Trabajo práctico N°2

Redes de Petri y Monitores de concurrencia

Integrantes del grupo:

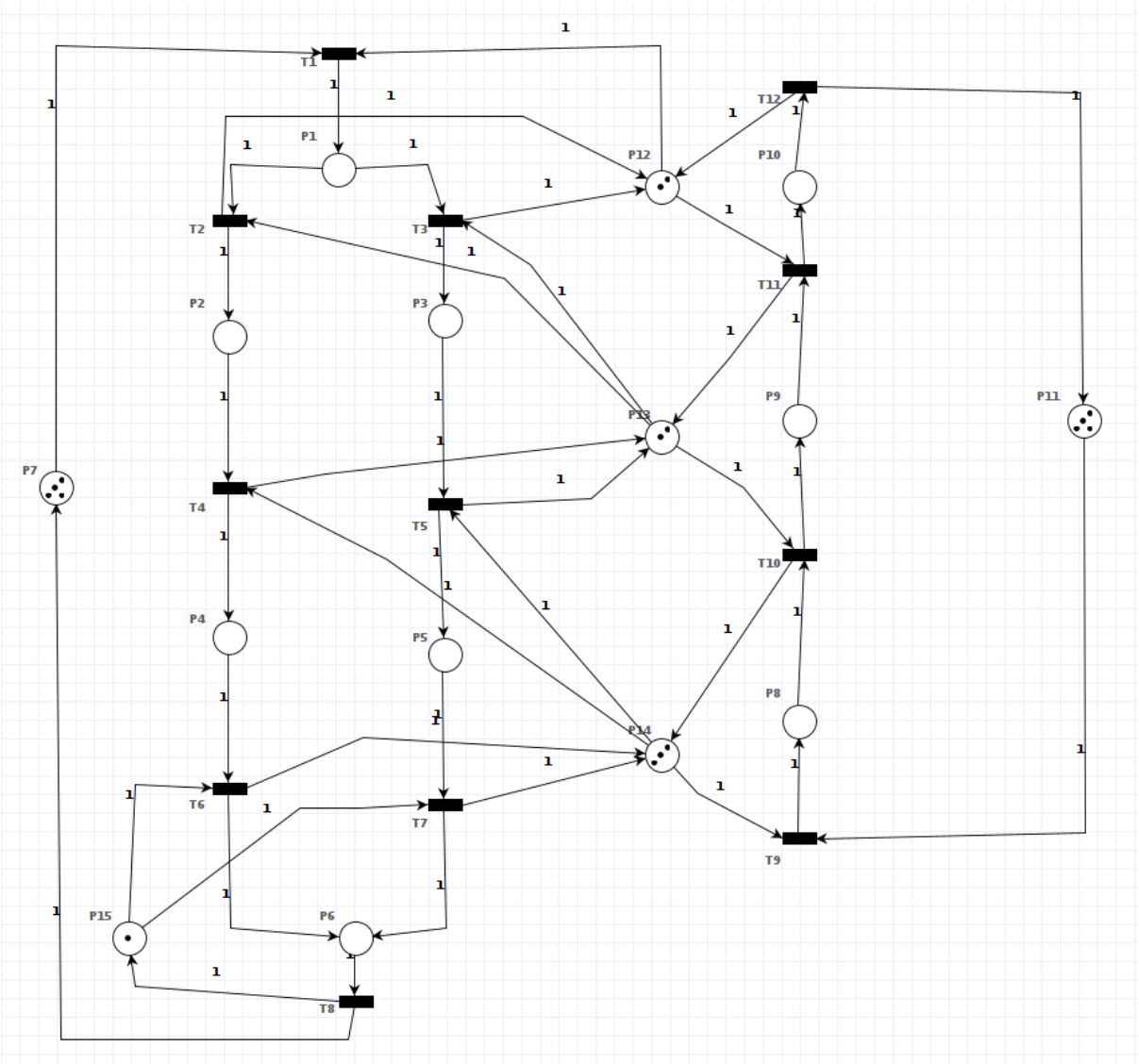
LAVEZZARI, Fausto	42344364
MAFFINI, Agustin	41813209
SIAMPICHETTI, Gino	41411540

Profesores:

Ventre, Luis Orlando
Ludemann, Mauricio

ÍNDICE

Semántica de la red



Decidimos aplicar la semántica de **Desarrollo de placas de circuito impreso (PCBs).** A continuación se describen el significado de las plazas y transiciones:

- **P1 a P6**: Representan las etapas del proceso de impresión de las placas:
  - P1: Pedidos de placas a imprimir
  - P2 y P3: Impresión de las placas
  - P4 y P5: Integración de componentes
  - P6: Empaquetado y etiquetado
- **P7 y P11:** Representan buffer auxiliares para contener los sistemas en diferentes etapas del desarrollo. Son lugares inactivos.
- **P8 a P10:** Representan las etapas del proceso de diseño de placas.
  - P8: Análisis y diseño preliminar
  - P9: Desarrollo del diseño en herramienta de software
  - P10: Control de calidad

- **P12 a P15:** Representan los recursos compartidos por las diferentes etapas del desarrollo. Principalmente trabajadores y equipo que están involucrados en ambas secciones del proceso.
- **T1 a T12:** Representan la actividad que provoca un cambio de etapa en el desarrollo
    - T1: Ingreso de pedido de impresión
    - T2 y T3: Aceptación de pedido
    - T4 y T5: Envío a integración
    - T6 y T7: Envío a empaquetado
    - T8: Fin del proceso total
    - T9: Inicio de etapa de diseño
    - T10: Aprobación del diseño preliminar
    - T11: Envío del diseño a control de calidad
    - T12: Fin de la etapa de diseño

Descripción de actividades.

P1: Pedidos de placas a imprimir

Esta plaza representa la etapa inicial del proceso, donde se reciben y acumulan los pedidos de las placas de circuito impreso. Actúa como el punto de entrada del sistema, indicando la demanda y las especificaciones de los PCBs que deben ser impresos.

P2 y P3: Impresión de las placas

Estas plazas representan la fase de impresión de las placas de circuito impreso. Por ejemplo la aplicación de la máscara de soldadura, serigrafía, aplicación de la capa de cobre y el grabado del diseño del circuito.

P4 y P5: Integración de componentes

Representan la etapa de integración de componentes en las placas. En estas etapas se realizan la selección y colocación de componentes, y la soldadura de estos a la placa.

P6: Empaquetado y etiquetado

Esta plaza representa la etapa final del proceso de producción, donde las PCBs terminadas son empaquetadas y etiquetadas para su envío. Incluye la revisión final de calidad y la preparación para la distribución.

P8 a P10: Etapas del proceso de diseño de placas

- P8: Análisis y diseño preliminar: Aquí se realiza el esbozo inicial del diseño de las PCBs, incluyendo la definición de requisitos y la conceptualización del diseño.
- P9: Desarrollo del diseño en herramienta de software: Esta plaza representa el uso de software especializado para desarrollar el diseño detallado del circuito y la distribución de los componentes.
- P10: Control de calidad: Esta etapa implica la revisión y verificación del diseño de las PCBs para asegurar que cumplen con los estándares y requisitos necesarios.

Propiedades de la red

Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	true

Imagen 1: net-state analysis - *Pipe*

Deadlock

En la red indicada para la realización de este trabajo, podemos notar que es una red que no está **libre de interbloqueo (Deadlock)**, esto significa que haya una cierta secuencia de disparo de las transiciones que evite que la red pueda seguir evolucionando. En el caso de un sistema real, un deadlock representa un estancamiento. Por ejemplo, en nuestro caso, esto se produce cuando para avanzar en una siguiente etapa se requiere de un recurso que no está disponible por estar siendo utilizado en otra.

Una secuencia que nos lleva a un estado de Deadlock es T1-T1-T2-T3-T1-T1-T9-T9-T9

En cuanto a la propiedad de **liveness (o vivacidad)** de la red, al no estar libre de interbloqueo, existe una marca en la cual no se puede disparar ninguna transición, por ende, la red no está viva. Ya que para esto, todas las transiciones tienen que ser alcanzables a partir de cualquier marca.

