Inference rule for Concurrent Execution

An incorrect attempt

A naive approach is to say that the concurrent execution of statements establishes postconditions of all the statements.

We might try the following inference rule

$$\frac{\begin{array}{l}\vdash P \Rightarrow P_0 \wedge P_1 \\ \vdash \{P_0\}\ S\ \{Q_0\} \\ \vdash \{P_1\}\ T\ \{Q_1\} \\ \vdash Q_0 \wedge Q_1 \Rightarrow Q\end{array}}{\vdash \{P\}\ \mathbf{co}\ S\ //\ T\ \mathbf{oc}\ \{Q\}} \text{(Co) [Incorrect!]}$$

It allows us to prove correct programs correct. For example

$\{x = X \wedge y = Y\}$

co

$\{x = X\}\ \langle x := x + 1;\rangle\ \{x = X + 1\}$

//

$\{y = Y\}\ \langle y := y + 1;\rangle\ \{y = Y + 1\}$

oc

$\{x = X + 1 \wedge y = Y + 1\}$

But it also allows us to prove incorrect programs are correct!

$\{x = X \wedge y = Y\}$
co
$\qquad \{y = Y\}\ \langle x := y + 1;\rangle\ \{x = Y + 1\}$
//
$\qquad \{x = X\}\ \langle y := x + 1;\rangle\ \{y = X + 1\}$
oc
$\{x = Y + 1 \wedge y = X + 1\}$

Why? Consider the following interleaving

0	$\{y = Y\}$	$\{x = X\}$
	$\langle x := y + 1;\rangle$	
1	$\{x = Y + 1\}$	
		$\langle y := x + 1;\rangle$
2		$\{y = X + 1\}$

At point 2, the precondition $x = X$ no longer true.

The assignment $x := y + 1$ **interferes with** the assertion $x = X$.

Thus the inference rule above *is not sound*.

The solution

Instead of Hoare triples, we use *proof outlines*.

A **proof outline** is a triple $\{P\}S\{Q\}$ where statement S is annotated by internal assertions. Each substatement of $\{P\}S\{Q\}$ is preceded by an assertion.

The precondition of each statement S is denoted $\mathrm{pre}(S)$.

Redo the logic using proof outlines instead of Hoare triples.

Assignment is as before.

$$\frac{\vdash P \Rightarrow Q_{x \leftarrow E}}{\vdash \{P\}\ x := E\ \{Q\}}\text{(Assign)}$$

Sequential composition requires an internal assertion

$$\frac{\begin{array}{c}\vdash \{P\}\ S\ \{Q\} \\ \vdash \{Q\}\ T\ \{R\}\end{array}}{\vdash \{P\}\ S\ \{Q\}\ T\ \{R\}}\text{(Seq)}$$

So does iteration

$$\frac{\begin{array}{c}\vdash P \wedge E \Rightarrow Q \\ \vdash \{Q\}\ S\ \{P\} \\ \vdash P \wedge \neg E \Rightarrow R\end{array}}{\vdash \{P\}\ \mathbf{while}(E)\ \{Q\}\ S\ \{R\}}\text{(While)}$$

Now we can state a rule for concurrent composition

$$\frac{\begin{array}{l} \vdash P \Rightarrow P_0 \wedge P_1 \\ \vdash \{P_0\}\ S\ \{Q_0\} \\ \vdash \{P_1\}\ T\ \{Q_1\} \\ \vdash Q_0 \wedge Q_1 \Rightarrow Q \\ S \text{ does not interfere with } \{P_1\}\, T \{Q_1\} \\ T \text{ does not interfere with } \{P_0\}\, S \{Q_0\} \end{array}}{\{P\}\ \mathbf{co}\ \{P_0\}\ S\, \{Q_0\} \mathbin{//} \{P_1\}\ T\, \{Q_1\}\, \mathbf{oc}\ \{Q\}}\text{(Co)}$$

Interference

An atomic action a interferes with an assertion P if it could cause P to change from true to false.

But a will only be executed from a state where $pre(a)$ is true, so we may assume $\text{pre}(a)$ is initially true.

