XX (101 598 1)

CINVESTAV - IPN

Centro de Investigación y de Estudios Avanzados del IPN
Unidad Guadalajara

Programación automática de procesos distribuidos especificados con redes de Petri

Tesis que presenta:
Grelty del Socorro Canul Novelo

Para obtener el grado de:
Maestro en Ciencias

En la especialidad de:
Ingeniería eléctrica

Guadalajara Jalisco, marzo de 2002

Programación automática de procesos distribuidos especificados con redes de Petri

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:

Grelty del Socorro Canul Novelo

Ingeniero en Sistemas Computacionales
Instituto Tecnológico de Mérida 1993-1998

Becario de CONACYT, expediente no. 142647

Director de Tesis
Dr. Luis Ernesto López Mellado

CINVESTAV del IPN Unidad Guadalajara, marzo de 2002

Agradecimientos

Primeramente a Dios por permitirme concluir ésta maestría satisfactoriamente.

A mis padres y hermanos por el apoyo y confianza dados en todo momento.

A mi asesor de tesis, el Dr. Luis Ernesto López Mellado, por su gran paciencia y por dirigirme en el desarrollo de esta tesis.

Al Dr. Antonio Treviño, por su valiosa ayuda en el desarrollo de la tesis.

Al Dr. Félix Ramos, por su ayuda incondicional y sus consejos a lo largo de toda la maestría.

A mis compañeros de la maestría, de la generación compu-98 por permitirme una estancia agradable en el CINVESTAV y en especial aquellos que alguna o muchas veces me ayudaron, con un especial cariño a Inés .

A Patricia Gutiérrez, antecesora de esta tesis.

A Víctor y Sofía García por auxiliarme en momentos difíciles.

RESUMEN

Esta tesis aborda el tema de generación de software distribuido. Se apoya en la herramienta de modelado Redes de Petri (RP) para la especificación de sistemas.

Una especificación global de un sistema expresada en RP requiere ser repartida en tareas que se ejecutarán en distintos computadores, así también es necesario transformar la especificación original a una especificación distribuida equivalente y correcta, en la que todo tipo de interacción es realizada por paso de mensajes.

Finalmente se obtiene código Java de la especificación distribuida mediante procedimientos implementados en un ambiente visual de especificación y generación de programas.

Índice General

Apéndices

Introducción

Los avances que han ocurrido en el campo de la computación durante las últimas décadas no tienen precedente en ninguna otra industria. De una máquina que costaba 10 millones de dólares y ejecutaba una instrucción por segundo, ahora se tienen máquinas que cuestan 1000 dólares y ejecutan 10 millones de instrucciones por segundo, una ganancia costo/eficiencia de 10^{11}. Otra mejora impresionante fue la invención de redes de computadoras de alta velocidad.

El resultado de estos avances tecnológicos es que ahora no sólo es posible, sino fácil, tener sistemas de cómputo compuestos de una gran cantidad de CPU's conectados mediante una red. Estos sistemas son usualmente llamados sistemas distribuidos, en contraste a los sistemas centralizados (sistemas con un sólo procesador) que consisten de un único CPU, memoria, periféricos y algunos terminales.

Una definición de un sistema distribuido es la siguiente:

Un sistema distribuido es una colección de computadoras independientes que aparecen a los usuarios del sistema como una sola computadora [1]

El hecho de que sea posible construir sistemas distribuidos no necesariamente significa que es una buena idea el hacerlo. Existen ventajas y desventajas de los sistemas distribuidos con respecto a los sistemas centralizados.

La relación costo/eficiencia es mayor, esto es debido a que es más económico adaptar un sistema distribuido (aumentando o eliminando elementos) que cambiar el sistema en su totalidad (en el caso de un sistema centralizado). Otra razón para construir sistemas distribuidos es que algunas aplicaciones son inherentemente distribuidas. Además, un sistema distribuido es más rápido que un sistema centralizado, esto se debe a que cuenta con un mayor número de procesadores. Otra ventaja de un sistema distribuido sobre uno centralizado es la confiabilidad, esto significa que aunque exista una falla parcial el sistema puede seguir funcionando. Finalmente, la extensibilidad de un sistema distribuido concierne al hecho de que nuevos elementos puedan ser acoplados o que elementos existentes puedan ser modificados o eliminados.

Aunque los sistemas distribuidos tienen sus ventajas, también tienen inconvenientes. El punto más débil en un sistema distribuido es el software. Con el actual estado del arte, no se tiene mucha experiencia en diseñar, implementar y usar software distribuido. Un problema más que se puede presentar es debido a la comunicación en la red (los mensajes pueden perderse). Aunque la comunicación es importante, la manera cómo cooperan y se sincronizan

los procesos no deja de ser crucial. En sistemas centralizados, las regiones críticas, exclusión mutua, y otros problemas de sincronización son resueltos mediante el uso de semáforos y monitores entre otros mecanismos. Estos métodos necesitan ser adaptados para ser usados en un sistema distribuido, porque carecen de la existencia de memoria compartida.

Básicamente, los problemas que deben ser resueltos para diseñar un sistema distribuido son los mismos que en un sistema centralizado, sólo que aumentan en dificultad a la hora de la implementación, esto es debido a que no es posible establecer un estado global. Como se mencionó anteriormente, algunos de estos problemas son: la repartición y la distribución de recursos de cálculo y de almacenamiento, problemas de exclusión mutua, interbloqueo, sobreflujo en buffers del sistema, inanición, etc.

Para aminorar la presencia de tales conflictos, el software requiere ser analizado desde los puntos de vista cualitativo (correctud) y cuantitativo (eficiencia). El análisis cualitativo busca propiedades como la ausencia de bloqueos, la ausencia de sobreflujos, o la presencia de situaciones de exclusión mutua en el uso de recursos compartidos. Su meta final es probar la correctud del sistema modelado. El análisis cuantitativo se refiere a la evaluación de la eficiencia del sistema modelado (desempeño, tiempo de ciclo, tiempo promedio de utilización de un recurso, etc.) [10]

El diseño de un sistema puede ser llevado a cabo mediante el uso de modelos. El uso de herramientas formales (las redes de Petri en este caso particular) permiten la construcción de modelos en los cuales se pueden efectuar análisis de eficiencia y de correctud.

Describir (especificar) un sistema es una tarea que consiste en elaborar un modelo del mismo. Según sean los objetivos que se persigan, de un sistema se pueden construir modelos de diferentes tipos. Así, un modelo que enumera las partes del sistema y sus interconexiones se denomina *estructural.* Un modelo que describe cómo opera o funciona un sistema se denomina *funcional [11]*

Las redes de Petri (RP) son un formalismo que posee un soporte matemático simple y claro. Más aún, las RP pueden ser consideradas como una herramienta gráfica adecuada que nos permite modelar (tanto estructural como funcionalmente) y analizar sistemas de eventos discretos (SED) que exhiben evoluciones paralelas. Esto ha conducido a su aplicación en una amplia variedad de campos, como lo son protocolos de comunicación, sistemas de cómputo, sistemas de manufactura flexible, etc.[11]

El trabajo de investigación que a continuación se presenta tiene como principal objetivo la generación automática de código distribuido, apoyándose en la herramienta de modelado RP para la especificación de sistemas. En esta tesis, mediante una herramienta visual se capturará una especificación en RP y se generará de modo automático código distribuido en lenguaje Java correspondiente a dicha especificación.

En el capítulo 1 se introducen formalmente las RP las cuales serán utilizadas para modelar los sistemas concurrentes considerados en esta tesis. En este capítulo se mencionan algunos de los métodos de especificación ascendentes, así también se plantea la manera de obtener esquemas de interconexión concurrentes correctos usando dichos métodos.

El capítulo 2 se centra en el estudio del paralelismo de las RP, es en esta fase donde se presenta un método para particionar la red, de manera que cada subred pueda ser ejecutada

concurrentemente, y así, preservar un alto grado de paralelismo.

En el capítulo 3 se propone un método de transformación de RP para su uso con sistemas distribuidos.

En el capítulo 4 se se presenta un algoritmo para generar código distribuido en lenguaje Java que se comporte de acuerdo a la especificación dada por la RP, además se ilustra las características principales del ambiente visual de especificación desarrollado. Este ambiente permite realizar la especificación de procesos mediante RP y genera automáticamente el código Java correspondiente.

En el apéndice I se tienen los conceptos básicos de concurrencia.

En el apéndice II se tiene la estructura del código Java generado.

Capítulo 1

Modelado de procesos concurrentes con Redes de Petri

Resumen En este capítulo se muestra la importancia del uso de sistemas concurrentes y de las Redes de Petri (RP) como herramienta de especificación y diseño de éstos. Aquí se presentan algunas características y definiciones importantes de las RP, entre las que destacarán las propiedades de vivacidad y acotamiento; se ilustran algunos métodos de construcción de modelos llamados *métodos ascendentes*, los cuales construyen redes incrementalmente, de tal manera que las propiedades de vivacidad y acotamiento sean garantizadas en cada paso sin necesidad de analizar la red final.

1.1 Introducción

La concurrencia es la ejecución simultánea de varias actividades a la vez llamadas procesos o tareas. Los sistemas concurrentes hoy en día han estado ganando terreno en el mundo de la informática debido a sus grandes beneficios. Los beneficios de la ejecución concurrente incluyen el aumento de rendimiento, de la utilización de recursos y de la velocidad de respuesta y mayor apego al mundo real en un sistema informático, sin embargo el manejo de concurrencia hace necesarios añadir mecanismos de sincronización y comunicación para control de las tareas, haciendo el manejo de los sistemas concurrentes más complejo que los sistemas secuenciales.

Los sistemas concurrentes pueden ser pseudoparalelos o paralelos, los primeros también llamados multitarea multiplexan la ejecución de las tareas en un solo procesador, los segundos también llamados multiprocesador manejan varias unidades de procesamiento pudiendo de ese modo ejecutar varias tareas en forma simultánea o paralela, cuando estos sistemas ejecutan sus tareas en unidades independientes e interconectadas entre si son llamados distribuidos.

Para reducir errores y tiempo en el desarrollo de software concurrente es recomendable usar alguna herramienta para su especificación y diseño. Las Redes de Petri (RP) son un un formalismo gráfico bien conocido para el modelado y análisis de sistemas discretos

concurrentes tales como sistemas de cómputo, protocolos de comunicación y sistemas de manufactura.

Una de las características importantes de las RP respecto a su capacidad de análisis es la separación entre la estructura de la red y el marcado. La estructura establece las relaciones entre las entidades que forman el sistema, mientras que el marcado define el estado del sistema. Esta separación entre estructura y marcado permite realizar diferentes clases de estudio sobre el modelo. Por un lado, estudio de propiedades estructurales, que son aquellas que se verifican para cualquier marcado. Por otro lado, estudio de propiedades comportamentales, en las que se verifican propiedades referentes a los marcados.

El uso de RP para representar concurrencia, conflictos y exclusión mutua en un sistema es conveniente, sin embargo, la principal desventaja de las RP radica en la complejidad de la realización del proceso de análisis para garantizar el buen comportamiento del sistema concurrente (en la práctica, algunas veces es muy ineficiente analizar estructuralmente una RP).

Para evitar realizar el proceso de análisis de una RP, el procedimiento de modelado debe ser guiado por metodologías o reglas sistemáticas. Las técnicas más difundidas para la construcción sistemática de modelos usando RP se pueden clasificar en ascendentes y descendentes[12]

En las técnicas descendentes (top-down) se tiene una visión global del sistema desde el inicio de la construcción del modelo. La estrategia consiste en definir inicialmente, un modelo de alto nivel de abstracción sin importar los detalles de bajo nivel. Posteriormente se llevan a cabo, paso a paso, operaciones de refinamiento para dar más detalle al modelo. El refinamiento se efectúa hasta que el nivel de detalle del modelo satisfaga la especificación del sistema. Éstas técnicas imponen condiciones en la expansión de los elementos de la RP, con el fin de que a cada paso se conserven las propiedades de la red original.

En las técnicas de síntesis ascendentes (bottom-up), la estrategia consiste en elaborar modelos de subsistemas correspondientes a las partes resultantes de una descomposición del sistema total, para después unirlos formando un modelo correspondiente al sistema completo. Cada subsistema se modela, ignorando en gran medida, las interacciones con los otros subsistemas; el modelo se construye relativamente rápido y su análisis es más fácil. Los diferentes modelos de subsistemas pueden tener recursos o actividades en común, que constituyen la interacción entre los subsistemas. Una vez definidos los modelos de los subsistemas, se procede a combinar los modelos fusionando las partes en común. Además, en las técnicas propuestas se hace énfasis sobre la labor de análisis del modelo obtenido, éste se reduce considerablemente apoyándose en el resultado del análisis de los modelos de los subsistemas, o bien no existe dicha labor.

La principal ventaja del enfoque ascendente sobre el descendente es la completa libertad que se tiene al especificar el sistema, considerándolo como una composición de subsistemas independientes; cada subsistema es modelado separadamente, facilitando así la labor de análisis.En esta tesis únicamente presentaremos los fundamentos de algunas de las técnicas ascendentes publicadas más difundidas para la construcción sistemática de modelos usando RP, si desea ver técnicas de síntesis descendente refiérase a [12].

1.2 Conceptos básicos y terminología sobre RP

En esta sección se introducen las principales definiciones de elementos relacionados con RP. Para profundizar en aspectos fundamentales, el lector puede consultar [11], [13].

1.2.1 Estructura de una RP

Definición 1.1 *Una RP generalizada (RPG) es un grafo bipartito dirigido representado por una 4-tupla* $N = (P, T, Pre, Post)$, *donde* $P = \{p_1, p_2, ..., p_n\}$ *es un conjunto finito y no vacío de elementos llamados lugares;* $T = \{t_1, t_2, ..., t_m\}$ *es un conjunto finito y no vacío de elementos llamados transiciones;* $P \cap T = \emptyset$; $P \cup T \neq \emptyset$; *Pre- (Post-) es la función de pre (post) incidencia que representa los arcos de entrada (salida) a (de) las transiciones,* $Pre, Post : P \times T \rightarrow \mathbb{N} \cup \{0\}$.

Una RP se representa gráficamente mediante un grafo orientado con dos tipos de nodos: lugares y transiciones. Los lugares se representan mediante círculos y las transiciones mediante barras o rectángulos. Existe un arco orientado de un lugar p a una transición t si y sólo si $Pre(p,t) \neq 0$. Análogamente, existe un arco de una transición t a un lugar p si y sólo si $Post(p,t) \neq 0$. Cada arco orientado se etiqueta por un número natural igual al valor de *Pre* o *Post*, que se denomina *peso del arco*, cuando el peso del arco es 1 puede omitirse en la gráfica.ver figura 1.1

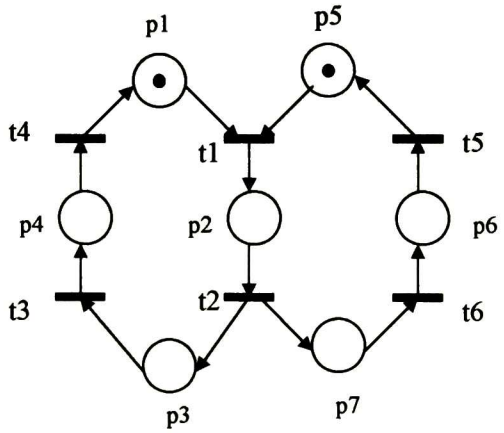


Figura 1.1: Red de Petri

Definición 1.2 *Una RP* $N = (P, T, Pre, Post)$ *se dice ordinaria (RPO) si y sólo si* $\forall p \in P$, $t \in T$, $Pre(p,t) \in \{0,1\}$ *y* $Post(p,t) \in \{0,1\}$.

Definición 1.3 *Un lugar p en una RP es pre-incidente a más de una transición t si* $Pre(p,t) > 0$.

Definición 1.4 *Un lugar p de una RP se dice que es compartido si es pre-incidente a más de una transición.*

Definición 1.5 *Un lugar p es implícito si su marcado se puede evaluar en función del marcado de otros lugares y jamás es el único lugar que impide la sensibilización de sus transiciones de salida ésto nos permite poder suprimir dicho lugar sin afectar el comportamiento del sistema.*

Definición 1.6 *Sea $N = (P, T, Pre, Post)$ una RP, y sean $t \in T$ y $p \in P$. Se definen los siguientes conjuntos: el conjunto $x^\bullet$, $x \in P \cup T$ es el conjunto de todos los sucesores inmediatos de x; el conjunto ${}^\bullet x$, $x \in P \cup T$ es el conjunto de todos los predecesores inmediatos de x.*

Las funciones de pre-, post-incidencia son representadas por las matrices $PRE = [a_{ij}]$ y $POST = [b_{ij}]$ respectivamente, donde $a_{ij} = Pre(p_i, t_j)$ y $b_{ij} = Post(p_i, t_j)$. PRE y $POST$ también son llamadas C^+ y C^- respectivamente.

Definición 1.7 *Sea $N = (P, T, Pre, Post)$ una RP La matriz de incidencia $C = [c_{ij}]$, $i = 1, ..., n$, $j = 1, ..., m$, está definida por $c_{ij} = a_{ij} - b_{ij}$. También se suele denominar matriz de flujo de la red.*

Definición 1.8 *Algunas subclases de RP son:*

1. Un grafo marcado es una RP ordinaria tal que $\forall p \in P$, $|{}^\bullet p| = |p^\bullet| = 1$.
2. Una máquina de estados es una RP ordinaria tal que $\forall t \in T$, $|{}^\bullet t| = |t^\bullet| = 1$.
3. Una red de libre elección es una RP ordinaria tal que $\forall t, t' \in T$, si ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$, entonces ${}^\bullet t = {}^\bullet t'$

1.2.2 Evolución de una RP

Definición 1.9 *Sea $N = (P, T, Pre, Post)$ una RP.*

1. *Un marcado M para N es una función $M : P \to \mathbb{N} \cup \{0\}$ que representa el número de marcas que contienen los lugares. Las marcas se representan gráficamente por puntos en los lugares.*
2. *Un sistema $S = \langle N, M_0 \rangle$ o red marcada es una RP con un marcado, donde N es una RP y M_0 un marcado inicial.*

Definición 1.10 *Sea $S = \langle N, M_0 \rangle$ una red marcada. Se dice que una transición $t_i \in T$ está habilitada para un marcado M si y sólo si $\forall p_j \in P : M(p_j) \geq Pre(p_j, t_i)$.*

Definición 1.11 *Sea $N = (P, T, Pre, Post)$ una RP y sea M un marcado para dicha red. Si una transición $t_i \in T$ está habilitada para un marcado M, puede ser disparada, conduciendo a un nuevo marcado M' definido como sigue: $M'(p) = M(p) - Pre(p, t_i) + Post(p, t_i)$, $\forall p_j \in P$.*

Definición 1.12 *El conjunto de todos los marcados alcanzables desde M_0 en la red N se representa por $\mathfrak{R}(\langle N, M_0 \rangle)$.*

En la figura 1.2 podemos ver ejemplos de evolución de marcados, en cada uno de los ejemplos la transición en cuestión está habilitada por el marcado inicial debido a que los lugares de entrada a la transición cumplen con la definición 1.10 conduciendo al disparo de ésta y produciendo un nuevo marcado.

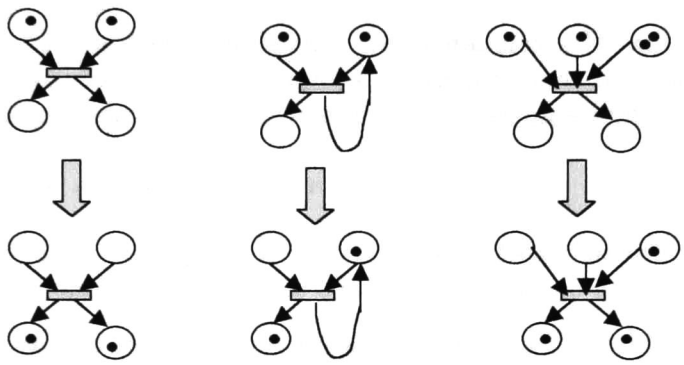


Figura 1.2: Ejemplos de secuencias de Marcados en RP.

1.2.3 Componentes estructurales

Definición 1.13 *Sea $N = (P, T, Pre, Post)$ una RP, y sea C su matriz de incidencia.*

1. *Un p-invariante es un vector $Y : P \to \mathbb{N}$ tal que $Y^T C = \mathbf{0}$ y $Y \geq \mathbf{0}$.*
2. *Un p-invariante se dice canónico si el máximo común divisor de sus componentes es 1.*
3. *Un p-invariante se dice mínimo si es canónico y su soporte (ver definición 1.20) no contiene el soporte de otro p-invariante.*
4. *Un t-invariante es un vector $X : T \to \mathbb{N}$ tal que $CX = \mathbf{0}$ y $X \geq \mathbf{0}$.*
5. *Un t-invariante se dice canónico si el máximo común divisor de sus componentes es 1.*
6. *Un t-invariante se dice mínimo si es canónico y su soporte no contiene el soporte de otro t-invariante.*

Definición 1.14 *Se denomina soporte de una componente Y (X) al conjunto de lugares (transiciones) asociados a los elementos no nulos de Y (X). $||Y||$ $(||X||)$ representa el soporte de la componente Y (X).*

1.2.4 Propiedades de las RP

Un sistema concurrente es, por las interacciones entre los distintos procesos, algo más que la simple adición de un conjunto de procesos secuenciales. En efecto, las relaciones de cooperación y competencia entre los procesos pueden generar "problemas" que no se plantean en la ejecución de cada uno de ellos por separado. Algunos de estos problemas son la no vivacidad o no acotamiento en una RP.

La vivacidad significa funcionamiento total, esto es, toda transición de la RP es potencialmente disparable en todos los marcados alcanzables. Si por lo menos una transición puede ser disparada en cualquier momento, la red se dice libre de bloqueos. A la situación en la cual ninguna transición puede ser disparada se denomina bloqueo. La ausencia de bloqueos suele ser un requisito básico en la especificación de un sistema concurrente.

Otro requisito que debe presentar un sistema es el acotamiento. El acotamiento tiene que ver con la estabilidad del sistema. Por ejemplo, si la RP modelada no es acotada, entonces esto podría indicar la ocurrencia de un desbordamiento en algunos buffers del sistema.

Definición 1.15 *Sea $\langle N, M_0\rangle$ una RP marcada*

1. *$\langle N, M_0\rangle$ se dice libre de bloqueos si y sólo si para todo marcado alcanzable $M \in \mathfrak{R}(\langle N, M_0\rangle)$ el conjunto de transiciones habilitadas por M no es vacío.*
2. *Una transición es viva en $\langle N, M_0\rangle$ si y sólo si desde todo marcado alcanzable, existe una secuencia de disparos que la habilita.*
3. *La red marcada $\langle N, M_0\rangle$ es viva si y sólo si todas sus transiciones son vivas.[18]*

Definición 1.16 *Una RP es estructuralmente viva si y solo si es viva para cualquier marcado inicial M_0 finito.[18]*

Definición 1.17 *Un lugar $p \in P$ es k-acotado para M_0 si y sólo si existe un número entero k tal que $M(p) \leq k < \infty$ para cualquier marcado $M \in \mathfrak{R}(\langle N, M_0\rangle)$. Se denomina cota del lugar p al menor entero k que verifica la desigualdad anterior.[16]*

Definición 1.18 *Una RP marcada N es k-acotada para M_0 si y sólo si todos sus lugares son k-acotados para M_0 : $\forall p \in P$, $\forall M \in \mathfrak{R}(\langle N, M_0\rangle)$, $M(p) \leq k$.[16]*

Definición 1.19 *Una RP es estructuralmente acotada si es acotada para cualquier marcado inicial y finito.[18]*

Definición 1.20 *Sea $N = (P, T, Pre, Post)$ una RP, y sea C su matriz de incidencia.*

1. *N se dice conservativa si y sólo si existe un p-invariante cuyo soporte sea todo P*
2. *N se dice repetitiva si y sólo si existe un t-invariante cuyo soporte sea todo T.[16]*

Proposición 1.1 *Sea N una RP, C su matriz de incidencia y $||Y|| = P$. Si $Y^T C = \mathbf{0}$ (RP conservativa), N es estructuralmente limitada (el marcado se conserva acotado).[16]*

1.3 Métodos de síntesis ascendentes

En las técnicas de síntesis ascendentes (bottom-up), la estrategia consiste en elaborar modelos de subsistemas correspondientes a las partes resultantes de una descomposición del sistema total, para después unirlos (siguiendo algunas reglas de unión) formando un modelo correspondiente al sistema completo.

Los diferentes modelos de subsistemas pueden tener lugares o transiciones representando recursos o actividades en común, que constituyen la interacción entre los subsistemas. Una vez definidos los modelos de los subsistemas, se procede a combinar los modelos fusionando las partes en común: un lugar, una transición, o un camino (secuencia de lugares y transiciones).

Además, en las técnicas propuestas se hace énfasis sobre la labor de análisis del modelo obtenido: ésta se reduce considerablemente apoyándose en el resultado del análisis de los modelos de los subsistemas, o bien no existe dicha labor.

1.3.1 Fusión de lugares

A cada paso, dos subredes se unen de tal manera que dos lugares, uno de cada red, se convierten en un solo lugar. El procedimiento es el siguiente:

Sean dos redes $N_1 = (P_1, T_1, Pre_1, Post_1)$ y $N_2 = (P_2, T_2, Pre_2, Post_2)$ a combinarse por la fusión de $p_i \in P_1$ y $p_j \in P_2$ en un lugar $p_k \notin P_1 \cup P_2$.

La red resultante $N = (P, T, Pre, Post)$ se obtiene como sigue:

$P = P_1 \cup P_2 - \{p_i, p_j\} \cup p_k; \qquad T = T_1 \cup T_2$

Pre y $Post$ se obtienen sobre T y se obtienen reemplazando p_i y p_j por p_k en Pre_1, $Post_1$, Pre_2, $Post_2$. Si las funciones de incidencia están expresadas en forma matricial, éstas se obtienen de las matrices

$$Pre_{12}=\begin{bmatrix} Pre_1 & 0 \\ 0 & Pre_2 \end{bmatrix}, \qquad Post_{12}=\begin{bmatrix} Post_1 & 0 \\ 0 & Post_2 \end{bmatrix},$$

a las que se adiciona la fila correspondiene a p_k, la cual se obtiene sumando los vectores fila correspondientes a p_i y p_j, y estas filas se eliminan. Para más detalle ver [18]

Ejemplo 1.1 *La figura 1.3 muestra los modelos de dos procesos que realizan una tarea simple; la operación de cada proceso necesita de la disponibilidad de un recurso que puede ser una impresora o algún dispositivo periférico; esto está representado para cada modelo por los lugares p_3 y p_6. Estos lugares son candidatos a fusionarse para obtener un modelo del sistema global.*

1.3.2 Fusión de transiciones

El procedimiento para combinar modelos fusionando transiciones es similar al descrito para fusionar lugares. El modelo resultante $N = (P, T, Pre, Post)$ se obtiene como sigue:

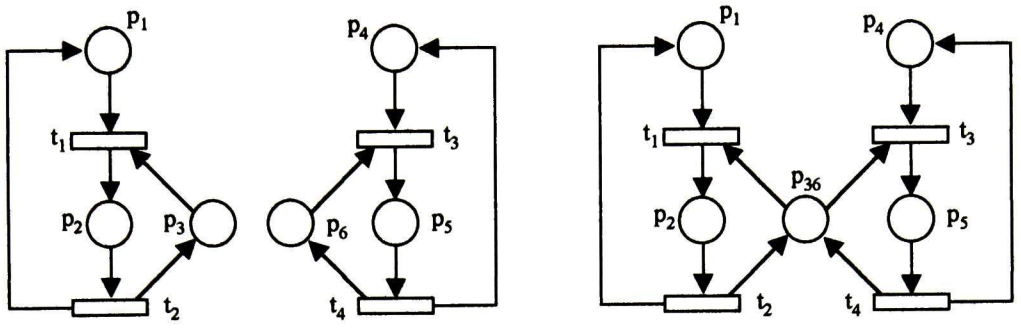


Figura 1.3: Fusión de dos procesos a través de un lugar

$P = P_1 \cup P_2$; $T = T_1 \cup T_2 - \{t_i, t_j\} \cup t_k$; *Pre* y *Post* se obtienen de Pre_{12} y $Post_{12}$ operando sobre las columnas correspondientes a t_i, t_j y t_k como se describió anteriormente. Para más detalle ver [18]

Ejemplo 1.2 *La figura 1.4 muestra los modelos de una máquina y un almacén que pueden ser combinados mediante la fusión de t_2 y t_3, para obtener un modelo más complejo.*

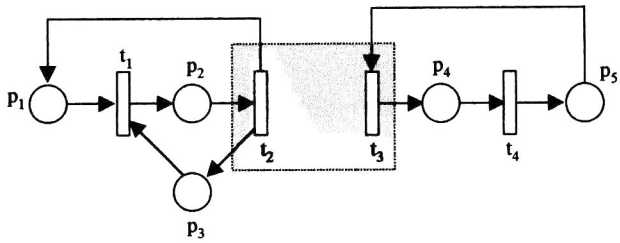


Figura 1.4: Fusión de dos procesos a través de una transición

1.3.3 Fusión de caminos

Los métodos basados en esta idea, consideran que los modelos de los subsistemas son circuitos elementales y las uniones se efectúan a través de una parte común llamada camino elemental; éste se define como una secuencia de lugares y transiciones que empieza y termina por lugares: camino lugar-lugar (ver figura 1.5) o que empieza y termina por transiciones: camino transición-transición (ver figura 1.6). Para mayor detalle ver [18] o [14].

