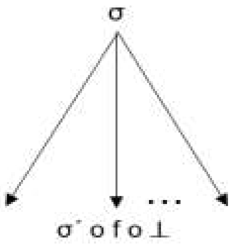
Clase 13

martes, 3 de junio de 2025 19:03

Los procesos pueden compartir variables y sincronizarse con un await.

Un programa puede tener varas computaciones y as[i a partir de un estado inicial producir varios finales.



Computaciones con 3 posibilidades

1) Finitas sin falla (estado final σ´)
2) Finitas con *deadlock* (estado de falla f)
3) Infinitas (estado indefinido ⊥)

Què implica ahora la completitud total? Cuando toda computaciòn de s termina en un estado final

Programa que calcula en la variable n el **factorial** de N:

```
                     c1 := true ; c2 := true ; i := 1 ; j := N ; n := N ;
[S1 :: while c1 do await true →                S2 :: while c2 do await true →
          if i + 1 < j then i := i + 1 ; n := n . i      ||      if j – 1 > i then j := j – 1 ; n := n . j
                   else c1 := false fi end                            else c2 := false fi end
          od                                             od]
```

Los *awaits* se utilizan para la **exclusión mutua** entre los dos procesos.

No es que usa c1 y c2 para la exclusiòn creo?

Correctitud parcial: si no le doy bola a las infinias o de deadlock, me importa que las que terminan lo hagan bien

Un problema en la concurrencia es que no es composicional:

- **Ya vimos antes la pérdida de la composicionalidad en la concurrencia:**
- Se cumple: $\{x = 0\}\ S_1 :: x := x + 2\ \{x = 2\}$ y $\{x = 0\}\ S_2 :: z := x\ \{z = 0\}$ **pero no se cumple:** $\{x = 0 \wedge x = 0\}\ [S_1 :: x := x + 2 \parallel S_2 :: z := x]\ \{x = 2 \wedge z = 0\}$

porque si en el programa $[S_1 \parallel S_2]$ se ejecuta S_2 después de S_1, al final se cumple z = 2. En efecto vale:

En lo secuencial si vale

Dos procesos que hacen lo mismo en apariencia no son intercambiables

Perdes la nociòn de caja negra

Mètodo de prueba en dos etapas:

Etapa 1: Agarro cada proceso y hago una prueba local

Etapa 2: chequeo global de consistencia de las pruebas de la primera etapa. Se revisa que todos los predcados sean verdaderos.

S_1		S_2		S_3
$\{p_1\}$		$\{p_2\}$		$\{p_3\}$
$[S_{11}$		S_{21}		S_{31}
$\{p_{12}\}$	$\parallel$	$\{p_{22}\}$	$\parallel$	$\{p_{32}\}$
S_{12}		S_{22}		S_{32}
.....		.....		.....
$\{p_{1k1}\}$		$\{p_{2k2}\}$		$\{p_{3k3}\}$
S_{1k1}		S_{2k2}		$S_{3k3}]$
$\{q_1\}$		$\{q_2\}$		$\{q_3\}$

pruebas locales $\longrightarrow$ prueba global

S_1 vs S_2

$\{p_1 \wedge p_2\}\ S_{21}\ \{p_1\}$
$\{p_1 \wedge p_{22}\}\ S_{22}\ \{p_1\}$
....
$\{q_1 \wedge p_{2k2}\}\ S_{2k2}\ \{q_1\}$

S_1 vs S_3

$\{p_1 \wedge p_3\}\ S_{31}\ \{p_1\}$
$\{p_1 \wedge p_{32}\}\ S_{32}\ \{p_1\}$
....
$\{q_1 \wedge p_{3k3}\}\ S_{3k3}\ \{q_1\}$

Y lo mismo con el resto de los pares

Los predicados deben ser **invariantes** (las *proof outlines deben ser* **libres de interferencia).**

Tengo que chequear que sean invariantes posta, que los demàs no caguen mi invariante

COmposiciòn paraela:

Regla de la composición paralela (PAR) para probar {p} [S_1 || … || S_n] {q}

- Definir *proof outlines* para cada proceso, las cuales deben cumplir:
- para todo par de *proof outlines* S_i^* y S_j^*
- para todo predicado r de S_i^*
- para toda instrucción T de S_j^* que sea una asignación o un await
- debe valer $\{r \wedge pre(T)\}$ T {r}, siendo pre(T) la precondición de T en S_j^*

Es decir, cada predicado debe ser *invariante* cualquiera sea el *interleaving*.

