# Lógica de Programas de Floyd-Hoare

## 1 Introducción a la Verificación de Programas

La verificación de programas (secuenciales), según un enfoque clásico del tema, se basa en la utilización de un *Sistema Lógico Formal* (SLF), para llevar primeramente a cabo una demostración de corrección parcial; es decir, demostrar que si el programa termina, entonces se ha de llegar necesariamente a un estado final deseado. La corrección total se demostraría posteriormente, probando la terminación del programa en todos los casos de interés.

Un *SLF* facilita la elaboración de proposiciones ciertas, con una base lógica precisa, acerca de estados concretos que un programa alcanza durante su ejecución. Un estado viene definido por los valores que tienen las variables del programa en él. Un *SLF* se define formalmente como sigue:

$$\text{SLF} = \{\text{Símbolos, Fórmulas, Axiomas, Reglas de Inferencia}\}$$

- Símbolos: {sentencias del lenguaje, variables proposicionales, operadores, etc.}
- Fórmulas: secuencias de símbolos *bien formadas.*
- Reglas de Inferencia: indican cómo derivar fórmulas ciertas a partir de axiomas (fórmulas que sabe son ciertas) y de otras fórmulas que se han demostrado ciertas.

Las reglas de inferencia poseen el siguiente significado: si todas sus hipótesis son ciertas, entonces su conclusión también lo es:

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

Tanto las hipótesis como la conclusión han de ser fórmulas o una representación esquemática de ellas. Los teoremas del *SLF* que vamos definir coinciden con las líneas o *sentencias lógicas* en las que se estructura la demostración de un programa. Los teoremas se pueden obtener a partir de los axiomas, o bien, mediante la aplicación de las reglas de inferencia a los axiomas y a otros teoremas. Una demostración de corrección de un programa es una secuencia de teoremas, tal que cada uno de ellos puede ser derivado de los anteriores mediante la aplicación de una regla de inferencia.

## 1.1 Propiedades de seguridad y complección

Las fórmulas de un *SLF* representan afirmaciones construidas en un determinado dominio del discurso. Un teorema es una fórmula que representa a una afirmación[1] cierta que pertenece al dominio del discurso. Para saber de la certeza de los asertos, es necesario el proporcionar una interpretación a las fórmulas, que se define como la siguiente correspondencia:

$$Interpretación \rightarrow \{V, F\}$$

Se dice que un *SLF* es seguro respecto de una interpretación, si todos sus axiomas y reglas de inferencia lo son:

- *Axioma Seguro*: la interpretación le hace corresponder la constante lógica *V*.
- *Regla de Inferencia Segura*: si a todas las hipótesis y a la conclusión les corresponden la constante *V*.

Si un *SLF* es seguro, entonces todos los teoremas que se puedan derivar con dicho sistema serán asertos ciertos y, en ese caso, a la interpretación se le llama un *modelo de la lógica*. Si definimos el conjunto $hechos = \{asertos\ ciertos\ que\ se\ expresan\ como\ fórmulas\}$ y $teoremas = \{conjunto\ de\ fórmulas\ demostrables\}$, entonces se cumplirá la siguiente relación de inclusión, $teoremas \subseteq hechos$, sii el *SLF* posee la propiedad de seguridad.

Si un *SLF* posee la propiedad de de *complección*, entonces todo aserto cierto es demostrable, ésto es, $hechos \subseteq teoremas$.

Desafortunadamente, los programas no pueden tener una axiomatización completa como sistema lógico (ningún *SLF* de utilidad para la verificación de programas cumple la propiedad de *complección*). Sin embargo, para la verificación de programas, es suficiente contar con un *SLF* que tenga la propiedad de *complección relativa*. En el caso de la *Lógica de Programas*, las sentencias del lenguaje no demostrables, tales como las propiedades aritméticas, son ciertamente verdaderas.

# 2 Lógica Proposicional

Es un claro ejemplo de *SLF* que formaliza lo que normalmente llamamos el razonamiento basado en el *sentido común*. Las fórmulas de esta lógica se llaman *proposiciones* y sus símbolos son:

- constantes proposicionales $\{V, F\}$,
- las variables proposicionales: $\{p, q, r, \ldots\}$,
- los operadores: $\{\neg, \wedge, \vee, \rightarrow \ldots\}$,
- expresiones que utilizan constantes, variables y operadores.