1.4 Síntesis modular: fusión de caminos

En esta sección se incluirán los principales resultados del método base adoptado para la construcción de modelos (fusión de modelos mediante caminos elementales MT, [14], método

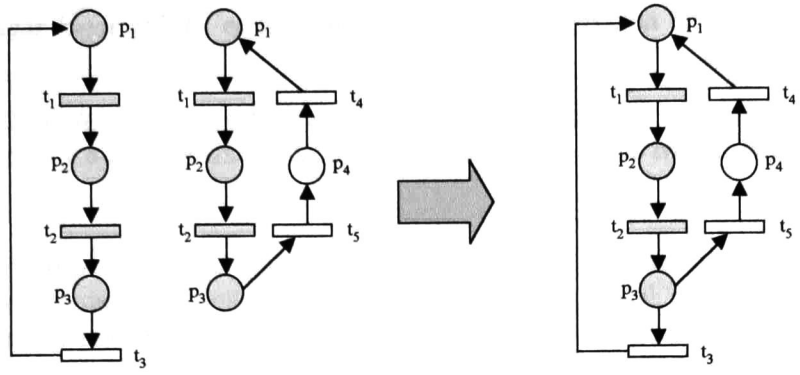


Figura 1.5: Fusión de procesos a través de un camino lugar-lugar

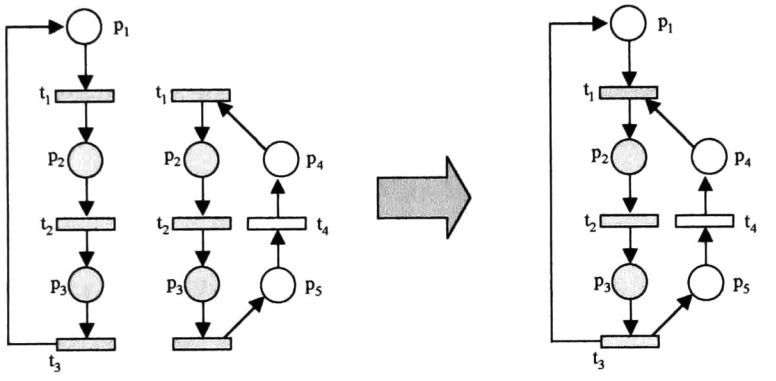


Figura 1.6: Fusión de procesos a través de un camino transición-transición

de transformación aumentado MTA [16]).

1.4.1 Circuitos vivos y acotados: definiciones básicas

Los métodos basados en este principio, consideran que los modelos de los subsistemas son circuitos elementales y las uniones se efectúan a través de una parte común llamada camino elemental (fusión de caminos elementales).

Definición 1.21 *Caminos elementales [14]*

1. *Un camino dirigido es una secuencia alternada de transiciones y lugares,* $t_1 p_1 t_2 ... p_{n-1} t_n$, *en el cual ningún lugar o transición ocurre más de una vez, donde* $PRE(p_{i-1}, t_i) \neq 0$ *y* $POST(p_j, t_j) \neq 0$, $\forall \leq i \leq n$, $1 \leq j \leq n-1$.

2. *Camino transición-transición (CTT) es un camino dirigido que termina en ambos extremos por una transición.*

3. *Camino lugar-lugar (CLL) es un camino dirigido que termina en ambos extremos por un lugar.*

4. *Camino elemental simple (CES) un CTT (CLL) en el cual cada lugar (transición) tiene solamente una transición (un lugar) de entrada y una transición (un lugar) de salida.*

El procedimiento para realizar síntesis de módulos incluye restricciones para asegurar la vivacidad y el acotamiento de los circuitos: un circuito elemental marcado, al ejecutarse no debe acumular ni perder marcas durante el disparo de sus transiciones; estos circuitos llamados *circuitos vivos y acotados* (CVA), se caracterizan en función de los parámetros *razón de arcos* (arc ratio) y *residuo* (remainder).

La razón de arcos indica una relación producción/consumo, expresada en enteros, en un lugar p_i, de un circuito. El residuo indica el número de marcas que quedan en un lugar p_i sin habilitar la transición de salida.

En un camino $t_1p_1t_2p_2t_3...p_nt_{n-1}$, la razón de arcos (r) y el residuo (s) entre t_1 y t_2 se calculan como sigue:

$r_{12} = INT(POST(p_1,t_1)/PRE(p_1,t_2))$, si $PRE(p_1,t_2) \neq 0$,

$r_{12} = 0$, si $PRE(p_1,t_2) = 0$.

$s_{12} = MOD(POST(p_1,t_1)/PRE(p_1,t_2))$, si $POST(p_1,t_1)/PRE(p_1,t_2) \geq 1$.

$s_{12} = 0$, si $POST(p_1,t_1)/PRE(p_1,t_2) < 1$.

Ejemplo 1.3 *Razón de arcos y residuo entre dos transiciones t_i y t_j.*

Considérese la RP con dos transiciones t_1, t_2 y un lugar p_1 como la de la figura 1.7. La transición t_1 produce cinco marcas y la transición t_2 consume dos marcas cuando estas dos transiciones son disparadas una vez. si la razón de arcos es mayor que dos, entonces t_2 puede ser disparada más de dos veces. Las marcas que permanecen en p_1 (el residuo de la razón de arcos) no pueden contribuir al disparo de la transición t_2 cuando la transición t_1 ha sido disparada sólo una vez. Sin embargo, estas marcas tienen que ver con el acotamiento de la red si ésta es un circuito.

Un caso similar ocurre con la RP de la figura 1.8

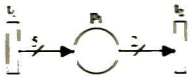


Figura 1.7: Razón de arcos $r_{12} = 2$ y residuo $s_{12} = 1$

Definición 1.22 *La razón de arcos r_{1n} y el residuo s_{1n} de una secuencia alternada de transiciones y lugares, $t_1p_1t_2p_2t_3...p_{n-1}t_n$ (el primer y el último nodo deben ser transiciones) se definen como sigue:*

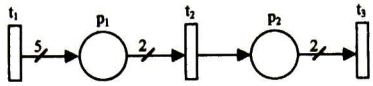


Figura 1.8: Razón de arcos $r_{13} = 1$ y residuo $s_{13} = 1$

$r_{1n} = INT(r_{1n-1} \times POST(p_{n-1}, t_{n-1})/PRE(p_{n-1}, t_n))$, *si* $PRE(p_{n-1}, t_n) \neq 0$

$r_{1n} = 0$, *si* $PRE(p_{n-1}, t_n) = 0$

$s_{1n} = \sum_{i=1}^{n-1} s_{1i} + MOD(r_{1n-1} \times POST(p_{n-1}, t_{n-1})/PRE(p_{n-1}, t_n))$,
si $r_{1n-1} \times POST(p_{n-1}, t_{n-1})/PRE(p_{n-1}, t_n) \geq 1$

$s_{1n} = \sum_{i=1}^{n-1} s_{1i} + r_{1n-1} \times POST(p_{n-1}, t_{n-1})$,
si $r_{1n-1} \times POST(p_{n-1}, t_{n-1})/PRE(p_{n-1}, t_n) < 1$

[14]

Un CVA es un caso especial de circuitos en RP generalizadas y se define de la siguiente manera:

Definición 1.23 *Un CVA es un camino dirigido* $t_1p_1t_2p_2t_3...p_nt_{n+1}$, *tal que*

1. $t_1 = t_{n+1}$,
2. $r_{1n+1} = 1$, $s_{1n+1} = 0$, *y*
3. *el circuito es vivo.*

Un *CVA estructural* es definido de la misma manera que un CVA excepto por la condición de marcado.

Los CVA pueden ser combinados fusionando caminos comunes (CLL o CTT) para construir modelos más complejos resultando redes vivas y acotadas.

El análisis de RP utilizando la técnica de CVA se basa en los siguientes teoremas y lemas (la demostración de éstos se encuentra en [14]):

Lema 1.1 *Un autolazo* $N\ (t_1, p_1, t_1)$ *es vivo y acotado si y sólo si es un CVA.*

Lema 1.2 *Un CVA* $t_bp_bt_2p_2...t_ip_i...p_nt_b$ *es vivo y acotado.*

Lema 1.3 *Una RP es viva y acotada si la RP es la unión de dos CVA* (N_1, N_2) *fusionados a lo largo de un CTT.*

Corolario 1.1 *Una RP es viva y acotada si la RP es la unión de un CVA N_1 y de un CVA estructural N_2 fusionados a lo largo de un CTT, y uno de los lugares del CTT en N_1 tiene suficientes marcas para hacer N_2 un CVA.*

Lema 1.4 *Sea p_{last} el último lugar de un CLL. La RP es viva y acotada si las siguientes condiciones son satisfechas:*

1. *La RP es la unión de dos CVA N_1, N_2 fusionados a lo largo de un CLL.*
2. *$r_n^1 = 1$ ó $r_m^2 = 1$, donde r_n^1 es la razón de arcos de $N_1(t_b^1 \in p_{last}^\bullet$ y $t_b^1 \in T_1)$, y r_m^2 es la razón de arcos de N_2 $(t_b^2 \in p_{last}^\bullet$ y $t_b^2 \in T_2)$.*

Corolario 1.2 *RP es viva y acotada si las siguientes condiciones son satisfechas:*

1. *RP es la unión de un CVA N_1 y un CVA estructural N_2 fusionados a lo largo de un CLL, y uno de los lugares del CLL en N_1 puede tener suficientes marcas para hacer N_2 un CVA.*
2. *$r_n^1 = 1$ ó $r_m^2 = 1$, donde r_n^1 es la razón de arcos de $N_1(t_b^1 \in p_{last}^\bullet$ y $t_b^1 \in T_1)$, y r_m^2 es la razón de arcos de N_2 $(t_b^2 \in p_{last}^\bullet$ y $t_b^2 \in T_2)$.*

Sea N la unión de tres CVA N_1, N_2, y N_3, donde N_2 está fusionado a N_1a lo largo de un CTT y N_3 está fusionado a N_1 a lo largo de un CLL. El acotamiento y vivacidad de la N depende de la relación de traslape entre CTT y CLL.

Sean B_1 y B_2 dos conjuntos de nodos de dos diferentes caminos dirigidos, donde $b_{first,i}$ es el primer nodo y $b_{last,i}$ es el último nodo del camino dirigido cuyo conjunto es B_i.

Definición 1.24 *Se dice que dos caminos dirigidos diferentes están parcialmente traslapados, si y sólo si:*

1. *$b_{first,1} \notin B_2$ y $b_{last,1} \in B_2$ o*
2. *$b_{first,1} \in B_2$ y $b_{last,1} \notin B_2$.*

Ejemplo 1.4 *Traslape entre CVA*

Tres CVA están fusionados a lo largo de un CLL y de un CTT en el ejemplo de la figura 1.9. En (a), el camino 1 (p_1) y el camino 2 ($t_3p_3t_4$) no se traslapan, donde el camino 1 es un CLL entre el CVA N_1 ($t_1p_1t_2p_2t_3p_3t_4p_4t_1$) y N_2 ($t_5p_5t_6p_1t_5$), y el camino 2 es un CTT entre N_1 y N_3 ($t_3p_3t_4p_7t_7p_6t_3$). La red es viva y acotada. Sin embargo un CTT y un CLL están parcialmente traslapados en (b) y (c). La red de (b) no es viva. La red de (c) es viva, pero no acotada.

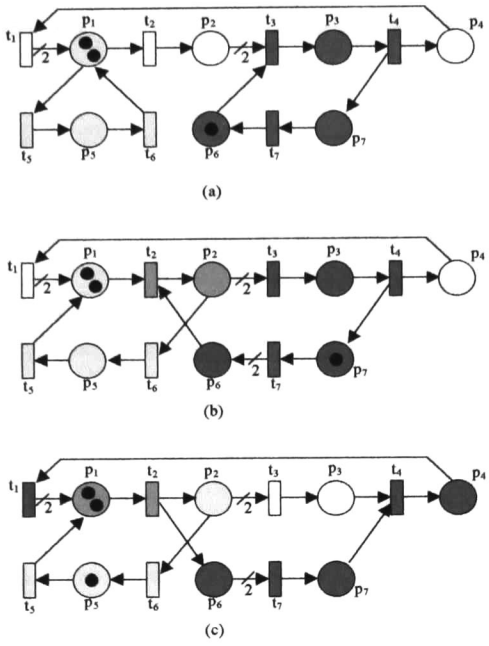


Figura 1.9: Ejemplos de CVA con traslapes parciales. (a) No hay traslape. (b) Traslape parcial, red no viva (c) Traslape parcial, red no acotada

Teorema 1 *Sea N la RP obtenida de fusionar un circuito vivo y acotado N_2 en una RP N_1 a lo largo de un CTT (CLL), tal que ningún camino en N_1 es traslapado en CTT (CLL). En el caso de un CLL, se requiere además la siguiente condición:*

$r_n = 1$, *donde* $t_b \in p^{\bullet}_{last}$ *y* $t_b \in T_2$ (p_{last} *es el último lugar en CLL), ó,*

$r^i_m = 1, \forall 1 \leq i \leq K$, *donde* r^i_m *es la razón de arcos de un circuito vivo y acotado incluyendo CLL en* N_1 *y* K *es su valor máximo* ($t^i_b \in p^{\bullet}_{last}$ *y* $t^i_b \in T_1$).

El siguiente resultado puede ser obtenido: N es viva y acotada si y sólo si N_1 es viva y acotada.

Teorema 2 *Sea N la RP obtenida de fusionar dos redes N_1 y N_2 a lo largo de un CTT, tal que CTT es un CES. Entonces N es viva y acotada si N_1 y N_2 son vivas y acotadas.*

Teorema 3 *Sea N_1 la RP obtenida de remover un circuito vivo y acotado N_2 de la RP N, en la cual N_2 está unido a N a lo largo de un CTT (CLL) de tal manera que ningún camino en N es traslapado en este CTT (CLL). En el caso de un CLL se requiere una condición más:*

$r_n = 1$, *donde* $t_b \in p^{\bullet}_{last}$ *y* $t_b \in T_2$ (p_{last} *es el último lugar en CLL), ó,*

$r^i_m = 1, \forall 1 \leq i \leq K$, *donde* r^i_m *es la razón de arcos de un circuito vivo y acotado incluyendo CLL en* N_1 *y* K *es su valor máximo* ($t^i_b \in p^{\bullet}_{last}$ *y* $t^i_b \in T_1$).

El siguiente resultado puede ser obtenido: N es viva y acotada si y sólo si N_1 es viva y acotada.

1.4.2 Método de reducción

El enfoque básico del método de reducción propuesto por Koh- Dicesare consiste en remover los CVA que están fusionados a la red a lo largo de un CTT o de un CLL. Como se ve en el teorema 3, el remover estos circuitos no cambia la vivacidad y acotamiento de la red si las condiciones descritas son satisfechas. Ver figura 1.10

Los circuitos elementales son simplificados en circuitos elementales conservando su razón de arcos y residuo

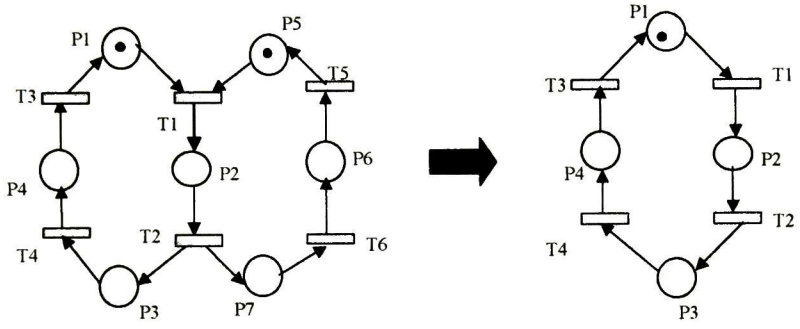


Figura 1.10: Reducción por eliminación de circuitos

1.4.3 Método de síntesis

Método de Transformación (MT)

El método de síntesis desarrollado por Koh & DiCesare es un enfoque modular ascendente. Si los modelos de los subsistemas son circuitos elementales simples conteniendo una sola marca, se trata de RP vivas y a salvo. La composición de subredes por fusión de caminos elementales, se efectúa paso a paso; inicialmente se tienen dos circuitos, al unirse, el modelo resultante puede unirse a un nuevo circuito y así sucesivamente. La única restricción es la de evitar traslapes al fusionar caminos elementales.

Respecto al marcado de la RP, para conservar su carácter monomarcado, se deben tener en cuenta las siguientes consideraciones: si la fusión de dos circuitos se efectúa a través de un CTT, cada circuito debe estar marcado o bien la parte común debe contener una marca; si la fusión se efectúa a través de un CLL, sólo uno de los circuitos, incluyendo la parte común, se marcará. Con estas reglas de construcción los modelos obtenidos son vivos y a salvo.

Para RPG se sigue un procedimiento similar. Un CVA simple es el punto de partida. Un CVA puede ser fácilmente expandido agregando lugares y transiciones manteniendo r_n

y s_n, en 1 y 0 respectivamente. Entonces sintetizamos la red fusionando los dos CVA a través de un CTT o de un CLL de acuerdo a la relación entre los dos circuitos. Un CTT puede ser interpretado como una relación AND entre dos redes, y un CLL como una relación OR. Si sintetizamos la red recursivamente usando el teorema 1, entonces la vivacidad y el acotamiento de la red son garantizados.

Método de Transformación Aumentado (MTA)

El método de transformación aumentado MTA propuesto por P. Gutiérrez [16] que a continuación se presenta es una extensión del MT de Koh & Dicesare [14] visto en la subsección anterior para poder representar recursos compartidos y el paso de mensajes

- Recursos Compartidos

Cuando sintetizamos un sistema con múltiples recursos compartidos, las relaciones entre los subsistemas tal vez no pueden ser descritas por un CTT o un CLL como en el MT. En este caso, podemos verificar la vivacidad de la red resultante mediante MTA para recursos compartidos el cual presentamos seguidamente:

Consideremos la siguiente red, en la cual se comparte más de un recurso entre k procesos.

Sea $N = (P, T, Pre, Post)$ una RPO, donde

$P = P_1 \cup P_2 \cup ... \cup P_k$; $P_i \neq P_j$ para $i \neq j$,

$T = T_1 \cup T_2 \cup ... \cup T_k$; $T_i \neq T_j$ para $i \neq j$,

$Pre = Pre_1 \cup Pre_2 \cup ... \cup Pre_k$,

$Post = Post_1 \cup Post_2 \cup ... \cup Post_k$,

y $\Pi_i = (P_i, T_i, Pre_i, Post_i)$ es una RPO construida con el MT propuesto por Koh & DiCesare.

En otras palabras, $N = \cup_{i=1}^{k} \Pi_i$.

Considérese la adición de un lugar l a N como sigue:

1. $\forall\ \Pi_i$, $\exists\ t_a^i, t_b^i \in T_i$ tal que existe un arco de l a t_a^i y un arco de t_b^i a l (TTP $\pi_i = t_a^i ... t_b^i$).
2. El circuito l, π_i, l no traslapa ningún circuito de Π_i.

Sea $N_l = N \cup l$, donde $N_l = (P_l, T_l, Pre_l, Post_l)$ y C_l es su matriz de incidencia. Nótese que N_l puede ser obtenida mediante una fusión apropiada de CVA.

Ahora, consideremos la adición de un lugar l_p a N de la siguiente manera: l_p está unido a algunas Π_i mediante un TTP contenido en π_i, digamos σ_i ($\sigma_i = t_c^i ... t_d^i$), tal que σ_i no traslapa ningún circuito de Π_i (C_i es la matriz de incidencia de Π_i).

Sea $N_{lp} = N \cup l_p$, donde $N_{lp} = (P_{lp}, T_{lp}, Pre_{lp}, Post_{lp})$ y C_{lp} es su matriz de incidencia.

Definamos una nueva RP $N' = N_l \cup N_{lp}$ (nótese que en N' las secciones críticas de l_p están contenidas dentro de las secciones críticas de l, esto es, $\sigma_i \subseteq \pi_i$).

La red N' es viva y acotada si N lo es y existe un marcado que la hace viva.

Para esta nueva red, N', ya no es posible aplicar el MT para asegurar la vivacidad y acotamiento, ya que se debe fusionar un circuito a través de dos caminos elementales, lo cual no es permitido por el MT (en la figura 1.11 se muestra una posible configuración de N').

De lo anterior podemos decir dado que un recurso compartido uno a dos redes por medio de dos caminos TTP's uno de cada red, esto es posible de construir por medio del MT por unión de circuitos, pero cuando se requiere agregar más de un lugar compartido entre dos redes, esto es posible si los caminos TTP's de unión de ambas redes se encuentran contenidos en caminos TTP's de recursos compartidos agregados anteriormente y los caminos no se traslapan entre sí. Así mismo los recursos compartidos deben pertenecer a algún circuito que los haga vivos.

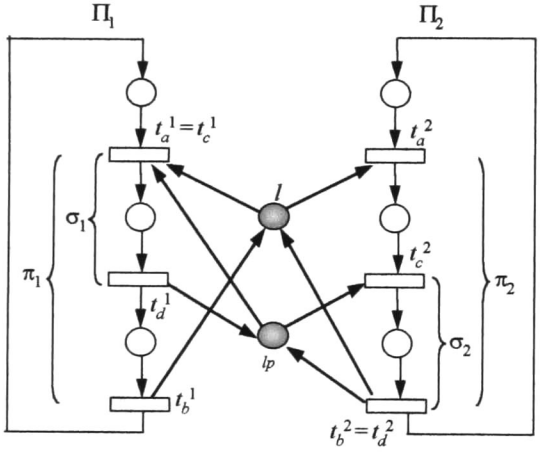


Figura 1.11: Configuración de recursos compartidos

- Método de Transformación de paso de mensajes

Una de las formas más eficientes de sincronizar tareas es mediante el paso de mensajes [17]. Sin embargo, al igual que con los recursos compartidos, esta relación no siempre puede ser expresada mediante el MT. Para sintetizar procesos que se sincronicen mediante paso de mensajes hay que hacer uso de otras técnicas de análisis.

Para mejorar en cierta medida el MT para que admita paso de mensajes, P. Gutiérrez [16] propone la siguiente estructura de RP.

Sea $M = (P, T, Pre, Post)$ una RPO construida mediante el MTA.

La adición de módulos que representen paso de mensajes m (ver figura 1.12) de tal manera que conecten dos circuitos (Π_i, Π_j) de M, esto es, conectan un circuito Π_i con un circuito

Π_j, $i \neq j$ obteniéndose así una red M' Sea N_m una red simplificada, que sólo contendrá a Π_i, Π_j y a m.

La red de petri M'es viva y acotada si N_m lo es.

Podemos ver que la red N_m es un grafo marcado, por lo que para determinar si es viva y acotada bastará con verificar que todos sus circuitos estén marcados.

De lo anterior podemos concluir que la unión de circuitos vivos y acotados por medio de mensajes resulta en redes vivas y acotadas si los mensajes se encuentran en circuitos marcados.

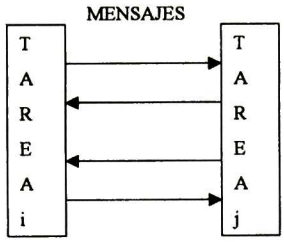


Figura 1.12: Paso de mensajes entre dos tareas

1.5 Esquemas de interconexión de procesos concurrentes

Una característica importante de las RP es su capacidad gráfica para la representación de modelos (a menos que el modelo sea excesivamente grande) de manera que, los principales elementos del sistema se ponen de relieve en su representación.

En Agha *et. al.* [17] se señala que el paso de mensajes es la forma más eficiente de comunicación en un sistema distribuido, sin embargo, argumenta que los lenguajes concurrentes deben proveer cierta cantidad de abstracciones de comunicación para simplificar la tarea de programación.

Es importante tener presente que los mecanismos fundamentales de sincronización para procesos distribuidos, tales como rendezvous, estructuras fork-joint, semáforos, recursos compartidos, paso de mensajes, toma de decisiones, son fácilmente modelables y representables mediante RP En la figura 1.13 se presentan gráficamente algunos esquemas de red modelando dichos elementos de sincronización.

1.5.1 Obtención de Esquemas de Interconexión Concurrentes

Una manera eficiente de realizar el proceso de especificación de un sistema es mediante el uso de un método ascendente (como los métodos descritos en la sección anterior). Así, cada una de las subtareas puede ser descrita progresiva e independientemente, y al término de

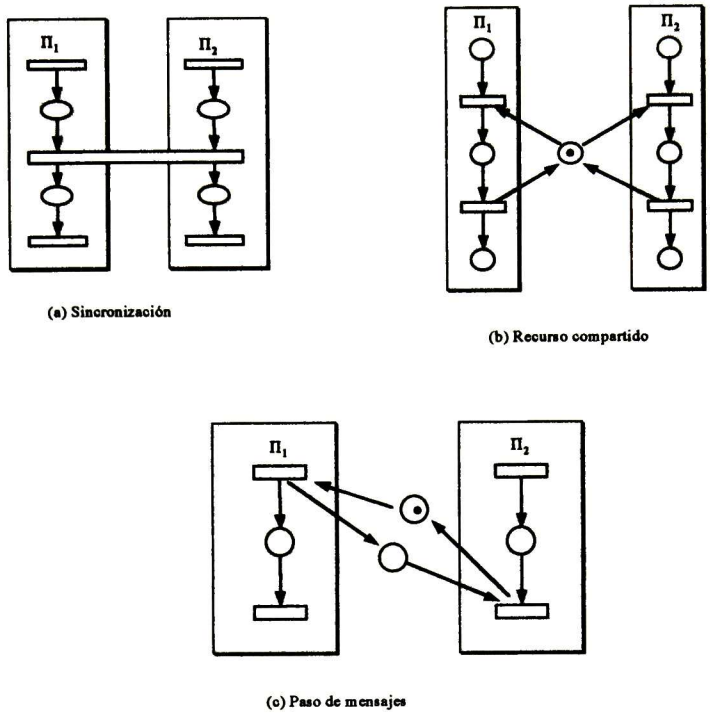


Figura 1.13: Mecanismos de interconexión de procesos concurrentes

la especificación se procede a unir los subsistemas correspondientes a la descomposición total. A su vez, durante el proceso de especificación es deseable obtener modelos correctos evitando así la labor de análisis de propiedades (vivacidad y acotamiento) o bien realizarla a cada paso. Además de garantizar la correctud de los modelos obtenidos es deseable que la metodología de modelado propuesta sea lo menos restrictiva posible, esto es, que permita modelar una amplia variedad de procesos; en este caso en particular, se desea que se permita el modelado de los esquemas presentados en la figura 1.13.

Sincronización de tareas

En el enfoque básico presentado por Koh & DiCesare para modelar sistemas (Método de Transformación, *MT*) [14], los lugares, transiciones y marcas se interpretan de la siguiente manera. Los lugares representan la disponibilidad de un recurso (por ejemplo, máquinas, robots, impresoras, etc.). Las transiciones representan el inicio o final de alguna actividad. Las marcas representan la cantidad de recursos disponibles o condiciones de verdad o falsedad.

Mediante este método podemos sintetizar una red fusionando dos CVA a través de un CTT o de un CLL de acuerdo a la relación entre los dos circuitos. Un CTT puede ser interpretado como una relación AND entre dos redes, y un CLL como una relación OR. De esta manera, podemos representar sincronización y alternativas entre procesos (ver figura 1.13.a-b).

Paso de mensajes

El paso de mensajes es la forma de comunicación de los sistemas distribuidos, la construcción de estos esquemas es válido por el método de transformación aumentado para paso de mensajes presentado por P. Gutiérrez [16] (ver figura 1.13.c).

Recursos Compartidos

El uso de recursos compartidos es una forma forma de optimización de recursos en los sistemas concurrentes.

La construcción de estos esquemas cuando solo se maneja un recurso compartido es válido por el por el método de Koh & DiCesare (MT) Mediante este método podemos fusionar el lugar compartido a dos CVA a través de dos TTP's, uno a la vez (ver figura 1.13.d).

Cuando se requiere unir dos CVA por más de un recurso compartidos es necesario seguir el método de P. Gutiérrez (MTA para recursos compartidos) presentado en la sección anterior [16].

1.6 Conclusiones

Los sistemas concurrentes aumentan el rendimiento de un sistema; pero también la complejidad de su control. Los sistemas concurrentes pueden ser pseudoparalelos o paralelos, los primeros, también llamados multitarea, multiplexan la ejecución de las tareas en un solo procesador; los segundos, también llamados multiprocesador, manejan varias unidades de procesamiento pudiendo de ese modo ejecutar varias tareas en forma simultánea o paralela, cuando estos sistemas ejecutan sus tareas en unidades independientes e interconectadas entre si son llamados distribuidos.

Para reducir errores y tiempo en el desarrollo de software concurrente es recomendable usar alguna herramienta eficaz para su especificación y diseño como son las RP.

Aunque las RP son un formalismo matemático y gráfico, los métodos de análisis para éstas tienen algunos inconvenientes. Es bien sabido que una limitación es la dificultad de analizar redes de gran tamaño. En otras palabras, el conjunto de todos los marcados alcanzables en RP grandes no pueden ser analizados en la práctica por tal motivo, la modularización de RP es deseable en el modelado de sistemas complejos.

En este capítulo de presentaron algunos métodos que permiten construir sistemas que preservan las propiedades de la red, esto se debe a que se establecen restricciones a la hora de realizar pasos de síntesis de tal manera que interacciones concurrentes entre las subredes sean apropiadamente controladas evitando de este modo el proceso de análisis de la RP.

Se presentó en detalle uno de estos métodos sistemáticos de construcción de redes. Este método es el de *fusión de circuitos*,así también se mostró una extensión al método anterior el cual consiste en fusión de modelos a través de varios esquemas (recursos compartidos, paso de mensajes, etc.) respetando ciertas restricciones para preservar la vivacidad y el acotamiento

de la RP. Estos métodos parecen ser promisorios ya que no es son muy restrictivos a la hora de construir modelos.

Por último se presentó una técnica de obtener esquemas de interconexión concurrente usando los métodos de construcción de redes vistos en este capítulo.

Capítulo 2

Especificación de Sistemas Distribuidos

Resumen En este capítulo se presenta algunos conceptos de paralelismo, seguidamente se propone una técnica de particionar una descripción global de un sistema expresada en Redes de Petri para programación de sistemas distribuidos.

2.1 Introducción

Cuando un sistema se compone de varios subsistemas que evolucionan simultáneamente, se dice que es un sistema con evoluciones paralelas; si este se ejecuta en distintas localidades, se dice que es distribuido.

La concurrencia se clasifica como concurrencia aparente o concurrencia real. En un ambiente con un único procesador, la concurrencia aparente es el resultado de realizar un intercambio entre las actividades concurrentes. En un ambiente multiprocesador no distribuido, la concurrencia real es el resultado de ejecutar actividades con el mismo reloj en una localidad centralizada. En un ambiente distribuido la concurrencia es el hecho de ejecutar actividades independientes quizás en paralelo en localidades independientes interconectadas entre si, a lo que podemos llamar un ambiente multiprocesador distribuido.

