

Tema 1

Sistemas Concurrentes y Distribuidos

Introducción

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 20 septiembre 2024

Nociones básicas y motivación

Conceptos básicos relacionados con la concurrencia

Notaciones de la Programación Concurrente

Exclusión mutua y sincronización

Propiedades de los sistemas concurrentes

Verificación de programas concurrentes

Introducción

Enfoque axiomático

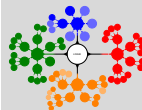
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

Concurrencia: programa y programación concurrente

Programa Concurrente

- ① Programa secuencial
 - Ejecución lineal de instrucciones.
- ② Programa Concurrente
 - Múltiples unidades de ejecución independientes (llamadas procesos).
 - Los procesos cooperan para realizar tareas esenciales.
- ③ Proceso
 - Definición: Entidad de software abstracta, dinámica y activa.
 - Estado: No sólo las instrucciones; incluye su estado actual y la capacidad de interactuar con el medio ambiente.
- ④ Estado del proceso
 - Valores en registros (Contador de Programa - PC, Puntero de Pila - SP, Memoria Heap).
 - Acceso a recursos como archivos y dispositivos.
 - El estado debe estar protegido de otros procesos concurrentes.
- ⑤ Gestión de la concurrencia
 - Esencial para evitar el acceso incontrolado entre procesos.





Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción
Enfoque axiomático

Conceptos básicos de procesos y concurrencia

1 Estructura de memoria del proceso

- Dividido en zonas:
 - Instrucciones: Secuencia a ejecutar.
 - Área de datos: Para variables globales/estáticas.
 - Pila: Para variables locales y parámetros de procedimientos.
 - Memoria Heap: Variables dinámicas no estáticas.

2 Programas Concurrentes vs Secuenciales

- Concurrencia: Los flujos de control se intercalan en menos núcleos.
- Preserva el paralelismo lógico independientemente del número de núcleos.
- Mejora la eficiencia al permitir la ejecución simultánea de múltiples hilos de control.
- Reduce los retrasos en los cálculos causados por las operaciones de E/S.

3 Concurrencia en Simulación

- Simula sistemas del mundo real de forma más natural.
- Las actividades del mundo real pueden modelarse mejor con procesos concurrentes independientes.

4 Definición de concurrencia

- El potencial de paralelismo en el código, independientemente de las limitaciones de hardware (número de núcleos/procesadores).



Conceptos clave de la programación concurrente

① Modelo Abstracto de Computación

- Expresa el paralelismo potencial a un alto nivel de abstracción.
- Independiente de la arquitectura hardware del ordenador.

② Concurrencia en la programación

- Su objetivo es simplificar la sincronización y la comunicación entre los procesos.
- Permite que el código se ejecute en diferentes arquitecturas.
- Fomenta programas portables e independientes de la arquitectura.

③ Beneficios del modelo abstracto

- Herramientas para diseñar y razonar sobre la concurrencia.
- Simplifica el lenguaje de programación con primitivas de sincronización/comunicación de alto nivel.
- Evita llamadas al sistema de bajo nivel o instrucciones específicas de la máquina.

④ Cinco axiomas de la programación concurrente

- (i)Atomicidad e Intercalación de Instrucciones
- (ii)Consistencia de datos después del acceso concurrente
- (iii)Irrepetibilidad de las secuencias de instrucciones
- (iv)Independencia de la velocidad de proceso
- (v)Hipótesis de progreso finito



(i)Atomicidad e intercalación de instrucciones

Concepto de **Sentencia atómica**(indivisible)

- **Atomicidad:** Cada instrucción de máquina o ensamblador se ejecuta hasta su finalización sin interrupción, lo que garantiza la coherencia en la ejecución concurrente.
- **Concurrencia:** Los programas escritos para múltiples procesos (paralelos o de tiempo compartido) producen conjuntos de instrucciones atómicas intercaladas.
- **Entrelazamiento:** Las secuencias intercaladas de instrucciones de dos procesos (por ejemplo, I1x y I2x) definen el comportamiento observable del programa.
- **No determinismo:** La secuencia de instrucciones intercaladas es impredecible, lo que pone de relieve el no determinismo en la programación concurrente.
- **Independencia del hardware:** Este modelo de concurrencia se aplica a nivel lógico, independientemente del hardware que ejecute el programa.





Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático

Como ejemplo de **instrucciones atómicas** podemos citar muchas de las instrucciones máquina del repertorio de un procesador, por ejemplo las tres siguientes:

- Leer una celda de memoria (variable de tipo simple **x**) y cargar su valor en un registro (**r**) del procesador : `LOAD x`
- Incrementar el valor de un registro (**r**) u otras operaciones aritméticas sobre registros del procesador: `ADD r, a`
- Escribir el valor de un registro (**r**) en una celda de memoria: `STORE x`

Sentencias no atómicas

La mayoría de las sentencias de los lenguajes de programación de alto nivel son típicamente no atómicas, por ejemplo, la sentencia:

`x := x + 1;` (incrementa el valor de la variable entera en 1 unidad)

El compilador utiliza una secuencia de tres sentencias:

- ❶ Leer el valor de **x** y cargarlo en un registro **r** del procesador
- ❷ Incrementar en una unidad el valor almacenado en el registro **r**
- ❸ Escribir el valor del registro **r** en la variable **x**

El valor que toma la variable **x** justo al terminar la ejecución de la sentencia dependerá de que haya o no otras sentencias ejecutándose a la vez y tratando de escribir simultáneamente en la variable **x**

Decimos en este caso que existe *indeterminación*: no se puede predecir el estado final del proceso a partir de su estado inicial



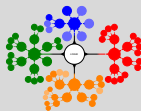
(i) Entrelazamiento de instrucciones atómicas

Programa concurrente C compuesto por 2 procesos secuenciales: P_A y P_B que se ejecutan a la vez

La ejecución de C puede dar lugar a cualquiera de los posibles entrelazamientos de sentencias atómicas de P_A y P_B :

Pr.	Posibles secuencias de instrucciones atómicas
P_A	$A_1 A_2 A_3 A_4 A_5$
P_B	$B_1 B_2 B_3 B_4 B_5$
C	$A_1 A_2 A_3 A_4 A_5 B_1 B_2 B_3 B_4 B_5$
C	$B_1 B_2 B_3 B_4 B_5 A_1 A_2 A_3 A_4 A_5$
C	$A_1 B_1 A_2 B_2 \dots$
C	$B_1 B_2 A_1 B_3 B_4 A_2 \dots$
C	\dots

Las secuencia se va ordenando a medida que acaba cada una de las sentencias atómicas que la componen (que es cuando tienen efecto)



Modelo de programación basado en el entrelazamiento de instrucciones atómicas

- Este modelo es una abstracción que estudia las secuencias de ejecución de procesos concurrentes.
- Solo se consideran las características relevantes para el resultado del programa.
- Detalles ignorados:
 - Estado de memoria asignado
 - Registros de cada proceso
 - Costos de cambios de contexto
 - Políticas de planificación
 - Diferencias de velocidad en multiprocesadores y monoprocesadores



(ii) Consistencia de los datos tras el acceso simultáneo

- **Instrucciones atómicas:** Cuando dos instrucciones atómicas acceden a la misma dirección de memoria, el resultado debe ser consistente, independientemente de si se ejecutan en paralelismo real o intercambiándolas en un orden impredecible.
- **Consistencia de datos:** Después de que ambos procesos terminen, la memoria debe permanecer en un estado válido según el tipo de datos de la variable, asegurando que no haya corrupción de datos.
- **Resultados impredecibles:** El valor final de una variable compartida no es predecible pero permanece consistente con su rango definido.
- **Soporte de Hardware:** La consistencia de la memoria está garantizada por el árbitro del bus de memoria del sistema, evitando la corrupción de datos por accesos simultáneos.



(iii) Irrepetibilidad de las secuencias de instrucciones

- **Trazas:** Los programas concurrentes generan un gran número de posibles secuencias de entrelazamiento de instrucciones, por lo que es poco probable que dos ejecuciones sigan el mismo camino.
- **Errores transitorios:** Estos errores aparecen en algunas trazas pero no en otras, lo que dificulta su detección y corrección.
- **Reto de depuración:** La imprevisibilidad de las secuencias de ejecución complica la depuración y el análisis de corrección en programas concurrentes.
- **Solución:** Los métodos formales basados en la lógica matemática son necesarios para verificar la corrección de los programas y eliminar los errores transitorios en el software concurrente.

