

# Sistema doble de procesamiento de imágenes

Proyecto Final Programación Concurrente

Integrantes:    Cabrera, Augusto Gabriel  
                     Mansilla, Josías Leonel  
                     Moroz, Esteban Mauricio  
                     Pallardó, Agustín  
                     Villar, Federico Ignacio  
Profesores:    Ludemann, Mauricio  
                     Ventre, Luis Orlando  
Fecha de entrega: 18 de diciembre de 2023  
Córdoba, Argentina

# Resumen

En el siguiente informe se adjunta el desarrollo del proyecto final de la materia Programación Concurrente. Para el desarrollo del mismo, en un inicio se plantea el una solución al problema de la implementación de un monitor de concurrencia para una red de Petri no temporal.

Para el primer monitor, se trabaja con transiciones que se sensibilizan por el simple hecho de que las plaza de entrada tengan los tokens necesarios para disparar la transición. Se tiene de esa forma dos diferentes políticas para trabajar:

1. Equitativa: se busca que se disparen una cantidad de veces similar cada uno de los segmentos de la red de Petri.
2. Prioritaria: en la tercera bifurcación de la red, se tiene preferencia por uno de los segmentos, en este caso, el que tiene preferencia tiene un 80 % de probabilidad de dispararse por sobre el 20 % del opuesto.

Luego, en segunda instancia, se trabaja con una red de Petri temporal, en donde se tiene simplemente un tiempo para sensibilizarse cada transición, pero se busca que a partir de allí, ya no se se pierda esa sensibilización. Para esta última parte del trabajo, se realizan análisis similares a los planteados en el primer trabajo práctico en cuanto a lo que tiempos se refiere.

# Índice de Contenidos

<b>1. Consigna</b>	<b>1</b>
1.1. Condiciones . . . . .	1
1.2. Red de Petri Sistema Doble de Procesamiento de Imágenes . . . . .	2
1.3. Enunciado . . . . .	2
1.4. Propiedades de la Red . . . . .	3
1.5. Implementación . . . . .	3
1.6. Tiempo . . . . .	4
1.7. Políticas . . . . .	5
1.8. Requerimientos . . . . .	5
1.9. Entregables . . . . .	6
<b>2. Marco teórico</b>	<b>7</b>
2.1. Cómo se construye una red de Petri . . . . .	7
2.2. Qué definen los tokens en una red de Petri . . . . .	7
2.3. Qué simboliza una transición en las redes de Petri . . . . .	7
2.4. Cómo se define el deadlock en una red de Petri . . . . .	8
2.5. Cuáles son los tipos de conflicto en una red de Petri . . . . .	8
2.6. Qué es un invariante . . . . .	8
2.6.1. Tipos de invariante . . . . .	9
2.7. Qué es un grafo de marcado . . . . .	9
2.8. Qué es un monitor . . . . .	9
2.8.1. Cómo trabaja un monitor . . . . .	9
<b>3. Estudio de la red de Petri</b>	<b>11</b>
3.1. Reordenamiento . . . . .	11
3.2. Estados . . . . .	12
3.3. Eventos . . . . .	12
3.4. Propiedades estructurales . . . . .	13
3.4.1. Deadlock . . . . .	13
3.4.2. Vivacidad . . . . .	14
3.4.3. Limitada . . . . .	14
3.4.4. Segura . . . . .	14
3.4.5. Máquina de estados . . . . .	14
3.4.6. Grafo de marcado . . . . .	14
3.4.7. Libre elección . . . . .	15
3.4.8. Libre elección extendida . . . . .	15
3.4.9. Red simple . . . . .	15
3.4.10. Red simple extendida . . . . .	16
3.5. Invariantes . . . . .	16
3.5.1. Invariantes de plaza . . . . .	16

3.5.2.	Invariantes de transición . . . . .	17
3.6.	Cantidad de hilos y su responsabilidad . . . . .	18
3.6.1.	Caso 1 . . . . .	21
3.6.2.	Caso 2 . . . . .	22
<b>4.</b>	<b>Implementación</b>	<b>23</b>
4.1.	Parseo de la expresión regular . . . . .	23
4.2.	Monitor de concurrencia . . . . .	25
4.3.	Clases implementadas . . . . .	28
4.3.1.	Clase CQueues . . . . .	28
4.3.2.	Main . . . . .	30
4.3.3.	Policy . . . . .	32
4.3.4.	Monitor . . . . .	35
4.3.5.	Threads . . . . .	40
4.3.6.	PetriNet . . . . .	42
4.3.7.	Log . . . . .	53
<b>5.</b>	<b>Análisis del funcionamiento</b>	<b>54</b>
5.1.	Análisis temporal . . . . .	54
5.1.1.	Funcionamiento totalmente secuencial . . . . .	54
5.1.2.	Completamente paralelo . . . . .	55
5.1.3.	Comparación de secuencialidad y concurrencia . . . . .	55
5.2.	Análisis de las políticas . . . . .	56
5.2.1.	Política equitativa . . . . .	57
5.2.2.	Política 80-20 . . . . .	57
<b>6.</b>	<b>Conclusiones</b>	<b>59</b>
	<b>Referencias</b>	<b>61</b>

## Índice de Figuras

1.	Red de Petri en la consigna . . . . .	2
2.	Responsabilidades por hilos . . . . .	4
3.	Partes de una red de Petri . . . . .	7
4.	Deadlock en una red de Petri . . . . .	8
5.	Red de Petri analizada y modelada . . . . .	11
6.	Diagrama de eventos de la red de Petri . . . . .	13
7.	Justificación máquina de estados . . . . .	14
8.	Justificación grafo de marcado . . . . .	15
9.	Justificación libre elección . . . . .	15
10.	Justificación red simple . . . . .	16
11.	Invariantes de transición (expresión regular) . . . . .	18

12.	Segmentos de la red . . . . .	21
13.	Segmentos totales de la red . . . . .	22
14.	Interfaz gráfica del parser de la expresión regular . . . . .	24
15.	Monitor para la red de Petri no temporal . . . . .	25
16.	Monitor para la red de Petri no temporal . . . . .	26
17.	Diagrama de clases del sistema implementado . . . . .	28
18.	Red de Petri implementada . . . . .	54
19.	Promedio distribución 50-50 . . . . .	57
20.	Promedio distribución 80-20 . . . . .	58

## Índice de Tablas

1.	Estados posibles del sistema . . . . .	12
2.	Tabla de eventos del sistema . . . . .	13
3.	Tabla de marcados posibles para 20 imágenes a procesar (resumida, hay más de 107.000 marcados posibles) . . . . .	20
4.	Conjunto de plazas y marcado por segmento . . . . .	20
5.	Tiempos de disparo para casos secuencial y concurrente . . . . .	56
6.	Análisis de los cumplimientos de los invariantes de transición para la política equitativa . . . . .	57
7.	Análisis de los cumplimientos de los invariantes de transición para la política 80-20 . . . . .	58

## Índice de Códigos

1.	Expresión regular obtenida . . . . .	18
2.	Script para parseo de expresión regular . . . . .	23
3.	Código del método testTime() . . . . .	27
4.	Clase CQueues . . . . .	29
5.	Clase Main . . . . .	30
6.	Clase Policy . . . . .	33
7.	Clase Monitor . . . . .	35
8.	Clase Threads . . . . .	40
9.	Clase PetriNet . . . . .	44
10.	Ejemplo de selección de una transición . . . . .	56

# 1. Consigna

## 1.1. Condiciones

- El trabajo es grupal, de 5 alumnos (grupo de 4 es la excepción).
- La defensa del trabajo se coordinará con el grupo respecto a modalidad presencial o videoconferencia.
- La evaluación es individual (hay una calificación particular para cada integrante).
- Solo se corrigen los trabajos que hayan sido subidos al aula virtual (LEV).
- Los problemas de concurrencia deben estar correctamente resueltos y explicados.
- El trabajo debe implementarse en lenguaje Java.
- Se evaluará la utilización de objetos y colecciones, como así también la explicación de los conceptos relacionados a la programación concurrente.

## 1.2. Red de Petri Sistema Doble de Procesamiento de Imágenes

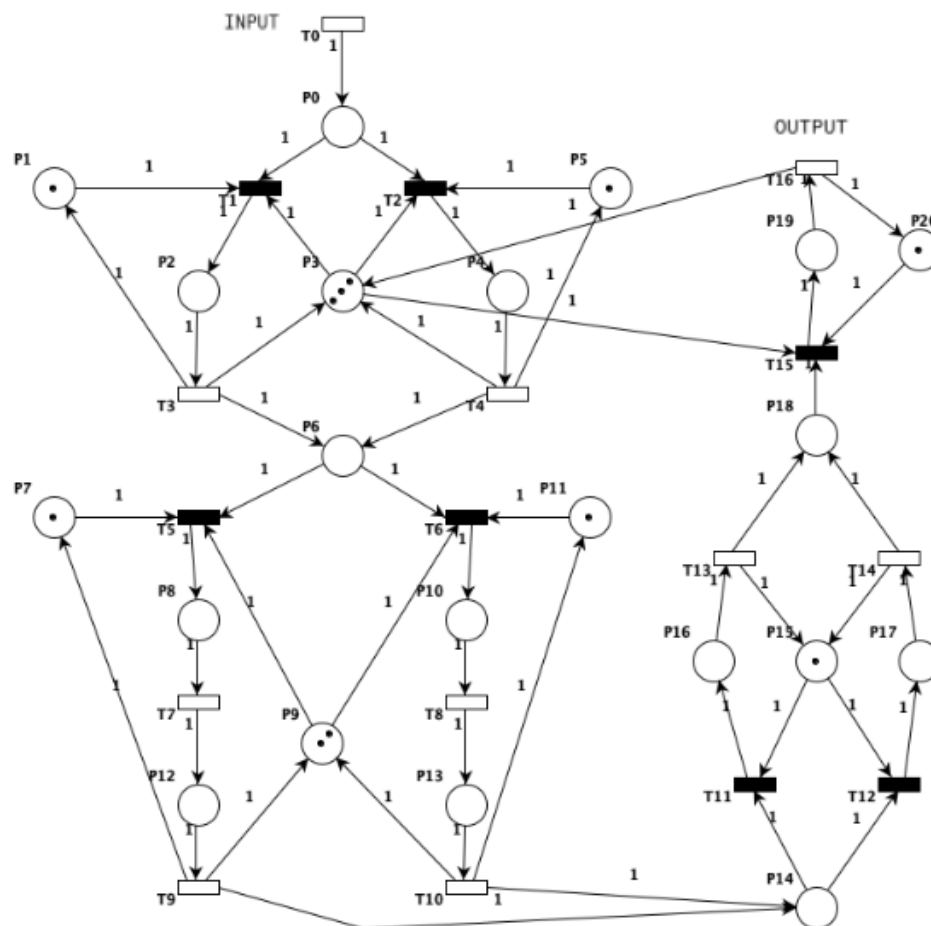


Figura 1: Red de Petri en la consigna

### 1.3. Enunciado

En la Figura 1 se observa una red de Petri que modela un sistema doble de procesamiento de imágenes.

- Las plazas  $\{P1, P3, P5, P7, P9, P11, P15\}$  representan recursos compartidos en el sistema.
- La plaza  $\{P0\}$  es una plaza idle que corresponde al buffer de entrada de imágenes al sistema.
- En las plazas  $\{P2, P4\}$  se realiza la carga de imágenes en el contenedor para procesamiento.

- La plaza {P6} representa el contenedor de imágenes a procesar.
- Las plazas {P8, P10, P11, P13} representan los estados en los cuales se realiza un ajuste de la calidad de las imágenes. Este ajuste debe hacerse en dos etapas secuenciales.
- Con la plaza {P14} se modela el contenedor de imágenes mejoradas en calidad, listas para ser recortadas a su tamaño definitivo.
- En las plazas {P16, P17} se realiza el recorte.
- En la plaza {P18} se depositan las imágenes en estado final.
- La plaza {P19} representa el proceso por el cual las imágenes son exportadas fuera del sistema.

## 1.4. Propiedades de la Red

- Es necesario determinar con una herramienta (simulador Ej: Petrinator, Pipe), la cual deberá justificar, las propiedades de la red (deadlock, vivacidad, seguridad). Cada propiedad debe ser interpretada para esta red particular.
- Indicar cual o cuales son los invariantes de plaza y los invariantes de transición de la red. Realizar una breve descripción de lo que representan en el modelo.
- NOTA IMPORTANTE 1: Para realizar el análisis de propiedades estructurales en la herramienta PIPE, es necesario unir {T16} con {P0} y eliminar {T0}.
- NOTA IMPORTANTE 2: Para realizar el análisis de invariantes en la herramienta PIPE marcar todas las transiciones como inmediatas (no temporizadas).

## 1.5. Implementación

Es necesario implementar un monitor de concurrencia para la simulación y ejecución del modelo:

- Realizar una tabla, con los estados del sistema
- Realizar una tabla, con los eventos del sistema
- Determinar la cantidad de hilos necesarios para la ejecución del sistema con el mayor paralelismo posible.
  - Caso 1: si el invariante de transición tiene un conflicto, con otro invariante, debe haber un hilo encargado de la ejecución de la/s transición/es anterior/es al conflicto y luego un hilo por invariante.
  - Caso 2: si el invariante de transición presenta un join, con otro invariante de transición, luego del join debe haber tantos hilos, como token simultáneos en la plaza, encargados de las transiciones restantes dado que hay un solo camino.



- Realizar un gráfico donde pueda observarse las responsabilidades de cada hilo con diferentes colores. A modo de ejemplo se observa en la figura 2 una red y cómo presentar lo solicitado, las flechas coloreadas representan cada tipo y cantidad de hilos.

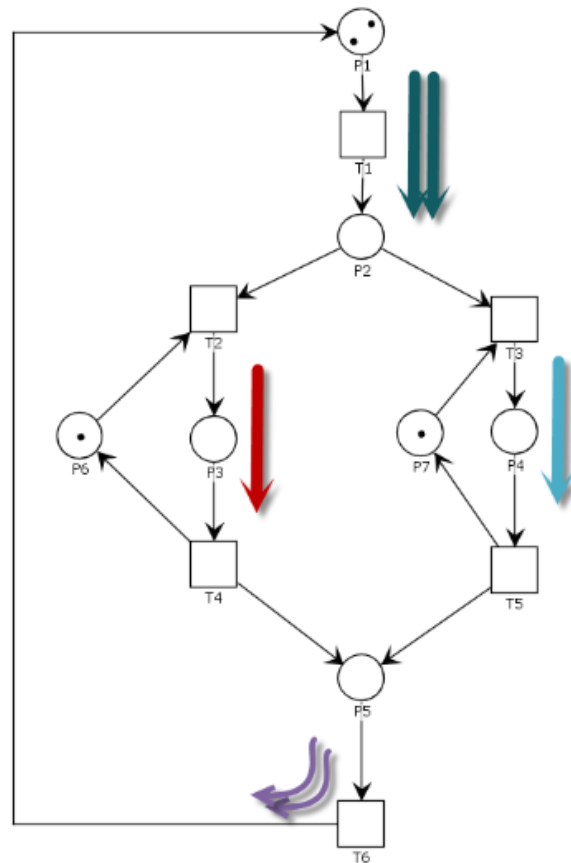


Figura 2: Responsabilidades por hilos

## 1.6. Tiempo

Una vez implementado el monitor de concurrencia, con el sistema funcionando, se deberá implementar la semántica temporal. Las transiciones  $\{T0, T3, T4, T7, T8, T9, T10, T13, T14, T16\}$  son transiciones temporales. Implementarlas y asignarles un tiempo (a elección del grupo) en milisegundos.

Se debe hacer un análisis temporal tanto analítico como práctico (ejecutando el proyecto múltiples veces), justificando los resultados obtenidos. Además, es necesario también variar los tiempos elegidos, analizar los resultados y obtener conclusiones.

## 1.7. Políticas

Es necesario para el modelado del sistema implementar políticas que resuelvan los conflictos. Se requiere considerar dos casos (ejecutados y analizados por separado e independientes uno de otro):

1. Una política de procesamiento de imágenes balanceada. La cantidad de imágenes procesadas por cada segmento de la red al finalizar la ejecución, debe ser equitativa (izquierda vs derecha). Esto se debe corroborar al finalizar la ejecución mostrando la cantidad de imágenes procesadas por cada segmento. Para ello, se debe mostrar la cantidad de veces que se ejecutaron cada una de las transiciones pertenecientes a cada segmento.
2. Una política de procesamiento que priorice el segmento izquierdo en la etapa 3. Considere que el segmento izquierdo reciba el 80 % de la carga {T11, T13}. Esto se debe corroborar de la misma manera que se indicó en el punto anterior.

