

UNIVERSIDAD NACIONAL DE
CÓRDOBA
Facultad de Ciencias Exactas,
Físicas y Naturales



Programación Concurrente
Trabajo Práctico Final

Grupo: Game of Threads

Integrante:

- Patiño, Jonathan Armando

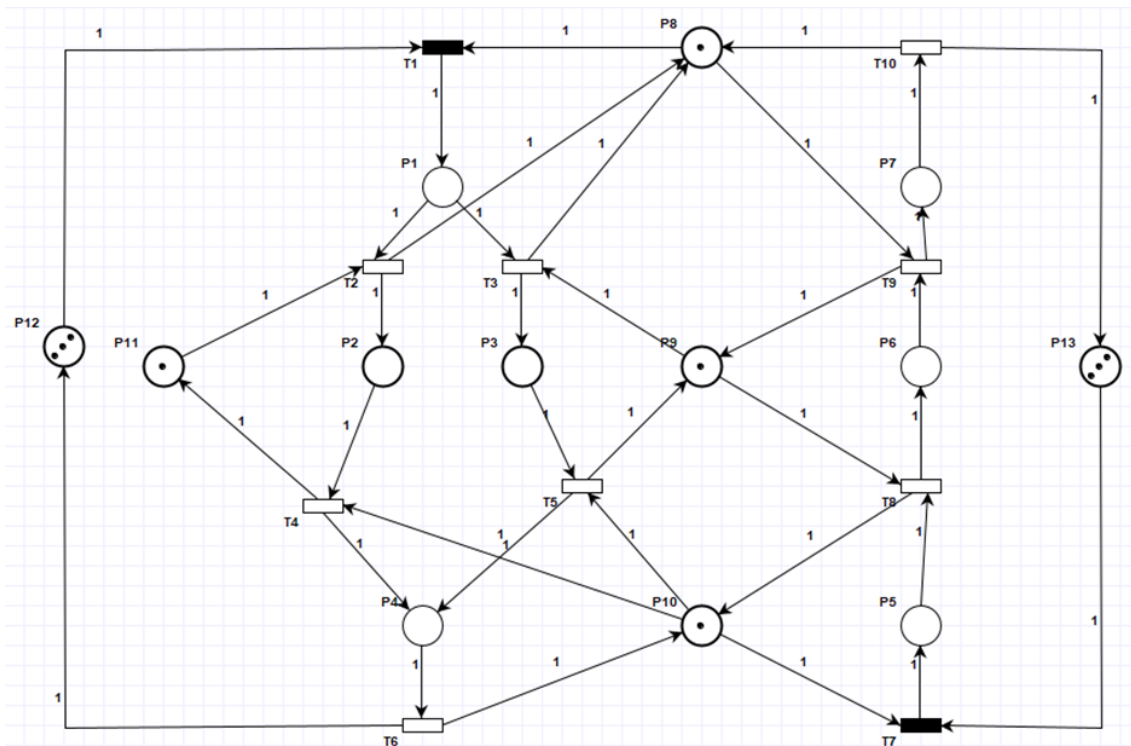
DNI: 35.636.684

I) Sistema modelado por la Red de Petri	2
Propiedades de la red original	2
Desbloqueo de la Red de Petri	4
II) Implementación en Java	6
III) Dinámica de ejecución	7
Mecanismos de control de la finalización del programa	7
Mecanismos de control de la concurrencia de los hilos	9
IV) Implementación de la política	10
V) Diagrama de clase	12
VI) Diagrama de secuencia	13
VII) Determinación de la cantidad de hilos necesarios	13
VIII) Análisis con mil Disparos	15
IX) Registro de resultados	16
X) Interpretación de los invariantes	17
XI) Verificación de los invariantes de plaza	18
XII) Expresiones regulares	19
XIII) Tabla de eventos	22
XIV) Tabla de estados	23
XV) Conclusión	24

I) Sistema modelado por la Red de Petri

Propiedades de la red original

Se partió de la Red de Petri del enunciado y se utilizó la herramienta Pipe para analizar sus propiedades.



Se ejecutó la clasificación de la red y el “State Space Analysis” que ofrece la herramienta, obteniendo los siguientes resultados:

- La red es acotada (bounded = true): esto significa que la cantidad de tokens en la red se mantiene constante cuando la red evoluciona. Tiene sentido, considerando que la red tiene 2 circuitos cerrados que comienzan y terminan en la misma plaza. Uno de ellos es el indicado por las transiciones T1, T2T4 ó T3T5, T6, que utiliza como buffer a la plaza P12, inicialmente con 3 tokens. El otro es el camino indicado por las transiciones T7, T8, T9 y T10, con P13 como buffer.

Que sean circuitos cerrados implica que todo lo que sale de P12 vuelve en igual cantidad. Lo mismo sucede con P13. El token que sale del buffer va avanzando en algún proceso y debe liberar el recurso que está usando para que pueda ingresar otro token. En ningún momento hay una transición que consuma o genere más de un token, y todo recurso compartido que se usa, se devuelve dentro del mismo circuito cerrado. Las

plazas en sí no están limitadas a una cantidad de tokens, pero la forma en que está diseñado el modelo ocasiona que nunca haya más de 3 tokens entre las plazas de tareas de cada circuito cerrado (P1, P2, P3, P4, P5, P6 y P7) y los buffers (P12 y P13), y que nunca haya más de un token en las plazas correspondientes a los recursos compartidos (P8, P9, P10 y P11). La red sería no acotada si, por ejemplo, se agregara una plaza de control que cuente la cantidad de veces que se disparó T7.

- La red no es segura (safe = false): una plaza de la red es k-limitada para un marcado inicial M0 si, para cualquier otra marca, el número de tokens en p es menor o igual a k. Una red es k-limitada para un marcado inicial M0 si cada plaza es k-limitada. Una RdP es segura si todas sus plazas, y por lo tanto la red en sí, son 1-limitadas. Esto no se cumple, porque desde el inicio hay 3 tokens en cada uno de los buffers.
- La red tiene deadlock (deadlock = true), por lo tanto, la red no está viva: como se ve en el modelo, las transiciones T3, T4, T5, T7, T8 y T9 utilizan recursos que son compartidos con otras transiciones y, a priori, no hay una relación de orden respecto a qué transición toma el token. Tampoco hay una liberación del recurso sin antes tomar el siguiente recurso compartido. Se terminan generando interbloqueos porque puede que haya dos transiciones que necesiten intercambiar recursos y que ninguna suelte el suyo hasta no obtener el siguiente. Por ejemplo, si se dispara T3, va a estar utilizando el recurso de P9, y para disparar T4 va a necesitar el token de P10. Si luego de T2 se disparó T7, va a tomar el token de P10 y no lo va a liberar hasta no tener acceso al token de P9, que ya lo tiene tomado T3. Es necesario así imponer restricciones para que se inhiban ciertas transiciones si ocurrieron otras previamente, como se verá a continuación.
- La red no es persistente: una red es persistente si todas sus transiciones cumplen con que una vez habilitada la transición, ésta sólo puede deshabilitarse mediante su disparo. Esto no se cumple, precisamente porque hay transiciones que comparten recursos. Luego, una transición que estaba sensibilizada puede perder su estado si otra transición utiliza primero el recurso en común.
- La red no es reversible: una RdP es reversible para una marca inicial M0 si M0 es un Home State, es decir, si M0 es una marca alcanzable desde cualquier otra marca. Si bien, la marca M0 de esta red pareciera ser un home state, esto no se cumple porque la red tiene deadlock, luego hay marcas en las que no va a haber ninguna transición sensibilizada y, por lo tanto, no se podrá volver al estado inicial.
- Conflictos: esta red presenta conflictos del tipo estructural y del tipo efectivo, porque hay transiciones que tienen un lugar de entrada común. No se consideran conflictos generales porque el grado de habilitación de las transiciones es siempre 1.

