TESIS DOCTORAL

Modelado e Implementación de Sistemas de Tiempo Real Mediante Redes de Petri con Tiempo

Francisco José García Izquierdo

UNIVERSIDAD DE LA RIOJA

TESIS DOCTORAL

Modelado e Implementación de Sistemas de Tiempo Real Mediante Redes de Petri con Tiempo

Francisco José García Izquierdo

Universidad de La Rioja
Servicio de Publicaciones
2003

Esta tesis doctoral, dirigida por el Doctor D. José Luis Villarroel Salcedo, fue leida el 17 de diciembre de 1999, y obtuvo la calificación de Sobresaliente cum Laude

Edita: Universidad de La Rioja
Servicio de Publicaciones

ISBN 84-688-1526-8

Universidad de Zaragoza
Departamento de Informática e Ingeniería de Sistemas

Modelado e implementación de Sistemas de Tiempo Real mediante Redes de Petri con Tiempo

Tesis Doctoral

Doctorando: Francisco José García Izquierdo
Director: Dr. D. José Luis Villarroel Salcedo
Zaragoza, Diciembre 1999

Índice general

Índice de figuras

Capítulo 1

Planteamiento general y objetivos de la tesis

1.1. Ámbito de la tesis

Un *sistema de tiempo real* es un sistema informático que responde a estímulos producidos externamente dentro de un intervalo de tiempo finito y especificable [You82]. Así, la corrección de un sistema tiempo real depende no sólo del resultado lógico de la computación, sino también del instante en el cual se producen dichos resultados. Debido a sus características, estos sistemas son utilizados en aplicaciones de control, monitorización y comunicaciones.

Los sistemas tiempo real se suelen clasificar en *críticos* y *acríticos*. Los segundos se caracterizan por que se puede permitir incumplimientos ocasionales de requisitos temporales sin que ello suponga un mal funcionamiento. Tal es el caso, por ejemplo, de los sistemas de control de procesos industriales con gran inercia. Por contra, en un sistema de tiempo real crítico el cumplimiento estricto de todos los requisitos temporales es fundamental para el buen funcionamiento del sistema. Un incumplimiento, incluso ocasional, de algún requisito se traduce en un funcionamiento incorrecto e incluso peligroso del sistema. Tal es el caso de sistemas de control de vuelo o sistemas de control de robots. Evidentemente son los sistemas de tiempo real crítico los que plantean una problemática más compleja.

Es bien conocida la dificultad que entraña la construcción de software para sistemas de tiempo real. Más aún, en muchos casos la fiabilidad, tanto del hardware como del software, es un aspecto crítico debido a los posibles efectos catastróficos de un fallo (por ejemplo, un sistema de control de vuelo). Tomando el software como principal objetivo, el uso de *métodos formales* en su desarrollo puede responder a la comentada necesidad de fiabilidad. Estos pueden permitir la verificación de requisitos tanto funcionales como temporales en etapas tempranas del desarrollo, previas a la codificación. Los formalismos existentes en la actualidad aplicables a sistemas tiempo real son extensiones que incorporan el tiempo a otros más tradicionales:

- Extensiones a las álgebras de procesos como *Timed CCS* [Wan90].
- Extensiones a las lógicas temporales como *RTL* [JM86].
- Extensiones a las máquinas de estados finitas como los *Statecharts* [Har87]
- Extensiones a las redes de Petri como las redes de Petri con tiempo [BD91] o las redes de Petri temporizadas [Ram74]

Las redes de Petri (RdP en lo sucesivo) constituyen una conocida alternativa para describir comportamientos de sistemas concurrentes [Pet81, Bra83, Sil85, Mur89]. Una característica importante de los sistemas de tiempo real es la concurrencia, el sistema debe interaccionar con un entorno que evoluciona en paralelo. Las RdP definen çoncurrencia real"(en oposición al entrelazado o "interleaving"propuesto por otros modelos) hasta el punto de que suelen ser utilizadas como punto de referencia para la comparación de diferentes modelos de concurrencia.

Por ello no es de extrañar que las redes de Petri ya hayan sido aplicadas a los sistemas de tiempo real, sobre todo en relación con el modelado y la verificación formal de propiedades. Trabajos relacionados son por ejemplo, [BCMP92], donde las RdP modelan un sistema de tiempo real automóvil, con el objetivo de realizar una planificación monoprocesador; [FGP93], cuyo objetivo es la especificación formal de requisitos; [MMP+96], donde se modela y analiza el conocido problema del cruce de vías de tren; [LS87], donde las RdP con tiempo se dedican al análisis de la seguridad, recuperabilidad y tolerancia a fallos de sistemas de seguridad crítica; [AO95], donde se utilizan RdP para el modelado y estudio de estaciones de tren; [Sac94], que se dedica al modelado de especificaciones ejecutables; [BV95], dónde se extienden las RdP para considerar el diseño modular de los sistemas de tiempo real; [TYC95], que modela sistemas de tiempo real y estudia su planificabilidad; [CMP98], dónde se aplican RdP al diseño de software para el control de robots.

Sin embargo, no son muy habituales los trabajos que relacionan un modelo de sistema de tiempo real construido con redes de Petri con el software que debe animar el sistema. Será precisamente este aspecto, el de la generación de código, el principal objetivo cubierto por la tesis.

1.2. Objetivos de la tesis

La presente tesis debe situarse en el entorno de un proyecto más general, cuyo principal objetivo es el planteamiento de un método para el desarrollo de sistemas de tiempo real basado en una herramienta formal: las redes de Petri. Se pretenden cubrir todas las etapas del ciclo de vida de un sistema de tiempo real, desde la especificación de requisitos, hasta la fase de pruebas, pasando por el análisis de las propiedades, la planificación y la implementación o generación de código. Dentro de este amplio marco, esta tesis se ha enfocado sobre dos aspectos del ciclo en los que, de partida, se cuenta con mayor experiencia: el modelado y la implementación.

Por todo ello, los objetivos que se pretenden cubrir en este trabajo se puede resumir en los siguientes puntos:

1. Planteamiento de un marco formal para el ciclo de vida de los sistemas de tiempo real, basado en el empleo de redes de Petri extendidas con tiempo, las redes de Petri con tiempo (*Time Petri Nets*, [BD91]).

2. Modelado de los aspectos de control y temporales de sistemas de tiempo real utilizando redes de Petri extendidas con tiempo.

3. Generación de código (implementación) para los modelos anteriores. Éste será el objetivo fundamental de la tesis, que puede ser desglosado en:

   a) Aplicación y extensión de las técnicas de implementación de redes de Petri al caso de las redes de Petri con tiempo y estudio de su impacto sobre las prestaciones del sistema.

   b) Formalización y ampliación de las técnicas de extracción de máquinas de estados de una RdP, orientándolas a la generación de código, y prestando especial atención a las situaciones de conflicto entre transiciones, que no han sido tratados en profundidad en la literatura relacionada.

   c) Generación automática de código para los procesos anteriormente extraídos de la red.

Dejaremos fuera de los objetivos de esta tesis el estudio de la planificación de sistemas de tiempo real a través de las redes de Petri, el análisis de sus propiedades, y aspectos de tolerancia a fallos.

1.3. Contenido de la tesis

La tesis está dividida en siete capítulos y una serie de anexos, cuyos contenidos se resumen a continuación:

- Capítulo 1 (*Planteamiento general y objetivos de la tesis*). El presente capítulo, dónde se hace un planteamiento general de la tesis, enmarcándola, y exponiendo sus objetivos principales y su estructura.
- Capítulo 2 (*Introducción. Redes de Petri para el modelado, análisis e implementación de sistemas de tiempo real*) En él se hace un planteamiento general del ciclo de vida de un sistema de tiempo real basado en la utilización de redes de Petri. Se justifica la elección del formalismo utilizado para el modelado de estos sistemas mediante redes de Petri, para lo que se hace un repaso de los formalismos de redes existentes. Así mismo, se repasan ideas fundamentales sobre modelado y análisis de sistemas de tiempo real mediante redes de Petri, para terminar con un compendio de los conceptos básicos sobre implementación de redes.
- Capítulo 3 (*Implementación centralizada*). Este método de implementación, que ya se ha empleado en la literatura para redes de Petri generalizadas, es aplicado al caso de las redes con tiempo que modelan sistemas de tiempo real. Se muestra la estructura general de esta técnica y se repasan varios algoritmos de implementación interpretados. Por último se realiza una evaluación del impacto de este método en las prestaciones del sistema implementado, haciendo evidente la sobrecarga introducida, así como otras desventajas, y se realiza un balance con los puntos a favor de la misma.
- Capítulos 4 (*Implementación descentralizada I. Descomposición de una RdPT en procesos secuenciales*) y 5 (*Implementación descentralizada II. Realización software*). Los problemas de las técnicas centralizadas, desvelados en el capítulo anterior, hacen recomendable la aplicación de métodos descentralizados de implementación, que explotan la concurrencia real del sistema modelado. En estos dos capítulos, que constituyen el verdadero núcleo de la tesis, se cubren los aspectos de la implementación descentralizada de redes de Petri con tiempo que modelan sistemas de tiempo real. El primero de ellos se dedica a la extracción de los procesos concurrentes que se pueden reconocer en la red, repasando técnicas conocidas, formalizándolas y ampliándolas. El segundo, cubre los aspectos de codificación (implementación) en Ada 95 de los procesos anteriores. Se justificará que la codificación propuesta es correcta, utilizando para ello modelos de RdP de las primitivas Ada 95 empleadas que permitirán comprobar que la semántica de éstas es equivalente a la de las redes que implementan.
- Capítulo 6 (*Modelado de instrucciones Ada 95*) El capítulo muestra una serie de modelos de las primitivas de concurrencia y tiempo real de Ada 95. Estos modelos son utilizados en el capítulo 5 para justificar que la implementación software es correcta, pero pueden considerarse como una aportación en sí mismos, pues contribuyen a la clarificación de la semántica de las primitivas comentadas.
- Capítulo 7 (*Conclusiones y líneas abiertas*) Se sintetizan las conclusiones y aportaciones más importantes de la tesis, así como las posibles líneas de investigación abiertas hacia dónde merecerá la pena enfocar nuestros próximos esfuerzos.
- En los *anexos* se recogen una serie de conceptos básicos, sobre redes de Petri, reglas de reducción de redes, así como ejemplos o justificaciones cuya extensión ha aconsejado su

separación del capítulo en el que deberían estar incluidos. De especial interés es el anexo F, en el que se plantea la aplicación de las técnicas descritas en la tesis al desarrollo de un sistema de control de un robot móvil.

Capítulo 2

Introducción. Redes de Petri para el modelado, análisis e implementación de sistemas de tiempo real

Las *redes de Petri* (RdP) son un formalismo apropiado para el modelado de sistemas dinámicos de eventos discretos. Se han aplicado satisfactoriamente en campos tales como redes de comunicaciones, automatismos lógicos, sistemas de fabricación, etc. La naturaleza gráfica de los modelos de redes facilita la conexión entre diseñadores y usuarios. Los fundamentos matemáticos del formalismo permiten además el análisis de corrección y de eficiencia. Así mismo, estos modelos pueden ser implementados de forma automática utilizando técnicas hardware o software, y pueden ser utilizados con fines de monitorización del sistema una vez que éste se encuentra en funcionamiento. En otras palabras se pueden usar durante todo el ciclo de vida del sistema.

En el presente capítulo se pretende enmarcar el ciclo de vida del software de un sistema de tiempo real dentro de una metodología, que se basa en la utilización de las redes de Petri, y que tiene como objetivo fundamental la generación del código de control del sistema. Para ello, el primer paso ha de ser el modelado de los aspectos de control y tiempo real del mismo, utilizando a tal fin una red de Petri que permita el modelado de características temporales. La justificación de la elección de un determinado formalismo de RdP con tiempo será el primero de los temas que trataremos en el capítulo. Fijado el formalismo, seremos capaces de utilizarlo en la citada etapa de modelado.

El sistema modelado ha de ser implementado sobre una plataforma de ejecución concreta, que dispondrá de unos recursos finitos. Así pues, la plataforma de ejecución incorporará nuevas restricciones sobre el sistema. Por ello, será preciso planificar la ejecución de los procesos que lo compongan, de tal forma que se garanticen los requisitos temporales bajo las nuevas restricciones impuestas. El efecto de esta etapa será la asociación de determinadas prioridades a los procesos que se reconozcan en la red-modelo, prioridades que serán tenidas en cuenta durante la ejecución del sistema.

Las redes de Petri cuentan con una gran base matemática que permite la verificación de propiedades, tanto de vivacidad (las cosas pasan) como de corrección (las cosas pasan de manera adecuada). El estudio de estas propiedades, teniendo en cuenta la asignación de prioridades de la etapa de planificación, permite la verificación de los requisitos funcionales y temporales del sistema modelado en etapas anteriores. De esta manera es posible la detección temprana de comportamien-

tos erróneos, facilitando su corrección antes de la puesta a punto del sistema.

Una vez obtenido un modelo correcto del sistema que se comporta de acuerdo con las especificaciones, incluso teniendo en cuenta las restricciones de la plataforma de implementación, es preciso generar dicha implementación. La realización de esta etapa de forma automática a partir del modelo construido evita la aparición de posibles errores de codificación. Ésta será la labor a realizar en la fase de implementación o codificación. Las implementaciones descentralizadas, en las que cada uno de los procesos en los que se descompone la parte de control actúa sobre una red secuencial, será uno de los objetivos fundamentales de esta tesis. El problema se planteará en términos de encontrar una distribución de la red con el menor número procesos secuenciales.

Así mismo, y por último, la utilización de redes de Petri facilita la inclusión en el modelo de un conjunto de elementos redundantes que permitan la comprobación, en ejecución, del comportamiento del sistema. Esto facilitará la etapa de mantenimiento y prueba del mismo.

Debido a la gran extensión de este trabajo se han decidido dejar sin cubrir los aspectos de tolerancia a fallos, tales como, por ejemplo, el modelado y tratamiento de excepciones, que podrán ser materia de estudio en un futuro. En el anexo F se puede consultar un ejemplo real en el que se usan las RdP para el desarrollo del sistema de control de un robot móvil. En él se detalla el proceso seguido en cada etapa del ciclo de vida.

2.1. Repaso de los formalismos de redes de Petri

Más que un único formalismo, las redes de Petri pueden considerarse como una familia de formalismos, con distintos grados de descripción, desde alto a bajo nivel, cada uno apropiado para distintos propósitos. En todos los casos, y a pesar de la simplicidad del modelo (conjunto de elementos, relaciones y reglas), son capaces de representar comportamientos complejos.[1]

En síntesis, un modelo de red de Petri *autónomo* de un sistema dinámico consiste en (ver una descripción más completa en el anexo A):

1. Una *estructura de red*, constituida por un grafo bipartido dirigido, con etiquetas, que representa la parte estructural del sistema. Los dos tipos de nodos, *lugares* y *transiciones*, corresponden, respectivamente, a las variables que definen el estado del sistema y a sus transformadores. Las etiquetas pueden ser de muy distintos tipos, con distintos significados y niveles de abstracción, lo que conduce a distintas interpretaciones del modelo básico.

2. Un *marcado*, representado gráficamente por marcas dentro de los lugares. Representa el estado del sistema, de modo que la evolución del marcado describe la dinámica del sistema, su comportamiento. Esta evolución está descrita por reglas, siendo la más básica: una transición será ejecutada (disparada) cuando sus variables de estado (lugares) de entrada tengan el valor adecuado (suficientes marcas). La ejecución de la transición actualizará el estado del sistema cambiando el valor de las variables de estado adyacentes, retirando marcas de los lugares de entrada y colocándolas en los de salida, de acuerdo a las etiquetas de los arcos de conexión.

El mero formalismo matemático subyacente tras una RdP no asocia ningún significado a los objetos que la integran; por ejemplo los lugares no tienen por qué representar datos o estados. La conexión del formalismo con la realidad es propiciada por la *interpretación*. Las RdP autónomas son semi-interpretadas, pues aportan algún significado a los elementos de la red: los lugares describen variables de estado, las transiciones, los transformadores de los estados, y la lógica de la sensibilización/disparo aporta algunas reglas para la descripción del comportamiento dinámico.

[1] Para realizar un seguimiento de los resultados más relevantes que se han venido publicando sobre teoría de RdP y su aplicación al diseño de sistemas conviene acudir a: (1) las actas de la "International Conference on Application and Theory of Petri Nets", auspiciada por la EATCS, (2) la subserie que, bajo el título genérico de "Advances in Petri Nets", publica anualmente Springer–Verlag en su colección Lecture Notes in Computer Science y (3) las actas de la "International Conference on Petri Nets and Performance Models", auspiciada por el IEEE y el ACM.

Además es posible asociar etiquetas a los elementos de la red que, sin modificar su comportamiento, describan de forma más precisa su intención o significado (ej.: este lugar representa este estado; esta transición representa la activación de esta secuencia, etc.).

Pero en muchas ocasiones, la interpretación tiene implicaciones más importantes: p.ej., si una transición modela la finalización de una determinada actividad, puede haber restricciones temporales para su disparo una vez que se encuentra sensibilizada; o si dos o más transiciones se encuentran en conflicto, su significado puede aportar información sobre cómo debe resolverse este conflicto. Si las restricciones aportadas por la interpretación son tenidas en cuenta, el comportamiento del modelo se altera, se restringe, en realidad.

Se han propuesto diversas extensiones interpretadas de redes de Petri, algunas de las cuales consideran el tiempo. Describiremos a continuación algunas de estas interpretaciones, con especial atención sobre las extensiones con tiempo, que son el formalismo base de esta tesis. Todas toman como punto de partida el formalismo autónomo, denominado redes lugar/transición, *Place/Transition Nets (P/T Nets)* en terminología inglesa. Supondremos que el lector conoce suficientemente este formalismo. En caso contrario recomendamos la consulta de referencias clásicas, como [Sil85] (en español) o [Mur89, Bra83, Pet81], o la lectura previa del anexo A.

2.1.1. Extensiones del formalismo básico. Semánticas interpretativas

Redes de alto nivel

Una forma común de incorporar aspectos relacionados con la representación de los datos en los modelos de RdP, conduciendo frecuentemente a modelos más concisos, es permitir la presencia en los lugares de marcas distinguibles (tipadas, o "coloreadas"). Este formalismo se denomina *redes de Petri coloreadas, Coloured Petri Nets* (CPN),[JR91, Jen94]. Un propuesta alternativa de redes de Petri de alto nivel son las redes *predicado/transición, predicate/transition nets* [GL81], que se basan en lógica de predicados.

Cada marca en una CPN puede incluir información cuyo tipo (*color*) depende del lugar en el que está localizada, por lo que la definición de una CPN debe incluir la determinación de un *dominio de color* para cada lugar de la red, que indicará el tipo de dato asociado a las marcas de ese lugar. Del mismo modo que en redes P/T, el cambio de estado del sistema es consecuencia de disparos de transiciones. Lo nuevo es que, como las marcas de las CPN son distinguibles, el disparo puede ser selectivo en cuanto al tipo de marcas que retira de los lugares de entrada y las que coloca en los de salida. De esta manera, las transiciones puede ser disparadas en distintos *modos* dependientes del color de las marcas que consumen. Si el número de colores posibles es finito, las CPN tiene el mismo poder expresivo que las redes P/T, y mediante un algoritmo de desplegado (*unfolding*) es posible obtener una red P/T equivalente a un CPN determinada.

Arcos inhibidores

Otra modificación de la definición básica son los *arcos inhibidores* [Pet81]. Son representados por un arco acabado en un círculo que toca la transición. Un arco inhibidor de un lugar a una transición indica que para que la transición esté sensibilizada, el marcado del lugar debe ser menor que el peso del arco. Típicamente, los arcos inhibidores tienen pesos unitarios, por lo que son utilizados para comprobar que ciertos lugares no están marcados. En redes limitadas, los arcos inhibidores pueden ser simulados por medio de transformaciones que utilizan lugares complementarios [Sil85].

Semánticas relacionadas con el grado de sensibilización

El grado de sensibilización de una transición ($E(t)$) es una indicación del número de veces que una transición puede ser disparada concurrentemente (ver anexo A). Este concepto tiene influencia en la definición de semánticas como la de servidor sencillo o múltiple. Cuando no se considera el

disparo simultáneo y concurrente de las transiciones de la red, se habla de semántica de servidor sencillo (*single server semantic*). Sin embargo, si el grado de sensibilización de una transición es mayor que uno, es posible considerar la ejecución concurrente de la misma transición. Estaremos entonces ante una semántica de servidor múltiple (*multiple server semantic*)

Redes con prioridad

Recogeremos en esta introducción otra extensión que nos será de utilidad: la asociación de prioridades (un número entero positivo $\pi(t)$) a las transiciones de la red [ABCC87]. La asociación de prioridades tiene influencia en la resolución de conflictos en RdP, de modo que se sensibilizará (y por tanto, podrá ser disparada) únicamente la transición con mayor prioridad de entre las que cumplan la condición de sensibilización en la RdP sin prioridad.

2.1.2. Interpretaciones con tiempo

Tiempo determinista vs. tiempo estocástico

Las redes de Petri fueron propuestas inicialmente como un modelo causal, que no consideraba el tiempo. En el formalismo de RdP básico, o autónomo, la abstracción del entorno en el que la red evoluciona (incluyendo el tiempo como parte de este entorno) es total. Así mismo existe cierto indeterminismo en cuanto a él: no se especifica cuándo se disparará una transición que está sensibilizada (incluso si se disparará realmente), ni cuál de entre un grupo de transiciones en conflicto es la disparada. Las distintas interpretaciones con tiempo de las RdP han tratado de reducir el indeterminismo de distintas maneras. Por ejemplo, las redes de Petri estocásticas, *Stochastic PN* [Mol82, ABC$^+$95] lo hacen introduciendo una estimación estocástica del instante de disparo de las transiciones, mientras que las redes de Petri temporizadas, *Timed PN* [Ram74], las redes de Petri con tiempo, *Time PN* (TPN) [MF76, BD91], o las redes de Petri de tiempo-difuso, *fuzzy-timing PN* [Mur96] acotan los instantes en los que la transición puede o debe ser disparada. Las redes de Petri estocásticas temporizadas, *Stochastic Timed PN* [Ata93] combinan ambas aproximaciones. Un tratamiento estocástico no es admisible en sistemas de tiempo real, por lo que, en lo que resta, nos concentraremos en las aportaciones que reducen el indeterminismo en el disparo de una transición acotando el instante en el que éste se produce.

Semántica de tiempo de disparo o de tiempo de sensibilización

En la literatura es posible encontrar dos formas de interpretar (semántica) el parámetro temporal asociado a una transición, lo que puede afectar a la regla de evolución del marcado en dos formas, según se mantenga o no la atomicidad del disparo:

- Cuando el parámetro temporal determina el tiempo que ha de transcurrir desde que una transición queda sensibilizada hasta que se dispara, procediéndose entonces a la retirada y colocación de marcas de forma atómica, se habla de *tiempo de sensibilización* (*enabling time*).
- El parámetro temporal puede determinar también el tiempo que debe transcurrir entre la retirada (instantánea) de marcas de los lugares de entrada, y la colocación (instantánea) de marcas en los lugares de salida; en este caso se habla de *tiempo de disparo* (*firing time*). Esto es, el disparo de la transición tiene tres fases (retirada de marcas de entrada, disparo, colocación de marcas de salida) y no es atómico, sino que tienen una "duración". Por ello esta interpretación es también conocida como *semántica de duración*.

Es posible considerar la segunda interpretación como un caso particular de la primera, que puede ser simulado por una secuencia 'disparo inmediato de comienzo de transición' + 'actividad en curso' + 'disparo inmediato de transición de fin de transición'. Entre otras cosas, la interpretación de

tiempo de disparo no permite el modelado de actividades interrumpibles, lo que la incapacitará para su uso en el modelado de sistemas de tiempo real. Por ello, en este trabajo elegiremos la semántica interpretativa de tiempo de sensibilización para el disparo de una transición con tiempo.

Formas de especificar el tiempo asociado a las transiciones

Debemos elegir cómo especificar el tiempo asociado al disparo de una transición (*delay*). Suponiendo que cada transición modela el final de una actividad de nuestro sistema, el tiempo asociado representará la duración de la actividad. A veces este tiempo puede depender de factores diversos: tiempo, valores de datos internos, etc. Es posible hacer explícita esta dependencia especificando los delays de las transiciones mediante expresiones en función de *timestamps* o colores asociados a las marcas, dependiendo de si la duración de la actividad depende del tiempo o de otros factores respectivamente.

O, por otra parte, es posible simplificar la especificación del tiempo de disparo de una transición mediante la asociación de un intervalo que abarque todas las posibilidades de duración de la actividad. Esto puede ser útil para realizar análisis cuantitativos del comportamiento en el tiempo, como análisis del peor caso, o el cálculo de límites de tiempo de ejecución. Este convencionalismo, propio de las *Redes de Petri con tiempo* (RdPT), o *Time Petri Nets* (TPN), descritas en [Mer74, MF76, BD91], será el adoptado en esta tesis para especificar las duraciones de las actividades asociadas a las transiciones.

Semánticas de tiempo débil y fuerte

En el formalismo autónomo, nada se indica sobre cuándo una transición sensibilizada debe ser disparada. Este indeterminismo se ha reducido en cierta medida en el formalismo de las RdPT, al dar un intervalo de tiempo en el cual el disparo debe producirse. Por tanto las RdPT tiene semántica de tiempo fuerte (*strong time semantic*) que es la apropiada para el modelado de sistemas de tiempo real. Este término aparece en [GMMP91] en contraposición a la semántica de tiempo débil (*weak time semantic*), según la cual la transición no está obligada a disparar, pero si lo hace, debe ser en el intervalo especificado. Esta semántica puede ser útil cuando se trata de modelar una acción que puede ocurrir únicamente durante un período de tiempo, pero que no necesariamente debe ocurrir. Sin embargo, esta situación puede ser modelada, de acuerdo a la semántica fuerte, con una transición en conflicto que aborte la posibilidad de ejecución. Esta última alternativa, además, es más clara.

2.1.3. Redes de Petri con Tiempo (RdPT)

Esta sección da una definición formal de las redes de Petri con Tiempo, inicialmente descritas en [Mer74, MF76] y posteriormente tratadas en [BD91]. Nosotros adoptaremos la definición propuesta en esta última referencia por las razones que expondremos más adelante.

Definición Una Red de Petri con Tiempo (RdPT), $\mathbf{N}$, es una tupla $\mathbf{N}<P, T; Pre, Post; M_0; CIS>$, donde:

1. P es un conjunto finito y no vacío de lugares
2. T es un conjunto finito y no vacío de transiciones
3. $P \cap T = \emptyset$
4. $Pre : P \times T \rightarrow \mathbb{N}$ es la función de incidencia previa, donde $\mathbb{N}$ es el conjunto de números naturales

5. $Post : T \times P \rightarrow \mathbb{N}$ es la función de incidencia posterior, donde $\mathbb{N}$ es el conjunto de números naturales
6. M_0 es la función de marcado inicial $M_0 : P \rightarrow \mathbb{N}$
7. CIS es una correspondencia de intervalos estáticos, $CIS : T \rightarrow \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \infty)$, donde $\mathbb{Q}^+$ es el conjunto de los números racionales positivos junto con el cero [2].

La última función asocia a cada transición un par $(CIS(t_i) = (\alpha_i, \beta_i))$, que define un intervalo temporal, por lo que se debe verificar: $0 \leq \alpha_i < \infty$, $0 \leq \beta_i \leq \infty$, y $\alpha_i \leq \beta_i$ si $\beta_i \neq \infty$ ó $\alpha_i < \beta_i$ si $\beta_i = \infty$.

Regla de disparo Este intervalo permite definir la regla de disparo de una transición en una RdPT: suponiendo que la transición t_i comienza a estar sensibilizada en el instante τ_i, y que continúa sensibilizada, el disparo de la transición se producirá no antes del instante $\tau_i + \alpha_i$, y no más tarde del instante $\tau_i + \beta_i$. El intervalo de tiempos de disparo válidos para t_i será, por tanto, $[\tau_i + \alpha_i, \tau_i + \beta_i]$. La semántica de este disparo es del tipo de tiempo fuerte, el disparo se debe producir, y de tiempo de sensibilización, por lo que las marcas permanecen en los lugares de entrada durante el tiempo necesario, y una vez que se produce el disparo, éste no consume tiempo: es instantáneo.

Algunos casos particulares de intervalos, que aparecen frecuentemente, son: un intervalo puntual, $[\alpha_i, \alpha_i]$ para transiciones de tiempo de sensibilización fijo, o un intervalo $[0, \infty)$ para transiciones sin restricción temporal, como las del formalismo autónomo de redes de Petri. Por tanto el formalismo autónomo está contemplado por las RdPT, asociando intervalos $[0, \infty)$ a las transiciones. Por convención, no suele representarse explícitamente este tipo de intervalos en las RdPT.

Otros conceptos relacionados con esta definición:

- Al intervalo $[\alpha_i, \beta_i]$ se le llama *intervalo estático de disparo.*
- Al valor α_i se le llama *instante de disparo más cercano* (*earliest firing time*, EFT).
- Al valor β_i se le llama *instante de disparo más lejano* (*latest firing time*, LFT).

Estado Debido a sus propiedades temporales, el *estado* de una RdPT es algo más que un simple marcado. Es preciso registrar la cantidad de tiempo que una transición ha estado sensibilizada, ya que esta información es considerada para decidir sobre la posibilidad de disparo de una transición. Por todo ello el estado de una RdPT se define como un par $S = (M, I)$, donde M es el marcado e I es un vector de todos los posibles intervalos de disparo de todas las transiciones sensibilizadas por el marcado M.

En las referencias citadas pueden encontrarse otros resultados aplicables a las RdPT. Así mismo en [LS87] se aplican estas redes al estudio de la seguridad, recuperabilidad y tolerancia a fallos en sistemas de seguridad crítica.

2.1.4. Subclases de RdPT y otras interpretaciones temporales

Redes de alto nivel con tiempo: redes TER y relacionadas

Un formalismo más general, que incluye las RdPT como caso particular, son las redes *TER, Time Environment/Relationship nets* [GMMP91]. Las redes TER son una especialización de las redes ER, que a su vez son un formalismo de redes de alto nivel similar a las CPN.

[2] En la elección de $\mathbb{Q}^+$ radica la diferencia entre la definición propuesta por Merlin en [Mer74, MF76] y la de [BD91]. En la propuesta de Merlin la función CIS se define sobre $\mathbb{R}^+$. En nuestro caso hemos elegido $\mathbb{Q}^+$ ya que nuestro propósito último es la implementación en Ada 95, y en ella utilizaremos el tipo de datos `TIME`, que es racional, no real

En las redes TER, entre otros datos, las marcas llevan información temporal en forma de "timestamp", una variable llamada *chronos*. El disparo de una transición está condicionado por el valor de chronos de las marcas de los lugares de entrada, y produce el valor de chronos de las marcas de salida, de acuerdo a la expresión asociada a la transición. Para poder ser fieles el concepto físico del tiempo, los valores de chronos deben estar restringidos: el valor de chronos de las marcas producidas en un disparo debe ser idéntico, y no puede ser inferior al mayor de los valores de chronos de las marcas de entrada.

En la misma referencia, se hace mención a las redes TB (*Time Basic nets*), caso particular de las TER en las que el único dato que pueden almacenar las marcas es la variable chronos. Estas mismas redes son utilizadas en [GMP93], en el marco del proyecto IPTES, para la validación formal de especificaciones de requisitos de sistemas de tiempo real. La validación se realiza mediante el cómputo del grafo de alcanzabilidad. Otra simplificación de las redes ER que considera el tiempo son las HLTPN (*High Level Timed Petri Nets*) [FGP93], en las que las marcas tienen asociado un timestamp, y las transiciones dos funciones temporales que especifican un instante máximo y otro mínimo (relativos al instante de sensibilización) para el disparo de la misma.

Redes con interpretación de tiempo de disparo

Como ya hemos comentado, otra interpretación temporal de las redes de Petri consiste en asociar una duración al disparo de las transiciones. Esta semántica impide el modelado de actividades interrumpibles, lo que es muy restrictivo para sistemas de tiempo real. Sin embargo, es razonable para el modelado, análisis y planificación de sistemas de fabricación y logística. De hecho, la investigación en estos campos usando RdP se basa habitualmente en este tipo de formalismos ([CC88, RCS93, HM95, PM95]).

El primero de los formalismos que adopta esta interpretación es el de las *redes de Petri temporizadas (timed PN)* [Ram74], asociando una duración constante a las transiciones. Otra referencia que utiliza este tipo de interpretación temporal es [BCMP92], donde se utiliza un tipo de redes de alto nivel enriquecidas con tiempo, para el estudio de la planificación de sistemas de tiempo real. Una interpretación equivalente asocia la duración a los lugares [Sif79][CR83]: las marcas que llegan a los lugares están deshabilitados para sensibilizar a las transiciones durante un instante (esta interpretación puede modelarse con una RdP temporizada sustituyendo cada lugar temporizado con una secuencia 'lugar de entrada' + 'transición temporizada' + 'lugar de salida').

Otra interpretación permite que la duración sea especificada en forma de intervalos (*Interval Timed Coloured Petri Nets*, ITCPN [Aal93, AO95]). En ellas las marcas tienen asociado un sello temporal (timestamp), además de otros posibles valores (son coloreadas). La transición puede ser disparada si el valor de este timestamp en las marcas que deben ser consumidas no es mayor que el instante actual. El disparo es inmediato, pero las marcas son colocadas en los lugares de salida con un valor de timestamp más grande que el que tenían a la entrada. La diferencia es especificada por un intervalo asociado a los arcos de la red. Esta interpretación puede ser simulada mediante una RdPT sustituyendo un arco con intervalo $[\alpha, \beta]$ por la secuencia 'lugar' + 'transición con tiempo $[\alpha, \beta]$' (ver figura 2.1.a).

Otras redes con interpretación de tiempo de sensibilización

Volviendo a la semántica de tiempo de sensibilización (la de las RdPT), se pueden encontrar dos interpretaciones más en la literatura: [Han93] y [Sac94]. En ambos trabajos el tiempo es asociado a los arcos que unen los lugares con las transiciones, reflejando que el arco es permeable durante los instantes especificados.

En [Sac94] se presentan unas redes basadas en procesos secuenciales con variables locales que se comunican a través de sincronizaciones. En ellas se asocian funciones a las transiciones y condiciones a los arcos (funciones booleanas que se basan en variables locales). La interpretación temporal

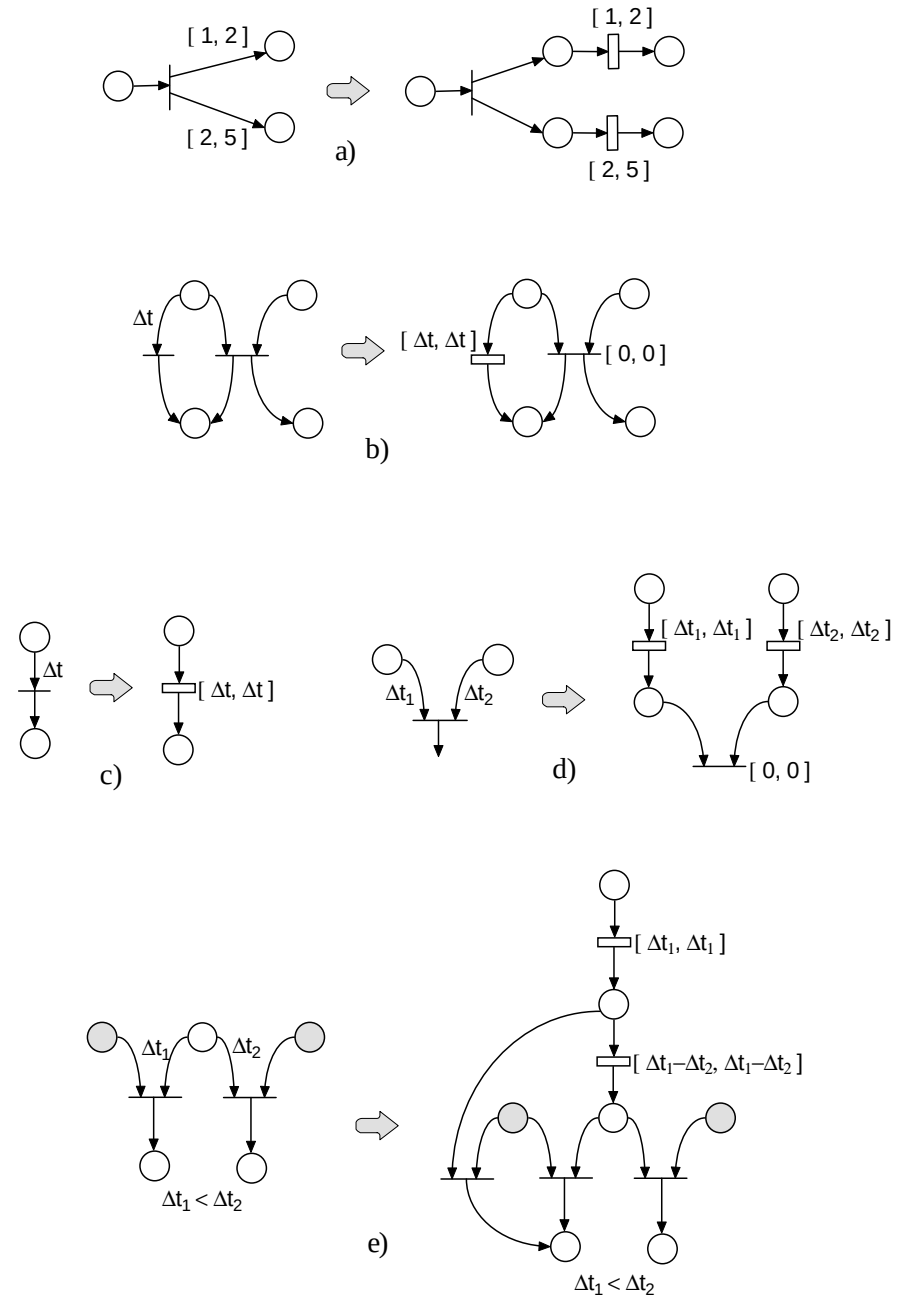


Figura 2.1: Correspondencias entre las RdPT y otros formalismo con tiempo: a) Correspondencia en RdPT de una estructura modelada según [Aal93]; b) a e) Correspondencias en RdPT de varias estructuras modeladas según [Sac94]

se basa en la asociación de un tiempo (valor constante) a los arcos que unen los lugares a las transiciones, representando que el arco se abre, es permeable al disparo, tras esa cantidad de tiempo. Si las transiciones no son de sincronización, la situación es equivalente a una transición de una RdPT con un intervalo de tipo puntual (figura 2.1.b y 2.1.c). En otros casos, a menudo es posible simular el comportamiento con una RdPT más o menos complicada (ver algunos ejemplo en la figura 2.1.d y 2.1.e).

Diferente es la aproximación descrita en [Han93]. En este caso el arco se cierra en un intervalo de tiempo. Por ello es posible que las marcas mueran en los lugares si no se produce el disparo de su transición de salida. Otra diferencia con las RdPT, radica en que en vez de registrar el instante en que una marca llega al lugar, se toma en consideración la cantidad de tiempo que la marca permanece en el lugar (antigüedad).

2.1.5. Formalismo a utilizar durante la tesis

En esta tesis modelaremos e implementaremos sistemas de tiempo real utilizando como formalismo base las redes de Petri con tiempo tal como se describen en [BD91]. Hemos elegido este formalismo por su simplicidad, y porque permite modelar gran variedad de situaciones, tal como veremos más adelante. Así mismo se ha mostrado en secciones anteriores que otros formalismos que consideran el tiempo son modelables mediante RdPT. Sin embargo, puede ser insuficiente para tratar con situaciones de duración dependiente de algún factor. En este caso el empleo de CPN, o alguna extensión de ellas que considere el tiempo (que han sido tratadas en esta sección), es imprescindible. Hasta el momento esto queda fuera de los objetivos de la tesis.

La elección de las RdPT conlleva que se asuma una semántica de tiempo de sensibilización (*enabling time semantic*) y de tiempo fuerte (*strong time semantic*). Con el fin de reducir aún más el indeterminismo remanente en este modelo base consideraremos el uso de las siguientes extensiones:

- Eventualmente asociaremos predicados a las transiciones, que condicionarán el disparo de las mismas. Los predicados dependerán de datos del sistema que no serán explícitamente representados en la red.
- Podremos asignar prioridades a las transiciones, que serán utilizadas para resolver conflictos.
- Para las transiciones que no tengan asociado un intervalo de disparo explícito supondremos un intervalo [0, 0], es decir el disparo será inmediato (en el modelo de [BD91] se suponía un intervalo $[0, \infty)$).

2.2. Modelado de sistemas de tiempo real mediante RdPT

Siendo nuestro objetivo principal la generación de código para los aspectos de control y temporales de sistemas de tiempo real, nos ocuparemos en esta sección de la elaboración de modelos, orientados a la generación de código, de este tipo de sistemas. No es un objetivo de esta tesis la aplicación de ninguna metodología de diseño en el modelado de sistemas de tiempo real, por lo que los modelos serán construidos directamente en términos de redes de Petri. Pese a ello, conocemos varios trabajos que tratan sobre la integración de formalismos de redes de Petri y de metodologías de diseño: ver por ejemplo [Gio90] o [MBV98].

En el modelado de sistemas de tiempo real es preciso representar de manera no ambigua determinados aspectos como los siguientes:

- Eventos, tanto internos como externos, a los que debe responder el sistema.
- Los patrones temporales que rigen estos eventos, es decir, su periodicidad o aperiodicidad.

- Las acciones activadas por estos eventos y sus características temporales, por ejemplo, su tiempo de computo en el mejor y el peor de los casos.

- Las interacciones entre la distintas acciones, que pueden consistir en sincronizaciones, comunicaciones, relaciones de precedencia u otras.

Las RdPT proporcionan los elementos necesarios para su representación, por lo que los modelos serán confeccionados en términos de redes de Petri con tiempo [BD91], con las salvedades y complementos comentados en la sección anterior.

2.2.1. Tipos de transiciones. Unidades de ejecución

Para el modelado de sistemas de tiempo real mediante RdP con tiempo usaremos básicamente las mismas técnicas que para RdP clásicas, salvo algunas modificaciones referidas a las transiciones que comentaremos seguidamente. En esta tesis supondremos que el lector está familiarizado con los convencionalismos sobre RdP que permiten el modelado de sistemas. En caso contrario se pueden consultar las siguientes referencias [Sil85, Mur89, Bra83, Pet81].

En el modelo de redes de Petri con tiempo propuesto por [BD91] todas las transiciones son del mismo tipo. Todas tienen la misma funcionalidad. Sin embargo, en un sistema de tiempo real existen diferentes situaciones susceptibles de ser modeladas mediante una transición. Con el fin de mostrar en nuestros modelos los diferentes papeles que una transición puede representar, y con el propósito último de hacer más clara la generación de código, vamos a distinguir entre tres tipos de transiciones que, junto con sus lugares de entrada, conformarán los elementos de modelado o *unidades de ejecución* básicos con los que construiremos nuestros modelos. Los tipos de transiciones son los siguientes:

1. Transiciones CODE. Cada una de estas transiciones junto con sus lugares de entrada representa una actividad desarrollada por el sistema (código en ejecución). Este código comenzará a ejecutarse en el momento en que la transición sea sensibilizada, es decir, todos sus lugares de entrada estén marcados. Este tipo de transiciones están etiquetadas con dos valores temporales $[\alpha, \beta]$, de acuerdo al estilo de las RdPT. En el modelo, el significado de estos valores está asociado al tiempo de cómputo del código que representa. En el mejor caso la ejecución durará α u.t. y en el peor β u.t. La finalización de la ejecución está representada por el disparo de la transición. Por tanto, la ejecución de la actividad asociada ocupa un tiempo entre $[\alpha, \beta]$ u.t. Dibujaremos una transición CODE con un rectángulo de color negro.

2. Transiciones TIME. Son transiciones asociadas a alguna actividad temporal, como un timeout o la activación periódica de un proceso. Del mismo modo que las anteriores, están etiquetadas con un intervalo temporal, si bien en este caso es puntual, $[\alpha, \alpha]$, donde α representa el instante, a partir del de sensibilización de la transición, en el que se producirá el evento. El disparo de este tipo de transiciones representa la ocurrencia del evento temporal y, por tanto, provocará que se desarrollen acciones de control en el sistema. Dibujaremos una transición TIME como un rectángulo de color blanco.

3. Transiciones SYCO. Son el resto de transiciones, y no tienen significado temporal explícito asociado. Su disparo será inmediato, lo que supone un intervalo temporal implícito $[0, 0]$. Son usadas para modelar sincronizaciones ("SYnchronizations") y tareas de control (ÇOntrol") sin requisitos temporales. El disparo de este tipo de transiciones conduce a simples cambios de estado del sistema o es utilizado para sincronizar actividades, modelar eventos no temporales, excepciones, etc. Estas transiciones pueden tener asociados predicados que condicionen su disparo. Dibujaremos una transición SYCO como un segmento delgado.

Puede deducirse fácilmente que el modelado de la parte operativa del sistema recae sobre las transiciones CODE, mientras que el de la parte de control y de supervisión temporal depende de transiciones SYCO y TIME.

Cualquier transición podrá tener asignada una prioridad, aunque la forma de interpretar ésta será distinta dependiendo del tipo de transición al que corresponda. Mientras que en el caso de las SYCO y TIME, será utilizada para deshacer conflictos (ver sección 2.5.1) entre varias transiciones sensibilizadas (*prioridad de control*), la prioridad de una transición CODE está relacionada con la planificación del sistema y será tenida en cuenta durante la ejecución del mismo (*prioridad de código*). Hablaremos de ello más adelante, en la sección 2.3 y en el capítulo 5.

2.2.2. Unidades de ejecución

Estos tres tipos de transiciones son la base lo que hemos denominado *unidades de ejecución*, o componentes básicos que representan acciones elementales en el sistema. Como consecuencia de la anterior clasificación distinguiremos entre tres unidades de ejecución:

1. Unidad CODE: la formada por una transición CODE y sus lugares de entrada, que modelará la ejecución de un cierto código en el sistema; esta unidad comenzará a ejecutarse cuando se sensibilice su transición, estará en ejecución mientras sus lugares de entrada permanezcan marcados, y terminará en un instante determinado por su intervalo temporal, o si deja de estar sensibilizada (es abortada).

2. Unidad TIME: la integrada por una transición TIME y sus lugares de entrada, que corresponderá a la posible ocurrencia de un evento temporal; la unidad se ejecutará cuando transcurra la cantidad de tiempo especificada por su intervalo temporal, siempre que la transición permanezca sensibilizada de forma continuada durante esa cantidad de tiempo.

3. Unidad SYCO: la formada por una transición SYCO que representa acciones de control o sincronización; se ejecuta inmediatamente en el momento en que se detecta su sensibilización

El resto de elementos, lugares, de la red serán utilizados para modelar recursos, datos, condiciones, etc., de acuerdo a la forma general de modelar utilizando redes de Petri.

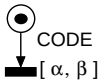
2.2.3. Modelado de situaciones habituales

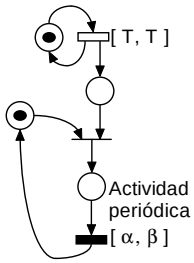
Con estos tres tipos de transiciones-unidades de ejecución es posible modelar multitud de situaciones: ejecución de códigos, timeouts, ejecuciones periódicas, aperiódicas, sincronizaciones, citas, citas extendidas, etc.[3] Para mostrar la potencia de modelado de las RdPT se han seleccionado un conjunto de situaciones comunes que aparecen en sistemas de tiempo real (se pueden consultar en la figura 2.2, junto con un fragmento de código Ada 95 que clarifica su significado):

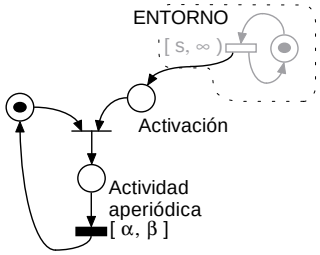
1. *Ejecución de un código secuencial.* Como ya hemos comentado en la sección 2.2.2, está representada por una transición CODE junto con sus lugares de entrada y se caracteriza por un valor de tiempo de ejecución máximo y otro mínimo.

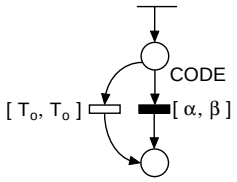
2. *Eventos de activación.* Los eventos en sistemas de tiempo real pueden ser clasificados por su origen [KRP$^+$93]: eventos de entorno (eventos provenientes del exterior), eventos internos (ocurrencia de un cambio de estado dentro del sistema) y eventos temporales (causados por el transcurso de una cantidad de tiempo). Desde un punto de vista temporal, las secuencias de eventos se caracterizan por su patrón de llegada. Hay cinco tipos de patrones de llegada: periódico, irregular, acotado, en ráfaga y no acotado. Nosotros mostraremos, a modo de ejemplo:

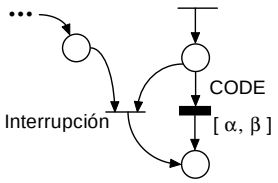
[3] En esta tesis no consideraremos el estudio del modelado e implementación de excepciones u otros mecanismos de tolerancia a fallos.

CÓDIGO SECUENCIAL	`code;`	

ACTIVACIÓN PERIÓDICA	`task body Periodica is`
	`  next: time := CLOCK;`
	`  T: constant duration := ...;`
	`begin`
	`  loop`
	`  next := next + T;`
	`  delay until next;`
	`  actividad_periodica;`
	`  end loop;`
	`end Periodica;`
		

ACTIVACIÓN APERIÓDICA	`task body Aperiodica is`
	`begin`
	`  loop`
	`  accept activacion;`
	`  actividad_aperiodica;`
	`  end loop;`
	`end Aperiodica;`
		

TIME-OUT EN EJECUCIÓN DE CÓDIGO	`select`
	`  delay To;`
	`then abort`
	`  code;`
	`end select;`
		

TRANSFERENCIA ASÍNCRONA DE CONTROL	`select`
	`  accept interrupcion;`
	`then abort`
	`  code;`
	`end select;`
		

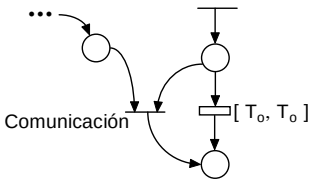
TIME-OUT EN COMUNICACIÓN	`select`
	`  accept comunicacion;`
	`or`
	`  delay To;`
	`end select;`
		

Figura 2.2: Modelos, en términos de RdPT, de situaciones dependientes del tiempo que aparecen habitualmente en sistemas de tiempo real. La segunda columna muestra la semántica de la situación en Ada 95

   a) el periódico (dada una ocurrencia cualquiera del evento en el instante t, la próxima ocurrencia será en el instante $t+T$, dónde T es constante y se llama *período* del evento)

   b) otros. En este caso la activación del evento se producirá al marcarse el lugar que lo representa. Este marcado puede ocurrir en cualquier momento, por una decisión de control interna, o por un evento externo, por ejemplo en el caso de un evento acotado, dejando una separación mínima entre dos ocurrencias. Como razonaremos seguidamente, este tipo de activaciones externas no serán modeladas, salvo que se precise un análisis del sistema.

3. *Eventos externos*, entendidos como aquellas situaciones dependientes del entorno que puedan alterar el comportamiento del sistema. La notificación de estos eventos puede realizarse, por ejemplo, mediante interrupciones. Consideramos que este tipo de eventos son parte del entorno y no del control del sistema (centro de nuestro interés). Por ello no deben ser modelados explícitamente ni generarse código para ellos, aunque sí debe ser considerada la parte que debe responder ante ellos. Por lo tanto, los eventos en sí, no cómo se producen, deben estar presentes en el modelo de alguna manera, por ejemplo asociando predicados a transiciones, cuyo disparo representará la ocurrencia del evento. Los predicados dependerán de variables o datos actualizados por el evento externo (interrupción u otros), datos que no serán representados explícitamente. Únicamente si se desean analizar las propiedades del sistema, será necesario el modelado completo del entorno del sistema y sus efectos sobre él, es decir, los eventos externos (véase el ejemplo del sistema de control de una mina en la sección 2.2.5).

4. *Suspensión o aborto de una actividad*. Es común en sistemas de tiempo real incorporar mecanismos de suspensión de la ejecución por motivos de seguridad o por cambios de modo. Los mecanismos que hemos seleccionado son (ver figura 2.2):

   a) *Timeout en ejecución* (la ejecución de una actividad se abandona si no es finalizada antes de un instante especificado). La situación se modela mediante una transición TIME en conflicto con una CODE. Si la transición CODE no es disparada (fin de ejecución) en To u.t. la transición TIME es disparada, abortando la ejecución del código (ya que la marca del lugar de entrada es extraída, desensibilizando la transición CODE).

   b) *Transferencia asíncrona de control* (que permite abandonar la actividad si se produce determinado evento). La ejecución del código, modelada por la transición CODE, puede ser abortada por la sensibilización, y consiguiente disparo inmediato, de una transición SYCO en conflicto con ella.

   c) *Timeout en una comunicación* (para evitar una espera infinita, la solicitud de comunicación se abandona tras un tiempo de espera sin obtener respuesta). Esta situación se modela mediante el conflicto entre una transición SYCO y otra TIME. Si el comienzo de la comunicación, representado por el disparo de la transición SYCO, no se produce en To u.t., entonces la transición TIME es disparada, abortando la posibilidad de comunicación.

Modelado con otras extensiones con tiempo de las RdP

Con respecto al modelado de estas situaciones con otras interpretaciones temporales, se puede considerar lo siguiente:

- *Ejecución de código secuencial*. El uso de intervalos temporales asociados a las transiciones permite la representación de código cuyo tiempo de ejecución dependa del estado o de entradas del sistema. El *earliest firing time* (EFT) de la transición representa el mejor caso de ejecución y el LFT, el peor. Las interpretaciones con duración de disparo fija (como las RdP temporizadas) sólo representan uno de estos tiempos, generalmente el de peor caso.

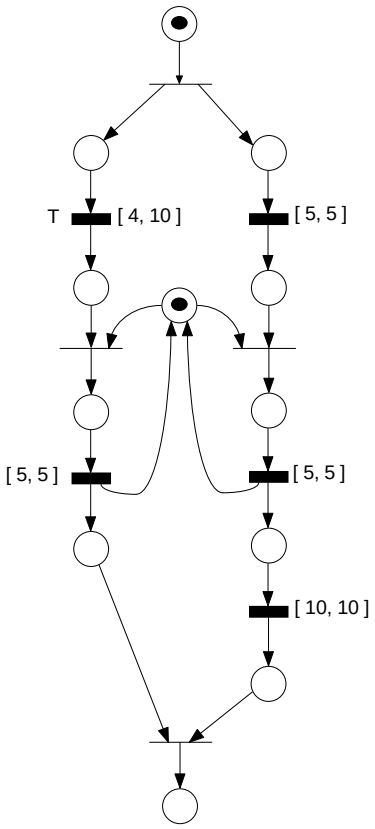


Figura 2.3: La ejecución de todos los elementos en su peor caso no equivale al peor caso de ejecución del sistema completo. Si el tiempo de ejecución de la transición T es el de peor caso (10 u.t.) la red se ejecutará en 20 u.t.; sin embargo, si el tiempo es el de mejor caso (4 u.t.), la red tardará 24 u.t.

Por tanto, no se representan todas las posibilidades de evolución del sistema, y, en consecuencia, el peor caso de ejecución del sistema puede no ser considerado (ver figura 2.3). Una posibilidad es la discretización del tiempo de ejecución en varios instantes de ejecución fijos. Sin embargo esta solución presenta dos inconvenientes, ya que se pierde la correspondencia entre la estructura del sistema y la estructura del modelo, siendo ésta última artificialmente complicada, y, por otra parte, si la discretización no corresponde con todas las posibilidades de tiempo de ejecución del código, entonces el modelo no permitirá la validación del sistema.

- *Activación periódica.* Todas las interpretaciones que añaden tiempo a las RdPT permiten el modelado de esta situación.

- *Timeout en ejecución.* Debido a que en semánticas de tiempo de disparo las transiciones no son interrumpibles, esta situación no puede ser modelada en redes temporizadas. Además, sin intervalos de tiempo, es imposible modelar actividades de duración variable que puedan elevar un timeout.

- *Timeout en comunicación.* En redes con semántica de tiempo de sensibilización, el modelado de esta situación se realiza como se indicó en la figura 2.2. Sin embargo en modelos con semántica de tiempo de disparo, y por la misma razón que en el punto anterior, no es posible modelar esta situación, pues el timeout no puede interrumpirse si la comunicación se produce.

- *Transferencia asíncrona de control.* Por la misma razón que en los puntos anteriores, esta situación puede ser modelada por una red con semántica de tiempo de sensibilización, pero no en redes con semántica de tiempo de disparo.

2.2.4. Restricciones sobre el modelo

Al ser la generación de código nuestro objetivo final hemos de verificar que el modelo cumpla ciertas condiciones que permitirán esta generación. Estas condiciones son las siguientes:

1. Hasta el momento sólo nos ocuparemos de redes ordinarias (arcos de peso unitario, ver anexo A). Consideramos que es suficiente la potencia de modelado aportada, y en muchas ocasiones (caso de redes limitadas) los arcos con pesos e inhibidores pueden sustituirse por estructuras ordinarias [Sil85]. Esto no significa que las técnicas aquí descritas no se puedan ampliar a otro tipo de redes (véanse trabajos como [BC93, Bri95] que abordan la implementación centralizada de redes de Petri con arcos con pesos e inhibidores).

2. Se deberá verificar que las transiciones de la red puedan ser disparadas al menos una vez. De esta manera no es posible modelar partes de un sistema que no pueden ser ejecutadas. Notar que no se exige que la red sea viva, ya que, si se exigiese, no se podrían modelar las fases de inicialización ni terminación de los sistemas. En el capítulo 4 se argumentará el porqué de este requisito.

3. Como consecuencia de lo anterior, el marcado inicial de los lugares origen (lugares sin transiciones de entrada) debe ser uno, para permitir el disparo de sus transiciones de salida. Así mismo los lugares terminales (lugares in transiciones de salida) deben ser 1-limitados, ya que estos lugares representan el final del funcionamiento del sistema, que sólo puede terminar una vez.

4. El grado de sensibilización (ver anexo A) de las transiciones debe ser como máximo uno (transiciones 1-disparables). Sólo consideramos sistemas secuenciales sin código reentrante. Para ello es suficiente que uno de los lugares de entrada de la transición sea 1-limitado.

5. No incluiremos en nuestros modelos transiciones sin lugares de entrada, transiciones sin precondición que constantemente pueden ser disparadas e introducir marcas en el sistema.

6. Cada lugar puede tener como máximo, una transición CODE de salida, si bien puede contar simultáneamente con varias TIME y varias SYCO. De este modo se evita que puedan existir conflictos entre transiciones CODE (lo que modela que un sistema está ejecutando simultáneamente dos códigos distintos). Consideramos que estas situaciones de carrera entre códigos, además de difíciles de tratar, no son muy habituales y, en conclusión, no las contemplaremos en nuestros modelos.

7. Los valores temporales asociados a las transiciones TIME deben ser iguales, es decir $[\alpha, \alpha]$, ya que se representan eventos temporales y estos deben producirse en un instante determinado. Además se debe verificar que $\alpha > 0$, ya que en caso de ser 0 se trataría de una transición SYCO.

2.2.5. Ejemplos de modelado

a) Un proceso periódico sencillo

Como muestra, consideremos la RdPT de la figura 2.4, en la que se modela un fragmento de un sistema de tiempo real compuesto por un proceso periódico (T= 100 u.t.). El proceso adquiere un recurso necesario para la ejecución de un código (CODE) que tiene un timeout (T.O.1) asociado de 75 unidades de tiempo. Tras la ejecución del código, el recurso es liberado procediéndose al establecimiento de una comunicación con un segundo proceso por medio de una cita (entry_A), que también tiene un timeout asociado de 5 u.t. (T.O.2). Se adjunta el código Ada para facilitar el entendimiento del modelo:

```
task body Proceso_A is
  siguiente: Time := CLOCK + 100.0;
begin
  loop
    delay until siguiente;
    Recurso.Adquirir;
    select
      delay 75.0;
    then abort
      CODE;
    end select;
    Recurso.Liberar;
    select
      Proceso_B.entry_A;
    or
      delay 5.0;
    end select;
    siguiente := siguiente + 100.0;
  end loop;
end Proceso_A;
```

```
task body Proceso_B is
  ...
begin
  ...
  accept entry_A do
    CODE2;
  end;
  ...
end Proceso_B;
```

Se han destacado tres elementos de la figura 2.4 que corresponden con estructuras típicas en sistemas de tiempo real:

- La caja A modela la activación periódica del proceso. Cada 100 u.t. la transición Activación Periódica se dispara, sensibilizando la transición Comenzar Ciclo y provocando la ejecución del resto del proceso.
- La caja B (transición CODE) muestra una acción, es decir, código, que debe ser ejecutado por el proceso. La ejecución comienza en el momento en el que se marca el lugar de entrada de la transición y debe acabar en un instante no inferior a 60 u.t., si bien es posible que se bloquee durante un tiempo indeterminado. Cuando la ejecución acaba, la transición CODE es disparada. Para prevenir los efectos del eventual bloqueo, el código tiene asociado un timeout (transición T.O.1) que se disparará 75 u.t. después de que el código haya iniciado su ejecución.
- La caja C muestra una comunicación síncrona (cita extendida) con el proceso_B que tiene un timeout (transición T.O.2). Supongamos que el lugar Espera_Comunicar se marca en el instante t. Si la transición entry_A no se dispara (comienzo de la comunicación) antes de $t + 5$ u.t., la transición T.O.2 se disparará, retirando la marca del lugar Espera_Comunicar y abortando el comienzo de la comunicación. Si la sincronización tiene lugar, se ejecutará un código (transición CODE 2).

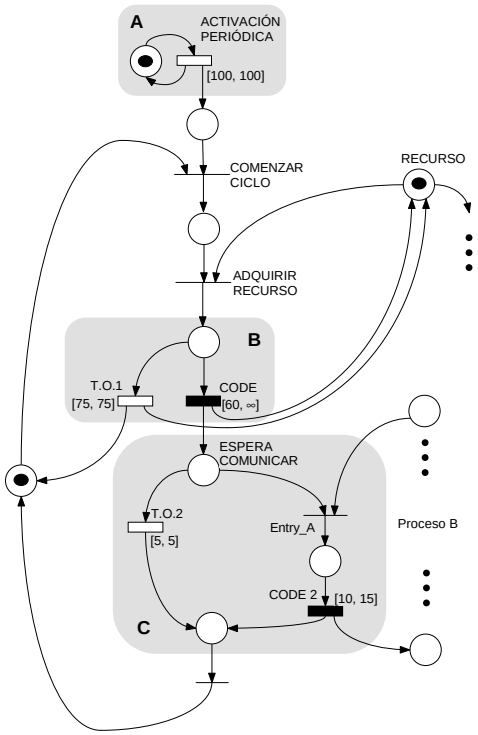


Figura 2.4: Ejemplo de sistema de tiempo real compuesto por un proceso periódico modelado con una RdPT

b) Sistema de control de una mina

Este ejemplo se ha tomado de [BW97]. Con el propósito de simplificarlo, se ha realizado en él algunas modificaciones: (a) no se ha considerado ni el operador de la consola ni el registro de datos del sistema, y (b) se han eliminado el sensor de flujo de agua y de aire. Con estas modificaciones las especificaciones del problema se pueden definir como sigue. El sistema es usado para extraer agua de una mina (se proporciona un esquema simplificado de la situación en la figura 2.5). Los sensores de nivel de agua se gestionan a través de interrupciones, mientras que los demás son leídos directamente.

Requisitos funcionales Las especificaciones funcionales del sistema se pueden dividir en dos componentes: el control de la bomba y la monitorización de la bomba. El *sistema de control de la bomba* debe supervisar el nivel de agua en el pozo. Cuando ésta alcance el nivel de alarma superior la bomba es encendida, drenando el pozo hasta que el agua alcance el nivel mínimo. Entonces la bomba es apagada. La bomba sólo debe funcionar si el nivel de metano en la mina es inferior a un cierto nivel. El *entorno* debe ser monitorizado para medir el nivel de metano en el aire: hay un nivel a partir del cual no es seguro extraer mineral ni operar con la bomba de agua. También debe medirse el nivel de monóxido de carbono en el aire.

Requisitos no funcionales Sólo consideraremos para este ejemplo el estudio de los requisitos temporales.

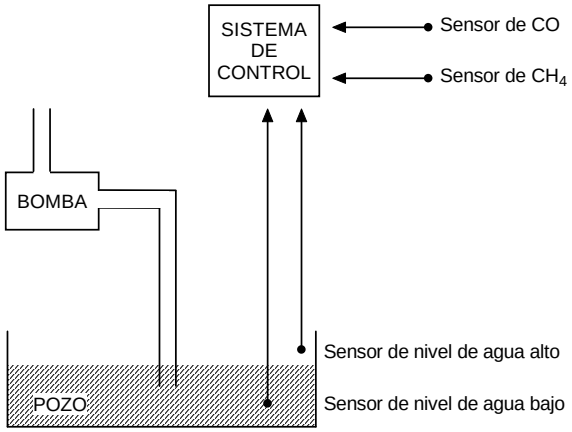


Figura 2.5: Sistema de control de una mina

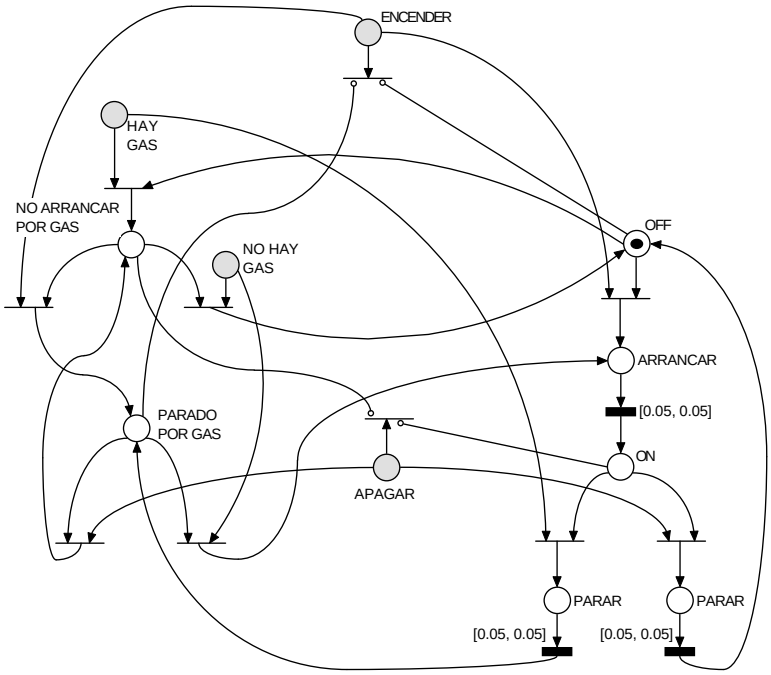


Figura 2.6: RdPT que modela el sistema de control de la bomba de extracción de agua de la mina

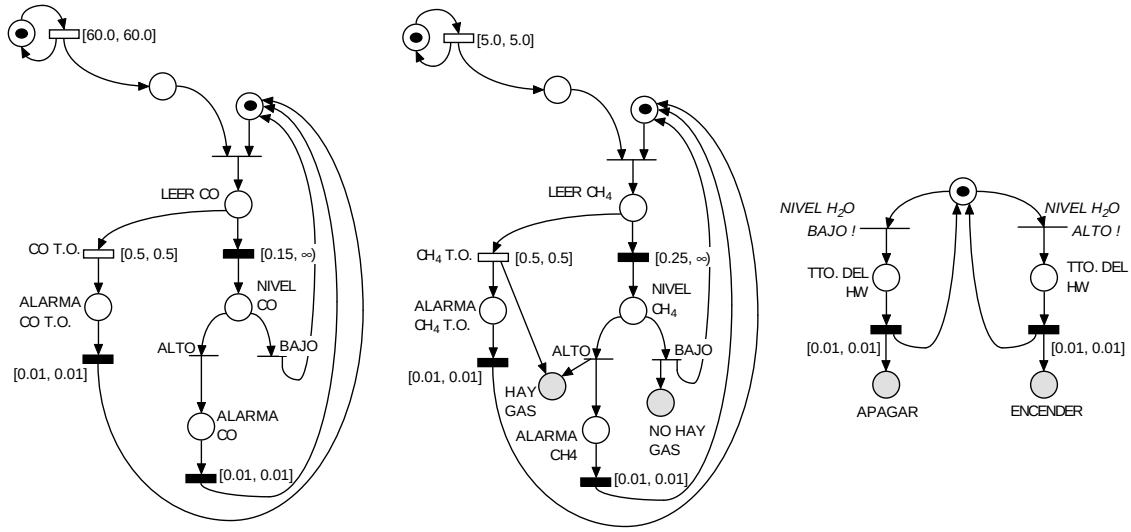


Figura 2.7: Sistema de control de una mina: RdPT que modela la lectura de los sensores del sistema

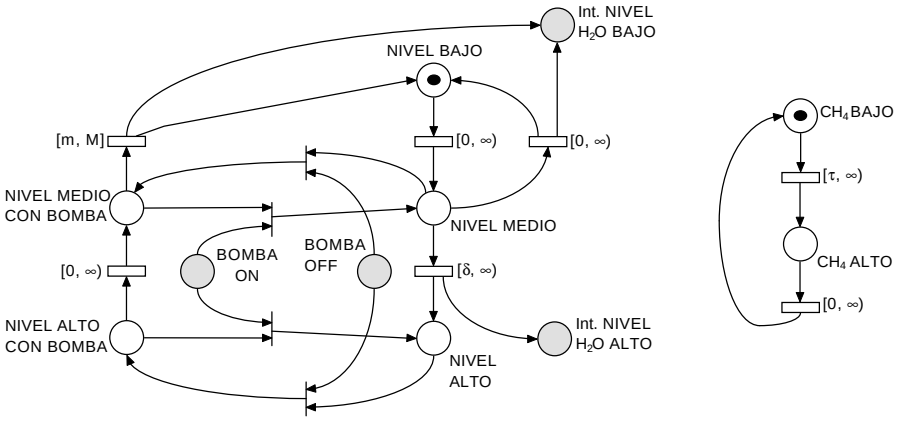


Figura 2.8: Sistema de control de una mina: RdPT que modela la variación del entorno

1. *Período de monitorización.* Consideraremos un período de 60 segundos para el sensor de CO; en el caso del sensor de metano será de 5 segundos; supondremos que los detectores de nivel de agua son dirigidos por eventos, y que el sistema debe responder a ellos en 20 segundos.
2. *Plazo de apagado* en presencia de un nivel de metano superior al umbral máximo de 1 segundo.
3. *Plazo de información al operador* de 1 segundo, ante niveles de metano y/o monóxido de carbono superiores a los permitidos.

La RdPT que modela el sistema La red de la figura 2.6 muestra el modelo del controlador de la bomba. En funcionamiento normal, ésta puede estar encendida (ON) o apagada (OFF). La bomba puede permanecer deshabilitada si el nivel de metano es muy alto, debiendo volver al estado anterior si el nivel desciende por debajo del umbral de seguridad. Por ello el controlador puede ser visto como una máquina de estados finita con cuatro estados: ON, OFF (si no hay alarma de gas), no arrancar por gas (la bomba está OFF y no debe arrancar por alarma de metano) y parada por gas (la bomba debería estar ON, pero la alarma de metano hizo que se apagara). La figura 2.7 muestra las actividades periódicas del sistema: el muestreo de los niveles de metano y monóxido de carbono. Ambos modelos son similares. La activación periódica de las actividades es modelada con una transición TIME con un bucle y un intervalo de disparo de límites iguales al período de muestreo. Para proteger al software de los eventuales fallos del dispositivo de muestreo, se ha protegido el acceso a éstos mediante un timeout: si el dispositivo no responde, la lectura es suspendida y el operador es informado.

En la misma figura 2.7, se muestra el modelo del sensor de nivel de agua. En él es destacable el modelado de las interrupciones. Ante la imposibilidad de vincular el disparo de una transición con la ocurrencia de una interrupción, una manera de modelar la situación es asociar un predicado a la transición. El valor del predicado será modificado por el manejador de interrupción, de modo que cuando ésta se produzca, el disparo de la transición será inmediato. Con vistas a la implementación del sistema, ni las variables que conforman el predicado, ni la forma en la que éstas varían, serán modeladas.

Modelado del entorno Sólo a efectos de análisis del comportamiento del sistema es posible modelar el entorno que rodea el sistema de control de la mina, y que provocará la activación de estas interrupciones. En este sentido, la figura 2.8 muestra los modelos de comportamiento de los niveles de metano y agua.

2.3. Planificación

A pesar de no ser un objetivo de la tesis, es preciso aclarar ciertas cuestiones relacionadas con la planificación de un sistema de tiempo real modelado mediante una RdPT.

El modelo teórico de concurrencia de las redes de Petri supone la existencia de infinitos recursos, de modo que cualquier unidad de ejecución que pueda ser ejecutada lo será en paralelo con las demás en la misma situación. Sin embargo, la implementación software cuenta con un conjunto de restricciones impuestas por la plataforma de ejecución: *restricciones hardware*, referentes a la existencia de un conjunto de recurso finitos, como el número de procesadores, y *restricciones software*, debidas al sistema operativo y al lenguaje de codificación, que aportan unas determinadas primitivas de concurrencia y tiempo real. Por ello es necesario decidir qué unidad de ejecución será ejecutada cuando varias puedan serlo concurrentemente y no se disponga de un procesador para cada una.

Este problema de planificación ("scheduling") de procesos y recursos ha sido profusamente tratado en la literatura especializada en temas de programación tiempo real, dando lugar a múltiples clasificaciones [But98], como la que distingue entre planificación estática y dinámica dependiendo

de si las decisiones de planificación se toman antes o durante la ejecución de los procesos. Uno de los enfoques que se ha adoptado tradicionalmente, y que será el utilizado en esta tesis, ha sido la asignación de prioridades a los procesos, de modo que, en cada momento, se ejecute el proceso más prioritario. Las prioridades pueden ser:

- estáticas, dando lugar a algoritmos de planificación como el de prioridad al más frecuente (*rate monotonic scheduling*) [LL73], o el algoritmo de prioridad al más urgente (*deadline monotonic scheduling*) [LW80],
- dinámicas, como en el caso del *earliest deadline first*.

A partir de ellos se han desarrollado gran cantidad de trabajos adecuando estos algoritmos a situaciones más reales (véase por ejemplo [SLR86], [LSS87], [GS88], [SSL89], [SRL90] o [BMS93]).

El objetivo de la etapa de planificación será garantizar, mediante la adecuada asignación de prioridades u otros medios, que todos los requisitos de tiempo real sean cumplidos. En nuestro caso, la plataforma de ejecución será monoprocesador, y el lenguaje Ada 95, por lo que adoptaremos el sistema de prioridades de Ada 95 (una descripción detallada del sistema de prioridades de Ada 95, si se utiliza el anexo de sistemas de tiempo real, puede consultarse en el Ada Reference Manual [ARM94], anexo D).

Existen trabajos en la literatura que tratan la planificación de redes de Petri, pero se centran en la planificación estática de redes temporizadas. Se pueden consultar, por ejemplo [CCG83, CC85, CC88], [GWN91], [BCMP92], [OSW93], [WY93], [CFGZ93], [TYC95]. Sin embargo no se conocen trabajos en el ámbito de las redes de Petri con tiempo.

No es objetivo de esta tesis presentar una técnica de planificación por asignación de prioridades basada en redes de Petri con tiempo. Esto es otra línea de investigación que bien puede ser materia de una o varias tesis. Hasta el momento, proponemos la utilización de métodos heurísticos basados las técnicas existentes de planificación por prioridades y que hemos enumerado previamente. El resultado de esta etapa será la asignación de una prioridad estática a cada transición CODE de la red.

2.4. Análisis

Con las prioridades asignadas puede realizarse un análisis del sistema, para comprobar si se cumplen todos los requisitos temporales. Los métodos de análisis de redes de Petri extendidas con tiempo están basados principalmente en técnicas enumerativas relacionadas con el cálculo del grafo de alcanzabilidad. Por ejemplo, en [CC88] se presenta el *Earliest State Graph* para el análisis de redes de Petri temporizadas. La construcción de un grafo de estados en redes de Petri con tiempo es más complicado, ya que los intervalos temporales hacen que el número de estados alcanzables (estado e instante en el que se alcanza) sea muy grande, e incluso infinito en tiempo continuo. El problema se ha solucionado mediante discretización temporal [Pop91] o por medio de la definición de clases de estados y la construcción de un grafo de clases de estados, *State Class Graph* [BD91].

Como muestra de una técnica de análisis consideraremos el caso de la figura 2.9. En ella se muestra un proceso periódico ($T = 10$ u.t.) que está integrado dentro de un sistema con más procesos de prioridad superior, merced a los cuales, en cierto momento, el sistema es desalojado de la CPU durante 6 u.t. El propósito del análisis es determinar si los procesos del sistema cumplen sus restricciones temporales, y, en concreto, si el proceso modelado termina su ejecución antes de la activación del siguiente período. Al generar el grafo de clases de estado es preciso tener en cuenta las características de la plataforma de implementación: número de procesadores, prioridades asociadas a las transiciones, etc.

Para la verificación del cumplimiento de los plazos de ejecución el modelo de partida es modificado, añadiéndole un lugar implícito (p_M), que facilitará la detección de marcados que reflejen

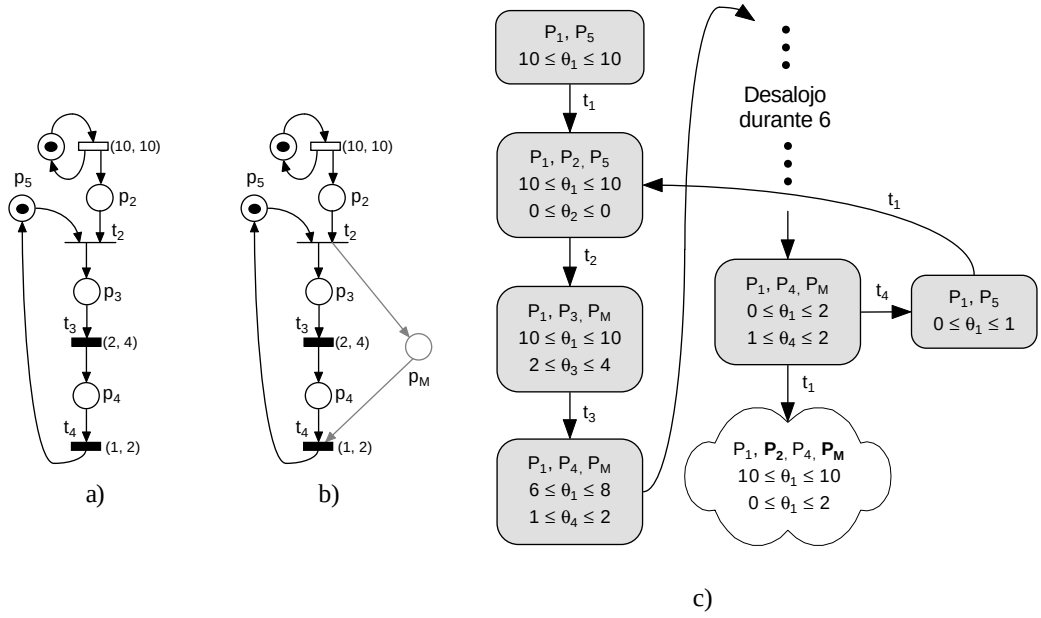


Figura 2.9: a) RdPT que modela un proceso periódico; b) Un lugar implícito se añade para facilitar el análisis; c) Grafo de clases de estados de la red así modificada.

estados erróneos. Según se comprueba fácilmente, el lugar p_M permanece marcado mientras el proceso ejecuta el código de las transiciones t_3 y t_4. El marcado simultáneo de los lugares p_M y p_2 será síntoma de que la ejecución del código no ha terminado antes de la activación de la siguiente ejecución periódica, por lo que las restricciones temporales del proceso no se cumplen. En el grafo de clases de estado de la figura puede comprobarse que uno de los estados muestra esta posibilidad, por lo que puede concluirse que el sistema no es ejecutable.

2.5. La implementación. Aspectos generales

Como último paso en el desarrollo se encuentra la *implementación*. Una implementación programada o "software"de una RdP es un programa o algoritmo que simula el disparo de las transiciones de la red (en función de las marcas disponibles), respetando las reglas de evolución del modelo. Conviene matizar esta definición para distinguir el término del de *simulación*. Cuando el objeto de esta etapa sea la generación de un producto (programa, sistema de control, etc.) se habla de implementación. Sin embargo, a veces solamente interesa estudiar el comportamiento del sistema para determinar sus propiedades de buen funcionamiento, por ejemplo durante un período de tiempo demasiado largo inviable en una fase de pruebas. En este caso nos estaremos refiriendo a la simulación.

En la definición se ha hecho referencia a las reglas de evolución del modelo, siendo éste uno de los aspectos más importantes de toda implementación: definir una *regla de disparo* que respete las reglas de la evolución del marcado de la red y, en conclusión, del estado del sistema modelado. Una formulación básica de esta regla puede ser de la siguiente manera:

1. Establecimiento de la sensibilización de una transición, evaluando su *condición de sensibilización*, definida por el marcado de los lugares de entrada (ver anexo A) y en la que pueden influir las interpretaciones añadidas a la red.

2. Disparo de las transiciones sensibilizadas (en esto influye la interpretación de la red).

3. Actualización del marcado, retirando marcas de los lugares de entrada y depositándolas en los de salida.

Por tanto, la función básica de toda implementación consiste en hacer evolucionar el estado de la red, y en consecuencia las interpretaciones de ésta influyen muy directamente en los algoritmos de implementación (ver sección 2.1), reduciendo el indeterminismo implícito en el segundo punto de la anterior regla de disparo.

2.5.1. Nuestra política de ejecución. Política de resolución de conflictos

Antes de diseñar o elegir una técnica de implementación debe haberse fijado la semántica interpretativa, la política de resolución de conflictos, y en general todos los aspectos que permiten determinar con precisión qué transición disparar en cada momento. A esto se le llama genéricamente *política de ejecución*. Si bien la semántica interpretativa ya ha sido definida en la sección 2.1, no lo hemos hecho todavía con la política de resolución de conflictos, que, en nuestro caso, deberá ser coherente con la interpretación aplicada en la etapa de modelado, merced a la cual hemos distinguido entre tres tipos de transiciones en función de su papel en el sistema. Así mismo, la interpretación de tiempo de sensibilización también tendrá su influencia en esta política, ya que es posible que varias transiciones TIME y/o una CODE se encuentren sensibilizadas, y por tanto en ejecución, simultáneamente. Habitualmente esta semántica va asociada en la implementación a una *política de carrera*: la transición que antes complete su ejecución será la disparada.

En síntesis, nuestra política de disparo se fundamentará en los siguientes puntos (coherentes con la interpretación del formalismo de redes de Petri elegido en la sección 2.1.5):

1. Semántica de servidor único (single server), no considerando la reentrancia de código (de hecho, éste ha sido uno de los requisitos que hemos exigido en nuestros modelos).

2. Disparo inmediato de transiciones SYCO, en cuanto se detecte su sensibilización

3. Comienzo inmediato de la ejecución del código asociado a las transiciones CODE y establecimiento inmediato del instante de disparo de las TIME (instante de vencimiento del evento asociado).

4. Semántica de tiempo de sensibilización.

Política de resolución de conflictos

Supongamos una situación de conflicto efectivo en la que aparezcan transiciones de los tres tipos comentados (CODE, TIME y SYCO), con las restricciones reseñadas en 2.2.4. Para la resolución del conflicto seguiremos la siguiente política, coherente con la del punto anterior y con el hecho de que las transiciones SYCO y TIME son responsables del modelado de los aspectos de control del sistema, que deben tener preferencia ante los aspectos operativos, representados por las transiciones CODE:

1. En primer lugar se atenderá al disparo de transiciones SYCO, que debe ser instantáneo. Si hubiese varias sensibilizadas se elegirá la más prioritaria. En caso de empate, se elegirá cualquiera de ellas.

2. Si no existiese ninguna transición SYCO sensibilizada, se comenzará a considerar el disparo de las transiciones CODE y TIME (se comenzará la ejecución del código y se considerarán los eventos temporales asociados). Se establecerá una política de carrera entre las transiciones de ambas clases presentes en el conflicto, de modo que la que antes termine será disparada (tiempo de sensibilización menor). Si varias acabasen al mismo tiempo, se atenderá el disparo de la TIME más prioritaria. La CODE será la última en ser considerada.

