Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales

Tomas Sarquis

Agosto 2020

Programación Concurrente

El presente documento detalla la realización del **trabajo práctico número 3** de la asignatura **programación concurrente**, correspondiente al año **2019** por el alumno Tomas Sarquis, matrícula 39884977

Índice

1. Introducción

1.1. Objetivo

El objetivo del siguiente trabajo práctico es que el estudiante sea capaz de diseñar, implementar y analizar un programa que realice una simulación mediante redes de Petri, conociendo sus propiedades y monitorizando que éstas se cumplan.

Algunos de los conocimientos aplicados en este trabajo son:

- Programación en *Java*
- Concurrencia y manejo de hilos en *Java*
- Redes de Petri: propiedades y ventajas
- *UML*[1]: diagramas para modelar

1.2. Enunciado

Se debe implementar un simulador de un procesador con dos núcleos. A partir de la red de Petri de la *figura 1*, la cual representa a un procesador mono-núcleo, se deberá extender la misma a una red que modele un procesador con dos núcleos. Además, se debe implementar una política que resuelva los conflictos que se generan con las transiciones que alimentan los buffers de los núcleos.

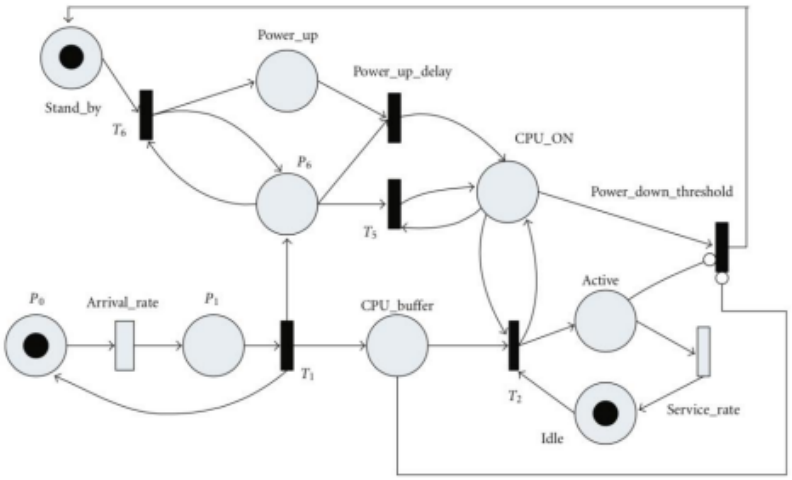

Figura 1: Red de Petri mono-núcleo

[1] *Unified Modeling Language*, `en.wikipedia.org/wiki/Unified_Modeling_Language`

2. Implementación

2.1. Red de Petri

En la *figura 2* se observa cómo fue modelada la red de Petri de acuerdo al enunciado.

La red de Petri modelada cuenta con:

- 16 plazas
- 15 transiciones

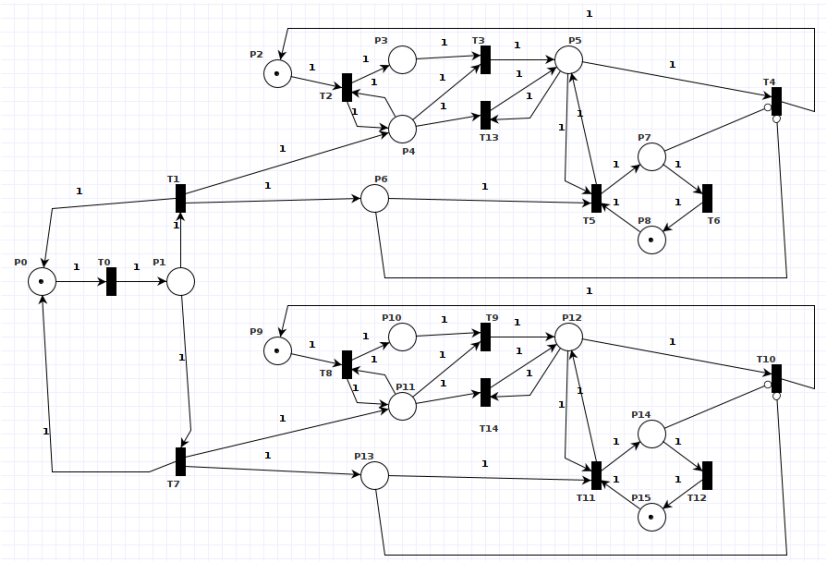

Figura 2: Red de Petri doble núcleo

2.2. Hilos

Teniendo en cuenta la red de la *figura 2*, lo siguiente es pensar cuántos hilos serán los encargados de ejecutarse en el programa.

