UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

UN SISTEMA PARA MANEJAR REDES DE PETRI, BASADO EN UN AMBIENTE DE PROGRAMACION ORIENTADO A OBJETOS

T E S I S

QUE PARA OBTENER EL TITULO DE:

A C T U A R I O

P R E S E N T A :

MAURICIO SOLIS GRANADOS

D I R E C T O R :
M. EN C. JOSE ANTONIO DELGADO VILLEGAS

A S E S O R A :
MAT. GUADALUPE IBARGÜENGOITIA GONZALEZ

MEXICO, D. F. SEPTIEMBRE DE 1990

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

I N D I C E

INTRODUCCION i

CAPITULO I LAS REDES DE PETRI Y EL MODELADO DE SISTEMAS . . . 1

I.1 Modelado. 1

I.2 Características de los sistemas. 1

I.3 Desarrollo inicial de las Redes de Petri (RP). 2

I.4 Teorías pura y aplicada de RP. 3

I.5 Aplicación de la teoría de las RP. 3

I.6 Definiciones básicas de RP. 3
- I.6.1 Estructura de una RP. 4
- I.6.2 Gráficas de RP. 5
- I.6.3 Marcajes de RP. 8
- I.6.4 Reglas de ejecución de RP. 9
- I.6.5 Espacio de estados de RP. 11

I.7 Modelado con RP. 14
- I.7.1 Eventos y condiciones. 14
- I.7.2 Concurrencia y paralelismo. 16
- I.7.3 Modelado de hardware. 18
  - I.7.3.1 Computadoras acanaladas (pipelined). 18
  - I.7.3.2 Unidades funcionales múltiples. 21
- I.7.4 Modelado de software. 23
  - I.7.4.1 Diagramas de flujo. 24
  - I.7.4.2 Paralelismo. 26
  - I.7.4.3 El problema de la exclusión mutua. 27
  - I.7.4.4 El problema productor-consumidor. 29
  - I.7.4.5 El problema de los filósofos comensales. 30
  - I.7.4.6 El problema de los lectores-escritores. 31

CAPITULO II LA PROGRAMACION ORIENTADA A OBJETOS Y SMALLTALK 36

II.1 Antecedentes. 36

II.2 Características de la POO. 37

II.2.1 Encapsulación. 38
II.2.2 Herencia. 38
II.2.3 Polimorfismo. 39

II.3 Principios de diseño de Smalltalk. 40
II.3.1 El lenguaje. 40
II.3.2 Los objetos comunicantes. 41
II.3.3 La organización. 42
II.3.4 La interfaz del usuario. 44

II.4 Conceptos básicos de Smalltalk. 45
II.4.1 Las clases y sus instancias. 47
II.4.2 Herencia. 48

II.5 Ventajas y desventajas de la POO. 49

CAPITULO III UN SISTEMA PARA MANEJAR REDES DE PETRI 50

III.1 Introducción. 50

III.2 Las clases de objetos, sus variables de instancia y sus métodos. 52
III.2.1 La clase PetriNode. 52
III.2.1.1 La clase Place. 53
III.2.1.2 La clase Transition. 54
III.2.2 La clase PetriNet. 55
III.2.3 La clase PetriNetDiskBrowser. 58
III.2.4 La clase PetriNetShowResults. 60
III.2.5 Métodos creados en clases de objetos ya existentes. 62

III.3 Interacción de las distintas clases de objetos. 63

III.4 Especificaciones del archivo que contiene la estructura de la RP 64

III.5 Manual de uso del SMRP. 67
III.5.1 Ingreso a Smalltalk. 67
III.5.2 Manejo de los menús. 67
III.5.3 Ingreso al SMRP. 68
III.5.4. Manejo de la ventana múltiple del explorador de discos. 68
III.5.3.1 Ventana del listado de directorios. 69
III.5.3.2 Ventana del listado de archivos. 70
III.5.3.3 Ventana del editor de texto. 70
III.5.4 Manejo de la ventana gráfica. 73
III.5.4.1 Manejo de las ventanas de resultados. 76

CONCLUSIONES 77

APENDICE A 79

APENDICE B 82

APENDICE C 91

BIBLIOGRAFIA 93

INTRODUCCION

El modelo matemático de Redes de Petri (RP) está basado en la tesis doctoral del alemán Carl Adam Petri, quien presentó su trabajo en 1962. De ahí en adelante, la teoría de las RP se ha desarrollado y aplicado en diversas universidades y centros de investigación, sobre todo en Europa y Estados Unidos.

Aunque la computación ha sido tradicionalmente el campo de aplicación de las RP, también podemos ver la utilidad de este modelo en áreas como ingeniería de control, química, economía o sociología. La motivación para crear un sistema de cómputo que maneje RP proviene precisamente del constante incremento en la utilización del modelo, dejando así a la computadora el trabajo de cálculo, mecánico y laborioso, con lo que se evitan errores en la información que proporciona el modelo.

El sistema de cómputo que aquí se presenta está basado en el ambiente de programación orientado a objetos (POO) de Smalltalk/V R2.0, y puede ser utilizado en equipos IBM PC o compatibles. Decidimos utilizar Smalltalk por dos razones principales: por la amplia gama de clases de objetos que ofrece y por la facilidad de crear nuevas clases que interactúen con las ya existentes.

La elaboración de esta tesis consistió de las siguientes etapas:

1) La investigación sobre la teoría y las aplicaciones de las RP.

2) La investigación sobre el paradigma de POO.

3) El aprendizaje del manejo del ambiente y de la sintaxis de programación de Smalltalk.

4) La programación del Sistema para Manejar Redes de Petri (SMPR), en el ambiente de Smalltalk.

El capítulo I, "Las Redes de Petri y el Modelado de Sistemas", muestra los antecedentes y conceptos básicos del modelo, así como ejemplos de sistemas que pueden modelarse con las RP. Se utiliza la notación de Peterson por considerarla sencilla de manejar.

La investigación sobre los conceptos de la POO se vierte en el capítulo II, "La Programación Orientada a Objetos y Smalltalk". Existe una gran diversidad de definiciones de lo que es un lenguaje orientado a objetos (LOO), aunque la característica común de las diferentes herramientas es que utilizan la encapsulación en distintas formas. En este capítulo se hace énfasis en los principios de diseño y en los conceptos de Smalltalk, por considerarlos importantes para la comprensión del funcionamiento interno y externo del SMRP.

Las distintas clases de objetos que conforman el SMRP se detallan en el capítulo III, "Un Sistema para Manejar Redes de Petri". Las variables y métodos de instancia de cada clase están brevemente explicadas, así como la interacción de las distintas clases. Al final del capítulo se encuentra el manual de uso del SMRP.

En el apéndice A aparece una breve teoría sobre las bolsas, las cuales se utilizan en la definición de las funciones de entrada y salida de las RP.

Por ahorro de espacio, en el apéndice B sólo se exponen los ejemplos de métodos de instancia que se consideran más representativos.

Los requerimientos de hardware y software, así como la instalación de Smalltalk y del SMRP, se explican en el apéndice C.

El director e impulsor de esta tesis es José Antonio Delgado V., de la Unidad de Cómputo de El Colegio de México. Guadalupe Ibargüengoitia, del Departamento de Matématicas de la Facultad de Ciencias de la UNAM, accedió amablemente a asesorar el trabajo. El autor asume su exclusiva responsabilidad en los errores cometidos.

Por último, si algún lector se interesa por tener una copia del SMRP, puede llamar al teléfono 568-60-33, extensión 197, a donde también puede hacer llegar sus sugerencias para mejorar este trabajo, las cuales agradeceré encarecidamente.

M. S. G.

CAPITULO I

LAS REDES DE PETRI

Y EL MODELADO DE SISTEMAS

I.1 Modelado.

En muchos campos del conocimiento un fenómeno no se estudia directamente, sino a través de un modelo de dicho fenómeno. Un modelo es una representación, generalmente en términos matemáticos, de lo que se cree que son características importantes del objeto bajo estudio. Al manipular dicha representación se espera obtener nuevos conocimientos acerca del fenómeno modelado, sin el peligro, costo o inconveniencia de manejar el fenómeno real mismo.

La mayoría de las técnicas de modelado utilizan las matématicas, aunque no se descartan representaciones de otro tipo. Las características importantes de muchos fenómenos físicos pueden describirse numéricamente, y las relaciones entre esas características pueden a su vez describirse por ecuaciones o desigualdades. Sin embargo, para utilizar con éxito la aproximación del modelado se requiere un conocimiento del fenómeno y las propiedades de la técnica del modelado.

El desarrollo de las computadoras ha incrementado el uso y la utilidad del modelado. Al representar un sistema con un modelo matemático, convirtiendo ese modelo en instrucciones para una computadora y ejecutando estas últimas en la máquina, es posible modelar más grandes y más complejos sistemas como nunca antes. Así, las computadoras están relacionadas con el modelado en dos formas: como herramienta para modelar y como sujeto del modelado.

I.2 Características de los sistemas.

Los sistemas computacionales, económicos, legales, de control de tráfico, físicos, químicos, etc., son complejos y están formados de muchos componentes interactuantes. Cada componente puede ser tan complejo como sus interacciones con otros componentes en el sistema.

A pesar de la diversidad de sistemas que queremos modelar, varios puntos son obvios. Una idea fundamental es que los sistemas se forman de componentes interactuantes y separados. Cada componente puede en sí mismo ser un sistema, pero su conducta puede describirse independientemente de los otros componentes del sistema, excepto por interacciones bien definidas con otros

componentes. Cada componente tiene su propio estado, el cual es una abstracción de la información relevante y necesaria para describir sus acciones futuras. A menudo el estado de un componente depende de la historia del componente. De este modo, el estado de un componente puede cambiar con el tiempo.

Los componentes de un sistema exhiben concurrencia o paralelismo. Las actividades de un componente de un sistema pueden ocurrir simultáneamente con las actividades de otros componentes. La naturaleza concurrente de la actividad en un sistema crea algunos problemas difíciles en el modelado. Ya que los componentes de los sistemas interactúan, es necesario que haya sincronización. La transferencia de información de un componente a otro requiere que las actividades de los componentes relacionados estén sincronizadas mientras la interacción ocurre. Esto puede resultar en un componente esperando por otro componente. El registro del tiempo de las acciones de diferentes componentes puede ser muy complejo y las interacciones resultantes entre los componentes difíciles de describir.

I.3 Desarrollo inicial de las Redes de Petri (RP).

Las RP se diseñaron para modelar sistemas con componentes concurrentes interactuantes. En la tesis doctoral "Comunicación con autómatas", del matemático alemán Carl Adam Petri, se formula la base de una teoría de comunicación entre componentes asíncronos de un sistema computacional. Petri estaba interesado en la descripción de las relaciones causales entre eventos. Su trabajo fue principalmente un desarrollo teórico de los conceptos básicos a partir de los cuales las RP se han desarrollado.

El trabajo de Petri llamó la atención de A. W. Holt y otras personas del Proyecto de Teoría de Sistemas de Applied Data Research. Mucha de la teoría inicial, notación y representación de las RP se desarrollaron a partir de los trabajos "Teoría de Sistemas de Información" (Holt et al, 1968), y "Condiciones y Eventos" (Holt y Commoner, 1970). Este último trabajo mostró cómo las RP podrían aplicarse al modelado y análisis de sistemas de componentes concurrentes.

Asimismo el trabajo de Petri llamó la atención del Proyecto MAC en el Instituto Tecnológico de Massachusets. El Grupo de Estructuras Computacionales, bajo la dirección de Jack B. Dennis, ha sido fuente de investigaciones y publicaciones sobre RP, así como organizador de dos conferencias en Estados Unidos, en 1970 y 1975.

En los últimos años se ha extendido el uso y el estudio de las RP. En París se realizó un taller sobre RP, en 1977, y en Hamburgo un curso avanzado sobre Teoría General de Redes, en 1979. Recientemente se han organizado talleres sobre RP en Turín, Italia en 1985 y en Wisconsin, EUA, en 1987. Además se ha formado en Alemania Federal un grupo especial de interés sobre RP.

I.4 Teorías pura y aplicada de RP.

El estudio de las RP se ha desarrollado en dos direcciones: la teoría aplicada y la teoría pura. La primera se interesa por la aplicación de las RP al modelado de sistemas, el análisis de esos sistemas y los conocimientos resultantes en el sistema modelado. La teoría pura se encarga del estudio de las RP para desarrollar las herramientas básicas, técnicas y conceptos necesarios para la aplicación de RP. Aunque la motivación por la investigación sobre RP se basa en las aplicaciones, se necesita una teoría de RP bien fundamentada para poder aplicarlas. Mucho del trabajo sobre RP en este punto se ha concentrado en la teoría fundamental, desarrollando herramientas que pueden ser útiles en la aplicación de RP a problemas reales específicos.

I.5 Aplicación de la teoría de las RP.

La aplicación de las RP al diseño y análisis de sistemas puede llevarse a cabo de varias formas. Un enfoque considera a las RP como una herramienta auxiliar de análisis. De esta manera, las técnicas convencionales de diseño se utilizan para especificar un sistema. Después, éste se modela como una RP y el modelo es analizado. Cualquier problema encontrado en el análisis nos lleva a errores en el diseño, el cual debe modificarse para corregir los errores. El diseño modificado puede modelarse y analizarse nuevamente. El ciclo se repite hasta que el análisis no muestre problemas.

En un enfoque alternativo, el diseño y el proceso de especificación se lleva a cabo en términos de RP. Las técnicas de análisis se aplican sólo para crear un diseño de RP que no tenga errores. Más tarde, el problema es transformar la representación de RP en un sistema real de trabajo.

En el primer enfoque debemos desarrollar técnicas de modelado para transformar sistemas en representaciones de RP; en el segundo debemos desarrollar técnicas de implementación para transformar representaciones de RP en sistemas. En ambos casos necesitamos técnicas de análisis para determinar las propiedades de nuestro modelo de RP.

I.6 Definiciones básicas de RP.

Los conceptos de RP están basados en la teoría de las bolsas, una extensión de la teoría de los conjuntos. En el apéndice A podemos encontrar una breve explicación de la teoría de las bolsas.

I.6.1 Estructura de una RP.

Una RP se compone de cuatro partes: un conjunto P de sitios (también llamados lugares), un conjunto T de transiciones, una función I de entrada y una función O de salida. La función I de entrada es un mapeo de una transición t_j a una bolsa de sitios $I(t_j)$, los cuales se conocen como los sitios de entrada de la transición. La función O de salida mapea una transición t_j en una bolsa de sitios $O(t_j)$, conocidos como los sitios de salida de la transición.

Definición I.1 Una RP es la cuarteta $C = (P,T,I,O)$ tal que $P = \{p_1,p_2,...,p_n\}$ es un conjunto finito de sitios, $n \geq 0$; $T = \{t_1,t_2,...,t_m\}$ es un conjunto finito de transiciones, $m \geq 0$. P y T son disjuntos, i. e., $P \cap T = \emptyset$. $I : T \rightarrow P^{\infty}$ es la función de entrada y $O : T \rightarrow P^{\infty}$ es la función de salida. Ambas mapean transiciones en bolsas de sitios. (P^{∞} es el conjunto de bolsas sobre el dominio P, en las cuales no existe límite en el número de ocurrencias de un elemento).

A continuación tenemos dos ejemplos de estructuras de RP:

Ejemplo I.1

$$C = (P,T,I,O)$$

$P = \{p_1,p_2,p_3,p_4,p_5,p_6\}$, $T = \{t_1,t_2,t_3,t_4,t_5\}$,
$I : T \rightarrow P^{\infty}$, $O : T \rightarrow P^{\infty}$, tales que:

$I(t_1) = \{p_2,p_3\}$	$O(t_1) = \{p_5,p_6\}$
$I(t_2) = \{p_4\}$	$O(t_2) = \{p_1,p_3,p_3\}$
$I(t_3) = \{p_1,p_2\}$	$O(t_3) = \{p_2\}$
$I(t_4) = \{p_5,p_6\}$	$O(t_4) = \{p_2,p_3\}$
$I(t_5) = \{p_1,p_6\}$	$O(t_5) = \{p_4,p_5,p_6\}$

Ejemplo I.2

$$C = (P,T,I,O)$$

$P = \{p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9\}$, $T = \{t_1,t_2,t_3,t_4,t_5,t_6\}$,
$I : T \rightarrow P^{\infty}$, $O : T \rightarrow P^{\infty}$, tales que:

$I(t_1) = \{p_1\}$	$O(t_1) = \{p_2,p_3\}$
$I(t_2) = \{p_8\}$	$O(t_2) = \{p_1,p_7\}$
$I(t_3) = \{p_2,p_5\}$	$O(t_3) = \{p_6\}$
$I(t_4) = \{p_3\}$	$O(t_4) = \{p_4\}$
$I(t_5) = \{p_6,p_7\}$	$O(t_5) = \{p_9\}$
$I(t_6) = \{p_4,p_9\}$	$O(t_6) = \{p_5,p_8\}$

Un sitio p_i es un sitio de entrada de una transición t_j si $p_i \in I(t_j)$; p_i es un sitio de salida si $p_i \in O(t_j)$. Las entradas y salidas de una transición son bolsas de sitios, lo que permite a un sitio ser entrada o salida múltiple de una transición. La multiplicidad de un sitio de entrada p_i de una transición t_j es el número de ocurrencias del sitio en la bolsa de entrada de la transición, es decir, $\#(p_i,I(t_j))$. Análogamente, la multiplicidad de un sitio de salida p_i para una transición t_j es

$\#(p_i,O(t_j))$. Si las funciones de entrada y salida mapean transiciones en conjuntos de sitios, entonces la multiplicidad de cada sitio es cero o uno.

I.6.2 Gráficas de RP.

Una gráfica de RP es la representación de una estructura de RP. Esta última consiste de sitios y transiciones. Correspondientemente, una gráfica de RP tiene dos tipos de nodos: un círculo O representa un sitio y una barra I representa una transición. Los sitios y las transiciones se conectan con arcos dirigidos (flechas), con algunos de éstos llevados de los sitios a las transiciones y viceversa. Un arco dirigido de un sitio p_i a una transición t_j hace que p_i sea una entrada de t_j. Las entradas múltiples de una transición se indican por arcos múltiples de los sitios de entrada a la transición. Un sitio de salida se indica por un arco dirigido de la transición al sitio. Igualmente, las salidas múltiples se representan por arcos múltiples.

Una gráfica de RP es una multigráfica bipartita dirigida. Es múltiple ya que permite arcos múltiples de un nodo a otro de la gráfica. Además, como los arcos son dirigidos, es una gráfica múltiple dirigida. Ya que los nodos de la gráfica pueden dividirse en dos conjuntos (de sitios y de transiciones, aunque en este caso estrictamente no son conjuntos sino bolsas), tales que cada arco se dirige de un elemento de un conjunto (sitio o transición) a un elemento de otro conjunto (transición o sitio), es una gráfica múltiple dirigida bipartita.

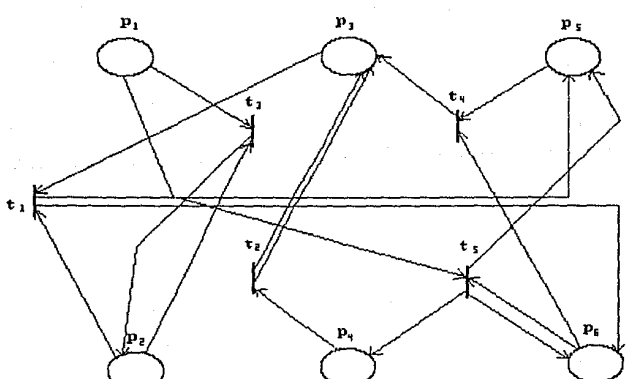


Figura I.1

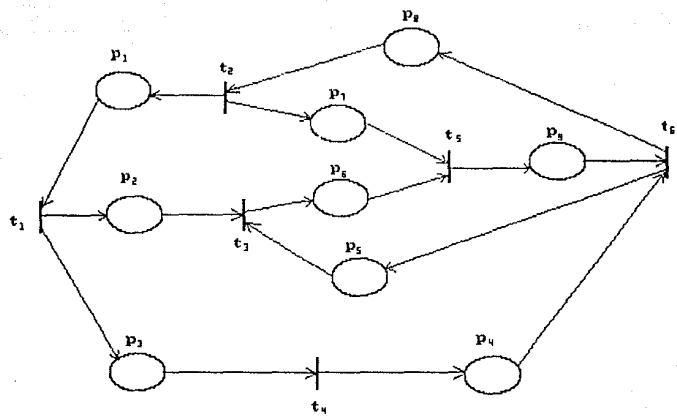


Figura I.2

Definición I.2 Una gráfica G de RP es una gráfica múltiple dirigida bipartita, G = (V,A), donde V = $\{v_1,v_2,...,v_s\}$ es un conjunto de vértices, $s \geq 0$, y A = $\{a_1,a_2,...,a_r\}$ es una bolsa de arcos dirigidos, $r \geq 0$; $a_i = (v_j,v_k)$, con v_j, $v_k \in V$. El conjunto V puede partirse en dos conjuntos, X y Z, tales que $V = X \cup Z$, $X \cap Z = \emptyset$, y para cada arco dirigido $a_i \in A$, si $a_i = (v_j,v_k)$, entonces $v_j \in X$ y $v_k \in Z$ o $v_j \in Z$ y $v_k \in X$.

En las Figuras I.1 y I.2 tenemos las gráficas de RP que representan las estructuras de RP de los ejemplos I.1 y I.2, respectivamente.

Para demostrar la equivalencia de las dos representaciones de una RP, la estructura de RP y la gráfica de RP, veremos como transformar una en otra.

Supongamos que tenemos una estructura de RP, C = (P,T,I,O), con P = $\{p_1,p_2,...,p_n\}$ y T = $\{t_1,t_2,...,t_m\}$. Entonces podemos definir una gráfica de RP como sigue:

Definición I.3 Sea V = P U T. Sea A la bolsa de arcos dirigidos tales que para toda $p_i \in P$ y $t_j \in T$

$\#((p_i,t_j),A) = \#(p_i,I(t_j))$
$\#((t_j,p_i),A) = \#(p_i,O(t_j))$

G = (V, A) es una gráfica de RP equivalente a la estructura de RP C = (P,T,I,O).

Ahora supongamos que tenemos una gráfica de RP, G = (V, A), donde $V = \{v_1, v_2, ..., v_s\}$, con $s \geq 0$, es un conjunto de vértices y $A = \{a_1, a_2, ..., a_r\}$, con $r \geq 0$, es una bolsa de arcos dirigidos; $a_i = (v_j, v_k)$, con $v_j, v_k \in V$. Así podemos definir una estructura de RP de la siguiente forma:

Definición I.4 Sabemos que $V = X \cup Z$, donde $X \cap Z = \emptyset$. Sean P = X el conjunto de sitios y T = Z el conjunto de transiciones. $P = \{p_1, p_2, ..., p_n\}$, $n \geq 0$, $T = \{t_1, t_2, ..., t_m\}$, $m \geq 0$, y $n + m = s$. Sean I, $O : T \rightarrow P^{\infty}$ tales que

i) Si $a_i = (v_j, v_k)$, $v_j \in X$ y $v_k \in Z$, $i \in \{1,2,...,r\}$, $j, k \in \{1,2,...,s\}$, entonces $v_j = p_q$, para alguna $q \in \{1,2,...,n\}$, $v_k = t_g$, para alguna $g \in \{1,2,...,m\}$ y $p_q \in I(t_g)$.

ii) Si $a_i = (v_j, v_k)$, $v_j \in Z$ y $v_k \in X$, $i \in \{1,2,...,r\}$, $j, k \in \{1,2,...,s\}$, entonces $v_j = t_g$, para alguna $g \in \{1,2,...,m\}$, $v_k = p_q$, para alguna $q \in \{1,2,...,m\}$ y $p_q \in O(t_g)$.

Con la definición anterior surge un problema: si el conjunto de vértices V está partido en X y Z, ¿cuál conjunto debería ser el de sitios y cuál el de transiciones? Ambas elecciones posibles permiten definir una estructura de RP, aunque las dos estructuras resultantes tienen intercambiados los sitios y las transiciones. Lo anterior nos lleva a definir el dual de una RP.

