Trabajo práctico final

Programación Concurrente

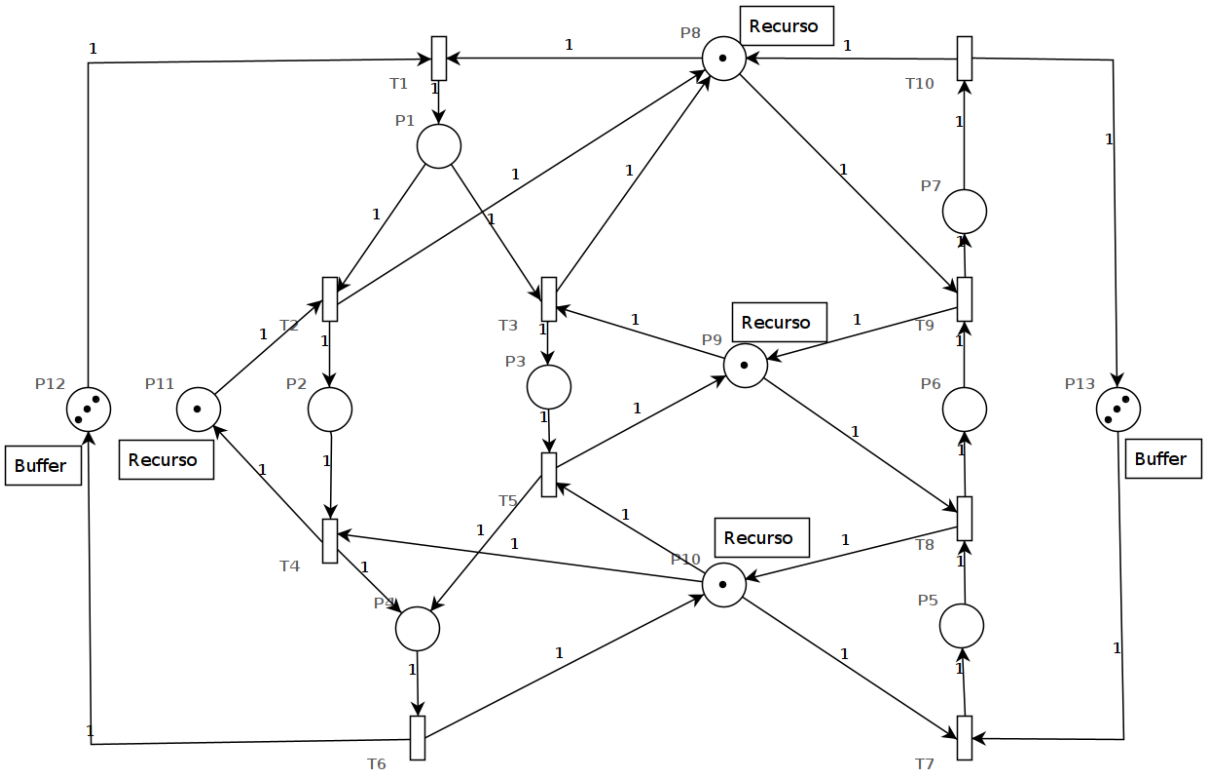
Ingeniería en Computación

Alumnos:
Castellano, Evangelina
Gazzoni, Fabio

Profesores:
Ing. Ventre, Luis
Ing. Ludemann, Mauricio

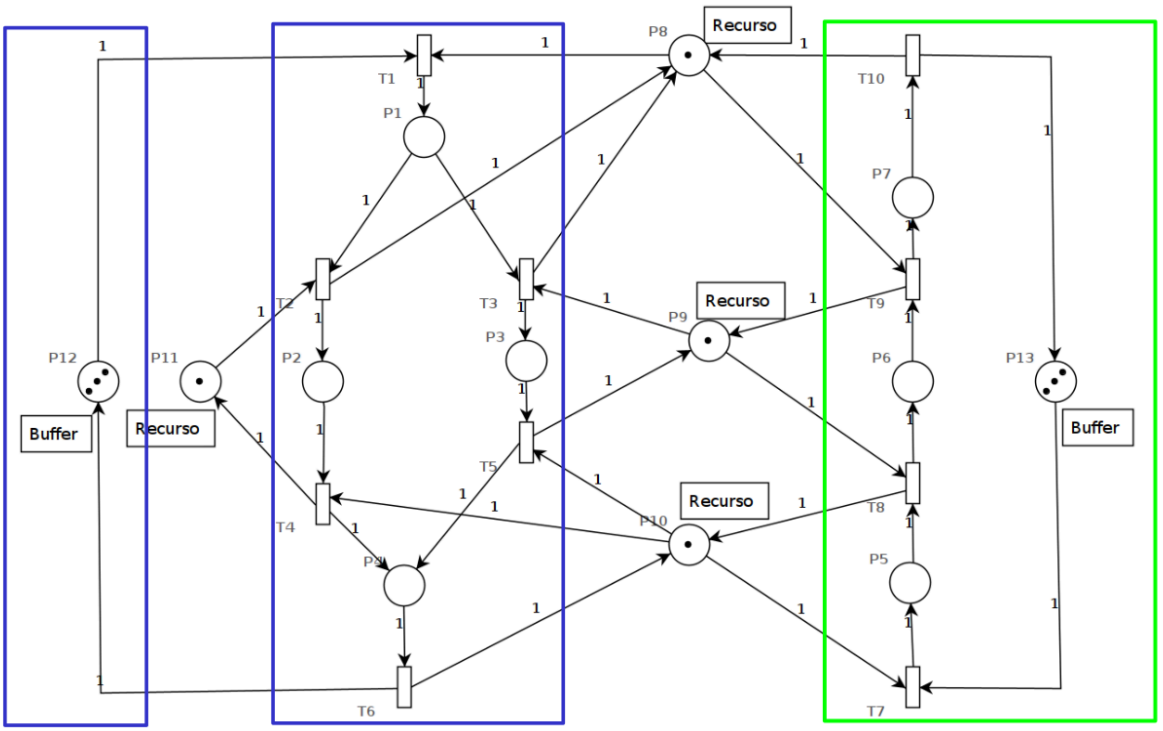
Planteo del problema

En la figura se observa una red de Petri con el modelado de un sistema de producción (industrial, informático, cyber physical systems, etc). Las plazas {P8, P9, P10, P11} representan recursos compartidos en el sistema. Las plazas {P12, P13} son plazas idle, que corresponden a buffers del sistema. El resto de las plazas son plazas donde se realizan actividades relacionadas con el proceso.



