Universidad
Nacional
de Córdoba

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 a la entrada de imágenes al sistema. En las plazas P2 y P4 se lleva a cabo la carga de imágenes destinadas al procesamiento, mientras que la plaza P6 representa un contenedor de esas imágenes. 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.

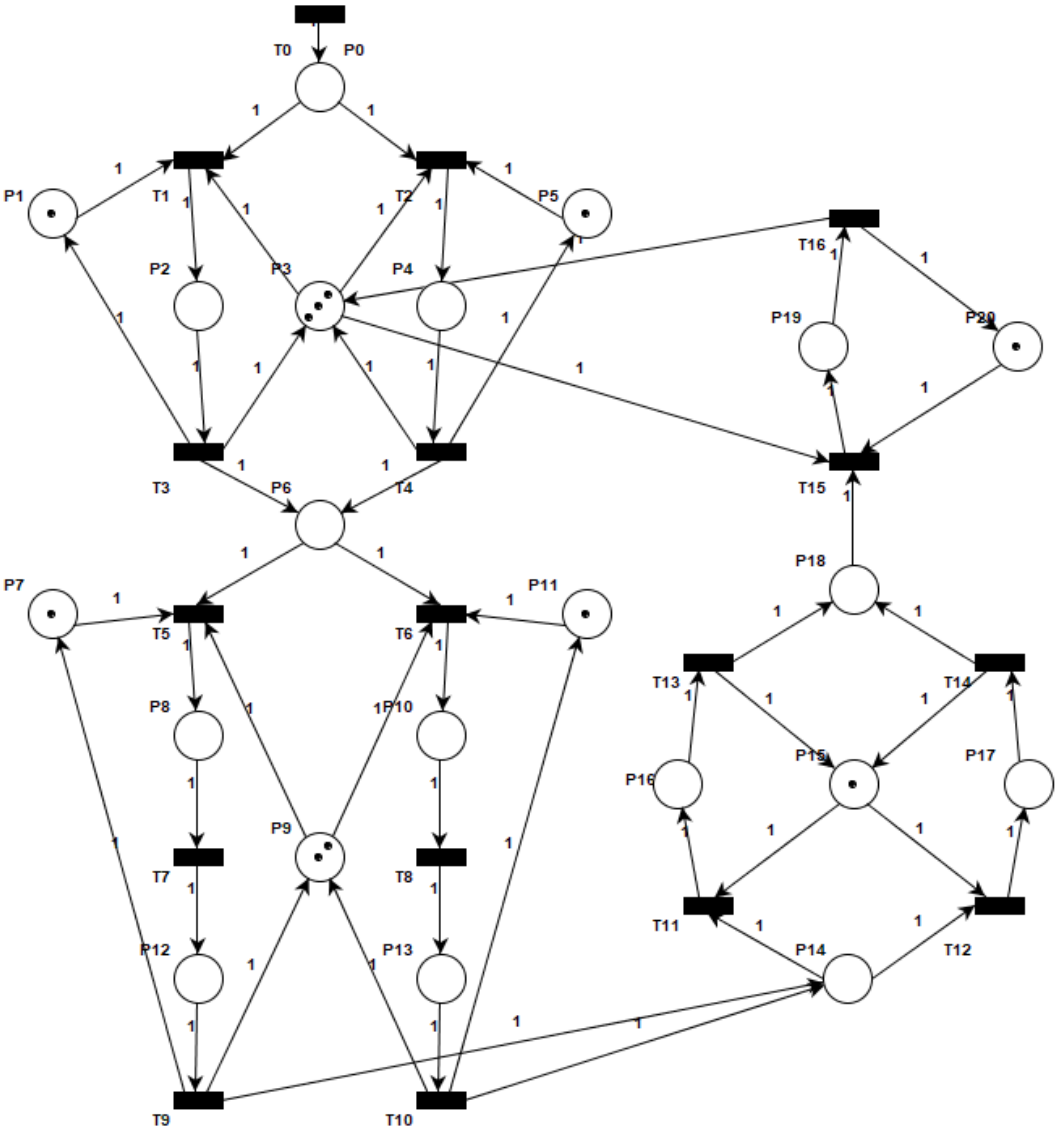


Figura 1. Red de Petri que modela el sistema de imágenes.

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.

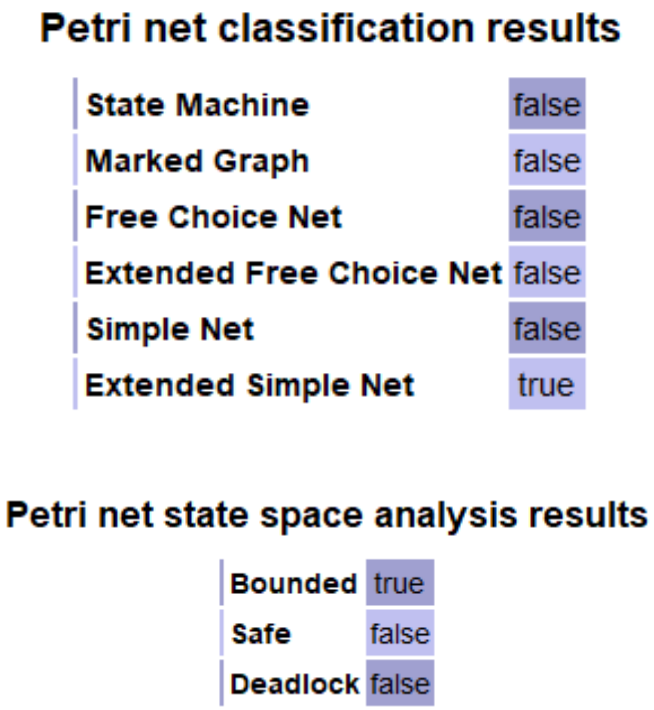


Figura 2. Clasificación de la Red de Petri con PIPE.