## 1.8. Requerimientos

1. Implementar la red de Petri de la Figura 1 haciendo uso de una herramienta, ej: PIPE. Verificar todas sus propiedades.
2. El proyecto debe ser modelado con objetos en Java, haciendo uso de un monitor de concurrencia para guiar la ejecución de la red de Petri.
3. Implementar un objeto Política que cumpla con los objetivos establecidos en el apartado Políticas.
4. Hacer el diagrama de clases que modele el sistema.
5. Hacer el diagrama de secuencia que muestre el disparo exitoso de una transición que esté sensibilizada, mostrando el uso de la política.
6. Indicar la cantidad de hilos necesarios para la ejecución y justificar de acuerdo a lo mencionado en el apartado Implementación.
7. Realizar múltiples ejecuciones con 200 invariantes completados (para cada ejecución), y demostrar con los resultados obtenidos:
  - a) Cuán equitativa es la política implementada en el balance de carga en los invariantes.
  - b) La cantidad de cada tipo de invariante, justificando el resultado.
8. Registrar los resultados del punto 7 haciendo uso de un archivo de log para su posterior análisis.
9. Hacer un análisis de tiempos, de acuerdo a lo mencionado en el apartado Tiempo.

10. Mostrar e interpretar los invariantes de plazas y transiciones que posee la red.
11. Verificar el cumplimiento de los invariantes de plazas luego de cada disparo de la red.
12. Verificar el cumplimiento de los invariantes de transiciones mediante el análisis de un archivo log de las transiciones disparadas al finalizar la ejecución. El análisis de los invariantes debe hacerse mediante expresiones regulares.
13. El programa debe poseer una clase Main que al correrla, inicie el programa.

## 1.9. Entregables

1. Un archivo de imagen con el diagrama de clases, en buena calidad.
2. Un archivo de imagen con el diagrama de secuencias, en buena calidad.
3. En caso de que el punto 1 de los requerimientos requiera modificar la red de Petri, se debe entregar:
  - a) Un archivo de imagen con la red de Petri final, en buena calidad.
  - b) El archivo fuente de la herramienta con la que se modeló la red.
4. El código fuente Java (proyecto) de la resolución del ejercicio.
5. Un informe obligatorio que documente lo realizado, explique el código, los criterios adoptados y que explique los resultados obtenidos.

## 2. Marco teórico

Una Red de Petri es una representación matemática o gráfica de un sistema a eventos (informática y control) en el cual se puede describir la topología de un sistema distribuido, paralelo o concurrente.

Es útil en la concurrencia ya que se hacen visibles las ejecuciones concurrentes por su formalismo gráfico y matemático amplio.

### 2.1. Cómo se construye una red de Petri

- Una red de Petri consiste de una estructura y un estado, el estado se lo llama marca.
- Es una red bipartida (ya que va de Plazas a transiciones o de Transiciones a plazas), nunca de plazas a plazas o de transiciones a transiciones.

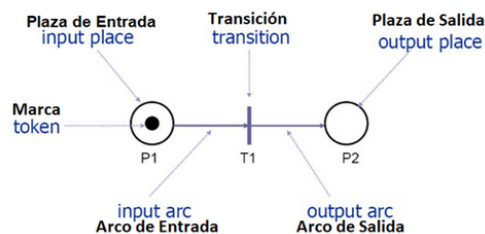


Figura 3: Partes de una red de Petri

### 2.2. Qué definen los tokens en una red de Petri

- La marca define el estado de la red de Petri, o más precisamente el estado del sistema descrito. La evolución del estado corresponde así a una evolución del marcado, una evolución que es causada por el disparo de transiciones.
- La dinámica de un sistema descrito por una red de Petri se representa por la evolución de las marcas.
- Las redes de Petri marcadas se consideran prácticamente siempre. Llamadas simplemente redes de Petri. Por otro lado, las redes no marcadas se especificarán cuando sea necesario.

### 2.3. Qué simboliza una transición en las redes de Petri

- Una transición es un detector de condiciones y transformador de estados, ya que si las condiciones se cumplen la transición se dispara y el estado cambia.
- Para que una transición se dispare cada plaza de entrada a esa transición tiene que tener al menos el mismo número de tokens del peso del arco que las conecta.

- Al realizar ese disparo:
  - Se consumen el/los tokens de las plazas que entran a la transición el número de tokens del peso del arco que une la plaza con la transición.
  - Se genera a las plazas de salida de la transición el número de tokens del peso del arco que une la transición con la plaza.

## 2.4. Cómo se define el deadlock en una red de Petri

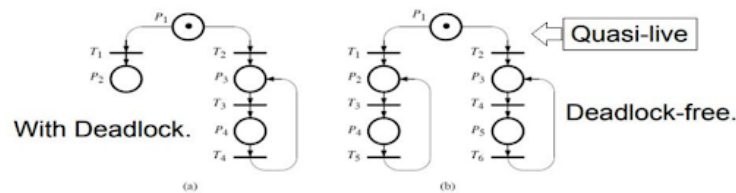


Figura 4: Deadlock en una red de Petri

- Es una marca sumidero tal que no se puede disparar ninguna transición, desde ese punto en adelante el marcado ya no puede evolucionar.
- Nota: En algunos sistemas el Deadlock es esperable, pero en los sistemas reactivos y los vistos en esta materia, no lo son. Tanto el liveness y el deadlock dependen del estado inicial

## 2.5. Cuáles son los tipos de conflicto en una red de Petri

- Estructural: es producto de su construcción, no dependen de las marcas
- Efectivo: depende de las marcas al momento del disparo. (es un conflicto estructural)
- General: si se quisiera disparar en este momento las transiciones con su grado de habilitación, de no ser posible, se está ante un conflicto general (es también un conflicto estructural).

## 2.6. Qué es un invariante

- Son propiedades de las redes de Petri.
- Los invariantes permiten caracterizar ciertas propiedades de las marcas alcanzables y de las transiciones inalterables, independiente de la evolución.

### 2.6.1. Tipos de invariante

- Invariantes de plaza (Componentes conservativos) : es un conjunto de plazas que la suma de sus tokens se mantiene constantes a lo largo de todo el marcado de la red.
- Invariantes de transición (componentes repetitivos) : es un conjunto de transiciones el cual yo las puedo disparar y vuelvo al estado al que estaba (se consiguen con expresiones regulares). Para el invariante de transición sólo interesa la mínima repetición.

## 2.7. Qué es un grafo de marcado

- Es un grafo dirigido que representa el comportamiento de una red de Petri.
- Los nodos son los marcados y las aristas son los disparos.
- Permite observar de manera gráfica el avance del sistema y la presencia de deadlocks.

## 2.8. Qué es un monitor

- Los monitores son módulos de alto nivel de abstracción que se utilizan para gestionar recursos que se van a utilizar de manera concurrente.
- El objetivo de un monitor es centralizar la sincronización y la concurrencia en un solo punto, evitando la dispersión de semáforos por todo el código. Esto permite secuenciar la toma de decisiones, mientras que las tareas se resuelven de manera concurrente.
- Un monitor es una instancia de una clase que puede ser utilizada de manera segura por múltiples hilos.
- Un monitor no es un objeto vivo en sí mismo, ya que su “vida” es proporcionada por los hilos que lo ejecutan.
- Un monitor puede ser visto como una aduana, donde se permite o no el acceso a recursos compartidos, o se espera.

### 2.8.1. Cómo trabaja un monitor

- Los hilos consultan al monitor para determinar si pueden realizar una acción. El monitor, en función del recurso, determinará si la acción puede llevarse a cabo. Si la respuesta es afirmativa, el hilo se desplaza fuera del monitor para realizar su trabajo.
- Dentro del monitor, solo puede haber un hilo activo a la vez.
- Se deben cumplir las siguientes condiciones:
  - Todos los métodos del monitor deben ejecutarse en exclusión mutua.
  - La sincronización se define por la condición de cada uno de los métodos del monitor.

- Los procesos deben ser bloqueados hasta que se cumplan las condiciones para su ejecución.
- Un proceso bloqueado debe ser notificado (señalizado) para que pueda continuar su ejecución cuando se cumplan las condiciones para su ejecución.

## 3. Estudio de la red de Petri

### 3.1. Reordenamiento

La red de Petri que se implementa en el trabajo práctico es la siguiente. Puede apreciarse que se modifican los nombres de las plazas y transiciones para mayor comodidad. Además de lo anterior cuenta con las modificaciones mencionadas por los profesores en cuanto a la red en sí, para la correcta simulación del sistema de doble procesamiento de imágenes.

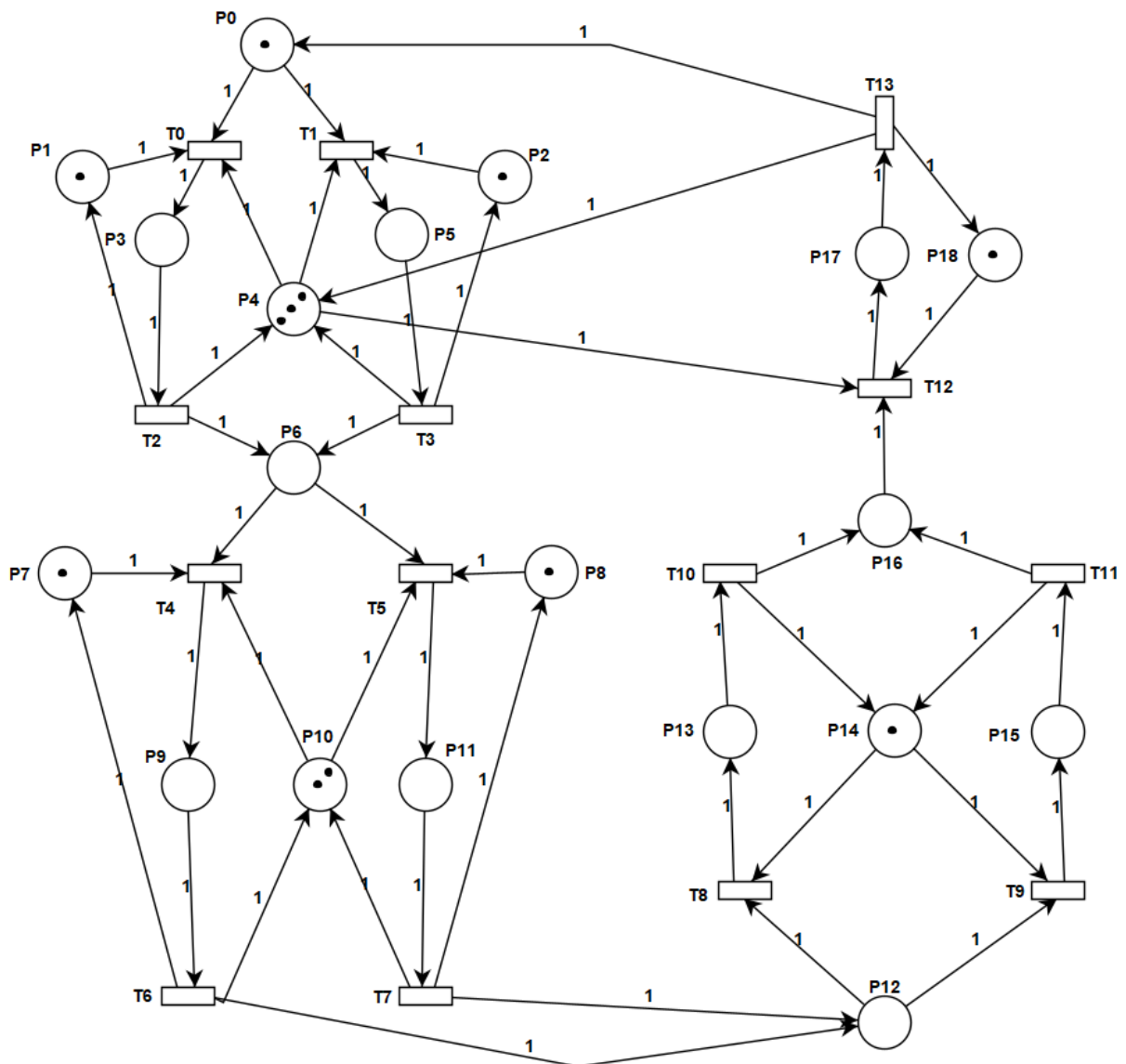


Figura 5: Red de Petri analizada y modelada

Se tiene entonces la siguiente caracterización:



- La plaza  $P_0$  es una plaza idle que corresponde al buffer de entrada de imágenes al sistema.
- Las plazas  $P_1, P_2, P_4, P_7, P_8, P_{10}, P_{14}, P_{18}$  representan recursos compartidos del sistema.
- Las plazas  $P_3, P_5$  representan la carga de imágenes en el contenedor para procesamiento.
- La plaza  $P_6$  representa el contenedor de imágenes a procesar.
- Las plazas  $P_9, P_{11}, P_{13}$  representan los estados en los cuales se realiza un ajuste de la calidad de las imágenes.
- La plaza  $P_{12}$  modela el contenedor de imágenes mejoradas en calidad, listas para ser recortadas a su tamaño definitivo.
- En las plazas  $P_{13}, P_{15}$  se realiza el recorte.
- En la plaza  $P_{16}$  se depositan las imágenes en estado final.
- La plaza  $P_{17}$  representa el proceso por el cual las imágenes son exportadas fuera del sistema.

## 3.2. Estados

De la red, se obtiene la siguiente tabla de estados.

Tabla 1: Estados posibles del sistema

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18
M0	1	1	1	0	3	0	0	1	1	0	2	0	0	0	1	0	0	0	1
M1	0	1	0	0	2	1	0	1	1	0	2	0	0	0	1	0	0	0	1
M2	0	0	1	1	2	0	0	1	1	0	2	0	0	0	1	0	0	0	1
M3	0	1	1	0	3	0	1	1	1	0	2	0	0	0	1	0	0	0	1
M4	0	1	1	0	3	0	0	1	0	0	1	1	0	0	1	0	0	0	1
M5	0	1	1	0	3	0	0	0	1	1	1	0	0	0	1	0	0	0	1
M6	0	1	1	0	3	0	0	1	1	0	2	0	1	0	1	0	0	0	1
M7	0	1	1	0	3	0	0	1	1	0	2	0	0	0	0	1	0	0	1
M8	0	1	1	0	3	0	0	1	1	0	2	0	0	0	1	0	0	0	1
M9	0	1	1	0	3	0	0	1	1	0	2	0	0	0	1	0	1	0	1
M10	0	1	1	0	2	0	0	1	1	0	2	0	0	0	1	0	0	1	0

## 3.3. Eventos

Ahora, en cuanto a los eventos del sistema, se tiene un diagrama, que se adjunta a continuación, además de la tabla que muestra la información.

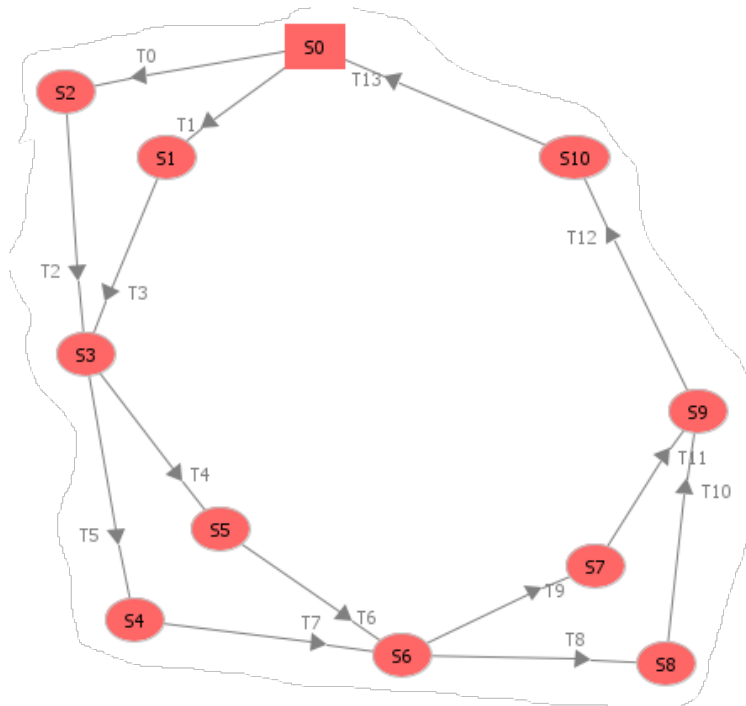


Figura 6: Diagrama de eventos de la red de Petri

Tabla 2: Tabla de eventos del sistema

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
Marking	(1, 1, 2, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 3, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 2, 1, 0, 1, 1, 0)	(0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 2, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 3, 0, 1, 1, 1, 0)	(0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 3, 0, 0, 1, 0, 0)	(0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 3, 0, 0, 0, 1, 1)	(0, 1, 2, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 3, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 3, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 3, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 3, 0, 0, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 0, 1, 0, 0, 1, 0, 2, 0, 0, 1, 1, 0)
Edges from	S10 (T13)	S0 (T1)	S0 (T0)	S1 (T3); S2 (T2)	S3 (TS)	S3 (T4)	S4 (T7); S5 (T6)	S6 (T9)	S6 (T8)	S7 (T11); S8 (T10)	S9 (T12)
Edges to	S1 (T1); S2 (T0)	S3 (T3)	S3 (T2)	S4 (T5); S5 (T4)	S6 (T7)	S6 (T6)	S7 (T9); S8 (TS)	S9 (T11)	S9 (T10)	S10 (T12)	S0 (T13)

### 3.4. Propiedades estructurales

Para el análisis de las propiedades estructurales de la red, se trabajó con la herramienta PIPE.