3. Notar que, por 1, si durante el tiempo de sensibilización de alguna de estas transiciones se detecta la sensibilización de alguna transición SYCO, ésta será la disparada. Esto supondrá el aborto de la ejecución del código y de los eventos asociados al resto de transiciones. Hemos de advertir que en esta tesis no se tratará sobre la recuperación del sistema después del aborto de código no finalizado.

2.5.2. Tipos de implementaciones

Se expondrán en esta sección algunos criterios de clasificación de las implementaciones de RdP.

Implementación Interpretada/Compilada

En la implementación *interpretada*, o *interpretación dirigida por la red*, tanto la estructura de la red como su marcado se codifica en estructuras de datos de las cuales uno o varios programas extraen la información necesaria para hacer evolucionar la red. Estos programas, que denominaremos *intérpretes*, son independientes de la RdP que se implementa. Para implementar una RdP sólo es necesario generar la estructura de datos que la representa. Esto hace que la técnica sea especialmente adecuada en etapas de prototipado, cuando el diseño final no está del todo definido, y se necesita flexibilidad para realizar cambios. Sobre esta técnica volveremos en capítulo 3.

La segunda forma, implementación *compilada* o *compilación dirigida por la red*, consiste en la generación de uno o varios programas, de forma que el comportamiento dinámico de la red esté implícito en su flujo de control. En este tipo de implementaciones es posible representar una marca que evoluciona por dentro de la RdP mediante el contador de programa de un proceso secuencial, mientras que el resto del marcado puede ser implementado con variables y "buffers"de comunicación, según se expondrá en los siguientes capítulos.

Mientras que las implementaciones interpretadas disminuyen el tiempo de generación de la implementación (no hay que generar el programa intérprete) y la ocupación de memoria en grandes redes, las homólogas compiladas presentan menor tiempo de ejecución. No debe deducirse que la interpretación y la compilación son dos opciones excluyentes. Puede pensarse en esquemas híbridos donde parte de la red se implemente de forma interpretada y el resto de forma compilada.

Implementaciones Secuencial/Concurrente

Una RdP es un formalismo que puede representar *concurrencia*, esto es, la posibilidad de ejecución simultánea. Sin embargo, las reglas de evolución no obligan a que se produzca dicha simultaneidad, de hecho, la concurrencia en el disparo de transiciones a veces se interpreta como entrelazado ("interleaving"[CMP87] [Rei88]) que es una secuencialización de la ejecución. Desde este punto de vista, puede pensarse en implementaciones secuenciales construidas con *procedimientos*. Este tipo de implementación ha sido muy utilizada, por ejemplo en [SV82] y [VCBA83] en el ámbito de la programación de automatismos lógicos. En este tipo de aplicaciones el código asociado a las transiciones es mínimo, por lo que la serialización no supone un gran inconveniente en la práctica.

En una implementación secuencial el orden de disparo de las transiciones concurrentes tiene gran importancia desde el punto de vista del comportamiento del modelo. En [GJBL88] se presentan

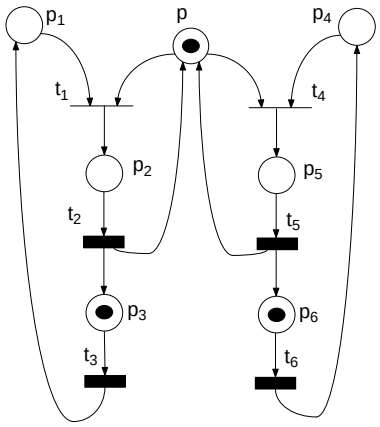


Figura 2.10: Si en un principio se dispara t_3, con la regla "Depth-first"no se llega a disparar t_6

posibles reglas para resolver esta serialización. Además de la regla de elección aleatoria se plantean otras dos de gran interés:

1. *Depth-first*: se considera en primer lugar para el siguiente disparo el conjunto de transiciones sensibilizadas por las marcas producidas en el último disparo.

2. *Breadth-first*: no se consideran nuevas sensibilizaciones hasta que se han disparado todas las transiciones sensibilizadas por un cierto marcado.

El ordenamiento que producen estas reglas es completado mediante la asignación de ciertas prioridades a las transiciones. Como puede pensarse, el comportamiento de una implementación es bien diferente si se atiende a una u otra regla. La primera de ellas hace evolucionar parte de la red mientras que el resto espera, pudiendo llegar a darse problemas de monopolio si existen circuitos en la red (figura 2.10). Con la regla "Breadth-first"se consigue una evolución homogénea de la red.

Una implementación secuencial supone establecer una restricción suplementaria sobre el modelo. Aunque en RdP en las que se consideren disparos instantáneos o cuasi-instantáneos dicha restricción puede llegar a ser irrelevante, deja de serlo cuando existe un código asociado al disparo de las transiciones (esto es, "cuestaçierto tiempo). En estos casos es importante la concurrencia en la ejecución del código asociado a las transiciones. Considérese el caso de la figura 2.11 donde una secuencialización puede suponer el retraso en la evolución de una parte de la red.

Una implementación concurrente se puede construir mediante *tareas* o procesos secuenciales que comunican entre sí, ya sea mediante "buffers"(*comunicación asíncrona*) o mediante "rendezvous"(*comunicación síncrona*). Dentro de la categoría de implementaciones concurrentes se incluyen también aquellas que se realizan sobre un único procesador pero en tiempo compartido. En este caso, que denominaremos pseudo-concurrente, se produce también entrelazamiento. Pero a diferencia de una implementación secuencial, el entrelazamiento se produce en la ejecución del código asociado a las transiciones, dejando de ser la transición una unidad indivisible. Así, el procesador se reparte entre todas las transiciones activas, evitándose problemas de esperas. El reparto del tiempo del procesador lo suele hacer el núcleo del sistema operativo o el núcleo del lenguaje concurrente sobre el que se realiza la implementación.

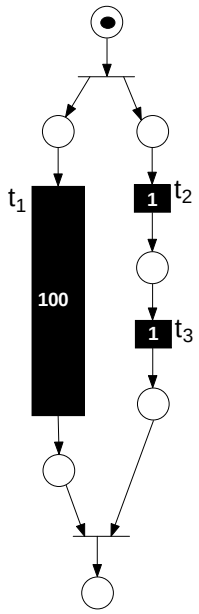


Figura 2.11: En una implementación secuencial, si se dispara en primer lugar t_1, la ejecución de la rama derecha se retrasa 100 unidades

Implementaciones Centralizada/Descentralizada

Esta clasificación de las implementaciones hace referencia a cómo se realiza el algoritmo de control de la red, esto es, el algoritmo encargado de establecer qué transiciones son disparables y de actualizar el marcado cuando un disparo se produce. En una implementación centralizada, el algoritmo de control se implementa como una única unidad de programa que actúa sobre la totalidad de la red. En una implementación descentralizada este algoritmo se distribuye en varias tareas, cada una de ellas encargada de hacer evolucionar una cierta parte de la red localmente.

Tradicionalmente [CSV86] se ha estructurado el esquema de implementación centralizado de la siguiente manera (ésta será la base de la arquitectura utilizada en el capítulo 3):

- Un conjunto de tareas, posiblemente una por transición, que ejecutan el código asociado a las transiciones.
- Una tarea especial, llamada *coordinador*, que ejecuta el algoritmo de control de la red en su totalidad.

El coordinador será el encargado de activar la ejecución del resto de las tareas, esto es, de establecer qué transiciones son disparables y ordenar su disparo. Cuando una tarea finaliza su actividad, informa al coordinador que seguirá ejecutando el algoritmo de control. El coordinador es una tarea activa que puede ser vista como el núcleo de un sistema multitarea. Proporciona una sincronización indirecta entre las actividades de las tareas asociadas al disparo de transiciones.

Básicamente las ventajas de una implementación centralizada son su simplicidad y la facilidad para modificar el esquema de sincronización/comunicación sin alterar el código asociado a las transiciones. De entre sus problemas cabe reseñar que un fallo en el coordinador puede ser catastrófico.

En una implementación centralizada concurrente existe una clara separación entre la *parte de control* (PC), realizada por el coordinador, y la *parte operativa* (PO), compuesta por las tareas

que ejecutan las actividades asociadas a las transiciones. Esta separación puede mantenerse en una implementación descentralizada, manteniendo el conjunto de tareas que ejecutan el código de las transiciones y distribuyendo el control entre varios procesos *sincronizadores locales* [CSV86] que controlan cada uno una parte de la red.

Sin embargo, para nosotros serán más interesantes las implementaciones descentralizadas en las que cada una de las tareas en las que se descompone el algoritmo de control actúa sobre una subred secuencial [CSV86]. En este caso no existirá concurrencia entre las transiciones y por lo tanto el código asociado a ellas y la parte correspondiente de control pueden agruparse en una tarea secuencial. Este esquema de implementación se construye pues, mediante un conjunto de tareas con PC y PO que se comunican directamente entre sí mediante "buffers"(posiblemente representando un lugar de la red) o "rendezvous". Ésta será la base de la técnica aplicada en el capítulo 4.

Implementaciones Síncrona/Asíncrona

Una implementación *síncrona* basa la ejecución de la red en pasos ("steps"[GSW80]). Esto es, dispara todas las transiciones concurrentemente disparables a partir de un marcado M y no se plantea un nuevo disparo hasta que no se ha alcanzado el marcado sucesor de M. Existe, pues, sincronía tanto en el disparo de transiciones como en la actualización del marcado. Desde el punto de vista de las redes temporizadas, el tiempo transcurrido entre dos "pasosçonsecutivos lo marcará la transición disparada con un tiempo asociado mayor. Una formalización de esta regla de disparo así como un estudio de las propiedades derivadas se recoge en [IC87].

En una implementación *asíncrona*, cuando el disparo de cualquier transición ha concluido se actualiza el marcado, pudiendo derivarse de esta actualización nuevos disparos sin espera alguna.

El soporte de una implementación

El soporte de una implementación es el conjunto de dispositivos que la hacen posible su elaboración y ejecución. Distinguiremos entre soporte físico (procesadores, memorias, etc.) y soporte lógico (sistema operativo y el lenguaje de implementación).

Soporte físico El soporte físico de una implementación puede ser mono o multiprocesador. Sólo si la arquitectura del sistema tiene un número de procesadores igual o superior a la concurrencia de una red[4], ésta podrá ser implementada sin ningún tipo de restricción, esto es, el paralelismo modelado será efectivo. Si esto no sucede (el número de procesadores no es suficiente) la implementación deberá construirse mediante serialización. Esta serialización puede ser total, por ejemplo una implementación secuencial en monoprocesador, o parcial, un procesador puede ser utilizado por diversas tareas en tiempo compartido (implementación pseudo-concurrente). Desde otro punto de vista, el soporte físico puede ser de propósito general (especializado a través del "software") o de arquitectura especializada, donde parte del control se realiza mediante "hardware"[Sil85].

Soporte lógico El lenguaje de implementación es aquél en el cual es programada la red en implementaciones compiladas, o bien el intérprete de la estructura de la red en implementaciones interpretadas. Puede ser de alto o bajo nivel. Otro aspecto a tener en cuenta es si el lenguaje permite la programación de situaciones concurrente. En implementaciones concurrentes, los lenguajes concurrentes aportan primitivas específicas para el control de la concurrencia, mientras que los lenguajes secuenciales deben ser extendidos con primitivas de un sistema operativo multitarea.

Soporte a utilizar en esta tesis En la presente tesis utilizaremos un lenguaje concurrente de alto nivel, *Ada 95* [ARM94, Rat95, Bar95, BW98], para la realización de las implementaciones. El

[4]Denominaremos concurrencia de una red al número máximo de disparos simultáneos que pueden ocurrir

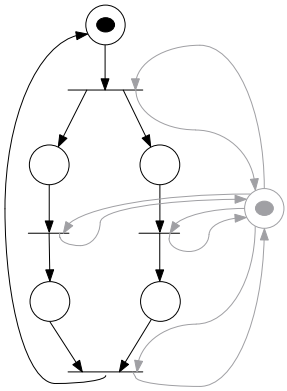


Figura 2.12: Representación de una implementación secuencial, modelo de implementación

lenguaje Ada 95, posee un conjunto de potentes primitivas para controlar la concurrencia y los aspectos de gestión del tiempo necesarias para la generación de código para sistemas de tiempo real. Estas primitivas tienen una semántica bien definida, y son altamente independientes de la plataforma de ejecución. Además, consideraremos el empleo del anexo de Tiempo Real, que matiza el comportamiento de las citadas primitivas. En cuanto a la plataforma hardware, sólo consideraremos *sistemas monoprocesador*. No se descarta la ampliación posterior de las técnicas para la distribución de las implementaciones en varios procesadores.

2.5.3. Criterios de calidad

En esta sección se van a presentar un conjunto de criterios (tanto cualitativos como cuantitativos) que permitirán evaluar la calidad de una implementación.

Corrección y completitud

Una implementación de una RdP se dice *correcta* si el lenguaje de disparo de las transiciones en la implementación está incluido en el lenguaje de disparo de la RdP. Dicho de otra forma, toda traza de disparos de transiciones en la implementación debe estar incluida en las trazas de disparo de la RdP.

Relacionado con el concepto de corrección está el de *completitud* de una implementación. Diremos que una implementación de una RdP es completa si el lenguaje de disparos que genera es coincidente con el de la RdP. Una implementación, en general, introduce restricciones adicionales al disparo de transiciones reduciendo el lenguaje de disparo o las trazas posibles (Implementación NO COMPLETA), pero nunca puede relajar restricciones explícitas de la RdP que provoquen trazas prohibidas (Implementación INCORRECTA).

Las restricciones adicionales impuestas por una implementación pueden, en la mayoría de los casos, ser descritas en términos de redes de Petri (lugares y transiciones), dando como resultado otra RdP que llamaremos *modelo de implementación*. Es el caso, por ejemplo, del modelado de la plataforma de implementación monoprocesador, o el de una implementación secuencial. En este tipo de implementaciones no existe concurrencia, la implementación impone restricciones adicionales al disparo de transiciones de tal forma que todas las transiciones de una red estén en exclusión mutua. Estas restricciones pueden ser modeladas mediante un lugar monomarcado que es precondición y postcondición de todas las transiciones (figura 2.12). La secuencialización sólo supone una restricción cuando las transiciones tienen asociado un tiempo de disparo (piénsese en el

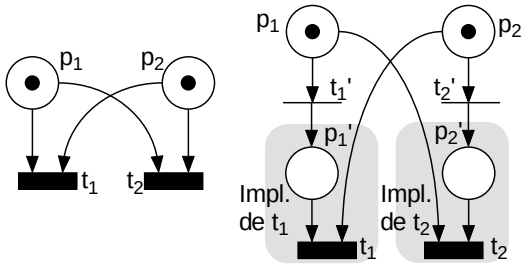


Figura 2.13: Implementación bloqueante: (a) SubRdP; (b) Modelo de implementación

entrelazamiento). En este caso, una implementación secuencial impide la aparición de las trazas de disparo con concurrencia real. Sin embargo, en el modelo teórico nada obliga a que las transiciones se ejecuten en concurrencia, sólo se recoge la posibilidad de que esto ocurra. Una implementación secuencial de una RdP no introduce incorrección, si bien es generalmente incompleta.

Bloqueos, inanición

Diremos que una implementación de una RdP presenta un *bloqueo* cuando la RdP es viva y sin embargo el modelo de implementación no lo es.

A continuación se ilustra cómo una implementación puede introducir un bloqueo en una RdP viva. Supóngase la subRdP de la figura 2.13.a. Las transiciones t_1 y t_2 se implementan como dos procesos con una parte de control y una parte operativa que es el código asociado a las transiciones. Los lugares pueden considerarse como semáforos. Si en la parte de control de las transiciones se serializa el acceso a los lugares, como puede observarse en el código que sigue, de tal forma que la transición t_1 accede primero a P_1 para desmarcarlo y luego al P_2, mientras que la transición t_2, en paralelo, accede a los lugares en sentido inverso, se produce un bloqueo. La implementación descrita se modela en la Figura 2.13.b añadiendo a la subRdP los lugares P_1' y P_2' y las transiciones t_1' y t_2' que son de control. El bloqueo se producirá, por ejemplo, si, disparando t_1' en primer lugar, t_2' es disparada antes que t_1.

```
task t1;                    task t2;
task body t1 is             task body t2 is
begin                       begin
  P1.desmarcar;               P2.desmarcar;
  P2.desmarcar;               P1.desmarcar;
  ...                         ...
  -- código asociado          -- código asociado
  ...                         ...
end;                        end;
```

La existencia de un bloqueo en una implementación provoca incompletitud, pero no incorrección, ya que únicamente impide la ejecución de ciertas ramas de la traza, pero no crea otras.

Desde un punto de vista más conceptual, la interpretación que se hace de la red para implementar puede provocar la no ejecución de subredes cuya activación depende de la resolución de un conflicto. Diremos en este caso que las transiciones de la subred sufren de *inanición* en la implementación. De todas formas el problema de la inanición se reduce a la existencia de un bloqueo parcial en el modelo de implementación. El problema de inanición puede surgir por varias razones:

1. *Método de resolución de conflictos.* El caso más trivial en el que se da inanición es el de asociar prioridades para la resolución de conflictos en RdP. En un conflicto siempre disparará aquella que tenga la prioridad más alta (figura 2.14.a). La transición t_3 de la figura

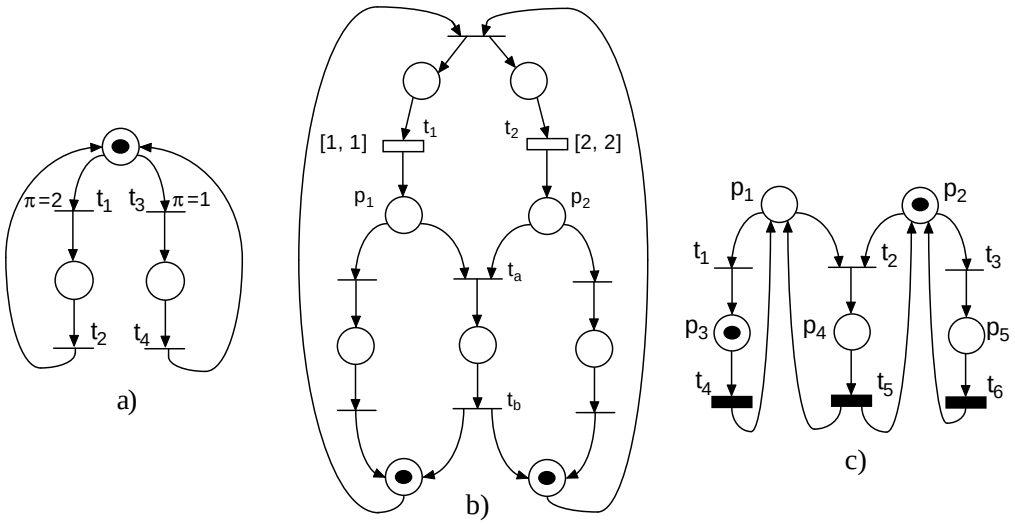


Figura 2.14: a) Las transiciones t_3 y t_4 sufren de inanición por el método de resolución de conflictos; b) Las transiciones t_a y t_b sufren de inanición debido a los tiempos de disparo de t_1 y t_2; c) t_2 nunca se dispara si la estrategia de disparo es la síncrona

nunca disparará debido a que siempre está sensibilizada a la vez que t_1 y su prioridad es menor, por lo tanto t_3 y t_4 sufren de inanición.

2. *Tiempos de ejecución de las transiciones.* En la figura 2.14.b se muestra un ejemplo donde los tiempos de disparo de las transiciones impiden la activación de una subred. Dado que las transiciones SYCO se disparan inmediatamente, los lugares p_1 y p_2 nunca estarán marcados simultáneamente por lo tanto t_a (y por lo tanto t_b) nunca estará sensibilizada (t_a y t_b sufren de inanición).

3. *Estrategia de disparo.* Para ilustrar este punto veremos cómo una implementación síncrona (la estrategia de disparo en una implementación síncrona ha sido presentada en la sección 2.5.2) puede provocar inanición. En la Figura 2.14.c se presenta un ejemplo. La interpretación síncrona de la RdP de la figura 2.14.c hace que t_3 y t_4 "disparen a la vez", sincronizadamente. Lo mismo ocurre con t_1 y t_6. Se observa fácilmente que p_1 y p_2 están en exclusión mutua por lo tanto t_2 nunca se sensibilizará produciéndose un bloqueo parcial en el modelo de implementación.

Generalidad y eficiencia

La *generalidad* de un método de implementación hace referencia a la aplicabilidad de éste a distintos tipos de redes. Así un tipo de implementación específico para RdP binarias con una representación del marcado basado en variables booleanas no será válida para RdP generalizadas, mientras que un tipo de implementación con la representación del marcado basado en contadores es válido para ambos tipos de redes, es más general.

La generalidad de un tipo de implementación depende del máximo grado de complejidad de las redes que pueda implementar. Así una implementación específica para redes ordinarias puede implementar así mismo redes de menor complejidad (simples, de libre elección, etc.).

Dado un método de implementación de una cierta generalidad, el aumento de complejidad de la red suele suponer una degradación del resto de las propiedades. Cuando la complejidad de la red supera a la generalidad del método se suele comprometer incluso la corrección. Por otra parte,

dado un tipo de red, el aumento de la generalidad del método supone una degradación de las prestaciones.

Prestaciones

A continuación se presentan aquellos aspectos de una implementación que mayor incidencia tienen en las prestaciones temporales.

(a) *Habilidad.* Se define la habilidad de una implementación [Vil90] como el inverso del número medio de transiciones sensibilizadas por unidad de tiempo partido por el número medio de transiciones que han sido analizadas para establecer su sensibilización por unidad de tiempo. Este ratio da una medida del número de test innecesarios que se realizan en una implementación. Haremos mención a la habilidad en el capítulo 3, dónde la mejora de este parámetro será el factor diferenciador entre las distintas técnicas presentadas.

(b) *Tráfico.* Se define el tráfico existente en una implementación [Vil90] como el número medio de comunicaciones realizadas entre procesos por transición disparada (similar a la "ocurrence complexity"[Tau88]). Las comunicaciones, en general, suponen un gran coste para cualquier implementación, de ahí, la importancia de disminuir al máximo el tráfico manteniendo la concurrencia de la implementación. El tráfico depende de la implementación y, en general, de la red implementada. Hay que hacer notar que no todas las comunicaciones tienen el mismo coste. Así hablaremos de diferentes costes en la comunicación si los procesos que comunican, lo hacen asíncronamente o síncronamente; comparten el mismo procesador o no; comunican directamente o a través de elementos intermedios; etc. Será importante, pues, usar un tráfico ponderado por el coste de cada comunicación.

(c) *Número de procesos.* Evidentemente, en lo que respecta a las prestaciones, es muy importante el número de procesadores de que se dispone (y la arquitectura del sistema) que darán la concurrencia real de la implementación. Sólo si la arquitectura del sistema soporte tiene suficiente número de procesadores, la concurrencia potencial de una implementación (número de procesos) puede llegar a ser efectiva. En caso contrario los procesadores deben ser compartidos por varios procesos y se hace importante *minimizar el número de procesos* que componen la implementación (sin comprometer la concurrencia potencial de la RdP), así como realizar una buena distribución de la carga en los procesadores.

Cuando un procesador trabaja en tiempo compartido es interesante analizar la influencia que tiene el tipo de los procesos en las prestaciones de la implementación. Si la implementación está construida con procesos cíclicos de bucle infinito habrá que tomar en consideración el coste de los cambios de contexto que se producen. Si por el contrario los procesos no son cíclicos sino que "comienzanz "acaban", al coste de los cambios de contexto habrá que añadir el de la creación de los procesos.

La concurrencia real (número de procesadores) de una implementación influye en la importancia del tráfico en las prestaciones temporales. Así, en una implementación con concurrencia real se pueden establecer comunicaciones en paralelo que, desde el punto de vista de las prestaciones, pueden contabilizarse como una sola [Tau88].

2.6. Resumen de resultados y conclusiones. Líneas abiertas

En este capítulo se han repasado las distintas etapas del ciclo de vida un sistema de tiempo real, enfocándolas desde la óptica de la utilización de las redes de Petri. La simplicidad y gran número de situaciones que pueden ser modeladas por las redes de Petri con Tiempo [BD91] han sido argumento suficiente para elegirlas como formalismo a utilizar en la tesis, pese a que es preciso

completarlas mediante la asociación de predicados y prioridades a las transiciones. Así, mediante la construcción de un modelo formal, la semántica del sistema queda especificada sin ambigüedad ninguna.

Se ha mostrado cómo las RdPT pueden ser utilizadas para el modelado de muchas situaciones propias de los sistemas de tiempo real y se ha hecho un análisis comparativo con otros formalismos evidenciándose un menor poder expresivo que las RdPT. Se han presentado dos ejemplos en los que se pueden encontrar procesos periódicos, timeouts, sincronizaciones, eventos (tanto internos como externos), etc., comprobando que todas estas situaciones son cubiertas por las RdPT.

Tras ello se ha mostrado cómo es posible abordar el análisis de las propiedades del sistema, estudiando para ello propiedades de la red de Petri que lo modela (análisis del grafo de clases de estado). En este punto nos hemos limitado a mostrar cuales son las ideas principales, sin entrar en detalles. Ésta es precisamente una de las líneas de trabajo que se esperan ampliar en un futuro.

Para terminar, se han repasado los conceptos básicos relacionados con la implementación de redes de Petri, clasificando las técnicas, e incidiendo en la política de ejecución y en la de resolución de conflictos. Los protocolos descritos en estos puntos serán determinantes en la fase de generación de código. Así mismo se han presentado parámetros de calidad que permiten la evaluación y la comparación de técnicas de implementación.

Como conclusión principal, por tanto, puede decirse que las RdPT son válidas para su uso durante todo el ciclo de vida de los sistemas de tiempo real. Su utilización conlleva varias ventajas: en primer lugar se trata de un formalismo fácil de entender debido a su naturaleza gráfica; permite la verificación y validación de la corrección del sistema; las técnicas de análisis de las RdPT incrementarán la flexibilidad del diseño, pues ya no será necesario imponer restricciones al sistema para poder analizar su comportamiento ni para verificar el cumplimiento de sus restricciones temporales; además el formalismo es ejecutable, lo que permite el prototipado y la simulación de sistemas, y la generación de código puede ser automatizada, reduciendo de este modo los errores de codificación.

Presentado el marco general, dedicaremos la tesis a la fase de implementación (generación de código), utilizando técnicas que hasta el momento han sido aplicadas a formalismos de redes de Petri que no consideraban el tiempo y que modelaban sistemas de propósito general.

Capítulo 3

Implementación centralizada

3.1. Introducción

Este capítulo se dedicará a la aplicación de las técnicas clásicas de implementación centralizada al caso de las redes de Petri con Tiempo, para su utilización en la realización de sistemas de tiempo real.

El capítulo recoge los resultados presentados en el trabajo [GV96].

Antecedentes El capítulo está basado en trabajos clásicos que tratan sobre implementación centralizada de redes de Petri, como [Sil85], [CSV86] y, sobre todo [Vil90], y trabajos más modernos como [Bri95]. En este último se hace un magnífico repaso de las técnicas existentes, y se propone un método de generalización de lo que hasta entonces habían sido técnicas de implementación de redes binarias, ampliándolas para la implementación centralizada de redes no binarias, y con arcos con pesos e inhibidores.

Objetivos Se analizarán las técnicas, así como su funcionamiento e impacto en las prestaciones de la implementación, para concluir que su aplicabilidad debe circunscribirse a las etapas de diseño y prototipado, en las que la técnica aporta la necesaria flexibilidad con que se debe contar para realizar cambios en el sistema. Como ya hemos avanzado, utilizaremos el lenguaje de programación Ada 95 para la codificación de la implementación, debido a sus características de concurrencia y de tiempo real.

Estructura del capítulo El capítulo comenzará exponiendo los fundamentos de la implementación centralizada, esto es, la división de la parte de control y operativa del sistema, presentando la estructura simplificada de sus elementos básicos: el *coordinador*, encargado del control y la supervisión temporal, y los *procesos-CODE*, a los que corresponde la parte operativa. Dedicaremos la segunda sección al coordinador, presentándolo como un intérprete, y mostrando sus detalles y los elementos y características especiales de los que debe disponer para implementar la extensión de redes de Petri que se está utilizando, las RdPT. Pasaremos seguidamente a analizar cómo mejorar la *habilidad* [Vil90] del coordinador, mediante la aplicación de dos métodos ya conocidos, pero que deben ser adaptados al caso de las RdPT. Éstos son los *lugares representantes marcados* y las *transiciones sensibilizadas*. Tras ello, dedicaremos una sección al análisis del impacto de la presencia del coordinador en la prestaciones del sistema implementado, centrando el estudio en la reducción del factor de utilización del procesador. Previamente, habremos mostrado cuál es el funcionamiento detallado del coordinador, con el fin de justificar la reducción de prestaciones a la

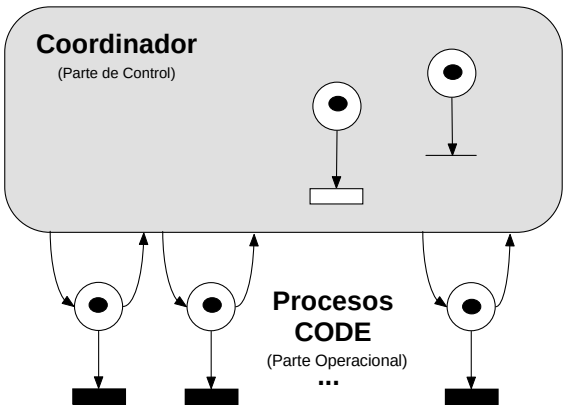


Figura 3.1: Estructura de una implementación centralizada

que aludíamos. Por último se expondrán unas conclusiones sobre las ventajas e inconvenientes de esta técnica de implementación en el ámbito de los sistemas de tiempo real.

3.2. Fundamentos de la implementación centralizada

Según la clásica arquitectura de una implementación centralizada [CSV86], ésta se basa en el empleo de dos tipos de procesos:

- Cada unidad de ejecución compuesta por una transición CODE junto con sus lugares de entrada, que representa la ejecución de una cierta actividad en el sistema (ejecución de código), será implementada en un proceso (tarea en Ada 95), al que llamaremos *proceso-CODE*, que será concurrente con los demás de la implementación. De esta forma se preserva en la implementación la concurrencia propia de la parte operativa del sistema. Cada proceso-CODE ejecutará el código asociado a la transición CODE que le corresponda.

- Un proceso *coordinador*. El coordinador se encarga del control de la red, incluidos los aspectos temporales. Cada proceso-CODE se comunica con el coordinador, que es el responsable de tomar la decisión sobre cuándo las acciones asociadas a una transición deben comenzar, o sobre cuándo ésta debe ser disparada. Se tratará del proceso más prioritario de la implementación, ya que es el encargado de supervisar las acciones de control y de temporización, que deben ser ejecutadas inmediatamente.

Podemos asimilar el coordinador al núcleo de un sistema operativo multitarea, y los procesos-CODE con los procesos ejecutados y gestionados por él (la estructura descrita se representa en la figura 3.1). Como consecuencia de esta arquitectura, las partes de control y operativa del sistema son separadas, siendo competencia del coordinador la primera de ellas, y la segunda de los procesos-CODE.

3.2.1. Estructura básica de un *proceso-CODE*

Básicamente, el proceso-CODE será una tarea Ada 95, que deberá ejecutar el código asociado a la transición CODE que representa.

La ejecución del código debe comenzar cuando la transición CODE esté sensibilizada, condición que es determinada por el coordinador. Hay dos formas de proceder a la activación del código:

- A través de la activación dinámica del proceso-CODE. Aunque hemos renunciado a la reentrancia en el código, esta forma de proceder permitiría la ejecución de varias instancias del mismo proceso-CODE, si el grado de sensibilización de la transición (ver anexo A) fuera mayor que uno.
- Mediante la sincronización del comienzo del proceso-CODE por medio de una comunicación (`Comienzo_CODE`) con el coordinador, que será aceptada por éste cuando determine que la transición está sensibilizada. Del mismo modo, al acabar la ejecución, el coordinador debe ser informado de ello, para que se produzca el disparo de la transición y, como consecuencia, la adecuada actualización del marcado. Esta notificación se realizará mediante otra comunicación (`Fin_CODE`) que será aceptada por el coordinador. Es necesario que el proceso-CODE se identifique frente al coordinador en estas comunicaciones, por lo que deberá ser consciente de su identidad (ver el discriminante `Ident` en el algoritmo 3.2.1).

Debido a que la ejecución del código de una transición CODE puede ser abortado por el disparo de otra transición, de tipo TIME o SYCO, con la que esté en conflicto, así también se deberá poder abortar el código del proceso-CODE. Este aborto asíncrono puede abordarse de dos maneras:

1. Se puede abortar totalmente el proceso, lo que obligará a volver a crearlo de nuevo inmediatamente después (esto supone activación dinámica de tareas).
2. O se puede utilizar una primitiva de codificación que permita el aborto asíncrono, como la transferencia asíncrona de control de Ada 95 (ATC). En este último caso el aborto se producirá mediante una comunicación (`Aborta_CODE`) que será aceptada por el coordinador.

Basado en las segundas opciones de las últimas enumeraciones, en el algoritmo 3.2.1 se muestra la estructura del código, en Ada 95, de un proceso-CODE.

Algoritmo 3.2.1 Estructura de un proceso-CODE

```
task type procesoCODE (Ident: Transiciones_CODE); -- Identificador del proceso-CODE
task body procesoCODE is
begin
   loop
     Coordinador.Comienzo_CODE( Ident );
     select
       Coordinador.Aborta_CODE( Ident );
     then abort
       Código( Ident );
       Coordinador.Fin_CODE( Ident );
     end select;
   end loop;
end;
```

3.2.2. Arquitectura básica del *coordinador*

El algoritmo de control de la red, ejecutado por el coordinador, se basará en la ejecución continuada de un bucle sin fin, al que denominaremos *ciclo de tratamiento*. En él básicamente se desarrollarán las acciones de control y supervisión temporal de la ejecución de los procesos-CODE.

En nuestro caso, la implementación será síncrona (ver sección 2.5.2) lo que supone la ejecución de la red en pasos (*steps*). La ejecución de un paso supone el disparo de todas las transiciones sensibilizadas para un marcado M dado, no procediéndose a nuevos disparos hasta que no se alcanza un marcado sucesor de M. Por ello el algoritmo de control se ha dividido en dos fases:

1. *Fase de análisis de la sensibilización.* En ella se determinará la sensibilización de las transiciones, que depende del estado de la red, descrito por el marcado de los lugares de la misma y de los predicados y prioridad eventualmente asociados a las transiciones. Si se determina que una transición está sensibilizada, se procederá al comienzo de la ejecución de sus acciones asociadas. Más adelante detallaremos concretamente las acciones a realizar.

2. *Fase de disparo de transiciones.* En la segunda fase se realizarán los disparos de las transiciones, procediéndose a la ulterior actualización del marcado. Esto generará un nuevo marcado que será tenido en cuenta en la siguiente ejecución del ciclo de tratamiento, que continúa de forma cíclica.

3.3. El coordinador

Como avanzamos en la sección anterior, toda implementación centralizada está basada en un proceso coordinador cuya función es ejecutar la red. Para ello repite un ciclo de tratamiento en el que se promueven las acciones de control (transiciones SYCO), supervisión temporal (TIME) y ejecución de código (CODE). Analicemos seguidamente los detalles de la ejecución de este ciclo, así como algún elemento software del que se debe disponer para la implementación de sistemas de tiempo real.

3.3.1. Fase de análisis de la sensibilización

En la fase de análisis de la sensibilización, se promoverá el disparo de las transiciones SYCO sensibilizadas (único tipo de transición de disparo inmediato), se calcularán los instantes de disparo de las transiciones TIME, y se comenzará la ejecución de los procesos-CODE, asociados a las transiciones CODE. Todo ello debe realizarse respetando la política de resolución de conflictos enunciada con anterioridad (sección 2.5.1), que prima el disparo de las transiciones SYCO frente a las transiciones CODE y/o TIME. Por ello:

1. En primer lugar se realizará el estudio de la sensibilización de la transiciones SYCO. Si se encuentran varias SYCO en conflicto, se elegirá una de ellas atendiendo a las prioridades asociadas en el modelo.

2. Las transiciones de tipo SYCO se comenzarán a disparar inmediatamente, desmarcando sus lugares de entrada. De este modo, se desensibilizarán el resto de transiciones en conflicto con ella, evitando que sean consideradas candidatas al disparo.

3. Tras el comienzo del disparo de las SYCO, se tomarán en consideración el resto de transiciones.

4. En caso de que la transición sea de tipo TIME, se anotará su instante de disparo en una *lista de eventos* gestionada por el coordinador. Sobre esta lista de eventos comentaremos algo seguidamente.

5. Si la transición es CODE, se comenzará la ejecución del código asociado, activando la ejecución del proceso-CODE adecuado mediante la aceptación de la comunicación `Comienzo_CODE`.

3.3.2. Fase de disparo de transiciones

Una vez iniciado el disparo de las transiciones SYCO, registrados los eventos temporales, e iniciada la ejecución de los procesos-CODE oportunos, el coordinador entra en la fase de disparo de transiciones. En ella, el coordinador se suspende esperando la finalización de las actividades de

las transiciones CODE o TIME representada la ocurrencia de algún evento de tipo temporal o de finalización de código. Por ello, en esta fase se realizarán la siguientes acciones:

1. Finalizará el disparo de las transiciones SYCO, que se había comenzado en la fase anterior con el fin de evitar que otras transiciones en conflicto fueran disparadas.
2. Se esperará al vencimiento de los eventos temporales de las transiciones TIME (mediante expresiones `delay`)
3. Se esperará la finalización de los procesos-CODE, mediante la aceptación de la comunicación `Fin_CODE`.
4. La situación es más compleja si la transición CODE está en conflicto con otra TIME. Como se especificó en su momento, se establecerá una política de carrera entre las transiciones CODE y TIME para determinar qué transición es la que realmente se dispara. Si el evento registrado para la transición TIME vence antes de que el código termine, el proceso-CODE será abortado, (en lo que es una representación de un *timeout*). El caso contrario también es posible.

La fase de disparo puede simplificarse si se procede de la siguiente manera: el fin del disparo de una transición SYCO (pto. 1), se realizará mediante la inserción en la lista de eventos de un evento temporal de vencimiento inmediato. De esta forma el mismo esquema utilizado para el tratamiento de las transiciones TIME servirá para las SYCO.

Para la codificación software, se deberá utilizar una primitiva Ada 95 de tipo `select`, que permita al mismo tiempo la espera de una comunicación y el vencimiento de una plazo de tiempo (esto es, una estructura `select` con alternativa `delay`). Esta fase deberá seguir activa mientras haya procesos-CODE que quieran notificar su finalización, o, en la lista de eventos, haya algún evento pendiente que venza en el instante actual. Por todo ello, la estructura básica será:

```
loop
  select
    accept Fin_CODE (Tr : in Transiciones_CODE) do -- Disparo de la transición CODE
      Actualizar Marcado( Tr );
    end Fin_CODE;
  or
    when not Lista_Eventos.Vacia =>
      delay until Tiempo_Evento_Mas_Urgente;
      loop
        Actualizar Marcado(Transicion_del_Evento_Mas_Urgente);
        Lista_Eventos.Pop_Evento;
        exit when Lista_Eventos.Vacia or else Tiempo_Evento_Mas_Urgente > CLOCK;
      end loop;
  end select;
  exit when Fin_CODE'COUNT = 0;
end loop;
```

3.3.3. La lista de eventos temporales

El coordinador, para realizar la supervisión temporal impuesta por las transiciones TIME de la red, cuenta con una lista[1] de eventos temporales donde se registrarán los instantes de vencimiento de los eventos (tiempos de disparo) y la transición a la que corresponden.

[1] Nada se dice sobre la forma de implementar esta lista (por ejemplo con una lista de prioridad o montículo). Incluso puede no ser una lista en sentido estricto.

Se debe disponer, además de métodos para insertar o eliminar eventos en la lista y comprobar si está vacía, de operaciones para consultar la siguiente información:

- `Tiempo_Evento_Mas_Urgente`, o instante del evento más próximo a cumplirse.
- `Transicion_del_Evento_Mas_Urgente`, que devolverá la transición a la que corresponde el instante `Tiempo_Evento_Mas_Urgente`.
- `Tipo_Evento_Urgente`, que indicará el tipo de transición que corresponde al evento de vencimiento más cercano. Esta información es utilizada por el coordinador en la segunda fase del ciclo de tratamiento, para determinar si la transición a disparar es de tipo TIME o SYCO (recuérdese que el tratamiento de ambos tipos se había uniformizado a través de la lista de eventos, pero, en el momento del disparo, debe diferenciarse).

La inserción de los eventos en la lista se producirá de forma ordenada, por lo que, en el caso de que dos transiciones TIME deban ser disparadas al mismo tiempo, la operación de inserción de eventos temporales deberá tener en cuenta la prioridad de las transiciones, para insertar antes el más prioritario.

3.3.4. Evitación de la deriva acumulativa

Una implementación centralizada de una red de Petri con tiempo puede provocar la aparición de deriva acumulativa [BW97]. Debido a la propia ejecución del coordinador (consultar sección 3.6, existe una diferencia temporal entre el instante en el que una transición TIME es sensibilizada (en la fase de disparo) y el instante en el que el coordinador determina esa sensibilización y registra el evento en la lista de eventos (en la fase de estudio de la sensibilización).

El problema se ha solucionado utilizando una variable de tipo `Ada.Real_Time.Time`, a la que denominaremos `T_Ultimo_Disparo`. La variable almacenará el instante en el que se haya producido la última actualización de marcado. El cambio de valor de esta variable se realizará en la fase de disparo. Cada vez que una transición CODE sea disparada, la variable tomará el valor del `CLOCK` del sistema. Si la transición disparada es de tipo TIME, el nuevo valor de `T_Ultimo_Disparo` será el valor del evento asociado. El disparo de una transición SYCO no deberá modificar la variable, ya que su disparo es instantáneo. El valor de `T_Ultimo_Disparo` será tenido en cuenta para el cálculo de los siguientes instantes de vencimiento de transiciones TIME. Haciendo esto puede evitarse la utilización de marcas con sello temporal (*timestamp*), como en [Aal93], [AO95] o [FGP93].

3.3.5. Estructuras de datos y actualización del marcado

Al haber sido construido como un intérprete, el coordinador deberá disponer de estructuras de datos que representen la estructura de la red a ejecutar y su marcado. De esta forma, podrá determinar la sensibilización de las transiciones, y, tras el disparo de las mismas, proceder a la actualización del marcado de los lugares. Así mismo deberá contar con información temporal sobre los intervalos de tiempo asociados a las transiciones TIME, para poder registrarlos en la lista de eventos.

La actualización del marcado no es trivial si como descendientes de un lugar existen varias transiciones en conflicto, sobre todo si éstas son de tipo TIME o CODE. Los lugares de entrada de este tipo de transiciones están marcados mientras sus acciones asociadas están siendo desarrolladas, permaneciendo en este estado hasta que éstas finalizan. El disparo de una transición, y el consiguiente desmarcado de lugares de entrada, desensibiliza las transiciones de salida, lo que debe provocar la interrupción de las acciones que corresponden a las transiciones CODE y TIME de salida. La interrupción del evento temporal de la transición TIME se producirá extrayendo el evento de la lista de eventos, mientras que en el caso de una transición CODE se producirá mediante la

interrupción de la ejecución proceso-CODE que la implementa, por ejemplo habilitando la aceptación de la entrada `Aborta_CODE` que, a su vez, activará la alternativa de disparo de la estructura ATC del proceso-CODE (ver sección 3.2.1). Para que el coordinador pueda acometer estas acciones deberá disponer de una estructura de datos que refleje qué transiciones deben ser abortadas tras el desmarcado de cada lugar (pueden consultarse, por ejemplo, las estructuras propuestas en [Vil90]).

3.3.6. Algoritmo básico del coordinador

Basándonos en todos los comentarios realizados hasta el momento, proponemos el algoritmo 3.3.1, en pseudo-Ada 95, para el proceso coordinador. En él puede apreciarse el ciclo de tratamiento con las dos fases descritas, así como la utilización de la variable `T_Ultimo_Disparo` para la evitación de la deriva acumulativa.

3.4. Mejora de las prestaciones del coordinador de una implementación centralizada

En la fase de determinación de la sensibilización de transiciones del algoritmo 3.3.1 se propone un recorrido sistemático por todas las transiciones de la red. Evidentemente, la carga computacional de este recorrido es grande, sobre todo cuando el número de transiciones es considerable. Por ello han aparecido mejoras ([Sil85] [CSV86]) basadas en la utilización de una lista de transiciones parcial o totalmente sensibilizadas, a la que se ha dado en llamar *lista de tratamiento*, $\mathcal{LT}$. De este modo, la búsqueda de transiciones disparables se circunscribe a las almacenadas en $\mathcal{LT}$. Como la implementación es síncrona, las transiciones que van quedando parcial o totalmente sensibilizadas como resultado de los disparos de las transiciones de $\mathcal{LT}$, deberán ser almacenadas en otra *lista en formación* $\mathcal{LF}$. Como en nuestra interpretación de las RdPT hemos distinguido tres tipos de transiciones, del mismo modo, en nuestra implementación utilizaremos tres pares de listas, uno para cada tipo de transiciones. Al final del ciclo de tratamiento, la lista de tratamiento $\mathcal{LT}$, que habrá quedado vacía, debe ser actualizada con las transiciones contenidas en $\mathcal{LF}$ ($\mathcal{LT} := \mathcal{LF}$). Además, como se indica en [Bri95], esta estrategia permite la gestión de la resolución de los conflictos, si la inserción de las transiciones en las listas se hace de acuerdo a su prioridad.

La discusión debe centrarse ahora en qué criterio se utilizará para introducir las transiciones en las listas de tratamiento. De ello nos ocuparemos en las siguientes subsecciones.

3.4.1. Método de los lugares representantes marcados (LRMs)

En esta técnica [Sil85], la búsqueda de transiciones sensibilizadas es mejorada mediante una preselección de ciertos lugares de entrada, que se denominarán *lugares representantes*. La técnica se basa en los siguientes puntos:

- Cada transición se representa mediante uno de sus lugares de entrada, su *lugar representante*.
- Sólo las transiciones cuyo lugar representante esté marcado son consideradas candidatas al disparo, y el test de sensibilización se realizará sólo sobre ellas. De esta manera se mejorará la habilidad de la implementación.
- El resto de lugares de entrada, que no sean representantes, se llamarán *lugares de sincronización*. Serán consultados durante la determinación de la condición de sensibilización de una transición cuyo lugar representante esté marcado.

De esto se deduce que, con vistas a la implementación, es preciso realizar una elección de los lugares representantes de las transiciones de una red. Esto determinará una clasificación del

Algoritmo 3.3.1 Algoritmo básico del coordinador de una implementación centralizada

```
task body Coordinador is
   – declaración de variables y subprogramas
begin
   T_Ultimo_Disparo := CLOCK;
   loop
     -- Fase de análisis de la sensibilización
     for Tr in Transic_SYCO'RANGE loop -- Estudio de las transic. SYCO y comienzo de su disparo
       if Sensibilizada(Tr) then
         Resolver posible conflicto(Tr) -- Si Tr está en conflicto con otras devolverá la transición a disparar
         Desmarcar_Lugares_Entrada(Tr);
         Lista_Eventos.Registra_Evento(Tr, T_Ultimo_Disparo);
       end if;
     end loop;

     for Tr in Transic_TIME'RANGE loop -- Estudio de las transic. TIME y registro de los eventos temporales
       if Sensibilizada(Tr) and not Ya_Registrada(Tr) then
         Lista_Eventos.Registra_Evento(Tr, T_Ultimo_Disparo + Tiempo_Disparo(Tr));
       end if;
     end loop;

     for Tr in Transic_CODE'RANGE loop -- Estudio de las transic. CODE e inicio de su ejecución
       if Sensibilizada(Tr) and not En_Ejecucion(Tr) then
         accept Comienzo_CODE(Tr);
       end if;
     end loop;

     -- Fase de disparo de transiciones
     loop
       select
         accept Fin_CODE (Tr : in Transic_CODE) do -- Disparo de la transición CODE
           T_Ultimo_Disparo := CLOCK;
           Actualizar Marcado( Tr );
         end Fin_CODE;
       or
         when not Lista_Eventos.Vacia =>
           delay until Tiempo_Evento_Mas_Urgente;
           loop
             T_Ultimo_Disparo := Tiempo_Evento_Mas_Urgente;
             if Evento_Urgente_Es_TIME then  -- Disparo de transición TIME
               Actualizar Marcado(Transicion_del_Evento_Mas_Urgente);
             else -- Finalizar disparo de transición SYCO
               Marcar_Lugares_Salida(Transicion_del_Evento_Mas_Urgente);
             else
             Lista_Eventos.Pop_Evento;
             exit when Lista_Eventos.Vacia or else Tiempo_Evento_Mas_Urgente > CLOCK;
           end loop;
       end select;
       exit when Fin_CODE'COUNT = 0;
     end loop;
   end loop;
   end Coordinador;
```

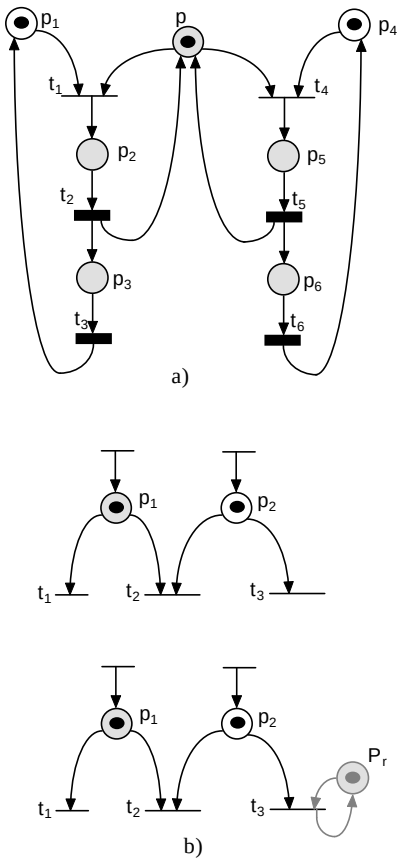


Figura 3.2: a) Una posible clasificación de los lugares de una red según la técnica LRMs: los lugares sombreados son los representantes de las transiciones de la red y los no sombreados los de sincronización; b) Si se desea clasificación disjunta, a veces es preciso añadir lugares implícitos: el lugar p_1 representa a t_2 y p_2 su lugar de sincronización, por lo que p_2 no puede ser representante de t_3. Por eso se añade el implícito p_r que representa a t_3.

conjunto de lugares de la misma en dos conjuntos P_R (*lugares representantes*) y P_S (*lugares de sincronización*). Un ejemplo se muestra en la figura 3.2.a. A diferencia de [Sil85] y [Vil90], en [Bri95] se indica que esta clasificación no tiene por qué necesariamente ser disjunta, es decir, puede ocurrir que $P_R \cap P_S \neq \emptyset$. Éste será el enfoque que adoptemos aquí. Volveremos sobre ello un poco más adelante. En otra serie de referencias, [Val86][Bar87][Bak90][VB90], también se distingue entre los lugares representantes (*places pivot*), con partición disjunta, y los lugares de disparo (*triggering places* o *places de déclenchement*), con clasificación no disjunta.

Peculiaridades del ciclo de tratamiento en LRMs

En el algoritmo LRMs, las listas de tratamiento, $\mathcal{LT}$, almacenarán las transiciones cuyo lugar representante esté marcado al comienzo del ciclo de tratamiento, y la listas en formación, $\mathcal{LF}$, las transiciones cuyo lugar representante estará marcado en el próximo ciclo de tratamiento. Es preciso remarcar que las listas de tratamiento contienen únicamente transiciones candidatas a ser disparadas, ya que un lugar representante caracteriza parcialmente la sensibilización de una transición. Para determinarla completamente es preciso consultar todos sus lugares de entrada.

El coordinador implementado según la técnica LRMs escrutará las listas de tratamiento, evaluando la sensibilización de las transiciones en ella contenidas. Se procesará primero la lista de transiciones SYCO, y luego la de TIME y CODE. Si una vez extraída de la $\mathcal{LT}$, se determina que una transición está sensibilizada, se procederá a realizar las acciones descritas en la fase de análisis de la sensibilización, esto es, desmarcar los lugares de entrada de una transición SYCO, registrar el evento temporal de una TIME o aceptar el comienzo de la ejecución si es CODE. Si la transición extraída de $\mathcal{LT}$ no está sensibilizada, pero su lugar representante sigue marcado, la transición debe volver a ser introducida en la lista $\mathcal{LF}$. Una vez agotada la $\mathcal{LT}$, se pasará a la fase de disparo de transiciones, donde se actualizará el marcado de los lugares. En este último paso se incluirán en la listas en formación, $\mathcal{LF}$, aquellas transiciones cuyo lugar representante haya sido marcado en la operación (actualización síncrona del marcado). Al final del ciclo de tratamiento, la listas de tratamiento $\mathcal{LT}$, que han quedado vacías, deben ser actualizadas con las transiciones contenidas en $\mathcal{LF}$ ($\mathcal{LT} := \mathcal{LF}$).

A pesar de no ser la opción de implementación elegida, comentaremos por su interés la técnica descrita en [Sil85] y [Vil90]. Aquí el esquema es distinto: las listas $\mathcal{LF}$ y $\mathcal{LT}$ contienen lugares representantes marcados. Por ello el coordinador, que irá extrayendo los lugares representantes de la listas $\mathcal{LT}$, debe disponer de la estructura de transiciones descendientes de cada lugar representante. Tras el disparo de una transición, se deberán introducir en la lista $\mathcal{LF}$ los lugares representantes marcados como consecuencia del disparo, así como aquellos representantes que permanezcan marcados porque ninguna de sus transiciones de salida se haya disparado. Por otra parte, en la técnica de los lugares de disparo [Val86], la actualización de marcado es asíncrona, utilizándose una única lista en la que se van insertando lugares de disparo según se van marcando.

Resolución de conflictos mediante LRMs

Trataremos aquí sobre conflictos entre transiciones SYCO, pues, como ya se ha mencionado, entre otro tipos de transiciones se establece una política de carrera (ver sección 2.5.1). Se elegirá la transición más prioritaria de entre las sensibilizadas. Esta política de resolución puede implantarse de dos maneras:

- En la línea de [Vil90] y [Sil85], se puede añadir una estructura de datos adicional que indique qué transiciones están en conflicto con una en particular, de modo que la sensibilización de una transición SYCO se estudia en conjunto con aquellas transiciones con las que se encuentra en conflicto efectivo. Determinada qué transición será la disparada, es preciso eliminar de la lista de tratamiento todas aquellas que estuvieran en conflicto con la disparada y que, tras el disparo de ésta, dejen de estar sensibilizadas. El coste computacional de esta serie de operaciones, hace que sea preferible dejar en la lista de tratamiento las transiciones a eliminar, y desecharlas cuando se produzca el estudio de su sensibilización. Si la red es simple, y se elige el lugar común como representante de un conjunto de transiciones en conflicto simple, la resolución del conflicto se simplifica en gran medida [Vil90].

- La política puede estar implícita en el propio ciclo de tratamiento, como se indica en [Bri95], donde se propone la inserción de las transiciones en $\mathcal{LT}$ y $\mathcal{LF}$ según su prioridad. De esta manera, las más prioritarias serán tenidas en cuenta antes, y, en caso de ser disparadas impedirán el disparo del resto de las del conflicto, ya que los lugares de entrada comunes serán desmarcados.

Elección de lugares representantes

Está claro que la bondad del método de LRMs dependerá en gran medida de la determinación de los lugares representantes de las transiciones. En la literatura se encuentran diferentes aproximaciones que tratan este problema.

En [Sil85] (sección 9.6.3) y [Vil90] (sección 5.3.4) lo que se pretende es reducir la duración del ciclo de coordinador, para lo que se propone que cada lugar represente el mayor número de transiciones posible. Como consecuencia, se priman las representaciones con menor número de lugares representantes. Por ejemplo, en el caso de redes simples, el lugar común de un conflicto será elegido como representante de todas sus transiciones de salida, lo que, además, simplifica la resolución de los conflictos. Así mismo, en estos trabajos se supone que la clasificación de los lugares de la red es disjunta ($P_R \cap P_S \neq \emptyset$). Según [Sil85], si la clasificación no fuese disjunta, se precisaría de una doble representación para los lugares que fuesen simultáneamente representantes y de sincronización, lo que supondría una peor utilización de la memoria, además de una sobrecarga en el algoritmo, que debe gestionar esta doble representación. Además, al utilizarse listas de tratamiento y en formación compuestas de lugares, si la clasificación no fuera disjunta, no se podría garantizar que un lugar representante p_k para el cual no se ha disparado ninguna de sus transiciones representadas en un ciclo del coordinador, siga marcado en el siguiente ciclo, por lo que debería tomarse precauciones para su inclusión en la lista de formación; esto es, el disparo posterior de alguna transición que tenga a p_k como lugar de sincronización podría desmarcarlo, lo que obligaría a eliminarlo de la lista en formación. Por último, se indica que la partición disjunta de lugares en redes no simples puede no ser posible, por lo que se propone la adición de implícitos para conseguirla (ver figura 3.2.b).[2]

En la línea anterior, en [Vil90] se propone una variante del método de LRMs, denominada *implementación dirigida por conflictos*, en la que se construyen representantes de un conjunto de transiciones en conflicto, no necesariamente simple, mediante la adición de un lugar implícito resultante de la suma de los lugares de entrada de las transiciones en conflicto. Como consecuencia, cada conflicto tendrá su lugar representante, por lo que el conflicto se convierte en la unidad de implementación. Por construcción, el implícito añadido no será binario, y los arcos que lo unen con sus transiciones de salida pueden tener pesos distintos de 1, por lo que la red modificada no será binaria ni ordinaria.

En [Val86][Bak90][VB90] se propone escoger como lugares de disparo los marcados menos frecuentemente, ya que así se reduce el número de transiciones candidatas. En [Bar87] y [Bak90] se propone una optimización de la técnica, basada en el empleo de lugares de disparo dinámicos (*places de déclenchement dynamiques*), según la cual los lugares de disparo se van escogiendo dinámicamente tras el estudio de la sensibilización de las transiciones. En este sentido, si una transición candidata a ser disparada no puede serlo a causa de que alguno de sus lugares de entrada no la sensibiliza, se escoge como próximo lugar de disparo uno cualquiera de entre los de entrada que no la sensibilizan. Según [Bak90] esta optimización no mejora apreciablemente las prestaciones de la implementación.

En [Bri95] el problema se aborda desde otro punto de vista. Se introduce la figura del lugar *representante ideal de una transición, lri*, que cumple que, para cualquier marcado alcanzable, la transición está sensibilizada sii el *lri* está marcado. Si no es posible obtener un lugar que verifique esta condición, se propone que la cumpla para el mayor número de marcados alcanzables. De esta manera se mejorará la habilidad de la técnica. Para un estudio más profundo sobre como realizar la elección de lugares representantes ideales referimos a [Bri95].

Estructuras de datos necesarias en LRMs

Para la implementación del algoritmo se necesitarán las siguientes estructuras de datos (supondremos que la red es ordinaria):

- Una representación del marcado, posiblemente mediante un vector que indique el número de marcas de cada lugar.

[2] Notar que estos problemas provocados por la utilización de una partición disjunta de lugares P_R y P_S se eliminan si se adopta la aproximación de [Bri95], en la que las listas de tratamiento se forman con transiciones

- Para cada transición de la red será necesario conocer:
  - el tipo de transición (SYCO, TIME, CODE),
  - su lugar representante,
  - una lista de sus lugares de sincronización,
  - las características temporales de la transición, dependiendo de su tipo,
  - una lista de lugares representantes marcados tras su disparo,
  - una lista de lugares de sincronización marcados tras su disparo, y
  - una lista de transiciones en conflicto abortables por su disparo.
- Para cada lugar, la lista de transiciones a las que representa.

El algoritmo LRMs

El algoritmo del coordinador de una implementación centralizada según la técnica de los lugares representantes marcados, se recoge en el algoritmo 3.4.1. En él sólo se especifican las peculiaridades de la primera fase respecto al algoritmo 3.3.1, pues la segunda fase, de disparo de transiciones es idéntica.

3.4.2. Método de las transiciones sensibilizadas (TS)

Para mejorar la habilidad de la implementación en la primera fase del ciclo de tratamiento, el método de las transiciones sensibilizadas [SV82, Sil85] propone considerar únicamente el disparo de transiciones completamente sensibilizadas. Para ello, en redes binarias, a cada transición se le asocia un contador, $E(i)$, que registra el número de sus lugares de entrada no marcados. De esta forma, el test de sensibilización se reduce a una única comparación, $E(i) = 0$. En redes no binarias, con pesos y/o arcos inhibidores, la determinación de la sensibilización resulta más complicada. En [BC93] y [Bri95] se generaliza el método de las TS a través de la utilización de funciones lineales de sensibilización (LEFs) y funciones lineales de sensibilización parcial (PLEFs).

Peculiaridades del ciclo de tratamiento en TS

En el método de las transiciones sensibilizadas las listas de tratamiento, $\mathcal{LT}$, contienen únicamente transiciones totalmente sensibilizadas. El marcado de la red no tiene porque ser explícito, ya que añadir o quitar marcas de los lugares supone simples decrementos o incrementos, respectivamente, de los contadores de las transiciones de las que el lugar es salida o entrada.

Como en LRMs, el ciclo comenzará examinando la lista $\mathcal{LT}$ de transiciones SYCO. Como es sabido, si una transición SYCO está sensibilizada, su disparo debe comenzar instantáneamente, retirando las marcas de sus lugares de entrada. Esto supone que su contador $E(i)$ y el de todas las transiciones con las que comparte algún lugar de entrada deberá ser actualizado inmediatamente (si la red es binaria, el nuevo valor será el del número de lugares de entrada). De esta forma, se evitará que dos transiciones en conflicto se disparen en el mismo ciclo de tratamiento. Esta estrategia, lleva implícita otra consecuencia. A pesar de haberse dicho que la lista $\mathcal{LT}$ contiene transiciones totalmente sensibilizadas, esto será cierto únicamente al comienzo del ciclo de tratamiento, pues durante su desarrollo, el disparo de alguna transición puede desensibilizar a otras con las que esté en conflicto. Por ello sigue siendo necesario realizar el test de sensibilización.

Para las transiciones TIME y CODE, se actuará de forma análoga, comprobando el valor de su contador, y si éste es cero, se procederá, al registro del evento temporal o a la aceptación del inicio del proceso-CODE, según corresponda.

En la fase de fin de disparo, se procederá a la actualización (decremento) de los contadores de las transiciones descendentes de la disparada. Por ello es necesario que el coordinador disponga de

Algoritmo 3.4.1 Algoritmo básico del coordinador según la técnica de LRMs

```
task body Coordinador is
   – declaración de variables y subprogramas
begin
   T_Ultimo_Disparo := CLOCK;
   loop
     −− Fase de análisis de la sensibilización
     while not LT_SYCO.Vacia loop −− Transic. SYCO
       if Sensibilizada(Tr) then
         Resolver posible conflicto(Tr) −− Devolverá la transición que haya que disparar
         Desmarcar_Lugares_Entrada(Tr);
         Lista_Eventos.Registra_Evento(Tr, T_Ultimo_Disparo);
       elsif Representante(Tr).Marcado then
         LF_SYCO.Insertar(Tr);
       end if;
       LT_SYCO.Quitar;
     end loop;

     while not LT_TIME.Vacia loop −− Transic. TIME
       if Sensibilizada(Tr) then
         Lista_Eventos.Registra_Evento(Tr, T_Ultimo_Disparo + Tiempo_Disparo(Tr));
       elsif Representante(Tr).Marcado then
         LF_TIME.Insertar(Tr);
       end if;
       LT_TIME.Quitar;
     end loop;

     while not LT_CODE.Vacia loop −− Transic. CODE
       if Sensibilizada(Tr) then
         accept Comienzo_CODE(Tr);
       elsif Representante(Tr).Marcado then
         LF_CODE.Insertar(Tr);
       end if;
       LT_CODE.Quitar;
     end loop;

     −− Fase de disparo de transiciones
           ...
       −− Tal como se indica en el algoritmo 3.3.1
           ...

     −− Actualización de las listas de tratamiento
     LT_SYCO := LF_SYCO;
     LT_TIME := LF_TIME;
     LT_CODE := LF_CODE;
   end loop;
   end Coordinador;
```

una estructura de datos que muestre qué transiciones son descendientes de una dada (ver, p.ej., la propuesta de [Vil90]). Se incluirán en la lista en formación $\mathcal{LF}$, aquellas transiciones que queden totalmente sensibilizadas tras el disparo de alguna de las del paso anterior.

Resolución de conflictos mediante TS

La técnica no añade ninguna novedad sobre el tema, por lo que la resolución de conflictos se realizará como se indica en la sección anterior (sobre LRMS).

Estructuras de datos necesarias en TS

Para cada transición t_i de la red se necesitarán los siguientes datos (supondremos que la red es binaria):

- el tipo de transición (SYCO, TIME, CODE),
- las características temporales de la transición, dependiendo de su tipo,
- el valor de su contador de sensibilización $E(t_i)$,
- el número de lugares de entrada, o, de otra manera, el número de marcas necesarias para su disparo,
- una lista de transiciones en conflicto abortables por su disparo, junto con el número de lugares que comparte con cada una de esas transiciones,
- una lista de transiciones descendentes cuya sensibilización se ve afectada por su disparo, junto con el número de lugares que marca de cada una de esas transiciones.

El algoritmo TS

El algoritmo 3.4.2 corresponde al del coordinador de una implementación centralizada según la técnica de las transiciones sensibilizadas. En él, respecto al algoritmo 3.3.1, sólo se especifican las peculiaridades de la primera fase, pues la segunda, de disparo de transiciones, es idéntica.

3.5. Influencia de la planificación de una RdPT en su implementación centralizada

Durante la etapa de planificación se habrá asignado una prioridad estática a cada transición de la RdPT. En la implementación centralizada, estas prioridades serán tenidas en cuenta de dos maneras:

- Para asignar la prioridad de los procesos-CODE, prioridades que serán utilizadas por el núcleo de Ada 95, para planificar la ejecución de estos procesos.
- Por el coordinador, para resolver los conflictos entre transiciones SYCO o TIME.

En cualquier caso el coordinador será el proceso más prioritario de toda a implementación, pues de él depende el control y supervisión del resto de la misma.

Algoritmo 3.4.2 Algoritmo básico del coordinador según la técnica de TS

```
task body Coordinador is
   – declaración de variables y subprogramas
begin
   T_Ultimo_Disparo := CLOCK;
   loop
     -- Fase de análisis de la sensibilización
     while not LT_SYCO.Vacia loop -- Transic. SYCO
       if E(Tr) = 0 then
         Resolver posible conflicto(Tr) -- Devolverá la transición que haya que disparar
         Desmarcar_Lugares_Entrada(Tr);
         Lista_Eventos.Registra_Evento(Tr, T_Ultimo_Disparo);
       end if;
       LT_SYCO.Quitar;
     end loop;

     while not LT_TIME.Vacia loop -- Transic. TIME
       if E(Tr) = 0 then
         Lista_Eventos.Registra_Evento(Tr, T_Ultimo_Disparo + Tiempo_Disparo(Tr));
       end if;
       LT_TIME.Quitar;
     end loop;

     while not LT_CODE.Vacia loop -- Transic. CODE
       if E(Tr) = 0 then
         accept Comienzo_CODE(Tr);
       end if;
       LT_CODE.Quitar;
     end loop;

     -- Fase de disparo de transiciones
           ...
       -- Tal como se indica en el algoritmo 3.3.1
           ...

     -- Actualización de las listas de tratamiento
     LT_SYCO := LF_SYCO;
     LT_TIME := LF_TIME;
     LT_CODE := LF_CODE;
   end loop;
   end Coordinador;
```

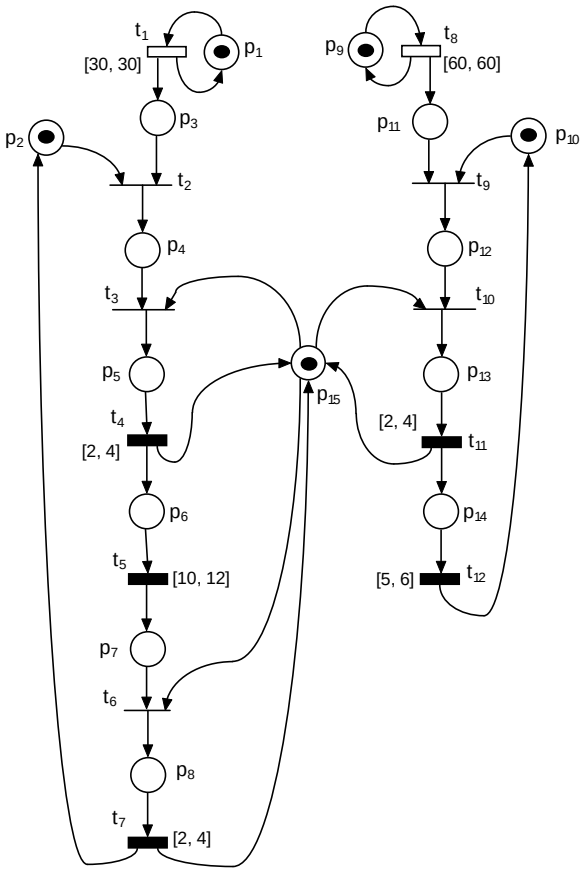


Figura 3.3: Red de Petri con tiempo para mostrar el funcionamiento de la implementación centralizada

3.6. Funcionamiento de la implementación centralizada

Con el fin de analizar las prestaciones de la implementación centralizada de redes de Petri con tiempo, consideremos el análisis de la ejecución de la implementación de la red de la figura 3.3. En ella se muestran dos procesos periódicos, de período 30 y 60 unidades de tiempo respectivamente, y que ejecutan una serie de acciones (transiciones t_3, t_4, t_5, t_7, t_{11} y t_{12}), utilizando un recurso común (modelado mediante el lugar p_{15}). Cada 30 unidades de tiempo la transición t_1 es disparada, marcando el lugar p_3, lo que promueve las acciones del primer proceso (análogamente cada 60 u.t. la transición t_8 se dispara, marcando p_{11}, y activando la ejecución del otro proceso). Supondremos que las transiciones del primer proceso (t_4, t_5 y t_7) tienen más prioridad que las del otro.

En la figura 3.4 se muestra el cronograma de la ejecución de la implementación, que básicamente pasa por las siguientes etapas:

1. Al comenzar la ejecución, el coordinador detecta la sensibilización de las dos transiciones TIME, por lo que inserta dos eventos en la lista de eventos. La ejecución se suspende, en el `delay` de la segunda fase del coordinador, hasta que vence el tiempo del primero de los eventos almacenados.

2. En este momento (30 u.t.) la transición t_1 es disparada, insertando un nuevo evento, para el instante 60, en la lista de eventos. El coordinador actualiza el marcado y dispara t_2 y t_3,

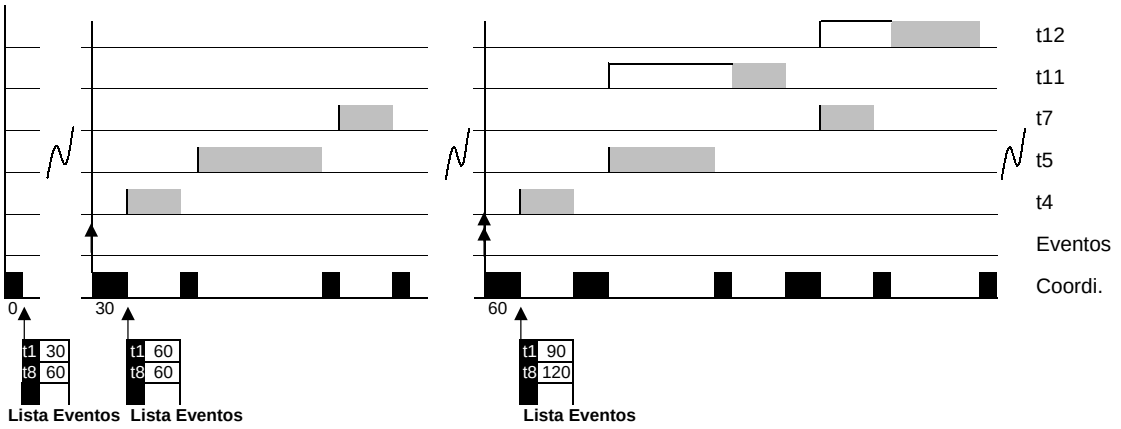


Figura 3.4: Cronograma de funcionamiento de la implementación centralizada de la red de la figura 3.3

tras lo que acepta el comienzo de la transición t_4. El proceso-CODE t_4 se ejecutará entonces, mientras el coordinador espera su finalización. Cuando está se produce, la comunicación de fin de ejecución es aceptada, lo que hace que el proceso-CODE de la transición t_5 se ejecute. Tras él, le tocará el turno al de la transición t_7. Tras esta primera ejecución del primer proceso, que tardará un mínimo de 14 u.t. y un máximo de 20 u.t., sin contar el tiempo de ejecución del coordinador, el coordinador se suspende hasta el instante del siguiente evento en la lista de eventos (60 u.t.).

3. Ahora se disparan dos transiciones, t_1 y t_8, insertando nuevos eventos, actualizando el marcado y llegando a la situación en la que los lugares p_4, p_{12} y p_{15} están marcados. Esto representa un conflicto efectivo, que debe ser resuelto por el coordinador. Como hemos supuesto que las transiciones del primer proceso son más prioritarias que las del segundo, la transición t_3 será la disparada, comenzándose a ejecutar el código de t_4.

4. Tras el final de esta ejecución, la transición t_5 pasa a ser ejecutada, y el lugar p_{15} vuelve a estar marcado, lo que, tras el disparo de t_{10}, sensibiliza la transición t_{11} y hace que se acepte el comienzo de su proceso-CODE. En este momento hay dos procesos-CODE activos, aunque sólo uno de ellos, el de t_5, se ejecuta, pues es el más prioritario. El otro es suspendido hasta que tenga oportunidad de actuar. Este momento llega cuando el coordinador trata de disparar la transición t_6, no puede hacerlo y se suspende hasta que t_{11} termina marcando el lugar p_{15}.

5. El resto de la ejecución continúa como se muestra en el cronograma de la figura 3.4.

En el cronograma se aprecia el efecto de la presencia del coordinador (en negro denso en el cronograma 3.4). Cada acción de control o temporización que debe acometer el sistema debe ser desarrollada por el coordinador. Cada vez que se dispara una transición, se ejecuta un código o vence un evento temporal, el coordinador actúa y promueve el disparo del resto de la red. Esta supervisión introduce necesariamente una sobrecarga en el sistema implementado, reduciendo la cantidad de tiempo disponible para la ejecución de las transiciones CODE, que son la parte operativa del sistema. Así mismo, es la causa de la aparición de la deriva acumulativa, ya que la ejecución real de las actualizaciones de marcado, análisis de la sensibilización y la gestión de eventos no es instantánea, como se presupone en el modelo teórico. En la siguiente sección evaluaremos esta presencia del coordinador.

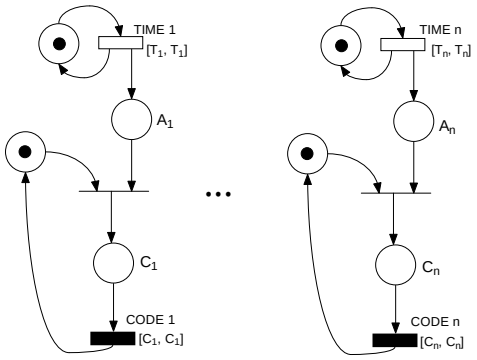


Figura 3.5: Modelo para la evaluación de prestaciones en la implementación centralizada

3.7. Evaluación de prestaciones

En los estudios tradicionales sobre implementación centralizada de redes de Petri se han realizado análisis de prestaciones (véase, p.ej., [Vil90] o [Bri95]). Estos trabajos se han centrado en la evaluación de parámetros tales como la habilidad, o el tiempo medio por disparo de transición, obteniéndose siempre que los resultados del método de LRM eran comparables a los de LRMs, dependiendo de las topologías de la red implementada. En esta tesis se enfocará el estudio desde otro punto de vista, evaluándose un parámetro de mayor interés para los sistemas de tiempo real. En concreto, nos ocuparemos del estudio de la sobrecarga introducida por el coordinador en el sistema implementado.

Tal como se puso de manifiesto en la sección anterior, la presencia del coordinador resta tiempo de utilización del procesador al conjunto de procesos-CODE de la implementación. Esto supone una reducción del factor de utilización máximo que se puede obtener del sistema.

Con el objetivo de evaluar el impacto del coordinador, hemos realizado un experimento consistente en la implementación de un modelo escalable compuesto por N tareas periódicas, todas ellas del mismo período (T) (ver modelo en la figura 3.5). El tiempo de ejecución (C) del código de cada proceso se ajusta para aprovechar al máximo el procesador (idealmente $C = T/N$). Teóricamente el factor de utilización (U) de un sistema de tales características debe ser el 100 %, pero si se consideran factores como la influencia del sistema operativo, la exactitud del delay, la sobrecarga por cambios de contexto y otros factores debidos al núcleo de Ada 95, el límite máximo de factor de utilización se reduce, en el peor de los casos que analizaremos, hasta al 94 %.

El experimento ha consistido en variar el número de procesos, de 5 a 30, y su período, de 1 a 0,2 segundos. En cada caso se ha registrado el valor máximo de factor de utilización. Para ello se ha medido el valor máximo del tiempo de ejecución del código de cada proceso sin que se produzca ninguna violación de período. Estas violaciones o retrasos se detectan cuando los lugares A_i y C_i de alguna de las redes del modelo de la figura 3.5 aparecen marcados simultáneamente. Esto indica que, en el momento de activación de un período de ejecución, el código del período anterior no ha terminado. El experimento que nos ha conducido a la tabla 3.1 (se representa en forma de gráfico en la figura 3.6). La ejecución se ha realizado sobre una estación de trabajo Sun SPARC CPU -5V, con un procesador microSPARC-II, utilizando el compilador de Ada 95 GNAT 3.03 para Solaris, y mediante la técnica de las transiciones sensibilizadas. Resultados similares se han obtenido en otras plataformas de ejecución.

En la interpretación de los resultados de la tabla 3.1 es preciso advertir que los resultados obtenidos incluyen la sobrecarga debida tanto al coordinador como al núcleo de Ada 95. Como ya hemos indicado, el límite real de factor de utilización del sistema formado por N procesos se reduce hasta el 94 % en el peor de los casos ($N = 30$, $T = 0{,}2$), debido a la influencia del núcleo de

N \ T	1	0.8	0.6	0.4	0.2
5	96,92	96,17	95,17	92,60	85,02
10	95,83	94,75	93,11	89,81	80,59
15	95,03	93,55	90,93	86,80	75,28
20	93,90	91,69	89,60	83,61	70,00
25	92,05	90,16	87,67	81,18	64,53
30	91,02	89,51	85,65	78,28	58,41

Cuadro 3.1: Factor de utilización máximo (U %) de la red de la figura 3.5 en función del número de procesos (N) y su período (T)

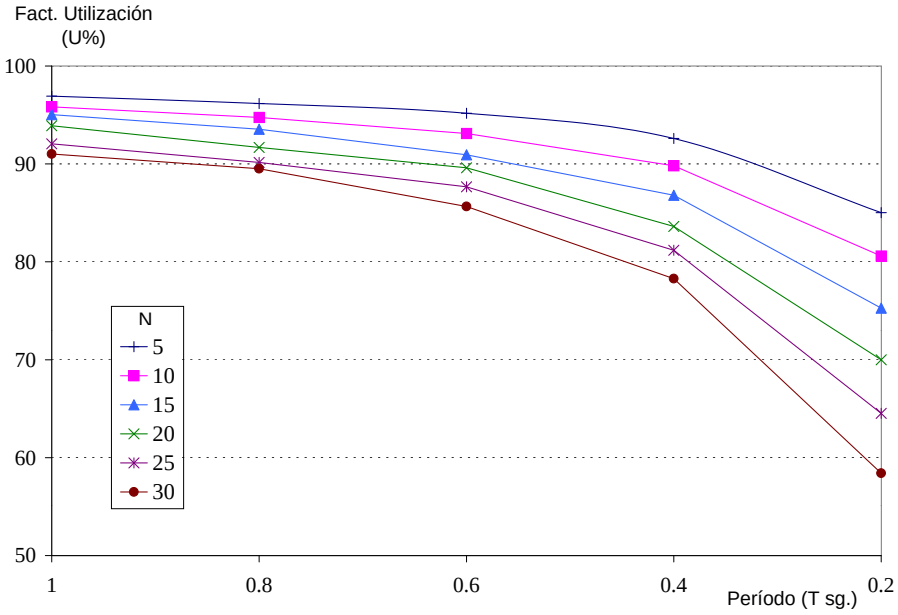


Figura 3.6: Diagrama que muestra la influencia del coordinador en la reducción del factor de utilización de una implementación centralizada

Ada 95 y al sistema operativo. El resto de la reducción, hasta el 58,41 % en ese caso, se deberá al coordinador.

3.8. Resumen de resultados y aportaciones del capítulo

En este capítulo se ha presentado una técnica para la implementación de sistemas de tiempo real modelados mediante redes de Petri con tiempo. Se ha presentado una particularización de las técnicas tradicionales de implementación centralizada, aplicadas a las RdPT, lo que ha supuesto un esfuerzo para solucionar el problema de la gestión del tiempo en métodos de implementación que no estaban adaptados para tal fin. Las técnicas tratadas son *interpretadas*, basadas, por tanto, en el uso de un proceso *coordinador* que gestiona la evolución de la red, la supervisa temporalmente y controla la ejecución de un conjunto de procesos (*procesos-CODE*). Como consecuencia de esta arquitectura, las partes de control y operativa del sistema son separadas en la implementación. La primera corresponde al coordinador y la segunda a los procesos-CODE.

Así mismo se ha presentado el algoritmo general que ejecuta el coordinador en su *ciclo de tratamiento*. Se ha elegido un esquema de implementación síncrona, por lo que en el ciclo de tratamiento, se han distinguido dos fases. Primero se evalúa la sensibilización de todas las transiciones disparables y se inicia su disparo, respetando las convenciones establecidas para la resolución de conflictos: primero se estudian las transiciones SYCO, luego las TIME y por último las CODE. En una segunda fase, el coordinador espera que se produzca el final de la ejecución de las acciones asociadas a alguna transición, lo que provoca una actualización de marcado, que deberá volver a ser tratado en otra ejecución del ciclo de tratamiento.

La carga computacional de la primera fase del ciclo de tratamiento del coordinador (detección de transiciones sensibilizadas e inicio de su disparo) es grande, por lo que tradicionalmente se han propuesto mejoras, que reducen la cantidad de transiciones sobre las que se realiza el test de sensibilización. En este capítulo se han adaptado las técnicas ya existentes (*lugares representantes marcados* y *transiciones sensibilizadas*) al caso de las redes de Petri con Tiempo, tal como nosotros las entendemos y utilizamos.

Todo ello conduce a un método de implementación que destaca por su simplicidad. Al tratarse de una técnica interpretada, la implementación es tan sencilla como construir una estructura de datos, que represente la estructura de la red y que será ejecutada por el coordinador. Por la misma razón es muy adecuada para la simulación de redes. Además, el estado completo del sistema está representado internamente en el coordinador, por lo que aumenta la trazabilidad del sistema.

Tales características hacen a esta técnica, en comparación con las compiladas (ver sección 2.5.2), especialmente indicada en las primeras etapas del desarrollo del sistema, en las fases de diseño y prototipado, lo que redunda en un incremento de la flexibilidad del diseño. Además la técnica es muy fácil de automatizar.

Sin embargo, y pese a su simplicidad, no todo son ventajas. La utilización de un único proceso coordinador responsable del control de la evolución de la red provoca importantes problemas que van en detrimento de la eficiencia (entendida en términos de factor de utilización máximo) y de las prestaciones de la implementación, sobre todo considerando que su campo de aplicación han de ser los sistemas de tiempo real. Enumeremos los problemas detectados:

- *Sobrecarga.* El coordinador debe actuar en cada disparo de transición de la red, evaluando su sensibilización y actualizando el marcado tras el disparo (ver seccción 3.7). Esta intervención introduce una sobrecarga en el sistema implementado que puede reducir de forma significativa el factor de utilización máximo del sistema implementado.

