# Derivación de Multiprogramas[1]

Nicolás Wolovick

16 de Febrero de 2000

[1] Trabajo final de la Licenciatura en Computación, bajo la dirección del Dr. Javier Blanco.

# Índice General

# Capítulo 1

# Introducción

## 1.1 Descripción General

En los días que corren los programas de computadoras se presentan en casi cualquier ámbito de nuestras vidas. Servicios esenciales como el transporte, la seguridad, el comercio y las comunicaciones se apoyan sobre cimientos construidos con programas. La necesidad de poder verificar que éstos satisfacen nuestros requerimientos, se ha convertido en una de las áreas más activas de las Ciencias de la Computación.

Cada vez más estos programas de computadoras trabajan en conjunto para poder desempeñar de manera eficaz, rápida y segura una tarea determinada. Ellos interactúan y se sincronizan para cumplir con un objetivo. Este campo es conocido como *Programación Concurrente*. Ejemplos clásicos incluyen bases de datos distribuidas como los sistemas bancarios o la reserva de pasajes aéreos, el procesamiento numérico de grandes cantidades de datos como las simulaciones físicas, los sistemas operativos multitarea y distribuidos y los entornos gráficos de ventanas entre otros.

Mucho se ha dicho y escrito acerca de como los métodos formales de verificación de programas secuenciales evitan que la corrección de éstos dependa de manera exclusiva de la pericia (¿hacking?) del programador para encotrar casos extraños o límites, donde nuestros requerimientos puestos en forma de propiedades no se ven satisfechos. Además de minimizar la posibilidad de falla, estos métodos han forjado una manera distinta de pensar los programas y su proceso de creación.

En los programas concurrentes la complejidad se eleva de tal forma que la intuición o la experiencia de un programador hábil resultan totalmente insuficientes, debido a que la cantidad de secuencias posibles de ejecución a tener en cuenta crece de manera exponencial respecto a la cantidad de líneas de código y número de componentes del programa concurrente. ¿Quién puede asegurar que en la siguiente corrida una componente se atrase unas pocas instrucciones, y provoque que el programa llegue a un estado que no fue tenido en cuenta y alguna propiedad fundamental no se cumpla?.

Los errores son más la regla que la excepción en la programación concurrente, y el habitual razonamiento informal de tipo operacional no basta para asegurar propiedades en esta clase de programas. La cantidad de algoritmos concurrentes o demostraciones informales de éstos que son incorrectos dan la pauta de la necesidad de un método formal y sistemático para luchar contra esta complejidad.

Uno de los acercamientos más usados para la verificación de programas secuenciales es el *asercional*, donde los programas son decorados con fórmulas de la lógica denominadas justamente *aserciones*.

Hoare en 1969 creó un sistema deductivo que influyó de manera definitiva en el campo de la verificación sistemática de programas. Con axiomas y reglas de inferencia, este sistema permitía demostrar de manera sintáctica ternas de la forma $\{P\}S\{Q\}$ para programas secuenciales determinísticos $S$. Además, este juego sintáctico era consistente respecto a un modelo operacional abstracto de computación que seguía de manera razonable lo que una computadora era capaz de realizar. En este sistema, cada *terna de Hoare* $\{P\}S\{Q\}$ se interpreta de la siguiente forma: si antes de ejecutar $S$ tenemos un estado de entrada $p$ tal que

se da $P(p)$, entonces luego de ejecutar $S$ y en caso de que termine esta sentencia, el estado de salida $q$ hará verdadero a $Q$. Esto es llamado *corrección parcial.*

Esta manera asercional de probar corrección de programas secuenciales fue extendida en 1976 por la tesis de postgrado de Susan Owicki bajo la supervisión de David Gries. En ella se ampliaba el lenguaje para incluir operadores de *composición paralela* y *sincronización.* El sistema deductivo fue transformado convenientemente para estos *programas paralelos con variables compartidas.* La consistencia fue demostrada respecto del modelo operacional tradicional de ejecución paralela de programas, conocido como *modelo de interleaving.*

Como es bien sabido, la corrección puede ser respecto a *safety properties* o a *liveness properties* [And91], y la programación paralela introduce nuevas propiedades de seguridad que deben ser comprobadas, como *interference freedom*, *deadlock freedom* y exclusión mutua entre otras. Probablemente las *liveness properties* sean las más complicadas de demostrar, y entre ellas están la terminación y el progreso, donde ésta resulta de generalizar el concepto de terminación para programas que no lo hacen. El punto conflictivo está en que estas propiedades de *liveness* son afectadas por lo que se conoce como *fairness*, que es un concepto que surge cuando se involucra el no-determinismo.

En esta tesis se trataron los problemas de corrección parcial, ausencia de interferencia, terminación y ausencia de deadlock. Las pruebas de *liveness* no tienen una técnica definida, debido a que esta teoría no brinda un marco adecuado para su tratamiento, y además el problema es realmente complicado en su caso general. Sin embargo podemos encontrar casos particulares que resultan tratables dentro de la teoría de Owicki-Gries.

La efectividad de este método formal fue puesta a prueba por sus creadores dando demostraciones de corrección de varios algoritmos paralelos tradicionales como el *findpos*, *producer-consumer*, diferentes algorimos de exclusión mutua y el algoritmo de Dijkstra para recolección de basura al vuelo. Este último fue la prueba definitiva debido a la complejidad y el fino grado de interleaving que presenta, donde sólo los accesos a memoria se suponían indivisibles.

Sin embargo el método de Owicki-Gries tiene el defecto de que está pensado para realizar una demostración *a posteriori* de la construcción del programa. Dado un texto de programa y un conjunto de propiedades que son escritas como fórmulas de la lógica (dan por ejemplo la condición inicial, la final y algún conjunto de propiedades que se necesitan mantener invariantes), es necesario "rellenar" todo el resto y en la mayoría de los casos no triviales tendremos que inventar aserciones intermedias, variables e invariantes, así como transformar comandos por equivalentes, para que, utilizando el sistema deductivo podamos ver que las propiedades que nos interesan resultan teoremas.

Esta dificultad es, en parte, la culpable que los métodos formales para verificación *a posteriori* de la corrección no sean del todo aceptados, o más aún, rechazados en la comunidad que desarrolla software.

Dijkstra primero en 1976 y luego Gries en 1981 propusieron métodos sistemáticos para el desarrollo de programas junto con sus pruebas de corrección dentro de los sistemas deductivos "*à la Hoare*", definiendo la semántica de los *transformadores de predicados.* Con estos métodos de *derivación de programas*, la pruebas y el programa son construidos en forma conjunta. Teniendo como

base el comportamiento entrada/salida del algoritmo, las reglas de inferencia y los axiomas nos guían en la construcción de este. Dentro de este esquema la corrección del programa es implícita. Se lo ha construído en base a las reglas (de corrección) y no es posible, dentro de este juego, generar programas que no las cumplan.

Se ha pasado del arte a la disciplina y de la disciplina a la ciencia de la programación. Los resultados están a la vista cuando algoritmos difíciles de resolver diez o veinte años atrás son usados como ejemplos o ejercicios en cursos introductorios de programación.

Asi como Dijkstra utilizó como base el trabajo de Hoare para el desarrollo sistemático de programas secuenciales, la escuela de Eindhoven se basó en el sistema deductivo de Owicki y Gries para demostrar corrección de programas paralelos, y a partir del inicio de la década pasada comenzaron a realizar experimentos tendientes a demostrar que era posible derivar programas paralelos teniendo como base esta teoría. Trabajos de Feijen, van Gasteren, van der Sommen, Moerland y Hoogerwoord entre otros, pusieron en claro que el modelo criticado por su pobre manejo de problemas básicos de la programación concurrente, era una base sólida para la derivación de lo que ellos llaman *multiprogramas*. La simplicidad fue la principal razón que llevó a este grupo a elegir esta teoría como base de sus trabajos.

Estos métodos de derivación de multiprogramas se han demostrado útiles, y varios algoritmos concurrentes no triviales fueron derivados, como por ejemplo el algoritmo de exclusión mutua del pastelero de Lamport, el algoritmo de lectores/escritores con prioridad para los escritores, varios algoritmos de sincronización de fase, así como algunos protocolos básicos de comunicación.

## 1.2 Un Ejemplo Motivador

Trataremos de mostrar, a través de un ejemplo simple, algunos de los errores en los cuales podemos incurrir al tratar de construir programas paralelos en base a razonamientos operacionales informales. También indirectamente se apuntarán problemas propios de la programación paralela que han sido mencionados en la introducción.

**Problema 1.1 (búsqueda lineal paralela)** *Dada la función* $f : \mathcal{Z} \rightarrow Bool$, *tal que* $(Ni :: f.i) = 1$ *dar un algoritmo paralelo que encuentre esa única ocurrencia.*

Para paralelizar la búsqueda utilizaremos dos componentes, una se encargará de buscar en los enteros no negativos y la otra en los enteros no positivos.

Podemos pensar a las dos componentes secuenciales como independientes, salvo que comparten una variable *found* que servirá para finalizar la búsqueda cuando se haya encontrado el valor verdadero de la función. El texto del programa que busca en los enteros positivos será $S_1$, mientras que $S_2$ tratará de localizar el valor verdadero en la parte negativa de la función. Con $[S_1 \parallel S_2]$ estamos denotando la ejecución paralela de las instrucciones pertenecientes a las componentes secuenciales $S_1$ y $S_2$.

**Solución 1**

$S_1 \equiv$ `found:=false;`
`x:=0;`
**do** ¬`found` → `found:=f.x;`
`x:=x+1`
**od**

$S_2 \equiv$ `found:=false;`
`y:=0;`
**do** ¬`found` → `found:=f.y;`
`y:=y-1`
**od**

`FIND1` $\equiv [S_1 \parallel S_2]$

Está claro que cada componente tratada por separado resuelve el problema siempre y cuando el valor se encuentre en su rango.

Pensemos ahora qué es lo que sucede cuando se lanza la ejecución paralela $[S_1 \parallel S_2]$. Un primer argumento podría indicar que todo está bien, dado que si una componente termina esto implica que la variable *found* se hace verdadera, y la otra componente también termina. Sin embargo, una posible definición del operador de composición paralela es:

> "Cada una de las componentes avanza de manera independiente con una velocidad relativa de ejecución no especificada"

y esto implica que un escenario posible de ejecución sería:

- Se comienza a ejecutar la componente $S_1$ y se encuentra el valor, con lo que *found* se hace verdadero y termina su ejecución. La composición paralela no termina, debido a que todavía no terminó $S_2$.
- Comienza la ejecución de $S_2$, y el valor de *found* vuelve a ser falso y al no haber ningún valor verdadero en la parte negativa de la función, $S_2$ no termina y por lo tanto la composición paralela que la contiene tampoco lo hace.

Queda claro que la *composicionalidad* del nuevo operador deja mucho que desear. Cada una de las componentes cumple con lo que queremos, pero su composición paralela no se muestra como el efecto sumado de ambas.

Como el problema parece estar en la inicialización de *found*, modificamos el código para llegar a la siguiente etapa.

**Solución 2**

```
S1 ≡ x:=0;
     do ¬found → found:=f.x;
                 x:=x+1
     od

S2 ≡ y:=0;
     do ¬found → found:=f.y;
                 y:=y-1
     od

FIND2 ≡ found:=false; [S1 || S2]
```

El código ya no reinicializa la variable *found* en el inicio de cada componente, sin embargo se produce un fenómeno más o menos similar si se presenta el siguiente escenario de ejecución.

- La componente $S_1$ encuentra el valor verdadero en un $x$ dado y asigna a la variable *found* con ese valor.
- Luego de esto y antes que $S_1$ llegue a evaluar la guarda de la repetición, $S_2$ ejecuta la asignación a *found*, y el valor verdadero que tenía se pierde.
- Se evalúa la guarda de $S_1$ y como *found* es falso, el ciclo continúa de manera indefinida. Lo mismo sucede con la componente $S_2$.

Nuevamente el problema está en la falsificación de *found*. Para evitarlo sólo haremos asignaciones de esta variable con el valor verdadero dentro de las componentes de la composición paralela.

**Solución 3**

```
S1 ≡ x:=0;
     do ¬found → if f.x → found:=true
                    ¬f.x → skip
                 x:=x+1;
                 fi
     od

S2 ≡ y:=0;
     do ¬found → if f.y → found:=true
                    ¬f.y → skip
                 y:=y-1;
                 fi
     od

FIND3 ≡ found:=false; [S1 || S2]
```

Finalmente tenemos un texto de programa que parece hacer lo que buscábamos. La variable *found* es inicialmente falsa, y cada una de las componentes busca de manera independiente el valor verdadero en la parte positiva y negativa de la función. Si alguna encuentra ese valor, la variable *found* se hace verdadera,

por lo que las dos repeticiones terminan. Además, ya no tenemos el problema de la falsificación de *found* cuando éste es verdadero, pues dentro del bucle la variable de finalización sólo se asigna con el valor *true*.

No obstante, volviendo a lo que anteriormente dijimos que se entiende como ejecución paralela, no es descabellado pensar en una historia de ejecución donde sólo se ejecutan instrucciones de, por ejemplo, la componente $S_1$. Si la ejecución paralela está modelada por el *interleaving* de las instrucciones de ambas componentes, donde un *demonio* es el encargado de elegir de qué componente será la siguiente instrucción a ejecutar, nuestro caso patológico se presentará cuando el demonio tenga preferencia absoluta por $S_1$.

Está claro que si el valor verdadero de $f$ se encuentra en el dominio negativo, $S_1$ jamás terminará.

Podríamos objetar este tipo de escenario, puesto que si se da, más allá de que el valor se encuentre en la parte positiva o negativa de la función, ninguna instrucción de la componenete $S_2$ será ejecutada, esta no podrá terminar y por lo tanto la composición paralela así como nuestro programa tampoco lo harán.

Aunque esta asunción es más o menos descabellada, tratemos de solucionar este inconveniente.

Introduzcamos un nuevo constructor que permite la *sincronización* en la ejecución paralela. Su sintaxis es **await** $B \to S$, donde $B$ es una expresión booleana y $S$ es un comando. Su semántica dice: mientras la condición booleana no sea verdadera, la componente que tenga este comando como el próximo a ejecutar no podrá progresar. En términos de interleaving y demonios, podemos decir que este último no puede elegir una componente cuya siguiente instrucción es un **await** con su guarda que en el estado actual evalúa a falso. Si la condición es verdadera, entonces $S$ es ejecutado sin posibilidad de interrupción o interferencia de otras componentes, esto último es también llamado *ejecución atómica*

Creamos entonces una nueva versión donde la variable *turn* podrá valer 1 o 2 e indica qué componente tiene el turno para buscar.

**Solución 4**

```
S1 ≡ x:=0;
     do ¬found → await turn=1 → turn:=2;
                 if f.x → found:=true
                    ¬f.x → skip
                 fi
                 x:=x+1;
     od

S2 ≡ y:=0;
     do ¬found → await turn=2 → turn:=1;
                 if f.y → found:=true
                    ¬f.y → skip
                 fi
                 y:=y+1;
     od

FIND4 ≡ turn:=1; found:=false; [S1 || S2]
```

Con esta línea de código agregada a cada componente, aseguramos la ejecución alternada de las dos componentes más allá de como el demonio elije la siguiente componente a ejecutar. A lo más, el demonio podrá ejecutar la misma componente "una vuelta y media", luego la condición del **await** será falsa y la única posibilidad para seguir ejecutando será la otra componente.

Si revisamos el código, vemos que el paralelismo es posible todavía. Con la inclusión de estos nuevos constructores estamos solamente sincronizando las componentes para acotar la diferencia de la cantidad de veces que cada una ha completado un ciclo.

No obstante, este operador genera un nuevo problema en nuestros programas paralelos. Veamos la siguiente historia de ejecución

- Ambas componentes están por ejecutar sus **await**. Como el valor de $turn$ es 1, $S_1$ puede proseguir, y ahora tenemos $turn = 2$.
- Progresa $S_2$, y esto es válido porque la condición de la instrucción de sincronización es verdadera, con lo que la asignación se produce y tenemos $turn = 1$.
- Continúa $S_2$ hasta llegar a la instrucción de espera.
- Pasa el control a $S_1$, que encuentra el valor verdadero, y termina la ejecución del ciclo.
- Solo la segunda componente está activa, pero el demonio no tiene componentes para elegir, pues una terminó y la otra espera una condición que no se satisface.

Está condición de $S_1$ finalizado y $S_2$ esperando que se satisfaga $turn = 2$ es un estado estable, por lo que el bloqueo será infinito y nuestro programa jamás terminará. La situación donde existen componentes activas pero ninguna puede progresar se conoce como *estado de deadlock*.

No podemos dejar de notar que los problemas se vuelven cada vez más sutiles, y las historias de ejecución que nos llevan a ellos resultan cada vez más complicadas.

A fin de evitar que la finalización de una componente impida el progreso de la otra, incluimos una asignación correspondiente a *turn* al final de cada componente.

**Solución 5**

```
S1 ≡ x:=0;
     do ¬found → await turn=1 → turn:=2;
                 if f.x → found:=true
                    ¬f.x → skip
                 fi
                 x:=x+1;
     od;
     turn:=2

S2 ≡ y:=0;
     do ¬found → await turn=2 → turn:=1;
                 if f.y → found:=true
                    ¬f.y → skip
                 fi
                 y:=y+1;
     od;
     turn:=1

FIND5 ≡ turn:=1; found:=false; [S1 || S2]
```

Se podría decir que prácticamente tenemos un programa paralelo que satisface nuestros requerimientos. Sin embargo apuntaremos un detalle que permite mejorar sutilmente el paralelismo del algoritmo [1].

Cada vez que se ejecuta alguna de las instrucciones de sincronización, la otra componente no puede progresar, pues la evaluación de la guarda junto a su asignación son un bloque atómico. Si se pudiera demostrar que aunque se interrumpa la ejecución del **await** entre la evaluación de la guarda y la asignación, el efecto es el mismo sea cual fuere la o las instrucciones de la otra componente que se entremezclen, entonces podríamos dar un código totalmente equivalente, con la sentencia de sincronización desacoplada, que presenta un *mayor grado de paralelismo*, pues su *grado de atomicidad* es menor. Realicemos este desacople y generemos la siguiente solución.

[1] Notar que el paralelismo fue anteriormente limitado al introducir la restricción de la "vuelta y media".

**Solución 6**

```
S1 ≡ x:=0;
     do ¬found → await turn=1;
                 turn:=2;
                 if f.x → found:=true
                    ¬f.x → skip
                 fi
                 x:=x+1;
     od;
     turn:=2

S2 ≡ y:=0;
     do ¬found → await turn=2;
                 turn:=1;
                 if f.y → found:=true
                    ¬f.y → skip
                 fi
                 y:=y+1;
     od;
     turn:=1

FIND6 ≡ turn:=1; found:=false; [S1 || S2]
```

Veamos ahora que en todas las posibilidades de interrupción por parte de $S_2$ entre el fin de la ejecución del **await** y el inicio de la ejecución de la asignación de *turn*, el valor de esta variable no cambia.

Pero esto es absolutamente trivial, porque todas las asignaciones a *turn* por parte de $S_2$ simplemente reafirman lo que queremos mantener, es decir $turn = 1$. Para ponerlo en otras palabras, entre la sincronización y la asignación el hecho $turn = 1$ no es *interferido*, ya sea porque las instrucciones no modifican directa o indirectamente las variables libres de esta aserción o bien porque aunque modifiquen la variable en juego, la validez de la fórmula no cambia.

El otro caso es simétrico, con lo que nuestro programa FIND6 parece ser correcto.

Hemos realizado un breve recorrido por los problemas típicos a los que nos enfrentamos cuando tratamos de atacar la creación de un algoritmo paralelo utilizando como armas la intuición y el razonamiento operacional.

Surge claramente la necesidad de una herramienta más poderosa.

En el siguiente capítulo introduciremos la teoría de Owicki-Gries, para luego en el tercer capítulo ver cómo ésta se aplica al desarrollo sistemático de programas siguiendo los lineamientos de la escuela de Eindhoven. Uno de los ejemplos a derivar con estas técnicas será este mismo algoritmo, por lo que podremos contrastar directamente ambos acercamientos, el informal y el formal.

# Capítulo 2

# Corrección de Programas Paralelos

## 2.1 Introducción

Como vimos en la sección 1.2, razonar de manera informal acerca de un programa paralelo es habitualmente una tarea ardua y propensa a errores. Iremos acercándonos gradualmente a un método asercional de verificación de programas paralelos con variables compartidas y sincronización, presentado por Susan Owicki y David Gries como una extensión a la lógica de Hoare para programas secuenciales.

Las características propias de estos programas paralelos serán introducidas una a una dando en cada caso un sistema deductivo que nos permita demostrar los programas decorados con aserciones.

## 2.2 Programas Secuenciales no Determinísticos

La base de nuestros programas paralelos descansa sobre los programas secuenciales no determinísticos, por lo que presentaremos un breve resumen.

Hemos elegido los generados por los comandos guardados de Dijkstra, cuya sintaxis está dada por:

| | | |
|---|---|---|
| $S ::=$ **skip** | | comando vacío |
| \| `v:=E` | | asignación |
| \| $S_1$; $S_2$ | | constructor de composición secuencial |
| \| **if** | $B_1 \to S_1$ | constructor de selección |
| | $\Box B_2 \to S_2$ | |
| | $\vdots$ | |
| | $\Box B_n \to S_n$ **fi** | |
| \| **do** | $B_1 \to S_1$ | constructor de repetición |
| | $\Box B_2 \to S_2$ | |
| | $\vdots$ | |
| | $\Box B_n \to S_n$ **od** | |

donde $B_i$ es una expresión booleana y $E$ es una expresión booleana o entera. Suponemos además que el lenguaje tiene operadores para generar todas las expresiones enteras y booleanas habituales.

Aunque el modelo operacional resulta ampliamente conocido, hagamos un rápido paseo por la semántica informal de estos comandos y constructores.

El comando **skip** es el que no opera sobre el estado, su único efecto es el de terminar siempre. Su contraparte está en la asignación, pues modifica el estado igualando la variable indicada en el lado izquierdo con el resultado de evaluar la expresión del lado derecho en el estado actual. Una extensión de ésta última es la multiasignación, que opera sobre un conjunto de variables de manera simultánea.

El constructor más sencillo es el de composición secuencial. Este sólo compone los efectos de las sentencias $S_1$ y $S_2$.

La selección introduce el no-determinismo y su semántica es que dado el estado inicial, se ejecutará la sentencia de alguna de las expresiones booleanas que evalúen a verdadero. Si todas estas guardas resultan falsas entonces el

constructor aborta junto con todo el programa que lo contenía.

Finalmente, la repetición también es no-determinista y comienza su ejecución como el constructor alternativo evaluando las guardas en el estado actual. Si todas son falsas la repetición termina, en cambio si alguna resulta verdadera, ejecuta una de las sentencias asociadas y vuelve a evaluar las guardas en el nuevo estado.

En principio el no-determinismo introducido por la selección y la repetición puede resultar molesto dado que se aleja de la implementación, pero en contrapartida permite escribir programas más claros, estructurados y no tan sobre-especificados, además permite realizar derivaciones de programas de manera más modular. Siempre tendremos en mente que todos los programas aquí presentados necesitarán todavía algún tipo de transformación antes de ser implementados en lenguajes y modelos de ejecución concretos.

Si quisiéramos que los programas fuesen determinísticos, usaríamos los constructores de selección alternativa y de repetición a la manera del trabajo original de Hoare.

$S$::=**skip**
$\vdots$
| **if** $B$ **then** $S_1$
**else** $S_2$ **end**
| **while** $B$ **do** $S$ **end**

La semántica axiomática seguida es la de las triplas de Hoare $\{P\}S\{Q\}$ para demostrar *corrección parcial* donde $P$ y $Q$ son fórmulas de la lógica que relacionan las variables y se denominan *aserciones*, mientras que $S$ es una sentencia del lenguaje expresado anteriormente.

Los axiomas y reglas de inferencia son:

$$\frac{}{\{P\}\mathbf{skip}\{P\}} \quad \textit{skip}$$

$$\frac{}{\{P(x:=E)\}\mathtt{x:=E}\{P\}} \quad \textit{asignación}$$

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

$$\frac{\{P\wedge B_1\}S_1\{Q\},\cdots,\{P\wedge B_n\}S_n\{Q\}}{\{P\}\mathbf{if}\,B_1\to S_1\,\square\cdots\square\,B_n\to S_n\,\mathbf{fi}\{Q\}} \quad \textit{selección}$$

$$\frac{\{P\wedge B_1\}S_1\{P\},\cdots,\{P\wedge B_n\}S_n\{P\}}{\{P\}\mathbf{do}\,B_1\to S_1\,\square\cdots\square\,B_n\to S_n\,\mathbf{od}\{P\wedge\neg(B_1\wedge\cdots\wedge B_n)\}} \quad \textit{repetición}$$

$$\frac{P\to P_1,\{P_1\}S\{Q_1\},Q_1\to Q}{\{P\}S\{Q\}} \quad \textit{consecuencia}$$

donde $P(x := E)$ es el operador de substitución aplicado a $P$ y $P \to Q$ indica que a partir de $P$ podemos probar $Q$ dentro de un sistema deductivo que no está explicitado, pero que nos permite deducir relaciones aritméticas y booleanas básicas.

Diremos que $\{P\}S\{Q\}$ es teorema de este sistema de inferencia cuando a partir de los axiomas y reglas de deducción podamos concluir justamente $\{P\}S\{Q\}$

Sin embargo esta prueba normalmente resulta engorrosa y la estructura del programa se pierde en ella. Las pruebas son más comprensibles si se realiza una *proof outline* donde las aserciones se intercalan con el texto del programa [Owi75, Cou90]. Usando este tipo de pruebas podemos hablar de las funciones *pre* y *post*, que van de sentencias a aserciones, donde la primera nos da la aserción previa más cercana mientras que la otra nos devuelve la posterior más cercana a una sentencia dada.

Veamos a continuación el sistema deductivo para estos programas decorados con aserciones, donde $S^*$ es un programa $S$ anotado con aserciones.

$$\frac{}{\{P\}\mathbf{skip}\{P\}}$$

$$\frac{}{\{P(x:=E)\}\mathtt{x:=E}\{P\}}$$

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

$$\frac{\{P\wedge B_1\}S_1^*\{Q\},\cdots,\{P\wedge B_n\}S_n^*\{Q\}}{\{P\}\mathbf{if}\,B_1\to\{P\wedge B_1\}S_1^*\{Q\}\,\square\cdots\square\,B_n\to\{P\wedge B_n\}S_n^*\{Q\}\,\mathbf{fi}\{Q\}}$$

$$\frac{\{P\wedge B_1\}S_1^*\{P\},\cdots,\{P\wedge B_n\}S_n^*\{P\}}{\{inv{:}P\}\mathbf{do}\,B_1\to\{P\wedge B_1\}S_1^*\{P\}\,\square\cdots\square\,B_n\to\{P\wedge B_n\}S_n^*\{P\}\,\mathbf{od}\{P\wedge\neg(B_1\wedge\cdots\wedge B_n)\}}$$

$$\frac{P\to P_1,\{P_1\}S^*\{Q_1\},Q_1\to Q}{\{P\}\{P_1\}S^*\{Q_1\}\{Q\}}$$

$$\frac{\{P\}S^*\{Q\}}{\{P\}S^{**}\{Q\}}$$

donde en la última regla $S^{**}$ sale de $S^*$ borrando algunas anotaciones que no sean de la forma $\{inv : P\}$.

Es posible demostrar la equivalencia entre los dos sistemas deductivos, por lo que de ahora en adelante trabajaremos con este último que en general resulta más conveniente y además es uno de los pilares sobre los que descansa la prueba de no interferencia, piedra fundamental de la teoría de Owicki-Gries. En secciones siguientes veremos esto en más detalle.

Otra posibilidad de presentar estas reglas es como lo hace Dijkstra utilizando equivalencias y cuantificaciones universales. Éstas también resultan totalmente equivalentes a las anteriores formulaciones, pero hacen que las demostraciones sean más claras en su presentación.