Análisis del problema

El sistema modelado se puede visualizar como dos procesos cíclicos diferentes, de los cuales uno presenta una bifurcación y luego un join, como se ilustra en la siguiente imágen:



- P12 y P13 son buffers con 3 tokens.
- P1, P2, P3, P4, P5, P6 y P7 son procesos del sistema donde se realizan actividades determinadas.
- P8, P9, P10 y P11 son recursos que limitan la cantidad de tokens de las plazas mencionadas anteriormente. Cada plaza presenta un único recurso.
- El número máximo de tokens por plaza, sin considerar los buffers (P12 y P13), es de uno.

Propiedades de la red del sistema

Petri net classification results

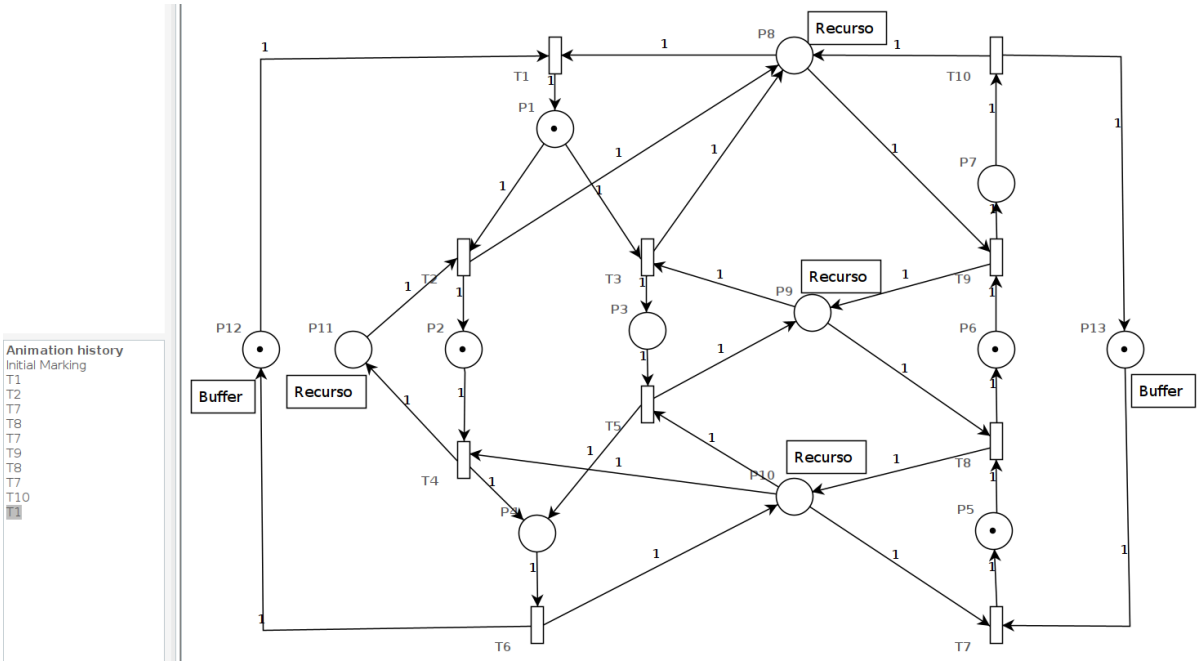
State Machine	false
Marked Graph	false
Free Choice Net	false
Extended Free Choice Net	false
Simple Net	false
Extended Simple Net	false

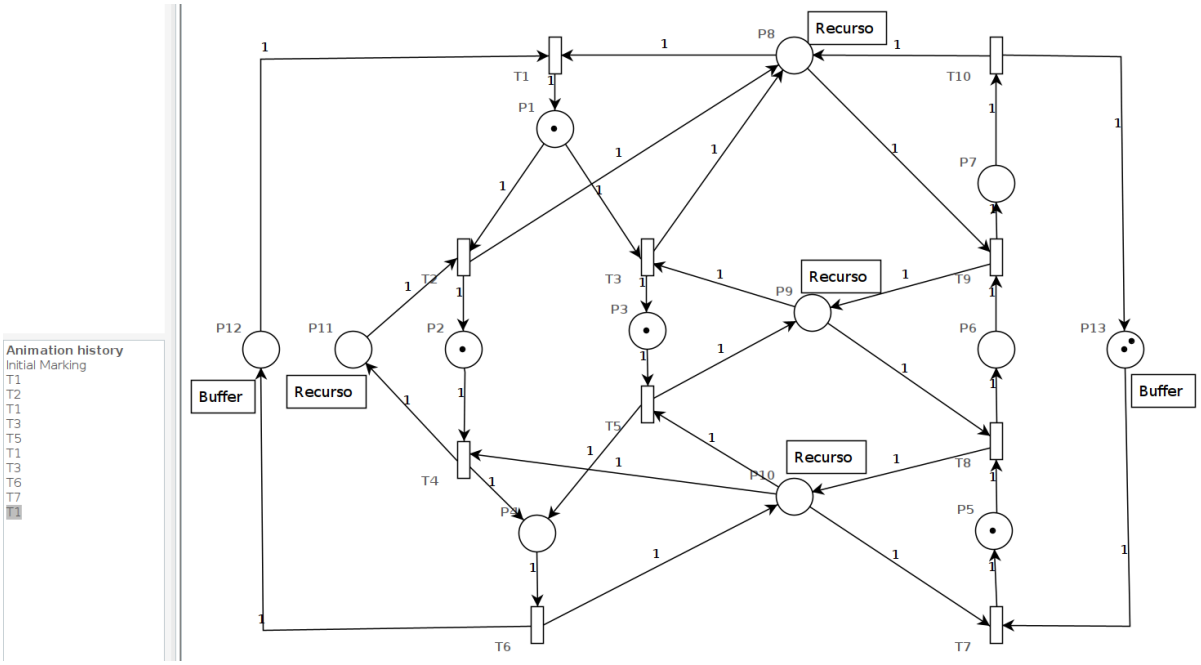
Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	true

