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

Programación Concurrente

Trabajo Final Integrador

Preparado por:

Borgatello, Ignacio

Profesores:

Ventre, Luis

Ludemann, Mauricio

Índice

1. Propiedades

La Red de Petri que modela el sistema de procesamiento de imágenes se presenta en la Figura 1. En este modelo, las plazas P1, P3, P5, P7, P9, P11 y P15 representan los recursos compartidos dentro del sistema. La plaza P0 actúa como una plaza idle y corresponde al buffer de entrada de imágenes al sistema. En las plazas P2 y P4 se lleva a cabo la carga de imágenes en el contenedor destinado al procesamiento, mientras que la plaza P6 representa dicho contenedor. Las plazas P8, P10, P12 y P13 modelan los estados en los que se realiza el ajuste de calidad de las imágenes, un proceso que debe completarse en dos etapas secuenciales. La plaza P14 representa el contenedor donde se almacenan las imágenes con la calidad mejorada, listas para ser recortadas a su tamaño definitivo. Este recorte se realiza en las plazas P16 y P17, y las imágenes procesadas se depositan en estado final en la plaza P18. Finalmente, la plaza P19 modela el proceso de exportación de las imágenes fuera del sistema.

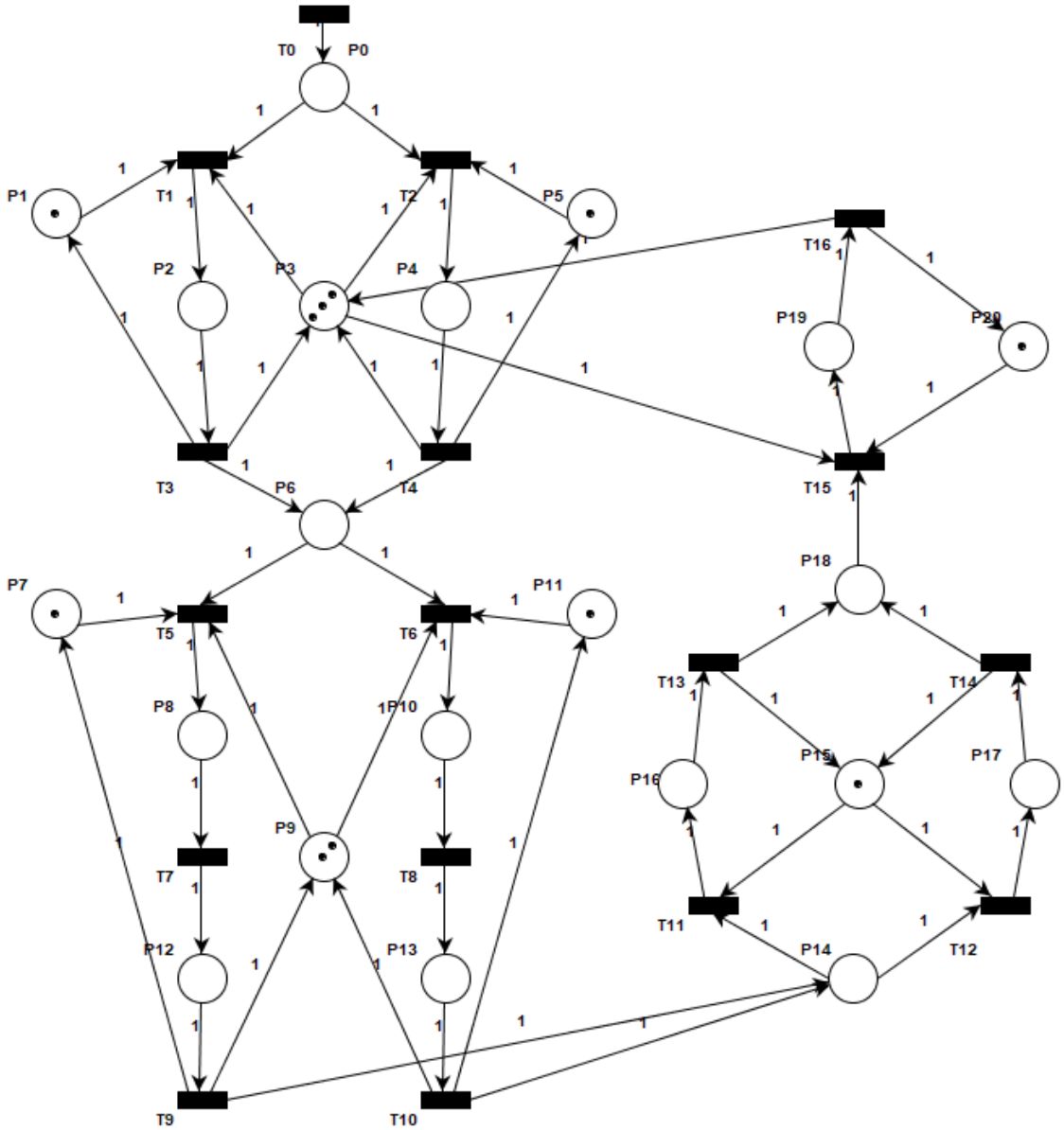


Figura 1. Red de Petri que modela el sistema de imágenes.

La plaza P0 depende del disparo de la transición T0, la cual modela la entrada de imágenes al sistema desde el exterior. Esto indica que se trata de un evento externo, y, por lo tanto, la red es no autónoma. Se observa que todas las transiciones tienen más de un lugar de entrada, lo que implica que la red no puede representar una máquina de estados.

Además, la plaza P0 cuenta con dos transiciones de salida, la plaza P18 tiene dos transiciones de entrada, y las plazas P6, P9 y P15 presentan dos transiciones de entrada y dos de salida. Debido a esto, la red no cumple con las condiciones de un grafo de marcado. También se observa que las plazas P0, P3, P6, P9, P14 y P18 pueden contener un marcado mayor a 1, lo que significa que la red no es segura.

Existen conflictos estructurales en las plazas P0, P6 y P15, ya que todas tienen dos transiciones de salida. En el caso de la plaza P15, el conflicto es efectivo porque esta plaza no puede tener más de un token. Por otro lado, los conflictos en las plazas P0, P6 y P14 pueden no ser efectivos si ingresan varias imágenes al sistema, es decir, si la transición T0 se dispara más de una vez.

Las plazas P1, P5, P7, P11 y P20 actúan como limitadores, restringiendo las plazas P2, P4, P8, P10, P12, P13 y P19 a un máximo de un token. Por su parte, las plazas P3, P9 y P15 son recursos compartidos, siendo la plaza P15 un caso particular que modela una exclusión mutua. En consecuencia, las secuencias de disparo T1T3, T2T4, T5T7T9, T6T8T10, T11T13, T12T14 y T15T16 representan secciones críticas del sistema.

2. Invariantes

Para definir los invariantes de plaza y de transición de la red, se utilizó el software TINA.
Nota 1: Para realizar el análisis de propiedades estructurales en la herramienta, fue necesario unir la transición {T16} con la plaza {P0} y eliminar la transición {T0}.
Nota 2: Para llevar a cabo el análisis de invariantes en la herramienta, se marcaron todas las transiciones como inmediatas (no temporizadas).