El conjunto de equivalencias es:

$$\{P\}\mathbf{skip}\{Q\} \equiv [P \Rightarrow Q]$$
$$\{P\}\mathtt{x:=E}\{Q\} \equiv [P \Rightarrow Q(x := E)]$$
$$\{P\}S_0\,;S_1\{Q\} \equiv \exists H :: \{P\}S_0\{H\} \wedge \{H\}S_1\{Q\}$$
$$\{P\}\mathbf{if}\ \square_i B_i \to S_i\ \mathbf{fi}\{Q\} \equiv \bigwedge_i \{P \wedge B_i\}S_i\{Q\}$$
$$\{P\}\mathbf{do}\ \square_i B_i \to S_i\ \mathbf{od}\{Q\} \equiv$$
$$\quad \exists H :: [P \Rightarrow H]\ \wedge\ \bigwedge_i(\{H \wedge B_i\}S_i\{H\})\ \wedge\ [H \wedge (\bigwedge_i \neg B_i) \Rightarrow Q]$$

Con la notación $[Q]$ estamos indicando cuantificación universal sobre todas las variables de estado del predicado $Q$.

Vemos que no hay nada parecido a la regla de la consecuencia, pues ella se encuentra distribuída entre todas las reglas, y esto también ayuda a que las pruebas sean menos tediosas.

Finalmente, un programa anotado será correcto respecto a estas equivalencias, si logramos demostrar que la fórmula equivalente a este resulta verdadera.

Existe aún otra formulación equivalente a las anteriores: los *transformadores de predicados de Dijkstra.*

Se define la función $wlp$[1] (*weakest liberal precondition*) que dada una sentencia y un predicado, devuelve la precondición más débil necesaria para que luego de ejecutar la sentecia la postcondición se haga verdadera.

La relación con las ternas de Hoare es la siguiente:

$$\{P\}S\{Q\} \equiv [P \Rightarrow wlp.S.Q]$$

y se la define asi:

$$[wlp.\mathbf{skip}.P \equiv P]$$
$$[wlp.(\texttt{x:=E}).P \equiv P(x := E)]$$
$$[wlp.(S_0\,;S_1).P \equiv wlp.S.(wlp.T.P)]$$
$$[wlp.(\mathbf{if}\ \Box_i B_i \rightarrow S_i\ \mathbf{fi}).P \equiv \bigwedge_i B_i \Rightarrow wlp.S_i.P]$$
$$[wlp.DO.P \equiv$$
$$(\bigwedge_i \neg B_i) \Rightarrow P\ \wedge\ \bigwedge_i (B_i \Rightarrow wlp.S.(wlp.DO.P))]$$

donde $DO$ : **do** $\Box_i B_i \rightarrow S_i$ **od** y para calcular efectivamente la $wlp$ de una repetición deberemos solucionar una ecuación recursiva.

La principal característica de estos sistemas es su *consistencia* con el modelo operacional. Obviamente para demostrar esta consistencia se requiere ser más formales respecto a la presentación del sistema deductivo y tener un modelo operacional concreto capaz de ser descripto también a través de métodos formales.

Podemos ver desarrollos completamente formales de este tema en diversos textos como [AO91, Cou90].

A los fines prácticos, esta consistencia se traduce en una interpretación de lo que significan las triplas $\{P\}S\{Q\}$ o las aserciones dentro de una *proof outline.* El siguiente teorema muestra esta consistencia.

**Teorema 2.1 (Consistencia fuerte)** *Sea $S$ una sentencia del programa $T$ y $pre(S)$ la precondición de ésta en la prueba $\{P\}T\{Q\}$. Entonces si el programa $T$ comienza su ejecución en un estado que satisface $P$, y llega al punto en que su próximo paso es ejecutar $S$ y el estado está representado por $m$, entonces se da $pre(S)(m) \equiv true$*

Ahora podemos decir que nuestro programa salta de aserción en aserción, con el estado global del sistema que las va cumpliendo a medida que se pasa por ellas, sin importar cómo o qué historia de ejecución hizo que llegara hasta allí. En este punto vemos claramente que las semánticas axiomáticas resultan más abstractas que las operacionales.

Esta definición será crucial para comprender de manera informal el test de no interferencia que propone la teoría que nos permite demostrar programas paralelos decorados con aserciones.

[1] Contrastar con la definición para corrección total $wp$ dada en los textos clásicos [Dij76, Gri81]

## 2.3 Programas Paralelos Disjuntos

Estudiaremos primero la forma más sencilla de paralelismo dada por lo que se conoce como *programas paralelos disjuntos.*

El paralelismo se introduce a través del *operador de composición paralela* cuya sintaxis es:

$S$::=**skip**
$\vdots$
$\mid [S_1 \parallel \cdots \parallel S_n]$ constructor de composición paralela

donde $n \geq 2$ y $S_i$ son sentencias que no contienen este operador (es decir programas secuenciales), con lo que evitamos el paralelismo anidado[2].

La semántica informal de este constructor dice que las *componentes secuenciales* comienzan a ejecutarse en paralelo y cuando todas estas terminan, el operador de composición paralela finaliza. Nada se asume acerca de las velocidades relativas de cada uno de los procesos secuenciales.

**Definición 2.1 (Programa Paralelo Disjunto)** *Un programa paralelo será* disjunto *cuando las componentes secuenciales $S_i$ de cada uno de los operadores de composición paralela cumplan con:*

$$\forall i, j : i \neq j : change(S_i) \cap var(S_j) = \emptyset$$

*es decir que las variables accedidas en una componente no pueden ser modificadas por otra.*

Donde *change* es una función que dada una sentencia nos da el conjunto de identificadores de variables que aparecen en el lado izquierdo de las asignaciones, y *var* es todo el conjunto de identificadores de variables que aparecen en una sentencia.

El siguiente es un ejemplo de un programa paralelo disjunto.

$$S \equiv [\texttt{x:=1; x:=x+1} \parallel \texttt{y:=1; y:=y+2}]$$

Como la semántica informal dada resulta demasiado vaga utilizaremos como herramienta para analizar y hacer demostraciones sobre el operador de composición paralela el denominado *modelo de interleaving.*

En éste, la computación se da *serializando* la ejecución de las componentes secuenciales mediante el *entrelazado* de sus ejecuciones, por lo que nunca dos o más instrucciones se ejecutan de manera simultánea.

El interleaving es el más utilizado para modelar ejecución paralela. Más adelante mostraremos como este modelo predice el comportamiento de las ejecuciones paralelas reales si se dan ciertas condiciones en el hardware subyacente.

[2] Aunque la mayoría de las formalizaciones permite el paralelismo anidado, preferimos esta definición más restrictiva que sigue la noción de *multiprograma* que será sobre lo que se trabajará en el capítulo de derivación de multiprogramas.

Para el ejemplo, la cantidad de serializaciones posibles es igual a

$$\frac{(2+2)!}{2!\ 2!} = 6$$

Es decir que tenemos seis *escenarios* posibles de ejecución para nuestro programa dentro del modelo.

Si generalizamos un poco, veremos que la cantidad de posibles escenarios de ejecución para un programa paralelo disjunto de dos componentes donde la cantidad de instrucciones de cada una está dada por $n_1$ y $n_2$ es

$$\frac{(n_1+n_2)!}{n_1!\ n_2!} = \binom{n_1+n_2}{n_1}$$

Esta cantidad crece exponencialmente con $n_1$ y $n_2$ y aún más rápido respecto a la cantidad de componentes consideradas.

Si tuviéramos que verificar cada uno de estos escenarios de ejecución para demostrar que nuestro texto de programa satisface ciertos requerimientos, entonces estamos condenados al fracaso.

No obstante, notamos que en todos los posibles casos de entrelazado, nuestro programa de ejemplo arriba al mismo resultado. Esta situación no resulta casual, y es posible demostrar que todas las secuencias de ejecución de un programa paralelo disjunto determinístico llegan al mismo estado [AO91, Lema 4.10].

Pero la propiedad general que estamos buscando es que al trabajar con programas paralelos disjuntos, no importa como se entremezclen las sentencias (determinísticas o no) de las componentes secuenciales, el efecto será el mismo, pues cada componente actúa sobre un espacio de estados disjunto a las otras.

Gracias a esta propiedad, cualquier forma de entremezclar las instrucciones resulta equivalente. En particular la ejecución secuencial de una componente tras otra también tendrá esta propiedad. Este hecho se ve reflejado en la siguiente regla[3]:

$$\frac{\{P\}S_1\,;\cdots;\,S_n\{Q\}}{\{P\}[S_1\|\cdots\|S_n]\{Q\}} \quad \textit{serialización}$$

Una consecuencia más o menos directa de esta última regla nos permite razonar composicionalmente sobre el comportamiento entrada/salida de los programas paralelos disjuntos. Dadas las pre y post condiciones de los programas secuenciales podemos deducir la pre y post condición de sus composición paralela.

$$\frac{\{P_1\}S_1\{Q_1\},\cdots,\{P_n\}S_n\{Q_n\}}{\{\bigwedge_{i=1}^{n} P_i\}[S_1\|\cdots\|S_n]\{\bigwedge_{i=1}^{n} Q_i\}} \quad \textit{paralelismo disjunto}$$

con $free(P_i, Q_i) \cap change(S_j) = \emptyset\ \forall i \neq j$

Donde $free$ es una función que devuelve las variables no ligadas de un predicado y la condición de disjunción de variables que se pide para las aserciones es necesaria para mantener la consistencia de la regla.

Ahora para tener una prueba de

$$\{true\}[\texttt{x:=1;}\ \texttt{x:=x+1} \parallel \texttt{y:=2;}\ \texttt{y:=y+2}]\{x = 2 \wedge y = 4\}$$

[3]Esta regla no está demostrada en los textos que fueron consultados [AO91, Cou90] para el caso no-determinístico que estamos planteando, sin embargo una demostración no resulta complicada si recurrimos a las ideas planteadas en los ejercicios 4.5 y 4.8 de [AO91]

sólo necesitamos ver que

$$\{true\}\texttt{x:=1; x:=x+1}\{x = 2\}$$
$$\{true\}\texttt{y:=2; y:=y+2}\{y = 4\}$$

y estos son teoremas fáciles de probar en el sistema deductivo para programas secuenciales.

## 2.4 Programas Paralelos con Variables Compartidas

En la sección anterior vimos que en los programas paralelos disjuntos las demostraciones de corrección parcial se reducían a demostraciones de corrección parcial sobre sus componentes. Éstos eran disjuntos, cada uno ejecutaba sobre un espacio de estados independientes de los otros y la única conexión que existía entre ellos era el inicio y final simultáneo.

Sin embargo esta clase de programas compensa su sencillez en las demostraciones con su falta de utilidad. Justamente, el objetivo de la programación concurrente es que los procesos se comuniquen y sincronicen para realizar una tarea y en particular para los programas paralelos la comunicación y sincronización se da por medio de variables compartidas.

Al introducir la posibilidad de que más de una componente pueda modificar el estado, se pone de manifiesto la importancia del no-determinismo en la forma en que se ejecutarán las componentes del programa paralelo. El comportamiento de entrada/salida depende de manera directa de este no-determinismo.

En el modelo de interleaving para ejecución paralela, el no-determinismo es usualmente asociado a un *demonio*, y a éste lo podemos pensar como el encargado de decidir en cada paso cuál de los *hilos de control* de cada una de las componentes podrá avanzar. Este será el *scheduler* de las componentes para ponerlo en términos de sistemas operativos. En secciones posteriores veremos que es necesario especificar el no-determinismo pues este afecta de manera directa propiedades de nuestros multiprogramas.

Veamos un ejemplo donde se pone de manifiesto la relación entre el comportamiento entrada/salida del programa y la forma en que el demonio elige. El texto de programa es el siguiente

$$[\texttt{x:=0} \parallel \texttt{x:=x+2}]$$

Dependiendo que asignación se ejecute primero, el resultado podrá ser 0 o 2, con lo que la propiedad de composicionalidad de entrada/salida que mostraban los programas paralelos disjuntos ya no es válida. En todo caso una anotación correcta del programa sería

$$\{true\}[\texttt{x:=0} \parallel \texttt{x:=x+2}]\{x = 0 \vee x = 2\}$$

donde la postcondición se ha debilitado para contemplar los dos posibles escenarios de ejecución.

Otro factor a tener en cuenta es el *grado de atomicidad* de los comandos. En el ejemplo anterior supusimos que ambas instrucciones eran ejecutadas sin interrupción por parte del demonio. Sin embargo es perfectamente posible que la asignación `x:=x+2` pueda ser interrumpida; no nos olvidemos que en una máquina concreta el programa anterior puede ser traducido para su ejecución en uno equivalente a

$$[\texttt{x:=0} \parallel \texttt{a:=x+2; x:=a}]$$

debido a que, por ejemplo, la unidad de ejecución no es capaz de realizar incrementos directamente en la memoria.

Los tres interleavings o escenarios posibles arrojan tres resultados distintos, por lo que la anotación correcta ahora sería

$$\{x = X\}[\texttt{x:=0} \parallel \texttt{a:=x+2; x:=a}]\{x = X + 2 \vee x = 0 \vee x = 2\}$$

donde la precondición simplemente le asocia a $x$ un valor arbitrario antes de empezar la ejecución.

El nuevo resultado surge cuando el demonio elige entrelazar las componentes de forma tal que escrito de manera serial sería

```
a:=x+2; x:=0; x:=a
```

Toda esta problemática se resume en el siguiente párrafo de [AO91]:

> "Cualquier intento de entender paralelismo con variables compartidas comienza con la comprensión del acceso a las variables compartidas y cualquier explicación involucra la noción de *acción atómica*"

En el programa anterior vemos nuevamente el fenómeno de la no-composicionalidad. Los programas secuenciales `x:=x+2` y `a:=x+2; x:=a` son equivalentes pero los dos programas paralelos que se generan no presentan el mismo comportamiento entrada/salida.

Resumiendo, los programas paralelos con variables compartidas no tienen un comportamiento composicional, por lo que no existe una regla bajo condiciones triviales que a partir de especificaciones entrada/salida de cada componente pueda determinar la especificación entrada/salida de su composición paralela.

### 2.4.1 No Interferencia y la Regla del Paralelismo

Todo el planteamiento anterior nos lleva a buscar condiciones que sumadas a la corrección de cada componente nos permita deducir el comportamiento entrada/salida de la composición paralela.

La solución está en el *test de no interferencia* propuesto en el trabajo de Owicki-Gries, donde además de la corrección de cada componente secuencial se pide que por cada aserción $P$ de la *proof outline* de una componente y por cada sentencia $S$ con preaserción $pre(S)$ de una componente distinta, la siguiente tripla sea válida[4].

[4]En [Dij82] podemos ver una motivación para la introducción de estas triplas.

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

En términos operacionales, esta tripla puede ser interpretada de la siguiente manera: si iniciamos desde un estado que satisface $P$ y $pre(S)$, luego de ejecutar $S$ el estado de salida de la sentencia seguirá cumpliendo con $P$. Es decir, el ejecutar la sentencia $S$ de otra componente no cambia o mejor dicho *no interfiere* con el valor de verdad de $P$.

Supongamos ahora dos programas secuenciales anotados, donde además de haber demostrado la corrección de las triplas de cada uno hemos demostrado la invariancia o no interferencia de cada una de las aserciones de uno en el otro, es decir si tenemos los programas con sus pruebas de corrección

$$\{P_i\}R_i\{P_{i+1}\}$$
$$\{Q_i\}S_i\{Q_{i+1}\}$$

y además las pruebas de invariancia

$$\{P_i \wedge Q_j\}R_i\{Q_j\}$$
$$\{Q_i \wedge P_j\}S_i\{P_j\}$$

entonces sin importar cómo se ejecuten entrelazadamente un programa y el otro, las aserciones de cada uno de ellos seguirán siendo válidas.

Pidiendo este *test de no interferencia* entre las componentes hemos vuelto a lo que sucedía con los programas paralelos disjuntos. Tenemos la condición necesaria para concluir que el comportamiento entrada/salida del programa paralelo es la composición de los comportamientos entrada/salida de sus componentes siempre y cuando las triplas generadas por el test sean demostrables.

La regla es la siguiente

$$\frac{\{P_1\}S_1\{Q_1\},\cdots,\{P_n\}S_n\{Q_n\} \text{ y no interferencia}}{\{\bigwedge_{i=1}^{n} P_i\}[S_1 \| \cdots \| S_n]\{\bigwedge_{i=1}^{n} Q_i\}} \quad paralelismo$$

Vemos que el antecedente pide dos condiciones: la corrección de cada una de las componentes y la no interferencia. Normalmente a la primera se la conoce como *corrección local* y a la segunda como *corrección global*.

Con el test de no interferencia estamos logrando que no importe que el demonio cambie a otro hilo de control, pues si estaba en el programa $P_1$ a punto de ejecutar $S_i$ entonces por corrección local el estado cumple con $pre(S_i)$ y por más de que se cambie el hilo de control esta aserción seguirá siendo válida gracias a la corrección global. Cuando eventualmente el hilo de control regrese, su precondición seguirá siendo válida y la relación $\{pre(S_i)\}S_i\{post(S_i)\}$ se cumplirá.

Queda claro que este test contempla la interrupción entre las sentencias $S_i$ no en medio de ellas. Es decir, la regla condiciona que la sentencia, posiblemente compuesta, sea ejecutada atómicamente entre aserción y aserción. A los fines de Owicki-Gries, la atomicidad está dada por la *proof outline*.

No podemos dejar de notar la enorme cantidad de pruebas que involucra el test de no interferencia, sin embargo en la mayoría de los casos éstas resultan

triviales, y en secciones siguientes veremos técnicas para simplificar la tarea de prueba. Además respecto a un razonamiento operacional puro, donde la cantidad de "condiciones" (escenarios) a testear resulta exponencial, la regla sólo nos obliga a verificar una cantidad polinómica de "condiciones" (triplas)

A manera de ejemplo demostraremos la corrección del siguiente programa paralelo, donde notamos que las aserciones de cada componente son más débiles que lo que podría esperarse analizando cada componente por separado.

$$\begin{array}{l}\{P : x = 0\}\ [\\ \qquad \{P_1 : x = 0 \vee x = 1\}\texttt{x:=x+2}\{Q_1 : x = 2 \vee x = 3\} \parallel \\ \qquad \{P_2 : x = 0 \vee x = 2\}\texttt{x:=x+1}\{Q_2 : x = 1 \vee x = 3\}\\ ]\ \{Q : x = 3\}\end{array}$$

Primero probaremos que cada una de las componentes está correctamente anotada, y para esto tenemos que ver que las triplas

$$\begin{array}{c}\{x = 0 \vee x = 1\}\texttt{x:=x+2}\{x = 2 \vee x = 3\}\\ \{x = 0 \vee x = 2\}\texttt{x:=x+1}\{x = 1 \vee x = 3\}\end{array}$$

son teorema, pero hay muy poco para demostrar pues éstas son consecuencia directa del axioma de asignación.

Resta ver que las triplas generadas por el test de no interferencia también son demostrables.

La invariancia de $P_1$ y $Q_1$ respecto a la asignación `x:=x+1` está dada por

$$\begin{array}{c}\{P_1 \wedge P_2\}\texttt{x:=x+1}\{P_1\}\\ \{Q_1 \wedge P_2\}\texttt{x:=x+1}\{Q_1\}\end{array}$$

mientras que la corrección global de $P_2$ y $Q_2$ se prueba con las triplas

$$\begin{array}{c}\{P_2 \wedge P_1\}\texttt{x:=x+2}\{P_2\}\\ \{Q_2 \wedge P_1\}\texttt{x:=x+2}\{Q_2\}\end{array}$$

Veamos que el primero es teorema utilizando el sistema deductivo propuesto por Hoare

$$\frac{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2) \rightarrow x = 0,\ \frac{}{\{x=0\}\texttt{x:=x+1}\{x=1\}},\ x = 1 \rightarrow x = 0 \vee x = 1}{\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\}\texttt{x:=x+1}\{x = 0 \vee x = 1\}}$$

Usemos ahora las equivalencias de Dijkstra para mostrar la prueba de invariancia de $Q_1$

$$\begin{array}{l}\{(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2)\}\texttt{x:=x+1}\{x = 2 \vee x = 3\} \equiv\\ \quad [(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2) \Rightarrow (x + 1 = 2 \vee x + 1 = 3)] \equiv\\ \quad [x = 2 \Rightarrow (x = 1 \vee x = 2)]\end{array}$$

y esta última es verdadera, con lo que la terna resulta válida.

Las dos demostraciones que faltan son tan sencillas como éstas y no las incluiremos.

Finalmente como $P_1 \wedge P_2 \equiv x = 0$ y $Q_1 \wedge Q_2 \equiv x = 3$, todas las condiciones de aplicabilidad de la regla del paralelismo están dadas, por lo que tenemos nuestra primera demostración de un programa paralelo con variables compartidas. El único requerimiento que debemos imponer a un modelo de ejecución para que la demostración se refleje en él, es que las asignaciones sean atómicas.

### 2.4.2 Las Variables Auxiliares son Necesarias

Tomemos el siguiente programa paralelo, el que resulta de una pequeñísima variación del que hemos demostrado correcto.

$$[\texttt{x:=x+1} \parallel \texttt{x:=x+1}]$$

Claramente en un modelo donde los incrementos se supongan atómicos, si el estado inicial cumple con $x = 0$ entonces el estado final hará verdadero $x = 2$. Anotemos este programa de una manera similar al anterior para luego ver si las triplas que pide como antecedente la regla de paralelismo pueden ser probadas.

$$\begin{array}{l} \{P : x = 0\}\ [ \\ \qquad \{P_1 : x = 0 \vee x = 1\}\texttt{x:=x+1}\{Q_1 : x = 1 \vee x = 2\} \parallel \\ \qquad \{P_2 : x = 0 \vee x = 1\}\texttt{x:=x+1}\{Q_2 : x = 1 \vee x = 2\} \\ ]\ \{Q : x = 2\} \end{array}$$

Como las componentes son totalmente simétricas, con ver la mitad de las pruebas alcanza. La corrección local de $\{P_1\}\texttt{x:=x+1}\{Q_1\}$ sale de manera directa con el axioma de asignación y la regla de la consecuencia.

Para demostrar la corrección global de $P_1$ respecto a `x:=x+1` tenemos que ver que la tripla $\{P_1 \wedge P_2\}\texttt{x:=x+1}\{P_1\}$ se deduzca de la teoría, es decir

$$\{x = 0 \vee x = 1\}\texttt{x:=x+1}\{x = 0 \vee x = 1\}$$

pero esta tripla no es teorema pues no resulta válida en el modelo y nuestra teoría es consistente. Además, la conjunción de las postcondiciones de las dos componentes no implica la postcondición global que buscamos.

Esta anotación particular no nos sirve para demostrar la corrección, ¿Pero habrá alguna? La respuesta es negativa y llegaremos a un absurdo suponiendo que tenemos predicados $P_1$, $P_2$, $Q_1$ y $Q_2$ que sí hacen que el siguiente sea teorema

$$\{P : x = 0\}\ [\{P_1\}\texttt{x:=x+1}\{Q_1\} \parallel \{P_2\}\texttt{x:=x+1}\{Q_2\}]\ \{Q : x = 2\}$$

Las condiciones que llevaron a esta tripla son

1. corrección local
   (a) $\{P_1\}\texttt{x:=x+1}\{Q_1\}$
   (b) $\{P_2\}\texttt{x:=x+1}\{Q_2\}$

2. no interferencia

   (a) $\{P_1 \wedge P_2\}$`x:=x+1`$\{P_1\}$

   (b) $\{P_2 \wedge P_1\}$`x:=x+1`$\{P_2\}$

   (c) $\{Q_1 \wedge P_2\}$`x:=x+1`$\{Q_1\}$

   (d) $\{Q_2 \wedge P_1\}$`x:=x+1`$\{Q_2\}$

3. regla de consecuencia

   (a) $x = 0 \Rightarrow P_1 \wedge P_2$

   (b) $Q_1 \wedge Q_2 \Rightarrow x = 2$

y supondremos además que $var(P_1) \cup var(P_2) \cup var(Q_1) \cup var(Q_2) = \{x\}$

Por (2a) y (2b) usando las equivalencias tenemos

$$[P_1 \wedge P_2 \Rightarrow (P_1 \wedge P_2)(x := x + 1)]$$

que junto con (3a) forman las condiciones necesarias para que por inducción podamos decir

$$\forall x : 0 \leq x : P_1 \wedge P_2 \tag{2.1}$$

Ahora por (1a) y (1b) tenemos que

$$[P_1 \wedge P_2 \Rightarrow (Q_1 \wedge Q_2)(x := x + 1)]$$

y como la precondición vale para cualquier $0 \leq x$ por (2.1), entonces la post-condición cumplirá con

$$\forall x : 1 \leq x : Q_1 \wedge Q_2 \tag{2.2}$$

luego, por (3b) y (2.2) tenemos

$$\forall x : 1 \leq x : x = 2$$

lo cual resulta absurdo.

Notamos que (2c) ni (2d) fueron utilizados, pero al demostrar que el conjunto de soluciones es vacío, restringir aún más este conjunto no puede dar soluciones.

La clave está en las *variables auxiliares.* Veamos su definición.

**Definición 2.2 (Variables Auxiliares)** *Sea $AV$ un conjunto de variables que aparecen en $S$ sólo en asignaciones* `x:=E`*, donde $x \in AV$. Entonces $AV$ se denomina conjunto de variables auxiliares.*

El nombre de este conjunto de variables surge de que ellas no participan en el flujo de control (no pueden estar en las guardas), y tampoco lo hacen en el flujo de datos pues sus valores no son utilizados para asignar variables fuera de $AV$.

Pero entonces si este conjunto de variables no participa en el flujo de datos ni influye en el flujo de control, la siguiente regla adquiere sentido

$$\frac{\{P\}S'\{Q\}}{\{P\}S\{Q\}} \quad variables\ auxiliares$$

donde $S'$ es un programa que tiene $AV$ como conjunto de variables auxiliares, $P$ y $Q$ son aserciones que no contienen variables libres de $AV$ y $S$ es el resultado de borrar todas las asignaciones a las variables del conjunto $AV$. La condición que $free(P) \cap AV = \emptyset$ puede ser relajada para evitar un paso mecánico donde lo primero que haríamos sería la asignación de las variables auxiliares con sus valores iniciales.

Nuestra teoría ahora es capaz de demostrar el programa que motivó la introducción de esta regla.

Si el siguiente programa anotado es teorema

$$\begin{array}{l} \{x = 0 \wedge p = 0 \wedge q = 0\} \; [ \\ \qquad\qquad \{p = 0 \wedge x = p + q\}\texttt{x,p:=x+1,1}\{p = 1 \wedge x = p + q\} \parallel \\ \qquad\qquad \{q = 0 \wedge x = p + q\}\texttt{x,q:=x+1,1}\{q = 1 \wedge x = p + q\} \\ ] \; \{x = 2 \wedge p = 1 \wedge q = 1\} \end{array} \tag{2.3}$$

entonces tomamos $\{p, q\}$ de todos los conjuntos posibles de variables auxiliares y tenemos por la nueva regla que

$$\{x = 0\}[\texttt{x:=x+1} \parallel \texttt{x:=x+1}]\{x = 2\}$$

resulta finalmente teorema de nuestra teoría.

Para demostrar (2.3) necesitamos hacer una prueba muy al estilo de la primera que realizamos, por lo que suponemos que el lector podrá completarla sin inconvenientes.

Aunque el uso de variables auxiliares o variables fantasma —como también se las conocen— se introdujo para los programas paralelos con variables compartidas, ellas también son necesarias para el caso de los programas paralelos disjuntos, donde por ejemplo

$$\{x = y\}[\texttt{x:=x+1} \parallel \texttt{y:=y+1}]\{x = y\}$$

es válido en el modelo pero no es demostrable en la teoría sin la regla para eliminación de variables auxiliares[5].