- Las transiciones T2 y T3: el conflicto es estructural, porque tienen a P1 en común, y también efectivo porque en ningún momento puede haber más de un token en P1 permitiendo que se disparen las dos transiciones a la vez (para que haya un token en P1 se debe disparar T1 que requiere un token de P8, el cual solo se repone al dispararse T2 o T3).
- Las transiciones T4, T5 y T7: las tres transiciones comparten el token de P10. Similar al caso anterior, el conflicto es tanto estructural como efectivo, ya que en ningún momento P10 puede tener más de un token.
- Las transiciones T3 y T8: comparten a P9, conflicto estructural y efectivo.
- Las transiciones T1 y T9: comparten a P8, conflicto estructural y efectivo.
- Exclusión mutua: todas las transiciones que presenten conflicto con otras, se deberán realizar en exclusión mutua, lo que se ve representado con los recursos compartidos que tienen un solo token y son consumidos por una transición a la vez.

Desbloqueo de la Red de Petri

Se analizaron los distintos casos en los que la red se bloqueaba para entender cómo evitar que dos o más transiciones se bloqueen mutuamente. En particular, se determinó que para que la red no se bloquee se debe cumplir que:

1. Caso 1 de interbloqueo: hay token en P2 y P3 y se dispara T7. T3 toma el recurso de P9 y no lo libera hasta que se dispara T5, para ello requiere del token de P10 que está en P5. Por su parte, T8 no puede disparar y liberar el recurso de P10 porque necesita del recurso de P9 que está en P3.
 - Solución: si se disparó T2 o T3, ya no se debería poder disparar T7 hasta que no se dispare T6, porque si no T7 consumiría el token de P10 y los tokens de P2 y P3 quedarían atrapados hasta que se dispare T8. Por la misma razón, si se disparó primero T7 ya no se debería poder disparar T3 hasta que no se dispare T9. Notar que no es necesario restringir la transición T2 si se disparó T7 porque T2 no utiliza recursos compartidos, como sí lo hace T3.
2. Caso 2 de interbloqueo: hay un token en P2 y se dispara T7 T8 T1 T7, o bien, T1 T7 T8. Para que el token de P6 pueda continuar necesita del token de P8 que, como se disparó T1, está en P1. T2 no se puede disparar porque ya hay un token en P2 que no puede avanzar porque T4 necesita el recurso de P10 que tomó T7. T3 tampoco puede disparar porque necesita el token de P9 que tomó T8.

- Solución: no dejar disparar T1 si previamente se disparó T8. Así el token de P6 no se queda atrapado.

Para implementar las soluciones, se consideraron varias alternativas pero se optó por la siguiente. Se puede comprobar que no se alteran los invariantes de la red original.

Utilizando arcos inhibidores: se agregaron 4 brazos inhibidores:

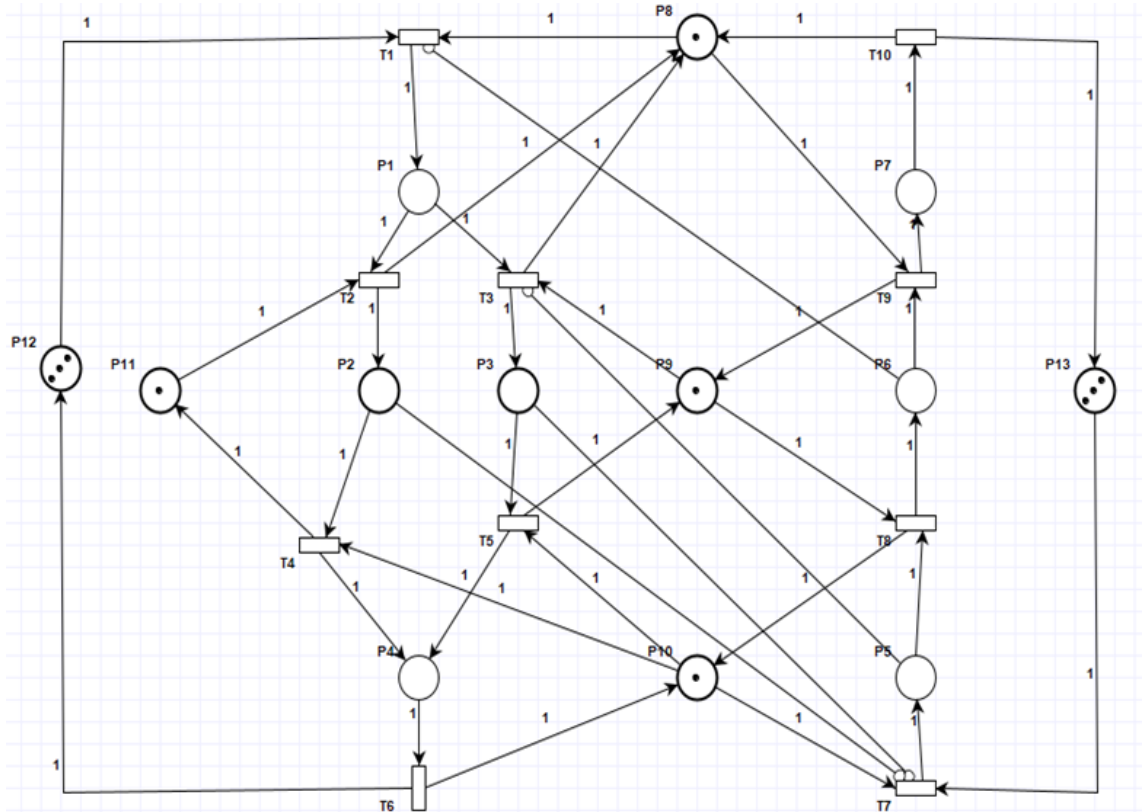


Figura 1: Red de Petri.

1. Entre P3 y T7 para que T7 no se dispare si previamente se disparó T3 y así el token de P3 no queda encerrado.
2. Entre P2 y T7 para que T7 no se dispare si previamente se disparó T2 y así el token de P2 no queda encerrado.
3. Entre P5 y T3 para que no se dispare T3 si previamente se disparó T7 y el token de P5 no quede encerrado.
4. Entre P6 y T1 para que no se dispare T1 si previamente se disparó T8 y así el token de P6 no quede encerrado.