#### 3.4.1. Deadlock

Para la red en análisis, no existe secuencia de disparo tal que se llegue a un estado de Deadlock, por lo que se dice que está “Deadlock free”.

### 3.4.2. Vivacidad

La red es viva, lo que quiere decir que en cualquier marcado posible siempre existe una secuencia para disparar cualquiera de las transiciones. Por lo tanto, decimos que las imágenes siempre podrán ser procesadas por el sistema en algún momento.

### 3.4.3. Limitada

La red es limitada, ya que para cualquiera de los estados posibles existe una cantidad finita de tokens en las plazas.

### 3.4.4. Segura

Esta red en análisis no es segura, esto puede verse de forma inmediata, ya que no todas las plazas tienen un único token.

### 3.4.5. Máquina de estados

No se está ante una máquina de estados, ya que algunas transiciones poseen más de un arco de entrada y/o salida.

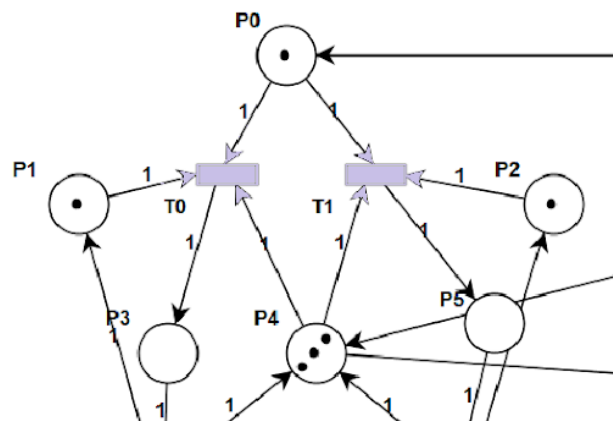


Figura 7: Justificación máquina de estados

### 3.4.6. Grafo de marcado

Esta red no es un grafo de marcado porque existen plazas con más de un arco de entrada y/o salida.

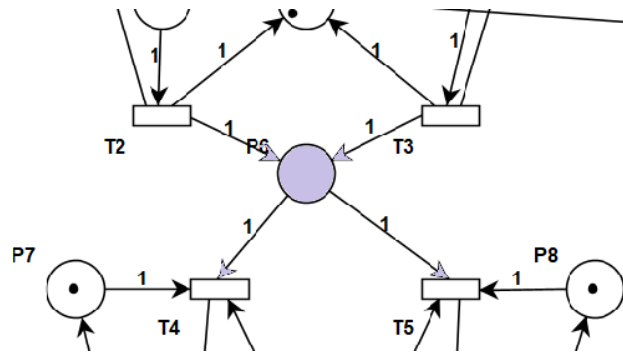


Figura 8: Justificación grafo de marcado

### 3.4.7. Libre elección

El sistema modelado no es de libre elección por poseer varios conflictos estructurales, y además esos conflictos no son producidos por las mismas plazas que sensibilizan esas transiciones.

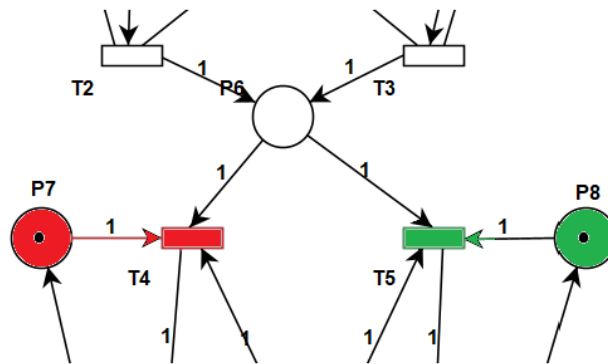


Figura 9: Justificación libre elección

### 3.4.8. Libre elección extendida

La red de Petri no es de Libre elección extendida, ya que no hay dos plazas asociadas a una misma transición, y no tienen asociadas las mismas plazas.

### 3.4.9. Red simple

Esta red de Petri no simple ya que hay por lo menos una transición la cual está en conflicto con más de una transición.



$$M_{17} + M_{18} = 1 \quad (8)$$

$$M_3 + M_4 + M_5 + M_{17} = 3 \quad (9)$$

Ahora, en cuanto a cada uno de los invariantes, se tiene:

1. Representa el recorrido de la imagen por el sistema completo.
2. Primer proceso que carga imagen en un contenedor (capacidad limitada).
3. Segundo proceso que carga imagen en un contenedor (capacidad limitada).
4. Recurso compartido en el proceso de recorte.
5. Primer proceso de mejora de calidad (capacidad limitada).
6. Segundo proceso de mejora de calidad (capacidad limitada).
7. Dos recursos que permiten recortar la imagen.
8. Capacidad limitada en el buffer de salida.
9. Relación entre buffers de entrada y salida.

### 3.5.2. Invariantes de transición

En la red en cuestión aparecen 8 invariantes de transición, estos representan diferentes caminos que puede tomar la imagen a lo largo del proceso modelado, se enumeran:

1.  $T_1 \rightarrow T_3 \rightarrow T_5 \rightarrow T_7 \rightarrow T_9 \rightarrow T_{11} \rightarrow T_{12} \rightarrow T_{13}$
2.  $T_1 \rightarrow T_3 \rightarrow T_5 \rightarrow T_7 \rightarrow T_8 \rightarrow T_{10} \rightarrow T_{12} \rightarrow T_{13}$
3.  $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_6 \rightarrow T_9 \rightarrow T_{11} \rightarrow T_{12} \rightarrow T_{13}$
4.  $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_6 \rightarrow T_8 \rightarrow T_{10} \rightarrow T_{12} \rightarrow T_{13}$
5.  $T_0 \rightarrow T_2 \rightarrow T_5 \rightarrow T_7 \rightarrow T_9 \rightarrow T_{11} \rightarrow T_{12} \rightarrow T_{13}$
6.  $T_0 \rightarrow T_2 \rightarrow T_5 \rightarrow T_7 \rightarrow T_8 \rightarrow T_{10} \rightarrow T_{12} \rightarrow T_{13}$
7.  $T_0 \rightarrow T_2 \rightarrow T_4 \rightarrow T_6 \rightarrow T_9 \rightarrow T_{11} \rightarrow T_{12} \rightarrow T_{13}$
8.  $T_0 \rightarrow T_2 \rightarrow T_4 \rightarrow T_6 \rightarrow T_8 \rightarrow T_{10} \rightarrow T_{12} \rightarrow T_{13}$

De los invariantes de transición se puede obtener una expresión regular, cuya gráfica se ve adjuntada a continuación.

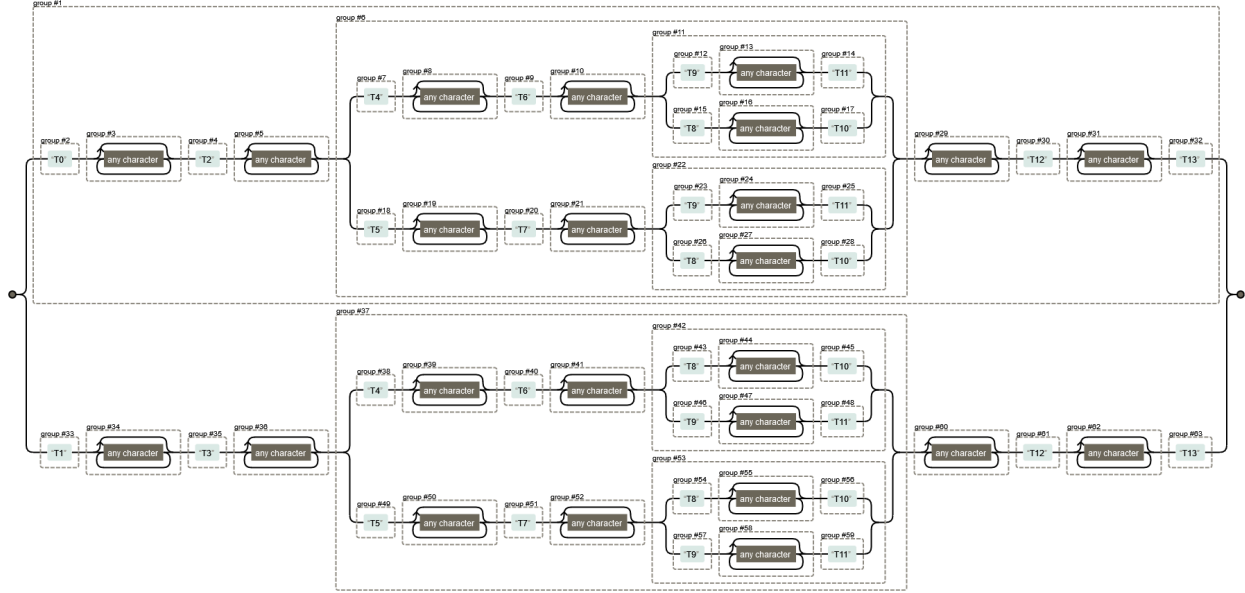


Figura 11: Invariantes de transición (expresión regular)

La expresión regular, puede colocarse en diferentes herramientas, como por ejemplo **Re-gexper** con el siguiente código:

Código 1: Expresión regular obtenida

1

```
('((T0)(.??)(T2)(.??)((T4)(.??)(T6)(.??)((T9)(.??)(T11)|(T8)(.??)(T10))|(T5)(.??)(T7)
(.??)((T9)(.??)(T11)|(T8)(.??)(T10)))(.??)(T12)(.??)(T13))|(T1)(.??)(T3)(.??)((T4)
(.??)(T6)(.??)((T8)(.??)(T10)|(T9)(.??)(T11))|(T5)(.??)(T7)(.??)((T8)(.??)(T10)|(T9)
(.??)(T11)))(.??)(T12)(.??)(T13))
```

### 3.6. Cantidad de hilos y su responsabilidad

Se toma como referencia para el cálculo de los hilos, la bibliografía mencionada por la cátedra. Con esto en mente, se menciona que se debe primero obtener los conjuntos de plazas asociadas a cada uno de los invariantes de transición, se tiene así (cada  $P_i$ ):

$$PI_1 = \{P_0, P_2, P_4, P_5, P_6, P_8, P_{10}, P_{11}, P_{12}, P_{14}, P_{15}, P_{16}, P_{17}, P_{18}\} \quad (10)$$

$$PI_2 = \{P_0, P_2, P_4, P_5, P_6, P_8, P_{10}, P_{11}, P_{12}, P_{13}, P_{14}, P_{16}, P_{17}, P_{18}\} \quad (11)$$

$$PI_3 = \{P_0, P_2, P_4, P_5, P_6, P_7, P_9, P_{10}, P_{12}, P_{14}, P_{15}, P_{16}, P_{17}, P_{18}\} \quad (12)$$

$$PI_4 = \{P_0, P_2, P_4, P_5, P_6, P_7, P_9, P_{10}, P_{12}, P_{13}, P_{14}, P_{16}, P_{17}, P_{18}\} \quad (13)$$

$$PI_5 = \{P_0, P_1, P_3, P_4, P_6, P_8, P_{10}, P_{11}, P_{12}, P_{14}, P_{15}, P_{16}, P_{17}, P_{18}\} \quad (14)$$

$$PI_6 = \{P_0, P_1, P_3, P_4, P_6, P_8, P_{10}, P_{11}, P_{12}, P_{13}, P_{14}, P_{16}, P_{17}, P_{18}\} \quad (15)$$

$$PI_7 = \{P_0, P_1, P_3, P_4, P_6, P_7, P_9, P_{10}, P_{12}, P_{14}, P_{15}, P_{16}, P_{17}, P_{18}\} \quad (16)$$

$$PI_8 = \{P_0, P_1, P_3, P_4, P_6, P_7, P_9, P_{10}, P_{12}, P_{13}, P_{14}, P_{16}, P_{17}, P_{18}\} \quad (17)$$

Ahora se quitan las plazas que aportan con restricción, recursos y fuentes. De esa forma, se elimina:

$$\{P_0, P_1, P_2, P_4, P_6, P_7, P_8, P_{10}, P_{12}, P_{14}, P_{16}, P_{17}, P_{18}\} \quad (18)$$

De esa forma, se obtienen las siguientes plazas de acción asociadas a cada invariante de transición:

$$PA_1 = \{P_5, P_{11}, P_{15}, P_{17}\} \quad (19)$$

$$PA_2 = \{P_5, P_{11}, P_{13}, P_{17}\} \quad (20)$$

$$PA_3 = \{P_5, P_9, P_{15}, P_{17}\} \quad (21)$$

$$PA_4 = \{P_5, P_9, P_{13}, P_{17}\} \quad (22)$$

$$PA_5 = \{P_3, P_5, P_{11}, P_{15}, P_{17}\} \quad (23)$$

$$PA_6 = \{P_3, P_5, P_{11}, P_{13}, P_{17}\} \quad (24)$$

$$PA_7 = \{P_3, P_5, P_9, P_{15}, P_{17}\} \quad (25)$$

$$PA_8 = \{P_3, P_5, P_9, P_{13}, P_{17}\} \quad (26)$$

De las plazas de acción, se puede plantear la siguiente tabla, en donde se resumen algunos estados, lo importante es visualizar que la máxima suma de tokens es 6, lo que implica que la cantidad máxima activa de hilos en simultáneo en el sistema es de **6**.

Luego, mediante la aplicación del algoritmo para asignar la responsabilidad a los hilos, se



Tabla 3: Tabla de marcados posibles para 20 imágenes a procesar (resumida, hay más de 107.000 marcados posibles)

$P_3$	$P_5$	$P_9$	$P_{11}$	$P_{13}$	$P_{15}$	$P_{17}$	Suma de tokens
1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	1	1	1	0	1	6
1	1	1	1	0	1	1	6

debe obtener el conjunto de plazas de cada segmento, y luego a cada segmento se le obtiene el marcado.

