PROGRAMACIÓN CONCURRENTE

Resumen

v4.1.2

UNC – FCEFyN

Ingeniería en Computación

Bonino, Francisco Ignacio

2020

ÍNDICE

– I –

Conceptos básicos

En esta materia vamos a estudiar las aplicaciones de la programación concurrente para los sistemas críticos.

- **Sistema crítico**: Sistema cuyas fallas pueden ocasionar daños de gran importancia (pérdida de vidas humanas, catástrofes ecológicas, pérdidas financieras...).

Debemos tener en cuenta que el *testing* (testeo) de programas puede confirmar la presencia de errores pero nunca garantizar su ausencia.

Muchos programas concurrentes suelen ser *reactivos*, es decir, su funcionalidad involucra la interacción permanente con el ambiente (y otros procesos). Los sistemas reactivos tienen características diferentes a las de los programas transformacionales. En muchos casos, éstos no computan resultados, y suele no requerirse que terminen. Ejemplos de sistemas reactivos son los sistemas operativos (OS), un software de control, hardware, etc.

Los programas concurrentes están compuestos por procesos (o *threads*, o componentes) que necesitan interactuar. Existen varios mecanismos de interacción entre procesos; entre estos se encuentran la memoria compartida y el pasaje de mensajes. Además, los programas concurrentes deben, en general, colaborar para llegar a un objetivo común, para lo cual la sincronización entre procesos es crucial.

Problemas comunes de los programas concurrentes

- Violación de propiedades universales (invariantes).
- ***Starvation* (inanición)**: Uno o más procesos quedan esperando indefinidamente un mensaje o la liberación de un recurso.
- ***Deadlock* (punto muerto)**: Dos o más procesos esperan mutuamente el avance del otro. Es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema o bien se comunican entre ellos.
- **Problemas de uso no exclusivo de recursos compartidos**.
- ***Livelock* (traba en vivo)**: Dos o más procesos no pueden avanzar en su ejecución porque continuamente responden a los cambios en el estado de otros procesos.

Semántica de programas concurrentes

Una semántica típica para programas concurrentes está basada en sistemas de transición de estados. Un sistema de transición de estados es un grafo dirigido en el cual:

- Los nodos son los estados del sistema (posiblemente infinitos estados).
- Las aristas son las transiciones atómicas de estados en estados, dadas por las sentencias del sistema.
- Hay un nodo distinguido que reconoceremos como el estado inicial.

Razonamiento sobre programas concurrentes

En general, es muy difícil razonar sobre programas concurrentes. Luego, garantizar que un programa concurrente sea correcto es también muy difícil. Una de las razones tiene que ver con que diferentes *interleavings* de acciones atómicas pueden llevar a diferentes resultados o comportamientos de los sistemas concurrentes. El número de interleavings posibles, por su parte, es en general muy grande, lo que hace que el testing difícilmente pueda brindarnos confianza de que nuestros programas concurrentes funcionen correctamente.

Definiciones

- **Proceso**: Una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados. Un proceso podría crear nuevos procesos (proceso padre, proceso hijo), y una vez creado un proceso hijo, la ejecución de padre e hijo transcurre de manera concurrente. Un proceso es básicamente un programa en ejecución.
- **Thread**: Es una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un OS. Un thread es simplemente una tarea que puede ser ejecutada al mismo tiempo que otra tarea. Los threads de ejecución que comparten los mismos recursos, sumados a estos recursos, son en conjunto conocidos como un proceso. Lo que es propio de cada thread es el contador de programa (PC, Program Counter), la pila de ejecución (stack) y el estado de la CPU (incluyendo el valor de los registros).

Diferencias entre threads y procesos

Los threads se distinguen de los tradicionales procesos en que los procesos son –generalmente– independientes, llevan bastante información de estados, e interactúan solo a través de mecanismos de comunicación dados por el sistema. Por otra parte, muchos threads generalmente comparten otros recursos de forma directa. En muchos de los OS que dan facilidades a los threads, es más rápido cambiar de un thread a otro dentro del mismo proceso, que cambiar de un proceso a otro. Este fenómeno se debe a que los threads comparten datos y espacios de direcciones, mientras que los procesos, al ser independientes, no lo hacen. Al cambiar de un proceso a otro, el OS (mediante el *dispatcher*[1]) genera lo que se conoce como *overhead*, que es tiempo desperdiciado por el procesador para realizar un cambio de contexto (*context switch*); en este caso, pasar del estado de ejecución (*running*) al estado de espera (*waiting*) y colocar el nuevo proceso en ejecución. En los threads, como pertenecen a un mismo proceso, al realizar un cambio de thread el tiempo perdido es casi despreciable.

---

[1]: La misión del *dispatcher* (planificador a corto plazo) consiste en asignar la CPU a uno de los procesos ejecutables del sistema. El dispatcher conmuta la CPU entre procesos.

Procesos

Estados de los procesos

- **Nuevo**/**New**: Está siendo creado.
- **En ejecución**/**Activo**/**Running**: Se está ejecutando en la CPU.
- **Bloqueado**/**Suspendido**/**Waiting**/**Blocked**/**Blocking**: Está esperando la respuesta de algún otro proceso para poder continuar con su ejecución. Espera a que se termine una operación de E/S o que se reciba una señal de sincronización.
- **Listo**/**Preparado**/**Ready**: Puede pasar a estado de ejecución. Todas las tareas están listas para ejecutarse pero se espera a que un/el procesador quede libre (hay otros procesos más prioritarios en ejecución).
- **Nonato:** El programa realmente existe pero todavía no es conocido por el OS.
- **Muerto**/**Terminated**: Ha terminado su ejecución satisfactoriamente o el OS ha detectado un error fatal.

Los estados *en ejecución*, *preparado/listo*, y *bloqueado* son considerados estados activos.

Los estados *suspendido bloqueado* (proceso que fue suspendido en espera de un evento, sin que hayan desaparecido las causas de su bloqueo) y *suspendido programado* (proceso que ha sido suspendido, pero no tiene causa para estar bloqueado) son considerados estados inactivos.

Registro de control de procesos

El bloque de control de procesos (BCP, PCB en inglés: Processes Control Block) es un registro especial donde el OS agrupa toda la información que necesita conocer respecto a un proceso particular; es decir, almacena:

- **Identificador del proceso** (PID, Process Identificator).
- **Estado del proceso**.
- **Contador de programa**: Dirección de la próxima instrucción a ejecutar.
- **Valores de registro de la CPU**: Se utilizan también en el cambio de contexto.
- **Espacio de direcciones de memoria**.
- **Prioridad:** En caso de utilizarse dicho algoritmo para planificación de la CPU.
- **Lista de recursos asignados** (incluyendo descriptores de archivos y *sockets* abiertos).
- **Estadísticas del proceso**.
- **Datos del propietario** (*owner*).
- **Permisos asignados**.
- **Signals pendientes de ser servidos** (almacenados en un mapa de bits).

Threads

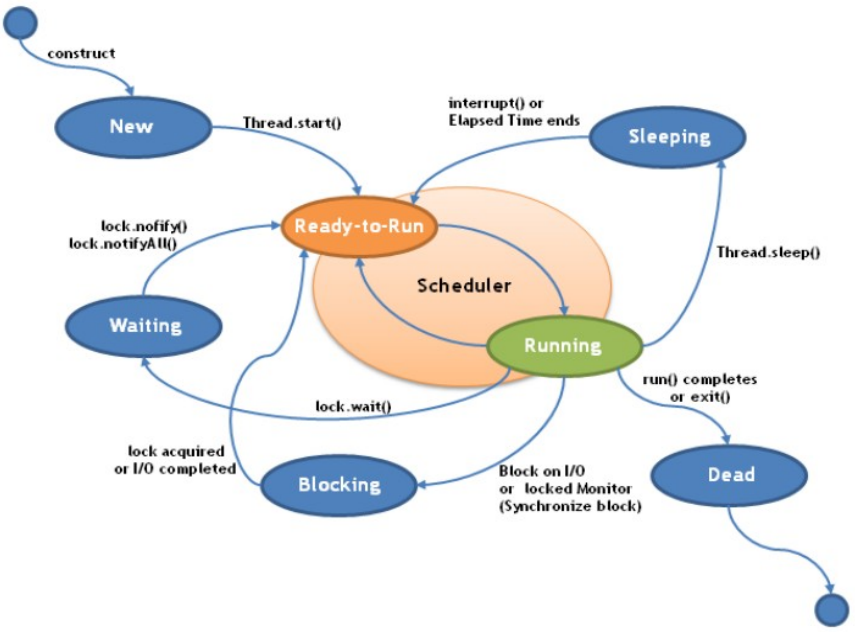
Componentes de un thread

- **Estado**.
- **Contexto del procesador**: Punto en el que estamos ejecutando la instrucción. Se usa para reanudar un thread que fue interrumpido, para poder continuar su ejecución.
- **Pila de ejecución**: Donde se irán metiendo y sacando instrucciones (lugar donde almacenaremos las instrucciones que van a ser ejecutadas).
- **Espacio de almacenamiento estático**: Donde almacenará las variables.
- **Acceso a los recursos de la tarea que son compartidos por todos los threads de la tarea**.

Estados posibles de los threads

- **Ejecución, listo y bloqueado**: Mismo concepto que para los procesos.
- **Cambio de estados**:
  - **Creación**: Cuando se crea un proceso se crea un thread para ese proceso. Luego, este thread puede crear otros threads dentro del mismo proceso. El thread tendrá su propio contador, registros, espacio de pila, y pasará a la cola de *threads listos*.
  - **Bloqueo**: Cuando un thread necesita esperar por un suceso, se bloquea (salvando sus registros de usuario, contador de programa y punteros de pila). Ahora el procesador podrá pasar a ejecutar otro thread que esté en la cola de *threads listos* mientras el anterior permanece bloqueado.
  - **Desbloqueo**: Cuando el suceso por el que el thread se bloqueó se produce, el mismo pasa a la cola de *threads listos*.
  - **Terminación**: Cuando un thread finaliza, se liberan tanto su contexto como sus pilas.

Gráficamente:



Hay que tener en cuenta que no se puede optimizar un programa agregando tantos threads como uno quiera. Hay un límite que está impuesto por la cantidad de *cores* de la PC (si una PC tiene 4 cores, entonces 4 threads físicos podrán ser utilizados sin problema; si usamos más, estaremos usando threads virtuales y puede que en algunos casos no ande mejor, sino que presenten fallas en el interleaving).

Ventajas de los threads contra procesos

- Se tarda mucho menos tiempo en crear un thread nuevo en un proceso existente que en crear un proceso (algunas investigaciones llegan al resultado de que esto es así en un factor de 10).
- Se tarda mucho menos tiempo en terminar un thread que un proceso.
- Cuando se elimina un proceso, se debe eliminar el BCP del mismo, mientras que con un thread se elimina sólo su contexto y su pila.
- Se tarda mucho menos tiempo en cambiar entre dos threads de un mismo proceso que entre dos procesos distintos.
- Los threads aumentan la eficiencia de la comunicación entre programas en ejecución.
- En la comunicación entre procesos debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación misma.
- La comunicación entre threads no requiere la invocación al núcleo; por lo tanto, si hay una aplicación que debe implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de threads que con una colección de procesos separados.
- Cuando un thread está en ejecución, posee el acceso a todos los recursos que tiene asignados la tarea.

Conceptos de concurrencia

- **Multiprogramación**: Consiste en la gestión de varios procesos dentro de un sistema monoprocesador.
- **Multiprocesamiento**: Consiste en la gestión de varios procesos, dentro de un sistema multiprocesador.
- **Procesamiento distribuido**: Consiste en la gestión de varios procesos, ejecutándose en sistemas de computadores múltiples y distribuidos.

Diferencias entre programación concurrente y paralela

La programación concurrente consiste en trabajar con threads que hagan interleaving (comparten recursos, se turnan de forma muy rápida para utilizarlos y da la sensación de paralelismo cuando en realidad no lo es).

La programación paralela es, por el contrario, literalmente la ejecución de procesos utilizando recursos distintos para lograr un paralelismo real.

En la programación concurrente, los problemas son consecuencia de la velocidad de ejecución de los procesos que no pueden predecirse y dependen de las actividades de otros procesos y de la forma en que el OS trata las interrupciones.

Sincronización

La comunicación de un mensaje entre dos procesos implica cierto nivel de sincronización entre ambos. El receptor no puede recibir un mensaje hasta que sea enviado por otro proceso. Además, hace falta especificar qué le sucede a un proceso después de ejecutar una primitiva SEND o RECEIVE. Aclaramos que una primitiva se define como una instrucción indivisible.

Cuando se ejecuta una primitiva SEND en un proceso, hay dos posibilidades: o bien el proceso emisor se bloquea hasta que recibe el mensaje, o no se bloquea.

Igualmente, cuando un proceso ejecuta una primitiva RECEIVE, existen dos opciones:

- Si previamente se ha enviado algún mensaje, éste es recibido y continúa la ejecución.
- Si no hay ningún mensaje, puede pasar alguna de las siguientes situaciones:
  - El proceso se bloquea hasta que llega un mensaje.
  - El proceso se continúa ejecutando, abandonando el intento de recepción.

El emisor y el receptor pueden ser *bloqueantes* o *no bloqueantes*.

Existen tres tipos de combinaciones:

- **Envío bloqueante, recepción bloqueante**: Tanto el emisor como el receptor se bloquean hasta que llega el mensaje; esta técnica se conoce como *rendezvous*.
- **Envío no bloqueante, recepción bloqueante**: Aunque el emisor puede continuar, el receptor se bloquea hasta que llega el mensaje solicitado. Es la combinación más útil.
- **Envío no bloqueante, recepción no bloqueante**: Nadie debe esperar nada.

El SEND no bloqueante es la forma más natural para muchas tareas de programación concurrente. Un posible riesgo del SEND no bloqueante es que por error puede llevar a una situación en la que el proceso genere mensajes repetidamente.

Para el RECIEVE, la versión bloqueante es la más natural para muchas tareas de programación concurrente. En general, un proceso que solicita un mensaje necesitará la información esperada antes de continuar

Resumen de términos clave en concurrencia

- **Operación atómica (atomic operation)**: Las operaciones atómicas son aquellas operaciones que siempre se ejecutan hasta terminarse completamente. O todas ellas se ejecutan juntas, o ninguna de ellas se ejecuta. Si una operación es atómica, entonces no puede estar parcialmente completa; estará completa o no comenzará en absoluto, pero no estará incompleta.
- **Sección crítica (critical section)**: Los recursos compartidos pueden conducir a un comportamiento inesperado o erróneo, por lo que las partes del programa donde se accede al recurso compartido deben protegerse de manera que se evite el acceso concurrente. Esta sección protegida es la sección crítica o región crítica. No puede ser ejecutada por más de un proceso/thread a la vez.
- **Exclusión mutua (mutual exclusion)**: Es una propiedad para el control de concurrencia que se instituye con el propósito de prevenir las *condiciones de carrera* . Es el requisito de que un thread en ejecución nunca entre en la sección crítica al mismo tiempo en el que otro thread la esté ejecutando.
- **Condición de carrera (race condition)**: Concepto que aparece cuando el resultado de la ejecución de varios threads concurrentes puede ser correcto o incorrecto dependiendo de cómo se entrelacen los threads. Básicamente, el problema aparece cuando varios threads hacen ciclos de modificación de una variable compartida (lectura + modificación + escritura).
- **Punto muerto (deadlock)**: Un punto muerto es un estado en el que cada miembro de un grupo de procesos/threads está esperando que otro miembro, incluido él mismo, tome medidas (como enviar un mensaje o, más comúnmente, liberar un bloqueo). El deadlock es un problema común en los sistemas de multiprocesamiento, la computación paralela y los sistemas distribuidos, donde se utilizan bloqueos de software y hardware para arbitrar los recursos compartidos e implementar la sincronización de procesos.
- **Traba en vivo (livelock)**: Se producen cuando dos o más procesos repiten continuamente la misma interacción en respuesta a cambios en los otros procesos sin realizar ningún trabajo útil. Estos procesos no están en estado de espera y se ejecutan simultáneamente. Esto es diferente de un deadlock porque en un deadlock todos los procesos están en estado de espera.
- **Inanición (starvation)**: Es un problema que se encuentra en programación concurrente donde a un proceso se le niega perpetuamente recursos para procesar su trabajo. La inanición puede ser causada por errores en un algoritmo de programación o exclusión mutua, pero también puede ser causada por fugas de recursos, y puede ser causada intencionalmente a través de un ataque de denegación de servicio.

– II –

Manejo básico de threads

Creación

Todo ejemplo de programación en este resumen será desarrollado en lenguaje Java.

Heredando de Thread

Esta es, quizás, la forma más intuitiva de crear un thread. Se debe crear una clase y hacerla heredar de la súper clase ***Thread***, sobrescribiendo el método ***run()***; luego, creamos un objeto de tipo *[clase creada]* y lo instanciamos. Esto es de la forma:

```
public class myThread extends Thread {

	//Código...

	@Override
	public void run() {
		//Código...
	}

	//Código...
}

public class Main() {

	//Código...

	public static void main(String[] args) {
		myThread thread1 = new myThread();

		//Código...
	}

	//Código...
}
```

Otra forma de hacerlo es extendiendo de Thread y, en la clase Main, creamos un objeto de tipo Thread y lo instanciamos con la clase creada. Esto es:

```
public class myThread extends Thread {

	//Código...

	@Override
	public void run() {
		//Código...
	}

	//Código...
}

public class Main() {

	//Código...

	public static void main(String[] args) {
		Thread thread1 = new myThread();

		//Código...
	}

	//Código...
}
```

Implementando la interfaz Runnable

La segunda forma de hacerlo es creando una clase que implemente la interfaz ***Runnable***, sobrescribiendo el método ***run()***; luego, creamos un objeto de tipo **Thread** y lo instanciamos asignándole un objeto de tipo *[clase creada]*.

Ejemplo:

```
public class myThread implements Runnable {

	//Código...

	@Override
	public void run() {
		//Código...
	}

	//Código...
}

public class Main() {

	//Código…

	public static void main(String[] args) {
		Thread thread1 = new Thread(new myThread());

		//Código...
	}

	//Código...
}
```

Ambas formas de crear threads son válidas y funcionan exactamente igual.

Ejecución

Sea como sea que hayamos creado los threads (heredando de *Thread* o implementando *Runnable*), la forma de ejecutarlos es la misma para ambos.

Hay dos formas de ejecutar un thread: con el método ***run()*** o el método ***start()***.

Si lo ejecutamos con el método **start()**, el **main thread** del programa le indicará al thread que tiene que trabajar (siempre que un thread comienza a trabajar, empieza por la ejecución del método *run()* sobrescrito), y le asigna un PC aparte, independizándose uno del otro. Acto seguido, el main thread continúa la ejecución del método main de la clase Main.

Si, por el contrario, lo ejecutamos con el método **run()**, el main thread será el que ejecute el método *run()* de la clase, para luego volver al método main de la clase Main y seguir ejecutando lo que sigue (ésto logra que no aprovechemos la concurrencia, sino que hagamos todo secuencial).

Prioridad

Los threads tienen **prioridad**. Ésta es un número $x \mid x \in [1, 10] \land x \in \mathbb{N}$, el cual establece, valga la redundancia, la prioridad que tendrá el thread a la hora de ser ejecutado.

Existen constantes dentro de la súper clase Thread que indican los niveles de prioridad "notables":

- MAX_PRIORITY = 10. Ésta constante indica que aquellos threads que tengan establecida como prioridad el valor 10 tendrán primacía para con los demás threads.
- NORM_PRIORITY = 5. Valor medio de prioridad.
- MIN_PRIORITY = 1. Aquellos threads con prioridad 1 o cercana a este valor, serán los que tengan menor prioridad a la hora de elegir cuáles threads ejecutar.

Interrupciones

Un programa en Java con más de un thread en ejecución sólo finaliza cuando todos sus threads (no *dæmons*) terminan su ejecución. Sin embargo, puede ser necesario finalizar la ejecución de sólo un thread en particular; por ejemplo, si queremos terminar un programa o finalizar la tarea que el thread está haciendo. Java provee un mecanismo para indicarle al thread nuestra voluntad (*gentil*) de detenerlo. Una peculiaridad del mecanismo es que el thread puede ignorar la petición y continuar trabajando.

Esta interrupción es, en código, de la forma:

thread.interrupt();

Si queremos saber si un thread recibe una petición para ser interrumpido, existe el método ***isInterrupted()***, que devuelve TRUE si fue interrumpido, o FALSE si no se solicitó interrupción.

En el caso de que se interrumpa un thread, será lanzada una excepción de tipo ***InterruptedException*** que deberá ser *catcheada* en un bloque *try-catch-finally*.

El mecanismo visto para la interrupción de un thread es adecuado para programas simples, sin recursividad ni complejidad en cantidad de métodos. La interrupción por excepción InterruptedException puede ser lanzada al detectar la interrupción del thread y hacer el catch de la misma desde el método *run()*.

En diferentes ocasiones es necesario suspender la ejecución de un thread sólo por un determinado tiempo. Por ejemplo, un thread que lee un sensor y luego, durante un minuto, no realiza tarea alguna. Puede utilizarse el método ***sleep()*** de la clase Thread para este propósito. Este método recibe un número entero como argumento, que representa el tiempo, en milisegundos, que el thread suspende su ejecución. Cuando el tiempo de *sleep()* finaliza, el thread continúa con la ejecución de la instrucción siguiente a *sleep()* cuando la JVM le asigne tiempo de CPU. Otra posibilidad es utilizar el método *sleep()* con un argumento del enumerado ***TimeUnit*** (en vez de ingresar el tiempo de suspensión en milisegundos, podemos ingresarlo en segundos o minutos).

A veces necesitamos esperar a que un thread finalice su ejecución. Por ejemplo: un programa que comience inicializando los recursos antes de comenzar con el resto de la ejecución. Para este propósito podemos usar el método ***join()*** de la clase Thread. Cuando llamamos a este método usando un objeto Thread, suspende su ejecución hasta que finalice la ejecución del Thread llamado. Es decir, si ejecutamos las siguientes líneas de código:

```
        //Código...

n       thread1.start();
n+1     thread1.join();
n+2
n+3     thread2.start();
n+4     thread2.join();
n+5
n+6     System.out.println("Ambos threads han terminado de ejecutarse");
```

La línea de código *n* indica al main thread que debe despertar a thread1 para que comience su ejecución, y en la línea *n+1* le indicamos al main thread que no debe moverse de allí hasta que thread1 termine su ejecución. Cuando esto suceda, el main thread procederá con las siguientes líneas de código, consiguiendo así que los threads se ejecuten uno a la vez y en orden.