- La red no es segura, ya que posee plazas con múltiples marcados (P12 y P13).
- La red es limitada, ya que la suma de tokens de las plazas del sistema siempre es la misma.
- La red presenta deadlock, por lo que la red no es viva.

En las siguientes imágenes se pueden ver casos donde el sistema presenta deadlock:





Invariantes

Petri net invariant analysis results

T-Invariants

T1	T10	T2	T3	T4	T5	T6	T7	T8	T9
1	0	1	0	1	0	1	0	0	0
1	0	0	1	0	1	1	0	0	0
0	1	0	0	0	0	0	1	1	1

The net is covered by positive T-Invariants, therefore it might be bounded and live.

La red presenta 3 invariantes de transición, que son la secuencia mínima de disparos para alcanzar el marcado inicial.

P-Invariants

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

The net is covered by positive P-Invariants, therefore it is bounded.

P-Invariant equations

$$M(P10) + M(P4) + M(P5) = 1$$
$$M(P11) + M(P2) = 1$$
$$M(P1) + M(P12) + M(P2) + M(P3) + M(P4) = 3$$
$$M(P13) + M(P5) + M(P6) + M(P7) = 3$$
$$M(P1) + M(P7) + M(P8) = 1$$
$$M(P3) + M(P6) + M(P9) = 1$$

Los invariantes de plazas son el conjunto de plazas que para cualquier secuencia de disparos dada la suma de sus tokens se mantienen.

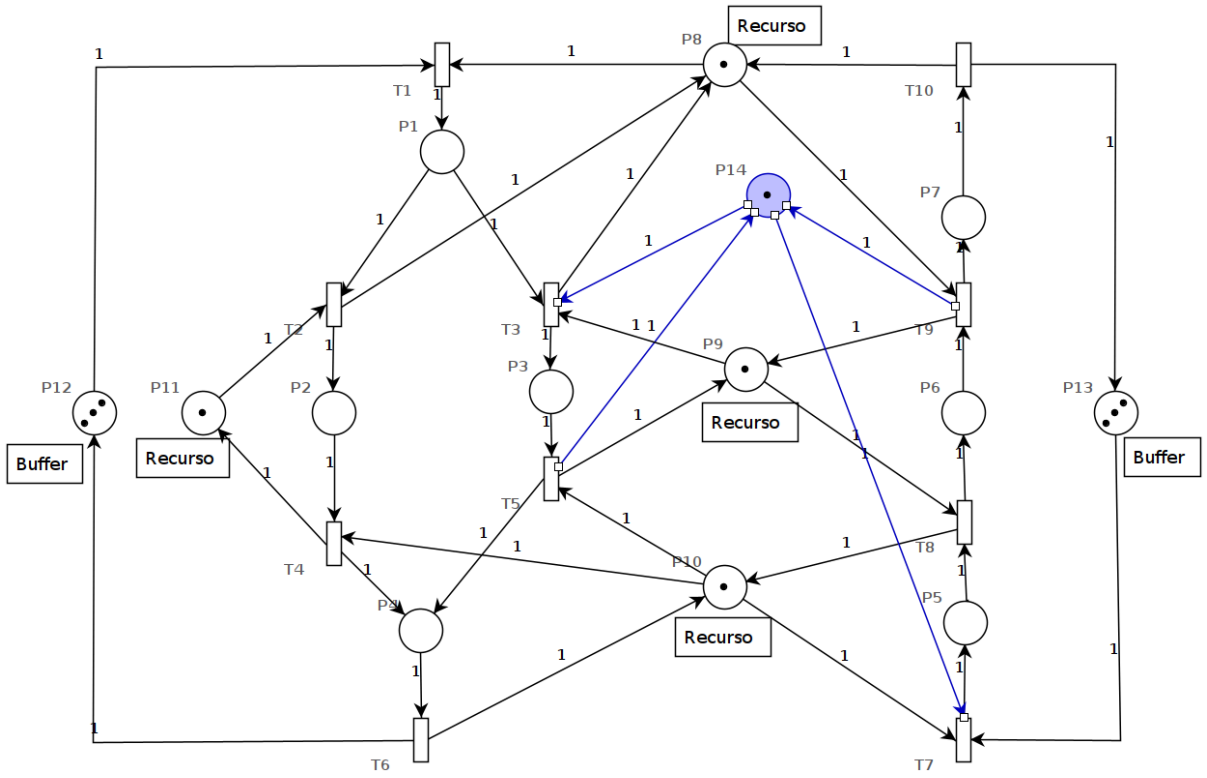
Análisis de concurrencia de la red

A partir de todos los marcados posibles de la red se puede analizar la concurrencia en las plazas de actividades (P1, P2, P3, P4, P5, P6, P7), donde el valor que representa la concurrencia será utilizado para comparar con la concurrencia de la red sin deadlock.

