# 1.6 El reto de la programación concurrente

El reto de la programación concurrente proviene de la necesidad de sincronizar la ejecución de diferentes procesos y de hacerlos capaces de comunicarse. Si los procesos fuesen totalmente independientes, la implementación de la concurrencia solamente requeriría un simple scheduler para asignar recursos entre ellos. Pero si un proceso I/O acepta un carácter tipeado en el teclado, este debe de alguna manera comunicarlo al proceso que está corriendo el procesador de palabras, y si hay múltiples ventanas sobre una pantalla, los procesos deben de algún modo sincronizarse para acceder a la pantalla de tal modo que las imágenes sean enviadas a la ventana con el foco actual.

Resulta ser extremadamente difícil implementar sincronización y comunicación segura y eficiente. Cuando su computadora personal se congela (“freezes up”) o cuando el uso de una aplicación causa que otra aplicación choque (“crash”), la causa es generalmente un error en sincronización o comunicación. Tales problemas dependen del tiempo y de alguna situación, y esos problemas son difíciles de reproducir, diagnosticar y corregir.

Transición

Hemos definido programación concurrente informalmente, basado en nuestra experiencia con sistemas de computadora. Nuestro objetivo es estudiar la concurrencia de forma abstracta, más que una implementación en un lenguaje de programación específico o sistema operativo. Tenemos que especificar cuidadosamente la abstracción que describe las estructuras de datos y operaciones permisibles. En el siguiente capítulo definiremos la abstracción de programación concurrente y justificaremos su relevancia. También haremos un repaso de los lenguajes y sistemas que pueden ser usados para escribir programas concurrentes.

# 2. La abstracción de programación concurrente

Las descripciones científicas del mundo están basadas en abstracciones. Un animal viviente es un sistema construido de órganos, huesos y así sucesivamente. Esos órganos están compuestos de células, las cuales en su turno están compuestas de moléculas, las cuales están compuestas de átomos, los cuales en su turno están compuestos de partículas elementales. Los científicos encuentran conveniente (y de hecho necesario) limitar sus investigaciones a un nivel, o tal vez dos niveles y hacer abstracción de los niveles más bajos. Entonces su físico (revisor físico/médico) escuchará su corazón o mirará en sus ojos, pero el generalmente no pensará acerca de las moléculas de las cuales estos órganos están compuestos. Hay otros especialistas, farmacólogos y bioquímicos, que son quienes estudian ese nivel de abstracción, en su turno haciendo abstracción de la teoría cuántica que describe la estructura y comportamiento de las moléculas.

En ciencia de la computación, las abstracciones también son así de importantes. Los ingenieros de software generalmente tratan con a lo más tres niveles de abstracción:

**Sistemas y bibliotecas** Sistemas operativos y bibliotecas ---frecuentemente llamadas Interfaces de Programa de Aplicación (API)--- definen recursos computacionales que están disponibles para el programador. Usted puede abrir un archivo o enviar un mensaje invocando al procedimiento adecuado o llamando a una función, sin saber cómo es implementado el recurso.

**Lenguajes de programación** Un lenguaje de programación le da a usted la posibilidad de emplear el poder computacional de una computadora, mientras usted hace abstracción de los detalles de las arquitecturas específicas.

**Conjuntos de instrucciones** La mayoría de los fabricantes de computadoras diseñan y construyen familias de CPUs las cuales ejecutan el mismo conjunto de instrucciones como es visto por el programador de lenguaje ensamblador o el escritor del compilador. Los miembros de una familia podrían ser implementados en formas totalmente diferentes ---emulando algunas instrucciones en software o usando memoria para los registros--- pero un programador puede escribir un compilador para ese conjunto de instrucciones sin conocer los detalles de la implementación.

Dos de las herramientas más importantes para la abstracción de software son la encapsulación y la concurrencia.

La encapsulación consigue abstracción dividiendo un módulo de software en una especificación pública y una implementación escondida. La especificación describe las operaciones disponibles sobre una estructura de datos o modelo del mundo real. La implementación detallada de la estructura o modelo es escrita dentro de un módulo separado que no es accesible desde el exterior. Entonces los cambios en la representación de datos interna y algoritmo puede ser hecha sin afectar la programación del resto del sistema. Los lenguajes de programación modernos directamente soportan encapsulación.

La concurrencia es una abstracción que está diseñada para hacer posible razonar acerca del comportamiento dinámico de programas. Esta abstracción será cuidadosamente explicada en el resto de este capítulo. Primero definiremos la abstracción y entonces mostraremos como relacionarla con varias arquitecturas de computadoras. Para los lectores que estén familiarizados con programación en lenguaje de máquina, las secciones 2.8 – 2.9 relacionan la abstracción a instrucciones de máquina; la conclusión es que no hay conceptos de concurrencia importantes que no puedan ser explicados en el nivel de abstracción más alto, así que esas secciones se las pueden saltar si así lo desean. El capítulo concluye con una introducción a programación concurrente en varios lenguajes y una sección suplementaria sobre un crucigrama que podría ayudarle a entender el concepto de estado y diagrama de estado.

## 2.2 Ejecución concurrente como entrelazado de sentencias atómicas

Ahora definimos la abstracción de programación concurrente que estudiaremos en este libro de texto. La abstracción está basada en el concepto de un proceso (secuencial), el cual no definiremos formalmente. Considérelo como un fragmento de un programa “normal” escrito en un lenguaje de programación. Usted no estará errado si piensa en un proceso como un nombre bonito para un procedimiento o método en un lenguaje de programación ordinario.

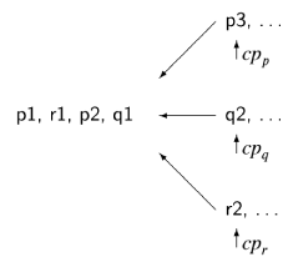
### 2.1 Definición

Un programa concurrente consiste de un conjunto finito de procesos (secuenciales). Los procesos son escritos usando un conjunto finito de sentencias atómicas. La ejecución de un programa concurrente procede ejecutando una secuencia de sentencias atómicas obtenida entrelazando arbitrariamente las sentencias atómicas del proceso. Una computación es una secuencia de ejecución que puede ocurrir como resultado del entrelazado. Las computaciones también son llamadas escenarios.

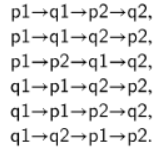
### 2.2 Definición

Durante una computación el puntero de control de un proceso indica la siguiente sentencia que puede ser ejecutada por ese proceso\footnote{Términos alternos para este concepto (contro pointer) son instruction pointer y location pointer}. Cada proceso tiene su propio puntero de control.

Las computaciones son creadas a través del entrelazado, el cual mezcla varios flujos de sentencias. En cada paso durante la ejecución del programa actual, la siguiente sentencia a ser ejecutada será “escogida” de las sentencias a las que apuntan los punteros de control cp de los procesos.

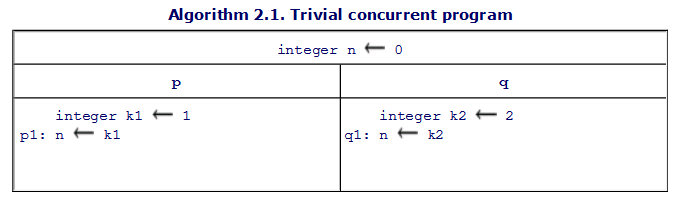


Suponga que tenemos dos procesos, p compuesto de las sentencias p1 seguida por p2 y q compuesto de las sentencias q1 seguida por q2, y que la ejecución es comenzada con los punteros de control de los dos procesos apuntando a p1 y a q1. Suponiendo que las sentencias son sentencias de asignación que no transfieren control, los posibles escenarios son:



Note que p2🡪p1🡪q1🡪q2 no es un escenario, porque respetamos la ejecución secuencial de cada proceso individual, así que p2 no puede ser ejecutado antes que p1.

Presentaremos programas concurrentes en una forma independiente del lenguaje, porque los conceptos son universales, si son implementados como llamadas al sistema operativo, directamente en lenguajes de programación como Ada o Java, o en un lenguaje de especificación de modelo como Promela. La notación es demostrada por el siguiente algoritmo concurrente de dos procesos trivial:

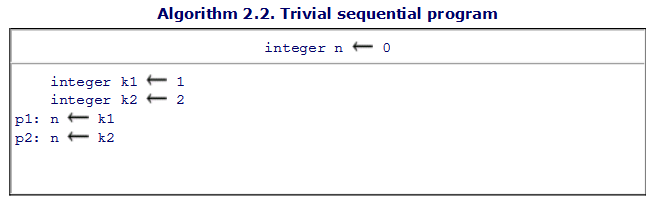


Al programa se le da un título, seguido por declaraciones de variables globales, seguido por dos columnas, una para cada uno de los dos procesos, los cuales por convención son llamados proceso p y proceso q. Cada proceso podría tener declaraciones de variables locales, seguidas por sentencias de los procesos. Usamos la siguiente convención:

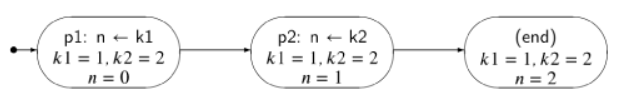
Cada línea etiquetada representa una sentencia atómica.

### Estados

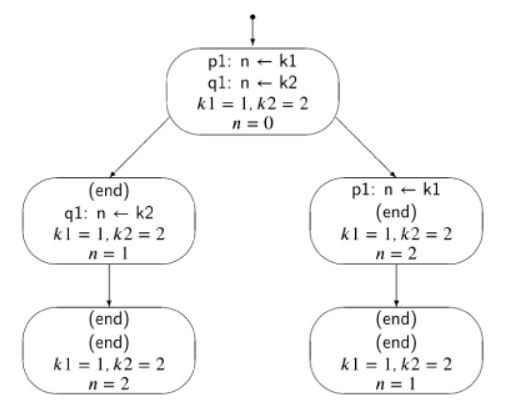
La ejecución de un programa concurrente está definida por estados y transiciones entre estados. Observemos primero esos conceptos en una versión secuencial del algoritmo de arriba:



En cualquier momento durante la ejecución de este programa, este debe estar en un estado definido por el valor del puntero de control y el valor de las tres variables. Ejecutar una sentencia corresponde a hacer una transición de un estado a otro. Es claro que este programa puede estar en uno de tres estados: un estado inicial y otros dos estados obtenidos ejecutando las dos sentencias. Esto es mostrado en el siguiente diagrama, donde un nodo representa un estado, las flechas representan las transiciones y el estado inicial es apuntado por la flecha corta en el lado izquierdo:



Considere ahora el programa concurrente trivial Algorithm 2.1. Hay dos procesos, así que el estado debe incluir los punteros de control de ambos procesos. Además, en el estado inicial hay una elección de cuál sentencia ejecutar, así que hay dos transiciones desde el estado inicial.



Los estados del lado izquierdo corresponden a la ejecución de p1 seguido de q1, mientras que los estados del lado derecho corresponden a la ejecución de q1 seguido por p1. Note que las computaciones pueden terminar en dos estados diferentes (con valores de n diferentes), dependiendo del entrelazado de las sentencias.

