Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales
Escuela de Ingeniería en Computación

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: 5 de febrero de 2024
Córdoba, Argentina

Resumen

En el siguiente informe se adjunta el desarrollo del pryecto 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

Índice de Figuras

Índice de Tablas

Índice de Códigos

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

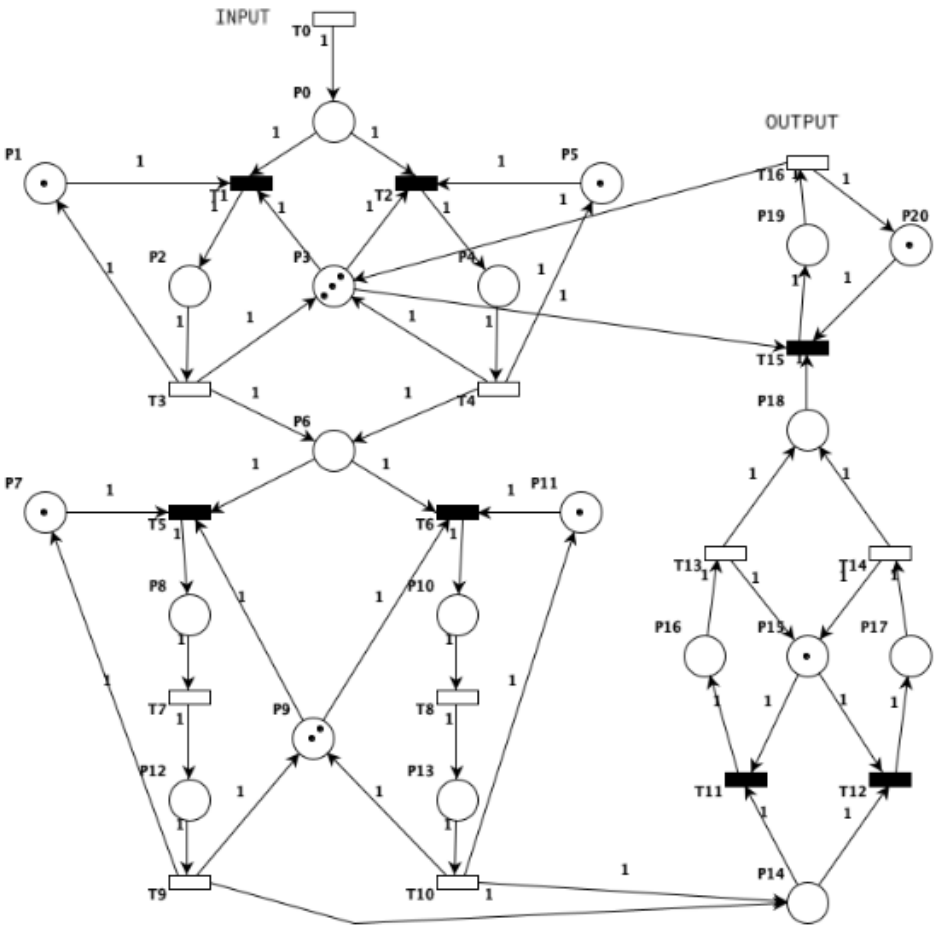

Figura 1: Red de Petri en la consigna

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.

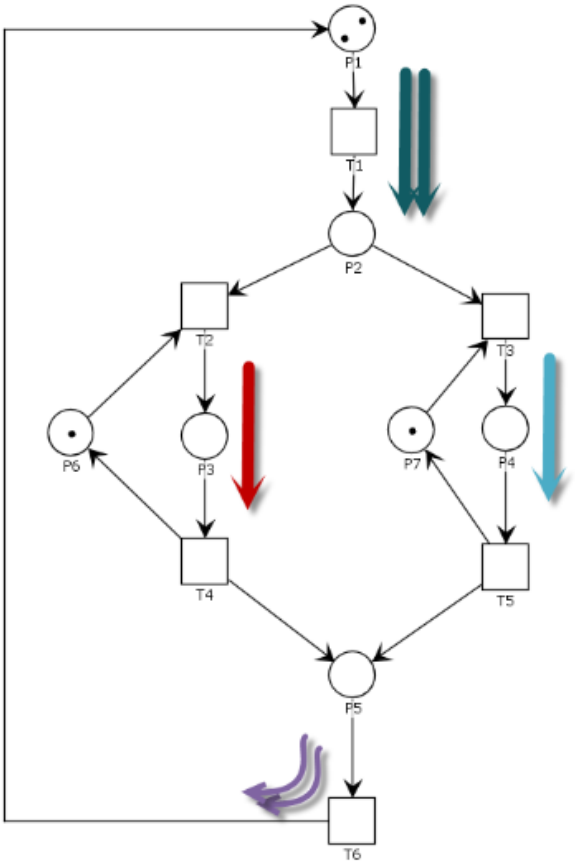

Figura 2: Responsabilidades por hilos

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.

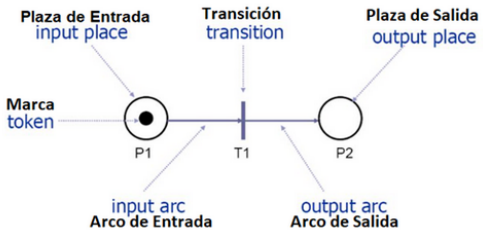

Figura 3: Partes de una red de Petri

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 una la plaza con la transición.
  - Se genera a las plazas de salida de la transición el numero 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

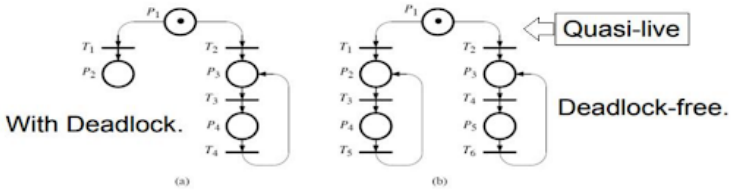

Figura 4: Deadlock en una red de Petri

- 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 secuencializar 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.

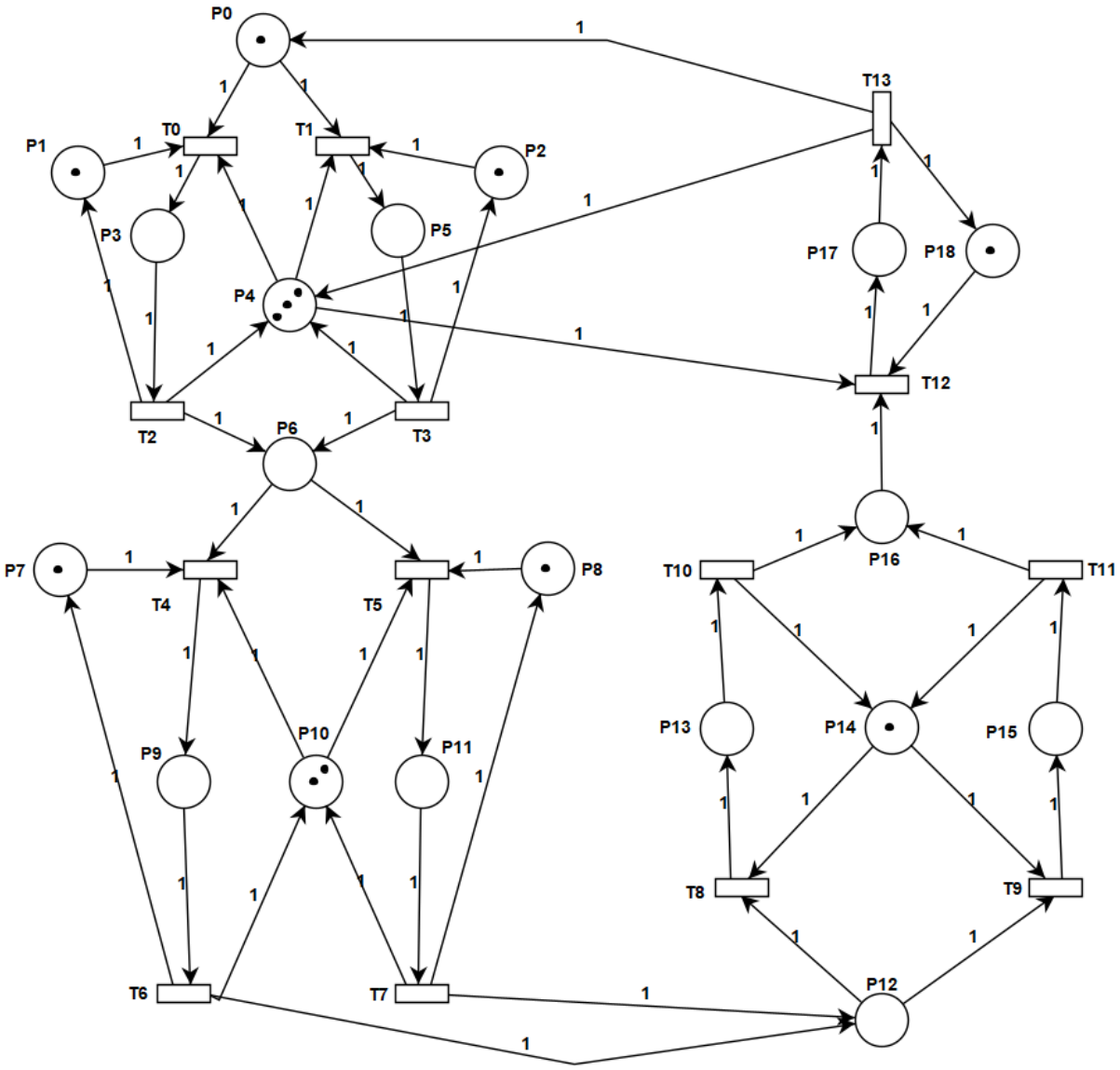

Figura 5: Red de Petri analizada y modelada

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	1	0	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.

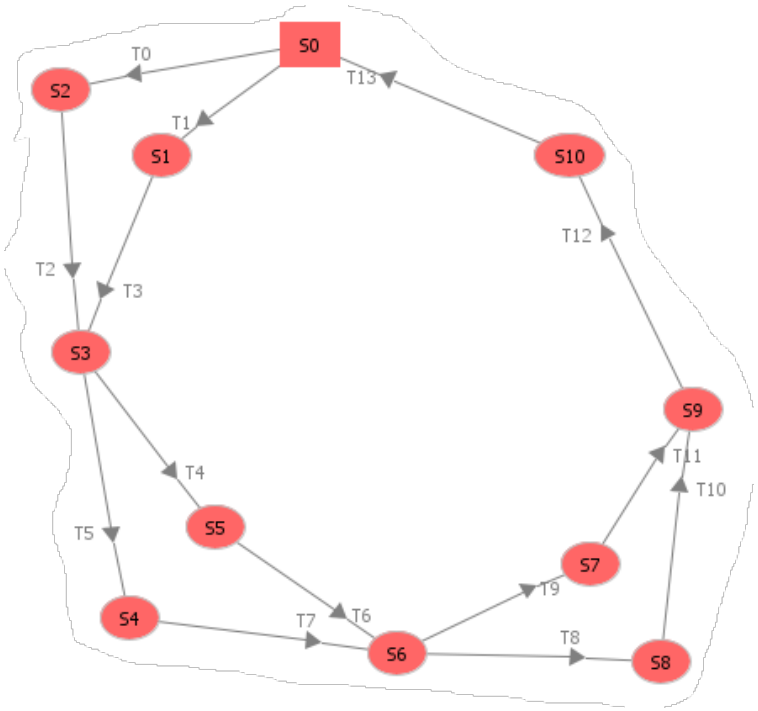

Figura 6: Diagrama de eventos de la red de Petri

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, 0, 0, 1, 1, 0, 3, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 3, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 1, 0, 0, 0, 1, 1, 0, 3, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 3, 0, 0, 0, 1, 1, 0)	(0, 1, 2, 0, 0, 0, 1, 0, 0, 1, 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 tiene 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.

