IDENTIFICACIÓN DE CUELLOS DE BOTELLA EN CIRCUITOS ASÍNCRONOS

Autor: Luis Jesús Jorge Salcines
Director: Jorge Emilio Júlvez Bueno

Departamento de Informática e Ingeniería de Sistemas
Centro Politécnico Superior
Universidad de Zaragoza

Zaragoza, mayo de 2010

IDENTIFICACIÓN DE CUELLOS DE BOTELLA EN CIRCUITOS ASÍNCRONOS

RESUMEN

A diferencia de los circuitos digitales síncronos, cada uno de los componentes que integra un circuito asíncrono se caracteriza por no estar regido por un reloj central que marque su ritmo de funcionamiento. En este tipo de circuitos, el rendimiento del sistema suele estar determinado por una pequeña parte del mismo, denominada cuello de botella. El objetivo principal de este proyecto es desarrollar una serie de técnicas que identifiquen correctamente estos cuellos de botella.

En el presente proyecto se ha desarrollado un programa en Matlab capaz de analizar y simular circuitos asíncronos modelados con Redes de Petri. El programa identifica qué transiciones están habilitadas y selecciona, de forma aleatoria, sus respectivos tiempos de disparo de acuerdo a la función de probabilidad que tengan asociada. El simulador ejecuta la transición que tenga menos tiempo de disparo y recalcula el conjunto de transiciones habilitadas. El proceso se repite hasta que termine el experimento.

Simular una Red de Petri que modela un circuito asíncrono es computacionalmente muy costoso, dada la inmensa cantidad de lugares y transiciones con que puede llegar a contar. Para aligerar ese coste, se obtendrá primero el cuello de botella del circuito, y posteriormente se simulará.

Para ello se ha implementado otro programa en Matlab que optimiza Redes de Petri, quedándose sólo con aquellos lugares y transiciones que tienen especial relevancia en el comportamiento del circuito (es decir, el cuello de botella), ahorrando tiempo a la hora de realizar la simulación, sin perder exactitud en la obtención de resultados.

Este programa simplificará matricialmente todos los elementos que definen una Red de Petri, quedando lista para que el simulador realice el mismo experimento en una red de características similares pero mucho más reducida (optimizada).

Todos estos algoritmos han sido aplicados a 22 Redes de Petri ya modeladas para analizar sus resultados, estimar el comportamiento estadístico de cada red en cuanto a rendimiento (*throughput*) se refiere y determinar si los valores finales son coherentes.

Se presentan resultados de rendimiento mínimo, máximo y medio (en función de si se simula el peor de los casos, el mejor de los casos o simulaciones aleatorias respectivamente), diferencia del resultado de simulación respecto de la media entre los casos extremos (error de simulación), varianza de la simulación (para determinar si el comportamiento es estable con el tiempo) y por último, intervalos de confianza para un nivel de confianza dado, que dará una idea del rango en el que, estadísticamente, se encontrará el rendimiento medio de cada red.

Como complemento a lo anterior, también se ha implementado otro programa que analice la evolución en el tiempo del marcado del circuito. Todo el proyecto basa sus análisis y resultados en términos de rendimiento. En cambio, este complemento estudia el comportamiento lugar por lugar, mostrando dónde se acumula carga de proceso, dónde se libera con rapidez, y qué lugares funcionan bien, mal, o de forma alternativa.

Tabla de contenido

I. Introducción

I.1. Las Redes de Petri

En multitud de campos de la ingeniería, como la robótica, electrónica, fabricación, mecánica, etc., los procesos y sistemas pueden expresarse por medio de modelos. Las Redes de Petri proporcionan un método para construir esos modelos de forma que se pueda analizar su eficiencia y funcionamiento, facilitando además su implementación por hardware, software, etc.

Una Red de Petri no es más que un modelo matemático de un sistema, fácil de comprender y generar, que permite una amplia versatilidad para representar, con muy pocos elementos, multitud de comportamientos de sistemas. El método está relacionado con la matemática discreta y más concretamente, con la teoría de grafos. Por ello, el sistema es especialmente útil para modelar y analizar sistemas dinámicos de eventos discretos, como pueden ser componentes electrónicos, ordenadores, redes de comunicaciones, sistemas de fabricación, robots, etc., todos ellos caracterizados por ser, en general, sistemas síncronos.

Como se trata de un modelo matemático, es necesario aplicar una correspondencia entre los elementos del sistema a modelar y los elementos que ponen a nuestra disposición las Redes de Petri. Es decir, asociar al modelo matemático un significado físico. Es fundamental que exista una perfecta relación entre ambos modelos y tener en cuenta las restricciones que impone el sistema real, ya que las Redes de Petri también permiten realizar operaciones que no tendrían sentido físico y desvirtuarían el resultado final, o directamente no tendrían nada que ver con la realidad, como ocurre con cualquier sistema susceptible de ser modelado.

I.1.a. Estructura de una Red de Petri

Una Red de Petri consta de:

- La estructura de la red, estática
- El marcado de la red, dinámico

La **estructura de la red** es un conjunto de lugares y transiciones unidos entre sí por un diagrama de flujo (o grafo) con distintos pesos, que representan el comportamiento del sistema. Un lugar podría asimilarse a una variable de estado del sistema, por ejemplo, un componente donde un dato se procesa, o un material se transforma; o bien, el lugar donde esperan a ser procesados, indistintamente.

Ese proceso o transformación viene representado por las mencionadas transiciones, cada una con su tiempo de disparo o ejecución (es decir, el tiempo que tardan en procesar los datos de entrada). Partimos del supuesto de que una transición sólo comienza a ejecutarse cuando todos sus lugares previos conectados a ella tienen carga de trabajo suficiente, lo que también se denomina “habilitación de una transición”. Esa carga de trabajo necesaria viene definida por el peso de cada tramo de la red, que se explicará algo más adelante en este mismo capítulo.

El **marcado de la red** indica el estado actual del sistema, representado por la cantidad de *tokens* (puntos) que cada lugar de la red tiene en cada momento. El *token* representa a esa entidad procesada, o a la espera de ser procesada.

Y al hilo de lo anterior, en el siguiente gráfico podemos observar la diferencia que existe entre un dato o material que espera a ser procesado, de un dato que está siendo procesado:

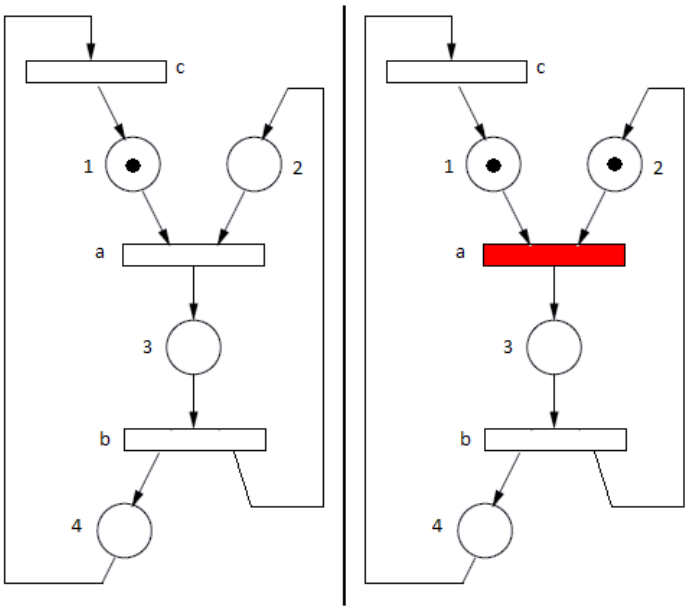


Figura 1: Habilitación de una transición

En el primero de ellos, observamos que la transición *"a"* no tiene todos sus lugares previos con carga de trabajo (marcados), por tanto, los datos que ya están situados en esos lugares esperan a ser procesados. En cambio, en el segundo caso están siendo procesados. Una vez que la transición termine (o como se suele decir, se "dispare") ambos *tokens* desaparecerán de los lugares 1 y 2 y la transición colocará un nuevo *token* en el lugar 3.

I.1.b. Representación matemática

Pese a que existen varias formas de representar una Red de Petri, se expone sólo la que este proyecto utiliza para el análisis y optimización de los modelos.

Una Red de Petri se define como la tupla siguiente [9]:

$$N = < P, T, Pre, Post >$$

Definición 1: Función matemática de una Red de Petri

Está compuesta por:

- Dos conjuntos *P* y *T*, finitos, no vacíos y disjuntos de lugares y transiciones respectivamente.
- Una matriz *Pre*, función de Pre-incidencia o *input*, de dimensiones *P* filas por *T* columnas y representada por la siguiente aplicación lineal:

$$Pre: P \times T \rightarrow \mathbb{N}$$

Definición 2: Función de Pre-incidencia

- Una matriz *Post*, función de Post-incidencia u *output*, también de dimensiones *P* filas por *T* columnas y representada por esta aplicación lineal:

$$Post: P \times T \rightarrow \mathbb{N}$$

Definición 3: Función de Post-incidencia

A su vez, una Red de Petri **marcada** se define como la tupla:

$$R =< N, M_0 >$$

Definición 4: Función matemática de una Red de Petri marcada

Donde *N* representa la Red de Petri definida anteriormente y M_0, un vector de *P* filas con el marcado inicial de la red (dónde están ubicados los *tokens* y cuántos hay en cada lugar, antes de arrancar el sistema), y representada por la siguiente aplicación lineal:

$$M_0: P \rightarrow \mathbb{N}$$

Definición 5: Función Marcado inicial de la red

Por último, se asocia a cada transición un tiempo de disparo denominado δ.

En este proyecto, se considera que una transición puede tener dos tiempos distintos de disparo, cada uno con una probabilidad de ocurrencia. De esta manera, se puede modelar un comportamiento que aparece con frecuencia en circuitos asíncronos, y que consiste en que un componente puede tener dos modos de funcionamiento, y por tanto, dos tiempos distintos de ejecución. De este modo, a cada transición *t* se le asocia un $\delta(t)=[(\delta_1, p_1), (\delta_2, p_2)]$, donde δ_1 será el tiempo de ejecución con probabilidad p_1, y δ_2 será el tiempo de ejecución con probabilidad p_2.

Así mismo, se considera que la máxima cantidad de *tokens* que pueden producir y consumir las transiciones en cada lugar es 1. Es decir, las funciones de Pre y Post-incidencia asocian a cada pareja (lugar, transición) un 0 ó un 1.

Esto no supone una pérdida de generalidad, puesto que una Red de Petri cuyas transiciones consuman (o produzcan) más de un *token* en algún lugar concreto del sistema, pueden ser fácilmente adaptadas o remodeladas por un conjunto de transiciones que consuman o produzcan sólo un *token*.

Se define un arco de entrada a una transición como [9]:

$$El\ arco\ que\ une\ un\ lugar\ p_i\ con\ una\ transición\ t_j\ si\ y\ sólo\ si\ Pre(p_i,t_j)\neq 0$$

Definición 6: Arco de entrada a la transición t_i

A su vez, se define un arco de salida de una transición como:

$$El\ arco\ que\ une\ una\ transición\ t_k\ con\ un\ lugar\ p_l\ si\ y\ sólo\ si\ Post(p_l,t_k)\neq 0$$

Definición 7: Arco de salida de la transición t_k

La matriz *Pre* indica qué lugares "vierten" su contenido en qué transiciones. Es decir, de qué lugares adquieren la carga de trabajo. Cada elemento de la matriz muestra el marcado necesario para proceder a disparar la transición, que como se ha dicho más arriba, en este proyecto será como máximo uno.

La matriz Post indica aquellos lugares donde la transición deposita el elemento procesado. De la misma manera, cada elemento de la matriz muestra el número de *tokens* que será depositado en los lugares posteriores, que en este proyecto será como máximo uno.

La forma más sencilla de interpretar la red de forma global es operando ambas matrices:

$$Post - Pre = C$$

Ecuación 1: Obtención de la matriz de incidencia C

Para este sencillo ejemplo:

$$Pre = \begin{pmatrix} & a & b & c \\ p1 & 1 & 0 & 0 \\ p2 & 1 & 0 & 0 \\ p3 & 0 & 1 & 0 \\ p4 & 0 & 0 & 1 \end{pmatrix} Post = \begin{pmatrix} & a & b & c \\ p1 & 0 & 0 & 1 \\ p2 & 0 & 1 & 1 \\ p3 & 1 & 0 & 0 \\ p4 & 0 & 1 & 0 \end{pmatrix}$$

$$C = \begin{pmatrix} & a & b & c \\ p1 & -1 & 0 & 1 \\ p2 & -1 & 1 & 1 \\ p3 & 1 & -1 & 0 \\ p4 & 0 & 1 & -1 \end{pmatrix}$$

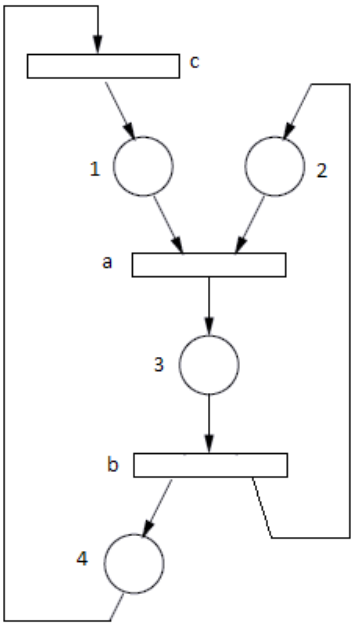


Figura 2: Estructura de la Red de Petri

Las matrices son de dimensiones 4x3 (cuatro lugares *p1..p4* y tres transiciones *a..c*). Las matrices anteriores se corresponden con la red de la derecha.

Analizando por ejemplo la columna correspondiente a la transición *"a"*, se deduce que:

1.- La transición *"a"* resta un *token* del lugar 1 y otro del lugar 2 (sendos elementos '-1' en la columna 'a', filas 'p1' y 'p2' de la matriz C).

2.- La transición *"a"* procesa esos *tokens* y coloca uno nuevo (elemento ya procesado) en el lugar 3 (valor '+1' en la columna 'a', fila 'p3' de la matriz C). Como la transición no hace nada con el lugar 4, su elemento correspondiente de la matriz se rellena con un cero.

Y así sucesivamente con cada columna de la matriz C.