Para capturar los requerimientos de sistemas concurentes se requiere de una técnica de especificación capaz de modelar paralelismo, además, se debe permitir que los requerimientos puedan ser analizados para verificar la correctud del modelo. En las RP nos permite hacerlo de manera fácil y entendible. La técnica que se usará en este capítulo para capturar especificaciones correctas será el método de composición (MTA) presentado en el capítulo anterior.

Así también para manejar un ambiente distribuido complejo se requiere de una técnica que nos permita dividir la especificación del sistema en sus componentes distribuibles la cual expondremos en este capítulo. En este capítulo expondremos nuestra propuesta para realizar este trabajo.

2.2 Principios básicos

En esta sección se presentan algunas características interesantes para el modelado de sistemas con eventos concurrentes que nos permitirán abordar el problema de asignación de procesadores, un aspecto crucial en el caso de sistemas distribuidos.

Se planteará el problema de particionar una RP de manera óptima, de tal manera que se pueda asignar cada subred a un procesador y se ejecuten en paralelo.

Para esto se necesitará el concepto de invariantes positivos. Los invariantes positivos son de gran interés para analizar propiedades estructurales de una RP debido a que nos permiten saber si la especificación no se bloquea y si se repite (RP vivas y repetitivas). Se pueden encontrar diferentes algoritmos para obtener estos invariantes, tal como los que se presentan en [11] y en [20].

A continuación se introducen los conceptos de paralelismo en datos y estructural así como el de invariantes. Los algoritmos que se presentarán posteriormente se centran en el paralelismo estructural (partición de redes debido a su estructura).

2.2.1 Paralelismo en datos

El paralelismo en datos es un concepto que una RP marcada representa de una manera natural usando el marcado de la RP. Informalmente, existe paralelismo en datos cuando se presenta alguno de los siguientes casos:

1. Si el problema puede ser modelado de tal manera que la red resultante consiste de varias componentes conexas idénticas.
2. Si existe un p-invariante(componente conservativa) que contenga más de una marca.

Puesto que en general una RP puede contener varias marcas, es posible modelar la reentrancia, poniendo más de una marca en algún lugar; este comportamiento se presenta al tratar con el paralelismo en datos.

Cabe destacar que el paralelismo en datos depende del marcado mas no de la estructura, es decir, cada marca representa un dato a ser procesado. Tal como se muestra en la figura 2.1, la existencia de múltiples marcas en una p-componente, permite que haya varias transiciones habilitadas y por lo tanto, que puedan ser disparadas simultáneamente.

2.2.2 Paralelismo estructural

Las propiedades estructurales son independientes del marcado. La distribución/sincronización es un tipo de paralelismo estructural muy importante para el modelado de sistemas distribuidos concurrentes. Una distribución en una RP se modela cuando una transición tiene más de un lugar de salida, una sincronización se modela cuando una transición tiene más de un lugar de entrada (figura 2.2).

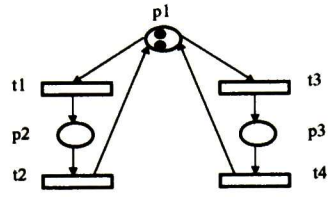


Figura 2.1: Paralelismo en datos

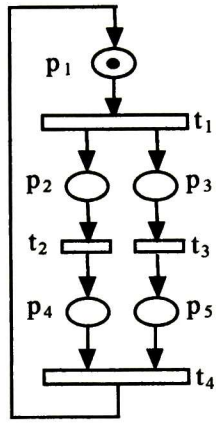


Figura 2.2: Paralelismo estructural

2.2.3 Componentes de una RP

Los invariantes de una RP (apartado 1.2.3) particionan el grafo en subgrafos llamados componentes. Son útiles en análisis matricial de RP para determinar su propiedad de conservación y repetición.

Los soportes de P-invariantes de una RP la descomponen en componentes conservativas llamadas en ocasiones P-componentes. La suma de los marcados de los lugares de un soporte, ponderada en x, permanece constante.

Si cada lugar de una RP pertenece a algún soporte(es decir pertenece a un P-invariante total) la red es estructuralmente *acotada*. La afirmación en sentido contrario no es válida.

La red es conservativa respecto a un vector de ponderación igual a un invariante.

Los soportes de T-invariantes mínimos de una RP la descomponen en *componentes repetitivas*(llamadas también T-componentes); el disparo de las transiciones de un soporte (no está definida la secuencia), a partir del marcado inicial conduce a la red al mismo marcado. Si cada transición de una RP pertenece a algún soporte (existe un T-invariante total) la red es repetitiva. Desafortunadamente con la determinación de los invariantes no es posible concluir la vivacidad de una RP, si bien podemos afirmar que una red viva tiene un soporte total de T-invariantes(repetitiva) el inverso no es válido.

Ejemplo 2.1 *Sea la red de la fig. 2.3 con marcado inicial* $Mo = [11100]^T$ *cuya matriz de incidencia es la siguiente:*

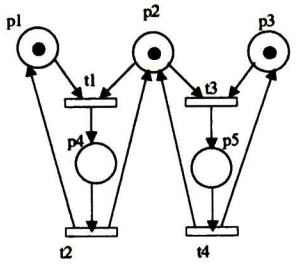


Figura 2.3: RP con dos P-invariantes

$$A = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \end{array} \begin{array}{c} \begin{array}{cccc} t1 & t2 & t3 & t4 \end{array} \\ \begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 1 & -1 & 1 \\ 0 & 0 & -1 & 1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \end{array}$$

La expresión matricial $X^t A = \begin{bmatrix} x1 & x2 & x3 & x4 & x5 \end{bmatrix} A = 0$ *y conduce al siguiente conjunto de ecuaciones:*

$-x1 - x2 + x4 = 0,\ \ x1 + x2 - x4 = 0,\ \ -x2 - x3 + x5 = 0,\ \ x2 + x3 - x5 = 0$

de las cuales obtenemos los P- invariantes mínimos y sus correspondientes soportes:

$X_1 = [10010]^T \qquad < X_1 >= \{p1, p4\}$
$X_2 = [01011]^T \qquad < X_2 >= \{p2, p4, p5\}$
$X_3 = [00101]^T \qquad < X_3 >= \{p3, p5\}$

visto en forma matricial el resultado queda de la siguiente manera:

$$\begin{array}{c} \\ P1 \\ P2 \\ P3 \\ P4 \\ P5 \end{array} \begin{array}{c} \begin{array}{ccc} X_1 & X_2 & X_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \end{array}$$

El conjunto de P-invariante obtenido particiona la RP en las componentes conservativas ilustradas en la figura 2.4
De similar manera la expresión matricial $AY = A\begin{bmatrix} y1 & y2 & y3 & y4 \end{bmatrix} = 0$ *conduce al siguiente conjunto de ecuaciones:-y1+y2=0, -y1+y2-y3+y4=0, y1-y2=0, y3-y4=0 de las cuales obtenemos los T- invariantes mínimos y sus correspondientes soportes:*

$Y_1 = [1100]^T \qquad < Y_1 >= \{t1, t2\}$
$Y_2 = [0011]^T \qquad < Y_2 >= \{t3, t4\}$

expresándolo de forma matricial tenemos:

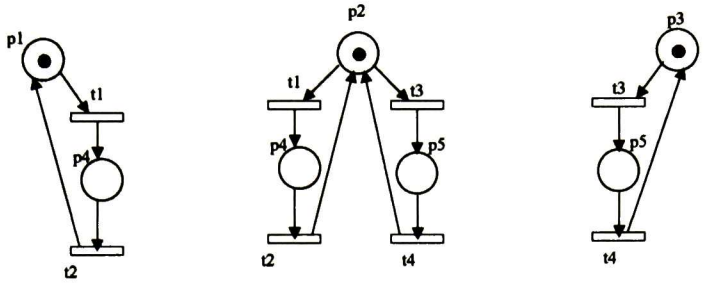


Figura 2.4: Invariantes de lugares (P-invariantes)

$$\begin{array}{c} \\ t1 \\ t2 \\ t3 \\ t4 \end{array} \begin{array}{c} \begin{array}{cc} Y_1 & Y_2 \end{array} \\ \left[\begin{array}{cc} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{array}\right] \end{array}$$

El conjunto de T-invariantes obtenido particiona la RP en las componentes repetitivas ilustradas en la figura 2.5

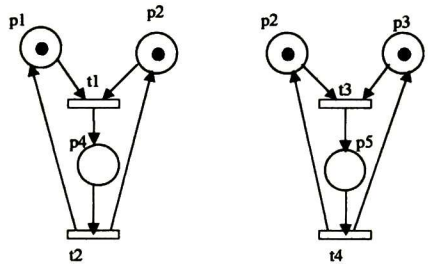


Figura 2.5: Invariantes de transiciones (T-invariantes)

2.3 Partición de especificación global

Esta sección introduce un enfoque estructural del problema de scheduling, es decir, se trata el problema de particionar una RP en subredes que se ejecutarán concurrentemente. Los algoritmos que se presentan utilizan la noción de P-invariantes.

Nos basaremos en el algoritmo de P. Gutiérrez [16], para dividir una RP en componentes conservativas en el cual dado los P-invariantes elementales se reparten las transiciones de cada uno de estos en distintas tareas. El algoritmo de obtención de P-invariantes elementales no será mostrado. Sin embargo es importante subrayar que este algoritmo es óptimo para sistemas concurrentes no distribuidos pero no así para los distribuidos, puesto que este divide solo las actividades de RP (transiciones) en componentes conservativas, pero en

un SD donde donde a diferencia de un sistema no distribuido no se comparte memoria es imposible mantener los estados de las tareas (lugares de RP) en forma común haciéndose también necesario dividirlos junto con las actividades. Es decir se requiere dividir las RP en componentes estructurales completamente independientes, pero a la vez relacionadas funcionalmente. En este capítulo se propone una extensión al método de P. Gutiérrez para la especificación adecuada de sistemas distribuidos.

2.3.1 Obtención de componentes conservativas

Los tipos de redes permitidas para generar programas concurrentes serán aquellos construidos mediante el MTA. Como se vio en el capítulo anterior, las redes construidas con el MTA son conservativas, esto es, están cubiertas por p-invariantes. Además, cada vez que se fusionaba un nuevo proceso a la red, se agregaba paralelismo si esta fusión era realizada a través de un CTT, lo cual implicaba que un nuevo p-invariante era generado. Así, puede pensarse que la partición de la red en p-invariantes elementales y su distribución en procesadores induce un alto grado de paralelismo.

Si desea ver un algoritmo de obtención de P-componentes referirse a [16]

2.3.2 Repartición de Transiciones de una RP.

El algoritmo que a continuación se presenta tiene como objetivo dividir las transiciones de las RP en componentes conservativas previamente obtenidas por medio de un algoritmo que calcule P-invariantes mínimos.

Algoritmo 1

Pre^T es la transpuesta de la matriz de Pre-incidencia de la RP.

Y es matriz de P-invariantes elementales de la RP.

ΛT es una matriz de dimensión $[m, n]$ donde m=número de transiciones y n= número de procesadores. Un elemento $\Lambda T[i][j]$ es igual a 1 si y sólo si una transición t_i es asignada al procesador j, es igual a 0 de otra manera.

Entradas: Pre, Y^T

Salidas: ΛT
j:=1
mientras exista un renglón igual a **0** en ΛT **hacer**
  si renglón $[Y^T \times Pre]_{:j} <> 0$
    concatenar una columna $[Y^T \times Pre]_{:j}$ a ΛT
    para i:=1 .. número de transiciones **hacer**
      si $\Lambda T[i][j] = 1$ **entonces** para l:=1..número de lugares $Pre[l][i] = 0$
  fin si
    j:=j+1
fin mientras

Nota 2.1 *Note que* $[Y^T \times Pre]$ *es un conjunto de invariantes mínimos expresados en términos de transiciones (cada columna está compuesta de las transiciones presentes en un invariante dado).*

El algoritmo 1 particiona las transiciones una red, la cual está cubierta por p-invariantes. La función principal de este algoritmo es ir tomando p-invariantes uno a uno, e ir asignando sus transiciones a procesadores (se toma como principio que las actividades de la red son efectuadas por las transiciones). Este paso se realiza mientras haya transiciones a asignar. Al término del algoritmo, se tendrán a lo más y procesos concurrentes (donde y es la cantidad de p-invariantes mínimos).

Ejemplo 2.2 *Considere la RP modelada en la figura 2.6 Después de ejecutar un algoritmo*

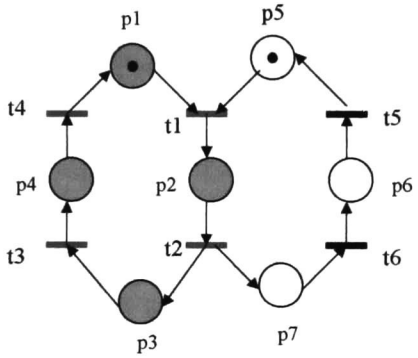


Figura 2.6: RP con dos P-invariantes

de obtención de p-invariantes se obtiene el siguiente resultado:

$$Y = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cc} Y_1 & Y_2 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \end{array}$$

De la red podemos obtener

$$Pre = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cccccc} t1 & t2 & t3 & t4 & t5 & t6 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Ahora apliquemos el algoritmo 1.

Paso 1. Tomar el invariante $[Y^T \times Pre]_{:1}$ *y asignarlo al procesador*

$$Pre = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cccccc} t1 & t2 & t3 & t4 & t5 & t6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}, \Lambda_T = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \Lambda_1 \\ \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \end{array}$$

Paso 2. Tomar el invariante $[Y^T \times Pre]_{:2}$ *y asignarlo al procesador 2.*

$$Pre = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cccccc} t1 & t2 & t3 & t4 & t5 & t6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}, \Lambda_T = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \Lambda_1 \quad \Lambda_2 \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \end{array}$$

Así, para la RP de la figura 2.6, se requiere particionar la red en dos tareas, teniendo una de ellas a las transiciones t1, t2,t3,t4 y la otra a t5 y t6.

Ejemplo 2.3 *Considere la RP modelada en la figura 2.7*

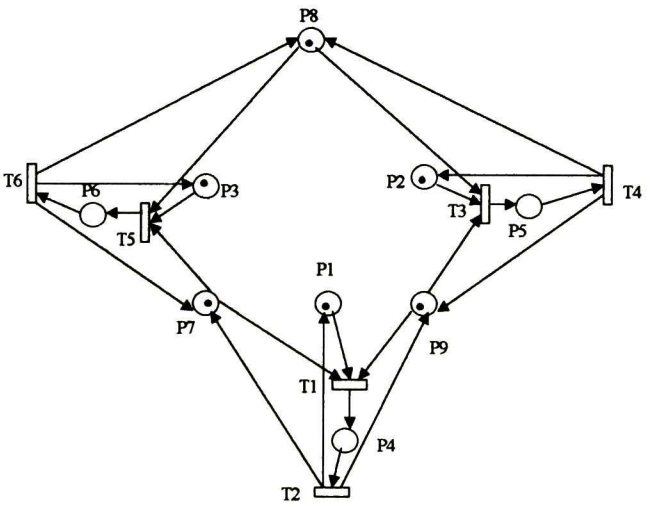


Figura 2.7: RP que modela el problema de tres filósofos comensales

De la red podemos obtener la matriz Pre y después de ejecutar un algoritmo de obtención de

p-invariantes se obtiene la matriz Y de invariantes como sigue:

$$\mathrm{Pre} = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad Y = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Ahora apliquemos el algoritmo 1 a la red de la figura 2.7.

Paso 1. Tomar el invariante $[Y^T \times\ Pre]_{:1}$ el cual es distinto de 0 , asignarlo al procesador 1. y eliminar las columnas de transiciones asignadas en Λ_T en Pre

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}, \Lambda_T = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{c} \Lambda_1 \end{array} \\ \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{array}$$

Paso 2. Tomar el invariante $[Y^T x\ Pre]_{:2}$ el cual es distinto de 0 , asignarlo al procesador 2. y eliminar las columnas de transiciones asignadas en ΛT en Pre

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}, \Lambda_T = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{cc} \Lambda_1 & \Lambda_2 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \end{array}$$

Paso 3. El invariante $[Y^T x\ Pre]_{:3}$ es igual a 0. continuar con el paso 4.

Paso 4. Tomar el invariante $[Y^T x\ Pre]_{:4}$ el cual es distinto de 0 , asignarlo al procesador 3. y eliminar las columnas de transiciones asignadas en ΛT en Pre.

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \qquad \Lambda_T = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

Así, para la RP de la figura2.7, se requiere particionar la red en tres tareas. La tarea 1 toma las transiciones t1 y t2, la tarea 2 toma t3 y t4 y la tarea 3 toma t5 y t6.

2.3.3 Repartición de Lugares de una RP.

La repartición de lugares en una RP puede hacerse de varias maneras. En esta tesis se tomaron en cuenta dos posibles soluciones basándose en p-semiflujos y en la repartición de transiciones vistos en la subsección anterior. En ambas soluciones el número de tareas es el mismo que el número de tareas obtenido en el algoritmo de división de transiciones. A continuación explicaremos cada una de estas soluciones.

Solución 1

Consiste en asignar lugares entre las tareas basándose en p-semiflujos. Esto se realiza en 2 fases:

Fase 1- Asignar lugares no compartidos:

En forma general la tarea 1 toma los lugares no compartidos del p-semiflujo 1, la tarea siguiente intenta tomar los lugares no compartidos aún no asignados pertenecientes al p-semiflujo siguiente pero si no puede obtener lugares de éste ya sea porque ya han sido todos asignados a otras tareas o porque están compartidos intenta obtener los lugares del siguiente p-semiflujo.y así sucesivamente con cada tarea hasta que se asignen todos los lugares no compartidos.

Fase2- Asignar lugares compartidos.

Estos pueden asignarse de modo que cada tarea obtenga todos los lugares compartidos que requiera que aún no estén asignados, sin embargo este proceder no es justo entre las tareas que comparten estos lugares, puesto que es posible que una tarea que comparte lugares con otra, tome los lugares compartidos dejando a la otra sin éstos. Debido a lo anterior se

propone repartir estos lugares en forma de baraja entre las tareas que los requieren haciendo así una repartición más justa.

Ejemplo 2.4 *Sea la RP de la figura 2.6 cuya matriz de p-semiflujos Y y de tareas Λ_t son:*

$$Y = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cc} Y_1 & Y_2 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \end{array} \qquad \Lambda_t = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{cc} \Lambda_1 & \Lambda_2 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \end{array}$$

Fase 1- Asignar lugares no compartidos.
La tarea 1 toma los lugares del p-semiflujo 1 (y1): p1,p2,p3 y p4.
La tarea 2 toma los lugares del p-semiflujo 2 aún no asignados esto es p5, p6 y p7. Note que p2 está en el p-semiflujo 2 pero ya ha sido asignado a la tarea 1.

Fase 2- Asignar lugares compartidos de modo baraja
En esta RP. no existen lugares compartidos. Todos los lugares se repartieron en la fase 1.
El resultado de esta división de lugares puede verse matricialmente de la forma siguiente:

$$\Lambda_L = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cc} \Lambda_1 & \Lambda_2 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \end{array}$$

La RP. particionada se ilustra en la figura 2.8

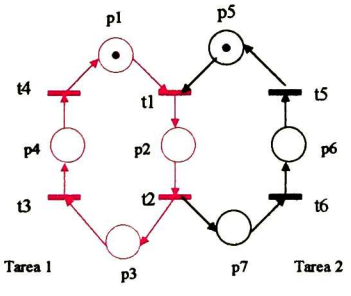


Figura 2.8: Repartición de los elementos de una RP entre dos tareas

Ejemplo 2.5 *Sea la RP de la figura 2.7 cuya matriz de p-semiflujos Y y de tareas Λ_t son:*

$$Y = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 \end{array} \\ \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array} \qquad \Lambda_t = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{array}\right] \end{array}$$

Fase1- Repartir los lugares no compartidos (p1,p2,p3,p4,p5,p6)
La tarea 1 toma los lugares del p-semiflujo 1: p1,p4.
La tarea 2 toma los lugares del p-semiflujo 2: p2,p5.
La tarea 3 toma los lugares p3 y p6 del p-semiflujo 4. Note que esta tarea no toma lugares del p-semiflujo 3 puesto que son lugares compartidos (p7) o ya asignados (p4,p5).

Fase 2- Repartir los lugares compartidos de modo baraja.
La tarea 1 toma el lugar p7, la tarea 2 a p8 y la tarea 3 a p9.

En forma matricial el resultado de esta división se expresa en la siguiente matriz:

$$\begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \end{array}$$

La RP dividida se ilustra en la figura 2.9

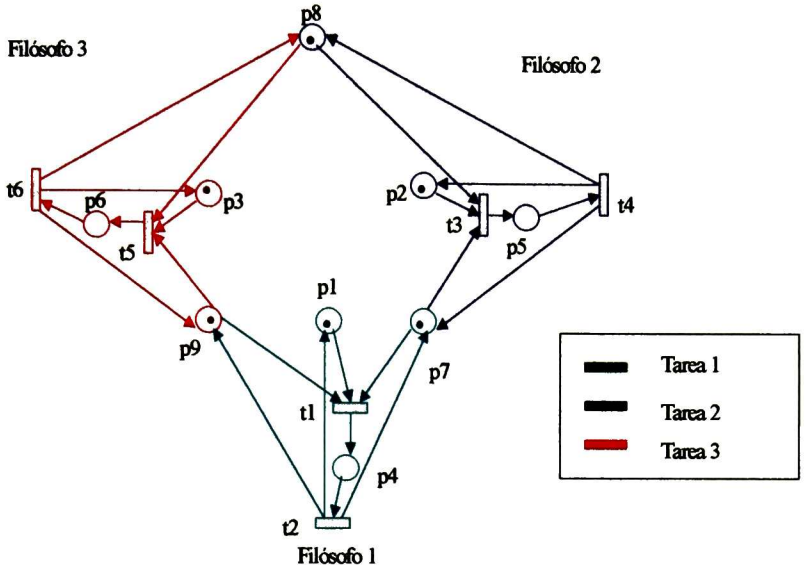


Figura 2.9: Repartición de elementos de una RP que ilustra el problema de "Tres filósofos comensales"

Solución 2

Esta solución reparte los lugares entre tareas en dos fases basándose en la matriz de pre-incidencia de la RP y en el núm. de tareas obtenido en el algoritmo de división de transiciones del modo siguiente:

Fase 1- Repartir lugares no compartidos.

En esta fase cada tarea toma los lugares no compartidos pre-incidentes a las transiciones pertenecientes a dicha tarea si aún no han sido asignados.

Fase 2- Repartir los lugares compartidos de modo baraja entre las tareas involucradas.

Como se mencionó en la solución anterior, la repartición de estos lugares se hace de este modo para una repartición más equitativa de los recursos.

Esta solución es la que se toma en esta tesis debido a que reduce el número de mensajes entre tareas como se verá en el capítulo siguiente.

A continuación se presenta un algoritmo para esta solución:

Algoritmo 2

Pre es la de la matriz de Pre-incidencia de la RP.

Λ_T es una matriz de dimensión $[m, n]$ donde m = número de transiciones y n = número de procesadores o tareas. Un elemento $\Lambda_T[i][j]$ es igual a 1 si y sólo si una transición t_i es asignada a una tarea j, es igual a 0 de otra manera.

Λ_L es una matriz de dimensión $[l, n]$ donde m = número de lugares y n = número de procesadores o tareas. Un elemento $\Lambda_L[i][j]$ es igual a 1 si y sólo si un lugar t_i es asignado al procesador j, es igual a 0 de otra manera. *ntran* es el número de transiciones.

Entradas: Pre, Λ_T

Salidas: Λ_L

lugares asignados = 0
Fase 1
para task= 1 .. número de tareas hacer
  para tran = 1 ..número de transiciones hacer
    si la transición t pertenece a la tarea task ($\Lambda_T[tran][task] = 1$)
    lugar := 1;
      mientras lugar <= número de lugares
        si lugar es pre-incidente a la transición tran y no es compartido
        ($\Pr e[lugar][tran] = 1$ y $\sum_{i=1}^{ntran} pre[lugar][i] = 1$) entonces
        asignar lugar a la tarea task ($\Lambda_L[lugar][task] := 1$)
            lugares asignados:= lugares asignados + 1
              Eliminar disponibilidad de lugar (Hacer fila $Pre_{lugar} = 0$)
        fin si
        lugar:=lugar+1
      fin mientras
    fin si
  fin para
fin para

Fase 2
mientras lugares asignados < número de lugares
 para task= 1 .. número de tareas hacer
   para tran = 1 ..número de transiciones hacer
     si transición tran pertenece a la tarea task ($\Lambda_T[tran][task] = 1$) entonces
       asignado = falso
       lugar = 0
       mientras lugar < número de lugares y asignado = falso
         si lugar es pre-incidente a la transición tran ($\Pr e[lugar][tran] = 1$) entonces
          asignar a la tarea task el lugar ($\Lambda_L[lugar][task] = 1$)
             asignado = verdadero
             lugares asignados = lugares asignados + 1
              Eliminar disponibilidad de lugar (Hacer fila $Pre_{lugar} = 0$)
         fin si
       fin mientras

fin si
fin para
fin para
fin mientras

Ejemplo 2.6 *Para la RP de la figura 2.6 se tiene la siguiente matriz de pre-incidencia Pre y la de tareas en base a transiciones Λ_T obtenida en la subsección anterior:*

$$Pre = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \quad \Lambda_T = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{cc} \Lambda_1 & \Lambda_2 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \end{array}$$

Interpretando la matriz se tienen dos tareas, Λ_1 con transiciones t1,t2,t3,t4 y Λ_2 con transiciones t5 y t6.
Aplicando el algoritmo 2 se tiene:

Fase 1. Repartir a cada tarea los lugares no compartidos:

Paso 1. asignar a la tarea 1 (Λ_1) los lugares pre-incidentes a las transiciones t1,t2,t3,t4 que no sean compatidos ($pre[lugar][tx] = 1$ con $tx = 1 \cup 2 \cup 3 \cup 4$ y $\sum_{i=1}^{6} pre[lugar][i] = 1$) Los lugares que cumplen las condiciones son: p1,p2,p3,p4 y p5, se asignan a Λ_p y se elimina la diponibilidad de estos ($Pre_{lugar} = 0$)

$$Pre = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \quad \Lambda_p = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cc} \Lambda_1 & \Lambda_2 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \end{array}$$

Paso 2. asignar a Λ_2 los lugares pre-incidentes a las transiciones t5 y t6 que no son compartidos ($pre[lugar][tx] = 1$ con $tx = 5 \cup 6$ y $\sum_{i=1}^{6} pre[lugar][i] = 1$) Los lugares que cumplen las condiciones son: p6 y p7 se asignan a Λ_p y se elimina su disponibilidad ($Pre_{lugar} = 0$).

$$Pre = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \quad \Lambda_p = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{cc} \Lambda_1 & \Lambda_2 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \end{array}$$

Fase 2- No se realiza todos los lugares se asignaron en la fase 1.

Interpretando el resultado de la matriz Λ_p se tiene que los lugares $p1$,$p2$,$p3$ y $p4$ y $p5$ pertenecen a la tarea 1 y los lugares $p6$ y $p7$ pertenecen a la tarea 2.

El resultado de la división de la RP se ilustra en la figura 2.10

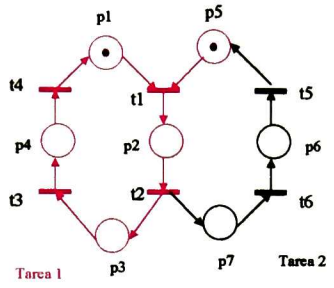


Figura 2.10: Repartición de los elementos de una RP en dos tareas

Ejemplo 2.7 *En laRP de la figura 2.7 cuya matriz de pre-incidencia Pre y de tareas en base a transiciones Λ_T son:*

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array} \quad \Lambda_T = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{array}\right] \end{array}$$

Interpretando la matriz se tienen tres tareas: Λ_1 con transiciones t1,t2, Λ_2 con t3,t4 y Λ_3 con transiciones t5 y t6.
Ahora apliquemos el algoritmo 2.

Fase 1. Repartir lugares no compartidos

Paso 1. asignar a la tarea 1, Λ_1 los lugares pre-incidentes a las transiciones t1,t2 que no sean compartidos ($pre[lugar][tx] = 1$ con $tx = 1 \cup 2$ y $\sum_{i=1}^{6} pre[lugar][i] = 1$) Los lugares que cumplen las condiciones son: p1 y p4, se asignan a Λp y se elimina la disponibilidad de esos lugares(fila $Pre_{lugar} = 0$).

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array} \quad \Lambda p = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \end{array}$$

Paso 2. asignar a la tarea 2 Λ_2 los lugares pre-incidentes a las transiciones t3,t4 que no sean compartidos ($pre[lugar][tx] = 1$ con $tx = 3 \cup 4$ y $\sum_{i=1}^{6} pre[lugar][i] = 1$) Los lugares que cumplen las condiciones son p2, p5. Posteriormente se eliminan de la matriz Pre.

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array} \quad \Lambda p = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \end{array}$$

Paso 3. asignar a Λ_3 los lugares pre-incidentes a las transiciones t3,t4 que sean pre-incidentes a una sola transición ($pre[lugar][tx] = 1$ con $tx = 3 \cup 4$ y $\sum_{i=1}^{6} pre[lugar][i] = 1$) Los lugares que cumplen las condiciones son p3, p6 los cuales se eliminan de la matriz Pre.

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array} \quad \Lambda p = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \end{array}$$

Fase 2- Repartir en forma de baraja los lugares compartidos a las tareas.

Paso 1.Asignar a Λ_1 un lugar pre-incidente a t1 o t2. Los lugares p7 y p9 cumplen la condición,asignamos a p7 y lo borramos de la matriz Pre

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \quad \Lambda p = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

Paso 2. Asignar a Λ_2 un lugar pre-incidente a t3 o t4. El lugar p8 cumple la condición. Asignamos p8 y lo borramos de la matriz Pre

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \quad \Lambda p = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

Paso 3. Asignar a Λ_3 un lugar pre-incidente a t5 o t6. El lugar p9 cumple la condición. Asignamos p9 y lo borramos de la matriz Pre

$$Pre = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \quad \Lambda p = \begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \end{array} \begin{array}{c} \begin{array}{ccc} \Lambda_1 & \Lambda_2 & \Lambda_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

Interpretando el resultado de la matriz resultante Λp se tiene que la tarea 1 contiene a los lugares $p1, p4, p7$, la tarea 2 a $p2, p5, p8$ la tarea 3 a $p3, p6, p9$

El resultado gráfico de la división se ilustra en la figura 2.9. Note que se obtiene el mismo resultado que con el método de la solución 1.