En la Figura 2 se observa un token inicial en la plaza P0. Este token tiene una función auxiliar y se utiliza para mantener la red desbloqueada durante el análisis de invariantes. A partir de la red de Petri modificada, se determinaron los siguientes invariantes de plaza:

N°	Invariante
IP_1	P1 + P2 = 1
IP_2	P4 + P5 = 1
IP_3	P2 + P3 + P4 + P19 = 3
IP_4	P7 + P8 + P12 = 1
IP_5	P10 + P11 + P13 = 1
IP_6	P8 + P9 + P10 + P12 + P13 = 2

IP_7	P15 + P16 + P17 = 1
IP_8	P19 + P20 = 1
IP_9	P0 + P2 + P4 + P6 + P8 + P10 + P12 + P13 + P14 + P16 + P17 + P18 + P19 = n

Tabla 1. Invariantes de plaza

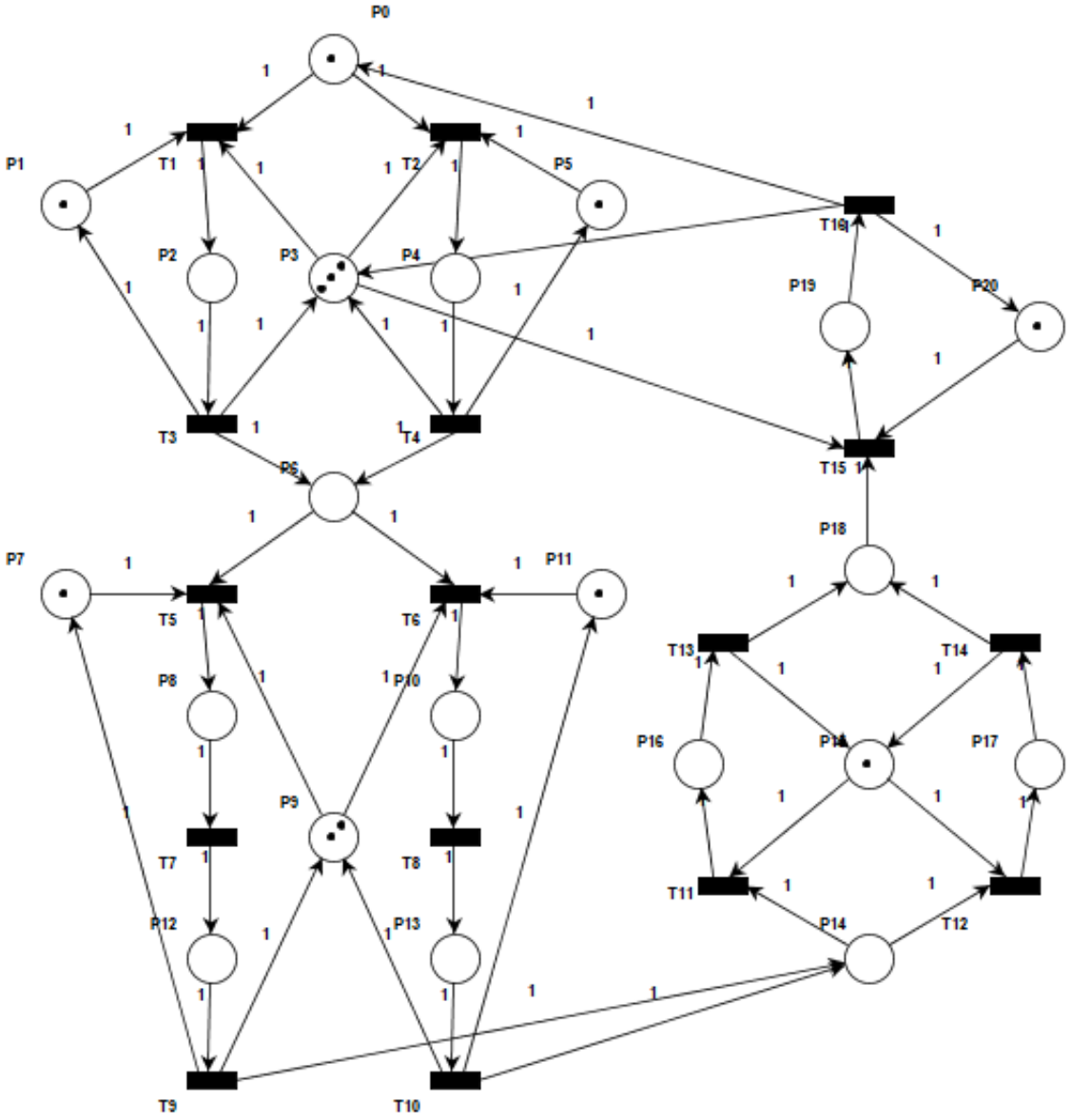


Figura 2. Red de Petri modificada para cálculo de invariantes

El noveno invariante de plaza, para la red modificada, depende del marcado inicial en P0, el cual es: donde n corresponde al marcado inicial en P0.

Los invariantes de transición determinados a partir de la red son:

N°	Invariante
IT_1	{T1, T3, T5, T7, T9, T11, T13, T15, T16}
IT_2	{T1, T3, T5, T7, T9, T12, T14, T15, T16}
IT_3	{T1, T3, T6, T8, T10, T11, T13, T15, T16}
IT_4	{T1, T3, T6, T8, T10, T12, T14, T15, T16}
IT_5	{T2, T4, T5, T7, T9, T11, T13, T15, T16}
IT_6	{T2, T4, T5, T7, T9, T12, T14, T15, T16}
IT_7	{T2, T4, T6, T8, T10, T11, T13, T15, T16}
IT_8	{T2, T4, T6, T8, T10, T12, T14, T15, T16}

Tabla 2. Invariantes de transición

Para modelar la evolución de la red y desarrollar el modelo del sistema en el lenguaje de programación Java, se obtuvo la matriz de incidencia de la red y, a partir de ella, la ecuación de estado. La matriz de incidencia fue determinada utilizando la herramienta PETRINATOR, y se presenta en la Tabla 3.

A partir de la matriz de incidencia, es posible calcular el estado (marcado) de la red para cualquier disparo de transición, utilizando la ecuación de estado (1):

$$M_{i+1} = M_i + i * \alpha \qquad (1)$$

Donde M_i corresponde a un marcado de la red, M a la matriz de incidencia (Tabla 1), M_{i+1} al marcado luego de realizar el disparo y α al vector de disparo; es un vector columna de la forma:

$$\alpha = [t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}]^T$$

donde cada t_i corresponde a la transición disparada en el estado M_i para llegar al estado M_{i+1}.

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16
P0	1	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P1	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
P2	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
P3	0	-1	-1	1	1	0	0	0	0	0	0	0	0	0	0	-1	1
P4	0	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
P6	0	0	0	1	1	-1	-1	0	0	0	0	0	0	0	0	0	0
P7	0	0	0	0	0	-1	0	0	0	1	0	0	0	0	0	0	0
P8	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0	0	0
P9	0	0	0	0	0	-1	-1	0	0	1	1	0	0	0	0	0	0
P10	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0	0
P11	0	0	0	0	0	0	-1	0	0	0	1	0	0	0	0	0	0
P12	0	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0
P13	0	0	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0
P14	0	0	0	0	0	0	0	0	0	1	1	-1	-1	0	0	0	0
P15	0	0	0	0	0	0	0	0	0	0	0	-1	-1	1	1	0	0
P16	0	0	0	0	0	0	0	0	0	0	0	1	0	-1	0	0	0
P17	0	0	0	0	0	0	0	0	0	0	0	0	1	0	-1	0	0
P18	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	-1	0
P19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-1
P20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	1

Tabla 3. Matriz de incidencia de la red de Petri

3. Cantidad máxima de hilos simultáneos

Utilizando el algoritmo desarrollado en Ref/1/, se calculó la cantidad máxima de hilos activos simultáneos para el sistema modelado por la red. Para este análisis, se empleó la red modificada (Figura 3), ya que el algoritmo requiere los invariantes de plaza y de transición, los cuales fueron obtenidos a partir de dicha red.

