Framework para Sincronización de Eventos, Gestión de Recursos y Manejo de Concurrencia Conducido por Red de Petri

Proyecto integrador - Ingeniería en Computación

Ariel Iván Rabinovich
Matrícula: 36771223
Teléfono: 0351-6108740
airabinovich@gmail.com

Juan José Arce Giacobbe
Matrícula: 37194512
Teléfono: 0351-6170901
juanjo.arce7456@gmail.com

Director
Dr. Orlando Micolini
omicolini@compuar.com

Codirector
Ing. Luis Orlando Ventre
lventre@gmail.com

Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales
Laboratorio de Arquitectura de Computadoras

Agradecimientos

Este trabajo integrador simboliza la culminación de nuestros estudios de grado, lo que es un gran logro para nosotros. Existen personas sin las cuales este logro no habría sido posible y queremos expresar nuestro agradecimiento con ellas.

Juan

A mi familia y amigos.
A mis profesores, de todos los niveles académicos.
A las casualidades y eventos aleatorios que de alguna forma hayan ayudado.

Ariel

A mis padres por darme la oportunidad de estudiar una carrera universitaria.
A Ruth por toda la ayuda en mis primeros pasos en la carrera.
A Orlando por toda la dedicación a la realización de este proyecto.
A Luis y Augusto por las horas dedicadas a discusiones y revisiones.
A Nico por tantos consejos y discusiones, siempre con un vaso de Coca Cola helada.

Índice general

Capítulos 1, 2, 3, 6 y 9

Índice de figuras

Índice de cuadros

Parte I

Introducción y Objetivos

Capítulo 1

Introducción

1.1. Introducción

En la actualidad, existe un crecimiento en la complejidad de los sistemas reactivos debido a un constante aumento en la interacción entre los procesos y su entorno. Dichas interacciones se representan como eventos de software asincrónicos respecto a la ejecución de dichos sistemas. Estos eventos desencadenan cambios en el estado global de los sistemas, que a su vez, acarrean problemas de concurrencia. El manejo del asincronismo y la concurrencia de estos sistemas sin las herramientas adecuadas es una tarea por demás compleja, propensa a la generación de errores de ejecución si no se realiza con especial precaución. En general, los sistemas más interactivos son: los sistemas embebidos, las interfaces gráficas, los servicios en la nube y los dispositivos IoT. Todos estos sistemas entran en la clasificación de aplicaciones dirigidas por eventos o event-driven applications. [AF15] Los mismos se presentan en cantidades masivas y crecientes en la actualidad. Este proyecto integrador pretende diseñar e implementar un framework para el desarrollo de aplicaciones, utilizando redes de Petri como mecanismo de modelado y ejecución para procesar eventos y representar los estados globales y locales del sistema. Es decir, se pretende utilizar una Red de Petri como modelo de la lógica del sistema y de sus interacciones con el medio.

En el marco del Laboratorio de Arquitecturas de Computadoras se han realizado numerosos proyectos siguiendo esta línea de trabajo. Se destacan: "Implementación de un sistema multicore heterogéneo embebido con procesador de Petri sobre FPGA" [GP12], "Desarrollo de un IP core con procesamiento de Redes de Petri Temporales para sistemas multicore en FPGA" [NPM12], "Reducción de recursos en un procesador de redes de Petri implementado en un IP Core" [BAM13], "Modularización del Procesador de Petri y Optimización para Sistemas Embebidos" [Dan15] y "Estudio e Implementación de un Caso Testigo para el Desarrollo de Sistemas Embebidos, Críticos y Reactivos" [BL17]. Por otro lado y dentro del mismo marco, dos trabajos sentaron las bases de este proyecto integrador. En primer lugar "Generación de Código de Sistemas Reales, Paralelos y Concurrentes a partir de Redes de Petri Orientadas a Procesos" [CF14] propone un esquema básico de comunicación entre el software y la red de Petri mediante colas de entrada, de salida y etiquetas.También permite generar código estático a partir de una red de Petri y una configuración. En segundo lugar, "Desarrollo de un Framework para Aplicar el Paradigma de Programación Reactiva Utilizando Redes de Petri como Procesador de Eventos" [AF15] donde se introduce la idea de un framework para el desarrollo de sistemas reactivos dirigido por RdP. El objetivo de este trabajo es diseñar e implementar un nuevo framework. Se tienen en cuenta ciertos aspectos de diseño desarrollados en [AF15], pero a su vez se realiza un amplio trabajo de rediseño. Los principales puntos a destacar del nuevo diseño son: el desacoplamiento entre la red de petri (RdP) y la aplicación de usuario, la modificación del modo de sincronización utilizado para las comunicaciones con la RdP, el énfasis en mantener la inversión de control, la no imposición de restricciones al usuario desarrollador sobre las herramientas propias del lenguaje, la expansión de funcionalidades del monitor de RdP para adoptar la ecuación de estado generalizada descrita en [DIOM16] y la simplificación de las interfaces de programación ofrecidas al usuario desarrollador. Debido a los cambios en el diseño mencionados se realiza una re-implementación total del código del framework. A lo largo de este informe se presenta el proceso de diseño e implementación del framework en su totalidad, se brindan las bases teóricas necesarias para la comprensión del mismo y se ejemplifica con casos de uso en aquellos puntos donde los autores lo consideran necesario.

Capítulo 2

Objetivos

2.1. Objetivo General

El objetivo general de este proyecto integrador es diseñar e implementar un framework que permita desarrollar sistemas reactivos utilizando modelos basados en Redes de Petri no autónomas. El framework resultante debe aislar el control de la aplicación en un módulo que ejecute a la RdP. Este módulo debe manejar el asincronismo del sistema de forma transparente al código de software de la aplicación, delegando las decisiones a la RdP, tanto de la lógica del sistema como de la política.

2.2. Objetivos Secundarios

A continuación se mencionan los objetivos secundarios de este proyecto:

- Separar la lógica del sistema del código que implementa sus funcionalidades.
- Estudiar Redes de Petri ordinarias, orientadas a procesos y no autónomas.
- Investigar implementaciones en Proyectos Integradores previos y analizar la reutilización o reimplementación del código existente.
- Obtener un software capaz de ejecutar RdP utilizando la ecuación de estado generalizada descrita en [DIOM16].
- Implementar la inversión de control del framework.
- Priorizar la mantenibilidad del código generado respetando estándares y estilos de programación.
- Generar tests automáticos para garantizar el correcto funcionamiento de las funcionalidades del software generado
- Documentar detalladamente el código fuente siguiendo una metodología estándar.
- Ofrecer interfaces de programación sencillas.
- Resolver problemas conocidos de programación concurrente para poner a prueba el framework.
- Resolver un problema real de concurrencia utilizando el framework desarrollado.
- Documentar el proceso de desarrollo de aplicaciones particulares utilizando el framework.
- Ofrecer ejemplos de uso del framework para facilitar la asimilación de los usuarios.
- Ofrecer el software resultante de forma pública y accesible para cualquier usuario en cualquier parte del mundo.

2.3. Análisis Previo

Antes de la definición del objetivo general se realizó un trabajo de investigación y comparación para determinar el tipo de herramienta a desarrollar (Biblioteca de software, API, Framework, Generación de código, etc). (ver capítulo 7)

Parte II

Marco Teórico

Introducción

En esta parte del presente informe se expone el marco teórico necesario para la comprensión de los temas tratados en este proyecto integrador.

En el capítulo 3 se introduce al lector a *Autómatas* o *Máquinas de Estado Finitas (FSM)* y se generaliza este concepto para dar lugar a *Redes de Petri*. Se formaliza la definición de RdP y la semántica del disparo de transiciones. Luego, se extiende esta semántica hasta llegar a la ecuación de estado desarrollada en [DIOM16]. Finalmente se presenta una comparación entre FSM y RdP.

En el capítulo 4 se desarrollan paradigmas y herramientas de la programación necesarias para entender los detalles del diseño detallado en las próximas secciones. Se expone el paradigma *Dataflow* y el *Reactivo*, luego se explican los conceptos de la *Programación Orientada a Aspectos (AOP)* y finalmente se presenta el concepto y utilización de la *Reflexión*.

En el capítulo 5 se comienza desarrollando el concepto de un generador automático de código fuente para luego presentar *Frameworks*. Más adelante se brinda una comparación entre estos y la generación automática de código.

Finalmente, en el capítulo 6 se introduce al lector en el concepto de *Concurrencia* junto con sus propiedades y problemas. A continuación se explican los distintos mecanismos de sincronización entre hilos y procesos. Luego, se desarrollan *Semáforos* y *Monitores* como herramientas de sincronización, haciendo especial foco en Monitores.

Capítulo 3

Modelos

3.1. Autómatas o Máquinas de Estado

Existen muchas formas de modelar el comportamiento de los sistemas, y el uso de máquinas de estado finitas es una de las más antiguas y más conocidas. Las máquinas de estado finitas o autómatas nos permiten pensar acerca del "estado" de un sistema en un instante en particular y caracterizar el comportamiento de dicho sistema basado en ese estado. El uso de esta técnica de modelado no está limitada al desarrollo de sistemas de software. [Wri05]

3.1.1. Definición Conceptual de Máquina de Estado

Si una máquina de estados M, en un instante dado, se encuentra en el estado E_0 y ocurre un evento e_0 que lleva a M al estado E_1, se dice que ocurrió una *transición* del estado E_0 al estado E_1. A partir de esto se puede deducir que M no puede estar en E_0 y E_1 a la vez, y por lo tanto los estados de una máquina de estados, son **estados globales** del sistema modelado.

Analizando la semántica de las máquinas de estado, se pueden identificar algunas características clave de un sistema que puede ser modelado con máquinas de estados finitas:

- El sistema debe ser descripto por conjunto finito de estados.
- El sistema debe tener una cantidad finita de entradas y/o eventos que puedan disparar transiciones entre estados.
- El comportamiento del sistema en un instante dado depende del estado actual y de sus entradas o eventos que ocurran en ese instante.
- Para cada estado posible en que el sistema pueda encontrarse existe un comportamiento definido para cada posible entrada o evento.
- El sistema tiene un estado inicial único y definido.

[Wri05]

3.1.2. Definición Formal de Máquina de Estado

A fin de eliminar la ambigüedad existente en una definición conceptual, se introduce una definición formal de Autómata Finito:

Definición: Un autómata finito M está definido por una tupla $(\Sigma, Q, q_0, F, \sigma)$, donde:

- Σ es el conjunto de símbolos de entrada de M
- Q es el conjunto de estados de M
- q_0 es el estado inicial de M
- $F \subseteq Q$ es el conjunto de estados finales de M
- $\sigma : Q \times \Sigma \to Q$ es la función de transición

[Wri05]

3.2. Redes de Petri

Tomando el concepto de transición en una máquina de estados, se lo puede extender a una entidad propia. Esta transitión t_i será denotada por una barra, un rectángulo o un cuadrado, y puede tener múltiples arcos de entrada (entrantes) y de salida (salientes) a la vez [Pet09].

De la misma forma que en una máquina de estados los círculos denotan estados del sistema, en una RdP se utilizan círculos para denotar las *plazas* o *lugares* de la red. Estas plazas no representan estados globales, sino **estados locales**. [Pet09] El estado de una plaza está dado por la cantidad de marcas o *tokens* que esta contiene.

Las *plazas* y las *transiciones* de una RdP se conectan entre sí mediante *arcos* dirigidos, pudiéndose unir una plaza únicamente con cero o más transiciones y viceversa. La unión entre plazas o entre transiciones no respeta la estructura del modelo.

Como consecuencia de esto, una RdP puede ser representada por un grafo bipartito, donde los nodos pertenecen a uno de dos conjuntos (*plazas* o *transiciones*).

En la figura 3.1 se pueden visualizar las partes de una Red de Petri.

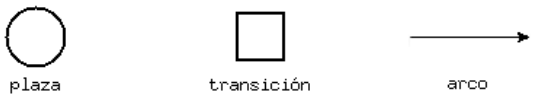


Figura 3.1: Partes de una Red de Petri

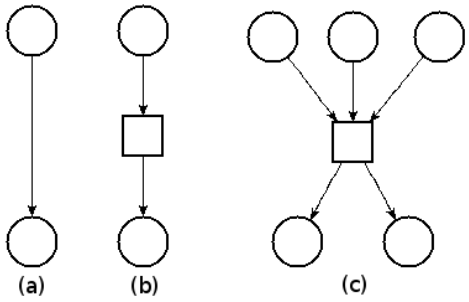


Figura 3.2: Equivalencia entre una Máquina de Estados y una Red de Petri

En la figura 3.2 se aprecia:

(a) Una máquina de estados de dos estados y una transición.

(b) Una RdP equivalente a la máquina de (a).

(c) Una RdP con una transición con múltiples arcos de entrada y de salida.

Se puede extraer como consecuencia directa de esta extensión de la semántica de un autómata que en una Red de Petri:

- Múltiples tokens pueden existir en el modelo al mismo tiempo, y particularmente en una plaza.
- No existe un estado global explícito.
- El estado global del sistema es el conjunto de todos los estados parciales, representados por las plazas y sus tokens. A este conjunto se lo denomina el **marcado** de la red.

3.2.1. Definición Formal de Red de Petri

A fin de eliminar ambigüedades, se presenta una serie de definiciones sobre Redes de Petri.

- Definición 1: Una Red de Petri R está definida por la tupla $(P, T, Pre, Post)$ donde:
  - $P = \{p_1, p_2, \ldots, p_p\}$ un conjunto de plazas.[1]
  - $T = \{t_1, t_2, \ldots, t_t\}$ un conjunto de transiciones, donde $P \cap T = \emptyset$. [2]
  - $Pre : P \times T \to \mathbb{N}^p$ aplicación de precedencia.[3]
  - $Post : P \times T \to \mathbb{N}^p$ aplicación de incidencia.

  $Pre(p_i, t_j)$ contiene el peso del arco que va de p_i a t_j, y $Post(p_i, t_j)$ contiene el peso del arco que va de t_j a p_i

- Definición 2: Una Red de Petri Marcada está definida por el par (R, M), donde R es una RdP y $M : P \to \mathbb{N}^p$ (donde $|P| = p$) es una aplicación llamada *marcado*.
  $m(R)$, o más simplemente m si la red es conocida, define el marcado de la RdP y $m(p_i)$ o m_{p_i} indica el marcado de la plaza p_i, es decir, el número de tokens contenido en la plaza p_i.
  La marca inicial se denota m_0 y da la cantidad inicial de tokens en todas las plazas de la red, por lo que especifica el estado inicial del sistema.

[1] Se utiliza p como la cantidad de plazas de la RdP en todo momento dentro de este informe por simplicidad para el lector
[2] Se utiliza t como la cantidad de transiciones de la RdP en todo momento dentro de este informe por simplicidad para el lector
[3] Se toma la definición de números naturales incluyendo el cero por simplicidad de notación.

- Definición 3: Para una marca m, una transición t_j está sensibilizada, y por lo tanto es disparable, si y solo si:

$$\forall p_i \in P, m(p_i) \geq Pre(p_i, t_j)$$

Conceptualmente, una transición está sensibilizada si todas sus plazas de entrada contienen al menos la cantidad de tokens que indica el peso de los arcos que las unen.

En la figura 3.3 se observa gráficamente esta definición mediante dos casos de transiciones no sensibilizadas. Nótese el peso de los arcos.

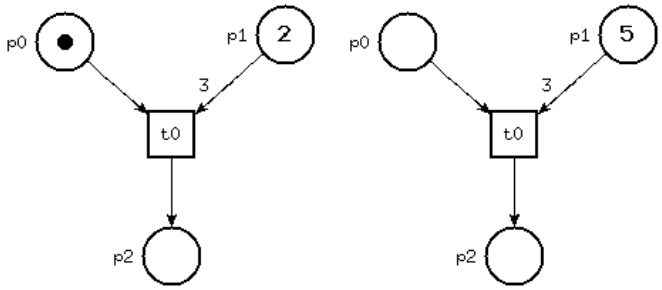


Figura 3.3: Ejemplos de transiciones no sensibilizadas.

- Definición 4: La estructura de una Red de Petri se denota $N = \{P, T, F, W\}$ donde,
  - P es en conjunto de plazas.
  - T es el conjunto de transiciones, donde se cumple que $P \cap T = \emptyset$
  - F es el conjunto de arcos, donde $F \subseteq (P \times T) \cup (F \times P)$.
  - W es la función de peso de los arcos.
- Definición 5: Conjunto de transición y plaza de entrada y de salida.

  El conjunto de las plazas de entrada a la transición t se denota $\bullet t$ y se define,

$$\bullet t = \{p \in P : (p, t) \in F\}$$

  El conjunto de las plazas de salida de la transición t se denota $t\bullet$ y se define,

$$t\bullet = \{p \in P : (t, p) \in F\}$$

  El conjunto de las transiciones de entrada a la plaza p se denota $\bullet p$ y se define,

$$\bullet p = \{t \in T : (t, p) \in F\}$$

  El conjunto de las transiciones de salida de la plaza p se denota $p\bullet$ y se define,

$$p\bullet = \{t \in T : (p, t) \in F\}$$

3.2.2. Disparo de una Transición

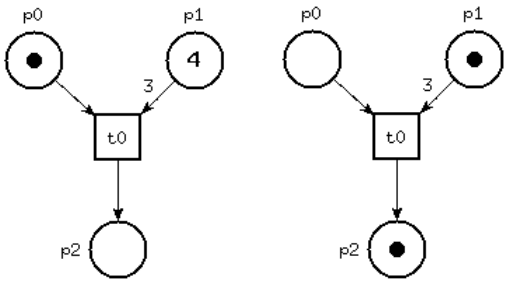
La condición de disparo relacionada a $Pre(p_i, t_j)$ significa que para todas las plazas p_i de entrada a t_j, es decir, todas las plazas que tienen arcos que apuntan hacia t_j, el número de tokens presentes debe ser mayor o igual al peso de dicho arco.

- Definición 6: En una RdP, dada una marca $m_n(p)$, cualquier transición t_j que se encuentre sensibilizada puede ser disparada, y su disparo lleva a una marca $m_{n+1}(p)$ dada por:

$$m_{n+1}(p) = m_n(p) + Post(p_i, t_j) - Pre(p_i, t_j), \forall p_i \in P$$

Como se indica en la ecuación, al disparar la transición t_j, se quitan tantos tokens de $\bullet t$ como indiquen los arcos que las unen a t_j, y se añaden a $t\bullet$ la cantidad de tokens que indiquen los arcos que unen a t_j con ellas.
El disparo de una transición t_j se denota $m_n \rightarrow t_j \rightarrow m_{n+1}$

En la figura 3.4 se observa el estado de una RdP antes y después del disparo de una transición.



(a) t_0 sensibilizada (b) Disparo de t_0

Figura 3.4: Disparo de una transición

- Definición 7: Matriz de Incidencia.
  La matriz de incidencia de una RdP se define como,

$$I = Post - Pre$$

  Notas:

  - El disparo de una transición t_j se reformula como,

$$m_{n+1}(p) = m_n(p) + I(p_i, t_j), \forall p_i \in P$$

  - A partir de las matrices Pre y $Post$ se puede reconstruir la estructura de la red, a partir de I no es posible.

3.2.3. Sucesión de Disparos

Si en lugar del disparo de una transición se requiere disparar múltiples transiciones, se puede reescribir la ecuación de cambio de estado de la red de la siguiente forma,

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

En esta ecuación, σ representa la sucesión de disparos a realizar. Se cumple $\sigma \in \mathbb{N}^t$ y el elemento σ_i contiene la cantidad de disparos a realizar sobre t_i.
Si se comienza a realizar la sucesión de disparos σ_i a partir del marcado inicial m_0 y todos los disparos son exitosos, se llega a un marcado m_i y se dice que m_i es *alcanzable*.
De la misma forma, si existe un marcado m_j alcanzable desde m_0, debe exitir una sucesión de disparos σ_j que permita alcanzarlo.

3.3. Extensión de la Semántica de las Redes de Petri

Las RdP descriptas anteriormente constituyen la versión más simple de este modelo, conocidas como *Redes de Petri Plaza-Transición* o *Redes de Petri Ordinarias*.

Se pueden realizar modificaciones sobre el modelo a fin de aumentar la semántica y permitir modelar mayor cantidad de sistemas del mundo real.

Algunas de las variantes introducidas a las RdP desde su aparición son:

- Semántica Temporal: Permiten modelar restricciones temporales sobre la acciones.
- Semántica Estocástica: Permiten modelar restricciones ligadas a variables aleatorias.
- Arcos Especiales: Tienen alguna característica que aumenta la expresividad de la red.
- Tokens Coloreados: Permite asociar un dato (color) a cada token y tomar decisiones a partir de su color.

[Pet09]

A continuación se detallan las extensiones que son de interés a este proyecto integrador.

3.3.1. Arcos Especiales

Durante el modelado de un sistema, en algunas ocasiones es necesario verificar condiciones más allá de la simple existencia de un recurso esperado. Cuando esto sucede, una RdP ordinaria no tiene la suficiente expresividad para modelar esta situación.

Por esto se introducen algunos tipos especiales de arcos que afectan a la sensibilización de la transición a la que apuntan.

3.3.1.1. Arcos Inhibidores

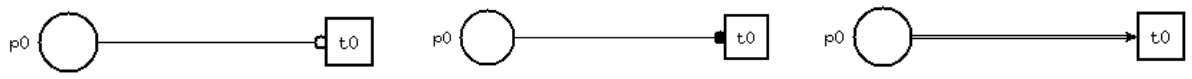
Un arco inhibidor conecta una plaza p_i con una transición t_j. Si $m_{p_i} > 0$, entonces t_j queda des-sensibilizada. Este tipo de arcos permite modelar prioridades y ausencia de ciertos recursos.

3.3.1.2. Arcos de Reset

Un arco de reset conecta una plaza p_i con una transición tj, habilitándola si $m_{p_i} > 0$. Cuando t_j es disparada, m_{p_i} pasa a valer cero.

3.3.1.3. Arcos Lectores

Un arco lector conecta una plaza p_i con una transición t_j y tiene un peso w. p_i sensibiliza a t_j sólo si $m_{p_i} > w$, de la misma manera que sucede con los arcos estándar, pero el disparo de t_j no modifica m_{p_i}.



(a) Arco Inhibidor (b) Arco Lector (c) Arco de Reset

Figura 3.5: Arcos Especiales

3.3.2. Guardas

Como se describió en la sección 3.2.1, una transición t_j se encuentra sensibilizada si

$$\forall p_i \in \bullet t_j, m(p_i) \geq Pre(p_i, t_j)$$

Esto es, la sensibilización de t_j depende únicamente del estado de la red para un instante dado.

Si se pretende relacionar la RdP al estado del medio (estado de un sensor, del programa que se ejecuta, etc), el modelo actual resulta no ser suficientemente expresivo. Por esta razón, se introduce el concepto de *guarda*.

Sea V un conjunto de *variables booleanas* con $v_i \in V$ y tal que $|V| \leq |T|$

Si se asocia v_i a t_j, se obtiene la guarda g_j que puede tomar el valor de v_i o de su complemento $!v_i$ o $\sim v_i$.

Cada variable v_i afecta al sensibilizado de todas las transiciones que la tengan asociada a través de cada guarda g_j, definiendo una nueva semántica de sensibilización. Esta se resuelve realizando una operación AND entre el estado de sensibilización de t_j y el estado de su guarda asociada g_j.

A fin de formalizar este concepto, se construye un vector booleano SG de dimensión $t = |T|$, que se obtiene a partir de la siguiente ecuación:

$$SG = (V \times RP) \vee (NV \times RN) \vee (NGT)$$

donde:

- $NV \in B^{1\times|V|}/nv_i = \neg v_i$, el vector de variables v_i negadas.
- $RP \in B^{|V|\times|T|}/rp_{i,j} = True \Leftrightarrow g_j \in G \wedge g_j = v_i$, es decir que t_j tiene asociada g_j a v_i.
- $RN \in B^{|V|\times|T|}/rp_{i,j} = True \Leftrightarrow g_j \in G \wedge g_j = \neg v_i$, es decir que t_j tiene asociada g_j a $!v_i$.
- $NGT \in B^{1\times|T|}/ngt_j = True \Leftrightarrow g_j \notin G$, es decir que t_j no tiene guarda asociada.
- $\vee : B^{1\times x} \to B^{1\times x}$ es la función OR elemento a elemento (bitwise).
- $\times : B^{n\times m} \times B^{m\times p} \to B^{n\times p}$ es la función multiplicación de matrices booleanas.

De esta manera, sg_j será $True$ si:

- t_j no tiene una guarda asociada
- El valor de la variable v_i coincide con el de la guarda g_j asociada a v_i

Luego, si f es la función de sensibilización utilizada hasta este punto, la nueva función de sensibilización f_{ext} es:

$$f_{ext} = f \wedge SG$$

En la figura 3.6 se observa una plaza unida a una transición, con una guarda asociada, referida a la variable ***var***, habilitada por $True$.

Figura 3.6: Transición con guarda

3.3.3. Semántica Temporal

Muchos sistemas reales son dependientes del tiempo entre otras variables. Si resulta de interés modelar un sistema de esta naturaleza, una RdP ordinaria no es suficiente para hacerlo.

A raíz de esto es que se agrega comportamiento temporal a la semántica del disparo de las Redes de Petri para obtener *Redes de Petri Temporales*.

Entre las propuestas para agregar semántica temporal a las RdP, son del interés de este proyecto integrador las Redes de Petri con semántica de *Tiempo Fuerte* y de *Tiempo Débil*.

3.3.3.1. Semántica de Tiempo Fuerte

En sistemas del mundo real, las actividades no ocurren instantáneamente. Cada actividad en un sistema tiene una duración distinta de cero, y se deberá asumir que termina en un tiempo finito para poder modelarla. [Ram74]

En las RdP con semántica de tiempo fuerte, se asume que el disparo de una transición toma un tiempo limitado, distinto de cero.

- Definición 8: Una Red de Petri Temporal con Semántica de Tiempo Fuerte es una tupla (R, Ω) donde:
  - R es una Red de Petri Ordinaria $R = \{P, T, F, W\}$
  - $\Omega : T \to \mathbb{R}^+$ es una función que asigna a cada transición $t_i \in T$ un número real no negativo τ_i, donde τ_i es el *tiempo de disparo de* t_i

Disparo de una Transición con Semántica de Tiempo Fuerte: Si t_i es una transición que se encuentra sensibilizada, se la puede disparar. Cuando se inicia el disparo, se retira la cantidad de tokens correspondiente de $\bullet t_i$ y se dice que la transición t_i *se está ejecutando*. La ejecución de t_i dura un tiempo τ_i, el tiempo de disparo de t_i. Cuando termina de transcurrir el tiempo τ_i, se colocan los tokens correspondientes en $t_i \bullet$ y se dice que el disparo *ha finalizado*.

No se permite comenzar el disparo de una transición que está ejecutando un disparo anterior, es decir que por transición puede existir un único disparo en ejecución.

Nótese que el disparo de una transición, pese a haber dejado de ser instantánteo, sigue siendo atómico como en una RdP ordinaria.

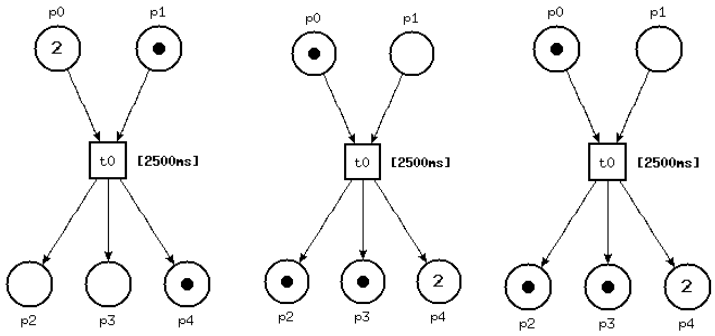
En la figura 3.7 se observa una transición con semántica de tiempo fuerte con tiempo de disparo de 2500ms en las tres fases del disparo.

3.3.3.2. Semántica de Tiempo Débil

De forma más general que la Semántica de Tiempo Fuerte, la Semántica de Tiempo Débil asigna a una transición t un intervalo $[\alpha, \beta]$ denominado el *intervalo de disparo* de t. Este intervalo puede ser cerrado o abierto en cualquiera de sus extremos, y puede extenderse hasta el infinito. El disparo de una transición sólo está permitido dentro de su intervalo de disparo.

Los tiempos α y β son relativos al último instante de sensibilización de t. Esto es, si t se sensibilizó por última vez en el instante θ, el disparo de t sólo será posible en el intervalo $[\theta + \alpha, \theta + \beta]$. [Pet09]

- Definición 9: Una Red de Petri Temporal con Semántica de Tiempo Débil es una tupla (R, IS) donde:



(a) Antes (b) Ejecución (c) Terminado

Figura 3.7: Disparo de una transición con semántica de tiempo fuerte

- R es una Red de Petri Ordinaria $R = \{P, T, F, W\}$
- $IS : T \to \mathbb{Q}^+ \times \{\mathbb{Q}^+ \cup \{\infty\}\}$ es la función de *intervalo estático.*

La función IS asigna a cualquier transición $t \in T$ un intervalo con límites racionales $IS(t) = [\alpha, \beta]$, con $0 \leq \alpha \leq \beta$. Solamente β puede adquirir un valor infinito.

El disparo de t sólo está permitido en el intervalo de tiempo que tenga relacionado. En el instante inicial (tiempo = 0), si t está sensibilizada por el marcado inicial, este intervalo coincide con el intervalo estático $IS(t)$. Cuando el tiempo transcurre, el intervalo de t avanza, corriéndose hacia el origen una cantidad de tiempo igual al transcurrido desde el instante de sensibilización. A este intervalo se le llama *intervalo dinámico de disparo.*

Estos intervalos dinámicos pueden ser expresados como una aplicación I que asigna a cada transición t, un intervalo de tiempo $I(t)$ en el cual puede ser disparada. Los límites del intervalo $I(t)$ son denominados el *instante menor de disparo* y el *instante mayor de disparo* correspondientemente. y son denotados $DMin(t)$ y $DMax(t)$. [Pet09]

Estado de una Red de Petri Temporal: El estado de una RdP con Semántica de Tiempo Débil es un par $E = (M, I)$ donde M es el marcado de la red e I es la *aplicación de intervalos de disparo*. El estado inicial E_0 consiste en la marca inicial M_0 y la aplicación I_0 que asigna a cada transición sensibilizada su intervalo estático y a las transiciones no sensibilizadas, el intervalo vacío. Disparar una transición t_i en el instante θ está permitido desde un estado E, únicamente si se cumple:

- La transición t_i está sensibilizada por el marcado M.
- θ no es menor que el instante menor de disparo de t_i

$$\theta \geq DMin(t_i)$$

- θ no es mayor que el instante mayor de disparo de cualquier transición habilitada por M

$$\forall k, M \geq Pre(k) \Rightarrow \theta \leq DMax(k)$$

Disparo de una transición con Semántica de Tiempo Débil: El disparo de una transición t_i en el instante θ, desde el estado $E = (M, I)$ lleva al estado $E' = (M', I')$ determinado de la siguiente manera:

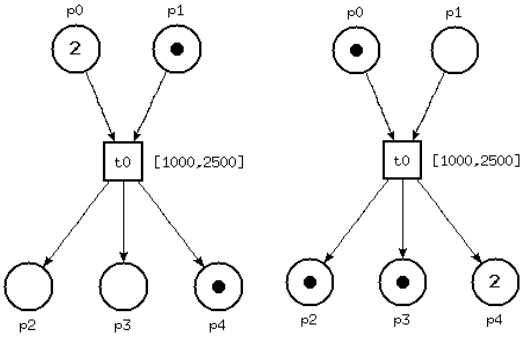
- El marcado M' se determina igual que en una RdP Ordinaria.
- El intervalo $I'(t_j)$ para cada transición t_j se define:

$$I'(t_i) = \begin{cases} \text{Intervalo vacío} & si \quad t_j \text{ no esta sensibilizada por } M \\ IS(t_j) & si \quad t_j \text{ entra en conflicto con } t_i \\ [max(0, DMin(t_j) - \theta), DMax(t_j) - \theta] & si \quad t_j \text{ está sensibilizada y } DMax(t_j) \in \mathbb{Q} \\ [max(0, DMin(t_j) - \theta), \infty] & si \quad t_j \text{ está sensibilizada y } DMax(t_j) = \infty \end{cases}$$

Nótese que, a diferencia de la semántica de tiempo fuerte, el disparo de t_i es instantáneo.

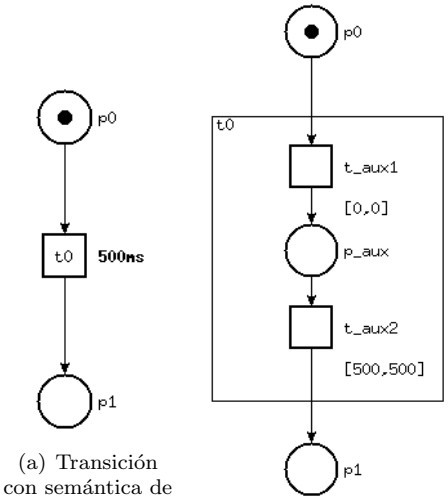
En la figura 3.8 se observa el instante anterior y posterior al disparo de una transición con semántica de tiempo débil. Para que esto ocurra, el instante de disparo tiene que encontrarse dentro del intervalo dinámico de disparo de t_0.

En la figura 3.9 se observa cómo, utilizando transiciones temporales de semántica de tiempo débil, se puede modelar el comportamiento de una transición de semántica de tiempo fuerte. Para entender esta equivalencia, es conveniente repasar mediante un ejemplo el disparo en ambos casos.



(a) Antes (b) Después

Figura 3.8: Disparo de una transición con semántica de tiempo débil



(a) Transición con semántica de tiempo fuerte

(b) Transición con semántica de tiempo fuerte modelada por transiciones con semántica de tiempo débil

Figura 3.9: Transición de tiempo fuerte modelada por tiempo débil

Disparo en Tiempo Fuerte: Como se observa en la figura 3.9(a), t_0 se encuentra sensibilizada.

- En el instante $\tau = 0$ se dispara t_0.
- Automáticamente se retira un token de p_0.
- Durante 500 milisegundos no ocurre nada.
- En $\tau = 500ms$ se coloca un token en p_1.

Disparo en Tiempo Débil: Nuevamente, en la figura 3.9(b), t_0 se encuentra sensibilizada.

- En el instante $\tau = 0$ se dispara t_{aux1}. Esto es posible ya que el intervalo de tiempo asociado lo permite.
- Se retira un token de p_0 y se coloca otro en p_{aux}.
- En ese instante se sensibiliza t_{aux2} y se establece su intervalo dinámico.
- Durante 500 milisegundos no se puede disparar t_{aux2}.
- En $\tau = 500ms$ se dispara t_{aux2}, colocándo un token en p_1.

Así, visto desde el punto de vista de p_0 y p_1, en ambos casos el comportamiento fue idéntico, con lo que se puede afirmar que la semántica de tiempo débil tiene mayor expresividad y por lo tanto, es capaz de modelar más situaciones del mundo real que la semántica de tiempo fuerte.

3.3.4. Autonomía de una RdP

Las redes de Petri analizadas en las secciones anteriores permiten modelar el comportamiento de sistemas describiendo **qué** sucede en un proceso pero no **cuándo** sucede.

Comunicando la RdP con el exterior del modelo, se puede controlar el momento (o al menos el orden) con el que suceden los eventos. De esta manera, la RdP pasa a ser *no autónoma*.

3.3.4.1. Red de Petri Autónoma

En una RdP autónoma, se sabe que una transición puede ser disparada si está sensibilizada, pero no se sabe cuándo o por qué será disparada.

Una extensión sobre este modelo son las ***Redes de Petri Estocásticas***, donde a cada transición se le asigna una variable aleatoria booleana X con función de distribución de probabilidades f, que determina la probabilidad de que se realice un disparo en un instante dado, si la marca M lo permite.

Este tipo de modelos resulta súmamente útil para simular el comportamiento de muchos sistemas del mundo real, ligados a variables aleatorias.

3.3.4.2. Red de Petri No Autónoma

En una RdP no autónoma, se asocia un evento E^i a cada transición t_j y el disparo se da si t_j está sensibilizada y E^i ocurre.

Los *eventos externos* corresponden a un cambio en el estado del medio (incluyendo el tiempo). En cambio, un cambio del estado interno (del marcado), se denomina *evento interno*. La ocurrencia de un evento no tiene duración. [Hyb10]

- Definición 10: Una RdP sincronizada es una tupla $RdP_{Sync} = \{R, E, Sync\}$ tal que:
  - $R = \{P, T, F, W\}$ es una Red de Petri marcada.
  - E es un conjunto de eventos externos.
  - $Sync : T \to (E \cup \{e\})$ donde e es el evento que ocurre siempre.
- Definición 11: En una RdP no autónoma, una transición t es *inmediata* o *automática* si es disparada q veces cuando está *q-sensibilizada* $\forall q > 0$, a menos que existan conflictos entre dos o más transiciones.

Se dice que una transición sujeta a un evento externo es *disparada*, de otro modo será *automática*. Esto se identifica en el modelo con una etiqueta D o F para una transición disparada, y A para una automática.

Eventos estocásticos en redes de Petri no autónomas: Si una transición t_j está sensibilizada, es disparada y espera la ocurrencia del evento E^n se disparará cuando E^n ocurra. Si a su vez se construye un generador de eventos E^n sujeto a una variable aleatoria X con distribución de probabilidad f, se puede replicar la semántica de una RdP estocástica utilizando una RdP no autónoma. [4]

3.3.5. Informes de Disparo

De la misma forma que una RdP no autónoma permite actuar en función de eventos provenientes del mundo exterior, los informes de disparo le brindan a la RdP la posibilidad de emitir eventos hacia el medio. A su vez, estos eventos pueden desencadenar acciones en observadores del mundo externo.

Una transición que emite eventos se denomina *informada*. Esto se denota con una etiqueta en la transición, indicando I si es informada, o N si no lo es.

Para demostrar el potencial de este mecanismo, se analiza el ejemplo de la figura 3.10:

Si se relaciona una plaza p_0 con la ejecución de una *acción_1* en el modelo, se puede aprovechar el mecanismo de informes para sincronizar esta acción.

A continuación se describe una sucesión de eventos que aprovecha este mecanismo, referido a la RdP de la figura 3.10:

- Sucede un evento E^0_{ext}, t_0 lo escucha y de dispara.
- t_0 emite un evento E^0_{int} avisando que fue disparada.
- Un observer de E^0_{int} ejecuta *acción_1*.
- La finalización de *acción_1* emite un evento E^1_{ext} hacia la RdP.
- t_1 escucha el evento E^1_{ext} y se dispara.

[4] Esta técnica se puede utilizar para depurar el modelo, agregando eventos mediante simulación

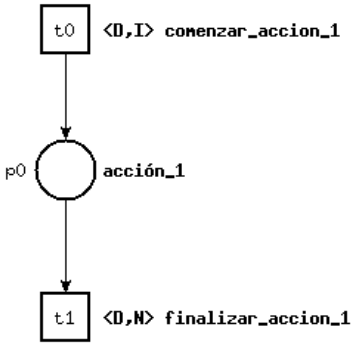


Figura 3.10: Emisión de Eventos ligados a acciones

3.3.6. Política de Selección de Disparo

Si existen múltiples transiciones sensibilizadas en una RdP, resulta de interés tener un mecanismo de decisión que indique cuál debería ser la próxima transición disparada. Esta decisión resulta particularmente significativa en el caso de que dos o más transiciones se encuentren en conflicto, ya sea un conflicto inmediato o un conflicto que se genere luego de aplicar una sucesión de disparo σ.

El análisis del surgimiento de conflictos y de su posible resolución en redes de Petri, lleva a algoritmos de búsqueda en árboles, con lo que su complejidad crece exponencialmente con la profundidad del árbol a analizar, es decir, con la cantidad de disparos futuros a analizar y con la cantidad de transiciones analizadas. Esto lo hace imposible de aplicar para resolver problemas de decisión de disparo por medio de la heurística.

La *Política de Selección de Disparo* se refiere a la elección de la próxima transición a disparar entre todas las sensibilizadas. Para una RdP, incluidas la no autónomas o temporales, si esta elección es aleatoria el sistema resulta no determinístico. Para que el sistema sea determinístico hay diferentes soluciones, como la inclusión de: prioridades, probabilidades, arcos inhibidores, arcos lectores. etc. [DIOM16]

La política de selección de disparo $P : TS \rightarrow t/TS \subseteq T \wedge t \in TS$ es una función que dado un vector de transiciones sensibilizadas TS indica cuál transición t deberá ser disparada.

Para un determinado problema a resolver con una RdP, con conocimiento del dominio del problema se puede diseñar una política de selección de disparos que contemple los conflictos que puedan surgir, y priorice el disparo de transiciones que generen situaciones más favorables para la solución de dicho problema.

Cabe destacar que, pese a obtener un diseño correcto del modelo, una política desfavorable al problema a tratar puede beneficiar ampliamente a parte de la ejecución del modelo y perjudicar a otras, generando inanición sobre estas.

3.3.7. Redes de Petri Orientadas a Procesos

Las RdP orientadas a procesos son aquellas que, como su nombre lo indica, modelan procesos. En este tipo de redes se tienen las siguientes consideraciones:

- Las plazas pueden representar recursos u operaciones
- Las transiciones representan el inicio o fin de una operación
- Un token en el marcado inicial representa:
  - Una constante en el caso de una plaza de recurso
  - Una variable en el caso de una plaza de operación

Frecuentemente, se suele llamar *POPN* a las RdP orientadas a procesos por sus siglas en inglés *(Process Oriented Petri Net)*.

3.4. Comparación entre Redes de Petri y Autómatas

Comparado a los autómatas, las RdP ofrecen una representación compacta de sistemas concurrentes. Por esto, el modelado de este tipo de sistemas es usualmente más natural usando redes de Petri. [IA06]

Para comparar RdP y autómatas, se consideran sistemas compuestos por varios subsistemas y se compara el tamaño de la RdP y el autómata que modelan el sistema completo.

A fin de realizar esta comparación, es necesario previamente definir la composición de autómatas y de redes de Petri. [IA06]

- Definición 12: Sean $G_1 = (Q_1, \Sigma_1, \delta_1, s_1, F_1)$ y $G_2 = (Q_2, \Sigma_2, \delta_2, s_2, F_2)$ autómatas no determinísticos. La *Composición Paralela* o *Síncrona* de G_1 y G_2 es un autómata $G = G_1 \parallel G_2$ que se define:

$$G = Ac(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (s_1, s_2), F_1 \times F_2)$$

donde Ac es la función que elimina los estados no alcanzables de un autómata y δ se define de la siguiente forma:

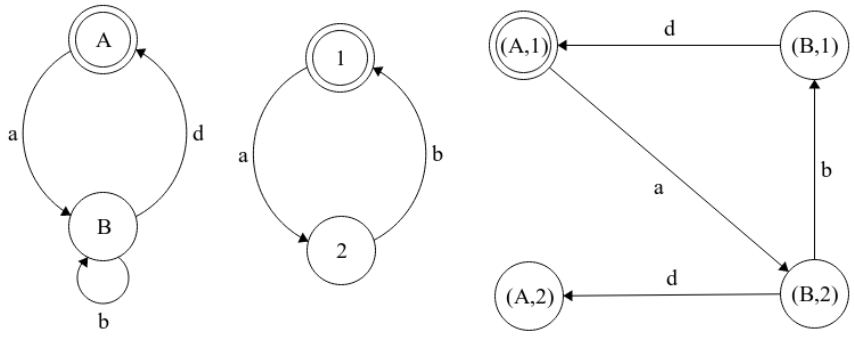
- Definición 12.1: Sea

$$\bar{\delta_i}(q, \alpha) = \begin{cases} \delta_i(q, \alpha) & \forall \alpha \in \Sigma_i \\ \{q\} & \forall \alpha \in (\Sigma_1 \cup \Sigma_2 \cup \{\lambda\}) \backslash \Sigma_i \end{cases}$$

para $i = 1, 2$ y donde λ es el evento nulo y $\backslash$ es la operación resta de conjuntos. Entonces:

$$\delta((q_1, q_2), \alpha) = \bar{\delta_1}(q_1, \alpha) \times \bar{\delta_2}(q_2, \alpha)$$

A fin simplificar el entendimiento de la composición de autómatas, se presenta el siguiente ejemplo:



(a) G_1 (b) G_2 (c) $G_1 \parallel G_2$

Figura 3.11: Composición de Autómatas

Sean

$$G_1 = (\{A, B\}, \{a, b, d\}, \delta_1, A, \{B\})$$

y

$$G_2 = (\{1, 2\}, \{a, b\}, \delta_2, 1, \{2\})$$

dos autómatas, entonces $G = G_1 \parallel G_2$ tendrá:

- Estado inicial $(A, 1)$ ya que A y 1 son ambos estados iniciales.
- Estados finales $\{(B, 2)\}$ ya que B y 2 son estados finales.
- Las transiciones son:
  - De $(A, 1)$ a $(B, 2)$ etiquetada con a porque hay una transición de A a B y otra de 1 a 2, ambas etiquetadas con a.
  - De $(B, 2)$ a $(B, 1)$ etiquetada con b porque hay una transición de B a B y otra de 1 a 2, ambas etiquetadas con b.
  - De $(B, 1)$ a $(A, 1)$ etiqueada con d, porque en G_1 hay una transición de B a A etiquetada con d, y como d no existe en el conjunto de símbolos de G_2 no puede generar restricciones y según la definición genera una transición de 1 hacia 1.
  - De $(B, 2)$ a $(A, 2)$ etiquetada con d por la misma razón que la transición anterior. La transición de A a B existe en G_1, y se genera una de 2 a 2 en G_2

Tamaño del autómata compuesto: Analizando el tamaño del modelo compuesto, asumiendo que todos los estados de $Q_1 \times Q_2$ son alcanzables desde el estado inicial, si G_1 tiene m_1 estados y G_2 tiene m_2 estados, entonces $G_1 \parallel G_2$ tiene $m_1 m_2$ estados. Es decir que la cantidad de estados de la composición de autómatas es (o está acotada por) el **producto** de la cantidad de estados de los autómatas que lo componen.

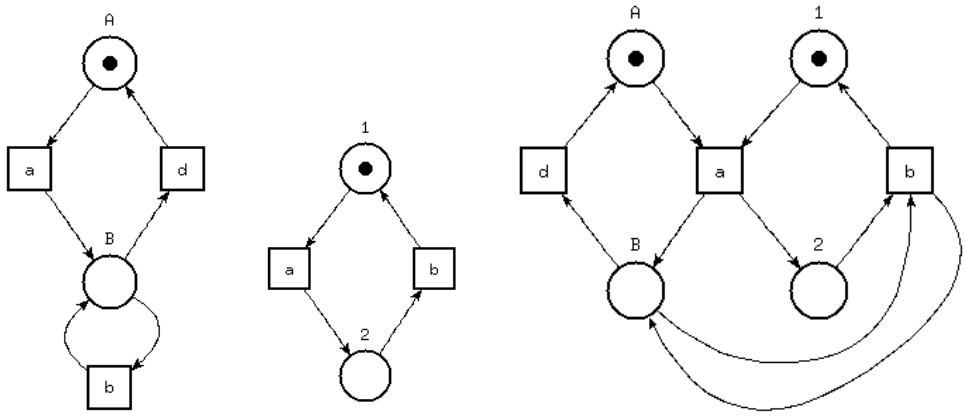
- Definición 13: Sean $R_1 = (P_1, T_1, Pre_1, Pos_1, \rho_1)$ y $R_2 = (P_2, T_2, Pre_2, Pos_2, \rho_2)$ redes de Petri etiquetadas donde:

- $\rho_i : T_i \to \Sigma_i \cup \{\lambda\}$ para $i = 1, 2$ es la función de etiquetado de las transiciones de R_i.
- Σ_i es el conjunto de símbolos del autómata equivalente a la i-ésima red de Petri
- λ es el evento nulo

La composición resulta en una red $R = (P, T, Pre, Pos, \rho)$ tal que T contiene las transiciones de T_1 y de T_2 que no están etiquetadas con λ y las transiciones t que modelan sincronizaciones de pares $(t^1, t^2) \in T_1 \times T_2$ que comparten etiquetas distintas de λ. Se puede construir la red compuesta utilizando el siguiente algoritmo:

1. Sea $P = P_1 \cup P_2$ y $T = \emptyset$.
2. Sean $L_1 = \bigcup_{t \in T_1} \rho_2(t) \backslash \{\lambda\}$ y $L_2 = \bigcup_{t \in T_2} \rho_1(t) \backslash \{\lambda\}$.
3. Para $i = 1, 2$, sea $T_{i,-} = \{t \in T_i : \rho_i \backslash L_i \neq \emptyset\}$.
4. Para $i = 1, 2, \forall t \in T_{i,-}$, sea $T = \{t\} \cup T$. $\rho(t) = \rho_i(t) \backslash L_i$. Sea $Pre(p, t) = Pre_i(p, t)$ y $Pos(p, t) = Post_i(p, t) \forall p \in P$ y sea $Pre(p, t) = 0$ y $Pos(p, t) = 0$ $\forall p \in P \backslash P_i$.
5. Para cada par de transiciones $(t^1, t^2) \in T_1 \times T_2$ tal que $(\rho_1(t^1) \cap \rho_2(t^2)) \backslash \{\lambda\} \neq \emptyset$, crear una nueva transición $t : T = t \cup T$, $\rho(t) = (\rho_1(t^1) \cap \rho_2(t^2)) \backslash \{\lambda\}$ y hacer $Pre(p, t) = Pre_i(p, t)$ y $Pos(p, t) = Pos_i(p, t)$ $\forall p \in P_i$ para $i = 1, 2$.

Retomando el ejemplo de la composición de autómatas de la figura 3.11, se presenta la composición de las redes de Petri equivalentes a dichos autómatas.



