Universidad Nacional de Córdoba
Facultad de Ciencias Exactas Físicas y Naturales

INFORME TRABAJO FINAL

“Programación concurrente”

Carrera: Ingeniería en Computación

Profesor:

- Ing. Ventre, Luis
- Ing. Ludemann, Mauricio

Integrantes:

- Pablo Ariel Cazón
- Matias Exequiel Garcia

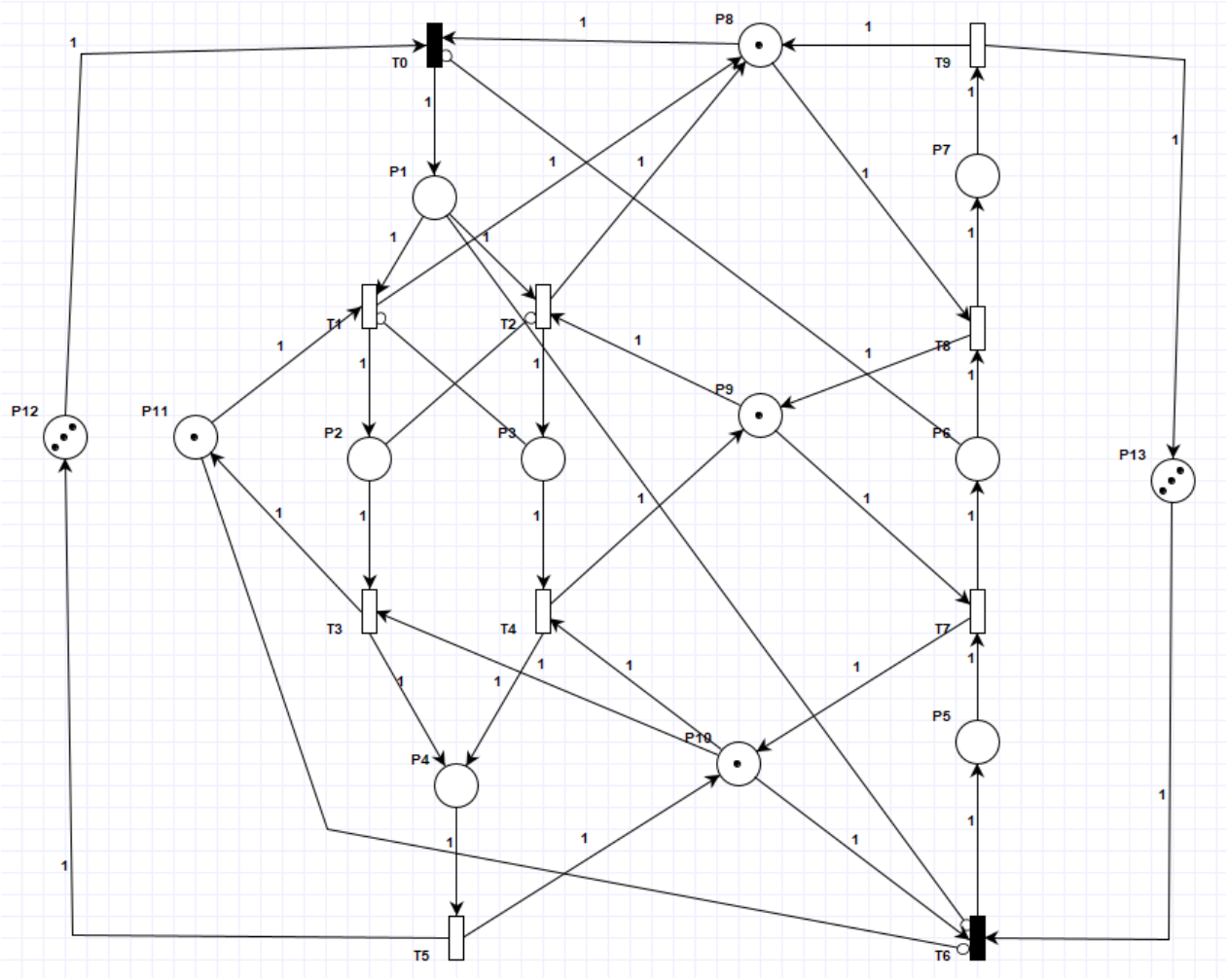
Fecha de entrega: 27 de febrero del 2023

Introducción

Este trabajo práctico consiste en la simulación de una red de Petri utilizando los mecanismos de concurrencia estudiados en el cursado de la materia. La red que vamos a simular es una red de Petri con el modelado de un sistema de producción (industrial, informático, cyber physical systems, etc), que anteriormente poseía deadlocks y que logramos desbloquear sin modificar sus invariantes de transición, y conservando el mayor paralelismo posible en ella.

Desarrollo

Red de petri con bloqueo



Observamos que la red de Petri para este caso va a tener 10 transiciones las cuales T1 y T7 van a ser inmediatas y el resto van a ser temporales. También consta con 13 plazas las cuales P8, P9, P10, P11 van a representar recursos compartidos del sistema, y las plazas P12, P13 van a corresponder a buffers del sistema.
Analizamos las propiedades y la clasificación de la red utilizando el software Pipe y nos arrojó las siguientes conclusiones:

Bounded	true
Safe	false
Deadlock	true

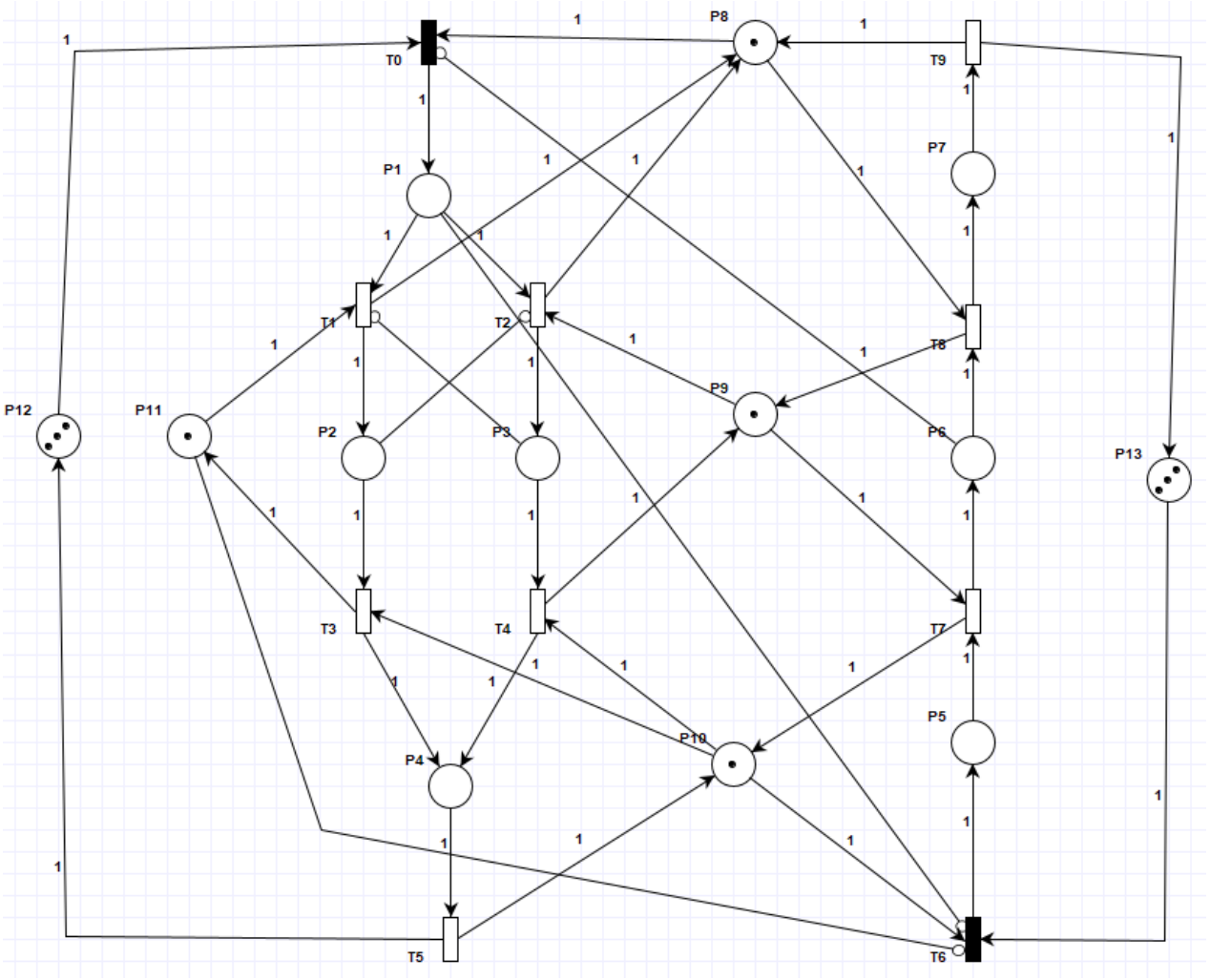
Nos interesa una red de Petri sin deadlock y acotada (Bounded), ya que tener bloqueos o plazas con recursos que tienden a infinito provocan errores en el monitor, si la red no está acotada el uso de hilos para este caso no sería fijo, debería ir incrementando en el transcurso de la ejecución. Y si la red tiene deadlocks se bloquearía la ejecución y no podríamos disparar ninguna transición.

Acotada: Vemos que la red está acotada por las plazas P12 y P13, donde se obtienen el máximo de tokens que puede alcanzar una plaza en la red, que es de 3 tokens.

Segura: Vemos que la red de Petri no es segura ya que contiene más de un token en las plazas P12 y P13. Vemos que la definición de seguridad en una red de Petri es que en todos sus estados no contenga más de un token.

DeadLock: Analizamos la red dada y vimos que presentaba un deadlock en distintos estados. Analizamos cuál era el estado de la red en el momento que se bloquea

Red de petri sin bloqueo



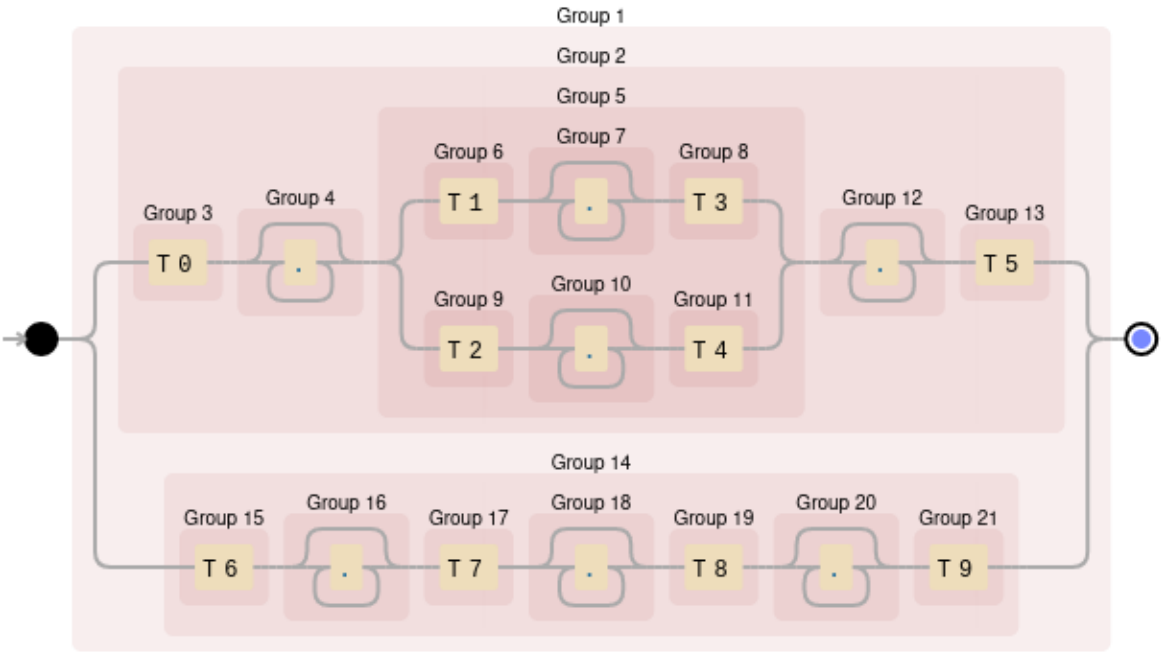
Después de varios intentos para desbloquear la red de Petri, utilizamos unos arcos inhibidores ya que no encontramos forma de no modificar los invariantes de transición solamente agregando plazas sin tokens.