[1] técnicamente se les suelen llamar asertos

## 2.1 Interpretación de una fórmula proposicional $p$

Dado un estado $s$ de un programa, descrito por una fórmula $P$, reemplazar cada variable proposicional $p$ por su valor en dicho estado y luego utilizar la tabla de verdad de los conectores lógicos para obtener el resultado. Se tienen, entonces, las siguientes definiciones:

- una fórmula se *satisface* en un estado "$s$" sii posee una interpretación cierta en dicho estado,
- *fórmula satisfascible* sii existe algún estado de programa en el cual la fórmula se puede satisfacer,
- *fórmula válida* sii se puede satisfacer en cualquier estado.

A las proposiciones válidas se les llama tautologías. En una lógica proposicional que cumple con la propiedad de seguridad, todos los axiomas son tautologías, ya que éstos han de ser siempre válidos.

**Tautologías o leyes de equivalencia proposicionales más utilizadas**
Son leyes de equivalencia que permiten reemplazar una proposición por su equivalente y, de esta forma, permiten la simplificación de fórmulas complejas:

1. Ley de negación: $\mathrm{P} = \neg(\neg\mathrm{P})$
2. Ley de los *medios excluidos*: $\mathrm{P} \vee \neg\mathrm{P} = V$
3. Ley de contradicción: $\mathrm{P} \wedge \neg\mathrm{P} = F$
4. Ley de implicación $\mathrm{P} \rightarrow \mathrm{Q} \equiv \neg\mathrm{P} \vee \mathrm{Q}$
5. Ley de igualdad $(\mathrm{P} \rightarrow \mathrm{Q}) \wedge (\mathrm{Q} \rightarrow \mathrm{P}) \equiv (\mathrm{P} = \mathrm{Q})$
6. Leyes de simplificación *Or*:
   - $\mathrm{P} \vee \mathrm{P} = \mathrm{P}$
   - $\mathrm{P} \vee V = V$
   - $\mathrm{P} \vee F = P$
   - $\mathrm{P} \vee (\mathrm{P} \wedge \mathrm{Q}) = \mathrm{P}$
7. Leyes de simplificación-*And*:
   - $\mathrm{P} \wedge \mathrm{P} = \mathrm{P}$
   - $\mathrm{P} \wedge V = P$
   - $\mathrm{P} \wedge F = F$
   - $\mathrm{P} \wedge (\mathrm{P} \vee \mathrm{Q}) = \mathrm{P}$
8. Leyes conmutativas:
   - $(\mathrm{P} \wedge \mathrm{Q}) = (\mathrm{Q} \wedge \mathrm{P})$
   - $(\mathrm{P} \vee \mathrm{Q}) = (\mathrm{Q} \vee \mathrm{P})$

- $(P = Q) = (Q = P)$

9. Leyes asociativas:
   - $P \wedge (Q \wedge R) = (P \wedge Q) \wedge R$
   - $P \vee (Q \vee R) = (P \vee Q) \vee R$
10. Leyes distributivas:
   - $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
   - $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
11. Leyes de Morgan:
   - $\neg(P \wedge Q) = \neg P \vee \neg Q$
   - $\neg(P \vee Q) = \neg P \wedge \neg Q$
12. Eliminación-*And*: $(P \wedge Q) \rightarrow P$
13. Eliminación-*Or*: $P \rightarrow (P \vee Q)$

La regla 12 se puede explicar diciendo que el conjunto de estados que cumplen P incluye al conjunto de los que cumplen $P \wedge Q$; se dice que la proposición P es más débil y, por tanto, se *ve implicada* por la expresión (*más fuerte*) $P \wedge Q$. De acuerdo con lo anterior, la constante $F$ es la proposición *más fuerte*, ya que implica a cualquier otra de la lógica. La constante $V$ es la proposición más débil, ya que sería *implicada* por cualquier proposición.

# 3 Lógica de Predicados

La Lógica Proposicional nos da la base para el razonamiento sobre las propiedades que cumplen los programas, construyendo fórmulas interpretables a partir de los valores que alcanzan las variables durante la ejecución de los programas. Sin embargo, se trata de un *SLF* demasiado restrictivo, ya que sólo define variables de un solo tipo de datos (proposiciones lógicas), y los programas utilizan más tipos de datos , además del tipo lógico. Para resolver este problema y otros derivados de la manipulación de expresiones y representación de conjuntos de valores, la Lógica de Predicados introduce los siguientes símbolos: {Símbolos Lógica de Predicados} = {{símbolos lógico−proposicionales}, {variables no−proposicionales}, {operadores relacionales},-{cuantificadores}}. Los cuantificadores se refieren al existencial $\exists$ y al universal $\forall$ y se utilizan para construir asertos definidos sobre conjuntos de valores. Se define el concepto de *predicado* como aquella fórmula que puede incluir expresiones relacionales y cuantificadores.

