

INF4140 - Models of concurrency

Program Analysis, lecture 6

Høsten 2014

3.10.2014



Program Analysis

- PL lets us *express* and *prove* properties about programs
- *Formulas* are on the form

“triple”

$$\{ P \} S \{ Q \}$$

- S : program statement(s)
- P and Q : assertions over program states
- P : Pre-condition
- Q : Post-condition

If we can use PL to prove some property of a program, then this property will hold for all executions of the program

$$\frac{\{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}} \text{Seq}$$

$$\frac{\{P \wedge B\} S \{Q\} \quad P \wedge \neg B \Rightarrow Q}{\{P\} \text{ if } B \text{ then } S \{Q\}} \text{Cond}'$$

$$\frac{\{I \wedge B\} S \{I\}}{\{I\} \text{ while } B \text{ do } S \{I \wedge \neg B\}} \text{While}$$

$$\frac{\{P\} S \{Q\} \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\{P'\} S \{Q'\}} \text{Consequence}$$

- Cannot control the execution in the same manner as for if statements
 - Cannot tell from the code how many times the loop body will be executed

$$\{ y \geq 0 \} \text{ while } (y > 0) \ y := y - 1$$

- Cannot speak about the state after the first, second, third iteration
- **Solution:** Find an assertion I that is maintained by the loop body
 - Loop invariant: express a property preserved by the loop
- Often hard to find suitable loop invariants
 - This course is *not* an exercise in finding complicated invariants

$$\frac{\{ I \wedge B \} S \{ I \}}{\{ I \} \text{ while } B \text{ do } S \{ I \wedge \neg B \}} \text{ While}$$

Can use this rule to reason about the more general case:

$$\{ P \} \text{ while } B \text{ do } S \{ Q \}$$

where

- P need not be the loop invariant
- Q need not match $(I \wedge \neg B)$ syntactically

Combine While-rule with Consequence-rule to prove:

- **Entry:** $P \Rightarrow I$
- **Loop:** $\{ I \wedge B \} S \{ I \}$
- **Exit:** $I \wedge \neg B \Rightarrow Q$

While rule: example

$\{0 \leq n\} k := 0; \{k \leq n\} \text{ while } (k < n) k := k + 1; \{k = n\}$

Composition rule splits a proof in two: assignment and loop.

Let $k \leq n$ be the loop invariant

- **Entry:** $k \leq n$ follows from itself
- **Loop:**

$$\frac{k < n \Rightarrow k + 1 \leq n}{\{k \leq n \wedge k < n\} k := k + 1 \{k \leq n\}}$$

- **Exit:** $(k \leq n \wedge \neg(k < n)) \Rightarrow k = n$

Rule for await

$$\frac{\{ P \wedge B \} S; \{ Q \}}{\{ P \} \langle \text{await}(B) S \rangle \{ Q \}} \text{Await}$$

Remember: we are reasoning about safety properties

- Termination is assumed/ignored
- the rule does not speak about waiting or progress

Concurrent execution

Assume two statements S_1 and S_2 such that:

$$\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \text{and} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}$$

Note: to avoid further complications right now: S_i 's are enclosed into “ \langle atomic brackets \rangle ”.

First attempt for a co...oc rule in PL:

$$\frac{\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}}{\{ P_1 \wedge P_2 \} \text{co}\langle S_1 \rangle \parallel \langle S_2 \rangle \text{oc} \{ Q_1 \wedge Q_2 \}} \text{Par}$$

Example (Problem with this rule)

$$\frac{\{ x = 0 \} \langle x := x + 1 \rangle \{ x = 1 \} \quad \{ x = 0 \} \langle x := x + 2 \rangle \{ x = 2 \}}{\{ x = 0 \} \text{co}\langle x := x + 1 \rangle \parallel \langle x := x + 2 \rangle \text{oc} \{ x = 1 \wedge x = 2 \}}$$

but this conclusion is not true: the postcondition should be $x = 3!$

Concurrent execution

Assume two statements S_1 and S_2 such that:

$$\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \text{and} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}$$

Note: to avoid further complications right now: S_i 's are enclosed into “ \langle atomic brackets \rangle ”.

First **attempt** for a $\text{co} \dots \text{oc}$ rule in PL:

$$\frac{\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}}{\{ P_1 \wedge P_2 \} \text{co} \langle S_1 \rangle \parallel \langle S_2 \rangle \text{oc} \{ Q_1 \wedge Q_2 \}} \text{Par}$$

Example (Problem with this rule)

$$\frac{\{ x = 0 \} \langle x := x + 1 \rangle \{ x = 1 \} \quad \{ x = 0 \} \langle x := x + 2 \rangle \{ x = 2 \}}{\{ x = 0 \} \text{co} \langle x := x + 1 \rangle \parallel \langle x := x + 2 \rangle \text{oc} \{ x = 1 \wedge x = 2 \}}$$

but this conclusion is not true: the postcondition should be $x = 3!$

Concurrent execution

Assume two statements S_1 and S_2 such that:

$$\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \text{and} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}$$

Note: to avoid further complications right now: S_i 's are enclosed into “ \langle atomic brackets \rangle ”.