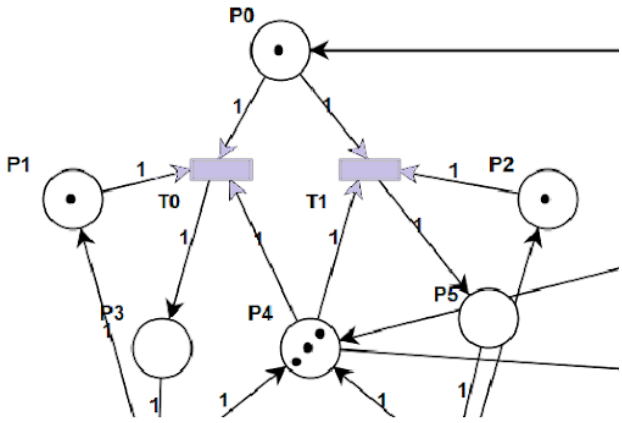

Figura 7: Justificación máquina de estados

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.

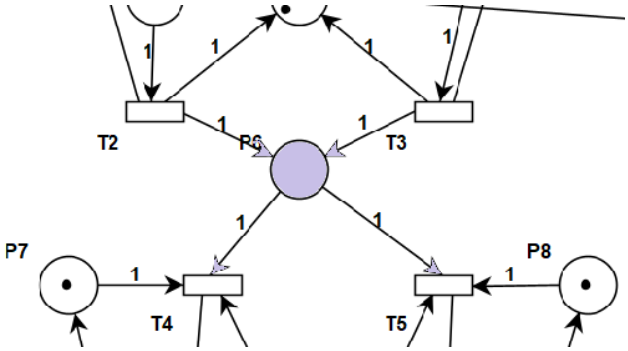

Figura 8: Justificación grafo de marcado

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.

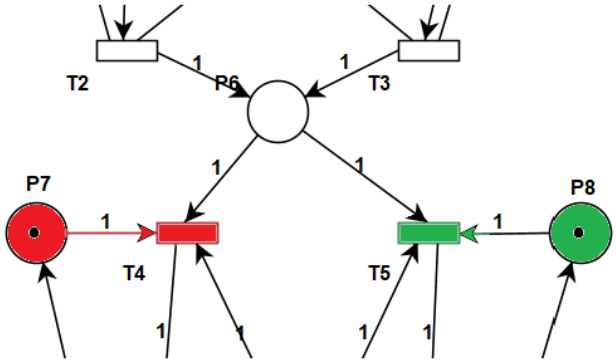

Figura 9: Justificación libre elección

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.

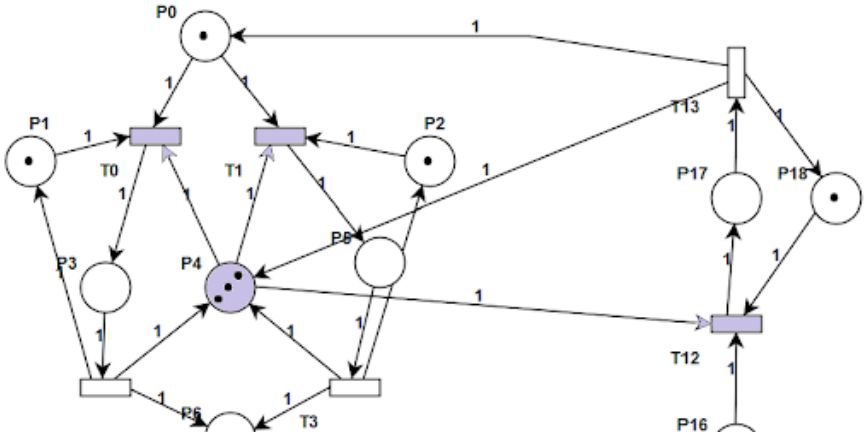

Figura 10: Justificación red simple

3.4.10. Red simple extendida

La red es simple extendida, ya que las transiciones asociadas a un conflicto estructural tienen la misma cantidad de conflictos.

3.5. Invariantes

3.5.1. Invariantes de plaza

Esta red de Petri contiene 9 invariantes de plaza. En el código a implementar se chequeará constantemente el cumplimiento de los mismos. Los invariantes de plaza son los siguientes:

$$M_0 + M_3 + M_5 + M_6 + M_9 + M_{11} + M_{12} + M_{13} + M_{15} + M_{17} + M_{17} = 1 \tag{1}$$

$$M_1 + M_3 = 1 \tag{2}$$

$$M_2 + M_5 = 1 \tag{3}$$

$$M_{14} + M_{13} + M_{15} = 1 \tag{4}$$

$$M_7 + M_9 = 1 \tag{5}$$

$$M_8 + M_{11} = 1 \tag{6}$$

$$M_9 + M_{10} + M_{11} = 2 \tag{7}$$

$$M_{17} + M_{18} = 1 \tag{8}$$

$$M_3 + M_4 + M_5 + M_{17} = 3 \tag{9}$$

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.

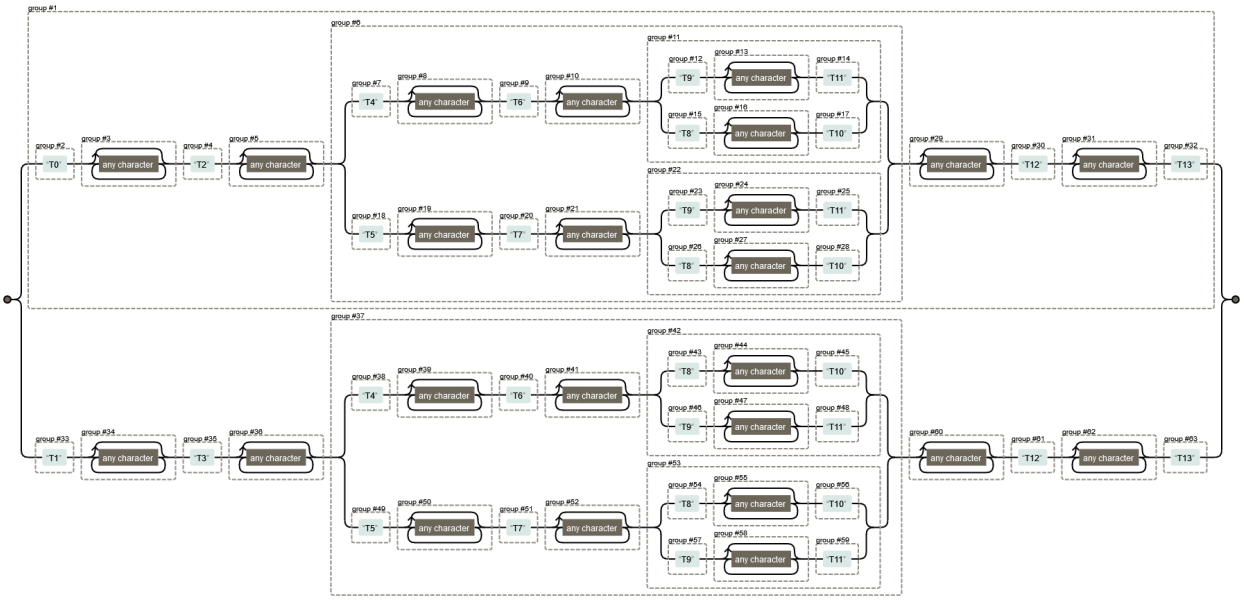

Figura 11: Invariantes de transición (expresión regular)

La expresión regular, puede colocarse en diferentes herramientas, como por ejemplo Regexper con el siguiente código:

Código 1: Expresión regular obtenida

```
('((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):

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

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

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

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

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

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

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

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

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

$$\{P_0, P_1, P_2, P_4, P_6, P_7, P_8, P_{10}, P_{12}, P_{14}, P_{16}, P_{18}\} \tag{18}$$

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

$$PA_1 = \{P_5, P_{11}, P_{15}, P_{17}\} \tag{19}$$

$$PA_2 = \{P_5, P_{11}, P_{13}, P_{17}\} \tag{20}$$

$$PA_3 = \{P_5, P_9, P_{15}, P_{17}\} \tag{21}$$

$$PA_4 = \{P_5, P_9, P_{13}, P_{17}\} \tag{22}$$

$$PA_5 = \{P_3, P_{11}, P_{15}, P_{17}\} \tag{23}$$

$$PA_6 = \{P_3, P_{11}, P_{13}, P_{17}\} \tag{24}$$

$$PA_7 = \{P_3, P_9, P_{15}, P_{17}\} \tag{25}$$

$$PA_8 = \{P_3, P_9, P_{13}, P_{17}\} \tag{26}$$

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
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
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
$PS_1 = P_3$	$MS_1 = 0\|1$
$PS_2 = P_5$	$MS_2 = 0\|1$
$PS_3 = P_9$	$MS_3 = 0\|1$
$PS_4 = P_{11}$	$MS_4 = 0\|1$
$PS_5 = P_{13}$	$MS_5 = 0\|1$
$PS_6 = P_{15}$	$MS_6 = 0\|1$
$PS_7 = P_{17}$	$MS_7 = 0\|1$

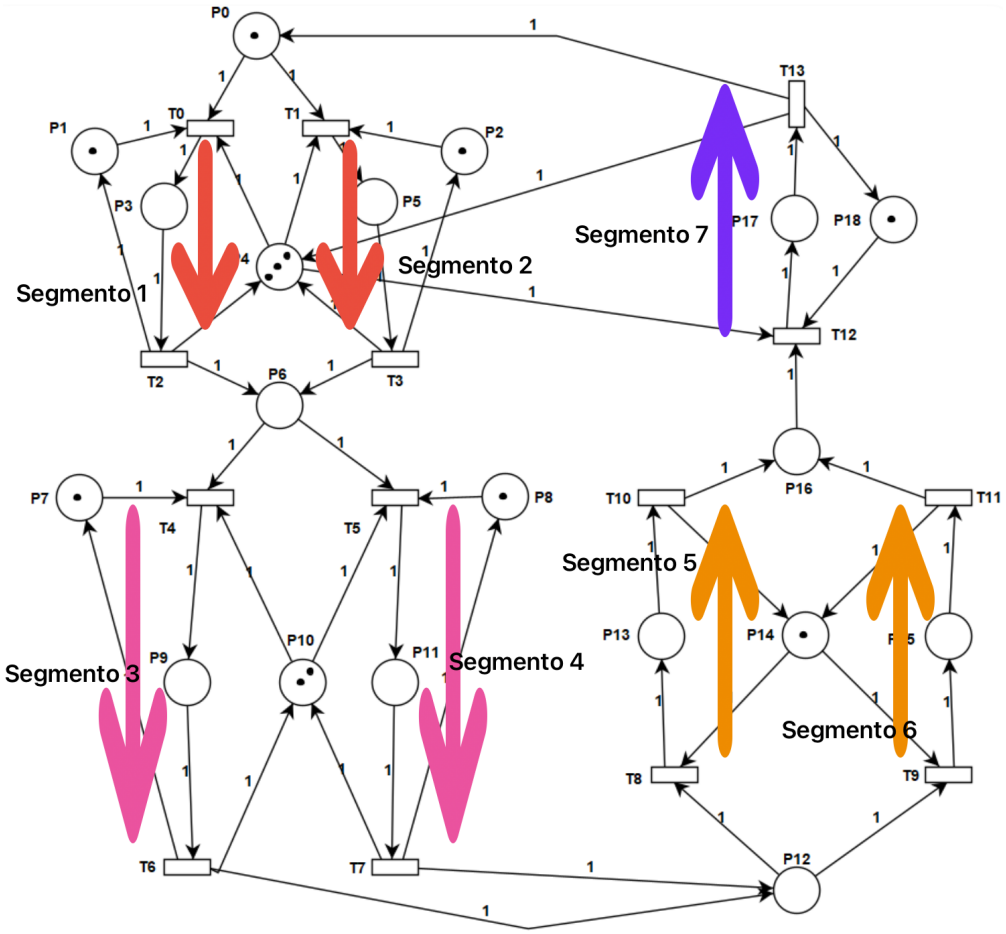

Figura 12: Segmentos de la red

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 hilos**.

3.6.1. Asignación de hilos según criterio del grupo

Por la estructura que presenta esta red de Petri, puede verse que el máximo paralelismo posible se lograría con el uso de 7 hilos, cada uno representando a los segmentos mencionados anteriormente. En donde:

- Segmento 1: transiciones 0 y 2.
- Segmento 2: transiciones 1 y 3.
- Segmento 3: transiciones 4 y 6.
- Segmento 4: transiciones 5 y 7.
- Segmento 5: transiciones 8 y 10.
- Segmento 6: transiciones 9 y 11.

- Segmento 7: transiciones 12 y 13.

Este máximo paralelismo sería aprovechado en el caso en el que se tengan más de una imagen a procesar. Ahora, esto último tiene otra consideración, podrían cambiar los invariantes de transición, por lo que no se tiene exactamente el mismo caso de estudio.

Se considera que el uso de 7 hilos es beneficioso para el sistema, ya que se aprovecha al máximo el paralelismo con una menor cantidad de hilos. Estos segmentos vienen dados por la dependencia que tienen entre sí los disparos de los grupos de 2 transiciones mencionados arriba. Si se aumenta la cantidad de hilos, poniendo por ejemplo, uno por transición a disparar, no se ganaría en paralelismo, ya que, por ejemplo, para el caso de las transiciones 0 y 2, el token único inicial en la plaza 1, limita el disparo de T2 a ser inmediatamente posterior a T0.

Se puede apreciar también que, según el marcado inicial de la red, hay dos situaciones a considerar:

- El marcado inicial de la plaza P4, podría ser de 2 tokens y el sistema seguiría funcionando igual, incluso con más imágenes a procesar.
- El único token disponible en el marcado inicial en la plaza P14 hace que, pueda eliminarse un par de transiciones (8 y 10, o 9 y 11) y el sistema podría seguir funcionando de forma similar, ya que en esa sección de la red no es posible aprovechar el paralelismo, independientemente de la cantidad de imágenes a procesar.

3.6.2. Asignación de hilos según bibliografía

En la bibliografía mencionada por la cátedra, así como en la consigna del trabajo práctico, se menciona el análisis de los casos especiales de joins y forks en los invariantes de una red de Petri. En la red en estudio, se pueden apreciar:

- Joins: T2 y T3 en la plaza P6, T6 y T7 en la plaza P12, T10 y T11 en la plaza P16.
- Forks: T0 y T1 en la plaza P0, T4 y T5 en la plaza P6, T8 y T9 en la plaza P12.

De esa forma, utilizando el criterio, citado textualmente del paper:

> *En el caso de que dos (o más) ITs compartan transiciones en conflicto estructural (fork). La responsabilidad de ejecución de los ITs es segmentada. Esto tiene como ventaja que se elimina la lógica interna del hilo para decidir frente a un conflicto, por lo que la decisión del conflicto es tomada por solo por el componente responsable, el cual es la política. La responsabilidad de ejecución de este IT es asignada a distintos segmentos de ejecución; un segmento antes del conflicto (fork) y dos (o más) segmentos posteriores.*
>
> *En el caso de que dos (o más) ITs tengan en sus estructuras una plaza de unión (join). La responsabilidad de ejecución de los ITs es segmentada. Hasta el punto*

> *de unión (join) en dos o más segmentos, uno por IT, y después de la unión (join) solo un segmento de ejecución extra. Esto tiene como ventaja que mejora el paralelismo de la ejecución, dado que permite la ejecución de los segmentos posteriores y anteriores simultáneamente.*

Y, de la consigna del trabajo práctico:

> *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.*

Con las consideraciones citadas, y, con los casos de estudio mencionados anteriormente como los joins y forks presentes en la red, se pueden entonces obtener:

1. Fork en P0, con T0 y T1. Se asigna un hilo a T13, uno a T0 y uno a T1.
2. Fork en P6, con T4 y T5. Se asigna un hilo a T4 y uno a T5.
3. Join en P6, con T2 y T3. Se asigna un hilo a T2 y uno a T3.
4. Fork en P12, con T8 y T9. Se asigna un hilo a T8 y uno a T9.
5. Join en P12, con T6 y T7. Se asigna un hilo a T6 y uno a T7.
6. Join en P16, con T10 y T11. Se asigna un hilo a T10, uno a T11 y uno a T12.

Finalmente, se obtienen de esa forma, 14 hilos, en donde cada uno se encarga de una transición, y, en donde se tiene en cuenta los joins y forks presentes en la red.

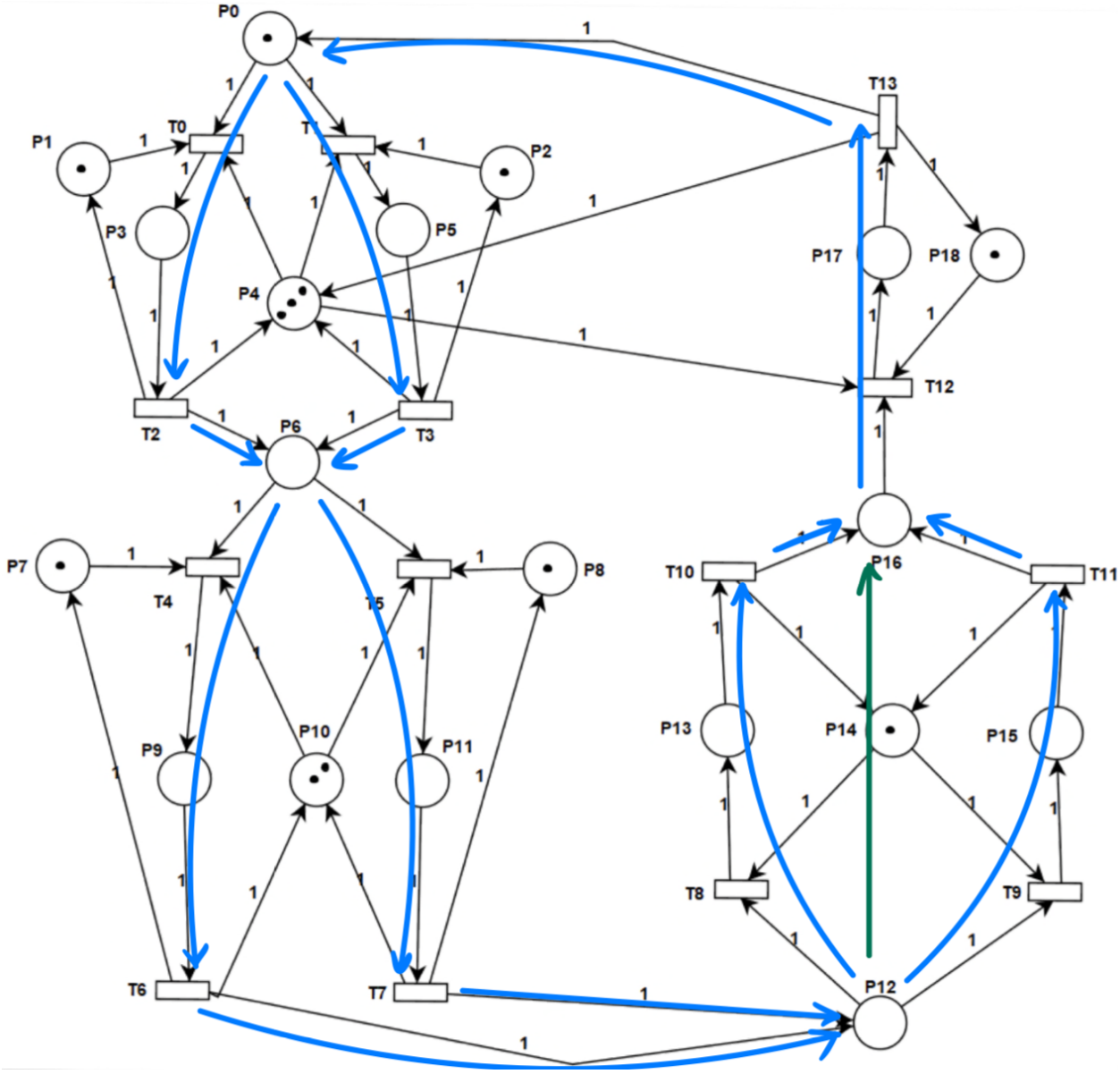

Figura 13: Hilos en la red

Puede apreciarse en la imagen que la mayoría de las flechas están en color azul, pero hay una en específico que es de color verde. Esto es porque, según el criterio expresado por la bilbiografía, se debería tener un único hilo luego del join en la plaza P12. Este único hilo viene dado por la máxima cantidad de tokens en la plaza P14 (un único), pero esta asignación no permitiría el uso de una política preferencial en donde se priorice la ejecución de un segmento por sobre el otro, es por eso, que se agrega un nuevo hilo para poder permitir cumplir con los requerimientos de diseño del sistema. Se sobreentiende que, las flechas que representan a los 14 hilos son las de color azul, la verde está a modo ilustrativo para la justificación del agregado de un nuevo hilo.

4. Implementación

4.1. Hilos de ejecución

Pudo apreciarse en la sección de estudio de la red de Petri, cómo se obtiene una cantidad de hilos para la implementación en software del sistema modelado según un criterio del grupo, y otro estrictamente teórico.

Se decide finalmente implementar el criterio del grupo, esto por que la cantidad de hilos resultantes del análisis meramente teórico no tienen ventaja alguna en la posibilidad de brindar paralelismo por la estructura misma de la red. Con esto se hace referencia a que, el par de transiciones T0 y T2, por ejemplo, así como también T1 y T3 se encuentran relacionadas directamente por el hecho de la plaza limitadora que tienen asociados. Estas plazas que limitan las capacidades de tokens permiten modelar a la perfección recursos en un sistema (como un procesador de imágenes en este caso), pero la interpretación del par de transiciones como diferentes hilos no permite ninguna mejora en el rendimiento, ni tampoco en el paralelismo implementado, es por eso que se decide implementar un sistema con 7 hilos de ejecución.

Se entiende que los pares de transiciones permiten más bien representar el inicio de la actividad, con la primera transición del par inmediata, y la segunda temporizada permitiría representar la finalización de la actividad cuya duración no es nula. Los hilos son representados como:

- Loader Left: transiciones 0 y 2.
- Loader Right: transiciones 1 y 3.
- Resizer Left: transiciones 4 y 6.
- Resizer Right: transiciones 5 y 7.
- Improver Left: transiciones 8 y 10.
- Improver Right: transiciones 9 y 11.
- Exit: transiciones 12 y 13.

4.2. Parseo de la expresión regular

Para el parseo de la expresión regular se hizo un ejecutable en Python que se encarga se 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

```
import re

source = open('transitions.txt','r')
candidate = source.readlines()[0]
source.close()
output = open('output.txt','w')
output.write(candidate)
output.close()
while True:
    f = open('output.txt','r')
    transitions = f.readlines()[0]
    f.close()
    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)
    print(match)
    f = open('output.txt','w')
    f.write(match[0])
    f.close()
    if match[1] == 0:
        print('FAIL: sobraron transiciones')
        break
    if match[0] == '':
        print('SUCCESS, Test OK')
        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.