(a) R_1 (b) R_2 (c) $R_1 \parallel R_2$

Figura 3.12: Composición de Redes de Petri

Tamaño de la RdP compuesta: Si R_1 tiene m_1 plazas y R_2 tiene m_2 plazas, entonces $R = R_1 \parallel R_2$ tendrá $m_1 + m_2$ plazas. En contraste con el caso del autómata, la red de Petri compuesta crece linealmente en cantidad de plazas respecto de las redes originales, es decir que crece con la **suma** de las plazas de las redes originales. Además, las transiciones de la red compuesta son las de las redes originales, unificando las que comparten etiqueta. Es decir que la cantidad de transiciones del modelo compuesto está acotada por la **suma** de las cantidades de los modelos que lo componen.

Conclusión sobre la comparación de modelos: Analizando la forma en que crece el modelo resultado de la composición de autómatas contra la composición de RdP se ve que estas últimás son más aptas para el modelado de manera compacta de sistemas paralelos formados por subsistemas menores. De la misma manera, una RdP permite modelar sistemas paralelos mucho mayores que un autómata sin que el modelo resulte demasiado complejo para su análisis. Otro punto a destacar es la existencia de un algoritmo para la composición de redes de Petri, lo que permite la automatización de este proceso, simplificando la tarea de construcción del modelo.

Capítulo 4

Paradigmas de Programación

4.1. Paradigma Dataflow

El paradigma de programación *Dataflow* se basa en la idea de evitar que el programador piense en términos del flujo de control del programa y se centre en el flujo de los datos que son procesados. De esta manera, las aplicaciones son representadas como un conjunto de nodos (o bloques) con puertos de entrada y/o salida. Estos nodos pueden ser productores, consumidores o bloques de procesamiento de información que fluye por el sistema. Los nodos están conectados por aristas que definen el flujo de información por el sistema. La mayoría de los lenguajes de programación visuales que usan una arquitectura basada en bloques están basados en el paradigma dataflow. [Sou]

Los nodos son ejecutados únicamente cuando reciben y/o envían mensajes, lo que sucede asíncronamente respecto de los demás nodos. Por esto, las aplicaciones dataflow son inherentemente paralelas. [Dat]

La programación dataflow es capaz de proveer paralelismo sin la complejidad de la gestión de hilos. Esto es posible gracias a que cada nodo es un bloque de procesamiento independiente de los demás y no produce efectos colaterales [Sou]

Hay una amplia variedad de lenguajes dataflow. Se pueden mencionar software de hojas de cálculo, Labview y Erlang, entre otros. La programación se hace alterando diagramas de flujo. Una característica principal de este tipo de lenguajes es que contienen un sistema de ejecución (runtime system). [Dat]

Los programas imperativos tradicionales están compuestos de rutinas que se llaman entre sí. Por ejemplo, en una secuencia de llamada un llamador construye un paquete de datos y transfiere el control y el paquete de datos a una rutina llamada. Cuando la rutina llamada termina, contruye otro paquete de datos para pasar de vuelta al llamador y le transfiere nuevamente el control.

En los programas dataflow las "rutinas" no se llaman entre sí, en su lugar son activadas por el sistema de ejecución cuando hay entrada para ellos. Cuando se crean salidas, el sistema de ejecución se hace cargo de mover la salida al destino que requiere esas salidas. Cuando las "rutinas" terminan, transfieren el control de vuelta al sistema de ejecución.

Una diferencia entre la programación imperativa y dataflow es la semántica utilizada para la transferencia de datos. Mientras la programación imperativa utiliza semántica LIFO, la dataflow usa semántica FIFO [Dat]. Eso es, un programa imperativo pone datos en una pila y obtiene datos desde la misma pila. En cambio en programas dataflow, cada elemento obtiene datos de una cola y pone datos en otras colas. Otra diferencia es que la conectividad de los programas procedurales está embebida en el código. Para pasar datos de la rutina A a B, A debe llamar explícitamente a B, es decir que un llamado tiene que especificar el destino de los datos. Por otro lado, en programas dataflow la conectividad puede estar separada del código, A no pasa datos directamente a B; en su lugar, le pasa datos al sistema de ejecución, quien le pasa los datos a B. El llamador no tiene que especificar hacia dónde van los datos y hasta puede no saberlo. [Dat]

Ventajas y Desventajas del paradigma Dataflow

Entre las ventajas de utilizar el paradigma dataflow se encuentran:

- La concurrencia y paralelismo son naturales. El código se puede distribuir entre cores y a través de redes. Algunos problemas relacionados a hilos desaparecen
- Las redes dataflow son representaciones naturales e intuitivas para representar procesos.
- El paso de mensajes permite deshacerse de problemas asociados a memoria compartida y locks.
- Los elementos pueden ser agrupados en elementos compuestos, mejorando la escalabilidad de los sistemas.

Por otro lado, resulta poco ventajoso utilizar este paradigma por los siguientes motivos:

- El modelo de pensamiento de programación dataflow es poco familiar para la mayoría de los programadores profesionales.
- La mayoría de los lenguajes de programación dataflow son lenguajes de un nicho usado por programadores no profesionales.
- La intervención del sistema de ejecución puede tener aparejado un alto costo computacional. La gran ventaja de la semántica LIFO es que se implementa en código de manera inmediata y poco costosa.
- No utilizar memoria compartida tiene sus costos. Los mensajes deben ser copiados o deben ser inmutables.
- No son compatibles con otros lenguajes de programación. De esta manera, convertir programas tradicionales en programas dataflow es complejo.

4.2. Paradigma Reactivo

El paradigma reactivo es un paradigma de programación construído en torno a flujos de datos, y la propagación de los cambios sobre ellos. Esto significa que los lenguajes que implementan este paradigma deben permitir expresar flujos de datos de manera estática o dinámica con facilidad, y el modelo de ejecución debe propagar automáticamente los cambios en los datos cuando ocurran, actualizando todos los valores correspondientes de manera transparente para el programador.

A fin de comprender las características principales de este paradigma se presenta el siguiente ejemplo:

```
a = 1
b = 2
c = a + b
a = 3
```

En programación imperativa, terminada la ejecución de esta sección de código, la variable c contiene el valor 3 y se mantendrá de esta manera indefinidamente hasta que el programador le asigne un nuevo valor. En cambio en programación reactiva el valor de c se mantiene siempre actualizado, es decir, la expresión declarada como c se vuelve a computar automáticamente ante un cambio en a o en b, y en este ejemplo pasa a valer 5. Se dice que c es *dependiente* de a y b. [BCC+13]

Al igual que en el paradigma dataflow, en el paradigma reactivo tiene relevancia el flujo de datos, en lugar del flujo de control. La diferencia radica en que, bajo el paradigma reactivo, las "conexiones" de datos pueden ser alteradas dinámicamente en tiempo de ejecución. Además se introducen restricciones de tiempo real blando, para lo cual se definen dos conceptos:

- *Behaviours (Comportamientos)* representan eventos de variación contínua en el tiempo. El behaviour por excelencia es el tiempo, de hecho los lenguajes reactivos ofrecen primitivas para representar al tiempo.
- *Events (Eventos)* representan eventos discretos. Suelen estar representados en forma de flujos de cambios de valores. A diferencia de los behaviours, los eventos cambian en instantes puntuales del tiempo. Los lenguajes reactivos ofrecen primitivas para combinar y procesar eventos.

[BCC+13]

El Paradigma Reactivo y El Patrón Observer

El patrón de diseño *observer* [GHJV95] nace de la necesidad de mantener consistencia de datos en sistemas particionados, sin generar acoplamiento entre capas de dichos sistemas. Permite que un *sujeto* publique cambios en su estado a sus *observadores*, quienes se susbribieron previamente a estas actualizaciones.

El patrón observer se debe utilizar en alguna de las siguientes situaciones:

- Cuando una abstracción tiene dos partes, una dependiente de la otra.
- Cuando el cambio en un objeto implica el cambio en otros, y no se sabe de antemano cuántos ni quiénes deben aplicar estos cambios.
- Cuando un objeto debe notificar a otros sin conocer nada de ellos, es decir sin generar acomplamiento.

[GHJV95]

Analizando este patrón de diseño se encuentran similitudes con el paradigma reactivo. Por ejemplo, el patrón observer describe flujos de datos a nivel de clases, mientras que el paradigma reactivo describe flujos a nivel de clases, miembros de clases y variables.

En programación reactiva, cuando se forma una expresión dependiente de otras, se genera una suscripción implícita de manera automática y el modelo de ejecución es el encargado de propagar los cambios de manera transparente para el programador.

4.3. Programación Orientada a Aspectos

4.3.1. Concepto

La programación orientada a aspectos es un paradigma de programación que tiene como objetivo incrementar la modularidad mediante la separación de intereses transversales (cross-cutting concerns). Los intereses transversales son aspectos de un programa que afectan a otros intereses. Son partes de un programa que afectan o dependen de muchas otras partes del sistema. Estos intereses usualmente no pueden separarse claramente del resto del sistema, y pueden resultar en duplicación de código o un alto grado de dependencia entre partes del sistema.

Los intereses transversales son la base para el desarrollo de aspectos. Estos no pueden ser representados claramente en los paradigmas de programación orientado a objetos o programación procedural. [Asp03] La separación de intereses transversales se realiza añadiendo comportamientos adicionales al código existente, llamados advices o consejos, sin modificar el mismo. Para lograrlo, se especifican puntos de ejecución (mediante la definición de pointcuts) donde se aplican los advices previamente mencionados.

La programación orientada a aspectos complementa a la programación orientada a objetos al permitir al desarrollador modificar dinamicamente el modelo estático orientado a objetos para crear un sistema que puede crecer para cumplir nuevos requerimientos. Tal como los objetos en el mundo real pueden cambiar sus estados a lo largo de su vida, una aplicación puede adoptar nuevas características a medida que se va desarrollando. [O'R04]

4.3.2. Terminología

- Intereses Transversales (Cross-cutting concerns): Aunque la mayoría de las clases en un modelo orientado a objetos está destinada a perfeccionar una función única y específica, usualmente comparten requerimientos secundarios en común con otras clases. Por ejemplo, se puede desear añadir mecanismos de logueo a las clases dentro de la capa de acceso de datos y también a las clases en la capa de interfaz de usuario cada vez que un hilo entre o salga de un método. Aunque la funcionalidad principal de cada clase es muy diferente, el código necesario para realizar la tarea secundaria es usualmente idéntico. [O'R04]

- Consejos (Advices): Es el código adicional que se desea aplicar al modelo existente. Siguiendo con el ejemplo anterior, es el código de logueo que se quiere aplicar cada vez que un hilo ingrese o salga de un método. [O'R04]

- Punto de unión (Join-point): Es el término que se le otorga al punto de ejecución en la aplicación en el cual los intereses transversales deben ser aplicados. En el ejemplo, un punto de unión es alcanzado cuando un hilo ingresa a un método, y un segundo punto de unión es alcanzado cuando un hilo sale de un método.

- Punto de corte (Point-cut): Un punto de corte es un conjunto de puntos de unión. Un point-cut permite definir dónde aplicar exactamente un consejo, lo cual permite la separación de intereses y ayuda a modularizar la lógica de negocios [SH05].

- Aspecto (Aspect): La combinación de un punto de corte y un consejo se denomina aspecto. [O'R04]

- Tejido (Weaving): Proceso de aplicar aspectos a los objetos destinatarios para crear los nuevos objetos resultantes en los puntos de unión especificados. De acuerdo al momento del ciclo de vida del sistema en el cual se aplica el tejido, se realiza la siguiente clasificación:

  - Aspectos en Tiempo de Compilación.
  - Aspectos en Tiempo de Carga.
  - Aspectos en Tiempo de Ejecución.

4.4. Programación Orientada a Objetos

Existe extensa bibliografía de consulta acerca del paradigma de Programación Orientada a Objetos. Es de interés en particular aquella bibliografía basada en el lenguaje Java (ver [Fla05] [Laf03]). Dentro del marco de la progrmaación orientada a objetos, tiene gran relevancia en este proyecto el concepto de Reflection, expuesto en la sección 4.4.1.

4.4.1. Reflection

En determinados escenarios de programación es útil conocer en tiempo de ejecución la estructura interna de los objetos y las operaciones del programa. De esta manera, en base a esta información y al flujo de ejecución, el programa modifica su propio comportamiento.

Estas capacidades se obtienen por medio de la programación basada en la *reflexión* del programa.

Reflexión (o reflection) es la habilidad de un programa de examinarse a sí mismo y a su entorno en tiempo de ejecución, y de cambiar su propio comportamiento.

Para realizar esta autoexaminación, un programa necesita tener una representación de sí mismo. Esta información se llama *metainformación* o *metadata*. En particular, en un entorno de programación orientada a objetos la metadata se organiza en *metaobjetos*. La revisión en tiempo de ejecución de los metaobjetos se llama *introspección* o *introspection*. [FFI+04]

La metainformación es el elemento estructural más importante de un sistema reflectivo. Examinando su autorepresentación, un programa obtiene información acerca de su estructura y comportamiento para tomar decisiones importantes.

En general, la introspección está seguida de un cambio del comportamiento del propio programa. Existen tres técnicas que una interfaz de reflection ofrece para generar cambios de comportamiento:

- Modificación de los metaobjetos
- Operaciones con la metadata: como la invocación dinámica de métodos.
- Intercesión: El código intercede en varias fases de la ejecución para alterar el comportamiento del programa.

Estas características permiten diseñar software más flexible, que se adapta a cambio de requerimientos, favoreciendo a la mantenibilidad.

Existen tres problemas relacionados al uso de reflection en el diseño de un programa que deben tenerse en cuenta:

- Seguridad
- Complejidad del código
- Performance

Los problemas mencionados se mitigan mediante el uso de buenas prácticas de programación y un correcto diseño del software.

Capítulo 5

Generación de Código Frameworks y APIs

5.1. Generación de Código Fuente

La idea de generación automática de código fuente y de código ejecutable es casi tan antigua como la programación en sí misma. Debido a que ahorra mucho tiempo y costo de desarrollo de sistemas, ha sido y sigue siendo foco de investigación.

La generación automática de código fuente está englobada por el concepto de *Programación Automática*. El significado de este término ha avanzado junto a la programación a lo largo de los años:

- En la década de 1940, se llamó de esta forma a la automatización del proceso de perforar cintas de papel para escribir el programa [Gor40]. Lo que Gorn llamó programación automática, es en realidad un lenguaje assembler.
- En el comienzo de los lenguajes de alto nivel se le llamó de esta manera a los compiladores. Tanto es así que uno de los primeros compiladores se llamó Autocode.
- Actualmente se identifica este término como la generación de código fuente escrito en un lenguaje de programación (compilable o interpretable a código máquina) a partir de una descripción de más alto nivel.

Algunos ejemplos de generadores de código son:

- Apache Thrift: Desarrollado por Facebook y actualmente liberado bajo licencia Apache, Thrift es un generador de servicios para múltiples lenguajes orientado a la comunicación por medio de llamadas a procedimiento remoto (remote procedure call – RPC). Para lograr esto expone un lenguaje de definición de interfaces (interface definition language – IDL) propio, utilizado para describir el servicio que luego Thrift generará en alguno de los múltiples lenguajes que soporta. [Apa]
- Acceleo: Es un generador de código que implementa el estándar *MOFM2T* desarrollado por The Eclipse Foundation y su código fuente está liberado bajo licencia EPL. Acceleo permite la especificación del software en modelos como UML (v1 y v2), EMF (eclipse model framework) y lenguajes de modelado personalizados (DSL). Además permite especificar plantillas definidas por el usuario. Genera código en múltiples lenguajes de programación. [Acc]
- Actifsource: Desarrollado por actifsource GmbH y de código cerrado, es un generador de código a partir de modelos similares a UML. Soporta la creación de múltiples modelos y la unión de estos, y la utilización de modelos generados en cualquier software que tolere formato ecore, definido por el Eclipse Modeling Framework. Está desarrollado como un plugin para Eclipse. [Act]
- Spring Roo: Desarrollado conjuntamente por DISID y Pitvotal bajo licencia Apache 2.0, es un generador enfocado al desarrollo acelerado de software empresarial en Java. La aplicación generada utiliza tecnologías Java comunes como Spring Framework, Java Persistence API, Apache Maven, etc. A diferencia de otros generadores de código, Roo expone una interfaz por línea de comandos con sus propios comandos. [Spr]
- GeneXus: Desarrollado por ARTech bajo licencia cerrada y con primer lanzamiento en 1988, es un generador de código fuente a partir de un lenguaje declarativo de alto nivel. A partir de este lenguaje, se genera código fuente en C#, COBOL, Java, Objective-C, RPG, Visual Basic, Ruby y Visual FoxPro. Además tolera múltiples lenguajes para gestión de bases de datos como Microsoft SQL, Oracle, DB2, Informix, PostgreSQL y MySQL. [Gen]

La programación automática siempre ha sido un eufemismo para la programación con un lenguaje de más alto nivel del disponible para el programador. Investigar en programación automática es simplemente desarrollar la implementación de lenguajes de programación de más alto nivel [Par85].

En conclusión, los generadores automáticos de código fuente son en realidad traductores de un lenguaje de programación a otro. Esto brinda un mayor nivel de abstracción para el programador pero lo obliga a especificar su software en el lenguaje provisto por el generador.

5.2. Frameworks

5.2.1. Definición

Los frameworks son una técnica de reutilización de prácticas, conceptos y criterios orientadas a facilitar la solución de un tipo de problemáticas en particular. Son estructuras concretas de software, que proveen una manera estándar de construir aplicaciones. Sirven como base para el diseño y desarrollo de software orientado a resolver problemas específicos. De acuerdo a [Joh97b] dos de las definiciones más comunes de framework son:

- "Un framework es un diseño reusable de todo o parte de un sistema que es representado por un conjunto de clases abstractas y la forma en que sus instancias interactúan"
- "Un framework es el esqueleto de una aplicación que puede ser personalizado por un desarrollador de aplicaciones."

La primer definición describe la estructura de un framework, mientras que la segunda describe su propósito.

Un framework es una técnica de reutilización de código porque facilita la creación de una aplicación a partir de una biblioteca de componentes existentes. Es posible la creación de nuevos componentes extendiendo los provistos por el framework. Una característica que distingue a los frameworks de otras técnicas de reutilización es la inversión de control (ver sección5.2.2).

Debe pensarse en frameworks y componentes de software como tecnologías diferentes, pero que cooperan entre sí [Joh97a]:

- Un framework provee un contexto reusable para los componentes.
- Un framework es más abstracto y flexible que los componentes.

Por otro lado, los frameworks son más concretos y simples de reutilizar que un diseño puro [Joh97a].

5.2.2. Inversión de Control

La inversión de control es una característica principal de los frameworks. Es un principio de diseño en el cual porciones de código personalizado por el usuario son controladas por un framework.

Al implementar un sistema sin utilizar un framework, generalmente el desarrollador escribe el código de un programa principal que realiza llamadas a componentes de una biblioteca. El desarrollador decide en el código cuándo llamar al componente y se encarga de la estructura y el flujo de control del programa.

En un software basado en un framework el programa principal es reutilizado. El desarrollador solamente conecta componentes existentes al framework, o implementa nuevos componentes para conectar. Las porciones de código del desarrollador son llamadas por el framework. De esta manera, el framework determina la estructura y el flujo de control del programa.

La inversión de control sirve para los siguientes propósitos de diseño:

- Desacoplar la ejecución de una tarea de su implementación.
- Mantener el foco en la tarea para la que fue diseñado el módulo.
- Guiar el diseño respetando las interfaces entre módulos.
- Evitar efectos colaterales al reemplazar un módulo.

5.2.3. Ventajas de los frameworks

- Al utilizar un framework se aplican técnicas de reutilización de software y de diseño.
- Son personalizables: Los frameworks son más personalizables que la mayoría de los componentes. Tienen interfaces más complejas.
- Sirven para múltiples aplicaciones: Un framework está orientado a facilitar la implementación de aplicaciones de un tipo determinado. En consecuencia, puede ser utilizado para implementar diversas aplicaciones que pertenezcan a dicho tipo.
- Facilitan el trabajo del desarrollador.
- La uniformidad reduce los costos de mantener el código: Los programadores encargados de mantenerlo pueden cambiar de una aplicación a otra que utiliza el mismo framework sin tener que aprender un nuevo diseño.
- Los frameworks obligan al usuario a respetar patrones de diseño en las aplicaciones.

5.2.4. Desventajas de los frameworks

- Curva de Aprendizaje: Los programadores deben aprender las interfaces antes de poder utilizar el framework. Generalmente aprender un nuevo framework es difícil.
- Restricción de elección del lenguaje de programación: Uno de los problemas de utilizar un framework implementado en un lenguaje en particular es que restringe a los sistemas a utilizar dicho lenguaje. La relación efectividad-costo es baja al construir una aplicación en un lenguaje con un framework escrito en otro.
- Debido a que los frameworks son descritos con lenguajes de programación, es difícil para los desarrolladores aprender los patrones colaborativos de un framework mediante la lectura del código.

5.2.5. Frameworks desde la perspectiva del usuario

Según [Joh97a] existen tres formas de utilizar un framework desde la perspectiva de un usuario desarrollador de software:

- Black-Box Frameworks: Consiste en conectar componentes ya existentes. De esta forma no se modifica el framework ni se crean nuevas clases concretas sino que se reutilizan las interfaces del framework y sus reglas para interconectar componentes. Este método es similar a la construcción de un circuito eléctrico. El desarrollador necesita conocer la interfaz de conexión entre un objeto A y un objeto B, pero no es necesario que conozca la especificación exacta de A o B.
- White-Box Frameworks: Consiste en definir clases concretas, que extienden de clases abstractas definidas en el framework, y utilizarlas para implementar una aplicación. Las subclases están estrechamente acopladas a sus superclases. De esta forma se requiere más conocimiento acerca de la implementación de las clases que conforman el framework.
- Extensión o Modificación del núcleo del framework: Consiste en extender el framework cambiando las clases abstractas que forman su núcleo para añadir nuevas variables u operaciones. Requiere conocimientos avanzados acerca del diseño del framework. Cambiar las clases abstractas puede provocar fallos en las clases concretas existentes. Este modo de utilización no es aplicable si el propósito es crear un sistema abierto.

Entre las formas de utilización mencionadas existen combinaciones intermedias. Es común que los frameworks se utilicen como Black-Box la mayor parte del tiempo y sean extendidos cuando la ocasión lo demande.

5.3. Comparación entre Frameworks y APIs

En la tabla 5.1 se observa una comparación entre frameworks y APIs.

Categoría	Framework	API (Biblioteca)
Gestión del Flujo Principal	El framework toma el flujo principal del software	A cargo del programador
Confiabilidad	El flujo principal está ampliamente testeado por todos los usuarios del framework	No brinda ninguna garantía de flujo
Extensibilidad	Por parte de los desarrolladores del framework. Si es de código abierto cualquiera puede extenderlo.	Por parte del fabricante de la librería. O cualquiera si es de código abierto
Reusabilidad	Objetivo principal del diseño de un framework. Se aplica a nivel de arquitectura de software	Es reutilizable a nivel de llamada a métodos
Complejidad de Uso	Gran complejidad al principio, se simplifica a medida que el usuario aprende el framework	Complejidad inicial menor que un framework
Aplicación de Patrones de diseño	Usualmente un framework fuerza al usuario a utilizar uno o varios patrones	No obliga al usuario a utilizar ningún patrón de diseño
Especificidad / Generalización	Son de uso específico, están diseñados para resolver una familia de problemas. Por esto mantienen una arquitectura	De uso general donde una funcionalidad pueda ser utilizada
Restricciones de lenguaje	Obliga al usuario a desarrollar en el mismo lenguaje en el que está hecho el framework	No restringe a un lenguaje. Permite llamadas desde cualquier lenguaje mientras se respete la firma de las funciones expuestas

Cuadro 5.1: Comparación entre Frameworks y APIs

Capítulo 6

Concurrencia

6.1. Introducción

"La idea de programación concurrente siempre estuvo asociada al mundo de los *Sistemas Operativos*. No en vano, los primeros programas concurrentes fueron los propios Sistemas Operativos de multiprogramación en los que un solo procesador debía repartir su tiempo entre muchos usuarios." [Pal03]

En las últimas dos décadas, la programación concurrente ganó gran interés y actualmente está presente en la mayoría de las aplicaciones. Esto se debe principalmente a algunos grandes hitos en la programación:

- La generalización del concepto de *hilo* o *thread*. Permiten la ejecución de programas de manera más rápida y eficiente que los programas basados en procesos.
- La disponibilidad de lenguajes de alto nivel con soporte para programación de hilos y de procesos.
- Los entornos de internet, donde la concurrencia se hace necesaria en todo aspecto.
- El desarrollo y gran avance de hardware capaz de ejecutar múltiples hilos y procesos de forma paralela. Esto permite aprovechar las ventajas de performance de la concurrencia. Las principales arquitecturas capaces de explotar el paralelismo a nivel de hilo y/o de proceso son
  - Procesadores Multi-Core
  - Procesadores Many-Core
  - Procesadores con soporte Multi-Thread

6.2. Programación Concurrente

La *programación concurrente* es la disciplina que se encarga del estudio de las notaciones que permiten especificar la ejecución concurrente de las acciones de un programa, así como resolver los problemas inherentes a la ejecución concurrente (ver 6.2.2). Es de interés formalizar el concepto de ejecución concurrente y de ejecución paralela a fin de poder diferenciarlos:

- Definición 14: Dos hilos [1] son *concurrentes* si la primera instrucción de uno de ellos se ejecuta después de la primera del otro y antes de la última.
- Definición 15: Dos hilos se están ejecutando de manera *paralela* si son concurrentes y la ejecución de ambos se da al mismo tiempo.

Para que dos hilos sean concurrentes no es necesario que se ejecuten al mismo tiempo, es suficiente que exista un intercalado entre la ejecución de sus instrucciones [Pal03]. En este proyecto integrador, es de interés fundamentalmente la ejecución concurrente.

Anteriormente en esta sección se mencionó que existen problemas aparejados a la programación de sistemas concurrentes. Sabiendo esto resulta necesario conocer las ventajas de la programación concurrente, que justifiquen su uso por encima de las dificultades que genera.

6.2.1. Ventajas de la Programación Concurrente

Los beneficios de programar de manera concurrente se engloban en tres categorías:

- Incremento en la velocidad de ejecución: Cuando se ejecuta un programa concurrente en un entorno multiprocesador, los distintos hilos que lo forman se ejecutan de manera paralela, con lo que el tiempo total de ejecución se reduce. Esto es especialmente ventajoso en programas de cálculo numérico.
- Solución de problemas inherentemente concurrentes: Existen problemas cuya naturaleza es concurrente, por lo que un modelo de programación de este tipo se adapta más naturalmente a la resolución de estos problemas.
- Mejor aprovechamiento del tiempo de CPU: Un sistema operativo con un ambiente de multiprogramación que permita la concurrencia es capaz de desalojar a un hilo que está esperando por un evento y no está haciendo uso de la CPU para brindarle este tiempo a otro hilo que lo requiera.

[1] En adelante, se hablará de concurrencia de hilos dado que los sistemas de referencia son de memoria compartida. En el caso que corresponda hablar de procesos se lo mencionará explícitamente.

6.2.2. Problemas y Propiedades de la Concurrencia

Como se introdujo en la sección 6.2, existen problemas que aparecen al programar de manera concurrente. Esto lleva a que los programas concurrentes deban satisfacer una serie de propiedades (además de su especificación técnica del dominio del problema) para funcionar correctamente.

Estas propiedades se dividen en dos grupos:

Propiedades de Seguridad

Las propiedades de seguridad aseguran que "nada malo" va a pasar en la ejecución del programa [Pal03]. Estas son:

- Exclusión Mutua: Existen recursos que no deben ser accedidos concurrentemente para evitar problemas de coherencia y consistencia. Por esto se debe garantizar que a lo sumo un hilo está accediendo a un recurso de este tipo en un instante dado.
- Condición de Sincronización: Existen situaciones donde un hilo debe esperar la ocurrencia de un evento para continuar su flujo. Ante estos casos se debe garantizar que el hilo espere por dicha ocurrencia, de otro modo el resultado puede ser indefinido o inesperado.
- Interbloqueo *(Deadlock)*: Sucede cuando dos o más hilos están esperando a que suceda un evento que nunca ocurrirá para continuar sus flujos de ejecución. El evento no ocurre porque las condiciones para que suceda están bloqueadas por los propios hilos.
  Para que el interbloqueo suceda efectivamente se tienen que cumplir las siguientes condiciones:
  - Exclusión Mutua: si no se exige exlusión mutua, no puede haber interdependencia entre los hilos.
  - Retención y Espera: cada hilo debe retener un recurso y esperar a que se libere otro.
  - No Apropiación: no se puede forzar a un hilo a que desaloje un recurso
  - Circulo Vicioso de Espera: Se forma una cadena cerrada de solicitudes, donde cada hilo retiene al menos un recurso que necesita el próximo hilo.

  Las tres primeras condiciones son necesarias pero no suficientes para que efectivamente ocurra el interbloqueo. La cuarta condición nace como consecuencia de las tres primeras, siempre que se produzca una secuencia de eventos que desemboque en un círculo de espera irresoluble [Sis97].

Propiedades de Vivacidad

Si se aseguran las propiedades de vivacidad, "algo bueno" pasará eventualmente en la ejecución del programa.

- Interbloqueo Activo *(Livelock)*: Se produce un interbloqueo activo cuando un sistema ejecuta una serie de instrucciones sin hacer ningún progreso. Esto se da cuando N hilos necesitan N recursos y se los intercambian sin obtener nunca el conjunto completo.
- Inanición *(Starvation)*: Ocurre cuando al menos una parte del sistema nunca recibe los recursos necesarios para continuar, o los recibe demasiado tarde para lograr el resultado esperado. No es necesario que todo el sistema se bloquee para estar en una situación de inanición.

6.3. Mecanismos de Sincronización

A fin de garantizar el cumplimiento de las propiedades introducidas en la sección 6.2.2, es necesario sincronizar la ejecución de los hilos. De lo contrario, emergen problemas de coherencia y/o consistencia de datos, o corrupción.

6.3.1. Cooperación vs Competencia

La sincronización de hilos se implementa basada en dos principios [Sis97]:

- Cooperación: Los hilos se comunican entre ellos para cooperar en la compartición de recursos. A su vez, existen dos tipos de cooperación:
  - Cooperación por Compartición: Los hilos interactúan para gestionar los recursos. No tienen conocimiento explícito de los demás.

- Cooperación por Comunicación: Los hilos interactúan para gestionar los recursos mediante el paso explícito de mensajes entre ellos.

- Competencia: Los hilos compiten entre sí por los recursos. La gestión de los recursos se efectúa por otra entidad, como el sistema operativo.

6.3.1.1. Cooperación por Compartición de Recursos

Los hilos inteactúan entre ellos sin tener conocimiento explícito de la existencia de los demás.

Existen regiones de almacenamiento de datos compartidas (espacios de memoria, archivos, bases de datos, etc), accedidas por múltiples hilos.

Si bien un hilo no hace referencia a ningún otro, es consciente de que los datos compartidos son accedidos y modificados por los demás. Por lo que el conjunto debe cooperar para asegurar que los datos compartidos se gestionen correctamente. Es responsabilidad de los mecanismos de control asegurar la integridad de los datos compartidos [Sis97].

Como los datos se almacenan en recursos compartidos, existen los problemas de exclusión mutua, interbloqueo e inanición vistos en la sección 6.2.2. La principal diferencia es que existen dos modos de acceder a los datos: para *lectura* y para *escritura*. Únicamente se debe asegurar la exclusión mutua para operaciones de escritura ya que son las únicas capaces de romper la *coherencia* y *consistencia* de los datos.

Un conjunto de datos son coherentes si independientemente de quién haya sido el último escritor, cualquier lector obtiene el último conjunto de valores escrito. Por otro lado, un dato es consistente si un lector obtiene un valor que fue realmente escrito por un escritor y no un dato corrupto.

Algunos mecanismos para gestionar el uso de los datos compartidos son:

- Semáforos: desarrollado en la sección 6.3.2

- Monitores: desarrollado en la sección 6.3.3

6.3.1.2. Cooperación por Comunicación entre Hilos o Procesos

Cuando los hilos o procesos cooperan por comunicación, participan en alcanzar un objetivo en común. La comunicación es una manera de sincronizar o coordinar las distintas actividades.

La comunicación está formada por está formada por el emisor, el receptor, el canal y el mensaje. El envío y recepción de mensajes son explícitos. Las herramientas para este paso de mensajes están dadas por el lenguaje de programación, alguna biblioteca o por el sistema operativo.

Al no haber compartición de datos entre los hilos o procesos, no es necesaria la ejecución en exclusión mutua. Pese a esto, el interbloqueo y la inanición siguen siendo problemas que pueden afectar a los hilos o proceos [Sis97].

6.3.1.3. Competencia entre Hilos

Los hilos no tienen forma de comunicarse entre ellos para gestionar los recursos.

Si dos hilos desean acceder a un mismo recurso, el sistema operativo se lo asignará a uno de ellos y el otro tendrá que esperar. Se debe garantizar:

- La toma de los recursos en exlusión mutua.

- Correcta gestión de los recursos para evitar interbloqueos.

- La reactivación de los hilos bloqueados en un tiempo prudente a fin de evitar su inanición.

El control de la competencia involucra al sistema operativo inevitablemente, porque es él quien asigna los recursos del sistema. Además, los hilos deben ser capaces por sí mismos de expresar de algún modo los requisitos de exclusión mutua, como puede ser bloqueando los recursos antes de usarlos. Cualquier solución conlleva alguna ayuda del sistema operativo, como la provisión del servicio de bloqueo. [Sis97]

> **Nota:** A los fines de este proyecto integrador sólo es de interés la concurrencia basada en memoria compartida. Dentro de este modelo se destacan dos mecanismos de sincronización por competencia: los *semáforos* y los *monitores*.

6.3.2. Semáforos

Los semáforos fueron el primer mecanismo de sincronización de hilos por cooperación. Fueron desarollados por E. Dijkstra en 1965 como mecanismos eficientes y fiables para dar soporte a la cooperación de hilos en un sistema operativo.

El principio en el que se basan es simple. Un conjunto de hilos pueden cooperar utilizando señales, de manera que se pueda obligar a un hilo a detener su ejecución en un punto específico hasta recibir una señal conocida. La señalización está a cargo de los semáforos.

Para transmitir una señal sobre el semáforo s, el hilo p debe ejecutar $signal(s)$, y para recibir una señal de s, debe ejecutar $wait(s)$. Si la señal no fue transmitida, p se bloquea hasta recibir la señal.

Efectivamente, las operaciones sobre los semáforos son tres:

- $init(sem\ s,\ uint\ n)$: inicializa al semáforo s con un entero positivo n.
- $wait(sem\ s)$: decrementa el valor del semáforo. Si se hace negativo, el hilo que realiza la llamada se bloquea. También se la llama *acquire*.
- $signal(sem\ s)$: incrementa el valor del semáforo. Si había un hilo bloqueado por una llamada a $wait(s)$, se desbloquea. También se la llama *release*.

Las llamadas a $signal(s)$ y $wait(s)$ son atómicas para asegurar la modificación del contador del semáforo en eclusión mutua.

Los hilos que esperan una señal luego de bloquearse por una llamada a $wait(s)$ deben hacerlo en una cola de espera. Esta cola implementa una política que decide cuál hilo bloqueado se libera ante la llegada de una señal. El caso más típico es FIFO, pero se puede implementar otro. Sea cual fuera la política implementada, se debe asegurar que ningún hilo bloqueado sufrirá inanición por ella.

Los semáforos descriptos hasta este punto son de tipo *semáforo general*. Existe una versión más reducida que sólo puede adquirir valores 0 y 1 llamada *semáforo binario*. Los semáforos binarios son de implementación más simple que los generales y se demuestra que tienen la misma potencia de expresividad. [Sis97]

6.3.3. Monitores

Los semáforos son herramientas simples y potentes para la gestión de la concurrencia. Permiten gestionar la ejecución en exclusión mutua y coordinar hilos. El problema de los semáforos radica en que las operaciones signal(s) y wait(s) están distribuidas por el código de todos los hilos que lo usan, con lo que resulta muy difícil entender y predecir el efecto de una operación sobre todos los hilos que dependen del mismo semáforo.

Para solucionar este problema, C. Hoare definió el concepto de monitor en su artículo "Monitors: An Operating System Structuring Concept." en 1974.

Los monitores, al igual que los semáforos, son herramientas de gestión de la concurrencia entre hilos. Los hilos los usan para asegurar el acceso en exclusión mutua a recursos y para sincronizar y comunicarse con otros hilos.

El propósito de un monitor de concurrencia es centralizar la gestión de los recursos compartidos en una sección del código del programa. De esta manera, la responsabilidad de sincronizar a los hilos para evitar problemas de concurrencia es enteramente del monitor y no de cada hilo que quiera acceder a un recurso.

Un monitor consiste en un grupo de datos y un conjunto de rutinas exportadas (llamadas *rutinas de entrada*). Estas rutinas realizan operaciones sobre los datos. Los datos del monitor representan recursos compartidos para múltiples hilos (ya sean de software o de hardware) y pueden ser modificados únicamente dentro de las rutinas del monitor.

La forma que tiene un monitor de gestionar concurrencia es:

- Asegurando la ejecución de sus rutinas en exclusión mutua.
- Gestión de los recursos de forma implícita o explícita

Para asegurar la ejecución en exclusión mutua, sólo se permite que un único hilo pueda ejecutar una rutina del monitor a la vez. Este hilo recibe el nombre de *hilo activo*. El hilo activo bloquea la entrada al monitor cuando ejecuta una rutina y la desbloquea cuando cede *voluntariamente* el control del monitor. Si otro hilo llama a una rutina de entrada mientras el monitor está bloqueado, se bloquea en una cola de entrada al monitor hasta que este pase a estado desbloqueado.

6.3.3.1. Sincronización Explícita

En muchas ocasiones resulta necesario no sólo garantizar la exclusión mutua dentro del monitor sino sincronizar hilos dentro de él para la correcta gestión de los recursos compartidos (cola de cortesía). Para esto el monitor representa los recursos con variables de condición (o colas de eventos)

Variables de condición: Son variables especiales, sobre las que se pueden realizar dos acciones:

- delay: Suspende al hilo que la llama, a la espera de una señal.
- signal: Levanta el estado de suspensión de un hilo suspendido por una llamada a *delay()* sobre ella. Si no hay ningún hilo suspendido no tiene efecto.

Si existe más de un hilo suspendido en una variable de condición cuando otro hace una llamada a *signal()*, el hilo a despertar será elegido aplicando una política determinada. Esta política puede ser FIFO, por prioridades por hilo, etc.

El hilo activo del monitor puede suspenderse a sí mismo temporalmente bajo una condición x ejecutando *delay(x)*. Al suspenderse deja de ser el hilo activo y se sitúa al final de la cola de la condición x, a la espera de volver a entrar al monitor cuando la condición cambie. Previo a esto debe desbloquear la entrada al monitor para no generar un interbloqueo con los demás hilos que intentan acceder. Por otro lado, otro hilo que sea el activo puede hacer una llamada a *signal(x)* si detecta un cambio en x, desbloqueando un hilo suspendido en su cola de condición asociada.

Es común asociar una variable de condición a una proposición lógica sobre el estado de un recurso gestionado por el monitor, por ejemplo "El buffer A no está lleno". De esta manera esperar por esta condición equivale a esperar a que el buffer A no esté lleno. Esta asociación suele ser implícita, es decir que le da semántica al monitor pero no forma parte de su funcionamiento.

6.3.3.2. Sincronización Implícita

Como alternativa a la señalización manual, Hoare propone los monitores de señalización automática. Este tipo de monitor elimina las variables de condición modificando la directiva *wait* para que reciba una proposición lógica.

Un hilo que llame *wait(prop)* se mantiene bloqueado mientras la proposición *prop* sea falsa. Cuando *prop* cambie a ser verdadera, los hilos que estén bloqueados se desbloquean automáticamente. Un inconveniente con este mecanismo es que su implementación suele llevar a la señalización repetida de muchos hilos, con los consecuentes cambios de contexto.

A los efectos de este proyecto integrador, es de mayor interés el monitor de sincronización explícita. Por esto, los siguientes apartados al respecto son referidos únicamente a este tipo de monitor.

6.3.3.3. Estructura de un Monitor

De forma general, un monitor de concurrencia de sincronización explícita está compuesto por las siguientes partes:

- Variables de condición
- Colas de condición
- Rutinas exportadas o de entrada
- Cola de entrada
- Cola de espera
- Cola de cortesía o del señalizador

En la figura 6.1 se observa la estructura de un monitor de concurrencia:

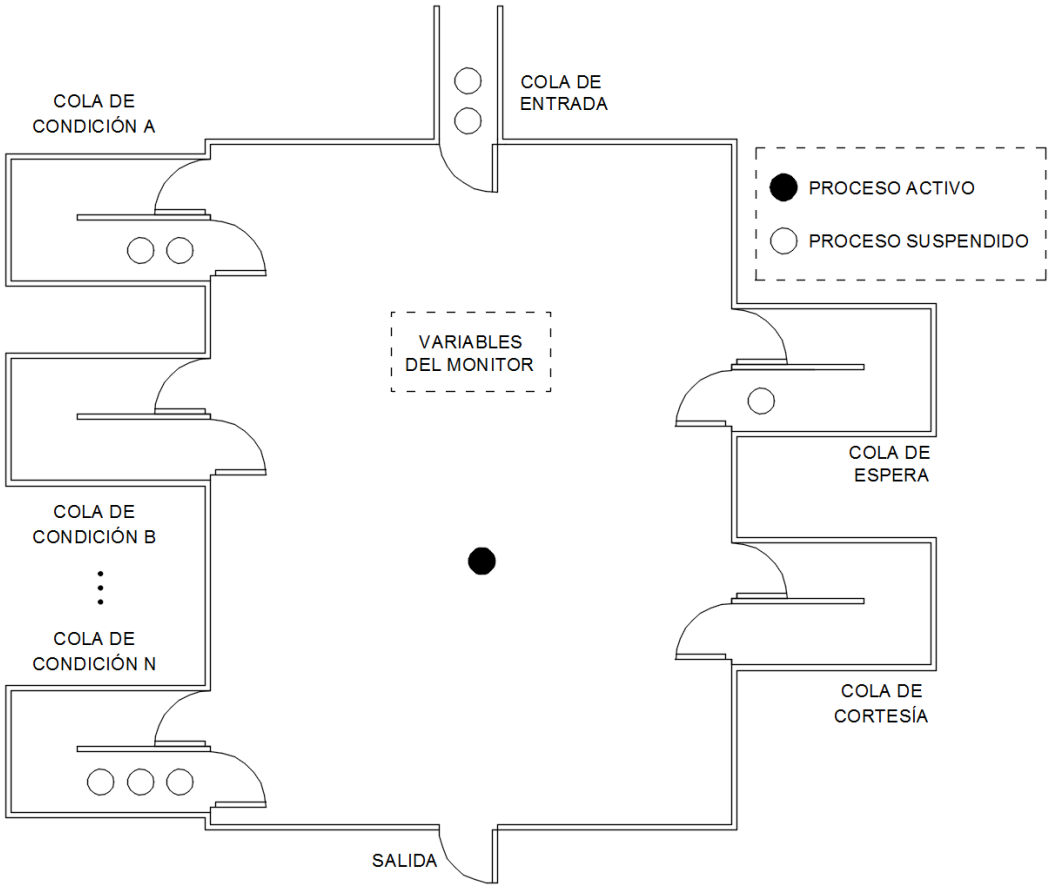


Figura 6.1: Estructura de un Monitor de Concurrencia

Aunque un hilo puede entrar al monitor llamando a cualquiera de sus procedimientos expuestos, y puesto que se debe asegurar la ejecución en exclusión mutua se puede considerar que existe un único punto de entrada al monitor. De ahí que existe una única cola de entrada.

6.3.3.4. Máquina de Estados de un Monitor

Un monitor no es un proceso en sí mismo, por lo que no tiene un hilo de ejecución. En su lugar, es ejecutado por los hilos de los procesos que llaman a alguna de sus rutinas.

El estado del monitor, incluyendo si está o no bloqueado determina si un hilo que intenta ejecutar una rutina de entrada puede continuar o si se bloquea.

Se puede representar el funcionamiento de un monitor por dos máquinas de estado. La primera indica si el monitor está bloqueado o desbloqueado. La segunda representa el estado de las colas del monitor.

- Estados del Primer Autómata:
  - Bloqueado: Un hilo está ejecutando una rutina del monitor
  - Desbloqueado: No hay hilo activo en el monitor
- Estados del Segundo Autómata: Las colas que influyen en el estado del monitor son las internas a este (las de condición, la de espera y la de cortesía). La cola de entrada no influye en el estado del monitor porque no refleja la situación interna del mismo.
  Como los estados que puede adquirir una cola son *"vacía"* y *"no vacía"*, los estados del segundo autómata son todas las combinaciones posibles de las tres colas internas del monitor en cada uno de sus estados.

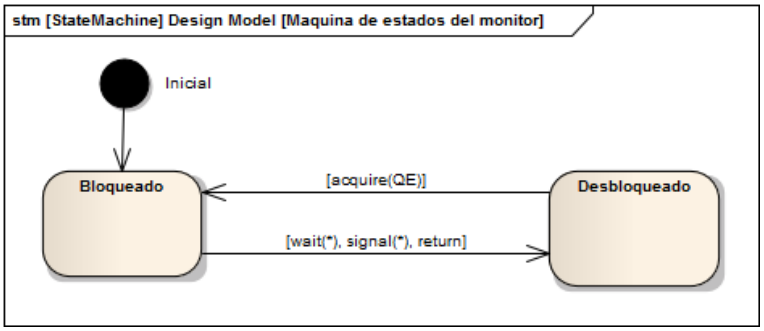


Figura 6.2: Primer Autómata de un Monitor de Concurrencia

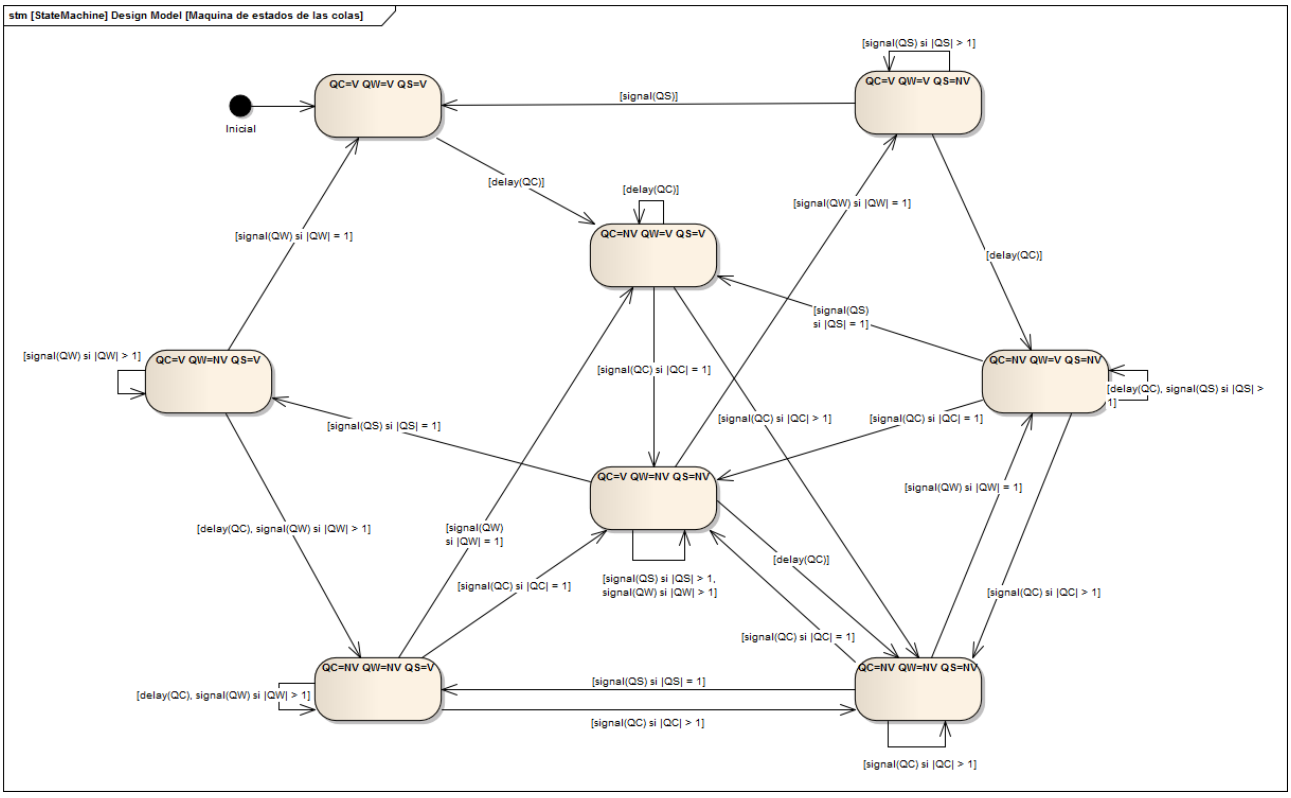


Figura 6.3: Segundo Autómata de un Monitor de Concurrencia

Nota: Como se explicó en la sección 3.4 se puede obtener un único autómata a partir de estos dos, pero es de la opinión de los autores que esto resultaría en una explicación más confusa.

6.3.3.5. Políticas de Desbloqueo de Hilos

El desbloqueo de un hilo suspendido en la cola de condición x debe ser hecho por el hilo que produjo el cambio sobre esta condición. La siguiente acción a realizar luego del desbloqueo dependerá del tipo de monitor en cuestión. Se puede generar una clasificación de monitores basándose en el comportamiento luego del desbloqueo de un hilo. A continuación se presentan los tipos de monitores introducidos en [Pal03]

Nota: Para todos los siguientes casos, se considera que el hilo A desbloquea al hilo B ejecutando *signal(x)*, condición sobre la que B se encuentra bloqueado inicialmente. Por lo tanto, al comenzar cada párrafo, A está bloqueado en la cola de cortesía y B en la de espera, a menos que se indique lo contrario.

Desbloquear y continuar (Signal and Continue) Se desbloquea a A de la cola de cortesía y continúa su ejecución dentro del monitor, ya sea hasta terminar la llamada al procedimiento o hasta bloquearse en una cola

de condición. Una vez A sale del monitor, B ejecuta la instrucción siguiente al *delay(x)* que lo bloqueó. En este punto, B debe volver a verificar la condición que lo suspendió porque no se puede garantizar que A no la haya modificado luego de la llamada a *signal(x)*.