Análisis de Invariantes

Esta red va a constar con 3 invariantes de transición:

Invariante 1: [T0, T1, T3, T5]
Invariante 2: [T0, T2, T4, T5]
Invariante 3: [T6, T7, T8, T9]

Analizando la expresión regular se obtiene:



Expresión regular:
(((T0)(.*?)((T1)(.*?)(T3)|(T2)(.*?)(T4))(.*?)(T5))|((T6)(.*?)(T7)(.*?)(T8)(.*?)(T9)))

Invariantes de plaza:

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

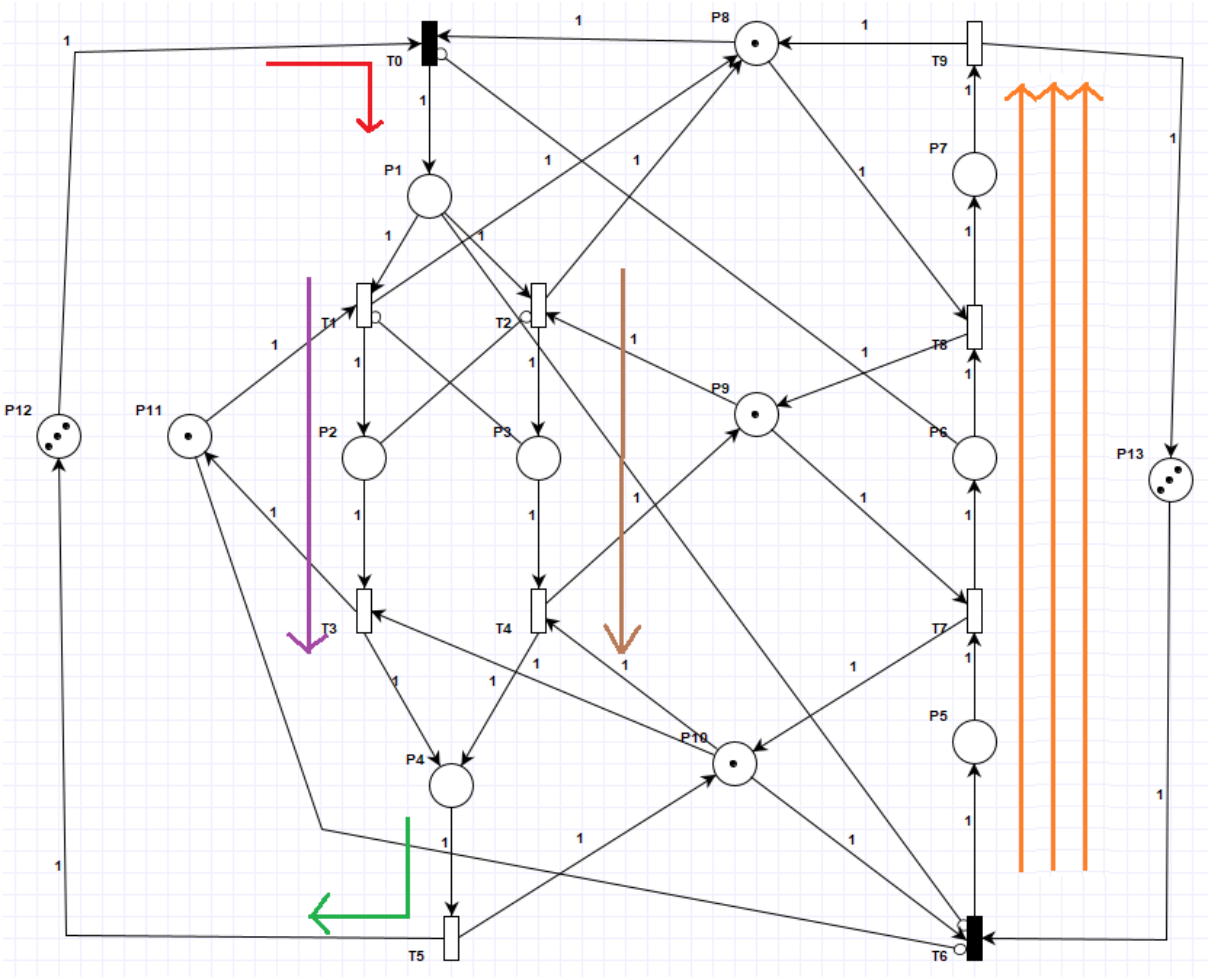
Propiedades de la red

Bounded	true
Safe	false
Deadlock	false

La red sigue acotada como nos interesa, pero ya no presenta los deadlocks que bloqueaban la red. Las otras propiedades que poseía anteriormente no cambiaron.