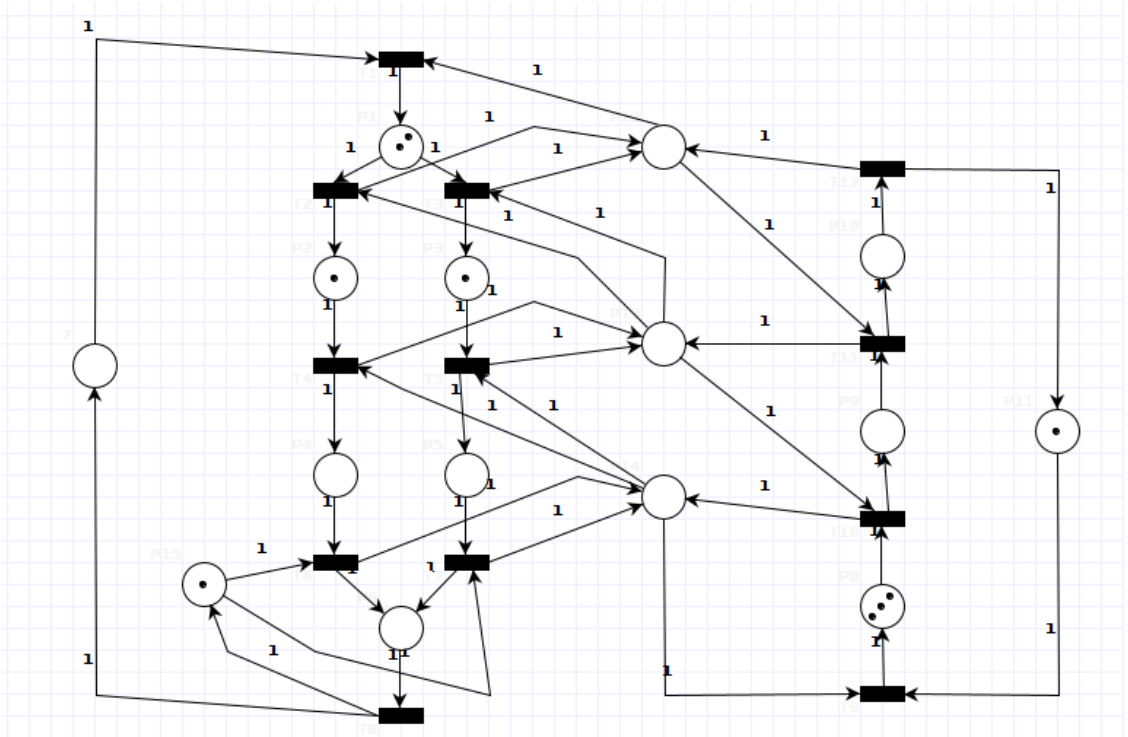


Figura 1: Red Interbloqueada

Bounded

La propiedad de **limitación (o Bounded**) de una red, significa que el marcado de una plaza nunca puede superar un valor constante *k.* Esto significa que el número de tokens de una plaza nunca podrá crecer de manera inesperada, ya que estará limitada por este valor *k.* Por ejemplo, en un sistema de desarrollo de PCBs, una red de Petri limitada podría asegurar que nunca se acumulen más pedidos de los que se pueden procesar, o que los recursos como maquinaria o personal no se sobrecarguen más allá de su capacidad operativa.

La propiedad de **seguridad (Safe)**, asegura que la red no puede alcanzar estados no deseados, ya que significa que en ningún mercado de la red las plazas pueden tener un número de tokens mayor a uno. Esta propiedad es una caso particular de una red limitada o bounded, donde el valor límite *k* es igual a uno.

Invariantes

Un invariante es una expresión lógica que, cuando es verdadera en un estado inicial del sistema, también es verdadera en cualquier estado alcanzable mediante la ejecución de transiciones. Son propiedades intrínsecas de la red y su marcado inicial

Invariantes de Transición

Una **invariante de transició**n es una secuencia de disparo de las transiciones de forma tal que desde una marca ***m****,* se puede alcanzar nuevamente la marca ***m****.* Esto también se conoce como **componente repetitivo** y permite capturar el estado cíclico de la red. En el caso de nuestra red tenemos los siguiente invariantes de transición:

T-Invariants

T1	T10	T11	T12	T2	T3	T4	T5	T6	T7	T8	T9
0	1	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	1	0	1	0	1	1	0
1	0	0	0	1	0	1	0	1	0	1	0

$$m \rightarrow T9T10T11T12 \rightarrow m$$
$$m \rightarrow T1T3T5T7T8 \rightarrow m$$
$$m \rightarrow T1T2T4T6T8 \rightarrow m$$

En nuestro caso la primera invariante indica un patrón repetitivo en la utilización y liberación de recursos para la etapa de análisis y diseño. Una vez ejecutada la secuencia, todos los recursos vuelven a su estado inicial. Los dos invariantes restantes indican ciclos repetitivos para la etapa de impresión, integración de componentes y de empaquetado y etiquetado de los PCBs, para lo cual también se utilizan y liberan recursos.

Invariantes de Plaza

Un invariante de plaza es una constante válida para cualquier marca **m** dada por la sumatoria de los tokens de un subconjunto de plazas de la red. Este subconjunto es el **componente conservador** de la red, es decir, es independiente del marcado inicial $\mathbf{m_0}$. Sin embargo, dicha constante resultante de la sumatoria **SÍ depende de $\mathbf{m_0}$.**

P-Invariants

P7	P1	P10	P11	P12	P13	P14	P15	P2	P3	P4	P5	P6	P8	P9
1	1	0	0	0	0	0	0	1	1	1	1	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	1	1
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	1	0	0	0	0	1
0	0	0	0	0	0	1	0	0	0	1	1	0	1	0
0	0	0	0	0	0	0	1	0	0	0	0	1	0	0

Si consideramos M(Pi) como la cantidad de tokens presente en la plaza Pi, entonces nos quedan determinadas las siguientes ecuaciones.

P-Invariant equations

$$M(P7) + M(P1) + M(P2) + M(P3) + M(P4) + M(P5) + M(P6) = 4$$
$$M(P10) + M(P11) + M(P8) + M(P9) = 4$$
$$M(P1) + M(P10) + M(P12) = 2$$
$$M(P13) + M(P2) + M(P3) + M(P9) = 2$$
$$M(P14) + M(P4) + M(P5) + M(P8) = 3$$
$$M(P15) + M(P6) = 1$$

Estos invariantes en un sistema real como el nuestro pueden representar la cantidad de recursos disponibles para un conjunto de actividades (Plazas). Podemos tomar como ejemplo la última invariante M(P15) + M(P6) = 1, si observamos la red podemos analizar que tanto la transición T6 como la T7 que abastecen la plaza P6 requieren de un token de P15, una vez consumido por alguno de ellas se abastece con 1 token a P6 y solo cuando se dispare T8 y retorne el token a P15 van a poder volver a abastecer P6, de esta forma queda limitado que tanto P6 como P15 van a tener como máximo un recurso en ellas de forma excluyente entre sí. Todo este razonamiento queda expresado de forma resumida en el invariante mencionado.

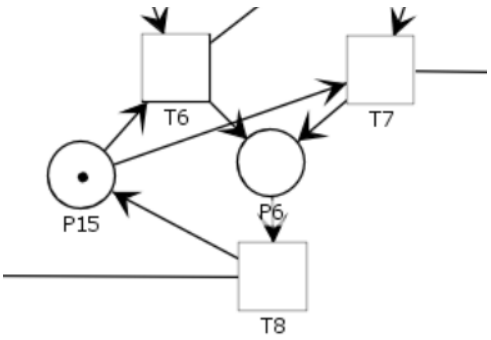


Figura 2: Ciclo de abastecimiento P6

Desbloqueo de la Red

Sifones

Para convertir la Rdp en una red libre de interbloqueo es útil utilizar el concepto de sifones. Estos se pueden definir como sigue:

Sea P´ = {P_1, P_2, …., P_r} un conjunto de plazas de una red de petri, el conjunto de transiciones de entrada de los lugares de P´y el conjunto de transiciones de salida de los lugares de P´, se denotan °P´y P´° respectivamente. Un sifón es un conjunto de lugares P´ tal que el conjunto de transiciones de entrada o °P´se incluye en el conjunto de transiciones de salida o P´°, es decir:

$$°P' \subseteq P'°$$

Dicho de otra manera, un conjunto de plazas P´ en una red de Petri se considera un sifón si para cada transición t que tiene una plaza de P´ como salida, al menos una de las plazas de entrada de t también pertenece a P´.

Podemos definir como **sifón mínimo** a aquellos sifones que no contienen dentro de ellos un subconjunto de plazas que forman otro sifón. Los sifones mínimos de nuestra red de petri son los siguientes.

Minimal siphons

{P14, P4, P5, P8 }
{P13, P2, P3, P9 }
{P1, P10, P12 }
{P15, P6 }
{P10, P11, P8, P9 }
{P7, P1, P2, P3, P4, P5, P6 }
{P10, P12, P13, P2, P3 }
{P13, P14, P4, P5, P9 }
{P10, P12, P13, P14, P4, P5 }

¿Por qué es importante la búsqueda de sifones?

Dada la definición, podemos concluir que un sifón va a perder tokens solamente si se disparan transiciones que se encuentran en el conjunto de salida pero no en el de entrada. Si esto sucede el sifón podría vaciarse completamente. Si esto sucede en plazas críticas para la red, la posibilidad de un interbloqueo se incrementa.

Asegurando que un sifón no pueda perder todos los tokens, es decir, que se genere un ciclo cerrado que lo realimente, siempre existirán transiciones dentro del mismo sensibilizadas y evitaremos el deadlock.

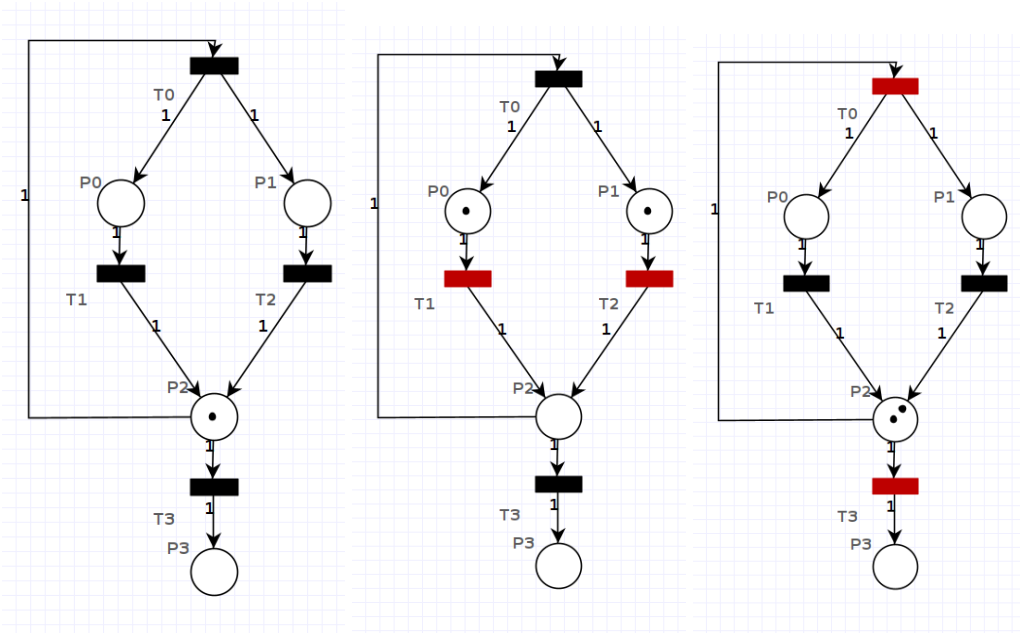
Entonces, ahora debemos encontrar aquellos sifones que puedan perder tokens, a estos los llamamos **sifones mínimos estrictos (SMS).** Para saber esto podemos observar los invariantes de plaza, ya que si un sifón mínimo contiene un invariante de plaza, quiere decir que siempre va a contener dentro de él al menos la cantidad de tokens dada por la constante del invariante y no es necesario controlarlo.

