Programación Concurrente

Ingeniería en Computación FCEFyN - UNC
Informe Trabajo Final

Gonzalez Gustavo. Mat.: 7721064
Maero Facundo. Mat.: 38479441

August 1, 2017

Abstract

Las Redes de Petri conforman una herramienta gráfica y matemática que puede aplicarse a cualquier sistema. Es de especial utilidad en el análisis dinámico de sistemas concurrentes, ya que permite garantizar el correcto funcionamiento de los mismos y brinda un mecanismo formal para su estudio. Este documento presenta el problema presentado por Naiqi y MengChu en 2010 conocido como "Sistema de manufacturación robotizado", el cual muestra una planta de producción con recursos compartidos (máquinas) y se deben evitar los interbloqueos logrando la producción estipulada de piezas.

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 desarrollado en Java. El objetivo de este trabajo es asegurar que todo el sistema funcione de acuerdo a la lógica y restricciones planteadas en el enunciado (temporales y de producción deseada de cada pieza). De esta manera, el productor podrá definir las proporciones de producción de cada pieza a obtener.

2 Problema Propuesto

El sistema de manufacturación consiste en tres robots: **R1, R2 y R3**, cuatro máquinas: **M1, M2, M3 y M4**, y tres tipos diferentes de piezas a procesar: **A, B y C**.
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**.

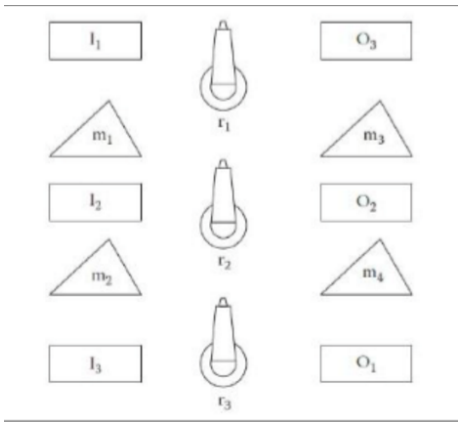


Figure 1: Esquema de la planta de producción propuesto.

Para producir cada pieza hay un camino preestablecido, y las máquinas a utilizar tienen tiempos de demora conocidos:

- Pieza A (camino 1): M1=30; M2=5
- Pieza A (camino 2): M3=24; M4=10
- Pieza B : M2=15
- Pieza C: M4=21; M3=18

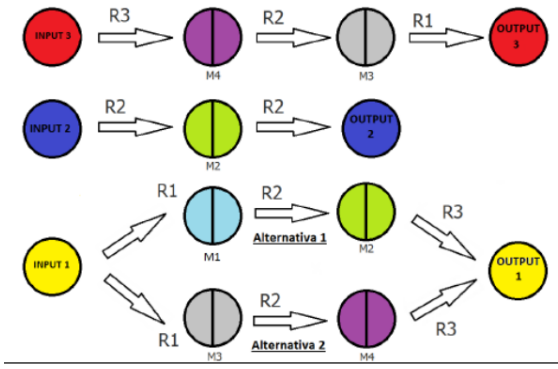


Figure 2: Diferentes vías para producir cada tipo de pieza

3 Análisis de Red de Petri

La red que modela el problema propuesto se muestra en 3. Presenta problemas de interbloqueo que fueron solucionados con la ayuda de las herramientas de análisis **Pipe** y **Tina**.

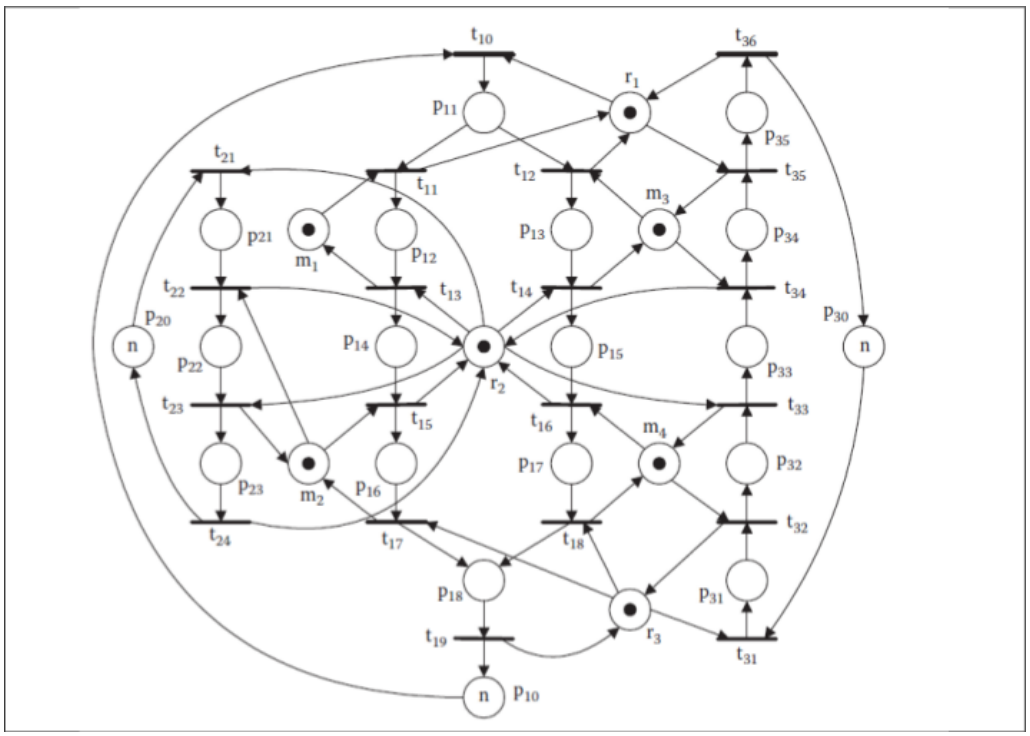


Figure 3: Red de Petri propuesta.

3.1 Solución a Interbloqueos

Para resolver los problemas de la red se incluyeron 3 restricciones, representadas por 3 plazas extras.