Criterios adoptados para el desarrollo del trabajo

Criterio de Hilos a utilizar según consigna del tp:



En la ejecución del programa establecimos que hayan 8 hilos:

- El hilo main que creará y lanzará el resto de los hilos.
- Un hilo para la transición T0
- Un hilo para las transiciones T1 y T3
- Un hilo para las transiciones T2 y T4
- Un hilo para la transición T5
- Tres hilos para las transiciones T6, T7, T8, T9

Evolución del Desarrollo

Una vez que conseguimos tener una red desbloqueada, se nos pidió hacer una comparación con otra red para ver cual presentaba mayor paralelismo.

Realizamos la comparación con la red dada en la consulta y con la red que se nos envió para que la comparemos.

De un análisis visual comparando ambas redes desbloqueadas, son igual de restringidas por la estructura de la red de petri ya que dos de los tres invariantes de transición se ven bloqueados al tener estados en el que algunas plazas en el que nunca le llegan recursos. Por lo cual aunque en los primeros disparos la red puede funcionar con dos hilos cuando la plaza P13 se quede sin tokens van a funcionar solamente las transiciones T0 T1 T3 T4 o T0 T2 T4 T5 dependiendo de la red (original o a comparar). Si comparamos con la red que modificamos nosotros no se da el mismo caso ya que a todas las plazas le llegan tokens aunque favorece algunas plazas más que otras.

De un análisis analítico obtuvimos los posibles estados para ambas redes de Petri obteniendo todos los marcados posibles utilizando el mapa REACHABILITY/COVERABILITY, considerando las plazas de procesos, obviando las plazas idle (buffers) y las que no son de recursos.

Red original (Consigna / Bloqueada) : Cantidad de estados = 15

Red original (Consigna / Bloqueada)								
Plazas	P1	P2	P3	P4	P5	P6	P7	Suma
Tokens	9	7	4	0	11	5	0	36
Promedio	0,6	0,47	0,27	0	0,73	0,33	0	2,4

Red modificada por nosotros : Cantidad de estados = 24

Red modificada por nosotros								
Plazas	P1	P2	P3	P4	P5	P6	P7	Suma
Tokens	10	15	4	8	5	8	5	55
Promedio	0,42	0,63	0,17	0,33	0,21	0,33	0,21	2,29

Red a comparar : Cantidad de estados = 12

Red a comparar								
Plazas	P1	P2	P3	P4	P5	P6	P7	Suma
Tokens	7	6	0	3	5	5	0	26
Promedio	0,58	0,5	0	0,25	0,42	0,42	0	2,17

Del análisis obtenemos que nuestra red de Petri presenta mayor cantidad de estados (24), por lo cual al sumar la cantidad de tokens en las plazas nos da una suma mayor que en el resto de los casos. Si calculamos el promedio, es decir, la suma de cantidad de tokens dividido la cantidad de estados de la red, obtenemos que nuestra red presenta mayor paralelismo que la red a comparar. Podemos ver en los cálculos que aunque la red original presenta mayor paralelismo, también sabemos que está bloqueada, por lo cual no entraría en una comparación justa.

Luego de desbloquear la red empezamos con el código, y logramos generar un monitor que funcionaba sin colas, ni política, y tampoco sin transiciones temporales. Y lo probamos con

un red simple que no presentaba conflictos en su estructura, y vimos que el vector de estado y las transiciones que se estaban sensibilizando se calculaban correctamente.

Pasamos a utilizar la red que habíamos desbloqueado nosotros, pero el problema era que todavía no habíamos implementado los arcos inhibidores en nuestro código que requería nuestra red. Una vez que solucionamos esto incluyendo el cálculo del vector B, agregamos las colas de los hilos en las transiciones y la política. En este punto tuvimos muchas complicaciones, ya que los hilos se iban todos a esperar en las colas y no se despertaban adecuadamente bloqueando el programa, probamos distintos métodos de concurrencia y distintos tipos de datos para solucionar estos problemas, que al final pudimos solucionar.

Ya con el monitor funcionando para transiciones inmediatas decidimos implementar las transiciones temporales como nos pedía el enunciado. Al principio teníamos de nuevo el problema de que los hilos se iban todos a esperar en las colas del monitor, cambiando un poco la lógica para el disparo de transiciones temporales pudimos solucionar los problemas y nos quedó un monitor que dispara transiciones temporales, utilizando la política y las colas para los hilos en las transiciones.

Explicación de código

Vamos a ir detallando cada una de las clases implementadas, explicando los métodos y variables utilizadas, y la funcionalidad que ocupa en el código.

Política:

La clase política se encarga de administrar que política se va a utilizar en el monitor a la hora de liberar un hilo en las colas del monitor. La definieron distintos modos para la política.
Si el modo es 0 la política está desactivada, se retorna una transición aleatoria del vector m.
Si el modo es 1 la política va a decidir por la transición que menos se disparó y que esté en el vector m.
Si el modo es 2 la política primero se va a fijar que invariante se disparó menos y va a sacar la transición que se disparó menos de ese invariante, y que también se encuentre en el vector m.

Dentro de la clase Política tenemos otra clase definida que la llamamos InvariantesMap, esta es una clase auxiliar que utilizamos para implementar la política en el modo 2.

Métodos:

Política(int modo) : El constructor se va a encargar de definir el modo en el que va a actuar la política en el monitor, también va a construir los 3 elementos del arreglo de la clase Temp que nos servirán a la hora de utilizar la política de modo 2.