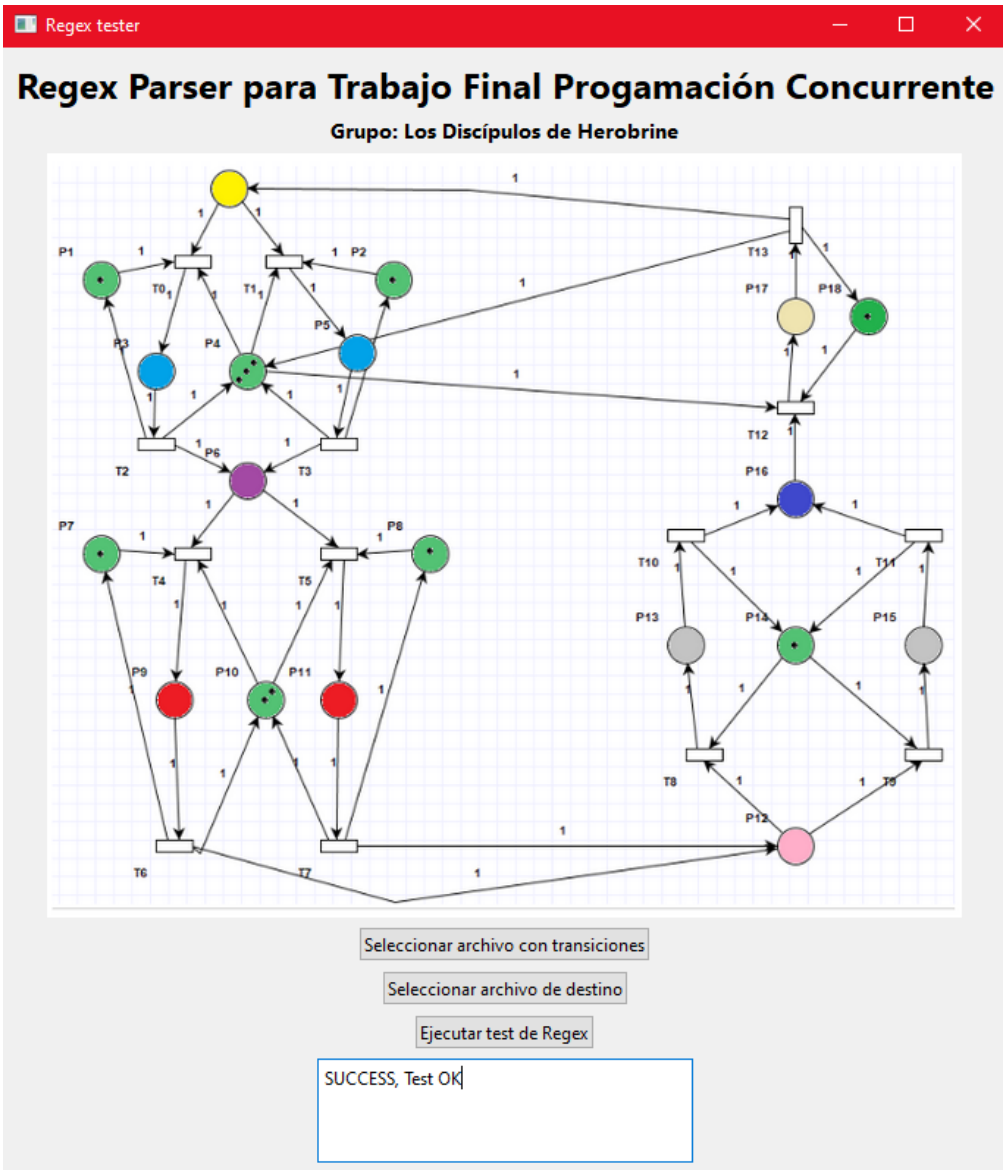

Figura 14: Interfaz gráfica del parser de la expresión regular

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

4.3. Monitor de concurrencia

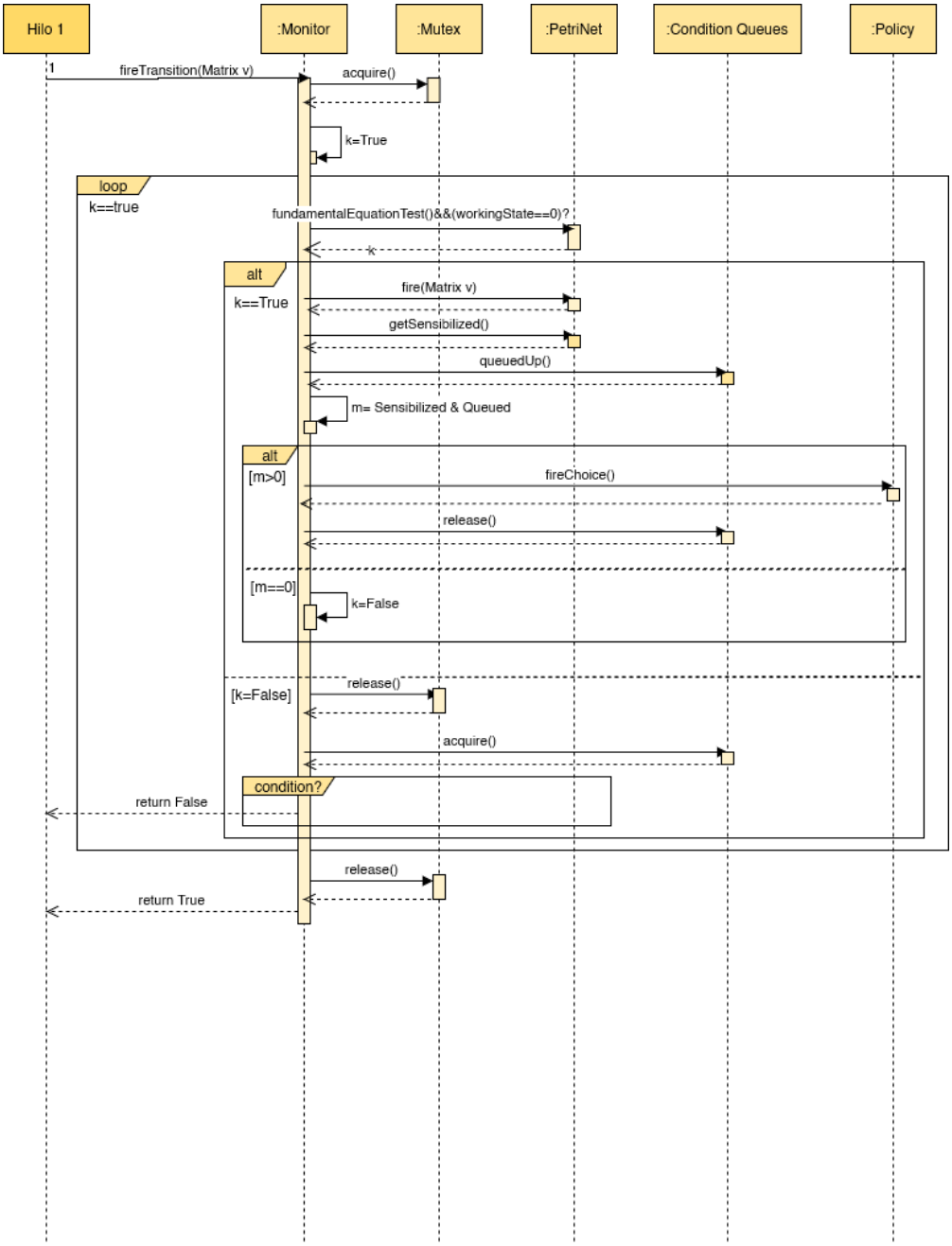

Figura 15: Diagrama de secuencia monitor para la red de Petri no temporal

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.

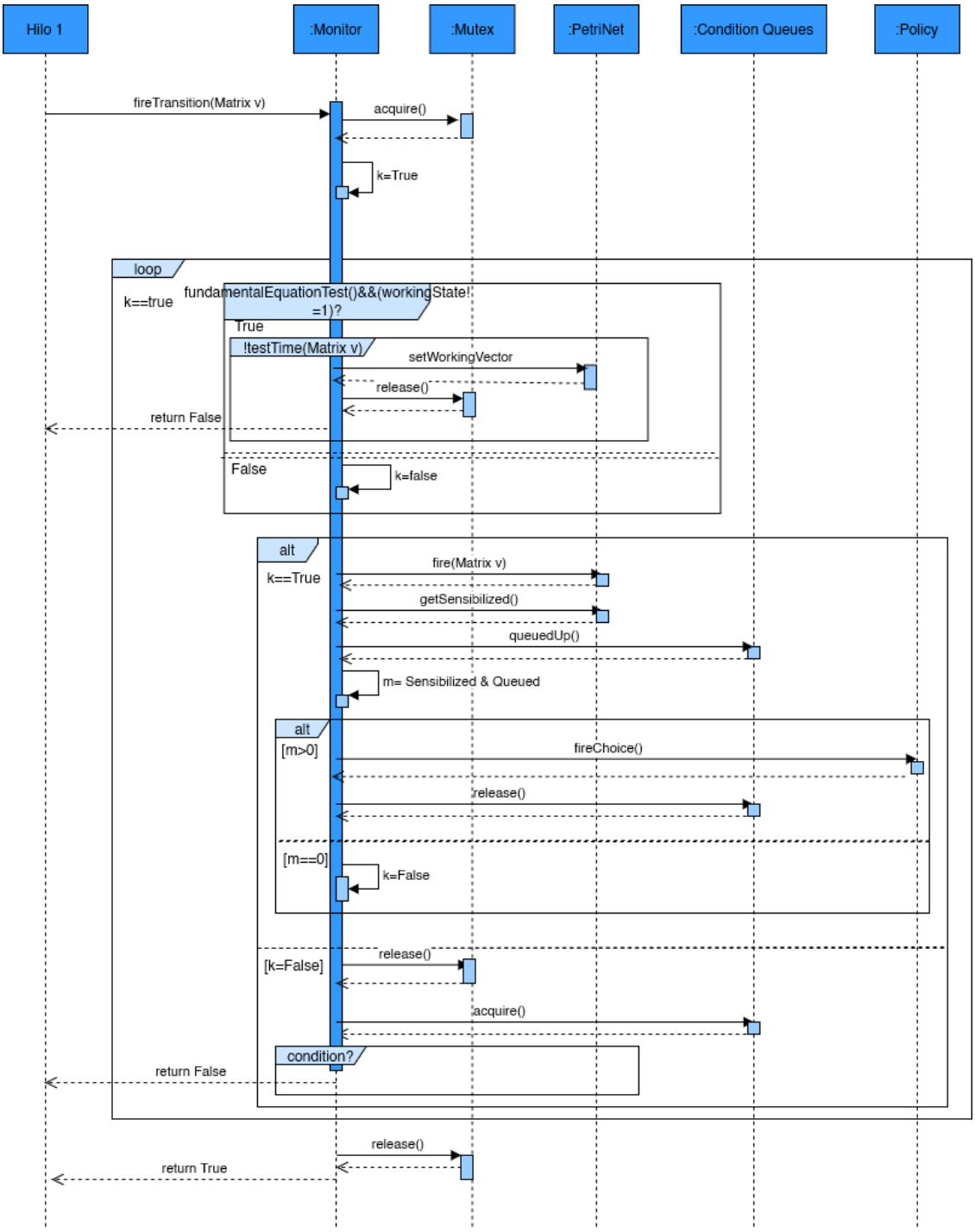

Figura 16: Diagrama de secuencia monitor para la red de Petri temporal

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 α 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 α.

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.

Luego, con objetivo de independizarse de la variable "kz tener un código y diagrama más fácil de leer e interpretar, se realizaron algunas modificaciones a la versión de la figura 16. En la figura 17 se puede observar la versión final, que, a fines prácticos coincide con la explicación anterior.