## 3.1 Cuantificador universal

Permite afirmar que todos los elementos de un conjunto satisfacen una propiedad $\forall b_1, b_2, \ldots b_n : R : P$. Por lo tanto, las variables $b_1, b_2, \ldots b_n$ se dice que son *ligadas*. $R$ es una fórmula que indica el conjunto de valores que pueden alcanzar las variables ligadas y $P$ es un predicado. La

interpretación es que el predicado $P$ será cierto para todas los valores de las variables ligadas en $R$. $(\forall i : 1 \leq i \leq n : a[i] = 0)$, indica que en ese conjunto de valores a = 0.

## 3.2 Cuantificador existencial

permite afirmar que alguno o todos los valores pertenecientes a un conjunto satisfacen una determinada propiedad. $(\exists b_1, b_2, \ldots, b_n : R : P)$, indica que $P$ es cierto si $P$ lo es para alguna combinación de valores de las variables ligadas.

Una variable se dice que es *libre* si no está afectada por un cuantificador o, incluso, si apareciera en el ámbito del cuantificador, si es distinta de cualquiera de las variables ligadas de dicho cuantificador. $i = j \wedge (\exists i, k : i, k > 0 : a[i] = a[k])$: la primera aparición de $i$ es libre, pero las otras apariciones de $i$ en la fórmula son ligadas.

**El problema de la *captura* de variables**
Puede existir conflicto entre los nombres de las variables libres y ligadas en las fórmulas de la lógica. Se conoce con el nombre del *problema de la captura*, cuando el nuevo nombre de una variable ligada ocasiona que las apariciones anteriores de una variable libre de la fórmula se conviertan en *ligadas*.

**Leyes de equivalencia de los cuantificadores**

1. Morgan:
   - $(\exists B : R : P) = \neg(\forall B : R : \neg P)$
   - $(\forall B : R : P) = \neg(\exists B : R : \neg P)$
2. Conjunción: $(\forall B : R : P \wedge Q) = (\forall B : R : P) \wedge (\forall B : R : Q)$
3. Disyunción: $(\exists B : R : P \vee Q) = (\exists B : R : P) \vee (\exists B : R : Q)$
4. Rango vacío:
   - $(\forall B : 0 : P) = V$
   - $(\exists B : 0 : P) = F$

# 4 Lógica de Programas

La Lógica de Programas (LP) es un *SLF* que permite construir proposiciones acerca de la ejecución de programas, facilitando la demostración de propiedades relativas a la ejecución del programa. Los símbolos de la LP incluyen a las sentencias de los lenguajes de programación y sus fórmulas, que se denominan *triples*, tienen la forma $\{P\}\, S\, \{Q\}$; donde P y Q son predicados, S es una sentencia simple o estructurada de un lenguaje de programación. Las variables libres de P y Q pertenecen al programa o son *variables lógicas*. Esta últimas actúan como un recipiente de los valores de las variables *comunes* del programa y no pueden ser asignadas más de una

vez, ésto es, mantienen siempre el valor al que fueron asignadas la primera vez. Aparecen sólo en predicados, no en las sentencias del lenguaje de programación y se suelen representar con letras mayúsculas para distinguirlas de la variables del programa.

**Interpretación de los triples:**
$\{P\}\,S\,\{Q\}$ se interpreta diciendo que será cierto siempre que la ejecución de S comience en un estado del programa que satisfaga el predicado P (*precondición*) y que el estado final, después de cualquier entrelazamiento de instrucciones atómicas resultado de ejecutar S, ha de satisfacer el predicado Q (*postcondición*). Los predicados asociados a un triple se les denomina *asertos*, ya que afirman que un estado del programa ha de satisfacer el predicado para que la interpretación del triple proporcione el valor V (*verdadero*). Un aserto caracteriza un estado *aceptable* del sistema, és decir, un estado que podría ser alcanzado por el programa si sus variables tomaran unos determinados valores. Por lo tanto, el aserto representado por la constante lógica $V$ caracteriza a todos los estados del programa, ya que dicho aserto se cumple en cualquier estado del programa independientemente de los valores que tomen las variables. Por otra parte, un aserto que sea equivalente a la constante lógica $F$ no se cumple en ningún estado del programa.

## 4.1 Axiomas y reglas de inferencia de la Lógica de Programas

1. Axioma de la sentencia nula $\{P\}\,null\,\{P\}$: si el predicado es cierto antes de ejecutarse la sentencia nula, permanecerá como cierto cuando dicha sentencia termine.