First **attempt** for a $\text{co} \dots \text{oc}$ rule in PL:

$$\frac{\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}}{\{ P_1 \wedge P_2 \} \text{co} \langle S_1 \rangle \parallel \langle S_2 \rangle \text{oc} \{ Q_1 \wedge Q_2 \}} \text{Par}$$

Example (Problem with this rule)

$$\frac{\{ x = 0 \} \langle x := x + 1 \rangle \{ x = 1 \} \quad \{ x = 0 \} \langle x := x + 2 \rangle \{ x = 2 \}}{\{ x = 0 \} \text{co} \langle x := x + 1 \rangle \parallel \langle x := x + 2 \rangle \text{oc} \{ x = 1 \wedge x = 2 \}}$$

but this conclusion is not true: the postcondition should be $x = 3!$

Interference problem

$$S_1 \quad \{ x = 0 \} \langle x := x + 1 \rangle \{ x = 1 \}$$

$$S_2 \quad \{ x = 0 \} \langle x := x + 2 \rangle \{ x = 2 \}$$

- execution of S_2 interferes with pre- and postconditions of S_1
 - The assertion $x = 0$ need not hold when S_1 starts execution
- execution of S_1 interferes with pre- and postconditions of S_2
 - The assertion $x = 0$ need not hold when S_2 starts execution

Solution: weaken the assertions to account for the other process:

$$S_1 \quad \{ x = 0 \vee x = 2 \} \langle x := x + 1 \rangle \{ x = 1 \vee x = 3 \}$$

$$S_2 \quad \{ x = 0 \vee x = 1 \} \langle x := x + 2 \rangle \{ x = 2 \vee x = 3 \}$$

Interference problem

$$S_1 \quad \{ x = 0 \} \langle x := x + 1 \rangle \{ x = 1 \}$$

$$S_2 \quad \{ x = 0 \} \langle x := x + 2 \rangle \{ x = 2 \}$$

- execution of S_2 interferes with pre- and postconditions of S_1
 - The assertion $x = 0$ need not hold when S_1 starts execution
- execution of S_1 interferes with pre- and postconditions of S_2
 - The assertion $x = 0$ need not hold when S_2 starts execution

Solution: *weaken* the assertions to account for the other process:

$$S_1 \quad \{ x = 0 \vee x = 2 \} \langle x := x + 1 \rangle \{ x = 1 \vee x = 3 \}$$

$$S_2 \quad \{ x = 0 \vee x = 1 \} \langle x := x + 2 \rangle \{ x = 2 \vee x = 3 \}$$

Interference problem

Now we can try to apply the rule:

$$\frac{\begin{array}{l} \{ x = 0 \vee x = 2 \} \langle x := x + 1 \rangle \{ x = 1 \vee x = 3 \} \\ \{ x = 0 \vee x = 1 \} \langle x := x + 2 \rangle \{ x = 2 \vee x = 3 \} \end{array}}{\{ PRE \} \text{co} \langle x := x + 1 \rangle \parallel \langle x := x + 2 \rangle \text{oc} \{ POST \}}$$

where:

$$PRE : (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)$$

$$POST : (x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)$$

Interference problem

Now we can try to apply the rule:

$$\frac{\begin{array}{l} \{ x = 0 \vee x = 2 \} \langle x := x + 1 \rangle \{ x = 1 \vee x = 3 \} \\ \{ x = 0 \vee x = 1 \} \langle x := x + 2 \rangle \{ x = 2 \vee x = 3 \} \end{array}}{\{ PRE \} \text{ co } \langle x := x + 1 \rangle \parallel \langle x := x + 2 \rangle \text{ oc } \{ POST \}}$$

where:

$$PRE : (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)$$

$$POST : (x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)$$

which gives:

$$\{ x = 0 \} \text{ co } \parallel x = x + 1 \parallel \langle x := x + 2 \rangle \text{ oc } \{ x = 3 \}$$

Assume $\{ P_i \} S_i \{ Q_i \}$ for all S_1, \dots, S_n

$\{ P_i \} S_i \{ Q_i \}$ are *interference free*

$\{ P_1 \wedge \dots \wedge P_n \} \text{co } S_1 \parallel \dots \parallel S_n \text{oc } \{ Q_1 \wedge \dots \wedge Q_n \}$

Interference freedom

A process **interferes** with (the specification of) another process, if its execution changes the values of the assertions^a of the other process.

^aOnly “critical assertions” considered

- assertions inside awaits: not endangered
- **critical assertions** or **critical conditions**: assertions outside await statement bodies.¹

¹More generally one could say: outside mutex-protected sections.

Interference freedom

- S : statement some process, with pre-condition $pre(S)$
- C : critical assertion in **another** process
- S **does not interfere** with C , if

$$\{ C \wedge pre(S) \} S \{ C \}$$

is derivable in PL (= theorem).