3.1.1 Interbloqueo 1

Para la producción de la pieza B se observó que la producción se bloquea si se encuentra un token en las plazas 1 y 2 simultáneamente, por lo que se agregó la plaza 5, que limita a 1 el número de tokens en ambas posiciones. 4

3.1.2 Interbloqueo 2

El caso de interbloqueo 2 se da entre la producción de la pieza B y una de las líneas de la pieza A. Se incluyó una plaza que limite el flujo de tokens en ambos caminos, de manera que se evite el bloqueo de la red en las plazas 2 y 10. 5a

3.1.3 Interbloqueo 3

Aquí la restricción es mayor, ya que debido a la manera en la que se encuentran las máquinas en la planta, la producción de la pieza C tiene el sentido inverso a las demás. Esto causa un conflicto con una de las líneas de la pieza A. Por lo tanto, se debe evitar que se produzcan piezas de estos tipos al mismo tiempo por los caminos mencionados. 5b

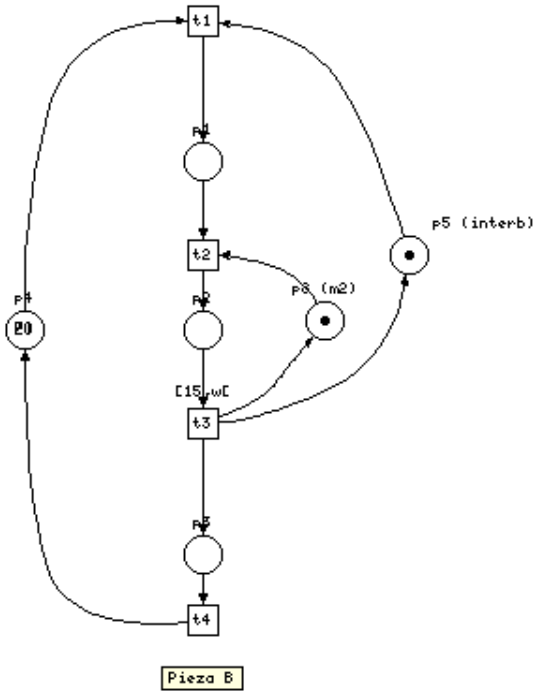
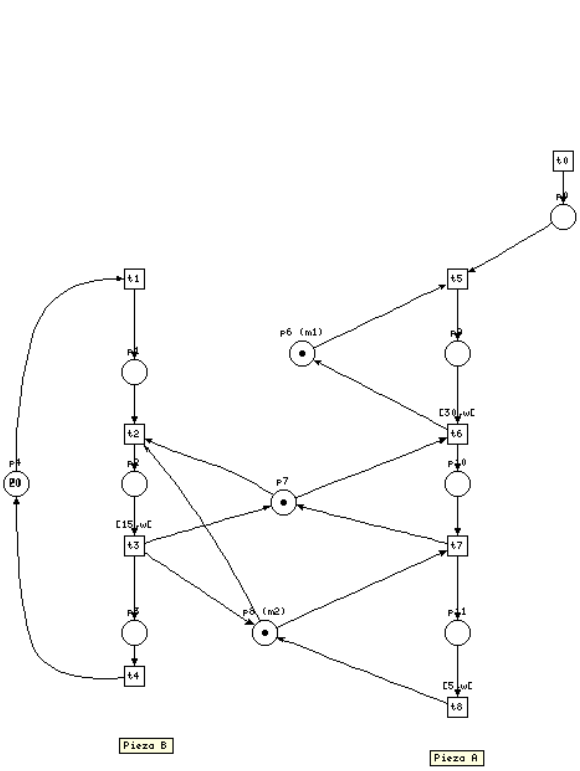
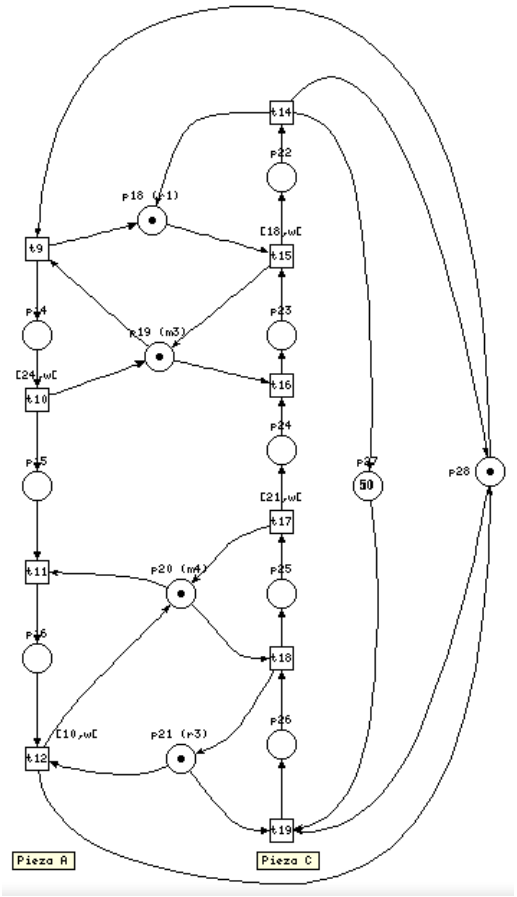


Figure 4: Solución al problema de Interbloqueo 1.



(a)



(b)

Figure 5: Solución a problemas de Interbloqueo 2 y 3

3.2 Comprobación de la Solución

La herramienta **Pipe** cuenta con opciones de análisis y caracterización de redes, las cuales fueron utilizadas para verificar que las restricciones agregadas evitan el interbloqueo.
Concretamente el análisis de espacio de estados nos arroja **Deadlock: false**. 6

Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	false

Figure 6: .Análisis de la red con Pipe.

Esto nos asegura que la red por sí misma no se bloqueará.