   *Sustitución textual*: $\{P_e^x\}$ es el resultado de sustituir la expresión $e$ en cualquier aparición libre de la variable $x$ en P. Los nombres de las variables libres de la expresión $e$ no deben entrar en conflicto con las variables ligadas que existan en $P$, para evitar el problema de la captura. Su significado es que cualquier relación del estado de programa que tenga que ver con la variable $x$ y que sea cierto después de la asignación, también ha de haber sido cierto antes de la asignación.

2. Axioma de asignación $\{P_e^x\}\,x := \{P\}$: una sentencia de asignación asigna un valor $e$ a una variable $x$ y, por lo tanto, generalmente, cambia el estado del programa. Una asignación cambia sólo el valor de la variable objetivo, el resto de las variables conservan los mismos valores que antes de la asignación. $\{V\}\,x := 5\ \{x = 5\}$ es un *teorema* (triple cierto), ya que $\{x = 5\}_5^x \equiv V$.

   Existe una regla de inferencia, en la Lógica de Programas, para cada una de las sentencias que afectan al flujo de control en un programa secuencial estructurado. Así como una regla de inferencia adicional para *conectar* los triples en las demostraciones de los programas.

3. Regla de la consecuencia (1)

$$\frac{\{P\}\,S\,\{Q\},\ \{Q\} \rightarrow \{R\}}{\{P\}\,S\,\{R\}}$$

   el significado de esta regla es que siempre se puede hacer más débil la postcondición de un triple y que su interpretación se mantenga ( que el triple siga siendo cierto).

4. Regla de la consecuencia (2)

$$\frac{\{\mathrm{R}\} \rightarrow \{\mathrm{P}\},\ \{\mathrm{P}\}\mathrm{S}\{\mathrm{Q}\}}{\{\mathrm{R}\}\,\mathrm{S}\,\{\mathrm{Q}\}}$$

el significado de esta regla es que siempre se puede hacer más fuerte la precondición de un triple y que su interpretación se mantenga ( que el triple siga siendo cierto).

5. Regla de la composición

$$\frac{\{P\}\,S_1\,\{Q\},\ \{Q\}\,S_2\,\{R\}}{\{P\}\,S_1; S_2\,\{R\}}$$

permite obtener la postcondición y la precondición de 2 sentencias juntas, a partir de la precondición de la primera y de la postcondición de la segunda, si la postcondición de la primera coincide con la precondición de la segunda.

6. Regla del *if*

$$\frac{\{P\} \wedge \{B\}\,S_1\,\{Q\},\ \{P \wedge \neg B\}\,S_2\,\{Q\}}{\{P\} if\ B\ then S_1\ else\ S_2\ fi\,\{Q\}}$$

suponemos que precondición de la sentencia (if) que queremos demostrar es $\{P\}$ y la postcondición a la que queremos llegar es $\{Q\}$, entonces, para demostrarlo, sólo debemos probar que los 2 triples que constituyen las hipótesis de la regla tienen una interpretación cierta.

7. Regla de la iteración

$$\frac{\{I \wedge B\}\,S\,\{I\}}{\{I\}\,while\,B\,do\ S\ enddo\,\{I \wedge \neg B\}}$$

Una sentencia {while} podrá iterar un número arbitrario de veces, incluso 0. Por dicha razón, la regla de la inferencia iterativa está basada en un invariante del bucle: un predicado $I$ que se satisface antes y después de cada iteración del bucle.

## 4.2 Verificación de sentencias concurrentes y de sincronización

En la verificación de los programas concurrentes se produce un problema, conocido como el de la *interferencia*, que cuando se produce invalida las demostraciones individuales de los procesos. Se debe a que un proceso puede ejecutar una instrucción atómica que haga falsa la precondición (o postcondición) de una sentencia simultáneamente ejecutada por otro proceso.

Si entendemos la ejecución de un programa concurrente como un entrelazamiento de las instrucciones atómicas ejecutadas por los procesos del programa, entonces hemos de tener en cuenta para la demostración de corrección que no todas las secuencias de entrelazamiento resultan ser aceptables. Para poder programar correctamente, se utilizan sentencias de sincronización en los lenguajes de programación concurrentes:

- para combinar *instrucciones atómicas elementales* en acciones atómicas compuestas. Se introducen *secciones críticas* en el código del proceso que han de ser ejecutadas respetando la propiedad de exclusión mutua.