En primer lugar, se calcularon los invariantes de transición. Posteriormente, se identificaron los conjuntos de plazas asociadas a cada invariante de transición, es decir, las plazas que participan en la secuencia de disparo correspondiente a cada invariante. Los conjuntos determinados son:

Conjunto	Plazas
PI_1	{P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P16, P18, P19, P20}
PI_2	{P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P17, P18, P19, P20}
PI_3	{P0, P1, P2, P3, P6, P9, P10, P11, P13, P14, P15, P16, P18, P19, P20}
PI_4	{P0, P1, P2, P3, P6, P9, P10, P11, P13, P14, P15, P17, P18, P19, P20}
PI_5	{P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P16, P18, P19, P20}
PI_6	{P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P17, P18, P19, P20}
PI_7	{P0, P3, P4, P5, P6, P9, P10, P11, P13, P14, P15, P16, P18, P19, P20}
PI_8	{P0, P3, P4, P5, P6, P9, P10, P11, P13, P14, P15, P17, P18, P19, P20}

Tabla 4. Conjuntos de plazas asociadas a los invariantes de transición

Luego, en base a la descripción de las plazas presentada anteriormente, se determinaron los siguientes conjuntos de plazas que no son plazas de acción (Tabla 5):

Conjunto	Plazas
Plazas Idle	P0, P6, P14, P18
Plazas de recursos	P1, P3, P5, P7, P9, P11, P20
Plazas de restricción	P15

Tabla 5. Conjuntos de plazas idle, recursos y restricción

Al sustraer los conjuntos de plazas idle, de recursos y de restricción de las plazas asociadas a cada invariante de transición, se obtuvieron los conjuntos de plazas de acción correspondientes a cada invariante:

Conjunto	Plazas
PA_1	{P2, P8, P12, P16, P19}
PA_2	{P2, P8, P12, P17, P19}
PA_3	{P2, P10, P13, P16, P19}
PA_4	{P2, P10, P13, P17, P19}
PA_5	{P4, P8, P12, P16, P19}
PA_6	{P4, P8, P12, P17, P19}
PA_7	{P4, P10, P13, P16, P19}
PA_8	{P4, P10, P13, P17, P19}

Tabla 6. Conjuntos de plazas de acción de cada invariante

Finalmente, el conjunto de plazas de acción está compuesto por todas las plazas que pertenecen al conjunto de plazas de acción de todos los invariantes de transición:

PA	P2, P4, P8, P10, P12, P13, P16, P17, P19}

La cantidad máxima de hilos activos se determina tomando el valor máximo de la suma de tokens en las plazas de acción, considerando todos los marcados posibles. Dado que los marcados posibles dependen del marcado inicial y que el número de tokens en la plaza P0 es variable (representando las imágenes que ingresan al sistema), se calcularon los marcados posibles (es decir, se generaron los árboles de alcanzabilidad) para diferentes cantidades de tokens iniciales en la plaza P0. Se evaluaron marcados iniciales con valores de tokens en la plaza P0 que varían de 1 a 10. Los resultados obtenidos se presentan en la Tabla 7.

m_0(P0)	Max(Suma(PA))
1	1
2	2
3	3
4	4
5	5
6	6
7	6

8	6
9	6
10	6

Tabla 7. Máximo de suma de tokens de plazas de acción para diferentes marcados iniciales en P0.

Se puede observar que la suma máxima de marcas en las plazas de acción aumenta conforme incrementa el marcado inicial en P0, alcanzando un máximo de 6. Esto implica que, si el invariante de plaza IP9 es menor a 6, el sistema (la red) tiene recursos no utilizados o desperdiciados. En cambio, si IP9 es mayor o igual a 6, el sistema puede operar a su máxima capacidad sin desperdiciar recursos. Por lo tanto, el número máximo de hilos activos simultáneos para el sistema es 6.

Cabe señalar que, como se mencionó previamente, el análisis se realizó con la red modificada, lo que excluye la transición T0, que modela el evento externo de ingreso de imágenes al sistema. Si se incluye la ocurrencia de este evento en el modelo y se le asigna una temporalidad (como se muestra en la Figura 1), se debe considerar dicha acción como un hilo adicional. En este caso, al incluir T0, la cantidad máxima de hilos activos simultáneos sería de 7.

4. Responsabilidad de hilos y segmentos

De acuerdo con el procedimiento desarrollado por Micolini y Ventre, se analizó la red con el objetivo de asignar responsabilidades específicas a los hilos y segmentar la red. Se puede observar que la red presenta varios forks (puntos de conflicto) y joins (puntos de unión). El primer fork ocurre en la plaza P0, de la cual surgen dos segmentos. Estos segmentos se unen en un join en la plaza P6, que a su vez funciona como un nuevo fork. Nuevamente, se generan dos segmentos que se unen en la plaza P14. La plaza P14 también es un fork de dos segmentos, pero estos deben ejecutarse en exclusión mutua debido a la presencia de la plaza P15. Finalmente, estos segmentos se unen en un último join en la plaza P18, que da lugar al último segmento de la red. Por lo tanto, los segmentos de la red son Sa, Sb, Sc, Sd, Se, Sf, Sg, Sh. Las plazas asociadas a cada segmento son:

PS_a	{P0}
PS_b	{P2}
PS_c	{P4}
PS_d	{P8, P13}
PS_e	{P10, P13}
PS_f	{P16}
PS_g	{P17}
PS_h	{P19}

Y las transiciones asociadas a cada segmento son:

TS_a	{T0}
TS_b	{T1, T3}
TS_c	{T2, T4}
TS_d	{T5, T7, T9}
TS_e	{T6, T8, T10}
TS_f	{T11, T13}
TS_g	{T12, T14}
TS_h	{T15, T16}

Los roles de los segmentos son:

- S_a: Ingreso de imágen
- S_b y S_c: Carga de la imágen
- S_d y S_e: Filtrado de la imágen
- S_f y S_g: Recorte de la imágen
- S_h: Exportado de la imágen

Se utilizó el mismo procedimiento empleado para determinar la cantidad máxima de hilos simultáneos para calcular la cantidad máxima de hilos por segmento. Es decir, se aplicó el algoritmo para encontrar el máximo de hilos activos simultáneos a cada segmento individualmente. Los resultados obtenidos se presentan en la Tabla 8.

Segmento	Max(Suma(PS))
S_a	1
S_b	1
S_c	1
S_d	1
S_e	1
S_f	1
S_g	1
S_h	1

Tabla 8. Máximo de suma de plazas de acción para cada segmento (m_0(P0)=15)