- *Pérdida de la concurrencia real en la parte de control de la red.* Al ser el coordinador el único responsable del control de la red, la evolución de la misma, teóricamente concurrente, es secuencializada inevitablemente.

- *Reducida tolerancia a fallos.* Toda la evolución de la red depende de un único elemento de control, lo que hace al sistema implementado altamente sensible a la ocurrencia de fallos: si el coordinador falla, todo el sistema fallará con él.
- *Gran número de procesos en la implementación.* Al dedicar un proceso para cada una de las transiciones CODE presentes en el modelo, el número de procesos de la implementación puede ser mucho mayor que la concurrencia real. Considérese, por ejemplo, el caso de un grafo de estados (GE) compuesto por N transiciones CODE. El número total de procesos en una implementación centralizada sería N+1 (uno por cada transición, más uno por el coordinador), cuando, por ser un GE, la concurrencia real es 1.
- *Gran número de comunicaciones en la implementación (tráfico alto).* Puede considerarse una consecuencia directa del punto anterior. Cada proceso-CODE debe realizar dos comunicaciones con el coordinador: una para comenzar su ejecución y otra para notificar el final de ésta (lo que acarreará el disparo de la transición asociada y por tanto la actualización del marcado).

La solución a estos problemas pasará por abordar la implementación de manera descentralizada, sin utilizar ningún proceso coordinador, y haciendo que cada parte de la implementación sea responsable de su propio control. De esta idea nacen las implementaciones descentralizadas, que serán tratadas en los dos próximos capítulos.

Capítulo 4

Implementación descentralizada I. Descomposición de una RdPT en procesos secuenciales

4.1. Introducción

La arquitectura de una implementación centralizada (capítulo 3) se basa en la separación de las partes de control y operativa del sistema, utilizando un proceso coordinador responsable de la primera, y varios procesos encargados de la segunda (uno por transición CODE). Como ya se ha comentado (sección 3.8), la utilización de un único proceso coordinador responsable del control de la evolución de la red, esto es, análisis de la sensibilización, elección de la transición a disparar y actualización del marcado, provoca un detrimento de las prestaciones de la implementación (en términos de factor de utilización máximo). Los problemas detectados han sido:

- La sobrecarga introducida por el coordinador, que puede reducir de forma significativa el factor de utilización máximo del sistema implementado (ver los detalles de esta influencia en la sección 3.7).
- Pérdida de concurrencia en la parte de control de la red, debido a que el coordinador, único responsable del control de la red, secuencializa su evolución.
- Reducida tolerancia a fallos, ya que si el responsable del control, el coordinador, falla, todo el sistema fallará con él.
- Gran número de procesos en la implementación, al dedicar un proceso para cada una de las transiciones CODE presentes en el modelo.
- Gran número de comunicaciones en la implementación (tráfico alto), ya que cada proceso-CODE debe realizar dos comunicaciones, de inicio y final, con el coordinador.

La solución a todos estos problemas pasa por evitar la presencia del coordinador, origen de los mismos. En las implementaciones descentralizadas el control de la red es dividido en varios procesos secuenciales concurrentes que se comunican entre sí. Claramente, el primer paso en la implementación es el reconocimiento de estos procesos en la red de Petri, tarea no trivial si se considera, como bien se menciona en [Bri95], que en las redes de Petri no existe el concepto de proceso, aunque sí que modelan correctamente sistemas compuestos por actividades secuenciales que evolucionan en conjunto de forma concurrente. Por tanto puede ser interesante, continúa

Briz, incorporar de algún modo el concepto de proceso a las redes de Petri. Esto se puede hacer de dos modos: bien descomponiendo la red en un conjunto de procesos secuenciales (máquinas de estados) de evolución concurrente y que se comunican entre sí sincronizándose o mediante el envío de mensajes; bien construyendo la red de Petri mediante la agrupación de máquinas de estados, desarrolladas por separado e interconectadas de distintas maneras. Es decir, puede hacerse utilizando *técnicas de descomposición* o *técnicas de construcción*, respectivamente. En este trabajo se utilizarán técnicas de descomposición o partición.

Antecedentes En la literatura pueden encontrarse varias referencias que abordan la descomposición de una red de Petri en procesos. Quizás la manera más simple [Tau88] consiste en explotar al máximo la concurrencia de la red, de modo que cada transición sea un proceso (*proceso-transición*) capaz de evaluar su sensibilización de forma autónoma (obsérvese la diferencia con las implementaciones centralizadas, donde la sensibilización era evaluada por el coordinador). Para ello se comunica con sus lugares de entrada, implementados como recursos comunes (*procesos-lugar*), para determinar si estos están o no marcados. Tiene la ventaja de ser muy simple, ya que se obvia el problema de la partición en procesos, pero no soluciona el problema del gran número de procesos de la implementación incluso lo empeora y sobrecarga el sistema con un gran número de comunicaciones entre procesos-transición y lugares para comprobar el marcado de estos y actualizarlo tras el disparo. Además, ante la presencia de conflictos en la red, esta técnica obliga al mantenimiento de protocolos de reserva de las marcas de los lugares compartidos entre varias transiciones. Obsérvese, a modo de ejemplo la figura 4.1.a donde aparece una situación de conflicto entre los procesos-transición t_1, t_2, t_3, t_4: para que t_2 pueda disparar es necesario que consulte el marcado de p_1 y, antes de consultar el marcado de p_2, lo proteja contra intentos de desmarcado por parte de t_1.

En [Vil90] se expone un interesante concepto para reducir el número de procesos y de comunicaciones en la implementación: los *sincronizadores locales mínimos* ($\mathcal{SL}m$). La idea es dividir la red en conjuntos de lugares llamados sincronizadores locales teniendo en cuenta para ello una regla: agrupar en un mismo sincronizador todos los lugares de entrada de un conjunto de transiciones en conflicto acoplado[1]. De esta forma todas las precondiciones de un conjunto de transiciones en conflicto acoplado (CA) están agrupadas en un mismo sincronizador local que puede decidir de forma autónoma sobre la sensibilización de sus transiciones de salida sin necesidad de mantener protocolos de reserva (ver figura 4.1.b). Por añadidura, se logra reducir el número de procesos de la implementación al reducir el de procesos-lugar. La partición obtenida de esta manera coincide con la obtenida por medio de la aplicación de otra técnica [NHS83] basada en el concepto de "lockset". En el mismo trabajo [Vil90], Villarroel expone cómo y cuándo es posible fundir un $\mathcal{SL}m$ con sus transiciones de entrada y de salida e incluso varios sincronizadores locales entre sí, reduciendo aún más el número de procesos y comunicaciones sin disminuir la concurrencia efectiva de la implementación

Por último se pueden encontrar técnicas estructurales de obtención de procesos que fueron expuestas por primera vez en [CSV86], posteriormente ampliadas en [Vil90] y que serán desarrolladas y ampliadas en esta memoria. La idea se basa en la extracción a partir de la red de Petri de conjuntos de transiciones en exclusión mutua y que son asimilables al concepto de proceso secuencial que perseguimos, siendo de este modo reunidas las partes operativa y de control del sistema, que hasta ahora habían permanecido separadas, en un único ente. Los procesos se comunicarán entre sí por medio de mecanismos de comunicación síncrona (ej. *rendezvous*) o asíncrona (ej.*buffer*). La misma técnica ha sido utilizada en otros trabajos, y con otras extensiones de redes de Petri. Por ejemplo, en [Kor92, KP91, BP94] para el caso de las redes de Petri coloreadas, o [CF93] para redes de Petri temporizadas.

[1] Dos transiciones están en relación de conflicto acoplado si comparten lugares de entrada, o existe alguna otra transición que esté en conflicto acoplado con ambas (ver Anexo sobre Redes de Petri, anexo A, para una definición formal)

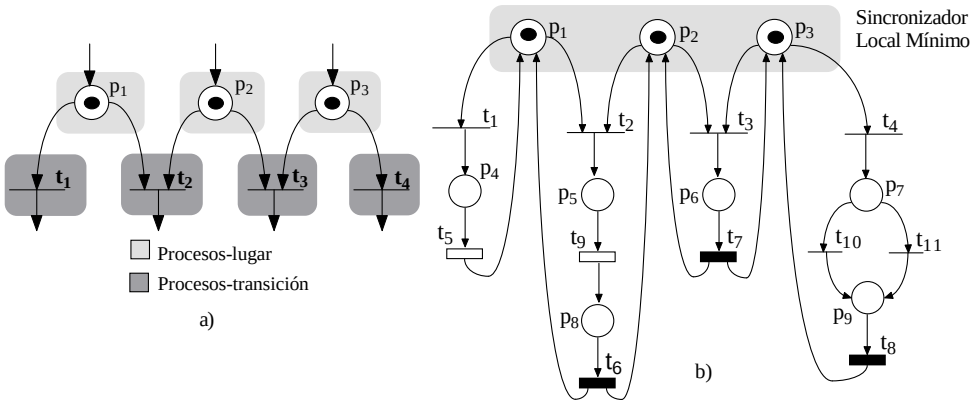


Figura 4.1: a) La existencia de conflictos fuerza el mantenimiento de protocolos de reserva para poder decidir sobre la sensibilización de las transiciones. b) Un sincronizador local mínimo es el responsable de determinar la sensibilización de sus transiciones de salida, evitando los mencionados protocolos

Acabaremos mencionando que el presente capítulo y el siguiente recogen, entre otros, los resultados presentados en el trabajo [GV98].

Objetivos del capítulo En este capítulo nos ocuparemos de sentar las bases teóricas en las que se basará la obtención de procesos en RdPT, dejando para el siguiente los aspectos de programación o implementación software de los mismos:

- Serán formalizadas las ideas sobre partición de una red en procesos ya expuestas en [Vil90].
- Se expondrá cómo establecer el flujo de control de cada uno de los fragmentos en los que se ha partido la red, cada uno de los cuales constituirá un proceso. Este flujo de control será el que defina el flujo de ejecución cuando se programe el proceso en algún lenguaje de programación.
- Se presentará un exhaustivo estudio de la partición de redes con conflictos.

Estructura del capítulo Comenzaremos por una exposición de las ideas sobre las que se fundamenta la descomposición de una red en procesos, esto es, la agrupación de las transiciones de la red en conjuntos, atendiendo para ello a un criterio de exclusión mutua entre ellas. No es sencillo concluir sobre la exclusión mutua de dos transiciones de una red de Petri, motivo por el cual en la sección 4.3 se reformulará la descomposición en procesos en términos de conjuntos de lugares en exclusión mutua, para cuyo cálculo se conocen herramientas de fácil aplicación, por ejemplo el cómputo de invariantes de marcado. La siguiente sección (sección 4.4) resalta el hecho de que en algunas redes no es posible encontrar suficientes p-invariantes como para descomponerlas en procesos, por lo que se plantea un método alternativo basado en la construcción, mediante la adición de lugares implícitos, de p-invariantes donde no los hay. Este resultado nos permitirá enunciar, en la sección 4.5, un método a seguir para la descomposición de una red en procesos. Tras ella comentaremos cómo se pueden aplicar a las RdPT las técnicas descritas para redes de Petri clásicas. Hasta este punto en el capítulo se habrán evitado alusiones a la descomposición de redes en las que aparecen conflictos, dejando esta labor para la siguiente sección de este capítulo. Ésta es una de sus mayores aportaciones y en ella se plantea un análisis exhaustivo de la distribución de conflictos entre procesos. Para terminar, la última sección resaltará las principales ideas y aportaciones del capítulo.

4.2. Descomposición de una RdP en procesos secuenciales

4.2.1. Ideas básicas

Abordamos en este punto la descripción de la técnica utilizada para la determinación de un conjunto de partes de la red de Petri con tiempo a implementar que puedan ser equiparados a procesos secuenciales. Nuestros objetivos son:

- Recuperar la concurrencia de la parte de control del sistema, perdida en las implementaciones centralizadas, al evitar la utilización de un único proceso coordinador responsable de toda la ejecución.
- Incrementar las prestaciones de la implementación respecto a la correspondiente centralizada. Para ello deberemos:
  - Reducir el número de procesos de la implementación hasta hacerlo igual, o al menos acercarlo, a la concurrencia real del sistema modelado que se quiere implementar. Éste será el objetivo cuando la plataforma de implementación sea monoprocesador. En este caso, uno de los recursos a compartir es la CPU, por lo que cuanto menor sea el número de procesos, mayor será la eficiencia de la implementación. Sin embargo, reducir el número de procesos puede no ser lo más importante cuando se disponga de una plataforma distribuida. En este caso pueden primar otros criterios como la localización geográfica de los recursos utilizados, reducción del número de comunicaciones u otros. En esta memoria se consideran solamente plataformas monoprocesador (sección 2.5.2), por lo que la reducción del número de procesos será nuestro principal objetivo.
  - Reducir el tráfico de la implementación (número de comunicaciones entre los procesos). Al agrupar en un único proceso varias transiciones se consigue reducir el número de comunicaciones entre las mismas y sus lugares para determinar su sensibilización. Además los disparos de las transiciones en la mayoría de las ocasiones no generan nuevas comunicaciones sino que se ven transformados en simples cambios internos de estado en cada uno de los procesos.
- Obtener el código Ada 95 correspondiente a los procesos de forma directa y fácil de automatizar.

Se considerarán para su estudio únicamente RdP con las restricciones expuestas en el capítulo 2 (sección 2.2.4). Consideraremos:

- redes ordinarias,
- transiciones disparables al menos una vez,
- grado de sensibilización de las transiciones debe ser como máximo uno, para lo que es suficiente que alguno de sus lugares de entrada sea 1-limitado
- marcado inicial de lugares origen debe ser uno y los lugares terminales deben ser 1-limitados,
- ninguna transición carecerá de lugares de entrada,
- no es posible que existan conflictos entre transiciones CODE.

En este capítulo se comprenderá el porque de estos requisitos. Puede comprobarse que el estudio es más general que [Vil90], donde se trataban redes binarias y vivas que no presentasen situaciones de conflicto acoplado entre sus transiciones.

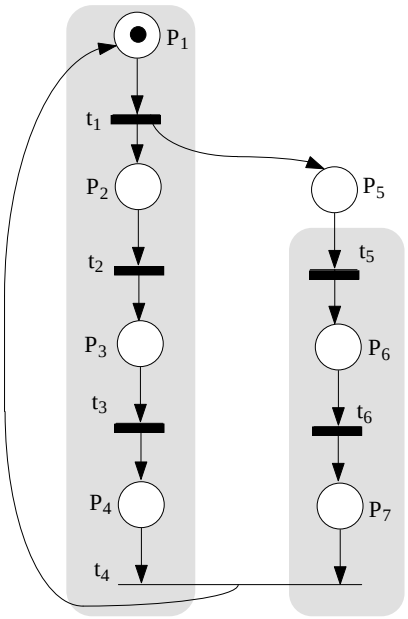


Figura 4.2: RdP que muestra dos conjuntos de transiciones en $\mathcal{EM}$ asimilables a procesos secuenciales

La idea de partida para la descomposición es la siguiente. Nuestra forma de modelar sistemas de tiempo real mediante RdPT hace uso de tres tipos de *unidades de ejecución* (sección 2.2.2): confía a las transiciones SYCO las acciones de control y sincronización; las labores de temporización recaen sobre la unidad formada por una transición TIME y sus lugares de entrada; y la ejecución de código sobre las transiciones CODE junto con sus lugares de entrada. Por lo tanto, para construir un proceso a partir de una RdPT trataremos de encontrar conjuntos de estas unidades, cuyas acciones asociadas describirán el comportamiento del proceso. Para que un grupo de estas unidades de ejecución pueda componer un proceso será necesario que no puedan ejecutarse a la vez, es decir que no se pueda ejecutar código al mismo tiempo que vence un evento o se dispara una acción de control, o que no se puedan ejecutar varios eventos y/o acciones de control simultáneamente. De este modo, podrán realizarse en secuencia dentro de un mismo proceso sin degradar la concurrencia del sistema implementado.

La aparente dificultad del problema de encontrar grupos de unidades de ejecución de estas características se reduce si se considera que es suficiente buscar grupos de transiciones en exclusión mutua. Dos unidades de ejecución no podrán ser ejecutadas al mismo tiempo si sus transiciones están en exclusión mutua ($\mathcal{EM}$), ya que si es así el disparo de una de las transiciones desensibilizará a las demás que pudieran estar sensibilizadas, abortando su ejecución. El único caso en el que podría haber dos unidades en ejecución simultánea, aun estando sus transiciones en $\mathcal{EM}$, sería el caso de dos transiciones CODE en conflicto, pero este caso se ha excluido en los requisitos de modelado.

Una vez centrados en el problema, nuestro objetivo será, por tanto, encontrar conjuntos de transiciones en exclusión mutua que serán el esqueleto de procesos secuenciales. Estos se comunicarán entre sí por medio de algún mecanismo de comunicación ya sea síncrono o asíncrono. La filosofía del método puede comprenderse considerando el ejemplo de la figura 4.2. En ella se han destacado dos conjuntos de transiciones $\{t_1, t_2, t_3, t_6\}$ y $\{t_4, t_5, t_6\}$ que se encuentran en exclusión

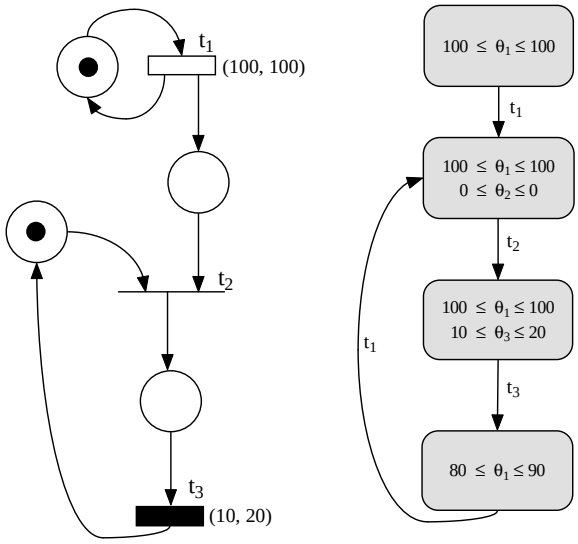


Figura 4.3: RdP con tiempo cuyas transiciones están en $\mathcal{EM}$, tal como se muestra en el grafo de clases de estados. Sin embargo las transiciones no están en $\mathcal{EM}$ en la red de Petri subyacente.

mutua y por lo que pueden ser considerados procesos secuenciales. Además, y a pesar de que este tema se comentará más detalladamente en secciones futuras, en la misma figura pueden apreciarse dos de los mecanismos de comunicación entre procesos: una transición compartida (t_6), que da lugar a una *comunicación síncrona* entre los procesos, y un lugar que no pertenece a ninguno de los procesos (p_5), que sirve como mecanismo de *comunicación asíncrona*. Así mismo avanzaremos que distinguiremos entre *lugares privados*, si están en el interior de algún proceso (por ejemplo p_2 o p_6), y *lugares de comunicación*, si sirven para conectar dos o más procesos (por ejemplo p_5).

Llegados a este punto es preciso advertir de que, mientras no se indique lo contrario, en esta sección del trabajo se utilizará como base del cómputo de procesos la red de Petri subyacente, es decir la red de Petri sin información temporal. Esta elección está fundamentada en el hecho de que todavía no se conocen técnicas para la determinación de transiciones en exclusión mutua en una RdPT (se debería recurrir al grafo de clases de estado para determinarlo). Esta decisión no será fuente de errores, ya que dos transiciones en exclusión mutua en la red de Petri subyacente permanecen en exclusión mutua cuando se consideran las características temporales aportadas por la RdPT, por lo que se puede concluir sobre la exclusión mutua temporal (definición A.4.5 en anexo A) en términos de exclusión mutua no temporal. Sin embargo lo contrario no es cierto: dos transiciones en exclusión mutua temporal no tienen por qué estar en exclusión mutua en la red de Petri subyacente. Por ejemplo, tal como muestra el grafo de clases de estado en la figura 4.3, las tres transiciones t_1, t_2, t_3, están en exclusión mutua en la RdPT, pero no en la subyacente, en la que la red ni siquiera es limitada. Las técnicas estructurales empleadas en este punto están desarrolladas sobre redes de Petri ordinarias y sus resultados permiten determinar propiedades en la red de Petri subyacente, no en la RdPT. Aunque esto sea una limitación en la potencia de cómputo de la técnica, se utilizará al menos en esta primera etapa de la exposición. Posteriormente (sección 4.6) se añadirán comentarios sobre cómo ampliarla para considerar propiedades de exclusión mutua temporal y su posible relación con la planificabilidad del sistema.

4.2.2. Formulación del problema de cobertura

Hemos visto que, en síntesis, el problema de encontrar procesos en una red de Petri pasa por encontrar conjuntos de transiciones en exclusión mutua. Como uno de nuestros objetivos es reducir al máximo el número de procesos, estos conjuntos deberán ser lo más grandes posible.

La exclusión mutua es una relación de compatibilidad (cumple las propiedades reflexiva y simétrica, no la transitiva), por lo que se puede hablar de *clases de compatibilidad de transiciones en exclusión mutua máximas*, que serán la base de los conjuntos de transiciones que buscamos. Una forma de encontrar estas clases es calcular las parejas de transiciones en exclusión mutua y construir una tabla de pares compatibles, que, tras la aplicación del algoritmo de cómputo de clases de compatibilidad (ver anexo B), dé como resultado clases máximas de compatibilidad de transiciones en exclusión mutua.

Obtenidas estas clases de compatibilidad, para la determinación de los procesos se ha de resolver un *problema de cobertura* donde:

- los elementos a cubrir son las transiciones de la red y
- los elementos cobertores son las clases máximas de compatibilidad calculadas anteriormente.
- Cada proceso contendrá un conjunto de transiciones en exclusión mutua.

Ejemplo Como ejemplo consideremos la figura 4.4.a, en la que se muestra la red de Petri subyacente correspondiente a una RdPT. En la figura 4.4.c puede verse la tabla de compatibilidad que muestra las relaciones de exclusión mutua entre las transiciones de la red, a partir de la cual, y utilizando el algoritmo del anexo B, se pueden calcular dos clases máximas de transiciones en exclusión mutua (C_1 y C_2) que cubren todas las transiciones de la red (figura 4.4.d). La cobertura se puede realizar de múltiples maneras, por ejemplo como en la figura 4.4.b, donde se muestran dos procesos. △

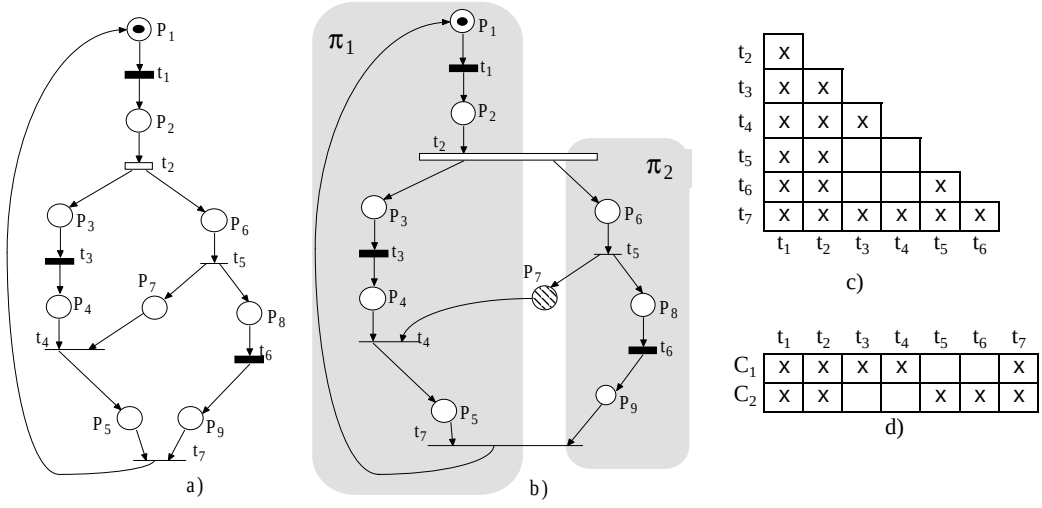


Figura 4.4: a) Red de Petri. b) Una posible partición, c) Tabla de compatibilidad de transiciones en exclusión mutua. d) Clases máximas de compatibilidad de transiciones en $\mathcal{EM}$.

Puede observarse que la técnica se basa en la determinación de los pares de transiciones en exclusión mutua. Dicha determinación puede hacerse a través del grafo de marcados o bien a través

de técnicas estructurales [Col89] que permiten obtener condiciones suficientes. Ambos métodos no son muy adecuados desde el punto de vista de la automatización del método debido a la explosión combinatoria en el número de estados del grafo de marcados, o a que no se disponen de condiciones necesarias y suficientes que nos permitan concluir con seguridad sobre la exclusión mutua. En [Vil90] se enuncia otro método que será el adoptado y completado en este trabajo. El problema de encontrar conjuntos de transiciones en exclusión mutua es transformado en otro para el que se conocen herramientas de fácil aplicación: el cálculo de conjuntos de lugares en exclusión mutua.

4.2.3. Cuestiones de nomenclatura. Definiciones

Antes de comenzar a describir la técnica de partición que adoptaremos es preciso detenerse a introducir algo de nomenclatura que será utilizada durante el resto de la exposición.

Proceso es un término muy amplio, que recoge varias interpretaciones. En esta memoria hemos estado utilizado hasta este momento la palabra proceso para referirnos a aquella parte secuencial de una red de Petri asimilable a una máquina de estados. Un poco más adelante describiremos cómo realizar la implementación software de estos procesos utilizando un lenguaje de alto nivel como Ada 95. Con el fin de matizar el significado del término proceso en ambos aspectos de la implementación, preferiremos emplear otro término para nominar las máquinas de estados en que descompondremos las redes, y dejar el término proceso para referirnos a la implementación software de las mismas. De este modo hablaremos de:

Definición 4.2.1 (Protoproceso) *Llamaremos protoproceso de una red de Petri* $\mathbf{N}<P, T; Pre, Post; M_0>$ *marcada y ordinaria, y lo denotaremos con la letra* π*, a una subred de la misma,* $\mathbf{N}' < P', T'; Pre', Post'; M'_0 >$*, formada por un conjunto de transiciones en exclusión mutua (*$T' \subseteq T$, $T \neq \emptyset$ *t.q.* $\forall t_i, t_j \in T', i \neq j : t_i \; \mathcal{EM} \; t_j$*), los lugares exclusivamente conectados a ellas (*$P' = \{p \in P |\; p^\bullet \cup {}^\bullet p \subseteq T'\}$*), los arcos que los conectan (*$Pre' = Pre|_{P' \times T'}$ *y* $Post' = Post|_{T' \times P'}$*) y siendo* M'_0 *la restricción de* M_0 *sobre P' (*$M'_0 = M_0|_{P'}$*).*

De esta definición es fácil deducir la siguiente propiedad:

Propiedad 4.2.1 *Sea* $\pi < P', T'; Pre', Post'; M'_0 >$ *un protoproceso de* $\mathbf{N} < P, T; Pre, Post >$*. Sea un lugar* $p \in P$*. Si* $p \in P'$ *entonces* ${}^\bullet p \cup p^\bullet \subseteq T'$*. Es decir, si un lugar pertenece a un protoproceso, todas sus transiciones de entrada y salida pertenecen al protoproceso también.*

Demostración. De la definición de protoproceso se deduce que el conjunto de lugares del mismo debe conectarse a transiciones exclusivamente pertenecientes al protoproceso ($P' = \{p \in P \mid p^\bullet \cup {}^\bullet p \subseteq T'\}$), por lo que es trivial comprobar que si $p \in P'$ entonces ${}^\bullet p \cup p^\bullet \subseteq T'$. △

Ejemplo En la red de la figura 4.4.b se muestran dos protoprocesos compuestos por los siguientes elementos: $\pi_1 = \{t_1, t_2, t_3, t_4, t_7; p_1, p_2, p_3, p_4, p_5\}$ y $\pi_2 = \{t_2, t_5, t_6, t_7; p_6, p_8, p_9\}$[2]. △

Definición 4.2.2 *Un protoproceso* $\pi < P', T'; Pre', Post'; M'_0 >$ *será un* protoproceso cerrado $\iff \forall t \in T', \exists p \in P'$ *t.q.* $p \in {}^\bullet t \wedge \exists p' \in P'$ *t.q.* $p' \in t^\bullet$*. En caso contrario el protoproceso será un* protoproceso abierto.

La definición anterior garantiza que un protoproceso cerrado no tiene *transiciones frontera* conectadas únicamente a otros lugares de la red no pertenecientes al protoproceso.

[2]Con ánimo de simplificar, siempre que en el texto mostremos un desarrollo de los elementos que componen un protoproceso, enumeraremos solamente los lugares y las transiciones, obviando los arcos que los interconectan.

Definición 4.2.3 *En un protoproceso abierto hablaremos de:*

- Transiciones de entrada, t_e, *aquellas transiciones que, perteneciendo al protoproceso, no tengan un lugar de entrada perteneciente al protoproceso, es decir,* $\nexists p \in P'$ *t.q.* $p \in {}^\bullet t_e$
- Transiciones de salida, t_s, *aquellas transiciones que, perteneciendo al protoproceso, no tengan un lugar de salida perteneciente al protoproceso, es decir,* $\nexists p \in P'$ *t.q.* $p \in t_s^\bullet$
- Transiciones frontera, *nombre genérico de las dos anteriores.*

Ejemplo El protoproceso π_1 de la figura 4.4.b es cerrado mientras que π_2 es abierto, con transición de entrada t_2 y de salida t_7. △

Obsérvese que al no tratarse necesariamente de redes vivas, los protoprocesos cerrados no tienen por qué ser cíclicos. Por contra, pueden contener lugares origen o terminales, entendiendo por este tipo de lugares lo siguiente:

Definición 4.2.4 *Sea* $\mathbf{N} < P, T; Pre, Post; M_0 >$ *una red de Petri marcada. Llamaremos:*

- lugar origen *a un lugar* $p \in P$ *que no tenga transiciones de entrada y sí de salida, es decir,* $|{}^\bullet p| = 0$ *y* $|p^\bullet| \geq 1$ *y*
- lugar terminal *a un lugar* $p \in P$ *que no tenga transiciones de salida y sí de entrada, es decir,* $|{}^\bullet p| \geq 1$ *y* $|p^\bullet| = 0$.

Ejemplo En la figura 4.5 se muestra una red de Petri descompuesta en dos protoprocesos, uno de los cuales (π_1) es un protoproceso cerrado que tiene un lugar terminal (p_4). En la misma figura 4.5 el protoproceso π_2 es un protoproceso abierto que tiene una transición de entrada t_1 y otra de salida t_4. △

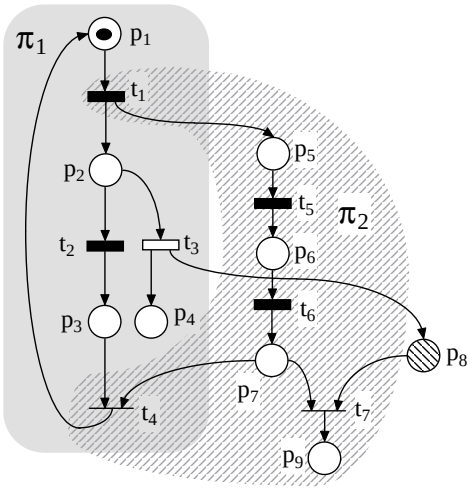


Figura 4.5: Partición de una red de Petri con lugares terminales en dos protoprocesos, uno cerrado (π_1) y otro abierto (π_2)

Observaciones

- Un protoproceso cerrado es una *trampa* y un *cerrojo* [Sil85], ya que el conjunto de transiciones de entrada de sus lugares es igual al de salida de los mismos. Por ello si inicialmente está marcado, siempre habrá marcas en el interior del protoproceso.
- Un protoproceso describe una máquina de estados secuencial ya que se compone únicamente de transiciones en exclusión mutua (por lo que no es posible que existan evoluciones paralelas en ella), si bien no definen un grafo de estados en el sentido de las redes de Petri [Sil85].

Definición 4.2.5 (Proceso) *Denominaremos proceso, y lo denotaremos* Π*, a la implementación software, utilizando algún lenguaje de programación, de un determinado protoproceso* π*.*

Ejemplo Como ejemplo consideramos la implementación software en Ada 95 del protoproceso π_1 de la figura 4.5.

```
task Proc_1 is
begin
  loop
    accept t1 do
      Codigo(t1); -- Transición t1
    end;
    select
      delay t3;   -- Transición t3
      p8.Mark;    -- Marcar p8
      exit;       -- Marcar p4
    then abort;
      Codigo(t2); -- Transición t2
    end select;
    Proc_2.t4;    -- Transición t4
  end loop;
end task;
```

△

4.3. Descomposición mediante conjuntos de lugares en exclusión mutua

Como ya se ha comentado, el problema de encontrar conjuntos de transiciones en exclusión mutua puede transformarse en el de calcular conjuntos de lugares en exclusión mutua. Seguidamente se demostrará por qué. Para ello se mostrará que las transiciones de entrada y de salida de un conjunto lugares en exclusión mutua están también en exclusión mutua, por lo que pueden pertenecer al mismo protoproceso. Esta proposición será la base de nuestra técnica de obtención de protoprocesos ya que se conocen métodos sencillos para la obtención de conjuntos de lugares en exclusión mutua.

Proposición 4.3.1 *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada. Sea* $p \in P$ *un lugar 1-limitado de la red. Entonces, el conjunto de transiciones de entrada y de salida de p están en exclusión mutua,* $\forall t_i, t_j \in {}^\bullet p \cup p^\bullet, i \neq j : t_i \; \mathcal{EM} \; t_j$

Demostración. Se demostrarán los siguientes supuestos:

1. Si $t_i, t_j \in {}^\bullet p$. Supongamos que $t_i \neg\mathcal{EM}\ t_j$, por lo que $\exists M \in R(\mathbf{N}, M_0)$, tal que t_i, t_j están sensibilizadas y pueden ser disparadas a la vez. Si las dos fueran disparadas el marcado de p podría ser $M(p) > 1$, por lo que no sería 1-limitado, lo cual entra en contradicción con el enunciado.

2. Si $t_i, t_j \in p^\bullet$, al ser p 1-limitado existirá un conflicto efectivo entre t_i y t_j, por lo que $t_i\ \mathcal{EM}\ t_j$.

3. Si $t_i \in {}^\bullet p$ y $t_j \in p^\bullet$. Supongamos que $t_i \neg\mathcal{EM}\, t_j$ por lo que $\exists M \in R(\mathbf{N}, M_0)$, tal que t_i, t_j están sensibilizadas. Si t_j está sensibilizada se debe verificar $M(p) = 1$. Si t_i está sensibilizada, puede disparar por lo que al ser disparada el marcado de p podría ser $M(p) > 1$, por lo que no sería 1-limitado, lo cual entra en contradicción con el enunciado. △

Definición 4.3.1 *Sea* $\mathbf{N} < P, T; Pre, Post >$ *una RdP y sea* $\Gamma \in \{1,0\}^n$, *con* $n = |P|$ *e* $\Gamma \neq 0$, *el vector característico de un conjunto de lugares de* P. *Se definen los siguientes conjuntos:*

1. *Transiciones cubiertas por un lugar* $p \in P : T(p) = {}^\bullet p \cup p^\bullet$
2. *Transiciones de salida de* $\Gamma : \Gamma^\bullet = \bigcup_{p_i \in \|\Gamma\|} p_i^\bullet$
3. *Transiciones de entrada a* $\Gamma : {}^\bullet\Gamma = \bigcup_{p_i \in \|\Gamma\|} {}^\bullet p_i$
4. *Transiciones cubiertas por* $\Gamma : T(\|\Gamma\|) = {}^\bullet\Gamma \cup \Gamma^\bullet$

Proposición 4.3.2 *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada. Sea* $\Gamma \in \{1,0\}^n$, *con* $n = |P|$ *e* $\Gamma \neq 0$, *el vector característico de un conjunto de lugares de* P *que se encuentran en exclusión mutua y que son 1-limitados* $(\forall M \in R(\mathbf{N}, M_0), \Gamma^T M \leq 1)$, $\forall p_i, p_j \in \|\Gamma\| : p_i\ \mathcal{EM}\ p_j$. *Se cumple que todas las transiciones cubiertas por* Γ *están en exclusión mutua:* $\forall t_i, t_j \in T(\|\Gamma\|), i \neq j : t_i\ \mathcal{EM}\ t_j$.

Demostración. Solamente será necesario demostrar que cualquier par de transiciones cubiertas por distintos lugares de $\|\Gamma\|$ están en exclusión mutua, ya que por la proposición 4.3.1 cualquier par de transiciones cubiertas por el mismo lugar están en exclusión mutua. Sean dos lugares $p_i, p_j \in \|\Gamma\|$.

- Sean dos transiciones $t_i \in {}^\bullet p_i$ y $t_j \in {}^\bullet p_j$, entonces $t_i\ \mathcal{EM}\ t_j$, ya que si no fuese así ambas podrían ser disparadas marcando p_i y p_j, por lo que estos lugares no estarían en exclusión mutua.

- Sean dos transiciones $t_i \in p_i^\bullet$ y $t_j \in {}^\bullet p_j$, entonces $t_i\ \mathcal{EM}\ t_j$. Supongamos primero que t_i está sensibilizada. Por tanto $M(p_i) = 1$. Entonces t_j no puede estar sensibilizada porque de estarlo podría ser disparada y marcaría p_j, por lo que estos lugares no estarían en exclusión mutua. Supongamos ahora que t_j está sensibilizada. Entonces t_i no puede estarlo, ya que si lo estuviera $M(p_i)$ sería 1. Como t_j está sensibilizada puede disparar y marcar p_j, por lo que p_i y p_j no estarían en exclusión mutua.

- Sean dos transiciones $t_i \in p_i^\bullet$ y $t_j \in p_j^\bullet$, entonces $t_i\ \mathcal{EM}\ t_j$, ya que si no fuese así podrían ser disparadas concurrentemente para lo que deberían estar sensibilizadas simultáneamente, siendo necesario entonces que $M(p_i) = 1$ y $M(p_j) = 1$, por lo que estos lugares no estarían en exclusión mutua. △

4.3.1. Reformulación del problema de cobertura

La proposición anterior es clave para nuestros intereses, ya que, gracias a ella, el *problema de cobertura*, enunciado en la sección 4.2.2, y que conduce a la partición de la red en una serie de protoprocesos equiparables a máquinas de estados, puede reformularse en los siguientes términos:

- Los elementos cobertores son conjuntos de lugares 1-limitados en exclusión mutua definidos por sus vectores característicos, $\Gamma \in \{1,0\}^n$, $n = |P|$, $\Gamma \neq 0$. Nos referiremos a estos cobertores con el término *elementos de partición* y lo denotaremos por $\mathcal{EP}$.
- Los elementos a cubrir son las transiciones de la red.

Nótese que la exigencia de que los lugares sean 1-limitados no provocará que queden sin considerar ninguna de las transiciones TIME o CODE del modelo. Debido a las restricciones de modelado, alguno de los lugares de entrada de este tipo de transiciones es siempre 1-limitado.

Enunciaremos seguidamente la proposición que servirá para formalizar el replanteamiento del problema de partición a través de conjuntos de lugares en exclusión mutua. Antes, definiremos un operador que nos facilitará la escritura de la estructura de los protoprocesos y que sirve para expresar el conjunto de lugares que pertenecen a un protoproceso. Como se deduce de la definición de protoproceso (definición 4.2.1), el conjunto de lugares de un protoproceso viene determinado por las transiciones del mismo (lugares que se conectan exclusivamente a las transiciones del protoproceso):

Definición 4.3.2 *Sea* $\mathbf{N} < P, T; Pre, Post >$. *Sea* $T' \subseteq T$ *un conjunto de transiciones. Denotaremos por* $Int(T') = \{p \in P |\ p^\bullet \cup {}^\bullet p \subseteq T'\}$, *es decir, al conjunto de lugares de* P *que se conectan exclusivamente a transiciones de* T'.

Proposición 4.3.3 *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada y ordinaria, y* P' *un subconjunto de lugares de* $\mathbf{N}$ *que están en exclusión mutua. La subred formada por* $< Int(T(P')), T(P'); Pre', Post'; M'_0 >$, *siendo* $Pre' = Pre|_{Int(T(P'))\times T(P')}$, $Post' = Post|_{T(P')\times Int(T(P'))}$ *y* $M'_0 = M_0|_{Int(T(P'))}$, *es un protoproceso.*

Demostración. La demostración es evidente. Como los lugares de P' están en exclusión mutua, por la proposición 4.3.2, las transiciones $T(P')$ están en exclusión mutua. Por construcción, los lugares $Int(T(P'))$ se conectan, únicamente a transiciones de $T(P')$. Por tanto π es un protoproceso. $\triangle$

Definamos ahora el nuevo término que se ha avanzado en el anterior planteamiento del problema de cobertura:

Definición 4.3.3 (Elemento de partición) *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada. Sea* $\Gamma \in \{1,0\}^n$, *con* $n = |P|$ *e* $\Gamma \neq 0$, *el vector característico de un subconjunto de lugares 1-limitados de* P *que se encuentran en exclusión mutua. Llamaremos* elemento de partición *de la red* $\mathbf{N}$, *y lo denotaremos por* $\mathcal{EP}$, *al soporte de* Γ, $\|\Gamma\|$.

Si se considera que se desea reducir al máximo el número de protoprocesos en los que descompondremos las redes de Petri, será conveniente que los elementos de partición sean máximos, es decir, no pueda ser ampliado, $\nexists p \in P$ t.q. $\forall p_i \in \mathcal{EP}, p\ \mathcal{EM}\ p_i$. Cuanto más numerosos sean los lugares del elemento de partición mayor será la clase de transiciones en exclusión mutua que cubre, por lo que serán necesarias menos clases de transiciones, y con ellas menos protoprocesos, para cubrir toda la red.

Definición 4.3.4 (Elemento de partición esencial) *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada. Sea* $\mathcal{EP}_i$ *un elemento de partición de* $\mathbf{N}$. *Diremos que* $\mathcal{EP}_i$ *es esencial* $\iff \exists t \in T(\mathcal{EP}_i)$, *transiciones cubiertas por* $\mathcal{EP}_i$, *t.q.* $\forall \mathcal{EP}_j, i \neq j, t \notin T(\mathcal{EP}_j)$, *es decir existe alguna transición de la red que es cubierta únicamente por el elemento de partición* $\mathcal{EP}_i$.

Definición 4.3.5 *Diremos que un protoproceso* $\pi_i < P_i, T_i; Pre_i, Post_i; M_0' >$ es generado *por un elemento de partición* $\mathcal{EP}_i$ *si* $\forall t \in T_i$, $t \in T(\mathcal{EP}_i)$.

Obtención de elementos de partición

La cuestión que surge inmediatamente es cómo encontrar estos conjuntos de lugares en exclusión mutua. Y la respuesta no puede ser más sencilla: encontraremos elementos de partición en los p-invariantes monomarcados.

Proposición 4.3.4 *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada, e* $Y \in \{1,0\}^n$, $n = |P|$ *e* $Y \neq 0$ *un p-invariante elemental monomarcado de la red* $\mathbf{N}$ ($Y^T C = 0$ *y* $Y^T M_0 = 1$). *El soporte del p-invariante* Y *constituye un elemento de partición,* $\mathcal{EP} = \|Y\|$.

Demostración. Al ser Y un p-invariante monomarcado todos los lugares de su soporte son 1-limitados y se verifica que $\forall M \in R(\mathbf{N}, M_0) : \sum_{p_i \in \|Y\|} M(p_i) = 1$, con lo que $\forall p_i, p_j \in \|Y\|$, $\nexists M \in R(\mathbf{N}, M_0)$ t.q. $M(p_i) = 1 \wedge M(p_j) = 1$, por lo que $p_i \, \mathcal{EM} \, p_j$. △

El cálculo de p-invariantes será el primer paso hacia la solución del problema de cobertura.

Ejemplo Consideremos como ejemplo el caso de la figura 4.4.a: el cálculo de p-invariantes pone de manifiesto tres de ellos, que resultan ser monomarcados, $I_1 = \mathcal{EP}_1 = \{p_1, p_2, p_3, p_4, p_5\}$, $I_2 = \mathcal{EP}_2 = \{p_1, p_2, p_5, p_6, p_7\}$ y $I_3 = \mathcal{EP}_3 = \{p_1, p_2, p_6, p_8, p_9\}$. Con ellos se puede construir la tabla de cobertura de la figura 4.6. En ella se puede apreciar que sólo dos elementos de partición son esenciales para la partición, los que resultan de los p-invariantes I_1 e I_2, y que la tabla resultante coincide con la de la figura 4.4.d obtenida por medio de las clases de compatibilidad de transiciones en exclusión mutua. △

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
$T(I_1)$	x	x	x	x			x
$T(I_2)$	x	x		x	x		x
$T(I_3)$	x	x			x	x	x

Figura 4.6: Tabla de partición correspondiente a la red de la figura 4.4.a calculada a partir de sus p-invariantes monomarcados

4.3.2. Solución del problema de cobertura.

Hasta el momento disponemos de una RdPT, para la cual, y obviando para ello las propiedades temporales de la misma, hemos obtenido una serie de conjuntos de lugares en exclusión mutua (elementos de partición, $\mathcal{EP}_i$). Con estos elementos es necesario resolver el problema de cobertura de las transiciones de la red, lo que nos proporcionará una partición de la red en protoprocesos ($\pi_i < P_i, T_i; Pre_i, Post_i; M_{0i} >$). Para ello se debe considerar que:

1. En virtud de la proposición 4.3.2, cada elemento de partición determina una clase de transiciones en exclusión mutua ($C_i = T(\mathcal{EP}_i)$).

2. Se agruparán en un mismo protoproceso transiciones cubiertas por los lugares de un mismo elemento de partición $\mathcal{EP}_i$, es decir de la misma clase de transiciones en exclusión mutua. Un protoproceso π_i puede contener todas las transiciones de una de estas clases ($T_i = T(\mathcal{EP}_i)$) o bien un subconjunto de ellas ($T_i \subset T(\mathcal{EP}_i)$), siempre que el resto de transiciones sea implementado por otro protoproceso.

3. El protoproceso generado a partir de las transiciones T_i tendrá la forma $< Int(T_i), T_i; Pre_i, Post_i; M_{0i} >$, según se puede deducir de la proposición 4.3.3.

4. Las transiciones no cubiertas por ningún elemento de partición se implementarán solas, cada una en un protoproceso distinto.

En definitiva, la partición debe garantizar que para una red de Petri $\mathbf{N} < P, T; Pre, Post >$, la unión de las transiciones pertenecientes a todos los protoprocesos π_i sea igual a T, $T = \cup T_i$. Así mismo, el protoproceso puede contener todos los lugares del elemento de partición o parte de ellos, verificándose, en cualquier caso, las siguientes proposiciones:

Proposición 4.3.5 *Sea $< \mathbf{N}, M_0 >$ una RdP marcada y ordinaria. Sea $\pi <$ P', T'; Pre', $Post'$; $M'_0 >$ un protoproceso de $\mathbf{N}$ generado por un elemento de partición $\mathcal{EP}$. El protoproceso contendrá todas y solamente las transiciones cubiertas por $\mathcal{EP}$ si y sólo si el protoproceso contiene todos los lugares del $\mathcal{EP}$, $T' = T(\mathcal{EP}) \iff \mathcal{EP} \subseteq P'$*

Demostración. La demostración es muy simple y se deduce casi inmediatamente del método propuesto para la solución del problema de cobertura:

- Demostremos primero que si $T' = T(\mathcal{EP}) \Rightarrow \mathcal{EP} \subseteq P'$. Se verifica por construcción: como $T' = T(\mathcal{EP}) = \bigcup_{p \in \mathcal{EP}} {}^\bullet p \cup p^\bullet$, se verificará que $\forall p \in \mathcal{EP}$, ${}^\bullet p \cup p^\bullet \subseteq T'$ lo que, considerando la definición de protoproceso (definición 4.2.1), implica que $\forall p \in \mathcal{EP}$, $p \in P'$.

- Veamos ahora que si $\mathcal{EP} \subseteq P' \Rightarrow T' = T(\mathcal{EP})$. Como todos los lugares del $\mathcal{EP}$ pertenecen al protoproceso, por la proposición 4.2.1, todas las transiciones cubiertas por ellos pertenecerán al mismo. Además, por construcción, sólo se toman las transiciones cubiertas por los lugares del elemento de partición ("se agruparan en un mismo protoproceso transiciones cubiertas por los lugares de un mismo elemento de partición"). △

Proposición 4.3.6 (Condición suficiente para protoproceso cerrado) *Sea $< \mathbf{N}, M_0 >$ una RdP marcada y ordinaria. Sea π un protoproceso de la red $\mathbf{N}$. Si π se ha generado a partir de un p-invariante elemental monomarcado de la red $Y \in \{1, 0\}^n$, con $n = |P|$ e $Y \neq 0$, e implementa todas las transiciones de $T(\|Y\|)$, es decir, π es de la forma $\pi < Int(T(\|Y\|)), T(\|Y\|); Pre', Post'; M'_0 >$, entonces π es cerrado.*

Demostración. Como Y es un elemento de partición (proposición 4.3.4), $\pi < Int(T(\|Y\|)), T(\|Y\|); Pre', Post'; M'_0 >$ es un protoproceso. Como es sabido, si Y es un p-invariante ($Y^T C = 0$), la subred formada con los lugares de $\|Y\|$, las transiciones cubiertas por ellos, $T(\|Y\|)$, y los arcos que los conectan es una subred cerrada. Por ello la subred $\pi < Int(T(\|Y\|)), T(\|Y\|); Pre', Post'; M'_0 >$, será también cerrada, pues sólo añade algún lugar a la anterior. △

La proposición anterior viene a demostrar que si existe un p-invariante que cubre todas las transiciones de un protoproceso, éste será cerrado. La implicación contraria no es cierta, es decir, si un protoproceso $\pi < P', T'; Pre', Post'; M'_0 >$ es cerrado no se verifica que $\exists Y$ p-invariante monomarcado que cubre todas las transiciones del protoproceso. Un contraejemplo que demuestra

esta aseveración se encuentra en la figura 4.11. En ella, tal como se demostrará en la sección 4.4, se muestra una red en la que todas sus transiciones están en exclusión mutua, todos los lugares son privados y, por tanto, constituye un protoproceso cerrado. Sin embargo, no es posible encontrar un p-invariante monomarcado en ella. Además no basta que un protoproceso contenga todas las transiciones cubiertas por un p-invariante monomarcado (o contenga todos sus lugares), sino que se debe verificar que contenga todas las transiciones y ninguna más.

En resumen, los p-invariantes monomarcados son utilizables como elementos de partición para la resolución del problema de cobertura de las transiciones de una red de Petri. Además, si agrupamos todas las transiciones cubiertas por un p-invariante en un mismo protoproceso, éste será cerrado.

Tabla de cobertura

Para ayudar en el planteamiento del problema de cobertura es conveniente construir una *tabla de cobertura*. En ella aparecerán, en cada fila, las transiciones cubiertas por el elemento de partición asignado a la fila y en las columnas, las transiciones de la red.

Para solucionar el problema de cobertura deberemos estudiar cómo elegir los elementos de partición para cubrir todas las transiciones de la red. Para ello es preciso considerar en primer lugar los elementos de partición esenciales, pues sin su concurso quedarían transiciones sin cubrir. El resto de transiciones deberá ser cubierto por el mínimo número de elementos de partición, ya que uno de nuestros objetivos es reducir el número de procesos de la implementación[3].

Ejemplo Como ejemplo consideremos la figura 4.7, en la que se muestra la red de Petri subyacente correspondiente a la red de la figura 4.4. El cálculo de p-invariantes ha servido para determinar tres elementos de partición, cada uno de los cuales cubre una clase de transiciones en exclusión mutua, como puede verse en la tabla de cobertura de la figura 4.7.e. △

Distintas soluciones del problema de cobertura

La solución del problema de cobertura no tiene por qué ser única. En la tabla 4.7.e se puede comprobar cómo algunas transiciones están cubiertas por varios elementos de partición, lo que indica que en la partición final de la red bien podrían pertenecer a cualquiera de los protoprocesos generados o a todos a los que son comunes. La pertenencia de las transiciones a uno u otro protoproceso es el punto clave a la hora de realizar la partición de la red. Podemos considerar los siguientes casos:

1. *Transición compartida.* En este caso una transición puede pertenecer a dos o más protoprocesos al mismo tiempo. En las figuras 4.7.b se muestran dos protoprocesos en los que se han compartido dos transiciones (t_2 y t_7).

2. *Cada transición pertenece a un único protoproceso.* Esta estrategia divide la red en conjuntos de transiciones disjuntos. Éste es el criterio aplicado para la obtención de los protoprocesos de la figura 4.7.c.

3. *Transiciones y lugares conexos compartidos.* Esta situación se manifiesta cuando varios elementos de partición presentan intersección no vacía, es decir, comparten lugares. En este caso también pueden compartirse las transiciones de entrada y salida de estos lugares. Por así decirlo, es como si se compartiese un trozo de protoproceso, (obsérvese que el caso 1. es una particularización de éste ya que una transición aislada es un protoproceso por sí misma).

[3]Otras consideraciones (por ejemplo la distribución de la implementación) podrían desaconsejar esta estrategia y aconsejar otra serie de criterios para realizar la partición. En ese caso la distribución geográfica puede ser más interesante que la obtención de un mínimo número de procesos.

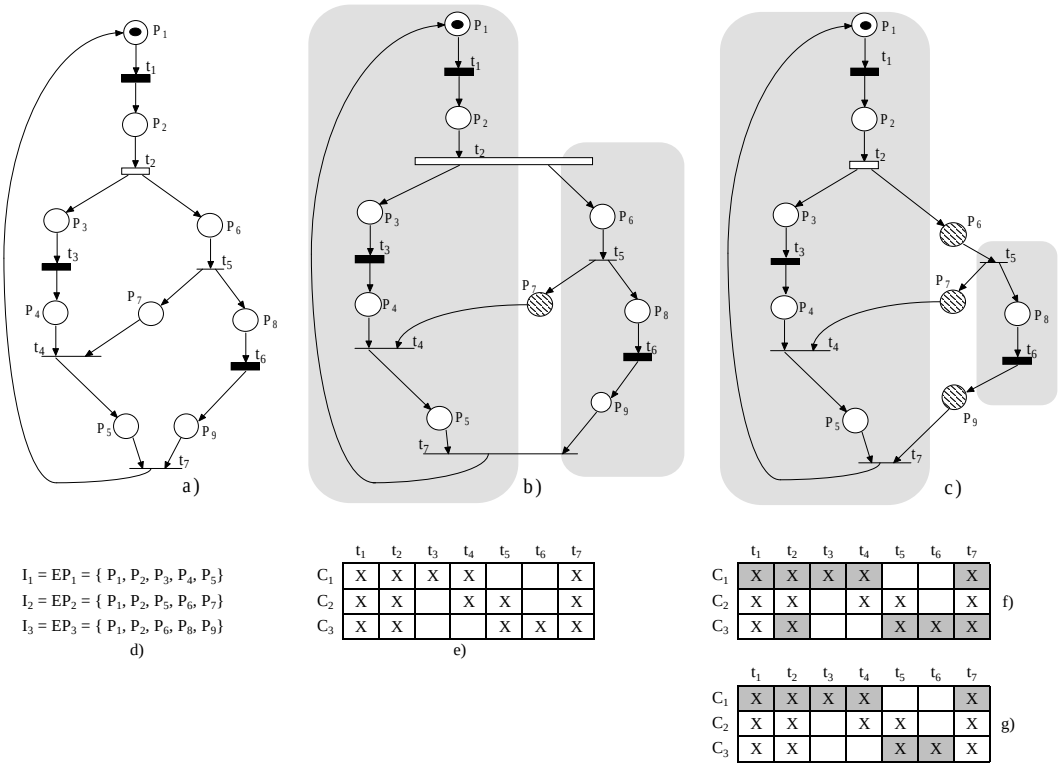


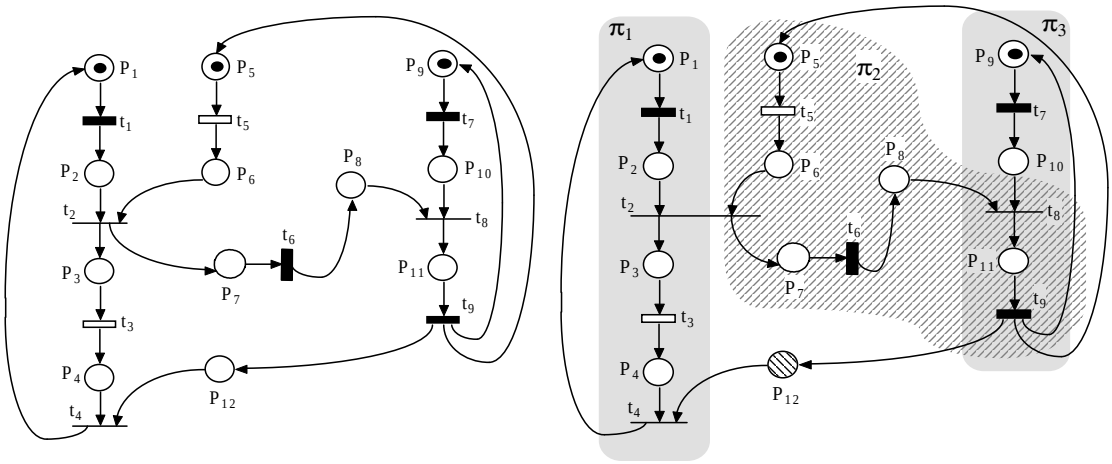
Figura 4.7: a) Red de Petri. b) Partición primando el compartimiento de transiciones. c) Partición sin compartir transiciones. d) Elementos de partición de la red. e) Clases de compatibilidad de transiciones en $\mathcal{EM}$. f) Partición correspondiente a la figura b). g) Partición correspondiente a la figura c).

La única condición que deben cumplir estas transiciones y lugares compartidos es que exista un camino en la red que los una a todos ellos, es decir, que sean conexos.

4.3.3. Comunicación entre protoprocesos.

Cada una de las tres posibilidades de resolución del problema de cobertura comentadas en el punto anterior inciden de distinta forma en el modo de comunicación entre protoprocesos. Ilustrémoslas con un ejemplo, la red de la figura 4.8.a. Un cálculo de p-invariantes monomarcados permite determinar cuatro elementos de partición $\mathcal{EP}_1 = \{p_1, p_2, p_3, p_4\}$, $\mathcal{EP}_2 = \{p_1, p_2, p_7, p_8, p_{11}, p_{12}\}$, $\mathcal{EP}_3 = \{p_5, p_6, p_7, p_8, p_{11}\}$ y $\mathcal{EP}_4 = \{p_9, p_{10}, p_{11}\}$. Las transiciones cubiertas por cada uno de los elementos de partición se reflejan en la tabla 4.8.c. En ella apreciamos que $\mathcal{EP}_1$, $\mathcal{EP}_3$ y $\mathcal{EP}_4$ son esenciales y entre ellos cubren todas las transiciones de la red. Además, se puede comprobar como varias transiciones son cubiertas por varios elementos de partición, por lo que se pueden encontrar distintas soluciones al problema de cobertura.

La solución propuesta en la figura 4.8.b, está integrada por tres protoprocesos: $\pi_1 = \{p_1, p_2, p_3, p_4; t_1, t_2, t_3, t_4\}$ (generado por $\mathcal{EP}_1$); $\pi_2 = \{p_5, p_6, p7, p_8, p_{11}; t_5, t_2, t_6, t_8, t_9\}$ (generado por $\mathcal{EP}_3$); $\pi_3 = \{p_9, p_{10}, p_{11}; t_7, t_8, t_9\}$ (generado por $\mathcal{EP}_4$), que, al cumplirse los requisitos de la proposición 4.3.6, son protoprocesos cerrados. En esta solución se ponen de manifiesto las tres formas en las que se pueden comunicar los protoprocesos:



$I_1 = EP_1 = \{ P_1, P_2, P_3, P_4\}$
$I_2 = EP_2 = \{ P_1, P_2, P_7, P_8, P_{11}, P_{12}\}$
$I_3 = EP_3 = \{ P_5, P_6, P_7, P_8, P_{11}\}$
$I_4 = EP_4 = \{ P_9, P_{10}, P_{11}\}$

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
$T(EP_1)$	x	x	x	x					
$T(EP_2)$	x	x		x		x		x	x
$T(EP_3)$		x			x	x		x	x
$T(EP_4)$							x	x	x

c)

Figura 4.8: a) Red de Petri. b) Una de las posibles particiones en protoprocesos. c) Tabla de cobertura.

1. *Transición compartida.* Una transición compartida entre dos o más protoprocesos representa una acción compartida entre ellos, que debe ser desarrollada a la vez en los protoprocesos (la transición se dispara atómicamente y a la vez en todos los protoprocesos). Por tanto, se trata de es una primitiva de *sincronización entre protoprocesos*, que puede ser entendida como una simple comunicación síncrona, o como una invocación remota, si en ella se realiza alguna acción [BW97]. En la figura 4.8.b, la transición compartida t_2 sincroniza los protoprocesos π_1 y π_2.

2. *Lugar de comunicación.* Un lugar que no pertenezca a ningún protoproceso es sinónimo de una *comunicación asíncrona.* Este tipo de lugares (que llamaremos de comunicación) permite el paso de marcas de un protoproceso a otro de forma asíncrona, ya que el protoproceso que marca el lugar de comunicación puede evolucionar independientemente del instante en que el otro protoproceso retira la marca del lugar. Es el caso del lugar p_{12} en la figura 4.8.b. El lugar juega el papel de "buffer"que desacopla los dos protoprocesos comunicados. Esta forma de comunicación aparece generalmente cuando una transición cubierta por varios elementos de partición es asignada a un único protoproceso. Obsérvese, por ejemplo, que si la transición t_2 de la figura 4.8.b hubiese sido asignada exclusivamente al protoproceso π_1, se hubiesen generado dos lugares de comunicación asíncrona entre π_1 y π_2: p_6 y p_7.

3. *Subred compartida.* Cuando se comparten varias transiciones y lugares conexo se está de nuevo ante una sincronización entre los protoprocesos, esta vez en forma de *cita extendida.* Los protoprocesos se sincronizan para desarrollar unas determinadas acciones, que conforman

el cuerpo de la cita. En la figura 4.8.b, los protoprocesos π_2 y π_3 se comunican síncronamente a través de las transiciones $\{t_8, t_9\}$ y el lugar $\{p_{11}\}$.

Las distintas soluciones del problema de partición de una red de Petri diferirán unas de otras en la forma de repartir las transiciones y, por tanto, en la forma de comunicar los protoprocesos resultantes. De ello dependerán el comportamiento y las prestaciones de unas soluciones y otras.

4.3.4. Clasificación de los lugares

Una vez obtenida una partición de una red y teniendo en cuenta las formas de comunicación entre los protoprocesos, podemos realizar una clasificación de los lugares de la red atendiendo a un criterio de localización de los mismos respecto de los protoprocesos. Sea $\mathbf{N} < P, T; Pre, Post; M_0 >$ una red de Petri y sean $\pi_i < P_i, T_i; Pre_i, Post_i; M_{0i} >$ y $\pi_j < P_j, T_j; Pre_j, Post_j; M_{0j} >$ dos protoprocesos de la misma. Hablaremos de:

- *Lugar privado.* Un lugar $p \in P$ será privado respecto de $\pi_i \iff p \in P_i$, o lo que es lo mismo, está conectado únicamente a transiciones de π_i: $\forall t \in {}^\bullet p \cup p^\bullet, t \in T_i$. Por ejemplo, en la figura 4.8.b, el lugar p_1 es privado respecto de π_1.

- *Lugar externo.* Un lugar $p \in P$ será externo respecto de $\pi_i \iff p \notin P_i \wedge \forall t \in {}^\bullet p \cup p^\bullet, t \notin T_i$, es decir si está únicamente conectado a transiciones no pertenecientes a π. Por ejemplo, en la figura 4.8.b, el lugar p_1 es externo respecto de π_2.

- *Lugar compartido.* Un lugar $p \in P$ será compartido por π_i y $\pi_j \iff p \in P_i \wedge p \in P_j$ y $\forall t \in {}^\bullet p \cup p^\bullet, t \in T_i \wedge t \in T_j$, es decir si se conecta únicamente con transiciones pertenecientes simultáneamente a ambos protoprocesos. Por ejemplo, en la figura 4.8.b, el lugar p_{11} es compartido por π_2 y π_3.

- *Lugar de comunicación.* Un lugar $p \in P$ será de comunicación entre los protoprocesos π_i y $\pi_j \iff p \notin P_i \wedge p \notin P_j$ y $\forall t \in {}^\bullet p \cup p^\bullet, t \in T_i \wedge t \notin T_j \vee t \notin T_i \wedge t \in T_j$, es decir el lugar está conectado a transiciones que pertenecen sólo a π_i y por otra parte a transiciones que sólo pertenecen a π_j. Por ejemplo, en la figura 4.8.b, el lugar p_{12} es de comunicación entre π_1 y π_3. Hablaremos de lugares de comunicación de:

  - *destino único* si las transiciones de salida del lugar pertenecen a un único protoproceso
  - *destino múltiple* si las transiciones de salida del lugar pertenecen a varios protoprocesos.

4.3.5. Construcción del flujo de control de un protoproceso a través de p-invariantes monomarcados.

Como resultado del problema de cobertura se obtienen una serie de protoprocesos que implementan todas o parte de las transiciones cubiertas por el elemento de partición que los genera. Sin embargo la utilidad de los elementos de partición, si estos son p-invariantes monomarcados, no se limita a la solución del problema de cobertura.

Los lugares de los elementos de partición pueden utilizarse para describir el flujo de control de los protoprocesos que generan, esto es, para definir la secuencia de disparos de transiciones que se producen durante su ejecución. En efecto, los lugares de cada elemento de partición están en exclusión mutua, por lo que sólo uno de ellos puede estar marcado en cada instante. De este modo, sólo las transiciones de salida del lugar marcado son susceptibles de ser disparadas. Para ello es necesario poder garantizar que cada transición de un protoproceso está precedida por un lugar del elemento de partición, lo que es cierto únicamente si el elemento de partición es un p-invariante monomarcado, ya que entonces el protoproceso es cerrado y existe un lugar del p-invariante a la

entrada y a la salida de cada transición cubierta por él. Sin embargo es falso si el $\mathcal{EP}$ no es un p-invariante. Un contraejemplo puede verse en la figura 4.11, donde hay un $\mathcal{EP}$ compuesto por los lugares $\{p_1, p_2, p_4\}$, que no es un p-invariante monomarcado (más adelante se explicará cómo se ha obtenido). El $\mathcal{EP}$ genera un protoproceso cerrado que presenta una transición, t_6, que no tiene a su entrada un lugar del elemento de partición. Por ello advertimos que todo lo que se expondrá en este punto es válido únicamente para elementos de partición obtenidos a través de p-invariantes monomarcados.

En síntesis, la evolución del marcado a través de los lugares de los p-invariantes monomarcados determina el flujo de control de los protoprocesos generados por éstos: sólo se podrán disparar las transiciones de salida del lugar que contiene la marca en cada instante. El resto de lugares privados no pertenecientes al p-invariante serán implementados como variables locales al protoproceso que serán tenidas en cuenta para resolver conflictos o como comprobación de la corrección de la ejecución: observar que si se trata de lugares interiores al protoproceso deben ser marcados y desmarcados por éste, ya que sólo se conectan a sus transiciones; si esto no se produjese será síntoma de algún mal funcionamiento en el protoproceso.

Si un protoproceso contiene todos los lugares del p-invariante monomarcado que lo genera, es decir, es cerrado (proposiciones 4.3.5 y 4.3.6), siempre habrá una marca en los lugares del p-invariante que, a modo de contador de programa, describen el flujo de control del protoproceso. De este modo el estado del protoproceso es, en todo momento, evidente. Éste es el caso del protoprocesos π_1 de la figura 4.2, generado a partir del p-invariante monomarcado $\{p_1, p_2, p_3, p_4\}$ que define el flujo de control del protoproceso. Sin embargo, esto no sucede si el protoproceso es abierto, aun procediendo de un p-invariante monomarcado, pues no siempre se conserva la marca del p-invariante en lugares privados del protoproceso. Existirán transiciones de salida o de entrada (sección 4.2.3) que retiren o introduzcan la marca en el protoproceso. Véase, por ejemplo, en la misma figura 4.2, el protoproceso π_2, que es abierto, generado a partir del p-invariante monomarcado $\{p_1, p_5, p_6, p_7\}$. Puede apreciarse que no siempre existe una marca que permanezca en sus lugares privados, p_6 y p_7, que sirva para mostrar el estado y el flujo de control del protoproceso. Esto es debido a las transición t_4 que retira la marca del mismo y t_5 que la vuelve a introducir.

Lo que perseguimos es que para cada protoproceso exista un conjunto de lugares privados por los que, a modo de contador de programa, circule continuamente una marca indicando que transiciones pueden ser ejecutadas, o que, como en el caso de los lugares terminales, sean lugares en los que la marca resida sin posibilidad de proseguir. Como hemos visto, esto es trivial para los protoprocesos cerrados, pero no para los abiertos. Por lo tanto para determinar el flujo de control de un protoproceso abierto procederemos a cerrarlo. Por la proposición 4.3.6 sabemos que si existe un p-invariante monomarcado que cubra todas las transiciones de un protoproceso, y solamente éstas, éste será cerrado, por lo que para cerrarlo trataremos de encontrar un p-invariante monomarcado mediante la adición de algún lugar implícito a sus lugares privados. Esto es lo que se ha hecho con el protoproceso π_2 de la figura 4.9.a. Al añadir el lugar marcado p_c, que es implícito, construimos un p-invariante monomarcado $\{p_5, p_6, p_c\}$ que cubre todas las transiciones del protoproceso y solamente éstas, por lo que conseguimos cerrarlo y hacer patente el flujo de control a través de los lugares de este p-invariante.

Si el elemento de partición del que proviene el protoproceso no es un p-invariante de marcado, para determinar su flujo de control habrá que construir uno, como se mostrará en la sección 4.4. Si no es posible hacerlo, se deberá recurrir al estudio de la sensibilización de las transiciones y de las posibles secuencias de disparo en la red, probablemente a través del cálculo del grafo de marcados.

4.3.6. Cierre de protoprocesos abiertos procedentes de p-invariantes monomarcados.

Dada una red de Petri $\mathbf{N} < P, T; Pre, Post; M_0 >$ seguidamente expondremos cómo cerrar un protoproceso abierto $\pi < P', T'; Pre', Post'; M'_0 >$ que se ha generado a partir de un elemento de

partición que es soporte de un p-invariante monomarcado, $\mathcal{EP} = \|Y\|$.

Introduzcamos términos que usaremos en la descripción del método. Dada una transición $t \in T$:

- Diremos que un lugar $p \in P$ *está a la salida de una transición* t si $p \in t^\bullet$ ó $\exists t_k \in t^{\bullet\bullet}$ t.q. p está a la salida de t_k.
- Diremos que un lugar $p \in P$ *está a la entrada de una transición* t si $p \in {}^\bullet t$ ó $\exists t_k \in {}^{\bullet\bullet} t$ t.q. p está a la entrada de t_k.
- Diremos que un lugar $p' \in P$ *está a la salida de un lugar* p si $\exists t_k \in p^\bullet$ t.q. $p' \in t_k^\bullet$.
- Diremos que un lugar $p' \in P$ *está a la entrada de un lugar* p si $\exists t_k \in {}^\bullet p$ t.q. $p' \in {}^\bullet t_k$.

Como el protoproceso π es abierto, tendrá al menos una transición de entrada (t_e) y/o de salida (t_s), pudiendo éstas ser varias e incluso estar cubiertas por distintos lugares. Así mismo se verifica que no contendrá todas las transiciones cubiertas por el p-invariante Y (si lo hiciera sería cerrado, proposición 4.3.6) ni tampoco todos los lugares de $\|Y\|$ (si lo hiciera también sería cerrado). Escojamos una de las transiciones de entrada, t_e, y una de las de salida, t_s, de modo que las elegidas estén conectadas por el camino más corto de entre todas las parejas $t_e - t_s$ que se nos presenten[4]. Si el protoproceso sólo tuviese una transición de salida, sin ninguna de entrada, o una transición de entrada, sin ninguna de salida, se escogerá únicamente ésta. De esta forma podemos dividir los lugares del soporte del p-invariante monomarcado que genera el protoproceso π en dos subconjuntos:

- conjunto de *lugares de* $\|Y\|$ *entre entrada y salida*, P^Y_{ES}, calculado como los lugares de todos los caminos que se pueden formar, partiendo de $p_i \in \|Y\|$ y $p_i \in t_e^\bullet$, siguiendo con $p_j \in \|Y\|$ y p_j a la salida de p_i, y así consecutivamente hasta $p_k \in \|Y\|$ y $p_k \in {}^\bullet t_s$. Si el protoproceso sólo tiene transición de entrada el conjunto se definirá $P^Y_{ES} = \{p \in \|Y\| \,|\, p \text{ está a la salida de } t_e\}$. Análogamente, si sólo tiene transición de salida, $P^Y_{ES} = \{p \in \|Y\| \,|\, p \text{ está a la entrada de } t_s\}$.
- y el resto de lugares, *lugares de* $\|Y\|$ *entre salida y entrada*, $P^Y_{SE} = \|Y\| - P^Y_{ES}$.

Al ser Y un p-invariante, se cumple $Y^T C = 0$, o lo que es lo mismo $\sum_{p_i \in \|Y\|} l(p_i) = 0$, siendo $l(p_i)$ la línea de la matriz C, matriz de incidencia de $\mathbf{N}$, correspondiente al lugar p_i. Supongamos, sin pérdida de generalidad, que los lugares del $\|Y\|$ están ordenados de forma que $P^Y_{SE} = \{p_1, \ldots, p_k\}$ y $P^Y_{ES} = \{p_{k+1}, \ldots, p_q\}$, siendo $q = |\,\|Y\|\,|$. Por tanto puede escribirse:

$$l(p_1) + \cdots + l(p_k) + l(p_{k+1}) + \cdots + l(p_q) = 0, \text{ por lo que } \sum_{p_i \in P^Y_{SE}} l(p_i) = -\sum_{p_i \in P^Y_{ES}} l(p_i) \quad (4.1)$$

Es decir, si añadimos a la red un lugar p_c cuya línea en la matriz de incidencia se calcule como

$$l(p_c) = -\sum_{p_i \in P^Y_{ES}} l(p_i) \quad \text{ó} \quad l(p_c) = \sum_{p_j \in P^Y_{SE}} l(p_j) \quad (4.2)$$

tendremos que la red $\mathbf{N}^c < P', T; Pre', Post'; M'_0 >$, red ampliada con el lugar p_c de tal forma que $P' = P \cup \{p_c\}$ y p_c se conecta como se indica en 4.2:

1. Los lugares $p_c \cup P^Y_{ES}$ son el soporte de un p-invariante, Y^c, de la red $\mathbf{N}^c$ ya que $(Y^c)^T C = l(p_c) + \sum_{p_i \in P^Y_{ES}} l(p_i) = 0$, en virtud de (4.2). El marcado de p_c debe elegirse de modo que el p-invariante Y^c sea monomarcado: será 0 si los lugares P^Y_{ES} ya están marcados, y 1 en caso contrario.

[4] El método descrito también es válido en el caso de que haya varias transiciones descendentes o ascendentes de un mismo lugar que sean de salida o entrada al protoproceso. En este caso se escogerá una cualquiera de entre estas transiciones

2. El lugar p_c es siempre implícito con el marcado calculado en 1. Para demostrar que es implícito [Sil85] demostraremos que:

   a) La fila de la matriz de incidencia C correspondiente a p_c es una combinación lineal de coeficientes racionales positivos de las filas de sus implicantes en la matriz C, $l(p_c) = \sum_{p_i \in \text{implicantes de } p_c} \lambda_i l(p_i)$, $\lambda_i \in \mathbb{Q}^+$. Los implicantes de p_c son los lugares de P^Y_{SE} ya que, por construcción, $l(p_c) = -\sum_{p_i \in P^Y_{ES}} l(p_i)$ y por (4.1) $l(p_c) = \sum_{P^Y_{SE}} l(p_i)$, combinación lineal con coeficientes unidad.

   b) El marcado inicial de p_c es una combinación lineal, con los mismos coeficientes λ_i, de los marcados de los lugares implicantes más una constante racional μ, $M_0(p_c) = \sum_{p_i \in \text{implicantes de } p_c} \lambda_i M_0(p_i) + \mu$, $\lambda_i \in \mathbb{Q}^+$, $\mu \in \mathbb{Q}$. Esta μ será siempre 0 ya que $\forall i,\ \lambda_i = 1$, y además:

      1) si los lugares de P^Y_{ES} están marcados, para que Y^c sea monomarcado, $M_0(p_c) = 0$; como P^Y_{ES} y P^Y_{SE} son un p-invariante monomarcado, si P^Y_{ES} está marcado, P^Y_{SE} no lo puede estar, por lo que $\sum_{p_i \in \text{implicantes de } p_c} M_0(p_i) = 0$, y de ahí $\mu = 0$;

      2) si P^Y_{ES} no están marcados, para que Y^c sea monomarcado $M_0(p_c) = 1$ y P^Y_{SE} estará marcado, por lo que $\sum_{p_i \in \text{implicantes de } p_c} M_0(p_i) = 1$, y de ahí $\mu = 0$.

   c) Según [Sil85] si $\mu \geq 0$ y ${}^\bullet p_c \cap p_c^\bullet = \emptyset$ el lugar es implícito. Por construcción el lugar p_c se conecta como lugar de entrada de las transiciones de entrada y lugar de salida de las transiciones de salida del protoproceso π y éstas no pueden coincidir [5], por lo que ${}^\bullet p_c \cap p_c^\bullet = \emptyset$. Por ello el lugar p_c es implícito.

3. La red $\mathbf{N}^c$ es equivalente a la red a implementar $\mathbf{N}$, pues al ser p_c implícito, el lenguaje de disparos de ambas redes es el mismo. Por tanto implementar $\mathbf{N}$ es lo mismo que implementar $\mathbf{N}^c$.

4. El lugar p_c así construido se conecta a las transiciones de entrada y de salida de π por lo que el p-invariante Y^c, por construcción, cubre todas las transiciones del protoproceso que se quiere cerrar. Si no cubriese ninguna más (lo que puede ocurrir si no se tiene cuidado en la partición de transiciones en conflicto, como en el ejemplo de la siguiente sección), en virtud de la proposición 4.3.6, tendremos un nuevo protoproceso π^c, formado al añadir p_c y los arcos que lo conectan con sus transiciones a π, que será cerrado.

Ejemplos El método de cálculo puede aplicarse a las redes de la figura 4.9.

a. El protoproceso π_2 se genera a partir del p-invariante monomarcado $\{p_1, p_5, p_6, p_7\}$. La transición de entrada es t_5 y la de salida t_4, por lo que $P^Y_{ES} = \{p_6, p_7\}$. El lugar p_c será tal que $l(p_c) = -l(p_6) - l(p_7)$ por lo que se conectará a t_5 como lugar de entrada y a t_4 como lugar de salida. El marcado inicial del lugar deberá ser 1 para que los lugares $\{p_c, p_6, p_7\}$ formen un p-invariante monomarcado, que exactamente cubre las transiciones de π_2^c (π_2 junto con el lugar p_c), por lo que éste es cerrado.

b. El caso de la figura 4.9.b muestra una red con dos protoprocesos que tiene lugares terminales y origen. El segundo de ellos, π_2, es un protoproceso abierto que sólo tiene una transición de entrada. Proviene del p-invariante monomarcado $\{p_1, p_4, p_5, p_6\}$, por lo que $P^Y_{ES} = \{p_5, p_6\}$. El lugar p_c será $l(p_c) = -l(p_5) - l(p_6)$, con marcado 1, y, en este caso, se conectará a la transición t_3 como lugar de entrada. El nuevo p-invariante monomarcado $\{p_5, p_6, p_c\}$ cubre todas las transiciones de π_2^c, y sólo éstas, por lo que π_2 será cerrado.

[5] Notar que si existiese una transición $t \in T'$ que fuera de entrada y de salida simultáneamente se verificaría que $\nexists p \in {}^\bullet t$, por ser transición de entrada y $\nexists p' \in t^\bullet$, por ser transición de salida t.q. $p, p' \in P'$ por lo que se deduciría que t no pertenece al protoproceso $\pi < P', T; Pre', Post' >$

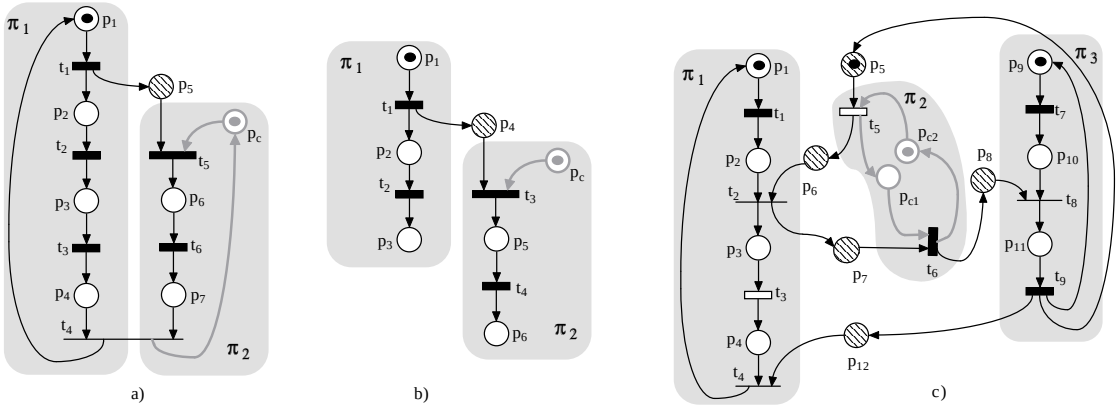


Figura 4.9: a) Red de Petri partida en dos protoprocesos, uno cerrado (π_1) y otro abierto (π_2) que se ha cerrado añadiendo el lugar implícito p_c. b) Red de Petri con lugares terminales partida en dos protoprocesos, uno cerrado (π_1) y otro abierto (π_2) (cerrado al añadir el lugar p_c). c) Otro ejemplo de protoproceso abierto (π_2 cerrado mediante la adición de dos implícitos p_{c1} y p_{c2})

c. En esta red se propone una partición para la red de la figura 4.8 que, independientemente de consideraciones sobre si es la mejor posible, muestra un protoproceso, π_2, generado a partir de un p-invariante monomarcado $Y = \{p_5, p_6, p_7, p_8, p_{11}\}$. El protoproceso no posee lugares privados, aunque si dos transiciones, t_5 y t_6, y está comunicado con los otros protoprocesos por medio de lugares de comunicación asíncrona. Para cerrar este protoproceso habrá que añadir dos lugares implícitos. En primer lugar consideremos que t_5 es la transición de salida y t_6 la transición de entrada. En este caso $P^Y_{ES} = \{p_5, p_8, p_{11}\}$, por lo que $l(p_{c1}) = -l(p_5) - l(p_8) - l(p_{11})$, de modo que p_{c1} se conecta a t_5 como lugar de salida y a t_6 como lugar de entrada. Como los lugares P^Y_{ES} ya están marcados, el marcado inicial de p_{c1} debe ser 0 para que Y^c, formado por $\{p_5, p_8, p_{11}, p_{c1}\}$, sea un p-invariante monomarcado. Sin embargo el protoproceso resultante π^c_2 todavía no está cerrado porque no contiene todas las transiciones cubiertas por este nuevo p-invariante. Por lo tanto debemos aplicar otra vez el método a π^c_2. Ahora la transición de salida será t_6 y t_5 la de entrada. Los conjuntos de lugares serán $P^{Y^c}_{SE} = \{p_5, p_8, p_{11}\}$ y $P^{Y^c}_{ES} = \{p_{c1}\}$, por lo que $l(p_{c2}) = -l(p_{c1})$, resultando que p_{c2} se conecta a t_6 como lugar de salida y a t_5 como lugar de entrada. Como el lugar de $P^{Y^c}_{ES}$ no está marcado, el marcado inicial de p_{c2} deberá ser 1 para que el nuevo p-invariante Y'_c formado por $\{p_{c1}, p_{c2}\}$ sea monomarcado. Ahora el protoproceso $\pi^{c'}_2$ (π_2 junto con p_{c1} y p_{c2}) sí contiene todas las transiciones cubiertas por un p-invariante monomarcado, Y'_c, por lo que es cerrado y el flujo de control está determinado por los lugares de este p-invariante. △

4.3.7. Restricciones en la solución del problema de cobertura. Partición en presencia de conflictos.

Aunque esta cuestión será discutida en detalle en secciones futuras (ver sección 4.7) realizaremos seguidamente un breve comentario sobre el problema de cobertura de las transiciones de la red cuando existen conflictos en ella. Sirva este punto sólo como avance. Consideremos el caso de la figura 4.10. El ejemplo sirve para mostrar que cuando un lugar tiene varias transiciones de salida

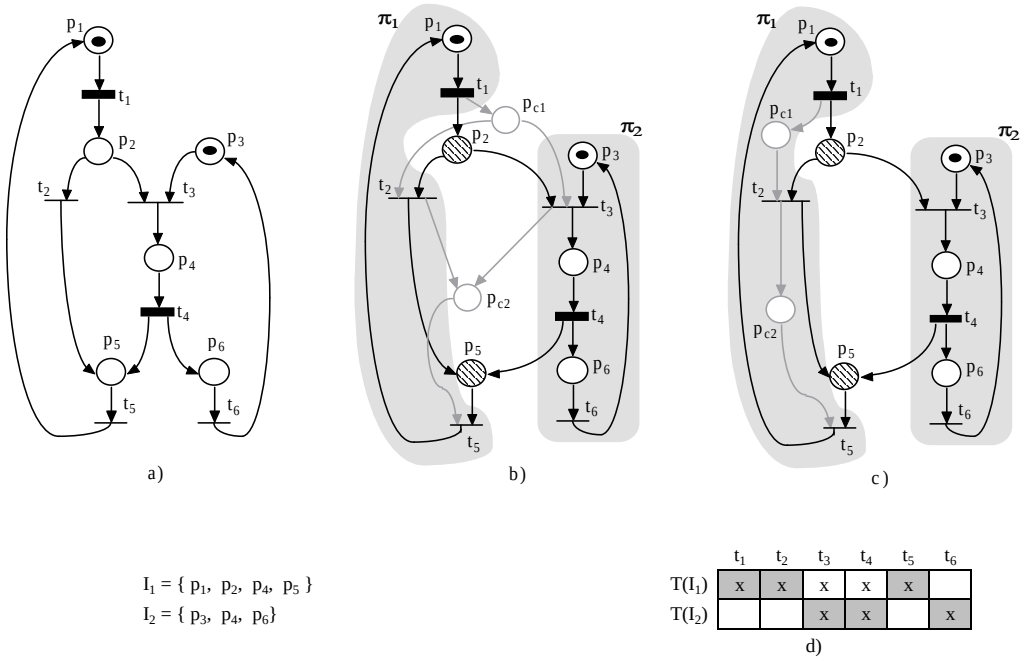


Figura 4.10: a) Red de Petri con un conflicto entre las transiciones t_2 y t_3. b) Partición en la que se ha cerrado el protoproceso π_1. c) Partición en la que se ha cerrado el protoproceso π_1 de forma errónea.

hay que actuar con precaución al realizar la cobertura de las mismas. En la red se muestra un conflicto entre dos transiciones, t_2 y t_3, generado por el lugar p_2.