2.4 Conclusiones

El objetivo de este capítulo fue proporcionar una metodología para la realización de un scheduling estructural. Algunos de los puntos que fueron tocados son:

- Se explica la diferencia entre paralelismo en datos y paralelismo estructural.
- Se muestran dos soluciones para realizar un scheduling estructural de RP para su uso en sistemas distribuidos. Estos son una extensión del algoritmo de P. Gutiérrrez

El método de partición puede ser mejorado, esto se debe a que la heurística para tomar el siguiente p-invariante a asignar puede ser cambiada por una más eficiente. La heurística del algoritmo de P. Gutierrez presentada aquí es la de tomar los p-invariantes en el orden que aparezcan en la matriz Y Sin embargo, no siempre se obtiene un número óptimo de procesadores. Lo único que se garantiza es que este número siempre será a lo más la cantidad de p-invariantes mínimos de la red.

Capítulo 3

Transformación de RP para procesos distribuidos

Resumen En esta capítulo se muestran los esquemas de interconexión de sistemas concurrentes en Redes de Petri y se propone su transformación a modelos equivalentes para su uso inmediato en sistemas distribuidos.

3.1 Introducción

Los procesos en los sistemas en general en muchas ocasiones son inherentemente concurrentes y distribuidos. Por ejemplo en un sistema de manufactura un robot en un sitio A podría estar manipulando un material X y al mismo tiempo otro robot en un sitio B estar manipulando un material Y.

Dado que los sistemas distribuidos se ejecutan en unidades independientes para lograr una adecuada cooperación de sus partes y sincronización de éstas, se requiere comunicarción por mensajes haciendo su especificación y control más complejos que en los sistemas centralizados. Este capítulo se orienta hacia la construcción de modelos de sistemas distribuidos.[25]

Para facilitar la especificación de los sistemas distribuidos, partiremos de una especificación centralizada la cual transformaremos en una distribuida añadiendo esquemas de paso de mensajes. Las RP representan bien las especificaciones de trabajo concurrente y la comunicación por mensajes.

Para la tranformación de estos esquemas se requiere además del formalismo de RP para realizar el modelado de sistemas, una técnica de transformación que garantice su completa equivalencia conservando así las propiedades y el buen funcionamiento del esquema centralizado original.

En el capítulo anterior se expusieron metodologías para repartir los elementos de una RP en componentes concurrentes; en este capítulo nos serán de utilidad para realizar nuestra transformación.

En capítulo 1 se expuso el concepto de lugar implícito y un método de reducción propuesto por Koh & DiCesare el cual consiste en eliminar circuitos vivos y acotados CVA de una RP formada por la unión de éstos. Con estos conceptos demostraremos la validez estructural de

la transformación de esquemas centralizados a distribuidos; la validez de comportamiento será demostrada por los lenguajes que se generan.

3.2 Obtención de Esquemas de Interconexión distribuidos

En muchas ocasiones los sistemas concurrentes requieren estar distribuidos ya sea por razones de optimización de rendimiento o porque los elementos que se requieren controlar se encuentran en distintas localidades.

Para realizar especificaciones de sistemas distribuidos proponemos partir de esquemas concurrentes centralizados y transformar estos a esquemas distribuidos sin alterar su comportamiento, ni correctud como demostraremos en el capítulo 4.

Para la transformación hemos clasificado a los esquemas de interconexión concurrente en dos tipos: Los que manejan recursos compartidos y los que manejan sincronización, cualquier otro es una combinación de estos (ver figura 3.1). Para nuestra transformación nos apoyaremos en el esquema de paso de mensajes.

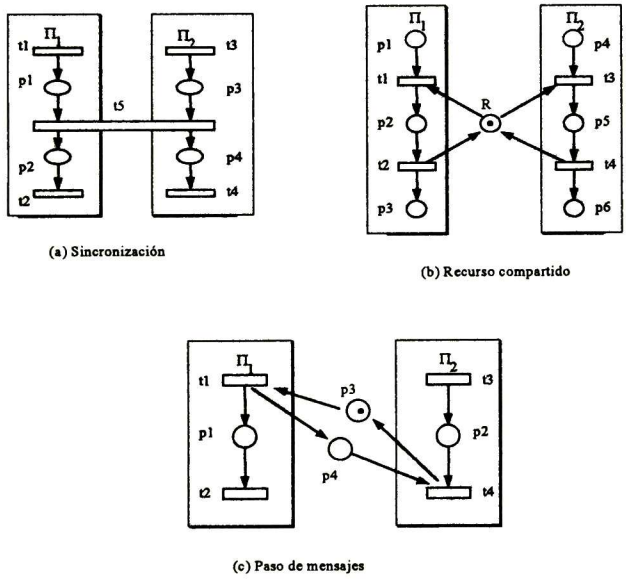


Figura 3.1: Interconexión de esquemas concurrentes

A continuación veremos las transformaciones propuestas de los esquemas de interconexión concurrentes con sus respectivas pruebas de validez.

3.2.1 Esquemas con Recursos Compartidos

La figura 3.1.b muestra un esquema típico de asignación de recursos. El lugar R representa el recurso o grupo de recursos compartidos; los recursos compartidos son entrada a dos o más

transiciones que pertenencen a distintos procesos; el lugar R es entrada a las transiciones t1 y t3 pertenecientes a los procesos Π_1 y Π_2 respectivamente.

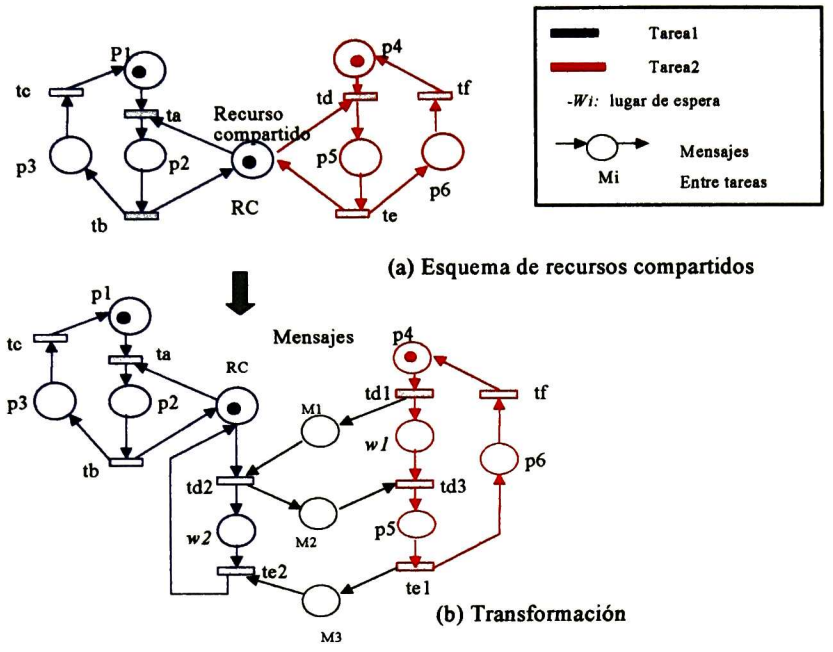


Figura 3.2: Transformación de esquema de recursos compartidos

Transformación

Antes de transformar el esquema se requiere hacer una división explícita de tareas, esto es dividir lugares y transiciones de la RP. En la figura 3.2.a , se muestra una RP dividida en tareas usando los algoritmos 1 y 2 del capítulo anterior. Para la transformación de este esquema se usa el paso de mensajes del siguiente modo:

Cuando una tarea requiere acceder a un recurso compartido el cual no le pertenece debe de solicitarlo a la tarea "propietaria" mandándole un mensaje de solicitud del recurso y espera a recibir un mensaje de autorización, llegado este puede usar el recurso/lugar compartido y al término devuelve el recurso con un mensaje, quedando éste libre para su uso por cualquier otra tarea que requiera utilizarlo. Note que al enviar el mensaje de solicitud se bloquea únicamente la transición que envía dicho mensaje, no toda la tarea.

En la figura 3.2.a se ilustra una RP dividida en dos tareas las cuales comparten un lugar o recurso RC, el cual pertenenece a la tarea 1. En el inciso b se muestra el esquema del inciso a ya transformado, en éste se puede ver como la tarea 1 accede al RC en forma local por ser propietario de éste, pero la tarea 2 accede a este realizando los siguientes pasos:

Solicita el recurso mediante el mensaje M1 y espera la autorización en el estado w1.

Obtiene la autorización al recibir el mensaje M2 desde la tarea1

- Usa el recurso (ejecuta el código de td si éste tiene) y pasa al estado p5.

- Libera el recurso enviando un mensaje M3 a la tarea 1.

Note que la tarea 1 al enviar un mensaje de autorización de uso del recurso a otra tarea se pone en espera (estado w3) hasta obtener el mensaje de liberación de éste antes de volverlo disponible de nuevo.

A continuación veremos las pruebas de validez de la transformación hecha al esquema de recursos compartidos

Validez de la transformación

Para demostrar que la transformación del esquema de recursos compartidos es válida, es suficiente con demostrar la equivalencia en lenguaje y en estructura del ejemplo de la figura 3.2, puesto que toda especificación en RP de recusos compartidos realizada con el método de [14] y [15] usado en esta tesis mantiene la misma estructura que la de la figura 3.2.a

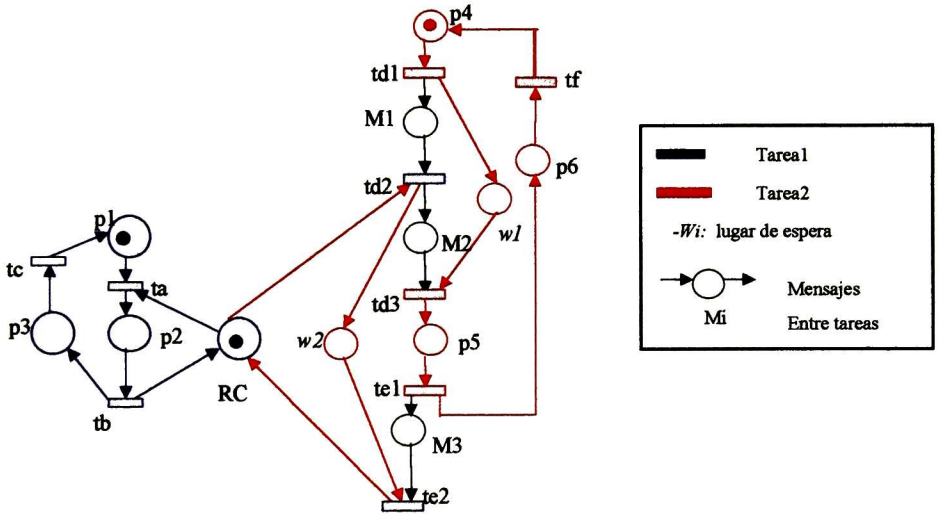


Figura 3.3: Esquema distribuido de recursos compartidos

- *Equivalencia en Lenguaje:*

Sea ta =a, tb=b, tc=c,td=d te=e tf=f en la RP original (figura 3.2.a)

y ta1 = a ,ta2=ε, ta3=ε, tb1 =b, tb2=ε,tc=c,td=d te=ε tf=f en la red transformada (figura 3.2.b), donde ε es símbolo nulo.

El lenguaje producido por la red original es: (abc)* $\cup$ (def)*

y en la red transformada: (a$\varepsilon\varepsilon$bεc)*$\cup$(def)*, el cual es equivalente a: (abc)* $\cup$ (def)* de la red original

- *Equivalencia Estructural*

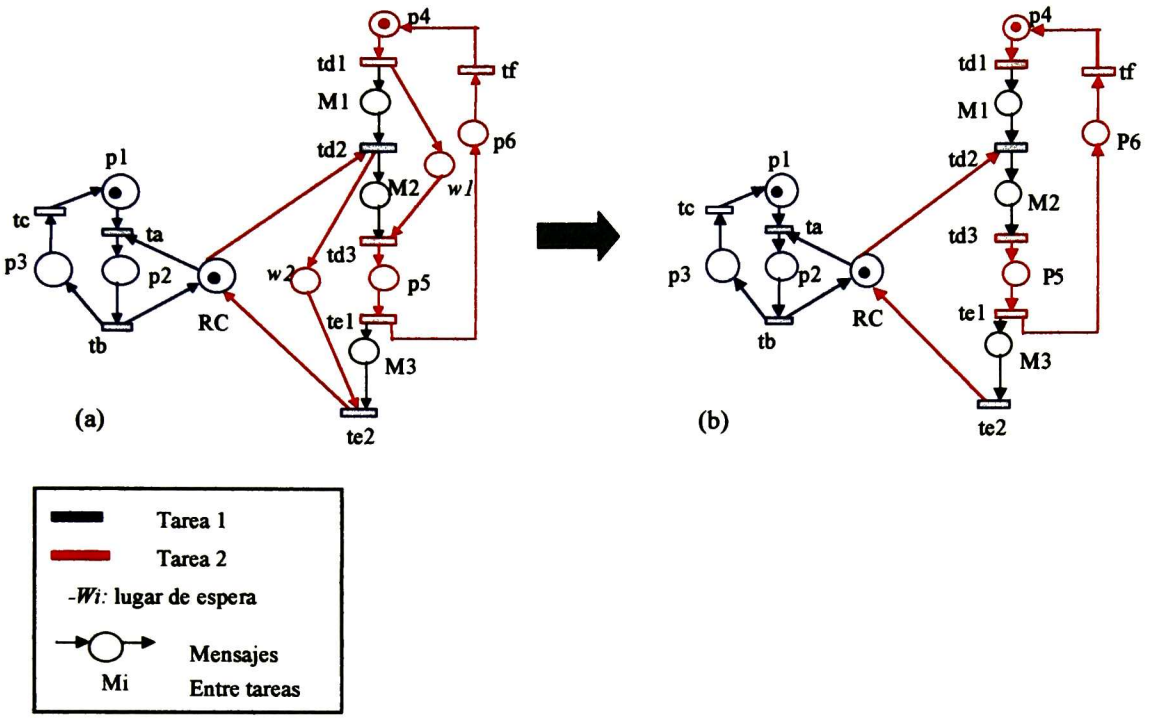


Figura 3.4: Reducción de RC por eliminación de lugares implícitos

Para facilitar la demostración redibujamos la red transformada de la figura 3.2 obteniendo la figura 3.3

A continuación procederemos a la demostración reduciendo la RP redibujada apoyándonos en la definición de lugar implícito y de los teoremas de los resultados de [10]:

1.- Haciendo uso de la definición de lugar implícito eliminamos los lugares w1 y w2 de la figura 3.3. (Ver figura 3.4)

2.- Usando el Teorema 3 de CVA de [10] procedemos a remover circuitos vivos y acotados de la RP obtenenida en el paso 1: (En este paso siga la figura 3.5)

a) Eliminamos el circuito p1-ta–p2-tb-p3-tc-p1(vea inciso a y b)

b)Eliminamos el circuito RC-ta-p2-tb-RC.(vea inciso b y c)

c)Eliminamos el circuito RC-td2-M3-p3-te1-M3-te2-RC (vea inciso c y d)

La RP resultante (inciso d) nos queda reducida en un CVA Por tanto por el Teorema 3 de Koh & DiCesare la RP de la figura 3.3 es una red válida, y por consecuencia también la red de la figura 3.4 lo es, con lo que podemos decir que la transformación hecha al esquema de recursos compartidos no afecta las propiedades estructurales de la red original.

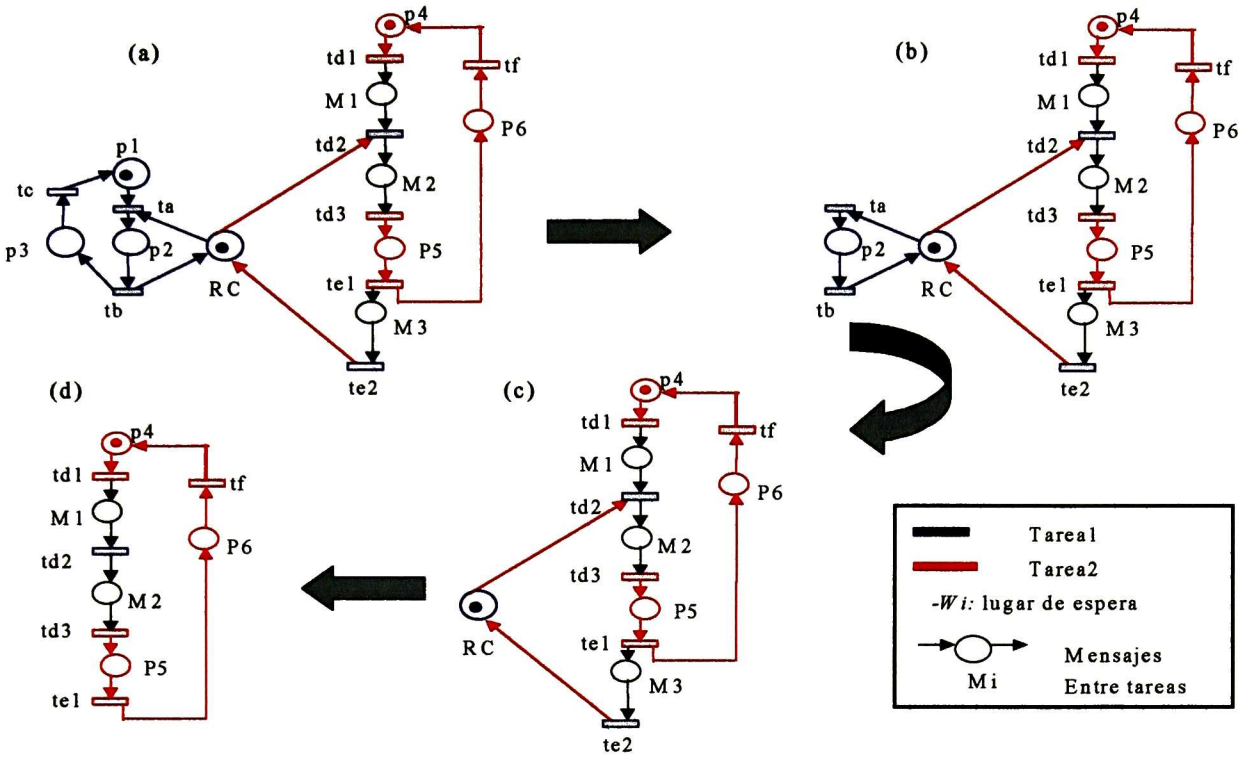


Figura 3.5: Reducción de esquema de recursos compartidos distribuidos

3.2.2 Esquemas de Sincronización

Estos esquemas modelan procesos en los que el inicio de ciertas etapas están sincronizadas, es decir en las que se necesita que dos o más actividades estén en un determinado estado para proseguir. Ver esquema en la figura 3.1.a

Las tareas en una RP se sincronizan por medio de sus transiciones. Una tarea en un punto de sincronización (transición) cumple su sincronía si sus lugares de entrada tienen las suficientes marcas para habilitar dicha transición. Además una tarea con sincronización en algún punto sólo puede continuar si ha logrado la sincronía en ese punto.

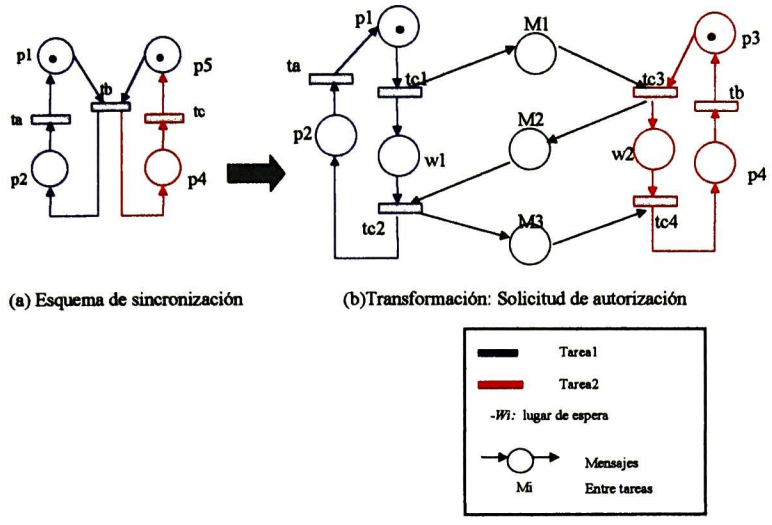
Transformación

Para la transformación de esquemas de sincronización mostraremos tres posibles soluciones.

- **Solución 1- Solicitud de Autorización**

En este método de transformación se requiere una previa división explícita de tareas en p-semiflujos como se muestra en la RP de la figura 3.6.a.

El comportamiento de transformación de este esquema es el siguiente: la tarea "propietaria" de la transición de sincronización debe saber si los lugares no pertenecientes a la



(a) Esquema de sincronización (b)Transformación: Solicitud de autorización

Figura 3.6: Sincronización distribuida "Solicitud de autorización"

tarea "propietaria" de la transición, pre-incidentes a ésta la habilitan para poder continuar, esto lo hace por medio de mensajes solicitando un permiso de continuar (ver figura 3.6.b). Esto es:

Para cualquier tarea Ti, siendo t1 una transición de sincronización perteneciente a Ti y Tj cualquier otra tarea distinta a Ti con algún lugar de entrada (pre-incidente) a t1, se realizan los pasos siguientes:

1- Ti verifica que los lugares de entrada a t1 pertenecientes a Ti, habiliten a t1.

2- Ti manda un mensaje M1 de solicitud de continuar a toda tarea Tj y espera los mensajes de respuesta en el estado w1.

3- Ti recibe mensajes M2 con la autorización desde las tareas Tj

(Al término de estos pasos se cumple la sincronía de la transición.)

5- Ejecutar código de t1 (El código puede ser nulo)

6- Envía un mensaje M3 a toda tarea Tj para que éstas continúen.

Note que esta solución usa el principio de la transformación de esquema de recursos compartidos, pues los mensajes de solicitud de continuar desde Ti a toda Tj, es equiparable al de solicitar recurso, así también la tarea espera una respuesta de autorización, como en el esquema de recursos compartidos; cuando obtiene el recurso, luego ejecuta el código de la transición y manda un mensaje de continuar a toda Tj, equiparable a devolver recurso.

- **Solución 2- Envío de avisos**

En este método de transformación se requiere una previa división explícita de tareas en p-semiflujos como se muestra en la figura 3.7.a.

El comportamiento de transformación de esta solución es el siguiente:

Caso 1- La tarea es dueña de la transición de sincronización.

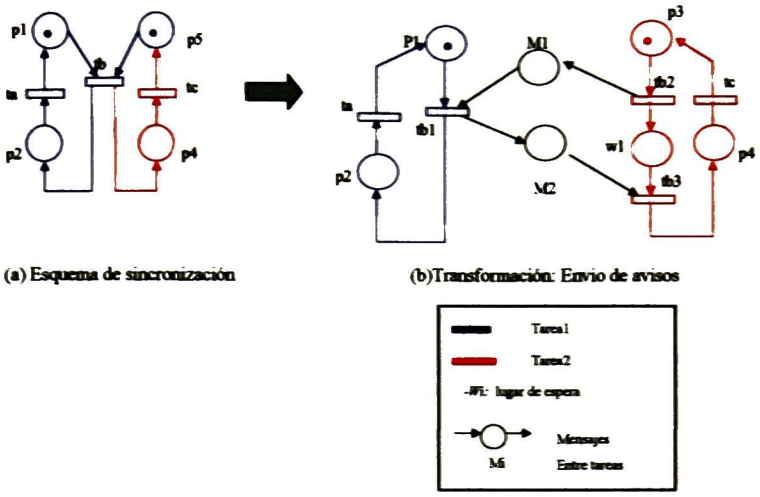


Figura 3.7: Transformación de esquema de sincronización "envío de avisos"

1- Validar que todos sus lugares de entrada la habiliten

En el caso de los lugares pre-incidentes a la transición no pertenecientes a la tarea, recibe mensajes de aviso que le indican que éstos ya están en el estado que habilita a dicha transición.

2- Ejecutar código asociado a la transición, si ésta lo tuviera

3- Marcar lugares de salida.

En el caso de los lugares de salida no pertenencientes a la tarea en cuestión, envía un mensaje de continuar a cada tarea "propietaria" de éstos para marcarlos.

Caso 2: La tarea no es dueña de transición de sincronización, pero tiene lugares pre-incidentes (de entrada) a dicha transición.

En este caso cuando sus lugare pre-incidentes a la transición de sincronización la habiliten envia mensajes de aviso a la tarea "propietaria" de ésta para informarle el evento.

En la RP de la figura 3.7.a se tienen dos tareas sincronizadas por la transición tb de la tarea 1. En la figura 3.7.b la tarea 2 envía un mensaje M1 siempre que p3 habilite a la transición tb1 y la tarea1 logra su sincronía siempre que p1 y p3 la habiliten, al término de la ejecución de tb1, la tarea 2 envía un mensaje M2 de continuar a la tarea 2.

- **Solución 3- Mensajes Cruzados**

En este método de transformación se requiere una previa división explícita de tareas en p-semiflujos como se muestra en la RP de la figura 3.8. El comportamiento de transformación de este esquema por cada tarea es el siguiente:

Si la transición de sincronización está habilitada por los lugares pre-incidentes a ella que pertenecen a la misma tarea que dicha transición, hacer lo siguiente:

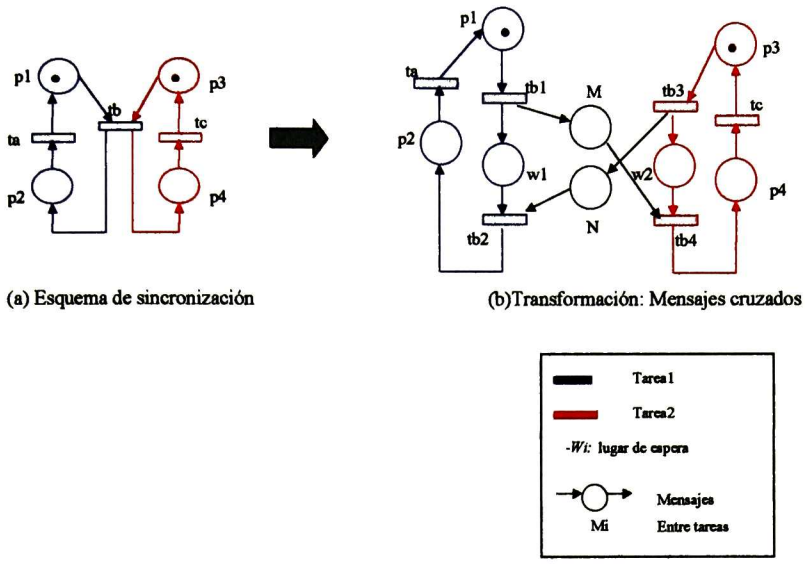


Figura 3.8: Transformación de esquema de sincronización "Mensajes cruzados"

1- Enviar mensajes de aviso M a todas las otras tareas con lugares pre-incidentes a dicha transición.

2- Esperar en el estado w hasta recibir todos los mensajes de aviso N de todas las otras tareas que tengan lugares pre-incidentes a dicha transición.

3- Continuar.

Esta solución es válida sólo cuando la transición de sincronización no tiene código, pues si esta lo tuviera, las tareas no "propietarias" de esta transición continuarían antes que la tarea "propietaria" terminara de ejecutar el código de la transición perdiéndose parte de la sincronía.

En la RP de la figura 3.8 se tienen dos tareas sincronizadas por la transición tb perteneciente a la tarea 1.La tarea 1 envía un mensaje M a la tarea1 siempre que p1 habilite a la transición tb1, de igual modo la tarea 2 envía un mensaje N a la tarea 1 siempre que p3 habilite a tb3. En este punto la tarea 1 espera en el estado w1 hasta recibir el mensaje N desde la tarea 2 y a su vez la tarea 2 espera en el estado w2 hasta recibir el mensaje M desde la tarea 1; cuando esto sucede, las tareas se encuentran sincronizadas, entonces las tareas continúan su ejecución.

- **Solución 4- Marcar post-incidentes**

En este método de transformación requiere que se dividan explícitamente las tareas utilizando la técnica descrita en el capítulo 2 (algoritmos 1 y 2). Esto es, las transiciones se reparten en p-semiflujos y los lugares se reparten en las tareas donde existen transiciones a las que son pre-incidentes.(Ver figura 3.9.a)

Dada la repartición de lugares, todo lugar no compartido se encuentra en la misma tarea que la transición a la cual entra; ésto lleva a no necesitar mensajes para habilitarla. El único

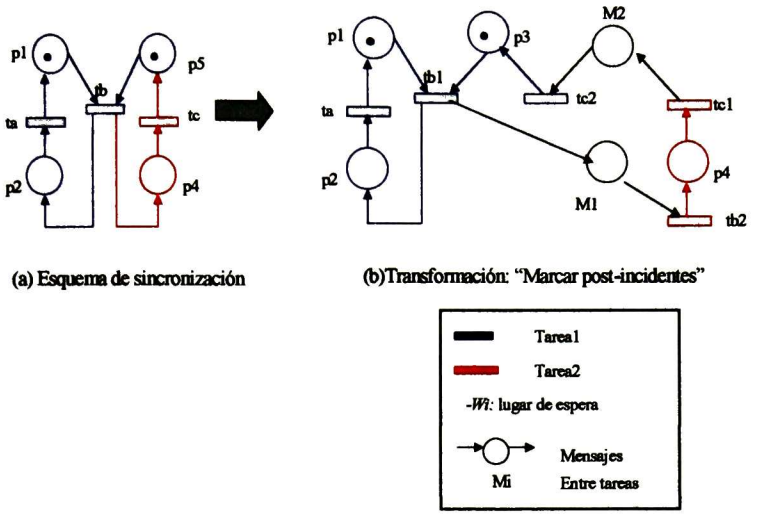


Figura 3.9: Transformación de esquema de sincronización: "Marcar post-incidentes"

tipo de mensaje requerido es el que se utiliza para marcar a los lugares de salida de una transición, que no se encuentran en la misma tarea que ésta.