### 2.3 Definición

El estado de un algoritmo (concurrente) es una tupla que consiste de un elemento por cada proceso que es una etiqueta de ese proceso, y un elemento por cada variable global o local que es un valor cuyo tipo es el mismo que el tipo de la variable.\footnote{La palabra tupla es una generalización de la secuencia par, tripleta, cuadrupla, etc. Significa una secuencia ordenada de valores de alguna longitud fija}

El número de estados posibles ---el número de tuplas--- es muy grande, pero en una ejecución de un programa, no todos los estados posibles pueden ocurrir. Por ejemplo, dado que no se asignan valores a las variables k1 y k2 en el Algorithm 2.1, ningún estado puede ocurrir en el cual esas variables tengan valores diferentes de sus valores iniciales.

### 2.4 Definición

Sean s1 y s2 estados. Hay una transición entre s1 y s2 si ejecutando una sentencia en el estado s1 cambia el estado a s2. La sentencia ejecutada debe ser una de aquellas apuntadas por un puntero de control en s1.

### 2.5 Definición

Un diagrama de estado es un grafo definido inductivamente. The initial state diagram contiene un solo nodo etiquetado con el estado inicial. Si el estado s1 etiqueta un nodo en el diagrama de estados y si hay una transición de s1 a s2, entonces hay un nodo etiquetado s2 en el diagrama de estados y un edge dirigido de s1 a s2.

Para cada estado, hay solamente un nodo etiquetado con ese estado.

El conjunto de estados alcanzables es el conjunto de estados en un diagrama de estados.

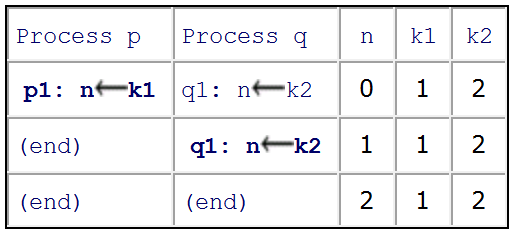
Se sigue de las definiciones que una computación (escenario) de un programa concurrente es representado por una ruta dirigida a través del diagrama de estados comenzando desde el diagrama inicial, y que todas las computaciones pueden ser representadas así. Los ciclos en el diagrama de estados representan la posibilidad de computaciones infinitas en un grafo finito.

El diagrama de estados para el Algorithm 2.1 muestra que hay dos escenarios diferentes, cada uno de los cuales contiene 3 de los 5 estados alcanzables.

Antes de proceder, usted podría desear leer la sección suplementaria 2.14, la cual describe el diagrama de estados para un puzzle interesante.

### Escenarios

Un escenario es definido como una secuencia de estados. Dado que los diagramas pueden ser difíciles de dibujar, especialmente para programas grandes, es conveniente usar una representación tabulada de escenarios. Esto se hace simplemente listando la secuencia de estados en una tabla; las columnas para los pointers de control son etiquetadas con los procesos y las columnas para los valores de variable con los nombres de las variables. La siguiente tabla muestra el escenario de Algorithm 2.1 correspondiente a la ruta del lado izquierdo:



En un estado podría haber más de una sentencia que puede ser ejecutada. Usamos bold font para denotar la sentencia que fue ejecutada para llegar al estado en el siguiente renglón.

Los renglones representan estados. Si la sentencia ejecutada es una sentencia de asignación, el nuevo valor que es asignado a la variable es un componente del siguiente estado en el escenario, el cual se encuentra en el siguiente renglón.

AL principio esto podría ser confuso, pero usted pronto se acostumbrará a los escenarios y lo que se acaba de comentar.

# 3. El problema de la sección crítica

## 3.1 Introducción

Se presenta una secuencia de intentos para resolver el problema de la sección crítica para dos procesos, culminando con el algoritmo de Dekker. Los mecanismos de sincronización serán construidos sin el uso de sentencias atómicas adicionales a las de carga y almacenamiento atómico. Siguiendo esa secuencia de algoritmos, en breve se presentarán soluciones al problema de la sección crítica que usan sentencias atómicas más complejas.

Algoritmos como el algoritmo de Dekker rara vez son usados en la práctica, primeramente, porque los sistemas reales soportan primitivas de sincronización más “altas”. Sin embargo, este capítulo es el corazón del libro, porque cada algoritmo incorrecto demuestra algún comportamiento patológico que es típico de algoritmos concurrentes. Estudiando esos comportamientos dentro del framework de esos algoritmos elementales lo preparará a usted para identificarlos y corregirlos en sistemas reales.

Las pruebas de corrección de los algoritmos estarán basadas en la construcción explícita de diagramas de estados en los cuales todos los escenarios están representados. Algunas de las pruebas son pospuestas para el siguiente capítulo, porque estas requieren de herramientas más complejas, en particular el uso de temporal logic para especificar y verificar propiedades de corrección.

## 3.2 Definición del problema

Comenzamos con una especificación de la estructura del problema de sección crítica y las suposiciones bajo las cuales debe ser resuelto:

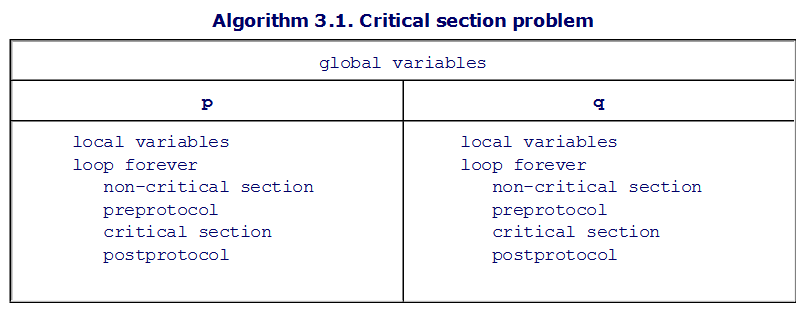
* Cada uno de los N procesos está ejecutando en un ciclo infinito una secuencia de sentencias que puede ser dividida en dos subsecuencias: la sección crítica y la sección no crítica.
* Las especificaciones de corrección de cualquier solución son:

**Exclusión mutua** Las sentencias de las secciones críticas de dos o más procesos no deben ser intercaladas.

**Libre de deadlock** Si algunos procesos están tratando de entrar a sus regiones críticas, entonces eventualmente uno de ellos debe tener éxito.

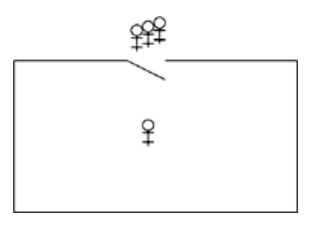
**Libre de inanición (individual) [starvation]** Si algún proceso está tratando de entrar a su región crítica, entonces ese proceso eventualmente debe tener éxito.

* Se debe proporcionar un mecanismo de sincronización para asegurar que los requerimientos de corrección se cumplan. El mecanismo de sincronización consiste de sentencias adicionales que son colocadas antes y después de la sección crítica. Las sentencias colocadas antes de la sección crítica son llamadas el preprotocolo y las que se colocan después son llamadas el postprotocolo. Entonces, la estructura de una solución para dos procesos es como sigue:



* Los protocolos podrían requerir variables locales o globales, pero suponemos que ninguna de las variables usadas en las secciones críticas y no críticas son usadas en los protocolos, y viceversa.
* La sección crítica debe progresar, esto es, una vez que un proceso empieza a ejecutar las sentencias de su sección crítica, eventualmente debe terminar de ejecutar esas sentencias.
* La sección no crítica no necesita progresar, esto es, si el apuntador de control de un proceso está en su sección no crítica, el proceso podría terminar o entrar a un ciclo infinito y no abandonar la sección no crítica.

El siguiente diagrama ayudará a visualizar el problema de sección crítica



Cada una de las figuritas representan procesos, y la caja es una región crítica en la cual a lo más un proceso en el tiempo podría estar ejecutando las sentencias que forman su sección crítica. La solución al problema es dada por los protocolos para abrir y cerrar la puerta a la región crítica de tal manera que las propiedades de corrección sean satisfechas.

Con el problema de la sección crítica se intenta modelar un sistema que realiza computación compleja, pero ocasionalmente necesita acceder a datos o a hardware que son compartidos por varios procesos. Es razonable esperar que procesos individuales nunca terminarán, o que nada malo sucederá con los programas del sistema. Por ejemplo, en un kiosco de check-in en un aeropuerto espera a que un pasajero deslice su tarjeta de crédito o toque la pantalla. Cuando eso ocurre, el programa en el kiosco accederá a una base de datos de pasajeros central, y es importante que solamente uno de esos programas actualice la base de datos a la vez. La sección crítica se usaría únicamente para modelar esta operación, mientras que la sección no crítica modelaría toda la computación del programa excepto por la actualización real de la base de datos.

No tendría sentido requerir que el programa realmente participe en la actualización de la base de datos en otros kioscos, así que permitimos que espere indefinidamente a recibir entrada, y tomamos en cuenta que el programa podría funcionar mal. En otras palabras, no requerimos que la sección no crítica progrese. Por otra parte, sí requerimos que la sección crítica progrese de tal manera que eventualmente termine. La razón es que el proceso que está ejecutando su sección crítica típicamente mantiene un “candado” o “recurso permiso,” y el candado o recurso eventualmente debe ser liberado para que otros procesos puedan entrar a sus respectivas secciones críticas. El requerimiento de progreso es razonable, porque las secciones críticas son usualmente muy cortas y su progreso puede ser verificado formalmente.

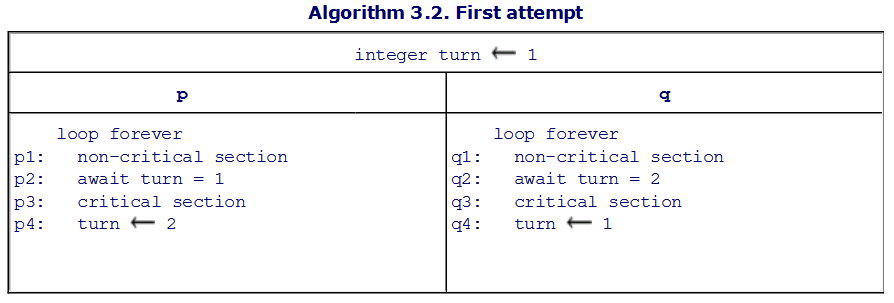
Obviamente, el deadlock (“mi computadora se congela”) debe ser evitado, porque los sistemas deben proporcionar un servicio. Aun si hay progreso local dentro de los protocolos mientras los procesos establecen y revisan las variables de protocolo, si ningún proceso tiene éxito en hacer la transición del preprotocolo a su sección crítica, el programa está deadlocked.

Libre de inanición (starvation) es un requerimiento fuerte en el sentido de que se debe mostrar que ninguna secuencia de ejecución posible, no importa que tan improbable sea, pueda causar la inanición de algún proceso. Este requerimiento puede ser debilitado; se verá un ejemplo en el algoritmo de la sección 5.4.

Una buena solución al problema de sección crítica también será eficiente, en el sentido de que los pre y post protocolos usarán tan poco tiempo y memoria como sea posible. En particular, si solo un proceso desea entrar a su sección crítica este deberá tener éxito casi inmediatamente.