Luego de realizar un análisis, se llegó a la conclusión de que en la red deben trabajar **7 hilos** en forma concurrente. Estos hilos son llamados:

- ProcessGenerator
- CPUPower x 2
- CPUProcessing x 2
- CPUGarbageCollector x 2

ProcessGenerator: Encargado de generar los procesos comunes a ambos CPUs. Dispara las transiciones 0, 1 y 7.

CPUPower: Maneja el encendido/apagado de cada CPU. Dispara las transiciones 2, 3 y 4 o 8, 9 y A.

CPUProcessing: Procesa las tareas de cada CPU. Dispara las transiciones 5 y 6 o B y C.

CPUGarbageCollector: Previene que no se junten *tokens* "basura". Dispara las transiciones E o D.

Cabe destacar que las letras {**A, B, C, D, E**}, en éstas últimas entradas, equivalen a las transiciones {**10, 11, 12, 13, 14**}.

2.3. Clases en Java

El programa encargado de simular el sistema se desarrolla en el lenguaje de programación *Java*.

A continuación, se describen las clases, implementadas en dicho lenguaje, que modelan el sistema:

- **Cola:** Colas (*queues*), las cuales son usadas para que los hilos puedan dormir en caso que lo necesiten
- **Colors:** Clase sin funciones que contiene códigos de colores (para loggear)
- **CPUGarbageCollector:** Modelo del hilo CPUGarbageCollector anteriormente descrito
- **CPU:** Modela un CPU. Clase encargada de comunicar los hilos Generator, Processing y GarbageCollector
- **CPUPower:** Modelo del hilo CPUPower anteriormente descrito
- **CPUProcessing:** Modelo del hilo CPUProcessing anteriormente descrito
- **Main:** Clase principal y ejecutable
- **Monitor:** Encargada de administrar los tiempos y el uso de la red de Petri
- **Politica:** A la hora de despertar un hilo, la política es quien decide a cuál hacerlo
- **ProcessBuffer:** Modelo de un buffer en el cuál se depositan Process
- **ProcessGenerator:** Modelo del hilo ProcessGenerator anteriormente descrito
- **Process:** Clase que simula un proceso de CPU. Dichos Process van a parar a un ProcessBuffer

- **RedDePetri:** Implementación de la red de Petri
- **XMLParser:** Parsea las características de la red de Petri desde un archivo *xml*

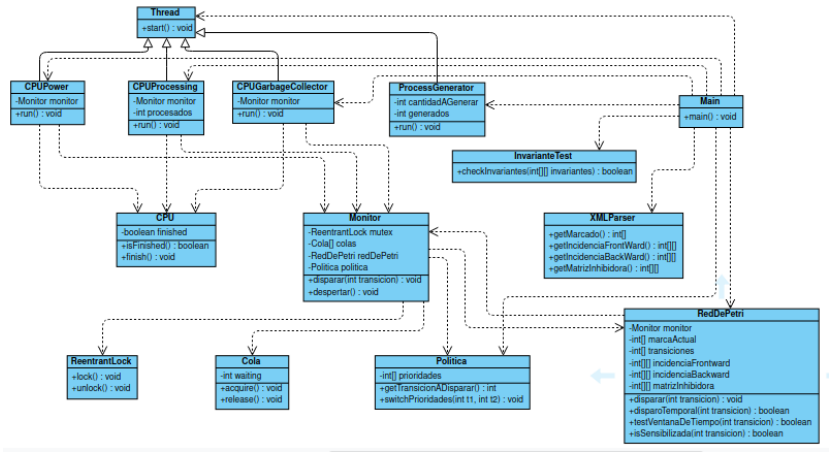

Figura 3: Diagrama de clases simplificado

Varios métodos y/o atributos fueron omitidos para mayor claridad en el diagrama de clases de la *figura 3*.

