Agradecimientos

Deseo agradecer al Dr. Sergio V. Chapa Vergara, director de esta tesis, por todo el apoyo y esfuerzo incondicional, dedicado en el desarrollo de este trabajo.

A Hugo García Monroy, Alfonso Briones García, Silvana Bravo Hernández y al grupo de discusión de Cocoa-Developers, por el intercambio de ideas sobre de desarrollo en OpenStep/Mac OS X, y diseño orientado a objetos. Su ayuda facilitó la implantación de *PetrA*.

A mi esposa Julieta, por haberme soportado en los momentos de incertidumbre.

Y, finalmente, a todo el personal de investigación del Departamento de Aplicación de Microcomputadoras, del Instituto de Ciencias de la Universidad Autónoma de Puebla, y a la Fundación Telmex por haber creido en mí, al apoyarme durante la maestría.

Resumen

Las redes de Petri son una estructura matemática que nos permite modelar sistemas que se definen en términos de causa-evento. En ocasiones, dependiendo del sistema que se modela, se adecúan extensiones y restricciones a las redes de Petri, lo cual a generado una gran variedad de este tipo de redes, como son: redes de Petri basadas en tiempo, coloreadas, y jerárquicas, por mencionar algunas.

Existen una gran variedad de sistemas de software que trabajan con redes de Petri, sin embargo carecen de la capacidad de extensión por parte del usuario, ya que sólo se proporciona o vende el código ejecutable y, por lo general, está orientado al manejo de un sólo tipo de redes de Petri.

Este trabajo de tesis está orientado al desarrollo de la aplicación *PetrA*. En esta aplicación se plantea solventar la carencia de los sistemas de redes de Petri que hay, proporcionando un sistema de software libre, y con capacidad de realizar extensiones por parte del usuario de manera consistente y sencilla. Por esta razón *PetrA* de ha desarrollado como una aplicación orientada a objetos en los ambientes de desarrollo de OpenStep y MacOS X, los cuales proporcionan un entorno de trabajo adecuado y sencillo, además de permitir la exportación de la aplicación a Windows utilizando el producto Yellow Box de Apple.

Contenido

Capítulo 1

Introducción

El análisis y diseño de sistemas requiere que los sistemas que están en desarrollo y los ya desarrollados se comporten de manera confiable. Entendamos como confiabilidad a la *"habilidad del sistema para complir su tarea predefinida (a pesar de las fallas de hardware y software)"*.

Existen factores que deterioran la confiabilidad de un sistema como son: fallas, errores, y descuidos (a nivel hardware, sistema operativo, e interacción con otros sistemas, entre otros), por mencionar algunos. En ingeniería de software contamos varios métodos para procurar una buena confiabilidad, como son las tecnologías de desarrollo de software, métodos de validación asistidos por computadora, y manejo de excepciones, entre otras.

La validación es el mecanismo para remover fallas en la fase de desarrollo y así asegurarnos de que el grado de confiabilidad del sistema es aceptable. La validación se puede realizar por modelación y por ejecución.

Las técnicas de validación por modelación se pueden realizar con una inspección del contexto, con una verificación y por una evaluación.

- Cuando se realiza la inspección del contexto se intentan obtener la descripción del flujo de datos y de control, o interacciones entre diferentes elementos del sistema, por ejemplo el análisis de la semántica estática que proporciona UML[1]. Las propiedades que se analizan con estas técnicas son los aspectos pragmáticos (prácticos): anomalías en el flujo de datos y de control. Esto es, analizan las propiedades semánticas generales (cualitativas) y no dependen del tiempo.

[1] UML: Lenguaje unificado de modelación.

- Cuando se realiza una verificación se intenta comprobar la ejecución de un programa mediante simulación de prototipos, o de una ejecución simbólica. Los aspectos que se evalúan con estas técnicas son la funcionalidad, la robustez, la seguridad y la protección de un sistema. Esto es, se inspeccionan las propiedades semánticas especiales (qualitativas) y no dependen del tiempo.

- Las técnicas de evaluación son analíticas y simulativas. Las propiedades que se evalúan son el desempeño, la integridad, y disponibilidad, entre otras. Obtiene propiedades con respecto al tiempo (cuantitativas).

En la validación por ejecución se realizan pruebas cualitativas y cuantitativas para obtener el grado de confiabilidad que se desea. El resultado de dichas pruebas se obtiene en el ambiente real de ejecución y se basan en el tiempo. Se obtienen propiedades de funcionalidad, robustez, integridad y desempeño, principalmente.

Lamentablemente, los métodos de validación no son completos, es decir, un método no puede garantizar todas las medidas de confiabilidad, sin embargo, se complementan unos a otros.

Una herramienta que se utiliza para la validación son las redes de Petri, debido a que son una herramienta matemática que tiene una representación adecuada para la especificación y programación de diferentes lenguajes, los sistemas que simula están en términos de causa-evento, se aplica en distintos métodos de validación y se pueden utilizar en diferentes fases en el ciclo de desarrollo de los sistemas de software. Además de proporcionar un gran poder de modelación para sistemas que tengan aspectos de concurrencia y paralelismo, y se aplica en diferentes niveles de abstracción. Las redes de Petri tienen ventaja sobre otros métodos de validación, como el uso de lenguajes UML, diagramas E-R, o autómatas finitos, en que, además de permitir una visualización del comportamiento dinámico y estático del sistema, se utiliza, como ya se mencionó, en distintas fases del ciclo de software, lo cual lo hace una herramienta más completa para validación que las demás.

Existe una gran variedad de tipos de redes de Petri, debido a que dependiendo del problema al que se apliquen, es necesario hacer extensiones o restricciones a las redes de Petri clásicas. Así, tenemos de redes de Petri coloreadas, jerárquicas, orientadas a objetos y estocásticas, entre otras. El punto débil de las redes de Petri es su tamaño, debido a que cuando se utilizan para modelas sistemas muy grandes tienden a crecer demasiado, para

solucionar este problema, se han ideado diferentes mecanismo de análisis y manejo de estas redes. El mecanismo que más se utiliza es el agrupamiento de partes de una red de Petri en pequeñas subredes.

Se han desarrollado varios sistemas comerciales y otros gratuitos, en centros de investigación, que manejan redes de Petri. Sin embargo, para el usuario se presentan como aplicaciones que manejan un sólo tipo de redes de Petri. Por otro lado estas herramientas no permiten al usuario hacer extensiones o restricciones a la red de Petri, ya sea porque el sistema en si no lo permite, o porque no tiene acceso al código fuente.

Los sistemas que permiten hacer un manejo de redes de Petri más completos son los comerciales, sin embargo, el costo de su licencia es elevado. Y los sistemas libres para redes de Petri sólo manejan un tipo de red de Petri, como se descriobe en la siguiente 1.1.

En este trabajo de tesis, se propone la elaboración de *PetrA*: herramienta para la modelación sy simulación de redes de Petri. Con la ventaja de poder tener un sistema base para realizar extensiones y restricciones, tal que se puedan trabajar con distintas redes de Petri.

1.1 Herramientas de redes de Petri

En la base de datos de redes de Petri de presentada en [30], se presenta una tabla con una descripción general de distintas aplicaciones, comerciales y gratuitas, que trabajan con redes de Petri. En esta lista aparecen herramientas muy completas y poderozas como: *Artifex, Alpha/sim, EDS Petri Net Tool* y *SYROCO*. Esta herramientas están orientadas a resolver problemas de modelación de aplicaciones distribuidas u orientadas a objetos, y que además ayudan en la generación y verificación de código, sin embargo su licencia es bastante alta, varia entre $2000 y $1000 dólares.

Por otro lado, las aplicaciones libres desarrolladas en centros de investigación y escuelas de Ciencias de la Computación se presentan como sistemas que se ejecutan un sólo tipo de redes de Petri —por ejemplo, *Maria* es un sistema orientado a la resolución de la propiedad alcanzabilidad—, o están montadas sobre un Kit de herramientas CASE experimental (*CPN-AMI* y *GDToolKit*), o están escritos en Java (*CoopnTools, Predator, DaNAMiCS, JARP*).

Con la popularización de Java han aparecido distintos editores y simu-

ladores de redes de Petri, ya que varios grupos de trabajo concentran su atención en la portabilidad de estas aplicaciones. Sin embargo estas aplicaciones tienen varios puntos débiles:

- Estas herramientas se desarrollan para trabajar con un tipo de red de Petri, por lo que no se pueden hacer extensiones.
- Tienen problemas de rendimiento (ya que Java, al ser interpretado es lento[2]) y de portabilidad heredados de Java (ver [31]).
- No se discute de manera adecuada su implantación, por lo que resulta tremendamente dificil hacer modificaciones al código fuente que ofrecen.
- Las aplicaciones escritas en Java no manejan todas las características y facilidades que ofrece un sistema operativo para la interacción de sus aplicaciones "nativas".

1.2 PetrA

En este trabajo de tesis hemos desarrollado *PetrA*, una herramienta que permita modelar y simular redes de Petri y que ademas permita incorporarle extensiones de manera fácil y consistente.

Para este fin, *PetrA* se ha diseñado utilizando tecnología orientada a objetos, tanto en el diseño, la codificación, herramientas y el ambiente de desarrollo. Se ha procurado que el diseño de *PetrA* sea consistente para soportar los tipos de redes de Petri más comunes, tal que si el usuario desea egregar un nuevo tipo de red de Petri, sólo tendrá que generar una nueva subclase de alguno de las clases que componen a *PetrA*. Lo cual hace que la creación de nuevos tipos de redes de Petri para el usuario sea una tarea sencilla.

Características que sobresalen de *PetrA* es su objeto principal basado en una matriz de incidencia, tal que toda la información de la red de Petri queda agrupada en estos objetos. Con lo que resulta cobsistente hacer extensiones y módulos para que en *PetrA* se trabajen otros tipos de redes de Petri y objetos de análisis.

[2]Recuerde que todo programa interpretado es de 60 a 70 veces más lento que un programa que se ejecuta en código de máquina

Además *PetrA* se ha desarrollado en los ambientes de desarrollo OpenStep y Mac OS X por diversas razones, entre las que destacan la facilidad de desarrollar aplicaciones gráficas orientadas a objetos de manera sencilla y confiable. En Mac OS X y en OpenStep las aplicaciones se desarrollan con C Objetivo, lo cual permite que se hagan buenos diseños orientados a objetos con alto rendiemiento. Por otro lado, la portabilidad del código está garantizada entre las diferentes versiones de OpenStep, incluyendo la migración al sistema MacOS X, y Windows. Para MacOS X se trabajó con Cocoa, y se modifcaron únicamente los métodos para dibujar. En el caso de Windows, sólo se agregan los módulos de interfaz gráfica (Nibs) y se compila la aplicación con el sistema Yellow Box de Apple. Finalmente, otro atractivo de trabajar en Mac OS X, es que no existen aplicaciones de este tipo en esta plataforma de trabajo.

Este trabajo de tesis está dividido en 6 capítulos. En el primer capítulo se discute la teoría de redes de Petri. En el segundo, se hace una análisis de los distintos tipos de redes de Petri, y se plantea un modelo base para trabajar con redes de Petri. El tercer capítulo, tiene el planteamiento del problema de la aplicación para manejar las redes de Petri, el método para atacarlo, el diseño y la implantación de la aplicación *PetrA*. En el cuarto capítulo se muestra la forma de manejar la aplicación, y algunos ejemplos de su uso. En el quinto capítulo se disctuten las formas de hacer extensiones a *PetrA*. Finalmente, en el sexto capítulo se presentan los resultados, conclusiones y trabajos a futuro de este proyecto.

Capítulo 2

Redes de Petri

Las redes de Petri son una herramienta gráfica y matemática de modelación que se puede aplicar en muchos sistemas. Particularmente son ideales para describir y estudiar sistemas que procesan información y con características concurrentes, asíncronas, distribuidas, paralelas, no determinísticas y/o estocásticas. El objetivo de este capítulo es presentar los conceptos básicos de las redes de Petri. A partir de las definiciones formales de redes de Petri, se presentan aspectos de modelado, comportamiento y técnicas de análisis.

2.1 Generalidades de las redes de Petri

El concepto de red de Petri apareció en 1962 con la tesis doctoral de Carl Adam Petri *"Comunicación con Autómata"* en la Universidad de Bonn. A partir de entonces se difundió en Europa y Estados Unidos, y ya en 1970 aparecieron grupos de investigación que incorporaban las redes de Petri a sus trabajos y/o que se dedicaban a estudiar sus propiedades. Con todo el trabajo acumulado se crearon ciclos de conferencias, libros, actas y revistas en esta área. Aunque no hay memorias y actas de todas las conferencias, los artículos más importantes de éstas, se condensaron en varios libros y revistas.

Las redes de Petri se pueden incorporar informalmente en cualquier área o sistema que pueda describirse gráficamente como diagrama de flujo y que necesitan algunos medios de representar actividades paralelas o concurrentes. Particularmente, en el área de desarrollo de software, las redes de Petri son una herramienta de validación que puede aplicarse en distintas étapas en el

desarrollo de sistemas. Sin embargo, el punto débil de las redes de Petri radica en la complejidad del problema, esto es, los modelos basados en redes de Petri, siempre tienden a ser muy grandes para su análisis. Con el fin de solucionar este problema se han desarrollado técnicas de reducción y extensiones a las redes de Petri. Por lo general, para aplicar las redes de Petri a un problema, se le realizan modificaciones o restricciones.

Algunas áreas donde se aplican las redes de Petri son: evaluación de rendimiento, protocolos de comunicación, modelado y análisis de sistemas distribuidos, sistemas de bases de datos distribuidas, programas paralelos y concurrentes.

2.2 Definiciones formales

Definición 1. Una red de Petri es un tipo particular de grafo dirigido que consiste de dos tipos de nodos (lugares y transiciones). Una red de Petri es una estructura algebraica[1] $PN = (P, T, I, O)$ donde:

- $P = p_1, p_2, \ldots, p_m$ es el conjunto de lugares.
- $T = t_1, t_2, \ldots, t_n$ es el conjunto de transiciones.
- $I : P \times T \rightarrow N$ es la función de entrada en la cual se especifican los lugares de entrada de la transición, con $N = 1, 2, \ldots$
- $O : P \times T \rightarrow N$ es la función de salida en la cual se especifican los lugares de salida de la transición, con $N = 1, 2, \ldots$

Los conjuntos P y T cumplen con $P \cap T = \emptyset$.

En la representación gráfica, la red de Petri se dibuja como un grafo con dos tipos de nodos: lugares y transiciones. Los lugares se representan como círculos y las transiciones, como barras o cajas. Un arco dirigido de un lugar p a una transición t define una entrada de dicha transición. Un arco dirigido de una transición t a un lugar p define la salida de la transición. En ocasiones es necesario colocar valores de peso a los arcos y se denota por $w(p, t)$, donde w es la función

$$w : (P \times T) \cup (T \times P) \rightarrow N.$$

[1] Existen divesas notaciones de redes de Petri, nosotros preferimos utilizar la que aparece en[2].

Cuando un arco no tiene señalado su valor de peso, por omisión se toma con valor 1.

Definición 2. Una *marca* m de una red de Petri es una función $m : P \to N$, lo cual asigna a cada lugar $p \in P$ un número de *token*s. La presencia o ausencia de *token*s indica el estado de un lugar, y la marca de lugares representa la disponibilidad de un recurso, o la ocurrencia de operaciones.

La marca asigna a cada lugar un número entero no negativo. Gráficamente colocamos k puntos en un lugar p, si éste tiene asociado k tokens. Una marca se denota por M, el cual es un m vector donde m es el número total de lugares. La componente p-iésma de M, denotado por $M(p)$, es el número de tokens en el lugar p.

Cuando se modelan sistemas se toman en cuenta dos conceptos fundamentales: las condiciones y los eventos (que se generan a partir de las condiciones). Las redes de Petri representan las condiciones como lugares y los eventos como transiciones. Una transición (evento) tiene un cierto número de lugares de entrada y salida, las cuales representan las precondiciones y las postcondiciones del evento respectivamente.

El comportamiento de muchos sistemas se puede describir en términos de los estados del sistema y sus cambios. En las redes de Petri, para simular el comportamiento dinámico de un sistema, un estado o *marca* de la red cambia de acuerdo con las siguientes reglas de transición:

1. Se dice que una transición t está habilitada si cada lugar p de entrada de t tiene al menos $w(p,t)$ *token*s, donde $w(p,t)$. es el peso del arco de p a t.

2. Una transición habilitada puede o no dispararse (dependiendo en que evento tome o no el lugar).

3. Un disparo de una transición habilitada t remueve $w(p,t)$ *token*s de cada lugar de entrada p de t, y agrega un $w(t,p)$ *token*s por cada lugar de salida p de t, donde $w(t,p)$ es el peso del arco de t a p.

Las transiciones que no tienen lugares de entrada se les llama *transiciones fuente*. Una transición fuente siempre está habilitada. Por otro lado una transición sin lugares de salida consume tokens, pero no los produce.

Se dice que hay un autociclo, cuando un par de nodos, un lugar p y una transición t, cumplen con: p es entrada y salida de t. Una red de carece de autociclos se denomina red simple.

Un ejemplo de una transición se muestra en la figura 2.1. En esta figura se muestra el resultado parcial del comportamiento poblacional de zorros y conejos, donde cada zorro hambriento debe comer 3 conejos para satisfacer su apetito. Podemos observar que la transición t tiene 2 lugares de entrada —1 para los zorros hambrientos y otra para los conejos—, y uno de salida (¡para el zorro feliz!).

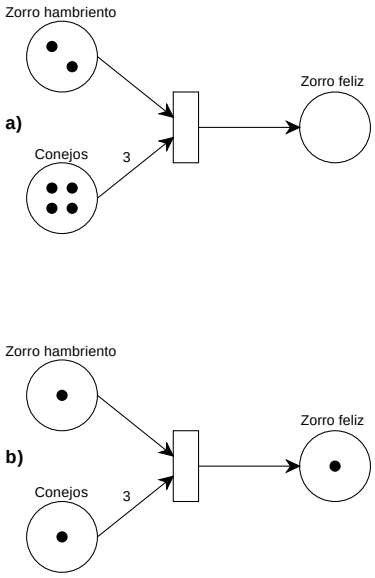

Figura 2.1: Ilustración de una regla de transición.

En la figura 2.1a, se muestra que hay dos zorros y cuatro conejos disponibles, por lo que la transición t se habilita. Después de que t se dispara, la marca se cambia al que se muestra en la figura 2.1b, y la transición t ya no está habilitada. Lo que significa que después de que se ha producido el evento, queda 1 conejo, un zorro hambriento y un zorro feliz.

Observemos que las transiciones asumen que cada lugar puede almacenar un número ilimitado de tokens, es decir, se asume que cada red de Petri es una red de capacidad infinita. Sin embargo, en muchos casos de modelación es preferible tener un límite superior para el número de *token*s que se pueda

almacenar cada lugar. A estas redes se les considera como redes de capacidad finita y se denotan como (PN, M_0), donde M_0 representa la marca inicial, y cada lugar de p de la red tiene asociado una capacidad máxima de $k(p)$ tokens. Para que una transición t de una red de capacidad finita se habilite se agrega la *regla estricta de transición*[2], en la cual de debe cumplir que cada lugar p de salida, no exceda su capacidad $k(p)$ después de disparar t.