Transicion cualDisparo(Boolean[] m, RedDePetri rdp) : Este es el método que llamamos del monitor cuando tenemos al menos una transición sensibilizada que tiene un hilo en su cola, y lo utilizamos para liberar un hilo de las colas del monitor. Dependiendo el modo de la política va a ejecutar un determinado algoritmo. Si el modo es 2 utilizamos la clase InvariantesMap y generamos un arreglo con 3 elementos (cantidad de invariantes de la red), donde cada elemento tiene la cantidad de disparos por invariante y tiene el índice del

invariante a la cual pertenece esa cantidad, de tal forma que si ordenamos este arreglo con el criterio de cual invariante se disparó menos al que se disparó más, podemos acceder a las transiciones que tiene ese invariante y poder buscar invariante por invariante las transiciones que dispararon menos allí. Si el modo es 1, vamos a ordenar las transiciones por cuales se dispararon menos y vamos a buscar desde la transición que menos se disparó cual de ellas tiene un true en el vector m, que a diferencia del modo 2, no nos fijamos cual invariante se disparó menos y solamente nos fijamos en las transiciones. Si el modo es 0, guardamos en una lista aquellas transiciones que tienen un true en el vector m, y retornamos aleatoriamente alguna de esas transiciones.

Monitor:

Clase encargada del manejo de los hilos que disparan la red de petri.

Métodos empleados:

Monitor() : Constructor del monitor en donde se guardan los objetos a utilizar durante el manejo de los hilos.

Boolean[] quienesEstan () : Recorre el array de las colas chequeando si existe algún hilo esperando en las colas.

void disparaTransicion (Transicion transicion) : Se encarga de manejar los hilos, mandándolos a dormir o esperar si no pudieron disparar. si pudieron disparar y existen algún hilo en las colas lo despierta según la política para que puedan volver a intentar a disparar.
Este método en su mayoría se encuentra sincronizado con un semáforo binario, soltando el semáforo solamente cuando la transición que quiere disparar el hilo está habilitada y antes de la ventana o cuando los hilos no pudieron disparar y deben esperar en la cola a que algún otro hilo los notifique.

void setCondicion () : Establece la condición de corte para la finalización de los hilos, que depende de la cantidad de invariantes que queremos disparar en la red de Petri. A este método también lo utilizamos para liberar las colas y desbloquear todos los hilos para que se finalicen de forma correcta.

Cola:

Todo aquel hilo que no pudo disparar la red de petri, vuelve al monitor y se lo coloca en la cola de la transición que se quiere disparar.
Cada cola inicialmente va a tener un semáforo inicializado con 0 token. Cuando un hilo se tiene que esperar en una cola, hacemos un acquire de este semáforo e incrementa una variable llamada hilosCola. Cuando un hilo es despertado por la política hacemos un release a este semáforo y decrementamos la variable hilosCola.

Cola() : Vamos a tener un objeto Cola por cada transición de la red. En el constructor seteamos en 0 la cantidad de hilos asociados a la cola e inicializamos el semáforo con tokens = 0.

void increment() : Incrementa el valor de la variable hilosCola. Este método lo vamos a llamar antes de liberar el semáforo del monitor y siempre que se llame al método acquire para una cola.

void acquire() : Realiza un acquire al semáforo que pertenece a la cola de transición que se quiere dormir. Como el semáforo se construye con tokens igual a 0, el hilo que realice el acquire no va a poder continuar y va a ser bloqueado hasta que otro hilo haga un release a la cola de transición que se durmió. Logramos un funcionamiento similar a un wait().

void release() : Decrementa el valor de la variable hilosCola y realiza el release al semáforo de esa cola de transición. Con esto liberamos el hilo que estaba esperando en el acquire previo.

RedDePetri:

Esta clase se encarga de cargar y calcular las matrices y los vectores. Administra la lógica de los disparos temporales e inmediatos, es utilizada por el monitor para ver si una transición se puede disparar o no.

boolean disparar(Transicion transicion): Cuando el monitor dispara la red se fija si la transición esté sensibilizada, si es inmediata la dispara en caso contrario es temporal Entonces se fija en tres cosas el hilo:

1. Se fija si está dentro de la ventana del disparo, si está setea a estado con -1.
2. Si no está en la ventana chequea si está antes y que no haya nadie esperando o que el que está esperando fue el mismo. Si lo anterior es verdadero entonces calcula el tiempo y setea a estado con -2.
3. Al no estar en la ventana del disparo o antes, quiere decir que está después del beta por lo cual el programa termina, ya que no es una situación que nos interesa a nosotros resolver.

Luego el hilo dispara la red o sale de la red de petri sin modificarla, con la información de lo que deben hacer los hilos en el monitor.

boolean antesDeLaVentana(int posicion) : Si la suma entre el timeStamp y el alpha es mayor que el tiempo actual quiere decir que el hilo llegó antes de la ventana por lo cual retorna *true*, en caso contrario *false.*

void verificarPInvariantes() : Cada vez que disparamos una transición verificamos que se cumplan las ecuaciones de los invariantes de plaza.

boolean esDisparoValido(int[] marcado_siguiente) : Le pasamos un marcado y se fija si las plazas del marcado actual son menores a cero. Esto lo utilizamos para verificar si un disparo puede realizarse o no.

int[] marcadoSiguiente(int[] old, int position): Cálculo del vector de estado después de disparar una transición sumando el vector viejo con la columna de la posición de la transición de la matriz de incidencia.

Boolean[] getVectorE(): Calcula y devuelve el vector de las transiciones sensibilizadas según el vector de estado actual de la red de petri.

Boolean[] getVectorQ(): Itera el vector de estado, si un elemento es distinto de cero *guarda* true en un elemento de un array y en caso contrario *false.* Luego retorna el array anterior.