Con todo esto, se tiene definida la ya mencionada estructura (estática) de la Red de Petri. Ahora sólo falta el marcado inicial, que en este proyecto se designa por el vector *M* (M_0 para el inicial y M_i para los siguientes).

Por ejemplo, para el caso que nos ocupa, definiremos un vector M_0 como sigue:

$$M_0 = \begin{matrix} p1 \\ p2 \\ p3 \\ p4 \end{matrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

Lo cual significa que tendremos un *token* en los lugares 1 y 2, con el resto de lugares vacíos, quedando un esquema inicial como éste:

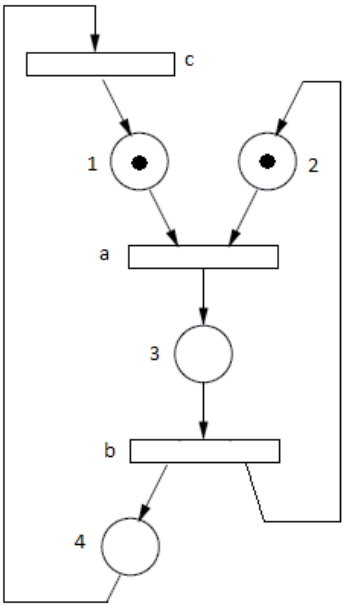


Figura 3: Estructura y marcado de la Red de Petri

En el apartado III.2 de esta memoria, figuran las operaciones matemáticas para simular el disparo de una transición y, en definitiva, cómo operar todas las matrices anteriores para hacer "evolucionar" el circuito.

I.1.c. Restricciones físicas al modelo matemático

Anteriormente se ha indicado que los sistemas físicos imponen ciertas restricciones a la hora de operar su modelo en Red de Petri. Por ejemplo, matemáticamente es posible disparar una transición sin que todos sus lugares previos tengan carga de trabajo (lo que provocaría, por ejemplo, que el marcado de alguno de esos lugares pasara de cero a un número negativo). Matemáticamente es perfectamente correcto, pero físicamente no tiene ningún sentido, puesto que cada transición representa un proceso. Si ese proceso no tiene carga de trabajo (una máquina no tiene una pieza o un componente electrónico no tiene un dato), no tiene sentido ejecutarlo.

I.2. Marco del proyecto

Los sistemas síncronos dominan prácticamente todas las áreas de diseño de la electrónica. Estos sistemas, por incluir un único reloj central, simplifican enormemente las tareas de diseño.

Sin embargo, para según qué tipo de especificaciones, un sistema síncrono puede resultar muy poco flexible a posibles cambios en los tiempos de procesado de datos, señales, comunicaciones, etc. Los circuitos convencionales no se diseñan (normalmente) para aceptar cambios parciales, sin que ello afecte al comportamiento del sistema completo. Así mismo, un factor tan importante como la temperatura del circuito integrado, ahora que cada vez se trabaja a mayores frecuencias de proceso, no es fácilmente controlable o regulable con circuitos síncronos, donde todos sus elementos trabajan a la misma frecuencia impuesta por el reloj central.

Por tanto, en circuitos electrónicos donde cada vez se integran más componentes diversos, cada uno con tiempos de computación y retrasos distintos, la eliminación del reloj central de todo el sistema es una alternativa a tener en cuenta. La eliminación del reloj central convierte el circuito en asíncrono y permite a cada módulo tener su propio ritmo de funcionamiento. La única sincronía que existiría sería aquella definida por cada par de componentes que realiza un intercambio de datos.

Los sistemas asíncronos suponen mayor complejidad de diseño. Por ejemplo, no se pueden producir *glitches* (también llamados picos espurios de señal) o estados transitorios que puedan generar fallos en la interpretación de la información, o al menos deben limitarse a aquellos intervalos de tiempo en los que la señal no es analizada.

A lo largo de los años, varios tipos de sistemas se han ido aproximando a la idea de los circuitos asíncronos, por ejemplo: emulación síncrona de circuitos asíncronos, circuitos sincronizados por *hand-shakes* o sistemas síncronos elásticos. En todos estos sistemas mencionados, se toleran cambios en los retrasos de cada componente, pero todo sigue estando sincronizado por un reloj central.

I.3. Objetivo y alcance del proyecto

En contraste con los circuitos síncronos, tal y como se acaba de indicar, los circuitos asíncronos carecen de un reloj global y sincronizan sus operaciones por medio de *hand-shakes* locales. De esta manera los circuitos asíncronos evitan el calentamiento del circuito a frecuencias elevadas y ofrecen una mayor flexibilidad al diseñador.

Normalmente, el rendimiento de un circuito asíncrono está limitado por unos pocos componentes que se denominan cuellos de botella. Identificar correctamente los cuellos de botella es fundamental a la hora de optimizar el rendimiento del circuito.

El objetivo de este proyecto consiste en identificar de manera eficiente cuellos de botella de un circuito asíncrono con retardos variables. Los cuellos de botella obtenidos serán utilizados para estimar el rendimiento de todo el circuito.

Para la consecución de estos objetivos, se han llevado a cabo las siguientes tareas:

- **Implementación en Matlab de un simulador de circuitos modelados con Redes de Petri**: Este simulador permite analizar cómo evoluciona un circuito en el tiempo, permitiendo así estimar su rendimiento medio y el rango de valores en que se encuentra, siguiendo una distribución estadística *t-Student* con intervalos de confianza. Puesto que la simulación de un circuito completo puede resultar inabordable en cuanto el número de lugares o la complejidad de la red se incrementan, es necesario implementar un algoritmo de optimización como el que sigue.
- **Desarrollo de un método basado en programación lineal (optimización) para la obtención de cuellos de botella**, permitiendo con ello reducir los circuitos al máximo, tratando en la medida de lo posible que la red simplificada conserve un comportamiento similar al del circuito completo. Este sistema se centra en aquellos lugares de la red que determinan el o los “caminos” más lentos de la red, que son a fin de cuentas los que determinan su rendimiento final.

- **Fusión de ambos métodos (optimización más simulación)** para que, una vez optimizado convenientemente un circuito modelado con Redes de Petri, se pueda evaluar con el simulador presentado más arriba.
- **Aplicación de los métodos anteriores a una serie de redes relativamente grandes** (superiores a los 1.000 lugares) para verificar la utilidad del sistema.
- **Extracción de conclusiones de los análisis.**
- **Diseño de otro sistema que simula un test de estrés al circuito que se quiere analizar**, como complemento a los anteriores aunque sin profundizar en los resultados, para estudiar la evolución con el tiempo de la carga de trabajo en cada lugar de la red.

Como punto de partida del proyecto, el tutor puso a mi disposición una serie de ficheros con esquemas de circuitos, así como un programa implementado en Matlab que permite leer dichos archivos y convertirlos en las matrices características que definen una Red de Petri (matrices *Pre*, *Post*, marcado inicial y tiempos de disparo de cada transición). Con ello, ya es posible realizar los análisis y operaciones matemáticas pertinentes para simular y optimizar las distintas redes.

II. Lectura de esquemas de circuitos

Tal y como se ha indicado en la introducción, esta aplicación, junto con una serie de circuitos ya esquematizados siguiendo un formato concreto, representan uno de los puntos de partida del proyecto. La aplicación se llama “readgr.m” y su código fuente se puede encontrar en el anexo XI.1.a de esta memoria.

Como parámetro de entrada se introduce el fichero que contiene el esquema del circuito y el programa proporcionará las cuatro matrices que definen la Red de Petri de dicho esquema, según se desarrolla en los siguientes subapartados:

II.1. Formato de los esquemas de circuitos

Para que este módulo del proyecto pueda transformar correctamente el esquema de un circuito en sus respectivas matrices que definen una Red de Petri, el archivo (de texto) ha de seguir el siguiente formato:

Una primera sección, en la que se define la situación de cada lugar de la red. Cada línea representa un lugar diferente con tres datos separados entre sí por una tabulación: número de transición de entrada (es decir, la transición que añade un *token* al lugar), número de transición de salida (es decir, aquella que quita un *token*) y marcado inicial de dicho lugar (número de *tokens* con los que parte el lugar al arrancar el sistema).

Para separar la primera de la segunda sección, se añade una línea intermedia con un “-1” (sin comillas).

En la segunda sección, se definen los tiempos (deltas) de disparo de cada transición así como la probabilidad, en tanto por uno, de que aparezca uno u otro tiempo. Cada transición va en una línea aparte. En total, serán cuatro datos diferentes por fila, separados también por una tabulación, a saber:

- Probabilidad de que dicha transición tarde un tiempo "x" en dispararse o ejecutarse.
- Tiempo "x" de disparo.
- Probabilidad de que tarde un tiempo "y".
- Tiempo "y".

Es importante que no falte ningún dato en el fichero. Si alguna transición tuviese siempre el mismo delta de disparo, habrá que indicar una probabilidad 1, seguido de su tiempo de disparo y añadir otros dos ceros para completar la fila (probabilidad cero de que aparezca un delta cero).

Como ejemplo ilustrativo, tomaremos la Figura 3: Estructura y marcado de la Red de Petri, para ver qué aspecto tendría su fichero en formato texto. A su vez, vamos a suponer que la transición *"a"* tiene un δ de disparo de 2 unidades de tiempo con probabilidad del 75%, y 4 unidades de tiempo con probabilidad del 25%. Para el resto de transiciones, asumiremos que todas tienen un único δ de disparo de 3 unidades de tiempo.

Los datos del fichero son todos numéricos, así que las transiciones pasarán de llamarse *"a, b, c"* a *"1, 2, 3"*:

	Transición de entrada	***Transición de salida***	***Marcado inicial***	
Lugar 1	3	1	1	
Lugar 2	2	1	1	
Lugar 3	1	2	0	
Lugar 4	2	3	0	
Separador	-1			
	Probabilidad de δ_1	***δ_1***	***Probabilidad de δ_2***	***δ_2***
Transición 1 (a)	0,75	2	0,25	4
Transición 2 (b)	1	3	0	0
Transición 3 (c)	1	3	0	0

Tabla 1: Formato del archivo de texto que contiene la Red de Petri

Por tanto, el fichero adquirirá este aspecto:

3	1	1	
2	1	1	
1	2	0	
2	3	0	
-1			
0,75	2	0,25	4
1	3	0	0
1	3	0	0

Tabla 2: Aspecto del archivo de texto que contiene la Red de Petri

II.2. Datos de salida

El programa devuelve cuatro matrices por este orden: *Pre, Post, M_i y Delta*.

En las dos primeras, cada fila de la matriz representa un lugar de la red y cada columna una transición. En el apartado I.1, Redes de Petri, se puede ver una explicación algo más detallada del funcionamiento de estas matrices.

La tercera tabla, M_i, de una única columna, representa el marcado inicial de cada lugar.

La cuarta matriz, *Delta*, se organiza en cuatro columnas: la primera es el "Dato 1" (explicado en el apartado anterior), la segunda es el "Dato 2" y así sucesivamente. Cada línea es una transición.

III. Simulación de circuitos

Este procedimiento se ha diseñado de dos formas diferentes: una, que permita simular un circuito directamente desde el archivo donde se encuentra su esquema, o bien dos, a partir de las cuatro matrices de una Red de Petri (que resultará útil cuando se vaya a simular una red optimizada). Cada una de las formas se inicia con un código diferente, denominados "simularArchivo.m" y "simular.m" respectivamente. La única diferencia entre ambos es el uso de la función "readgr.m".

III.1. Datos de entrada y arranque de la simulación

Para el caso de "simular.m", la simulación se inicia introduciendo como valores de entrada las cuatro matrices de la Red de Petri (en el mismo orden en que las facilita el lector de esquemas presentado en el capítulo anterior), el número de veces que ha de iniciarse la simulación aleatoria desde el principio, el tiempo de ejecución de cada simulación y el nivel de confianza en tanto por uno (cuya fórmula se puede encontrar más adelante en este capítulo).

Para proceder a simular el comportamiento del circuito, el programa comprueba transición por transición cuáles están habilitadas (tienen todos sus lugares previos marcados) y acto seguido, ejecuta aquélla con menor tiempo de disparo.

En el caso de aquellas transiciones con distintas posibilidades de tiempos de disparo (cuya explicación puede encontrarse en la introducción), el programa genera un valor aleatorio para determinar qué tiempo escoger, en función de su probabilidad de ocurrencia.

A partir de la segunda ejecución, hay que tener en cuenta aquellas transiciones que ya estuvieran habilitadas con anterioridad, ya que también han estado ejecutándose al mismo tiempo que la transición disparada (aunque tarden más en terminar). Eso no quita para que haya que seguir evaluando la red entera para comprobar las nuevas transiciones que se hayan habilitado desde el último disparo.

III.2. Disparo de transiciones

Para facilitar la reutilización de código para otros propósitos, las operaciones para detectar transiciones habilitadas y el disparo de la más rápida se realizan mediante una llamada a otro subprograma, denominado "avanzarTransicion.m".

La subrutina creará un vector con los tiempos de disparo de todas las transiciones habilitadas, o bien, para el caso de aquellas que ya estuvieran ejecutándose con anterioridad, respetará los tiempos remanentes para su disparo.

Una vez determinada la siguiente transición a disparar, la operación matemática para modificar las matrices de la Red es muy rápida y sencilla:

$$M_i = M_{i-1} + C \cdot d$$

Ecuación 2: Operación para disparar una transición

Siendo *'M_i'*, una matriz columna con el nuevo marcado de la red tras la ejecución de la transición; *'M_{i-1}'*, el marcado anterior a la ejecución; *'C'*, matriz resultado de operar *'Post–Pre'*; y por último *'d'*, matriz columna, con tantas filas como transiciones tiene la red y todos los elementos nulos, salvo aquel cuya transición se desea disparar (que tendrá un '1').

En el caso de que hubiera algún fallo en la descripción o funcionamiento del circuito, como por ejemplo que ninguna transición estuviese habilitada, el programa lo notificará.

III.3. Eficiencia en el análisis

Tal y como se ha comentado con anterioridad, simular una red sin ninguna optimización previa puede resultar costosísimo en cuanto el número de lugares o la complejidad del circuito crece. Hay que tener en cuenta que, pese a tratarse de operaciones matriciales muy sencillas y con matrices en las que una gran mayoría de los elementos son nulos, es necesario realizar muchas ejecuciones de transiciones y varios arranques del sistema completo para poder obtener resultados fiables.