Probar el programa de los dos incrementos paralelos, requirió la introducción de dos variables nuevas. La demostración es clara, pero todavía se puede preguntar acerca del rol de las variables fantasma, y más aún, se puede preguntar si existe alguna forma sistemática de introducirlas.

Observando el programa aumentado con las variables $p$ y $q$, ellas están indicando el paso por los incrementos; $p$ indica si ya se efectuó el incremento en la primera componente y $q$ hace lo propio con la segunda. Con las variables auxiliares estamos marcando el flujo de control del programa.

La teoría de Lamport [Lam77] lleva al extremo esta idea y cada componente tiene un contador de programa que indica en qué lugar se encuentra el flujo de control del programa, con lo que existe una manera sistemática de introducir variables auxiliares.

Sin embargo el enfoque de Lamport es extremo en cuanto a que tenemos toda la información del flujo de control. Con las variables auxiliares podemos

[5] La demostración de este hecho, así como otro ejemplo de incompletitud de las reglas para programas paralelos con variables compartidas pueden ser encontrados en [AO91].

marcar parcialmente este flujo, y así evitar inundar nuestro programa con información irrelevante[6], además el uso medido de éstas parece ser conveniente para la derivación de multiprogramas utilizando Owicki-Gries.

### 2.4.3 El Modelo Subyacente

La regla de paralelismo será consistente con el modelo subyacente de interleaving siempre que éste respete la ininterrumpibilidad de las sentencias entre aserción y aserción. Pero surge la pregunta de cuán alejado se encuentra este modelo de la realidad, es decir del hardware que consta de varias unidades procesadoras con algún espacio de memoria compartida. Lamentablemente el modelo requerido dista del hardware convencional, donde lo único que podemos asegurar es el acceso ininterrumpido o atómico a una variable de tipo simple (una localidad de memoria), es decir, si dos o más procesadores ejecutando componentes de un programa paralelo tratan de acceder simultáneamente a una variable compartida, el hardware arbitra estos accesos serializándolos en algún orden.

En un programa como el que sigue

$$[\texttt{x:=1} \parallel \texttt{x:=2}]$$

los resultados posibles se reducen a $x = 1 \vee x = 2$.

Si en cambio suponemos que los accesos a memoria no tienen atomicidad, esto llevará a que este simple programa pueda devolver $x = 1 \vee x = 2 \vee x = 3$, y éste resulta el mínimo grado de atomicidad para cualquier computadora digital, es decir la referencia a bits. De más está decir que un hardware que sólo garantice ininterrumpibilidad en acciones tan finas en su granularidad, resulta poco conveniente tanto para los desarrolladores de hardware como para las personas que utilizan este hardware para ejecutar programas que satisfagan sus requerimientos.

La condición que nos acerca al modelo real es la siguiente.

**Definición 2.3 (Sentencia de 1-acceso)** *una sentencia de 1-acceso podrá referirse a una sola variable compartida y esta variable podrá ocurrir a lo sumo una vez dentro de ella.*

Si todas las sentencias entre aserción y aserción de nuestro programa paralelo son de 1-acceso, entonces habrá un paralelo total entre el modelo de interleaving que ejecuta atómicamente las sentencias entre aserción y aserción y el hardware subyacente que brinda accesos ininterrumpidos a memoria. No importa cómo se descompongan las sentencias para poder ser ejecutadas, ni si existe ejecución verdaderamente simultánea o si ésta resulta serializada en algún aspecto, lo fundamental es que cada acceso a memoria sea ininterrumpido para así evitar valores espúreos en la memoria.

Ninguno de los programas que hemos presentado hasta ahora tiene todas sus sentencias de 1-acceso y, aunque sepamos que el comportamiento entrada/salida en la ejecución real no será el dado por las aserciones, es completamente válido realizar pruebas utilizando una *atomicidad virtual*. No es necesario que estemos atrapados en el grado de atomicidad brindado por el hardware pues hay

[6]Lamport no estuvo de acuerdo con esta idea [Lam88] y en [Cou90, 8.9.2.5.2] podemos ver una reseña general del tema.

una relación directa entre grado de atomicidad y facilidad de demostrar que el multiprograma es correcto, resumiendo:

- a mayor grado de atomicidad más nos alejamos del hardware concreto pero más sencillas resultan las demostraciones.
- a menor grado de atomicidad más real resulta el programa pero en contrapartida la prueba se torna compleja.

## 2.5 Programas Paralelos con Variables Compartidas y Sincronización

En la sección anterior posibilitamos la comunicación entre componentes de un programa paralelo, en ésta completaremos las características básicas agregando un constructor que nos provea de algún mecanismo de sincronización.

La sincronización surge como la necesidad de limitar el grado de no-determinismo que posee un programa paralelo. El mecanismo propuesto será el de *región atómica condicional.*

Ahora tenemos una nueva cláusula para generar programas que se agrega a las ya dadas:

$S$::=**skip**
⋮
| **await** $B$ **then** $S_0$ constructor de sincronización

donde $B$ es una expresión booleana y $S_0$ una sentencia que no contiene **await**.

La semántica de este constructor dice que mientras $B$ sea falso, esta sentencia no es elegible para ser ejecutada y en caso de darse que $B$ evalúa a verdadero en el estado actual, la sentencia $S$ se ejecuta de manera atómica con la evaluación de la guarda.

Nuestro nuevo constructor *bloquea* la componente a la cual pertenece mientras la condición booleana no sea verdadera, y ésto introduce un nuevo problema cuando un subconjunto de componentes se encuentran bloqueadas en una condición que permanecerá invariantemente falsa, por más de que las otras *componentes activas* progresen y eventualmente terminen. Vimos un ejemplo de esto en la solución 4 de la sección 1.2.

Se denomina a este *estado de deadlock* y es deseable que los programas paralelos muestren la propiedad de *deadlock freedom.* En secciones siguientes veremos cómo atacar este problema y además describiremos posibles comportamientos del demonio en relación al constructor de sincronización.

Notamos además que este constructor carece de sentido si no se encuentra dentro de un contexto que involucre paralelismo, pues si tenemos sincronización en un programa secuencial y la condición es falsa cuando el flujo de control llega a un **await**, entonces ésta lo seguirá siendo infinitamente pues no hay otras componentes que puedan cambiar el estado y permitir el progreso de la computación.

La regla de corrección parcial para este constructor es

$$\frac{\{P \wedge B\} S \{Q\}}{\{P\}\textbf{await } B \textbf{ then } S\{Q\}} \quad \textit{sincronización}$$

Notamos que presenta la misma regla que el constructor de selección para el caso de una sola guarda, es por esto que suele sobrecargarse el **if**...**fi** asumiendo la semántica de ambos, es decir de selección y sincronización. Para esto debemos especificar que un constructor alternativo que no tenga alguna guarda que evalúe a verdadero no será elegible por el demonio, y la componente que la contiene (estamos de nuevo suponiendo que se encuentra en un contexto de paralelismo) permanecerá bloqueada.

El único punto ambiguo es que en el **await** suponemos ininterrumpible la evaluación de la condición o guarda junto con la ejecución de la sentencia asociada. En el constructor de selección ésto sólo se cumplirá si éste no contiene aserciones intermedias.

Para remediar esta situación podemos hacer explícito en nuestro programa el deseo de ejecutar un sentencia de manera atómica, mediante el constructor de *atomicidad virtual*

$S$::=**skip**
⋮
| <$S_0$> constructor de atomicidad virtual

con $S_0$ que no contiene constructores de atomicidad virtual.

Con este nuevo constructor podemos plantear las siguientes equivalencias

$$\textbf{<if } B \rightarrow\ S \textbf{ fi>} \equiv \textbf{await } B \textbf{ then } S$$
$$\textbf{<}S\textbf{>} \equiv \textbf{await } \texttt{True} \textbf{ then } S$$

Un caso básico del constructor de sincronización escrito como selección es de mucha utilidad y recibe el nombre de *skip guardado*

$$\textbf{<if } B \rightarrow\ \textbf{skip fi>} \equiv \textbf{await } B\ \equiv [B]$$

donde éste evita el progreso hasta que la condición dada en la guarda sea verdadera.

De todas las posibilidades usuales para lograr sincronización entre componentes, se eligió una que resulta primitiva, pero a la vez potente y flexible, a tal punto que las otras primitivas de sincronización como semáforos, regiones críticas y monitores pueden ser expresadas en función de nuestra región atómica condicional.

Lamentablemente no es posible lograr una implementación más eficiente de nuestro constructor que el *busy waiting*, y usualmente se trata de diseñar o transformar los programas para que estas regiones atómicas condicionales puedan ser implementadas utilizando mecanismos de sincronización que sí resultan eficientes, a menos que en el contexto dado esta implementación si resulte conveniente[7].

[7]En [MCS91, sec.1] podemos encontrar razones que validan el uso del *busy waiting* en implementaciones de primitivas de sincronización para computadoras MIMD con memoria compartida.

## 2.6 Técnicas para Reducir la Cantidad de Demostraciones

Veremos ahora teoremas válidos en el cálculo propuesto que simplifican en gran medida la tarea de probar la no interferencia. Las nuevas reglas que daremos no agregan nada al cálculo, sólo nos brindan atajos en las demostraciones de corrección global.

### 2.6.1 Invariantes Globales

Un *invariante global* será una aserción que puede ser puesta en conjunción con todas las aserciones de un programa paralelo. Como está en todos lados es usual adehrir a la convención de escribirlo una sola vez, como una nota aparte del texto del programa.

A manera de ejemplo, el programa (2.3) de la página 26 puede ser ligeramente transformado agregando el predicado $x = p + q$ como conjunción a la pre y postcondición de todo el programa. Este programa también es correcto, pero ahora $x = p + q$ cumple con la definición de invariante global, y lo reescribimos convenientemente.

$$
\begin{array}{l}
Inv : x = p + q \\
\\
\{x = 0 \wedge p = 0 \wedge q = 0\} \; [ \\
\qquad\qquad\qquad\qquad \{p = 0\}\texttt{x,p:=x+1,1}\{p = 1\} \parallel \\
\qquad\qquad\qquad\qquad \{q = 0\}\texttt{x,q:=x+1,1}\{q = 1\} \\
] \; \{x = 2 \wedge p = 1 \wedge q = 1\}
\end{array}
$$

El punto importante es que los invariantes globales no requieren que hagamos el test de no interferencia, o mejor dicho, demostrando la corrección local del invariante también estamos demostrando su correción global.

Supongamos que tenemos el invariante $H$ con su corrección local demostrada, entonces para ver que este predicado no es interferido por una sentencia $S$ de otra componente, tenemos que ver que la tripla

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

es teorema, pero esto ya ha sido demostrado en la prueba de corrección local de la componente en donde está $S$.

Operacionalmente podemos ver que el test de no interferencia no es necesario para los invariantes globales, pues todas las componentes cumplen con ellos, y por lo tanto ninguna lo falsificará.

Para el programa anterior las triplas a probar son:

1. Invariante global
   (a) $\{Inv \wedge p = 0\}\texttt{x,p:=x+1,1}\{Inv\}$
   (b) $\{Inv \wedge q = 0\}\texttt{x,q:=x+1,1}\{Inv\}$
2. Corrección local

(a) $\{p = 0\}$`x,p:=x+1,1`$\{p = 1\}$

(b) $\{q = 0\}$`x,q:=x+1,1`$\{q = 1\}$

3. Corrección global

(a) $\{p = 0 \wedge q = 0\}$`x,q:=x+1,1`$\{p = 0\}$

(b) $\{p = 1 \wedge q = 0\}$`x,q:=x+1,1`$\{p = 1\}$

(c) $\{q = 0 \wedge p = 0\}$`x,p:=x+1,1`$\{q = 0\}$

(d) $\{q = 1 \wedge p = 0\}$`x,p:=x+1,1`$\{q = 1\}$

En este caso, la cantidad de triplas a probar es mayor, pues se agregan las dos relacionadas con el invariante. Sin embargo la complejidad involucrada en cada una de las pruebas resulta menor. Este fenómeno se nota claramente en las triplas del test de no interferencia, donde siempre tenemos que el conjunto de variables modificadas por la sentencia es disjunto al conjunto de variables libres de predicado que necesitamos ver que es invariante. En la siguiente subsección veremos esto con más detalle.

Las demostraciones de estas ocho triplas son directas y dejamos que el lector se convenza de su validez.

### 2.6.2 Regla de Ortogonalidad

Si tenemos un predicado $P$ y una asignación `x:=E` tal que

$$x \notin free(P)$$

entonces podemos asegurar que

$$\{P\}\texttt{x:=E}\{P\} \quad ortogonalidad$$

La demostración de este hecho se basa en que $wlp(\texttt{x:=E}).P \equiv P(x := E)$ y $P(x := E) \equiv P$ cuando $x \notin free(P)$.

Sólo estamos diciendo que $P$ permanece invariante si sus variables no se modifican.

Con esta regla y la de la consecuencia probamos fácilmente las ternas de corrección global del último ejemplo.

Aunque éste y otros resultados parezcan triviales, son ampliamente utilizados y la experiencia muestra que es útil tener un nombre para un fenómeno que se repite frecuentemente.

### 2.6.3 Regla de *Widening*

Ésta también habla de la invariancia de un predicado en presencia de una asignación, sólo que en este caso la variable modificada sí forma parte de las variables libres del predicado. El punto es cómo poder asegurar que esta asignación no falsifica el predicado.

Si tenemos una relación transitiva $\preceq$ y $E$ es una expresión donde $x$ no ocurre entonces podemos dar la regla

$$\frac{P \Rightarrow x \preceq f.x}{\{P \wedge E \preceq x\}\texttt{x:=f.x}\{E \preceq x\}} \quad widening$$

Como $\{P \wedge E \preceq x\}$`x:=f.x`$\{E \preceq x\} \equiv P \wedge E \preceq x \Rightarrow E \preceq f.x$, y este último predicado es válido si se da el antecedente de la regla, vemos como ésta resulta válida.

Normalmente diremos que la asignación "hace más verdadero" o "valida" el predicado. Triplas demostradas con esta regla podrían ser

$$\{P \wedge b\}\texttt{b:=True}\{b\}$$
$$\{P \wedge x \leq K\}\texttt{x:=x-1}\{x \leq K\}$$

donde en la primera $\preceq \equiv \Rightarrow$, mientras que la segunda $\preceq \equiv \geq$

### 2.6.4 Regla de las Aserciones Disjuntas

Cuando tenemos un predicado $P$ cuyo espacio de estados es disjunto al que requiere como precondición una sentencia $S$ de otra componente, entonces la sentencia no interfiere con el predicado. La regla es la siguiente

$$\frac{P \wedge pre(S) \equiv false}{\{P \wedge pre(S)\}S\{P\}} \quad aserciones\ disjuntas$$

y surge del hecho que $\{false\}S\{P\}$ es teorema para cualquier $S$ y $P$.

Operacionamente, suponiendo la corrección de la anotación, podemos entender esta regla como que no es posible que las sentencia $S$ esté para ejecutarse en un estado que satisfaga $P$. Es decir, no puede darse el caso de que ambas sentencias estén habilitadas para ejecutarse al mismo tiempo, pues sus condiciones de inicio son disjuntas.

## 2.7 Terminación, Progreso, Deadlock Total y la Multicota

Todo lo hecho hasta ahora tuvo como propósito demostrar la corrección parcial de los programas paralelos con variables compartidas y sincronización. Nada se habló acerca de terminación y sólo lateralmente se mencionó el problema del deadlock.

Es posible extender la teoría dada hasta ahora a fin de poder garantizar dos propiedades de seguridad (*safety properties*) fundamentales: *divergence freedom* o terminación y *deadlock freedom*. La primera asegura la ausencia de computaciones infinitas, mientras que la segunda afirma que no habrá bloqueos infinitos. La corrección parcial junto con estas dos propiedades forman la denominada *corrección total*.

Esta corrección total sólo habla del subconjunto de programas que terminan, y excluye cosas tan usuales como sistemas operativos y protocolos de comunicación cuyas computaciones son divergentes por definición. En estos casos, la noción de terminación ya no es válida y se reemplaza por la de *liveness* o *progreso*. Demostrar esta propiedad no está al alcance de nuestra teoría, por lo que apuntaremos a pruebas semi-formales.

### 2.7.1 Corrección Total

***Divergence Freedom o Terminación***

El primer aspecto a tener en cuenta es la divergencia y ésta sólo puede ocurrir en presencia de programas que incluyan constructores de repetición.

Para los programas secuenciales no-determinísticos, el acercamiento usual para demostrar la ausencia de divergencia es la inclusión de expresiones de cota para las repeticiones. A cada bucle se le asocia una expresión entera $t$ y se demuestra que en cada iteración esta cota decrece estrictamente y además $t = 0$ implica que todas las guardas son falsas y por ende el bucle termina.

La regla del **do**...**od** se ve modificada para asegurar estas condiciones

$$\frac{\forall i\{P \wedge B_i\}S_i\{P\},\ \forall i\{P \wedge B_i \wedge t{=}C\}S_i\{t{<}C\},\ P \wedge t{=}0 \Rightarrow \bigwedge_i \neg B_i}{\{P\}\mathbf{do} B_1 \rightarrow S_1 \,\square \cdots \square\, B_n \rightarrow S_n \mathbf{od}\{P \wedge \neg(B_1 \wedge \cdots \wedge B_n)\}}$$

Sin embargo cuando el constructor de repetición está dentro de uno de composición paralela, para poder asegurar que las cotas sigan siendo decrecientes tenemos que modificar el test de interferencia, a fin de que las otras componentes además de no interferir en la corrección local de las aserciones no incrementen las cotas de los lazos.

Ahora el test de interferencia pide además que dado el lazo $W$ con cota $t$, que otras componentes pueden interrumpir, todas las sentencias $S$ de las otras componentes deben cumplir:

$$\{t = C \wedge pre(S)\}S\{t \leq C\}$$

***Deadlock Freedom***

Es ahora el turno de ver que es lo que se necesita para asegurar la ausencia de *deadlock*.

Las componentes de los programas paralelos que incluyen constructores de selección con guardas no exhaustivas, usualmente resultan bloqueadas y desbloqueadas a lo largo de la computación. Cuando algún subconjunto de componentes se bloquea de manera permanente (por más que las otras componentes avancen y en algún momento terminen) diremos que se está en un *estado de deadlock*. Llegar a este estado es poco deseable y el objetivo será dar condiciones necesarias para asegurar *deadlock freedom*.

El siguiente método está incluido en la tesis de Susan Owicki y se basa en enumerar todas las situaciones de deadlock potencial y luego demostrar que ninguna de ellas puede ocurrir.

Llamaremos a la tupla $(R_1, \cdots, R_n)$ un *deadlock potencial* del programa paralelo $S \equiv [S_1 \parallel \cdots \parallel S_n]$ cuando cada $R_i$ es una sentencia **if**...**fi** ó el símbolo $E$ que denota la sentencia vacía y significa la terminación de la componente. Además se pide que haya al menos un $R_i$ distinto al programa vacío.

Con cada una de las ternas $\{P_i\}S\{Q_i\}$ de la *proof outline* de cada componente generamos tuplas de aserciones $(r_1, \cdots, r_n)$ asociadas a cada deadlock potencial donde

$$r_i = \begin{cases} pre(R_i) \wedge \neg(\bigvee_j B_j^i) & si\ R_i \equiv \mathbf{if}\ \square_j B_j^i \rightarrow S_j^i\ \mathbf{fi} \\ Q_i & c.c. \end{cases}$$

Entonces si logramos demostrar $\neg \bigwedge_i r_i$ para cada tupla $(r_1, \cdots, r_n)$, habremos dado condiciones suficientes para que cada uno de estos *deadlocks* potenciales no ocurran, pues alguna de las selecciones siempre podrá continuar.

Hay dos tipos de tuplas en las que la condición resulta verdadera de manera directa

- Cuando un $R_i$ es un constructor de selección que tiene guardas exhaustivas, es decir que se da $\bigvee_i B_i$.
- Cuando un $R_i$ es la sentencia vacía $E$ y la postcondición $Q_i$ es falsa debido a que la componente no termina.

### 2.7.2 Progreso

Con todo lo anterior tenemos la corrección total, sin embargo sólo estamos hablando de programas que terminan. Cuando tratamos con programas paralelos que no lo hacen, la premisa de libertad de divergencia no se puede cumplir y usualmente es reemplazada por la noción de *progreso de cada componente*. Lamentablemente existen dos problemas fundamentales:

- Resulta difícil definir esta noción dentro de nuestro formalismo "*à la Hoare*".
- El problema de progreso o *liveness* en toda su generalidad es uno de los problemas denominados difíciles en las Ciencias de la Computación

El enfoque general involucra utilizar *lógicas temporales* para demostrar no-deadlock, no-inanición individual y no-bloqueo "después de usted, después de usted" entre otros, además de tener que incluir en la discusión el concepto de *fairness* para limitar, o mejor dicho, para especificar el no-determinismo de las computaciones paralelas.

Nosotros manejaremos argumentos semi-formales, pero totalmente rigurosos. Probaremos progreso para una familia de multiprogramas frecuentemente aparecen. Para el resto, deberemos definir mediante aserciones qué entendemos por progreso en cada caso y mostraremos que éstas resultan válidas. Sólo estaremos diciendo que ciertas propiedades se cumplen, y que dichas propiedades, en algún sentido, reflejan para nosotros el significado de progreso[8].

### 2.7.3 Deadlock Total y la Multicota

Ahora veremos un caso especial donde podemos demostrar progreso en la composición paralela de componentes secuenciales que no terminan.

Como *deadlock total* entenderemos la situación en que todas las componentes están bloqueadas esperando una condición. Como ninguna componente puede progresar, el estado resulta estable y por lo tanto el bloqueo es infinito. Sea cual fuere nuestra noción de progreso, está claro que el deadlock total va en contra de ella.

En el test de *deadlock freedom* dado anteriormente, cuando todas las componentes paralelas no terminan, sus postcondiciones $Q_i$ son falsas y cualquier

[8] Y esto es exactamente lo que hace [vdS94, sec.8] con los *zeno invariants*.

tupla de deadlock potencial que incluya el programa vacío será trivialmente verdadera. Entonces, nuestra tarea consistirá en enumerar los *deadlocks* potenciales $(R_1, \cdots, R_n)$ donde todos los $R_i$ son constructores de selección, y por cada uno demostrar que

$$\bigwedge_i \left( pre(R_i) \Rightarrow \bigvee_j B_j^i \right)$$

o equivalentemente

$$\bigwedge_i pre(R_i) \;\Rightarrow\; \bigvee_{i,j} B_j^i$$

es decir, dadas la conjunción de las precondiciones de las guardas, alguna de éstas resulta verdadera y el multiprograma puede continuar en al menos una componente. Aclaramos que la expresión anterior es una reescritura de la condición de *deadlock freedom* de la sección 2.7.1.

Ahora supongamos que las componentes están acopladas de tal manera que avanza una si y solo si avanzan todas. Una forma de probar esto es encontrar un invariante global de la forma

$$x_1 \leq x_2 + K_1 \wedge x_2 \leq x_3 + K_2 \wedge \cdots \wedge x_n \leq x_1 + K_n$$

siendo $x_i$ una variable privada de $S_i$ y $K_i$ una constante natural.

Entonces si nuestra noción de progreso en cada componente está asociada al incremento de su variable $x_i$, tendremos que el progreso de cualquier componente implica el progreso de todas.

Si podemos demostrar la ausencia de deadlock total y además tenemos la existencia de una multicota, entonces el progreso de todas las componentes estará asegurado.

Aunque el caso resulta completamente particular, ayuda en una clase de problemas comunes y será utilizado para demostrar progreso en algunos de nuestros algoritmos paralelos.

### 2.7.4 *Fairness*

Nuestros programas paralelos presentan un alto grado de no-determinismo, pues por un lado tenemos el de las guardas y por el otro el del paralelismo. Ciertamente existen casos en los que el no-determinismo debe ser limitado a fin de poder asegurar propiedades que van más allá de la corrección parcial, como lo son el progreso, la ausencia de deadlock parcial, la asignación de prioridades, la ausencia de bloqueo "después de usted, después de usted" y la inanición individual.

Usualmente este no-determinismo está asociado a un demonio, por lo que el concepto de *fairness* se aplicará a la forma que este demonio elige.

Existen básicamente dos variantes para este concepto: *weak fairness* y *strong fairness* y además la manera de expresarlo cambia dependiendo si se aplica al no determinismo del paralelismo o al de las guardas.

Introduzcamos mediante ejemplos estas cuatro variantes.

Cuando dijimos en la sección 1.2 que la solución 3 no era válida, estábamos suponiendo que el demonio no era siquiera *weakly fair*

**Definición 2.4 (*weak fairness* — paralelismo)** *El demonio elige de tal forma que a la larga todas las componentes son escogidas infinitas veces, o equivalentemente, ninguna componente es ignorada de manera permanente.*

Básicamente estamos diciendo que las componentes progresan a una velocidad no especificada pero positiva y esta es una condición que siempre supondremos que cumple el demonio.

En el siguiente ejemplo, la *weak fairness* no alcanza para asegurar el progreso de la primera componente de la composición paralela.

$$\texttt{a:=false;}[\textbf{if } a \rightarrow \textbf{skip fi} \parallel \textbf{do } \texttt{true} \rightarrow \texttt{a:=true; a:=false } \textbf{od}]$$

La guarda $a$ es infinitas veces verdadera e infinitas veces falsa, pero si nuestro demonio *weakly fair* siempre elige la primera componente cuando la condición es falsa, no habrá progreso en ésta.

**Definición 2.5 (*strong fairness* — paralelismo)** *El demonio activa infinitas veces una componente bloqueada en una guarda que se hace verdadera infinitas veces.*

Restringiendo de esta forma el no-determinismo de la composición paralela aseguramos que el ejemplo anterior progresa en su primera componente.

Para el caso del no-determinismo en las guardas de un programa en el lenguaje de comandos guardados, las situaciones son más o menos similares.

En el siguiente programa vemos que la terminación no está garantizada a menos que el demonio elija alguna vez la segunda guarda.

```
a:=False;
do
      ¬a → skip
   □ ¬a → a:=true
od
```

Para que termine deberemos pedir *weak fairness*

**Definición 2.6 (*weak fairness* — guardas)** *Todo comando guardado de un constructor de repetición* ***do*** *...* ***od*** *cuya guarda se habilita de manera continua en algún momento, resulta elegida infinitas veces.*

Mediante otro ejemplo veamos que con la condición anterior no nos alcanza en general para asegurar progreso.

```
a:=False;
b:=True;
do
      ¬a → b:=¬b
   □ ¬a ∧ b → a:=true
od
```

Claramente si cada vez que ambas guardas están activas, el demonio elige la primera, $a$ será constentamente falsa y nuestro programa no terminará. Si ahora en cambio, nuestro no-determinismo para elegir las guardas está limitado por la *strong fairness*, la terminación de este último texto de programa queda asegurada.

**Definición 2.7 (*strong fairness* — guardas)** *Todo comando guardado que es habilitado infinitas veces, será activado infinitas veces.*

Las nociones de *fairness* tanto para paralelismo como para programas secuenciales no-deterministas, pueden ser unificadas mediante la transformación de programas paralelos en programas secuenciales no-determinísticos. En [AO91, sec.7.6] se propone una transformación de este estilo, que claramente unifica ambas formas de expresar *fairness*.

# Capítulo 3

# Derivación de programas Paralelos

## 3.1 Introducción

Lo que hemos presentado hasta ahora es la teoría de Owicki-Gries , que resulta una extensión de la lógica de Hoare para tratar con programas paralelos. En ella, además de la corrección pre-post pedida en la lógica de Hoare, que aquí se denomina corrección local, se pide la corrección global o no-interferencia. Usualmente la teoría de Owicki-Gries se resume en el siguiente párrafo:

> *Cada aserción es establecida por la componente en la que ocurre y es preservada por las sentencias atómicas de todas las otras componentes*

También mostramos cómo podemos reducir la cantidad de demostraciones a realizar recurriendo a nuevas reglas que engloban en una sola referencia demostraciones sencillas, que resultan muy frecuentes en la verificación de multiprogramas.