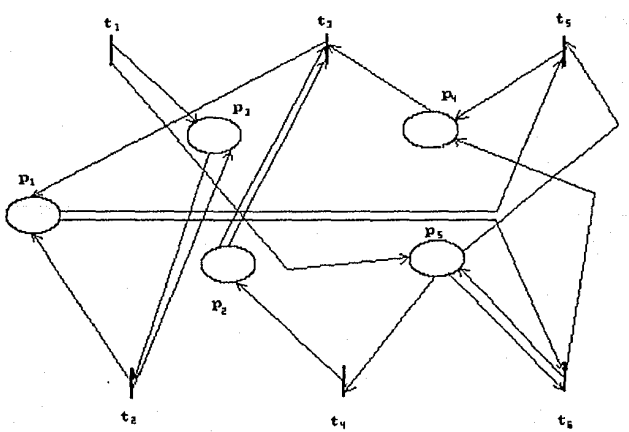


Figura I.3

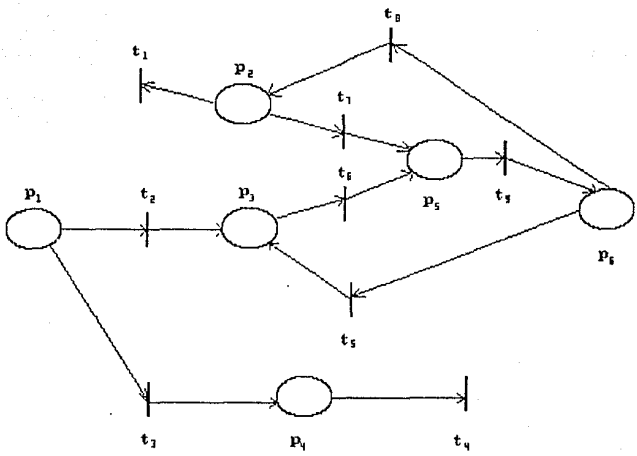


Figura I.4

Definición I.5 El dual de una estructura de RP C = (P,T,I,O), es la estructura C' = (T,P,I,O), la cual resulta de intercambiar los sitios y las transiciones.

En una gráfica del dual de una RP, la estructura se mantiene y simplemente se intercambian los círculos y barras de la gráfica para indicar el cambio en los sitios y las transiciones. En las fig. I.3 y I.4 presentamos las gráficas de los duales de las estructuras de los ejemplos I.1 y I.2, respectivamente.

Las gráficas de RP son multigráficas porque un sitio puede ser una entrada o salida múltiple de una transición. Esto produce una gráfica con varios arcos entre el sitio y la transición, lo que es satisfactorio para multiplicidades pequeñas (hasta tres quizá), aunque sería inconveniente para multiplicidades muy grandes. Así, una representación alternativa para estructuras con multiplicidad grande es el uso de un haz de arcos. Un haz es un arco especial dibujado muy grueso y etiquetado con su multiplicidad. La fig. I.5 muestra una transición con multiplicidad de entrada 5 y de salida 12.

I.6.3 Marcajes de RP.

Un marcaje es una asignación de marcas (o señales) a los sitios de una RP. El número y posición de las marcas puede cambiar durante la ejecución de la RP (de este último concepto hablaremos posteriormente).

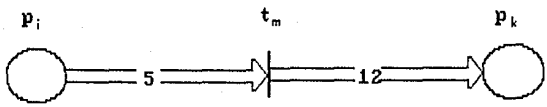


Figura I.5

Definición I.6 El marcaje μ de una RP C = (P,T,I,O) es una función $\mu : P \rightarrow \mathbf{N}$.

El marcaje puede definirse también como un vector de dimensión n, $\mu^\circ = (\mu^\circ_1,\mu^\circ_2,...,\mu^\circ_n)$, donde $n = |P|$ y cada $\mu^\circ_i \in N$, $i \in \{1,2,...,n\}$. El vector μ° nos da, para cada sitio p_i en una RP, el número de marcas en dicho sitio, μ°_i, $i \in \{1,2,...,n\}$. Las definiciones de marcaje como función y como vector están relacionadas por $\mu(p_i) = \mu^\circ_i$.

Definición I.7 Una RP marcada M = (C,μ) es una estructura de RP C = (P,T,I,O) y un marcaje μ. Esto también se escribe como M = (P,T,I,O,μ).

En una gráfica de RP las marcas se representan como pequeños puntos dentro de los círculos que representan a los sitios de una RP. En la fig. I.6 tenemos una gráfica de RP marcada.

Ya que el número de marcas que pueden asignarse a un sitio de una RP es no acotado, hay una infinidad de marcajes para una RP.

I.6.4 Reglas de ejecución de RP.

La ejecución de una RP se controla por el número y distribución de marcas en la RP. Las marcas residen en los sitios y controlan la ejecución de las transiciones en la red. Una RP se ejecuta encendiendo (o disparando) las transiciones. Una transición se enciende al remover marcas de sus sitios de entrada y creando nuevas marcas que se distribuyen en sus sitios de salida.

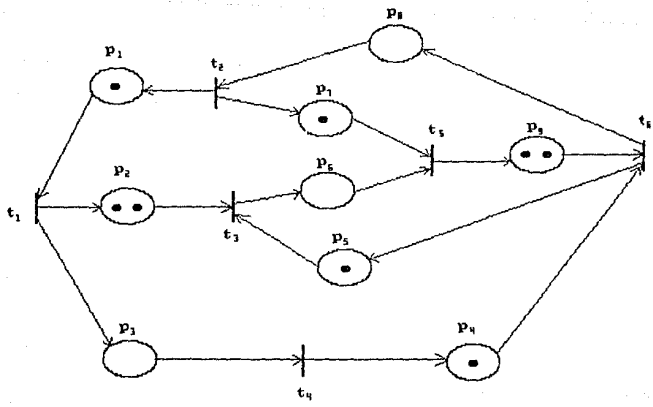


Figura I.6

Una transición puede encenderse si está habilitada. Una transición está habilitada si cada uno de sus sitios de entrada tiene al menos tantas marcas como arcos a la transición. Se necesitan marcas múltiples para arcos múltiples de entrada. Las marcas en los sitios de entrada que habilitan una transición son sus marcas habilitantes.

Definición I.8 Una transición t_j en una RP marcada $M = (P,T,I,O,\mu)$ está habilitada si para toda $p_i \in P$, $\mu(p_i) \geq \#(p_i,I(t_j))$.

Una transición se enciende removiendo todas sus marcas habilitantes de sus sitios de entrada y depositando dentro de cada uno de sus sitios de salida una marca por cada arco de la transición al sitio. Marcas múltiples se producen por arcos múltiples de salida.

Al encender una transición se cambiará el marcaje μ de la RP en un nuevo marcaje μ'. Ya que sólo transiciones habilitadas pueden encenderse, el número de marcas en cada sitio siempre es no negativo cuando una transición se enciende. Al encender una transición nunca se puede intentar remover una marca de donde no la hay. Si no hay suficientes marcas en cualquier sitio de entrada de una transición, entonces dicha transición no está habilitada y no puede encenderse.

Definición I.9 Una transición t_j en una RP marcada con marcaje μ puede encenderse siempre y cuando esté habilitada. Al encender una transición habilitada t_j se obtiene un nuevo marcaje μ', definido por $\mu'(p_i) = \mu(p_i) - \#(p_i,I(t_j)) +$

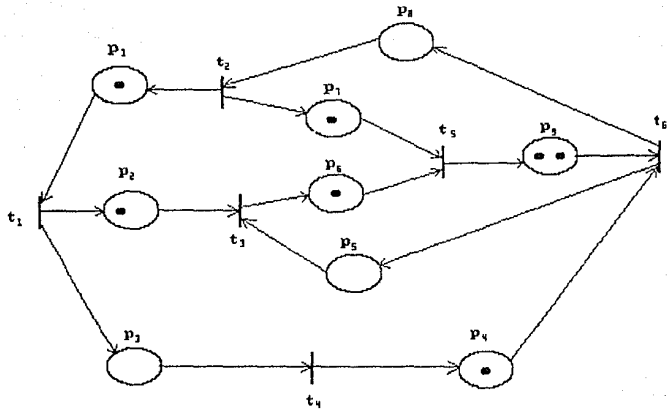


Figura I.7

$\#(p_i,O(t_j))$, $p_i \in P$, $t_j \in T$.

En las figuras I.7, I.8 y I.9 tenemos ejemplos de la ejecución de una RP. En la fig. I.7 la RP tiene habilitadas las transiciones t_1,t_5 y t_6. Al encender t_5 se toma una marca de p_6 y una de p_7 (que son los sitios de entrada de t_5) y se agrega una marca a p_9 (que es el sitio de salida de t_5), con lo que se obtiene un nuevo marcaje, que se muestra en la gráfica de la fig. I.8. Ahora t_1 y t_6 están habilitadas. Encendiendo t_6 se toma una marca de p_4 y una de p_9, y se agrega una marca a p_8, con lo que se obtiene el marcaje presentado en la gráfica de la fig. I.9.

Los encendidos de transiciones pueden continuar siempre que exista por lo menos una transición habilitada. Cuando no existen transiciones habilitadas, la ejecución se detiene.

I.6.5 Espacio de estados de RP.

El estado de una RP se define por su marcaje. El encendido de una transición representa un cambio en el estado de la RP mediante un cambio en el marcaje de la red. El espacio de estados de una RP con n sitios es el conjunto de todos los marcajes, i. e., $\mathbf{N}^n$. El cambio en el estado causado por el encendido de una transición se define por la función de cambio δ, llamada la función estado-siguiente. Cuando δ se aplica a un marcaje (estado) μ y a una transición t_j, produce el nuevo marcaje (estado) que resulta de encender la transición t_j en el

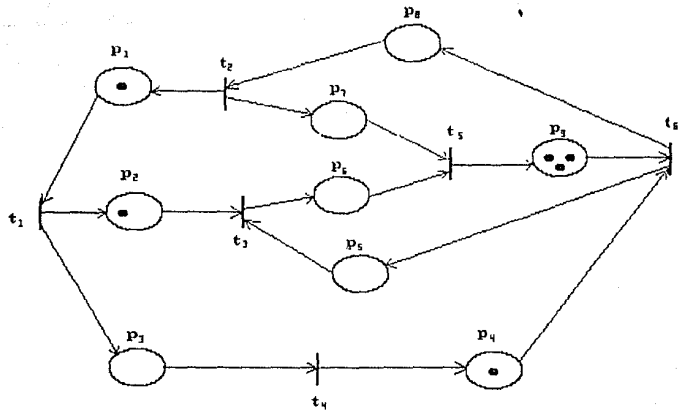


Figura I.8

marcaje μ. Ya que t_j puede sólo encenderse si está habilitada, δ no está definida si t_j no está habilitada en el marcaje μ. Si t_j está habilitada, entonces $\delta(\mu,t_j) = \mu'$, donde μ' es el marcaje que resulta de remover las marcas desde las entradas de t_j y añadiendo marcas a las salidas de t_j.

Definición I.10 La función estado siguiente $\delta : \mathbf{N}^n \times T \rightarrow \mathbf{N}^n$, para una RP $C = (P,T,I,O)$ con marcaje μ y transición $t_j \in T$, está definida si y sólo si $\mu(p_i) \geq \#(p_i,I(t_j))$, para todo $p_i \in P$. Si así sucede entonces $\delta(\mu,t_j) = \mu'$, donde $\mu'(p_i) = \mu(p_i) - \#(p_i,I(t_j)) + \#(p_i,O(t_j))$, para toda $p_i \in P$.

Dada una RP $C = (P,T,I,O)$ y un marcaje inicial μ^o, podemos ejecutar la RP por encendidos sucesivos de transiciones. Al encender una transición habilitada t_j en el marcaje inicial, se produce un nuevo marcaje $\mu^1 = \delta(\mu^o,t_j)$. En este nuevo marcaje podemos encender cualquier otra transición habilitada, digamos t_k, resultando un nuevo marcaje $\mu^2 = \delta(\mu^1,t_k)$. Esto puede continuar siempre y cuando exista una transición habilitada en cada marcaje. Si alcanzamos un marcaje en el que ninguna transición está habilitada, entonces ninguna transición puede encenderse, la función estado-siguiente no está definida y la ejecución debe detenerse.

Dos sucesiones resultan de la ejecución de una RP: la sucesión de marcajes $(\mu^o,\mu^1,\mu^2,...)$ y la sucesión de transiciones que se encendieron $(t_{j0},t_{j1},t_{j2},...)$. Estas dos sucesiones se relacionan por la función δ, tal que $\delta(\mu^k,t_{jk}) = \mu^{k+1}$, para $k \in NU\{0\}$. Dada una sucesión de transiciones y μ^o, podemos fácilmente obtener la

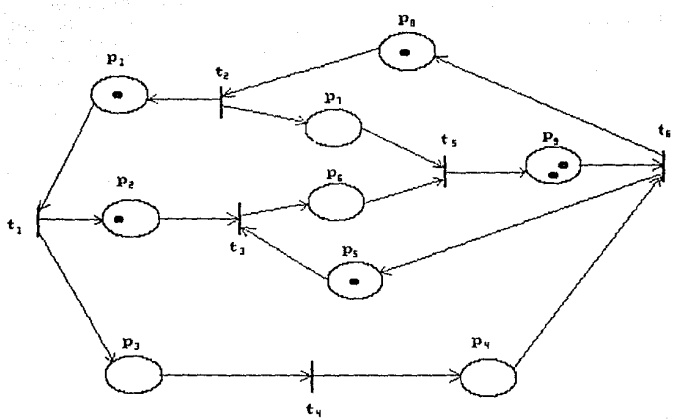


Figura I.9

sucesión de marcajes para la ejecución de la RP y dada la sucesión de marcajes, excepto por unos cuantos casos degenerados, podemos obtener la sucesión de transiciones. Ambas sucesiones nos proporcionan un registro de la ejecución de la RP.

Al encender una transición en un marcaje μ obtenemos un nuevo marcaje μ'. Decimos que μ' es alcanzable inmediatamente desde μ.

Definición I.11 Para una RP C = (P,T,I,O) con marcaje μ, un marcaje μ' es alcanzable inmediatamente si existe una transición $t_j \in T$ tal que $\delta(\mu,t_j) = \mu'$.

Podemos extender este concepto para definir el conjunto de marcajes alcanzables para una cierta RP marcada. Si μ' es inmediatamente alcanzable desde μ y μ'' es inmediatamente alcanzable desde μ', entonces decimos que μ'' es alcanzable desde μ. Definimos el conjunto de estados alcanzables $R(C,\mu)$ de una RP C con marcaje μ, como aquél que está formado por todos los marcajes que son alcanzables desde μ. Un marcaje μ' está en $R(C,\mu)$ si existe una sucesión de encendido de transiciones que pase del marcaje μ al marcaje μ'.

Definición I.12 El conjunto de estados alcanzables $R(C,\mu)$, para una RP C = (P,T,I,O) con marcaje μ, es el conjunto más pequeño de marcajes definido por:

i) $\mu \in R(C,\mu)$.

ii) Si $\mu' \in R(C,\mu)$ y $\mu'' = \delta(\mu',t_j)$ para alguna $t_j \in T$, entonces $\mu'' \in R(C,\mu)$.

I.7 Modelado con RP.

Las RP se diseñaron y se utilizan principalmente para modelar. Muchos sistemas, especialmente aquéllos con componentes independientes, pueden modelarse con una RP, considerando la ocurrencia de diversos eventos y actividades, así como el flujo de información u otros recursos dentro de los sistemas.

I.7.1 Eventos y condiciones.

La inspección simple de un sistema en términos de RP se concentra en dos conceptos primitivos: eventos y condiciones. Los eventos son acciones que tienen lugar en el sistema, cuyo estado controla la ocurrencia de dichos eventos. El estado de un sistema puede describirse como un conjunto de condiciones. Una condición es un predicado o descripción lógica del estado de un sistema. Como tal, una condición puede mantenerse (ser verdadera) o no (ser falsa).

Como los eventos son acciones, éllos pueden ocurrir. Para que un evento ocurra es necesario que ciertas condiciones se mantengan. Estas son las precondiciones, que dejan de mantenerse y pueden causar otras condiciones, las postcondiciones, que se vuelven verdaderas.

Consideremos el problema de modelar un taller mecánico. Este espera hasta que una orden aparece y entonces trabaja la parte ordenada y la remite para su entrega. Las condiciones para el sistema son:

a) El taller está esperando.
b) Una orden ha llegado y está en espera.
c) El taller está trabajando en la orden.
d) La orden está completa.

Los eventos son:

1) Una orden llega.
2) El taller comienza a trabajar la orden.
3) El taller termina la orden.
4) La orden es enviada para su entrega.

Las precondiciones del evento 2) el taller comienza con la orden, son obvias: a) el taller está en espera, y b) una orden ha llegado y está en espera. La postcondición del evento 2) es c) el taller está trabajando en la orden. Análogamente podemos definir las precondiciones y postcondiciones de los demás eventos, y construir la siguiente tabla:

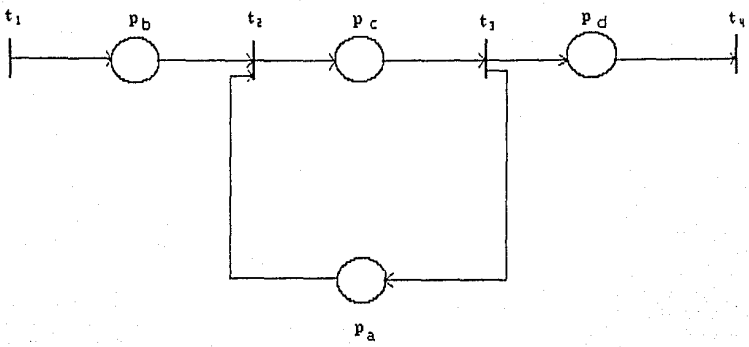


Figura I.10

Evento	Precondiciones	Postcondiciones
1	Ninguna	b
2	a, b	c
3	c	d, a
4	d	Ninguna

Esta perspectiva de un sistema puede modelarse fácilmente como una RP. Las condiciones se modelan por sitios y los eventos por transiciones. Las entradas de una transición son las precondiciones del evento correspondiente, así como las salidas son las postcondiciones. La ocurrencia de un evento corresponde al encendido de la transición respectiva. El mantenimiento de una condición se representa por una marca en el sitio correspondiente a la condición. Cuando la transición se enciende se remueven las marcas habilitantes que representan el mantenimiento de las precondiciones y se crean nuevas marcas que representan el mantenimiento de las postcondiciones.

En la fig. I.10 tenemos el modelo de RP del ejemplo del taller mecánico.

I.7.2 Concurrencia y paralelismo.

Un punto importante acerca de las RP y los sistemas que pueden modelar es su inherente paralelismo o concurrencia. En el modelo de RP dos eventos que están habilitados y no interactúan pueden ocurrir independientemente. No hay necesidad de sincronizar eventos, a menos que lo requiera el sistema a ser modelado. Cuando es necesaria la sincronización también es fácil modelarla. Así las RP son ideales para modelar sistemas de control distribuido con procesos múltiples ejecutándose concurrentemente en el tiempo.

Otra característica de las RP es su naturaleza asíncrona. No hay una medida inherente del tiempo o del flujo del tiempo en una RP. Esto refleja una filosofía del tiempo que establece que la única propiedad importante del tiempo, desde el punto de vista lógico, está en la definición de un orden parcial de ocurrencia de los eventos. Estos toman cantidades variables de tiempo en la vida real y esta variabilidad se refleja en el modelo de RP al no depender de una noción del tiempo para controlar la sucesión de eventos. La estructura de RP misma contiene toda la información necesaria para definir las posibles sucesiones de eventos.

Una ejecución de RP (y la conducta del sistema que modela) se ve como una sucesión de eventos discretos. El orden de ocurrencia de los eventos es uno de muchos posibles permitidos por la estructura básica. Esto conduce a un aparente no-determinismo en la ejecución de la RP. Si en cualquier tiempo más de una transición está habilitada, entonces cualquiera de las varias transiciones habilitadas puede ser la siguiente en encenderse. Desde el punto de vista de la ejecución clásica del modelo, la elección de qué transición se enciende está hecha en una forma no-determinística, i. e., aleatoriamente. Esta característica de las RP refleja el hecho de que en la vida real las situaciones en que varias cosas pasan concurrentemente, el orden aparente de la ocurrencia de eventos no es único, sino cualquiera de un conjunto de sucesiones de eventos pueden ocurrir. Sin embargo, el orden parcial en que los eventos ocurren es único.

"Todas las acciones están predeterminadas en el universo, y no existe la aleatoriedad. Esta es un resultado del conocimiento incompleto del estado del universo"[1]. En este sentido, la selección de un conjunto de transiciones habilitadas para encenderse se determina en el sistema modelado, pero no en el modelo porque éste no representa la información completa del sistema.

También debemos considerar la teoría de la relatividad. Uno de los principios básicos de esta teoría establece que la comunicación no es instantánea, sino más bien la información acerca de la ocurrencia de un evento se propaga en el espacio limitada con la velocidad de la luz. Esto significa que si dos eventos pueden ocurrir simultáneamente, entonces el orden de ocurrencia puede parecer distinto a dos observadores separados.

---

[1] Peterson, James L. Petri Net Theory and the Modeling of Systems. Ed. Prentice-Hall. Estados Unidos. 1981. pp 37.

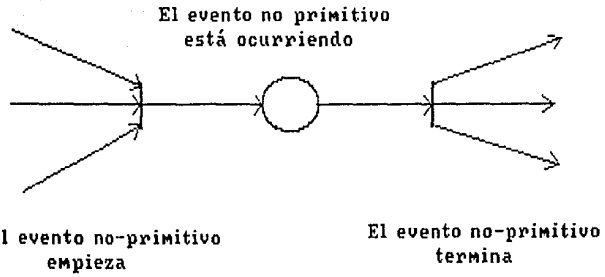


Figura I.11

Las condiciones anteriores son necesarias pero introducen una considerable complejidad en la descripción y análisis de una RP. Para disminuir esta complejidad se acepta una limitación en el modelado con RP. El encendido de una transición (y el evento asociado a élla) se considera como un evento instantáneo, i. e., que toma un tiempo cero, y las ocurrencias de dos eventos no pueden suceder simultáneamente. Los eventos modelados se llaman eventos primitivos, los cuales son instantáneos y no simultáneos.

Un evento no-primitivo es aquél que no toma un tiempo cero en ocurrir. Las operaciones no primitivas no son simultáneas y por lo tanto pueden traslaparse en el tiempo. Como la mayoría de los eventos en la realidad toman cierto tiempo, éstos son no-primitivos y no pueden ser modelados propiamente por transiciones en una RP. Sin embargo, un evento no-primitivo puede descomponerse en dos eventos primitivos, "el evento no-primitivo comienza" y "el evento no-primitivo termina", y una condición, "el evento no-primitivo está ocurriendo". En la fig. I.11 se muestra cómo modelar un evento no-primitivo.

El encendido no-determinístico y no-simultáneo de transiciones en el modelado de sistemas concurrentes se revela en dos formas. En la primera situación, dos transiciones habilitadas no se afectan una a la otra en forma alguna, y las posibles sucesiones de eventos incluyen algunas en las cuales una transición ocurre primero, y algunas en las cuales la otra transición ocurre primero. Lo anterior se llama concurrencia, y lo representamos en la figura I.12.

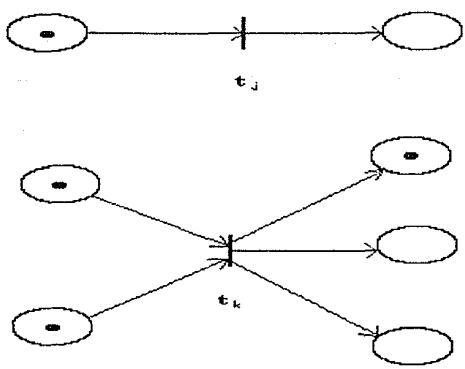


Figura I.12

En la otra situación, donde la simultaneidad es más difícil de manejar, tenemos que $p_i \in I(t_j)$ y $p_i \in I(t_k)$, para algunas $i \in \{1,2,...,n\}$, $j,k \in \{1,2,...,m\}$ y $\mu(p_i) = 1$. Aquí tenemos dos eventos que no ocurren simultáneamente. Las dos transiciones habilitadas están en conflicto. Sólo una transición puede encenderse, ya que al hacerlo, áquella remueve la marca de la entrada compartida y deshabilita la otra transición. En la figura I.13 tenemos la gráfica que representa la situación anterior.