2.3 Aspectos de Modelación con redes de Petri

Las redes de Petri se prestan para la modelación si se conoce la estructura causa-evento[3] de un sistema y se utiliza para definir el modelo. Los lugares representan causas o condiciones, y las transiciones eventos.

En sistemas donde se conoce la estructura causa-evento, las redes de Petri resultan ser una excelente herramienta para establecer un modelo de dicho sistema. En esta representación gráfica, los nodos son lugares que representan causas o condiciones, y las transiciones eventos. Las redes de Petri modelan sistemas dinámicos discretos. En este orden de ideas, los eventos se generan, en una parte local del estado actual del sistema, como variables discretas.

Una red de Petri es una estructura matemática, que permite una rerepresentación gráfica, en donde se incluyen los elementos: lugares transiciones, arcos y tokens, en un diagrama que tiene una sintaxis.

- Los lugares son los elementos pasivos de la red de Petri y, junto con los tokens, se utilizan para modelar los estados del sistema.
- Las transiciones son los elementos activos de la red de Petri, y representan las acciones de un sistema. Estas acciones originan cambios en el estado de la red.
- El conjunto de lugares, transiciones y arcos sin finitos y estáticos. Lo que indica que el sistema no puede tener mas causas y eventos que los que originalmente tiene representados en el modelo.

[2] A la regla que no considera capacidades obligatorias se le denomina *regla débil de transición*

[3] tambien se le conoce como condición-evento

- El conjunto de tokens y marcas pueden cambiar durante la ejecución de la red, describiendo las características dinámicas del sistema modelado.

La propiedad de *valor de peso* a los arcos, hace posible que se especifique el número de tokens que consume la transición de los lugares de entrada y el conjunto de tokens que produce en la salida. Las redes de Petri de capacidad finita y peso en los arcos se les llama *sistemas de lugar/transición.*

Las clases originales de redes de Petri, y los sistemas de lugar/transición son muy conocidos por su uso en modelos de un alto grado de abstracción que tienen que analizarse de manera formal. Pero si el modelo debe respetar más detalles del sistema, o si se debe respetar el tiempo en el modelo, entonces se deben desarrollar más clases de redes de Petri que consideran los aspectos deseados del modelo. Así, surgen las redes de Petri coloreadas, estocásticas, y orientadas a objetos, por mencionar algunas, que en general forman el grupo de redes de Petri extendidas.

Ejemplos de modelación con redes de Petri son el flujo de información y los protocolos de comunicación. Se muestran ejemplos de estos modelos.

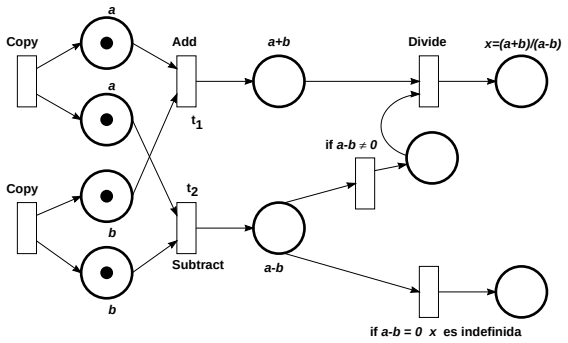

Figura 2.2: Se muestra el flujo de datos de la computación para $x = \frac{(a+b)}{(a-b)}$.

2.3.1 Ejemplo de Modelación de flujo de datos

La red de Petri que se muestra en la figura 2.2 representa un sistema donde se realiza una computación de flujo de datos cuando las instrucciones se habilitan para ejecutarse cuando llegan sus operandos, y se pueden ejecutar concurrentemente. En la representación de redes de Petri de la computación

de flujo de datos, los tokens denotan los valores actuales de los datos, así como la disponibilidad del dato. En la red que se muestra en la figura 2.2, las instrucciones están representadas por las transiciones t_1 y t_2, y se pueden ejecutar concurrentemente y depositar sus resultados —$(a + b)$ o $(a - b)$— en sus respectivos lugares de salida.

2.3.2 Ejemplo de Modelación de protocolos de comunicación.

Los protocolos de comunicación son otra área donde las redes de Petri se pueden utilizar para representar algunas características específicas y esen-

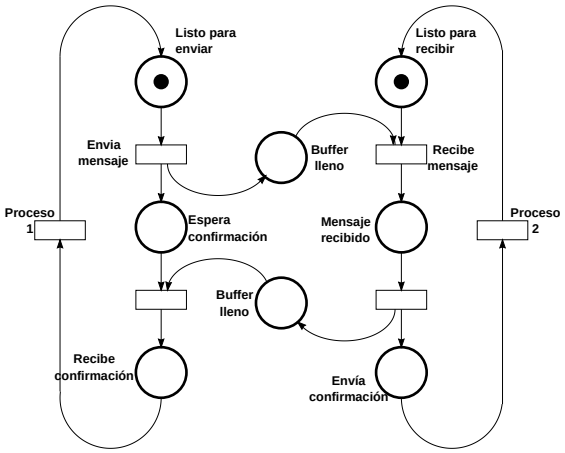

Figura 2.3: Modelo simplificado de un protocolo de comunicación.

ciales. Frecuentemente las propiedades de las redes de Petri como activa y seguridad se utilizan como criterios de validación en los protocolos de comunicación. La red de Petri que se muestra en la figura 2.3 es un ejemplo muy sencillo de un protocolo de comunicación entre dos procesos. La figura 2.4 muestra la representación de una espera de proceso no determinista donde t_{r1}, t_{r2} o t_{out} se disparan si se recibe la respuesta 1,2, o no hay respuesta en un tiempo especificado (t_{out}).

2.4 Propiedades de comportamiento

La fuerza principal de las redes de Petri, son su soporte para el análisis de propiedades y problemas asociados con sistemas concurrentes. Dos propiedades se pueden estudiar con un modelo de red de Petri:

- Propiedades de comportamiento. Estas propiedades dependen de la marca inicial M_0, estas propiedades son alcanzabilidad, acotamiento, activa, reversibilidad y estado inicial, persistencia, distancia sincrónica.
- Propiedades estructurales. Estas propiedades son independientes de la marca inicial M_0.

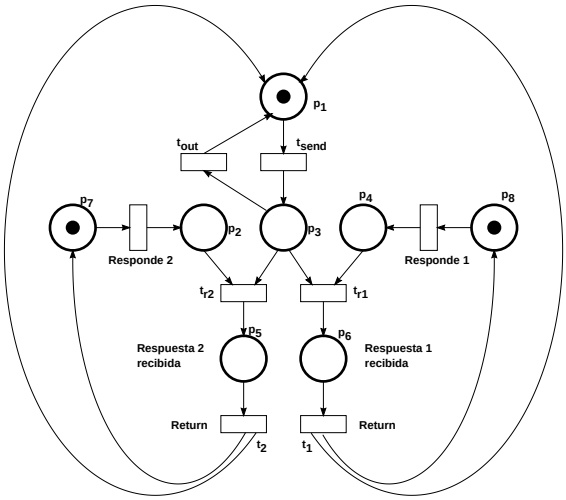

Figura 2.4: Representación de una espera no determinista en un proceso.

2.4.1 Alcanzabilidad

La alcanzabilidad es una base fundamental para estudiar las propiedades dinámicas de cualquier sistema. El disparo de una transición habilitada cambiará la distribución de los tokens en una red, de acuerdo a las reglas de transición mencionadas en la definición 3. Una secuencia de disparos dará como resultado una secuencia de marcas. Se dice que una marca

M_n es alcanzable de una marca M_0 si existe una secuencia de disparos que transformen a M_0 en M_n. Un secuencia de disparos se denota por $\sigma = M_0\ t_1\ M_1\ t_2\ M_2\ \cdots\ t_n\ M_n$ o simplemente por $\sigma = t_1\ t_2\ \cdots\ t_n$ En este caso M_n es alcanzable por M_0 y se denota como $M_0[\sigma > M_n$. El conjunto de todas las marcas posibles alcanzables por M_0 en una red (N, M_0) se denota como $R(N, M_0)$. El conjunto de todas las secuencias de disparo desde M_0 en una red (N, M_0) se denota como $L(N, M_0)$.

El problema de alcanzabilidad en las redes de Petri consiste en encontrar una $M_n \in R(N, M_0)$ deseada.

2.4.2 Acotamiento

Se dice que una red de Petri (N, M_0) es k-acotada o simplementa acotada si el número de tokens en cada lugar de la red no excede a un número finito k en cada marca alcanzable desde M_0. Se dice que una red de Petri es *libre* si es 1-acotada.

2.4.3 Activa

El concepto de red activa está muy relacionado con la ausencia completa de candados mortales en sistemas opertaivos. Se dice que una red de Petri está activa si, no importando que marca se alcance desde M_0, si aún se puede realizar una secuencia de disparo. Esto significa que una red de Petri activa garantiza la ausencia total de candados mortales, no importando la secuencia de disparos que se seleccione.

La propiedad de red activa es el ideal para muchos sistemas, sin embargo, verificar esta propiedad en sistemas grandes resulta resulta poco práctico y muy costoso.

La condición activa se define (o mide) en diversos niveles: Se dice que una transición t en una red de Petri (N, M_0) es:

1. *Muerta* ($L0$-viva) si t nunca se dispara en ninguna secuencia en $L(N, M_0)$.

2. $L1$-activa (*potencialmente disparable*) si t puede dispararse al menos una vez en alguna secuencia en $L(N, M_0)$.

3. $L2$-activa si dado cualquier entero positivo k, t puede dispararse al menos k en alguna secuencia en $L(N, M_0)$.

4. $L3$-activa si t aparece infinitamente, frecuentemente en alguna secuencia en $L(N, M_0)$.

5. $L4$-activa o activa si t es $L1$-activa para cualquier marca en $R(N, M_0)$.

Se dice que una red de Petri (N, M_0) es Lk-viva si cualquier transicción en la red es Lk-activa (con $k = 0, 1, 2, 3, 4$).

2.4.4 Reversibilidad y estado inicial

Se dice que una red de Petri es reversible si M_0 es alcanzable para cualquier marca $M \in R(N, M_0)$. Así, una red reversible regresará a su estado inicial.

2.4.5 Cubrir

Se dice que una marca M en una red de Petri (N, M_0) se puede cubrir si existe una marca $M' \in R(N, M_0)$ tal que $M'(p) \geq M(p)$ para cada p en la red. La propiedad de cubrir está muy relacionada con la propiedad $L1$-activa de las transiciones. Sea M la marca mínima para habilitar la transición t. Entonces t está muerto sí y solo sí, M no se puede cubrir. El corolario a lo anterior nos dice que si t es $L1$-activa sí y solo sí, M sepuede cubrir.

2.4.6 Persistencia

Se dice que una red de Petri (N, M_0) es persistente, si para cualesquier par de transiciones habilitadas, el disparo de una transición no deshabilita a la otra. Una transición en una red persistente, estará habilitada hasta que se dispare. Esta noción de persistencia es muy utilizada en el contexto de esquemas de programas paralelos y circuitos asíncronos altamente independientes.

2.4.7 Distancia sincrónica

La noción de distancia sincrónica es un concepto fundamental introducido por C.A. Petri. Es una métrica muy relacionada al grado de dependencia mutua entre dos eventos en un sistema de condición/evento. Definiremos la distancia sincrónica entre dos transiciones t_1 y t_2 en una red de Petri (N, M_0) por

$$d_{12} = \max |\overline{\sigma}(t_1) - \overline{\sigma}(t_2)|$$

donde σ es una secuencia que inicia en cualquier marca M en $R(N, M_0)$ y $\overline{\sigma}(t_i)$ es el número de veces que la transición t_i (con $i = 1, 2$) dispara en σ.

2.4.8 Equidad

En la literatura de redes de Petri[23, 24], se han propuesto distintas nociones de equidad, de las cuales presentamos sólamente dos conceptos básicos: equidad acotada y equidad incondicional o global:

Equidad Acotada. Se dice que dos transiciones t_1 y t_2 están en una relación de equidad acotada si el número máximo de veces en que una dispara mientras la otra no, está acotada. Se dice que una red de Petri (N, M_0) es de equidad acotada si cada par de transiciones en la red esta en una relación equidad acotada.

Equidad incondicional o global . Se dice que una secuencia σ es de equidad incondicional si es finita o si cada transición en la red aparece infinitamente en σ. Se dice que una red (N, M_0) es de equidad incondicional si cada secuencia de disparo σ de $M \in R(N, M_0)$ es de equidad incondicional.

2.5 Métodos de Análisis

Los métodos de análisis para las redes de Petri se pueden clasificar en tres grupos generales:

1. Método de aŕbol de alcanzabilidad o cubrimiento.
2. Enfoque de matriz de ecuaciones.
3. Técnicas de reducción o descomposición.

2.5.1 Árbol de alcanzabilidad.

Dada una red de Petri (N, M_0) desde una marca inicial M_0 podemos obtener tantas nuevas marcas como transiciones habilitadas. Así, de cada nueva marca podemos obtener más marcas. Este proceso genera un árbol de marcas. Los nodos representan las marcas generadas a partir de M_0 (la raíz) y

sus sucesores, y cada arco representa un disparo de una transición, la cual transforma una marca en otra.

Algunas de las propiedades que se pueden estudiar utilizando el árbol de alcanzabilidad T para una red de Petri (N, M_0) son las siguientes:

1. Una red (N, M_0) es acotada y así $R(N, M_0)$ es finito si y solo si ω no aparece en nigún nodo etiquetado en T.

2. Una red (N, M_0) es libre si y solo si solo aparecen ceros y unos en las etiquetas de los nodos de T.

3. Una transición t es muerta si y solo si no aparece como etiqueta de un arco en T.

4. Si M es alcanzable desde M_0, entonces existe un nodo etiquetado M' tal que $M \leq M'$.

2.5.2 Matriz de incidencia y ecuación de estado

La matriz de incidencia y las ecuaciones de estado han resultado ser un enfoque muy apropiado para describir y analizar completamente el comportamiento dinámico de las redes de Petri. Su planteamiento a través de las ecuaciones lineales (con matrices) permite utilizar el bagaje teórico para analizar el comportamiento dinámico. Sin embargo, debido a la naturaleza no determinística de los modelos de redes de Petri, se tiene que las ecuaciones involucradas tienen soluciones limitadas.

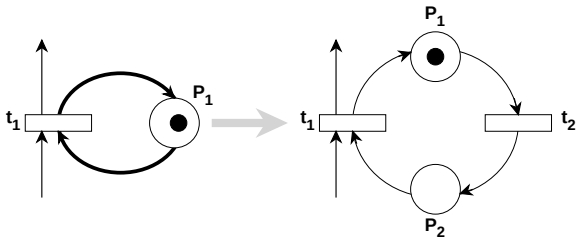

Figura 2.5: Transformación de autociclos a ciclos

Matriz de incidencia

Para una red de Petri pura[4] PN con n transiciones y m lugares, la matriz de incidencia $A = [a_{ij}]$ es una matriz $m \times n$ de enteros y su entrada esta definida por:

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-}$$

donde $a_{ij}^{+} = w(i,j)$ es el peso del arco de la transición i a su lugar de salida j, y $a_{ij}^{-} = w(i,j)$ es el peso del arco de la transición j a su lugar de entrada i. Esto es, la matriz de incidencia representa los lugares de entrada con valores negativos, y los valores positivos representan salidas. Además una transición j se encuentra habilitada en una marca M si

$$a_{ij}^{-} \leq M(j), \quad j = 1, 2, \ldots, m.$$

Para hacer pura a una red de Petri se deben sustituir los autociclos por ciclos, como se muestra en la figura 2.5

Ecuación de estado

En una red de Petri, la matriz de estado se define de forma recurrente de la forma siguiente:

$$M_k = M_{k-1} + A^T u_{k'} \tag{2.1}$$

donde M_k es el vector columna, de dimensión $m \times 1$, que representa el estado actual k del sistema; M_{k-1} es el vector columna, también de dimensión $m \times 1$, que representa el estado anterior $k-1$ del sistema; A^T es la matriz transpuesta de la matriz de incidencia de la red de Petri; y $u_{k'}$ es el vector columna con $n-1$ ceros y un 1 en la j-ésima posición, indicando la transición que habrá de dispararse.

Condición necesaria de alcanzabilidad: Suponga que una marca destino M_d se alcanza desde una marca M_0 a través de una secuencia de disparos $u_1, u_2, \ldots, u_d$. Escribiendo la ecuación de estado (2.1) para $j = 1, 2, \cdots, d$ y sumándolos tenemos el siguiente desarrollo:

$$M_1 = M_0 + A^T u_1$$

[4]Una red de Petri pura es aquella red que no tiene lugares que son entrada y salida de la misma transición.

$$
\begin{aligned}
M_2 &= M_1 + A^T u_2 = M_0 + A^T u_1 + A^T u_2 \\
M_3 &= M_2 + A^T u_3 = M_0 + A^T u_1 + A^T u_2 + A^T u_3 \\
&\vdots \\
M_d &= M_0 + A^T \sum_{k=1}^{d} u_k
\end{aligned} \tag{2.2}
$$

La ecuación 2.2 puede escribirse como:

$$A^T x = \Delta M \tag{2.3}$$

donde $\Delta M = M_d - M_0$ y $x = \sum_{k=1}^{d} u_k$. El vector columna x de dimensión $n \times 1$, con entradas de enteros no negativos, se denimona vector de conteo de disparos. La i-ésima entrada de x denota el número de veces que la transición i se debe disparar para transformar M_0 en M_d. Además de [25] se tiene que el conjunto de ecuaciones algebraicas lineales 2.3 tiene una solución x, sí y solo sí, ΔM es ortogonal a cada solución y de este sistema homogeneo:

$$Ay = 0 \tag{2.4}$$

Sean r el rango de A, y la siguiente partición de A

$$
\mathrm{A} = \begin{array}{c} \overset{\text{m-r}}{\longleftrightarrow} \quad \overset{\text{r}}{\longleftrightarrow} \\ \left[\begin{array}{cc} \mathrm{A_{11}} & \mathrm{A_{12}} \\ \mathrm{A_{21}} & \mathrm{A_{22}} \end{array}\right] \begin{array}{l} \updownarrow \ \text{r} \\ \updownarrow \ \text{n-r} \end{array} \end{array}
$$

donde A_{12} es una matriz cuadrada no singular de orden r. Un conjunto de $(m-n)$ soluciones linealmente independientes y para 2.4 se puede dar como los $(m-r)$ renglones de la siguiente matriz B_f de $(m-r) \times m$ elementos

$$B_f = [I_\mu : -A_{11}^T (A_{12}^T)^{-1}]$$

donde I_μ es la matriz identidad de orden $\mu = m - r$. Note que $AB_f^T = 0$, lo que quiere decir es que el espacio vectorial obtenido por los vectores renglón de A es ortogonal al espacio vectorial obtenido por los vectores renglón de B_f. La matriz B_f corresponde a la matriz de circuito fundamental para el

caso de grafos marcados[20]. Ahora, la condición de que ΔM sea ortogonal a cada solución para $Ay = 0$ es equivalente a la siguiente condición:

$$B_f \Delta M = 0$$

Así, si M_d es alcanzable desde M_0, entonces el vector correspondiente x de conteo de disparos, debe existir y ser válido.

