem

- Desired: Respect the recursive program structure in proof.
- Note interplay between sequential and concurrent aspects.
- Entire code is not available.
- Safety: Finally s has one item, the fold of the original items. Easy.
- Progress: f_{n-1} terminates. Hard.

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

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.:

  $s;\ t$, **if** b **then** s **else** t, **while** b **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: starts 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.:

  $s;\ t$, **if** b **then** s **else** t, **while** b **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: starts all components simultaneously.

  Terminates when they all do.

Effective Execution

- An action may have an optional guard: $\langle g \quad \longrightarrow \; \alpha \rangle$.
- A blocking action, e.g. *get*, has an implicit guard.
- Non-blocking actions, e.g. $x := x + 1$, have guard *true*.
- Execution of a guarded action is:
  - Effective: the guard holds and the action execution completes.
  - Ineffective: the guard does not hold, execution completes and nothing changes.
- An action execution always terminates, never blocks.

Annotation

- Traditional proof rules for actions and sequencer.

- Proof rule for Fork:

$$\frac{(\forall i :: \{p_i\}\ c_i\ \{q_i\})}{\{\forall i :: p_i\}\ (\ \| i :: c_i)\ \{\forall i :: q_i\}}$$

The annotation is not necessarily valid.

Definition of Valid Annotation

- Action: Annotation is always valid.
- Sequencer: annotation is valid if each direct subcomponent's is.
- Fork, $f \parallel g$: annotation is valid if f's and g's are, plus (OG-condition):
  - For every $\alpha \in f$ and $\beta \in g$,
  - where pre_α is the precondition of α in the annotation,
  - $\{pre_\alpha \wedge pre_\beta\}\ \alpha\ \{pre_\beta\}$ holds, and
  - dually for action β.

Stable Predicate

- Given a valid annotation of f, α preserves predicate p means:

  $\{pre_\alpha \wedge p\}\ \alpha\ \{p\}$.

- p stable in f: every action of f preserves p in the given valid annotation.

- Ineffective execution preserves all p.

- Stable predicates are closed under conjunction and disjunction.

Environment

- A sequential program has no concurrently executing environment.
- In $f \,[\!]\, g$, f is g's environment and conversely.
- In most cases, code of the environment is not available, e.g. Unix.
- Determine properties of a component from the specification of the environment.

Demon

- Treat the environment of f as a demon or adversary.
- It may modify the global state.

P-Demon

- P a set of predicates. The demon preserves all predicates in P.
- Any demon preserves all local predicates of f.

  P^*: conjuctive, disjunctive closure of P with the local predicates.
- Closed execution of f: Demon preserves all predicates, i.e., the demon is *skip*.

Specification

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.
- Augmented proof rules are derived from the regular proof rules.

Later: Generalize Q to assert both safety and progress properties.

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 is a P-demon, the predicates in Q are preserved by f.

Notes:

- Predicates in P and Q may not be stable in f or the demon.

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

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

Proof Rule for Action

- Original inference:

$$\{I\}\ \alpha\ \{E\}$$

- Augmented proof rule:

$$\{I\}\ \alpha\ \{E\},$$

$$I \in P^*,\ E \in P^*,$$

For all q in Q: α preserves q, i.e.,

$$\{I \wedge q\}\ \alpha\ \{q\}$$

$$\{I \mid P\}\ \ \alpha\ \ \{Q \mid E\}$$

Proof Rule for Sequencer

Component f a sequencer with direct subcomponents f_i:

- Original proof rule:

$$\frac{(\forall i :: \{I_i\}\ f_i\ \{E_i\})}{\{I\}\ f\ \{E\}}$$

- Augmented proof rule:

$$\frac{(\forall i :: \{I_i \mid P_i\}\ f_i\ \{Q_i \mid E_i\})}{\{I \mid \cup_i P_i\}\ f\ \{\cap_i Q_i \mid E\}}$$

Proof Rule for Fork

- Original proof rule:

$$\frac{\{I\}\ f\ \{E\},\ \{I'\}\ g\ \{E'\}}{\{I \wedge I'\}\ f \parallel g\ \{E \wedge E'\}}$$

- Augmented proof rule:

$$\frac{\begin{array}{c}\{I \mid P\}\ f\ \{Q \mid E\},\ \{I' \mid P'\}\ g\ \{Q' \mid E'\},\\ P \subseteq Q',\ P' \subseteq Q \qquad \text{Linkage}\end{array}}{\{I \wedge I' \mid P \cup P'\}\ f \parallel g\ \{Q \cap Q' \mid E \wedge E'\}}$$

Justification for the Proof Rules

Claim: Given $\{I \mid P\}$ f $\{Q \mid E\}$,

- f has a valid annotation in which the entry and exit assertions are I and E, and every assertion is from P^*, including I and E.