So a **does not interfere with** P iff

$$\vdash \{P \wedge \text{pre}(a)\}\ a\ \{P\}$$

A critical assertion of $\{P_0\}\ T\ \{Q_0\}$ is an assertion not inside an await statement.

S **does not interfere with** $\{P_0\}\ T\ \{Q_0\}$ iff no atomic action in S interferes with any critical assertion in $\{P_0\}\ T\ \{Q_0\}$.

Techniques for avoiding interference

Disjoint variables

if no variable in an assertion is in the write set of the action, there is no interference

[[Need example]]

Weakened assertions

Consider

$x = 0$
co
        ## $x = 0$
        $\langle\ x := x + 1;\ \rangle$
        ## $x = 1$
//
        ## $x = 0$
        $\langle\ x := x + 2;\ \rangle$
        ## $x = 2$
oc
$x = 1 \wedge x = 2$

There is interference:

$$\nvdash \{x = 0\}\ x := x + 2;\ \{x = 0\}$$

We can use a weaker precondition to start each process

$x = 0$
co
    ## $x = 0 \lor x = 2$
    $\langle\ x := x + 1;\ \rangle$
    ## $?$
//
    ## $x = 0 \lor x = 1$
    $\langle\ x := x + 2\ ;\ \rangle$
    ## $?$
oc
$?$

No interference, so far:

$$\begin{aligned} \vdash\ \ &\{(x = 0 \lor x = 2) \land (x = 0 \lor x = 1)\} \\ &x := x + 2 \\ &\{x = 0 \lor x = 2\} \end{aligned}$$

and

$$\begin{array}{ll} \vdash & \{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\} \\ & x := x + 1 \\ & \{x = 0 \vee x = 1\} \end{array}$$

Now complete the outline with the strongest possible postconditions, & check for interference.

$x = 0$

co

    ## $x = 0 \vee x = 2$
    $\langle\ x := x + 1;\ \rangle$
    ## $x = 1 \vee x = 3$

//

    ## $x = 0 \vee x = 1$
    $\langle\ x := x + 2\ ;\ \rangle$
    ## $x = 2 \vee x = 3$

oc

$x = 3$

Global invariants

Global invariants are implied by the over-all precondition, and preserved by all atomic actions.

If G is a global invariant we write

	for	
## P		## P
## Global invariant G		**co**
co		## $G \wedge L_0$
## L_0		a_0
a_0		## $G \wedge L_1$
## L_1		a_1
a_1		## $G \wedge L_2$
## L_2		//
//		## $G \wedge M_0$
## M_0		b_0
b_0		## $G \wedge M_1$
## M_1		b_1
b_1		## $G \wedge M_2$
## M_2		**oc**
oc		## Q
## Q		

Now we need to check

- **Global invariance:** that G is implied by P and preserved by each action.

$$P \Rightarrow G$$
$$\{G \wedge L_i\}\ a_i\ \{G\}$$
$$\{G \wedge M_i\}\ b_i\ \{G\}$$

- **Remaining Noninterference:** the remaining parts of non-interference

$$\{L_i \wedge G \wedge M_j\}\ \ b_j\ \ \{L_i\}$$
$$\{M_i \wedge G \wedge L_j\}\ \ a_i\ \ \{M_i\}$$

- **Remaining Local Correctness:** the remaining parts of local correctness

$$P \Rightarrow L_0 \wedge M_0$$
$$\{L_i \wedge G\}\ \ a_i\ \ \{L_{i+1}\}$$
$$\{M_i \wedge G\}\ \ b_i\ \ \{M_{i+1}\}$$
$$G \wedge L_2 \wedge M_2 \Rightarrow Q$$

When all the local assertions L_i and M_i use only variables not changed by the other process, the second step is not needed (by disjoint variables): global invariance implies freedom from interference.