Retorno forzado Se desbloquea a A, quien ejecuta una instrucción de salida del monitor (*return* o *delay(n)*) justo después. De esta manera, no es necesario que B vuelva a comprobar su condición ya que la exclusión mutua asegura que no fue modificada.

Desbloquear y esperar A está en la cola de entrada del monitor en lugar de la de cortesía.
Se desbloquea a B de la cola de espera para que continúe su ejecución en el monitor. Este enfoque tiene la ventaja de que B no necesita comprobar su condición de bloqueo una vez desbloqueado, pero A cede su lugar en el monitor y debe volver a competir por el ingreso para poder terminar su ejecución.

Desbloquear y espera urgente Esta política soluciona el problema de inequidad de *Desbloquear y Esperar*. Se desbloquea a B, pero A se suspende en la cola de cortesía. De esta manera, el desbloqueo de A tendrá prioridad sobre cualquier hilo que intente entrar al monitor.

Clasificación Generalizada de Políticas de Desbloqueo: En [BFC95] el autor hace un análisis más exhaustivo de las posibilidades existentes para diseñar una política de desbloqueo de hilos. Dadas las tres colas de donde se puede elegir un hilo para desbloquear se plantea una prioridad para cada una, resultando en:

- EP: prioridad de la cola de entrada (entry queue priority)
- WP: prioridad de la cola de espera (waiting queue priority)
- SP: prioridad de la cola de cortesía (signaler queue priority)

Asignando pesos relativos a las tres prioridades se llega a que existen 13 distintas posibilidades.

En la tabla 6.1 se Enumeran las posibilidades. La tercera columna se refiere a los monitores definidos en [Howard, J. "Proving Monitors"]

	Prioridades relativas	Monitor Tradicional Correspondiente
1	$EP = WP = SP$	Random
2	$EP = WP < SP$	Wait and Notify
3	$EP = SP < WP$	Signal and Wait
4	$EP < WP = SP$	
5	$EP < WP < SP$	Signal and Continue
6	$EP < SP < WP$	Signal and Urgent Wait
7	$EP > WP = SP$	RECHAZADO
8	$EP = SP > WP$	RECHAZADO
9	$SP > EP > WP$	RECHAZADO
10	$EP = WP > SP$	RECHAZADO
11	$WP > EP > SP$	RECHAZADO
12	$EP > SP > WP$	RECHAZADO
13	$EP > WP > SP$	RECHAZADO

Cuadro 6.1: Tipos de monitores según las prioridades relativas de sus colas

Las propuestas 7 a 13 son rechazadas porque si la prioridad de entrada es mayor que cualquiera de las otras dos, ante un flujo constante de hilos de entrada, habría al menos una cola que nunca sería atendida, lo que lleva a posible inanición de los hilos que esperan en ella.

6.3.3.6. Uso de un Monitor

En el diagrama 6.4 se describen las actividades de un hilo que accede a un monitor.

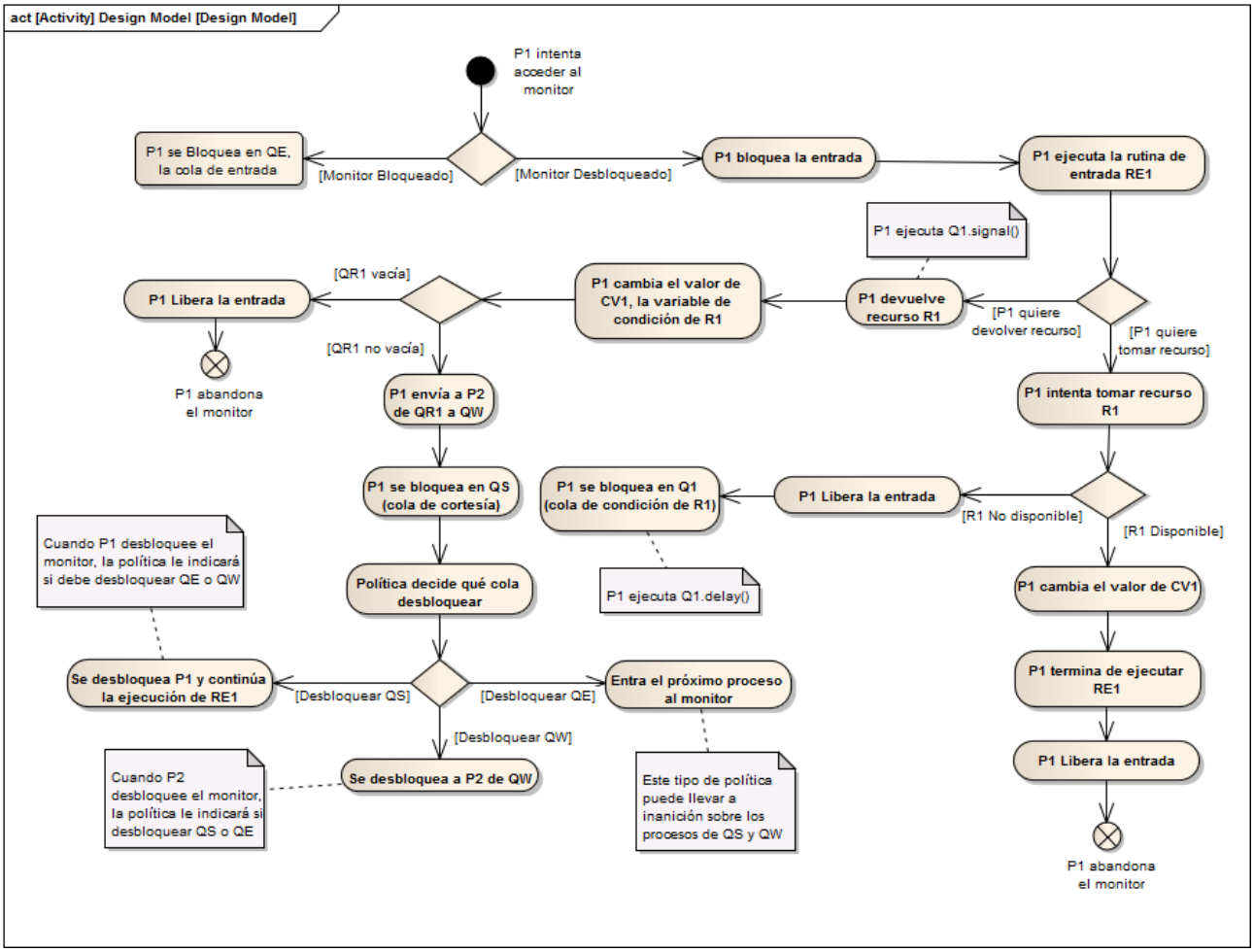


Figura 6.4: Diagrama de actividades UML de un hilo ejecutándo una rutina de un monitor

El diagrama de la figura 6.4 sugiere que un hilo puede optar por uno de dos caminos al ejecutar una rutina del monitor: tomar o devolver un recurso.

Existe otra opción que es tomar y devolver un recurso en la misma rutina. Este caso no está especificado en el diagrama por simplicidad y por tratarse de una superposición de los otros dos casos.

6.3.3.7. Conclusión

La ventaja de los monitores sobre los semáforos es que todas las funciones de sincronización quedan confinadas dentro del monitor. De este modo, es más sencillo verificar que la sincronización se ha realizado correctamente y detectar los fallos. Una vez que un monitor está correctamente programado, el acceso al recurso protegido es correcto para todos los hilos. En el caso de los semáforos, en cambio, el acceso al recurso es correcto sólo si **todos los hilos** que acceden al recurso están correctamente programados. [Sis97] Por otro lado, las políticas de desbloqueo permiten especificar prioridades de ejecución para los hilos, ya sea por orden de llegada o por algún otro criterio. Un monitor diseñado de forma modular para modificar la política brinda la posibilidad de alterar la planificación de los hilos de acuerdo a cada caso particular. La misma planificación resulta por demás complicada si se realiza utilizando semáforos. Por estas razones, un punto fuerte a favor de los monitores frente a los semáforos es la mantenibilidad del código. Por otro lado, al ser el único punto del programa donde se toman decisiones, un monitor se convierte en un cuello de botella para el sistema. El impacto negativo a la performance del sistema se reduce con una arquitectura de monitores jerárquicos.

Parte III

Desarrollo

Capítulo 7

Investigación

7.1. Introducción

En este capítulo se describe el proceso de investigación realizado con el objetivo de determinar las características de la herramienta de software a desarrollar en este Proyecto Integrador.

7.2. Objetivos de la Investigación

Con la intención de definir la orientación del proyecto, se realizó una primera definición aproximada del objetivo principal:
"El sistema a desarrollar es una herramienta para facilitar a desarrolladores de software la utilización de una Red de Petri como lógica de su producto."
En este sentido, se realizó un listado de herramientas candidatas para cumplir con dicho objetivo:

- Generación de código
- APIs
- Frameworks

Se desea que la herramienta resultante posea las siguientes condiciones:

- La lógica del sistema debe quedar expresada en una Red de Petri.
- El grado de acoplamiento entre el código de usuario y la Red de Petri debe ser el mínimo.
- La arquitectura de la herramienta tiene que contemplar el uso de patrones de diseño en el código de usuario.
- El flujo de ejecución debe quedar definido por la herramienta.
- Determinar el grado de escalabilidad de los sistemas desarrollados utilizando la herramienta.
- La herramienta debe favorecer la mantenibilidad del código de usuario.
- En caso de ser posible, analizar experiencias previas de sistemas similares utilizando dichas herramientas.

7.3. Desarrollo de la Investigación

La investigación se basa en el análisis de generación de código, APIs y Frameworks expuestas en el capítulo 5. Además se estudiaron las experiencias previas desarrolladas en [CF14], [AF15] y [BL17]:

- En [CF14] se propone una solución basada en la generación de código. Tras analizar los ejemplos de uso de la herramienta se pudieron detectar desventajas importantes. El código generado utiliza las interfaces del monitor directamente, provocando un alto grado de acoplamiento entre el software del usuario y la Red de Petri. Esto provoca que los sistemas desarrollados tienen reducida escalabilidad y mantenibilidad.
- En [AF15] se propone una solución basada en el desarrollo de un framework como herramienta superadora a la generación de código. Los ejemplos de uso muestran un claro avance respecto a [CF14]. Los sistemas desarrollados son mantenibles y la herramienta permite la utilización de patrones de diseño en las aplicaciónes. Sin embargo, la implementación presenta problemas de acoplamiento a la Red de Petri ya que carece de una capa de abstracción entre eventos físicos y eventos lógicos (ver sección 10.3).
- En [BL17] se desarrolla un sistema dirigido por RdP y se utilizan las interfaces del monitor de Petri como una API o biblioteca de funciones. Si bien se logra obtener un sistema funcional, en la conclusión del proyecto sus autores expresan que es necesario un framework para desacoplar el código de la red. Además el flujo de ejecución es determinado por el desarrollador, aumentando el grado de acoplamiento.

7.4. Conclusión de la Investigación

Tras lo expuesto en la sección 7.3 se determinó que la herramienta a desarrollar debe tener la forma de un Framework por las siguientes razones:

- Funciona como una capa de abstracción entre las acciones de software, los eventos lógicos de la red de Petri, los eventos físicos del mundo exterior al sistema, las secuencias de acciones de software, las políticas de prioridad y los estados de la Red de Petri.

- El flujo de control queda contenido dentro de la estructura del Framework.
- El desacoplamiento facilita la mantenibilidad y legibilidad del sistema.

Capítulo 8

Requerimientos

8.1. Introducción

En este capítulo se hace una definición priorizada de los requerimientos que debe cumplir el framework a desarrollar.

8.2. Prioridades de los requerimientos

Los requerimientos se priorizan utilizando la semántica expuesta en la Tabla 8.1, descripta en [J.05].

Atributo de Prioridad	**Semántica**
Debe Tener	Requisitos obligatorios que son fundamentales para el sistema
Debería tener	Requisitos importantes que se pueden omitir
Podría tener	Requisitos que son opcionales (se realizan si hay tiempo)
Quiere tener	Requisitos que pueden esperar para versiones posteriores del sistema

Cuadro 8.1: Semántica de los Requerimientos

8.3. Definición de Requerimientos

A continuación se definen los requerimientos del framework a desarrollar:

1. El framework debe interactuar con un monitor de Redes de Petri.
2. El framework debe delegar el control del flujo de ejecución en el monitor de Redes de Petri.
3. Para un usuario con conocimiento intermedio [1] en Java y Redes de Petri, el framework podría aprender a usarse en una semana o menos. [2]
   - La utilización del sistema podría incorporar como máximo diez conceptos nuevos a aprender por un usuario con un nivel intermedio de conocimientos en Java y redes de Petri.
   - El sistema debería ser acompañado con al menos dos ejemplos de uso en los cuales se muestre de un mínimo del 80 % de las interfaces del mismo.
4. El sistema debería ser compatible con las versiones actuales de monitores de Petri desarrollados en el Laboratorio de Arquitectura de Computadoras de la Facultad de Ciencias Exactas y Naturales de la Universidad Nacional de Córdoba.
   - El framework quiere tener la posibilidad de elegir el monitor de Petri que desea usar (monitor en Java, monitor en IP Core, monitor en driver, etc.)
   - El framework debe utilizar las interfaces expuestas por el monitor de Petri para hacer manejo de la RdP.
5. El framework debe tener casos de test para probar sus funcionalidades.
   - Las cobertura de los tests debería ser del 80 % de las líneas de código.
   - El framework debe tener al menos un test unitario automatizado por cada funcionalidad implementada.
6. El framework debe tener documentación del código
   - La documentación debe tener un formato estándar (por ejemplo Javadoc).
   - La documentación debe realizarse sobre cada objeto y método del framework.
   - La documentación debe explicar la funcionalidad y la forma de uso del elemento documentado.
   - La documentación debe evitar brindar detalles de implementación.
   - La documentacion debe estar disponible de forma pública.

[1] Se considera *conocimiento intermedio* como experiencia mínima de 6 meses

[2] Se considera que un usuario ha aprendido a utilizar el framework cuando puede generar un programa de funcionalidades mínimas, contando con la documentación del framework a disposición

Capítulo 9

Monitor de Concurrencia con Redes de Petri

9.1. Introduccion

Como el modelo de concurrencia obtenido por las RdP es centralizado, resulta directo ejecutar una RdP como lógica secuencial de un sistema reactivo dentro de un monitor de concurrencia. Luego, para relacionar los eventos con el monitor se utilizan RdP no autónomas. Por lo cual en este apartado desarrollamos un monitor de concurrencia con Redes de Petri.

9.2. Requerimientos del monitor

Del estudio de [DIOM16], [CF14], [AF15] y [BL17] emergen los requerimientos del monitor:

1. Debe ofrecer interfaces para la carga de una Red de Petri en formato estándar PNML (dialecto Tina).
2. Debe soportar Redes de Petri ordinarias y temporales.
3. Debe ofrecer interfaces para el disparo de transiciones.
4. Debe implementar la ecuación de estado generalizada descrita en [DIOM16] para la ejecución de RdP.
5. Debe soportar las etiquetas de transición definidas en [CF14] (transiciones automáticas/disparadas y transiciones informadas/no informadas).
   Esto implica los siguientes requerimientos:
   a) Debe ofrecer interfaces para la suscripción a informes de transiciones informadas.
   b) Debe prohibir el disparo manual de transiciones automáticas.
   c) Debe disparar automáticamente transiciones automáticas sensibilizadas.
6. Debe ofrecer interfaces para la carga de políticas de prioridad de disparo de transiciones.
7. Debe brindar alguna forma de asociar guardas con transiciones.
8. Debe ofrecer interfaces para la modificación del valor de guardas.
9. Debería soportar disparos perennes y no perennes.
10. Debería soportar arcos normales, de reset, lectores e inhibidores.

9.3. Java Petri Concurrency Monitor

Java Petri Concurrency Monitor es un monitor de concurrencia que ejecuta Redes de Petri, hecho en lenguaje de programación Java. Provee al usuario de una interfaz de programación de aplicaciones (API) para ejecutar una RdP, pretegiéndola de los problemas de concurrencia con la exclusion mutua del monitor.

9.4. Diseño y Funcionamiento

En primera instancia se analizó la reutilización y expansión de las funcionalidades del monitor construido en el desarrollo de [CF14], y reutilizado en [AF15]. Este software no satisface los requerimientos, por lo cual es necesario diseñarlo y construirlo completamente en este proyecto integrador.

9.4.1. Arquitectura de Alto Nivel

Según las clasificaciones vistas en las secciones 6.3.3.1 y 6.3.3.5, Java Petri Concurrency Monitor (JPCM) es un monitor de sincronización explícita y aplica una política de desbloqueo de hilos de retorno forzado. Teniendo en cuenta los requerimientos expresados en el apartado anterior se determinan los principales componentes de JPCM:

- PetriNet: contiene la información de la RdP a utilizar y la lógica de la ejecución
- PnmlParser + PetriNetFactory: su responsabilidad es convertir la información del archivo que describe a la RdP en un formato ejecutable para JPCM.
- PetriMonitor: es el monitor en sí mismo. Expone las interfaces de programación y tiene la responsabilidad de gestionar los hilos, para evitar la ejecución concurrente de la RdP.
- TransitionsPolicy: representa la política de gestión de los recursos del monitor. No es la misma política analizada en la sección 6.3.3.5. Esta política permite decidir cuál transición, de un conjunto dado, es la próxima a ser disparada.

La figura 9.1 es un diagrama de arquitectura de JPCM, mostrando sus principales bloques e interacciones.

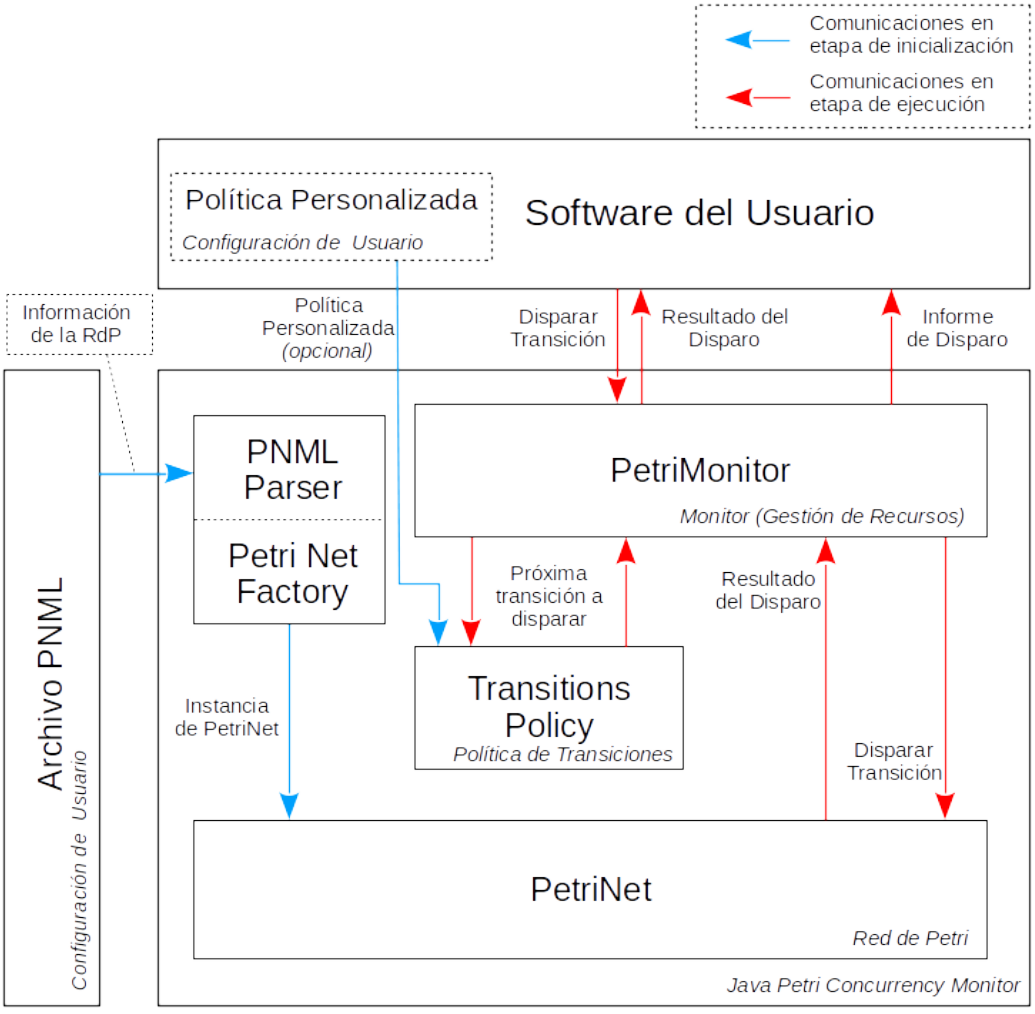


Figura 9.1: Arquitectura de JPCM

9.4.2. Gestión de los Recursos con RdP

En la sección 6.3.3 se analiza cómo se gestionan los recursos de un monitor con variables de condición en un monitor de sincronización explícita. En dicha sección se explicó cómo un hilo que desea tomar o devolver un recurso debe señalizar a una variable de condición, y bloquearse en caso de no tener el recurso disponible.

Por otro lado, en la sección 3.3.7 se explica cómo una RdP representa procesos. De esta manera, para ejecutar un proceso A, un hilo debe señalizar el comienzo y fin de la ejecución en la RdP disparando un subconjunto de transiciones de inicio y otro subconjunto de transiciones de finalización.

De esta forma se establece una relación entre las operaciones de una variable de condición y los disparos de una transición. Se logra obtener el mismo comportamiento sobre los hilos que las acceden con la ayuda de colas auxiliares.

El intento de disparo sobre una transición no sensibilizada es equivalente a una llamada a *signal()* sobre una variable de condición (ver sección 6.3.3.1). De la misma manera, el disparo de una transición que resulta en la sensibilización de otra, es equivalente a una llamada a *signal()* sobre una variable de condición.

Por inducción podemos decir que una variable de condición verdadera equivale a disparar una transición y una variable de condición falsa equivale a no poder disparar una transición.

A aprtir de esta semejanza queda en evidencia que las operaciones a realizar dentro de un monitor que ejecuta una RdP consisten consisten del disparo de transiciones para evolucionar el estado de los recursos y operaciones del sistema modelado.

9.4.3. Estructura Interna de JPCM

Basándose en el diagrama de la figura 9.1, se puede dividir a JPCM en dos secciones:

- Modelo: Tiene como eje central a la clase *PetriNet*. Dentro de esta sección está el modelo a ejecutar. Contiene las matrices de la RdP (parte estática del modelo) y al vector de estado (parte dinámica). Define y expone el método de cambio de estado (disparo de una transición).
- Conducción: Tiene como eje central a la clase *PetriMonitor*. Se encarga de conducir a los hilos que ejecutan el modelo. Incluye clases auxiliares para implementar las políticas del monitor (de transiciones y de colas).

9.4.3.1. Sección Modelo

En el diagrama de la figura 9.2 se observan las clases que componen a esta sección, sus relaciones y colaboraciones:

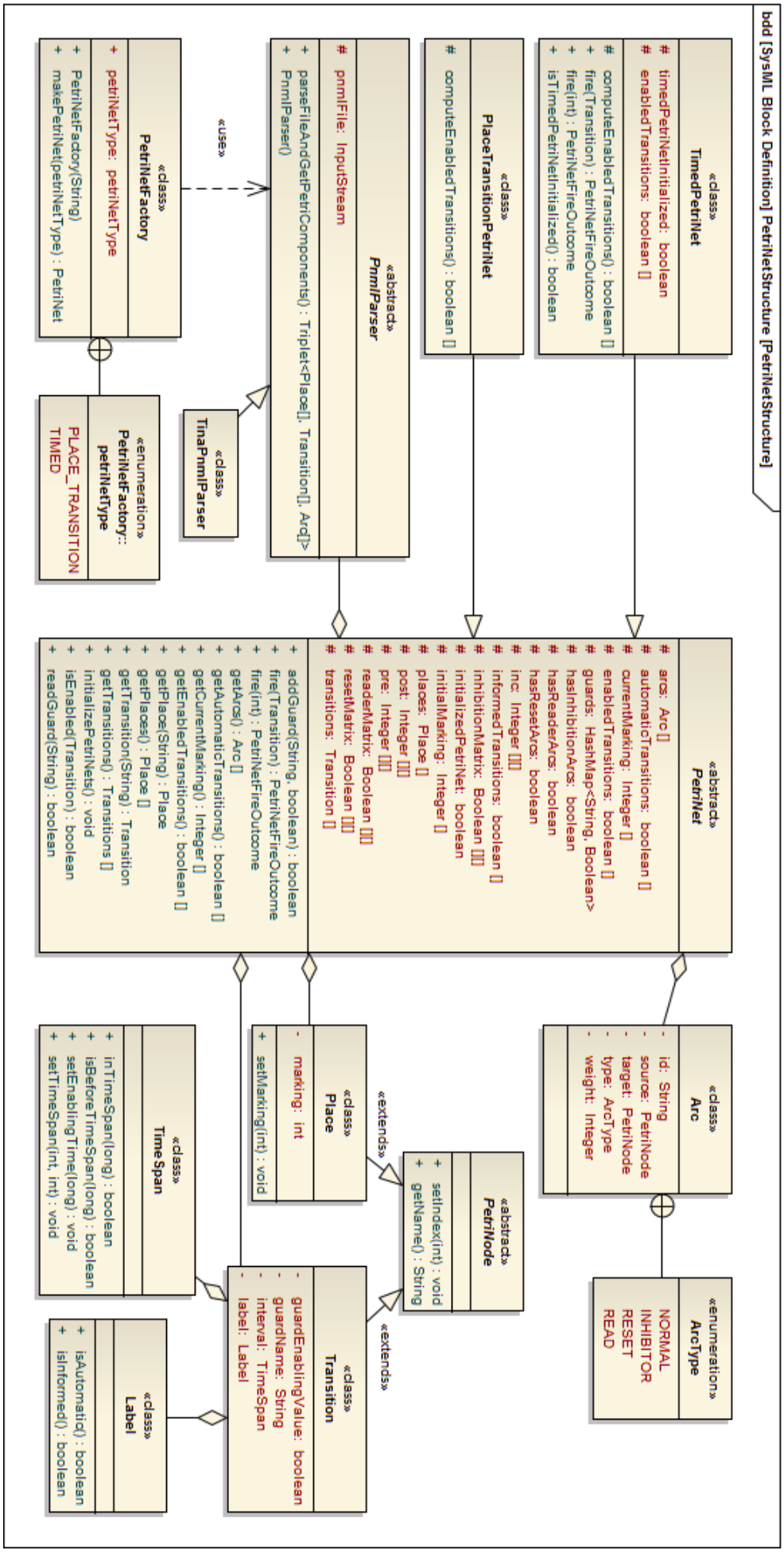


Figura 9.2: Diagrama de clases de la sección *Modelo*

Se observan las clases que modelan a los componentes de una RdP estructural (ver sección 3.2.1) (*Arc*, *Transition*, *Place*), la especializaciones concretas de *PetriNet* (*PlaceTransitionPetriNet* para RdP plaza-transición y *TimedPetriNet* para RdP temporales), las clases *PetriNetFactory* y *PnmlParser* analizadas en la sección 9.4.1 y la especialización de *PnmlParser* que comprende el dialecto PNML de TINA, *TinaPnmlParser*. Además se observan las relaciones entre estas clases que colaboran entre sí.

9.4.3.2. Sección Conducción

En el diagrama de la figura 9.3 se observan las clases que componen a esta sección, sus relaciones y colaboraciones:

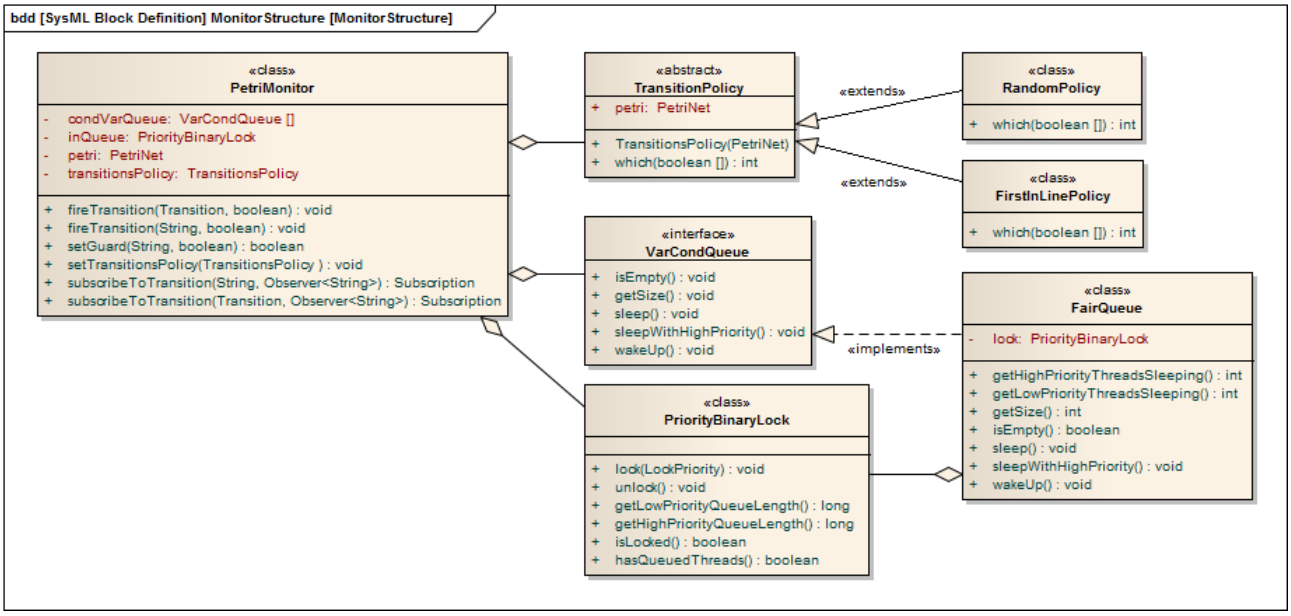


Figura 9.3: Diagrama de clases de la sección *Conducción*

Junto a la clase *PetriMonitor* se puede ver por un lado la subsección que aplica la política de transiciones. Esta subsección contiene la clase abstracta *TransitionsPolicy*, cuya responsabilidad es elegir cuál transición disparar dado un conjunto de transiciones sensibilizadas. Se proveen dos especializaciones:

- *RandomPolicy*: Política aleatoria. Basa su decisión en un generador de números aleatorios.
- *FirstInLinePolicy*: Política de primer transición en la lista. Elige siempre a la primer transición sensibilizada que encuentre en la lista.

Por otro lado, la clase *PriorityBinaryLock* implementa la política de colas por prioridad (ver sección 9.4.8) y es utilizada tanto para la cola de entrada como para las colas de condición.

Las colas de condición asociadas a las transiciones (ver sección 9.4.2) están descriptas en la interfaz *VarCondQueue* e implementadas en la clase *FairQueue*.

9.4.4. Interfaces de Programación

JPCM ofrece una interfaz de entrada al monitor y dos de salida en tiempo de ejecución, y dos interfaces de carga de datos de configuración en tiempo de inicialización.

En tiempo de ejecución, la interfaz de entrada es el método público `PetriMonitor.fireTransition()`. Este método permite disparar una transición de la RdP dentro del monitor de concurrencia, ya sea utilizando el objeto *Transition* (transición) de la RdP, o el nombre de la misma.

Las interfaces salida son dos:

- Retorno del disparo de una transición: la finalización de ejecución del método `PetriMonitor.fireTransition()` asegura que el hilo que hizo la llamada, efectivamente disparó de la transición deseada (excepto en disparos no-perennes donde no se desea esta garantía, ver sección 9.5.6)
- Informes de disparo: ante el disparo de una transición informada, se emite un evento con información sobre la transición disparada. Todos los observadores suscritos a estos eventos lo recibirán.

En tiempo de inicialización se proveen dos interfaces al usuario:

- Carga de una RdP: Se hace mediante un archivo descriptor en formato PNML. El bloque *PnmlParser + PetriNetFactory* lo utiliza para generar una RdP ejecutable.
- Carga de una política personalizada: Es opcional. Permite al usuario definir una política de transiciones que se ajuste al problema a resolver por su software.

Estas interfaces proveen al usuario de los mecanismos necesarios para inicializar el monitor, y para luego utilizarlo.

9.4.5. Inicialización de JPCM

Los pasos para inicializar JPCM son:

- Instanciar la clase *PnmlParser* con la ruta al archivo de descripción de la RdP a utilizar
- Instanciar la clase *PetriNetFactory* con el objeto *PnmlParser*
- Utilizar el objeto *PetriNetFactory* para generar una instancia de *PetriNet*
- Instanciar una política (clases *FirstInLinePolicy* o *RandomPolicy* o la desarrollada por el usuario)
- Instanciar la clase *PetriMonitor* con el objeto *PetriNet* y el objeto de la política
- Crear los hilos que van a ejecutar las acciones del sistema
- Inicializar la RdP llamando a `PetriNet.initializePetriNet()` sobre la instancia de *PetriNet*
- Lanzar los hilos creados anteriormente
- Evitar que el hilo principal termine antes que los hilos trabajadores

Luego de realizar todas estas tareas, el sistema se ejecuta guiado por la RdP.

9.4.5.1. Generación de una RdP Ejecutable a Partir de un Archivo PNML

Como se dijo anteriormente, JPCM requiere de un archivo PNML con la descripción de la RdP a utilizar. Los pasos a seguir para obtener una RdP ejecutable a partir del archivo PNML son los siguientes:

- Una instancia de PnmlParser analiza el archivo para obtener los componentes de RdP contenidos en él (Plazas, Arcos y Transiciones)
- Con la información obtenida genera objetos de tipo *Place*, *Transition* o *Arc* dependiendo del caso
- Ordena los objetos generados en tres arrays, uno por cada tipo de componente y los empaqueta en una tupla
- Cuando se llama a `PetriNetFactory.makePetriNet()` sobre la instancia de *PetriNetFactory*, ésta obtiene la tupla de arrays de componentes generada por *PnmlParser*
- Con los objetos componentes de la RdP, *PetriNetFactory* calcula y almacena las matrices de precedencia, pos-incidencia, incidencia, inhibición, reset y lectura, y el vector del marcado inicial
- Con los objetos componentes, las matrices de la RdP y el tipo de RdP a generar, crea una instancia de la subclase de *PetriNet* correspondiente (*TimedPetriNet* para RdP temporales y *PlaceTransitionPetriNet* para RdP ordinarias)

El objeto resultante es una RdP ejecutable que se utiliza en conjunto con el monitor.

9.4.6. Disparo de una Transición en JPCM

El disparo de una transición en el monitor mediante la ejecución del método `PetriMonitor.fireTransition(t)` desencadena las siguientes acciones sobre el hilo que realiza la llamada:

- Verifica que la transición t no sea automática: En cuyo caso falla con un error explicando la situación
- Verifica que la red esté inicializada: En caso contrario falla con un error explicando la situación
- Toma el lock sobre la entrada del monitor: Si no lo puede tomar se bloquea en la cola de entrada hasta poder tomarlo

- Dispara la transición en la RdP: Devuelve un código de estado indicando el resultado (ver sección 9.4.6)
- Maneja el resultado del disparo: se analiza si se debe liberar el lock de entrada, disparar una transición automática, liberar un hilo bloqueado o bloquearse por una condición
- Libera el lock de entrada: Sólo si es necesario. Algunas situaciones requieren que no se permita la entrada de un nuevo hilo, como la liberación de un hilo bloqueado en una cola de condición

La figura 9.4 es un diagrama de secuencias donde se muestra el flujo de un hilo que realiza un disparo de una transición:

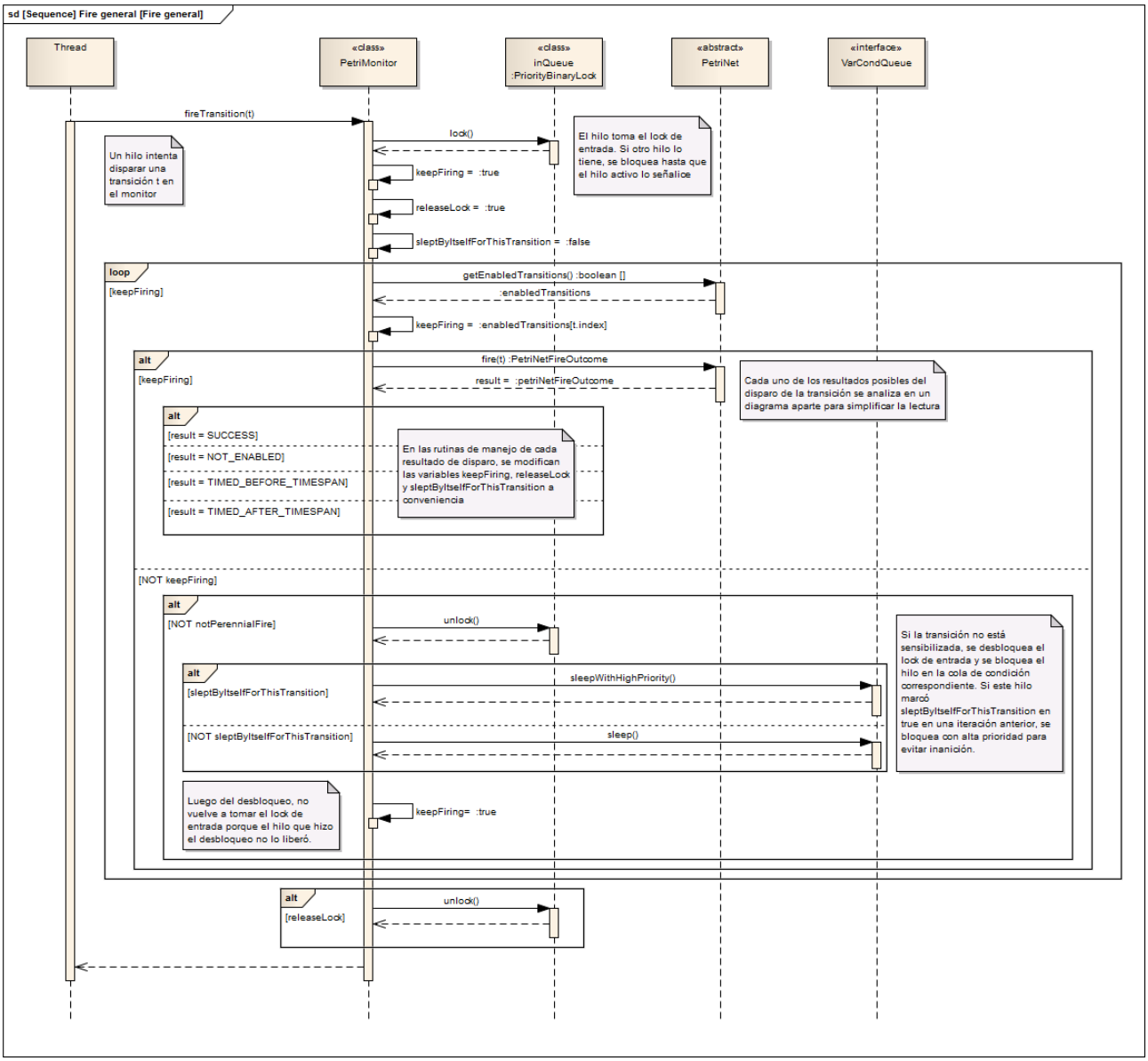


Figura 9.4: Disparo de una transición

Manejo del código de estado del disparo

Cuando se realiza el disparo efectivo de la transición en la Red de Petri, ésta devuelve un código de estado, que es el resultado del disparo. Estos pueden ser:

- *SUCCESS:* el disparo fue exitoso
- *NOT_ENABLED:* la transición no está sensbilizada
- *TIMED_BEFORE_TIMESTAMP:* Sólo para transiciones temporales. El instante de disparo es anterior al *instante menor de disparo* (α) de la transición (ver sección 3.3.3.2)
- *TIMED_AFTER_TIMESTAMP:* Sólo para transiciones temporales. El instante de disparo es posterior al *instante mayor de disparo* (β) de la transición (ver sección 3.3.3.2)

Caso del disparo exitoso

Ante un disparo exitoso, la llamada a `PetriNet.fire(Transition t)` devuelve un código *SUCCESS*. Una vez que esto ocurre, el hilo que realizó el disparo debe:

- Emitir un evento para todos los suscriptores (en el caso que la transición sea informada).
- Determinar si existen nuevas transiciones sensibilizadas producto del último disparo
  - Si no existen, libera el lock de entrada y abandona el monitor
  - Si existen:
    - Selecciona a la próxima transición t_n a ser disparada, de acuerdo con la política de transiciones.
    - Si t_n es de tipo *automática* el disparo lo hace el mismo hilo, por lo que itera para realizar un nuevo disparo sobre dicha transición.
    - Si t_n es de tipo *disparada* se debe señalizar al hilo que esté esperando en su cola de condición y abandonar el monitor inmediatamente sin liberar el lock de entrada.

En el diagrama de secuencias de la figura 9.5 se observa en detalle el flujo de un disparo exitoso de una transición:

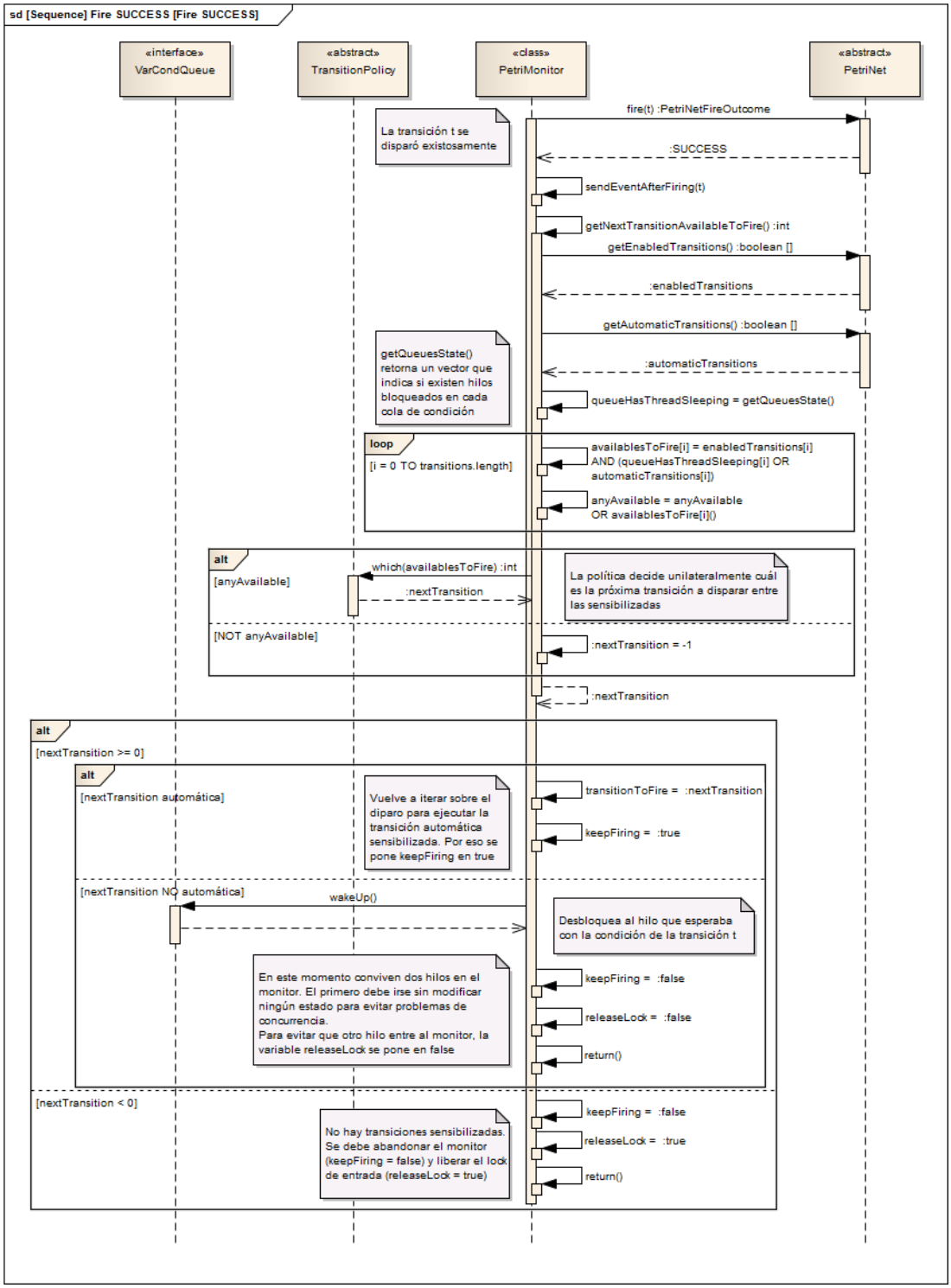


Figura 9.5: Manejo del disparo exitoso de una transición

Caso del disparo no exitoso

Cuando la transición no está sensibilizada, su disparo devuelve *NOT_ENABLED*. Ante este caso, un hilo que realizó un disparo perenne debe esperar en la cola de condición asociada a la transición. A su vez, si en una iteración anterior realizó una espera temporal por la transición, debe bloquearse con alta prioridad.

En el diagrama de secuencias de la figura 9.6 se muestra detallado el caso del disparo no exitoso.

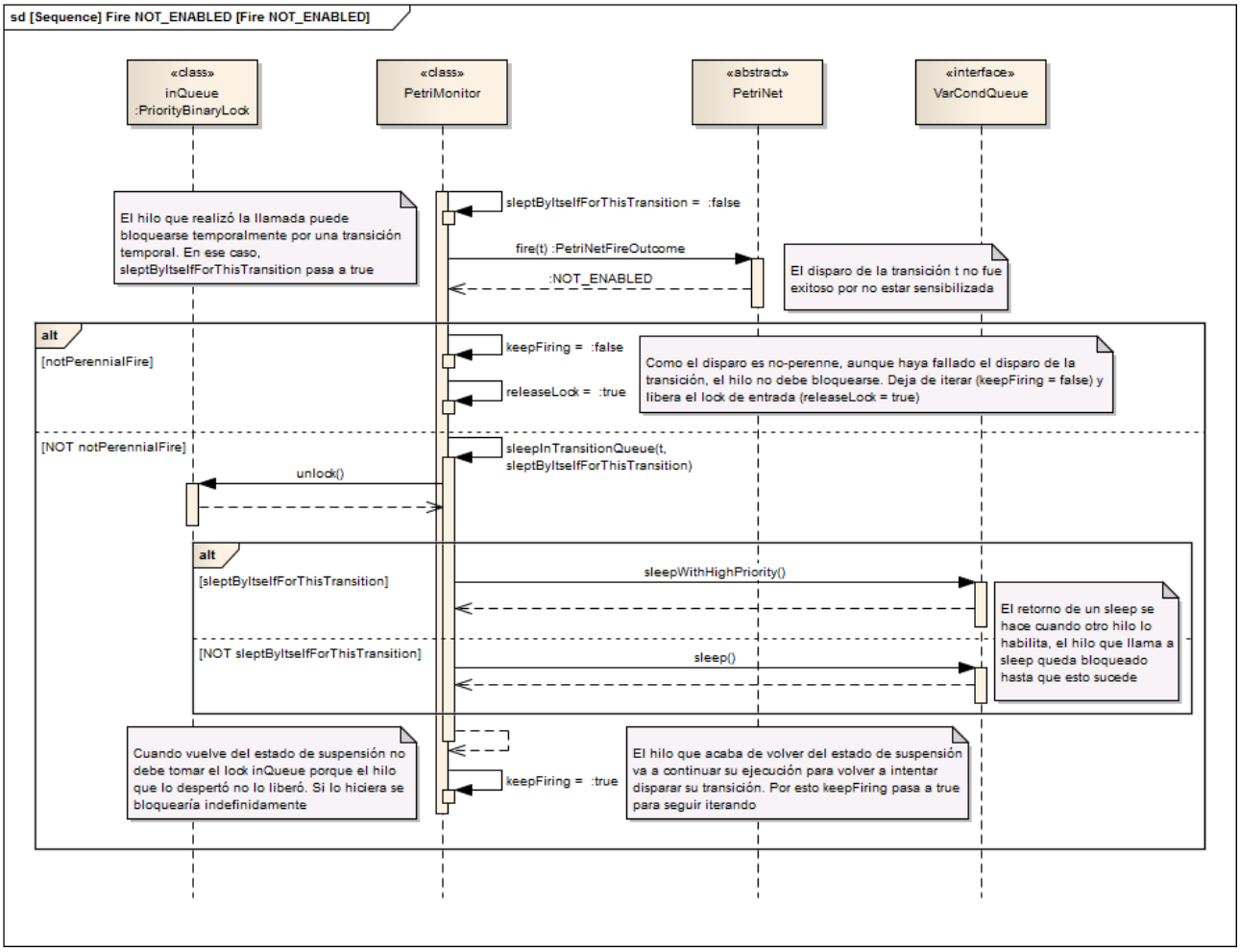


Figura 9.6: Manejo del disparo no exitoso de una transición

Caso del disparo temporal antes del intervalo

Si un hilo intenta disparar una transición temporal antes del instante menor de disparo, la llamada a `PetriNet.fire(t)` devuelve un código de estado *TIMED_BEFORE_TIMESPAN*. En este caso, el procedimiento a seguir es el siguiente:

- Si no hay hilo bloqueado por t:
  - Liberar el lock de entrada
  - Esperar temporalmente hasta el *instante menor de disparo*
  - Tomar el lock de entrada con alta prioridad (ver sección 9.4.7)
  - Señalizar que ya se suspendió temporalmente para evitar futuras inversiones de prioridad
  - Iterar sobre el disparo para reintentarlo
- Si existe un hilo bloqueado por t:
  - Si el disparo es no-perenne
    - Liberar el lock de entrada
    - Abandonar el monitor
  - Si el disparo es perenne
    - Liberar el lock de entrada
    - Bloquearse en la cola de condición de t. Si ya se suspendió temporalmente por t antes, lo hace con alta prioridad (ver sección 9.4.7)
    - Cuando se desbloquea itera para reintentar el disparo

En el diagrama de secuencias de la figura 9.7 se detalla el flujo de este caso de resultado de disparo.