Existen dos p-invariantes monomarcados $\mathcal{EP}_1 = \{p_1, p_2, p_4, p_5\}$ y $\mathcal{EP}_2 = \{p_3, p_4, p_6\}$ que generan una tabla de partición que se muestra en la figura 4.10.d. La transición t_3 está cubierta por un elemento de partición ($\mathcal{EP}_2$) que no incluye al lugar p_2, lo que no ocurre con t_2. La partición elegida asigna al protoproceso π_1 las transiciones $\{t_1, t_2, t_5\}$ y a π_2 las transiciones $\{t_3, t_4, t_6\}$ de modo que el protoproceso π_1 es abierto. Al intentar cerrar el protoproceso observamos que no es posible hacerlo, porque al calcular los lugares implícitos de cierre p_{c1} y p_{c2} se observa que no son privados a ningún protoproceso. En la figura 4.10.c se muestra la solución que debería obtenerse para lograr cerrar el protoproceso, pero no es válida ya que los lugares añadidos no son implícitos. Evitaremos estas situaciones si colocamos todas las transiciones de un conflicto en el mismo protoproceso o realizando la partición con arreglo a lo que se expondrá en la sección 4.7.

4.4. Partición mediante el cálculo de tuberías.

Hasta ahora hemos planteado la descomposición de la red a través del cálculo de elementos de partición basados en p-invariantes monomarcados. Sin embargo, no siempre es posible encontrar suficientes p-invariantes monomarcados para cubrir todas las transiciones de la red. Un claro ejemplo puede apreciarse en la figura 4.11, en la que se presenta una red que, pese a su aspecto, es secuencial pero no tiene p-invariantes monomarcados. El único p-invariante es

$2m(p_1) + m(p_2) + m(p_3) + m(p_4) + m(p_5)$, que no es monomarcado.

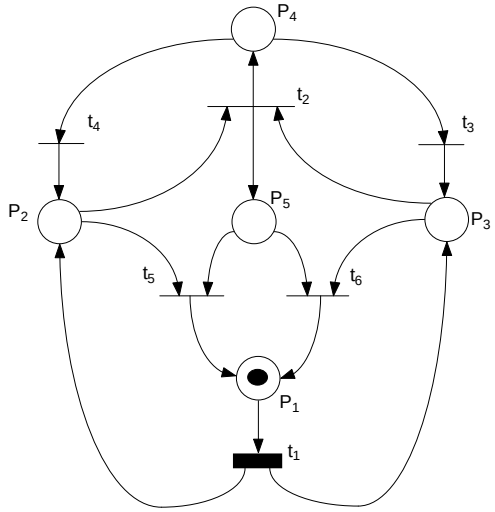


Figura 4.11: Red de Petri que no tiene p-invariantes monomarcados

Se hace, por tanto, necesaria otra técnica que nos permita encontrar elementos de partición en redes sin suficientes p-invariantes monomarcados para cubrir todas sus transiciones. En [Vil90] se presentó una técnica basada en el cálculo de tuberías que será expuesta, modificada y completada en esta memoria.

El concepto de tubería está relacionado con la construcción de p-invariantes monomarcados en redes de Petri que no los poseen, mediante la adición de lugares implícitos. Una vez construidos los p-invariantes monomarcados, será aplicable todo lo expuesto en anteriores secciones.

4.4.1. Concepto de tubería.

En [Vil90] se definió el concepto de tubería partiendo la definición de una estructura en la red que se denominó *T-subred*. Una T-subred era una subred conexa cuyos lugares aportaban todas sus transiciones de entrada y de salida, y no existía ningún lugar sin al menos una transición de entrada y otra de salida. Era, como se puede deducir fácilmente, una subred cuyas fronteras eran transiciones.

En esta memoria se están presentando casos que no serían cubiertos por la anterior definición. Casos en los que las redes a implementar presentan lugares origen o terminales, lugares que quedarían fuera de cualquier T-subred, pues no tienen transiciones de entrada y salida simultáneamente. Por ello realizaremos la exposición del concepto de tubería a partir de otro concepto de subred menos restrictivo.

Sea $\mathbf{N} < P, T; Pre, Post >$ una RdP.

Definición 4.4.1 *Sea* $\mathbf{N}' < P', T'; Pre', Post' >$ *una subred de* $\mathbf{N}$. $\mathbf{N}'$ *es una* **PT-subred** *de* $\mathbf{N}$ *sii* $\mathbf{N}'$ *es una subred de* $\mathbf{N}$, $\mathbf{N}'$ *es conexa y* $\forall p \in P' : {}^\bullet p \cup p^\bullet \in T'$.

Por lo tanto una PT-subred es una subred conexa de $\mathbf{N}$ cuyos lugares aportan todas sus transiciones de entrada y de salida, $T' = T(P')$. El concepto es parecido estructuralmente al de protoproceso, si bien una PT-subred no tiene por qué contener todos los lugares privados a sus transiciones, ni se ha exigido que sus transiciones estén en exclusión mutua, si bien algunas de ellas

(las cerradas monomarcadas) sí las tienen. Del mismo modo que en protoprocesos, pueden existir transiciones de entrada y transiciones de salida de una PT-subred.

Definición 4.4.2 *Sea* **N** *una red de Petri ordinaria.* **N'** *es una* **Estructura no generadora** *(ENG) de* **N** *sii* $\mathbf{N}'$ *es una PT-subred de* **N** *y* $\forall t \in T' : |t^\bullet \cap P'| \leq 1$.

Definición 4.4.3 *Sea* **N** *una red de Petri ordinaria.* **N'** *es una* **Estructura no consumidora** *(ENC) de* **N** *sii* $\mathbf{N}'$ *es una PT-subred de* **N** *y* $\forall t \in T' : |{}^\bullet t \cap P'| \leq 1$.

Por tanto, dentro de una estructura no generadora no se producen marcas ya que todas sus transiciones tienen como máximo un lugar de salida. Del mismo modo, dentro de una estructura no consumidora no se consumen marcas ya que todas sus transiciones tienen como máximo un lugar de entrada.

Definición 4.4.4 (Tubería) *Sea* **N** *una red de Petri ordinaria.* $\mathbf{N}'$ *es una tubería de* **N** *sii* $\mathbf{N}'$ *es una ENG y una ENC simultáneamente.*

En una tubería también se puede hablar de transiciones de entrada ($\nexists p \in P'$ t.q. $p \in {}^\bullet t_e$) y transiciones de salida ($\nexists p \in P'$ t.q. $p \in t_s^\bullet$).

Definición 4.4.5 N' *es una* tubería cerrada $\iff$ $\mathbf{N}'$ *es una tubería y no tiene transiciones de entrada ni de salida. En caso contrario hablaremos de* tubería abierta.

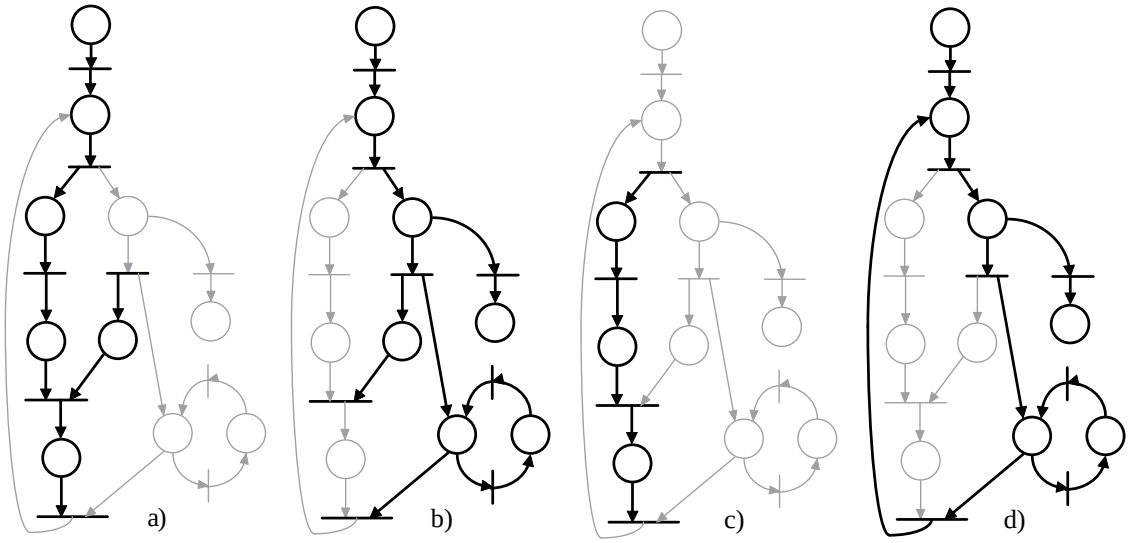


Figura 4.12: a) Estructura no generadora (ENG); b) estructura no consumidora (ENC); c) tubería abierta; d) tubería cerrada.

En conclusión, una tubería es una estructura en la que no se producen ni consumen marcas, por lo que una vez que una marca entra en ella a través de una de sus transiciones de entrada, fluye por ella hasta alcanzar una de sus transiciones de salida, única posibilidad de abandono de la tubería. En la figura 4.12 se aprecian ejemplos de aplicación de las definiciones anteriores a una red de Petri.

Es inmediato deducir que una tubería cerrada mantiene su marcado constante, pues carece de transiciones de entrada y de salida, únicos elementos por donde entran o salen marcas. Por tanto

es natural pensar que el concepto de tubería esté relacionado con el de p-invariante. En [Vil90] se demuestra que en una red de Petri marcada y ordinaria, *una tubería cerrada monomarcada es una máquina de estados y por esta razón define un p-invariante.* El resultado no se reproduce aquí, a pesar del cambio introducido porque éste no afecta a la demostración.

En particular, si la tubería cerrada resulta ser monomarcada, se obtiene un p-invariante monomarcado, cuyo soporte,será un elemento de partición a considerar para la resolución del problema de cobertura de la red. Además la siguiente propiedad mostrará que un p-invariante monomarcado de una red de Petri cuyas transiciones pueden dispararse al menos una vez, es una tubería cerrada y monomarcada de la misma.

Propiedad 4.4.1 *Sea* $\mathbf{N} < P, T; Pre, Post; M_0 >$ *una RdP marcada y ordinaria, cuyas transiciones pueden dispararse al menos una vez. Sea* $Y \in \{1,0\}^n$ *un p-invariante monomarcado elemental de* $\mathbf{N}$*, siendo* $n = |P|$*. Entonces la subred formada por* $< \|Y\|, T(\|Y\|); Pre', Post'; M'_0 >$*, siendo* $Pre' = Pre|_{\|Y\| \times T(\|Y\|)}$*,* $Post' = Post|_{T(\|Y\|) \times \|Y\|}$ *y* $M'_0 = M_0|_{\|Y\|}$*, es una tubería cerrada y monomarcada de* $\mathbf{N}$*.*

Demostración. Por ser Y un p-invariante elemental la subred será conexa, pues si no lo fuese podría ser descompuesto en otros p-invariantes, por lo que Y no sería elemental. Por ser Y un p-invariante $< \|Y\|, T(\|Y\|); Pre', Post'; M'_0 >$ será una subred cerrada ya que todas las transiciones cubiertas por los lugares de un p-invariante tienen a su entrada y a su salida lugares del p-invariante. De no ser así, las marcas "escaparían"del p-invariante. Por ello $\forall t \in T(\|Y\|)$, $| \, {}^\bullet t \cap \|Y\| \, | > 0$ y $| \, t^\bullet \cap \|Y\| \, | > 0$.

Veamos ahora que la subred es una ENG y ENC.

- Al ser monomarcado $\forall t \in T(\|Y\|)$ se verificará que $| \, {}^\bullet t \cap \|Y\| \, | = 1$. Supongamos que no fuese así, por lo que $\exists t \in T(\|Y\|)$ t.q. $| \, {}^\bullet t \cap \|Y\| \, | \neq 1$. Supongamos que $| \, {}^\bullet t \cap \|Y\| \, | > 1$, es decir una transición tiene más de un lugar de entrada perteneciente al p-invariante. Al haberse exigido que las transiciones de la red puedan ser disparadas al menos una vez se debe cumplir que $\exists M \in R(\mathbf{N}, M_0)$ t.q. $\forall p \in {}^\bullet t, M(p) \geq 1$, por lo que aquellos lugares de $\|Y\|$ que son lugares de entrada de t estarán simultáneamente marcados, lo cual es contradictorio, pues, de ser así Y no sería un monomarcado.

- Del mismo modo $| \, t^\bullet \cap \|Y\| \, | = 1$. Si no fuese así alguna transición tendría más de un lugar de salida perteneciente al p-invariante, $\exists t \in T(\|Y\|)$ t.q. $| \, t^\bullet \cap \|Y\| \, | > 1$, por lo que cuando la transición t sea disparada todos sus lugares de salida serán marcados, incluidos aquellos que pertenezcan al $\|Y\|$, lo cual, por el mismo motivo que el descrito anteriormente, es contradictorio.

Por lo tanto $< \|Y\|, T(\|Y\|); Pre', Post'; M'_0 >$ es una tubería, y al ser Y monomarcado, la tubería será monomarcada. △

De los dos resultados mostrados, la propiedad 6.6.1 que aparece en [Vil90] y el de la propiedad 4.4.1, se deduce la siguiente proposición:

Proposición 4.4.1 *Sea* $\mathbf{N} < P, T; Pre, Post; M_0 >$ *una RdP marcada y ordinaria, cuyas transiciones pueden dispararse al menos una vez. Sea* $Y \in \{1,0\}^n$*, con* $n = |P|$*, el vector característico de un conjunto de lugares 1-limitados de* P*. La subred formada por* $< \|Y\|, T(\|Y\|); Pre', Post'; M'_0 >$*, con* $Pre' = Pre|_{\|Y\| \times T(\|Y\|)}$*,* $Post' = Post|_{T(\|Y\|) \times \|Y\|}$ *y* $M'_0 = M_0|_{\|Y\|}$*, es una tubería cerrada monomarcada de* $\mathbf{N}$ $\iff$ *Y es un p-invariante monomarcado de* $\mathbf{N}$ $(Y^T C = 0$ *y* $Y^T M_0 = 1)$*.*

Demostración. La demostración es inmediata a partir de la propiedad 4.4.1 y la propiedad 6.6.1 de [Vil90], que demuestran, cada una, una de las implicaciones. △

Por lo tanto, en redes ordinarias en las que las transiciones pueden dispararse al menos una vez (como son las redes que proponemos para nuestros modelos de sistemas de tiempo real, ver sección 2.2.4), el concepto de tubería cerrada monomarcada y de p-invariante monomarcado son equivalentes, de lo que se concluye que *una tubería cerrada monomarcada es un elemento de partición.*

4.4.2. Cierre de tuberías abiertas

Como ya hemos comentado, no siempre es posible cubrir todas las transiciones de una red con p-invariantes monomarcados, es decir, tuberías cerradas monomarcadas (sirva de ejemplo la red de la figura 4.11). Sin embargo, sí es posible hacerlo con tuberías abiertas (considérese que una transición por sí misma es una tubería abierta). Por tanto es interesante, según señala [Vil90], desarrollar un método que permita obtener una tubería cerrada a partir de una abierta. El procedimiento descrito en [Vil90] está basado en la adición de un lugar comportamentalmente implícito que cierre la tubería. Su cálculo se basa en la siguiente proposición:

Sean

- $\mathbf{N} < P, T; Pre, Post; M_0 >$ una RdP
- $\mathcal{TB}_1 < P_1, T_1; Pre_1, Post_1 >$ una tubería de $\mathbf{N}$
- p_c un lugar cuya línea en la matriz de incidencia de $\mathbf{N}$ se calcula como la suma cambiada de signo de las líneas de los lugares de la tubería, $l(p_c) = -\sum_{p_i \in P_1} l(p_i)$.
- $\mathbf{N}' < P', T; Pre', Post' >$ la RdP ampliada con el lugar p_c de tal forma que $P' = P \cup \{p_c\}$ y $Pre' = Pre|_{P' \times T}$ y $Post' = Post|_{T \times P'}$.

entonces la red $\mathcal{TB}_1^* < P_1^*, T_1; Pre_1^*, Post_1^* >$, con $P_1^* = P_1 \cup \{p_c\}$ y $Pre_1^* = Pre|_{P_1^* \times T_1}$ y $Post_1^* = Post|_{T_1 \times P_1^*}$ es una tubería cerrada de la red ampliada $\mathbf{N}'$.

Si el lugar p_c con que se cierra la tubería es comportamentalmente implícito (ver anexo A) entonces habremos construido una red con el mismo lenguaje de disparos que $\mathbf{N}$, por lo que ambas redes serán equivalentes. Si $\mathbf{N}$ es limitada, el lugar p_c es por construcción estructuralmente implícito [Col89]. La demostración se basa en un teorema de [Col89] que asegura que un lugar $p \in P$ será estructuralmente implícito si $\exists Y \geq 0$ t.q. $Y^T C \geq l(p)$. Es evidente que un lugar calculado como $l(p) = \sum_{p_i \in P_1} l(p_i)$ cumple el teorema anterior, por lo que es estructuralmente implícito. Si este lugar es limitado, que lo será si la red es limitada, el complementario $\overline{p} = -\sum_{p_i \in P_1} l(p_i)$ también será estructuralmente implícito. El lugar complementario es precisamente el lugar de cierre p_c. Dicho lugar será comportamentalmente implícito, y por tanto implícito, si elegimos de forma adecuada el marcado inicial de p_c. Ésta es la principal limitación de la técnica ya que en algunos casos no es sencillo encontrar el marcado inicial para que el lugar sea implícito, y, una vez encontrado, éste no tiene por qué ser 1, por lo que la tubería cerrada con él, no tiene por qué ser monomarcada.

Nótese que si el lugar de cierre es implícito y su marcado es tal que se verifica que $m_o(p_c) + \sum_{p_i \in P_1} p_i = 1$, habremos construido una tubería cerrada monomarcada y por tanto, tal como pretendíamos, un p-invariante monomarcado. En el mismo trabajo [Vil90] se enuncia y demuestra otra proposición para el cierre de varias tuberías abiertas inicialmente disjuntas. El procedimiento de cierre es el mismo, procediéndose al cálculo del lugar de cierre del mismo modo que en la proposición anterior.

Definición 4.4.6 (Tubería 1-limitada) *A las tuberías abiertas que se pueden cerrar con un lugar implícito y que a su cierre son monomarcadas las llamaremos* tuberías 1-limitadas.

Propiedad 4.4.2 *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada y ordinaria, y* $\mathcal{TB} < P', T'; Pre', Post' >$ *una tubería 1-limitada de* $\mathbf{N}$. *Entonces los lugares de* $\mathcal{TB}$ *constituyen un elemento de partición de* $\mathbf{N}$.

Demostración. Al ser $\mathcal{TB}$ una tubería 1-limitada, puede ser cerrada con un lugar implícito (p_c) que define un p-invariante monomarcado. Por ello, en virtud de la proposición 4.3.4, los lugares de $\mathcal{TB}$ junto con el lugar implícito definen un elemento de partición, por lo que están en exclusión mutua. Como el lugar de cierre es implícito puede ser obviado sin afectar a ninguna de las propiedades comportamentales de la red, y en concreto no afectará a la exclusión mutua de los lugares de la tubería, que seguirá manteniéndose, por lo que éstos seguirán siendo un elemento de partición. △

Aunque la propiedad anterior garantiza que para definir un elemento de partición son suficientes los lugares de la tubería 1-limitada, para poder aplicar el resultado sobre construcción del flujo de control en un protoproceso descrito en el punto 4.3.5, necesitamos p-invariantes monomarcados, de modo que las tuberías 1-limitadas siempre serán consideradas junto con sus respectivos lugares de cierre.

Una observación importante es que siempre se puede encontrar un conjunto de tuberías 1-limitadas que cubra todas las transiciones de la red. Si consideramos que cada transición de la red es una tubería abierta, cada una de estas tuberías puede cerrarse añadiendo un lugar marcado que sea entrada y salida de la transición. El conjunto lugar-transición así construido es una tubería cerrada y monomarcada. A esta solución le llamaremos *solución trivial*.

Ejemplo Como ejemplo de aplicación de la técnica de cierre de tuberías abiertas, consideremos el caso de la red de la figura 4.11. En ella se puede encontrar una tubería abierta ($\mathcal{TB} = \{p_1, p_2, p_4; t_1, t_2, t_3, t_4, t_5, t_6\}$) cuyas transiciones frontera son t_3 y t_6 (ver figura 4.13). La tubería se ha cerrado añadiendo un lugar implícito p_c, de modo que se ha construido una tubería cerrada monomarcada ($\mathcal{TB}^c = \{p_1, p_2, p_4, p_c; t_1, t_2, t_3, t_4, t_5, t_6\}$) y, por tanto, un p-invariante monomarcado $\{p_1, p_2, p_4, p_c\}$. Este p-invariante será el elemento de partición que genere y describa el flujo de control del protoproceso que implementa la red. Todos los lugares de la red son privados al protoproceso. △

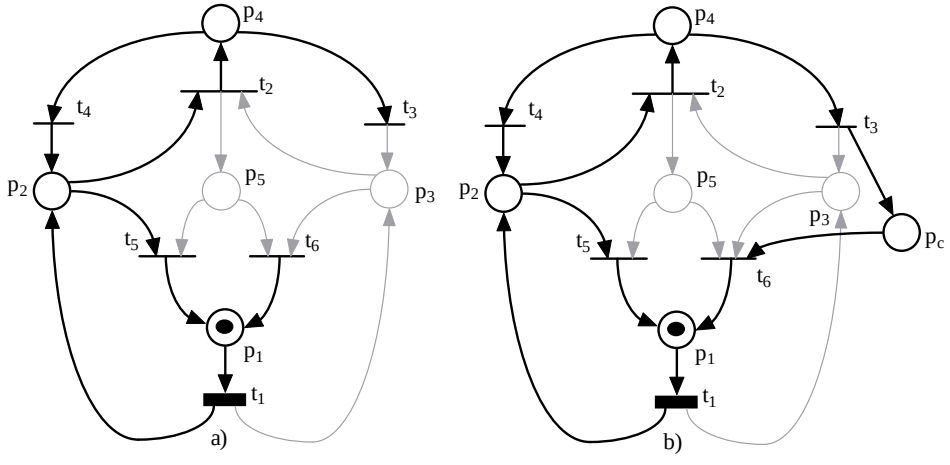


Figura 4.13: a) Tubería abierta (en negrita) en una RdP que no tiene p-invariantes monomarcados; b) el lugar p_c cierra la tubería

Aunque el problema de partición de la red en protoprocesos siempre tenga solución en términos de tuberías 1-limitadas (al menos la solución trivial), no se tienen garantías de que la solución encontrada sea óptima en términos de correspondencia con la concurrencia real de la red, de modo

que podría alcanzarse una solución con más protoprocesos que los necesarios.

Como conclusión, podemos decir que los soportes de los p-invariantes construidos por medio de tuberías cerradas monomarcadas, o por el cierre tuberías 1-limitadas, junto con los propios p-invariantes de la red, definirán conjuntos de lugares en exclusión mutua que serán utilizados como elementos de partición para la solución del problema de cobertura de las transiciones de la red de acuerdo a lo descrito en el punto 4.3.

4.4.3. Cálculo de tuberías

Siguiendo con los resultados de [Vil90], a continuación se muestra cómo calcular las tuberías de una red. El enunciado de la proposición en la que se basa este cálculo se reproduce seguidamente, ligeramente modificado:

Proposición 4.4.2 *Sea* $\mathbf{N} < P, T; Pre, Post; M_0 >$ *una RdP marcada y ordinaria, siendo* $n = |P|$. *Sean* C^+ *y* C^- *las matrices de incidencia previa y posterior de la red* $\mathbf{N}$. *Sea* $Y \in \{1,0\}^n$ *el vector característico de un conjunto de lugares de* P. *Supóngase el siguiente sistema de ecuaciones:*

$$Y^T C^+ \leq \mathbb{1}$$
$$Y^T C^- \leq \mathbb{1}$$

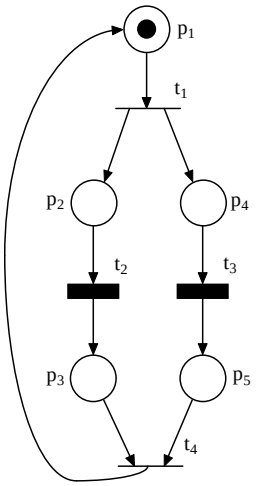
Entonces, aquellas PT-subredes $\mathbf{N}^* < P^*, T^*; Pre^*, Post^* >$ *de* $\mathbf{N}$ *que cumplan que* $P^* = \|Y\|$, $T^* = T(\|Y\|)$, $Pre^* = Pre|_{P^* \times T^*}$ *y* $Post^* = Post|_{T^* \times P^*}$, *siendo* Y *una solución del sistema anterior, y que sean conexas, son tuberías de* $\mathbf{N}$.

La primera de las ecuaciones de la proposición impedirá que en las soluciones aparezcan dos o más lugares descendentes de la misma transición (estructura generadora) y la segunda que aparezcan dos o más lugares de entrada a la misma transición (estructura consumidora). Ya que nuestros objetivos son reducir el número de protoprocesos y encontrar p-invariantes monomarcados, las soluciones han de cumplir además los siguientes requisitos:

1. Ser máximas (contener el mayor número de lugares posible).
2. Estar compuestas por lugares 1-limitados y corresponder con tuberías cerradas monomarcadas, o abiertas que a su cierre resulten monomarcadas (nuestro objetivo sigue siendo encontrara elementos de partición)
3. Deben ser descartadas las soluciones no conexas, descomponiéndolas en subsoluciones conexas que cumplan los requisitos anteriores.

Ejemplo Ilustraremos el proceso de cálculo con un ejemplo que puede verse en la figura 4.14. La aplicación de las ecuaciones de cálculo a la red da como resultado dos restricciones: $Y^T C^+ \leq \mathbb{1}$ impone que $y_2 + y_4 \leq 1$ (con lo que en ninguna solución podrán aparecer simultáneamente p_2 y p_4) e $Y^T C^- \leq \mathbb{1}$ impone que $y_3 + y_5 \leq 1$ (siendo imposible que p_3 y p_5 aparezcan a la vez en alguna solución). Con estas premisas las soluciones máximas serán: (11100) y (10011), que representan tuberías cerradas, y (11001) y (10110) que representan tuberías abiertas que deben ser descartadas porque a su cierre no son monomarcadas. △

El sistema de ecuaciones de la proposición 4.4.2 siempre tendrá solución si la red es ordinaria, como son las redes consideradas en esta memoria. Para demostrar esta afirmación se debe considerar que una solución cuyo soporte sea un único lugar siempre será solución de ambas ecuaciones si en las matrices C^+ y C^- solamente aparecen unos y ceros, tal como ocurre en redes ordinarias. Sin embargo, esta solución puede no ser válida al no cumplir el resto de requisitos, a no ser que el lugar



$$C = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{array} \begin{pmatrix} t_1 & t_2 & t_3 & t_4 \\ -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}$$

$$C^+ = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad C^- = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figura 4.14: Red de Petri y su matrices de incidencia, incidencia previa y posterior.

sea 1-limitado, en cuyo caso siempre será una tubería 1-limitada. Un lugar con sus transiciones de entrada y salida no tiene por qué definir una tubería 1-limitada. En ausencia de otras soluciones, la solución trivial $Y = 0$ será la única válida, y corresponderá con la implementación, ya comentada, de una tubería por cada transición.

El criterio de maximalidad de cada una de las soluciones del sistema de ecuaciones de la proposición 4.4.2 no siempre es compatible con el requisito de ser monomarcada. Como muestra, considérese la red de la figura 4.15 en la cual las tuberías máximas de las figuras 4.15.b y 4.15.c no son monomarcadas por lo que no pueden utilizarse como elementos de partición. Remarquemos, por tanto, que nuestro objetivo es encontrar no cualquier tubería máxima, sino aquellas tuberías, lo más grandes posible, que sean cerradas y monomarcadas, o que a su cierre sean monomarcadas (1-limitadas). En el ejemplo, la tubería 1-limitada más grande se muestra en las figura 4.15.f.

En resumen, no se requiere una maximalidad en términos absolutos, sino junto con la condición de monomarcada. Para evitar confusiones, nos referiremos a ellas como tuberías "más grandes"en vez de máximas. Esto hará que las tuberías buscadas sean PT-subredes de las tuberías máximas. Con el fin de evitar este problema en el cálculo de las tuberías mediante la proposición 4.4.2 en el punto siguiente presentaremos un algoritmo de cálculo de las tuberías monomarcadas más grandes en una red. El algoritmo es incremental de modo que el cálculo de la tubería se detiene en el momento en que se detecte que no es monomarcada o que es cerrada.

4.4.4. Algoritmo para el cálculo de tuberías

El algoritmo está basado en el que parece en [Vil90] y difiere de él en algunos detalles que serán comentados más adelante. Sea $\mathbf{N} < P, T; Pre, Post; M_0 >$ una RdP marcada y ordinaria. El objetivo del algoritmo es calcular todas las tuberías $\mathcal{TB} < P_{\mathcal{TB}}, T_{\mathcal{TB}}; Pre_{\mathcal{TB}}, Post_{\mathcal{TB}}; M_{0\mathcal{TB}} >$ más grandes cerradas o que a su cierre sean monomarcadas de la red $\mathbf{N}$. El cálculo de cada tubería se realiza de modo incremental ampliando en cada paso una tubería 1-limitada a partir de una de sus transiciones de entrada o de salida.

1. La construcción comienza por la elección de cualquier lugar 1-limitado de la red (al que llamaremos semilla, p_s (ver (2) en el algoritmo). En esta elección se preferirán los lugares marcados a los no marcados, ya que si la tubería está marcada el proceso se simplifica

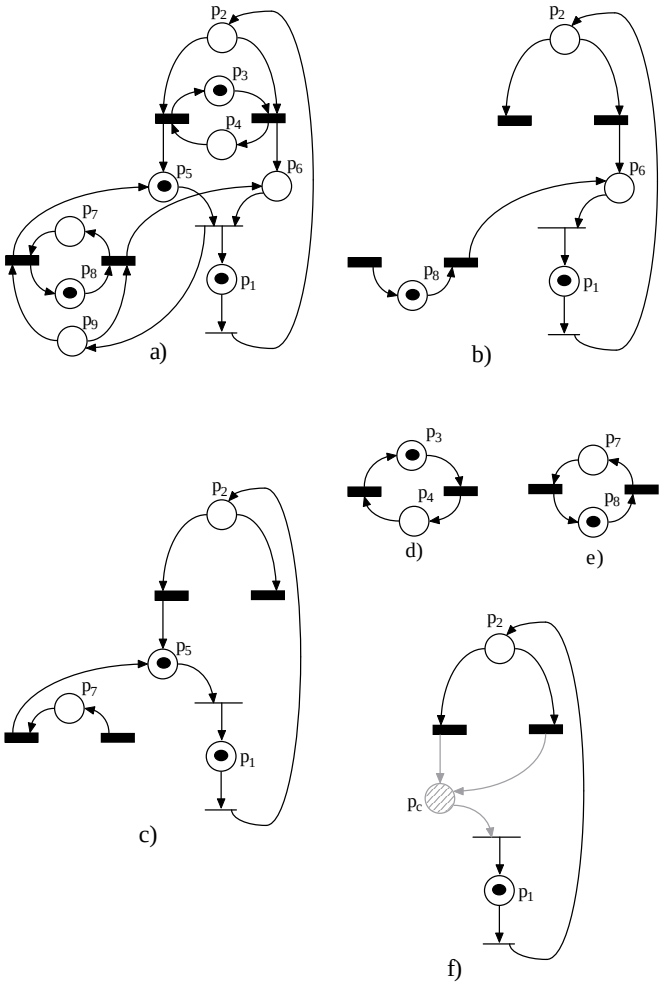


Figura 4.15: a) Red de Petri. b) c) d) y e) Tuberías máximas de la red. f) Tubería más grande que a su cierre es monomarcada.

pudiendo descartar para la ampliación lugares marcados.

2. En cada paso (ver (3) y (13)) se calculan las transiciones de entrada ($T_e^{\mathcal{TB}}$) y de salida ($T_s^{\mathcal{TB}}$) de la tubería. La ampliación se realiza a partir de una de estas transiciones (4) eligiendo uno de sus lugares de entrada o de salida (5,7) que sean 1-limitados, verificando si la PT-subred así ampliada es una tubería (8) y si resulta ser cerrada (9) o, en su defecto, si una vez cerrada es monomarcada (12). Si es así, la tubería se ampliará con ese lugar. Si no, se intentará con otro lugar (14) u otra transición de entrada o salida (15).

3. La ampliación de cada tubería finaliza cuando se obtiene una tubería cerrada monomarcada o una tubería abierta que a su cierre es monomarcada y que no puede ser ampliada con ningún lugar.

El proceso se repite para cada nueva tubería hasta que no queden más lugares semilla por los que comenzar tuberías.

Algoritmo 4.4.1 Algoritmo de cálculo de tuberías

algoritmo Cálculo de tuberías
variables
$Y_{\mathcal{TB}} \in (\{0,1\})^n$, con $n = |P|$ de modo que $P_{\mathcal{TB}} = \|Y_{\mathcal{TB}}\|$ −− Vector representante de la tubería
$Y_{p_a} \in (\{0,1\})^n$ −− Vector característico del lugar de ampliación
$Y_{p_c} \in (\{0,1\})^n$ −− Vector característico del lugar de cierre
$T_e^{\mathcal{TB}}$ −− Conjunto de las transiciones de entrada a la tubería candidatas para la ampliación
$T_s^{\mathcal{TB}}$ −− Conjunto de las transiciones de salida de la tubería candidatas para la ampliación
$P_{semilla}$ −− Lugares semilla para el comienzo de cada tubería
$P_{candidatos}$ −− Conjunto de lugares de ampliación de la tubería
p_s, p_a, p_c −− Lugares de comienzo, de ampliación y de cierre, respectivamente, de la tubería
principio
(1) $P_{semilla} = P$ - lugares no 1-limitados
mientras que $P_{semilla} \neq \emptyset$
(2) Elegir $p_s \in P_{semilla}$, lugar de comienzo de la tubería y que sea tubería 1-limitada por sí mismo
$Y_{\mathcal{TB}}(p_s) = 1 \ \wedge \ Y_{\mathcal{TB}}(p') = 0 \ \forall p' \in P, p' \neq p_s$
(3) $T_e^{\mathcal{TB}} = {}^\bullet p_s - {}^\bullet p_s \cap p_s^\bullet$; $T_s^{\mathcal{TB}} = p_s^\bullet - {}^\bullet p_s \cap p_s^\bullet$

−− Ampliación de la tubería
mientras que $T_e^{\mathcal{TB}} \cup T_s^{\mathcal{TB}} \neq \emptyset$ **and not** AMPLIADA
(4) Elegir $t \in T_e^{\mathcal{TB}} \cup T_s^{\mathcal{TB}}$, transición a partir de la cual se ampliará la tubería
(5) $P_{candidatos} = \begin{cases} {}^\bullet t & \text{si} \quad t \in T_e^{\mathcal{TB}} \\ t^\bullet & \text{si} \quad t \in T_s^{\mathcal{TB}} \end{cases}$ − lugares no 1-limitados
(6) **si** $Y_{\mathcal{TB}}^T M_0 = 1$ **entonces** $P_{candidatos} = P_{candidatos} - \{p \in P_{candidatos} \mid M(p) = 1\}$

−−Añadir lugar
mientras que $P_{candidatos} \neq \emptyset$ **and not** AÑADIDO
(7) Elegir $p_a \in P_{candidatos}$, lugar a partir de la cual se ampliará la tubería
$Y_{p_a}(p_a) = 1 \ \wedge \ Y_{p_a}(p') = 0 \, \forall p' \in P, p' \neq p_a$
(8) **si** $(Y_{\mathcal{TB}} + Y_{p_a})^T C^+ \leq 1$ **and** $(Y_{\mathcal{TB}} + Y_{p_a})^T C^- \leq 1$ −− Si cumple la condición
(9) **si** $(Y_{\mathcal{TB}} + Y_{p_a})^T C = 0$ −− Si la tubería es ya cerrada
(10) **si** $(Y_{\mathcal{TB}} + Y_{p_a})^T M_0 = 1$ −− Si es monomarcada
$Y_{\mathcal{TB}} = Y_{\mathcal{TB}} + Y_{p_a}$ −− Añade lugar a la tubería
AÑADIDO = CIERTO; AMPLIADA = CIERTO −− Tubería calculada
fsi
sino
$Y_{p_c} = -(Y_{\mathcal{TB}} + Y_{p_a})^T C$ −− Cálculo del lugar de cierre
(11) Cálculo del marcado inicial de p_c ($M_0(p_c)$) para que éste sea implícito
(12) **si** $(Y_{\mathcal{TB}} + Y_{p_a})^T M_0 + M_0(p_c) = 1$ −− Si la tubería así cerrada es monomarcada
$Y_{\mathcal{TB}} = Y_{\mathcal{TB}} + Y_{p_a}$ −− Añade lugar a la tubería
AÑADIDO = CIERTO
(13) $T_e^{\mathcal{TB}} = \{t \mid t \in {}^\bullet Y_{\mathcal{TB}} \wedge t \notin Y_{\mathcal{TB}}^\bullet\}$ y $T_s^{\mathcal{TB}} = \{t \mid t \in Y_{\mathcal{TB}}^\bullet \wedge t \notin {}^\bullet Y_{\mathcal{TB}}\}$
fsi
fsi
fsi
(14) $P_{candidatos} = P_{candidatos} - \{p_a\}$
fmq −−Añadir lugar
(15) **si not** AÑADIDO **entonces** $T_e^{\mathcal{TB}} = T_e^{\mathcal{TB}} - \{t\}$; $T_s^{\mathcal{TB}} = T_s^{\mathcal{TB}} - \{t\}$
fmq −− Ampliación de la tubería
$Y_{\mathcal{TB}}$ es una tubería
(16) $P_{semilla} = P_{semilla} - p_s$
fmq
fin

Comentarios sobre el algoritmo

- Al principio del algoritmo es posible que cualquier lugar 1-limitado de la red sea el lugar de comienzo o semilla de la tubería (1).

- Para la elección de los lugares de ampliación de la tubería (conjunto $P_{candidatos}$) se puede atender a varios criterios según se quiera o no obtener tuberías disjuntas. Si éste es el caso, ninguno de los lugares ya incluidos en otras tuberías deberá ser candidato para la ampliación de otra. Además, cada vez que se obtenga una tubería, se deben eliminar del conjunto de lugares semilla ($P_{semilla}$) todos los lugares de la tubería ($P_{semilla} = P_{semilla} - \|Y_{\mathcal{TB}}\|$ (16)). Ésta sería la solución más rápida computacionalmente para encontrar un conjunto de tuberías que cubra toda la red. Sin embargo, si se desean encontrar todas las tuberías 1-limitadas más grandes de la red, incluso cuando tienen intersección no vacía entre sí, se deben considerar como lugares candidatos de ampliación todos los lugares de la red. De esta forma es posible encontrar todas las tuberías. El algoritmo mostrado responde a esta política de resolución. En cualquiera de los dos casos, si durante el proceso de cálculo, la tubería está marcada no se deben considerar como candidatos para ampliación lugares marcados, ya que la tubería resultante no sería monomarcada (6).

- (3) y (13) Las transiciones no puras no deben considerarse como transiciones de ampliación de una tubería ya que, o bien no tienen ningún otro lugar de entrada o salida, en cuyo caso no es posible ampliar a partir de ellas, o, si sí los tienen, ninguno de ellos podría ser incluido en la tubería ya que se trataría de una estructura generadora o consumidora.

- (11) El cálculo del marcado de lugar de cierre de la tubería para que éste sea implícito puede simplificarse si se considera el marcado de la tubería en ese momento. Si ésta ya está marcada, sólo hay que comprobar si el lugar de cierre es implícito con marcado cero, ya que cualquier otro marcado haría que, al cierre, la tubería no fuera monomarcada. Si la tubería no está marcada, el único marcado que se debe comprobar para el lugar de cierre es uno, ya que queremos construir una tubería monomarcada. En principio ningún marcado mayor que uno sería válido ya que daría lugar a una tubería no monomarcada. Un marcado cero en este caso tampoco tiene sentido, pues construiríamos una tubería cerrada sin marcado.

- La fase de ampliación de la tubería podría mejorarse. Durante la ampliación a partir de una determinada transición algunos de los lugares candidatos de ampliación generados a partir de ella pueden no cumplir la condición de tubería. Si el algoritmo fuese capaz de recordar estos lugares se evitaría que volvieran a aparecer como lugares candidatos a partir de otra transición.

- Notar que la conectividad de las soluciones está garantizada por el método constructivo, así como que las tuberías resultantes son monomarcadas.

El algoritmo anterior no garantiza que se calculen todas las tuberías de una red de Petri, aunque si que se obtiene un conjunto de ellas que cubren todas las transiciones de la misma. Otro posible algoritmo, basado en el anterior, que realiza un cálculo extensivo de tuberías en la red se muestra a continuación. Se basa en el empleo de conjuntos de tuberías en formación, en el que se almacenan todas las tuberías en proceso de cómputo. En cada paso se comprueban todas las posibilidades de ampliación, en cuanto a transiciones candidatas y lugares candidatos a partir de las anteriores transiciones. Cuando una tubería es cerrada o no puede ampliarse más, es colocada en un conjunto de tuberías definitivas.

Algoritmo 4.4.2 Algoritmo extensivo de cálculo de tuberías

algoritmo Cálculo de tuberías
variables
$Y_{\mathcal{TB}} \in (\{0,1\})^n$, con $n = |P|$ de modo que $P_{\mathcal{TB}} = \|Y_{\mathcal{TB}}\|$ −− Vector representante de la tubería
$Y_{p_a} \in (\{0,1\})^n$ −− Vector característico del lugar de ampliación
$Y_{p_c} \in (\{0,1\})^n$ −− Vector característico del lugar de cierre
$T_e^{\mathcal{TB}}$ −− Conjunto de las transiciones de entrada a la tubería candidatas para la ampliación
$T_s^{\mathcal{TB}}$ −− Conjunto de las transiciones de salida de la tubería candidatas para la ampliación
$P_{semilla}$ −− Lugares semilla para el comienzo de cada tubería
$P_{candidatos}$ −− Conjunto de lugares de ampliación de la tubería
Tuberías −− Conjunto de tuberías de la red
Tuberías_en_formación −− Conjunto de tuberías intermedias en formación
p_s, p_a, p_c −− Lugares de comienzo, de ampliación y de cierre, respectivamente, de la tubería
principio
(1) $P_{semilla} = P$ - lugares no 1-limitados
mientras que $P_{semilla} \neq \emptyset$
(2) Coger $p_s \in P_{semilla}$, lugar de comienzo de la tubería y que sea tubería 1-limitada por sí mismo
$Y_{\mathcal{TB}}(p_s) = 1 \ \wedge \ Y_{\mathcal{TB}}(p') = 0 \ \forall p' \in P, p' \neq p_s$
Tuberías_en_formación = Tuberías_en_formación $\cup\, Y_{\mathcal{TB}}$
mientras que Tuberías_en_formación $\neq \emptyset$
Coger $Y_{\mathcal{TB}} \in$ Tuberías_en_formación
AMPLIADA = FALSO
(3) $T_e^{\mathcal{TB}} = \{t \mid t \in {}^\bullet Y_{\mathcal{TB}} \wedge t \notin Y_{\mathcal{TB}}^\bullet\}$; $T_s^{\mathcal{TB}} = \{t \mid t \in Y_{\mathcal{TB}}^\bullet \wedge t \notin {}^\bullet Y_{\mathcal{TB}}\}$
−− Ampliación de la tubería
mientras que $T_e^{\mathcal{TB}} \cup T_s^{\mathcal{TB}} \neq \emptyset$
(4) Coger $t \in T_e^{\mathcal{TB}} \cup T_s^{\mathcal{TB}}$, transición a partir de la cual se ampliará la tubería
$P_{candidatos} = \begin{cases} {}^\bullet t & \text{si} \quad t \in T_e^{\mathcal{TB}} \\ t^\bullet & \text{si} \quad t \in T_s^{\mathcal{TB}} \end{cases}$ − lugares no 1-limitados
(5) **si** $Y_{\mathcal{TB}}^T M_0 = 1$ **entonces** $P_{candidatos} = P_{candidatos} - \{p \in P_{candidatos} \mid M(p) = 1\}$
−−Añadir lugar
mientras que $P_{candidatos} \neq \emptyset$
(6) Coger $p_a \in P_{candidatos}$, lugar a partir de la cual se ampliará la tubería
$Y_{p_a}(p_a) = 1 \ \wedge \ Y_{p_a}(p') = 0 \, \forall p' \in P, p' \neq p_a$
(7) **si** $(Y_{\mathcal{TB}} + Y_{p_a})^T C^+ \leq \mathbb{1}$ **and** $(Y_{\mathcal{TB}} + T_p)^T C^- \leq \mathbb{1}$ −− Si cumple la condición
(8) **si** $(Y_{\mathcal{TB}} + Y_{p_a})^T C = 0$ −− Si la tubería es ya cerrada
(9) **si** $(Y_{\mathcal{TB}} + Y_{p_a})^T M_0 = 1$ −− Si es monomarcada
AMPLIADA = CIERTO
Tuberías = Tuberías $\cup (Y_{\mathcal{TB}} + Y_{p_a})$ −− Tubería calculada
fsi
sino
$Y_{p_c} = -(Y_{\mathcal{TB}} + Y_{p_a})$ −− Cálculo del lugar de cierre
(10) Cálculo del marcado inicial de p_c $(M_0(p_c))$ para que éste sea implícito
(11) **si** $(Y_{\mathcal{TB}} + Y_{p_a})^T M_0 + M_0(p_c) = 1$ −− Si la tubería así cerrada es monomarcada
Tuberías_en_formación = Tuberías_en_formación $\cup\, (Y_{\mathcal{TB}} + Y_{p_a})$
AMPLIADA = CIERTO
fsi
fsi
fsi
(13) $P_{candidatos} = P_{candidatos} - \{p_a\}$
fmq −−Añadir lugar
$T_e^{\mathcal{TB}} = T_e^{\mathcal{TB}} - \{t\}$; $T_s^{\mathcal{TB}} = T_s^{\mathcal{TB}} - \{t\}$
fmq −− Ampliación de la tubería
fmq
(14) **si not** AMPLIADA **entonces** Tuberías = Tuberías $\cup\, Y_{\mathcal{TB}}$
(15) $P_{semilla} = P_{semilla} - p_s$
fmq
fin

4.5. Proceso de implementación.

Expondremos seguidamente, a modo de resumen, un procedimiento a seguir para la partición de una red de Petri, con los requisitos reseñados en la sección 2.2.4, en un conjunto de protoprocesos. Supondremos que la red no presenta conflictos. Más adelante, en la sección 4.7.5, se indicarán las cosas a tener en cuenta para realizar la partición en presencia de conflictos.

1. Realizar un cómputo de p-invariantes, seleccionando los monomarcados como primeros elementos de partición.
2. Si todas las transiciones de la red están cubiertas por los elementos de partición encontrados en el punto anterior, el proceso continuará a partir del punto 4.
3. Realizar un cómputo de tuberías cerradas monomarcadas o tuberías 1-limitadas cerradas para obtener el resto de elementos de partición para las transiciones no cubiertas por p-invariantes monomarcados. Para este cómputo se debe partir de un conjunto de lugares semilla del que se hayan eliminado todos los lugares ya pertenecientes a algún p-invariante monomarcado.
4. De las anteriores etapas se obtiene un conjunto de elementos de partición compuestos por p-invariantes monomarcados (calculados o construidos) que cubren todas las transiciones de la red. Con ellos construiremos la tabla de cobertura de acuerdo a lo expuesto en la sección 4.3.2.
5. Solucionar el problema de cobertura atendiendo en primer lugar a los elementos de partición esenciales. Además durante la cobertura se deberá utilizar el menor número de elementos de partición posible, con el fin de reducir el número de protoprocesos generados.
6. Construcción del flujo de control de los protoprocesos obtenidos en los dos puntos anteriores. Puede suponer la inclusión de lugares implícitos para cerrar protoprocesos abiertos.
7. Implementación software de cada protoproceso de acuerdo al flujo de control establecido en el punto 6.

4.6. Aplicación a las redes de Petri con tiempo

Hasta ahora no se ha tenido en cuenta las propiedades temporales de la RdPT. Es el momento de realizar algún comentario al respecto. Si bien en términos generales todo lo dicho es válido para cualquier tipo de transición es preciso hacer algún comentario sobre qué sentido tienen ciertas cuestiones con relación a los tipos de transiciones que nos encontramos en una RdPT. De momento dejaremos de lado las referencias a varios tipos de transiciones en conflicto. La implementación software en Ada 95 de estas configuraciones tiene implicaciones sobre la partición de la red y será comentado en el capítulo siguiente.

Una de las cuestiones a comentar es la referente a las transiciones compartidas como medio de sincronización entre protoprocesos. De los tres tipos de transiciones que pueden aparecer en una RdPT, son las transiciones SYCO las que mejor se ajustan al papel de sincronización pura entre protoprocesos. Los otros dos tipos, TIME o CODE, también pueden compartirse, aunque la labor que desarrollada por estas transiciones es más compleja. Actuarán como citas extendidas en las que se suspende la ejecución durante el período de tiempo asociado a la transición TIME, o se ejecuta el código asociado a la CODE (cita extendida).

Además de estas precisiones, el planteamiento del problema de cobertura puede cambiar, al cambiar las propiedades de exclusión mutua entre transiciones si se consideran sus características temporales. Consideremos un ejemplo: la red de la figura 4.16.a trata de modelar un proceso periódico que se activa cada 100 unidades de tiempo. Un cálculo de p-invariantes pone de manifiesto

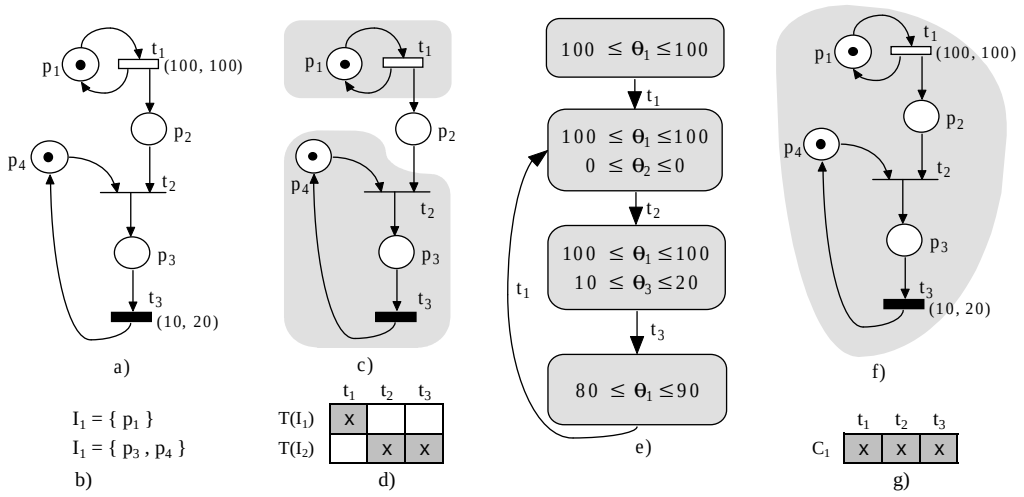


Figura 4.16: a) Red de Petri con tiempo; b) p-invariantes monomarcados; c) Partición sin considerar propiedades temporales; d) tabla de partición. e) Grafo de clases de estado que muestra que las transiciones están en exclusión mutua temporal; f) Partición considerando propiedades temporales; g) Clases máximas de compatibilidad de transiciones en exclusión mutua temporal.

dos de ellos, $I_1 = \{p_1\}$ e $I_2 = \{p_3, p_4\}$, cada uno de los cuales genera un protoproceso (ver figura 4.16.c). En total dos protoprocesos, cuando en realidad se intenta modelar un proceso.

Si tuviéramos en cuenta las propiedades temporales y, en concreto la exclusión mutua temporal entre transiciones[6], nos daríamos cuenta de que las tres transiciones de la red están en exclusión mutua temporal, tal como demuestra el grafo de clases de estado de la figura 4.16.e. Esto implica que existe una única clase de transiciones en exclusión mutua que agrupa todas las transiciones de la red, por lo que sólo es necesario un protoproceso. El principal escollo para la determinación de clases de transiciones en exclusión mutua temporal es la ausencia de herramientas para determinarlas, por lo que es necesario acudir al grafo de clases de estado para concluir sobre este particular. Una posible estrategia de cómputo podría comenzar realizando un cálculo de protoprocesos sin tomar en consideración el tiempo, para luego estudiar la posible fusión entre varios de ellos, si se logra determinar que todas sus transiciones están en exclusión mutua temporal.

Así mismo, una posible línea de estudio podría dedicarse a estudiar la relación entre la planificabilidad del sistema y la exclusión mutua temporal. Como avance obsérvese que en la figura 4.16 se representa un proceso periódico que es planificable, es decir su tiempo de cómputo es menor que su período, y para el cual se puede demostrar fácilmente que todas sus transiciones están en exclusión mutua temporal. Es por tanto razonable pensar que exista una relación entre exclusión mutua temporal y planificabilidad.

4.7. Implementación de conflictos. Distribución de conflictos

Discutiremos en esta sección la implementación de conflictos entre transiciones en una RdPT. La tarea que abordamos es sensiblemente más complicada de lo que un análisis superficial podría

[6] A grosso modo, la exclusión mutua temporal (ver anexo A, definición A.4.5) garantiza que no existe ningún instante en el que dos transiciones puedan ser disparadas concurrentemente

hacer pensar. En concreto, es preciso responder a tres cuestiones:

1. ¿Dónde, en qué protoproceso, se implementarán las transiciones en conflicto? Es posible que todas ellas residan en un único protoproceso, o bien que sean repartidas (distribuidas[7]) entre varios.

2. ¿Cómo se resolverá el conflicto? Es decir, qué protocolo se deberá aplicar cuando varias transiciones en conflicto estén sensibilizadas simultáneamente para que sólo una de ellas sea la disparada. Será necesario implantar alguna política de elección de transiciones que resuelva el indeterminismo inherente a las redes de Petri en lo que a conflictos se refiere.

3. ¿Dónde se resolverá el conflicto? Es decir, en qué elemento de la implementación se programará la política de resolución del conflicto.

Distribución del conflicto En cuanto a la primera de las cuestiones, los problemas aparecen al plantearnos la implementación de la operación de disparo de una transición, la cual es atómica en el formalismo de las redes de Petri, pero que es descompuesta en tres fases en cualquier implementación software: determinación de la sensibilización de la transición, disparo de la misma y actualización de marcado. En presencia de conflictos, es especialmente importante que estas tres operaciones sean realizadas garantizando el acceso en exclusión mutua a los lugares comunes, pues de lo contrario pueden alcanzarse estados incorrectos o situaciones de interbloqueo en la red.

El enfoque para la solución de este problema adoptado en este trabajo diferirá en objetivos y métodos del planteado en [Vil90] y [CF93], donde la descomposición en protoprocesos de la RdP se abordaba con la premisa de no distribuir las transiciones en conflicto, agrupándolas en el mismo protoproceso. En [Vil90] se argumentaba que si los lugares de entrada comunes a las transiciones en conflicto, y las propias transiciones, estaban repartidos en varios protoprocesos, sería necesario mantener un protocolo de reserva de marcas para poder disparar una transición del conflicto sin interferencias de otros protoprocesos que pretendieran hacer lo mismo. Es, por ejemplo, el caso de la figura 4.17, en la que, según [Vil90], para disparar la transición t_1 era necesario reservar la marca del lugar p antes de que el otro protoproceso desmarcase p para disparar t_4. Estos protocolos incrementaban el número de comunicaciones entre los protoprocesos de la implementación, lo que reducía las prestaciones de la misma. La situación era especialmente costosa cuando se trataba de transiciones relacionadas a través de conflictos acoplados (CA). La solución adoptada era implementar los conflictos en el interior de los protoprocesos, de modo que ellos eran capaces de determinar de forma autónoma sobre qué transición disparar. Para ello, una vez determinadas las clases de compatibilidad de transiciones en $\mathcal{EM}$, la cobertura se realizaba, no sobre las transiciones en solitario, sino sobre las mismas agrupadas en conjuntos de transiciones en conflicto.

Hechas así las cosas, la partición de la red podía llevar a la obtención de un número de protoprocesos mayor que la concurrencia máxima[8] y, por tanto, a generar más protoprocesos de los necesarios en la implementación. Considérese como ejemplo el caso de la figura 4.17, en la que se muestra una red de Petri que modela dos procesos que comparten un dato o recurso común. Es posible apreciar que existe un conflicto entre las transiciones t_1 y t_4. Si estas transiciones han de pertenecer al mismo protoproceso se obtiene una partición que agrupa las transiciones en tres protoprocesos: $\pi_1 = \{p, p_2, p_5; t_1, t_2, t_4, t_5\}, \pi_2 = \{t_3\}$ y $\pi_3 = \{t_6\}$, las cuales se comunican mediante los lugares de comunicación p_1, p_4, p_3 y p_6 (también existe la posibilidad de compartir las transiciones t_2 y t_5). En cualquier caso la implementación se compondría de tres protoprocesos

[7] En este contexto no hablaremos de distribución en el sentido de los sistemas distribuidos, sino significando una simple repartición de las transiciones entre distintos protoprocesos

[8] Concurrencia máxima: máximo número de transiciones que pueden dispararse a la vez en cualquier marcado alcanzable

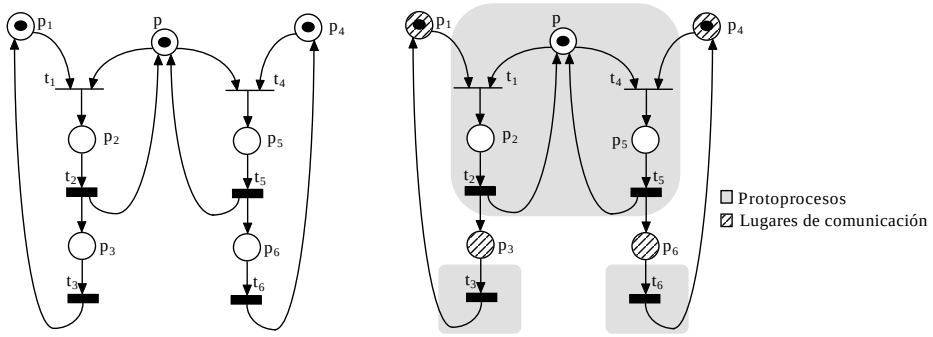


Figura 4.17: Red de Petri con un conflicto simple y su partición en protoprocesos que no distribuyen las transiciones en conflicto

(procesos en definitiva), acoplados entre sí por medio de lugares de comunicación (cuya implementación software por medio de *mensajeros* exige tareas activas, es decir, más procesos), mientras que la concurrencia real es dos, ya que se están modelando dos procesos que comparten un dato común implementable como una variable compartida (de carácter pasivo).

Esta situación puede evitarse si se elimina la restricción de que las transiciones en conflicto deban ser implementadas en el mismo protoproceso. Por el contrario, en este trabajo, consideraremos que las transiciones de salida de un conflicto podrán ser distribuidas en distintos protoprocesos si así es determinado durante el proceso de descomposición, que se realizará con el criterio de la obtención de un número mínimo de protoprocesos. El problema de cobertura tratará de cubrir las transiciones por sí mismas, no agrupándolas por razón de estar en conflicto, y se reunirán las precondiciones o lugares de entrada de las transiciones en conflicto en un objeto común compartido, accesible por todos los protoprocesos. De esta forma el número de comunicaciones se verá limitado a una única con el objeto compartido (evitando así los protocolos de reserva de marcas).

Política de resolución del conflicto En cuanto a la segunda cuestión, la de la política de resolución del conflicto, ya ha sido descrita en esta tesis, en la sección 2.5.1, por lo que referimos directamente a ella.

Lugar de implementación de la política de resolución del conflicto Por último, en cuanto a la tercera cuestión planteada al principio, la política de resolución del conflicto residirá allá donde se sitúe el lugar o lugares que generan el conflicto. Éste puede estar en el interior de alguno de los protoprocesos generados o en algún elemento común y externo a los protoprocesos. Las únicas funciones de este objeto común serán las de contener el o los lugares que provocan el conflicto e implementar la política de resolución del mismo. Su papel será, por lo tanto, exclusivamente de control, no operativo, ya que no contendrá la implementación de ninguna transición.

Pasemos seguidamente a comentar cada uno de los tipos de conflictos que pueden aparecer y particularicemos la respuesta a las cuestiones planteadas para cada uno de ellos. Se considerarán para su estudio conflictos de libre elección, simples y acoplados, de los cuales se muestran ejemplos en la figura 4.18, provocados por lugares 1-limitados, es decir, se debe cumplir que cualquier

conflicto estructural sea un conflicto efectivo [Sil85], o, lo que es lo mismo, que cualquier lugar con más de una transición de salida genere un conflicto efectivo. Al igual que en secciones anteriores se prescindirá en la exposición, salvo que se indique lo contrario, de las características temporales de las RdPT.

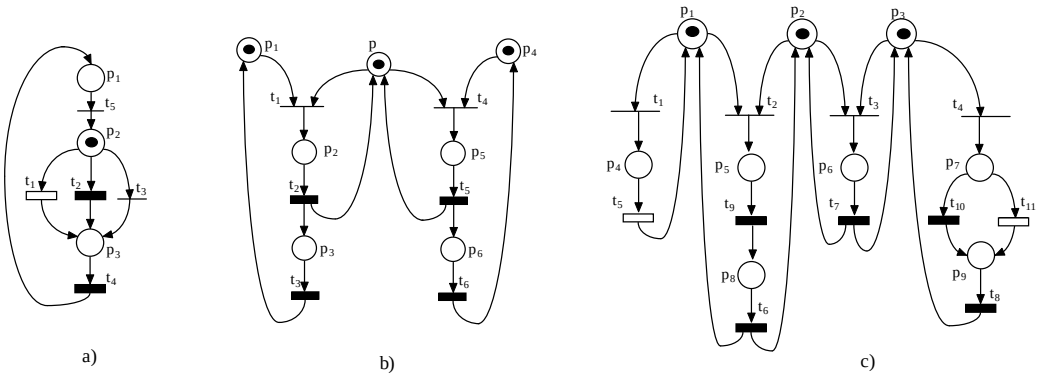


Figura 4.18: a) Conflicto de libre elección. b) Conflicto simple. c) Conflicto acoplado.

4.7.1. Conflictos de libre elección

Definición 4.7.1 *Diremos que un conjunto de transiciones de una red de Petri* **N**<*P, T; Pre, Post; M_0*> *están en conflicto de libre elección si comparten un único lugar de entrada p t.q.* $\forall t \in p^\bullet$, $^\bullet t = \{p\}$.

Si el lugar p es 1-limitado, cuando esté marcado, sus transiciones de salida se encontrarán en situación de conflicto efectivo. Al ser 1-limitado sus transiciones de salida están en $\mathcal{EM}$, por lo que deben pertenecer a la misma clase de compatibilidad de transiciones en $\mathcal{EM}$, aquella generada por el elemento de partición que contenga el lugar común p. Además no es posible que pertenezcan a ninguna otra clase de transiciones en $\mathcal{EM}$ ya que no tienen ningún otro lugar de entrada, por lo que no es posible que una parte de las transiciones estén cubiertas por otro elemento de partición.

Por ello todas las transiciones de un conflicto de libre elección serán implementadas en un único protoproceso, el generado por el elemento de partición al que pertenezca el lugar común que provoca del conflicto. El protoproceso contendrá el lugar común e implementará la política de resolución del conflicto.

Ejemplo En la red de la figura 4.18.a aparece un conflicto de libre elección entre las transiciones t_1, t_2 y t_3 provocado por p_2. La red tiene un único elemento de partición, un p-invariante mono-marcado, $I_1 = \{p_1, p_2, p_3\}$, que contiene el lugar común y cubre todas las transiciones en conflicto, además del resto de transiciones de la red, por lo que la implementación se compondrá de un único protoproceso $\pi_1 = \{p_1, p_2, p_3; t_1, t_2, t_3, t_4, t_5\}$. △

4.7.2. Conflictos simples

Definición 4.7.2 *Sea* $\mathbf{N} < P, T; Pre, Post; M_0 >$ *una red de Petri marcada. Sea* $\{t_1, t_2, \ldots t_q\} \subseteq T$ *un subconjunto de transiciones de* **N**. *Diremos que estas transiciones están en conflicto simple*

si cada transición del conjunto tiene como máximo un lugar de entrada compartido con otras transiciones de T [Sil85], o lo que es lo mismo, ninguna de las transiciones interviene en ningún otro conflicto.

Por tanto todas las transiciones en conflicto simple tienen un único lugar común, $p = \bigcap_{i=1}^{q} {}^\bullet t_i$, para el que se cumple $\forall t_i \in p^\bullet \; i = 1 \ldots q$, $\forall p_j \in \{{}^\bullet t_i - p\}$, $|p_j^\bullet| \leq 1$. Si el lugar p es 1-limitado, cuando esté marcado, sus transiciones de salida se encontrarán en situación de conflicto efectivo.

Las situaciones de conflicto simple aparecen con frecuencia en sistemas de tiempo real. Modelan, entre otras, situaciones como la utilización de recursos comunes, o datos compartidos utilizados en exclusión mutua por distintos procesos. El control del acceso al recurso puede ser representado por medio del lugar común, que actuará a modo de semáforo. El proceso que solicite la utilización del recurso deberá desmarcar el lugar antes de poder adquirirlo. Las transiciones de salida de un conflicto simple representan, por tanto, las solicitudes de utilización del recurso por parte de los distintos procesos, y la resolución del conflicto a favor de una de las transiciones, representa el acceso a la utilización del recurso por el proceso al cual pertenece la transición.

Desde el punto de vista de la partición de la red, si el lugar común es 1-limitado, sus transiciones de salida se encuentran en $\mathcal{EM}$ y, por lo tanto, pueden pertenecer a la misma clase de compatibilidad de transiciones en $\mathcal{EM}$, C_i: la generada por el elemento de partición al que pertenece el lugar común p. Por ello puede ser implementadas todas juntas en el mismo protoproceso. Ésta era la estrategia utilizada en [Vil90]. En el presente trabajo se propone otra aproximación que tiene en cuenta que las transiciones pueden tener otros lugares de entrada (de otro modo estaríamos ante un conflicto de libre elección). Estos pueden pertenecer a otros elementos de partición, por lo que es posible que las transiciones puedan estar cubiertas por otros elementos de partición y, por tanto, pertenecer a distintos protoprocesos.

El tipo y número de transiciones que pueden aparecer en conflictos simples está limitado solamente por la restricción de la presencia de una única transición CODE en él. El resto de transiciones pueden repetirse. En el siguiente capítulo (sobre realización software) se hará una consideración, que tendrá influencia en la partición, sobre la distribución de conflictos simples cuando éstos se forman con distintos tipos de transiciones. Hasta entonces prescindiremos de las características temporales de las transiciones.

Al poder distribuir las transiciones de salida del conflicto entre los protoprocesos obtenidos durante la partición de la red, se alcanza una implementación más coherente con la concurrencia real que aquella en la que todas las transiciones de salida del conflicto están en el mismo protoproceso (como sucede en la figura 4.17).

Distribución de las transiciones

Para poder distribuir las transiciones de salida de un conflicto simple entre distintos protoprocesos, éstas deben estar cubiertas por distintos elementos de partición ($\mathcal{EP}$). De esta manera cada transición del conflicto se implementará en el protoproceso (π) a que dé lugar el elemento de partición por el que está cubierta. El lugar común, que al tener transiciones de salida distribuidas en varios protoprocesos no será privado a ninguno de ellos, será implementado por separado y encapsulará la política de resolución del conflicto, que se resolverá en su interior. Evidentemente, para poder realizar una distribución de las transiciones es condición necesaria que el lugar común (p) no esté contenido en ningún elemento de partición esencial utilizado en la partición ya que, de otro modo, p sería privado a algún protoproceso, lo que forzaría a implementar todas sus transiciones descendentes (las del conflicto) en ese protoproceso, evitando la distribución que pretendemos.

Proposición 4.7.1 (Condición necesaria para la distribución de tr. en conflicto simple) *Sea* $\mathbf{N} < P, T; Pre, Post; M_0 >$ *una red de Petri marcada. Sea* $\mathcal{TCS} = \{t_1, t_2, \ldots t_q\}$, $\mathcal{TCS} \subseteq T$, *un conjunto de transiciones de* $\mathbf{N}$ *que están en conflicto simple. Sea p el lugar común 1-limitado que*

origina el conflicto ($p^\bullet = \mathcal{TCS}$). Si $\forall t_i \in \mathcal{TCS} \quad \exists\ \mathcal{EP}_i$ *t.q.* $t_i \in T(\mathcal{EP}_i) \wedge p \notin \|\mathcal{EP}_i\|, i : 1 \ldots q$, *entonces las transiciones de $\mathcal{TCS}$ no tienen por qué pertenecer todas al mismo protoproceso π_i.*

Es decir, si todas las transiciones de $\mathcal{TCS}$ están cubiertas por distintos elementos de partición ($\mathcal{EP}_i$) que no incluyan a p entonces las transiciones de $\mathcal{TCS}$ pueden distribuirse entre distintos protoprocesos (π_i), generados a partir de los elementos de partición (π_i generado a partir de $\mathcal{EP}_i$).

Demostración.. Es trivial si se considera que el protoproceso π_i puede generarse a partir de $\mathcal{EP}_i$. Si el protoproceso π_i implementa todas las transiciones descendentes de los lugares de $\|\mathcal{EP}_i\|$ será de la forma $\pi_i < Int(T(\mathcal{EP}_i)), T(\mathcal{EP}_i); Pre_i, Post_i; M_{0i} >$, siendo $Int(T))$ el conjunto de lugares privados entre un conjunto de transiciones, y con $Pre_i = Pre|_{Int(T(\mathcal{EP}_i))\times T(\mathcal{EP}_i)}$, $Post_i = Post|_{T(\mathcal{EP}_i)\times Int(T(\mathcal{EP}_i))}$ y $M_{0i} = M_0|_{Int(T(\mathcal{EP}_i))}$. Como cada $t_i \in T(\mathcal{EP}_i)$ se verificará que $t_i \in \pi_i$. Esto ocurrirá con cada transición de $\mathcal{TCS}$, por lo que no todas estarán en el mismo protoproceso.

Notar que es necesario imponer la no pertenencia del lugar común p a ningún elemento de partición esencial. De otro modo, si hubiese una transición t_k cubierta únicamente por un elemento de partición ($\mathcal{EP}_k$) que contiene a p ($p \in \|\mathcal{EP}_k\|$), el único protoproceso que puede implementar la transición t_k contendrá el lugar p, y en consecuencia, en virtud de la propiedad 4.2.1, debería contener todas las transiciones de salida de p, es decir, $\mathcal{TCS}$, sin distribuirlas (justo lo contrario de lo que pretendíamos). △

Se hace notar que no tiene por qué existir un protoproceso para cada una de las transiciones del conflicto, sino que más de una puede estar en un mismo protoproceso (ver ejemplo 4 en esta misma sección). La única condición es que éste no contenga el lugar común.

La proposición plantea la posibilidad de distribuir las transiciones, no que sea obligado hacerlo. De hecho, más adelante en esta misma sección, se mostrará un ejemplo en el que la solución más eficiente es la no distribución de las transiciones, pese a que es posible hacerla. En suma, ante un conflicto simple tendremos dos posibilidades:

1. Si se verifican las condiciones de la proposición 4.7.1:

   - El único lugar común no deberá pertenecer a ninguno de los protoprocesos, por lo que no será privado a ninguno de ellos y será implementado por separado como un lugar de comunicación de destino múltiple.
   - En él residirá la implementación de la resolución del conflicto.
   - El resto de lugares de entrada de las transiciones distribuidas podrá ser privados a los protoprocesos que las implementan o de comunicación con otros. En este último caso, será necesario utilizar un concepto que será formalizado en la siguiente sección, el *sincronizador local*, y así incluir estos lugares de comunicación en el mismo objeto que contenga el lugar generador del conflicto (véase un ejemplo en la figura 4.19). De esta forma, cada protoproceso conocerá el estado de los lugares de entrada de su transición mediante una única consulta (recordar que queremos evitar los protocolos de reserva de marcas).

2. Si no se realiza la distribución de las transiciones en conflicto, todas ellas se implementarán en el mismo protoproceso, que además contendrá el lugar común y con él, la política de resolución del conflicto. El resto de lugares de entrada de las transiciones en conflicto podrán ser lugares privados del protoproceso o lugares de comunicación de destino único hacia alguna de las transiciones del conflicto, pero nunca más de una a la vez, ya que, de otro modo, se estaría ante un conflicto acoplado.

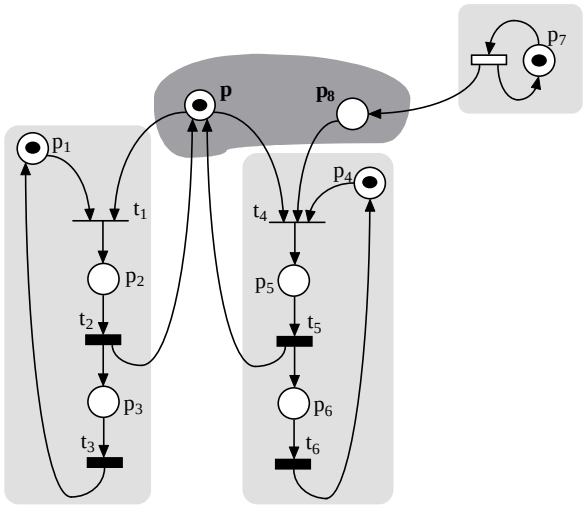
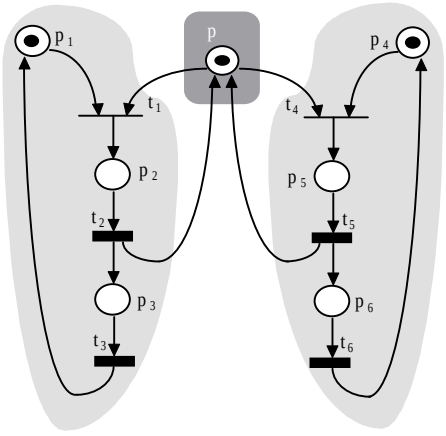


Figura 4.19: Conflicto simple con lugar común (p)y lugar de comunicación (p_8) agrupados en un sincronizador local reducido

Si no todas las transiciones de un conflicto simple cumplen el requisito de la proposición 4.7.1, es decir, no existe para todas ellas un elemento de partición que las cubra y que no contenga al lugar común, todavía puede intentarse la distribución. Si es posible, pueden añadirse lugares implícitos para construir p-invariantes monomarcados (ver sección 4.3.5), de forma que haya uno para cada transición en conflicto y, por tanto, se cumplan los requisitos de la proposición (ver ejemplo 3 de esta misma sección).

Ejemplo 1 (Figura 4.20.a) El cálculo de p-invariantes monomarcados ($I_1 = \{p_1, p_2, p_3\}$; $I_2 = \{p_4, p_5, p_6\}$; $I_3 = \{p, p_2, p_5\}$) pone de manifiesto como dos de ellos cubren todas las transiciones de la red y la parten en dos protoprocesos: $\pi_1 = \{p_1, p_2, p_3; t_1, t_2, t_3\}$ y $\pi_2 = \{p_4, p_5, p_6; t_4, t_5, t_6\}$. Como puede comprobarse las transiciones en conflicto (t_1 y t_4) están distribuidas entre los dos protoprocesos. El lugar común entre ambas transiciones, que no es privado a ninguno de los protoprocesos, será implementado como lugar de comunicación de destino múltiple, haciendo el papel de dato compartido para el que se debe garantizar acceso en exclusión mutua. La resolución de conflicto reside en este lugar, decidiéndose en favor del primer protoproceso que trate de desmarcarlo. Este objeto tiene un carácter pasivo por lo que, de este modo, la implementación se compondrá de dos procesos frente a los tres que se obtendrían si se agruparan las transiciones en conflicto en un único protoproceso tal como se hacía en [Vil90] (figuras 4.20.b o 4.20.c). Se trata por tanto de una implementación que coincide con la concurrencia real del sistema modelado.

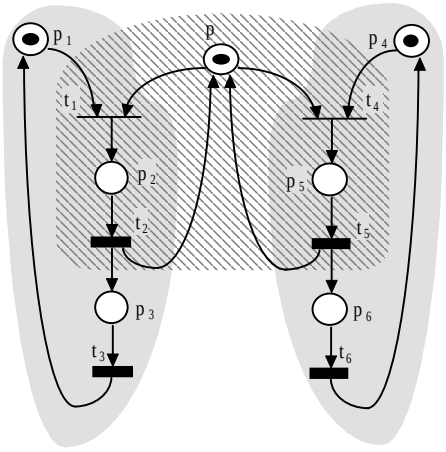
Otras formas de realizar la partición sin considerar la distribución de transiciones [Vil90] (ver figuras 4.20.b y 4.20.c), hubieran arrojado una solución con tres protoprocesos: el protoproceso que reúne las transiciones del conflicto y el lugar p, con sus transiciones de entrada y salida ($\pi_1 = \{p, p_2, p_5; t_1, t_2, t_3, t_4\}$), y otros dos para las transiciones t_3 y t_5, que no pueden ser cubiertas por el mismo elemento de partición. Existen varias posibilidades para comunicar los tres protoprocesos: mediante el compartimiento de lugares y transiciones conexos ($\{t_1, p_2, t_2\}$ y $\{t_4, p_5, t_5\}$), mostrado en la figura 4.20.b; o renunciar a cualquier compartimiento y comunicar los protoprocesos a través de lugares de comunicación asíncrona p_1, p_3, p_4 y p_6 (figura 4.20.c). △



$I_1 = \{p_1, p_1, p_3\}$
$I_2 = \{p_4, p_5, p_6\}$
$I_3 = \{p, p_2, p_5\}$

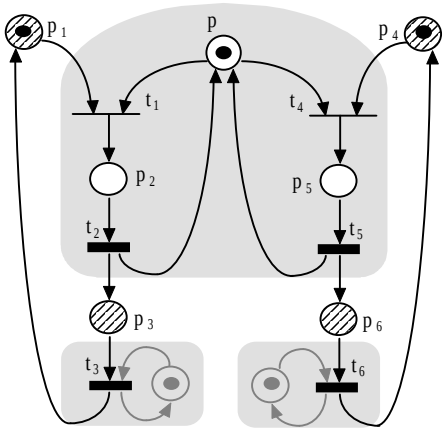
	t_1	t_2	t_3	t_4	t_5	t_6
$T(I_1)$	x	x		x	x	
$T(I_2)$	x	x	x			
$T(I_3)$				x	x	x

a)



	t_1	t_2	t_3	t_4	t_5	t_6
$T(I_1)$	x	x		x	x	
$T(I_2)$	x	x	x			
$T(I_3)$				x	x	x

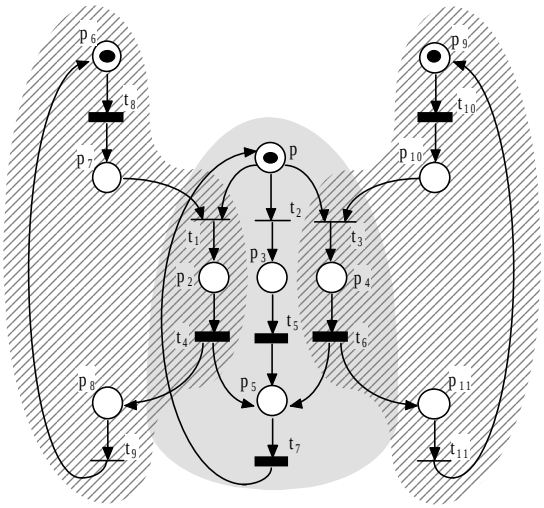
b)



	t_1	t_2	t_3	t_4	t_5	t_6
$T(I_1)$	x	x		x	x	
$T(I_2)$	x	x	x			
$T(I_3)$				x	x	x

c)

Figura 4.20: Distintas descomposiciones de una red con conflicto simple con y sin distribución de transiciones



	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
$T(I_1)$	x	x	x	x	x	x	x				
$T(I_2)$	x			x				x	x		
$T(I_3)$			x			x				x	x

Figura 4.21: Conflicto simple en el que no todas las transiciones en conflicto están cubiertas por elementos de partición a los que no pertenece el lugar común. Tabla de partición en la que se muestra que los tres $\mathcal{EP}$ son esenciales.

Ejemplo 2 Las transiciones en conflicto de la red de la figura 4.21 no se podrán distribuir porque no todas están cubiertas por elementos de partición que no incluyen al lugar común p. El cálculo de p-invariantes monomarcados da como resultado tres de ellos: $I_1 = \{p, p_2, p_3, p_4, p_5\}$, $I_2 = \{p_2, p_6, p_7, p_8\}$ e $I_3 = \{p_4, p_9, p_{10}, p_{11}\}$. La tabla de cobertura que se muestra en la figura muestra como la transición t_2 está cubierta únicamente por el elemento de partición I_1, que contiene el lugar común, es decir, no se cumplen los requisitos de la proposición 4.7.1. Por ello, no se podrán distribuir las transiciones al implementar el protoproceso generado por I_1 se implementaran en él todas las transiciones del conflicto. Este protoproceso será $\pi_1 = \{p, p_2, p_3, p_4, p_5; t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. El resto de transiciones no pueden ser cubiertas por el mismo elemento de partición, por lo que se generarán dos protoprocesos más. Como en el caso anterior existen varias alternativas en cuanto a las transiciones compartidas: compartir transiciones y lugares conexos ($\{t_1, p_2, t_4\}$ y $\{t_3, p_4, t_6\}$), sólo las transiciones, o no compartir nada. La partición mostrada en la figura 4.21 corresponde a la primera de estas opciones, por lo que el resto de protoprocesos es $\pi_2 = \{p_2, p_6, p_7, p_8; t_1, t_4, t_8, t_9\}$ y $\pi_3 = \{p_4, p_9, p_{10}, p_{11}; t_3, t_6, t_{10}, t_{11}\}$. El protoproceso π_1, que encapsula el conflicto, será el encargado de implementar la política de resolución del conflicto.