Además, cada vez que se dispara una transición, el programa debe evaluar el circuito completo para ver qué nuevas transiciones han entrado en juego. Acto seguido, debe asignar a todas ellas su tiempo de disparo correspondiente, o bien respetar el que tuvieran en el caso de las transiciones ya habilitadas y todavía no disparadas.

Por otro lado, se ha de realizar una operación matricial de tamaño considerable: *'C'* tiene de dimensiones: número de lugares por número de transiciones, lo cual, en cuanto la red tenga por ejemplo 150 lugares y 100 transiciones, dicha matriz adquirirá una dimensión de 15.000 elementos (aunque muchos de ellos sean cero).

Es decir, para el ejemplo sugerido, cada vez que se ejecuta una transición, el programa ha de revisar 100 transiciones y multiplicar matrices [150x100] · [100x1]. El tiempo de ejecución de estas tareas es del orden de milisegundos, pero se trata de decenas de simulaciones completas y aleatorias, de miles de disparos por simulación, para poder obtener resultados aceptables.

De ahí la necesidad de optimizar el esquema antes de simularlo.

III.4. Resultados de la simulación

El objetivo fundamental de las simulaciones es determinar el rendimiento global del circuito, también llamado *Throughput* o número de disparos medio por unidad de tiempo.

En el proceso de desarrollo de código, se incluyó un resultado adicional que guardaba el número de veces que se disparaba cada transición de la red en cada simulación aleatoria completa. A partir de ese resultado obtenido, se verificó el comportamiento de toda esta clase de Redes de Petri: **todas las transiciones se ejecutan el mismo número medio de veces**. Las leves variaciones observadas en los diferentes ensayos sólo están relacionadas con el momento en que se detenga cada simulación.

Y este hecho viene avalado por la propia definición y características de las Redes de Petri que este proyecto estudia, a saber:

- **Las redes están compuestas por ciclos lugar-transición**.
- **La suma de tokens en cada ciclo es constante.** En otras palabras, el marcado de la red (entendido como la suma de todos los *tokens* del esquema) es estacionario (que no constante): no existe acumulación ni destrucción de *tokens*, aunque haya momentos en que sí pueda existir variación, por ejemplo cuando una transición consume dos *tokens* y genera sólo uno. Dicha variación quedaría compensada en cuanto el ciclo se complete.

Así pues, es lógico que todas las transiciones se disparen un mismo número de veces. Y por tanto, a la hora de elaborar los resultados finales con el rendimiento del circuito, se toma el número de disparos de una sola transición.

Los datos de salida que proporciona este programa son:

- Un vector, con el *throughput* del circuito en cada una de las simulaciones aleatorias completas realizadas.
- Un vector con dos valores: el primero es el *throughput* medio del esquema y el segundo, su varianza. Este último dato es de gran utilidad, ya que da una idea de la exactitud del rendimiento del circuito y si la aleatoriedad de los tiempos de disparo provoca grandes diferencias en aquél. Es posible que exista algún elemento de la red (como por ejemplo, un multiplicador en un circuito electrónico) que ralentice in extremis su funcionamiento en según qué circunstancias (en función del número de bits que deba multiplicar). Esto nos permite saber si hay que hacer especial hincapié en alguna de las zonas de la red.
- Por último, un rango de valores del *throughput* del circuito (intervalo de confianza), para un nivel de confianza dado, que se introduce como variable de entrada al programa, según se ha explicado al comienzo de este capítulo. Para un nivel *"x"* (en tanto por uno), este resultado expresa que en el *x%* de las veces, el rendimiento del circuito se ubicará dentro de dicho rango de valores. Cuanto menor sea el rango, mejor.

La fórmula empleada para el cálculo del intervalo de confianza para un nivel de confianza dado es la siguiente:

$$IC = \left[\bar{X} - \left(t\left(1 - \frac{1-\alpha}{2}, n-1\right) \cdot \sqrt{\frac{\vartheta}{n}}\right), \bar{X} + \left(t\left(\frac{\alpha}{2}, n-1\right) \cdot \sqrt{\frac{\vartheta}{n}}\right)\right]$$

Ecuación 3: Intervalo de confianza

Siendo:

- $\bar{X}$, rendimiento medio de todas las simulaciones.
- $t\left(1 - \frac{1-\alpha}{2}, n-1\right)$, función de densidad de la distribución *t-Student* con parámetro $1 - \frac{1-\alpha}{2}$ y con $n-1$ grados de libertad.
- α, nivel de confianza especificado.

- n, número de iteraciones de la simulación.
- ϑ, varianza de la muestra.

En el apartado V. Análisis completo, se puede encontrar un algoritmo descriptivo, a modo de pseudocódigo, que explica el funcionamiento de la simulación.

IV. Optimización de circuitos

Ya se ha comentado anteriormente que la simulación de la estructura completa es inviable en muchos casos, por lo que es necesario recurrir a una optimización o simplificación del circuito.

Matlab ofrece la función *"linprog"* para resolver problemas de programación lineal. Esta función puede ser utilizada para obtener un ciclo de la red (cuello de botella), que contiene los lugares y transiciones que determinan el camino más lento de todo el sistema [8]. Es decir, descarta todos los lugares y transiciones "rápidos" que no están vinculados y se queda exclusivamente con un camino único y cerrado, sin bifurcaciones.

La solución del problema de programación lineal debe interpretarse correctamente, ya que dicho problema no tiene en cuenta los diferentes tiempos de disparo que puedan tener las transiciones. De hecho, antes de resolverlo, hay que indicar qué tiempo de disparo concreto se va a utilizar para cada transición, con lo cual, se pierde la aleatoriedad del circuito, representada por las probabilidades de ocurrencia de cada tiempo. Más aún, el resultado de la función objetivo (el *throughput* del sistema optimizado) será siempre el mismo, puesto que es un camino cerrado y siempre con los mismos tiempos de disparo.

Y dado que lo que se busca es eficiencia y rapidez de simulación, es evidente que no se puede realizar una optimización del sistema para cada combinación posible de tiempos de disparo. En primer lugar, porque cada resolución del problema de optimización no es inmediata y hay demasiados casos que optimizar, y en segundo lugar, porque tras múltiples pruebas de optimización en varios circuitos, se ha comprobado que los cuellos de botella obtenidos son iguales, o muy parecidos; y en cualquier caso, se generan pocos circuitos cerrados diferentes. Además, en el caso de que los cuellos de botella fueran muy dependientes del tiempo de disparo, cada combinación ofrecería un circuito cerrado distinto, que habría que unir al anterior circuito, con lo cual no se reduciría lo más mínimo el tamaño del esquema a simular.

Así pues, es necesario elegir los casos que son representativos del comportamiento del circuito. Si se unen todas las soluciones obtenidas en cada caso representativo, se obtiene un circuito de comportamiento similar al del sistema original, que además cumple con la necesidad de tener un tamaño reducido para ser simulable de manera eficiente.

IV.1. Casos representativos del comportamiento del circuito

Para determinar adecuadamente los casos representativos mencionados, se han realizado numerosas pruebas en varios circuitos.

Pasos seguidos:

Se parte del peor de los supuestos (es decir, que todas las transiciones tomen su tiempo máximo de disparo) obteniendo un cuello de botella o círculo cerrado para esa situación y se observa el *throughput* de dicha solución.

Volviendo al esquema completo del circuito, se toma la primera transición que tenga dos tiempos posibles de disparo y se cambia el tiempo máximo por el tiempo mínimo de ejecución. De nuevo, se optimiza, se obtiene el cuello de botella y se observa el *throughput* y los lugares de dicho cuello de botella.

De forma sucesiva, se vuelve al esquema completo, se deja el tiempo de las anteriores transiciones como están (su valor mínimo), se toma la siguiente transición y se cambia también el tiempo máximo por el mínimo.

Se presenta a continuación, en pseudocódigo, el algoritmo que sigue este método:

```
Seleccionar Deltas máximos para todas las transiciones;

Optimizar red;
Obtener y guardar throughput de optimización;
Obtener y guardar lista de lugares del cuello de botella;
Deshacer Optimizar red;

Para cada transición de la red hacer
{
 Si transición tiene dos tiempos de disparo hacer
 {
  Cambiar Delta máximo por Delta mínimo;

  Optimizar red;
  Obtener y guardar throughput de optimización;
  Obtener y guardar lista de lugares del cuello de botella;
  Deshacer Optimizar red;
 }
}

Representar evolución de throughput;
Representar evolución de listas de lugares;
```

De esta forma, se tendrá un vistazo rápido de la evolución del *throughput* y del cuello de botella. Se ha observado que, en general, el cuello de botella se modifica muy pocas veces, es decir, el camino cerrado se conserva durante varias modificaciones de los tiempos de disparo, cambiando sólo en determinados momentos. Así mismo, con el *throughput* se tiene un comportamiento similar: evoluciona muy lentamente (de forma casi inapreciable, con modificaciones en el quinto o sexto decimal) hasta que da un salto importante.

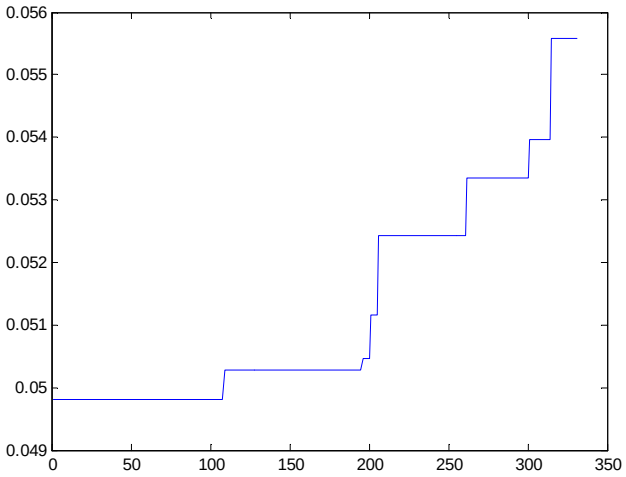


Figura 4: Evolución del *throughput* en los sucesivos cambios de los tiempos de disparo

En esta gráfica se aprecia como el *throughput* se modifica de forma considerable en muy pocas ocasiones.

Por tanto, se ha considerado que, para obtener un reflejo fiable del sistema real en el esquema simplificado, hay que tomar tres casos diferentes representativos, a saber:

- El cuello de botella formado en el caso más desfavorable, con los tiempos de disparo máximos de cada transición.
- El cuello de botella formado en el caso más favorable, con los tiempos de disparo mínimos de cada transición.
- El cuello de botella formado en el caso más posible, tomando los tiempos de disparo más probables de cada transición.

De esta forma, tenemos un circuito cerrado para cada situación extrema del comportamiento del circuito, y otra para el comportamiento esperable del mismo (los tiempos de disparo con más probabilidad de ocurrencia).

En el capítulo V. Análisis completo (optimización y simulación), se explicará cómo unir los circuitos cerrados obtenidos para poder simularlos posteriormente.

IV.2. Planteamiento del problema de programación lineal

La función “linprog” de Matlab mencionada anteriormente utiliza por defecto un método de optimización basado en una variante del algoritmo de predicción-corrección de Mehrotra (que es un método de optimización de tipo primal-dual). Matlab tiene implementados otros métodos diferentes, pero éste ha resultado útil, funcional y rápido en todas las pruebas realizadas a todos los circuitos, por lo que no se han estudiado otras posibilidades (aparte de que la solución de optimización es única para cada caso que se plantee y no varía en función del algoritmo utilizado).

El problema de programación lineal a resolver es el siguiente [8]:

$$\Gamma_j \geq \Gamma_j^{mín} = máx_y\left(y \cdot Pre \cdot D^j\right)$$
$$sujeto\ a: \qquad y \cdot C = 0, y \geq 0$$
$$y \cdot m_0 = 1$$

Ecuación 4: Problema de programación lineal a resolver

Siendo *'Γ'*, la función a optimizar (en la ecuación anterior, *'Γ'* es el inverso del *throughput*); *'y'*, las variables de las que depende la función objetivo, que en el caso que nos ocupa, son tantas como lugares tiene la red; *'D'*, los deltas de disparo elegidos para las transiciones; *'C'*, la matriz de incidencia (*'Post-Pre'*); y *'m_0'*, el marcado inicial de la red.

La función *"linprog"* de Matlab resuelve problemas de minimización, con lo cual basta con cambiar el signo de la función objetivo (maximizar una función es lo mismo que minimizar la misma función cambiada de signo).

En el código "optimizacion.m" (ver anexo XI.1.h) se puede ver el procedimiento que se ha seguido para facilitar todos los parámetros a la función de Matlab para resolver cada problema de programación lineal. En primer lugar, cambiar el signo a la función objetivo para minimizarla. En segundo lugar, representar en forma matricial las restricciones (sistema de ecuaciones *y·C*) que afectan al problema de optimización. Hay que tener en cuenta que Matlab exige que estas restricciones estén de la forma *A·y=b*, lo cual se consigue trasponiendo la matriz *C*. Por último se establecen los valores límite de las variables de las que depende la función objetivo (valores de *'y'* sólo positivos).

Por tanto el problema de optimización a resolver se convierte en:

$$-\Gamma_j \leq -\Gamma_j^{mín} = mín_y\left(-y \cdot Pre \cdot D^j\right)$$
$$sujeto\ a: \qquad m_0' \cdot y = 1$$
$$C' \cdot y = 0$$
$$y \geq 0$$

Ecuación 5: Problema real de optimización a resolver

El cuello de botella vendrá determinado por aquellos valores de *'y'* distintos de cero. Sin embargo, se ha observado que en el resultado de la optimización que devuelve Matlab, no hay ningún valor que dé exactamente cero, aunque sí son valores despreciables en comparación con otros, dato a tener en cuenta a la hora de indicarle al programa qué valores debe discriminar. Si se programa para que elimine aquellos y_i exactamente iguales a 0, no hará nada. Se ha tomado como valor cercano a cero, los inferiores a 10^{-8}.

También se ha observado (aunque no se ha podido determinar con exactitud el motivo) que las restricciones se deben suministrar en distinto orden del que aparecen en la Ecuación 5: Problema real de optimización a resolver. Primero ha de ir la ecuación de normalizado (*y·m_0=1*) seguida del resto. Se tuvo que cambiar el orden al comprobar que algunas de las soluciones que facilitaba el programa eran negativas, lo que tiraba por tierra las restricciones impuestas al problema.

Finalmente, el tiempo máximo entre dos ejecuciones de una transición es $-\Gamma_j^{mín}$, y el *throughput* será su inverso.