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 3 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:

P-Invariante	Invariante
1	P1 + P2 = 1
2	P4 + P5 = 1
3	P2 + P3 + P4 + P19 = 3
4	P7 + P8 + P12 = 1
5	P10 + P11 + P13 = 1
6	P8 + P9 + P10 + P12 + P13 = 2
7	P15 + P16 + P17 = 1
8	P19 + P20 = 1
9	P0 + P10 + P12 + P13 + P14 + P16 + P17 + P18 + P19 + P2 + P4 + P6 + P8 = n donde n es m0(P0)

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

T-Invariante	Invariante
1	{T1, T3, T5, T7, T9, T11, T13, T15, T16}
2	{T1, T3, T5, T7, T9, T12, T14, T15, T16}
3	{T1, T3, T6, T8, T10, T11, T13, T15, T16}
4	{T1, T3, T6, T8, T10, T12, T14, T15, T16}
5	{T2, T4, T5, T7, T9, T11, T13, T15, T16}
6	{T2, T4, T5, T7, T9, T12, T14, T15, T16}

7 {T2, T4, T6, T8, T10, T11, T13, T15, T16}

8 {T2, T4, T6, T8, T10, T12, T14, T15, T16}

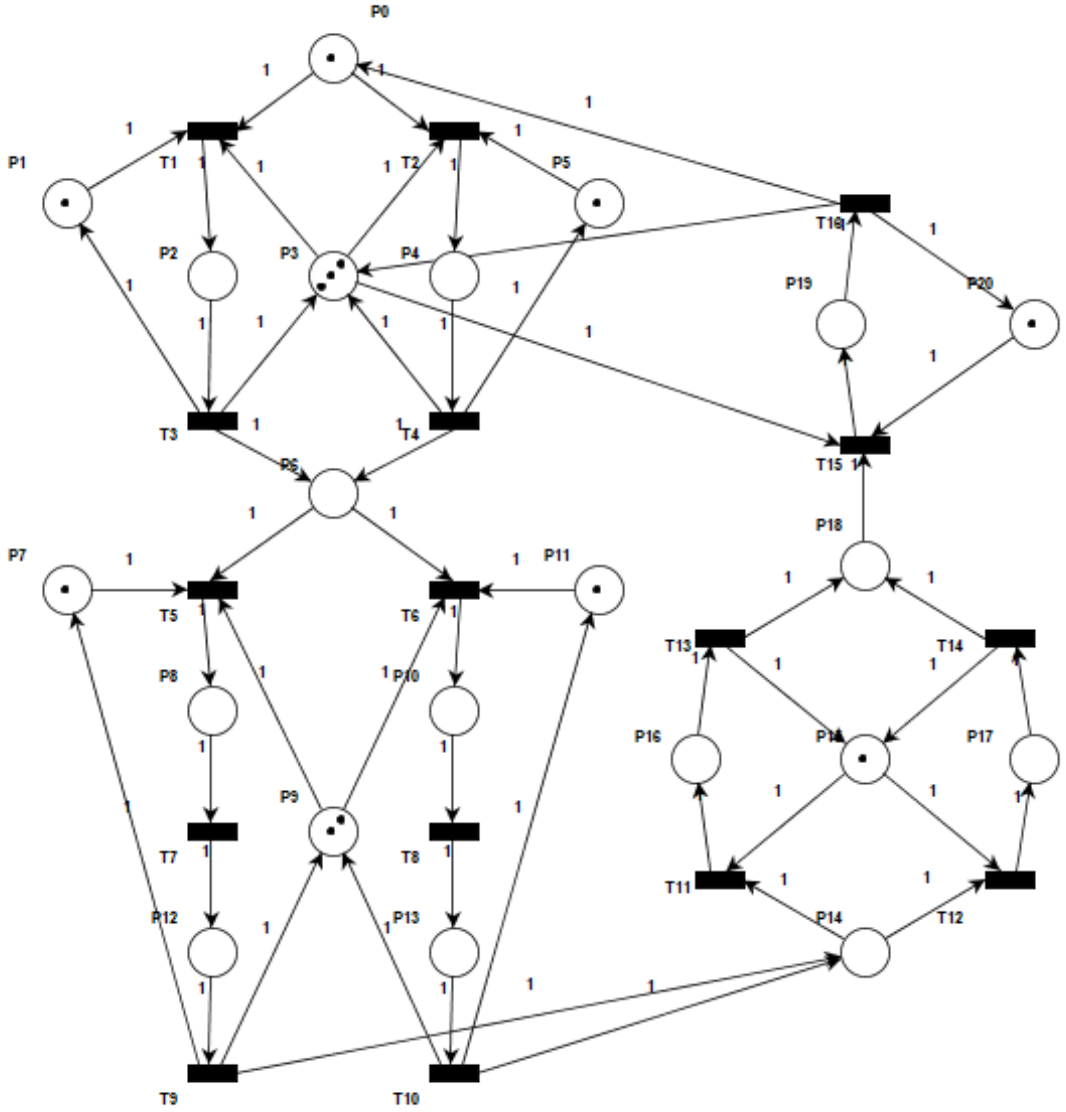


Figura 3. Red de Petri modificada para cálculo de invariantes.

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

3. Cálculo de hilos y segmentos

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.

Se calcularon los invariantes de transición y se identificaron los conjuntos de plazas asociadas a cada invariante de transición.