“ C is *invariant* under the execution of the other process”

$$\frac{\{ P_1 \} S_1 \{ Q_1 \} \quad \{ P_2 \} S_2 \{ Q_2 \}}{\{ P_1 \wedge P_2 \} \text{co } S_1 \parallel S_2 \text{oc } \{ Q_1 \wedge Q_2 \}}$$

Four interference freedom requirements:

$$\begin{array}{ll} \{ P_2 \wedge P_1 \} S_1 \{ P_2 \} & \{ P_1 \wedge P_2 \} S_2 \{ P_1 \} \\ \{ Q_2 \wedge P_1 \} S_1 \{ Q_2 \} & \{ Q_1 \wedge P_2 \} S_2 \{ Q_1 \} \end{array}$$

"Avoiding" interference: Weakening assertions

$$S_1 : \{ x = 0 \} \langle x := x + 1; \rangle \{ x = 1 \}$$

$$S_2 : \{ x = 0 \} \langle x := x + 2; \rangle \{ x = 2 \}$$

Here we have interference, for instance the precondition of S_1 is not maintained by execution of S_2 :

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

is not true

However, after **weakening**:

$$S_1 : \{ x = 0 \vee x = 2 \} \langle x := x + 1 \rangle \{ x = 1 \vee x = 3 \}$$

$$S_2 : \{ x = 0 \vee x = 1 \} \langle x := x + 2 \rangle \{ x = 2 \vee x = 3 \}$$

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

(Correspondingly for the other three critical conditions)

Avoiding interference: Disjoint variables

- V set: global variables referred (i.e. read or written) to by a process
- W set: global variables written to by a process
- *Reference set*: global variables in critical assertions/conditions of one process

S_1 and S_2 : in 2 different processes. **No interference**, if:

- W set of S_1 is disjoint from reference set of S_2
- W set of S_2 is disjoint from reference set of S_1

Alas: variables in a critical condition of one process will often be among the written variables of another

global invariants

- Some conditions. that only refer to global (shared) variables
- Holds initially
- Preserved by all assignments

We avoid interference if critical conditions are on the form $\{I \wedge L\}$ where:

- I is a global invariant
- L only refers to local variables of the considered process

Avoiding interference: Synchronization

- Hide critical conditions
- MUTEX to critical sections

$$\text{co } \dots; S; \dots \parallel \dots; S_1; \{ C \} S_2; \dots \text{ oc}$$

S might interfere with C

Hide the critical condition by a critical region:

$$\text{co } \dots; S; \dots \parallel \dots; \langle S_1; \{ C \} S_2 \rangle; \dots \text{ oc}$$

Example: Producer/ consumer synchronization

Let process Producer deliver data to a Consumer process

$$PC : c \leq p \leq c + 1 \wedge (p = c + 1) \Rightarrow (buf = a[p - 1])$$

Let PC be a *global invariant* of the program:

```
int buf, p := 0; c := 0;
```

```
process Producer {  
  int a[N];...  
  while (p < N) {  
    < await (p = c) ; >  
    buf := a[p];  
    p := p+1;  
  }  
}
```

```
process Consumer {  
  int b[N];...  
  while (c < N) {  
    < await (p > c) ; >  
    b[c] := buf;  
    c := c+1;  
  }  
}
```

Example: Producer/ consumer synchronization

Let process Producer deliver data to a Consumer process

$$PC : c \leq p \leq c + 1 \wedge (p = c + 1) \Rightarrow (buf = a[p - 1])$$

Let PC be a *global invariant* of the program:

```
int buf, p := 0; c := 0;
```

```
process Producer {  
  int a[N];...  
  while (p < N) {  
    < await (p = c) ; >  
    buf := a[p];  
    p := p+1;  
  }  
}
```

```
process Consumer {  
  int b[N];...  
  while (c < N) {  
    < await (p > c) ; >  
    b[c] := buf;  
    c := c+1;  
  }  
}
```

Example: Producer

Loop invariant of Producer:

$$I_P : PC \wedge p \leq n$$

```
process Producer {
  int a[n];
  {  $I_P$  } // entering loop
  while (p < n) {
    < await (p == c); > {  $I_P \wedge p < n$  }
    {  $I_P \wedge p < n \wedge p = c$  }
    {  $I_P p \leftarrow p+1 \text{ buf} \leftarrow a[p]$  }
    {  $I_P p \leftarrow p+1$  }
    {  $I_P$  }
  } {  $I_P \wedge \neg(p < n)$  } // exit loop
     $\Leftrightarrow$  {  $PC \wedge p = n$  }
}
```


Example: Producer

Loop invariant of Producer:

$$I_P : PC \wedge p \leq n$$

```
process Producer {
  int a[n];
  {  $I_P$  }
  while (p < n) {
    < await (p == c); >
    buf = a[p];
    p = p + 1;
  } {  $I_P \wedge \neg(p < n)$  }
     $\Leftrightarrow$  {  $PC \wedge p = n$  }
}
```

// entering loop
{ $I_P \wedge p < n$ }
{ $I_P \wedge p < n \wedge p = c$ }
{ $I_P p \leftarrow p+1 \text{ buf} \leftarrow a[p]$ }
{ $I_P p \leftarrow p+1$ }
{ I_P }
// exit loop

Example: Producer

Loop invariant of Producer:

$$I_P : PC \wedge p \leq n$$

```
process Producer {
  int a[n];
  {  $I_P$  }
  while (p < n) {
    < await (p == c); >
    buf = a[p];
    p = p + 1;
  } {  $I_P \wedge \neg(p < n)$  }
     $\Leftrightarrow$  {  $PC \wedge p = n$  }
}
```

// entering loop
{ $I_P \wedge p < n$ }
{ $I_P \wedge p < n \wedge p = c$ }
{ $I_P p \leftarrow p+1 \text{ buf} \leftarrow a[p]$ }
{ $I_P p \leftarrow p+1$ }
{ I_P }
// exit loop