Example: Synchronizing loops (barrier synchronization)

Assume that $A0$ and $A1$ are independent of $\{c0, c1, s0, s1\}$

$P : c0 = c1 = s0 = s1;$
global inv. $G0 : s0 \leq c1 + 1$
global inv. $G1 : s1 \leq c0 + 1$

## $P0 : s0 = c0 \leq c1$ 	
while(true) {
    ## $P0 : s0 = c0 \leq c1$
    $s0 \mathrel{+}= 1;$
    ## $Q0 : s0 = c0 + 1$
    $A0$
    ## $Q0 : s0 = c0 + 1$
    $c0 \mathrel{+}= 1;$
    ## $R0 : s0 = c0$
    $\langle \textbf{await}(\ c0 \leq c1\ )\rangle$
}
	## $P1 : s1 = c1 \leq c0$
	while(true) {
	    ## $P1 : s1 = c1 \leq c0$
	    $s1 \mathrel{+}= 1;$
	    ## $Q1 : s1 = c1 + 1$
	    $A1$
	    ## $Q1 : s1 = c1 + 1$
	    $c1 \mathrel{+}= 1;$
	    ## $R1 : s1 = c1$
	    $\langle \textbf{await}(\ c1 \leq c0\ )\rangle$
	}

Let $G = G0 \wedge G1$. What we need to show is:

Global invariance (dv means the proof is by disjoint variables)

- $\vdash P \Rightarrow G0$
- $\vdash \{G \wedge P0\}\ \ s0 \mathrel{+}= 1;\ \{G0\}$
- $\vdash \{G \wedge Q0\}\ \ c0 \mathrel{+}= 1;\ \{G0\}$ (dv)
- $\vdash \{G \wedge P1\}\ \ s1 \mathrel{+}= 1;\ \{G0\}$ (dv)
- $\vdash \{G \wedge Q1\}\ \ c1 \mathrel{+}= 1;\ \{G0\}$

Remaining Noninterference

- $\vdash \{P0 \wedge G \wedge P1\}\ \ s1 \mathrel{+}= 1;\ \{P0\}$ (dv)
- $\vdash \{P0 \wedge G \wedge Q1\}\ \ c1 \mathrel{+}= 1;\ \{P0\}$
- $\vdash \{Q0 \wedge G \wedge P1\}\ \ s1 \mathrel{+}= 1;\ \{Q0\}$ (dv)
- $\vdash \{Q0 \wedge G \wedge Q1\}\ \ c1 \mathrel{+}= 1;\ \{Q0\}$ (dv)
- $\vdash \{R0 \wedge G \wedge P1\}\ \ s1 \mathrel{+}= 1;\ \{R0\}$ (dv)
- $\vdash \{R0 \wedge G \wedge Q1\}\ \ c1 \mathrel{+}= 1;\ \{R0\}$ (dv)

Remaining local correctness

- $\vdash P \Rightarrow P0$
- $\vdash \{G \wedge P0\}\ \ s0 \mathrel{+}= 1;\ \{Q0\}$
- $\vdash \{G \wedge Q0\}\ \ c1 \mathrel{+}= 1;\ \{R0\}$
- $\vdash \{G \wedge R0\}\ \ \langle \mathbf{await}(\ c0 \le c1\ )\rangle\ \ \{P0\}$

And symmetrically for postconditions $G1$, $P1$, $Q1$, $R1$.

Ghost Variables

Ghost variables (aka **thought variables**, **dummy variables**, and **auxiliary variables**) are variables that are used for the purpose of proof, but do not need to be implemented.

Consider

$x = 0$
co
    ## $x = 0$
    $\langle\ x := x + 1;\ \rangle$
    ## $x = 1$
//
    ## $x = 0$
    $\langle\ x := x + 1;\ \rangle$
    ## $x = 1$
oc
$x = 1$

Again there is interference. Note that weakening preconditions to $\{x = 0 \vee x = 1\}$ is to no avail.

Introduce integer ghost variables a and b, initially 0.

---

int $a := 0$, $b := 0$;
$x = 0 \wedge a = 0 \wedge b = 0$
Global Inv: $a + b = x$
co

    ## $a = 0$
    $\langle\ x := x + 1;\ \ a := a + 1;\ \rangle$
    ## $a = 1$

//

    ## $b = 0$
    $\langle\ x := x + 1;\ \ b := b + 1;\ \rangle$
    ## $b = 1$

oc
$x = 2$

Since a and b are each only in the write set of one process, there is no interference.