T-Invariante	Plazas asociadas
1	{P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P16, P18, P19, P20}
2	{P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P17, P18, P19, P20}
3	{P0, P1, P2, P3, P6, P9, P10, P11, P13, P14, P15, P16, P18, P19, P20}
4	{P0, P1, P2, P3, P6, P9, P10, P11, P13, P14, P15, P17, P18, P19, P20}
5	{P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P16, P18, P19, P20}
6	{P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P17, P18, P19, P20}
7	{P0, P3, P4, P5, P6, P9, P10, P11, P13, P14, P15, P16, P18, P19, P20}
8	{P0, P3, P4, P5, P6, P9, P10, P11, P13, P14, P15, P17, P18, P19, P20}

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

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

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:

T-Invariante	Plazas
1	{P2, P8, P12, P16, P19}
2	{P2, P8, P12, P17, P19}
3	{P2, P10, P13, P16, P19}

4	{P2, P10, P13, P17, P19}
5	{P4, P8, P12, P16, P19}
6	{P4, P8, P12, P17, P19}
7	{P4, P10, P13, P16, P19}
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 las plazas:

{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 PA, considerando todos los marcados posibles MA. Dado que los MA dependen del marcado inicial y que el número de tokens en la plaza P0 es variable ya que representa las imágenes que ingresan a la red, se generaron los árboles de alcanzabilidad para diferentes cantidades de tokens iniciales en la plaza P0.

Los árboles de alcanzabilidad fueron analizados utilizando la herramienta TINA. A continuación, se muestra la salida correspondiente, con el marcado inicial establecido en P0 igual a 1:

```
des(0,29,13)
(0,"S.`p0` S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7`
S.`p9`*2",0)
(0,"E.`t1`",1)
(0,"E.`t2`",2)
(1,"S.`p11` S.`p15` S.`p2` S.`p20` S.`p3`*2 S.`p5` S.`p7` S.`p9`*2",1)
(1,"E.`t3`",3)
(2,"S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*2 S.`p4` S.`p7` S.`p9`*2",2)
(2,"E.`t4`",3)
(3,"S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p6` S.`p7`
S.`p9`*2",3)
(3,"E.`t5`",4)
(3,"E.`t6`",5)
(4,"S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p8` S.`p9`",4)
(4,"E.`t7`",6)
(5,"S.`p1` S.`p10` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7` S.`p9`",5)
(5,"E.`t8`",7)
(6,"S.`p1` S.`p11` S.`p12` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p9`",6)
(6,"E.`t9`",8)
(7,"S.`p1` S.`p13` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7` S.`p9`",7)
(7,"E.`t10`",8)
(8,"S.`p1` S.`p11` S.`p14` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7`
S.`p9`*2",8)
(8,"E.`t11`",9)
(8,"E.`t12`",10)
(9,"S.`p1` S.`p11` S.`p16` S.`p20` S.`p3`*3 S.`p5` S.`p7` S.`p9`*2",9)
```

```
      (9,"E.`t13`",11)
      (10,"S.`p1`  S.`p11`  S.`p17`  S.`p20`  S.`p3`*3  S.`p5`  S.`p7`
S.`p9`*2",10)
      (10,"E.`t14`",11)
      (11,"S.`p1`  S.`p11`  S.`p15`  S.`p18`  S.`p20`  S.`p3`*3  S.`p5`  S.`p7`
S.`p9`*2",11)
      (11,"E.`t15`",12)
      (12,"S.`p1`  S.`p11`  S.`p15`  S.`p19`  S.`p3`*2  S.`p5`  S.`p7`
S.`p9`*2",12)
      (12,"E.`t16`",0)
```

Por ejemplo, esto se lee de la siguiente manera:

(index,marcado,transiciones)

```
(0,"S.`p0` S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7` S.`p9`*2",0)
```

(index,transición habilitada,destino)

```
(0,"E.`t1`",1)
```

Para procesar este archivo, se analiza su contenido utilizando el código que se describe a continuación:

```
import re  # Importamos el módulo 're' para trabajar con expresiones
regulares

def parse_marked_lines(file_path):
      # Función para parsear las líneas de un archivo de texto con
formato específico

    # Abrimos el archivo en modo lectura
    with open(file_path, 'r') as file:
        # Leemos todas las líneas del archivo
        lines = file.readlines()

    resultados = []  # Lista para almacenar los resultados finales

    # Iteramos sobre cada línea del archivo
    for line in lines:
         # Utilizamos una expresión regular para extraer la parte de la
línea que contiene las etiquetas "S.p0", etc.
        match = re.search(r'"(S\.[^"]*)"', line)
        if match:
              # Si encontramos una coincidencia, extraemos el texto de
las etiquetas S.p0, S.p1, etc.
            marcado_texto = match.group(1)
```

```
                        # Inicializamos una lista con 21 elementos (correspondientes a p0, p1,..., p20), todos con valor 0
            marcado = [0] * 21

             # Ahora buscamos todos los valores "p0", "p1" o "p3*3", con o sin multiplicador (p3*3)
              # La expresión regular `(?:\*(\d+))?` permite detectar el multiplicador
            tokens = re.findall(r'`p(\d+)`(?:\*(\d+))?', marcado_texto)

                  # Iteramos sobre los tokens encontrados (pares p y multiplicador)
            for p, mult in tokens:
                  index = int(p)  # Convertimos el número de p a entero (índice de la lista)
                    multiplicador = int(mult) if mult else 1  # Si hay multiplicador, lo usamos, si no, ponemos 1

                     # Verificamos que el índice esté dentro del rango permitido (0-20)
                if 0 <= index < len(marcado):
                     # Sumamos el valor del multiplicador en la posición correspondiente
                     marcado[index] += multiplicador

               # Calculamos la suma de las posiciones que nos interesan (2, 4, 8, 10, 12, 13, 16, 17, 19)
               suma_especificas = sum(marcado[i] for i in [2, 4, 8, 10, 12, 13, 16, 17, 19])

                # Almacenamos el marcado (lista de p0 a p20) y la suma calculada como un tupla
            resultados.append((marcado, suma_especificas))

    # Retornamos la lista de resultados (marcado y suma)
    return resultados

# Ruta del archivo de entrada
file_path = 'm1.kts'

# Pasamos el archivo al parser para que lo procese
parsed_results = parse_marked_lines(file_path)
```

```
# Encontramos la línea con el mayor valor de suma usando la función max y una expresión lambda
max_sum = max(parsed_results, key=lambda x: x[1])  # La clave de la comparación es la suma (x[1])

# Mostrar los resultados de todas las líneas con su marcado y suma
for i, (marcado, suma) in enumerate(parsed_results):
    print(f"Línea {i + 1}: {marcado} | Suma: {suma}")

# Al final, imprimimos la línea con la mayor suma
print("\nLínea con la mayor suma:")
print(f"Marcado: {max_sum[0]} | Suma: {max_sum[1]}")
```