Example: Producer

Loop invariant of Producer:

$$I_P : PC \wedge p \leq n$$

```
process Producer {
  int a[n];
  {  $I_P$  } // entering loop
  while (p < n) {
    < await (p == c); > {  $I_P \wedge p < n$  }
    {  $I_P \wedge p < n \wedge p = c$  }
    {  $I_P p \leftarrow p+1 \text{ buf} \leftarrow a[p]$  }
    {  $I_P p \leftarrow p+1$  }
    {  $I_P$  }
  } {  $I_P \wedge \neg(p < n)$  } // exit loop
     $\Leftrightarrow$  {  $PC \wedge p = n$  }
}
```

Example: Producer

Loop invariant of Producer:

$$I_P : PC \wedge p \leq n$$

```
process Producer {
  int a[n];
  {  $I_P$  } // entering loop
  while (p < n) {
    < await (p == c); > {  $I_P \wedge p < n$  }
                       {  $I_P \wedge p < n \wedge p = c$  }
                       {  $I_P p \leftarrow p+1 \text{ buf} \leftarrow a[p]$  }
                       {  $I_P p \leftarrow p+1$  }
                       {  $I_P$  }
    buf = a[p];
    p = p + 1;
  } {  $I_P \wedge \neg(p < n)$  } // exit loop
     $\Leftrightarrow$  {  $PC \wedge p = n$  }
}
```

Example: Producer

Loop invariant of Producer:

$$I_P : PC \wedge p \leq n$$

```
process Producer {
  int a[n];
  {  $I_P$  } // entering loop
  while (p < n) {
    < await (p == c); > {  $I_P \wedge p < n$  }
    {  $I_P \wedge p < n \wedge p = c$  }
    {  $I_P p \leftarrow p+1 \text{ buf} \leftarrow a[p]$  }
    {  $I_P p \leftarrow p+1$  }
    {  $I_P$  }
  } {  $I_P \wedge \neg(p < n)$  } // exit loop
     $\Leftrightarrow$  {  $PC \wedge p = n$  }
}
```

Proof obligation:

$$\{ I_P \wedge p < n \wedge p = c \} \Rightarrow \{ I_P \} p \leftarrow p+1 \text{ buf} \leftarrow a[p]$$

Example: Consumer

Loop invariant of Consumer:

$$I_C : PC \wedge c \leq n \wedge b[0 : c - 1] = a[0 : c - 1]$$

```
process Consumer {
  int b[n];
  {I_C} // entering loop
  while (c < n) {
    < await (p > c) ; > {I_C ∧ c < n}
    {I_C} c ← c + 1, b[c] ← buf
    b[c] = buf; {I_C} c ← c + 1
    c = c + 1; {I_C}
  } {I_C ∧ ¬(c < n)} // exit loop
  ⇔ {PC ∧ c = n ∧ b[0 : c - 1] = a[0 : c - 1]}
}
```

Example: Consumer

Loop invariant of Consumer:

$$I_C : PC \wedge c \leq n \wedge b[0 : c - 1] = a[0 : c - 1]$$

```
process Consumer {
  int b[n];
  {I_C} // entering loop
  while (c < n) {
    < await (p > c) ; > {I_C ∧ c < n}
    {I_C} c ← c + 1, b[c] ← buf
    {I_C} c ← c + 1
    b[c] = buf;
    c = c + 1;
  } {I_C ∧ ¬(c < n)} // exit loop
  ⇔ {PC ∧ c = n ∧ b[0 : c - 1] = a[0 : c - 1]}
}
```

Example: Consumer

Loop invariant of Consumer:

$$I_C : PC \wedge c \leq n \wedge b[0 : c - 1] = a[0 : c - 1]$$

```
process Consumer {
  int b[n];
  {I_C} // entering loop
  while (c < n) {
    < await (p > c) ; > {I_C \wedge c < n}
    {I_C} c \leftarrow c + 1, b[c] \leftarrow buf
    b[c] = buf; {I_C} c \leftarrow c + 1
    c = c + 1; {I_C}
  } {I_C \wedge \neg(c < n)} // exit loop
  \Leftrightarrow {PC \wedge c = n \wedge b[0 : c - 1] = a[0 : c - 1]}
}
```


Example: Consumer

Loop invariant of Consumer:

$$I_C : PC \wedge c \leq n \wedge b[0 : c - 1] = a[0 : c - 1]$$

```
process Consumer {
  int b[n];
  {I_C} // entering loop
  while (c < n) {
    < await (p > c) ; > {I_C \wedge c < n}
    {I_C} c \leftarrow c + 1, b[c] \leftarrow buf
    b[c] = buf; {I_C} c \leftarrow c + 1
    c = c + 1; {I_C}
  } {I_C \wedge \neg(c < n)} // exit loop
  \Leftrightarrow {PC \wedge c = n \wedge b[0 : c - 1] = a[0 : c - 1]}
}
```

Example: Consumer

Loop invariant of Consumer:

$$I_C : PC \wedge c \leq n \wedge b[0 : c - 1] = a[0 : c - 1]$$

```
process Consumer {
  int b[n];
  {I_C} // entering loop
  while (c < n) {
    < await (p > c) ; > {I_C ∧ c < n}
    {I_C} c ← c + 1, b[c] ← buf {I_C ∧ c < n ∧ p > c}
    b[c] = buf; {I_C} c ← c + 1
    c = c + 1; {I_C}
  } {I_C ∧ ¬(c < n)} // exit loop
  ⇔ {PC ∧ c = n ∧ b[0 : c - 1] = a[0 : c - 1]}
}
```