I.7.3 Modelado de hardware.

I.7.3.1 Computadoras acanaladas (pipelined).

La técnica de canales es similar a la operación de una línea de ensamblado y es útil para procesamiento de vectores y arreglos. Un canal se compone de un número de etapas, las cuales pueden estar en ejecución simultáneamente. Cuando la etapa k termina, ésta entrega sus resultados a la etapa k+1 y mira hacia la etapa k-1 para un nuevo trabajo. Si cada etapa toma t unidades de tiempo y hay n etapas, entonces la operación completa para un producto toma n*t unidades de tiempo. Sin embargo, si el canal se mantiene abastecido con nuevos operandos, puede producir resultados a la velocidad de uno por cada t unidades de tiempo.

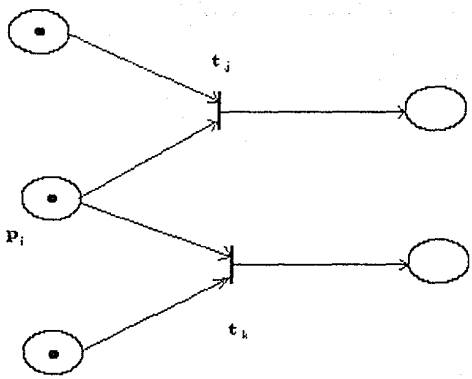


Figura I.13

Como un ejemplo, consideremos la suma de dos números de punto flotante. A grandes rasgos se tiene que realizar lo siguiente:

1. Extraer los exponentes de los dos números.
2. Comparar los exponentes e intercambiarlos si es necesario para ordenarlos en forma descendente.
3. Desplazar la fracción más pequeña para igualar los exponentes.
4. Sumar las fracciones.
5. Postnormalizar, i. e., verificar que la primera cifra de la fracción resultante de la suma sea distinta de cero, o que todas las cifras de dicha fracción sean iguales a cero.
6. Considerar el excedente o el faltante en el exponente y empaquetar el exponente del resultado.

Cada uno de los pasos anteriores puede realizarse por una unidad computacional separada, con un operando particular pasando de una unidad a otra para completar la suma. Esto permitiría, abasteciendo continuamente cada unidad, realizar no menos de seis sumas encaminadas.

La coordinación de las diferentes unidades puede manejarse en varias formas. Típicamente, el control de canales es síncrono; el tiempo permitido para cada paso en el canal es un tiempo constate fijo t. Cada t unidades de tiempo, el resultado de cada unidad es desplazado hacia abajo en el canal y se convierte en la entrada para la siguiente unidad. El enfoque síncrono puede innecesariamente

detener el proceso, ya que el tiempo necesario puede variar de unidad a unidad y también puede variar dentro de una unidad dada para diferentes entradas. En este caso, como el tiempo t debe seleccionarse como el tiempo máximo que puede necesitar la unidad más lenta del canal, podría darse el caso de que todas las unidades estuvieran ociosas la mayor parte del tiempo, esperando el resto de las t unidades de tiempo.

Un canal asíncrono puede acelerar, en promedio, cuando cada etapa del canal está completa y lista para entregar sus resultados y recibir nuevos operandos. Los resultados de la etapa k del canal pueden enviarse a la etapa k+1 tan pronto como la etapa k termine y la etapa k+1 esté libre. Consideremos una etapa arbitraria en el canal. Obviamente, debe existir un sitio para las entradas y salidas mientras están utilizándose o produciéndose. Típicamente, ésto incluye registros; la unidad utiliza valores en su registro de entrada para producir valores en su registro de salida. Esto debe esperar hasta que 1) su registro de salida se haya vaciado al copiarse en el registro de entrada de la siguiente etapa, y 2) una nueva entrada está disponible en su registro de entrada. Así, el control de la etapa k del canal necesita conocer cuando las siguientes condiciones se mantengan:

a) El registro de entrada está lleno.
b) El registro de entrada está vacío.
c) El registro de salida está lleno.
d) El registro de salida está vacío.
e) La unidad está ocupada.
f) La unidad está desocupada.

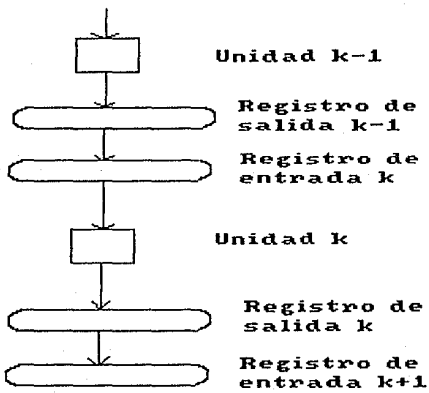


Figura I.14

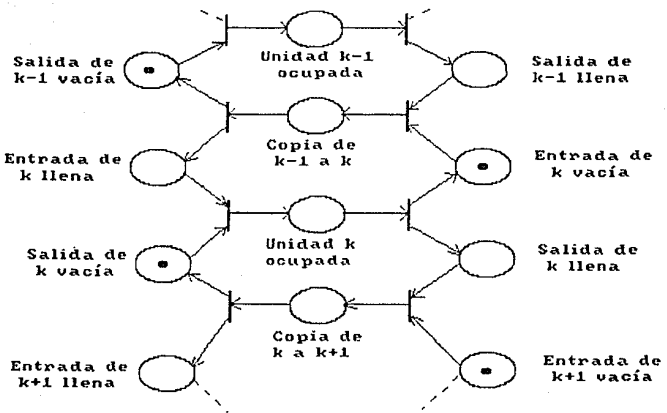


Figura I.15

g) La copia se está llevando a cabo.

Las figuras I.14 y I.15 muestran como un canal asíncrono de este tipo puede modelarse.

I.7.3.2 Unidades funcionales múltiples.

Otra aproximación para la construcción de grandes y muy rápidas computadoras es proveer unidades funcionales múltiples. Consideremos el ejemplo de la máquina CDC 6600. Esta computadora tiene diez unidades disponibles: 1 unidad booleana, 1 unidad de desplazamiento, 1 unidad de suma en punto flotante, 1 unidad de suma en punto fijo, 1 unidad de multiplicación, 1 unidad de división, 2 unidades de incremento y 1 unidad de bifurcación. Además, múltiples registros están dispuestos para mantener las entradas y salidas de las unidades funcionales. La unidad de control de la computadora se encarga de mantener en operación simultáneamente varias de las unidades independientes.

Veamos las siguiente sucesión de instrucciones basadas en el sistema CDC 6600:

1. Multiplicar X1 por X1, dando X0.
2. Multiplicar X3 por X1, dando X3.
3. Sumar X2 a X4, dando X4.
4. Sumar X0 a X3, dando X3.

5. Dividir X0 por X4, dando X6.

Cuando estas instrucciones se ejecutan, la unidad de control emite la primera instrucción a la unidad de multiplicación. Como hay dos unidades de multiplicación, la segunda instrucción puede emitirse. Notemos que ambas unidades pueden leer los contenidos de X1 sin problema. La instrucción 3 puede emitirse a la unidad de suma. Para emitir la instrucción 4, debemos esperar hasta que las instrucciones 1, 2 y 3 estén completas, ya que la instrucción 4 utiliza la unidad de suma (la cual se usa en la instrucción 3) para procesar X0 (que es calculado por la instrucción 1) y X3 (que es calculado por la instrucción 2). La instrucción 5 debe esperar a que la instrucción 1 termine de calcular X0 y que la instrucción 3 termine con X4.

La introducción de este tipo de paralelismo, ejecutando varias instrucciones de un programa simultáneamente, debe controlarse de tal forma que los resultados de ejecutar el programa, con y sin paralelismo, sean los mismos. Ciertas instrucciones en el programa requerirán que los resultados de instrucciones previas hayan sido computados exitosamente, antes de que puedan proceder las instrucciones siguientes. Un sistema que introduce paralelismo en un programa secuencial, de tal forma que mantiene correctos los resultados, se llama determinado. Las condiciones para conservar la condición de determinado en un sistema son las siguientes: Para dos operaciones a y b, tales uqe a precede a b en la precedencia lineal del programa, b puede comenzar antes que a esté realizada si y sólo si b no necesita los resultados de a como entradas, y los resultados de b no cambian las entradas o los resultados de a.

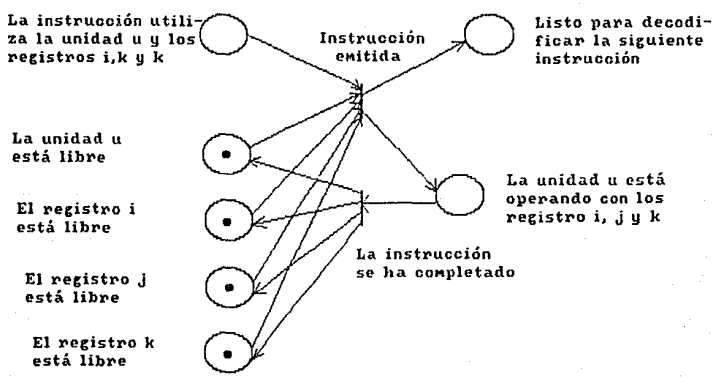


Figura I.16

Una tabla de reservación es un método de aplicar estas restricciones a la construcción de una unidad de control que emita instrucciones a unidades funcionales separadas. Una instrucción para la unidad funcional u, utilizando los registros i, j y k puede emitirse sólo si los cuatro componentes no están reservados; cuando la instrucción se emite, dichas componentes se vuelven reservadas. Si la instrucción no puede emitirse en este tiempo, ya sea porque la unidad funcional o uno de los registros está en uso, la unidad de control espera hasta que la instrucción pueda emitirse antes de continuar con la siguiente instrucción.

Este tipo de esquema puede modelarse como una RP. Para cada unidad funcional y cada registro, asociamos un sitio. Si la unidad o registro está libre, una marca estará en el sitio; si no lo está, el sitio no tendrá marca. Las unidades funcionales múltiples e idénticas pueden indicarse mediante múltiples marcas en los sitios. La figura I.16 muestra una parte de una RP que puede usarse para modelar la ejecución de una instrucción usando la unidad u y los registros i, j y k. El modelado de la unidad de control completa requeriría de una RP mucho más grande.

I.7.4 Modelado de software.

El modelado de software es quizá donde se utilizan con mayor frecuencia las RP, sobre todo en los últimos años. Mucho del reciente esfuerzo en el modelado de software está relacionado con el análisis, especificación y

```
begin
   Input(y1);
   Input(y2);
   y3 :=1;
   while y1 > 0
      do begin
         if odd(y1)
           then begin
                  y3 := y3 * y2;
                  y1 := y1 - 1;
           end;
         y2 := y2 * y2;
         y1 := y1 -2;
       end;
    Output(y3);
end;
```

Figura I.17

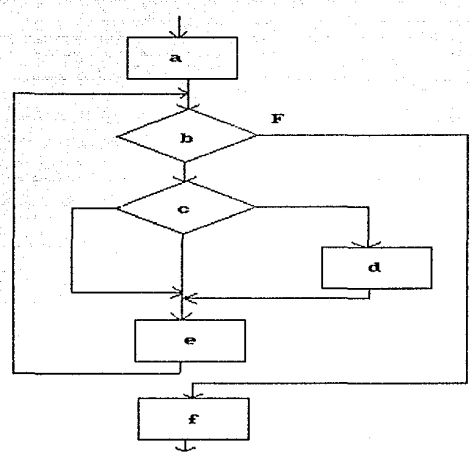


Figura I.18

descripción de programas secuenciales y también con sistemas de procesos concurrentes.

I.7.4.1 Diagramas de flujo.

El caso degenerado de un sistema de procesos concurrentes es aquel sistema con exactamente un proceso. Este puede describirse con un programa escrito en algún lenguaje de alto nivel o en lenguaje ensamblador. El programa representa dos aspectos separados del proceso: la computación (las operaciones aritméticas y lógicas) y el control (el orden de ejecución de las operaciones).

Las RP pueden representar mejor la estructura de control de los programas. Las RP son propuestas para modelar la sucesión de instrucciones y el flujo de información y computación, pero no los valores mismos de la información.

Un diagrama de flujo es un medio de representar la estructura de control de un programa. Por ejemplo, el programa de la fig. I.17 se representa con el diagrama de la fig. I.18. Notemos que este último no especifica los cálculos a realizarse, sino la estructura del programa. Este diagrama de flujo no tiene interpretación. La fig. I.19 muestra una interpretación de las acciones del diagrama de flujo que representa al programa.

Un diagrama de flujo pareciera muy similar a una RP. Está compuesto de nodos de dos tipos: las decisiones se representan por rombos y los

Acción	Interpretración
a	Input(y1); Input(y2); y3 := 1;
b	¿ y1 > 0 ?
c	¿ odd(y1) ?
d	y3 := y3 * y2; y1 := y1 - 1;
e	y2 := y2 * y2; y1 := y1 - 2 ;
f	Output(y3);

Figura I.19

cálculos se representan por rectángulos; además
hay arcos dirigidos entre los nodos. Cuando las instrucciones se ejecutan, las marcas se mueven alrededor del diagrama de flujo. La similaridad entre estas representaciones gráficas parecería indicar que podemos reemplazar en el diagrama los nodos por sitios y los arcos por transiciones, para crear un RP equivalente.

Sin embargo, consideremos que en el modelo de RP las transiciones modelan acciones, mientras que en el diagrama de flujo los nodos modelan acciones. También, si nuestra marca en un diagrama de flujo quisiera esperar, lo haría entre los nodos, en un arco, no dentro de una caja. Así, la traducción apropiada de un diagrama de flujo en una RP reemplaza los nodos del diagrama con transiciones de la RP y los arcos del diagrama con sitios de la RP. Cada arco del diagrama se representa con exactamente un sitio en la correspondiente RP. Los nodos del diagrama se representan en diferentes formas, dependiendo del tipo de nodo: de cálculo o de decisión. La fig. I.20 ilustra los dos métodos de traslación. En la fig. I.21 se aplica esta traslación al diagrama de flujo de la fig. I.18 para producir una RP equivalente.

En la fig. I.21 podemos preguntarnos por el significado de los sitios. Una marca que reside en un sitio significa que el contador del programa está listo para ejecutar la siguiente instrucción. Cada sitio tiene una única transición de entrada, excepto los sitios que preceden a las decisiones; estos sitios tienes dos transiciones de salida, correspondientes a los resultados falso y verdadero del predicado de la decisión.

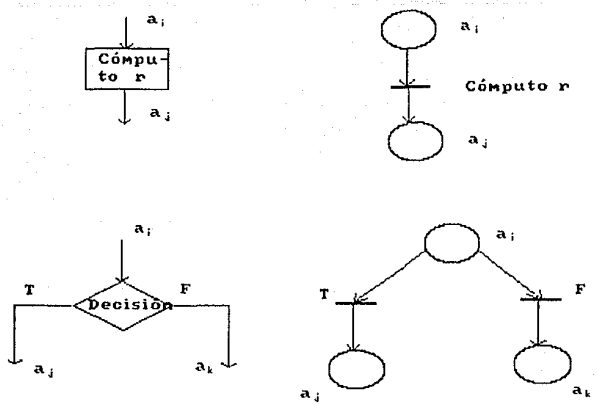


Figura I.20

Las transiciones se asocian con las acciones del programa: los cálculos y las decisiones. Si deseamos interpretar la RP, debemos proporcionar una interpretación para cada transición.

I.7.4.2 Paralelismo.

El paralelismo o la concurrencia puede introducirse en varias formas. Consideremos el caso de dos procesos concurrentes. Cada proceso puede representarse por una RP. Así la RP compuesta, que es simplemente la unión de las RP para cada uno de los dos procesos, puede representar la ejecución concurrente de los dos procesos. El marcaje inicial de la RP compuesta tiene dos marcas, una en cada sitio representando el contador inicial de un proceso.
Esto introduce un paralelismo que no puede representarse en un diagrama de flujo, y que no es muy útil.

Otra aproximación es considerar cómo el paralelismo sería introducido normalmente en un sistema de cómputo. Una de las proposiciones más simples es la que es la que comprende las operaciones "Fork" y "Join". Una operación "Fork" j ejecutada en la posición i, resulta en la continuación del proceso en la posición i+1, y un nuevo proceso se crea con el inicio de la ejecución en la posición j. Una operación "Join" recombinará dos procesos en uno (o equivalentemente destruirá uno de los dos y dejará continuar el otro). Estas operaciones pueden modelarse en una RP como la de la fig. I.22.

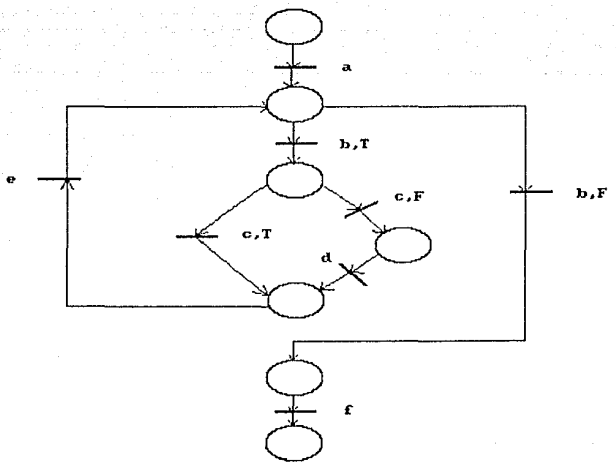


Figura I.21

Otra proposición para introducir el paralelismo es la estructura "parbegin-parend". Esta estructura tiene la forma general "parbegin", S_1, S_2,...,S_n, "parend", donde S_i son instrucciones, $i \in \{1,2,...,n\}$, y sirve para ejecutar las instrucciones en paralelo. En la fig. I.23 tenemos la RP que representa esta estructura.

El paralelismo es útil en la solución de un problema sólo si los procesos componentes pueden cooperar en la solución del problema. Dicha cooperación necesita que se comparta la información y los recursos entre los procesos, lo que debe controlarse para asegurar la correcta operación del sistema en conjunto. A continuación se exponen problemas clásicos de paralelismo, donde la sincronización es fundamental. Estos problemas son: el problema de la exclusión mutua, el problema del productor-consumidor, el problema de los filósofos comensales y el problema de los lectores-escritores.

I.7.4.3 El problema de la exclusión mutua.

Supongamos que varios procesos comparten una variable, registro, archivo u otro ente de datos. Estos datos compartidos pueden utilizarse en varias formas por los procesos, pero en general los usos pueden ser para lectura de los datos o escritura de un nuevo valor de dichos datos. Las operaciones de lectura y escritura son a menudo las únicas operaciones primitivas. Esto significa que, para actualizar los datos compartidos, un proceso debe primeramente leer el valor

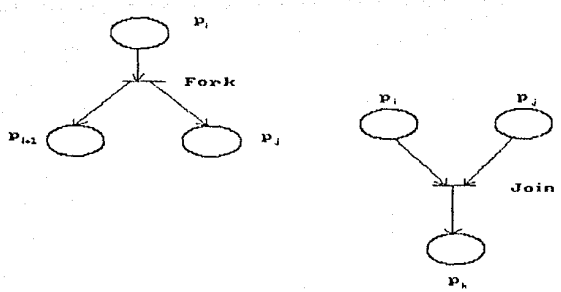


Figura I.22

anterior, después calcular el nuevo valor y finalmente escribir el nuevo valor. Un problema puede ocurrir si dos procesos intentan ejecutar la sucesión de instrucciones al mismo tiempo. Como ejemplo, pensemos en las siguientes instrucciones:

1. El primer proceso lee el valor x de los datos compartidos.
2. El segundo proceso lee el valor x de los datos compartidos.
3. El primer proceso calcula y actualiza el valor x' = f(x).
4. El segundo proceso calcula y actualiza el valor x'' = g(x).
5. El primer proceso escribe x' en los datos compartidos.
6. El segundo proceso escribe x'' en los datos compartidos, destruyendo el valor x'.

El efecto de cálculo del primer proceso se ha perdido, ya que ahora el valor de los datos compartidos es g(x), cuando debería ser g(f(x)) o f(g(x)).

Para prevenir este tipo de problemas, es necesario proporcionar un mecanismo de exclusión mutua. Esta es una técnica para definir la entrada y salida de código, de tal forma que sólo un proceso tiene acceso a los datos compartidos a un tiempo. El código que accede a los datos compartidos, y necesita protegerse de la interferencia de otros procesos, se llama sección crítica. La intención es que un proceso que va a ejecutar su sección crítica, espere hasta que ningún otro proceso esté ejecutando su propia sección crítica. Después cierra el acceso a la sección crítica, previniendo que otro proceso entre a dicha sección. El proceso entra a su sección crítica, la ejecuta, y cuando la deja abre el acceso para permitir que otro proceso entre.

El problema puede ser resuelto con una RP como la de la fig. I.24. El sitio m representa el permiso para entrar a la sección crítica. Para que un proceso

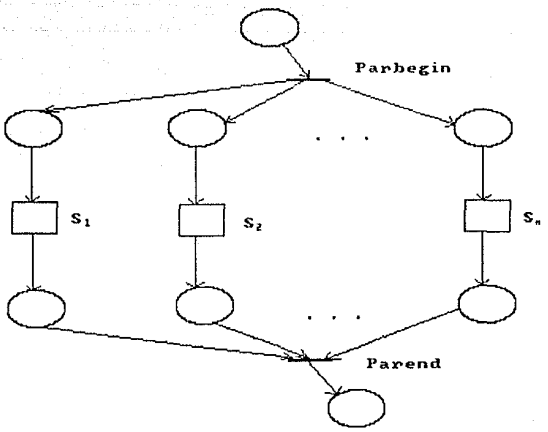


Figura I.23

entre a ésta, debe tener una marca en p_1 o p_2, según el caso, señalando que desea entrar a la sección crítica y debe haber una marca en m dando el permiso para entrar. Si ambos procesos desean entrar simultáneamente, entonces la transición t_1 y t_2 están en conflicto y sólo una de ellas puede encenderse. Al encender t_1 se deshabilitará t_2, haciendo que el segundo proceso espere hasta que el primer proceso salga de su sección crítica y ponga una marca en el sitio m.

I.7.4.4 El problema productor-consumidor.

El problema productor-consumidor también involucra un conjunto de datos compartido, que en este caso será un "buffer" o unidad intermedia de almacenamiento (uia). El proceso del productor crea objetos que son puestos en la uia; el consumidor espera hasta que un objeto se haya puesto en la uia, lo remueve y lo consume. Esto puede modelarse como mostramos en la fig. I.25. El sitio B representa la uia; cada marca en B representa un objeto que se ha producido pero no consumido.

Una variante de este problema es el problema productor-consumidor múltiple. Aquí múltiples productores crean objetos que se instalarán en una uia común a los múltiples consumidores. La fig. I.26 es una solución a este problema. Es la misma que la fig. I.25, excepto que representa s productores y t consumidores; comenzamos el sistema con s marcas en el sitio inicial del proceso del productor, y con t marcas en el sitio inicial del proceso del consumidor.