Tabla 4: Conjunto de plazas y marcado por segmento

Conjunto de plazas por segmento	Marcado por segmento
$PS_A = P_3$	$MS_A = 0 1$
$PS_B = P_5$	$MS_B = 0 1$
$PS_C = P_9$	$MS_C = 0 1$
$PS_D = P_{11}$	$MS_D = 0 1$
$PS_E = P_{13}$	$MS_E = 0 1$
$PS_F = P_{15}$	$MS_F = 0 1$
$PS_G = P_{17}$	$MS_G = 0 1$

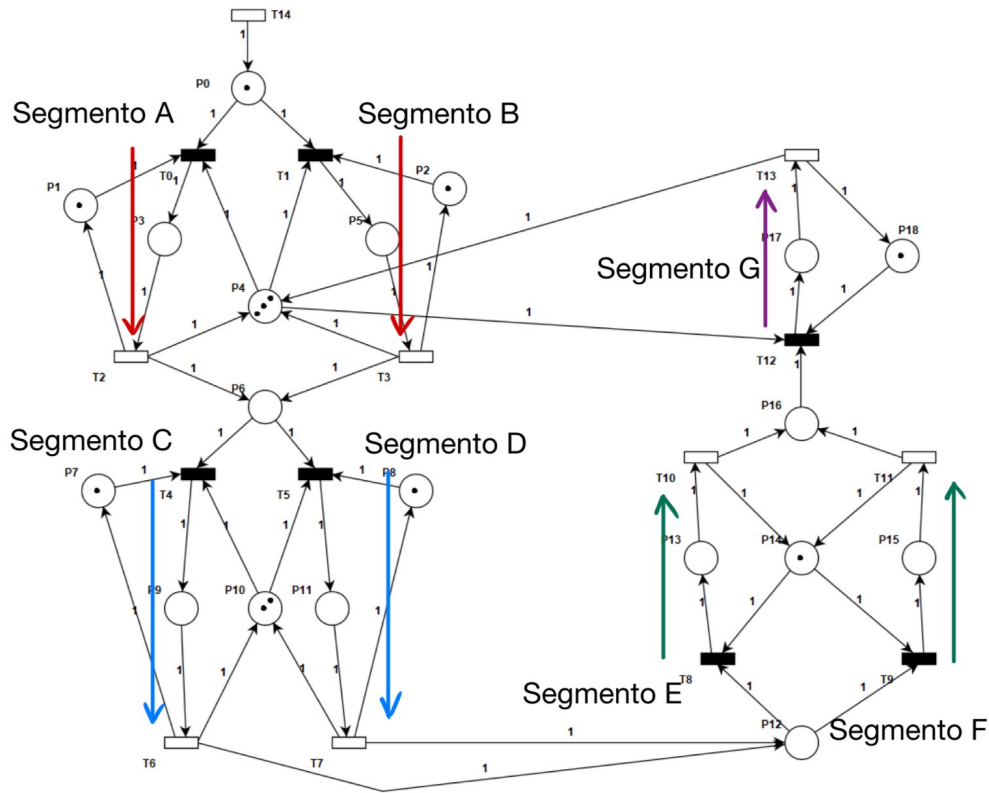


Figura 12: Segmentos de la red

El marcado máximo de cada segmento es la cantidad de hilos necesario para ese segmento, de esta forma, se tiene entonces que el sistema necesitaría **7**.

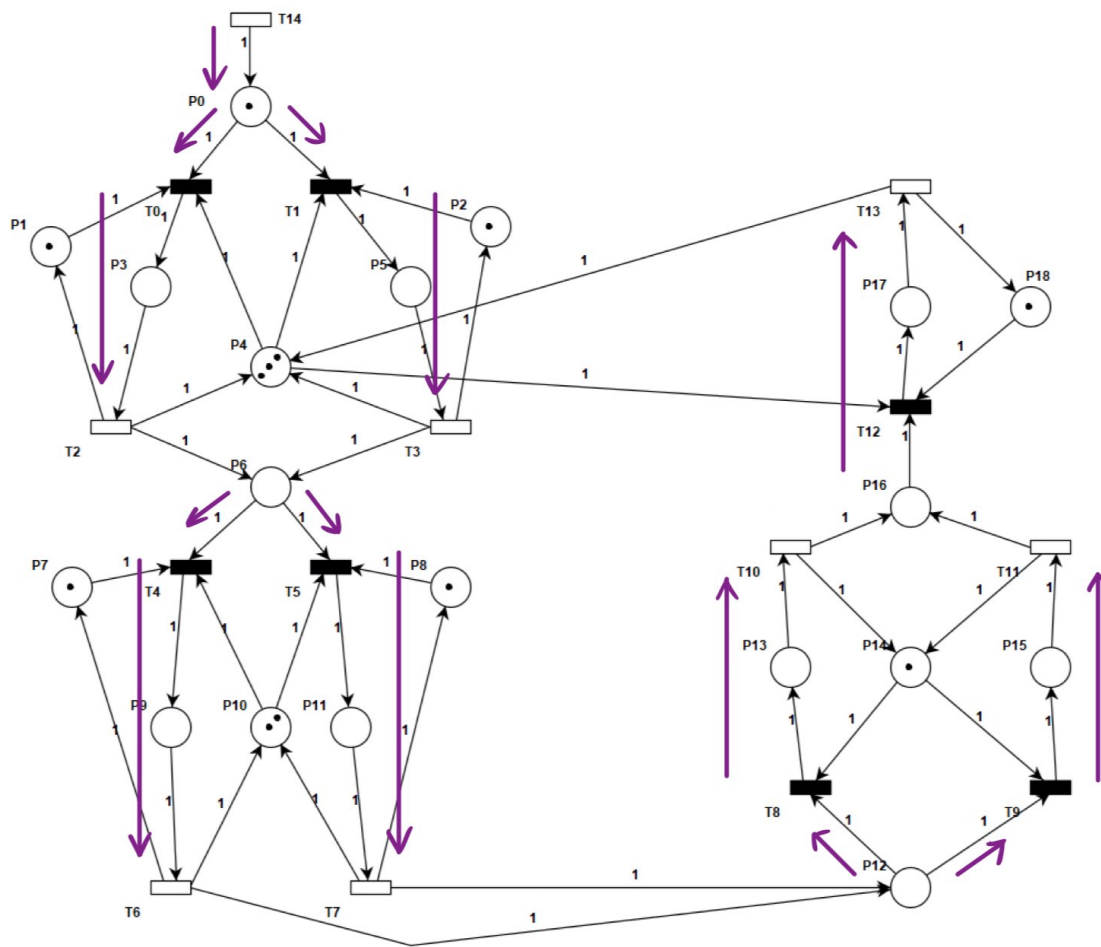
### 3.6.1. Caso 1

En el caso de presentarse conflictos o uniones, la responsabilidad de los hilos se separa en segmentos, pudiéndose decir así, que se tendría un segmento antes del conflicto y otros dos segmentos después. La ventaja de esta forma de análisis, es que se elimina la necesidad de solucionar el conflicto en el hilo, pasando a tomar protagonismo la política a implementar en el monitor.

Para la red de Petri del sistema modelado, el conflicto que aparece entre el buffer de imágenes cuya calidad fue mejorada, y los procesos recorte, se puede solucionar con un segmentado opuesto, agregando 2 hilos más al análisis anterior, que se encargue cada uno de disparar esas transiciones, llegando así a necesitar una cantidad máxima de hilos de **10**.

Se aplica el mismo criterio para los otros dos conflictos, de modo que en realidad, el total de hilos a utilizar en el modelo es de **14**.

Queda finalmente, conformado el sistema por los segmentos (hilos) a continuación.



### 3.6.2. Caso 2

---

*Proyecto Final Programación Concurrente*

## 4. Implementación

### 4.1. Parseo de la expresión regular

Para el parseo de la expresión regular se hizo un ejecutable en Python que se encarga de recibir un archivo de texto con la información de la secuencia de transiciones disparada, y con eso se encarga de verificar si matchea con la expresión regular obtenida de la red de Petri. El script original simplemente recibe la expresión de un archivo en el directorio en donde se encuentre el archivo de Python, de eso se adjunta el código a continuación:

Código 2: Script para parseo de expresión regular

```

1  import re
2
3  source = open('transitions.txt','r')
4  candidate = source.readlines()[0]
5  source.close()
6  output = open('output.txt','w')
7  output.write(candidate)
8  output.close()
9  while True:
10     f = open('output.txt','r')
11     transitions = f.readlines()[0]
12     f.close()
13     match = re.subn('((T12)(.?.*)(T13)(.?.*)((T10)(.?.*)(T1)(.?.*)((T3)(.?.*)(T5)|(T2)(.?.*)(
    ↪ T4))|(T9)(.?.*)(T0)(.?.*)((T3)(.?.*)(T5)|(T2)(.?.*)(T4)))|.?.*)(T6)(.?.*)(T7))|(T8)(.?.*)
    ↪ (T11)(.?.*)((T10)(.?.*)(T1)(.?.*)((T3)(.?.*)(T5)|(T2)(.?.*)(T4))|(T9)(.?.*)(T0)(.?.*)((T3
    ↪ ).?.*)(T5)|(T2)(.?.*)(T4)))|.?.*)(T6)(.?.*)(T7))','\g<3>\g<5>\g<8>\g<10>\g<13>\g
    ↪ <16>\g<19>\g<21>\g<24>\g<27>\g<29>\g<31>\g<34>\g<36>\g<39>\g<41>\g<44>\g
    ↪ <47>\g<50>\g<52>\g<55>\g<58>\g<60>\g<62>',transitions)
14     print(match)
15     f = open('output.txt','w')
16     f.write(match[0])
17     f.close()
18     if match[1] == 0:
19         print('FAIL: sobraron transiciones')
20         break
21     if match[0] == '':
22         print('SUCCESS, Test OK')
23         break

```

Para testear, se hace uso de una interfaz gráfica, implementada con la librería **PyQT6**. Se puede ver una ventana como la adjuntada a continuación.

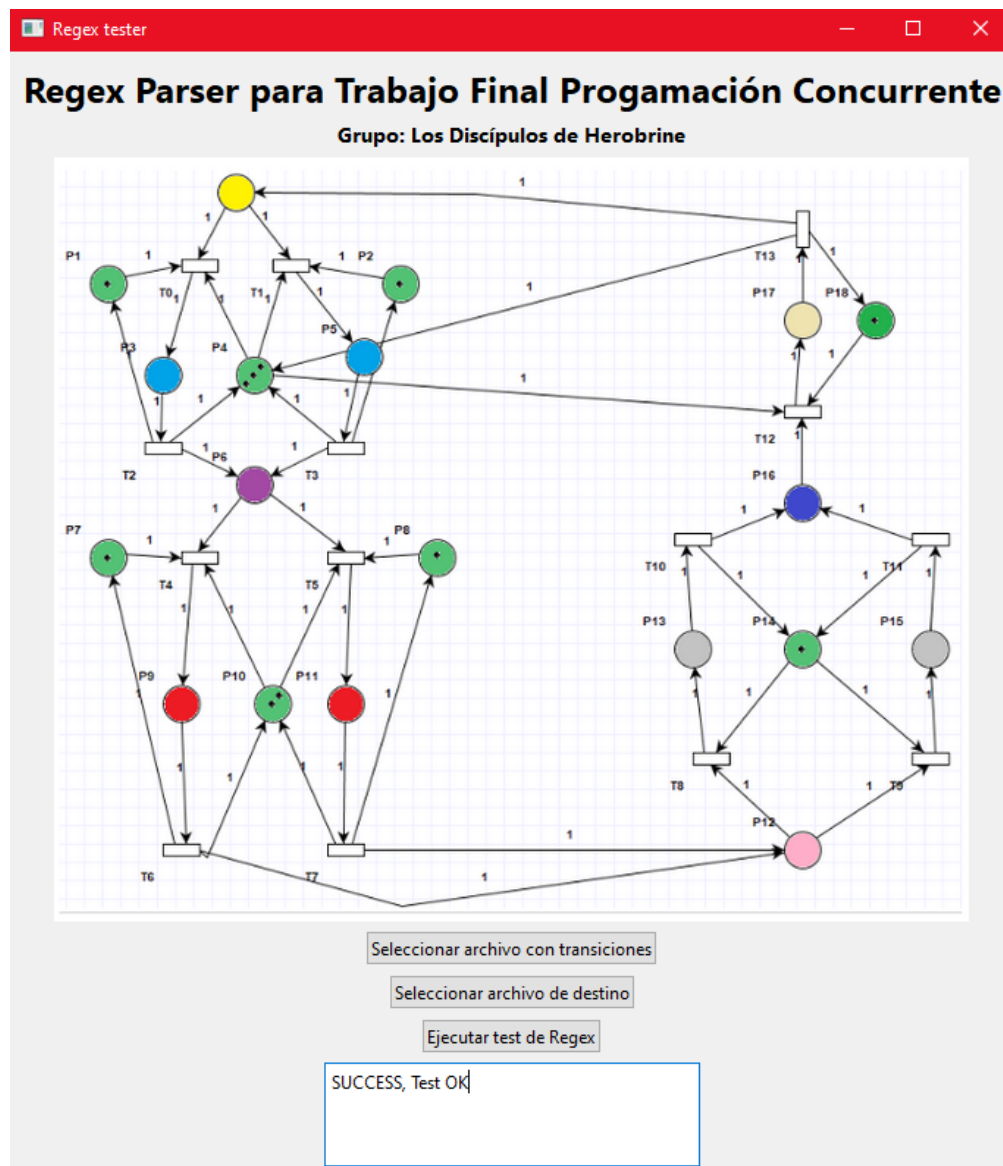


Figura 14: Interfaz gráfica del parser de la expresión regular

En ese script se envía un mensaje de éxito o fracaso, dependiendo de si la expresión regular matchea con la secuencia de transiciones disparadas.

## 4.2. Monitor de concurrencia

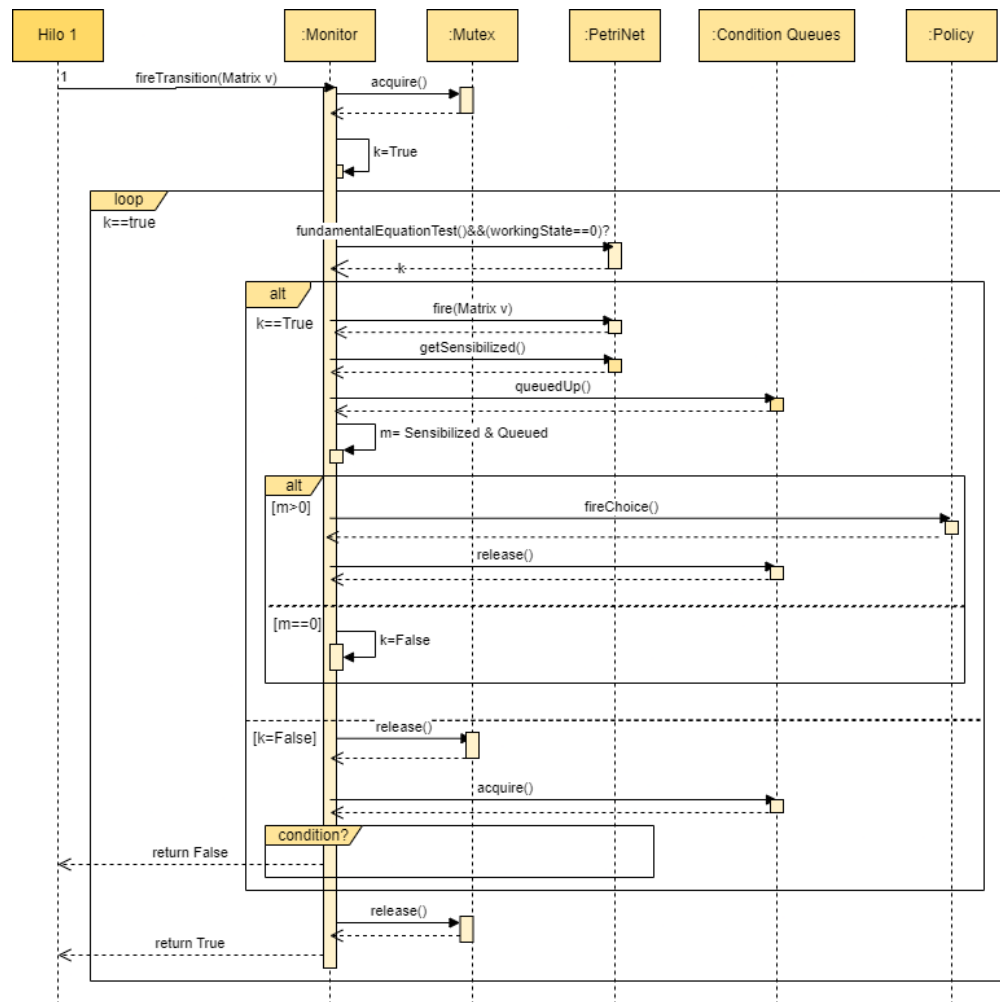


Figura 15: Monitor para la red de Petri no temporal

En un plano general, la idea sería que los distintos hilos estén todo el tiempo tratando de disparar la o las transiciones que le corresponden durante el ciclo de vida del programa, y luego el monitor gestione quién puede dispararlo. Para el caso del monitor de la red de petri sin tiempo, al entrar al gestor del monitor se toma el mutex, si esto se hace sin problemas se entra en un loop (**while(k)**) donde se pregunta si la transición está sensibilizada y ningún otro hilo está trabajando sobre esta transición (esto último es despreciable en esta instancia), en caso de no cumplirse se setea **k = false**. Ahora con **k = false** pasamos a soltar el mutex del monitor, mandar a dormir el hilo a su cola de condición y salir del monitor retornando True. En caso de que las condiciones se cumplan y K siga siendo True, se elige mediante la política entre las transiciones que estén sensibilizadas y encoladas en sus respectivas colas de condición, cuál disparar. Seguimos nuestro procedimiento soltando el mutex del monitor y retornando True.