Comparando los sifones mínimos con nuestros invariantes de plaza tenemos que los SMS son:

$$S_1 = \{P_{10}, P_{12}, P_{13}, P_2, P_3\}$$
$$S_2 = \{P_{13}, P_{14}, P_4, P_5, P_9\}$$
$$S_3 = \{P_{10}, P_{12}, P_{13}, P_{14}, P_4, P_5\}$$

Para evitar el vaciado de estos vamos a agregar plazas de control diseñadas específicamente para evitar que esto suceda. Para esto se sigue la siguiente lógica.

Un sifón solamente puede ganar tokens cuando se disparan transiciones que forman parte de más entradas que de salidas. O sea, “ponen” más tokens de los que sacan. Veamos un ejemplo

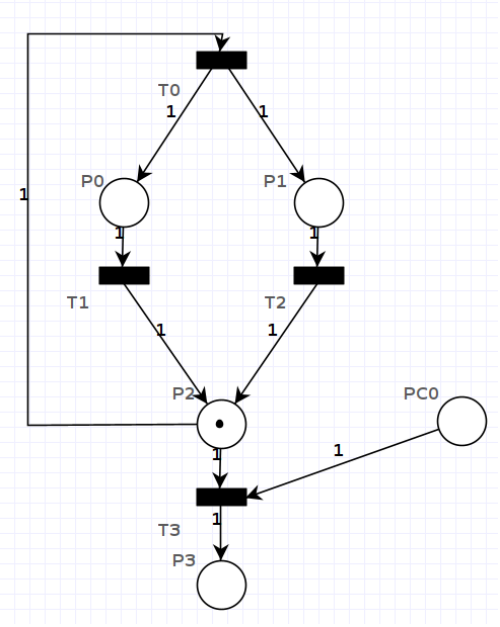


En esta red de ejemplo, tenemos un sifón formado por P0,P1 y P2. La transición T0 es entrada de 2 plazas {P0,P1} y salida de 1 {P2}. Como podemos ver, cada vez que se ejecuta T0 esta genera más tokens de los que consume, por ende, **agrega** tokens al sifón.

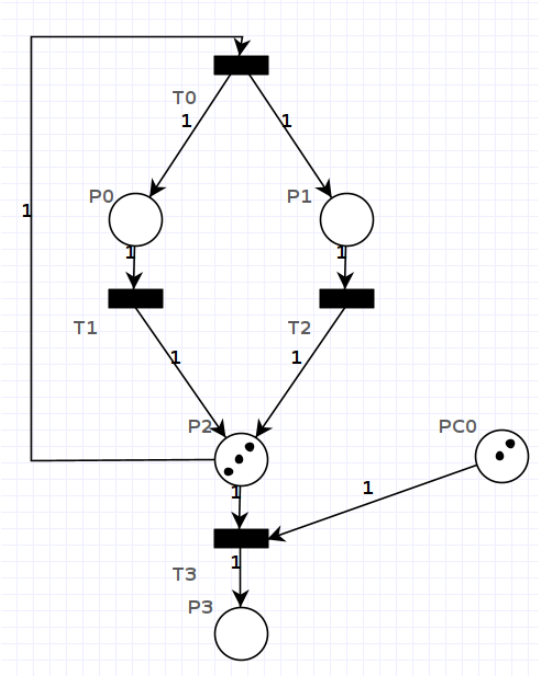
En el caso contrario tenemos T3, que cada vez que se ejecuta **quita** tokens al sifón, esto es porque forma parte de más salidas {P2}, que entradas {Ø}. Es decir, consume más tokens de los que genera.

Ahora sabiendo este concepto, podemos utilizarlo para diseñar nuestras plazas de control. Vamos a suponer que lo único que queremos es que el sifón no se vacíe del todo, que al menos siempre haya un token dentro de él. Para esto limitamos las transiciones que quitan

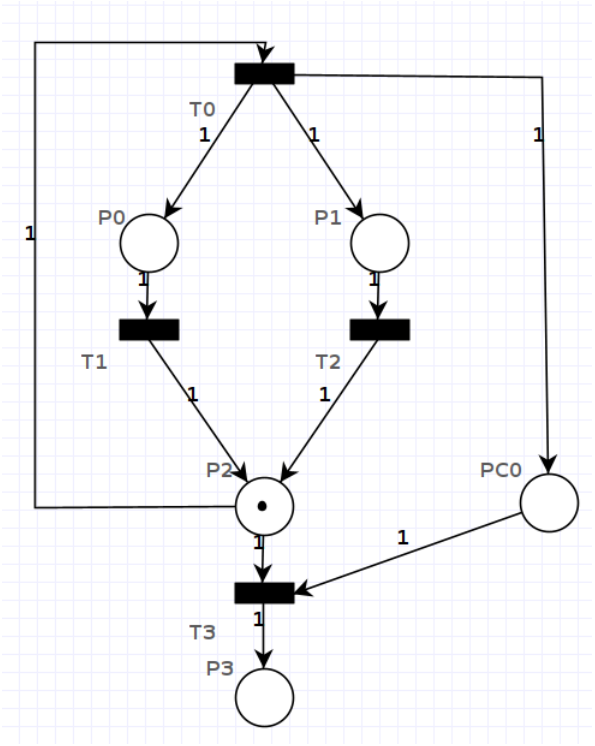
tokens (T3 en nuestro ejemplo) de forma que solo puedan dispararse cuando la cantidad de tokens dentro del sifón sea mayor a 1.



Conectando PC0 a T3 ahora esta no se va a poder disparar y quitar el único token disponible, ya que PC0 indica la cantidad de veces que se puede disparar sin vaciar el sifón (0 veces en este caso). Supongamos otro caso.



Ahora PC0 nos indica que se puede disparar T3 hasta 2 veces, dejando al menos un token en el sifón. Pero es notorio un inconveniente, no tiene en cuenta que justamente T0 agrega tokens, reponiendo o aumentando la cantidad de veces que podemos disparar T3. Como T0 agrega 1 token cada vez que se dispara, colocamos un arco de peso 1 como entrada de PC0



Nuestra plaza de control queda completamente implementada. Ahora cada vez que un token se sume al sifón la marca de PC0 se incrementa en 1 y cada vez que salga se resta 1. La marca inicial del sifón menos la marca inicial de PC0 nos da la cantidad de tokens mínima que va a tener el sifón en todo momento.

Ahora estamos en condiciones de definirlo de forma formal:

Tomando a V_Z como un conjunto de sifones mínimos no controlados de la red, y a z Si como a cada uno de los sifones pertenecientes a V_Z . Para cada uno de estos sifones, z creamos una invariante agregando a la red una plaza C_{Si} , llamada “Plaza de Control Local”. Si

Para cada $S_i \in V_Z$ asociamos una plaza con:

- $C_{Si^\circ} = \{ t \in S_i^\circ \ / \ |^\circ t \cap S_i| > |t^\circ \cap S_i| \}$
- $^\circ C_{Si} = \{ t \in {}^\circ S_i \ / \ |t^\circ \cap S_i| > |^\circ t \cap S_i| \}$
- $M_0(C_{Si}) = K_i - 1 \ ; \ K_i = M_0(S_i)$

Determinar Plazas de Control

Para determinar las plazas de control primero necesitamos las transiciones de entrada y de salida de los sifones que vamos a utilizar. Para ellos necesitamos la matriz de incidencia para saber cómo afectan las transacciones a las plazas:

Combined incidence matrix *I*

	T1	T10	T11	T12	T2	T3	T4	T5	T6	T7	T8	T9
P7	-1	0	0	0	0	0	0	0	0	0	1	0
P1	1	0	0	0	-1	-1	0	0	0	0	0	0
P10	0	0	1	-1	0	0	0	0	0	0	0	0
P11	0	0	0	1	0	0	0	0	0	0	0	-1
P12	-1	0	-1	1	1	1	0	0	0	0	0	0
P13	0	-1	1	0	-1	-1	1	1	0	0	0	0
P14	0	1	0	0	0	0	-1	-1	1	1	0	-1
P15	0	0	0	0	0	0	0	0	-1	-1	1	0
P2	0	0	0	0	1	0	-1	0	0	0	0	0
P3	0	0	0	0	0	1	0	-1	0	0	0	0
P4	0	0	0	0	0	0	1	0	-1	0	0	0
P5	0	0	0	0	0	0	0	1	0	-1	0	0
P6	0	0	0	0	0	0	0	0	1	1	-1	0
P8	0	-1	0	0	0	0	0	0	0	0	0	1
P9	0	1	-1	0	0	0	0	0	0	0	0	0

Para que el análisis sea más sencillo, hacemos las matrices de incidencia por sifones. Determinaremos las marcas para que el sifón al menos tenga un token disponible.