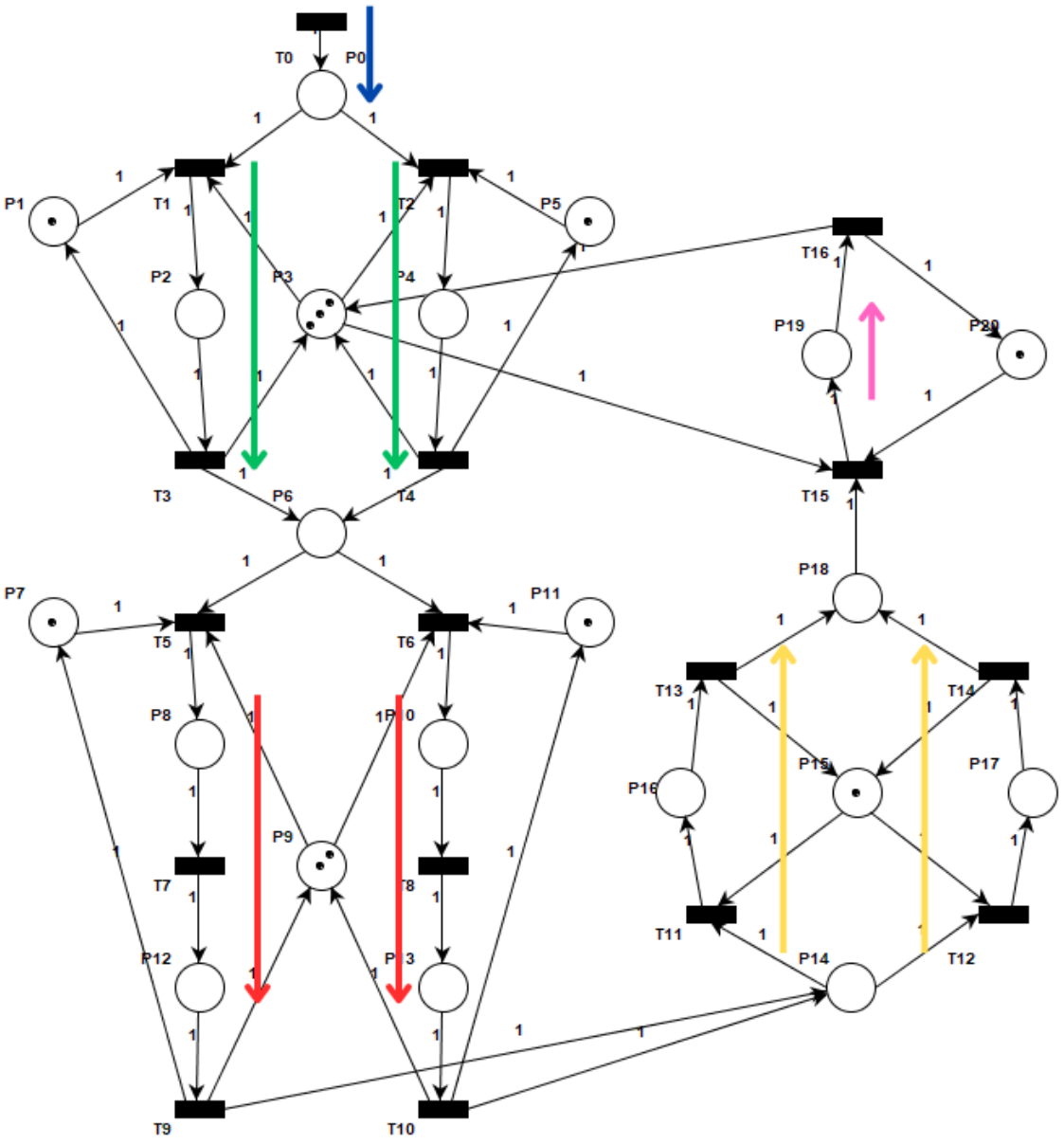


Figura 4. Red de petri segmentada en hilos de acción.

5. Implementación

5.1 Proceso

La clase Proceso es una clase abstracta que representa un concepto genérico de "proceso" que puede ejecutarse en un hilo (Runnable). Su propósito principal es servir como base para crear clases concretas que implementan la lógica de ejecución de un proceso con transiciones específicas, sincronizado mediante un monitor.

```
package src;

public abstract class Proceso implements Runnable {

   private final String nombre;            // Nombre del proceso.
    protected boolean stop;                    // Variable que indica si el proceso debe detenerse.
    protected final long tiempo;              // Tiempo que el proceso debe esperar al realizar la tarea (en milisegundos).
   protected final int[] transiciones;     // Arreglo de transiciones que el proceso debe manejar.
    protected int index;                          // Índice que indica la transición actual que el proceso disparará.
    private final int[] cuenta;               // Cuenta de disparos de cada transicion.
     protected final Monitor monitor;           // Objeto src.Monitor que sincroniza las transiciones.

   /**
    * Constructor de la clase src.Proceso.
    *
    * @param nombre       Nombre del proceso.
      * @param transiciones Arreglo de transiciones que el proceso debe manejar.
    * @param tiempo       Tiempo de espera entre tareas (en milisegundos).
       * @param monitor          Objeto src.Monitor para coordinar las transiciones.
    */
    public Proceso(String nombre, int[] transiciones, long tiempo, Monitor monitor) {
       this.nombre = nombre;
       this.stop = false;
       this.transiciones = transiciones;
       this.index = 0;
       this.tiempo = tiempo;
       this.cuenta = new int[transiciones.length];
       this.monitor = monitor;
   }

   public String getNombre() {
       return nombre;
   }
```

```
    public void setStop(boolean stop) {
        this.stop = stop;
    }

    public boolean isStop() {
        return stop;
    }

    public int[] getCuenta() {
        return cuenta;
    }

    public void setCuenta(int transicion) {
        this.cuenta[transicion]++;
    }
}
```

5.2 Importador, Cargador, Filtro, Redimensionador, Exportador

Todas estas clases son una implementación concreta de la clase abstracta Proceso. Su función principal es ejecutar de manera cíclica un conjunto de transiciones a través de un objeto Monitor, respetando un intervalo de tiempo entre cada transición. Cabe destacar que en el caso del importador, recibe un parámetro adicional que es la cantidad de imágenes a importar dentro del sistema.

```
package src;

import java.util.concurrent.TimeUnit;

public class Importador extends Proceso {

    private final int cantMaxima;

    /**
     * Constructor de la clase src.Importador.
     *
     * @param nombre       Nombre del proceso.
       * @param transiciones Lista de transiciones que este proceso debe
manejar.
       * @param tiempo          Intervalo de espera entre cada disparo de
transición (en milisegundos).
     * @param monitor      Objeto src.Monitor utilizado para sincronizar las
transiciones.
     * @param cantMaxima   Cantidad maxima de imagenes a importar dentro del
sistema.
     */
    public Importador(String nombre, int[] transiciones, long tiempo, Monitor
monitor, int cantMaxima) {
           // Llama al constructor de la clase padre (src.Proceso) para
inicializar los atributos comunes.
        super(nombre, transiciones, tiempo, monitor);
        this.cantMaxima = cantMaxima;
    }
```

```
    /**
     * Metodo que ejecuta el proceso en un hilo separado.
          * Este metodo es llamado automáticamente cuando se ejecuta
`Thread.start()`.
     */
    @Override
    public void run() {
        while (getCuenta()[0]<this.cantMaxima && !isStop()) {
            try {
                // Dispara la transición correspondiente al índice actual en
el arreglo de transiciones.
                this.monitor.dispararTransicion(transiciones[index]);

                  // Llevamos la cuenta de cuantos disparos se hizo en cada
transicion.
                setCuenta(index);

                      // Actualiza el índice para avanzar a la siguiente
transición de forma cíclica.
                index = (index + 1) % transiciones.length;

                TimeUnit.MILLISECONDS.sleep(tiempo);
            } catch (InterruptedException | RuntimeException e) {
                setStop(true);
            }
        }
        System.out.println(getNombre());
        for (int i=0;i< getCuenta().length;i++){
                                System.out.printf("Transicion:  %d  Disparos:
%d\n",transiciones[i],getCuenta()[i]);
        }
    }
}
```

5.3 Colas

La clase Colas implementa un sistema de sincronización que utiliza **Semáforos** para gestionar la espera y liberación de hilos en función de transiciones específicas. Su objetivo principal es coordinar múltiples hilos que necesitan acceso a recursos compartidos o deben realizar tareas sincronizadas asociadas con diferentes colas o transiciones.

```
private final int[] listaBloqueadas;           // Arreglo que indica si hay
hilos bloqueados en cada cola (1 = bloqueado, 0 = no bloqueado).
private final ArrayList<Semaphore> colas;      // Lista de semáforos, donde
cada semáforo representa una cola.

/**
* Constructor de la clase src.Colas.
* Inicializa la cantidad de colas y crea una lista de semáforos asociada a
cada transición.
*
* @param n Número de transiciones (y por lo tanto, número de colas).
*/
```

```
public Colas(int n) {
    // Cantidad de colas, equivalente al número de transiciones.
    this.colas = new ArrayList<>(n);
    this.listaBloqueadas = new int[n];

    // Inicializa los semáforos para cada cola con el valor inicial en 0.
    for (int i = 0; i < n; i++) {
        this.colas.add(new Semaphore(0));
    }

    // Inicializa la lista de bloqueadas con todos los valores en 0 (ningún
hilo bloqueado inicialmente).
    Arrays.fill(this.listaBloqueadas, 0);
}
```