En la figura 3.9.a se tienen dos tareas y una transición de sincronización tb perteneciente a la tarea 1. En la figura 3.9.b la tarea 1 envía un mensaje M1 para marcar el lugar p4 perteneciente a la tarea 2 y la tarea 2 envía un mensaje M2 para marcar el lugar p3 de la tarea 1.

Note en 3.9.a que p4 es lugar post-incidente de la transición tb y p5 es lugar post-incidente de la transición tc.

Solución Escogida

La solución 1 (solicitud de autorización) fue descartada, en cuanto a que maneja un mayor número de mensajes que las otras tres soluciones; la solución 2 (envío de avisos) fue descartada, en cuanto hace diferencia de comportamiento entre si la tarea es o no "propietaria" de la transición, perdiendo así generalidad; la solución 3 maneja un número de mensajes mayor o igual que las soluciones 2 y 4, aunque es la que usa menor tiempo en intercambio de mensajes, por manejarlos simultáneamente, fue descartada, debido a que sólo es válida para sincronizaciones mediante una transición sin código asociado.

La solución 4 (marcar post-incidentes) maneja un solo tipo de mensaje en las tareas sin importar si es "propietaria" o no de la transición de sincronización, los cuales son para marcar lugares post-incidentes no pertenecientes a la tarea en cuestión, lo cual equivale al mensaje devolver recurso del esquema distribuido de recursos compartidos, simplificando así el código que se generará para la RP en el siguiente capítulo; además esta solución tiende a agilizar la habilitación de las transiciones tiende debido a que todos los lugares no compartidos pre-incidentes a la transición de sincronización pertenecen a la misma tarea que dicha transición. Por estas razones se prefiere la solución 4 para transformar esquemas de sincronización.

A continuación veremos la demostración de validez de esta solución.

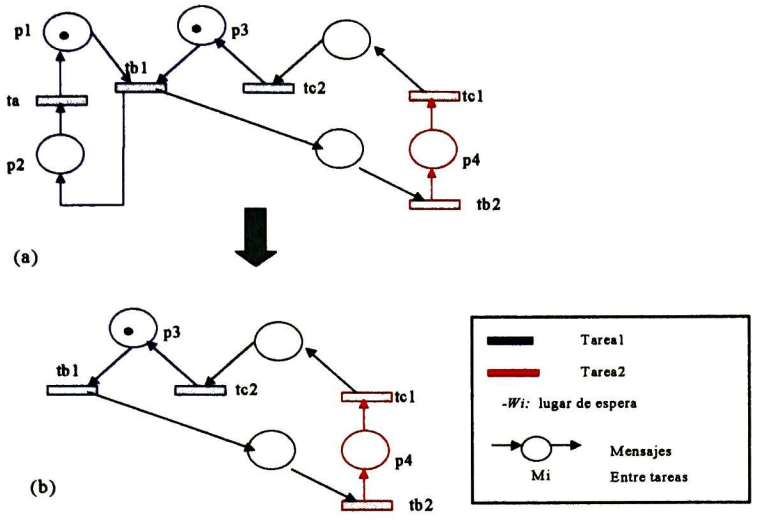


Figura 3.10: Reducción de esquema "Marcar Remoto"

Demostración de validez (solución 4)

Para demostrar que la transformación de la red es válida demostraremos su equivalencia en lenguaje y en estructura.

Equivalencia en Lenguaje:

Sea Ta =a, Tb=b, Tc=c, en la RP original (figura 3.9 inciso a)

y Ta = a ,Tb1 =b,Tb2=ε,Tc1=ε,Tc2=c en la red transformada (figura 3.9 inciso b), donde ε es caracter nulo.El lenguaje producido por la red original es: (bca)* $\cup$ (bac)* y en la red transformada: (b$\varepsilon\varepsilon$ca)*$\cup$(ba$\varepsilon\varepsilon$c)*, el cual es equivalente a: (bca)*$\cup$ (bac)* de la red original.

Equivalencia Estructural:

Usando el Teorema 3 de CVA de Koh & DiCesare visto en el capítulo 1 procedemos a remover circuitos vivos y acotados de la RP obtenida en el paso 1: (En este paso siga la figura 3.10- a) Eliminamos el circuito p1-tb1–p2-p1 (fig. 3.10- b). La RP resultante nos queda reducida en un CVA Por tanto por el Teorema 3 de Koh & DiCesare la RP de la figura 3.10.a es una red válida, con lo que podemos decir que la transformación hecha al esquema de sincronización no afecta las propiedades estructurales de la red original.

3.2.3 Ejemplos de transformación

En los siguientes ejemplos se mostrarán la tranformación de un modelo sencillo de RP usando el método de transformación para recursos compartidos y el de sincronización "marcar post-incidentes" Para su transformación usamos mensajes con la nomenclatura siguiente: Mij es un mensaje desde la tarea i a la tarea j.

Ejemplo 1

El sistema de este ejemplo consiste en dos procesos generales que actualizan archivos, tomando éstos desde un servidor de archivos compartidos y pidiendo los nuevos datos al usuario para su actualización. Cada proceso a su vez es dividido en dos subprocesos uno que trae el archivo desde el servidor y el otro que obtiene los datos desde el usuario y actualiza el archivo. Este sistema se encuentra modelado en la RP de la figura 4.8, la partición de tareas se obtiene de aplicar los algoritmo 1 y 2 del capítulo 2. La interpretación de claves de lugares y transiciones del modelo es la siguiente:

RC es el servidor de archivos compartido entre las tareas 3 y 4.

t1, t5 Obtener datos desde el usuario.

t2, t6 Sincronización de archivo obtenido y datos del usuario.

t3, t7 Actualización de archivo.

t4, t8 Fin de actualizar (reiniciar proceso de actualización y desconectarse de servidor).

t9, t11 Conectarse a servidor.

t10, t12 Obtener archivo desde servidor.

p1,p5 Lugares de inicio de las tareas 1 y 2 respectivamente.

p2,p6 Datos de usuario ya obtenidos.

p3,p7 Archivo y datos de usuario ya obtenidos.

p4,p8 Archivo actualizado.

p9,p12 Lugares de inicio de las tareas 3 y 4 respectivamente.

p10,p13 En conexión con el servidor.

p11,p14 Archivo obtenido.

p5 Servidor compartido (RC)

La red transformada se encuentra en la figura 3.12.

Los mensajes M43 y M34 se utilizan para pedir y obtener el uso del servidor compartido, el mensaje M23 libera el uso del servidor. Note que usamos el método de transformación de esquemas para recursos compartidos.

Los mensajes M31 y M13 se utilizan para realizar la de sincronización entre las tareas 1 y 3, aquí se manejó el método de trasformación de esquemas de sincronización por "marcar post-incidentes".

Los mensajes M42 y M24 se utilizan para realizar la sincronización entre las tareas 2 y 4 de igual modo que entre las tareas 2 y 3.

Ejemplo 2

En este ejemplo ilustramos el problema de los filósofos comensales para tres filósofos el cual consiste en tres filósofos sentados alrededor de una mesa redonda, cada uno con un plato. Tres tenedores están compartidos colocados uno entre cada plato; la secuencia de cada filósofo

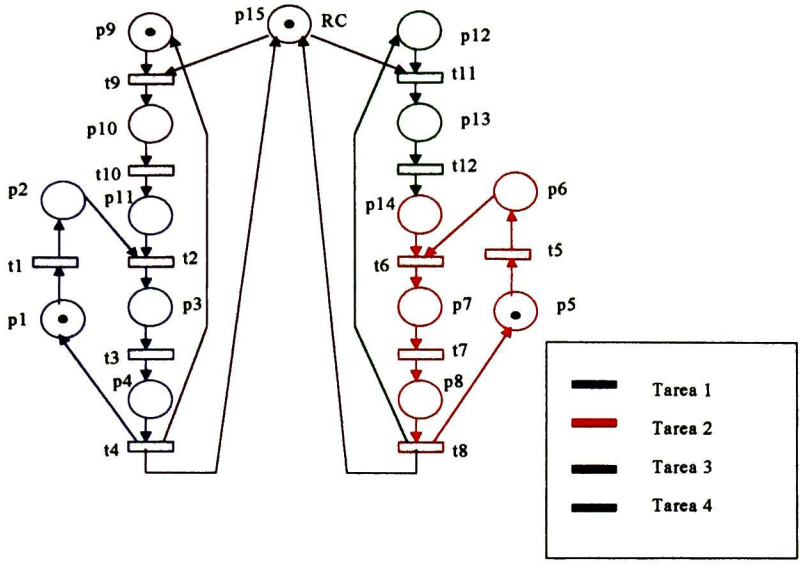


Figura 3.11: RP que ilustra un sistema de actualización de archivos con uso de servidor compartido y sincronización de tareas

es pensar, comer, pensar..., cada filósofo para comer necesita el tenedor de su derecha y el de su izquierda, al término de esta actividad devuelve los tenedores y regresa a pensar.

Este ejemplo se encuentra modelado en la RP de la figura 3.14, la partición de tareas se obtiene de aplicar los algoritmo 1 y 2 del capítulo 2, observe que se obtienen una tarea por cada filósofo. La interpretación de claves de lugares y transiciones del modelo es la siguiente:

p7,p8,p9- tenedores, recursos compartidos.

p1- Filósofo 1 pensando.	p4- Filósofo 1 comiendo.
p2- Filósofo 1 pensando.	p5- Filósofo 2 comiendo.
p3- Filósofo 1 pensando.	p6- Filósofo 3 comiendo.

t1- comer: Filósofo 1.	t2- pensar: Filósofo 1.
t3- comer: Filósofo 2.	t4- pensar: Filósofo 2.
t5- comer: Filósofo 3.	t6- pensar: Filósofo 3.

La red transformada se encuentra en la figura 3.14.

Los mensajes M13a, M21a y M32a se utilizan para pedir recursos compartidos (tenedores) no propios; los mensajes M31,M12 y M23 se usan para obtener los recursos compartidos solicitados; Los mensajes M13b, M21b y M32b se utilizan para devolver dichos recursos compartidos. Los lugares w son lugares de espera.

3.3 Conclusiones

En este capítulo se mostró cómo trasformar una especificación en RP a un modelo equivalente, en el cual se distinguen los procesos que se ejecutarán en distintos procesadores.

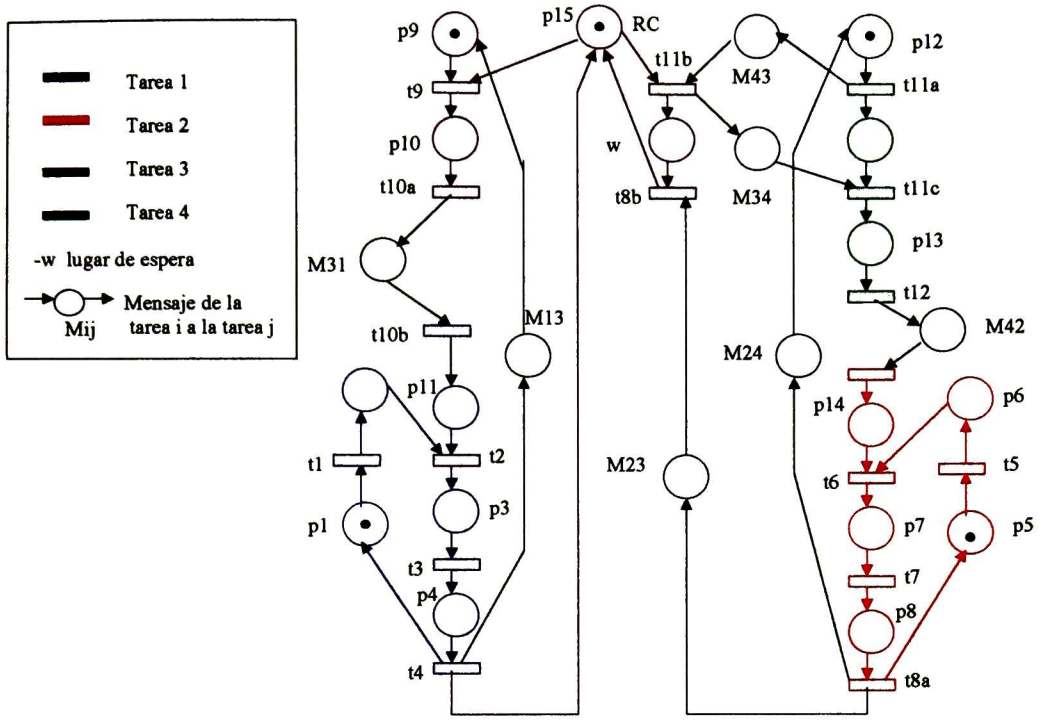


Figura 3.12: RP que ilustra un sistema distribuido de actualización de archivos

Esta transformación parte de una especificación en RP centralizada, cuyos elementos han sido repartidos en procesos, como se realizó en el capítulo 2, posteriormente se realiza la transformación en una especificación distribuida equivalente.

Las interacciones entre procesos para sincronización o para asignación de recursos, son modelados por mecanismos de paso de mensajes. No se tomaron en cuenta otros esquemas, debido a que cualquier otra relación entre procesos es considerada una combinación de éstas. Así también se demostró la validez de estas transformaciones por el lenguaje que generan y por su estructura apoyándose en los resultados de Koh & DiCesare vistos en el capítulo 1 y por último se ilustró dos ejemplos de transformación de pequeños sistemas.

La transformación de esquemas de sincronización mostrada en éste capítulo, es buena para las sincronizaciones donde la transición de sincronía contiene código asociado a ella; pero no es óptima para cuando no existe dicho código. Para este esquema, es más eficiente la transformación por mensajes cruzados, debido a que reduce tiempo de paso de mensajes.

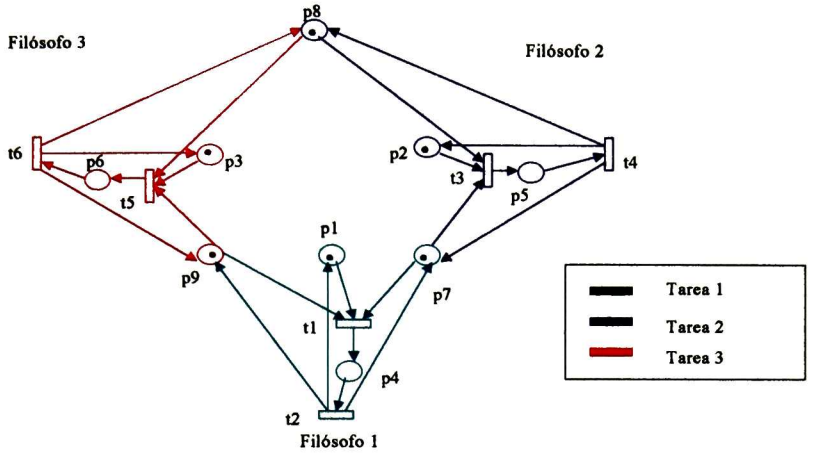


Figura 3.13: RP que ilustra el problema de "Tres filósofos comensales"

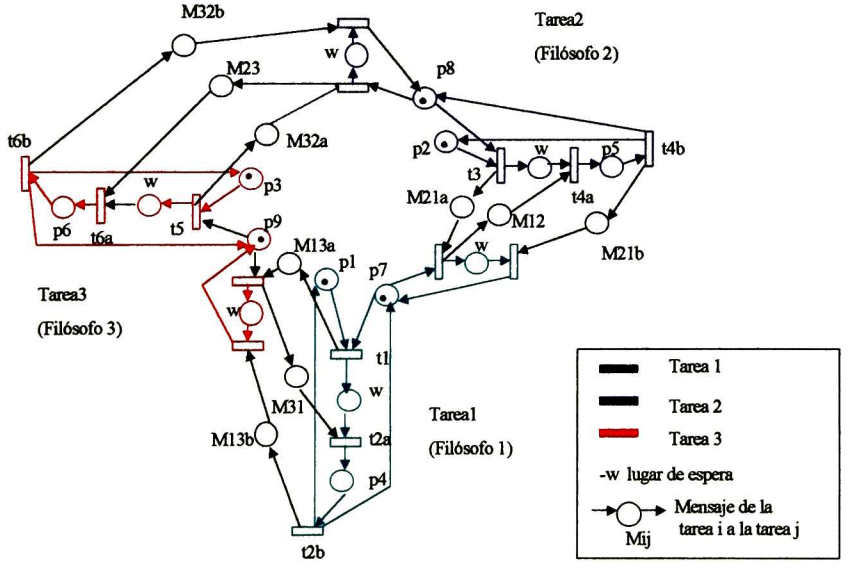


Figura 3.14: RP que ilustra un sistema distribuido del problema de "Tres Filósofos comensales".

Capítulo 4

Generación de Software Distribuido

Resumen En este capítulo se muestra la manera de obtener la codificación de un sistema a partir de una especificación expresada en RP, así también se describe la herramienta visual de especificación de RP donde se implementó dicha codificación.

4.1 Java como lenguaje de programación

En esta tesis el código que se genera a partir de una especificación aparte en RP es código Java, el cual fue seleccionado por numerosas razones.

- Primero. Dado que Java es un lenguaje de programación interpretado, es portable para un amplio número de plataformas de hardware y de sistemas operativos, actualmente existen intérpretes para IBM, MacOS,Apple Solaris, HP Windows 95-98- NT, Unix-Linux, Irix, Aix etc. Esto permite el uso de este código a un amplio tipo de usuarios finales.

- Segundo. Java es un lenguaje orientado a objetos, hoy en día la tendencia de las aplicaciones es en dirección de la tecnología de objetos, dado sus beneficios como su naturalidad y su manejo de herencia para el reuso de código.

- Tercero. Java maneja programación distribuida. Java se ha construido con extensas capacidades de interconexión TCP/IP permitiendo el acceso a información tanto a través de una red local como através del internet. Nuestra aplicación generada a partir de una especificación en RP es distribuida.

- Cuarto. Java es Multitarea o Multihilo, esto le permite relizar muchas actividades simultáneas un un programa.[21].

La estructura de un programa Java es el siguiente:

class nombre_de la clase{

atributos o datos

-métodos o funciones

}

En las siguientes secciones comentaremos el código Java con su sintaxis propia, esto es:

//comentario de línea

/*comentario de bloque*/

4.2 Java y la programación Multitarea

Java es un lenguaje que permite la programación de varias tareas que se ejecutarán simultáneamente.

En el lenguaje Java un sistema multitarea es llamado multihilo (multithread) y cada proceso o tarea individual ejecutándose en el sistema se le denomina hilo de ejecución (thread). Cada hilo controla un único aspecto dentro del programa. Gráficamente los hilos se parecen en su funcionamiento a lo que se muestra en la figura 4.1.

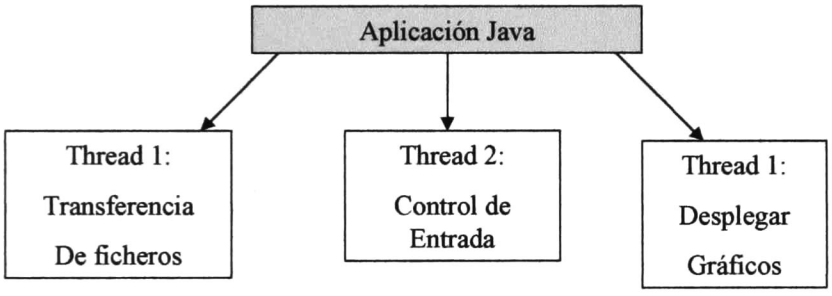


Figura 4.1: Hilos en Java (Threads)

4.2.1 Sincronización de hilos

Otra clave del éxito y la ventaja de utlizar múltiples tareas (hilos) en una aplicación, es que pueden comunicarse entre sí. Se pueden diseñar tareas que puedan utilizar objetos comunes de modo independiente pero también sincronizado. Estos objetos compartidos son protegidos por medio de monitores, los cuales se acompañan de seguros.

En Java cada objeto con métodos sincronizados es un monitor. Un hilo sólo puede ejecutar un método sincronizado a la vez y si hay varios métodos sincronizados sólo uno puede estar activo en un objeto a la vez, todos los demás hilos intentando entrar a un monitor (método sincronizado) deben esperar. Cuando un método sincronizado termina su ejecución el seguro en el objeto del monitor se libera quedando libre para su uso por algún hilo con la mayor prioridad que invoque a este método.

Un hilo ejecutándose en un método sincronizado puede determinar que no puede proseguir y se pone en estado de espera (wait), esto ocasiona que el hilo libere al procesador y se salga del monitor y se pone en una cola de hilos en espera de algún evento. Cuando un hilo termina la ejecución de un método monitor puede notificar por medio de eventos (notify)

para que los hilos en espera puedan quedar listos para ejecución. La notificación actúa como una señal, la pareja wait-notify equivale a un semáforo.

El ejemplo clásico de la sincronización entre tareas es un modelo productor/consumidor. Una tarea produce una salida que otra tarea usa (consume), sea lo que sea. Por ejemplo considere un productor que escribe caracteres a su salida; y sea un consumidor que los lee y un monitor que controla el proceso de sincronización entre productor-consumidor. Ver figura 4.2

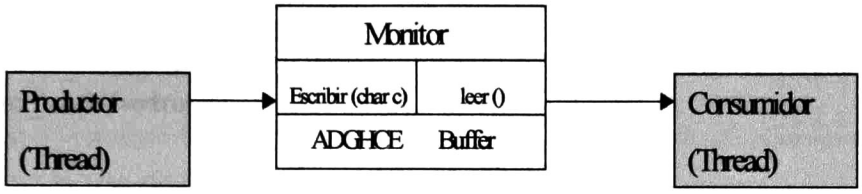


Figura 4.2: Modelo productor-consumidor

Este ejemplo podría ser representado por una RP como en la figura 4.3

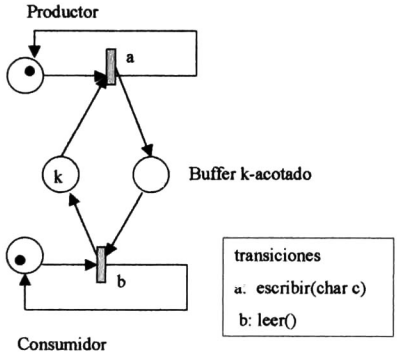


Figura 4.3: RP productor-consumidor

El código del monitor del modelo productor/consumidor ilustrado en la figura 4.2 podría ser escrito de la forma siguiente:

```
class Monitor{
public synchronized void escribir (char c){
while (buffer_lleno == true){
  try{
   wait(); //esperar
  catch(InterruptedException){...}
}
```

```
buffer[siguiente] = c;
siguiente = siguiente + 1;
if (siguiente == k)
 buffer_lleno = true;
buffer_vacio = false;
notify(); //notifica estado del buffer
}
public  synchronized recoger(){
while (buffer_vacio==true){
 try{
  wait() //esperar
  catch(InterruptedException){...}
}
 siguiente=siguiente - 1;
 if (siguiente == 0){ buffer_vacio = true}
 buffer_lleno = false;
 notify(); //notifica estado de buffer
 return buffer[siguiente];
}
}
```

//Código del productor

```
class productor extends Thread{
Monitor m;
productor(Monitor monitor){ m = monitor;}
public void run(){
...
m.escribir(caracter);
...
}
}
```

//Código del consumidor

```
class consumidor extends Thread{
Monitor m;
consumidor(Monitor monitor){ m = monitor;}
public void run(){
```

```
...
m.recoger()
...
}
}

//código de la aplicación
class productor-consumidor{
 public void main(String args[]){
   Monitor m = new monitor();
   productor p = new productor(m);
   consumidor c = new consumidor(m);
    p.start();
    c.start();
 }
}
```

Los métodos de acceso sincronizado impiden que los productores y consumidores corrompan un objeto compartido. Mientras el productor está añadiendo una letra al buffer el consumidor no la puede retirar y viceversa. Esta sincronización es vital para mantener la integridad de cualquier objeto compartido.

4.3 Comunicación Distribuida en Java

En este apartado no se pretende profundizar sobre la comunicación distribuida, el objetivo es dar una idea clara del flujo de información entre procesos distribuidos.

Una red de computadores es un conjunto de máquinas o dispositivos que están interconectados a través de un medio que les permite intercambiar datos. Sabemos que los componentes de un sistema distribuido se encuentran repartidos en una red de computadores interconectados físicamente entre si. Estas partes se comunican intercambiando mensajes.

Java maneja para las comunicaciones el esquema de cliente-servidor. La función del cliente es solicitar algún servicio o información a un servidor y la de éste es proporcionar el servicio o información solicitada por los clientes.

4.4 Estructura de Código Java Distribuido

Como vimos en los capítulos anteriores, convertimos un modelo global en RP a un modelo distribuido, esto es dividimos la RP en tareas independientes relacionadas entre si por mecanismos de sincronización o/y recursos compartidos. En un ambiente distribuido cada una de estas tareas será ejecutada en un computador o dispositivo de una red de estos. Cada

tarea actuará de forma independiente y se comunicará con las demás tareas por medio de mensajes siempre que requiera sincronizarse o necesite usar algún recurso compartido que se encuentre en otra tarea. Ver figura 4.4

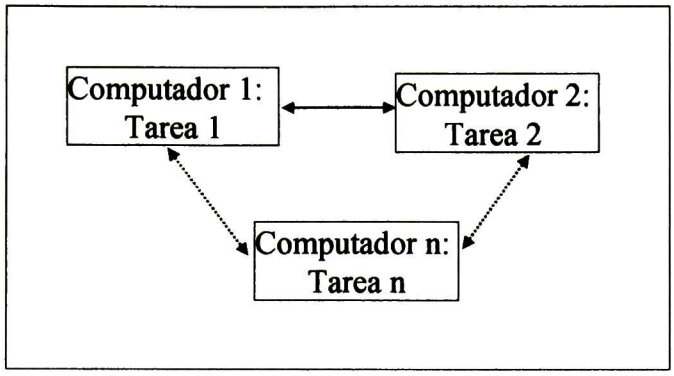


Figura 4.4: Tareas distribuidas

Note que al ejecutarse las tareas de modo independiente en distintos computadores se maximiza el paralelismo.

Para la comunicación entre tareas cada una de éstas tiene un módulo servidor para dar servicio a otras tareas y un módulo de clientes para solicitar datos o servicios a sus homólogas.

Asimismo la ejecución de cada tarea consiste en la ejecución de sus actividades representadas por sus transiciones las cuales también deben de realizarse con la mayor concurrencia posible, esto es si existen dos o más transiciones habilitadas en una misma tarea, éstas pueden ejecutarse concurrentemente (reentrancia de la tarea o concurrencia local); además una misma transición puede estar ejecutándose más de una vez simultáneamente si su dehabilitación lo permite.

Para lograr la independencia de las transiciones, éstas se manejan como flujos de ejecución independientes(hilos o threads) y para mantener la sincronía e integridad de la tarea usa un monitor donde se ponen las partes comunes (Clase Shared). Ver figura 4.5

Toda transición tiene un comportamiento general, el código de estas varía según sus entradas/salidas, condición y código como se verá más adelante.

4.4.1 Codificación de transiciones

La primera parte de esta subsección explica la generación del código del comportamiento general de una transición el cual permanece constante, en la segunda parte se da una ligera idea del código de las transiciones particulares las cuales variarán de acuerdo al modelo. En la siguiente subsección se verá el algoritmo para obtener el código de cada transición

Parte I.

En la figura 4.6 se ilustra el comportamiento general de código asociado a una transición.

Tarea i		
Area de datos: Contiene datos de la tarea.		
Area de Transiciones Hilo ta Hilo tb ... Hilo tx Donde ta,tb,tx son transiciones pertenecientes a la tarea i. Función: ejecuta las actividades de la tarea i	Area de Sincronización Monitor(Shared) Función:Sincroniza la ejecución de la tarea	Area de comunicaciones Cliente-Conexión Servidor Función: Comunica a la tarea con sus análogas

Figura 4.5: Partes de una tarea

Note que una misma transición puede ejecutarse más de una vez simultáneamente si cumple su condición de activación (reentrancia). En la RP de la figura 4.7 se tiene un marcado que ilustra la concurrencia en la ejecución de una transición o/y bien de transiciones distintas.

A continuación explicaremos las actividades de una transición en general ilustradas en el diagrama de flujo 4.6

Condición para habilitar transiciones

Sabemos que la condición para que una transición t_a esté habilitada es que los marcados M de sus lugares pre-incidentes la habiliten, esto es:

si $M(p_i) \geq Pre(a,i)$ para todo p_i tal que $Pre(a,i) \geq 1$ ta está habilitada

Desmarcar lugares pre-incidentes

Consiste en reducir el marcado M de todo lugar p_i pre-incidente a la transición t_a, esto es: $M(p_i) = M(p_i) - Pre(i,a)$, si $Pre(i,a) \geq 1$.

Marcar lugares post-incidentes

Consiste en aumentar el marcado M de todo lugar p_i post-incidente a la trasición t_a, esto es: $M(p_i) = M(p_i) + Post(i,a)$,si $Post(i,a) \geq 1$.

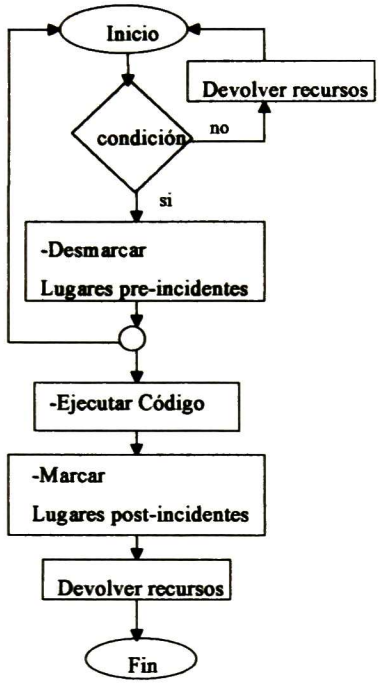


Figura 4.6: Diagrama de flujo del comportamiento de una transición

Código

Actividad que ejecuta la transición al activarse. Esta actividad puede no existir.

Devolver recursos

Libera a los lugares pre y post-incidentes que haya usado en sus partes anteriores.

Código java del comportamiento de una transición.