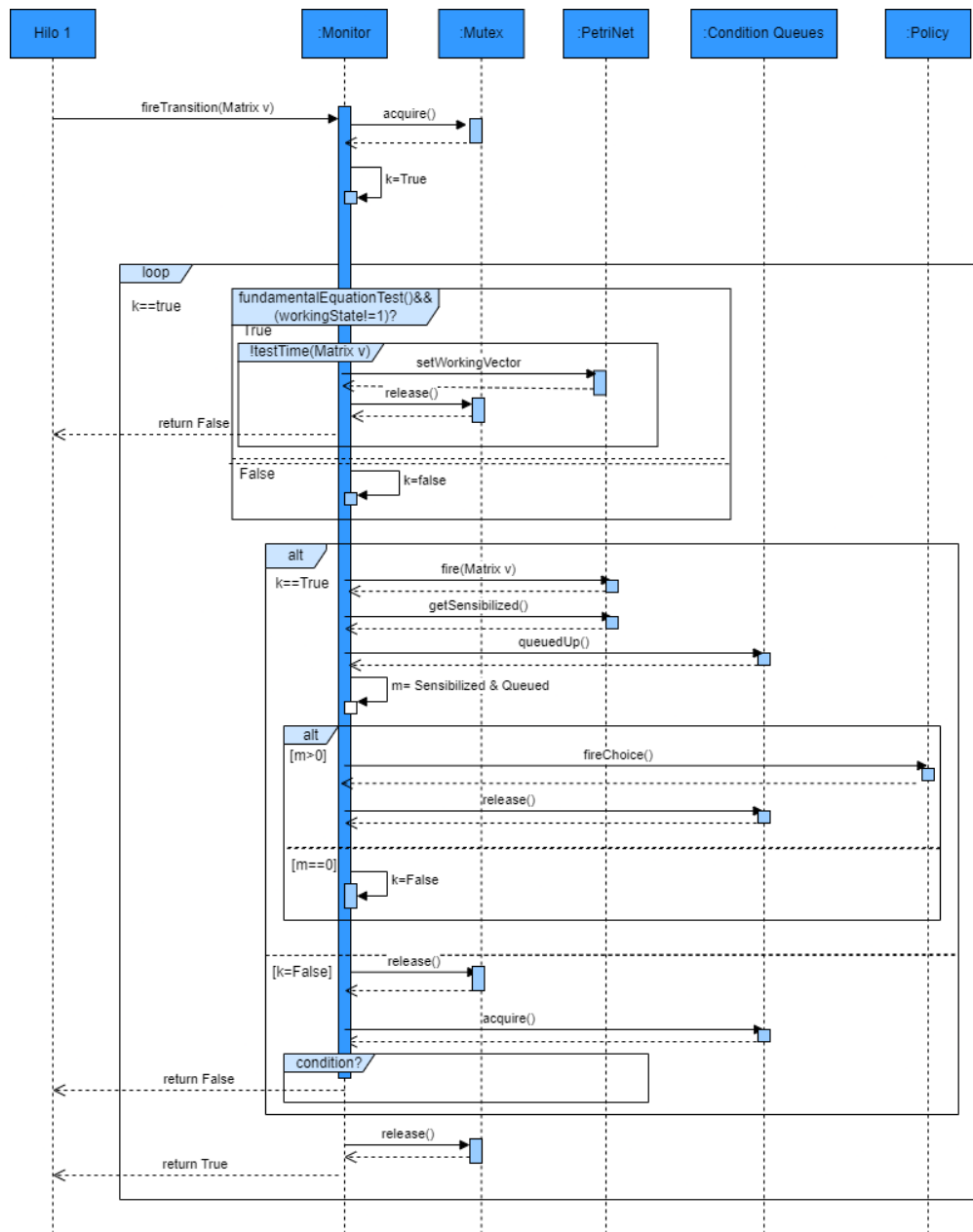


Figura 16: Monitor para la red de Petri no temporal

Para el caso del monitor para redes de Petri temporales, es muy similar pero se le agrega una condición extra. Es necesario recordar que para estas redes se tiene una mezcla entre las transiciones con delay y con tiempo, ya que se tiene un tiempo asociado a las transiciones y a su vez un intervalo. Por lo tanto se debe verificar que se cumpla el tiempo mínimo primero. Esta verificación se hace mediante el método **testTime()** que, de no cumplirse el tiempo mínimo  $\alpha$  desde que se sensibilizó la transición, retorna false y guarda en un vector el tiempo faltante. En el gestor del monitor, si la condición de tiempo no se cumple se suelta el mutex y se sale del monitor retornando false.

Código 3: Código del método testTime()

```
1 private boolean testTime(Matrix v) {  
2     long time = System.currentTimeMillis();  
3     long alpha = (long) petrinet.getAlphaTimes().get(0, getIndex(v));  
4     long initTime = (long) petrinet.getSensibilizedTime().get(0, getIndex(v));  
5     if (alpha < (time - initTime) || alpha == 0) {  
6         return true;  
7     } else {  
8         setTimeLeft(Thread.currentThread().getId(), alpha - (time - initTime));  
9         return false;  
10    }  
11 }
```

Los return del gestor del monitor son utilizados en el run de los hilos, ya que al recibir **true** actualizan la secuencia hacia la próxima transición posible de disparar, y al recibir 'false' manda a dormir al hilo el tiempo necesario para llegar al tiempo mínimo  $\alpha$ .

El tipo de monitor utilizado es **Signal And exit**, éste mueve los subprocesos que esperan una variable de condición a una cola de condición. El proceso señalizador sigue en el monitor hasta terminar y al salir llama al proceso que estaba esperando segun la politica seleccionada. En el codigo implementado, cuando se toma el monitor, el proceso que esta por disparar una transición, al salir del monitor, llama a la política y decide cual es el proceso que debe despertarse en el proximo paso, haciendo un release del semáforo que modela los lugares de la cola de condición, cuando un proceso quiere tomar el monitor, y no tiene lugar cuando entra, ejecuta un **acquire**, indicando que ocupo un lugar en la cola de condición.



### 4.3. Clases implementadas

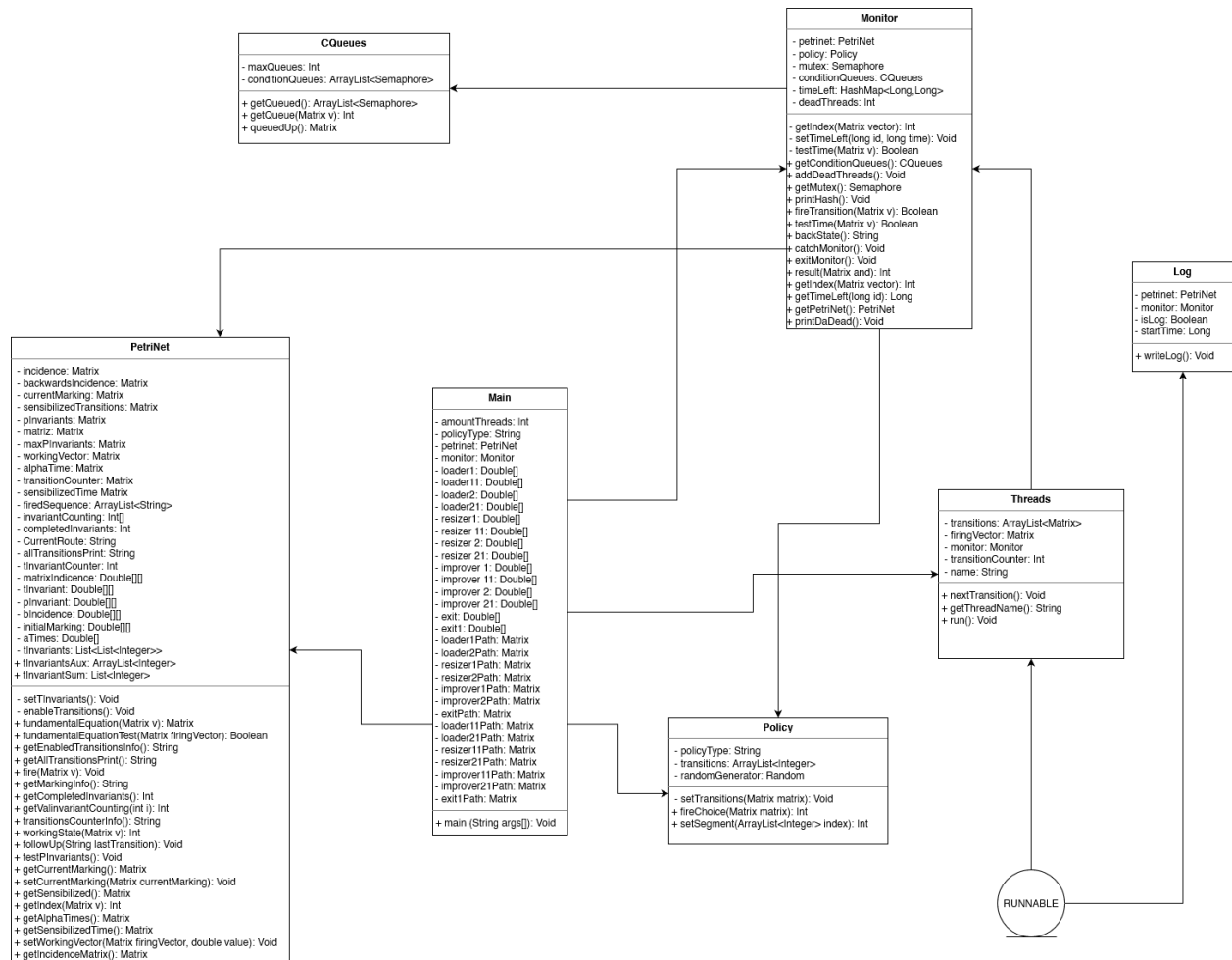


Figura 17: Diagrama de clases del sistema implementado

#### 4.3.1. Clase CQueues

Esta clase es la encargada de gestionar las colas de condición, que es a donde se van las transiciones que no cumplen con las condiciones. Se tiene una cola para cada transición, y posee métodos que permiten el manejo de las mismas, se tiene un ArrayList de semáforos (forma en que normalmente se implementa un monitor). Los métodos implementados son:

- **CQueues()**: constructor de la clase.
- **getQueued()**: devuelve las colas de condición.
- **getQueue(Matrix v)**: devuelve la cola para la transición correspondiente.
- **queuedUp()**: devuelve una matriz con las transiciones encoladas.

Código 4: Clase CQueues

```

1  import java.util.concurrent.Semaphore;
2  import Jama.Matrix;
3
4  public class CQueues {
5      private final int maxQueues = 14; // cambiar (num transiciones)
6      private ArrayList<Semaphore> conditionQueues;
7      public CQueues() {
8          conditionQueues = new ArrayList<Semaphore>();
9          for (int i = 0; i < maxQueues; i++)
10             conditionQueues.add(new Semaphore(0));
11     }
12     /*
13     * *****
14     * **** PUBLIC METHODS ****
15     * *****
16     */
17     /*
18     * *****
19     * *** getters y setters ***
20     * *****
21     */
22     public ArrayList<Semaphore> getQueued() {
23         return conditionQueues;
24     }
25     /*
26     * Returns the index of the first thread queued up for the transition associated with the
27     * ↪ vector v.
28     *
29     * @param v: firing vector
30     * @return the index of the first thread queued up for the transition associated with the
31     * ↪ vector v
32     */
33     public int getQueue(Matrix v) {
34         int index = 0;
35         for (int i = 0; i < v.getColumnDimension(); i++) {
36             if (v.get(0, i) == 1)
37                 break;
38             else
39                 index++;
40         }
41         return index;
42     }
43     /*
44     * Returns the number of threads queued up transitions.
45     */

```

```

45  * @param
46  * @return the number of threads queued up for the transition associated with
47  */
48  public Matrix queuedUp() {
49      double[] aux = new double[this.maxQueues];
50      for (Semaphore queue : conditionQueues) {
51          if (queue.hasQueuedThreads())
52              aux[conditionQueues.indexOf(queue)] = 1;
53          else
54              aux[conditionQueues.indexOf(queue)] = 0;
55      }
56      Matrix waitingThreads = new Matrix(aux, 1);
57      return waitingThreads;
58  }
59  }

```

### 4.3.2. Main

Esta clase es la encargada de ejecutar el programa principal, en donde se instancia una política, un monitor, una red de Petri, y los diferentes hilos que fueron calculados anteriormente. Cada uno de los hilos instanciados tiene una matriz que tiene la transición que representa, para de esa forma chequear durante la ejecución del programa las sensibilizaciones.

Se instancia también un objeto **Log** que brinda dos salidas por archivos de texto:

- Una incluye la secuencia de disparos de la red.
- Y la otra muestra estadísticas de la ejecución realizada.

Código 5: Clase Main

```

1  import Jama.Matrix;
2
3  public class Main {
4      private static final int amountThreads = 14;
5      private static final String policyType = "Equitative";
6      private static PetriNet petrinet; // Red de petri representativa del sistema.
7      private static Monitor monitor; // Monitor que controlará la red de Petri que modela el
      ↪ sistema.
8      private static double[] loader1 = { 0 };
9      private static double[] loader11 = { 2 };
10     // 1 y 3
11     private static double[] loader2 = { 1 };
12     private static double[] loader21 = { 3 };
13     // 4 y 6
14     private static double[] resizer1 = { 4 };
15     private static double[] resizer11 = { 6 };

```

```

16 // 5 y 7
17 private static double[] resizer2 = { 5 };
18 private static double[] resizer21 = { 7 };
19 // 8 y 10
20 private static double[] improver1 = { 8 };
21 private static double[] improver11 = { 10 };
22 // 9 y 11
23 private static double[] improver2 = { 9 };
24 private static double[] improver21 = { 11 };
25 // 12 y 13
26 private static double[] exit = { 12 };
27 private static double[] exit1 = { 13 };
28 // matrices con transiciones asociadas a los hilos
29 private static Matrix loader1Path = new Matrix(loader1, 1); // version traspuesta de P0
30 private static Matrix loader2Path = new Matrix(loader2, 1); // version traspuesta de P1
31 private static Matrix resizer1Path = new Matrix(resizer1, 1); // version traspuesta de
32 ↪ P2
33 private static Matrix resizer2Path = new Matrix(resizer2, 1); // version traspuesta de
34 ↪ P3
35 private static Matrix improver1Path = new Matrix(improver1, 1); // version traspuesta
36 ↪ de P4
37 private static Matrix improver2Path = new Matrix(improver2, 1); // version traspuesta
38 ↪ de P5
39 private static Matrix exitPath = new Matrix(exit, 1); // version traspuesta de P6
40 private static Matrix loader11Path = new Matrix(loader11, 1); // version traspuesta de
41 ↪ P0
42 private static Matrix loader21Path = new Matrix(loader21, 1); // version traspuesta de
43 ↪ P1
44 private static Matrix resizer11Path = new Matrix(resizer11, 1); // version traspuesta de
45 ↪ P2
46 private static Matrix resizer21Path = new Matrix(resizer21, 1); // version traspuesta de
47 ↪ P3
48 private static Matrix improver11Path = new Matrix(improver11, 1); // version
49 ↪ traspuesta de P4
50 private static Matrix improver21Path = new Matrix(improver21, 1); // version
51 ↪ traspuesta de P5
52 private static Matrix exit1Path = new Matrix(exit1, 1); // version traspuesta de P6
53 /**
54  * Método principal.
55  *
56  * Aquí se instancian y ejecutan los hilos con sus caminos asociados.
57  * También se inicializan tanto la red de Petri con su marcado inicial
58  * como el monitor y el hilo logger.
59  */
60 //
61 public static void main(String args[]) {
62     long start = System.currentTimeMillis();

```

```

53     petrinet = new PetriNet();
54     Policy policy = new Policy(policyType);
55     System.out.println("Policy type: " + policyType + " \n");
56     monitor = new Monitor(petrinet, policy);
57     Threads[] threads = new Threads[amountThreads];
58     petrinet.enableTransitions();
59     try {
60         long startTime = System.currentTimeMillis();
61         Log log = new Log(petrinet, monitor, startTime, true);
62         log.start();
63
64         Log transition = new Log(petrinet, monitor, startTime, false);
65         transition.start();
66     } catch (Exception e) {
67         e.printStackTrace();
68         System.out.println("Error creating logger.");
69     }
70     threads[0] = new Threads(loader1Path, monitor, "Loader 1");
71     threads[1] = new Threads(loader2Path, monitor, "Loader 2");
72     threads[2] = new Threads(resizer1Path, monitor, "Resizer 1");
73     threads[3] = new Threads(resizer2Path, monitor, "Resizer 2");
74     threads[4] = new Threads(improver1Path, monitor, "Improver 1");
75     threads[5] = new Threads(improver2Path, monitor, "Improver 2");
76     threads[6] = new Threads(exitPath, monitor, "Exit");
77     threads[7] = new Threads(loader11Path, monitor, "Loader 1.1");
78     threads[8] = new Threads(loader21Path, monitor, "Loader 2.1");
79     threads[9] = new Threads(resizer11Path, monitor, "Resizer 1.1");
80     threads[10] = new Threads(resizer21Path, monitor, "Resizer 2.1");
81     threads[11] = new Threads(improver11Path, monitor, "Improver 1.1");
82     threads[12] = new Threads(improver21Path, monitor, "Improver 2.1");
83     threads[13] = new Threads(exit1Path, monitor, "Exit1");
84     for (Threads thread : threads) {
85         thread.start();
86     }
87     monitor.printDeadThreads();
88     System.out.println(petrinet.transitionsCounterInfo());
89 }
90 }

```

### 4.3.3. Policy

La clase policy es la encargada de “decidir” qué acción tomar ante un conflicto estructural en la red de petri.