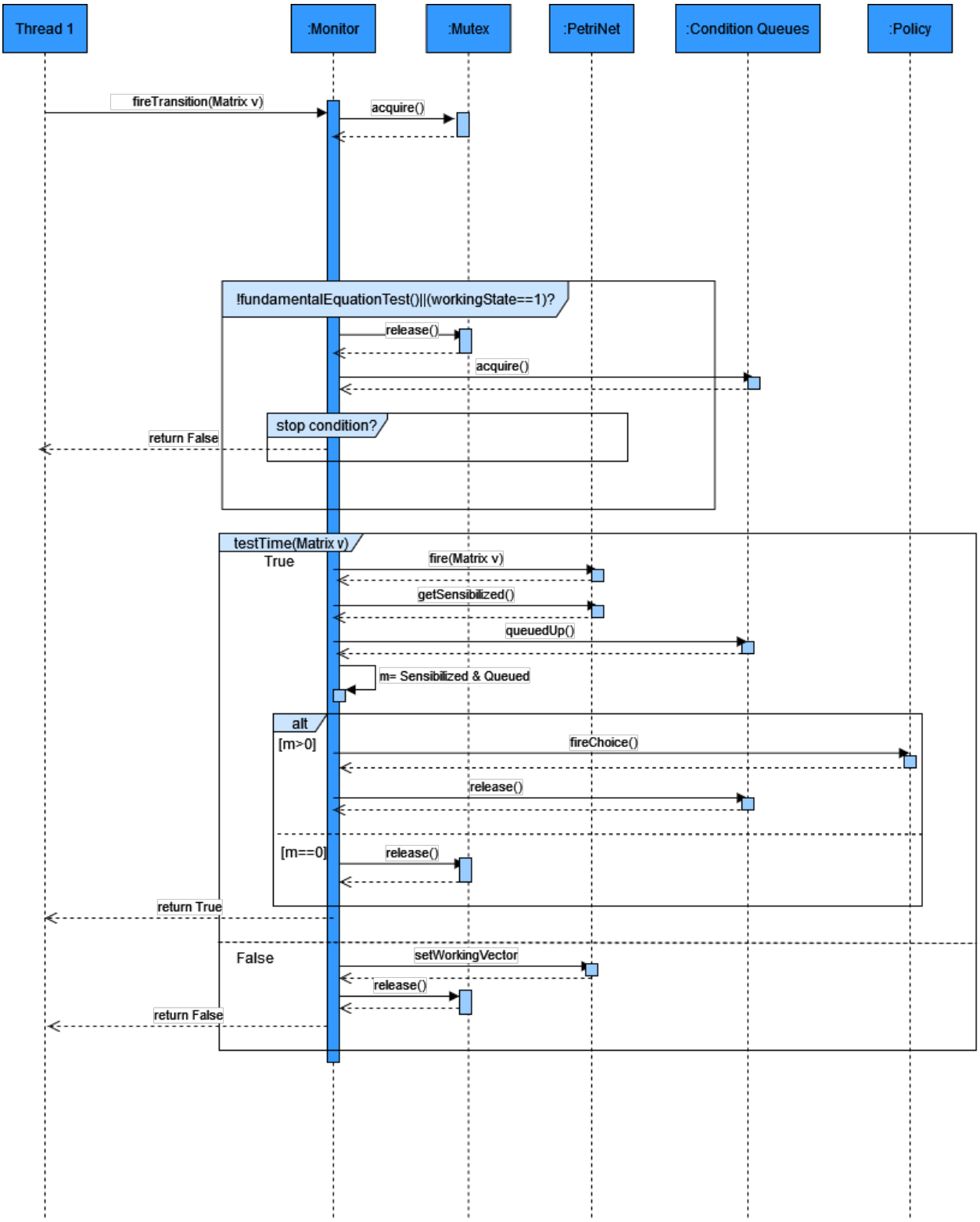

Figura 17: Diagrama de secuencia monitor para la red de Petri temporal (última versión)

4.4. Clases implementadas

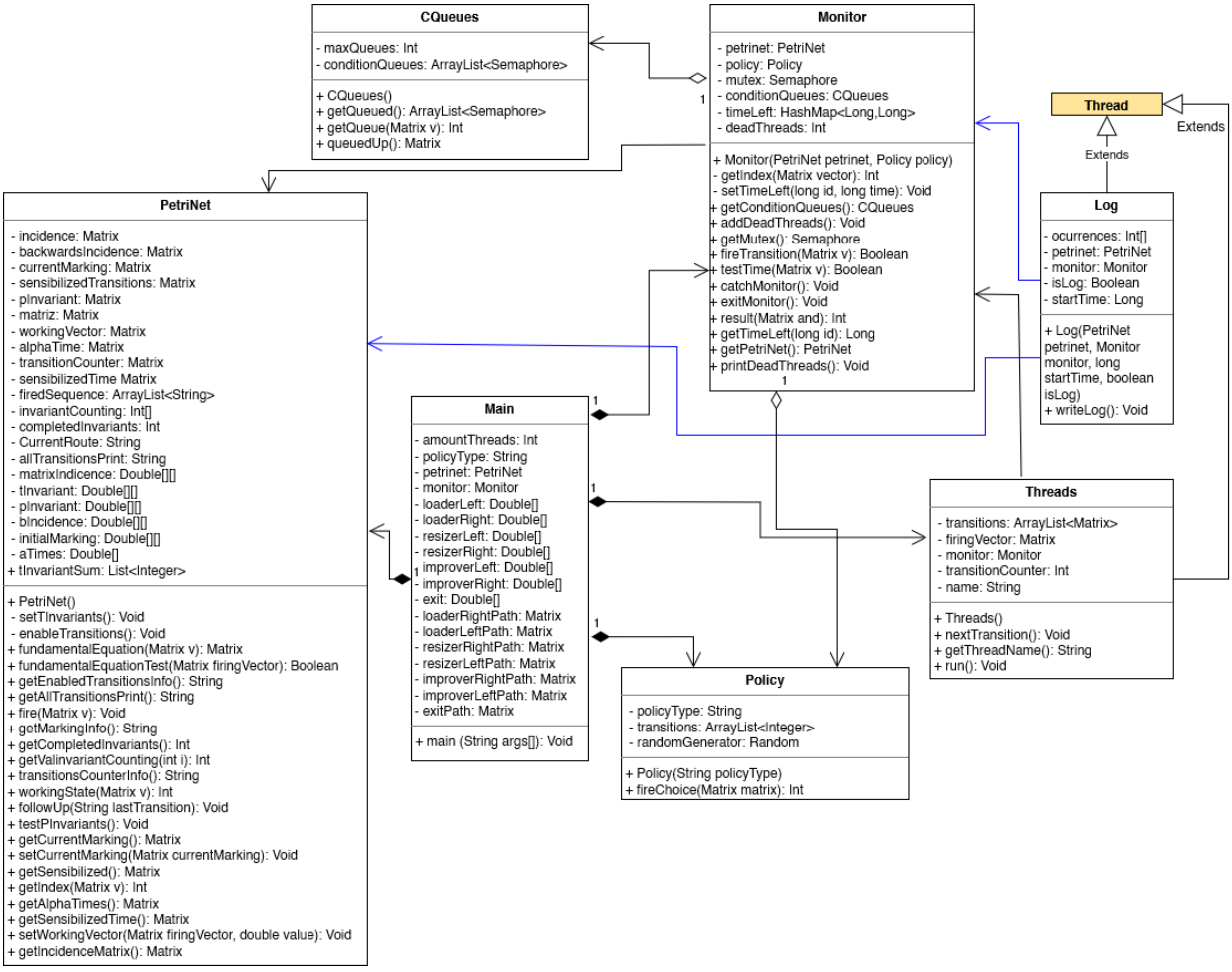

Figura 18: Diagrama de clases del sistema implementado

4.4.1. 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.*;
2  import java.util.concurrent.Semaphore;
3  import Jama.Matrix;
4
5  public class CQueues {
6
7    private final int maxQueues = 14;
8    private ArrayList<Semaphore> conditionQueues;
9
10   public CQueues() {
11     conditionQueues = new ArrayList<Semaphore>();
12     for (int i = 0; i < maxQueues; i++) {
13       conditionQueues.add(new Semaphore(0));
14     }
15   }
16
17
18   public ArrayList<Semaphore> getQueued() {
19     return conditionQueues;
20   }
21
22
23   /*
24     * Returns the index of the first thread queued up for the transition associated with the
       ↪ vector v.
25     *
26     * @param v: firing vector
27     * @return the index of the first thread queued up for the transition associated with the
       ↪ vector v
28     */
29
30   public int getQueue(Matrix v) {
31     int index = 0;
32
33     for (int i = 0; i < v.getColumnDimension(); i++) {
34       if (v.get(0, i) == 1)
35         break;
36       else
37         index++;
38     }
39     return index;
40   }
41
42   /*
43     * Returns the number of threads queued up transitions.
44     *
```

```
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  }
```

4.4.2. 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 le ejecución realizada.

Código 5: Clase Main

```
1  import Jama.Matrix;
2
3  public class Main {
4    private static final int amountThreads = 7;
5    private static final String policyType = "Equitative";
6    //private static final String policyType = "8020";
7    private static PetriNet petrinet;            // Petri net representative of the system.
8    private static Monitor monitor;              // Monitor that will control the Petri net
       ↪ that models the system.
9    private static double[] loaderLeft = { 0,2 };
10
11   private static double[] loaderRight = { 1,3 };
12
13   private static double[] resizerLeft = { 4, 6 };
14
15   private static double[] resizerRight = { 5,7 };
```

```
16
17  private static double[] improverLeft = { 8,10 };
18
19  private static double[] improverRight = { 9,11 };
20
21  private static double[] exit = { 12,13 };
22
23  // arrays with transitions associated with threads
24  private static Matrix loaderLeftPath = new Matrix(loaderLeft, 1);        // transposed
      ↪ version of de loaderLeft
25  private static Matrix loaderRightPath = new Matrix(loaderRight, 1);      // transposed
      ↪ version of de loaderRight
26  private static Matrix resizerLeftPath = new Matrix(resizerLeft, 1);      // transposed
      ↪ version of de resizerLeft
27  private static Matrix resizerRightPath = new Matrix(resizerRight, 1);    // transposed
      ↪ version of de resizerRight
28  private static Matrix improverLeftPath = new Matrix(improverLeft, 1);    // transposed
      ↪ version of de improverLeft
29  private static Matrix improverRightPath = new Matrix(improverRight, 1);  //
      ↪ transposed version of de improverRight
30  private static Matrix exitPath = new Matrix(exit, 1);                    // transposed
      ↪ version of de exit
31
32  /**
33    * Main method.
34    *
35    * Here the threads with their associated paths are instantiated and executed.
36    * Both the Petri net are also initialized with their initial marking
37    * like the monitor and the logger thread.
38    */
39
40  public static void main(String args[]) {
41    Long initTime = System.currentTimeMillis();
42    petrinet = new PetriNet();
43
44    Policy policy = new Policy(policyType);
45    System.out.println("Policy type: " + policyType + " \n");
46
47    monitor = new Monitor(petrinet, policy);
48
49    Threads[] threads = new Threads[amountThreads];
50
51    petrinet.enableTransitions();
52
53    try {
54      long startTime = System.currentTimeMillis();
55      Log log = new Log(petrinet, monitor, startTime, true);
```