Historia o traza de un programa concurrente: Secuencia de estados $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$, producida por una secuencia concreta de entrelazamiento



(iv) Independencia de la velocidad del proceso

- **Independencia de la velocidad de ejecución:** La corrección en programas concurrentes no debe depender de la velocidad relativa de los procesos. A evitar:
 - Falta de Portabilidad: Los programas pueden fallar en diferentes plataformas si se hacen suposiciones sobre la velocidad de ejecución.
 - Condiciones de carrera: El acceso a variables compartidas puede producir resultados impredecibles e incorrectos basados en el orden y la velocidad de ejecución.
- **Ejemplo de condición de carrera:** Dos procesos (P1 y P2) modificando la misma variable pueden producir resultados impredecibles dependiendo del orden de ejecución del proceso.
- **Excepción de los sistemas en tiempo real:** En las aplicaciones de tiempo real, la velocidad y el orden de ejecución de los procesos son críticos, y se asignan niveles de prioridad a los procesos.
- **Suposición de tiempo finito:** Todos los procesos deben completarse en un tiempo finito para garantizar que se mantienen las propiedades de corrección, como la propiedad de *vivacidad* de los procesos.



(v) Hipótesis del progreso finito

- **Progreso global:** Si al menos un proceso está listo para ejecutarse, eventualmente se le debe permitir ejecutarse, asegurando que el programa no entre en punto muerto o interbloqueo (*deadlock*).
- **Progreso local:** Una vez que un proceso comienza a ejecutar una sección de código, debe terminar dicha sección.
- **Hipótesis de progreso finito:** Ningún proceso debe detenerse indefinidamente debido a condiciones internas (como valores en contadores o registros) durante la ejecución del programa. Todo proceso debe seguir progresando durante la ejecución del programa.



Notaciones para expresar ejecución concurrente



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción
Enfoque axiomático

Propuestas iniciales: no separan la definición de los procesos de su sincronización

Propuestas posteriores: separan ambos conceptos e imponen una estructura al programa concurrente, diferente de uno secuencial

Declaración de procesos: rutinas específicas de programación concurrente \Rightarrow Estructura del programa concurrente más clara

Sistemas Estáticos:

- Número de procesos fijado en el fuente del programa
- Los procesos se activan al lanzar el programa
- Ejemplo: Message Passing Interface (MPI-1)

Sistemas Dinámicos:

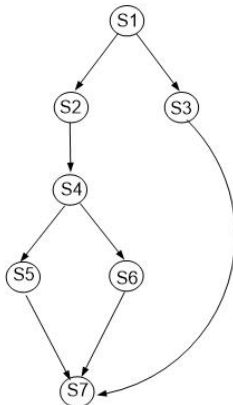
- Número variable de procesos/hebras que se pueden activar en cualquier momento de la ejecución
- Ejemplos: OpenMP, PThreads, Java Threads, MPI-2

Grafo de sincronización

El Grafo de Sincronización es un Grafo Dirigido Acíclico (DAG) donde cada nodo representa una secuencia de sentencias del programa (actividad):

Dadas dos actividades, S_1 y S_2 , una arista (flecha) desde S_1 hacia S_2 significa que S_2 no puede comenzar su ejecución hasta que S_1 haya finalizado

Muestra las restricciones de precedencia que determinan cuándo una actividad puede empezar en un programa



Definición estática de procesos

```
var ... \\variables compartidas

process Uno;
var ... \\variables locales
begin
    ... \\codigo
end;
process Dos;
var ... \\variables locales
begin
    ... \\codigo
end;
... \\otros procesos
```

Se *lanza* la ejecución concurrente de ambos procesos en 1 bloque del programa:

```
cobegin  Uno  ||  Dos  coend;
```

El programa acaba cuando acaban de ejecutarse todas las instrucciones de 2 los procesos. Las variables compartidas se inicializan antes de comenzar la ejecución concurrente de los procesos.



Definición estática de vectores de procesos



```
var ... \\variables compartidas

process NomP[ind : a.. b];
var ... \\variables locales
begin
    ... \\codigo
    ...\\ (ind vale a, a+1, ... b)
end;
... \\otros procesos
```

Nociones básicas y motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la Programación Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

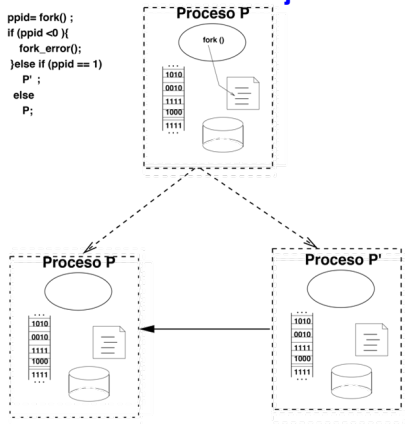
Verificación de
programas
concurrentes

Introducción
Enfoque axiomático

Creación de procesos no estructurada: fork-join

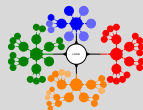
```
process P2;  
begin  
  D;  
end;  
process P1;  
begin  
  A;  
  fork P2;  
  B;  
  join P2;  
  C;  
end;
```

```
ppid= fork() ;  
if (ppid <0 ){  
  fork_error();  
}else if (ppid == 1)  
  P' ;  
else  
  P;
```