$$S_1 = \{P_{10}, P_{12}, P_{13}, P_2, P_3\}$$

	T1	T2	T3	T4	T5	T6	T10	T11	T12
P2	0	1	0	-1	0	0	0	0	0
P3	0	0	1	0	-1	0	0	0	0
P10	0	0	0	0	0	0	0	1	-1
P12	-1	1	1	0	0	0	0	-1	1
P13	0	-1	-1	1	1	0	-1	1	0

T1 -> Quita un token
T2 -> Agrega un token
T3 -> Agrega un token
T10 -> Quita un token
T11-> Agrega un token

{T2,T3,T11} -> Cs1 -> {T1,T10}
M(S1) = 4 -> M(Cs1) = 3

$S_2 = \{P_{13}, P_{14}, P_4, P_5, P_9\}$

	T2	T3	T4	T5	T6	T7	T9	T10	T11
P4	0	0	1	0	-1	0	0	0	0
P5	0	0	0	1	0	-1	0	0	0
P9	0	0	0	0	0	0	0	1	-1
P13	-1	-1	1	1	0	0	0	-1	1
P14	0	0	-1	-1	1	1	-1	1	0

T2 -> Quita un token
T3 -> Quita un token
T4 -> Agrega un token
T5 -> Agrega un token
T9 -> Quita un token
T10 -> Agrega un token

{T4,T5,T10} -> Cs2 -> {T2,T3,T9}
M(S2) = 5 -> M(Cs2) = 4

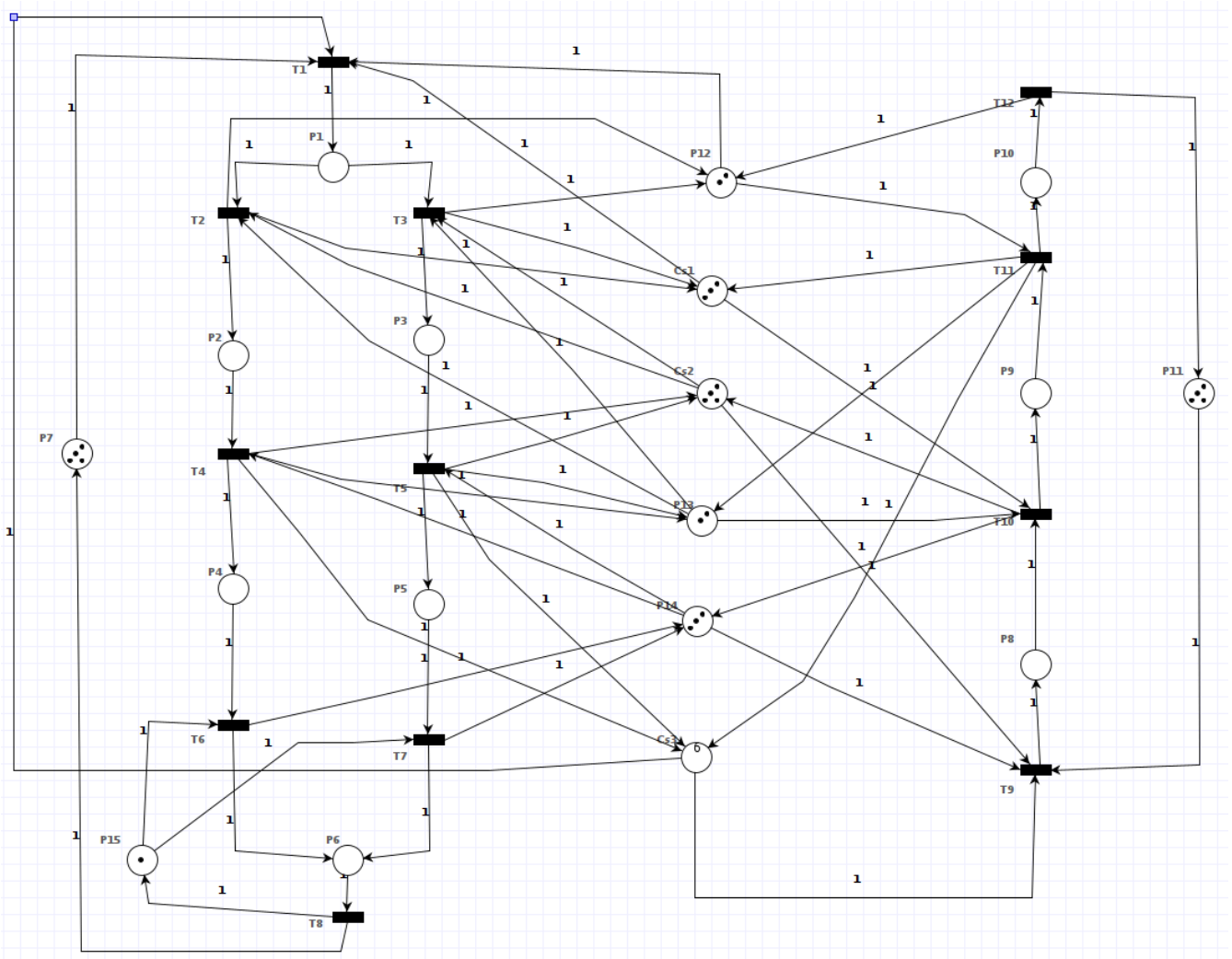
$S_3 = \{P_{10}, P_{12}, P_{13}, P_{14}, P_4, P_5\}$

	T1	T2	T3	T4	T5	T6	T7	T9	T10	T11	T12
P4	0	0	0	1	0	-1	0	0	0	0	0
P5	0	0	0	0	1	0	-1	0	0	0	0
P10	0	0	0	0	0	0	0	0	0	1	-1
P12	-1	1	1	0	0	0	0	0	0	-1	1
P13	0	-1	-1	1	1	0	0	0	-1	1	0
P14	0	0	0	-1	-1	1	1	-1	1	0	0

T1 -> Quita un token
T4 -> Agrega un token
T5 -> Agrega un token
T9 -> Quita un token
T12 -> Agrega un token

{T4,T5,T11} -> Cs3 -> {T1,T9}
M(S3) = 7 -> M(Cs3) = 6

Red Desbloqueada



Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	false

T-Invariants

T1	T10	T11	T12	T2	T3	T4	T5	T6	T7	T8	T9
1	0	0	0	0	1	0	1	0	1	1	0
1	0	0	0	1	0	1	0	1	0	1	0
0	1	1	1	0	0	0	0	0	0	0	1

Como se puede observar, se logró eliminar el problema de interbloqueo de la red sin modificar los invariantes de transición (como solicita la consigna)

Análisis de marcado

Ahora que tenemos la red desbloqueada podemos hacer un análisis del marcado de las mismas. Para esto fue necesario utilizar el software *petrinator*, ya que este nos brinda todos los marcados posibles.

En una primer instancia podemos comparar la cantidad de marcas alcanzables

Red original: 1488
Red libre de deadlock: 1423

Podemos ver que la red nueva tuvo un leve decremento en la cantidad total de marcas. Esto no es algo malo, ya que producto de las plazas de control lo que logramos fue limitar marcas que nos lleven a un posible estado de deadlock.

Otra característica a analizar es la cantidad de tokens promedio en las plazas de actividades. Una mayor cantidad de estos podría significar un aumento en el paralelismo de la red, ya que más plazas estarán activadas al mismo tiempo.

Petrinator nos brinda un html con todas las marcas posibles, junto con las marcas alcanzables desde estas. Para poder analizar los tokens promedio realizamos un script en python que lee este html, se queda solamente con el dato de las marcas y hace el cálculo del promedio. Los resultados fueron los siguientes.

Plazas de Actividad			
	Red Desbloqueada	Red Bloqueada	Diferencia (%)
P1	0.49	0.62	-20,9%
P2	0.38	0.49	-22,4%
P3	0.38	0.39	-2,5%
P4	0.59	0.57	3,5%
P5	0.59	0.59	0%
P6	0.42	0.41	2,4%
P8	0.81	0.87	-6,9%
P9	0.53	0,43	23,2%
P10	0.66	0.46	43,5%

Como podemos observar, los resultados no son inequívocos respecto a la mejora de la red desbloqueada. Por un lado tenemos que las plazas involucradas en el invariante de transición de la derecha de la red {P8,P9,P10} en rasgos generales aumentaron su cantidad de tokens promedio. En cambio por otro lado en resto de plazas, salvo {P1,P2} los valores se mantuvieron relativamente similares. Si bien existieron algunos decrementos, la red ahora está libre de deadlock, cosa sumamente importante, así que nos vemos en condiciones de afirmar que se hizo una mejora a la misma.

Implementación

Para poder implementar la red de petri, previamente tenemos que determinar el número de hilos que va a necesitar la red y qué responsabilidad debe tener cada uno. Todo esto con el fin de garantizar el mayor paralelismo en la ejecución del sistema.
Vamos a utilizar una metodología llamada “Sistema de procesos Secuenciales Simples con Recursos (*S^3PR*)”, la cual nos permitirá determinar el número de hilos máximos que podemos tener simultáneamente, el número de hilos máximo para que se pueda llevar a cabo toda la ejecución del sistema, y además nos permitirá determinar la responsabilidad de cada uno.

Determinación de hilos máximos activos simultáneos

Paso 1: Obtener los IT del sistema.