A continuación, se muestra la salida correspondiente, con el marcado inicial establecido en P0 igual a 1:

```
Línea 1: [1, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 0
Línea 2: [0, 0, 1, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 3: [0, 1, 0, 2, 1, 0, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 4: [0, 1, 0, 3, 0, 1, 1, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 0
Línea 5: [0, 1, 0, 3, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 6: [0, 1, 0, 3, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 7: [0, 1, 0, 3, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 8: [0, 1, 0, 3, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 9: [0, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1] | Suma: 0
Línea 10: [0, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1] | Suma: 1
Línea 11: [0, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1] | Suma: 1
Línea 12: [0, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1] | Suma: 0
Línea 13: [0, 1, 0, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0] | Suma: 1

Línea con la mayor suma:
Marcado: [0, 0, 1, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
```

Realizando esto para distintos marcados iniciales en P0, los resultados obtenidos fueron los siguientes:

marcado(P0)	Max(SumaTokens(PA))
1	1
2	2
3	3
4	4
5	5
6	6

7	6
8	6
9	6
10	6

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. Por lo tanto, el número máximo de hilos activos simultáneos para el sistema es 6.

Cabe señalar que el análisis se realizó con la red modificada, lo que excluye la transición T0. 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**.

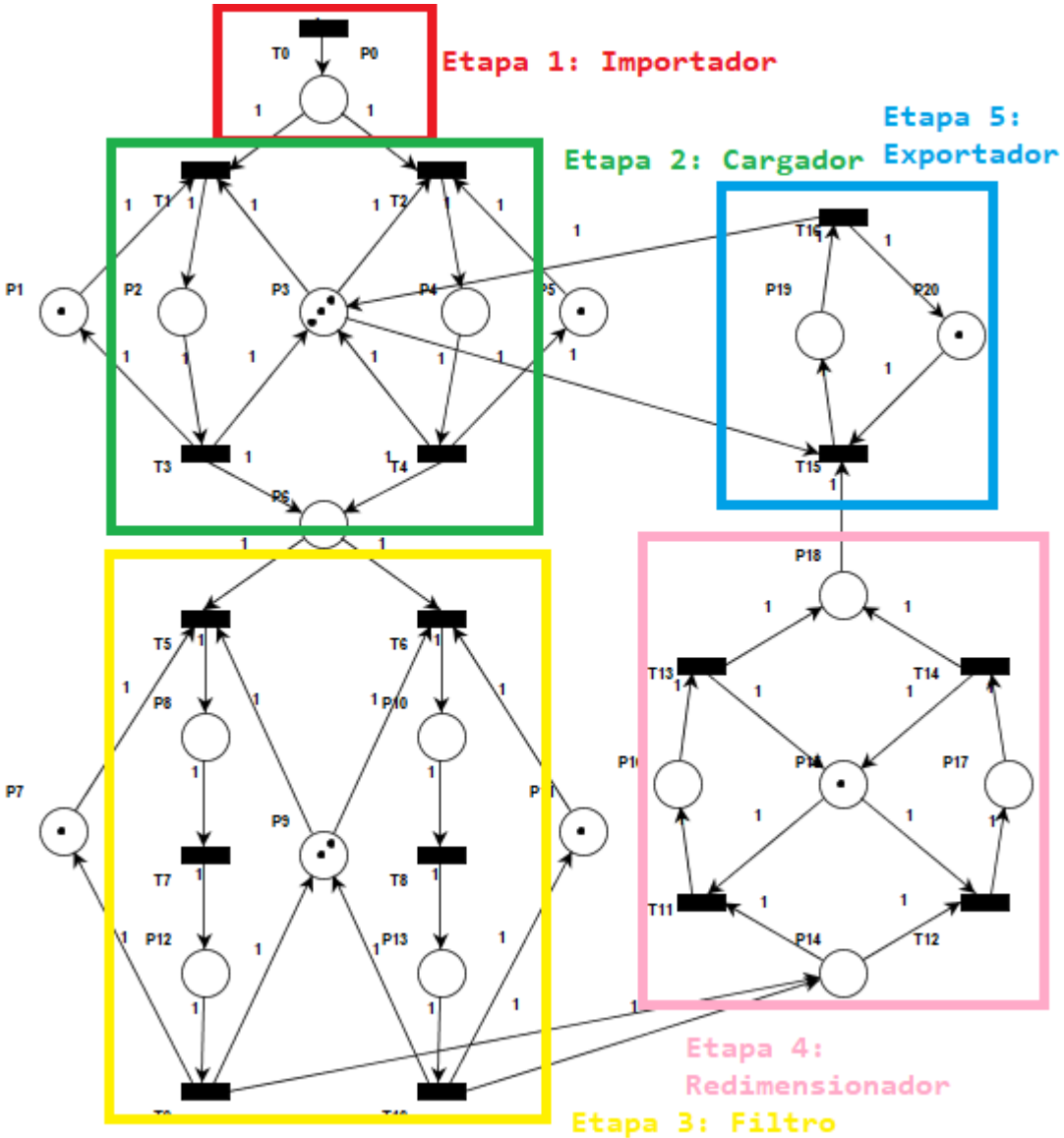
La red presenta forks (puntos de conflicto) y joins (puntos de unión). El primer fork ocurre en la plaza P0, de la cual surgen dos segmentos. Estos segmentos se unen en un join en la plaza P6, que a su vez funciona como un nuevo fork. Nuevamente, se generan dos segmentos que se unen en la plaza P14. 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 y las plazas asociadas a cada segmento son:

Etapa	Segmento	Plazas asociadas
Importar	S1	{P0}
Cargar	S2	{P2}
	S3	{P4}
Filtrar	S4	{P8, P12}
	S5	{P10, P13}
Redimensionar	S6	{P16}
	S7	{P17}
Exportar	S8	{P19}

Y las transiciones asociadas a cada segmento son:

Etapa	Segmento	Transiciones
Importar	S1	{T0}
Cargar	S2	T1, T3

	S3	{T2, T4}
Filtrar	S4	{T5, T7, T9}
	S5	{T6, T8, T10}
Redimensionar	S6	{T11, T13}
	S7	{T12, T14}
Exportar	S8	{T15, T16}



La cantidad máxima de hilos por segmento se presentan en la tabla siguiente:

Etapa	Segmento	Max(Suma(PS))

Importar	S1	1
Cargar	S2	1
	S3	1
Filtrar	S4	1
	S5	1
Redimensionar	S6	1
	S7	1
Exportar	S8	1

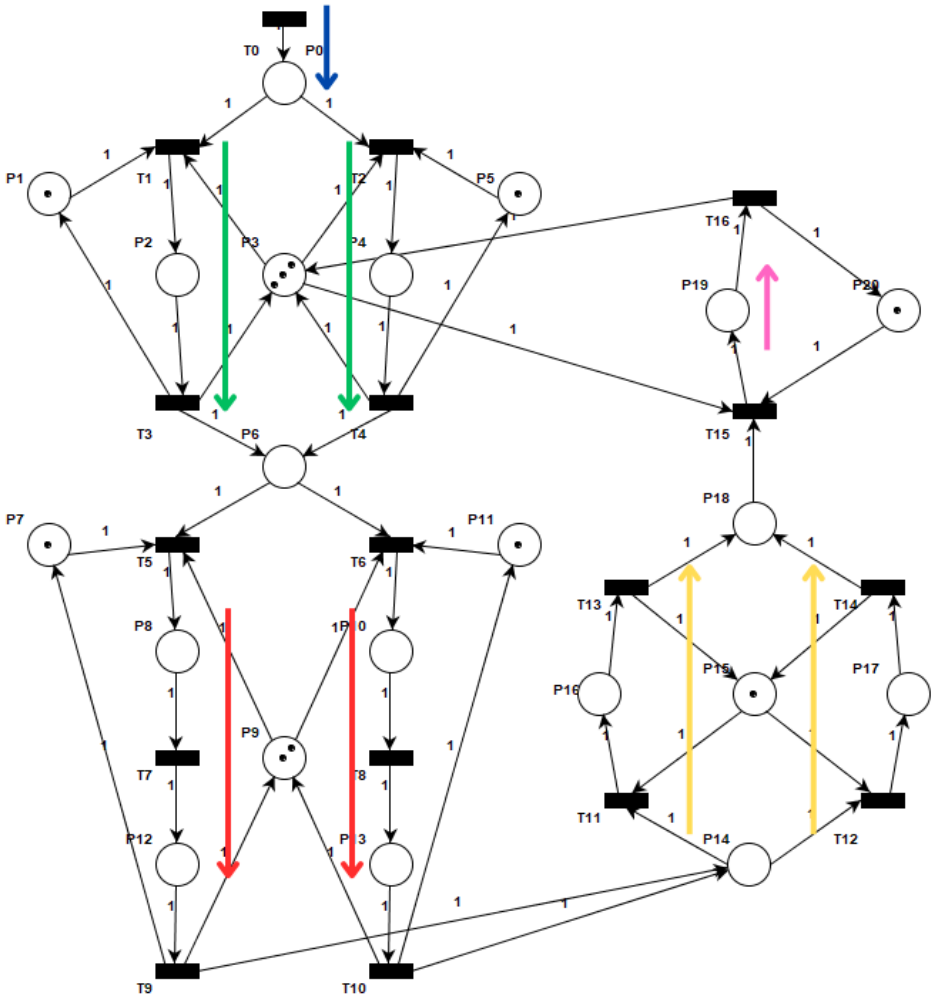


Figura 4. Red de petri segmentada en hilos de acción.

4. Implementación

4.1 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.

```
    /**
    * 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 Monitor para coordinar las transiciones.
    */
    public Proceso(String nombre, int[] transiciones, long tiempo, Monitor monitor) {
        this.nombre = nombre;
        this.transiciones = transiciones;
        index = 0;
        this.tiempo = tiempo;
        cuenta = new int[transiciones.length];
        this.monitor = monitor;
    }
```

4.2 Importador, Cargador, Filtro, Redimensionador, Exportador

Todas estas clases se heredan de la clase abstracta Proceso. Su función principal es ejecutar de manera continua 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 y es el encargado de detener el comienzo de la detención del programa al finalizar la ejecución.

```
    /**
    * Metodo que ejecuta el proceso en un hilo separado.
    * Este metodo es llamado automáticamente cuando se ejecuta `Thread.start()`.
    */
    @Override
    public void run() {
        while (!monitor.debeDetener()) {
            try {
                monitor.dispararTransicion(transiciones[index]);
                if (!monitor.debeDetener()) setCuenta(index);
                index = (index + 1) % transiciones.length;
                TimeUnit.MILLISECONDS.sleep(tiempo);
```

```
            } catch (InterruptedException e) {
                                System.err.println(getNombre()  +  ":  Proceso
interrumpido.");
                break;
            } catch (RuntimeException e) {
                    System.err.println(getNombre()  +  ": Error durante el
disparo de transición: " + e.getMessage());
                break;
            }
        }
        printStats();
    }
```