Teniendo en cuenta todo lo explicado hasta el momento sobre interrupciones, podemos introducir el concepto de código ***thread–safe***:

- **Thread safe**: Característica que brinda la posibilidad de ejecutar el código en un entorno con múltiples threads garantizando consistencia.

Aquello que necesitamos para lograr que nuestro código sea thread safe es la **sincronización**. Se explica en el capítulo siguiente.

– III –

Sincronización basada en memoria compartida

Cuando hablamos de sincronización, estamos hablando del concepto de exclusión mutua aplicado en la programación concurrente.

Podemos lograr esta exclusión mutua de varias formas:

- **Por medio de establecimiento de secciones críticas (*synchronized*)**: Son mecanismos de nivel medio de abstracción orientados a su implementación en el contexto de un lenguaje y que permiten la ejecución de un bloque de sentencias de forma segura.
- **Por medio de semáforos**: Los semáforos tienen una variable entera no negativa que indica la cantidad de permisos (cantidad máxima de threads que "podrán pasar") y se comparte entre threads. Se utiliza para resolver el problema de la sección crítica y para lograr la sincronización del proceso en entornos de multiprocesamiento. Java proporciona la clase *Semaphore* en el paquete ***java.util.concurrent*** que implementa este mecanismo.
- **Por medio de monitores**: Son módulos de alto nivel de abstracción orientados a la gestión de recursos que van a ser usados concurrentemente.

Synchronized

Todos los objetos tienen un bloqueo asociado, "***lock***" o *cerrojo*, que puede ser adquirido y liberado mediante el uso de métodos y sentencias *synchronized*. La sincronización fuerza a que la ejecución de dos (o más) threads sea mutuamente exclusiva en el tiempo.

Con los bloques synchronized logramos implementar las llamadas *secciones críticas*. Este modificador de acceso hace que cuando un thread acceda a este bloque (o método), se bloquee automáticamente el ingreso para absolutamente cualquier otro thread hasta que el bloque (o método) synchronized haya sido abandonado por el thread que lo estaba ejecutando.

Si bien es muy útil, está muy limitado y consume mucho tiempo de procesamiento, y el costo computacional es más alto que el de otros métodos de sincronización.

Podemos implementar la sentencia synchronized a un método, a un bloque de código o a una clase. Supongamos que se lo aplicaremos a un método. Esto puede hacerse de la forma:

```
public class Clase {

	//Código...

	public synchronized void metodo1(this) {
		//Código...
	}

	//Código...
}
```

Si necesitamos hacer esto con algún objeto que usaremos como lock, podemos hacerlo de las siguientes dos formas:

Forma 1

```
public class Clase {

	//Código...

	public synchronized void metodo1(Object object) {
		//Código...
	}

	//Código...
}
```

Forma 2

```
public class Clase {

	//Código...

	private Type objetoLock = new Type();

	public synchronized void metodo1(objetoLock) {
		//Código...
	}

	//Código...
}
```

De esta forma logramos que todo lo que esté encerrado en el método *metodo1* se vuelva atómico. Así podemos proteger variables que pueden ser corrompidas por el acceso simultáneo de dos o más threads a la critical section.

Si el método synchronized es estático (*static*), el lock al que hace referencia es de clase y no de objeto, por lo que se hace en exclusión mutua con cualquier otro método estático synchronized de la misma clase.

Consideraciones sobre synchronized

- El lock es tomado por el thread, por lo que mientras un thread tiene tomado el lock de un objeto puede acceder a otro método synchronized del mismo objeto.
- El lock es por cada instancia del objeto.
- Los métodos de clase (static) también pueden ser synchronized.
  - Por cada clase hay un lock y es relativo a todos los métodos synchronized de la clase.
  - Este lock no afecta a los accesos a los métodos synchronized de los objetos que son instancia de la clase.
- Cuando una clase se extiende y un método se sobrescribe, éste se puede definir como synchronized o no, con independencia de cómo era y cómo sigue siendo el método de la clase madre.

Recursos para sincronizar threads

Algunos métodos básicos para utilizar en un contexto de sincronización de threads son:

- **public final void wait()** */throws InterruptedException/*: Le dice al thread de llamada que abandone el bloqueo y se vaya a dormir hasta que algún otro thread ingrese al mismo monitor y llame a ***notify()***. Este método libera el bloqueo antes de esperar y vuelve a adquirir el bloqueo antes de regresar de dormir.
- **public final void wait(long** timeout**)** */throws InteruptedException/*: El thread que ejecuta este método se suspende hasta que, o bien recibe una notificación, o bien transcurre el *timeout* establecido en el argumento.
  - **wait(**0**):** Representa una espera indefinida hasta que llegue la notificación.
- **public final wait(long** timeout**, int** nanos**)**: Es un wait en el que el tiempo de timeout es 10^5*timeout + nanos [ns].
- **public final void notify()**: Despierta un solo thread que invocó a *wait()* en el mismo objeto. Cabe señalar que las llamadas en *notify()* en realidad no ceden el bloqueo de un recurso. Le dice a un thread en espera que ese thread puede despertarse. Sin embargo, el bloqueo no se abandona hasta que el bloqueo sincronizado del notificador se haya completado. En resumen, notifica al objeto un cambio de estado; esta notificación es transferida a sólo uno de los threads que esperan sobre el objeto. No se puede especificar cuál de los objetos que esperan en el objeto será despertado.
- **public final void notifyAll()**: Despierta todos los threads que llamaron a *wait()* en el mismo objeto.

Semaphore

Un semaphore es un mecanismo de mas alto nivel que synchronized. Un semaphore es un tipo de objeto, y tiene un contador que protege el acceso a uno o más recursos compartidos que no necesariamente están en el mismo bloque.

Un semaphore, como cualquier tipo de dato, queda definido por:

- Conjunto de valores que se le pueden asignar.
- Conjunto de operaciones que se le pueden aplicar.
- Una lista/cola de procesos, en la que se incluyen los procesos suspendidos a la espera de su cambio de estado.

Los semaphores son herramientas básicas de la programación concurrente y son proporcionados por la mayoría de los lenguajes de programación. Siempre hay que asegurarse de seleccionar el mecanismo apropiado de sincronización de acuerdo con las características de la aplicación que se esté diseñando. Un mecanismo equivocado puede resultar siendo perjudicial y poco eficiente.

Valores de un semaphore

Los semaphores se clasifican según el rango de valores que toman:

- **Binary semaphore**: Semaphore que puede tomar sólo dos valores: 0 y 1.
- **General semaphore**: Semaphore que puede tomar cualquier valor $x \in \mathbb{N}$.

Si el valor de un semaphore es 0, esto nos indica que el lock está cerrado (nadie puede pasar). Si, por el contrario, el valor de semaphore es $x > 0$, entonces podrán entrar x threads a la sección crítica.

Métodos básicos de un semaphore

A la hora de trabajar con un semaphore, debemos tener en cuenta que los dos métodos más básicos que vamos a utilizar son aquellos que nos indican que ha entrado un thread a la sección crítica y que, por ende, hay que decrementar el número del semaphore (cuántos espacios libres quedan para acceder a esta sección crítica), y el método que indica que un thread ha terminado su ejecución en la sección crítica y que hay que incrementar el número del semaphore.

Estos métodos son:

- **acquire()**: Indica que un thread entrará a la sección crítica y decrementa en 1 el número de permisos del semaphore (si es que el contador interno del semaphore es mayor a cero, sino el semaphore pone al thread a dormir hasta que el contador sea mayor a cero).
- **release()**: Indica que un thread salió de la sección crítica e incrementa el número de permisos del semaphore.

Dado que puede que no se permita el acceso a la sección crítica (el valor de semaphore puede ser 0), deberá encerrarse en un bloque *try-catch-finally* la parte que involucre al método acquire().

Creación

Para la creación de un semaphore será requisito importar previamente la librería *Semaphore* de la siguiente forma:

import java.util.concurrent.Semaphore;

Luego, podemos crear un semaphore de la forma:

***Semaphore** semaphore = **new Semaphore**(locksAmount);*

Siendo *locksAmount* la cantidad de locks que le daremos al semaphore (si locksAmount = 1, estaremos hablando de un binary semaphore).

Lock

Un lock es una herramienta de bloqueo que se utiliza para cuidar la concurrencia de los trabajos. Los tipos de locks que vamos a utilizar son:

- **ReentrantLock:** Implementación concreta de la interfaz Lock. Tiene las siguientes ventajas frente a synchronized:
  - Podemos establecerle ***fairness*** (justicia).
  - Tenemos varios métodos para aplicarle:
    - lock(): Bloquea.
    - tryLock() || tryLock(time, timeUnit): Intenta bloquear hasta que pueda. Retorna TRUE o FALSE. El otro hace lo mismo, pero si pasa *[time]* tiempo y no pudo bloquear, retorna FALSE.
    - unlock(): Desbloquea.
  - Podemos lockear en un método y deslockear en otro método, no es necesario que lock() y unlock() estén en el mismo bloque de código.
- **ReadWriteLock:** Interfaz que especifica otro bloqueo. Mantiene un par de bloqueo para acceso de lectura y escritura. La idea detrás de los bloqueos de lectura y escritura es que, por lo general, es seguro leer variables mutables simultáneamente, siempre y cuando nadie escriba en esta variable. Por lo tanto, el bloqueo de lectura puede ser mantenido simultáneamente por múltiples threads, siempre y cuando ningún thread mantenga el bloqueo de escritura. Esto puede mejorar el rendimiento. Los métodos se aplicarán de la forma:
  - *lock.**writeLock**().**lock**()*: lock para escribir.
  - *lock.**writeLock**().**unlock**()*: unlock para escribir.
  - *lock.**readLock**().**lock**()*: lock para leer.
  - *lock.**readLock**().**unlock**()*: unlock para leer.

Locks para múltiples eventos simultáneos

Cuando necesitamos que múltiples eventos simultáneos terminen de ejecutarse para poder ejecutar algo que los involucre a todos ellos (por ejemplo, esperar a que lleguen los n participantes de una conferencia para comenzar), podemos utilizar dos clases:

- **CountDownLatch class**: Para que un thread espere la ejecución de estas operaciones, se ejecuta el método ***await()***. El thread duerme hasta que las operaciones se completen. Cuando se termina una de las operaciones, el thread que la terminó ejecuta el método ***countDown()*** para disminuir el contador interno de la clase CountDownLatch. Cuando el contador llega a 0, la clase despierta todos los threads que estaban durmiendo en el método await(). Para crear un objeto de este tipo, lo haremos de la forma:

  private final CountDownLatch *nombreVariable* = ***new CountDownLatch***(*amount*);

  Siendo *amount* la cantidad de eventos que debemos esperar a que sucedan. En el punto que queramos esperar a que se terminen estos eventos, deberemos indicarlo de la forma:

  nombreVariable.***await***();

  De esta forma, sólo avanzaremos de esta línea de código cuando la espera haya llegado a 0.
- **CyclicBarrier class**: Es una clase similar a CountDownLatch, pero más poderosa. Se inicia con el número de threads que se sincronizarán en un punto determinado. Cuando uno de esos threads llega al punto, llama al método await() para esperar a los otros threads. Cuando el thread llama a ese método, la clase de CyclicBarrier bloquea al thread. Cuando el último thread llama al método await() de la clase CyclicBarrier, se despierta a todos los threads que estaban esperando y continúan con su trabajo. Una ventaja de esta clase es que se le puede pasar un objeto ejecutable como parámetro de inicialización, y la clase CyclicBarrier ejecuta este objeto como un thread cuando todos los threads han llegado al punto común.

CyclicBarrier

La clase CyclicBarrier es provista por Java, y nos permite la sincronización de dos o más threads en un determinado punto. Esta clase comparte conceptos con la clase CountDownLatch, pero es más potente y flexible que ésta.

Para inicializar un objeto de la clase CyclicBarrier, es requisito asignarle un número $n \in \mathbb{N}$, el cual indica el número de threads que serán sincronizados donde lo necesitemos.

Una ventaja sobre esta clase que vale la pena mencionar, es que podemos utilizar en el constructor de la clase CyclicBarrier un objeto de tipo Runnable que se ejecutará una vez que todos los threads se sincronizaron. Esta primitiva está implementada de manera tal que primero se ejecuta el objeto Runnable opcional y luego se despiertan los threads que están durmiendo.

Proceso de sincronización

Cuando un thread llega al punto de sincronización deseado, ejecuta el método await(), el cual ordena que espere el arribo de los demás threads.

Una vez que el último de los threads ejecuta este método, la clase CyclicBarrier despierta a todos los threads para que prosigan su ejecución.

Mencionamos también que la clase CyclicBarrier tiene otro método sobrecargado await(**long** time, **TimeUnits** units), el cual puede recibir hasta dos parámetros de tiempo de tipo *long* y *TimeUnits*.

Estos parámetros sirven para indicar que el thread estará dormido en la CyclicBarrier hasta que todos lleguen o hasta que se transcurra el tiempo determinado.

Métodos útiles

await()

Este método ordena al thread que duerma hasta que todos los threads hayan arribado.

Si hay varios threads esperando en este método y uno de ellos es interrumpido, este thread recibe una interrupción del tipo *InterruptedException*, pero los otros threads reciben una excepción del tipo *BrokenBarrierException* y el objeto de tipo CyclicBarrier pasa a estado de *Broken*.

getNumberWaiting()

Este método devuelve el número de threads dormidos en la CyclicBarrier.

getParties()

Este método indica el número de tareas que serán sincronizadas con la CyclicBarrier.

reset()

Este método resetea la CyclicBarrier a su valor de inicialización. Si se ejecuta este método, todos los threads dormidos en el método await() reciben una excepción del tipo *BrokenBarrier* y el objeto de tipo CyclicBarrier pasa a estado de *Broken*.

isBroken()

Este método retorna si el estado del objeto de tipo CyclicBarrier es Broken o no.

Primitiva Phaser

Esta primitiva es una de las más potentes y complejas que nos brinda Java para ejecutar tareas concurrentes por fases. Este mecanismo puede ser muy útil cuando hay tareas concurrentes divididas en etapas.

Esta primitiva provee el mecanismo de sincronización al final de cada etapa, para que ningún thread comience la siguiente etapa hasta que no lleguen todos.

Como cualquier otra primitiva, Phaser debe ser inicializado con la cantidad de "participantes" a esperar para avanzar de etapa ($m \in \mathbb{N}^+$) con la ventaja de poder modificar este valor de forma dinámica durante la ejecución del programa.

Métodos útiles

arriveAndAwaitAdvance()

Este método decrementa el contador interno del Phaser y avisa que el thread ha llegado al punto de sincronización y pasa a estado de *sleep* a la espera del resto de los threads.

arriveAndDeregister()

Este método notifica al Phaser que el thread ha cumplido la etapa en cuestión pero no participará en las próximas sincronizaciones. Luego de haber ejecutado este método, Phaser no debería esperar más por este thread en las demás sincronizaciones.

isTerminated()

Este método retorna *true* si todos los threads han ejecutado el método *arriveAndDeregister()* (se han desregistrado del Phaser). Caso contrario, retorna *false*.

arrive()

Este método avisa que el thread arribó, pero no espera a ningún otro thread.

register()

Este método agrega un participante al Phaser. Es considerado como no arrivado para la fase actual.

bulkRegister(int parties)

Este método agrega el número especificado de participantes al Phaser.

Primitiva Exchanger

Esta es otra primitiva ofrecida por Java, que brinda un mecanismo que permite el intercambio de datos entre dos tareas concurrentes y la sincronización entre dos threads en un determinado punto. Cuando los dos threads llegan al punto establecido, la estructura de datos de un thread pasa al otro.

Esta primitiva sirve para sincronizar sólo dos threads, por lo que puede ser implementada en el caso productor-consumidor con sólo una instancia de cada uno.

Proceso de sincronización e intercambio de datos

Si tomamos el ejemplo del sistema productor-consumidor, tenemos la siguiente secuencia:

- El consumidor comienza con un buffer vacío y llama al Exchanger para sincronizarse. Necesita datos para consumir.
- El productor comienza con un buffer vacío; genera 10 productos, los almacena en un buffer y llama a la primitiva para intercambiar esta estructura de datos (buffer).
- Ahora, ambos threads están en el Exchanger e intercambian estructuras.
- El primer thread que ejecuta el Exchanger pasa a estado *sleep*, por cuestiones lógicas, y espera el arribo del otro thread.

Cabe resaltar que la clase Exchanger tiene otra versión que permite indicar el tiempo máximo que estará esperando el thread para la sincronización.

Abstracciones para la creación de threads

A la hora de crear threads tenemos las dos opciones mencionadas al principio de este resumen:

- Creando objetos de tipo Thread cuyos argumentos sean otros objetos creados de una clase que hereda de Runnable.
- Creando objetos de tipo *<nombreDeClase>* donde la misma hereda de Thread.

Cuando creamos threads y trabajamos con ellos (sincronización, ejecución de sus métodos run(), etcétera) podemos llegar a dejar engorroso el código en ciertas clases importantes, lo cual dificultaría la lectura y comprensión de la misma.

Para evitar esto y lograr un nivel mayor de abstracción, favoreciendo así la escritura y comprensión de código extenso y complejo implementando threads, Java nos provee el patrón Factory.

Patrón Factory

Este es uno de los patrones más utilizados en la OOP.

Es un patrón creacional, y provee una interfaz que permite la creación de una familia de objetos sin especificar la clase concreta. Abstrae al usuario de los posibles tipos de objetos concretos existentes. Esto nos permite, en vez de crear una nueva instancia de un objeto (método new()), utilizar el patron Factory y prescindir del método mencionado.

Ventajas

- Es fácil cambiar la clase de objetos creados y la manera en que se crean.
- Es fácil limitar la creación de objetos para limitados recursos (podemos indicar que sólo tendremos n objetos de una determinada clase).
- Es fácil obtener información estadística respecto a la creación de objetos.

Para la implementación de este patrón, Java nos provee la interfaz *ThreadFactory*.

Esta interfaz debe ser implementada por una clase que creemos nosotros, y tendremos que sobrescribir el método newThread(**Runnable** r). Este método recibe un objeto Runnable como parámetro y retorna un objeto de tipo Thread.

Aunque muchos ThreadFactory tienen simplemente una línea donde crean y retornan un thread (*return new Thread(r)*), esta interfaz provee flexibilidad para crear threads personalizados, guardar estadísticas, validar la cantidad de threads creados con algún criterio, y más.

Esta interfaz es útil para la creación de threads, pero se queda atrás cuando hablamos de implementación de los mismos.

Thread executor

Como mencionamos anteriormente, uno generalmente debe implementar todo el código relacionado a la creación y manejo de threads (un thread por cada tarea); y, en caso de desarrollar una aplicación con muchas tareas, el código puede volverse muy difícil de comprender y esto puede afectar al rendimiento.

Para solucionar este problema, Java nos brinda las siguientes herramientas:

- Executors framework
- ThreadPoolExecutor class

Executors framework

Este framework es el responsable de la instanciación, ejecución y uso de los threads necesarios; pero, además de esto, tiene como objetivo mejorar la performance con un objeto de tipo ThreadPoolExecutor (una *pileta* de threads).

Con este framework, nosotros sólo debemos preocuparnos por crear tasks de tipo Runnable y enviarlas al ThreadPoolExecutor.

Podemos también asignar a este executor un *handler* que es utilizado cuando el executor no puede ejecutar una tarea.

Métodos útiles

shutdown()

Este método impide que el executor acepte nuevas tareas, y lo cierra una vez que haya finalizado las tareas que aún no se terminaron de ejecutar.

shutdownNow()

Este método finaliza el executor de inmediato. No ejecuta las tareas pendientes, pero las que quedaron ***sin ejecutarse*** las devuelve en una lista. Las tareas que ***no terminaron*** de ejecutarse se finalizan.

En cualquiera de los dos casos, si se envía una nueva tarea a ejecutar, la misma será rechazada activando el handler de tipo RejectedTaskController que se le asignó al executor. Aquí se ejecutará el método *rejectedExecution()*.

ThreadPoolExecutor class

Podemos pensar en un objeto de este tipo como una pileta donde se almacenarán threads. Cuando el executors framework ordena que se ejecute una tarea, se escoge un thread "ocioso" de esta pileta y se le asigna la tarea a ejecutar.

Métodos útiles

*newFixedThreadPool(**int** maxThreads)*

Este constructor indica que se deben crear y reutilizar como máximo la cantidad de threads que le pasamos como parámetro en el constructor. La ventaja de reutilizar threads es el ahorro de tiempo en la creación de threads.

Una vez creado el objeto de tipo ThreadPoolExecutor se pueden enviar tareas para ejecución con el método *execute()*.

getPoolSize()

Este método devuelve el número actual de threads en el pool.

getActiveCount()

Este método devuelve el número de threads que están efectivamente en ejecución.

getTaskCount()

Este método devuelve el número de tareas que alguna vez fueron programadas para ejecutarse.

getCompleteTaskAccount()

Este método devuelve el número de tareas finalizadas por el executor.

shutdown()

Este método se ejecuta sobre el executor. Una vez ejecutado, el executor no acepta nuevas tareas.

Monitores

Definición y concepto

En programación paralela, los monitores son estructuras de datos abstractas destinadas a ser usadas sin peligro por más de un thread de ejecución. Son clases comunes y corrientes de Java, con la salvedad de que todos sus métodos públicos deben tener el modificador de acceso *synchronized*, garantizando así la exclusión mutua a cada uno de ellos. El resto de sus métodos deben ser privados (mientras menos métodos públicos tenga un monitor, mejor). En otras palabras, es un mecanismo de software de alto nivel para control de concurrencia que contiene los datos y procedimientos necesarios para realizar la asignación de un determinado recurso o grupo de recursos compartidos *reutilizables en serie*.

En los monitores, el comportamiento de *synchronized* es el mismo de siempre; es decir, si tenemos n métodos públicos con el modificador de acceso *synchronized* y un thread ejecuta alguno de estos métodos, entonces el resto de threads no podrá ejecutar ningún otro método del monitor hasta que el thread que está en ejecución termine su tarea y libere el lock. El lock es el mismo monitor.

Los monitores están pensados para ser usados en entornos multiproceso o multithread, y por lo tanto muchos procesos o threads pueden llamar a la vez a un procedimiento del monitor. Los monitores garantizan que en cualquier momento, a lo sumo un thread puede estarse ejecutando dentro de un monitor. Ejecutarse dentro de un monitor significa que sólo un thread estará en estado de *ejecución* mientras dura la llamada a un método del monitor. El problema de que dos threads ejecuten un mismo método dentro del monitor es que se pueden dar condiciones de carrera, perjudicando el resultado de los cálculos. Para evitar esto y garantizar la integridad de los datos privados, el monitor hace cumplir la exclusión mutua implícitamente, de modo que sólo un método esté siendo ejecutado a la vez. De esta forma, si un thread llama a un método mientras otro thread está dentro del monitor, se bloqueará y esperará en la cola de entrada hasta que el monitor quede nuevamente libre. Aunque se llama *"cola de entrada"* no debería suponerse ninguna política de encolado (FIFO, LIFO...), ya que no controlaremos qué thread ingresa en qué momento.