En nuestra red, los invariantes de transiciones ya fueron determinados y son los siguientes:

$$IT_1 = \{T9, T10, T11, T12\}$$
$$IT_2 = \{T1, T2, T4, T6, T8\}$$
$$IT_3 = \{T1, T3, T5, T7, T8\}$$

Paso 2: Para cada IT, debemos determinar el conjunto de plazas asociadas al mismo:

$$PI_i = U_{\forall\ t\ \varepsilon\ Inv\ °t} \cup U_{\forall\ t\ \varepsilon\ Inv\ t°}$$

Donde PI_i es el conjunto de plazas asociadas al i ésimo IT.
En el caso de nuestra red, tenemos que los conjuntos de plazas asociadas a cada IT son:

$$PI_1 = \{P8, P9, P10, P11, P12, P13, P14, Cs1, Cs2, Cs3\}$$
$$PI_2 = \{P1, P3, P5, P6, P7, P12, P13, P14, P15, Cs1, Cs2, Cs3\}$$
$$PI_3 = \{P1, P2, P4, P6, P7, P12, P13, P14, P15, Cs1, Cs2, Cs3\}$$

Paso 3: Obtener los conjuntos de plazas activas PA_i para cada IT.

Esto consiste en eliminar de los conjuntos PI_i las plazas de recursos o idle, ósea:

$$PA_i = PI_i - \{P_{Irestricciones} \cup P_{Irecursos} \cup P_{Iidle}\}$$

Entonces, para los PI_i de nuestra red tenemos:

$$PA_1=\{P8, P9, P10\}$$
$$PA_2=\{P1, P3, P5, P6\}$$
$$PA_3=\{P1, P2, P4, P6\}$$

Paso 4: Obtener el conjunto MA, el cual es el conjunto de todos los marcados posibles de las plazas de los conjuntos PA_i
Para ello obtuvimos todos los marcados posibles de la red utilizando la herramienta “*Petrinator”*. Pero estos marcados incluyen plazas de recursos, de control e idle, por lo tanto fue necesario removerlas y finalmente nos quedamos con los marcados de las plazas de actividades.

Paso 5: De cada marcado de MA, realizar la suma de cada uno. De todas estas sumas, nos quedamos con el valor máximo, el cual indica el número de hilos máximos activos simultáneos que puede tener nuestra red.

Después de realizar todos los pasos del algoritmo en nuestra red, obtuvimos que el **número máximo de hilos simultáneos activos** que puede tener nuestra red **es de 8**

Determinación de responsabilidad de hilos

La ejecución del sistema consiste en la ejecución de las transiciones de los diferentes IT. Dicha ejecución va a estar a cargo de hilos, y la misma puede estar segmentada si el IT no está libre de conflictos.
Vamos a definir un **conflicto** cuando un IT comparte transiciones con otro IT, lo cual se representa por medio de forks (bifurcaciones) o joins (uniones)
Entonces, para determinar la segmentación de cada transiciones pueden existir 3 casos:

- **Caso 1**: El IT está libre de conflictos o uniones, por lo tanto se trata de un IT lineal (o secuencial), y la ejecución del mismo se llevará a cabo por un solo segmento de ejecución.
- **Caso 2:** En el caso de que dos o más IT compartan transiciones en conflicto estructural (fork), la ejecución del IT se segmenta y queda a cargo de múltiples segmentos de ejecución, uno para las transiciones antes del conflicto, y un segmento por invariante posterior al conflicto.
- **Caso 3:** En el caso de que dos o más IT presenten plazas de unión (join), la ejecución del IT se segmenta y queda a cargo de múltiples segmentos de ejecución, uno por invariante antes del join, y posterior al join habrá un segmento para todas las transacciones.

Siendo los invariantes de nuestra red:

$IT_1 = \{T9, T10, T11, T12\}$
$IT_2 = \{T1, T2, T4, T6, T8\}$
$IT_3 = \{T1, T3, T5, T7, T8\}$

Se observa que **IT_1**, cumple con el **caso 1**, osea es una transición lineal o secuencial, por lo tanto s**e define un solo segmento de ejecución**

$S_1 = \{T9, T10, T11, T12\}$
$PS_1 = \{P8,P9,P10\}$

Para **IT_2 e IT_3**, se observa que h**ay conflicto estructural** entre ambas, lo cual pertenece al **caso 2**, **entonces se definen tres segmentos de ejecución**, uno para las transiciones antes del conflicto y dos por IT (en nuestro son IT_2 e IT_3)

$S_2=\{T1\}$
$S_3=\{T2, T4, T6\}$
$S_4=\{T3, T5, T7\}$

$PS_1=\{P1\}$
$PS_2=\{P2, P4, P6$
$PS_3=\{P3, P5, P7\}$

Se puede apreciar que **IT_2 e IT_3**, **presentan** además **una plaza de unión o join**, por lo tanto también cumplen con el caso 3, entonces **podemos definir 3 segmentos de ejecución**, uno por cada invariante y otro posterior al join.
Los segmentos previos al join, son los mismos segmentos S_3 y S_4.

S_5={T8}
S_3={T2, T4, T6}
S_4={T3, T5, T7}

PS_5={P6}
PS_2={P2, P4,P6}
PS_3={P3, P5,P7}

Finalmente combinando todos los casos obtenemos los 5 segmentos involucrados.

S_1 = {*T9, T10, T11, T12*}
S_2={T1}
S_3={T2, T4,T6}
S_4={T3, T5, T7}
S_5={T8}

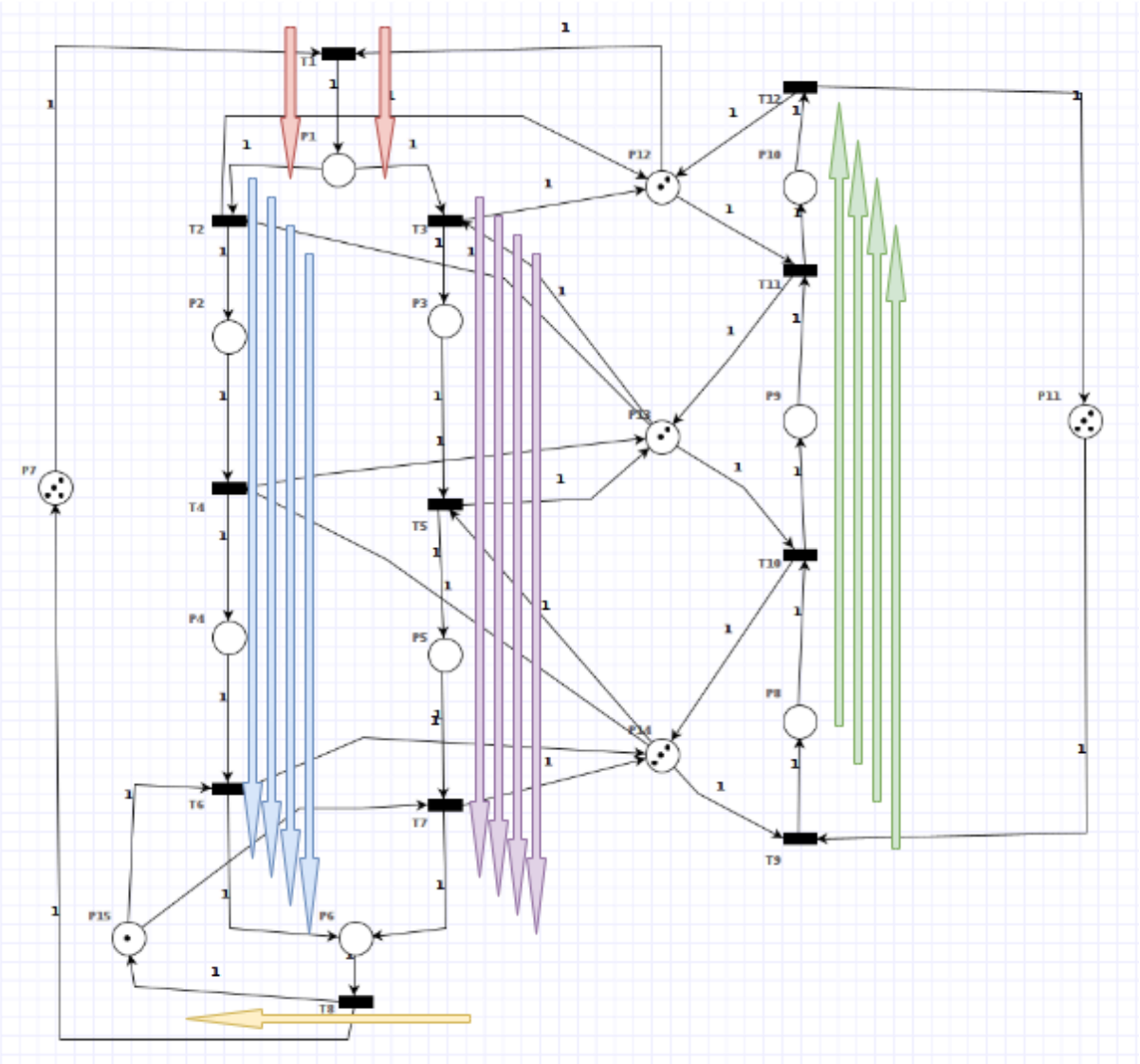
Determinación de hilos máximos por segmento.

Para el cálculo de hilos máximos necesarios se sigue un procedimiento muy parecido al de hilos máximos simultáneos.