- Any q, $q \in Q$ is stable according to the given annotation.

The claim is proved by induction on the program structure.

Proving augmented assertions directly

To prove $\{I \mid P\}\ f\ \{Q \mid E\}$, construct an annotation of f in which:

- Entry, exit assertions are I and E.
- Every assertion is from P^*.
- Every q in Q is stable in the given annotation.

Note: Non-interference holds by construction.

Basic Inference Rules

- Given $\{I \mid P\}\ f\ \{Q \mid E\}$
  - (lhs expansion) $\{I \mid P \cup P'\}\ f\ \{Q \mid E\}$
  - (rhs contraction) $\{I \mid P\}\ f\ \{Q \cap Q' \mid E\}$
- (Conjunction)

$$\frac{\{I \mid P\}\ f\ \{Q \mid E\},\quad \{I' \mid P'\}\ f\ \{Q' \mid E'\}}{\{I \wedge I' \mid P \cup P'\}\ f\ \{Q \cup Q' \mid E \wedge E'\}}$$

Stable, Co-stable, Bistable Predicates

Given $\{I \mid P\}\ f\ \{Q \mid E\}$, for f a predicate in:

- Q is stable,
- P is co-stable,
- in both P and Q is bistable.

Bistable Inference Rule:

$$\frac{\{I \mid P\}\ f\ \{Q \mid E\}, \quad \text{bistable } r}{\{I \wedge r \mid P\}\ f\ \{Q \mid E \wedge r\}}$$

Returning to Andreas

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

$$
\begin{array}{c}
\{g > 0\} \\
x_i := g; \\
\{g > 0 \wedge x_i > 0\} \\
g := g + x_i \\
\{g > 0\} \\
\ldots
\end{array}
$$

Observation: Construct an annotation of a program in which every assertion is of the form $p \wedge I$, p is local to the program point and I is any fixed predicate. Then the annotation is valid.

Proof: By induction on the structure of the program.

Commutative Associative Fold of a bag

put and get are atomic operations on bag s.

put is non-blocking, get blocking.

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

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

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.

Specification of Commutative Associative Fold

Introduce auxiliary variable q_k in f_k:
the bag of items acquired from s and as yet unfolded.

$$f_1 :: \text{initially } q_1 = \{\}$$

$$get(x) \;\&\; q_1 := q_1 \cup \{x\};$$

$$get(y) \;\&\; q_1 := q_1 \cup \{y\};$$

$$put(x \oplus y) \;\&\; q_1 := q_1 - \{x, y\}$$

$f_k = f_1 \;[\!]\; f_{k-1}$, where $q_k = q_1 \cup q_{k-1}$.

Proof of Commutative Associative Fold

Prove for all k, $k \geq 1$, and constant D:

$\{q_k = \{\} \mid \phi\}$ f_k $\{\oplus(s \cup q_k) = D \mid q_k = \{\}\}$.

Proof by induction on k. For $k = 1$:

$$\{q_1 = \{\}\}$$

$$get(x)\ \&\ q := q \cup \{x\};$$

$$\{q_1 = \{x\}\}$$

$$get(y)\ \&\ q := q \cup \{y\};$$

$$\{q_1 = \{x, y\}\}$$

$$put(x \oplus y)\ \&\ q := q - \{x, y\}$$

$$\{q_1 = \{\}\}$$

Check **stable** $\oplus(s \cup q_1) = D$ against the annotation.