Para información más detallada acerca de las clases y sus funciones, se puede consultar la documentación[2] hecha por *Doxygen*[3].

2.4. Monitor

La clase *Monitor* es la que se encarga de administrar los hilos de nuestro programa. A continuación, para un mejor entendimiento de la clase, se observa un diagrama de secuencia que ilustra cómo un hilo entra al monitor, hace su trabajo y sale del mismo.

[2]La documentación se encuentra en el archivo *html* "documentacion/html/index.html"
[3]`www.doxygen.nl`

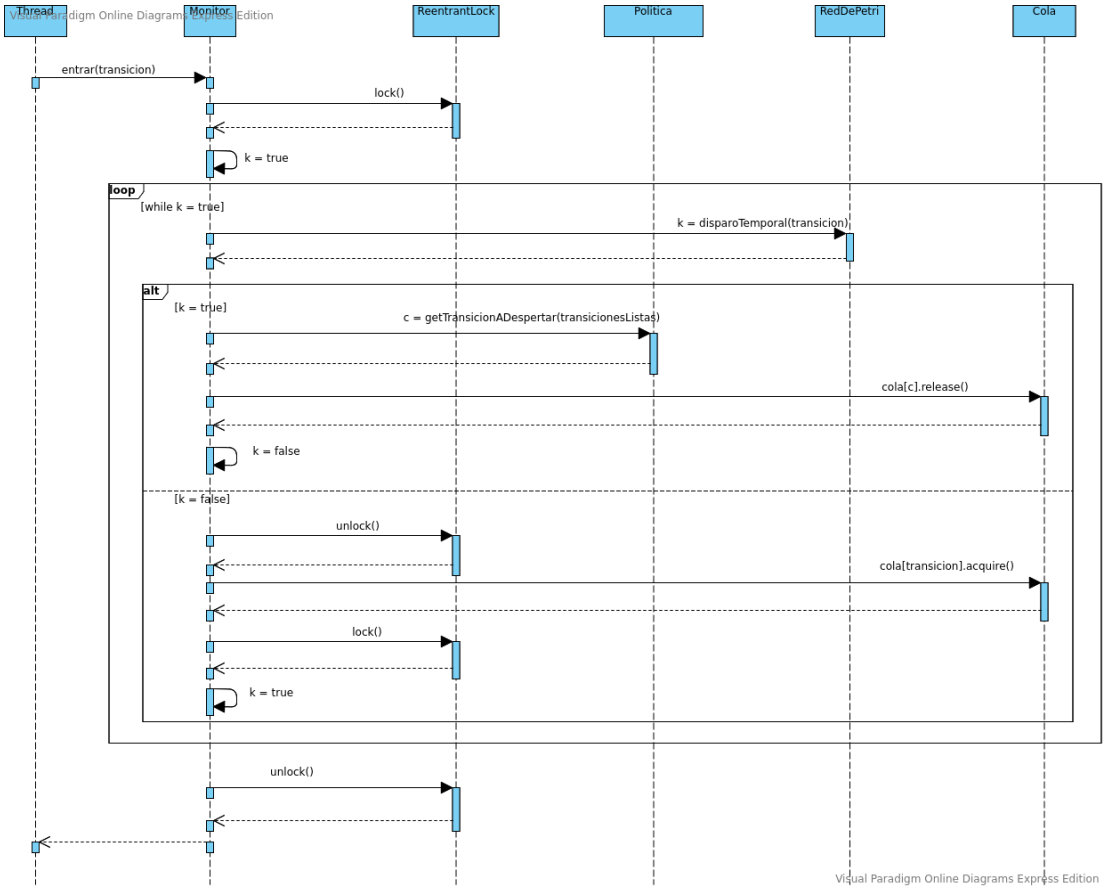

Figura 4: Diagrama de secuencia del monitor

2.5. Política

La clase *política* se encarga de facilitar la elección de a qué hilo despertar en el caso en que haya más de uno durmiendo.

La forma de hacer esto es mediante prioridades: se le asigna una prioridad **distinta** a cada una de las transiciones de la red.

Cuando queramos hacer uso de la política, simplemente le indicamos cuáles son las transiciones listas para dispararse y ella nos indicará cuál de esas transiciones es la que tiene la mayor prioridad.