4.3 Colas

La clase Colas implementa semáforos para gestionar la espera y liberación de hilos en función de que estén habilitadas transiciones específicas. Sus métodos más importantes son el esperar() y liberar(). Ambos funcionan de manera semejante a los métodos acquire() y release() de la clase semáforos, sin embargo, estas funciones reciben como parámetro la transición específica para tomar y liberar el recurso.

```
    /**
     *  Bloquea un hilo en la cola asociada a una transición específica.
     *   La cola utiliza un semáforo con un contador inicial de 0,
obligando al hilo a esperar hasta que se llame a `release`.
     *
     * @param transicion Índice de la transición/cola en la que se debe
bloquear el hilo.
     */
    public void esperar(int transicion) {
        validarIndice(transicion);
        listaBloqueadas[transicion] = 1;
        try {
            colas.get(transicion).acquire();
        } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // Restaura el estado
interrumpido
               throw new IllegalStateException("Hilo interrumpido mientras
esperaba en la transicion " + transicion, e);
        }
    }

    /**
     * Metodo que libera un hilo bloqueado en una transición específica.
     *
     * @param transicion Índice de la transición/cola en la que se debe
liberar el hilo.
     */
    public void liberar(int transicion) {
        validarIndice(transicion);
         colas.get(transicion).release();                // Incrementa el
contador del semáforo, permitiendo que un hilo continúe.
```

```
            listaBloqueadas[transicion] = 0;                    // Marca la
transición como no bloqueada en el arreglo de bloqueos.
        }
```

4.4 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. Su método más importante es seleccionarTransición(). Este decide qué transición disparar basándose en la política elegida por el usuario.

```
        /**
        * Selecciona una transición para disparar en función del tipo de
política.
        * BALANCEADA: Prioriza la transición habilitada con menos disparos
realizados.
        * PRIORITARIA: Prioriza un segmento en la etapa 3 si cumple la
relación definida por la prioridad. Si no es posible, utiliza la política
BALANCEADA.
        *
        * @param transicionesHabilitadas Arreglo binario (1 = habilitada, 0 =
no habilitada).
        * @param disparos                Arreglo que lleva el conteo de los
disparos realizados por cada transición.
        * @return El índice de la transición seleccionada, o -1 si ninguna
transición está habilitada.
        * @throws IllegalArgumentException Si los arreglos tienen tamaños
inválidos.
        */
        public int seleccionarTransicion(int[] transicionesHabilitadas,
double[] disparos) {

            validarArreglos(transicionesHabilitadas, disparos);

            int transicionSeleccionada = -1;    // Inicializa la transición
seleccionada como no válida (-1).
            double minValue = disparos[0];      // Inicializa el valor mínimo
con el número de disparos de la primera transición.

            switch (tipo){
                case BALANCEADA:
                        // Recorre todas las transiciones para encontrar la
habilitada con menor cantidad de disparos.
                    for (int i = 1; i < cantidadTransiciones; i++) {
                        // Verifica si la transición está habilitada.
                        if (transicionesHabilitadas[i] == 1) {
                                // Si la transición tiene menos disparos que el
valor mínimo actual, la selecciona.
                            if (disparos[i] < minValue) {
```

```
                                minValue = disparos[i];       // Actualiza el
valor mínimo.
                                transicionSeleccionada = i;  // Actualiza la
transición seleccionada.
                            }
                        }
                    }
                    break;
                case PRIORITARIA:
                    /*
                         Política de procesamiento que prioriza un segmento en
la etapa 3.
                        Este segmento incluye las transiciones T11, T12, T13 y
T14.
                    */
                     int j=0;    // Usado en el bucle siguiente para omitir la
eleccion de las transiciones prioritarias.

                    if(Objects.equals(segmento, Segmento.IZQUIERDA)){
                        j=12;
                        if(transicionesHabilitadas[j]==1 && disparos[j-1]!=0){
                            double relacion = disparos[j]/disparos[j-1];
                            if (relacion<(1-prioridad)){
                                return 12;
                            }
                        }
                    } else if (Objects.equals(segmento, Segmento.DERECHA)){
                        j=11;
                        if(transicionesHabilitadas[j]==1 && disparos[j+1]!=0){
                            double relacion = disparos[j]/disparos[j+1];
                            if (relacion<(1-prioridad)){
                                return 11;
                            }
                        }
                    }
                     // Recorre todas las transiciones para encontrar la
habilitada con menor cantidad de disparos.
                    // Esta etapa es igual a la Politica BALANCEADA.
                    for (int i = 1; i < cantidadTransiciones; i++) {
                        // Verifica si la transición está habilitada.
                        if (transicionesHabilitadas[i] == 1 && i!=j) {
                              // Si la transición tiene menos disparos que el
valor mínimo actual, la selecciona.
                            if (disparos[i] < minValue) {
                                minValue = disparos[i];       // Actualiza el
valor mínimo.
                                transicionSeleccionada = i;  // Actualiza la
transición seleccionada.
                            }
                        }
                    }
                    break;
                default:
```

```
                System.out.println("Política inválida.");
            }
            return transicionSeleccionada;
        }
```