	P1	P2	P3	P4	P5	P6	P7	TOTAL
Prom	0,37	0,48	0,24	0,30	0,35	0,37	0,24	2,35
Min	0	0	0	0	0	0	0	
Max	1	1	1	1	1	1	1	

Eliminación del deadlock de la red

Para liberar el deadlock de la red se agregó una plaza P14 con un token. Esta plaza cumple el rol de controlar la adquisición y liberación de los recursos que poseen las plazas P8, P9 y P10 que son utilizados por los dos grandes procesos que posee el sistema.



Analizando las propiedades de la red modificada se puede ver que ahora está libre de deadlock y que los invariantes de transición se mantienen:

Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	false

T-Invariants

T1	T10	T2	T3	T4	T5	T6	T7	T8	T9
1	0	1	0	1	0	1	0	0	0
1	0	0	1	0	1	1	0	0	0
0	1	0	0	0	0	0	1	1	1

Los invariantes de plaza si cambiaron con respecto a la red original ya que se agregó una nueva plaza.

P-Invariants

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

The net is covered by positive P-Invariants, therefore it is bounded.

P-Invariant equations

$$M(P10) + M(P4) + M(P5) = 1$$
$$M(P11) + M(P2) = 1$$
$$M(P1) + M(P12) + M(P2) + M(P3) + M(P4) = 3$$
$$M(P13) + M(P5) + M(P6) + M(P7) = 3$$
$$M(P14) + M(P3) + M(P5) + M(P6) = 1$$
$$M(P1) + M(P7) + M(P8) = 1$$
$$M(P3) + M(P6) + M(P9) = 1$$

Análisis de concurrencia de la red modificada

A partir de todos los marcados posibles de la red se puede analizar la concurrencia en las plazas de actividades (P1, P2, P3, P4, P5, P6, P7), donde se puede ver que el valor obtenido es levemente menor al de la red original, esto quiere decir que la concurrencia disminuye pero la gran ventaja es que ya no hay deadlock.

	P1	P2	P3	P4	P5	P6	P7	TOTAL
Prom	0,36	0,47	0,19	0,39	0,17	0,31	0,25	2,14
Min	0	0	0	0	0	0	0	
Max	1	1	1	1	1	1	1	

Determinación de la cantidad de hilos y su responsabilidad

A partir del paper “Algoritmos para determinar cantidad y responsabilidad de hilos en sistemas embebidos modelados con Redes de Petri S^3PR” se determinará la cantidad de hilos necesarios para la red propuesta.

1) La red presenta 3 invariantes de transición:
   $IT_1 = \{T1, T2, T4, T6\}$
   $IT_2 = \{T1, T3, T5, T6\}$
   $IT_3 = \{T7, T8, T9, T10\}$

2) Las plazas asociadas al invariante son:
   $PA_1 = \{P1, P2, P4, P8, P10, P11, P12\}$
   $PA_2 = \{P1, P3, P4, P8, P9, P10, P12, P14\}$
   $PA_3 = \{P5, P6, P7, P8, P9, P10, P13, P14\}$

3) Filtrando las plazas relacionadas con acciones:
   $PA_1 = \{P1, P2, P4\}$
   $PA_2 = \{P1, P3, P4\}$
   $PA_3 = \{P5, P6, P7\}$

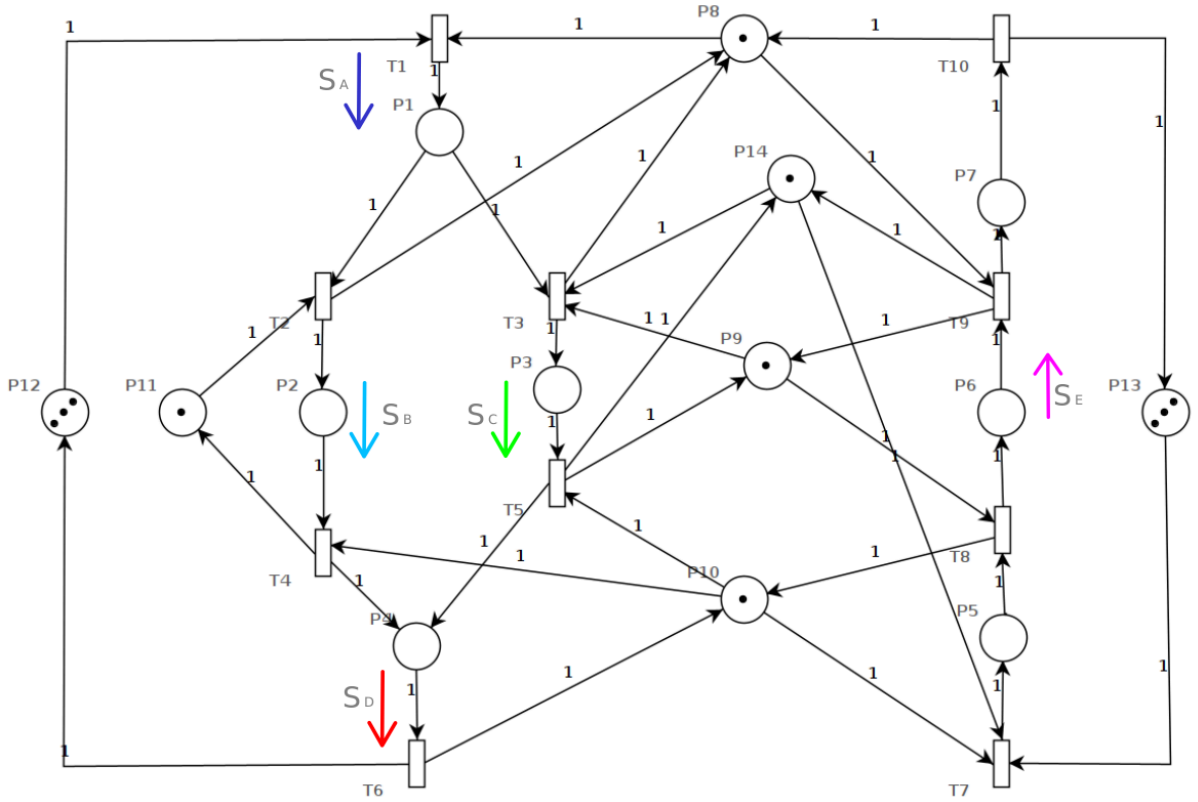
4) En la siguiente tabla se tabulan todos los marcados posibles del conjunto de plazas PA (con $PA = \{P1, P2, P3, P4, P5, P6, P7\}$)
   Marcados posibles de la RdP
   El máximo marcado es 4, esto determina la máxima cantidad de hilos activos simultáneos.