- para retrasar la ejecución de un proceso hasta que se satisface un predicado, el cual indica que el programa ha alcanzado un determinado estado. A este tipo de construcción se le llama *sincronización con una condición.*

**Acción atómica elemental:**
realiza una transformación indivisible del estado del programa. Cualquier estado intermedio que pudiera existir en la implementación de dicha sentencia no sería visible para el resto de los procesos.

En los programas secuenciales, las asignaciones aparecen como acciones atómicas, puesto que no hay ningún estado intermedio visible al resto de los procesos; sin embargo, ésto no ocurre en los programas concurrentes, ya que, a menudo, una asignación es implementada mediante una secuencia de operaciones atómicas elementales. El siguiente programa puede producir como resultado final x $\in \{0, 1, 2, 3\}$, dependiendo de que se cargue (load y) y se incremente (add z) el registro [2] antes, después o al mismo tiempo, de que se ejecuten las 2 operaciones de asignación ($y := 1$ y $z := 2$) del componente derecho de la instruccción `||` de composición concurrente de los procesos:

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

Una peculiaridad del programa anterior es que puede producir el valor final de $x = 2$ a pesar de que $z + y = 2$ no se corresponde con ningún estado del programa.

**Evaluación atómica de expresiones:**
Una expresión que no hace referencia a ninguna variable modificada por otro proceso será evaluada de forma atómica por el proceso que la contiene, incluso si dicha expresión está compuesta por varias intrucciones atómicas elementales. Ésto es debido a que ninguno de los valores de los que depende la expresión puede modificarse mientras dicha expresión resulta evaluada. El programa siguiente cumple con esta condición y, por esta razón, las 2 asignaciones que realiza se pueden considerar atómicas. Las variables del programa alcanzan siempre los valores x=y=1, independientemente de la secuencia de entrelazamiento de los procesos.

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

**Propiedad como máximo una vez**:
Como la mayoría de los programas concurrentes no satisafacen la propiedad anterior, se define una *condición más débil* para considerar la evaluación de las expresiones como atómica. Dicha condición dice que la evaluación de una expresión se realiza de forma atómica si las variables compartidas son leídas o escritas por un único proceso, ésto es, una sola vez. Una variable compartida puede ser leída por un proceso diferente del que la escribe siempre que en la expresión no figure ninguna otra variable modificada externamente. El siguiente proceso (1) no cumple la condición anterior, pero los procesos (2) y (3) sí la cumplen:

1. `x:=0; y:=0; cobegin x:= y+1 || y:= x+1 coend;`

[2] ($x := y + z \equiv$ load y; add z; store x)

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

## 4.3 Especificación abstracta de una instrucción de sincronización

Para garantizar que una expresión, que no cumple con la condición *como máximo una vez*, se ejecuta de forma atómica, necesitaremos que el lenguaje de programación posea un mecanismo de sincronización para construir una instrucción atómica no elemental. Se puede expresar, de forma abstracta, como sigue:

1. acción atómica elemental: indicando que un conjunto de instrucciones del código del programa pasa a ejecutarse como una sección crítica: `<x:=x+1 ; y:=y+1>`. La sección crítica indicada en este caso engloba a ambas asignaciones, por lo que el estado interno en el cual $x$ ha sido incrementada, pero $y$ todavía no lo ha sido, no es visible para el resto de los procesos que hacen referencia a $x$ ó $y$.
2. acción atómica condicional: se le llama sentencia de espera (*await*) con *guarda*. La *guarda* se trata de una condición de espera y la sentencia componente $S$ es una secuencia de instrucciones que han de terminar necesariamente (no puede incluir bucles indefinidos): $<$ await B $\rightarrow$ S $>$. La condición B que compone la guarda ha de ser cierta cuando comience la ejecución de $S$. Ningún estado intermedio (dentro de la secuencia interna que compone S) es visible al resto de los procesos del programa. La sentencia *await* puede ser utilizada para expresar acciones atómicas compuestas de una complejidad arbitraria en lenguajes de programación concretos. En general, dicha sentencia tiene un valor más bien teórico, como una abstracción de una operación de sincronización, ya que resulta difícil de implementar eficientemente.

Regla de la sincronización:

$$\frac{\{P \wedge B\}\, S\, \{Q\}}{\{P\} < await\, B \rightarrow S > \{Q\}}$$

Existen 2 casos particulares de esta regla que es interesante comentar: (1) si la guarda es idénticamente equivalente a la constante $V$, entonces la acción condicional degenera en la acción atómica $<$ S $>$ y la demostración de la regla se simplifica a probar el siguiente triple: {P} S {Q}; (2) si no existe la sentencia componente $S$, la acción atómica condicional degenera en una instrucción de sincronización elemental: $<$ await B $>$ y la demostración de la regla se simplifica a {P $\wedge$ B} $\rightarrow$ {Q}.