IV.3. Cálculo de cuellos de botella

Si lo que se desea es calcular exclusivamente los sucesivos cuellos de botella y la evolución del *throughput*, sin simulación posterior, se puede recurrir al programa "calculoCuellos.m", facilitándole como único parámetro de entrada, el archivo donde se encuentre el esquema del sistema.

Dicho programa calculará los circuitos optimizados para los tres casos representativos indicados anteriormente (tiempos máximos, tiempos mínimos y tiempos más probables). Posteriormente volverá al caso más desfavorable (todos los deltas máximos) e irá mostrando en sucesivas líneas de información, cómo evoluciona el *throughput* hasta que alcance el caso más favorable (todos los deltas mínimos).

Si lo que se quiere es obtener también una relación (ratio) de simplificación del circuito, es decir, a cuántos lugares ha quedado reducido respecto del total de lugares del esquema, se puede ejecutar el código "calculoCuellosConRatio.m".

Por el contrario, si no se quiere estudiar esta evolución (puesto que sólo se utiliza a modo de comprobación de los casos más representativos del circuito), se recurre al Análisis completo, otro código que optimiza y simula tantos circuitos como incluyamos en un directorio y que se expone a continuación. A su vez, en este apartado puede encontrarse también un pseudocódigo que explica el algoritmo de cálculo de cuellos de botella.

V. Análisis completo (optimización y simulación)

La suma de todo lo anterior se centraliza en uno de los códigos, denominado "analisisCompleto.m". En el directorio de trabajo de Matlab se colocarán todos los esquemas que quieran analizarse, debiendo nombrar a todos los archivos con la extensión ".g", lo que permitirá que el programa los estudie uno por uno sin detenerse y sin necesidad de más actuaciones por parte del usuario hasta que el proceso termine.

Se inicia indicando tres parámetros al programa:

- Bucles: número de veces que se simulará, durante un tiempo *t*, cada circuito optimizado.
- Tiempo: tiempo *t* de cada simulación.
- *"Pconfint"*: nivel de confianza, en tanto por uno, ya explicado en el último párrafo del apartado III.4. Resultados de la simulación.

Como complemento a este programa, se han creado dos subrutinas denominadas:

- **"optimCalculoThroughput.m"**: una versión muy reducida del algoritmo "calculoCuellos.m", que se limita a calcular los tres casos representativos del comportamiento del circuito, devolviendo al programa principal una lista con los lugares que conforman cada cuello calculado y el *throughput* del circuito con los deltas seleccionados (no válido para simulación).
- **"simplificarCircuito.m"**: con cada lista de lugares que facilita la anterior subrutina, el programa principal las unifica en una sola lista, que es parámetro de entrada de este

código. A partir de ella, se encarga, como su propio nombre indica, de simplificar el circuito, dejando sólo los lugares y transiciones seleccionados. A todos los efectos, reduce considerablemente el tamaño de las matrices *Pre* y *Post*.

El programa procederá a calcular los tres cuellos de botella que determinen los deltas máximos, los deltas mínimos y los deltas más probables. Se ha observado que no necesariamente los tres cuellos de botella han de compartir lugares entre sí. Es más, lo normal es ver los tres cuellos completamente independientes. Sin embargo, no supone ningún problema a la hora de simular y de hecho, es un proceso totalmente transparente al usuario (no se ve si los caminos son disjuntos o no).

El *throughput* que proporciona el algoritmo de optimización sirve para los casos de deltas máximos y mínimos de disparo (mostrarán los casos límite de rendimiento del circuito), ya que no es necesario realizar una simulación para estos dos supuestos. El único *throughput* que se simula realmente, es aquel que surge de la unión de los tres casos representativos, simulados con tiempos de disparo aleatorios (los tiempos de aquellas transiciones que hayan permanecido en el circuito simplificado).

Una vez que la subrutina de cálculo de cuellos ha proporcionado la lista de lugares de cada cuello, el programa principal los unifica todos en una matriz, teniendo por tanto un listado de todos aquellos lugares que deben permanecer en el circuito simplificado.

Para realizar la simplificación, se toman las matrices *Pre, Post,* M_0 y la lista de lugares y se siguen los siguientes pasos, tomando como ejemplo un circuito similar al de la Figura 2, página 8, con alguna modificación que permita mostrar todos los pasos. Vamos a suponer, a su vez, que el cálculo de los cuellos de botella ha determinado que éste viene definido siempre por los lugares 1, 3 y 4:

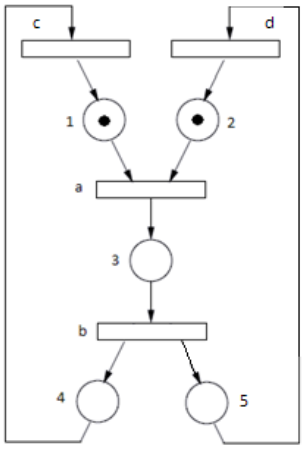


Figura 5: Simplificación de un circuito

1.- Eliminar todas aquellas filas de las tres matrices que no estén en la lista (los lugares que deben desaparecer del circuito por ser irrelevantes), desplazando hacia arriba el resto de filas remanentes. En el caso del ejemplo, los lugares que han de desaparecer son los números 2 y 5:

$$Pre = \begin{pmatrix} & a & b & c & d \\ p1 & 1 & 0 & 0 & 0 \\ p2 & 1 & 0 & 0 & 0 \\ p3 & 0 & 1 & 0 & 0 \\ p4 & 0 & 0 & 1 & 0 \\ p5 & 0 & 0 & 0 & 1 \end{pmatrix} \quad Post = \begin{pmatrix} & a & b & c & d \\ p1 & 0 & 0 & 1 & 0 \\ p2 & 0 & 0 & 0 & 1 \\ p3 & 1 & 0 & 0 & 0 \\ p4 & 0 & 1 & 0 & 0 \\ p5 & 0 & 1 & 0 & 0 \end{pmatrix} \quad M_0 = \begin{matrix} p1 \\ p2 \\ p3 \\ p4 \\ p5 \end{matrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Aunque no es necesario hacer lo mismo (todavía) con la matriz C, habría que quitar igualmente las filas 2 y 5:

$$C = \begin{pmatrix} & a & b & c & d \\ p1 & -1 & 0 & 1 & 0 \\ p2 & -1 & 0 & 0 & 1 \\ p3 & 1 & -1 & 0 & 0 \\ p4 & 0 & 1 & -1 & 0 \\ p5 & 0 & 1 & 0 & -1 \end{pmatrix}$$

Quedando las matrices como sigue:

$$Pre = \begin{pmatrix} & a & b & c & d \\ p1 & 1 & 0 & 0 & 0 \\ p3 & 0 & 1 & 0 & 0 \\ p4 & 0 & 0 & 1 & 0 \end{pmatrix} \quad Post = \begin{pmatrix} & a & b & c & d \\ p1 & 0 & 0 & 1 & 0 \\ p3 & 1 & 0 & 0 & 0 \\ p4 & 0 & 1 & 0 & 0 \end{pmatrix} \quad M_0 = \begin{matrix} p1 \\ p3 \\ p4 \end{matrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

2.- Obtener la matriz C, a partir de Post-Pre. Es evidente que la matriz quedará idéntica a la C inicial, pero sin las filas 2 y 5:

$$C = \begin{pmatrix} & a & b & c & d \\ p1 & -1 & 0 & 1 & 0 \\ p3 & 1 & -1 & 0 & 0 \\ p4 & 0 & 1 & -1 & 0 \end{pmatrix}$$

3.- A partir de la nueva matriz C, eliminar las transiciones que hayan quedado huérfanas tras la eliminación de lugares. Se puede observar que la columna de la transición *'d'* ha quedado con todos sus valores nulos (es decir, dicha transición ya no sirve: ni resta ni coloca *tokens*). Basta con eliminar aquellas columnas que tengan todos sus valores cero:

$$Pre = \begin{pmatrix} & a & b & c \\ p1 & 1 & 0 & 0 \\ p3 & 0 & 1 & 0 \\ p4 & 0 & 0 & 1 \end{pmatrix} \quad Post = \begin{pmatrix} & a & b & c \\ p1 & 0 & 0 & 1 \\ p3 & 1 & 0 & 0 \\ p4 & 0 & 1 & 0 \end{pmatrix} \quad M_0 = \begin{matrix} p1 \\ p3 \\ p4 \end{matrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

$$C = \begin{pmatrix} & a & b & c \\ p1 & -1 & 0 & 1 \\ p3 & 1 & -1 & 0 \\ p4 & 0 & 1 & -1 \end{pmatrix}$$

4.- De la matriz con los deltas de disparo de las transiciones, eliminar aquellas filas que se correspondan con las transiciones eliminadas en el paso 3.

Tras todos estos pasos, ya se tienen las nuevas matrices que definen el esquema de la Red de Petri simplificada para poder simularla. Esquema que quedaría ahora con el siguiente aspecto:

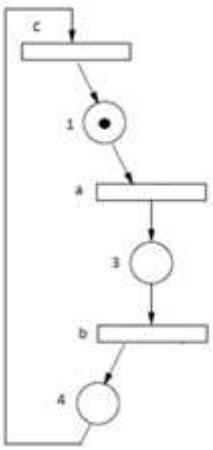


Figura 6: Red simplificada

El código "analisisCompleto.m" se limita a llamar a la función "simular.m", ya explicada en apartados anteriores, pasando como parámetros de entrada: este circuito simplificado, junto con el número de simulaciones a realizar, el tiempo de cada simulación y el nivel de confianza (los tres datos que el usuario facilita al ejecutar el programa).

En el apartado VI. Algoritmos de optimización y simulación, se puede encontrar, a modo de resumen, los algoritmos escritos en forma de pseudocódigo para comprender los pasos que sigue cada uno de los procedimientos explicados anteriormente. En cualquier caso, el código fuente real de cada programa se puede encontrar, como se ha indicado, en los anexos a la memoria.

El programa de análisis devolverá finalmente una matriz con los siguientes resultados para cada circuito evaluado. Cada línea representa un circuito diferente y cada columna, lo siguiente:

- Columna 1: *Throughput* mínimo, para deltas máximos, obtenido directamente del algoritmo de optimización (los deltas son fijos y por tanto el *throughput* también).
- Columna 2: *Throughput* de simulación, para el circuito optimizado y simulado con deltas aleatorios.
- Columna 3: *Throughput* máximo, para deltas mínimos, también obtenido del algoritmo de optimización.
- Columna 4: Error del *throughput* de simulación, entendido como la diferencia respecto de la media entre *throughput* mínimo y máximo.
- Columna 5: Varianza del *throughput* de simulación.
- Columna 6: Valor inferior del intervalo de confianza para el nivel especificado.
- Columna 7: Valor superior del intervalo de confianza para el nivel especificado.

Al respecto del error del *throughput* mostrado en la columna 4, no tiene por qué ser necesariamente un error, sino más bien, cuánto se acerca o se aleja el comportamiento del circuito de los extremos. Si los tiempos de disparo del circuito más probables están, por ejemplo, más cercanos al mínimo que al máximo, la simulación se acercará más al *throughput* máximo, sin que esto se pueda considerar un error (no todas las simulaciones han de dar un *throughput* medio respecto de los extremos).

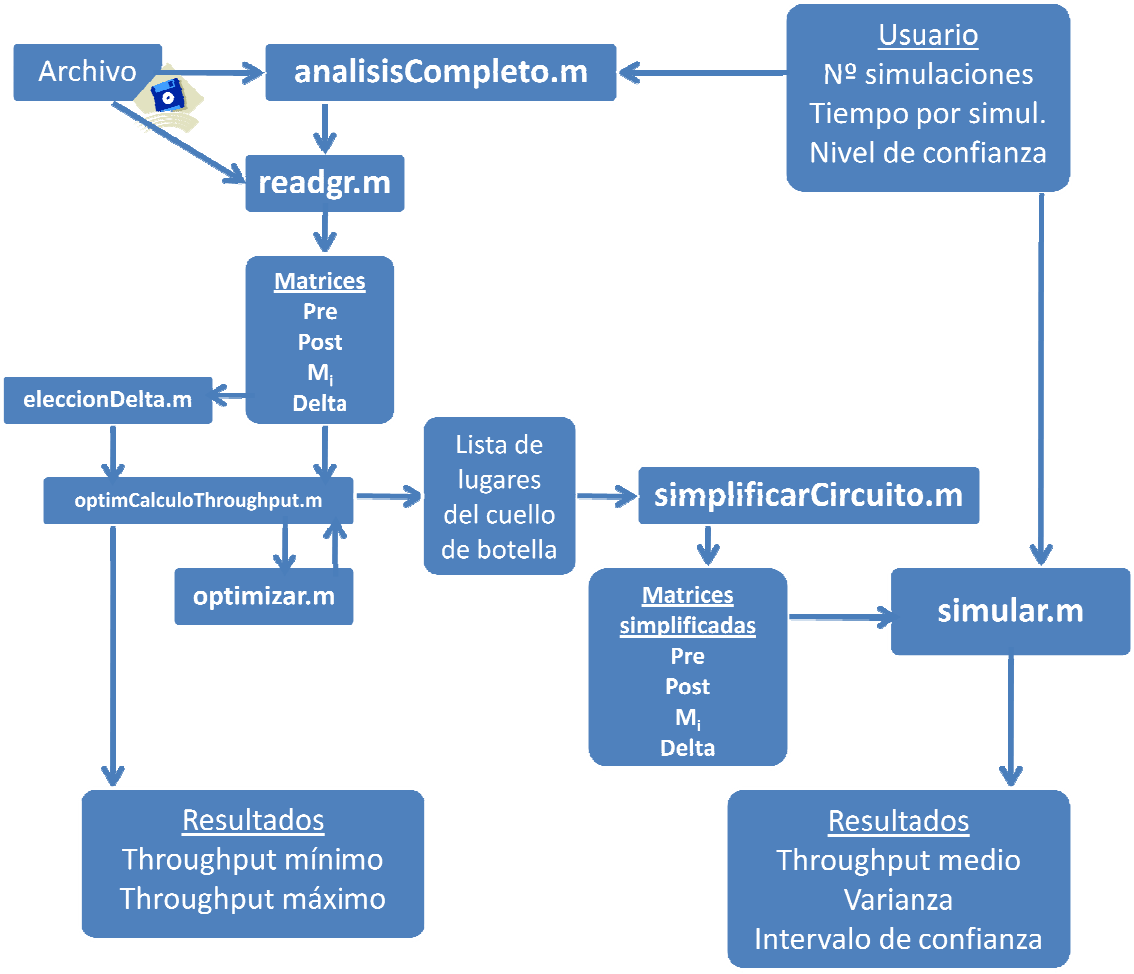
Además, este dato puede ser comprobado con las columnas 5, 6 y 7. La varianza permite observar si el dato de la media está bien ajustado o no, igual que ocurre con las columnas 6 y 7: cuánto más reducido sea el intervalo que marcan, más correcto o "probable" será que el *throughput* se encuentre en el valor indicado por la columna 2.