```
56        log.start();
57
58        Log transition = new Log(petrinet, monitor, startTime, false);
59        transition.start();
60
61      } catch (Exception e) {
62        System.err.println("Error creating logger.");
63        System.exit(1);     // Stop the program with a non-zero exit code
64      }
65
66      threads[0] = new Threads(loaderLeftPath, monitor, "Loader Left");
67      threads[1] = new Threads(loaderRightPath, monitor, "Loader Right");
68      threads[2] = new Threads(resizerLeftPath, monitor, "Resizer Left");
69      threads[3] = new Threads(resizerRightPath, monitor, "Resizer Right");
70      threads[4] = new Threads(improverLeftPath, monitor, "Improver Left");
71      threads[5] = new Threads(improverRightPath, monitor, "Improver Right");
72      threads[6] = new Threads(exitPath, monitor, "Exit");
73
74
75      for (Threads thread : threads) {
76        thread.start();
77      }
78      try {
79        for (Threads thread : threads) {
80          thread.join();
81        }
82      }
83      catch(Exception e) {
84        System.err.println("I can't wait, I'm tired.");
85        System.exit(1);     // Stop the program with a non-zero exit code
86      }
87      Long finalTime = System.currentTimeMillis();
88      //monitor.printDeadThreads();
89      System.out.println("\nEnding program!");
90      System.out.println(petrinet.transitionsCounterInfo());
91      System.out.println("\nT invariants:");
92      petrinet.tInvariantsInfo();
93      System.out.println("\nElapsed Time: " + (double)((finalTime-initTime)/1000.0) + "
        ↪ seconds");
94    }
95  }
```

4.4.3. 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 politica carga 80 % de las decisiones sobre la transición T8, 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     this.policyType = policyType;
13     this.transitions = new ArrayList<>();
14     randomGenerator = new Random();
15   }
16
17   /*
18     * Segment 1: T0 T1
19     * Segment 2: T4 T5
20     * Segment 3: T8 T9
21     */
22   public int fireChoice(Matrix matrix) {
23     int indexChosen = 0;
24     if (policyType == "8020") { // asks the policy type
```

```
25        if (matrix.get(0, 8) >= 1 || matrix.get(0, 9) >= 1) { // if transitions 8 or 9 are
    ↪ enabled, changes the
26                                                  // probability to 80-20
27          double probT8 = 0.8;
28          // generates a random number between 0 and 1
29          double randomNum = randomGenerator.nextDouble();
30          if (randomNum < probT8) {
31            indexChosen = 8; // Chose index 8
32            System.out.println("Firing T8");
33          } else {
34            indexChosen = 9; // Chose index 9
35            System.out.println("Firing T9");
36          }
37        } else {
38          // If the sensibilized transitions are not T8 or T9, chooses randomly
39          transitions.clear(); // clear array
40          for (int i = 0; i < matrix.getColumnDimension(); i++) {
41            if (matrix.get(0, i) > 0) {
42              transitions.add(i);
43            }
44          }
45          int choice = (int) Math.round(randomGenerator.nextInt(transitions.size()));
46          indexChosen = (int) Math.round(transitions.get(choice));
47        }
48      } else if (this.policyType == "Equitative") {
49        // if policy type is equitative chooses randomly with a normal distribution of
    ↪ probabilities    // red
50        transitions.clear(); // Clears array
51        for (int i = 0; i < matrix.getColumnDimension(); i++) {
52          if (matrix.get(0, i) > 0) {
53            transitions.add(i);
54          }
55        }
56        int choice = (int) Math.round(randomGenerator.nextInt(transitions.size()));
57        indexChosen = (int) Math.round(transitions.get(choice));
58      }
59      return indexChosen;
60    }
61  }
```

4.4.4. 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 α. 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
      ↪ 'testTime' method.
44      * In case all conditions are met, the transition is fired and the method returns true.
45      * Otherwise, the thread is queued up and the method returns false.
46      *
47      * @param v: firing vector
48      * @return true if the transition is fired, false otherwise
49      */
50    public boolean fireTransition(Matrix v)
51    {
52
53      try {
54        if (petrinet.getCompletedInvariants() < 200) {
55          catchMonitor();
56        } else {
57          return false;
58        }
59      } catch(Exception e) {
60        System.err.println("I was interrupted with the monitor in my hands");
61        System.exit(1);     // Stop the program with a non-zero exit code
62      }
63
64
65      if(!petrinet.fundamentalEquationTest(v) || (petrinet.workingState(v) == 1)) { //
      ↪ Someone is working on it, but it is not the thread that is requesting it. STATE =
      ↪ OTHER
66        exitMonitor();
67
68        int queue = conditionQueues.getQueue(v);
69
70        try {
71          conditionQueues.getQueued().get(queue).acquire();
72          if (petrinet.getCompletedInvariants() >= 200)
```

```
73                  return false;
74          } catch(Exception e) {
75              System.err.println("current thread is interrupted");
76              System.exit(1);     // Stop the program with a non-zero exit code
77          }
78      }
79
80      if(testTime(v)) {
81          petrinet.fire(v);
82
83          // Checks for threads queued up in sensibilized transitions to wake them up with the
    ↪ established policy.
84          Matrix sensibilized = petrinet.getSensibilized();
85
86          Matrix queued = conditionQueues.queuedUp();
87          Matrix and = sensibilized.arrayTimes(queued); //  'and' '&'
88
89          int m = result(and); // sensibilized and queued transitions
90
91
92          if(m > 0) {
93              int choice = policy.fireChoice(and);
94              // release
95              conditionQueues.getQueued().get(choice).release();
96          }else {
97              exitMonitor();
98          }
99          return true;
100     } else {
101             petrinet.setWorkingVector(v, (double) Thread.currentThread().getId());
102             exitMonitor();
103             return false;
104     }
105 }
106
107 /*
108   * Test if the transition is "time enabled". Checking if the elapsed time since
109   * the sensibilization of the transition is greater than the alpha time.
110   * Returns true if the transition is "time enabled", if it's not, returns false and save's the
    ↪ remaining time.
111   *
112   * @param v: firing vector
113   * @return true if the transition is "time enabled", false otherwise
114   */
115 private boolean testTime(Matrix v)
116 {
117     long time = System.currentTimeMillis();
```

```
118       long alpha = (long) petrinet.getAlphaTimes().get(0, getIndex(v));
119       long initTime = (long) petrinet.getSensibilizedTime().get(0, getIndex(v));
120       if (alpha < (time - initTime) || alpha == 0) {
121         return true;
122       } else {
123         setTimeLeft(Thread.currentThread().getId(), alpha - (time - initTime));
124         return false;
125       }
126     }
127
128     /*
129       * *************************
130       * **** PUBLIC  METHODS ****
131       * *************************
132       */
133
134     public void printDeadThreads()
135     {
136       System.out.println("Dead threads: " + deadThreads + "/14");
137     }
138
139     public void catchMonitor() throws InterruptedException {
140       mutex.acquire();
141     }
142
143     public void exitMonitor() {
144       mutex.release();
145     }
146
147     /*
148       * Returns the number of enabled and queued transitions.
149       *
150       * @param and: matrix resulting from the 'and' operation between the sensibilized and
        ↪ queued transitions.
151       * @return the number of enabled and queued transitions.
152       */
153     public int result(Matrix and)
154     {
155       int m = 0;
156
157       for (int i = 0; i < and.getColumnDimension(); i++)
158         if (and.get(0, i) > 0)
159           m++;
160
161       return m;
162     }
163
```

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

4.4.5. 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,
     ↪ 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 " + getThreadName() + ": started run()");
48        while (this.monitor.getPetriNet().getCompletedInvariants() < 200)
49        {
50          this.firingVector = transitions.get(transitionCounter);
51          if (monitor.fireTransition(firingVector))
52          {
53            nextTransition();
54          } else
55          {
56            long sleepTime;
57            try
58            {
59              sleepTime = this.monitor.getTimeLeft(Thread.currentThread().getId());
60            }
61            catch (Exception e)
62            {
63              sleepTime = 0;
64            }
65            if(!(this.monitor.getPetriNet().getCompletedInvariants() < 200))
66            {
67              try
68              {
69                TimeUnit.MILLISECONDS.sleep(sleepTime);
70              } catch(Exception e)
71              {
72                System.err.println("interrupted while sleeping");
73                System.exit(1);     // Stop the program with a non-zero exit code
74              }
75            }
76          }
77        }
78        this.monitor.addDeadThreads();
79        System.out.println("Thread " + getThreadName() + ": finished run()");
80      }
81  }
```

4.4.6. 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.List;
3
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 matriz;
13    private Matrix workingVector;
14    private Matrix alphaTime;
15    private Matrix transitionCounter;
16    private Matrix sensibilizedTime;
17    public ArrayList<Integer> tInvariantsAux;
18    private ArrayList<String> firedSequence;
19    private int[] invariantCounting;
20    private static int completedInvariants;
21    private String CurrentRoute;
22    private String allTransitionsPrint;
23
24    private final double[][] matrixIndicence = {
25        // T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13
26        { -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
27        { -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
28        { 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
29        { 1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
30        { -1, -1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, -1, 1 },
31        { 0, 1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
32        { 0, 0, 1, 1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0 },
33        { 0, 0, 0, 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0 },
34        { 0, 0, 0, 0, 0, -1, 0, 1, 0, 0, 0, 0, 0, 0 },
35        { 0, 0, 0, 0, 1, 0, -1, 0, 0, 0, 0, 0, 0, 0 },
36        { 0, 0, 0, 0, -1, -1, 1, 1, 0, 0, 0, 0, 0, 0 },
37        { 0, 0, 0, 0, 0, 1, 0, -1, 0, 0, 0, 0, 0, 0 },
38        { 0, 0, 0, 0, 0, 0, 1, 1, -1, -1, 0, 0, 0, 0 },
39        { 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1, 0, 0, 0 },
40        { 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 1, 1, 0, 0 },
41        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1, 0 },
42        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, -1, 0 },
43        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1 },
44        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 1 }
45    };
46
47    private final double[][] tInvariant = {
48        { 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1 }, // T1 T3 T5 T7 T9 T11 T12 T13
49        { 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1 }, // T1 T3 T5 T7 T8 T10 T12 T13
50        { 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1 }, // T1 T3 T4 T6 T9 T11 T12 T13
51        { 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1 }, // T1 T3 T4 T6 T8 T10 T12 T13
52        { 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1 }, // T0 T2 T5 T7 T9 T11 T12 T13
53        { 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1 }, // T0 T2 T5 T7 T8 T10 T12 T13
54        { 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1 }, // T0 T2 T4 T6 T9 T11 T12 T13
55        { 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1 }, // T0 T2 T4 T6 T8 T10 T12 T13
56    };
57    private final double[][] pInvariant = {
```

```
58        // 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
59        { 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
60        { 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 2 1,3,
61        { 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 3 2,5
62        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0 }, // 4 13,14,15
63        { 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 5 7,9
64        { 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 }, // 6 8,11-----------------
65        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 }, // 7 9,10,11
66        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 }, // 8 17,18---------------
67        { 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 }, // 9 3,4,5,17
68    };
69
70    private final double[][] bIncidence = {
71        // T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13
72        { 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
73        { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
74        { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
75        { 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
76        { 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },
77        { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
78        { 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 },
79        { 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
80        { 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 },
81        { 0, 0, 0, 0, 0, 0, 1, 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, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 },
84        { 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 },
85        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 },
86        { 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 },
87        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },
88        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },
89        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
90        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 }
91    };
92    // 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
93    private final double[] initialMarking = { 1, 1, 1, 0, 3, 0, 0, 1, 1, 0, 2, 0, 0, 0, 1, 0, 0, 0, 1 };
94
95    // 0 1 2 3 4 5 6 7 8 9 10 11 12 13
96    private final double[] aTimes = { 0, 0, 100, 100, 0, 0, 200, 200, 0, 0, 200, 200, 0, 100 };
97    public List<Integer> tInvariantSum;
98
99    public PetriNet()
100   {
101     this.CurrentRoute = "";
102     this.completedInvariants = 0;
103     this.invariantCounting = new int[8];
104     this.incidence = new Matrix(matrixIndicence);
```