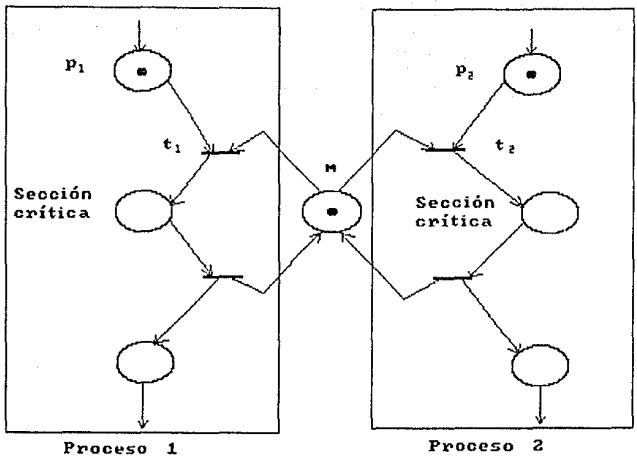


Figura I.24

Otra variante es el problema productor-consumidor con uia acotada. En esta versión, la uia entre el productor y el consumidor tiene sólo n posiciones para objetos. Así el productor no siempre puede crear tan rápido como desee, sino que debe esperar si el consumidor es lento y la uia está llena. La fig. I.27 muestra una solución a este problema. La uia acotada se representa por dos sitios: B representa el número de objetos que se han producido pero que no se han consumido; B' representa el número de posiciones vacías en la uia. Originalmente B' tiene n marcas y B tiene cero. Si la uia se llena, entonces B' tendrá cero marcas y B tendrá n. En este punto, si el productor intenta poner otro objeto en la uia será detenido porque no hay marca en B para habilitar esa transición.

I.7.4.5 El problema de los filósofos comensales.

En este problema tenemos cinco filósofos que comen y piensan alternativamente. Los filósofos están sentados alrededor de un mesa redonda, sobre la cual una gran cantidad de comida china. Entre cada filósofo hay un palillo chino, aunque se necesitan dos palillos para comer. Por lo tanto cada filósofo debe levantar el palillo de la izquierda y el de la derecha. El problema es que si todos los filósofos levantan el palillo de la izquierda y esperan el palillo de su derecha, todos esperarán por siempre y morirán de hambre.

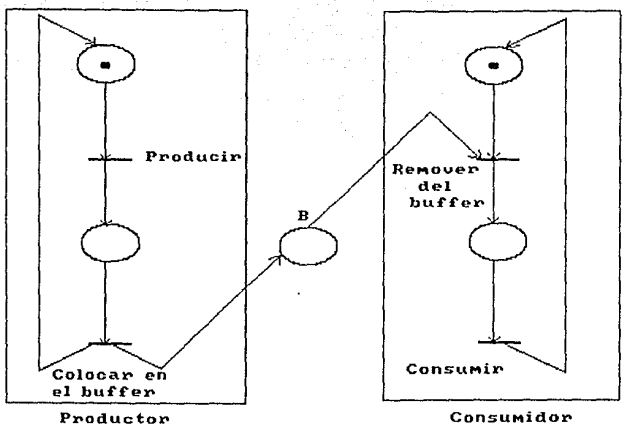


Figura I.25

La fig. I.28 ilustra la RP que soluciona este problema. Los sitios C_1, C_2,..., C_5 representan los palillos, y como cada uno de ellos es inicialmente libre, una marca reside en ellos en la marcaje inicial. Cada filósofo se representa por los sitios M_i y E_i, $i \in \{1,2,3,4,5\}$, que son los estados de meditación y alimentación, respectivamente. Para que un filósofo cambie del estado de meditación al de alimentación, el palillo de la izquierda y el de la derecha deben estar disponibles.

I.7.4.6 El problema de los lectores-escritores.

Aquí los procesos son de lectura y escritura. Todos los procesos comparten un archivo, una variable o una unidad de datos. Los procesos de lectura nunca modifican la unidad, mientras que los de escritura sí lo hacen. Estos últimos deben excluir todos los otros procesos de lectura y escritura, pero varios procesos de lectura pueden tener acceso a los datos simultáneamente. El problema es definir una estructura de control que no llegue a un abrazo mortal -en esta situación los procesos se detienen debido a que ninguno de éllos puede tener disponibles todos los recursos que necesita para trabajar, puesto que los recursos son compartidos por dichos procesos- o que no permita violaciones al criterio de exclusión mutua.

La fig. I.29 muestra una solución, cuando el número de procesos de lectura es acotado por n. En un sistema donde el número de procesos de lectura no es acotado, entonces sólo n lectores pueden leer a un tiempo.

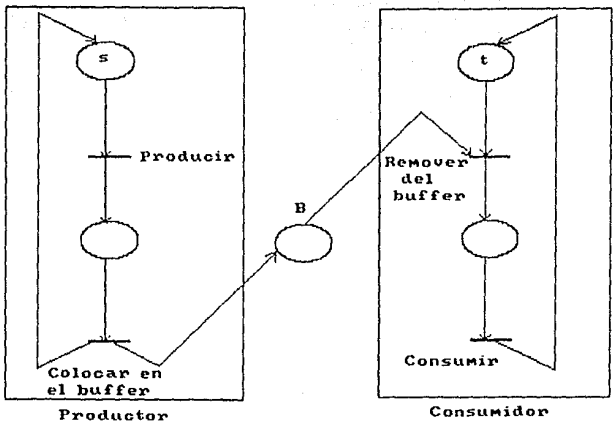


Figura I.26

Un problema ocurre si el número de lectores no es acotado y deseamos permitir que lean simultáneamente. En este caso será necesario que los lectores mantengan un contador del número de ellos que están leyendo. Cada lector suma un uno a su contador cuando comienza a leer, y resta un uno cuando termina de hacerlo. Esto puede modelarse fácilmente con un sitio que tenga un número de marcas igual al número de lectores. Sin embargo, para permitir a un escritor que comience a trabajar, es necesario que el contador esté en cero, o sea, que el sitio correspondiente esté vacío.

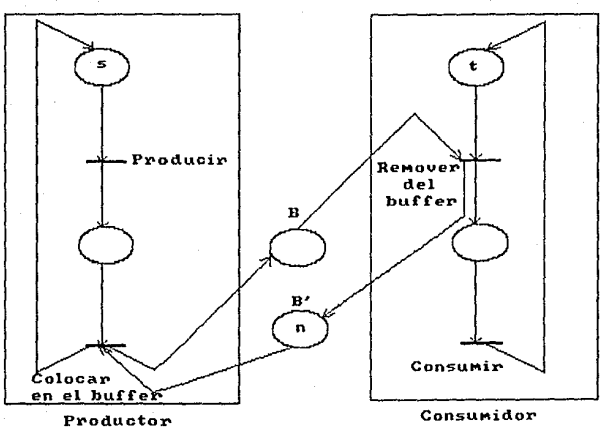

s
Producir
Colocar
en el buffer
Productor
B
B'
n
t
Remover
del
buffer
Consumir
Consumidor

Figura I.27

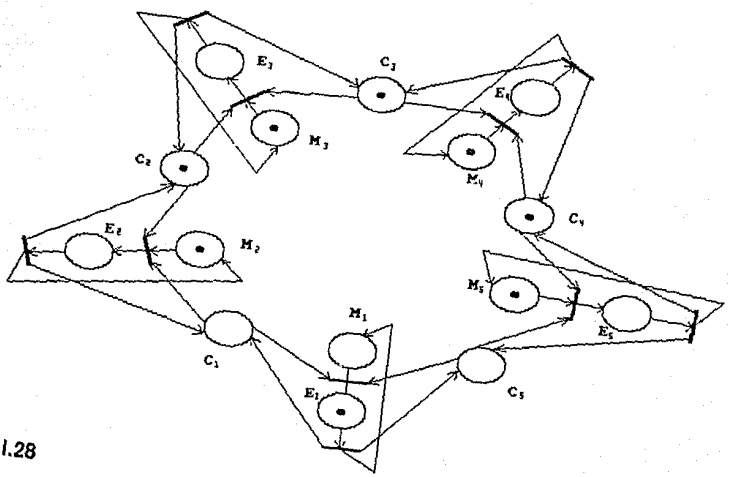


Figura I.28

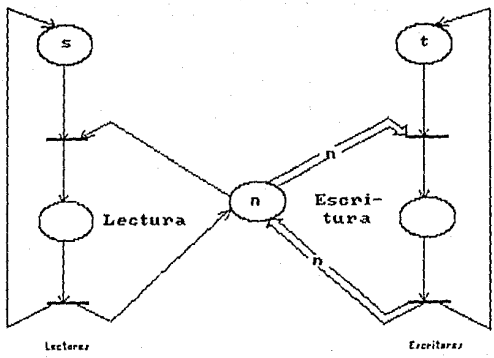


Figura I.29

CAPITULO II

LA PROGRAMACION ORIENTADA A OBJETOS Y SMALLTALK[1]

II.1 Antecedentes.

El término programación orientada a objetos (POO) se deriva del concepto de objeto que aparece en el lenguaje de programación Simula 67[2], que fue desarrollado en Noruega en 1967. En Simula 67 la ejecución de un programa se organiza como la ejecución conjunta de una colección (posiblemente variable) de objetos. La colección como un todo se representa como un objeto del sistema y los objetos que comparten una estructura común constituyen una clase, descrita en el programa por una declaración de clase común.

Sin embargo, el concepto de programación orientada a objetos tiene antecedentes previos, con la introducción de la simulación digital como una herramienta importante para el análisis. Debido a lo anterior es que comenzaron a aparecer una serie de lenguajes de simulación como Simscript, GPSS, CSL, Sol y Simula I.

Aunque Simula I fue diseñado como un lenguaje de simulación (también por investigadores noruegos, en 1962-64), pronto se utilizó como un lenguaje de propósito general. Los procesos podían organizarse como un sistema de ejecuciones de programas interactuantes y este enfoque probó su utilidad en una amplia gama de aplicaciones. En Simula I las clases y objetos de Simula 67 se llamaban actividades y procesos, respectivamente.

El enfoque de POO se utilizó, en sus primeras etapas, en áreas como simulación, organización de concurrencia, programación estructurada, diseño de VLSI y tipos de datos abstractos.

La utilización de la POO se amplió como resultado del desarrollo del lenguaje Smalltalk-80, el cual constituye un lenguaje y un ambiente de programación integrados. Smalltalk fue la primera implementación sustancial, interactiva, basada en información visual, de un lenguaje orientado a objetos (LOO). Smalltalk y sus antecedentes directos Flex, Smalltalk-72 y Smalltalk-76 fueron desarrollados en el

---

[1] Smalltalk/V es la versión de Smalltalk-80 para microcomputadoras compatibles con la IBM PC. Nos referiremos a ambos indistintamente como Smalltalk.

[2] Nygaard, Kristen. Basic Concepts in Object Oriented Programming. SIGPLAN Notices. Vol. 21, N° 10, Octubre 1986, pp 128-134.

Centro de Investigaciones de Palo Alto de Xerox (Xerox PARC). En Smalltalk la ejecución incremental de programas de LISP se combinó con los conceptos de clase-subclase y de procesos virtuales de Simula 67.

También es necesario mencionar como investigadores pioneros de la POO a gente dedicada a la inteligencia artificial, quienes produjeron sistemas orientados a objetos para la ingeniería del conocimiento. Podemos poner como ejemplo a los desarrolladores de Actors, Flavors, Loops, Strobe y Knowledge Engineering Environment. Además, la encapsulación (concepto del que se hablará posteriormente) ha sido la guía en el desarrollo de sistemas de tipos de datos abstractos en lenguajes como Alphard, CLU y Ada.

En la década de los ochentas apareció en el mercado una gran variedad de LOO, entre los que podemos nombrar a C++, Objective-C, CLOS (Common Lisp Object System), Eiffel, Orient 84K y Vulcan, los cuales han contribuido a la popularización de la POO, sobre todo en Estados Unidos, en Japón y en muchos países europeos.

En este capítulo se mostrarán primeramente las características de la POO, continuando con algunos principios de diseño de Smalltalk y posteriormente con definiciones básicas de conceptos de dicho lenguaje. Este hecho no significa que las definiciones de Smalltalk se usen aquí como las de la POO en general, sino que se presentan debido a que el sistema manejador de RP, el cual se describe en el capítulo III, está construido en el ambiente de programación de Smalltalk.

II.2 Características de la POO.

En los últimos años el interés puesto en la POO ha llevado a una gran variedad de definiciones e interpretaciones de ese término. Por esto, es muy difícil establecer el significado de lo que es un lenguaje de programación, un programa o una interfaz de usuario "orientado(a) a objetos".

Los distintos enfoques de la POO tienen en común el uso de la encapsulación en diversas formas. La encapsulación ha sido importante en las ciencias de la computación por la sencilla razón de que es necesario descomponer grandes sistemas en pequeños subsistemas cerrados, los cuales pueden más fácilmente desarrollarse, mantenerse y transportarse. Los LOO formalizan la encapsulación y fomentan la programación en términos de objetos, más que de programas y datos. Cada uno de los enfoques adopta un modelo particular de objeto, dependiendo de las propiedades que de dicho objeto se necesiten encapsular. Una definición completa de lo que se entiende por orientado a objetos no es posible, aunque quizás se puede juzgar cuándo un sistema o un lenguaje es más orientado a objetos que otro.

II.2.1 Encapsulación.

Un objeto consiste de una representación encapsulada, cuyo estado se ve en sus variables privadas, y de un conjunto de mensajes que hacen referencia a métodos, los cuales modifican el estado de los objetos. Uno no puede manipular o ver el estado de los objetos directamente, sino a través del envío de un mensaje, para el cual el objeto mismo selecciona el método con el cual responderá al mensaje.

La encapsulación es importante para asegurar la confiabilidad y la facilidad para modificar los sistemas de software, ya que reduce las dependencias entre los componentes de dichos sistemas. Además, como no se tiene acceso a las variables privadas de un objeto fuera de éste, una interfaz permite cambios a las variables y los métodos sin afectar otras partes del sistema.

Los objetos cuyo estado sólo puede manejarse a través de sus operaciones o métodos se llaman abstracciones de datos. Estas esconden de los usuarios la representación de datos de un objeto. De esta forma los usuarios no tienen que preocuparse sobre cómo se implementó un objeto.

II.2.2 Herencia.

La herencia tiene muchas formas, dependiendo de lo que se desee heredar y de cuando y como la herencia tiene lugar. En la mayoría de los casos, la herencia es estrictamente un mecanismo de reutilización para compartir la conducta entre los objetos.

La herencia de clases se representa a menudo como la característica fundamental que distingue a los LOO de otros lenguajes de programación. Con ésto se sobrevalora la importancia de un solo aspecto de la POO y por lo tanto se devalúa la contribución de otros lenguajes que no proporcionan un mecanismo explícito para la herencia de clases. Sin embargo, la herencia de clases es un mecanismo importante que, cuando se aplica apropiadamente, puede simplificar enormes bloques de software al explotar las similitudes entre ciertas clases de objetos.

La idea clave de la herencia de clases es la de proporcionar un sencillo y poderoso mecanismo para definir nuevas clases que hereden propiedades de las clases existentes. Con la herencia singular, una subclase puede heredar las variables de instancia y métodos de una única clase "progenitora", agregando posiblemente algunos métodos y variables de instancia de su propiedad.

Una extensión natural de la herencia singular es la herencia múltiple, ésto es, la herencia de una subclase de múltiples clases "progenitoras". Algunos LOO que soportan la herencia múltiple son Flavors, Trellis/Owl y Eiffel.

Una variante interesante de la herencia de clases es lo que se llama

herencia parcial. En este caso se heredan algunas propiedades y se suprimen otras. La herencia parcial es conveniente para compartir código, pero puede crear un conflicto en la herencia de clases.

Podemos ver a la herencia de clases como una forma estática de herencia: las nuevas clases heredan propiedades cuando se definen y no en tiempo de corrida. Cuando se define una clase, las propiedades de sus instancias (variables de instancia y métodos) se determinan para todo el tiempo. Por otra parte, llamamos herencia dinámica a los mecanismos que permiten a los objetos alterar su conducta, en el curso normal de interacciones entre dichos objetos. La herencia dinámica se da dentro del modelo de objeto y sólo es posible dentro de sistemas basados en objetos prototípicos. En tales sistemas, un objeto puede tener variables de instancia y métodos, pero también puede delegar ciertos mensajes a un conocido, que se llama objeto prototípico.

II.2.3 Polimorfismo.

La habilidad de distintos objetos para responder en forma diferente al mismo mensaje se llama polimorfismo. Este es en parte responsable de lo que en POO se llama programación diferencial o programación por modificación, con la cual es muy sencillo incorporar nuevos objetos dentro del sistema si ellos responden a los mismos mensajes que los objetos ya existentes.

Una función polimórfica es aquella que puede aplicarse uniformemente a diversos objetos. Por ejemplo, puede utilizarse la misma notación para sumar dos enteros, dos reales, o un entero y un real, o aún para sumar números complejos, y éstos a enteros o reales, suponiendo que el manejo de estas combinaciones está definido. En estos casos, la misma operación mantiene su conducta en forma transparente para diferentes tipos de argumentos.

Por otro lado, la operación "open" puede aplicarse a archivos de datos o ventanas. Aquí estamos tratando con dos operaciones que tienen, por coincidencia, el mismo nombre, aunque su comportamiento es completamente distinto. A ésto se le llama polimorfismo ad hoc. Esta forma de polimorfismo es útil, pero puede llevar a complicaciones si se abusa de élla. Los programadores tienen la responsabilidad de escoger nombres significativos para las operaciones, y evitar la reutilización de nombres que puedan malinterpretarse.

El polimorfismo intensifica la reutilización del software al posibilitar la implementación de métodos genéricos que no sólo funcionarán para un tipo de objetos existentes, sino también para objetos que se agreguen posteriormente. Por ejemplo, una función "sorter" ordenará cualquier lista de objetos que soporten un operador de comparación.

II.3 Principios de diseño de Smalltalk.

El propósito de Smalltalk es proporcionar apoyo computacional para el espíritu creativo en cualquier persona. El trabajo del grupo creador de Smalltalk se concentró en dos áreas de investigación: un lenguaje de descripción que sirviera como una interfaz entre los modelos en la mente humana y aquéllos dentro del hardware, y un lenguaje de interacción que adapte el sistema de comunicación humana al de la computadora.

Un principio fundamental en el proyecto Smalltalk es el siguiente:

1) Si un sistema va a servir al espíritu creativo, debe ser enteramente comprensible por un individuo.

El potencial humano se manifiesta en los individuos. Para encauzar este potencial, se debe proporcionar un medio que pueda ser dominado por un individuo. Cualquier barrera entre el usuario y alguna parte del sistema será una barrera eventual para la creatividad. Cualquier parte del sistema que no pueda cambiarse o que no sea suficientemente general es una fuente de impedimento a dicha creatividad. Si una parte del sistema trabaja en forma diferente de la del resto, esa parte demandará esfuerzo adicional para su control. De lo anterior se llegó al principio general de diseño:

2) Un sistema debe construirse con un conjunto mínimo de partes inalterables; esas partes deben ser lo más generales posible; todas las partes del sistema deben mantenerse en una estructura uniforme.

II.3.1 El lenguaje.

Al diseñar un lenguaje de programación, todo lo que se conoce sobre cómo piensa y se comunica la gente es aplicable. Se deben hacer los modelos de computación compatibles con la mente humana.

Podemos representar a una persona como un ente con cuerpo y mente. El cuerpo es el lugar de la experiencia primera y el canal físico a través del cual el universo se percibe y los propósitos se llevan a cabo. La experiencia se registra y procesa en la mente, por lo que el propósito del lenguaje debe ser proporcionar una estructura de comunicación.

Existen dos tipos de comunicación: la explícita, que son las palabras y movimientos articulados y percibidos; y la implícita, que es la cultura y la experiencia compartidas que forman el contexto de la comunicación explícita. En la interacción humana, mucha de la comunicación se lleva a cabo a través de la referencia al contexto compartido, y el lenguaje humano se construye alrededor de tal alusión.

Este también es el caso de las computadoras. El cuerpo hace un

despliegue visual de la información y percibe la información del usuario. La mente de una computadora incluye la memoria interna y los elementos de procesamiento y su contenido.

3) El diseño de un lenguaje para programación de computadoras debe tratar con modelos internos, con medios externos y con la interacción entre éstos y el ser humano y la computadora.

Smalltalk no es simplemente una mejor manera de organizar procedimientos o una técnica diferente para el manejo del almacenamiento de información. No es sólo una jerarquía extensible de tipos de datos, o una interfaz gráfica del usuario. Es todas esas cosas y cualquiera más necesaria para soportar las interacciones anteriormente descritas.

II.3.2 Los objetos comunicantes.

La mente observa un vasto universo de experiencia, inmediata y registrada. Uno puede derivar un sentido de unidad con el universo simplemente con la experiencia. Sin embargo, si uno desea tomar parte en el universo se deben hacer distinciones. Al hacer ésto uno identifica un objeto en el universo, y al mismo tiempo el resto se convierte en no-ese-objeto. Después viene el acto de referencia, donde podemos asociar un único identificador con un objeto, y a partir de aquí, sólo la mención del identificador es necesaria para referirse al objeto original. Por lo tanto:

4) Un lenguaje de programación de computadoras debe soportar el concepto de objeto y proporcionar un medio uniforme para referirse a los objetos en su universo.

El manejador de almacenamiento de Smalltalk proporciona un modelo orientado a objetos. La referencia uniforme se realiza asociando un entero único con cada objeto en el sistema. Esta uniformidad es importante porque significa que las variables en el sistema pueden tomar valores diferentes y todavía pueden implementarse como celdas de memoria sencillas. Los objetos se crean cuando las expresiones se evalúan y pueden entonces manejarse con referencia uniforme, por lo que ninguna medida es necesaria para su almacenamiento en los procedimientos que manipulan a dichos objetos. Cuando todas las referencias a un objeto han desaparecido del sistema, el objeto mismo desaparece y su almacenamiento es requerido.

5) Para ser verdaderamente orientado a objetos, un sistema de cómputo debe proporcionar un manejo automático del almacenamiento.

Una forma de saber si el lenguaje está trabajando bien es ver si en los programas se nota que se está haciendo lo que se modela. Si los programas están salpicados con instrucciones relativas al manejo del almacenamiento, entonces su modelo interno no está bien adaptado al de los humanos.

Cada objeto en el universo tiene una vida propia. Análogamente, el cerebro se encarga del procesamiento independiente así como del almacenamiento de cada objeto mental.

6) El manejo de los objetos debe verse como una capacidad intrínseca de ellos, la cual puede invocarse uniformemente con el envío de mensajes.

Así como los programas lucen desordenados si el almacenamiento de objetos es tratado explícitamente, el control en el sistema se vuelve complicado si el procesamiento se lleva a cabo en forma extrínseca.

Para efectuar una operación con un objeto, Smalltalk envía el nombre de la operación deseada junto con ciertos argumentos, como un mensaje al objeto, sabiendo que el objeto receptor conoce la mejor forma de llevar a cabo dicha operación. En vez de un triturador de bits que viola y saquea las estructuras de datos, se tiene un universo de objetos bien educados, que cortesmente se solicitan favores entre ellos. La transmisión de mensajes es el único proceso realizado fuera de los objetos, ya que los mensajes viajan entre dichos objetos.

7) Un lenguaje debe diseñarse alrededor de una metáfora que pueda aplicarse en cualquier área.

En Smalltalk, la interacción de los objetos más primitivos se maneja de la misma forma que la interacción al más alto nivel entre la computadora y el usuario. Cada objeto tiene un conjunto de mensajes que forman su protocolo, el cual define la comunicación explícita a la cual el objeto puede responder. Internamente, los objetos pueden tener almacenamiento local y acceso a otra información compartida, los cuales componen el contexto implícito de la comunicación.

II.3.3 La organización.

Varios principios de organización, relacionados entre sí, contribuyen al exitoso manejo de sistemas complejos.

8) Ningún componente en un sistema complejo debe depender de los detalles internos de cualquier otro componente.

Si existen n componentes en un sistema, entonces existirán aproximadamente n^2 dependencias potenciales entre ellos. Si los sistemas de cómputo van a ayudar en las complejas tareas humanas, entonces deben diseñarse para minimizar tales interdependencias. El envío de mensajes proporciona modularidad al desacoplar el propósito de un mensaje (incorporado en su nombre) del método utilizado por el receptor para realizar tal propósito. La información estructural es protegida del mismo modo, porque todo el acceso al estado interno

de un objeto es a través de la misma interfaz de mensajes. La complejidad de un sistema puede reducirse al agrupar componentes similares. Tal agrupación se realiza a través de la tipificación de datos en los lenguajes de programación convencionales, y a través de clases en Smalltalk. Una clase describe ciertos objetos, su estado interno, el protocolo de mensajes que reconocen y los métodos internos para responder a esos mensajes. Los objetos así descritos se llaman instancias de la clase. Aún las mismas clases se ajustan a esta estructura: ellas son instancias de la clase "Class", la cual describe el protocolo apropiado y la implementación para la descripción de los objetos.