Las *proof outlines* de esta forma se denominan *libres de interferencia*.

Ejemplo de prueba de correctitud parcial usando la regla PAR

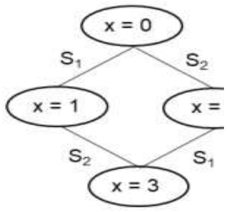
- Se quiere probar: $\{x = 0\}\ [S_1 :: x := x + 1 \ ||\ S_2 :: x := x + 2]\ \{x = 3\}$
- Las *proof outlines* naturales de los procesos S_1 y S_2, considerados aisladamer

$$\begin{array}{lcl} \{x = 0\} & & \{x = 0\} \\ S_1 :: x := x + 1 & || & S_2 :: x := x + 2 \\ \{x = 1\} & & \{x = 2\} \end{array}$$

pero **no son libres de interferencia** (por ejemplo, al final de S_1 también pue(

- Proponemos las siguientes *proof outlines*, justificadas por el diagrama que la: los estados inicial, intermedios y final del programa y cómo se obtienen:

$$\begin{array}{ll} \{x = 0 \vee x = 2\} & \{x = 0 \vee x = 1\} \\ [S_1 :: x := x + 1 \ ||\ S_2 :: x := x + 2] & \\ \{x = 1 \vee x = 3\} & \{x = 2 \vee x = 3\} \end{array}$$



- También son **libres de interferencia**:
  las siguientes fórmulas, requeridas por la regla PAR, son verdaderas:

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

Tengo que probar si se me caga o no si se me ejecuta algo en el medio
Notar que para obtener proof outlines libres de interferencia, hay que debilitar los predicados utilizados en las pruebas locales. Esta técnica no siempre logra una prueba exitosa

Hay un ejemplo de una que se caga en la pàgina 9

El estado intermedio es siempre x=1. Entonces no puedo ver de dònde viene ese x=1, si de s1 o s2. Es medio dèbil.
Se usan variables auxiliares para reforzar los predicados que no tienen que alterar el flujo bàsico.

Prueba de terminaciòn:
A proof outline es la base de todo.
Cada propiedad que quiero probar voy a hacer una que me sirve. Con solo probar que no hay infinitud estoy.
La prueba debe contemplar también las hipótesis de fairness existentes:
débil: “todo proceso siempre habilitado para ejecutarse se ejecuta alguna vez”
fuerte: “todo proceso infinitamernte habilitado para ejecutarse se ejecuta alguna vez”.
Fuerte implica debil pero no al revès: pregunta de examen el por que
Como pruebo que no haya inhanicion
Como pruebo que no haya interferencia
Segùn con què fairness evaluo cambia

Regla de deadlock:

TODOS tienen que terminar sin deadlock.
Ninguna puede fallar

Paso 1. Se especifican todas las tuplas de predicados que representan potenciales casos de *deadlock*:

$(p_{11}, p_{12}, \ldots, p_{1n})$
$(p_{21}, p_{22}, \ldots, p_{2n})$
....
$(p_{k1}, p_{k2}, \ldots, p_{kn})$

Los p_{ij} se refieren a awaits bloqueados o son postcondiciones

Paso 2. Se chequea en **cada tupla** $(p_{i1}, p_{i2}, \ldots, p_{in})$ que el predicado $p_{i1} \wedge p_{i2} \wedge \ldots \wedge p_{in}$ **sea falso**.

Por ejemplo, dado el programa:

$\{p_1\}$	$\{p_2\}$
[await B ⟶ S end ||	T]
$\{q_1\}$	$\{q_2\}$

la única tupla a considerar es **($p_1 \wedge \neg B$, q_2** y el paso 2 consiste en chequear que:
($p_1 \wedge \neg B \wedge q_2$) sea falso