Una posible interpretación de la red que ayuda a la compresión de la implementación puede ser un proceso (π_1) que ejecuta citas extendidas (las transiciones y lugares conexos compartidos) con los otros dos. Las comunicaciones están priorizadas del mismo modo que las transiciones SYCO en conflicto. Por ejemplo se podría haber asignado a t_2 la menor prioridad de modo que se elija para disparar sólo en el caso de que ninguna de las dos citas (las otras transiciones) estén disponibles, modelando de este modo una cita con parte alternativa. △

Ejemplo 3 En la figura 4.22 se muestra una red en la que las transiciones del conflicto no se pueden distribuir ya que no todas están cubiertas por un $\mathcal{EP}$ que no contenga al lugar común. En efecto, los únicos p-invariantes que se pueden calcular son: $I_1 = \{p, p_2, p_4\}$ e $I_2 = \{p_3, p_4, p_5\}$. No hay un elemento de partición que no contenga el lugar común p para cada una de las transiciones en conflicto. Por tanto, no pueden ser distribuidas.

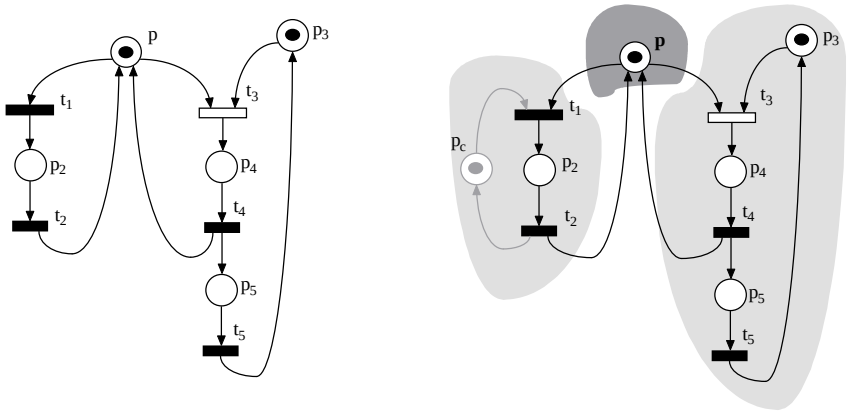


Figura 4.22: Conflicto simple distribuible si se añade un lugar implícito (p_c).

Sin embargo, si añadimos un lugar implícito de forma que se construya un p-invariante monomarcado que cubra la transición t_2, podremos aplicar la proposición 4.7.1 y alcanzar la partición de la figura 4.22.

Ejemplo 4 En la figura 4.23 se muestra una red en la que se pueden distribuir las transiciones del conflicto simple, y varias de ellas pertenecen al mismo protoproceso. El cálculo de p-invariantes arroja como resultado los siguientes: $I_1 = \{p_6, p_7, p_8\}$, $I_2 = \{p_1, p_3, p_5, p_7\}$, $I_3 = \{p_4, p_5\}$ e $I_4 = \{p_2, p_3\}$. Con su ayuda puede construirse la tabla de cobertura transiciones en la que se aprecia que dos elementos de partición son esenciales. Existen varias posibilidades de partición, de las cuales la más eficiente (figura 4.23.a) sería implementar las transiciones t_1 a t_6 en un protoproceso que contenga al lugar común p_1 y otro dedicado a las transiciones cubiertas por el p-invariante I_1, que compartirá con el anterior las transiciones t_3, y t_6 y el lugar p_7.

Sin embargo, con este ejemplo se desea ilustrar otra solución en la que se distribuirán las transiciones del conflicto. Las transiciones pueden distribuirse pues todas ellas están cubiertas por p-invariantes monomarcados que no incluyen al lugar común p_1. Por ello obtendríamos una solución con los protoprocesos $\pi_{1a} = \{p_2, p_3; t_1, t_4\}$, $\pi_{1b} = \{p_4, p_5; t_2, t_5\}$ y $\pi_2 = \{p_6, p_7, p_8; t_3, t_6, t_7\}$. Además debe considerarse que las transiciones de los protoprocesos π_{1a} y π_{1b} están en exclusión mutua, por lo que pueden estar en el mismo protoproceso. Por ello la solución final consta de dos protoprocesos: $\pi_1 = \{p_2, p_3, p_4, p_5; t_1, t_2, t_4, t_5\}$ y $\pi_2 = \{p_6, p_7, p_8; t_3, t_6, t_7\}$, dejando el lugar común fuera de los dos protoprocesos. La elección de la transición a disparar se implementará en el lugar común. △

Este último ejemplo se ha incluido para mostrar que a veces, aunque sea posible, la distribución de las transiciones no es la mejor solución. En la solución sin distribuir son necesarios exclusivamente dos protoprocesos, mientras que en la distribuida se aprecian dos protoprocesos más un objeto compartido para contener el lugar común e implementar el protocolo de resolución del conflicto, objeto, que aunque sea pasivo, complicará y sobrecargará la implementación.

Por lo tanto, ante la posibilidad de distribuir transiciones en conflicto entre varios protoprocesos, deberemos plantearnos cuándo es conveniente hacerlo. La respuesta puede estar inspirada en este último ejemplo: se distribuirá cuando la implementación resultante se componga de menos procesos, y en caso de igualdad, cuando conste de menos objetos compartidos para contener lugares comunes.

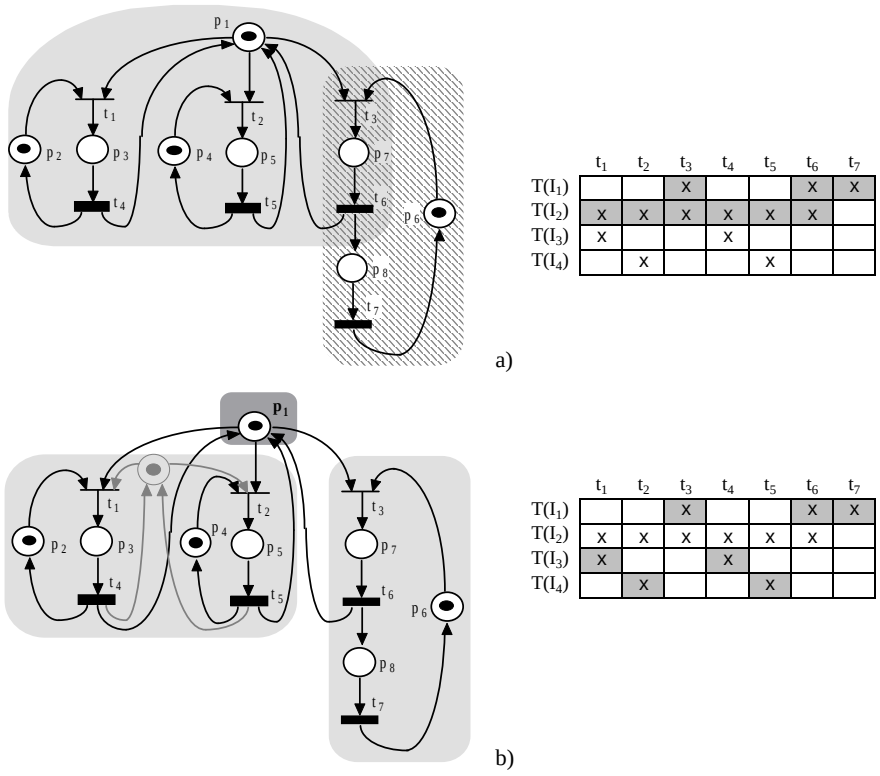


Figura 4.23: Conflicto simple con varias transiciones en el mismo protoproceso.

4.7.3. Conflictos acoplados

Se tratarán aquí el resto de conflictos no recogidos en los anteriores apartados, mostrando que el caso anterior puede considerarse un caso particular de éste. Se trata de conflictos en los que puede existir concurrencia entre las transiciones en conflicto, lo que dificulta la técnica de implementación.

Definición 4.7.3 *Sea* $\mathbf{N} < P, T; Pre, Post; M_0 >$ *una red de Petri marcada. Dos transiciones* t_i, $t_j \in T$ *están en conflicto acoplado (*$\mathcal{CA}$*) si se cumple para ellas que* ${}^\bullet t_i \cap {}^\bullet t_j \neq \emptyset$ *ó* $\exists t_k \in T$ *t.q.* $t_i \,\mathcal{CA}\, t_k \wedge\ t_k \,\mathcal{CA}\, t_j$.

Nótese que la primera condición engloba también a las transiciones en conflicto simple (caso particular de conflicto acoplado). Esta situación se verá reflejada en la técnica de implementación descrita en este apartado que también podrá ser aplicada a los conflictos simples. Se describe, por tanto, una técnica más general.

De nuevo, la principal dificultad que plantean estos conflictos es la no disponibilidad de operaciones atómicas en el lenguaje de implementación para implementar el disparo de las transiciones. Esto hace que los lugares comunes no puedan ser separados unos de otros, pues de ser así, se podrían provocar situaciones de interbloqueo o de marcados no alcanzables. Situaciones que deberían ser prevenidas mediante algún protocolo de reserva de marcas que se ponga en marcha cuando se desea disparar alguna transición. Consideremos, por ejemplo, la red de la figura 4.18.c, en la que las transiciones t_1, t_2, t_3 y t_4 están en conflicto acoplado provocado por los lugares p_1, p_2 y p_3. Estudiemos sobre la figura el disparo de la transición t_2. La secuencia de acciones a realizar para el disparo de la transición pueden verse a continuación, junto con la secuencia de acciones para el

disparo de la transición t_1.

disparo de t_2	disparo de t_1
desmarcar(p_2)	desmarcar(p_1)
desmarcar(p_1)	disparar(t_1)
disparar(t_1)	marcar(p_4)
marcar(p_5)	

El protoproceso que implemente la transición t_2 deberá retirar las marcas de los lugares p_1 y p_2. Supongamos que se desmarca p_2 pero antes de poder desmarcar p_1, la marca de este lugar es retirada por t_1, que también intenta su disparo a la vez que t_2. El resultado es que tenemos un estado erróneo en la red de Petri, con los lugares p_1 y p_2 desmarcados pero habiendo disparado t_1. Para evitar estas situaciones, se debe implementar un protocolo de reserva de marcas: la transición t_2, antes de desmarcar p_2, deberá reservar la marca de este lugar y solicitar además la de p_1. Una vez que se ha asegurado la posesión de ambas marcas, la transición puede disparar. El empleo de estos protocolos aumenta el número de comunicaciones entre los lugares y los protoprocesos, ya que no sólo hay que marcar o desmarcar, sino reservar y liberar reservas de marcas. Además es posible que se eviten evoluciones paralelas en la red si el protocolo no está bien implementado. Por ejemplo si la transición t_2, después de reservar la marca de p_2 no puede obtener la marca de p_1, debe liberar p_2, ya que de otro modo impediría el disparo de t_3. La evitación de estas situaciones anómalas será materia de estudio en este punto.

La exposición del método de distribución de las transiciones en conflicto acoplado, para evitar el mantenimiento de los mencionados protocolos de reserva de marcas, se abordará desde dos perspectivas diferentes. En primer lugar recuperaremos el antiguo concepto de *sincronizador local* expuesto en [Vil90]. Más adelante se analizará una modificación de los sincronizadores que simplificará la implementación. Las diferencias entre estas dos aproximaciones se basan en quién lleva la iniciativa en la solución del conflicto: el sincronizador local o el protoproceso.

Distribución mediante el uso de sincronizadores locales

Con el fin de evitar cualquier protocolo de reserva de marcas todos los lugares de entrada de un conjunto de transiciones en conflicto acoplado son agrupados en un único objeto llamado *sincronizador local* ($\mathcal{SL}$). Al contener todas las precondiciones de las transiciones, el $\mathcal{SL}$ es capaz de concluir autónomamente, y mediante una única consulta, sobre la sensibilización de éstas. De este modo el $\mathcal{SL}$ puede solucionar el conflicto en favor de una u otra transición, implementando para ello cualquier política de resolución en su interior. Así mismo, será él quien promueva el disparo de las transiciones en conflicto una vez que detecte su sensibilización. Si hay varias sensibilizadas, decidirá cuál de ellas será la disparada. Por tanto, el sincronizador debe ser un objeto de implementación activo, similar al coordinador de las implementaciones centralizadas.

Construcción del sincronizador local Dada una red de Petri $\mathbf{N} < P, T; Pre, Post; M_0 >$ marcada, sea $\mathcal{TCA} \subseteq T$ un conjunto de transiciones en conflicto acoplado provocado por lugares 1-limitados, entonces el sincronizador local se construirá como la unión de todos los lugares de entrada de las transiciones en conflicto, $\mathcal{SL} = \cup_{t_i \in \mathcal{TCA}} {}^\bullet t_i$. En él pueden aparecer lugares de comunicación, de destino único o múltiple. Esta situación puede verse en el esquema de la figura 4.24.

Distribución de las transiciones en conflicto acoplado Las transiciones en conflicto acoplado podrán distribuirse si es posible obtener elementos de partición que no contengan lugares del sincronizador local. Estos elementos de partición podrán ser construidos mediante la adición de algún lugar implícito, tal como se muestra en el siguiente ejemplo.

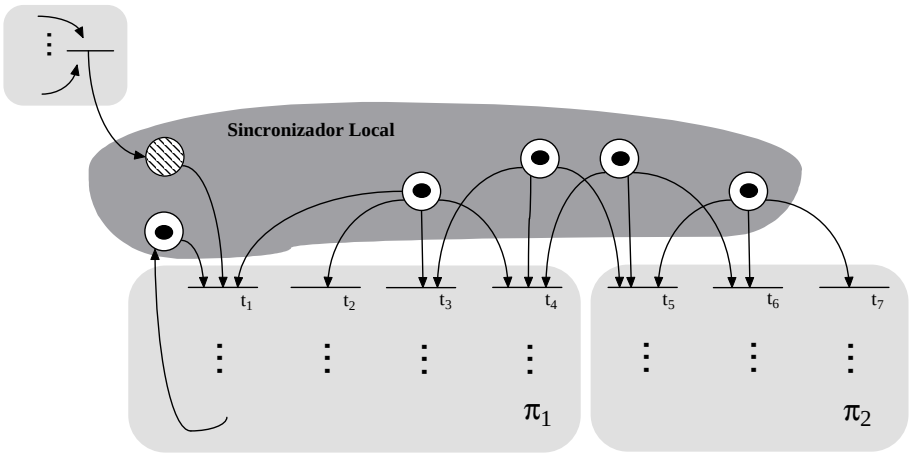


Figura 4.24: Cómo se forma un sincronizador local

Ejemplo Consideremos la red de la figura 4.25.a. En ella existen cuatro transiciones en conflicto acoplado: t_1, t_2, t_3 y t_4 provocado por los lugares p_1, p_2 y p_3. Mediante el cálculo de p-invariantes ($I_1 = \{p_1, p_4, p_5, p_8\}; I_2 = \{p_2, p_5, p_6, p_8\}; I_3 = \{p_3, p_6, p_7, p_9\}$) y la construcción de la tabla de cobertura se puede alcanzar una partición con dos protoprocesos que implementan, cada uno, dos de las transiciones en conflicto acoplado: $\pi_1 = \{p_1, p_4, p_5, p_8; t_1, t_2, t_5, t_6, t_9\}$ y $\pi_2 = \{p_3, p_6, p_7, p_9; t_3, t_4, t_7, t_8, t_{10}, t_{11}\}$. De ellos hay que excluir los lugares que pertenecen al sincronizador local, $\mathcal{SL} = \{p_1, p_2, p_3\}$, que serán implementados en éste. Como consecuencia los protoprocesos quedan abiertos, por lo que deben ser cerrados mediante la adición de lugares implícitos. La partición resultante se muestra en la figura 4.25.b. △

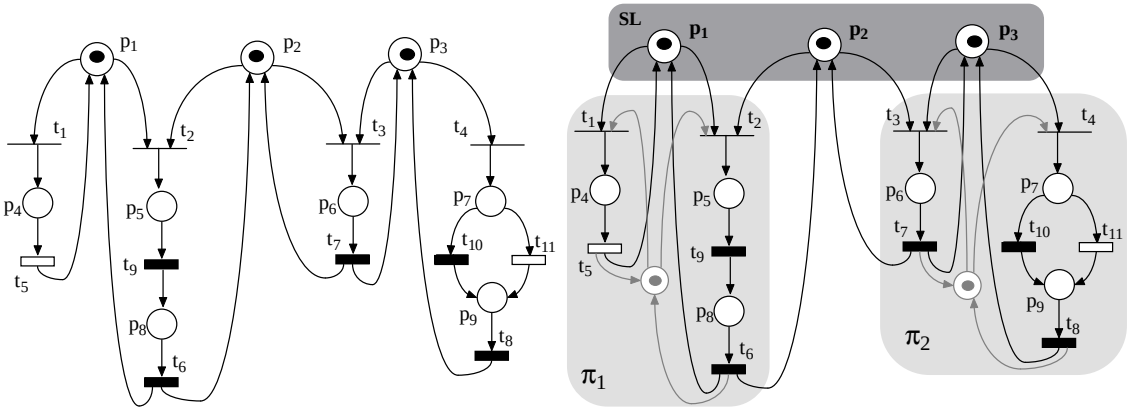


Figura 4.25: a) RdP con conflicto acoplado; b) Una posible implementación utilizando un sincronizador local

Implementación mediante sincronizadores locales reducidos

El enfoque del punto anterior considera la resolución del conflicto desde el punto de vista del sincronizador local. Es él quien decide qué transición será la disparada en una situación de conflicto efectivo, atendiendo al estado de los lugares de entrada. Por ello debe disponer de todas las precondiciones de las transiciones en conflicto. Sin embargo en esta subsección se abordará el problema desde el lado del protoproceso, comprobando que no es necesario que el sincronizador local encapsule todos los lugares de entrada de las transiciones, sino una parte de ellos. Incluso se mostrará que es posible distribuir los lugares de entrada en varios sincronizadores.

Volvamos a considerar la red de la figura 4.25. Dos de los lugares que originan el conflicto, p_1 y p_3, son privados a los protoprocesos determinados en la partición. Estudiemos el caso de p_1 (el de p_3 es similar). El lugar es privado al protoproceso π_1 por lo que sólo puede ser marcado o desmarcado por las transiciones de π_1. Que el lugar esté marcado es señal de que el flujo de control del protoproceso habrá alcanzado la situación de conflicto, que debe ser resuelta consultando el resto de lugares de entrada. En este caso se necesita consultar el marcado de p_2, pero no el de p_1, pues el protoproceso ya conoce que está marcado (si no, no estaría tratando de resolver el conflicto). Por lo tanto, los lugares privados de los protoprocesos, pueden ser retirados de los sincronizadores locales porque es seguro que se encuentran marcados si se está planteando la resolución del conflicto. Al sincronizador resultante lo denominamos *sincronizador local reducido* ($\mathcal{SL}r$).

La resolución del conflicto será repartida entre el protoproceso y el sincronizador local reducido. Para que el conflicto pueda resolverse es necesario que el protoproceso tenga decidido de antemano un orden de prioridad para el disparo de sus transiciones en conflicto, y que entonces consulte al sincronizador local reducido el estado del resto de lugares de entrada. Por tanto el sincronizador local reducido puede ser un objeto pasivo desde el punto de vista de la resolución del conflicto. La consulta del estado de los lugares comunes deberá ser realizada de forma atómica para evitar situaciones de interbloqueo o marcados ficticios. Por ello los lugares no privados serán encapsulados en el sincronizador local reducido. Por así decirlo, los protoprocesos en situación de conflicto, competirán por las marcas de los lugares comunes, como en el caso del conflicto simple, aunque en este caso es posible que se compita por varios lugares en vez de uno sólo. Es por ello que, tal como avanzábamos, los conflictos simples son un caso particular de los acoplados.

Ejemplo 1 Aplicando este nuevo enfoque a la red de la figura 4.25.a, obtendremos la partición que se muestra en la figura 4.26. El sincronizador local ha quedado reducido al lugar p_2, único lugar no privado a ningún protoproceso. Los protoprocesos deberán priorizar el disparo de sus transiciones. Por ejemplo, una asignación de prioridades podría dar mayor prioridad a t_2 que a t_1, de modo que π_1 disparara t_2 si consiguiese capturar la marca de p_2 y si no, disparase t_1. Por tanto los protoprocesos compiten por la marca del lugar p_2, que, en caso de que los dos soliciten la marca al mismo tiempo, deberá ser retirada por aquel protoproceso que posea la transición más prioritaria de entre todas las que necesiten la maca del lugar. △

No sólo es posible retirar los lugares privados del sincronizador local, sino que los lugares del sincronizador local reducido pueden ser distribuidos. Consideremos para ilustrar este caso la red de la figura 4.27, en la que las transiciones t_1 a t_6 están en conflicto acoplado provocado por los lugares p_1 a p_5. Durante el proceso de partición es posible descomponer la red en tres protoprocesos, cada uno de los cuales contiene dos de las transiciones en conflicto acoplado. El sincronizador local debería contener los lugares p_1 a p_5, pero como tres de ellos (p_1, p_3 y p_5) son privados a los protoprocesos, pueden ser retirados del sincronizador, con lo que obtendríamos un sincronizador local reducido constituido por los lugares p_2 y p_4. Como puede verse en la figura, ninguna de la transiciones del conflicto precisa consultar ambos lugares a la vez. Por tanto, no es posible que exista ninguna situación que necesite una reserva de marcas. En concreto t_2 y t_3 competirán por la marca de p_2, y t_4 y t_5 competirán por la de p_4. Por ello los lugares pueden ser separados en dos

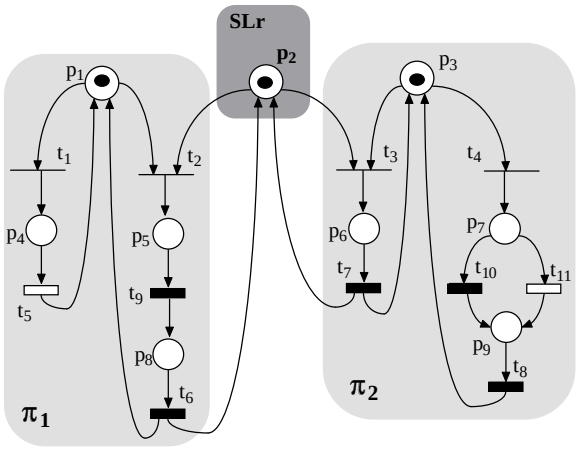


Figura 4.26: Partición de la red de la figura 4.25.a utilizando un sincronizador local reducido

sincronizadores locales reducidos.

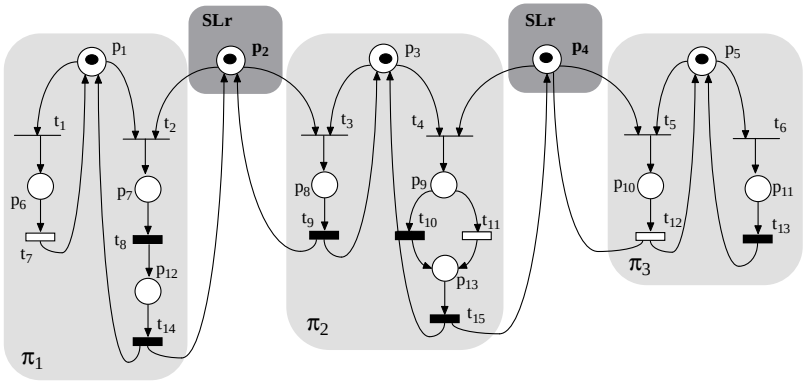


Figura 4.27: Distribución de sincronizadores locales reducidos

Construcción de sincronizadores locales reducidos El ejemplo anterior nos da la idea para formular un método de cálculo de los sincronizadores locales reducidos. Sea una red de Petri $\mathbf{N} < P, T; Pre, Post; M_0 >$ marcada, y sea $\mathcal{TCA} \subseteq T$ un conjunto de transiciones en conflicto acoplado, provocado por una serie de lugares 1-limitados. Los sincronizadores locales reducidos se calcularán siguiendo los siguientes pasos (ver el esquema de la figura 4.28):

1. Comenzaremos asociando un sincronizador local reducido a cada transición en conflicto, encapsulando en él los lugares de entrada de la transición. $\forall t_i \in \mathcal{TCA}$ calcular $\mathcal{SLr}_i = {}^\bullet t_i$

2. Podremos quitar del $\mathcal{SLr}_i$ los lugares privados al protoproceso que implementa la transición t_i. $\mathcal{SLr}_i = \mathcal{SLr}_i - \{p \in \mathcal{SLr}_i \mid p \in \pi_i\}$, siendo π_i el protoproceso que implementa t_i, $t_i \in \pi_i$.

3. Reduciremos el número de sincronizadores uniendo los que compartan transiciones de salida. Dejaremos sólo los que no tengan intersección con ningún otro y uniremos aquellos que tengan intersección no vacía. $\forall i, j$ si $\mathcal{SLr}_i \cap \mathcal{SLr}_j \neq \emptyset$ formar $\mathcal{SLr}_k = \mathcal{SLr}_i \cup \mathcal{SLr}_j$

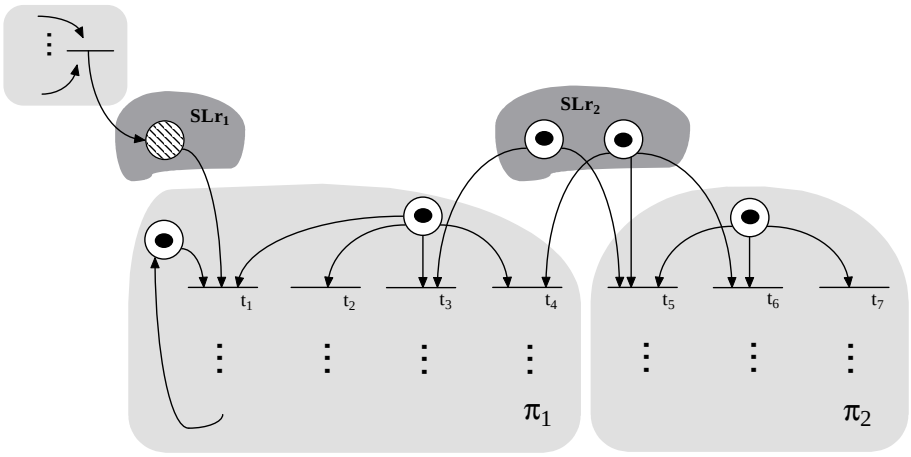


Figura 4.28: Cómo se forma un sincronizador local reducido

4. Descartaremos los sincronizadores que estén contenidos en otros. Descartar los $\mathcal{SL}r_i$ t.q. $\exists \mathcal{SL}r_j$ t.q. $\mathcal{SL}r_i \subseteq \mathcal{SL}r_j$

Es fácil comprobar como al aplicar el procedimiento anterior al caso de los conflicto simples se obtiene el resultado descrito en la sección 4.7.2.

4.7.4. Limitaciones de método de implementación de conflictos

Consideremos el caso de la red de la figura 4.29.a, en la que estudiaremos una situación en la cual la topología del conflicto tiene influencia en la solución del problema de cobertura o partición de la red. Aunque este tema será tratado más en profundidad en el siguiente capítulo (realización software), determinadas topologías son no-implementables en el lenguaje elegido para la codificación, en nuestro caso, Ada 95 (ver sección 2.5.2). Veamos un ejemplo.

En la red de la figura observamos un conflicto entre las transiciones t_2 y t_3. Propongamos una primera implementación como en la figura 4.29.b, para analizarlo y comprobar que no podrá ser implementado de esta forma. La implementación de la figura 4.29.b no distribuye las transiciones en conflicto y además comparte el conjunto t_2 - p_5 - t_4 entre los protoprocesos π_2 y π_3. Con estas premisas el sincronizador local queda reducido a p_2, ya que es el único lugar de entrada no privado a ninguno de los protoprocesos que implementan transiciones del conflicto.

Debido a que el lenguaje de implementación (Ada 95) no contempla como primitiva las comunicaciones a tres bandas, la partición propuesta no podrá ser implementada. Al realizar la comunicación entre π_2 y π_3 correspondiente al conjunto t_2 - p_5 - t_4, se debería contar como tercer interlocutor en la comunicación con el lugar p_2. Esto no es implementable en Ada 95, por lo que estas situaciones deben ser evitadas.

Por ello, siempre que una transición de un conflicto deba consultar el estado de algún lugar externo o sincronizador local reducido, esta transición no podrá ser compartida, lo cual debe ser tenido en cuenta en la solución del problema de cobertura.

Las figuras 4.29.c y 4.29.d corresponden a particiones de la red que respetan esta nueva regla. La primera asigna la implementación de t_2 a π_3. De este modo p_3 deja de ser lugar privado de π_2 y pasa a formar parte del sincronizador local reducido. El protoproceso π_2 se reduce a la transición t_6 (otra solución podría haber agrupado t_4 y t_6 en π_2). La partición de la figura 4.29.d asigna la transición t_2 a π_2. El lugar p_4 pasa a formar parte del sincronizador local reducido y la transición

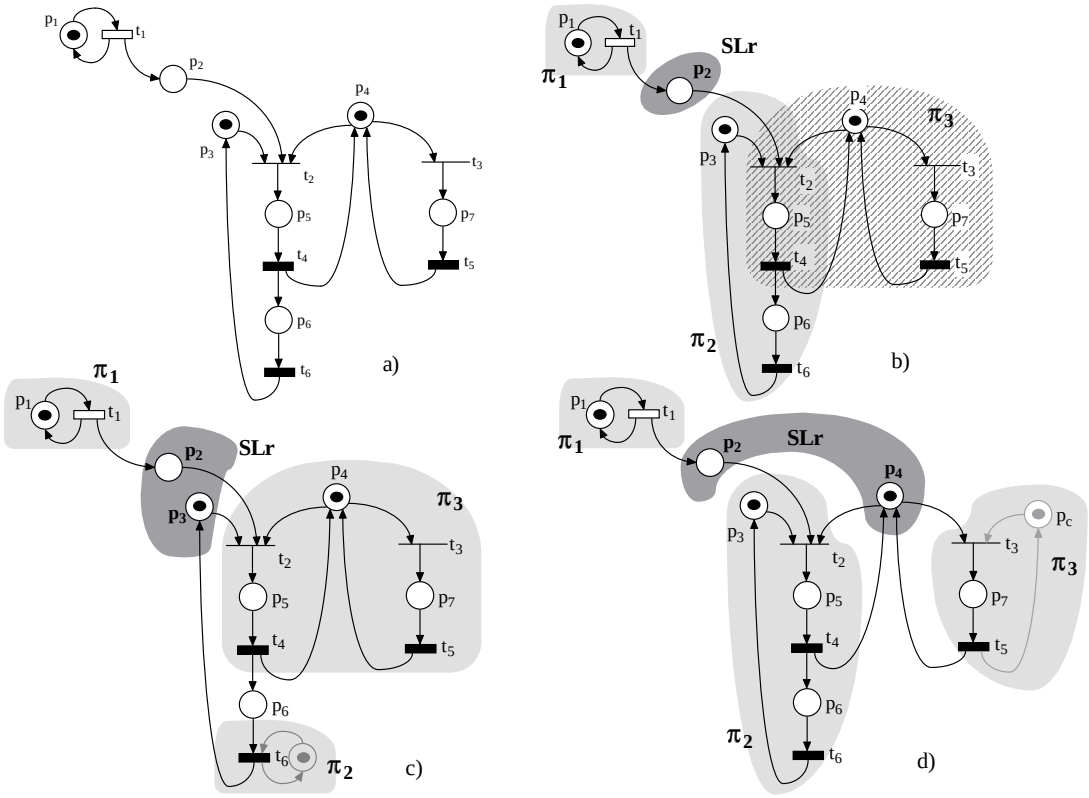


Figura 4.29: Red de Petri con conflicto y tres posibles implementaciones.

t_4 también es asignada exclusivamente a π_2, pues no es posible añadir un lugar de cierre implícito que cubra las transiciones t_3, t_4 y t_5 sin cubrir también t_2. El protoproceso π_3 es cerrado añadiendo un lugar implícito, p_c.

4.7.5. Método de implementación de conflictos simples y acoplados

Proponemos el siguiente procedimiento a seguir para la implementación de conflictos acoplados y simples, que, como hemos visto, se pueden considerar un caso particular de los primeros.

1. Calcular los elementos de partición de la red y plantear la tabla de cobertura.

2. Resolver la partición de la red distribuyendo las transiciones entre los protoprocesos si se desea, pero sin compartir transiciones en conflicto si éstas deben desmarcar lugares externos (de comunicación o pertenecientes a algún sincronizador local) al protoproceso donde son implementadas. De esta forma, si el lenguaje de implementación no lo permite, se evitan las comunicaciones a tres bandas.

3. Formar los sincronizadores locales reducidos del modo descrito en la sección 4.7.3. Esto puede significar la retirada forzosa de algún lugar privado a algún protoproceso, por lo que éste quedaría abierto (en el siguiente ejemplo veremos un caso en el que ocurre esto).

4. Cerrar los protoprocesos abiertos mediante la adición de lugares implícitos, de forma que el protoproceso resultante resulte monomarcado.

5. Si no se pueden añadir lugares implícitos a la red para cerrar los protoprocesos abiertos, la red no es implementable de la manera que se pretende. Por ello las transiciones en conflicto no podrán ser distribuidas.

4.7.6. Ejemplo

Proponemos para su implementación la red de la figura 4.30. En ella se muestran varios conflictos acoplados y simples. Mediante un cálculo de p-invariantes monomarcados se puede determinar la tabla de partición de la figura 4.31.

El conflicto acoplado que se presenta entre las transiciones t_2, t_3, t_7, t_8, t_{13} y t_{14} puede ser distribuido a partir de los elementos de partición mostrados en la figura 4.31. El caso de la transición t_{13} es especial, ya que, a pesar de que se puede compartir entre el protoproceso generado a partir de $T(I_7)$ y el de $T(I_8)$, esta transición desmarca un lugar externo a estos protoprocesos. Eso supondría una comunicación a tres bandas que, como ya se ha comentado, es no-implementable en Ada 95. Por ello el conflicto entre las transiciones t_{13} y t_{14}, que podía pertenecer al protoproceso generado por $T(I_7)$, debe ser distribuido. Ello obliga a añadir un lugar implícito para cerrar el protoproceso π_6.

El sincronizador puede calcularse utilizando las reglas descritas anteriormente. Estará integrado por los lugares p_8, p_{14}, p_{16}.

4.8. Resumen de resultados y aportaciones del capítulo. Líneas abiertas

En este capítulo se ha planteado la implementación distribuida de la red mediante la extracción de un conjunto de procesos concurrentes (que hemos llamado *protoprocesos*) que se comunican entre sí. De esta forma se recupera la concurrencia de la parte de control del sistema modelado, perdida en las implementaciones centralizadas. Así mismo se mejoran las prestaciones de la implementación con respecto a las centralizadas, ya que se disminuye el número de procesos y el de comunicaciones, y se aumenta la tolerancia a fallos.

La descomposición se ha abordado por medio del cálculo de conjuntos de lugares en exclusión mutua, *elementos de partición*, que cubren las transiciones de las unidades de ejecución que conforman la red. De este modo, se determina qué transiciones, cubiertas por cada elemento de partición, pueden pertenecer a qué protoproceso, cuya configuración definitiva es determinada resolviendo un *problema de cobertura* en el que los elementos cobertores son los elementos de partición, y los elementos a cubrir las transiciones de la red.

La solución del problema de cobertura producirá una partición de la red en protoprocesos que se comunican a través de primitivas síncronas, *transiciones compartidas* o conjuntos conexos de transiciones y lugares compartidos (*subredes compartidas*), o asíncronas, *lugares de comunicación*.

Para el cálculo de los elementos de partición se propone un cómputo de *p-invariantes monomarcados*. Estos elementos de partición son especialmente interesantes debido a la existencia de técnicas conocidas para su obtención, y, por añadidura, pueden ser utilizados para definir el *flujo de control* del protoproceso que generan. Sin embargo, no es posible cubrir todas las transiciones de una red con p-invariantes monomarcados, por lo que se ha propuesto un método para su construcción en redes que no los presentan. El método se apoya en el concepto de *tubería*.

Por último, y quizás sea la aportación más importante, se ha realizado un completo estudio de la partición de redes con conflictos (de libre elección, simple o acoplado), estudiando la posibilidad de la distribución de las transiciones en conflicto entre distintos protoprocesos.

El capítulo se apoya y se basa sobre todo en los resultados de [Vil90]. Frente a él y a otros trabajos relacionados, las principales aportaciones de este capítulo han sido:

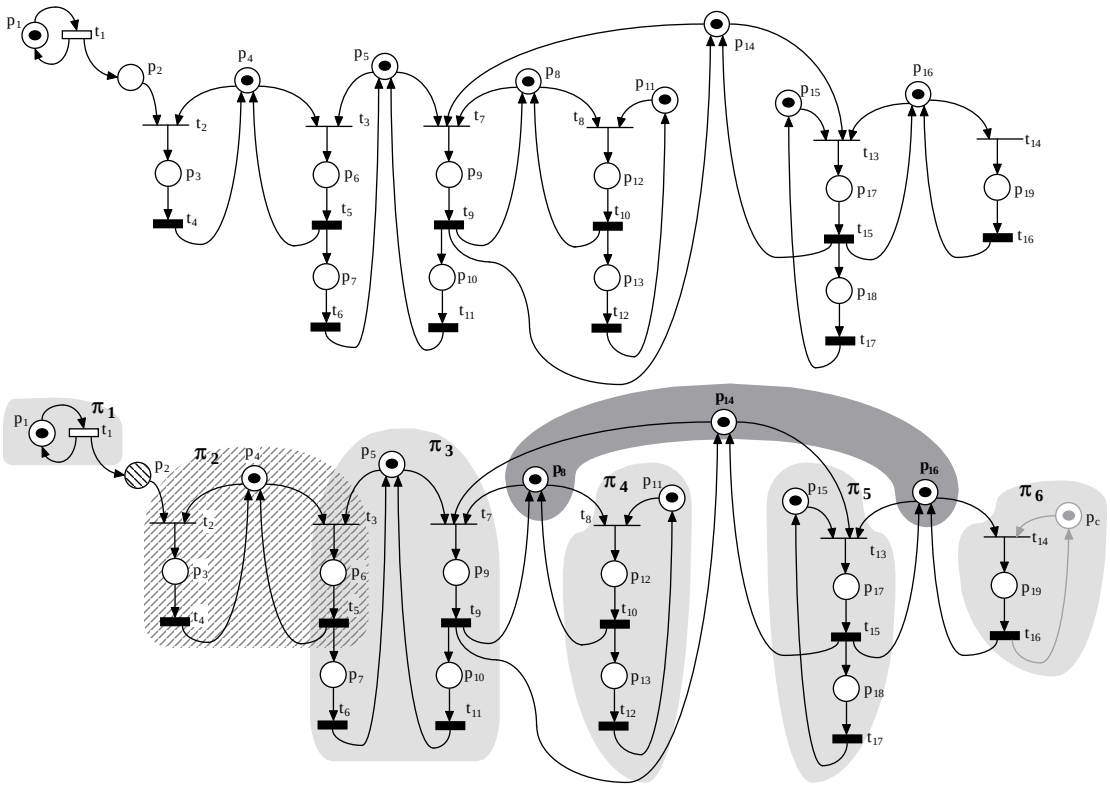


Figura 4.30: Red de Petri con conflictos acoplados y una partición propuesta

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	
$T(I_1)$	X																	Esencial
$T(I_2)$		X	X	X	X													Esencial
$T(I_3)$							X	X	X	X								
$T(I_4)$			X		X	X	X		X		X							Esencial
$T(I_5)$								X		X		X						Esencial
$T(I_6)$							X		X				X		X			
$T(I_7)$													X	X	X	X		Esencial
$T(I_8)$													X		X		X	Esencial

$I_1 = \{p_1\}$ $I_5 = \{p_{11}, p_{12}, p_{13}\}$

$I_2 = \{p_3, p_4, p_6\}$ $I_6 = \{p_9, p_{14}, p_{17}\}$

$I_3 = \{p_8, p_9, p_{12}\}$ $I_7 = \{p_{16}, p_{17}, p_{19}\}$

$I_4 = \{p_5, p_6, p_7, p_9, p_{10}\}$ $I_8 = \{p_{15}, p_{17}, p_{18}\}$

Figura 4.31: Tabla de cobertura de la red de la fig. 4.30

- Se ha realizado un esfuerzo en la formalización del método de cálculo de protoprocesos, de forma que sea sencillo de automatizar.
- Se ha constatado que las técnicas de implementación descentralizada existentes para RdP pueden ser aplicadas al caso de las RdPT, aunque todavía deben ser ampliadas para contemplar las restricciones temporales aportadas por las últimas.
- No se ha limitado el estudio a redes binarias y vivas, como en [Vil90], eliminando este requisito, que limitaba en gran medida la aplicabilidad de la técnica (piénsese que muchos sistemas reales tienen estados transitorios y finales que, como tales, no son vivos).
- Se han ampliado los tipos de comunicación entre los protoprocesos extraídos de la red de Petri con una nueva primitiva de comunicación síncrona, lo que abre nuevas posibilidades en la resolución del problema de cobertura que pueden simplificar la implementación, e incluso minimizar el número de procesos de la misma.
- Se ha sistematizado la construcción del flujo de control de los protoprocesos, de forma que su realización software sea más fácil de abordar.
- Se ha estudiado la equivalencia entre el concepto de tubería y de p-invariante monomarcado.
- Se ha propuesto un nuevo algoritmo de cálculo extensivo de tuberías
- Se ha estudiado la distribución de conflictos, eliminando la restricción impuesta en [Vil90], o en [CF93], por la que los conflictos debía ser implementado de forma indivisible en el interior de los protoprocesos. De esta forma es posible obtener particiones de la red con un menor número de protoprocesos y comunicados de forma más simple.
- Se ha propuesto el concepto de sincronizador local reducido, que simplifica la implementación de los conflictos acoplados.

En cuanto a las líneas de investigación abiertas, es destacable el estudio de la aplicación de la exclusión mutua temporal para reducir el número de procesos de la implementación de una RdPT. Este estudio pasaría por encontrar técnicas para la determinación de este tipo de exclusión mutua. Otra línea de interés estriba en el estudio de la planificabilidad de sistemas de tiempo real y su relación con la exclusión mutua temporal de las transiciones de la RdPT que los modelen.

Capítulo 5

Implementación descentralizada II. Realización software

5.1. Introducción

En el capítulo anterior se ha descrito como obtener una partición de una RdPT compuesta por una serie de protoprocesos, máquinas de estados secuenciales intercomunicadas, que deben ser ejecutadas concurrentemente. Cada protoproceso se ha generado agrupando conjuntos de lo que entonces denominamos *unidades de ejecución*, que constituyen los elementos básicos utilizados en el modelado de un sistema de tiempo real mediante RdPT. Recordaremos que estas unidades de ejecución estaban constituidas por: transiciones SYCO, encargadas de las acciones de control y sincronización; transiciones TIME y sus lugares de entrada, unidad encargada de las labores de temporización; y la unidad integrada por una transición CODE junto con sus lugares de entrada, que modela la ejecución de código.

La agrupación de un conjunto de unidades de ejecución en un protoproceso se ha realizado mediante la reunión de las transiciones de estas unidades, cuando éstas están cubiertas por una serie de lugares en exclusión mutua (*elementos de partición*, $\mathcal{EP}$). Los elementos de partición se obtienen a través del cálculo de p-invariantes monomarcados o por construcción de los mismos, mediante la adición de algún lugar implícito (*tubería*). Tal como se indicó en la sección 4.3.5, el interés de estos elementos de partición (p-invariantes monomarcados) no es únicamente la solución del problema de cobertura que da lugar a la partición de la red, sino la determinación del flujo de control de los protoprocesos: en cada momento sólo uno de los lugares de un $\mathcal{EP}$ estará marcado, por lo que sólo sus transiciones de salida podrán ser disparadas. Es decir, sólo las unidades de ejecución que incluyan este lugar podrán ser ejecutadas. En este sentido, los lugares del $\mathcal{EP}$ actuarán a modo de contador de programa del protoproceso a que dan lugar, pautando el orden de ejecución de las unidades de ejecución que lo conforman.

El capítulo recoge los resultados presentados recientemente en el trabajo [GV99].

Objetivos y métodos En el presente capítulo procederemos a la realización o implementación software de los protoprocesos que se han extraído de la RdPT. La estrategia general será estudiar la configuración de unidades de ejecución que integran el protoproceso, para lo que básicamente habrá que centrarse en las transiciones que hay a la salida de cada lugar del $\mathcal{EP}$. El resultado final de este capítulo será la proposición de una estructura de programación para la codificación de cada tipo de unidad de ejecución, o conjunto de ellas en conflicto. De este modo la generación de código para el proceso puede realizarse concatenando las estructuras de programación propuestas en el orden marcado por el flujo de control descrito por el $\mathcal{EP}$ (en la figura 5.1 se muestra, mediante

un ejemplo, un esquema del proceso a seguir). Cuando el flujo de control llegue a un determinado lugar del $\mathcal{EP}$ se deberán ejecutar las unidades de ejecución que lo contengan o que estén a su salida. Es posible que varias de estas unidades estén en conflicto (compartan lugares de entrada), siendo el tratamiento de estos conflictos será uno de los principales objetivos de este capítulo.

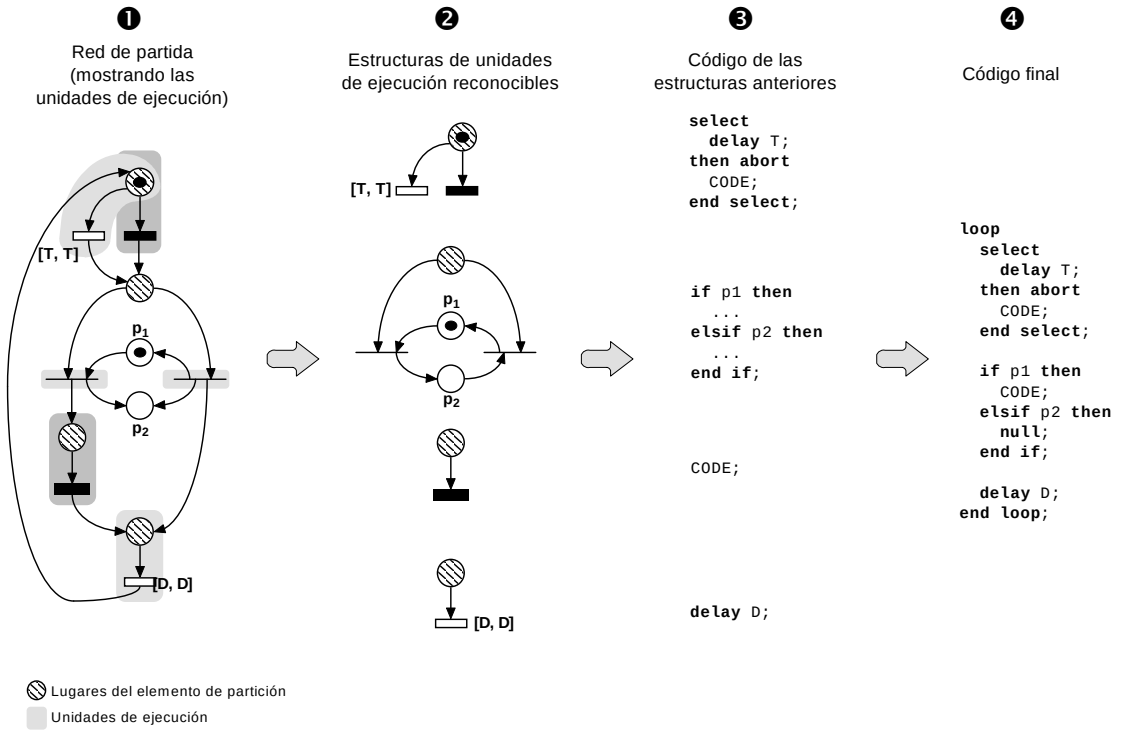


Figura 5.1: Proceso de realización software de una RdPT

Como se expondrá en la sección 5.11, la generación del código del protoproceso puede abordarse de dos maneras: generando *código estructurado*, o utilizando una *estructura* `case`. En la primera de las opciones, el código se obtendrá por concatenación de las diferentes unidades de ejecución que integran el protoproceso, según el orden que marque el flujo de control definido por el $\mathcal{EP}$. De esta forma, el $\mathcal{EP}$ está implícito en el proceso. En la segunda opción, los lugares del $\mathcal{EP}$ serán explícitamente implementados en cada una de las ramas del `case`, que implementarán las unidades de ejecución dependientes de cada lugar del $\mathcal{EP}$. En lo sucesivo, los ejemplos que se presenten en el texto serán codificados con código estructurado (esperamos así aumentar su legibilidad).

Sólo queda tomar una decisión, importante en cierta medida, pues, aunque se trata de dar una visión general sobre como implementar RdPT, hemos de elegir un lenguaje de programación para la codificación de los procesos. Esta decisión tendrá, en determinadas ocasiones, influencia sobre la partición de la red, ya que es posible que ciertas configuraciones determinadas en el proceso de descomposición sean no-implementables en el lenguaje elegido. Esto puede suponer el replanteamiento de la solución del problema de cobertura. En nuestro caso, hemos elegido Ada 95 ([ARM94, Rat95, Bar95, BW98]) como lenguaje de implementación, debido a las posibilidades que este lenguaje ofrece para la confección de programas multitarea y a sus primitivas para la gestión del tiempo. Estas características lo hacen adecuado para el desarrollo de sistemas de tiempo real, que es nuestro objetivo de partida. Además consideraremos el empleo del anexo de tiempo real

(`Ada.Real_time`). Tomada esta decisión, podemos avanzar la primera consecuencia de la misma sobre la implementación: la implementación software de un protoproceso será una *tarea Ada 95*.

Es nuestra intención, en la medida en que la complejidad de los conjuntos de instrucciones Ada 95 utilizados no lo desaconsejen, mostrar que el código obtenido en esta etapa de codificación es correcto en términos de que tenga el mismo comportamiento que la red que implementa. No se tratará de una demostración matemática rigurosa, sino más bien de una justificación. En esencia, nuestra propuesta será el empleo de unas determinadas primitivas Ada 95 para la implementación de unas determinadas estructuras de unidades de ejecución reconocidas en la RdPT. Será un proceso parecido a una traducción de la RdPT a Ada 95. Para que esta traducción sea correcta, deberemos justificar que el comportamiento de la instrucción Ada 95 utilizada es el mismo que el de la estructura para la cual ha sido propuesta. Para ello modelaremos el comportamiento de la instrucción Ada 95 mediante una RdPT[1]. Si esta red es la misma que la estructura de la red original que se está traduciendo, podremos deducir que el comportamiento es equivalente y, por tanto, que la implementación es correcta. La dificultad estribará en que los modelos de las instrucciones Ada 95 utilizadas son muy complejos, por lo que para poder compararlos a los de la red original, será necesario reducirlos utilizando reglas de reducción y transformación de redes de Petri y de redes de Petri con tiempo (ver anexo C).

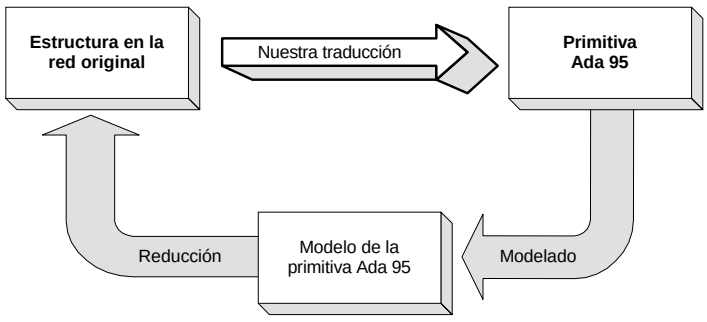


Figura 5.2: Proceso de justificación de la traducción de estructuras de RdPT a Ada 95

Estructura del capítulo Tras un breve comentario sobre los aspectos de planificación de los protoprocesos reconocidos en la partición de la red, en la sección 5.3 se comenta la codificación de los elementos básicos de un protoproceso (unidades de ejecución, lugares y transiciones) y de las primitivas de comunicación entre ellos. Las siguientes tres secciones se dedican el estudio de las situaciones de conflicto: conflictos internos (sección 5.4), conflictos simples (5.5) y conflictos acoplados (5.6), analizando las distintas posibilidades que pueden presentarse y proponiendo la primitiva más adecuada en cada caso. Las restantes secciones analizan una serie de situaciones particulares que no deben descuidarse: la sección 5.7 indica las precauciones a considerar cuando aparecen distintos tipos de transiciones en un conflicto, sugiriendo recomendaciones que tendrán influencia en la solución del problema de cobertura de transiciones que sirve para obtener los protoprocesos; esta influencia se sigue estudiando en las secciones 5.8, donde se repasan las situaciones en las que las limitaciones del lenguaje de codificación condiciona la partición de la red, y en la 5.9, donde se analiza la influencia del marcado inicial. Pese a todo, existen situaciones no implementables por el lenguaje de programación elegido. En la sección 5.10 se enumeran éstas, proponiendo una técnica mixta para su implementación. Para terminar, se hace un breve comentario sobre la generación de

[1] los modelos de las estructuras multitarea y de tiempo real de Ada 95 utilizadas pueden ser consultados en el capítulo 6

código basado en las indicaciones de codificación descritas en el capítulo.

5.2. Asignación de prioridades a los procesos. Planificación

Ya se comentó en la introducción (sección 2.3) que la plataforma de implementación (monoprocesador en este caso) imponía ciertas limitaciones que obligan a planificar, mediante asignación de prioridades, la ejecución de los protoprocesos reconocidos en el proceso partición.

No ha sido objetivo de esta tesis presentar una técnica de planificación por asignación de prioridades basada en redes de Petri con tiempo. Hasta el momento, proponemos la utilización de métodos heurísticos basados en técnicas existentes de planificación por prioridades: métodos de prioridades fijas como el *Rate Monotonic Scheduling* [LSD89, LL73, ABRW92], o el *Deadline Monotonic Scheduling* [LW80]; o métodos de prioridades dinámicas como el *Earliest Deadline First* (también pueden consultarse tratados generales sobre planificación en [BW97] o [But98]).

El resultado de la etapa de planificación será la asignación de una prioridad a cada protoproceso extraído de la red. Pueden utilizarse prioridades globales, en cuyo caso la prioridad será asignada a protoproceso en conjunto; o puede asociarse una prioridad, en principio distinta, para cada transición CODE que forme parte del protoproceso. En este último caso sería necesario el uso de prioridades dinámicas, ya que la prioridad del proceso que codifique el protoproceso deberá cambiar en función del trozo de código que ejecute. No se ha estudiado cuál de las estrategias es mejor, o cuál es la influencia de una u otra aproximación en las prestaciones de la implementación.

Prioridad de control y prioridad de código

En una RdPT se están modelando las dos partes de un sistema: la relacionada con el control (parte de control), y las acciones o código que el sistema debe ejecutar (parte operativa). Es necesario que las acciones de control sean realizadas en el mismo instante que se determine su ocurrencia, por lo que la prioridad de una acción de control debe ser superior a la de cualquier acción de código.

Si se opta por prioridades fijas, será necesario definir dos rangos de prioridades: uno para elementos de control, como lugares compartidos utilizados para gestionar el acceso en exclusión mutua, sincronizadores locales, etc. (ver próximas secciones en este mismo capítulo); y otro rango, inferior al anterior, para la planificación de los procesos tal como se entiende habitualmente. Esta diferenciación puede comprenderse más fácilmente si se compara con un sistema operativo, en el que el núcleo se ejecuta a la más alta prioridad, mientras que los procesos lo hacen a prioridades siempre inferiores.

Si se utilizan prioridades dinámicas, es posible promocionar la prioridad de un proceso cuando éste realice acciones de control, como, por ejemplo, el desmarcado de un lugar de comunicación (un ejemplo se tratará en la solución de conflictos simples, en la sección 5.5.2).

Así mismo adoptaremos el protocolo de techo de prioridad [SRL90, GS88] (más concretamente su variante proporcionada por el anexo de tiempo real de Ada 95 denominada *techo de prioridad inmediato* o *emulación de techo de prioridad* [BW97]) cuando sea necesario regular el acceso a alguna tarea servidor u objeto protegido.

Resolución de conflictos basados en prioridades

Además de las prioridades de los procesos durante su ejecución, determinadas durante la etapa de planificación, existe otro tipo de prioridades en el modelo. Se trata de las asociadas por el diseñador a las transiciones SYCO, y cuya función es deshacer las situaciones de conflicto en el modelo: en este sentido, cuando varias transiciones SYCO estén sensibilizadas se disparará aquella que tenga mayor prioridad asociada.

Durante la implementación software, se deberá tener en cuenta esta prioridad cuando aparezcan conflictos, proponiendo protocolos que la utilicen para su resolución. La forma de hacerlo será indicada en las futuras secciones, cuando se trate cada tipo de conflicto.

5.3. Implementación software de las unidades de ejecución

Tal como se ha avanzado en la introducción, los lugares del elemento de partición que genera un protoproceso actúan a modo de contador de programa del mismo, determinando su flujo de control. El flujo de marcas a través de los lugares de un elemento de partición irá determinando en cada momento, qué unidades de ejecución pueden ser ejecutadas. Por ejemplo, si a la salida de uno de estos lugares existe una transición CODE, sus acciones podrán comenzar. La generación del esqueleto del proceso a partir de los lugares del elemento de partición será tratado más adelante (ver sección 5.11 sobre generación de código).

La implementación de una unidad de ejecución está en gran parte determinada por el tipo de transición que la integra. En unos casos supondrá la ejecución de código, en otros la toma de decisiones de control, y en otros la ejecución de alguna acción temporal. Como regla general, la implementación de cualquier unidad de ejecución de una RdP debe considerar las siguientes tres fases:

1. *Test de sensibilización de la transición.* Aunque el marcado del lugar del elemento de partición habilite la ejecución de la transición de la unidad de ejecución, el estado del resto de sus lugares de entrada la condiciona. Se comenzarán las acciones asociadas a la transición sólo si todos sus lugares de entrada están marcados, esto es, la transición está sensibilizada.

2. *Ejecución de las acciones asociadas a la transición.* Si la transición está sensibilizada es posible comenzar la ejecución de las acciones que representa. En función del tipo de transición se ejecutará código, se promoverá un evento temporal, se tomará una decisión de control o se sincronizarán determinados procesos. La complejidad de la implementación generada dependerá de si la unidad de ejecución involucrada está o no en conflicto con otras de la red. Si es así, se debe tener en cuenta que el disparo de una de las transiciones supondrá la desensibilización del resto de transiciones en conflicto y, por tanto, el aborto de sus acciones asociadas. Considérese como ejemplo el caso de una transición CODE en conflicto con una TIME, que está representando un *timeout* sobre la ejecución de un código; el disparo de la transición TIME provoca que se abandone la ejecución del código.

3. *Disparo de la transición y actualización del marcado de la red.* La finalización de las acciones asociadas a la transición se representa mediante el disparo de la misma, y, en consecuencia, se deberá proceder a la actualización del marcado en la red. Para ello se desmarcarán los lugares de entrada y se marcarán los de salida de la transición, lo que dará paso a la ejecución de otras unidades de ejecución.

Comentemos seguidamente como implementar los lugares y las transiciones de las unidades de ejecución.

5.3.1. Implementación software de lugares

Los lugares son utilizados en la primera y tercera fase de la ejecución de una unidad de ejecución. El estado de los lugares es consultado durante la determinación de la sensibilización de la transición y es modificado, marcado o desmarcado, tras el disparo de la misma. Por ello, la implementación de un lugar debe proporcionar métodos para la realización de estas tres actividades:

- Comprobación del estado del lugar, para utilizarlo como condición que permita el disparo de sus transiciones de salida.
- Desmarcado del lugar
- Marcado del lugar

En algunas ocasiones, como en presencia de algún tipo de conflicto, determinados lugares que modelan *recursos compartidos* pueden ser consultados concurrentemente por varios procesos. En estas situaciones es de vital importancia garantizar que el proceso que compruebe el estado de un lugar sea el que consiga desmarcarlo. De esta forma se evitaran evoluciones erróneas de la red, como las que se comentaron en la sección 4.7 del capítulo anterior. Es necesario, por tanto, que las operaciones de comprobación del estado del lugar y desmarcado del mismo se hagan de forma atómica.

Con respecto a un proceso, un lugar puede ser privado, de comunicación o compartido (ver sección 4.3.4). Tomaremos esta clasificación como referencia para la descripción de la forma de implementación, pues cada tipo de lugar tiene características propias que deben ser reflejadas en su realización software. En la figura 5.3 se muestran tres fragmentos de redes en los que aparecen los distintos tipos de lugares que vamos a comentar.

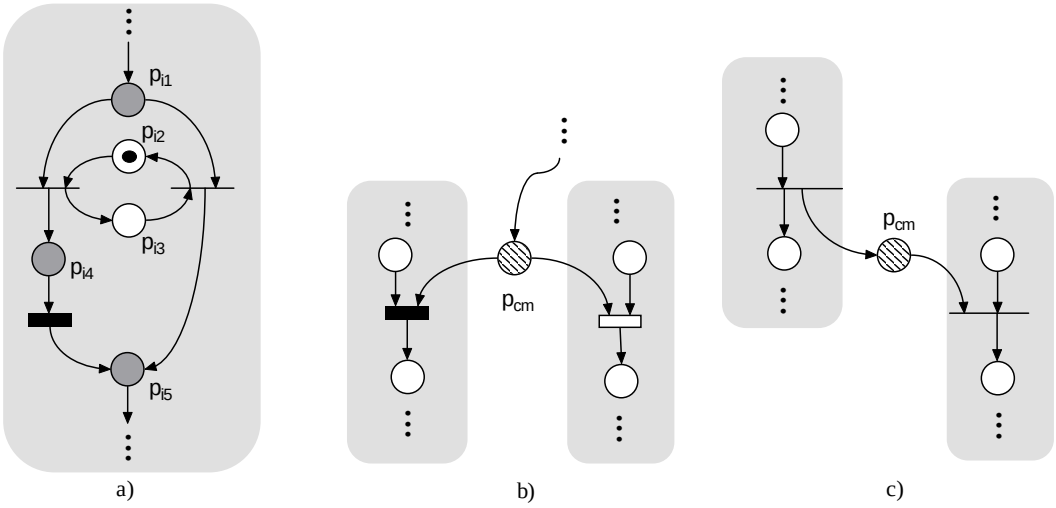


Figura 5.3: Distintos tipos de lugares: a) lugares internos, pertenecientes a elemento de partición (sombreados) u otros lugares internos; b) lugar de comunicación de destino múltiple; c) lugar de comunicación de destino privado

a) Implementación software de lugares privados

Los lugares privados de un protoproceso son utilizados para describir el estado interno del mismo y para promover y condicionar la ejecución de sus acciones. Entre ellos estarán los lugares del elemento de partición que da lugar al protoproceso y cuya implementación, como comentaremos en la sección 5.11, puede estar implícita en el flujo de control del proceso, si se genera código estructurado, o puede ser explícita si se opta por la realización de un interprete del protoproceso. Como ejemplo de la primera opción, considérense los lugares p_{i1}, p_{i4} y p_{i5} de la figura 5.3.a.

El resto de lugares internos simplemente condiciona el disparo de las transiciones de las unidades de ejecución (por ejemplo, los lugares p_{i2} y p_{i3} de la figura 5.3.a). Pueden ser consultados, marcados

o desmarcados únicamente por el protoproceso que los contiene, por lo que no es posible que haya interacciones externas que obliguen a proteger las operaciones sobre este tipo de lugares. Por ello la forma más eficiente de implementarlos es mediante una *variable local* al proceso que los contiene: si el lugar es 1-limitado, será utilizada una variable booleana, que contendrán valor `TRUE` si el lugar está marcado y `FALSE` en caso contrario; si no es 1-limitado, se deberá emplear una variable entera cuyo valor represente el número de marcas del lugar.

En el test de sensibilización, para la comprobación del estado del lugar será suficiente el planteamiento de una condición basada en el valor de la variable que implementa el lugar. Para la implementación de las operaciones de marcado y desmarcado, bastará con actualizar convenientemente el valor de la variable.

b) Implementación software de lugares de comunicación

Un lugar de comunicación no pertenece a ningún protoproceso, por lo que deberá ser implementado externamente a ellos, en forma de algún objeto que pueda ser consultado y modificado en exclusión mutua por los procesos de la implementación. Por ello, estas operaciones se deberán encapsular en el interior de algún objeto Ada 95 que garantice el acceso en exclusión mutua, esto es, una *tarea* o un *objeto protegido*, prefiriéndose el último, debido a su carácter pasivo (de esta forma no se incrementará el número de procesos de la implementación).

El objeto protegido también debe permitir la comprobación del estado (marcado) del lugar, que será encapsulado en su interior, representándolo, por ejemplo, por medio de una variable, booleana o entera, dependiendo de si el lugar es o no 1-limitado. Esta operación será bloqueante, para evitar una espera activa del proceso antes de la ejecución de la unidad de ejecución. Así mismo, si el lugar puede ser desmarcado por varios procesos concurrentemente, se debe proporcionar un método para comprobar si el lugar deja de estar marcado. Esto será de utilidad para abortar la ejecución de las unidades de ejecución que dependan del lugar, en el momento en que sea desmarcado (trataremos este tema en profundidad cuando hablemos sobre conflictos entre distintos tipos de transiciones en la sección 5.7). Un ejemplo de este tipo de lugares se muestra en la figura 5.3.b, lugar p_{cm}.

Los lugares de comunicación desempeñan en la implementación tareas relacionadas con el control de la red, por ello deberán ser más prioritarios que los procesos que los marcan o desmarcan. Además, se deberá imponer el protocolo de *techo de prioridad*, en su variante proporcionada por el anexo de tiempo real de Ada 95. De esta manera se evitará la inversión de prioridad no acotada en el acceso al lugar.

Proponemos, por tanto, la siguiente estructura de implementación para un lugar de comunicación:

```
protected Lugar_Comunicacion is
  entry Esta_Marcado;
  entry Esta_Desmarcado;
  entry Desmarcar;
  procedure Marcar;
private
  Marcado: integer := ...; -- Valor inicial de marcado
end Lugar_Comunicacion;

protected body Lugar_Comunicacion is
  entry Esta_Marcado when Marcado > 0 is
  begin
    null;
  end;
```

```
  entry Esta_Desmarcado when Marcado = 0 is
  begin
    null;
  end;

  entry Desmarcar when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
  end;

  procedure Marcar is
  begin
    Marcado := Marcado + 1;
  end;
end Lugar_Comunicacion;
```

c) Simplificación de la implementación software de lugares de comunicación.

La anterior es la manera más general de implementar un lugar de comunicación. Sin embargo, en ciertas situaciones puede simplificarse:

1. Si el *lugar es de destino único* (un sólo proceso puede desmarcarlo),
2. o *su destino son transiciones SYCO* (que desmarcan el lugar inmediatamente después de detectar su sensibilización, sin involucrar la ejecución de ninguna acción)

Un ejemplo de este tipo de lugares se muestra en la figura 5.3.c, lugar p_{cm}. En ambos casos no será necesario que el lugar proporcione operaciones de comprobación del marcado (`Esta_Marcado` y `Esta_Desmarcado`). Como hemos comentado, esta operación sólo tiene sentido cuando la ejecución de determinadas acciones (transiciones CODE o TIME) dependen del estado de un lugar de comunicación, y éste pueda ser desmarcado asíncronamente por otro proceso. En ese caso sí es necesario vigilar que el lugar no sea desmarcado, pues ello supondría el aborto de las acciones que se están ejecutando. Sin embargo, en los casos que aquí se refieren, esta operación no tiene sentido, porque, si es de destino único, nadie compite por el lugar y nadie puede desmarcarlo asíncronamente, y en el segundo caso, no hay nada que abortar si el lugar se desmarca.

Proposición 5.3.1 (Implementación de lugar de comunicación como objeto protegido) *El siguiente objeto protegido corresponde con la implementación de un lugar de comunicación de destino único o cuyo destino son transiciones SYCO.*

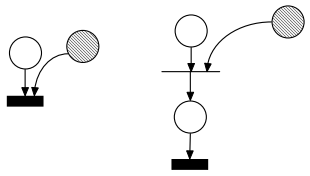
```
protected Lugar_Comunicacion_Simple is
  entry Desmarcar;
  procedure Marcar;
private
  Marcado: integer := ...; -- Valor inicial de marcado
end Lugar_Comunicacion_Simple;

protected body Lugar_Comunicacion_Simple is
  entry Desmarcar when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
  end;
```

```
  procedure Marcar is
  begin
    Marcado := Marcado + 1;
  end;
end Lugar_Comunicacion_Simple;
```

JUSTIFICACIÓN Consúltese el anexo E △

Notar que la implementación es la de un *semáforo*. En este caso, la barrera asociada a la entrada `Desmarcar` representa el estado del lugar. Así, una transición que vaya a ser disparada verá bloqueado su disparo hasta que la barrera sea cierta (sincronización basada en condiciones), momento en el cual el lugar es desmarcado. Esto es conceptualmente correcto si se trata de transiciones SYCO, ya que su disparo es instantáneo tras la determinación de la sensibilización de la transición. Sin embargo implica una transformación en el disparo de una transición TIME o CODE, para no violar el orden de la secuencia de pasos para la ejecución de una unidad de ejecución (test de sensibilización - ejecución - actualización de marcado). Para permitir que el lugar de comunicación sea desmarcado antes de comenzar las acciones asociadas a la transición, se debe realizar la transformación que se muestra en la siguiente figura. La transformación es correcta porque el lugar es de destino privado y puede ser desmarcado únicamente por la transición CODE.



d) Implementación software de lugares de comunicación. Versión activa.

A pesar de preferirse la implementación utilizando un objeto protegido, porque es pasiva y de este modo se reduce el número de procesos de la implementación, existen situaciones en las que no puede utilizarse. La implementación de ciertas configuraciones de conflictos (sección 5.4.2) hará necesario que el lugar de comunicación deba ser implementado de forma activa, a modo de *tarea mensajero* [NS87], que realice las llamadas para desmarcar el lugar, en vez de aceptarlas. Por ello proponemos esta segunda implementación de lugar de comunicación, en versión activa, que solamente es válido para lugares binarios y de destino único. Nótese que en este caso no es necesario implementar explícitamente el marcado del lugar de comunicación, ya que las acciones de marcado y desmarcado están representadas mediante dos citas entre los procesos que comunica y la tarea que representa el lugar. Supondremos que inicialmente el lugar está desmarcado (es posible realizar una implementación independiente del marcado inicial).

Proposición 5.3.2 (Implementación de lugar de comunicación como tarea mensajero) *La siguiente tarea corresponde con la implementación de un lugar de comunicación binario y que puede ser desmarcado por un único proceso (destino único).*

```
task Lugar_Comunicacion_Activo is
  entry Marcar;
end Lugar_Comunicacion_Activo;
```

```
task body Lugar_Comunicacion_Activo is
begin
  loop
    accept Marcar;
    Proceso.Desmarcar; -- Llamada a la transición del proceso que desmarcará el lugar
  end loop;
end Lugar_Comunicacion_Activo;
```

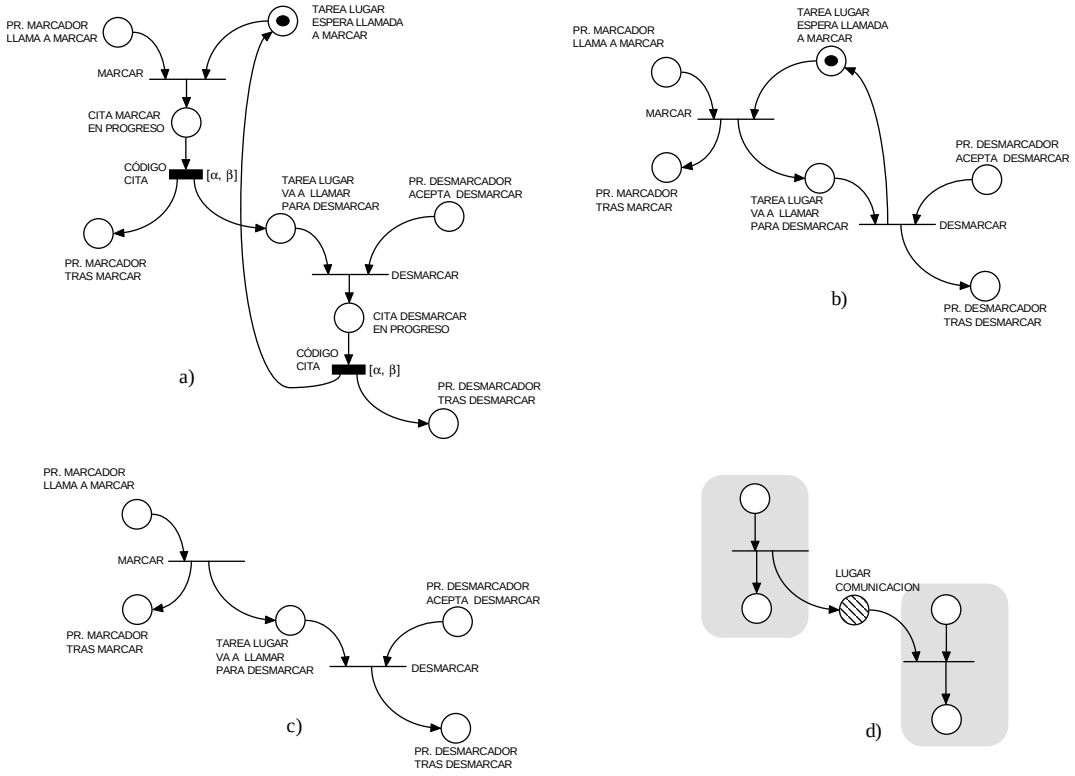


Figura 5.4: a)-c) Justificación de la implementación de un lugar de comunicación en su versión activa; d) Estructura típica de lugar de comunicación

JUSTIFICACIÓN El modelo de la figura 5.4.a corresponde al modelo de la tarea propuesta. En él se muestran los modelos de las dos citas, `Marcar` y `Desmarcar` (adaptaciones del mostrado para una cita en la sección 6.3.1). La tarea `Lugar_Comunicacion_Activo` (en lo que sigue, tarea–lugar) espera, en el lugar tarea–lugar espera llamada a marcar, a que un proceso marcador realice una llamada a su entrada Marcar. Tras la ejecución de esta cita, la tarea–lugar pasará a realizar la llamada en la cita `Desmarcar`, tras la cual volverá a la situación inicial.

Debido a que en el interior de las citas no se realiza ninguna operación, las transiciones CODE código cita pueden ser eliminadas, obteniéndose el modelo de la figura 5.4.b. En ella, el lugar tarea–lugar espera llamada a marcar es el complementario de tarea–lugar va a llamar para desmarcar. Éste lugar es 1-limitado, ya que el sistema que se está modelando es un lugar binario, lo que supone que una vez realizada la cita de marcar (transición marcar), no se puede producir otro disparo de esta transición hasta que no se lleve a cabo la acción de desmarcar (disparo de la transición desmarcar). Por todo ello, el lugar tarea–lugar espera llamada a marcar es implícito y puede eliminarse (anexo C,

regla C.10). Al hacerlo se obtiene el modelo de la figura 5.4.c, en el que se reconoce directamente la estructura de un lugar de comunicación (figura 5.4.d).

e) Lugares compartidos

Hablaremos de ellos más adelante al referirnos a la implementación de varias transiciones y lugares compartidos agrupados en una subred. Simplemente avanzaremos que se trata de lugares internos al fragmento de red que es compartido entre dos protoprocesos, por lo que su implementación será similar a la indicada para los lugares privados.

5.3.2. Implementación software básica de las transiciones de una RdP con tiempo

Dedicaremos esta sección a comentar como se programarán las acciones correspondientes a los tres tipos de transiciones que hemos distinguido en una RdPT: transiciones SYCO, CODE y TIME. En esta primera aproximación nos referiremos a la implementación software de estas transiciones cuando no están en conflicto, por lo que su disparo puede considerarse de forma autónoma y sin competir con el de otras en la red. Distinguiremos los casos en los que puede existir comunicación con otras partes de la red, es decir, la transición está compartida, de la situación en la que la transición no es compartida y se aloja en el interior de un protoproceso.

a) Implementación de transiciones no compartidas y no en conflicto

Las acciones asociadas a las transiciones comenzarán a partir del momento en el que se detecte su sensibilización, y diferirán dependiendo del tipo de transición del que se trate:

- Si la transición es de tipo SYCO, no se ejecutará ninguna acción. El disparo de estas transiciones supone únicamente cambios de estado (marcado) en la red.
- Si la transición es CODE, el código asociado a su unidad de ejecución será ejecutado.
- Si la transición es TIME, la ejecución de su unidad supondrá la suspensión del proceso durante el tiempo especificado en el intervalo temporal de la transición, $[D, D]$. Esta suspensión se implementará en Ada 95 mediante la instrucción `delay D`, ya que el modelo de esta instrucción cuando el valor de $D > 0$, corresponde con una transición TIME (ver sección 6.2 en el capítulo 6).

El test de sensibilización de esta clase de transiciones se reducirá a la comprobación del estado de sus lugares de entrada, que son todos privados. Para esta comprobación basta utilizar una estructura condicional tipo `if`. En consecuencia, la implementación completa de este tipo de transiciones, junto con sus lugares de entrada, queda como sigue:

```
if Test_sensibilizacion then
  ... -- Acciones asociadas a la transición (código, delay o nada)
  -- Desmarcar lugares de entrada
  -- Marcar lugares de salida
end if;
```

donde el test de sensibilización se refiere al estado de los lugares de entrada internos, que son implementados como variables locales.

b) Implementación de transiciones compartidas y no en conflicto

La implementación de transiciones compartidas presenta la dificultad añadida de que la determinación de la sensibilización de la transición deberá realizarse de forma conjunta en los procesos que comparten la transición. Los lugares de entrada estarán repartidos entre los procesos, por lo que se deberá esperar a que el flujo de control de ambos alcance el punto de ejecución de la transición. Esta situación es implementable por medio de una *cita* Ada 95. Esta primitiva permite la comunicación entre dos tareas, por lo que sólo seremos capaces de implementar transiciones compartidas entre dos protoprocesos. Un proceso cualquiera de los dos interpretará el papel de tarea llamadora (cliente) y el otro el de tarea llamada (servidor), de modo que el cliente realice la llamada al punto de entrada y el servidor contenga la instrucción `accept`.

Proposición 5.3.3 (Implementación software de una transición compartida) *La implementación de una transición compartida entre dos procesos de una RdP corresponde con la ejecución de una cita entre ambos procesos. Esto es:*

```
Proceso 1                                          Proceso 2

accept Transicion_Compartida do                    Proceso1.Transicion_Compartida;
  ... -- Acciones asociadas a la transición
end;
```

JUSTIFICACIÓN Si consideramos el modelo básico de una cita Ada entre dos tareas obtenemos el modelo de la figura 5.5.a (se puede consultar una descripción más detallada del mismo en el capítulo 6, sección 6.3.1). El modelo está simplificado y no contempla otras características de la cita, como el mecanismo de parámetros, la elevación ni manejo de excepciones, el aborto de llamadas en la cita, ni la cola de llamadas asociada a la entrada, ya que no son necesarios en este ámbito.

El modelo puede ser fácilmente transformado en el de la figura 5.5.b por medio de fusiones de lugares serie. Aplicando la regla de *fusión serie de transiciones con tiempo* (anexo C, regla C.6), la transición CODE y la transición cita pueden ser reunidas, obteniéndose la figura 5.5.c. que corresponde directamente con la situación básica de transición compartida entre dos procesos en una RdP que se muestra en la figura 5.5.d. △

La ejecución de las acciones de la transición se realizará en el cuerpo de la cita. Si se trata de una transición SYCO, no se hará nada; se ejecutará el código asociado, si la transición es CODE; o se suspenderá la ejecución mediante un `delay` si la transición es TIME.

La última fase del disparo, actualización del marcado, se realizará como en una transición normal, pero cada proceso actualizará el conjunto de lugares de entrada y salida que le corresponda.

Por todo ello, la implementación completa de una unidad de ejecución que incluya una transición compartida (figura 5.5.e) que tenga varios lugares de entrada (además de los propios del elemento de partición) tendrá un esquema como el que sigue:

```
Proceso 1                                          Proceso 2

if <Test_sensibilización_en_Proc_1> then           if <Test_sensibilización_2> then
  accept Transicion_Compartida do                     Proceso1.Transicion_Compartida;
    ... -- Acciones asociadas a la transición         -- Desmarcar lugares de entrada en Proc 2
  end;                                                -- Marcar lugares de salida en Proc 2
  -- Desmarcar lugares de entrada en Proc 1        end if;
  -- Marcar lugares de salida en Proc 1
end if;
```

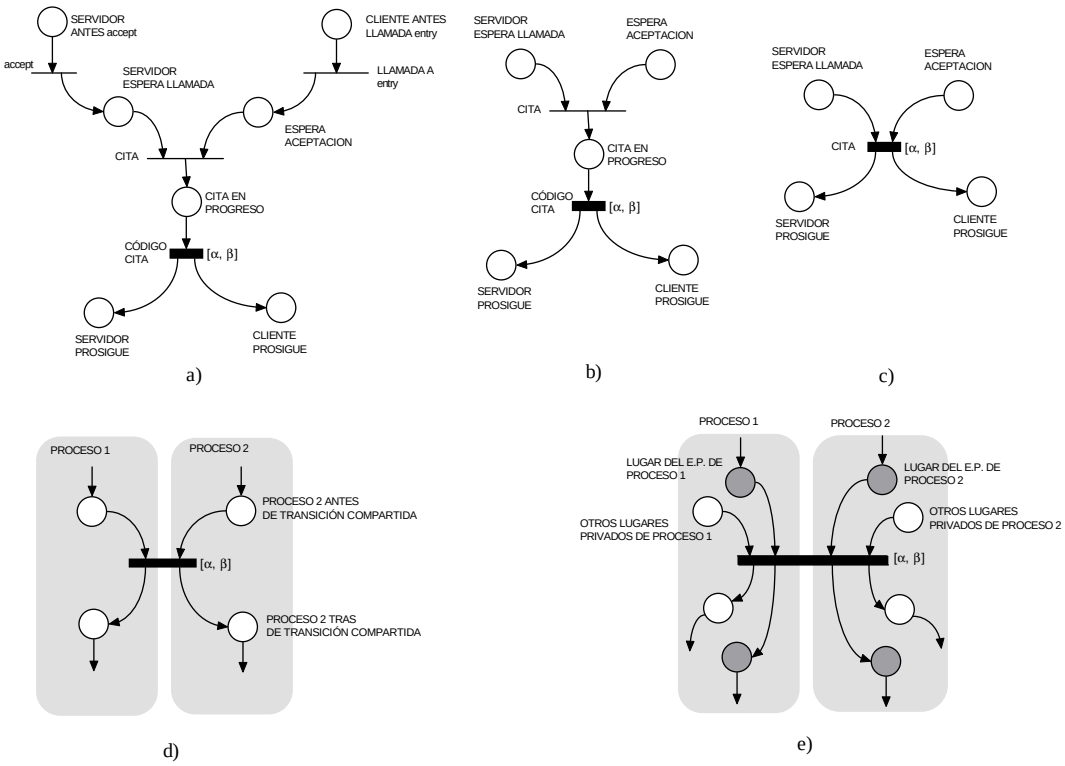


Figura 5.5: a)-c) Justificación de la implementación de una transición compartida como una cita Ada; d) Transición compartida elemental; e) Transición compartida con más lugares de entrada y salida

donde el test de sensibilización se refiere al estado de los lugares de entrada internos implementados como variables locales.

c) Consecuencias de la implementación de transiciones en Ada 95 sobre la partición de la red

1. Como la cita Ada 95 es un mecanismo de comunicación entre dos tareas, sólo se podrán implementar transiciones compartidas entre dos procesos. Por ello, si durante la solución del problema de cobertura se detecta que una transición puede pertenecer a más de dos protoprocesos, esta transición no podrá ser compartida, debiendo ser asignada a un único protoproceso.