2.5.3 Técnicas de reducción o descomposición

Una estrategia para facilitar el análisis, es reducir el sistema a un modelo más simple el cual pueda conservar sus propiedades de análisis. Las técnicas

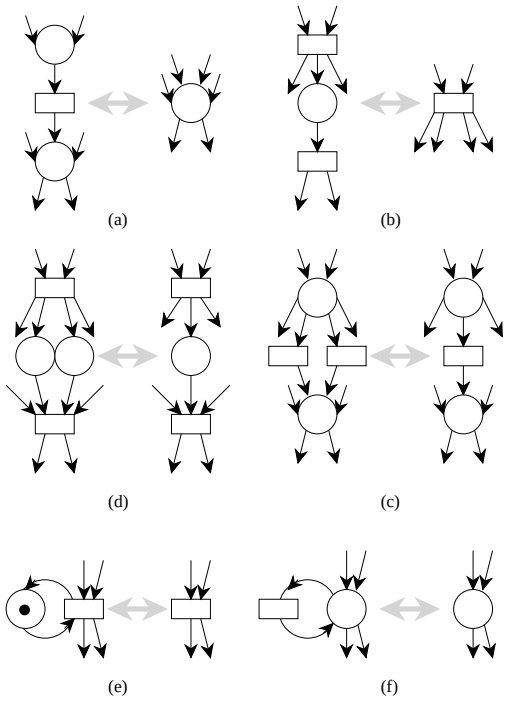

Figura 2.6: Transformaciones que conservan propiedades de análisis

para transformar un modelo abstracto a un modelo más refinado, en una forma jerárquica, se pueden utilizar como síntesis.

Existen muchas técnicas de transformación para redes de Petri. No es difícil ver que las siguientes seis operaciones conservan las propiedades de acotamiento, seguridad y activación [2].

La figura 2.6 muestra distintas transformaciones que conservan propiedades de análisis según la especificación en cada caso.

a Fusión de una serie de lugares.

b Fusión de una serie de transiciones.

c Fusión de lugares paralelos.

d Fusión de transisiones paralelas.

e Eliminación de auto-ciclos de lugares.

f Eliminación de auto-ciclos de transiciones

2.6 Comentarios Finales

Las redes de Petri son una herramienta matemática que nos permite hacer modelacion y análisis de sistemas cuya estructura esté expresada en términos de causa-evento. Las redes de Petri son una estructura algebraica $PN = (P, T, I, O)$, con lugares, transiciones, funciones de entrada y salida, y marcas. Las redes de Petri también pueden definirse como un tipo particular de grafo con dos tipos de nodos: lugares y transiciones. Los lugares transiciones forman la parte estática de la red de Petri, y las funciones de entrada y salida y las marcas forman la parte dinámica de la red.

La parte fuerte de las redes de Petri son su ayuda para analizar diferentes propiedades de un sistema, estas propiedades son: alcanzabilidad, acotamiento, activa, reversibilidad y estado inicial, cubrir, persistencia, distancia sincrónica, y equidad entre otras. Las redes de Petri tienen distintas técnicas de análisis, las que aqui se han discutido son: árbol de alcanzabilidad, enfoque de matriz de ecuaciones y técnicas de reducción o descomposición.

Una buena introducción a las redes de Petri y sus mecanísmos de análisis se pueden encontrar en[1, 2, 18] y [22].

Las redes de Petri tienen su principal problema cuando modelan sistemas grandes,debido a que la red de vuelve compleja y demasiado extensa. Por

otro lado, para aplicar redes de Petri a muchos problemas específicos, es necesario aplicarle restricciones o hacerle extensiones al modelo original. En el siguiente capítulo se discuten las clases principales de redes de Petri —que intentan resolver este problema y que hacen extensiones al modelo original— y las herramientas de softwareque que se tienen actualmente para utilizarlas.

Capítulo 3

Manejo de redes de Petri

Cuando se aplican redes de Petri a problemas específicos, es necesario realizar restricciones o agregar nuevas propiedades a sus elementos (lugares, transiciones, tokens y arcos). Actualmente, se han desarrollado una gran variedad de tipos de redes de Petri, y las herramientas de software que se utilizan para manejarlas, están orientadas a trabajar un sólo tipo de red, o se enfocan a especializar un tipo de simplificación o de análisis. Esta especialización limita al usuario para que realice alguna extención, o restricción, a la red de Petri para modelar las propiedades del sistema que se desean. Esta es la principal motivación que tenemos para construir una herramienta que tenga capacidades de extensión desde sus niveles básicos.

En este capítulo se discuten las características de los distintos tipos de redes de Petri —para identificar sus puntos comúnes y diferentes—, lo cual nos permitirá establecer un modelo base de red de Petri, que pueda tener capacidades de extensión para manejar distintos tipos de redes.

3.1 Tipos de redes de Petri

Actualmente, se han desarrollado una gran variedad de tipos de redes de Petri (para poder aplicarse a problemas específicos) y existen distintas clasificaciones para ellas, en este documento, se utiliza la descrita por Wil v.d. Aalst [21]. En esta clasificación se distinguen las redes de bajo nivel (redes lugar/transición) y las extensiones más comunes (coloreadas, estocásticas y jerárquicas).

3.1.1 Redes de Petri clásicas

Las redes de Petri clásicas mantienen la estructura que se ha mencionado en la Definicion 1, de la sección 1.2. Donde la red es un grafo bipartita con dos tipos de nodos: arcos y transiciones. Estos nodos están conectados entre sí a través de arcos —con la regla de que un lugar no sea una entrada y salida de una transición—. Los arcos definen las funciones de entrada y salida la red. Los lugares tienen asociados elementos llamados tokens. Los tokens definen el estado de la red y se utilizan para establecer las condiciones de disparo de una transición —esta acción modifica el estado de la red, por lo que los tokens defienen las características dinámicas del sistema—.

Las redes de Petri clásicas también se les conoce como redes lugar/transición. Otros autores les llaman reded de Petri de bajo nivel [6, 7, 11, 13, 19, 27, 28], debido a que las modificaciones o restricciones hechas a la red no involucran más que condiciones de conexión de arcos y el número de tokens que pueden almacenar los lugares. Así, tenemos a las redes de Petri de capacidad finita (lugares con límite en los tokens que pueden almacenar), puras (no contienen autociclos), y ordinarias (si el peso de los arcos es 1), como se discute en [2] y en [22].

3.1.2 Extensiones comúnes de Redes de Petri

Las redes de Petri clásicas permiten modelar estados, eventos, y condiciones, sincronizaciones, y paralelismo, entre otras características del sistema. Sin embargo, las redes de Petri que describen el mundo real tienden a ser complejas y extremadamente grandes. Mas aún, las redes de Petri clásicas no permiten la modelación de datos y tiempo. Para solucionar este problema, se han propuesto muchas extensiones. Las extensiones se pueden agrupar en tres categorías: extensiones con tiempo, con colores y con jerarquías.

Extensiones con tiempo

Un punto importante en los sistemas reales es la descripción del comportamiento temporal del sistema. Debido a que las redes de Petri cásicas no son capaces de manejar tiempo de forma "cuantitativa", se le agrega al modelo el concepto de tiempo. Las redes de Petri con tiempo pueden dividirse, a su vez, en dos clases: redes de Petri de tiempo determinístico (o redes

de Petri regulares) y redes de Petri de tiempo estocástico (o redes de Petri estocásticas).

La familia de redes de petri con tiempo determinístico incluyen a las redes de Petri que tienen asociadas un tiempo de disparo determinado en sus transiciones, lugares o arcos.

La familia de redes de petri con tiempo estocástico incluyen a las redes de Petri que tienen asociadas un tiempo de disparo estocástico en sus transiciones, lugares o arcos.

Debido a que este tipo de red asocia un tiempo de retardo para ejecutar una transición habilitada, no se puede establecer un arbol de alcanzabilidad ya que la evolución no es determinística. Los métodos de análisis asociados a este tipo de redes son las cadenas de Markov, donde la probabilidad de la aparición de un nuevo estado depende únicamente del estado anterior. En [17] y [18] se da una explicación detallada al respecto.

Extensiones con jerarquías

El problema de tamaño que tienen las redes de Petri cuando se modelan sistemas reales, se puede tratar con el uso de las redes de Petri jerárquicas. Estas redes proporcionan, como su nombre lo indica, una jerarquía de subredes. Una sub red es un conjunto de lugares, transiciones, arcos e, incluso, subredes. De tal forma que la construcción de un sistema grande, se basa en un mecanismo de estructuración dos o más procesos, representados por subredes. Tal que, en un nivel se da una descripción simple de los procesos, y en otro nivel queremos dar una descripción más detallada de su comportamiento.

Las extensiones a paartir de jerarquías aparecieron como extensiones a las redes de Petri coloreadas [27, 28]. Sin embargo, se han desarrollado tipos de redes de Petri jerárquicas que no son redes de Petri coloreadas [6, 9, 13, 19].

Un problema principal es la manera de trabajar con redes jerárquicas. Se han realizado estudios de comportamiento y análisis de transformaciones de una subred en una transición y transformaciones de una subred un lugar, para determinar su equivalencia de comportamiento y análisis con una red que no es jerárquica [27, 28]. Se han intentado implantar otros mecanismos, como en [9], de agrupamiento que incluyen un nuevo elemento gráfico (la subred) con la finalidad de que la semántica de los grafos se mantenga, sin emgargo, en este tipo de agrupamiento no se aclara si se mantiene la equivalencia entre

la red de Petri jerárquicon las redes de Petri clásicas lo que puede generar problemas de análisis. Para solucionar este problema se han propuesto varias ideas como trabajar a las redes de Petri en un esquema cliente-servidor[19], donde las subredes se comunican entre sí,a través de un grafo lugar. Otros trabajos han demostrado que es más fácil, establecer esta comunicacón con la subred, a través de lugares y tratarla, con sus debidas restricciones, como una transformación de la subred a transición [20]. De esta forma, los elementos en la subred que se comunican con el exterior son transiciones, y los elementos de la red que se comunican con las subredes son lugares.

Extensiones con color

Las redes de Petri coloreadas son una extensión a las redes de Petri que tiene incorporado un lenguaje de modelación. Las redes de Petri coloreadas se consideran un lenguaje de modelación desarrollados para sistemas en los cuales, la comunicación, sincronización y el uso compartido de recursos son importantes. Así, las redes de Petri coloreadas combinan las ventaja de las redes de Petri clásicas y los lenguajes de programación de alto nivel. Para hacer más claro esta afirmación, listaremos las características de los elementos gráficos de este tipo de redes.

Token Este es el elemento más distintivo de estas redes. Las redes de Petri coloreadas permiten trabajar con tokens de distintos colores para representar valores y los tipos de datos que trabaja el sistema modelado. Esta distinción en los colores del token clarifica, al usuario, la interpretación de los tokens durante la ejecución de la red —recuerde que en las redes de petri clásicas por convención se utilizan los tokens de color negro—.

Arcos Los arcos tienen asociados una función que determina el valor del token de salida. Esta función está dada en términos del lenguaje de modelación asociado a la red.

Transiciones Las transiciones permiten cambiar el estado de la red cuando se disparan. En las redes de Petri coloreadas, las transiciones permiten que el usuario seleccione algún valor arbitrario para forzar el valor del token de salida. Esta selección de valor se realiza ejecutando las funciones que tienen asociados los arcos de entrada y salida. En las

implantaciones, las transiciones deben permitir una ejecución manual, para que el usuario pueda modificar estos valores, o una automática, dejando que las probabilidades decidan algunos valores internos de las funciones de los arcos.

Lugares Los lugares pueden tener distintas representaciones iconográficas. Estos objetos almacenan tokens de diferentes colores. Aunque no es regla que todos los lugares manejes manejen tokens de todos los colores que utiliza el modelo, en realidad esto depende del resultado de las funciones asociadas a las transiciones de entrada.

Las redes de Petri coloreadas fueron las primeras en incluir el concepto de jerarquía en las redes, a través del concepto de página [11, 27, 28]. Un modelo de redes de Petri coloreadas consiste en un conjunto de módulos (páginas), las cuales tienen redes de lugares, transiciones y arcos. Y los módulos interactúan entre si a través de interfaces bien definidas.

Las redes de Petri coloreadas, también pueden considerar referencias de tiempo, teniendo a las redes de Petri con tiempo como un subconjutno de este tipo de redes.

Las redes de Petri coloreadas permiten varios métodos de verificación formal: grafo de ocurrencia, matriz de incidencia y ecuaciones de estado, y reducciones.

Una característica fundamental de las redes de Petri coloreadas es que están muy ligadas de su lenguaje de modelación, por lo que su uso depende de la herramienta que lo soporte.

3.2 Una base de manejo común

Las redes de Petri coloreadas intentan manejar todos los tipos de redes de Petri. Sin embargo, tienen el problema de que no hay aún, una definición estandar del lenguaje de modelación, y su uso depende del programa que se maneje —con todo lo que esto implica: su costo, su incapacidad de realizar extensiones por parte del usuario, y su grado de especialización hacia un tipo de problema—, además, con este tipo de herramientas se pierde la claridad en las propiedades regulares de la red de Petri, complicando la interpretación de resultados [14, 15]. Por lo que resulta deseable desarrollar una base para

manejar redes de Petri, con capacidad de hacer extensiones para trabajar con redes de Petri de alto nivel.

Primero se debe definir la estructura básica para trabajar con cualquier tipo de res de Petri. La idea obvia es tener una red de Petri clásica, debido a que las redes de Petri de alto nivel se han desarrollado a partir de extensiones a las redes de Petri clásicas, resultando que las redes de bajo nivel resultan ser un caso especial de las redes de Petri de alto nivel. Las redes de Petri con tiempo con retardo 0 en su valor tiempo se comportan como una red lugar/transición. También se obtiene el comportamiento de una red clásica, si se considera un solo nivel en las redes de Petri jerárquicas. Y si en las redes de Petri coloreadas, se utiliza un sólo color de token y las funciones de los arcos sólo pasan el token sin hacerle ninguna modificación, se obtiene el comportamiento de una red clásica.

Evidentemente existen diferencias, que se reflejan en las propiedades y métodos de análisis, entre las redes de bajo y alto nivel, sin embargo se pueden establecer equivalencias entre ambos tipos de redes [28]. Por esta razón las características de comportamiento y modelación no chocan entre los diferentes tipos de redes.

Toda la información de una red de Petri puede almacenarse en una matriz de incidencia. Las redes de Petri coloreadas y estocásticas no tienen problema con esta representación. No hay problemas para trabajar con el esquema de matriz de incidencia, debido a que en [20] se prueba que se puede tartar a las subredes como transiciones ya que la mejor manejar de comunicar a las subredes con los elementos de la red, es a través de lugares (pero esto debe aplicarse con cuidado en los mecanismos de análisis).

Además, los nodos transición pueden tener asociado un hilo de control para simular la ejecución de la red. Con este hilo asociado a la transición se pueden manejar los retardos en la ejecución de un disparo para simular el tiempo asociado a los distintos elementos de la red de Petri: lugar, transición y arco [4].

Los nodos lugar pueden manejar listas con el número de nodos de colores con los que está marcada. Esto no representa problemas para trabajar con las redes de Petri clásicas, jerárquicas y con tiempo, ya que únicamente trabajan con un tipo de token. Además, los arcos pueden tener asociado un lenguaje que se utilice como macro, y la transición se encarge de leerlo y ejecutarlo cuando se dispare. En este punto se puede pensar en un lenguaje como REC para que se utilice como lenguaje de modelación [29].

La gran diferencia entre los distintos tipos de redes son los métodos de análisis. Sin embargo este problema se puede solucionar si se establece un modelo orientado a objetos y utiliza el esquema de delegado para que la red de Petri pueda cambiar, en momento de ejecución, el comportamiento de una acción de análisis (dependiendo del tipo de red que se trabaje).

3.3 Comentarios Finales

Muchas herramientas no permiten al usuario hacer extensiones y/o restricciones a una red de Petri para trabajar con nuevos tipos. Y las que trabajan con más tipos de redes mantienen esta limitación para el usuario. Se puede diseñar una base que puede trabajar con los distintos tipos de redes de Petri partiendo de una red de Petri clásica. Esta base puede estar en términos de la matriz de incidencia, ya que las extensiones para manejar las redes basadas en tiempo, las coloreadas y las jerárquicas son factibles.

Capítulo 4

PetrA: **Petri Nets Application**

Existen diferentes tipos de redes de Petri, dependiendo del problema de aplicación. La mayoría de los programas que trabajan con redes de Petri, se enfocan a trabajar con un sólo tipo de red, lo cual limita la explotación de las redes de Petri por parte del usuario. De esta forma, planteamos el diseño de una aplicación que trabaje con redes de Petri y que le ofresca facilidades al usuario de incorporar nuevos tipos de redes de Petri de una manera sencilla. A esta aplicación le hemos denominado *PetrA* (*Petr*i Nets *A*pplication). Para obtener una aplicación que pueda manejar la mayoría de los tipos de redes de Petri y que permita la incorporación de nuevos tipos de redes, se debe cuidar su diseño y su implantación. Para el diseño se eligió el modelo orientado a objetos, y se trabajó a partir de la base que se desprende del análisis realizado en las sección 3.2, incorporándole características de una aplicación que trabaja con múltiples documentos. En la fase de implantación se eligió el lenguaje de programción Objective-C y la plataforma de desarrollo de OpenStep y Mac OS X.

En este capítulo se discute el diseño orientado a objetos de la aplicación miltihilos *PetrA*, y su implantación sobre las plataformas OpenStep y MacOS X. Cuando se explican las distintas clases que componen a *PetrA*, se hacen referencias técnicas de su implantación. La finalidad es tener un documento con detalles del comportamiento interno que permita posteriormente, entender la implantación a las personas que estén interezadas en extender el trabajo.

4.1 Planteamiento

Como cualquier aplicación, *PetrA* debe cumplir con conjunto de propiedades que le permitan al usuario trabajar con redes de Petri de una manera sencilla y confiable.

La aplicación *PetrA* tiene como propiedades las siguientes:

- Capacidad de almacenar la red de Petri como un documento.
- Capacidad de trabajar con diferentes documentos en forma simultánea.
- Contar con un editor gráfico para el diseño de la red de Petri.
- Ejecutar la simulación de la red de Petri sobre el diseño gráfico correspondiente.
- Capacidad de poder incorporar los diferentes tipos de redes de Petri que existen (por ejemplo redes coloreadas, orientadas a objetos y redes basadas en tiempo).

4.1.1 La plataforma de desarrollo

Originalmente se seleccionó como plataforma de desarrollo a OpenStep por varios motivos, entre los que destacaron fue la elegancia de sus herramientas y bibliotecas para el desarrollo y manejo de aplicaciones orientadas a objetos, la capacidad de utilizar múltiples hilos de control en una aplicación e implantarlo de una manera sencilla, la factibilidad de portar de manera casi transparente a GNUStep y Mac OS X, y de exportarlo a Windows (compilando la aplicación con el producto Yellow Box de Apple).

Finalmente, la aplicación terminó desarrollándose en Mac OS X, ya que con la aparición de este ambiente de trabajo también se incorporaron nuevas bibliotecas y algunos cambios técnicos que dificultaron la portabilidad de la aplicación de manera directa de OpenStep a Mac OS X —principalmente en el asignamiento de memoria de memoria al momento de crear objetos—.