9) Un lenguaje de programación debe proporcionar un medio para clasificar objetos similares y para añadir nuevas clases de objetos, al igual que con las clases nucleares del sistema.

Las clases son el mecanismo de extensión en Smalltalk. El sistema creado se utilizará tal como se diseñó. A cada paso del diseño, la persona escogerá naturalmente la representación más efectiva si le es proporcionada.

10) Un programa debe especificar sólo la conducta de los objetos, no su representación.

Un ejemplo del principio anterior es que un programa nunca debe declarar que un objeto es SmallInteger o LargeInteger, por ejemplo, sino sólo responder al protocolo de los enteros.

11) Cada componente independiente debe aparecer en un solo lugar en el sistema.

El principio anterior tiene varias razones. Primero, ahorra tiempo, esfuerzo y espacio si las adiciones o cambios al sistema sólo necesitan hacerse en un lugar. Segundo, los usuarios pueden localizar más fácilmente un componente que les satisfaga una necesidad. Tercero, al no tomar en cuenta el principio anterior, los problemas surgen al sincronizar cambios y al asegurar que los componentes interdependientes son consistentes.

La herencia en Smalltalk fomenta los diseños bien factorizados, i.e., donde el código que ejecutará cierta tarea se encuentra en un solo lugar. Cada clase hereda la conducta de su superclase. Esta herencia se extiende a través de clases cada vez más generales, terminando con la clase Object, la cual describe la conducta por omisión de todos los objetos en el sistema.

12) Cuando un sistema está bien factorizado, se tiene un gran poder para los usuarios e implementadores en la misma forma.

Los beneficios para los implementadores son obvios. Para empezar, habrá menos funciones primitivas que tengan que crear. Por ejemplo, todas las gráficas en Smalltalk se realizan con una sencilla operación primitiva. Con sólo

hacer una tarea, un implementador puede prestar atención a cada instrucción sabiendo que cada mejoría en la eficiencia será ampliada a través del sistema. Es natural preguntarse qué conjunto de operaciones primitivas serían suficientes para soportar un sistema de cómputo completo.

13) La especificación de una máquina virtual establece una estructura para la aplicación de la tecnología.

La máquina virtual de Smalltalk establece un modelo orientado a objetos para almacenamiento, un modelo orientado a mensajes para el procesamiento y un modelo de mapeo de bits para el despliegue visual de la información. A través del uso de microcódigo y hardware, el desempeño del sistema puede mejorarse sustancialmente sin comprometer las otras virtudes del sistema.

II.3.4 La interfaz del usuario.

La interfaz del usuario es simplemente un lenguaje en el cual la mayor parte de la comunicación es visual. Ya que la representación visual se superpone fuertemente con la cultura humana, la estética desempeña un papel muy importante en esta área. Como la capacidad total de un sistema de cómputo es liberada a través de la interfaz del usuario, la flexibilidad también es esencial aquí.

14) Cada componente accesible al usuario debe ser capaz de presentarse en una forma significativa para la observación y la manipulación.

Por definición, cada objeto proporciona un protocolo de mensajes apropiado para interactuar. Este protocolo es esencialmente un microlenguaje para este tipo de objeto. Al nivel de la interfaz del usuario, el lenguaje apropiado para cada objeto en la pantalla se presenta visualmente (como texto, menús, gráficas) y es sensible al teclado y al uso del ratón (mouse, dispositivo de punteo).

Debe notarse que los sistemas operativos no se ajustan a las características anteriores, por lo que Smalltalk incorpora los siguientes componentes de sistema operativo:

a) Manejo del almacenamiento. Es completamente automático. Los objetos se crean mediante un mensaje a su clase y su almacenamiento se recupera cuando no existen más referencias a ellos. La expansión del espacio de direcciones a través de la memoria virtual es transparente.

b) Sistema de archivos. Se incluye en la estructura normal a través de los objetos tales como Files y Directories, con protocolos de mensajes que soportan acceso a los archivos.

c) Manejo de la imagen. La imagen en el monitor es una instancia de la clase Form, la cual es visible continuamente, y los mensajes de manipulación

gráfica definidos en esta clase se usan para cambiar la imagen visible.

d) Teclado. Los dispositivos de entrada son igualmente modelados como objetos, con mensajes apropiados para determinar su estado o leer su historia como una sucesión de eventos .

e) Acceso a subsistemas. Los subsistemas se incorporan naturalmente como objetos independientes dentro de Smalltalk.

f) Corrector de errores (Debugger). Es un subsistema que tiene acceso a manejar el estado de un proceso suspendido. Casi el único error en corrida que puede ocurrir en Smalltalk se debe a un mensaje no reconocido por su receptor.

II.4 Conceptos básicos de Smalltalk.

La perspectiva tradicional de los sistemas de software es que éstos se componen de una colección de datos, que representan cierta información, y un conjunto de procedimientos que manipulan dichos datos. Las acciones se realizan en el sistema al invocar un procedimiento y darle algunos datos a manipular.

Un problema con este punto de vista es que los datos y los procedimientos se tratan como si fuesen independientes, cuando en realidad no lo son. Todos los procedimientos hacen suposiciones acerca de la forma de los datos que ellos manipulan.

En un sistema que funciona correctamente, la selección apropiada de procedimiento y datos siempre se realiza. Cuando el sistema no funciona correctamente, los datos a ser manipulados por un procedimiento pueden ser de una forma completamente distinta de la esperada. Aún en un sistema de funcionamiento correcto, la selección del procedimiento apropiado y los datos debe ser hecha siempre por el programador.

En lugar de dos tipos de entidades que representan la información y su manejo independientemente, Smaltalk tiene un tipo sencillo, el objeto, que representa a ambos. Un objeto es un paquete de información y las descripciones de su manipulación.

Como piezas de datos, los objetos pueden manipularse. Sin embargo, como procedimientos, los objetos describen al mismo tiempo su manipulación. La información se manipula al enviar un mensaje al objeto que representa la información. Un mensaje es la especificación del manejo de un objeto.

Cuando un objeto recibe un mensaje, aquél determina como manipularse a sí mismo. El objeto a ser manipulado se llama el receptor del

mensaje. Un mensaje incluye el nombre simbólico que describe el tipo de manipulación deseada. Este nombre se llama el selector del mensaje. El selector es sólo un nombre para la manipulación deseada; describe lo que el programador quiere que suceda, no cómo debería suceder. El receptor del mensaje contiene la descripción de cómo la manipulación debe ejecutarse.

Los procedimientos también tienen nombres, que se utilizan en las llamadas a dichos procedimientos. Sin embargo, existe sólo un procedimiento por cada nombre, por lo que el nombre especifica el procedimiento exacto a llamar y exactamente lo que debería suceder. Un mensaje, sin embargo, puede interpretarse de diferentes formas por distintos receptores. Así, un mensaje no determina exactamente lo que sucederá; el receptor del mensaje es el que lo hace.

Además de un selector, un mensaje puede contener otros objetos que toman parte en la manipulación. Estos se llaman los argumentos del mensaje.

La descripción de la manipulación de la información de un objeto es una entidad de forma procedural que se llama método. Este, como un procedimiento, es la descripción de una sucesión de acciones que serán ejecutadas por un procesador. Sin embargo, a diferencia de un procedimiento, un método no puede llamar a otro método. En lugar de esto, en un método deben enviarse mensajes. Lo importante es que los métodos no pueden separarse de los objetos. Cuando un mensaje se envía, el receptor determina el método a ejecutar sobre la base del selector del mensaje.

Los objetos lucen en forma distinta por fuera que por dentro. El exterior de un objeto es la apariencia que éste muestra a los otros objetos con los que interactúa. Desde el exterior sólo se puede pedir a un objeto que realice algo (al enviar un mensaje). El interior de un objeto es lo que ve el programa que implementa su conducta. Desde el interior se puede decir a un objeto cómo hacer algo (en un método).

El conjunto de mensajes a los que un objeto puede responder se llama su protocolo. La vista interna de un objeto es algo parecido a un sistema procedural. Un objeto tiene un conjunto de variables que refiere a otros objetos. Estas se llaman variables privadas. También tiene un conjunto de métodos que describen qué hacer cuando un mensaje se recibe. Los valores de las variables privadas desempeñan el papel de los datos y los métodos desempeñan el papel de los procedimientos. Esta distinción entre datos y procedimientos se localiza estrictamente al interior del objeto.

Los métodos, como otros procedimientos, deben saber acerca de la forma de los datos que manipulan directamente. Parte de los datos que un método puede manipular son los valores de las variables privadas del objeto.

Un mensaje debe enviarse a un objeto a fin de encontrar cualquier cosa sobre el objeto. Esto es necesario porque no se quiere que la forma del

interior de un objeto se conozca al exterior de él. La respuesta a un mensaje puede ser cambiar el estado del objeto que recibe dicho mensaje, enviar mensajes a otros objetos, regresar un valor, crear nuevos objetos o todo lo anterior al mismo tiempo.

II.4.1 Las clases y sus instancias.

Smalltalk hace una distinción entre la descripción de un objeto y el objeto mismo. Muchos objetos similares pueden describirse con la misma descripción general. La descripción de un objeto se llama clase y ésta puede describir un conjunto de objetos relacionados. Cada objeto descrito por una clase se llama una instancia de esa clase.

Cada objeto es una instancia de alguna clase. La clase describe las similitudes de sus instancias. Cada instancia contiene la información que la distingue de las otras instancias. Esta información es un subconjunto de sus variables privadas que se llaman variables de instancia. Todas las instancias de una clase tienen el mismo número de variables de instancia. Los valores de las estas variables son diferentes de instancia a instancia.

El software de un objeto (i. e., los métodos que describen su respuesta a los mensajes) se encuentra en su clase. Todas las instancias de una clase utilizan el mismo método para responder a un tipo particular de mensaje. La diferencia en la respuesta de dos instancias distintas radica en la diferencia de sus variables de instancia. Los métodos en una clase utilizan un conjunto de nombres para referirse al conjunto de variables de instancia. Cuando se envía un mensaje, esos nombres en el método invocado se refieren a las variables de instancia del receptor del mensaje. Algunas de las variables privadas de un objeto se comparten por todas las instancias. Estas variables se llaman variables de clase y son parte de la clase.

El programador que desarrolla un nuevo sistema crea las clases que describen los objetos que componen el sistema. En un sistema que es uniformemente orientado a objetos, una clase es un objeto mismo. Una clase sirve para varios propósitos. Proporciona la descripción de cómo se comportan los objetos en respuesta a los mensajes. El procesador que corre un sistema orientado a objetos ve hacia la clase receptora cuando un mensaje se envía a un objeto, para determinar el método apropiado a ejecutar. Para este uso de las clases no es necesario que se representen como objetos, ya que el procesador no interactúa con ellos a través de mensajes. Una clase proporciona una interfaz al programador para interactuar con la definición de los objetos. Para este uso de las clases es extremadamente útil ser objetos, ya que pueden manipularse en la misma forma que las otras descripciones. Las clases son las fuentes de nuevos objetos; la creación de un objeto puede llevarse a cabo con un simple mensaje.

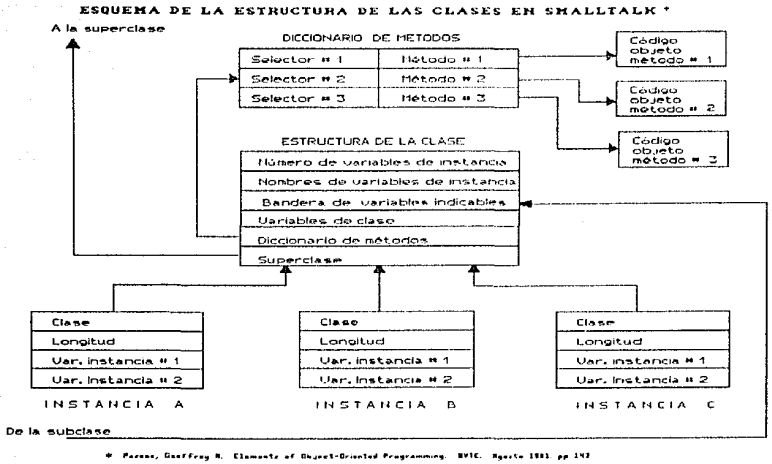


Figura II.1

II.4.2 Herencia.

Un objeto hereda los atributos de otro objeto, cambiando algunos atributos que distinguen a los dos. Algunos LOO permiten la herencia entre todos los objetos, pero la mayoría la permite sólo entre clases, como es el caso de Smalltalk. Una clase puede modificarse para crear otra clase. En tal relación, la primera clase se llama la superclase y la segunda se llama la subclase. Una subclase hereda todas las características de su superclase. Por ejemplo, en una RP la clases Transition y Place son subclases de la clase PetriNode. Las siguientes modificaciones pueden hacerse a una subclase:

1) Añadir variables de instancia.
2) Proporcionar nuevos métodos para algunos de los mensajes comprendidos por la superclase.
3) Proporcionar métodos para nuevos mensajes (mensajes que no entiende la superclase).
4) Añadir variables de clase.

La herencia tiene varias ventajas. Una es que permite la reutilización del código, ya que el código compartido por varias clases puede localizarse en una superclase común y las nuevas clases pueden comenzar teniendo código disponible en la superclase. La herencia también proporciona una manera de organizar y clasificar los objetos, ya que las clases con la misma superclase están muy relacionadas. Otra ventaja de la herencia es que permite extender una clase

mientras el código original permanece intacto. Así, los cambios hechos por un programador tendrán menos probabilidad de afectar el trabajo de otro programador.

En la figura II.1 tenemos un esquema que ilustra los conceptos de Smalltalk anteriormente descritos.

II.5 Ventajas y desventajas de la POO.

La POO ofrece varias ventajas sobre los lenguajes procedurales. Incrementa la flexibilidad al permitir agregar nuevas clases de objetos sin tener que modificar el código existente. La herencia, como ya se dijo, permite que el código sea reutilizable. Con ésto se reduce la cantidad de código y se incrementa la productividad del programador, el cual no comienza a programar desde cero (aunque para el implementador del lenguaje la complejidad en el trabajo es mayor que en otro tipo de lenguajes). Asimismo, la herencia hace que el código que llevará a cabo una tarea específica se encuentre en un solo lugar, lo que facilita el mantenimiento del software.

Entre las desventajas de la POO podemos mencionar la gran cantidad de recursos que consume, sobre todo en los lenguajes donde existe un ambiente de programación integrado, como lo es Smalltalk. Otra desventaja es que la implementación de los LOO es más compleja que en los lenguajes procedurales, debido a que la brecha semántica entre aquellos lenguajes y el hardware es mayor. Además, un programador debe aprender una extensa biblioteca de clases de objetos antes de convertirse en un experto en POO.

CAPITULO III

UN SISTEMA PARA MANEJAR REDES DE PETRI

III.1 Introducción.

Como se puede ver, el capítulo I de este trabajo trata de las RP. Estas constituyen un modelo de sistemas con componentes independientes, asíncronos y concurrentes, que puede aplicarse en áreas del conocimiento como computación (tanto en hardware como en software), ingeniería de control, química, economía, sociología, etc. El primer problema al que se enfrentaría un investigador, en el caso de querer modelar un sistema real con una RP, es precisamente construir esta última. Para esto podría aplicar desde el sentido común hasta técnicas complejas de modelado.

Cuando la RP se ha creado, surge el problema de su manejo, i.e., el dibujar su gráfica, el conocer qué eventos se pueden llevar a cabo (las transiciones que están habilitadas), qué condiciones existen para que se realice cierto evento y qué condiciones aparecen cuando tal evento se realiza (lo que es explicado por las marcas en los sitios de entrada y de salida de una transición), a qué estado llega la RP si se realiza un evento cualquiera o una sucesión de eventos (el marcaje de la RP), etc.

Es este segundo problema, el del manejo de la RP, el que se intenta resolver con el sistema de cómputo que aquí se presenta. Pero ¿por qué con un sistema de cómputo? El manejar una RP implica actividades hasta cierto punto mecánicas, como por ejemplo la comparación de números enteros para saber si una transición está habilitada o no, lo que se convierte en algo laborioso si el número de transiciones y el número de sitios de entrada de cada transición son muy grandes y que puede originar errores en la información dentro del modelo, además del tiempo que hay que invertir para dichas actividades. Por lo anterior es que se decidió crear un sistema de cómputo para el manejo de estructuras básicas de RP (de aquí en adelante, SMRP) como una herramienta interactiva para el investigador que utilice el modelo referido.

El SMRP se ideó para trabajar en microcomputadoras PC o compatibles, ya que son equipos relativamente baratos y con los cuales cuentan la mayoría de las universidades y centros de investigación de nuestro país, así como muchos investigadores en lo personal. Se pensó en la interactividad y la facilidad de uso como características principales del SMRP, además de la fortaleza del mismo para el manejo del modelo de RP, a partir de tener la estructura básica en un archivo con formato ASCII. De esta manera los cambios en el modelo pueden hacerse sin dificultad cambiando la estructura básica, es decir, modificando el

archivo referido, con ahorro de tiempo y recursos.

Fueron varias las razones por las que se decidió utilizar el ambiente de programación de Smalltalk. Una razón fue la posibilidad de trabajar en un ambiente de POO, y sentir en carne propia las ventajas y desventajas de este paradigma de programación que se ha popularizado a gran velocidad. Otra razón poderosa consistió en las ventajas que Smalltalk ofrece para crear sistemas interactivos, donde se puede hacer uso del mouse (que aunque no es necesario, es muy útil), manejando la información en ventanas de distinto tipo (de texto, de gráficas, de listas de elementos). La razón más importante fue la amplia gama de clases de objetos que ofrece Smalltalk, en particular las clases de conjuntos, bolsas y diccionarios (Set, Bag y Dictionary, respectivamente), y la facilidad de crear nuevas clases (las necesarias para manejar las RP) que interactúen con las ya existentes, algo que no puede encontrarse en lenguajes procedurales como C o Pascal. Además, el trabajar con Smalltalk permite que las modificaciones a los métodos de ciertas clases no afecten a las demás clases del ambiente en general, con lo que el SMRP puede ampliarse para manejar estructuras de RP que incorporen nuevos elementos en el modelo (RP coloreadas, por ejemplo).

El hecho de elegir Smalltalk como lenguaje de programación nos obligó a invertir tiempo y esfuerzo en el aprendizaje del lenguaje y del manejo del ambiente y en la investigación sobre las generalidades de la POO, de las cuales trata el capítulo II. Por otra parte, Smalltalk consume bastantes recursos de la computadora en la que se instala, sobre todo de memoria principal, lo que puede considerarse una desventaja en el rendimiento del SMRP. Aún así, consideramos que éste es una herramienta útil, que puede perfeccionarse y ampliarse con facilidad.

En la estructuración de los objetos (más precisamente, en las clases de objetos) se aplicaron los conceptos de herencia y polimorfismo. Se echó mano de la herencia para reutilizar software de algunas clases existentes. Por ejemplo, al tener un explorador de discos (DiskBrowser) en Smalltalk, que presenta en una ventana múltiple los directorios del disco, los archivos de cada directorio y el contenido de los archivos, se creó una subclase de DiskBrowser, PetriNetDiskBrowser, la cual sirve para crear o modificar los archivos en los que se guarda la estructura de la RP. Para ésto se modificaron en PetriNetDiskBrowser algunos de los métodos de DiskBrowser, con lo que se logró tener un explorador de discos apropiado a las necesidades del SMRP. A su vez, la herencia se utilizó para discriminar clases de objetos con características comunes, como es el caso de Transition y Place, que son subclases de PetriNode y que representan las transiciones y los sitios de una RP.

El polimorfismo se aplicó al escribir métodos con el mismo nombre para distintas clases. De esta forma, a dos o más objetos distintos se envían mensajes iguales, que tienen diferente respuesta en cada objeto. Por ejemplo, se puede enviar el mensaje draw a una instancia de PetriNet o a una de PetriNetShowResults, y aunque tienen relación ambos mensajes, la respuesta en

cada clase es distinta.

III.2 Las clases de objetos, sus variables de instancia y sus métodos.

A continuación se describirá la estructura del SMRP, en términos de clases de objetos en Smalltalk. Primeramente se presentarán las clases principales con las que está formado el SMRP, describiendo las variables de instancia y los métodos de cada clase[1]. Después se darán ejemplos de piezas de código en Smalltalk, y la respuesta que da Smalltalk al correr tal código.

La estructura del SMRP está basada en cuatro clases principales: PetriNode, PetriNet, PetriNetShowResults y PetriNetDiskBrowser[2]. Las tres primeras son subclases de la clase Object y la última, como se mencionó en líneas arriba, es subclase de DiskBrowser. PetriNode tiene a su vez como subclases a Place y Transition. También se crearon métodos en algunas clases ya existentes en Smalltalk, como en la clase File, donde se programó el método de clase creaFilePetriNet, gracias al cual se lee la estructura de la RP de un archivo en formato ASCII.

III.2.1 La clase PetriNode.

PetriNode constituye lo que en Smalltalk se llama una clase abstracta. PetriNode no tiene instancias, pero los métodos y las variables de instancia que en ella se definen se utilizan en sus subclases Place y Transition.

Variables de instancia.

1) name - Es una instancia de la clase String, y consta de una cadena de caracteres (Por ejemplo: 't5', 'p12'). Se utiliza como el nombre del nodo.

2) position - Es una instancia de Point. En position se almacenan las coordenadas, en la pantalla del monitor, del punto medio del nodo. Los límites de las coordenadas son 640 en las abscisas y 480 en las ordenadas[3]. (Por ejemplo: 100@200. En Smalltalk el símbolo @ se utiliza como separador). Hay que notar que, en la pantalla, el origen (0@0) se encuentra en el ángulo superior derecho y que las abscisas crecen hacia la derecha y las ordenadas hacia abajo.

3) randompos - También es una instancia de Point. Contiene las

---

[1] En realidad sólo se presentarán los selectores de los métodos con su argumento, cuando así sea el caso, y se describirá la respuesta del método respectivo. En el apéndice B se presentan algunos métodos de las clases principales que forman el SMRP.

[2] Se escribirán los nombres de clases, variables de instancia y métodos tal y como aparecen dentro de Smalltalk.

[3] A decir verdad, los límites de las coordenadas están en función del tipo de monitor con el que se esté trabajando.

coordenadas del punto de inicio o de fin del arco o arcos que comunican a los nodos, dependiendo de si el arco sale del nodo o llega a él.

Métodos de instancia.

1) name - Da como respuesta el valor de la variable name.

2) name: aString position: aPoint - Asigna aString, una cadena de caracteres, a la variable name, y aPoint, un punto, a la variable position.

3) position - Da el valor de la variable position.

4) printOn: aStream - Presenta en la pantalla el nombre del nodo, tomado de la variable name.

5) randompos - Responde el valor de la variable randompos.

6) randompos: aPoint - Asigna aPoint, un punto, a la variable randompos.

III.2.1.1 La clase Place.

En Place se ubica la estructura de los sitios o lugares de una RP. Como se dijo antes, Place hereda las variables de instancia y los métodos de PetriNode.

Variables de instancia.

1) mark - Es un entero, instancia de la clase Integer, el cual representa la marca del sitio.

Métodos de instancia.

1) draw - Dibuja un círculo cuyo centro está almacenado en la variable position. Dentro del círculo se representa la marca con puntos, si la marca es menor o igual que tres, o con un número, si la marca es mayor que tres. Además, escribe sobre el círculo el nombre del sitio, almacenado en la variable name.

2) mark - Su respuesta es el valor de la variable mark.

3) mark: anInteger - Coloca anInteger, un entero, como valor de la variable mark.

4) redraw - Realiza el mismo proceso de draw, sólo que con un nuevo valor de mark.

III.2.1.2 La clase Transition.