## 3.3 Primer intento

Aquí está un primer intento por resolver el problema de la sección crítica para dos procesos:



La sentencia await turn = 1 es una notación independiente de implementación para una sentencia que espera hasta que la condición turn = 1 se vuelve verdadera. Esta puede ser implementada (aunque ineficientemente) por un ciclo de espera ocupada (busy-wait loop) que hace nada hasta que la condición es verdadera.

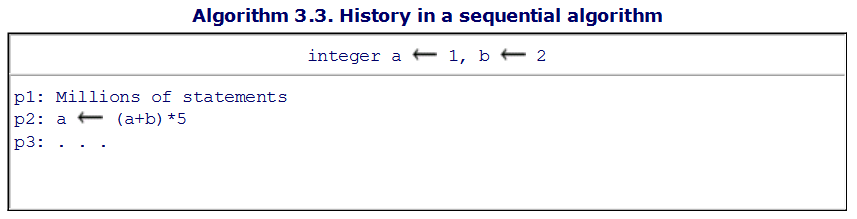
La variable global turn puede tomar los valores 1 y 2, y es inicialmente establecida al valor arbitrario 1. El significado que se le da a esta variable es que indica de quién es el “turno” de entrar a su región crítica. Un proceso que desea entrar a su región critica ejecutará un preprotocolo que consiste en una secuencia que espera hasta que el valor de turn indica que su turno ha llegado. Cuando el proceso sale de la sección critica, el proceso establece el valor de turn al número del otro proceso.

Queremos demostrar que este algoritmo satisface las tres propiedades requeridas de una solución al problema de la sección crítica. Para ese fin primero explicamos la construcción de diagramas de estado para programas concurrentes, un concepto que fue introducido en la sección 2.2.

## 3.4 Demostrando corrección con diagramas de estados

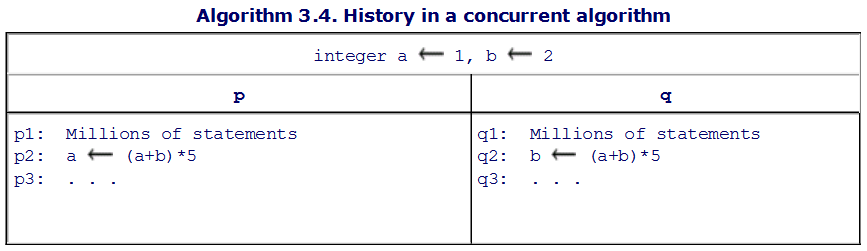
### Estado

Usted no necesita conocer la historia de la ejecución de un algoritmo para predecir el resultado del siguiente paso a ser ejecutado. Primero examinemos esta exclamación para algoritmos secuenciales y entonces consideraremos cómo tiene que ser modificada para algoritmos concurrentes. Considere el siguiente algoritmo:



Supóngase ahora que la computación ha alcanzado la sentencia p2 y que los valores de a y b son 10 y 20, respectivamente. Usted puede ahora predecir con absoluta certeza que como un resultado de la ejecución de la sentencia p2, el valor de a se convertirá en 150, mientras que el valor de b seguirá siendo 20; además, el apuntador de control de la computadora ahora contendrá p3. La historia de la computación --- cómo exactamente esta llegó a p2 con esos valores de las variables --- es irrelevante.

Para un algoritmo concurrente, la situación es similar:



Suponga que la ejecución ha alcanzado el punto donde el primer proceso ha llegado a la sentencia p2 y el segundo proceso ha llegado a la sentencia q2, y los valores de a y b son otra vez 10 y 20, respectivamente. Debido al intercalado, no podemos predecir si la siguiente sentencia a ser ejecutada vendrá del proceso p o del otro proceso q; pero podemos predecir que será de uno ó del otro, y podemos especificar lo que será el resultado en cualquiera de los dos casos.

En el algoritmo secuencial 3.3, los estados son tripletas tales que %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{i}=(p2,10,20)
\]
\end{document} . De las semánticas de las sentencias de asignación, podemos predecir que ejecutando la sentencia p2 de %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{i}
\]
\end{document} causará que el estado de la computación cambie a %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{i+1}=(p3,150,20)
\]
\end{document} . Entonces, dado el estado inicial de la computación %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{0}=(p1,1,2)
\]
\end{document}, podemos predecir el resultado de la computación. (Si hay sentencias de entrada, los valores colocados en las variables de entrada son también parte del estado.) Es precisamente esta propiedad lo que hace práctica la depuración: si usted encuentra un error, usted puede establecer un punto de ruptura y reiniciar una computación en el mismo estado inicial, con la confianza de que el estado de la computación en el punto de ruptura es el mismo que fue en la computación errónea.

En el algoritmo concurrente 3.4, los estados son cuadruplas (tuplas de 4 elementos) tales como %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{i}=(p2,q2,150,20)
\]
\end{document}. No podemos predecir cuál será el siguiente estado, pero podemos estar seguros de que será %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{i+1}^{p}=(p3,q2,150,20)
\]
\end{document} ó %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{i+1}^{q}=(p2,q3,10,150)
\]
\end{document} dependiendo de si la siguiente sentencia es tomada del proceso p o del proceso q, respectivamente. Mientras no podemos predecir cuáles estados aparecerán en alguna ejecución particular de un algoritmo concurrente, el conjunto de estados alcanzables (Definición 2.5) son los únicos estados que pueden aparecer en alguna computación. En este ejemplo, comenzando en el estado %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{i}
\]
\end{document}, el siguiente estado será un elemento del conjunto %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\{s_{i+1}^{p},s_{i+1}^{q}\}
\]
\end{document}. De cada uno de esos estados, hay posiblemente dos nuevos estados que pueden ser alcanzados, y así sucesivamente. Para revisar las propiedades de corrección, solamente es necesario examinar el conjunto de estados alcanzables y las transiciones entre ellos; estos son representados en el diagrama de estados.

Para el primer intento, algoritmo 3.2, los estados son tripletas de la forma %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p_{i},q_{j},turn)
\]
\end{document}, donde turn denota el valor de la variable turn. Recuerde que estamos asumiendo que cualesquiera variables usadas en las secciones críticas y no críticas son distintas de las variables usadas en los protocolos y de esa forma no pueden afectar la corrección de la solución. Por lo tanto, las dejamos fuera de la descripción del estado. La propiedad de corrección de exclusión mutua se cumple si el conjunto de todos los estados accesibles no contiene un estado de la forma %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p3,q3,turn)
\]
\end{document} para algún valor de turn, porque p3 y q3 son las etiquetas de las secciones críticas.

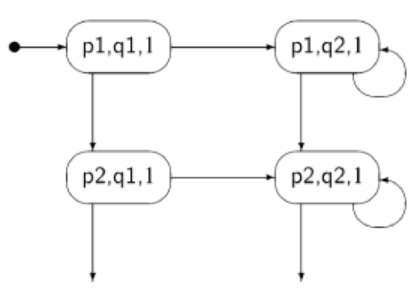
### Diagramas de estado

¿Cuántos estados hay en un diagrama de estados? Suponga que el algoritmo tiene %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N
\]
\end{document} procesos con %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
n_{i}
\]
\end{document} sentencias en el procesos %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
i
\]
\end{document}, y %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
M
\]
\end{document} variables donde la variable %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
j
\]
\end{document} tiene %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
m_{j}
\]
\end{document} posibles valores. El número de estados posibles es el número de tuplas que puede ser formado de esos valores, y podemos elegir cada elemento de la tupla independientemente, el número de estados es %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
n_{1}\times\ldots n_{N}\times m_{1}\times\ldots m_{M}
\]
\end{document}. Para el primer intento, el número de estados es %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
n_{1}\times n_{2}\times m_{1}=4\times 4\times 2=32
\]
\end{document}, dado que es claro del texto del algoritmo que la variable turn puede solamente tener dos valores, 1 y 2. En general, las variables pueden tomar tantos valores como su representación puede contener, por ejemplo, %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
2^{32}
\]
\end{document} para una variable entera de 32 bits.

Sin embargo, es posible que no todos los estados puedan ocurrir realmente, esto es, es posible que algunos estados aparezcan en algún escenario que comience en el estado %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
s_{0}=(p1,q1,1)
\]
\end{document}. De hecho, eso esperamos, esperamos que los estados %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p3,q3,1)
\]
\end{document} y %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p3,q3,2)
\]
\end{document}, los cuales violan el requerimiento de corrección de exclusión mutua, no sean accesibles desde el estado inicial.

Para demostrar una propiedad de un algoritmo concurrente, construimos un diagrama de estados y entonces analizamos si la propiedad se cumple en el diagrama. El diagrama es construido incrementalmente, comenzando con el estado inicial y considerando cuáles son los siguientes estados potenciales. Si un potencial estado próximo ya ha sido construido, entones podemos conectar hacia ese estado, entonces obteniendo una presentación finita de una ejecución no acotada del algoritmo. Por la naturaleza de la construcción incremental, turn solamente tomará solamente los valores 1 y 2, porque estos son los únicos valores que en algún momento le son asignados por el algoritmo.

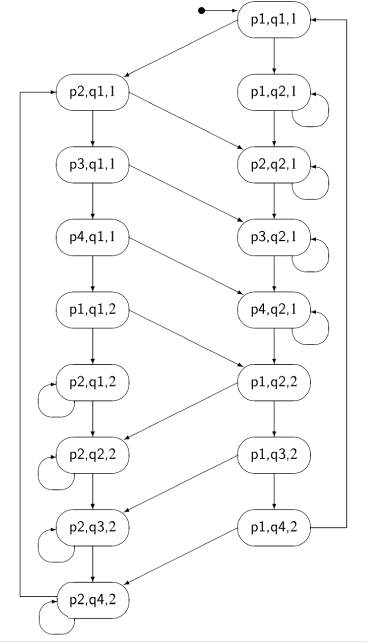
El siguiente diagrama muestra los primeros cuatro pasos de la construcción incremental del diagrama de estados para el algoritmo 3.2:



El estado inicial es %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p1,q1,1)
\]
\end{document}. Si ejecutamos p1 del proceso p, el siguiente estado es %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p2,q1,1)
\]
\end{document}: el primer elemento de la tupla cambia porque ejecutamos una sentencia del primer proceso, el segundo elemento no cambia porque no ejecutamos una sentencia de ese proceso, y el tercer elemento no cambia dado que --- por suposición --- la sección no crítica no cambia el valor de la variable turn. Si, sin embargo, ejecutamos q1 del proceso q, el siguiente estado es %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p1,q2,1)
\]
\end{document}. Desde este estado, si tratamos de ejecutar otra sentencia, q2 del proceso q, permanecemos en el mismo estado. La sentencia es await turn=2, pero turn=1. No dibujamos otra instancia de %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p1,q2,1)
\]
\end{document}; en lugar de eso, la flecha (la flecha circular) que representa la transición apunta hacia el mismo estado que ya existe.

La construcción incremental termina después de que16 de 32 estados posibles han sido construidos, como se muestra en la Figura 3.1. Usted podría (o podría no!) querer revisar si la construcción ha sido realizada correctamente.