fork: sentencia que especifica que la rutina nombrada puede comenzar su ejecución, al mismo tiempo que comienza la sentencia siguiente (bifurcación)

join: sentencia que espera la terminación de la rutina nombrada, antes de comenzar la sentencia siguiente (unión)



Introducción a los conceptos de *exclusión mutua* y *sincronización*

No todas las secuencias de entrelazamiento de las instrucciones de los procesos a que da lugar la ejecución de un programa concurrente son aceptables

- Los procesos no suelen ejecutarse de una forma totalmente independiente, sino que que **colaboran** entre ellos
- **Condición de sincronización:** restricción en el orden en que se pueden entremezclar las instrucciones que generan los procesos de un programa
- **Exclusión mutua:** se da en secuencias finitas de intrucciones del código de un programa, que han de ejecutarse de principio a fin por un único proceso, que no puede ser desplazado del procesador mientras ejecuta esta **sección crítica** de instrucciones





En un programa concurrente, una condición de sincronización establece para asegurarnos de que todas las trazas del programa son correctas

- Suele ocurrir cuando, en un punto concreto de su ejecución, uno o varios procesos deben esperar a que se cumpla una determinada condición global (depende de varios procesos)

Un ejemplo sencillo de **condición de sincronización**: en el ejemplo del productor-consumidor con 1 variable compartida, el productor espera a que el consumidor haya leído el valor escrito en la variable antes de volver a escribirla.

Nociones básicas y motivación

Conceptos básicos relacionados con la concurrencia

Notaciones de la Programación Concurrente

Exclusión mutua y sincronización

Propiedades de los sistemas concurrentes

Verificación de programas concurrentes

Introducción

Enfoque axiomático

El paradigma del *Productor Consumidor*

Dos procesos cooperantes en los cuales uno de ellos (*productor*) genera una secuencia de valores (p.ej.enteros) y el otro (*consumidor*) utiliza cada uno de estos valores

```
var x : integer ; { contiene cada valor producido }
{ Proceso productor: calcula 'x' }
process Productor ;
var a : integer ; { no compartida }
begin
  while true do begin
    { calcular un valor }
    a := ProducirValor() ;
    { escribir en mem. compartida }
    x := a ; { sentencia E }
  end
end
process Consumidor ; { Proceso Consumidor: lee 'x' }
var b : integer ; { no compartida }
begin
  while true do begin
    { leer de mem. compartida }
    b := x ; { sentencia L }
    { utilizar el valor leído }
    UsarValor(b) ;
  end
end
```



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción
Enfoque axiomático

Trazas incorrectas de un programa concurrente

Los procesos sólo *funcionan* como se espera si el orden en el que se entrelazan las sentencias elementales etiquetadas como E (escritura) y L (lectura) es: E, L, E, L, E, L, . . .

Trazas incorrectas del programa *Productor Consumidor*:

- **L, E, L, E, . . .** es incorrecta: se hace una lectura de **x** previa a cualquier escritura (se lee valor indeterminado)
- **E, L, E, E, L, . . .** es incorrecta: hay dos escrituras sin ninguna lectura entre ellas (se produce un valor que no se lee)
- **E, L, L, E, L, . . .** es incorrecta (para el código del P/C anterior): hay dos lecturas de un mismo valor que, por tanto, es usado dos veces

La secuencia válida asegura la condición de sincronización:

- *Consumidor* no lee hasta que *Productor* escribe un nuevo valor en **x** (cada valor producido es usado una sola vez)
- *Productor* no escribe un nuevo valor hasta que *Consumidor* lea el último valor almacenado en **x** (ningún valor producido se pierde)



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
ConcurrenteExclusión mutua y
sincronizaciónPropiedades de los
sistemas concurrentesVerificación de
programas
concurrentes

Introducción
Enfoque axiomático

Instrucciones compuestas e instrucciones atómicas

En pseudo-código, podemos convertir las sentencias compuestas a atómicas: indicando que se deben de ejecutar sin ser interrumpidas, usando los caracteres `<` y `>`:

```
{ instr. compuestas (no atómicas) }
begin
x := 0 ;
cobegin
x:= x+1 ;
x:= x-1 ;
coend
end
////////////////////////////////
{ instr. atómicas }
begin
x := 0 ;
cobegin
< x:= x+1 > ;
< x:= x-1 > ;
coend
end
```