Ahora se determinarán los segmentos de la red y las responsabilidades de los hilos. Los segmentos son S_A, S_B, S_C, S_D y S_E

Los invariantes de transiciones IT_1 e IT_2 son no lineales ya que comparten transiciones que provocan conflictos estructurales, por lo que los segmentos y las plazas de acción asociadas son:

- S_A, segmento previo al conflicto (fork) y $P_{SA} = \{P1\}$.
- S_B, segmento izquierdo y $P_{SB} = \{P2\}$.
- S_C, segmento derecho y $P_{SC} = \{P3\}$.
- S_D, segmento final (join) y $P_{SD} = \{P4\}$.

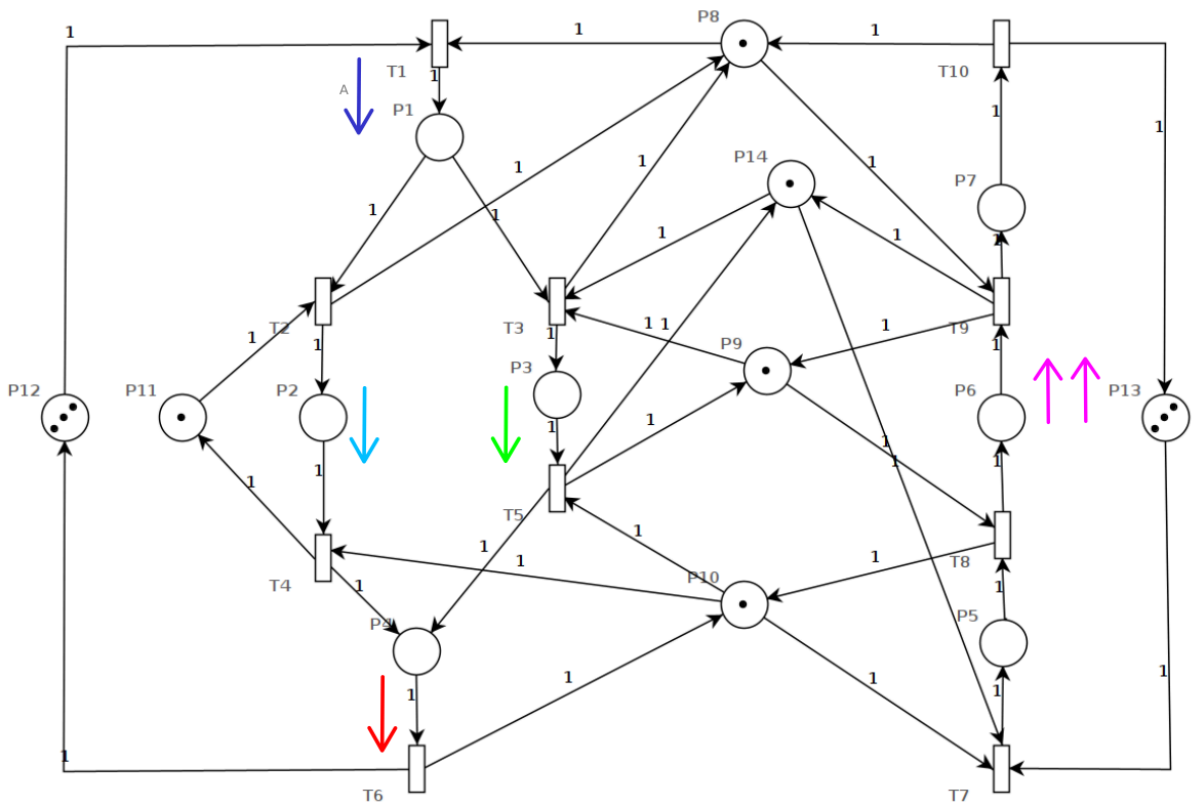
El invariante de transición IT_3 es lineal ya que no comparte transiciones con ninguno de los otros dos invariantes de la red, su segmento es S_E y las plazas de acción asociadas al segmento son $P_{SE} = \{P5, P6, P7\}$.



Luego se deben determinar los hilos máximos necesarios por segmento de ejecución, por lo que se debe elegir el máximo marcado de cada conjunto de marcados correspondientes a cada segmento:

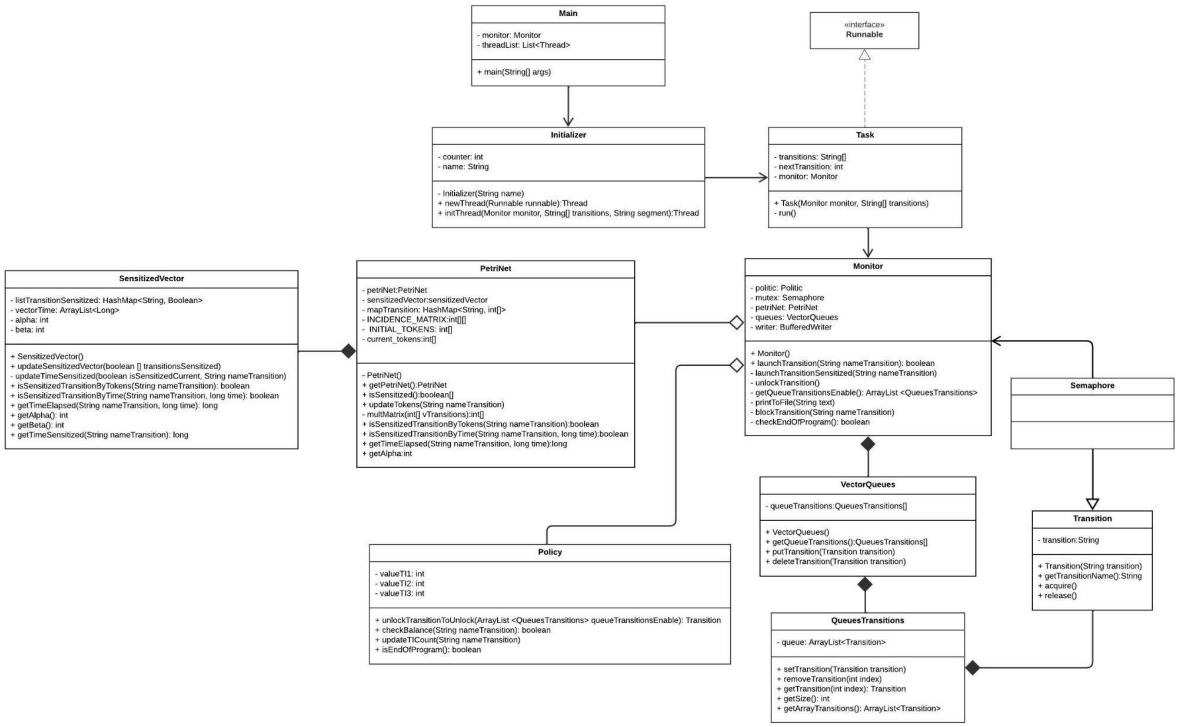
- $MAX(M_{SA}) = 1$
- $MAX(M_{SB}) = 1$
- $MAX(M_{SC}) = 1$
- $MAX(M_{SD}) = 1$
- $MAX(M_{SE}) = 2$

Por lo que el sistema está formado por 6 hilos, aunque, por lo calculado anteriormente, la cantidad de hilos que trabajarán concurrentemente son 4, los 2 restantes estarán esperando la liberación de recursos para poder avanzar.


P8
T1
T10
A
P1
P14
P7
T2
T3
T9
P9
P12
P11
P2
P3
P6
P13
T5
T4
T8
P10
P5
P4
T6
T7

Implementación en Java

A continuación se muestra el diagrama de clases del programa implementado:



Clase Main

Es es launcher del programa ya que le solicita a *Initializer* la creación de los threads y luego los inicia con el método *start().*

Clase Initializer

Se encarga de crear los threads para cada segmento de la red de Petri.
Se implementó utilizando el patrón de diseño Factory.

Clase Task

Implementa la interfaz Runnable y modela la tarea que ejecuta cada thread.
En su método *run()* intenta ingresar al monitor para disparar una transición, el objetivo de esta clase es la de realizar el disparo de una secuencia de transiciones hasta que el programa finalice.

Clase Monitor

Es el monitor de concurrencia, tiene un semáforo para controlar el acceso de los threads.
Cuando un thread logra ingresar al monitor se verifica si se puede disparar la transición deseada, es decir, si la red está sensibilizada por tokens, si se cumple con

los criterios de la política y si se cumplen las condiciones de temporalidad. Si todas las condiciones necesarias se cumplen se realiza el disparo de la transición y si hay threads bloqueados en la cola se desbloquea uno a elección de la política. Si no se cumplen las condiciones la transición no se dispara y se bloquea en la cola.

Clase PetriNet

Modela la red de Petri propuesta y lleva el estado de la red con el vector del marcado actual, además gestiona el vector de transiciones sensibilizadas.

Clase Policy

Es la clase que decide qué transición puede desbloquear el monitor. El criterio es que los T-invariantes de la red de Petri se mantengan equilibrados, por lo que se lleva la cuenta de cuantas veces se realizó cada T-invariante.

Clase SensitizedVector

Modela un vector de transiciones sensibilizadas que se actualiza cada vez que se dispara una transición. También lleva las marcas de tiempo de sensibilización de cada una.

Clase VectorQueues

Representa un vector de colas de transiciones bloqueadas.