4.5 Monitor

La clase Monitor implementa una estructura para sincronizar 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. Su método más importante es dispararTransición(). Este recibe la transición a disparar y lo hace dependiendo del estado de la red de Petri.

```
/**
* Dispara una transición en la red de Petri de forma concurrente.
* Gestiona las colas, sincronización mediante semáforos y verificaciones de
condiciones para el disparo.
*
* @param transicion Índice de la transición a disparar.
*/
public void dispararTransicion(int transicion) {

    // Intenta adquirir el semáforo para entrar al monitor.
    try {
        mutex.acquire();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }

    boolean k = true;

    while (k) {
        /*
            Inicio de logica para detener el programa
         */
        if (debeDetener()){
            int index = -1;
            for (int i=0;i<colas.getColas().size();i++){
                if(colas.getColas().get(i).hasQueuedThreads()){
                    index=i;
                    break;
                }
            }
            if (index!=-1){
                // Hay hilos esperando en las colas de condicion
                colas.liberar(index);
            } else {
                mutex.release();
            }
            return;
        }
```

```
        /*
            Fin de logica para detener el programa
        */

        k = petriNet.disparar(transicion);
        if (k) {

            // Incrementa el contador de disparos para la transición actual.
            contarDisparo(transicion);

            // Registra el disparos de la transición actual.
            try {
                log.addTransicion(transicion);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }

            System.out.printf("Se disparo la transicion %d\n",transicion);

            // Obtiene las transiciones habilitadas después del disparo.
            int[] sensibilizadas = petriNet.getTransicionesHabilitadas();

             // Obtiene la lista de transiciones bloqueadas (colas con hilos
esperando).
            int[] listaBloqueadas = colas.getListaBloqueadas();

            // ¿Quienes estan?
                        int[]  listaDeEspera  =  quienesEstan(sensibilizadas,
listaBloqueadas);

             if (!Arrays.stream(listaDeEspera).allMatch(valor -> valor == 0))
{
                  // Selecciona alguna de las transiciones habilitadas con
hilos esperando y la despierta.
                                                    int   transicionADisparar   =
politicas.seleccionarTransicion(listaDeEspera, cantDisparos);
                if (transicionADisparar == -1) {
                    /*
                         Si no hay ninguna transición seleccionada, libera el
mutex y finaliza.
                           En el caso PRIORITARIA, lo libera si la relacion
(prioridad,1-prioridad) no se cumple y el unico en la cola es la transicion
involucrada en la relacion
                     */
                    mutex.release();
                } else {
                     colas.liberar(transicionADisparar); // Libera un hilo de
la cola correspondiente.
                }
                return;
            } else {
                  // Si no hay hilos esperando en transiciones habilitadas,
termina el bucle.
```

```
                k = false;
            }
        } else {
            /*
                    Si la transición no fue disparada, puede deberse a las
siguientes razones:
                    1. La transición no cuenta con los tokens necesarios.
                      2. El tiempo alfa aún no se ha cumplido, por lo que el
hilo debe esperar un tiempo igual a alfa - ahora antes de volver a intentar
disparar.
                    3. El tiempo beta ha sido superado, lo que impide volver
a disparar la transición.
             */
            if(!petriNet.estaHabilitada(transicion)){
                 // 1. La transición no cuenta con los tokens necesarios. El
hilo se bloquea en la cola correspondiente.
                mutex.release();
                 colas.esperar(transicion); // El hilo entra en la cola de la
transición.
                k = true;
            } else if (petriNet.getVentanaTiempos()[transicion]==1) {
                 // 2. El tiempo alfa aún no se ha cumplido, por lo que el
hilo debe esperar un tiempo igual a alfa - ahora antes de volver a intentar
disparar.
                long ahora = System.currentTimeMillis();
                                                        long    tiempo    =
(petriNet.getTimeStamp()[transicion]+petriNet.getTiempos()[transicion][0])-ah
ora;
                mutex.release();
                try {
                    TimeUnit.MILLISECONDS.sleep(tiempo);
                    mutex.acquire();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                k=true;
            } else if (petriNet.getVentanaTiempos()[transicion]==2) {
                 // 3. El tiempo beta ha sido superado, lo que impide volver
a disparar la transición.
                      System.out.printf("Se intento disparar la transicion
%d\n",transicion);
                long ahora = System.currentTimeMillis();
                System.out.println(ahora-petriNet.getTimeStamp()[transicion]);
                throw new RuntimeException("Fuera de la ventana de tiempos");
            }
        }
    }
    mutex.release(); // Libera el semáforo al salir del monitor.
}
```

4.6 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. Su método más importante es disparar(). Este recibe una transición y evalúa si está habilitada por token y, en caso de estar temporizada, si el disparo llegó dentro de la ventana de tiempos para disparar.

```
	/**
	 * Dispara una transición en la red de Petri.
	 *  Verifica las condiciones necesarias (habilitación por tokens y ventanas de tiempo) antes de proceder con el disparo.
	 *
	 * @param transicion Índice de la transición a disparar.
	 * @return `true` si el disparo se realizó correctamente, `false` si no fue posible.
	 */
	public boolean disparar(int transicion) {

	    // Verifica si la transición es válida.
	    if (transicion < 0 || transicion >= cantidadTransiciones) {
	            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)) {

	            int[] nuevoMarcado = operar(transicion);

	             if (Arrays.stream(nuevoMarcado).anyMatch(valor -> valor < 0)) {
	                 System.out.println("Los valores del marcado no pueden ser negativos.");
	                return false;
	            }

	             // Verifica que no se hayan corrompido los invariantes de plaza.
	            if (!verificarInvariantesPlaza(nuevoMarcado)) {
	                // Actualiza el marcado y las transiciones habilitadas según los tokens disponibles.
	                actualizarHabilitadas(nuevoMarcado);
	                return true;
	            } else {
```

```
                            throw new RuntimeException("Se corrompieron los
invariantes de plaza.");
                    }
                }
            }
            return false;
        }
```