- En el primer código, al acabar, **x** puede tener un valor cualquiera del conjunto $\{-1, 0, 1\}$
- En el segundo código **x** finaliza siempre con el valor 0

Concepto de corrección de un programa concurrente

Propiedad de un programa concurrente: Algo que se puede afirmar del programa que es cierto para todas las posibles trazas del programa

Hay 2 tipos de propiedades:

- Propiedad de seguridad (safety).
- Propiedad de vivacidad (liveness).



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción
Enfoque axiomático

Condiciones que deben cumplirse en cada instante de la traza del programa; son del tipo: *nunca pasará nada malo*

- Requeridas en especificaciones estáticas del programa
- Son fáciles de demostrar y para cumplirlas se suelen impedir determinadas posibles trazas

Ejemplos:

- *Exclusión mutua*: 2 procesos nunca entrelazan ciertas subsecuencias de operaciones
- *Ausencia Interbloqueo* (Deadlock-freedom): Nunca ocurrirá que los procesos se encuentren esperando algo que nunca sucederá
- *Propiedad de seguridad en el Productor-Consumidor*: El consumidor debe consumir todos los datos producidos por el productor en el orden en que se van produciendo



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
conurrencia

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción
Enfoque axiomático

Son propiedades que deben cumplirse en algún momento futuro y no especificado del programa; son del tipo: *realmente sucede (o va a suceder) algo bueno*

- Son propiedades dinámicas, más difíciles de demostrar que las propiedades de seguridad

Ejemplos:

- *Ausencia de inanición* (starvation-freedom): Un proceso o grupo de procesos del programa no puede ser indefinidamente pospuesto. Asegura que, en algún momento, podrá avanzar
- *Equidad* (fairness): Tipo particular de propiedad de vivacidad. Un proceso que pueda progresar debe hacerlo con justicia relativa con respecto a los demás procesos del programa. Más ligado a la implementación y a veces se incumple: existen distintos grados



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrencia

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción
Enfoque axiomático

Pruebas simples de programas concurrentes

¿Cómo demostrar que un programa cumple una determinada propiedad ?

- *Posibilidad*: realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad
- *Problema*: Sólo permite considerar un número finito y muy limitado de trazas del programa y no demuestra la ausencia de casos indeseables (en alguna traza no explorada)
- Ejemplo: Comprobar que el proceso P produce al final $x == 3$:

```
process P ;  
var x : integer := 0 ;  
cobegin  
x = x+1 ; x = x+2 ;  
coend
```

(hay varias trazas que llevan al resultado: $x==1$ o $x==2$, y estas historias podrían ocurrir en algunas ejecuciones, por tanto, produciendo un resultado no-determinado)





- *Enfoque operacional*: Análisis exhaustivo de casos. Se comprueba la corrección de todas las posibles trazas
- *Problema*: Su utilidad está muy limitada cuando se aplica a los programas concurrentes, ya que el número de entrelazamientos crece exponencialmente con el número de instrucciones de los procesos

Para el sencillo programa P, formado por 2 procesos y 3 sentencias atómicas por proceso, tendríamos que estudiar 20 historias diferentes (*=permutaciones con repetición de 6 elementos con 2 grupos de 3 elementos*)

Nociones básicas y motivación

Conceptos básicos relacionados con la concurrencia

Notaciones de la Programación Concurrente

Exclusión mutua y sincronización

Propiedades de los sistemas concurrentes

Verificación de programas concurrentes

Introducción

Enfoque axiomático

Verificación: enfoque axiomático

Se define un sistema lógico formal que permite establecer propiedades de programas en base a axiomas y reglas de inferencia

- Se usan fórmulas lógicas (llamados: *asertos* o *predicados*) para caracterizar un conjunto de estados
- Las sentencias atómicas actúan como transformadores de tales asertos. Los teoremas se escriben de la forma:

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

Interpretación de un teorema (llamado también: *triple*) en esta lógica: “*Si la ejecución de la sentencia S comienza en algún estado en el que es verdadero el aserto P (precondición), entonces el aserto Q (poscondición) será verdadero en el estado resultante*”

Menor Complejidad que el enfoque operacional: El trabajo que conlleva la prueba de corrección es proporcional al número de sentencias atómicas en el programa