II) Implementación en Java

Para la implantación en Eclipse se necesita una versión de la máquina virtual de Java, mayor o igual que 11.

Descripción de las clases

Clase Hilo

Esta clase recibe como parámetro una secuencia a disparar. En su método `run()`, llama al método `dispararTransicion()` de la clase `Monitor` y dispara secuencialmente las transiciones que le fueron asignadas dentro de un bucle `while`. Continúa haciéndolo hasta que desde el hilo del `main` se llama al método `setFin()` que pone en `false` la variable `continuar`, permitiendo que el hilo finalice la ejecución de su método `run()`.

Clase Matriz

Es una clase auxiliar que contiene todos los métodos necesarios para definir y operar con las matrices de la Red de Petri, como generar la matriz a partir de los datos de un archivo, escribir o leer un dato determinado, multiplicar o hacer un AND con otra matriz, etc.

Clase RDP

Contiene todos los métodos necesarios para implementar una Red de Petri a partir de archivos de texto con:

- La matriz de incidencia
- La matriz de entrada
- La matriz de inhibición
- El vector de marcado inicial
- Los vectores de intervalos de tiempo

Esta clase, además de los métodos para cargar los datos de las matrices en estructuras de datos adecuadas, también contiene métodos para poder describir el estado de la red en cualquier momento y la inicialización de la red de petri.

Clase SensibilizadaConTiempo

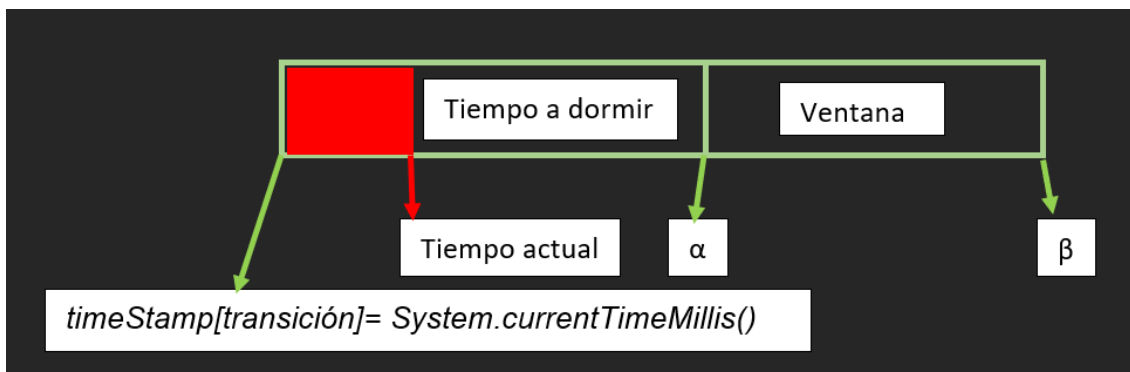
En esta clase se encuentra todo lo referido a una transición temporal, por ejemplo, el método `ActualizarTimeStamp()` contiene las siguientes condiciones para modificar el vector `timeStamp[]`, el cual cuenta con una marca de tiempo en donde la transición comenzó a estar sensibilizada pero inhibida por la falta de tiempo para, poder entrar en la ventana de disparo. Las siguientes condiciones tienen que darse para colocar, o no, una marca de tiempo en el vector:

- Si la transición se encontraba sensibilizada antes del disparo y sigue sensibilizada después de disparar, no se resetea el contador de la misma, salvo que sea la misma transición que pregunta sea la que cambió de des-sensibilizada a sensibilizada.

- Si la transición estaba sensibilizada antes de disparar, y después del disparo no se encuentra sensibilizada, el contador de la misma se pone en cero ($timeStamp[transición] = 0$).
- Si no estaba sensibilizada y después de disparar si lo está, se inicia la cuenta del tiempo, es decir, se coloca una marca ($timeStamp[transición] = System.currentTimeMillis()$).
- Si no estaba sensibilizada y sigue sin estarlo después del disparo, no se empieza la cuenta.

Cálculo para la ventana de tiempo:

Marca de tiempo + alfa - tiempo Actual = tiempo que debe dormir un hilo.



Clase Cola

Modela las colas donde van a esperar los hilos para acceder al monitor y contiene los métodos que agregan y sacan transiciones de la cola. Genera un arreglo de semáforos, con tantos semáforos como transiciones tenga la Red de Petri. Todos son inicializados con 0 permisos disponibles. Esto es porque sólo se quiere usar al semáforo para registrar el orden en el que los hilos van llegando a la cola, y así poder darles acceso al monitor cuando se habilite su transición.

El control de los *acquire* y *release* de los semáforos los tiene el monitor cuando llama a los métodos *ponerEnCola()* y *sacarDeCola()*, respectivamente.

Clase Política

Posee como método más relevante el método *cual(Matriz m)* que permite la elección entre dos o más transiciones, las cuales deben estar sensibilizadas y también deben estar en la cola de espera en el monitor. La elección se lleva a cabo de acuerdo a cuál transición se disparó menos en el transcurso del tiempo. La clase política también registra cada disparo exitoso de una transición.

Clase Monitor

Se encuentra lo necesario para controlar la sección crítica. A esta clase sólo accede el hilo que tomó el token del semáforo binario, el cual se encuentra en la clase

semaphore y puede hacer uso de los recursos que necesite para avanzar o esperar hasta que tenga los recursos disponibles.

Clase Main

Es la clase principal para un programa realizado en Eclipse. Todo programa debe tener una clase main. Esta clase está controlada por el proceso principal, y es la encargada de crear los hilos y, luego, finalizarlos.

Clase Log

Se registran la cantidad de veces que se dispararon las transiciones y la cantidad de veces que se completaron los invariantes. También se puede ver la evolución del marcado de red opcionalmente.

III) Dinámica de ejecución

Mecanismos de control de la finalización del programa

Los hilos comienzan en el proceso principal mediante el método *start()*, luego, el proceso principal, el cual mantiene conexión con los hilos, se duerme en el tiempo de corrida del programa establecido, luego despierta y llama al método *set_Fin()* de cada hilo, y también al método *interrupt()* el cual interrumpe al hilo ya que un hilo puede estar durmiendo y este método se encargará de despertarlo.

```
threads = new Thread[numeroHilos];
for(int i=0; i<numeroHilos;i++)threads[i] = new Thread(hilos[i], "" +i);

for(Thread T : threads)T.start();

try {
    Thread.sleep(tiempo_ejecucion);
} catch (InterruptedException e) {
    e.printStackTrace();
}

for(Hilo H:hilos)H.set_Fin();
monitor.vaciarcolas();
for(Thread t : threads)t.interrupt();
```

También se encuentra un método dentro de la clase monitor que vaciará la cola donde duermen los hilos y luego los saca del monitor.

```
public void vaciarcolas() {
    try {
        mutex.acquire();
        fin = true;
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    for (int i = 0; i < red.get_numero_Transiciones(); i++) cola.sacar_de_Cola(i);
    mutex.release();
}
```