Inductive Proof

$$\{q_1 = \{\} \mid \phi\}\ f_1\ \{\oplus(s \cup q_1) = D \mid q_1 = \{\}\}$$

, proved

$$\{q_1 = \{\} \mid \phi\}\ f_1\ \{\oplus(s \cup q_{k+1}) = D \mid q_1 = \{\}\}$$

, $q_{k+1} = q_1 \cup q_k$, q_k constant in f_1 (1)

$$\{q_k = \{\} \mid \phi\}\ f_k\ \{\oplus(s \cup q_k) = D \mid q_k = \{\}\}$$

, inductive hypothesis

$$\{q_k = \{\} \mid \phi\}\ f_k\ \{\oplus(s \cup q_{k+1}) = D \mid q_k = \{\}\}$$

, $q_{k+1} = q_1 \cup q_k$, q_1 constant in f_k (2)

$$\{q_1 = \{\}\ \wedge q_k = \{\} \mid \phi\}\ f_{k+1}\ \{\oplus(s \cup q_{k+1}) = D \mid q_1 = \{\}\ \wedge q_k = \{\}\}$$

, Composition rule (linkage satisfied)

$$\{q_{k+1} = \{\} \mid \phi\}\ f_{k+1}\ \{\oplus(s \cup q_{k+1}) = D \mid q_{k+1} = \{\}\}$$

, $q_{k+1} = q_1 \cup q_k$

Establish Exit Condition

$$\{q_k = \{\} \mid \phi\} \;\; f_k \;\; \{\oplus(s \cup q_k) = D \mid q_k = \{\}\}$$

, Proved

$$\{q_k = \{\} \mid ALL\} \;\; f_k \;\; \{\oplus(s \cup q_k) = D \mid q_k = \{\}\}$$

, lhs expansion to closed execution

$$\{q_k = \{\} \;\wedge \oplus(s \cup q_k) = D \mid ALL\}$$
$$f_k$$
$$\{\oplus(s \cup q_k) = D \mid \; \oplus(s \cup q_k) = D \;\wedge q_k = \{\}\}$$

, $\oplus(s \cup q_k) = D$ is bistable

$$\{\oplus s = D\} \;\; f_k \;\; \{\oplus s = D\}$$

, simplifying

Counting Completed Threads

Introduce auxiliary variable nc_k in f_k:
the number of completed threads.

$$f_1 :: \text{initially } q_1, nc_1 = \{\}, 0$$

$$get(x)\ \&\ q_1 := q_1 \cup \{x\};$$

$$get(y)\ \&\ q_1 := q_1 \cup \{y\};$$

$$put(x \oplus y)\ \&\ q_1, nc_1 := q_1 - \{x, y\}, nc_1 + 1$$

$f_k = f_1 \ [\!]\ f_{k-1}$, where $q_k = q_1 \cup q_{k-1}$ and $nc_k = nc_1 + nc_{k-1}$.

Specification: A safety property about nc

Prove for all k, $k \geq 1$, and constant C:
$\{nc_k = 0 \mid \phi\}$ f_k $\{|s| + |q| + nc_k = C \mid nc_k = k\}$.

- Proof by induction on k. Similar to the previous proof.
- Establish exit condition similarly:

  $\{\oplus s = D, |s| = C\}$ f_k $\{\oplus s = D, |s| + k = C\}$

- Does not prove that f_k halts.

General Theory

- So far, only stable predicates as properties.
- In practice, more general safety and progress properties are needed.
- Allow Q to include more general properties that can be proved from a valid annotation.

Properties Introduced in Unity

For predicates p and q:

- p co q: now p implies q after the next step.
- p en q: now p implies eventually q and p until then.
- $p \longmapsto q$: now p implies eventually q.

Some typical Unity Inference rules

- $$\frac{p \text{ co } q \text{ in } f, \quad p \text{ co } q \text{ in } g}{p \text{ co } q \text{ in } f \; [\!]\; g}$$

- $$\frac{p \text{ en}^{+} q \text{ in } f, \quad p \wedge \neg q \text{ co } p \vee q \text{ in } g}{p \text{ en}^{+} q \text{ in } (f \; [\!]\; g)}$$

- $$\frac{p \mapsto q \text{ in } f, \quad q \mapsto r \text{ in } f}{p \mapsto r \text{ in } f}$$

Integration with Unity

- Meaning of:

$$\frac{p \ \textcolor{red}{\mathrm{en}^+} \ q \ \text{ in } \ f, \qquad p \wedge \neg q \ \textcolor{red}{\mathrm{co}} \ p \vee q \ \text{ in } \ g}{p \ \textcolor{red}{\mathrm{en}^+} \ q \ \text{ in } \ (f \parallel g)}$$