4.1.2 El modelo base de la red de Petri

Como se ha descrito en la sección 3.2. La información general de la red de Petri la podemos agrupar en la matriz de incidencia. Partiendo de esta

matriz de incidencia podemos saber cuantos nodos tiene la red (lugares y transiciones) y como están conectados entre sí, lo que nos permite cononcer las funciones de entrada y salida de la red. Por el hecho de trabajar a la red de Petri como una matriz de incidencia se está obligando que el modelo base trabajará cpm redes de Petri puras, esto es, no se permitirá manejo de autociclos en las redes.

Con este orden de ideas podemos establecer los elementos principales de la red de Petri que se incorporarán a PetrA. Los elementos que se extraen de la teoría de redes de Petri, y de nuestro modelo base son: la red de Petri, los grafos lugar y transición, los arcos y los tokens. Como PetrA se ha desarrollado con el paradigma orientado a objetos, presentaremos los elementos como objetos.

Red de Petri Un elemento red de Petri se encargar de mantener la información de la red de Petri (nodos, funciones y estados) de incorporar mecanismo o elementos que le permitan hacer una representación gráfica de la red, y de mantener facilidades para manejar diferentes mecanismo de análisis. Con estas consideraciones se ha seleccionado a la matriz de incidencia como una clase para manejar a las redes de Petri.

Lugar Los objetos de esta clase deben encargarse de representar a los nodos lugares de la gráfica de red de Petri. El atributo principal de estos objetos es el número de tokens que almacena. Atributos que pueden considerarse son: limite de tokens que puede almacenar (si se trabaja con una red de Petri de capacidad finita), y, en caso de trabajar con redes de Petri coloreadas, puedira sustituir el elemnto del valor del token por un arreglo que indique el número de tokens que trabaja, junto con sus colores.

Transición Los objetos de esta clase deben representar las características gráficas y de comportamiento de este tipo de nodo. Estos objetos deben conocer sus lugares de entrada y salida. Estos objetos deben de tener una unidad de ejecución para realizar los cambios de marcas de la red.

Arco Los objetos de esta clase deben representar gráficamente el arco dirigido entre una transición y un lugar, o entre un lugar y una transición. Este objeto tiene asociado el atributo de peso y debe tener la propiedad

de tener algoritmos para dibujar, no sólo con líneas rectas los arcos entre dos nodos.

Token Los objetos de esta clase representan a los tokens. Para una red sencilla pudieran no tener relevancia, pero son relevantes si se desea extender a *PetrA* para trabajar con redes de Petri coloreadas. Por lo que su atributo será el color.

4.1.3 La arquitectura general de la aplicación

En la sección anterior se ha hecho la descripción de los objetos principales de una red de Petri, sin embargo estos objetos no bastan para desarrollar una aplicación, pues hay que tomar en consideración las interacciones con el ambiente de trabajo. La figura 4.1 muestra la estructura de la aplicación PetrA.

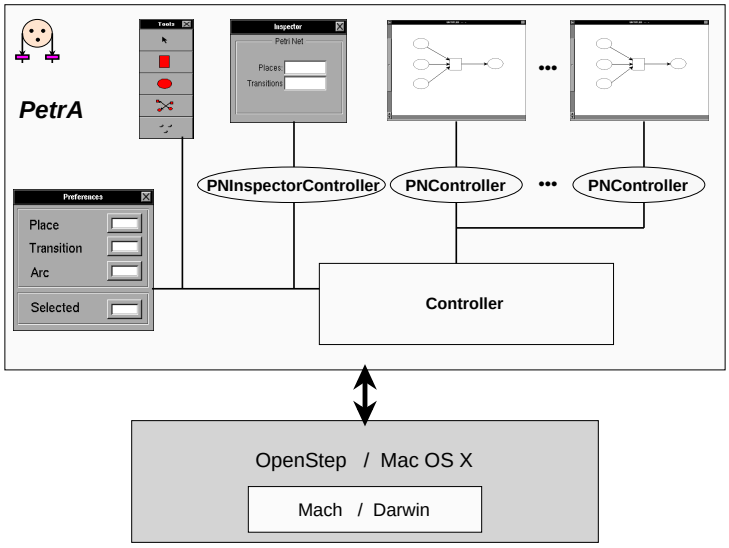

Figura 4.1: PetrA: Aplicación Multihilos

El diseño de *PetrA* se basa en el modelo vista-control[1], donde cada objeto principal gráfico tiene asociado un objeto de control que se encarga de manejar las acciones que aparentemente realizan los objetos visibles. Así,

[1]Model View-Controll

cada panel y ventana tiene asociado un objeto de control. De esta forma, el objeto de la clase *Controller* es el objeto de control para el panel de herramientas y el panel de preferencias. Esto es, el objeto *Controller* se encarga de manejar los eventos que se generan desde estos paneles para afectar a un documento red de Petri (en el caso del panel), o a toda la aplicación (en el caso de las preferencias). Además este objeto se encarga de realizar las interacciones entre los controladores de cada red de Petri, el sistema operativo y algunas operaciones del usuario a través del menú de la aplicación y el panel de herramientas.

Cada red de Petri se maneja como un documento, representado por una ventana y una área de dibujo, y tiene asociado un objeto de control propio (*PNController*) que se encarga de manejar las operaciones específicas de la red de Petri con el ambiente de trabajo. Este objeto de control se encarga de mantener el identificador de la ventana de trabajo y el identificador del objeto gráfico que dibuja la red.

4.2 Diseño e implantación

Expuestas las consideraciones del modelo base para manejar redes de Petri y de la arquitetura general de la aplicación, describiremos las diferentes clases que componen a *PetrA* y algunas características fundamentales de su implantación. En la figura 4.1 se puede visualizar la relación entre las diferentes clases de la aplicación.

La figura hace referencia a algunas clases de la biblioteca de objetos de OpenStep y Mac OS X (especialmente a los *frameworks* Fundation y AppKit). No se muestran algunas relaciones entre las clases de estas bibliotecas ya que no es intención de este trabajo profundizar mucho en ese aspecto. Sin embargo nos interesa revisar la relación que se tiene con las clases que se escribieron para *PetrA*. Las clases con la que se desarrolla la aplicacion son: *Connection, Matrix, Controller, PNController, Place, Element, PNInspectorController, Token, Figure, PNMatrix, Transition, IOConnections, PNView* y *PNRenderingView*.

A continuación se muestra un descripción de cada una de las clases, su relación con las demás y los aspectos tecnicos importantes de su implantación.

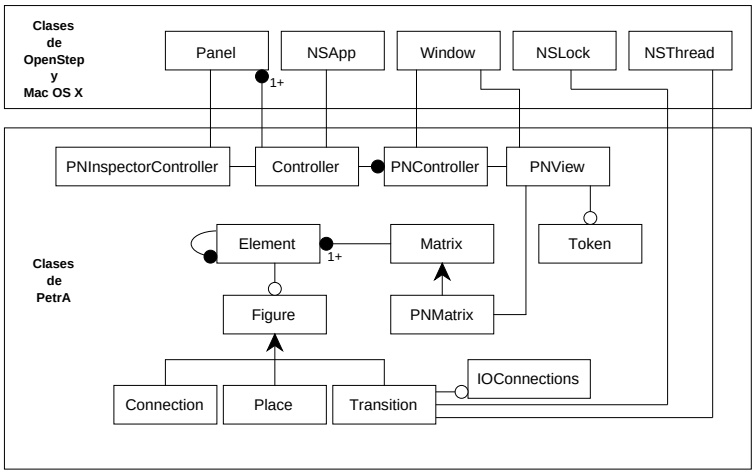

Figura 4.2: Diagrama de clases de *PetrA*

4.2.1 La clase *Controller*

La clase que se encarga de mantener el control de la aplicación en su nivel más bajo es *Controller*, ya que mantiene una relación con *NSApp*. Esto implica que un objeto *Controller* se encarga de establecer las interacciones entre el ambiente de trabajo (del sistema y de la aplicación) y la parte de control del objeto red de Petri (*PNController*). Esta clase también se encarga de manejar al menos un panel (el panel *Tools*), además puede manejar otros paneles como *Inspector* y *Preferences*, y mantiene una relación con un objeto *PNInspectorController*. Los objetos de la clase *Controller* tienen una relación con cero o más objetos *PNController* —esto es porque se puede tener la aplicación cargada con ningún documento abierto—.

Esta clase se encarga de atender los mensajes que manda el usuario a través del menú principal. Se ha mencionado que las redes de Petri se manejaran como documentos, asi pues, el objeto *Controller* atiende mensajes que provienen del menú que manejan documentos para crear, abrir, salvar e imprimir documentos (redes de Petri). También mantiene opciones para desplegar los paneles *Preferences* e *Inspector*, para controlar aspectos generales y particulares de una red de Petri.

Sólo se crea un objeto de la clase *Controller* cuando se ejecuta *PetrA*. Este objeto tiene varias variables de instancia y métodos *sender-target*:

```
@interface Controller : NSObject
{
    id tools;            //Identificador del panel Tools
    id currentDoc;       //Identificador del documento actual
    id infoPanel;        //Identificador del panel Info
    id preferencesPanel; //Identificador del panel Preferences
    id inspectorPanel;   //Identificador del panel Inspector
    id inspector;        //Identificador del controlador del Inspector
    ...
    Class currentGraphic;//Identificador de la clase de herramienta
}
- (void)newDoc:(id)sender;
- (void)openTheFile:(id)sender;
- (void)showInfoPanel:(id)sender;
- (void)showInspectorPanel:(id)sender;
- (void)showPreferencesPanel:(id)sender;
- (void)setCurrentGraph:(id)sender;
...
@end
```

Los identificadores *tools, infoPanel, preferencesPanel* e *inspectorPanel* hacen refrencia a los paneles que llevan el nombre del identificador. El identificador *currentDoc* hace referencia al documento actual de trabajo. Cuando un usuario crea o carga un nuevo documento, entonces este identificador hace referencia a él. Cuando un usuario cambia de documento de trabajo con el ratón (puede tener múltiples documentos abiertos al mismo tiempo), el objeto delegado de la ventana asociada al documento (que es el controlador de un documeto red de Petri: *PNController*), se encarga de enviarle un mensaje el objeto de *Controller*, indicándole que ahora el documento actual es el que a recibido el evento de ratón.

El identificador *inspector* hace referencia el objeto controlador del panel *Inspector* donde se muestran los artibutos principales de los objetos que componen la red de Petri (lugares, transiciones, arcos y la misma red de Petri). Debido a que la información que muestra este panel cambia dinámicamente conforme el usuario trabaja con la red de Petri, el objeto *PNController* mantienen una comunicación indirecta con este objeto, a través del objeto *COntroller*.

El identificador *currentGraphic* hace referencia a la clase que corresponde a la herramienta que selecciona el usuario cuando elige el elemento gráfico, en el panel *Tools*. Esta referencia a la clase se mantiene con el nombre del

icono que representa al objeto gráfico y el nombre de la clase. Esto es, si se selecciona en el panel *Tools*, el icono correspondiente a un objeto *Place*, el icono tiene el nombre *Place*, que también corresponde con el nombre de la clase del objeto que se va a agregar a la red de Petri. Esto permitirá a los usuarios que deseen incorporar nuevos objetos que se elijan desde el panel *Tools*, para utilizarse en la red de Petri, que sólo tengan que incorporar un botón con un icono con el nombre de la clase que van a agregar, en el panel *Tools*, e incorporar al proyecto *PetrA*, los archivos de la clase correspondiente. Esto es, *PetrA* reconoce automáticamente a la nueva clase y no se deben agregar líneas de código para reconocerla.

PetrA responde a los eventos que le envía el usuario a través del menú de la aplicación, con sus métodos *sender-target*, los cuales actúan de la siguiente manera:

newDoc Este método se ejecuta cuando el usuario selecciona la opción *New* del menú *File*. Cuando el objeto *Controller* recibe este mensaje, se crea un nuevo objeto de la clase *PNController* y el identificador *currentDoc* hace referencia a él. Y al objeto *PNInspectorController* se le manda esta información para que actualice la información que despliega.

openTheFile Este método se ejecuta cuando el usuario selecciona la opción *Open* del menú *File*. Cuando el objeto *Controller* recibe este mensaje, se despliega un panel *NSOpenPanel* para que el usuario pueda seleccionar el archivo que desea abrir (el nombre del archivo debe tener extensión *.pn*). Si el usuario selecciona un archivo, entonces se procede a crear e inicializar un objeto *PNController*, se actualiza la referencia del identificador *currentDoc*, y al nuevo objeto *PNController* se le envía el mensaje *readTheFile* con el nombre del archivo seleccionado como argumento, para que carge y despliegue el contenido de este archivo.

showInfoPanel, showInspectorPanel, showPreferencesPanel Estos métodos se ejecutan cuando el usuario selecciona las opciones *Info*, *Inspector* y *Preferences* del menú de la aplicación. Los hemos juntado porque los tres realizan la misma tarea: desplegar el contenido archivo de interfaz (*.nib*) para los páneles de información, del inspector y de preferencias.

setCurrentGraph Este método se ejecuta cuando el usuario selecciona

cualquier botón de herramientas (*Place, Transition*, etc.) del panel *Tools*. El objeto *Controller* ejecuta el siguiente código:

```
1  - (void)setCurrentGraph:(id)sender {
2    id cell;
3    NSString *className;
4    cell = [sender selectedCell];
5    if ((className = [[cell image] name]))
6         currentGraphic = NSClassFromString(className);
7    else
8         currentGraphic = nil;
9  }
```

En la línea 4 se oberva que en la variable *cell* se almacena la referencia al objeto que recibió el evento de ratón (los botones con los iconos de los objetos gráficos están agrupados en un objeto *NSMatrix*). La línea cinco muestra que se extrae el nombre de la imagen asociada al botón y se almacena el nombre en el objeto cadena referenciado por *className*. Después, en la línea 6 se actualiza el valor de la variable instancia *currentGraphic*, del objeto *Controller*, con el contructor de la clase del objeto seleccionado. Si no se encuentra una clase que corresponda con el nombre de la imagen, entonces *currentGraphic* tendrá el valor nulo (en el caso de Objective C:*nil*).

4.2.2 La clase PNController

Los objetos *PNController* son el control (del modelo MCV) de los documentos de una red de Petri. Como se aprecia en la figura 4.2, los objetos de la clase *PNController* mantienen una relación con la ventana asociada al documento, con un objeto de la clase *PNView* —la cual representa al objeto red de Petri, o al documento red de Petri— y con el objeto *Controller* de la aplicación.

Los objetos de esta clase se encargán de mandar los mensajes, provenientes del objeto de control, a la red de Petri, representada con la clase *PNView* y la ventana que la contiene.

Los objetos *PNController* contienen varias variables y metodos instancia, lo más importantes son:

```
@interface PNController : NSObject
{
    PNView   *pnet;
```

```
    id       window;
    NSString *docName;
    id       controller;
}
- (void)readTheFile:(id)file;
- (void)saveDocument:(id)sender;
- (void)saveDocumentAs:(id)sender;
- (void)cut:(id)sender;
- (void)print:(id)sender;
- (Class)currentGraphic;
// window's petri net delegate method
- (void)windowDidBecomeKey:(NSNotification *)notification;
@end
```

La variable *pnet* hace referencia a un objeto del tipo *PNView*[2], el cual es el objeto red de Petri.

La variable *window* hace referencia a la ventana asociada al objeto red de Petri. Se utiliza principalmente para desplegar el nombre del documento asociado a la red. Un atributo especial de este objeto es el nombre del documento que tiene la red de Petri, el cual se almacena en un objeto *NSString*, referenciado en la variable *docNmae*. Cuando un objeto *PNController*se inicializa, el nombre asociado que tiene es *UNTITLED* (y que se despliega en la barra de estado de la ventana).

Este objeto se encarga de manejar los métodos que le mandan el objeto *Controller* y el *FirstResponder* de *PetrA*, y de comunicar al objetos *PNView* asociado con el objeto *Controller*. También los objetos *PNController* se utilizan como los delegados de las ventanas a las que hacen referencia.

Estos métodos instancia trabajan de la siguiente manera:

readTheFile Este método se ejecuta cuando el usuario ha seleccionado la opción *Load* del menú *File*, originalmente el objeto *Controller* recibe el mensaje, y una vez que crea e inicializa el correspondiente objeto *PNController*, le manda este mensaje para que cargue el archivo que le manda como argumento. Cuando el objeto *PNController* recibe este mensaje, manda el mensaje *setPNMatrix* al su objeto *pnet*, con un objeto *PNMatrix* como parámetro, como se muestra a continuación:

```
1 - (void)readTheFile:(id)file {
2 [pnet setPNMatrix: [NSUnarchiver unarchiveObjectWithFile:file]];
3 }
```

[2]Se declara directamente como un apuntador, para permitirle al compilador realizar el ligado estático y aumentar el rendimiento de la aplicación

En la línea 3 de este listado aparece un llamado a la clase *NSArchiver* — esta clase es la encargada de proporcionar los mecanísmos para poder almacenar objetos hacia un archivo—y se le envía el mensaje *unarchiveObjectWithFile* con nombre del archivo como parámetro de entrada.

saveDocument Este mensaje le llega al objeto a través del *FirstResponder*. Cuando el documento no tiene asociado ningún nombre —es decir, la variable *docName* hace referencia a un valor nulo— se manda a ejecutar al método *saveDocumentAs* (ver líneas 2 y 6 del siguiente listado). Si se tiene asociado un nombre, entonces se procede a llamar a la clase *NSArchiver* y se le manda el mensaje *archiveRootObject: toFile* enviando como parametro de entrada al objeto *PNMatrix* (asociado al objeto *PNView*) y al nombre del archivo, para que se almacene el objeto principal de la red de Petri. Como se aprecia en la línea 3 del siguiente listado

```
1 - (void)saveDocument:(id)sender {
2    if (docName != nil) {
3          [NSArchiver archiveRootObject:[pnet pnMatrix]
                                  toFile:docName];
4          return;
5    }
6    [self saveDocumentAs: sender];
7 }
```

saveDocumentAs Este método le llega al objeto *PNController* por el mismo (ver desripción del método *saveDocument*) y por el usuario a través del *FirstResponder*. Este método asocia un nombre a la variable *docName* a través de un objeto *NSSavePanel*.

cut Este mensaje le llegua al objeto a través del *FirstResponder* y lo que se hace es reenviarlo al objeto *NSView*.

print Al igual que el mensaje *cut*, el mensaje *print* le llega al objeto a través del *FirstResponder* y lo que se hace es reenviarlo al objeto *NSView*.

currentGraphic Este mensaje le llega al objeto a través del objeto *PNView*, y lo que hace reenviarlo al objeto *Controller*.

windowDidBecomeKey Este mensaje lo envía una ventana a su delegado, en este caso al objeto *PNController*, su función consiste en enviar

el mensaje *setCurrentDocument* al objeto de control de *PetrA* (*Controller*), para que mantenga actualizado el valor del documento actual con el que trabaja y facilite la operación al *Inspector*.