Buscamos implementar monitores con el objetivo de reemplazar a los Semaphores, ya que éstos son primitivas con las cuales es difícil expresar una solución a grandes problemas de concurrencia, y su presencia en el código fuente de programas concurrentes incrementa la ya existente dificultad para probar que los programas son correctos.

Los semáforos tienen debilidades importantes, como que el control de la concurrencia es responsabilidad del programador, las primitivas de control están esparcidas por todo el sistema, y la sección crítica y la sincronización se implementan usando las mismas primitivas y esto lleva a una notable dificultad para identificarlas, haciendo así que los programas que usen semáforos sean difíciles de mantener.

Componentes

Los componentes de un monitor son:

- *Variables que representan el estado del recurso.*
- *Inicialización* (código del constructor).
- *Procedimientos que implementan operaciones sobre el recurso* (métodos).
- *Cola de entrada:* Cola que contiene a los threads que han llamado a algún método del monitor, pero no han podido adquirir permiso para ejecutarlo aún.

Habiendo visto estos componentes, podemos decir entonces que un monitor se divide en dos partes lógicas:

- Algoritmo para la manipulación del recurso y sincronización.
- Mecanismo para la asignación del orden en el cual los procesos asociados pueden compartir el recurso y/o son sincronizados.

A la hora de usar monitores, los métodos más usados son *wait()* para cuando debemos esperar a que se cumpla una condición y ponemos a dormir ese thread para dejar que se ejecuten los demás entregando el lock del monitor, y *notifyAll()* para cuando terminamos de ejecutar algún método del monitor que puede llegar a dejar durmiendo algún thread. De esta forma avisamos que terminamos la ejecución y que puede entrar otro thread.

Para asegurar que un proceso en estado de espera obtenga el recurso que está esperando, la lógica del monitor debe darle la prioridad sobre los procesos que solicitan entrar al monitor. De otro modo, los nuevos procesos tomarán el recurso antes de que el proceso en espera lo haga, y esto puede llevar al proceso en estado de espera a una postergación indefinida.

– IV –

Gramáticas, lenguajes y autómatas

En este capítulo veremos las gramáticas, lenguajes y autómatas, sus relaciones entre sí y cómo podemos vincularlos con la programación concurrente.

Motivación

Nuestras motivaciones serán, principalmente, los sistemas reactivos (RS) y la arquitectura dirigida por eventos (EDA).

Sabemos que los RS son sistemas informáticos que responden continuamente a su entorno (es decir, el estado del entorno afecta al sistema).

Diferenciamos a los RS de los TS (sistemas transformacionales) y los IS (sistemas interactivos) de la siguiente forma:

- Los TS realizan cálculos cuyas salidas están disponibles al finalizar la ejecución. Cuando terminan, el estado del sistema no es significativo.
- Los IS realizan procesos no terminados basados en estados que están interactuando con el entorno para habilitar, aplicar o prevenir cierto comportamiento en el mismo. Pueden estar sujetos a requisitos rigurosos de tiempo real (podemos necesitar que el sistema reaccione de cierta forma en algún instante de tiempo particular) y, a menudo, realizar varios procesos en paralelo.

Por otro lado, la EDA es un patrón de arquitectura de software impulsado por la producción, detección, consumo, y reacción a eventos. Una EDA facilita un mayor grado de reacción, debido a que están diseñados para entornos no predecibles y asíncronos. Estudiaremos sistemas sensibles a estados y eventos, similares a los vistos en la materia *electrónica digital I*.

Definiciones previas

Símbolo

Es una entidad abstracta, un axioma. Normalmente, los símbolos son letras minúsculas del alfabeto (a, ... , z), dígitos (0, ... , 9) y otros caracteres (+, -, *, /, ? ...). Los símbolos también pueden estar formados por varias letras o caracteres (ip, 0a?, ...).

Vocabulario o alfabeto

Es un conjunto no vacío finito de símbolos. Los alfabetos se definen por enumeración de los símbolos que contienen. También se puede definir las tablas ASCII y EBCDIC como alfabetos.

Para denotar que un símbolo **a** pertenece a un alfabeto **V** utilizamos la notación:

$$\mathbf{a} \in \mathbf{V}$$

Cadena

Una cadena es una secuencia finita de símbolos de un determinado alfabeto. Su longitud se define como el número de símbolos que la cadena contiene. La notación para indicar la longitud de una cadena es:

$$| \text{ abcd } | \rightarrow 4$$
$$| \text{ a + 2 * b } | \rightarrow 5$$

Existe una cadena denominada *cadena vacía*, que no tiene símbolos y se denota con la letra griega lambda minúscula (λ). Su longitud es:

$$|\lambda| \rightarrow 0$$

Concatenación de cadenas

Sean α y β dos cadenas cualesquiera. Se denomina *concatenación de α y β* a una nueva cadena $\alpha\beta$ constituida por los símbolos de la cadena α seguidos por los de la cadena β .

El elemento neutro de la concatenación cumple:

$$\alpha\lambda = \lambda\alpha = \alpha$$

Universo del discurso

Es el conjunto de todas las cadenas que se pueden formar con el alfabeto utilizado.

Se denota de la forma W(V) siendo V el vocabulario utilizado para generar este universo del discurso.

Evidentemente, W(V) es un conjunto infinito (si $a \in V$, podemos armar las cadenas a, aa, aaa, ...). En cuanto a la cadena vacía: $\lambda \in V$. Esta cadena vacía se llama *cierre* o *clausura* del alfabeto.

Definiciones

- **Clausura positiva de un alfabeto**: $W(V)^+ = W(V) \setminus \{\lambda\}$ (aquí, $\lambda \notin W(V)$). Es el universo del discurso (W) del alfabeto V sin incluir la cadena vacía λ.
- **Clausura de un alfabeto**: $W(V)^* = W(V) \cup \{\lambda\}$ (aquí, $\lambda \in W(V)$). Es el universo del discurso (W) del alfabeto V incluyendo la cadena vacía λ.

Lenguajes

Se denomina *lenguaje sobre un alfabeto V* a un subconjunto de W(V). Lo podemos denotar como L(W(V)) y vemos que $L \subseteq W(V)$. Para simplicidad, lo denotamos con L. Puede definirse también como un conjunto de palabras de un determinado alfabeto.

La enumeración de las cadenas $C \in L$ es ineficiente porque los lenguajes pueden tener una cantidad muy grande de cadenas y, en algunos casos, puede ser imposible esta enumeración porque el lenguaje contiene infinitas cadenas ($\#L \to \infty$).

Los lenguajes se definen por las propiedades que cumplen las cadenas de dicho lenguaje (reglas gramaticales del lenguaje). Si el lenguaje L depende de $W(V) = \{\lambda,0,1\}$ y la condición es que las cadenas sean palíndromos, entonces $L = \{\lambda, 0, 1, 00, 11, 000, 010, 101, 111, 0000, 0110, ...\}$. En este caso vemos claramente que $\#L \to \infty$ ya que no hay ninguna restricción sobre la longitud de las cadenas.

Lenguaje vacío

El lenguaje vacío es un conjunto vacío: $L = \{\emptyset\}$.

Si $L_1 = \{\emptyset\}$ y $L_2 = \{\lambda\} \Rightarrow L_1 \neq L_2 \because \#L_1 = 0 \neq \#L_2 = 1$.

Es decir, el lenguaje vacío no es aquel que contiene la cadena vacía, puesto que si $\lambda \in L \Rightarrow \#L = 1$, y si $L = \{\emptyset\} \Rightarrow \nexists\ x \in L \therefore \#L = 0$ ($\emptyset$ no es un símbolo, es sólo la notación de que no hay ningún elemento en el conjunto).

Decimos que una cadena de símbolos $C \in L \Leftrightarrow C$ se ha formado obedeciendo las reglas gramaticales del lenguaje L.

Definiciones básicas importantes

- **Gramática**: Ente formal para especificar, de una manera finita, el conjunto de cadenas de símbolos de un alfabeto V que constituyen un lenguaje L.
- **Autómata**: Construcción lógica que recibe una entrada y produce una salida en función de todo lo recibido hasta ese instante.

Gramáticas

Una gramática es una cuádrupla de la forma: $G = (V_T, V_N, P, S)$, donde:

- V_T: *Vocabulario terminal.* Conjunto finito de símbolos terminales (estados). Todas las cadenas del lenguaje L definido por la gramática G están formados con símbolos de V_T.
- V_N: *Vocabulario no terminal.* Conjunto finito de símbolos no terminales (eventos). Es el conjunto de símbolos introducidos como elementos auxiliares para la definición de la gramática, y que no figuran en las cadenas del lenguaje L.
- S: *Símbolo inicial.* Se destaca que $S \in V_N$. A partir de este símbolo inicial se aplican las reglas de P para obtener las distintas cadenas $C \in L$.
- P: *Producciones.* Conjunto de reglas de derivación. Son las reglas que se aplican desde S para obtener las cadenas $C \in L$. P se define por medio de la enumeración de las distintas producciones, en forma de reglas o por medio de un metalenguaje.

Tanto V_T como V_N se definen por enumeración de los símbolos terminales y no terminales respectivamente.

Resaltamos que $\forall\, x \in V_N : x \notin V_T \Rightarrow V_T \cap V_N = \emptyset \wedge V_T \cup V_N = V$.

Hacemos también la misma distinción para vocabularios V que la que hicimos para el universo del discurso W(V) respecto a la inclusión o exclusión de la cadena vacía:

- $V^+ = V \setminus \{\lambda\}$ $(\lambda \notin V)$.
- $V^* = V \cup \{\lambda\}$ $(\lambda \in V)$.

Notación

- **Vocabulario**: Conjunto compuesto por $V_T \cup V_N$. Se denota con las letras mayúsculas del final de abecedario (U, V, W, X, Y, Z).
- **Vocabulario terminal**: Sus elementos se representan por:
  - Letras minúsculas del comienzo del abecedario (a, b, c, d, e, f, g).
  - Operadores (+, -, *, /, ...).
  - Caracteres especiales (#, @, (,), ., ;, ...).
  - Dígitos decimales (0, ... , 9).
  - Palabras reservadas de lenguajes de programación con letras minúsculas y en negrita (**if**, **then**, **else**, **while**).
- **Vocabulario no terminal**: Sus elementos se representan por:
  - Letras mayúsculas del comienzo del abecedario (A, B, C, D, E, F, G), a excepción del símbolo inicial S.
  - Nombres en minúscula encerrados entre cuñas (<expresión>, <operador>, ...).
- **Cadena**: Secuencia de símbolos terminales y no terminales indiferenciados. Se representan con letras minúsculas griegas (α, β, γ, δ, ε, ...).
- **Cadena terminal**: Cadenas compuestas en su totalidad por símbolos terminales. Se representan con las letras minúsculas del final del abecedario (t, u, v, w, x, y, z).

Jerarquía de gramáticas

Noam Chomsky definió cuatro tipos distintos de gramáticas en función de la forma de las reglas de derivación P. Esta clasificación comienza con un tipo de gramática que pretende ser universal, y aplicando restricciones a sus reglas de derivación se obtienen los otros tres tipos de gramáticas.

Esta clasificación jerárquica es de la forma:

- **Gramáticas de tipo 0**: *Gramáticas no restringidas* o *gramáticas con estructura de frase.*
- **Gramáticas de tipo 1**: *Gramáticas sensibles al contexto.*
- **Gramáticas de tipo 2**: *Gramáticas de contexto libre* o *gramáticas libres de contexto.*
- **Gramáticas de tipo 3**: *Gramáticas regulares* o *gramáticas lineales a la derecha.* Sus reglas de producción comienzan por un símbolo $x \in V_T$ que puede ser seguido o no por un símbolo $x \in V_N$.

Especificación jerárquica

Gramática de tipo 0

En las gramáticas de tipo 0, las reglas de derivación son de la forma:

$$\alpha \rightarrow \beta$$

Siendo:

- $\alpha \in (V_T \cup V_N)^+$, i.e.: $\forall\, \alpha : \alpha \neq \lambda$
- $\beta \in (V_T \cup V_N)^*$

Es decir, con una gramática de tipo 0 podemos obtener derivaciones cuyo miembro izquierdo es cualquier símbolo terminal o no terminal excepto λ, y cuyo término derecho es cualquier símbolo terminal o no terminal incluyendo λ.

De forma más técnica, decimos que la única restricción que posee una gramática de tipo 0 es que no se puede realizar la siguiente derivación:

$$\lambda \rightarrow \beta$$

Gramática de tipo 1

En las gramáticas de tipo 1, las reglas de derivación son de la forma:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Siendo:

- $A \in V_N$
- $\gamma \in (V_T \cup V_N)^+$
- $\alpha, \beta \in (V_T \cup V_N)^*$

Es decir, podemos realizar derivaciones con términos izquierdo y derecho compuestos por símbolos terminales, no terminales y λ. En esta gramática se suma la restricción de que la longitud de la parte derecha de la producción debe ser mayor o igual a la longitud de la parte de la izquierda.

De forma matemática, la restricción es:

$$|\alpha A \beta| \leq |\alpha \gamma \beta|$$

Gramática de tipo 2

En las gramáticas de tipo 2, las reglas de derivación son de la forma:

$$A \rightarrow \alpha$$

Siendo:

- $A \in V_N$
- $\alpha \in (V_T \cup V_N)^+$

En esta gramática, se suma la restricción de que cada regla es un par ordenado (A, α). Es decir, en el lado izquierdo de una producción pueden aparecer el símbolo distinguido o un símbolo no terminal, y en el lado derecho de una producción cualquier cadena de símbolos terminales y/o no terminales de longitud mayor o igual que 1.

Gramática de tipo 3

En las gramáticas de tipo 3, las reglas de derivación son de la forma:

$$A \rightarrow aB$$
$$\text{ó}$$
$$A \rightarrow a$$

Siendo:

- $A, B \in V_N$
- $a \in V_T$

En esta gramática se suma la restricción de que la regla de producción (término derecho) siempre comienza por un símbolo terminal que puede ser seguido o no por un símbolo no terminal.

Definiciones útiles

- **Lenguaje formal**: Lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados.
- **Gramática formal**: Estructura lógico-matemática con un conjunto de reglas de formación que definen las cadenas de caracteres admisibles en un determinado lenguaje formal.
- **Gramática regular**: Gramática formal que puede ser clasificada como regular izquierda o regular derecha. No merece la pena ahondar demasiado en esta definición por el alcance del curso.
- **Lenguaje regular**: Lenguaje generado por medio de una gramática regular. Son aquellos lenguajes cuyas cadenas está formadas por la concatenación de símbolos en las cuales no hay relación entre una parte de la cadena y otra parte de la cadena.

Correspondencia entre gramáticas y lenguajes

Definimos genéricamente un lenguaje de tipo *j* como un lenguaje generado por una gramática de tipo $j : j \in [0, 3]$.

La relación matemática entre lenguajes puede denotarse de la forma:

$$L(G_3) \subset L(G_2) \subset L(G_1) \subset L(G_0)$$

Expresiones regulares

Se introducirá la herramienta conocida como *expresiones regulares* que sirven para describir lenguajes de tipo 3, o lenguajes regulares. Estas expresiones se denominan *metalenguajes*, ya que su función es describir un lenguaje regular.

Operaciones con lenguajes regulares

Dado un vocabulario V, definimos las siguientes operaciones:

- **Unión**: $\forall$ $L_1(V)$, $L_2(V)$: $L_1(V) \cup L_2(V) = \{x \mid x \in L_1(V) \vee 1x \in L_2(V)\}$
- **Concatenación**: $\forall$ $L_1(V)$, $L_2(V)$: $L_1(V)L_2(V) = \{x_1x_2 \mid x_1 \in L_1(V) \wedge x_2 \in L_2(V)\}$
- **Potencia i-ésima de un lenguaje**: $\forall$ L(V) se define la potencia i-ésima del lenguaje L(V) como la concatenación de L consigo mismo i veces. Esto es: $L(V)^n = \{x^n \mid x \in L_1(V)\}$
- **Cierre**: $\forall$ L(V) : $L(V)^* = \bigcup_{n=0}^{\infty} L(V)^n$. Esto es la unión del lenguaje L(V) con todas sus potencias posibles.
- **Cierre positivo**: $\forall$ L(V) : $L(V)^+ = \bigcup_{n=1}^{\infty} L(V)^n$. Este es un caso particular de cierre de un lenguaje en el que no se considera la unión con la potencia 0.

Operaciones con expresiones regulares

Si α es una expresión regular $\Rightarrow \{\alpha\}$ es el conjunto descrito por la expresión regular α. También puede decirse que α denota el lenguaje de la cadena α.

- **Unión**: $\forall$ α, β | α, β son expresiones regulares : $\alpha|\beta = \{\alpha\} \cup \{\beta\}$
- **Concatenación**: $\forall$ α, β | α, β son expresiones regulares : $\alpha\beta = \{\alpha\beta\} = \{\alpha\}\{\beta\}$
- **Cierre**: $\forall$ α | α es expresión regular : $\alpha^* = \{\alpha\}^+$

Teoremas para expresiones regulares

Teorema #1

$\forall$ α, β | α, β son expresiones regulares : $\alpha = \beta \Leftrightarrow \alpha$ y β denotan al mismo conjunto regular.

Propiedades para las expresiones regulares

Concatenación

1. **Asociatividad**: $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
2. **Distributividad**: $\alpha\beta|\alpha\gamma = \alpha(\beta|\gamma)$
3. **Elemento neutro**: λ es el elemento neutro para la concatenación. Esto es: $\alpha\lambda = \lambda\alpha = \alpha$

Cierre

1. $(\alpha|\beta)^* = (\alpha^*|\beta^*)^* = (\alpha^*|\beta^*)$
2. $(\alpha|\lambda)^* = (\alpha^*|\lambda) = \alpha^*$
3. $\alpha\alpha^*|\lambda = \alpha^*$
4. $\lambda^* = \lambda$

Edición de expresiones regulares

Las expresiones regulares (E_R) utilizan tres operadores matemáticos básicos:

- **Estrella de Kleene**: *. Representa que el término que abarca a su izquierda puede aparecer cero veces, una vez, o más de una vez.
- **Concatenación**: Implícito al hacer adyacentes dos expresiones.
- **Unión**: +
- **λ**: !

Ejemplos de E_R:

- a+b+cd
- a(b+c)d
- abc*
- (abc)*
- a+b*
- (!+a)bc*

Autómatas

Un autómata es una quíntupla de la forma: A = (E, S, Q, f, g), donde:

- E: *Vocabulario de entrada.* Conjunto finito cuyos elementos se llaman *entradas* o *símbolos de entrada.*
- S: *Vocabulario de salida.* Conjunto finito cuyos elementos se llaman *salidas* o *símbolos de salida.*
- Q: *Estados.* Conjunto de estados posibles. Puede ser finito o infinito.
- f : E×Q → Q: *Relación de transición.* Para cada par del conjunto E×Q, la relación f devuelve un estado $\varepsilon \in Q$.
- g : E×Q → S: *Relación de salida.* Para cada par del conjunto E×Q, la relación g devuelve un símbolo de salida $s \in S$.

Podemos decir que un autómata es un dispositivo que manipula cadenas de símbolos (información codificada), produciendo otras cadenas de símbolos a su salida.

Los símbolos de entrada son recibidos por el autómata de forma secuencial; uno tras otro.

El símbolo de salida en un instante determinado producido por un autómata depende de toda la secuencia de símbolos recibida hasta ese instante; tanto de los símbolos que se utilizaron como del orden en el que vinieron.

Podemos definir ahora el estado de un autómata:

- **Estado de un autómata**: Es toda la información necesaria en un momento preciso para poder deducir, dado un símbolo de entrada en ese momento, cuál será el símbolo de salida.

Decimos, entonces, que conocer el estado de un autómata es lo mismo que conocer la "historia" de los símbolos de entrada, así como el estado inicial del autómata.

Otras definiciones útiles para un autómata son:

- **Configuración de un autómata**: Situación del autómata en un instante de tiempo dado.
- **Movimiento de un autómata**: Tránsito entre dos configuraciones.

Un autómata puede utilizarse para determinar si una cadena pertenece o no al lenguaje que éste autómata define.

Representación gráfica de autómatas

Mediante tablas de transición

En las tablas de transición representamos un autómata colocando una tabla de $n \times m$ donde n son los estados posibles y m son las entradas posibles. En la posición (0,0) se escribe el nombre de la relación a representar (f ó g). Se listan los posibles estados en la primer columna desde arriba hacia abajo (desde (1, 0) hasta (n, 0)), las entradas posibles se listan en la primer fila de izquierda a derecha (desde (0, 1) hasta (0, m)), y en cada posición (i, j) restante se colocará el resultado que devuelve la relación cuyos elementos de entrada son (0, j) y (i, 0).

Si se desea representar a f y g en una sola tabla, se siguen los mismos pasos con la diferencia de que en la posición (0, 0) se escribirá "f / g" y en cada casillero (i, j) se colocarán ambos resultados (el de f y el de g) dadas las entradas (0, j) y (i, 0) separadas por una barra inclinada (/).

Gráficamente:

f	*a*	*b*
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_3	q_1

g	*a*	*b*
q_1	*0*	*1*
q_2	*0*	*0*
q_3	*1*	*0*

f / g	*a*	*b*
q_1	q_1/*0*	q_2/*1*
q_2	q_3/*0*	q_2/*0*
q_3	q_3/*1*	q_1/*0*

Mediante máquinas de Mealy

Definición formal de una máquina de Mealy

Una máquina de Mealy es una 6-tupla de la forma: $\text{Mealy}_M = (E, S, Q, Q_0, f, g)$, donde:

- E: *Vocabulario de entrada.*
- S: *Vocabulario de salida.*
- Q: *Estados.*
- Q_0: *Estado inicial.*
- f : E×Q → Q: Mapeo entre un par compuesto por un símbolo de entrada y un estado para el siguiente estado correspondiente.
- g : E×Q → S: Mapeo entre un par compuesto por un símbolo de entrada y un estado para el símbolo de salida correspondiente.

Los diagramas de máquinas de Mealy son grafos dirigidos tales que cada nodo corresponde a un estado, y cada transición se corresponde con un evento.

La salida en una máquina de Mealy está expresada en los arcos junto al evento (la salida depende del estado y de dónde se viene).

Estructura gráfica

En la representación gráfica del autómata, la identificación del estado inicial se da con el símbolo distinguido (símbolo inicial) S. A cada estado del autómata (a excepción del estado inicial) se le asocia un símbolo $x \in V_N$ (el símbolo no terminal es como una "etiqueta" para identificar el estado dentro de la representación gráfica del autómata). Si al estado inicial S llega un arco proveniente de otro estado, entonces el estado inicial debe distinguirse con el símbolo inicial S y otro símbolo $x \in V_N$. No se debe asociar un símbolo no terminal a aquellos estados finales de los que **no salen arcos**.