5.4 Política

La clase Políticas implementa la lógica para determinar cómo seleccionar transiciones en el sistema. Su objetivo principal es definir diferentes políticas para elegir qué transición debe dispararse, considerando factores como balanceo de disparos, priorización de segmentos, y las transiciones habilitadas.

```
private final int cantTransiciones; // Número total de transiciones en el
sistema.
private final String tipo;              // Politica implementada durante la
ejecucion del programa.
                                        // BALANCEADA o PRIORITARIA
private final String segmento;          // Segmento de la Etapa 3 que se
prioriza.
                                        // DERECHA o IZQUIERDA
private final double prioridad;

/**
* Constructor de la clase src.Politicas.
*
* @param cantTransiciones Número total de transiciones en el sistema.
*/
public Politicas(int cantTransiciones, String tipo, String segmento, double
prioridad) {
    this.cantTransiciones = cantTransiciones;
        if  (!Objects.equals(tipo,  "BALANCEADA")  &&  !Objects.equals(tipo,
"PRIORITARIA")){
        this.tipo = "BALANCEADA";
    } else {
        this.tipo = tipo;
    }
      if  (!Objects.equals(segmento,  "DERECHA")  &&  !Objects.equals(segmento,
"IZQUIERDA")){
        this.segmento = "";
    } else {
        this.segmento = segmento;
    }
    if (prioridad<0 || prioridad>1){
```

```
        this.prioridad = 0;
    }else {
        this.prioridad = prioridad;
    }
}
```

5.5 Monitor

La clase Monitor implementa una estructura sincronizada para la gestión de una red de Petri, manejando disparos de transiciones y controlando el acceso concurrente mediante semáforos. Esta clase combina lógica de sincronización, control de políticas, manejo de tiempos, y registro de disparos para garantizar un comportamiento seguro y eficiente en sistemas concurrentes.

```
private final Semaphore mutex;           // Semáforo binario para garantizar
la exclusión mutua.
private final PetriNet petriNet;         // Instancia de la red de Petri que
se gestionará.
private final Colas colas;                 // Estructura para gestionar las
colas de espera de los hilos.
private final Politicas politicas;          // Define las políticas para
seleccionar qué transición disparar.
private final double[] cantDisparos;     // Contador para registrar cuántas
veces se dispara cada transición.
private final Log log;

/**
* Constructor del src.Monitor.
* Inicializa los componentes necesarios para gestionar la red de Petri.
*/
public Monitor(Log log) {
    this.mutex = new Semaphore(1);
    this.petriNet = new PetriNet();
    this.colas = new Colas(this.petriNet.getCantTransiciones());
                                        this.politicas            =            new
Politicas(this.petriNet.getCantTransiciones(),"BALANCEADA","",0);
                                    //this.politicas            =            new
src.Politicas(this.petriNet.getCantTransiciones(),"PRIORITARIA","DERECHA",0
.8);
    this.cantDisparos = new double[this.petriNet.getCantTransiciones()];
    this.log = log;
}
```

5.6 PetriNet

La clase modela el comportamiento dinámico de una Red de Petri, permitiendo representar su estructura, evaluar su estado (marcado) y gestionar disparos de transiciones bajo restricciones de habilitación por tokens y ventanas temporales.

```
/**
* Metodo que intenta disparar una transición en la red.
*
* @param transicion Índice de la transición a disparar.
```

```
* @return `true` si el disparo fue exitoso, `false` si no se pudo disparar.
*/
public boolean disparar(int transicion) {

   // Verifica si la transición es válida.
   if (transicion < 0 || transicion >= cantTransiciones) {
       throw new IllegalArgumentException("Índice de transición inválido");
   }

    // Verifica si la transición está habilitada por cantidad de tokens en
las plazas.
   if (estaHabilitada(transicion)) {

       // Verifica si la transición está dentro de la ventana de tiempos.
       long tiempo = System.currentTimeMillis();
       if (validarTiempos(transicion,tiempo)) {

           // Crea el vector de disparo para la transición solicitada.
           crearVectorTransicion(transicion);

           // Arrays auxiliares para el cálculo del nuevo marcado.
           int[] temp = new int[cantPlazas];
           int[] nuevoMarcado = new int[cantPlazas];

             // Calcula el nuevo marcado usando la ecuación fundamental de
estado: m' = m + C * v.
           // temp = C * v
           for (int i = 0; i < cantPlazas; i++) {
               temp[i] = 0;
               for (int j = 0; j < cantTransiciones; j++) {
                   temp[i] += matrizIncidencia[i][j] * vector[j];
               }
           }

            // Calcula el marcado resultante sumando el marcado actual con
el resultado.
           // nuevoMarcado = m + temp
           for (int i = 0; i < cantPlazas; i++) {
               nuevoMarcado[i] = marcado[i] + temp[i];
           }

           // Verifica que el marcado resultante no tenga valores negativos
(inviables).
           for (int i = 0; i < cantPlazas; i++) {
               if (nuevoMarcado[i] < 0) {
                   return false;
               }
           }

           // Verifica los invariantes de plaza.
           if (verificarInvariantesPlaza(nuevoMarcado)) {
               throw new RuntimeException("Se corrompieron los invariantes de
plaza.");
```

```
            }

             // Actualiza el marcado y las transiciones habilitadas según los
tokens disponibles.
            actualizarHabilitadas(nuevoMarcado);

            return true;
        }
    }
    return false;
}

/**
* Actualiza las transiciones habilitadas según los tokens disponibles en
las plazas.
*
* @param marcado Marcado actual de la red.
*/
public void actualizarHabilitadas(int[] marcado) {
    this.marcado = marcado; // Actualiza el marcado.

    for (int i = 0; i < cantTransiciones; i++) {

        // Registra el valor anterior de las transiciones habilitadas.
        transicionesHabilitadasAnteriores[i] = transicionesHabilitadas[i];

        // Recalcula las transiciones habilitadas por tokens.
        transicionesHabilitadas[i] = 1; // Habilita inicialmente.
        for (int j = 0; j < cantPlazas; j++) {
            if (matrizIncidencia[j][i] == -1 && marcado[j] == 0) {
                    transicionesHabilitadas[i] = 0; // Deshabilita si no hay
tokens suficientes.
                break;
            }
        }

        if (matrizTiempos[i][0]!=-1) {
                        if  (transicionesHabilitadasAnteriores[i]  ==  0  &&
transicionesHabilitadas[i] == 1) {
                // Transición habilitada por primera vez
                vectorTiempos[i] = System.currentTimeMillis();
                   } else if (transicionesHabilitadasAnteriores[i] == 1 &&
transicionesHabilitadas[i] == 0) {
                // Transición deshabilitada
                vectorTiempos[i] = 0L;
            } else if (i == 0) {
                // Caso particular de T0
                vectorTiempos[i] = System.currentTimeMillis();
            }
        }
    }
}
```

5.7 Main

La clase Main coordina la ejecución de procesos concurrentes que representan distintas etapas de un sistema de producción (como importar, cargar, filtrar, redimensionar y exportar). Su objetivo es asegurar que se cumpla un número máximo de disparos de transiciones en una red de Petri, deteniendo los procesos de forma ordenada al finalizar.