# 5 Semántica de la ejecución concurrente

Dado un triple cualquiera {P} S {Q} de un programa concurrente, los asertos críticos son: $\{Q\}$ y las pre- y post-condiciones de las acciones atómicas incluidas en la sentencia componente $S$

que no estén incluidas dentro de una sentencia await [3].

Se dice que la instrucción compuesta $a$ es una *acción de asignación* si se ejecuta de forma atómica y contiene 1 ó más instrucciones elementales de asignación.

**Interferencia en la demostración de un proceso**
La única forma de que un *aserto crítico* C que aparece en la demostración de un proceso pueda ser interferido sería que otro proceso ejecute una acción de asignación que haga falso el valor de C. El primer ejemplo que se muestra a continuación corresponde a una demostración válida, los valores finales de las variables serán siempre x=y=1, puesto que en el primer caso no puede existir *interferencia* entre los asertos críticos y las acciones de asignación; sin embargo, el segundo ejemplo representa una construcción que no es segura, ya que cada una de las asignaciones interfiere con los asertos críticos de la otra,

1. $\{x = 0 \wedge y = 0\}$cobegin $< x := x + 1 > \; || \; < y := y + 1 >$ coend; $\{x = 1 \wedge y = 1\}$
2. $\{x = 0\}$cobegin $< x := x + 1 > \; || \; < x := x + 1 >$ coend; $\{x = 2\}$

## 5.1 Regla de la No-Interferencia

La acción de asignación $a$ no interfiere con el aserto crítico $C$ si el triple

$$\mathrm{NI(a,\ C)} \equiv \{\mathrm{C} \wedge \mathrm{pre(a)}\}\,\mathrm{a}\,\{\mathrm{C}\}$$

puede demostrarse como un teorema con las reglas y axiomas de la Lógica de Programas. El significado del aserto anterior sería que el aserto C es invariante con respecto a la ejecución de la acción de asignación $a$, la cual, para poder ejecutarse, ha de iniciarse en un estado que satisfaga $pre(a)$. Si fuera necesario, para poder llevar a cabo la demostración correctamente, habría que renombrar las variables locales de C para evitar el problema de su *captura* por las variables locales de $a$ y $pre(a)$.

Se dice que un conjunto de procesos está libre de interferencia si no existe ninguna acción de asignación dentro de ningún proceso que interfiera con algún aserto crítico de otro proceso.

Regla de la Concurrencia:

$$\frac{\{\mathrm{P_i}\}\,\mathrm{S_i}\,\{\mathrm{Q_i}\} \text{ son teoremas libres de interferencia, } 1 \leq \mathrm{i} \leq \mathrm{n}}{\{\mathrm{P_1} \wedge \mathrm{P_2} \wedge \ldots \wedge \mathrm{P_n}\}\mathrm{cobegin}\mathrm{S_1}||\mathrm{S_2}||\ldots||\mathrm{S_n}\mathrm{coend}\{\mathrm{Q_1} \wedge \mathrm{Q_2} \wedge \mathrm{Q_n}\}}$$

El significado de la regla anterior es que si un conjunto de procesos concurrentes está libre de interferencia, entonces las demostraciones de los procesos secuenciales individuales son válidas, incluso si los procesos se ejecutan concurrentemente.

**Invariantes Globales**
se pueden utilizar para demostrar que los procesos de un programa concurrente cooperan, y por tanto la corrección parcial del programa, sin necesidad de demostrar todos los teoremas de no-interferencia que serían necesario probar aplicando la regla anteriormente referida. Un Invariante Global (IG) es un predicado que captura la relación existe entre las variables (globales) compartidas por los procesos. IG es un invariante global válido si,

[3] ya que si lo estuvieran no serían visibles al resto de los procesos del programa

1. es cierto para los valores iniciales de las variables,
2. se mantiene cierto después de la ejecución de cada acción de asignación, ésto es, $\{I \wedge pre(a)\}\, a\, \{I\}$

Si todo aserto crítico C de un proceso secuencial del programa, cuya corrección queremos demostrar, puede ser escrito como la conjunción de un invariante global (I) y de un *invariante local* al proceso que contiene el aserto C, $C \equiv I \wedge L$, entonces las demostraciones de los procesos secuenciales están ya libres de interferencia y no sería necesario el aplicar la *regla de la concurrencia.*