4.2.3 La clase *PNView*

Esta clase es de las más importantes en *PetrA*, debido a que los objetos de esta clase representan gráficamente a la red de Petri. La clase *PNView* es una subclase de *NSView* del AppKit de Apple, lo cual le permite recibir eventos del usuario (a través del teclado y del ratón) y desplegar información en su área de dibujo. Como toda clase *View* del AppKit, *PNView* tiene relación con la ventana que lo contiene. En el modelo vista-control, los objetos *PNView* representa la vista y mantiene una relación con su objeto de control asociado (objetos *PNController*). Además esta clase mantiene una estrecha relación con la clase *PNMatrix*, que es la clase que representa las distintas interacciones entre la red de Petri en forma de matriz de incidencia.

Los objetos *PNView* se encargan de desplegar los elementos gráficos de la red de Petri (lugares, transiciones y arcos), por lo cual, tamién tiene una relación con la familia de clases *Figure* (de las cuales se desprenden *Place*, *Transition* y *Connection*).

PNView tiene declaradas varias variables y metodos de instancia, los más significativo se muestran a continuación:

```
@interface PNView : NSView
{
   PNMatrix  *matrixC; //Matriz de Incidencia
   ...
   id    controller;   //Controller Object Id
   id    lastGraph;    //Last Graph builded
   Class factory;      //Clase del nuevo objeto
   BOOL  running;
}
...
- (void)runPetriNet:(id)sender;
- (void)cut:(id)sender;
- (void)print:(id)sender;
...
- (void)mouseDown:(NSEvent *)event;
- (void)mouseUp:(NSEvent *)theEvent;
- (void)mouseDragged:(NSEvent *)theEvent;
- (void)drawRect:(NSRect)rect;
```

```
...
@end
```

La variable *matrixC* hace referencia a un objeto que representa la matriz de incidencia la red de Petri. Y de donde los objetos *PNView* obtienen información de la red de *Petri* (conexiones de los arcos, número de lugares, número de transiciones, etc). Al inicializar un objeto *PNView*, esta matriz no tiene ninguna información acerca de la red de Petri, sin embargo, conforme el usuario cargue de un archivo la red de Petri, o la edite gráficamente, este objeto irá almacenando toda la información necesaria, de hecho muchos métodos de la clase *PNView* están orientados atrabajar exclusivamente con la matriz de incidencia.

La variable *controller* hace referencia al objeto de control de la red de Petri (objeto de la clase *PNController*). Este objeto es el encargado de enviarle los mensajes provenientes del objeto de control de la red de Petri (*Controller*), principalmente para el manejo de la red de Petri como documento (salvar, obtener nueva matriz de incidencia, e imprimir) y para actualizar a la clase que se ha seleccionado en el panel *Tools* para que la instancia correspondiente se agregue a la red de Petri.

La variable *lastGraph* hace referencia al último objeto que se ha creado en la red de Petri durante los eventos de ratón: *mouseDown, mouseDragged* y *mouseUp*. Esto permite que el ultimo objeto creado en el área de dibujo del objeto *PNView* pueda recibir eventos del usuario para definir su origen y dimensión (en el caso de los objeto *Connection*).

La variable *factory* hace referencia al constructor de la clase del objeto que se agregará a la red de Petri. Esta referencia se obtiene del objeto *Controller*, a través del objeto *PNController*.

Finalmente la variable *running*, se utiliza para indicar si el documento red de *Petri* se está ejecutando. Ya que en caso de estarlo, no se pueden agregar o leiminar elementos de la red. Esta variable cobra importancia cuando se estan ejecutando los hilos de las transiciones, ya que una condición para que continúen su ejecución es que esta variable este encendida.

Los métodos instancia de los objetos *PNView* se utilizan, principalmente para la edición gráfica, para manejar a la red de Petri como un documento y para desplegar gráficamente a la red. Estos nétodos trabajan de la siguiente forma:

runPetriNet Este método lo envía el usuario directamente al objeto *PN-*

View a través de un botón en la ventana del documento. Cuando el objeto *PNView* recibe este mensaje le manda un mensaje al objeto *PNMatrix*, referenciado por la variable *matrixC*, el mensaje *runTransitions* para mandar a ejecutar a la red de Petri, y la variable *running* se activa. Si la variable *running* se encuentra activa cuando se manda a ejecutar este método, entonces se apaga esta bandera —con esta acción lo hilos de las transiciones terminan su ejecución, ya que sensan a esta variable cada vez que se ejecutan.

cut Este método lo envía el usuario a través del objeto *PNController*. Cuando se recibe este mensaje, el objeto *PNView* elimina los objetos gráficos que el usuario tenga seleccionados.

print Este método lo envía el usuario a través del objeto *PNController* o directamente con el botón que esta en la ventana del documento. Cuando se recibe este mensaje se revisan las dimensiones y orientación que se define en los documentos a través del panel *Page Setup*. Después se crea un objeto *PNRenderinView* donde se dibujará la red de Petri con la resolución de la impresora. Y se manda como argumento al controlador de impresora a través de la clase *NSPrintOperation*. Esta clase se encarga de establecer comunicación entre *PetrA* y el manejador de impresora para definir el tipo de impresora o si se redirecciona la salida a un archivo *pdf*.

mouseDown Este método lo envía directamente el usuario a través del ratón. La implantación de este método es un poco compleja, debido a las distintas respuestas que se esperan de una acción como esta. Básicamente este método se utiliza para agregar un nuevo elemento a la red de Petri —el que se haya seleccionado en el panel *Tools* y del cual se haga referencia a su constructor con la variable *factory*—, para seleccionar un objeto para una operación posterior (eliminar, ver sus atributos en el *Inspector*, o mover su posición), o para agregarlo a los objetos seleccionados.

Cuando se recibe este mensaje, lo primero que se hace es definir el punto en el área de dibujo donde se realizó el evento del usuario. Después se verifica que se tenga algún objeto fabrica seleccionado (esto garantiza que se tiene seleccionado una herramienta del panel *Tools*) y de que no

se este ejecutando la red. En caso de que se cumplan estas condiciones, se procede a crear el nuevo objeto y se incorpora a la red de Petri en caso de ser *Place* o *Transition*, se actualiza la información de la red de Petri en el *Inpector* y, finalmente, se les manda el mensaje *mouseDown* a estos objetos para que registren el punto donde se ha ejecutado el evento del usuario (esto les es de utilidad cuando se recibe el mensaje *mouseDragged*). Si el objeto que se ha creado es un objeto *Token* se debe verificar que el evento del ratón se haya hecho sobre un objeto *Place*, si es así, se le envía al objeto *Place* correpondiente el mensaje *addToken* y se elimina el objeto *Token* que se creo (por el momento los objetos Token son auxiliares en la edición de la red de Petri). Si el objeto que se ha creado es un objeto *Transition* se verifica que el evento se haya realizado sobre algún objeto *Transition* o *Place*, de ser así, se mantiene vivo el objeto, pero aún no se agrega a la red de Petri, pues se debe revisar que el objeto destino sea válido.

Si no se tiene referencia a algún constructor o si la red de Petri se está ejecutando, entonces se procede a verificar su algún objeto gráfico de la red de Petri contiene al punto donde se realizó el evento. Si algún objeto lo contiene, entonces se procede a seleccionarlo o deseleccionarlo según sea el caso. Si el número de objetos seleccionados es 1, entonces se despliega el *Inspector* con los atributos de este objeto, si el número de objeto seleccionados es mayor a 1 o cero, entonces el *Inspector* despliega la información general de la red de Petri.

Después de ejecutar cualquiera de los casos anteriores, se manda a redibujar el objeto *PNView*.

mouseUp Este método lo envía directamente el usuario a través del ratón. Cuando se recibe este mensaje, el objeto *PNView* identifica si se tiene por almacenar un objeto *Connection* (en caso de tenerlo se debe verificar que el objeto destino sea válido), o nó. Si el objeto que se ha creado (referenciado por *lastGraph*) es *Place* o *Transition*, se les envía el mensaje *mouseUp* para que actualicen algunas variables de instancia —que se utilizan en caso de haber respondido a un evento *mouseDragged*—. Si se comprueba que la referencia de *lastGraph* es de un objeto *Connection*, entonces se verifica que el objeto final que contiene al punto donde se realizó el evento *mouseDown* es válido para el objeto *Con-*

nection —es decir, si el objeto origen es *Place* entonces el destino es *Transition*, o viceversa—, entonces se procede a incorporar a este objeto a la matriz de incidencia asociada a *PNView* enviándole el mensaje *addConnection*. El objeto *PNMatrix* le regresa a *PNView* el resultado de la anexión del objeto *Connection* para enviarle la información el objeto *Controller* para que se pueda actualizar contenido del *Inspector*. Si el punto donde se realiza este evento no pertenece a ningún objeto válido entonces se procede a eliminar definitivamente al objeto *Connection*.

mouseDragged Este mensaje es enviado directamente por el usuario a través del ratón. Cuando el objeto recibe este mensaje lo que debe hacer es cambiar la posición de los objetos *Place* o *Transition* o dirigir a un nuevo objeto *Connection* hacia el objeto su final. Cuando se ejecuta este mensaje se debe tener en consideración de si se ha seleccionado o agregado una figura en el método *mouseDown*. Si la variable *factory* hace referencia a algún contructor, entonces el método *mouseDown* ha agregado un nuevo objeto gráfico y este método, en consecuencia, debe cambiar la posición del objeto (si es un objeto *Place* o *Transition*); si *lastGraph* hace referencia a un objeto *Connection* se debe cambiar su dimensión para dar el efecto de mover el arco hacia otro objeto en el área de dibujo. Si *factory* hace referencia a un valor nulo, entonces se tiene que se ha hecho una selección y se deben mover sólo los objetos *Place* o *Transition*, y por consiguiente, se deben redibujar los arcos que se conectan este objeto.

drawRect La manera de representar los objetos gráficos de la red de Petri, se realiza con ayuda de dos arreglos auxiliares, uno para todos los objetos gráficos, y otra para los objetos seleccionados. En ambos arreglos se colocan al inicio los arcos y después los lugares y transiciones.

4.2.4 La clase *PNMatrix*

Esta clase es el objeto principal de la red de Petri, pues mantiene los identificadores de los objetos gráficos que conforman la red de Petri. Aprovechando las características que ofrece la matriz de incidencia para representar a una red de Petri de una manera sencilla y compacta (ver sección 1.5.2), se eligió

manejar internamente a las redes de Petri como un objeto que trabaja como una matriz de incidencia, manteniendo en su primer renglón (renglón 0) identificadores a los objetos *Transition*, en la primera columna (columna 0) identidicadores a los objetos *Place*, y en las intersecciones de los renglones y columnas a los objetos arcos que únen a los lugares con las transiciones, como se puede apreciar en la figura 4.3.

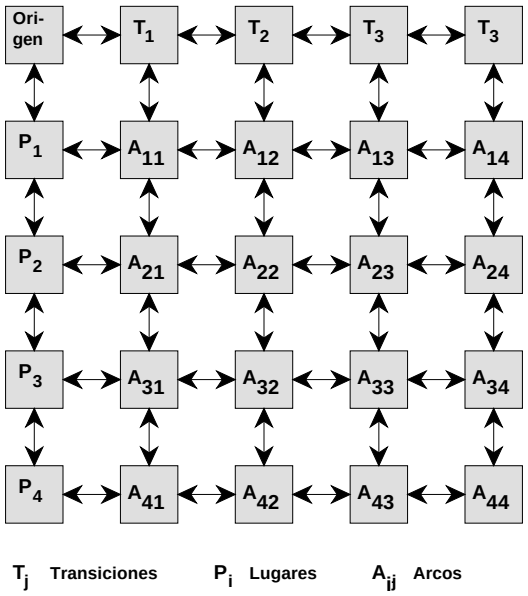

Figura 4.3: Los objetos de la clase *PNMatrix*, representan a una red de Petri, como una matriz de incidencia

La ventaja que representa manejar a una red de Petri de esta forma, sobre las tradicionales (lista de elementos, y matriz de conectividad, ver figura 4.4), es que la forma de matriz de incidencia es más compacta y mantiene toda la información necesaria de la red e Petri, por ejemplo: si se trabajara con las listas ligadas se deberían implantar mecanismos para conocer la funciones de entrada y salida, y con la matriz de conectividad se tendría el problema del espacio en memoria. Por otro lado la forma de matriz de incidencia permite que se pueda pensar a futuro en una implantación de redes jerárquicas.

Además la matriz de incidencia permite a los objetos *Place* y *Transition*, conocer cuales son los arcos con los que están conectados y obtener

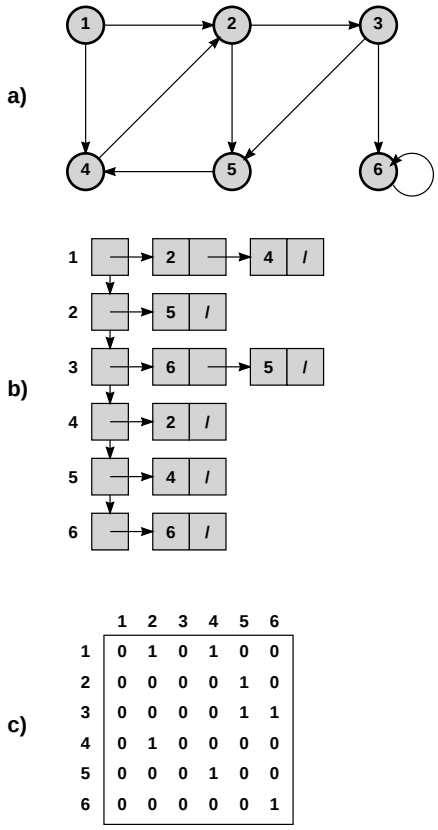

Figura 4.4: Representación tradicional de un grafo dirigido

información de estos para determinar, por ejemplo, cuales son los lugares de entrada y salida en una transición.

Si trabajamos con un enfoque de matriz de incidencia (figura 4.3)de elementos gráficos obtenemos:

- Los objetos *Place* se almacenan el la primera columna de la matriz (teniendo una lista ligada entre los diferentes lugares).
- Los objetos *Transition* se almacenan en el primer renglón de la matriz (teniendo una lista ligada entre las diferentes transiciones).
- Los objetos *Connection* se encuentran en los elementos que forman la intersección entre columnas (*Transitions*) y renglones (*places*).

Para implantar la clase *PNMatrix* se pensó en tener una clase padre que manejará a una matriz (la clase *Matrix*) y a la clase *PNMatrix* como una subclase que manejara específicamente tareas propias de la matriz de incidencia, como agregar transiciones, lugares, o enviar un mensaje a todo los objetos *Transition*, por mencionar algunos.

Las variables y métodos de instancia de la clase *PNMatrix* son:

```
@interface PNMatrix : Matrix
{
   int nPlaces;     //Numero de lugares
   int nTransitions;//Numero de transiciones
   int nLinks;      //Numero de ligas
}
- (void)addPlace: (id)newPlace;
- (void)addTransition: (id)newTransition;
- (void)deletePlaceInRow: (int)i;
- (void)deleteTransitionInColumn: (int)j;
- (result_add)addConnection: (id)newConnection;
- (id)placeCanResponseToEvent:(NSEvent *)theEvent;
- (id)transitionCanResponseToEvent:(NSEvent *)theEvent;
- (id)graphCanResponseToEvent:(NSEvent *)theEvent;
- (void)reDrawAllConnectionsInColumn: (int)j;
- (void)reDrawAllConnectionsInRow: (int)i;
- (void)runTransitions;
```

Las variables *nPlaces, nTransitions* y *nLinks* se utilizan para mantener un control sobre, como sus nombres lo indican, el número de lugares, renglones y conexiones, respectivamente.

Los métodos de instancia de los objetos *PNMatrix*. Los mesajes son enviados a estos objetos por los objetos *PNView*.

addPlace Este mensaje le llega al objeto por parte de un objeto *PNView*. Cuando se recibe este mensaje se agrega una nueva columna a la matriz y se hace referencia el objeto *Place* que se manda como parámetro de entrada en la primera columna del nuevo rengón.

addTransition Cuando se recibe este mensaje se agrega un nuevo rengón a la matriz y se hace referencia el objeto *Transition* que se manda como parámetro de entrada en el primer rengón de la nueva columna.

deletePlaceInRow Cuando se recibe este mensaje se borra el renglón especificado, lo que resulta en la eliminación de lugar y de todos sus arcos de entrada y salida.

deleteTransitionInColumn Cuando se recibe este mensaje se borra la columna especificado, lo que resulta en la eliminación de la transición y de todos sus arcos de entrada y salida.

addConnection Cuando se recibe este mensaje, se verifica la existencia de una referencia en la intersección de la transición y el lugar que contiene el objeto *Connection* que se manda como parámetro de entrada. Si no existe ninguna referencia, entonces se procede a hacer la referencia y se termina el trabajo. Si existe alguna referencia, entonces se verifica si el objeto existente mantiene la misma dirección que el nuevo objeto *Connection*, de ser así, entonces se aumenta el valor del peso del arco en el objeto existente. Si la dirección es contraria entre el objeto existente y el nuevo objeto *Connection*, entonces se disminuye el valor del peso del arco en el objeto existente. Si este peso llega a cero, entonces se procede a eliminar al objeto *Connection* existente.

placeCanResponseToEvent Se recorre la primera columna de la matriz, y a los objetos *Place* se les envía el mensaje *canResponseToEvent*.

transitionCanResponseToEvent Se recorre el primer rengón de la matriz y a los objetos *Transition* se les envía el mensaje *canResponseToEvent*.

graphCanResponseToEvent Cuando se recibe este mensaje, el objeto *PNMatrix* se manda los mensajes *placeCanResponseToEvent* y *canResponseToEvent*.

reDrawAllConnectionsInColumn Cuando se recibe este mensaje, se recorre toda la columna, indicada en el parámetro de entrada, y si hay objetos *Connection* se les envía el mensaje *reDraw*.

reDrawAllConnectionsInRow Cuando se recibe este mensaje, se recorre todo el rengón, indicado en el parámetro de entrada, y si hay objetos *Connection* se les envía el mensaje *reDraw*.

runTransitions Cuanlo se recibe este mensaje, se recorre el primer rengón de la matriz y a los objetos *Transition* se les envía el mensaje *run*.

4.2.5 La clase *Matrix*

Esta clase representa una matriz genérica de $[(nRows) \times (nColumns)]$ objetos. La clase *Matrix* representa a una matriz de objetos referenciados entre si. Esta clase contiene las primitivas para el manejo de una matriz, como el manejo de renglones y columnas, asi como de movimiento a elementos específicos de la matriz. Esta clase mantiene relación con la clase *Element*, la cual representa a los objetos elementos de la matriz. La implantación de esta clase tiene variable y métodos de instancia importantes, como se muestra en el siguiente listado:

```
@interface Matrix  : NSObject <NSCoding>
{
   unsigned int nRows;
   unsigned int nColumns;
   Element      *base;
   Element      *current;
}
...
- (Element *)gotoI:(int) iI J:(int) iJ;
- (void)addColumn;
- (void)addRow;
- (void)deleteColumn: (int) iJ;
- (void)deleteRow: (int) iI;
@end
```

Las variables *nRows* y *nColumns* mantienen el número de renglones y columnas. El número de renglones es de $n-1$, y el número de columnas es de $nColumns-1$, ya que los índices i, j de los elementos de la matriz están en los rangos: $i \in [0, nRows)$, y $j \in [0, nColumns)$.

La variable *base* se utliliza para mantener la referencia al primer objeto *Element* de la matriz (objeto en la posición $(0,0)$). La variable *current* se utiliza para moverse entre los diferentes elementos de la matriz.