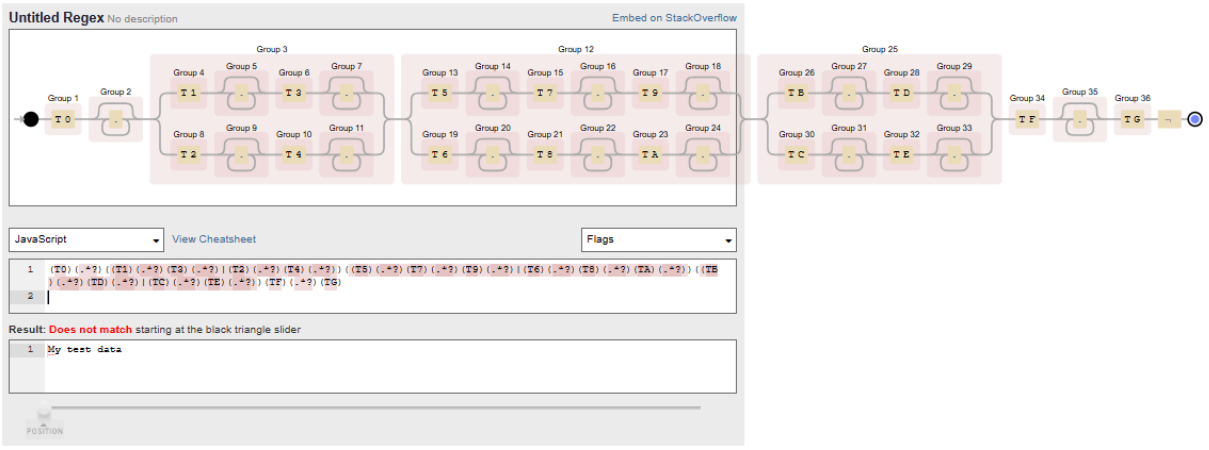
```
public class Main {
   public static void main(String[] args) {
        // Formato de fecha
        SimpleDateFormat formatter = new SimpleDateFormat("EEE MMM dd
HH:mm:ss z yyyy");

        // Hora de inicio
        Date start = new Date();
        System.out.println("Se inició la ejecución de la red: " +
formatter.format(start));

        Log log;
        String path = "log/log.txt";
        try {
            log = new Log(path);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

        // Cantidad de invariantes de transicion a cumplir durante la
ejecucion del programa.
        int cantMaxima = 1000;

        // Crear la instancia del src.Monitor que gestiona las transiciones.
        Monitor monitor = new Monitor(log);

        // Definición de las transiciones asociadas a cada proceso.
        Proceso[] procesos = getProcesos(cantMaxima,monitor);

        // Crear y ejecutar un hilo para cada proceso.
        Thread[] hilos = new Thread[procesos.length];
        for (int i = 0; i < procesos.length; i++) {
            hilos[i] = new Thread(procesos[i]);
            hilos[i].start();
        }

        // Verificamos que se cumplan todos los invariantes de transicion.
        while (monitor.getDisparos()[16]!=cantMaxima){
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        // Interrumpimos los procesos.
```

```
        for (int i = 0; i < procesos.length; i++) {
            hilos[i].interrupt();
        }

        // Esperamos a todos los procesos.
        for (int i = 0; i < procesos.length; i++) {
            try {
                hilos[i].join();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        // Obtener y mostrar la cantidad de disparos por transición.
        double[] disparos = monitor.getDisparos();
        System.out.println("\nResultados:");
        System.out.printf("\nCantidad de invariantes: %d\n",cantMaxima);
        for (int i = 0; i < disparos.length; i++) {
            System.out.printf("La transición %d se disparó %d veces.%n", i,
(int) disparos[i]);
        }

        try {
            log.closeFile();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

        // Hora de finalización
        Date end = new Date();
        System.out.println("Se finalizó la ejecución de la red: " +
formatter.format(end));
    }
```

6. Análisis de Invariantes de Transición

Debemos asegurar que luego de una ejecución de n transiciones, una red de petri siempre cumpla con sus invariantes de transición. Para ello, fue necesario realizar un script en el lenguaje Python, que lea de un archivo .txt las líneas de caracteres y las procese con una expresión regular. A continuación, se detalla la expresión regular utilizada y el script generado para el análisis.

```
(T0)(.*?)((T1)(.*?)(T3)(.*?)|(T2)(.*?)(T4)(.*?))((T5)(.*?)(T7)(.*?)(T9)(.*?)|
(T6)(.*?)(T8)(.*?)(TA)(.*?))((TB)(.*?)(TD)(.*?)|(TC)(.*?)(TE)(.*?))(TF)(.*?)(
TG)
```




```python
import re

def reemplazar_numeros_por_letras(archivo_entrada, archivo_salida):
    # Definir los reemplazos
    reemplazos = {
        "10": "A",
        "11": "B",
        "12": "C",
        "13": "D",
        "14": "E",
        "15": "F",
        "16": "G"
    }

    # Abrir el archivo de entrada y leer el contenido
    with open(archivo_entrada, "r") as archivo:
        contenido = archivo.read()

    # Realizar los reemplazos en el contenido
    for numero, letra in reemplazos.items():
        contenido = contenido.replace(numero, letra)

    # Escribir el contenido modificado en el archivo de salida
    with open(archivo_salida, "w") as archivo:
        archivo.write(contenido)

# Llamada a la función (asegurate de cambiar los nombres de los archivos
según corresponda)
reemplazar_numeros_por_letras("log/log.txt", "log/log.txt")

# Abrir el archivo log.txt y leer la cadena
with open("log/log.txt", "r") as archivo_log:
```

```
    log = archivo_log.read().strip()  # Leemos y eliminamos posibles saltos
de línea

# Definir la expresión regular y la cadena de reemplazo
regex =
r'(T0)(.*?)((T1)(.*?)(T3)(.*?)|(T2)(.*?)(T4)(.*?))((T5)(.*?)(T7)(.*?)(T9)(.*?
)|(T6)(.*?)(T8)(.*?)(TA)(.*?))((TB)(.*?)(TD)(.*?)|(TC)(.*?)(TE)(.*?))(TF)(.*?
)(TG)'

# Usamos una cadena cruda para evitar problemas con las secuencias de escape
grupos =
r'\g<2>\g<5>\g<7>\g<9>\g<11>\g<14>\g<16>\g<18>\g<20>\g<22>\g<24>\g<27>\g<29>\
g<31>\g<33>\g<35>'

result = ""
found = 0
total = 0

# Bucle para aplicar la expresión regular y hacer los reemplazos
while True:
    result, found = re.subn(regex, grupos, log, count=0)
    if found == 0:
        break
    total += found
    log = result

# Mostrar resultados
print("Resultado:", result)
print("Número de reemplazos realizados:", total)

# Mostrar resultados
if not result:  # Si el resultado está vacío
    print("Éxito")
else:
    print("Falló")
```

7. Tests

Se evaluaron las funcionalidades principales de las clases críticas del sistema, asegurando que cada componente se comporte de acuerdo con los requisitos esperados. Se realizaron tests unitarios para verificar la correcta inicialización de objetos, la lógica detrás de la sincronización y el disparo de transiciones en la clase Monitor, hasta la implementación de políticas de selección de transiciones en Políticas.