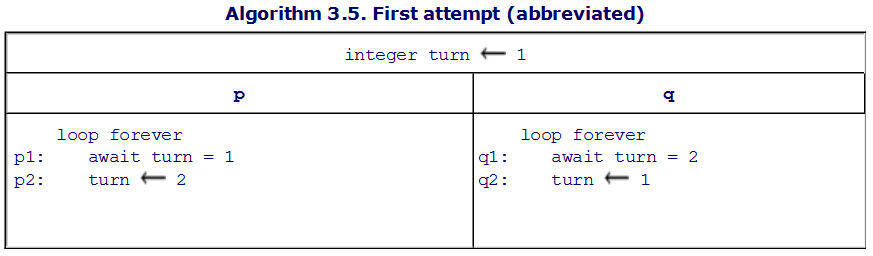




Una rápida revisión muestra ninguno de los estados %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p3,q3,1)
\]
\end{document} ni %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p3,q3,1)
\]
\end{document} ocurren en el diagrama; entonces hemos demostrado que la propiedad de exclusión mutua se cumple para el primer intento.

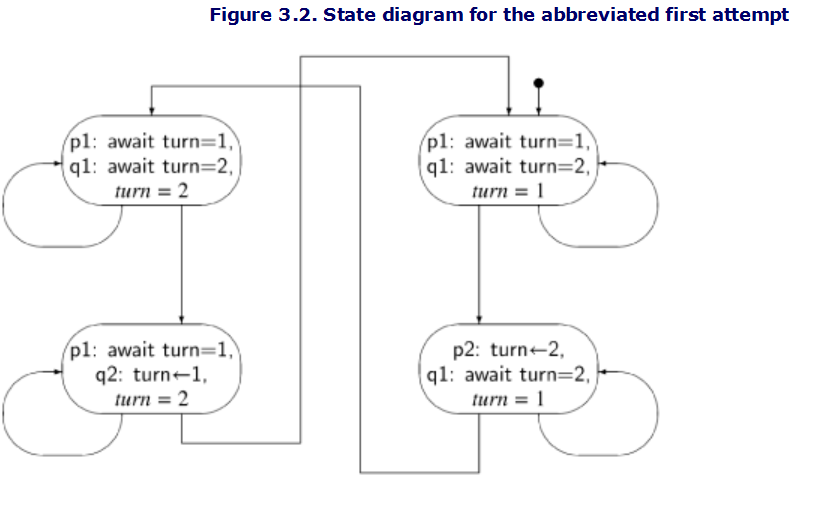
### Abreviando el diagrama de estados

Claramente el diagrama de estados del algoritmo simple en el primer intento es difícil (unwieldy). Cuando se construyen diagramas de estados, es importante minimizar el número de estados que deben ser construidos. En este caso, podemos reducir el número de estados accesibles de 16 a 4, reduciendo el algoritmo a la forma del que se muestra abajo, donde hemos removido las dos sentencias para las secciones críticas y no críticas.



Definitivamente, esto es sospechoso. El punto completo del algoritmo es asegurar que la exclusión mutua se cumple durante la ejecución de las secciones críticas, así que ¿cómo podemos simplemente borrar esas sentencias? La respuesta es que cualesquiera que sean las sentencias que son ejecutadas por la sección crítica son totalmente irrelevantes para la corrección del algoritmo de sincronización.

Si en el primer intento, remplazamos la sentencia p3: critical section por el comentario p3: //critical section, la especificación de las propiedades de corrección no cambia, por ejemplo, debemos mostrar que no podemos estar en alguno de los estados %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p3,q3,1)
\]
\end{document} o %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p3,q3,2)
\]
\end{document} . Pero si estamos en p3 y esa sentencia es un comentario, es justo como si estuviéramos en p4, así que podemos remover el comentario; un razonamiento similar se cumple para p1. En el primer intento abreviado, las secciones críticas han sido absorbidas por turn 🡨 2 y turn 🡨 1, y las secciones no crítiticas han sido absorbidas por await turn = 1 y await turn = 2. El diagrama de estados para este algoritmo es mostrado en la figura 3.2, donde por claridad hemos dado las sentencias y nombres de variables y no solo etiquetas.



## 3.5 Corrección del primer intento

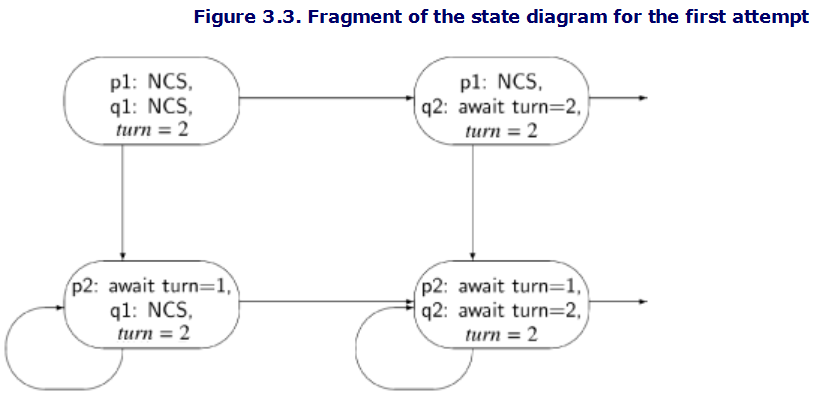
Estamos ahora en posición de tratar de demostrar la corrección del primer intento. Como se hizo notar arriba, la demostración de que la exclusión mutua se cumple es inmediata de la examinación del diagrama de estados.

A continuación, tenemos que demostrar que el algoritmo es libre de deadlock. Recuerde que esto significa que si algunos procesos están tratando de entrar a sus regiones críticas entonces uno de ellos debe eventualmente tener éxito. En el algoritmo abreviado, un proceso está tratando de entrar a su sección crítica si está tratando de ejecutar su sentencia await. Tenemos que revisar esto para los cuatro estados; dado que los dos estados del lado izquierdo son simétricos con los dos estados del lado derecho, es suficiente con revisar uno de los dos pares.

Considere el estado de arriba del lado izquierdo (await turn=1, await turn=2, turn=2). Ambos procesos están tratando de ejecutar sus secciones críticas; si el proceso q trata de ejecutar await turn=2, tendrá éxito y entrará a su sección crítica.

Considere ahora el estado de abajo del lado izquierdo (await turn=1, turn 🡨1, turn=2). El proceso p podría tratar de ejecutar await turn=1, pero dado que turn = 2, el programa no cambia de estado. Por la suposición de progreso sobre la sección crítica y la sentencia de asignación, el proceso q eventualmente ejecutará turn 🡨 1, conduciendo al estado de arriba del lado derecho. Ahora el proceso p puede entrar a su región crítica. Por lo tanto, la propiedad libre de deadlock se satisface.

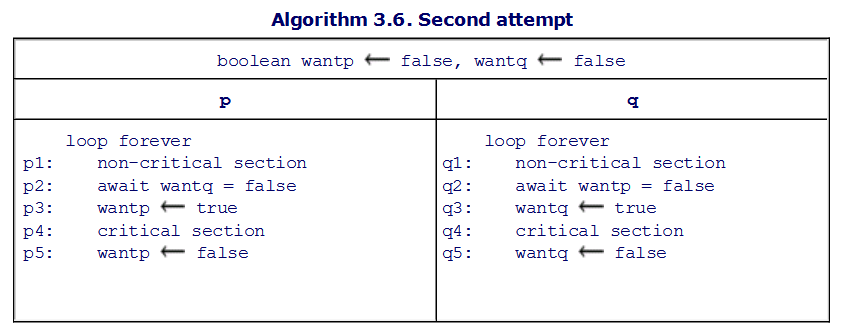
Finalmente, tenemos que revisar que el algoritmo es libre de inanición, lo que significa que si algún proceso se dispone a ejecutar su preprotocolo (indicando así su intención de entrar a su sección crítica), entonces eventualmente tendrá éxito en entrar a su sección crítica. El problema será más fácil de entender si consideramos el diagrama de estados para el algoritmo no abreviado; el fragmento apropiado del diagrama es mostrado en la Figura 3.3, donde NCS denota la sección no crítica. Considere el estado en la parte de abajo del lado izquierdo. El proceso p está tratando de entrar a su sección crítica ejecutando p2: await turn=1, pero dado que turn = 2, el proceso estará en un ciclo indefinidamente en su sentencia await hasta que el proceso q ejecuta q4 🡨 1. Pero el proceso q está en q1: NCS y no hay suposición de progreso para secciones no críticas. Por lo tanto, la inanición (starvation) ha ocurrido: el proceso p está continuamente revisando el valor de turn tratando de entrar a su sección crítica, pero el proceso q no necesita salir de su sección no crítica, lo cual es necesario si p ha de entrar a su sección crítica.



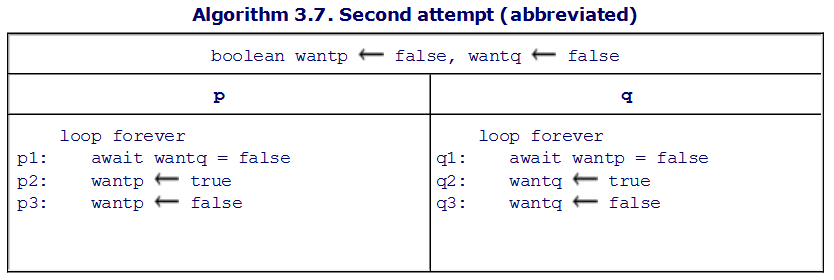
Informalmente, turn sirve como un recurso de permiso para entrar a la sección crítica, con su valor indicando cual proceso tiene el recurso. Hay siempre algún proceso que tiene el recurso de permiso, así que siempre algún proceso puede entrar a su sección crítica, asegurando que no hay deadlock. Sin embargo, si el proceso que tiene el recurso de permiso permanece indefinidamente en su sección no crítica ---como está permitido por las suposiciones del problema de sección crítica--- el otro proceso nunca recibirá el recurso y nunca entrará a su sección crítica. En nuestro próximo intento, aseguraremos que un proceso en su sección no crítica no puede ocasionar que otro proceso no entre a su sección crítica.

## 3.6 Segundo intento

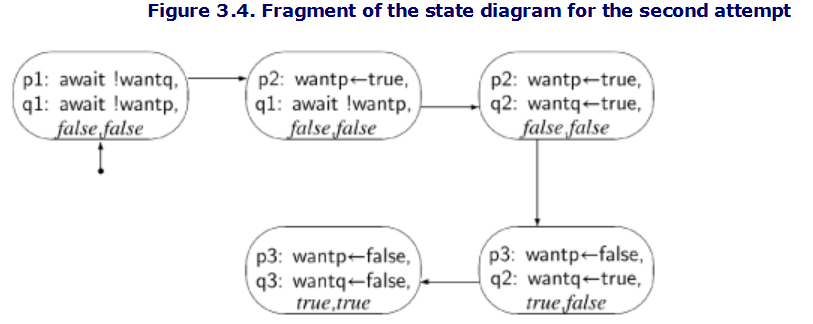
El primer intento fue incorrecto porque ambos procesos establecen y prueban una sola variable global. Si un proceso muere, el otro es bloqueado. Tratemos de resolver el problema de la sección crítica proporcionando a cada proceso su propia variable (Algoritmo 3.6). La interpretación que se dará a las variables es que wanti es verdadera desde el paso donde el proceso i quiere entrar a su sección crítica hasta que éste deja su sección crítica. Las sentencias await aseguran que un proceso no entra a su región crítica mientras otro proceso tiene su bandera establecida. Esta solución no sufre del problema de la anterior: si un proceso se cuelga en su sección no crítica el valor de su variable want permanecerá en falso y los otros procesos siempre tendrán éxito en entrar inmediatamente a su sección crítica.