Las transiciones involucradas son  $T_0$ ,  $T_1$ ,  $T_4$ ,  $T_6$ ,  $T_8$ ,  $T_9$ , al momento de llegar a decidir en el monitor a cual de los hilos correspondientes a las transiciones le toca despertar, la clase policy entra en juego, decidiendo por medio de variables generadas aleatoriamente, la elegida

para ser disparada.

En el caso de encontrarse con el conflicto entre  $T_8$  y  $T_9$ , la política carga 80 % de las decisiones sobre la transición  $T_8$ , haciendo que esta en el resultado de las transiciones disparadas quede en un desequilibrio con respecto al resto de los conflictos donde se mantienen con un 50 % y 50 %.

En el caso que se quiera decidir de manera equitativa, es decir, 50/50, se puede colocar como parámetro del constructor de la clase el string correspondiente:

- Para 50/50 = “Equitativa”.
- Para 80/20 = “8020”.

El método utilizado desde la clase monitor para decidir qué proceso despertar es **fireChoice(Matrix matrix)**, siendo “matrix” la matriz de transiciones habilitadas o sensibilizadas para realizar un disparo.

Para construir una instancia de esta clase (única), se utiliza el constructor **policy(String policyType)**.

Código 6: Clase Policy

```

1  import Jama.Matrix;
2  import java.util.ArrayList;
3  import java.util.Random;
4  import java.lang.Math;
5
6  public class Policy {
7      private String policyType;
8      private ArrayList<Integer> transitions;
9      private Random randomGenerator;
10
11     public Policy(String policyType) {
12
13         this.policyType = policyType;
14         this.transitions = new ArrayList<>();
15         randomGenerator = new Random();
16     }
17     // segment = matrix with queued and sensibilized transitions
18     /*
19     * Segment 1: T0 T1
20     * Segment 2: T4 T5
21     * Segment 3: T8 T9
22     */
23     public int fireChoice(Matrix matrix) {
24         int indexChosen = 0;
25         if (policyType == "8020") { // asks the policy type
26
27             if (matrix.get(0, 8) >= 1 || matrix.get(0, 9) >= 1) { // if transitions 8 or 9 are
                ↪ enabled, changes the

```

```

28         // probability to 80-20
29         double probabilidadT8 = 0.8;
30         // generates a random number between 0 and 1
31         double numeroAleatorio = randomGenerator.nextDouble();
32         if (numeroAleatorio < probabilidadT8) {
33             indexChosen = 8; // Chose index 8
34             System.out.println("Ocurrió la transición T8");
35         } else {
36             indexChosen = 9; // Chose index 9
37             System.out.println("Ocurrió la transición T9");
38         }
39     } else {
40         // If the sensibilized transitions are not T8 or T9, chooses randomly
41         transitions.clear(); // clear array
42
43         for (int i = 0; i < matrix.getColumnDimension(); i++)
44             if (matrix.get(0, i) > 0)
45                 transitions.add(i);
46
47         int choice = (int) Math.round(randomGenerator.nextInt(transitions.size()));
48         indexChosen = (int) Math.round(transitions.get(choice));
49     }
50 } else if (this.policyType == "Equitative") {
51     // if policy type is equitative chooses randomly with a normal distribution of
52     ↪ probabilities // red
53     transitions.clear(); // Clears array
54
55     for (int i = 0; i < matrix.getColumnDimension(); i++)
56         if (matrix.get(0, i) > 0)
57             transitions.add(i);
58     int choice = (int) Math.round(randomGenerator.nextInt(transitions.size()));
59     indexChosen = (int) Math.round(transitions.get(choice));
60 }
61 return indexChosen;
62 }
63
64 private void setTransitions(Matrix matrix) {
65     this.transitions.clear(); // Clears array.
66
67     for (int i = 0; i < matrix.getColumnDimension(); i++)
68         if (matrix.get(0, i) > 0)
69             transitions.add(i);
70 }
71 }

```

#### 4.3.4. Monitor

Esta clase gestiona el disparo de las transiciones mediante el gestor del monitor (método `fireTransition(Matrix v)`). Su funcionamiento puede verse más a detalle en el diagrama de secuencias. Los métodos más relevantes son:

- `catchMonitor()`: toma el mutex del monitor.
- `exitMonitor()`: libera el mutex del monitor.
- `fireTransition(Matrix v)`: gestor del monitor. Este método verifica las transiciones sensibilizadas por token y por tiempo. Esto último se hace llamando el método `testTime`. En caso de todas las condiciones se den, transición completamente sensibilizada y encolada, se dispara la transición y el método retorna true, caso contrario retorna false.
- `testTime(Matrix v)`: en este método se verifica que la transición esté sensibilizada temporalmente. Esto se hace chequeando que el tiempo que pasó desde que la transición se sensibilizó (por tokens) es mayor al tiempo  $\alpha$ . En caso de que la transición esté sensibilizada temporalmente, retorna true, caso contrario guarda en el vector “leftTime” el tiempo faltante y retorna false.
- `result()`: retorna la cantidad de hilos sensibilizados y encolados.

Código 7: Clase Monitor

```
1  import java.util.HashMap;
2  import java.util.concurrent.Semaphore;
3  import Jama.Matrix;
4
5  public class Monitor {
6
7      private PetriNet petrinet;
8      private Policy policy;
9      private Semaphore mutex;
10     private CQueues conditionQueues;
11     private HashMap<Long, Long> timeLeft;
12
13     private int deadThreads;
14
15     public Monitor(PetriNet petrinet, Policy policy) {
16         this.conditionQueues = new CQueues();
17         this.petrinet = petrinet;
18         this.policy = policy;
19         this.timeLeft = new HashMap<Long, Long>();
20         this.mutex = new Semaphore(1, true);
21         this.deadThreads = 0;
22     }
```



```

23
24     public CQueues getConditionQueues() {
25         return conditionQueues;
26     }
27
28     public void addDeadThreads() {
29         this.deadThreads++;
30     }
31
32     public Semaphore getMutex() {
33         return mutex;
34     }
35
36     /*
37     * *****
38     * *** PRINCIPAL METHOD ***
39     * *****
40     */
41
42     /*
43     * The method checks if the transition is enabled an "time enabled" calling the
44     ↪ 'testTime' method.
45     * In case all conditions are met, the transition is fired and the method returns true.
46     * Otherwise, the thread is queued up and the method returns false.
47     *
48     * @param v: firing vector
49     * @return true if the transition is fired, false otherwise
50     */
51     public boolean fireTransition(Matrix v) {
52         // printHash();
53         try {
54             if (petrinet.getCompletedInvariants() < 200) {
55                 catchMonitor();
56             } else {
57                 return false;
58             }
59         } catch (InterruptedException e) {
60             e.printStackTrace();
61         }
62         // -----
63         boolean k = true;
64         while (k) {
65             if (petrinet.fundamentalEquationTest(v)
66                 && ((petrinet.workingState(v) == 0) || (petrinet.workingState(v) == 2))) {
67                 if (testTime(v)) {
68                     k = true;
69                 } else {

```

```

69         petrinet.setWorkingVector(v, (double) Thread.currentThread().getId());
70         k = false;
71         exitMonitor();
72         return false;
73     }
74 } else {
75     k = false;
76 }
77 if (k) {
78     petrinet.fire(v);
79     Matrix sensibilized = petrinet.getSensibilized();
80     // sensibilized.print(0,2);
81     Matrix queued = conditionQueues.queuedUp();
82     Matrix and = sensibilized.arrayTimes(queued); // 'and' '&'
83     int m = result(and); // sensibilized and queued transitions
84     if (m > 0) // queued and enabled transitions
85     {
86         // cual
87         int choice = policy.fireChoice(and);
88         // release
89         conditionQueues.getQueued().get(choice).release();
90
91         System.out.println("Thread ID: " + Thread.currentThread().getId() + " wakes
92 ↪ up");
93     } else // there's none transition enabled and queued
94     {
95         System.out.println("k turns to false");
96         k = false;
97     }
98 } else {
99     exitMonitor();
100    int queue = conditionQueues.getQueue(v);
101    try {
102        conditionQueues.getQueued().get(queue).acquire();
103        if (petrinet.getCompletedInvariants() >= 200)
104            return false;
105    } catch (InterruptedException e) {
106        e.printStackTrace();
107    }
108 }
109 }
110 exitMonitor();
111 return true;
112 }
113
114 /*

```

```

115  * Test if the transition is "time enabled". Checking if the elapsed time since
116  * the sensibilization of the transition is greater than the alpha time.
117  * Returns true if the transition is "time enabled", if it's not, returns false and save's the
    ↪ remaining time.
118  *
119  * @param v: firing vector
120  * @return true if the transition is "time enabled", false otherwise
121  */
122  private boolean testTime(Matrix v)
123  {
124      long time = System.currentTimeMillis();
125      long alpha = (long) petrinet.getAlphaTimes().get(0, getIndex(v));
126      long initTime = (long) petrinet.getSensibilizedTime().get(0, getIndex(v));
127      if (alpha < (time - initTime) || alpha == 0) {
128          return true;
129      } else {
130          setTimeLeft(Thread.currentThread().getId(), alpha - (time - initTime));
131          return false;
132      }
133  }
134
135  /*
136  * *****
137  * **** PUBLIC METHODS ****
138  * *****
139  */
140
141  public void printDeadThreads()
142  {
143      System.out.println("Dead threads: " + deadThreads + "/14");
144  }
145
146  public void catchMonitor() throws InterruptedException {
147      mutex.acquire();
148  }
149
150  public void exitMonitor() {
151      mutex.release();
152  }
153
154  /*
155  * Returns the number of enabled and queued transitions.
156  *
157  * @param and: matrix resulting from the 'and' operation between the sensibilized and
    ↪ queued transitions.
158  * @return the number of enabled and queued transitions.
159  */

```

```
160 public int result(Matrix and)
161 {
162     int m = 0;
163     for (int i = 0; i < and.getColumnDimension(); i++)
164         if (and.get(0, i) > 0)
165             m++;
166     return m;
167 }
168
169 /*
170  * Returns the index of the transition that is going to be fired.
171  *
172  * @param v: firing vector
173  * @return index of the transition
174  */
175 private int getIndex(Matrix vector) {
176     int index = 0;
177
178     for (int i = 0; i < vector.getColumnDimension(); i++) {
179         if (vector.get(0, i) == 1)
180             break;
181         else
182             index++;
183     }
184
185     return index;
186 }
187
188 /*
189  * *****
190  * *** Getters & Setters ***
191  * *****
192  */
193 public HashMap<Long, Long> getTimeLeftMap() {
194     return this.timeLeft;
195 }
196
197 public synchronized long getTimeLeft(long id) {
198     return this.timeLeft.get(id);
199 }
200
201 private synchronized void setTimeLeft(long id, long time) {
202     timeLeft.put(id, time);
203 }
204
205 public PetriNet getPetriNet() {
206     return this.petrinet;
```

```

207     }
208 }

```

### 4.3.5. Threads

Esta clase está compuesta por 2 métodos públicos, además de un método run y su constructor. El main creará instancias de esta clase, y luego le les asignará el nombre, el monitor y la secuencia de transiciones. Una vez asignados, se inician todos.

Cuando entra al run, se imprimirá por pantalla el nombre del hilo, dando aviso que inició y procede a verificar que los invariantes de transición no hayan superado los 200 disparos (en el caso propuesto). Mientras esto no se supere, se intentará disparar la transición correspondiente. Si el monitor me lo permite, se dispara, caso contrario, se le asigna un tiempo de sleep, el cuál se determinará con un método de la clase Monitor, pasando el id del hilo como parámetro. Se intenta asignar el delay al hilo con un try catch. Una vez que se completan los 200 disparos, se incrementa el contador de hilos finalizados, luego se imprime por pantalla avisando qué hilo en particular finalizó, y desde el monitor, avisa cuántos finalizaron del total.

Hay otros 2 métodos, entonces, se tiene un getter del nombre del hilo, y el otro cambia el número de la próxima transición. Si esta llega a su límite, vuelve a 0.

Código 8: Clase Threads

```

1  import java.util.ArrayList;
2  import Jama.Matrix;
3  import java.util.concurrent.TimeUnit;
4
5  public class Threads extends Thread {
6
7      private ArrayList<Matrix> transitions;
8      private Matrix firingVector;
9      private Monitor monitor;
10     private int transitionCounter;
11     private String name;
12
13
14     public Threads(Matrix transitionsSequence, Monitor monitor, String name)
15     {
16         this.transitions = new ArrayList<Matrix>();
17         this.name = name;
18         transitionsSequence.print(2,0);
19
20         for (int i = 0; i < transitionsSequence.getColumnDimension(); i++)
21         {
22             int index = (int) transitionsSequence.get(0, i);
23             Matrix aux = new Matrix(1,
24             ↪ monitor.getPetriNet().getIncidenceMatrix().getColumnDimension());

```

```
24     aux.set(0, index, 1);
25     this.transitions.add(aux);
26 }
27 this.monitor = monitor;
28 this.transitionCounter = 0;
29 }
30
31 public void nextTransition()
32 {
33     this.transitionCounter++;
34     if (transitionCounter >= transitions.size())
35     {
36         this.transitionCounter = 0;
37     }
38 }
39
40 public String getThreadName() {
41     return this.name;
42 }
43
44 @Override
45 public void run()
46 {
47     //System.out.println(Thread.currentThread().getId() + ": started run()");
48     System.out.println("Thread " + getThreadName() + ": started run()");
49     while (this.monitor.getPetriNet().getCompletedInvariants() < 200)
50     {
51         this.firingVector = transitions.get(transitionCounter);
52
53         //System.out.println(getThreadName());
54
55         //firingVector.print(2,0);
56         if (monitor.fireTransition(firingVector))
57         {
58             nextTransition();
59         }else
60         {
61             long sleepTime;
62             try {
63                 sleepTime = this.monitor.getTimeLeft(Thread.currentThread().getId());
64             }
65             catch (Exception e) {
66                 sleepTime = 0;
67             }
68             if(!(this.monitor.getPetriNet().getCompletedInvariants() < 200))
69             {
70                 try {
```

```
71         TimeUnit.MILLISECONDS.sleep(sleepTime);
72     } catch (Exception e) {
73         e.printStackTrace();
74     }
75
76     }
77 }
78 }
79 this.monitor.addDeadThreads();
80 //System.out.println(Thread.currentThread().getId() + ": finished run()");
81 System.out.println("Thread " + getThreadName() + ": finished run()");
82 this.monitor.printDeadThreads();
83 }
84 }
```

#### 4.3.6. PetriNet

Se tiene la implementación de la red de Petri modelada. Es la clase más importante, por lo que se explicará con más detalle.

- Realiza un seguimiento del tiempo de activación y sensibilización de las transiciones.
- Maneja la concurrencia al verificar el estado de trabajo de las transiciones antes de dispararlas.
- Simula y monitorea la dinámica de una Red de Petri, verificando invariantes y registrando la secuencia de transiciones disparadas.