1) Obtener los IT de la RdP, los cuales ya vimos son los siguientes
   IT_1 = {T9, T10, T11, T12}
   IT_2 = {T1, T2, T4, T6, T8}
   IT_3 = {T1, T3, T5, T7, T8}
2) Determinar los segmentos Si y las plazas PSi, usando el árbol de alcanzabilidad obtener todos los marcados para cada Si (MSi). Esto lo haremos utilizando la herramienta petrinator y filtrando las plazas que no nos interesan.
3) De cada MSi obtener el marcado máximo, este será la cantidad necesaria de hilos para ese Si (Max(MSi))

   Max(MS_1)= 4
   Max(MS_2)= 2
   Max(MS_3)= 4
   Max(MS_4)= 4
   Max(MS_5)= 1

4) Sumando todos los Max(MSi) obtenemos la cantidad necesaria para toda la red.
   Max(MS_1)+Max(MS_2)+Max(MS_3)+Max(MS_4)+Max(MS_5) = 15


T1
P1
T2
T3
P2
P3
T4
T5
P4
P5
T6
T7
P15
P6
T8
P7
P12
P13
P14
T12
P10
T11
P9
T10
P8
T9
P11

Construcción en Java

Podemos decir que una red de petri está compuesta por los siguientes elementos:

- Plazas
- Transiciones
- Tokens
- Arcos

Es por esto que debemos abordar este problema buscando representar estos elementos de alguna forma con objetos de java. Para esto tomamos las siguientes decisiones de diseño. Es importante destacar que dada la naturaleza de la red asumimos que el peso de todos los arcos de 1.

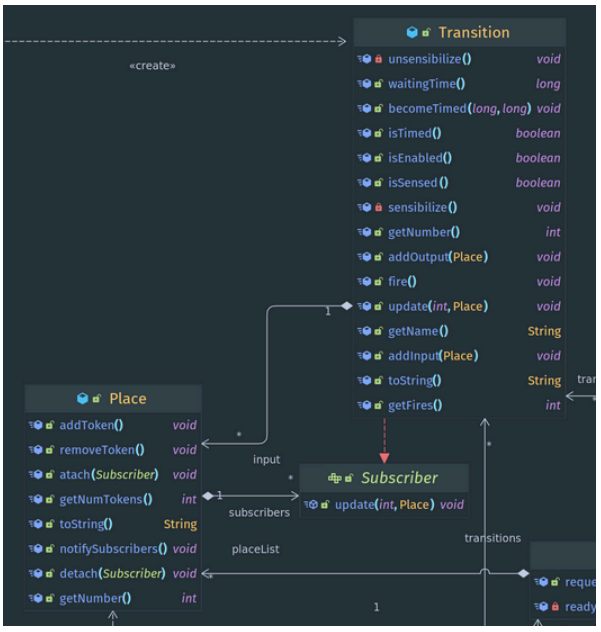
Plazas, Transiciones y Arcos

Lo primero que decidimos abordar fue el cómo relacionar las plazas con las transiciones, esto en la red está representado por los arcos, pero al ser una simple relación pensamos que era posible encararlo solamente marcando esa relación en algún campo.

Así, en nuestro sistema cada transición tiene como campos, plazas de salida y un mapa<Plazas, boolean> para las plazas de entrada con un flag para indicar si tiene tokens disponibles.

Con la adición de estos simples campos ya tenemos representada todas las relaciones de la red. Las plazas son simples contenedores de tokens que se referencian en el input y output de las transiciones.

Para que las transiciones se actualicen con los valores correspondientes a las plazas optamos por construir un **patrón observer** en el cual dichas transiciones se suscriben a las plazas, para que cuando se actualicen, mediante un notify() les llegue el nuevo valor de tokens.



Segmentos

Los segmentos son un conjunto de transiciones a los cuales se les puede asignar una cantidad de hilos de ejecución encargados de dispararlas. Es por esto que esta será nuestra clase Runnable que vamos a usar para instanciar los hilos. El método run() solamente intenta disparar las transiciones del segmento en bucle

```
public class Segment implements Runnable{
   private final String name;
   private final List<Transition> transitions;
   private final Monitor monitor;

   public Segment(String name,List<Transition> transitions, Monitor monitor)
{
       this.name = name;
       this.transitions = transitions;
       this.monitor = monitor;
   }

   public String toString() {
       return this.name;
   }

   public void run() {
       while(true) {
           for (Transition transition : transitions) {
               boolean fired = false;
               while(!fired)
                   fired = monitor.requestFire(transition);
           }
       }
   }
}
```

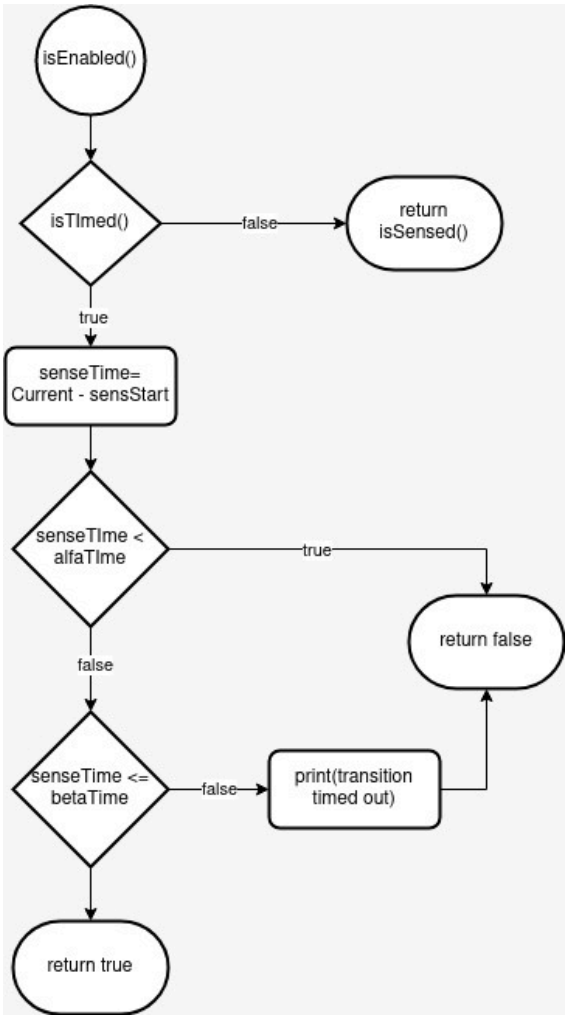
Transiciones temporales

Para considerar la existencia de transiciones temporales añadimos algunos campos y métodos a las transiciones. Por ejemplo alfaTime, betaTime y senseStart. A este último se le asigna un timestamp al momento de sensibilizar la transición, y cuando se consulta su estado se utiliza para calcular el tiempo total de sensibilizado y se lo compara con alfa y beta.

Se diferencia entre estar *Available* y *Sensed*. El primero indica si la transición está **lista para ser disparada**, mientras que el segundo solo si esta está sensibilizada. De esta forma se crean los métodos isEnabled() e isSensed(). En caso de que la transición no esté temporizada, llamar al método isEnabled() redirecciona directamente a isSensed().

También se agrega el método waitingTime(). indicando el tiempo en [ms] restante para que se supere alfaTime. En caso de que se sobrepase se retorna un valor negativo indicando cuánto pasó desde que se superó dicha barrera. Si la transición no está temporizada se

devuelve 0 ya que no se debe esperar. Si es temporizada pero no está sensibilizada se devuelve alfaTime, ya que es lo mínimo que va a tener que esperar.

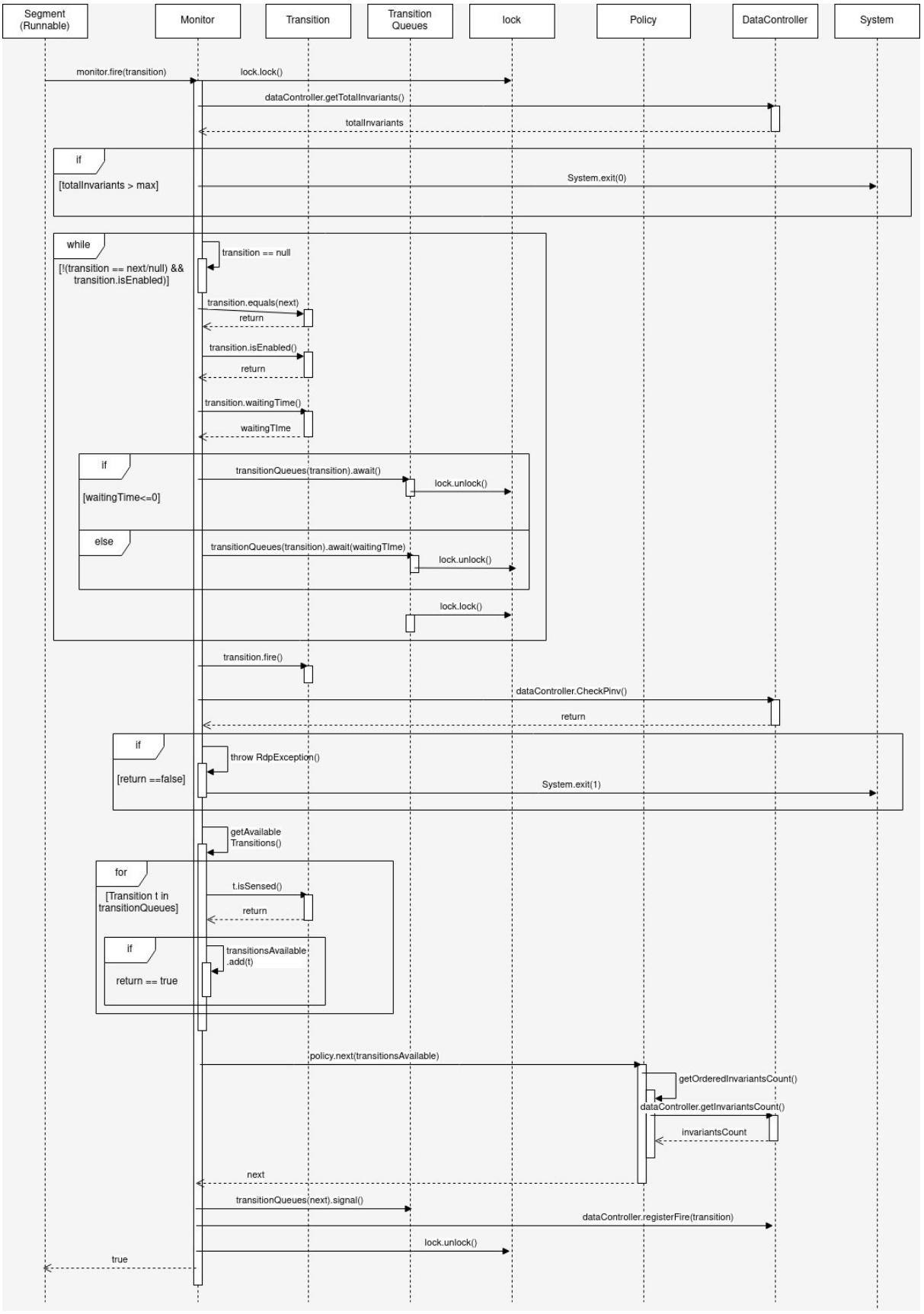


Monitor

Es necesario construir un monitor de concurrencia para controlar a los hilos que desean disparar las transiciones. El disparo debe realizarse en exclusión mutua para evitar conflictos.