tests in src: 21 total, 21 passed

864 ms

Collapse | Expand

ColasTest		227 ms
ColasTest.testAcquire	passed	117 ms
ColasTest.testAcquireAndRelease	passed	104 ms
ColasTest.testRelease	passed	1 ms
ColasTest.testConstructor	passed	5 ms
ImportadorTest		606 ms
ImportadorTest.testRunExecution	passed	606 ms
ImportadorTest.testGetNombre	passed	0 ms
ImportadorTest.testIsStop	passed	0 ms
ImportadorTest.testSetAndGetCuenta	passed	0 ms
MonitorTest		16 ms
MonitorTest.testMutex	passed	0 ms
MonitorTest.testDispararTransicionExito	passed	0 ms
MonitorTest.testTransicionesBloqueadasHabilitadas	passed	0 ms
MonitorTest.testDispararTransicionNoHabilitada	passed	16 ms
MonitorTest.testConstructor	passed	0 ms
PetriNetTest		0 ms
PetriNetTest.testDisparar	passed	0 ms
PetriNetTest.testDispararTransicionNoHabilitada	passed	0 ms
PetriNetTest.testActualizarHabilitadas	passed	0 ms
PoliticasTest		15 ms
PoliticasTest.constructorTest	passed	0 ms
PoliticasTest.seleccionarTransicionPrioritarialzquierdaTest	passed	0 ms
PoliticasTest.seleccionarTransicionPrioritariaDerechaTest	passed	0 ms
PoliticasTest.seleccionarTransicionSinHabilitadasTest	passed	0 ms
PoliticasTest.seleccionarTransicionBalanceadaTest	passed	15 ms

Generated by IntelliJ IDEA on 1/23/25, 10:24 AM

8. Resultados

Los tiempos elegidos inicialmente para las tareas son los siguientes:

Tareas	Tiempo [ms]
Importador	100
Cargador	100
Filtro	100
Redimensionador	100
Exportador	100

La figura a continuación muestra la ejecución del programa con 1000 invariantes como objetivo. La política seleccionada es BALANCEADA y las transiciones son inmediatas.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 500 veces.
La transición 12 se disparó 500 veces.
La transición 13 se disparó 500 veces.
La transición 14 se disparó 500 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 10:38:20 GMT-03:00 2025
```

Se observa que el balance entre las distintas ramas de la red se mantiene constante a lo largo de toda la ejecución.

A continuación, se muestra la ejecución del programa con 1000 invariantes como objetivo, la política seleccionada en PRIORITARIA, con un peso del 80% para el segmento izquierdo de la etapa 3, es decir, las transiciones T11 y T13 reciben el 80% de las imágenes, y las transiciones son inmediatas.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 833 veces.
La transición 12 se disparó 167 veces.
La transición 13 se disparó 833 veces.
La transición 14 se disparó 167 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 10:45:22 GMT-03:00 2025
```

Teniendo en cuenta la figura anterior,

$$\frac{833}{1000}100 \ = \ 83,3\%$$

Se observa que el segmento izquierdo de la etapa 3 recibe 83,3% de las imágenes que la red procesa a lo largo de toda la ejecución. Por lo tanto, concluimos que la política PRIORITARIA funciona correctamente. Por último, cabe destacar que el resto de las transiciones mantienen el balance en los disparos.

A continuación, se muestra la ejecución del programa con 1000 invariantes como objetivo, la política seleccionada en PRIORITARIA, con un peso del 70% para el segmento derecho de la etapa 3, es decir, las transiciones T12 y T14 reciben el 70% de las imágenes, y las transiciones son inmediatas.

Con esto se busca demostrar que la política PRIORITARIA funciona tanto para el segmento izquierdo como el derecho, y el peso a considerar en el porcentaje de imágenes que procesa un lado y el otro es a libre elección del usuario.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 501 veces.
La transición 6 se disparó 499 veces.
La transición 7 se disparó 501 veces.
La transición 8 se disparó 499 veces.
La transición 9 se disparó 501 veces.
La transición 10 se disparó 499 veces.
La transición 11 se disparó 232 veces.
La transición 12 se disparó 768 veces.
La transición 13 se disparó 232 veces.
La transición 14 se disparó 768 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 10:53:36 GMT-03:00 2025
```

Teniendo en cuenta la figura anterior,

$$\frac{768}{1000}100 \ = 76,8\%$$

Se observa que el segmento derecho de la etapa 3 recibe 76,8% de las imágenes que la red procesa a lo largo de toda la ejecución. Por lo tanto, concluimos que la política PRIORITARIA funciona correctamente para ambos segmentos de la etapa 3 de la red.

9. Tiempos

Los tiempos elegidos inicialmente para las transiciones son los siguientes:

Transición	(α, β) [ms]
T0	(100, 300)
T3	(100, 300)
T4	(100, 300)
T7	(100, 300)
T8	(100, 300)

T9	(100, 300)
T10	(100, 300)
T13	(100, 300)
T14	(100, 300)
T16	(100, 300)

La figura a continuación muestra la ejecución del programa con 1000 invariantes como objetivo. La política seleccionada es BALANCEADA y las transiciones son temporizadas según la tabla anterior.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 500 veces.
La transición 12 se disparó 500 veces.
La transición 13 se disparó 500 veces.
La transición 14 se disparó 500 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 11:08:40 GMT-03:00 2025
```

Nuevamente se observa que el balance entre las distintas ramas de la red se mantiene constante a lo largo de toda la ejecución, independientemente de las transiciones temporales.

A continuación, se muestra la ejecución del programa con 1000 invariantes como objetivo, la política seleccionada en PRIORITARIA, con un peso del 80% para el segmento izquierdo de la etapa 3, es decir, las transiciones T11 y T13 reciben el 80% de las imágenes, y las transiciones no son inmediatas.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 833 veces.
La transición 12 se disparó 167 veces.
La transición 13 se disparó 833 veces.
La transición 14 se disparó 167 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 11:15:45 GMT-03:00 2025
```

Teniendo en cuenta la figura anterior,

$$\frac{833}{1000}100 \ = \ 83,3\%$$

Se observa que el segmento izquierdo de la etapa 3 recibe 83,3% de las imágenes que la red procesa a lo largo de toda la ejecución. Por lo tanto, concluimos que la política PRIORITARIA funciona correctamente a pesar de incluir los tiempos en las transiciones. Por último, cabe destacar que el resto de las transiciones mantienen el balance en los disparos.

A continuación, se muestra la ejecución del programa con 1000 invariantes como objetivo, la política seleccionada en PRIORITARIA, con un peso del 80% para el segmento derecho de la etapa 3, es decir, las transiciones T12 y T14 reciben el 80% de las imágenes, y las transiciones no son inmediatas.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 167 veces.
La transición 12 se disparó 833 veces.
La transición 13 se disparó 167 veces.
La transición 14 se disparó 833 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 11:21:43 GMT-03:00 2025
```

Teniendo en cuenta la figura anterior,

$$\frac{833}{1000}100 \ = 83,3\%$$

Se observa que el segmento derecho de la etapa 3 recibe 83,3% de las imágenes que la red procesa a lo largo de toda la ejecución. Por lo tanto, concluimos que la política PRIORITARIA funciona correctamente para ambos segmentos de la etapa 3 de la red independientemente de las transiciones temporales.

9.1 Tiempos propuestos para las tareas

Es necesario hacer un análisis considerando paralelismo, y además mostrar la variación del tiempo total de ejecución con base en los tiempos elegidos para cada proceso. Para ello, se realizaron pruebas, donde en cada una de ellas tomamos como pivote cada proceso, probando con distintos tiempos y manteniendo el tiempo de los demás, y así observar cuánto afecta en el tiempo total de ejecución.

Este análisis se realizó bajo la política BALANCEADA y considerando transiciones inmediatas. Se realizaron 200 invariantes de transición en cada ejecución.