Boolean[] getVectorB() : El vector B es un vector booleano que se encarga de desensibilizar aquellas transiciones que están inhibidas por un arco inhibidor. Es el producto entre la matriz de inhibidora y el vector Q.

void getVectorEandB() : Modifica el vector de vectorEandB realizando una operación AND entre el vector B y el vector E.

Boolean[] getSensebilizadas() : Primero chequea si hay alguna transición inmediata sensibilizada, si existe alguna parsea todas las transiciones inmediatas y pone como true las transiciones inmediatas. Si no es asi devuelve el vector de vectorEandB.

Transición:

La clase transición la utilizamos para tener las transiciones con su id, la posición en la red, y si es temporizada o no. Esto lo utilizamos en la lógica temporal de la red, y al id para las expresiones regulares.

SensibilizadasConTiempo:

Esta clase se encarga de contener información de las transiciones temporales. La variable *esperando es del tipo* AtomicBoolean ya que es editada afuera del semáforo del monitor se puede dar el caso que varios hilos intenten modificarlas, por lo cual debe estar sincronizada.

boolean testVentanaTiempo(): Este método se fija si el hilo que quiere disparar esta entra el tiempo alpha y beta que posee la transición, el resultado depende del tiempo actual y el timeStamp.

void nuevoTimeStamp(): Guarda el tiempo el cual una transición temporal se sensibilizó, y resetea las variables *id* y *esperando.*

void setEsperando(): Seteamos el atributo esperando y el id del que estaba esperando.

void resetEsperando(): reseteamos el atributo esperando y el id del que estaba esperando.

Gráfico:

Una vez se completaron los invariantes y todos los hilos terminaron su ejecución, esta clase se encarga de cargar las librerías jfreecharts y mostrar en una ventana la cantidad de disparos por invariante en el tiempo, la clase ajusta los ejes y los valores en 3 series distintas una por cada invariante y tomando como dominio el tiempo en el cual se tomó la muestra. La muestra de los datos va a depender de la cantidad de invariantes que disparemos en la red. Siendo la relación = cantidad de disparos/10, esto quiere decir que si disparamos 1000 invariantes la relación va a ser de 100, entonces cada 100 veces que los hilos entren al monitor se va a registrar el dato de la cantidad disparada por invariante y el tiempo en el que se toma la muestra.

Resultados Obtenidos

Decidimos realizar el análisis con cada una de nuestras computadoras individualmente ya que tienen características diferentes. También optamos por utilizar 2 configuraciones distintas de la red para realizar las comparaciones. Y para ver que efectivamente la política está balanceando los disparos en los invariantes realizamos estos cálculos con ambas políticas. Cada análisis va a constar de 10 ejecución de 1000 invariantes y haciendo un promedio entre las 10 ejecuciones.

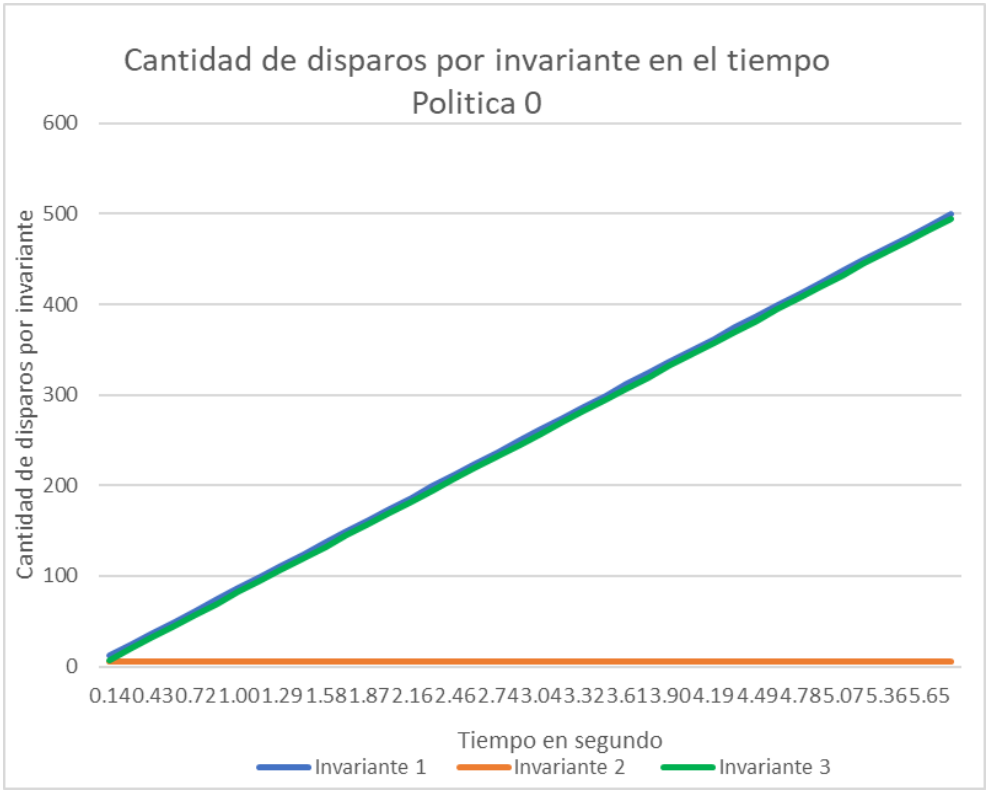
Política 0 → Random, se toma una transición aleatoria.
Política 2 → Intentamos tomar la transición menos disparada del invariante de transición menos disparado.

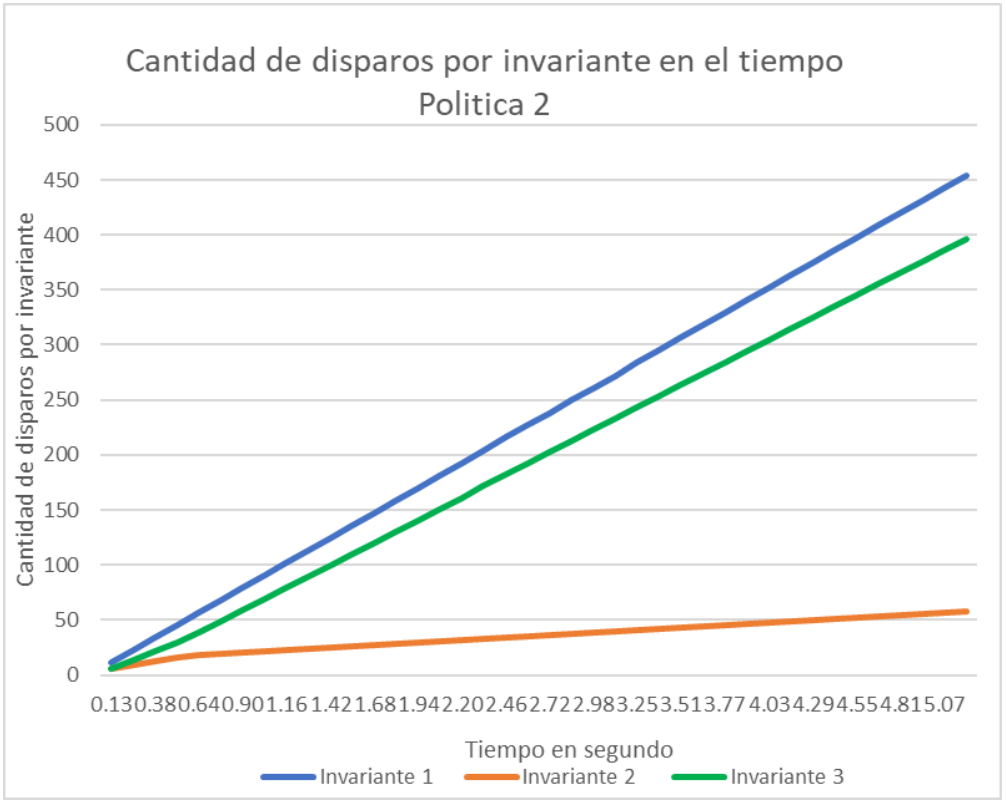
Con cada ejecución obtenemos la serie de datos que utilizamos para armar los gráficos, estás series contienen la cantidad de disparos por cada uno de los invariantes y el tiempo en el que se realizó esa muestra.
Calculando el promedio de las series en las 10 ejecuciones obtenemos las series de datos que graficamos.

El primer análisis lo vamos a hacer sobre la red que logramos desbloquear de la consigna, conservando las transiciones temporales e inmediatas. Vamos a optar por utilizar los tiempos según la siguiente planilla:

Tiempos Usados	
alfa	beta
-1	-1
2	100000
2	100000
4	100000
2	100000
2	100000
-1	-1
3	100000
2	100000
2	100000

Obtenemos los siguientes gráficos:





Notamos que hay una leve mejora cuando activamos la política pero al tener transiciones inmediatas nos dificulta balancear los invariantes con la política, sumado a la particular geometría de la red. Dado esto, para un mejor análisis, vamos a optar por quitar ambas

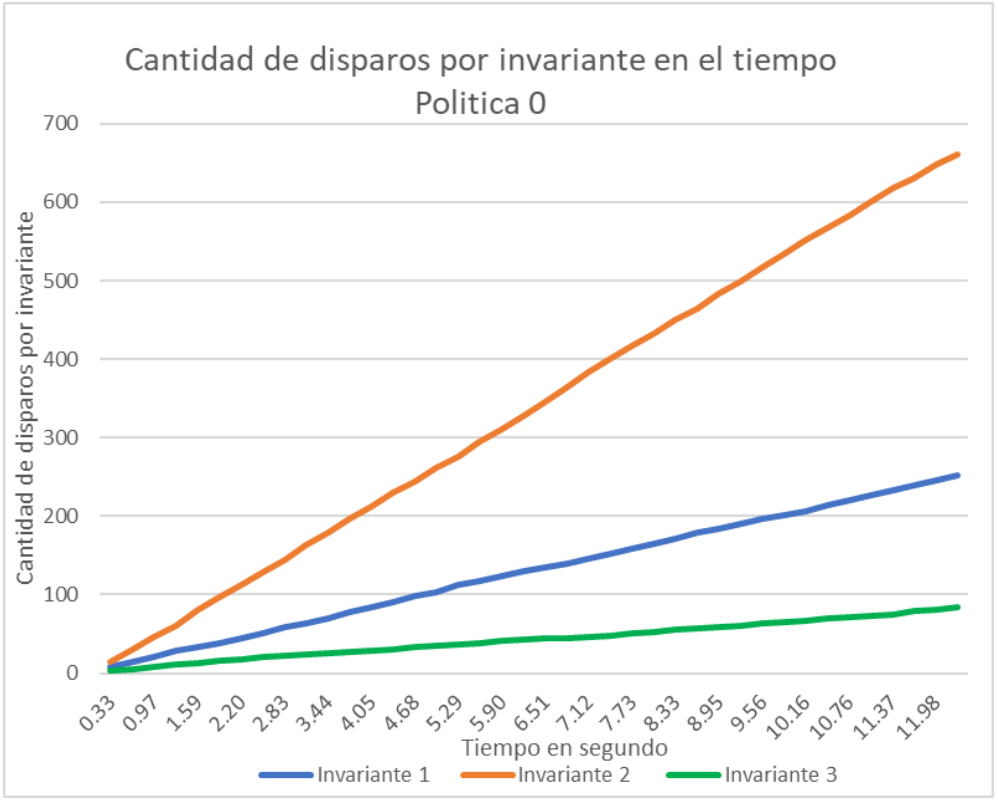
transiciones inmediatas para que se nos haga más notorio el análisis a la hora de ver si la política logra mejorar el balanceo en los disparos de los invariantes.