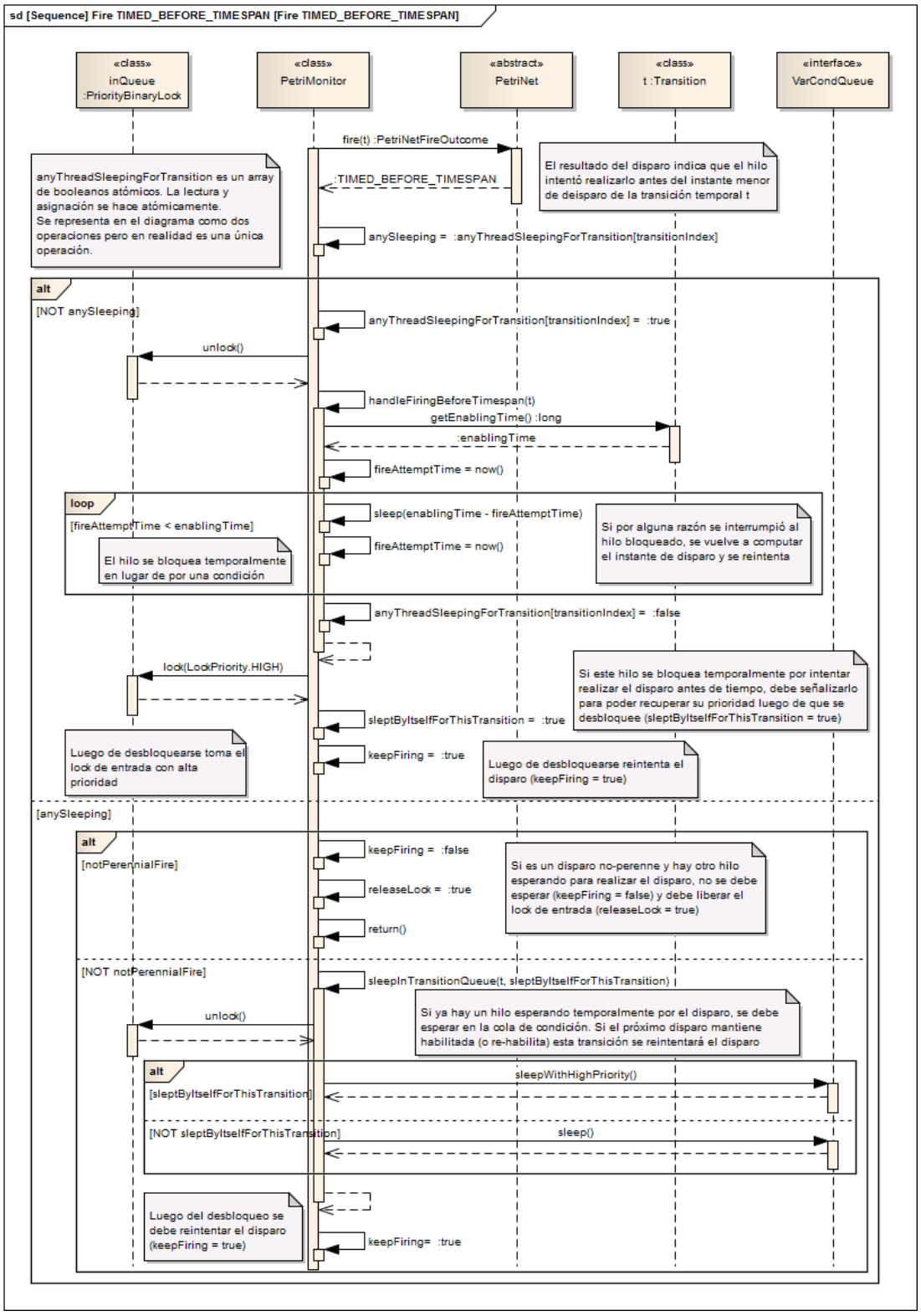


Figura 9.7: Manejo del disparo de una transición temporal antes del instante menor de disparo

Caso del disparo temporal después del intervalo

Ante el intento de disparo de una transición temporal en un instante posterior al instante mayor de disparo, el resultado es *TIMED_AFTER_TIMESPAN*. Al no cumplirse la condición de sensibilización temporal, este intento de disparo es equivalente al caso del disparo no exitoso. Por esto, la rutina de manejo de ambos casos es la misma y correponde al diagrama de secuencias de la figura 9.6.

9.4.7. Problema de la Inversión de Prioridades

Durante el desarrollo de JPCM se detectaron dos casos donde se produce una inversión de prioridades entre los hilos gestionados. A continuación se analizan estos casos.

9.4.7.1. Inversión de prioridades en la cola de entrada

Existe inversión de prioridad en la cola de entrada si ocurren los siguientes eventos en orden:

- Se sensibiliza la transición temporal t_0 con intervalo $[a, b]$ en el instante t_i
- El hilo th_0 intenta disparar t_0 en $t_d < (t_i + a)$
- El hilo th_0 libera la entrada y se bloquea temporalmente
- Entra otro hilo a realizar un disparo y bloquea la entrada
- Llegan al monitor N hilos a intentar realizar disparos y se encolan a la entrada
- Cuando th_0 se desbloquea, intenta tomar el lock de entrada y va al final de la cola

En este caso, th_0 cedió su prioridad a los N hilos que llegaron después que él. Esto provoca que el intervalo de disparo termine antes de que th_0 recupere su turno dentro del monitor, provocando su inanición.

La figura 9.7 explica esta secuencia. Está formada por una sucesión de subfiguras, una por cada instante de interés. Los elementos que aparecen en todas ellas son:

- El monitor de concurrencia (rectángulo mayor)
- La cola de entrada
- Las colas de condición
- Una Red de Petri con tres plazas y cuatro transiciones, donde t_0 es temporal con intervalo $[\alpha, \beta]$
- Un reloj que indica en instante de cada subfigura
- Un área donde se simboliza el bloqueo temporal de un hilo (rectángulo redondeado con líneas de punto)
- Uno o más hilos. El hilo rojo es el de interés

Las subfiguras grafican los siguientes eventos:

a) La transición t_0 se sensibiliza en el instante t_i. Luego, el hilo th_0 llama a `petriMonitor.fireTransition("t0")`.

b) Como no hay hilo activo en el monitor y la cola de entrada está vacía, th_0 toma el lock de entrada e intenta disparar t_0. Mientras tanto se encolan algunos hilos en la cola de entrada

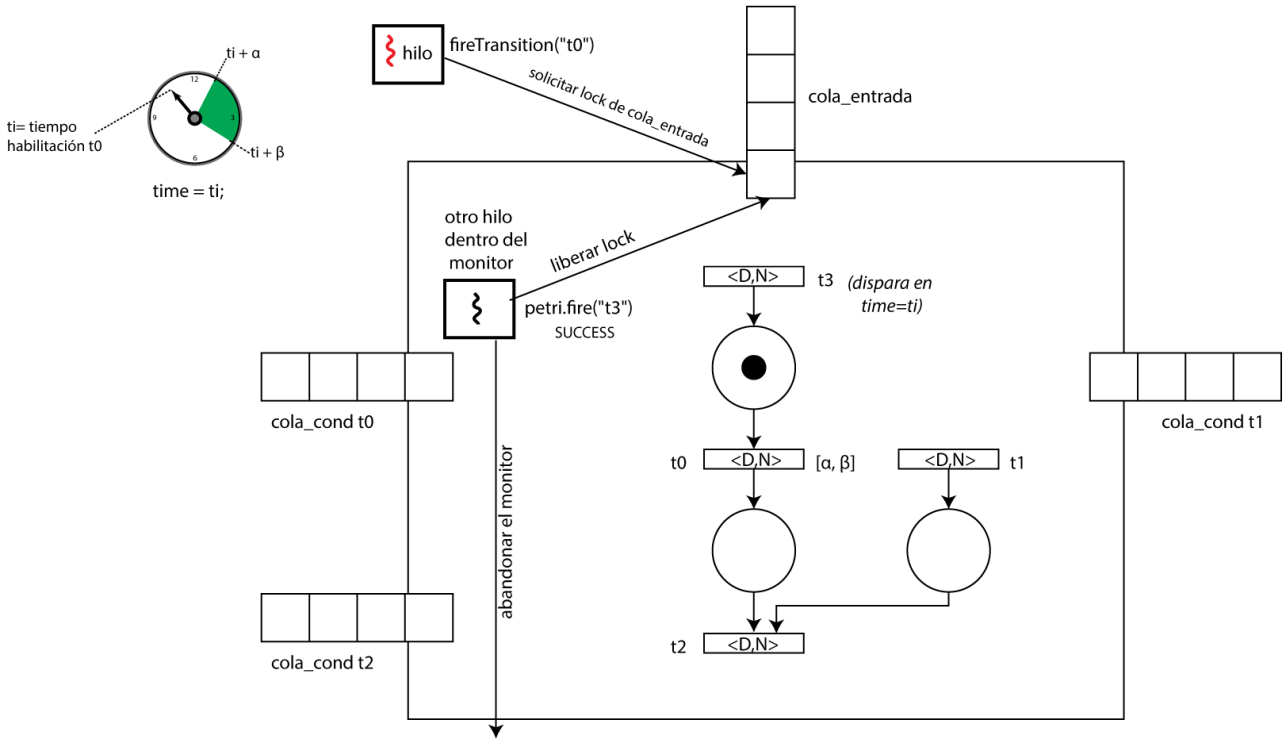
c) La llamada a `petri.fire("t0")` retornó un código `TIMED_BEFORE_TIMESPAN`, por lo que th_0 se bloquea temporalmente hasta el instante $t_i + a$.

d) Antes de bloquearse, th_0 libera la entrada al monitor. Otro hilo ingresa al monitor a disparar a t_1

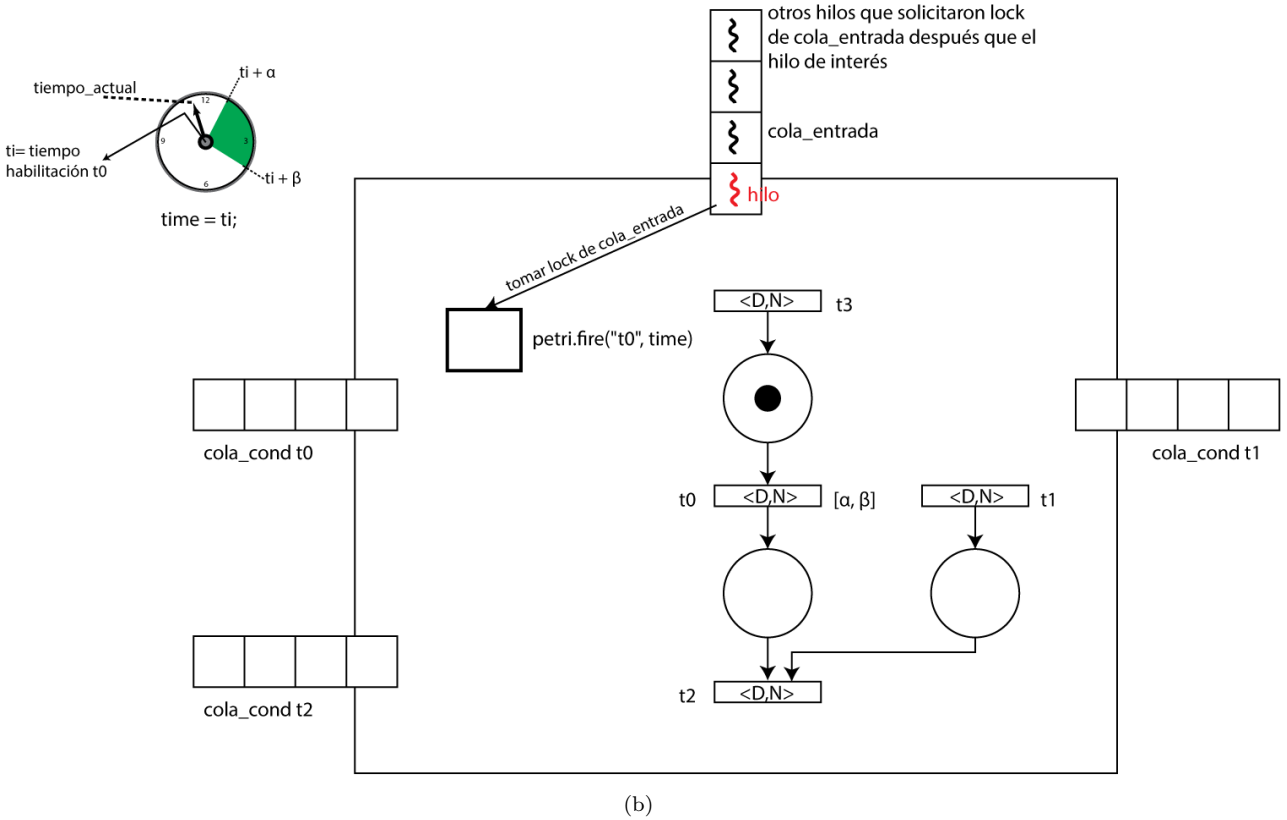
e) Se alcanza el instante $t_i + a$ por lo que th_0 se desbloquea y reintenta el disparo. Como la cola de entrada no está vacía, se encola al final

f) Otro hilo ingresa al monitor a disparar a t_2

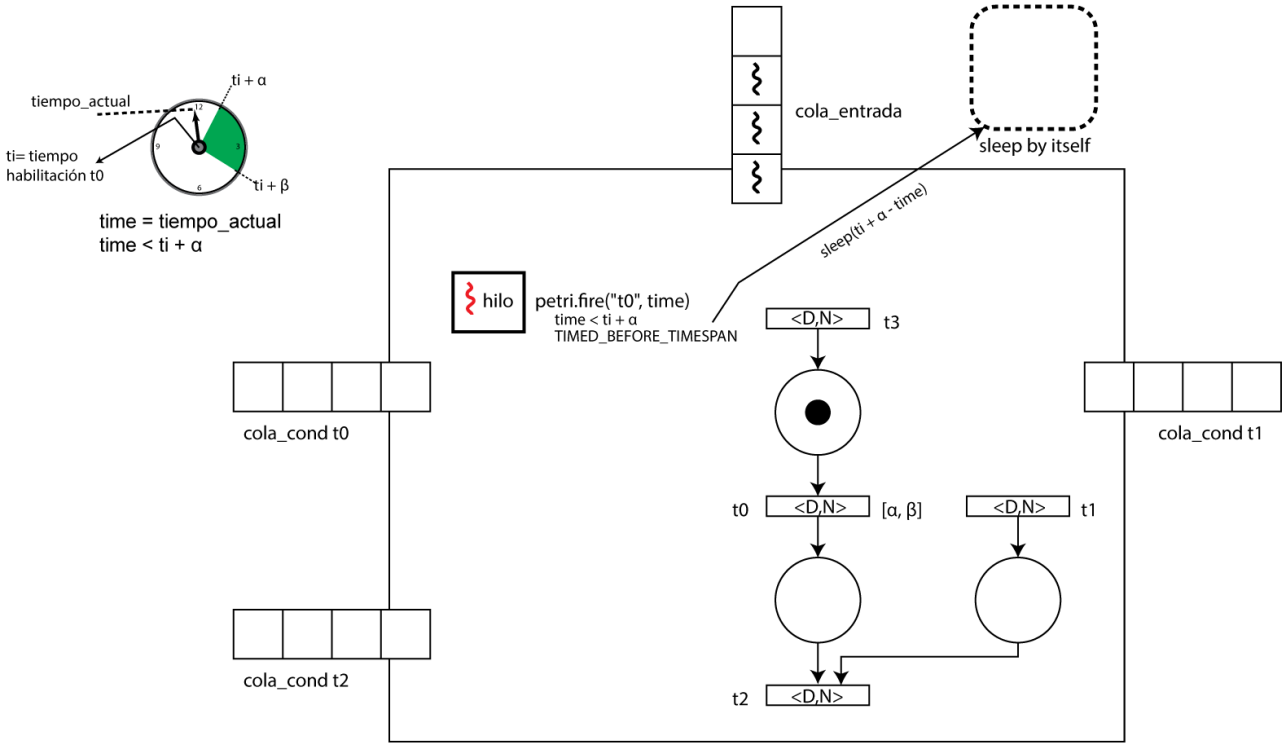
g) Se superó el instante $t_i + b$ (fin del intervalo de disparo de t_0) sin que th_0 haya podido hacer el disparo, provocando su inanición


ti + α
ti= tiempo
habilitación t0
ti + β
time = ti;
hilo
fireTransition("t0")
solicitar lock de cola_entrada
cola_entrada
otro hilo
dentro del
monitor
liberar lock
petri.fire("t3")
SUCCESS
<D,N>
t3
(dispara en
time=ti)
cola_cond t0
cola_cond t1
t0
[α, β]
t1
abandonar el monitor
cola_cond t2
t2

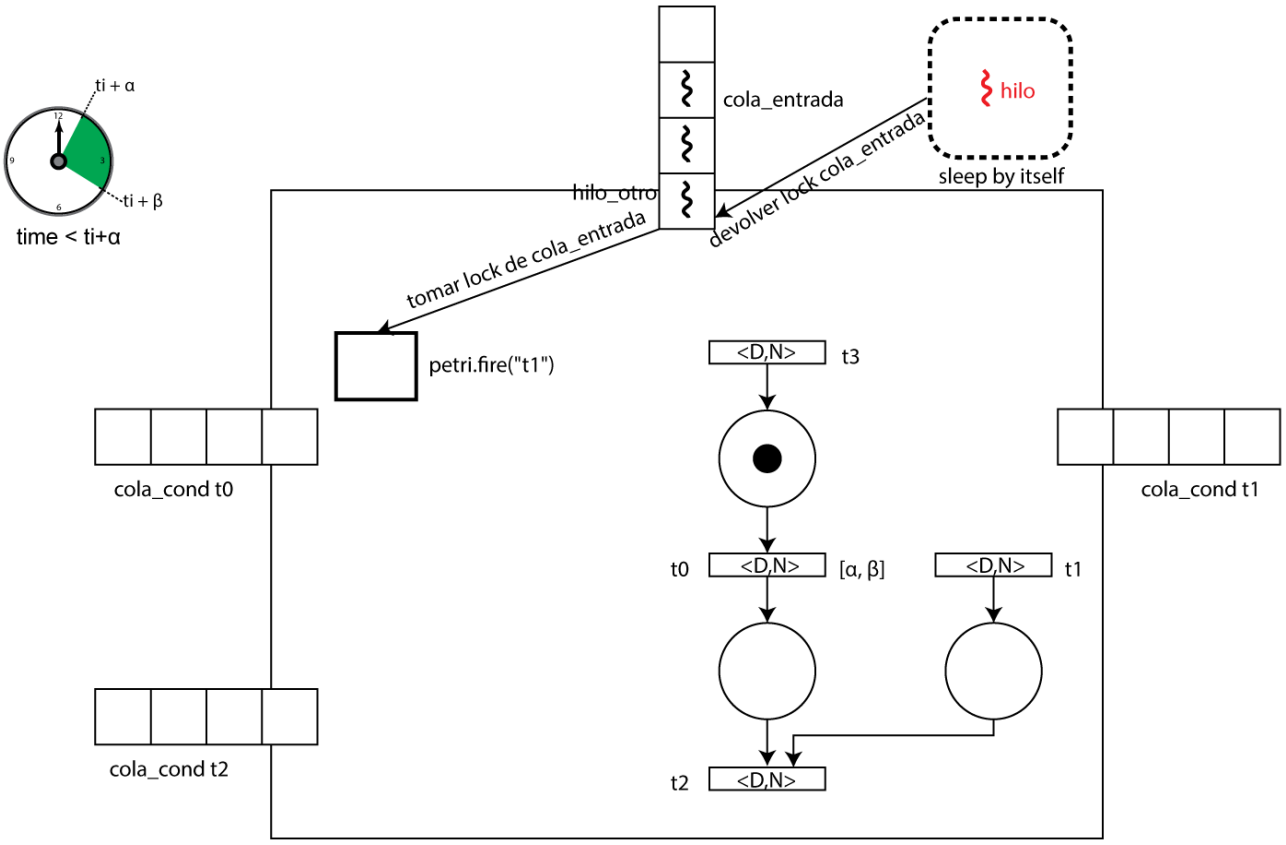
(a)


otros hilos que solicitaron lock
de cola_entrada después que el
hilo de interés
cola_entrada
hilo
tiempo_actual
ti + α
ti= tiempo
habilitación t0
ti + β
time = ti;
tomar lock de cola_entrada
petri.fire("t0", time)
<D,N>
t3
cola_cond t0
cola_cond t1
t0
[α, β]
t1
cola_cond t2
t2

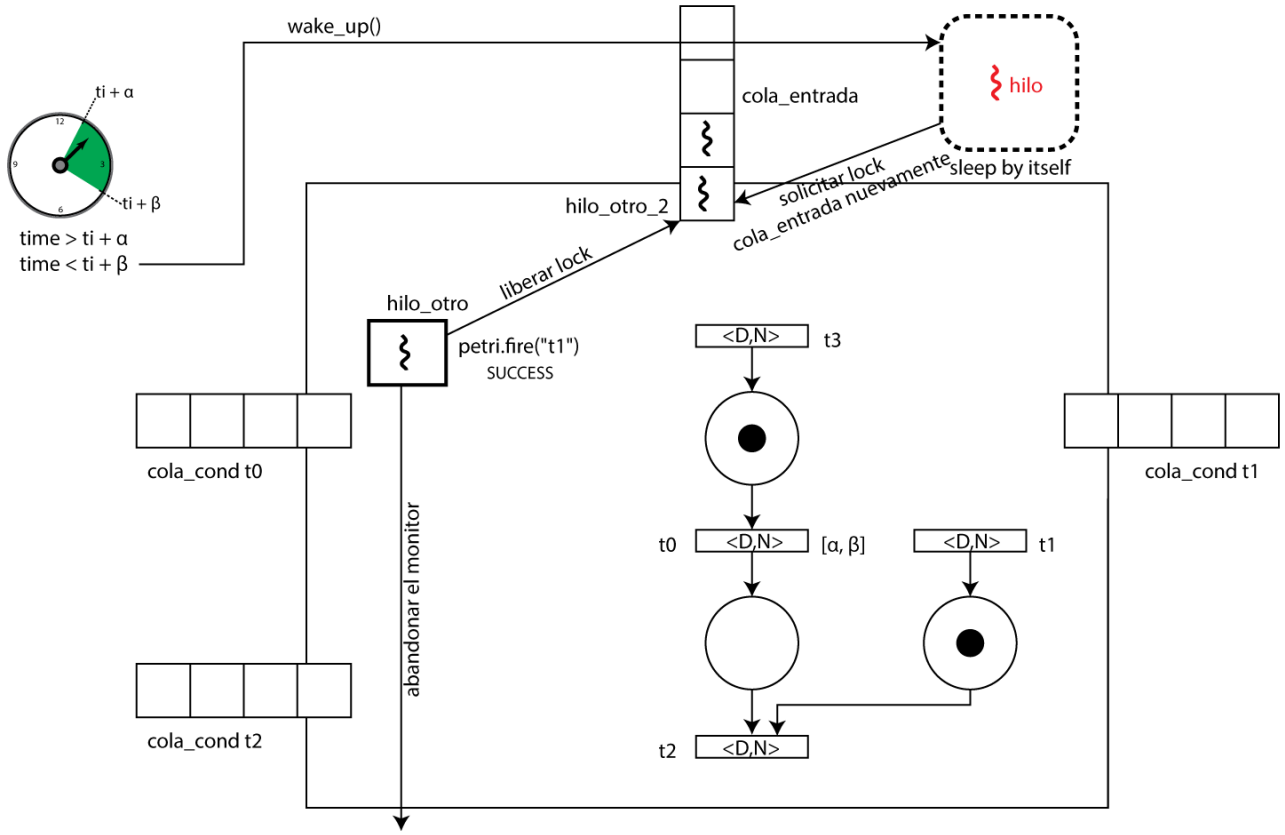
(b)


ti + α
tiempo_actual
ti= tiempo
habilitación t0
ti + β
time = tiempo_actual
time < ti + α
cola_entrada
sleep by itself
sleep(ti + α - time)
hilo
petri.fire("t0", time)
time < ti + α
TIMED_BEFORE_TIMESPAN
<D,N>
t3
cola_cond t0
cola_cond t1
t0
[α, β]
t1
cola_cond t2
t2

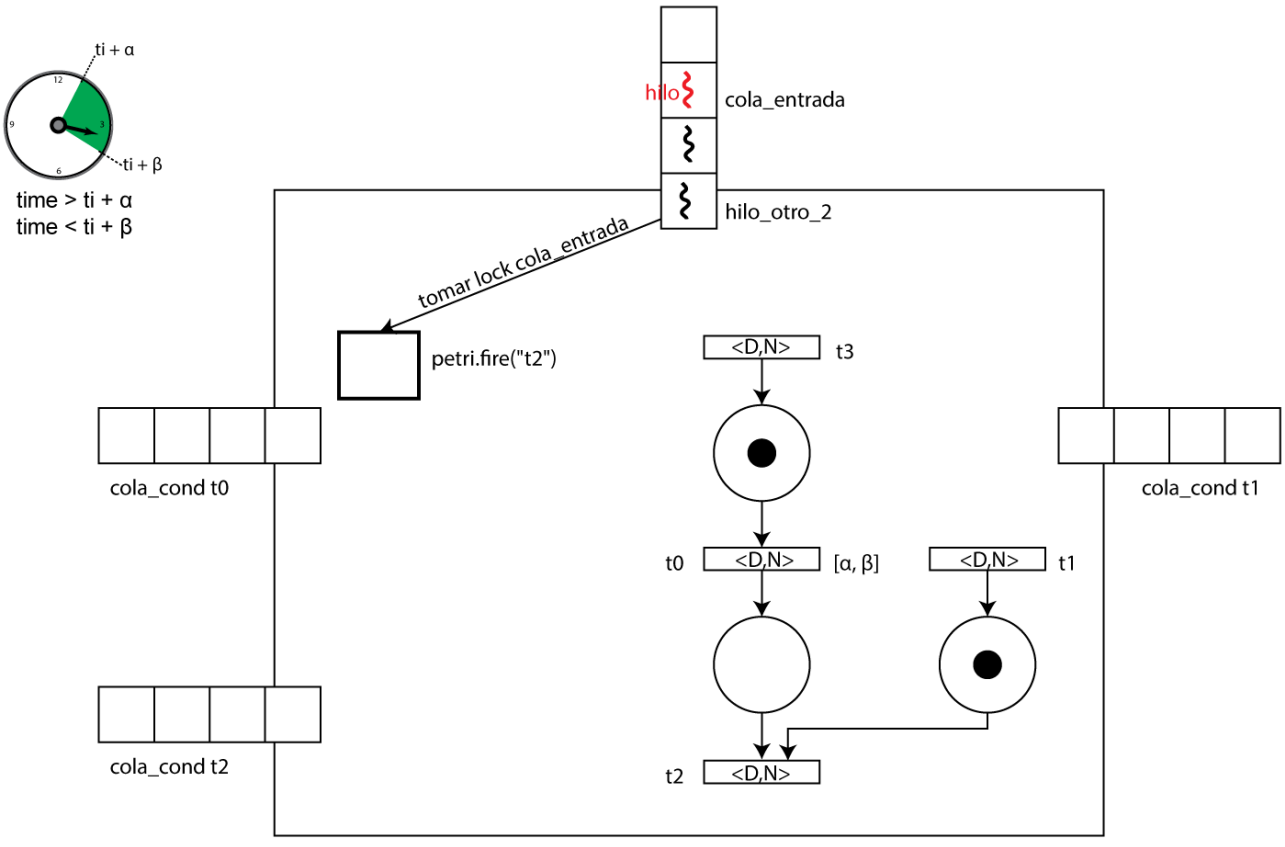
(c)


ti + α
ti + β
time < ti+α
cola_entrada
hilo
sleep by itself
devolver lock cola_entrada
hilo_otro
tomar lock de cola_entrada
petri.fire("t1")
<D,N>
t3
cola_cond t0
cola_cond t1
t0
[α, β]
t1
cola_cond t2
t2

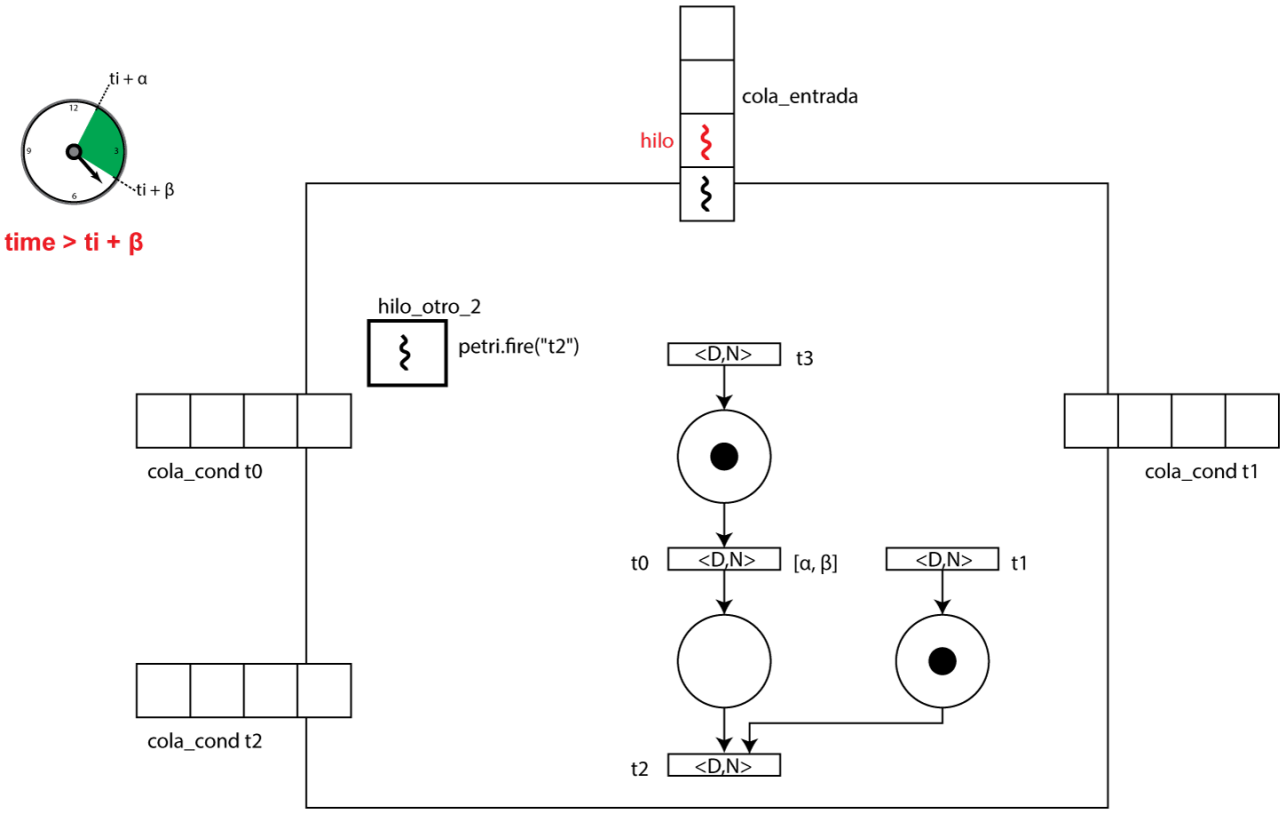
(d)


wake_up()
ti + α
ti + β
time > ti + α
time < ti + β
cola_entrada
hilo
hilo_otro_2
solicitar lock
cola_entrada nuevamente
sleep by itself
liberar lock
hilo_otro
petri.fire("t1")
SUCCESS
abandonar el monitor
cola_cond t0
cola_cond t1
cola_cond t2
<D,N>
t3
t0
[α, β]
t1
t2

(e)


ti + α
ti + β
time > ti + α
time < ti + β
hilo
cola_entrada
hilo_otro_2
tomar lock cola_entrada
petri.fire("t2")
cola_cond t0
cola_cond t1
cola_cond t2
<D,N>
t3
t0
[α, β]
t1
t2

(f)



(g)

Figura 9.7: Inversión de prioridades en la cola de entrada del monitor

9.4.7.2. Inversión de prioridades en una cola de condición

Existe inversión de prioridad en la cola de condición de una transición si ocurren los siguientes eventos en orden:

- Se sensibiliza la transición temporal t_0 con intervalo $[a, b]$ en el instante t_i
- El hilo th_0 intenta disparar t_0 en $td < (t_i + a)$
- El hilo th_0 libera la entrada y se bloquea temporalmente
- Llegan N hilos que intentan disparar a t_0 y se encolan en su cola de condición
- El hilo th_1 toma el lock de entrada y dispara la transición t_1, que deshabilita a t_0
- Cuando th_0 se desbloquea, toma el lock de entrada e intenta disparar t_0. Como la transición no está sensibilizada, th_0 se bloquea en la cola de condición de t_0

En este caso, th_0 se encola al final de la cola de condición, cediéndole incorrectamente su prioridad a los N hilos que llegaron después que él al monitor a disparar a t_0.

9.4.8. Solución a los problemas de inversión de prioridad

Del análisis de los casos de inversión de prioridades expuestos en la sección anterior, se llega a la conclusión de que ambos problemas se solucionan mediante una modificación de la política de prioridad de desbloqueo de hilos en las colas de entrada y de condición. De esta manera, un hilo que espera por la sensibilización temporal de una transición temporizada tendrá máxima prioridad al momento de reintentar un disparo fallido.

Para implementar colas de prioridad, `PetriMonitor` utiliza una instancia de la clase `PriorityBinaryLock` para la cola de entrada, y `FairQueue` utiliza una para las colas de condición. (ver figura 9.3). Esta clase define un lock que implementa dos métodos básicos para su uso:

- `lock(LockPriority priority)`: Su parámetro es opcional y por defecto es de baja prioridad. Permite a un hilo tomar el lock. En caso de ya estar tomado, el hilo se bloquea en la cola asociada.
- `unlock()`: Libera el lock. Si existe al menos un hilo en la cola despierta al primero.

Internamente, `PriorityBinaryLock` utiliza una cola de prioridades para gestionar los hilos bloqueados que almacena. De esta forma, siendo A y B dos hilos bloqueados en la cola de prioridades, A estará más próximo a ser desbloqueado si:

- A tiene mayor prioridad que B
- A y B tienen la misma prioridad pero el instante de bloqueo de A es menor al de B

De esta manera, dentro de un mismo rango de prioridades se respeta el esquema FIFO utilizado comúnmente en una cola.

El ordenamiento de la cola es equivalente a que existan dos colas FIFO, una con prioridad alta y otra baja. Así, cada vez que se libere el lock, se prefiere desbloquear a un hilo en la cola de alta prioridad.

En la figura 9.7 se observa la secuencia de pasos que lleva a la solución del caso de inversión de prioridades de la sección 9.4.7.1.

Las subfiguras grafican los siguientes eventos:

a) La transición t_0 se sensibiliza en el instante t_i. El hilo th_0 llama a `petriMonitor.fireTransition("t0")`

b) Como no hay hilo activo en el monitor y la cola de entrada está vacía, th_0 toma el lock de entrada e intenta disparar t_0. Mientras tanto se encolan algunos hilos en la cola de entrada

c) La llamada a `petri.fire("t0")` retornó un código `TIMED_BEFORE_TIMESPAN`, por lo que th_0 se bloquea temporalmente hasta el instante $t_i + a$

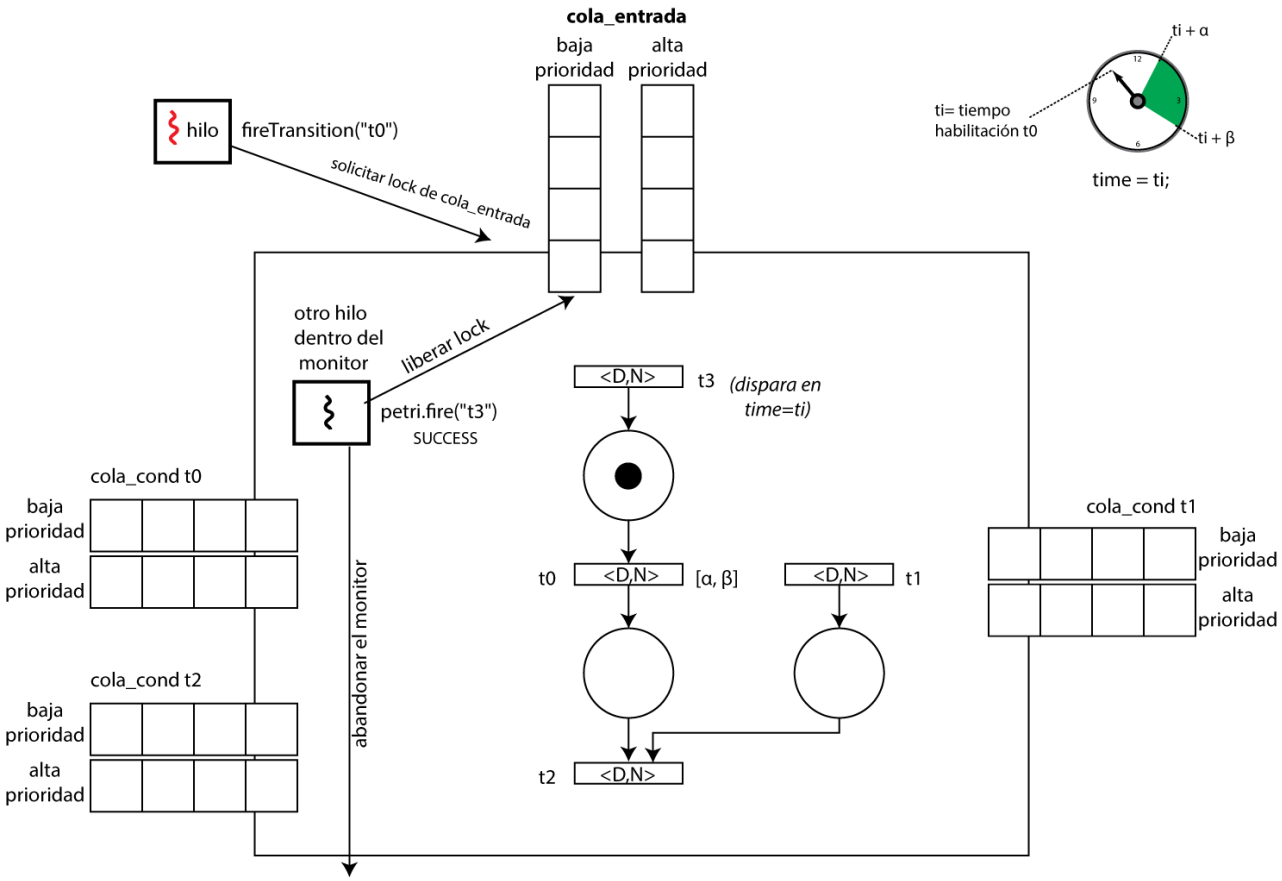
d) Antes de bloquearse, th_0 libera la entrada al monitor. Otro hilo ingresa al monitor a disparar a t_1

e) Se alcanza el instante $t_i + a$ por lo que th_0 se desbloquea y reintenta el disparo. Como existe un hilo activo se bloquea con alta prioridad.

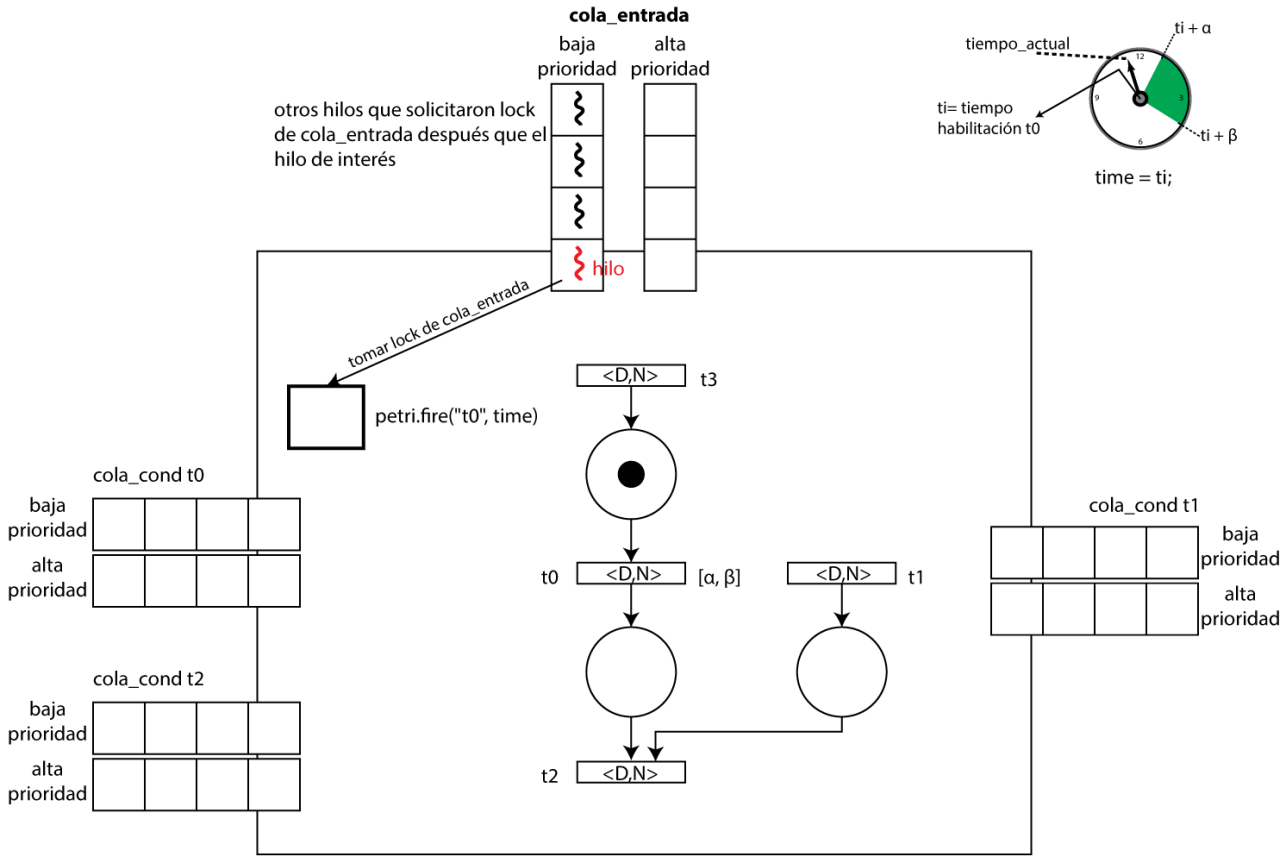
f) El hilo activo abandona el monitor y libera la entrada. Se habilita a th_0 por estar en la cola de alta prioridad.

g) th_0 ingresa al monitor dentro del intervalo de sensibilización de t_0 para reintentar el disparo

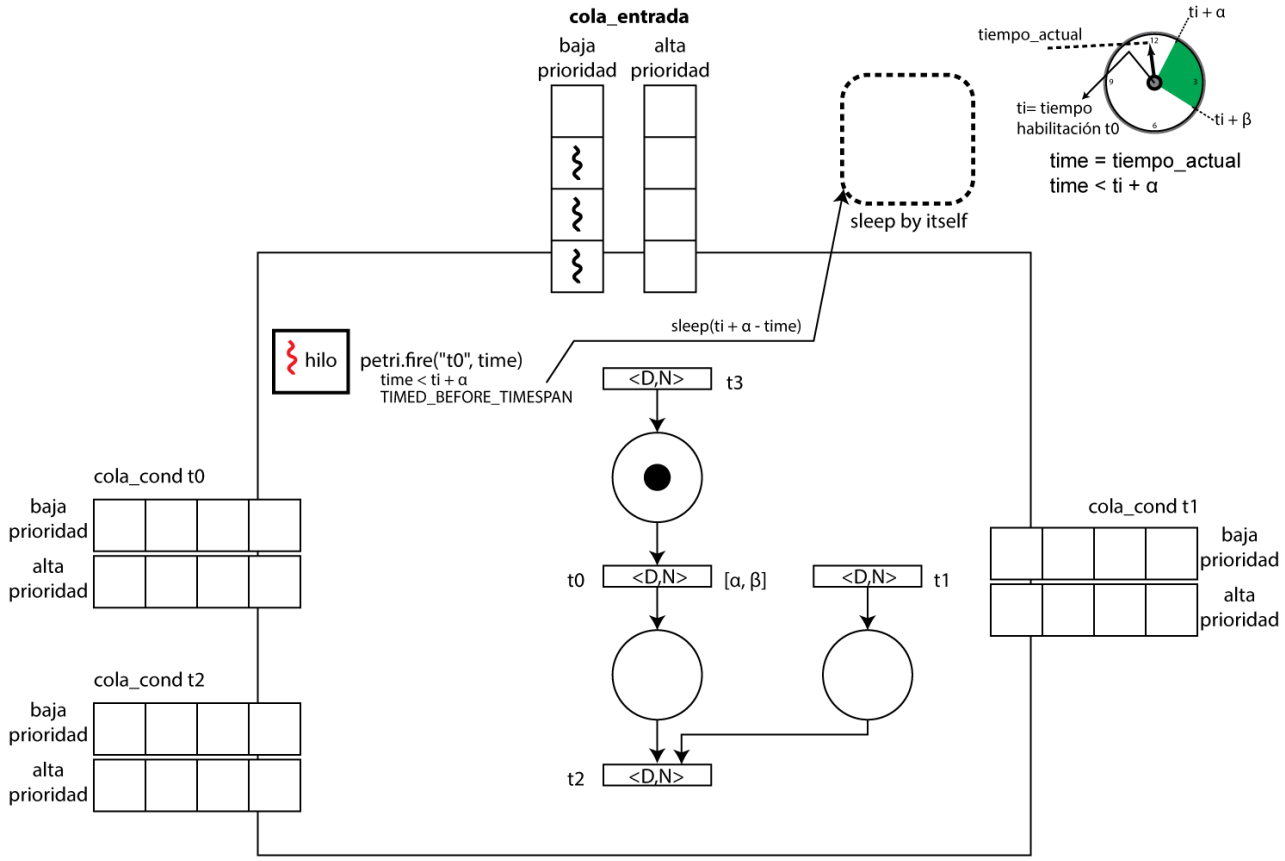
h) th_0 logró disparar a t_0 dentro del intervalo de sensibilización y abandona el monitor, liberando la entrada para un nuevo hilo



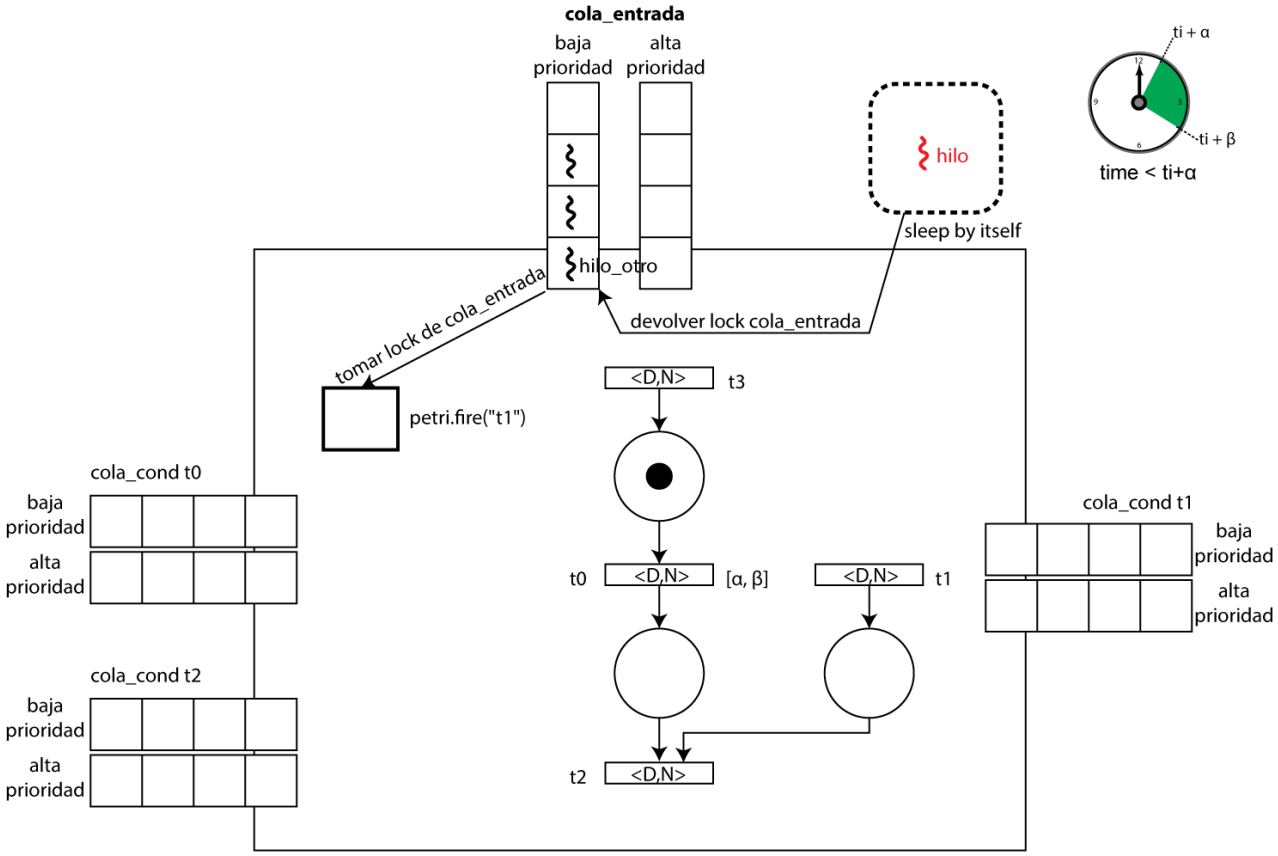
(a)


cola_entrada
baja prioridad
alta prioridad
otros hilos que solicitaron lock de cola_entrada después que el hilo de interés
hilo
tomar lock de cola_entrada
petri.fire("t0", time)
tiempo_actual
ti + α
ti= tiempo habilitación t0
ti + β
time = ti;
<D,N>
t3
t0
[α, β]
t1
t2
cola_cond t0
cola_cond t1
cola_cond t2