Cada transición será definida de la forma $\delta(e_i, a) = e_j$, donde:

- e_i: Estado de origen.
- a: Símbolo de entrada (evento).
- e_j: Estado de destino.

Para cada transición se debe agregar un elemento al conjunto de producciones P. Esto se verá en el algoritmo para obtener la gramática regular desde el autómata finito.

Mediante máquinas de Moore

Definición formal de una máquina de Moore

Una máquina de Moore es una 6-tupla de la forma: $Moore_M = (E, S, Q, Q_0, f, g)$, donde:

- E: *Vocabulario de entrada.*
- S: *Vocabulario de salida.*
- Q: *Estados.*
- Q_0: *Estado inicial.*
- $f : E \times Q \to Q$: Mapeo de un estado y el alfabeto de entrada al siguiente estado.
- $g : Q \to S$: Mapeo de cada estado al alfabeto de salida.

Estructura gráfica

A diferencia de las máquinas de Mealy, en las máquinas de Moore indicamos las salidas en los estados, y las entradas en los arcos.

Diferencias entre máquinas de Mealy y máquinas de Moore

Las diferencias más importantes entre estos dos tipos de máquinas son:

- **Estados**: Las máquinas de Mealy tienden a tener menos estados.
- **Salidas (reacción a entrada)**:
  - *Máquinas de Moore:* Las salidas cambian cuando pasa un ciclo de reloj.
  - *Máquinas de Mealy:* Las salidas cambian automáticamente luego de procesar la lógica de la entrada.
- **Nodos y arcos**:
  - *Máquina de Moore:* Cada nodo (estado) está etiquetado con un valor de salida.
  - *Máquina de Mealy:* Cada arco (transición) está etiquetado con un valor de salida.
- **Poder de análisis de un lenguaje regular**: Como ambas máquinas son de estado finito, son igualmente expresivas. Cualquiera de estos tipos de máquinas se puede usar para analizar un lenguaje regular.
- **Poder de implementación**:
  - Todos los circuitos secuenciales pueden implementarse como máquinas de Moore.
  - Algunos circuitos secuenciales sólo pueden ser implementados mediante máquinas de Moore.
  - No todos los circuitos se pueden implementar con máquinas de Mealy.
  - Si un circuito secuencial puede implementarse con una máquina de Mealy, entonces también puede implementarse con una máquina de Moore.

Definiciones útiles

- **Estado accesible**:
  $\forall\ A = (E, S, Q, f, g)$ con $q_j, q_i \in Q : q_j$ es accesible desde $q_i \Leftrightarrow \exists\ e \in E^* \mid f(q_i, e) = q_j$

  Podemos deducir que todo estado es accesible desde sí mismo, puesto que: $f(q_i, \lambda) = q_i$

- **Autómata conexo**: $\forall\ A = (E, S, Q, f, g) : A$ es conexo $\Leftrightarrow \forall\ q_i \in Q : q_i$ es accesible desde Q_0.
- **Autómata determinista**: $\forall\ A = (E, S, Q, f, g) : A$ es determinista $\Leftrightarrow f$ es determinista.
- **Autómata no determinista**: $\forall\ A = (E, S, Q, f, g) : A$ es no determinista $\Leftrightarrow f$ es no determinista.

Consideraciones

Con el fin de extender los dominios de las definiciones de las funciones f y g, se abarca también la posibilidad de que al sistema "no le entre nada", es decir, no tengamos entrada alguna. Esto lo podemos hacer modificando el dominio de f de la forma:

$$\text{Dom(f)} = \text{E} \cup \{\lambda\}$$

Vale la pena destacar que esto no puede ser adaptado a g porque la salida depende del estado anterior. Esto genera una ambigüedad que debe ser evitada.

Algoritmo para obtener la gramática regular desde el autómata finito

Este algoritmo es muy sencillo. Dado un autómata finito, se comienza tomando el símbolo inicial S y se escriben los estados destinos posibles (transiciones posibles) dadas ciertas entradas posibles (eventos de entrada posibles). Si tenemos un elemento $X \in V_N$ que denota un estado origen, una entrada $e \in E$ y un elemento $Y \in V_N$ que denota un estado destino, entonces escribimos la transición de la forma:

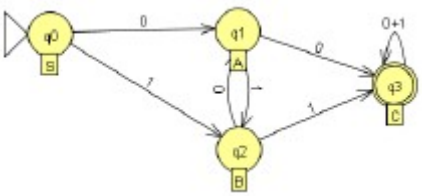
$$X \rightarrow eY$$

Podemos pensar estas transiciones como "si estoy en X y tengo una entrada e, voy a Y".

Se repite este procedimiento confeccionando una tabla constituida por las transiciones dado cada estado del autómata con cada entrada posible y salida posible de dicho estado.

Ejemplo:

Autómata



Gramática regular del autómata

LHS		RHS
C	→	λ
B	→	1C
C	→	0+1C
S	→	1B
B	→	0A
A	→	1B
S	→	0A
A	→	0C

Si e_j es un estado final con símbolo no terminal asociado A, entonces se debe agregar también a P la producción A → λ.

Si e_j es un estado inicial al que le llega un arco desde otro estado, y e_j posee un símbolo no terminal A y el símbolo distinguido S, entonces se debe utilizar para la producción el símbolo no terminal de e_j para evitar que el símbolo distinguido S aparezca a la derecha de una producción.

Si el estado inicial es también estado final, se debe agregar también a P la producción:

S → ε.

Tipos de autómatas

Para poder definir sin problemas los distintos tipos de autómatas, debemos aclarar que un autómata consta de un cabezal que puede ser móvil o no, y que éste puede leer y/o escribir una cinta que contiene una cadena a analizar.

Máquina de Turing

Una máquina de Turing (o autómata de tipo 0) es un autómata cuyo cabezal es móvil tanto hacia la izquierda como hacia la derecha (puede quedarse quieto si se desea) y puede leer y escribir la cinta a analizar, la cual puede ser considerada infinita. Este es el autómata con mayor capacidad de representación. Este tipo de autómata puede reconocer los lenguajes recursivamente enumerables (lenguajes de tipo 0).

El movimiento del cabezal de la máquina de Turing depende del símbolo leído y del estado en el que se encuentra la máquina. El resultado puede ser:

- Cambiar de estado.
- Imprimir un símbolo en la cinta reemplazando el símbolo leído.
- Mover el cabezal hacia la izquierda.
- Mover el cabezal hacia la derecha.

Por supuesto, el resultado puede ser también la combinación de varios de estos resultados posibles.

Definición formal de una máquina de Turing

Una máquina de Turing es un autómata de la forma: $M_T = (E, S, Q, f, g)$. Sin embargo, una notación más acertada y utilizada (equivalente pero más explícita) es la siguiente:

$$M_T = (Q, \Sigma, \Gamma, \delta, Q_0, B, F)$$

Donde:

- Q: *Estados.*
- Γ: *Símbolos permitidos en la cinta.*
- $B \in \Gamma$: *Símbolo blanco.*
- $\Sigma \subset \Gamma = \Gamma \setminus \{B\}$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{I, D, S\}$: *Siguiente movimiento.*
  - I: Movimiento hacia la izquierda.
  - D: Movimiento haca la derecha.
  - S: No moverse (stop).
- $Q_0 \in Q$: *Estado inicial.*
- $F \subset Q$: *Estado/s final/es.*

El lenguaje aceptado o reconocido por una máquina de Turing se denota $L(M_T)$. $L(M_T)$ es el conjunto de palabras formadas con el alfabeto Σ^* | M_T alcanza el/un estado/s final/es.

Teoremas y corolarios para máquinas de Turing

Teorema #1

$$\forall\ G_0 : \exists\ M_T \mid L(G_0) = L(M_T)$$

Teorema #2

$$\forall\ M_T : \exists\ G_0 \mid L(G_0) = L(M_T)$$

Teorema #3

Todo algoritmo es equivalente a una M_T.

Corolario #1

Existe una correspondencia entre lenguajes de tipo 0 y autómatas de tipo 0.

Transiciones

Una transición se define de la forma: r; w, d , donde:

- $r \in \Gamma$: *Símbolo leído en la cinta.*
- $w \in \Gamma$: *Símbolo a escribir en la cinta.*
- $d \in \{I, D, S\}$: *Dirección de movimiento del cabezal.*

Autómata lineal acotado

Este autómata (también conocido como autómata de tipo 1) es similar a una máquina de Turing, con la única diferencia de que la cinta a leer posee dos símbolos particulares que denotan el comienzo y fin de la misma (la cinta no puede ser infinita). El cabezal del autómata no puede desplazarse por fuera de estos límites indicados. Denotamos el fin de la cinta con el símbolo # y el comienzo de la cinta con el símbolo \$, y estos símbolos no son considerados como parte de la sentencia a reconocer; sólo son indicadores. Estos símbolos se escriben en el extremo izquierdo y derecho de la cinta respectivamente. Este tipo de autómata puede reconocer los lenguajes dependientes del contexto (lenguajes de tipo 1).

Definición formal de un autómata lineal acotado

Podemos definir un autómata de este tipo de la forma: $A_{LA} = (Q, \Sigma, \Gamma, \delta, Q_0, \#, \$, B, F)$, donde:

- Q: *Estados.*
- Γ: *Símbolos permitidos en la cinta.*
- $B \in \Gamma$: *Símbolo blanco.*
- $\Sigma \subset \Gamma = \Gamma \setminus \{B\}$
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{I, D, S\}$: *Siguiente movimiento.*
  - I: Movimiento hacia la izquierda.
  - D: Movimiento haca la derecha.
  - S: No moverse (stop).
- $Q_0 \in Q$: *Estado inicial.*
- $F \subset Q$: *Estado/s final/es.*
- \$: *Símbolo indicador de comienzo de cinta.*
- #: *Símbolo indicador de fin de cinta.*

Podemos definir matemáticamente los lenguajes aceptados por un A_{LA} de la forma:

$$L(A_{LA}) = \{W \mid W \in \{\Sigma \setminus \{\#, \$\}\}^* \wedge \exists\ q \in F \mid Q_0 \# W \$ \rightarrow \alpha q \beta\}$$

Teoremas y corolarios para autómatas lineales acotados

Teorema #1

$$\forall\ G_1 : \exists\ A_{LA} \mid L(G_1) = L(A_{LA})$$

Teorema #2

$$\forall\ L(A_{LA}) : \exists\ G_1 \mid L(G_1) = L(A_{LA})$$

Corolario #1

Existe una correspondencia entre lenguajes de tipo 1 y autómatas de tipo 1.

Autómata de pila

Este tipo de autómatas se denota A_P. En este autómata, el cabezal es estático. La cinta es la que se mueve, y lo hace en una sola dirección (de derecha a izquierda). El cabezal puede sólo leer la cinta pero puede escribir sobre una pila que contiene. Esta pila está limitada en un extremo por definición. Cuando se lee un elemento de la pila, este elemento desaparece o se saca, y cuando se escribe en la pila, se introduce un elemento. Este tipo de autómata puede reconocer los lenguajes libres de contexto (lenguajes de tipo 2).

Definición formal de un autómata de pila

Podemos definir un autómata de este tipo como una 7-tupla de la forma:

$$A_P = (Q, \Sigma, \Gamma, \delta, Q_0, Z_0, F)$$

Donde:

- Q: *Estados.*
- Σ: *Alfabeto de entrada* (finito).
- Γ: *Alfabeto de pila.*
- $\delta : Q \times \{\Sigma \cup \{\lambda\}\} \times \Gamma \rightarrow Q \times \Gamma^*$: *Función de transición.*
- $Q_0 \in Q$: *Estado inicial.*
- $Z_0 \in \Gamma$: *Símbolo inicial de la pila.* Es el elemento con el que comienza la pila.
- $F \subset Q$: *Estado/s final/es.*

Tal y como ha sido definida δ, el autómata en general es no determinista. Esto es, el resultado de la función δ no está determinado; es decir, pueden resultar dos o más valores sin que la función precise cuál va a tomar de ellos.

Las operaciones elementales que se pueden realizar con un A_P son de dos tipos:

- **Dependientes de la entrada**: Se lee $e_i \in \Sigma$ y se desplaza la cinta; y en función de e_i, q_j (el estado actual del autómata), y Z (el valor leído de la pila), el control pasa a otro estado q_m, y en la pila se introduce Z', o se extrae Z, o no se hace nada.
- **Independientes de la entrada**: Se lee $e_i \in \Sigma$ y se desplaza la cinta; pero las acciones a realizar no dependen de esta lectura.

Podemos definir matemáticamente a los lenguajes reconocidos por un A_P de dos formas:

- **Si se llega a un estado final**: Sabemos que si, dada una cadena de entrada, el autómata llega a un estado $q \in F$, entonces dicho lenguaje es reconocido por nuestro autómata.
  Esto es:
  $$L(A_P) = \{W \mid W \in \Sigma^* \land \exists\ q \in F \mid (Q_0, W, Z_0) \rightarrow (q, \lambda, \alpha)\}$$

- **Si se vacía la pila**: En estos autómatas tenemos también la posibilidad de que la pila se vacíe por completo. Esto también indicará que el lenguaje es reconocido por nuestro autómata, aunque no se haya llegado a un $q \in F$.
  Esto es:
  $$L(A_P) = \{W \mid W \in \Sigma^* \land \exists\ q' \in Q \mid (Q_0, W, Z_0) \rightarrow (q', \lambda, \lambda)\}$$

Se puede demostrar que ambas definiciones son equivalentes.

Teoremas y corolarios para autómatas de pila

Teorema #1

$$\forall\ G_2 : \exists\ A_P \mid L(G_2) = L(A_P)$$

Teorema #2

$$\forall\ L(A_P) : \exists\ G_2 \mid L(G_2) = L(A_P)$$

Corolario #1

Existe una correspondencia entre lenguajes de tipo 2 y autómatas de tipo 2.

Autómata finito

Al igual que en un autómata de pila, en un autómata finito el cabezal es estático y la cinta se mueve en una sola dirección (de derecha a izquierda), pero no posee una pila. Este tipo de autómata puede reconocer los lenguajes regulares (lenguajes de tipo 3).

Decimos que un autómata finito reconoce una sentencia de un lenguaje determinado si el autómata llega a un estado final luego de leer la cinta.

Definición formal de un autómata finito

Un autómata de este tipo puede definirse como una 6-tupla de la forma:

$$A_F = (E, Q, f, Q_1, F)$$

Donde:

- E = *Símbolos de entrada.*
- Q = *Estados* (finito).
- $f : E^* \times Q \rightarrow Q$: *Función de transición.*
- $Q_i \in Q$: *Estado inicial.*
- $F \subset Q$: *Estados finales.*

Teoremas y corolarios para autómatas finitos

Teorema #1

$$\forall\ G_3 : \exists\ A_F \mid L(G_3) = L(A_F)$$

Teorema #1

$$\forall\ A_F : \exists\ G_3 \mid L(G_3) = L(A_F)$$

Corolario #1

$$L(G_3) \subseteq L(A_F) \land L(A_F) \subseteq L(G_3) \therefore L(A_F) = L(G_3) = \{L_{Regulares}\}$$

Clasificación de los autómatas finitos

Autómatas finitos no deterministas

Un autómata finito no determinista (A_F^{ND}) se caracteriza por la posibilidad de que dada una entrada *e* en un estado q_i, se pueda pasar a un estado q_j, q_k, ... , q_n sin saber a ciencia cierta a cuál de estos estados pasará, existiendo la misma probabilidad de que pase a cualquiera de los estados posibles.

La definición formal de un A_F^{ND} coincide con la de un A_F a excepción de lo siguiente:

- $f : E^* \times Q \rightarrow P(Q)$: Función de transferencia no determinista.
- $P(Q)$: *Potencia de Q*. Es el conjunto de todos los subconjuntos posibles de Q.
  - ***Ejemplo***:

Si $Q = \{1, 2, 3\} \Rightarrow P(Q) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

- Q_i : *Estado inicial*. Puede no ser único.

A menudo, los A_F^{ND} son más compactos y fáciles de diseñar que los A_F^{D} (autómata finito determinista); además, siempre es posible convertir un A_F^{ND} en un A_F^{D}.

El procedimiento de un A_F^{ND} es el de comenzar en el/los estado/s inicial/es especificado/s y leer una cadena de caracteres pertenecientes al alfabeto. El autómata utiliza la función de transferencia f para determinar el siguiente estado en base al estado actual y el símbolo que acaba de leer de la cadena, y hasta que se producen estos acontecimientos no es posible determinar en qué estado se encuentra la máquina.

Cuando el autómata ha terminado de leer y se encuentra en un estado de aceptación, se dice que el A_F^{ND} acepta la cadena. De lo contrario, se dice que la cadena de caracteres es rechazada.

Autómatas finitos deterministas

Un autómata finito determinista (A_F^{D}) es un caso particular de los A_F^{ND} en el que la función de transferencia f no presenta ninguna ambigüedad (presenta una sola salida posible para una entrada dada).

Transformación de autómatas finitos no deterministas en deterministas

Teorema #1

$$\forall\, A_F^{ND} = (E, Q, f, Q_1, F) : \exists\, A_F^{D} = (E, Q', f', Q'_1, F) \mid L(A_F^{D}) = L(A_F^{ND})$$

– V –

Redes de Petri

Bases de las redes de Petri

Las redes de Petri son herramientas muy útiles que se usan para modelar una gran variedad de fenómenos de informática y control. Estas redes permiten modelar y visualizar comportamientos con paralelismo, concurrencia, sincronización e intercambio de recursos. Denotamos una red de Petri de la forma: P_N.

Componentes de una red de Petri

Una red de Petri se compone por dos tipos de elementos: nodos y arcos. A continuación se explican en detalle.

Nodo

Un nodo de una red de Petri puede ser de dos tipos: plaza o transición.

Plaza

Una plaza se simboliza en una red de Petri como un círculo. Las plazas no necesariamente representan datos o estados, pero para nuestro estudio tomamos la analogía de plazas con estados.

La cantidad de plazas que puede tener una red de Petri es un número finito $n \in \mathbb{N}$.

Si tenemos una red de Petri con m plazas, entonces el conjunto de plazas de esa red de Petri se denota:

$$P = \{P_1, ..., P_m\} = \{P_j \mid j \in [1, m] \land j, m \in \mathbb{N}\}$$

Transición

Una transición es un detector de condiciones y transformador de estados. Si las condiciones definidas por la transición se cumplen, decimos que la transición está *sensibilizada*, la transición se dispara y el estado de la red cambia.

Una transición se simboliza en una red de Petri como una barra recta. Es el equivalente a las transiciones de un autómata.

La cantidad de transiciones que puede tener una red de Petri es un número finito $n \in \mathbb{N}$.

Si tenemos una red de Petri con m transiciones, entonces el conjunto de transiciones de esa red de Petri se denota:

$$T = \{T_1, ..., T_m\} = \{T_j \mid j \in [1, m] \land j, m \in \mathbb{N}\}$$

Arco

Un arco se simboliza en una red de Petri como una flecha, y tiene dirección. Es obligatorio que cada arco tenga un nodo en cada uno de sus extremos. La cantidad de arcos posibles uniendo dos nodos es sólo uno (no pueden salir dos arcos de un mismo nodo con dirección a otro mismo nodo).

Las únicas dos direcciones posibles de un arco son:

- De una plaza a una transición.
- De una transición a una plaza.

Un arco puede tener "peso". El peso indica cuántos *tokens* se necesitan para sensibilizar una transición, o cuántos tokens se colocarán en una plaza.

Cuando un arco A es dirigido desde una plaza P a una transición T, decimos que tal plaza P es una plaza *upstream*. Esta transición se conoce como transición *de entrada*.

Cuando un arco A es dirigido de una transición T a una plaza P, decimos que tal plaza P es una plaza *downstream*. Esta transición se conoce como transición *de salida*.

Una transición sin una plaza de entrada es una transición *de fuente*. Una transición sin una plaza de salida es una transición *de sumidero*.

Marcas

Este concepto es sólo aplicable para plazas individualmente y para redes de Petri en su totalidad. La marca es un número finito $m \in \mathbb{N}$ que indica la cantidad de tokens que posee una plaza o una red de Petri.

La marca de una plaza P_i se denota de la forma: $m(P_i)$ ó m_i.

La marca de una red de Petri se define como un vector μ de $k \in \mathbb{N}$ componentes, siendo k la cantidad de plazas de la red de Petri. El componente j de μ es la cantidad de tokens de P_j.

Es decir:

$$\mu = (m(P_j) \mid j \in [0,\ k] \land j,\ k \in \mathbb{N})$$

Podemos pensar a un vector de marcas como una matriz de una fila por k columnas (dimensión $1 \times k$), cuya traspuesta es una matriz de una columna por k filas (dimensión $k \times 1$). Esto es:

$$\mu = (m(P_j)) = [m(P_j)]^T$$

La marca inicial con que comienza una red de Petri se denota con μ_0.

Disparo de una transición

Por motivos introductorios, vamos a limitarnos a dar una introducción al disparo de una transición considerando la misma con duración de 0 U.T. (unidades de tiempo).

Cuando la condición impuesta por una transición es cumplida, decimos que la transición está *sensibilizada*, y ésta puede ser *disparada*.

Para que una transición se dispare (*firing* de una transición), cada plaza de entrada a la transición tiene que tener al menos el mismo número de tokens que el peso del arco que conecta la plaza de entrada con la transición.

El firing consume de las plazas de entrada a la transición el número de tokens del peso del arco que une la plaza con la transición. También genera a las plazas de salida de la transición el número de tokens del peso del arco que une la transición con la plaza.

Cuando una transición está sensibilizada, no necesariamente será disparada inmediatamente. Esto sólo sigue siendo una posibilidad.

El firing de una transición es *atómico*; es decir, es indivisible, y tiene una duración cero (es instantáneo).

Clasificación de las redes de Petri

Red de Petri viva

Llamamos a una red de Petri "viva" cuando la red puede evolucionar.

Red de Petri sin interbloqueo

Es el mismo concepto que para los threads de un programa. Los componentes no deben interbloquearse.

Red de Petri reversible

Llamamos a una red de Petri "reversible" si la misma puede volver a su estado inicial.

Red de Petri libre de conflicto

Llamamos a una red de Petri "libre de conflicto" si todas las transiciones de la red pueden ser definidas lógicamente. Si una red tiene conflicto, hay que resolverlo aplicando alguna política. Esto se estudiará en detalle más adelante.

Red de Petri acotada

Llamamos a una red de Petri "acotada" o "limitada" (P_N^L) cuando la misma tiene limitada la cantidad máxima de tokens posibles.