Además, la clase *política* cuenta con un método capaz de intercambiar prioridades entre dos transiciones. Esto podría ser útil, en el caso en que se quiera depositar más tareas en un CPU que en el otro.

2.6. Invariantes

Sabemos que una de las maneras de chequear que la red (y nuestro programa) funciona de forma correcta, es analizando las invariantes. Estas son, las **t-invariantes** y las **p-invariantes**.

Para el análisis, se usó la herramienta Pipe[4]. Los resultados fueron los siguientes:

P-Invariantes:

P0	P1	P10	P11	P12	P13	P14	P15	P2	P3	P4	P5	P6	P7	P8	P9
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1

Figura 5: Análisis de P-Invariantes en Pipe

La *figura 5* nos enseña qué plazas están relacionadas mediante una invariante de plazas. Esta relación indica que la suma de los *tokens* en dichas plazas, en cada relación, sea siempre unitaria. Estos es:

$$M(P0) + M(P1) = 1$$

$$M(P14) + M(P15) = 1$$

$$M(P2) + M(P3) + M(P5) = 1$$

$$M(P10) + M(P12) + M(P9) = 1$$

Para comprobar que lo anterior se cumpla siempre, la clase *RedDePetri* hace un chequeo cada vez que se dispara una transición. El método de la clase que hace dicho chequeo se llama *isPInvariantesCorrecto*.

T-Invariantes:

De la misma forma, podemos observar en la *figura 6* las invariantes de transición que existen en nuestro sistema.

[4] `http://pipe2.sourceforge.net`

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	TA	TB	TC	TD	TE
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	0	0	0	1	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	0	1	0	0	0	1	1	1

Figura 6: Análisis de T-Invariantes en Pipe

Para este análisis, la clase *Main* se encarga de grabar todas las transiciones (en orden) que han sido disparadas en un archivo[5] de texto. Luego, un programa en Python (*InvariantesTest.py*) usa esa información para, mediante un algoritmo de remplazo con regex, ir eliminando las coincidencias (o *matches*) entre el *log* de transiciones y las invariantes. Si el sistema funciona correctamente, deben quedar eliminadas todas las transiciones.

Se realizó un análisis en los *softwares* Charlie/Snoopy para respaldarnos nó solamente en Pipe. En este análisis consideramos un CPU mono-core. Los resultados fueron los mismos:

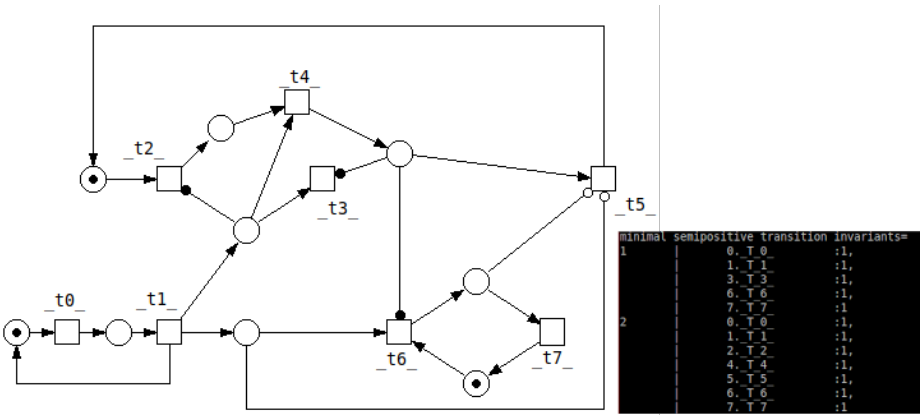

Figura 7: Análisis de T-Invariantes en Charlie/Snoopy

[5] src/T-Invariantes.txt

3. Resultados

3.1. Análisis de invariantes

P-invariantes: el análisis de las invariantes de plaza consiste en chequear, cada vez que se dispara una transición, que las ecuaciones de la sección 2.5 se cumplen. No hubo mayor problema con las mismas, comprobándose que el análisis siempre indica que las invariantes de plaza se llevan a cabo correctamente.

Los resultados de dicho análisis se muestran al finalizar cada ejecución del programa.