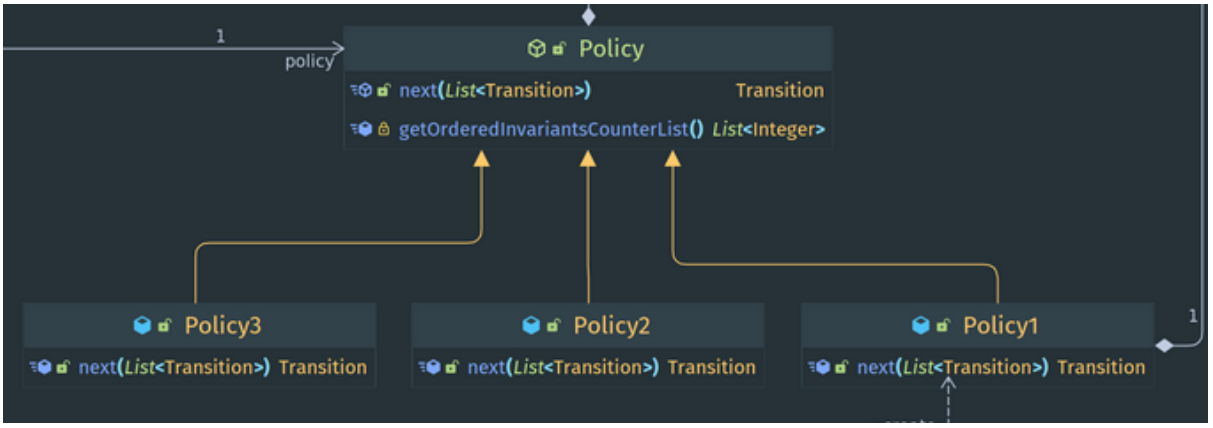
Para esto utilizamos un ReentrantLock, junto con colas de Condiciones, una para cada transición. Un hilo entrante comprueba si es el siguiente a ejecutar, en caso afirmativo prosigue, si no, o no está habilitada la transición aún, se pone a la espera en la cola respectiva a su transición. Las transiciones temporales habilitadas y las no temporales esperan de forma indefinida en la cola hasta recibir un signal, las temporales no habilitadas esperan en la misma cola pero durante el periodo dado por waitingTime(), para que puedan despertarse y volver a comprobar su estado, evitando la inanición del sistema. Luego de disparar se decide la siguiente transición mediante la política.

A continuación un diagrama de secuencia que indica el proceso de disparar una transición utilizando el monitor.


Segment (Runnable)
Monitor
Transition
Transition Queues
lock
Policy
DataController
System
monitor.fire(transition)
lock.lock()
dataController.getTotalInvariants()
totalInvariants
if
[totalInvariants > max]
System.exit(0)
while
[!(transition == next/null) && transition.isEnabled)]
transition == null
transition.equals(next)
return
transition.isEnabled()
return
transition.waitingTime()
waitingTIme
if
[waitingTime<=0]
transitionQueues(transition).await()
lock.unlock()
else
transitionQueues(transition).await(waitingTIme)
lock.unlock()
lock.lock()
transition.fire()
dataController.CheckPinv()
return
if
[return ==false]
throw RdpException()
System.exit(1)
getAvailable Transitions()
for
[Transition t in transitionQueues]
t.isSensed()
return
if
return == true
transitionsAvailable .add(t)
policy.next(transitionsAvailable)
getOrderedInvariantsCount()
dataController.getInvariantsCount()
invariantsCount
next
transitionQueues(next).signal()
dataController.registerFire(transition)
lock.unlock()
true

Política

Como lo que queremos hacer es probar diferentes políticas de decisión creamos una clase abstracta policy y luego clases que la heredan e implementan de diferente forma el método next(List<Transition> availableTransitionList) para decidir. Los detalles de los criterios de estas políticas los veremos más adelante.



DataController

Para llevar un control de la ejecución de la red creamos esta clase auxiliar que nos va a servir para diferentes funciones, como por ejemplo.

- Obtener la cantidad de invariantes disparados
- Determinar la cantidad de invariantes en ejecución
- registrar disparos
- verificar invariantes de plaza

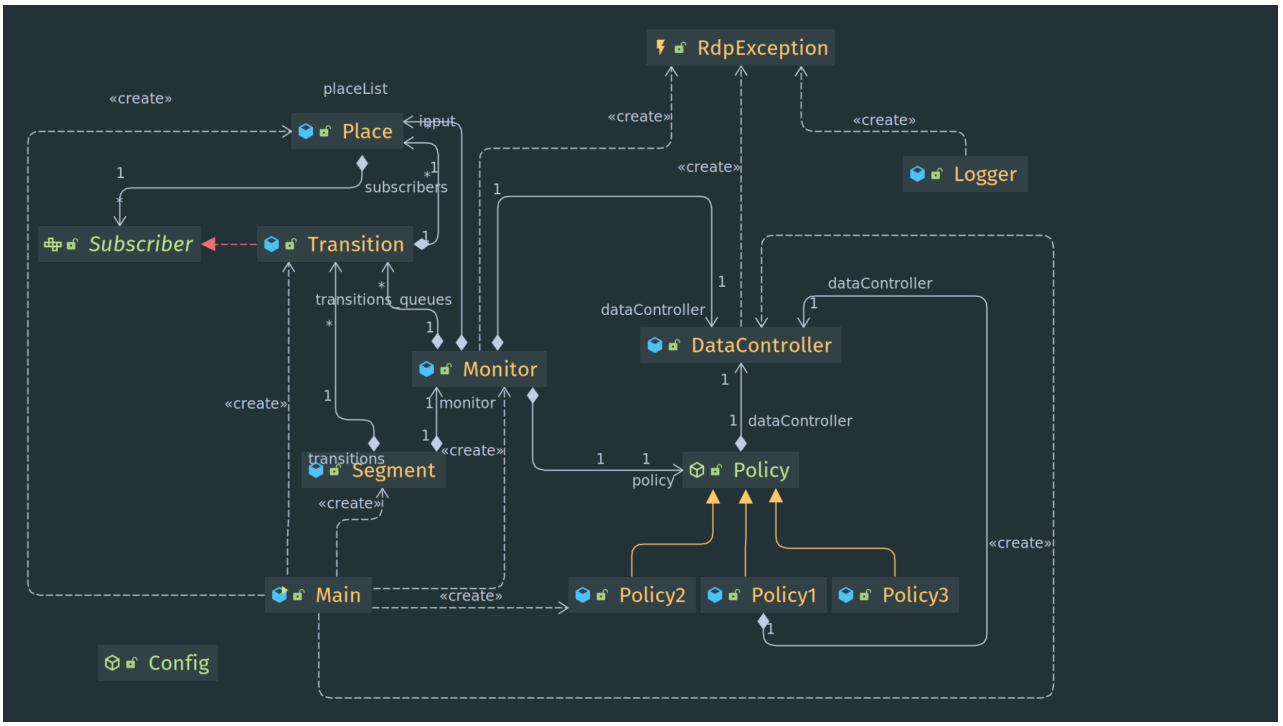
Logger

Simplemente una clase encargada de recibir toda la información proveniente del sistema y loguearse tanto en la consola como en el archivo final de log.

RdpException

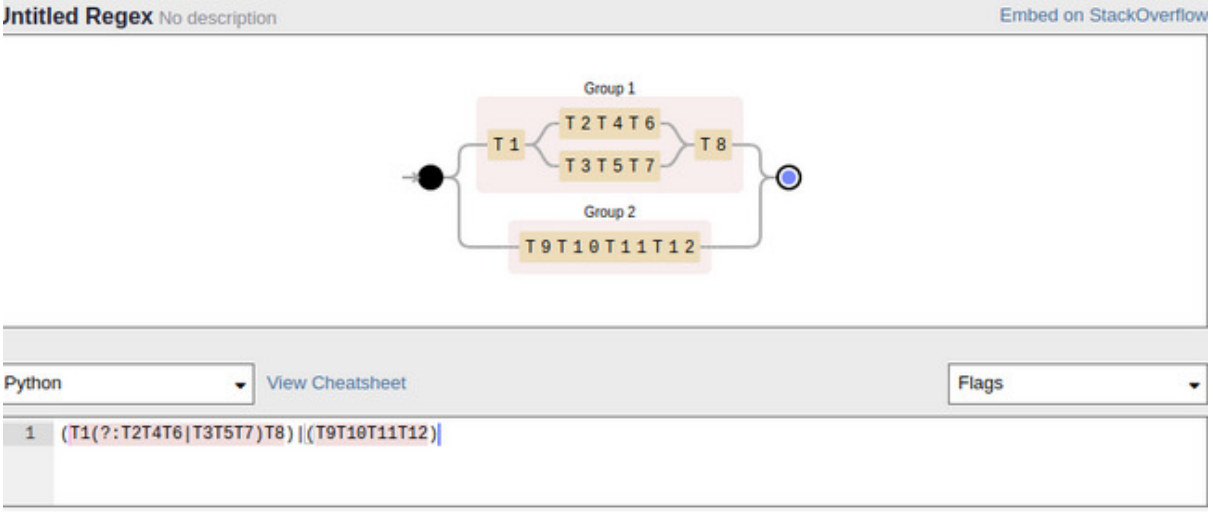
Excepciones personalizadas para administrar errores propios de este sistema, por ejemplo disparar transiciones no habilitadas.