A continuación se ilustra el código fuente del comportamiento de una transición ilustrado en la figura 4.6. Este código permanece constante en cada tarea.

```
public abstract class transicion extends Thread
int tiempo;
transicion(int r){r = tiempo;}
public void retardo(){
try {sleep(int Math.random()*r)}
catch(InterruptedException e){
```

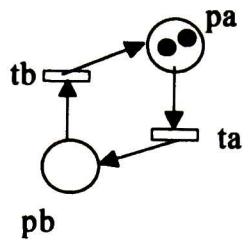


Figura 4.7: Reentrancia de disparo de transiciones

```
 System.err.println(e.toString());
}
public void run(){
 while(true) {
  if(condición()){
     Desmarcar();
```

/*accion: clase tipo Thread que ejecuta código de la transición t, luego marca lugares post-incidentes a t y por último devuelve recursos:*/

```
     accion code_accion = new accion(this);
     Thread a = new Thread(code accion);
     a.start();
  }
  else
    devolver_recursos();
  retardo();

}
```

/*Los siguientes métodos forman parte del comportamiento de una transición pero son implementados por cada transición en particular*/

```
abstract boolean condicion()
abstract void marcar();
abstract void desmarcar();
abstract void devolver_recursos();
abstract void código();


class accion extends Thread(){
```

```
transicion tr;
accion(transicion tran){tr = tran }
public void run(){
tr.codigo();
tr.marcar();
}
}
```

Parte II- Código de transición

Toda transicion, tiene un comportamiento general (código de clase transición) pero varían de acuerdo a su renglón de pre y post-incidencia. El código de cada transición tr_a sólo implementa los métodos vacíos (abstractos) de la clase transición. Esto es:

```
class tra{
...
public int marcar(){
//código
}

public int desmarcar(){
//código
}
...
```

4.4.2 Sección de monitor

Sincroniza las tareas de la aplicación distribuida, a continuación veremos un bosquejo del código Java de este, si desea ver el código completo refiérase al apéndice B.

Nomenclatura:

i = lugar

fuente = identifica a la de la tarea fuente o local

destino = identifica a la tarea destino o remota

Código:

```
class Shared{
...
public synchronized getMarcado(int i){
  // código que devuelve el marcado del lugar i perteneciente a la tarea local
}
```

```
public int getMarcado(int lugar, int fuente, int destino){
//código en el que una tarea fuente solicita el marcado de un lugar
que se encuentra en una tarea destino, usa paso de mensajes.
}

public int setMarcado(int i   int ofset){
/*código que cambia el marcado de un lugar i en una tarea local sumandole el número ofset el cual es positivo para marcar y negativo para desmarcar.*/
//Bloquea lugar i.
//Usa paso de mensajes.
}

public int setMarcado(int i, int ofset, int fuente, int destino){
/*código en la que una tarea fuente solicita cambiar el marcado de un lugar i que se encuentra en una tarea destino sumandole el número ofset donde ofset es positivo para marcar y negativo para desmarcar.*/
// Bloque al lugar i.
//usa paso de mensajes.
}
public synchronized int devolver_recursos(int i)
//pone el lugar i disponible para lectura/escritura
}
public synchronized int devolver_recursos(int i, int fuente, int destino){
//código en la que una tarea fuente solicita liberar un lugar i que se encuentra en
//una tarea destino.
}
}
```

4.4.3 Sección de Comunicaciones

La sección de comunicaciones está formada por las siguientes partes:

Cliente: Envía mensajes de una tarea a otra, en cada tarea existen tantos clientes como conexiones hay a los servidores de otras tareas.

Servidor: El servidor de una tarea es un Thread que recibe las peticiones de otras tarea para devolver el marcado de algún lugar en su tarea, cambiar éste o bien liberar recursos; esto es trabaja de modo independiente a su tarea. Se apoya de los métodos de su monitor(clase Shared). Maneja un hilo(thread) de atención por cliente.

Un acercamiento al código java de este es:

```
class servidor extends Thread{
 Shared Monitor;
servidor(Shared m){
 Monitor = m;
}
public void run()....
while true(){
  if (hay una coneccion)
        //crear Thread de atención  al cliente (id_cliente) y ejecutarlo.
}
public void analizar_cadena(String cadena, int destino){
  .//proceso extraer orden de cadena
  if (orden == "getMarcado"){
   int resultado = Monitor.getMarcado(lugar)
   enviar mensaje(resultado, tarea)
 }
 if (orden == "setMarcado"){
  int acuse = Monitor.setMarcado(lugar);
  enviar_mensaje(acuse);
 }
 if(orden == "devolver_recursos"){
  int acuse = Monitor.devolver_recursos(lugar);
 }
}
}


class atencion_cliente extends Thread
atencion(servidor s, int id_cliente){
...
}
public  void run(){
while(true){
 leer_mensaje()
 s.analizar_cadena(cadena)
```

}
}

El código cliente/servidor permanece constante en cada tarea y puede ser sustituido por cualquier otro de comunicaciones añadiéndole la parte que analiza y ejecuta las peticiones.

4.5 Algoritmo para generar código Java

1. Obtener los p-invariantes de la red (capítulo 1)

2.- Particionar la red en tareas (capítulo 2), las columnas de Λ_t representan las tareas, un 1 en la columna j significa que una transición t_y es asignada a la tarea j, es igual a 0 de otra manera.

3.- Distribuir los lugares de la RP entre las tareas obtenidas en 2 (capítulo 3) las columnas de Λ_l representan las tareas, un 1 en la columna j significa que un lugar l_z es asignada a la tarea j, es igual a 0 de otra manera.

4.- Para cada columna j de Λ_t generar una tarea TAREA $_j$ en un directorio de trabajo Tareaj como sigue:

4.1.- Generar el archivo de datos de la tarea, datos.java de la siguiente manera:

class datos{

para cada una de las n tareas:

i) especificar el nombre de los ordenadores (servidores) donde se ejecutarán.

ii) especificar el número de tareas.

iii) especifar el número de lugares.

iv) especificar el número de transiciones.

v) inicializar marcado inicial de cada uno de los lugares.

vi) finalizar clase datos.

Esto es:

i) *String[] cpu_name = ("-1", "nombre 1", "nombre 2"..."nombre n");*

ii) *int ntask = número de tareas;*

iii) *int nplaces = número de lugares;*

iv) *int ntran = número de transiciones;*

v) *int[] lugar =*{-1,m_1,m_2,m_3,...m_{ntask}}-

vi) }

4.2 Generar archivo de código principal.java como sigue:

4.2.1- Escribir lo siguiente:

class principal{

public static void main(String args[]){

int puerto = núm.detarea × 1000;
conexion cx = new conexion(); //crea clientes
Shared sh = new Shared(cx); //crea el monitor de la tarea
servidor s = new servidor(sh,puerto); //crea servidor
s.start(); //pone el servidor en ejecución
transicion t; //declara comportamiento de transición
//establece retardo:
try{Thread.sleep(10000);}
catch(InterruptedException e){
System.err.println("Exception" + e.toString());}

4.2.2.- crear cada transición tr_y perteneciente a la tarea j ($\Lambda_t[y][j] = 1$) como sigue:
tr_y u_y = *new* tr_y*(sh);*

4.2.3– Poner en ejecución a las transiciones creadas en 4.2.2 del modo siguiente:
$t = u_y$;
t.start();

4.2.4.- finalizar escribiendo lo siguiente:
}
}

4.3.- Para cada transición tr_y perteneciente a la tarea j ($\Lambda_t[y][j] = 1$) generar archivo de código tr_y.java del modo siguiente:

4.3.1.- Escribir lo siguiente:
class tr_y *extends transicion{*
private Shared s; //usa objeto Shared(Monitor)
tr_y*(Shared s){*
super(100);
this.s = s;
}

El código que resta hace diferencia entre lugares pertenecientes a la tarea j (a) y los pertenecientes a otra (b).

4.3.2- Escribir Método Marcar de la transición tr_y como sigue:

Escribir encabezado
public void Marcar(){
Escribir el cuerpo del método Marcar del modo siguiente:

a) Para todo lugar m tal que $Pre[m][y] \geq 1$ y $(\Lambda_l[m][j] = 1)$ escribir:

s.setMarcado(m, Pre[m][y], y);

b) Para todo lugar m tal que $Pre[m][y] \geq 1$ y $\Lambda_l[m][k] = 1$ donde $k \neq j$ escribir:

s.setMarcado(m,Pre[m][y],y,j,k);

-Finalizar método

}

4.3.3- Escribir Método Desmarcar de la transición tr_y de la siguiente manera:

-Escribir Encabezado

public void Desmarcar(){

-. Escribir el cuerpo del método. Esto es:

a) Para todo lugar m tal que $Post[m][y] \geq 1$ y $(\Lambda_l[m][j] = 1)$ escribir:

s.setMarcado(m, -Post[m][y], y);

b) Para todo lugar m tal que $Post[m][y] \geq 1$ y $\Lambda_l[m][k] = 1$ donde $k \neq j$ escribir:

s.setMarcado(m,Post[m][y],y,j,k);

-Finalizar método

}

4.3.4- Escribir Método Devolver_recursos de la transición y de la siguiente manera:

-Escribir encabezado

public void Devolver_recursos(){

-Escribir el cuerpo del método. Esto es:

Para todo lugar m tal que $\Pr e[m][y] \geq 1$ y $(\Lambda_l[m][j] = 1)$ escribir:

s.true_read(m, -Post[m][y], y);

Para todo lugar m tal que $\Pr e[m][y] \geq 1$ y $\Lambda_l[m][k] = 1$ donde $k \neq j$ escribir:

s.true_read(m,Post[m][y],y,j,k);

Finalizar método:

}

4.3.5.- Escribir el método de la condición de la transición tr_y de la siguiente forma:

-poner encabezado e iniciar condicional

public boolean condicion(){

if(

-añadir condición de la siguiente manera: (cada adición se concatena con &)

a) Añadir a la condición todo lugar m tal que $Pre[m][y] \geq 1$ y $(\Lambda_l[m][j] = 1)$ de la siguiente manera:

(s.getMarcado(m,y)>=Pre[m][y])

b) Añadir a la condición todo lugar m tal que $Pre[m][y] \geq= 1$ y $(\Lambda_l[m][k] = 1$ y $j \neq k)$

de la siguiente manera:

(s.getMarcado(m,y,j,k)>=Pre[m][y])

Cerrar condición:

)

-Escribir:

return true;

else

return false

}

4.3.6.- Escribir método de código de la transición como sigue:

public void codigo(){

//aquí va el código de la transición

}

4.4- Copiar al directorio Tareaj los archivos constantes: transición.java(comportamiento de toda transición), cliente.java, servidor.java(manejo de comunicaciones entre tareas), Shared.java (Monitor de tareas).

Ejemplo El siguiente ejemplo muestra el código Java obtenido por el algoritmo 1 aplicado a la RP de la figura 4.8

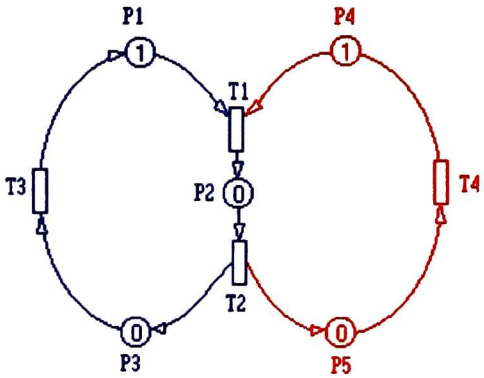


Figura 4.8:

En el directorio de trabajo...\tarea1 se encuentran los siguientes archivos:

1.- Archivo: datos.java

```
class datos{
```

```
String[] cpu_name = {"nada"  "nombre 1" "nombre2"};
int ntask = 2;
int nplaces = 5;
int ntran = 4;
int[] lugar = {-1,1,0,0,1,0};
}
```

//sustituir "nombre i" por los nombres de los ordenadores donde se ejecutarán las tareas o bien por sus direcciones IP de red eliminando las comillas.

2.-Archivo principal.java:

```
class principal{
public static void main(String args[]){
int puerto = 1000;
conexion cx = new conexion();
Shared sh = new Shared(cx);
servidor s = new servidor(sh,puerto);
s.start();
transicion t;
try{Thread.sleep(10000);}
catch(InterruptedException e){
System.err.println("Exception" + e.toString());}
//———DEFINICION y arranque DE TRANSICIONES————————
tr1 u1= new tr1(sh);
tr2 u2= new tr2(sh);
tr3 u3= new tr3(sh);
t = u1;
t.start();
t = u2;
t.start();
t = u3;
t.start();
}
}
```

3.-Archivos de cada transición perteneciente a la tarea 1

3.1- Archivo tr1.java

```
class tr1 extends transicion{
private Shared s;
//-----------------------------------
super(100);
tr1(Shared s){
this.s = s;
}
//-----------------------------------
public void Marcar(){
s.setMarcado(2,1,1);
}
//-----------------------------------
public void Desmarcar(){
s.setMarcado(1,-1,1);
s.setMarcado(4,-1,1);
}
//-----------------------------------
public void Devolver_recursos(){
s.true_read(1,1);
s.true_read(4,1);
}
//-----------------------------------
public boolean condicion(){
if((s.getMarcado(1,1)>=1)&&
(s.getMarcado(4,1)>=1))
return true;
else
return false;
}
//-------Codigo de la transicion---
public void codigo(){
 //aqui va el código de la transición
}
}
```

3.2-Código tr2.java

```
class tr2 extends transicion{
private Shared s;
//-------------------------------------
tr2(Shared s){
super(150);
this.s = s;
}
//-------------------------------------
public void Marcar(){
s.setMarcado(3,1,2);
s.setMarcado(5,1,2,1,2);
}
//-------------------------------------
public void Desmarcar(){
s.setMarcado(2,-1,2);
}
//-------------------------------------
public void Devolver_recursos(){
s.true_read(2,2);
}
//-------------------------------------
public boolean condicion(){
if((s.getMarcado(2,2)>=1))
return true;
else
return false;
}
//-------Codigo de la transicion---
public void codigo(){
//aquí va código de la transición
}
}
```

3.3 Código tr3.java

```
class tr3 extends transicion{
```

```
private Shared s;
//-------------------------------------
tr3(Shared s){
super(200);
this.s = s;
}
//-------------------------------------
public void Marcar(){
s.setMarcado(1,1,3);
}
//-------------------------------------
public void Desmarcar(){
s.setMarcado(3,-1,3);
}
//-------------------------------------
public void Devolver_recursos(){
s.true_read(3,3);
}
//-------------------------------------
public boolean condicion(){
if((s.getMarcado(3,3)>=1))
return true;
else
return false;
}
//-------------------------------------
public void codigo(){
//aquí va código de la transición
}
}
```

En el directorio de trabajo...\tarea2 se encuentran los siguientes archivos:

1.- Archivo: datos.java (mismo código en todas las tareas)

2.-Archivo principal.java

```
class principal{
 public static void main(String args[]){
```

```
int puerto = 2000;
conexion cx = new conexion();
Shared sh = new Shared(cx);
servidor s = new servidor(sh,puerto);
s.start();
transicion t;
try{Thread.sleep(10000);}
catch(InterruptedException e){
System.err.println("Exception" + e.toString());}
//-------DEFINICION y arranque DE TRANSICIONES----------------
tr4 u4= new tr4(sh);
t = u4;
t.start();
}
}
```

3.-Archivos de cada transición perteneciente a la tarea 2

3.1 Archivo tr4.java

```
class tr4 extends transicion{
private Shared s;
//-----------------------------------
tr4(Shared s){
super(250);
this.s = s;
}
//-----------------------------------
public void Marcar(){
s.setMarcado(4,1,4,2,1);
}
//-----------------------------------
public void Desmarcar(){
s.setMarcado(5,-1,4);
}
//-----------------------------------
public void Devolver_recursos(){
s.true_read(5,4);
```

```
}
//-----------------------------------
public boolean condicion(){
if((s.getMarcado(5,4)>=1))
return true;
else
return false;
}
//-----------------------------------
public void codigo(){
//aquí va código de la transición
}
}
```

4.- Copia de los siguientes archivos: transicion.java, Shared.java, cliente.java, servidor.java

4.6 Ambiente visual de especificación y programación

El objetivo tangible de este trabajo de investigación es la generación automática de código distribuido Java de una especificación de RP, para esto se retomó el ambiente visual de especificación y programación, PN-Spec de P.Gutiérrez ([16]) el cual genera código ADA-95 a partir de una especificación de RP; a este trabajo se le añadió las implementaciones necesarias para cubrir nuestro objetivo obteniendo PN-Spec2.

En éste apartado ilustraremos el ambiente visual de programación PN-Spec2.

4.6.1 Características de PN-Spec2

El proyecto PN-Spec2 fue desarrollado por completo en lenguaje C++, mediante el compilador Borland Builder C++ 3.0[23],[24].

Los módulos principales con los que cuenta PN-Spec2 son los siguientes:

- **Edición**: maneja el ambiente gráfico de la herramienta, también cuenta con las operaciones de manejo de archivos e impresión.
- **Análisis**: realiza el cálculo de invariantes de la red, cálculo de las matrices de pre, post y de incidencia útiles para implementar el analizador de propiedades.
- **Generación de código ADA-95**: contiene los algoritmos necesarios para generar código multitarea en ADA-95. Este código es centralizado, no orientado a objetos y las tareas se conmutan por tasas de tiempo (paralelismo aparente)

- **Generación de código Java:** contiene los algoritmos necesarios para generar código en Java. Este código es distribuido, orientado a objetos y permite el paralelismo, de tal modo que en una misma tarea, pueden estar ejecutándose distintas partes de ella, conmutándose por tiempos de procesador; así también es posible que dos o más tareas puedan ejecutarse simultáneamente en distintos procesadores.

Los tres primeros módulos fueron tomados de PN-Spec y se les hicieron las agregaciones necesarias para obtener el módulo 4.

Otros módulos menores son los encargados de actualizar las propiedades de la red, esto es, mantienen la información referente al marcado, etiquetas, colores, código Java asociado a las transiciones, etc.

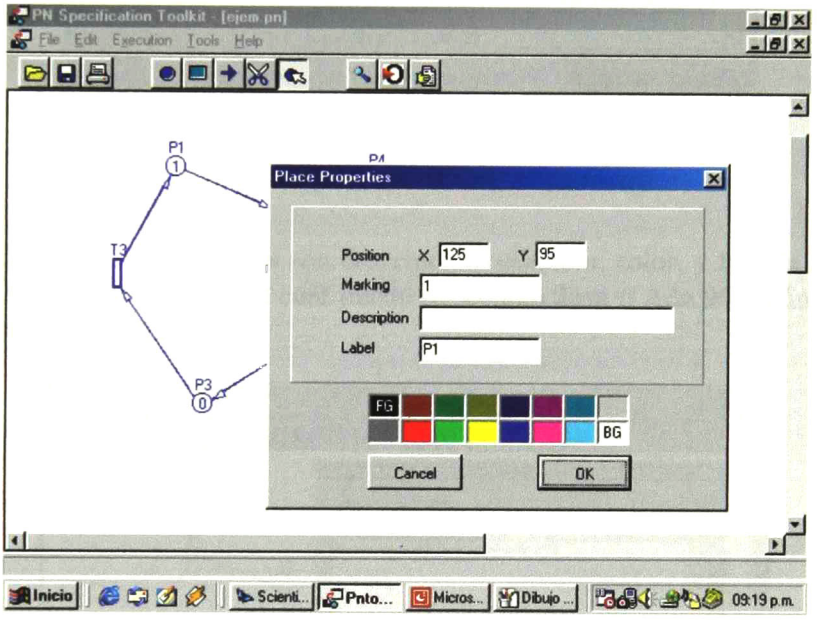


Figura 4.9: Propiedades de los lugares

Módulo de edición

PN-Spec2 de modo heredado de PN-Spec, permite la edición de RP. Este módulo cuenta con las opciones necesarias para dibujar lugares, transiciones y arcos. Para dibujarlos sólo hay que seleccionar un botón de la barra de herramientas (el que tiene la imagen de un círculo, una barra o un arco) y enseguida pulsar el botón izquierdo del ratón en la ventana de edición.

Una vez editada la red, se permite modificar las propiedades a los lugares; por ejemplo el marcado, asignar una descripción al lugar, cambiar su etiqueta, etc. (ver figura 4.9). Esto se hace pulsando el botón derecho del ratón sobre el lugar que se desea actualizar.

Las transiciones también poseen propiedades, las cuales pueden ser alteradas mediante su ventana de edición. Para accesar a esta ventana de propiedades, se procede igual que con los

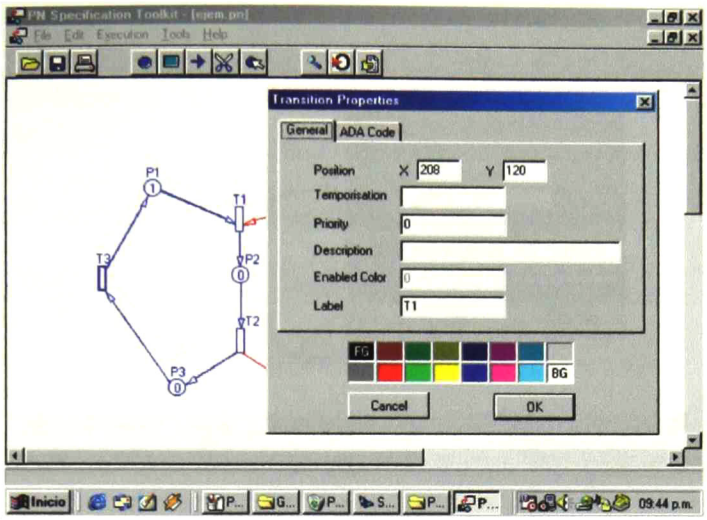


Figura 4.10: Propiedades de las transiciones

lugares. Algunas de estas propiedades son descripción, etiqueta, color, y la más importante, el código asociado a esa transición, el cual puede ser código Java o Ada 95 según el lenguaje requerido (figuras 4.10 y 4.11).

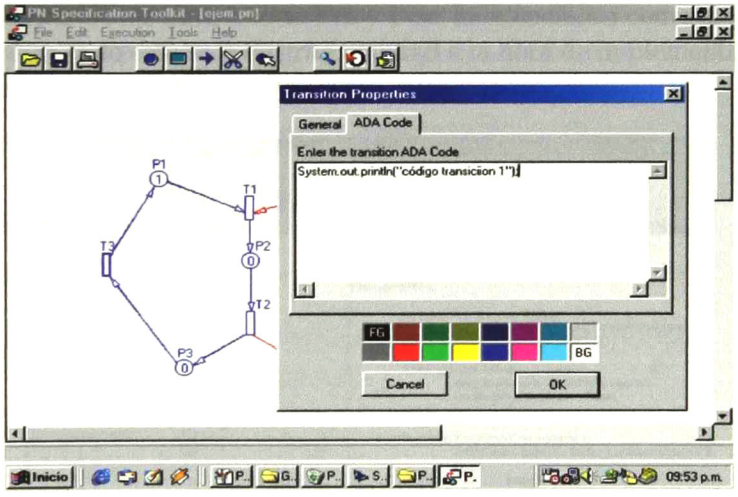


Figura 4.11: Código Java asociado a una transición

Los arcos, al igual que los lugares y transiciones, también pueden ser modificados mediante su ventana de propiedades (figura 4.12).

Además de lo anterior, la herramienta también permite la eliminación de lugares, transiciones y arcos, y cambiar de posición los lugares y transiciones.

También se cuenta con las operaciones básicas de cualquier editor como manejo de archivos e impresión. Estas opciones se encuentran en el menú "File".

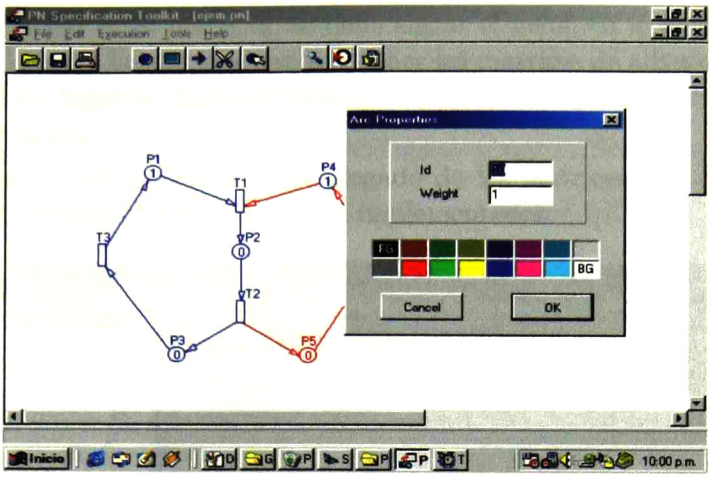


Figura 4.12: Propiedades de los arcos

Módulo de análisis

El software desarrollado por PN-Spec, cuenta con la implementación de los algoritmos de cálculo de los invariantes de lugares y de transiciones, así también desarrolla el algoritmo para obtención de la matriz de adyacencia (C) de la especificación dada en RP. PN-Spec2 añadió a este módulo la obtención de las matrices de Pre y Post incidencia,así como el reconocimiento de lugares compartidos; todo esto será de gran utilidad a la hora de implementar el analizador automático de propiedades, el cual no está aún implementado.

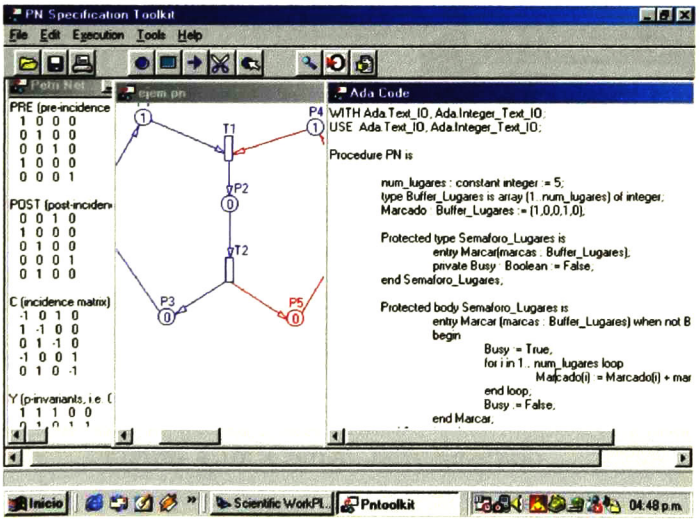


Figura 4.13: Generación de código Ada

El análisis que la herramienta puede desarrollar consiste en calcular la matriz C, calcular los p-invariantes y los t-invariantes, calcular las matrices de Pre y Post incidencia.

Para realizar desplegar éstas matrices, se debe entrar el menú en la opción "Execution" y ahí se encuentran listadas las opciones para obtener las matrices de adyacencia, incidencia y de los invariantes de lugares. Esta información es también mostrada cuando se genera código de la especificación.

A éste módulo se le podría añadir el desplegado de las matrices de t-invariantes y de recursos compartidos, usando los algoritmos ya implementados.

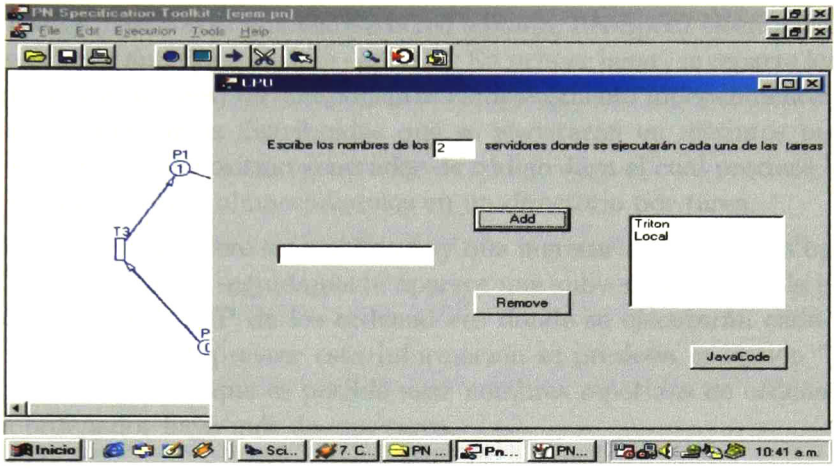


Figura 4.14: Generación de código Java

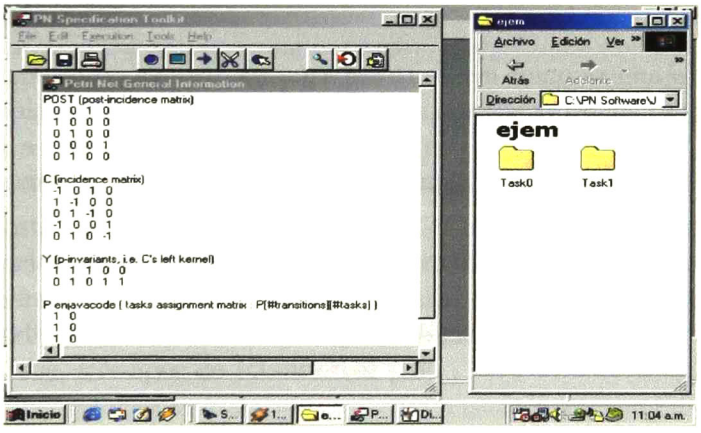


Figura 4.15: Código Java generado

Módulo de generación de código ADA-95

PN-Spec2 toma de PN-Spec el módulo de generación de código ADA 95. Para generar el código ADA correspondiente a la red editada, PN-Spec ejecuta algoritmos para repartir la RP en tareas (transiciones) las cuales serán ejecutadas concurrentemente en un solo procesador,

finalmente, se ejecuta el algoritmo generador de códigoADA-95. Para que la herramienta genere este código hay que ingresar al menú en la opción "Execution" y ahí "Get ADACode". El archivo de código generado es guardado en un directorio de disco y también desplegado en pantalla.(Figura 4.13)

Módulo de generación de código Java

Para generar el código Java correspondiente a la red editada, PN-Spec2 ejecuta los algoritmos presentados en secciones anteriores de este capítulo. En primer lugar, se reparte los elementos de la RP (lugares y transiciones) en componentes completamente independientes, los cuales serán considerados como tareas distribuidas que se ejecutarán en distintos procesadores; posteriormente se ejecuta el algoritmo generador de código Java el cual produce los archivos de código Java para cada tarea almacenándolos en un directorio por tarea.

Para que la herramienta genere este código hay que ingresar al menú en la opción "Execution" y ahí "GetJava Code", seguidamente aparece una subventana donde la herramienta solicita el nombre o dirección IP de los ordenadores donde se ejecutarán cada una de las tareas, finalmente, después de proveer esta información se presiona la opción "JavaCode" (figura 4.14). Cabe mencionar que es posible usar nombres repetidos de ordenadores, si se desea que en un ordenador haya más de una tarea.