Redes de Petri seguras

Llamamos a una red de Petri "segura" (P_N^S) cuando la cantidad de tokens que puede tener una plaza es sólo uno.

Propiedades simples para redes de Petri acotadas y seguras

Estas propiedades dependen fuertemente de μ_0.

- Basta con que μ_0 sea tal que al menos una plaza tenga dos tokens para que una P_N no sea $P_N{}^S$.
- Si una P_N no tiene límites para μ_0, $\Rightarrow$ P_N no tiene límites para una marca $\mu_1 \geq \mu_0$.
- No podemos extender directamente el concepto de $P_N{}^S$ a $P_N{}^{Cont}$ ya que las marcas de las plazas no son números enteros (la definición de $P_N{}^{Cont}$ se da más adelante).

Red de Petri autónoma

Una P_N cuyos instantes de firing de transiciones son desconocidos o no están indicados se considera una red de Petri "autónoma".

Estas P_N son "semi interpretadas", pues aportan algún significado a los elementos de la red, como por ejemplo:

- Las plazas describen variables de estado.
- Las transiciones y la lógica de sensibilización aportan algunas reglas para la descripción del comportamiento dinámico del sistema.

Red de Petri no autónoma

Una red de Petri cuyas transiciones se disparan cumpliendo condiciones de sensibilizado y están asociadas a un evento externo se considera una red de Petri *no autónoma.*

Estas redes de Petri describen el funcionamiento de un sistema cuya evolución está condicionada por eventos externos y/o por el tiempo.

Si queremos indicar en una transición el requisito para el firing luego de la sensibilización, tendremos que anexar una etiqueta a esta transición con dicha condición.

Definición formal de una red de Petri

Una red de Petri puede definirse como una terna de la forma: $P_N = (P, T, F)$, donde:

- P: *Plazas.*
- T: *Transiciones.*
- $F \subseteq (P \times T) \cup (T \times P)$: *Arcos de transición.*
  - $(P \times T)$: *Arcos de entrada a la transición.* $\forall x \in (P \times T) : x \triangleq \{\bullet\ t\}$
  - $(T \times P)$: *Arcos de salida de la transición.* $\forall x \in (T \times P) : x \triangleq \{t\ \bullet\}$

Si no se especifica, el peso de los arcos es, por defecto, 1.

Red de Petri ordinaria

Una red de Petri ordinaria ($P_N{}^O$) es un caso particular de una P_N donde el peso de todos los arcos es 1 y comenzamos con una marca inicial μ_0:

$$P_N{}^O = (\mu_0, P_N) = (\mu_0, (P, T, F))$$

Casos particulares de redes de Petri

Máquina de estados

Una máquina de estados (M_S) es una P_N donde cada transición tiene una plaza de entrada y una plaza de salida.

Esto es, matemáticamente:

$$|\bullet\ t| = |t\ \bullet| \leq 1\ \forall\ t \in T$$

El símbolo $\leq$ indica que puede darse el caso en el que la transición no posea un arco de entrada o de salida.

Por otro lado, debemos sumar la condición de que la cantidad total de tokens de P_N debe ser 1.

Grafo de marcado

Un grafo de marcado (G_M) es una P_N donde cada plaza tiene una transición de entrada y una de salida.

Esto es, matemáticamente:

$$|\bullet\ p| = |p\ \bullet| = 1\ \forall\ p \in P$$

Estos grafos suelen usarse principalmente para representar matemáticamente operaciones simultáneas. Esto significa que no puede haber conflicto (ya que una plaza es entrada y salida de una transición única), pero puede haber concurrencia.

Red de Petri libre de conflicto

En una red de Petri libre de conflicto (P_N^{CF}), cada plaza tiene, como máximo, una transición de salida. De esta forma se evitan los llamados "conflictos estructurales".

Un conflicto estructural se da cuando una plaza tiene más de una transición de salida. Lo llamamos "conflicto" porque una vez que la plaza entrega el token a una de las transiciones (las dos están sensibilizadas), la otra queda no sensibilizada y no podremos saber a cuál transición se le entregará el token.

En una red de Petri libre de conflicto, el número de tokens máximo que cualquier plaza puede recibir es limitado o está limitado linealmente por la marca inicial μ_0. Esto último significa, matemáticamente:

$$\exists\ c \in \mathbb{N} \mid \text{Si la cantidad de tokens iniciales de } \mu_0 = x \Rightarrow \text{máx}_{\text{tokens}}(P_i) = cx$$

Red de Petri de libre elección

Definimos matemáticamente una red de Petri de libre elección (P_N^{FC}) de la forma:

$$P_N^{FC} = \{P_N \mid \forall\ P_1, P_2 \in P \wedge P_1 \neq P_2 : P_1 \bullet \cap P_2 \bullet = \emptyset\}$$

En palabras, una P_N^{FC} es una P_N donde una plaza tiene más de un arco de salida, por lo que tendremos dos o más transiciones a las que podemos llegar desde la plaza, pero no puedo acceder a estas transiciones desde otras plazas (los arcos de salida de P_1 no llevan a las transiciones que llevan los arcos de P_2).

Red de Petri de libre elección extendida

Una red de Petri de libre elección extendida (P_N^{FCE}) es una P_N^{FC} tal que para cada conflicto donde una plaza P_1 puede llevar a transiciones T_1 y T_2, estas transiciones tienen el mismo conjunto de plazas de entrada. Es decir, si T_1 tiene a P_1 y P_2 como plazas de entrada, entonces T_2 también tiene a P_1 y a P_2 como plazas de entrada.

Podemos definirla matemáticamente de la forma:

$$P_N^{FCE} = \{P_N^{FC} \mid \forall\ P_1,P_2 \in P \text{ donde } P_1,P_2 \text{ están en conflicto} \wedge P_1 \neq P_2 : P_1 \bullet = P_2 \bullet\}$$

Red de Petri simple

Una red de Petri simple (P_N^{S}) es una P_N en la que cada transición sólo puede verse involucrada en un conflicto como máximo.

Para este tipo de redes tenemos su definición matemática de la forma:

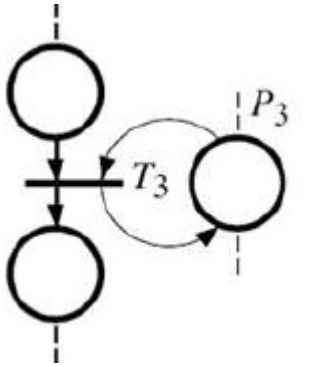
$$P_N^{S} = \{P_N \mid \forall\ P_1, P_2 \in P \wedge P_1 \neq P_2 : P_1 \bullet \cap P_2 \bullet = \emptyset\}$$

Red de Petri pura

Definimos primero lo que se conoce como "autoloop":

Llamamos "autoloop" al par $(P_i, T_j) \Leftrightarrow P_i$ es plaza de entrada y salida de T_j.

Gráficamente:



Decimos, entonces, que una P_N es "pura" ($P_N{}^P$) si no posee autoloops.

Matemáticamente:

$$P_N{}^P = P_N \mid \nexists\ (P_i, T_j) : P_i \text{ es plaza de entrada y salida de } T_j.$$

Extensión de conceptos para redes de Petri

Arco inhibidor

Así como existen los arcos comunes que conectan una plaza con una transición y notifican a esa transición si la plaza posee o no la cantidad de tokens necesaria indicada por el peso del arco, también existen los llamados *arcos inhibidores*, cuyo propósito es el opuesto.

Los arcos inhibidores cumplen la tarea de sensibilizar una transición sólo si la plaza cuyo arco de salida es inhibidor no contiene tokens.

Los arcos de salida se simbolizan como un arco normal, pero en vez de tener una punta de flecha tienen un círculo vacío.

Prioridad

Antes de proceder con la definición de prioridad, establecemos la notación para un conflicto entre plazas y transiciones.

Si tenemos una plaza P que posee n arcos que llevan a n transiciones y la capacidad máxima de P es cap(P) = 1, entonces denotamos este conflicto de la forma:

$$K = <P, \{T_i \mid i \in [m,\ m + n] \land i,\ m,\ n \in \mathbb{N}\}>$$

El concepto de prioridad es: si tenemos un conflicto de la forma:
$K = <P, \{T_i \mid i \in [m,\ m + n] \land i,\ m,\ n \in \mathbb{N}\}>$, podemos introducir prioridades sobre las transiciones T_i para evitar la incertidumbre.

Ejemplo:

Tenemos un conflicto $K = <P, \{T_1, T_2\}> = <P, \{T_i \mid i \in [0, 2] \land i \in \mathbb{N}\}>$ y queremos que, cuando ambas transiciones estén sensibilizadas, se entregue el token a T_2, y cuando T_2 no esté sensibilizada, se entregue el token a T_1.

De esta forma, colocamos una etiqueta junto a la parte de la P_N donde se presenta este conflicto K con la inscripción:

$$T_1 < T_2$$

Y así le indicamos a la P_N que T_2 tiene prioridad por sobre T_1.

Peso de un arco

Podemos agregarle a un arco un complemento de "peso". Esto puede indicar dos cosas:

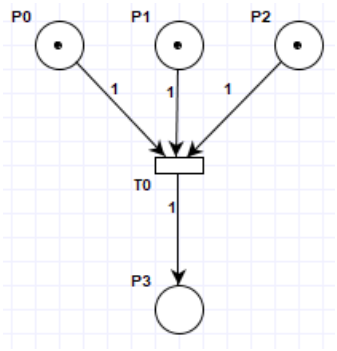
- **Si el arco une una plaza con una transición**: En este caso, el peso del arco indicará cuántos tokens son necesarios para sensibilizar la transición (estos tokens son quitados de la plaza origen).
- **Si el arco une una transición con una plaza:** En este caso, el peso del arco indicará cuántos tokens serán agregados a la plaza destino una vez se haya disparado la transición.

Conceptos modelados con redes de Petri

Sincronización

Una forma de representar la sincronización mediante una P_N puede ser con n plazas que conducen a una misma transición.

Gráficamente:



Confusión simétrica

El concepto de "confusión simétrica" es una combinación de conflicto y concurrencia. La forma más compacta, gráfica y fácil de representar y comprender la confusión simétrica es con dos plazas P_1 y P_2 y tres transiciones T_1, T_2 y T_3, donde el conflicto de P_1 es:

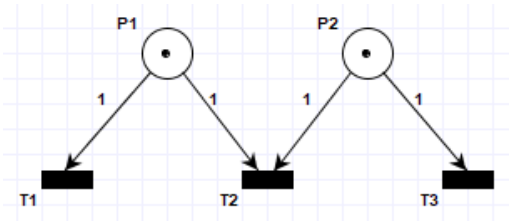
$$K_1 = <P_1, \{T_1, T_2\}> = <P_1, \{T_i \mid i \in [0, 2] \land i \in \mathbb{N}\}>$$

y el conflicto de P_2 es:

$$K_2 = <P_2, \{T_2, T_3\}> = <P_2, \{T_i \mid i \in [2, 3] \land i \in \mathbb{N}\}>$$

En este caso, al resolver, por ejemplo, el conflicto K_1, estamos quitándole la sensibilidad a la transición T_2, pero la transición T_3 sigue sensibilizada.

Gráficamente:

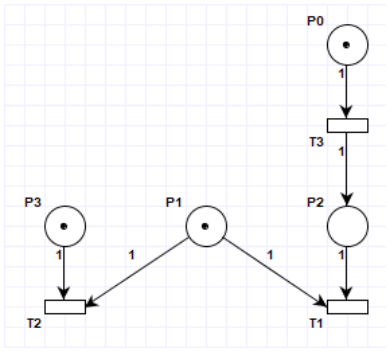


Confusión asimétrica

La confusión asimétrica se da cuando se causa una confusión simétrica en un caso particular.

Si tenemos una plaza P_1 con dos arcos de salida que llevan a dos transiciones distintas T_1 y T_2 (un conflicto), y una de estas transiciones tiene un arco de entrada adicional proveniente de una plaza P_2 que tiene un arco de entrada proveniente de una transición sensibilizada T_3 (no sensibilizada por P_1) y P_2 entrega el token que recibe por T_3 hacia la transición compartida con P_1 justo antes de que P_1 entregue su token a la otra transición, pasaremos a estar en la situación de confusión simétrica; es decir, si en este punto entregamos un token a una transición (ambas están sensibilizadas) automáticamente le quitamos la sensibilidad a la otra transición.

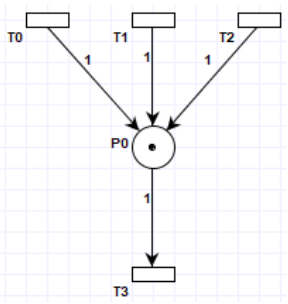
Gráficamente:



Merging

Es un concepto similar pero distinto a la sincronización. En el merging no tenemos nada que requiera que los tokens lleguen a la vez. Es decir, para la sincronización teníamos plazas $P_i \mid cap(P_i) = 1$ que sensibilizaban la transición T sólo cuando todas las plazas tenían su token. Para el caso del merging, tenemos transiciones T_i que concurren a una plaza P; por lo tanto no tenemos que esperar a nadie, cada token ingresa cuando la transición sensibilizada se dispara.

Gráficamente:

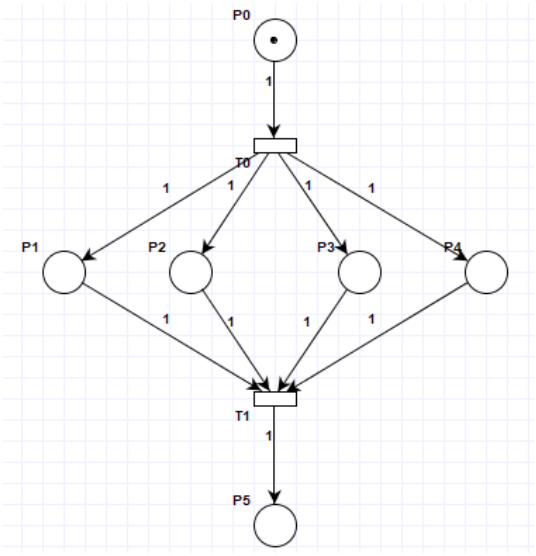


Cobegin y coend

El concepto de "cobegin" es aquel que permite indicar la ejecución concurrente de varios procesos. Es decir, una plaza origen sensibiliza una transición T y T le asigna un token a cada plaza destino.

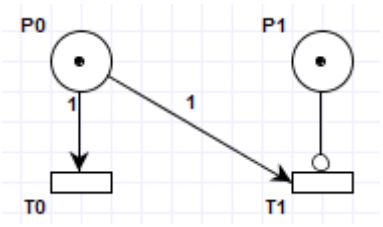
Luego, podemos agregar el concepto de "coend" que es similar pero a la inversa; es decir, varias plazas origen sensibilizan una transición que entrega un token a una sola plaza destino.

Gráficamente:



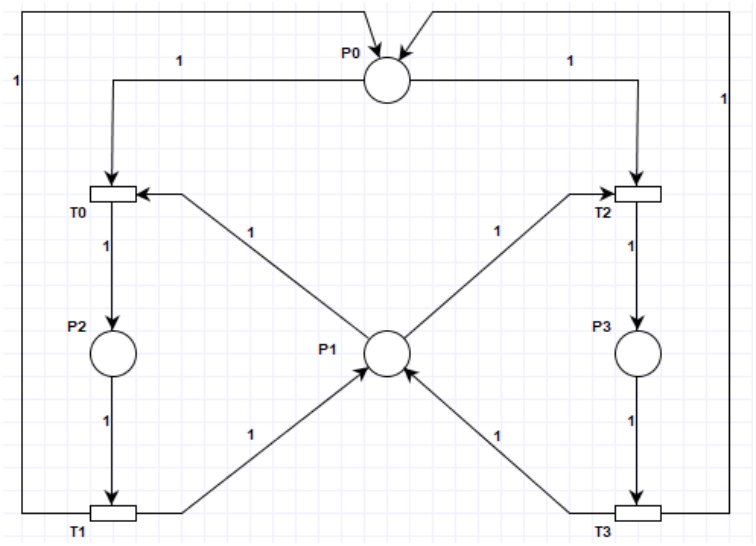
Prioridad

Podemos simbolizar la prioridad en una P_N utilizando arcos inhibidores.
Por ejemplo:



Sistema read-write con exclusión mutua

Este sistema puede representarse con una P_N de la siguiente forma:



Extensión de tipos de redes de Petri

Red de Petri no autónoma

Podemos redefinir a una Red de Petri no autónoma (P_N^{NA}) como una P_N que está sincronizada y/o con eventos exteriores a la red.

Cabe resaltar que una P_N^{NA} no puede convertirse en una P_N^{O}.

Red de Petri continua

Denotamos una red de Petri continua de la forma: P_N^{Cont}.

En estas P_N el marcado de una plaza es un número real positivo o cero.

Matemáticamente, para una P_N^{Cont} con m plazas:

$$P_N^{Cont} = P_N \mid \forall\, i \in [1,\, m] \land i,\, m \in \mathbb{N} : m(P_i) = n \in \mathbb{R}^+ \cup \{0\}$$

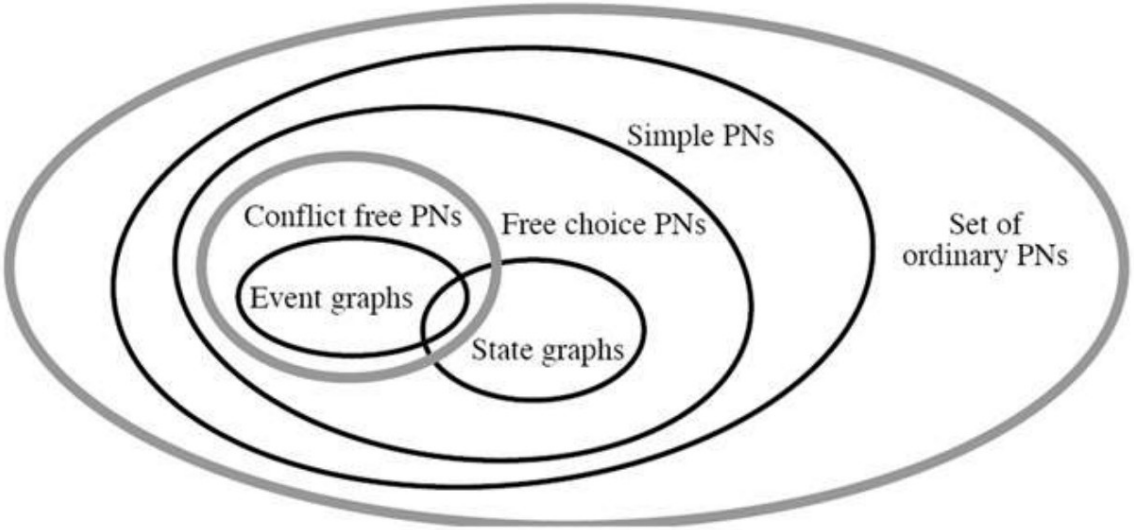
Además, el firing de una transición tiene lugar como un flujo continuo. Estas redes permiten modelar sistemas que no pueden modelarse mediante una P_N^{O}.

Relación entre distintos tipos de redes de Petri

Podemos establecer una relación matemática del siguiente tipo para los tipos de redes de Petri descritos anteriormente:

$$(G_M \subset P_N^{CF} \land M_S) \subset P_N^{FC} \subset P_N^{S} \subset P_N^{O}$$

Gráficamente:



Propiedades de las redes de Petri

Notación y definiciones

Secuencia de firings

Definimos una secuencia de firings S como la secuencia de transiciones sensibilizadas que fueron disparadas y nos llevaron desde la marca inicial μ_0 hasta una marca μ.

Si la secuencia de firings consta de disparar (en orden) las secuencias T_i, T_{i+1}, ... , T_{i+m}, entonces denotamos a esa secuencia de la forma:

$$S = T_j \mid j \in [i, i+m] \land i, j, m \in \mathbb{N}$$
$$\implies S = T_i T_{i+1} ... T_{i+m}$$

Secuencia de firings finita

Una secuencia de firings finita es una secuencia de firings de transiciones que puede efectuarse de principio a fin un cierto número de veces, a partir del cual deja de ser posible la ejecución de esta secuencia.

Denotamos al conjunto de secuencias de firings de transiciones finitas de la forma T^F cuyos elementos se denotan con σ_F.

Secuencia de firings infinita

Una secuencia de firings infinita es una secuencia de firings de transiciones que puede efectuarse de principio a fin tantas veces como se desee sin impedimento alguno. Podemos repetir esta secuencia infinitamente.

Denotamos al conjunto de secuencias de firings de transiciones infinitas de la forma $T^{\neg F}$ cuyos elementos se denotan con $\sigma_{\neg F}$.

El conjunto de secuencias finitas e infinitas lo denotamos de la forma: $T^\infty = T^F \cup T^{\neg F}$. Los elementos de este conjunto se denotan con σ.

Decimos entonces, matemáticamente, que la condición de existencia de una secuencia de firings infinita $\sigma_{\neg F}$ es:

$$\exists\, \sigma_{\neg F} \Leftrightarrow \exists\, \sigma \in T^\infty \mid \forall\, \sigma' \subseteq \sigma : \sigma' \in T^F$$

Para una mejor notación, denotamos al conjunto de todas las secuencias de firings de transición posibles de una P_N de la forma: $\mathbb{S}$.

Marcas

Si bien las marcas ya fueron introducidas con anterioridad, ahora agregamos conceptos nuevos relacionados a estas.

Vector de marcas global

Así como definimos el vector de marcas μ para una P_N de tal forma que μ contiene las marcas de cada plaza de la P_N, ahora definimos un vector de marcas *global* que indica todas las marcas posibles de obtener en esta P_N partiendo desde μ_0.

Si la P_N puede tener m marcas, entonces denotamos al vector de marcas global de P_N de la forma:

$$\mathfrak{M}(\mu_0) = \{\mu_j \mid j \in [0,\ m] \land j,\ m \in \mathbb{N}\} = \{\mu_0,\ ...,\ \mu_m\}$$

Secuencia de firings que alcanza una marca

Denotamos que, a través del firing de una secuencia S podemos pasar de una marca μ_i a una marca μ_j de la forma:

$$\mu_i \xrightarrow{S} \mu_j$$

Marca alcanzable

$$\forall\ \mu_i \in \mathfrak{M}(\mu_0) : \mu_i \text{ es alcanzable} \Leftrightarrow \exists\ \mu_j \in \mathfrak{M}(\mu_0) \land \exists\ S \in \mathbb{S} \mid \mu_j \xrightarrow{S} \mu_i$$

Red de Petri pseudo-viva

Notación: P_N^{PV}

Definición matemática:

$$P_N^{PV} = P_N \mid \forall\ \mu \in \mathfrak{M}(\mu_0) : \exists\ t \in T \mid \mu \xrightarrow{t}$$

Esto significa que una red de Petri es pseudo-viva si existe al menos una transición viva (transición que puede ser disparada), por lo que la red de Petri no se bloquea totalmente.