Notación: *Triples* de Hoare

- Se introduce la notación de *especificación de corrección parcial* para especificar lo que hace un programa

$$\{P\} C \{Q\}$$

- C es un programa del lenguaje cuyos programas están siendo especificados
- P y Q son **asertos** definidos con las variables programa que representa C
- Las variables libres de P y Q son del programa o son variables lógicas
- Los asertos caracterizan los estados aceptables del programa:
 - V caracteriza a todos los estados del programa; sin embargo, F no caracteriza a ninguno
 - Los *triples* ($\{P\} C \{Q\}$) son los teoremas (= *proposiciones demostrables*) con la lógica que estamos tratando

Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrencia

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático

$\{P\} C \{Q\}$ es cierto si

- siempre que C es ejecutado en un estado que
- satisface P y *si* la ejecución de C termina,
- el estado en que C termina satisface Q

Ejemplo: $\{X == 1\} X = X + 1 \{X == 2\}$

- el aserto P dice que el valor de X es 1
- el aserto Q dice que el valor de X es 2
- C es la sentencia de asignación $X = X + 1$
- $\{X == 1\} X = X + 1 \{X == 2\}$ es cierto
- $\{X == 1\} X = X + 1 \{X == 3\}$ es falso
- $\{X == 1\} \text{WHILE } T \text{ DO NULL } \{Y == 3\}$ ¡es cierto!

La estructura de las demostraciones

Una demostración es una secuencia de líneas (o *triples*) de la forma $\{P\} C \{Q\}$, cada una de los cuales es un *axioma* o deriva de los anteriores aplicando una *regla de inferencia* de la lógica

- Una demostración consiste en una secuencia de *líneas*:
 - como la definición de $()^2$ del ejemplo siguiente
- Cada una de las líneas es una instancia de un *axioma*
- o se deriva de las líneas anteriores por medio de una *regla de inferencia*
- La sentencia de la última línea de una demostración es lo que se quiere demostrar
 - como ocurre con: $(x + 1)^2 == x^2 + 2 \times x + 1$



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático

- Ejemplo de una demostración formal:

$$1. (X + 1)^2 == (X + 1) \times (X + 1)$$

Definición de $()^2$

$$2. (X + 1) \times (X + 1) == (X + 1) \times X + (X + 1) \times 1$$

Distributiva izquierda de \times con $+$

$$3. (X + 1)^2 == (X + 1) \times X + (X + 1) \times 1$$

Sustituir línea 2 en 1

$$4. (X + 1) \times 1 == X + 1 \quad \text{Ley identidad para 1}$$

$$5. (X + 1) \times X == X \times X + 1 \times X$$

Distributiva derecha de \times con $+$

$$6. (X + 1)^2 == X \times X + 1 \times X + X + 1$$

Sustituir líneas 4 y 5 en 3

$$7. 1 \times X == X \quad \text{Ley identidad para 1}$$

$$8. (X + 1)^2 == X \times X + X + X + 1$$

Sustituir línea 7 en 6

$$9. X \times X == X^2 \quad \text{Definición de } ()^2$$

$$10. X + X == 2 \times X \quad 2 == 1 + 1, \text{ ley distributiva}$$

$$11. (X + 1)^2 == X^2 + 2 \times X + 1$$

Sustituir líneas 9 y 10 en 8

Propiedades de seguridad y complección

- Un aserto $\{P\}$ caracteriza un estado **aceptable** del programa, es decir, un estado que podría ser alcanzado por el programa si sus variables tomaran unos determinados valores

- Concepto de **Interpretación** de las fórmulas de un SLF:

$$\text{asertos} \rightarrow \{V, F\}$$

- Fórmula satisfascible
- Fórmula válida
- el aserto $\{V\}$ caracteriza a todos los estados del programa (o *precondición más débil*)
- el aserto $\{F\}$ no se cumple por parte de ningún estado del programa (o *precondición más fuerte*)
- **SLF seguro**: $\text{asertos} \subseteq \{\text{hechos ciertos}\}$ – expresados como fórmulas de la lógica
- **SLF completo**: $\{\text{hechos ciertos}\} \subseteq \{\text{asertos}\}$
- Los sistemas lógicos para demostración de programas no suelen poseer la propiedad de **complección**, pero a efectos prácticos es suficiente con la de **complección relativa** (todas las proposiciones son demostrables excepto si contienen expresiones aritméticas)





Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático

- Leyes distributivas:
 - $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
 - $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
- Leyes de De Morgan:
 - $\neg(P \wedge Q) = \neg P \vee \neg Q$
 - $\neg(P \vee Q) = \neg P \wedge \neg Q$
- Eliminación-*And*: $(P \wedge Q) \rightarrow P$
- Eliminación-*Or*: $P \rightarrow (P \vee Q)$

Axiomas y reglas de inferencia

Sirven para poder llevar a cabo las demostraciones de programas, ya que cada línea de la demostración es o bien un axioma o se deriva de la anterior mediante una regla de inferencia

- **Axioma:** fórmulas que sabemos que son ciertas en cualquier estado del programa
- **Regla de inferencia:** para derivar fórmulas ciertas a partir de los axiomas o de otras que se han demostrado ciertos previamente

Notación de una **Regla de inferencia:**

(*nombre de la regla*) $\frac{H_1, H_2, \dots, H_n}{C}$

Un **aserto** (o *teorema*) es una fórmula con una interpretación cierta que pertenece al dominio de los hechos que queremos probar (o *dominio del discurso*)

Los **asertos** de nuestra lógica coinciden con las líneas o sentencias lógicas de las que se compone la demostración de un programa





Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático

- ① **Axioma de la sentencia nula:** $\{P\} \text{ NULL } \{P\}$: si el aserto es cierto antes de ejecutarse esta sentencia, tendrá la misma interpretación cuando la sentencia termine
- ② **Sustitución textual:** $\{P_e^x\}$ es el resultado de sustituir la expresión e en cualquier aparición de la variable x en P
- ③ **Axioma de asignación:** $\{P_e^x\}_x = e\{P\}$: una asignación cambia solo el valor de la *variable objetivo*, el resto de las variables conservan los mismos valores.

Ejemplos:

- A $\{V\} x = 5 \{x == 5\} \equiv \{P\}$ es un aserto pues:
 $P_5^x \equiv \{x == 5\}_5^x \equiv \{5 == 5\} \equiv V$
- B $\{x > 0\} x = x + 1 \{x > 1\} \equiv \{P\}$ es un aserto pues:
 $P_{x+1}^x \equiv \{x > 1\}_{x+1}^x \equiv \{x + 1 > 1\} \equiv \{x > 0\}$

Reglas para *conectar* los triples en las demostraciones:

- ④ **Regla de la consecuencia (1):**
$$\frac{\{P\}S\{Q\}, \{Q\} \rightarrow \{R\}}{\{P\}S\{R\}}$$



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático

① Regla de la consecuencia (2): $\frac{\{R\} \rightarrow \{P\}, \{P\} S \{Q\}}{\{R\} S \{Q\}}$

② Regla de la composición: $\frac{\{P\} S_1 \{Q\}, \{Q\} S_2 \{R\}}{\{P\} S_1; S_2 \{R\}}$: para obtener la pre y pos-condición de 2 sentencias juntas

③ Regla del IF: $\frac{\{P\} \wedge \{B\} S_1 \{Q\}, \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{Q\}}$

④ Regla de la iteración: $\frac{\{I \wedge B\} S \{I\}}{\{I\} \text{while } B \text{ do } S \text{ enddo } \{I \wedge \neg B\}}$: podrá iterar un número arbitrario de veces (incluso 0); el *invariante* I se satisface antes y después de cada iteración

Verificación de sentencias concurrentes

- La ejecución concurrente de los procesos hace que el SLF deje de ser seguro: *problema de la interferencia*

```
y=0; z=0;
cobegin
  x= y+z || y=1; z=2
coend;
```

estados del programa	valores de x
$x==y==z==0$	0
$x==y==1, z==0$	1
$x==3, y==1, z==2$	3
-	2

¿Cómo evitar la interferencia?

- Evaluación de expresiones que no hacen referencia a variables modificadas concurrentemente

```
{x==0; y==0}
x= 0; y=0;
cobegin x= x+1 || y= y+1 coend;
z=x+y
{x==1, y==1, z==2}
```

- ¡Condición demasiado restrictiva!: casi ningún programa la cumpliría.