La figura 4.15 nos muestra los directorios de archivos de código Java generados para la RP de la figura 4.8, así como la información general de ésta generada por la herramienta.

4.7 Conclusiones

En este capítulo se expusieron las razones para utilizar Java como lenguaje de programación para sistemas distribuidos. También se dieron las grandes líneas de los procedimientos para generar código Java a partir de una especificación modularizada en RP. En el esquema propuesto el código generado es orientado a objetos, multitarea y distribuido.

Por último se mostró la utilización de PN-Spec2,el cual es un ambiente gráfico de especificación y generación de programas distribuidos en Java y multitarea, centralizado en ADA-95. Para la generación de código en Java se usa el algoritmo presentado en este capítulo y la generación de código ADA-95 es tomada del ambiente de especificicación y generación de programas multitarea, PN-Spec de P.Gutierrez [16].

Conclusiones

El trabajo desarrollado en esta tesis se ha centrado en el estudio de dos aspectos ligados al diseño de sistemas concurrentes: la especificación de los sistemas y la generación de programas distribuidos a partir de éstos.

La especificación de sistemas debe garantizar la preservación de ciertas propiedades, la vivacidad y el acotamiento fueron tratados en este trabajo de investigación. Para que estas propiedades sean garantizadas se requiere de una técnica de especificación que permita construir modelos que las preserven.

La complejidad de los sistemas reales hace deseable disponer de metodologías que simplifiquen el trabajo del modelador. Es en el capítulo 1 donde se propone una técnica de construcción de modelos (que permite especificar sistemas que se sincronizan, comparten recursos, o se comunican a través de paso de mensajes) donde mediante la imposición de restricciones estructurales a la arquitectura, se puede asegurar que el modelo subyacente sea vivo y acotado.

El trabajo propuesto en el capítulo 2 trata el problema de realizar una repartición del sistema, una asignación de tareas a procesadores. La primera fase tomada de P. Gutierrez [16] consiste en repartir las transiciones de la RP especificada y asignar las partes a un conjunto de procesadores. Una segunda fase consiste en repartir los lugares de la RP especificada y asignarlos a un conjunto de procesadores. Con esta repartición obtenemos modelos concurrentes centralizados. Cabe mencionar que la asignación no siempre es óptima, puesto que no toma en cuenta la carga, velocidad y localización de los procesadores, por lo que deben buscarse heurísticas que mejoren este algoritmo, tomando en cuenta dichos parámetros; esta repartición también se podría mejorar, permitiendo al usuario repartir la especificación según sus necesidades.

Para la especificación de sistemas distribuidos se requiere una transformación de los sistemas concurrentes centralizados que garanticen la correctud y buen funcionamiento del modelo. En el capítulo 3 se propone una técnica de transformación de modelos concurrentes centralizados a modelos concurrentes distribuidos, mediante el uso de paso de mensajes.En ésta técnica cabe destacar que la transformación de los esquemas de sincronización no es la más óptima cuando la transición de sincronización no contiene código asociado, puesto que la tarea no "propietaria" de dicha transición, no requiere esperar que se ejecute un código asociado a tal transición. Como trabajo futuro para mejorar esta deficiencia, se propone tratar de modo diferente a este tipo de sincronización, usando para su transformación el esquema de mensajes cruzados, el cual intercambia mensajes en forma simultánea; sin realizar esperas, es decir en una unidad de tiempo.

Posteriormente en el capítulo 4 se presentaron algoritmos para generar código distribuido dada una especificación centralizada de RP. Dichos programas fueron generados en lenguaje Java. Una desventaja es que el programa generado maneja el marcado de la red sólo para control de evolución del sistema no como datos, para esto se requiere manejar las marcas como registros u objetos lo cual se considera como trabajo futuro.

Por último, se retomó la herramienta visual que permite la edición de modelos y generación de código concurrente no distribuido en ADA95 añadiéndole la implementación de los algoritmos de generación de código mencionados anteriormente de manera que el código Java distribuido correspondiente a una RP puede ser generado automáticamente. El analizador automático de propiedades de RP no fue implementado, por lo que puede ser considerado como trabajo futuro.

Bibliografía

[1] Andrew Tanenbaum "Distributed Operating Systems". Prentice Hall. 1994

[2] Ida Flynn and Ann Mclever "Sistemas Operativos" 3era. ed.Thomson Learning 2001.

[3] Milan Milenkovic "Sistemas Operativos conceptos y diseño" 2da. ed. Mc Graw Hill

[4] H. M. Deitel "Introducción a los Sistemas Operativos" 2da. ed Pearson.1993

[5] Denni Torres "Apuntes de Sistemas operativos" CINVESTAV del IPN 1998

[6] Calingaert P. "Operating system elements: An user perspective" Prentice Hall 1982

[7] "7066 Concurrent Programming" O. Nierstrasz 1998 Universidad de Bern Portugal.

[8] Miguel.García Hofman "Interconexión de periféricos a microprocesadores: Protocolos de Comunicación y Sincronización" revista "Mundo Electrónico" pp 29-30

[9] Andrew Tchernykh "Parallel Computing" CICESE 1998

[10] F.DiCesare, G.Harhalakis, J.M. Proth, M.Silva, F:B Vernadat. "Practice of PetriNets in Manufacturing" Chapman &Hall. 1993

[11] M.Silva. "Las Redes de Petri: en la Automática y la Informática". Editorial AC. Madrid, España. 1985

[12] Mu Der Jeng and Frank DiCesare. "A review of Synthesis Techniques for Petri Nets with Applications to Automated Manufacturing Systems" IEEE Transactions on Systems, Man, and Cybernetics. Pag. 301-312. Vol. 23. No. 1. January/February 1993

[13] J.Desel, J.Esparza. "Free Choice Petri Nets" Cambridge University Press.

[14] Inseon Koh, Frank DiCesare. "Modular transformation methods for generalizated Petri Nets and their application to automated manufacturing systems".IEEE Transactions on System, Man and Cybernetics. Pag. 1512-1522. Nov/Dic 1991

[15] P.Gutiérrez Robles, E. López Mellado, A. Ramírez Treviño. "Modular Task Modelling of Automated Flexible Manufacturing Systems" Proceeding of the International Symposium on Robotics and Automation 1998, pp. 379-386.

[16] P. Gutiérrez Robles Tesis de Maestría en Ciencias de la Computación: "Automatización de Sistemas Concurrentes" junio 1998

[17] G. Agha, F. Frolund, W. Y. Kim, R. Panwar and D. Sturman. "Abstracction and Modularity Mechanism for Concurrent Computing" Research Directions in Concurrent Object-Oriented Programming, G.Agha, P. Wegner and A. Yonezawa(eds.), pp. 3-21, MIT Press 1993.

[18] Ernesto López M. "Introducción a las Redes de Petri " FCFM-UANL octubre 1997

[19] Glasser Edward, Beta Phi "Dictionary of Computing" Oxford University Press.

[20] J. Colom. "Análisis estructural de Redes dePetri, Programación Lineal y Geometría Convexa" Tesis Doctoral. Universidad de Zaragoza. España. 1989.

[21] Froufe Quintas Agustin "Java 2"edit. AlfaOmega-Ra-Ma 2da. edición

[22] Deitel & Deitel "Java How To Program" edit. Prentice Hall 2da. ed.

[23] Francisco Charte Ojeda. "Programación con C++ Builder". Ediciones Anaya Multimedia, S. A., 1997.

[24] Ken Reisdorph "Teach YourSelf Borland C++Builder3 in 21 days" SAMS Publishing, 1998

[25] Richard Scott Brink "A Petri Nets Design Simulation and Verification Tool" Tesis de maestría. Instrituto Tecnológico de Rochester New York.

Apéndice I - Introducción a los sistemas concurrentes

Resumen En este apéndice se muestran las características de los sistemas concurrentes paralelos y pseudoparalelos, sus ventajas y desventajas así como los mecanismos de sincronización y comunicación.

A Introducción

La concurrencia es esencialmente la ejecución simultánea de varias actividades a la vez llamadas tareas o procesos. Los sistemas concurrentes hoy en día han estado ganando terreno en el mundo de la informática debido a sus grandes beneficios. Los beneficios de la ejecución concurrente incluyen el aumento de rendimiento, de la utilización de recursos y de la velocidad de respuesta y mayor apego al mundo real en un sistema informático, sin embargo el manejo de concurrencia hace necesarios añadir mecanismos de sincronización y comunicación para control de las tareas, haciendo el manejo de los sistemas concurrentes más complejo que los sistemas secuenciales.

Los sistemas concurrentes pueden ser pseudoparalelos o paralelos, los primeros también llamados multitarea multiplexan la ejecución de las tareas en un solo procesador, los segundos también llamados multiprocesador manejan varias unidades de procesamiento pudiendo de ese modo ejecutar varias tareas en forma simultánea o paralela, cuando estos sistemas ejecutan sus tareas en unidades independientes e interconectadas entre si son llamados distribuidos.

B Conceptos Básicos

En esta sección se verán algunos conceptos básicos de concurencia.

Definition 1 *Proceso o tarea: Son términos sinónimos que representan una secuencia de código en ejecución*

Definition 2 *.En esencia un proceso o tarea es una instancia de un programa en ejecución*

Definition 3 *Un sistema o programa centralizado es aquel que se ejecuta en su totalidad en forma local.*

Definition 4 *Un sistema o programa distribuido es aquel que se ejecuta en distintas localidades de procesamiento independientes interconectadas entre si.*

En un ambiente distribuido cada tarea puede procesar datos locales libremente y tomar decisiones locales.

Los procesos intercambian información por medio de una red de comunicación de datos, procesando información o leyendo decisiones que afectan a otras tareas.

Definition 5 *Un sistema o programa secuencial especifica la ejecución secuencial de tareas.*

Definition 6 *Un sistema o programa .secuencial es aquel en el que sus instrucciones se ejecutan una después de otra.*

En un programa secuencial un tarea solo puede ejecutarse hasta que el tarea antecesora haya terminado su ejecución.

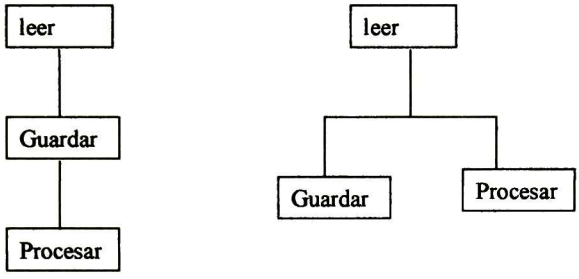
Definition 7 *Un sistema o programa concurrente especifica dos o mas programas que pueden ser ejecutados concurrentemente es decir de modo simultáneo. [7]*

Definition 8 *Un programa concurrente está compuesto de dos o más programas secuenciales que se ejecutan simultáneamente.*

Para realizar un sistema concurrente es necesario hacer una división explícita de las tareas para poderlos ejecutar concurrentemente, sin embargo por las interaciones entre estas se presentan "problemas" que no se plantean en la ejecución de cada una de ellas por separado. Algunos de estos problemas son:bloqueos, exclusión mutua y comunicación entre procesos.

Ejemplo de Sistema Secuencial y Sistema Concurrente.

Supongamos que se tiene un sistema que lee datos de entrada por medio de un sensor, los guarda en disco, los procesa y los imprime. En un sistema secuencial estas operaciones se harían secuencialmente, es decir una después de otra, en un sistema concurrente, después de leer los datos, procedería a guardar y a procesar los datos simultáneamente. Ver figura B



a) Sistema secuencial

b) Sistema Concurrente

Figura 1: Sistemas secuenciales y concurrentes

C Grados de Concurrencia

C.1 Sistemas concurrentes

Por sus características los sistemas concurrentes pueden ser:

- Multitarea-Se Ejecutan varias tareas a la vez.
- Multiproceso- Multitarea con múltiples unidades de procesamiento(CPU)
- Distribuido-Multiproceso con computadores independientes distribuidos interconectados entre si.(más de un computador)

Nota 9 *Note que un sistema distribuido es multiproceso y este es multitarea pero lo inverso no es simpre cierto*

C.2 Sistemas multitarea

Cuando en un sistema se ejecutan varias tareas a la vez se dice que es multitarea. Los sistemas multitarea con un solo procesador (CPU) son la forma más simple de concurrencia su procesamiento es llamado **multiprogramación** en estos sistemas el procesador ejecuta las tareas entrelazadas en el tiempo, lo que se

denomina **tiempo compartido.** En los sistemas multitarea con múltiples procesadores(multiproceso), los sistemas están no solo entrelazados en el tiempo sino tambien traslapados;[5] ambas situaciones se ilustran en la fig.2. Como podemos ver en la figura existe un paralelismo aparente en los sistemas multitarea con un solo procesador, dado que las tareas activas se multiplexan en el tiempo pareciendo que están ejecutándose al mismo tiempo.

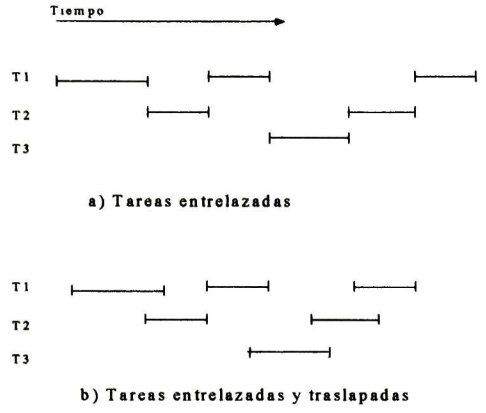


Figura 2: Entrelazado y Traslape

Entre las razones comunes para aplicar sistemas multitarea se encuentran las siguientes:[3]

- *Ganancia de Velocidad-* La división explícita de tareas puede dar lugar a una ejecución más rápida de una aplicación.
- *Manejo de Latencia-*En una aplicación cuando una tarea tiene que esperar .la disponibilidad de un recurso de E/S o el resultado de otra tarea el resto de la aplicación puede progresar si contiene otras tareas que pueden hacer trabajo útil mientras tanto.
- *Conveniencia del usuario:* El usuario puede dividir una aplicación en tareas para manejar acciones individuales, por ejemplo: Un interfaz gráfica puede permitir a los usuarios desencadenar varias acciones concurrentes mediante la pulsación sobre íconos de acción antes de que acaben de procesarse órdenes previas.
- *Multiprocesamiento:*Un programa codificado como colección de tareas puede ser transportado con relativa facilidad a un multiprocesador, en donde las tareas individuales pueden ser ejecutadas sobre diferentes procesadores.

C.2.1 Desventajas

En general los sistemas concurrentes son más difíciles de manejar en cuanto a que la manera como se comunican y sincronizan los procesos no deja de ser crucial, pues se necesitan mecanismos que aseguren la consistencia y garanticen el progreso de esto. La secuencia del tiempo en la ejecución puede variar según las condiciones del sistema haciendo más difícil su depuración en comparativa con los sistemas secuenciales.

C.3 Sistemas Multiproceso

El procesamiento de los sistemas multiproceso se llama **multiprocesamiento** también llamado **procesamiento paralelo** el cual es una situación en la cual dos o más procesadores(CPU) funcionan al unísono. Esto quiere decir que dos o más CPU ejecutan instruccionesde manera simultánea. Por lo tanto con más de un proceso activo a la vez , cada CPU puede tener al mismo tiempo una tarea en estado de *ejecución.*[2] Para las actividades de multiprocesamiento, se debe sincronizar la interacción entre los CPU.

C.3.1 Ventajas

Dos de las principales beneficios que los sistemas multiproceso añaden a los sistemas concurrentes son[2]:

- Incrementa la potencia de cómputo
- Incremento de rendimiento

El incremento de la potencia de cómputo se logra en cuanto que en el sistema se explota mejor sus recursos al haber un procesamiento paralelo, por ejemplo supongamos dos tareas en el sistema una tarea con un CPU requiere usar una impresora y otra tarea con otro requiere usar un escaner ambas pueden ejecutarse usando el recurso que requieren sin interrupciones de tiempo por compartición de CPU incrementado la potencia del sistema.

El *incremento de rendimiento* se logra porque las instrucciones se pueden procesar en paralelo, dos o más a la vez y esto se efectúa en una de varias maneras. Algunos sistemas asignan un CPU a cada programa o trabajo. Otros asignan uno a cada conjunto o a partes de trabajo tratando de que cada subdivisión se pueda procesar al mismo tiempo (lo cual se conoce como "**programación concurrente**").

C.3.2 Desventajas

La mayor flexibilidad de los sistemas multiproceso significa también mayor complejidad. Esto es en el como conectar los procesadores en configuraciones de hardware y el manejo de la interacción de las tareas en el software.[2]

C.4 Sistemas Distribuidos

Un sistema informático distribuido es una colección de sistemas informáticos autónomos capaces de comunicación y cooperación a través de interconexiones hardware y software. Un sistema distribuido es en esencia un sistema multiproceso con sus elementos repartidos en distintos computadores interconectados entre si para lograr un fin en conjunto.

Los sistemas informáticos distribuidos se caracterizan generalmente por la falta de memoria compartida, los retardos impredecibles de comunicación entre nodos y por la práctica ausencia de un estado global del sistema observable por las máquinas componentes.

Debido a la falta de memoria compartida, la comunicación entre nodos es llevada a cabo por medio de *paso de mensajes*. En consecuencia, toda la comunicación y sincronización entre nodos está sujeta a retardos que son órdenes de magnitud mayores que los experimentados por los agentes de comunicación que residen en el mismo nodo. Debido a los retardos de comunicación en que se incurre al ensamblar mensajes de estado y a los componentes dinámicos de los cambios de estado, incluyendo potenciales fallos de los enlaces y de los nodos, es casi imposible que un nodo determinado evalúe el estado global un sistema distribuido en un momento dado.[3]

C.4.1 Ventajas

Sus principales beneficios potenciales incluyen[3]:

- Compartición de recursos y equilibrado de cargas.
- Comunicación y compartición de información.
- Crecimiento integral.

- Fiabilidad, disponibilidad, tolerancia a fallos.
- Rendimiento.

La compartición de recursos es una de las más importantes ventajas potenciales de los sistemas distribuidos. Los superávit y déficit temporales de recursos tales como potencia de procesamiento, capacidad de almacenamiento e información precedente de bases de datos pueden ser equilibrados para mejorar la relación coste efectividad y el rendimiento de un sistema distribuido.

La comunicación y compartición de información son formas de compartición de recursos. La necesidad y el deseo de los usuarios de comunicarse y compartir información fue determinante para el uso de sistemas distribuidos, hoy en día el internet y el correo electrónico son formas comunes esto.

El crecimiento incremental puede conseguirse en un sistema distribuido mejorando gradualmente los equipos conforme se modifiquen las necesidades de las aplicaciones y los requisitos de los usuarios

El crecimiento incremental puede conseguirse en un sistema distribuido mejorando gradualmente los equipos conforme se modifiquen las necesidades de las aplicaciones y de los requisitos de los usuarios. Además la distribución permite actualizaciones selectivas haciendo posible añadir recursos de un tipo particular cuando sea necesario para eliminar un cuello de problemas específico del sistema. De igual modo, las modificaciones de software sin afectar las aplicaciones existentes, como por ejemplo la adición de nuevas funciones, son posibles en sistemas distribuidos utilizando el modelo cliente-servidor.

La fiabilidad, disponibilidad y tolerancia a fallos son cuestiones bastante diferentes, pero su nexo es que tienen ventajas potencialmente significativas en sistemas distribuidos superiores a las del entorno centralizado. El aumento en la fiabilidad proviene de la duplicación de equipos y de la posibilidad de almacenar datos replicados en posiciones diferentes. Esto puede conseguirse de una manera gradual configurando la capacidad incremental en exceso o replicando parcialmente los datos importantes.

Las mejoras de rendimiento provienen principalmente del potencial de operación paralela cuando múltiples nodos cooperan en la resolución de un único problema. El tiempo de respuesta, una media del rendimiento, puede mejorar en sistemas distribuidos en virtud de la capacidad para colocar los datos frecuentemente utilizados cercanos a sus usuarios.

Otras ventajas potenciales de los sistemas distribuidos incluyen la reducción costes, su mayor capacidad en comparación con un único procesador y su mejor disposición para reflejar la estructura organizativa paralela del mundo real debido al control local de los datos locales.

C.4.2 Desventajas de los sistemas Distribuidos

Las principales desventajas que añade el procesamiento distribuido a los sistemas concurrentes son:[3]

- Reducida capacidad para mantener depósitos de recursos de memoria y procesador entre nodos distintos.
- Aumento de la dependencia con respecto al rendimiento y a la fiablidad de la red.
- Debilidad de la seguridad.
- Administración y mantenimiento más complejos del sistema.

Nota 10 *Note que la interacción de tareas es posible en todo procesamiento concurrente, sin importar si es multiprocesador .o no*

D Estados de tareas

Cuando un programa cambia total o parcialmente las tareas cambian de estado. El estado de una tarea está definido por su actividad. En la literatura se han definido de manera general los siguientes estados: nuevo, listo, en ejecución, bloqueado y muerto. La figura 3 muestra estos estados y algunas de sus relaciones

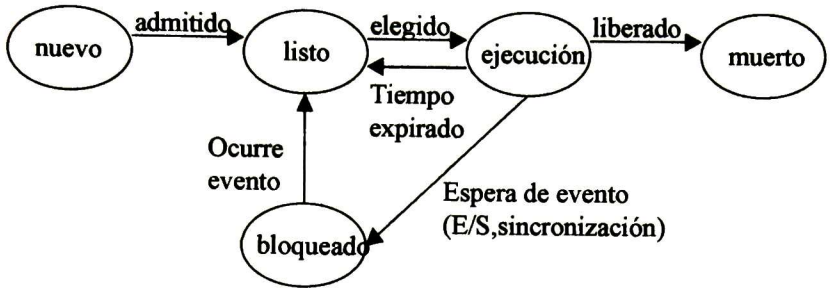


Figura 3: Diagrama de estados de una tarea

El significado de los estados es:

- **nuevo:** la tarea ha sido recién creada, pero no ha sido admitida como una de las tareas ejecutables.
- **listo**: la tarea está lista para ser ejecutada cuando obtenga un CPU.
- **ejecución o activo:** La tarea se está ejecutando. Una tarea en ejecución posee todos los recursos necesarios para su ejecución incluyendo el CPU. Un CPU sólo puede ejecutar una tarea como máximo en cada instante, de modo que los sistemas con un solo CPU solo podrán ejecutar una tarea a la vez.
- **bloqueado o suspendido:**la tarea está detenida hasta que ocurra cierto evento como una operación de E/S o una señal de sincronización.
- **muerto o liberado:** la tarea liberado del grupo de tareas ejecutables por alguna causa como fin de ejecución

D.1 Relaciones entre Tareas Concurrentes

Existen relaciones fundamentales entre tareas concurrentes:

-Competición.

-Cooperación

En virtud de la compartición de recursos de un solo sistema, todas las tareas concurrentes compiten unas con otras por la asignación de los recursos del sistema necesarios para sus operaciones respectivas.Además una colección de procesos relacionados que representan colectivamente una sola aplicación lógica suele cooperar entre si. La cooperación es habitual entre las tareas creadas como resultado de una división explícita de estas.[3].Para la adecuada cooperación y competición de los tareas del sistema es necesario mantener la *sincronía* entre ellas.

D.2 Problemas de competencia entre tareas concurrentes

Cuando en un sistema varias tareas compiten por relativamente pocos recursos y el sistema no es capaz de dar servicio a todas las tareas del sistema. Una carencia de *sincronización* de las tareas puede dar como resultado condiciones extremas:[2]

1. bloqueo mutuo también conocido como abrazo mortal o deadlock
2. inanición.

El **bloqueo mutuo** es una situación que puede presentarse cuando dos o más tareas(procesos) activas por separado compiten por conseguir recursos. El problema se genera cuando otras tareas ocupan los recursos que necesitan los primeros y no pueden liberarlos y seguir corriendo porque también aguardan otros recursos que no están disponibles. El bloqueo mutuo es completo si todo el sistema se detiene.[2] Para ilustrar el problema supongamos que la tarea P solicita los recursos X e Y y los solicita en ese orden y al mismo tiempo, la tarea Q necesita los recursos Y y X y los pide en ese orden. Si la tarea P ha obtenido el recurso X y, simultáneamente, el proceso Q ha conseguido el recurso Y, entonces ninguno de las tareas puede actuar al necesitar cada uno de ellas un recurso que se ha asignado al otro. En los sistemas grandes donde se tienen más de dos tareas y más de sos recursos compartidos la detección de estos problemas se dificulta.[19]

La **inanición** Es una situación que tiene lugar cuando la velocidad a la que una tarea puede actuar se reduce mucho por su incapacidad en acceder a un recurso particular[19]

Cuando una tarea espera por un tiempo indefinido un recurso se dice que hay inanición para esta tarea, puede darse el caso que el recurso nunca esté disponible y la tarea permanezca esperando para siempre..

El bloque mutuo es más serio que la inanición o posposición.indefinida porque este afecta a más de una tarea o a todo el sistema.

E Sincronización de tareas

La sincronización entre tareas concurrentes cooperativos es esencial para preservar las relaciones de precedencia y para evitar los problemas de temporización relacionados con la concurrencia.[9]

Por ejemplo si desearamos hacer la siguiente operación aritmética a + b*c necesitamos manejar un orden de cálculo para obtener resultados consistentes, esto es hacer la multiplicación de b*c y pósteriormente sumarle a. Veremos un poco más acerca de esto en la sección de cooperación de procesos.

Cuando se comparten recursos entre tareas concurrentes es necesario sincronizar los accesos a estos. El éxito de la sincronización de recursos compartidos se basa en la capacidad del sistema de poner un recurso fuera del alcance de otras tareas, mientras una de ellas la está usando. A esto suele llamarse *exclusión mutua.*

Estos "recursos" pueden incluir dispositivos pueden incluir dispositivos de entrada/salida, una localidad de almacenamiento, un archivo de datos. En esencia, el recurso utilizado debe quedar bloqueado, alejado de otros procesos, hasta que queda liberado.Solo cuando está libre se permite que un proceso en espera lo utilece. Aquí es donde la sincronización es vital. Un error pudiera dejar a una tarea esperando de modo indefinido o producir resultados inconsistentes.[2]

Cuando una tarea está accediendo a algún recurso compartido se dice que el proceso se encuentra en su *sección crítica* también llamado *región crítica.*[3]

Como ilustración sencilla de la naturaleza del problema , consideremos dos tareas cooperativos que pueden existir en un gestor de terminal. Supongamos que los procesos **teclado** y **pantalla** tienen a su cargo aceptar entradas de teclado y visualizarlas en pantalla respectivamiente. Supongamos que los dos procesos comparten un búfer común en el cual se almacenan los caracteres de entrada para ser visualizados. El proceso **teclado** responde a las interrupciones de teclado, recibe la entrada y la coloca en el búfer. El proceso **pantalla** hece el eco de los caracteres que hay en el búfer mostrándolos en la pantalla. Supongamos también que cada proceso mantiene un puntero privado para marcar su posición actual de trabajo en el búfer, y que se utiliza la variable compartida **eco** para llevar la cuenta del número actual de caracteres en espera de visualización.

Teclado incrementa la variable eco cada vez que se introduzca un caracter:

```
...
eco:=eco + 1
```

...

y **pantalla** decrementa eco cada vez que se visualiza un caracter:

...

eco:=eco - 1

...

supongamos que el proceso **teclado** va por delante de **pantalla** en un caracter, y que la variable compartida **eco** vale 1. Supongamos que existe una interrupción de salida en la cual la **pantalla** responde visualizando el siguiente caracter, no quedando más caracteres que procesar. La aparición de un caracter de entrada en ese momento puede hacer que el **teclado** se interponga en la secuencia de decremento de **pantalla** pudiendo ocasionar inconsistencias por ejemplo si ambos procesos leen la variable eco= 1, es decir leen la variable antes que el otro proceso en cuestión cambie la variable, el proceso pantalla la disminuiría a 0 y el proceso teclado la aumentaría a 2, el resultado final de la variable eco sería 0 o 2 según el orden de escritura dando de ese modo posibles resultados erroneos Si se usara exclusión mutua este error podría evitarse. En el momento que la interrupción de entrada se presente el proceso teclado no podrá actualizar la variable eco hasta que el proceso **pantalla** termine de actualizarla poniendo eco = eco-1= 0, y posteriormente el proceso **teclado** podría accesar la variable eco haciendo eco = eco + 1 = 1, dando así un resultado correcto.[4]

E.1 Mecanismos de Exclusión Mutua

Se han desarrollado varios mecanismos de exclusión mutua entre ellos están: probar y establecer, WAIT y SIGNAL y los semáforos.[2] A continuación veremos cada uno de estos

E.1.1 Probar y Establecer

Probar y establecer es también conocido como "TS" (por las siglas en inglés Test and Set).

Es un método de sincronización que se implementa como un arreglo de "cerradura y llave" La llave real es una variable que puede contener un cero cuando el recurso compartido está libre y un uno cuando está ocupado. Antes de que una tarea pueda entrar a su sección crítica, es necesario que obtenga la "llave" Una vez que la tiene, los demás procesos se quedan "encerrados afuera" hasta que termina; entonces libera la entrada a la sección crítica devolviendo así la oportunidad a otra tarea de usar el recurso. Esta secuencia está formada por tres acciones: 1)La tarea debe ver si la llave está disponible (llave = 0) y 2) si está disponible, la tarea debe tomarla y ponerla en cerradura (llave = 1) para que no esté al acceso de las demás tareas 3) Al terminar de usar el recurso debe liberar la cerradura llave(llave = 0). Para que este esquema funcione las hay que ejecutar estas funciones de modo indivisible para evitar bloqueos entre tareas.

Aunque este método es bastante simple de implementar, y funciona bien en un número pequeño de tareas, tiene dos grandes inconvenientes. Primero, cuando muchos procesos esperan entrar a una sección crítica, podría ocurrir la carencia de recursos o inanición porque los procesos obtienen el acceso de modo arbitrario. El segundo es que los procesos en espera se conservan en iteraciones de espera no productivas, pero consumen recursos, esto se conoce como **espera activa.**

E.1.2 Wait y Signal

es una modificación de probar y establecer diseñada para eliminar la espera activa. Dos nuevas operaciones, mutuamente excluyentes: WAIT y SIGNAL.