Red de Petri quasi-viva

Debemos primero definir este concepto para una transición.
Una transición es quasi-viva si se puede disparar al menos una vez.

Ahora, la definición matemática de una red de Petri quasi-viva (P_N^{QV}) es:

$$P_N^{PV} = P_N \mid \forall\ t \in T : \exists\ m \in \mathfrak{M}(\mu_0) \mid m \xrightarrow{t}$$

Es decir, una red de Petri es quasi-viva si contiene al menos una transición quasi-viva.

Vida de una red de Petri (liveness)

El marcado de una P_N evoluciona mediante el firing de transiciones. Cuando algunas transiciones ya no son habilitadas y toda (o parte de) la red ya "no funciona", es probable que haya un problema en el diseño del sistema descrito (o puede que no, un OS no termina).

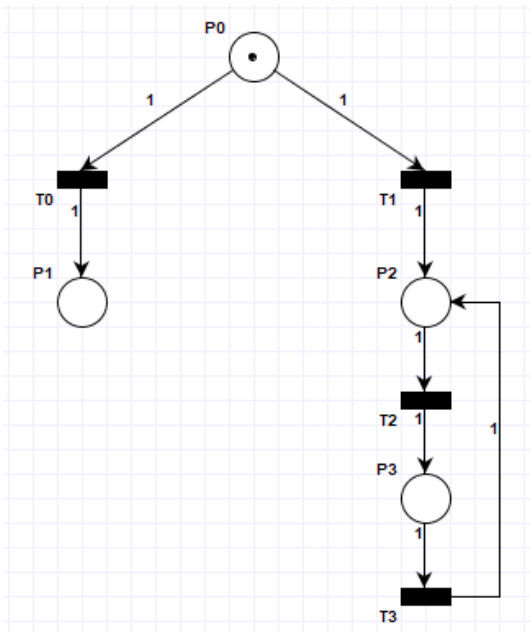
Para hacer énfasis en esto, definimos a una transición T como transición "viva" para una marca inicial μ_0 de la forma:

$$T \text{ es viva} \Leftrightarrow \forall\ \mu_j \in \mathfrak{M}(\mu_0) : \exists\ S \in \mathbb{S} \mid T \subseteq S$$

Deadlock en una red de Petri

Un deadlock para una red de Petri es similar a un deadlock para un thread o un proceso. En una red de Petri podemos interpretar un deadlock como una plaza que tiene una transición de entrada y ninguna transición de salida. Aquí, la plaza se queda con el token y no lo transfiere a ninguna transición, por lo que se corre el riesgo de dejar "muerta" una parte de la red de Petri o a la red completa.

He aquí un ejemplo:

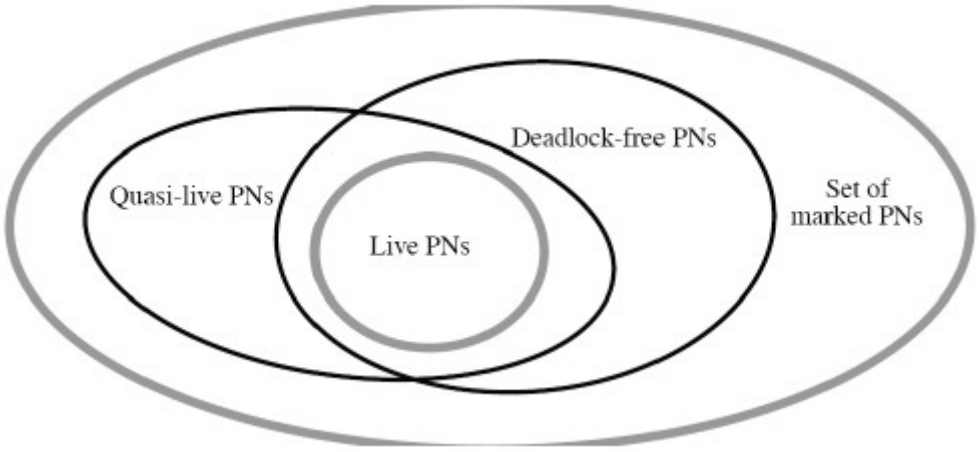


En este caso, la P_N posee un solo token, por lo que si se resuelve el conflicto $K = <P_0, \{T_i \mid i \in [0, 1] \land i \in \mathbb{N}\}>$ optando por T_0, dejamos inutilizable el resto de la red, ya que el único token es tomado por la plaza P_1 y todas las transiciones mueren.

Decimos que una red de Petri está libre de interbloqueo para una marca inicial μ_0 si:

$$\nexists\ \mu_i \in \mathfrak{M}(\mu_0) \mid \mu_i \text{ denota un deadlock}$$

Relación entre liveness y deadlock



Propiedades de liveness y deadlock

- Si $\exists\ t \in T \mid t$ es quasi-viva para $\mu_0 \Rightarrow t$ es quasi-viva para $\mu_1 \geq \mu_0$
- Si $\exists\ t \in T \mid t$ es viva para $\mu_0 \Rightarrow t$ no es necesariamente viva para $\mu_1 \geq \mu_0$
- Si $\exists\ P_N \mid P_N$ está libre de deadlocks para μ_0, no necesariamente es así para $\mu_1 \geq \mu_0$
- Los conceptos de liveness, quasi-liveness y deadlock se aplican a todas las P_N vistas y pueden generalizarse a todas las extensiones de P_N.

Definiciones útiles

Home state

Denotamos un "*home state*" de la forma: μ_H.
Definición matemática:

$$P_N \text{ tiene un } \mu_H \text{ para un } \mu_0 \Leftrightarrow \forall\ \mu_i \in \mathfrak{M}(\mu_0) : \exists\ S \in \mathbb{S} \mid \mu_i \xrightarrow{S} \mu_H$$

Es decir, son marcas alcanzables desde cualquier marcado en el que la red de Petri se pueda encontrar. La existencia de un home state depende de la marca inicial.

Home space

El conjunto de todos los μ_H es conocido como "*home space*" y lo denotamos de la forma: $\mathfrak{M}_{HS}$

Red de Petri reversible

Notación: $P_N{}^R$.
Definición matemática:

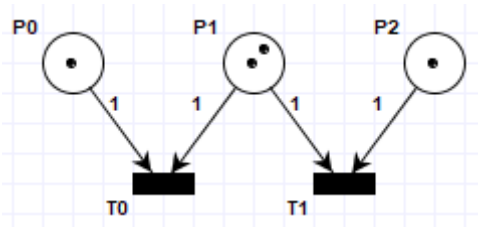
$$P_N{}^R = P_N \mid \exists\ \mu_H \in \mathfrak{M}(\mu_0)\ \mu_H = \mu_0$$

El concepto de una $P_N{}^R$ ya fue introducido anteriormente.

Conflicto efectivo

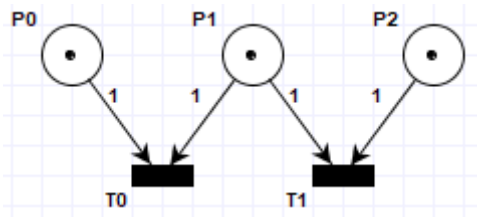
Recordando que una confusión simétrica es definida como la situación en la que al menos dos plazas entran en disputa sobre qué transición sensibilizada disparar, dejando al menos una sin sensibilizar, podemos extender este concepto agregando el rol de la marca de la P_N, logrando el concepto de conflicto efectivo.

Si tenemos un conflicto como el siguiente:



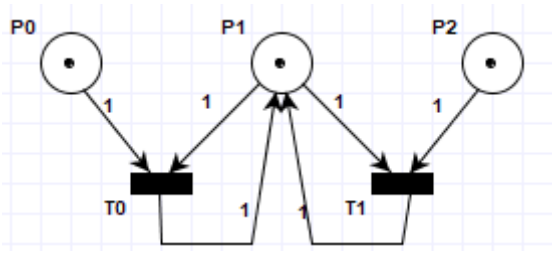
decimos que tenemos un conflicto no efectivo, ya que P_1 tiene dos tokens, y si se resuelve el conflicto por T_0, aún tengo posibilidad de disparar T_1 porque sigue sensibilizada. Aquí podemos disparar ambas transiciones (una a la vez o incluso ambas al mismo tiempo).

Si tenemos un conflicto como este:



tenemos un conflicto efectivo, ya que si se resuelve el conflicto por T_0, T_1 ya no queda sensibilizada. Aquí sólo podemos disparar una transición.

Si tenemos un conflicto como el que se muestra a continuación:



tenemos un conflicto efectivo y persistente, ya que disparar una transición no mata completamente a la otra, sino que el token de P_1 vuelve a sí mismo y permite el firing de la otra transición. Esta configuración nos permite disparar ambas transiciones pero sólo una a la vez.

Denotamos matemáticamente un conflicto efectivo ligado a una marca μ de la forma:

$$K^E = <P, \{T_i \mid i \in [m,\ m+n] \land i,\ m,\ n \in \mathbb{N}\},\ \mu>$$

Múltiples firings

Cuando tenemos una situación en la que podemos disparar más de una transición a la vez, entramos en el concepto de múltiples firings.

Denotamos de la forma $[T_1T_2]$ al firing *múltiple* de T_1 y T_2. Esto significa que las transiciones T_1 y T_2 se disparan *en simultáneo*.

Denotamos de la forma $\{T_1T_2\}$ al firing *concurrente* de T_1 y T_2. Esto significa que las transiciones T_1 y T_2 se disparan de manera *concurrente*.

Cuando pasamos de una marca μ_0 a una marca μ_1 mediante un multiple firing de transiciones, lo denotamos de la forma:

$\mu_0 \xrightarrow{[T_1T_2]} \mu_1$ si el firing es simultáneo.

$\mu_0 \xrightarrow{\{T_1T_2\}} \mu_1$ si el firing es concurrente.

Transiciones concurrentes

Decimos que dos o más transiciones son simultáneas si están sensibilizadas y son causalmente independientes. Esto es, una transición puede dispararse antes, después, o simultáneamente con la otra.

En la mayoría de los casos, estas transiciones no tienen una plaza de entrada común.

Si las transiciones tienen una plaza de entrada común, entonces tienen que haber suficientes tokens para sensibilizar ambas transiciones.

Si por cada marca alcanzable no hay conflicto efectivo, entonces en cualquier momento las transiciones habilitadas son concurrentes. Esto es, matemáticamente:

$$\text{Si } \forall\, \mu \in \mathfrak{M}(\mu_0) : \nexists\ K^E = <P, \{T_i \mid i \in [m,\ m+n] \wedge i,\ m,\ n \in \mathbb{N}\}, \mu> \Rightarrow \forall\ (T_i,\ T_j) \in T : \text{si Ti, Tj están sensibilizadas} \Rightarrow T_i,\ T_j \text{ son concurrentes}$$

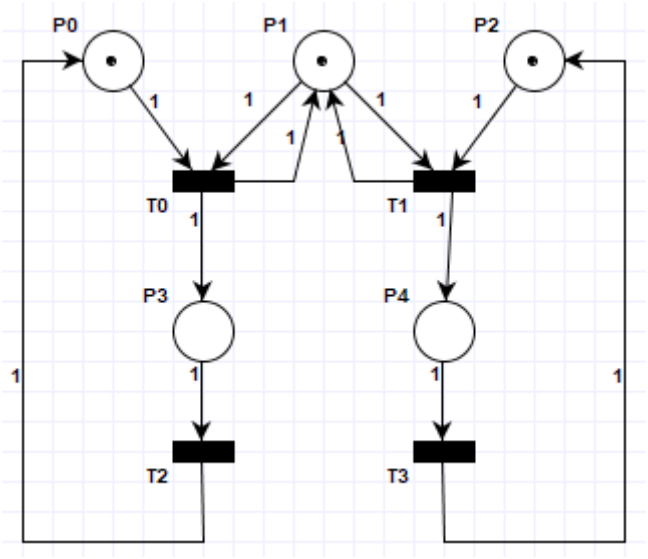
Extensión de propiedades de las redes de Petri

Red de Petri persistente

Decimos que una P_N es persistente (P_N^{Per}) si para cada marca alcanzable $\mu_i \in \mathfrak{M}(\mu_0)$ existe la siguiente propiedad:

$\forall$ (T_i, T_j) $\in$ T : T_i, Tj están sensibilizadas marcando $\mu_i \Rightarrow$ S = T_iT_j es una secuencia de firing de μ_i, así como S' = T_jT_i (simetría)

En otras palabras, decimos que una P_N es persistente si y sólo si para dos transiciones habilitadas, el firing de una no desactiva la otra. Si una P_N no tiene un K^E; es decir, las transiciones sensibilizadas son siempre concurrentes, entonces la P_N es persistente. Por otro lado, si hay un K^E entre dos transiciones, la persistencia es posible si existe una autointerrupción entre la plaza considerada y cada transición, pero estas transiciones no son concurrentes. Esto puede verse en el siguiente caso:



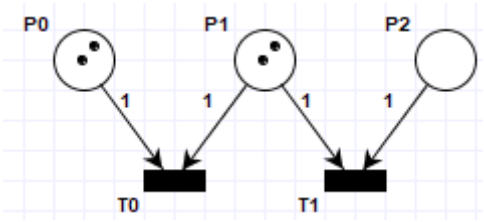
Cabe resaltar que, en una P_N^{Per}, una transición sensibilizada permanecerá así hasta que se dispare.

Grado de sensibilidad

La habilitación de una transición se considera como una propiedad booleana. Es decir, la transición puede tener sólo dos estados: *sensibilizada* o *no sensibilizada*.

Decimos que el grado de sensibilización es igual al número de veces que puede dispararse una transición.

Como ejemplo, en el siguiente caso:



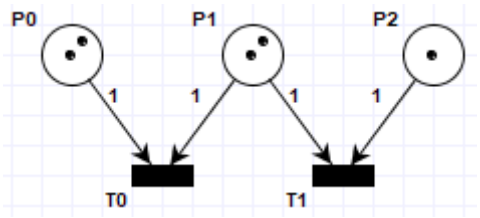
la transición T_0 puede dispararse dos veces, por lo que su grado de sensibilidad es 2.

Conflicto general

Concepto

Un conflicto general es un tipo de conflicto en el que un cierto número de transiciones no se pueden disparar simultáneamente de acuerdo con los grados de sensibilidad de la plazas.

En el siguiente caso:



los múltiples firings $\{T_0T_0\}$ y $\{T_0T_1\}$ son posibles, pero no $[T_0T_0T_1]$.

Definición formal

Un conflicto general K^G es la existencia de un conflicto estructural $K^E = <P_i, \{T_i \mid i \in [m, m+n] \land i, m, n \in \mathbb{N}\}, \mu>$ siendo μ una marca tal que el número de tokens en P_i no es suficiente para disparar todas las transiciones de salida de P_i según sus grados de sensibilidad.

Denotamos un conflicto general de la forma:

$$K^G = <P, \{T_i \mid i \in [m, m+n] \land i, m, n \in \mathbb{N}\}, \mu>$$

Invariantes

A partir de una marca inicial μ_0, el marcado de una P_N puede evolucionar mediante el firing de transiciones, y si no hay un deadlock, el número de firings es ilimitado. Sin embargo, no se puede alcanzar cualquier marca. Todas las marcas alcanzables tienen algunas propiedades en común. Decimos que una propiedad que no varía cuando se activan las transiciones es *invariante*.

Del mismo modo, no se puede disparar cualquier secuencia de transición. Algunas propiedades invariantes son comunes a las posibles secuencias de firing.

Los invariantes son importantes porque nos permiten caracterizar ciertas propiedades de las marcas alcanzables y de las transiciones inalterables, independientemente de la evolución.

Repetitividad

Las secuencias de firing S_R que nos llevan de μ_0 a μ_0 son conocidas como secuencias repetitivas, y las denotamos de la forma:

$$\mu_0 \xrightarrow{S_R} \mu_0$$

Una secuencia repetitiva que contiene todas las transiciones, al menos una vez cada una, se conoce como una secuencia repetitiva completa.

Una secuencia $S_K \mid \mu_0 \xrightarrow{S_K} \mu'_0 \geqq \mu_0$ se conoce como una *secuencia creciente repetitiva*. Si no se proporciona precisión, la expresión *secuencia repetitiva* hará referencia a una secuencia repetitiva estacionara: $S_K \mid \mu_0 \xrightarrow{S_K} \mu_0$.

El concepto de repetitividad implica que la sensibilización de alguna transición (o todas) ocurre infinitamente a menudo para alguna secuencia de firings.

Decimos que, dada una P_N con su conjunto T, y dada una secuencia repetitiva S_K tal que las transiciones que aparecen en S_K están definidas por $T' = \{T_i \mid i \in [0, r] \land i, r \in \mathbb{N}\} \subseteq T \Rightarrow T'$ es un componente repetitivo.

Matemáticamente:

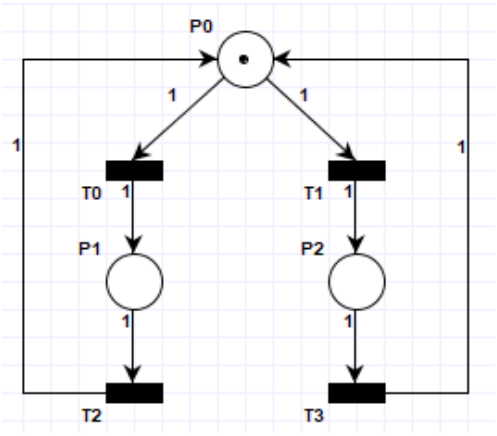
$$\forall P_N = (P, T, F) : \text{Si } \exists S_K \mid \mu_0 \xrightarrow{S_K} \mu_0 \land \forall T \in S_K : T' = \{T_i \mid i \in [0, r] \land i, r \in \mathbb{N}\} \subseteq T \Rightarrow T' \text{ es un componente repetitivo.}$$

Consistencia

El concepto de consistencia implica que alguna transición (o todas) ocurre/n al menos una vez en alguna secuencia de firings S_R que impulsa un μ_0 a sí mismo ($\mu_0 \xrightarrow{S_R} \mu_0$).

Ejemplo

En este ejemplo de una P_N:



podemos identificar dos secuencias repetitivas (logran $\mu_0 \xrightarrow{S_i} \mu_0$):

$$S_1 = T_0T_2$$
$$S_2 = T_1T_3$$

Y los dos componentes repetitivos son:

$$T'_1 = \{T_0,T_2\}$$
$$T'_2 = \{T_1,T_3\}$$

Ninguno de estos componentes contiene todas las transiciones de la P_N.

Podemos escribir al conjunto de todas las secuencias repetitivas como una expresión regular de la forma:

$$L = (T_0T_2 + T_1T_3)^*$$

Finalmente, la secuencia:

$$S = T_0T_2T_1T_3$$

es una secuencia repetitiva completa.

Como aquí todas las transiciones se disparan al menos una vez en la secuencia repetitiva completa S, decimos que esta P_N es consistente.

Propiedad

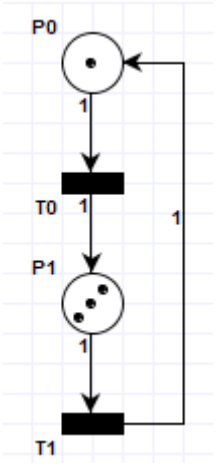
Sea $\mathcal{L}(\mu_0)$ el conjunto de secuencias de firings desde μ_0:

$$\text{Si } \mu'_0 \geq \mu_0 \Rightarrow \mathcal{L}(\mu_0) \subseteq \mathcal{L}(\mu'_0)$$

En particular, si S es una secuencia repetitiva para μ_0, entonces lo será también para μ'_0.
Esta propiedad no es válida para P_N que contengan arcos inhibidores.

Las secuencias repetitivas nos dan una idea del comportamiento cíclico de una P_N, y podemos aspirar a expresar esta relación más matemáticamente.

Consideremos la siguiente P_N:



Aquí vemos que la cantidad de veces que podemos disparar T_0 sin disparar T_1 es 1, y la cantidad de veces que podemos disparar T_1 sin disparar T_0 es 3. Para describir esto en términos matemáticos, introducimos la siguiente notación:

$$\text{número de firings de } T_j \text{ en } S_K \coloneqq N_K(T_j)$$

De esta forma, tenemos:

$$N_K(T_0) = 1$$
$$N_K(T_1) = 3$$

Finalmente, podemos decir que:

$$-3 \leq N_K(T_0) - N_K(T_1) \leq 1$$

Propiedades estructurales

Limitación estructural (P_N^{SB})

$\forall\ P_N : P_N = P_N^{SB} \Leftrightarrow \exists\ X > 0 \mid X^T C \leq 0$ siendo X el vector de estados y C la matriz de adyacencia

Conservación ($P_N^{Conserv}$)

$\forall\ P_N : P_N = P_N^{Conserv} \Leftrightarrow \exists\ X > 0 \mid \forall\ M \in \mathfrak{R}(\mu_0) : X^T M = X^T M_0$ siendo $\mathfrak{R}(\mu_0)$ el conjunto de marcas alcanzables desde μ_0.

Repetitividad (P_N^{Rep})

$\forall\ P_N : P_N = P_N^{Rep} \Leftrightarrow \exists\ Y > 0 \mid CY \geq 0$ siendo Y una secuencia de firings.

En otras palabras, una P_N es repetitiva si existe μ_0 y una secuencia de firings de modo que cada transición aparece infinitamente repetida.

Consistencia ($P_N^{Consist}$)

$\forall\ P_N : P_N = P_N^{Consist} \Leftrightarrow \exists\ Y > 0 \mid CY = 0$

Invariante de plaza

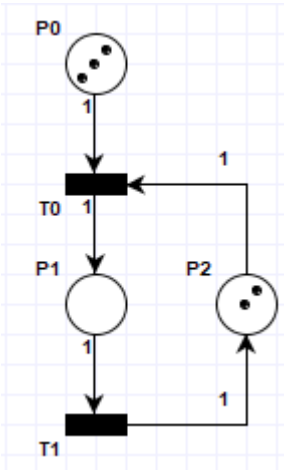
Veamos cómo podemos obtener un invariante de plaza P_I (también llamado *invariante de marca*, *invariante de plaza lineal*, o simplemente *invariante*):

Podemos obtener un P_I si:

- $\exists$ P' = $\{p_i \mid i \in [1, r] \land i, r \in \mathbb{N}\} \subseteq P$.
- $\exists$ un vector de ponderación $V_P = (q_i \mid i \in [1, r] \land i, r \in \mathbb{N}), q_i \in \mathbb{N} \mid \forall \mu \in \mathfrak{M}(\mu_0)$:
  $\sum_{i=1}^{r} q_i m(P_i) = cte.$, y cada q_i es un "peso".