Verificación de sentencias concurrentes-II

Acción atómica elemental $\langle \dots \rangle$

Propiedad **como máximo una vez**: “La evaluación de una expresión por un proceso concurrente se hace de forma atómica si las variables que comparte son leídas o escritas por un único proceso”

Ejemplo 1: *Evaluación de expresiones*

- ① $x = 0; y = 0; \text{cobegin } x = y + 1 || y = x + 1 \text{ coend};$
¡Incorrecta!
- ② $x = 0; y = 0; z = Z; \text{cobegin } x = y + 1 || y = z + 1 \text{ coend};$
- ③ $x = 0; y = 0; z = Z; \text{cobegin } x = z + 1 || y = x + 1 \text{ coend};$

No interfiere. entre un aserto y una instrucción atómica a:

$$\{C \wedge \text{pre}(a)\} a \{C\}$$

Ejemplo de no-interferencia:

- ① $\{y = 0, z = Z\} \langle y = z + 1 \rangle$ no interfiere con la poscondición de $\langle x = y + 1 \rangle \{x = Y + 1, z = Z\}$
- ② $\{x = 0, z = Z\} \langle x = y + 1 \rangle$ no interfiere con la poscondición de $\langle y = z + 1 \rangle \{y = Z + 1, z = Z\}$



Regla de inferencia de la composición concurrente de sentencias

Si $\{P_i\} S_i \{Q_i\}$ no interfieren, entonces se puede demostrar el triple:

$$\begin{array}{c} \{P_1 \wedge P_2 \dots \wedge P_n\} \\ \text{COBEGIN} \\ S_1 \parallel S_2 \parallel \dots \parallel S_n \\ \text{COEND} \\ \{Q_1 \wedge Q_2 \dots \wedge Q_n\} \end{array}$$

Si los asertos de los procesos no se invalidan entre sí, entonces su composición concurrente transforma la conjunción de sus precondiciones en la conjunción de sus postcondiciones:

$$\begin{array}{c} \{x == 0\} \\ \text{COBEGIN} \\ \langle x = x + 1; \rangle \parallel \langle x = x + 2 \rangle \\ \text{COEND} \\ \{x == 3\} \end{array}$$



Regla de inferencia de la composición concurrente de sentencias -II

Traza de la demostración libre de interferencias:

$$\begin{array}{c} \{x == 0\} \\ \text{COBEGIN} \\ \{x == 0 \vee x == 2\} \quad \{x == 0 \vee x == 1\} \\ \quad < x = x + 1; > \parallel < x = x + 2 > \\ \{x == 1 \vee x == 3\} \quad \{x == 2 \vee x == 3\} \\ \text{COEND} \\ \{x == 3\} \end{array}$$

Aplicando la *regla de la no interferencia* a la expresión de asertos concurrentes anterior, la precondition de la sentencia `cobegin` ha de ser equivalente a la conjunción de las preconditiones de sus sentencias componentes $(x == 0 \vee x == 2) \wedge (x == 0 \vee x == 1) \equiv (x == 0)$ Así mismo ocurre con la poscondición de `coend` $(x == 1 \vee x == 3) \wedge (x == 2 \vee x == 3) \equiv (x == 3)$



Invariante global: se trata de un predicado que referencia variables globales del programa que se demuestra cierto en el estado inicial de cada proceso y se mantiene cierto ante cualquier *asignación* dentro de los procesos

Ejemplo de invariante global:

- En una solución correcta del Productor-Consumidor, un invariante global sería:

$$\text{consumidos} \leq \text{producidos} \leq \text{consumidos} + 1$$

- Condición de invariante global (IG):** Dado un aserto I definido a partir de las *variables compartidas* entre los procesos de un programa concurrente, puede ser considerado un IG válido si y sólo si:
 - Es cierto para los valores iniciales de las variables
 - Se mantiene cierto después de la ejecución de cada *instrucción atómica* a del programa: $\{I \wedge \text{pre}(a)\} a \{I\}$



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrencia

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático

Para más información, ejercicios, bibliografía adicional, se puede consultar:

1.1. [Conceptos básicos y Motivación](#), Palma (2003): capítulo 1

1.2. [Modelo abstracto y Consideraciones sobre el hardware](#), Ben-Ari (2006), capítulo 2. Andrews (2000), Palma (2003): capítulo 1

1.3. [Exclusión mutua y sincronización](#), Palma (2003), capítulo 1.

1.4. [Propiedades de los Sistemas Concurrentes](#), Palma (2003), Capel (2022): capítulo 1

1.5. [Verificación de Programas concurrentes](#), Andrews (2000), capítulo 2. Capel (2022): capítulo 1



Nociones básicas y
motivación

Conceptos básicos
relacionados con la
concurrency

Notaciones de la
Programación
Concurrente

Exclusión mutua y
sincronización

Propiedades de los
sistemas concurrentes

Verificación de
programas
concurrentes

Introducción

Enfoque axiomático