WAIT se activa cuando la tarea encuentra un código de condición ocupado pone el proceso en estado de bloqueado esto es inactivo sin consumir recursos y lo vincula a una cola de tareas que esperan entrar a su sección crítica. Un planificador de tareas debe seleccionar una tarea para su ejecución. SIGNAL queda activado cuando un proceso sale de la sección crítica y el código de condición se establece como "libre" El

planificador comprueba la cola de tareas que esperan la entrada a la sección crítica y lo coloca en el estado de LISTO y luego lo escoge para su ejecución.

E.1.3 Semáforos

Un **semáforo** es una variable entera no negativa que se utiliza como bandera[6]

Dijkstra(1965) propuso los semáforos como mecanismo de sincronización que ha ganado reconocimiento general.

Los semáforos son un mecanismo relativamente sencillo pero poderoso para asegurar la exclusión mutua entre tareas concurrentes para acceder a un recurso compartido.

Dijkstra introdujo dos operaciones indivisibles para manejar el semáforo los cuales llamó P y V las cuales actúan del modo siguiente:

Si suponemos que s es una variable semáforo la operación V sobre s es incrementar s La acción puede enunciarse del modo siguiente:

V(s): s:= s+1

La operación P sobre s es probar el valor de esta y, en caso de no ser cero, reducirlo en uno. La acción se puede enunciar de la siguiente manera:

P(s): si s > 0 entonces s:= s-1.

si s = 0, la tarea que llama la operación P debe esperar hasta que la operación pueda correr y esto no ocurre sino hasta s > 0, esto puede enunciarse del modo siguiente:

P(s): **while** not(s>0) **do**{};

s:=s-1;

Un semáforo cuyos variable sólo tiene permitido tomar los valores 0 (ocupado) y 1 (libre) se denomina *semáforo binario*.Para los semáforos binarios la lógica de P(s) debería interpretarse como la espera hasta que la variable semáforo s sea igual a **libre**, seguido de su modificación indivisible para que indique ocupado antes de devolver el control al invocador. **V(s)** pone el valor de la variable semáforo a **libre**.La operación **P** implementa la fase de negociación para la obtención del dato o recurso compartido y **V** implementa su liberación.

Un semáforo general puede tomar cualquier valor entero. La lógica de las operaciones P y V es la misma en semáforos binarios como en generales.

A manera de ejemplo veamos la tabla 1 la cual nos muestra una posible secuencia de estados de cuatro tareas solicitando operaciones P y V en un semáforo binario s.Vea que la tarea T3 se coloca es espera(para el semáforo) en el estado 4. Como también se ve en dicha tabla para los estados 6 y 8, cuando la tarea sale de la región crítica, el valor de s se reestablece a 1, esto a su vez dispara el despertar de uno de los procesos bloqueados, su entrada en la sección crítica y reestablecimiento de s a cero. En el estado 7 T1 y T2 no intentan ninguna acción y T4 continúa bloqueado. En el estado 8 T3 sale de la sección crítica y T4 entra.y sale; en el estado 9 se reestablece s a 0. [3][2]

Numero de estado	tarea que realiza la llamada	operación	tarea en la región crítica	tarea en espera s	valor de s
0					1
1	T1	P(s)	T1		0
2	T1	V(s)			1
3	T2	P(s)	T2		0
4	T3	P(s)	T2	T3	0
5	T4	P(s)	T2	T3,T4	0
6	T2	V(s)	T3	T4	0
7			T3	T4	0
8	T3	V(s)	T4		0
9	T4	V(s)		-	1

Tabla 1-Secuencia de estados de cuatro procesos solicitando P y V en un semáforo binario s.

Nota 11 *El problema de exclusión mutua es aplicable a cualquier sistema concurrente ya sea es sistemas multitarea donde las tareas interactivas (codependientes) usan recursos compartidos ejecutándose en un solo procesador a tasas diferentes de ejecución o en sistemas multiprocesador distribuidos o no donde de igual modo las tareas relacionadas pueden cooperar y competir por recursos comunes.*

F Cooperación entre tareas

En ocasiones varias tareas trabajan juntos para completar una tarea común. Los ejemplos famosos son los problemas de "productores y consumidores"y "lectores y escritores" Cada caso requiere la exclusión mutua y la sincronización , en esta sección se ilustran dichos problemas y dan una idea de su resolución por medio de semáforos.

F.1 Productores y Consumidores

En el problema clásico de productores y consumidores, un proceso produce algunos datos que otro consume después. A pesar que describiremos el caso de un productor y un consumidor se puede expandir a varios pares de productores y consumidores.

Considere el caso de una unidad de procesamiento (CPU) prolífico: El CPU (productor) puede generar datos de salida mucho más aprisa de lo que puede imprimir una impresora en línea (consumidor). Por lo tanto, como esto comprende a un productor y a un consumidor con dos velocidades diferentes, el sistema debe sincronizar al procesador y a la impresora, como ambas tareas se encuentran locales es posible usar una memoria intermedia común, donde el productor pueda almacenar temporalmente datos que pueda recuperar el consumidor a una velocidad más apropiada.

Dado que la memoria intermedia sólo puede contener una cantidad finita de datos, el proceso de sincronización debe retrasar al productor, para que genere menos datos cuando la memoria intermedia está llena. También debe estar preparado para recuperar la recuperación de datos por parte del consumidor cuando la memoria intermedia está vacía. Esta proceso puede implementarse mediante dos semáforos de conteo: uno para indicar el número de posiciones llenas en la memoria intermedia y el otro para indicar el número de posiciones vacías en dicha memoria.

Un tercer semáforo, mutex asegura la exclusión mutua entre las tareas.[2] A continuación presentamos las definiciones de los procesos de productor y de consumidor:

PRODUCTOR	CONSUMIDOR
produce datos	P(lleno)
P(vacío)	P(mutex)
P(mutex)	lee datos de la memoria intermedia
escribe datos en la memoria intermedia	V(mutex)
V(mutex)	V(vacío)
V(lleno)	consume datos

La siguiente lista da las definiciones de las variables y de las funciones utilizadas en el algoritmo de productores y consumidores.

Dados: Lleno,vacío, mutex, definidos como semáforos

n: número máximo de posiciones en la memoria intermedia.

V(x): x:=x +1 (x es cualquier variable definida como semáforo)

P(x): x:=x - 1

mutex = 1 significa que se permite que la tarea entre en la sección crítica

A continuación está el algoritmo que implementa la interacción entre el productor y el consumidor. COBEGIN y COEND son delimitadores que se utilizan secciones de código que se van a ejecutar al mismo tiempo.

Vacío := n

Lleno := 0

mutex = 1

COBEGIN

repetir hasta que no haya más datos del PRODUCTOR

repetir hasta que la memoria intermedia esté vacía CONSUMIDOR

COEND

El concepto de productor/consumidor se puede extender a la memoria intermedia que contiene registros u otros dato, así como a otras situaciones en que se requiere la comunicación por medio de mensajes.

F.2 Lectores y Escritores

El problema de los "lectores" y "escritores" se presenta cuando un grupo de tareas necesitan tener acceso a un recurso compartido como un archivo una base de datos para obtener información(leer) o modificarla(escribir).

Un buen ejemplo es un sistema de reservaciones de una aerolínea Los lectores son los que desean información sobre vuelos. Son lectores porque sólo leen los datos, no los modifican. Si no hay nadie que modifique la base de datos, el sistema puede permitir muchos lectores activos al mismo tiempo, no hay necesidad de exclusión mutua entre ellos.

Los escritores efectúan las reservaciones sobre un vuelo en particular. Los escritores deben organizarse con cuidado, porque modifican los datos existentes en la base de datos. El sistema no puede permitir que cualquiera escriba mientras alguna otra persona lee o escribe. Por lo tanto debe obligar exclusión mutua si existen grupos de lectores y un escritor o existen varios escritores en el sistema. El sistema debe ser justo cuando obligue a esta política, para evitar la posposición indefinida de lectores o escritores.

El sistema se puede implementar usando dos semáforos para asegurar la exclusión mutua entre lectores

y escritores. Un recurso se puede dar a todos los lectores siempre y cuando no haya lectores leyendo. Un recurso se puede proporcionar a un escritor, siempre y cuando no haya no haya lectores leyendo ni escritores escribiendo.

G Comunicación en sistemas concurentes

Las tareas cooperativas concurrentes deben comunicarse para sincronizar las acciones de las múltiples tareas participantes o bien para transmitir información de un tarea a otra.

En el intercambio de información, la comunicación debe realizarse en momentos determinados y con la aquiescencia de los elementos que en ella participan. [8]

Existen dos formas de comunicación entre tareas concurrentes, estas son: uso de memoria compartida y paso de mensajes.

G.1 Memoria compartida entre tareas

Este es un medio sencillo para la comunicación de sistemas concurrentes el cual consiste en poner en una memoria común o buffer la información común de los tareas relacionadas. En la figura 4 se ilustra esta forma de comunicación.

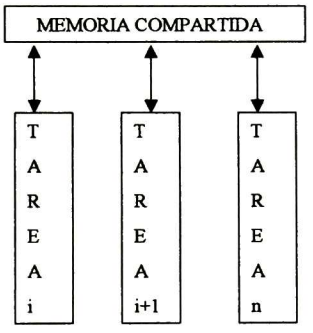


Figura 4: Comunicación entre tareas por memoria común

G.2 Paso de Mensajes

El paso de mensajes es una forma de comunicación que consiste en pasar bloques de información entre tareas sin tener una memoria compartida entre estas. En un sistema distribuido donde no existe memoria común entre tareas la comunicación y sincronización de las tareas cooperantes se realiza por paso de mensajes.En la figura 5 se ilustra esta forma de comunicación.

En general puede decirse que los sistemas envían y reciben mensajes llamadas del modo siguiente:

send (destino, mensaje)

receive(fuente, mensaje)

Existen dos formas básicas de enviar mensajes: de forma directa o indirecta los cuales explicamos a continuación.

- Envíos directos.-es un modelo asíncrono de paso de mensajes. Los mensajes pueden tener la siguiente estructura

Tarea A: **send** (B,mensaje)

Tarea B: **receive**(A,mensaje)

- Envíos indirectos.- Estos modelos usan estructuras auxiliares para el manejo de mensajes. Algunas de estas estructuras son: buzones, puertos, pipes o conductos etc. los cuales se manejan del modo siguiente:

-buzones: El modelo de de mensajes, usa estructuras auxiliares buzón para el depósito y recepción de mensajes. Los mensajes pueden tener la siguiente estructura:

Proceso A:

send (buzón, mensaje)

Proceso B:

receive(buzón, mensaje)

-puertos: El modelo de mensajes es semejante al de buzones en el sentido que varias tareas pueden enviar al mismo puerto sus mensajes, pero sólo uno de ellos puede tomarlos de allí, ejemplo de ello es el buzón de e-mail.

-pipes o conductos.: Un conducto es un buzón que permite extraer un número fijo de caracteres a la vez. El conducto no tiene la noción de mensajes sino que es el usuario el que deberá implementar los mensajes atendiendo a las particularidades del conducto

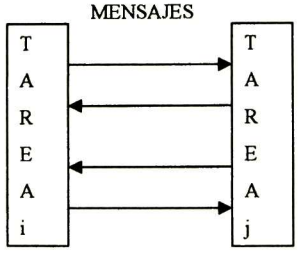


Figura 5: Comunicación por paso de mensajes

H Conclusiones

El uso de sistemas concurrentes nos aumenta el rendimiento del sistema pero también la complejidad de su control. En la concurrencia prevalecen relaciones de competencia por recursos compartidos y colaboración para lograr un fin común entre tareas; para el control de estas relaciones existen mecanismos de sincronización y comunicación para su manejo. La sincronización tienen como objetivo establecer la exclusión mutua entre tareas concurrentes con recursos compartidos regulando así la competencia por estos y ayudar a establecer un orden de ejecución de estas para su adecuada coolaboración. Algunos de los mecanismos de exclusión mutua son: probar y establecer, wait y signal, semáforos entre otros.

Los mecanismos de comunicación nos permiten intercambiar información entre tareas ayudando a la obtención de datos para la exclusión mutua y la sincronización del orden de ejecución de las mismas. Existen dos mecanismos de comunicación entre tareas concurrentes: por uso de memoria común o por paso de mensajes.

Según las características de concurrencia estos se clasifican en multitarea , multiproceso y distribuido, el primero usa las características básicas de concurencia que es la ejecución simultánea de tareas, el segundo es un sistema multitarea con varios procesadores para la ejecución de las tareas y el tercero es un sistema multiprocesador con tareas repartidas en unidades independientes interconectadas entre si, estas unidades son independientes porque no comparten procesador, ni memoria etc.siendo la única forma de comunicación entre sus tareas por paso de mensajes.

En este capítulo se explicaron las características de los sistemas concurrentes multitarea, multiproceso y distribuido, sus ventajas y desventajas, el manejo de comunicación y sincronización de tareas concurrentes así como los estados de estas.

Apéndice II- Estructura de código generado en Java

Resumen En esta tesis mediante procedimientos implementados en un ambiente visual de especificación de sistemas en RP se generó código Java distribuido. Este apéndice tiene como objetivo mostrar la estructura del código Java generado, éste código es orientado a objetos por lo que nos apoyaremos en la herramienta de modelado de sistemas orientados a objetos UML para mostrar las principales clases de el código y sus principales diagramas de secuencia, así también se mostrará un pequeño diccionario de datos de dichos digramas.

A Diagrama de clases

En la figura 6 se muestra el diagrama general de clases del código Java generado en esta tesis.

A.1 Diccionario de Datos

CLASE: **Servidor.**

FUNCION: Proveer servicio a las tareas que se lo solicitan.do de algún lugar,o bien liberar éste.Estos servicios son:Proveer el marca cambiar el marcado de algún lugar

ATRIBUTOS	DESCRIPCIÓN
-Puerto	-Establece el número de puerto de servicio a clientes.
-Input	-Canal de entrada de comunicación.
-Output	-Canal de salida de comunicación.
-Monitor	-Objeto tipo Shared que contiene los métodos de servicio a clientes.
-Connection	-punto de conexión por cliente. (tipo Socket)
-Servidor	-canal de comunicación. (tipo ServerSocket)
METODOS	DESCRIPCIÓN
-conectar	-Método que establece los canales de comunicación.
-escribir_red	-Envía mensajes de respuesta a las tareas clientes.
-leer Red	-Lee los mensajes de las tareas clientes.
-Run	-Pone en ejecución al servidor para la atención independiente de múltiples tareas clientes.
-Analizar_cadena	-Analiza los mensajes recibidos y realiza los servicios peticionados en estos.

CLASE: **Cliente.**

FUNCION: Pedir servicios a los servidores de otras tareas.

ATRIBUTOS	DESCRIPCIÓN
-puerto	-Establece el número de puerto para comunicaciones.
-in_red	-Canal de entrada de comunicación.
-out_red	-Canal de salida de comunicación.
METODOS	DESCRIPCIÓN
-conectar	-Método que establece los canales de comunicación.
-escribir_red	-Manda un mensaje a un servidor de otra tarea y obtiene la respuesta de éste.

CLASE: **Shared.**

FUNCION: Contiene los métodos de servicio usados por el servidor y las transiciones, sincroniza a las tareas.

ATRIBUTOS	DESCRIPCIÓN
-writeable[]	-Vector que contiene los permisos de escritura de los lugares.
-readable[]	-Vector que contiene los permisos de lectura de los lugares.
METODOS	DESCRIPCIÓN
-Getmarcado	-Devuelve el marcado de un lugar
-SetMarcado	-Cambia el marcado de un lugar.
-true_read	-Libera un lugar para su posterior uso.

CLASE: **Transicion.**

FUNCION: Contiene el código de comportamiento general de una transición.

ATRIBUTOS	DESCRIPCION
METODOS	DESCRIPCION
-Delay	-Ejecuta un retardo.
-Run	-Pone en ejecución a la transición.
-Condicion	-Contiene el código de comportamiento general de una transición.
-Marcar	-Método implementado por cada transición trx . Son usados en Run.
-Desmarcar	
-RestoreReads	- „

CLASE: **Trx.** Hereda el comportamiento de la clase Transicion.(x es el número de transición)

FUNCION: Contiene el código de los métodos variables de la transición.

ATRIBUTOS	DESCRIPCIÓN
METODOS	DESCRIPCIÓN
-Condicion	-Verifica que la condición de habilitación de la transición se cumpla. Usa el método GetMarcado de la clase Shared.
-Marcar	-Método que marca los lugares de salida de una transición. Usa el método SetMarcado de la clase Shared.
-Desmarcar	-Método que desmarca los lugares de salida de la transición. Usa el método GetMarcado de la clase Shared.
-RestoreReads	-Método que libera a los lugares de entrada de una transición. Usa el método true_read de la clase Shared.

CLASE: **Datos.**

FUNCION: Contiene los datos de la tarea.

ATRIBUTOS	DESCRIPCION
-CPUName[]	-Vector que contiene los nombres de los computadores donde se ejecutarán cada una de las tareas.
-Ntasks	-Número de tareas del sistema.
-Nplaces	-Número de lugares de la tarea.
-Ntran	-Número de transiciones de la tarea.
METODOS	DESCRIPCION

CLASE: **Principal.**

FUNCION: Iniciar la ejecución de la tarea, iniciando todos los componentes de la misma.

ATRIBUTOS	DESCRIPCION
-puerto	-número de puerto del servidor de la tarea.
-sh	-objeto tipo Shared.
-t	-objeto tipo transición.
-s	-objeto tipo servidor.
METODOS	DESCRIPCION
-main	-Ejecuta la tarea.

B Diagrama de Secuencias

En esta sección mostramos los principales diagramas de secuencia del comportamiento del sistema distribuido en Java.

B.1 Secuencias de operaciones que involucran a una sola tarea

En la figura 7 se ilustra el comportamiento para obtener el marcado de un lugar perteneciente a la misma tarea que la transición que lo solicita.

En la figura 8 se ilustra el comportamiento para cambiar el marcado de un lugar perteneciente a la misma tarea que la transición que lo solicita.

.En la figura 7(8) podemos ver como la transición trx solicita el marcado (cambiar el marcado) de un lugar perteneciente a su misma tarea, llamando a un método de la clase Shared, el cual le devuelve dicho marcado (una confirmación).

B.2 Secuencias de operaciones que involucran a más de una tarea

En la figura 9 se ilustra el comportamiento para obtener el marcado de un lugar que se encuentra en una tarea distinta a la tarea de la de la transición que lo solicita.

De similar modo en la figura 10 se ilustra el comportamiento para cambiar el marcado de un lugar que se encuentra en una tarea distinta a la tarea de la de la transición que lo solicita.

En la figura 9(10) podemos observar como una transición trx solicita la obtención (el cambio) del marcado de un lugar haciendo uso del método GetMarcado (SetMarcado) de la clase Shared, esta clase determina que se trata de tareas distintas por lo que pide una conexión a la clase Conexion la cual mediante la creación de un objeto cliente pide enviar un mensaje usando su método SendMensaje, posteriormente este objeto Cliente envía el mensaje a el servidor de la tarea que contiene dicho lugar usando su método escribir_red,; el servidor al recibir el mensaje determina el tipo de servicio requerido y efectúa éste auxiliándose del método de la clase Shared GetMarcado(SetMarcado), posteriormente devuelve el resultado del servicio efectuado (Marcado de lugar/ confirmación de operación) mediante su método escribir_red.

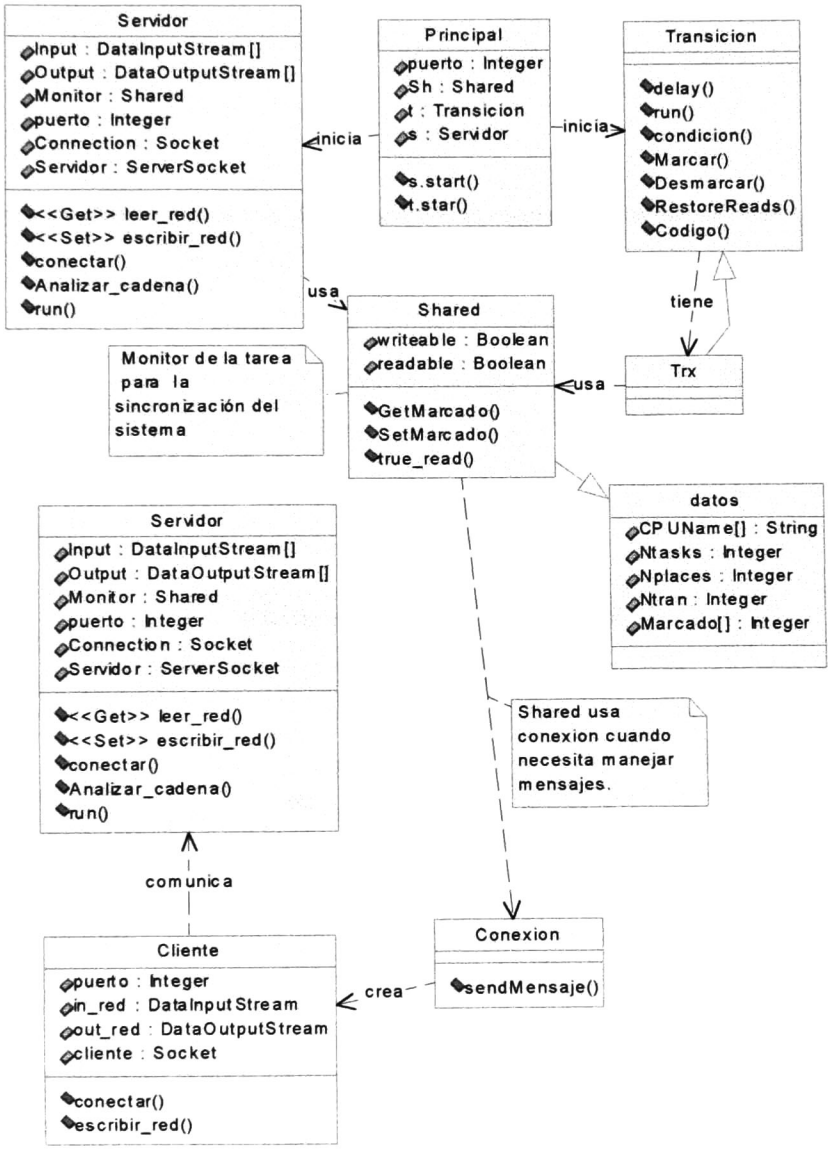


Figura 6: Diagrama de clases del código Java generado correspondiente a una especificación en RP.

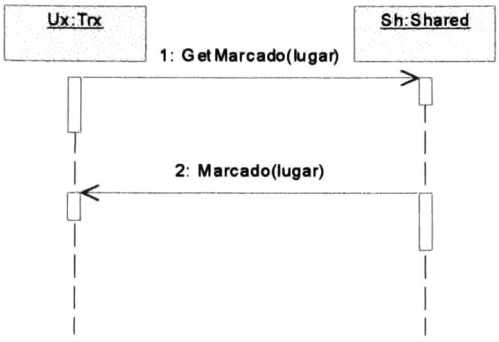


Figura 7: Secuencia para obtener el marcado de un lugar que se encuentra en la misma tarea que la transición solicitante.

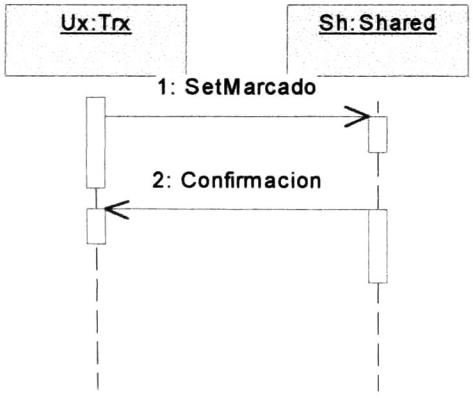


Figura 8: Secuencia para cambiar el marcado de un lugar que se encuentra en la misma tarea que la transición solicitante.

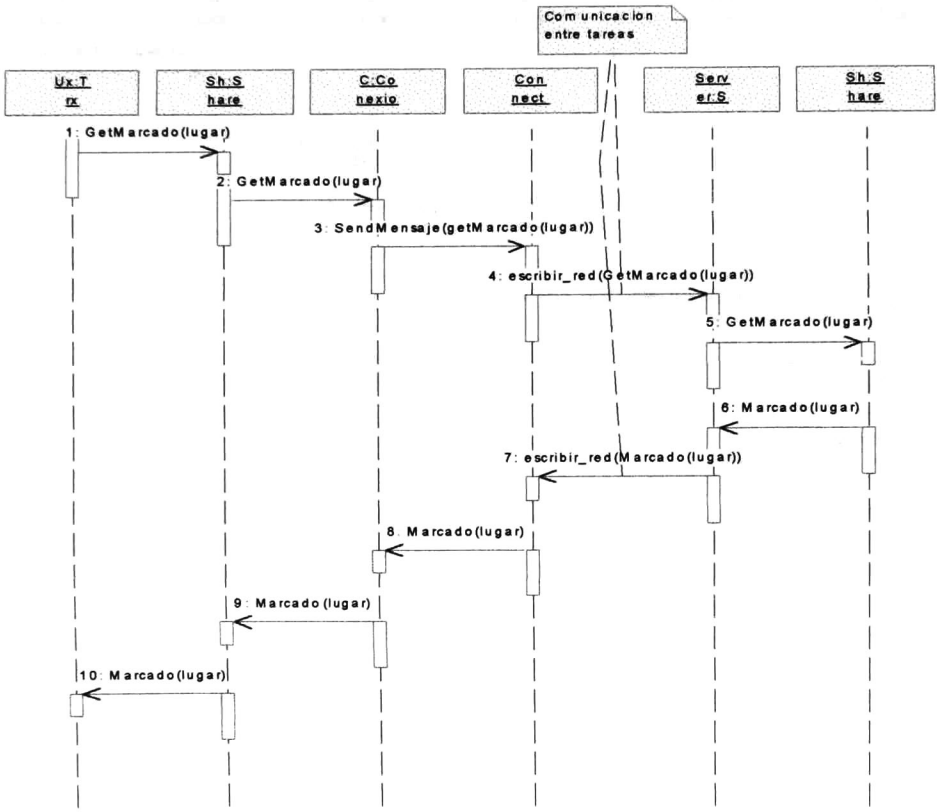


Figura 9: Secuencia para obtener el marcado de un lugar que se encuentra en distinta tarea que la transición solicitante.

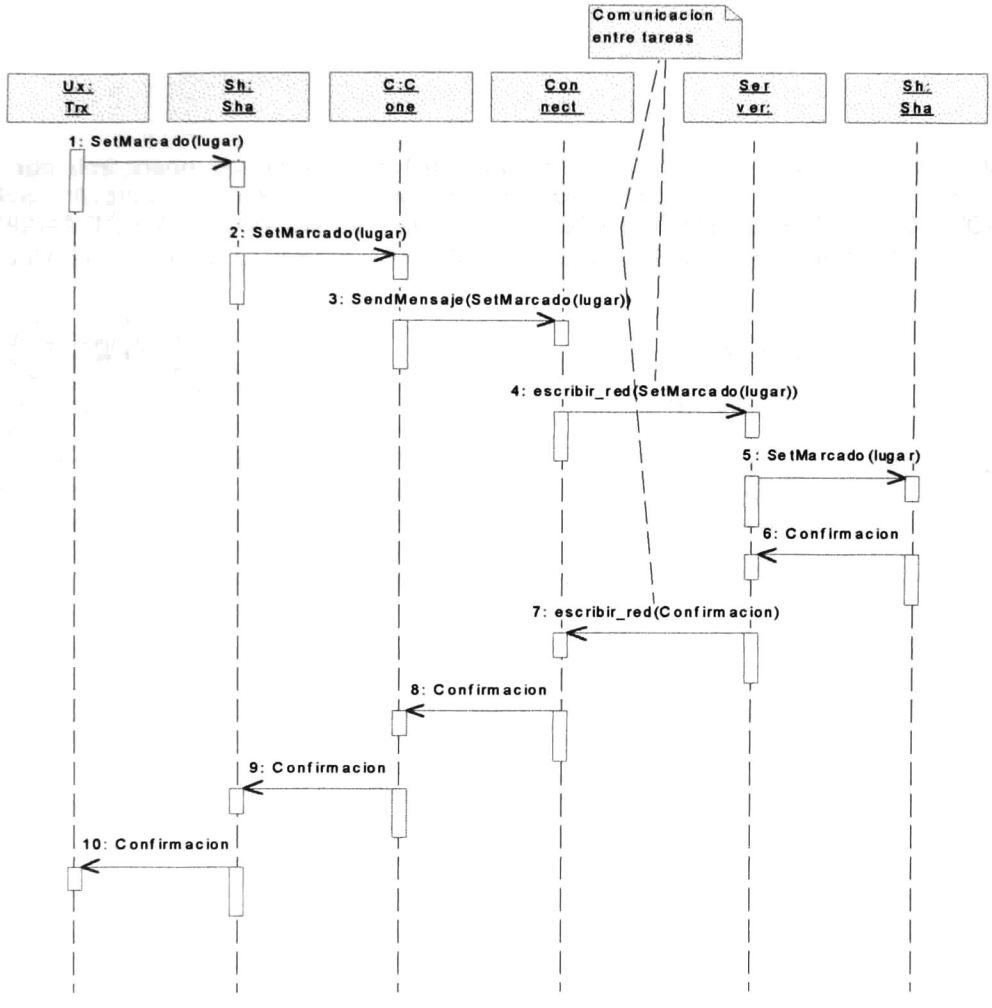
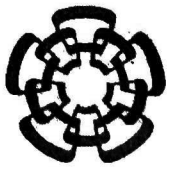


Figura 10: Secuencia para cambiar el marcado de un lugar que se encuentra en distinta tarea que la transición solicitante.



Cinvestav

Centro de Investigación y de Estudios Avanzados del IPN

Unidad Guadalajara

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, aprobó la tesis: PROGRAMACIÓN AUTOMÁTICA DE PROCESOS DISTRIBUIDOS ESPECIFICADOS CON REDES DE PETRI del(a) C. Grelty CANUL NOVELO el día 19 de Marzo de 2002 .

Dr. Luis Ernesto López Mellado
Investigador Cinvestav 3A
CINVESTAV GDL
Guadalajara

Dr. Antonio Ramirez Treviño
Investigador Cinvestav 2A
CINVESTAV GDL
Guadalajara

Dr. Félix Francisco Ramos Corchado
Investigador Cinvestav 2A
CINVESTAV GDL
Guadalajara