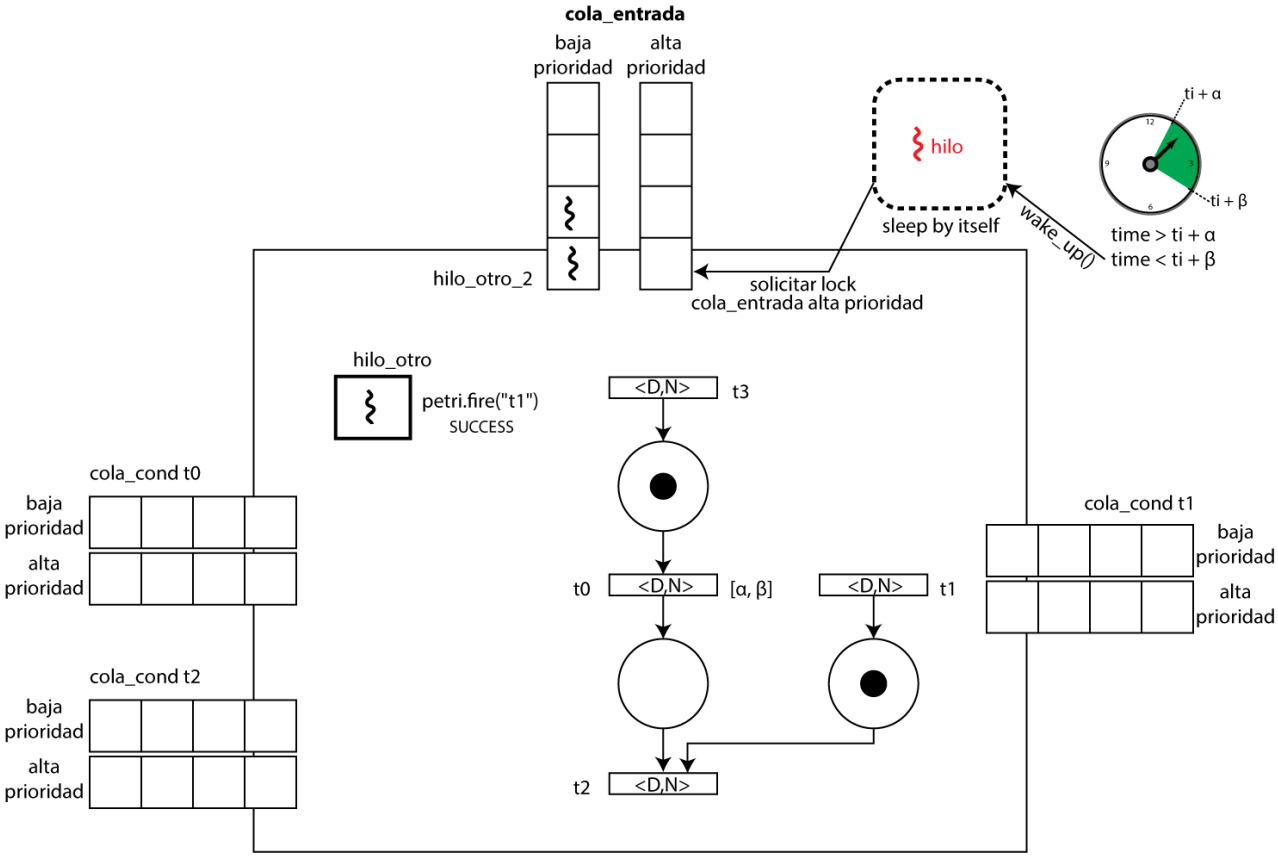
(b)


cola_entrada
baja prioridad
alta prioridad
sleep by itself
tiempo_actual
ti + α
ti= tiempo habilitación t0
ti + β
time = tiempo_actual
time < ti + α
sleep(ti + α - time)
hilo
petri.fire("t0", time)
time < ti + α
TIMED_BEFORE_TIMESPAN
<D,N>
t3
t0
[α, β]
t1
t2
cola_cond t0
cola_cond t1
cola_cond t2

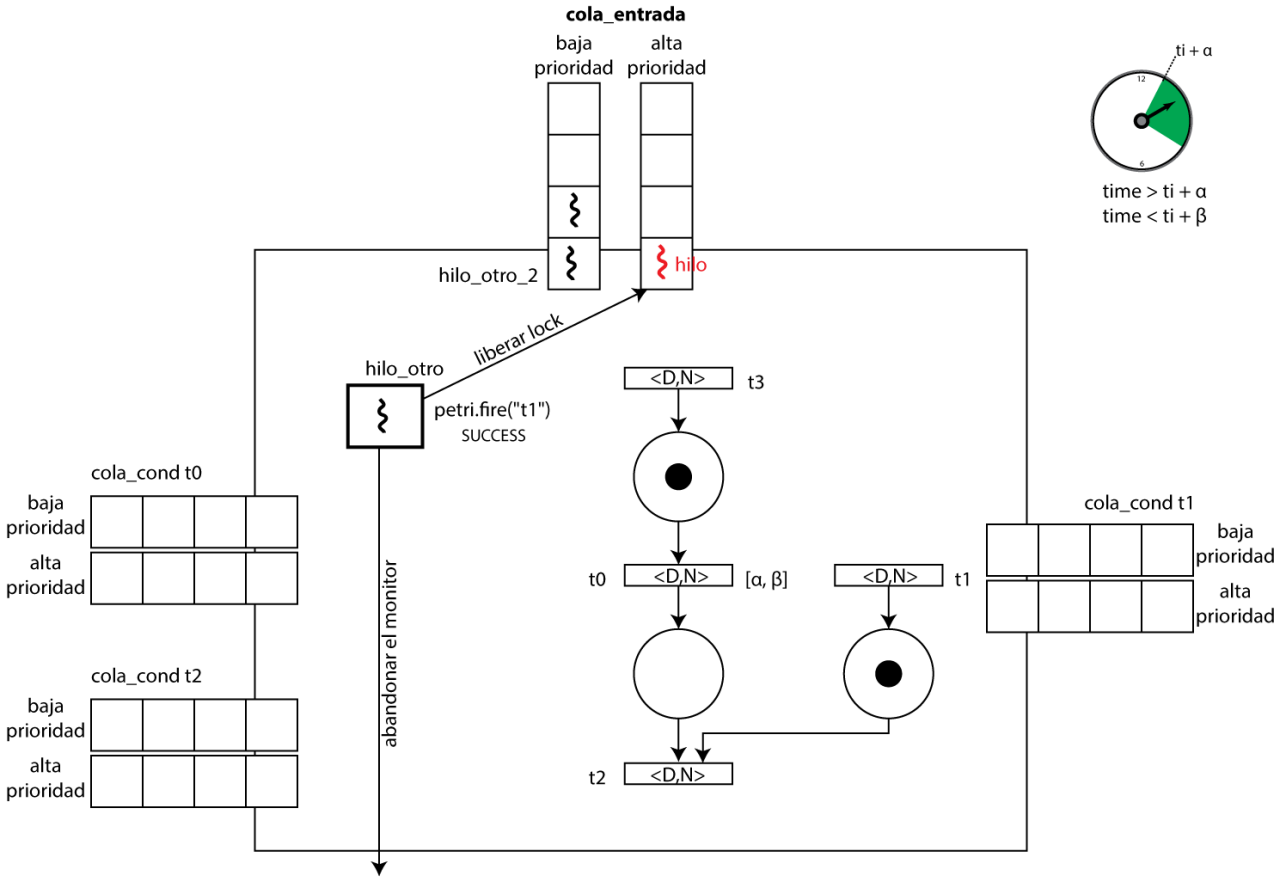
(c)


cola_entrada
baja prioridad
alta prioridad
hilo
ti + α
ti + β
time < ti+α
sleep by itself
hilo_otro
tomar lock de cola_entrada
devolver lock cola_entrada
petri.fire("t1")
<D,N>
t3
t0
[α, β]
t1
t2
cola_cond t0
cola_cond t1
cola_cond t2

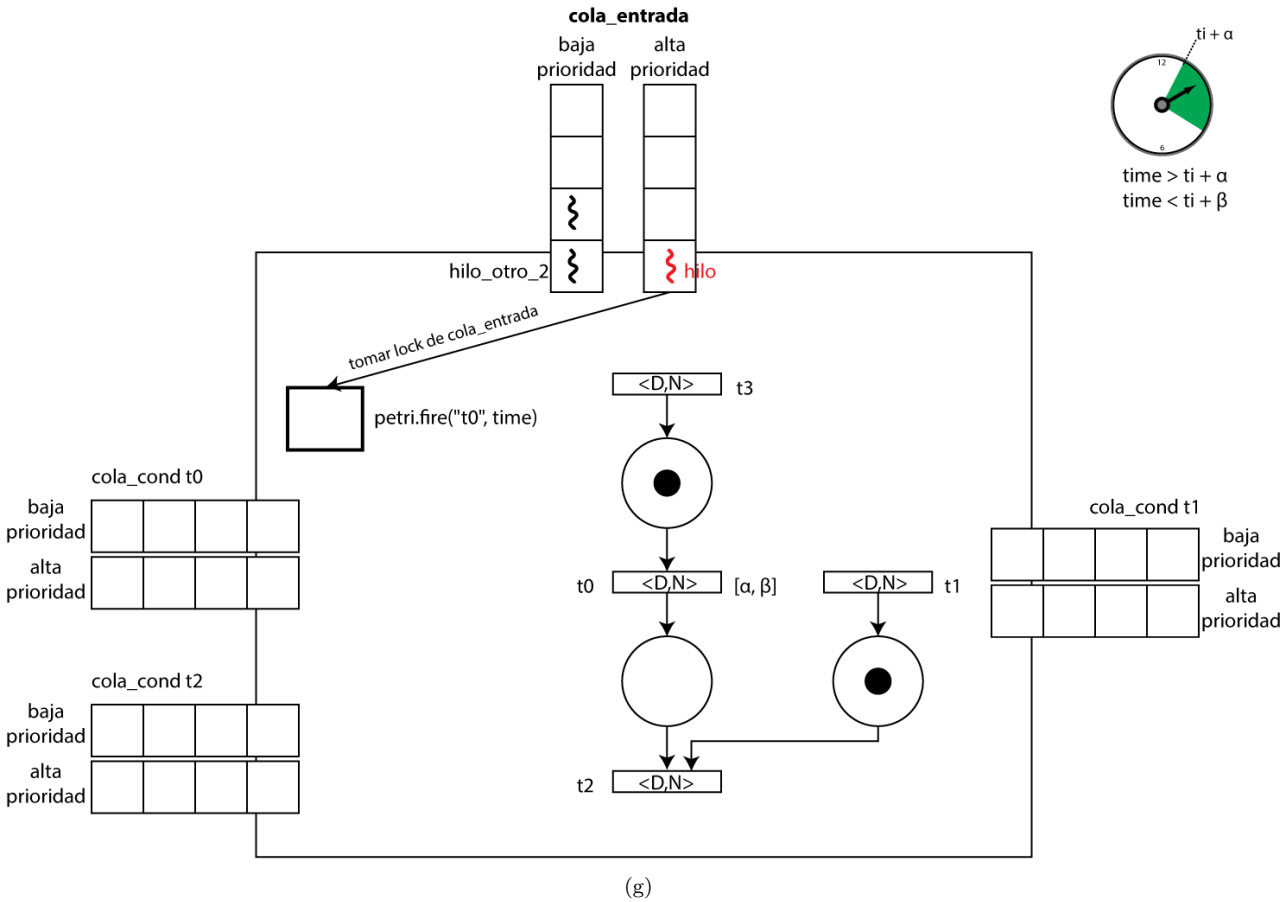
(d)


cola_entrada
baja prioridad
alta prioridad
hilo
ti + α
ti + β
sleep by itself
wake_up()
time > ti + α
time < ti + β
hilo_otro_2
solicitar lock
cola_entrada alta prioridad
hilo_otro
petri.fire("t1")
SUCCESS
<D,N>
t3
t0
[α, β]
t1
t2
cola_cond t0
cola_cond t1
cola_cond t2

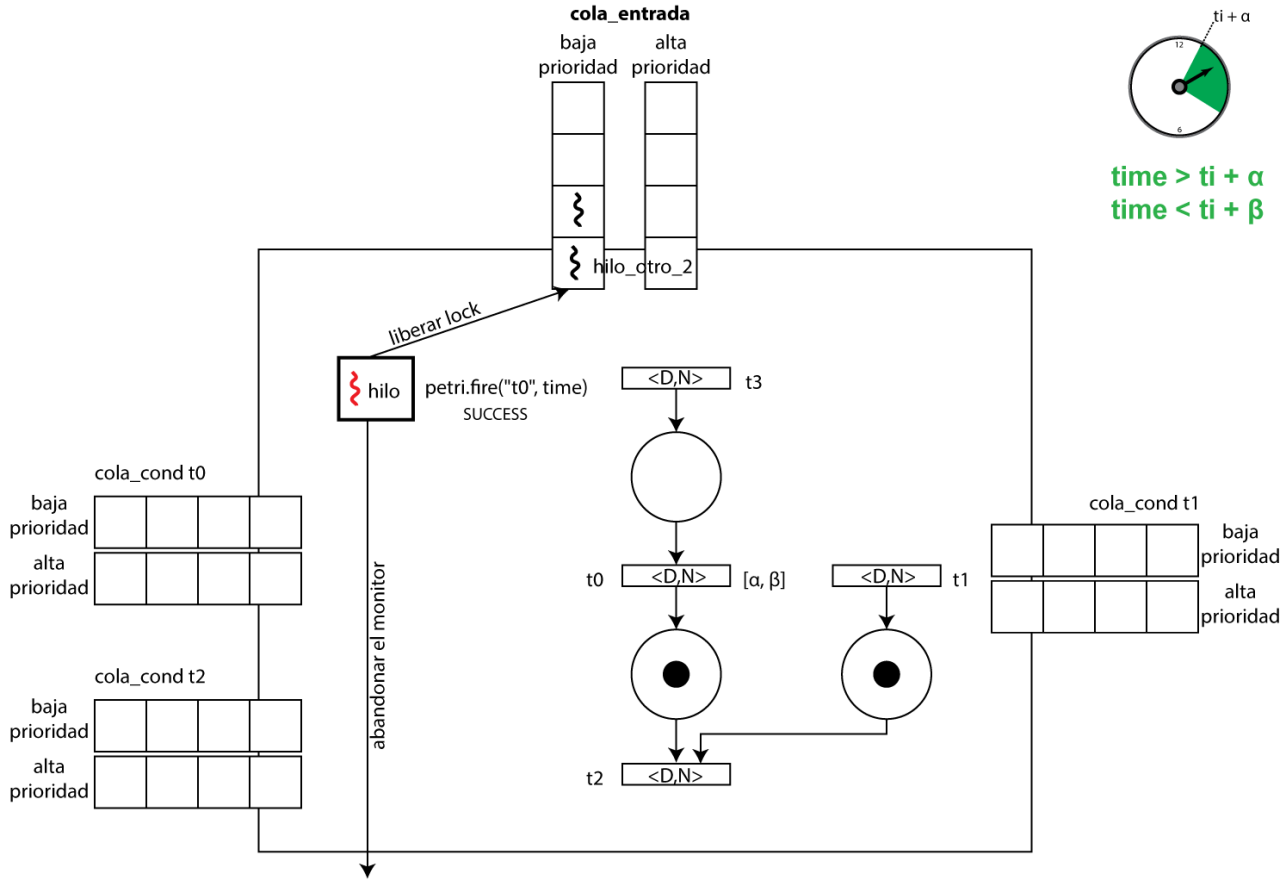
(e)


cola_entrada
baja prioridad
alta prioridad
hilo_otro_2
hilo
ti + α
time > ti + α
time < ti + β
liberar lock
hilo_otro
petri.fire("t1")
SUCCESS
<D,N>
t3
t0
[α, β]
t1
t2
cola_cond t0
cola_cond t1
cola_cond t2
abandonar el monitor

(f)


cola_entrada
baja prioridad
alta prioridad
hilo_otro_2
hilo
ti + α
time > ti + α
time < ti + β
tomar lock de cola_entrada
petri.fire("t0", time)
<D,N>
t3
t0
[α, β]
t1
t2
cola_cond t0
cola_cond t1
cola_cond t2

(g)



(h)

Figura 9.7: Inversión de prioridades en la cola de entrada del monitor

9.4.9. Informes de Disparo de una Transición

Existen casos en los que es de interés del programador de un sistema que utilice a JPCM recibir una notificación cuando se produce el disparo de una transición. Para esto, JPCM ofrece una interfaz de suscripción a eventos de disparo de transición.

Como se puede observar en el diagrama de la figura 9.5, luego de un disparo exitoso se envía un evento de disparo. Para que un observador reciba este evento debe haberse suscrito previamente a los eventos de la transición en cuestión. La suscripción se realiza con una llamada al método `PetriMonitor.subscribeToTransition()`.

Un intento de suscripción a una transición que no sea de tipo *informada* resulta en el lanzamiento de una excepción `IllegalArgumentException` con un mensaje que explica la situación. De la misma manera, no se realizan envíos de eventos para transiciones que no sean de tipo *informada*.

9.4.10. Guardas

Como se explicó en la sección 3.3.2, a una transición se pueden asociar valores booleanos que modifican su semántica de sensibilización. JPCM provee soporte para guardas con habilitación por valor `true` o por valor `false`.

Las guardas son cargadas en tiempo de inicialización durante la construcción del objeto `PetriNet` y son inicializadas con valor `false`.

En cualquier momento un hilo puede modificar el valor de una guarda accediendo al método `PetriMonitor.setGuard()` con el nombre de la guarda a modificar y su nuevo valor. Esto es coordinado por el lock de entrada de forma conjunta con el disparo de una transición para evitar problemas de acceso concurrente sobre la RdP. Se contempla la posible sensibilización de transiciones, ya sean de tipo *automática* (donde el hilo que hizo el cambio de valor de la guarda realiza el disparo) o de tipo *disparada* (en cuyo caso desbloquea a un hilo que estuviera esperando en su cola de condición si lo hubiera) de la misma forma que en el disparo de una transición.

9.5. Manual de Uso

9.5.1. Formato del Archivo

Java Petri Concurrency Monitor utiliza el formato estándar PNML para descripción de redes de Petri, específicamente el dialecto del editor TINA [Tin].

Para más información sobre el estándar PNML, el sitio online de referencia del lenguaje es [Pnm].

Nota: Como TINA no tiene soporte para arcos de reset, el usuario debe especificar este tipo de conexión modificando el archivo PNML con un editor de texto. Se recomienda utilizar otro tipo de arco en TINA y luego cambiar el valor del campo *"type"* del arco correspondiente en el archivo PNML.

Ej:

```
<arc id="e-1" source="p-1" target="t-1">
  <type value="reset"/>
</arc>
```

9.5.2. Etiquetas

Las etiquetas especifican atributos para una transición. En TINA, una etiqueta se agrega a una transición como un atributo *label*. Una etiqueta especifica tres propiedades de una transición:

- Automática: Una transición automática no requiere de un evento para dispararse. En su lugar, se dispara automáticamente cuando la lógica de sensibilizado y la política lo indican.
- Informada: Una transición informada acepta peticiones de suscripción a informes, y envía informes a sus suscriptores ante un disparo (ver sección 9.5.3).
- Guarda: Es el nombre de la guarda asociada a la transición. Se especifica también el estado de habilitación (ver sección 3.3.2).

La sintaxis a utilizar es la siguiente:

```
<valor_automatica,valor_informada,(nombre_guarda)>
```

Donde:

- **valor_automatica** es un string no sensible a mayúsculas que puede adoptar valor A para una transición automática, o F ó D para una disparada.
- **valor_informada** es un string no sensible a mayúsculas que adopta el valor I para una transición informada, y N para una no informada.
- **nombre_guarda** es el nombre de la guarda asociada a la transición etiquetada. Se asocia por habilitación por *false* si antes del nombre se agrega el símbolo ! ó $\sim$.
  Es posible asociar la misma guarda a múltiples transiciones, cada una con su estado de sensibilización.
  El nombre de una guarda debe respetar las limitaciones del nombramiento de variables de Java.

Un ejemplo de etiqueta es:

```
<D,I,(!foobar)>
```

La transición etiquetada es *disparada*, *informada*, y tiene una guarda asociada llamada *foobar*, que la habilita por valor **false**.

Todos los valores de etiqueta son opcionales. Para especificar uno de ellos, todos los valores a la izquierda deben estar explícitos. Los valores por defecto son:

- valor_automatica: D
- valor_informada: N
- nombre_guarda: Ninguno

9.5.3. Mensajes de Eventos

Cuando se dispara una transición informada, si existe al menos un observador suscripto a sus eventos, se envía un mensaje. Este mensaje respeta el formato JSON y contiene la siguiente información sobre la transición disparada:

- Nombre de la transición: Provisto por el usuario o asignado automáticamente por el editor TINA.
- Id de la transición: Asignado automáticamente por TINA.
- Índice de la transición: Índice correspondiente a la columna de la matriz de incidencia de la RdP para la transición disparada. Este índice se computa internamente y es útil para el depurado de la red.

El formato del mensaje es el siguiente:

```
{
  "name": "nombre_de_la_transición",
  "id": "id_de_la_transición",
  "index": "índice_de_la_transición"
}
```

Generar y Anular una Suscripción a Eventos: Para generar una suscripción a los eventos de una transición debe haber una implementación concreta de `rx.Observer<String>` (http://reactivex.io/RxJava/javadoc/rx/Observer.html), que en su implementación de `onNext()` (http://reactivex.io/RxJava/javadoc/rx/Observer.html#onNext(T)) maneje el mensaje de evento descripto en esta sección.

A su vez, la suscripción se realiza con el objeto `Transition` o alternativamente con el nombre de la transición.

A modo de ejemplo se asume que existe una clase `ConcreteObserver` que se ajusta a las restricciones antes mencionadas. Se generan dos suscripciones del mismo objeto observer a dos transiciones distintas.

```
// código

// primera transición a la que se quiere suscribir
Transition t0 = petri.getTransitions()[0];

// segunda transición a la que se quiere suscribir
String t1Name = "start_process_01";

Observer<String> observer = new ConcreteObserver();

// suscripción mediante objeto Transition
Subscription subscription0 = monitor.subscribeToTransition(t0, observer);

//suscripción mediante nombre de la transición
Subscription subscription1 = monitor.subscribeToTransition(t1Name, observer);

// más código
```

La suscripción a un evento devuelve un objeto `Subscription` (http://reactivex.io/RxJava/javadoc/rx/Subscription.html), que es usado para cancelar la suscripción. Una llamada a `subscription.unsubscribe()` cancela la suscripción.

> **Nota:** La generación y cancelación de suscripciones utiliza la biblioteca RxJava. Más información en [RxJb] y [RxJa]

9.5.4. Guardas

Las guardas son variables booleanas asociadas a una o más transiciones (ver sección 3.3.2).

En la figura 9.8 se muestra cómo la guarda *"fooGuard"* determina si se efectúa el disparo de t_0 ó t_1.

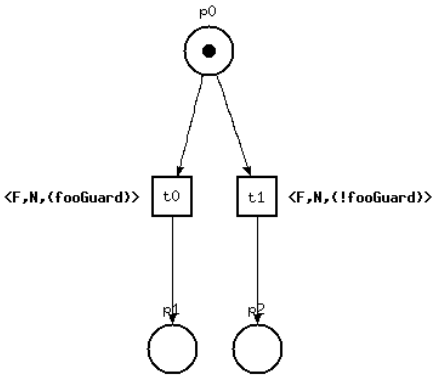


Figura 9.8: Guarda como toma de una decisión.

> **Nota:** Todas las guardas se inicializan con valor `false` al principio de la ejecución del programa.

Modificación del valor de una Guarda: En el siguiente fragmento de código se muestra cómo cambiar el valor de una guarda *fooGuard*:

```
// código

// ya fue declarado un PetriMonitor llamado "monitor"

monitor.setGuard("fooGuard", true);

// más código

monitor.setGuard("fooGuard", false);

// más código
```

9.5.5. Inicialización del Monitor de Redes de Petri

Antes de operar con el monitor, el usuario debe inicializar algunos objetos con la información del modelo.

A modo de ejemplo, se asume que existe un archivo PNML que describe la RdP a utilizar en la dirección `"/path/a/mi/red/de/petri.pnml"`. El siguiente fragmento de código inicializa el entorno:

```
public void setUp() {
  PetriNetFactory factory = new PetriNetFactory("/path/a/mi/red/de/petri.pnml");
  PetriNet petri = factory.makePetriNet(petriNetType.PLACE_TRANSITION);
  TransitionsPolicy policy = new FirstInLinePolicy();
  PetriMonitor monitor = new PetriMonitor(petri, policy);

  // declarar los hilos de trabajo aquí

  // de querer agregar un observer, se lo debe instanciar
  // y suscribir en este punto

  // se debe inicializar la RdP antes de poder utilizarla
  petri.initializePetriNet();

  // iniciar la ejecución de los hilos de trabajo

  // el hilo principal puede realizar otra tarea
  // mientras los hilos de trabajo se ejecutan
  // siempre que no finalice antes que ellos

  // en este ejemplo, se imprime el marcado de la red cada 5 segundos

  while(true){
    try{
      Thread.sleep(5000);
      System.out.println(petri.getCurrentMarking());
    } catch (InterruptedException e){
      // código de manejo de la interrupción
    }
  }
}
```

9.5.6. Disparo de una Transición

El disparo de una transición se realiza en exclusión mutua dentro del monitor. Se ejecuta mediante el método `fireTransition()` sobre una instancia de la clase `PetriMonitor`.

Existen dos formas de especificar la transición a disparar, ya sea utilizando el objeto `Transition` o el nombre de la transición. En el siguiente ejemplo se muestra la declaración de un hilo que dispara una transición por cada una de las variantes.

```
Thread worker = new Thread( new Runnable() {

  @Override
  public void run() {
    try {

      // sentencias a ejecutar fuera de la exclusión mutua

      // disparo por nombre de transición
      monitor.fireTransition("NombreDeUnaTransicion");

      // otras tareas

      // disparo por objeto Transition
      Transition t0 = petri.getTransition()[0];
      monitor.fireTransition(t0);

      // otras tareas

      // quizá disparar otra transición de ser necesario

    } catch (IllegalArgumentException | NotInitializedPetriNetException e) {
      // manejar las excepciones
    }
  }
});
```

Disparos Perennes

Existe un segundo parámetro para el disparo de una transición: `boolean notPerennialFire`. Tiene valor por defecto `false` e indica si el disparo a realizar es no-perenne.

Si un hilo intenta disparar una transición no sensibilizada de forma perenne, se bloquea en la cola de condición asociada (comportamiento por defecto). Por otro lado, si el disparo es no-perenne, el hilo que intenta hacer el disparo no se bloquea y abandona el monitor.

El concepto de disparo no-perenne es útil para modelar acciones no bloqueantes. Un ejemplo de acción no bloqueante es el encendido de una luz:

- Si la luz está apagada, se acciona el interruptor y la luz se enciende
- Si la luz está encendida no se toma ninguna acción

En ambos casos el resultado final es el esperado aunque en el segundo no se haya efectuado ninguna acción. De la misma forma, un hilo que realiza un disparo no-perenne sobre una transición no sensibilizada no se bloquea esperando a que ésta se sensibilice.

En el caso del disparo no-perenne de una transición temporal, el hilo que realiza el disparo puede bloquearse únicamente si el intento de disparo ocurre antes del principio del intervalo dinámico de la transición (ver sección 3.3.3.2). Este tipo de bloqueo es temporizado y no requiere de la activación por medio de otro hilo.

> **Nota:** Dado que el disparo perenne de una transición no sensibilizada bloquea al hilo que lo ejecuta, no debe usarse el hilo principal para realizar disparos.

> **Nota:** El disparo explícito de una transición automática provoca un error *IllegalTransitionFiringError*.

9.5.7. Política de Transiciones

Ante la modificación de una condición de sensibilización (marcado de la red o valor de una guarda), ocurre un cambio potencial en el conjunto de transiciones sensibilizadas. En este caso, se deben disparar las transiciones automáticas sensibilizadas y se debe señalizar a los hilos que estaban esperando por una transición recientemente sensibilizada.

Si un cambio en las condiciones de sensibilización habilita una sola transición es trivial cuál transición disparar o cuál hilo señalizar. En caso contrario, la *política de transiciones* es la que debe decidir qué transición tiene la mayor prioridad entre el conjunto de todas las transiciones sensibilizadas.

Java Petri Concurrency Monitor incluye dos políticas por defecto. Estas son:

- `FirstInLinePolicy`: Elige la primer transición sensibilizada del conjunto dado.
- `RandomPolicy`: Elige una transición de forma aleatoria de entre todas las sensibilizadas.

Existen dos formas de asignar una política de transiciones al monitor. Estas son:

- Durante la construcción del objeto `PetriMonitor`, utilizando una instancia de `TransitionsPolicy` en el constructor.
- En tiempo de ejecución, mediante la llamada al método
  `PetriMonitor.setTransitionsPolicy(TransitionsPolicy _transitionsPolicy)`

Creación de una Política de Transiciones: Cualquier instancia de una clase que extienda la clase abstracta `TransitionsPolicy` es una política de transiciones aceptada por el monitor. La forma de especificar la próxima transición a elegir sobre un conjunto es implementando el método `public int which(boolean[] enabled)` donde:

- El array `enabled` está ordenado por índice de las transiciones, que se corresponde con el orden de `PetriNet.getTransitions()`. Si la posición i del array tiene valor `true`, la *i-ésima* transición está sensibilizada.
- El valor de retorno es el índice de la próxima transición a disparar o un valor negativo si no hay transiciones sensibilizadas.

En el siguiente ejemplo se muestra cómo definir una política de transiciones para utilizar con el monitor de RdP. En este caso, se utiliza una clase anónima de Java y se realiza en el momento de inicialización del monitor:

```
// código

// se asume que existe una instancia de PetriNet llamada "petri"

PetriMonitor monitor = new PetriMonitor(petri,
  new TransitionsPolicy(petri) {

    @Override
    public int which(boolean[] enabled) {
      int ret = 0;
      // se debe dar algún valor a "ret"
      // siguiendo el criterio de la política a aplicar
      return ret;
    }

});

// más código
```

En el próximo ejemplo, se muestra el cuerpo de una clase que implementa una política estática, definiendo un órden de prioridad de las transiciones mediante sus nombres:

```
public class OrderedPrioritiesPolicy extends TransitionsPolicy {

    private int[] priorityArray = {
        petri.getTransition("fin_proceso_01").getIndex(),
        petri.getTransition("fin_proceso_02").getIndex(),
        petri.getTransition("comienzo_proceso_01").getIndex(),
        petri.getTransition("comienzo_proceso_02").getIndex()
    };

    public OrderedPrioritiesPolicy(PetriNet _petri){
        super(_petri)
    }

    @Override
    public int which(boolean[] enabled) {
        for(int index : priorityArray) {
            if(enabled[index]) {
                return index;
            }
        }

        return -1;
    }
}
```

Nota: La implementación de una política de transiciones de forma incorrecta lleva a situaciones de conflicto donde se produce la inanición de uno o más hilos de ejecución.

Nota: Si existe al menos un valor `true` en el vector `enabled`, el método `which` no debe retornar un valor negativo porque genera inanición sobre uno o más hilos de ejecución.

Capítulo 10

Diseño de Baboon Framework

10.1. Introducción

En este capítulo se detalla el diseño de Baboon Framework. En primer lugar se fundamenta la decisión de elaborar un Framework teniendo en cuenta el análisis de experiencias previas. Se realiza una clasificación de los eventos que se intercambian en sistemas reactivos desarrollados utilizando el monitor de RdP. Se detalla el diseño de la arquitectura del framework en base a dicho intercambio de eventos.

Se realiza un análisis de las formas en que se pueden sincronizar las acciones de un sistema utilizando un monitor de RdP. Este análisis tiene el objetivo de definir el modo de sincronización más adecuado para la arquitectura del framework.

Se define el concepto de controlador de acción y sus clasificaciones. A su vez, se define el concepto de Guard Provider. Finalmente se define la relación entre los eventos y los controladores de acción, formalizando el intercambio de eventos entre el software de usuario y el framework.

10.2. Fundamentos del Framework

En la sección 9.1 se propone utilizar RdP como la lógica secuencial de un sistema concurrente. Para logralo, se implementó el monitor de petri por software descripto en la sección 9.3. Este monitor permite delegar la concurrencia y asincronismo del sistema a una red de Petri [Mic15]. Un ejemplo de uso exitoso se describe en [BL17].

La utilización directa del monitor es engorrosa y genera un alto grado de acoplamiento entre el software de usuario y la red de Petri puesto que los eventos de la red quedan asociados directamente a los eventos del mundo real que modela. La principal desventaja de un sistema acoplado a la red de Petri está dada por la reducción de la escalabilidad del sistema. Esto se debe a que una modificación de la lógica, que conlleva una sustitución de la red, implica también un cambio en el código del software. En consecuencia, dificulta el proceso de desarrollo y su mantenibilidad. Otra desventaja es que impide la reutilización de redes de Petri genéricas, útiles para resolver diferentes problemas de características similares.

Un requerimiento importante de este proyecto consiste en la facilidad de uso. Como se explicó en el párrafo anterior, la utilización del monitor en forma directa presenta una complejidad elevada y favorece a la generación de errores, ya que deben crearse todos los hilos de ejecución y deben programarse los disparos de transición de forma manual en el código. Este problema se manifiesta, por ejemplo, en soluciones como CodeGen [CF14]. Ante un cambio en la Red de Petri deben modificarse algunos o todos los disparos de transición distribuidos a lo largo del código. En caso de un error en esta tarea, se genera una sincronización incorrecta de los hilos.

A su vez, para desacoplar las acciones que debe realizar el sistema de los eventos, es necesario incorporar una entidad encargada de manejar y ejecutar dichas acciones.

Como resultado de este análisis, se llegó a la conclusión de que es necesario embeber el monitor de Petri en un framework que se encargue de desacoplar el código de usuario de la lógica de disparos. Una conclusión de similares características se desprende de [BL17], donde los autores expresan: “La debilidad encontrada en el proceso de elaboración del software, es que resultó problemático vincular los hilos con las transiciones de la RdP. Esto se debe a que entre las acciones y las transiciones no existe una capa de abstracción para mapear las mismas. Por lo cual, queda en evidencia que es necesaria la existencia de un framework para automatizar y facilitar la vinculación entre eventos, acciones y transiciones.”

10.3. Sincronización por Red de Petri a través de Eventos

Los sistemas a desarrollar utilizando el monitor descripto en la sección 9.3, son programas de software que intercambian eventos con la red de Petri y con su entorno físico.

Figura 10.1: Intercambio de eventos en un programa sincronizado por Red de Petri

El programa tiene la posibilidad de acceder a hardware del mundo físico, ya sea para realizar una acción (por ejemplo utilizando actuadores) o para obtener eventos del mundo exterior y comunicarlos a la red de Petri (por ejemplo con sensores).

La red toma los eventos del mundo exterior y, dependiendo de las condiciones del problema y del estado global, calcula los eventos hacia el programa. La red es un procesador de eventos. [Mic15] [AF15]

Este concepto se amplía en la sección Eventos Físicos y Eventos Lógicos de [AF15]. En esta sección se distingue la existencia de los dos tipos de eventos mencionados, y se los define como:

- Eventos Lógicos: eventos que son comprensibles por el monitor de redes de Petri, y están inherentemente asociados a transiciones de la red misma y sus colas.
- Eventos Físicos: suceden en el mundo físico y representan sucesos del dominio del problema. Están conectados con el software.

Tras la incorporación del concepto de eventos lógicos y físicos, en [AF15] se propone un diagrama de arquitectura de alto nivel como el de la Figura 10.2.

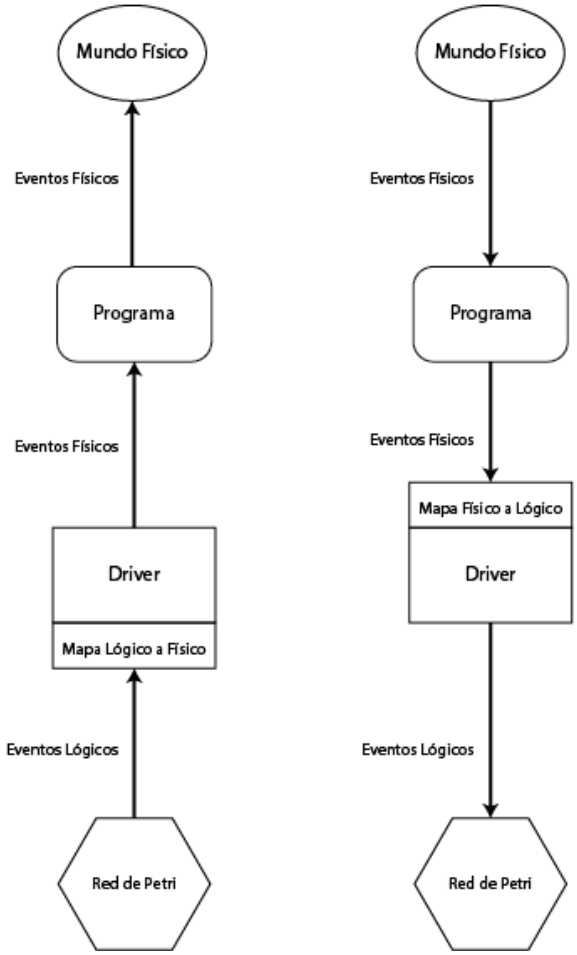


Figura 10.2: Arquitectura con Eventos Físicos y Lógicos

Los eventos físicos representados en la Figura 10.2 abarcan dos tipos de eventos. Uno de ellos está efectivamente relacionado con el hardware o software externos al sistema (mundo físico). El otro tipo de eventos está relacionado con el manejo de las acciones que van a realizar dichos elementos del mundo exterior, el cual se ejecuta a través del software del sistema. Por este motivo se determinó que la clasificación de los eventos en dos tipos no es suficiente para explicar la comunicación en un sistema de estas características.

A partir de lo expuesto en el párrafo previo, se modifica la clasificación de eventos existente:

- Eventos Lógicos: Conserva la definición descripta previamente. Este tipo de eventos se comunica utilizando las interfaces proporcionadas por el monitor de redes de Petri.
- Eventos Físicos: suceden en el mundo físico y representan sucesos del dominio del problema. Este tipo de eventos se comunican a través de las interfaces que expone el elemento del mundo exterior y las interfaces que ofrece el lenguaje de programación utilizado para desarrollar el software. Por ejemplo pueden comunicarse a través de Event Listeners, mecanismos de IPC, Sockets, Serial, Bluetooth, etc.
- Eventos de Acción: Evento intermedio entre los eventos físicos y lógicos. Este tipo de eventos es manejado por el framework durante la inicialización del sistema. Cumplen una función de abstracción entre las

acciones de software y los eventos lógicos, necesaria para desacoplar la red de Petri del software y permitir la inversión de control descripta en la sección 5.2.2.

A partir de la nueva clasificación de los eventos del sistema emerge una nueva arquitectura de alto nivel. A su vez emergen nuevos requerimientos, relacionados con el requerimiento número 2 de la sección 8.3. A continuación se listan los nuevos requerimientos:

- El monitor de RdP debe manejar los eventos lógicos del sistema, mediante las interfaces de disparo y evaluación de guardas.
- El framework debe ofrecer interfaces para mapear eventos lógicos de una RdP a eventos de acción especificados por los usuarios.

10.3.1. Arquitectura de alto nivel de Baboon

Un sistema informático consiste en una secuencia de acciones que se ejecutan ante el cumplimiento de determinadas condiciones. Desde el punto de vista del programa que analiza dichas condiciones, se las clasifica en síncronas y asíncronas.

- Síncronas: Están sicronizadas con la ejecución del programa. Se desencadenan en un momento específico, conocido de antemano. Por ejemplo, condiciones booleanas derivadas del estado del sistema que realizan cambios en el flujo de instrucciones del mismo (saltos condicionales).
- Asíncronas: Se desencadenan en cualquier momento, de forma independiente a la ejecución del programa. Por ejemplo, eventos provenientes del mundo exterior o mensajes entre procesos.

El objetivo de este trabajo es implementar un framework dirigido por redes de Petri para controlar la ejecución de todas aquellas acciones que respondan a estos tipos de condiciones. De esta forma, será el monitor de redes de Petri quien analice las condiciones y explicite el estado del sistema. Así, es responsabilidad de la red:

- Disparo de eventos provenientes de sistemas externos, que pueden llegar en cualquier instante de tiempo y sin un orden preestablecido. Los estados locales de la red se mantienen en causalidad de las acciones ejecutadas con anterioridad. El monitor es el encargado de mantener el estado lógico del sistema.
- Condiciones de sincronización para el ordenamiento de la ejecución de acciones en el tiempo.
- Condiciones impuestas por el dominio del problema. El monitor es el encargado de impedir la ejecución de una acción hasta que la misma pueda ser realizada sin riesgos.

De acuerdo a lo estudiado en la sección 5.2.2, una característica principal de un framework es la inversión de control. Por este motivo, el diseño de la arquitectura del framework contempla el control del flujo de ejecución del código de usuario.

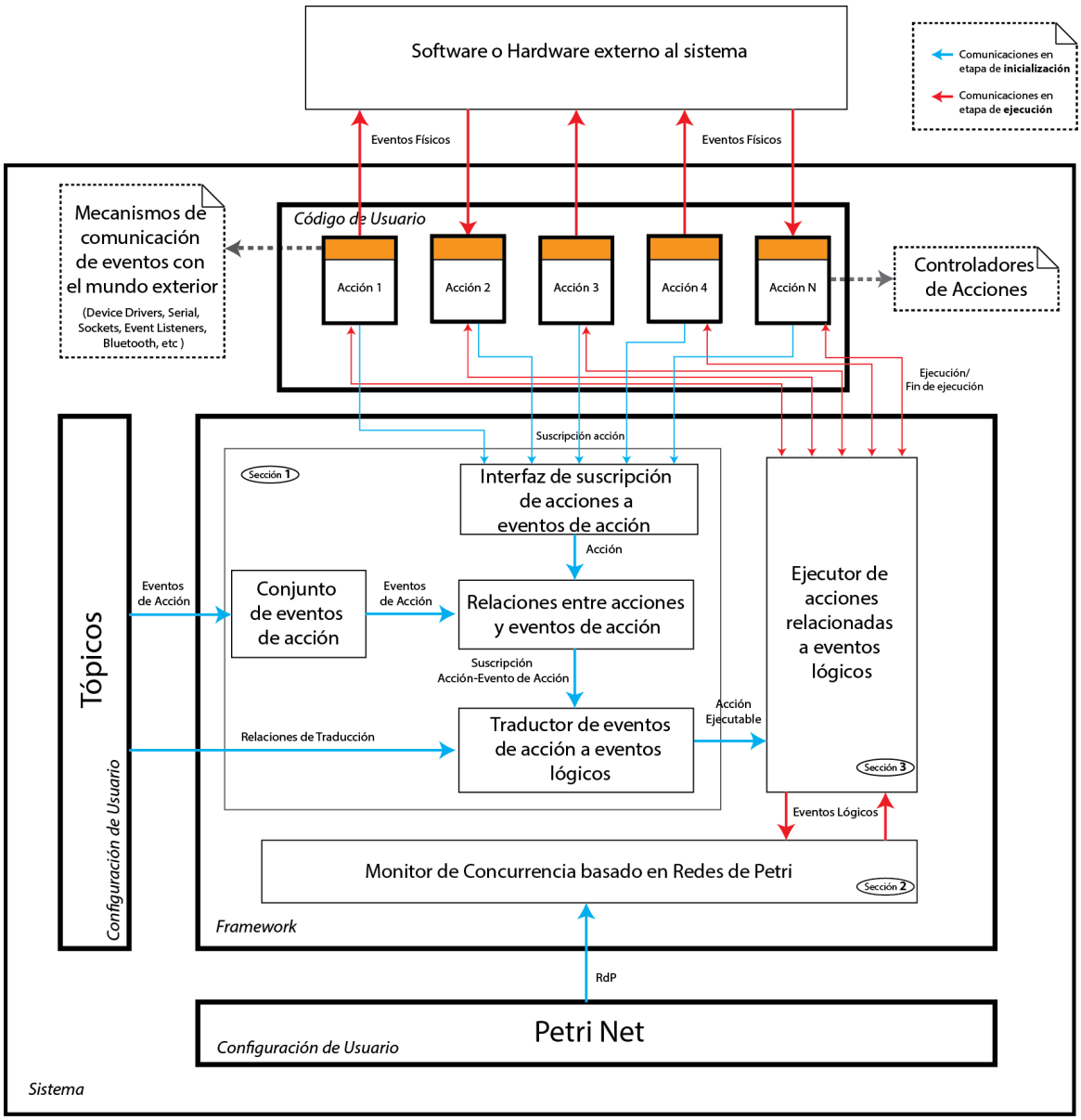


Figura 10.3: Diagrama de Arquitectura de Alto Nivel

Como puede apreciarse en la Figura 10.3, BaboonFramework puede dividirse en tres partes a nivel de arquitectura:

1. Un conjunto de módulos para la suscripción a eventos de acción (ver sección 1) encargado de:
   - Ofrecer interfaces para definir los eventos de acción del sistema.
   - Ofrecer interfaces para suscribir las acciones del sistema a los eventos de acción correspondientes.
   - Ofrecer interfaces para definir las reglas de traducción entre eventos de acción y eventos lógicos.
   - Encapsular las acciones junto a los eventos lógicos necesarios para su sincronización.
2. Un monitor de redes de Petri (ver sección 2) cuyas responsabilidades son:
   - Brindar las interfaces para la incorporación del modelo de Red de Petri del sistema, el cual contiene la definición de los eventos lógicos.
   - Garantizar el cumplimiento de las condiciones de sincronización y exclusión mutua de la concatenación de acciones.
3. Un sistema ejecutor de las acciones definidas por el usuario (ver sección 3). Se encarga de:
   - Intercambiar eventos lógicos con el monitor para asegurar la sincronización y exclusión mutua en la ejecución de las acciones.

- Ejecutar las acciones cuando las condiciones de ejecución estén dadas.
- Intercambiar eventos lógicos con el monitor para informar acerca de la finalización de la ejecución de una acción.

Por su parte, el programa de usuario puede dividirse en:

1. Una Red de Petri conteniendo el modelo de la lógica del sistema.
2. Tópicos que contienen la definición de los eventos de acción y sus reglas de traducción a eventos lógicos.
3. El código de usuario. Contiene todas las acciones de software concretas a realizar, con sus correspondientes suscripciones a eventos de acción. Dichas acciones pueden comunicarse con el mundo exterior. Por esta razón, ***el manejo de los eventos físicos es responsabilidad del usuario***.

Notas:

- Los controladores de acciones se ejecutan cuando el monitor de red de Petri lo dispone. El monitor es el encargado de bloquear o liberar los hilos de una acción de acuerdo a las condiciones de sincronizacion. Por otro lado, cuando una acción finaliza, el sistema de ejecución se encarga de dar aviso al monitor.
- La definición y el desarrollo de las acciones de software, y su asociación a los eventos de acción correspondientes quedan a cargo del usuario desarrollador.
- El usuario no decide en qué momento se ejecuta la acción, ya que con el fin de lograr la inversión de control, dicha responsabilidad es otorgada al monitor de redes de Petri.

10.4. Modos de Sincronización de Acciones utilizando Redes de Petri

En esta sección se analizan dos formas de llevar la sincronización de las acciones mediante la utilización de las interfaces proporcionadas por el monitor de Redes de Petri. Estos modos se denominan:

1. Sincronización por aviso de ejecución.
2. Sincronización por petición de ejecución.

Para estudiar los modos de sincronización mencionados se hará uso de un caso de ejemplo, descripto a continuación:

Ejemplo Cinta transportadora con 3 estaciones. Se depositan piezas en la primer estación de manera asincrónica. Cuando esto sucede, la cinta avanza a la estación 1, donde un operario realiza una transformación a la pieza. Una vez el operario realizó la transformación, presiona un pulsador y la cinta avanza a la estación 2, donde el mismo operario empaqueta la pieza. El operario presiona otro pulsador al finalizar su tarea y luego la cinta avanza una vez más y la pieza se deposita en un contenedor. Una vez que la pieza llega al contenedor se habilita el ingreso de una nueva pieza al proceso.

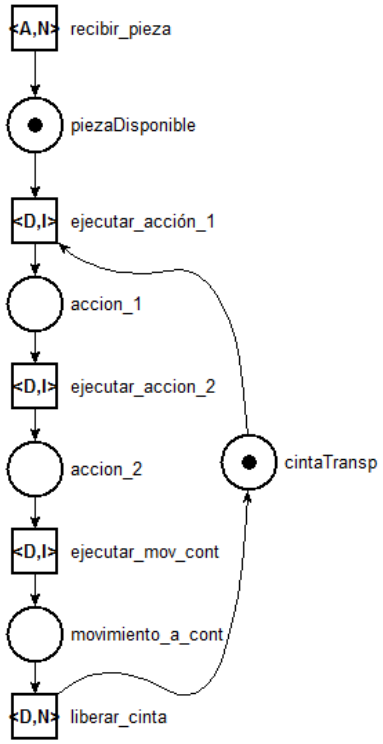


Figura 10.4: Red de Petri de una cinta transportadora

10.4.1. Análisis de ejecución del caso de estudio, utilizando sincronización por aviso de ejecución

De acuerdo al modo de ejecución por aviso, el monitor recibe eventos desde el resto del framework, los procesa y luego desencadena la ejecución de las acciones. A continuación se describe la ejecución del caso de ejemplo siguiendo el modo de sincronización por aviso de ejecución:

1. Debe insertarse un evento en la cola de entrada de "ejecutar_accion_1" cuando la acción que escucha al sensor de pieza disponible detecte la llegada de una nueva pieza. Si la cinta Transportadora se encuentra disponible, el monitor de petri dispara "ejecutar_accion_1" y se genera un evento que se deposita en la cola de salida de "ejecutar_accion_1".
2. Un manejador de eventos lee el evento de salida de "ejecutar_accion_1" y llama a ejecutar la acción "accion_1", que mueve la pieza a la estación 1 y espera el trabajo del operador. Una vez que el operador realiza su trabajo, presiona el pulsador. Esto genera un evento que se envía a la cola de entrada de

"ejecutar_accion_2". El monitor de petri dispara "ejecutar_accion_2" y se genera un evento que se deposita en la cola de salida de "ejecutar_accion_2".