En el método “*vaciarcolas()*” se cambia la variable de condición “*fin*” de su valor false a true, lo que hará que los hilos no tomen el “*acquire*” del mutex dentro del monitor, de esta manera finaliza la tarea de los hilos.

```
public void dispararTransicion(int T_Disparar) {
    try {
        if(fin == true) return;
        mutex.acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

La clase hilo contiene lo siguiente:

```
public class Hilo implements Runnable {
    private Monitor monitor;
    private int[] secuencia;
    private static boolean continuar;

    public Hilo(Monitor monitor, int[] secuencia) {
        this.monitor = monitor;
        this.secuencia = secuencia;
        continuar = true;
    }

    @Override
    public void run() {
        while(continuar){
            for(Integer i:secuencia) {monitor.dispararTransicion(i - 1);}
        }
        // Fin del hilo
        public void set_Fin() {
            continuar = false;
        }
    }
}
```

Mecanismos de control de la concurrencia de los hilos

Para el control de la concurrencia se utiliza un semáforo binario, es decir, de dos estados (libre u ocupado). Luego, con los métodos *acquire()* y *release()* propios de la clase *semaphore* se accede a la sección crítica o se libera de la misma.

```
public void dispararTransicion(int T_Disparar) {  
    try {  
        if(fin == true) return;  
        mutex.acquire();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
  
    k = red.Disparar(T_Disparar); // Hilo "+ Thread.currentThread().getName()  
  
    if (k) {  
        registrar_log(T_Disparar);  
        politica.registrarDisparo(T_Disparar);  
        m = calcularVsAndVc();  
        if (m.esNula()) {  
            mutex.release();  
            return;  
        } else {  
            nTransicion = politica.cual(m);  
            cola.sacar_de_Cola(nTransicion);  
            mutex.release();  
            return;  
        }  
    }  
    mutex.release();  
    cola.poner_EnCola(T_Disparar);  
    dispararTransicion(T_Disparar);  
}
```

Con el método `acquire()` se adquiere un permiso para poder intentar disparar una transición, si el disparo fue correcto entonces el hilo tiene dos opciones: pregunta si alguien está en la cola y sensibilizado, si no lo está, entonces ejecuta el método `release()`, es decir, libera el semáforo y sale del monitor; si no, saca un hilo de la cola mediante el uso de la política y luego libera el recurso, el hilo que sale de la cola debe pelear por el ingreso al monitor de nuevo y tiene la misma posibilidad que los demás hilos que están esperando en adquirir el semáforo, repitiendo el ciclo.

IV) Implementación de la política

La política es una parte crucial de la simulación ya que permite equilibrar el uso de los recursos compartidos por los hilos. Para implementar una política equilibrada se tuvieron en cuenta los disparos que realizó cada transición y de ese modo se obtuvo la cantidad de veces que se disparan los invariantes de transición. En la clase monitor se puede observar el método `registrarDisparo(T_Disparar)` que pertenece a la clase política, el cual se observa a continuación:



```

public void registrarDisparo(int nTransicion) {

    if (nTransicion == 0) { // T1
        if (inv1 == inv2)
            Transiciones[0].setInvariante(aleatorio());
        else if (inv1 > inv2)
            Transiciones[0].setInvariante(1);
        else
            Transiciones[0].setInvariante(0);
    }
    if (nTransicion == 3) { // T4
        ultimaTrancisionDisparada = 3;
    }
    if (nTransicion == 4) { // T5
        ultimaTrancisionDisparada = 4;
    }
    if (nTransicion == 9) { // T10
        inv3 = inv3 + 1;
        actualizar_invariante(2);
    }
    if (nTransicion == 5) { // T6
        if (ultimaTrancisionDisparada == 3) { // T4 --> T6
            inv1 = inv1 + 1;
            actualizar_invariante(0);
        } else if (ultimaTrancisionDisparada == 4) { // T5 --> T6
            inv2 = inv2 + 1;
            actualizar_invariante(1);
        }
    }
    disparos.set(nTransicion, (disparos.get(nTransicion) + 1));
}

```

El registro de los disparos es de gran utilidad ya que permitirá tomar una decisión respecto a algún conflicto en la red. Esta red posee un conflicto estructural descriptos anteriormente. Dos eventos e_1 y e_2 se encuentran en conflicto cuando pueden ocurrir individualmente, pero no juntos.

En el registro de disparos se puede observar que existe una variable llamada “*Transiciones*” la cual es un vector de tipo “*Info*” que contiene información de la cantidad de veces que se disparó una transición, la cantidad de veces que se completó el invariante al cual pertenece esa transición luego, con esta información, cuando llegue un conflicto, por ejemplo estructural, como en el caso de que se haya disparado T1 y luego se deba elegir entre T2 o T3, y se usa la información que contienen las transiciones para saber cuál de la dos tiene menor veces cumplido su invariante y esa será la transición que se debe disparar. En caso de que sean iguales se procederá de forma aleatoria, y en caso que el conflicto sea entre dos transiciones del mismo invariante se procede a disparar la más próxima a completar el mismo, es decir, por ejemplo si el conflicto está en decidir entre T6 y T1, se priorizará T6. Cabe recordar que la decisión acerca de a qué invariante pertenece T1 está dada en el método *registrarDisparo()* en donde se puede observar que si son iguales la elección será de manera aleatoria. En cambio, T6 pertenece a un invariante dependiendo cuál transición se disparó primero, si T4 o T5. Si disparo T4, entonces T6 va a pertenecer al invariante 1; si disparo T5, pertenecerá al invariante 2. Los demás conflictos efectivos se resuelven de la misma forma.

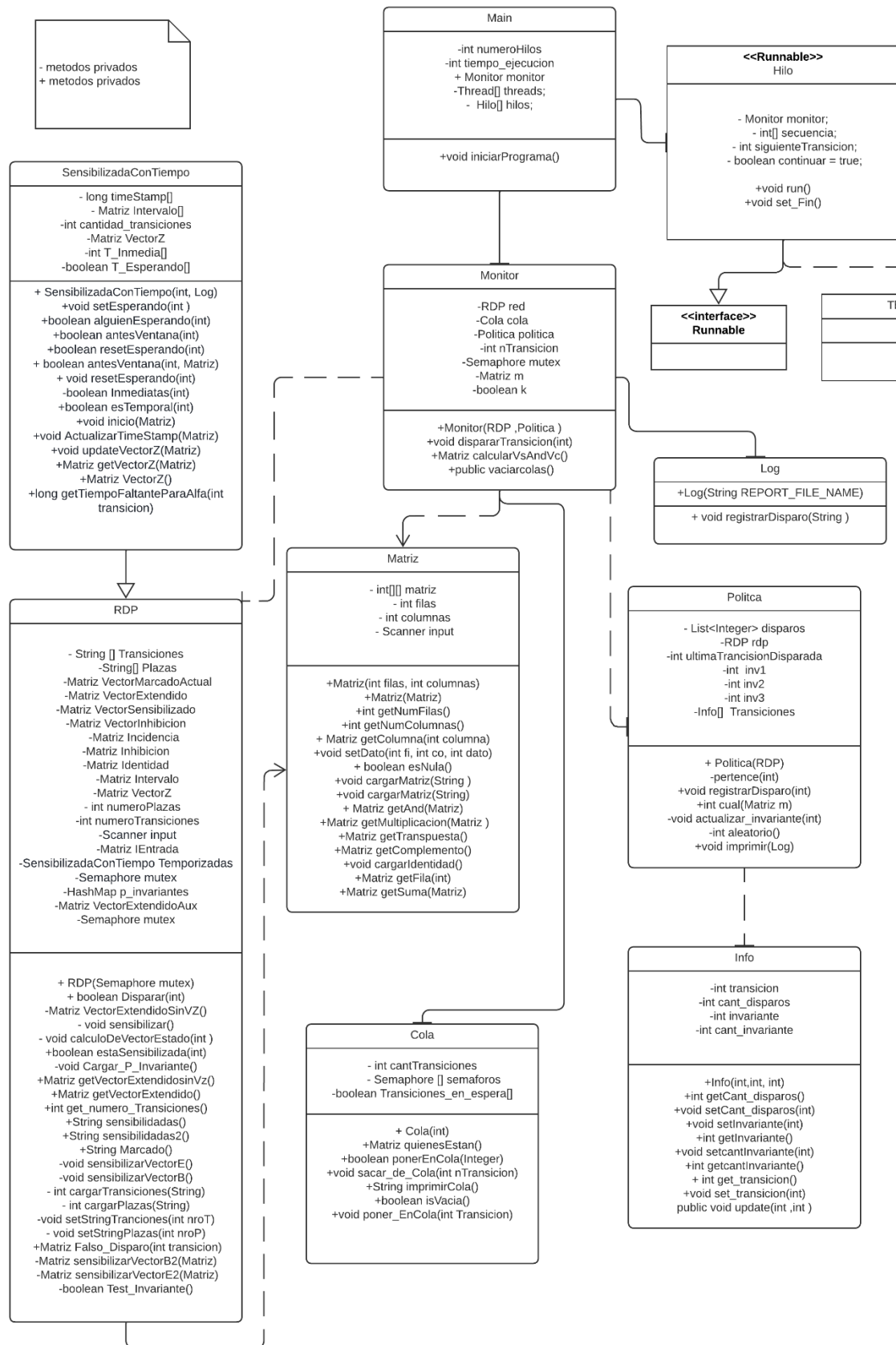
La secuencia a seguir en elegir cuál transición disparar es la siguiente:

- La cantidad de transiciones habilitadas y transiciones que están en la cola en mayor a uno, es decir el vector m tiene dos o más unos. Ej: $m = 0\ 1\ 1\ 0\ 0\ 0\dots$
- Se considera la primera transición recorriendo el vector m .

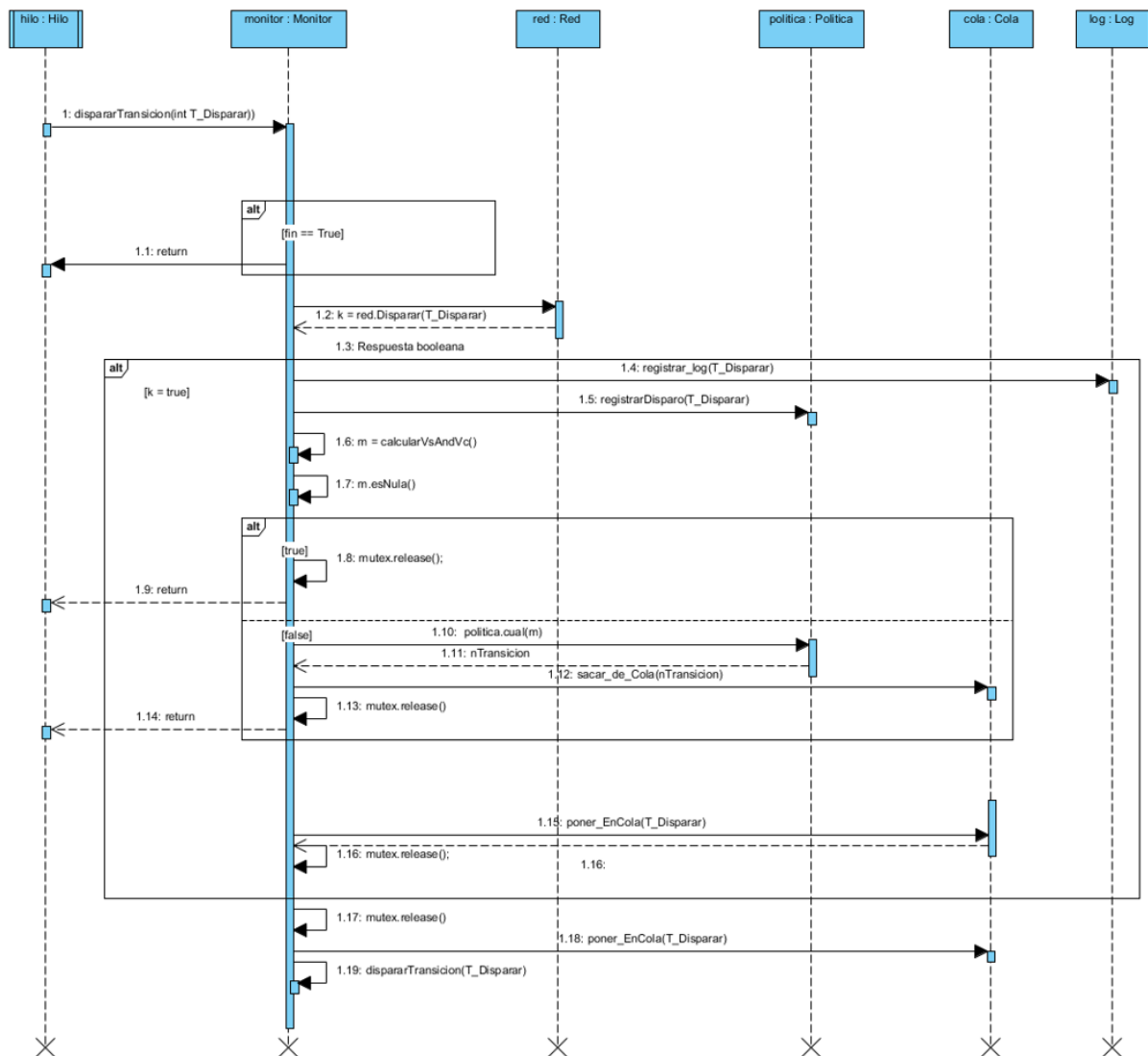