Sin embargo poco se dijo acerca del carácter *"a posteriori"* de la teoría, es decir que ella nos permite, dado un un programa decorado con aserciones, decir si estas últimas resultan correctas o no. De dónde se sacó el programa junto con sus anotaciones, no resulta para nada relevante a la teoría.

Éste no es un problema menor, pues dado un programa paralelo, usualmente resulta bastante difícil inventar aserciones que por un lado resulten correctas y por el otro sean relevantes a las propiedades que deseamos probar.

La respuesta a este problema está en imitar lo hecho por Dijkstra en el desarrollo estructurado de programas secuenciales. Teniendo como base la teoría de Owicki-Gries, la escuela de Eindhoven diseñó una metodología que nos permite paso a paso, en demanda y de manera sistemática, construir multiprogramas anotados que cumplan con nuestras especificaciones iniciales.

Teniendo en cuenta lo fuerte y establecida que es esta área en la programación secuencial, es de esperar que se obtengan buenos resultados con su extensión a multiprogramas

Antes de mostrar esta metodología con todo el detalle, veremos algunas transformaciones que resultan útiles en el proceso de derivación asi como varios teoremas. Para finalizar la sección, mostrartemos dos técnicas que son muy utilizadas en las derivaciones.

## 3.2 Transformaciones y Teoremas

Brindaremos en esta sección una serie de herramientas teóricas que nos ayudarán en el proceso de derivación de multiprogramas.

### 3.2.1 Transformación de Leibniz

Con esta regla podemos cambiar la expresión de una asignación siempre que hayamos probado la igualdad de las expresiones en la aserción previa a la sentencia. Es decir, el fragmento de programa/prueba

$$\{E = F\}\texttt{x:=E}$$

puede ser transformado a

$$\{E = F\}\texttt{x:=F}$$

sin modificar la corrección del multiprograma anotado.

Para validar esta afirmación sólo tenemos que ver que toda prueba que involucre el fragmento original puede ser transformada en una que contenga la asignación cambiada sin modificar la prueba.

Una prueba con la asignación original tendrá la forma

$$\{P \wedge E = F\}\texttt{x:=E}\{Q\}$$

y esto por definición de *wlp* o por las equivalencias, resulta

$$P \wedge E = F \Rightarrow Q[x := E]$$

Pero esta, por regla de Leibniz del cálculo proposicional, es equivalente a

$$P \wedge E = F \Rightarrow Q[x := F]$$

y de nuevo por las reglas de equivalencia podemos llegar a

$$\{P \wedge E = F\}\texttt{x:=F}\{Q\}$$

### 3.2.2 Transformación de Debilitamiento de Guarda

Si tenemos un fragmento de programa correctamente anotado de la siguiente forma

$$\{B\}\textbf{if } B \wedge C \rightarrow S \textbf{ fi}$$

lo podemos cambiar por

$$\{B\}\textbf{if } C \rightarrow S \textbf{ fi}$$

manteniendo la corrección.

Para demostrar este hecho tenemos que ver que toda prueba que involucre este fragmento es de la forma

$$\{P \wedge B\}\textbf{if } B \wedge C \rightarrow S \textbf{ fi}\{Q\}$$

por lo que anteriormente hemos tenido que demostrar el antecedente de la regla de selección

$$\{P \wedge B \wedge B \wedge C\}S\{Q\}$$

y esto es suficiente para concluir

$$\{P \wedge B\}\textbf{if } C \rightarrow S \textbf{ fi}\{Q\}$$

luego nuestra transformación resulta válida.

### 3.2.3 Transformación de Fortalecimiento de la Guarda

El fragmento de programa

$$\textbf{if } B \rightarrow S \textbf{ fi}$$

puede ser reemplazado por

$$\{C \Rightarrow B\}\textbf{if } C \rightarrow S \textbf{ fi}$$

sin modificar la corrección de la prueba, siempre que hayamos demostrado la corrección local y global de la preaserción $C \Rightarrow B$. Veamos la demostración.

Cada prueba del fragmento de programa original es de la forma

$$\{P\}\textbf{if } B \rightarrow S \textbf{ fi}\{Q\}$$

entonces el antecedente

$$\{P \wedge B\}S\{Q\}$$

debe ser válido. Utilizando la regla de la consecuencia tenemos que de lo anterior se deduce

$$\{P \wedge B \wedge C\}S\{Q\}$$

Reescribiendo la precondición esta regla es equivalente a

$$\{P \wedge (C \Rightarrow B) \wedge C\}S\{Q\}$$

de la que se deduce por la regla de la selección

$$\{P \wedge (C \Rightarrow B)\}\textbf{if } C \rightarrow S \textbf{ fi}\{Q\}$$

Y como por hipótesis tenemos la corrección local y global de la precondición $C \Rightarrow B$, esto resulta válido dentro del contexto que estaba la sentencia original.

Un caso particular, pero muy útil, dice que el siguiente fragmento de código correctamente anotado

$$\{C \Rightarrow B\}\textbf{if } B \rightarrow S \textbf{ fi}$$

puede ser reemplazado por

$$\{C \Rightarrow B\}\textbf{if } C \rightarrow S \textbf{ fi}$$

sin modificar la corrección de la prueba.

Un punto a destacar de esta transformación es que, aunque resulta transparente respecto a la corrección parcial, puede hacer peligrar el progreso. La guarda es más fuerte y por lo tanto mayor es el espacio de estados que produce bloqueo.

Un caso extremo es cuando fortalecemos a tal punto la guarda que reemplazamos de manera totalmente válida el fragmento

$$\textbf{if } B \rightarrow S \textbf{ fi}$$

por

$$\{false \Rightarrow B\}\textbf{if } \texttt{False} \rightarrow S \textbf{ fi}$$

donde la corrección parcial se mantiene, pero ahora nuestro programa carecerá de posiblidades de progresar en la componente que contiene la transformación.

Finalmente diremos que esta transformación es muy utilizada para reducir el grado de atomicidad, desacoplando multiasignaciones de variables compartidas. En la sección 3.4.2 se verá esta técnica en detalle.

### 3.2.4 Transformación de la Conjunción de Guardas

Si reemplazamos el fragmento de programa

$$\{P\}\textbf{if } B \wedge C \rightarrow S \textbf{ fi}\{Q\}$$

por la secuencia de acciones atómicas

$$\{P\}\textbf{if } B \rightarrow \textbf{skip fi};\{P \wedge B\}\textbf{if } C \rightarrow S \textbf{ fi}\{Q\}$$

entonces se preserva la corrección de la prueba siempre que la aserción intermedia $\{P \wedge B\}$ se pruebe globalmente correcta.

Veremos como el reemplazo surge desde el fragmento original aplicando algunas de las transformaciones vistas anteriormente.

Si agregamos un skip guardado que siempre se satisface, no modificamos nada, pues éste resulta equivalente a un **skip**.

$$\{P\}\textbf{if } \texttt{True} \rightarrow \textbf{skip fi}\{P\}$$
$$\{P\}\textbf{if } B \wedge C \rightarrow S \textbf{ fi}\{Q\}$$

Como $B \Rightarrow \mathit{true}$ es universalmente válido, podemos usar el fortalecimiento de la guarda y obtener

$$\{P\}\textbf{if } B \rightarrow \textbf{skip fi}\{P\}$$
$$\{P\}\textbf{if } B \wedge C \rightarrow S \textbf{ fi}\{Q\}$$

Luego resulta directo ver que si adjuntamos $B$ a la aserción intermedia $P$, esta resulta local y globalmente correcta. La parte local esta dada por el skip guardado, mientras que la global es una de las condiciones de aplicabilidad de la transformación. Entonces tenemos

$$\{P\}\textbf{if } B \rightarrow \textbf{skip fi}\{P\}$$
$$\{P \wedge B\}\textbf{if } B \wedge C \rightarrow S \textbf{ fi}\{Q\}$$

Finalmente por el teorema de debilitamiento de la guarda, llegamos a la equivalencia

$$\{P\}\textbf{if } B \rightarrow \textbf{skip fi}\{P\}$$
$$\{P \wedge B\}\textbf{if } C \rightarrow S \textbf{ fi}\{Q\}$$

Como se apuntó anteriormente las pruebas de corrección de multiprogramas con fino grano de atomicidad resultan complicadas. Esta transformación reduce el grado de atomicidad de una forma casi gratuita, pues solamente debemos probar la corrección global de una nueva aserción.

Teoremas como éste escasean, y será una práctica común tratar de que nuestros programas cumplan las condiciones de aplicabilidad a fin de poder utilizarlo para reducir el grado de atomicidad.

### 3.2.5 *Modus Ponens* para el Skip Guardado

Este teorema dice que si en el siguiente fragmento de programa

$$\{B \Rightarrow C\}\textbf{if } B \rightarrow \textbf{skip fi}\{C\}$$

la preaserción $B \Rightarrow C$ es correcta, entonces la postaserción $\{C\}$ será localmente correcta.

La prueba resulta de la aplicación de la defición del *wlp* para el constructor de selección y la regla del *modus ponens* de la lógica.

Un corolario inmediato nos dice que en

$$\textbf{if } B \rightarrow \textbf{skip fi}\{B\}$$

la postcondición $\{B\}$ es localmente correcta.

La utilidad de este teorema reside en que podemos asegurar condiciones *locales* a partir de condiciones globales como puede ser una implicación invariante. El corolario nos permite hacer localmente correcta una aserción mediante la introducción de un constructor de selección que actúa como primitiva de sincronización.

### 3.2.6 Debilitamiento de la Aserción

Dado un multiprograma anotado, si $P$ es una aserción correcta y

$$[P \Rightarrow Q]$$

entonces $Q$ también lo será.

La prueba resulta inmediata de la interpretación operacional de las aserciones. Si en cierto punto, el estado de nuestro multiprograma cumple con un predicado, entonces este estado hará verdadera cualquier condición más débil.

La utilidad de este teorema reside en que durante la derivación de un multiprograma, resulta usual fortalecer las aserciones. Cuando finalmente llegamos a demostrar todo el multiprograma, la solución que encontremos servirá para el problema original con aserciones más débiles.

### 3.2.7 Topología de Programas

Bajo este nombre se agrupan varios teoremas que prueban la corrección de aserciones que resultan evidentes de la "forma" que tiene el multiprograma. Solamente veremos uno de ellos, aunque debemos remarcar que existe toda variedad en este respecto.

Dado el siguiente multiprograma que es general en cuanto al número de componentes[1], el teorema entonces dice que el invariante así como las aserciones $0 < x$ de cada componente son correctas.

```
Pre:  x=0
Inv:  0≤x
Comp.i:  do True → x:=x+1;
                   {0<x}
                   x:=x-1
         od
```

[1] Aunque se ha usado una notación alternativa para mostrar un multiprograma con sus aserciones, queda claro como se relaciona con la forma que veníamos utilizando.

Para ver que esto es cierto, centremos nuestra atención en el siguiente multiprograma

```
Pre:  ∀i::c.i=0
Inv:  ∀i::0≤c.i
Comp.i:  do True → c.i:=c.i+1;
                   {1≤c.i}
                   c.i:=c.i-1
         od
```

donde $c.i$ es una variable local a la componente i-ésima. Demostremos la corrección del invariante y de las aserciones intermedias.

Para ver que el invariante es correcto sólo nos basta con probar su corrección local y que este resulte válido en la precondición. Esto último se da de manera inmediata pues $c.i = 0 \Rightarrow 0 \leq c.i$. La sentencia `c.i:=c.i+1` lo mantiene por la regla del *widening*, y en el caso de `c.i:=c.i-1`, podemos ver rápidamente que

$$\{Inv \wedge 1 \leq c.i\}\texttt{c.i:=c.i-1}\{Inv\}$$

siempre que supongamos que $1 \leq c.i$ es una aserción que resulta válida. Aseguremos esto ahora.

Como $wlp.(\texttt{c.i:=c.i+1}).(1 \leq c.i) \equiv 0 \leq c.i$, y esto es consecuencia del invariante, hemos probado lo que queríamos.

Ahora unimos los dos programas y mediante el nuevo invariante establecemos la relación entre ellos.

```
Pre:  x=0 ∧ ∀i::c.i=0
Inv:  ∀i::0≤c.i ∧ x=∑i::c.i ∧ 0≤x
Comp.i:  do True → x,c.i:=x+1,c.i+1;
                   {1≤c.i} {0<x}
                   x,c.i:=x-1,c.i-1
         od
```

El invariante agregado resulta válido en la precondición y claramente es mantenido por las dos multiasignaciones. Además nos permite concluir que el invariante $0 \leq x$ es válido, así como la aserción intermedia $0 < x$.

Para terminar la prueba basta con notar que $\{c.i\}$ es un conjunto de variables auxiliares y por regla correspondiente podemos llegar al programa-teorema inicial.

## 3.3 Dos Ejemplos

El objetivo de esta sección será, a través de dos ejemplos sencillos, proponer un estilo de derivación de multiprogramas.

Primero derivaremos un algoritmo muy simple que sincroniza la ejecución repetida de dos componentes. Luego nos avocaremos a la tarea de derivar el algoritmo de la búsqueda lineal paralela que nos sirvió de motivación en la introducción de este trabajo. Ésto último nos permitirá ver claramente las

ventajas del desarrollo estructurado de programas frente al enfoque intuitivo que se planteaba en la sección 1.2.

A lo largo de ambos ejemplos introduciremos notación y nomenclatura, además serán resaltadas heurísticas que nos guiarán en el desarrollo.

### 3.3.1 Sincronización de Fase

La motivación para este problema es mantener sincronizadas dos componentes que se ejecutan cíclicamente en paralelo, evitando en todo momento que una se adelante a la otra en más de un ciclo.

Tenemos dos componentes de la siguiente forma:

```
Comp.A: do                    Comp.B: do
          True → bodyA                  True → bodyB
        od                            od
```

El siguiente paso será transformar este código para que podamos poner nuestro requerimiento de sincronización en términos de aserciones, a fin de poder trabajarlas con la teoría de Owicki-Gries.

Agreguemos dos variables, $a$ y $b$, que contarán la cantidad de veces que cada cuerpo se ejecuta. Obviamente pedimos que el espacio de estados de `bodyA` y `bodyB` sea ortogonal al espacio generado por $\{a, b\}$.

Nuestro requerimiento de sincronización entonces estará dado por el predicado $a \leq b + 1 \wedge b \leq a + 1$, y como nada importa lo que hagan los cuerpos de las componentes $A$ y $B$, los eliminamos de la discución a fin de quedarnos con lo esencial a nuestro problema.

Además supondremos que en el inicio de la computación paralela, las variables $a$ y $b$ son iguales aunque de valor desconocido.

La siguiente especificación resume lo que decimos.

```
Pre :  a = b
Inv :  a≤b+1 ∧ b≤a+1
A: *[ a:=a+1 ]                B: *[ b:=b+1 ]
```

Promesa: - Sólo los incrementos modifican $a$ y $b$.

**Notación** `*[ A ]` $\equiv$ **do** `True` $\rightarrow$ `A` **od**

Con la promesa indicamos explícitamente una condición que queremos mantener a lo largo de toda la derivación, y en este caso es la ortogonalidad de cualquier cosa que hagamos o agregemos respecto a las variables que cuentan la cantidad de ciclos. Nada nos impide leerlas.

Entonces esta última será nuestra *computation proper*[2] que representa el punto de partida de la derivación y además marca la estructura general la cual no podrá ser modificada, y por lo tanto será la que tenga la solución.

[2]Esta expresión significa "la computación en sí misma".

El invariante expresado anteriormente es una aserción que todavía no ha sido probada. El objetivo de ahora en adelante será tratar de hacer válida esta aserción extendiendo el texto del programa mediante sentencias, aserciones e invariantes.

Nuestra primera *decisión de diseño* será indicar los lugares donde podemos insertar código mediante los identificadores $A_1$, $A_2$, $B_1$ y $B_2$, además marcaremos de manera explícita que tenemos una obligación de prueba con el invariante, utilizando un signo de pregunta y una referencia a una *nota*.

```
Pre :  a = b
Inv :  a≤b+1 ∧ b≤a+1?N0
A: *[ A1 ;                    B: *[ B1 ;
      a:=a+1;                       b:=b+1;
      A2 ]                          B2 ]
```

Promesa: - Sólo los incrementos modifican $a$ y $b$.

El siguiente paso es, mediante la nota 0, demostrar o bien que la aserción con signo de pregunta es válida o encontrar qué es lo que tenemos que agregar al texto de programa para que lo sea.

Usualmente ocurre que para probar la validez de una aserción debemos introducir nuevas aserciones que a su vez necesitarán ser probadas y, como en la anterior, nuestro marcador será un signo de pregunta asociado a una nota.

Este será el motor de nuestra derivación. Paso a paso iremos levantando necesidades de prueba generando código y nuevas necesidades de prueba, cuando finalmente hayamos quitado todos los signos de pregunta habremos llegado a una solución que sabremos que respeta la *computation proper* y además resulta correcta.

Ahora trabajemos en la nota 0. Como vimos en la sección 2.6.1, para demostrar que un invariante es válido, no es necesario ver la corrección global pues está implicada por la corrección local, sin embargo sí tendremos que demostrar que el invariante es implicado por la precondición. Esta prueba será indicada con I, mientras que la primera será marcada con una L.

Nota 0:

I: Como $a = b \Rightarrow a \leq b + 1 \wedge b \leq a + 1$ entonces el invariante es válido inicialmente.

L: Al conocer dos instrucciones, sólo tendremos que ver que preaserciones necesitamos para validar las ternas $\{P_1 \wedge Inv\}$`a:=a+1`$\{Inv\}$ y $\{P_2 \wedge Inv\}$`b:=b+1`$\{Inv\}$, es decir que el invariante no sea falsificado por ninguna de las dos. Claramente necesitamos fortalecer la preaserción de los incrementos pues las ternas $\{Inv\}$`a:=a+1`$\{Inv\}$ y $\{Inv\}$`b:=b+1`$\{Inv\}$ no son válidas en el modelo.

- $\{P_1\}$`a:=a+1`

  Tenemos que encontrar $P_1$ tal que $P_1 \wedge Inv \Rightarrow wlp.$(`a:=a+1`$).Inv$, o sea $P_1 \Rightarrow (Inv \Rightarrow Inv(a := a + 1))$, que resulta equivalente a $P_1 \Rightarrow a \neq b + 1$.

Podemos utilizar desde la solución más débil ($P_1 \equiv a \neq b+1$) hasta la más fuerte ($P_1 \equiv false$). Nos alcanzará con hacer que la precondición de `a:=a+1` sea $a \leq b$, que sirve para resolver el problema y además es sencilla de evaluar en caso de que tengamos que convertirla en la guarda de una sentencia.

- $\{P_2\}$`b:=b+1`

  Aunque podemos hacer un desarrollo completamente simétrico al anterior, vamos a ver una forma alternativa de encontrar $P_2$. Como[3]

  $$\{P\}S\{Q_1 \wedge Q_2\} \equiv \{P\}S\{Q_1\} \wedge \{P\}S\{Q_2\}$$

  entonces podemos descomponer el invariante en dos.

  La terna $\{P_2 \wedge Inv\}$`b:=b+1`$\{a \leq b+1\}$ sale directamente por la regla del *widening*, mientras que $\{P_2 \wedge Inv\}$`b:=b+1`$\{b \leq a+1\}$ es equivalente a $P_2 \wedge Inv \Rightarrow b \leq a$.

  Vemos que con $P_2 \equiv b \leq a$ se cumple con la condición anterior y además resulta simétrico a $P_1$.

En todas las pruebas para la nota hicimos economía de notación, pues todos los predicados se suponen universalmente cuantificados, es decir, donde escribimos $P_1 \wedge Inv \Rightarrow b \leq a$ estamos expresando esto mismo pero entre corchetes.

Hemos resuelto la incógnita en la aserción invariante, generando dos nuevas aserciones que deberán ser probadas en notas posteriores. Indicamos la última aserción demostrada con un corazón y las dos nuevas generadas con un signo de pregunta con referencia a sus respectivas notas.

```
Pre :  a = b
Inv :  a≤b+1 ∧ b≤a+1♡
```

```
A: *[ A1 ;                    B: *[ B1 ;
      {a≤b?N1}                      {b≤a?N2}
      a:=a+1;                       b:=b+1;
      A2 ]                          B2 ]
```

Promesa: - Sólo los incrementos modifican $a$ y $b$.

También podríamos haber incluido entre las promesas que el invariante no será falsificado por las sentencias que todavía no fueron elegidas ($A_1$, $A_2$, $B_1$, $B_2$), pero este hecho es implicado por la promesa existente.

En este simple avance ya podemos visualizar nuestra metodología. Paso a paso iremos desarrollando nuevas versiones, donde cada una sólo tendrá en cuenta su inmediata anterior, y la interfase serán las aserciones con signos de pregunta. Nuestro proceso de desarrollo seguirá un patrón parecido al de un cálculo, donde en todo momento estamos atentos a obtener la siguiente expresión a partir de la última versión. Las decisiones de diseño serán las que definan el

[3]Esta es una regla básica de las triplas de Hoare, conocida como la *regla de la conjuntividad*.

camino cuando éste se bifurque, y en este punto las heurísticas nos ayudarán a tomar las decisiones. Citando a [Kal90, p.ix-x]

> "Durante los '80 W. H. J. Feijen y otros refinaron este método que se conoce como *programación con estilo de cálculo*: en gran medida, los programas son derivados a partir de su especificación a través de manipulaciones de fórmulas. Los cálculos que llevan al algoritmo son hechos en pequeños pasos, y así cada paso es verificado fácilmente. De esta forma las decisiones de diseño se ponen de manera manifiesta. Estas decisiones se basan en distintas consideraciones, tales como eficiencia, simplicidad y simetría. Este método no sólo nos ayuda a encontrar la solución, también puede dar nuevas soluciones que a menudo son bastante sorprendentes. La derivación de programas no es un acto mecánico: es una actividad desafiante que requiere creatividad. Esta manera de programar muestra de dónde viene la creatividad."

El siguiente paso en la derivación es asegurar la nota 1 y para esto deberemos probar que la aserción resulta local y globalmente correcta, lo primero será denotado con L y lo segundo con G.

Nota 1:
L: Utilizando el *modus ponens* para skip guardados de la sección 3.2.5 hacemos $A_1$ : **if** $a \leq b \rightarrow$ **skip fi**, nos aseguramos la corrección local de la aserción y además seguimos cumpliendo con la promesa. En este punto vemos el porqué de elegir $a \leq b$ como precondición y no la aserción más débil $a \neq b + 1$.

G: Tenemos que demostrar la invariancia de la aserción con respecto a cada instrucción de la componente $B$. Como sólo hay una definida, lo haremos respecto a ella. Podríamos también poner que esta aserción no resulte falsificada por $B_1$ y $B_2$, pero esto nuevamente resulta implicado por la promesa existente y la regla de ortogonalidad.

- $\{b \leq a\}$`b:=b+1`

  Por *widening* tenemos que $\{a \leq b\}$`b:=b+1`$\{a \leq b\}$, y por regla de consecuencia entonces se da $\{a \leq b \wedge b \leq a\}$`b:=b+1`$\{a \leq b\}$.

La siguiente versión es:

```
Pre :  a = b
Inv :  a≤b+1 ∧ b≤a+1
A: *[ if a≤b → skip fi;        B: *[ B1 ;
      {a≤b♡}                        {b≤a?N2}
      a:=a+1;                       b:=b+1;
      A2 ]                          B2 ]
```

Promesa: - Sólo los incrementos modifican $a$ y $b$.

Hemos borrado el corazón del invariante y transformado la nota 1 en corazón mediante la sentencia de sincronización surgida de la regla del modus ponens.

Avancemos un paso más, pero esta vez lo haremos rápidamente pues la situación para la nota 2 es completamente simétrica a la nota 1, y este será el ritmo que normalmente utilizaremos en las derivaciones.

Nota 2:
L: Prefijamos con un skip guardado, es decir $B_1$ : **if** $b \leq a \rightarrow$ **skip fi**.
G:

- $\{a \leq b\}$`a:=a+1`: por *widening*.

Ahora nuestro programa ya no contiene aserciones con obligaciones de prueba, sin embargo todavía tenemos que definir las sentencias $A_2$ y $B_2$. Como todo está demostrado elegimos eliminarlas o equivalentemente decir que ambas son **skip**.

La versión final es entonces:

```
Pre :  a = b
Inv :  a≤b+1 ∧ b≤a+1
A: *[ if a≤b → skip fi;        B: *[ if b≤a → skip fi;
      {a≤b}                          {b≤a♡}
      a:=a+1 ]                       b:=b+1 ]
```

Como hemos terminado, ya no es necesario llevar la lista de promesas a cumplir, por lo que ésta fue eliminada.

En este punto tenemos un programa correctamente anotado que cumple con la *computation proper* propuesta inicialmente, sin embargo la corrección obtenida es parcial y poco podemos decir respecto a las propiedades de *liveness* de nuestro multiprograma.
Usualmente la tarea en el tramo final de la derivación será esta: tratar de demostrar el progreso o la terminación de las componentes de nuestro multiprograma dependiendo si las computaciones son todas divergentes o no.

Afortunadamente este caso sigue el patrón de lo expresado en la sección 2.7.3, es decir tiene multicota ($MB : a \leq b + 1 \wedge b \leq a + 1$) y además no se puede producir deadlock total pues su único deadlock potencial es

$$(\textbf{if } a \leq b \rightarrow \textbf{ skip fi},\ \textbf{if } b \leq a \rightarrow\ SKIP\ \textbf{fi})$$

y vemos que sin importar las precondiciones de los skips guardados tenemos

$$a \leq b \vee b \leq a$$

luego podemos concluir que ambas componentes progresan.

Este problema resulta de una particularización de lo que se puede ver en [vdS94, sec.5] o del problema del productor-consumidor derivado en [FvG95b].

### 3.3.2 Búsqueda Lineal Paralela

Ahora es el turno de derivar el algoritmo de la búsqueda lineal paralela que nos sirviera de motivación en el capítulo 1.

El problema consistía en, mediante dos componentes, buscar el valor verdadero (del que conocemos su existencia) de una función $f$ que va de los naturales a los booleanos. Una componente busca en el dominio no-negativo y la otra en el no-positivo.

Demos una especificación inicial que, nuevamente, expresará la computación en sí misma y servirá de punto de partida para nuestra derivación.

```
Pre:  ∃i::f.i ∧ x=0 ∧ y=0
Inv:  0≤x ∧ y≤0
Post:  f.x ∨ f.y

A: do CA → x:=x+1          B: do CB → y:=y-1
   od                         od
   {QA}                       {QB}
```

Promesa: - $f$ no es modificada, $x$ es local a $A$ y $y$ a $B$.

La principal diferencia con la anterior derivación será que en ésta ambas componentes deben terminar, por lo que tenemos una postcondición que debe cumplirse. Además el esquema para demostrar progreso con la multicota y la ausencia de deadlock ya no servirá, por lo que demostraremos terminación con argumentos semi-formales particulares al caso.

Nuestra primera tarea será definir los predicados $CA$, $QA$, $CB$ y $QB$ para que cumplan con la especificación. Por la regla del paralelismo y la de la consecuencia tenemos que

$$QA \wedge QB \Rightarrow f.x \vee f.y$$

mientras que por la regla del constructor de repetición se debe cumplir con

$$\neg CA \Rightarrow QA \wedge \neg CB \Rightarrow QB$$

donde en esta última omitimos el invariante de las repeticiones que está incluído en el invariante global del multiprograma.

Una primera aproximación sería definir $QA$ y $QB$ como $f.x \wedge f.y$, pero esto daría una guarda de la repetición que sería de gruesa granularidad ($CA : CB : \neg f.x \wedge \neg f.y$, y recordemos lo de sentencias 1-acceso), o bien daría una guarda que no aseguraría la terminación del bucle ($CA : \neg f.x$ y $CB : \neg f.y$) debido que la condición de existencia de un valor verdadero no implica que habrá un valor verdadero en el dominio no-negativo y otro en el no-positivo.