Si bien el conjunto P' es un componente conservador, y la propiedad de ser un componente conservador es independiente de μ_0 (porque es una propiedad estructural), la constante del marcador invariante depende de μ_0.

Por ejemplo, consideremos la siguiente P_N:



En este caso, tenemos:

$$\mu(P_2) + \mu(P'_2) = 2$$

pero si hubiésemos tenido un μ_0 tal que $\mu_0(P_2) + \mu_0(P'_2) = N$, entonces tendríamos esa misma ecuación para cada marca alcanzable μ.

Podemos aclarar que, en términos generales, un componente conservador tiene un significado físico. En un ejemplo de una red de Petri con cuatro plazas y cuatro transiciones una detrás de la otra representando las estaciones podremos ver que sólo puede haber un token en las plazas y no puede haber un token en dos plazas o más a la vez. Esto podemos interpretarlo como la limitación física de que no puede ser verano e invierno a la vez, por ejemplo.

Grafo de marcas

El grafo de marcas (también conocido como gráfico de alcanzabilidad) está formado por vértices que corresponden a marcas alcanzables representadas por vectores de n filas y una columna, y arcos correspondientes a firings o transiciones.

Es decir, comenzamos desde una marca inicial μ_0 representando los tokens que poseen las plazas de la P_N de la forma:

$$\left[\begin{matrix} m(P_1) \\ ... \\ m(P_n) \end{matrix}\right]_{\mu_0} = \left[m(P_i) : i \in [1,n] \wedge i,n \in \mathbb{N}\right]_{\mu_0}$$

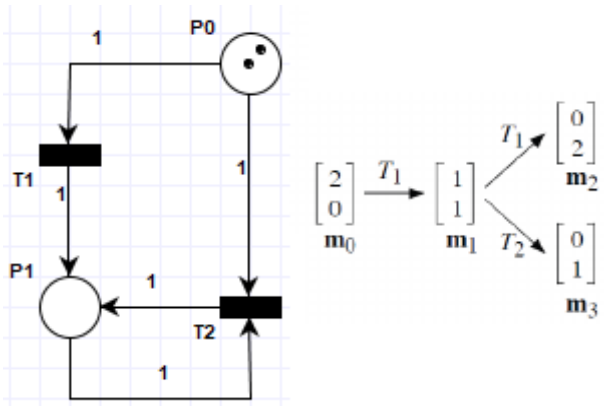
Aquí representamos que la P_N tiene n plazas, siendo que la plaza P_1 tiene $m(P_1)$ tokens y así sucesivamente hasta la plaza P_n que tiene $m(P_n)$ tokens. Denotamos que esta marca pertenece a la marca inicial escribiendo μ_0 en el costado inferior derecho. Si correspondiera a una marca μ_m, tendríamos que reemplazar μ_0 por μ_m, escribiendo el número de tokens correspondientes a cada plaza en su posición respectiva.

Si desde esta marca evolucionamos a una marca μ_1 a través de una transición T_1, entonces lo simbolizamos de la forma:

$$\left[\begin{matrix} m(P_1) \\ ... \\ m(P_n) \end{matrix}\right]_{\mu_0} \xrightarrow{T_1} \left[\begin{matrix} m(P_1)^1 \\ ... \\ m(P_n)^1 \end{matrix}\right]_{\mu_1} = \left[m(P_i) : i \in [1,n] \wedge i,n \in \mathbb{N}\right]_{\mu_0} \xrightarrow{T_1} \left[m(P_i)^1 : i \in [1,n] \wedge i,n \in \mathbb{N}\right]_{\mu_1}$$

Si tuviésemos dos posibilidades (dos transiciones sensibilizadas), simplemente agregamos las flechas que llevan a la marca correspondiente con la transición que haya sido disparada.

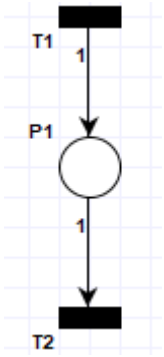
En el siguiente ejemplo se puede interpretar mejor este concepto:



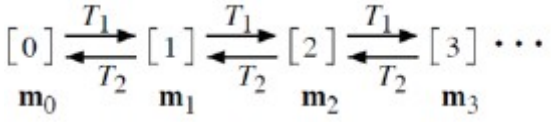
Con estos grafos de marcas podemos encontrar las propiedades de nuestra P_N, por ejemplo si es viva, segura, reversible, etcétera.

Árbol raíz de cobertura

Consideremos la siguiente P_N:



Si quisiéramos armar el grafo de marcas de esta P_N, tendríamos algo así:



Vemos que en este grafo tenemos un número finito de vértices. Por esto no podemos representarlo completamente, así que vamos a recurrir a una representación gráfica mediante un árbol raíz de cobertura *(coverability root tree)*.

Para esto, comenzamos por μ_0 (la raíz, *root*). Aquí tenemos sólo una transición sensibilizada (T_1) que cumple: $\mu_0 \xrightarrow{T_1} \mu_1$.

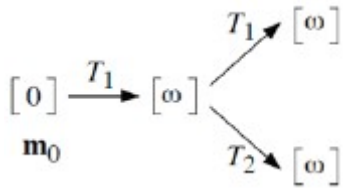
Desde $\mu_1 \gneqq \mu_0$ podemos disparar T_1 tantas veces como queramos, sin límite. Es aquí donde queremos resolver esta indeterminación introduciendo el símbolo ω tal que:

- Si $n \in \mathbb{N} \Rightarrow n < \omega \land n + \omega = \omega$

Esto es básicamente como el concepto de un número infinitamente grande.

Llamaremos a ω "*macro marcado*", ya que representa un conjunto de posibles marcas.

Aplicamos el macro marcado a los vértices que surgen de aplicar T_1 y T_2 al vértice de μ_1, obteniendo un grafo finito que representa la P_N en su totalidad:

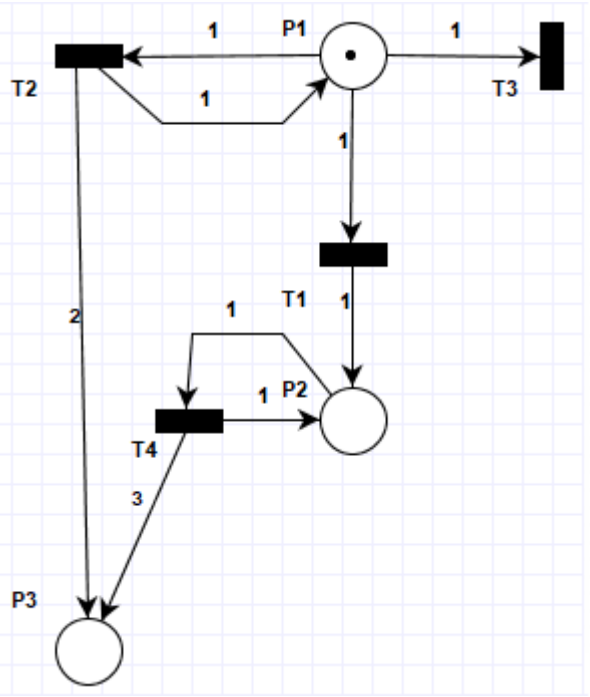


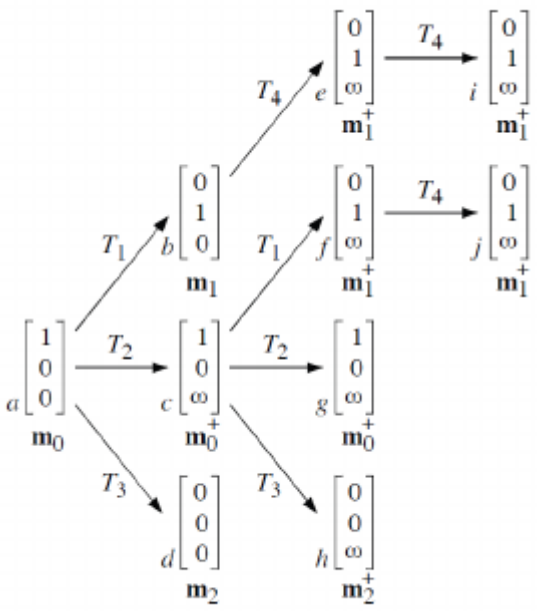
Este concepto de macro marcado no necesariamente tiene que ser aplicado para casos en los que podemos tener infinitos tokens en una plaza. También podemos aplicarlo en los casos en los que alguna plaza puede tener varios tokens pero no necesitamos la cantidad precisa de los mismos.

Construcción de un árbol raíz de cobertura

- Cada vértice corresponde a una marca (o macro marca).
- A partir de μ_0 se indican todas las transiciones sensibilizadas y las marcas sucesivas correspondientes. Si cualquiera de estas marcas es $\mu_m \geq \mu_0 \Rightarrow \omega$ se coloca para cada uno de los componentes de μ_m mayor que los componentes correspondientes de μ_0.
- Para cada nuevo vértice μ_i:
  - Si $\exists\ \mu_j = \mu_i$ en el camino de μ_0 a μ_i (μ_i no incluido) $\Rightarrow$ μ_i no tiene sucesor.
  - Si $\nexists\ \mu_j = \mu_i$ en el camino de μ_0 a μ_i (μ_i no incluido) $\Rightarrow$ el árbol se amplía sumando todos los sucesores de μ_i.
  - Si $\exists\ \mu_k \gneqq \mu_j$ en el camino de μ_0 a $\mu_k \Rightarrow \omega$ se coloca para cada uno de los componentes mayores que los componentes de μ_j.

Ejemplo:





Consideraciones de los árboles raíces de cobertura

Un árbol de raíz de cobertura siempre tiene un número finito de vértices. Sin embargo, para una P_N sin límites, el problema de accesibilidad no se puede resolver debido a la capacidad de cobertura del árbol raíz, ya que se pierde información al usar el símbolo ω.

Podemos tomar como ejemplo la P_N anterior para demostrar esto.

Si tomamos la secuencia de firings $S = T_2T_17T_4$, terminaremos con una marca $\mu_9 = (0, 1, 23)$. y esto no se vería reflejado en el árbol raíz de cobertura con el uso de ω en lugar del 23 de la forma $(0, 1, \omega)$.

Si quisiéramos verificar si la marca $(1, 0, 23)$ es alcanzable desde μ_0, no podríamos con el árbol raíz de cobertura.

Sabemos que no podríamos llegar; es decir, si $m(P_1) = 1 \Rightarrow$ no hubo firing de T_1 ni de $T_4 \therefore m(P_3) = 2k$ con $k \in \mathbb{N}$, pero esta información no podemos obtenerla del árbol raíz de cobertura.

De manera similar, no podemos resolver el problema de liveness de una P_N mediante el árbol raíz de cobertura.

Como detalle, agregamos el comentario de que el árbol raíz de cobertura para una P_N limitada contiene todas las marcas alcanzables posibles.

Álgebra lineal aplicada a redes de Petri

Definiciones

Red de Petri ordinaria no marcada

Una P_N ordinara no marcada es una cuádrupla de la forma: Q = (P, T, Pre, Post) donde:

- $P = \{P_i \mid i \in [1, n] \land i, n \in \mathbb{N}\}$: Conjunto de plazas.
- $T = \{T_i \mid i \in [1, n] \land i, n \in \mathbb{N}\}$: Conjunto de transiciones.
  - $P \cap T = \emptyset$: Los conjuntos P y T son inconexos.
- Pre: $P \times T \to \{0, 1\}$: Incidencia de entrada.
- Post: $T \times P \to \{0, 1\}$: Incidencia de salida.

Decimos que $Pre(P_i, T_j)$ es el peso del arco $P_i \to T_j$ y $Post(T_j, P_i)$ es el peso del arco $T_j \to P_i$.

Red de Petri generalizada no marcada

Una P_N generalizada no marcada se define como una P_N normal sin marcar, excepto que posee los siguientes elementos:

- Pre: $P \times T \to \mathbb{N}$
- Post: $T \times P \to \mathbb{N}$

Usaremos la siguiente notación de aquí en adelante:

- ${}^\circ T_j = \{P_i \in P \mid Pre(P_i, T_j) > 0\}$: Conjunto de plazas de entrada a T_j.
- $T_j^\circ = \{P_i \in P \mid Post(P_i, T_j) > 0\}$: Conjunto de plazas de salida a T_j.

Red de Petri marcada

Una P_N marcada es un par ordenado de la forma: R = (Q, μ_0) donde:

- Q: P_N no marcada.
- μ_0: Marca inicial.

Transición sensibilizada

Decimos que una transición T_j está sensibilizada para una marca $\mu_K \Leftrightarrow m_K(P_i) \geq Pre(P_i, T_j)$.

Matriz de incidencia de entrada

$$W^- = [w_{ij}^-] = Pre(P_i, T_j)$$

Esta es la matriz de incidencia de entrada a las transiciones, por lo que saca tokens de las plazas.

Matriz de incidencia de salida

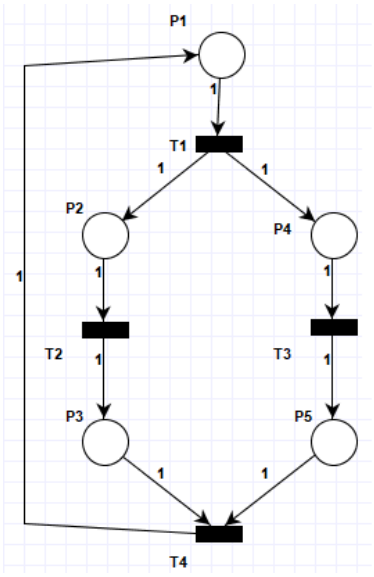
$$W^+ = [w_{ij}^+] = Post(T_j, P_i)$$

Esta es la matriz de incidencia de salida de las transiciones, por lo que agrega tokens a las plazas.

Ejemplo de matrices de incidencia de entrada y salida

A continuación se muestra un ejemplo de las matrices de incidencia de entrada y salida para una P_N con el fin de lograr mayor claridad.

Estas matrices representan cuántos tokens agrega/quita una transición a/desde una plaza. El número a colocar depende del peso del arco que une la transición con la plaza (o viceversa):



$$\mathbf{W}^- = \begin{array}{c} \begin{matrix} T_1 & T_2 & T_3 & T_4 \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{matrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \end{matrix} \quad \text{y} \quad \mathbf{W}^+ = \begin{array}{c} \begin{matrix} T_1 & T_2 & T_3 & T_4 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \end{matrix}$$

<u>Observación:</u> Si una P_N es pura, su matriz de incidencia permite que se construya la P_N no marcada.

Matriz de flujo

La definición de la matriz de flujo (o matriz de incidencia) es meramente matemática:

$$W = W^+ - W^-$$

Ecuación de estado de una red de Petri

Las P_N se utilizan para representar gráficamente el comportamiento dinámico de un sistema; por lo tanto, las características gráficas de las P_N pueden explotarse para inspeccionar la dinámica del sistema que representan. Este enfoque es adecuado para sistemas pequeños, la metodología gráfica no es eficiente cuando los sistemas son grandes y complejos. Con esta ecuación es posible obtener el siguiente estado del sistema. Esta es una manera más simple que la metodología gráfica para analizar la evolución de los sistemas.

La ecuación de estado de una P_N con n plazas y m transiciones, con arcos con peso mayor o igual a uno y marca inicial μ_0 es:

$$\mu_{j+1} = \mu_j + I\sigma$$

Siendo:

- I : Matriz de incidencia, con dimensión $n \times m$.
- σ : Vector de firing, con dimensión $m \times 1$. El número de componente de este vector corresponde al número de firings de la transición T_j en la secuencia S disparada.
- μ_j : Vector de marcado actual, con dimensión $n \times 1$.
- μ_{j+1} : Vector de marcado del estado siguiente, con dimensión $n \times 1$.

Los elementos i_{ij} de la matriz I se obtienen de la forma:

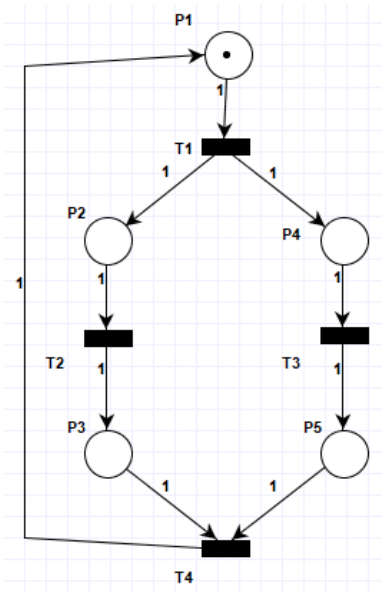
$$i_{ij} = w(t_i, p_j) - w(p_j, t_i)$$

Donde w son los pesos de los arcos (valores enteros con signo):

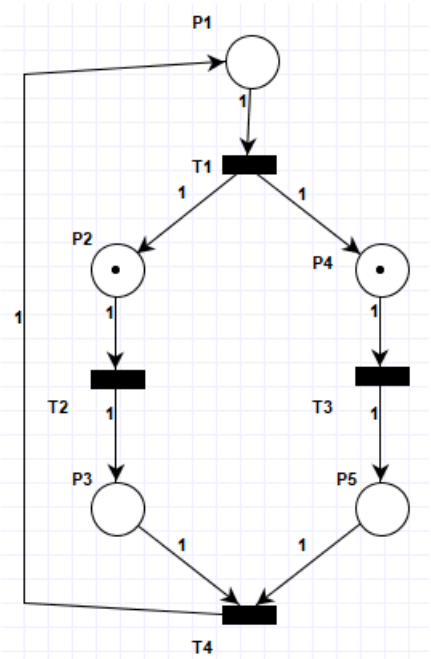
- $w(p_j, t_i)$: Arco de la plaza i a la transición j.
- $w(t_i, p_j)$: Arco de la transición j a la plaza i.

En pocas palabras, cuando se dispara una transición sensibilizada, se puede calcular el siguiente estado usando la ecuación de estado descrita aquí.

Consideremos la siguiente P_N:



Aquí comenzamos con la marca inicial $\mu_0 = (1, 0, 0, 0, 0)$. Luego de hacer el firing de T_1 obtenemos:



En este punto tenemos la marca $\mu_1 = (0, 1, 0, 1, 0)$. Si desde este punto efectuamos las secuencias de transiciones $S_1 = T_2T_3T_4$ y $S_2 = T_1T_3$, podemos juntarlas en una misma transición $S_3 = S_1S_2 = T_2T_3T_4T_1T_3$.

Si analizamos los vectores de firing de transiciones de cada secuencia (cuántas veces se disparan las transiciones en cada secuencia), tenemos:

$$\mathbf{s}_1 = (0,\ 1,\ 1,\ 1,\ 0)$$
$$\mathbf{s}_2 = (1,\ 0,\ 1,\ 0,\ 0)$$

y armamos nuestro vector σ para la ecuación de estado.

Este vector, para este caso, será:

$$\sigma = \mathbf{s}_1 + \mathbf{s}_2 = (1,\ 1,\ 2,\ 1,\ 0)$$

ya que las transiciones T_1, T_2 y T_4 se disparan una vez, y T_3 se dispara dos veces. Este vector es el que usaremos en la ecuación de estados.

Definiciones

Ahora que tenemos el concepto de ecuación de estado, podemos definir nuevamente conceptos anteriores de una manera más formal.

Transición sensibilizada

Una transición está sensibilizada si todas las plazas de entrada a la transición tienen una marca igual o mayor al peso del arco que une cada plaza con la transición. Esto se expresa como:

$$m(p_i) \geq w(p_j,\ t_i)\ \forall\ p_j \in I(t_i)$$

Las transiciones sensibilizadas se expresan como un vector binario E de dimensión $m \times 1$. Cada componente del vector indica con un 1 la transición sensibilizada y un 0 una transición que no lo está:

$$E = (\text{sign}(S^i) \mid i \in [0,\ n\text{-}1] \land i,\ n \in \mathbb{N},\ n \neq 0)$$

Donde la relación $\text{sign}(S^i)$ de cada vector S^i es:

- $S^i = \mu_j + (I^i)^T$: Vector con el estado que se obtendría de disparar la i-ésima transición una vez.
- $I^i \mid i \in [0,\ n\text{-}1] \land n \in \mathbb{N},\ n \neq 0$: Columnas de la matriz I.
- $\text{sign}(S^i)$: Símbolo binario, siendo 0 si alguna de las componentes de S^i es negativa; de otra forma es 1.

Firing de una transición

Se puede efectuar el firing de una transición sólo si la misma está sensibilizada. Cuando se dispara, se calcula el nuevo estado μ_{j+1} y el nuevo vector de transiciones sensibilizadas E.

Matriz de incidencia

De la semántica de firing de una P_N se interpreta la matriz de incidencia como la evaluación conjuntiva entre las columnas (transiciones) y las restricciones que imponen las filas (plazas). Es decir, en una matriz de incidencia de dimensión $m \times n$ se evalúan n combinaciones de m variables lógicas, como se expresa en la siguiente ecuación:

$$e_i = \left(\bigwedge_{h=0}^{m-1} m(P_h) \geq i_{hi} \right) \forall\, i \in [0,\, n\text{–}1] \;\; \wedge\, i,\, n \in \mathbb{N},\, n \neq 0$$

Donde:

- i_{hi} : Elementos de la matriz I.
- $m(P_h)$: Marca de la plaza h.
- e_i : i-ésimo elemento del vector E.

Observaciones

- Decimos que el vector σ es un vector *característico posible* si existe al menos una secuencia de firings posibles que lo involucre desde algún marcado de la red.
  - No todos los vectores cuyos componentes son $\sigma_i \in \mathbb{N}$ son posibles. Por ejemplo, si volvemos al ejemplo de la P_N mostrada en la página 95, el vector $\sigma = (0, 1, 0, 1, 0)$ no es posible si partimos desde μ_1, sin embargo sí es posible desde μ_0 si realizamos el firing de T_1.
- Varias secuencias de firings pueden corresponder a un vector característico posible. Por ejemplo, volviendo a la P_N de la página 95, las siguientes secuencias (partiendo de μ_1) llevan al mismo vector característico posible $\sigma = (0, 1, 1, 0)$:

$$S_1 = T_2 T_3$$
$$S_2 = T_3 T_2$$

De acuerdo con la ecuación de estado, dos secuencias de firing con el mismo σ dan como resultado la misma marca (el orden de firing de las transiciones en la secuencia no tiene importancia, la marca alcanzada será la misma). Sin embargo, lo recíproco no tiene validez; es decir, dos secuencias que dan como resultado la misma marca no necesariamente tienen el mismo σ.