# 6 Técnicas para evitar la interferencia en las demostraciones de los procesos concurrentes

El número de maneras en que los procesos de un programa concurrente pueden interferir depende sólo del número de acciones atómicas diferentes que aparecen en el texto del programa. Si hay $n$ procesos y cada uno de ellos contiene $a$ acciones atómicas y $c$ asertos críticos, entonces deberíamos demostrar, en el peor de los casos, $n \cdot (n-1) \cdot a \cdot c$ teoremas de no interferencia. Aunque este último número es mucho menor que el del número de posibles entrelazamientos de instrucciones atómicas del programa, el cual tiene una complejidad exponencial con respecto al número de procesos, sigue siendo un número muy elevado. No obstante, el método basado en la demostración de los teoremas de no interferencia es factible de llevar cabo puesto que muchos teoremas coincidirán, ya que los procesos son a menudo sintácticamente idénticos o, al menos, simétricos.

## 6.1 Variables disjuntas

Si el *conjunto de escritura* de un proceso (compuesto por las variables a las que asigna valores)es disjunto de su *conjunto de referencia* (variables que *lee* el proceso en sus asertos), no puede existir interferencia entre los asertos críticos de uno y las acciones atómicas del otro, ya que el axioma de asignación sólo afecta a aquellos asertos de la demostración que contienen referencias a la variable objetivo de la asignación. El triple $\{x = 0 \wedge y = 0\}$ cobegin $< x := x + 1 >$ || $< y := y + 1 >$ coend$\{x = 1 \wedge y = 1\}$ satisface la demostración de *no-interferencia*, ya que NI($y := y + 1$, $\{x = 0\}$), NI($x := x + 1$, $\{y = 0\}$), etc. son teoremas.

## 6.2 Asertos debilitados

Cuando los conjuntos de lectura y de escritura de los procesos se solapan, a veces, se puede evitar la interferencia *debilitando* los asertos para tener en cuenta los efectos de la ejecución concurrente de los procesos en la siguiente demostración,

$$x := 0;\ \text{cobegin}\ P_1 :< x := x + 1 >\ ||\ P_2 :< x := x + 1 > \text{coend}$$

Los triples referidos a los 2 procesos secuenciales anteriores son válidos de manera individual:

1. $P_1 : \{x = 0\} < x := x + 1 > \{x = 1\}$
2. $P_2 : \{x = 0\} < x := x + 2 > \{x = 2\}$

pero cada asignación interfiere con lo asertos del otro triple. Además la conjunción de las postcondiciones no produce el resultado correcto que indicaría el valor final (x=3). La interferencia se puede corregir si se tiene en cuenta la posible ejecución del otro proceso antes comenzar la ejecución del proceso actual:

- $\{x = 0 \vee x = 1\} < x := x + 2 > \{x = 2 \vee x = 3\}$
- $\{x = 0 \vee x = 2\} < x := x + 1 > \{x = 1 \vee x = 3\}$

la inclusión de las condiciones que indican la posible ejecución del otro proceso, como una disyunción, producen unos asertos más débiles que los iniciales. Sin embargo, con los nuevos asertos debilitados, si es posible demostrar la no interferencia entre las demostraciones individuales de $P_1$ y $P_2$: $NI(\{x = 0 \vee x = 1\}, x := x + 1)\{x = 0\}\, x := x + 1\, \{x = 0 \vee x = 1\}$ es un triple cierto y los otros 3 triples se demuestran de forma análoga.

Por lo tanto, ya que hemos demostrado que no existe interferencia, podemos concluir que el triple que incluye la pre y postcondición del programa completo: $\{x = 0\}$ cobegin $< x := x + 1 > \,||\, < x := x + 2 >$ coend $\{x = 3\}$ es también un teorema.

**Variables auxiliares**

Con las técnicas estudiadas anteriormente para evitar la interferencia entre las demostraciones de los procesos individuales de un programa concurrente no se puede llegar a demostrar la validez de todos los triples. Este efecto es debido a que falla la propiedad de *complección* para la Lógica de Programas, ya que, como dijimos anteriormente, este *SLF* es sólo *relativamente completo.* El origen del problema se haya en que necesitaremos a menudo hacer asertos explícitos en las demostraciones acerca de los valores que toman los contadores de programa de los diferentes procesos. Sin embargo, dichos contadores no se almacenan en ninguna variable del programa, pertenecen a lo que se llama el *estado oculto* del programa. Por ejemplo, considérese el siguiente programa:

$$\{x = 0\};\ \text{cobegin } < x : +x + 1 > \,||\, < x : +x + 1 > \ \text{coend};\{x = 2\}$$