3. El manejador de eventos lee el evento de salida de "ejecutar_accion_2" y llama a ejecutar la acción "accion_2", que mueve la pieza a la estación 2 y espera el trabajo del operador. Una vez que el operador realiza su trabajo, presiona el pulsador. Esto genera un evento que se envía a la cola de entrada de "ejecutar_mov_cont". El monitor de petri dispara "ejecutar_mov_cont" y se genera un evento que se deposita en la cola de salida de "ejecutar_mov_cont".

4. El manejador de eventos lee el evento de salida de "ejecutar_mov_cont" y llama a ejecutar la accion "movimiento_a_cont", que mueve la pieza al contenedor. Una vez terminada esa acción envía un evento a la cola de entrada de "liberar_cinta". El monitor de petri dispara "liberar_cinta" y libera la cinta Transportadora para procesar otra pieza.

A partir del análisis de los pasos de ejecución anteriores, se detectó una desventaja en este modo de sincronización. Al utilizar la suscripción a transiciones informadas el módulo encargado de manejar los eventos provenientes de la Red de Petri asume una parte del control de la lógica de ejecución del sistema. De esta manera, el monitor de Redes de Petri delega parte de su responsabilidad, y se descentraliza el control del sistema. En el caso particular del framework realizado en [AF15], el uso de la sincronización por avisos de ejecución lleva a que el bloqueo y liberación de hilos se realice fuera de la estructura del monitor, desaprovechando una de las principales ventajas de la arquitectura del sistema.

10.4.2. Análisis de ejecución del caso de estudio, utilizando sincronización por petición de ejecución

El modo de sincronización por petición de ejecución es un mecanismo de sincronización alternativo al modo por aviso de ejecución. En este modo, los hilos que ejecutan acciones realizan una petición de ejecución al monitor, sin tener en cuenta el estado actual de la red de Petri. El monitor es el encargado de bloquear aquellos hilos cuyas acciones no pueden ser ejecutadas en el momento de la petición del permiso ejecución. Una vez que las condiciones son las adecuadas para realizar la acción, el monitor libera al hilo encargado de ejecutarla. De esta forma, el manejo de la concurrencia del sistema es realizado íntegramente dentro del monitor. El sistema de ejecución basado en peticiones es más adecuado para una arquitectura controlada por monitor.

1. Se generan eventos que se encolan en la cola de entrada en "ejecutar_accion_1", "ejecutar_accion_2", "ejecutar_mov_cont", y "liberar_cinta".

2. El monitor bloquea los hilos que generaron eventos para "ejecutar_accion_2", "ejecutar_mov_cont", y "liberar_cinta" por no estar sensibilizadas las transiciones en ese momento.

3. El monitor dispara "ejecutar_accion_1". Y se envía un evento a la cola de salida de "ejecutar_accion_1", liberando al hilo bloqueado en dicha transición.

4. El hilo ejecuta la acción "accion_1".

5. Existe un problema, ya que al disparar "ejecutar_accion_1", el monitor tiene permitido disparar "ejecutar_accion_2", pero la operación "accion_1" aun no ha finalizado, generando un problema de sincronización.

Dada la red de petri de la Figura 10.4 surgen problemas de sincronización. Por ejemplo, uno de estos problemas tiene origen al iniciar la acción "accion_1" cuando existe una petición de ejecución de la acción "accion_2". En este caso el monitor otorga el permiso de ejecución de "accion_2" de forma inmediata, sin tener en cuenta si "accion_1" ha finalizado.

Del análisis de este caso se desprenden las siguientes conclusiones:

- La petición de ejecución permite concentrar el control del flujo de ejecución en el monitor.
- Es necesario que el framework dé aviso al monitor de un evento de finalización de acción, cuando existen otras partes de la red que dependen de este evento.

En consecuencia, utilizar un sistema de ejecución basado en peticiones requiere un nuevo modelo en red de Petri del problema, que sea capaz de sincronizar los eventos de finalización de acción. A continuación se procede a estudiar tres formas de sincronizar dichos eventos:

1. Evento de finalización de acción por grupo transición-plaza

2. Evento de finalización de acción por guardas

3. Evento de finalización de acción por cola de condición de disparo no perenne.

10.4.2.1. Evento de Finalización de Acción por Grupo Transición-Plaza

En la Figura 10.5 se observa un modelo de RdP para la sincronización de acciones mediante petición de ejecución, con aviso de finalización de acción por grupo transición-plaza. Este método consiste en añadir una transición y una plaza extra por cada acción que requiera enviar un evento de finalización de acción a la red. Esta transición y plaza de aviso de finalización deben colocarse en cadena con la plaza que representa el estado de ejecución de la acción.

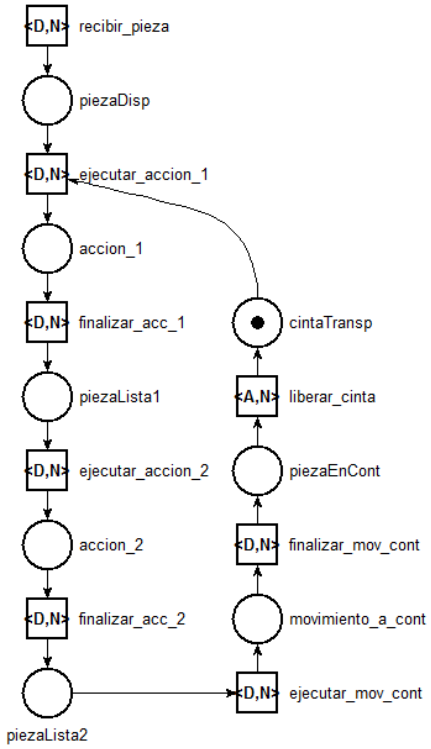


Figura 10.5: Red de Petri de una cinta transportadora sincronizada por inserción de plaza-transición

A continuación se detalla la ejecución de la red de la Figura 10.5:

1. Se generan eventos que se encolan en la cola de entrada en "ejecutar_accion_1", "ejecutar_accion_2", "ejecutar_mov_cont".
2. El monitor bloquea los hilos que generaron eventos para "ejecutar_accion_1", "ejecutar_accion_2", "ejecutar_mov_cont" por no estar sensibilizadas las transiciones en ese momento.
3. Llega una pieza y se genera un evento de entrada en "recibir_pieza"
4. El monitor dispara "recibir_pieza" y se coloca un token en "piezaDisp", sensibilizando "ejecutar_accion_1".
5. El monitor libera el hilo bloqueado en "ejecutar_accion_1" ya que ahora tiene permiso de ejecución.
6. Se ejecuta la acción "accion_1". Una vez finalizado se genera un evento que se envía a la cola de entrada de "finalizar_acc_1".
7. Como "finalizar_acc_1" está sensibilizada el monitor la dispara y se coloca un token en "piezaLista1", sensibilizando "ejecutar_accion_2".
8. El monitor libera el hilo bloqueado en "ejecutar_accion_2" ya que ahora tiene permiso de ejecución.
9. Se ejecuta la acción "accion_2". Una vez finalizado se genera un evento que se envía a la cola de entrada de "finalizar_acc_2".
10. Como "finalizar_acc_2" está sensibilizada el monitor la dispara y se coloca un token en "piezaLista2", sensibilizando "ejecutar_mov_cont".
11. El monitor libera el hilo bloqueado en "ejecutar_mov_cont" ya que ahora tiene permiso de ejecución.

12. Se ejecuta la acción “movimiento_a_cont”. Una vez finalizado se genera un evento que se envía a la cola de entrada de “finalizar_mov_cont”

13. Como “finalizar_mov_cont” está sensibilizada el monitor la dispara y se coloca un token en “piezaEnCont”.

14. Se dispara la transición “liberar_cinta”, que es automática, y se libera el recurso “cintaTransp”.

La principal ventaja de este método consiste en no modificar la semántica de la red y no añadir nuevos conceptos ni cambios en la forma de ejecución. La desventaja más importante es que provoca un incremento considerable de la cantidad de plazas y transiciones de la red, lo que conlleva el procesamiento de matrices de mayor tamaño. En la sección 10.7 se estudia un método que permiten contrarrestar el aumento de tamaño de la RdP para procesos secuenciales.

10.4.2.2. Evento de Finalización de Acción por Guardas

En la Figura 10.6 se observa un modelo de RdP para la sincronización de acciones mediante petición de ejecución, con aviso de finalización de acción por guardas. Este método consiste en la utilización de una guarda como forma de sincronización entre acciones consecutivas.

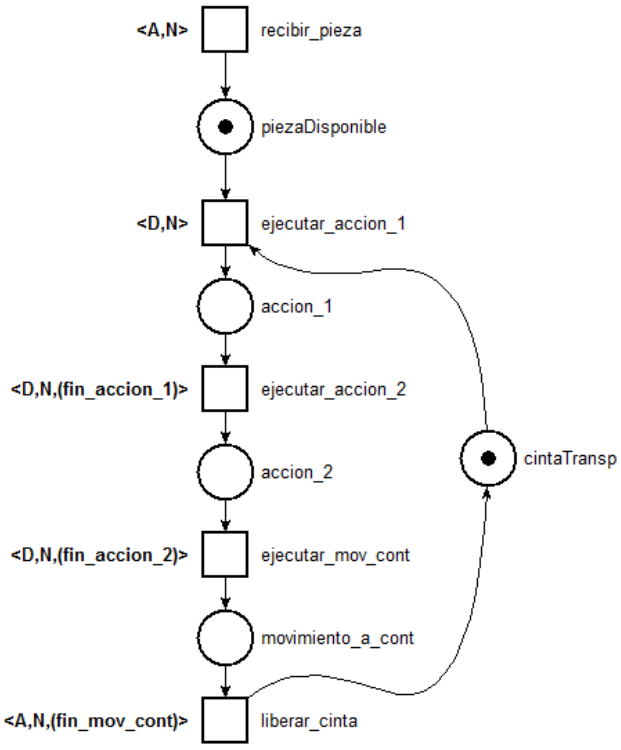


Figura 10.6: Red de Petri de una cinta transportadora sincronizada por guardas.

A continuación se detalla la ejecución de la red de la Figura 10.6

1. Se generan eventos que se encolan en la cola de entrada en “ejecutar_accion_1”, “ejecutar_accion_2”, “ejecutar_mov_cont”.

2. El monitor bloquea los hilos que generaron eventos para “ejecutar_accion_2” y “ejecutar_mov_cont” por no estar sensibilizadas las transiciones en ese momento.

3. Se dispara “ejecutar_accion_1” y se coloca un token en “accion_1”. Comienza la ejecución de la acción “accion_1”. La transición “ejecutar_accion_2’ no se encuentra sensibilizada dado que la guarda “fin_accion_1” tiene estado “false”.

4. Finaliza la ejecución de “accion_1” y se establece la guarda “fin_accion_1” con estado “true”.

5. Al estar sensibilizada “ejecutar_accion_2”, se dispara y se libera el hilo bloqueado en su cola de condición. Se coloca un token en “accion_2” y comienza la ejecución de esta acción. La transición “ejecutar_mov_cont” no se encuentra sensibilizada dado que la guarda “fin_accion_2” tiene estado “false”. Se establece la guarda “fin_accion_1” a “false” nuevamente.

6. Finaliza la ejecución de “accion_2” y se establece la guarda “fin_accion_2” con estado “true”.

7. Al estar sensibilizada “ejecutar_mov_cont”, se dispara y se libera el hilo bloqueado en su cola de condición. Se coloca un token en “movimiento_a_cont” y comienza la ejecución de esta acción. La transición “liberar_cinta” no se encuentra sensibilizada dado que la guarda “fin_mov_cont” tiene estado “false”. Se establece la guarda “fin_accion_1” a “false” nuevamente.

8. Finaliza la ejecución de “movimiento_a_cont” y se establece la guarda “fin_mov_cont” con estado “true”.

9. Al estar sensibilidada, se dispara la transición “liberar_cinta”, que es automática, y se libera el recurso “cintaTransp”. Se establece la guarda “fin_mov_cont” a “false” nuevamente.

La ventaja de este método es que permite resolver el problema de sincronización sin aumentar la cantidad de componentes de la red de Petri. Como desventaja se puede mencionar que el diseño del monitor de petri soporta una única guarda por transición, por lo tanto esta solución impide la utilización de la guarda para otros propósitos. Otra desventaja importante de la utilización de guardas es que al ser un valor binario, lleva a una pérdida de eventos de finalización cuando múltiples hilos realizan una misma acción.

Con el fin de ejemplificar la pérdida de eventos de finalización al utilizar sincronización de finalización de acción por guardas se analiza el siguiente caso de ejemplo:

Ejemplo Una “tarea A” es realizada por multiples hilos de manera independiente, y cada hilo realiza la “tarea A” en su totalidad. A su vez una “tarea B”, que debe realizarse luego de la finalización de la “tarea A”, es ejecutada por un único hilo. En este caso la utilización de guardas podría llevar a una pérdida de eventos de finalización de la “tarea A” debido a la condición binaria de la guarda. En la red de la Figura 10.7, un máximo de 5 hilos puede ejecutar la “tarea A” al mismo tiempo. En el estado que muestra la figura existen tres hilos corriendo la “tarea A”. De acuerdo al planteo de este problema, la “tarea B” es ejecutada por un único hilo. Cuando dos o más hilos finalizan la “tarea A” y establecen la guarda “Fin_TareaA” en “true”, existe la posibilidad de que otro hilo dispare “t1” antes de comenzar la ejecución de la “tareaB”. En este momento, el hilo que dispara “t1” modifica el valor de la guarda “Fin_TareaA” a “false”, sobreescribiendo el aviso de finalización de acción de los hilos que ya habían establecido “Fin_TareaA” en “true”. De esta manera, se pierden eventos de aviso de fin de ejecución, provocando la incorrecta sincronización del sistema.

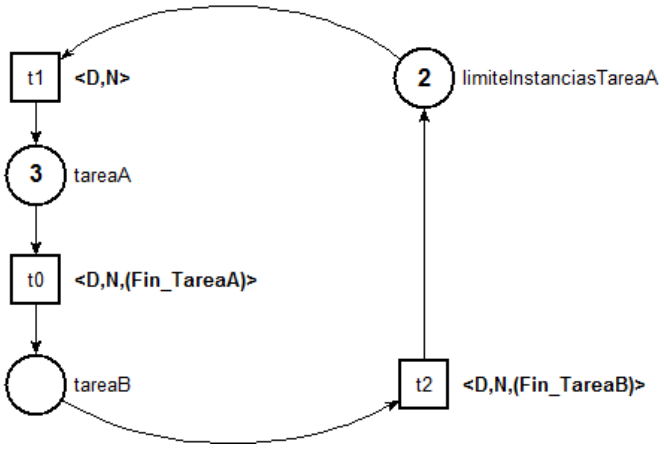


Figura 10.7: RdP: Problema de sincronización de acciones dependientes usando guardas, debido a su condición binaria

10.4.2.3. Evento de Finalización de Acción por Cola de Condición de Disparo No Perenne

En la Figura 10.8 se observa un modelo de RdP para la sincronización de acciones mediante petición de ejecución, con aviso de finalización de acción por cola de condición de disparo no perenne. Esta forma de solucionar la sincronización de acciones dependientes supone añadir una nueva propiedad “P” a las transiciones. Los hilos bloqueados en la cola de condición de una transición con propiedad “P” sólo se liberan cuando la transición se encuentra habilitada y además un hilo externo realiza un disparo no perenne sobre la transición.

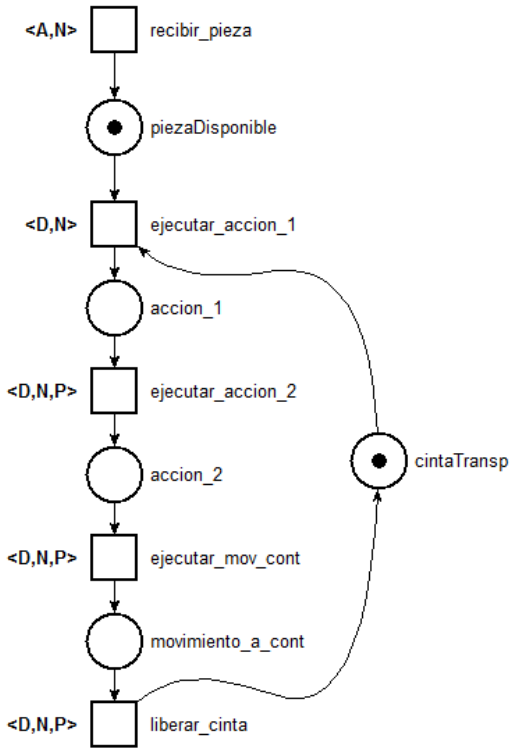


Figura 10.8: Red de Petri de una cinta transportadora sincronizada por propiedad "P".

1. Se generan eventos que se encolan en la cola de entrada en "ejecutar_accion_1", "ejecutar_accion_2", "ejecutar_mov_cont", y "liberar_cinta".

2. El monitor bloquea los hilos que generaron eventos para "ejecutar_accion_2", "ejecutar_mov_cont", y "liberar_cinta" por no estar sensibilizadas las transiciones en ese momento.

3. Se dispara "ejecutar_accion_1' y se coloca un token en "moverEst1". Comienza la ejecución de la acción "accion_1". La transición "ejecutar_accion_2" no se dispara ya que es de tipo "P".

4. Finaliza la ejecución de "accion_1" y un hilo dispara "ejecutar_accion_2" de forma no perenne para dar aviso de la finalización de la acción.

5. Se libera el hilo bloqueado en cola de condición de "ejecutar_accion_2". Se coloca un token en "accion_2" y comienza la ejecución de esta acción. La transición "ejecutar_mov_cont" no se dispara ya que es de tipo "P".

6. Finaliza la ejecución de "accion_2" y un hilo dispara "ejecutar_mov_cont" de forma no perenne para dar aviso de la finalización de la acción.

7. Se libera el hilo bloqueado en cola de condición de "ejecutar_mov_cont". Se coloca un token en "movimiento_a_cont" y comienza la ejecución de esta acción. La transición "liberar_cinta" no se dispara ya que es de tipo "P".

8. Finaliza la ejecución de "movimiento_a_cont" y un hilo dispara "liberar_cinta" de forma no perenne para dar aviso de la finalización de la acción.

9. Se libera el recurso "cintaTransp".

La ventaja de esta solución es que no incrementa el numero de componentes de la red. Su principal desventaja consiste en que añade una nueva etiqueta a la RdP, dificultando su validación matemática. Esta solución supone añadir una interfaz al monitor de Petri para bloquear hilos en una cola de condición de una transición tipo P y que los hilos bloqueados en esta cola de condición sólo puedan liberarse por medio de un disparo no perenne ocasionado por un hilo externo. El hilo que realiza el disparo sobre la transición debe realizar una operación *release* sobre la cola de condición de disparos no perennes (sin importar si existen o no hilos bloqueados en la cola) para evitar la pérdida de eventos de finalización de acción.

10.4.3. Resumen de Modos de Sincronización

Existen dos maneras de coordinar la ejecución de las acciones a partir de una red de Petri.

1. **Sincronización por aviso de ejecución:** Consiste en suscribir una acción a una transición, que envía una notificación en el momento en que es disparada. Ante un informe de esta transición, la acción suscripta comienza su ejecución. Al finalizar una acción, se dispara la transición a la cual está suscripta la siguiente acción a ejecutar. La principal desventaja de este método consiste en la descentralización del manejo de los hilos por parte del monitor, lo cual va en contra de los objetivos del proyecto.
2. **Sincronización por petición de ejecución:** Los hilos que ejecutan acciones realizan una peticion de ejecución al monitor (mediante el disparo de transiciones), sin tener en cuenta el estado actual de la red de Petri. De este modo el monitor bloquea los hilos que no pueden ejecutarse y libera los hilos que cumplan con las condiciones de ejecución. El manejo de la concurrencia es llevado a cabo íntegramente por el monitor. Para sincronizar acciones dependientes entre sí (una debe comenzar luego de la finalización de la otra) se requiere un modo de dar aviso al monitor de la finalización de una acción. Se analizan tres opciones:

   - Evento de finalización de acción por grupo transición-plaza
   - Evento de finalización de acción por guardas
   - Evento de finalización de acción por cola de condición de disparo no perenne.

   Se optó por adoptar el evento de finalización de acción por grupo transición-plaza. Esta forma presenta la ventaja de no añadir conceptos nuevos a la RdP, facilitando el entendimiento de la misma. Además no presenta pérdida de eventos de finalización de acción. La principal desventaja del grupo transición-plaza consiste en el incremento del tamaño de la RdP. En procesos secuenciales, puede contrarrestarse mediante el uso del controlador de acciones descripto en la sección 10.7.

10.5. Clasificación de Eventos Físicos: Eventos Task y Eventos Happening

El monitor de RdP es el único responsable del bloqueo o habilitación de un hilo que ejecuta una acción emisora de eventos físicos de salida. El hilo realiza el disparo de petición de ejecución al inicio de la ejecución del programa, sin tener en cuenta el estado actual de la red (ver sección 10.4.2). En consecuencia, el hilo queda bloqueado en el monitor hasta que el mismo decide desbloquearlo, dependiendo del estado de la red y la política de prioridades (ver Figura 10.9).

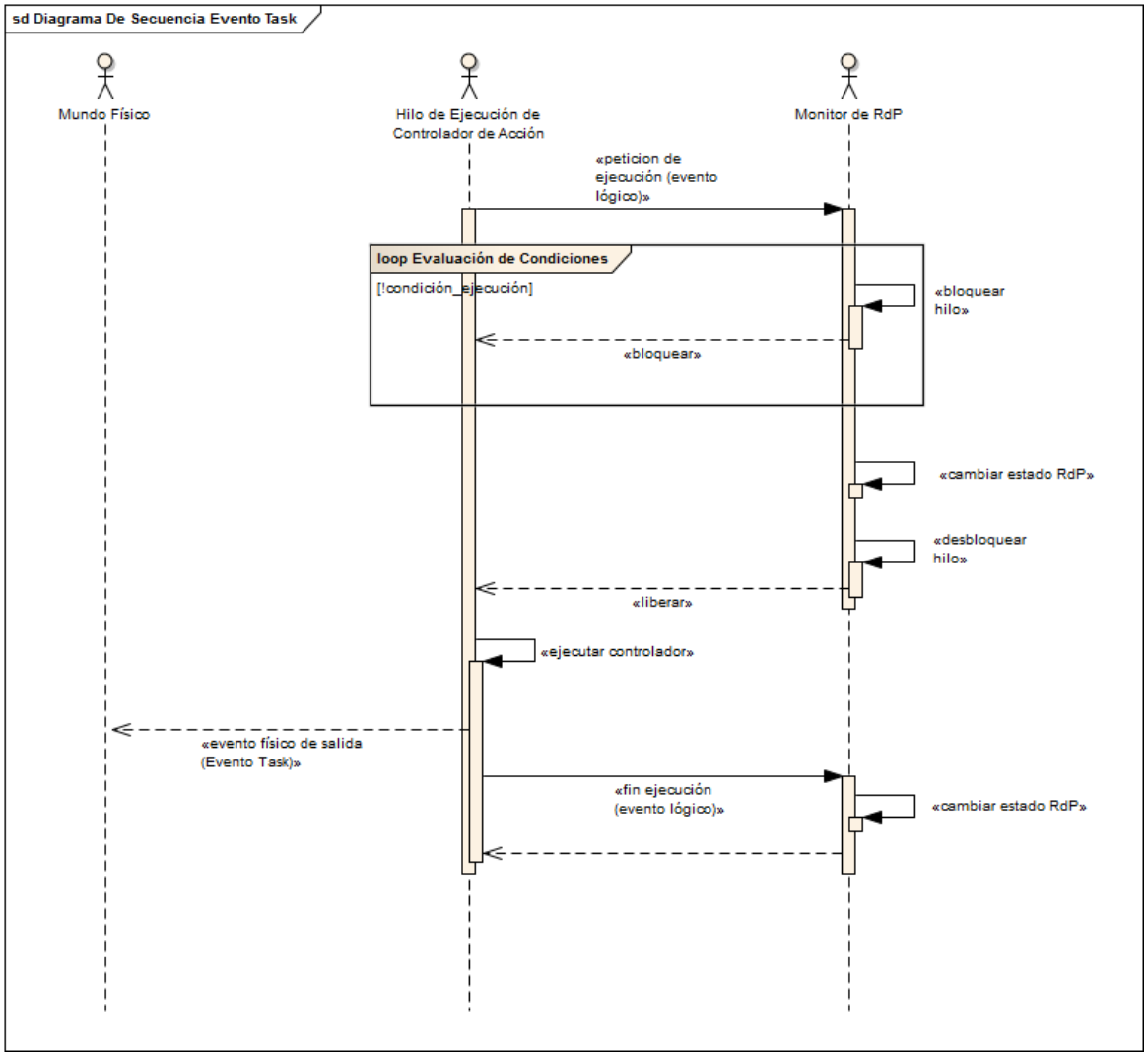


Figura 10.9: Diagrama de Secuencia de la Ejecución de una Acción que Emite un Evento Físico de Salida

En la Figura 10.10 se observa que la ejecución de una acción encargada de recibir un evento físico de entrada depende, en primera instancia de la ocurrencia de dicho evento. Luego, también es sincronizada con la RdP mediante el uso del monitor. De este modo, las acciones que reciben eventos físicos de entrada presentan una restricción extra respecto a aquellas que emiten eventos físicos de salida.

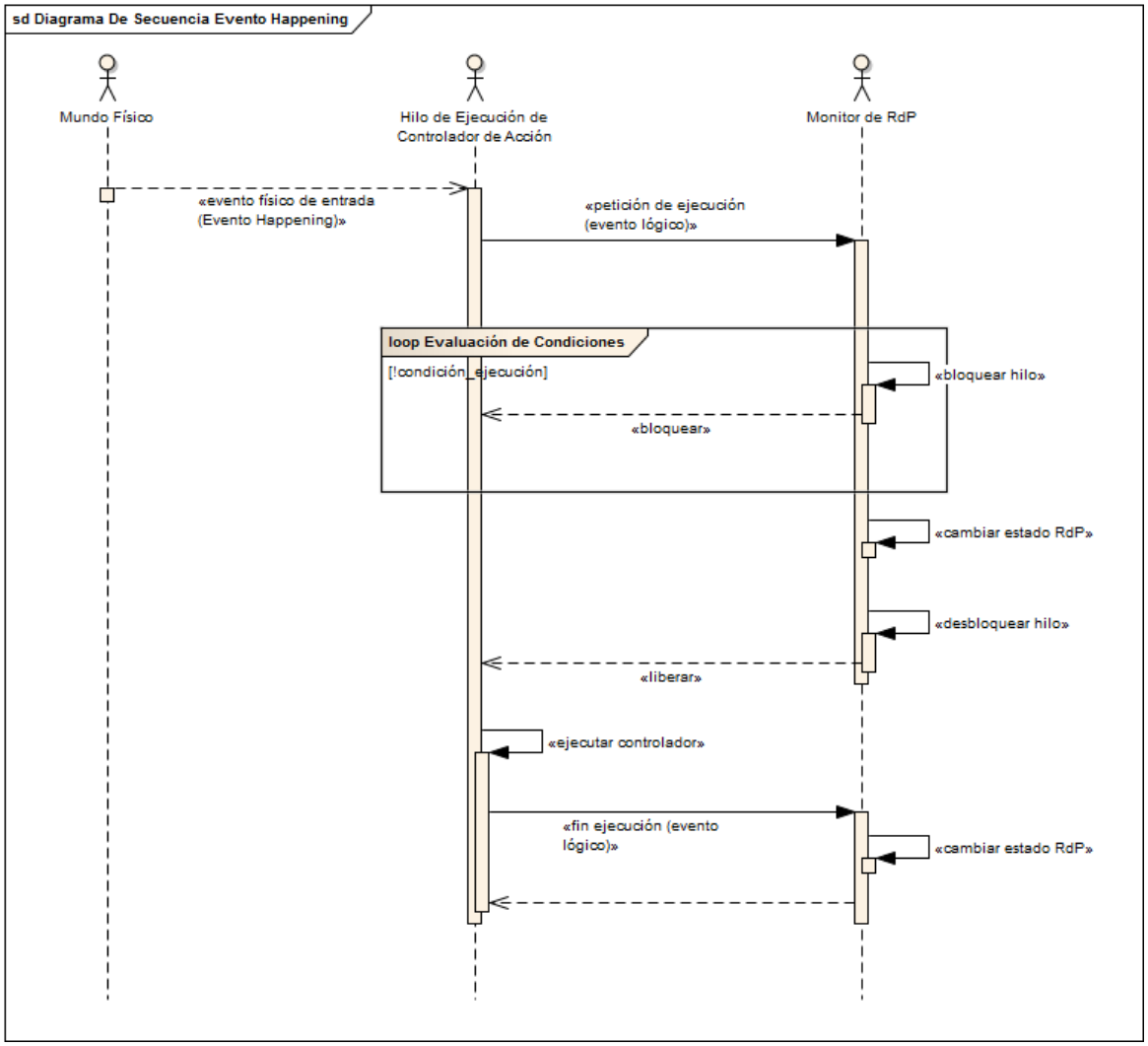


Figura 10.10: Diagrama de Secuencia de la Ejecución de una Acción que Recibe un Evento Físico de Entrada

Desde este punto de vista, se decidió dar una clasificación más significativa a los eventos físicos. Esta clasificación estará presente a lo largo de todo el desarrollo del framework.

- Eventos Task (Tarea): Eventos físicos de salida. Son desencadenados y sincronizados exclusivamente por eventos lógicos que dependen de condiciones ya presentes en el monitor de Redes de Petri al momento de su emisión. Si bien el monitor no produce directamente eventos físicos, puede advertirse que un evento task tiene una relación directa con determinados eventos lógicos. Dichos eventos lógicos se encuentran definidos en un tópico (evento de acción).
- Eventos Happening (Suceso): Eventos físicos de entrada. Son desencadenados por el mundo externo de manera totalmente asincrónica respecto al sistema. La ejecución de las acciones que reciben y manejan estos eventos es sincronizada por el monitor de redes de petri. De esta forma, el monitor conserva su responsabilidad frente al manejo del asincronismo del sistema. Los eventos lógicos requeridos para la sincronización se encuentran definidos en un tópico (evento de acción).

10.6. Controladores de Acciones: Task Controllers y Happening Controllers

Las acciones que debe realizar el sistema, junto con las interfaces necesarias para la comunicación de los eventos físicos correspondientes, se encuentran embebidos dentro de controladores de acción. En la Figura 10.3 se muestran dichos controladores.

A partir de la clasificación de eventos físicos propuesta en la sección 10.5, surge la necesidad de clasificar los controladores de acción respecto a su condición de receptores de Eventos Happening, o de emisores de Eventos

Task. En consecuencia, emerge un nuevo requerimiento del framework, relacionado con el requerimiento número 2 definido en la sección 8.3:

- El framework debe ofrecer interfaces para especificar si un controlador de acción responde a un evento físico de entrada o a un evento físico de salida.

En respuesta a este requerimiento se definen los controladores de tipo Happening Controller y Task Controller.

10.6.1. Ejecución de un Task Controller

De acuerdo al diseño del framework, el comportamiento dinámico de los eventos Task es dirigido únicamente por la evolución de los estados de la RdP. Esto se debe a las siguientes condiciones:

- En la sección 10.4.3 se explicita que el modo de sincronización adoptado es el de petición de ejecución al monitor. De esta forma se admiten disparos asíncronos a transiciones realizados desde diferentes hilos de ejecución, delegando en el monitor la responsabilidad de bloquear los hilos que no cumplen con las condiciones necesarias para la ejecución.
- En la sección 10.5 se determina la existencia de una relación directa entre un Evento Task y un conjunto de eventos lógicos.
- Un Evento Task es desencadenado por condiciones que estan presentes en el monitor al momento de la emisión de este evento

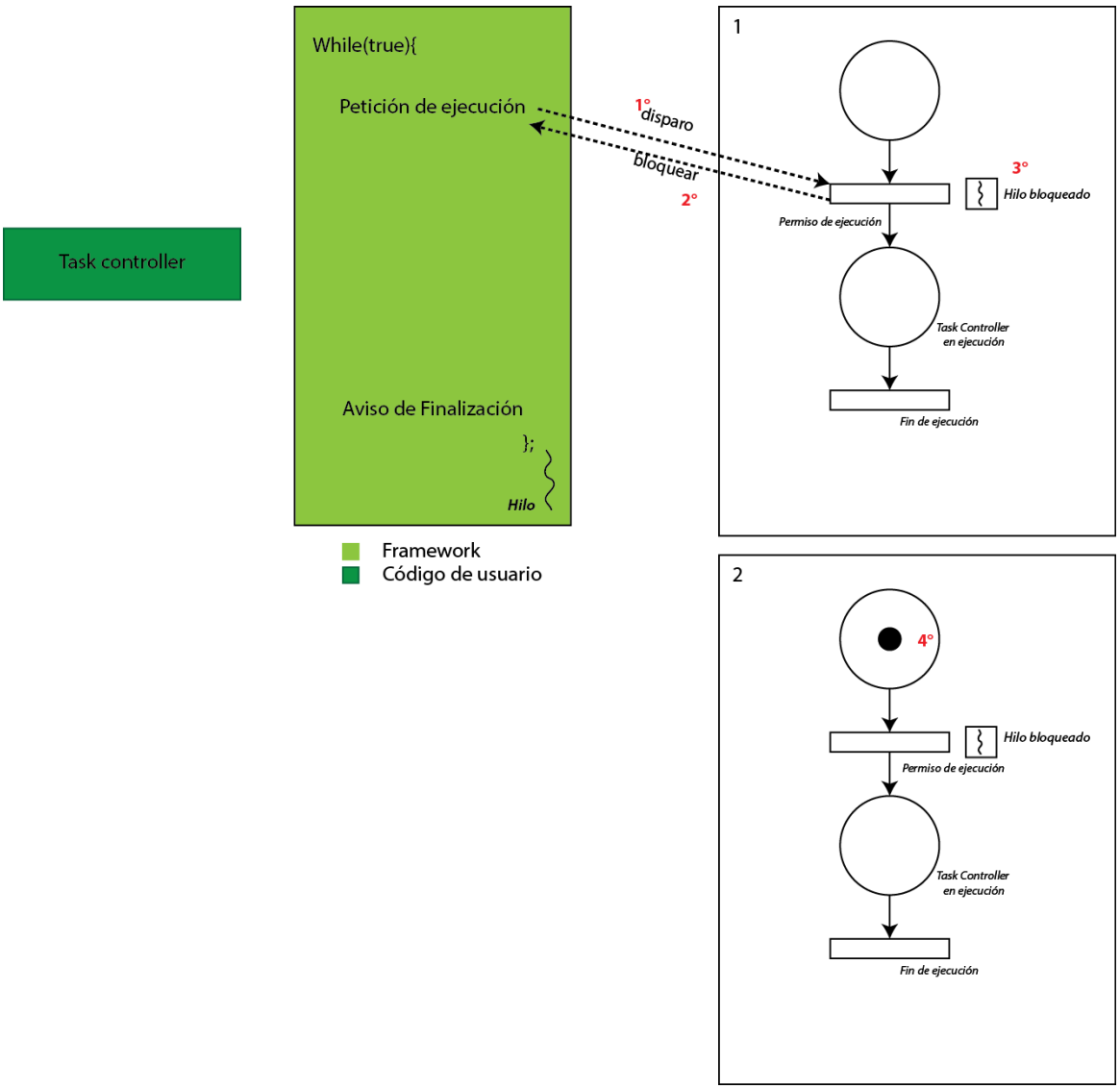
Del analisis anterior emerge un nuevo requerimiento, relacionado con el requerimiento 2 de la sección 8.3:

- El framework debe ser responsable de crear y controlar la ejecución de los hilos de ejecución para controladores de acción que generan eventos físicos de salida.

Como corolario, se determina que la ejecución de un Task Controller debe encapsularse en un hilo al momento de inicialización del programa. El hilo se encarga de:

1. Realizar las peticiones de los permisos de sincronización al monitor de forma directa, sin tener en cuenta el estado de la red.
2. Ejecutar el código correspondiente al Task Controller, una vez que el monitor otorga el permiso.
3. Realizar el aviso de finalización de ejecución al monitor de Petri.
4. Repetir los pasos de ejecución de forma infinita, delegando en el monitor el control de la ejecución.

La responsabilidad de la creación de los hilos para ejecutar los Task Controllers pertenece al framework. En consecuencia, la ejecución de las acciones encapsuladas en un Task Controller es transparente al usuario desarrollador.


Task controller
While(true){
Petición de ejecución
Aviso de Finalización
};
Hilo
1°disparo
bloquear
2°
Framework
Código de usuario
1
3°
Hilo bloqueado
Permiso de ejecución
Task Controller en ejecución
Fin de ejecución
2
4°
Hilo bloqueado
Permiso de ejecución
Task Controller en ejecución
Fin de ejecución

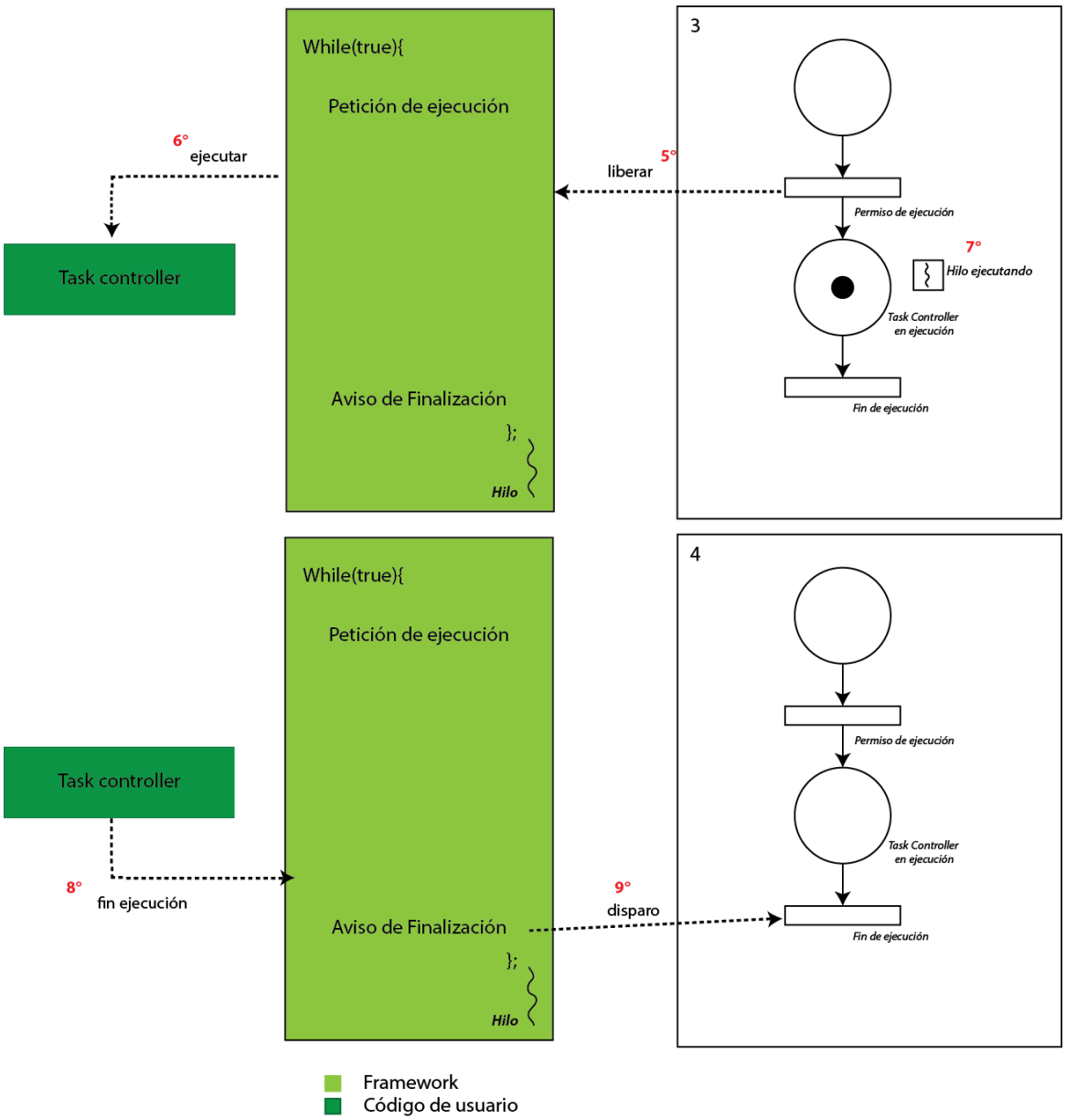


Figura 10.11: Pasos de la Ejecución de un Task Controller

10.6.2. Ejecución de un Happening Controller

Un Evento Happening se desencadena en el mundo externo y de forma asíncrona al sistema (ver sección 10.5). En consecuencia, el código de usuario es responsable de realizar la llamada a ejecución de un Happening Controller, encargado de manejar dicho evento.

En principio, realizar una llamada desde el código de usuario podría generar un grave problema en cuanto a las responsabilidades de sincronización. Las acciones del Happening Controller quedarían fuera del mecanismo de sincronización del monitor de Petri, atentando contra el objetivo de delegar el asincronismo del sistema en la RdP.

Del análisis anterior emerge un nuevo requerimiento, relacionado con el requerimiento 2 de la sección 8.3:

- El framework debe ser responsable de controlar la ejecución de los hilos creados por el usuario, correspondientes a controladores de acción que manejan eventos físicos de entrada.

Este requerimiento se cumple mediante la utilización de herramientas de la programacón orientada a aspectos (ver sección 4.3). Se optó por encapsular las instrucciones de sincronización necesarias dentro de advices. Dichos advices son aplicados en joinpoints, definidos por los puntos de ejecución del programa donde existen llamadas a ejecución o retornos de rutinas de tipo Happening Controller.

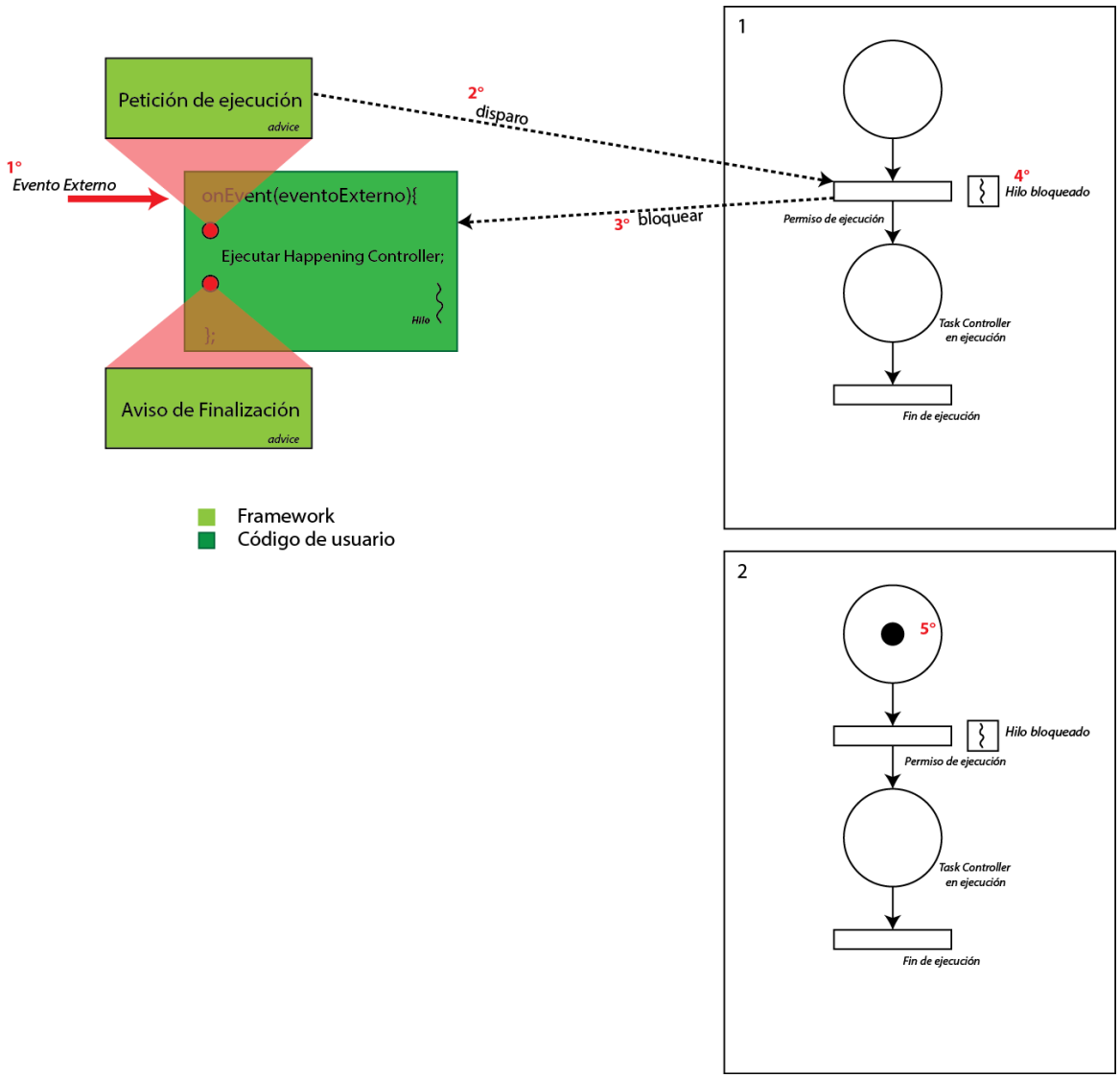
Nota: Un controlador de acción que reacciona ante eventos Happening (Happening Controller) debe ser ejecutado dentro del contexto de un Listener u Observer del evento deseado. Para maximizar las libertades de elección del desarrollador en la recepción de eventos externos, la responsabilidad de crear estos Listeners se delega en el usuario.

El flujo de ejecución de un Happening Controller es el siguiente:

1. Cuando el código de usuario hace un llamado a la ejecución del Happening Controller, la aplicación alcanza un joinpoint.
2. En el momento en que se alcanza el joinpoint se aplica un advice que realiza los pedidos de permiso de sincronización al monitor. Dicho advice es aplicado automáticamente en tiempo de compilación, por lo tanto el usuario no tiene la responsabilidad de realizar la sincronización manualmente. De esta manera se logra conservar la inversión de control.
3. Una vez que el monitor libera el hilo, indicando que la ejecución es posible, se procede a ejecutar el código del controller.
4. Al finalizar la ejecución del Happening Controller, se alcanza un nuevo joinpoint.
5. En el momento en que se alcanza el joinpoint descripto en el punto anterior se aplica un advice que realiza el aviso de finalización de ejecución al monitor. Este advice se aplica de manera análoga a la descripta en el punto 2.

Nota: Los advices descriptos en esta sección son ejecutados por el mismo hilo que ejecuta el código del Happening Controller. Dado que la llamada al código del Happening Controller es responsabilidad del usuario, es él quien debe generar el hilo encargado de ejecutar el código. Se recomienda no utilizar el hilo principal del programa para ejecutar Happening Controllers ya que puede ocasionar el bloqueo del sistema.

En conclusión, la sincronización de la ejecución de los Happening Controllers sigue siendo una responsabilidad del monitor y es realizada de forma transparente al usuario. Es posible realizar La llamada a un HappeningController en cualquier punto del código de la aplicación, de esta forma el usuario tiene libertad para realizar el manejo de eventos físicos de entrada. Se destaca que aunque el usuario tiene la libertad de llamar al código del Happening Controller en cualquier momento, el monitor es responsable de determinar si la ejecución del controlador comenzará al momento de la llamada a ejecución, o si bloqueará el hilo por no cumplirse las condiciones de ejecución del controlador.