T-invariantes: el análisis de las invariantes de transicion consiste en observar el log de transiciones e ir remplazando las invariantes que van coincidiendo. Como se dijo anteriormente, éste análisis se realizó en Python haciendo uso de la librería de regex.

Se tuvieron en cuenta dos métodos distintos de análisis. Debido a esto, el programa consta de dos funciones de interés:

- test_colectivo_4inv(): Se realiza el análisis con las invariantes obtenidas por Pipe. Se hace un reemplazo recursivo hasta que no haya más coincidencias.

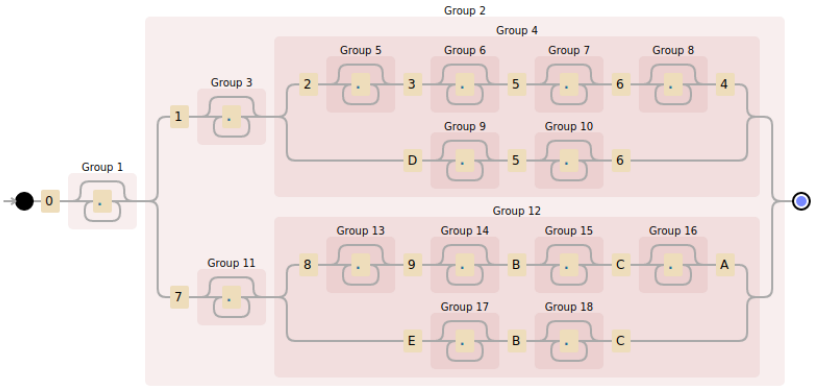

Figura 8: Regex utilizado en la función de 4 invariantes

- test_colectivo_2inv(): Se realiza el análisis con dos invariantes escogidas por el diseñador. Se hace un reemplazo recursivo hasta que no haya más coincidencias. Luego, se reemplazan las transiciones equivalentes al *GarbageCollector*.

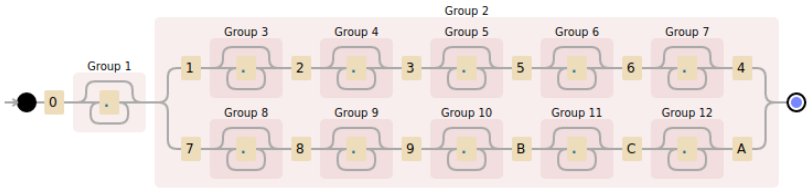

Figura 9: Regex utilizado en la función de 2 invariantes

Recordar que las letras {**A, B, C, D, E**}, en éstas últimas imágenes, equivalen a las transiciones {**10, 11, 12, 13, 14**}.

Resultados

Procesos	Trans. totales	Trans. restantes (4inv)	Trans. restantes (2inv)
250	1486	17	373
500	2984	24	771
750	4478	22	1183
1000	5956	24	1527
1250	7430	17	2008
1500	8938	19	2434

Cuadro 1: Resultados de análisis de T-Invariantes

Cabe destacar que para éste análisis se tuvo SERVICERATE = ARRIVALRATE = 10 [ms].

Como se puede ver, en ambos casos **hay transiciones restantes**. Obviamente no es lo deseado.

En el caso del método de **2 invariantes**, las transiciones restantes son muchas. Esto era esperado, ya que el método no contempla que, cada vez que se procesa una tarea, no necesariamente el CPU se enciende/apaga.

En el caso del método de **4 invariantes**, se esperaba que las transiciones restantes sean nulas. Si bien esto no es asi, podemos observar que la cantidad de transiciones restantes no aumenta con la cantidad de transiciones totales, si no, que se mantienen en el mismo rango de valores.

A pesar de que los análisis indican que el programa funciona de forma incorrecta, se invita al lector a leer la sección del apéndice **Caso de análisis**. En dicha sección, se demuestra el correcto funcionamiento del programa.