Clase QueuesTransitions

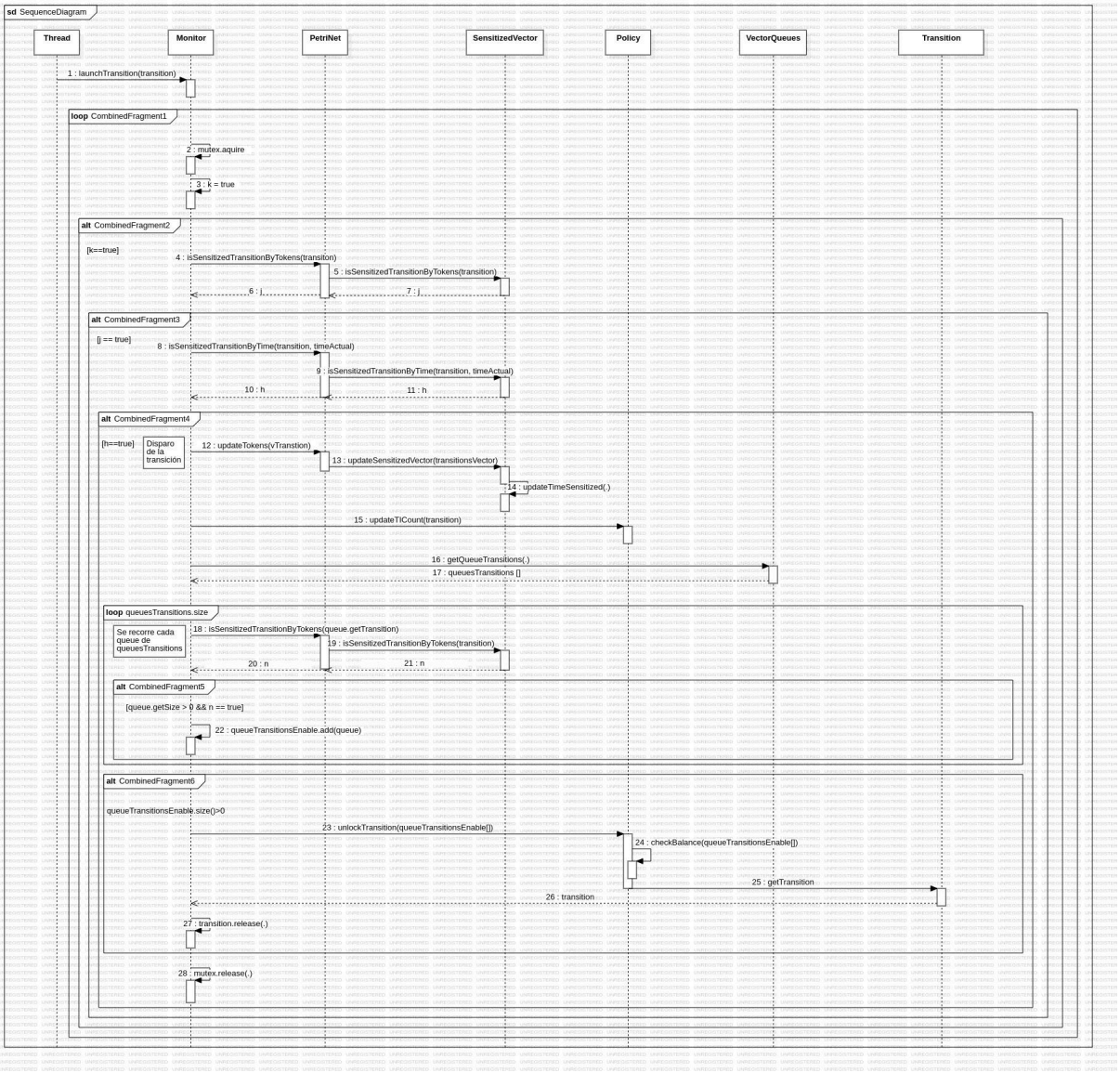
Representa la cola donde se bloqueará cada transición, existe una cola por cada transición.

Clase Transition

Modela una transición. Extiende de la clase *Semaphore,* lo que le permite poder bloquearse.

Diagrama de secuencia

En el siguiente diagrama de secuencia se muestra el disparo exitoso de una transición de la red de Petri:



Resultados sin semántica temporal

En la siguiente imágen se pueden observar que se dispararon aproximadamente 1000 veces los invariantes de transición:

```
TI1: 403 - TI2: 297 - TI3: 300
```

En algunas ejecuciones hay preferencia por el TI1 ya que puede darse el caso en el que solo T2 está sensibilizada en toda la red.

En promedio, el tiempo en que se demoró en ejecutarse el programa sin semántica temporal, fue el siguiente:

```
Tiempo :208ms
```

Resultados con semántica temporal

Lógicamente la ejecución es más lenta por los valores de alfa.

```
TI1: 452 - TI2: 272 - TI3: 276
```

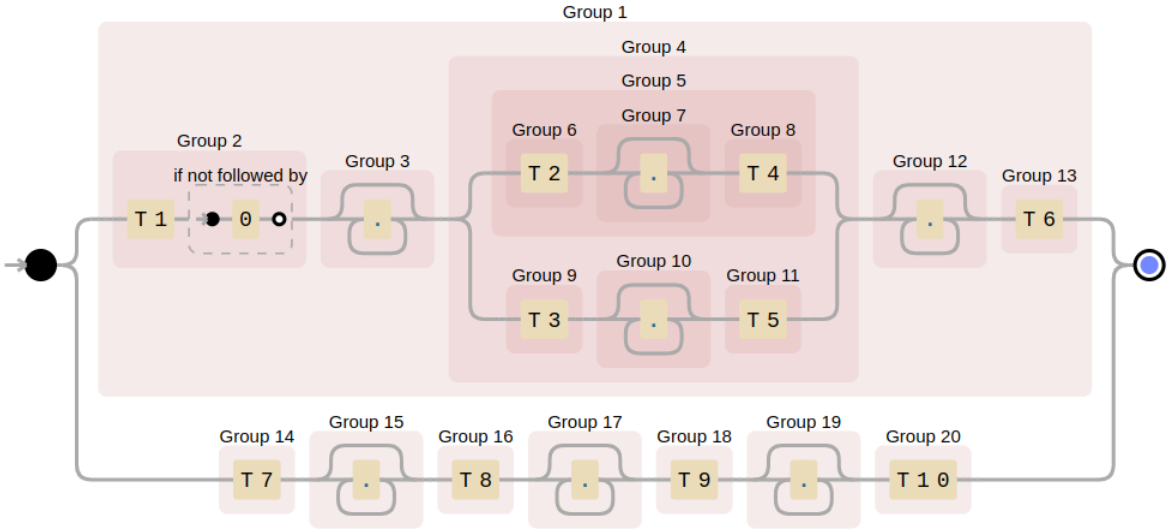
El tiempo en que se demoró en ejecutarse el programa, esta vez con un alfa de 1 ms es:

```
Tiempo :2559ms
```

Se puede evidenciar que el tiempo de ejecución varía, ya que las transiciones temporales deben verificar que haya transcurrido el tiempo alfa desde que se sensibilizan, de lo contrario deben dormir hasta que transcurra el tiempo hasta alcanzar la marca alfa. Para el caso de la marca de tiempo beta se eligió uno tan grande que no bloquee la transición por haberse pasado del tiempo asignado.