Otra posibilidad sería hacer $QA : f.x$ y $QB : f.y$, pero de nuevo esto lleva a guardas $CA : \neg f.x$ y $CB : \neg f.y$ que no aseguran la terminación de ambas componentes.

La tercera opción es definir $QA$=$QB$=$B$ y por lo tanto obtenemos $CA$=$CB$=$\neg B$, con lo que necesitaremos una nueva variable de programa para contener el valor de verdad de $B$ ya que lo necesitamos como guarda en las

repeticiones. Esta se llamará $b$, por lo que las guardas quedarán definidas como $\neg b$ y tendremos que cumplir con $b \Rightarrow f.x \vee f.y$. Para esto último pedimos que la condición sea un invariante global del multiprograma.

Notamos que ante las diversas posibilidades que presentaba la elección de los predicados $CA$, $QA$, $CB$ y $QB$ hemos tomado una desición de diseño considerando cuestiones tales como terminación, granularidad de las sentencias y simetría. Este tipo de consideraciones resultará bastante habitual en todas las derivaciones que realicemos.

Otro hecho notable es la creación de una variable para mantener el valor de un predicado.

Ahora nuestra especificación ya no presenta incógnitas, pero empiezan a aparecer las obligaciones de prueba.

```
Pre:  ∃i::f.i ∧ x=0 ∧ y=0
Inv:  0≤x ∧ y≤0 ∧ b⇒f.x∨f.y?N1
Post:  f.x ∨ f.y
```

```
A: do ¬b → x:=x+1          B: do ¬b → y:=y-1
   od                         od
   {b?N0}                     {b?N0}
```

Promesa: - $f$ no es modificada, $x$ es local a $A$ y $y$ a $B$.

Vemos que ya no indicamos los lugares en donde podemos insertar sentencias. Aunque esto es una buena práctica, pues especifica más concretamente lo que queremos, en la mayoría de los casos se sobreentiende cuáles son los lugares donde se insertará código. En este ejemplo, el lugar será en la sentencia guardada previo al incremento.

El problema está planteado de manera totalmente simétrica, por lo que las notas compartidas serán respecto a la componente $A$. Para obtener la versión de la componente $B$, nos alcanza con substituir las ocurrencias de $x$ por $y$.

Nota 0:

L: implicado por la guarda del comando de repetición

G: por ortogonalidad, pues $b$ no es modificado por la otra componente.

Para que esta aserción no se invalide generamos la promesa que toda asignación a la variable $b$ será con el valor verdadero (*widening*).

Como en la anterior nota sólo produjo que la especificación cambiara en las promesas, sin mostrar la nueva especificación, continuamos directamente con la nota 1.

Nota 1:

I: como nada sabemos de $f.x$ o $f.y$, hacemos válido el invariante pidiendo $\neg b$ en la precondición.

L: los incrementos de $x$ e $y$ pueden cambiar el valor de verdad de esta aserción. Veamos qué precondiciones deben tener para que se mantenga el invariante.

- $\{P\}$`x:=x+1`

  Tenemos que hacer universalmente válido

  $$P \wedge Inv \Rightarrow wlp.(\texttt{x:=x+1}).Inv$$

  que es equivalente a

  $$P \Rightarrow [(b \Rightarrow f.x \vee f.y) \Rightarrow (b \Rightarrow f.(x+1) \vee f.y)]$$

  simplificando un poco esto también resulta equivalente a

  $$P \Rightarrow \neg f.x \vee \neg b \vee f.(x+1) \vee f.y$$

  Como nada podemos decir de $f.(x+1)$, $f.y$ no puede ser referenciado en esta componente y tratar de asegurar $\neg b$ implicaría romper con la promesa hecha anteriormente, la elección resulta sencilla y $P \equiv \neg f.x$.

- {Q}`y:=y-1`

  De manera simétrica

Agregando la promesa de que la parte del invariante probada anteriormente no será invalidada llegamos a la siguiente especificación.

```
Pre:  ∃i::f.i ∧ x=0 ∧ y=0 ∧ ¬b
Inv:  0≤x ∧ y≤0 ∧ b⇒f.x∨f.y♡
Post:  f.x ∨ f.y
A: do ¬b → {¬f.x?N2}          B: do ¬b → {¬f.y?N2}
           x:=x+1                        y:=y-1
   od                            od
   {b♡}                          {b♡}
```

Promesas: - $f$ no es modificada, $x$ es local a $A$ y $y$ a $B$.
- asignaciones a $b$ con verdadero.

Sigamos avanzando en el desarrollo y tratemos de establecer la validez de la nota 2.

Nota 2:

L: usamos un constructor de selección para guardar la sentencia con un predicado igual al de la aserción.

Notamos que si hubiésemos usado la regla del modus ponens habríamos generado una sentencia de selección que actuaría como primitiva de sincronización cuya guarda no puede ser validada por otras componentes, luego si la guarda no resulta verdadera en algún momento, se llega a un estado de deadlock.

Nuestra selección deberá ser entonces exhaustiva, pero por ahora nos contentaremos con esta sola guarda y por lo tanto la selección tendrá posibilidades de bloquearse.

G: por ortogonalidad, asegurada en la primera promesa.

El texto del programa es ahora:

```
Pre:  ∃i::f.i ∧ x=0 ∧ y=0 ∧ ¬b
Inv:  0≤x ∧ y≤0 ∧ b⇒f.x∨f.y
Post:  f.x ∨ f.y
A: do ¬b → if ¬f.x → {¬f.x♡}
                     x:=x+1
           fi
   od
   {b}

B: simétrico
```

Promesas: - $f$ no es modificada, $x$ es local a $A$ y $y$ a $B$.
- asignaciones a $b$ con verdadero.

Estamos en el punto habitual, un multiprograma correcto, pero con poca información con respecto al progreso o terminación. Una rápida inspección nos convence que $\neg b$ es invariante y por lo tanto ninguna de las repeticiones termina.

Será necesario entonces agregar en algún lugar dentro del lazo una asignación de la forma `b:=True`, que tenga como preaserción un predicado tal que se mantengan las promesas y los invariantes. El invariante en peligro es $b \Rightarrow f.x \vee f.y$, y como la precondición más débil nos dice

$$wlp.(\texttt{b:=true}).(b \Rightarrow f.x \vee f.y) \equiv f.x \vee f.y$$

podemos pedir como precondición $f.x$, pues nada podemos decir de $y$ en la componente $A$. Tenemos que probar la nueva nota dentro de este pedazo de código que sabemos que va dentro de la repetición, pero que no conocemos con exactitud su ubicación.

```
{f.x?N3} b:=True
```

Nota 3:
L: guardamos la sentencia dentro de un condicional como en la nota anterior
G: una vez más se da por ortogonalidad.

Resulta entonces que la guarda para este pedazo de código es complementaria a la que teníamos, con lo que juntamos ambas guardas en un mismo constructor de selección y obtenemos la selección no bloqueante que buscábamos.

```
Pre:  ∃i::f.i ∧ x=0 ∧ y=0 ∧ ¬b
Inv:  0≤x ∧ y≤0 ∧ b⇒f.x∨f.y
Post:  f.x ∨ f.y
A: do ¬b → if ¬f.x → {¬f.x}
                     x:=x+1
              □f.x → {f.x♡}
                     b:=true
           fi
   od
   {b}

B: simétrico
```

Promesas: - $f$ no es modificada, $x$ es local a $A$ y $y$ a $B$.
- asignaciones a $b$ con verdadero.

En este punto la terminación es posible, aunque aún tenemos que demostrarla. Vemos que para este caso particular, la terminación es equivalente a que se ejecute la asignación `b:=True`, y esto se produce cuando tenemos $f.x \vee f.y$.

El hecho de que el siguiente predicado es invariante

$$(\forall i : 0 \leq i < x : \neg f.i) \wedge (\forall i : y < i \leq 0 : \neg f.i)$$

junto a que sabemos que existe al menos un valor verdadero (dado en la precondición y asegurado por las promesas), implica que alguna de las dos variables está acotada.

Todo esto más la suposición que el demonio es *weakly fair* nos permite deducir que se llegará a tal cota, $f.x$ o $f.y$ será verdadero, la asignación a $b$ se producirá y entonces nuestro multiprograma terminará.

Finalmente, veamos el código completo despojado de la aserciones intermedias y promesas:

```
Pre:  ∃i::f.i ∧ x=0 ∧ y=0 ∧ ¬b
Inv:  0≤x ∧ y≤0 ∧ b⇒f.x∨f.y
Post:  f.x ∨ f.y
A: do ¬b → if ¬f.x → x:=x+1
              □f.x → b:=true
           fi
   od

B: do ¬b → if ¬f.y → y:=y+1
              □f.y → b:=true
           fi
   od
```

Utilizando el desarrollo estructurado de programas propuesto, llegamos a un texto de programa cuyo funcionamiento y estructura son casi idénticos al de la tercera solución de la sección 1.2, pero ésta tiene el valor agregado de la corrección.

## 3.4 Algunas Técnicas Usuales de Derivación

### 3.4.1 Fortalecimiento de la Anotación

Cuando deseamos establecer la corrección global de una aserción $\{P\}$, la teoría de Owicki-Gries dice que tenemos que buscar todos los patrones $\{Q\}S$ de otras componentes para luego por cada uno de estos demostrar que

$$P \wedge Q \Rightarrow wlp.S.P$$

resulta universalmente válido. Sin embargo alguno puede ser indemostrable, y en tal caso podemos recurrir a la técnica de fortalecer las aserciones $P$ y $Q$ con co-aserciones $C$ y $D$ de forma tal que

$$(P \wedge C) \wedge (Q \wedge D) \Rightarrow wlp.S.P$$

y como esto es equivalente a

$$C \wedge D \Rightarrow (P \wedge Q \Rightarrow wlp.S.P)$$

siempre tendremos alguna solución para esta proposición mediante las incógnitas $C$ y $D$, como por ejemplo haciendo $C \wedge D \equiv false$.

Veamos ahora una derivación cuyo nudo principal se desata utilizando esta técnica. El problema es un ejercicio propuesto en [Moe93] que a su vez es una particularización del problema planteado en [Fei90]

La especificación inicial (o propiedad de computación) a resolver es:

```
P: v:=p;                    Q: v:=q;
   ⋮                           ⋮
   y.p:=(v=p)                  y.q:=(v=q)
   {y.p≡(v=p)?NO}              {y.q≡(v=q)?NO}
```

donde supondremos $p \neq q$. Notamos que no hay lazos, por lo que nuestro multiprograma termina. Además vemos la simetría total entre las componentes, y esto nos permite compartir las notas de la misma manera que en derivaciones anteriores.

Para comenzar el proceso de desarrollo, tomemos la aserción final y veamos qué es necesario para que ésta resulte válida.

Nota 0:
L: Directo por la asignación anterior
G: La única instrucción que puede interferir es la asignación `v:=q` que aún no tiene precondición explícita, es decir ésta es $\{true\}$

- `v:=q`

  $(y.p \equiv (v = p)) \Rightarrow wlp.(\texttt{v:=q}).(y.p \equiv (v = p))$
  $\equiv$ { def. $wlp$ y $p \neq q$ }
  $(y.p \equiv (v = p)) \Rightarrow \neg y.p$
  $\equiv$ { simplificando }
  $v \neq p \vee \neg y.p$

Lamentablemente esto dista de ser un teorema, por lo debemos recurrir al fortalecimiento de las anotaciones.

Adjuntamos $C.p$ a la postaserción final de la componente $P$ y $D.p$ a la preaserción de la primera asignación. La componente $Q$ resulta simétrica.

```
P: {D.p?N1}                    Q: {D.q?N1}
   v:=p;                          v:=q;
   ⋮                              ⋮
   y.p:=(v=p)                     y.q:=(v=q)
   {C.p?N2}{y.p≡(v=p)?N3}         {C.q?N2}{y.q≡(v=q)?N3}
```

**Notación** $\{P\}\{Q\}$ es una manera alternativa de escribir $\{P \wedge Q\}$

Nota 3:
L: De nuevo por la asignación previa.
G:

- `v:=q`

  Ahora con las nuevas coaserciones tenemos que ver:

  $(D.p \wedge C.p \wedge y.p \equiv v = p) \Rightarrow wlp.(\texttt{v:=q}).(y.p \equiv (v = p))$
  $\equiv$ {por cálculo proposicional}
  $D.p \wedge C.p \Rightarrow [y.p \equiv (v = p) \Rightarrow wlp.(\texttt{v:=q}).(y.p \equiv (v = p))]$
  $\equiv$ {utilizando la equivalencia demostrada en la nota 0}
  $D.p \wedge C.p \Rightarrow v \neq p \vee \neg y.p$

  Las expresiones $D.q$ y $C.p$ aún no han sido especificadas y nos permitirán que lo anteriormente expresado sea teorema. Elegimos poner $C.p$ en función del resto y ésto lo obtenemos mediante la siguiente proposición, que es equivalente a las anteriores

  $$C.p \Rightarrow \neg D.q \vee v \neq p \vee \neg y.p$$

  En este punto la elección lógica es hacer $C.p : \neg D.q \vee v \neq p \vee \neg y.p$, pero preferimos fortalecerla un poco a fin de independizarla de $y.p$, pues el valor de ésta ya está restringido en la coaserción de $C.p$ y además $C.p$ es una postaserción de una asignación a $y.p$.

  $$C.p : \neg D.q \vee v \neq p$$

  y con esto logramos la corrección global que no obtuvimos en la nota 0.

Nota 2:
L: La solución, una vez más, está en el *modus ponens* para skip guardado que introduce primitivas de sincronización en el texto de programa. A ésta la pondremos previa a la asignación de $y.p$, pues podemos hacerlo ($C.p$ es independiente de $y.p$) y además nos conviene, dado que ya hemos probado la corrección con este esquema, es decir, con $C.p$ como postaserción de la asignación.

Hacerlo de la otra manera nos llevaría a tener que volver a asegurar $C.p$ previo al skip guardado, y esto implicaría una vuelta a la situación inicial de la nota. Esta recursión sólo puede ser terminada si ponemos la sincronización previa a la asignación.

Notamos que como $D.q$ ahora está dentro de una guarda, debemos convertir esta expresión en variable.

G:

- `v:=q`

  Por *widening*.

  Además notamos que podríamos agregar asignaciones `D.q:=False` sin modificar esta corrección global ni la validez de otra aserción. Enseguida veremos el porqué de esta aclaración.

Nota 1:

L: Pedimos que la preaserción del multiprograma sea $D.p \wedge D.q$

G: Nadie modifica esta variable (ortogonalidad) y queremos que esto siga así pidiendo que sea una promesa.

De momento nuestro texto de programa se encuentra en este estado

```
Pre:  D.p ∧ D.q

P: {D.p♡}                          Q: {D.q♡}
   v:=p;                              v:=q;
   ⋮                                  ⋮
   if ¬D.q ∨ v≠p → skip fi            if ¬D.p ∨ v≠q → skip fi
   y.p:=(v=p)                         y.q:=(v=q)
   {C.p♡}{y.p≡(v=p)♡}                 {C.q♡}{y.q≡(v=q)♡}
```

Promesa: - P no modifica a D.q y Q a D.p.

Todas las aserciones han sido demostradas, por lo que hemos logrado la corrección parcial. Pero, como es costumbre, analizaremos algunos aspectos que escapan a ésta.

Por ejemplo, podemos ver que $D.p$ y $D.q$ son constantemente verdaderas, con lo que las guardas de las sincronizaciones pueden ser simplificadas a $v \neq p$ y $v \neq q$, y toda referencia a $D.p$ y $D.q$ puede ser eliminada. Cuando hacemos esto, nuestro multiprograma despojado de toda aserción es:

```
P: v:=p;                           Q: v:=q;
   if v≠p → skip fi                   if v≠q → skip fi
   y.p:=(v=p)                         y.q:=(v=q)
```

y vemos que si la secuencia de ejecución es $(P, Q, P, P)$, entonces tendremos $v = q$, la computación de la componente $P$ habrá terminado y la segunda no lo podrá hacer pues estará continuamente bloqueada, es decir, nuestro multiprograma adolecerá de *riesgo de deadlock*.

Es ahora cuando la acotación acerca de que la introducción de asignaciones `D.q:=false` dentro de la componente $Q$ empieza a tener sentido. La única restricción respecto al lugar es que la asignación debe estar después de la sentencia `v:=q`, a fin de no invalidar la precondición de ella. Vemos además que la promesa no resulta invalidada.

Para abrir al máximo las posibilidades de progreso, colocamos la asignación tan arriba como se pueda, a fin de que $D.q$ sea falso lo más pronto posible.

```
Pre:  D.p ∧ D.q
P: {D.p}                          Q: {D.q}
   v:=p;                             v:=q;
   D.p:=False;                       D.q:=False;
   ⋮                                 ⋮
   if ¬D.q ∨ v≠p → skip fi           if ¬D.p ∨ v≠q → skip fi
   y.p:=(v=p)                        y.q:=(v=q)
   {C.p}{y.p≡(v=p)}                  {C.q}{y.q≡(v=q)}
```

Promesa: - P no modifica a D.q y Q a D.p.

En esta especificación, el progreso de ambas está asegurado en tanto y en cuanto las asignaciones `D.p:=False` y `D.q:=False` eventualmente se ejecuten y esto sólo depende de que supongamos, nuevamente, *weak fairness* en el demonio.

Resulta un buen ejercicio ver cómo se restringe el paralelismo si elegimos alguna de las otras tres posibilidades para $C.p$, y éste es otro claro ejemplo que justifica la heurística de tratar de tomar siempre la aserción o la guarda más débil posible cada vez que se presenta una elección.

En [Fei90] podemos encontrar, además de una derivación más general de este problema, lo que motiva la *computation proper* que inició nuestra derivación, y esta es:

```
Post:  (Ni::y.i) = 1?
P.i:   y.i:=B.i
```

Podemos ver claramente que este no es un *toy problem*, pues una vez resuelta la derivación, tenemos un algoritmo de elección de componente que por ejemplo puede ser usado para darle un *token* a algún proceso o designar un *master*.

### 3.4.2 Reducción del Grado de Atomicidad

Una de las principales características que buscamos cuando diseñamos programas paralelos es que el grado de paralelismo de estos no esté restringido. En muchos casos podemos derivar multiprogramas correctos de manera rápida, pero que presentan un paralelismo bastante pobre.

Las sentencias atómicas de gruesa granularidad son uno de los factores que evitan un mayor paralelismo, pues éstas requieren de varios pasos para completarse, los cuales deben ser realizados bajo exclusión mutua, y esto detiene el

progreso de los bloques de programa de otras componentes que también participan en ella.

En la sección 3.2.4 vimos una transformación que nos permitía reducir este grado de manera casi gratuita en los constructores de selección/sincronización. Ahora veremos una técnica que permite reducir el grado de atomicidad en las asignaciones múltiples (multiasignaciones) utilizando la transformación de fortalecimiento de guarda vista en la sección 3.2.3.

El siguiente programa, que es una simplificación de otro ejercicio de [Moe93], servirá como un sencillo modelo de muestra para esta técnica.

```
Pre:  ¬c ∧ d
A: *[ if c → skip fi;          B: *[ if d → skip fi;
      c,d:=False,True ]              d,c:=False,True ]
```

Nuestro primer objetivo será anotarlo convenientemente, generar obligaciones de prueba y cumplirlas de manera ordenada utilizando notas.

Lo primero que se puede observar es que $c$ y $d$ están en contrafase, o para expresarlo propiamente, $c \equiv \neg d$ resulta invariante. Lo segundo notable es que cada componente invalida su guarda y hace lo contrario con la otra, y este es un principio muy utilizado en la construcción de programas paralelos [Hoo95, ej.3] [And91, 3.16] [FvG95a, p.12]. En definitiva esto se traduce en que podemos asegurar después de las guardas las condiciones que ellas comprueban.

```
Pre:  ¬c ∧ d
Inv:  c≡ ¬d?N0
A: *[ if c → skip fi;          B: *[ if d → skip fi;
      {c?N1}                         {d?N1}
      c,d:=False,True ]              d,c:=False,True ]
```

Nota 0:
I: Se da ya que $\neg c \wedge d \Rightarrow c \equiv \neg d$
L:

- $\{c\}$`c,d:=False,True`

  Sin importar la precondición, esta sentencia de gruesa granularidad mantiene el invariante, pues

  $$wlp.(\texttt{c,d:=False,True}).(c \equiv \neg d) \equiv true$$

  y $[P \Rightarrow true]$.

- $\{d\}$`d,c:=False,True`

  Idem.

Nota 1:
L: Directamente usando el corolario del *modus ponens* para el skip guardado o viendo que

$$wlp.(\textbf{if } c \to \textbf{skip fi}).c \equiv c \Rightarrow wlp.\textbf{skip}.c \equiv true$$

G: Por *widening*

Estas dos primeras notas no modifican el texto del programa, sólo lo verifican.

El objetivo de ahora en adelante será introducir dos variables ($p$ y $q$) que tomen los roles de $c$ y $d$, de forma tal que por un lado podamos transformar el programa para que $c$ y $d$ resulten auxiliares y por el otro nos permitan escribir asignaciones a $p$ y $q$ que estén desacopladas.

Uno de los pasos para que $c$ y $d$ sean auxiliares es que éstas no formen parte del flujo de control. Si aseguramos que previo a la selección de la componente $A$ resulta válido $p \Rightarrow c$, entonces podríamos utilizar la transformación de fortalecimiento de la guarda y eliminar $c$ de ella. Como siempre, para la componente $B$ el argumento se repite intercambiando $c$ con $d$ y $p$ con $q$.
A fin de hacer válidas estas aserciones, pedimos que se mantenga el invariante global $p \Rightarrow c \wedge q \Rightarrow d$

```
Pre:  ¬c ∧ d
Inv:  c≡ ¬d♡, p⇒c ∧ q⇒d?N2
A: *[ if c → skip fi;          B: *[ if d → skip fi;
      {c♡}                           {d♡}
      c,d:=False,True ]              d,c:=False,True ]
```

Nota 2:
I: Pedimos que en la precondición también valga $\neg p \wedge q$
L:

- $\{c\}$`c,d:=False,True`

  Como

  $$wlp.(\texttt{c,d:=False,True}).(p \Rightarrow c \wedge q \Rightarrow d) \equiv \neg p$$

  y $p$ será la futura guarda, decidimos usar el principio "cada uno falsifica su propia guarda".

  Cambiamos la multiasignación por `c,d,p:=False,True,False` y entonces la tripla se cumple de manera directa.

- $\{d\}$`d,c:=False,True`

  Se prueba de manera similar.

Sin mostrar la especificación aplicamos el fortalecimiento de la guarda ya que $p \Rightarrow c$ es válido en todos lados, y en particular antes de la guarda. Lo mismo para $B$ con su guarda $d$ y su variable asociada $q$.

```
 Pre:  ¬c ∧ d ∧ ¬p ∧ q
 Inv:  c≡¬d, p⇒c ∧ q⇒d♡
A: *[ if p → skip fi;          B: *[ if q → skip fi;
      {p}                           {q}
      c,d,p:=False,True,False ]     d,c,q:=False,True,False ]
```

Tranquilamente podríamos eliminar $c$ y $d$ del programa, ya que éstas forman un conjunto de variables auxiliares, pero antes de hacer esto tenemos que transformar un poco el multiprograma que claramente no progresa en sus dos componentes, luego de haber efectuado las asignaciones múltiples.

La solución para el progreso está en agregar en la componente $A$ una asignación `q:=True` y en $B$ una similar pero para $p$. Resta averiguar el lugar correcto para emplazarlas.

Existen tres posibilidades. Tomamos una decisión de diseño y forzamos la adyacencia de las asignaciones, con lo que sólo resta averiguar el orden relativo entre ellas.

Del invariante podemos deducir $p \Rightarrow \neg q$. Aplicando el *modus ponens* de la lógica, y viendo que $\neg q$ es invariante respecto de las multiasignaciones, y que la falsificación de $p$ no es interferida por la otra componente, llegamos a:

```
 Pre:  ¬c ∧ d ∧ ¬p ∧ q
 Inv:  c≡¬d, p⇒c ∧ q⇒d
A: *[ if p → skip fi;          B: *[ if q → skip fi;
      {p ∧ ¬q}                      {q ∧ ¬p}
      c,d,p:=False,True,False       d,c,q:=False,True,False
      {¬q ∧ ¬p}                     {¬p ∧ ¬q}
   ]                             ]
```

Si ponemos la asignación `q:=True` previa a la multiasignación, podemos demostrar que el estado entre las asignaciones cumple con $p \wedge q$ y esto implica $c \wedge d$, con lo que se invalida el invariante.

Nos quedamos con la última posibilidad, que respeta el invariante y permite el progreso (que esperamos que el lector sea capaz de demostrar).

Si del siguiente texto de programa quitamos las variables auxiliares, finalmente obtenemos el multiprograma con sentencias 1-acceso que resultan las de menor grado de atomicidad que podemos pretender.

```
 Pre:  ¬c ∧ d ∧ ¬p ∧ q
 Inv:  c≡¬d, p⇒c ∧ q⇒d
A: *[ if p → skip fi;          B: *[ if q → skip fi;
      {p ∧ ¬q}                      {q ∧ ¬p}
      c,d,p:=False,True,False;      d,c,q:=False,True,False;
      {¬q ∧ ¬p}                     {¬p ∧ ¬q}
      q:=True ]                     p:=True ]
```

Para finalizar haremos tres acotaciones:

- Podemos ver cómo en este caso la técnica nos permitió decidir cuál era el orden correcto para el desacople de la multiasignación.
- Si quitamos la restricción de la *computation proper* (no explicitada) que marca el uso de dos variables, podríamos haber transformado la guarda de $B$ en $\neg c$, pues $c \equiv \neg d \Rightarrow (\neg c \Rightarrow d)$, con lo que la eliminación de la variable $d$ es posible y el multiprograma sólo tendría asignaciones de 1-acceso.
- Una posible solución al ejercicio original planteado en [Moe93] introduce una variable $x$ y un invariante

$$x = \begin{cases} 1 & si\ c \\ 2 & si\ d \\ 3 & si\ e \end{cases}$$

  Mapeando ésto a nuestra particularización obtenemos una solución más.

# Capítulo 4

# Un Problema Concreto: las Barreras

## 4.1 Introducción

Existen dos problemas que sobresalen entre los de sincronización de procesos paralelos debido a su importancia y a la cantidad de material que se ha escrito sobre ellos. Uno de ellos es la exclusión mutua o región crítica, mientras que el otro es la barrera o sincronización de fase, motivo principal de esta sección.

La barrera es un constructor de sincronización que permite obtener sistemas síncronos en base a sistemas asíncronos. Una barrera se define entre un grupo de procesos que se ejecutan en paralelo (asincrónicamente), y la semántica informal dice que:

- Ninguno de los procesos podrá trasponer la barrera hasta que todos hayan llegado a ella.
- Cuando todos los procesos hayan llegado a ella, ninguno quedará bloqueado.

Estos son los requerimientos de seguridad y progreso respectivamente, que definen una barrera.

Muchos algoritmos paralelos hacen uso de este constructor de sincronización y básicamente todos ellos emplean la barrera para dividir "fases" de la computación, es por ello que en [Mis91] este problema se denomina *sincronización de fase*.

Probablemente los algoritmos más conocidos que hacen uso de las barreras son los *algoritmos paralelos de datos*, donde varios procesos repetidamente realizan fases, consistiendo cada una de éstas en leer desde un almacenamiento común, computar y almacenar el resultado en el mismo almacenamiento compartido, cumpliendo con la condición de que ningún proceso $P$ comenzará la fase $n+1$ hasta que todos hayan completado la fase $n$. Un ejemplo típico de esta familia de algoritmos paralelos, es el de aproximación de la ecuación de LaPlace [Pfi98], también llamado *poisson solver* [Gup89]. Veamos un pseudocódigo para la componente paralela $P_{ij}$:

```
k:=0;
do
  k<K → A[i][j]:=(A[i][j+1]+A[i][j-1]+A[i+1][j]+A[i-1][j])/4;
        barrera();
        k:=k+1
od
```

Este constructor también puede ser utilizado para abortar una computación paralela, pues si alguna de sus componentes aborta, todas las que participan en la barrera no podrán avanzar.