En el apartado VIII. Resultados, se exponen todos los resultados obtenidos para la optimización y simulación de varios circuitos modelados como Redes de Petri.

VI. Algoritmos de optimización y simulación

VI.1. Diagrama de flujo de los datos

Para comprender de forma gráfica el comportamiento de todo el proceso, se presenta el siguiente diagrama de flujo, mostrando qué información entra a qué programa y los resultados que produce:



VI.2. Pseudocódigo de optimización y simulación

A continuación se exponen, en pseudocódigo, los pasos que siguen los procedimientos principales para realizar el análisis completo de cada circuito:

```
Para cada fichero con extensión '.g' hacer
{
 Convertir fichero en matrices Pre, Post, Mi, Delta;
```

```
Optimizar circuito seleccionando Deltas máximos
{
 Calcular cuello de botella 1;
 Obtener lista de lugares del cuello de botella 1;
 Obtener throughput 1; //será el más desfavorable
}
Optimizar circuito seleccionando Deltas mínimos
{
 Calcular cuello de botella 2;
 Obtener lista de lugares del cuello de botella 2;
 Obtener throughput 2; //será el más favorable
}
Optimizar circuito seleccionando Deltas más probables
{
 Calcular cuello de botella 3;
 Obtener lista de lugares del cuello de botella 3;
}

Unificar listas de lugares del cuello de botella 1, 2 y 3;

Simplificar circuito según lista de lugares unificada
{
 Para cada lugar del circuito hacer
 {
  Si lugar no existe en lista hacer
  {
   Eliminar fila de Pre;
   Eliminar fila de Post;
   Eliminar fila de Mi;
  }
 }
 C := Post – Pre;

 Para cada columna de C hacer
 {
  Si toda la columna = 0 hacer
  {
   Eliminar columna de C;
   Eliminar columna de Pre;
   Eliminar columna de Post;
   Eliminar fila de Delta;
  }
 }
}

Simular circuito según núm. de veces, tiempo simulación y nivel de confianza
{
 Durante número de veces hacer
 {
  Reiniciar tiempo ejecución;
  Mientras que tiempo ejecución < tiempo simulación hacer
  {
   Buscar transiciones habilitadas; //Con todos sus nodos previos marcados
   Para cada transición habilitada hacer
   {
    Si transición estaba ya habilitada hacer
    {
     Respetar Delta de disparo remanente;
     Si no
      Generar número aleatorio;
      Seleccionar Delta de disparo según número aleatorio;
    }
```

```
    }

    i := Seleccionar número de transición con Delta más bajo;
    t := Guardar Delta de transición seleccionada;
    Crear matriz d
    {
     Elemento i := 1;
     Resto := 0;
    }

    //Disparar
    Mi := Mi + (Post – Pre) * d;

    //Actualizar Deltas del resto de transiciones habilitadas
    Para cada transición habilitada hacer
    {
     Delta remanente := Delta remanente – t;
    }
    Deshabilitar transición disparada;
    Actualizar número de veces que la transición se ha disparado;

    Tiempo de ejecución := Tiempo de ejecución + t;
   }

   Rendimiento de una simulación := número de veces / tiempo de ejecución
  }
  Obtener Throughput medio; //media de todos los rendimientos
  Obtener Varianza;
  Obtener Intervalo de confianza según nivel de confianza;
 }
}
```

VII. Otros métodos de análisis

Como alternativa al método de análisis de *throughputs* expuesto anteriormente, se propone también, aunque sin profundizar excesivamente en los resultados, obtener una serie de valores sobre la evolución del marcado de la red, pudiendo recurrir también a un "test de estrés" para comprobar el funcionamiento del circuito.

Este procedimiento, incluido en el código "analisisMarcado.m" consiste en analizar las variaciones que se producen en cada lugar de la red, y así poder observar cómo evoluciona la carga de trabajo de todo el sistema (si se acumula en determinados puntos, si otros se descargan demasiado rápido, etc.). Y si se desea, se puede prescindir del marcado inicial impuesto al circuito y asignar a todos los lugares de la red un número elevado de *tokens* (por defecto, 30), que es lo que se denominaría "test de estrés" al circuito.

Este programa se inicia facilitando los siguientes parámetros de partida:

- Nombre del archivo con el esquema de la red.
- Número de disparos de transiciones a simular.
- Tipo de deltas a elegir (tiempos máximos de disparo, mínimos, más probables, o aleatorios).
- Test de estrés activado o desactivado.
- Número del lugar del que se desea observar una gráfica de comportamiento.

Y devolverá una única matriz de varias columnas con los datos significativos de la evolución del marcado de la red. Cada fila de la matriz se corresponde con un lugar de la red, y las columnas significan lo que sigue a continuación:

- 1ª columna: Marcado inicial (30, en caso de test de estrés activado).
- 2ª columna: Marcado mínimo registrado en cada lugar de la red, a lo largo de todo el test.
- 3ª columna: Marcado máximo registrado en cada lugar.
- 4ª columna: Incremento máximo de *tokens* ($M_i^{máx} - M_0$), para observar acumulaciones en la carga de trabajo o cuellos de botella.
- 5ª columna: Decremento máximo de *tokens* ($M_i^{mín} - M_0$), para observar las zonas donde más carga de trabajo se libera o donde se procesa más rápidamente lo que llega.
- 6ª columna: Diferencia entre el incremento y decremento máximos ($M_i^{máx} - M_i^{mín}$) para observar si cada lugar trabaja bien siempre, mal siempre, o alternativamente en el caso de deltas aleatorios.

VIII. Resultados experimentales

Para este proyecto, se disponía de una serie de circuitos ya modelados con Redes de Petri (veintidós concretamente) donde poder aplicar todo lo diseñado en este trabajo. A continuación se presentan tablas con los resultados obtenidos, junto con análisis e interpretaciones de los mismos.

VIII.1. Simulación tipo 1 (parámetros correctos y fiables)

Para obtener resultados fiables, es necesario simular los circuitos optimizados un amplio número de veces, con tiempo también elevado en cada iteración. Se ha considerado apropiado una cifra de 500 simulaciones de 1.000 unidades de tiempo cada una. El intervalo de confianza considerado ha sido del 95%. En la tabla presentada a continuación se tienen los siguientes datos:

- Columna 1 (“Circuito”): Nombre del circuito estudiado.
- Columna 2 (“Ratio”): Relación de optimización o simplificación de la red, calculado según:

$$Ratio = \frac{Número\ de\ lugares\ remanentes\ después\ de\ optimizar}{Número\ total\ de\ lugares\ de\ la\ red\ antes\ de\ optimizar}$$

Ecuación 6: Cálculo del ratio de simplificación u optimización de la red

- Columna 3 (“*Th.* Mínimo”): *Throughput* de la red, obtenido al seleccionar los tiempos de disparo máximos.
- Columna 4 (“*Th.* Medio”): *Throughput* de la red, obtenido al simular el circuito optimizado (con tiempos de disparo aleatorios).
- Columna 5 (“*Th.* Máximo”): *Throughput* de la red, obtenido al seleccionar los tiempos de disparo mínimos.
- Columna 6 (“*Th.* Error”): Diferencia entre el *throughput* medio de la red (de simulación) y el punto medio entre el mínimo y el máximo:

$$Error = 100 \cdot \frac{\frac{Th_{mín} + Th_{máx}}{2} - Th_{med}}{Th_{med}}$$

Ecuación 7: Error del *throughput* medio, respecto de la media entre el máximo y el mínimo

- Columna 7 ("*Th.* Varianza"): Varianza del *throughput* medio de la red.
- Columnas 8 y 9 ("Intervalo de confianza"): Límites inferior y superior del intervalo de confianza, calculados según la Ecuación 3: Intervalo de confianza.

	Optimización	*Throughput*					Intervalo de confianza (95%)	
Circuito	Ratio	Mínimo	Medio	Máximo	Error	Varianza	Límite inferior	Límite superior
s1423.g	99,4580%	0,0475	0,0520	0,0558	0,5845%	0,0000	0,0519	0,0520
s1488.g	98,6918%	0,0407	0,0417	0,0434	0,9133%	0,0000	0,0416	0,0417
s1494.g	99,1688%	0,0267	0,0274	0,0279	0,0692%	0,0000	0,0273	0,0274
s208.g	72,2222%	0,1118	0,1141	0,1151	0,5691%	0,0000	0,1141	0,1142
s27.g	94,4444%	0,0555	0,0555	0,0555	0,0040%	0,0000	0,0554	0,0555
s298.g	99,9289%	0,0176	0,0180	0,0183	0,3287%	0,0000	0,0180	0,0180
s349.g	98,3957%	0,0579	0,0603	0,0733	8,7397%	0,0000	0,0603	0,0604
s382.g	94,5946%	0,0526	0,0572	0,0586	2,7446%	0,0000	0,0571	0,0572
s38417.g	99,9754%	0,0356	0,0356	0,0356	0,0021%	0,0000	0,0355	0,0356
s38584.g	99,9785%	0,0215	0,0216	0,0220	0,3372%	0,0000	0,0216	0,0217
s386.g	97,2308%	0,0695	0,0742	0,0791	0,0889%	0,0000	0,0741	0,0743
s400.g	85,6250%	0,0816	0,0837	0,0900	2,4542%	0,0000	0,0837	0,0838
s444.g	94,5652%	0,0322	0,0327	0,0330	0,3244%	0,0000	0,0327	0,0328
s510.g	98,8439%	0,0305	0,0315	0,0368	6,7527%	0,0000	0,0315	0,0316
s526.g	87,1681%	0,0677	0,0698	0,0762	3,0483%	0,0000	0,0698	0,0699
s5378.g	99,3191%	0,0501	0,0521	0,0556	1,4917%	0,0000	0,0520	0,0521
s641.g	92,9515%	0,0954	0,1020	0,1116	1,4676%	0,0000	0,1019	0,1021
s713.g	97,4170%	0,0805	0,0832	0,0955	5,7684%	0,0000	0,0831	0,0833
s820.g	98,7522%	0,0240	0,0260	0,0295	2,8637%	0,0000	0,0260	0,0260
s832.g	98,2599%	0,0523	0,0527	0,0530	0,1188%	0,0000	0,0527	0,0528
s9234.g	99,3857%	0,0498	0,0531	0,0556	0,8085%	0,0000	0,0530	0,0532
s953.g	96,6265%	0,0703	0,0723	0,0780	2,5440%	0,0000	0,0723	0,0724
scirculo.g	0%	0,0377	0,0375	0,0377	0,6853%	0,0000	0,0374	0,0375
scirculo2.g	25%	0,2500	0,2500	0,2500	0,0000%	0	0,2500	0,2500
sprueba.g	40%	0,0164	0,0218	0,0323	11,6703%	0,0000	0,0216	0,0219

Tabla 3: Resultados de optimización y simulación

Respecto a los ratios de optimización (simplificación) del circuito, se observa que las tasas son muy elevadas en prácticamente todos ellos, excepto los tres últimos: 'scirculo.g' no es más que un circuito cerrado sin bifurcaciones, por tanto es imposible optimizarlo. Los otros dos últimos circuitos son también muy pequeños con pocos caminos alternativos y por tanto tampoco son muy simplificables.

VIII.1.a. Análisis de la estabilidad en el comportamiento del circuito

El error da una idea de lo que el circuito se acerca al comportamiento medio entre los supuestos extremos de deltas de disparo mínimos y máximos. Todos los errores son muy

bajos, respaldados por una varianza también infinitesimal, lo cual viene a decir que todos ellos tienen un comportamiento **estable** en torno al centro. Una varianza elevada significaría que los resultados fueran más dispares entre sí, aunque su media fuese la misma.

Además, estos resultados también vienen avalados por la estrechez del rango de valores del intervalo de confianza, en la parte derecha de la tabla. En el 95% de las iteraciones (estadísticamente), el rendimiento medio del circuito se encontrará cercano al punto medio entre los casos extremos. Por tanto, de nuevo, se verifica el comportamiento estable en torno al centro.

Es decir, para obtener una idea de la estabilidad del circuito, hay que fijarse en:

- El rendimiento medio de simulación, que debe estar entre el mínimo y el máximo (en caso contrario, entraríamos dentro de los errores numéricos, explicados a continuación). Da una idea de si el circuito tiende a trabajar rápido o despacio.
- La varianza, que debería ser pequeña, cercana a cero, lo cual significa que todas las simulaciones se acercan al *throughput* medio (muy diferente a que unos resultados extremos se compensen -en media- en un valor intermedio).
- El intervalo de confianza, cuyo tamaño indicará, estadísticamente, qué probabilidad existe de que el rendimiento del circuito se encuentre cercano al valor medio de simulación (no en el valor medio entre mínimo y máximo). En el ejemplo presentado, se puede afirmar que en el 95% de las simulaciones, el *throughput* se ubicará dentro del intervalo indicado.

VIII.1.b. Posibles errores numéricos de simulación

Uno de los valores de la tabla ha sido resaltado en rojo (“scirculo.g”). Aquí es donde se puede observar qué tipo de errores numéricos puede aparecer en estas simulaciones. A todas luces se observa que el valor del rendimiento medio no cae dentro del intervalo que marcan (de forma estricta) los casos extremos de comportamiento del circuito, lo cual es matemáticamente imposible. Los casos extremos son calculados de forma matemática (para deltas de disparo fijos), y no de forma estadística, como los resultados de la simulación propiamente dichos.

Por ello es necesario saber identificar qué valores son incorrectos y por tanto deban corregirse o desecharse.