Petición de ejecución
advice
1°
Evento Externo
onEvent(eventoExterno){
Ejecutar Happening Controller;
Hilo
};
Aviso de Finalización
advice
2°
disparo
3°
bloquear
Framework
Código de usuario
1
4°
Hilo bloqueado
Permiso de ejecución
Task Controller
en ejecución
Fin de ejecución
2
5°
Hilo bloqueado
Permiso de ejecución
Task Controller
en ejecución
Fin de ejecución

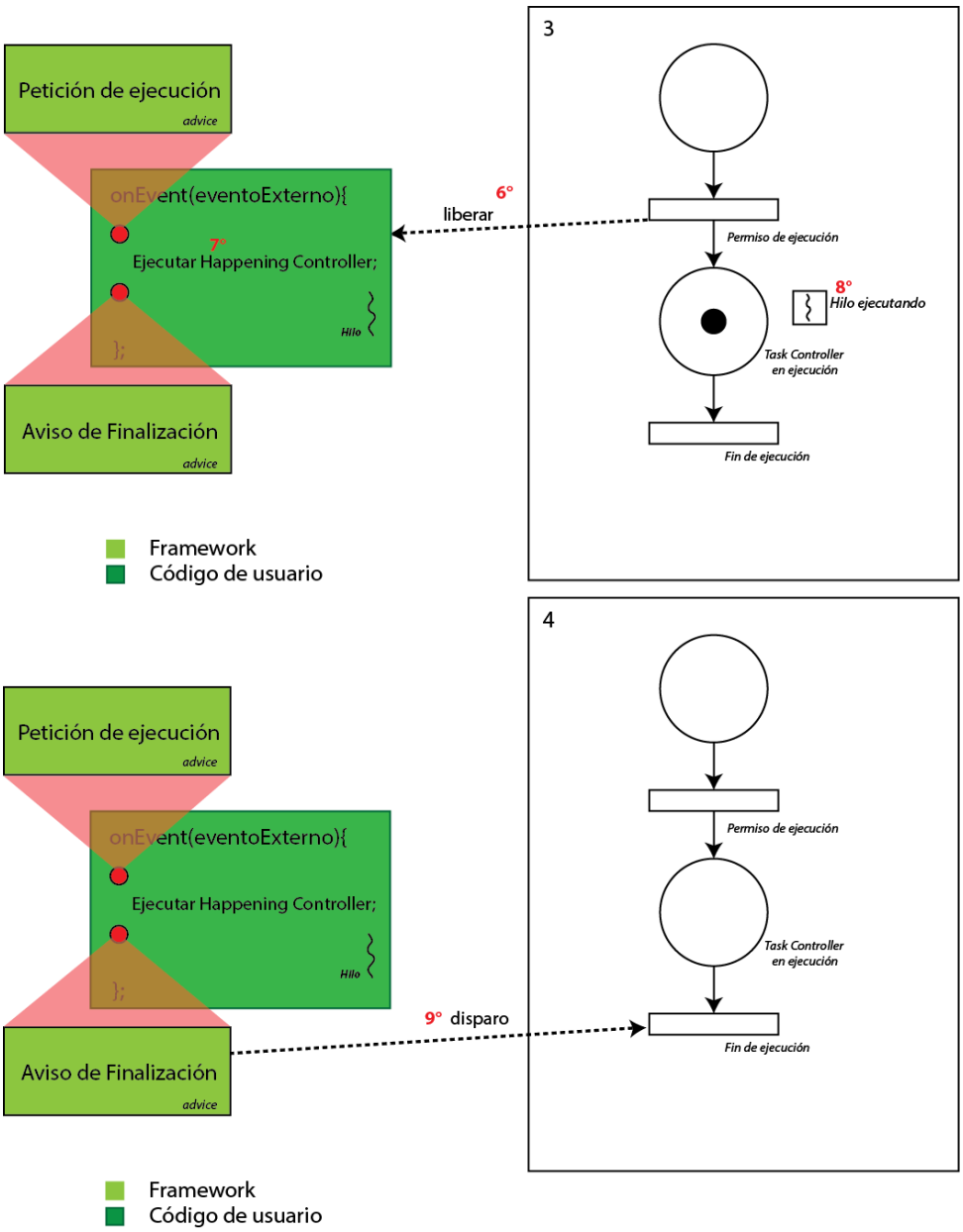


Figura 10.12: Pasos de la Ejecución de un Happening Controller

10.7. ComplexSequentialTaskController

En esta sección se presenta una optimización de la ejecución de Task Controllers (ver sección 10.6.1) para sistemas de caracter secuencial, como el que se estudia en la sección 10.4. Debido a que no existe un paralelismo aprovechable en una secuencia de tareas dependientes resulta innecesario utilizar un hilo por cada Task Controller correspondiente a cada tarea secuencial.

En respuesta a este problema, se determinó que los Task Controllers de acciones correspondientes a un proceso secuencial deben ser ejecutados por un mismo hilo. Esta agrupación de controladores se denomina Complex Sequential Task Controller.

La utilización de Complex Sequential Task Controllers reduce la cantidad de hilos necesarios para ejecutar un sistema con tareas secuenciales. Para lograrlo, se incorpora una etapa de petición de permisos y una etapa de ejecución de controlador por cada Task Controller que forma parte del ComplexSequentialTaskController. Cada una de estas etapas debe seguir el orden preestablecido por la secuencia de acciones que conforman el proceso secuencial. Dicho orden se establece en el tópico, al definir el evento de acción para la sincronización de la tarea compleja.

Una ventaja de agrupar tareas secuenciales en un mismo hilo es que permite eliminar la plaza-transición incorporada por el Modo de Sincronización de petición de de ejecución (ver sección 10.4.2.1).

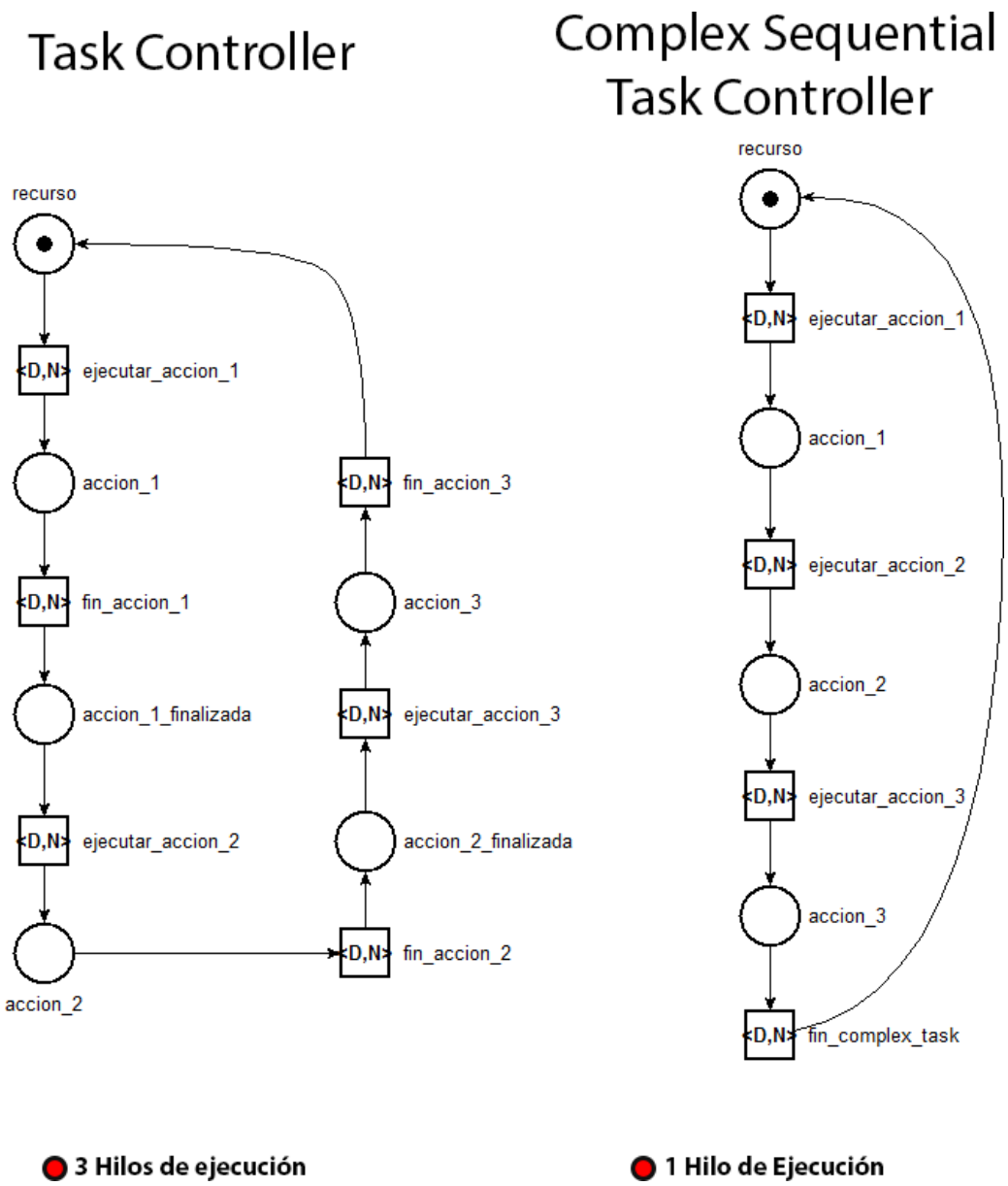
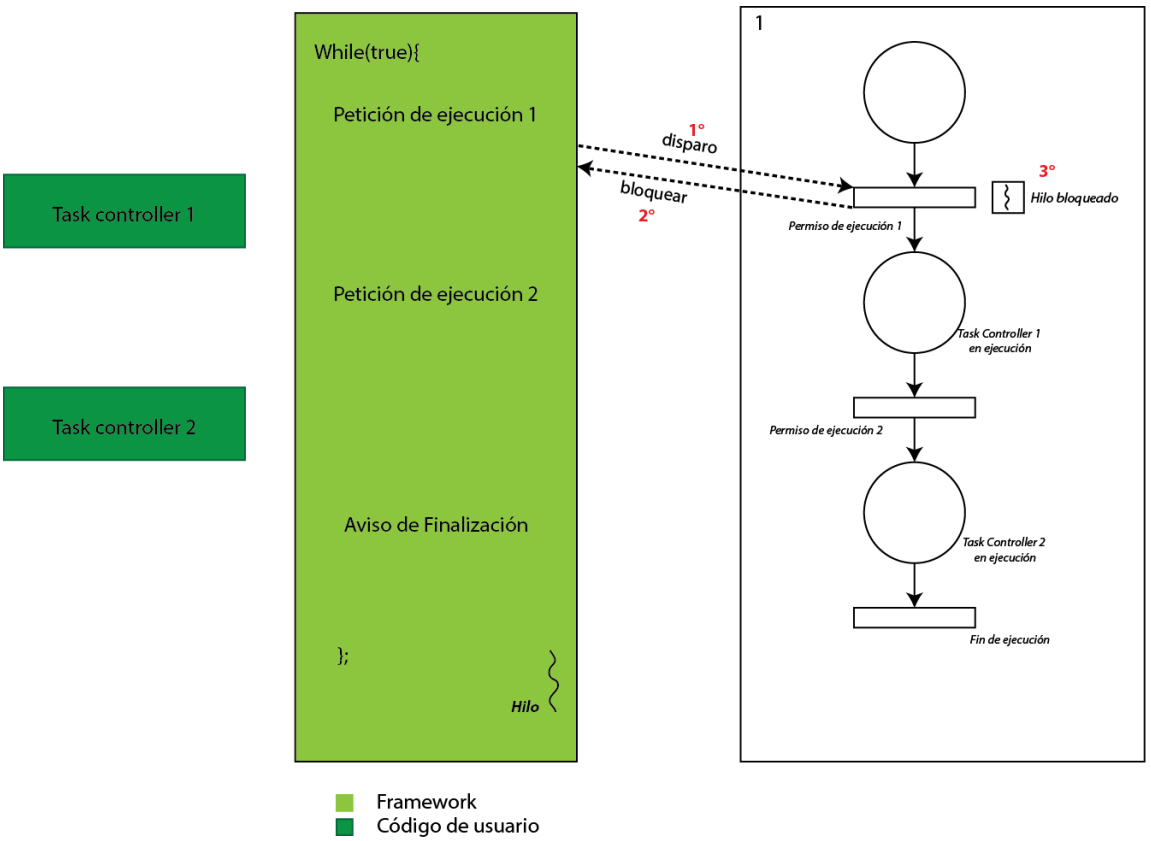
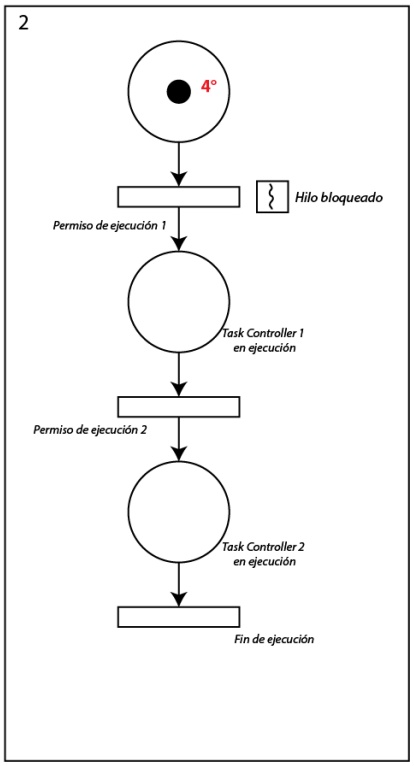
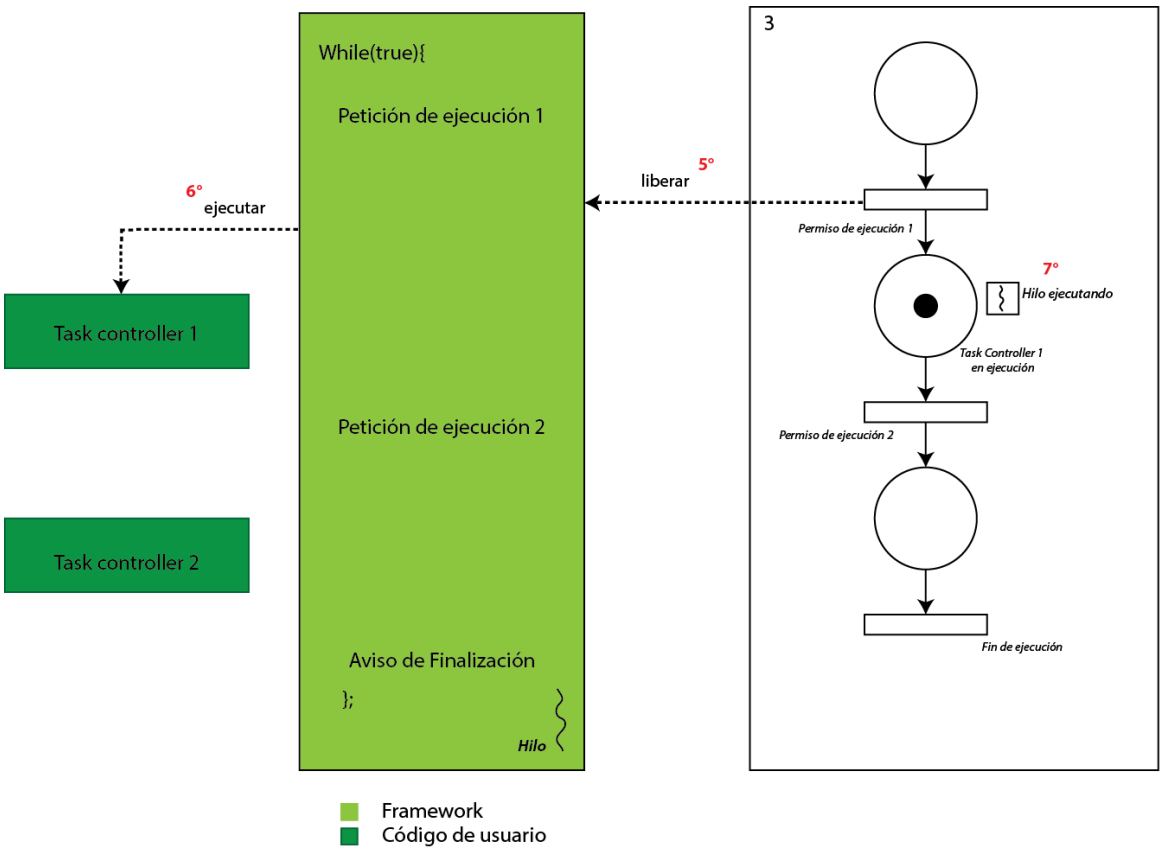
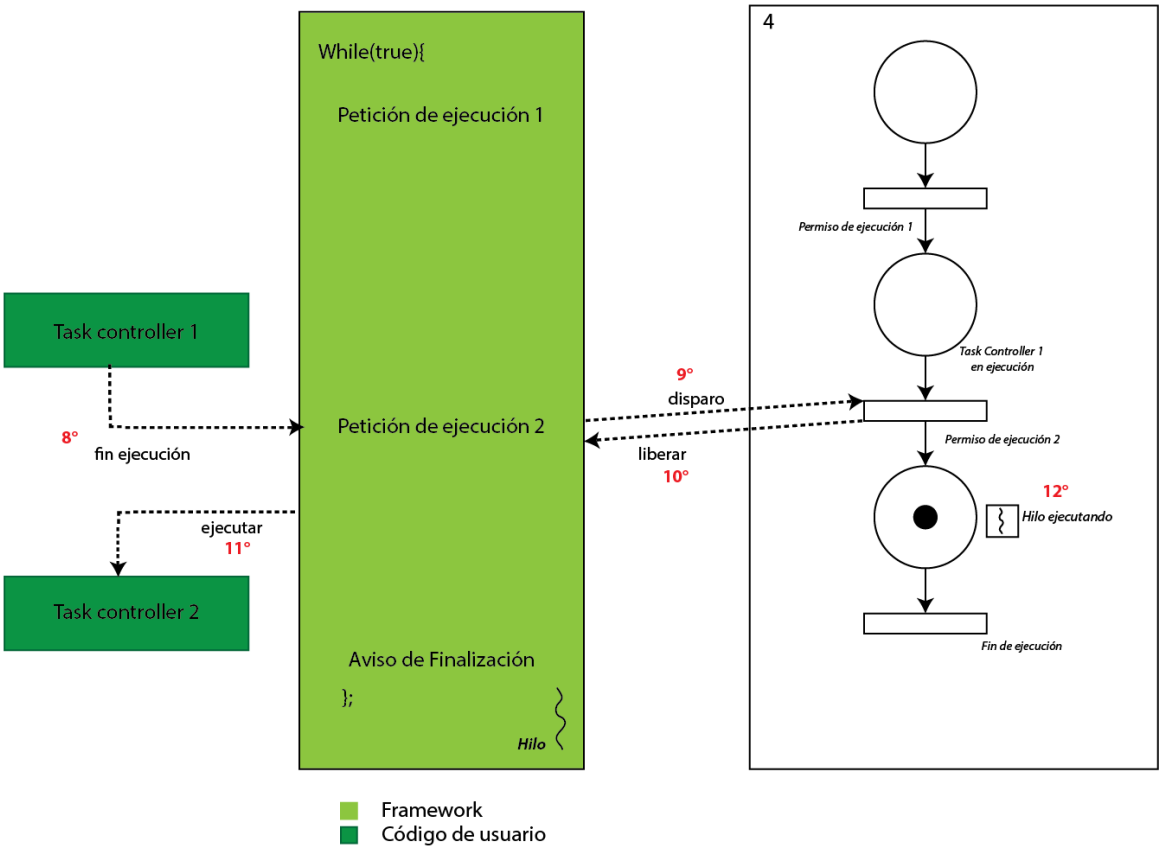


Figura 10.13: Comparación del modelo en RdP de un sistema con tres acciones secuenciales para Task Controllers simples y para Complex Sequential Task Controller.


Task controller 1
Task controller 2
While(true){
Petición de ejecución 1
Petición de ejecución 2
Aviso de Finalización
};
Hilo
1°
disparo
2°
bloquear
1
3°
Hilo bloqueado
Permiso de ejecución 1
Task Controller 1
en ejecución
Permiso de ejecución 2
Task Controller 2
en ejecución
Fin de ejecución
Framework
Código de usuario


2
4°
Hilo bloqueado
Permiso de ejecución 1
Task Controller 1
en ejecución
Permiso de ejecución 2
Task Controller 2
en ejecución
Fin de ejecución


While(true){
Petición de ejecución 1
Petición de ejecución 2
Aviso de Finalización
};
Hilo
Task controller 1
Task controller 2
6°
ejecutar
5°
liberar
3
Permiso de ejecución 1
7°
Hilo ejecutando
Task Controller 1
en ejecución
Permiso de ejecución 2
Fin de ejecución
Framework
Código de usuario


While(true){
Petición de ejecución 1
Petición de ejecución 2
Aviso de Finalización
};
Hilo
Task controller 1
8°
fin ejecución
ejecutar
11°
Task controller 2
9°
disparo
liberar
10°
4
Permiso de ejecución 1
Task Controller 1
en ejecución
Permiso de ejecución 2
12°
Hilo ejecutando
Fin de ejecución
Framework
Código de usuario

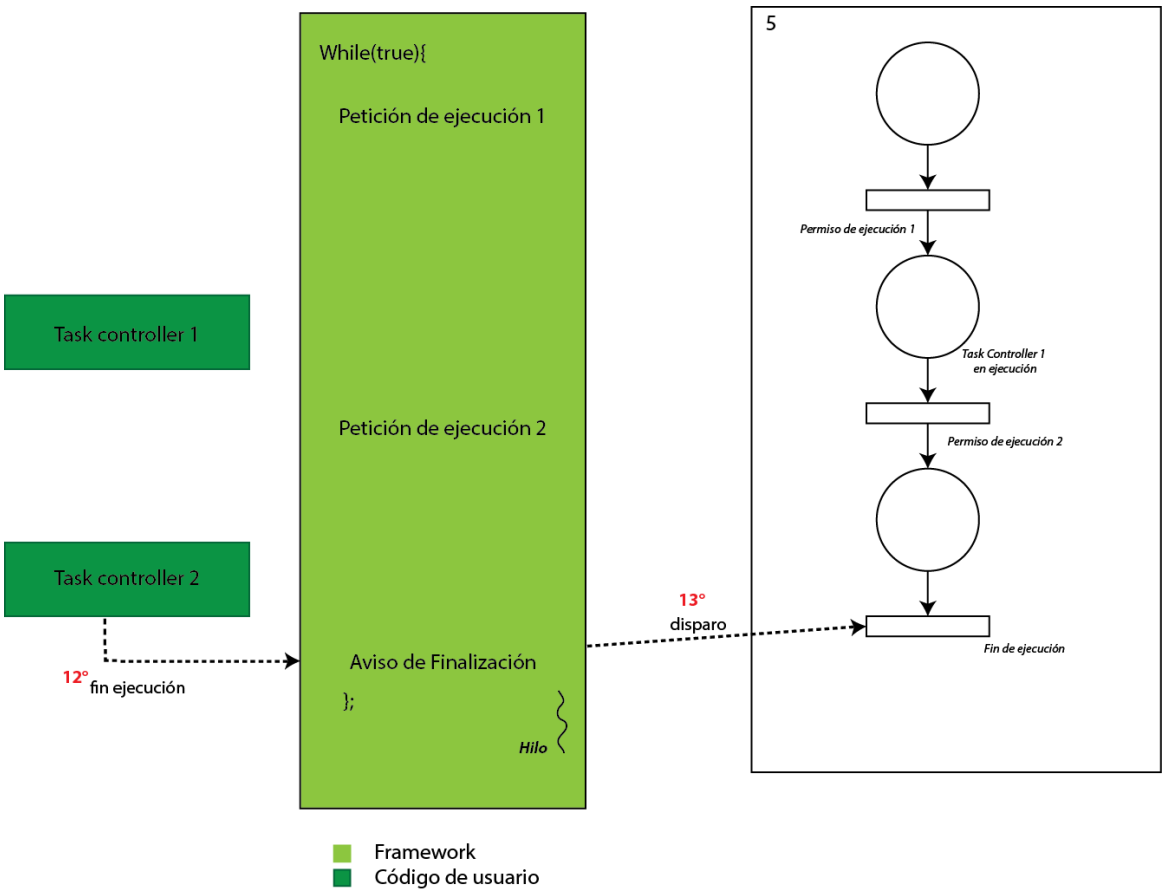


Figura 10.14: Pasos de la Ejecución de un Complex Sequential Task Controller con Dos Acciones Task

10.8. Manejo de Guardas: Guard Providers

Las guardas permiten relacionar la RdP con el estado del medio, y representar condiciones de ejecución síncronas propias del mismo sistema (ver secciones 10.3.1 y 3.3.2).

El monitor de Petri ofrece una interfaz para establecer el valor de una guarda. De esta manera se permite al usuario indicar directamente dicho valor. Sin embargo, el uso de esta alternativa trae aparejada una pérdida de la inversión de control debido a que se modifica el estado de la red de manera directa y en cualquier instante, permitiendo al usuario tomar decisiones sobre el control del flujo de ejecución.

Para lidiar con este problema se propone el concepto de Guard Provider. Un Guard Provider es un método asociado a una guarda que retorna una variable de tipo boolean. Este método es llamado automáticamente por el framework luego de ejecutar un controlador de acción que requiera modificar dicha guarda. Para lograr la llamada automática a un Guard Provider se utiliza reflection (ver sección 4.4.1). El método Guard Provider retorna el valor que debe tomar la guarda asociada.

El concepto de Guard Provider permite limitar el acceso a las guardas. En consecuencia, la modificación de una guarda se realiza sólo tras la ejecución de las acciones que deben modificar en la RdP el estado representado por dicha guarda. El usuario tiene la capacidad de definir la lógica que dará el valor a la guarda, pero el monitor conserva la decisión de ejecución del Guard Provider, ya que la misma esta asociada a la ejecución de los controladores de acciones (ver sección 10.6)

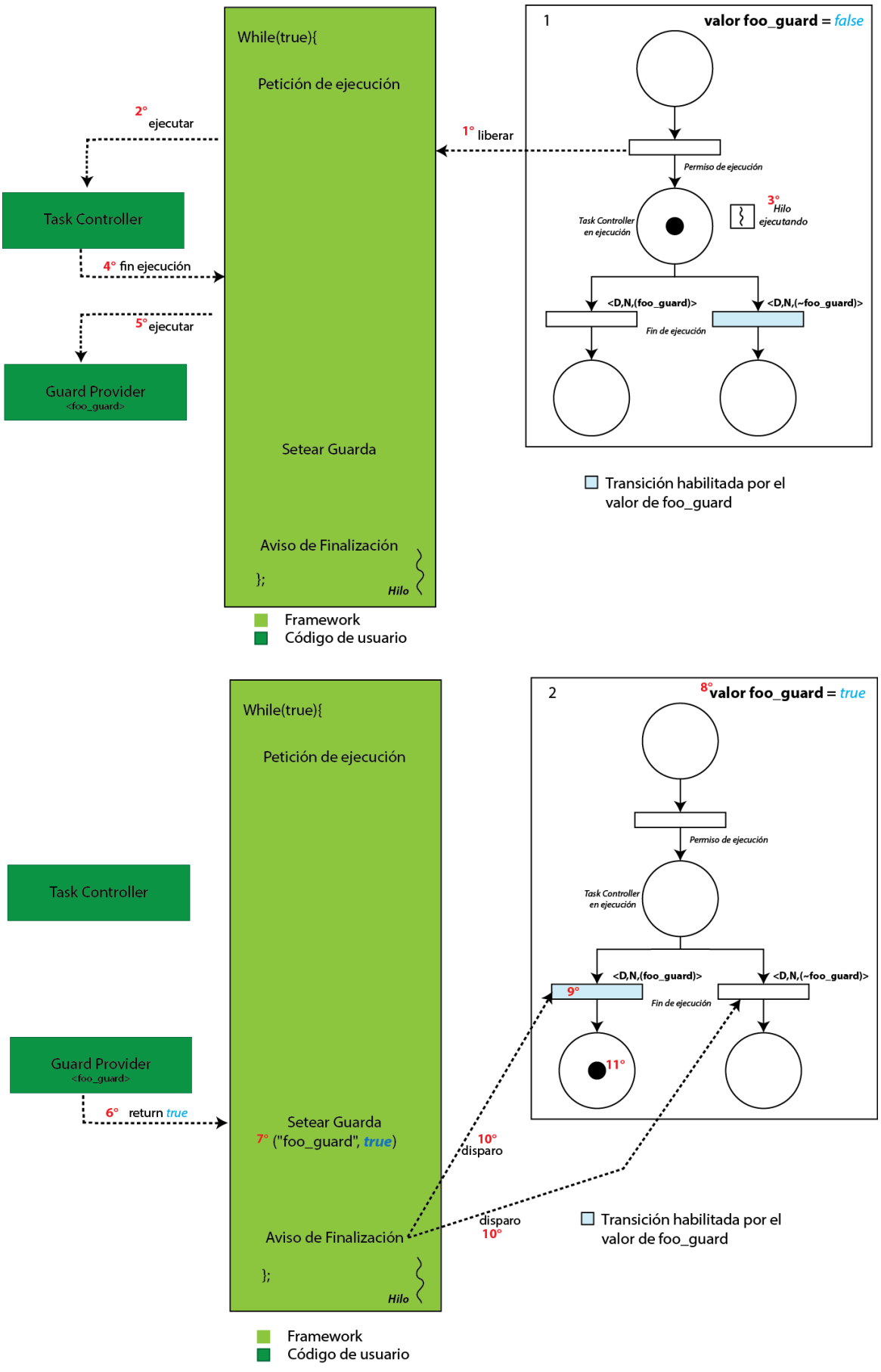


Figura 10.15: Pasos de la Ejecución de un Guard Provider asociado a un Task Controller

10.9. Relación entre Eventos Lógicos, Eventos de Acción y Controladores de Acción

En la sección 10.3 se explica que las acciones de software, embebidas en controladores de acción, son desencadenadas por eventos de acción. A su vez, en la sección 10.6 se definen dos tipos de controladores de acción, cuyo modo de ejecución se explica en las secciones 10.6.1 y 10.6.2.

La ejecución de ambos tipos de controladores tiene tres etapas marcadas:

- Petición de permiso de ejecución al monitor.
- Ejecución de controlador de acción.
- Aviso de finalización de ejecución al monitor.

Si incorporamos el concepto de Guard Provider definido en la sección 10.8, se adiciona una etapa a la ejecución, resultando en:

- Petición de permiso de ejecución al monitor.
- Ejecución de controlador de acción.
- Ejecución del método Guard Provider y evaluación de la guarda.
- Aviso de finalización de ejecución al monitor.

A partir del análisis de las etapas de ejecución del controlador de acción, emergen nuevos requerimientos, relacionados con el requerimiento 2 de la sección 8.3:

- Un evento de acción debe definir todos los eventos lógicos necesarios para la sincronización de la ejecución de un controlador de acción:
  - Permiso de ejecución del controlador de acción (disparo de transición).
  - Cambio de estado del sistema (evaluación de guardas).
  - Aviso de finalización de ejecución del controlador de acción (disparo de transición).
- El framework debe ofrecer interfaces para la suscripción de controladores de acción a eventos de acción.

En consecuencia, se define a un evento de acción como el conjunto de tres eventos lógicos, claves para definir la sincronización de la ejecución de un controlador de acción y sus influencias sobre el estado de la RdP:

Permiso de ejecución: Consiste en un evento lógico de disparo de transición al monitor de manera bloqueante (perenne).

Callback de guardas: Consiste en un evento lógico de evaluación de guarda. El sistema de ejecución obtiene el valor a establecer en la guarda a partir de la ejecución automática de un método de tipo Guard Provider. Este método está asociado a la guarda y al controlador de acción desencadenado por el evento de acción.

Callback de aviso de finalización: Consiste en un evento lógico de disparo de transición al monitor de manera no bloqueante (no perenne). Se trata de una devolución de recursos al monitor, no es una petición de sincronización.

> **Nota:** Un evento de acción debe contener un permiso de ejecución obligatoriamente, ya que la petición de ejecución es el principio de la inversión de control del framework. Sin embargo los callbacks de guardas y de aviso de finalización son opcionales y están sujetos a las características del modelo para el controlador de acción correspondiente.

10.9.1. Tópicos

Un tópico es una representación de un evento de acción. Los controladores de acciones se suscriben a tópicos. Un tópico esta compuesto por:

- Un nombre único: identifica al tópico y se utiliza al momento de realizar la suscripción al mismo.

- Una lista ordenada de nombres de transición. Constituye el permiso de ejecución de cada controlador de acción suscripto al tópico. En el caso de Task Controllers simples y de Happening Controllers, esta lista contiene un solo nombre de transición mientras que en el caso de ComplexSequentialTask Controllers contiene uno por cada sub tarea.
- Una lista ordenada de conjuntos de nombres guardas. Cada conjunto dentro de la lista ordenada constituye el callback de guardas de cada controlador de acción suscripto al tópico. En un ComplexSequentialTask Controller las guardas se evalúan al finalizar cada una de las acciones individuales que componen la tarea compleja. En el caso de Task Controllers simples y de Happening Controllers, la lista ordenada contiene un único conjunto de nombres de guardas mientras que en el caso de ComplexSequentialTask Controllers contiene un conjunto por cada sub tarea.
- Un conjunto de nombres de transiciones que constituye el callback de aviso de finalización de ejecución. Contiene los nombres de todas las transiciones que se disparan de manera no bloqueante al finalizar la ejecución del controlador de acción. En un ComplexSequentialTask Controller el callback de transiciones se dispara luego de finalizar la última sub tarea.

Capítulo 11

Implementación de Baboon Framework

11.1. Introducción

En este capítulo se describe la implementación de los componentes del framework y se detallan sus interfaces de programación. Se detalla la implementación de un controlador de acción y el proceso de ejecución de los mismos. A su vez, se detalla la implementación de los tópicos.

La implementación se realizó de acuerdo a los aspectos de diseño expuestos en el capítulo 10.

11.2. Detalles de Implementación de Baboon Framework

Las interfaces expuestas por Baboon Framework permiten al usuario acceder a la mayoría de las funcionalidades ofrecidas sin necesidad de conocer su estructura interna, como un framework de caja negra (ver sección 5.2.5). Sin embargo, en determinados casos de uso (por ejemplo para añadir políticas de disparo de transiciones) se requiere de la extensión de clases del framework.

Para la utilización de Baboon Framework el usuario tiene la responsabilidad de definir los siguientes componentes, previamente expuestos en el capítulo 10:

- Controladores de acciones.
- Tópicos.
- Modelo de la lógica del sistema en RdP.

Por su lado, el framework se encarga de:

- Proveer interfaces para interconectar los componentes definidos por el usuario.
- Controlar el flujo de ejecución del programa.

11.3. Implementación de un Controlador De Acción

La implementación de un controlador de acción queda definida por dos elementos:

Acción a realizar: Método con una anotación de Java identificando el tipo de controlador de acción. Dicha anotacion puede ser de dos tipos:

- *@TaskController*
- *@HappeningController*

Ejecutante de la Acción: Es la instancia del objeto que ejecuta el método. Este objeto pertenece a una clase que contiene la declaración del método.

11.4. Componentes del Framework

En la Figura 11.1 se muestran los módulos que componen el framework implementado:

- **BaboonConfig:** Este componente se encarga de manejar la suscripción de controladores de acciones a los tópicos (eventos de acción). Para ello provee interfaces para:
  - Recibir el archivo de tópicos definido por el usuario *(addTopics)*.
  - Suscribir los controladores de acciones a tópicos específicos *(subscribeToTopic, createNewComplexTask, appendTaskToComplexTask)*.
  - Consultar las suscripciones existentes *(getTaskControllerSubscription, getHappeningControllerSubscripton)*.
- **PetriCore:** Este componente es un wrapper sobre el monitor de Redes de Petri descripto en el Capítulo 9, y provee las interfaces del monitor al resto del framework.
- **BaboonApplication:** Es una interfaz Java que define dos métodos. Esta interfaz debe ser implementada por una clase de usuario, que será utilizada para inicializar el sistema. Los métodos son:
  - ***declare():*** En este método el usuario debe:
    1. Proveer un archivo JSON con la definición de los tópicos.
    2. Proveer un archivo PNML con la definición del modelo de RdP.

3. Declarar e inicializar los objetos del sistema. Estos objetos (desarrollados por el usuario) contienen la implementación de los controladores de acción.

- ***subscribe():*** En este método el usuario define todas las suscripciones de controladores de acción a tópicos.

- **DummyThread:** Objeto que implementa la interfaz Callable y, en consecuencia, es ejecutable por un hilo. Encapsula el controlador de acción con los eventos lógicos necesarios para su sincronización. Se lo denomina "dummy" (tonto) porque no posee información propia, simplemente envía los eventos de sincronización de acuerdo a los contenidos del tópico y ejecuta las acciones embebidas en el TaskController.

- **DummiesExecutor:** Pool de hilos encargado de ejecutar los objetos DummyThread.

- **HappeningControllerJoinpointReporter:** Aspect donde se definen los puntos de unión del sistema y los advices que se aplican en dichos puntos. Es observable por aquellos objetos que implementen la interfaz JoinPointObserver. Los puntos de union se alcanzan:

  - En el momento previo a iniciar la ejecución de un método anotado con *@HappeningController*.
  - En el momento posterior a finalizar la ejecución de un método anotado con *@HappeningController*.

  Durante la ejecución del advice se realiza una actualización sobre los observadores suscriptos al objeto, brindando el estado del joinpoint alcanzado. El estado del joinpoint permite identificar a un HappeningController en particular a partir de la instancia de objeto que hizo el llamado a ejecución, y de la firma del método que se llamó a ejecutar (ver Sección 11.3).

- **HappeningControllerSyncronizer:** Objeto que implementa la interfaz JoinPointObserver. Se suscribe como observador del objeto HappeningControllerJoinpoint. Cuando se alcanza un joinpoint, intercambia con el monitor los eventos lógicos necesarios para la sincronización del HappeningController. Obtiene los eventos lógicos utilizando las interfaces de BaboonConfig para consultar las suscripciones a tópicos.

- **Main:** Componente que contiene el método principal del programa. Al existir inversión de control, el método main forma parte del framework y no debe ser implementado por el usuario.

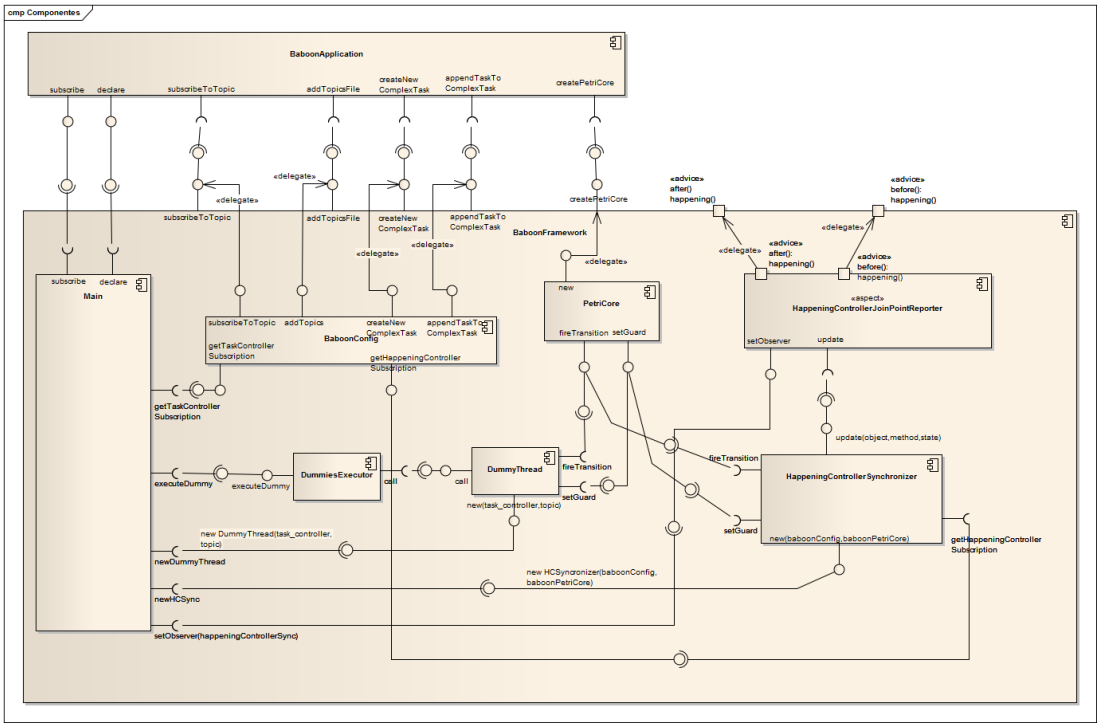


Figura 11.1: Diagrama de Componentes de la implementación de BaboonFramework

11.5. Ejecución de Baboon Framework

- **Main:** La ejecución de este método consta de los siguientes pasos (ver Figura 11.2):

1. Obtiene la clase de usuario que implementa la interfaz BaboonApplication (utilizando Reflection) e instancia un nuevo objeto de dicha clase.
2. Ejecuta los métodos declare() y subscribe() (en ese orden) del objeto mencionado en el punto anterior.
3. Crea el Objeto HappeningControllerSyncronizer y lo suscribe como observer de HappeningControllerJoinPoint.
4. Utiliza las interfaces de BaboonConfig para obtener las suscripciones a tópicos de los Task Controllers.
5. Encapsula los TaskControllers en DummyThreads y los envía al objeto DummiesExecutor para su ejecución.

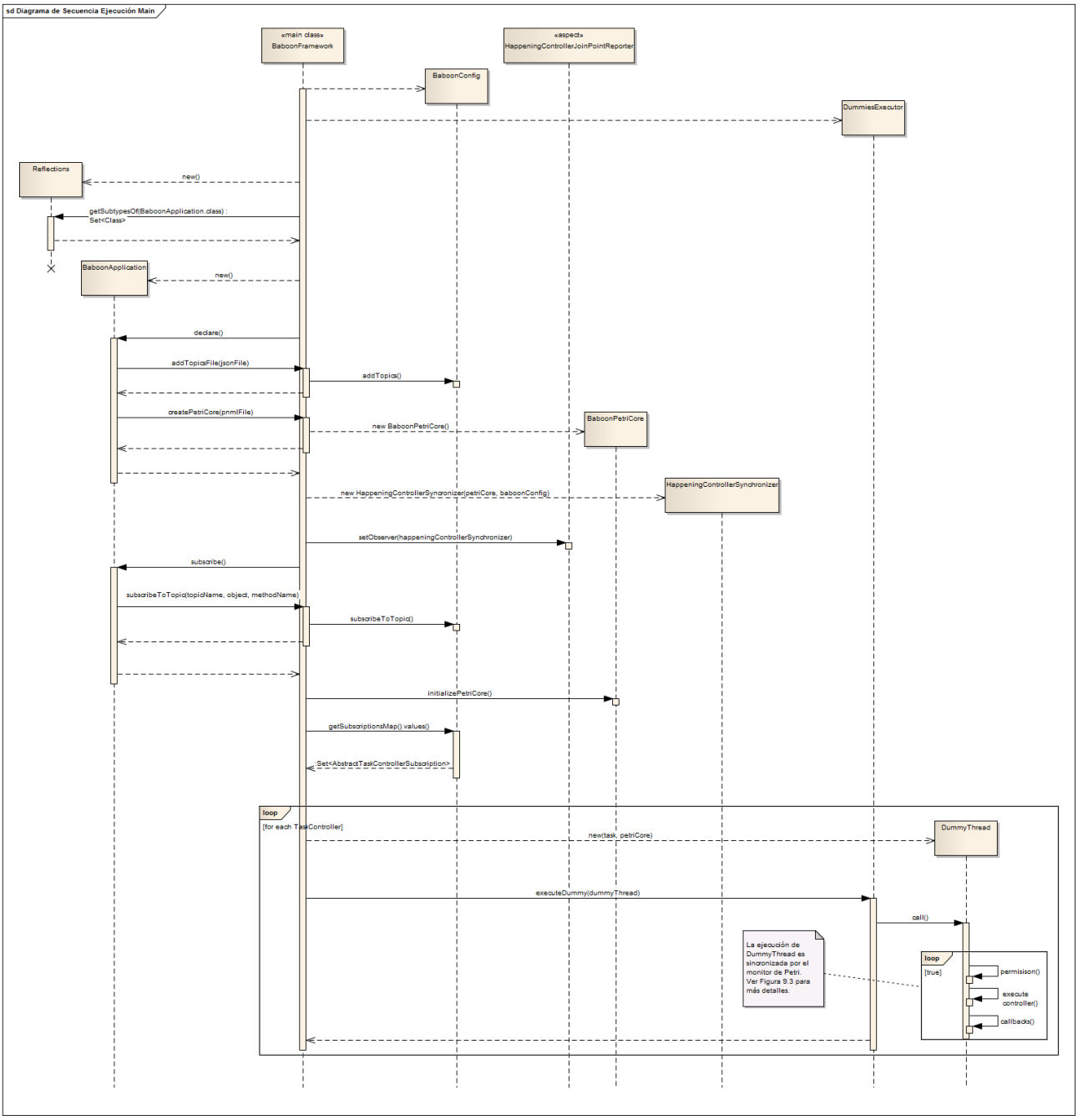


Figura 11.2: Diagrama de Secuencia de la Implementación del Método Principal

- **TaskController:** La ejecución del TaskController implementada consta de los siguientes pasos (ver Figura 11.3):
  1. El pool de hilos DummiesExecutor llama a ejecutar el método *call()* del DummyThread.
  2. El método *call()* del DummyThread inicia un bucle infinito

3. Utilizando el tópico de la suscripción se obtiene la transición que conforma el permiso de ejecución.
4. El DummyThread realiza un disparo perenne de la transición de permiso de ejecución.
5. Cuando el hilo que ejecuta al DummyThread es liberado por el monitor, llama a ejecutar el método del TaskController.
6. El código del TaskController emite un evento físico de salida (Evento Task).
7. Una vez finalizada la ejecución, se obtienen los nombres de las guardas asociadas al controlador a través del tópico de la suscripción.
8. Se ejecutan los métodos GuardProvider que corresponden a las guardas asociadas.
9. El objeto DummyThread establece en el monitor de Petri, para cada guarda asociada, el valor correspondiente que retornan los métodos GuardProvider.
10. Se obtienen del tópico los nombres de las transiciones de aviso de finalización de ejecución.
11. El objeto DummyThread realiza disparos no perennes de las transiciones de aviso de finalización.
12. Se repite el bucle infinito del método *call()* del DummyThread.

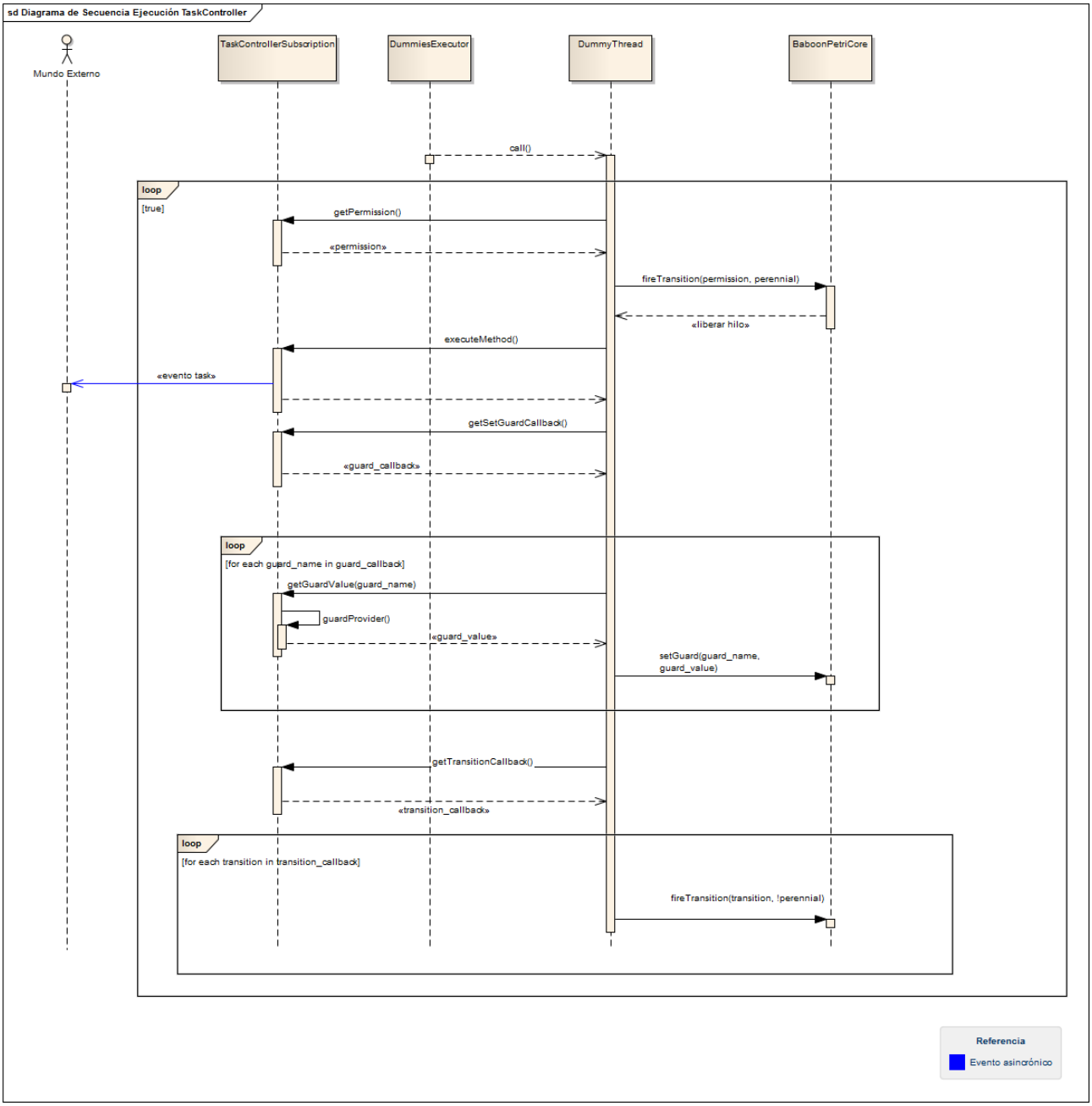


Figura 11.3: Diagrama de Secuencia de la Ejecución Implementada de un TaskController

- **HappeningController:** La ejecución implementada de un HappeningController es de la siguiente forma (ver Figura 11.4):

  1. El código de usuario recibe un evento asincrónico del Mundo Exterior (Evento Happening).
  2. El código de usuario realiza en un hilo el llamado a ejecución del HappeningController encargado de manejar el Evento Happening.
  3. En este punto de la ejecución se alcanza un joinpoint, por lo tanto antes de ejecutar el HappeningController se ejecuta el advice *before()* del objeto HappeningControllerJoinPoint.
  4. El advice *before()* realiza un update del estado del joinpoint al objeto HappeningControllerSincronizer. Dicho estado se compone del nombre del método anotado con *@HappeningController* que se llamó a ejecución, de la instancia del objeto que invocó dicho método y de un enum que indica que el joinpoint es previo a la ejecución del controlador.
  5. El HappeningControllerSincronizer obtiene de BaboonConfig la suscripción al tópico del HappeningController, a partir de los datos obtenidos del estado del joinpoint.
  6. Utilizando el tópico de la suscripción se obtiene la transición que conforma el permiso de ejecución.
  7. El HappeningControllerSincronizer realiza un disparo perenne de la transición de permiso de ejecución.
  8. Cuando el hilo de ejecución es liberado por el monitor, se ejecuta el código del HappeningController, donde se maneja el evento recibido.
  9. Al finalizar la ejecución del HappeningController se alcanza un joinpoint, y se ejecuta el advice *after()* del objeto HappeningControllerJoinPoint.
  10. El advice *after()* realiza un update del estado del joinpoint al objeto HappeningControllerSincronizer. Este estado es similar al descripto en el advice *before()*, con la diferencia de que en este caso el enum indica que el joinpoint es posterior a la ejecución del controlador.
  11. El HappeningControllerSincronizer obtiene de BaboonConfig la suscripción al tópico del HappeningController, a partir de los datos obtenidos del estado del joinpoint.
  12. Se obtienen los nombres de las guardas asociadas al controlador a través del tópico de la suscripción.
  13. Se ejecutan los métodos GuardProvider que corresponden a las guardas asociadas.
  14. El objeto HappeningControllerSincronizer establece en el monitor de Petri, para cada guarda asociada, el valor correspondiente que retornan los métodos GuardProvider.
  15. Se obtienen del tópico los nombres de las transiciones de aviso de finalización de ejecución.
  16. El objeto HappeningControllerSincronizer realiza disparos no perennes de las transiciones de aviso de finalización.