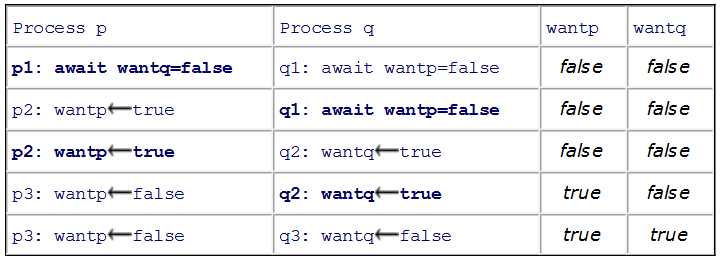
Construyamos un diagrama de estados para el algoritmo abreviado:



Desafortunadamente cuando comenzamos a construir incrementalmente el diagrama de estados (Figura 3.4), rápidamente encontramos el estado %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p3:wantp\leftarrow false, q3:wantq\leftarrow false, true,true)
\]
\end{document}, lo cual muestra que la propiedad de exclusión mutua no se cumple. (se usa %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
await\,!want
\]
\end{document} en lugar de %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
await\,\, want = false
\]
\end{document} debido a las restricciones de espacio en el nodo.)

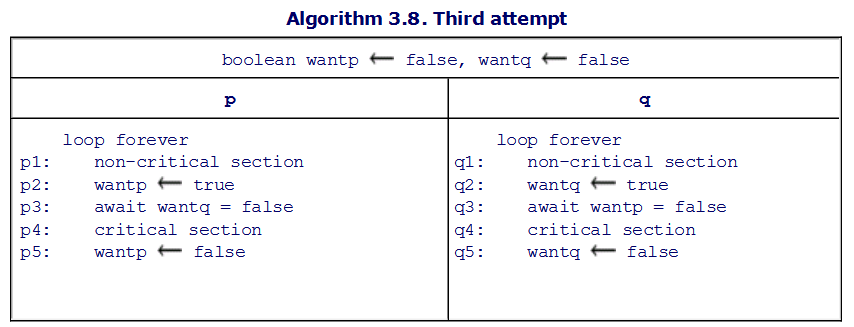


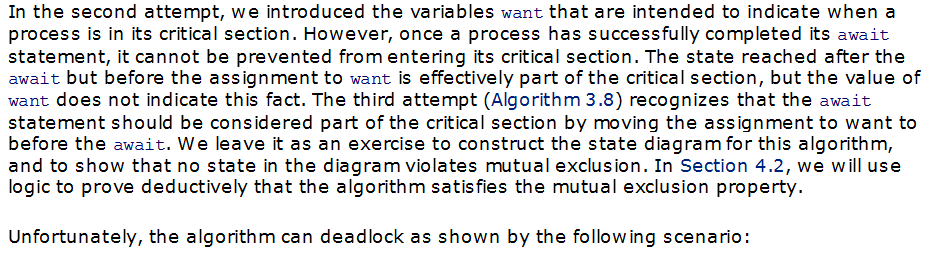
Las rutas en el diagrama de estados corresponden a escenarios; esta porción de escenario también puede ser desplegada en forma tabular:

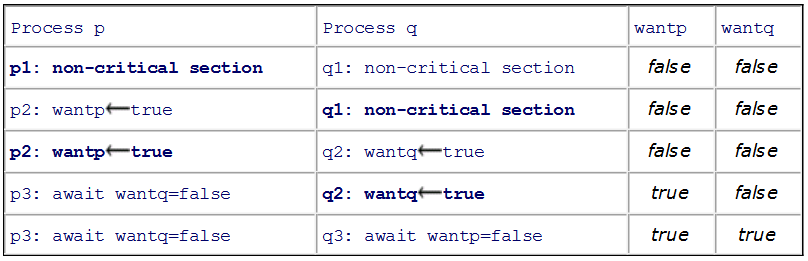


Para probar que la exclusión mutua se cumple, debemos revisar que ningún estado prohibido aparece en cualquier escenario. Así que, si la exclusión mutua de verdad se cumple, necesitamos construir el diagrama de estados completo para el algoritmo, porque cada ruta en el diagrama es un escenario; entonces debemos examinar cada estado para estar seguros de que no es un estado prohibido. Cuando se está construyendo el diagrama incrementalmente, revisamos cada estado conforme va siendo creado, así que podemos detener la construcción si se encuentra un estado prohibido, como en esta ocasión fue el caso. Se sigue que si la construcción incremental es completada (esto es, si no hay más estados alcanzables), entonces sabemos que ningún estado prohibido ocurre.

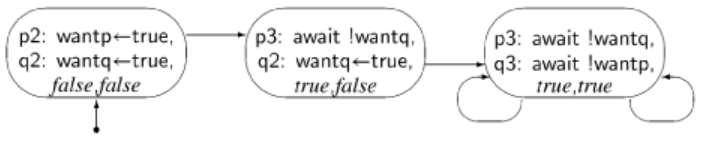
## 3.7 Tercer Intento

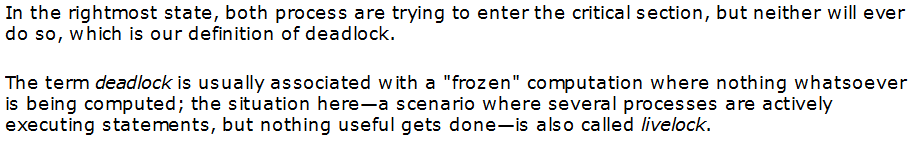




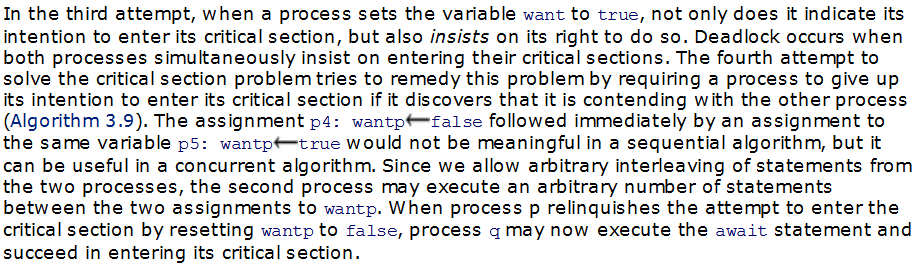


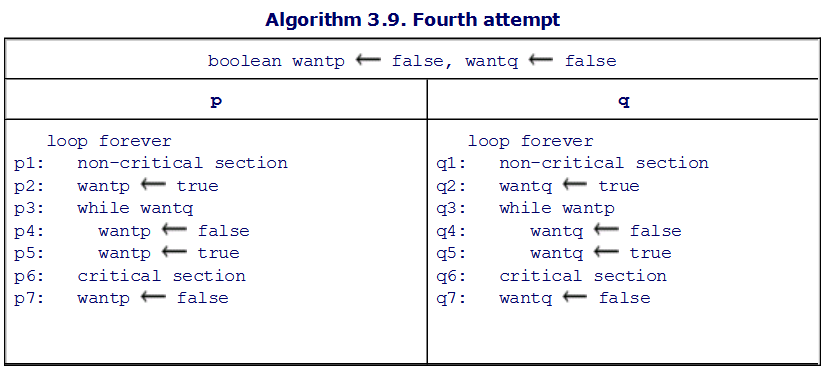
Esto también puede ser visto en el siguiente fragmento del diagrama de estados:

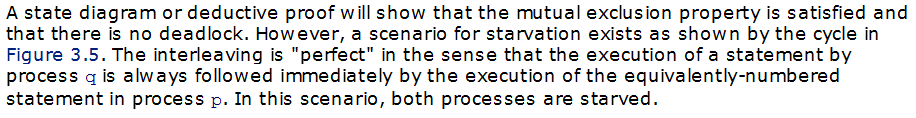


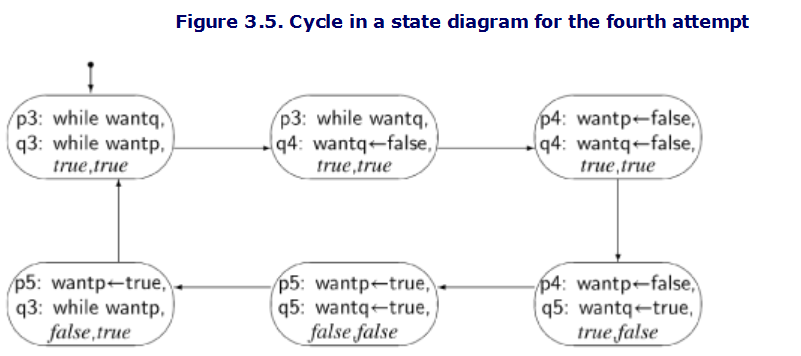


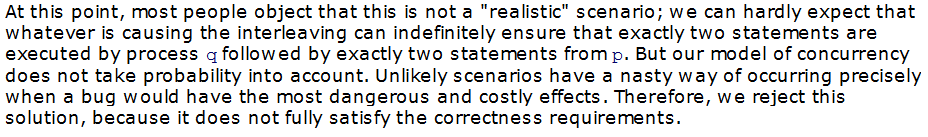
## 3.8 Cuarto Intento





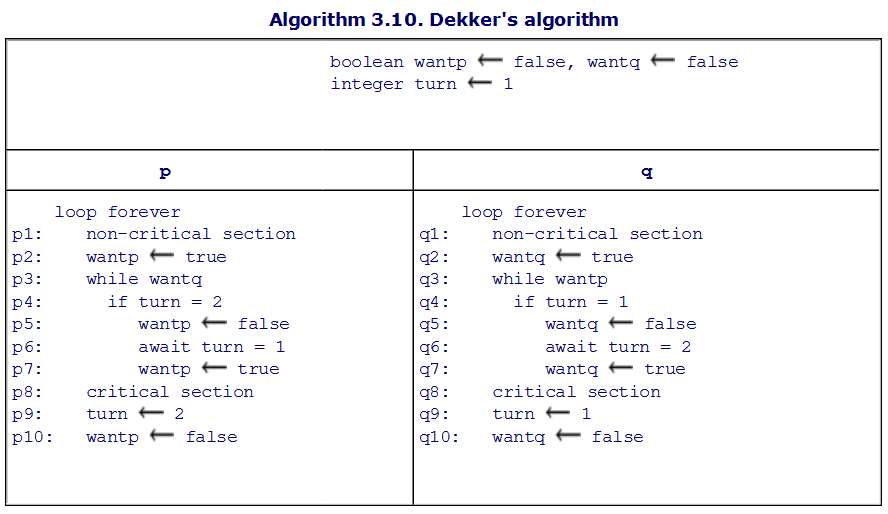






## 3.9 Algoritmo de Dekker

El algoritmo de Dekker para resolver el problema de la sección crítica es una combinación del primero y cuarto intentos:



Recordemos que en el primer intento explícitamente se pasaba el derecho a entrar a la sección crítica entre los procesos. Desafortunadamente, esto causaba que los procesos estuvieran demasiado acoplados evitando un comportamiento correcto en la ausencia de contención. En el cuarto intento, cada proceso tenía su propia variable la cual prevenía problemas en la ausencia de contención, pero en la presencia de contención ambos procesos insisten en entrar a sus secciones críticas.