4.7 Main

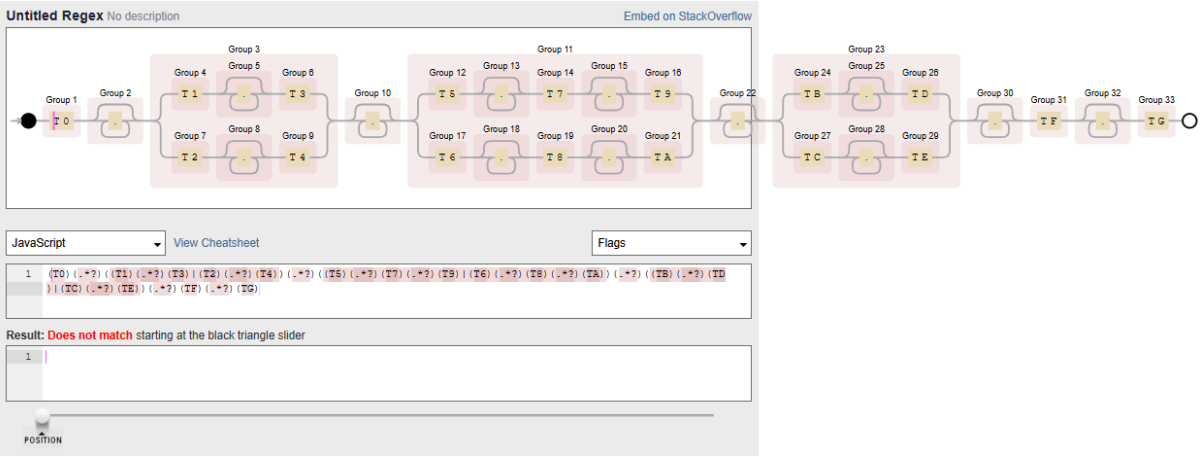
La clase Main coordina la ejecución de procesos concurrentes que representan distintas etapas de un sistema de producción (como importar, cargar, filtrar, redimensionar y exportar). Su objetivo es asegurar que se cumpla un número máximo de disparos de transiciones en una red de Petri, deteniendo los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

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

    logMessage("Se inició la ejecución de la red: " + formatter.format(new
Date()));

   try {
       // Ruta del archivo de log
       String LOG_PATH = "log/log.txt";
       Log log = new Log(LOG_PATH);
       Monitor monitor = new Monitor(log,politica,segmento,prioridad,red);
       Proceso[] procesos = inicializarProcesos(monitor);

       Thread[] threads = startProcesos(procesos);
       esperarProcesos(monitor);

       pararProcesos(monitor,threads);
       printStats(monitor);

       log.closeFile();
   } catch (IOException e) {
       System.err.println("Error al inicializar el log: " + e.getMessage());
   }

    logMessage("Se finalizó la ejecución de la red: " + formatter.format(new
Date()));
}
```

5. 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)|(T
6)(.*?)(T8)(.*?)(TA))(.*?)((TB)(.*?)(TD)|(TC)(.*?)(TE))(.*?)(TF)(.*?)(T
G)'

# Usamos una cadena cruda para evitar problemas con las secuencias de
escape
grupos                                                               =
r'\g<2>\g<5>\g<8>\g<10>\g<13>\g<15>\g<18>\g<20>\g<22>\g<25>\g<28>\g<30>
\g<32>'

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")
```

A continuación, se muestra el resultado para el análisis de un log donde la red ejecuto 5 invariantes de transición:

Salida del log:

T0T1T0T2T3T5T7T4T6T0T1T9T11T8T3T0T2T5T10T13T12T15T4T0T1T16T7T6T14
T15T3T9T11T8T16T5T10T13T12T15T16T7T14T15T9T11T16T13T15T16

El resultado obtenido es el siguiente:

```
Resultado:
Número de reemplazos realizados: 5
Éxito
```

Si incluimos una transición al azar dentro del log:

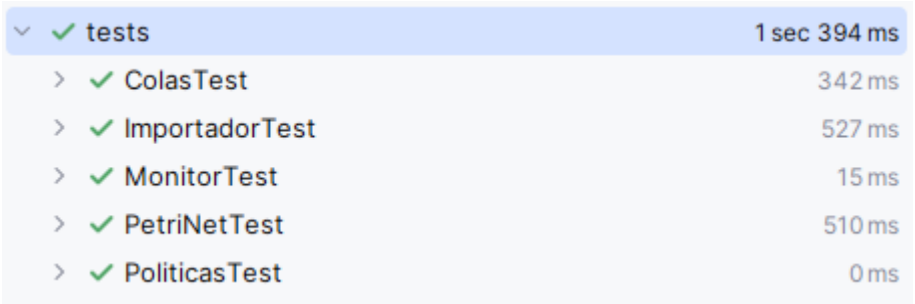
T0T16T1T0T2T3T5T7T4T6T0T1T9T11T8T3T0T2T5T10T13T12T15T4T0T1T16T7T6
T14T15T3T9T11T8T16T5T10T13T12T15T16T7T14T15T9T11T16T13T15T16

El resultado obtenido es el siguiente:

```
Resultado: TG
Número de reemplazos realizados: 5
Falló
```

6. 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: 17 total, 17 passed

1.39 s

Collapse | Expand

ColasTest		342 ms
ColasTest.testEsperar	passed	125 ms
ColasTest.testEsperaryLiberar	passed	110 ms
ColasTest.testLiberar	passed	107 ms
ColasTest.testConstructor	passed	0 ms
ImportadorTest		527 ms
ImportadorTest.testRunExecution	passed	527 ms
ImportadorTest.testGetNombre	passed	0 ms
ImportadorTest.testSetAndGetCuenta	passed	0 ms
MonitorTest		15 ms
MonitorTest.testQuienesEstan	passed	0 ms
MonitorTest.testMutex	passed	0 ms
MonitorTest.testDispararTransicionExito	passed	15 ms
PetriNetTest		510 ms
PetriNetTest.testDispararSinTiempos	passed	0 ms
PetriNetTest.testDispararConTiempos	passed	510 ms
PetriNetTest.testDispararTransicionNoHabilitada	passed	0 ms
PetriNetTest.testActualizarHabilitadas	passed	0 ms
PoliticasTest		0 ms
PoliticasTest.seleccionarTransicionPrioritariaTest	passed	0 ms
PoliticasTest.seleccionarTransicionSinHabilitadasTest	passed	0 ms
PoliticasTest.seleccionarTransicionBalanceadaTest	passed	0 ms

Generated by IntelliJ IDEA on 1/28/25, 11:00 AM

7. 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.

8. 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.

8.1 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
```

8.2 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
```

9. 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.

10. Referencias

/1/: Artículo cantidad de hilos.
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

11. Repositorio de GitHub

nachoborgatello/Trabajo-Final-Programacion-Concurrente