Verificación del cumplimiento de los T-invariantes

Para verificar el cumplimiento de los invariantes de transiciones se escribió en un archivo log las transiciones que se dispararon y se realizó un análisis utilizando expresiones regulares:



A través de un script de python se lee el log y este devuelve como resultado aquellos T-invariantes que no finalizaron, esto se debe a que el programa detiene todos los hilos inmediatamente después de que se cumple la condición de que se realizaron 1000 T-invariantes.

```
TI1: 1017 - TI2: 1001 - TI3: 1001
 ecastellano@notebook  ~/Facultad/Concurrente/TP_Final_Concurrente  rdp_temp ±  python3 scripts/test.py
Error de T-Invariantes:
T1T3
```

En la imagen anterior se puede ver como T1 y T3 pertenecen al T-Invariante de la secuencia T1T3T5T6 que no ha podido finalizar, si se agregan las transiciones T5 y T6 al archivo log el script da por exitoso el análisis de los invariantes.

```
 ecastellano@notebook  ~/Facultad/Concurrente/TP_Final_Concurrente  rdp_temp ±  python3 scripts/test.py
Éxito de T-Invariantes.
```

Conclusiones

En base a la red de Petri proporcionada, se encontró que la red no era viva, ya que presentaba un deadlock y se bloqueaba. Para solucionar este problema, se agregó una plaza extra (P14) con un token que protege los recursos de las plazas P8, P9 y P10, sin modificar los T-invariantes presentes en la red original. A través del análisis de concurrencia, se verificó que el grado de concurrencia disminuyó levemente respecto a la red original, pero se soluciona el problema del deadlock.

Luego, se desarrolló un programa en Java que modela el comportamiento de la red de Petri utilizando threads. Se implementó un monitor de concurrencia que protege los recursos junto a una política que determina el orden de ejecución de las transiciones.

La política verifica que se realice una cantidad equilibrada de T-invariantes, monitoreando los contadores de T-invariantes ejecutados hasta el momento y desbloqueando la transición que cumpla con el T-invariante menos ejecutado hasta ese momento.

Finalmente, en la implementación del programa, se agregó la semántica temporal a aquellas transiciones que deben ser temporizadas (T2, T3, T4, T5, T6, T8, T9 y T10), con un valor de alfa de 1 ms y un beta de 10000 ms (un valor muy grande para que nunca se desensibilice una transición por pasarse de la marca). Si una transición está sensibilizada por la cantidad de tokens necesarios pero no por el tiempo, esta transición se duerme el tiempo faltante hasta alcanzar la marca alfa y luego vuelve a verificar si está sensibilizada.

Para verificar el correcto funcionamiento de la red, se guarda en un archivo log todas las transiciones ejecutadas y, utilizando un script de Python, se verifica con una expresión regular si el resultado pertenece al dominio del lenguaje descrito por la red.

La implementación del monitor garantiza una ejecución concurrente de los procesos, a costa de una carga temporal en realizar una secuencia de T-invariante, esto se debe a que los hilos se bloquean ya sea por no estar sensibilizados o por no cumplir con el balance de los T-invariantes. Si los T-invariantes se ejecutan en un sistema totalmente paralelizado y con recursos ilimitados no se verán afectados por los demás T-invariantes, siendo este el caso ideal. Por el contrario si todos los T-invariantes se ejecutan de forma totalmente secuencial este sería el peor de los casos, ya que todos los T-invariantes se afectan continuamente.

Para analizar cuánto tiempo se demora en ejecutar 1000 T-invariantes respecto al peor de los casos y el ideal, se ejecutó el programa sin semántica temporal y se midió el tiempo que se demoró en que cada T-invariante sea ejecutado 1000 veces.

En promedio se registró el siguiente tiempo de ejecución con el programa funcionando concurrentemente:

```
Tiempo :443ms
```

Para el caso ideal se midió el tiempo de ejecución de solo un T-invariante a la vez, deshabilitando el chequeo de balance de la política.

Para el T-invariante 1 se registró:

```
Tiempo :263ms
```

Para el T-invariante 2 se registró:

```
Tiempo :290ms
```

Para el T-invariante 3 se registró:

```
Tiempo :190ms
```

Se puede observar que el tiempo de ejecución con el T-invariante más lento es menor que el tiempo de ejecución del programa implementado concurrentemente.

Por último el peor de los casos implica que el programa se ejecutó totalmente secuencial, implicando la suma de los 3 tiempos del caso anterior, por lo que el tiempo de ejecución sería de 743 ms.

De esta forma se puede concluir que el tiempo de ejecución del programa concurrente está más cerca del caso ideal que del peor de los casos.

Por otra parte si se analiza considerando que se busca que se disparen un total de 1000 T-Invariantes, para cada caso ideal serian 333 ejecuciones de T-Invariantes por cada TI existente, por lo que los tiempos serían de:

Para el T-invariante 1 se registró:

```
Tiempo :169ms
```

Para el T-invariante 2 se registró:

```
Tiempo :175ms
```

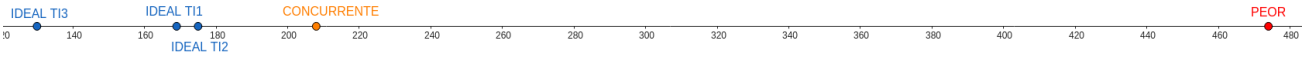
Para el T-invariante 3 se registró:

```
Tiempo :130ms
```

Para el peor de los casos nuevamente se realiza la suma de los 3 tiempos con un total de 474ms.

Anteriormente vimos que 1000 ejecuciones de T-Invariantes en total nos da un tiempo de:

```
Tiempo :208ms
```



Se puede observar nuevamente que se aproxima más a los valores ideales que al peor de los casos.