3.3 Red de Petri sin Interbloqueo

Se procedió a diagramar la red del problema con las correcciones previas en el software **Tina** 7

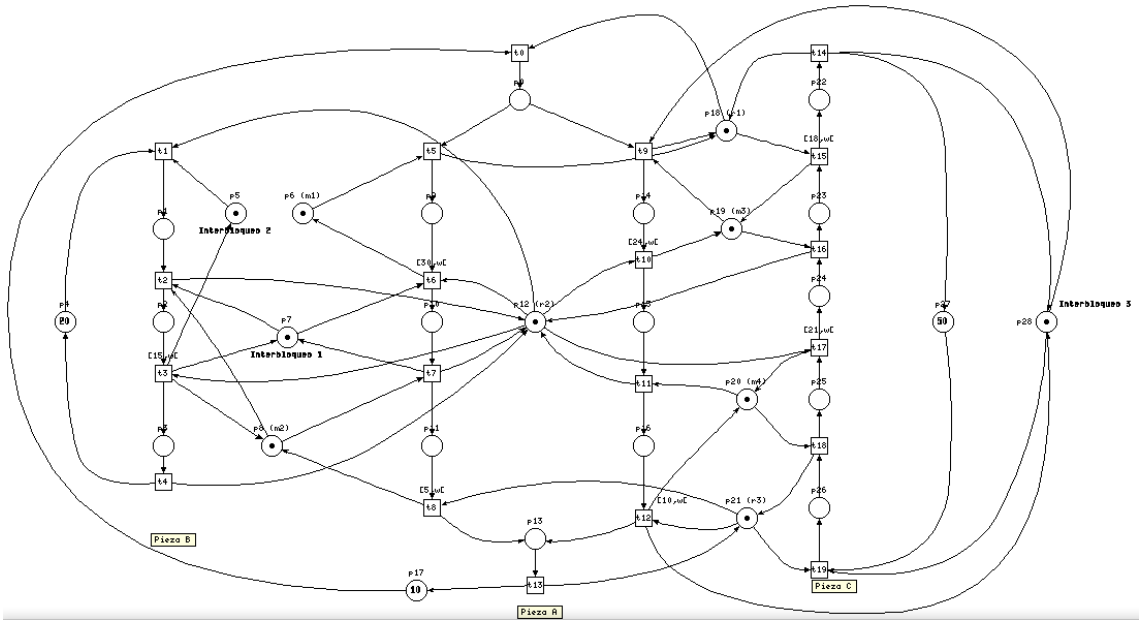


Figure 7: Red sin interbloqueo.

3.4 Tabla de Eventos

Detalla las transiciones presentes en la red y qué simboliza cada una.

Table 1: Tabla de Eventos

Transición	Significado
T0	Robot 1 toma una pieza A de I1
T1	Robot 2 toma una pieza B de I2
T2	Robot 2 deja la pieza B en Maquina 2
T3	Robot 2 toma la pieza B de Maquina 2
T4	Robot 2 deja la pieza B terminada en O2
T5	Robot 1 deja la pieza A en Maquina 1
T6	Robot 2 toma la pieza A de Maquina 1
T7	Robot 2 deja la pieza A en Maquina 2
T8	Robot 3 toma la pieza A de Maquina 2
T9	Robot 1 deja la pieza A en Maquina 3
T10	Robot 2 toma la pieza A de Maquina 3
T11	Robot 2 deja la pieza A en Maquina 4
T12	Robot 3 toma la pieza A de Maquina 4
T13	Robot 3 deja la pieza A terminada en O1
T14	Robot 1 deja la pieza C terminada en O3
T15	Robot 1 toma la pieza C de Maquina 3
T16	Robot 2 deja la pieza C en Maquina 3
T17	Robot 2 toma la pieza C de Maquina 4
T18	Robot 3 deja la pieza C en Maquina 4
T19	Robot 3 toma la pieza C de I3

3.5 Tabla de Estados

Detalla las plazas en la red, y qué simboliza cada una.

Table 2: Tabla de Estados

Plaza	Significado
P0	Pieza A en Robot 1, viene de I1
P1	Pieza B en Robot 2, viene de I2
P2	Pieza B en Maquina 2
P3	Pieza B en Robot 2, luego de pasar por Maquina 2
P4	Pool de recursos de Pieza B
P5	Plaza de protección contra Interbloqueo en producción de Pieza B
P6	Plaza de recurso. Representa a la Maquina 1
P7	Plaza de protección contra Interbloqueo en producción de Pieza B y Pieza A
P8	Plaza de recurso. Representa a la Maquina 2
P9	Pieza A en Maquina 1
P10	Pieza A en Robot 2, luego de pasar por Maquina 1
P11	Pieza A en Maquina 2
P12	Plaza de recurso. Representa al Robot 2
P13	Pieza A en Robot 3, luego de pasar por Maquina 2 o Maquina 4
P14	Pieza A en Maquina 3
P15	Pieza A en Robot 2, luego de pasar por Maquina 3
P16	Pieza A en Maquina 4
P17	Pool de recursos de Pieza A
P18	Plaza de recurso. Representa al Robot 1
P19	Plaza de recurso. Representa a la Maquina 3
P20	Plaza de recurso. Representa a la Maquina 4
P21	Plaza de recurso. Representa al Robot 3
P22	Pieza C en Robot 1
P23	Pieza C en Maquina 3
P24	Pieza C en Robot 2
P25	Pieza C en Maquina 4
P26	Pieza C en Robot 3, viene de I3
P27	Pool de recursos de Pieza C
P28	Plazade protección contra Interbloqueo en producción de Pieza C y Pieza A

4 Diseño de Solución

Para resolver el problema se programó un proyecto en **Java**, utilizando el entorno de desarrollo **Eclipse Oxygen**. Se implementó un Monitor, estructura de control de concurrencia de alto nivel, que permite gestionar los hilos que actúan sobre un objeto en común, y lo protegen de los problemas propios de la concurrencia, como la corrupción de datos y acceso simultáneo a objetos compartidos.