Se da la circunstancia de que el circuito que presenta el error es cerrado sin bifurcaciones y sin deltas variables (todas las transiciones tienen siempre el mismo tiempo de disparo). Eso implica que todas las simulaciones, al ejecutarse el mismo intervalo de tiempo, darán exactamente el mismo resultado de rendimiento y se detendrán siempre en el mismo lugar y con la misma situación de toda la red. Si el tiempo de simulación fuera múltiplo de la suma de los deltas de todas las transiciones, el error numérico desaparecería (puesto que cada simulación ejecutaría todas las transiciones el mismo número de veces). Pero al no serlo, cada iteración queda interrumpida antes de recorrer todo el circuito, y por tanto la última “vuelta” al circuito siempre será incompleta, corrompiendo el resultado final.

Es evidente que en estas circunstancias, los datos relativos a varianza e intervalo de confianza no tienen sentido, puesto que todos van referenciados al *throughput* medio de simulación.

VIII.2. Simulación tipo 2 (parámetros inapropiados)

A continuación se presentan, a modo de comparativa, los resultados de simulación si se planteara tan solo un análisis de 5 iteraciones, de 10 unidades de tiempo cada una. Se pueden observar amplias diferencias al hacer una simulación tan pequeña, como por ejemplo, errores numéricos considerables. De ahí que sea apropiado y necesario realizar cuantas más iteraciones mejor, así como establecer un tiempo de simulación muy grande para cada una de ellas:

	Optimización	*Throughput*					Intervalo de confianza (95%)	
Circuito	Ratio	Mínimo	Medio	Máximo	Error	Varianza	Límite inferior	Límite superior
s1423.g	99,4580%	0,0475	0,0550	0,0558	6,1017%	0,0025	-0,0073	0,1174
s1488.g	98,6918%	0,0407	0,0373	0,0434	12,7373%	0,0026	-0,0261	0,1007
s1494.g	99,1688%	0,0267	0,0378	0,0279	27,6865%	0,0027	-0,0266	0,1022
s208.g	72,2222%	0,1118	0,0880	0,1151	29,0074%	0,0000	0,0808	0,0951
s27.g	94,4444%	0,0555	0,0687	0,0555	19,2754%	0,0016	0,0193	0,1181
s298.g	99,9289%	0,0176	0,0188	0,0183	4,6538%	0,0018	-0,0335	0,0711
s349.g	98,3957%	0,0579	0,0555	0,0733	18,0997%	0,0026	-0,0075	0,1186
s382.g	94,5946%	0,0526	0,0449	0,0586	23,8023%	0,0017	-0,0062	0,0960
s38417.g	99,9754%	0,0356	0,0350	0,0356	1,5046%	0,0023	-0,0249	0,0949
s38584.g	99,9785%	0,0215	0,0178	0,0220	21,9655%	0,0016	-0,0316	0,0673
s386.g	97,2308%	0,0695	0,0762	0,0791	2,4869%	0,0018	0,0232	0,1291
s400.g	85,6250%	0,0816	0,0748	0,0900	14,7036%	0,0060	-0,0216	0,1712
s444.g	94,5652%	0,0322	0,0300	0,0330	8,6698%	0,0017	-0,0212	0,0812
s510.g	98,8439%	0,0305	0,0391	0,0368	13,8540%	0,0029	-0,0273	0,1055
s526.g	87,1681%	0,0677	0,0578	0,0762	24,5965%	0,0028	-0,0078	0,1233
s5378.g	99,3191%	0,0501	0,0197	0,0556	168,1680%	0,0019	-0,0350	0,0744
s641.g	92,9515%	0,0954	0,0943	0,1116	9,7429%	0,0087	-0,0217	0,2104
s713.g	97,4170%	0,0805	0,0555	0,0955	58,5776%	0,0070	-0,0482	0,1592
s820.g	98,7522%	0,0240	0,0188	0,0295	42,4224%	0,0018	-0,0334	0,0709
s832.g	98,2599%	0,0523	0,0392	0,0530	34,4102%	0,0029	-0,0274	0,1058
s9234.g	99,3857%	0,0498	0,0554	0,0556	4,9663%	0,0067	-0,0462	0,1571
s953.g	96,6265%	0,0703	0,0564	0,0780	31,5363%	0,0027	-0,0076	0,1203
scirculo.g	0%	0,0377	0,0182	0,0377	107,5472%	0,0017	-0,0323	0,0687
scirculo2.g	25%	0,2500	0,2600	0,2500	3,8462%	0,0030	0,1920	0,3280
sprueba.g	40%	0,0164	0,0545	0,0323	55,4028%	0,0025	-0,0073	0,1164

Tabla 4: Ejemplo de una simulación incorrecta

Se observa que la mayoría de rendimientos medios ni siquiera entran en el rango que definen los casos extremos.

En primer lugar, al realizar muy pocas simulaciones, se pierde la aleatoriedad del circuito: no se está comprobando el efecto que tienen los distintos tiempos de disparo de las transiciones en el rendimiento de la red.

En segundo lugar, al simular durante muy pocas unidades de tiempo, se pierde la exactitud en el cálculo del rendimiento, impidiendo que cada transición se dispare un considerable número de veces, o incluso que alguna de ellas no llegue siquiera a ejecutarse.

La conclusión que se extrae de lo anterior es que el tiempo de simulación ha de ser lo suficientemente grande como para que cada transición tenga tiempo de ejecutarse varias veces (el posible error aleatorio que se cometa en cada simulación queda atenuado, ya que

éste se divide entre el número total de disparos de las transiciones). Así mismo, han de realizarse muchas simulaciones para conocer el rango en que se mueve el rendimiento del circuito y estimar si éste es estable (valores que quedarán corroborados por una varianza baja y un intervalo de confianza estrecho para un nivel de confianza elevado, en torno al 95% ó superior).

- El tiempo de simulación aporta la exactitud al cálculo del rendimiento.
- El número de simulaciones aporta el componente estadístico, al no conocer de antemano cómo se va a comportar el circuito en una situación al azar.

IX. Conclusiones

IX.1. Resumen del trabajo

El objetivo de este proyecto consiste en diseñar un método eficaz que identifique cuellos de botella en circuitos asíncronos, modelados como Redes de Petri. Estos circuitos se caracterizan por no regirse por un reloj central que sincronice a todos sus componentes. Por tanto, alguno de ellos puede ralentizar todo o parte del sistema al funcionar a un ritmo de trabajo diferente, más lento que el resto.

Para lograr este objetivo se ha contado con:

- Un programa que interpreta un archivo de texto, de formato específico, transformándolo en las cuatro matrices que definen una Red de Petri (*Pre, Post,* M_i *y Delta*).
- Más de 20 esquemas ya modelados para simular, optimizar y analizar sus resultados.

A partir de lo anterior, se ha diseñado:

- Un simulador de Redes de Petri para hacer evolucionar el circuito durante un número arbitrario de iteraciones, de tiempo por iteración también arbitrario. Este simulador facilita resultados de rendimientos medios, varianza e intervalo de confianza.
- Un método de optimización de redes, capaz de determinar qué elementos del circuito definen su comportamiento de la forma más fiable posible. Se trata de un punto fundamental, ya que de él depende que los resultados finales estén acordes con la realidad: si la simplificación de la red es demasiado elevada, su comportamiento no se asemejará al de la red completa. En cambio, si la simplificación es muy pequeña, la simulación de la misma puede resultar lenta y poco eficiente. Una de las partes más costosas del trabajo fue determinar la mejor manera de simplificar el circuito obteniendo tanto fiabilidad de resultados como rapidez en la simulación.
- Se han integrado ambos módulos en uno solo capaz de procesar, sin más interacción con el usuario que arrancar la simulación, cuantos circuitos se desee: optimización, simulación del circuito optimizado y almacenado de resultados para su posterior representación.
- Como complemento al trabajo, también se ha implementado un método que analiza la evolución del marcado de cualquier circuito, para tener una visión general de todo el sistema: variaciones de *tokens* en cada lugar de la red (dónde se acumulan o se descargan), comprobación del correcto funcionamiento de cada lugar (si cada lugar

trabaja bien siempre, mal siempre, o de forma alternativa…), etc. Un lugar funciona "bien" cuando es capaz de desalojar sus *tokens* de manera fluida y trabaja "mal" si los *tokens* se acumulan.

Y por último, se han aplicado los programas anteriores para analizar los circuitos mencionados al principio de este capítulo, mostrando sus resultados y extraído conclusiones de los mismos.

Cabe destacar la dificultad de diseño del programa de optimización, que ha requerido de muchas pruebas para analizar la evolución del *throughput* de varios circuitos, conforme se iban modificando los deltas de disparo de cada transición. Mediante un programa casi específico para este propósito ("calculoCuellos.m"), incluido en los anexos, se observó cómo variaba el rendimiento del circuito para cada cambio en tiempos de disparo.

IX.2. Vistas a futuro

La identificación de cuellos de botella en esta clase de circuitos es un primer paso para continuar con procesos de mejora. Esta identificación servirá para estudiar qué partes de un sistema requieren de especial hincapié a la hora de mejorar su diseño. Cada circuito está compuesto de multitud de componentes que funcionan a distintos ritmos de trabajo, lo que, como se ha mencionado en esta memoria, puede provocar que varios de esos elementos ralenticen el funcionamiento de todo el sistema. Identificando adecuadamente esos cuellos de botella, se podrá centrar el esfuerzo de mejora sobre dichos componentes, bien modificándolos a mejor, ampliando su capacidad de proceso, o añadiendo más elementos de iguales características para dividir la carga de trabajo.

X. Bibliografía

[1] **Brams, G. W.** (1986). *"Las Redes de Petri".* Masson.

[2] **Cortadella, J., Kishinevsky, M., Bufistov, D., Carmona, J., & Júlvez, J.** (2008). *"Elasticity and Petri Nets".* Special Issue of Lecture Notes in Computer Science, Transactions on Petri Nets and Other Models of Concurrency.

[3] **García, M.** (2005). Material de la asignatura "Matemática Discreta", impartida en el CPS. Universidad de Zaragoza.

[4] **Jodrá, P.** (2003). Material de la asignatura "Métodos Estadísticos de la Ingeniería", impartida en el CPS. Universidad de Zaragoza.

[5] **Júlvez, J.** (2008). *"Polynomial Throughput Bounds for Equal Conflict Petri Nets with Multi-Guarded Transitions".* 5th International Conference on the Quantitative Evaluation of Systems, IEEE Computer Society.

[6] **Mar, P.** *Introducción a las Redes de Petri*. Obtenido de http://www.fortunecity.es/felices/lapaz/110/Petri_Index_Spa.html

[7] **Murata, T.** *Petri Nets: Properties, Analysis and Applications.*

[8] **Recalde, L., & Silva, M.** *Redes de Petri y evaluación de prestaciones.* Material de la asignatura "Sistemas de Eventos Discretos", impartida en el CPS. Universidad de Zaragoza.

[9] **Silva, M.** Introducing Petri Nets.

[10] **Teruel, E.** (2000). Material de las asignaturas "Teoría de Sistemas" y "Sistemas Automáticos", impartidas en el CPS. Universidad de Zaragoza.

[11] **The MathWorks, Inc.** (2008). *Matlab Product Documentation.*

XI. Anexos

XI.1. Código fuente de todos los programas

En los próximos subapartados se muestra una copia de los códigos fuente de todos los programas, que igualmente pueden encontrarse tanto en PDF como en formato texto-Matlab dentro del CD de esta memoria.

XI.1.a. readgr.m

Programa que lee un archivo de texto con el modelo de una Red de Petri y lo transforma en sus cuatro matrices características Pre, Post, M_i y Delta.

```
1 function [Pre, Post, M0, delta] = readgr(filename)
2 % This function reads a file 'filename' containing a marked graph
3 % with variable delay transitions.
4
5 % It returns:
6 % - the Pre and Post incidence matrices
7 % - the initial marking M0
8 % - the delays (and the probability of each delay)
9
10 fid = fopen(filename,'r');
11
12 Pre=sparse([]);
13 Post=sparse([]);
14 M0=sparse([]);
15
16 % Obtain Pre, Post and M0
17 nps=0; % Number of places
18 ta=fscanf(fid,'%d',1);
19 while ta>0
20 nps=nps+1;
21 tb=fscanf(fid,'%d',1);
22 Post(nps,ta)=1;
23 Pre(nps,tb)=1;
24 M0(nps,1)=fscanf(fid,'%d',1);
25 ta=fscanf(fid,'%d',1);
26 end
27
28 nts=size(Pre,2); % Number of transitions
29
30 % Obtain delta
31 % 1st column: probability for first delay; 2nd column: first delay
32 % 3rd column: probability for second delay; 4th column: second delay
33
34 delta=fscanf(fid,'%f',[4 nts]);
35 delta=delta';
```

XI.1.b. simularArchivo.m

Programa que simula una Red de Petri contenida en un archivo de texto, según los parámetros que el usuario facilite. Recurre a la función 'simular.m', contenida en el apartado siguiente.

```
1 function [Rendimientos, MediaVarianza, IntervaloRendimiento] =
simularArchivo(nombreArchivo, bucles, tiempo, pconfint)
2 %Función que ejecuta un sistema esquematizado como Red de Petri,a partir
3 %de un archivo
4 %Entradas: Nombre del archivo con el esquema de la red
5 % Número de ejecuciones
6 % Tiempo de cada ejecución
7 % Intervalo de confianza en tanto por 1
8 %Salidas: Matriz Rendimientos: throughputs de cada ejecución
9 % Matriz MediaVarianza: media y varianza de todas las ejecuciones
10 % Matriz IntervaloRendimiento: valores posibles del throughput
11 % para un intervalo de confianza
12 % dado (pconfint)
13
14 %Leer mapa inicial del circuito
15 [Pre, Post, Mi, Delta] = readgr(nombreArchivo);
16
17 %Simular el circuito
18 [Rendimientos, MediaVarianza, IntervaloRendimiento] = simular(Pre,
Post, Mi, Delta, bucles, tiempo, pconfint);
```