- Se compara la primera Transición encontrada con la próxima disponible en m y se toma una decisión sobre con cuál quedarse antes de seguir buscando en el vector. La decisión se toma en base a:

- La cantidad de veces que se disparó el invariante de esta transición es mayor que la cantidad de la transición actual.
- Si pertenece al mismo invariante se almacena la transición de mayor tamaño, por considerar prioridad que se complete el invariante.
- Si la cantidad de veces que se disparó el invariante de la transición es igual a la cantidad de la transición actual, pero pertenecen a distintos invariantes entonces se elige de manera aleatoria.

V) Diagrama de clase



VI) Diagrama de secuencia

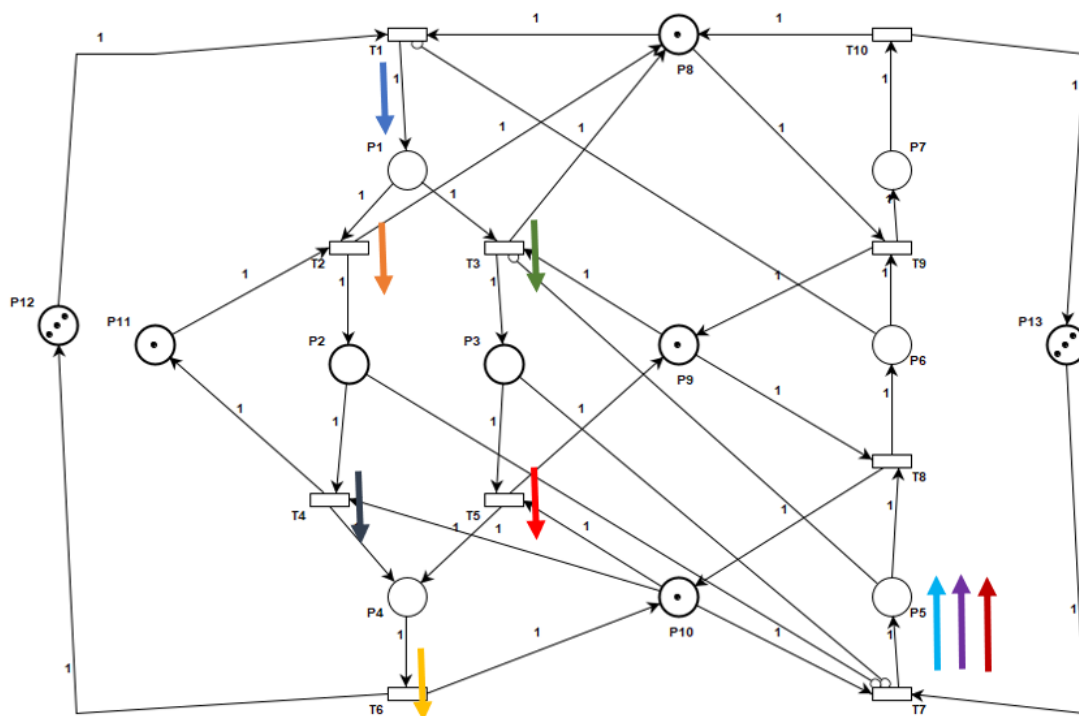


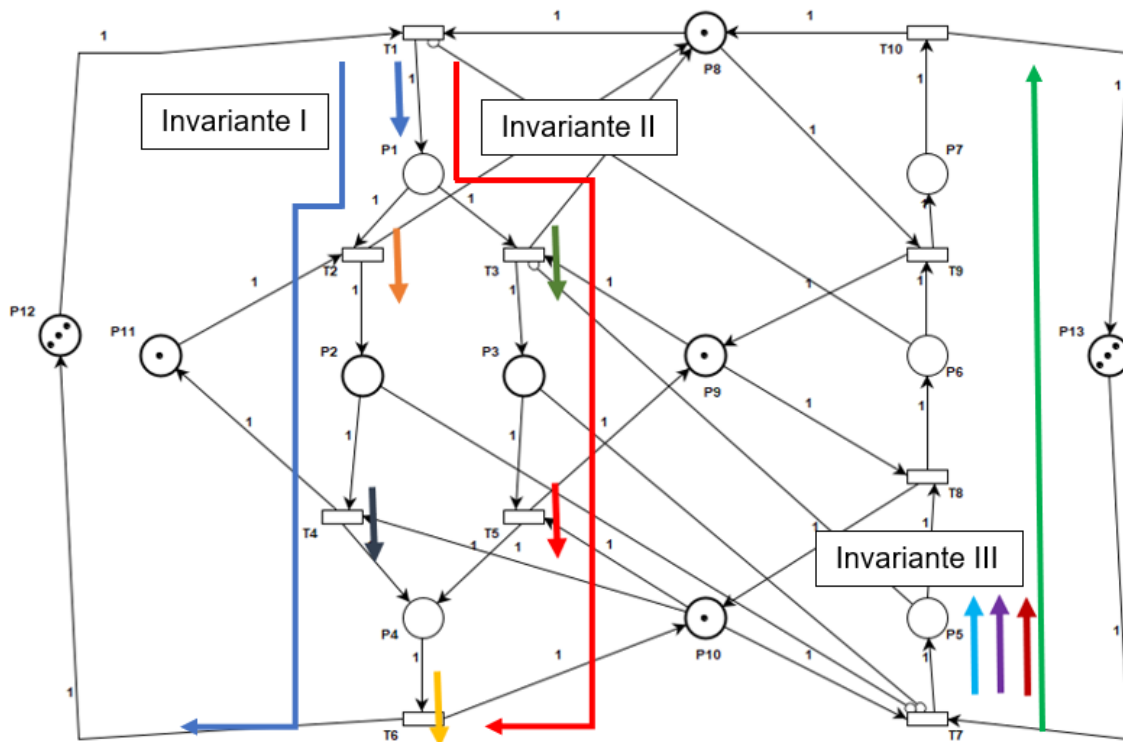
VII) Determinación de la cantidad de hilos necesarios

Los invariantes de la red representan las tareas relacionadas, por lo tanto, mientras no haya conflictos en el camino de invariante sigue siendo el mismo.

El análisis para la determinación de la cantidad de hilos se llevó a cabo en base a los invariantes y también a los resultados obtenidos a medida que se realizaron simulaciones. Por ejemplo, una secuencia de tareas realizada por hilos consecutivamente (invariante), es el caso del invariante I de la siguiente figura, donde se puede observar que el invariante es realizado por 4 hilos (las flechas de color representan un hilo). Esto es así, ya que los hilos se encargan de disparar las

transiciones T1, T2, T4 y T6. En una primera consideración se utilizaron 3 hilos para este invariante, siendo el hilo que dispara T2 el mismo que disparaba T4; luego, se observó que agregando un hilo más para el disparo de T4 la red alcanzaba un equilibrio entre los invariantes mucho más rápido que sin ese disparo, entonces se optó por utilizar un hilo más para T4. Lo mismo sucede para el invariante II el cual es encargado de disparar T1, T3, T5 y T6, en este caso la transición a la cual se le agrega un hilo extra es T5 obteniéndose un balance en menos tiempo de los invariantes y de esta manera se cumple con el requerimiento. En el caso del invariante III se considera la cantidad de estados posibles al mismo tiempo, es decir, la cantidad de plazas que pueden tener un token al mismo tiempo y de esta forma aumentar el paralelismo de la red. Como se puede observar, existen 3 plazas que pueden tener un token al mismo tiempo P5, P6 y P7, entonces se utilizan 3 hilos para este invariante. En conclusión, el motivo por el cual se utilizaron 9 hilos es porque de esta manera se obtuvo un mayor paralelismo y un menor tiempo en el balanceo de la red.





VIII) Análisis con mil Disparos

El programa devuelve dos archivos luego de la ejecución: uno llamado “*Reporte*”, el cual contiene la cantidad de veces que se dispararon las transiciones y la cantidad de veces que se cumplieron los invariantes; y otro, llamado “*log*”, el cual contiene la evolución de los eventos de la red y el vector sensibilizado extendido.

El tiempo de ejecución para obtener 1000 disparos de cada invariante es de aproximadamente 200 segundos.