2. Por el mismo motivo, no podrán implementarse situaciones en las que una transición compartida tenga como lugar de entrada un lugar de comunicación, ya que esto supondría una cita a tres. Del mismo modo, si esto ocurre la transición no podrá ser compartida, debiendo ser asignada a un único protoproceso.

d) Implementación de transiciones y lugares compartidos agrupados en una subred

Trataremos ahora de la última forma de comunicación entre procesos descrita en la sección 4.3.3. Como ya se mencionó entonces, se trata de un método de comunicación más general que la

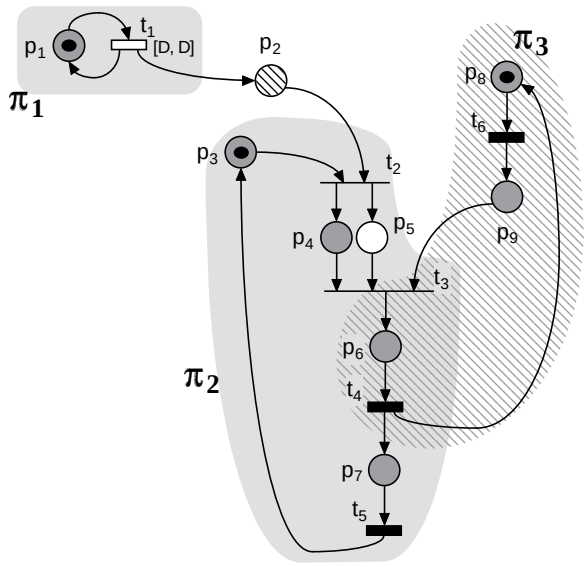


Figura 5.6: Red de ejemplo para su implementación software. Los lugares sombreados corresponden a lugares pertenecientes a elementos de partición. El lugar rayado es un lugar de comunicación

simple transición compartida, pero su implementación es la misma: una cita en la que se ejecutarán las acciones correspondientes a las unidades de ejecución que conforman la subred compartida, o trozo de protoproceso compartido. La justificación sigue el mismo esquema y razonamiento que el usado para la proposición 5.3.3. Si, en el modelo de la cita, en vez de considerar una única transición (código cita) para representar las acciones desarrolladas en ella, consideramos un modelo de las mismas más complejo, integrado por una subred compuesta de varios lugares y transiciones compartidos, estaremos ante la situación que queremos implementar.

Por ello se propone la siguiente estructura de implementación software:

```
Proceso 1                                          Proceso 2

if <Test_sensibilización_en_Proc_1> then        if <Test_sensibilización_2> then
  accept Subred_Compartida do                      Proceso1.Subred_Compartida;
    ... -- Acciones asociadas a la subred          -- Desmarcar lugares de entrada en Proc 2
  end;                                             -- Marcar lugares de salida en Proc 2
  -- Desmarcar lugares de entrada en Proc 1     end if;
  -- Marcar lugares de salida en Proc 1
end if;
```

donde el test de sensibilización se refiere al estado de los lugares de entrada internos implementados como variables locales.

Ejemplo

La red de la figura 5.6 ha sido descompuesta mediante la técnica descrita en el capítulo anterior en tres protoprocesos que darán lugar a tres tareas Ada 95. Los lugares de los elementos de partición que han originado los procesos y que marcan su flujo de control se muestran sombreados; el lugar de comunicación de destino privado (p_2) (rayado) y el resto de lugares en blanco. Se puede apreciar que existe un conjunto de transiciones y lugares compartidos (t_3 - p_6 - t_4) entre los protoprocesos π_2 y π_3. El código que implementará los tres procesos y el lugar de comunicación será el siguiente:

```
protected Lugar_P2 is
  entry Desmarcar;
  procedure Marcar;
private
  Marcado: integer := 0;
end Lugar_P2;

protected body Lugar_P2 is
  entry Desmarcar when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
  end;
  procedure Marcar is
  begin
    Marcado := Marcado + 1;
  end;
end Lugar_P2;

task Proc_1;

task body Proc_1 is
begin
  loop
    delay D;                    -- Transición t1
    Lugar_P2.Marcar;
  end loop;
end Proc_1;

task Proc_2 is
  entry Transicion_T3;
end Proc_2;
task body Proc_2 is
  Lugar_P5: boolean := FALSE;
begin
  loop
    Lugar_P2.Desmarcar;         -- Disparo de t2
    Lugar_P5 := TRUE;
    if Lugar_P5 then            -- Ejecución de t3-p6-t4 (compartido)
      accept Transicion_T3 do   -- Transición t3
        Codigo(T4);             -- Transición t4
        Lugar_P5 := FALSE;
      end;
    end if;
    Codigo(T5);                 -- Transición t5
  end loop;
end Proc_2;

task Proc_3;
task body Proc_3 is
begin
  loop
```

```
    Codigo(T6);                -- Transición t6
    Proc_2.Transicion_T3;      -- Ejecución de t3-p6-t4 (compartido)
  end loop;
end Proc_3;
```

5.3.3. Evitación de la deriva acumulativa en la implementación de transiciones TIME. Influencia en el resto de la implementación

Si implementamos las transiciones TIME con simples instrucciones `delay` produciremos procesos periódicos que presentan deriva acumulativa. Consideremos la implementación del Proceso 1 en el anterior ejemplo:

```
task body Proc_1 is
begin
  loop
    delay D;                    -- Transición t1
    Lugar_P2.Marcar;
  end loop;
end Proc_1;
```

Fácilmente puede comprenderse que el proceso coloca periódicamente una marca en el lugar p_2. Pero, de forma igualmente sencilla, se comprobar que esta implementación de un proceso periódico produce deriva acumulativa, debido a que el proceso puede ser desalojado del procesador durante más tiempo del indicado en el *delay*, y a que la operación de marcado del lugar y la ejecución de la instrucción `delay` no son instantáneas. Es este último motivo el que merece más atención por nuestra parte, ya que, hasta ahora, habíamos supuesto que las acciones de control de la red (tests de sensibilización, sincronizaciones, marcado y desmarcado de lugares, ...) eran instantáneas, como en el modelo teórico, cuando en realidad no lo son. La acumulación de los tiempos de ejecución de estas acciones puede provocar deriva acumulativa en el caso de la ejecución de una unidad con transición TIME.

Para evitar esta deriva proponemos la utilización de una variable de tipo `Ada.Real_time.Time` cuya función será registrar el instante en el que se produce el disparo de las transiciones en un proceso que contenga transiciones TIME. Llamaremos a esta variable `T_ultimo_disparo`, que será inicializada con valor `CLOCK` en el momento de su declaración. Su influencia en el código de implementación de las transiciones es la siguiente:

- El `delay` propio de la ejecución de una transición TIME será sustituido por estas dos instrucciones:

  ```
  delay until T_ultimo_disparo + D;
  T_ultimo_disparo := T_ultimo_disparo + D;
  ```

  debiendo ser el valor de `T_ultimo_disparo` el del instante de sensibilización de la transición TIME. De esta forma queda registrado exactamente el instante en el que se produce el disparo, instante que puede ser tenido en cuenta en el cálculo del siguiente tiempo de disparo, evitándose, de esta forma, la deriva acumulativa.

- La ejecución de una transición CODE deberá terminar con una instrucción que registre el instante en el que este código termina y marca sus lugares de salida (sensibilizando nuevas transiciones)

  ```
  Codigo_Transicion;
  T_ultimo_disparo := CLOCK;
  ```

- La ejecución de una transición SYCO no debe modificar el valor de esta variable, ya que estas transiciones son de disparo inmediato. Sólo será necesario cuando el disparo de la transición SYCO suponga una comunicación con algún otro elemento de la red, es decir, cuando la transición sea compartida o cuando su disparo desmarque algún lugar de comunicación. Volveremos sobre esto más adelante.

De esta forma, en las instrucciones `delay until` correspondientes a transiciones TIME, se absorbe la deriva acumulativa introducida por las acciones de control.

Con estas consideraciones la implementación del proceso 1 en el ejemplo anterior resultaría de la siguiente manera. Puede observarse que ya no presenta deriva acumulativa.

```
task body Proc_1 is
  T_ultimo_disparo: TIME := CLOCK;
begin
  loop
    delay until T_ultimo_disparo + D;         -- Transición t1
    T_ultimo_disparo := T_ultimo_disparo + D;
    Lugar_P2.Marcar;
  end loop;
end Proc_1;
```

Simplificación en el uso de la variable `T_ultimo_disparo`

En principio hemos considerado que el disparo de cualquier transición TIME o CODE deberá actualizar el valor de la variable `T_ultimo_disparo` del proceso. Imaginemos un proceso cíclico compuesto por una transición TIME seguida de una CODE. La implementación debería ser:

```
task body Ejemplo is
  T_ultimo_disparo: TIME := CLOCK;
begin
  loop
    delay until T_ultimo_disparo + D;         -- Transición TIME
    T_ultimo_disparo := T_ultimo_disparo + D;
    Codigo_Transicion_CODE;                   -- Transición CODE
    T_ultimo_disparo := CLOCK;
  end loop;
end Ejemplo;
```

Es evidente que los esfuerzos por evitar la deriva acumulativa son vanos a partir del momento en el que aparece la instrucción `T_ultimo_disparo := CLOCK` asociada al disparo de la transición CODE. Este ejemplo nos hace replantearnos la utilización de la variable `T_ultimo_disparo`.

Una transición CODE supone la ejecución de un código, que tiene un tiempo de ejecución y es desalojable del procesador. En este caso se puede considerar el tiempo de ejecución del control (transiciones SYCO, actualizaciones de marcado, ...) como parte del tiempo de ejecución del código. No tiene sentido, por tanto, la utilización de la variable `T_ultimo_disparo` cuando la secuencia de transiciones del proceso en ejecución incluye una transición CODE.

En consecuencia, utilizaremos la variable `T_ultimo_disparo` solamente en los procesos con transiciones TIME en los que exista un camino de ejecución que no atraviese ninguna transición CODE, pues en este camino se podrán registrar con exactitud los instantes de disparo de las transiciones.

Transmisión entre procesos del valor de `T_ultimo_disparo`

Se ha comentado que las transiciones SYCO no deben modificar el valor de la variable `T_ultimo_disparo` pues su disparo es instantáneo. Sin embargo existen situaciones en las que este disparo puede retrasarse debido a alguna interacción con el exterior del proceso, es decir, si la transición está compartida o tiene algún lugar de comunicación como lugar de entrada (ver las situaciones en la figura 5.7).

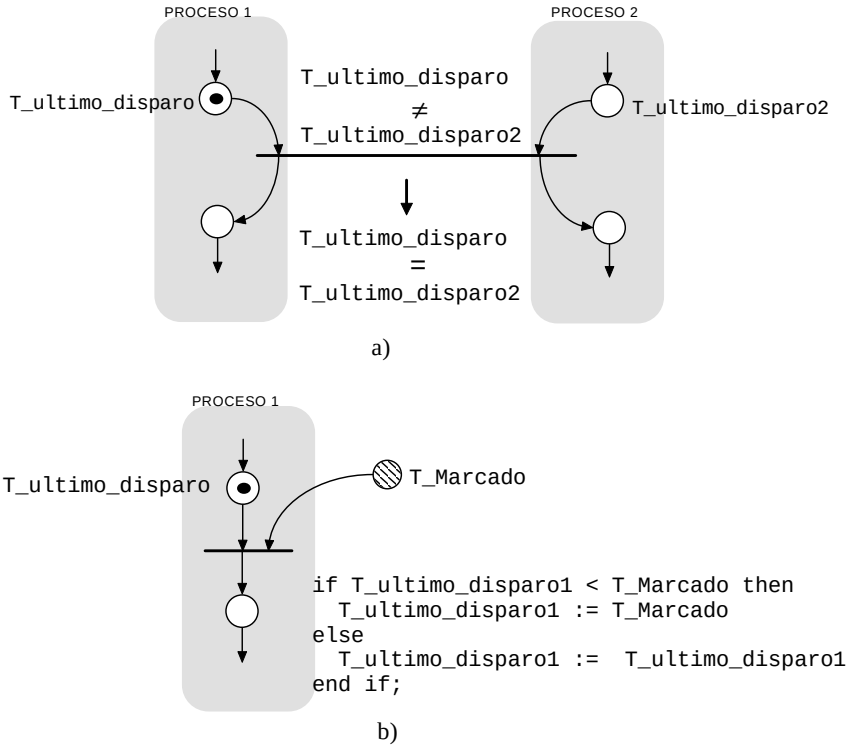


Figura 5.7: Situaciones en las que se debe transmitir el valor de la variable `T_ultimo_disparo`

En el caso de la transición compartida (figura 5.7.b), los procesos que realizan la cita no tienen por qué alcanzar el punto de disparo de la transición en el mismo instante. Sin embargo continuarán con el mismo valor de variable `T_ultimo_disparo` tras el disparo, por lo que deben sincronizarse. Para ello se debe incluir un parámetro de entrada/salida en la cita que almacene el instante en el que el proceso que hace la llamada alcanza el punto de disparo de la transición. En el cuerpo de la cita, este valor se compara con el de `T_ultimo_disparo` del proceso que contiene el `accept`, de modo que el mayor de los dos sea el nuevo valor de `T_ultimo_disparo` en ambos procesos. De este modo, la ejecución de una cita que representa una transición compartida queda de la siguiente manera (supondremos que el `accept` de la comunicación está en el proceso 1):

- Para una transición SYCO:

```
accept Tr_Compartida_SYCO(T_ultimo_disparo2: in out TIME) do
  if T_ultimo_disparo < T_ultimo_disparo2 then
    T_ultimo_disparo := T_ultimo_disparo2;
  else
```

```
        T_ultimo_disparo2 := T_ultimo_disparo;
      end if;
    end Tr_Compartida_SYCO;
```

- Para una transición CODE:

```
    accept Tr_Compartida_CODE(T_ultimo_disparo2: in out TIME) do
      -- Codigo_Transicion
      T_ultimo_disparo := CLOCK;
      T_ultimo_disparo2 := T_ultimo_disparo;
    end Tr_Compartida_CODE;
```

- Para una transición TIME:

```
    accept Tr_Compartida_TIME(T_ultimo_disparo2: in out TIME) do
      if T_ultimo_disparo < T_ultimo_disparo2 then
        T_ultimo_disparo := T_ultimo_disparo2;
      end if;
      delay until T_ultimo_disparo + D;
      T_ultimo_disparo  := T_ultimo_disparo + D;
      T_ultimo_disparo2 := T_ultimo_disparo;
    end Tr_Compartida_TIME;
```

El caso de que una transición desmarque algún lugar de comunicación (figura 5.7.b), el proceso debe esperar a que la marca del lugar esté disponible. Por ello la operación `desmarcar` del lugar de comunicación deberá modificar el valor de la variable `T_ultimo_disparo` del proceso en función del instante en el que se marcó el lugar. Por ello la implementación del lugar de comunicación debe cambiar:

```
protected Lugar_Comunicacion2 is
  entry Esta_Marcado;
  entry Esta_Desmarcado;
  entry Desmarcar;
  procedure Marcar;
private
  Marcado: integer := ...; -- Valor inicial de marcado
  T_Marcado: TIME;  -- Instante en el que se produce el marcado del lugar
end Lugar_Comunicacion2;

protected body Lugar_Comunicacion2 is
  entry Esta_Marcado(T_ultimo_disparo: in out TIME) when Marcado > 0 is
  begin
    if T_Marcado > T_ultimo_disparo then
      T_ultimo_disparo := T_Marcado;  -- actualiza valor en proceso que consulta
    end if;
  end;

  entry Esta_Desmarcado when Marcado = 0 is
  begin
    null;
  end;
```

```
  entry Desmarcar when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
  end;

  procedure Marcar(Instante: in TIME) is
  begin
    T_Marcado := Instante;
    Marcado := Marcado + 1;
  end;
end Lugar_Comunicacion2;
```

O en su versión simplificada:

```
protected Lugar_Comunicacion_Simple2 is
  entry Desmarcar;
  procedure Marcar;
private
  Marcado: integer := ...; -- Valor inicial de marcado
  T_Marcado: TIME;  -- Instante en el que se produce el marcado del lugar
end Lugar_Comunicacion_Simple2;
protected body Lugar_Comunicacion_Simple2 is
  entry Desmarcar(T_ultimo_disparo: in out TIME) when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
    if T_Marcado > T_ultimo_disparo then
      T_ultimo_disparo := T_Marcado;  -- actualiza valor en proceso que desmarca
    end if;
  end;
  procedure Marcar(Instante: in TIME) is
  begin
    T_Marcado := Instante;
    Marcado := Marcado + 1;
  end;
end Lugar_Comunicacion_Simple2;
```

5.4. Implementación software de conflictos internos

Se tratará en esta sección sobre la implementación software de una serie de transiciones en conflicto que pertenezcan a un mismo protoproceso. Dentro de un protoproceso los conflictos serán de libre elección o simples, sin concurrencia, pues los conflictos acoplados, con concurrencia, deben distribuirse, lo que impide que aparezcan en el interior de un protoproceso. Por tanto, y teniendo en cuenta las restricciones de modelado, la situación más general que deberá ser implementada se compondrá de un lugar común del cual descenderán varias transiciones SYCO, varias TIME y un máximo de una transición CODE. El lugar común pertenecerá al elemento de partición utilizado para la generación del protoproceso, ya que si no perteneciese, no todas las transiciones del conflicto estarían dentro del protoproceso, que es lo que estamos suponiendo. Las transiciones SYCO podrán tener más lugares de entrada, ya sean privados o de comunicación asíncrona con otros procesos, e incluso alguna de estas transiciones puede estar compartida con otro proceso. Así mismo, las transiciones CODE y TIME también pueden tener varios lugares de entrada, aunque en esta

sección supondremos que todos son privados al proceso. Si no lo fueran, la forma de implementar el conflicto será distinta (lo comentaremos en la sección 5.7).[2]

Tal como se indicó en la sección 4.7.2, la política de resolución de conflictos (sección 2.5.1) debe ser implementada dentro del protoproceso. Por ello la implementación software tratará de disparar primero las transiciones SYCO sensibilizadas, ya que éstas representan acciones de control o sincronizaciones que deben ser ejecutadas inmediatamente. Para el disparo de la transición TIME o CODE se establecerá una política de carrera: si la ejecución de la transición CODE acaba antes de la expiración del evento temporal, se disparará la CODE. En caso contrario, el evento expira antes de la finalización del código, se disparará la transición TIME, abortando el código. Del mismo modo, si durante la ejecución de la transición CODE o TIME, alguna de las transiciones SYCO se sensibilizase como resultado de alguna comunicación asíncrona, la ejecución del código o evento temporal también sería abortada.

La estructura de implementación software utilizada cambiará radicalmente dependiendo de la existencia o no de transiciones CODE en el conflicto. Si las hay, se necesitará una estructura de control que permita abortar la ejecución del código de la transición debido a la expiración de un evento temporal (transición TIME) o alguna acción de control (que sensibilice alguna transición SYCO). En Ada 95, la estructura a utilizar es la A.T.C. (*Asynchronous Transfer of Control*). Sin embargo, si en el conflicto no existiesen transiciones CODE, la estructura a utilizar puede variar desde una simple estructura condicional con varias ramas, una para cada una de las transiciones, hasta una estructura `select` con alternativa `delay` o `else`, y/o varios `accept`, si las transiciones están involucradas en alguna comunicación, bien porque tienen lugares de entrada que son de comunicación con otras subredes, o porque alguna de las transiciones está compartida.

Para la exposición de la forma de implementación de conflictos internos a procesos procederemos a describir la implementación de estructuras sencillas, hasta llegar al caso general. En nuestra opinión, la complejidad de la estructura que aparece hace aconsejable esta forma de abordar el problema. Además, en primer lugar nos referiremos a conflictos en los que hay involucrada ejecución de código (transiciones CODE) y dejaremos para más adelante los conflictos sin código.

5.4.1. Conflictos internos con código

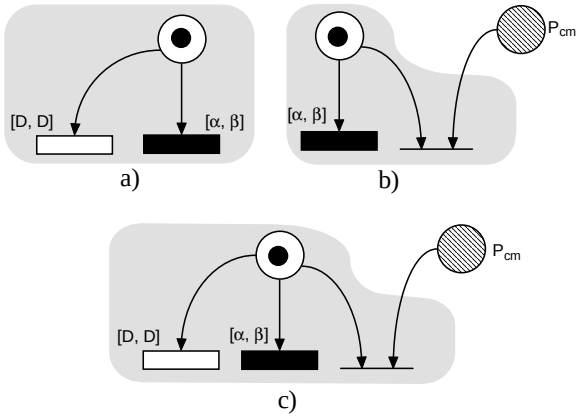


Figura 5.8: Varias configuraciones sencillas de conflictos internos con código

En primer lugar consideraremos para su estudio las configuraciones básicas de la figura 5.8:

[2]Cuando las transiciones CODE o TIME tiene varios lugares de entrada y no todos pertenecen al mismo protoproceso, la manera de implementar el conflicto es separando ambas transiciones e implementando el lugar común por separado

conflicto entre transición CODE y TIME (*timeout*), conflicto entre transición CODE y SYCO (código abortable asíncronamente), y mezcla de las anteriores.

a) Conflicto entre transición CODE y TIME

La configuración de la figura 5.8.a corresponde con el modelado de un *timeout* que actúa sobre la ejecución de un código. Si el plazo de la transición TIME vence antes de que termine el código, la transición TIME es disparada retirando la marca del lugar de entrada y, en consecuencia desensibilizando la transición CODE, lo que aborta la ejecución de su unidad de ejecución[3].

Proposición 5.4.1 (Implementación SW de conflicto entre transición CODE y TIME) *La implementación de la estructura de la figura 5.8.a corresponde con la siguiente primitiva Ada:*

```
select
  delay D;
then abort
  Codigo;
end select;
```

JUSTIFICACIÓN Partimos del modelo de estructura ATC con alternativa de disparo *delay* que se muestra en la figura 5.9.(1) (ver la descripción del modelo en el cap. 6, sección 6.4.1). En nuestro ámbito de aplicación, los valores de intervalo temporal de las transiciones TIME son siempre positivos. Por ello la transición $\mathsf{D} \leq 0$ del modelo carece de sentido y puede eliminarse. Una fusión serie de las transiciones ATC y $\mathsf{D} > 0$ nos conduce a la red de la figura 5.9.(2). Aplicando la regla de reducción de *lugares paralelos redundantes* (anexo C, regla C.9) los lugares delay en progreso y código 1 ejecuta puede ser reducidos a uno, con lo que obtenemos la red de la figura 5.9.(3), que corresponde con la configuración de conflicto que queríamos implementar. △

b) Conflicto entre transición CODE y SYCO

La configuración de la figura 5.8.b corresponde a la ejecución de un código que puede ser abortado asíncronamente por alguna acción de control externa, modelada por medio del marcado del lugar de comunicación p_{cm} que se encuentra a la entrada de la transición SYCO.

Proposición 5.4.2 (Implementación SW de conflicto entre transición CODE y SYCO) *La implementación de la estructura de la figura 5.8.b corresponde con la siguiente primitiva Ada:*

```
select
  Lugar_Comunicacion.Desmarcar;
then abort
  Codigo;
end select;
```

JUSTIFICACIÓN Para la demostración se debe partir de un modelo de estructura ATC con una llamada a un entrada de objeto protegido como alternativa de disparo. El modelo se muestra en la figura 6.22. Sin embargo, las características de la implementación permiten simplificar este modelo de partida. Debemos considerar lo siguiente:

[3]No es materia de estudio en esta tesis el estudio de técnicas de tolerancia y recuperación de fallos a aplicar en el caso de que la ejecución de una unidad de ejecución sea abortada.

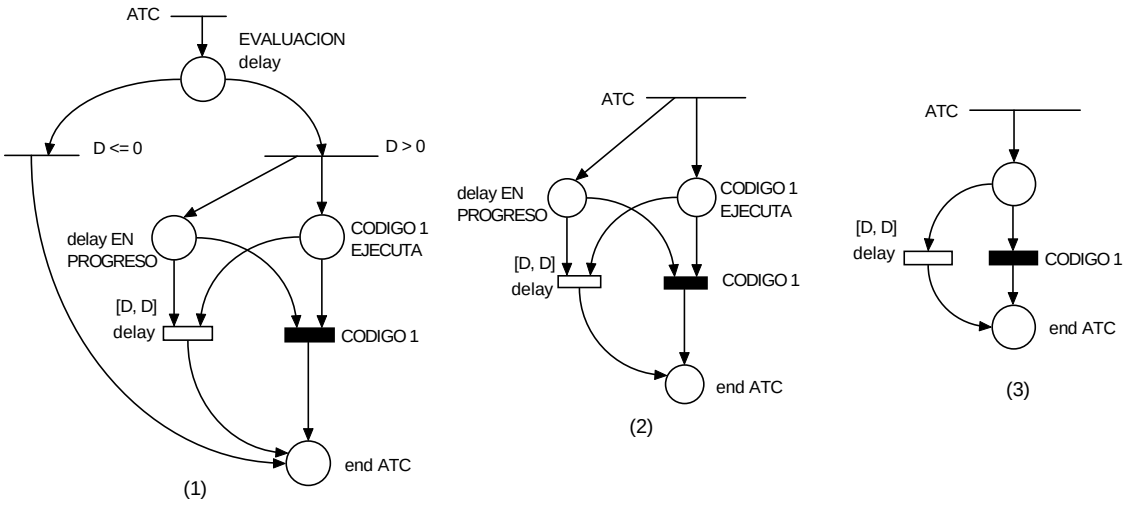


Figura 5.9: Justificación de la implementación de la estructura transición CODE + TIME en conflicto

- Utilizamos plataformas de implementación monoprocesador y prioridades estáticas.
- La prioridad del objeto protegido que implementa el lugar de comunicación es mayor que la de cualquier proceso involucrado, por lo que la ejecución de cualquiera de sus entradas suspenderá al resto de procesos.
- Si el lugar está marcado al comenzar la ejecución del conflicto, el código se abortará inmediatamente (de hecho, ni siquiera comenzará).
- Si el lugar está desmarcado, la llamada a desmarcar será encolada en el punto de entrada `desmarcar` del lugar de comunicación (ver proposición 5.3.1).
- Si el lugar está desmarcado, y la prioridad del proceso que marca el lugar es menor que la del proceso que contiene el conflicto, nunca se podrá abortar la ejecución del código, ya que no se podrá acceder a la CPU para marcar el lugar.
- Si el lugar está desmarcado, y la prioridad del proceso que lo marca es mayor que la del proceso que contiene el conflicto, el código del mismo puede ser abortado. En efecto, cuando el proceso que marca el lugar llama a la entrada `marcar` del lugar de comunicación, esta entrada es inmediatamente aceptada (el objeto protegido que implementa el lugar es más prioritario). Posteriormente, en la misma acción protegida, es atendida la llamada encolada en la entrada `desmarcar`, lo que provoca que el código de la estructura ATC sea abortado. Además la ejecución del código es suspendida desde el comienzo de la llamada a marcar, ya que hemos supuesto que el proceso que contiene el conflicto es menos prioritario (por lo que el código no puede terminar mientras el lugar es marcado y desmarcado).
- Si el código acaba es debido a que el lugar de comunicación no se ha marcado, ni se ha iniciado el marcado del mismo, ya que, de otra manera, alguna de las situaciones anteriormente descritas hubiera sido cierta, y el código se habría abortado. Por ello el lock de lectura/escritura del objeto protegido debe estar libre, por lo que se puede desencolar la llamada a `desmarcar` sin ninguna espera.

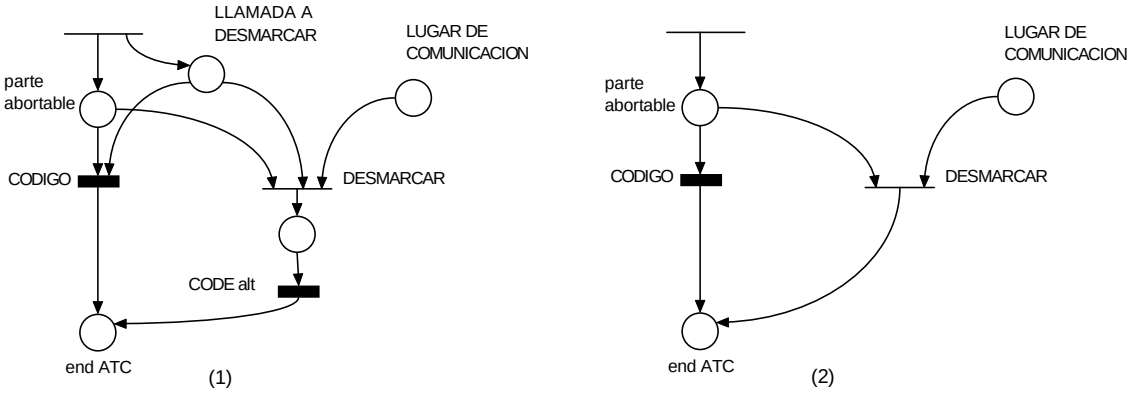


Figura 5.10: Justificación de la implementación de la estructura transición CODE + SYCO en conflicto

Estas consideraciones nos permiten hacer el modelo simplificado de la estructura propuesta, que se muestra en la figura 5.10.(1). El lugar llamada a desmarcar representa la permanencia en la cola de la llamada a la entrada `desmarcar` del lugar de comunicación que aparece en la alternativa de disparo de la ATC propuesta.

Al aplicar la regla *lugares paralelos redundantes* (anexo C, regla C.9) conseguimos la configuración de conflicto que queremos implementar (figura 5.10.(2)). △

La alternativa de disparo también puede corresponder a una cita con otro proceso, que implemente una transición o subred compartida:

```
select
  Proceso2.Cita;
then abort
  Codigo;
end select;
```

En este caso, la proposición 5.4.2 es igualmente válida. Si el segundo proceso tiene una prioridad inferior a la del que contiene el conflicto, la ejecución del código nunca podrá ser abortada. En caso contrario la ejecución del código será suspendida en el momento en que comience la cita, y tras ésta, abortada definitivamente.

c) Conflicto entre transición CODE, TIME y SYCO

La configuración de la figura 5.8.c corresponde a la ejecución de un código que puede ser abortado por el vencimiento de un *timeout* o asíncronamente por alguna acción de control externa, cuya ocurrencia se representa mediante el marcado del lugar de comunicación p_{cm}.

Proposición 5.4.3 (Implementación SW de conflicto entre tr. CODE, TIME y SYCO) *La implementación de la estructura de la figura 5.8.c corresponde con la siguiente estructura Ada:*

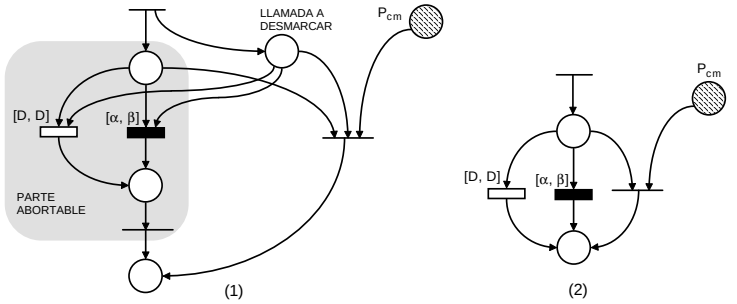


Figura 5.11: Justificación de la implementación de la estructura transición CODE + TIME + SYCO en conflicto

```
select
  Lugar_Comunicacion.Desmarcar;
then abort
  select
    delay D;
  then abort
    Codigo;
  end select;
end select;
```

JUSTIFICACIÓN Nos basaremos en las proposiciones anteriores. A partir de la red de la figura 5.10.(1), de la reducción de un ATC con llamada a punto de entrada (transiciones CODE + SYCO), sustituiremos el código de esta estructura por el modelo reducido de una estructura ATC con delay (paso (4) de la figura 5.9). De este modo obtenemos el modelo de la figura 5.11.(1). En el, el lugar llamada a desmarcar es redundante, por lo que puede ser eliminado. Una fusión de lugares en serie nos conduce al modelo de la figura 5.11.(2), que es la estructura que queremos implementar. △

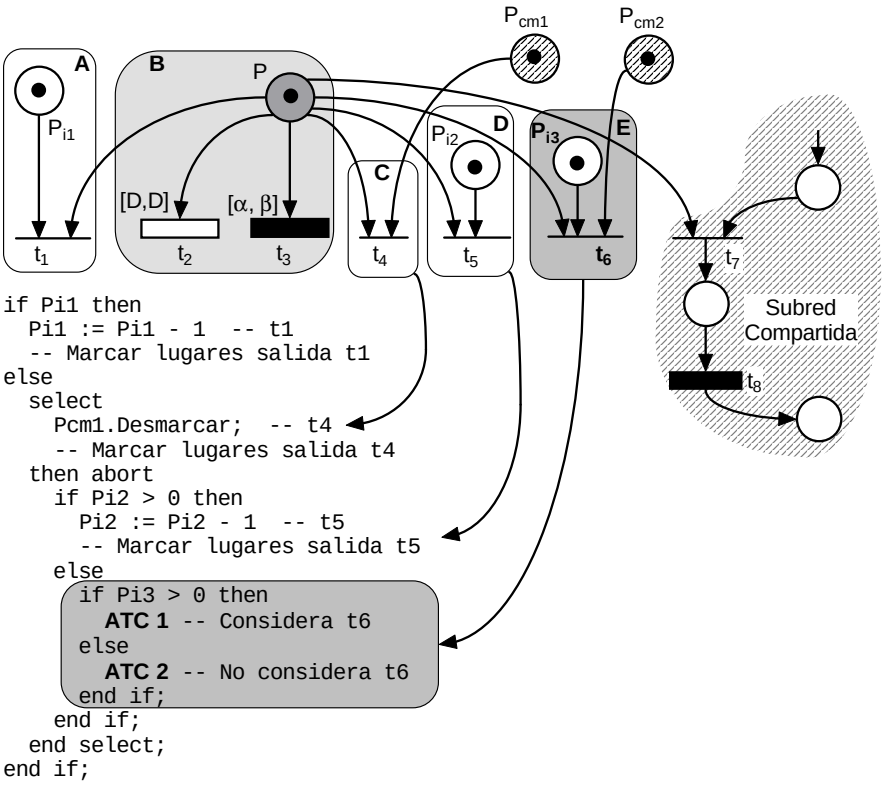
d) Caso general de conflicto interno con código

Consideraremos como caso general de conflicto interno el reflejado en la figura 5.12. En él aparecen transiciones de los tres tipos que comparten un único lugar de entrada (p). Con configuraciones como las de la figura se pueden modelar ejecuciones de código, abortables por expiración de un *timeout* o por alguna acción de control interna o externa. En la figura pueden diferenciarse varios elementos reseñables (supondremos que las prioridades de las transiciones están ordenadas de mayor a menor $\text{Prio}(t_1) > \text{Prio}(t_4) > \text{Prio}(t_5) > \text{Prio}(t_6) > \text{Prio}(t_7)$):

a. Acción de control interna, modelada con la transición SYCO t_1, que se activará si el lugar p_{i1} está marcado, evitando la ejecución del resto de transiciones en conflicto.

b. Ejecución de código, modelado mediante la transición CODE, que puede ser abortada por un *timeout*, modelado mediante la transición TIME.

c. Acción de control externa, modelada con la transición SYCO t_4, que abortará la ejecución del resto de las transiciones del conflicto cuando se detecte el marcado del lugar p_{cm1}, que es marcado como resultado de una comunicación asíncrona con alguna otra subred.




```
if Pi1 then
  Pi1 := Pi1 - 1  -- t1
  -- Marcar lugares salida t1
else
  select
    Pcm1.Desmarcar;  -- t4
    -- Marcar lugares salida t4
  then abort
    if Pi2 > 0 then
      Pi2 := Pi2 - 1  -- t5
      -- Marcar lugares salida t5
    else
      if Pi3 > 0 then
        ATC 1 -- Considera t6
      else
        ATC 2 -- No considera t6
      end if;
    end if;
  end select;
end if;
```

ATC 1

```
select
  Pcm2.Demark;      -- t6
  Pi3 = Pi3 - 1;
  -- Marcar lugares salida t6
then abort
  select
    Proceso2.Subred_Compartida;
    -- Marcar lugares salida t8
  then abort
    select
      delay D;
      -- Marcar lugares salida t2
    then abort
      Codigo;
      -- Marcar lugares salida t3
    end select;
  end select;
end select;
```

ATC 2

```
select
  Proceso2.Subred_Compartida;
  -- Marcar lugares salida t8
then abort
  select
    delay D;
    -- Marcar lugares salida t2
  then abort
    Codigo;
    -- Marcar lugares salida t3
  end select;
end select;
```

Figura 5.12: Ejemplo de conflicto interno general con código. En el conflicto de la figura se ha supuesto que las prioridades de las transiciones están ordenadas de mayor a menor $Prio(t_1) > Prio(t_4) > Prio(t_5) > Prio(t_6) > Prio(t_7)$.

d. Acción de control interna menos prioritaria que (a), modelada con la transición SYCO t_5, que se activará si el lugar p_{i2} está marcado.

e. Acción de control externa protegida con una condición, modelada con la transición SYCO t_6, que abortará la ejecución del resto de las transiciones del conflicto cuando se detecte el marcado del lugar p_{cm2} y si el lugar p_{i3} está también marcado.

f. Acción de control externa, activada mediante una cita con otro proceso, modelada mediante la subred compartida que comienza por la transición t_7.

Por inducción, a partir del caso anterior de conflicto entre transiciones CODE, TIME y SYCO, se puede deducir que la estructura de implementación para el caso general de conflicto interno con código será la ATC. En ella serán anidadas las distintas posibilidades de ejecución colocando en las alternativas de disparo las distintas transiciones que pueden abortar la ejecución del código, siendo la más exterior la más prioritaria. En definitiva, los distintos elementos que hemos distinguido en la figura serán implementados con las siguientes estructuras:

- La transición CODE y la transición TIME serán modeladas con una estructura ATC, en la que la parte abortable es el código y el alternativa de disparo el evento temporal. Esta estructura será colocada en la parte más interior de la implementación del conflicto, ya que son transiciones menos prioritarias que las SYCO.

- Las acciones de control internas, modeladas con transiciones SYCO conectadas a lugares privados, se implementarán por medio de estructuras condicionales tipo `if`, estableciéndose la condición en términos del marcado de los sus lugares de entrada internos. La colocación en la estructura `if` dependerá de la prioridad de estas transiciones, situándose al principio las más prioritarias.

- Las acciones de control externas, modeladas con transiciones SYCO conectados a lugares de comunicación o con subredes compartidas, se implementarán en forma de alternativa de disparo de la estructura ATC, que abortarán la ejecución del resto de las transiciones del conflicto cuando se detecte el marcado del lugar de comunicación o la llamada del procesos con el que se comparte la subred. La colocación de estas transiciones en la estructura ATC será tanto más interior conforme menor sea la prioridad de la transición. Eventualmente estas transiciones tendrán lugares de entrada privados que, en función de su marcado, harán que la transición sea o no considerada como alternativa de disparo. La imposibilidad de proteger con guardas las alternativas de disparo de una estructura ATC, hace necesario que en la implementación se consideren todas las posibles estructuras ATC que se generan contando con las transiciones con lugares internos y sin ellas (ver estructuras ATC 1 y ATC 2 en la figura 5.12).

- En la implementación de una subred compartida la colocación del `accept` de la cita se hará en el otro proceso, pues en la estructura ATC sólo puede haber llamadas.

En el caso concreto de la estructura de la figura 5.12 la implementación se ha realizado como sigue: la transición t_1 es implementada con una estructura `if` dependiente del marcado de su lugar de entrada. Si estuviese marcado, sería la transición a disparar (ya que es la más prioritaria). Si no está marcado se considera el resto de la estructura, en la que la transición t_4 está representando la aceptación de una comunicación asíncrona (lugar p_{cm1}). De estar disponible inmediatamente, se disparará, sin dar opción a que comience el resto de la estructura. En caso contrario, se considera la siguiente transición SYCO, que tiene lugares de entrada privados. Si el lugar privado p_{i2} está marcado se disparará t_5, abortando el resto de la estructura. Si no, consideraremos la siguiente transición, t_6, pero solamente en el caso de que el lugar privado p_{i3} esté marcado. Si lo está, t_6

puede abortar la ejecución del código y del evento temporal. Por último se considerará la comunicación síncrona que implementará la subred compartida. En el lugar más interno de la estructura se situará la ejecución del código de la transición CODE, que puede ser abortado por la expiración del evento temporal asociado a la transición TIME.

Dejaremos de momento la discusión de esta estructura de implementación, para retomarla en la sección 5.6, donde abordaremos el estudio de una anomalía que puede presentarse en ella.

e) Implementación del lugar de comunicación en conflictos internos con código

A la vista de las estructuras Ada 95 utilizadas para la implementación de los conflictos mostrados, se deduce que los lugares de comunicación deben ser llamados por el proceso que encierra el conflicto. Esto obliga a implementarlos dentro de algún objeto que permita aceptar llamadas, por lo que se elegirá la implementación basada en el objeto protegido que se muestra en la sección 5.3.1.

f) Actualización del marcado en la implementación de un conflicto con ATC

Reflexionemos sobre el siguiente código que representa la ejecución de un conflicto entre transiciones CODE y TIME. Supongamos que tras el disparo de cualquiera de las dos transiciones deben marcarse un gran número de lugares (situación de la figura 5.13.a). Para la implementación de dicha estructura utilizaremos una ATC como ésta:

```
select
  delay D; -- Intervalo de disparo de la transición TIME
  Marcar(lugar_T1);
  ...
  Marcar(lugar_Tn);
then abort
  Codigo;
  Marcar(lugar_C1);
  ...
  Marcar(lugar_Cm);
end select;
```

Tras la ejecución de la estructura uno, y sólo uno, de los dos conjuntos de lugares debe ser marcado, lo que ocurriría si estas operaciones fueran instantáneas. Sin embargo, con la implementación propuesta, esto no es así, pudiendo alcanzarse marcados intermedios que no son propios de la evolución de la red. Supongamos que una vez acabada la ejecución del código de la transición CODE, y comenzado el marcado de los lugares C_1 a C_m, pero antes de que todos sean marcados, se produce la activación de la alternativa de disparo. Esto provoca el marcado de los lugares T_1 a T_n, y aborta el proceso de marcado anteriormente iniciado y no terminado. El resultado es un marcado incorrecto.

El problema es originado por el marcado de los lugares de la parte abortable[4], que no es instantáneo. Para evitar esta situación evitaremos incluir el parte abortable otras instrucciones que no sean las propias de la implementación de la transición CODE. Se registrará de algún modo qué transición ha sido la disparada, y tras el ATC se realizará la actualización del marcado. Una posible implementación puede ser:

```
TIME_disparada : boolean;
...
```

[4]Nótese que el problema está en el marcado de lugares en la parte abortable. Por tanto, si una estructura ATC no marca ningún lugar en su parte abortable, no es necesario recurrir a esta técnica.

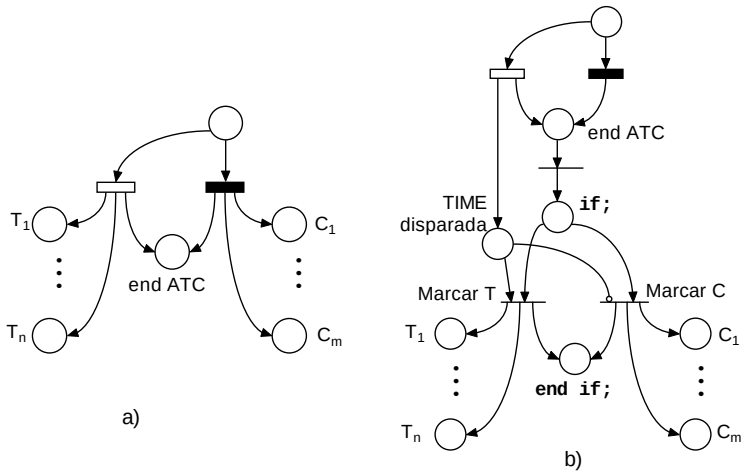


Figura 5.13: Marcado simultáneo de lugares tras un conflicto interno de transiciones

```
TIME_disparada := FALSE;
select
  delay D;
  TIME_disparada := TRUE; --  el disparo de la transición TIME
then abort
  Codigo;                 -- Ninguna instrucción más
end select;

if TIME_disparada then
  Marcar(lugar_T1);
  ...
  Marcar(lugar_Tn);
else                          -- Transición CODE
  Marcar(lugar_C1);
  ...
  Marcar(lugar_Cm);
end if;
```

Hemos de justificar que la transformación realizada tiene el mismo comportamiento que la estructura que se trata de implementar. Un modelo de la misma se puede observar en la figura 5.13.b. Puede deducirse fácilmente que desde el punto de vista del marcado de los lugares T_1 a T_n y C_1 a C_m, ambas redes se comportan igual, por lo que serán equivalentes. En efecto, en el grafo de estados de la red se comprueba que los lugares T_1 a T_n se marcan si se dispara la transición Marcar T, lo que ocurre únicamente si se dispara la transición TIME, que marca TIME disparada. Análogamente los lugares C_1 a C_m se marcan si se dispara la transición CODE, que no marca el lugar TIME disparada.

Si la estructura ATC tuviera más ramas, correspondientes a más transiciones en conflicto, la estructura debe ser completada con más variables booleanas y más ramas en la estructura `if`. Más adelante (ver, p.ej. la sección 5.6.3), se advertirá sobre la posibilidad de que varias opciones de una estructura ATC anidada se completen simultáneamente, lo que en este caso supondría que varias de esas variables booleanas serían ciertas. Si esto ocurriese, es posible protegerse contra este

comportamiento si en el esquema anterior se priorizan las ramas de la estructura condicional, que deben ordenarse, colocando antes las correspondientes a transiciones más prioritarias.

5.4.2. Conflictos internos sin código

En este tipo de conflictos no será necesario utilizar una estructura ATC ya que no hay ejecución de código que abortar. Es necesario distinguir entre dos situaciones: a) ninguna de las transiciones del conflicto está involucrada en comunicaciones con otras partes de la red, o b) alguna de ellas supone comunicación (asíncrona, con lugares de comunicación, o síncrona con transiciones o subredes compartidas). En cualquier caso consideraremos que las transiciones TIME están excluidas de cualquier comunicación (este tema será tratado más adelante en la sección 5.7).

a) Conflictos internos sin código y sin comunicaciones

En este caso, se dispondrá de una serie de transiciones exclusivamente con lugares de entrada privados. Cuando el flujo de control alcance la situación de conflicto bastará con evaluar las condiciones de sensibilización de todas las transiciones en conflicto. Esto se realizará de la forma descrita 5.3.2, mediante estructuras condicionales tipo `if`. En este caso, como son varias las transiciones, varias serán las ramas de la estructura `if`. El test de sensibilización de cada transición (estado de sus lugares de entrada) será la condición de cada una de estas ramas.

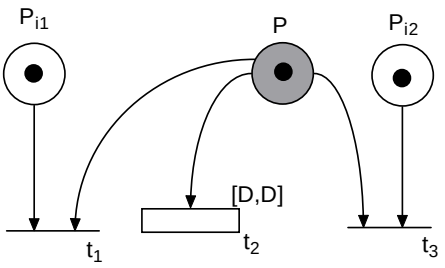


Figura 5.14: Ejemplo de conflicto interno sin código y sin comunicaciones

La posición de cada transición en la estructura anidada dependerá de su prioridad. Las transiciones SYCO se ordenarán de posiciones exteriores a más interiores según decrezca su prioridad. En cualquier caso, como el disparo de una transición SYCO debe ser más prioritario que el de una TIME, estas transiciones ocuparán el lugar más interno de la estructura, ordenándolas, si hay varias, según el valor de su intervalo temporal. Por ejemplo, la implementación de la estructura de la figura 5.14 será (suponemos que la prioridad de la transición t_1 es mayor que la de t_3):

```
if Pi1 > 0 then                    -- Disparo de t1
  Pi1 := Pi1 - 1;
  -- Marcado de los lugares de salida de t1
elsif Pi2 > 0 then                 -- Disparo de t3
  Pi2 := Pi2 - 1;
  -- Marcado de los lugares de salida de t3
else                               -- Disparo de TIME
  delay D;
  -- Marcado de los lugares de salida de t2
end if;
```

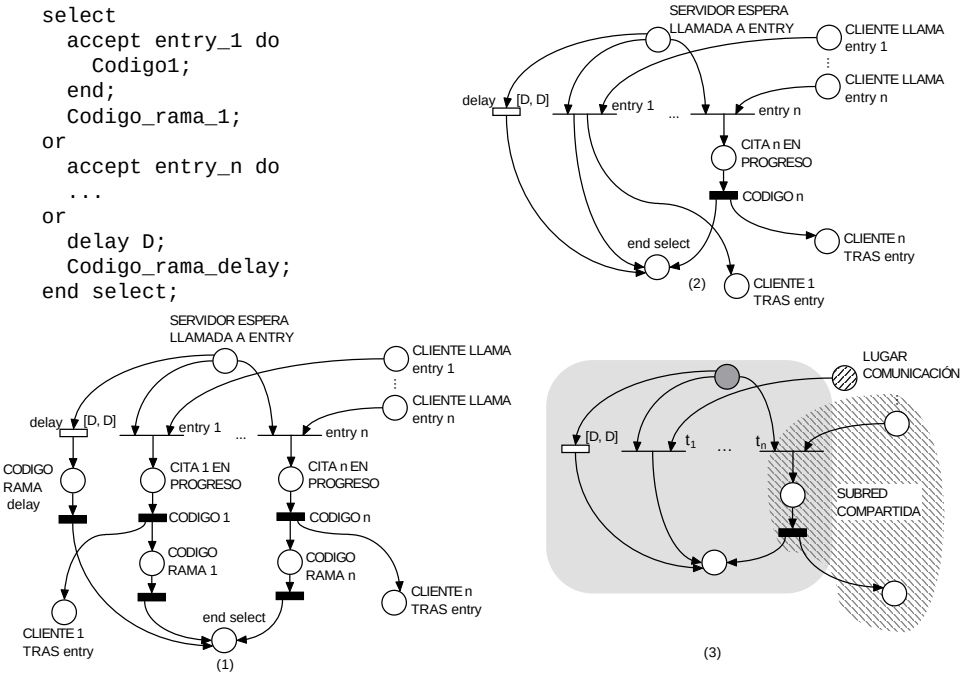


Figura 5.15: Justificación de la implementación de un conflicto interno sin código y con comunicaciones y transición TIME

b) Conflictos internos sin código y con comunicaciones

La estructura que nos ocupa responde a la situación en la que es posible recibir distintas comunicaciones que provienen de distintos lugares de comunicación, o de transiciones o subredes compartidas. Limitaremos las posibilidades de comunicaciones, restringiéndolas a las producidas a través de transiciones SYCO (las que se producen a través de transiciones TIME se implementan de otra manera, consultar la sección 5.7). En esta situación es preciso utilizar una estructura de programación que permita aceptar comunicaciones de distintos orígenes. Por ello emplearemos la estructura `select` con varios `accept`, uno por cada comunicación, alternativas `delay` si el conflicto tiene transiciones TIME, o la alternativa `else`.

Proposición 5.4.4 (Implementación SW de conflicto interno sin código 1) *La implementación de un conflicto simple sin código, y con comunicaciones y transición TIME corresponde con la siguiente estructura Ada 95:*

```
select
  accept t1;
...
or
  accept tn do ...
or
  delay D;
end select;
```

```
select
  accept entry_1 do
    Codigo1;
  end;
  Codigo_rama_1;
or
  accept entry_n do
  ...
else
  Codigo_rama_else;
end select;
```

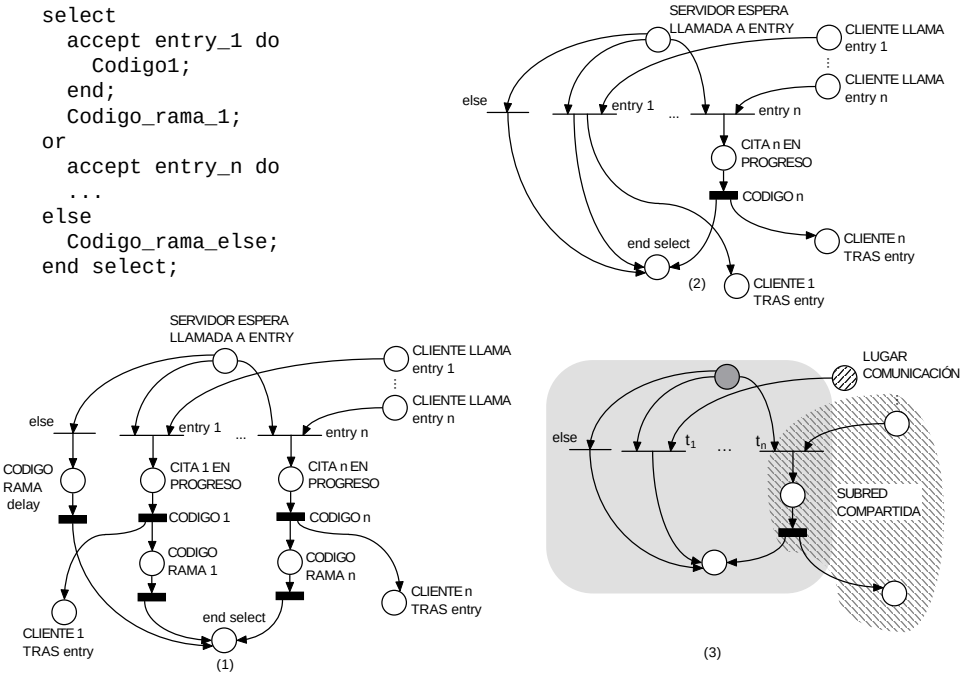


Figura 5.16: Justificación de la implementación de un conflicto interno sin código y con comunicaciones y sin transiciones TIME

JUSTIFICACIÓN Partimos del modelo de estructura `select` con *delay* que se muestra en la figura 5.15.(1) (una descripción del modelo se proporciona en el capítulo 6, sección 6.3.7, figura 6.10.a).

Si consideramos que en las citas que se realizan en el `select` no se ejecuta ningún código, pues representan el desmarcado de lugares de comunicación, y que tras las mismas no hay ningún código alternativo, el modelo puede ser simplificado mediante la eliminación de los lugares cita i en progreso y código rama i y transiciones CODE correspondientes. De esta forma se llega al modelo de la figura 5.15.(2), en la que una de las citas no ha sido reducida, pues pretende representar una subred compartida. Si consideramos que el lugar que representa la comunicación asíncrona es modelado como una tarea mensajero, y que en la proposición 5.3.2 se demostró que para el modelo de lugar de comunicación activo, la tarea-lugar podía reducirse a un único lugar, el lugar cliente tras entry se puede eliminar, de modo que se obtiene directamente el modelo de la estructura que queremos implementar, mostrado en la fig. 5.15.(3)), e integrado, en este caso, por una comunicación asíncrona a través de lugar de comunicación, y otra síncrona mediante una subred compartida. △

Proposición 5.4.5 (Implementación SW de conflicto interno sin código 2) *La implementación de un conflicto simple sin código, y con comunicaciones y sin transición TIME corresponde con la siguiente estructura Ada 95:*

```
select
  accept t1;
...
or
  accept tn do ...
else
```

```
    null;
  end select;
```

JUSTIFICACIÓN La demostración es similar a la del caso anterior, pero partiendo del modelo de estructura `select` con `else` que se muestra en el capítulo 6, sección 6.3.7, figura 6.10.b. △

Como no está permitido que una estructura `select` tenga alternativa `else` y `delay` al mismo tiempo, la transición SYCO sin ningún otro lugar de entrada anulará la implementación de la transición TIME, si ésta existe.

La presencia de lugares privados conectados a transiciones SYCO involucradas en comunicaciones se implementará mediante guardas asociadas a los `accept` de la estructura. Si hay transiciones SYCO conectadas únicamente a lugares privados las implementaremos como en el caso anterior de conflictos sin comunicaciones. Esto puede deducirse fácilmente si se consulta en el modelo de la estructura `select` con guardas que se muestra en el capítulo 6, sección 6.3.8, figura 6.11.

La priorización del disparo de las transiciones, si varias están sensibilizadas, se puede resolver de dos maneras:

- Atendiendo a la prioridad del proceso o lugar de comunicación involucrado en la comunicación. Para ello habrá que especificar una política de colas de puntos de entrada de tipo `Priority_Queuing`. Si los procesos llamadores son los lugares de comunicación asíncrona, es necesario que la prioridad de los lugares esté ordenada congruentemente con el orden de las transiciones SYCO. En el caso de que varios tengan la misma prioridad se seleccionará aquella entrada que aparezca primero en la enumeración de los `accept` del `select`, por lo que es preciso ordenar los `accept` de forma adecuada.

- Se puede recurrir [BW98] a poner guardas en los `accept` referentes al atributo `'COUNT` del resto de entradas más prioritarias. Sólo cuando el valor `'COUNT` de las más prioritarias sea cero, se podrá proceder a la aceptación de comunicaciones por un `accept` menos prioritario.

Una configuración de este tipo de conflictos puede contemplarse en la figura 5.17. En ella aparecen acciones de control internas (transición t_1), comunicaciones asíncronas por medio de los lugares p_{cm1} (guardada con un lugar interno p_{i2}) y p_{cm2}, y comunicaciones síncronas (subred compartida), todas ellas con un *timeout* asociado (transición TIME, t_2). Supondremos que las prioridades de las transiciones SYCO están ordenadas, siendo la mayor la de t_1 y la menor la de t_5. La transición TIME, según nuestra interpretación del modelo, tendrá la menor prioridad.

La implementación del conflicto de la figura será:

```
if Pi1 > 0 then
  -- Disparo de t1
  Pi1 := Pi1 - 1;
  -- Marcado de lugares de salida de t1
else
  select
    when Pi2 > 0 =>
      accept Desmarcar_Pcm1;  -- Disparo de t3
      Pi2 := Pi2 - 1;
      -- Marcado de lugares de salida de t3
  or
    accept Desmarcar_Pcm2;    -- Disparo de t4
    -- Marcado de lugares de salida de t4
  or
```

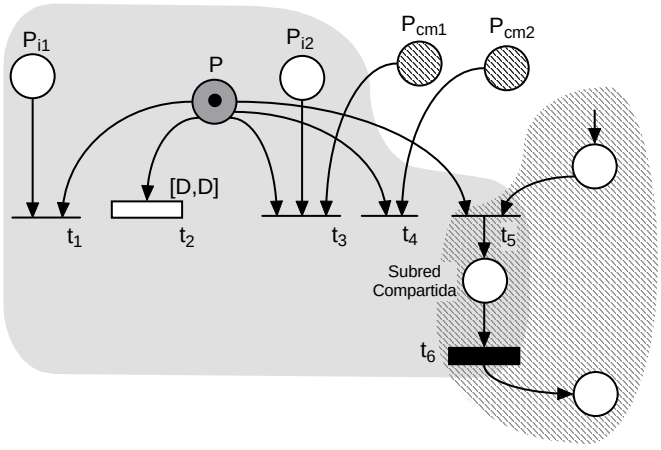


Figura 5.17: Ejemplo de conflicto interno sin código y con comunicaciones

```
    accept t5 do
      -- Implementación de la subred compartida
    end;
    -- Marcado de lugares de salida de t6 pertenecientes al proceso
  or
    delay D;
  end select;
end if;
```

c) Implementación del lugar de comunicación en conflictos internos sin código

Debido a la arquitectura de implementación (uso de `select`), los lugares de comunicación deberán llamar al proceso que contiene el conflicto. Por tanto deben ser implementados como tareas activas, según la versión activa propuesta en la sección 5.3.1. Esta implementación era válida únicamente para lugares binarios, por lo que si el lugar de comunicación no cumple este requisito, la estructura no podrá ser implementada de esta manera. En la sección 5.10 trataremos sobre cómo abordar la implementación cuando esto suceda.

Así mismo, si existiese alguna transición o subred compartida en el conflicto, el `accept` de la cita que la implemente deberá residir en la estructura `select` que implementa el conflicto, y la llamada, en el otro proceso.

Ejemplo

La red de la figura 5.18 ha sido descompuesta mediante la técnica descrita en el capítulo anterior en tres protoprocesos que darán lugar a tres tareas Ada. Los lugares de comunicación se muestran rayados. Se pueden apreciar distintas configuraciones de conflictos internos, con y sin código, que pueden ser implementados de la siguiente manera:

```
pragma Queuing_Policy(Priority_Queuing);
...
task Proc_1 is
  entry Desmarca_S1;
  pragma Priority(ALTA);
```

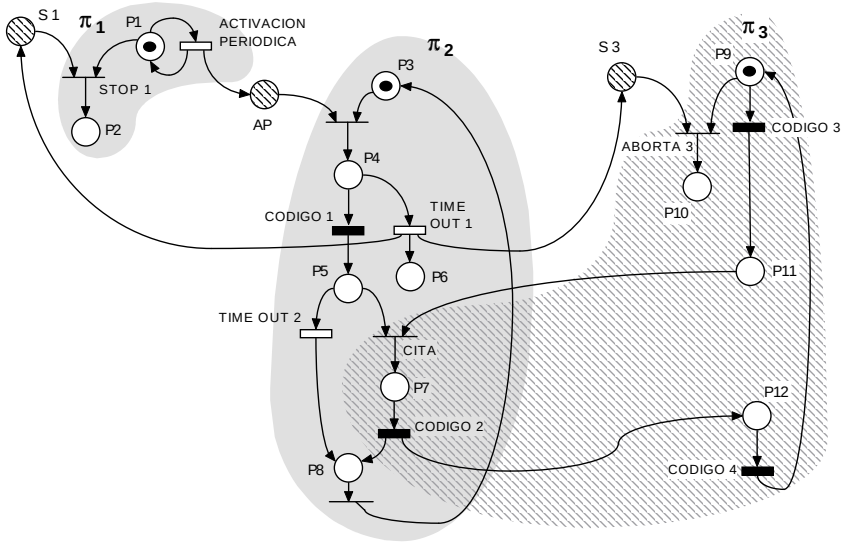


Figura 5.18: Red de ejemplo con conflictos internos para su implementación software. Los lugares rayados son lugares de comunicación

```
end Proc_1;

task body Proc_1 is
  T_ultimo_disparo: Time := CLOCK;
  ACTIVACION_PERIODICA: constant duration := ...;
begin
  loop
   select               -- Conflicto interno sin código con comunicación
    accept Desmarca_S1;
    exit;
   or
    delay until T_ultimo_disparo + ACTIVACION_PERIODICA;
    T_ultimo_disparo := T_ultimo_disparo + ACTIVACION_PERIODICA;
    AP.Marcar;
   end select;
  end loop;
end Proc_1;

task Proc_2 is
  entry Cita;
  pragma Priority(BAJA);
end Proc_2;

task body Proc_2 is
  Time_out_1: constant duration := ...;
  Time_out_2: constant duration := ...;
begin

  loop
```

```
    AP.Desmarcar;
    select                -- Conflicto interno con código CODE + TIME)
      delay Time_out_1;
      S1.Marcar;
      S3.Marcar;
      exit;
    then abort
      Codigo_1;
    end select;
    select                -- Conflicto interno sin código con comunicación
      accept Cita do
        Codigo_2;
      end Cita;
    or
      delay Time_out_2;
    end select;
  end loop;
end Proc_2;

task Proc_3 is
  pragma Priority(BAJA);
end Proc_3;

task body Proc_3 is
begin
  loop
    select                -- Conflicto interno con código CODE + SYCO)
      S3.Demarcar;
      exit;
    then abort
      Codigo_3;
    end select;
    Proc_2.Cita;
    Codigo_4;
  end loop;
end Proc_3;

task S1 is                -- Lugar de comunicación tipo mensajero
  entry Marcar;
  pragma Priority(ALTA);
end S1;
task body S1 is
begin
  accept Marcar;
  Proc_1.Desmarca_S1;
end S1;

protected S3 is           -- Lugar de comunicación tipo objeto protegido
  entry Desmarcar;
  procedure Marcar;
  pragma Priority(ALTA);
```

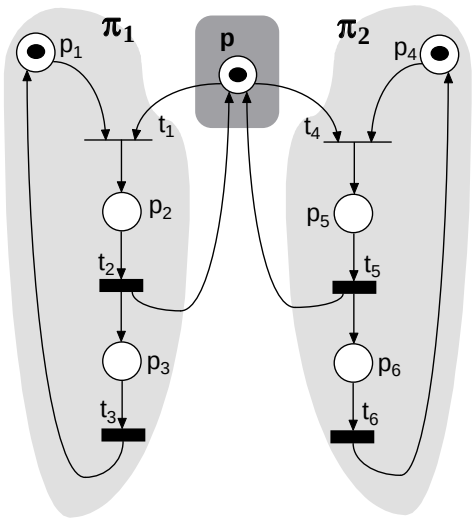


Figura 5.19: Red de ejemplo con conflicto simple para su implementación software

```
private
  Marcado: integer := 0;
end S3;

protected body S3 is
  entry Desmarcar when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
  end;
  procedure Marcar is
  begin
    Marcado := Marcado + 1;
  end;
end S3;

protected AP is  ...     -- Similar a S3
```

5.5. Implementación software de conflictos simples distribuidos

En la sección 4.7.2 se establecieron las bases para el estudio de la posibilidad de distribuir las transiciones de un conflicto simple (figura 5.19) entre varios protoprocesos. Entonces se indicó que los protoprocesos competirían por la marca del lugar común que generaba el conflicto y que la política de resolución debía residir en su interior.

Habiendo dejado la resolución del conflicto en manos del lugar común es preciso determinar qué objeto es válido para programar su implementación y como implantar en él una política adecuada para la resolución del conflicto. Como en otras secciones anteriores restringiremos los casos en estudio a conflictos simples cuyas transiciones descendentes sean de tipo SYCO. Más adelante, en la sección 5.7, analizaremos el caso en el que estas transiciones sean CODE y/o TIME.

5.5.1. Implementación software del lugar común en un conflicto simple

El lugar común que genera el conflicto es un lugar de comunicación de destino múltiple (varios protoprocesos pueden desmarcarlo y marcarlo). Por ello se puede implementar, tal como se indica en la sección 5.3.1, en forma de objeto protegido. Además, al haber supuesto que todas sus transiciones descendentes son de tipo SYCO, podemos utilizar la versión simplificada:

```
pragma Queuing_Policy(Priority_Queuing);
protected Lugar_Comun is
  entry Desmarcar;
  procedure Marcar;
private
  Marcado: integer := 1;
end Lugar_Comun;
protected body Lugar_Comun is
  entry Desmarcar when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
  end;
  procedure Marcar is
  begin
    Marcado := Marcado + 1;
  end;
end Lugar_Comun;
```

Ejemplo

La red de la figura 5.19 muestra un conflicto entre transiciones SYCO que se ha distribuido, generando dos protoprocesos que darán lugar a dos tareas Ada 95. La implementación de los procesos de la red será:

```
task Proc_1;
task body Proc_1 is
begin
  loop
    Lugar_Comun.Desmarcar;
    CODE (t2);
    Lugar_Comun.Marcar;
    CODE(t3);
  end loop;
end;

task Proc_2;
task body Proc_2 is
begin
  loop
    Lugar_Comun.Desmarcar;
    CODE (t5);
    Lugar_Comun.Marcar;
    CODE(t6);
  end loop;
end;
```

5.5.2. Implementación de la política de resolución del conflicto

Los distintos procesos que compiten por la marca del lugar, competirán por acceder al interior del objeto protegido a través de su entrada `desmarcar`. En caso de varias solicitudes simultáneas, la política de resolución deberá aceptar la llamada del proceso más prioritario, para lo cual se deberá especificar una política de cola en el objeto protegido basada en prioridades (`pragma Queuing_Policy(Priority_Queuing)`).

Para que la implementación sea correcta, la prioridad de los procesos que implementen las transiciones SYCO en conflicto debe ser congruente con la prioridad especificada para éstas en el modelo de RdPT. Sin embargo, esto no tiene por qué ser siempre así. La prioridad determinada para los procesos en la fase de planificación (recordar la sección 5.2) puede no corresponderse con la necesaria en el momento de resolver el conflicto. Supongamos, por ejemplo, que el proceso Π_1 de la figura 5.19 tenga una prioridad mayor que el proceso Π_2, pero que la prioridad asociada a la transición t_1 sea menor que la asociada a t_4. Para solucionar este desajuste será necesario promocionar la prioridad del proceso para que tenga el nivel de prioridad adecuado. El cambio de prioridad está justificado, pues en el momento de resolver el conflicto, el proceso está realizando labores de control, por lo que su prioridad debe ser mayor que durante la ejecución de sus acciones (*cambio de modo*). Para elevar la prioridad se utilizará la primitiva `Set_Priority`, que deberá elevar la prioridad del proceso antes de que se inicie la llamada al lugar común (ARM D.5(10)).

```
Prioridad_antigua := Get_Priority;
Set_Priority(Nueva_prioridad);
Lugar_Comun.Desmarcar;
Set_Priority(Prioridad_antigua);
```

Otra forma de abordar la solución del problema de la incongruencia entre las prioridades de los procesos y las transiciones SYCO que estos implementan se tratará en la sección 5.6.2. Entonces se indicará como completar la barrera de la entrada `desmarcar` para priorizar la llamada en función de la transición que solicita desmarcar el lugar y no del proceso que realiza la llamada.

Así mismo, para evitar la aparición inversiones de prioridad no acotadas en la implementación, se debe imponer el protocolo de techo de prioridad, en concreto su implementación en Ada 95 (*protocolo de techo de prioridad inmediato* [BW97]), mediante el `pragma Locking_Policy(Ceiling_Locking)`.

5.6. Implementación software de conflictos acoplados

El núcleo de la implementación de un conflicto acoplado reside en la implementación de los sincronizadores locales reducidos obtenidos durante el proceso de partición (ver sección 4.7.3).

5.6.1. Implementación del sincronizador local mediante un objeto protegido

Al tratarse de una generalización del concepto de lugar común de los conflictos simples, se propone utilizar una implementación similar, basada en el uso de un objeto protegido, que encapsule el estado de los lugares del sincronizador local (o sincronizador local reducido). El objeto protegido deberá tener tantas entradas como transiciones de salida tengan los lugares del sincronizador local, además de presentar entradas para permitir el marcado de los lugares que este encierra. Como las operaciones de desmarcado podrán retirar marcas de varios lugares simultáneamente, el código de las entradas afectará al marcado de varios lugares. Así mismo, las entradas de desmarcar deberán estar protegidas por barreras que hagan referencia al estado de los lugares de entrada de la transición que solicita ser disparada.

De forma análoga al caso de los conflictos simples, para evitar inversiones de prioridad no acotadas se debe imponer el protocolo del techo de prioridad (`pragma Locking_Policy (Ceiling_Locking)`).

Para facilitar la comprensión de la arquitectura del sincronizador local presentemos como ejemplo la implementación de la red de la figura 5.20. Procediendo como se indica en la sección 4.7.3, se ha determinado que los lugares p_2 y p_3 forman un sincronizador local reducido que es accedido por los protoprocesos π_1, a través de sus transiciones t_1 y t_2, y π_2, por medio de su transición t_3. La transición t_1 desmarca p_2, y las transiciones t_2 y t_3 desmarcan p_2 y p_3. Los lugares son marcados por las transiciones t_4, t_5, t_8 y t_9.

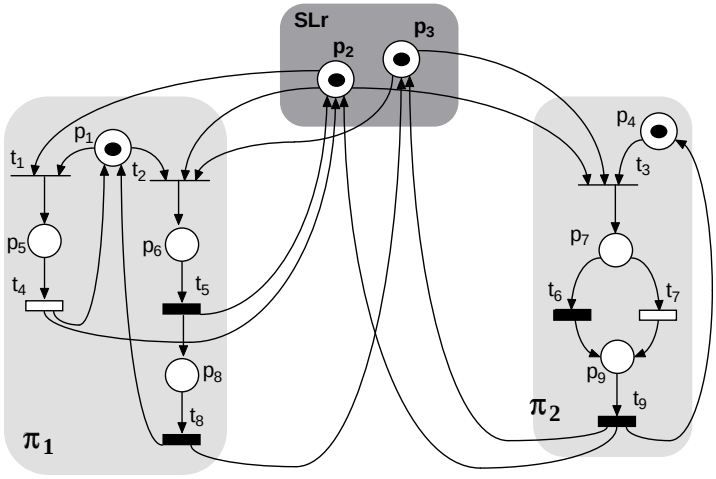


Figura 5.20: Red de ejemplo con conflicto acoplado para su implementación software

La implementación mediante un objeto protegido deberá proporcionar tres entradas para desmarcar lugares, una para cada transición de salida, y otras cuatro para marcar lugares, una para cada transición de entrada. Esto puede conseguirse mediante dos familias de entradas: `desmarcar` y `marcar`[5]. La primera debe estar protegida con barreras basadas en el estado de los lugares encapsulados.

```
pragma Queuing_Policy(Priority_Queuing);

type Transiciones_salida is (t1, t2, t3);
type Transiciones_entrada is (t4, t5, t8, t9);
type Lugares_SLr is (p2, p3);

protected Sincronizador_Local_Reducido is
  entry Desmarcar(Transiciones_salida);
  entry Marcar (Transiciones_entrada);
private
  Marcado: array (Lugares_SLr) of integer := (others => 1);
  -- estructura de lugares de entrada de las transiciones de salida
  -- estructura de lugares salida de las transiciones de entrada
end Sincronizador_Local_Reducido;
```

[5]A pesar de que la operación de marcar tiene características de procedimiento protegido, la necesidad de implementar una familia de entradas hace que se deba codificar como `entry`. Para simular el comportamiento de un procedimiento, su barrera siempre será `TRUE`

```
protected body Sincronizador_Local_Reducido is
  function Test_Sensibilizacion(t: Transiciones_salida) return boolean is
  begin
    -- devuelve TRUE si todos los lugares de entrada de t encerrados
    -- en el sincronizador están marcados
  end;

  entry Desmarcar(for t in Transiciones_salida) when Test_Sensibilizacion(t) is
  begin
    -- Decrementa el marcado de todos los lugares de entrada de t encerrados
    -- en el sincronizador
  end;

  entry Marcar(for t in Transiciones_entrada) when TRUE is
  begin
    -- Incrementa el marcado de todos los lugares de entrada de t encerrados
    -- en el sincronizador
  end;
end Sincronizador_Local_Reducido;
```

Los procesos que implementan las transiciones de salida del sincronizador local deberán realizar las llamadas a las entradas del mismo. Si el proceso implementa una única transición esto es sencillo (es el caso del proceso Π_2 que realizará la llamada `Sincronizador_Local_Reducido.Desmarcar(t3)`). Sin embargo, cuando un proceso implementa varias transiciones del conflicto, la codificación se complica, ya que el proceso debe priorizar el disparo de una de sus transiciones frente a las demás al mismo tiempo que lo intenta con todos a la vez. Para ello no sirve utilizar la estructura de llamada selectiva (`select ... else`), ya que si la primera alternativa no está disponible se pasará a esperar la segunda sin poder volver a la anterior. Es decir, la implementación en Π_1 de t_1 y t_2 no puede realizarse de la siguiente manera (supongamos que la prioridad de t_1 es menor que la de t_2:

```
  select
    Sincronizador_Local_Reducido.Desmarcar(t2);
    -- Marcado de lugares de salida de t2
  else
    Sincronizador_Local_Reducido.Desmarcar(t1);
    -- Marcado de lugares de salida de t1
  end select;
```

Es preciso intentar las dos llamadas a la vez, para lo cual se podría pensar en el uso de la estructura ATC. Sin embargo, un poco más adelante pondremos de manifiesto un problema que aparece en el empleo de esta estructura:

```
  select
    Sincronizador_Local_Reducido.Desmarcar(t2);
    transicion_disparada := 2;
  then abort
    Sincronizador_Local_Reducido.Desmarcar(t1);
    transicion_disparada := 1;
  end select;
  -- Actualizar marcado según transicion_disparada
```

5.6.2. Implementación de la política de resolución del conflicto

La priorización en la atención de las solicitudes de desmarcado de lugares de un sincronizador local se consigue mediante la política de colas adecuada (`Priority_Queuing`) y la ordenación de las llamadas al sincronizador en el interior de los procesos. Sin embargo, esta estrategia puede no ser suficiente para garantizar que el disparo de las transiciones SYCO descendentes del sincronizador se disparan en orden acorde a su prioridad. Analicémoslo en el siguiente ejemplo.

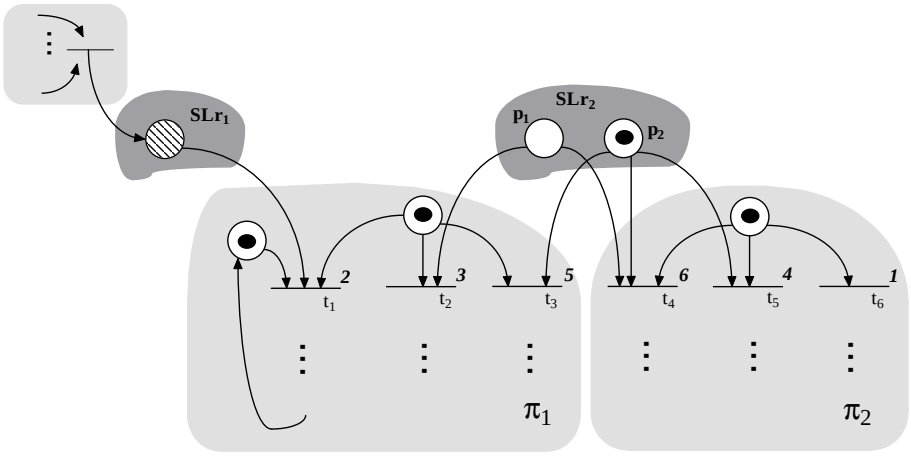


Figura 5.21: Ilustración de la anomalía en la priorización de la aceptación del disparo de transiciones en conflicto acoplado

La forma de priorizar la aceptación de una u otra entrada es atender a la prioridad de los procesos que implementan las transiciones del conflicto. Consideremos el caso de la figura 5.21. En ella se muestran las prioridades de las transiciones SYCO en conflicto. Para ser coherentes con esta asignación, la prioridad del proceso Π_2 debe ser mayor que la del Π_1, pues no en vano él contiene la transición más prioritaria (t_4). El estado de los lugares del sincronizador no permitirá el disparo de la transición t_4, por lo que se debería proceder al disparo de t_3. Sin embargo como la prioridad del proceso que contiene la transición t_3 es menor que la del que contiene t_5, se disparará esta última.

Para evitar esta situación se propone completar la expresión de la barrera asociada a cada entrada del sincronizador local para comprobar, además de la condición de sensibilización, el nivel de prioridad de la transición que hace la llamada. De esta forma, sólo se levantará la barrera de la entrada que corresponda a la transición sensibilizada con mayor prioridad de entre todas las encoladas. La barrera se expresará en términos parecidos a estos:

```
entry Desmarcar (for t in Transiciones_salida)
                when Test_Sensibilizacion(t) and then Maxima_Prioridad(t) is ...
```

siendo `Maxima_Prioridad(t)` una función que devuelve `TRUE` si la transición t es la transición sensibilizada más prioritaria de entre las encoladas. Una posible implementación de esta función es[6]:

```
function Maxima_Prioridad(t: in Transiciones_salida) return boolean is
  maxima: boolean := TRUE;
```

[6]La función podría simplificarse si en la definición del tipo `Transiciones_salida` las transiciones aparecen ordenadas de acuerdo a su prioridad.

```
begin
  for ti in Transiciones_salida loop
    if Desmarcar(ti)'COUNT /=0 and then Test_Sensibilizacion(ti)
       and then Prioridad(t) < Prioridad(ti) then
         maxima := FALSE;
         exit;
    end if;
  end loop;
  return maxima;
end Maxima_Prioridad;
```

5.6.3. Anomalía en la llamada múltiple de un proceso a uno o varios sincronizadores locales reducidos

Hemos comentado que la estructura que los procesos clientes de un sincronizador deben utilizar para realizar los intentos de desmarcado de los lugares de entrada de sus transiciones SYCO es la ATC. Las llamadas se ordenarán de forma que la más prioritaria sea la más externa. De este modo si ésta está disponible, no se intentarán las demás:

```
select
  SLr.T_más_prioritaria;
  transicion_disparada := 1;
then abort
  select
    SLr.T_intermedia;
    transicion_disparada := 2;
  then abort
    SLr.T_menos_prioritaria;
    transicion_disparada := 3;
  end select;
end select;
-- Actualizar marcado según transicion_disparada
```

La misma estructura aparecía en la implementación de conflictos internos con código (sección 5.4.1). Sin embargo, el comportamiento de la estructura ATC, junto con la arquitectura propuesta para el sincronizador no garantizan que se respete la semántica del conflicto, pues puede darse el caso de que varias llamadas de transiciones sean aceptadas a la vez, produciéndose marcados ficticios. Analicemos la siguiente situación, que a pesar de ser muy crítica, no es imposible: inicialmente el marcado de los lugares encerrados en el sincronizador no permite el disparo de la transición más prioritaria (situada en la alternativa de disparo de la ATC), aunque sí el de la siguiente, que comienza a ser disparada. Tras el disparo, pero antes de que termine la parte abortable (la instrucción que siguen a la llamada que registran qué transición ha sido disparada), se produce el marcado de los lugares que impedían el disparo de la más prioritaria, por lo que la llamada es aceptada (aún no ha terminado la parte abortable). A partir de este instante, ambas partes de la estructura ATC deben ejecutar hasta ser completadas, pues aunque la parte abortable termine, no puede abortar la alternativa de disparo ya que se trata de una cita, y ésta es una operación de aborto diferido. El resultado es que ambas transiciones son disparadas, es decir, sus lugares de entrada son desmarcados. El problema es incluso más grave si el proceso tiene transiciones SYCO que desmarcan distintos sincronizadores locales reducidos, como es el caso del proceso Π_2 de la figura 4.27, ya que entonces la ejecución de una de las llamadas puede ser interrumpida por la otra.

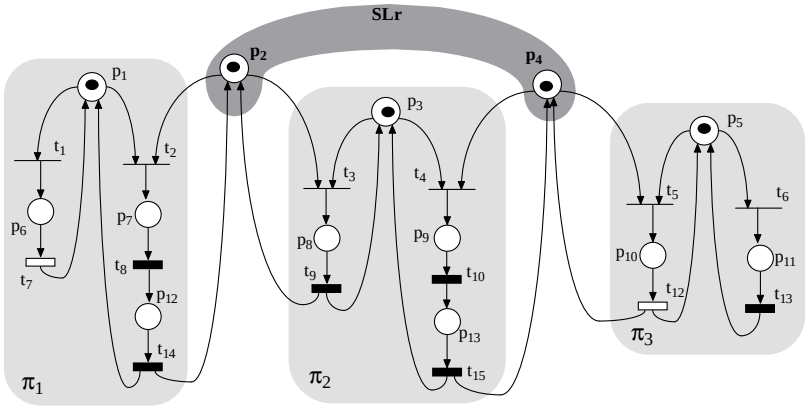


Figura 5.22: Sincronizadores locales reducidos fusionados para evitar la anomalía de la estructura ATC

Otra fuente de situaciones anómalas se produce debido a que la estructura encola en el sincronizador varias llamadas al mismo tiempo. Si una acción de marcado causase una apertura por reevaluación de todas las barreras de las llamadas encoladas, todas serían atendidas al acabar la acción de marcar, obteniéndose, de nuevo, marcados ficticios.