3.2. Tiempos

A continuación, en el *cuadro 2*, se describen los tiempos que lleva el programa en ejecutarse. Los tiempos de la red son los siguientes:

- $ArrivalRate = 10$ [ms]
- $ServiceRate = 15$ [ms]
- $StandByDelay = 30$ [ms]

Procesos	Tiempo de ejecución (avg)
500	2,59
1000	5,16
1500	7,76
2000	10,35
2500	12,95
3000	15,53
3500	18,13
4000	20,74

Cuadro 2: Resultados de análisis. Cada fila equivale a 5 muestras

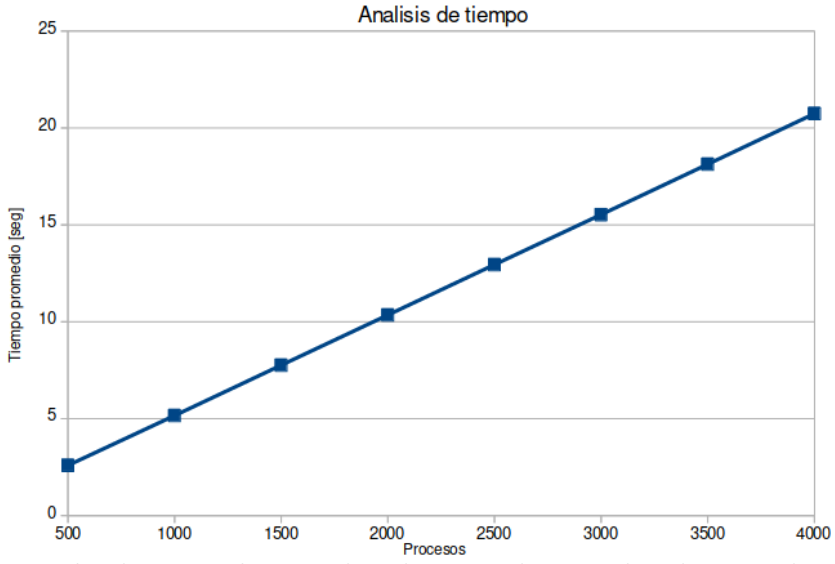

Figura 10: Gráfica de cantidad de procesos vs tiempos

Se observa una clara linealidad a medida que aumenta la cantidad de procesos.

4. Conclusión

En el presente trabajo práctico se llevó a cabo la implementación de un sistema que modela dos procesadores trabajando concurrentemente, orquestados por una red de Petri. Esta última nos da la ventaja de poder controlar que el sistema funcione correctamente, chequeando que las propiedades de la red sean las correctas. Además, el uso de redes de Petri nos da la posibilidad de escalar nuestro sistema sin mayores problemas.

Si bien me he encontrado con problemas (sección Resultados - Invariantes), se pudo comprobar, por otros medios, que el sistema de verdad funciona de forma correcta.

Dichos problemas se han intentado de resolver desde distintos frentes, no pudiéndose lograr.

5. Apéndice

5.1. Repositorio del proyecto

Dirección: `https://github.com/sarquis88/final-concurrente`

5.2. Variables globales del programa

Las siguientes variables se encuentran en la clase *Main* del programa:

- **CANTIDADPROCESOS**: cantidad de procesos a generar
- **ARRIVALRATE**: tiempo *alpha* de la llegada de nuevos procesos
- **SERVICERATE**: tiempo *alpha* de procesamiento general
- **FACTORA**: factor de multiplicación del tiempo de procesamiento para el CPU A
- **FACTORB**: factor de multiplicación del tiempo de procesamiento para el CPU B
- **STANDBYDELAY**: tiempo *alpha* del encendido de los CPUs
- **DUALCORE**: activación de *dual core*
- **LOGGING**: activación de mensajes en consola
- **REALBUFFER**: activación de bufferes *ProcessBuffer*

5.3. Caso de análisis

A continuación, se demuestra el correcto funcionamiento del sistema, para un caso particular en el cuál se procesan 8 tareas:

Procesos	Trans. totales	Trans. restantes (4inv)	Trans. restantes (2inv)
8	48	5	10

Cuadro 3: Resultados de análisis de T-Invariantes. Caso de análisis

En el cuadro anterior, se observan los resultados negativos del análisis de T-Invariantes. Las transiciones disparadas en éste caso, son:

078901B2C35A6401072801079BCBE07E07E356D5CBCB64CA

Se invita al lector, interesado, en chequear que dichas transiciones sean válidas para nuestra red de Petri. Para ésto, se debe simular la anterior secuencia de transiciones en algún *software* como Pipe.