That $x = 2$ finally, follows from the global invariant, together with $a = 1 \wedge b = 1$.

Await statements

Await statements force a delay until an assertion is true before proceeding.

$$\frac{\vdash \{P \wedge E\}\ S\ \{Q\}}{\vdash \{P\}\ \langle \mathbf{await}(\ E\ )\ S\rangle\ \{Q\}}\text{(Await)}$$

Two techniques:

1. 'Hide' assertions via mutual exclusion.
2. Strengthen the precondition via conditional synchronization.

Hide assertions

Derived inference rule

$$\frac{\vdash \{P\}\ S\ \{Q\}}{\vdash \{P\}\ \langle S\rangle\ \{Q\}}\text{(Mutual Exclusion)}$$

On the left the global invariant is interfered with.

```
int size := 0 ;
int front := 0 ;
int back := 0 ;
## Global Inv:
##     size = back − front
co
    ...
    ⟨front := front +1;⟩
    ⟨size := size - 1 ;⟩
    ...
oc
```

```
int size := 0 ;
int front := 0 ;
int back := 0 ;
## Global Inv:
##     size = back − front
co
    ...
    ⟨front := front +1;
    ## size = back − front − 1
    size := size - 1 ;⟩
    ...
oc
```

On the right the intermediate state is hidden in the atomic action.

Use conditional synchronization

Derived inference rule

$$\frac{\vdash P \wedge E \Rightarrow Q}{\vdash \{P\}\ \langle \textbf{await}(\ E\ )\ \rangle\ \{Q\}}\text{(Conditional synchronization)}$$

Example: On the left, $s := s - 1$does not respect the global invariant.

$$\nvdash \{s \geq 0\}\ s := s - 1;\ \{s \geq 0\}$$

int $s := 0$; ## Global Inv: $s \geq 0$ **co** ... $\langle s := s - 1; \rangle$ ... // ... $\langle s := s + 1; \rangle$... **oc**	**int** $s := 0$; ## Global Inv: $s \geq 0$ **co** ... $\langle \textbf{await}(s > 0); \rangle$ ## $s > 0$ $\langle s := s - 1; \rangle$ ... // ... $\langle s := s + 1; \rangle$... **oc**

Solution. Use conditional synchronization to strengthen the precondition to $s > 0$.

Data Refinement

Introducing one set of variable to represent another.

We do data refinement in three steps:

- Augment: Add new variables and an invariant establishing their relationship to the preexisting variables.
- Transform: Change the algorithm to use the new variables rather than certain preexisting variables.
- Diminish: Demote any variables no longer needed to the status of ghosts

Example of data refinement

Recall the producer and consumer with a shared buffer.

int buf
int $p := 0$; # The number of things produced.
int $c := 0$; # The number of things consumed.
Global inv: $0 \leq c \leq p \leq c + 1$

process Producer {
    while ($true$) {
        $\langle$**await**($p = c$)$\rangle$
        ## $p = c$
        buf := *next value* ;
        $p := p + 1$; } }
	process Consumer {
	    while (*true*) {
	        $\langle$**await**($p > c$)$\rangle$
	        ## $p = c + 1$
	        use buf
	        $c := c + 1$; } }

Augment with a boolean b.

b says whether $p = c$ or $p = c + 1$

---

int *buf* ;
bool $b := \mathit{true}$;
int $p := 0, c := 0$;
Global inv: $0 \leq c \leq p \leq c + 1$
Global inv: $b = (p = c)$

---

process Producer {
    while (true) {
        $\langle$**await**($p = c$)$\rangle$
        ## $(p = c) \wedge b$
        buf := *next value* ;
        $\langle p, b := p + 1, \mathit{false}; \rangle$
    }
}

process Consumer {
    while (true) {
        $\langle$**await**($p > c$)$\rangle$
        ## $(p = c + 1) \wedge \neg b$
        use buf
        $\langle c, b := c + 1, \mathit{true}; \rangle$
    }
}

Transform

Rewrite so that p and c are no longer needed to compute the result.

