Facultad de Ciencias Exactas, Físicas y Naturales Programacion Concurrente

Trabajo Práctico Final: Sistema de manufacturación robotizado simulado mediante Redes de Petri

AGUERREBERRY, Matthew Mat:93.739.112
mtaguerreberry@gmail.com
TOMATTIS, Natasha Mat:38.728.783
natitomattis@gmail.com
TROMBOTTO, Agustin Mat:39.071.116
agustin.trombotto@alumnos.unc.ed.ar

12 de abril de 2017

Resumen Las Redes de Petri conforman una herramienta gráfica y matemática que puede aplicarse a cualquier sistema. Especialmente, a los sistemas paralelos que requieran simulación y modelado de la concurrencia y el uso en recursos compartidos. En este trabajo, se presenta un sistema de manufactura con estas características y se modela por medio de Redes de Petri con el objeto de analizar su dinámica y hallar la secuencia óptima de funcionamiento del sistema.

Palabras Clave: Redes de Petri, sistema de manufactura , Concurrencia, Monitor.

1. Introducción

En un sistema robotizado, es de suma importancia el correcto funcionamiento de cada maquinaria garantizando, de esta manera, que no haya bloqueos entre las herramientas de producción. Es por ello que se realiza la simulación del proceso mediante una Red de Petri y su correspondiente ejecución en un programa en Java. El objetivo de este trabajo es asegurar que todo el sistema funcione de acuerdo a la lógica establecida y los tiempos pautados. A su vez se busca obtener ciertas cantidad de piezas producidas según la política de producción que se quiera brindar al sistema. De esta manera, el productor podría elegir cuanta cantidad de cada pieza puede realizar. Al finalizar y resolver esta situación obtendremos un sistema fiable, el cual no se dentrá y funcionará perfectamente en aquellas ocasiones criticas del sistema.

2. Problema propuesto

El sistema de manufactura consiste en tres robots R1, R2 y R3 cuatro máquinas M1, M2, M3 y M4, tres tipos diferentes de piezas a procesar A, B y C, como se observa en la figura. Las piezas provienen de tres contenedores de entrada distintos, I1, I2 e I3, de los cuales los robots las retiran, las colocan en las máquinas para su procesamiento y depositan en tres contenedores de salidas distintos, que son: O1, O2 y O3.

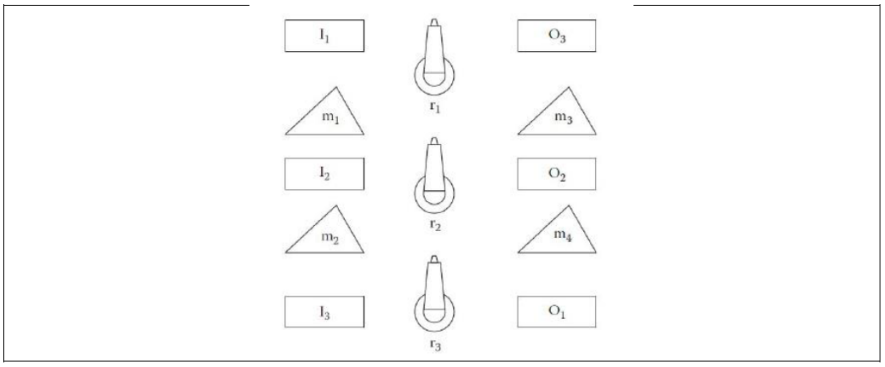

Figura 1: Esquema del sistema a implementar.

Adicionalmente se deberá agregar un tiempo asociado al trabajo de las maquinas, para aproximarse una aplicación en tiempo real.

3. Solución a la red de petri

A la presentación del problema se le adjunto una primera aproximación a una red de Petri que representa el problema, pero esta red no funciona correctamente ya que no cumple con ciertas propiedades inherentes a la concurrencia. Estas propiedades aseguran el correcto ejecución de las instrucciones del programa en paralelamente.
La propiedad que no se cumple en este caso es la de intrebloqueo; un problema de los sistemas concurrentes en donde todos los procesos están esperando un evento que nunca ocurrirá. Para evitar esta situación se utilizarán restricciones para que un solo hilo pueda acceder a la vez a secciones donde se puede producir interbloqueo.
Se realizara la verificación de esta propiedad de la red, por medio de un software de simulación, PIPE. El cual por medio de un modelo matemático obtiene las propiedades de la Red.
Se propone agregar tres plazas a la red para evitar el interbloqueo:
Plaza 1: Por medio de simulaciones de producción de una pieza única en todo el sistema, se comprobó que en el caso de la pieza B en su camino se bloqueaba cuando se intentaban producir dos piezas en el mismo camino. Esto se debe a que en la producción de esta pieza se emplea dos veces al mismo robot (R2), entones cuando una de las piezas se esta produciendo y entra otra que ocupa el robot la primera no podrá disparar la transición dado que el robot esta ocupado y la segunda no podrá disparar su transición siguiente porque la primera pieza esta usando la maquina que esta necesita.

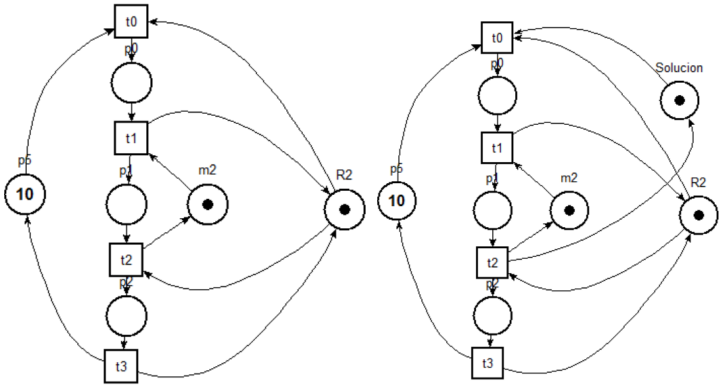

Figura 2: Esquema simplificado del primer problema de interbloqueo.

Plaza 2: Una vez finalizadas las simulaciones de la producción de piezas del mismo tipo, se procedió a realizar simulaciones de a dos piezas al mismo tiempo, dividiendo la producción de la pieza A en dos habilitando los caminos por separado. De este modo se descubrió un interbloqueo entre la linea de producción de la pieza B y uno de los caminos de la pieza A. Siendo en este caso nuevamente el robot R2 y la maquina M2 los recursos en conflicto. En este caso también se agrego una restricción para que accedan a estos recursos una linea de producción a la vez.

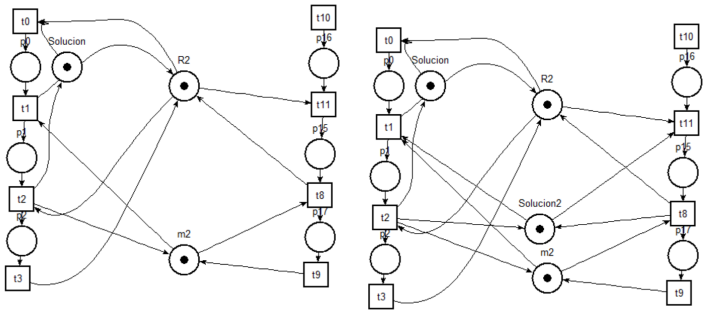

Figura 3: Esquema simplificado del segundo problema de interbloqueo.

Plaza 3: En este caso es encontró que la linea de producción de la pieza C era la misma que una de las alternativas de la pieza A, por lo tanto la única solución a este conflicto es agregar una restricción que no permita que ambas lineas produzcan al mismo tiempo.

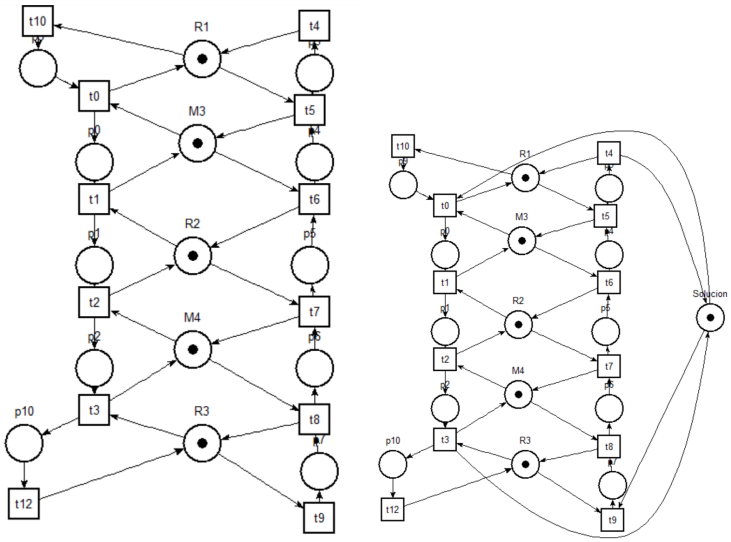

Figura 4: Esquema simplificado del tercer problema de interbloqueo.

Como resultado se obtiene la siguiente red, se puede comprobar con el software PIPE que no posee interbloqueo.

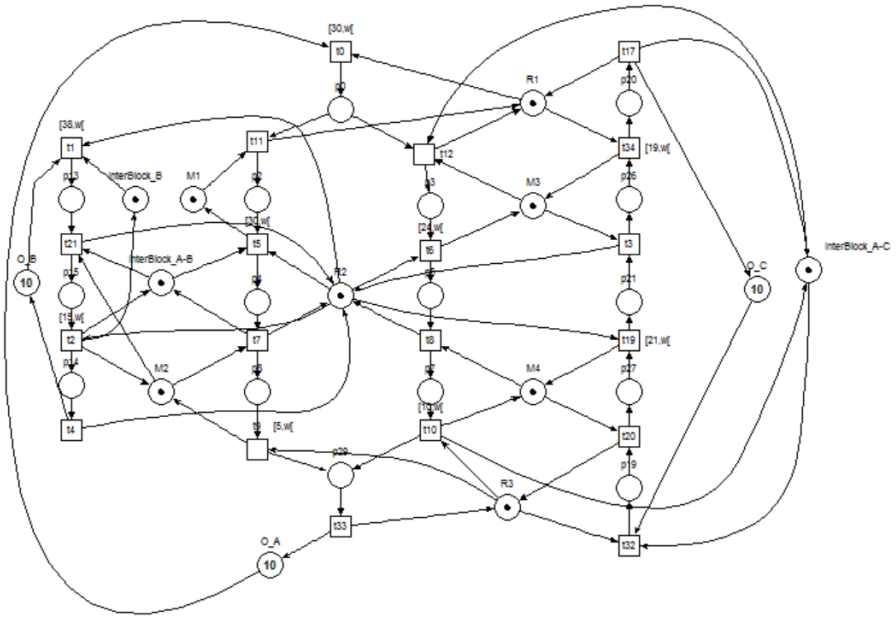

Figura 5: Red con problemas de interbloqueo solucionado.

En la figura también se incluyen los tiempos asociados a las transiciones, según se indico en el enunciado. También se debieron agregar tiempos adicionales a algunas transiciones para equilibrar la producción de piezas de distintos tipos.

Una vez construida y verificada la red, se procede al desarrollo de un programa que pueda simular su comportamiento.

4. Implementación de un monitor con Red de Petri

4.1. Clases

Con el objetivo de modularizar el código y procurar el desacoplo de cada parte, se decidió la implementación de las siguientes clases:

- Main
- Rdp
- GestorMonitor
- Politicas
- Colas
- Mutex
- Tiempo

Se muestra a continuación el diagrama de clases y luego se desarrollará una breve descripción de las clases más importantes y se detallan los métodos más relevantes

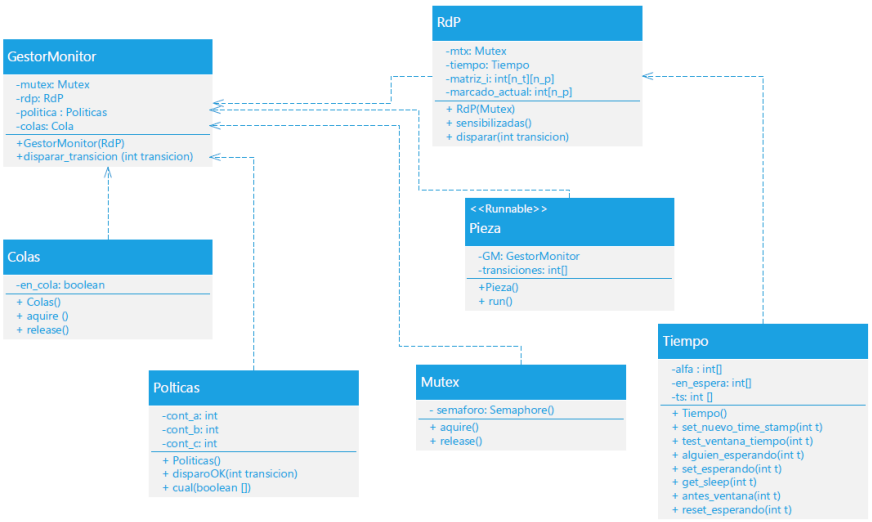

Figura 6: Diagrama de clases de un monitor.

4.1.1. Main

Se inicializa el gestor de monitor pasando los parámetros necesarios y los 5 hilos que ejecutaran sus correspondientes transiciones.

4.1.2. Rdp

Contiene toda la lógica del monitor, es decir controla la exclusión mutua y administra los recursos disponibles. De este modo esta clase contiene las matrices de incidencia pre y pos, y el vector de marcado; estas variables combinadas representan el estado de la red en un momento determinado.
Entre sus funciones mas relevantes se encuentran:

- *Sensibilizadas():* devuelve un arreglo de boolean que representa las transiciones sensibilizadas en un momento determinado. Este vector se calcula sumando el vector de marcado actual a la matriz de incidencia I, una vez realizada esta operación, se controla que en la columna correspondiente a una transición no existan elementos negativos. Si existe un valor negativo para una plaza en la columna de una transición, esta transición no esta sensibilizada; en el caso contrario, la transición esta sensibilizada.

- *Disparar():* realiza(en el caso que sea posible) el disparo de la red y devuelve un entero indicando el resultado del disparo. El disparo se realiza con la ecuación de estado de la Red de Petri:

  $$m_{n+1} = m_n + I \times d_j$$

  donde m_{n+1} es el marcado en el instante n+1, m_n el marcado en el instante n, I la matriz de incidencia y d_j el vector de disparo para una transición j.
  Una vez calculado el resultado del disparo de una transición se hace el cheque de tiempo, por medio de los métodos de la clase Tiempo

4.1.3. GestorMonitor

En esta clase se encarga de la gestión de hilos después de un disparo, tiene un solo método que es disparar transición, en el cual el hilo debe acceder al gestor por medio de un mutex de entrada. De este modo el gestor se asegura que solo un hilo pueda disparar la red a la vez.
Una vez realizad el disparo pueden ocurrir varias alternativas, el hilo puede haber disparado la transición correctamente, entonces el gestor le da la señal al hilo para que dispare la próxima transición. Luego existen varios casos en los que no se puede disparar la transición y según la situación el hilo puede irse a una cola FIFO, dormirse o simplemente salir del monitor para nuevamente intentar disparar la misma transición. Si un hilo realiza un disparo, si hay otros hilos en las colas del gestor deberá despertarlos, si hay mas de un hilo en las colas y con sus respectivas transiciones sensibilizadas se acudirá a la clase política para decidir cual disparar.
La imagen a continuación es un diagrama de secuencias que permite explicar el funcionamiento del gestor de monitor sin tiempo:

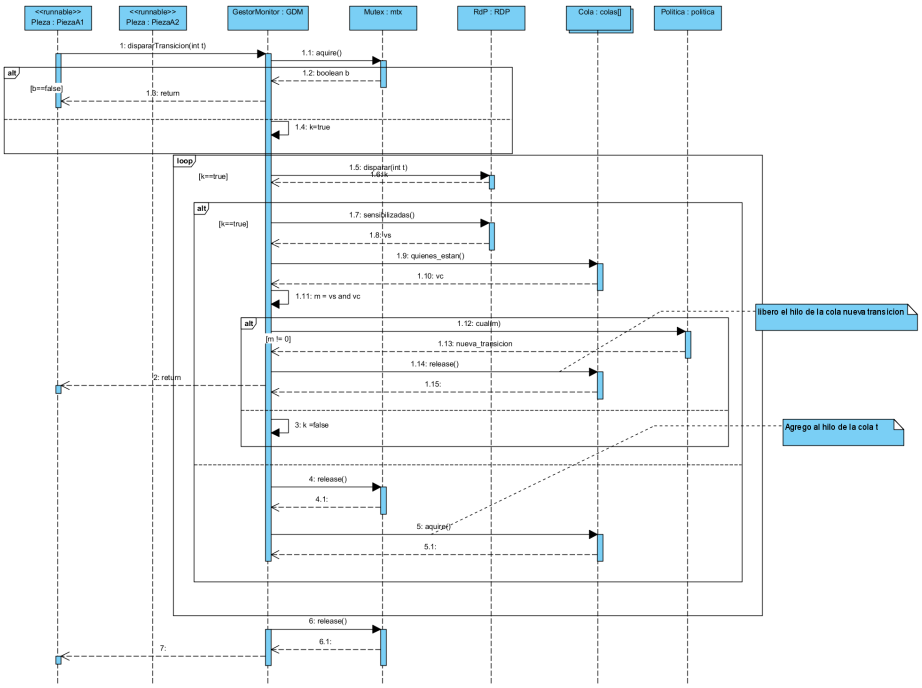

Figura 7: Diagrama de secuencia de disparo de una transición sin tiempo.

4.1.4. Politicas

Esta clase es la que toma la decisión de que transición disparar en el caso en que haya dos o mas transiciones sensibilizadas y con hilos en sus colas. Lleva contadores que se actualizan en cada disparo, cuentan la cantidad de piezas terminadas de cada tipo y como parámetro tiene las proporciones de producción de cada pieza. Los métodos mas relevantes son:

- Cual(): se ele pasa un vector con las transiciones sensibilizadas y devuelve la transición que se debe disparar. Se calcula multiplicando el vector por una matriz que lo ordena según la prioridad actual de producción de piezas
- ActualizarP(): actualiza la matriz que ordena el vector según la cantidad de piezas producidas.

4.1.5. Cola

La colas tienen un papel importante en el desarrollo de software concurrente aplicado a una red de petri. Las colas son las encargadas de establecer un ”fila”para los hilos que esperarán a disparar una transición determinada. Es por esto que cada transición de la red tendrá asociado un objeto de esta clase, la misma contendrá dos métodos:

- acquire(): el hilo ejecutor de este método se va a ”dormir.ª la cola (que en estado interrumpido) hasta la ejecución del release de esta clase
- release(): despierta el hilo dormido en la cola

4.1.6. Mutex

El mutex es la ”puerta de entrada.ªl monitor. Todo hilo que busque ejecutar una acción dentro del programa deberá ”pedir autorización”para entrar. Esta clase utiliza un Semaphore de la librería java.util.concurrent. El mismo posee una cola de los hilos que buscan entrar al monitor. Con sus métodos acquire() y release() (al igual que en la cola) posibilitarán entrar al hilo al monitor o mandarlos a dormir en espera de autorizacion para su ingreso.