```

IZ.txt  Reporte.txt x Politica.java
1Tiempo de ejecucion : 200seg.
2=====
3Invariante 1: 1063 veces [T1 T2 T4 T6] Tiempo:154
4Invariante 2: 1063 veces [T1 T3 T5 T6] Tiempo:58
5Invariante 3: 1056 veces [T7 T8 T9 T10] Tiempo:53
6=====
7Transicion: 1 disparos: 2128
8Transicion: 2 disparos: 1064
9Transicion: 3 disparos: 1063
10Transicion: 4 disparos: 1064
11Transicion: 5 disparos: 1063
12Transicion: 6 disparos: 2126
13Transicion: 7 disparos: 1056
14Transicion: 8 disparos: 1056
15Transicion: 9 disparos: 1056
16Transicion: 10 disparos: 1056
17
  
```

Los tiempos elegidos para cada transición son los siguientes:

	IZ.txt	Reporte.txt	Politica.java								
α	1	15	95	15	20	10	15	20	10	10	10
β	2	400	400	400	400	400	400	400	400	400	400
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	

Los valores de los tiempos fueron colocados de tal manera que se equilibraran los tres invariantes, la experiencia con la red indica que no es la única combinación de tiempos para lograr el equilibrio en los 1000 disparos, pero sí una de las posibilidades que se aproximó lo suficiente. El alfa o tiempo inferior es el tiempo en el cual la transición, una vez que fue sensibilizada, demorará en estar lista para disparar, luego se eligió un tiempo de beta mucho mayor ya que con un tiempo menor a 400 puede ocurrir inanición de la red ya que se puede estar ejecutando un invariante y dejar sin recursos a los otros.

IX) Registro de resultados

Como se dijo antes, los resultados de una ejecución de la simulación se almacenan en un archivo de formato .txt llamado log el cual contiene la evolución de la red, es decir los disparos de las transiciones. Luego, el archivo es analizado mediante expresiones regulares y utilizando Python, para obtener información que indica si la red cumplió con los invariantes de transición.

```
log.txt
1 T1T7T8T7T2T9T0T8T9T4T0T6T1T7T8T7T2T9T0T8T9T4T0T6T1T7T8T7T2T9T0T8T9T4T0T6T1T3T5T1T3T1T2T6T1T5T6
```

Existen otros archivos, reporte.txt y consola.txt, en el primero se obtiene la cantidad de veces que se dispararon las transiciones y los invariantes y el tiempo de ejecución; el segundo se utiliza para obtener la salida de la evolución de la red.

```
reporte.txt
1 Tiempo de ejecucion : 200seg.
2 =====
3 Invariante 1: 1060 veces [T1 T2 T4 T6]
4 Invariante 2: 1058 veces [T1 T3 T5 T6]
5 Invariante 3: 1058 veces [T7 T8 T9 T10]
6 =====
7 Transicion: 1 disparos: 2119
8 Transicion: 2 disparos: 1060
9 Transicion: 3 disparos: 1058
10 Transicion: 4 disparos: 1060
11 Transicion: 5 disparos: 1058
12 Transicion: 6 disparos: 2118
13 Transicion: 7 disparos: 1059
14 Transicion: 8 disparos: 1059
15 Transicion: 9 disparos: 1058
16 Transicion: 10 disparos: 1058
17
```

X) Interpretación de los invariantes

A partir de una marca inicial, el marcado de una RdP puede evolucionar mediante el disparo de transiciones (si no hay deadlock, dicho número es ilimitado). Sin embargo, no se puede alcanzar cualquier marca, y todas las marcas alcanzables tienen algunas propiedades en común. Se dice que una propiedad que no varía cuando se activan las transiciones, es invariante.

Si la matriz de incidencia por el vector de disparo da cero, entonces el vector de disparo se llama invariante de transición.

Los invariantes de plaza indican que el número de tokens en todas las marcas alcanzables satisface algún invariante lineal.

Salida del Pipe:

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.

P-Invariants

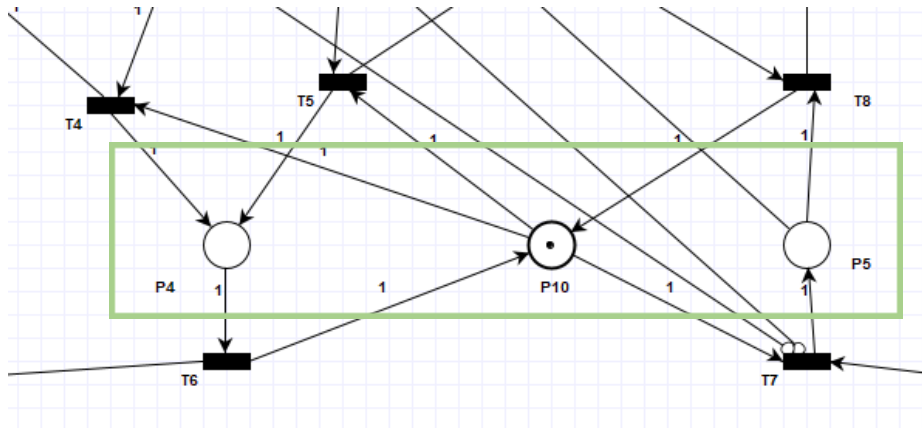
P1	P10	P11	P12	P13	P2	P3	P4	P5	P6	P7	P8	P9
0	1	0	0	0	0	0	1	1	0	0	0	0
0	0	1	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	1	1	0	0	0	0	0
0	0	0	0	1	0	0	0	1	1	1	0	0
1	0	0	0	0	0	0	0	0	0	1	1	0
0	0	0	0	0	0	1	0	0	1	0	0	1

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

P-Invariant equations

$$\begin{aligned}
 M(P10) + M(P4) + M(P5) &= 1 \\
 M(P11) + M(P2) &= 1 \\
 M(P1) + M(P12) + M(P2) + M(P3) + M(P4) &= 3 \\
 M(P13) + M(P5) + M(P6) + M(P7) &= 3 \\
 M(P1) + M(P7) + M(P8) &= 1 \\
 M(P3) + M(P6) + M(P9) &= 1
 \end{aligned}$$

La primera matriz define los invariantes de transición T1, T2, T4, y T6 en la primera fila; T1, T3, T5 y T6 en la segunda fila; y el tercer invariante: T7, T8, T9 y T10 en la tercera fila. En el caso de los invariantes de plaza aparte de la matriz, devuelve las ecuaciones, las cuales indican la cantidad de tokens (recursos) que están disponibles para un estado en particular. Por ejemplo, en la siguiente figura se observa el token que comparten P10, P4 y P5, que forma a un invariante de transición:



XI) Verificación de los invariantes de plaza

La verificación de los invariantes de plaza se realiza con cada disparo de una transición. Los invariantes de plaza no pueden cambiar a lo largo de la red, es por eso que se verifica si se cumplen en cada disparo. Para este caso se cargan las ecuaciones extraídas del Pipe en un HashMap (`p_invariantes`) -como se ve en la figura de los `p_invariantes` que salen del Pipe- y luego se las analiza de la siguiente manera:

```

public boolean Test_Invariante() {
    int Suma_Tokens_Plaza = 0;
    String valor = "";
    String[] corte = null;

    for (String plazas : p_invariantes.keySet()) {
        valor = p_invariantes.get(plazas);
        corte = plazas.split(" ");
        for (String elemt : corte) {
            int token = Integer.parseInt(elemt);
            Suma_Tokens_Plaza += VectorMarcadoActual.getDato((token - 1), 0);
        }
        if (Suma_Tokens_Plaza != Integer.parseInt(valor))
            return false;
        Suma_Tokens_Plaza = 0;
    }
    return true;
}

```

A modo de ejemplo si se modifica el archivo `P_invariantes.txt` de la siguiente manera:

```

*P_Invariantes.txt
1 M(P10) + M(P4) + M(P5) = 2 M(P11) + M(P2) = 1 M(P1) + M(P12) + M(P2) + M(P3) + M(P4) = 3 M(P13)

```

Se obtiene el siguiente error:

```
Console x
Main (19) [Java Application] C:\Users\Administrador\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_15.0.2.v20210201-0955\jre\bin\jav
=====
TP final
=====
Exception in thread "6" java.lang.RuntimeException: NO SE CUMPLE EL INVARIANTE DE PLAZA
    at codigo.RDP.Disparar(RDP.java:114)
    at codigo.RDP.Disparar(RDP.java:148)
    at codigo.Monitor.dispararTransicion(Monitor.java:42)
    at hilos.Hilo.run(Hilo.java:20)
    at java.base/java.lang.Thread.run(Thread.java:832)
```

El cual detiene la ejecución, y también se puede observar el mismo resultado en el archivo consola.txt.

```
consola.txt x
1 * INICIO
2 NO SE CUMPLE EL INVARIANTE DE PLAZA
3
```

XII) Expresiones regulares

El desarrollo de verificación de la expresión regular resultante de los disparos se realiza mediante Python, utilizando la librería **re**, y formando los caminos posibles mediante la página www.debuggex.com.

Un cambio que se debió hacer para poder llegar a la expresión regular necesaria fue cambiar el nombre de una transición de la siguiente manera:

```
if((T_Disparar+1) == 10)
    log.registrarDisparo("T"+0,0);
else
    log.registrarDisparo("T"+(T_Disparar+1),0);
```

Ya que para T10 la expresión regular considera T1 a T10, entonces se procedió a considerar T0 como T10. Dicha modificación no afecta el funcionamiento del programa.

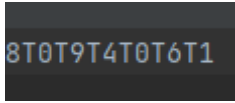


Ejecución del código:

21

La salida indica que sobra T1, lo cual es correcto ya que para este caso, cuando se finalizó la ejecución sólo se había disparado T1.

En el log se puede observar que finaliza T1:

A screenshot of a log output showing the sequence of thread execution: 8T0T9T4T0T6T1. The text is displayed in a monospaced font on a dark background.

8T0T9T4T0T6T1

XIII) Tabla de eventos

<u>Tabla de eventos</u>	
Transiciones	Eventos
T1	Ingreso a la primera actividad del invariante 1, o invariante 2
T2	Token ingresando a la segunda actividad del invariante 1
T3	Token ingresando a la segunda actividad del invariante 2
T4	Token saliendo de la segunda actividad del invariante 1
T5	Token saliendo de la segunda actividad del invariante 2
T6	Token saliendo del invariante 1 o del invariante 2
T7	Token ingresando a la primera actividad del invariante 3
T8	Token ingresando a la segunda actividad del invariante 3
T9	Token ingresando a la tercera actividad del invariante 3
T10	Token saliendo del invariante 3

XIV) Tabla de estados

<u>Tabla de estados o actividades</u>	
Plazas	Estados
P1	Primera actividad antes de proseguir por el invariante de transición 1, o el invariante de transición 2
P2	Segunda actividad realizándose en el invariante de transición 1
P3	Segunda actividad realizándose en el invariante de transición 2
P4	Tercera actividad realizándose, ya sea del invariante de transición 1, o del 2
P5	Primera actividad del invariante 3
P6	Segunda actividad del invariante 3
P7	Tercera actividad del invariante 3
P8	Token disponible para el avance hacia el invariante 1, o el invariante 2
P9	Token disponible para el avance hacia la segunda actividad del invariante de transición 2, o la segunda actividad del invariante de transición 3

P10	Token disponible para el avance hacia la tercera actividad del invariante de transición 1, o 2, o la primera actividad del invariante de transición 3
P11	Token disponible para el avance del invariante 2 hacia la segunda actividad
P12	Tokens disponibles para el avance por el invariante 1, o el invariante 2 hacia la primera actividad
P13	Tokens disponibles para el avance por el invariante 3 hacia la primera actividad

XV) Conclusión

Se observó que el uso de un monitor y una Red de Petri permite controlar la exclusión mutua, de manera tal, que todo se encuentre en un solo lugar y poder observar de manera ordenada la gestión de los recursos. Tener el manejo de la sección crítica (algo que anteriormente se realizaba en varias partes en el código mediante el uso de semáforos, lock, synchronized, etcétera), ahora se encuentra en el monitor. La red permite obtener una expresión matemática de la simulación, en base a matrices y vectores.

Los desafíos encontrados fueron:

- hallar la mejor red, es decir, la que paraleliza la mayor cantidad de procesos posibles. Para lograr esto se las comparó con otras redes y se analizaron los estados de las misma y se encontró una red en la cual se lograba mayor cantidad de actividades al mismo tiempo
- elegir la cantidad de hilos, los cuales, en algunos casos, fueron 7 ó 9. Esto no resultó fácil ya que con 7 hilos se completaron los requerimientos de la simulación, pero con 9 hilos se logra un balance de los invariantes mucho más rápido
- el tiempo en el cual debe suceder un evento (disparar la transición), manteniendo la política, estos tiempos tratan la mayor equidad entre los invariantes. En primera instancia se colocaron tiempos iguales en los tres invariantes y luego se fueron incrementando de a uno, e interpretando los



resultados para luego comprender qué invariante es el que se lleva la mayor carga de la red (desbalanceo) y, en este caso, resultó ser el invariante 1 al cual se le colocó mayor tiempo y, de esta forma, los otros dos invariantes logran equilibrarse.

GitHub: <https://github.com/Jonathan684/Programacion-Concurrente>