Para solucionar estas anomalías se propone modificar el sincronizador. En primer lugar hemos de agrupar en uno los distintos sincronizadores de los que puede ser cliente un proceso, lo que permitirá una gestión centralizada del protocolo que se expondrá seguidamente: se deberá bloquear el acceso del proceso al sincronizador local a partir del momento en que logre desmarcar los lugares de entrada de una de sus transiciones en el conflicto. Para ello se debe completar la expresión de la barrera de la familia de entradas `Desmarcar` e incluir, además del estado de los lugares, otra expresión que impida el acceso si el proceso que hace la llamada ya había accedido para desmarcar y todavía no ha abandonado la estructura ATC. Para implementar esta barrera es necesario que el sincronizador conozca en qué proceso está contenida cada una de las transiciones en conflicto. Así mismo, se debe añadir otra entrada para que los procesos indiquen que han finalizado su ATC. Basándonos en estas recomendaciones, la implementación del sincronizador de la figura 5.22 será:

```
pragma Queuing_Policy(Priority_Queuing);

type Transiciones_salida is (t2, t3, t4, t5);
type Transiciones_entrada is (t9, t12, t14, t15);
type Lugares_SLr is (p2, p4);
type Procesos is (Pi1, Pi2, Pi3);
protected Sincronizador_Local_Reducido is
  entry Desmarcar(Transiciones_salida);
  entry Marcar(Transiciones_entrada);
  entry Proceso_acaba(Procesos);
private
  Marcado: array (Lugares_SLr) of integer := (others => 1);
  Proceso_dentro: array (Procesos) of boolean := (others => FALSE);
  -- estructura de lugares de entrada de las transiciones de salida
  -- estructura de lugares salida de las transiciones de entrada
  -- estructura de procesos clientes del sincronizador y de las
     -- transiciones que implementan
end Sincronizador_Local_Reducido;
```

```
protected body Sincronizador_Local_Reducido is
  function Test_Sensibilizacion(t: Transiciones_salida) return boolean is ...

  function Proceso_ya_ha_accedido(t: Transiciones_salida) return boolean is ...
    -- devuelve TRUE si el proceso que contiene a t ya disparó alguna
    -- de sus transiciones en conflicto

  entry Desmarcar(for t in Transiciones_salida)
           when Test_Sensibilizacion(t) and not Proceso_ya_ha_accedido(t) is
  begin
    -- Actualiza marcado
    Proceso_dentro(Proceso_que_contiene_transicion(t)) := TRUE;
  end;

  entry Marcar(for t in Transiciones_entrada) when TRUE is ...
    -- Actualiza marcado

  entry Proceso_acaba(for p in Procesos) when TRUE is
  begin
    Proceso_dentro(p) := FALSE;
  end;
end Sincronizador_Local_Reducido;
```

Con esta implementación para el sincronizador local, la codificación correspondiente del proceso Π_2 será:

```
task body Proc_2 is
  P8, P9: boolean := FALSE;
begin
  loop
    select
      Sincronizador_Local_Reducido.Desmarcar(t3);
      P8 := TRUE;
    then abort
      Sincronizador_Local_Reducido.Desmarcar(t4);
      P9 := TRUE;
    end select;
    Proceso_acaba(Pi2);
    if P8 then
      Codigo(t9); P8 := FALSE;
      Sincronizador_Local_Reducido.Marcar(t9);
    elsif P9 then
      Codigo(t10); P9 := FALSE;
      Codigo(t15);
      Sincronizador_Local_Reducido.Marcar(t15);
    end if;
  end loop;
end Proc_2;
```

5.6.4. Implementación activa de sincronizador local

El anterior problema en el uso de la estructura ATC se debe a que más de una llamada es encolada en los puntos de entrada del sincronizador local. Al producirse una reevaluación de barreras varias de las llamadas son aceptadas en la misma acción protegida sin dar opción a evolucionar a la tarea llamadora. El problema desaparecería si la estructura ATC no se utilizase. En este punto se trata de hacer una implementación del proceso con transiciones SYCO en conflicto acoplado que no realice las llamadas al sincronizador, sino que las acepte. De este modo puede utilizarse una estructura `select` con varios `accept`. Considérese, por ejemplo el caso del proceso Π_2 de figura 5.22:

```
task body Proc_2 is
  P8, P9: boolean := FALSE;
begin
  loop
    select
      accept t3;
      P8 := TRUE;
    or
      accept t4;
      P9 := TRUE;
    end select;
    if P8 then
      Codigo(t9); P8 := FALSE;
      Sincronizador_Local_Reducido.Marcar(t9);
    elsif P9 then
      Codigo(t10); P9 := FALSE;
      Codigo(t15);
      Sincronizador_Local_Reducido.Marcar(t15);
    end if;
  end loop;
end Proc_2;
```

La priorización de la aceptación de las transiciones se producirá ordenando los `accept` en la estructura `select`.

Sin embargo, en este caso lo que se complica es la implementación del sincronizador local. Será él el encargado de hacer las llamadas a los puntos de entrada (transiciones) de los procesos. Para que esto sea posible el sincronizador debe conocer el estado todas las precondiciones (lugares de entrada) de las transiciones que descienden de él. Esto hace que sea necesario implementar todos los lugares de entrada del conflicto en el sincronizador local, incluidos los privados (de la forma indicada en la sección 4.7.3). Esto asegurará que cuando el sincronizador local haga una llamada, el proceso estará dispuesto a aceptarla. Es posible programar la política de resolución de conflictos más adecuada en el caso de que varias transiciones estén en conflicto efectivo. Como contrapartida, hacemos notar que se resiente la concurrencia de la implementación porque el sincronizador, responsable del disparo de transiciones concurrentes, es una tarea. Por lo tanto es secuencial y como consecuencia el disparo de las transiciones se secuencializa inevitablemente.

Para la implementación se puede utilizar un esquema parecido al coordinador de una implementación centralizada: primero se lanzan todas las llamadas al disparo de transiciones sensibilizadas y luego se aceptan marcados de lugares. El sincronizador será más prioritario que los procesos que dependen de él, ya que ejecuta de acciones de control (recuérdese que en las implementaciones centralizadas, el coordinador también era el proceso más prioritario). El orden de definición de las transiciones de salida puede aprovecharse para marcar un orden de prelación utilizable a la hora de

deshacer situaciones de conflicto efectivo entre transiciones. Por ejemplo, para la red de la figura 5.20[7].

```
type Transiciones_salida is (t1, t2, t3);
type Transiciones_entrada is (t4, t5, t8, t9);
type Lugares_SL is (p1, p2, p3, p4);
task body Sincronizador_Local is
  Marcado: array(Lugares_SL) of boolean := (others => TRUE);
begin
  loop
    for t in Transiciones_salida loop
      if Sensibilizada(t) then
        -- Llamada a la transición t del proceso adecuado
          -- Mediante el orden de declaración de Transiciones_salida se pueden
          -- priorizar los disparos de transiciones
        -- Desmarcar las lugares adecuados
      end if;
    end loop;
    loop
      accept Marcar (t: in Transiciones_entrada) do
        -- actualizar el marcado
      end
      exit when Marcar'COUNT = 0;
    end loop;
  end loop;
end Sincronizador_Local;
```

5.7. Influencia de los tipos de transiciones en la implementación software de un conflicto

Ante una situación de conflicto, y una hipotética distribución de las transiciones del mismo, es preciso considerar las características especiales de cada tipo de transición. A los razonamientos puramente estructurales o topológicos expuestos en anteriores apartados, hay que añadir consideraciones que surgen del significado temporal aportado por el modelo de redes de Petri con tiempo.

Según nuestra interpretación del modelo, una transición TIME o CODE puede ser disparada cuando, además de estar sensibilizada en la red de Petri subyacente (red sin interpretación temporal), la transición permanece sensibilizada durante un tiempo superior o igual al EFT (α_i) y un tiempo inferior o igual al LFT (β_i). Por tanto la presencia de una marca en un lugar no es sinónimo inmediato de que sus transiciones de salida puedan ser disparadas. La marca debe permanecer un tiempo determinado en el lugar. Es lo que se denomina tiempo de sensibilización de la transición. Si uno de los lugares de entrada deja de estar marcado durante el tiempo de sensibilización de alguna de las transiciones de salida, la transición no podrá ser disparada.

Este último aspecto es muy importante desde el punto el punto de vista de la implementación, ya que es preciso gestionar de alguna manera el tiempo de sensibilización. Hasta ahora el tiempo de sensibilización de una transición TIME o CODE estaba implícito en nuestra manera de implementar. Si se trata de una transición CODE el tiempo de sensibilización es el tiempo de ejecución del código asociado a la transición y si la transición es de tipo TIME será tiempo de suspensión del proceso que la contiene, por medio de un delay asociado. Una vez transcurrido el tiempo de

[7]Al retirar los lugares p_1 y p_4 de los procesos para incluirlos en el sincronizador local, los procesos quedan abiertos, por lo que deberán ser cerrados mediante la adición de implícitos

sensibilización, el disparo de la transición no hace sino indicar la terminación del código o la expiración del tiempo del evento temporal. Si alguno de los lugares de entrada es desensibilizado por el disparo de alguna otra transición, se abortará la ejecución del código o se suspenderá el evento.

Sin embargo, si las transiciones están en conflicto y se pretende distribuirlas entre varios procesos, lo que significa implementar el lugar de entrada común por separado, es preciso reflexionar sobre la gestión del tiempo de sensibilización de las transiciones de salida de ese lugar. Si una transición TIME o CODE se separa de alguno de sus lugares de entrada, será necesario mantener alguna estrategia para comprobar que el lugar común no deja de estar marcado. Analicemos tres opciones:

Opción 1.

Complicando la implementación del lugar común, se puede obligar a que, cuando es desmarcado, el lugar aborte las unidades de ejecución distintas de la disparada (de modo similar a las implementaciones centralizadas). Esto supone que el lugar debe tener registrado cuáles son sus transiciones de salida y cuales de ellas están actualmente en ejecución, lo cual no es trivial, pues no depende sólo del lugar común (pensar p.ej. en los conflictos simples). Además los procesos se deberían implementar de forma que permitan ser abortados y vueltos a activar, lo que obliga gestionar la activación y creación de tareas dinámicas.

Opción 2.

No separar las transiciones TIME y/o CODE de sus lugares de entrada. De esta forma, la gestión del tiempo de sensibilización puede mantenerse como hasta ahora: implícito en la implementación. Esta solución obliga a hacer privado a algún proceso el lugar común, por lo que es necesario implementar juntas todas las transiciones del conflicto (tal como se ha explicado en la sección 5.4). Sin embargo, en ciertas ocasiones esto no es posible en Ada 95. Consideremos las situaciones de la figura 5.23.

La única red fácilmente implementable es la de la figura 5.23.e. En ella se tiene un código (t_1) que puede ser abortado cuando se marca el lugar p_3. La implementación partirá la red en dos procesos: uno que se implementará mediante una estructura ATC que tenga el código en la parte abortable y como alternativa de disparo una llamada al otro proceso para ejecutar la cita que implementa la subred compartida $\{t_3, p_4, t_4\}$. Esta cita representa el disparo de la transición t_3, y abortará el código de t_1.

El problema se presenta cuando se tiene una transición tipo TIME con dos o más lugares de entrada (si son más de dos habrá que agruparlos en un sincronizador local). Es el caso de todas las demás redes de la figura. Se puede dar alguna interpretación a las redes de la figura 5.23: se puede interpretar que la figura (d) modela la ejecución de un código (t_1) que tiene un *timeout* asociado, pero el comienzo del *timeout* no es como en una situación normal, cuando comienza la ejecución del código, sino una vez comenzado éste, en cuanto se detecte el marcado de p_3. Es como si una vez comenzada una actividad y sin acabar alguien te dice: "Te queda 1 minuto para terminar". Esta situación no se puede implementar con ATC porque el tiempo del `delay` que se pone en la alternativa de disparo para abortar el código se refiere al instante del comienzo de la ejecución del código, y no se puede fijar una vez comenzado éste. La figura 5.23.c es equivalente a la 5.23.d, pero aquí lo que se aborta es un `delay` en vez de un código. En las figuras 5.23.a y 5.23.b la situación es distinta. Puede interpretarse que modelan (a) un proceso que adquiere inmediatamente un recurso si éste está libre y otro que, una vez detectado que está libre, espera un tiempo antes de adquirirlo, y (b) un par de procesos que esperan un tiempo antes de adquirir el recurso. Aquí el problema radica en que se debe abortar el delay asociado en cuanto se detecte que el lugar común es desmarcado por el otro proceso. Si se han de implementar las dos transiciones juntas, los lugares p_1 y p_4 serán usados como lugares de comunicación asíncrona. La estructura ATC no es válida debido a esta

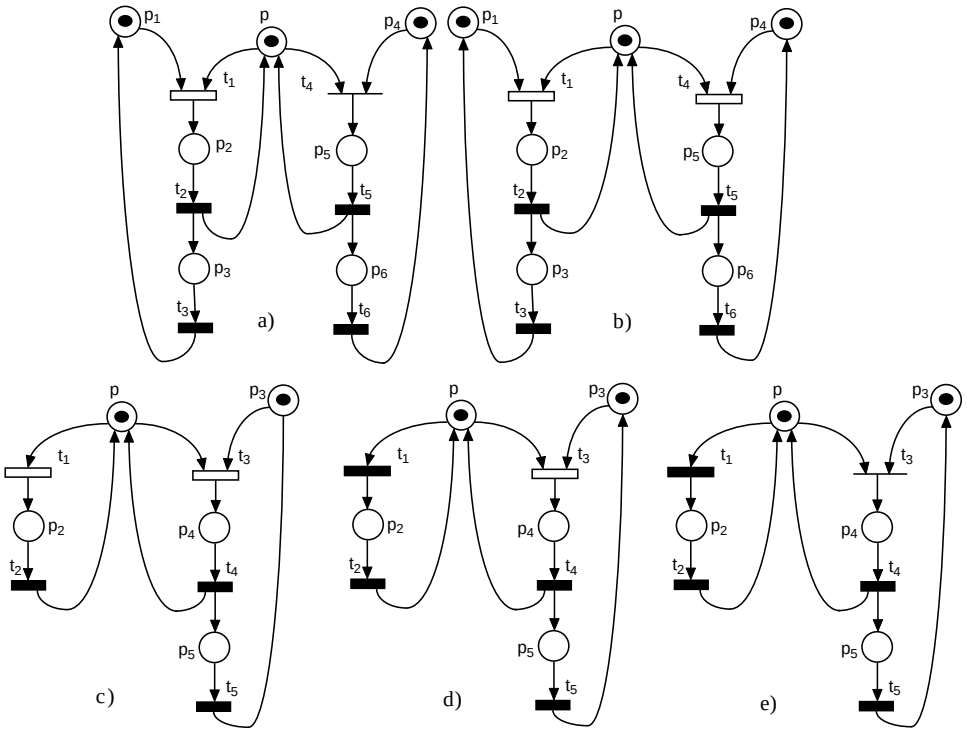


Figura 5.23: Distintos tipos de transiciones en conflicto

asincronía: si p está marcado no se puede empezar a descontar el tiempo de la transición t_1 (y t_4 en la (b)) hasta que p_1 no esté marcado (y p_4 en (b)). Pero este instante en el que estos lugares se marcan no se conoce a priori.

Opción 3. Forzar la distribución de las transiciones

La última de las opciones consiste en forzar la distribución de las transiciones del conflicto, aunque sea necesario añadir lugares implícitos para construir elementos de partición que cubran las transiciones del conflicto (como se comenta en el ejemplo 3 de la sección 4.7.2). El lugar común siempre será compartido entre los protoprocesos a los que de lugar la partición, y será de acceso común para ellos. Un ejemplo de aplicación para la figura 5.23.d se muestra en la figura 5.24.

Para poder implementar transiciones CODE o TIME que puedan ser abortadas porque su o sus lugares de entrada son desmarcados por otros procesos, hay que utilizar la implementación completa de lugar de comunicación que se dio en la sección 5.3.1 y que se reproduce un poco más adelante. Así mismo, la implementación de las transiciones TIME o CODE también debe completarse, para considerar la posibilidad de abortarlas en el momento en que se desmarque el lugar común. De esta forma, el código de la transición CODE (o el delay de la TIME) serán la parte abortable de un ATC que, como alternativa de disparo, tenga una llamada al lugar común que se acepta cuando éste está desmarcado. El código (o delay) asociado comenzará cuando el flujo de ejecución del proceso llegue a la transición y se detecte que el lugar común p está marcado. Una vez terminada la ejecución, se procederá a la retirada de la marca del lugar común. En este desmarcado, se puede producir una carrera entre varios procesos que hipotéticamente pueden terminar a la vez, por lo que la implementación debe protegerse contra la posibilidad de que, aún habiendo terminado, la marca sea retirada por otro proceso. Para ello se utiliza una llamada condicional a `desmarcar` y una variable booleana (`tr_acabada`) que adoptará valor `FALSE` si no se consigue desmarcar el lugar.

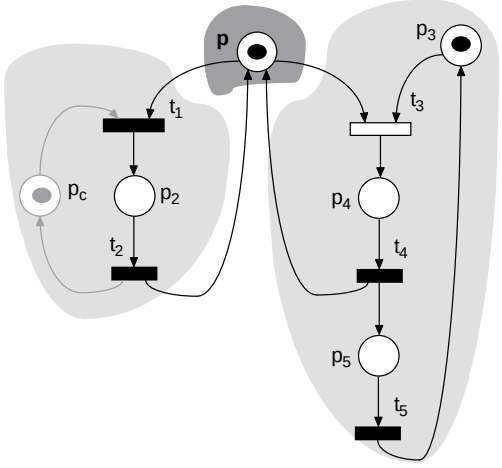


Figura 5.24: Red en la que se ha forzado la distribución de sus transiciones

En este caso se debe volver a intentar el disparo de la transición, volviendo a esperar a que el lugar común sea marcado. Por tanto el código de una transición CODE o TIME será el siguiente:

```
loop
  Lugar_Comun.Esta_Marcado;        -- Comienza cuando se detecta que el lugar está marcado
  select
    Lugar_Comun.Esta_Desmarcado;   -- El código o delay es abortado cuando se desmarca p
    tr_acabada := FALSE;
  then abort
      -- Acciones              -- Código o delay asociado a la transición
    tr_acabada := TRUE;
  end select;
  if tr_acabada then           -- Si la transición termina se trata de desmarcar el lugar
    select
      Lugar_Comun.Desmarcar;
    else                       -- Puede que el otro proceso lo haya intentado y haya
      tr_acabada := FALSE;     -- llegado antes. En este caso se vuelve al comienzo
    end select;
  end if;
  exit when tr_acabada;
end loop;
```

Si en el conflicto existen transiciones SYCO, su implementación tratará de desmarcar el lugar común con una llamada bloqueante a la entrada `Desmarcar`, implementado así (ver sección 5.3.1):

```
protected Lugar_Comun is
  entry Esta_Marcado;
  entry Esta_Desmarcado;
  entry Desmarcar;
  procedure Marcar;
private
  Marcado: integer := ...; -- Valor inicial de marcado
end Lugar_Comun;
```

```
protected body Lugar_Comun is
  entry Esta_Marcado when Marcado > 0 is
  begin
    null;
  end;
  entry Esta_Desmarcado when Marcado = 0 is
  begin
    null;
  end;
  entry Desmarcar when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
  end;
  procedure Marcar is
  begin
    Marcado := Marcado + 1;
  end;
end Lugar_Comun;
```

La técnica puede generalizarse para el caso de los conflictos acoplados. En este caso, el sincronizador local reducido deberá contar, además de con las familias de entradas para marcar y desmarcar los lugares que encapsula, con familias de entradas para que las transiciones detecten su sensibilización (comprueben que sus lugares de entrada contenidos en el sincronizador están marcados), y detecten su desensibilización (alguno de sus lugares de entrada deja de estar marcado).

Ejemplo

La implementación de los procesos de la figura 5.24 queda como sigue:

```
task body Proc_1 is
  tr_acabada: boolean := FALSE;
begin
  loop
    tr_acabada := TRUE;
    loop
      Lugar_Comun.Esta_Marcado;
      select
        Lugar_Comun.Esta_Desmarcado;
        tr_acabada := TRUE;
      then abort
        Codigo(t1);
        tr_acabada := TRUE;
      end select;
      if tr_acabada then
        select
          Lugar_Comun.Desmarcar;
        else
          tr_acabada := FALSE;
        end select;
      end if;
      exit when tr_acabada;
```

```
    end loop;
    Codigo(t2);
    Lugar_Comun.Marcar;
  end loop;
end;

task body Proc_2 is
  tr_acabada : boolean := FALSE;
begin
  loop
    tr_acabada := FALSE;
    loop
      Lugar_Comun.Esta_Marcado;
      select
        Lugar_Comun.Esta_Desmarcado;
        tr_acabada := FALSE;
      then abort
        delay t3;
        tr_acabada := TRUE;
      end select;
      if tr_acabada then
        select
          Lugar_Comun.Desmarcar;
        else
          tr_acabada := FALSE;
        end select;
      end if;
      exit when tr_acabada;
    end loop;
    Codigo(t4);
    Lugar_Comun.Marcar;
    Codigo(t5);
  end loop;
end;
```

Caracterización de los conflictos entre distintos tipos de transiciones en los que se debe forzar la distribución de sus transiciones

Se deberá forzar la distribución de las transiciones de un conflicto cuando en él aparezcan transiciones TIME o CODE con más de un lugar de entrada, y no todos estos lugares sean privados al protoproceso que implementa las transiciones. Se tratará de transiciones CODE o TIME en conflicto con otras transiciones y que estén involucradas en alguna comunicación (pues ésta es la consecuencia de que no todos sus lugares de entrada sean privados). La distribución se conseguirá mediante la adición de lugares implícitos que construyan elemenos de partición que cubran por separado las transiciones a distribuir.

Sin embargo, no siempre será posible añadir lugares implícitos para separar la implementación de transiciones TIME o CODE en conflicto. Por ejemplo, en la red de la figura 5.25 no es posible añadir un lugar implícito de forma que se construyan elementos de partición que cubran las transiciones del conflicto sin contener el lugar común. Si se separan las transiciones t_2 y t_3, y se intentan cerrar los protoprocesos resultantes añadiendo implícitos a la red, se vuelve a tener la situación inicial. En la sección 5.10 indicaremos como proceder a la implementación de estas situaciones.

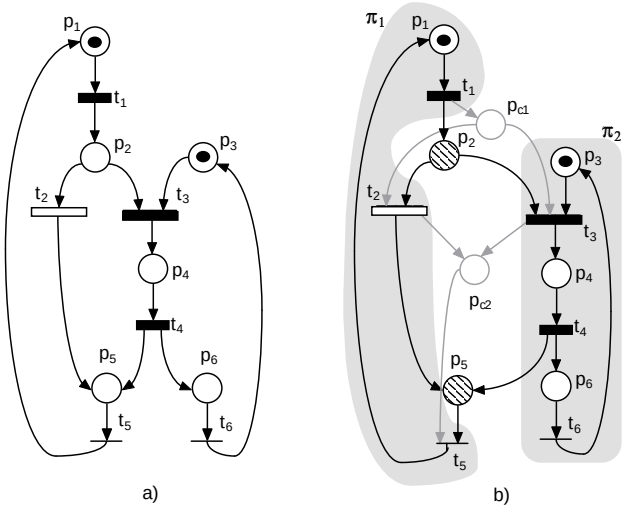


Figura 5.25: No siempre es posible añadir implícitos para distribuir las transiciones de un conflicto simple

5.8. Influencia de la implementación software sobre la partición de la red

Durante la exposición de los métodos de implementación software hemos detectado ciertas circunstancias en las que, debido a limitaciones de las primitivas del lenguaje de codificación (Ada 95), no es posible la realización software. Pese a ello, en algunos casos sí es posible modificar la partición de la red para conseguir implementarla. A pesar de que se han comentado en el contexto donde han surgido, recogemos aquí estas situaciones a modo de resumen:

1. Como se comentó en la sección 5.3.2, la primitiva de comunicación síncrona en Ada 95, la cita, permite únicamente la comunicación entre dos tareas. Por ello las estructuras de red que modelen comunicaciones a tres bandas no pueden ser implementadas. Estas situaciones pueden surgir cuando:

   - Una transición es compartida por más de un protoproceso
   - Una transición compartida tiene un lugar de entrada que es de comunicación con otro protoproceso.
   - Los dos casos anteriores, cuando lo que se comparte es un conjunto de transiciones y lugares conexos, o subred compartida.

   En todos los casos, la solución para obtener una implementación software pasa por no compartir la transición. Sin embargo esto no siempre será posible. Un contraejemplo se muestra en la figura 5.26, en la que aparece un conflicto acoplado entre las transiciones t_5 a t_7. Un cálculo de p-invariantes monomarcados evidenciaría la existencia de cuatro protoprocesos, dos de los cuales implementarían las transiciones del conflicto, siendo la transición t_6 compartida entre ambos. Sin embargo, esta transición desmarca un lugar de comunicación (p_{12}), por lo que para implementarla debe ser asignada en exclusividad a alguno de los protoprocesos, cerrando el que quede abierto mediante la adición de lugares implícitos. Pero, en este caso, de forma análoga al ejemplo de la figura 5.25, esto no es posible, por lo que esta red no puede ser implementada en Ada 95.

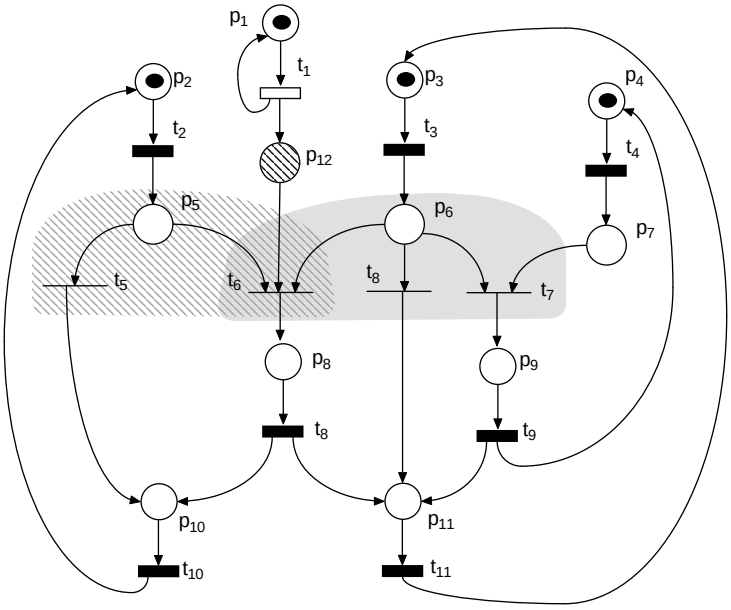


Figura 5.26: Conflicto acoplado no implementable en Ada 95

2. Para evitar la anomalía en la llamada múltiple de un proceso a uno o varios sincronizadores locales reducidos ($\mathcal{SL}r$), en la sección 5.6.3 se propuso agrupar, en un único sincronizador local, todos los lugares encapsulados por los $\mathcal{SL}r$ calculados para la implementación de un conflicto acoplado. De esta forma se podía implementar convenientemente el protocolo descrito para evitar la anomalía.

3. En la sección 5.7, se puso en evidencia la adecuación de la estructura ATC para la implementación de conflictos entre distintos tipos de transiciones, si éstas eran CODE o TIME y tenían lugares de entrada que no eran privados al protoproceso que las contiene. En este caso, se debe forzar la distribución de las transiciones en conflicto, para lo que es necesario añadir lugares implícitos que permitan la construcción de suficientes elementos de partición que no contengan al lugar común y que cubran todas las transiciones del conflicto (ver proposición 4.7.1).

5.9. Influencia del marcado inicial en la partición

Existe otra situación en la que la implementación software condiciona la solución del problema de cobertura de la red. Consideremos las redes de la figura 5.27. Ambas tienen la misma estructura, a pesar de que, para facilitar la comprensión, se hayan dibujado de distinta manera. Únicamente difieren en el marcado inicial. La implementación de la red de la figura 5.27.a se ha realizado mediante la compartición del conjunto de lugares y transiciones $\{t_4,\ p_1,\ t_1\}$, cuya codificación en Ada 95, como ya se ha comentado, corresponde con la ejecución de una cita extendida. Sin embargo, a pesar de tratarse de la misma red, esta implementación no es válida en el caso de la figura 5.27.b. En efecto, la marca que determina el flujo del proceso está colocada inicialmente en el lugar compartido que debería modelar la cita. Esto supondría que los procesos deberían comenzar su ejecución desde el interior de una cita, lo cual es imposible. Por ello, la implementación debe renunciar a compartir la estructura $\{t_4,\ p_1,\ t_1\}$ y conformarse con las transiciones t_4 y t_1.

La conclusión es que un lugar no puede compartirse si está inicialmente marcado, lo cual debe ser tenido en cuenta a la hora de resolver el problema de cobertura que conduce a la partición de la red.

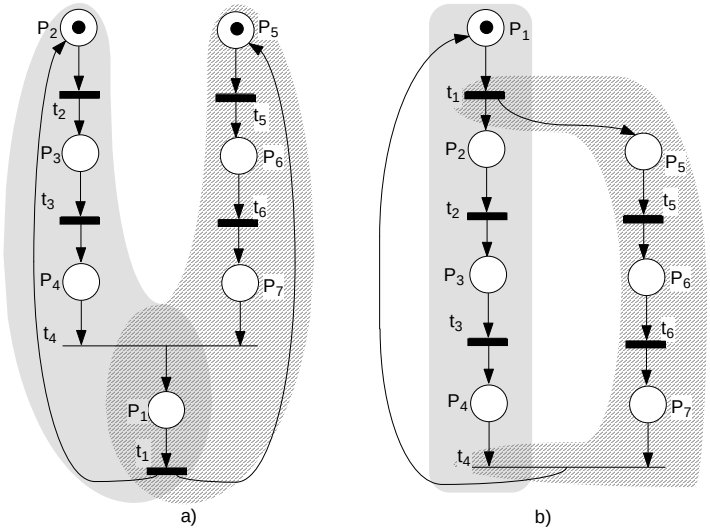


Figura 5.27: Influencia del marcado inicial en la partición de la red

5.10. Limitaciones de la técnica - Implementaciones mixtas

A lo largo de este capítulo se ha mostrado la forma de abordar la implementación software de los distintos elementos que pueden aparecer en un modelo de un sistema de tiempo real basado en una RdPT. Sin embargo, la semántica de concurrencia de las redes de Petri es más potente que la del lenguaje de implementación (Ada 95), por lo que hemos encontrado ciertas situaciones de conflictos en las que no se ha podido generar la codificación software, ni siquiera tras realizar las modificaciones en la partición de la red mencionadas anteriormente:

- Recordemos que en la sección 5.4.2, que se ocupó de la implementación de conflictos internos sin código, se mencionó que la implementación del lugar de comunicación en esta configuración debía ser del tipo tarea mensajero. Para que un lugar pueda ser implementado de esta manera es preciso que sea binario. Por ello, no podremos implementar conflictos internos sin código cuando los lugares de comunicación no sean binarios.
- En la sección 5.7 se ha comentado que no siempre es posible añadir lugares implícitos para forzar la distribución de las transiciones en conflicto.

En estos casos deberemos recurrir a técnicas mixtas, que resuelvan de forma centralizada la implementación software de ciertos conflictos de la red que no se han podido tratar con las técnicas descentralizadas.

La técnica está basada en el empleo de un objeto, similar al coordinador de una implementación centralizada, al que llamaremos *gestor de conflicto*. En él se centralizarán todos los elementos que pueden abortar la ejecución del código del conflicto, si lo hay, o todas las posibilidades de disparo de transiciones en un conflicto sin código. Gestionará el marcado de los lugares de comunicación y de los internos, y supervisará el tiempo de disparo de las transiciones TIME. De esta forma será él quien determine qué transición aborta la ejecución del código, o qué transición se dispara

en un conflicto sin código. La prioridad de este objeto será mayor que la de cualquier proceso que dependa de él, ya que es el responsable de acciones de control.

En el proceso que debe implementar el conflicto se utilizará una estructura ATC que implemente, en su parte abortable, el código a abortar, y que disponga de una única llamada a un punto de entrada del gestor de conflicto en la alternativa de disparo. Si el conflicto no cuenta con ninguna transición CODE, no será necesario utilizar la estructura ATC. Bastará con una llamada al gestor de conflicto que devolverá la transición que deba ser disparada.

El gestor de conflicto devolverá al proceso que implementa el conflicto, mediante un parámetro de salida, la transición que ha determinado disparar, para que el proceso actualice el marcado de forma adecuada. Por ello, la implementación del conflicto en el proceso será de la siguiente manera

```
select
  Gestor_Conflicto.Desmarcar(Transicion_a_disparar);
  case Transicion_a_disparar is
    when ... -- Actualizar el marcado de salida de la transición a disparar
  end case;
then abort
  Gestor_Conflicto.Comienza_codigo;
  Codigo_abortable;
  Gestor_Conflicto.Finaliza_codigo;
  -- Marcar lugares de salida de transición CODE
end select;
```

La situación descrita se esquematiza en la figura 5.28.

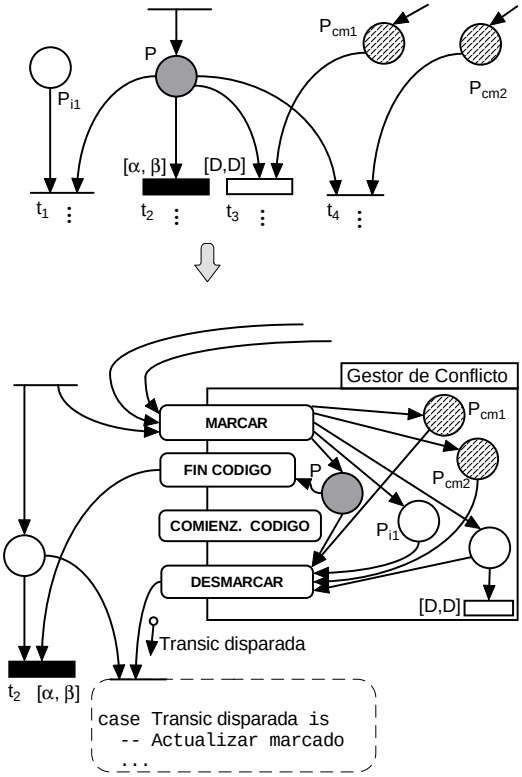


Figura 5.28: Estructura de implementación mixta de conflictos

Para poder decidir sobre la sensibilización de las transiciones, el gestor deberá implementar todos los lugares de entrada de las transiciones en conflicto, incluidos los privados. Es posible programar la política de resolución de conflictos más adecuada en el caso de que varias transiciones estén en conflicto efectivo. Como contrapartida, hacemos notar que se secuencializa la parte de control de la implementación porque el sincronizador, responsable del disparo de transiciones concurrentes, es una tarea y, por tanto, secuencial.

Para la implementación del gestor de conflicto se puede utilizar un esquema parecido al coordinador de una implementación centralizada: primero se acepta el disparo de las transiciones SYCO, si alguna de ellas está sensibilizada; tras ello, si alguna transición TIME está sensibilizada, se toma nota de su tiempo de disparo, gestionando estos tiempos en una lista de eventos temporales; tras ellos, si alguna transición CODE está sensibilizada, se acepta el comienzo de su ejecución. Una vez lanzada la ejecución de todas las transiciones sensibilizadas, el gestor quedará en disposición de aceptar el marcado asíncrono de lugares, que podrán interrumpir el código, o aceptar la notificación de finalización de la ejecución del código, o, si ninguna de estas circunstancias ocurre antes del vencimiento del evento de la transición TIME más urgente, se disparará la transición TIME. Un posible esquema de este comportamiento se muestra en el siguiente esqueleto de código:

```
task body Gestor_Conflicto is
  -- estructura de lugares encapsulados por el gestor
begin
  loop
    -- Resolver el conflicto entre las transiciones SYCO sensibilizadas
    if Alguna_SYCO_Sensibilizada then
      accept Desmarcar(Transicion_sensibilizada: out Transiciones_salida) do
        -- Actualizar marcado de los lugares del Gestor_Conflicto desmarcados
        -- por el disparo de Transicion_sensibilizada
      end;
    end if;

    for t in Transiciones_salida_TIME loop
      if Sensibilizada(t) then
        -- Anotar el tiempo de vencimiento de la transición en una lista de
        -- eventos temporales
      end if;
    end loop;

    if Sensibilizada(Transición_salida_CODE) then
      accept Comienza_codigo;
    end if;

    loop
      select
        accept Marcar(t: in Transiciones_entrada) do
          -- Actualizar marcado de los lugares del Gestor_Conflicto marcados
          -- por el disparo de la transición
        end;
      or
        accept Finaliza_codigo do
          -- Actualizar marcado de los lugares del Gestor_Conflicto marcados
          -- por el disparo de la transición CODE
        end;
```

```
        exit;
      or delay until tiempo_vencimiento_Transicion_TIME_mas_cercano;
        accept Desmarcar(Transicion_TIME_disparada: out Transiciones_salida) do
          -- Actualizar marcado de los lugares del Gestor_Conflicto desmarcados
          -- por el disparo de Transicion_TIME_disparada
        end;
        exit;
      end select;
      exit when Marcar'COUNT = 0;
    end loop;
  end loop;
end Gestor_Conflicto;
```

Es posible utilizar un esquema parecido cuando son varios los procesos que están en conflicto acoplado, pero en este caso es preciso proporcionar familias de entradas para que cada proceso desmarque y notifique el inicio y finalización de su código. Un ejemplo de implementación mixto para una red con conflicto acoplado se muestra en el anexo D.

5.11. Generación de código

A pesar de no ser un objetivo de la tesis, hemos de dedicar, al menos unas líneas, a tratar sobre cómo realizar la generación del código de cada uno de los procesos en los que se ha descompuesto una determinada RdPT.

Por construcción, cada protoproceso es una máquina de estados secuencial. Es conocido [Har80] que un proceso secuencial puede codificarse íntegramente utilizando exclusivamente una combinación de estructuras secuenciales, condicionales e iterativas, evitándose la utilización de primitivas de salto incondicional tipo `goto`, tradicionalmente desaconsejadas [Dij68].

La codificación puede realizarse de dos maneras: mediante código estructurado, o mediante el uso de una estructura `case`.

Generación de código estructurado

Es posible generar *código estructurado* a partir del flujo de control descrito por el elemento de partición que ha servido para determinar el protoproceso (recordar la sección 4.3.5), y siguiendo las reglas descritas en el presente capítulo, según las cuales, la estructura de transiciones de salida de cada uno de los lugares del elemento de partición determinan la primitiva o primitivas Ada 95 a utilizar.

No es propósito de esta tesis especificar completamente como se consigue una codificación estructurada del proceso a partir de su flujo de control. Para ello remitimos al trabajo de [Bri95], en el que se muestra una forma de obtener el código de un proceso secuencial, utilizando reglas de reducción de estructuras de la red. Si la red es reducible a un único *macrolugar* (*red estructurada*), la operación inversa de *despliegue* del macrolugar permite generar código siguiendo el secuenciamiento correcto. El código estructurado así generado es legible y fácil de manipular, además de ser en muchas ocasiones más rápido que el basado en `case`. Sin embargo, no todas las redes son reducibles a un único macrolugar, pese a las transformaciones referidas en [Bri95], por lo que, a veces, no es posible generar código estructurado para un proceso extraído de una red de Petri.

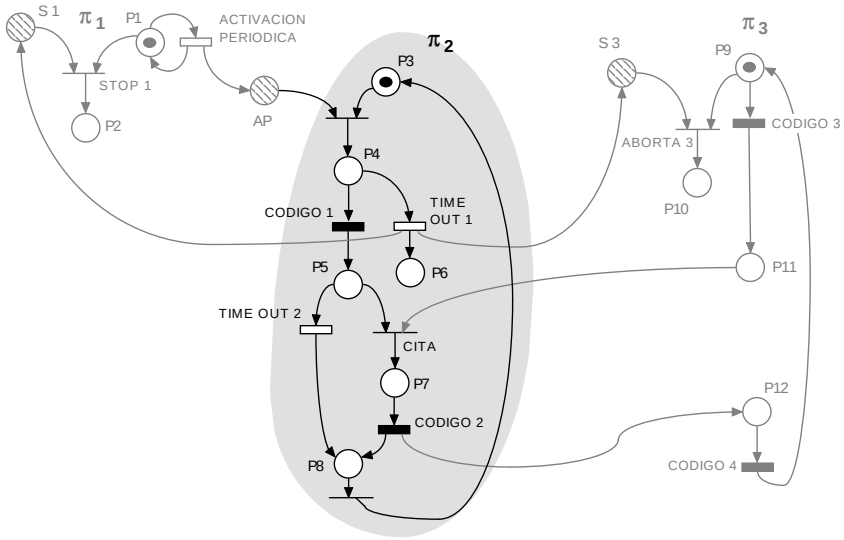


Figura 5.29: Red de ejemplo para su generar un interprete que la implemente

Generación de código basado en una estructura `case`

También puede abordarse la generación de código de acuerdo al Teorema de la Estructura o Teorema de Böhm y Jacopini[8]. La codificación estará basada en una primitiva `case` cuya ejecución se repite constantemente. Las distintas ramas de la estructura `case` estarán determinadas por el valor de una variable que almacenará el estado del proceso a implementar, y que hará las veces de contador de programa del proceso. En nuestro caso, este estado estará determinado por el elemento de partición generador del proceso, en concreto, por el lugar del mismo que esté marcado en cada instante determinado. En cada rama del `case` se codificarán las primitivas Ada 95 correspondientes a la estructura de transiciones de salida del lugar que determine la rama. Por ello nos referiremos a la variable selectora del `case` con el nombre de `contador_de_transiciones`. En cada iteración, la estructura `case` redirigirá la ejecución al próximo estado que corresponda, actualizando la variable `contador_de_transiciones`. Como ejemplo, presentaremos el código correspondiente al proceso π_2 de la figura 5.29 (compárese esta implementación con la que, para este mismo ejemplo se propuso en la sección 5.4).

```
task body Proc_2 is
  Time_out_1: constant Duration := ...;
  Time_out_2: constant Duration := ...;
  contador_de_transiciones: Lugar := P3;  -- Lugar marcado inicialmente
  TIME_disparada: boolean;
begin
  loop
    case Estado is
      when P3 => accept AP.Desmarcar;
                 contador_de_transiciones := P4;
      when P4 => TIME_disparada := FALSE;
                 select                -- Conflicto interno con código CODE + TIME)
                   delay Time_out_1;
```

[8]Existe un trabajo [Har80] en el que se recogen distintos nombres, demostraciones y comentarios sobre este teorema

```
                TIME_disparada := TRUE;
              then abort
                Codigo_1;
              end select;
              if TIME_disparada then
                S1.Marcar;
                S3.Marcar;
                contador_de_transiciones := P6;
              else
                contador_de_transiciones := P5;
              end if;
    when P5 => select               -- Conflicto interno sin código con comunicación
                accept Cita do
                  Codigo_2;
                end Cita;
                contador_de_transiciones := P8;
              or
                delay Time_out_2;
                contador_de_transiciones := P8;
              end select;
    when P6 => exit;
    when P8 => contador_de_transiciones := P3;
  end case;
 end loop;
end Proc_2;
```

La generación de código mediante la aplicación del Teorema de la Estructura es siempre aplicable, fácil de sistematizar, y además produce una representación explícita del estado, lo que aumenta la trazabilidad. Por ello será la estrategia de codificación adoptada en una futura automatización de la técnica de implementación de RdPT descrita en esta tesis. Pese a ello, y por razones de legibilidad, en los ejemplos de implementación software de RdPT de este capítulo, se ha generado código estructurado. Esta generación se ha realizado sin la aplicación de ningún algoritmo sistemático, como el de [Bri95], por lo que no es válido desde el punto de vista de la automatización del método.

5.12. Resumen de resultados y aportaciones del capítulo. Líneas abiertas

En este capítulo se ha mostrado una técnica de implementación software, en Ada 95, de los protoprocesos extraídos de una RdPT. La estrategia general se ha basado en concatenar bloques de instrucciones Ada 95 correspondientes a la programación de unidades de ejecución que integran un protoproceso. Estas instrucciones básicamente han venido determinadas por la configuración de las transiciones que hay a la salida de cada lugar del elemento de partición que genera el protoproceso, y que define su flujo de control.

A lo largo de las secciones del capítulo se ha indicado como realizar la codificación software de los distintos elementos de un protoproceso:

- Se comenzó con los lugares, distinguiendo entre lugares privados, implementables mediante variables locales al proceso, y lugares de comunicación, implementados mediante buffers o tareas mensajero (objeto pasivo o activo respectivamente), dependiendo la elección de las transiciones que los fueran a desmarcar.

- La implementación de las transiciones fue abordada a continuación, distinguiendo el caso de las compartidas de las situadas en el interior de un protoproceso. En el primer caso, la transición representa una sincronización entre los protoprocesos que la comparten, mientras que en el segundo, el protoproceso que la contenga ejecutará el código asociado, si la transición es CODE, se suspenderá durante un período de tiempo, si es TIME, o, en el caso de las SYCO, evaluará una condición o producirá un simple cambio de estado en el proceso. También se ha analizado la evitación de la deriva acumulativa que, en ciertas ocasiones, puede provocar la implementación de una transición TIME mediante una instrucción `delay`.
- Tras cubrir esta serie de elementos básicos, se trató la codificación de conflictos habiéndose realizado una revisión sistemática de los casos que se pueden presentar.
  - En el caso de los conflictos internos, las primitivas Ada a utilizar dependen de la presencia de transiciones CODE en el conflicto: si las hay, se deberá utilizar una estructura ATC, que permita abortar la ejecución del código si el resto de transiciones es disparado; si no las hay, la estructura a utilizar es un `select`, con eventuales ramas `delay` o `else`, o una estructura condicional.
  - En la implementación de los conflictos simples distribuidos en varios protoprocesos se ha comentado especialmente la implementación del lugar común generador del conflicto, y cómo implantar la política de resolución indicada en el capítulo de introducción.
  - De forma similar, en los conflictos acoplados se ha incidido en la implementación de los sincronizadores locales, y en cómo priorizar el acceso de los protoprocesos a los lugares en él encapsulados, de forma que se respete el protocolo de resolución de conflictos enunciado.
  - El tratamiento de los conflictos termina con una referencia a los distintos tipos de transiciones (CODE, TIME, SYCO) que pueden aparecer en un conflicto, y como las distintas configuraciones pueden tener influencia en la solución del problema de cobertura que determina la partición de la RdPT. Se indica cómo distribuir las transiciones si el conflicto está constituido por transiciones CODE o TIME cuyos lugares de entrada no son privados al protoproceso al que pertenecen.
- Cubiertas todas las posibles estructuras, se finalizó el capítulo analizando situaciones no implementables por limitaciones del lenguaje de codificación, situaciones que son evitables en ciertas ocasiones (por ejemplo, algunos casos de citas a tres bandas) mediante el replanteamiento de la partición de la red.
- Pese a ello, se han constatado determinadas situaciones en las que las limitaciones del lenguaje de codificación no permiten realizar la implementación software. La solución en este caso ha sido el empleo de técnicas mixtas de implementación, que se han presentado por primera vez en éste trabajo.

Nos parece destacable el hecho de que se ha mostrado, cuando la complejidad de la estructura generada no lo ha impedido, que la subred implementada y las primitivas Ada 95 para su codificación tienen el mismo comportamiento. Esto garantiza que la implementación propuesta es correcta.

Además de todo ello, se puede considerar otra aportación más: se han propuesto modelos sencillos, en términos de RdPT, de ciertas estructuras Ada 95, lo que puede ayudar a la generación de redes de Petri que permitan el análisis de programas Ada, como en la línea de trabajo propuesta en [STMD96].

Como líneas de investigación futura podemos resaltar el estudio de la generación automática de código estructurado, así como de las condiciones que se deben cumplir para que ésta sea posible. Esto permitirá la realización de una herramienta informática que realice la implementación software de manera automática.

Capítulo 6

Modelado de instrucciones Ada 95

6.1. Introducción

Las posibilidades que Ada 95 [ARM94, Rat95, Bar95, BW98] ofrece para la confección de programas multitarea, junto con sus instrucciones para la gestión del tiempo, hacen que este lenguaje sea adecuado para el desarrollo de sistemas de tiempo real. Por esta razón hemos adoptado Ada 95 como el lenguaje para la implementación software de RdPT que modelan sistemas de tiempo real.

Dedicaremos el presente capítulo a la construcción de RdPT que modelan las primitivas multitarea y de tiempo real Ada 95 de más interés para nuestro estudio, que ya han sido utilizadas en el capítulo anterior. Los modelos que aquí vamos a construir han sido la base de las justificaciones de corrección de las implementaciones software propuestas.

Antecedentes En la literatura se pueden encontrar varias referencias que proponen modelos de RdP y de RdPT para instrucciones Ada. La primera referencia que conocemos es [MZGT85], trabajo en el que se pretende clarificar la semántica del conjunto de instrucciones de Ada 83 relacionadas con la multitarea.

Otro conjunto de trabajos utiliza modelos de instrucciones Ada para el análisis de propiedades de vivacidad en programas Ada. Entre ellos encontramos [STMD96, DBDS94, SMBT90, TSM90, SC88] [BS93, BS94, BS95][BP99]. El objetivo de estos trabajos es estudiar propiedades de vivacidad, en concreto ausencia de bloqueos, de programas multitarea en Ada. Para ello proponen un modelo en términos de RdP del programa a analizar, al que llaman *Ada net* y, en vez de estudiar las propiedades del programa, estudian las de la red. Para ello se genera el grafo de estados asociado a la red. Para facilitar el análisis se trata de reducir el tamaño del grafo de estados utilizando técnicas que reducen el número de estados del grafo o reglas de reducción de redes que simplifican el modelo del programa.

Sin embargo, los modelos propuestos están bastante simplificados, basados en ejemplos particulares y enfocados al análisis de una propiedad en concreto. Además en ninguno de ellos se proporcionan modelos para las primitivas de concurrencia y tiempo real específicas de Ada 95. Para ello hay que recurrir a [GS97], donde ya aparecen modelos de objetos protegidos o de la estructura *ATC*, si bien de nuevo se trata de casos particulares de estas estructuras.

Objetivos Como se puede comprobar, la motivación de los trabajos existentes que relacionan las RdP y Ada ha sido la de servir de soporte al análisis de propiedades de vivacidad, todo ello aplicado a programas Ada ya existentes. Nuestro objetivo es distinto, pues perseguimos, como fin último, la generación de programas Ada 95. Por tanto, en este capítulo pretendemos mostrar

modelos, construidos con RdPT, que describan la semántica de las primitivas multitarea y de tiempo real de Ada 95. Estos modelos formales permitirán justificar, mediante técnicas de reducción y transformación, la corrección de las reglas de generación de código propuestas. El producto de este capítulo será un catalogo de redes con la correspondiente instrucción Ada 95 que modela.

Así mismo, no es desdeñable la consecución de un objetivo paralelo, como lo es para nosotros la clarificación de la semántica de las primitivas de concurrencia y tiempo real de Ada 95. Por tanto, este capítulo, puede tenerse en cuenta como contribución en las dos líneas de trabajo mencionadas.

Estructura del capítulo Las distintas secciones del capítulo irán presentando modelos de las primitivas Ada 95, de una forma gradual, comenzando por modelos básicos, y añadiendo, progresivamente, complejidad a la primitiva bajo estudio. Comenzaremos por el caso de las instrucciones `delay`, seguiremos con los modelos para la cita, para recalar en la estructura ATC, y, por fin, en el modelado de los objetos protegidos. La última sección repasa los resultados y aportaciones del capítulo.

Suposición En los modelos que se mostrarán a continuación se ha supuesto que el tiempo de ejecución de las acciones de control de las primitivas (como el comienzo de la ejecución de la primitiva, la evaluación de los parámetros o guardas, actualización de valores de variables, saltos y desviaciones de flujo, etc.) son despreciables frente a la ejecución de código que se realiza en la primitiva. Por ello se modelan con transiciones SYCO, cuyo disparo es inmediato. Esta aproximación ya ha sido utilizada en trabajos que han tratado temas relacionados, como, por ejemplo, en [BS95], si bien puede optarse por modelar teniendo en cuenta el tiempo de ejecución de las acciones mencionadas.

6.2. Modelado de las instrucciones `delay`

El efecto más importante de la instrucción `delay` es suspender la ejecución de una tarea hasta que se alcanza un instante especificado. La tarea así suspendida se dice que está bloqueada. Así mismo, sirve como alternativa en otras estructuras Ada 95 como la llamada temporizada, la aceptación de llamadas temporizada o la transferencia asíncrona de control. Por ello el modelo que se muestra en esta sección será utilizado más adelante en estas estructuras. Ada 95 distingue dos primitivas `delay`: el *delay relativo*, o simplemente `delay`, y el *delay absoluto*, o `delay until`.

En el *delay relativo* se debe especificar la cantidad de segundos que la tarea debe estar suspendida. El tiempo teórico en el que la tarea debe despertar es relativo al instante de ejecución de la instrucción, ya que éste se calcula sumando el valor de la expresión que acompaña a la cláusula `delay` (de tipo `Duration`) al valor del reloj del sistema en el momento de ejecución de la instrucción. El instante obtenido es redondeado por encima al siguiente tick de reloj del sistema. Sin embargo, en el *delay absoluto* el tiempo teórico de reactivación es el valor de la expresión que acompaña al `delay until` (que será de tipo `Time`) redondeado al próximo valor de tick de reloj del sistema (ARM 9.6(20)). En ambos casos, una instrucción `delay` con un valor cero o negativo[1] como tiempo de suspensión no provocará el bloqueo de la tarea (ARM D.9(5)). En realidad, la implementación de Ada 95 debería documentar el valor mínimo de tiempo de suspensión que hace que una tarea sea realmente bloqueada (ARM D.9(7) y (8)). Nosotros supondremos, sin pérdida de generalidad, que este valor es cero.

En ambas instrucciones el instante en el que la tarea debe despertar puede ser superior al que resulta de evaluar la expresión del `delay`. Influyen en esta desviación el valor de `Duration'small` y el redondeo por encima del instante de reactivación al tick de reloj más próximo. Valores máximos para estas desviaciones del instante de reactivación teórico deben ser documentados por la

[1] Un *delay relativo* con un valor negativo en su expresión es equivalente a un `delay 0.0` (ARM 9.6(32))

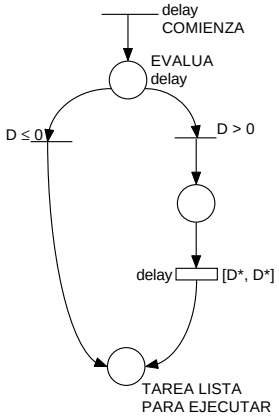


Figura 6.1: Modelo del `delay` *relativo*

implementación de Ada 95 (ARM D.9(12) y (13)). Además de la desviación debida a los redondeos, existe otra posible causa para que el retraso de una tarea sea superior al que se indica. Una vez que el instante de reactivación teórico es alcanzado, la tarea está teóricamente dispuesta para volver a ser ejecutada. Sin embargo, no despertará hasta que el planificador decida asignarle el procesador. En esta decisión influyen la prioridad de la tarea y la política de planificación elegida (mediante el `pragma Task_Dispatching_Policy`).

Centrándonos en el modelo, el elemento básico que nos permitirá modelar la instrucción `delay` es la transición TIME. Como hemos comentado, los intervalos de tiempo de disparo de las transiciones de las RdPT son especificados mediante dos valores racionales ($\mathbb{Q} \cup \{0\}$) [BD91], valores del mismo dominio que los posibles valores de las expresiones que acompañan a los `delay` (el tipo `Duration` es de tipo coma fija y es redondeado al tick de reloj, que siempre es un número racional). Además los intervalos de disparo son relativos al instante de sensibilización de la transición, lo que resulta adecuado para el modelado del `delay` relativo, pero es un obstáculo insalvable para el modelado del `delay` absoluto. El `delay` absoluto no puede ser modelado utilizando RdPT. Es necesario, tal como se mostrará en breve, recurrir a las redes TER (*Time Environment/Relationship Nets*) de [GMMP91] para proponer un modelo de la instrucción `delay until`.

6.2.1. Modelo del `delay` relativo

Se propone la red de la figura 6.1 para modelar la semántica de la instrucción `delay D`. Al ejecutarse la instrucción (disparo de la transición delay comienza) el lugar evalúa delay es marcado, representando el cálculo del instante de reactivación teórica de la tarea que ejecuta la instrucción. En este cálculo deben ser considerados los redondeos ya comentados, por lo que el producto de esta evaluación puede ser un intervalo de tiempo asociado a la transición TIME delay de valor $[D^*, D^*]$ distinto del esperado $[D, D]$, con $D^* \geq D$. Tras la evaluación, si la expresión D es mayor que cero, la tarea debe ser suspendida. Esto se representa mediante la sensibilización de la transición TIME delay (supongamos que esto ocurre en el instante θ_0). Por tanto, la tarea permanecerá bloqueada hasta que la transición TIME sea disparada, lo que no ocurre hasta el instante $\theta_0 + D^*$. Alcanzado este tiempo, la tarea podrá despertar y estará en disposición de volver a ejecutar, si el planificador se lo permite. Esto es representado mediante el marcado del lugar tarea lista para despertar.

Si la expresión del `delay` (D) es negativa o cero la transición TIME no es sensibilizada, pasándose directamente a marcar el lugar tarea lista para despertar. De esta forma queda reflejado en el modelo que la tarea no es bloqueada. Sin embargo hay que destacar que el lugar tarea lista para

despertar marcado no es garantía de que la tarea esté realmente siendo ejecutada, sino únicamente lista para ejecutar. Este comportamiento corresponde con la realidad y está descrito en el *anexo de tiempo real* de la [ARM94] en sus puntos D.2.1 y D.2.2. Si se utiliza la política de planificación `FIFO_Within_Priorities` (y en cualquier otra política definida por cualquier otra implementación de Ada 95 (ARM D.2.2(18)[2]), la ejecución de un `delay` que no cause un bloqueo de la tarea es un punto donde se replantea qué tarea debe ocupar el procesador (*task dispatching points*), lo que hace que ésta sea añadida al final de la cola de tarea listas para la ejecución que corresponda con su prioridad. La tarea es, por tanto, desalojada del procesador y realojada en una cola de tareas, del mismo modo que lo hubiera sido si realmente hubiera sido bloqueada y hubiera despertado del bloqueo transcurrido el tiempo especificado.

Éste es el modelo básico de `delay` que será utilizado en el resto de estructuras (basadas en la estructura `select`) que tengan alguna alternativa en la que se haga uso de él. Debido a que no es posible modelar la primitiva `delay until` ésta no será utilizada en ninguna de las estructuras `select` que vamos a mostrar. Todas ellas estarán basadas en el *delay relativo*. Así mismo, consideraremos por simplicidad que el valor de la expresión del `delay` será un múltiplo del tick del sistema, por lo que el intervalo de tiempo asociado a la transición TIME del modelo será el propio valor de la expresión, es decir, D.

6.2.2. Modelo del `delay` absoluto

La imposibilidad para especificar instantes de disparo absolutos en las transiciones TIME de las RdPT nos obliga a recurrir a las redes TER (*time environment/relationship nets*) [GMMP91] para el modelado de la instrucción `delay until`.

Conceptos básicos sobre redes TER

Para enriquecer la semántica de las RdP clásicas las redes TER unen el poder expresivo de las redes de alto nivel a la gestión del tiempo aportada por las extensiones temporales de las RdP. Básicamente son un tipo de redes de alto nivel, extensión de las *Environment/Relationship (ER) nets* propuestas también en [GMMP91].

De una forma muy simplificada (ver los detalles en [GMMP91]) una red ER es una RdP de alto nivel en la que las marcas, llamadas *entornos* (*environments*), son conjuntos de variables, es decir, un conjunto de identificadores y sus valores asociados, que residen en los lugares. Cada transición tiene asociada una *acción* que liga las marcas de los lugares de entrada y los de salida, y especifica que precondición deben cumplir las marcas de los primeros para que la transición pueda ser disparada, y qué entornos marcarán los lugares de salida una vez que la transición es disparada.

Las redes TER son una extensión de las anteriores en las que, por definición, cada marca tiene asociada una variable llamada *chronos* que registra el instante absoluto de tiempo en el que la marca fue creada. Así mismo, las acciones asociadas a las transiciones en las redes TER deben cumplir dos axiomas: el *axioma de monotonía local*, por el cual ninguna marca producida en un disparo puede tener un valor de *chronos* menor que ninguna de las marcas consumidas en el disparo; y el *axioma de restricción sobre el instante de creación*, por el que todas las marcas producidas en un disparo deben tener el mismo valor en la variable *chronos* (a este instante se le denomina *tiempo de disparo*).

En el mismo trabajo [GMMP91] se presenta un caso particular de redes TER, denominadas redes TB, en las que las marcas contienen únicamente la variable *chronos*. También son apropiadas para el modelo que nos ocupa.

[2]Sin embargo, en el modelo general de reparto del procesador entre tareas (*task dispatching model* (ARM D.2.1), no se especifica que la ejecución de un `delay` que no causa el bloqueo de la tarea sea un punto donde se replantea qué tarea debe ocupar el procesador (*task dispatching points*)

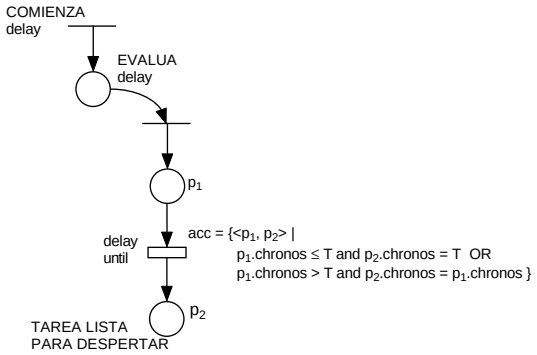


Figura 6.2: Modelo del `delay` *absoluto utilizando una red TER [GMMP91]*

Modelo del `delay until` utilizando redes TER

La red de la figura 6.2 modela la instrucción `delay until T` utilizando una red TER. En ella todas las transiciones, salvo la transición delay, son de disparo inmediato, y, por claridad, se han omitido sus acciones asociadas (sólo deberían especificar que el valor de *chronos* de las marcas producidas fuera igual al el *chronos* de las marcas de entrada, $acc = \{< p_e, p_s > | p_s.chronos = p_e.chronos\}$)

Tras comenzar la ejecución del `delay until` se evalúa el instante en el que la tarea debe ser reactivada (T). En este modelo la transición delay tiene asociada la siguiente acción de disparo, que se ajusta a la sintaxis de las redes TER: $acc = \{< p_1, p_2 > | p_1.chronos \leq T$ and $p_2.chronos = T$ OR $p_1.chronos > T$ and $p_2.chronos = p_1.chronos\}$. En esta expresión se recogen los dos casos posibles: si el instante de disparo ya ha pasado ($p_1.chronos > T$) el disparo es inmediato, produciéndose una marca con el mismo valor de *chronos* que el lugar de entrada p_1; o si el instante de disparo no ha pasado, la marca se generará en el lugar p_2 en el instante T.

6.3. Modelado de las citas

La *cita* es la primitiva de comunicación básica en Ada 95. Basada en el mecanismo de paso de mensajes, la cita puede ser vista como una primitiva de comunicación síncrona o como una invocación remota, si en ella se realizan acciones antes de devolver la respuesta (cita extendida, [BW97]). En la presente sección serán modeladas todas las primitivas Ada 95 que tienen que ver con la cita, es decir, las cláusulas `accept`, `select`, las llamadas a puntos de entrada (*entry*), llamadas condicionales, temporizadas, las guardas en los puntos de entra, las colas asociadas a los mismos, el aborto de tareas involucradas en citas, etc. Los modelos que se expondrán en esta sección complementan a los que pueden encontrarse en [MZGT85] y [GS97].

Se tratarán exclusivamente las citas entre tareas, sin considerar las llamadas a puntos de entrada de objetos protegidos, que serán estudiados en la sección 6.5. La exposición se realizará de modo gradual, de modo que en cada nuevo modelo se ilustre una característica más de la cita Ada 95. Hemos renunciado a la construcción incremental que tenga como objetivo el planteamiento de un modelo global de la cita, pues una RdP de tales características sería difícil de comprender. Pese a ello, no debe resultar difícil para el lector construir un modelo que aglutine varias o todas las características que sobre la cita iremos mostrando en apartados individuales.

6.3.1. Modelo básico de la cita

Revisión de la semántica (ARM 9.5.2; ARM 9.5.3) La cita está basada en una arquitectura cliente/servidor[3], en la que una tarea cliente es capaz de realizar llamadas a una serie de puntos de entrada (*entries*) que otra tarea servidora declara como públicos.

El servidor indica la disposición de aceptar solicitudes de servicio mediante la ejecución de una primitiva `accept`. La comunicación tendrá lugar cuando ambas tareas (cliente y servidor) hayan ejecutado sus respectivas petición de servicio e indicación de disponibilidad (puntos de sincronización). En ese momento tiene lugar la cita, procediéndose a la evaluación de los parámetros de entrada o entrada/salida enviados al servidor por parte del cliente. El servidor ejecuta el código asociado con la cita y una vez finalizado éste, los parámetros de salida o de entrada/salida son devueltos al cliente. A partir de entonces, ambas tareas evolucionan por separado.

Es posible que las tareas involucradas en la cita no alcancen sus puntos de sincronización al mismo tiempo, por lo que deberán esperar a que la otra tarea esté en disposición de establecer la comunicación. Si el servidor es el que espera una llamada de algún cliente, pasará a estado suspendido o bloqueado (liberando los recursos utilizados como, por ejemplo, la CPU). Si el cliente realiza una llamada y ésta no es atendida de inmediato, su llamada es encolada en una cola asociada a cada uno de los puntos de entrada del servidor. Pospondremos la discusión sobre las colas de entrada para más adelante (sección 6.3.5). Además es posible que un servidor tenga familias de entradas y entradas privadas. Este último aspecto no será estudiado en la presente memoria ya que no añade complejidad al mecanismo de la cita.

Descripción del modelo El modelo de la figura 6.3.a describe la semántica básica de la cita si no se consideran parámetros en la llamada. El modelo de la figura 6.3.b incluye el mecanismo de comunicación de parámetros.

La ejecución del `accept` por parte del servidor se modela mediante el disparo de la transición accept. Su disparo marca el lugar servidor espera llamada, lo que indica que el servidor está bloqueado esperando una llamada (ARM 9.5.2(24)). En el lado del cliente, la ejecución de una llamada al punto de entrada del servidor se modela mediante el disparo de la transición llamada a entry, que marca el lugar espera aceptación. La presencia de una marca en este lugar indica que el cliente está encolado en la entrada del servidor (ARM 9.5.3(12)). Cuando ambos lugares de entrada de la transición cita están marcados, esto es, tanto el servidor como el cliente han llegado a sus puntos de sincronización, la transición cita es disparada indicando el comienzo de la cita. Entonces los parámetros de entrada son evaluados procediéndose después a la ejecución, por parte del servidor, del código de la cita (lugar cita en progreso junto con la transición CODE correspondiente)(ARM 9.5.2(24)). Tras la finalización del código, los parámetros de salida son enviados al cliente (ARM 9.5.3(11)) que, tras recogerlos, prosigue su ejecución concurrentemente con el servidor (lugares servidor prosigue y cliente prosigue marcados)(ARM 9.5.2(25)). Todas las transiciones del modelo son inmediatas, salvo la transición CODE que modela el cuerpo de la cita.

6.3.2. Excepciones en la cita

Revisión de la semántica (ARM 9.5.2; ARM 9.5.3) El comportamiento básico de la cita se ve afectado de dos maneras si consideramos el efecto de las excepciones:

- Cualquier excepción elevada o producida en una cita puede ser manejada en su interior mediante el manejador de excepción adecuado. Si no es manejada en la cita, la excepción se propaga tanto al servidor como al cliente involucrados y la cita es terminada(ARM 9.5.2(24)).

[3]La nomenclatura utilizada en la ARM denomina a las tareas involucradas en una cita tarea llamadora y tarea llamada. En esta memoria se utilizará principalmente la nomenclatura *tarea cliente* (o simplemente cliente) por tarea llamadora y *tarea servidora* (o servidor) por tarea llamada, ya que, en nuestra opinión, es una nomenclatura más clara y fácil de leer

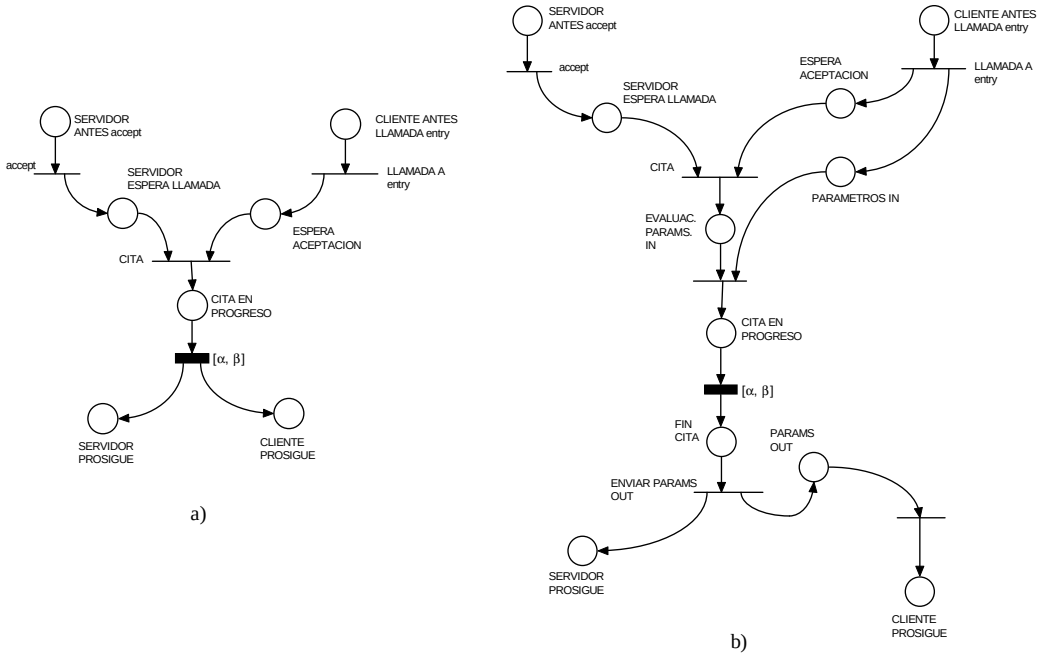


Figura 6.3: a) Modelo de la cita; b) Modelo que muestra la evaluación de los parámetros

- Cualquier intento por parte de una tarea de llamar a otra que haya terminado, esté completada o en estado anormal, será causa de elevación de la excepción `TASKING_ERROR` en la tarea llamadora (ARM 9.5.3(21)). Esta situación puede producirse incluso si la tarea llamada es terminada (por ejemplo como resultado de una sentencia `abort`) mientras la llamadora está en la cola de uno de sus puntos de entrada.

Descripción del modelo La inclusión de la primera de estas consideraciones en el modelo conduce a la figura 6.4. Por simplicidad, no se ha incluido en el modelo la parte correspondiente al mecanismo de paso de parámetros, aunque no debe ser difícil para el lector hacerlo si desea un modelo más completo. La elevación de una excepción está modelada por el marcado del lugar excepción. Este lugar puede ser marcado como resultado de una instrucción `raise` o por algún error o condición de excepción generada por alguna otra instrucción. Cuando este lugar es marcado, la transición SYCO descendiente del lugar excepción es disparada eliminando la marca del lugar `cita en progreso`. Esto provoca que la ejecución del código de la cita sea abandonada. Si el cuerpo de la cita incluye un manejador para la excepción la excepción es manejada (lugar excepción manejada marcado) y el bloque en el que se ha producido la excepción es terminado marcando fin cita (en el modelo se ha supuesto que este bloque es el cuerpo de la cita). Si no se dispone de manejador, la cita es terminada y la excepción se propaga al cliente y al servidor (ARM 9.5.2(24)). Todas las transiciones del modelo, salvo las transiciones CODE, son de disparo inmediato.

La segunda consideración sobre excepciones en citas será incluida en la siguiente sección.

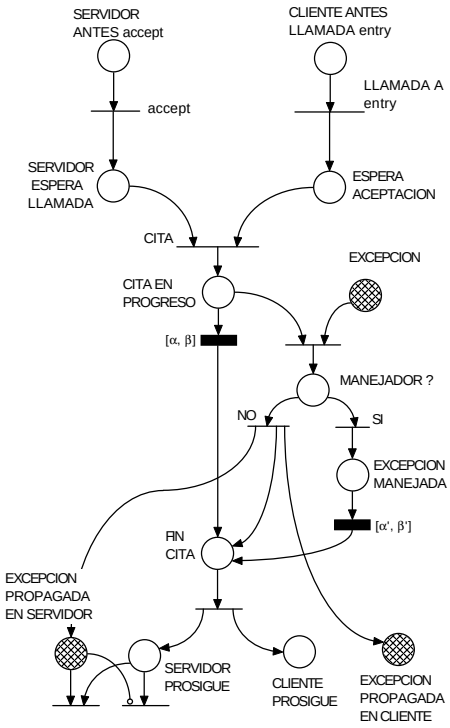


Figura 6.4: Modelo de la cita que incluye la posibilidad de que una excepción sea elevada en su cuerpo

6.3.3. `Abort` de una tarea que ejecuta una cita

Revisión de la semántica (ARM 9.5.3; ARM 9.8) Una tarea que se encuentra en una cita puede ser abortada. Los efectos de esta acción son muy distintos dependiendo del punto de ejecución en el que se encuentre la cita y del papel de la tarea en ella:

- Si la tarea abortada es el cliente y éste está encolado en algún punto de entrada del servidor, la tarea es sacada de la cola y la llamada cancelada (a no ser que su presencia en la cola sea debida a una instrucción `requeue` que no sea `requeue with abort`)(ARM 9.5.3(20)). Si la cita estaba ya en progreso, la cita debe ser finalizada completamente (pues se trata de una operación de aborto diferido para el cliente)(ARM 9.8(7)). Cuando esto ocurre la tarea llamadora se completa. El servidor no debe verse afectado en ninguno de los dos casos.
- Si la tarea abortada es el servidor, la cita se aborta[4] y la tarea que hace la llamada recibe una

[4]Como en cualquier sentencia `abort` la finalización del código de la cita puede no ser inmediata y no producirse hasta que éste no alcance algún punto de finalización de aborto (*abort completion point*), que, en cualquier caso, será alcanzado al final de la cita (el final de un `accept` es un punto de finalización de aborto)(ARM 9.8(15)). Si se está utilizando el anexo de tiempo real este aborto debe ser inmediato, al menos en sistemas monoprocesador (los que nosotros consideramos para nuestras implementaciones)

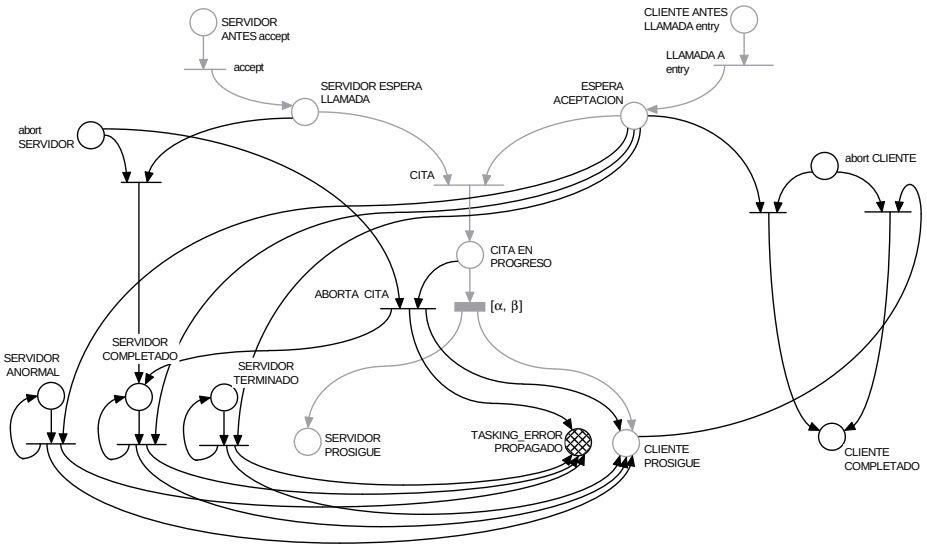


Figura 6.5: Modelo de la cita que considera el aborto de las tareas involucradas

excepción `TASKING_ERROR`(ARM 9.5.3(21)). Así mismo, todas las tareas que estén encoladas en todos los puntos de entrada del servidor también reciben `TASKING_ERROR`.

Descripción del modelo El modelo de la figura 6.5 muestra el comportamiento de la cita ante una situación de aborto de alguna de las tareas involucradas. Por claridad, se han suprimido las partes correspondientes al paso de parámetros y al modelado de excepciones durante la cita.

Por el lado del cliente, la indicación de que la tarea debe ser abortada se produce al ser marcado el lugar abort cliente. Si el cliente se encuentra bloqueado en la cola asociada al punto de entrada del servidor, la cita es cancelada inmediatamente, y al desmarcarse el lugar espera aceptación el cliente es eliminado de la cola y la tarea es completada marcándose el lugar cliente completado. Ahora bien, si la cita está en progreso, se debe esperar a que acabe antes de poder abortar el cliente. Esta situación se producirá cuando el lugar cliente prosigue sea marcado.

Análogamente, el servidor puede ser abortado cuando se encuentra bloqueado esperando en el `accept`. En este caso el servidor es completado inmediatamente (el `accept` es un punto de finalización de aborto, ARM 9.8(15)). Si el servidor es abortado cuando se encuentra ejecutando el cuerpo de la cita, la transición aborta cita es disparada lo que retira la marca del lugar cita en progreso y supone la finalización del código de la cita. Tras la finalización de la cita, el cliente debe ser informado mediante la propagación de una excepción `TASKING_ERROR` por lo que el lugar TASKING_ERROR propagado es marcado. En el modelo que se muestra en la figura se ha supuesto, por simplicidad que no se ejecuta ninguna operación de aborto diferido en el cuerpo de la cita. El aborto debe ser, por tanto, inmediato (si se usa el anexo de tiempo real). Si la cita incluyera alguna de estas operaciones, su código debería ser modelado de otra forma para reflejar este hecho, dividiéndolo en zonas donde el aborto es inmediato, y zonas en las que el aborto no es posible (instrucción de aborto diferido). Esta situación se refleja en la figura 6.6.

Así mismo se ha modelado el hecho de que un cliente intente establecer una cita con un servidor que está completado, terminado o en estado anormal debido, por ejemplo, a la ejecución de una sentencia de aborto. En este caso todos los clientes del servidor que estén encolados en sus puntos de entrada (con el lugar espera aceptación marcado) deben recibir una excepción `TASKING_ERROR` por lo que su lugar TASKING_ERROR propagado debe ser marcado.

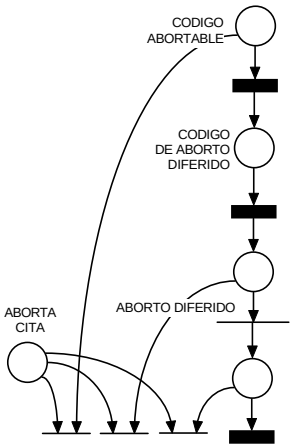


Figura 6.6: Aborto de un código con instrucciones de aborto diferido

6.3.4. Punto de entrada invocado por varios clientes

Revisión de la semántica Hasta ahora los modelos mostrados para las citas consideraban que un único cliente era el que podía establecer la cita con el servidor. Evidentemente, la primitiva no es tan restrictiva, de modo que cualquier número de clientes pueden llamar al punto de entrada de un servidor para establecer una cita con él. El modelo de la figura 6.7 describe este último comportamiento.

Descripción del modelo El servidor espera ser llamado en el lugar servidor espera llamada. Cuando uno de los lugares Ti espera aceptación se marque, se disparará la transición cita con Ti, es decir, se producirá la cita. Es importante que, al final de la ejecución del cuerpo de la misma, el control sea devuelto a la tarea que realizó la llamada. Por ello son necesarios los lugares Ti en cita, que registran qué tarea se encuentra ejecutando la cita con el servidor y que determinan cuál de las transiciones devolver control a Ti debe ser disparada. El modelo puede ser extendido fácilmente para considerar cualquier número de clientes añadiendo los correspondientes lugares Ti espera aceptación y Ti en cita y las transiciones cita con Ti y devolver control a Ti por cada nuevo cliente considerado.

Si son varios los clientes que llaman simultáneamente al punto de entrada del servidor, sólo uno de ellos puede ser atendido en cada momento. Es aquí cuando entrar en juego los colas asociadas con los puntos de entrada y sus políticas de servicio. Tratemos este tema en el punto siguiente.

6.3.5. Colas asociadas a los puntos de entrada

Revisión de la semántica (ARM 9.5.3; ARM D.4) Las llamadas a un punto de entrada son encoladas si éste no está abierto en el instante de la llamada (ARM 9.5.3(12)). Una entrada puede estar cerrada debido a que el servidor no está en disposición de aceptar llamadas (bien porque no ha alcanzado el `accept` o porque lo está ejecutando con otro cliente), o a que la guarda asociada a la entrada es falsa (ARM 9.5.3(5)).

Cuando uno o varios puntos de entrada de una tarea tienen llamadas encoladas, se debe elegir de algún modo qué llamada será atendida en primer lugar. Son dos las políticas básicas a considerar, siempre y cuando el anexo de sistemas de tiempo real sea tenido en cuenta (ARM D.4). La política de gestión cola se elige mediante el `pragma Queuing_Policy`, que admite dos valores, `FIFO_Queuing`, política por defecto, y `Priority_Queuing`. Si no se considera el anexo de sistemas de tiempo real la política es FIFO (ARM 9.5.3(17)). La política `Priority_Queuing` ordena las

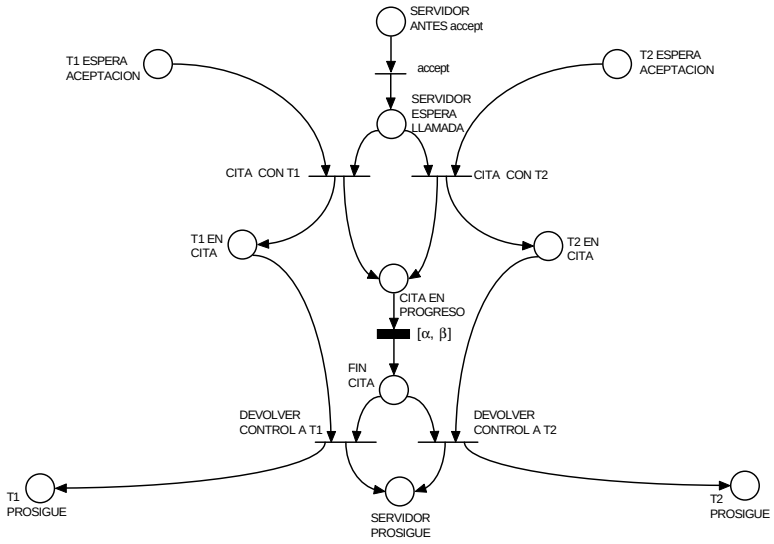


Figura 6.7: Modelo de la cita en la que un mismo punto de entrada puede ser invocado por dos clientes

llamadas de acuerdo a la *prioridad activa* de la tarea llamadora en el momento de la llamada (aunque luego puede variar), de forma que se elija la llamada correspondiente a la tarea con mayor prioridad de entre las que se encuentra encoladas. En caso de empate en el interior de una cola, se atiende al orden de llegada (orden FIFO)(ARM D.4(8)). Cambios en la prioridad activa de la tarea no producen cambios en la prioridad de la llamada, y, por tanto, tampoco en el orden de la cola. Lo anterior no es cierto si cambia la prioridad base de la tarea (ARM D.4(10)). Sin embargo la posibilidad de estos cambios no podrá ser modelado con RdPT, pues las prioridades asociada a las transiciones son estáticas. La política `FIFO_Queuing` atiende al orden de llegada de las llamadas dentro de cada cola.

Descripción del modelo En el caso de que exista un único cliente que pueda realizar llamadas, el propio lugar espera aceptación de los modelos anteriores sirve para indicar que el cliente está encolado en la entrada, esperando para ser atendido. Sin embargo, cuando son varios los clientes que pueden llamar al servidor, este modelo tan simple no sirve. Las colas deben ser modeladas explícitamente, así como su política de servicio.

La política `Priority_Queuing` es la más sencilla de reflejar en el modelo expuesto hasta el momento. Según esta política, las llamadas de una cola se ordenan según la prioridad de la tarea que las realiza. Para imponer esta política en el modelo de la figura 6.7 basta con asociar a cada transición cita con Ti la prioridad de la tarea cliente correspondiente. De este modo, esta prioridad será tenida en cuenta para deshacer el conflicto efectivo que aparezca cuando se produzcan varias llamadas simultáneas (es decir, hay varios lugares Ti espera aceptación marcados). Si varias de estas transiciones tienen la misma prioridad y se producen llamadas a través de ellas, la selección de una u otra está determinada por el orden de llegada, es decir orden FIFO (ARM D.4(9)). Los cambios de prioridad base en tareas encoladas exigirían cambios en la prioridad de las transiciones cita con Ti. Debido a que la prioridad de las transiciones SYCO en nuestros modelos de RdPT son estáticas, no podremos modelar estos cambios de prioridad base en tareas encoladas.