9.1.1 Prueba 1

N°	Importador	Resto de tareas	Tiempo total
1	1 ms	100 ms	45 s
2	100 ms	100 ms	44 s
3	200 ms	100 ms	45 s

9.1.2 Prueba 2

N°	Cargador	Resto de tareas	Tiempo total
4	1 ms	100 ms	45 s
5	100 ms	100 ms	44 s
6	200 ms	100 ms	45 s

9.1.3 Prueba 3

N°	Filtro	Resto de tareas	Tiempo total
7	1 ms	100 ms	44 s
8	100 ms	100 ms	46 s
9	200 ms	100 ms	63 s

9.1.4 Prueba 4

N°	Redimensionador	Resto de tareas	Tiempo total
10	1 ms	100 ms	45 s
11	100 ms	100 ms	45 s
12	200 ms	100 ms	45 s

9.1.5 Prueba 5

N°	Exportador	Resto de tareas	Tiempo total
13	1 ms	100 ms	35 s
14	100 ms	100 ms	45 s
15	200 ms	100 ms	85 s

Con base en los resultados obtenidos se presenta una propuesta de tiempos para cada tarea. Los tiempos asignados a cada una deben ser seleccionados considerando su impacto en el tiempo total de ejecución y priorizando la eficiencia de la red.

Tareas	Tiempo [ms]
Importador	100 ms
Cargador	100 ms
Filtro	80 ms
Redimensionador	100 ms
Exportador	50 ms

Con estos tiempos propuestos, volvemos a ejecutar el programa para 200 invariantes de transición a realizar y obtenemos el siguiente resultado:

```
Se inició la ejecución de la red: Thu Jan 23 12:31:50 GMT-03:00 2025
Resultados:
Cantidad de invariantes: 200
La transición 0 se disparó 200 veces.
La transición 1 se disparó 101 veces.
La transición 2 se disparó 99 veces.
La transición 3 se disparó 101 veces.
La transición 4 se disparó 99 veces.
La transición 5 se disparó 100 veces.
La transición 6 se disparó 100 veces.
La transición 7 se disparó 100 veces.
La transición 8 se disparó 100 veces.
La transición 9 se disparó 100 veces.
La transición 10 se disparó 100 veces.
La transición 11 se disparó 100 veces.
La transición 12 se disparó 100 veces.
La transición 13 se disparó 100 veces.
La transición 14 se disparó 100 veces.
La transición 15 se disparó 200 veces.
La transición 16 se disparó 200 veces.
Se finalizó la ejecución de la red: Thu Jan 23 12:32:18 GMT-03:00 2025
```

9.2 Tiempos propuestos para las transiciones

Este análisis se realizó bajo la política BALANCEADA y considerando transiciones temporales. Se realizaron 200 invariantes de transición en cada ejecución. Los tiempos de las tareas son los obtenidos en el apartado anterior.

Transición	(α1, β1) [ms]	Tiempo total	(α2, β2) [ms]	Tiempo total	(α3, β3) [ms]	Tiempo total
T0	(100, 300)	83 s	(100, 300)	79 s	(100, 300)	72 s
T3	(100, 300)		(100, 300)		(100, 300)	
T4	(100, 300)		(100, 300)		(100, 300)	
T7	**(1, 200)**		**(100, 300)**		**(200, 400)**	
T8	**(1, 200)**		**(100, 300)**		**(200, 400)**	
T9	**(1, 200)**		**(100, 300)**		**(200, 400)**	
T10	**(1, 200)**		**(100, 300)**		**(200, 400)**	
T13	(100, 300)		(100, 300)		(100, 300)	
T14	(100, 300)		(100, 300)		(100, 300)	
T16	(100, 300)		(100, 300)		(100, 300)	

Transición	(α1, β1) [ms]	Tiempo total	(α2, β2) [ms]	Tiempo total	(α3, β3) [ms]	Tiempo total
T0	(100, 300)	73 s	(100, 300)	76 s	(100, 300)	75 s
T3	(100, 300)		(100, 300)		(100, 300)	
T4	(100, 300)		(100, 300)		(100, 300)	
T7	(100, 300)		(100, 300)		(100, 300)	
T8	(100, 300)		(100, 300)		(100, 300)	
T9	(100, 300)		(100, 300)		(100, 300)	
T10	(100, 300)		(100, 300)		(100, 300)	
T13	(100, 300)		(100, 300)		(100, 300)	
T14	(100, 300)		(100, 300)		(100, 300)	
T16	**(1, 200)**		**(100, 300)**		**(200, 400)**	

Con base en los resultados obtenidos se presenta una propuesta de tiempos para cada transición.

Transición	(α, β) [ms]
T0	(100, 300)
T3	(100, 300)
T4	(100, 300)
T7	**(200, 400)**
T8	**(200, 400)**
T9	**(200, 400)**
T10	**(200, 400)**
T13	(100, 300)
T14	(100, 300)
T16	**(1, 200)**

Con estos tiempos propuestos, volvemos a ejecutar el programa para 200 invariantes de transición a realizar y obtenemos el siguiente resultado:

```
Se inició la ejecución de la red: Thu Jan 23 13:15:31 GMT-03:00 2025
Resultados:
Cantidad de invariantes: 200
La transición 0 se disparó 200 veces.
La transición 1 se disparó 100 veces.
La transición 2 se disparó 100 veces.
La transición 3 se disparó 100 veces.
La transición 4 se disparó 100 veces.
La transición 5 se disparó 100 veces.
La transición 6 se disparó 100 veces.
La transición 7 se disparó 100 veces.
La transición 8 se disparó 100 veces.
La transición 9 se disparó 100 veces.
La transición 10 se disparó 100 veces.
La transición 11 se disparó 100 veces.
La transición 12 se disparó 100 veces.
La transición 13 se disparó 100 veces.
La transición 14 se disparó 100 veces.
La transición 15 se disparó 200 veces.
La transición 16 se disparó 200 veces.
Se finalizó la ejecución de la red: Thu Jan 23 13:16:39 GMT-03:00 2025
```

10. Conclusiones

Se demostró la viabilidad de utilizar Redes de Petri para modelar y gestionar sistemas concurrentes. El diseño de la red permitió representar de manera eficiente la interacción entre hilos, asegurando la sincronización y el cumplimiento de invariantes estructurales y de transición. Estas propiedades destacan la robustez del sistema frente a posibles condiciones de carrera y conflictos de recursos compartidos.

Además, se realizaron análisis de las políticas de ejecución aplicadas. La política BALANCEADA logró mantener un equilibrio constante en la distribución de las tareas entre las ramas del sistema, mientras que la política PRIORITARIA permitió priorizar segmentos específicos de procesamiento, como lo demuestran los resultados obtenidos tanto en las transiciones inmediatas como en las temporizadas. Estas pruebas validaron la correcta implementación y versatilidad del sistema para adaptarse a distintas prioridades operativas.

Finalmente, el análisis de los tiempos asignados para cada tarea y transición subraya la importancia de un ajuste óptimo de parámetros para maximizar la eficiencia del sistema. Se determinó que ciertas configuraciones de tiempo influyen directamente en la duración total del procesamiento, lo que permitió proponer ajustes que optimizan la utilización de los recursos y reducen los tiempos sin comprometer la funcionalidad del sistema. Esto confirma la utilidad de este modelo en escenarios reales donde la concurrencia y la gestión eficiente de recursos son cruciales.

11. Referencias

/1/. O. Micolini, L.O. Ventre. Algoritmos para determinar cantidad y responsabilidad de hilos en sistemas embebidos modelados con Redes de Petri S3PR. Memorias del Congreso Argentino en Ciencias de la Computación - CACIC. 2021.

12. Repositorio de GitHub

nachoborgatello/Trabajo-Final-Programacion-Concurrente