Diagrama de clases completo



Obtención de datos

Como mencionamos anteriormente, mediante el DataController hacemos gran parte de la recopilación de datos que luego serán volcados en el log. De todas maneras hay otra información que la recopilamos haciendo un análisis mediante regex del archivo, por ejemplo la cantidad de disparo de las transiciones, o la verificación del completado de un invariante de forma estricta. Gracias a la herramienta debuggex generamos la siguiente expresión regular.



Los scripts de python utilizados se pueden encontrar en el repositorio.

Determinación de políticas

Realizamos 3 métodos diferentes para determinar cuál es la próxima transición a dispararse. En todos los casos el método next() recibe una lista con las transiciones **sensibilizadas**, esto quiere decir que una transición temporal que esté sensibilizada pero no dentro de la ventana de tiempo **es elegible por la política**.

Tomamos esta decisión porque si tomamos el subconjunto de transiciones habilitadas ocurre que en la mayoría de intentos no hay transiciones disponibles para elegir, haciendo que la política devuelva null dejando al primer hilo en llegar ejecutar la transición si está habilitada. Esto claramente favorece a las transiciones no temporizadas, provocando la inanición de las si temporizadas que comparten recursos. Aún así, el tiempo restante para estar habilitada puede ser un criterio a tener en cuenta.

Política 1

La más simple de todas, por primera intuición a la hora de buscar balancear la carga, se decide por la transición con menor cantidad de disparos históricos. En caso de empate, se decide por aquella que requiera esperar menos tiempo para poder ejecutarse.

Política 2

En este caso también tendremos en cuenta **el invariante menos disparado**. De esta forma se elige primero el invariante que se completó la menor cantidad de veces y luego dentro de este **la transición más próxima a finalizar el invariante**. En caso de no tener transiciones habilitadas, se prosigue con el siguiente invariante menos completado.

Política 3

Igual que la política 2, solo que esta vez se prioriza **la transición más lejana a finalizar el invariante.**

Tiempos de las transiciones temporales

Transición	Alfa [ms]	Beta [ms]
T4	20	1000
T5	20	1000
T6	30	1000
T7	30	1000
T8	15	1000
T10	100	1000
T11	100	1000
T12	100	1000

Consideramos los invariantes 1 y 2 como líneas de ensamblaje que hacen la misma tarea, es un proceso mucho más rápido en comparación a un proceso de diseño como representa el invariante 3.

Análisis de Resultados

Disparo de transiciones

Transición	Politica 1			Política 2			Politica 3		
	N° disparos	N° / 100[ms]	% del total	N° disparos	N° / 100[ms]	% del total	N° disparos	N° / 100[ms]	% del total
T1	500	0,29	11,11%	668	0,60	14,31%	670	0,74	14,32%
T2	250	0,14	5,56%	334	0,30	7,15%	336	0,37	7,18%
T3	250	0,14	5,56%	333	0,30	7,13%	334	0,37	7,14%
T4	250	0,14	5,56%	334	0,30	7,15%	335	0,37	7,16%
T5	250	0,14	5,56%	333	0,30	7,13%	334	0,37	7,14%
T6	250	0,14	5,56%	334	0,30	7,15%	334	0,37	7,14%
T7	250	0,14	5,56%	333	0,30	7,13%	333	0,37	7,12%
T8	500	0,29	11,11%	667	0,60	14,29%	667	0,74	14,26%
T9	500	0,29	11,11%	334	0,30	7,15%	335	0,37	7,16%
T10	500	0,29	11,11%	333	0,30	7,13%	334	0,37	7,14%
T11	500	0,29	11,11%	333	0,30	7,13%	334	0,37	7,14%
T12	500	0,29	11,11%	333	0,30	7,13%	333	0,37	7,12%
Total	4500	2,58	100,00%	4669	4,20	100,00%	4679	5,18	100,00%

Podemos ver que el objetivo de la política 1 de que la cantidad de disparos de las transiciones esté equilibrada no se logra muy bien. Esto tiene un motivo, al tener en cuenta sólo la cantidad de disparos de las transiciones, T1 se dispara la misma cantidad de veces que T9, pero T1 se encarga de dar comienzo a 2 ramas, por esto vemos como las transiciones relacionadas a estas ramas tienen justo la mitad de disparos. Por lo que entendemos, **es imposible disparar todas las transiciones por igual,** por este mismo motivo.

Esto también provoca que las transiciones relacionadas al invariante 3 tengan prioridad respecto a las demás, por el mismo motivo, su transición de inicio, T9, solo le pertenece a él.

La política 2 y 3 son bastante similares, la principal diferencia es que la 3 dispara a mayor velocidad, esto lo analizaremos posteriormente en el análisis temporal. Podemos ver también como el % de disparos sobre el total, salvo las transiciones compartidas T8 y T1 está equitativamente distribuido. La excepción de estas 2 transiciones es esperable, ya que es necesario disparar estas transiciones más veces para nivelar la carga del resto.

Disparo de Invariantes

Invariante	Politica 1		Politica 2		Politica 3	
	N° ejecuciones	% del total	N° ejecuciones	% del total	N° ejecuciones	% del total
1	250	25,00%	334	33,40%	334	33,40%
2	250	25,00%	333	33,30%	333	33,30%
3	500	50,00%	333	33,30%	333	33,30%
Total	1000	100,00%	1000	100,00%	1000	100,00%

En cuanto a la distribución de los invariantes obtenemos los resultados esperados, la primera política al no tener en consideración la cuenta de invariantes provoca un desbalance en el inv 3 por el mismo motivo que mencionamos anteriormente que los disparos de T9 son iguales a los de T1. Para las políticas 2 y 3 podemos ver un balanceo perfecto como era pensado.

	Disparo de invariantes completos
Política 1	1
Política 2	334
Política 3	0

Otro dato interesante es la cantidad de invariantes completos de la ejecución, llamamos invariante completo a la ejecución de un invariante de forma estricta, es decir toda la secuencia de transiciones sin ser interrumpida por otro invariante. Podemos ver que en el único caso en el que esto se cumple es en la política 2, esto ya nos está dando una idea de que parte de la ejecución se está haciendo de forma secuencial.

El fenómeno anterior sucede ya que la política 2 siempre intenta acercarse a terminar un invariante siempre que le es posible, esto provoca que si se requieren varias transiciones para terminar un invariante con menor cantidad de disparos las va a ir disparando una tras otra de forma secuencial hasta lograrlo.

Tiempo de ejecución

Para hacer un análisis del tiempo de ejecución de nuestras políticas primero vamos a buscar un valor de referencia sobre el cual comparar. Un buen punto de partida sería cuanto es el tiempo de ejecución teórico si se disparan los invariantes de forma secuencial uno tras otro.

$$t(inv1) = t(T1) + t(T2) + t(T4) + t(T6) + t(T8) = 65 \text{ [ms]}$$
$$t(inv2) = t(T1) + t(T3) + t(T5) + t(T7) + t(T8) = 65 \text{ [ms]}$$
$$t(inv3) = t(T9) + t(T10) + t(T11) + t(T12) = 300 \text{ [ms]}$$

Estamos realizando el análisis sobre el disparo de 1000 invariantes, si en nuestro supuesto decimos que se disparan de forma secuencial inv1 -> inv2 -> inv3 entonces el inv1 se dispara 334 veces mientras que inv2 e inv3 333. Entonces:

$$t = t(inv1)*334 + t(inv2)*333 + t(inv3)*333 = \mathbf{143255 \text{ [ms]}}$$

Ahora sí veamos cómo resultó nuestra implementación.

Política	Tiempo [ms]	Variación respecto a sec
Secuencial	143255	-
1	174477	21,79%
2	110512	-22,86%
3	90679	-36,70%

Algo destacable es que la política 1 resultó ser incluso peor que una política 100% secuencial, esto se debe a la mayor presencia del invariante 3, que además tiene un tiempo de ejecución mucho más lento que el resto, ralentizando todavía más el proceso.

Entre la política 2 y 3 tenemos a la 3 como la ganadora, esto se produce gracias al **mayor paralelismo** que esta causa. Al dar prioridad a iniciar invariantes o avanzar los más retrasados siempre hay una mayor cantidad de invariantes en ejecución a la vez, permitiendo que, por ejemplo, las transiciones temporizadas alcancen el umbral mientras se disparan transiciones más retrasadas o de otro invariante que no las afecte.

Tuvimos un adelanto de este resultado cuando vimos que la política 3 tenía una mayor tasa de disparos cada 100 [ms].

Conclusión

Después de analizar todos los resultados, podemos concluir que una política que favorezca el paralelismo, tenga en consideración las transiciones temporales y busque balancear la carga de los invariantes va a dar los mejores resultados, como es el caso de nuestra política 3.