Los métodos de los objetos *Matrix*, hacen operaciones muy simples de agregar y borrar renglones o columnas, y de posicionarse en algún elemento específico de la matriz.

4.2.6 La clase *Element*

Los objetos de esta clase son los que forman los nodos de la malla que forma la matriz. Estos objetos tiene identificadores a los elementos que se encuentren

en las cuatro posiciones adyacentes a un nodo. También tendrá un identificador para poder asociarle cualquier objeto, lo que genera que la matriz sea genérica. Un objeto *Matrix* se puede pensar como una malla cuadriculada de objetos *Element* unidos entre sí.

Las variables instancia de los objetos de esta clase son:

```
@interface Element  : NSObject <NSCoding>
{
   unsigned int posI;     //Posicion I en la matriz (renglon)
   unsigned int posJ;     //Posicion J en la matriz (columna)
   Element      *nextI;   //Siguiente elemento (renglon)
   Element      *forwardI;//Anterior elemento (renglon)
   Element      *nextJ;   //Siguiente elemento (columna)
   Element      *forwardJ;//Anterior elemento (columna)
   id           content;  //Contenido del elemento
}
...
@end
```

Como se aprecia en el listado anterior, las variables instancia de *Element* son *posI* y *posJ* que representan los índices i, j de posición de un elemento en una matriz. La variable *nextI*, hace referencia al siguiente elemento en el renglón. La variable *forwardI*, hace referencia al elemento anterior en el renglón. La variable *nextJ*, hace referencia al siguiente elemento en la columna. La variable *forwardJ*, hace referencia al elemento anterior en la columna. Y, finalmente, la variable *content* hace referencia al objeto que se desee almacenar en la matriz.

Esta clase mantiene una estrecha relación con elementos de la misma clase y, para el caso específico de *PetrA*, se mantiene una relación con la familia de clases derivada de la clase *Figure*.

4.2.7 La clase *Figure*

Esta es una clase genérica, y sirve de base para crear las clases de los objetos gráficos *Place*, *Transition* y *Connection*. Esta clase tiene implantados los mecanismos principales para manejar la parte gráfica y atender a los eventos de ratón del usuario. Esta clase copia algunas características de los objetos *View*, como son el manejo de un *frame*, para manejar su posición y tamaño respecto al view donde se dibuja, y un *bounds*, para establecer su propio

origen de coordenadas y tamaño. Sin embargo, esta clase tiene la gran diferencia con los objetos *View* en que no estan agrupados en una gerarquía *View*, lo que permite que si se hay dos o mas objetos encimados en el despliegue puedan decidir entre ellos cual es el objeto que recibe un evento de ratón.

La clase *Figure* mantiene una estrecha relación con el objeto *View* donde se despliega.

Las variables y métodos instancia de esta clase más importantes son:

```
@interface Figure  : NSObject <NSCoding>
{
    NSRect    bounds;
    NSRect    frame;
    id    superView;
    NSPoint       old;
    BOOL   selected;
}
...
- (NSImage *)drawFigure;
- (BOOL)canResponseToEvent:(NSEvent *)theEvent;
- (void)mouseDown:(NSEvent *)theEvent;
- (void)mouseUp:(NSEvent *)theEvent;
- (void)mouseDragged:(NSEvent *)theEvent;
- (tGraph)typeGraph;
...
@end
```

Las variables *bounds* y *frame* se utilizan para definir las dimensiones y origen respecto relativo y absoluto respectivamente. La variable *superView* hace referencia al objeto *PNView* donde se desplegará el objeto. La variable *old* se utiliza como auxiliar para desplazar la figura (actualizando los valores de la estructura *frame*) en los métodos que atienden los eventos del usuario. La variable *selected* se utiliza para determinar si la figura se encuentra seleccionada o no.

Los métodos se ejecutan como se describe a continuación:

drawFigure Este método no está implantado en esta clase, cada subclase lo debe implantar. Este método es parte de un protocolo que tienen los objetos de esta familia de clases y que utilizan los objetos *PNView* para desplegar a los objetos gráficos de la red de Petri.

canResponseToEvent Se verifica que el punto donde se realizó el evento del ratón esté en el *frame* de la figura, si está en esta área, se regresa un valor verdadero, y falso en caso contrario.

mouseDown Este método almacena en la variable *old* el punto donde se realizó el evento de usuario en el sistema de coordenadas del objeto *PNView*.

mouseUp Este método limpia el valor de la variable *old* para poder utilizarse en otro mensaje *mouseDown*

mouseDragged Este método actualiza la posición actual del objeto gráfico en su superview, es decir, modifica el origen de la variable *frame*.

typeGraph Este método regresa si la figura es una transición, lugar y arco.

4.2.8 La clase *Place*

Los objetos de esta clase representas los nodos lugar de la red de Petri, cuentan con un número de tokens. Su representación gráfica en el objeto *PNView* es un círculo.

Sus variables y métodos instancia más importantes son:

```
@interface Place : Figure
{
   int ntokens;
}
@end
- (NSImage *)drawFigure;
- (void)addToken;
- (void)setTokens:(int)n;
- (tGraph)typeGraph;
```

La variable *nTokens* establece el número de tokens asociado al lugar. Los métodos se ejecutan como se describe a continuación:

drawFigure Cuando se recibe este mensaje, el objeto *Place* dibuja un círculo en un objeto *NSBezierPath*. Si se tiene un número de tokens mayor a cero, entonces se procede a desplegarlos dentro del círculo que representa al lugar.

addToken Cuando se recibe este mensaje, el objeto incrementa en uno el valor de la variable *ntokens*.

setTokens Este meétodo, como el anterior, altera el valor de la variable *ntokens*, colocandole el valor que se le pase como argumento de entrada.

typeGraph Regresa que el objeto es del tipo *Place*.

4.2.9 La clase *Transition*

Los objetos de esta clase deben ejecutar acciones sobre los lugares de la red de Petri. Estos objetos tiene asociados un objetos NSThread, que se encargará de leer de sus lugares de entrada y arcos el número de tokens y pesos para verificar si está en posibilidades de disparar, en caso de estarlo, procederá a escribir en los lugares de salida el valor correspondiente. Esta clase utiliza una variable de clase (que hace referencia a un objeto *NSLock*) para evitar problemas de exclusión mútua entre los distintos hilos que se vana aejecutar en la aplicación.

```
@interface Transition : Figure
{
    BOOL active;
}
...
- (void)run;
- (void)execute: (id)connections;
- (NSImage *)drawFigure;
- (tGraph)typeGraph;
...
@end
```

La variable *active* indica si el usuario desea que la transición pueda ejecutarse. Con ayuda de esta variable, el usuario puede desactivar una transición desde el panel *Inspector*.

Los métodos se ejecutan como se describe a continuación:

run El mensaje *run* se hace desde el objeto *PNMatrix* cuando se ponde a ejecutar una red de Petri. Este método obtiene la lista de los lugares de entrada y salida, y después crea un objeto *NSThread*, el cual se encargará de ejecutar la transición. Esto se muestra en el siguiente listado:

```
1 - (void)run {
2    if (active) {
3        [self getConnection];
4        [NSThread detachNewThreadSelector: @selector(execute:)
                                  toTarget: self
                                withObject: ioConnections];
5    }
6}
```

En la linea 5, se manda a ejecutar el método *getConnection* para obtener la lista de los lugares de entrada y salida de la transición, esta información se almacena en el objeto *ioConnections* (el cual, se utiliza en la siguiente línea de código) La línea 6 manda a crear y ejecutar un objeto hilo (*NSThread*). Este hilo ejecuta el método *execute* de este mismo objeto, con el argumento *ioConnections*.

execute Este método lo ejecutan los hilos de cada transición, por lo que se debe tener cuidado ya que, como toda parte de código compartida, tiene secciones críticas que pueden causar problemas de exclusión mutua. Vale la pena revisar el código para explicarlo:

```
1  - (void)execute: (id)connections
2  {
3    NSAutoreleasePool *localPool=[[NSAutoreleasePool alloc] init];
4    while ([superView isRunning]) {
5        while (![myLock tryLock])
6            [NSThread sleepUntilDate:
7               [NSDate dateWithTimeIntervalSinceNow:0.7]];
8        // begin critical section
9        if ([self conditionsOK]) {
10           [self setCanRunTransition:YES];
11           [self eliminateInputs];
12           [self appendOutputs];
13           [[self superview] display];
14       }
15       // end critical section
16       [myLock unlock];
17       if ([self canRunTransition]) {
18           [self setCanRunTransition:NO];
19           [NSThread sleepUntilDate:
20              [NSDate dateWithTimeIntervalSinceNow:0.9]];
21       }
```

```
22   }
23   [localPool release];
24   [NSThread exit];
25   return;
26 }
```

Este método se ejecuta de la siguiente manera: primero se crea un área de memoria dedicada a almacenar objetos locales a cada hilo (linea 3). En la línea 4 inicia el ciclo que mantiene a los hilos en ejecución: mientras el objeto *PNView* mantenga encendida la bandera *running*, los hilos intentarán ejecutarse. Posteriormente, en la línea 5, se sensa el *lock* para entrar a la sección crítica, si el *lock* no está disponible, entonces el hilo debe dormirse, sin embargo los métodos de los objetos *NSthread* solo permiten que el hilo se duerma un determinado tiempo, se creen y ejecuten, y terminen su ejecución. Por lo que se manda a dormir un periodo de tiempo, si cuando despierte esta libre el *lock*, entonces podrá continuar su ejecución hacia la sección crítica. De no estar disponible el *lock*, el hilo se vuelve a dormir otra cantidad de tiempo. Si el hilo logra entrar a la sección crítica, en la línea 9, con el llamado al método *conditionsOK* se averigua si es posible ejecutar esta transición —se debe cumplir que cada lugar de entrada tenga mayor o igual número de token que el peso del arco que lo uno con la transición—. Si no se puede ejecutar, termina la ejecución de la sección crítica, libera el candado e intenta ejecutarse nuevamente. Si se cumplen las condiciones para ejecutar la transición, entonces se se eliminan los tokens de los lugares de entrada y se colocan lo tokens en los lugares de salida, evidentemente considerando el peso de los arcos—. Y finalmente, se manda a actualizar la imagen de la red de Petri. Una vez terminada la ejecución de la transición, se libera el candado y la transición queda inhabilitada un pequeño lapso de tiempo para permitir a otros transiciones, la oportunidad de ejecutarse.

drawFigure Con este método sólo se crea una figura rectangular en un cuadro donde se defiene el origen de coordenadas respectoa al *superView* y el tamaño de la transición.

typeGraph Este método regresa que el grafo ue se utiliza es del tipo *Transiton*.

4.2.10 La clase Connection

Esta clase tendrá asociada la representación gráfica del arco dirigido entre los nodos de la red. Los beneficios que obtenemos al tratar al arco un objeto es que el arco tendrá que manejar su valor de peso, y no dejar este atributo para la transición. Otra ventaja es que una vez que se conocen los nodos de entrada y salida podemos trazar una recta entre estos dos puntos.

La clase *Connection* tiene el atributos el peso del arco y los identificadores a los grafos de inicial y final. Sus métodos están orientados a desplegar la línea que une a ambos grafos, o, cuando recién se crea, a dar el efecto de la flecha que sigue al ratón para alcanzar su grafo final.

4.3 El manejo de archivos

Un aspecto importante en *PetrA* es que una red de Petri se maneja como un documento, y como tal, puede almacenarse en el disco. Para salvar una red de petri a disco, bastará con almacenar al objeto *PNMatrix*. Para grabar objetos a archivo se hacen llamados a las clases *NSArchiver* y *NSUnarchiver*. Sin embargo esto no basta, ya que esta clase desconoce que elementos de cada objeto deben almacenarse (inclusive, si los objetos que participan en una composición pueden almacenarse en el archivo). Para realizar esta labor se implanta el protocolo *NSCoding*. Para integrar este protocolo a una clase, se debe declarar en su archivo de interfaz, como se muestra a continuación.

```
@interface Matrix  : NSObject <NSCoding>
...
@end
```

En este código se muestra la definición del protocolo en la linea donde se define el nombre de la clase y la super clase.

Cuando se integra el protocolo a una clase, también se deben implantar los métodos *encodeWithCoder* e *initWithCoder* para codificar y decodificar un objeto. En estos métodos se describen los elementos de la instancia que serán codificados y decodificados. Por ejemplo veamos el caso de la implantación del protocolo *NSCoding* en la clase *Matrix*. En esta clase se codifican el número de renglones, columnas, y el identificador al elemento base o inicial. El identificador al elemento actual no se almacena ya que su valor puede inicializarse siempre con la referencia de la base.

```
- (void)encodeWithCoder:(NSCoder *)aCoder
{
    [aCoder encodeValuesOfObjCTypes:"ii@", &nRows, &nColumns, &base];
```

```
}

- (id)initWithCoder:(NSCoder *)aDecoder
{
    [aDecoder decodeValuesOfObjCTypes:"ii@", &nRows, &nColumns, &base];
    current = base;
    return self;
}
```

En este ejemplo se observa que los métodos *encodeWithCoder* y *initWithCoder*, tienen un objeto *NSCoder* como argumento de entrada, este objeto lo envían automáticamente las clase *NSArchiver* y *NSUnarchiver* cuando se le mandan los mensajes *archiveRootObject* (para almacenar un objeto), y *unarchiveObjectWithFile*, respectivamente.

Las clases que tienen implantado este protocolo son las que tienen relación con la clase *PNMatrix* (excepto la clase *PNView*). Estas clases son: *PNMatrix, Matrix, Element, Figure, Place, Transition* y *Connection.*

4.4 Comentarios finales

Se han creado varias clases para tener a *PetrA* como una aplicación, de OpenStep y Mac OS X, orientada a documentos. Se han incorporado a su implantación detalles técnicos para permitirle a futuros usuario la oportunidad de aumentar a *PetrA* de una manera sencilla. La clase que representa a la red de Petri a través de una matriz de incidencia es *PNMatrix.*

Capítulo 5

Uso de PetrA

El uso de la aplicación *PetrA*, permite al usuario modelar sistemas con redes de Petri, ver su comportamineto y hacer el analisis la red correspondiente. En este capítulo se explica el manejo de la aplicación *PetrA* y se muestran algunos ejemplos de su uso y funcionamiento.

5.1 La aplicación *PetrA*

PetrA es una aplicación que permite manejar redes de Petri como si fueran documentos, y editarlas gráficamente con ayuda del ratón. Para esto cuenta con varios elementos útiles como ventanas asociadas a cada documento, panel de herramientas, panel de preferencias y un inspector. El panel de herramientas permite al usuario seleccionar el objeto gráfico que empleará en la edición de la red de Petri. El panel de preferencias permite al usuario seleccionar algunas opciones generales de los objetos gráficos de la red de Petri, como el empleo de colores. El inspector permite obervar y manejar los atributos más importantes de los componentes gráficos de las redes de Petri (peso de los arcos, y número de tokens en un lugar, por mencionar algunos).

La ejecución de *PetrA* tiene las características de la aplicaciones de OpenStep y mac OS X, es decir, es un sistema con una interfaz de usuario gráfica con menúes y ventanas que responde a acciones del usuario. Como se aprecia en las figuras de este capítulo.

Al ejecutar *PetrA* se muestra su menú principal (en barra superior de la pantalla) y aparece el panel de herramientas. Como se puede apreciar en la

figura 5.1.

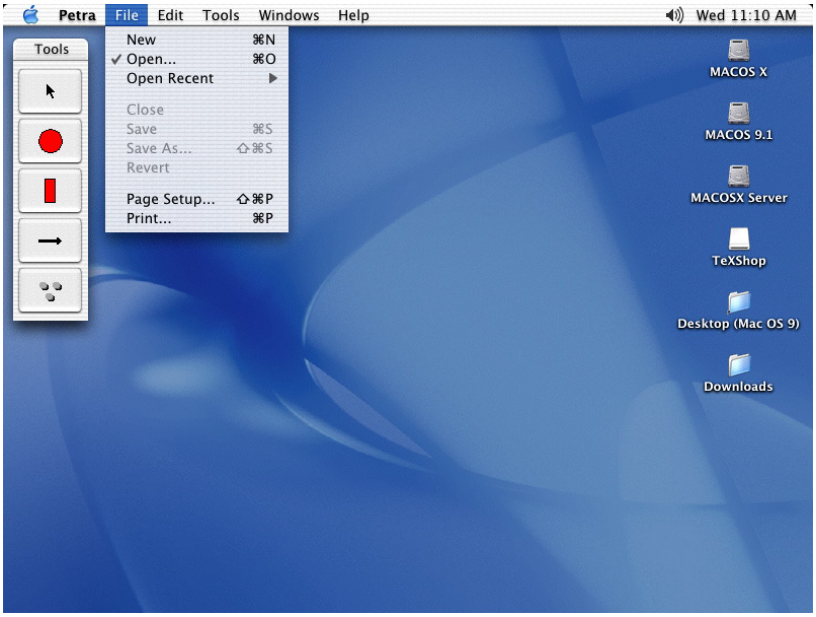

Figura 5.1: Ejecución de *PegtrA* en Mac OS X.

5.1.1 Manejo de documentos

PetrA maneja a las redes de Petri como documentos. Así, si se desea crear una nueva red de Petri, salvar, o recuperar habrá que utilizar las opciones *New*, *Save*, y *Open*, respectivamente, del menú *File* de *PetrA*. Los documentos de redes de Petri que maneja *PetrA* tienen la extensión *pn*.

Al crear un nuevo documento (opción *New* del menú *File*) el usuario obtiene una ventana con área de trabajo limpia donde podrá crear su modelo.

Cuando el usuario carga un documento (opción *Open* del menú *File*), se le muestra al usuario un panel para seleccionar el archivo que desea trabajar, de hecho los únicos archivos que se pueden abrir son los que tienen la extensión *pn*.

Si el usuario desea salvar su documento, entonces puede utlizar la opción *Save* o *Save As* (del menú *File*). Estas opciones se diferencian entre sí en lo siguiente. La opción *Save* guarda el documnento con su nombre actual —en

caso de que caresca de nombre (si aparece *UNTITLED* en la barra de estado la ventana del documento), procede a mostrar un panel para seleccionar el nombre del archivo y directorio donde se almacenará—. La opción *Save As* se utiliza para guardar el documento con otro nombre, por lo que siempre se mostrará el panel para seleccionar el nomnre y directorio de almacenamiento y se continúa trabajando con este nuevo archivo.