4.1 Diagrama de clases y definición de las mismas 8

La resolución del problema incluye la construcción de las siguientes estructuras, que en conjunto simulan la operación de la planta:

- Red de Petri: contiene la lógica de disparo y la estructura de la red del problema.
- Gestor de Monitor: estructura de control que maneja el acceso a la red para protegerla de los problemas de concurrencia.
- Hilo: trabajador, simboliza el flujo de trabajo para obtener piezas de distintos tipos.
- Políticas: dados múltiples caminos a seguir, elige el más conveniente en función de los requerimientos que posee.
- Tiempo: agrega un retardo temporal al disparo de ciertas transiciones en la red, que simboliza la demora de una máquina en trabajar con una pieza.

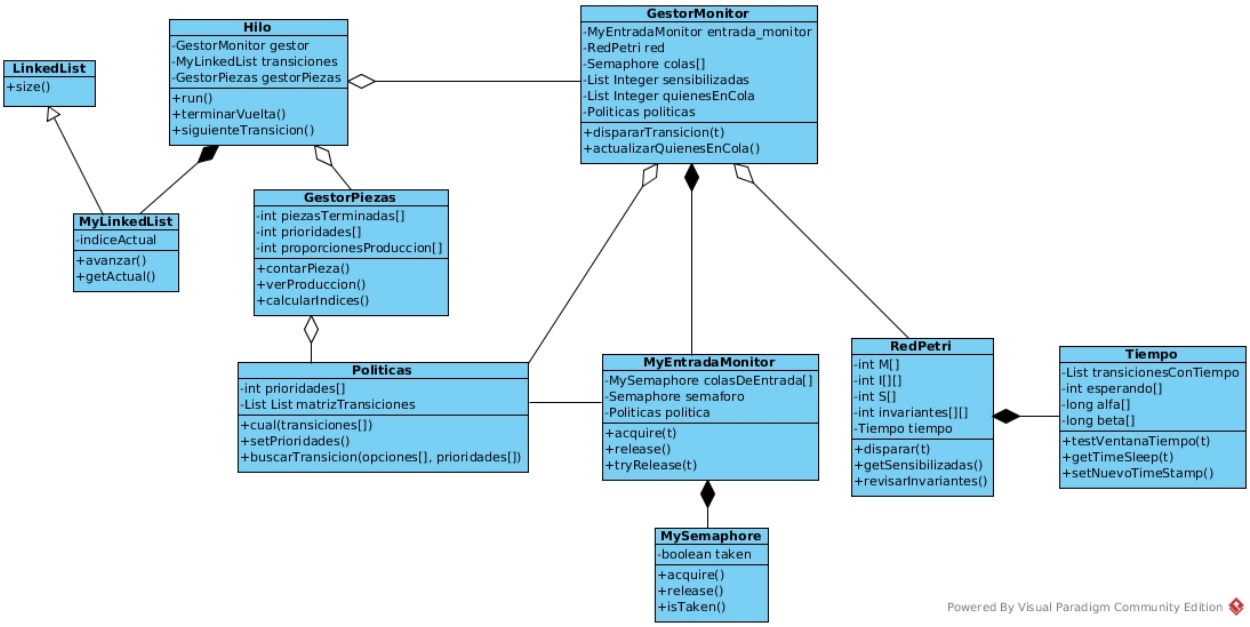


Figure 8: Diagrama de clases.

4.1.1 GestorMonitor

Es la clase central del proyecto. Una vez creada inicializa la Red de Petri, los semáforos para los hilos, su gestor de entrada y demás estructuras necesarias. Este gestor de entrada permite que solo un hilo se encuentre en el Monitor al mismo tiempo, funcionalidad lograda mediante un mutex especial. Cuando un hilo intenta disparar una transición en la Red, el Monitor decide qué debe hacer éste (despertar a otro hilo, salir, entrar en una cola, esperar un tiempo).

4.1.2 MyEntradaMonitor

Dada la complejidad del problema, la entrada al monitor no se pudo resolver solamente con un semáforo, por lo que se creó esta clase. Posee, al igual que el monitor, un conjunto de colas para

que los hilos duerman, esperando para poder entrar. Posee también un semáforo convencional que actúa como mutex. Esta clase debe conocer a las Políticas, para, teniendo múltiples hilos esperando para entrar, tomar la mejor decisión entre ellos. Los métodos principales son:

- *acquire(t)*: Dada una transición a disparar *t*, intenta tomar el mutex. Si no puede, el hilo se pone a esperar en la cola de la transición *t*.
- *release()*: Una vez disparada una transición, el hilo deja libre la entrada del monitor, o elige entre los hilos esperando cuál debe entrar.
- *tryRelease(t)*: método muy similar al anterior, pero además de elegir entre los hilos esperando, se tiene en cuenta a sí mismo también. Esto es útil ya que si el hilo actual es el que tiene más prioridad, volverá a entrar al monitor antes que los demás.

4.1.3 RedPetri

Esta clase representa a la Red, su marcado, transiciones, plazas, lógica de disparo, invariantes y sensibilizaciones. Posee una referencia a un objeto **Tiempo**, para utilizarlo si las transiciones son temporizadas. Posee el método *disparar(t)* que intenta realizar el disparo de *t*, pudiendo ser exitoso, no estar en la ventana de tiempo, o fallido.

4.1.4 Tiempo

Contiene la información temporal de las transiciones de la red actual, incluyendo el valor de **alfa**, **beta**, y el tiempo de sensibilización. Permite averiguar si el intento de disparo de un hilo está en la ventana correcta, calcular el tiempo que tiene que esperar, y actualizar los timestamps de las transiciones al cambiar el marcado de la red.

4.1.5 Hilo

Es la única clase viva en el proyecto. Posee un bucle en el que intenta disparar continuamente transiciones de la red, accediendo al Monitor por medio de su mutex de entrada. Cada hilo posee una lista circular de transiciones asignadas, en orden de disparo. Posee además una referencia a un Gestor de Piezas, clase global que lleva la cuenta de las piezas producidas de cada tipo, y permite gestionar las políticas.

4.1.6 GestorPiezas

Esta clase contabiliza la producción de las piezas A, B y C. Cada vez que un hilo termina su recorrido, y por lo tanto, fabrica una pieza, le informa al gestor, que la contabiliza, y procede a calcular las relaciones entre las piezas producidas. Luego reordena el vector con las prioridades y lo guarda en **Politicas** para así alterar el flujo del programa.

4.1.7 Políticas

Se encarga de tomar las decisiones con respecto a qué transición tiene prioridad, dado un conjunto de posibles opciones. Para ello tiene un vector de prioridades de 3 elementos, uno por cada tipo de pieza. Además, tiene vectores con las transiciones de cada tipo de pieza, ordenadas internamente también por prioridad. Así, cuando cambian las prioridades, el Gestor de Piezas le informa a esta clase. Esto permite que si un hilo consulta qué transición es más importante, se consideren los vectores en el orden mencionado.

4.2 Funcionamiento general

Para el funcionamiento del proyecto se utilizan 5 hilos: uno para las piezas B y C respectivamente, 2 para la pieza A (uno para cada camino), y uno más para la transición en común entre ambos caminos. A continuación se detallan partes fundamentales del proyecto, con diagramas que pretenden explicar en alto nivel su funcionamiento.

4.2.1 Comportamiento de disparo - Hilos

Cada hilo conoce el conjunto de transiciones que debe disparar. Se intenta disparar la primera. Si se tuvo éxito, se avanza en la lista, pero si no se pudo disparar (sucede si no se está en la ventana de tiempo), se intenta nuevamente con el mismo índice. Una vez que se complete el recorrido, se avisa al Gestor de Piezas, para que cuente la pieza recién producida.

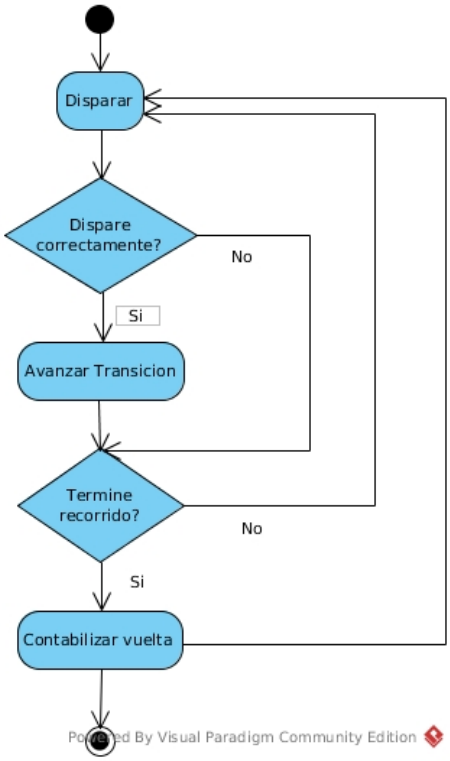


Figure 9: Diagrama de Actividades - Disparo de transiciones.

4.2.2 Entrada y salida del Monitor

La entrada al monitor comienza con un *tryAcquire* del semáforo. Si el método devuelve *true*, el hilo intenta disparar normalmente. Si se obtuvo *false*, el hilo espera en la cola correspondiente a su transición actual. Una vez se retornó de la red, pueden suceder dos escenarios:

- Si la transición no estaba sensibilizada, el hilo debe ir a la cola. Por lo tanto se pregunta quienes son los demás hilos esperando en la entrada. Si hay alguno, se consulta a la política por la decisión. Si no hay nadie, se libera el semáforo.
- Si la transición se disparó, o si se tiene que esperar hasta alfa, se procede de manera similar, pero al preguntar a la política por el hilo siguiente, se incluye a sí mismo también. Esto permite que si el hilo sigue siendo más importante que los demás que están esperando, pueda entrar nuevamente y volver a disparar. Por lo tanto, en lugar de despertar a otro hilo, libera el semáforo, como si no hubiera nadie esperando, y vuelve a intentar entrar.

Este comportamiento tiene la ventaja de "impedir" el disparo de otras transiciones, sin el peligro de que la red se bloquee, ya que el hilo actual volverá a ingresar al Monitor y disparará de nuevo.

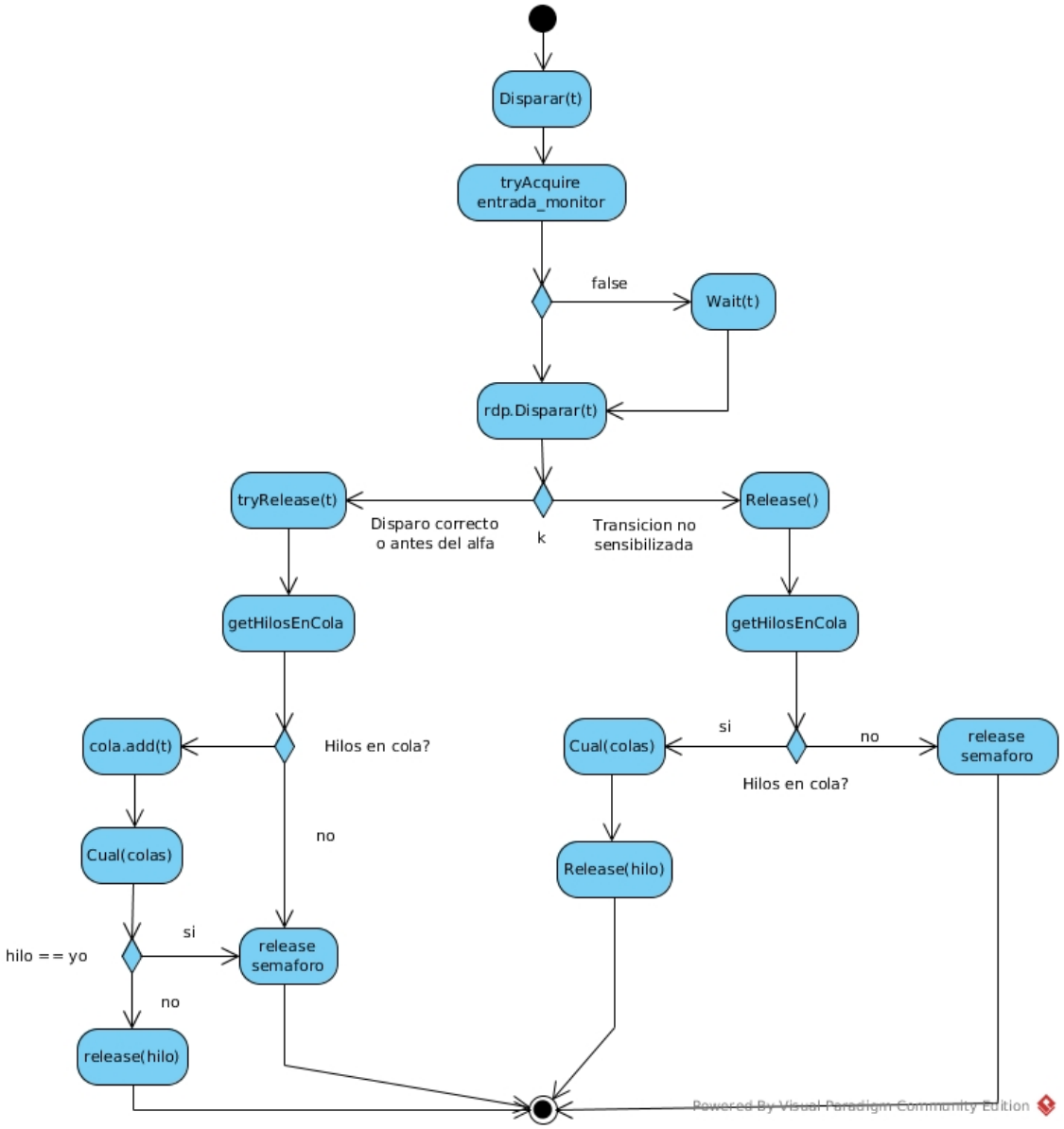


Figure 10: Diagrama de Actividades - Manejo de hilos dentro del Monitor.

4.2.3 Secuencia de Disparo

En 11 se muestra un diagrama de secuencia que explica el proceso de disparo en un monitor con tiempo. Se detalla la relación entre la clase GestorMonitor y RedPetri, utilizando también la clase Tiempo y Políticas para mostrar el desarrollo del programa en un caso típico.

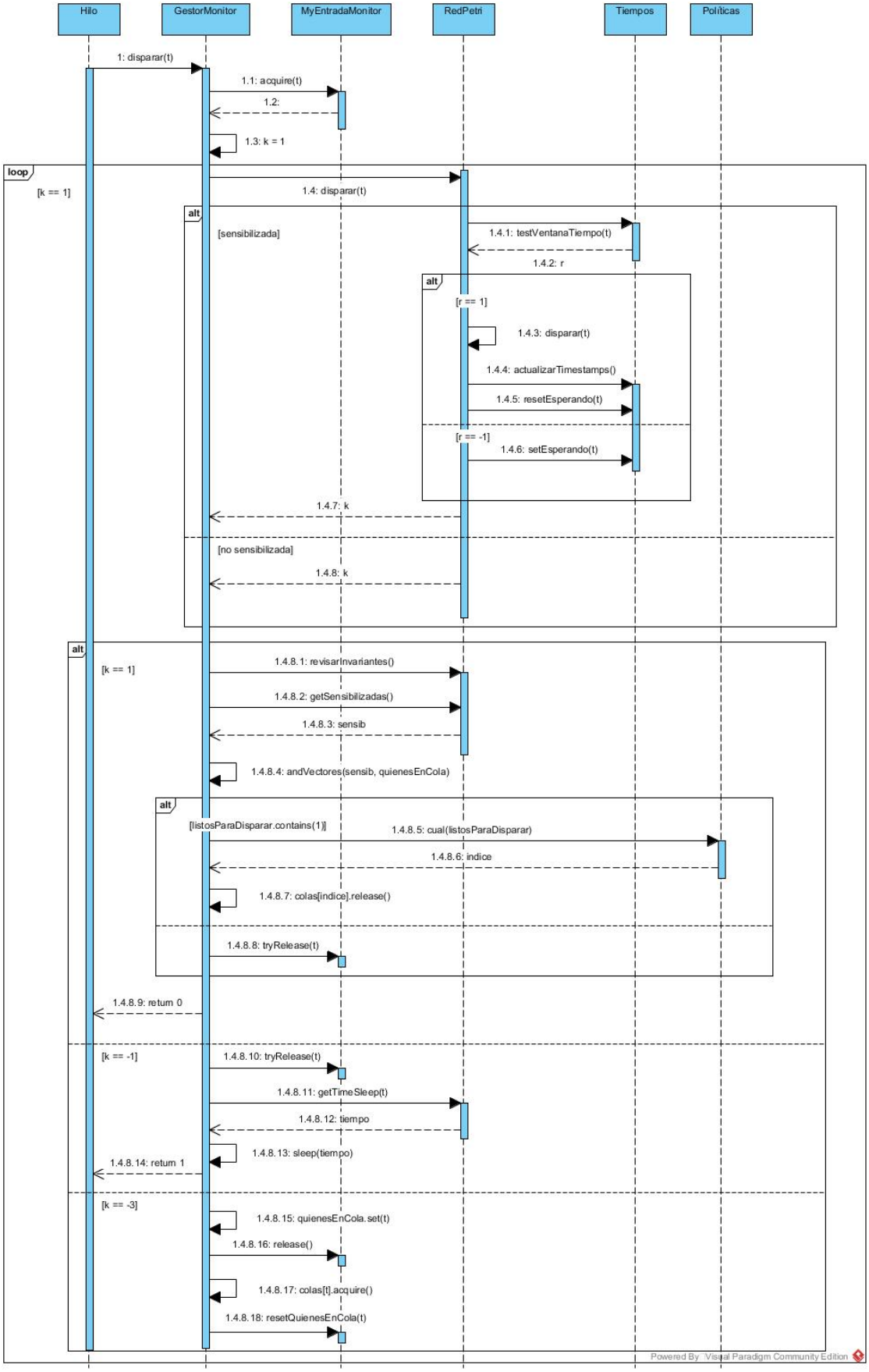


Figure 11: Diagrama de Secuencia - Disparo en Monitor.

5 Funcionamiento y Testing

5.1 Test de Política

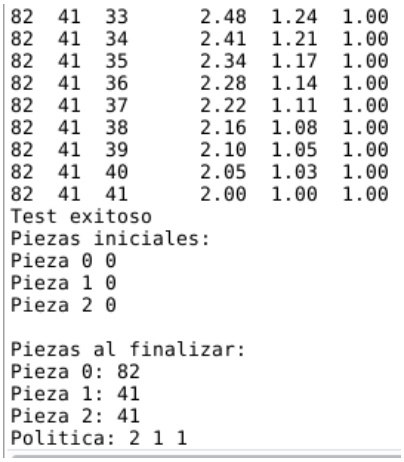
Se programó un test de políticas en la clase **Gestor de Piezas**, que termina la ejecución del programa cuando se alcanzan las proporciones ingresadas. Por medio del mismo se evalúa el correcto funcionamiento de todo el sistema.

Table 3: Test de Políticas 1

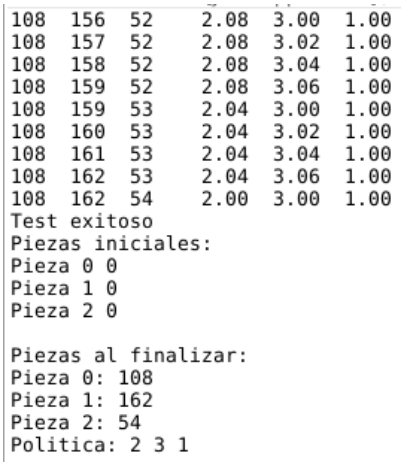
Descripción del Test	Comprueba que las políticas de producción se cumplan
Ejecución	Setear la proporción de producción deseada en la clase Gestor de Piezas en 2 1 1 Crear el Monitor, la Red de Petri, el Gestor de Piezas y el objeto Tiempo Producir 20000 piezas en total Comprobar en el archivo "log.txt" como fueron variando las proporciones de salida
Resultado Esperado Pass/Fail	Proporciones 2 1 1 con un margen de ± 0.1 Pass
Observaciones	El test es exitoso porque se subutilizan los recursos. Se prioriza alcanzar la proporcion solicitada. Si se quisiera utilizar la planta a su máxima capacidad, el test fallaría.

Table 4: Test de Políticas 2

Descripción del Test	Comprueba que las políticas de producción se cumplan
Ejecución	Setear la proporción de producción deseada en la clase Gestor de Piezas en 2 3 1 Crear el Monitor, la Red de Petri, el Gestor de Piezas y el objeto Tiempo Producir 20000 piezas en total Comprobar en el archivo "log.txt" como fueron variando las proporciones de salida
Resultado Esperado Pass/Fail	Proporciones 2 3 1 con un margen de ± 0.1 Pass
Observaciones	El test es exitoso porque se subutilizan los recursos. Se prioriza alcanzar la proporcion solicitada. Si se quisiera utilizar la planta a su máxima capacidad, el test fallaría.



(a) A Proporciones [2 1 1]



(b) A Proporciones [2 3 1]

Figure 12: Resultados Tests de Política

Como puede verse, los números de la derecha muestran la proporción real medida con respecto a la pieza C. Los dígitos de la izquierda muestran la producción actual.

En 13 y 14 puede verse la misma información de manera gráfica.

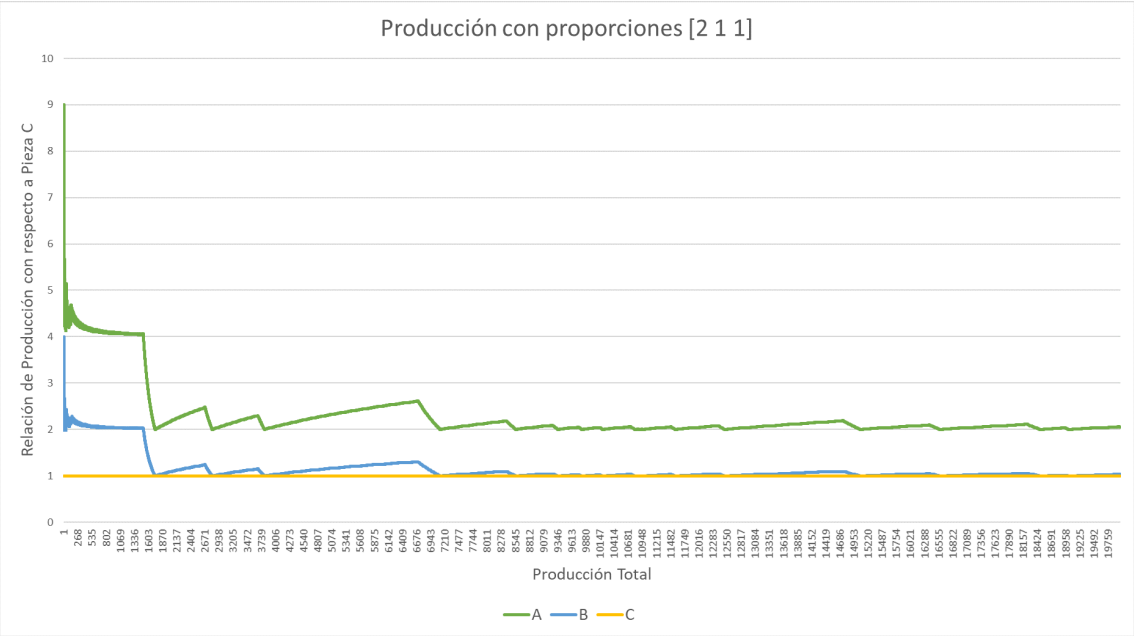


Figure 13: Produccion de 20.000 piezas con proporcion 2 1 1.