El algoritmo de Dekker es como la cuarta solución intentada, excepto que el derecho a insistir en entrar, y no el derecho a entrar, es pasado explícitamente entre los procesos. Las variables individuales wantp y wantq aseguran exclusión mutua. Suponga ahora que el proceso p detecta contención encontrando que wantq es true en la sentencia p3: while wantq. El proceso p ahora consultará la variable global turn para ver si es su turno de insistir para entrar a su sección crítica (turn %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\neq
\]
\end{document} 2, esto es, turno = 1). Si es así, ejecuta el ciclo en p3 y p4 hasta que el proceso q resetea wantq a false, ya sea terminando su sección crítica en q10 o posponiendo la terminación de esta en q5. Si no, el proceso p reseteará wantp a false y cede el paso al proceso q, esperando hasta que ese proceso cambia el valor de turn después de ejecutar su sección crítica.

El algoritmo de Dekker es correcto: satisface la propiedad de exclusión mutua y está libre de deadlock e inanición. Las propiedades de corrección pueden ser demostradas construyendo y analizando un diagrama de estados, pero en lugar de eso se dará una demostración deductiva en la sección 4.5.

## 4.1 Especificación Lógica de Propiedades de Corrección

Se asume que el lector entiende los conceptos básicos del cálculo proposicional, del cual se presenta un resumen en el apéndice B.

Las proposiciones atómicas serán proposiciones acerca de los valores de las variables y los apuntadores de control (contadores de programa) durante una ejecución de un programa concurrente. Dada una variable de tipo booleano tal como wantp, la proposición atómica *wantp* es verdadera en un estado si y solo si el valor de la variable %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
wantp
\]
\end{document} es verdadero en ese estado. Nótese que se están usando dos fuentes diferentes para denotar las variables de programa y los símbolos asociados en lógica. Similarmente, las relaciones booleano-valuadas sobre variables aritméticas son proposiciones: %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
turn\neq 2
\]
\end{document} es verdadera en un estado si y solo si el valor de la variable turn en ese estado no es 2.

Cada etiqueta de una sentencia de un proceso será usada como una proposición atómica cuya interpretación es “el apuntador de control de ese proceso está actualmente en esa etiqueta.” Nuevamente, usamos fuentes para distinguir entre la etiqueta p1 y la proposición %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p1
\]
\end{document} que podría ser verdadera o falsa, dependiendo del estado de la computación. Por supuesto, dado que el apuntador de control de un proceso puede solamente apuntar a una sentencia en cualquier instante de tiempo, si %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{i}
\]
\end{document} es verdadera, entonces %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{j}
\]
\end{document} es falsa para toda %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
j\neq i
\]
\end{document} ; se usará este hecho implícitamente en las demostraciones.

Para dar un ejemplo concreto, escribamos algunas fórmulas relacionadas al algoritmo 3.8, el tercer intento. La fórmula

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{1}\wedge q_{1}\wedge\lnot wantp\wedge\lnot wantq
\]
\end{document}

es verdadera exactamente en un estado, %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
(p_{1},q_{1},false,false)
\]
\end{document}, donde los apuntadores de control de ambos procesos están en sus sentencias iniciales y los valores de ambas variables son %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
false
\]
\end{document}. La fórmula es ciertamente verdadera en el estado inicial de cualquier computación (por construcción del programa), e igual de ciertamente es falsa en el segundo estado de cualquier computación (dado que %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{1}
\]
\end{document} ó %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
q_{1}
\]
\end{document} se convertirán en falsa), a menos que, por supuesto, ambos procesos se detengan en sus secciones no críticas, en cuyo caso la computación permanece en este estado indefinidamente.

Una fórmula más interesante es

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{4}\wedge q_{4}
\]
\end{document},

la cual es verdadera si los apuntadores de control de ambos procesos apuntan a sus secciones críticas; en otras palabras, si esta fórmula es verdadera, la propiedad de exclusión mutua no se satisface. Por lo tanto, el programa satisface la propiedad de exclusión mutua en los estados en los cuales

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\lnot(p_{4}\wedge q_{4})
\]
\end{document},

la negación de %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{4}\wedge q_{4}
\]
\end{document}, es verdadera.

%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{1}\wedge q_{1}\wedge\lnot wantp\wedge\lnot wantq
\]
\end{document} es un ejemplo de una fórmula que es verdadera en algunos estados (el estado inicial de la computación), pero falsa en otros estados (en la mayoría de ellos), en particular, en cualquier estado en el cual los apuntadores de control de los procesos son distintos de %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{1}
\]
\end{document} y %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
q_{1}
\]
\end{document}. Para que el programa satisfaga la propiedad de exclusión mutua, la fórmula %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\lnot(p_{4}\wedge q_{4})
\]
\end{document} debe ser verdadera en todos los estados posibles de todas las computaciones.

### Invariantes

La fórmula %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\lnot (p_{4}\wedge q_{4})
\]
\end{document} es llamada un invariante porque debe ser invariablemente verdadera en cualquier punto de cualquier computación. Los invariantes son demostrados usando inducción, no sobre los números naturales, sino sobre los estados de todas las computaciones. (Usted podría querer revisar el apéndice B.2 sobre inducción aritmética antes de leer la sección 4.2 Inductive Proofs of Invariants)

### 4.1 Lemma

En el algoritmo 3.8, la fórmula %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
A=p3..5\rightarrow wantp
\]
\end{document} es un invariante.

Una disjunción de ubicaciones consecutivas %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p_{i}\vee\ldots\vee p_{j}
\]
\end{document} es abreviada pi..j.

### 4.2 Lemma

La fórmula %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
B=wantp\rightarrow p3..5
\]
\end{document}

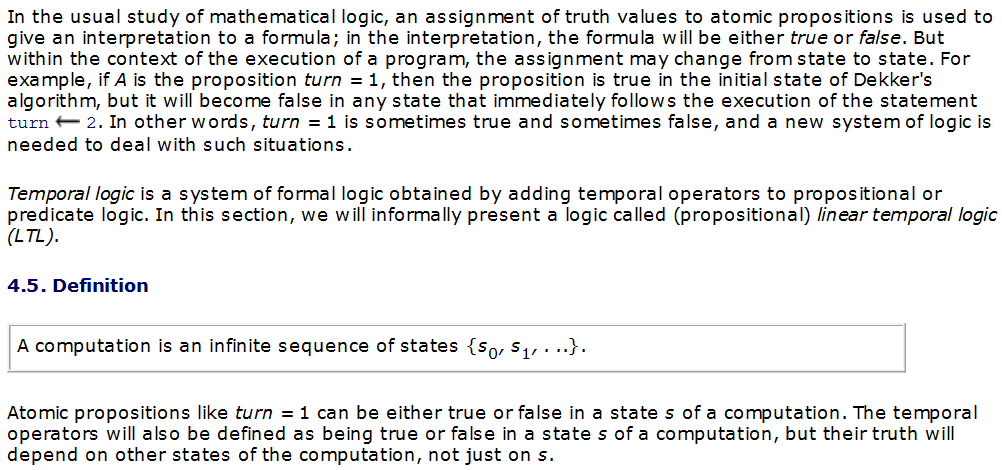
### 4.3 Lemma

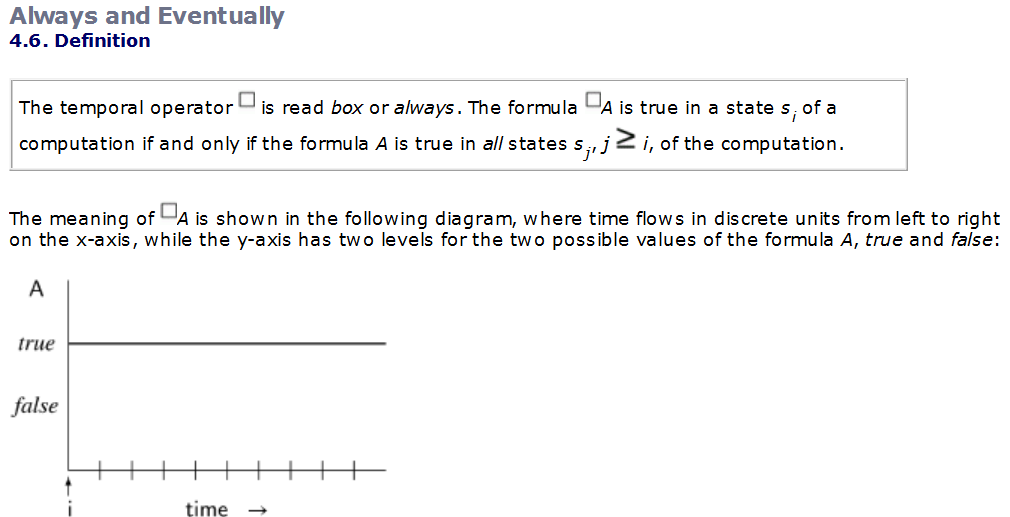
%FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
p3..5\leftrightarrow wantp
\]
\end{document} y %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
q3..5\leftrightarrow wantq
\]
\end{document} son invariantes

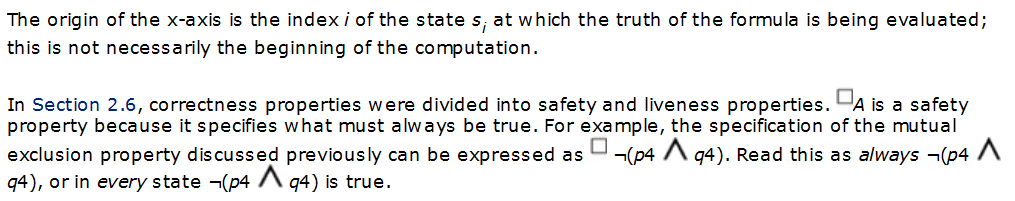
### 4.4 Teorema

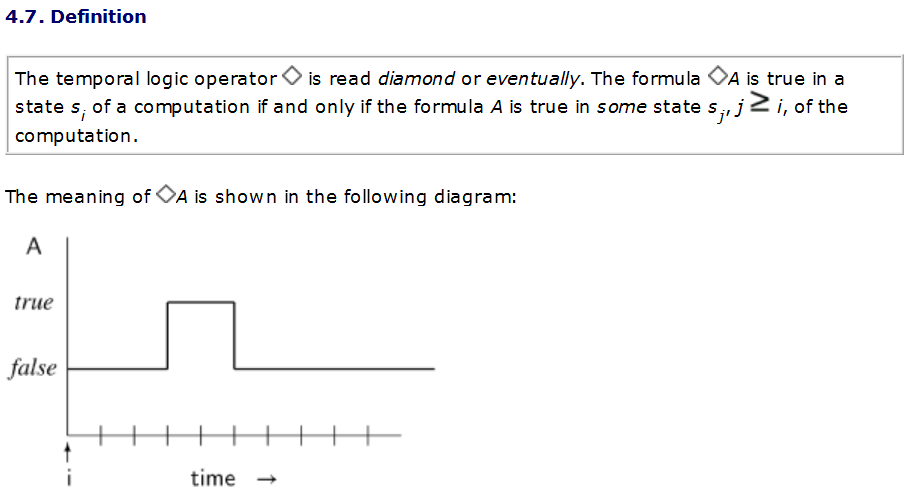
La fórmula %FontSize=10
%TeXFontSize=10
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\lnot(p_{4}\wedge q_{4})
\]
\end{document} es invariante.