Otro uso puede ser la inicialización de un almacenamiento compartido, donde cada componente en su primera fase inicializa la porción de almacenamiento que le corresponde, y la segunda efectúa la computación teniendo como precondición todo el almacenamiento compartido inicializado. En [vdS94] la barrera es llamada *protocolo de inicialización* pues justamente el algoritmo se piensa para este efecto.

La implementación de éste y otros mecanismos de sincronización puede seguir dos líneas: el bloqueo en colas y el *busy waiting*. La primera se basa en la idea de tratar de aprovechar al máximo los recursos de computación, reasignando a otros procesos el recurso que ha sido dejado de utilizar por uno de ellos que está bloqueado en, por ejemplo, un constructor de sincronización. Este tipo de implementación se basa en la administración de colas de procesos, y carece de sentido en los siguientes casos:

- Cuando el tiempo necesario para administrar las colas es mayor que el tiempo que los procesos permanecen bloqueados.
- Si los recursos de procesamiento no son utilizados para cumplir otras tareas.
- Cuando es imposible o inapropiado en el contexto que se encuentran (por ejemplo dentro de un sistema operativo).

En el algoritmo del *poisson solver* podemos aplicar la primera razón si suponemos que las velocidades de los procesadores no son tan dispares.

Es así que una implementación basada en *busy waiting* tiene sentido, por lo que el resultado de lo que logremos con nuestras derivaciones será código que no sufrirá mayores modificaciones para ser puesto en funcionamiento en máquinas concretas.

## 4.2 Derivación de Gruesa Granularidad

El objetivo inicial es obtener una *computation proper* que exprese nuestros requerimientos de sincronización. Si suponemos $N$ componentes $\{P.i\}_{i=0}^{N-1}$, los conjuntos de variables $\{c.i\}_{i=0}^{N-1}$ y $\{x.i\}_{i=0}^{N-1}$ marcarán respectivamente la entrada y la salida de la barrera. En la precondición pedimos que ambos conjuntos de variables estén inicializados a falso, indicando que todavía no se produjo la entrada ni la salida por parte de ninguna componente. El invariante será nuestra *safety property* indicando que si sale cualquiera es porque todos han entrado.

```
Pre :  ∀i : 0 ≤ i < N : ¬c.i ∧ ¬x.i
Inv :  ∀i : 0 ≤ i < N : x.i ⇒ (∀j : 0 ≤ j < N : c.j)

P.k:  ⋮
      c.k:=True;
       Barrera
      x.k:=True;
      ⋮
```

Nuestra derivación producirá sentencias entre las dos asignaciones que deberán cumplir con una promesa fundamental: toda variable concreta (no fantasma) no será restringida en la precondición. Este pedido resulta crucial, y en ausencia de él, las derivaciones resultan directas y las pruebas de progreso bastante sencillas [Mis91]. En la sección siguiente, cuando derivemos la versión *fine-grained*, esto se pondrá de manifiesto cuando ataquemos el problema de las barreras no iniciales.

Además, el requerimiento es razonable, pues poco sentido tiene derivar un algoritmo para una barrera, que requiere justamente de otra barrera para inicializar el almacenamiento compartido, es decir la precondición.

El punto de partida inicial para la derivación de nuestro multiprograma será:

```
Pre :  ∀i :: ¬c.i ∧ ¬x.i
Inv :  ∀i :: x.i ⇒ (∀j :: c.j)?NO

P.k:   ⋮
       c.k:=True;
       x.k:=True;
       ⋮
```

Promesa: - la pre no contiene variables usadas en el flujo de control.

donde hemos suprimido las cotas de la forma $0 \leq i, j < N$ para economizar notación. Comencemos entonces a cumplir con las obligaciones de prueba.

Nota 0:
I: vale inicialmente.
L:

- `c.k:=True`

  $wlp.(\texttt{c.k:=True}).Inv$
  $\equiv$ { def. de $Inv$ y $wlp$ }
  $(\forall i :: x.i \Rightarrow (\forall j :: c.j))(c.k := true)$
  $\equiv$ { separación de los $c.k$ }
  $((\forall i :: x.i \Rightarrow (\forall j : j \neq k : c.j)) \wedge (\forall i :: x.i \Rightarrow c.k))(c.k := true)$
  $\equiv$ { reemplazo, implicación trivial, neutro de la conjunción }
  $(\forall i :: x.i \Rightarrow (\forall j : j \neq k : c.j))$
  $\Leftarrow$ { fortalecimiento }
  $Inv$

- `x.k:=True`

  $wlp.(\texttt{x.k:=True}).Inv$
  $\equiv$ { def. de $Inv$ y $wlp$, separación de los $x.k$ }
  $((\forall i : i \neq k : x.i \Rightarrow (\forall j :: c.j)) \wedge x.k \Rightarrow (\forall j :: c.j))(c.k := true)$
  $\equiv$ { reemplazo, cálculo proposicional }
  $(\forall i : i \neq k : x.i \Rightarrow (\forall j :: c.j)) \wedge (\forall j :: c.j)$
  $\Leftarrow$ { fortalecimiento }
  $Inv \wedge (\forall j :: c.j)$

Como era de esperar, la entrada a la barrera no pide precondición, pero la salida necesita justamente que todas hayan entrado. La especificación entonces queda:

```
Pre :  ∀i :: ¬c.i ∧ ¬x.i
Inv :  ∀i :: x.i ⇒ (∀j :: c.j)♡

P.k:   ⋮
       c.k:=True;
       {∀j :: c.j?N1}
       x.k:=True;
       ⋮
```

Promesa: - la pre no contiene variables usadas en el flujo de control.

El siguiente paso es asegurar la nota 1. Como $c.k$ es local a $P.k$ entonces dividimos la aserción en dos

$$\{c.k?\texttt{N1.1}\}\{\forall j : j \neq k : c.j?\texttt{N1.2}\}$$

La nota 1.1 sale directamente por la asignación anterior `c.k:=True` y la localidad de esta variable.

Para asegurar la nota 1.2, no es posible asignar las variables $c.j$, pues estaríamos destruyendo la *computation proper*, y tampoco podemos usar sincronización, a menos que tengamos la intención de transformar $\{c.i\}$ en un conjunto de variables concretas. Para hacerla válida, creamos variables que deberán implicar la condición requerida utilizando *modus ponens*.

Nota 1.2:

L: prefijamos la aserción con un par aserción-condicional que introduce el arreglo de booleanos $\{f.i.j\}$. La aserción la elegimos más fuerte de la que necesita el *modus ponens*, pues esto permite derivar de manera más sencilla el algoritmo.

G: por *widening* en $\{c.j\}$ y ortogonalidad en las otras variables.

```
Pre :  ∀i :: ¬c.i ∧ ¬x.i
Inv :  ∀i :: x.i ⇒ (∀j :: c.j)

P.k:   ⋮
       c.k:=True;
       {∀j : j ≠ k : f.k.j ⇒ c.j?N2}
       {(∀j : j ≠ k : f.k.j) ⇒ (∀j : j ≠ k : c.j)}
       if (∀j:j≠k:f.k.j) → skip fi;
       {∀j :: c.j♡}
       x.k:=True;
       ⋮
```

Promesa: - la pre no contiene variables usadas en el flujo de control.

La elección de introducir el arreglo $\{f.i.j\}$ así como de fortalecer la guarda, parecerían ser, en principio, decisiones arbitratrias. Sin embargo estas *decisiones de diseño* se basan en dos ideas: una es que cada proceso consulte su propio conjunto de variables (una fila), y la otra es que se implemente una especie de *contador distribuído*.

Podríamos pedir para cumplir con la nota 2, que ésta fuera un invariante global, pero esto claramente haría que rompiéramos la promesa. Nada podemos suponer acerca del arreglo $\{f.i.j\}$ al iniciar el multiprograma.

Nota 2:

L: no podemos asignar los $\{c.j\}$ para no romper con la *computation proper*, por lo que asignamos la fila $k$ de $\{f.i.j\}$ con falsos. Hacemos esta multiasignación atómica con la de $c.k$, a fin de minimizar los lugares donde podemos tener interferencia. Esta última será una estrategia general para nuestra derivación de gruesa granularidad, en pos de generar un multiprograma que resulte sencillo para demostrar progreso.

G: por *widening* en los $\{c.j\}_{j\neq k}$ y ortogonalidad en los $\{f.k.j\}_{j\neq k}$.

```
Pre :  ∀i :: ¬c.i ∧ ¬x.i
Inv :  ∀i :: x.i ⇒ (∀j :: c.j)

P.k:   ⋮
       c.k,{f.k.j}_{j≠k}:=True,{False}[1] ;
       {∀j : j ≠ k : f.k.j ⇒ c.j♡}
       if (∀j:j≠k:f.k.j) → skip fi;
       {∀j :: c.j}
       x.k:=True;
       ⋮
```

Promesa: - la pre no contiene variables usadas en el flujo de control.

Hemos terminado con las obligaciones de prueba y una vez más tenemos corrección parcial sin progreso porque, aunque se cumple con el principio "cada una falsifica su propia guarda", ninguna componente las hace verdaderas. Entonces nuestra derivación es equivalente al programa correctamente anotado **if** `False` $\rightarrow$ `S` **fi**$\{$`Q`$\}$.

La idea fundamental para lograr progreso es asignar la mayor cantidad de elementos de $\{f.i.j\}$ con verdadero sin modificar la corrección de las aserciones. Resulta claro que la única aserción que puede resultar invalidada es la que aseguramos en la nota 2. La estrategia entonces será asignar los $\{f.i.k\}_{i\neq k}$ junto con la asignación a $c.k$ o por debajo de ella donde también vale la aserción $\{c.k\}$.

Elegimos ponerla junto con la multiasignación donde está $c.k$ y en la asignación de $x.k$. El multiprograma final de gruesa granularidad es el siguiente:

[1] Notación alternativa para
`<c.k:=True; f.k.0:=True; ...; f.k.k-1:=True; f.k.k+1:=True; ...f.k.N-1:=True>`.

```
Pre :  ∀i :: ¬c.i ∧ ¬x.i
Inv :  ∀i :: x.i ⇒ (∀j :: c.j)

P.k:   ⋮
       c.k,{f.k.j}_{j≠k},{f.i.k}_{i≠k}:=True,{False},{True};
       {∀j : j ≠ k : f.k.j ⇒ c.j}
       if (∀j:j≠k:f.k.j) → skip fi;
       {∀j :: c.j}
       x.k,{f.i.k}_{i≠k}:=True,{True};
       ⋮
```

### 4.2.1 Prueba de Progreso

Al instanciar el algoritmo anterior para una cantidad menor de procesos ($N$=2,3), y luego de seguir algunas de sus computaciones posibles, nos convencemos de que nuestro multiprograma final cumple con la *liveness property*, es decir que si todos han llegado a la barrera ($\forall i :: c.i$), entonces eventualmente todos pasarán ($\forall i :: x.i$).

En esta sección propondremos un invariante $InvP$ y demostraremos que nuestro multiprograma lo cumple. De este invariante deduciremos que la propiedad de progreso requerida se cumple en nuestro multiprograma si suponemos *weak fairness*.

Demos algunas definiciones iniciales:

$$
\begin{array}{rcl}
\texttt{Entry.k} & : & \texttt{c.k,\{f.k.j'\}}_{j'\neq k}\texttt{,\{f.i'.k\}}_{i'\neq k}\texttt{:=True,\{False\},\{True\}} \\
\texttt{Exit.k} & : & \texttt{x.k,\{f.i'.k\}}_{i'\neq k}\texttt{:=True,\{True\}} \\
c & : & (\sharp i' :: c.i') \\
x & : & (\sharp i' :: x.i') \\
InvP & : & (\forall n' : N-c \leq n' < N-x : (\exists i' : c.i' \wedge \neg x.i' : (\sharp j' : j' \neq i' : \neg f.i'.j') = n')) \\
InvP_1(n) & : & (\exists i' : c.i' \wedge \neg x.i' : (\sharp j' : j' \neq i' : \neg f.i'.j') = n)
\end{array}
$$

Para que los predicados sean más compactos, hemos eliminado todas las cotas que se presuponen, como por ejemplo $0 \leq i, j, k, i', j', k' < N$.

Probemos ahora algunos lemas sencillos que luego utilizaremos en la demostración de invariancia de $InvP$.

El primero es una consecuencia de la propiedad de seguridad $Inv$ y nos dice que si alguna componente no ha entrado, entonces ninguna puede haber salido.

**Lema 4.1** *En el multiprograma se da que* $\neg c.k \Rightarrow (\forall j :: \neg x.j)$.

**Demostración:** como se cumple el invariante tenemos

$$
\begin{array}{ll}
 & (\forall i :: x.i \Rightarrow (\forall j :: c.j)) \\
\equiv & \{\ \wedge\text{-introducción en cuantificador}\ \} \\
 & (\forall i :: (\forall j :: x.i \Rightarrow c.j)) \\
\equiv & \{\ \text{cálculo proposicional}\ \} \\
 & (\forall i :: (\forall j :: \neg c.j \Rightarrow \neg x.i))
\end{array}
$$

$\equiv$ { intercambio de cuantificadores, particularización }
$(\forall i :: \neg c.k \Rightarrow \neg x.i)$
$\equiv$ { $\vee$-extracción en cuantificador }
$\neg c.k \Rightarrow (\forall j :: \neg x.j)$

□

Los dos lemas que siguen, dan condiciones sobre algunos elementos de $\{f.i.j\}$ a partir de $\{c.i\}$ y $\{x.i\}$.

**Lema 4.2** *En el multiprograma, si* $i' \neq i$ *entonces* $\neg c.i \wedge c.i' \Rightarrow \neg f.i'.i \wedge f.i.i'$.

**Demostración:** Por la forma de las multiasignaciones `Entry.k` y `Exit.k`, podemos afirmar que los elementos $f.i.i'$ y $f.i'.i$ sólo son modificados por $P.i$ y $P.i'$.

Del lema anterior y la premisa $\neg c.i$, podemos concluir $(\forall j :: \neg x.j)$ y en particular podemos afirmar $\neg x.i \wedge \neg x.i'$.

Uniendo las condiciones anteriores podemos concluir $\neg c.i \wedge c.i' \wedge \neg x.i \wedge \neg x.i'$, es decir que de las cuatro posibles multiasignaciones de $P.i$ y $P.i'$ que modifican los $f.i.i'$ y $f.i'.i$, solamente `Entry.i'` se ejecutó, y esto finalmente implica $\neg f.i'.i \wedge f.i.i'$. □

**Lema 4.3** *En el multiprograma, si* $i' \neq i$ *entonces*

$$(\forall j :: c.j) \wedge \neg x.i \wedge \neg x.i' \Rightarrow (\neg f.i'.i \equiv f.i.i').$$

**Demostración:** Nuevamente recurrimos al hecho que $f.i.i'$ y $f.i'.i$ sólo son modificados por $P.i$ y $P.i'$. De las premisas tenemos $c.i \wedge c.i' \wedge \neg x.i \wedge \neg x.i'$, es decir que de las cuatro multiasignaciones de $P.i$ y $P.i'$ que modifican las variables, sólo `Entry.i` y `Entry.i'` se ejecutaron. La primera establece $\neg f.i.i' \wedge f.i'.i$ mientras que la segunda $\neg f.i'.i \wedge f.i.i'$, y de esto podemos concluir que $\neg f.i'.i \equiv f.i.i'$ □

Probemos ahora un lema sencillo que relaciona algunos aspectos de la pre y post condición de las multiasignaciones.

**Lema 4.4** *Dados* $X$ *y* $C$ *tales que* $0 \leq C, X < N$, *entonces se cumplen las siguientes triplas de Hoare*

- $\{c = C \wedge \neg c.k\}$ ***Entry.k*** $\{c = C + 1\}$
- $\{x = X \wedge \neg x.k\}$ ***Exit.k*** $\{x = X + 1\}$

**Demostración:** Veamos solamente que la primer tripla es teorema, para la segunda la demostración es completamente similar.

$wlp.(Entry.k).(c = C+1)$
$\equiv$ { def. $wlp$ y $c$ }
$(\sharp i' : 0 \leq i' < N : c.i') = C+1)(Entry.k)$
$\equiv$ { partición del $i' = k$ en $\sharp$ }
$(\sharp i' : 0 \leq i' < N \wedge i' \neq k : c.i') + (c.k \rightarrow 1 \Box \neg c.k \rightarrow 0) = C+1)(Entry.k)$

$\equiv$ { def. sustitución, `Entry.k` sólo tiene a `c.k:=True` que modifica }
$(\sharp i' : 0 \leq i' < N \wedge i' \neq k : c.i') + 1 = C+1$
$\equiv$ { cálculo }
$(\sharp i' : 0 \leq i' < N \wedge i' \neq k : c.i') = C$
$\equiv$ { cálculo }
$(\sharp i' : 0 \leq i' < N \wedge i' \neq k : c.i') + 0 = C$
$\Leftarrow$ { fortalecimiento }
$\neg c.k \wedge (\sharp i' : 0 \leq i' < N \wedge i' \neq k : c.i') + 0 = C$
$\equiv$ { def. selección y $\neg c.k$ }
$\neg c.k \wedge (\sharp i' : 0 \leq i' < N \wedge i' \neq k : c.i') + (c.k \rightarrow 1 \Box \neg c.k \rightarrow 0) = C$
$\equiv$ { def. $\sharp$ }
$\neg c.k \wedge (\sharp i' : 0 \leq i' < N : c.i') = C$
$\equiv$ { def. $c$ }
$\neg c.k \wedge c = C$

Resumiendo tenemos $\neg c.k \wedge c = C \Rightarrow wlp.(Entry.k).(c = C+1)$, y esto es equivalente a decir que la tripla es válida.

□

Probemos ahora dos lemas que son la base para demostrar la invariancia de $InvP$ respecto a `Entry.k` y `Exit.k`.

**Lema 4.5** *Dado $C < N$ se cumple que:*

- $\forall n : N-(C+1) \leq n < N-1 : \{c = C \wedge \neg c.k \wedge InvP\}$ ***Entry.k*** $\{InvP_1(n)\}$
- $\{c = C \wedge \neg c.k \wedge InvP\}$ ***Entry.k*** $\{InvP_1(N-1)\}$

**Demostración:** Para el primero supongamos $n \in [N-(C+1), N-1)$

$c = C \wedge \neg c.k \wedge InvP$
$\equiv$ { lema 4.1 y def. de $c$, $\neg c.k \Rightarrow x = 0$ }
$c = C \wedge \neg c.k \wedge x = 0 \wedge InvP$
$\Rightarrow$ { def. de $InvP$, reemplazo de $c$ y $x$, eliminación de conjunciones }
$\neg c.k \wedge (\forall n' : N-C \leq n' < N : (\exists i' : c.i' \wedge \neg x.i' : (\sharp j' : j' \neq i' : \neg f.i'.j') = n'))$
$\Rightarrow$ { particularización $n'$ a $n+1$ (cumple cotas), elección de $i'$ en $i$ (implica $i \neq k$) }
$\neg c.k \wedge c.i \wedge \neg x.i \wedge (\sharp j' : j' \neq i : \neg f.i.j') = n+1$
$\equiv$ { partición del $j' = k$ en $\sharp$ }
$\neg c.k \wedge c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i, k\} : \neg f.i.j') + (\neg f.i.k \rightarrow 1 \Box f.i.k \rightarrow 0) = n+1$
$\Rightarrow$ { lema 4.2 y $\neg c.k \wedge \neg c.i$ implica $\neg f.i.k$, def. de selección}
$\neg c.k \wedge c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i, k\} : \neg f.i.j') + 1 = n+1$
$\Rightarrow$ { cálculo y eliminación de conjunción }
$c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i, k\} : \neg f.i.j') = n$
$\equiv$ { cálculo }
$c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i, k\} : \neg f.i.j') + 0 = n$
$\equiv$ { def. selección, sustitución, `Entry.k` asigna `f.i.k:=True` }
$c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i, k\} : \neg f.i.j') + (\neg f.i.k \rightarrow 1 \Box f.i.k \rightarrow 0)(Entry.k) = n$
$\equiv$ { sustitución, $c.i$, $x.i$ y $f.i.j$ con $j \neq k$ no son tocados por `Entry.k` }
$(c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i, k\} : \neg f.i.j') + (\neg f.i.k \rightarrow 1 \Box f.i.k \rightarrow 0) = n)(Entry.k)$
$\equiv$ { def. de $\sharp$ }
$(c.i \wedge \neg x.i \wedge (\sharp j' : j' \neq i : \neg f.i.j') = n)(Entry.k)$
$\Rightarrow$ { instanciación }
$(\exists i' : c.i' \wedge \neg x.i' \wedge (\sharp j' : j' \neq i' : \neg f.i'.j') = n)(Entry.k)$
$\equiv$ { def. $wlp$ e $InvP_1$ }
$wlp.(\texttt{Entry.k}).InvP_1(n)$

Entonces tenemos que

$$\forall n : N-(C+1) \leq n < N-1 : c = C \wedge \neg c.k \wedge InvP \Rightarrow wlp.(\texttt{Entry.k}).InvP_1(n)$$

La demostración del segundo casi no necesita de la precondición, pues justamente `Entry.k` asigna la fila $k$ con $N-1$ valores falsos.

$c = C \wedge \neg c.k \wedge InvP$
$\Rightarrow$ { lema 4.1 implica $\neg c.k \Rightarrow \neg x.k$, borrado de conjunciones }
$\neg x.k$
$\equiv$ { neutro de conjunción, cálculo }
$\neg x.k \wedge N-1 = N-1$
$\equiv$ { $|\{j : j \neq k\}| = N-1$ }
$\neg x.k \wedge (\sharp j' : j' \neq k : true) = N-1$
$\equiv$ { neutro conjunción, sustitución $Entry.k$ hace $c.k$ y $(\forall j : j \neq k : \neg f.k.j)$ y no toca $x.k$ }
$(c.k \wedge \neg x.k \wedge (\sharp j' : j' \neq k : \neg f.k.j') = N-1)(Entry.k)$
$\Rightarrow$ { instanciación }
$(\exists i' : c.i' \wedge \neg x.i' \wedge (\sharp j' : j' \neq i' : \neg f.i'.j') = N-1)(Entry.k)$
$\equiv$ { def. de $wlp$ e $InvP_1$ }
$wlp.(\texttt{Entry.k}).InvP_1(N-1)$

Finalmente tenemos $c = C \wedge \neg c.k \wedge InvP \Rightarrow wlp.(\texttt{Entry.k}).InvP_1(N-1)$ y esto demuestra la segunda tripla de Hoare.
□

El siguiente lema establece postcondiciones para `Exit.k`.

**Lema 4.6** *Sea $X < N$, entonces se da:*

$$\forall n : 0 \leq n < N-(X+1) :$$
$$\{x = X \wedge (\forall j' :: c.j') \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge InvP\}\texttt{Exit.k}\{InvP_1(n)\}$$

**Demostración:** Sea $n \in [0, N-(X+1))$

$x = X \wedge (\forall j' :: c.j') \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge InvP$
$\equiv$ { def. de c y $\sharp$ }
$x = X \wedge c = N \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge InvP$
$\equiv$ { def. de $InvP$ }
$x = X \wedge c = N \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge$
$(\forall n' : N-c \leq n' < N-x : (\exists i' : c.i' \wedge \neg x.i' : (\sharp j' : j' \neq i' : \neg f.i'.j') = n'))$
$\equiv$ { reemplazo de $x$ y $c$, cálculo, eliminación de conjunción }
$c = N \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge$
$(\forall n' : 0 \leq n' < N-X : (\exists i' : c.i' \wedge \neg x.i' : (\sharp j' : j' \neq i' : \neg f.i'.j') = n'))$
$\Rightarrow$ { particularización de $n'$ a $n+1$, elección de $i'$ en $i$ (implica $i \neq k$),
$n+1$ está en rango }
$c = N \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge c.i \wedge \neg x.i \wedge (\sharp j' : j' \neq i : \neg f.i.j') = n+1$
$\Rightarrow$ { particularización de $j'$ en $i$, $i \neq k$ }
$c = N \wedge f.k.i \wedge \neg x.k \wedge c.i \wedge \neg x.i \wedge (\sharp j' : j' \neq i : \neg f.i.j') = n+1$
$\Rightarrow$ { lema 4.3 y $f.k.i$ tenemos $\neg f.i.k$, eliminación de conjunciones }
$\neg f.i.k \wedge c.i \wedge \neg x.i \wedge (\sharp j' : j' \neq i : \neg f.i.j') = n+1$
$\equiv$ { partición del $j' = k$ en $\sharp$ }
$\neg f.i.k \wedge c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i, k\} : \neg f.i.j') + (\neg f.i.k \rightarrow 1 \Box f.i.k \rightarrow 0) = n+1$
$\Rightarrow$ { def. de selección, eliminación de conjunción }
$c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i, k\} : \neg f.i.j') + 1 = n+1$
$\equiv$ { cálculo }

$c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i,k\} : \neg f.i.j') = n$
$\equiv$ { cálculo }
$c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i,k\} : \neg f.i.j') + 0 = n$
$\equiv$ { sustitución, def. selección, `Exit.k` asigna `f.i.k:=True` }
$c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i,k\} : \neg f.i.j') + (\neg f.i.k \rightarrow 1 \Box f.i.k \rightarrow 0)(Exit.k) = n$
$\equiv$ { sustitución, $c.i$, $x.i$ y $f.i.j$ con $j \neq k$ no son tocados por `Exit.k` }
$(c.i \wedge \neg x.i \wedge (\sharp j' : j' \notin \{i,k\} : \neg f.i.j') + (\neg f.i.k \rightarrow 1 \Box f.i.k \rightarrow 0) = n)(Exit.k)$
$\equiv$ { def. de $\sharp$ }
$(c.i \wedge \neg x.i \wedge (\sharp j' : j' \neq i : \neg f.i.j') = n)(Exit.k)$
$\Rightarrow$ { instanciación }
$(\exists i' : c.i' \wedge x.i' \wedge (\sharp j' : j' \neq i' : \neg f.i'.j') = n)(Exit.k)$
$\equiv$ { def. $wlp$ e $InvP_1$ }
$wlp.(\texttt{Exit.k}).InvP_1(n)$

□

Ya hemos reunido todos los elementos necesarios para demostrar todas las notas que serán agregadas. Veamos nuestro último multiprograma aumentado en cantidad de aserciones.

`Pre` : $\forall i :: \neg c.i \wedge \neg x.i$
`Inv` : $\{\forall i :: x.i \Rightarrow (\forall j :: c.j)\}$ $\{InvP$`?N3`$\}$

`P.k:` $\vdots$
$\{\neg c.k$`?N4`$\}$
`Entry.k;`
$\{\forall j : j \neq k : f.k.j \Rightarrow c.j\}$
**if** `(∀j:j≠k:f.k.j)` $\rightarrow$ **skip fi;**
$\{\forall j :: c.j\}$ $\{\forall j : j \neq k : f.k.j$`?N5`$\}$ $\{\neg x.k$`?N6`$\}$
`Exit.k;`
$\vdots$

Nota 4:
L: implicada por la precondición.
G: ortogonalidad, $c.k$ es local a $P.k$.

Nota 6:
L: implicada por la precondición y la ortogonalidad de las dos sentencias anteriores.
G: ortogonalidad, $x.k$ es local a $P.k$.

Nota 5:
L: por *modus ponens* para skip guardado.
G: Suponemos $i \neq k$

- $\{\neg c.i\}$`Entry.i`: por regla de las aserciones disjuntas.
- `Exit.i`: por *widening*, sólo hay asignaciones a verdadero.

Probemos finalmente la nota más importante

Nota 3:

I: la precondición implica $c = 0 \wedge x = 0$ y esto hace que el invariante se cumpla de manera trivial.

L: Veamos que permanece invariante respecto a cada una de las instrucciones.