Se tienen las siguientes estructuras de datos:

- **incidence**: matriz de incidencia que representa la relación entre transiciones y plazas.
- **backwardsIncidence**: matriz de incidencia invertida.
- **currentMarking**: vector de marcado actual.
- **sensibilizedTransitions**: vector de transiciones sensibilizadas.
- **pInvariants**: matriz de invariantes de plaza.
- **matriz**: matriz auxiliar para usar en el proceso de disparo de transiciones.
- **workingVector**: vector que representa el trabajo en curso para cada transición.
- **alphaTime**: vector que almacena el tiempo de activación de las transiciones.
- **transitionCounter**: matriz que cuenta la cantidad de veces que se ha disparado cada transición.

- **sensibilizedTime**: vector que almacena el tiempo de sensibilización de las transiciones.
- **tInvariantsAux**: lista auxiliar para los invariantes de transición.
- **firedSequence**: lista que almacena la secuencia de transiciones disparadas.

Como variables y constantes:

- **invariantCounting**: arreglo que cuenta la cantidad de veces que se cumplen ciertos invariantes.
- **completedInvariants**: contador de invariantes completados.
- **CurrentRoute**: cadena que representa la secuencia actual de transiciones disparadas.
- **allTransitionsPrint**: cadena que almacena información sobre todas las transiciones.

Los métodos principales son:

- **Constructor (PetriNet)**: inicializa las matrices y vectores necesarios. Establece los invariantes de transición.
- **fundamentalEquation(Matrix v)**: este método calcula la ecuación fundamental de la red de Petri:  $m_{i+1} = m_i + W * s$ . Donde:  $m_i$  es la marca actual,  $W$  es la matriz de incidencia y  $s$  es el vector de disparo.
- **fundamentalEquationTest(Matrix firingVector)**: verifica si la ecuación fundamental es válida.
- **enableTransitions()**: determina las transiciones sensibilizadas. Compara el marcado actual con el marcado solicitado para cada transición.
- **fire(Matrix v)**: este método activa una transición si está habilitado. Cambia la marca actual, actualiza el vector de trabajo actual, actualiza transiciones sensibilizadas y agrega a la secuencia disparada la transición disparada.
- **followUp(String lastTransition)**: realiza un seguimiento de las transiciones disparadas para verificar los invariantes de transición.
- **getEnabledTransitionsInfo()**: obtiene información sobre las transiciones habilitadas.
- **getAllTransitionsPrint()**: obtiene información sobre todas las transiciones.
- **getMarkingInfo()**: obtiene información sobre el marcado actual.
- **transitionsCounterInfo()**: obtiene información sobre la cantidad de veces que se ha disparado cada transición.
- **getCompletedInvariants()**: obtiene el número de invariantes completados.



- **getValinvariantCounting(int i)**: obtiene el valor del contador de invariantes para un índice dado.

Código 9: Clase PetriNet

```

1  import java.util.ArrayList;
2  import java.util.Arrays;
3  import java.util.List;
4  import Jama.Matrix;
5
6  public class PetriNet {
7
8      private Matrix incidence;
9      private Matrix backwardsIncidence;
10     private Matrix currentMarking;
11     private Matrix sensibilizedTransitions; // vector de transiciones sensibilizadas
12     private Matrix pInvariants;
13     private Matrix matriz;
14     private Matrix maxPInvariants;
15     private Matrix workingVector;
16     private Matrix alphaTime;
17     private Matrix transitionCounter;
18     private Matrix sensibilizedTime;
19     public ArrayList<Integer> tInvariantsAux;
20     private ArrayList<String> firedSequence;
21     private int[] invariantCounting;
22     private static int completedInvariants;
23     private String CurrentRoute;
24     private String allTransitionsPrint;
25
26     private int tInvariantCounter;
27
28     private final double[][] matrixIndicence = {
29         // T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13
30         { -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
31         { -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
32         { 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
33         { 1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
34         { -1, -1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, -1, 1 },
35         { 0, 1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
36         { 0, 0, 1, 1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0 },
37         { 0, 0, 0, 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0 },
38         { 0, 0, 0, 0, 0, -1, 0, 1, 0, 0, 0, 0, 0, 0 },
39         { 0, 0, 0, 0, 1, 0, -1, 0, 0, 0, 0, 0, 0, 0 },
40         { 0, 0, 0, 0, -1, -1, 1, 1, 0, 0, 0, 0, 0, 0 },
41         { 0, 0, 0, 0, 0, 1, 0, -1, 0, 0, 0, 0, 0, 0 },
42         { 0, 0, 0, 0, 0, 0, 1, 1, -1, -1, 0, 0, 0, 0 },

```

```

43     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1, 0, 0, 0 },
44     { 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 1, 1, 0, 0 },
45     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1, 0, 0 },
46     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, -1, 0 },
47     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1 },
48     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 1 }
49 };
50
51 private final double[][] tInvariant = {
52     { 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1 }, // T1 T3 T5 T7 T9 T11 T12 T13
53     { 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1 }, // T1 T3 T5 T7 T8 T10 T12 T13
54     { 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1 }, // T1 T3 T4 T6 T9 T11 T12 T13
55     { 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1 }, // T1 T3 T4 T6 T8 T10 T12 T13
56     { 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1 }, // T0 T2 T5 T7 T9 T11 T12 T13
57     { 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1 }, // T0 T2 T5 T7 T8 T10 T12 T13
58     { 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1 }, // T0 T2 T4 T6 T9 T11 T12 T13
59     { 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1 }, // T0 T2 T4 T6 T8 T10 T12 T13
60 };
61 private final double[][] pInvariant = {
62     // 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
63     { 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0 }, // 1 0,3,5,6,9,11,12,13,15,16,17
64     { 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 2 1,3,
65     { 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 3 2,5
66     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0 }, // 4 13,14,15
67     { 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 }, // 5 7,9
68     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0 }, // 6 8,11-----
69     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 }, // 7 9,10,11
70     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 }, // 8 17,18-----
71     { 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 }, // 9 3,4,5,17
72 };
73
74 private final double[][] bIncidence = {
75     // T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13
76     { 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
77     { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
78     { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
79     { 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
80     { 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },
81     { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
82     { 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 },
83     { 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
84     { 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 },
85     { 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 },
86     { 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 },
87     { 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 },
88     { 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 },
89     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 },

```

```

90      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 },
91      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },
92      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },
93      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
94      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 }
95  };
96  // 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
97  private final double[] initialMarking = { 1, 1, 1, 0, 3, 0, 0, 1, 1, 0, 2, 0, 0, 0, 1, 0, 0, 0, 1
↪ };
98
99  // 0 1 2 3 4 5 6 7 8 9 10 11 12 13
100 private final double[] aTimes = { 0, 0, 5, 5, 0, 0, 10, 10, 0, 0, 10, 10, 0, 5 };
101
102 public PetriNet() {
103     this.CurrentRoute = "";
104     this.completedInvariants = 0;
105     this.invariantCounting = new int[8];
106     this.incidence = new Matrix(matrixIndicence);
107     this.backwardsIncidence = new Matrix(bIncidence);
108     this.currentMarking = new Matrix(initialMarking, 1);
109     this.sensibilizedTransitions = new Matrix(1, incidence.getColumnDimension());
110     this.pInvariants = new Matrix(pInvariant);
111     this.maxPInvariants = new Matrix(incidence.getRowDimension(), 1); // esto esta
↪ mal creo, no hay que ver el la
112                                     // incidencia
113     this.workingVector = new Matrix(1, incidence.getColumnDimension());
114     this.alphaTime = new Matrix(aTimes, 1);
115     this.sensibilizedTime = new Matrix(1, incidence.getColumnDimension());
116     this.firedSequence = new ArrayList<String>();
117     this.transitionCounter = new Matrix(1, 14);
118     for (int j = 0; j < 14; j++) {
119         transitionCounter.set(0, j, 0.0);
120     }
121     this.tInvariantCounter = 0;
122 }
123 /*
124 * *****
125 * Public Methods *
126 * *****
127 */
128
129 /*
130 * This method calculates the fundamental ecuation of the petrinet:  $m_i+1 = m_i + W*s$ 
131 * where  $m_i$  is the current marking,  $W$  is the incidence matrix and  $s$  is the firing vector.
132 *
133 * @param v: firing vector
134 * @return fundamental equation

```

```

135  */
136  public Matrix fundamentalEquation(Matrix v) {
137      return (currentMarking.transpose().plus(incidence.times(v.transpose()))).transpose();
138      // (mi + w * s) transpose
139  }
140
141  public boolean fundamentalEquationTest(Matrix firingVector) {
142      matriz = fundamentalEquation(firingVector);
143      for (int i = 0; i < this.matriz.getColumnDimension(); i++)
144          if (this.matriz.get(0, i) < 0) {
145              return false;
146          }
147      return true;
148  }
149
150  /*
151  * Idea: compare the current marking to the marking requested for each transition.
152  * Current marking: vector with the individual marking of all the places
153  * Incidence matrix: columns = transitions | rows = places
154  * Then, if a transition of the current marking has fewer tokens than those requested by
155  ↪ the transition, it cannot be fired.
156  *
157  * @param
158  * @return
159  */
160  void enableTransitions() {
161      // currentMarking.print(2,0);
162      Long time = System.currentTimeMillis(); // tiempo actual
163      for (int i = 0; i < backwardsIncidence.getColumnDimension(); i++) {
164          boolean enabledTransition = true;
165          for (int j = 0; j < backwardsIncidence.getRowDimension(); j++) {
166              if (backwardsIncidence.get(j, i) > currentMarking.get(0, j)) {
167                  enabledTransition = false;
168                  break;
169              }
170          }
171          if (enabledTransition) {
172              sensibilizedTransitions.set(0, i, 1);
173              sensibilizedTime.set(0, i, (double) time);
174          } else {
175              sensibilizedTransitions.set(0, i, 0);
176          }
177      }
178      System.out.println("Enabled transitions: " + getEnabledTransitionsInfo());
179      // sensibilizedTransitions.print(1, 0);
180  }

```

```

181
182  /*
183  * returns a string with the enabled transitions info.
184  *
185  * @return enabled transitions info
186  */
187  public String getEnabledTransitionsInfo() {
188      String enabled = "";
189      for (int i = 0; i < 14; i++) {
190          if (sensibilizedTransitions.get(0, i) == 1) {
191              enabled += ("T" + i + " ");
192          }
193      }
194      return enabled;
195  }
196
197  public String getAllTransitionsPrint() {
198      return allTransitionsPrint;
199  }
200
201  /*
202  * - cambiar marcado actual
203  * - actualizar sensibilizadas
204  * - chequear invariantes
205  * - agregar disparo a la secuencia
206  *
207  * Este es el metodo fire transition pero quería usar el nombre
208  * fire transition en monitor
209  *
210  * This method fires a transition if it is enabled.
211  * - change current marking.
212  * - update current working vector.
213  * - update sensibilized transitions.
214  * - adds to the fired secuencia the fired transition.
215  *
216  * @param v: firing vector
217  * @return
218  */
219
220  void fire(Matrix v) // esta es la que hace el disparo literal, actualizando la rdp
221  {
222      setCurrentMarking(fundamentalEquation(v));
223      setWorkingVector(v, 0);
224      testPInvariants();
225      enableTransitions();
226      firedSequence.add("T" + getIndex(v) + ""); // tiene TODAS las secuencia de
    ↪ transiciones disparadas

```

```

227     System.out.println("Firing: T" + getIndex(v));
228     for (int i = 0; i < v.getRowDimension(); i++) {
229         for (int j = 0; j < v.getColumnDimension(); j++) {
230             if (v.get(i, j) != 0.0) {
231                 transitionCounter.set(i, j, (transitionCounter.get(i, j) + v.get(i, j)));
232             }
233         }
234     }
235     System.out.println(transitionsCounterInfo());
236     System.out.println("Current marking:\n" + getMarkingInfo());
237     // this.currentMarking.print(2,0);
238     // T1 T3 T5 T7 T9 T11 T12 T13
239     // T1 T3 T5 T7 T8 T10 T12 T13
240     // T1 T3 T4 T6 T9 T11 T12 T13
241     // T1 T3 T4 T6 T8 T10 T12 T13
242     // T0 T2 T5 T7 T9 T11 T12 T13
243     // T0 T2 T5 T7 T8 T10 T12 T13
244     // T0 T2 T4 T6 T9 T11 T12 T13
245     // T0 T2 T4 T6 T8 T10 T11 T13 */
246     String lastTransition = getIndex(v) + "";
247     allTransitionsPrint += "T" + lastTransition;
248     followUp(lastTransition);
249
250 }
251
252 public String getMarkingInfo() {
253     String marking = "P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16
↪ P17 P18\n";
254     for (int i = 0; i < currentMarking.getColumnDimension(); i++) {
255         if (i < 10) {
256             marking += ((int) currentMarking.get(0, i) + " ");
257         } else {
258             marking += ((int) currentMarking.get(0, i) + " ");
259         }
260     }
261     return (marking + "\n");
262 }
263
264 public int getCompletedInvariants() {
265     return completedInvariants;
266 }
267
268 public int getValinvariantCounting(int i) {
269     return invariantCounting[i];
270 }
271
272 /*

```

```

273  *
274  *
275  * Checks the 'state' of the transition that is going to be fired.
276  * 0 - No one is working on it. STATE = NONE
277  * 1 - Someone is working on it, but it is not the thread that is requesting it. STATE =
    ↪ OTHER
278  * 2 - The thread that is requesting it is already working on it. STATE = SELF
279  *
280  * @param v: firing vector
281  * @return
282  */
283
284  public int workingState(Matrix v) {
285      int index = getIndex(v);
286      if (workingVector.get(0, index) == 0)
287          return 0;
288      else if (workingVector.get(0, index) != Thread.currentThread().getId())
289          return 1;
290      else
291          return 2;
292  }
293
294
295  /*
296  * Follows the transitions fired in order to check the T-invariants.
297  *
298  * @param lastTransition: last transition fired
299  * @return
300  */
301  public void followUp(String lastTransition) {
302
303      if (!lastTransition.contains("13")) {
304          CurrentRoute += lastTransition;
305      } else {
306          // T1 T3 T5 T7 T9 T11 T12 T13
307          // T1 T3 T5 T7 T8 T10 T12 T13
308          // T1 T3 T4 T6 T9 T11 T12 T13
309          // T1 T3 T4 T6 T8 T10 T12 T13
310          // T0 T2 T5 T7 T9 T11 T12 T13
311          // T0 T2 T5 T7 T8 T10 T12 T13
312          // T0 T2 T4 T6 T9 T11 T12 T13
313          // T0 T2 T4 T6 T8 T10 T11 T13 */
314          if (CurrentRoute.contains("135791112")) {
315              invariantCounting[0] += 1;
316              completedInvariants++;
317          } else if (CurrentRoute.contains("135781012")) {
318              invariantCounting[1] += 1;

```

```

319         completedInvariants++;
320     } else if (CurrentRoute.contains("134691112")) {
321         invariantCounting[2] += 1;
322         completedInvariants++;
323     } else if (CurrentRoute.contains("134681012")) {
324         invariantCounting[3] += 1;
325         completedInvariants++;
326     } else if (CurrentRoute.contains("025791112")) {
327         invariantCounting[4] += 1;
328         completedInvariants++;
329     } else if (CurrentRoute.contains("025781012")) {
330         invariantCounting[5] += 1;
331         completedInvariants++;
332     } else if (CurrentRoute.contains("024691112")) {
333         invariantCounting[6] += 1;
334         completedInvariants++;
335     } else if (CurrentRoute.contains("024681012")) {
336         invariantCounting[7] += 1;
337         completedInvariants++;
338     }
339     else {
340         for (int i = 0; i < 8; i++) {
341             // System.out.println("INVARIANTE "+ i + " " + "OCURRIO " +
342             // getValinvariantCounting(i) + " VECES ");
343             System.out.println(i + " T-invariant appears " + getValinvariantCounting(i) +
↪ " times.");
344         }
345         System.out.println("Error in T-Invariant: " + CurrentRoute);
346     }
347     for (int i = 0; i < 8; i++) {
348         // System.out.println("INVARIANTE "+ i + " " + "OCURRIO " +
349         // getValinvariantCounting(i) + " VECES ");
350         System.out.println(i + " T-invariant appears " + getValinvariantCounting(i) + "
↪ times.");
351     }
352     CurrentRoute = "";
353 }
354 }
355
356 /*
357 * This method checks the P-invariants using the current marking of the places.
358 *
359 * @param
360 * @return
361 */
362 public void testPInvariants() {
363     boolean pInv0, pInv1, pInv2, pInv3, pInv4, pInv5, pInv6, pInv7, pInv8;