Example: Consumer

Loop invariant of Consumer:

$$I_C : PC \wedge c \leq n \wedge b[0 : c - 1] = a[0 : c - 1]$$

```
process Consumer {
  int b[n];
  {I_C} // entering loop
  while (c < n) {
    < await (p > c) ; > {I_C \wedge c < n}
    {I_C} c \leftarrow c + 1, b[c] \leftarrow buf
    b[c] = buf; {I_C} c \leftarrow c + 1
    c = c + 1; {I_C}
  } {I_C \wedge \neg(c < n)} // exit loop
  \Leftrightarrow {PC \wedge c = n \wedge b[0 : c - 1] = a[0 : c - 1]}
}
```

Proof Obligation:

$$\{I_C \wedge c < n \wedge p > c\} \Rightarrow \{I_C\}_{c \leftarrow c + 1, b[c] \leftarrow buf}$$

Example: Producer/Consumer

The final state of the program satisfies:

$$PC \wedge p = n \wedge c = n \wedge b[0 : c - 1] = a[0 : c - 1]$$

which ensures that all elements in a are received and occur in the same order in b

Interference freedom is ensured by the global invariant and `await`-statements

If we combine the two assertions after the `await` statements, we get:

$$I_P \wedge p < n \wedge p = c \wedge I_C \wedge c < n \wedge p > c$$

which gives *false!*

At any time, only one process can be after the await statement!

Example: Producer/Consumer

The final state of the program satisfies:

$$PC \wedge p = n \wedge c = n \wedge b[0 : c - 1] = a[0 : c - 1]$$

which ensures that all elements in a are received and occur in the same order in b

Interference freedom is ensured by the global invariant and `await`-statements

If we combine the two assertions after the `await` statements, we get:

$$I_P \wedge p < n \wedge p = c \wedge I_C \wedge c < n \wedge p > c$$

which gives *false*!

At any time, only one process can be after the await statement!

Monitor invariant

```
monitor name {  
    monitor variables      # shared global variable  
    initialization        # for the monitor's procedures  
    procedures  
}
```

- A monitor invariant (*I*): used to describe the monitor's inner state
- Express relationship between monitor variables
- Maintained by execution of procedures:
 - Must hold after initialization
 - Must hold when a procedure terminates
 -
 -
- Should be as *strong* as possible!

Monitor invariant

```
monitor name {  
    monitor variables      # shared global variable  
    initialization        # for the monitor's procedures  
    procedures  
}
```

- A monitor invariant (*I*): used to describe the monitor's inner state
- Express relationship between monitor variables
- Maintained by execution of procedures:
 - Must hold after initialization
 - Must hold when a procedure terminates
 - Must hold when we **suspend** execution due to a call to wait
 - Can **assume** that the invariant holds *after* wait and when a procedure starts
- Should be as *strong* as possible!

Axioms for signal and continue (1)

Assume that the monitor invariant I and predicate P *do not* mention cv . Then we can set up the following axioms:

$$\begin{array}{ll} \{ I \} \text{wait}(cv) \{ I \} & \\ \{ P \} \text{signal}(cv) \{ P \} & \text{for arbitrary } P \\ \{ P \} \text{signal_all}(cv) \{ P \} & \text{for arbitrary } P \end{array}$$

Monitor solution to reader/writer problem

Verification of the invariant over request_read

$$I : (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

```
procedure request_read() {  
    { I }  
    while (nw > 0) {      { I ∧ nw > 0 }  
        { I } wait(oktoread); { I }  
    }    { I ∧ nw = 0 }  
    { I_{nr ← nr + 1} }  
    nr = nr + 1;  
    { I }  
}
```

$$(I \wedge nw > 0) \Rightarrow I$$

$$(I \wedge nw = 0) \Rightarrow I_{nr \leftarrow nr + 1}$$

Monitor solution to reader/writer problem

Verification of the invariant over request_read

$$I : (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

```
procedure request_read() {  
    { I }  
    while (nw > 0) {      { I ∧ nw > 0 }  
        { I } wait(oktoread); { I }  
    }    { I ∧ nw = 0 }  
    { I_{nr ← nr + 1} }  
    nr = nr + 1;  
    { I }  
}
```

$$(I \wedge nw > 0) \Rightarrow I$$

$$(I \wedge nw = 0) \Rightarrow I_{nr \leftarrow nr + 1}$$

Monitor solution to reader/writer problem

Verification of the invariant over request_read

$$I : (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

```
procedure request_read() {  
    { I }  
    while (nw > 0) {      { I ∧ nw > 0 }  
        { I } wait(oktoread); { I }  
    }    { I ∧ nw = 0 }  
    { I_{nr ← nr+1} }  
    nr = nr + 1;  
    { I }  
}
```

$$(I \wedge nw > 0) \Rightarrow I$$

$$(I \wedge nw = 0) \Rightarrow I_{nr \leftarrow nr+1}$$

Monitor solution to reader/writer problem

Verification of the invariant over request_read

$$I : (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

```
procedure request_read() {  
    { I }  
    while (nw > 0) {      { I ∧ nw > 0 }  
        { I } wait(oktoread); { I }  
    }    { I ∧ nw = 0 }  
        { Inr←nr+1 }  
    nr = nr + 1;  
    { I }  
}
```

$$(I \wedge nw > 0) \Rightarrow I$$

$$(I \wedge nw = 0) \Rightarrow I_{nr \leftarrow nr+1}$$

Monitor solution to reader/writer problem

Verification of the invariant over request_read

$$I : (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

```
procedure request_read() {  
    { I }  
    while (nw > 0) {      { I ∧ nw > 0 }  
        { I } wait(oktoread); { I }  
    }    { I ∧ nw = 0 }  
    { Inr←nr+1 }  
    nr = nr + 1;  
    { I }  
}
```

$$(I \wedge nw > 0) \Rightarrow I$$

$$(I \wedge nw = 0) \Rightarrow I_{nr \leftarrow nr+1}$$

Axioms for Signal and Continue (2)

Assume that the invariant can mention the number of processes in the queue to a condition variable.