Aquí se almacena la estructura de las transiciones. Como Place, Transition hereda los métodos y variables de instancia de PetriNode.

Variables de instancia.

1) ang - Almacena el valor del ángulo de inclinación de la barra que representa la transición.

Métodos de instancia.

1) ang - Responde el valor de la variable ang.

2) ang: anInteger - Asigna anInteger, un entero, a la variable ang.

3) drawrot - Dibuja una barra, que representa la transición, con un ángulo de inclinación igual al valor de la variable ang.

En el ejemplo siguiente, que consta de código en Smalltalk, se crearán instancias de Place y Transition y se enviarán algunos mensajes a dichas instancias[4]. Los comentarios se escriben entre comillas.

```
"Variables temporales"
|p t|
"Creación de instancias de Place y Transition y asignación de ellas a
las variables temporales"
p := Place new.
t := Transition new.
"Nombre y posición del sitio y de la transición"
p name: 'p1' position: 100@250.
t name: 't1' position: 200@350.
"Marca del sitio"
p mark: 4.
"Angulo de inclinación de la transición"
t ang: 0.
p name                                    'p1'
```

Si corriésemos el código anterior, la respuesta que obtendríamos sería la que aparece subrayada a la derecha, i.e., el nombre de la transición p. Si la última línea (p name) la cambiáramos por la siguiente, obtendríamos la respuesta que se muestra a la derecha:

```
t position                                 200@350
```

[4] Todos los ejemplos donde aparece código de Smalltalk se deben correr en el ambiente de programación de dicho lenguaje, donde igualmente se obtienen las respuestas a los mensajes enviados a los distintos objetos.

III.2.2 La clase PetriNet.

PetriNet es la clase principal dentro del SMRP. En ella se almacena la estructura básica de una RP: sus sitios, transiciones, marcaje y funciones de entrada y salida. Además en ella se definen los métodos que cambian el estado de la RP.

En algunos métodos de PetriNet se envían mensajes a instancias de Transition y Place, con lo que se puede realizar un manejo completo de la RP.

Variables de instancia.

1) input - Es una instancia de la clase Dictionary. Se compone de parejas formadas por un índice, que es el nombre de cada transición, y por un valor, que en este caso es la bolsa de sitios de entrada correspondiente a la transición cuyo nombre aparece en el índice.

2) output - También es una instancia de Dictionary. Los índices son los nombres de las transiciones y los valores son las bolsas de sitios de salida de la transición respectiva.

3) marking - Es una instancia de Matrix y guarda una matriz de dimensión 1 x n, donde n es el número de sitios de la RP. El elemento 1j de marking es el valor de la marca del j-ésimo sitio de la RP, donde $j \in \{1,...,n\}$.

4) m - Es una instancia de la clase Integer y representa el número de transiciones de la RP.

5) n - También es una instancia de Integer y contiene el número de sitios de la RP.

6) places - Es una instancia de Array y su dimensión es igual al número de sitios n, ya que contiene a éstos.

7) transitions - También es una instancia de la clase Array. Contiene las transiciones de la RP receptora, por lo que su dimensión es m.

Métodos de instancia.

1) chanMarking: aMatrix - Cambia los valores de la variable de instancia marking por los de aMatrix.

2) chanMarkString - Convierte el marcaje de la RP receptora, contenido en marking, en una instancia de String, es decir, una cadena de

caracteres, dando como respuesta dicha cadena.

3) connectInp: nodeA to: nodeB - Agrega nodeB, que es un sitio, a la bolsa de entrada de nodeA, la cual es una transición.

4) connectOut: nodeA to: nodeB - Añade nodeB, un sitio, a la bolsa de salida de nodeA, una transición.

5) draw - Dibuja la RP receptora. Por cada transición, dibuja ésta, sus arcos y sitios de entrada. Después dibuja sus arcos y sitios de salida.

6) enable: aString - Responde verdadero o falso, dependiendo de si una transición, cuyo nombre es aString, está o no habilitada en el marcaje actual de la RP receptora.

7) enabledTransitions - Nos da el conjunto de nombres de las transiciones habilitadas en el marcaje actual de la RP receptora.

8) initialize - Crea las variables input y output como instancias de la clase Dictionary.

9) input - Da como respuesta la variable de instancia input.

10) marking - Responde la variable de instancia marking.

11) markMatrix: aDictionary - Crea la variable marking como instancia de Matrix y guarda en ella los valores de aDictionary. Además da como respuesta dicha variable marking.

12) matrixD - Produce y da como respuesta la matriz d, de dimensión m x n, tal que $d_{ij} = - \#(p_i,I(t_j)) + \#(p_i,O(t_j))$, i.e., d_{ij} es la suma del número de ocurrencias del sitio p_i en la bolsa de entrada de la transición t_j, multiplicada por -1, más el número de ocurrencias de p_i en la bolsa de salida de t_j, $i \in \{1,...,n\}$, $j \in \{1,...,m\}$.

13) matrixDinp - Crea y responde la matriz dinp, de dimensión m x n, tal que $dinp_{ij} = \#(p_i,I(t_j))$.

14) matrixDout - Produce y da como respuesta la matriz dout, tal que $dout_{ij} = \#(p_i,O(t_j))$.

15) nextMarking: aString - Da el marcaje que se obtiene al disparar la transición cuyo nombre es aString. Si la transición no está habilitada, así lo hace saber.

16) nextSeqMarking: aString - Responde el marcaje que se obtiene al disparar la sucesión de transiciones cuyos nombres están en aString (los nombres

deben estar separados por una coma). Si alguna transición en dicha sucesión no está habilitada, se advierte del caso.

17) occurr: nodeB bagInp: nodeA - Da el número de ocurrencias del sitio nodeB en la bolsa de entrada de la transición nodeA.

18) occurr: nodeB bagOut: nodeA - Responde el número de ocurrencias de nodeB en la bolsa de salida de la transición nodeA.

19) occurr: nodeB inp: nodeA - El número de ocurrencias de nodeB en la bolsa de entrada de nodeA lo escribe en el punto medio del arco de entrada de nodeB a nodeA. Lo anterior se realiza en la gráfica de la RP receptora.

20) occurr: nodeB out: nodeA - Como en el método 20, el número de ocurrencias de nodeB en la bolsa de salida de nodeA se escribe en el punto medio del arco de salida de nodeA a nodeB.

21) output - Responde la variable de instancia output.

22) places - Da la variable de instancia places, donde se almacenan los sitios de la RP receptora.

23) places: anArray - Crea y da como respuesta el arreglo places, donde guarda los valores del arreglo anArray.

24) redraw - Dibuja los sitios de la RP receptora con su marca actual. Lo anterior se realiza cuando hay un cambio en el marcaje de la RP.

25) transitions - Responde la variable de instancia transitions, donde se guardan las transiciones de la RP receptora.

26) transitions: anArray - Produce y da como respuesta el arreglo transitions, donde guarda los valores del arreglo anArray.

Para ejemplificar, a continuación se muestra una estructura sencilla de RP.

$P = \{p_1, p_2\}$.
$T = \{t_1, t_2\}$.
$I(t_1) = \{p_1\}$.
$O(t_1) = \{p_2\}$.
$I(t_2) = \{p_2\}$.
$O(t_2) = \{p_1\}$.
$\mu = [1,0]$.

El siguiente es el código en Smalltalk para crear la RP anterior.

```
"Variables temporales"
|p1 p2 t1 t2|

"Creación de la RP y de los diccionarios input y output.RedPetri es
una variable global y se mantiene en Smalltalk hasta que se indique lo
contrario."
RedPetri := PetriNet new initialize.

"Creación de los sitios, con nombre, posición y marca"
p1 := (Place new) name: 'p1' position: 100@200; mark: 1.
p2 := (Place new) name: 'p2' position: 200@200. mark: 0.

"Creación de las transiciones, con nombre, posición y ángulo de
inclinación"
t1 := (Transition new) name: 't1' position: 150@150; ang: 15700.
t2 := (Transition new) name: 't2' position: 50@250; ang: 15700.

"Asignación de las funciones de entrada y salida como diccionarios de
bolsas"
redPetri connectInp: t1 to: p1;
        connectOut: t1 to: p2:
        connectInp: t2 to: p2;
        connectOut: t2 to: p1.
redPetri draw.
```

Con el mensaje de la última línea se dibuja la RP, según la posición que se asignó a cada nodo.

III.2.3 La clase PetriNetDiskBrowser.

Como ya se mencionó en III.1 y III.2, PetriNetDiskBrowser es subclase de DiskBrowser y nos proporciona la ventana múltiple de un explorador de discos (ya sea discos duros o flexibles), en los cuales se podrá localizar el archivo que contiene la estructura de la RP. Algunos métodos de DiskBrowser se modificaron para tener la ventana del tamaño total de la pantalla, para presentar los menús en español y para indicarle al SMRP el momento en que se desea comenzar a manejar la RP. Esto último se realiza por medio de la clase PetriNetShowResults, la cual veremos posteriormente.

La ventana múltiple consta de tres subventanas, dos superiores y una inferior. En la ventana superior izquierda se muestra la lista de directorios que existen en el disco; en la ventana superior derecha se enlistan los archivos que contiene el directorio seleccionado; por último, la ventana inferior es un editor de texto, donde se puede crear o modificar un archivo.

Las variables de instancia que se enlistan a continuación son heredadas de DiskBrowser, excepto la variable pN, la cual es propia de

PetriNetDiskBrowser y que almacena una RP.

Variables de instancia.

1) allFileMenu - Contiene el menú del editor de texto cuando un archivo se ha leído completamente.

2) contentsPane - Almacena la instancia de TextPane (un panel para manejo de texto), donde se despliega el contenido de un archivo.

3) device - Guarda el carácter que define a la unidad de disco.

4) directoryList - Es una instancia de OrderedCollection, la cual contiene cadenas de caracteres que describen la jerarquía de los directorios.

5) noFileMenu - Es el menú del editor de texto cuando no se ha leído ningún archivo.

6) partFileMenu - Es el menú del editor de texto cuando sólo una parte del archivo se ha leído.

7) pathNameArray - Es una instancia de Array. Cada entrada del arreglo contiene el nombre de la trayectoria completa de un directorio del disco.

8) pN - Es una instancia de PetriNet y se obtiene al leer el archivo que contiene la estructura de la RP.

9) selectedDirectory - Contiene una instancia de Directory o cuando no se ha seleccionado ningún directorio, el objeto nil.

10) selectedFile - Es una instancia de String y contiene el nombre del archivo seleccionado.

11) sortCriteria - Contiene un bloque que describe cómo ordenar los archivos en un directorio: por nombre, tamaño o fecha de creación.

12) sortedFileList - Es una instancia de OrderedColllection, cuyos elementos son arreglos. Cada uno de éstos tiene cuatro elementos: el nombre, el tamaño, la fecha y hora de creación y los atributos de un archivo.

13) sortPane - Contiene una instancia de ListPane, i. e., un panel de lista, donde se describe el criterio de ordenamiento de los archivos.

14) volumeLabel - Es una cadena de caracteres que representa la etiqueta del disco.

15) wholeFileRequest - Es una variable booleana. Es verdadera si el

archivo completo se despliega en el panel de texto y falsa si sólo se muestran las partes inicial y final del archivo.

De los siguientes métodos de instancia sólo beginPn y petriNet son nuevos. Los restantes son métodos de DiskBrowser y se han modificado para presentar los menús en español.

Métodos de instancia.

1) beginPn - Copia el contenido del panel de texto en el archivo seleccionado y abre la ventana gráfica donde se comienza a ejecutar la RP.

2) changeFileMode - Pregunta por los nuevos atributos de un archivo y hace los cambios respectivos.

3) createFile - Pregunta por el nombre de un archivo y lo crea en el directorio seleccionado.

4) directorySort - Da un arreglo que contiene el texto que describe el criterio de ordenamiento.

5) fileListMenu - Responde el menú de la ventana que muestra la lista de archivos.

6) label - Responde una cadena de caracteres como la etiqueta de la ventana múltiple.

7) openOn: driveCharacter - Abre la ventana múltiple del explorador en el disco identificado por driveCharacter. La ventana se abre del tamaño total de la pantalla.

8) petriNet - Nos da la RP que se ha creado con el método beginPn.

9) promptForPathName - Pregunta por una trayectoria y crea un directorio con ella.

10) sortMenu - Responde el menú de ordenamiento de archivos.

12) textMenuInit - Crea los menús contenidos en las variables de instancia noFileMenu, allFileMenu y partFileMenu.

III.2.4 La clase PetriNetShowResults.

PetriNetShowResults sirve como intermediario entre el usuario del SMRP y la clase PetriNet. Permite desplegar una ventana gráfica del tamaño de la pantalla, donde se dibuja la gráfica de la RP, además de presentar un menú para solicitar algún resultado sobre la ejecución de la RP, lo que se despliega en

ventanas de texto en la parte superior derecha de la pantalla.

Variables de instancia.

1) form - Es una instancia de la clase Form, donde se almacena la matriz de bits para dibujar puntos en la pantalla.

2) graphPane - Es una instancia de GraphPane, o sea, un panel para manejo de gráficas.

3) petriNet - Contiene una instancia de PetriNet.

Métodos de instancia.

1) enabledTrans - Presenta, dentro de una ventana de texto, el conjunto de transiciones habilitadas en el marcaje actual.

2) graph: aRect - Crea el área de graficación con las dimensiones de aRect, un rectángulo, y dibuja la gráfica de la RP.

3) graphMenu - Nos da el menú de la ventana gráfica.

4) input - Pregunta por una transición. Después abre una ventana de texto donde muestra la función de entrada de dicha transición.

5) marking - Abre una ventana de texto y presenta el marcaje actual de la RP.

6) motor: aDepth in: aCurrent text: aTextEditor - Escribe en aTextEditor los marcajes que alcanza la RP y las transiciones habilitadas en dichos marcajes, desde el origen aCurrent hasta la profundidad aDepth.

7) nextMarking - Pregunta por una transición y si ésta se encuentra habilitada en el marcaje actual de la RP, muestra en una ventana de texto el marcaje que se alcanza al encender la transición referida.

8) nextMarkingSeq - Inquiere sobre una sucesión de transiciones y en una ventana de texto muestra los marcajes alcanzados al encender cada una de áquellas, o informa si alguna de las transiciones no está habilitada.

9) openIn: aPetriNet - Crea la ventana gráfica del tamaño de la pantalla, dibujando despúes la gráfica de la RP.

10) output - Pregunta por una transición y muestra su función de salida en una ventana de texto.

11) petriNet: aPetriNet - Asigna aPetriNet a la variable de instancia

petriNet.

12) places - Presenta el conjunto de sitios de la RP en una ventana de texto.

13) redraw - Dibuja el marcaje en la gráfica de la RP cuando se ha hecho algún cambio en él.

14) transitions - En una ventana de texto presenta el conjunto de transiciones.

15) tree - Abre una ventana de texto donde se presentarán los resultados del método motor: in: text:.

III.2.5 Métodos creados en clases de objetos ya existentes.

Los siguientes son métodos que se crearon en clases que ya existen en Smalltalk y que se utilizan en el funcionamiento del SMRP.

1) Clase File

creaPetriNet: aString - Lee el archivo cuyo nombre es aString, el cual contiene la estructura básica de una RP. Crea las instancias de Transition, Place y PetriNet necesarias para el manejo de la RP. Es un método de clase y no un método de instancia.

2) Clase Stream

nextNode - Lee de un flujo de caracteres la información referente a un nodo (transición o sitio) o marca de sitio, y responde una cadena de caracteres que contiene dicha información.

3) Clase Character

isAlphaNumericNode - Responde verdadero a la pregunta de si el receptor es un carácter del 0 al 9, una letra de la "a" a la "z" o de la "A" a la "Z", o alguno de los siguientes caracteres: "(", ")", "@" o ",".

4) Clase Integer

chanArrayChar - Nos da el arreglo de caracteres que componen el número entero receptor.

chanChar - Responde el carácter que corresponde al receptor, sólo en caso de que el entero sea de un dígito.

chanString - Nos da la cadena de dígitos que componen al receptor.

5) Clase ScreenDispatcher

petriNetEnv - Muestra la carátula del SMRP, pregunta por el nombre del disco donde se va a trabajar y abre la ventana múltiple del explorador de discos.

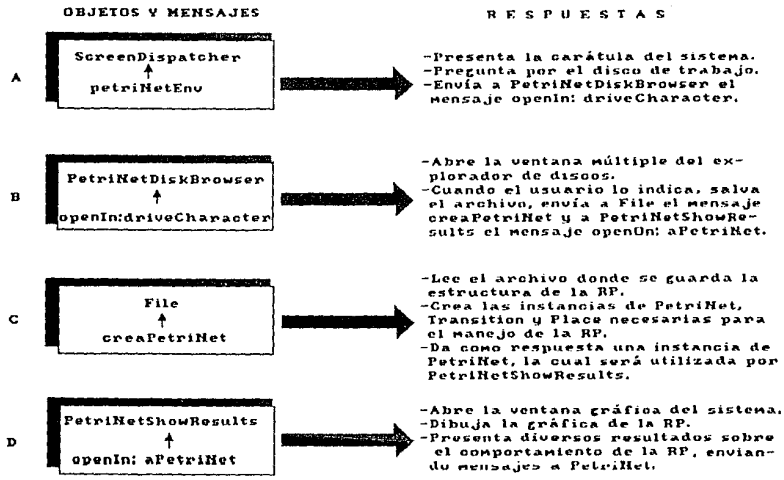


Figura III.1

III.3 Interacción de las distintas clases de objetos.

En la figura III.1 se explica la interacción de las clases de objetos que componen el SMRP. En las siguientes líneas se hará la explicación más detalladamente.

A) Cuando en el menú principal de Smalltalk se elige la alternativa de utilizar el SMRP, la primera clase que entra en acción es ScreeDispatcher, ya que una instancia de ésta es receptora del mensaje petriNetEnv. La respuesta a dicho mensaje consta de lo siguiente:

1) Se presenta la carátula del SMRP.

2) Se hace la pregunta sobre el disco de trabajo, ésto es, donde se almacena o almacenará el archivo que contiene la estructura de la RP.

3) Se envía el mensaje openOn: driveCharacter a una instancia de PetriNetDiskBrowser.

B) En este momento, una instancia de PetriNetDiskBrowser aparece en escena, actuando de la siguiente forma:

1) La respuesta al mensaje openOn: driveCharacter es la apertura de una ventana múltiple que permite la exploración algún disco, ya sea flexible o duro. Además hace posible el uso de un editor de texto para manejar la información de archivos.

2) En el editor antes mencionado se puede escoger de un menú la alternativa de ejecutar la RP. Al hacerlo así, se guarda el archivo de trabajo en el disco y se envían los mensajes creaPetriNet a la clase File y openOn: aPetriNet a una instancia de PetriNetShowResults.

C) La clase File, al recibir el mensaje creaPetriNet, responde de la siguiente manera:

1) Lee el archivo donde se guarda la estructura de la RP. Aquí se analiza cada línea del texto para detectar las cadenas de caracteres que representan a los sitios, las transiciones, el marcaje y las funciones de entrada y salida de una RP.

2) Crea las instancias de PetriNet, Transition y Place necesarias para el manejo de la RP. Las variables de instancia de cada clase se crean con los valores obtenidos al leer el archivo de datos.

3) Responde una instancia de PetriNet, la cual contiene en sus variables de instancia toda la información referente a la RP que se manejará. Dicha instancia se utilizará por PetriNetShowResults.

D) Una instancia de PetriNetShowResults responde al mensaje openOn: aPetriNet en los siguientes términos:

1) Abre la ventana gráfica del SMRP del tamaño de la pantalla.

2) Dibuja la gráfica de la RP de acuerdo a los datos de localización de los nodos que proporciona el usuario.

3) Mediante un menú se presentan, en ventanas de texto, diversos resultados sobre la ejecución de la RP. Para ésto, la instancia de PetriNetShowResults envía mensajes a la instancia de PetriNet que tiene almacenada en una variable de instancia.

III.4 Especificaciones del archivo que contiene la estructura de la RP.

Consideramos de suma importancia establecer las especificaciones del archivo que contiene la estructura de la RP a manejar, el cual puede crearse o editarse utilizando el editor de texto de la ventana múltiple del SMRP (ver III.5.3.3),

aunque también puede usarse un editor de texto cualquiera.

La estructura de la RP consta de los conjuntos de sitios, de transiciones, el vector de marcaje y las funciones de entrada y salida para cada transición.

El archivo debe escribirse de la siguiente forma:

a) Conjunto de sitios.

El primer elemento debe ser el conjunto de sitios, los que se escriben separados por comas y en orden creciente respecto a su índice. El conjunto está delimitado por los caracteres { y }, y puede escribirse en más de un renglón, siempre y cuando un sitio esté especificado en el mismo renglón.

Los elementos que conforman a un sitio son su nombre y su posición en la pantalla. El nombre consta de la letra p y un entero, y la posición se forma con dos enteros separados por el carácter @ y escritos entre paréntesis. Por ejemplo: p5(250@240).

El carácter "." establece el fin de la especificación del conjunto de sitios. Verbigracia:

{p5(100@240), p2(420@160), p3(420@350),p4(580@240),
p5(250@240) }.

b) Conjunto de transiciones.

En segundo lugar debe aparecer el conjunto de transiciones, con las mismas especificaciones del conjunto de sitios, excepto que las transiciones llevan en el nombre la letra t y deben tener además de la posición una inclinación. Por ejemplo: t3(490@320@7853).

Un conjunto de transiciones podría ser el siguiente:

{t1(190@240@15700), t2(300@240@15700), t3(490@320@7853), t4(490@240@15700) }.

c) Vector de marcaje.

El marcaje debe especificarse en un vector de enteros, cuya dimensión es igual a la cardinalidad del conjunto de sitios. El vector se delimita por los caracteres "[" y "]" y así podemos tener:

[4,5,0,1,2].

d) Funciones de entrada y salida.

Para cada transición, en orden creciente respecto a su índice, debe escribirse primero la función de entrada y después la de salida. La primera se denota por la letra I y la segunda por la letra O.

De esta manera, para la transición t1 sus funciones de entrada y salida pueden ser:

I(t1) ={p1}.
O(t1)={p2,p3,p3,p5}.

Si alguna bolsa de entrada o salida es vacía, se especificará de la siguiente forma:

I(t12) = { }.

La especificación completa de una RP puede ser la sucesiva:

{p1(100@240), p2(420@160), p3(420@350),p4(580@240),
p5(250@240) }.
{t1(190@240@15700), t2(300@240@15700), t3(490@320@7853),
t4(490@240@15700) }.
[4,5,0,1,2].
I(t1) ={p1}.
O(t1)={p2,p3,p3,p5}.
I(t2)={p2,p3,p5}.
O(t2)={p5}.
I(t3)={p3}.
O(t3)={p4}.
I(t4)={p4}.
O(t4)={p2,p3,p3}.

Es muy importante terminar la especificación de cada elemento con el carácter ".".

III.5 Manual de uso del SMRP.

Para utilizar el SMRP se debe entrar previamente al ambiente de trabajo de Smalltalk. El procedimiento de instalación de Smalltalk, o del SMRP si es que ya se tiene el primero, se puede encontrar en el apéndice C, donde también se detallan los requerimientos de hardware y software.

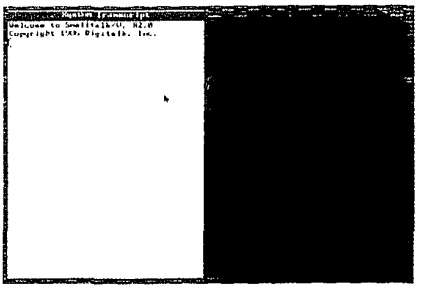


Figura III.2

III.5.1 Ingreso a Smalltalk.

Después de haber instalado Smalltalk, la instrucción para ingresar a su ambiente de trabajo, en el directorio que lo contiene, es:

v

A continuación aparecerá la pantalla que se muestra en la figura III.2. El cursor tiene forma de flecha y sirve para apuntar en la pantalla. Se puede mover el cursor con el mouse o las teclas →, ←, ↑ y ↓.