- $\{c = C \wedge \neg c.k\}$`Entry.i`

  Por ambos puntos del lema 4.5 tenemos

  $$\forall n : N-(C+1) \leq n < N : \{c = C \wedge \neg c.k \wedge InvP\}\texttt{Entry.k}\{InvP_1(n)\}$$

  Usando conjuntividad para las ternas de Hoare llegamos a

  $$\{c = C \wedge \neg c.k \wedge InvP\}\texttt{Entry.k}\{\forall n' : N-(C+1) \leq n' < N : InvP_1(n')\}$$

  Ahora por el lema 4.4, la regla de consecuencia y conjuntividad

  $$\{c = C \wedge \neg c.k \wedge InvP\}\texttt{Entry.k}\{\forall n' : N-c \leq n' < N : InvP_1(n')\}$$

  Como $0 \leq x \leq N$

  $$(\forall n' : N-c \leq n' < N : InvP_1(n')) \Rightarrow (\forall n' : N-c \leq n' < N-x : InvP_1(n'))$$

  finalmente por regla de consecuencia

  $$\{c = C \wedge \neg c.k \wedge InvP\}\texttt{Entry.k}\{InvP\}$$

- **if** `(∀j:j≠k:f.k.j)` $\rightarrow$ **skip fi**

  $InvP \Rightarrow wlp.(\textbf{if } \texttt{(∀j:j≠k:f.k.j)} \rightarrow \textbf{skip fi}).InvP$
  $\equiv$ { def. $wlp$ }
  $InvP \Rightarrow ((\forall j : j \neq k : f.k.j) \Rightarrow InvP)$
  $\equiv$ { cálculo proposicional }
  $true$

- $\{x = X \wedge (\forall j' :: c.j') \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k\}$`Exit.i`

  Por lema 4.6 y conjuntividad tenemos

  $$\{x = X \wedge (\forall j' :: c.j') \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge InvP\}\texttt{Exit.i}$$
  $$\{\forall n' : 0 \leq n' < N-(X+1) : InvP_1(n')\}$$

  Además por lema 4.4, regla de consecuencia y conjuntividad

  $$\{x = X \wedge (\forall j' :: c.j') \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge InvP\}\texttt{Exit.i}$$
  $$\{\forall n' : 0 \leq n' < N-x) : InvP_1(n')\}$$

  Claramente la aserción $\{\forall j' :: c.j'\}$ no es invalidada por `Exit.k`, luego en la postcondición tenemos $c = N$. Reescribiendo obtenemos:

  $$\{x = X \wedge (\forall j' :: c.j') \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge InvP\}\texttt{Exit.i}$$
  $$\{\forall n' : N-c \leq n' < N-x) : InvP_1(n')\}$$

  Con lo que finalmente llegamos a

  $$\{x = X \wedge (\forall j' :: c.j') \wedge (\forall j' : j' \neq k : f.k.j') \wedge \neg x.k \wedge InvP\}\texttt{Exit.i}\{InvP\}$$

Después de este esfuerzo, el paso lógico es ver porqué este invariante nos demuestra que el algoritmo cumple con los requerimientos de progreso.

Antes de esto, demos una idea intuitiva del significado de $InvP$.

El invariante nos está diciendo que para todos los $n$ que estén en $[N-c, N-x)$ $(\forall \mathbf{n'} : \mathbf{N-c} \leq \mathbf{n'} < \mathbf{N-x})$, habrá al menos una fila $(\exists \mathbf{i'})$ de la matriz $\{f.i.j\}$ correspondiente a la condición del skip guardado de un proceso que entró a la barrera pero que aun no salió $(\mathbf{c.i'} \wedge \neg \mathbf{x.i'})$ con $n'$ cantidad de elementos en falso $(\sharp \mathbf{j'} : \mathbf{j'} \neq \mathbf{i'} : \neg \mathbf{f.i'.j'}) = \mathbf{n'})$

Claramente nuestro interés está en las filas $i$ que tienen 0 elementos en falso, es decir las que cumplen $(\forall j' : j' \neq i : f.i.j')$, pues ésta es la condición de paso para los constructores de sincronización.

Sin embargo el invariante se muestra más poderoso de lo que parece, pues en una sola expresión reúne las propiedades deseadas de seguridad y progreso. Describamos el invariante dividiendo en casos los posibles valores de $c$ y $x$:

- $c < N$: no todos los procesos han entrado a la barrera. La cota de $n'$ es $1 \leq N-c \leq n' < N-x$, y esto implica que no habrá filas con todos sus valores en verdadero, lo cual implica que ninguno pasará (*safety property*).
- $c = N \wedge x < N$: todos han llegado, y la cota de $n'$ es $0 \leq n' < N-x$ y como $N-x > 0$, entonces existe una fila $i$ con ningún elemento falso, que ha llegado pero todavía no ha salido. Suponiendo *weak fairness* nos alcanza para ver que el proceso $i$ eventualmente pasará la barrera (*liveness property*).

La motivación inicial del problema fue extender el protocolo de inicialización presentado en [vdS94, sec.9] para una cantidad cualquiera de procesos. En el trabajo de van der Sommen, la derivación se realizó en dos partes, la primera dirigida por un invariante que capturaba la propiedad de seguridad, mientras que en la segunda la derivación resultaba guiada por un invariante que, según el autor, atrapaba la noción de progreso[2].

Nuestro acercamiento fue más convencional, en el sentido que derivamos un multiprograma usando como motor la propiedad de seguridad, para luego modificar el texto de programa y probar la corrección de un invariante que, para nosotros, implicaba progreso.

Aunque los *zeno predicates/invariants* resultan más coherentes con la técnica de *derivación*, usualmente resulta difícil escribir estos predicados que indiquen progreso y además nos permitan derivar el multiprograma. Y esta última fue la razón por la que el progreso en nuestro multiprograma fue resultado de una prueba y no de una derivación.

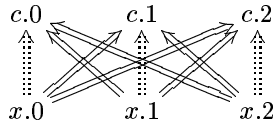
## 4.3 Derivación de Fina Granularidad

El algoritmo que fue derivado en la sección anterior es, sin ningún lugar a dudas, de muy gruesa granularidad. Hay sólo tres instrucciones, y todas acceden ya sea para leer o escribir a $\mathcal{O}(N)$ lugares compartidos de la memoria.

[2] *zeno invariants*

De más está decir que tal algoritmo presenta poca utilidad práctica, pues cada "instrucción" debería estar protegida por una sección crítica, es decir la otra primitiva importante de sincronización, y esto llevaría a un degradamiento total de la performance de algoritmos paralelos de fina granularidad que usen esta barrera, como por ejemplo el *poisson solver* que realiza solamente unos cálculos simples para luego sincronizarse.

Probablemente las multiasignaciones del algoritmo de gruesa granularidad puedan ser desacopladas (complicando la prueba de progreso), pero está claro que la guarda seguirá siendo de $\mathcal{O}(N)$ consultas a memoria compartida, pues el mismo invariante de seguridad nos está diciendo que el $x.i$ sólo debe ser asignado cuando todos los $\{c.j\}$ son verdaderos. El diagrama de dependencia para $N = 3$ sería el siguiente:




donde las implicaciones punteadas están indicando dependencias que se cumplen debido a la topología del programa.

Una posible solución al problema es derivar lo que se conoce como *algoritmo de contador compartido* o *barrera centralizada*. Marcaremos los puntos más importantes de la derivación, el resto puede ser completado por el lector a manera de ejercicio.

Durante la derivación de la barrera de gruesa granularidad, cuando nuestra especificación llegó al siguiente punto

```
Pre :  ∀i :: ¬c.i ∧ ¬x.i
Inv :  ∀i :: x.i ⇒ (∀j :: c.j)♡

P.k:   ⋮
       c.k:=True;
       {∀j :: c.j?N1}
       x.k:=True;
       ⋮
```

Decidimos entonces, crear el arreglo $\{f.i.j\}$ que seguiría los valores de $\{c.i\}$. Podemos tomar otro camino y crear una variable entera $n$ que mantenga la cantidad de valores *true* que tiene el conjunto $\{c.i\}$. Utilizando *modus ponens* pero ahora sobre un nuevo invariante, llegamos a:

```
Pre :  ∀i :: ¬c.i ∧ ¬x.i
Inv :  {∀i :: x.i ⇒ (∀j :: c.j)}{n = (♯i :: c.i)?N2}

P.k:   ⋮
       c.k:=True;
       if n=N → skip fi;
       {∀j :: c.j♡}
       x.k:=True;
       ⋮
```

Unos pasos más de derivación y obtenemos el algoritmo 3.12 de [And91].

A pesar de que el grado de atomicidad se redujo enormemente, pues la instrucción atómica de más accesos a memoria compartida es el incremento, este algoritmo presenta dos problemas:

- La introducción del invariante que relaciona $n$ con $\{c.i\}$ implica que en la precondición debemos limitar el espacio de estados de una variable que participa en el flujo de control, exigiendo que ésta sea cero, con lo cual rompemos con la promesa de la derivación anterior.
- El algoritmo introduce *hot spots* en la memoria, es decir un lugar muy consultado y modificado, lo cual genera en las arquitecturas paralelas[3] con memoria compartida sin coherencia de caché y difusión, una inaceptable cantidad de tráfico en los buses de memoria que degrada la performance de la arquitectura [YTL86, Gup89, MCS91], además de la lógica serialización que sufren los procesos al acceder a una porción de memoria compartida.

De todas maneras, este algoritmo compensa estas desventajas con su sencillez en la implementación así como en las pruebas de seguridad y progreso.

Estamos buscando dos cosas: fino grado de atomicidad y distribución de los *hot spots*. Muchos algoritmos de barrera cumplen con ambos requisitos, y, en rasgos generales, muestran una estructura arbórea en la forma de sincronizarse. Algunos algoritmos de este tipo son: árbol combinado, mariposa, diseminación, torneo, árbol MCS, entre otros. En [MCS91, GV93] podemos encontrar análisis comparativos de todos estos algoritmos.

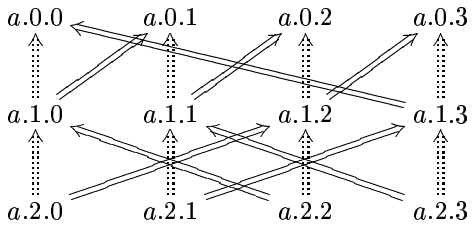
El camino que tomamos fue el de reexpresar la propiedad de computación inicial, siguiendo la forma básica de la *barrera de diseminación*, para luego aplicar la teoría de Owicki-Gries y derivar un algoritmo concreto que cumpla con los dos requisitos planteados.

En rasgos generales, la barrera de diseminación, que es una mejora a la barrera mariposa, consiste dividir la sincronización de cada proceso en sucesivas etapas, donde en la etapa $i$ el proceso $P_j$ se sincroniza con el vecino $P_{[j+2^i]_N}$. Cuando todos los procesos finalizan sus etapas, cada uno estará sincronizado directa o indirectamente con todos los otros. La cantidad de etapas necesarias

[3]La arquitectura paralela subyacente no es un parámetro menor en la elección de los algoritmos de sincronización basados en *busy waiting* [GV93].

es de $\lceil log_2 N \rceil$ y su nombre se debe a que sigue el patrón de la técnica de diseminación de información entre $N$ procesos.

Si el paso por la etapa $i$ del proceso $j$ está marcado por la variable $a.i.j$, entonces el patrón de dependencia para $N = 4$ es el siguiente:




donde de nuevo las implicaciones punteadas indican la dependencia topológica.

Escribamos entonces la *computation proper* que especifica de manera concreta todo lo que hemos dicho acerca de la barrera de diseminación.

```
Pre :  (∀i : 0 ≤ i ≤ l : (∀j : 0 ≤ j < N : ¬a.i.j))
Inv :  (∀i : 0 ≤ i < l : (∀j : 0 ≤ j < N :
         a.(i + 1).j ⇒ a.i.D_i(j) ∧ a.(i + 1).j ⇒ a.i.j))?N0

P.j:  ⋮
      for i:= 0 to l
          a.i.j:=True
      rof;
      ⋮
```

donde $N$ es la cantidad de procesos y

$$\begin{aligned} l &: \lceil log_2 N \rceil \\ D_i(j) &: [j]_N \oplus [2^i]_N \end{aligned}$$

Como $N$ y por lo tanto $l$ están estáticamente definidos, la iteración es simplemente una notación compacta del *loop unrolling*.

Antes de comenzar la derivación, veamos algunos lemas importantes.

El primero habla de la relación entre el invariante dado en la versión de gruesa granularidad y el nuevo invariante. Como el primero representa de manera general la propiedad de seguridad que se busca en las barreras, probaremos que el nuevo invariante resulta más fuerte. Utilizando el teorema de debilitamiento de aserción (3.2.6), concluimos que resolviendo este problema, también estamos resolviendo el de la barrera.

**Lema 4.7 (Dissemination⇒Barrier)** *Si tenemos $N$, $l$ y $D$ definidos como antes, entonces: $Inv \Rightarrow (\forall j : 0 \leq j < N : (\forall \tilde{j} : 0 \leq \tilde{j} < N : a.l.j \Rightarrow a.0.\tilde{j}))$*

**Demostración:** Dados $j$ y $\tilde{j}$ tal que $0 \leq j, \tilde{j} \leq N$ podemos definir $k$ de la siguiente manera:

$$k = \begin{cases} \tilde{j} - j & \text{si } j \leq \tilde{j} \\ N - (\tilde{j} - j) & \text{c.c.} \end{cases}$$

donde $k$ está bien definido y $0 \leq k < N$. Además se da que

$$[j + k]_N = [\tilde{j}]_N \tag{4.1}$$

Por desarrollos $s$-ádicos podemos reescribir $k$ de la siguiente manera:

$$k = \sum_{i=0}^{l'-1} k_i 2^i \tag{4.2}$$

con $0 \leq k_i \leq 1$ y donde $l' = \lceil log_2 N \rceil = l$.

Ahora podemos reescribir $[\tilde{j}]_N$ de la siguiente forma:

$[\tilde{j}]_N$
= { por 4.1 }
$[j + k]_N$
= { aritmética modular }
$[j]_N \oplus [k]_N$
= { por 4.2 }
$[j]_N \oplus [\sum_{i=0}^{l-1} k_i 2^i]_N$
= { aritmética modular }
$[j]_N \bigoplus_{i=0}^{l-1} [k_i 2^i]_N$

Entonces definimos

$$\begin{aligned} j'_l &= j \\ j'_i &= [j'_{i+1}]_N \oplus [k_i 2^i]_N \text{ con } 0 \leq i < l \end{aligned}$$

y claramente tenemos que

$$j'_0 = [j]_N \oplus [k_{l-1} 2^{l-1}]_N \oplus \cdots \oplus [k_0 2^0] = [\tilde{j}]_N = \tilde{j}$$

No resulta difícil ver que

$$j'_i = \begin{cases} D_i(j'_{i+1}) & \text{si } k_i = 1 \\ j'_{i+1} & \text{c.c. } (k_i = 0) \end{cases}$$

con lo cual de la definición de $Inv$ podemos realizar la siguiente cadena de implicaciones:

$$a.l.j = a.l.j'_l \Rightarrow a.(l-1).j'_{l-1} \Rightarrow \cdots \Rightarrow a.0.j'_0 = a.0.\tilde{j}$$

luego por transitividad de la implicación y generalización en $j$ y $\tilde{j}$ tenemos finalmente

$$(\forall j : 0 \leq j < N : (\forall \tilde{j} : 0 \leq \tilde{j} < N : a.l.j \Rightarrow a.0.\tilde{j}))$$

□

**Lema 4.8** *$D_i$ tiene inversa y esta es $D_i^{-1}(\tilde{j}) : [\tilde{j}]_N \oplus [-(2^i)]$.*

**Demostración:** Sea $j \in [0, N)$, entonces es inversa a izquierda

$$D_i^{-1}(D_i(j)) = D_i^{-1}([j]_N \oplus [2^i]_N) =$$
$$([j]_N \oplus [2^i]_N) \oplus [-(2^i)]_N = [j + 2^i - 2^i]_N = [j]_N = j$$

y a derecha

$$D_i(D_i^{-1}(j)) = D_i([j]_N \oplus [-(2^i)]_N) =$$
$$([j]_N \oplus [-(2^i)]_N) \oplus [2^i]_N = [j - 2^i + 2^i]_N = [j]_N = j$$

□

Demos algunas definiciones y probemos hechos relacionados con ellas.

$$\begin{array}{lcl} Inv_0 & : & (\forall i : 0 \leq i < l : (\forall j : 0 \leq j < N : a.(i+1).j \Rightarrow a.i.j)) \\ Inv_1 & : & (\forall i : 0 \leq i < l : (\forall j : 0 \leq j < N : a.(i+1).j \Rightarrow a.i.D_i(j))) \end{array}$$

Las fórmulas $Inv_0$ e $Inv_1$ son una partición de $Inv$ en su parte trivial, implicada por la topología del programa, y el resto. Probemos entonces este hecho.

**Lema 4.9** *$Inv \equiv Inv_0 \wedge Inv1$*

**Demostración:** Sale fácilmente aplicando dos veces la equivalencia

$$(\forall i : R.i : P.i \wedge Q.i) \equiv (\forall i : R.i : P.i) \wedge (\forall i : R.i : Q.i)$$

□

Definamos ahora un predicado que establece todas las implicaciones de $Inv_1$ excepto las que involucran $a.i.j$.

$$\begin{array}{lcl} Inv_2(i,j) & : & (\forall i' : 0 \leq i' < l : (\forall j' : 0 \leq j' < N \\ & & \quad \wedge (i' + 1 = i \Rightarrow j' \neq j) \wedge (i' = i \Rightarrow j' \neq D_{i'}^{-1}(j)) : \\ & & \quad a.(i'+1).j' \Rightarrow a.i'.D_{i'}(j'))) \end{array}$$

**Lema 4.10** *La fórmula $Inv_2(i,j)$ no involucra términos que contengan $a.i.j$.*

**Demostración:** Veamos justamente que el rango de los cuantificadores elimina esos elementos.

El rango resulta falso cuando se da

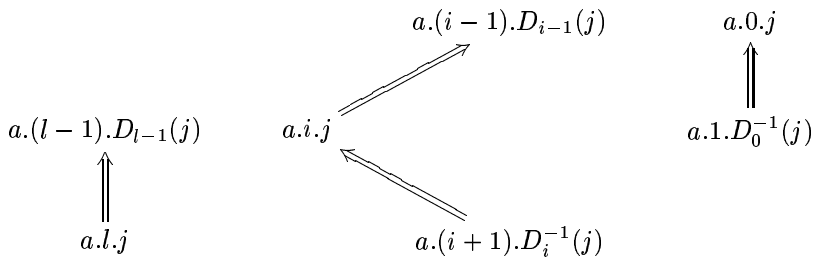
$$(i' + 1 = i \wedge j' = j) \vee (i' = i \wedge j' = D_{i'}^{-1}(j))$$

por lo que los pares $(i', j')$ que cumplan con esto no serán cuantificados. Reemplazando vemos que el primero es $a.i.j \Rightarrow a.(i-1).D_{i-1}(j)$ mientras que el segundo resulta $a.(i+1).D_i^{-1}(j) \Rightarrow a.i.j$.

Como las implicaciones son entre filas contiguas y la función $D_i$ es uno a uno, podemos concluir que no habrá otros términos eliminados.

□

Sin embargo dependiendo de la fila que se trate, $Inv_2(i,j)$ eliminará uno o dos términos. Los tres casos posibles están en el esquema




que nos induce a introducir el siguiente lema.

**Lema 4.11** *Sea $i \in [0,l]$, entonces:*

$$\begin{array}{rcl} i = l & \Rightarrow & \forall j : 0 \leq j < N : Inv_1 \equiv Inv_2(l,j) \wedge a.l.j \Rightarrow a.(l-1).D_{l-1}(j) \\ 0 < i < l & \Rightarrow & \forall j : 0 \leq j < N : Inv_1 \equiv Inv_2(i,j) \wedge \\ & & \quad a.(i+1).D_i^{-1}(j) \Rightarrow a.i.j \wedge a.i.j \Rightarrow a.(i-1).D_{i-1}(j) \\ i = 0 & \Rightarrow & \forall j : 0 \leq j < N : Inv_1 \equiv Inv_2(0,j) \wedge a.1.D_0^{-1}(j) \Rightarrow a.0.j \end{array}$$

**Demostración:** Por lema anterior y viendo que en cada caso los términos agregados son justamente los que excluye $Inv_2(i,j)$.

□

Estamos entonces en condiciones de comenzar a derivar el multiprograma.

Nota 0: El lema 4.9 nos dice que podemos dividir el invariante en dos y por topología del programa tenemos de manera gratuita $Inv_0$. El predicado $Inv_1$ es el que generará nuevas obligaciones de prueba.

I: resulta implicado, pues todos los $a.i.j$ son falsos en la precondición.

G:

- `a.i.j:=True`

  Dividimos la prueba en tres casos, dependiendo de si $i$ es 0, $l$ o un valor intermedio.

  - Caso $i = l$

    $wlp.$(`a.l.j:=True`)$.Inv_1$
    $\equiv$ { por lema 4.11 }
    $wlp.$(`a.l.j:=True`)$.(Inv_2(l,j) \wedge a.l.j \Rightarrow a.(l-1).D_{l-1}(j))$

$\equiv$ { def. de $wlp$, lema 4.10, cálculo proposicional }
$Inv_2(l, j) \wedge a.(l-1).D_{l-1}(j)$
$\Leftarrow$ { $Inv_2$ es más débil que $Inv_1$ }
$Inv_1 \wedge a.(l-1).D_{l-1}(j)$

– Caso $0 < i < l$

$wlp.(\texttt{a.l.j:=True}).(Inv_1)$
$\equiv$ { por lema 4.11, def. de $wlp$ }
$(Inv_2(i, j) \wedge a.(i+1).D_i^{-1}(j) \Rightarrow a.i.j \wedge a.i.j \Rightarrow a.(i-1).D_{i-1}(j))$
$(a.i.j := true)$
$\equiv$ { lema 4.10, cálculo proposicional }
$Inv_2(i, j) \wedge a.(i-1).D_{i-1}(j)$
$\Leftarrow$ { $Inv_2$ es más débil que $Inv_1$ }
$Inv_1 \wedge a.(i-1).D_{i-1}(j)$

– Caso $i = 0$

$wlp.(\texttt{a.0.j:=True}).(Inv_1)$
$\equiv$ { por lema 4.11 }
$wlp.(\texttt{a.l.j:=True}).(Inv_2(0, j) \wedge a.1.D_0^{-1}(j) \Rightarrow a.0.j$
$\equiv$ { def. de $wlp$, lema 4.10, cálculo proposicional }
$Inv_2(0, j) \wedge true$
$\Leftarrow$ { $Inv_2$ es más débil que $Inv_1$ }
$Inv_1$

El multiprograma queda entonces:

Pre : $(\forall i : 0 \leq i \leq l : (\forall j : 0 \leq j < N : \neg a.i.j))$
Inv : $(\forall i : 0 \leq i < l : (\forall j : 0 \leq j < N :$
$a.(i+1).j \Rightarrow a.i.D_i(j) \wedge a.(i+1).j \Rightarrow a.i.j))\heartsuit$

P.j:
```
⋮
a.0.j:=True;
for i:= 1 to l
    {a.(i-1).D_{i-1}(j)?N1}
    a.i.j:=True
rof;
⋮
```

Promesa: - la pre no contiene variables usadas en el flujo de control.

Nota 1: Copiamos la idea del algoritmo de la barrera *coarse grained* y decidimos crear un arreglo $\{f.i.j\}$ que siga a los $\{a.i.j\}$, pero manteniendo la promesa.

L: usando *modus ponens*, aseguramos la aserción con un skip guardado y una precondición.

G: por *widening* pues sólo hay asignaciones a verdadero.

```
Pre :  (∀i : 0 ≤ i ≤ l : (∀j : 0 ≤ j < N : ¬a.i.j))
Inv :  (∀i : 0 ≤ i < l : (∀j : 0 ≤ j < N :
         a.(i + 1).j ⇒ a.i.D_i(j) ∧ a.(i + 1).j ⇒ a.i.j))

P.j:  ⋮
      a.0.j:=True;
      for i:= 1 to l
          {f.(i − 1).D_{i−1}(j) ⇒ a.(i − 1).D_{i−1}(j)?N2}
          [ f.(i-1).D_{i−1}(j) ];
          {a.(i − 1).D_{i−1}(j)♡}
          a.i.j:=True
      rof;
      ⋮
```

Promesa: - la pre no contiene variables usadas en el flujo de control.

Nota 2:

L: para asegurar esta implicación seguimos lo hecho en el algoritmo de gruesa granularidad y prefijamos la aserción con una asignación a falso para el antecedente.

G:

- `a.i.j:=True`: directo por *widening*.
- `f.i.j:=False`: por la misma razón.

El multiprograma con todas las obligaciones de prueba cumplidas resulta

```
Pre :  (∀i : 0 ≤ i ≤ l : (∀j : 0 ≤ j < N : ¬a.i.j))
Inv :  (∀i : 0 ≤ i < l : (∀j : 0 ≤ j < N :
         a.(i + 1).j ⇒ a.i.D_i(j) ∧ a.(i + 1).j ⇒ a.i.j))

P.j:  ⋮
      a.0.j:=True;
      for i:= 1 to l
          f.(i-1).D_{i−1}(j):=False;
          {f.(i − 1).D_{i−1}(j) ⇒ a.(i − 1).D_{i−1}(j)♡}
          [ f.(i-1).D_{i−1}(j) ];
          {a.(i − 1).D_{i−1}(j)}
          a.i.j:=True
      rof;
      ⋮
```

Promesa: - la pre no contiene variables usadas en el flujo de control.

Tenemos un programa correctamente anotado, que nuevamente no progresa, sin embargo presenta las siguientes características positivas:

- No requiere de instrucciones especiales como `fetch-and-`$\Phi$ [MCS91], pues todas sus sentencias son 1-acceso, es decir, basta con que la arquitectura subyacente asegure accesos atómicos a memoria, y esta es una característica que comparten todas las arquitecturas paralelas.
- Las posiciones de memoria sobre las que cada proceso hace *busy waiting* están estáticamente determinadas, y además los conjuntos por proceso de estas *spinning variables* son disjuntos de a pares, lo cual permite lograr lo que se conoce como *local spinning* para máquinas con memoria local compartida o cachés coherentes[4].

En la sección siguiente repetiremos el esquema del algoritmo de gruesa granularidad, es decir agregaremos asignaciones `f.i.j:=True` que no modifican la corrección y permiten el progreso, aunque debemos adelantar que no pudimos encontrar una demostración general de este hecho.

### 4.3.1 Conjetura Sobre el Progreso

El método seguido para aumentar las posibilidades de progreso sin modificar la corrección de

$$\{f.i.D_i(j) \Rightarrow a.i.D_i(j)\}$$

se basa en agregar la sentencia `f.i.j:=True` en los siguientes lugares[5]:

- donde valga directa o indirectamente la aserción $\{a.i.j\}$
- junto con la asignación `a.i.j:=True`

Veamos entonces un lema que dice dónde son válidas las aserciones $\{a.i.j\}$ dentro de la componente $P.j$.

**Lema 4.12** *En el último multiprograma, la asignación* `a.i.j:=True` *tiene a* $\{a.i.j\}$ *como postaserción local y globalmente correcta. Además esta aserción continúa valiendo en el resto de la componente.*

**Demostración:** basta solamente con notar que $a.i.j$ sólo es asignado una vez en todo el multiprograma y con el valor verdadero.
□

[4]Esta característica también la tiene la derivación de gruesa granularidad.
[5]Claramente las asignaciones `f.l.j:=True` no tendrán efecto alguno sobre el progreso, pues no hay guardas con estas variables que lo impidan.