- Let $\#cv$ be the number of proc's waiting in the queue to cv .
- The test $\text{empty}(cv)$ thus corresponds to $\#cv = 0$

$\text{wait}(cv)$ is *modelled* as an extension of the queue followed by processor release:

$$\text{wait}(cv) : \{?\} \#cv = \#cv + 1; \{/\} \text{"sleep"} \{/\}$$

by assignment axiom:

$$\text{wait}(cv) : \{/\#cv \leftarrow \#cv + 1; \#cv := \#cv + 1; \{/\} \text{"sleep"} \{/\}$$

Axioms for Signal and Continue (2)

Assume that the invariant can mention the number of processes in the queue to a condition variable.

- Let $\#cv$ be the number of proc's waiting in the queue to cv .
- The test $\text{empty}(cv)$ thus corresponds to $\#cv = 0$

$\text{wait}(cv)$ is *modelled* as an extension of the queue followed by processor release:

$$\text{wait}(cv) : \{?\} \#cv = \#cv + 1; \{I\} \text{"sleep"} \{I\}$$

by assignment axiom:

$$\text{wait}(cv) : \{I \#cv \leftarrow \#cv + 1; \#cv := \#cv + 1; \{I\} \text{"sleep"} \{I\}$$

Axioms for Signal and Continue (2)

Assume that the invariant can mention the number of processes in the queue to a condition variable.

- Let $\#cv$ be the number of proc's waiting in the queue to cv .
- The test $\text{empty}(cv)$ thus corresponds to $\#cv = 0$

$\text{wait}(cv)$ is *modelled* as an extension of the queue followed by processor release:

$$\text{wait}(cv) : \{?\} \#cv = \#cv + 1; \{I\} \text{ "sleep" } \{I\}$$

by assignment axiom:

$$\text{wait}(cv) : \{I \#cv \leftarrow \#cv + 1; \#cv := \#cv + 1; \{I\} \text{ "sleep" } \{I\}$$

Axioms for Signal and Continue (2)

Assume that the invariant can mention the number of processes in the queue to a condition variable.

- Let $\#cv$ be the number of proc's waiting in the queue to cv .
- The test $\text{empty}(cv)$ thus corresponds to $\#cv = 0$

$\text{wait}(cv)$ is *modelled* as an extension of the queue followed by processor release:

$$\text{wait}(cv) : \{?\} \#cv = \#cv + 1; \{I\} \text{ "sleep" } \{I\}$$

by assignment axiom:

$$\text{wait}(cv) : \{I \#cv \leftarrow \#cv + 1; \#cv := \#cv + 1; \{I\} \text{ "sleep" } \{I\}$$

Axioms for Signal and Continue (2)

Assume that the invariant can mention the number of processes in the queue to a condition variable.

- Let $\#cv$ be the number of proc's waiting in the queue to cv .
- The test $\text{empty}(cv)$ thus corresponds to $\#cv = 0$

$\text{wait}(cv)$ is *modelled* as an extension of the queue followed by processor release:

$$\text{wait}(cv) : \{?\} \#cv = \#cv + 1; \{I\} \text{ "sleep" } \{I\}$$

by **assignment** axiom:

$$\text{wait}(cv) : \{I \#cv \leftarrow \#cv + 1; \#cv := \#cv + 1; \{I\} \text{ "sleep" } \{I\}$$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{ ? \} \text{ if } (\#cv \neq 0) \#cv := \#cv - 1 \{ P \}$$
$$signal(cv) : \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow \#cv - 1}) \\ \text{if } (\#cv \neq 0) \#cv := \#cv - 1 \\ \{ P \} \}$$

- $signal_all(cv) : \{ P_{\#cv = 0} \} \#cv := 0 \{ P \}$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{ ? \} \text{ if } (\#cv \neq 0) \#cv := \#cv - 1 \{ P \}$$
$$signal(cv) : \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow \#cv - 1}) \\ \text{if } (\#cv \neq 0) \#cv := \#cv - 1 \\ \{ P \} \}$$

- $signal_all(cv) : \{ P_{\#cv = 0} \} \#cv := 0 \{ P \}$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{ ? \} \text{ if } (\#cv \neq 0) \#cv := \#cv - 1 \{ P \}$$
$$signal(cv) : \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow \#cv - 1}) \\ \text{if } (\#cv \neq 0) \#cv := \#cv - 1 \\ \{ P \} \}$$

- $signal_all(cv) : \{ P_{\#cv = 0} \} \#cv := 0 \{ P \}$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{ ? \} \text{ if } (\#cv \neq 0) \#cv := \#cv - 1 \{ P \}$$
$$signal(cv) : \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow \#cv - 1}) \}$$
$$\text{if } (\#cv \neq 0) \#cv := \#cv - 1$$
$$\{ P \}$$