con la técnica de asertos debilitados no se puede demostrar que el valor final que toma la variable $x$, como resultado del programa anterior, es 2. El origen del problema está en que ambos procesos incrementan el valor de la variable $x$ en 1 y la postcodición de esa acción de asignación $\{x = 1\}$ no contiene ninguna información acerca de cual de los 2 procesos se ha ejecutado primero. Para poder resolverlo, hay que incluir en el aserto anterior información explícita acerca del orden de ejecución de los procesos:

$$\{x = 0\}; t_1 := 0; t_2 := 0;\ \text{cobegin } < x : +x{+}1 >; t_1 := 1 \,||\, < x : +x{+}1 > \ t_2 := 1\ \text{coend}; \{x = 2\}$$

En el estado inicial x = $t_1 + t_2$. Este invariante también se cumple en el estado final. Si escribimos las 2 instrucciones de asignación de cada proceso dentro de una acción atómica de asignación, entonces podemos decir que I : x = $t_1 + t_2$ es un invariante global. Podríamos re-escribir la demostración del programa completo de la siguiente forma:

```
var x:=0; t1:=0; t2:=0 {I: x= t1 + t2}
    {I && t1= 0 && t2= 0}
    cobegin P1:{I && t1 = 0} <x:=x+ 1; t1:=1> {I && t1 = 1}
     || P2:{I && t2 = 0} <x:=x+ 1; t2:=1> {I && t2 = 1}
    {I && t1= 1 && t2= 1}
```

Puesto que los asertos están escritos en la forma I $\wedge$ L, es decir, como una conjunción de un invariante global y de una variable no referenciada por los otros procesos, etonces podemos concluir que las demostraciones de los procesos están libres de interferencia, asímismo que el programa termina y que la postcondición final es (I $\wedge$ $t_1 = 1 \wedge t_2 = 1) \rightarrow x = 2$.

**Sincronización**

Puesto que una acción atómica aparece a los otros procesos como una unidad indivisible, sólo es necesario probar que la ejecución de la acción completa no ocasiona interferencia. Por ejemplo, dada la acción de asignación $< x := x + 1; y := y + 1 >$, ninguna de las 2 asignaciones puede ocasionar interferencia por sí sola, únicamente el par de asignaciones considerado globalmente puede ocasionar interferencia. Además, los estados internos de las acciones de asignación no son visibles a los otros procesos. Por lo tanto, el aserto A intermedio del triple siguiente no se le puede considerar como un aserto crítico,

$$\{x = 0 \wedge y = 0\} < x := x + 1 \{A\} y := y + 1 > \{x = 1 \wedge y = 1\}$$

$$A \equiv \{x = 1 \wedge y = 0\}$$

Hay 2 técnicas para evitar la interferencia en las demostraciones: (1) utilizar exclusión mutua (ésto es, imponerla mediante secciones críticas), (2) mediante sincronización con condiciones.

Considérense los siguientes esquemas de demostraciones:

```
P1:: ... {pre(a)} a ...
P2:: ... S1{C}S2 ...
```

Suponemos que la acción de asignación $a$ interfiere con el aserto crítico C. Una manera de evitar la interferencia en la demostración que tenemos que llevar a cabo sería el ocultar el aserto crítico C de $a$, construyendo una acción de asignación atómica con $S_1$ y $S_2$ : $< S_1 ; S_2 >$, lo cual convertiría al aserto crítico C en invisible para los otros procesos.

La otra forma de eliminar la interefernci en la demostración anterior sería el utilizar una condición de sincronización para fortalecer la precondición de $a$. En este caso, la condición de *no-interferencia* se satisfará si:

- C es falso cuando se ejecuta $a$ y, por tanto, el proceso $P_2$ no podría estar a punto de ejecutar $S_2$.

- El ejecutar $a$ hace que C sea cierto, ésto es, si postcondicion(a) $\rightarrow$ C

  Nótese que el caso que podría llevar a la interferencia sucede cuando C se evalúa como cierto y (simultáneamente) se ejecuta la acción $a$. Por lo tanto, podemos evitar la interferencia en la demostración anterior si reemplazamos $a$ por la acción atómica condicional:

  $$< \text{await NOT C or B} \rightarrow \text{a} >$$

  donde B es un predicado que caracteriza el mayor conunto posible de estados que, tras ejecutar $a$, harán cierto al aserto crítico C. Se puede utilizar el predicado *pre-condición más débil* B= wp(a, C) para caracterizar a dicho conjunto de estados.