---

int buf;
bool $b := true$;
int $p := 0, c := 0$;
Global inv: $0 \le c \le p \le c + 1$
Global inv: $b = (p = c)$

```
process Producer {
    while (true) {
        ⟨await( b ) ⟩
        ## (p = c) ∧ b
        buf := next value ;
        ⟨p, b := p + 1, false; ⟩
    }
}
```

```
process Consumer {
    while (true) {
        ⟨await( not b )⟩
        ## (p = c + 1) ∧ ¬b
        use buf
        ⟨c, b := c + 1, true; ⟩
    }
}
```

Diminish

Demote p and c to the status of ghost variables.

---

int buf;
bool $b := true$;
int $p := 0, c := 0$;
Global inv: $0 \leq c \leq p \leq c + 1$
Global inv: $b = (p = c)$

---

process Producer {
    while ($true$) {
      $\langle$**await**(b);$\rangle$
      ## $(p = c) \wedge b$
      $buf :=$ *next value* ;
      $\langle p, b := p + 1, false; \rangle$
    }
}
	process Consumer {
	    while ($true$) {
	      $\langle$**await**(not b);$\rangle$
	      ## $(p = c + 1) \wedge \neg b$
	      use buf
	      $\langle c, b := c + 1, true; \rangle$
	    }
	}

Now p and c are used only in the reasoning.

Safety Properties

A *property* characterizes a set of executions.

A program *satisfies* a property if every possible execution (history) of the program is in the set characterized by the property.

Safety property: Something must always be true (set of executions in which no undesirable states, or sequences of states, occur).

- e.g.,
  - partial correctness — program never enters a state that is both terminated and not described by the postcondition.
  - absence of deadlock (doesn't reach a deadlock state)
  - mutual exclusion
- finitely refutable: if a safety property does not hold, there is a finite history that demonstrates this.
- characterized by negation of 'bad' things

Proving Safety

Let B characterize undesirable states

- Show that for any critical assertion C, $C \Rightarrow \neg B$, or
- Show that $\neg B$ is a global invariant.
  - $\neg B$ is $true$ initially,
  - $\{pre(S) \wedge \neg B\}\ \ S\ \ \{\neg B\}$ is valid for all program statements S

Special Case: Exclusion of configurations

---

co # process 1
    ... { P } $\langle a \rangle$ ...
// # process 2
    ... { Q } $\langle b \rangle$ ...
oc

If

- P and Q are not interfered with, and
- $P \wedge Q \equiv false$ (i.e. $\neg P \vee \neg Q$)

then statements a and b can never both be about to be executed.

Liveness Properties

Something must eventually become true.

- e.g.,
  - termination: process must eventually stop
  - absence of starvation (processes must eventually get serviced)
- not finitely refutable: any execution can be extended to satisfy the property.

Fairness

Fairness assumptions are assumptions about the nature of the scheduler.

Often some fairness assumption is required in order for (liveness) properties to be provable.

An atomic action is *eligible* if it could be executed next

scheduling policy — determines which eligible action will be executed next.

---

bool $continue$ = $true$;
co
    while ($continue$) **skip**
//
    $continue$:= $false$;
oc

Degrees of fairness:

unconditional: Every unconditional atomic action that is eligible is executed eventually.

weak: Unconditionally fair & every conditional atomic action for which the condition is continuously true (until it is executed), will eventually be executed.

strong: Unconditionally fair & every conditional atomic action for which the condition is true infinitely often, will eventually be executed.

---

bool $continue$:= $true$, try := $false$;

co
    while ($continue$) {
        try := $true$;
        try := $false$; }
//
    $\langle$ **await(** try **)** $continue$:= $false$; $\rangle$
oc

Under weak fairness, the above may not terminate.

Under strong fairness, it must terminate eventually.

Use fairness:

Often to show liveness properties, one must make use of fairness assumptions.