XI.1.c. simular.m

Programa que simula una Red de Petri a partir de sus cuatro matrices características, en función de los parámetros que el usuario facilite. Recurre a la función 'avanzarTransicion.m', contenida en el apartado siguiente.

```
1 function [Rendimientos, MediaVarianza, IntervaloRendimiento] =
simular(Pre, Post, Mi, Delta, bucles, tiempo, pconfint)
2 %Función que ejecuta un sistema esquematizado como Red de Petri
3 %Entradas: Matrices Pre, Post, Mi y Delta con el esquema de la red
4 % Número de ejecuciones
5 % Tiempo de cada ejecución
6 % Intervalo de confianza en tanto por 1
7 %Salidas: Matriz Rendimientos: throughputs de cada ejecución
8 % Matriz MediaVarianza: media y varianza de todas las ejecuciones
9 % Matriz IntervaloRendimiento: valores posibles del throughput
10 % para un intervalo de confianza
11 % dado (pconfint)
12
13 %Inicialización de variables para resultados
14 %"Resultados" almacena el número de veces que se ha disparado cada
15 %transición en cada ciclo de ejecución. No se facilita el dato al acabar la
16 %simulación. Es simplemente un residuo de código para hacer pruebas
17 Resultados = [];
18 Rendimientos = [];
19
20 for i = 1 : bucles
21 %Matriz de tiempos de disparo
22 % Columna 1: tiempos remanentes o -1 si está deshabilitada
23 % Columna 2: veces que se ha ejecutado una transición
24 Transiciones = -1 * ones(size(Pre,2),1);
25 Transiciones = [Transiciones, zeros(size(Transiciones,1),1)];
26
27 tiempoEjecucion = 0;
28 while (tiempoEjecucion < tiempo)
29 %Mientras no se acabe el tiempo, se ejecuta una transición detrás
30 %de otra
31 [Mi, Transiciones, tdisparo] = avanzarTransicion(Pre, Post, Mi, Delta,
Transiciones);
32 if (tdisparo ~= -1)
33 tiempoEjecucion = tiempoEjecucion + tdisparo;
34 else
35 tiempoEjecucion = tiempo;
36 disp('No hay ninguna transición habilitada');
37 end
38 end
39
40 Resultados = [Resultados, Transiciones(:,2)];
41 %Se supone que, transcurrido un tiempo considerable de ejecución, todas
42 %las transiciones se habrán disparado un número similar de veces. Por
43 %ello, se toma sólo la primera de ellas como referencia
44 Rendimientos = [Rendimientos, (Transiciones(1,2)/tiempoEjecucion)];
45 end
46
47 %Obtención de medias y varianzas
48 Media = mean(Rendimientos,2);
49 Varianza = var(Rendimientos');
50 MediaVarianza = [Media, Varianza'];
51
52 %Rango de valores para un intervalo de confianza dado
53 confinter = icdf('t', 1-(1-pconfint)/2, bucles-1) * sqrt(Varianza/bucles);
54 IntervaloRendimiento = [Media - confinter, Media + confinter];
```

XI.1.d. avanzarTransicion.m

Programa que dispara la transición indicada como parámetro de entrada, junto con las cuatro matrices características de la Red de Petri.

```
1 function [Mi, Transiciones, tdisparo] = avanzarTransicion(Pre, Post, Mi,
Delta, Transiciones)
2 %Función que avanza una transición en una Red de Petri. Se ejecuta como
3 %llamada de otra función
4 %Entradas: Mapa del circuito (Pre, Post, Mi, Delta)
5 % Matriz Transiciones según la función simular.m
6 %Salidas: Nuevo mapa del circuito (Mi)
7 % Nueva situación de las transiciones
8 % Tiempo de ejecución
9
10 %Comprobar qué transiciones están habilitadas
11 for i = 1 : size(Pre,2)
12 nodo = 0;
13 marcado = 0;
14 for j = 1 : size(Pre,1)
15 if (Pre(j,i) == 1)
16 nodo = nodo + 1;
17 if (Mi(j) > 0)
18 marcado = marcado + 1;
19 end
20 end
21 end
22
23 %Crear matriz con tiempos de disparo de transiciones habilitadas
24 if ((marcado == nodo) && (Transiciones(i,1) == -1))
25 aleatorio = rand;
26 if (Delta(i,1) >= aleatorio)
27 Transiciones(i,1) = Delta(i,2);
28 else
29 Transiciones(i,1) = Delta(i,4);
30 end
31 end
32 end
33
34 %Comprobar transición con menor tiempo de disparo
35 tdisparo = inf;
36 ndisparo = 0;
37 for i = 1 : size(Transiciones,1)
38 if ((Transiciones(i,1) < tdisparo) && (Transiciones(i,1) ~= -1))
39 tdisparo = Transiciones(i,1);
40 ndisparo = i;
41 end
42 end
43
44 %Iniciar si hay alguna transición habilitada
45 if (tdisparo ~= inf)
46 %Disparar transición más rápida
47 disparo = sparse(size(Transiciones,1),1);
48 disparo(ndisparo,1) = 1;
49 Mi = Mi + (Post-Pre) * disparo;
50
51 %Actualizar tiempos de disparo
52 for i = 1 : size(Transiciones,1)
53 if ((Transiciones(i,1) ~= -1) && (i ~= ndisparo))
54 Transiciones(i,1) = Transiciones(i,1) - tdisparo;
55 end
56 end
57 Transiciones(ndisparo,1) = -1;
58
59 %Actualizar número de veces
60 Transiciones(ndisparo,2) = Transiciones(ndisparo,2) + 1;
61 else
62 tdisparo = -1;
63 end
```

XI.1.e. calculoCuellos.m

Obtiene sucesivos cuellos de botella cambiando los deltas de disparo de cada transición uno por uno, a partir del archivo que contenga una Red de Petri. Recurre a la función 'eleccionDelta.m', contenida en el apartado XI.1.g.

```
function [yFinal,thoFinal,listaNodos,evolucionThroughput] =
calculoCuellos(nombreArchivo)
%Función que calcula el cuello de botella por métodos de optimización
%Entradas: Nombre de archivo con el esquema de la red
%Salidas: Matriz yFinal: cada columna es el resultado con unas Deltas
% diferentes. Los valores distintos de cero
% representan el camino más lento (cuello de
% botella)
% Matriz thoFinal: cada columna es el throughput de cada
% optimización anterior
% Matriz evolucionThroughput: muestra la evolución del
% throughput en los sucesivos cambios
% de tiempo de disparo de las
% transiciones
% Matriz listaNodos: muestra los nodos que definen los cuellos de
% botella de los casos más desfavorable, más
% favorable y más probable, por columnas
% Gráfico con la evolución del throughput en los sucesivos
% cambios

yFinal = [];
thoFinal = [];
evolucionThroughput = [];

listaNodos = [];
contFilaListaNodos = 1;

listaTransicionesProb = [];
contFilaListaTransicionesProb = 1;

%Leer mapa inicial del circuito
[Pre, Post, Mi, Delta] = readgr(nombreArchivo);

%Para que funcione correctamente la simulación, es necesario adaptar la
%matriz de los deltas de disparo para el caso de las transiciones que
%tengan sólo un tiempo de disparo. Además, se elabora una lista con todas
%las transiciones que tienen distintos tiempos de disparo
for i = 1 : size(Delta,1)
if (Delta(i,3) == 0)
Delta(i,4) = Delta(i,2);
end

if (Delta(i,1) == 0)
Delta(i,2) = Delta(i,4);
end

if ((Delta(i,1) ~= 1) && (Delta(i,1) ~= 0))
listaTransicionesProb(contFilaListaTransicionesProb,1) = i;
contFilaListaTransicionesProb = contFilaListaTransicionesProb + 1;
end
end

%Problema a resolver
for i = 1 : 3
%Elección de distintos Deltas
if (i == 1)
D = eleccionDelta(Delta, 'DeltasMaximos');
sentencia = 'Nodos cuello de botella, deltas máximos :';
end
if (i == 2)
D = eleccionDelta(Delta, 'DeltasMinimos');
sentencia = 'Nodos cuello de botella, deltas mínimos :';
end
if (i == 3)
D = eleccionDelta(Delta, 'DeltasMasProbables');
sentencia = 'Nodos cuello de botella, deltas mas probables:';
end
```

```
67
68 %Con la matriz Delta obtenida de la funcion eleccionDelta se procede a
69 %optimizar el esquema
70 [y,tho] = optimizacion(Pre, Post, D, Mi);
71
72 yFinal = [yFinal, y];
73 thoFinal = [thoFinal, tho];
74
75 for j = 1 : size(y,1)
76 if (y(j,1) > 10e-8)
77 listaNodos(contFilaListaNodos,i) = j;
78 contFilaListaNodos = contFilaListaNodos + 1;
79 sentencia = [sentencia, ' ', int2str(j)];
80 end
81 end
82 contFilaListaNodos = 1;
83 disp(sentencia);
84 end
85
86 thoAntiguo = thoFinal(1,1);
87 evolucionThroughput(1,1) = thoAntiguo;
88 indiceEvolThro = 2;
89
90 %Volvemos al peor de los casos posibles
91 D = eleccionDelta(Delta, 'DeltasMaximos');
92
93 %Se optimiza el esquema para sucesivos cambios en los tiempos de
94 %transición
95 for i = 1 : size(listaTransicionesProb,1)
96 valorAntiguo = D(listaTransicionesProb(i,1),1);
97 if (Delta(listaTransicionesProb(i,1),2) > Delta(listaTransicionesProb(i,1),4))
98 D(listaTransicionesProb(i,1),1) = Delta(listaTransicionesProb(i,1),4);
99 else
100 D(listaTransicionesProb(i,1),1) = Delta(listaTransicionesProb(i,1),2);
101 end
102 valorNuevo = D(listaTransicionesProb(i,1),1);
103
104 [yNueva,thoNueva] = optimizacion(Pre, Post, D, Mi);
105 sentencia = ['Cambiando la transicion ', int2str(listaTransicionesProb(i,1))];
106 sentencia = [sentencia, ' de ', num2str(valorAntiguo), ' a '];
107 sentencia = [sentencia, num2str(valorNuevo), ' el throughput pasa de '];
108 sentencia = [sentencia, num2str(thoAntiguo,20), ' a ', num2str(thoNueva,20)];
109 disp(sentencia);
110 thoAntiguo = thoNueva;
111 evolucionThroughput(indiceEvolThro,1) = thoAntiguo;
112 indiceEvolThro = indiceEvolThro + 1;
113 end
114
115 plot(evolucionThroughput);
```

XI.1.f. calculoCuellosConRatio.m

Programa que obtiene los cuellos de botella e indica el ratio de simplificación del circuito, a partir del archivo que contenga la Red de Petri. Recurre a la función 'eleccionDelta.m', contenida en el apartado siguiente.

```
function [yFinal,thoFinal,listaNodos] = calculoCuellosConRatio(nombreArchivo)
%Función que calcula el cuello de botella por métodos de optimización
%Entradas: Nombre de archivo con el esquema de la red
%Salidas: Matriz yFinal: cada columna es el resultado con unas Deltas
% diferentes. Los valores distintos de cero
% representan el camino más lento (cuello de
% botella)
% Matriz thoFinal: cada columna es el throughput de cada
% optimización anterior
% Matriz listaNodos: muestra los nodos que definen los cuellos de
% botella de los casos más desfavorable, más
% favorable y más probable, por columnas
% Muestra por pantalla cómo evoluciona el ratio de simplificación
% del circuito en cada iteración

yFinal = [];
thoFinal = [];

listaNodos = [];
contFilaListaNodos = 1;

listaTransicionesProb = [];
contFilaListaTransicionesProb = 1;

%Leer mapa inicial del circuito
[Pre, Post, Mi, Delta] = readgr(nombreArchivo);
ratioAntiguo = size(Mi,1);
for i = 1 : size(Delta,1)
if (Delta(i,3) == 0)
Delta(i,4) = Delta(i,2);
end

if (Delta(i,1) == 0)
Delta(i,2) = Delta(i,4);
end

if ((Delta(i,1) ~= 1) && (Delta(i,1) ~= 0))
listaTransicionesProb(contFilaListaTransicionesProb,1) = i;
contFilaListaTransicionesProb = contFilaListaTransicionesProb + 1;
end
end

%Problema a resolver
for i = 1 : 3
%Elección de distintos Deltas
if (i == 1)
D = eleccionDelta(Delta, 'DeltasMaximos');
sentencia = 'Nodos cuello de botella, deltas máximos :';
end
if (i == 2)
D = eleccionDelta(Delta, 'DeltasMinimos');
sentencia = 'Nodos cuello de botella, deltas mínimos :';
end
if (i == 3)
D = eleccionDelta(Delta, 'DeltasMasProbables');
sentencia = 'Nodos cuello de botella, deltas mas probables:';
end

[y,tho] = optimizacion(Pre, Post, D, Mi);

yFinal = [yFinal, y];
thoFinal = [thoFinal, tho];

for j = 1 : size(y,1)
if (y(j,1) > 10e-8)
listaNodos(contFilaListaNodos,i) = j;
contFilaListaNodos = contFilaListaNodos + 1;
```

```
68 sentencia = [sentencia, ' ', int2str(j)];
69 end
70 end
71 contFilaListaNodos = 1;
72 disp(sentencia);
73 end
74
75 thoAntiguo = thoFinal(1,1);
76
77 D = eleccionDelta(Delta, 'DeltasMaximos');
78 for i = 1 : size(listaTransicionesProb,1)
79 valorAntiguo = D(listaTransicionesProb(i,1),1);
80 if (Delta(listaTransicionesProb(i,1),2) > Delta(listaTransicionesProb(i,1),4))
81 D(listaTransicionesProb(i,1),1) = Delta(listaTransicionesProb(i,1),4);
82 else
83 D(listaTransicionesProb(i,1),1) = Delta(listaTransicionesProb(i,1),2);
84 end
85 valorNuevo = D(listaTransicionesProb(i,1),1);
86
87 [yNueva,thoNueva] = optimizacion(Pre, Post, D, Mi);
88 listaNodos = [];
89 contFilaListaNodos = 1;
90 for j = 1 : size(yNueva,1)
91 if (yNueva(j,1) > 10e-8)
92 listaNodos(contFilaListaNodos,1) = j;
93 contFilaListaNodos = contFilaListaNodos + 1;
94 sentencia = [sentencia, ' ', int2str(j)];
95 end
96 end
97 disp(sentencia);
98 sentencia = [];
99 disp(sentencia);
100
101 tablaCuellosBotella = [];
102 for j = 1 : size(listaNodos,1)
103 tablaCuellosBotella(listaNodos(j,1),1) = 1;
104 end
105
106 guardarPre = Pre;
107 guardarPost = Post;
108 guardarMi = Mi;
109 guardarDelta = Delta;
110 [Pre, Post, Mi, Delta] = simplificarCircuito(Pre, Post, Mi, Delta,
tablaCuellosBotella);
111 disp(['Relación de simplificación: ',num2str(100-100*size(Mi,1)
/ratioAntiguo),'%']);
112 Pre = guardarPre;
113 Post = guardarPost;
114 Mi = guardarMi;
115 Delta = guardarDelta;
116
117 thoAntiguo = thoNueva;
118 end
```