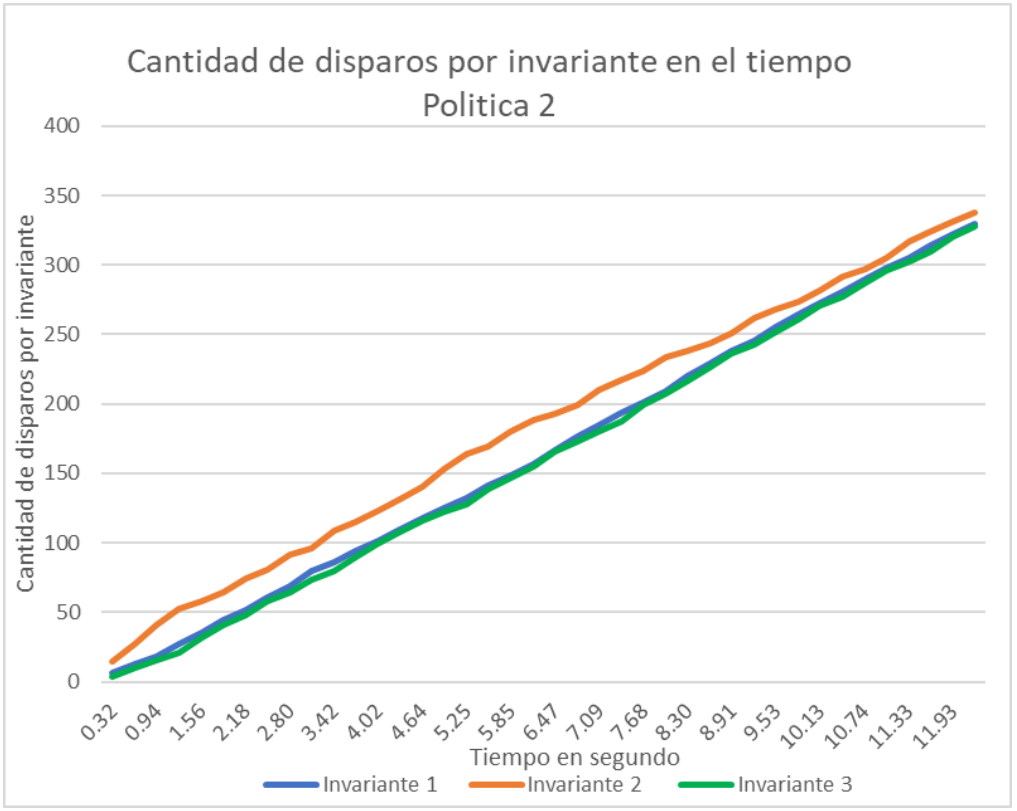
El segundo análisis lo vamos a realizar quitando las transiciones inmediatas por unas temporales. El análisis se va a realizar de la misma manera.

Vamos a utilizar los siguientes tiempos:

Tiempos Usados	
alfa	beta
5	100000
5	100000
4	100000
5	100000
2	100000
6	100000
5	100000
2	100000
1	100000
1	100000

Se obtienen los siguientes gráficos:





Vemos una diferencia notoria en el balance de los invariantes cuando usamos una política que elige una transición random con respecto a una política que va a elegir la transición menos disparada del invariante menos disparado.

Observamos que en ambos casos, si comparamos las gráficas utilizando la política y no utilizando, la política ayuda para que la cantidad de disparos por invariante se mantenga más equilibrada a lo largo del tiempo de ejecución. En el caso donde quitamos las transiciones temporales vimos que es más notoria el actuar de la política, pero también podemos ver que la ejecución cuando tenemos las transiciones inmediatas pudimos favorecer el balanceo de los invariantes.

Los resultados son en parte dependientes de los tiempos que tengan las transiciones temporales, el aumento o disminución del tiempo de una transición no se traslada en una disminución o aumento de la cantidad de disparos de un invariante, depende de la estructura de la red y el interleaving de los hilos. Además la lógica del programa también influye en la evolución de la red y la aplicación de la política, en otra implementación de la red de petri es muy probable que los tiempos elegidos en este informe no logren balancear los disparos de los invariantes, ya que pueden cambiar la interacción de los hilos y la ejecución de las tareas atómicas.

Conclusión

Con este trabajo pudimos interiorizarnos en el funcionamiento de las herramientas de concurrencia vistas en clase y otras que no pero que son de igual utilidad para el control de hilos de forma paralela o asincrónica.

Nos dimos cuenta de la complejidad y de los cuidados que se deben tener a la hora de trabajar con muchos hilos concurrentemente para que el programa se comporte de la forma deseada.

Una implementación de la política más focalizada en una red de petri teniendo en cuenta la estructura de ésta y la evolución que tiene en el transcurso del tiempo probablemente sea más robusta a los cambios de los tiempos de la transiciones temporales ya que una implementación simple como la nuestra no tiene en cuenta que al permitir el disparo de un invariante bloqueamos otros invariantes hasta que se vuelva al estado inicial.

Bibliografía

- https://fcefyn.aulavirtual.unc.edu.ar/pluginfile.php/11702/course/section/3806/Monitores_2020.pdf Filminas de clase
- Datos recopilados de los disparos de la red https://docs.google.com/spreadsheets/d/1Gid0bLHvMPYXIrFttlbNTjPBM8vi43YPuNjb_V4xpuc/edit?usp=sharing