Imprimir una red de Petri

Si el usuario desea imprimir la red de Petri que esta utilizando bastará con que oprima el botón *Print* asociado a la ventana de su documento. (Como se aprecia en la figura 5.2)

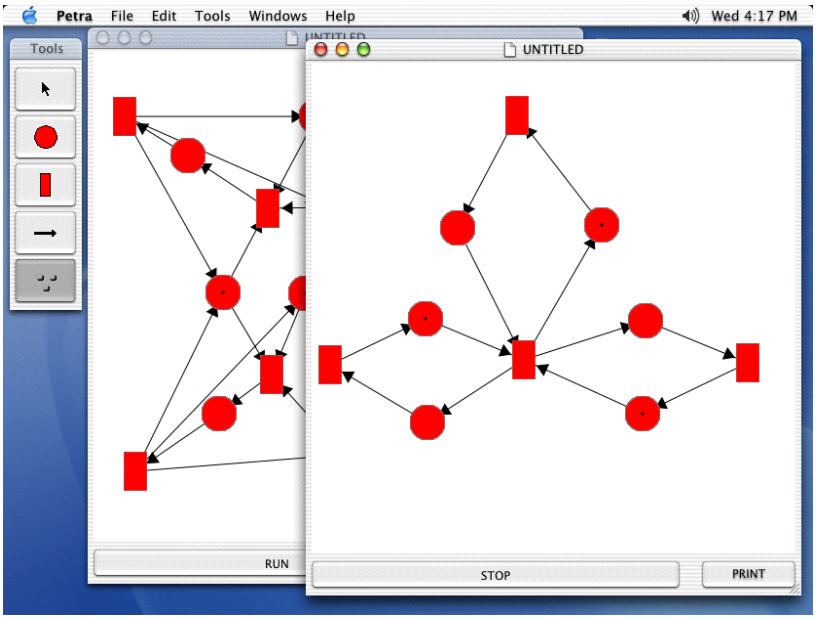

Figura 5.2: Manejo de múltiples documentos en *PegtrA* en Mac OS X.

5.1.2 Editando redes de Petri

Para editar una red de Petri (agregar, borrar, mover o cambiar atributos) se utilizan los paneles de herramientas, e inspector.

El panel de herramientas contienen iconos que muestran los elementos gráficos principales de una red de Petri: los lugares se representan con un círculo, las transiciones con un rectángulo, los arcos con una línea con una cabeza de flecha y los tokens con puntos. Además, el panel de herramientas proporcina un icono con la imagen del cursor que representa la opción de selección.

Para agregar un lugar a una red de Petri, basta con seleccionar el botón con el icono *Place* en el panel de herramientas (*Tools*), y después presionar el botón del ratón sobre el área de trabajo de la ventana de la red de Petri donde se desea colocar al lugar.

Si desea agregar tokens a los lugares se debe seleccionar el icono *Tokens* del panel de herramientas y después presionar el botón del ratón sobre un lugar en la red de Petri tantas veces como tokens se desee colocar. Si se presiona el botón del ratón sobre otro elemento de la red de Petri, no habrá ningún cambio en la red.

Si el usuario desea agregar una transición deberá seleccionar el botón con el icono *Transition* en el panel de herramientas y después apretar el botón en el área de trabajo de la red de Petri donde se desea colocar la transición.

Para agregar un arco se debe tener en consideración que las redes de Petri no permiten conexiones entre lugares, ni entre transiciones, es decir, unicamente se pueden conectar lugares con transiciones y transiciones con lugares. Para crear un arco se selecciona el icono *Arco* del panel de herramientas y se presiona el botón del ratón sobre algún lugar o transición de la red de Petri —si se presiona el botón del ratón sobre alguna otra área del documento no se produce ningún cambio en la red de Petri—. Después de presionar el botón del ratón sobre alguno de los elementos válidos se arrastra el ratón (con el botón presionado) hasta el lugar o transición destino —si se detecta que el origen y el destino son de la misma clase (lugar o transición) o si no existe destino, entonces se elimina el arco—.

PetrA maneja redes de Petri puras, es decir, redes de Petri que no tienen auto ciclos —arcos que conectan a un lugar A desde una transición B y a la transición B desde un lugar A, como se aprecia en la figura 2.5. Si se desea utilizar autociclos es conveninente cambiar a ciclos, como se puede muestra en la misma figura 2.5. Por default el peso de los arcos es uno. Si se desea cambiar el peso de los arcos se puede redibujar el arco tantas veces como el peso de arco se desee. Si se desea dismunuir el peso del arco se puede dibujar un arco en sentido inverso, esta operación decrementa en uno el peso del arco.

Si se traza un arco en sentido inverso y el valor del arco es uno, entonces se elimina el arco (ya que se crementa el peso a cero, lo cual equivale a no tener ningún arco).

Si se desea cambiar de posición alguno de los elementos gráficos, entonces se elige el icono *Cursor* en el panel de herramientas, y se presiona el botón del ratón sobre el objeto que se desea mover (lugar o transición), después se arrastra el cursor (sin dejar de presionar el botón del ratón) hasta la posición deseada. Una vez que se ha llegado a la nueva posición se suelta el botón del ratón. Los arcos conectados al objeto que se mueve se redibujan con el arrastre del ratón.

El icono *Cursor* representa selección —de hecho, el movimiento de un objeto se realiza con un objeto seleccionado—. Cuando se selecciona un objeto, esta cambia su color de acuerdo con el definido en el panel de preferencias, originalmente es gris. Al seleccionar un objeto se pueden cambiar sus atributos principales con ayuda del panel *Inspector*.

Eliminando objetos

Para eliminar un objeto se debe seleccionar —esto se realiza eligiendo el icono *Cursor* del panel de herramientas y después presionando el botón de ratón sobre el objeto deseado—, y después utilizar la opción *Cut* del menú *Edit*.

El manejo del panel *Inspector*

El *Inspector* se utiliza para revisar y editar los atributos principales de los objetos graficos que componen a la red de Petri (lugares, transiciones, arcos y la red de Petri en su totalidad). Cuando se selecciona un lugar se muestra el número de tokens. Se se ha seleccionado una transición se muestra una bandera que indica si la transición esta habilitada o no (por omisión todas las transiciones se encuentran activas). Si se tiene seleccionado un arco, entonces se muestra su peso.

Si no se tiene seleccionado ningún objeto gráfico, el *inspector* da la información general de la red de Petri (número de arcos y número de transiciones).

Además el *Inspector* permite al usuario cambiar el estado de los objetos gráficos. Si se desea cambiar el número de tokens de un lugar hay que modificar e valor del canpo de texto y presionar el botón *Apply* del *Inspector*. Si se desea cambiar el peso de un arco, entonces se coloca el valor deseado

Figura 5.3: Distintas vistas del panel *Inspector*

en el campo de texto y se presiona el botón *Apply* del *Inspector* para realizar esta acción. Para activar o desactivar una transición, se modifica el valor del switch *activo* del *Inspector*.

El panel de Preferencias

El panel *Preferences* permite al usuario editar algunas características generales de los documentos. Principalmente se hace enfoque en los colores asociados a los logares, transiciones, arcos y objetos seleccionados.

5.2 Ejemplos

Uno de los problemas clásicos que de prueban en este tipo de sistemas es el de los filósofos pensantes. En la figura 5.4 Los filósofos se modelan con dos

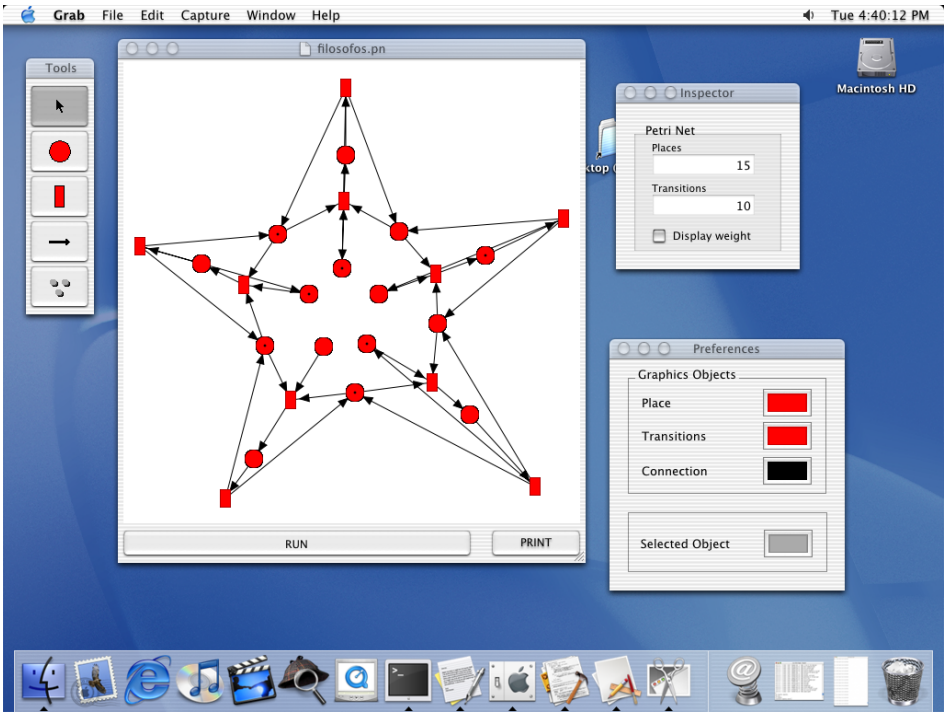

Figura 5.4: Modelado del problema de los filósofos pensantes en *PetrA*

estados: pensando p_i y comiendo c_i, Los lugares etiquetados con t_1, t_2, t_3, t_4 y t_5 representan los tenedores que comparten. Cuando se cambia de estado pensando a comiendo, se verifica que se tengan ambos tenedores y se procede a hacer el cambio. Si la transición que va a realizar el cambio no cuenta con tokens en los lugares que representan a los tenedores, entonces no se puede cambiar de estado y el filósofo sigue pensando. Si un filósofo está en estado comiendo y va a pasar al estado pensando, debe regresar los tenedores (colocar tokens en los lugares que representan a los tenedores) y se pasa al estado pensando.

La sección crítica del objeto *Transition*

La ejecución de *PetrA* implica que multiples hilos se ejecuten y compartan la información que se encuentra almacenada en los logares. Para evitar problemas de candados mortales, se implantó el uso de un candado (objeto *NSLock*, ver la clase *Transition* de la sección 2.2.1). Para verificar que la ejecución no causaría problemas se modelo en redes de Petri con *PetrA*. La figura 5.5 El lugar etiquetado con *th* indica cuanto hilos desean entrar en la sección

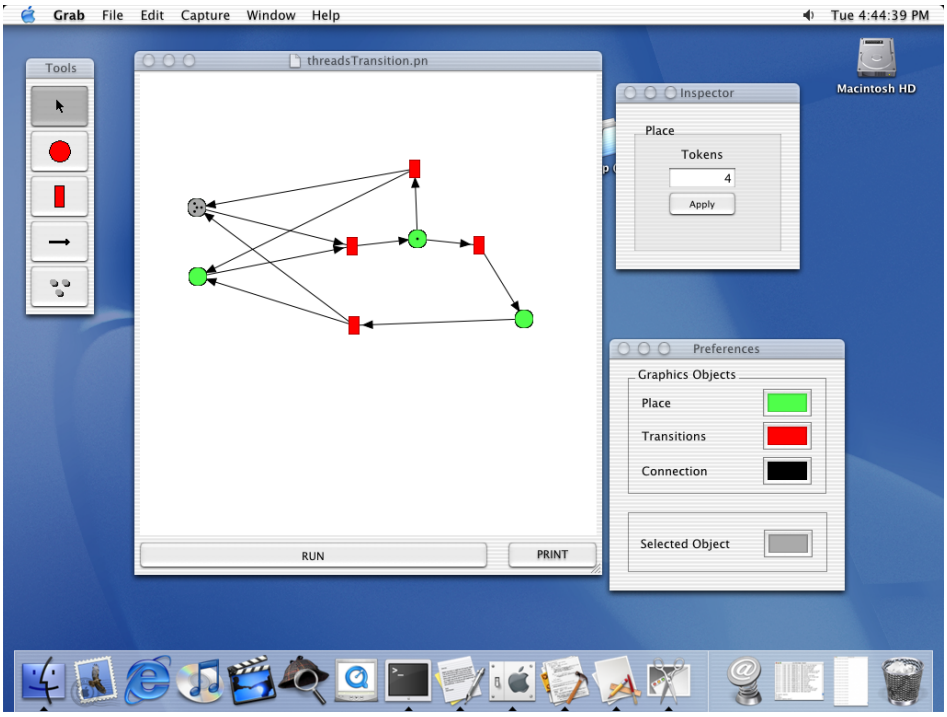

Figura 5.5: Modelado de el manejo de la sección crítica en la ejecucón de los objetos *Transition*.

crítica. El lugar etiquetado con *lock* indica si el candado esta encendido o nó. El lugar etiquetado con *can*, indica que el hilo esta verificando si puede ejecutarse (se cumplen las condiciones de entrada). El lugar etiquetado con *exe* indica que está ejecutandose la transición y se están cambiando los estado de la red de Petri.

Si el candado está disponible y, al menos, un hilo en *th*, entonces la

transición *begin* se ejcecuta. Entonces se manda un token al lugar *can*. Las transiciones *yes* y *no* pueden ejecutarse —estas transiciones indican si el hilo que tiene el control tiene las condiciones de entrada para ejecutarse—. Si se ejecuta *no*, entonces el hilo vuelve a esperar su turno en *th*, y se libera el candado. Si se ejecuta la transición *yes*, entonces el hilo pasa a ejecutar la transición en la red de Petri. Finalmente, cuando se ejecuta la transición *end*, el hilo regresa vuelve a esperar su turno y se libera el candado.

5.3 Comentarios Finales

PetrA ofrece al usuario varias elementos en su interfaz que le permiten al usuario trabajar con la red de Petri como un documento y editarla con el ratón. Los paneles auxiliares *Inspector* y *Preferences* se utilizan para definir o visualizar atributos de los elementos gráficos de la red de Petri. *PetrA* se ha utilizado para modelar su propio manejo de su sección crítica.

Capítulo 6

Posibles extensiones de PetrA

Uno de nuestros objetivo en este trabajo de tesis es que *PetrA* sea una base para manejar redes de Petri más complejas o con las restricciones que requiera el problema en que se apliquen. Este capítulo inicia con la descripción de un desarrollo sencillo para colocar restricciones a la red de Petri que maneja *PetrA*. tal que se manejen redes de Petri con capacidad finita. Y en las secciones restantes se proponen posibles soluciones para incorporar las redes de Petri estocásticas, coloreadas y jerárquicas.

6.1 Redes de Petri con capacidad finita

Una red de Petri limitada es una red de Petri que mantiene una restricción sobre el número de tokens que puede haber en cada lugar. Como es de imaginarse la solución obvia sería modificar la clase *Place*, incorporarle el atributo que establesca el límite, con sus respectivos métodos para modificar este atributo y para presentar su valor. Y, finalmente, modificar la clase *Transaction* para se contemple esta condición cuando se ejecuta su hilo de control.

Sin embargo podemos aprovechar la orientación a objetos para facilitarnos un poco la tarea y para ilustrar como agregar un nuevo objeto gráfico a la red de Petri.

La estrategia será la siguiente: crear una nueva clase de objeto *Place* llamada *PlaceLimited* con el nuevo atributo, insertarla en el panel *Tools*, para que el usuario pueda utilizarla. Después, se deberá modificar, inevitable-

mente, el código del método de instancia *conditionsOK*, de los objetos *Transition*.

1. El usuario debe tener una imagen que haga referencia a un lugar con un límite de tokens y que se llame *PlaceLimited*. Esta imagen debe insertarse en el archivo de interface principal de *PetrA* (*Petra.nib*) con ayuda de la aplicación *InterfaceBuilder*[1].

2. Una vez que la imagen está insertada en el archivo de interfaz, se coloca, en el botón que contiene la imagen *Place*, la imagen *PlaceLimited* como la imagen principal de ese botón. Y se salva el archivo de interfaz.

3. Ahora que se tiene la imagen, se procede a crear la clase *PlaceLimited* e integrarla al proyecto *PetrA* con ayuda de la aplicación *ProjectBuilder*[2]. La clase debe quedar como sigue:

```
#import "Place.h"
@interface PlaceLimited : Place
{
   int maxTokens;
}
- (id)initWithFrame:(NSRect)frameRect inView:(id)contentView;
- (int)maxTokens;
- (void)setMaxTokens:(int)n;
@end
```

   y su archivo de implantación quedaría:

```
#import "PlaceLimited.h"
@implementation PlaceLimited
- (id)initWithFrame:(NSRect)frameRect inView:(id)contentView {
   self=[super initWithFrame:frameRect: inView: contentView];
   return self;
}
- (int)maxTokens {
    return maxTokens;
}
- (void)setMaxTokens:(int)n {
    maxTokens=n;
```

[1] Una referencia para aprender el manejo del InterfaceBuilder se hace en [32, 32]

[2] Una buena referencia para aprender el manejo del ProjectBuilder se hace en [32, 32]

```
}
- (void)encodeWithCoder:(NSCoder *)aCoder {
    [super encodeWithCoder: aCoder];
    [aCoder encodeValuesOfObjCTypes: "i", &maxTokens];
}
- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super initWithCoder: aDecoder];
    [aDecoder decodeValuesOfObjCTypes: "i", &maxTokens];
    return self;
}
@end
```

Se puede observar que la clase *PlaceLimited*, es una subclase de *Place*, maneja un solo atributo (*maxTokens*), y tiene cinco métodos —uno de ellos es de inicialización, dos para accesar al atributo *macToken* y los métodos del protocolo $< NSCoding >$—.

4. Se procede a modificar a el método instancia *conditionsOK* del la clase *Transition*. En realidad se agrega código para verificar que el número de tokens de los lugares de salida, resultado de la ejecución de la transición, no excederán el límite de tokens.

```
- (BOOL)conditionsOK {
    int i, l, t, w;
    id cTemp;
    l = [[ioConnections inputs] count];
    for (i=0; i<l; i++) {
        cTemp = [[ioConnections inputs] objectAtIndex: i];
        w = [cTemp weight];
        t = [[cTemp startGraph] tokens];
        if (w>t) return NO;
    }
    // inicia parte agregada
    l = [[ioConnections outputs] count];
    for (i=0; i<l; i++) {
        cTemp = [[ioConnections outputs] objectAtIndex: i];
        w = [cTemp weight];
        w+= [[cTemp endGraph] tokens];
        t = [[cTemp endGraph] maxTokens];
        if (w>t) return NO;
    }
    // termina parte agregada
    return YES;
}
```

5. Se agrega un objeto *TextField*, al grupo de objetos gráficos que presentan los atributos de la clase *Place*, para que represente el límite superior de tokens de un lugar. Después se define su identificador en el la clase *PNInspectorController* y cuando se deba presentar o actualizar la información, se utiliza este identificador y los llamados *maxTokens* y *setMaxTokens* del objeto *PlaceLimited*.

6. Compilar y ejecutar.

Evidentemente el usuario debe tener conocimiento de las herramientas de desarrollo de OpenStep y Mac OS X para realizar estos pasos sin problemas. Sin embargo se tiene la confianza de que los desarrolladores futuros comprendan que la programación en esta plataforma de desarrollo es poderosa y muy conveniente.

6.2 Redes de Petri con tiempo

Las redes de Petri contiempo incorporan en valor del tiempo en su ejecución. Este valor toma importancia cuando se dispara una transición, entonces se toman los tokens de los lugares de entrada y después la transición deja pasar un período de tiempo (en el que se puede ejecutar alguna otra transición) y después continuar con la ejecución arrojando los tokens resultantes.

Para implantar el manejo de una red de Petri con tiempo hay que agregar un valor de tiempo como atributo de la transición, una vez que se ejecute el hilo asociado a ésta, se manda a dormir el tiempo deseado, y después se continúa la ejecución.