```
105     this.backwardsIncidence = new Matrix(bIncidence);
106     this.currentMarking = new Matrix(initialMarking, 1);
107     this.sensibilizedTransitions = new Matrix(1, incidence.getColumnDimension());
108     this.workingVector = new Matrix(1, incidence.getColumnDimension());
109     this.alphaTime = new Matrix(aTimes, 1);
110     this.sensibilizedTime = new Matrix(1, incidence.getColumnDimension());
111     this.firedSequence = new ArrayList<String>();
112     this.transitionCounter = new Matrix(1, 14);
113     this.tInvariantSum = new ArrayList<>();
114     this.tInvariantsAux = new ArrayList<>();
115     for (int j = 0; j < 14; j++)
116     {
117       transitionCounter.set(0, j, 0.0);
118     }
119   }
120
121
122   /*
123     * ********************
124     * Public Methods *
125     ********************
126     */
127
128   /*
129     * This method calculates the fundamental ecuation of the petrinet: mi+1= mi+W*s
130     * where mi is the current marking, W is the incidence matrix and s is the firing vector.
131     *
132   * @param v: firing vector
133   * @return fundamental equation
134   */
135   public Matrix fundamentalEquation(Matrix v)
136   {
137     return (currentMarking.transpose().plus(incidence.times(v.transpose()))).transpose();
138     // (mi + w * s) transpose
139   }
140
141   public boolean fundamentalEquationTest(Matrix firingVector)
142   {
143     matriz = fundamentalEquation(firingVector);
144     for (int i = 0; i < this.matriz.getColumnDimension(); i++)
145       if (this.matriz.get(0, i) < 0) {
146         return false;
147       }
148     return true;
149   }
150
151   /*
```

```
152     * Idea: compare the current marking to the marking requested for each transition.
153     * Current marking: vector with the individual marking of all the places
154     * Incidence matrix: columns = transitions | rows = places
155     * Then, if a transition of the current marking has fewer tokens than those requested by
      ↪ the transition, it cannot be fired.
156     *
157     * @param
158     * @return
159     */
160   void enableTransitions()
161   {
162     Long time = System.currentTimeMillis();// tiempo actual
163     for (int i = 0; i < backwardsIncidence.getColumnDimension(); i++)
164     {
165       boolean enabledTransition = true;
166       for (int j = 0; j < backwardsIncidence.getRowDimension(); j++)
167       {
168         if (backwardsIncidence.get(j, i) > currentMarking.get(0, j))
169         {
170           enabledTransition = false;
171           break;
172         }
173       }
174       if (enabledTransition)
175       {
176         sensibilizedTransitions.set(0, i, 1);
177         sensibilizedTime.set(0, i, (double) time);
178       } else
179       {
180         sensibilizedTransitions.set(0, i, 0);
181       }
182
183     }
184     //System.out.println("Enabled transitions: " + getEnabledTransitionsInfo());
185   }
186
187   /*
188     * returns a string with the enabled transitions info.
189     *
190     * @return enabled transitions info
191     */
192   public String getEnabledTransitionsInfo()
193   {
194     String enabled = "";
195     for (int i = 0; i < 14; i++) {
196       if (sensibilizedTransitions.get(0, i) == 1)
197       {
```

```
198           enabled += ("T" + i + "  ");
199         }
200       }
201       return enabled;
202     }
203
204     public String getAllTransitionsPrint() {
205       return allTransitionsPrint;
206     }
207
208     /*
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 secuence 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       {
230         for (int j = 0; j < v.getColumnDimension(); j++)
231         {
232           if (v.get(i, j) != 0.0)
233           {
234             transitionCounter.set(i, j, (transitionCounter.get(i, j) + v.get(i, j)));
235           }
236         }
237       }
238       System.out.println(transitionsCounterInfo());
239       System.out.println("Current marking:\n" + getMarkingInfo());
240       String lastTransition = getIndex(v) + "";
241       allTransitionsPrint += "T" + lastTransition;
242       followUp(lastTransition);
243
```

```
244   }
245
246   public String getMarkingInfo()
247   {
248     String marking = "P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16
      ↪ P17 P18\n";
249     for (int i = 0; i < currentMarking.getColumnDimension(); i++)
250     {
251       if (i < 10)
252       {
253         marking += ((int) currentMarking.get(0, i) + "  ");
254       } else
255       {
256         marking += ((int) currentMarking.get(0, i) + "   ");
257       }
258     }
259     return (marking + "\n");
260   }
261
262   public int getCompletedInvariants() {
263     return completedInvariants;
264   }
265
266   public int getValinvariantCounting(int i) {
267     return invariantCounting[i];
268   }
269
270   public String transitionsCounterInfo()
271   {
272     int totalCount = 0;
273     String arg = "Transitions:\n";
274     for (int i = 0; i < transitionCounter.getColumnDimension(); i++)
275     {
276       arg += ("T" + i + ": " + (int) transitionCounter.get(0, i) + " times\n");
277       totalCount += (int) transitionCounter.get(0, i);
278     }
279     arg += ("Total transitions: " +totalCount);
280     return arg;
281   }
282
283
284   /*
285     * Checks the 'state' of the transition that is going to be fired.
286     * 0 - No one is working on it. STATE = NONE
287     * 1 - Someone is working on it, but it is not the thread that is requesting it. STATE =
      ↪ OTHER
288     * 2 - The thread that is requesting it is already working on it. STATE = SELF
```

```
289     *
290     * @param v: firing vector
291     * @return
292     */
293
294   public int workingState(Matrix v)
295   {
296     int index = getIndex(v);
297
298     if (workingVector.get(0, index) == 0)
299       return 0;
300     else if (workingVector.get(0, index) != Thread.currentThread().getId())
301       return 1;
302     else
303       return 2;
304   }
305
306
307   /*
308     * Follows the transitions fired in order to check the T-invariants.
309     *
310     * @param lastTransition: last transition fired
311     * @return
312     */
313   public void followUp(String lastTransition)
314   {
315
316     if (!lastTransition.contains("13"))
317     {
318       CurrentRoute += lastTransition;
319     } else
320     {
321
322       // T1 T3 T5 T7 T9 T11 T12 T13
323       // T1 T3 T5 T7 T8 T10 T12 T13
324       // T1 T3 T4 T6 T9 T11 T12 T13
325       // T1 T3 T4 T6 T8 T10 T12 T13
326       // T0 T2 T5 T7 T9 T11 T12 T13
327       // T0 T2 T5 T7 T8 T10 T12 T13
328       // T0 T2 T4 T6 T9 T11 T12 T13
329       // T0 T2 T4 T6 T8 T10 T11 T13 */
330
331       if (CurrentRoute.contains("135791112"))
332       {
333         invariantCounting[0] += 1;
334         completedInvariants++;
335       } else if (CurrentRoute.contains("135781012"))
```

```
336         {
337           invariantCounting[1] += 1;
338           completedInvariants++;
339         } else if (CurrentRoute.contains("134691112"))
340         {
341           invariantCounting[2] += 1;
342           completedInvariants++;
343         } else if (CurrentRoute.contains("134681012"))
344         {
345           invariantCounting[3] += 1;
346           completedInvariants++;
347         } else if (CurrentRoute.contains("025791112"))
348         {
349           invariantCounting[4] += 1;
350           completedInvariants++;
351         } else if (CurrentRoute.contains("025781012"))
352         {
353           invariantCounting[5] += 1;
354           completedInvariants++;
355         } else if (CurrentRoute.contains("024691112"))
356         {
357           invariantCounting[6] += 1;
358           completedInvariants++;
359         } else if (CurrentRoute.contains("024681012"))
360         {
361           invariantCounting[7] += 1;
362           completedInvariants++;
363         } else
364         {
365
366           for (int i = 0; i < 8; i++)
367           {
368             System.out.println(i + " T-invariant appears " + getValinvariantCounting(i) + "
    ↪ times.");
369           }
370           System.out.println("Error in T-Invariant: " + CurrentRoute);
371         }
372
373         for (int i = 0; i < 8; i++)
374         {
375           System.out.println(i + " T-invariant appears " + getValinvariantCounting(i) + "
    ↪ times.");
376         }
377
378         CurrentRoute = "";
379       }
380
```

```
381   }
382
383   public void tInvariantsInfo()
384   {
385     for (int i = 0; i < 8; i++)
386     {
387       System.out.println(i + " T-invariant appears " + getValinvariantCounting(i) + "
      ↪ times.");
388     }
389   }
390
391   /*
392     * This method checks the P-invariants using the current marking of the places.
393     *
394     * @param
395     * @return
396     */
397   public void testPInvariants()
398   {
399     boolean pInv0, pInv1, pInv2, pInv3, pInv4, pInv5, pInv6, pInv7, pInv8;
400     pInv0 = (currentMarking.get(0, 0) + currentMarking.get(0, 3) + currentMarking.get(0,
      ↪ 5)
401         + currentMarking.get(0, 6) + currentMarking.get(0, 9) + currentMarking.get(0, 11)
402         + currentMarking.get(0, 12) + currentMarking.get(0, 13) + currentMarking.get(0,
      ↪ 15)
403         + currentMarking.get(0, 16) + currentMarking.get(0, 17)) == 1;
404     pInv1 = (currentMarking.get(0, 1) + currentMarking.get(0, 3)) == 1;
405     pInv2 = (currentMarking.get(0, 2) + currentMarking.get(0, 5)) == 1;
406     pInv3 = (currentMarking.get(0, 14) + currentMarking.get(0, 13) +
      ↪ currentMarking.get(0, 15)) == 1;
407     pInv4 = (currentMarking.get(0, 7) + currentMarking.get(0, 9)) == 1;
408     pInv5 = (currentMarking.get(0, 8) + currentMarking.get(0, 11)) == 1;
409     pInv6 = (currentMarking.get(0, 9) + currentMarking.get(0, 10) +
      ↪ currentMarking.get(0, 11)) == 2;
410     pInv7 = (currentMarking.get(0, 17) + currentMarking.get(0, 18)) == 1;
411     pInv8 = (currentMarking.get(0, 3) + currentMarking.get(0, 4) + currentMarking.get(0,
      ↪ 5)
412         + currentMarking.get(0, 17)) == 3;
413
414     if (!(pInv0 && pInv1 && pInv2 && pInv3 && pInv4 && pInv5 && pInv6 && pInv7
      ↪ && pInv8))
415     {
416       System.out.println("Error in a p-invariant.");
417     }
418   }
419
420
```

```
421   public Matrix getCurrentMarking() {
422     return currentMarking;
423   }
424
425   public void setCurrentMarking(Matrix currentMarking) {
426     this.currentMarking = currentMarking;
427   }
428
429   public Matrix getSensibilized() {
430     return sensibilizedTransitions;
431   }
432
433   /*
434     * Returns the index of the transition that is going to be fired.
435     *
436     * @param v: firing vector
437     * @return index of the transition
438   */
439   public int getIndex(Matrix v)
440   {
441     int index = 0;
442     for (int i = 0; i < v.getColumnDimension(); i++)
443     {
444       if (v.get(0, i) == 1)
445         break;
446       else
447         index++;
448     }
449     return index;
450   }
451
452   public Matrix getAlphaTimes() {
453     return alphaTime;
454   }
455
456   public Matrix getSensibilizedTime() {
457     return sensibilizedTime;
458   }
459
460   public void setWorkingVector(Matrix firingVector, double value)
461   {
462     this.workingVector.set(0, getIndex(firingVector), value);
463   }
464
465   public Matrix getIncidenceMatrix() {
466     return this.incidence;
467   }
```