XI.1.g. eleccionDelta.m

Programa que selecciona los deltas máximos, mínimos o más probables, según la elección del usuario o del programa que recurra a esta función.

```
1 function D = eleccionDelta(Delta, tipoEleccion)
2 %Función auxiliar. Se recurre a ella desde otras funciones
3 %Entradas: Matriz Delta: matriz con valores de probabilidades y tiempos
4 % tipoEleccion: selección de Deltas
5 %Salidas: Matriz D: valores Delta según la elección pedida
6
7 for i = 1 : size(Delta,1);
8 if (Delta(i,1) == 1)
9 Delta(i,4) = Delta(i,2);
10 end
11
12 if (Delta(i,1) == 0)
13 Delta(i,2) = Delta(i,4);
14 end
15 end
16
17 switch tipoEleccion
18 case {'DeltasMaximos'}
19 Delta = [Delta(:,2),Delta(:,4)];
20 D = (max(Delta'))';
21 case {'DeltasMinimos'}
22 Delta = [Delta(:,2),Delta(:,4)];
23 D = (min(Delta'))';
24 case {'DeltasMasProbables'}
25 for i = 1 : size(Delta,1)
26 if (Delta(i,1) >= Delta(i,3))
27 nuevaDelta(i,1) = Delta(i,2);
28 else
29 nuevaDelta(i,1) = Delta(i,4);
30 end
31 end
32 D = nuevaDelta;
33 otherwise
34 disp('Error inesperado');
35 D = 0;
36 end
```

XI.1.h. optimizacion.m

Programa que organiza los datos provenientes de las matrices Pre, Post, Mi y Deltas seleccionados, para introducirlos como parámetros a la función de optimización de Matlab.

```
1 function [y,tho] = optimizacion(Pre,Post,D,Mi)
2
3 f = Pre * D;
4 f = (-1)*f';
5 %Desigualdades
6 A = [];
7 b = [];
8 %Igualdades
9 Aeq = Post - Pre;
10 Aeq = [Mi,Aeq]';
11 beq = zeros(size(Aeq,1),1);
12 beq(1,1) = 1;
13 %Límites
14 lb = zeros(size(Aeq,2),1);
15
16 [y,funObjetivo] = linprog(f,A,b,Aeq,beq,lb);
17 tho = 1/(-funObjetivo);
```

XI.1.i. analisisCompleto.m

Programa que optimiza y simula todos los archivos contenidos en el directorio de trabajo de Matlab. Se introducen los mismos parámetros de simulación que en 'simular.m'. Recurre a las funciones 'simular.m', contenida en el apartado XI.1.c; a la función 'simplificarCircuito.m', contenida en el apartado XI.1.k; a la función 'eleccionDelta.m', contenida en el apartado XI.1.g y a la función 'optimCalculoThroughput.m', contenida en el apartado XI.1.j.

```
1 function [listaArchivos, resulThroughputs] = analisisCompleto(bucles,
tiempo, pconfint)
2
3 %Crear listado de circuitos
4 lista = dir('*.g');
5 listaArchivos = lista;
6
7 %Crear tabla de resultados
8 % Columna 1: Throughput mínimo, para deltas máximos
9 % Columna 2: Throughput de simulación, para circuito optimizado
10 % Columna 3: Throughput máximo, para deltas mínimos
11 % Columna 4: Error de simulación, respecto de la media de mínimo y máximo
12 % Columna 5: Varianza del throughput de simulación
13 % Columna 6: Valor inferior del intervalo de confianza especificado
14 % Columna 7: Valor superior del intervalo de confianza especificado
15 resulThroughputs = [];
16 indiceResultados = 1;
17
18 %Crear tabla de nodos con los cuellos de botella calculados
19 tablaCuellosBotella = [];
20
21 for i = 1 : size(lista,1)
22 nombreArchivo = lista(i).name;
23 disp(['Analizando: ', nombreArchivo]);
24
25 [Pre, Post, Mi, Delta] = readgr(nombreArchivo);
26 for j = 1 : size(Delta,1)
27 if (Delta(j,3) == 0)
28 Delta(j,4) = Delta(j,2);
29 end
30
31 if (Delta(j,1) == 0)
32 Delta(j,2) = Delta(j,4);
33 end
34 end
35
36 [resulThroughputs(indiceResultados, 1), listaNodos] =
optimCalculoThroughput(Pre, Post, Mi, Delta, 'DeltasMaximos');
37 for j = 1 : size(listaNodos,1)
38 tablaCuellosBotella(listaNodos(j,1),1) = 1;
39 end
40
41 [resulThroughputs(indiceResultados, 3), listaNodos] =
optimCalculoThroughput(Pre, Post, Mi, Delta, 'DeltasMinimos');
42 for j = 1 : size(listaNodos,1)
43 tablaCuellosBotella(listaNodos(j,1),1) = 1;
44 end
45
46 [datoInnecesario, listaNodos] = optimCalculoThroughput(Pre, Post, Mi,
Delta, 'DeltasMasProbables');
47 for j = 1 : size(listaNodos,1)
48 tablaCuellosBotella(listaNodos(j,1),1) = 1;
49 end
50
51 ratioAntiguo = size(Mi,1);
52 [Pre, Post, Mi, Delta] = simplificarCircuito(Pre, Post, Mi, Delta,
tablaCuellosBotella);
53 disp(['Relación de simplificación: ',num2str(100-100*size(Mi,1)
/ratioAntiguo),'%']);
54
55 [Rendimientos, MediaVarianza, IntervaloRendimiento] = simular(Pre,
Post, Mi, Delta, bucles, tiempo, pconfint);
56 resulThroughputs(indiceResultados, 2) = MediaVarianza(1,1);
57 resulThroughputs(indiceResultados, 5) = MediaVarianza(1,2);
```

```
58 resulThroughputs(indiceResultados, 6) = IntervaloRendimiento(1,1);
59 resulThroughputs(indiceResultados, 7) = IntervaloRendimiento(1,2);
60
61 indiceResultados = indiceResultados + 1;
62 tablaCuellosBotella = [];
63 end
64
65 for i = 1 : size(resulThroughputs,1)
66 resulThroughputs(i,4) = 100*((resulThroughputs(i,1)+resulThroughputs(i,3))/2-
resulThroughputs(i,2))/resulThroughputs(i,2);
67 end
```

XI.1.j. optimCalculoThroughput.m

Programa reducido de 'calculoCuellos.m', destinado exclusivamente al código 'analisisCompleto.m'. Recurre a las funciones 'eleccionDelta.m', contenida en el apartado XI.1.g y a la función 'optimizacion.m', contenida en XI.1.h.

```
1 function [valor, listaNodos] = optimCalculoThroughput(Pre, Post, Mi,
Delta, metodo)
2 %Función que devuelve el throughput del circuito optimizado según los
3 %deltas elegidos y la lista de lugares que conforman el cuello de botella
4 %Entradas: Matrices Pre, Post, Mi, Delta
5 % Deltas elegidos (DeltasMaximos o DeltasMinimos o
6 % DeltasMasProbables)
7 %Salidas: Throughput del circuito optimizado con los deltas elegidos
8 % Lista (matriz) con los lugares que conforman el cuello de
9 % botella
10
11 %Elección de distintos Deltas
12 switch metodo
13 case {'DeltasMaximos'}
14 D = eleccionDelta(Delta, 'DeltasMaximos');
15 case {'DeltasMinimos'}
16 D = eleccionDelta(Delta, 'DeltasMinimos');
17 case {'DeltasMasProbables'}
18 D = eleccionDelta(Delta, 'DeltasMasProbables');
19 otherwise
20 disp('Error inesperado');
21 D = 0;
22 end
23
24 [y,valor] = optimizacion(Pre, Post, D, Mi);
25
26 contFilaListaNodos = 1;
27 for i = 1 : size(y,1)
28 if (y(i,1) > 10e-8)
29 listaNodos(contFilaListaNodos,1) = i;
30 contFilaListaNodos = contFilaListaNodos + 1;
31 end
32 end
```

XI.1.k. simplificarCircuito.m

Programa que simplifica las cuatro matrices de una Red de Petri según la lista de lugares que se facilite.

```
1 function [Pre, Post, Mi, Delta] = simplificarCircuito(Pre, Post, Mi,
Delta, tablaCuellosBotella)
2
3 nuevaPre = [];
4 nuevaPost = [];
5 nuevaMi = [];
6 nuevaDelta = [];
7
8 % Simplificar filas (nodos) según lista de nodos
9 for i = 1 : size(tablaCuellosBotella,1)
10 if (tablaCuellosBotella(i,1) == 1)
11 nuevaPre = [nuevaPre; Pre(i,:)];
12 nuevaPost = [nuevaPost; Post(i,:)];
13 nuevaMi = [nuevaMi; Mi(i,:)];
14 end
15 end
16
17 Pre = nuevaPre;
18 Post = nuevaPost;
19 Mi = nuevaMi;
20 C = Post - Pre;
21
22 nuevaPre = [];
23 nuevaPost = [];
24
25 % Simplificar columnas (transiciones) según nueva matriz C
26 for i = 1 : size(C, 2)
27 hayTransicionesVivas = false;
28 for j = 1 : size(C, 1)
29 if (C(j,i) ~= 0)
30 hayTransicionesVivas = true;
31 end
32 end
33
34 if hayTransicionesVivas
35 nuevaPre = [nuevaPre, Pre(:,i)];
36 nuevaPost = [nuevaPost, Post(:,i)];
37 nuevaDelta = [nuevaDelta; Delta(i,:)];
38 end
39 end
40
41 Pre = nuevaPre;
42 Post = nuevaPost;
43 Delta = nuevaDelta;
```

XI.1.l. analisisMarcado.m

Programa que analiza la evolución del marcado del circuito. Recurre a las funciones 'eleccionDelta.m', contenida en el apartado XI.1.g y a la función 'avanzarTransicion.m', contenida en XI.1.d.

```
1 function evolucionMarcado = analisisMarcado(nombreArchivo, avances,
tipoDelta, testDeEstres, graficoNodo)
2 % Función que analiza cómo evoluciona el circuito, en base a la
3 % modificación en el marcado de la red
4 % Entradas: Nombre de archivo con el esquema de la red
5 % Número de avances para hacer la simulación
6 % Elección de los deltas: 'DeltasMinimos'
7 % 'DeltasMaximos'
8 % 'DeltasMasProbables'
9 % 'DeltasAleatorios' (deltas por defecto)
10 % Test de estrés: 1, activado; 0, desactivado
11 % Este test impone un marcado inicial de 30 en
12 % todos los nodos
13 % Gráfica de la evolución en el tiempo de un nodo
14 % Salidas: Matriz evoluciónMarcado:
15 % 1ª columna: Marcado inicial (30, en caso de test de estrés)
16 % 2ª columna: Marcado mínimo de cada nodo en todo el test
17 % 3ª columna: Marcado máximo de cada nodo en todo el test
18 % 4ª columna: Incremento máximo de tokens (para observar
19 % acumulaciones o cuellos de botella)
20 % 5ª columna: Decremento máximo de tokens (para observar las
21 % zonas donde más carga de trabajo se libera)
22 % 6ª columna: Diferencia entre el incremento y decremento
23 % máximos (para observar si cada nodo trabaja bien siempre, mal
24 % siempre, o alternativamente en el caso de deltas aleatorios)
25 % Gráfica con la evolución del nodo elegido
26
27 %Leer mapa inicial del circuito
28 [Pre, Post, Mi, Delta] = readgr(nombreArchivo);
29
30 if (strcmp(tipoDelta, 'DeltasAleatorios') == false)
31 D = eleccionDelta(Delta, tipoDelta);
32 Delta(:,2) = D;
33 Delta(:,4) = D;
34 end
35
36 if (testDeEstres == 1)
37 Mi(:,1) = 30;
38 end
39
40 marcadoInicial = Mi;
41 marcadoMinimo = Mi;
42 marcadoMaximo = Mi;
43 maximaAcumulacion = [];
44 maximaLiberacion = [];
45 maximaDiferencia = [];
46
47 datosGraficaNodo = [];
48
49 %Matriz de tiempos de disparo
50 % Columna 1: tiempos remanentes o -1 si está deshabilitada
51 % Columna 2: veces que se ha ejecutado una transición
52 Transiciones = -1 * ones(size(Pre,2),1);
53 Transiciones = [Transiciones, zeros(size(Transiciones,1),1)];
54
55 tiempoEjecucion = 0;
56 datosGraficaNodo(1,1) = tiempoEjecucion;
57 datosGraficaNodo(1,2) = Mi(graficoNodo,1);
58 contGraficaNodo = 2;
59
60 for i = 1 : avances
61 [Mi, Transiciones, tiempoDisparo] = avanzarTransicion(Pre, Post, Mi,
Delta, Transiciones);
62 tiempoEjecucion = tiempoEjecucion + tiempoDisparo;
63 datosGraficaNodo(contGraficaNodo,1) = tiempoEjecucion;
64 datosGraficaNodo(contGraficaNodo,2) = Mi(graficoNodo,1);
65 contGraficaNodo = contGraficaNodo + 1;
```

```
66 for j = 1 : size(Mi,1)
67 if (Mi(j,1) > marcadoMaximo(j,1))
68 marcadoMaximo(j,1) = Mi(j,1);
69 end
70
71 if (Mi(j,1) < marcadoMinimo(j,1))
72 marcadoMinimo(j,1) = Mi(j,1);
73 end
74 end
75 end
76
77 maximaAcumulacion(:,1) = marcadoMaximo(:,1) - marcadoInicial(:,1);
78 maximaLiberacion(:,1) = marcadoMinimo(:,1) - marcadoInicial(:,1);
79 maximaDiferencia(:,1) = marcadoMaximo(:,1) - marcadoMinimo(:,1);
80 evolucionMarcado = full([marcadoInicial, marcadoMinimo, marcadoMaximo,
maximaAcumulacion, maximaLiberacion, maximaDiferencia]);
81 plot(datosGraficaNodo(:,1), datosGraficaNodo(:,2))
```