- as an augmented assertion is:

$$\frac{\{|\, P\} \ \textcolor{red}{f} \ \{Q \cup \{p \ \textcolor{red}{\mathrm{en}^+} \ q\}|\}, \quad \{|\, P'\} \ \textcolor{red}{g} \ \{Q' \cup \{p \wedge \neg q \ \textcolor{red}{\mathrm{co}} \ p \vee q\} \,|\}, \quad P' \subseteq Q,\ P \subseteq Q'}{\{|\, P \cup P'\} \ \textcolor{red}{f \parallel g} \ \{Q \cap Q' \cup \{p \ \textcolor{red}{\mathrm{en}^+} \ q\} \,|\},}$$

Overview of integration of Unity

- Allow P and Q to include co properties.
  Earlier composition rules apply.

- Allow Q to include en and $\mapsto$ properties.
  Earlier composition rules used for linkage.
  New composition rules apply for each combinator.

- A typical rule:

$$\frac{p \ \mathrm{en}^{+} \ q \ \mathrm{in} \ f, \qquad p \wedge \neg q \ \mathrm{co} \ p \vee q \ \mathrm{in} \ g}{p \ \mathrm{en}^{+} \ q \ \mathrm{in} \ (f \parallel g)}$$

Finite P-demon

- Meaning of $\{\!| P\}\ f\ \{Q |\!\}$ with safety properties:
  If the environment is a P-demon, the predicates in Q are preserved by f.

  The property holds for the interleaved execution of f with P-demon.

- For a progress property, this interpretation is restrictive.

  $\{\!| P\}\ f\ \{p\ \mathrm{en}^{+}\ q |\!\}$, for example, now means:

  $p\ \mathrm{en}^{+}\ q$ holds for the interleaved execution of f with a P-demon that takes only a finite number of steps.

- This interpretation permits:
  - Deducing progress properties of f in a closed execution.
  - Specification composition.
  - Establishing strong progress properties.

Finite P-demon

- Meaning of $\{\!| P\} \ f \ \{Q |\!\}$ with safety properties:
  If the environment is a P-demon, the predicates in Q are preserved by f.

  The property holds for the interleaved execution of f with P-demon.

- For a progress property, this interpretation is restrictive.

  $\{\!| P\} \ f \ \{ p \ \ \mathrm{en}^+ \ \ q |\!\}$, for example, now means:

  $p \ \ \mathrm{en}^+ \ \ q$ holds for the interleaved execution of f with a P-demon that takes only a finite number of steps.

- This interpretation permits:
  - Deducing progress properties of f in a closed execution.
  - Specification composition.
  - Establishing strong progress properties.

Finite P-demon

- Meaning of $\{\!|\ P\}\ f\ \{Q\ |\!\}$ with safety properties:
  If the environment is a P-demon, the predicates in Q are preserved by f.

  The property holds for the interleaved execution of f with P-demon.

- For a progress property, this interpretation is restrictive.

  $\{\!|\ P\}\ f\ \{\ p\ \text{en}^+\ q|\!\}$, for example, now means:

  $p\ \text{en}^+\ q$ holds for the interleaved execution of f with a P-demon that takes only a finite number of steps.

- This interpretation permits:
  - Deducing progress properties of f in a closed execution.
  - Specification composition.
  - Establishing strong progress properties.

Progress Proof: Commutative Associative Fold

f_1 :: initially $q_1, nc_1 = \{\}, 0$

$get(x)$ & $q_1 := q_1 \cup \{x\}$;

$get(y)$ & $q_1 := q_1 \cup \{y\}$;

$put(x \oplus y)$ & $q_1, nc_1 := q_1 - \{x, y\}, nc_1 + 1$

$f_k = f_1 \; [\!] \; f_{k-1}$, where $q_k = q_1 \cup q_{k-1}$ and $nc_k = nc_1 + nc_{k-1}$.

Progress Proof: Commutative Associative Fold; Contd.

Show in f_k: if initially $|s| > k$ then eventually $q_k = \{\}$ and $nc_k = k$.

Formally, $\{|s| > k\}$ f_k $\{true \mapsto q_k = \{\} \wedge nc_k = k \,|\}$