Se puede agregar una subclase de *Transition* con el atributo *time*, como el valor para dormir al hilo. El proceso de agregar esta clase al proyecto *PetrA* es muy similar a como se incorporaron las redes de Petri limitadas, ya que se debe incorporar una imagen con el nombre de la clase, y modificar la parte del inspector para que muestre los nuevos atributos del objeto.

La ejecución del hilo de la transición es el la que incorpora el tiempo como parte dinámica de la red. Así, el código de la sección crítica quedaría:

```
1 - (void)execute: (id)connections {
2  NSAutoreleasePool *localPool=[[NSAutoreleasePool alloc] init];
3    while ([superView isRunning]) {
4       while (![myLock tryLock])
```

```
5             sleep();
6       if ([self conditionsOK]) {
7           [self setCanRunTransition:YES];
8           [self eliminateInputs];
9           [myLock unlock];
10          yield();
11          [NSThread sleepUntilDate:
                [NSDate dateWithTimeIntervalSinceNow:[self time]]];
12          while (![myLock tryLock])
13                sleep();
14          [self appendOutputs];
15          [[self superview] display];
16      }
17      [myLock unlock];
18   }
19   [localPool release];
20   [NSThread exit];
21   return;
22 }
```

En este listado se observa, en las líneas 9, 10 y 11, que una vez que el hilo, que ha logrado entrar en la selcción crítica, ha eliminado los tokens de las entradas, libera el candado (para que otro hilo pueda ejecutar la el disparo de la transición a la que está asociado) y se duerme el lapso de tiempo que tiene relacionado. Después que este hilo a despertado, intenta entrar nuevamente a la sección crítica para escribir los tokens de sus lugares de salida.

6.3 Redes de Petri jerárquicas

Un problema común de las redes de Petri, es su tamaño, ya que para sistemas complejos, el modelo crece demasiado. Una estrategia que se ha utilizado exitósamente es agrupar secciones de la red de Petri en subredes. Al hacer este agrupamiento la nueva red de Petri se presenta como una entidad con entradas y con salidas, por lo que se maneja como si fuera una transición. Por lo que, si se tiene una subred de Petri, se debe almacenar en una rengón del objeto *PNMatrix*. Esto también nos indica que las subredes de Petri están comunicados con los demás elementos por medio de transiciones.

Algo que es deseable al manejar este tipo de redes de Petri, es que las subredes tengan su propia ventana y área de trabajo, así como de poder

visualizar los tokens que se transmiten entre las distintas redes que forman el modelo principal.

Para implantar este modelo, hay que reutilizar elementos que ya se tienen como el objeto *PNView, PNMatrix* y asi. Sin embargo, la ejecución de una subred debe contemplar la posibilidad de poder tomar información de objetos que están fuera de la misma.

6.4 Redes de Petri coloreadas

Las redes de Petri coloreadas permiten que el usuario trabaje con distitas marcas al mismo tiempo sobre la red de Petri. Para poder implantarlas se debe agregar una subclase *Place* que tenga un arreglo de enteros. Este arreglo debe ser del tamaño del número de colores con lo que trabaje la red de Petri. y en cada localidad se almacenará el número de tokens de algún color. También se deberá crear una subclase de *Token* que tenga el atributo color (tal vez en este caso sea más fácil aumentar este atributo y sus métodos para eccesarlo). Sin embargo se deben realizar tres modificaciones al código de la aplicación:

1. Modificar el método *mouseDown* de la clase *PNView* para que agrege el objeto Token a la subclase de *Place* que se agregará (evidentemente con el fin de establecer la posición en el arreglo donde estará.

2. Modificar el método *execute* de la clase *Transition* para que los hilos se bloqueen cuando intenten modificar algún color de *Token* que alguna otra transisión esté ejecutando.

3. Agregar mecanismos en los distintos archivos de interfaz para que ele usuario pueda manipular los tikens de colores en la red.

6.4.1 Comentarios Finales

Para poder realizar extensiones a *PetrA* es recomendable que el usuario conosca las herramientas de desarrollo de OpenStep y Mac OS X, además de su filosofía de desarrollo. En algunos casos, como en el de las redes de Petri de capacidad finita y las redes de Petri con tiempo las cosas se facilitan mucho elaborando una subclase. Sin embargo, es claro que también se

deben conocer varios detalles técnicos de las clases de *PetrA* para que las extensiones sean agregadas exitósamente.

Capítulo 7

Conclusiones y perspectivas

Durante el desarrollo de *PetrA* se han aprendido muchas cosas referentes a redes de Petri, como son los mecanismos de modelación, de implantación y las perspectivas que tienen como herramienta de verificación. Además hemos captado nuevos conocimientos referentes al diseño orientado a objetos y su implantación, particularmente en Mac OS X. *PetrA* ha sufrido varios cambios de diseño e implantación durante su desarrollo, ya que se obervó que varias ideas iniciales no ofrecían un rendimiento y comportamiento aceptables para la aplicación. Por otro lado, cuando se inició el desarrollo de *PetrA*, no se contaban con muchas bibliotecas de objetos que ahora se han incorporado en las nuevas versiones de Mac OS X y del *Yellow Box*, por lo que se escribieron varias clases que hoy podrían considerarse obsoletas o innecesarias, sin embargo en su momento fueron una parte operativa que debía ser implantada.

En este capítulo final, se mencionan las conclusiones de este trabajo y las mejoras que pueden agregarsele a *PetrA* de manera inmediata y a largo plazo.

7.1 Conclusiones

Las redes de Petri representan una herramienta muy completa, tanto para el modelado de sistemas, como para establecer una buena confiabilidad de software en sus diferentes etapas de desarrollo y mantenimiento.

Es posible tener un sistema base que maneje redes de Petri y que tenga capacidad de expansión para poder manejar redes de Petri extendidas, sin

embargo se requiere que el usuario conosca la filosofía de desarrollo de este tipo de ambientes gráficos, y de algunos detalles técnicos de la redes de Petri, ya que por la complejidad de las mismas se requieren hacer cambios en varios elementos de control interno. Sin embargo esta tarea es menor que tener que escribir una aplicación dedicada exclusivamente al manejo de un tipo de red de Petri específico.

Por otro lado la aplicación contiene algunos elementos técnicos que valen la pena señalarse:

- Contiene un esquema orientado a objetos con un interfaces bien definidas para poder realizar las extensiones.
- Contiene técnicas para agregar nuevas clases sin que el usuario tenga que escribir mucho código en el controlador de la aplicación. Es decir, no tiene que agregar código *Tools*, y después seleccionar la nueva clase a la que pertenece el objeto que se desea agregar.
- Contiene hilos de control asociados a cada transición con el propósito de dejar que el Mac OS X se encarge de la selección de la transición que va a dispararse. Otras características favorables de este enfoque es que la implantación de las redes de Petri estocásticas es inmediata y, finalmente, los diferentes hilos permiten que el paso a un esquema paralelo y distribuido sea natural.
- El manejo de un objeto Matriz de Incidencia permite un buen manejo de varios tipos de redes de Petri extendidas.

Una conclusión final es que los ambientes de desarrollo gráficos (como Mac OS X y OpenStep), con su arquitectura de sistema operativo, sus aplicaciones y bibliotecas de objetos (o *frameworks*), permiten realizar aplicaciones orientadas a objetos de una manera fácil y rápida. Todo depende de las capacidades de los desarrolladores para poder aprovechar estas facilidades.

7.2 Perspectivas

La aplicación *PetrA* ha alcanzado un grado de operatividad y estabilidad adecuados, sin embargo, aún hay mucho trabajo que hacer. En esta sección se enumeran las mejoras que deben realizarsele a *PetrA* para que obtenga

mejor desempeño y presentación. Ahora solo se espera que este sea el inicio de una herramienta de modelado más poderoza.

Por otro lado, se confía en que la plataforma de desarrollo (Mac OS X), junto con sus herramientas y bibliotecas de objetos— continúe en uso y desarrollo para que *PetrA* siga vigente y permita su portabilidad a plataformas futuras.

Las tareas inmediatas o de mediano plazo que se le deben incorporar son:

- Incorporar el uso de las bibliotecas de objetos que ha desarrollado *Apple* para el manejo de aplicaciones orientadas a documentos, lo cual facilitará el desarrollo posterior de la aplicación. Esto implica la desaparición de la clase *Controller*, el cambio en algunas clases como *PNController* (la cual pasaría a ser una clase *DocumentController*) y de *PNView* (la cual pasaría a ser una clase *Document*), y la aparición de dos clases de control *PNToolsController* y *PNPreferencesController* (clases de objetos de control para los paneles *Tools* y *Preferences*, respectivamente, ya que originalmente el objeto de la clase *Controller* se encarga de su manejo.

- Incorporarle mecanismo de análisis. Este es el trabajo que más debe tenerse en consideración, lamentablemente el tiempo que se dedicó al desarrollo de *PetrA*, no nos alcanzó para incorporar algún tipo de técnica. Sin embargo, quedan claras dos cosas: primero debe incorporarse como un hilo de ejecución extra, u otra tarea, ya que las técnicas de análisis de redes de Petri grandes consumen grandes cantidades de tiempo de CPU y memoria. Segundo, como los mecanísmos de análsis varían se debe contemplar que los mecanismos para el análisis sean delegados del objetos PNMatrix, lo que permite flexibilidad de selección las *muy diferentes* técnicas de análisis que hay con los distintos tipos de redes de Petri.

- Reconsiderar el uso de los *NSThread* para el manejo dinámico de las transiciones, ya que el manejo de estos objetos es muy limitado para el usuario (pues solo tiene metodos para dar de alta el hilo, dormirlo por un determinado tiempo, y para terminarlo, y hace falta mensajes como *yield*, para dormir un hilo de manera indeterminada y permitir que otro hilo se ejecute) y, además, se ha observado que el rendimiento

decae. Se podría considerar la incorporación de un objeto *NSTimed-Screen*, sin embaro esto complicaría la implantación de las redes de Petri estocásticas. Se puede incorporar alguna de estas soluciones:

1. Utilizar los hilos de Posix, ya que la documentación indica que los NSThread están implantados sobre los hilos de Posix.
2. Crear una clase Thread que permita al usuario un mayor control sobre estas unidades dinámicas.

- Incorporar mejores directivas a *PNMatrix* para el controlar el despliegue de los objetos gráficos en el *PNView*, ya que se siguen utilizando arreglos para organizar la lista de objetos desplegados y de objetos seleccionados. Este punto debe tratarse con cuidado, ya que se podría pensar en erradicar el uso de los objetos *PNMatrix*, sin embargo debe recordarse que estos objetos fueron diseñados para incorporar el uso de redes de Petri jerárquicas.

- Mejorar el manejo del área de dibujo de *PetrA*, para que el usuario pueda tener mayor control en el diseño de sus modelos. Por el momento sólo nos hemos dedicado al manejo de un tamaño estático, lo cual puede ser un grave inconveniente, pues es bien sabido que si el modelo con redes de Petri de un sistema, es muy complicado, el tamaño de la red crece.

- Incorporar mecanísmos para realizar una distribución automática de la red (distribución topológica). Esta opción será de gran utilidad si se incorporan el manejo de un formato ASCII para representar a una red de Petri. Evidentemente, este formato deberá ser en forma de una matriz de conectividad.

- Incorporar al *Inpector* opciones para modificar el tamaño de los objetos de la red de Petri, así, como su orientación. Lo cual dará al usuario distintas opciones de trabajar con modelos amplios.

- Permitir al usuario un manejo de conecciones más flexibles, esto es, permitirle al usuario integrarle puntos de control para que las coneccione se representen como curvas. De esta forma, se facilitaría la visualización de modelos complejos.

Entre las implantaciones que se deben realizar a largo plazo en *PetrA* se encuentrán:

- Incorporar el manejo de distintas redes de *Petri*, como se ha mencionado en el capítulo 4 de este trabajo de tesis. Como son: redes de Petri con capacidad finita, redes de Petri con tiempo, jerárquicas y coloreadas.
- Incorporar una jerarquía de clases para el permitir distintos métodos de análisis, y los cuales no choquen entre las distintas redes de Petri que se pretenden manejar.
- Incorporar el manejo de distintos iconos para representar lugares, ya que muchas extensiones de redes de Petri representan a los lugares con diferentes figuras, lo que ayuda al usuario a entender mejor el modelo del sistema.
- Incorporar el uso de un lenguaje de modelación, que en este caso puede ser REC, como parte del lenguaje asociado a un arco.

Bibliografía

[1] J.L. Peterson, "Petri Nets Theory and The Modeling of Systems", Prentice-Hall, N.J., 1981.

[2] T. Murata, "Petri Nets: Properties, analisys and applications", Proc. IEEE, vol 77, pp. 541-579, Apr. 1989.

[3] Mu Der Jeng and Frank DiCesare, "A Review of Synthesis for Petri Nets with Applications to Automated Manufacturing Systems", IEEE Transc. on systems, man, and cybernetics, vol 23, no.1, January/February 1993.

[4] Fred D.J. Browden; "Modelling Time in Petri Nets"; July, 1996.

[5] Fred D.J. Browden; "Role-Based Extended Petri Net Modelsand their Applications"; International Congress on Modelling and Simulation, 1995, Newcastle, Australia.

[6] Holger Giese, Jörg Graf and Guido Wirtz; "Modeling Sofware Systems with Object Coordination Nets"¡ Intitut für Informatik; Westfälische Wilhelms-Universität, Germany, 1998.

[7] Stefan Schöf, Michael Sonnenschein, Ralf Wietin, "High-level Modeling with THORNs"; Oldenbuger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge- und Systeme (OFFIS); Escherweg 2. D-26121 Oldenburg (Germany).

[8] Patrick Lam; "A Petri Net Simulator in Java – Design document", February, 1999, http://www.sable.mcgill.ca/~plam/petri,

[9] Wass.M.; "Predator-A Hierarchical Petri Net Editor"; MSc Degree Thesis, Departement od Computing, Imperial College of Science, Technology and Medicine; University of London; September 2001.

[10] K. Jensen, "A Brief Introduction to Coloured Petri Nets", Computer Science Department, University of Aarthus, Denmark.

[11] K. Jensen, "An Introduction to the Theorical Aspects of Coloured Petri Nets", In: J.W. de Bakker W.-P. de Roever G. Rozenberg (eds.): A Decade of Concurrency, Lecture in Computer Science Vol. 803, pp. 230-272, Springer-Verlag, 1994.

[12] E. Clarke, O. Grumberg, and D. Long; "Verifications Tools for Finite-State Concurrent Systems', In: J.W. de Bakker, W.-P. de Roever G. Rozenberg (eds.): A Decade of Concurrency, Lecture Notes in Computer Science Vol. 803, pp. 124-175, Springer-Verlag, 1994.

[13] Bohuslav Krěna, Tomáš Vojnar; "Type Analysis in Object-Oriented Petri Nets; Department of Computer Science and Engineering, FEECS, Technical University of Brno, Czech Republic.

[14] Charles Lakos; "On the Abstraction of Coloured Petri Nets"; Proceedings of 18th International Conference on the Aplications and Theory of Petri Nets 1997, Lecture Notes in Computer Science 1248, Springer-Verlg, pp. 42-61, 1997.

[15] C.A. Petri "*Forgotten* Topics of Net Theory"; Advances in Petri Nets 1986-Part 2, W. Brauer, W. Reising, and G. Rozenberg (eds.), Lecture Notes in Computer Science 255, Springer-Verlag, 1996.

[16] C. Ramchandani; "Analysis of Asynchronous Concurrent Systems"; Thesis Degree of Doctor of Philosophy, Department of Electrical Engineering, Massachusetts Institute of Technology; February 1974.

[17] J. Wang; "Timed Petri Nets, Theory and Aplication"; Kluwer Academic Publishers; 1998.

[18] M. Ajmone Marsan, A. Bobbio, and S. Donatelli; "Petri Nets in Performance Analysis: An Introduction"; Lectures on Petri Nets I: Basic

Models, W. Reising, and G. Rozenberg (eds.), Lecture Notes in Computer Science 1491, Springer-Verlag, 1998.

[19] C. Sibertin-Blanc; "A Client-Server Protocol for the Composition of Petri Nets"; Proceedings of 14th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science 691, pp. 377-396, Springer-Verlag, 1993.

[20] J. Parrow; "Interaction Diagrams"; In: J.W. de Bakker, W.-P. de Roever G. Rozenberg (eds.): A Decade of Concurrency, Lecture Notes in Computer Science Vol. 803, pp. 477-529, Springer-Verlag, 1994.

[21] Wil c.d. Aalst; "The Applications of Petri Nets to Workflow Management"; Department of Mathematics and Computing Science, Eindhoven University of Technology; `http://wwwis.win.tue.nl/~wsinwa/jcsc/jcsc.html`; Nov. 1997.

[22] Jörg Desel and Wolfgang Reisig; "Place/Transition Petri Nets"; Lectures on Petri Nets I: Basic Models, W. Reising, and G. Rozenberg (eds.), Lecture Notes in Computer Science 1491, Springer-Verlag, 1998.

[23] D. Leu, M. Silva, J.M. Colom, and T. Murata; "Interrelationships amoung various concepts of fairness for Petri nets"; in Proc. 31th. Midwest Symposium Circuits and Systems", 1988.

[24] P.S. Thiagarajan and K. Voss, "A fresh look at free choice nets", Inform. Contr. Vol 61, no. 2, pp. 85-113, May 1984.

[25] F.E. Hohn, "Elementary Matrix Algebra", Macmillan, Ney York, 1958.

[26] Johnsonbaugh and T. Murata, "Petri nets and marked graph-mathematical model of concurrent computation"; The American Math. Monthly, vol. 89, no. 8, pp.552-566, Oct. 1982.

[27] P. Huber, K. Jensen, R.M. Shapiro; "Hierarchies in coloured Petri nets", Advances in Petri Nets 1990, Lecture Notes in Computer Science 483, Springer-Verlag, 1990.

[28] K. Jensen; "Coloured Petri nets: A hight level languaje for system design and analysis", Advances in Petri Nets 1990, Lecture Notes in Computer Science 483, Springer-Verlag, 1990.

[29] G. Cinsneros; "Configurable REC"; ACM SIGPLAN Notices, Vol. 59, No. 5, May 1994.

[30] `http://www.daimi.au.dk/PetriNets/tools/db.html`

[31] `http://www.salon.com/tech/col/garf/2001/01/08/bad_java/index1.html`

[32] Simson L. Garfinkel, Michael K. Mahoney; "NeXTSTEP Programming, Step one: Object-Oriented Applications"; TELOS, Springer-Verlag, 1993.

[33] Manuales de referencia de *Cocoa*; *Apple* Inc. 2000.

[34] James Martin, James J. Odell; "Métodos Orientados a Objetos, Consideraciones prácticas"; Prentice Hall, 1997.

[35] Noriega Ponce, Alfonzo; "Un ambiente para el desarrollo de controladores lgícos programables en computadoras personales"; Tesis M.C, CINVESTAV, Ing. Eléctrica, México D.F., 1992.

[36] Valdemar González Avila, "Simulación con Redes de Petri", Tesis de M.C., CINVESTAV, Ing. Eléctrica, CINVESTAV, 1993, México, D.F.

[37] G.J. Holzmann; "Design and Validation of Computer Protocols"; Prentice Hall, 1991.

[38] H.J. Schneider and A.I. Wasserman; "Automated Tools for Information Systems Designs", Edit. North Holland, 1991.