Invariante de plaza

Consideremos un vector de ponderación de la forma:

$$X_P = (q_i \mid i \in [1, n] \land i \in \mathbb{N}, n \in \mathbb{Z}^+)$$

siendo q_i el peso asociado a la plaza P_i.
Sea, también, P(x) el conjunto de plazas cuyo peso no es 0. Podemos ver claramente que:

$$P(x) \subseteq P$$

Para ver esto más concretamente, consideremos la P_N de la página 95.
El vector $x_1 = (1, 1, 1, 0, 0)$ corresponde a una ponderación de valor para P_1, P_2, P_3, P_4 y P_5. Como en este caso sólo P_1, P_2 y P_3 están ponderados con un valor mayor a 0, tenemos:

$$P(x_1) = \{P_1, P_2, P_3\}.$$

Definimos ahora la siguiente propiedad:

El conjunto P de plazas es un componente conservador $\Leftrightarrow \exists\ X_P \mid P(x) = P \land {X_P}^T W = 0$, siendo W la matriz de incidencia.

Si analizamos este concepto haciendo uso de la ecuación de estado, podemos ver lo siguiente:

Sabemos que:		$\mu_K = \mu_0 + W\sigma$
Multiplicamos por $X_P{}^T$:		${X_P}^T\mu_K = {X_P}^T(\mu_0 + W\sigma)$
		${X_P}^T\mu_K = {X_P}^T\mu_0 + {X_P}^T W\sigma$
Pero si $X_P{}^T W = 0$:	$\Rightarrow$	${X_P}^T\mu_K = {X_P}^T\mu_0$
Simplificamos $X_P{}^T$:		$\therefore \mu_K = \mu_0$

Vemos aquí que el vector de marcado de la P_N inicial y el vector de marcado del estado K son iguales; es decir, todas las plazas tienen un peso distinto de 0.

El conjunto de plazas con peso $X_i > 0$ se llama *soporte de invariante de plaza*, y se denota de la forma: $\|X\|$.
Se dice que un invariante de plaza X_1 es mínimo $\Leftrightarrow \nexists\ X_2 \mid X_2$ es invariante de plaza $\land\ X_1 \gneqq X_2$.
Es decir, el conjunto de variantes mínimas de invariantes de plaza con soporte mínimo es una base a partir de la cual se pueden obtener otros invariantes de plaza.

Propiedades

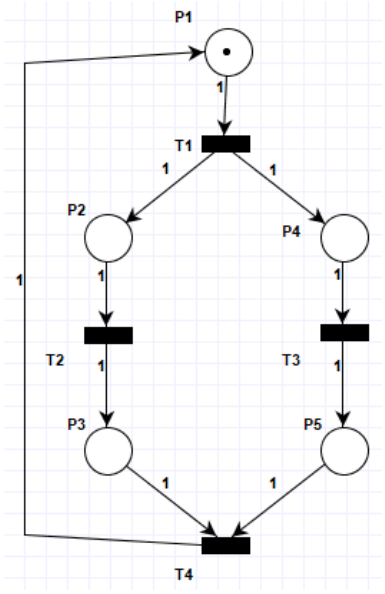
Propiedad #1

Sea $P(x) = \{P_i \mid i \in [1, r] \land i, r \in \mathbb{N}, r > 0\}$ un componente conservador, y sea
$x = (q_i \mid i \in [1, r] \land i, r \in \mathbb{N}, r > 0)$ el vector de ponderación correspondiente.
Todos los lugares $P_i \in P(x)$ están acotados, y se verifica que:

$$m(P_i) \leq \frac{x^T \mu_0}{q_i}$$

En pocas palabras, los invariantes de plaza son un sistema de ecuaciones lineales que se cumplen en la P_N para cualquier estado en la que ésta se encuentre.

Veamos un ejemplo con la P_N usada hasta el momento:



En esta P_N podemos ver que $P(x_1) = \{P_1, P_2, P_3\}$ y $P(x_2) = \{P_1, P_4, P_5\}$ son componentes conservativos, donde el sistema de ecuaciones de invariantes de plaza es:

$$\left\{ \begin{array}{c} \sum_{i=1}^{3} m(P_i) = 1 \\ \sum_{i=1}^{5} m(P_i) - \sum_{i=2}^{3} m(P_i) = 1 \end{array} \right\}$$

En este caso, $x_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \therefore x_1^T = [1 \;\; 1 \;\; 1 \;\; 0 \;\; 0]$

Es decir, sin importar en qué estado de la P_N nos encontremos, la suma de las marcas de las plazas P_1, P_2 y P_3 siempre será 1 (igual que con P_1, P_4 y P_5). Esto es fácil de verificar ejecutando distintas secuencias de firings de transiciones y haciendo la suma.

Para hallar esta/s ecuación/es que describen los invariantes de plaza de una P_N tenemos que resolver la ecuación:

$$x^T W = 0$$

Si lo expresamos como matrices y teniendo ya calculada W, debemos resolver la ecuación:

$$\mathbf{x}_1^T \cdot \mathbf{W} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 & +1 \\ +1 & -1 & 0 & 0 \\ 0 & +1 & 0 & -1 \\ +1 & 0 & -1 & 0 \\ 0 & 0 & +1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

De la solución de este sistema resultan las ecuaciones establecidas anteriormente como invariantes de plaza para esta P_N.

Propiedad #2

Sean x_1, x_2 dos invariantes de plaza. Entonces:

- Si $\alpha_1, \alpha_2 \in \mathbb{N} \Rightarrow \sum_{i=1}^{2} \alpha_i x_i$ es también un invariante de plaza.
- Si $\alpha_1, \alpha_2 \in \mathbb{N} \Rightarrow P(\sum_{i=1}^{2} \alpha_i x_i) = \bigcup_{i=1}^{2} P(x_i)$
- Si $\forall\, x_i \in x_1 - x_2 \mid x_i$ es componente del vector resultante de la resta : $x_i \in \mathbb{N} \Rightarrow x_1 - x_2$ es un invariante de plaza.

Podemos enunciar el primer inciso de esta propiedad como un teorema:

Teorema #1

Cualquier combinación lineal de invariantes de plaza es un invariante de plaza; y todo invariante de plaza es una combinación lineal no negativa de invariantes de plaza mínimos.

Invariante de transición

Definición

Un vector entero de conteo $Y \geq 0$ de dimensión $m = |T|$ es un invariante de transición $\Leftrightarrow CY = 0$ (recordemos que C es la matriz de adyacencia).

El conjunto de transiciones con $Y_i > 0$ se denomina *soporte de invariante de transición* y se denota de la forma: $\|Y\|$.

Se dice que un vector Y_i invariante de transición es mínimo $\Leftrightarrow \nexists\ Y_j \mid Y_j \neq Y_i \wedge Y_j \leq Y_i$.

Los invariantes de transición tienen propiedades similares a los invariantes de plaza.

Teoremas

Teorema #1

Dada una secuencia de transiciones de μ_n en μ_m con su vector de conteo $Y \Rightarrow \mu_n = \mu_m \Leftrightarrow Y$ es un invariante de transición.

Es decir, si hacemos el firing de una secuencia de transiciones partiendo desde μ_n y llegamos de nuevo a μ_n, entonces eso se considera un invariante de transición.

Teorema #2

Cualquier combinación lineal de invariantes de transición es otro invariante de transición; y todo invariante de transición es una combinación lineal no negativa de invariantes de transición mínimos.

En general, un invariante de transición mínimo corresponde a un proceso que puede repetirse para siempre.

Propiedades

Un vector Y que es solución de la ecuación $WY = 0$ se conoce como un invariante de transición. El vector de ponderación que consideramos debe ser un vector característico asociado con una secuencia de firings S (este vector corresponde a los elementos de transición distintos de 0, es decir, que se han disparado al menos una vez).

Si $\varsigma = Y$ es un invariante de transición, entonces el conjunto de transiciones que aparecen en S son el soporte del invariante de transición, y lo denotamos de la forma: $T(\varsigma)$.

Sea D un conjunto de transiciones:

D es un componente repetitivo estacionario $\Leftrightarrow \exists$ S secuencia de firing con vector característico $\varsigma \mid T(\varsigma) = D \land W\varsigma = 0 \therefore \varsigma$ es un invariante de transición.

Si $W\varsigma > 0 \Rightarrow$ D es un componente repetitivo.

Observaciones

- La existencia de un invariante de transición es una condición necesaria para que una P_N sea reversible, ya que podemos pensar un invariante de transición desde una marca μ_K como una secuencia de firings S tal que: $\mu_K \xrightarrow{S} \mu_K$.
- Si P_N es no marcada y $\exists$ invariante de transición $Y \mid Y > 0 \land WY = 0 \Rightarrow P_N$ se considera consistente.
- Si $\exists$ invariante de transición $Y \mid Y > 0 \land WY \geq 0 \Rightarrow P_N$ se considera estructuralmente repetitiva.
- Si S es una **secuencia repetitiva** de una marca $m_1 \in \mathfrak{M}(\mu_0)$, y S es una **secuencia de firing** de una marca $m_2 \in \mathfrak{M}(\mu_0) \Rightarrow$ S es una secuencia repetitiva de m_2.

Redes de Petri temporizadas

Como sabemos ya, las transiciones pueden tener un tiempo de espera para ser ejecutadas, o pueden ejecutarse instantáneamente.

Lo que ahora introduciremos es el tiempo que tarda en ejecutarse una transición.

Indeterminismo

Decimos que estamos en una situación de indeterminismo cuando no se especifica cuándo se hará el firing una transición sensibilizada; y, de hecho, tampoco está especificado si se hará el firing de esta transición. En este estado de indeterminismo tampoco se especifica cuál de las transiciones involucradas en un conflicto será la disparada.

Se ha tratado de reducir el indeterminismo de distintas maneras:

- Con P_N estocásticas, introduciendo una estimación estocástica del instante de firing de las transiciones.
- Con P_N temporizadas, introduciendo un tiempo para el firing de las transiciones.
- Con P_N "*fuzzy timing*", acotando los instantes en que la transición puede (o debe) ser disparada.

Interpretación de los tiempos establecidos

Podemos interpretar el tiempo de firing de una transición, según se mantenga o no la atomicidad del firing:

- **Cuando el parámetro temporal determina el tiempo que ha de transcurrir desde que una transición queda sensibilizada hasta que se dispara**: En este caso, la retirada y colocación de tokens se produce de forma atómica. Este tiempo establecido es conocido como "tiempo de sensibilización" o "*enabling time*".
- **Cuando el parámetro temporal puede determinar también el tiempo que debe transcurrir entre la retirada y colocación (instantáneas) de tokens de las plazas de entrada en las plazas de salida**: En este caso, el firing de la transición tiene tres fases:
  - Retiro de token de la plaza de entrada.
  - Firing de la transición.
  - Colocación de los tokens en la plaza de salida.

  Este firing no es atómico, sino que tiene una duración.

Red de Petri con delay

En una P_N con delay, el tiempo asociado al firing de una transición indica el tiempo que tarda en concretarse el firing de esta transición. Cada transición modela el final de una actividad (el tiempo representará la duración de la actividad).

Red de Petri con tiempo

En una P_N con tiempo (P_N^T) se especifica la duración del firing de una transición mediante la asociación de un intervalo que abarque todas las posibilidades de duración de la actividad. Esto presenta la ventaja de analizar un sistema en los peores escenarios (donde todas las actividades tarden su máximo tiempo posible), o el cálculo de tiempo de ejecución.

Formal y matemáticamente, definimos a una P_N^T como una 6-tupla de la forma:

$$P_N^T = (P, T, Pre, Post, \mu_0, C_{IS})$$

Donde:

- P: Conjunto finito no vacío. Conjunto de plazas de la red.
- T: Conjunto finito no vacío. Conjunto de transiciones de la red.
  - $P \cap T = \emptyset$
- $Pre : P \times T \to \mathbb{N}$: Función de incidencia previa descrita anteriormente.
- $Post : T \times P \to \mathbb{N}$: Función de incidencia posterior descrita anteriormente.
- μ_0: Marcado inicial de la red.
- $C_{IS} : T \times \mathbb{Q}^+ \to (\mathbb{Q}^+ \cup \infty)$: Correspondencia de intervalos estáticos, donde:

$$\mathbb{Q}^+ = \{x \mid x \in \mathbb{Q} \land x \geq 0\}^2$$

La función C_{IS} asocia a cada transición un par de la forma: $(C_{IS}(T_i), (\alpha_i, \beta_i))$, que define un intervalo de tiempo, por lo que debe verificarse:

- $\alpha_i \in [0,\infty)$
- $\beta_i \in [0,\infty)$
- $\alpha_i \leq \beta_i$ si $\beta_i \nrightarrow \infty$
- $\alpha_i < \beta_i$ si $\beta_i \longrightarrow \infty$

Este intervalo permite definir la regla de firing de una transición en una P_N^T.

Suponiendo que la transición T_i comienza a estar sensibilizada en el instante τ_i, y que continúa sensibilizada, el firing de la transición se producirá en un tiempo $\tau_j \in [(\tau_i+\alpha_i), (\tau_i+\beta_i)]$. En este firing, las marcas permanecen en las plazas de entrada durante el tiempo necesario, y una vez que se produce el firing, éste no consume tiempo; es instantáneo.

---

[2]: En la elección de $\mathbb{Q}^+$ radica la diferencia entre definiciones propuestas por conocedores del tema. Para algunos, C_{IS} debe definirse sobre $\mathbb{R}^+$. Nosotros optamos por $\mathbb{Q}^+$. No adentraremos en la justificación de esta decisión.

En estos firings, podemos definir:

- (α_i, β_i): Intervalo estático de firing.
- α_i: Instante de firing más cercano (EFT, *earliest firing time*).
- β_i: Instante de firing más lejano (LFT, *latest firing time*).

Podemos definir, además, para $P_N{}^T$ en general:

- *Intervalo puntual*: Transiciones con tiempo de sensibilización fijo. Estos intervalos de tiempo de sensibilización tienen la forma: $[\alpha_i, \alpha_i]$.
- *Intervalo sin restricción temporal*: Estas transiciones no tienen restricción con el tiempo de sensibilización. Estos intervalos de tiempo de sensibilización tienen la forma: $[0,\infty)$.

Semánticas

Semántica de tiempo fuerte

Una semántica de tiempo fuerte indica específicamente en qué momento y con qué condiciones se debe realizar el firing de la transición. Esta semántica es apropiada para el modelado de sistemas en tiempo real.

Semántica de tiempo débil

En una semántica de tiempo débil la transición no está obligada a ser disparada; pero, si lo hace, debe ser en el intervalo especificado. Esta semántica puede ser útil cuando se trata de modelar una acción que puede ocurrir únicamente durante un período de tiempo, pero que no necesariamente debe ocurrir.

Para decidir sobre las posibilidades de firing de una transición debemos tener en cuenta la sensibilización de la misma, y el tiempo que ha transcurrido desde que ha sido sensibilizada. Por este motivo, definimos a un estado de una P_N como un par ordenado de la forma:

$$S = (\mu, \mathbb{I})$$

donde:

- μ: Marcado de la P_N.
- $\mathbb{I}$: Conjunto de intervalos de firing. Es un vector de todos los posibles intervalos de firing de todas las transiciones sensibilizadas por el marcado μ.

– Anexo I –

Símbolos

Se provee a continuación una lista de los símbolos utilizados y su significado:

- $\neg$: Negación.
- $<$: Menor que.
- $>$: Mayor que.
- $\geq$: Mayor o igual.
- $\leq$: Menor o igual.
- $\gneqq$: Mayor estricto y estrictamente distinto.
- $=$: Igual.
- $\triangleq$: Igual por definición.
- $:=$: Se denota/define como.
- $\neq$: Distinto.
- $\cong$: Aproximado.
- $\#$: Cardinal.
- $\mathbb{N}$: Conjunto de los números naturales. Para este resumen, consideramos que el número '0' pertenece a este conjunto.
- $\mathbb{N}^+$: Conjunto de los números naturales excluyendo el cero.
  - $\mathbb{N}^+ = \{x \in \mathbb{N} \mid x > 0\} \therefore \mathbb{N}^+ = \mathbb{N} \setminus \{0\}$
- $\mathbb{Z}$: Conjunto de los números enteros.
- $\mathbb{Z}^+$: Conjunto de los números enteros positivos (excluyendo el cero).
  - $\mathbb{Z}^+ = \{x \in \mathbb{Z} \mid x > 0\} \therefore \mathbb{Z}^+ = \mathbb{N}^+ = \mathbb{N} \setminus \{0\}$
- $\mathbb{Q}$: Conjunto de los números irracionales.
- $\mathbb{Q}^+$: Conjunto de los números irracionales positivos (excluyendo el cero).
  - $\mathbb{Q}^+ = \{x \in \mathbb{Q} \mid x > 0\}$
- $\mathbb{R}$: Conjunto de los número reales.
- $\mathbb{R}^+$: Conjunto de los números reales positivos (excluyendo el cero).
  - $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$
- $\forall$: Para todo.
- $\exists$: Existe.
- $\nexists$: No existe.
- $\in$: Pertenece.

- $\notin$: No pertenece.
- {...} :
  - Conjunto de elementos.
  - Firing concurrente de transiciones.
- (...) :
  - Intervalo abierto (exclusive en ambos extremos).
  - Vector de dimensión $1 \times n$.
- [...] :
  - Intervalo cerrado (inclusive en ambos extremos).
  - Vector de dimensión $n \times 1$.
  - Matriz de dimensión $n \times m$.
  - Firing múltiple de transiciones.
- [...) : Intervalo semiabierto/semicerrado (inclusive en extremo izquierdo, exclusive en extremo derecho.
- (...] : Intervalo semiabierto/semicerrado (inclusive en extremo derecho, exclusive en extremo izquierdo.
- |...| : Valor absoluto.
- $\sum$: Sumatorio.
- ∞ : Infinito.
- $\emptyset$: Vacío.
- $\therefore$: Por lo tanto.
- $\because$: Porque.
- $\wedge$: Conjunción lógica.
- $\vee$: Disyunción lógica.
- $\cap$: Intersección.
- $\cup$: Unión.
- $\subseteq$: Está incluido en (o es igual a).
- $\not\subseteq$: No está incluido en (ni es igual a).
- $\subset$: Está incluido en.
- $\not\subset$: No está incluido en.
- $\setminus$: Excepto.

- $|$: Tal que (luego de $\exists$).
- $:$: Tal que (luego de $\forall$).
- $\Longrightarrow$: Entonces.
- $\Leftrightarrow$: Si y sólo si.
- $\longmapsto$: Relación.
- $\times$: Producto vectorial.
- ^{T}: Traspuesto/a.
- $\rightarrow$:
  - Tiende a.
  - Nos lleva a.

– Anexo II –

Ciclos y circuitos

Circuito elemental

Un circuito elemental es una ruta en un gráfico de tal manera que sólo el primer y último vértices son idénticos. El número de tokens de un circuito elemental en un gráfico de eventos es invariable, ya que no hay transiciones de entrada ni transiciones de salida.

Ciclo elemental

Es un ciclo donde no se repite ningún vértice (salvo el primero que coincide con el último). Lo notamos:

$$u_E = (u_1, \dots, u_n)$$

Propiedades de ciclos

Propiedad #1

Todo ciclo u_C es una suma de ciclos elementales sin aristas comunes.

Propiedad #2

Un ciclo es elemental si y sólo si es un ciclo minimal (es decir, que no se pueden deducir otros ciclos por supresión de aristas).

Pseudociclo

Es una cadena donde los extremos coinciden pero que una misma arista puede figurar más de una vez (también consecutivamente).

Ciclos y circuitos ciclo

Son una cadena simple, cuyos dos vértices extremos, inicial y terminal, coinciden (no tiene en cuenta la orientación). Si queremos describir la orientación en un ciclo designamos de la forma:

$$u^+ = \{u_i \mid u_i \text{ orientada en el sentido del ciclo}\}$$
$$u^- = \{u_i \mid u_i \text{ orientada en el sentido contrario al ciclo}\}$$

– Anexo III –

Comparación entre vectores

Sean Z_1, Z_2 dos vectores del mismo número de elementos (por ejemplo, dos marcas de la misma P_N).

Sabiendo que $Z_j(i)$ denota el i-ésimo componente de Z_j:

- $Z_1 \geq Z_2 \Leftrightarrow \forall\ Z_1(i) : Z_1(i) \geq Z_2(i)$
- $Z_1 \ngeq Z_2 \Leftrightarrow \forall\ Z_1(i) : Z_1(i) \geq Z_2(i) \wedge \exists\ Z_1(i) \mid Z_1(i) > Z_2(i)$ (al menos un elemento de Z_1 es mayor estricto)
- $Z_1 > Z_2 \Leftrightarrow \forall\ Z_1(i) : Z_1(i) > Z_2(i)$

– Fuentes –

- Material explicativo de todos los temas abarcados proporcionado por la cátedra de "*Programación Concurrente*".
- *"Ecuación de estado generalizada para redes de Petri no autónomas y con distintos tipos de arcos"* – Dr. Ing. Micolini, Orlando; Ing. Ventre, Luis Orlando; Ing. Eschoyez, Maximiliano Andrés; Geol. Cebollada, Marcelo y Verdaguer; Ing. Schild, Marcelo Ismael – Universidad Nacional de Córdoba – Octubre, 2016 – Paper de conferencia.
- *"Modelado e Implementación de Sistemas de Tiempo Real mediante Redes de Petri con Tiempo"* – García Izquierdo, Francisco José – Universidad de Zaragoza – Diciembre, 1999 – Tesis Doctoral.
- "*Java 7 Concurrency Cookbook*" – Fernández González, Javier.
- *"Java Threads, third edition"* – Oaks, Scott.
- *"Petri nets – Fundamental models – Verification and applications"* – Diaz, Michel.
- *"Sistemas operativos – Aspectos internos y principios de diseño – Quinta Edición"* – Stallings, William.
- Artículos de Wikipedia:
  - *https://es.wikipedia.org/wiki/Concurrencia_(inform%C3%A1tica)*
  - *https://es.wikipedia.org/wiki/Paralelismo_(inform%C3%A1tica)*
  - *https://es.wikipedia.org/wiki/Hilo_(inform%C3%A1tica)*
  - *https://es.wikipedia.org/wiki/Seguridad_en_hilos*
  - *https://es.wikipedia.org/wiki/Cierre_de_exclusi%C3%B3n_mutua*
  - *https://es.wikipedia.org/wiki/Sem%C3%A1foro_(inform%C3%A1tica)*
  - *https://es.wikipedia.org/wiki/Red_de_Petri*
  - *https://es.wikipedia.org/wiki/Monitor_(concurrencia)*
  - *https://es.wikipedia.org/wiki/Bloqueo_mutuo*
- Artículos de internet:
  - *https://www.monografias.com/trabajos51/sincro-comunicacion/sincro-comunicacion.shtml*
- Videos de los canales de YouTube: *"pildorasinformaticas"* y "*Empieza A Programar*".