- $signal_all(cv) : \{ P_{\#cv=0} \} \#cv := 0 \{ P \}$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{ ? \} \text{ if } (\#cv \neq 0) \#cv := \#cv - 1 \{ P \}$$
$$signal(cv) : \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow \#cv - 1}) \\ \text{if } (\#cv \neq 0) \#cv := \#cv - 1 \\ \{ P \} \}$$

- $signal_all(cv) : \{ P_{\#cv \leftarrow 0} \} \#cv := 0 \{ P \}$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{ ? \} \text{ if } (\#cv \neq 0) \#cv := \#cv - 1 \{ P \}$$
$$signal(cv) : \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow \#cv - 1}) \}$$
$$\text{if } (\#cv \neq 0) \#cv := \#cv - 1$$
$$\{ P \}$$

- $signal_all(cv) : \{ P_{\#cv \leftarrow 0} \} \#cv := 0 \{ P \}$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{ ? \} \text{ if } (\#cv \neq 0) \#cv := \#cv - 1 \{ P \}$$
$$signal(cv) : \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow \#cv - 1}) \}$$

if $(\#cv \neq 0) \#cv := \#cv - 1$
 $\{ P \}$

- $signal_all(cv) : \{ P_{\#cv \leftarrow 0} \} \#cv := 0 \{ P \}$

Axioms for Signal and Continue (4)

Together this gives:

Axioms for monitor communication

$$\{ I_{\#cv \leftarrow (\#cv+1)} \} \text{wait}(cv) \{ I \}$$

$$\{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow (\#cv-1)}) \} \text{signal}(cv) \{ P \}$$

$$\{ P_{\#cv \leftarrow 0} \} \text{signal_all}(cv) \{ P \}$$

If we know that $\#cv \neq 0$ whenever we signal, then the axiom for $\text{signal}(cv)$ be simplified to:

$$\{ P_{\#cv \leftarrow (\#cv-1)} \} \text{signal}(cv) \{ P \}$$

Note! $\#cv$ is not allowed in statements!, Only used for reasoning

Example: FIFO semaphore verification (1)

```
monitor Semaphore_fifo { # monitor invariant:  $s \geq 0$   
  int s := 0;           # value of the semaphore  
  cond pos;            # wait condition  
  
  procedure Psem() {  
    if (s=0)  
      wait (pos);  
    else  
      s := s - 1  
  }  
  
  procedure Vsem() {  
    if empty(pos)  
      s := s + 1  
    else  
      signal(pos);  
  }  
}
```

Consider the following monitor invariant:

$$s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

No process is waiting if the semaphore value is positive

Example: FIFO semaphore verification (1)

```
monitor Semaphore_fifo { # monitor invariant:  $s \geq 0$   
  int s := 0;           # value of the semaphore  
  cond pos;             # wait condition  
  
  procedure Psem() {  
    if (s=0)  
      wait (pos);  
    else  
      s := s - 1  
  }  
  
  procedure Vsem() {  
    if empty(pos)  
      s := s + 1  
    else  
      signal(pos);  
  }  
}
```

Consider the following monitor invariant:

$$s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

No process is waiting if the semaphore value is positive 

Example: FIFO semaphore verification: Psem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {  
  {I}  
  if (s=0) {I ∧ s = 0}  
    {I #pos ← (#pos+1)} wait(pos); {I}  
  else {I ∧ s ≠ 0}  
    {I s ← (s-1)} s := s-1; {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Psem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {  
  {I}  
  if (s=0) {I ∧ s = 0}  
    {I #pos ← (#pos+1)} wait(pos); {I}  
  else {I ∧ s ≠ 0}  
    {I s ← (s-1)} s := s-1; {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Psem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {  
  {I}  
  if (s=0) {I ∧ s = 0}  
    {I #pos ← (#pos+1)} wait(pos); {I}  
  else {I ∧ s ≠ 0}  
    {I s ← (s-1)} s := s-1; {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Psem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {  
  {I}  
  if (s=0) {I ∧ s = 0}  
    {I #pos ← (#pos+1)} wait(pos); {I}  
  else {I ∧ s ≠ 0}  
    {I s ← (s-1)} s := s-1; {I}  
  {I}  
}
```


Example: FIFO semaphore verification: Psem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {  
  {I}  
  if (s=0) {I ∧ s = 0}  
    {I #pos ← (#pos+1)} wait(pos); {I}  
  else {I ∧ s ≠ 0}  
    {I s ← (s-1)} s := s-1; {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Psem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {  
  {I}  
  if (s=0) {I ∧ s = 0}  
    {I#pos ← (#pos+1)} wait(pos); {I}  
  else {I ∧ s ≠ 0}  
    {I_s ← (s-1)} s := s-1; {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Psem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {  
  {I}  
  if (s=0) {I ∧ s = 0}  
    {I#pos ← (#pos+1)} wait(pos); {I}  
  else {I ∧ s ≠ 0}  
    {I_s ← (s-1)} s := s-1; {I}  
  {I}  
}
```

Example: FIFO semaphore verification (3)

$$I : s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

This gives two proof obligations:

If-branch:

$$\begin{aligned}(I \wedge s = 0) &\Rightarrow I_{\#pos \leftarrow (\#pos + 1)} \\ s = 0 &\Rightarrow s \geq 0 \wedge (s > 0 \Rightarrow \#pos + 1 = 0) \\ s = 0 &\Rightarrow s \geq 0\end{aligned}$$