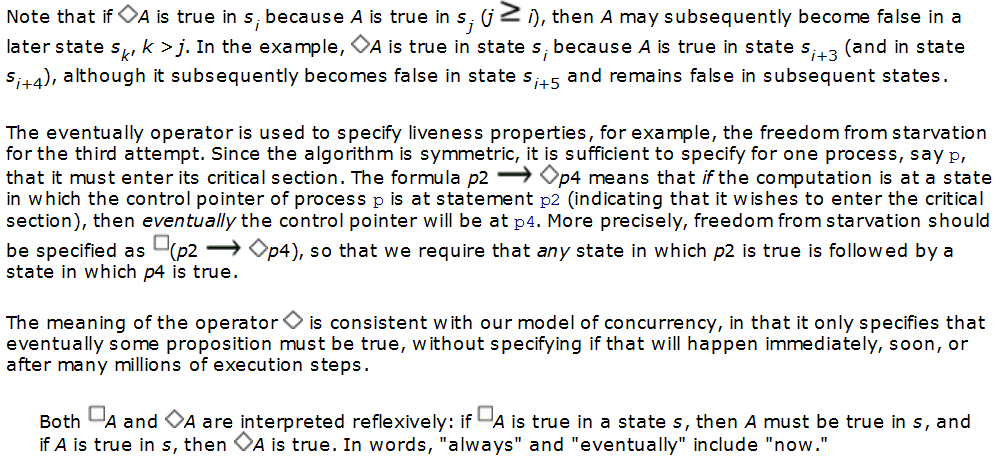
## 4.3 Conceptos Básicos de Lógica Temporal