```
468 }
```

4.4.7. 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.

5. Análisis del funcionamiento

La red de Petri implementada con transiciones inmediatas y temporales es la que se aprecia en la figura siguiente, y es la que se utiliza para los análisis.

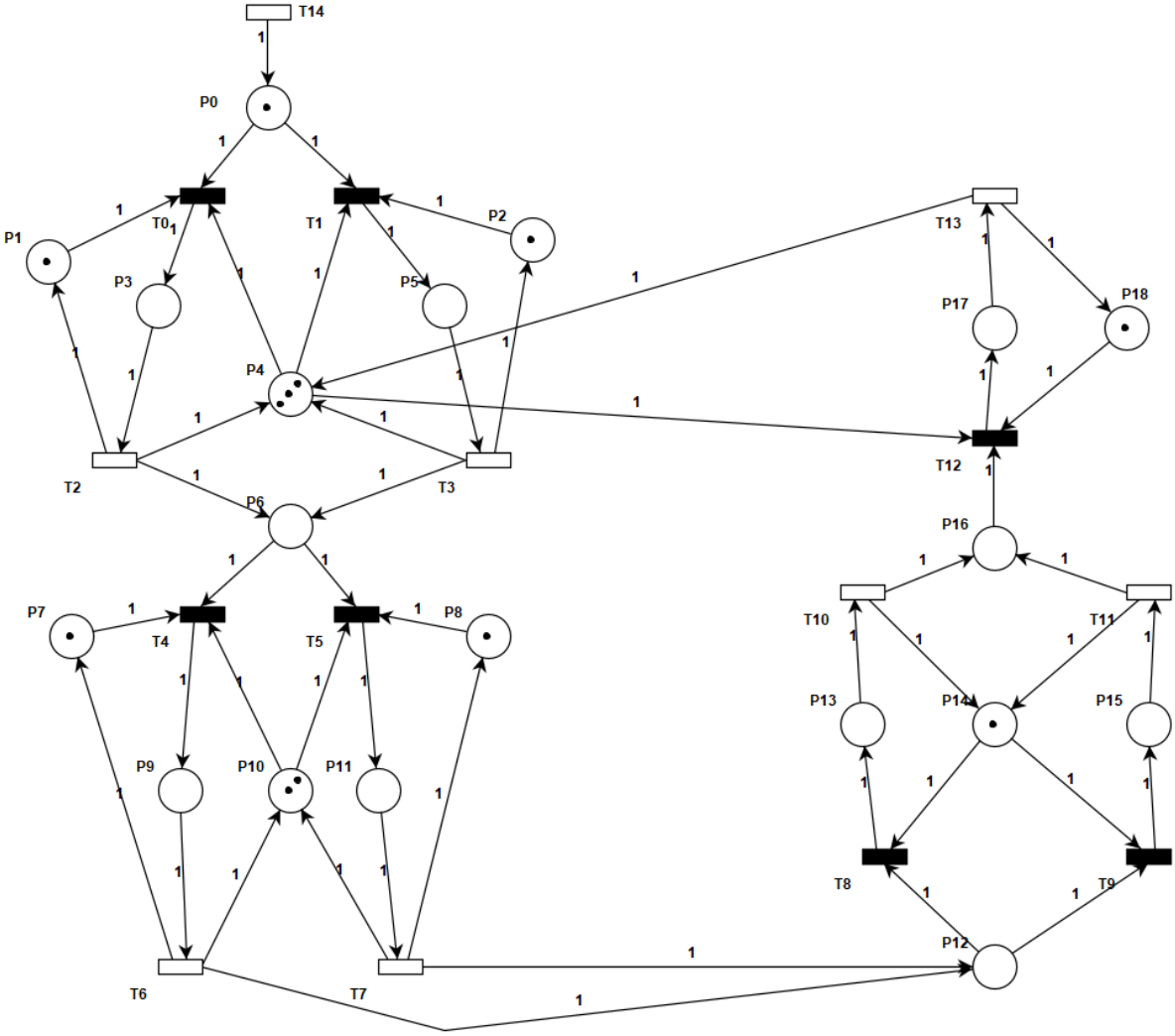

Figura 19: Red de Petri implementada

5.1. Análisis temporal

Para este análisis temporal se presentan dos casos teóricos:

- Funcionamiento totalmente secuencial.
- Funcionamiento completamente paralelo.

Luego, finalmente, el análisis práctico se hace con dos casos posibles de asignación de hilos al sistema modelado:

- 7 hilos de ejecución (elección del grupo).
- 14 hilos de ejecución (estudio según bibliografía).

5.1.1. Funcionamiento totalmente secuencial

Se tienen los siguientes tiempos para las transiciones temporalizadas:

- $t(T_2) = 100\ mS$
- $t(T_3) = 100\ mS$
- $t(T_6) = 200\ mS$
- $t(T_7) = 200\ mS$
- $t(T_{10}) = 200\ mS$
- $t(T_{11}) = 200\ mS$
- $t(T_{13}) = 100\ mS$

De donde se tiene que la suma total (para un invariante de transición) es:

$$t(T_2) + t(T_6) + t(T_{10}) + t(T_{13}) = 600\ ms$$

De esta manera, para cumplir con los requerimientos de la consigna, al ejecutar 200 invariantes de transición se obtendría un tiempo de ejecución total de $200 * 600\ ms = 120000\ ms = 120\ seg$.

5.1.2. Completamente paralelo

La **máxima** temporización de las transiciones seleccionadas, es de 200 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:

$$200\ ms * 200\ imagenes = 40000\ ms = 40\ seg$$

5.1.3. Ejecuciones del programa, análisis práctico

Se puede observar que el tiempo para completar los 200 invariantes es de aproximadamente 120 segundos, corroborando el correcto funcionamiento del programa, y que en un punto tiene sentido, al haber un solo token disponible para todos los hilos del sistema, es lógico que se encuentre al límite del tiempo de secuencializacion y no superar este límite de 120 segundos de secuencializacion o tiempo máximo.

Ahora, al utilizar 14 hilos, es decir, uno por transición, el tiempo de ejecución es en promedio 1 segundo más grande. Esto demuestra que la asignación de responsabilidades de

los hilos (7 hilos, planteado anteriormente) es correcto, y que no es necesario el uso de tantos hilos. En todo caso, la performance del software podría verse beneficiada en el caso de más de un token o más de una imagen a procesar, ya que podría haber concurrencia y/o paralelismo efectivo.

5.1.4. Estudio de la linealidad en las variaciones temporales

Para hacer este estudio, se realiza una ejecución con 7 hilos, y el cumplimiento de 100 invariantes de transición. La idea es comprobar si hay linealidad en la variación del tiempo de ejecución una vez modificados los tiempos de los pares de transiciones que representan el mismo recurso de la red.

Se muestran a continuación las gráficas obtenidas:

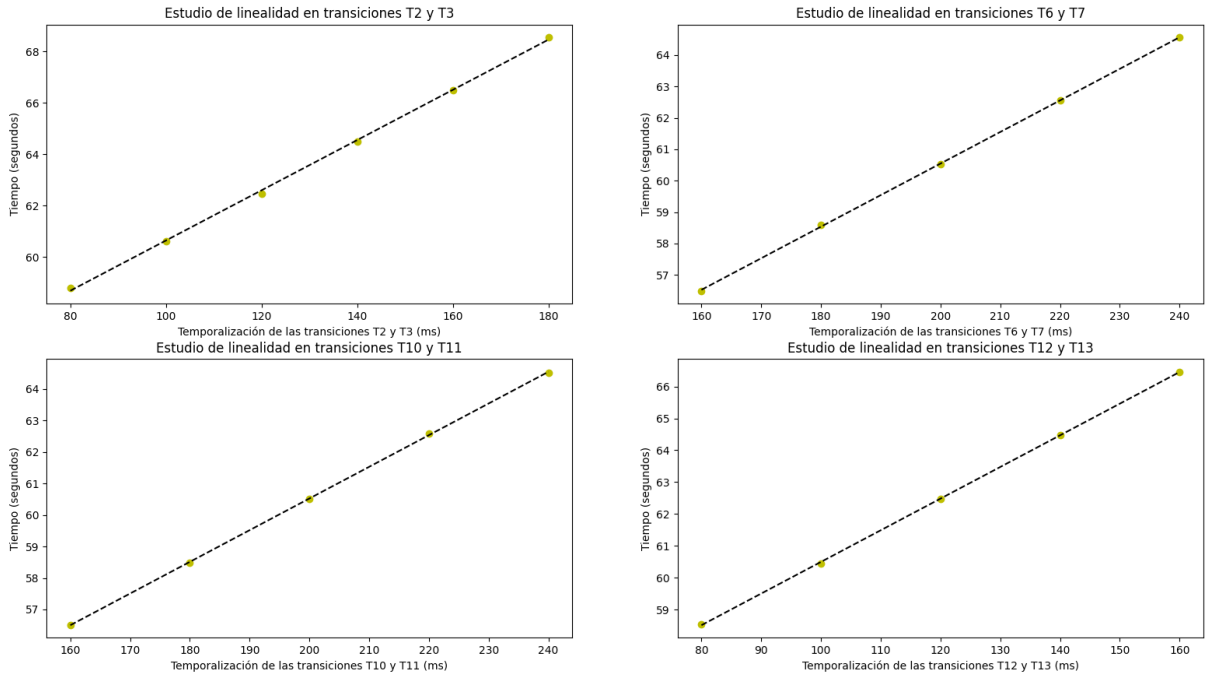

Figura 20: Influencia en el tiempo de las transiciones.

Rescatando lo obtenido anteriormente, puede verse en lo planteado con los siguientes items un cierto comportamiento no del todo deseado.

- 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

Resulta que con el escenario planteado, y los tiempos muy bajos de temporización de las transiciones (mostrados a continuación), el tiempo de decisión del scheduler pesa, no notandose así mejora alguna.

- $t(T_2) = 5\ mS$
- $t(T_3) = 5\ mS$
- $t(T_6) = 10\ mS$
- $t(T_7) = 10\ mS$
- $t(T_{10}) = 10\ mS$
- $t(T_{11}) = 10\ mS$
- $t(T_{13}) = 5\ mS$

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 %.

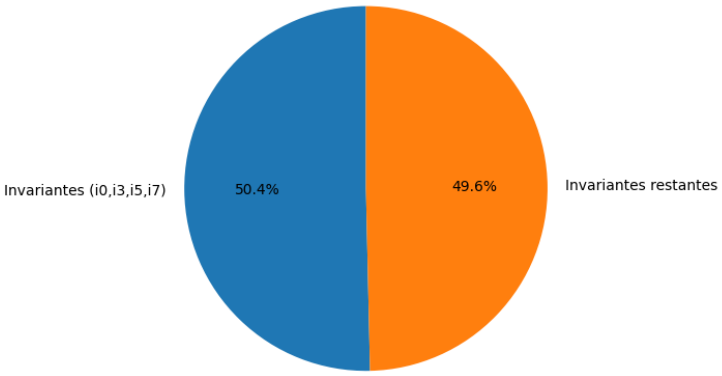

Figura 21: Promedio distribución 50-50

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.

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 %

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 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 %.

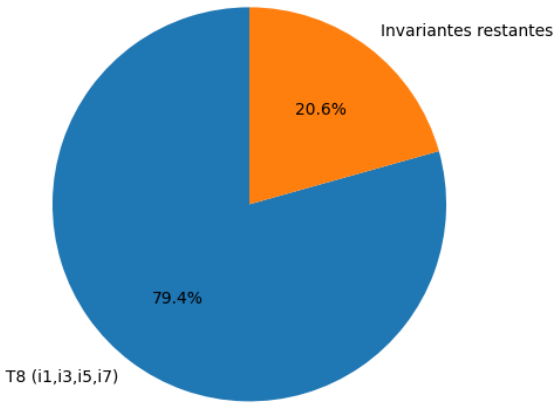

Figura 22: Promedio distribución 80-20

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 α que representa el tiempo de sensibilizado. Como se pidió que las transiciones no se de sensibilicen, entonces se elimina el llamado parámetro β (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_Redes_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