```



```

364     pInv0 = (currentMarking.get(0, 0) + currentMarking.get(0, 3) +
↪ currentMarking.get(0, 5)
365         + currentMarking.get(0, 6) + currentMarking.get(0, 9) + currentMarking.get(0,
↪ 11)
366         + currentMarking.get(0, 12) + currentMarking.get(0, 13) +
↪ currentMarking.get(0, 15)
367         + currentMarking.get(0, 16) + currentMarking.get(0, 17)) == 1;
368     pInv1 = (currentMarking.get(0, 1) + currentMarking.get(0, 3)) == 1;
369     pInv2 = (currentMarking.get(0, 2) + currentMarking.get(0, 5)) == 1;
370     pInv3 = (currentMarking.get(0, 14) + currentMarking.get(0, 13) +
↪ currentMarking.get(0, 15)) == 1;
371     pInv4 = (currentMarking.get(0, 7) + currentMarking.get(0, 9)) == 1;
372     pInv5 = (currentMarking.get(0, 8) + currentMarking.get(0, 11)) == 1;
373     pInv6 = (currentMarking.get(0, 9) + currentMarking.get(0, 10) +
↪ currentMarking.get(0, 11)) == 2;
374     pInv7 = (currentMarking.get(0, 17) + currentMarking.get(0, 18)) == 1;
375     pInv8 = (currentMarking.get(0, 3) + currentMarking.get(0, 4) +
↪ currentMarking.get(0, 5)
376         + currentMarking.get(0, 17)) == 3;
377
378     if (!(pInv0 && pInv1 && pInv2 && pInv3 && pInv4 && pInv5 && pInv6 && pInv7
↪ && pInv8)) {
379         System.out.println("Error in a p-invariant.");
380     }
381 }
382 /*
383  * *****
384  * *** Geters & Setters ***
385  * *****
386  */
387
388 public Matrix getCurrentMarking() {
389     return currentMarking;
390 }
391
392 public void setCurrentMarking(Matrix currentMarking) {
393     this.currentMarking = currentMarking;
394 }
395
396 public Matrix getSensibilized() {
397     return sensibilizedTransitions;
398 }
399
400 /*
401  * Returns the index of the transition that is going to be fired.
402  *
403  * @param v: firing vector

```

```
404     * @return index of the transition
405     */
406     public int getIndex(Matrix v) {
407         int index = 0;
408         for (int i = 0; i < v.getColumnDimension(); i++) {
409             if (v.get(0, i) == 1)
410                 break;
411             else
412                 index++;
413         }
414         return index;
415     }
416
417     public Matrix getAlphaTimes() {
418         return alphaTime;
419     }
420
421     public Matrix getSensibilizedTime() {
422         return sensibilizedTime;
423     }
424
425     public void setWorkingVector(Matrix firingVector, double value) {
426         this.workingVector.set(0, getIndex(firingVector), value);
427     }
428
429     public Matrix getIncidenceMatrix() {
430         return this.incidence;
431     }
432
433 }
```

#### 4.3.7. Log

Esta clase se encarga de generar los archivos de texto con la información de la ejecución del programa. Se tiene un archivo para la secuencia de transiciones disparadas, y otro para las estadísticas de la ejecución. Se muestra información de:

- Secuencia de transiciones disparadas.
- Cantidad de veces que se disparó cada transición.
- Cantidad de invariantes completados.
- Tiempo de ejecución del programa.



- $t(T_6) = 10 \text{ ms}$
- $t(T_7) = 10 \text{ ms}$
- $t(T_{10}) = 10 \text{ ms}$
- $t(T_{11}) = 10 \text{ ms}$
- $t(T_{13}) = 5 \text{ ms}$

De donde se tiene que la suma total es:

$$t(T_2) + t(T_6) + t(T_{10}) + t(T_{13}) = 30 \text{ ms}$$

Para el análisis del mismo, lo que se hizo es cumplir en el código correspondiente, un solo invariante de transición, representando este una sola imagen que circula por todo el circuito de trabajo. Al ser una sola, obliga a los hilos a trabajar uno a uno, secuencializando el escenario prácticamente a su máximo posible. Luego, se multiplica este tiempo obtenido por la cantidad de imágenes a procesar y se estima el tiempo total.

La salida del programa es de  $70 \text{ ms}$  para una sola imagen, es decir que hace el circuito de manera secuencial y espera a otra imagen. De modo, 200 imágenes serán el cálculo de:  $70 \text{ ms} * 200 \text{ imágenes} = 14000 \text{ ms}$ , es el tiempo **máximo** secuencializado.

Como conclusión se puede decir que el programa de manera real, tiene un tiempo “extra” de  $40 \text{ ms}$  para cada imagen o un tiempo extra de secuencialización de  $6000 \text{ ms}$ .

### 5.1.2. Completamente paralelo

La **máxima** temporización de las transiciones seleccionadas, es de  $10 \text{ ms}$ , esto se lo entiende como el mayor tiempo que puede ocurrir en una ejecución paralela, esto es relativo ya que el sistema primero debe cargarse de imágenes para poder trabajar de manera paralela, esto significando que en cierto punto hay un grado de secuencialidad, por lo que:

$$10 \text{ ms} * 200 \text{ imágenes} = 2000 \text{ ms}$$

En cuanto al cálculo teórico de la red trabajando completamente paralelo sin tener en cuenta la carga de la red, lo máximo que se puede corroborar con nuestro programa completando 200 imágenes es de 6950 milisegundos. Ubicándose así entre el mínimo teórico y el máximo posible secuencializado, la salida de este programa muestra un correcto funcionamiento. Y presentando una clara mejora en términos de tiempos de ejecución.

### 5.1.3. Comparación de secuencialidad y concurrencia

Para comparar el funcionamiento de la red en ambas situaciones, se planteó lo siguiente:

- Programa secuencial para una única imagen: se ejecuta el programa permitiendo que se ejecute un único invariante, y se guarda ese tiempo en “Secuencial [ms]”.

- Programa secuencial para 200 imágenes: se multiplica por 200 el tiempo obtenido anteriormente.
- Programa concurrente: es el run entero del programa tal cual fue implementado, se guarda el tiempo en “Concurrente [ms]”.

Tabla 5: Tiempos de disparo para casos secuencial y concurrente

Disparo i	Secuencial [mS]	x200 [mS]	Concurrente[mS]
1	76	15200	6965
2	80	16000	6917
3	71	14200	6915
4	67	13400	6916
5	73	14600	6929
6	69	13800	6903
7	69	13800	6956
8	72	14400	6963
9	68	13600	6917
10	73	14600	6926
11	70	14000	6956
12	74	14800	6925
13	70	14000	6963
14	66	13200	6946
15	63	12600	6932
16	66	13200	6944
17	90	18000	6989
18	66	13200	6940
19	64	12800	6991
Suma	1347	269400	131893
Promedio	71	14179	6942

## 5.2. Análisis de las políticas

Para la obtención de la transición a disparar, se plantea el uso de la clase **Math**. Se tiene como ejemplo lo siguiente:

Código 10: Ejemplo de selección de una transición

```

1 int choice = (int) Math.round(randomGenerator.nextInt(transitions.size()));
2 indexChosen = (int) Math.round(transitions.get(choice));

```

Se analiza cada política por separado.

### 5.2.1. Política equitativa

En la política en cuestión, se tiene la siguiente tanda de runs, en donde se puede apreciar el análisis de los cumplimientos de los invariantes de transición, y la carga de cada uno sobre el sistema, expresado en forma de porcentaje.

Tabla 6: Análisis de los cumplimientos de los invariantes de transición para la política equitativa

Run i	$I_0$	$I_1$	$I_2$	$I_3$	$I_4$	$I_5$	$I_6$	$I_7$	Porcentaje ( $I_0, I_3, I_5, I_7$ )
1	26	29	27	30	17	18	21	32	54.5 %
2	23	31	28	23	21	20	24	30	52 %
3	23	19	30	25	27	31	21	24	49.5 %
4	23	19	30	25	27	31	21	24	49.5 %
5	19	25	23	26	24	28	33	22	50.5 %
6	30	17	22	25	40	24	18	24	45 %
7	19	25	34	24	24	28	20	26	51.5 %

Se puede visualizar casi una distribución de 50-50 entre los distintos invariantes de la red. En promedio se tiene 50.36 %.

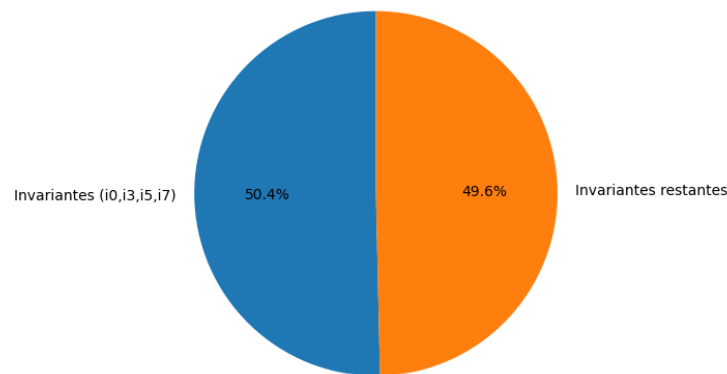


Figura 19: Promedio distribución 50-50

### 5.2.2. Política 80-20

En la política en cuestión, se tiene la siguiente tanda de runs, en donde se puede apreciar el análisis de los cumplimientos de los invariantes de transición, y la carga de cada uno sobre el sistema, expresado en forma de porcentaje.

La justificación a simple vista es que, los invariantes 1, 3, 5 y 7, poseen en ellos la transición  $T_8$ , esto significa que el monitor (la política) debe elegir entre la transición  $T_9$  o la antes

Tabla 7: Análisis de los cumplimientos de los invariantes de transición para la política 80-20

Run i	$I_0$	$I_1$	$I_2$	$I_3$	$I_4$	$I_5$	$I_6$	$I_7$	Porcentaje con $T_8$ ( $I_1, I_3, I_5, I_7$ )
1	17	41	8	34	9	48	5	38	80.5 %
2	7	25	16	30	8	55	16	43	76.5 %
3	9	39	13	33	13	41	16	36	74.5 %
4	6	43	6	38	13	47	9	38	83 %
5	8	34	9	27	8	55	12	47	81.5 %
6	9	44	11	38	14	36	7	41	79.5 %
7	15	42	10	45	6	40	9	33	80 %

mencionada. El peso que recae sobre  $T_8$ , es del 80 % entonces se puede observar que se cumple esta condición y que los invariantes que poseen a la transición  $T_8$  poseen alrededor de un 80 % (intervalo aproximado  $[74,5 - 81,5]$ ). En promedio, se tiene un 79,35 % de cumplimiento de los invariantes que poseen a  $T_8$ , prácticamente 80 %.

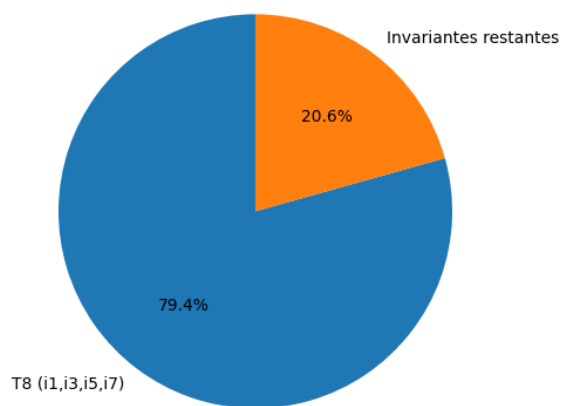


Figura 20: Promedio distribución 80-20

## 6. Conclusiones

Para finalizar este trabajo, como grupo se coincide en las siguientes conclusiones:

- La elección de la política adecuada para trabajar con las transiciones sensibilizadas es fundamental para garantizar el correcto funcionamiento del monitor de concurrencia. En este proyecto, se brindaron dos políticas distintas para observar diferentes resultados, y pudo comprobarse además el correcto funcionamiento de ambas. En un caso real, la política que tiene una cierta preferencia por un segmento podría representar un proceso que realiza su tarea a una mayor velocidad, permitiendo así que el paralelismo ayude a ahorrar tiempo de ejecución del sistema completo.
- La comparación entre el funcionamiento totalmente secuencial y el paralelo de la red de Petri permitió observar las ventajas y desventajas de cada enfoque. En particular, se pudo comprobar que la concurrencia mejora significativamente el rendimiento del sistema. No fue posible verificar el funcionamiento de la red en forma completamente pura por el hecho de que se planteó el trabajo práctico desde el principio en la consigna con el objetivo de procesar una única imagen. Esto hizo que todo el código fuera planteado de esa manera, y si se quisiera cambiar, sería necesario no solo modificar el marcado, sino también el chequeo de los invariantes. Y, este análisis, que debiera hacerse en un inicio, antes de realizar todas las implementaciones en código no pudo ser ejecutado por las computadoras del grupo (hablando de software de simulación).
- La implementación de un monitor de concurrencia para una red de Petri temporal requirió un análisis detallado de las interacciones entre los distintos hilos y la gestión de recursos compartidos, así como también de las representaciones de tiempos de sensibilización y duraciones de las tareas de cada hilo. Esto último fue lo que mayores complicaciones trajo, sobre todo en el chequeo de las temporizaciones de las diferentes transiciones (representando tiempos de trabajo). En cuanto a estos tiempos, se utilizaron los mismos que en el primer trabajo práctico (pero ahora divididos en un factor de 10), de forma de trabajar un modelo general para ambos con un enfoque continuista. Los problemas mencionados fueron todos solucionados, con una pequeña diferencia de lo charlado en clase, no se trabajó con un intervalo de tiempo  $[\alpha, \beta]$ , sino que se trabajó simplemente con un  $\alpha$  que representa el tiempo de sensibilizado. Como se pidió que las transiciones no se desensibilicen, entonces se elimina el llamado parámetro  $\beta$  (que debería ser de un valor grande), si no es del todo necesario.
- Pudo verse que la implementación de un monitor que centralice toda la concurrencia del sistema en un mismo punto, lleva a una secuencialización prácticamente total, algo que difiere completamente de lo simulado en el primer trabajo práctico, donde la vida de cada uno de los hilos permitía una ejecución concurrente. Puede verse que lo implementado anteriormente era más eficiente en cuanto al funcionamiento y los tiempos, sin embargo, con el monitor de concurrencia, se tiene un mayor control del sistema, gracias a las políticas implementadas, y que podría servir para mejorar diversos aspectos de la ejecución (por ejemplo, uso de políticas preferenciales).



- La comparación entre una política equitativa y una política prioritaria proporcionó una visión clara de cómo diferentes enfoques pueden afectar el comportamiento global del sistema. De esa forma, se podría plantear el funcionamiento de este sistema modelado en forma de simulación en, por ejemplo, diferentes arquitecturas de hardware o sistemas operativos.

## Referencias

- [1] Luis Orlando Ventre, Orlando Micolini. *Algoritmos para determinar cantidad y responsabilidad de hilos en sistemas embebidos modelados con Redes de Petri S3 PR*. (2021). Laboratorio de Arquitectura de Computadoras, FCEFYN-Universidad Nacional de Córdoba. [https://www.researchgate.net/publication/358104149\\_Algoritmos\\_para\\_determinar\\_cantidad\\_y\\_responsabilidad\\_de\\_hilos\\_en\\_sistemas\\_embebidos\\_modelados\\_con\\_Redde\\_de\\_Petri\\_S\\_3\\_PR](https://www.researchgate.net/publication/358104149_Algoritmos_para_determinar_cantidad_y_responsabilidad_de_hilos_en_sistemas_embebidos_modelados_con_Redde_de_Petri_S_3_PR)
- [2] Orlando Micolini, Marcelo Cebollada, Luis Orlando Ventre, Maximiliano Andrés Eschoyez. *Ecuación de estado generalizada para redes de Petri no autónomas y con distintos tipos de arcos*. (2016). Laboratorio de Arquitectura de Computadoras (LAC) FCEFYN Universidad Nacional de Córdoba. [https://www.researchgate.net/publication/328253053\\_Ecuacion\\_de\\_estado\\_generalizada\\_para\\_redes\\_de\\_Petri\\_no\\_autonomas\\_y\\_con\\_distintos\\_tipos\\_de\\_arcos](https://www.researchgate.net/publication/328253053_Ecuacion_de_estado_generalizada_para_redes_de_Petri_no_autonomas_y_con_distintos_tipos_de_arcos)