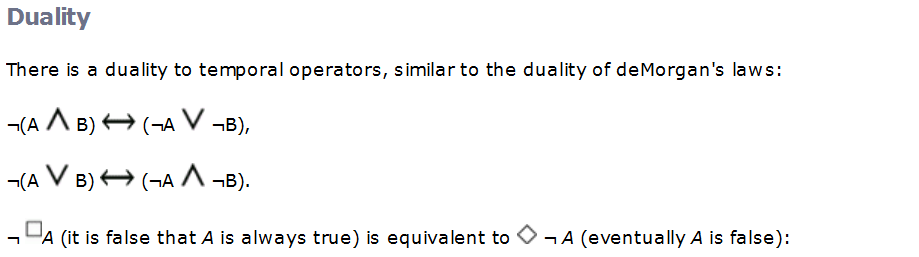


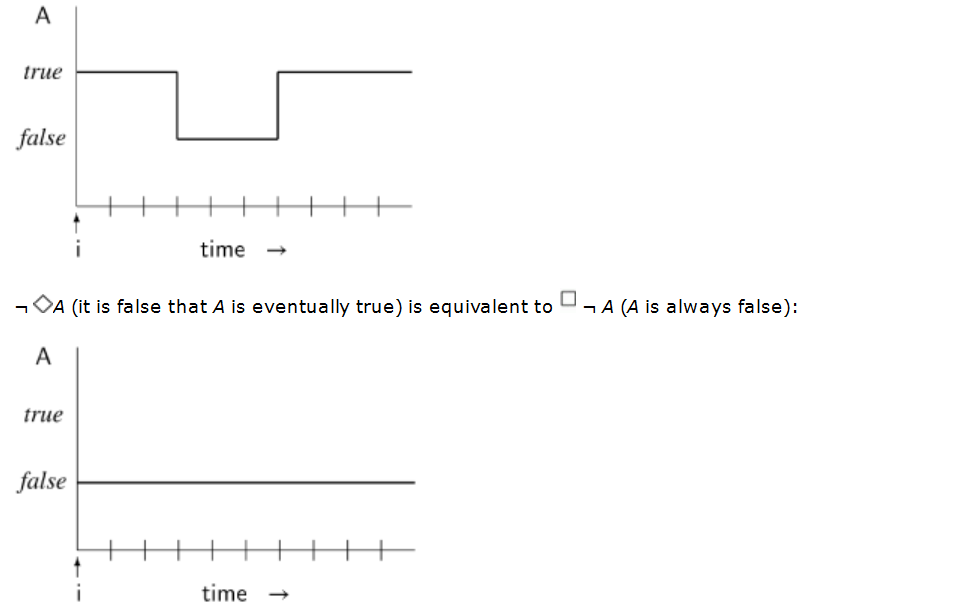


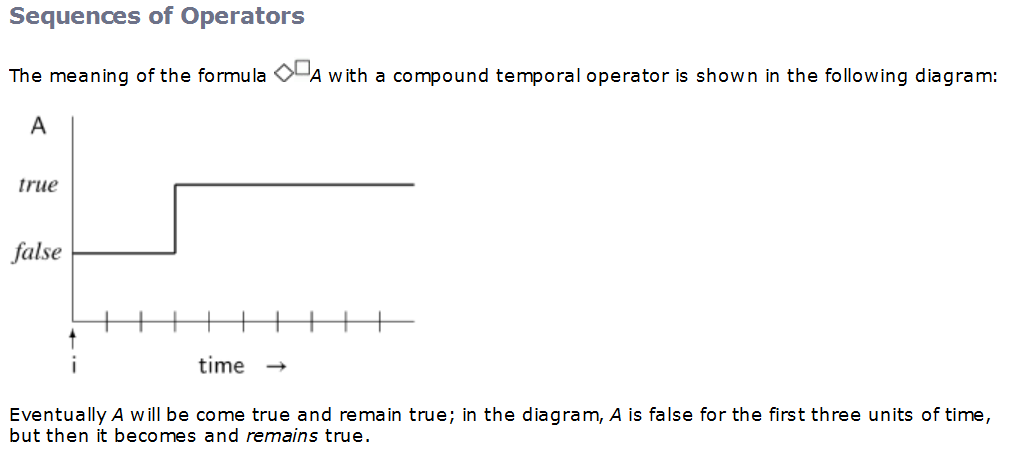


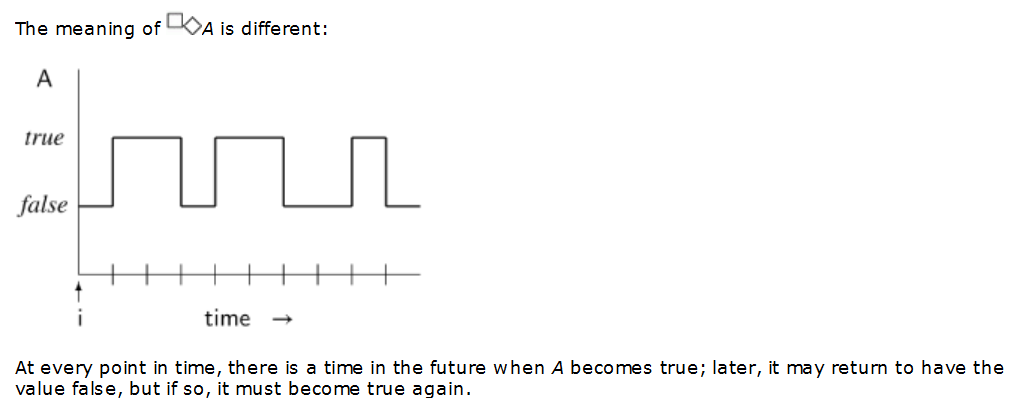




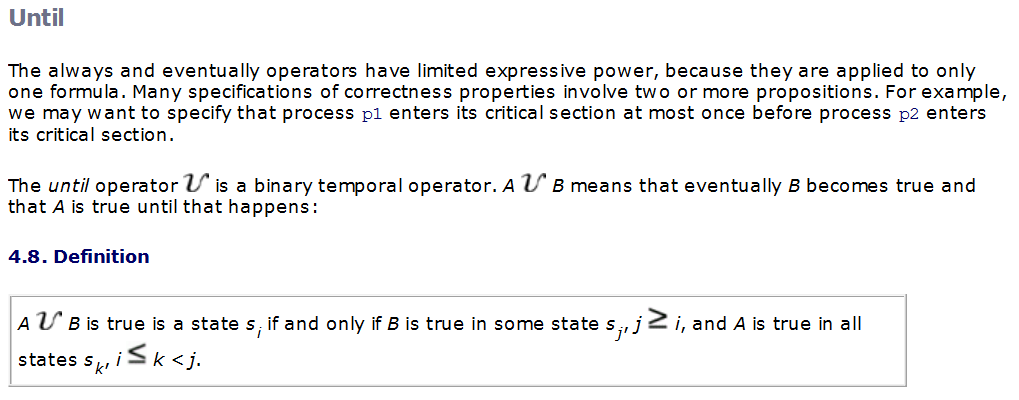


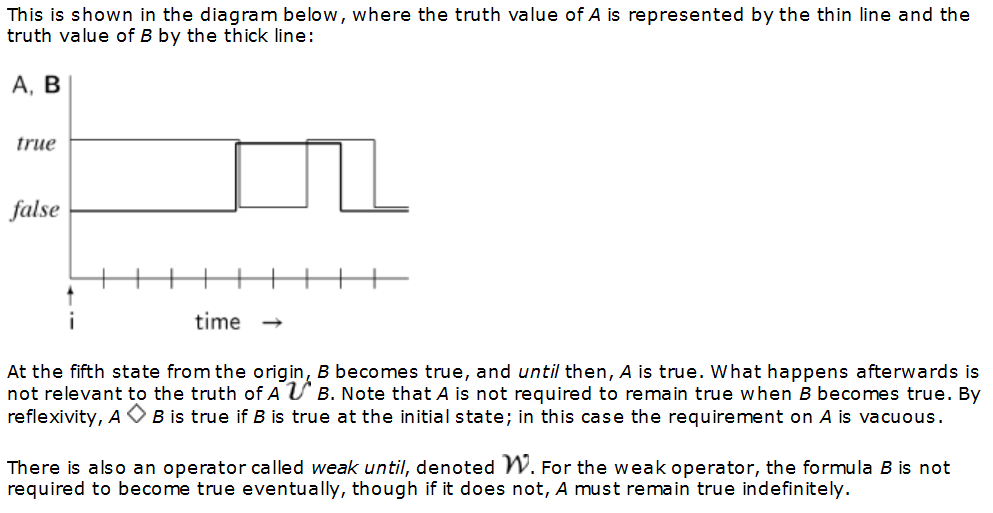


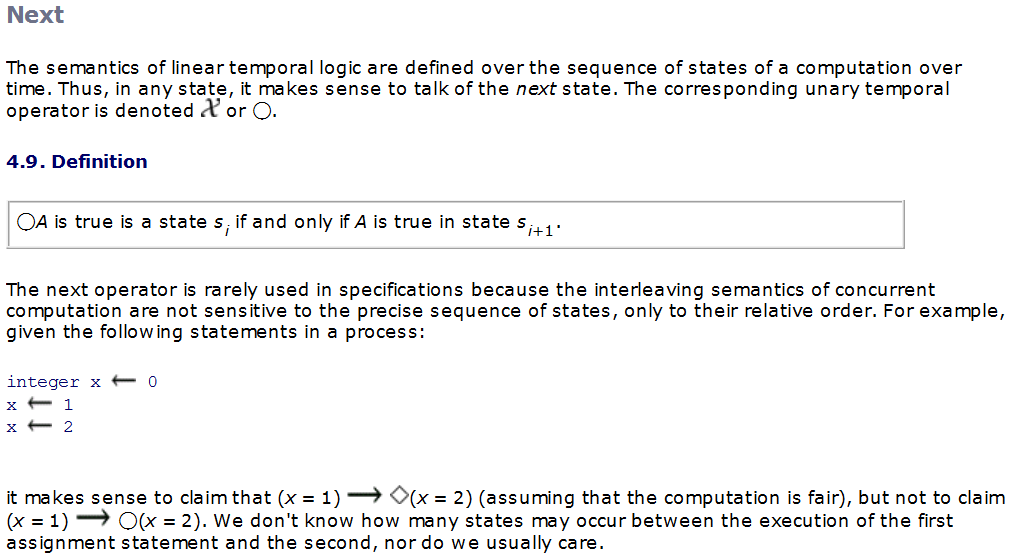


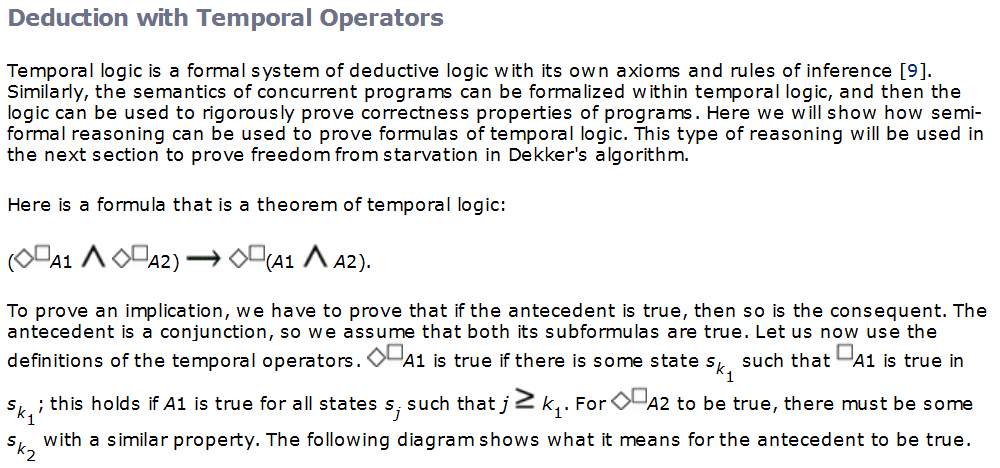


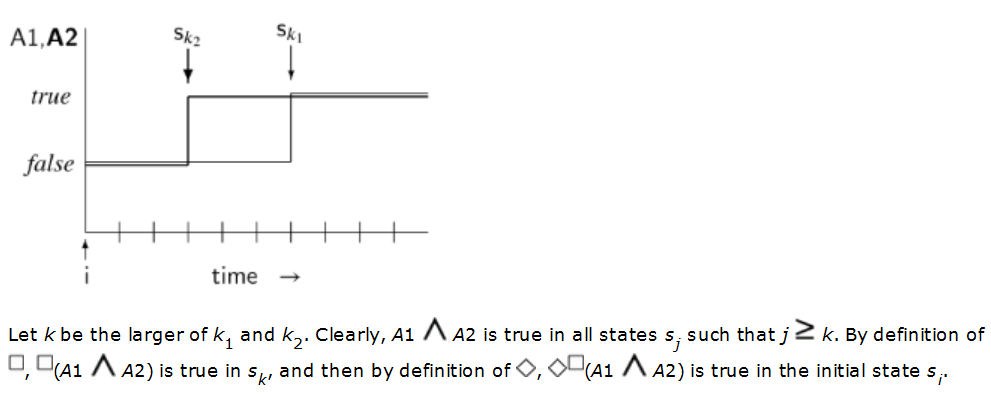
## 4.4 Conceptos Avanzados de Lógica Temporal

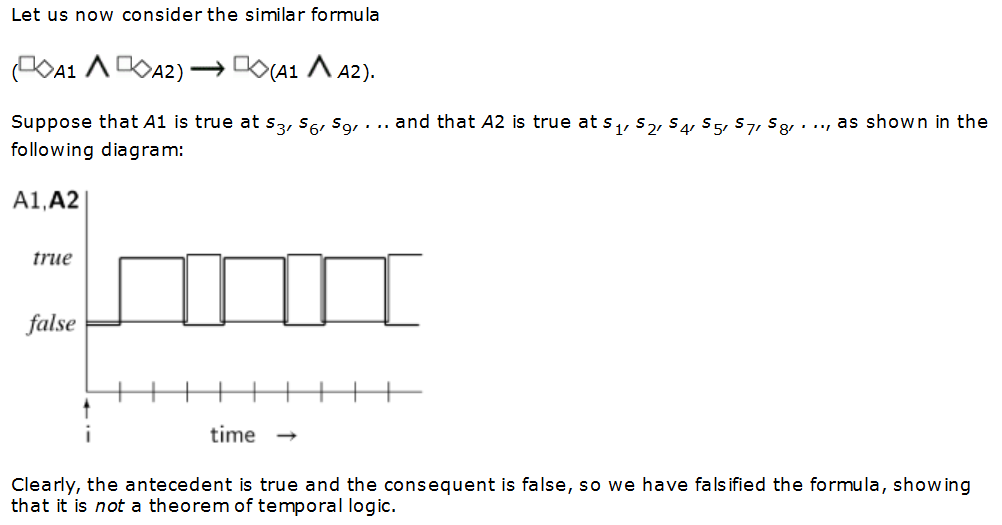


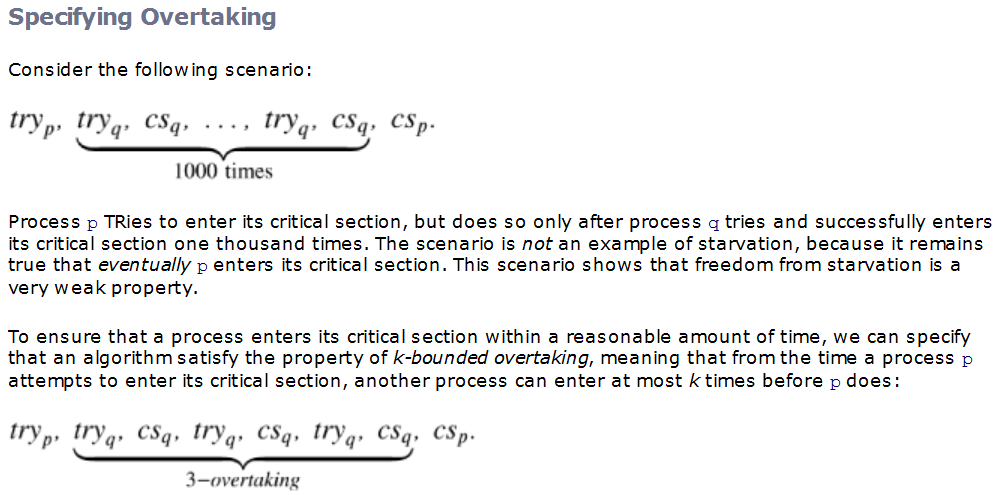


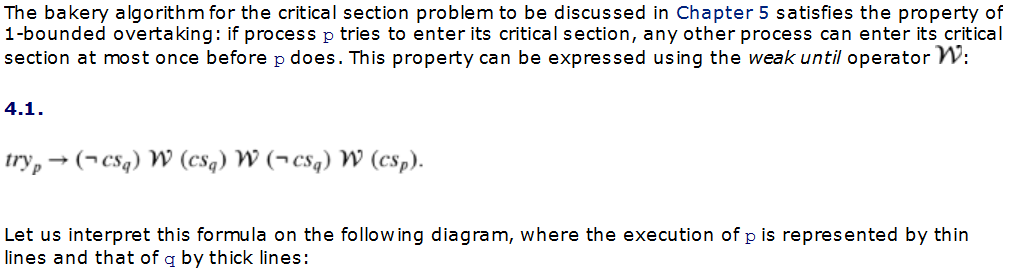


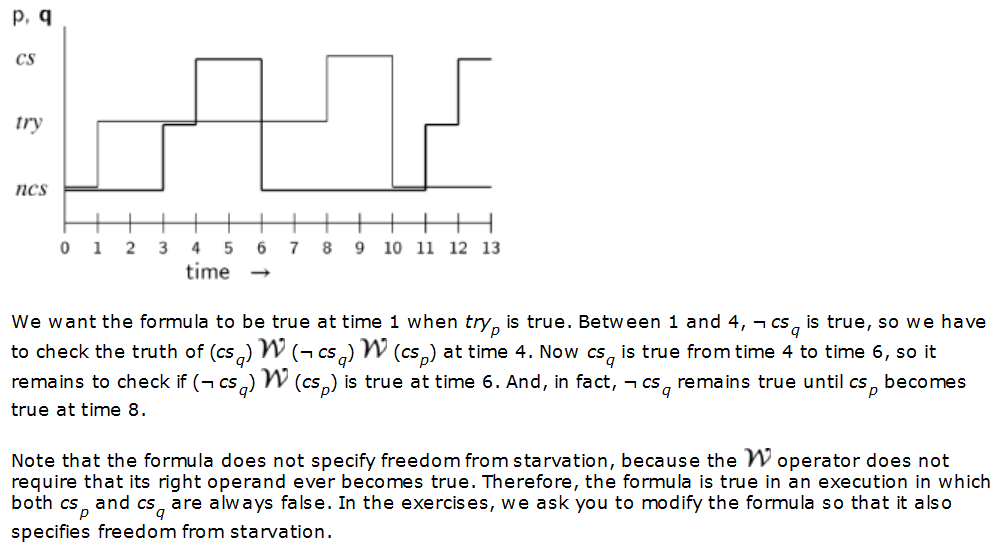












[1] M. Ben Ari, Principles of Concurrent and Distribuited Programming (Second Edition). Addison-Wesley, Harlow, UK, 2006.