4.1.7. Tiempo

Este objeto es el que almacena los tiempos de sensibilización de cada transición y los retrasos asociados a algunas transiciones. Algunas de sus funciones son:

- TestVentanaTiempo()
- setNuevoTimeStamp()
- alguien_esperando()
- antes_ventana()
- set_esperando()
- reset_esperando()
- get_sleep()

El diagrama a continuación representa el calculo de un disparo con tiempo y como se comportan las funciones anteriormente descriptas:

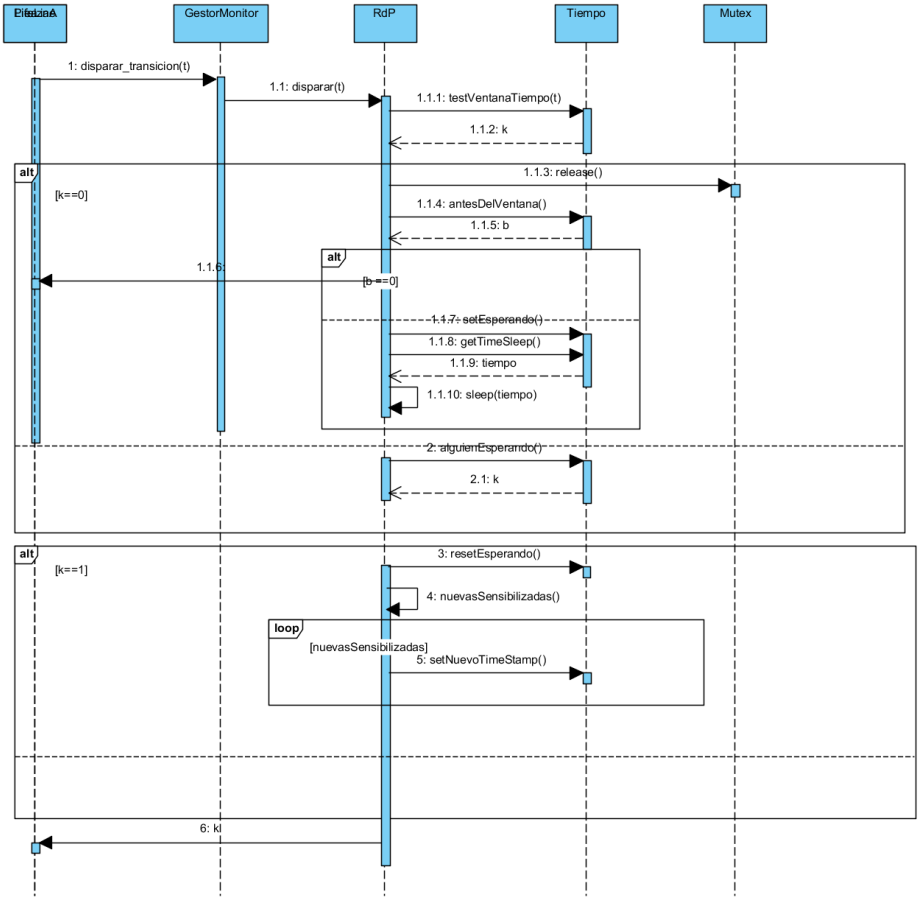

Figura 8: Diagrama de secuencia de disparo de una transición con tiempo.

5. Pruebas y Testing

En todo sistema automático y con secciones criticas, es necesario brindar la seguridad de que el sistema realizar perfectamente las acciones asociadas al mismo. Cabe destacar los fundamentos matemáticos proveídos por las Redes de Petri para este tipo de prueba, como son las P-Invariante. Estos asegurar el correcto funcionamiento de la lógica en el software.

5.1. Unit Test Java

Se generaron dos test unitarios de java para probar el método de disparar transición del objeto Rdp, con la finalidad de corroborar la lógica del programa. En las pruebas se utilizaron dos redes de petri diferentes, una la que corresponde a nuestro caso y la otra una red sencilla. En este caso se utilizó un productor/consumidor.
Para verificar que los disparos se produjeran correctamente y respetando la lógica de la red, estos test se hicieron probando las P-Invariante proporciondas por el software PIPE. Esta prueba garantiza que después de cada disparo el marcado de la red sea coherente con la lógica que representa.

Figura 9: Ejecucion del Unit Test Java. Assert con P-Invariantes

Referencias

[1] Redes de Petri Temporales - Material de Clase

[2] Java 7 Concurrency Cookbook. Javier Fernández González

[3] Generación de Código de Sistemas Concurrentes a partir de Redes de Petri Orientadas a Proceso *http://sedici.unlp.edu.ar/bitstream/handle/10915/41771/Documento_completo.pdf?sequence=1*