III.5.2 Manejo de los menús.

Como Smalltalk es un sistema ayudado por menús, el SMRP también lo es. Esto significa que las instrucciones se dan al seleccionar alternativas de los menús.

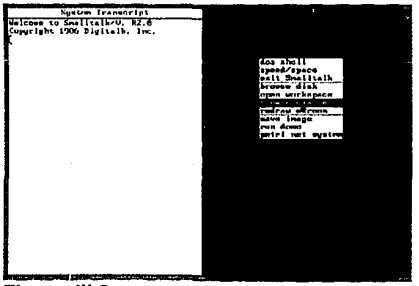


Figura III.3

Cada ventana o panel tiene uno o varios menús, los cuales aparecen al oprimir el botón derecho del mouse (o presionando la tecla Del o Supr). Si no se quiere escoger alguna alternativa, el menú desaparece moviendo el cursor hacia afuera del menú y oprimiendo cualquier botón del mouse (o moviendo el cursor con las teclas →, ←, ↑ o ↑).

Se selecciona una alternativa de los menús moviendo el cursor dentro de ellos. Cuando se ha marcado la alternativa deseada se debe presionar el botón izquierdo del mouse (o la tecla + del teclado numérico) y el proceso seleccionado se ejecutará.

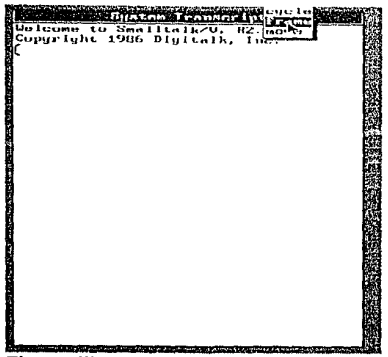


Figura III.4

El tipo de menú que se obtiene depende del lugar donde se halla el cursor. Si éste se coloca fuera de cualquier ventana, aparecerá el menú principal de Smalltalk (figura III.3). Si el cursor se encuentra sobre la etiqueta de una ventana -en la parte superior- se obtendrá un menú (figura III.4), distinto del que aparece dentro de la ventana (figura III.5).

III.5.3 Ingreso al SMRP.

Para comenzar a trabajar con el SMRP, se debe invocar el menú principal de Smalltalk (ver III.5.2) y escoger la alternativa "petri net system", como puede verse en la figura III.6.

Al principio aparecerá la carátula del sistema, donde se necesita oprimir el botón izquierdo del mouse (o la tecla + del teclado numérico). Posteriormente se mostrará la pantalla de la figura III.7, donde deberá escribirse la letra que designe la unidad de disco (a,...,f), la cual almacena o almacenará el archivo de la estructura de la RP -detallado en III.5- y después se oprimirá Return.

Si no se escribe alguna letra y se oprime Return, el sistema no trabajará y volverá a aparecer la pantalla de la figura III.2 o la última pantalla anterior al ingreso al SMRP.

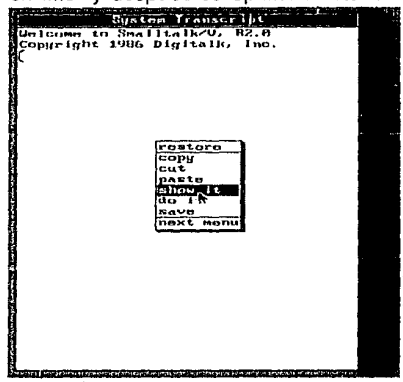


Figura III.5

III.5.4. Manejo de la ventana múltiple del explorador de discos.

Cuando el SMRP conoce el disco de trabajo, aparece la ventana múltiple del explorador de discos. Los paneles que la forman son dos superiores y uno inferior, como se ve en la figura III.8.

El menú de la ventana múltiple (el que aparece en la etiqueta de la parte superior), tiene las dos siguientes alternativas:

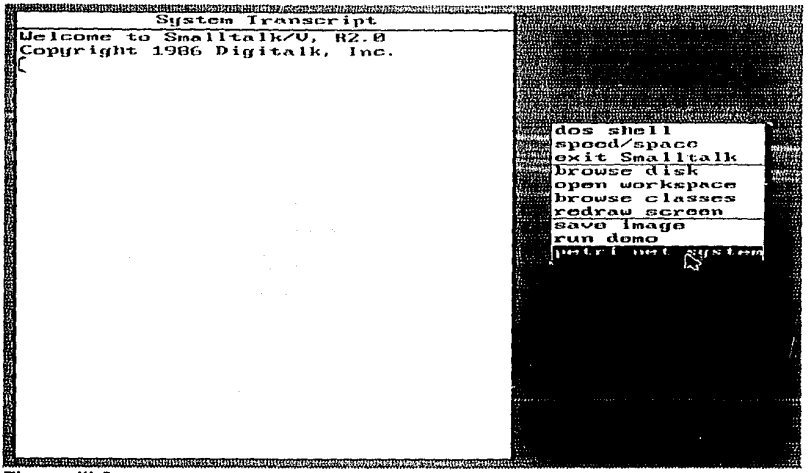


Figura III.6

1) cerrar ventana y salir - Cierra la ventana múltiple y muestra la pantalla de la figura III.9, con la que termina de ejecutarse el SMRP, apareciendo la pantalla de la figura III.2 o la última anterior al ingreso al SMRP.

2) continuar - Deja sin cambios la ventana múltiple, con lo que se puede continuar trabajando.

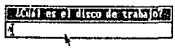
III.5.3.1 Ventana del listado de directorios.

En el panel superior izquierdo de la ventana múltiple se listan los directorios del disco de trabajo. Para seleccionar un directorio debe colocarse el cursor en el nombre del directorio y oprimir el botón izquierdo del mouse (o la tecla + del teclado numérico). Al realizar lo anterior, en el panel superior derecho se listan los archivos del directorio seleccionado y en el panel inferior aparece un texto que muestra el nombre, tamaño, fecha y hora de creación de cada archivo del directorio, como puede observarse en la figura III.10.

El menú de esta ventana tiene las siguientes opciones:

1) ayuda - Muestra la pantalla de ayuda, donde se explica el contenido del menú.

2) borrar - Borra el directorio seleccionado si éste se halla vacío.



BIENVENIDO AL SIRP

- LA ESTRUCTURA DE LA RP DEBE GUARDARSE EN UN ARCHIVO ASCII.

- ESCRIBE LA LETRA QUE DESIGNE LA UNIDAD DE DISCO ($a_1 \ldots_1 f$),

Y DESPUES PRESIONA LA TECLA DE RETURN O ENTER.

- LA TECLA F2 PERMITE IMPRIMIR CUALQUIER VENTANA DEL SISTEMA.

Figura III.7

3) actualizar - Lee nuevamente la estructura del disco.

4) crear - Crea un nuevo directorio con la trayectoria indicada.

5) árbol - Presenta la estructura árborea de los directorios dentro de una ventana especial. Es necesario cerrar esta ventana después de observarla.

III.5.3.2 Ventana del listado de archivos.

Cuando se ha seleccionado un directorio, en el panel superior derecho aparecen los archivos que áquel contiene. Ahora bien, al seleccionar un archivo (de la misma forma que un directorio) su contenido se despliega en el panel inferior, tal como se ve en la figura III.11.

Las siguientes son las alternativas del menú de esta ventana:

1) ayuda - Muestra la pantalla de ayuda, donde se explica el contenido del menú.

2) borrar - Borra el archivo seleccionado.

3) copiar - Copia el archivo seleccionado a otro indicado.

4) imprimir - Imprime el texto del archivo seleccionado.

5) crear - Crea un nuevo archivo con el nombre indicado.

6) protecciones - Muestra y cambia las protecciones del archivo seleccionado.

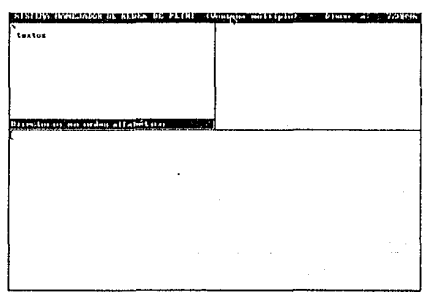


Figura III.8

III.5.3.3 Ventana del editor de texto.

El panel inferior es un editor de texto. El usuario puede utilizarlo para crear o modificar el archivo que contiene la estructura de la RP (ver III.4) y después

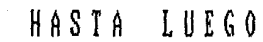


Figura III.9

almacenar el archivo en el disco de trabajo.

Si el archivo ya existe, debe seleccionarse de la lista de archivos. En caso contrario, debe escogerse del menú de la lista de archivos la alternativa "crear", y dar un nombre, con lo que podrá escribirse la estructura de la RP, la cual se detalla en III.3.

Cómo escribir un texto.

Para escribir un texto, debe moverse el cursor al punto donde se desee comenzar a escribir y después oprimir el botón izquierdo del mouse (o la tecla + del teclado numérico). En ese momento aparecerá una marca que indica el punto de inserción, con lo que se pueden teclear los caracteres deseados.

Cómo borrar un texto.

Si se desea borrar texto, debe ubicarse el cursor después del último carácter a borrar. Después se presiona el botón izquierdo del mouse (o la tecla + del teclado numérico) y se oprime la tecla Backspace tantas veces como caracteres se quiere borrar.

Cómo marcar un texto.

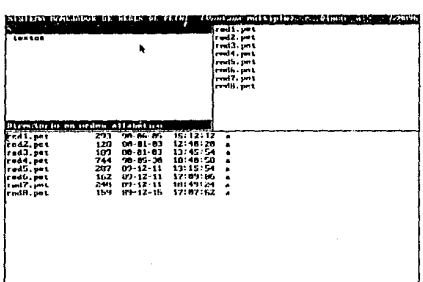


Figura III.10

Para las opciones 3, 4 y 6 siguientes es necesario marcar un bloque de texto. Para ésto debe ubicarse el cursor en un extremo del bloque a marcar; después se presiona el botón izquierdo del mouse y se mantiene así al deslizar el mouse hasta el extremo opuesto del bloque. Al mover el mouse, el texto se nota marcado. Si no se cuenta con mouse, se coloca el cursor en un extremo del bloque y se presiona la tecla + del teclado numérico; después se mueve el cursor al extremo opuesto del bloque y se oprime la tecla - del mismo teclado, con lo cual queda el bloque marcado.

Si el texto ocupa mayor espacio que el del panel, se puede utilizar las

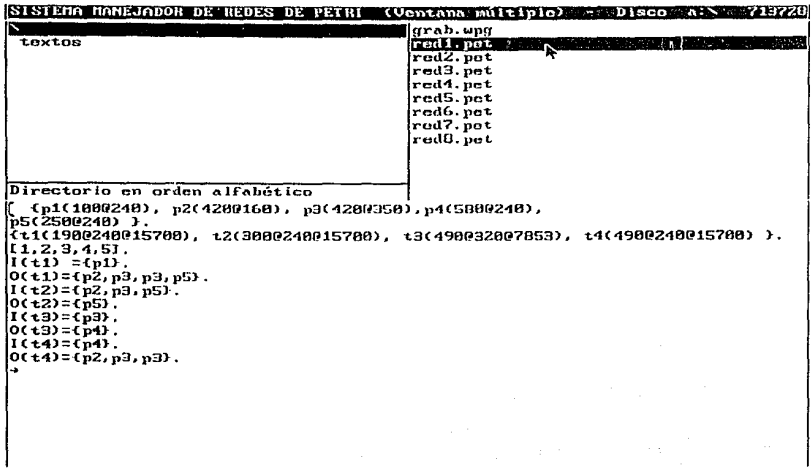


Figura III.11

teclas Home, End, PgUp y PgDn para recorrer el texto y poder verlo completo.

El menú del editor de texto muestra las opciones abajo listadas:

1) ayuda - Muestra la pantalla de ayuda, donde se explica el contenido del menú.

2) restaurar - Presenta la última versión en disco del archivo seleccionado.

3) copiar - Guarda en la memoria el bloque de texto marcado.

4) cortar - Borra el bloque de texto marcado.

5) agregar - Recupera de la memoria el último bloque copiado o cortado.

6) imprimir - Envía a la impresora el bloque de texto marcado.

7) salvar - Salva el archivo en disco, con el texto que contiene la ventana.

8) salvar otro nombre - Salva el texto en un archivo cuyo nombre se indica.

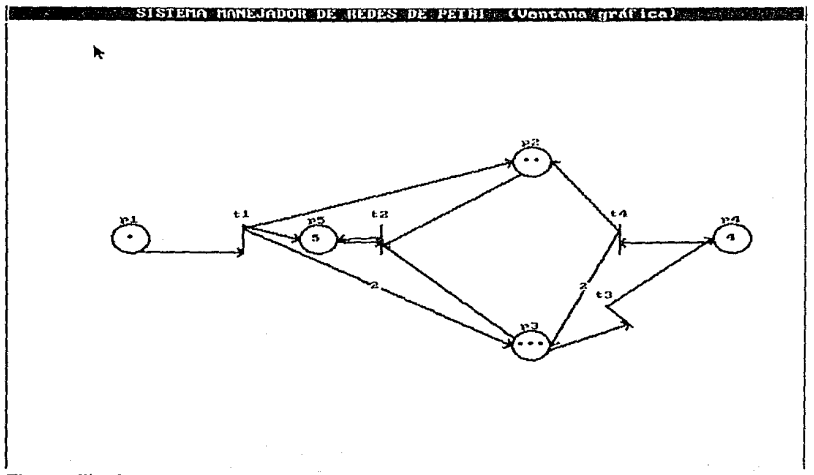


Figura III.12

9) ejecutar RP - Abre la ventana gráfica del sistema y dibuja la RP.

10) menú siguiente - Presenta las cuatro alternativas siguientes.

11) buscar - Busca hacia el final del texto la cadena de caracteres indicada.

12) buscar atrás - Busca hacia el principio del texto la cadena de caracteres indicada.

13) reemplazar - Reemplaza una cadena de caracteres por otra.

14) repetir - Repite la última acción de alguna de las tres anteriores.

Cuando se está seguro de que la estructura de RP está correctamente escrita, se debe seleccionar la opción "ejecutar RP",
con lo que se abre la ventana gráfica y se dibuja la RP.

III.5.4 Manejo de la ventana gráfica.

Al aparecer la ventana gráfica se dibuja la RP. como se muestra en la figura III.12. El menú de la parte superior tiene las opciones sucesivas:

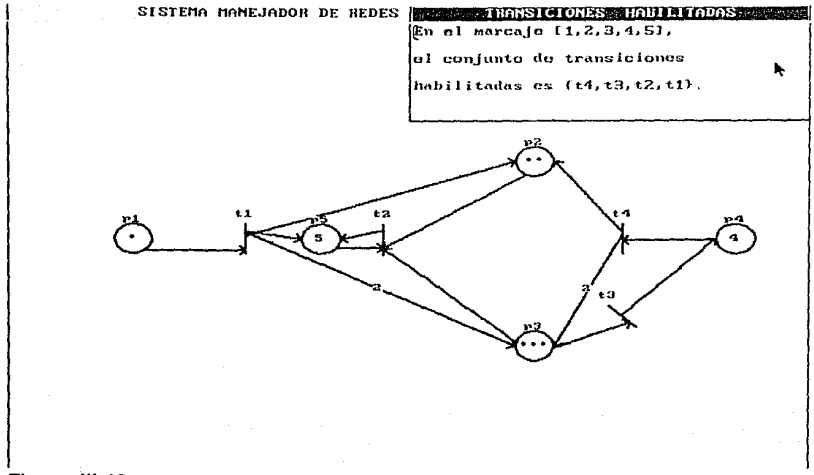


Figura III.13

1) cerrar ventana - Cierra la ventana gráfica y regresa a la ventana múltiple del explorador de discos (figura III.11)

2) continuar - Deja la ventana gráfica sin cambios y permite seguir trabajando en ella.

Las alternativas del menú interior de la ventana gráfica muestran sus resultados en ventanas de texto, las cuales aparecen en la parte superior derecha de la pantalla, como se detalla en la figura III.13. Después de observar los resultados es necesario cerrar dichas ventanas de texto (utilizando el menú propio de la parte superior de la ventana).

A continuación se presentan las opciones del menú interior de la ventana gráfica:

1) ayuda - Muestra la pantalla de ayuda, donde se explica el contenido del menú.

2) sitios - Presenta el conjunto de sitios de la RP.

3) transiciones - Da el conjunto de transiciones de la RP.

4) función entrada - Da la función de entrada de una transición

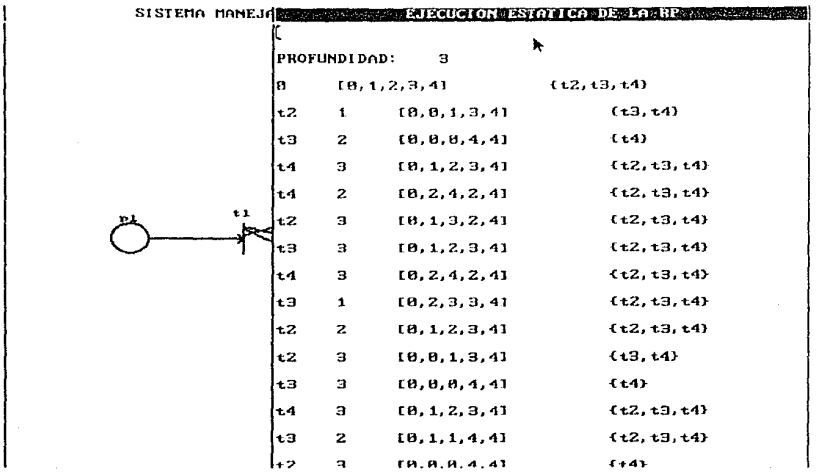


Figura III.14

indicada.

5) función salida - Muestra la función de salida de una transición indicada.

6) marcaje - Presenta el marcaje actual de la RP.

7) transiciones hab. - Muestra el conjunto de transiciones habilitadas en el marcaje actual de la RP.

8) marcaje siguiente - Da el marcaje obtenido al encender una transición, haciendo los cambios pertinentes en la gráfica. Si la transición no está habilitada se advierte del caso.

9) marcaje sucesión - Muestra los marcajes obtenidos al encender una sucesión de transiciones, cambiando también la gráfica. Cuando una de las transiciones no está habilitada se detiene el proceso.

10) ejecución estática - Hemos llamado ejecución estática al cálculo de los marcajes a los que puede llegar la RP y las transiciones habilitadas en tales marcajes. Para ésto se pregunta al usuario por la profundidad del cálculo, i. e., para cada transición habilitada en el marcaje inicial, cuántas veces se calcularán los marcajes alcanzados por la RP, encendiendo las transiciones habilitadas en cada marcaje alcanzado.

La figura III.14 muestra la ventana de la ejecución estática de una RP. El primer renglón indica la profundidad que el usuario indicó. En el segundo aparece un 0 (que indica el comienzo), el marcaje inicial al empezar la ejecución y las transiciones habilitadas en dicho marcaje.

Los siguientes renglones se leen así:

- El primer elemento es la transición encendida.

- Después aparece el nivel en el cálculo (un entero menor o igual que la profundidad indicada).

- El tercer elemento es el marcaje alcanzado al encender la transición que aparece al principio del renglón.

- Por último se muestra el conjunto de transiciones habilitadas en el marcaje que antecede al conjunto.

III.5.4.1 Manejo de las ventanas de resultados.

Las ventanas de resultados aparecen en la parte superior derecha de la pantalla. El menú de ellas permite escoger las alternativas siguientes:

1) ayuda - Muestra la pantalla de ayuda, donde se describe el menú.

2) imprimir - Envía a la impresora el bloque de texto marcado. (En III.5.3.3 se explica como marcar un bloque de texto).

3) ampliar - Agranda la ventana de resultados al tamaño de la pantalla o la regresa a su tamaño original, según sea el caso.

Después de ver los resultados, es necesario cerrar las ventanas que los muestran, pues el sistema se puede saturar. Esto puede realizarse mediante el menú de la parte superior de la pantalla, el cual tiene las siguientes opciones:

1) cerrar ventana - Cierra la ventana de resultados y regresa a la ventana gráfica.

2) continuar - Deja la ventana de resultados sin cambios y permite seguir viéndola.

CONCLUSIONES

I. El modelo de RP se aplica en distintas áreas del conocimiento, como computación, ingeniería de control, economía, química o sociología. Es deseable que cualquier investigador en el área de cómputo conozca, al menos, la teoría básica de las RP.

II. El manejo de las RP puede llegar a ser laborioso. Así, es justificable la realización de un programa de cómputo que maneje el modelo referido. Es por ésto que se creó el SMRP, que está desarrollado para trabajar en microcomputadoras IBM PC o compatibles y puede manejar una estructura básica de RP.

III. El SMRP se creó como una herramienta para la investigación en el modelado de sistemas, pero también puede utilizarse en la enseñanza del modelo de RP.

IV. Utilizamos un ambiente de POO, en este caso el de Smalltalk, por varias razones. La más importante fue la amplia gama de clases de objetos que ofrece Smalltalk, en particular las clases de conjuntos, bolsas y diccionarios (Set, Bag y Dictionary, respectivamente), que nos sirvieron para definir la estructura de una RP sin empezar de cero, y la facilidad de crear nuevas clases. Otra razón poderosa consistió en las ventajas de Smalltalk para crear sistemas interactivos, donde puede manejarse la información en ventanas de distinto tipo (de texto, de gráficas, de listas de elementos). Por último, quisimos sentir en carne propia las ventajas y desventajas de este paradigma de programación que se ha popularizado a gran velocidad.

V. Cuando se ha programado con lenguajes procedurales es difícil adecuarse a la forma de trabajo en la POO, en este caso con Smalltalk. Para conocer el manejo del ambiente de trabajo y las clases de objetos existentes, la inversión de tiempo y esfuerzo de nuestra parte fue considerable. Además, la sintaxis en la POO difiere sustancialmente de los tipos de sintaxis clásicas, lo que hace que haya descontrol al momento de aprender a programar.

VI. Smalltalk consume grandes recursos de cómputo: gran cantidad de memoria -la cual depende del tamaño del ambiente de trabajo-, espacio en disco duro y un adaptador de video aceptable para el manejo de gráficas. Todo esto es justificable, dado el gran poder de abstracción que ofrece el lenguaje, así como la libertad de creación y modificación de la estructura existente que el programador tiene.

VII. Nuestra experiencia al manejar el ambiente de trabajo de Smalltalk fue fascinante pero a la vez llena de complejidades. El respaldo que se tiene para manejar los recursos de la computadora facilita mucho el trabajo del

programador, pero el ahorro de tiempo puede ser nulo al tener que modificar la estructura del archivo "image", que es donde se guarda la estructura del ambiente de programación. Esperamos que las nuevas versiones de Smalltalk resuelvan este problema.

VIII. El SMRP es nuestro primer intento de utilización de la POO. Algo que creemos necesario es la creación de sistemas más complejos con esta metodología, tanto en Smalltalk como en otros LOO, para tener puntos de comparación con otros paradigmas de programación.

IX. En cuanto a las mejoras que se pueden realizar al SMRP está el construir un editor gráfico para dibujar la RP, mediante el uso de íconos que representen los elementos gráficos de una RP, como barras, círculos, puntos y flechas. El sistema también puede adaptarse para utilizar variantes del modelo de RP, como las RP coloreadas. Los cambios y mejoras son fáciles de hacer debido a la modularidad en la programación, mediante el uso de clases de objetos.

APENDICE A

TEORIA DE LAS BOLSAS*

La teoría de las bolsas es una extensión natural de la teoría de los conjuntos. Una bolsa, como un conjunto, es una colección de elementos sobre algún dominio. Sin embargo, a diferencia de los conjuntos, las bolsas permiten ocurrencias múltiples de elementos. En un conjunto un elemento es un miembro de él o no lo es. En una bolsa un elemento puede estar en ella cero, una, dos o cualquier número de veces.