Else branch:

$$\begin{aligned}(I \wedge s \neq 0) &\Rightarrow I_{s \leftarrow (s - 1)} \\ (s > 0 \wedge \#pos = 0) &\Rightarrow s - 1 \geq 0 \wedge (s - 1 \geq 0 \Rightarrow \#pos = 0) \\ (s > 0 \wedge \#pos = 0) &\Rightarrow s > 0 \wedge \#pos = 0\end{aligned}$$

Example: FIFO semaphore verification (3)

$$I : s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

This gives two proof obligations:

If-branch:

$$\begin{aligned}(I \wedge s = 0) &\Rightarrow I_{\#pos \leftarrow (\#pos + 1)} \\ s = 0 &\Rightarrow s \geq 0 \wedge (s > 0 \Rightarrow \#pos + 1 = 0) \\ s = 0 &\Rightarrow s \geq 0\end{aligned}$$

Else branch:

$$\begin{aligned}(I \wedge s \neq 0) &\Rightarrow I_{s \leftarrow (s - 1)} \\ (s > 0 \wedge \#pos = 0) &\Rightarrow s - 1 \geq 0 \wedge (s - 1 \geq 0 \Rightarrow \#pos = 0) \\ (s > 0 \wedge \#pos = 0) &\Rightarrow s > 0 \wedge \#pos = 0\end{aligned}$$

Example: FIFO semaphore verification: Vsem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Vsem() {  
  {I}  
  if empty(pos) {I ∧ #pos = 0}  
    {I_s ← (s+1)} s:=s+1; {I}  
  else {I ∧ #pos ≠ 0}  
    {I_#pos ← (#pos-1)} signal(pos); {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Vsem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Vsem() {  
  {I}  
  if empty(pos) {I ∧ #pos = 0}  
    {I_s ← (s+1)} s:=s+1; {I}  
  else {I ∧ #pos ≠ 0}  
    {I_#pos ← (#pos-1)} signal(pos); {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Vsem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Vsem() {  
  {I}  
  if empty(pos) {I ∧ #pos = 0}  
    {s ← (s+1)} s:=s+1; {I}  
  else {I ∧ #pos ≠ 0}  
    {I #pos ← (#pos-1)} signal(pos); {I}  
  {I}  
}
```


Example: FIFO semaphore verification: Vsem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Vsem() {  
  {I}  
  if empty(pos) {I ∧ #pos = 0}  
    {I_s ← (s+1)} s:=s+1; {I}  
  else {I ∧ #pos ≠ 0}  
    {I_#pos ← (#pos-1)} signal(pos); {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Vsem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Vsem() {  
  {I}  
  if empty(pos) {I ∧ #pos = 0}  
    {I_s ← (s+1)} s := s+1; {I}  
  else {I ∧ #pos ≠ 0}  
    {I_#pos ← (#pos-1)} signal(pos); {I}  
  {I}  
}
```

Example: FIFO semaphore verification: Vsem

$$I: s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Vsem() {  
  {I}  
  if empty(pos) {I ∧ #pos = 0}  
    {Is ← (s+1)} s := s+1; {I}  
  else {I ∧ #pos ≠ 0}  
    {I#pos ← (#pos-1)} signal(pos); {I}  
  {I}  
}
```

Example: FIFO semaphore verification (5)

$$I : s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

As above, this gives two proof obligations:

If-branch:

$$\begin{aligned}(I \wedge \#pos = 0) &\Rightarrow I_{s \leftarrow (s+1)} \\(s \geq 0 \wedge \#pos = 0) &\Rightarrow s + 1 \geq 0 \wedge (s + 1 > 0 \Rightarrow \#pos = 0) \\(s \geq 0 \wedge \#pos = 0) &\Rightarrow s + 1 \geq 0 \wedge \#pos = 0\end{aligned}$$

Else branch:

$$\begin{aligned}(I \wedge \#pos \neq 0) &\Rightarrow I_{\#pos \leftarrow (\#pos - 1)} \\(s = 0 \wedge \#pos \neq 0) &\Rightarrow s \geq 0 \wedge (s > 0 \Rightarrow \#pos - 1 = 0) \\s = 0 &\Rightarrow s \geq 0\end{aligned}$$

Example: FIFO semaphore verification (5)

$$I : s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

As above, this gives two proof obligations:

If-branch:

$$\begin{aligned}(I \wedge \#pos = 0) &\Rightarrow I_{s \leftarrow (s+1)} \\(s \geq 0 \wedge \#pos = 0) &\Rightarrow s + 1 \geq 0 \wedge (s + 1 > 0 \Rightarrow \#pos = 0) \\(s \geq 0 \wedge \#pos = 0) &\Rightarrow s + 1 \geq 0 \wedge \#pos = 0\end{aligned}$$

Else branch:

$$\begin{aligned}(I \wedge \#pos \neq 0) &\Rightarrow I_{\#pos \leftarrow (\#pos - 1)} \\(s = 0 \wedge \#pos \neq 0) &\Rightarrow s \geq 0 \wedge (s > 0 \Rightarrow \#pos - 1 = 0) \\s = 0 &\Rightarrow s \geq 0\end{aligned}$$

- [Andrews, 2000] Andrews, G. R. (2000).
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley.