FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES
UNIVERSIDAD NACIONAL DE CÓRDOBA

PROGRAMACIÓN CONCURRENTE

TRABAJO PRÁCTICO FINAL

CONTROL DE UN CIRCUITO FERROVIARIO

Docentes:

- Micolini, Orlando.
- Ventre, Luis.

Integrantes:

- Abratte, Diego.
- Moral, Ramiro.

ÍNDICE

ENUNCIADO

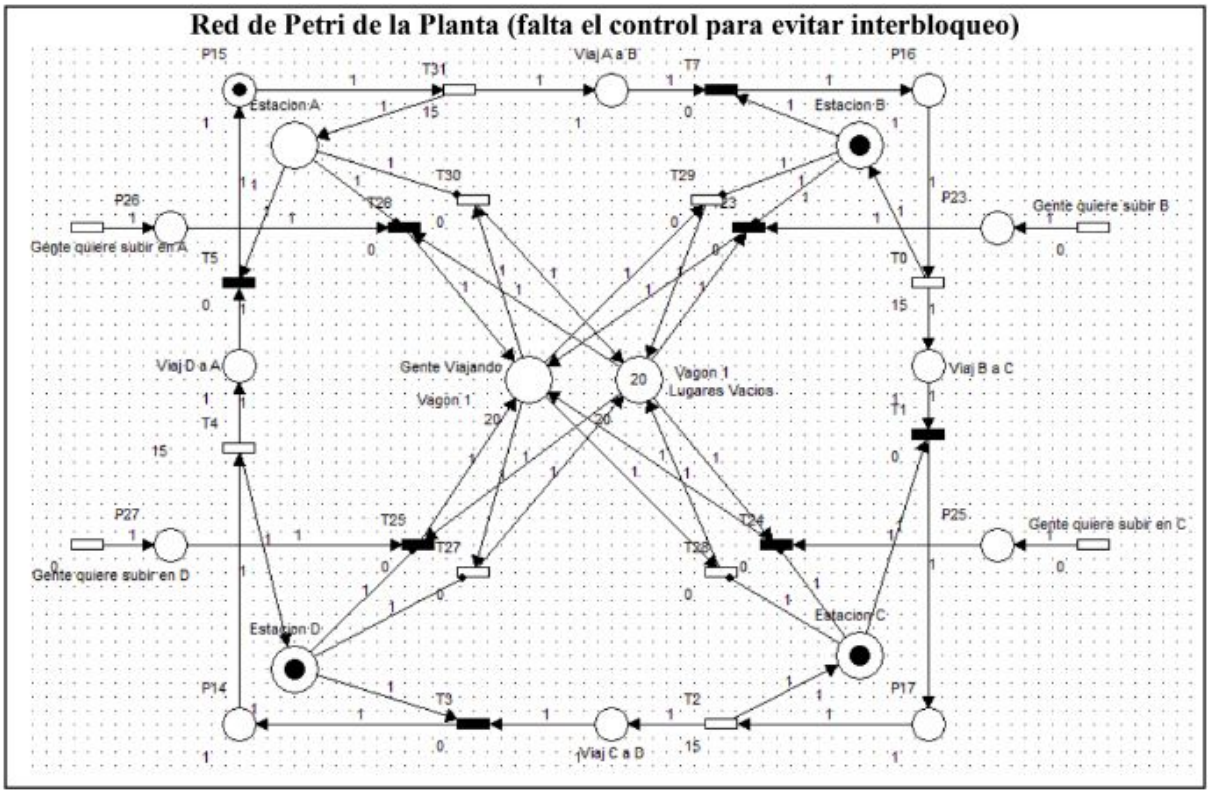
En este práctico se debe resolver el problema de control de un circuito ferroviario. Como dato se propone la red de Petri que modela una planta con 4 estaciones, un vagón, sin barreras. La red debe ser modificada con el fin de modelar la planta requerida y evitar interbloqueos. Luego simular la solución en un proyecto desarrollado con la herramienta adecuada (explique porque eligió la herramienta usada).

La planta requerida está formada por 4 estaciones (Estación A, Estación B, Estación C y Estación D), una máquina y un vagón. La capacidad de la máquina es de 30 pasajeros, mientras que la capacidad del vagón es de 20 pasajeros. En cada estación los pasajeros pueden subir o bajar; no pudiendo descender en cada estación los pasajeros que han ascendido en esa (no es necesario identificar los pasajeros, solo número).

Los tramos de unión entre las estaciones A y B y las estaciones C y D tienen un paso a nivel. En este paso a nivel se debe controlar la barrera para el paso de los vehículos y el tren. La barrera debe bajar 30 metros antes que llegue el tren a paso nivel y subir después de 20 metros que el tren a atravesado el paso a nivel.

El tren debe detenerse en cada estación no menos de 10 segundos y debe arrancar una vez que hayan subido todos los pasajeros o no haya lugar en máquina ni vagón.

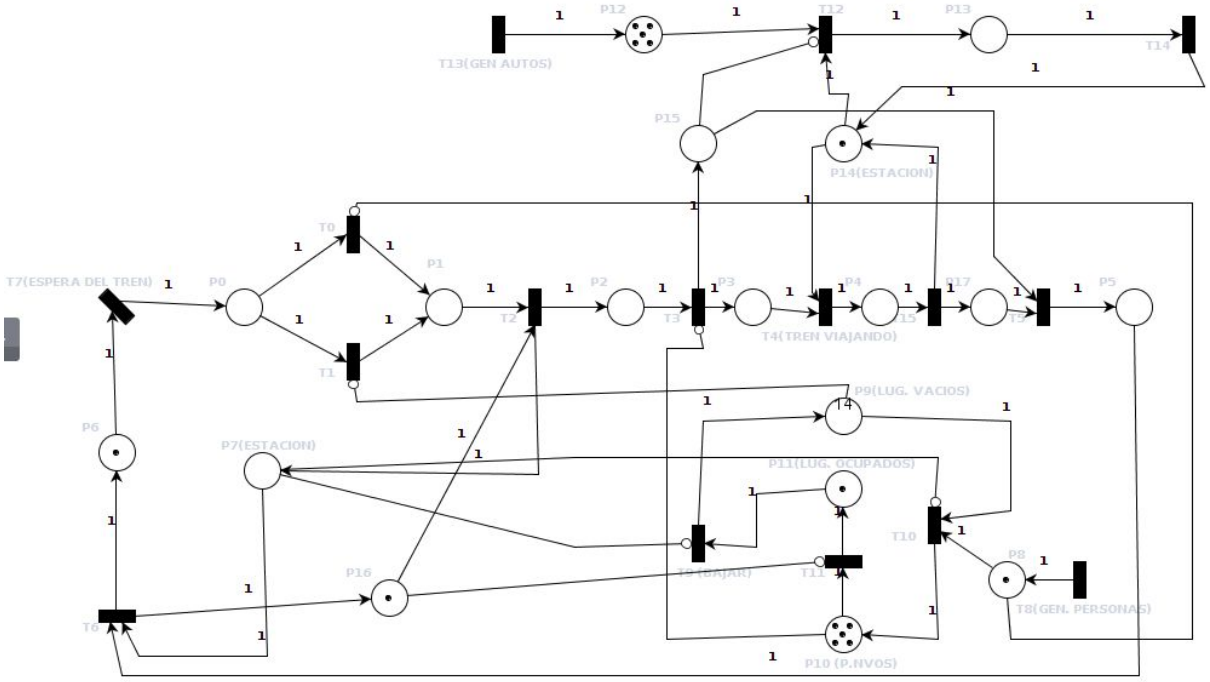
Originalmente, se nos dio la siguiente red para trabajar (bloqueada):



DESARROLLO

1. Red que modela el problema

El primer paso fue desbloquear la red que se nos dio como problema a resolver. Al ésta muy extensa, y por las características de la misma, se modeló el problema a una sola estación con el paso nivel y el generador de personas y autos. Esto es posible debido a la linealidad de la red, es decir, las cuatro estaciones tienen comportamiento exactamente idéntico por lo que modelando una sola, se consideran modeladas las otras tres. La red resultante quedó como la que sigue:



Con respecto a las condiciones impuestas a la salida del tren (debe arrancar una vez que hayan subido todos los pasajeros o no haya lugar en máquina ni vagón), se modelaron dos transiciones (T0 y T1), cada una de ellas inhibidas por los lugares disponibles en el vagón y subida de todos los pasajeros nuevos (plazas P9 y P8 respectivamente), que se encontrarán sensibilizadas cuando se hayan cumplido los 10 segundos de espera del tren en la estación y cuando no queden lugares vacíos (sensibilización de T1) o bien cuando no haya más personas por subir (sensibilización de T0), una vez que ambas transiciones puedan dispararse, consideramos al tren capaz de salir de la estación (modelado a través de la transición T2).

2. Tablas de estados y eventos

Tabla de estados	
P0	Tren cumplio el tiempo de espera
P1	Tren esperando que cierre la barrera
P2	Tren esperando que suban todos los pasajeros
P3	Tren viajando
P4	Tren ocupando paso nivel
P5	Tren viajando
P6	Tren en la estación
P7	Estación disponible
P8	Personas que quieren subir
P9	Lugares vacíos
P10	Personas autorizadas para subir
P11	Lugares ocupados
P12	Autos esperando paso nivel
P13	Autos ocupando el paso nivel
P14	Paso nivel
P15	Barrera de paso nivel
P16	Puerta del tren
P17	Tren viajando

Tabla de eventos	
T0	Verificación de existencia de nuevos pasajeros
T1	Verificación de ausencia de lugares vacíos
T2	Cierra entrada de estación
T3	Tren sale de estación
T4	Tren entra al paso nivel
T5	Tren se aleja del paso nivel
T6	Tren ingresa a la estación
T7	Tren cumple con tiempo de espera
T8	Personas llegando a la estación
T9	Pasajeros descendiendo del tren
T10	Pasajero compra boleto de entrada al tren
T11	Pasajero sube al tren
T12	Auto ingresa al paso nivel
T13	Auto llega al paso nivel
T14	Auto se va del paso nivel
T15	Tren sale del paso nivel

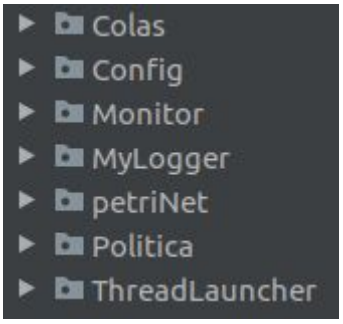
3. Cantidad de hilos

La cantidad de hilos para modelar el sistema puede variar. Se puede tener un hilo para la generación de autos, otro para modelar el recorrido del auto, otro hilo para manejar la subida/bajada de barrera del paso nivel, otro hilo para modelar el recorrido del tren, otros dos que modelen las condiciones de salida del tren de la estación (por lugares vacíos o personas pendientes para subir), otro que genere los pasajeros nuevos y otro que controle la subida/bajada de pasajeros. Si adoptamos este enfoque, pudimos identificar situaciones en las que el sistema puede llegar a no iniciar según el marcado inicial. Lo que inserta un error que no implica que el sistema esté mal modelado, si no que se da debido a la condición inicial del sistema. Es por esto que la cantidad de hilos elegida para modelar el sistema, es el de un hilo por transición independizándonos del problema antes mencionado debido a que cada hilo maneja la secuencia de una transición solamente.
Finalmente, se tienen 16 transiciones, por lo que se tienen 16 hilos.

4. Diagrama de clases

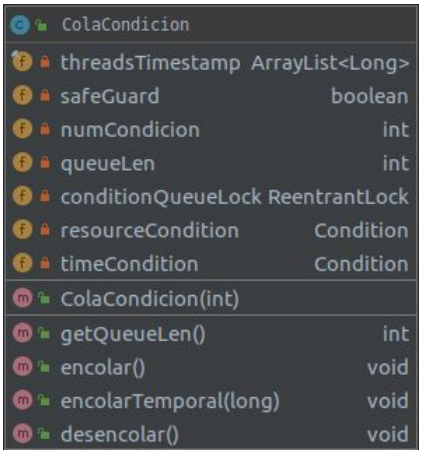
El diagrama de clases general se adjunta como Anexo I al final del presente informe. Los paquetes y clases creadas para el modelado del proyecto son las siguientes:

Esquema de paquetes:

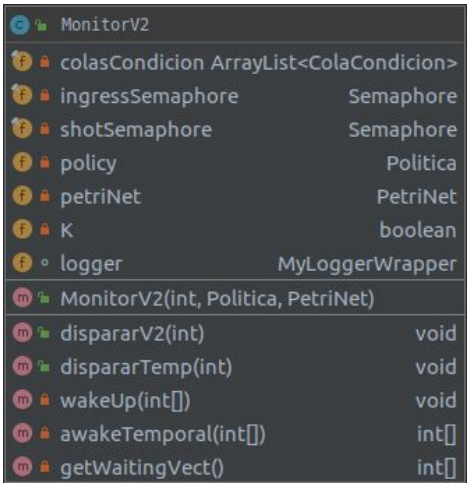


Paquetes y sus clases:

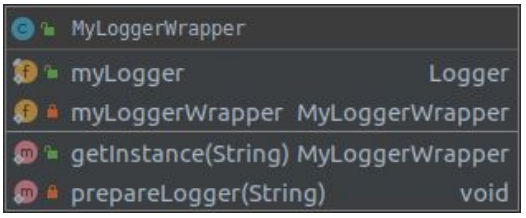
Paquete Colas:
Clase ColaCondición: Clase encargada de instanciar las colas de condición para el monitor.



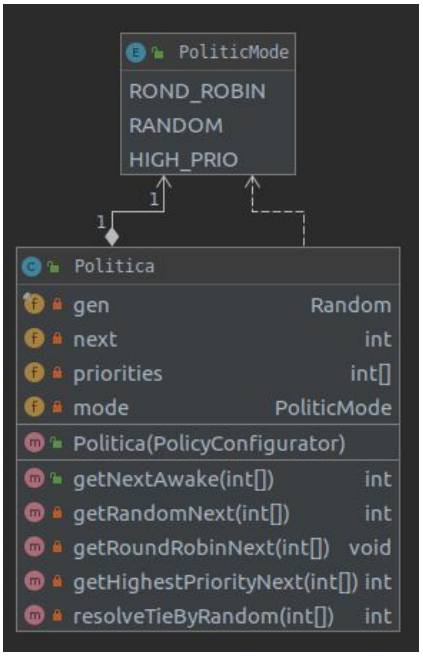
Paquete Monitor:
Clase MonitorV2: Clase que centraliza toda la lógica del monitor.



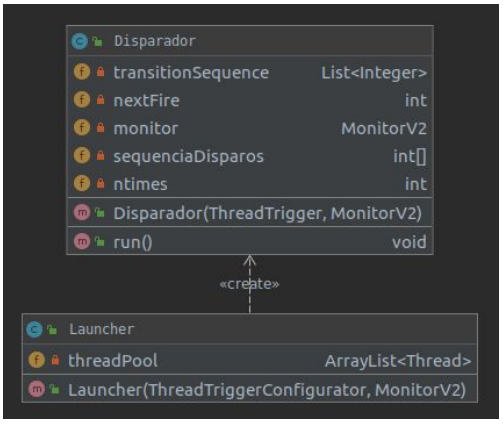
Paquete MyLogger:
Clase MyLoggerWrapper: Clase encargada del logueo de eventos del sistema. Está diseñada con el patrón Singleton.



Paquete Politica:
Clase Politica: Clase que implementa los métodos de selección de disparo en base a la política pasada como parámetro en el archivo JSON.
Clase PoliticMode: Clase tipo enum, contiene los diferentes tipos de política implementados en el sistema.



Paquete ThreadLauncher:
Clase Disparador: Clase que recibe la secuencia de disparos y la cantidad de veces a disparar dicha secuencia e implementa el método run() de los thread.
Clase Launcher: Clase encargada de crear el pool de threads para el sistema.



Paquete Config:

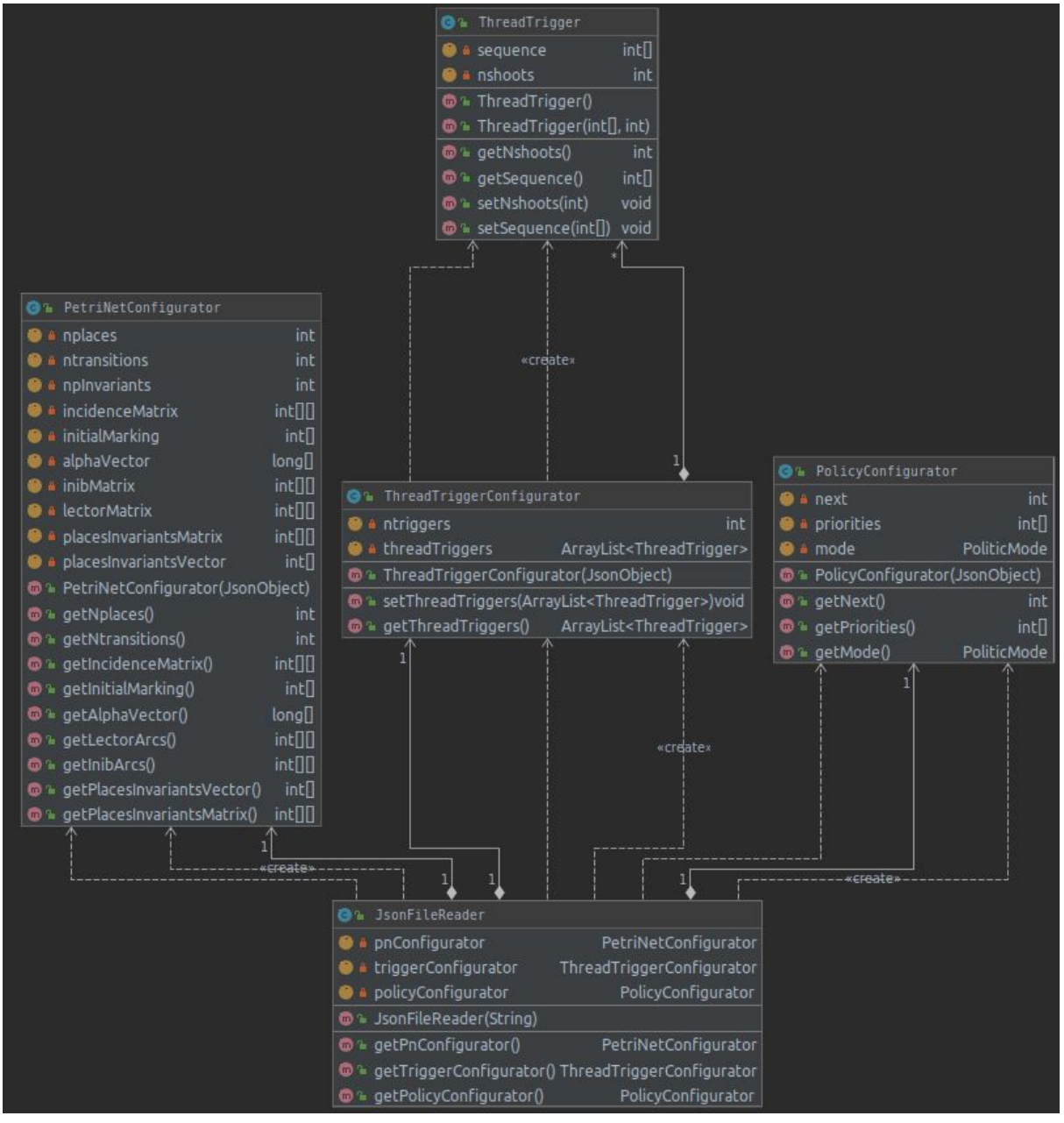
Clase JsonFileReader: Clase encargada de leer un archivo en formato json con el objetivo de parametrizar las clases PetriNet, Politica y Disparador (para los threads).

Clase PetriNetConfigurator: Clase encargada de transformar los parámetros leídos por el JsonFileReader a formatos con los que podamos trabajar en Java. Es la clase encargada de configurar a la Red de Petri.

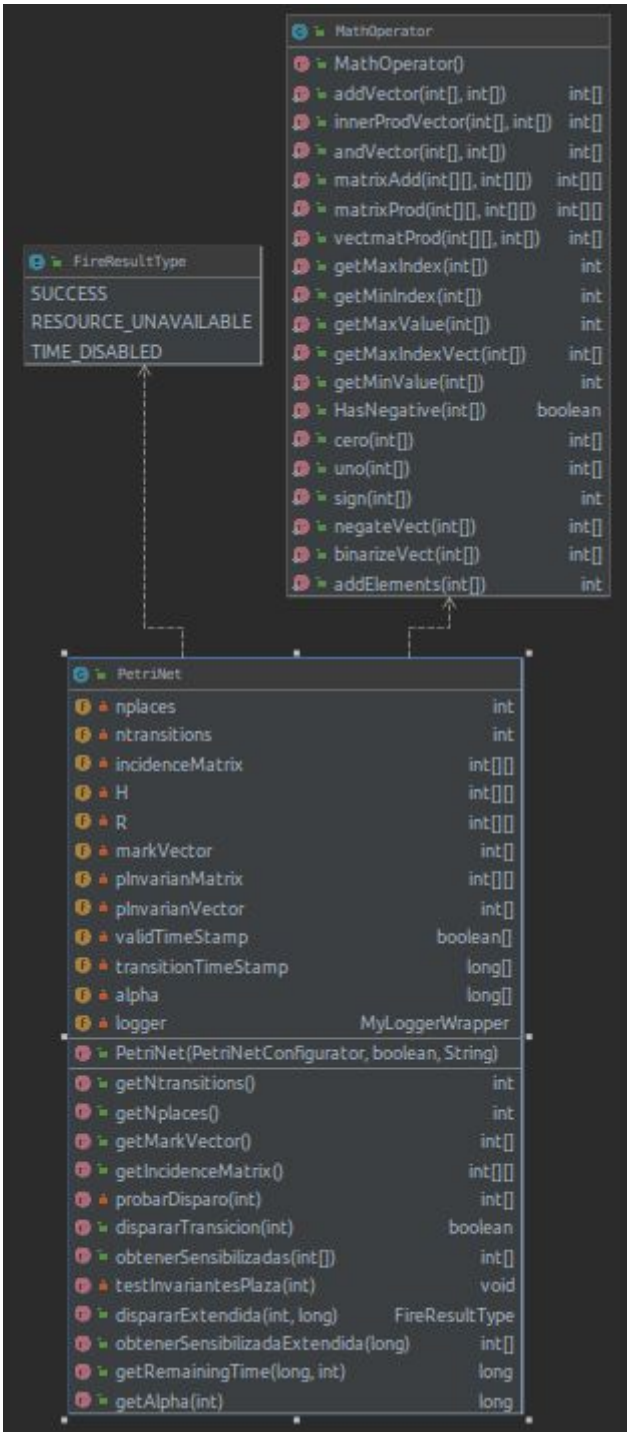
Clase PolicyConfigurator: Clase encargada de transformar los parámetros leídos por el JsonFileReader a formatos con los que podamos trabajar en Java. Es la clase encargada de configurar a la clase Política.

Clase ThreadTriggerConfigurator: Clase encargada de transformar los parámetros leídos por el JsonFileReader a formatos con los que podamos trabajar en Java. Es la clase encargada de configurar a la clase ThreadTrigger.

Clase ThreadTrigger: Clase cuya instancia almacena la secuencia de disparos y la cantidad de veces a disparar dicha secuencia para un hilo. Habrá tantas instancias de ésta como hilos existan en el sistema.

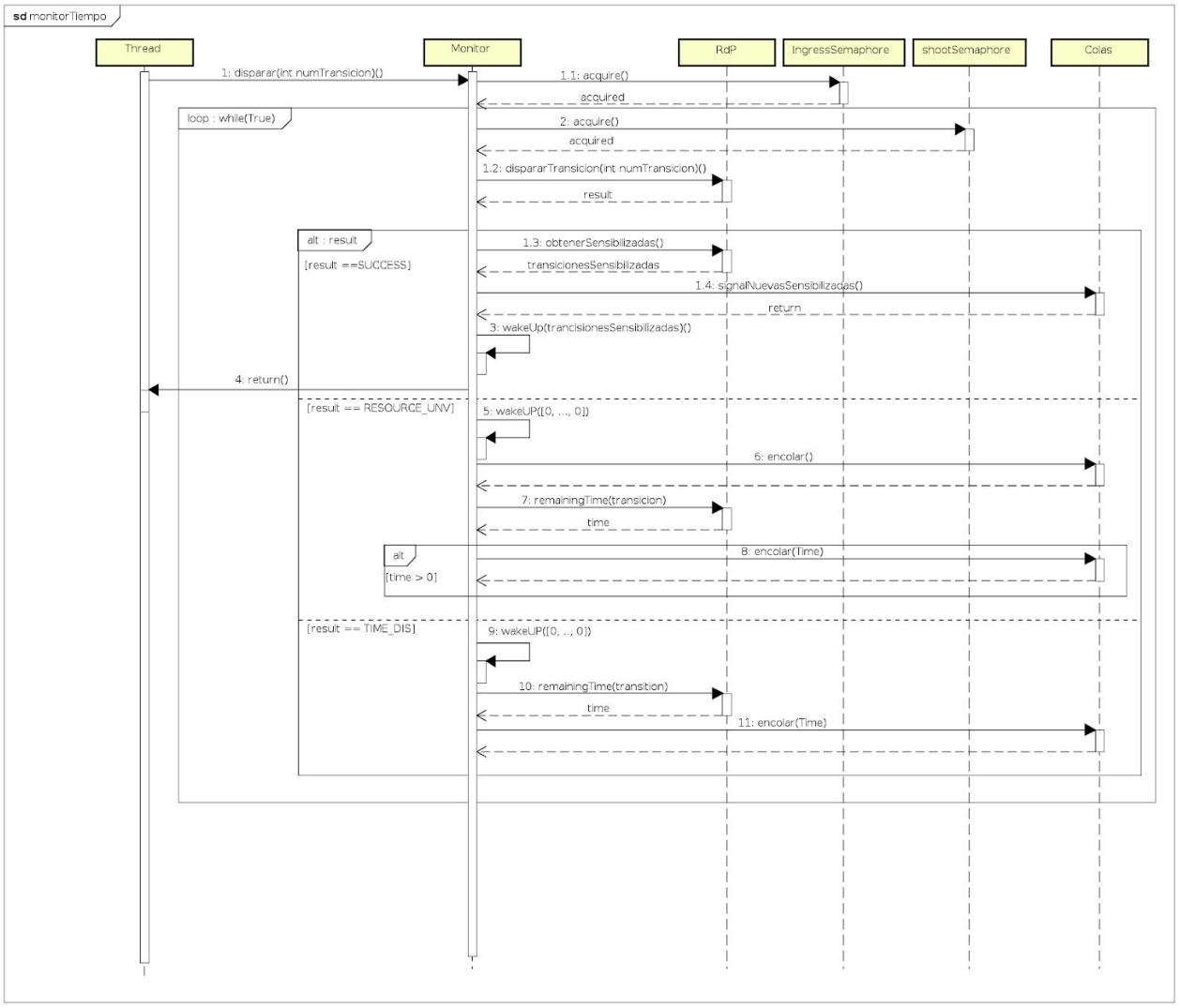


Paquete petriNet:
Clase FireResultType: Clase tipo enum, que devuelve el resultado del disparo en base a lo calculado en la Rdp.
Clase MathOperator: Clase estática que centraliza todas las operaciones matemáticas necesarias para el sistema.
Clase PetriNet: Clase que centraliza la lógica de la Red de Petri.

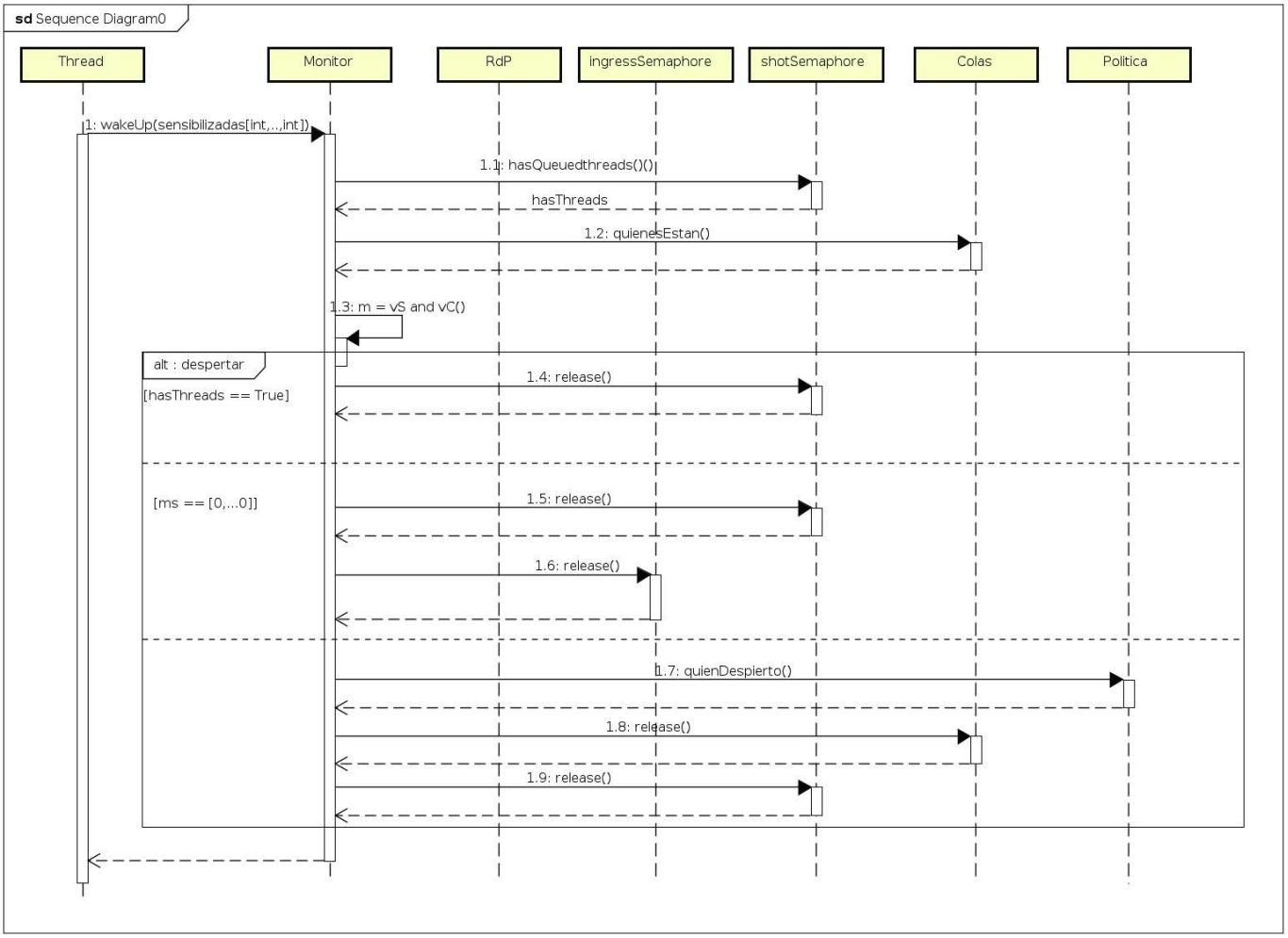


5. Diagramas de secuencia

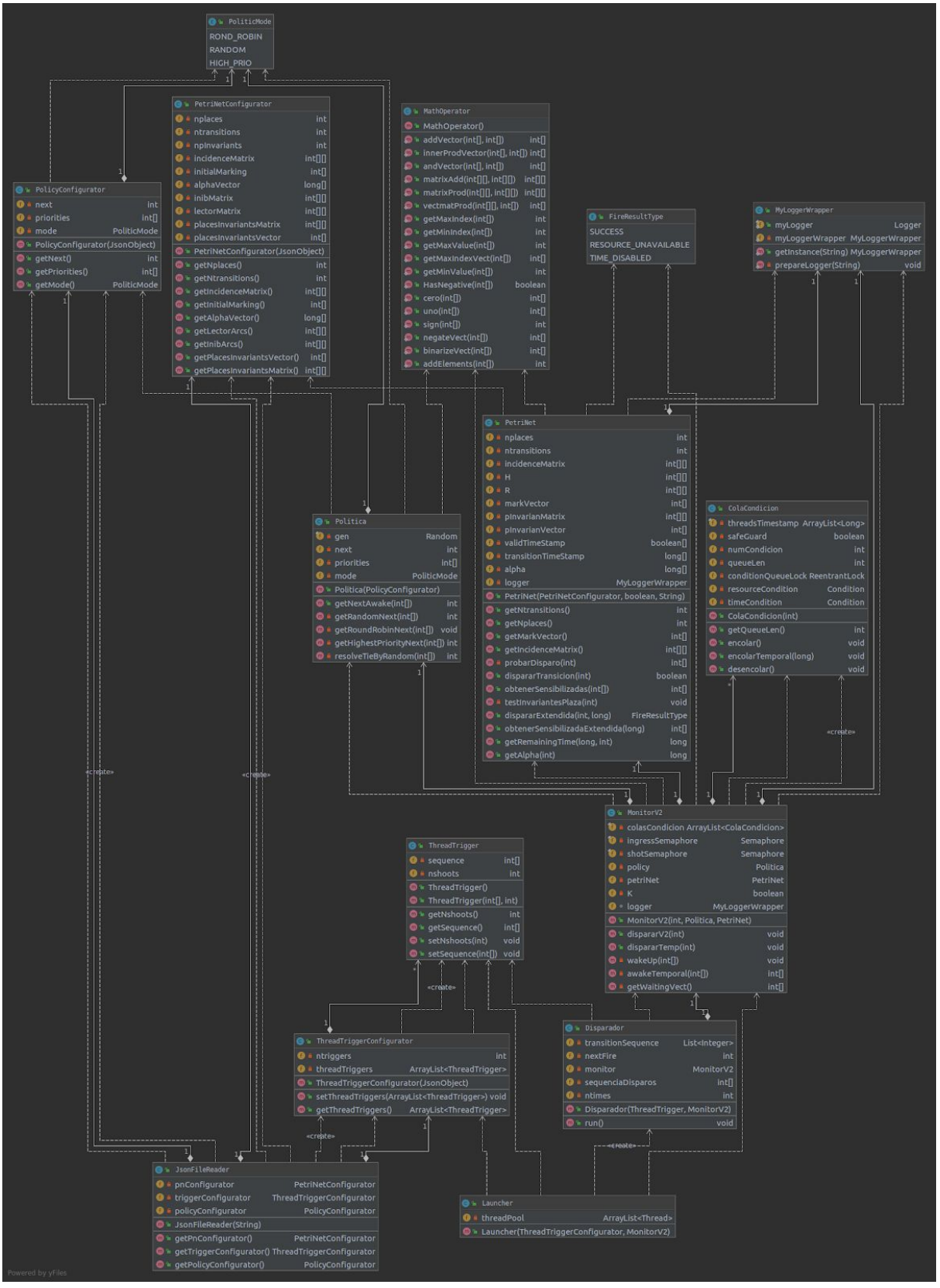
El diagrama de secuencia del monitor con tiempo es el siguiente:



Como el diagrama de secuencia se hace muy extenso, en la imagen anterior, el método wakeUp (encargado de despertar los hilos) fue modelado en un diagrama de secuencia a parte. Éste es:



ANEXO


PolicticMode
ROND_ROBIN
RANDOM
HIGH_PRIO
PetriNetConfigurator
nplaces int
ntransitions int
npInvariants int
incidenceMatrix int[][]
initialMarking int[]
alphaVector long[]
inibMatrix int[][]
lectorMatrix int[][]
placesInvariantsMatrix int[][]
placesInvariantsVector int[]
PetriNetConfigurator(JsonObject)
getNplaces() int
getNtransitions() int
getIncidenceMatrix() int[][]
getInitialMarking() int[]
getAlphaVector() long[]
getLectorArcs() int[][]
getInibArcs() int[][]
getPlacesInvariantsVector() int[]
getPlacesInvariantsMatrix() int[][]
MathOperator
MathOperator()
addVector(int[], int[]) int[]
innerProdVector(int[], int[]) int[]
andVector(int[], int[]) int[]
matrixAdd(int[][], int[][]) int[][]
matrixProd(int[][], int[][]) int[][]
vectmatProd(int[][], int[]) int[]
getMaxIndex(int[]) int
getMinIndex(int[]) int
getMaxValue(int[]) int
getMaxIndexVect(int[]) int[]
getMinValue(int[]) int
HasNegative(int[]) boolean
cero(int[]) int[]
uno(int[]) int[]
sign(int[]) int
negateVect(int[]) int[]
binarizeVect(int[]) int[]
addElements(int[]) int
PolicyConfigurator
next int
priorities int[]
mode PolicticMode
PolicyConfigurator(JsonObject)
getNext() int
getPriorities() int[]
getMode() PolicticMode
FireResultType
SUCCESS
RESOURCE_UNAVAILABLE
TIME_DISABLED
MyLoggerWrapper
myLogger Logger
myLoggerWrapper MyLoggerWrapper
getInstance(String) MyLoggerWrapper
prepareLogger(String) void
PetriNet
nplaces int
ntransitions int
incidenceMatrix int[][]
H int[][]
R int[][]
markVector int[]
pInvarianMatrix int[][]
pInvarianVector int[]
validTimeStamp boolean[]
transitionTimeStamp long[]
alpha long[]
logger MyLoggerWrapper
PetriNet(PetriNetConfigurator, boolean, String)
getNtransitions() int
getNplaces() int
getMarkVector() int[]
getIncidenceMatrix() int[][]
probarDisparo(int) int[]
dispararTransicion(int) boolean
obtenerSensibilizadas(int[]) int[]
testInvariantesPlaza(int) void
dispararExtendida(int, long) FireResultType
obtenerSensibilizadaExtendida(long) int[]
getRemainingTime(long, int) long
getAlpha(int) long
Politica
gen Random
next int
priorities int[]
mode PolicticMode
Politica(PolicyConfigurator)
getNextAwake(int[]) int
getRandomNext(int[]) int
getRoundRobinNext(int[]) void
getHighestPriorityNext(int[]) int
resolveTieByRandom(int[]) int
ColaCondicion
threadsTimestamp ArrayList<Long>
safeGuard boolean
numCondicion int
queueLen int
conditionQueueLock ReentrantLock
resourceCondition Condition
timeCondition Condition
ColaCondicion(int)
getQueueLen() int
encolar() void
encolarTemporal(long) void
desencolar() void
MonitorV2
colasCondicion ArrayList<ColaCondicion>
ingressSemaphore Semaphore
shotSemaphore Semaphore
policy Politica
petriNet PetriNet
K boolean
logger MyLoggerWrapper
MonitorV2(int, Politica, PetriNet)
dispararV2(int) void
dispararTemp(int) void
wakeUp(int[]) void
awakeTemporal(int[]) int[]
getWaitingVect() int[]
ThreadTrigger
sequence int[]
nshoots int
ThreadTrigger()
ThreadTrigger(int[], int)
getNshoots() int
getSequence() int[]
setNshoots(int) void
setSequence(int[]) void
ThreadTriggerConfigurator
ntriggers int
threadTriggers ArrayList<ThreadTrigger>
ThreadTriggerConfigurator(JsonObject)
setThreadTriggers(ArrayList<ThreadTrigger>) void
getThreadTriggers() ArrayList<ThreadTrigger>
Disparador
transitionSequence List<Integer>
nextFire int
monitor MonitorV2
sequenciaDisparos int[]
ntimes int
Disparador(ThreadTrigger, MonitorV2)
run() void
JsonFileReader
pnConfigurator PetriNetConfigurator
triggerConfigurator ThreadTriggerConfigurator
policyConfigurator PolicyConfigurator
JsonFileReader(String)
getPnConfigurator() PetriNetConfigurator
getTriggerConfigurator() ThreadTriggerConfigurator
getPolicyConfigurator() PolicyConfigurator
Launcher
threadPool ArrayList<Thread>
Launcher(ThreadTriggerConfigurator, MonitorV2)
«create»
Powered by yFiles