Consideremos las siguientes bolsas sobre el dominio $D_1 = \{a,b,c,d\}$:

$B_1 = \{a,b,c\}$ $B_2 = \{a\}$ $B_3 = \{a,b,c,c\}$ $B_4 = \{a,a,a\}$ $B_5=\{c,c,a,b\}$ $B_6 = \{a,a,a,a,a,b,b,c,d,d,d,d,d,d,d\}$

De los ejemplos anteriores, B_1 y B_2 son conjuntos. Como el orden de los elementos no es importante, B_3 y B_5 son la misma bolsa.

Podemos definir el dominio D como un conjunto de elementos con los cuales las bolsas están construidas. El espacio de bolsas D^n es el conjunto de todas las bolsas cuyos elementos están en D, tales que ningún elemento ocurre más de n veces. Así es que para toda $B \in D^n$:

1) Si $x \in B$ entonces $x \in D$.
2) $\#(x,B) \leq n$ para todo x.

El conjunto D^∞ es el conjunto de todas las bolsas sobre el dominio D; aquí no existe límite en el número de ocurrencias de un elemento en una bolsa.

El concepto básico de la teoría de las bolsas es la función número de ocurrencias #: $DxD^\infty \rightarrow NU\{0\}$, tal que para un elemento x y una bolsa B, denotamos el número de ocurrencias de x en B como #(x,B) (léase "el número de x en B").

La mayoría de los conceptos y notación se siguen de la teoría de los conjuntos de manera obvia. Si restringimos el número de elementos en una bolsa B tal que $0 \leq \#(x,B) \leq 1$, entonces resulta la teoría de los conjuntos.

La función # define el número de ocurrencias de un elemento x en una bolsa B. De lo anterior tenemos que $\#(x,B) \geq 0$ para todo x en B. Un elemento x es un miembro de una bolsa B si $\#(x,B) > 0$. Esto lo denotamos como $x \in B$. Análogamente, si $\#(x,B) = 0$, entonces $x \in B$.

---

* Peterson, James L. Petri Nets and the Modeling of Systems. Ed. Prentice-Hall. Estados Unidos, 1981 pp. 237-240

Definimos la bolsa vacía $\varnothing$ como aquella bolsa que no tiene elementos. Para todo x, $\#(x,\varnothing) = 0$.

La cardinalidad de una bolsa B es el número total de ocurrencias de elementos en la bolsa: $|B| = \Sigma\ \#(x,B)$.

Una bolsa A es una sub-bolsa de una bolsa B (denotado como $A \subset B$) si cada miembro de A es también miembro de B, al menos tantas veces: $A \subset B$ si y sólo si $\#(x,A) \leq \#(x,B)$ para todo x.

Dos bolsas son iguales ($A = B$) si y sólo si $\#(x,A) = \#(x,B)$ para todo x.

De las definiciones anteriores podemos concluir lo siguiente:

i) $A = B$ si y sólo si $A \subset B$ y $B \subset A$.
Si $A = B$ entonces $\#(x,A) = \#(x,B)$ para todo x. Esto implica que $\#(x,A) \leq \#(x,B)$ y $\#(x,B) \leq \#(x,A)$. Por lo tanto tenemos que $A \subset B$ y $B \subset A$.
Si $A \subset B$ y $B \subset A$ entonces $\#(x,A) \leq \#(x,B)$ y $\#(x,B) \leq \#(x,A)$, por lo que $\#(x,A) = \#(x,B)$, y por lo tanto $A = B$.

ii) $\varnothing \subset B$ para todas las bolsas B.
Sea B una bolsa. Sabemos que para todo x miembro de B tenemos que $\#(x,\varnothing) = 0$, por lo que $\#(x,\varnothing) \leq \#(x,B)$. Por lo tanto $\varnothing \subset B$, y esto se cumple para toda bolsa B.

iii) Si $A = B$ entonces $|A| = |B|$.
Si $A = B$ entonces $\#(x,A) = \#(x,B)$ para todo x. Esto implica que $\Sigma\ \#(x,A) = \Sigma\ \#(x,B)$. Por lo tanto $|A| = |B|$.

iv) Si $A \subset B$ entonces $|A| \leq |B|$.
Si $A \subset B$ entonces $\#(x,A) \leq \#(x,B)$ para todo x. Esto implica que $\Sigma\ \#(x,A) \leq \Sigma\ \#(x,B)$. Por lo tanto $|A| \leq |B|$.

Una bolsa A está contenida estrictamente en una bolsa B ($A \subseteq B$) si $A \subset B$ y $A = B$.

Podemos definir cuatro operaciones con las bolsas. Sean A y B bolsas. Tenemos que:

a) Unión ($A \cup B$). $\#(x,A \cup B) = \max(\#(x,A), \#(x,B))$.
b) Intersección ($A \cap B$). $\#(x,A \cap B) = \min(\#(x,A), \#(x,B))$.
c) Suma ($A + B$). $\#(x,A + B) = \#(x,A) + \#(x,B)$.
d) Diferencia ($A - B$). $\#(x,A - B) = \#(x,A) - \#(x,A \cap B)$.

La unión, la intersección y la suma son conmutativas y asociativas.

Además podemos ver que:

$$A \cap B \subseteq A \subseteq A \cup B \quad y$$

$$A - B \subseteq A \subseteq A + B.$$

La distinción entre la unión y la suma está claramente establecida por:

$$| A \cup B | \leq |A| + |B|$$

$$| A + B | = |A| + |B|.$$

APENDICE B

EJEMPLOS DE METODOS DE ALGUNAS CLASES DE OBJETOS

A continuación aparecen algunos ejemplos de métodos de instancia - los que consideramos más representativos- de las clases de objetos que componen al SMRP. Sólo de la clase File se muestra un método de clase.

B.1 Clase PetriNode.

```
Object subclass: #PetriNode
  instanceVariableNames:
    'name position randompos '
  classVariableNames: ''
  poolDictionaries: ''

name: aString position: aPoint
     "Asigna aString a name y aPoint a position."
      name := aString.
      position := aPoint

printOn: aStream
        "Imprime la descripción del receptor"
    aStream
        nextPutAll: name.
```

B.1.1 Clase Place.

```
PetriNode subclass: #Place
  instanceVariableNames:
    'mark '
  classVariableNames: ''
  poolDictionaries: ''

mark: anInteger
     "Asigna anInteger a la variable mark."
   mark := anInteger

draw
        "Dibuja el nodo receptor como un círculo.
         Arriba de éste escribe el nombre del nodo
```

```
        y dentro la marca."
    | font r aspect pen token|
    font := Font eightLine.
    (pen := Pen new)
        defaultNib: 2;
        mask: Form white.
    pen place: position.
    pen solidEllipse: (r := 2 * font width + 15 // 2)
        aspect: 1.
    pen mask: Form black.
    pen ellipse: r aspect: 1.
    pen place: position.
    mark <= 3
      ifTrue:[
              mark = 1
               ifTrue:[pen centerText: ' ' font: font].
              mark = 2
               ifTrue:[pen centerText: '  ' font: font].
              mark = 3
               ifTrue:[pen centerText: '   ' font: font].
            ]
      ifFalse:[ token := mark chanString.
               pen centerText: token font: font].
    pen place: position + (0 @ (r + 4) negated).
    pen centerText: name font: font
```

B.1.2 Clase Transition.

```
PetriNode subclass: #Transition
  instanceVariableNames:
    'ang '
  classVariableNames: ''
  poolDictionaries: ''

drawrot
        "Dibuja una barra, con la inclinación indicada en la,
         variable ang, que representa la transición receptora."
| font r  pen  v1 v2|
font := Font eightLine.
v1 := ang / 10000.
v2 := (3.1415 - v1).
(pen := Pen new)
 defaultNib: 2;
mask: Form black.
r := 2 * font width + 15 // 2.
pen place: position + ( (r*(v1 cos)) rounded @ (r*(v1 sin)) rounded) ;
     goto: position + ( (r*(v2 cos)) rounded @ (r*(v2 sin) negated) rounded).
```

```
(ang = 0) | (ang = 31415)
 ifTrue:[ (ang = 0)
          ifTrue:[ pen place: position
                   + ((r*(v2 cos)) rounded @ (r*(v2 sin) negated) rounded)
                   + (15 negated @ 0).
                   pen centerText: name font: font
                 ]
          ifFalse:[ pen place: position
                    + ((r*(v2 cos)) rounded @ (r*(v2 sin) negated) rounded)
                    + (15  @ 0).
                    pen centerText: name font: font
                  ]
        ]
 ifFalse:[ pen place: position
          + ( (r*(v2 cos)) rounded @ (r*(v2 sin) negated) rounded)
          + (0 @ 10 negated).
          pen centerText: name font: font]
```

B.2 Clase PetriNet.

```
Object subclass: #PetriNet
  instanceVariableNames:
    'input output marking m n places transitions '
  classVariableNames: ''
  poolDictionaries: ''

enabledTransitions
  "Nos da el conjunto de transiciones habilitadas en
   el marcaje actual del receptor."
|conj|
conj := Set new.
(self transitions) do:[ :tran |
                        (self enable: tran name)
                          ifTrue:[ conj add: tran name ].
                      ].
^conj

draw
        "Dibuja la red de Petri receptora."
    |drawedset r r1 font v1 v2 nodetemp a b|
    font := Font eightLine.
    drawedset := Set new.
    r := ( 2 * font width + 15 // 2) // 2.
    nodetemp := nil.
    input keys do: [ :nodeA |
```

```
                    v1 := (nodeA ang) / 10000.
                    nodeA drawrot.
                    drawedset add: nodeA.
                    (input at: nodeA) do:[ :nodeB |
                    nodeB = nodetemp
                    ifFalse:[
                    r1 := ( 2 * font width + 15 // 2) * Random new next.
                    b := nodeB randompos: (nodeB position
                    + ((r1*(v1 cos)) rounded @ (r1*(v1 sin)) rounded)).
                    a := nodeA randompos: (nodeA position
                    + ((r1*(v1 cos)) rounded @ (r1*(v1 sin)) rounded)).
                    b arrow: a.
                    nodeB draw.
                    (self occurr: nodeB bagInp: nodeA) >= 2
                    ifTrue:[ self occurr: nodeB inp: nodeA ].
                            ].
                    nodetemp := nodeB
                                    ].
                    nodetemp := nil
                ].

output keys do: [ :nodeA |
                    v1 := (nodeA ang) / 10000.
                    v2 := (3.1415 - v1).
                    (drawedset includes: nodeA)
                    ifFalse:[ nodeA drawrot].
                    (output at: nodeA) do: [ :nodeB |
                    nodeB = nodetemp
                    ifFalse:[
                    r1 := ( 2 * font width + 15 // 2) * Random new next.
                    nodeA position x < nodeB position x
                    ifTrue:[
                     a := nodeA randompos: (nodeA position
                    + ((r1*(v2 cos)) rounded @ (r1*(v2 sin) negated) rounded)).
                     b := nodeB randompos: (nodeB position
                    +  ((r negated * 2) - 2  @ 0)).
                     a arrow: b.
                     nodeB draw
                           ]
                     ifFalse:[
                     a := nodeA randompos: (nodeA position
                    + ((r1*(v2 cos)) rounded @ (r1*(v2 sin) negated) rounded)).
                     b := nodeB randompos: (nodeB position
                    +  ((r  * 2) + 2 @ 0)).
                     a arrow: b.
                     nodeB draw
                             ].
```

```
                (self occurr: nodeB bagOut: nodeA) >= 2
                ifTrue:[
                    self occurr: nodeB out: nodeA].
                        ].
                nodetemp := nodeB
                                        ].
                nodetemp := nil
            ].
```

B.3 Clase PetriNetDiskBrowser.

```
DiskBrowser subclass: #PetriNetDiskBrowser
  instanceVariableNames:
    'pN '
  classVariableNames: ''
  poolDictionaries: ''

openOn: driveCharacter
        "Abre la ventana múltiple del explorador de
         discos del tamaño de la pantalla, con driveCharacter
         como el disco de trabajo."
    | aTopPane listLineHeight ratio |
    device := driveCharacter.
    self textMenuInit.
    listLineHeight := ListFont height + 4.
    sortedFileList := SortedCollection
        sortBlock: [ :a :b | (a at: 1) < (b at: 1)].
    sortCriteria := #name.
    ratio == nil
        ifTrue: [ratio := 2 / 5].
    wholeFileRequest := false.
    aTopPane := TopPanePn new
        model: self;
        label: '[',volumeLabel, '] ',
            (String with: device), ':';
        minimumSize: 20 * SysFontWidth
            @ (10 * SysFontHeight);
        yourself.
    aTopPane addSubpane:
        (ListPanePn new
            model: self;
            name: #directories;
            change: #directory:;
            menu: #directoryListMenu;
            returnIndex: true;
```

```
        framingBlock: [:box|
            box origin extent:
            box width // 2 @
            ((box height * ratio) truncated
             - listLineHeight)]).
    aTopPane addSubpane:
        (ListPanePn new
            model: self;
            name: #files;
            change: #file:;
            menu: #fileListMenu;
            framingBlock: [:box|
                box origin+(box width//2 @ 0) extent:
                box width-(box width//2) @
                (box height * ratio) truncated]).
    aTopPane addSubpane:
        (sortPane := ListPanePn new
            model: self;
            name: #directorySort;
            change: #sortBy:;
            menu: #sortMenu;
            selection: 1;
            framingBlock: [:box|
                box origin + (0 @
                ((box height * ratio) truncated
                 - listLineHeight))
                extent: box width // 2 @
                    listLineHeight]).
    aTopPane addSubpane:
        (contentsPane := TextPane new
            model: self;
            name: #directory;
            menu: #textMenu;
            framingBlock: [:box|
                box origin+(0 @
                (box height * ratio) truncated)
                        corner: box corner]).
    ( aTopPane dispatcher openIn: (0@0 extent: 640@480)) scheduleWindow
```

B.4 Clase PetriNetShowResults.

```
Object subclass: #PetriNetShowResults
  instanceVariableNames:
    'form graphPane petriNet marking '
  classVariableNames: ''
```

```
  poolDictionaries: ''

openIn: aPetriNet
|topPane|
self petriNet: aPetriNet.
topPane := TopPanePnResults
          new label: 'SISTEMA MANEJADOR DE REDES DE PETRI',
                    '  (Ventana gráfica)'.
topPane addSubpane: ( graphPane := FreeDrawPane new
                    model: self;
                    name: #graph:;
                    menu: #graphmenu;
                    update: #redraw).
(topPane dispatcher openIn: (0@0 extent: 640@480)) scheduleWindow.

nextMarking
     "Nos da el marcaje de la RP, dentro
      de una ventana de texto."
 |logger  extent aString  resp aTran marcajeAnt|
    marcajeAnt := petriNet chanMarkString.
    aString := 'MARCAJE SIGUIENTE'.
    extent := ((Display width // 2 max:
        aString size * LabelFont width + 6) min:
        Display width) @ (Display height // 4).
    aTran := Prompter prompt: 'Transición a disparar'
             default: ''.
    (aTran == nil or: [aTran isEmpty])
    ifFalse:[
      (petriNet enable: aTran)
        ifTrue:[
                petriNet nextMarking: aTran.
                self redraw.
                logger := TextEditorPnResults
                          windowLabeled: aString
                          frame: (Display extent x - extent x @ 0
                                  extent: extent).
                CursorManager execute change.
                logger nextPutAll: 'Al encender la transición ', aTran; cr;cr;
                       nextPutAll: 'en el marcaje ', marcajeAnt, ','; cr;cr;
                       nextPutAll: 'se produce el marcaje ',
                                   petriNet chanMarkString.
              ]
        ifFalse:[
                 logger := TextEditorPnResults
                           windowLabeled: aString
                           frame: (Display extent x - extent x @ 0
                                   extent: extent).
```

```
                CursorManager execute change.
                logger nextPutAll:
                'la transición ', aTran,' no está habilitada'; cr;cr;
                nextPutAll: 'en el marcaje ', petriNet chanMarkString.
               ].
        logger positionAtBeginning.
        CursorManager normal change.
        logger topDispatcher scheduleWindow
            ].
```

B.5 Clase File.

```
creaPetriNet: aString
   "Nos da la PetriNet cuya estructura básica se encuentra
    en el archivo aString."
|inp word p p1 petri platra prv  funcion index1  tmp
 i1 i2 i3 i4 i5 |
inp := File pathName: aString.
petri := PetriNet new initialize.
p := Dictionary new.
platra := Array new: 2.
prv := 1.
index1 := 1.
[(word := inp nextNode) isNil]
 whileFalse:[
  ((word at: 1) = $.)
   ifTrue:[  (prv <= 2)
             ifTrue:[
              platra at: prv put: (Array new: p size).
              1 to: (p size) do:[ :a |
                    tmp := (p at:a) asStream upTo: $(.
                    i1 := (p at:a) indexOfCollection: '('.
                    i2 := (p at:a) indexOfCollection: '@'.
                    i3 := (p at:a) indexOfCollection: ')'.
                    (prv = 2)
                     ifTrue:[ i4 := ((p at:a) copyFrom: (i2 + 1) to: (i3 - 1)).
                              i5 := i4 indexOfCollection: '@'.
                              i4 := (i4 copyFrom: (i5 +1) to: i4 size)
                                    asInteger ].
                    i1 := ((p at:a) copyFrom: (i1 + 1) to: (i2 - 1)) asInteger.
                    i2 := ((p at:a) copyFrom: (i2 + 1) to: (i3 - 1)) asInteger.
                    (prv = 1)
                    ifTrue:[(platra at: prv) at: a
                            put: (Place new name: tmp position: i1@i2)].
                    (prv = 2)
```

```
                    ifTrue:[(platra at: prv) at: a
                            put: (Transition new name: tmp position: i1@i2 ;
                                 ang: i4)].
                    ].
                (prv = 3)
                ifTrue:[ petri markMatrix: p.
                        1 to: p size do:[ :i |
                            ((platra at: 1) at: i) mark: (p at: i)
                                                    asInteger
                                ].
                        ].
                (prv >= 4)
                ifTrue:[ funcion := (p at: 1) at: 1.
                        p1 := Array new: (p size ).
                        1 to: (p size ) do:[ :i |
                          p1 at: i put: ((p at: i) select:
                                        [ :c | c isDigit]) asInteger
                                    ].
                        (funcion = $I)
                         ifTrue:[ 2 to: (p1 size) do:[ :i |
                                 petri connectInp: ((platra at: 2)
                                                    at: (p1 at: 1))
                                                to: ((platra at: 1)
                                                    at: (p1 at: i))]
                                ]
                         ifFalse:[ 2 to: (p1 size) do:[ :i |
                                 petri connectOut: ((platra at: 2)
                                                    at: (p1 at: 1))
                                                to: ((platra at: 1)
                                                    at: (p1 at: i))]
                                ].

                        ].
                p := Dictionary new.
                index1 := 1.
                prv :=  prv + 1]
    ifFalse:[ p at: index1 put: word.
              index1 := index1 + 1]].
petri places: (platra at: 1).
petri transitions: (platra at: 2).
^petri
```

APENDICE C

INSTALACION DE SMALLTALK Y DEL SMRP

C.1 Requerimientos de hardware y software.

El SMRP requiere (como Smalltalk/V R2.0) de un equipo IBM PC, PC XT, PC AT o compatible, o IBM PS o compatible, con las siguientes características:

- 512 Kb de RAM.

- Dos unidades de diskette, o un disco duro con al menos 1.4 Mb de espacio libre y una unidad de diskette.

- Controlador gráfico (MCGA, CGA, EGA, Hercules o AT&T).

- Versión 2.0 o posterior de DOS.

Los siguientes elementos son opcionales:

- Expansión a 640 Kb en RAM.

- Mouse compatible con el de Microsoft.

- Coprocesador matemático (8087 u 80287).

C.2 Instalación de Smalltalk.

Para utilizar el SMRP es necesario contar con el ambiente de trabajo de Smalltalk. Si no sucede así, puede solicitar una copia de Smalltalk conteniendo el SMRP al teléfono 568-6033, ext. 197.

El procedimiento de instalación de Smalltalk es el siguiente:

1) Copiar los archivos de los diskettes al disco duro.

2) Si trabaja con dos unidades de diskette, dar la instrucción

install soft "adaptador de video"

Si tiene disco duro, el comando será:

install hard "adaptador de video"

La alternativa "adaptador de video" es cualquiera de las siguientes listadas a la derecha:

```
ATTmono   - ATT monochrome graphics mode 640 by 400
EGAcolor  - EGA color graphics mode 640 by 350
EGAlowRes - EGA low resolution mode 640 by 200
EGAmono   - EGA monochrome graphics mode 640 by 350
hercules  - Hercules monochrome mode 720 by 348
lowRes    - Color/monochrome mode 640 by 200
toshiba   - Toshiba T3100 monochrome 640 by 400
IBM3270   - IBM3270 mode 720 by 350
IBM640x480  - VGA or MCGA graphics mode 640 by 480
```

C.3 Instalación del SMRP.

El SMRP quedará instalado en Smalltalk si se corre el siguiente código:

```
(File pathName:'petrinet.st') fileIn.
(File pathName:'place.st') fileIn.
(File pathName:'transitn.st') fileIn.
(File pathName:'ptrntdsk.st') fileIn.
(File pathName:'ptrntshw.st') fileIn.
(File pathName:'toppanpn.st') fileIn.
(File pathName:'tpdspptr.st') fileIn.
(File pathName:'listpnpn.st') fileIn.
(File pathName:'tppnpnrs.st') fileIn.
(File pathName:'txtedtrp.st') fileIn.
(File pathName:'txtpnpnr.st') fileIn.
```

BIBLIOGRAFIA

REDES DE PETRI

Baer, J. *Legality and other properties of graph models of computations.* Journal of the ACM. Vol. 17 N° 3, 1970. pp 543-554.

Celko, Joe. *Petri Nets.* Abacus. Vol. 2 No. 1, 1984. pp 40-43.

Landeweber, L. H. *Properties of conflict-free and persistent Petri nets.* Journal of the ACM. Vol. 25 N° 3, 1978. pp 352-364.

Misunas, David. *Petri nets and speed independent design.* Communications of the ACM. Vol 16 #8. Agosto 1973. pp 474-481.

Peterson, James L. *Petri Nets.* Computing Surveys. Vol. 9 N° 3. Septiembre 1977. pp 223-252.

Peterson, James L. *Petri net theory and the modeling of systems.* Ed. Prentice-Hall. Estados Unidos. 1981.

PROGRAMACION ORIENTADA A OBJETOS

Budd, Timothy. *A Little Smalltalk.* Ed. Addison-Wesley. Estados Unidos. 1987.

Cox, Brad J. *Object-Oriented Programming.* An Evolutionary Approach. Ed. Adisson-Wesley. Estados Unidos. 1987.

Danforth, Scott y Tomlinson, Chris. *Type Theories and Object-Oriented Programming.* ACM Computing Surveys. Vol. 20, N° 1. Marzo 1988. pp 29-71.

Ingalls, Daniel. *Design Principles Behind Smalltalk.* BYTE. Agosto 1981. pp 286-298.

Johnson, Ralph E. y Foote, Brian. *Designing Reusable Classes.* Journal of Object-Oriented Programming. Vol.1, N° 2. Junio-Julio 1988. pp 22-35.

Nierstrasz, Oscar. "A Survey of Object Oriented Concepts", en *Object-Oriented Concepts, Databases and Applications.* Editado por Kim, Won y Lochovsky, Frederick H. Ed. Adisson-Wesley. Estados Unidos. 1989.

Nygaard, Kristen. *Basic Concepts in Object Oriented Programming.*

SIGPLAN Notices. Vol. 21, N° 10. Octubre 1986. pp 128-134.

Pascoe, Geoffrey A. *Elements of Object-Oriented Programming.* BYTE. Agosto 1986. pp 139-144.

Robson, David. *Object Oriented Software Systems.* BYTE. Agosto 1981. pp 74-86.

Smalltalk/V. Tutorial and Programming Handbook. Digitalk Inc. Estados Unidos. 1986.

Thomas, Dave. *What's In An Object?* BYTE. Marzo 1989. pp 231-240.

Wegner, Peter. *Classification in Object-Oriented Systems.* SIGPLAN Notices. Vol. 21, N° 10. Octubre 1986. pp 173-182.

Wegner, Peter. *Learning The Language.* BYTE. Marzo 1989. pp 245-253.