Nuestro programa instanciado en $N = 2$, utilizando la notación alternativa para el skip guardado, es el siguiente:

```
Pre :  ¬a.0.0 ∧ ¬a.0.1 ∧ ¬a.1.0 ∧ ¬a.1.1
Inv :  a.1.0 ⇒ a.0.1 ∧ a.1.1 ⇒ a.0.0
```

```
P.0:  a.0.0:=True;
      f.0.1:=False;
      {f.0.1 ⇒ a.0.1}
      [f.0.1];
      {a.0.1}
      a.1.0:=True;
```

```
P.1:  a.0.1:=True;
      f.0.0:=False;
      {f.0.0 ⇒ a.0.0}
      [f.0.0];
      {a.0.0}
      a.1.1:=True;
```

¿Cuál es la estrategia correcta para colocar los `f.i.j:=True`?. En este caso particular y según lo planteado en el párrafo inicial de la sección, tenemos tres lugares en $P.0$ para agregar `f.0.0:=True` y otros tantos en $P.1$ para `f.0.1:=True`.

Sin embargo, si quitamos las variables auxiliares y aserciones, y realizamos un renombre de variables, nos encontramos con un problema conocido.

```
A: y:=False;
   if y → skip fi;
```

```
B: x:=False;
   if x → skip fi;
```

Tenemos entre manos al multiprograma que da pie a [Fei94], el cual es un pequeño reporte técnico que de un solo golpe "*muestra los embrollos de la multiprogramación a las audiencias nóveles*". Justamente aquí se resalta que inundar de `x:=True` la componente $A$ y de `y:=True` la $B$, genera un multiprograma que muestra riesgo de deadlock. La solución, dice Feijen, está en quitar la primera asignación de cada componente, obteniendo el siguiente multiprograma:

```
A: y:=False;
   x:=True;
   if y → skip fi;
   x:=True;
```

```
B: x:=False;
   y:=True;
   if x → skip fi;
   y:=True;
```

La prueba de que este multiprograma progresa, es decir de que ambas componentes terminan, se deja como ejercicio para el lector[6].

A partir de este caso particular y de los algoritmos de sincronización de fase *sin inicialización de almacenamiento compartido* que se presentan en [Mis91], surge la siguiente heurística: poner todas las asignaciones a falso al tope de cada componente. Claramente la corrección del multiprograma no se ve afectada, pues las variables $\{f.i.D_i(j)\}_{i=0}^{l-1}$ son locales a la componente $P.j$.

[6]Hint: se necesita crear una variable auxiliar por componente que indicará el paso por la primera asignación a verdadero, luego podemos demostrar que previo a las guardas vale la aserción que dice que si ambas variables son verdaderas, entonces alguna de las guardas pasará. Entonces sólo resta demostrar que si una componente termina, entonces la otra hace lo propio.

Nuestro multiprograma se transforma en el siguiente:

```
Pre :  (∀i : 0 ≤ i ≤ l : (∀j : 0 ≤ j < N : ¬a.i.j))
Inv :  (∀i : 0 ≤ i < l : (∀j : 0 ≤ j < N :
         a.(i + 1).j ⇒ a.i.D_i(j) ∧ a.(i + 1).j ⇒ a.i.j))

P.j:   ⋮
       for i:= 0 to l-1
           f.i.D_i(j):=False
       rof;
       a.0.j:=True;
       for i:= 1 to l
           {f.(i − 1).D_{i−1}(j) ⇒ a.(i − 1).D_{i−1}(j)}
           [ f.(i-1).D_{i−1}(j) ];
           {a.(i − 1).D_{i−1}(j)}
           a.i.j:=True
       rof;
       ⋮
```

Notamos que $a.0.j$ está marcando el final del bloque de invalidación de guardas para la componente $P.j$.

Lo que resta es agregar asignaciones `f.i.j:=True` en las dos clases de lugares permitidos.

Hagamos un par de intentos simples, de forma tal que nos convenzamos de la complejidad que implica lograr que este multiprograma progrese.

Podemos lograr la menor modificación la aumentando `a.i.j:=True` a `a.i.j,f.i.j:=True,True`. Sin embargo, haciendo esto, instanciando para $N = 3$ y quitando toda variable auxiliar o aserción obtenemos:

```
P.0:  f.0.1:=False;   P.1:  f.0.2:=False;   P.2:  f.0.0:=False;
      f.1.2:=False;         f.1.0:=False;         f.1.1:=False;
      f.0.0:=True;          f.0.1:=True;          f.0.2:=True;
      [f.0.1];              [f.0.2];              [f.0.0];
      f.1.0:=True;          f.1.1:=True;          f.1.2:=True;
      [f.1.2];              [f.1.0];              [f.1.1];
```

Con la secuencia de ejecución ($P.1$,$P.1$,$P.1$,$P.0$,$P.0$,$P.0$) llegamos a un estado que cumple con $\neg f.0.1$ y no existe continuación de esta computación que haga válido $f.0.1$ y $P.0$ pueda progresar. El multiprograma sufre de riesgo de deadlock.

El problema es que $P.0$ invalida $f.0.1$ y éste es asignado una sola vez. Una posible solución es reasignar las variables $f.i.j$ a verdadero, luego de pasar por cada guarda.

El texto de programa con esta modificación para $N = 3$ es:

```
P.0:  f.0.1:=False;  P.1:  f.0.2:=False;  P.2:  f.0.0:=False;
      f.1.2:=False;        f.1.0:=False;        f.1.1:=False;
      f.0.0:=True;         f.0.1:=True;         f.0.2:=True;
      [f.0.1];             [f.0.2];             [f.0.0];
      f.1.0:=True;         f.1.1:=True;         f.1.2:=True;
      f.0.0:=True;         f.0.1:=True;         f.0.2:=True;
      [f.1.2];             [f.1.0];             [f.1.1];
      f.0.0:=True;         f.0.1:=True;         f.0.2:=True;
      f.1.0:=True;         f.1.1:=True;         f.1.2:=True;
```

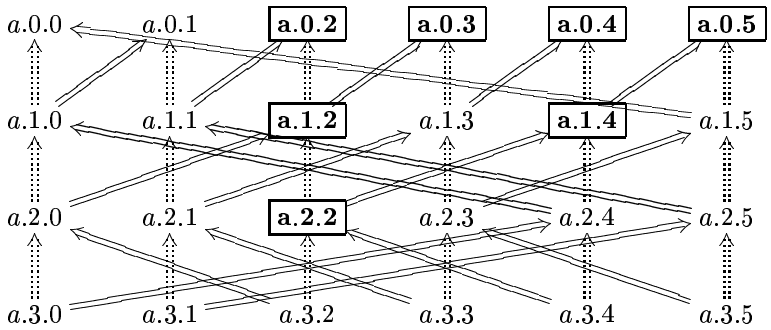
La secuencia de ejecución

$$(P.1, P.1, P.1, P.2, P.2, P.2, P.1, P.1, P.1, P.0, P.0, P.0, P.2, P.2, P.2, P.2, P.2, P.2)$$

hace que $P.2$ finalice, la guarda $f.0.1$ sea falsa y $P.1$, que puede hacerla verdadera, tampoco pueda progresar.

Los bloques iniciales que invalidan las guardas producen efectos muy malos en el progreso, y la única solución que hasta ahora se presenta como plausible, se basa en asignar todos los $f.i.j$ posibles luego de cada guarda. Para esto necesitamos saber todos los $a.i.j$ válidos en un punto del multiprograma.

Si graficamos la matriz de $\{a.i.j\}$ conectada por las implicaciones dadas por el invariante, o sea las topológicas y las generadas por el $D_i$, para el caso $N = 6$, vemos que si $a.2.2$ es verdadero, lo cual se da por debajo de la sentencia `a.2.2:=True`, entonces todos los $\{a.i'.j'\}$ encerrados en círculos lo serán y por lo tanto podremos reasignar todos esos $\{f.i'.j'\}$.




En general podemos decir que si $a.i.j$ es verdadero, entonces todos los $\{a.i'.j'\}$ alcanzables por él lo serán.

Si podemos dar una expresión concreta para la alcanzabilidad, es decir para la clausura transitiva y reflexiva del operador $\Rightarrow$ definido a partir del invariante sobre los valores $\{a.i.j\}$, entonces podremos saber exactamente cuáles $\{a.i.j\}$ valen después de una asignación `a.i.j:=True`, y por lo tanto cuáles $f.i.j$ podemos asignar a verdadero sin invalidar las aserciones.

**Lema 4.13** *Dados $i, j$ tales que $0 \leq i \leq l$ y $0 \leq i < N$, entonces se da*

$$a.i.j \Rightarrow (\forall \tilde{i} : 0 \leq \tilde{i} \leq i : (\forall n : 0 \leq n < 2^{i-\tilde{i}} : a.\tilde{i}.[j + n\,2^{\tilde{i}}]_N))$$

**Demostración:** por inducción en $i$.

Para $i = 0$ tenemos que el rango del cuantificador externo se limita a $\tilde{i} = 0$, y por lo tanto el rango del interno obliga $n = 0$, con lo cual tenemos

$$a.i.j \Rightarrow a.i.j$$

que resulta verdadero de manera trivial.

Veamos ahora que $P(i) \Rightarrow P(i+1)$.

Usando las dos posibles implicaciones de $a.(i+1).j$ y la hipótesis inductiva, tenemos:

$$a.(i+1).j \overset{def}{\Rightarrow} a.i.j \overset{HI}{\Rightarrow} (\forall \tilde{i} : 0 \leq \tilde{i} \leq i : (\forall n : 0 \leq n < 2^{i-\tilde{i}} : a.\tilde{i}.[j + n\,2^{\tilde{i}}]_N))$$

$$a.(i+1).j \overset{def}{\Rightarrow} a.i.[j + 2^i] \overset{HI}{\Rightarrow} (\forall \tilde{i} : 0 \leq \tilde{i} \leq i : (\forall n : 0 \leq n < 2^{i-\tilde{i}} : a.\tilde{i}.[j + 2^i + n\,2^{\tilde{i}}]_N))$$

Como $i \geq \tilde{i}$, entonces se da

$$j + 2^i + n\,2^{\tilde{i}} = j + 2^{i-\tilde{i}}2^{\tilde{i}} + n\,2^{\tilde{i}} = j + 2^{\tilde{i}}(\overbrace{n + 2^{i-\tilde{i}}}^{n'})$$

luego por cambio de coordenadas $n \to n'$ tenemos

$$a.(i+1).j \Rightarrow (\forall \tilde{i} : 0 \leq \tilde{i} \leq i : (\forall n' : 2^{i-\tilde{i}} \leq n' < 2^{i-\tilde{i}+1} : a.\tilde{i}.[j + 2^i + n'\,2^{\tilde{i}}]_N))$$

entonces por conjuntividad del $\forall$ podemos deducir

$$\begin{aligned} a.(i+1).j \quad \Rightarrow \quad & (\forall \tilde{i} : 0 \leq \tilde{i} \leq i : (\forall n : 2^{i-\tilde{i}} \leq n < 2^{i-\tilde{i}+1} : a.\tilde{i}.[j + 2^i + n\,2^{\tilde{i}}]_N) \wedge \\ & (\forall \tilde{i} : 0 \leq \tilde{i} \leq i : (\forall n : 0 \leq n < 2^{i-\tilde{i}} : a.\tilde{i}.[j + n\,2^{\tilde{i}}]_N)) \end{aligned}$$

ahora por disjunción de rangos llegamos a

$$a.(i+1).j \Rightarrow (\forall \tilde{i} : 0 \leq \tilde{i} \leq i : (\forall n' : 0 \leq n' < 2^{(i+1)-\tilde{i}} : a.\tilde{i}.[j + n'\,2^{\tilde{i}}]_N))$$

Como una extensión de rango de $\tilde{i}$ a $i+1$ sólo agrega $a.(i+1).j$, entonces tenemos finalmente $P(n+1)$

$$a.(i+1).j \Rightarrow (\forall \tilde{i} : 0 \leq \tilde{i} \leq i+1 : (\forall n' : 0 \leq n' < 2^{(i+1)-\tilde{i}} : a.\tilde{i}.[j + n'\,2^{\tilde{i}}]_N))$$

□

**Lema 4.14** *En el multiprograma, y luego de la asignación* `a.i.j:=True` *en la componente P.j, la siguiente aserción es local y globalmente correcta.*

$$\{\forall \tilde{i} : 0 \leq \tilde{i} \leq i : (\forall n : 0 \leq n < 2^{i-\tilde{i}} : a.\tilde{i}.[j + n\,2^{\tilde{i}}]_N)\}$$

**Demostración:** vale localmente por lema anterior y la asignación previa, y globalmente por *widening*.
□

El multiprograma con todas las asignaciones a $f.i.j$ posibles generadas por la alcanzabilidad es:

```
Pre :  (∀i : 0 ≤ i ≤ l : (∀j : 0 ≤ j < N : ¬a.i.j))
Inv :  (∀i : 0 ≤ i < l : (∀j : 0 ≤ j < N :
         a.(i+1).j ⇒ a.i.D_i(j) ∧ a.(i+1).j ⇒ a.i.j))

P.j:  ⋮
      for i:= 0 to l-1
          f.i.D_i(j):=False
      rof;
      a.0.j:=True;
      f.0.j:=True;
      for i:= 1 to l
          {f.(i-1).D_{i-1}(j) ⇒ a.(i-1).D_{i-1}(j)}
          [ f.(i-1).D_{i-1}(j) ];
          {a.(i-1).D_{i-1}(j)}
          a.i.j:=True
          {∀ĩ : 0 ≤ ĩ ≤ i : (∀n : 0 ≤ n < 2^{i-ĩ} : a.ĩ.[j + n 2^ĩ]_N)♡}
          for ĩ:= 0 to i
              for n:= 0 to 2^{ĩ-i}-1
                  f.ĩ.[j + n 2^ĩ]_N:=True
              rof
          rof
      rof;
      ⋮
```

Instanciando nuevamente para $N = 3$ este algoritmo, el último caso de deadlock mostrado ya no ocurre, y no pudimos detectar algún otro, pero tampoco obtuvimos alguna intuición acerca del porqué no ocurren deadlocks, lo único que podemos decir es que la reasignación de todos los $\{f.i.j\}$ dados por la alcanzabilidad están constantemente "reparando" los baches dejados por las invalidaciones iniciales.

Es probable que se pueda obtener una demostración formal de progreso para este caso particular, pero el objetivo es tratar de verlo en el caso general. De más está decir que nuestra conjetura es en favor del progreso.

Para finalizar, podemos notar que la cantidad de instrucciones por cada componente es $\mathcal{O}(N\,log^2 N)$, y esto se debe a las reasignaciones de $f.i.j$. Sin embargo existen dos formas disminuir esta cantidad de reasignaciones:

- Detectando los casos donde se está haciendo `f.i.j:=True` y en la preaserción tenemos $\{f.i.j\}$.
- Suponiendo que sí hay inicialización en la memoria compartida

Veamos entonces el primer caso que se basa en optimizar el código, utilizando por ejemplo alguno de los siguientes lemas.

**Lema 4.15** *En el multiprograma anterior, luego de la guarda* $[f.i.D_i(j)]$ *la aserción* $\{f.i.D_i(j)\}$ *empieza a ser válida y no deja de serlo hasta el final del texto de programa.*

**Demostración:** la corrección local se sigue de la regla de *modus ponens* para skip guardados y por *widening* y ortogonalidad sale la corrección global y las otras correcciones locales. □

**Lema 4.16** *En el último multiprograma, si tenemos en* $P.j$ *el bloque*

$$\cdots f.i.j{:=}True\{f.i.j?\}\cdots$$

*entonces la aserción será correcta si existe un bloque anterior*

$$\cdots [f.\tilde{i}.D_{\tilde{i}}(j)]\cdots$$

*tal que* $i, \tilde{i}$ *satisfacen* $N \mid 2^i + 2^{\tilde{i}}$.

**Demostración:** De manera directa podemos ver que la aserción es correcta localmente, sin embargo puede haber interferencia por parte de otra componente $\tilde{j} \neq j$, pues ésta puede falsificar la aserción con la asignación inicial $\texttt{f.i.}D_i(\tilde{j})\texttt{:=False}$, donde $D_i(\tilde{j}) = j$, con lo cual este $\tilde{j}$ es único y

$$\tilde{j} = D_i^{-1}(j)$$

Por otro lado resulta fácil demostrar que el siguiente fragmento de la componente $P.\tilde{j}$ es correcto pues está implicado por la precondición y el hecho de que los $\{a.i.\tilde{j}\}_i$ son locales a $P.\tilde{j}$

$$\cdots\{\forall i : 0 \leq i \leq l : \neg a.i.D_i^{-1}(j)\}\ \texttt{f.i.j:=False}\cdots$$

Luego nuestra aserción será globalmente correcta si logramos demostrar que

$$\{\exists i : 0 \leq i \leq l : a.i.\tilde{j}\}$$

resulta coaserción de ésta, pues por regla de aserciones disjuntas tenemos este hecho.

Estamos buscando que en $P.j$ se haga verdadero un $a.i.\tilde{j}$ que es asignado en $P.\tilde{j}$, luego la única posibilidad es que alguno de los bloques anteriores de $P.j$

P.j: $\vdots$
$\{f.(\tilde{i}).D_{\tilde{i}}(j) \Rightarrow a.(\tilde{i}).D_{\tilde{i}}(j)\}$
`[ f.`$\tilde{\text{i}}$`. `$D_{\tilde{i}}$`(j) ];`
$\{a.\tilde{i}.D_{\tilde{i}}(j)\}$
$\vdots$

haga justamente válido el $\{a.\tilde{i}.D_{\tilde{i}}(j)\}$ que cumpla con $\tilde{j} = D_{\tilde{i}}(j)$, con lo cual por *widening* y ortogonalidad, esta aserción seguirá siendo válida hasta que termine la componente $P.j$, y en particular lo será luego de la asignación `f.i.j:=True`.

Es decir se debe cumplir con:

$\tilde{j} = D_{\tilde{i}}(j)$
$\equiv$ { def. de $\tilde{j}$ }
$D_i^{-1}(j) = D_{\tilde{i}}(j)$
$\equiv$ { def. de $D$ y $D^{-1}$ }
$j - 2^i \equiv_N j + 2^{\tilde{i}}$
$\equiv$ { def. de $\equiv_N$ }
$N \mid (j - 2^i) - (j + 2^{\tilde{i}})$
$\equiv$ { simplificando }
$N \mid 2^i + 2^{\tilde{i}}$
□

Ambos lemas están diciendo lo mismo, una vez que $f.i.j$ resulta verdadero por implicación de una guarda y este hecho no puede ser invalidado, entonces este valor continuará siendo verdadero.

La otra estrategia para reducir la cantidad de reasignaciones, es precisamente no necesitarlas, y podemos lograr esto de manera directa levantando el requerimiento de no inicialización de memoria compartida y pidiendo en la precondición que el arreglo $\{f.i.j\}$ sea falso en todas sus entradas.

Sin realizar toda la derivación, veamos el resultado final. Los pasos intermedios pueden ser completados sin dificultad alguna por parte del lector asi como la correspondiente prueba de progreso.

`Pre :` $(\forall i : 0 \leq i \leq l : (\forall j : 0 \leq j < N : \neg a.i.j \wedge \neg f.i.j))$
`Inv :` $(\forall i : 0 \leq i < l : (\forall j : 0 \leq j < N :$
$a.(i+1).j \Rightarrow a.i.D_i(j) \wedge a.(i+1).j \Rightarrow a.i.j))$

P.j: $\vdots$
`a.0.j,f.0.j:=True,True;`
**for** `i:= 1 to l`
$\{f.(i-1).D_{i-1}(j) \Rightarrow a.(i-1).D_{i-1}(j)\}$
`[ f.(i-1).`$D_{i-1}$`(j) ];`
$\{a.(i-1).D_{i-1}(j)\}$
`a.i.j,f.i.j:=True,True`
**rof**;
$\vdots$

Éste es el algoritmo básico de barrera de diseminación que figura en el trabajo de Hensgen [HFM88], que consta de $\mathcal{O}(logN)$ pasos. Si embargo la reducción en la complejidad tiene como contrapartida que levantamos el requerimiento de no inicialización. Consultando el trabajo de Hensgen surge que podemos utilizar la estrategia de *episodios adyacentes* y de *inversión del sentido* para obtener una barrera, que en su primera y segunda fase no supone almacenamiento compartido inicializado y requiere $\mathcal{O}(N\,log^2N)$ operaciones, mientras que las restantes fases ya suponen el almacenamiento inicializado y trabajan en $\mathcal{O}(logN)$ pasos.

El esquema general es el siguiente:

- En la primera barrera aplicamos el algoritmo sobre el arreglo $\{f.i.j\}$ que no supone memoria inicializada.
- En la segunda aplicamos nuevamente el mismo algoritmo pero sobre el arreglo $\{f'.i.j\}$.
- Ahora ya podemos suponer que el arreglo $\{f.i.j\}$ está inicializado[7]. Sin embargo después de la segunda barrera, el arreglo $\{f.i.j\}$ contiene valores verdaderos solamente, y el algoritmo planteado suponía este arreglo en falso. Esto no es mayor problema, pues podemos construir el dual de este algoritmo.
- Hacemos lo propio con el arreglo $\{f'.i.j\}$ que también en este punto está inicializado a *true*.
- Ahora $\{f.i.j\}$ tiene todos sus valores en falso y usamos la barrera de sentido positivo.
- Lo mismo pero para $\{f'.i.j\}$

De ahora en más la situación se repite. Mostremos un esquema general:

BarreraNoInit-SentidoPos({f.i.j})
BarreraNoInit-SentidoPos({f'.i.j})
BarreraInit-SentidoNeg({f.i.j})
BarreraInit-SentidoNeg({f'.i.j})
BarreraInit-SentidoPos({f.i.j})
BarreraInit-SentidoPos({f'.i.j})
⋮
BarreraInit-SentidoNeg({f.i.j})
BarreraInit-SentidoNeg({f'.i.j})
BarreraInit-SentidoPos({f.i.j})
BarreraInit-SentidoPos({f'.i.j})
⋮

Queda bastante claro que cuando utilizamos una barrera miles o millones de veces, como en los algoritmos paralelos de datos, la penalidad de tiempo que se incurre en las dos primeras barreras no resulta significativa.

[7]Notar que podría haber procesos todavía ejecutándose en la primera barrera, mientras otros lo están haciendo en la segunda, y por lo tanto no todo el $\{f.i.j\}$ estaría a *true*.

# Capítulo 5

# Conclusión y Trabajos Futuros

Nuestro trabajo se basó en el estudio de la teoría de Owicki-Gries y su aplicación en la derivación de multiprogramas siguiendo la metodología desarrollada por un grupo de trabajo de la Universidad Tecnológica de Eindhoven en Holanda.

Durante el curso de los estudios y a medida que recabábamos más material, comprendimos las fortalezas y debilidades de la metodología propuesta para la derivación de multiprogramas, así como de los problemas particulares de la programación concurrente que, para nosotros, significaba un área bastante poco explorada.

A la hora de aplicar la metodología surgió el problema de generalizar el *protocolo de inicialización* propuesto en [vdS94], y lo que en principio parecía una derivación más o menos rutinaria, dio resultados positivos en cuanto a que conocimos la clase de problemas *que no imponen inicialización del almacenamiento compartido.*

Además confirmamos lo estudiado en relación a la teoría de Owicki-Gries y las pruebas de progreso. En multiprogramas sencillos, podemos capturar en un predicado la noción de progreso y demostrarlo dentro de la teoría, pero cuando la complejidad sube, ya sea porque tenemos muchos deadlocks potenciales o el multiprograma es de muy fina granularidad, resulta difícil obtener tales predicados y esto lo palpamos claramente en nuestro último multiprograma.

Tomando como base la diferencia de complejidad entre el problema de la barrera con y sin inicialización del almacenamiento compartido, y teniendo en cuenta la poca cantidad de material que pudimos recabar en este aspecto[1] ésta se presenta como una de las áreas hacia las cuales dirigiremos nuestros esfuerzos para trabajos futuros.

Otra área está bastante clara, las demostraciones de progreso fueron nuestro punto débil, y tendremos que reveer el material a fin de detectar si realmente la demostración faltante estaba a la mano de la teoría de Owicki-Gries o si debemos comenzar a estudiar otros formalismos que efectivamente capturen la noción de progreso, como por ejemplo las lógicas temporales.

[1] De los seis o siete referencias bibliográficas consultadas sobre sincronización de fase de $N$ procesos, sólo [Mis91] no supone inicialización

# Agradecimientos

No puedo dejar de agradecer a algunas de las personas, grupos y programas que siento hicieron posible que este trabajo esté terminado. Gracias a: Javier, Elisa, Sergio, Lucrecia, Pablo, Nico, María, el GTMC, GNU Linux, LaTeX y, por supuesto, a mis padres Olga y Daniel.

# Bibliografía


[And91] Gregory R. Andrews. *Concurrent Programming: principles and practice.* Benjamin Cummings, 1991.

[AO91] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs.* Graduate texts in Computer Science. Springer-Verlag, New York, 1991.

[Bes96] Eike Best. *Semantics of Sequential and Parallel Programs.* Prentice Hall International Series in Computer Science. Prentice Hall International, UK, 1996.

[Bro86] Eugene D. Brooks, III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[Cou90] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 15. Elsevier Science Publishers B.V., 1990.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[Dij82] E. W. Dijkstra. A personal summary of the Gries-Owicki theory. In *Selected Writings on Computing: A Personal Perspective*, pages 188–199. Springer-Verlag, New York, 1982.

[Fei90] W. H. J. Feijen. A little exercise in deriving multiprograms. In W. H. J. Feijen, A. J. M. van Gasteren, David Gries, and Jayadev Misra, editors, *Beauty is our Business*, chapter 13. Springer-Verlag, New York, 1990.

[Fei94] W. H. J. Feijen. A case of intuitive reasoning. Technical Report WF187, Eindhoven University of Technology, Department of Mathematics and Computer Science, August 1994.

[Fei95] W. H. J. Feijen. Coarse-grained handshake and alternating bit protocols or separate your concerns. Technical Report WF196, Eindhoven University of Technology, Department of Mathematics and Computer Science, February 1995.

[FvG95a] W. H. J. Feijen and A. J. M. van Gasteren. Lecture notes, design of multiprograms. Technical Report AvG122/WF208, Eindhoven University of Technology, Department of Mathematics and Computer Science, May 1995.

[FvG95b] W. H. J. Feijen and A. J. M. van Gasteren. On a method for the formal design of multiprograms. Technical Report AvG140/WF231, Eindhoven University of Technology, Department of Mathematics and Computer Science, May 1995.

[Gri81] David Gries. *The Science of Programming.* Text and Monographs in Computer Science. Springer-Verlag, New York, 1981.

[Gup89] Rajiv Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. *Third Int. Conf. on Architectural Support for Prog. Languages and Opertating Systems*, pages 54–63, April 1989.

[GV93] Dirk Grunwald and Suvas Vajracharya. Efficient barriers for distributed shared memory computers. Technical Report CU-CS-703-94, University of Colorado at Boulder, Departament of Computer Science, September 1993.

[HFM88] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[Hoo95] Rob R. Hoogerwoord. Progress, deadlock, and individual starvation. Technical Report RH219, Eindhoven University of Technology, Department of Mathematics and Computer Science, January 1995.

[Kal90] Anne Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice Hall International Series in Computer Science. Prentice Hall International, UK, 1990.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions in Software Engineer*, 3(2):125–143, 1977.

[Lam88] Leslie Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, 1988.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[Mis91] Jayadev Misra. Phase synchronization. *Information Processing Letters*, (38):101–105, 1991.

[Moe93] P.D. Moerland. Exercises in multiprogramming. Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, 1993.

[MV93] S. Mauw and G. J. Veltink, editors. *Algebraic Specification of Communication Protocols.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof techniques for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[Owi75] Susan S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Computer Science Department, Cornell University, 1975.

[Pfi98] Gregory F. Pfister. *In Search of Clusters.* Prentice Hall PTR, New Jersey, second edition, 1998.

[vdS94] F. W. van der Sommen. Multiprogram derivation. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, 1994.

[YTL86] P.C Yew, N.F. Tzeng, and D.H. Lawrie. Distributing hot-spot addressing in large scale multiprocessors. *IEEE Trans. on Computers*, C-36(4):388–395, April 1986.