> **Nota:** Los advices de AspectJ implementados se "tejen" al código de usuario en tiempo de compilación (ver Sección 4.3.2). Dichos advices son ejecutados por el mismo hilo que el HappeningController que produce el joinpoint. De esta forma, el hilo que realiza los disparos de transición desde el objeto HappeningControllerSincronizer es el mismo que ejecuta el controlador de acción, permitiendo su sincronización.

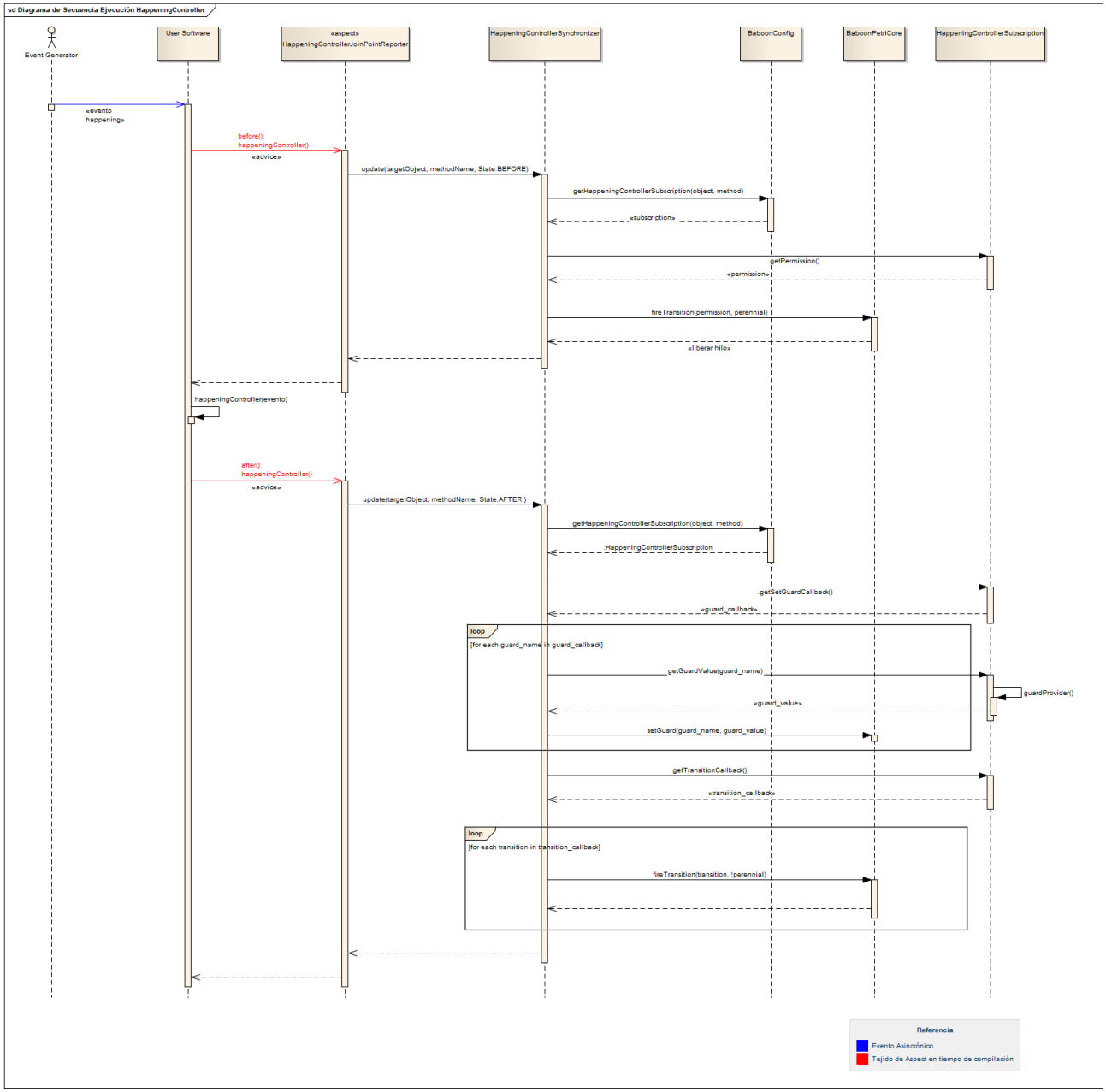


Figura 11.4: Diagrama de Secuencia de la Ejecución Implementada de un HappeningController

11.6. Implementación de Tópicos

Los tópicos, cuyo concepto se explica en la sección 10.9.1, son definidos por el usuario en un archivo JSON con el siguiente formato:

```
[

  {
    "name":"custom_name_1",
    "permission":["my_p", "my_p2", "my_p3"],
    "setGuardCallback": [["g1","g3"],["g1","g2"],["g2","g3"]]
    "fireCallback":["fc_1","fc_2"]
  },

  {
    "name":"test",
    "permission":["my_permission", "my_permission2", "perm3", "p4"],
    "setGuardCallback": [["g1","guard2","g3"],["guardaCuatro"],[],["g5","g_6"]]
  },

  {
    "name":"test",
    "permission":["my_transition"],
  },

  {
    "name":"topicN",
    "permission":["transition0"],
    "fireCallback":["t1","t2","tx"]
  }
]
```

El path a este archivo es incluido en el software de usuario al momento de inicializar Baboonframework. Un parser interno del framework se encarga de procesar este archivo JSON y asociar el tópico a los eventos lógicos.

Una vez configurado y cargado en el sistema el archivo de tópicos, el usuario puede usar el valor de “name” del tópico como identificador para suscribir los controladores de acciones.

Parte IV

Conclusiones

Capítulo 12

Conclusión

12.1. Conclusión

En este trabajo se realizó el diseño e implementación de *Baboon Framework*, un framework para el desarrollo de sistemas reactivos. Como resultado de este desarrollo se logró la centralización de la gestión de los recursos, la concurrencia y la sincronización de hilos y, además, se obtuvo la conducción del flujo de ejecución utilizando un modelo de Red de Petri que implementa la lógica del sistema. A su vez, se desarrolló un mecanismo de gestión de prioridad de ejecución de los hilos por medio de políticas configurables por el usuario. En consecuencia, se cumplió con el objetivo principal del proyecto, descripto en la sección 2.1.

El proceso de diseño e implementación también estuvo guiado por los objetivos secundarios planteados en la sección 2.2. A lo largo de los siguientes párrafos se explica cómo se cumplieron estos objetivos.

La centralización de gestión de recursos, manejo de concurrencia y sincronización de hilos mencionada previamente se obtuvo mediante:

- La transformación del modelo de RdP en código interpretado.
- El desarrollo de un mecanismo de suscripción a disparos de transiciones.
- La implementación de colas de espera y suspensión de hilos.

Estas características fueron implementadas como parte de Java Petri Concurrency Monitor (JPCM), un monitor de concurrencia que ejecuta Redes de Petri haciendo uso de la ecuación generalizada desarrollada en [DIOM16].

Se obtuvo una arquitectura de framework que gestiona los eventos y mecanismos de comunicación necesarios para desacoplar el modelo de RdP, el código de usuario y el entorno. Como resultado, se simplifica el diseño del software de usuario, el que queda definido por:

- El modelo de RdP
- El conjunto de acciones: Son porciones de código con una responsabilidad concreta y simple. Intercambian eventos con el entorno.
- El conjunto de eventos de acción: Contienen las reglas de traducción entre los eventos del entorno y eventos comprensibles para el modelo de RdP.
- El conjunto de suscripciones de acciones a eventos de acción

Se logró un diseño de framework no restrictivo sobre las herramientas que ofrece el lenguaje de programación. En consecuencia, el usuario dispone de todas las características de la programación orientada a objetos para el diseño de su sistema.

Se implementó la inversión de control del framework mediante la utilización de prácticas de *Reflection* y *Aspect Oriented Programming*. Como resultado, el flujo de control del programa es responsabilidad del framework y el código de usuario se centra en las funcionalidades concretas del sistema a implementar.

El código del framework se encuentra disponible de forma pública en repositorios en la red (ver sección 14.3) bajo licencia Apache 2.0. En los repositorios mencionados se encuentra también la documentación en formato *Javadoc* y los casos de test automatizados para las funcionalidades implementadas.

El desarrollo del framework se realizó siguiendo los requerimientos planteados en la sección 8.3. A continuación se presenta un listado de los resultados obtenidos:

- El framework interactúa con JPCM, el monitor de Redes de Petri desarrollado en este proyecto.
- El control del flujo de ejecución es delegado al monitor de RdP mediante la utilización de prácticas de *Reflection* y *Aspect Oriented Programming*.
- Se requiere aprender menos de diez conceptos para desarrollar un sistema utilizando Baboon Framework (task controller, happening controller, complex sequential task controller, interfaz declare, interfaz subscribe, transition policies, topic, guard provider, RdP).
- Se desarrollaron diversos casos de uso, disponibles de forma pública en el repositorio de ejemplos (https://github.com/juanjoarce7456/baboon_examples). En estos ejemplos se muestra la forma de uso de todas las interfaces de usuario del framework.
- El framework utiliza las interfaces que ofrece JPCM para interactuar con el modelo de RdP.
- Actualmente el framework no es compatible con otros monitores desarrollados en el Laboratorio de Arquitectura de Computadoras, pero su arquitectura fue diseñada para adaptarse a un cambio de monitor de forma simple.

- El reporte de cobertura de los casos de prueba arroja un resultado del 82,5 % de líneas de código cubiertas. Este porcentaje supera al requerido.
- El framework tiene al menos un test automatizado por cada funcionalidad implementada.
- Se realizó documentación de código utilizando el formato estándar *Javadoc*.
- Todos los objetos del framework tienen documentación sobre sus métodos, explicando la forma de uso y su función.
- La documentación no brinda detalles de implementación.
- La documentación se encuentra disponible de forma pública. Ver la sección 15.1

A modo de corolario se menciona que la utilización de Baboon Framework en conjunto con el proceso de diseño de sistemas reactivos expuesto en [BL17] permite el diseño y desarrollo de sistemas reactivos confiables, mantenibles y portables a múltiples plataformas.

Capítulo 13

Trabajo Futuro

13.1. Trabajo Futuro

Durante el desarrollo del framework y su posterior utilización, surgieron nuevos aspectos y mejoras a desarrollar en versiones futuras:

- Performance: el uso de técnicas de reflection y de programación orientada a aspectos brinda la posibilidad de realizar la inversión de control y de ofrecer una interfaz de usuario amigable para el programador de sistemas reactivos, pero sacrifica performance. Es de interés investigar la existencia de otras alternativas que permitan la implementación de la inversión de control de manera más performante.
- Utilización para desarrollo de sistemas distribuidos: analizar la posibilidad de centralizar la ejecución de la RdP y exponerla como un servicio para la coordinación de ejecución de sistemas remotos.
- Agregar soporte para otros dialectos de PNML: existen otros software de edición de RdP más potentes que TINA, cada uno con su dialecto de PNML. JPCM está preparado para agregar soporte para nuevos dialectos de forma simple.
- Soporte para ejecución de múltiples RdP: Se puede paralelizar la toma de decisiones mediante el uso de RdP jerárquicas o simplemente mediante el uso de múltiples RdP donde cada una modela una parte del sistema.
- Soporte para ejecución de otros tipos de modelo: en principio, dentro del monitor JPCM se podría ejecutar múltiples tipos de modelos (máquinas de estado finitas, máquinas de Turing, etc). Esta capacidad amplía la gama de usuarios interesados en la utilización de *Baboon Framework*.
- Embeber el manejo e intercambio de datos entre procesos dentro del framework: Actualmente, el framework controla el flujo de control de las instrucciones del programa, pero el manejo de los datos es responsabilidad total del usuario. En la versión actual del framework, el manejo de la dinámica de los datos agrega una complejidad innecesaria al concepto de acción de software, ya que además de la acción concreta debe programarse el manejo del flujo de datos. La implementación de nuevas interfaces para el intercambio de datos entre acciones de software facilitaría aún más la creación de sistemas reactivos utilizando *Baboon Framework*.
- Agregar una interfaz para la liberación de recursos en tiempo de finalización del programa. Esta interfaz debe formar parte de `BaboonApplication`. Así, el usuario deberá implementar esta funcionalidad en un método que será ejecutado por *Baboon Framework*.

Parte V

Anexos

Capítulo 14

Ejemplo de Uso de Baboon Framework

14.1. Introducción

A lo largo de este capítulo se analiza un ejemplo de aplicación con el propósito de demostrar las funcionalidades que ofrece Baboon Framework. El mismo consiste en un sistema de clasificación y lavado de botellas.

14.2. Sistema de Clasificación y Lavado de Botellas

A continuación se describen las características principales del sistema en estudio:

- El ingreso de botellas no es controlado por el sistema.
- Las botellas pueden ser depositadas en la máquina en cualquier instante de tiempo de forma asincrónica para el sistema.
- La clasificación de las botellas se realiza en un módulo que diferencia entre *cerveza*, *gaseosa* u *otras*.
- Existe un límite máximo de 20 botellas en la etapa de recepción y clasificación.
- La recepción de botellas tiene máxima prioridad siempre que pueda realizarse.
- Mientras la máquina se encuentra en proceso de clasificación de botellas, no puede iniciar la recepción de una nueva botella.
- Cuando la máquina no puede recibir una botella, la misma queda en espera para ser procesada.
- El proceso completo de lavado para una botella de gaseosa consta de los siguientes pasos:
  - Lavado con agua a presión.
  - Secado.
- El proceso completo de lavado para una botella de cerveza consta de los siguientes pasos:
  - Lavado con agua a presión y detergente.
  - Enjuague.
  - Secado.
- Si se inserta una botella de otro tipo, la misma es expulsada de la máquina sin lavar.
- La máquina sólo puede procesar una botella por vez en cada uno de sus módulos.
- Al finalizar el proceso de lavado las botellas son devueltas por la máquina, utilizando una salida independiente de la entrada.

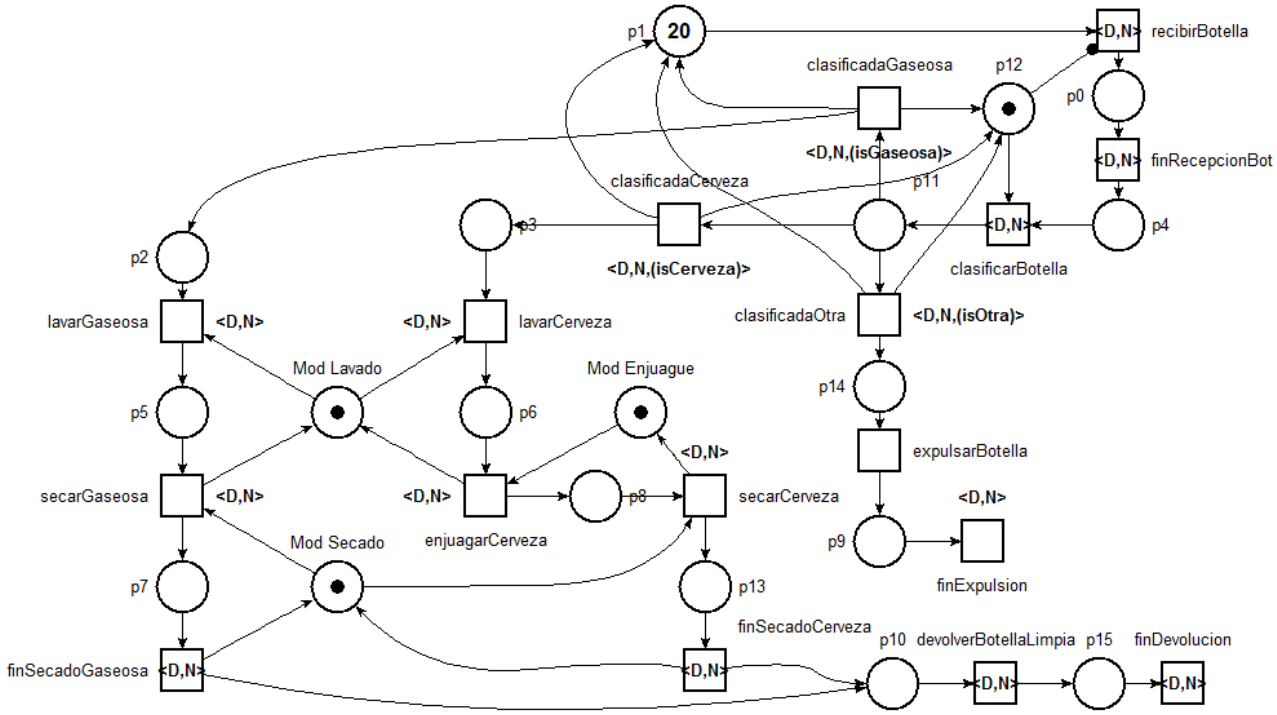


Figura 14.1: Modelo en Red de Petri de la Lógica de un Sistema de Clasificación y Lavado de Botellas

14.2.1. Pasos para el desarrollo del sistema utilizando Baboon Framework

Tras el modelado de la lógica del sistema en la Red de Petri (ver Figura 14.1), el proceso de desarrollo del sistema se descompone en una serie de pasos. De esta manera se comprende fácilmente el proceso para crear un sistema utilizando el framework.

1. **Determinar los objetos para conformar el sistema:**
   Como en cualquier desarrollo orientado a objetos, uno de los pasos principales del diseño es la identificación de los objetos que intervienen en el sistema, para luego obtener las clases.
   En el caso de ejemplo se identifican los siguientes:

   - Botellas insertadas.
   - Máquina lavadora compuesta por:
     - Módulo de clasificación de botellas.
     - Módulo de lavado.
     - Módulo de enjuague.
     - Módulo de secado.
     - Módulo de expulsión de botellas incorrectas.
     - Colas donde se almacenan las botellas durante el proceso.

2. **Determinar las acciones de los objetos del sistema:**
   La determinación de las acciones que realizan los objetos del sistema también está relacionada con el desarrollo orientado a objetos. Estas acciones serán embebidas en los métodos controladores de acciones de las clases desarrolladas.
   En este caso de ejemplo, las acciones son realizadas por la máquina lavadora, y se identifican las siguientes:

   - Recibir botellas.
   - Clasificar botellas.
   - Lavar botellas de gaseosa.
   - Secar botellas de gaseosa.
   - Lavar botellas de cerveza.
   - Enjuagar botellas de cerveza.

- Secar botellas de cerveza.
- Expulsar botellas incorrectas.
- Devolver botellas limpias.

3. **Clasificar Happening Controllers y Task Controllers:**
   Los controladores de acciones pueden procesar eventos físicos de entrada (Eventos Happening) o emitir eventos físicos de salida (Eventos Task). Es importante determinar a cuál grupo pertenece cada acción del sistema, para clasificarla como Happening Controller o Task Controller (ver sección 10.6). Esta clasificación se realiza en el software a través de las anotaciones Java @HappeningController y @TaskController.
   En el caso de ejemplo se distinguen:

   - Happening Controllers:
     - Recibir botellas.
   - Task Controllers:
     - Clasificar botellas.
     - Lavar botellas de gaseosa.
     - Secar botellas de gaseosa.
     - Lavar botellas de cerveza.
     - Enjuagar botellas de cerveza.
     - Secar botellas de cerveza.
     - Expulsar botellas incorrectas.
     - Devolver botellas limpias.

4. **Manejo de guardas:**
   En determinadas ocasiones, es necesario representar estados externos que no están modelados en la red. Para lograrlo, se asocia una variable booleana (que representa el estado externo) a una transición de la RdP. Esta variable actúa como un factor externo de sensibilización de la transición (ver secciones 9.5.4 y 3.3.2). Para alterar el valor de una guarda, debe definirse un método del tipo Guard Provider asociado a dicha guarda, utilizando la anotación Java @GuardProvider en el código del software (ver sección 10.8).

5. **Definir la prioridad de las acciones:**
   En determinados casos de uso deben definirse prioridades específicas para la ejecución de los controladores de acciones. Estas prioridades se manejan a través del monitor de redes de Petri, mediante la utilización de políticas de disparo de transición. El usuario tiene la opción de utilizar las políticas proporcionadas por el framework (transición aleatoria o primera transición en línea) o definir una política a su medida como se explica en la sección 9.5.7.
   De acuerdo a las características descriptas para este caso, la transición "recibirBotella" debe tener máxima prioridad.

6. **Definir la cantidad de Hilos de ejecución:**
   En términos generales se desea optimizar la performance mediante la minimización de la cantidad de hilos de ejecución, sin afectar al paralelismo del programa. En este sentido, el usuario debe identificar los grupos de TaskControllers secuenciales y agruparlos en ComplexSequentialTaskControllers, que solo requieren un único hilo para su ejecución (ver sección 10.7).
   En el ejemplo de estudio se distinguen los siguientes hilos de ejecución, donde se especifica quién está a cargo de la creación e inicialización del hilo (el control de la ejecución lo realiza el monitor en todos los casos):

   - **Happening Controllers *(A cargo del usuario)*:**
     1 Recibir botellas.
   - **Task Controllers *(A cargo del framework)*:**
     2 Clasificar botellas.
     3 Expulsar botellas incorrectas.
     4 Devolver botellas limpias.
   - **Complex Sequential Task Controllers *(A cargo del framework)*:**
     5 Proceso de lavado de botellas de gaseosa.
       - Lavar botellas de gaseosa.

- Secar botellas de gaseosa.

6 Proceso de lavado de botellas de cerveza.

- Lavar botellas de cerveza.
- Enjuagar botellas de cerveza.
- Secar botellas de cerveza.

7. **Determinar la relación entre la Red de Petri, los métodos y las clases:**
   La ejecución de los controladores de acciones está relacionada con el disparo de las transiciónes en la Red de Petri. Por este motivo, es necesario relacionar dichas transiciones con los objetos y métodos correspondientes. Dicha tarea se realiza a través de un archivo JSON, configurable por el usuario, donde se definen los tópicos (ver sección 10.9). Tras crear el archivo, el usuario debe suscribir cada objeto y método que sean parte de un controlador de acción a un tópico. Esta suscripción se realiza en el código del software.

14.2.2. Implementación del sistema utilizando Baboon

En esta sección se analiza una implementación del sistema de lavado y clasificación de botellas. En esta implementación en particular, el mundo físico externo al sistema es representado por una Interfaz Gráfica de Usuario.

En la figura 14.2 se observa el diagrama de clases del sistema. El usuario es el encargado de definir las interfaces y herencias necesarias para implementar las funcionalidades del sistema.

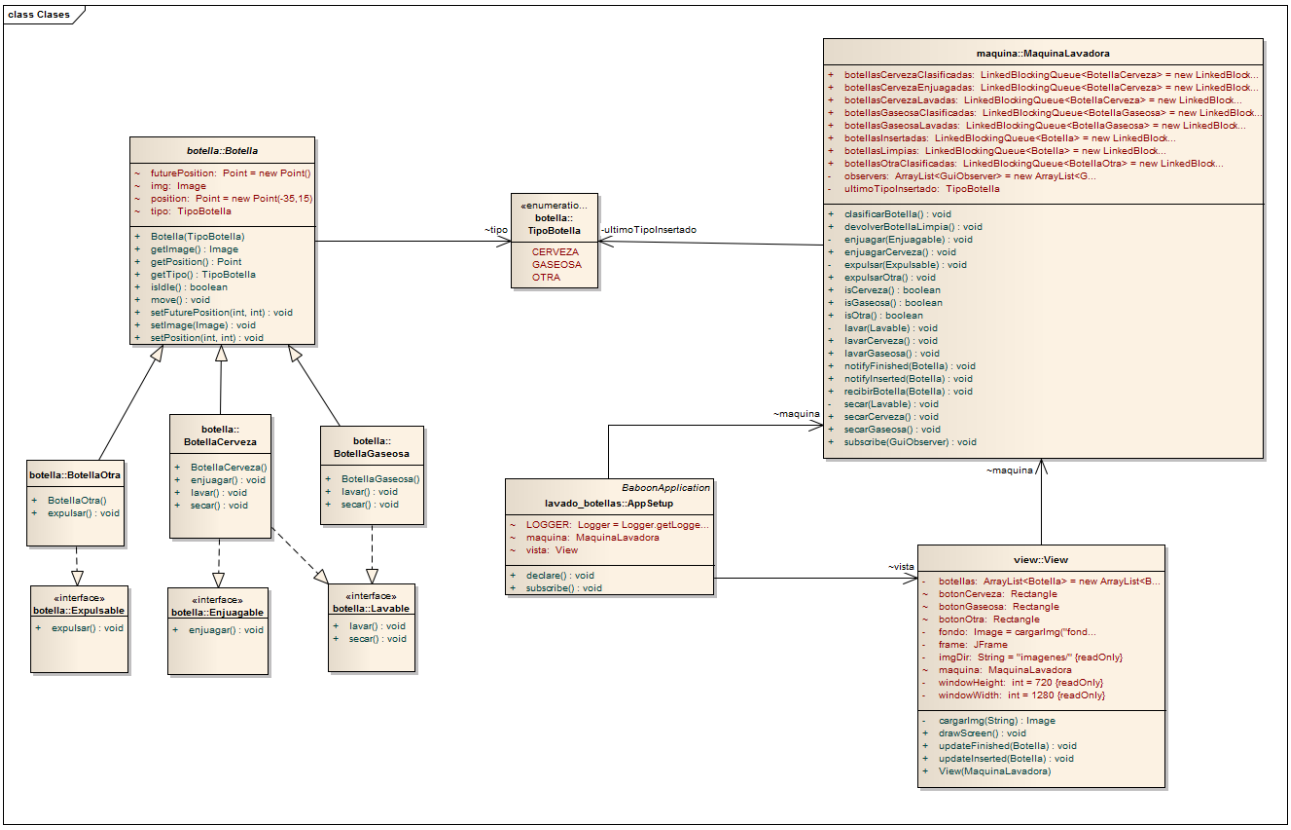


Figura 14.2: Diagrama de Clases de un Sistema de Clasificación y Lavado de Botellas.

Aquellos métodos que forman parte de un controlador de acción deben clasificarse y etiquetarse en el código con *@HappeningController* o *@TaskController*. Por ejemplo:

```
@TaskController
public void lavarGaseosa(){
  BotellaGaseosa botella = botellasGaseosaClasificadas.poll();
  lavar(botella);
  botellasGaseosaLavadas.add(botella);
}
```

```
@HappeningController
public void recibirBotella(Botella botella){
  botellasInsertadas.add(botella);
}
```

El manejo de las guardas se realiza mediante la definición de métodos anotados con *@GuardProvider*. Por ejemplo:

```
@GuardProvider("guardCerveza")
public boolean isCerveza(){
  return ultimoTipoInsertado.equals(TipoBotella.CERVEZA);
}
```

El manejo de prioridades se realiza mediante la implementación de una política de disparo de transiciones personalizada. En la definición de las características del problema se estableció que la transición "recibirBotella" debe tener máxima prioridad. Para ello se creó una clase *InsertBottlesFirstPolicy* que hereda de *TransitionsPolicy* e implementa el método *which()*.

> **Nota:** El usuario no debe instanciar un objeto de la clase *InsertBottlesFirstPolicy*. Al momento de crear el monitor de petri se indica cuál es la clase que implementa la política de disparos, y el framework se encarga luego de generar la instancia que será utilizada. De esta forma el usuario no tiene acceso directo a la RdP desde el código, previniendo modificaciones indeseadas del estado de la red.

```
public class InsertBottlesFirstPolicy extends TransitionsPolicy{

  private final int priorityIndex;

  public InsertBottlesFirstPolicy(PetriNet _petri) {
    super(_petri);
    priorityIndex = this.petri.getTransition("recibirBotella").getIndex();
  }

  @Override
  public int which(boolean[] enabled) {
    if(enabled[priorityIndex]){ //Si recibirBotella está habilitada.
      return priorityIndex; //retornar el indice de recibirBotella.
    }
    else {
      //Sino, disparar la primer transición
      //habilitada que encuentre.
      for(int i = 0; i < enabled.length; i++){
        if(enabled[i]){
          return i;
        }
      }
    }
    return -1;
  }
}
```

En la sección 14.2.1 se determinaron 6 hilos de ejecución. Cada uno de los hilos corresponde a un controlador de acción. A su vez, cada controlador de acción debe suscribirse a un tópico, por lo que es necesario definir en primer lugar dichos tópicos. Los mismos se definen a través de un archivo JSON, de la siguiente forma:

```
[
  {
    "name" : "recepcion_botella",
    "permission": ["recibirBotella"],
    "fireCallback": ["finRecepcionBot"]
  },
  {
    "name" : "clasificacion_botella",
    "permission": ["clasificarBotella"],
    "fireCallback": ["clasificadaGaseosa", "clasificadaCerveza", "clasificadaOtra"],
    "setGuardCallback" : [["guardCerveza","guardGaseosa","guardOtra"]]
  },
  {
    "name" : "lavado_cerveza",
    "permission" : ["lavarCerveza","enjuagarCerveza","secarCerveza"],
    "fireCallback": ["finSecadoCerveza"]
  },
  {
    "name": "lavado_gaseosa",
    "permission" : ["lavarGaseosa","secarGaseosa"],
    "fireCallback" : ["finSecadoGaseosa"]
  },
  {
    "name" : "expulsion_otra",
    "permission": ["expulsarBotella"],
    "fireCallback" : ["finExpulsion"]
  },
  {
    "name" : "devolucion_botella",
    "permission": ["devolverBbotellaLimpia"],
    "fireCallback" : ["finDevolucion"]
  }
]
```

Un tópico está relacionado con los eventos lógicos de disparo de transiciones y de modificación de valor de guardas. Dicha relación se determina a través del nombre de dichos componentes de la RdP, como se muestra en la figura 14.3.

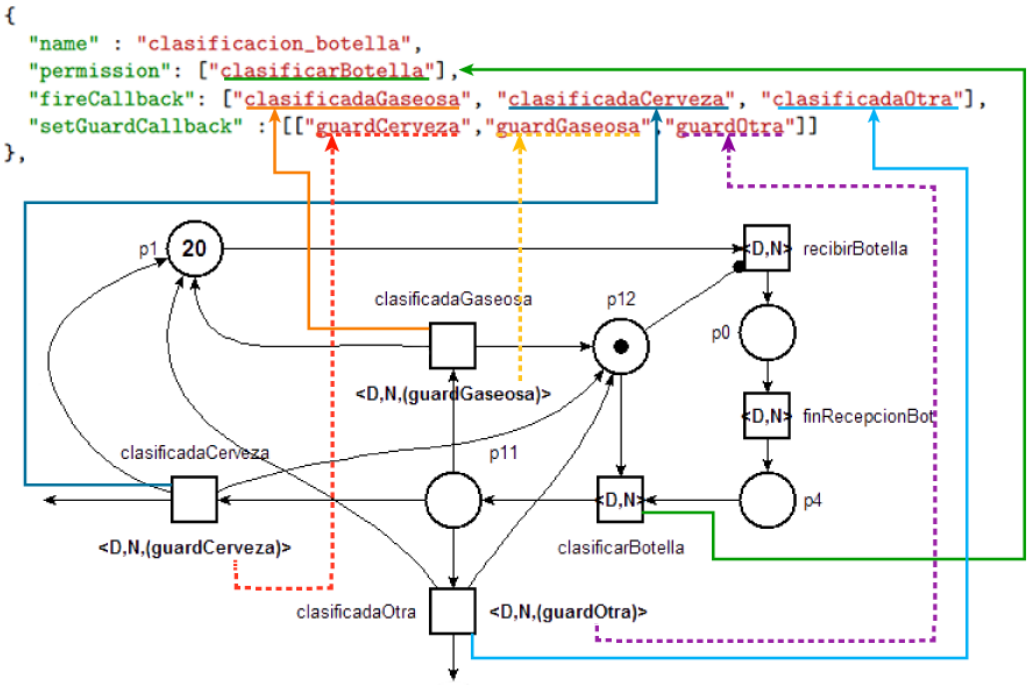


Figura 14.3: Relación entre un tópico y los componentes de la Red de Petri.

Otro aspecto que el usuario debe tener en cuenta, es la creación e inicialización de los hilos que se encargan de ejecutar los HappeningController. Esto es responsabilidad del código de usuario. Por ejemplo, cuando el EventListener que escucha eventos del mouse en la GUI detecta un evento, se crea un nuevo hilo que ejecuta el HappeningController encargado de manejar dicho evento (*recibirBotella()*). A continuación se observa el código descripto:

```
@Override
public void mouseClicked(MouseEvent arg0) {
  new Thread( () -> {
    if(botonCerveza.contains(arg0.getPoint())){
      maquina.recibirBotella(new BotellaCerveza());
    }
    else if(botonGaseosa.contains(arg0.getPoint())){
      maquina.recibirBotella(new BotellaGaseosa());
    }
    else if(botonOtra.contains(arg0.getPoint())){
      maquina.recibirBotella(new BotellaOtra());
    }
  }).start();
}
```

Finalmente, se define la clase *AppSetup*, que implementa la interfaz *BaboonApplication* (ver sección 11.4), donde se realizan las siguientes actividades:

- Inicializar el monitor de RdP con el archivo PNML y la política de disparos.
- Añadir el archivo de tópicos para configurar los eventos de acción en el sistema.
- Declarar e inicializar los objetos del sistema.
- Suscribir los controladores de acción a los tópicos.

A continuación se observa la implementación de la clase AppSetup:

```java
public class AppSetup implements BaboonApplication {
  private static Logger LOGGER = Logger.getLogger(AppSetup.class.getName());
  private MaquinaLavadora maquina;
  private View vista;

  @Override
  public void declare() {
    //Inicialización de los objetos del sistema
    maquina = new MaquinaLavadora();
    vista = new View(maquina);
    try {
      //Inicialización del core de petri.
      //Notar que se requiere el objeto Class de la política.
      BaboonFramework.createPetriCore("pnml/lavadoBotellas_v2.pnml",
          petriNetType.PLACE_TRANSITION, InsertBottlesFirstPolicy.class);
    }
    catch (BadPolicyException e) {
      LOGGER.log(Level.SEVERE, "Error configurando la política de disparos.
                    La aplicación terminará ahora.", e);
      System.exit(1);
    }
    try {
      //Inicialización de los tópicos.
      BaboonFramework.addTopicsFile("topic/topics.json");
    }
    catch (BadTopicsJsonFormat | NoTopicsJsonFileException e) {
      LOGGER.log(Level.SEVERE, "Error inicializando Baboon Framework.
                    La aplicación terminará ahora.", e);
      System.exit(1);
    }

    //Inicialización de thread de GUI
    new Thread( () -> {
      while(true){
        vista.drawScreen();
        try {
          Thread.sleep(25);
        } catch (InterruptedException e) {
           LOGGER.log(Level.INFO, "InterruptedException en thread de GUI", e);
        }
      }
    }).start();
  }
```

```
    @Override
    public void subscribe() {
      try {
        //Suscripción de un HappeningHandler.
        //Notar que se usa un objeto BotellaCerveza como parámetro.
        //El framework lo utiliza para determinar el método correcto a
        //utilizar como controlador de acción
        // En este caso, cualquier objeto que herede de
        //la clase abstracta Botella puede ser utilizado.
        BaboonFramework.subscribeControllerToTopic("recepcion_botella", maquina,
            "recibirBotella", new BotellaCerveza());

        //Suscripción de TaskControllers Simples
        BaboonFramework.subscribeControllerToTopic("clasificacion_botella",
              maquina, "clasificarBotella");
        BaboonFramework.subscribeControllerToTopic("expulsion_otra",
              maquina, "expulsarOtra");
        BaboonFramework.subscribeControllerToTopic("devolucion_botella",
              maquina, "devolverBotellaLimpia");

        //Creación de ComplexSequentialTaskControllers
        BaboonFramework.createNewComplexTaskController("lavarCerveza", "lavado_cerveza");
        BaboonFramework.createNewComplexTaskController("lavarGaseosa", "lavado_gaseosa");

        //Suscripción de TaskControllers a ComplexSequentialTaskControllers
        BaboonFramework.appendControllerToComplexTaskController("lavarCerveza",
              maquina, "lavarCerveza");
        BaboonFramework.appendControllerToComplexTaskController("lavarCerveza",
              maquina, "enjuagarCerveza");
        BaboonFramework.appendControllerToComplexTaskController("lavarCerveza",
              maquina, "secarCerveza");
        BaboonFramework.appendControllerToComplexTaskController("lavarGaseosa",
              maquina, "lavarGaseosa");
        BaboonFramework.appendControllerToComplexTaskController("lavarGaseosa",
              maquina, "secarGaseosa");

      } catch (NotSubscribableException e) {
        LOGGER.log(Level.SEVERE, "Error suscribiendo acciones a Baboon Framework.
          La aplicación terminará ahora.", e);
        System.exit(1);
      }
    }
  }
```

14.3. Configuración del Entorno de Desarrollo con BaboonFramework

La configuración del entorno de desarrollo para BaboonFramework consta de los siguientes pasos:

Eclipse

1. Descargar e instalar Java JDK version >= 8
2. Descargar e instalar Eclipse IDE
3. Instalar M2Eclipse (Maven):
   - En Eclipse, ir a Help ->Install New Software
   - Colocar la dirección del repositorio de m2e en el recuadro “Work with”.
4. Instalar AspectJ:

- En Eclipse, ir a Help ->Install New Software
- Colocar la dirección del repositorio de ajdt en el recuadro “Work with”.

5. Instalar Git Client
6. Importar Proyecto Maven:
   - En Eclipse, ir a File ->Import...->Maven ->Existing Maven Projects. Luego seleccionar la carpeta */path/baboon_examples/LavadoraBotellas*

Intellij

1. En Intellij, ir a File ->Open.... Luego seleccionar el archivo */path/baboon_examples/LavadoraBotellas/pom.xml* y hacer click en el botón “Open as Project”

14.4. Construcción y Ejecución del sistema utilizando Baboon Framework

Maven

Desde una consola, ubicado en el directorio */path/baboon_examples/LavadoraBotellas* correr el comando:

- mvn exec:java

La configuración de Maven escrita en el archivo *pom.xml* especifica la forma de ejecutar el proyecto. Gracias a esto resulta muy simple compilar y ejecutar el proyecto actual.

Eclipse

Para comenzar la ejecución del sistema desde la IDE Eclipse, debe seleccionarse *BaboonFramework* como punto de entrada al programa en las configuraciones:

- En Package Explorer, hacer click derecho sobre el proyecto.
- Seleccionar Run As ->Java Application.
- Seleccionar *BaboonFramework*.

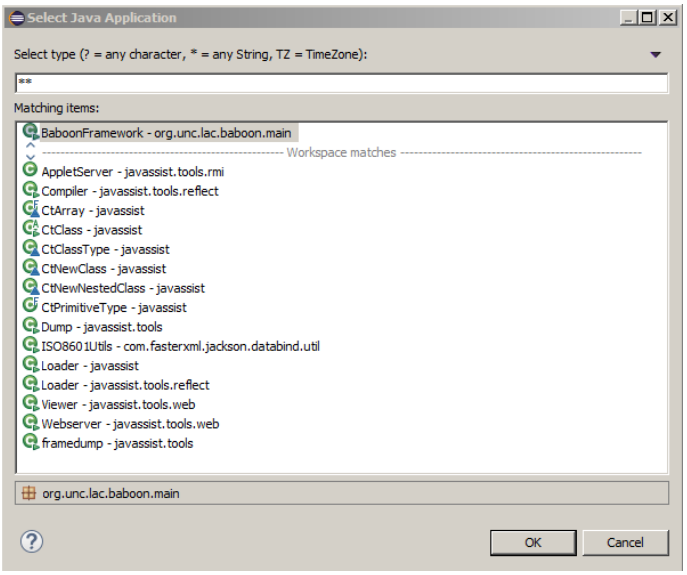


Figura 14.4: Clase Principal de un Sistema Desarrollado con Baboon Framework.

Intellij

Para ejecutar el proyecto desde Intellij, primero se deben realizas las configuraciones necesarias:

- Ir a Run ->Run...->Edit Configurations
- En la ventana Run hacer click sobre el signo '+' ->Maven
- Llenar los campos:
  - Name: *LavadoraBotellas*
  - Working Directory: */path/baboon_examples/LavadoraBotellas*
  - Command Line: *exec:java*

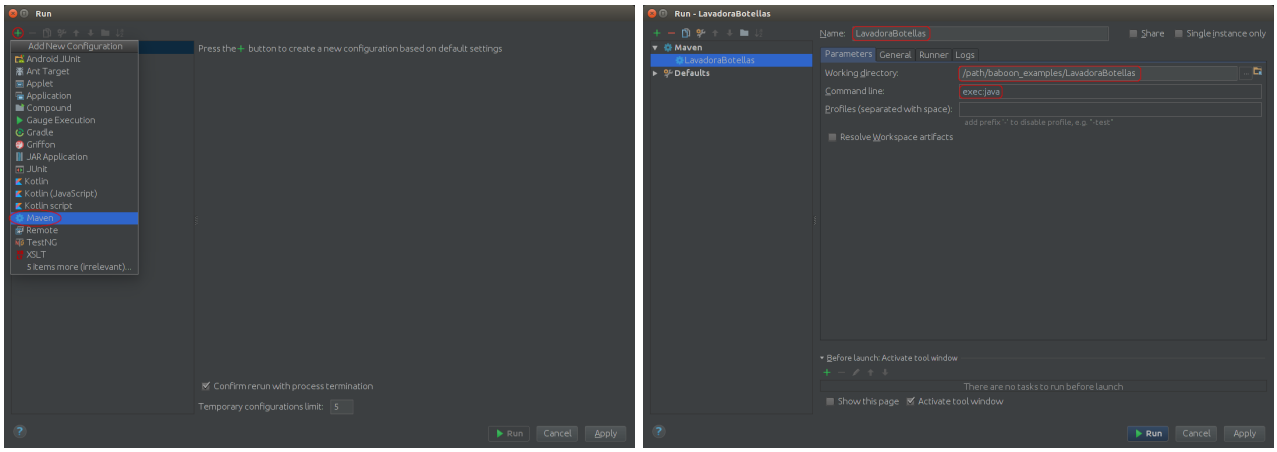


Figura 14.5: Configuración para correr LavadoraBotellas desde IntelliJ

Luego se ejecuta la aplicación con el botón *Run*

Exportar a un Archivo Ejecutable

Para generar un archivo *.jar* ejecutable, se debe correr el siguientes comando:

- mvn package

El archivo resultante es *./target/LavadoraBotellas-0.0.1-SNAPSHOT-jar-with-dependencies.jar* y para ejecutarlo basta con darle doble click, o desde una consola ejecutar:

- java -jar ./target/LavadoraBotellas-0.0.1-SNAPSHOT-jar-with-dependencies.jar

Capítulo 15

Documentación y Casos de Prueba de JPCM y Baboon Framework

15.1. Documentación del Código Fuente

Junto con el código fuente desarrollado se encuentra en formato Javadoc la documentación del mismo. El código mencionado se obtiene de los repositorios ubicados en:

- JPCM: https://github.com/airabinovich/java_petri_concurrency_monitor
- Baboon: https://github.com/juanjoarce7456/baboon

Cada repositorio tiene un botón en la página de inicio que lleva a su documentación online.

15.1.1. Generación de la Documentación

Si se desea construir la documentación, al ser Javadoc un formato estándar, existen intérpretes que lo llevan a una forma legible para desarrolladores de software. Múltiples IDEs incluyen herramientas que realiza esta tarea. Para utilizar esta herramienta deben realizarse los siguientes pasos:

Eclipse

- Dirigirse al menú *Project*
- Seleccionar la opción *Generate Javadoc...*
- Del árbol del proyecto seleccionar el directorio */src*
- Elegir el nivel de visibilidad deseado (*public* para usuarios del framework, *private* para desarrolladores)
- Elegir la ubicación donde se generará la documentación
- Hacer click en *Finish*

IntelliJ

- Dirigirse al menú *Tools*
- Seleccionar la opción *Generate JavaDoc...*
- Sleccionar *Custom Scope* y especificar *Project Production Files*
- Destildar la opción *Include test sources*
- Elegir el nivel de visibilidad deseado (*public* para usuarios del framework, *private* para desarrolladores)
- En el campo *Output Directory* Elegir la ubicación donde se generará la documentación
- Hacer click en *Ok*

15.2. Generación de la documentación de Test

En los repositorios mencionados previamente se encuentra el código fuente de los casos de prueba automatizados. A fin de conocer estos casos, se genera la documentación de los mismos. Esto se realiza siguiendo los pasos descriptos en la sección 15.1 con la salvedad de que se elige la carpeta */test* del árbol del proyecto en Eclipse, y se especifica *Custom Scope* como *Project Test Files* en IntelliJ.

De esta manera se genera una página HTML que contiene la descripción de los casos de test desarrollados. La descripción mencionada explicita *precondiciones*, *acciones* y *resultados esperados* de cada caso de prueba.

Para ejecutar los casos de prueba, debe ejecutarse el directorio */test* utilizando JUnit ó mediante Maven ejecutando el comando *mvn test*.

Bibliografía

[Acc] Página oficial acceleo. www.acceleo.org/.

[Act] Página oficial actifsource. www.actifsource.com/.

[AF15] Altman and Fretes. Desarrollo de un framework para aplicar el paradigma de programación reactiva utilizando redes de petri como procesador de eventos. 2015.

[Apa] Página oficial apache thrift. https://thrift.apache.org/.

[Asp03] *AspectJ In Action*. Manning, 2003.

[BAM13] Birocco Baudino, Arlettaz, and Micolini. Reducción de recursos en un procesador de redes de petri implementado en un ip core. 2013.

[BCC+13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, March 1995.

[BL17] Bentivegna and Ludemann. Estudio e implementación de un caso testigo para el desarrollo de sistemas embebidos, críticos y reactivos. 2017.

[CF14] Caro and Furey. Framework para la generación de código a partir de una red de petri. 2014.

[Dan15] Emiliano Daniele. Modularización del procesador de petri y optimización para sistemas embebidos. 2015.

[Dat] Data flow languages and programming.

[DIOM16] Ing. Maximiliano Eschoyez Ing. Luis Orlando Ventre Ing. Marcelo Ismael Schild Dr. Ing. Orlando Micolini, Geol. Marcelo Cebollada y Verdaguer. Ecuación de estado generalizada para redes de petri no autónomas y con distintos tipos de arcos. Technical report, Laboratorio de Arquitectura de Computadoras (LAC), FCEFyN, Universidad Nacional de Córdoba, 2016.

[FFI+04] Ira R. Forman, Nate Forman, Dr. John Vlissides Ibm, Ira R. Forman, and Nate Forman. Java reflection in action, 2004.

[Fla05] David Flanagan. *Java in a Nutshell*. .º'Reilly Media, Inc.", 2005.

[Gen] Página oficial genexus. https://www.genexus.com/.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gor40] Saul Gorn. Is automatic programming feasible? 1940.

[GP12] Gallia and Pereyra. Implementación de un sistema multicore heterogéneo embebido con procesador de petri sobre fpga. 2012.

[Hyb10] *Discrete, Continuous, and Hybrid Petri Nets*. Springer, 2010.

[IA06] Marian V. Iordache and Panos J. Antsaklis. *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach (Systems & Control: Foundations & Applications)*. Birkhauser, 2006.

[J.05] Arlow J. *UML 2 and the Unified Process*. Pearson, 2 edition, 2005.

[Joh97a] Ralph E Johnson. Components,frameworks, patterns. 1997.

[Joh97b] Ralph E Johnson. Frameworks = (components + patterns). *COMMUNICATIONS OF THE ACM*, 1997.

[Laf03] Robert Lafore. *Data structures and algorithms in Java*. Sams Publishing, 2003.

[Mic15] Orlando Micolini. Arquitectura asimétrica multi core con procesador de petri. 2015.

[NPM12] Nonino, Pisetta, and Micolini. Desarrollo de un ip core con procesamiento de redes de petri temporales para sistemas multicore en fpga. 2012.

[O'R04] Graham O'Regan. Introduction to aspect-oriented programming. *O'Reilly*, 2004.

[Pal03] *Programación Concurrente*. Thomson, 2003.

[Par85] David Lorge Parnas. Software aspects of strategic defense systems. *Commun. ACM*, 28(12):1326–1335, December 1985.

[Pet09] *Petri nets : fundamental models, verification, and applications*. Wiley, 2009.

[Pnm] Pnml reference site. http://www.pnml.org/. Accedido: 07-05-2017.

[Ram74] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[RxJa] Página de documentación de rxjava. http://reactivex.io/RxJava/javadoc/. Accedido: 07-05-2017.

[RxJb] Página de rxjava en github. https://github.com/ReactiveX/RxJava. Accedido: 07-05-2017.

[SH05] Stoerzer and Hanenberg. A classification of pointcut language constructs. *Software-engineering Properties of Languages and Aspect Technologies*, 2005.

[Sis97] *Sistemas Operativos*. Prentice Hall, 1997.

[Sou] Tiago Boldt Sousa. Dataflow programming concept, languages and applications. Technical report, INESC TEC, Faculty of Engineering, University of Porto.

[Spr] Página oficial spring roo. https://projects.spring.io/spring-roo/.

[Tin] Tina editor home page. http://projects.laas.fr/tina/. Accedido: 07-05-2017.

[Wri05] David R. Wright. Finite state machines. *CSC215*, 2005.