Figure 14: Produccion de 20.000 piezas con proporcion 2 3 1

Es importante destacar que se alcanzaron los resultados solicitados mediante la subutilización de los recursos. Esto se da porque si un hilo utiliza la función *tryRelease()*, estará modificando la política al intentar priorizarse a sí mismo y no dejando que los demás hilos ingresen al monitor.

5.2 Test de Invariantes

Table 5: Test de Invariantes

Descripción del Test	Checkea los P-Invariantes en la red
Ejecución	Crear una nueva Red de Petri con el marcado inicial del práctico. Disparar 20 transiciones. Comprobar la salida estándar del sistema.
Resultado Esperado	Ningun Assertion Error en consola
Pass/Fail	Pass
Observaciones	Los invariantes se comprueban luego del disparo de cada transición. Si se produce un fallo se arroja una excepción.

5.3 Test de Estados Inválidos

Table 6: Test de Estados Inválidos

Descripción del Test	Verifica que la red no ingrese en estados inválidos
Ejecución	Crear una nueva Red de Petri con el marcado inicial del práctico. Disparar 20 transiciones. Verificar que no se ingrese en 5 estados invalidos que bloquean el sistema Comprobar la salida estándar.
Resultado Esperado	Ningun Assertion Error en consola
Pass/Fail	Pass
Observaciones	Los estados invalidos son: P2 + P11 >1 P1 + P10 >1 P15 + P25 >1 P10 + P15 >1 P2 + P10 + P14 + P23 >1

5.4 Test Linked List Circular

Table 7: Test MyLinkedList

Descripción del Test	Verifica el funcionamiento de la Linked List Circular
Ejecución	Crear una nueva MyLinkedList Agregar los valores 4, 3 y 2 Avanzar el puntero 3 veces Leer el valor del puntero
Resultado Esperado	Valor del puntero = 4
Pass/Fail	Pass

5.5 Test Transiciones Sensibilizadas

Table 8: Test Transiciones Sensibilizadas

Descripción del Test	Comprueba las transiciones sensibilizadas en la red luego de varios disparos
Ejecución	Crear una nueva Red de Petri Disparar un conjunto de transiciones determinado Ver cantidad de transiciones sensibilizadas Ver cuales son las transiciones sensibilizadas
Resultado Esperado	Ningun Assertion Error
Pass/Fail	Pass
Observaciones	Las transiciones a disparar son: 0,1,2,19,5,0,18,17 La cantidad de transiciones sensibilizadas debería ser 1 La transición en cuestión debería ser la T16

5.6 Test Hilo Durmiendo

Table 9: Test Hilo Durmiendo

Descripción del Test	Comprueba que un hilo duerma correctamente en la cola que le corresponde
Ejecución	Leer los archivos con los datos necesarios Crear un Gestor de Monitor con los datos leidos Crear una Red de Petri Crear un Hilo nuevo Asignarle las transiciones 5,6,7,8,13, que corresponden al Hilo 1 Iniciar el hilo Dormir por 100ms para darle tiempo al hilo que interactúe con el Monitor Verificar que el hilo este durmiendo en la cola de T5
Resultado Esperado	Hilo durmiendo en cola de T5
Pass/Fail	Pass

5.7 Test Politicas

Table 10: Test Políticas

Descripción del Test	Verifica el funcionamiento de las Políticas al elegir la siguiente transición
Ejecución	Crear objeto Politicas con transiciones por pieza Crear vector de transiciones con opciones a elegir Asignar prioridades a Politicas Preguntar a Politicas cual es la transicion a disparar
Resultado Esperado	Ningun Assertion Error
Pass/Fail	Pass
Observaciones	Las transiciones sensibilizadas son T0, T1 y T19, que corresponden a las lineas de produccion A, B y C. El test se ejecutó 3 veces, modificando el vector de prioridades para que elija las 3 opciones, una cada vez.

6 Conclusión

Con la realización de este proyecto se comprobó que las Redes de Petri son una herramienta muy poderosa para el modelado de procesos en general. Gracias a su sustento matemático y propiedades conocidas, es posible tener certezas sobre determinadas características de nuestro problema, que utilizando otra herramienta no son comprobables.
Por otro lado, el Monitor es una estructura de control que, combinada con una Red de Petri, permite simular el sistema planteado en la misma, en un entorno multihilo de manera segura y eficiente.
Al centralizar el control de concurrencia en una sola estructura, se resuelven muchos problemas inherentes al uso de semáforos y locks en la programación en general, como lo son la distribución de los mismos por todo el código, y el tedioso mantenimiento de un proyecto con esas características.
A pesar de esto, el Monitor es una herramienta costosa en cuanto a recursos, y si su implementación no es correcta, pueden ocurrir interbloqueos indeseados, y problemas de gestión de hilos, que no estaban presentes al momento de analizar la red por separado.
En este caso particular fue necesaria la modificación del modelo visto en clase del Monitor, para introducir el uso de políticas en su mutex de entrada, y la capacidad de un hilo de disparar múltiples transiciones sin intervención de los demás. Este último cambio permitió que se respeten las políticas aún cuando los tiempos de las diferentes máquinas son muy dispares.
El costo de este mecanismo es la subutilización de los recursos, por lo que debe optarse entre mantener la proporción deseada, o utilizar al máximo cada robot y máquina, obteniendo cantidades diferentes a las esperadas para cada pieza.

7 Referencias

1. Redes de Petri Temporales - Material de Clase
2. Java 7 Concurrency Cookbook. Javier Fernández González
3. `http://pipe2.sourceforge.net/`
4. `http://projects.laas.fr/tina//`
5. `https://github.com/facundojmaero/FinalConcurrente`