En cuanto a la política `FIFO_Queuing` (figura 6.8) se debe modelar la cola mediante una serie de lugares, uno por cada tarea cliente y nivel que ésta puede ocupar en la cola (si hay N tareas que

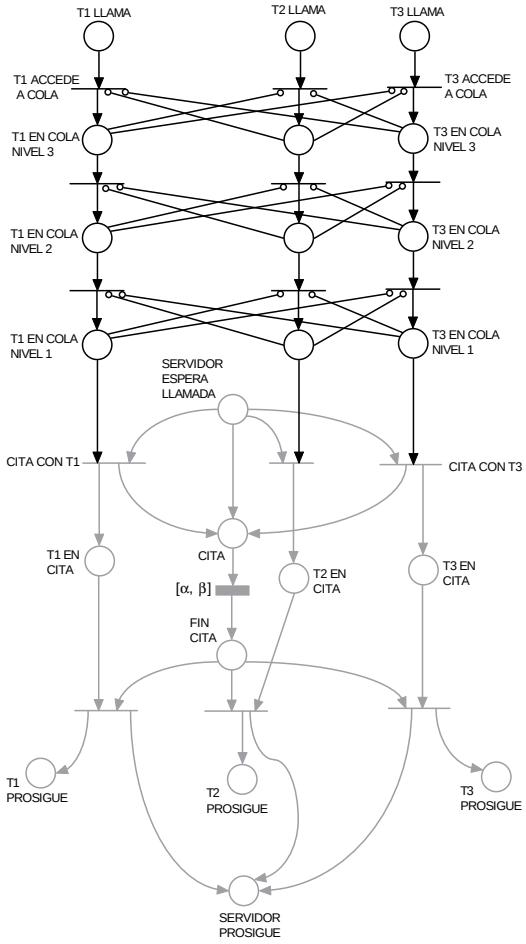


Figura 6.8: Modelo de la cola asociada a un punto de entrada con política de servicio FIFO

pueden llamar a la entrada, habrá N niveles por tarea, y en total $N \times N$ lugares), y un conjunto de transiciones que muestran el avance de las tareas encoladas por los niveles de la cola (figura 6.8). El paso de la tarea T_i del nivel k al $k-1$ de la cola se producirá cuando el nivel $k-1$ esté libre (los lugares Tj en cola nivel k–1, con $j \neq i$ no estén marcados). Una tarea que acceda a la cola irá descendiendo por ella hasta alcanzar el nivel más bajo que esté libre. Los niveles se irán ocupando por orden de llegada y sólo la tarea que ocupe el nivel 1 podrá realizar la cita con el servidor (orden FIFO). Cuando la cita comienza, el lugar Ti en cola nivel 1 queda libre, de modo que las tareas encoladas descienden al nivel inferior. Todas las transiciones del modelo de la cola son de disparo inmediato.

En ambos casos el modelado del valor del atributo `'COUNT` (ARM 9.9(5)) se consigue mediante un lugar que se marca cuando las transiciones Ti accede a la cola son disparadas y se desmarca al disparar alguna de las transiciones cita con Ti. El lugar será, por tanto, lugar de entrada de todas las transiciones cita con Ti y de salida de todas las transiciones Ti accede a la cola. Por claridad, este lugar no ha sido incluido en el modelo.

6.3.6. El `accept` selectivo. Modelado de `select`

Hasta el momento se ha considerado que un servidor podía esperar una única llamada en cada momento, ya fuera de un único cliente o de varios. El modelo debe completarse para reflejar una situación más general para un servidor, el cual ofrezca diferentes servicios que pueden ser solicitados simultáneamente y por distintos clientes. Además, hasta ahora, se ha modelado una situación de cita ineludible en la que una vez que las partes que desean comunicar llegan a sus puntos de comunicación no pueden hacer otra cosa que comunicar indefectiblemente. Es el momento de presentar modelos para las instrucciones que modelan la selección de varias posibles llamadas y el abandono del punto de espera de la cita una vez transcurrido un plazo, al no producirse la comunicación instantáneamente o cuando un servidor no pueda volver a ser llamado, porque sus clientes hayan dejado de existir.

La instrucción que permite estas variaciones a la cita básica es `select`, que tiene cuatro formas: `accept` selectivo, llamada condicional, llamada temporizada y `select` asíncrono o transferencia asíncrona de control. En este punto trataremos el `accept` selectivo, forma más básica de `select`.

Revisión de la semántica (ARM 9.7.1) El *accept selectivo* permite que un servidor espere simultáneamente varias llamadas a sus puntos de entrada. Si ninguna de las posibilidades de comunicación ofrecidas por el `select` es reclamada por ningún cliente, el servidor permanecerá bloqueado hasta que se reciba una llamada. Una vez que esta llamada se produce, la cita se lleva a cabo de la forma descrita en los puntos anteriores. Tras la cita, el cliente prosigue y el servidor puede ejecutar una serie de instrucciones antes de abandonar el `select` (ARM 9.7.1(16)).

Es posible que en el momento de ejecutar el `select` ya haya disponibles varias llamadas a la misma o distinta entrada. En este caso las llamadas permanecen encoladas en las colas asociadas a los puntos de entrada del `select`. La selección de una u otra cola cuando la llamada va a ser aceptada está sin especificar, aunque si se está utilizando el anexo de tiempo real se puede elegir una de las dos políticas de servicio de cola ya comentadas (ARM D.4). Si se elige la política `Priority_Queuing` las llamadas son encoladas por orden de prioridad de la tarea que las realiza, de modo que la llamada elegida será la correspondiente a la tarea con la mayor prioridad de entre todas las encoladas. Si dos colas tienen como primera tarea tareas de la misma prioridad, las entradas serán elegidas según el orden en el que aparecen en el `select` (ARM D.4(12)). La política `FIFO_Queuing` ordena las llamadas en las colas atendiendo al orden de llegada. Si hay varias entradas abiertas con llamadas encoladas, la elección de una u otra cola no está especificada. Ésta será la política si no se usa el anexo de tiempo real. Si sólo uno de los puntos de entrada tiene llamadas encoladas, el orden de servicio también está definido por la política de cola elegida y ya ha sido descrito anteriormente en la sección 6.3.5.

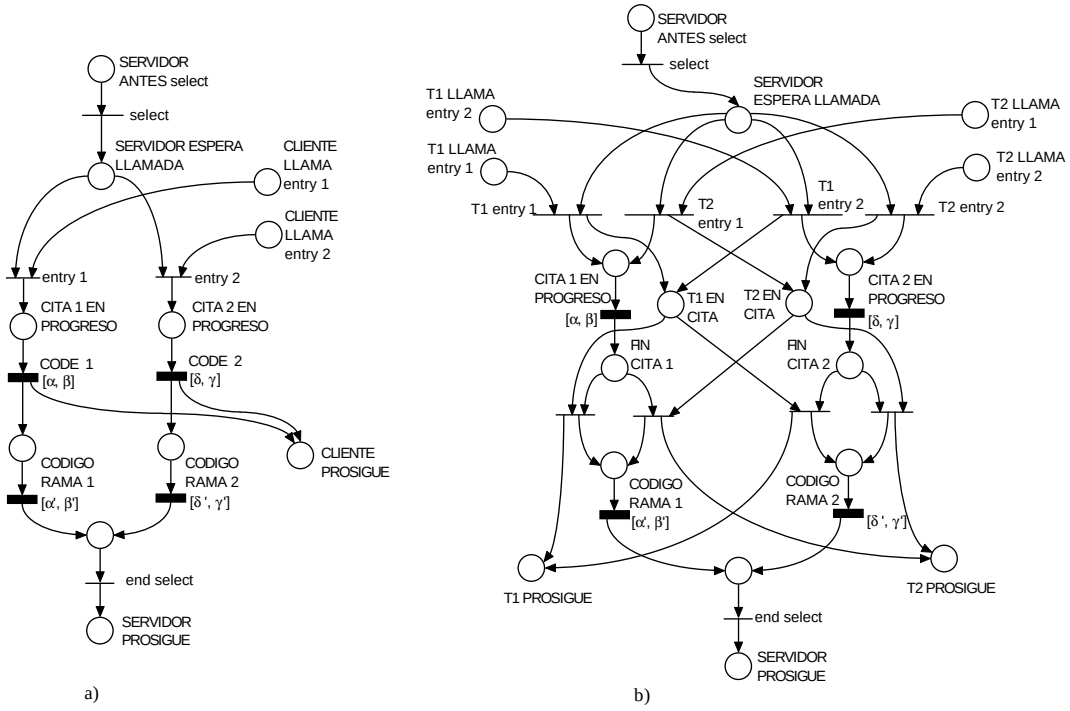


Figura 6.9: a) Modelo de `select` *en la que un cliente puede llamar indistintamente a dos puntos de entrada de un servidor; b) Modelo de un* `select` *con dos entradas en el que un cliente puede ser llamada por dos clientes a cualquiera de sus dos entradas*

Descripción del modelo El modelo de la figura 6.9 corresponde a la ejecución de un `select` con dos entradas, es decir, al siguiente código Ada 95:

```
select
  accept entry_1 do
    CODE_1;
  end entry_1;
  Codigo_Rama_1;
or
  accept entry_2 do
    CODE_2;
  end entry_2;
  Codigo_Rama_2;
end select;
```

En la figura 6.9.a se muestra el caso en el que un único cliente llama al servidor. La selección de las llamadas se realiza por medio de las transiciones SYCO entry i. El cliente indicará qué entrada solicita marcando uno de los lugares cliente llama entry i, que permanecerá marcado (cliente bloqueado en la cola de la entrada) hasta el momento en el que el servidor esté en disposición de aceptar la llamada (lugar servidor espera llamada marcado). Entonces, la transición entry i correspondiente se sensibilizará y será disparada inmediatamente, marcando el lugar cita i en progreso y comenzando la ejecución de la cita tal como se describió en la sección 6.3.1. Al final de ésta, el

disparo de la transición CODE i sensibiliza la transición código rama i que representa el código que el servidor puede ejecutar antes de alcanzar el fin del `select`.

La figura 6.9.b muestra una situación más compleja en la que el servidor puede ser llamado simultáneamente a cualquiera de sus dos puntos de entrada por dos clientes (T_1 y T_2). El modelo surge de la mezcla de los modelos de las figuras 6.7 y 6.9.a. El lugar servidor espera llamada debe tener tantas transiciones Tj entry i como posibilidades de comunicación existan entre clientes y puntos de entrada. Así mismo, como se describió en 6.3.4 es preciso registrar qué tarea ha sido aceptada para realizar la cita (marcando el correspondiente lugar Ti en cita) con el fin de devolver el control al cliente adecuado una vez que la cita haya terminado.

A pesar de que no se ha incluido explícitamente en el modelo propuesto, es sencillo implantar en él la política de servicio de colas de entrada `Priority_Queuing`. Bastará con asignar a las transiciones Ti entry j prioridades acordes con la prioridad de la tarea cliente que realiza la llamada (T_i). Si varias tareas tienen la misma prioridad las prioridades deberían ser reasignadas para tener en cuenta el orden de los puntos de entrada tal como parecen el `select`. Conscientemente se ha omitido un modelo de la estructura `select` que considerase la política FIFO de atención de colas de entrada. El tamaño de dicho modelo haría difícil su comprensión y opinamos que el lector interesado debe ser capaz de construir el modelo a partir de los de las figuras 6.9 y 6.8.

6.3.7. Ampliaciones del `select`: alternativas `delay` y `else`

Revisión de la semántica (ARM 9.7.1) Un servidor puede suspender la espera de llamadas por parte de sus clientes si éstas no se producen pasado un cierto tiempo (*timeout*) o no están disponibles inmediatamente. En el primer caso, el tiempo de espera es especificado utilizando una expresión `delay` como una alternativa más del `select`.[5] Es posible que aparezcan varios `delay` en un mismo `select` pero sólo el que se cumpla antes será tenido en cuenta. Si más de un `delay` vence en el mismo instante, no hay ningún punto de entrada disponible en ese instante, y se está usando la política de cola `Priority_Queuing` del anexo de tiempo real, se elegirá el delay que aparezca escrito antes en la estructura `select` (ARM D.4(13)).

Al comienzo de la estructura `select` se produce la evaluación de la expresión del `delay`. Tras ello el servidor queda bloqueado esperando alguna llamada. La alternativa `delay` será seleccionada si su tiempo ha expirado y no hay ninguna llamada a algún punto de entrada disponible en ese instante (ARM 9.7.1(18); ARM D.9(6)). Por ello, si la expresión es negativa o cero, la alternativa `delay` es equivalente a la alternativa `else`, ya que se requiere que las llamadas estén disponibles inmediatamente después del comienzo del `select`.

La alternativa `else` actúa si al comenzar el `select` no hay ninguna llamada esperando ser atendida o, como veremos en la siguiente sección, todas las guardas de las entradas son falsas (ARM 9.7.1(19)). Ambas alternativas, `delay` y `else`, no pueden aparecer al mismo tiempo en un mismo `select`.

Descripción del modelo El modelo propuesto en la figura 6.10 está basado en el de la figura 6.9.b, aunque se ha simplificado. En el modelo, una serie de clientes (lugares clientes llamando) pueden llamar a las entradas de un servidor (recordar que se deben modelar tantas transiciones de entrada como potenciales llamadas a los puntos de entrada por parte de los clientes puedan existir).

El objetivo del modelo es representar el funcionamiento de las siguientes estructuras basadas en la primitiva `select`:

[5] La alternativa `delay` puede ser absoluta o relativa. Tal como se ha comentado en la sección 6.2, el modelado de la primitiva `delay until` no puede conseguirse con la RdPT. Por ello en el modelo presentado sólo se considerarán alternativas `delay` relativas.

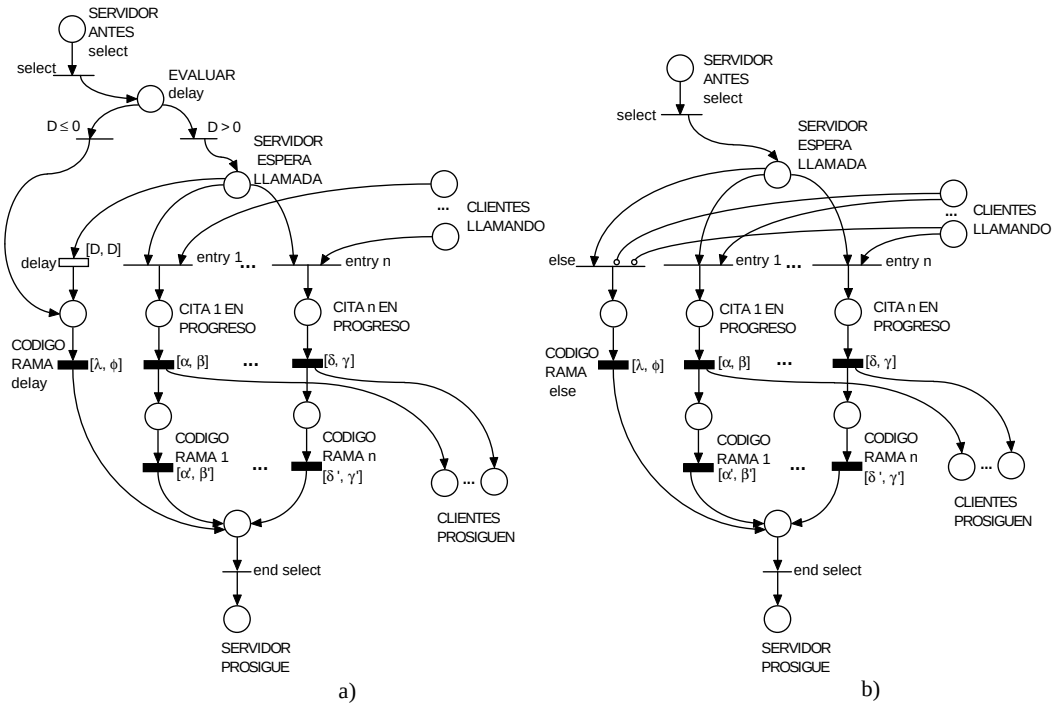


Figura 6.10: a) Modelo de `select` *con alternativa* `delay`*; b) Modelo de* `select` *con alternativa* `else`

```
select
  accept entry_1 do
    CODE_1;
  end entry_1;
  Codigo_Rama_1;
or
  accept entry_2 ...
  ...
or
  delay D;
  Codigo_rama_delay;
end select;
```

```
select
  accept entry_1 do
    CODE_1;
  end entry_1;
  Codigo_Rama_1;
or
  accept entry_2 ...
  ...
  ...
else
  Codigo_rama_delay;
end select;
```

En el modelo de la figura 6.10.a, una transición TIME hace el papel de `delay`. Al dispararse la transición select que da comienzo a la ejecución de la estructura, el lugar evaluar delay es marcado. Si la expresión del `delay` es negativa, el `delay` es equivalente a una transición SYCO con un tiempo asociado de [0, 0], que equivale al modelo de `select` con alternativa `else`. Si la expresión es positiva, el lugar servidor espera llamada es marcado, por lo que la transición TIME delay comienza a contar su tiempo de sensibilización. Si ninguno de los lugares clientes llamando es marcado antes de que transcurra el tiempo D, la transición TIME será disparada y retirará la marca de servidor espera llamada, abortando, de este modo, cualquier posibilidad de ejecución. Tras esto, el servidor puede ejecutar un código, modelado mediante la transición CODE código rama delay. Después el `select` finalizará. Si alguno de los lugares cliente llamando es marcado, la cita se producirá normalmente, tal como se ha descrito en la sección 6.3.6, desensibilizando la transición TIME y abortando, por

tanto, el timeout.

Como la prioridad de la transición TIME es menor que las prioridades del resto de transiciones SYCO correspondientes a puntos de entrada, si en el instante en el que expire el tiempo de sensibilización (tiempo del `delay`) alguna transición entry i está también sensibilizada, será esta última la disparada (ARM D.9(6)). Así mismo, si hubiera varias alternativas `delay`, las respectivas transiciones TIME deberían tener prioridades correspondientes a su lugar de aparición en la estructura `select`, para deshacer la ambigüedad en el caso de que varias de ellas venzan en el mismo instante.

El modelo de la alternativa `else` se muestra en la figura 6.10.b. El esquema es similar al anterior, salvo en la transición else, que será disparada en el caso de que ningún lugar cliente llamando esté marcado (de ahí los arcos inhibidores). Si éste es el caso, esta transición será disparada, procediéndose a la ejecución del código de la transición código rama else.

6.3.8. Alternativas con guardas en el `select`

Revisión de la semántica (ARM 9.7.1) Es posible asociar una condición booleana (guarda) a los puntos de entrada del `select` de una tarea y a su alternativa delay. La alternativa `else` no puede tener guarda. Al comienzo de la ejecución del `select`, todas las guardas son evaluadas (no se especifica en que orden, aunque se obliga a que la expresión del delay deba ser evaluada después de su guarda, si es que la tiene) (ARM 9.7.1(15)). Sólo aquellos puntos de entrada cuya guarda es cierta son tenidos en cuenta como puntos de aceptación de llamadas de la forma descrita en el punto 6.3.6. Las llamadas que se produzcan a puntos de entrada cerrados son encoladas (ARM 9.5.3(12)). Si todas las guardas de las alternativas de un `select` son evaluadas y su valor es falso y no hay una alternativa `else`, se elevará la excepción `PROGRAM_ERROR` en el servidor (ARM 9.7.1(21)). En este sentido, las alternativas sin guarda son tomadas como alternativas con guarda cierta.

Descripción del modelo El modelo que proponemos debe superponerse sobre el de la figura 6.10.a (o 6.10.b en el caso de la alternativa `else`, si bien esta alternativa no puede estar guardada) para obtener el modelo completo de un `select` con guardas. Cada transición entry i es enriquecida con un lugar entry i abierta que indica si la guarda está o no abierta. El resto del modelo será igual al de la figura 6.10. Por claridad, no se ha incluido la parte relativa a la evaluación del `delay`.

Uno de los primeros problemas a resolver para conseguir este modelo es la representación de las expresiones booleanas que conforman las guardas. Otros trabajos relacionados ([MZGT85, DBDS94]) han tratado el tema, decantándose para el modelado del valor de una expresión lógica por dos lugares complementarios que representan los posibles valores de la expresión. Si el valor de la expresión es cierto, el lugar que representa este valor estará marcado; si es falso, el otro lugar será el marcado. Si se quiere comprobar si una expresión es cierta, un arco unirá el lugar representativo del valor verdad con una transición. Otras formas de representación de expresiones lógicas utilizan un único lugar, que estará marcado si el valor de la expresión es cierto. Un arco inhibidor servirá para comprobar si la condición representada es falsa, y un arco normal, como en el caso anterior, bastará para comprobar si el valor es verdad. La última manera de modelar expresiones booleanas será asociar a una transición un predicado que coincida con la expresión. Si se desea comprobar si la expresión es falsa se asociará el predicado contrario.

En este modelo hemos preferido utilizar la primera opción descrita (pensamos que es la más clara) y, por tanto, modelar las guardas mediante parejas de lugares, Ci y $\overline{\text{Ci}}$, que sensibilizan cada uno una transición que se disparará en caso de que la guarda sea cierta o falsa (los lugares Cd y $\overline{\text{Cd}}$ corresponden a la guarda de la alternativa `delay`). Al comenzar el `select`, tras dispararse la transición select, son marcados todos los lugares evalúa i que indican que se está procediendo a la evaluación de los valores de las guardas[6]. Si una guarda es cierta marcará su correspondiente

[6]Aunque en este modelo las guardas son evaluadas en paralelo, esto no sucede en la estructura real. Hemos preferido modelarlo así por simplicidad y claridad. Esto no va en detrimento de la validez del resto del modelo y, en nuestra opinión, se trata de un hecho lateral de escasa importancia

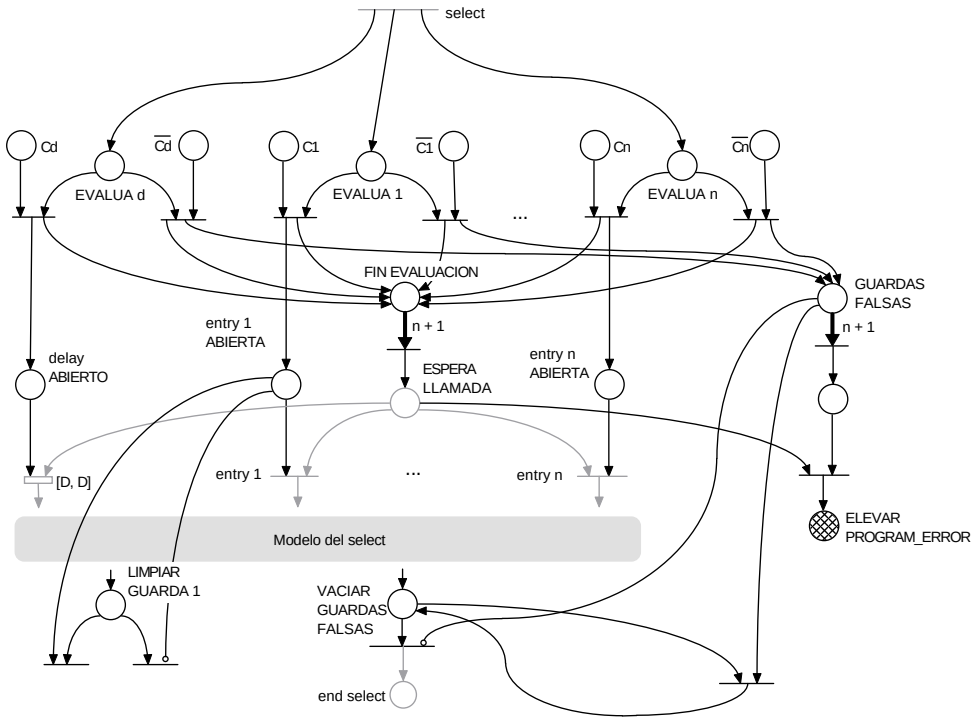


Figura 6.11: Modelo de las guardas asociadas a las entradas y alternativa `delay` *de una estructura* `select`

lugar de salida entry i abierta (delay abierto en el caso de la alternativa `delay`) permitiendo la posibilidad de que la transición entry i pueda ser disparada como respuesta a una llamada de alguna tarea cliente. A partir de este momento todo debe discurrir como en el modelo de la sección 6.3.7. Independientemente de su valor, toda guarda evaluada marca el lugar fin evaluación. Tras el proceso de evaluación de guardas este lugar debe contener $n + 1$ marcas (de n entradas y un `delay`. Cuando esto se produce, el lugar espera llamada es marcado. Ésta es la forma de asegurar que la aceptación de llamadas, así como la activación del `delay`, no se produzca antes de acabar la evaluación de las guardas.

Todas las guardas cuyo valor sea falso marcan el lugar guardas falsas. Si todas las guardas son falsas, en el citado lugar aparecerán $n + 1$ marcas, lo que sensibilizará su transición de salida y marcará el lugar elevar PROGRAM_ERROR, además de cancelar el `select` al retirar la marca del lugar espera llamada. Si las guardas no son todas falsas, el proceso descrito no se producirá, lo que propiciará que en el lugar guardas falsas queden algunas marcas espúreas, que deben ser eliminadas al acabar el `select`. Por ello tras la ejecución normal del `select`, se debe marcar el lugar vaciar guardas falsas para que su transición de salida vaciar limpie el mencionado lugar. Una vez conseguido esto, el lugar end select puede ser marcado, dando fin a la ejecución del `select`. Así mismo, todas aquellas guardas que hayan sido evaluadas como ciertas, y cuya entrada correspondiente no haya sido disparada, permanecerían marcadas (lugar entry i abierta). Por ello es necesario limpiar esas marcas tras las finalización del select, para lo que se deben marcar los correspondientes lugares limpiar guarda i.

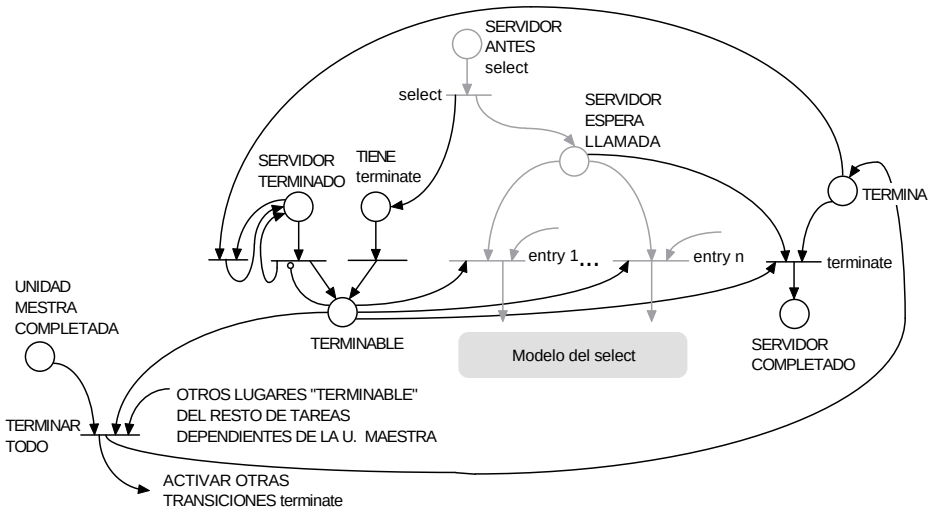


Figura 6.12: Modelo de `select` *con alternativa* `terminate`

6.3.9. Ampliaciones del `select`: alternativa `terminate`

Revisión de la semántica (ARM 9.7.1) La alternativa `terminate` permite que un servidor no espere indefinidamente a ser llamado por clientes que han dejado de existir. La alternativa puede estar guardada, pero no tener instrucciones asociadas. Así mismo la alternativa `terminate` no puede convivir con la alternativa `delay` ni la `else`.

Una tarea servidora que está suspendida esperando llamadas en un `select` con una alternativa `terminate` abierta puede ser completada si se verifican las siguientes condiciones (ARM 9.3(6)):

- La tarea servidora depende de alguna unidad maestra cuya ejecución se ha completado
- Toda tarea que dependa de esa unidad maestra está ya terminada o bloqueada en una estructura `select` con una alternativa `terminate` abierta.

Si se cumplen las condiciones anteriores el servidor, junto con todas las tareas bloqueadas que dependen de la unidad maestra, es completado.

Descripción del modelo El modelo se muestra en la figura 6.12. El servidor podrá ser completado cuando esté bloqueado esperando en un `select` con una alternativa `terminate`. Por ello cuando la estructura `select` comienza, un lugar etiquetado como terminable es marcado, indicando que el servidor puede ser completado. Si cualquiera de las transiciones que representa una llamada al servidor es disparada, este lugar es desmarcado. El lugar terminable debe estar presente en todas las tareas que dependen de la misma unidad de la que depende el servidor. El lugar terminable también será marcado si el servidor está ya terminado (lugar servidor terminado marcado).

Cuando la unidad maestra del que depende el servidor está completado (lugar unidad maestra completada marcado) y todas las tareas que dependen de él son terminables (sus lugares terminable están marcados) la transición terminar todo será disparada, marcando el lugar terminar y sensibilizando la transición terminate que cancelará el select y hará que la tarea servidora sea completada. El mismo proceso se repetirá en todas las tareas terminables que dependen de la unidad maestra. Si el servidor estaba ya terminado, con el fin de evitar que el lugar termina quede marcado sin posibilidad de desmarcarse, se ha incluido una transición de salida del lugar servidor terminado que retirará la marca del lugar comentado.

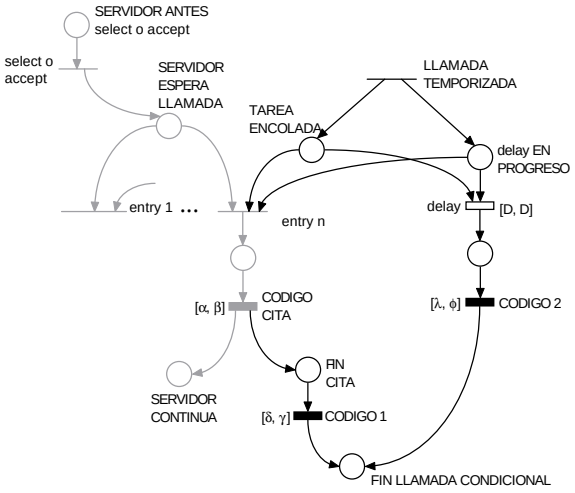


Figura 6.13: Modelo de llamada temporizada

6.3.10. Llamada temporizada a un punto de entrada

Revisión de la semántica (ARM 9.7.2) Del mismo modo que un servidor puede suspender la espera para recibir llamadas en sus puntos de entrada mediante las alternativas `delay` o `else`, un cliente puede cancelar una llamada a un punto de entrada si no es atendido en un periodo de tiempo determinado. La especificación del tiempo de espera se realiza por medio de un `delay` que será una de las dos únicas alternativas de esta forma de `select`, que tendrá como primera opción una llamada a un punto de entrada de un servidor. Estas alternativas no pueden tener guardas asociadas:

```
select
  servidor.nombre_entry;
  Codigo_1;
or
  delay D;
  Codigo_2;
end select;
```

Al comenzar la ejecución del `select` la expresión del `delay` es evaluada y luego se realiza la llamada. Si ésta es encolada y no se atiende antes de que el tiempo del `delay` expire, se intenta cancelar la llamada. Si se consigue cancelar, el código opcional que acompaña a la alternativa `delay` es ejecutado. Si la llamada es atendida, una vez acabada la cita se ejecuta el código opcional que acompaña a la llamada al punto de entrada (ARM 9.7.2(5)). La llamada es intentada aunque el valor de la expresión del `delay` sea negativa o cero (ARM D.9(6)).

Descripción del modelo El modelo se muestra en la figura 6.13 y corresponde únicamente al caso de llamadas a entradas de tareas, no de llamadas a un punto de entrada de un objeto protegido (este particular se tratará en la sección 6.5.4). Así mismo se ha obviado la parte correspondiente a la evaluación de la expresión del `delay`. Por ello el modelo es válido únicamente para expresiones $D > 0$. Si el valor de la expresión es $D \leq 0$, la estructura es equivalente a la llamada condicional, que será modelada en la siguiente sección.

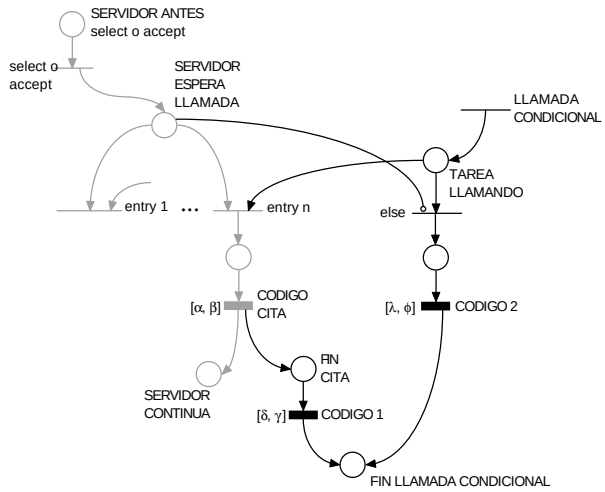


Figura 6.14: Modelo de llamada condicional

Al dispararse la transición llamada temporizada, que representa el comienzo de la estructura, los lugares delay en progreso y tarea encolada son marcados. Ambos lugares son entrada a las transiciones entry n, que representa una llamada a una entrada cualquiera de un servidor, y delay que representa el *timeout* asociado a la llamada. De este modo, el disparo de cualquiera de las transiciones desensibilizará a la otra. Si la disparada es la transición TIME, seguidamente se ejecutará el código de la transición código 2. Sin embargo, si se dispara la transición entry n, la cita comenzará y, a su término, se ejecutará el código de la transición código 1.

El lugar tarea encolada representa la permanencia de la tarea llamadora en la cola asociada al punto de entrada al que llama. Este modelo es suficiente si la política de servicio de colas del servidor es `Priority_Queuing`, ya que para modelar esta política no son necesarios más elementos. Si la política es `FIFO_Queuing` el lugar tarea encolada debería ser sustituido por la serie de lugares que representan la permanencia de la tarea llamadora en cualquiera de los niveles de cola FIFO del servidor (recordar la sección 6.3.5).

6.3.11. Llamada condicional a un punto de entrada

Revisión de la semántica (ARM 9.7.3) Similar a la opción anterior, la llamada condicional permite al cliente suspender la llamada si ésta no es inmediatamente aceptada. Es equivalente a la anterior con un tiempo de `delay` nulo o negativo (ARM 9.7.3(3)). Si la llamada es suspendida se puede proceder a ejecutar un código opcional que acompaña a la alternativa `else`.

```
select
  servidor.nombre_entry;
  Codigo_1;
else
  Codigo_2;
end select;
```

Descripción del modelo El modelo se muestra en la figura 6.14. Al marcarse el lugar tarea llamando se comprueba si el lugar servidor esperando llamada está o no marcado. En caso afirmativo se producirá la cita. Si, por el contrario, no está marcado, la transición `else` será disparada abortando la comunicación y seguidamente se ejecutará el código de la transición código 2. Este

modelo, al igual que el anterior, tampoco es válido si la llamada se produce a un punto de entrada de un objeto protegido.

6.4. Modelado de la transferencia asíncrona de control (A.T.C.)

La utilidad principal de la transferencia asíncrona de control es permitir a una tarea responder lo más rápidamente posible a un evento asíncrono externo a la misma. El evento suele ser notificado a través de una llamada a un punto de entrada, aunque también puede tratarse de un evento temporal de tipo *timeout* o *deadline*. Por ello presentaremos dos modelos para la transferencia asíncrona de control, uno para cada tipo de evento.

En toda estructura ATC existen dos partes: la *parte abortable* que contiene la secuencia de instrucciones que puede ser abortada por la ocurrencia del evento; y la *alternativa de disparo*, cuya finalización supone la ocurrencia del evento que debe abortar la parte anterior. La sintaxis de la estructura se muestra a continuación y en ella se recogen las dos posibilidades de eventos (`triggering_statement`) comentadas anteriormente.

```
asynchronous_select ::=
  select
    triggering_alternative
  then abort
    abortable_part
  end select;

triggering_alternative ::=
  triggering_statement [sequence_of_statements]
triggering_statement ::=
  entry_call_statement | delay_statement

abortable_part ::= sequence_of_statements
```

6.4.1. ATC con un `delay` como alternativa de disparo

Revisión de la semántica (ARM 9.7.4) Al comienzo de la ejecución de la estructura se evalúa la expresión del `delay` y, si el tiempo de expiración del mismo no ha pasado, se comienza la ejecución de la parte abortable (ARM 9.7.4(7)). Si el tiempo del expiración del `delay` se alcanza antes de que la parte abortable sea completada, esta última es abortada (ARM 9.8). Si la alternativa de disparo, es decir, el `delay`, es completada de forma normal (no debido a una cancelación) la serie opcional de instrucciones que le acompaña es ejecutada después de que la parte abortable haya sido abandonada (ARM 9.7.4(9)). Sin embargo, si la parte abortable es completada antes de que venza el tiempo del `delay`, éste es cancelado y se completa el ATC (ARM 9.7.4(8)).

Descripción del modelo La red de la figura 6.15.a responde a la siguiente estructura ATC (la figura 6.15.b es una simplificación en la que se considera que la parte abortable se compone de una única secuencia de instrucciones):

```
select
  delay D;
  Codigo_delay;
then abort
```

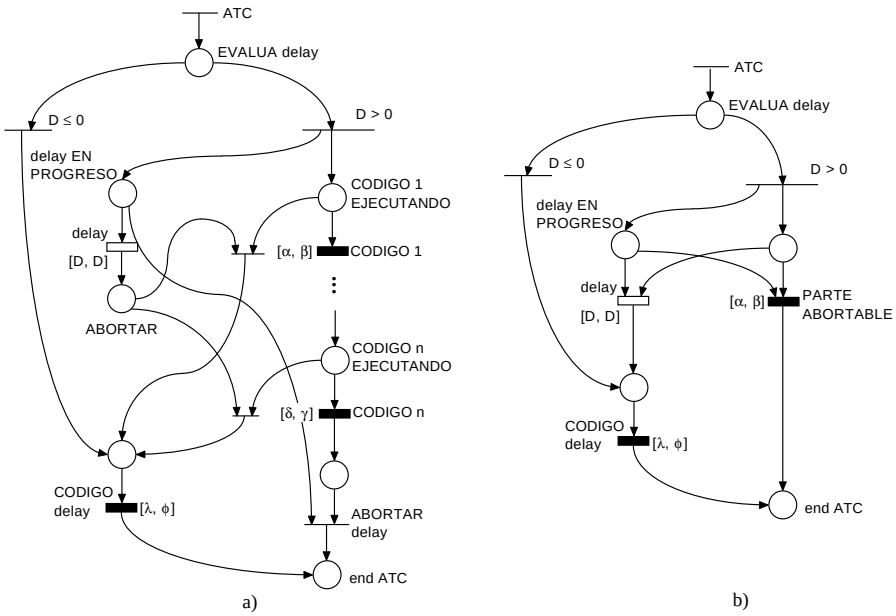


Figura 6.15: a) Modelo de ATC con una alternativa de disparo de tipo `delay` *b) Simplificación del modelo anterior si la parte abortable se compone de una única secuencia de instrucciones*

```
  Codigo_1;
  ...
  Codigo_2;
end select;
```

Al dispararse la transición ATC, que da comienzo a la ejecución de la estructura, se marca el lugar evalúa delay, que representa la evaluación de la expresión del `delay`. Si el valor de esta expresión es positivo, los lugares delay en progreso y código 1 ejecutando son marcados, lo que supone que la ejecución de la parte abortable está en curso y, al mismo tiempo, el evento temporal se ha activado. Si la secuencia de ejecuciones de las transiciones CODE código 1 a código n acaba antes de que la transición delay sea disparada, es decir antes del instante D, la transición abortar delay será disparada, desmarcando el lugar delay en progreso y abortando, por tanto, el `delay`. Por el contrario, si la transición delay es disparada antes de que se sensibilice la transición abortar delay, esto es, el `delay` expira antes de la finalización de la secuencia de códigos a ejecutar, el lugar abortar es marcado sensibilizando alguna de sus transiciones de salida, que retirará la marca del lugar código i ejecutándose que corresponda en ese momento. El aborto de una serie de instrucciones no tiene por qué ser inmediato. Puede retrasarse si, por ejemplo, se está ejecutando alguna instrucción de aborto diferido (ARM 9.8(5)). Si esto ocurriese se debería modelar el código en ejecución como se indicó en la sección 6.5, figura 6.6. En el modelo se ha supuesto que no hay operaciones de aborto diferido incluidas en la parte abortable. Una vez abortado el código de la parte abortable, se procede a la ejecución del código de la transición código delay.

Si el valor de la expresión del `delay` es menor o igual que cero, el código ni siquiera comenzará, pasándose directamente a la ejecución del código opcional que acompaña a la alternativa de disparo.

Si la parte abortable se compone de un único código (figura 6.15.b) el modelo es mucho más sencillo, ya que las transiciones delay y parte abortable comparten los mismos lugares de entrada. Por ello, el disparo de una de las transiciones desensibilizará, y por tanto abortará, a la otra.

6.4.2. ATC con llamada a punto de entrada como alternativa de disparo

Revisión de la semántica (ARM 9.7.4) Al comienzo de la ejecución de la estructura se realiza la llamada al punto de entrada especificado en la alternativa de disparo. Si la llamada es encolada, se comienza la ejecución de la parte abortable (ARM 9.7.4(6)). Si, tras ser encolada, la llamada es aceptada y la cita completada antes de que lo sea la parte abortable, esta última es abortada (ARM 9.8), tras lo cual la serie opcional de instrucciones que acompaña a la alternativa de disparo es ejecutada (ARM 9.7.4(9)). Sin embargo, si la parte abortable es completada antes de que se acepte la llamada, se intenta abortar la llamada (ARM 9.7.4(8);9.5.3(20)). Al tratarse de una operación de aborto diferido, la llamada sólo puede ser abortada si no ha comenzado. En este caso la estructura ATC termina. Pero si la cita ya ha comenzado, se debe esperar a que termine. Como en este caso la alternativa de disparo termina normalmente, en virtud de (ARM 9.7.4(9)) la secuencia de instrucciones que la acompaña es ejecutada. Por tanto, es posible que la parte abortable y la alternativa de disparo sean ambas completadas concurrentemente, situación que se producirá si la llamada es aceptada una vez comenzada la ejecución de la parte abortable, pero antes de que ésta termine.

Por otra parte, si en el momento de comenzar la ejecución del ATC, la llamada es aceptada inmediatamente la parte abortable ni siquiera empieza (ARM 9.7.4(6)).

El modelo que se mostrará seguidamente no contempla la posibilidad de que la llamada se produzca a un punto de entrada de un objeto protegido (de ello se hablará en la sección 6.5.6).

Descripción del modelo El modelo que se muestra en la red de la figura 6.16 corresponde a la siguiente estructura ATC (se ha supuesto por simplicidad que la parte abortable está compuesta únicamente de una secuencia de instrucciones):

```
select
  servidor.cita; -- En ella se ejecuta un código
  Codigo_tras_cita;
then abort
  Parte_aborable;
end select;
```

Al dispararse la transición ATC, que da comienzo a la ejecución de la estructura, se marca el lugar llamando a entry, que representa el intento de llamada a una entrada de un servidor. Si el servidor está en disposición de aceptar la llamada inmediatamente (lugar accept ya marcado) la transición entry inmediato es disparada, representando la ejecución de la alternativa de disparo. La parte abortable ni siquiera comienza porque el lugar de entrada de la transición parte abortable no es marcado. Este hecho es registrado en el modelo al marcarse el lugar parte abortable no comienza. La cita se lleva acabo de la forma habitual, ejecutándose la transición código cita. Tras ella se marca el lugar abortar, que en este caso no representa ninguna acción de abortar, pues, de sus transiciones de salida, la única sensibilizada es la que tiene como lugar de entrada a parte abortable no comienza. Después, el código de la transición código tras cita es ejecutado.

Por otra parte, si el servidor no está en disposición de aceptar la llamada inmediatamente (el lugar accept no está marcado), se comienza la ejecución de la parte abortable, al marcarse el lugar de entrada de la transición parte abortable. Así mismo, el lugar espera accept es marcado, representado que la llamada está encolada y que se está esperando a que el evento de disparo ocurra. En esta situación, si la transición parte abortable finaliza y se dispara antes de que el lugar accept sea marcado, la transición abortar espera será sensibilizada, pudiendo ser disparada y quitando la marca del lugar espeta accept. Esto representa que la llamada que estaba encolada en la cola de entrada del servidor ha sido cancelada. Sin embargo, si la transición parte abortable no finaliza antes de que el lugar accept sea marcado, cuando esto ocurra la transición entry podrá ser disparada, representando el comienzo de la cita de la alternativa de disparo. La cita se ejecuta

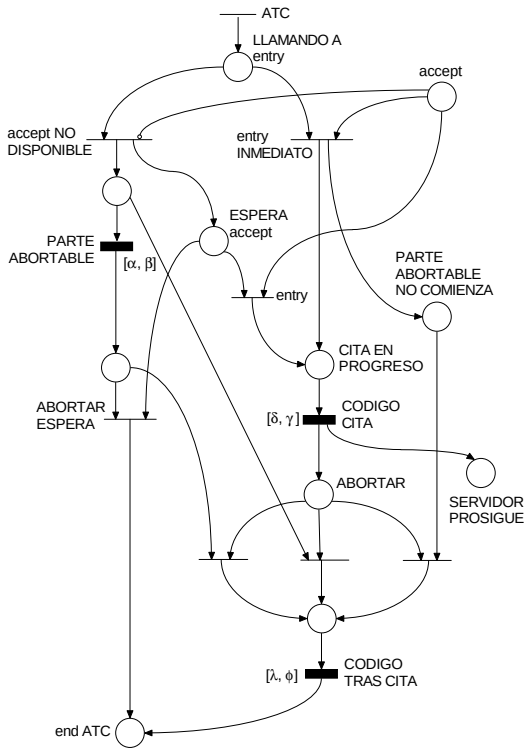


Figura 6.16: Modelo de ATC con una llamada a un punto de entrada como alternativa de disparo

normalmente y a su fin (cuando se dispara la transición código cita) de nuevo pueden ocurrir dos cosas. Si la transición parte abortable todavía no ha terminado, la transición de salida del lugar abortar situada en el centro será sensibilizada, pudiendo ser dispara, lo que representará el aborto de la parte abortable. Se está suponiendo que el aborto es inmediato (si hubiese alguna operación de aborto diferido el código de la parte abortable se debería haber modelado como en la sección 6.5, figura 6.6). La otra posibilidad es que la transición parte abortable ya haya terminado. En este caso el lugar abortar espera estará marcado, por lo que la transición de salida del lugar abortar situada más a su izquierda será disparada, procediéndose luego a la ejecución de código tras cita.

6.5. Modelado de objetos protegidos

Los *protected objects* (en adelante, objetos protegidos o `protected`) proporcionan un mecanismo de acceso a datos compartidos, en el que se permite la consulta concurrente de los datos por parte de varias tareas y se garantiza la exclusión mutua en las modificaciones de los mismos. En este sentido es similar a un *monitor* ([And91][Hoa74]). Así mismo proporciona un mecanismo de sincronización basado en condiciones (*condition synchronization*) más sencillo que el propuesto por [Hoa74] para los monitores y similar al propio de las *regiones críticas* [BH72]. Por tanto el objeto protegido de Ada 95 aglutina lo mejor de los dos mundos: agrupa un conjunto de datos y operaciones sobre los mismos en un único ente, y proporciona un mecanismo sencillo para controlar el acceso a los mismos que puede estar basado en ciertas condiciones imponibles fácilmente. Los objetos protegidos tienen carácter pasivo, por lo que dejan de existir cuando su entorno desaparece (incluso pueden ser creados directamente en la pila)

Cada objeto protegido tiene asociado un *lock* conceptual[7]. Este elemento es el responsable de garantizar el acceso en exclusión mutua a los datos encapsulados en el objeto protegido. Este acceso se realiza a través de las denominadas *operaciones protegidas*, que se dividen en subprogramas protegidos (funciones y procedimientos) y puntos de entrada protegidos (*protected entries*) (ARM 9.5.1(4)).

- Los procedimientos protegidos proporcionan acceso total, de modificación y consulta (lectura y escritura), a los datos del objeto protegido. Si se producen varias llamadas a un procedimiento protegido éstas deberán ser ejecutadas en exclusión mutua. Cuando se está ejecutando una llamada a un procedimiento protegido se dice que el objeto protegido tiene un *lock de lectura/escritura activo* ([BW98]).

- Los puntos de entrada protegidos son similares a los procedimientos protegidos, pero los enriquecen añadiendo a la entrada una barrera, condición booleana normalmente asociada a los datos encapsulados (ARM 9.5.2(30)). Sólo si la barrera está abierta (la condición booleana asociada es cierta) se podrá ejecutar la llamada a la entrada (ARM 9.5.3(7)). Si la barrera está cerrada, la tarea es encolada en una cola asociada a la entrada. Este mecanismo de barreras permite hacer sincronización basada en condiciones. Al igual que en los procedimientos, la ejecución de un punto de entrada protegido activa el *lock de lectura/escritura*.

- Las *funciones protegidas* proporcionan acceso únicamente para consulta (lectura) de los datos del objeto protegido. Se pueden ejecutar varias llamadas a funciones protegidas concurrentemente, pero se debe garantizar exclusión mutua con las operaciones de actualización (procedimientos y puntos de entrada). Cuando se está ejecutando una llamada a una función protegida se dice que el objeto protegido tiene activo un *lock de lectura*.

Hay una cola asociada a cada uno de los puntos de entrada de un objeto protegido. En ellas se almacenan las llamadas a entradas con barreras cerradas (ARM 9.5.3(12)). Si la barrera está ya abierta cuando una tarea se adueña del lock para ejecutar una llamada a un punto de entrada, ésta es realizada directamente, sin que la tarea pase por la cola (9.5.3(8,10)).

Las barreras asociadas a los puntos de entrada son evaluadas en instantes determinados (ARM 9.5.3)(aunque se permite que una implementación de Ada 95 las evalúe más veces de las estrictamente necesarias (ARM 9.5.3(29))). Esta evaluación es parte de una acción protegida por lo que necesario capturar el lock de lectura/escritura para realizarla. En particular las barreras son evaluadas cuando la llamada es realizada por primera vez y cuando se acaba la ejecución de un procedimiento o entrada protegida, pero antes de que se libere el lock. El ARM no especifica quién hace esta evaluación de barreras (puede ser el Run Time System o cualquiera de las tareas involucradas en las llamadas). Este aspecto no debe ser relevante.

Al ejecutar una operación protegida de lectura/escritura y antes liberar el lock, las barreras de las colas que no están vacías son reevaluadas, de forma que si alguna de ellas está abierta y su cola asociada contiene llamadas, éstas son atendidas antes de liberar el lock (ARM 9.5.1(7);9.5.3(15)). Por ello, una operación protegida puede continuar con otras sucesivas, de modo que se habla de *acciones protegidas* como el conjunto de acciones desde la apropiación del lock hasta su liberación. Estas acciones protegidas pueden ser una única, o una secuencia de varias de entre las tres siguientes: una llamada a un subprograma protegido desde fuera del objeto protegido (ARM 9.5.1(3)), la ejecución del cuerpo de un punto de entrada (ARM 9.5.3(10)), la inclusión (ARM 9.5.3(12)) o retirada de una llamada a un punto de entrada en su cola asociada (ARM 9.5.3(20)). Como consecuencia de este mecanismo, las llamadas encoladas en los puntos de entrada tienen preferencia ante las llamadas externas, por lo que una tarea puede ver retrasado su acceso al objeto protegido más tiempo del que correspondería a una simple ejecución de una operación protegida. El retraso

[7] El lock puede ser explícitamente implementado o a veces implementarse a través de las políticas de planificación (`Task_Dispatching_Policy` y `Locking_Policy`)

que sufre una tarea que quiere acceder al interior de un objeto protegido ocupado por otra no se considera como una suspensión (ARM 9.5.1(19)), por lo que no es tenido en cuenta en las llamadas condicionales o temporizadas. Estas llamadas no son consideradas hasta que la tarea es encolada en una cola asociada a un punto de entrada (ARM 9.7.2(5);9.7.3(3)). Estas especiales características nos obligarán a replantear modelos de algunas estructuras ya tratadas para adaptarlos a las llamadas a los puntos de entrada de un objeto protegido. En concreto replantearemos las llamadas condicionales y temporizadas y el ATC con una llamada a punto de entrada como alternativa de disparo.

El lenguaje no especifica quién debe ejecutar las llamadas encoladas como resultado de una reevaluación de las barreras (ARM 9.5.3(22)). Puede ser la tarea que originó la llamada o la que provoca la reevaluación, y, de hecho, esto es lo recomendado pues de esta forma el número de cambios de contexto se reduce (el *Rationale* de Ada 95 [Rat95] dedica el punto 9.1.3 a comentar este hecho).

La semántica de los objetos protegidos es todavía más compleja de lo que resumidamente se ha expuesto en estas líneas. Quedan temas por tratar como las familias de entradas o la prohibición de ejecutar acciones potencialmente bloqueantes en el cuerpo de las entradas (ARM 9.5.1(8)). Estos temas no serán tratados en este capítulo, ya que no modifican el mecanismo de utilización del objeto protegido. Además, en aras de la simplicidad de unos modelos ya de por sí complicados, no aparecerán en ellos el tratamiento de los parámetros en las llamadas (tema tratado en la sección 6.3.1 de este mismo capítulo), ni las excepciones que pueden elevarse en el interior del objeto protegido. A este respecto diremos que si la evaluación de una barrera eleva una excepción, la excepción `Program_Error` es propagada a todas la tareas que están en todas las colas del objeto protegido que utiliza la barrera (ARM 9.5.3(7)); lo mismo ocurrirá si el objeto protegido deja de existir cuando todavía existen llamadas encoladas en sus entradas (ARM 9.4(20)); por otra parte, una excepción elevada y no manejada en una operación protegida (subprograma o entrada) hace que la excepción se propague a la tarea llamadora (ARM 11.4(6)).

Además de las llamadas comentadas, es posible que una acción protegida llame a otra operación protegida desde su interior. Para esto no es necesario competir por el lock, pues ya se posee. La única restricción en este sentido es que una función protegida sólo puede llamar a otra función, no a un procedimiento o entrada (ARM 9.5.1(21)).

Es complicado modelar un objeto protegido mediante una RdPT. En primer lugar, el modelo debería incluir una representación de los datos encapsulados y lo que las operaciones protegidas realizan con ellos. Además estos datos suelen ser utilizados en las barreras de los puntos de entrada, lo cual también debería estar reflejado en el modelo, lo cual no es posible a no ser que se modele un objeto protegido en particular. En vez de ello propondremos modelos generales que no evidencian los datos protegidos y que representan las operaciones protegidas mediante la ejecución de códigos en general. Confiamos en que no sea muy difícil para el lector adaptar esta serie de modelos generales a su problema en particular. Como en casos anteriores, la exposición se realizará de modo gradual. Comenzaremos con el modelo de las funciones protegidas.

6.5.1. Modelado de las funciones protegidas

Las funciones protegidas son las únicas operaciones que pueden ser ejecutadas concurrentemente en el interior del objeto protegido, aunque en exclusión mutua con el resto de operaciones de lectura/escritura (procedimientos y entradas). Por ello se chequea el estado del lock de lectura/escritura, comprobando que el lugar lock de L/E libre de la figura 6.17 esté marcado. En las RdPT que nosotros utilizamos para modelar, la ejecución de código no es reentrante (los lugares de entrada de las transiciones CODE deben ser 1-limitados). Por ello no es posible modelar las funciones protegidas mediante una única transición CODE. Es necesario añadir tantas réplicas del código de la función (transiciones código función) como posibles llamadas concurrentes puedan existir.

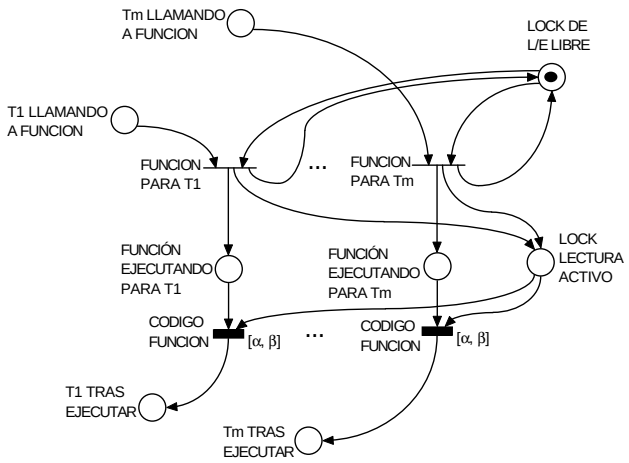


Figura 6.17: Modelo de objeto protegido con una función protegida

La llamada a la función se producirá a través del marcado por parte de alguna tarea (supongamos que existen m tareas, T_1 a T_m) del lugar Ti llamando a función. Si el lock de lectura escritura está libre (lugar lock de L/E libre marcado), la transición función para Ti se disparará, marcando el lugar función ejecutando para Ti y sensibilizando la transición CODE código función. Además se coloca una marca en el lugar lock lectura activo, que indica que alguna función está siendo ejecutada. Se retira una marca de este lugar al finalizar la ejecución del código de la función. El lugar lock lectura activo será chequeado en el resto de operaciones protegidas, con el fin de mantener la exclusión mutua.

6.5.2. Modelado de procedimiento protegidos

La red de la figura 6.18 modela la ejecución de un procedimiento protegido que puede ser llamado por m tareas, T_1 a T_m. Los procedimientos protegidos deben ser ejecutados en exclusión mutua con cualquier otra operación sobre el objeto protegido. Por ello debe comprobarse que el lock de lectura se encuentra libre (lugar Lock de L/E libre) y que no hay ninguna función ejecutándose (lugar lock lectura activo desmarcado). Al producirse una llamada al procedimiento, mediante el marcado del correspondiente lugar Ti llamando a procedimiento, y tras comprobar el estado de los locks, la transición Ti ejecuta procedimiento es disparada, marcándose el lugar procedimiento en progreso, que sensibiliza la transición CODE, y marcándose el lugar Ti dentro. Este lugar tiene la misma función que su correspondiente en el modelo de la cita, esto es, registrar que tarea ha accedido al interior del objeto protegido para que, al acabar la acción protegida, se devuelva el control a la tarea adecuada, mediante el disparo de la transición devolver control a Ti correspondiente.

Un aspecto importante que no ha sido reflejado explícitamente en el modelo es el hecho de que la ejecución de un procedimiento protegido puede provocar la reevaluación de las barreras asociadas a los puntos de entrada del objeto protegido, si los tiene. Este mecanismo de reevaluación comienza tras la ejecución de la transición CODE, y está esbozado en la figura 6.18 mediante el marcado del lugar reevaluar barreras. Volveremos sobre este punto, completándolo, en la siguiente sección.

6.5.3. Modelado de puntos de entrada protegidos

La red de la figura 6.19 modela la ejecución de un conjunto de n puntos de entrada protegidos (`entry`) que puede ser llamados por m tareas, T_1 a T_m, aunque en la red se muestran explícitamente

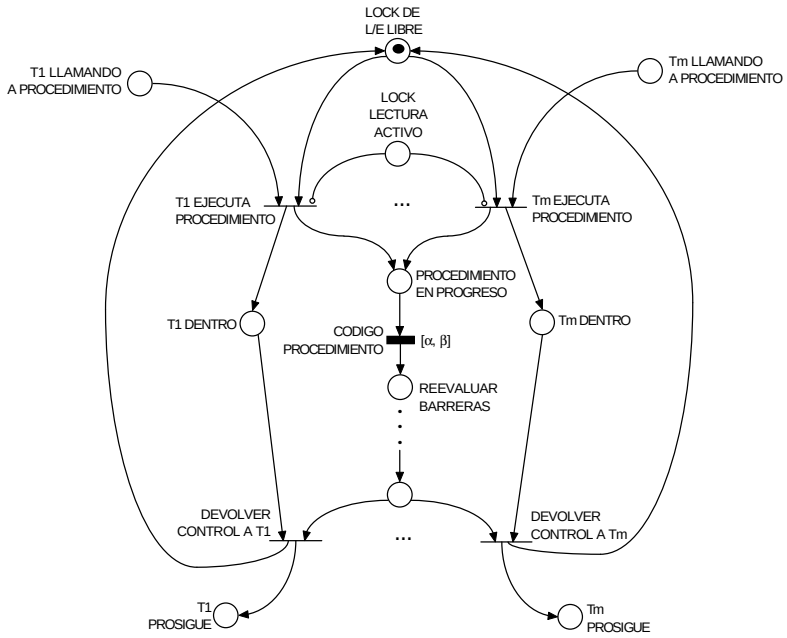


Figura 6.18: Modelo de objeto protegido con un procedimiento protegido

como T_1 y T_m llaman a la entrada 1, y como T_m llama a la entrada n.

Antes de comenzar con la descripción del modelo comentaremos las suposiciones que se han considerado en su elaboración. Se refieren a puntos no especificados claramente en la ARM:

- En primer lugar, se ha tomado el orden de aparición de las entradas en el objeto protegido (supongamos que sea 1, ... n) para imponer un orden de reevaluación de las barreras (no comentado en la ARM).

- Se ha supuesto que la reevaluación es llevada a cabo por la tarea que consigue el acceso al interior del objeto protegido, ejecuta la entrada y, por tanto, provoca la reevaluación de las barreras.

- Del mismo modo, esta tarea se encarga de la ejecución de las llamadas encoladas en colas que se abren tras una reevaluación de barreras.

- Supondremos que las barreras se reevalúan siempre, tras la ejecución de cualquier entrada.

- Por último, supondremos que la política de colas elegida es `Priority_Queuing` ya que modelas la política `FIFO_Queuing` añadiría mucha complejidad a la red (ver sección 6.3.5).

Cuando una tarea llama a un punto de entrada del objeto protegido, se marca el correspondiente lugar Ti llama a entry j. Si el objeto protegido está libre (lugares lock de L/E libre marcado y lock lectura activo desmarcado) la transición Ti entry j correspondiente es disparada. Si se produjesen varias llamadas simultáneas de distintas tareas, se deberá atender a la tarea con mayor prioridad. Por ello las transiciones Ti entry j deberán tener asociada la prioridad de la tarea T_i a la que sirvan. El disparo de esta transición, marca el lugar Ti dentro, que indica qué tarea ha conseguido el acceso al interior del objeto protegido.

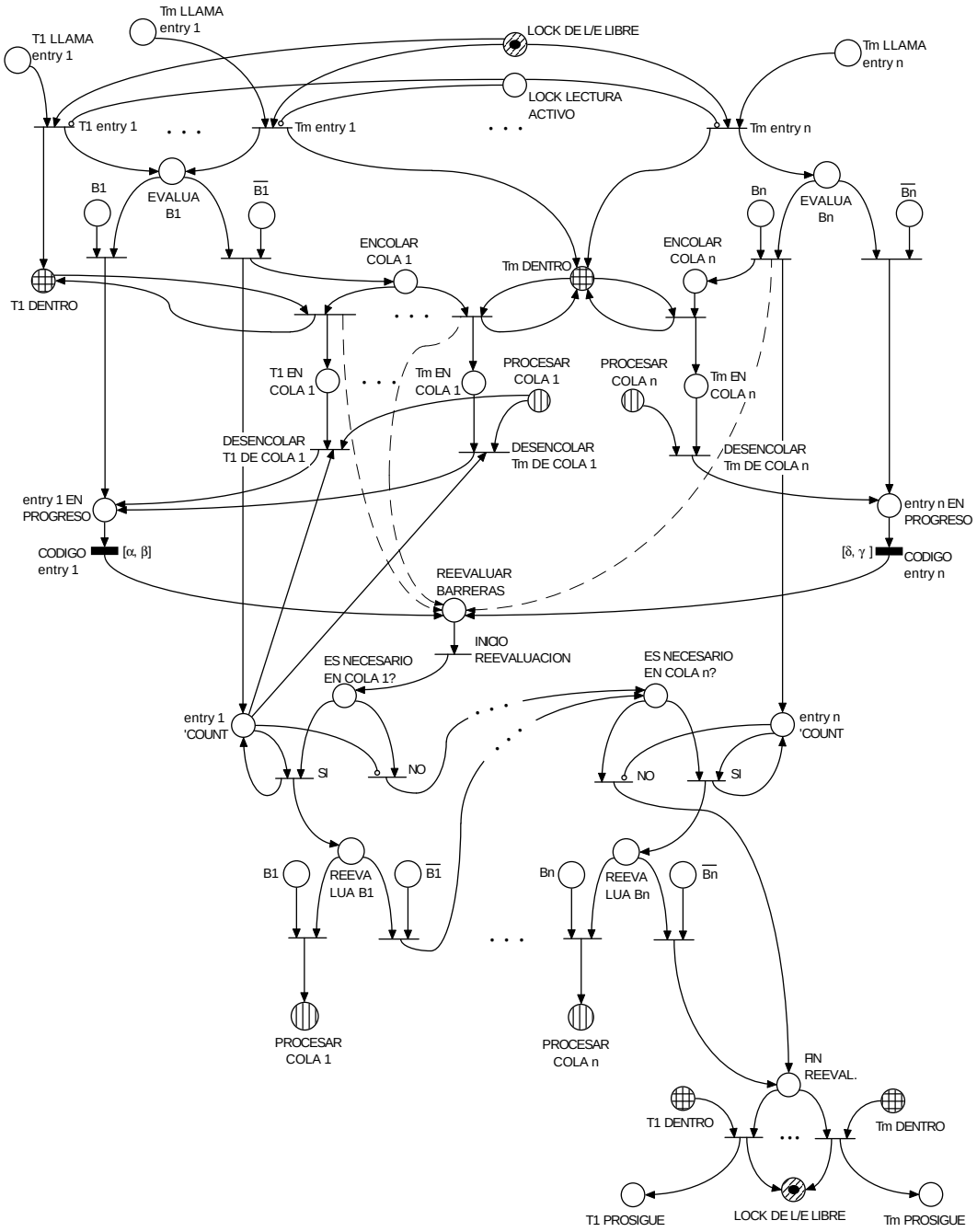


Figura 6.19: Modelo de objeto protegido con n puntos de entrada protegidos. Por razones de claridad, algunos lugares están duplicados (las correspondencias se indican mediante distintos entramados)

Una vez disparada una transición de entrada, el lugar evalúa Bj es marcado, representando la evaluación de la barrera asociada a la entrada. Si la barrera está abierta (lugar Bj marcado) el lugar entry j en progreso es marcado, sensibilizando la transición código entry j y dando comienzo a la ejecución del código de la entrada. Una vez acabada, el lugar reevaluar barreras es marcado comenzándose el proceso de reevaluación y ejecución de llamadas encoladas. A este respecto hemos supuesto que todas las entradas protegidas provocan la reevaluación de las barreras.

Si la barrera está cerrada (lugar $\overline{\text{Bj}}$ marcado), la llamada debe ser encolada, lo que se representa mediante el marcado del lugar encolar cola j. La cola es representada mediante una serie de lugares, Ti en cola j, uno por cada tarea que puede llamar a la entrada. El lugar correspondiente a la tarea T_i y la entrada j será marcado y se añade una marca al lugar entry j'COUNT. Para ello se consulta el lugar Tj dentro correspondiente. Esta marca se mantendrá en este lugar hasta que el lugar procesar cola j sea marcado. Como alguna de las barreras puede estar basada en el valor del tributo `'COUNT`, hemos representado que el encolado de una llamada provoca la reevaluación de las barreras (reflejado en la red de la figura 6.19 mediante los arcos dibujados con tramos).

La reevaluación de las barreras se produce tras la ejecución de alguna entrada protegida (disparo de transición código entry j), o finalización de un procedimiento protegido el encolado de alguna llamada. Cualquiera de estas acciones provoca el marcado del lugar reevaluar barreras que dispara la transición inicio reevaluación. Hemos supuesto que las barreras son estudiadas en orden.

Supongamos ahora, que a tarea T_i por medio de su llamada a la entrada j hace que las barreras sean reevaluadas, encontrándose que la barrera k, que mantiene encolada una llamada de la tarea T_h, queda abierta. La reevaluación de cada barrera comienza por el estudio de su necesidad, en función de que existan o no llamadas encoladas en la cola que gobierna la barrera. A este fin se chequea el valor del atributo `'COUNT` asociado a la cola, reflejado en la red mediante el lugar entry k 'COUNT. Si este lugar está marcado se produce la reevaluación, mediante el marcado del lugar reevalúa Bk. Si la barrera reevaluada es cierta, el lugar procesar cola k es marcado, sensibilizando las transiciones desencolar Th de cola k que correspondan con lugares Th en cola k marcados. Si varios de estos lugares están marcados, es decir, hay varias llamadas encoladas, el conflicto se deberá resolver a favor de aquella llamada correspondiente a la tarea más prioritaria. Por ello es necesario asociar a las transiciones desencolar Th de cola k la prioridad de la tarea T_h. Tras producirse el desencolado de la llamada, se retira una marca del lugar entry k 'COUNT y el lugar entry k en progreso es marcado, procediéndose a la ejecución de la llamada encolada. Observar que durante todo el proceso el lugar Ti dentro está marcado, lo que representa que es la tarea que inició la llamada la encargada de realizar la reevaluación y la ejecución de los códigos de las llamadas desencoladas.

En caso de que la barrera sea falsa o el lugar entry k 'COUNT esté desmarcado, se procede al planteamiento de la reevaluación de la siguiente barrera, en los mismos términos ya descritos. El proceso continuará hasta que todas las colas estén vacías o ninguna barrera sea reevaluada y encontrada abierta. Entonces el lugar fin reeval. será marcado, devolviéndose el control a la tarea que inició todo el proceso y que ha sido registrada mediante el lugar Ti dentro. Tras ello el lock de lectura/escritura es liberado y la tarea T_i prosigue (marcado de los lugares lock de L/E libre y Ti prosigue).

6.5.4. Llamada temporizada a punto de entrada de objeto protegido

Según se comentó en la sección 6.3.10, la red de la figura 6.13 reflejaba el modelo de una llamada temporizada a una entrada de una tarea. El caso de la llamada a una entrada de un objeto protegido es más complicado, debido a que el retraso que sufre la tarea para conseguir acceder al interior del objeto protegido no es considerado un bloqueo de la misma y, por tanto, no es tenido en cuenta para comenzar la consideración del tiempo del `delay` de la llamada (ARM 9.5.1(19)). Éste no

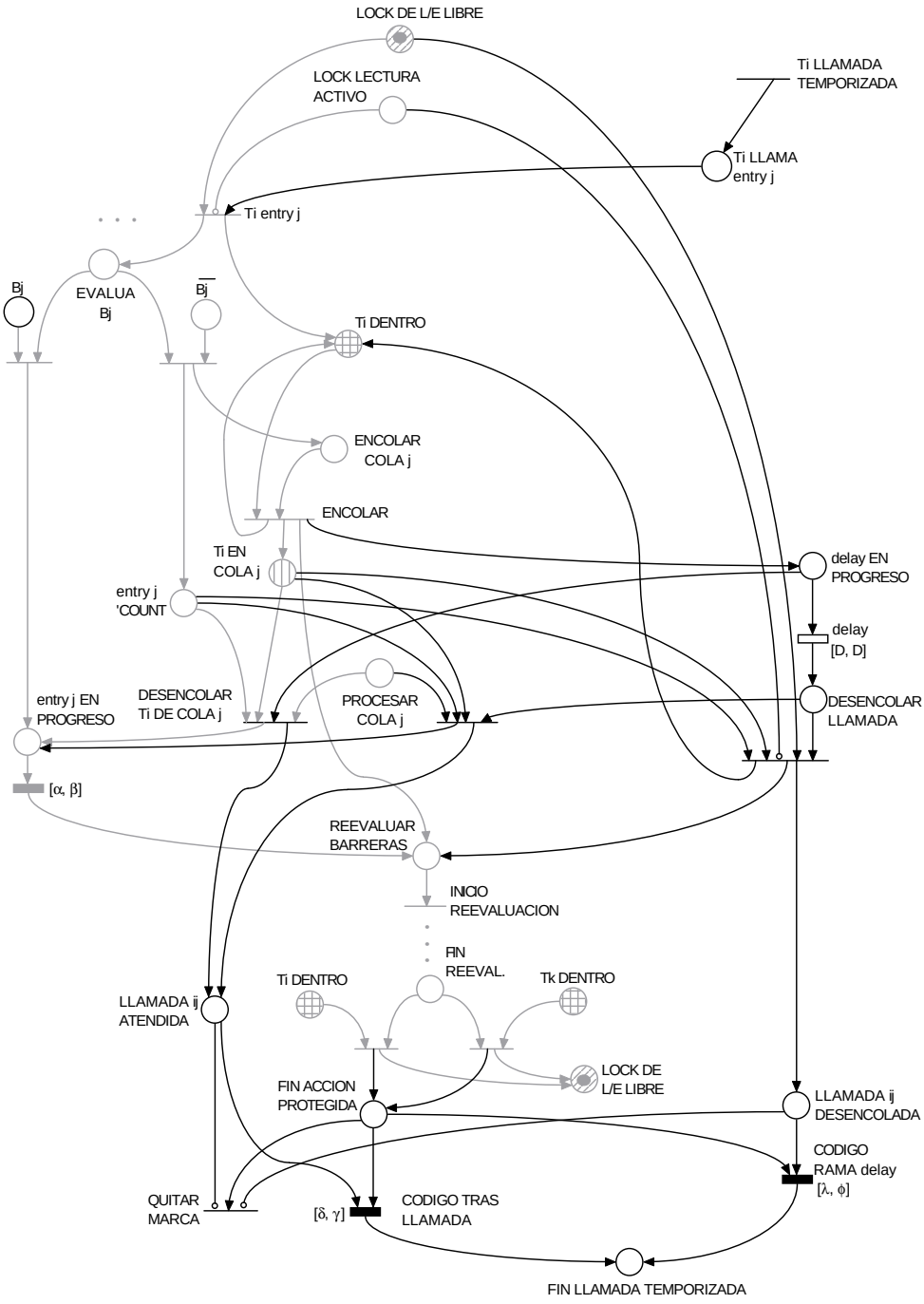


Figura 6.20: Modelo de la llamada temporizada a un punto de entrada de un objeto protegido

empieza a contar hasta el momento en que la llamada es encolada[8], lo cual es una acción protegida que requiere de la apropiación del lock de lectura/escritura. Del mismo modo, si el tiempo del `delay` expira antes de que la llamada sea atendida, ésta debe ser desencolada, lo que vuelve a ser una acción protegida, que, además, puede provocar la reevaluación de las barreras del resto de entradas y la ejecución de alguna llamada encolada. Por tanto, la ejecución del código alternativo puede verse retrasada hasta que se termine la reevaluación de las barreras y la atención de las llamadas encoladas. Incluso puede darse el caso de que venza el `delay`, que se trate de descolar la llamada, y que no se pueda porque el objeto protegido esté ocupado atendiendo una entrada que haya provocado una reevaluación de barreras que, a su vez, provoque la ejecución de la llamada que se trata de desencolar. Estaríamos ante el caso de que la llamada es atendida más tarde de lo que en principio marca el `delay`.

La red de la figura 6.20 describe este comportamiento. Cuando se produce la llamada, el lugar Ti llama a entry j es marcado. Cuando el lock de lectura/escritura está libre, la llamada se atiende de la forma descrita en el punto anterior. Si la barrera está cerrada y la llamada es encolada, al dispararse la transición encolar el lugar delay en progreso es marcado, comenzando a computarse el tiempo del `delay`. El encolado de la llamada produce, como ya se ha comentado, una reevaluación de barreras. Tras ello la tarea T_i abandona el objeto protegido liberando el lock (marcado lock de L/E libre y marcando el lugar fin acción protegida y disparando la transición quitar marca).

Si como resultado de una reevaluación de barreras provocada por alguna otra tarea, por ejemplo T_k, la llamada encolada es atendida antes de que la transición delay sea disparada, ésta transición es desensibilizada merced al disparo de la transición desencolar Ti de cola j, que retira la marca del lugar delay en progreso y, a su vez, marca el lugar llamada atendida. A partir de entonces la llamada prosigue como se ha descrito en la sección anterior, hasta que se alcanza el final de la acción protegida. Entonces, al estar marcado el lugar llamada atendida, la transición CODE código tras llamada es ejecutada. Esto representa la ejecución del código que aparece tras la llamada en la estructura `select` de la llamada temporizada. Tras esta ejecución la llamada temporizada termina.

Si la transición TIME delay es disparada antes de que el lugar Ti en cola j sea desmarcado, es decir, antes de que la llamada sea atendida, el lugar desencolar llamada es marcado. En este punto pueden ocurrir dos cosas

- Si el objeto protegido está libre (lock de L/E libre marcado y lock lectura activo desmarcado), la tarea llamadora puede acceder al interior del objeto protegido para desencolar la llamada (desmarcar Ti en cola j). Además el lugar llamada desencolada es marcado. El desencolado de la llamada puede provocar una reevaluación de barreras, que se produce como se ha descrito anteriormente. Una vez que la acción protegida así iniciada termina, el lugar fin acción protegida es marcado, lo que junto con el lugar llamada desencolada hace que la transición CODE código rama delay sea ejecutada. Tras ella la llamada temporizada termina.

- Si el objeto protegido no está libre, se deberá esperar a que lo esté antes de que la llamada pueda ser desencolada en los términos descritos anteriormente. Si como resultado de una reevaluación de barreras provocada por alguna otra tarea T_k, la llamada encolada es atendida, el intento de desencolar la llamada es cancelado. A partir de aquí se procede como en el caso en el que el `delay` es cancelado.

6.5.5. Llamada condicional a un punto de entrada de un objeto protegido

El modelo para esta estructura, que puede verse en la figura 6.21, es similar al del punto anterior. La llamada debe ser encolada para que la estructura comience. La única diferencia estriba en que el

[8] Este comportamiento tiene una importancia menor si se trata de un delay absoluto, pero puede retrasar mucho más de lo esperado la repuesta a la llamada si se trata de un delay relativo

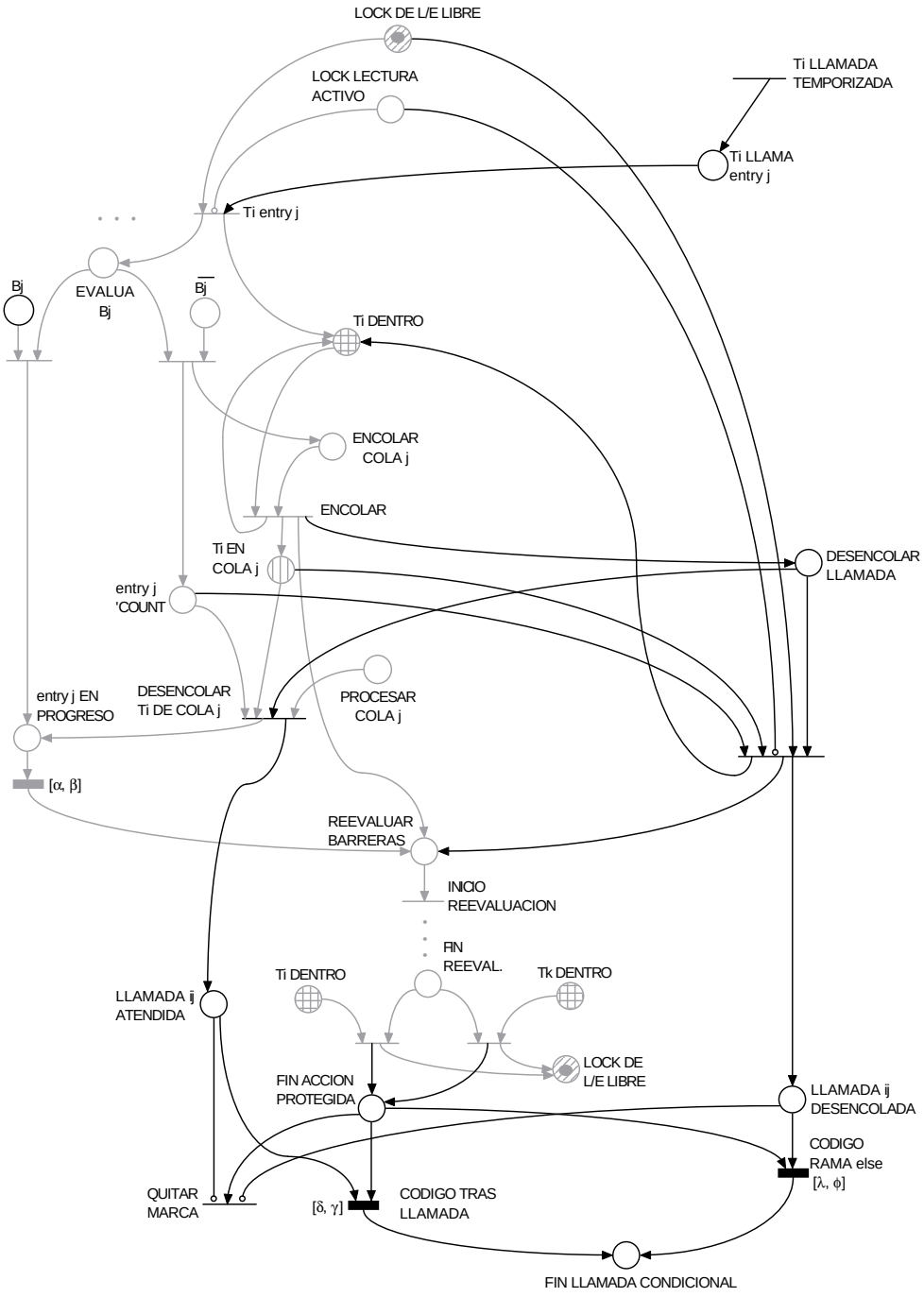


Figura 6.21: Modelo de la llamada condicional a un punto de entrada de un objeto protegido

intento de desencolado de la llamada es inmediato, en vez de esperar a que transcurra una cantidad de tiempo. Debido a las similitudes con el anterior no lo comentaremos más en detalle.

6.5.6. ATC con una llamada a un punto de entrada de un objeto protegido como alternativa de disparo

El modelo para esta estructura, que puede verse en la figura 6.22, y es una mezcla de las ideas expuestas en las secciones anteriores y el modelo de la figura 6.16. Cuando la alternativa de disparo es una llamada a un punto de entrada de un objeto protegido, la parte abortable no comienza hasta que la llamada no es encolada (ARM 9.7.4(6)), lo que es una acción protegida. El mecanismo es similar al del punto 6.5.4 pero al terminar la acción protegida debe plantearse qué acción ha sido la finalizada. Si ha terminado tras la ejecución de la llamada, se debe abortar la parte abortable, como en el punto 6.4.2. Si lo que se ha terminado es la acción protegida de desencolamiento de la llamada, provocada por la finalización de la parte abortable, se debe terminar el ATC. Debido a las similitudes con los anteriores modelos no lo comentaremos más en detalle.

6.6. Resumen de resultados y aportaciones del capítulo

El capítulo ha presentado modelos, construidos con RdPT, de las primitivas de concurrencia y tiempo real de Ada 95. La presentación de los modelos se ha realizado de forma progresiva, considerando en primer lugar la primitiva básica y añadiendo posteriormente elementos que completaran su semántica.

Entre las estructuras estudiadas se encuentran:

- Modelos de la instrucción `delay`, tanto relativo como absoluto. Se ha estudiado la adecuación del formalismo de RdPT para el modelado de estas primitivas, haciéndose notar que para el caso del delay absoluto es necesario el uso de un tipo de RdP con mayor poder expresivo, por ejemplo las redes TER [GMMP91].
- Modelos de la cita Ada 95, considerando varias tareas llamadoras, las colas asociadas a los puntos de entrada, y el efecto de las excepciones y del aborto de una cita.
- Modelos de la instrucción `select`, con varios puntos de entrada, que puedan estar "guardados", y las alternativas `delay`, `else` y `terminate`.
- Modelos de llamada temporizada y condicional a puntos de entrada de una tarea.
- Modelos de la estructura ATC, con sus dos posibles alternativas de disparo: un `delay` o una llamada a un punto de entrada de una tarea.
- Modelos de funciones, procedimientos y entradas de un objeto protegido, considerando el mecanismo de reevaluación de barreras.
- Modelo de ATC con una llamada a un punto de entrada de un objeto protegido como alternativa de disparo.
- Modelos de llamadas temporizadas y condicionales a puntos de entrada de un objeto protegido.

Al no haber sido utilizadas en el capítulo de implementación software, ha quedado fuera de este estudio la consideración de puntos de entrada privados y las instrucciones de reencolado de llamadas (`requeue` y `requeue with abort`).

Con los modelos aquí recogidos hemos cubierto el principal objetivo del capítulo: la construcción de estos modelos ha sido el primer paso para una justificación de la corrección de las estructuras de

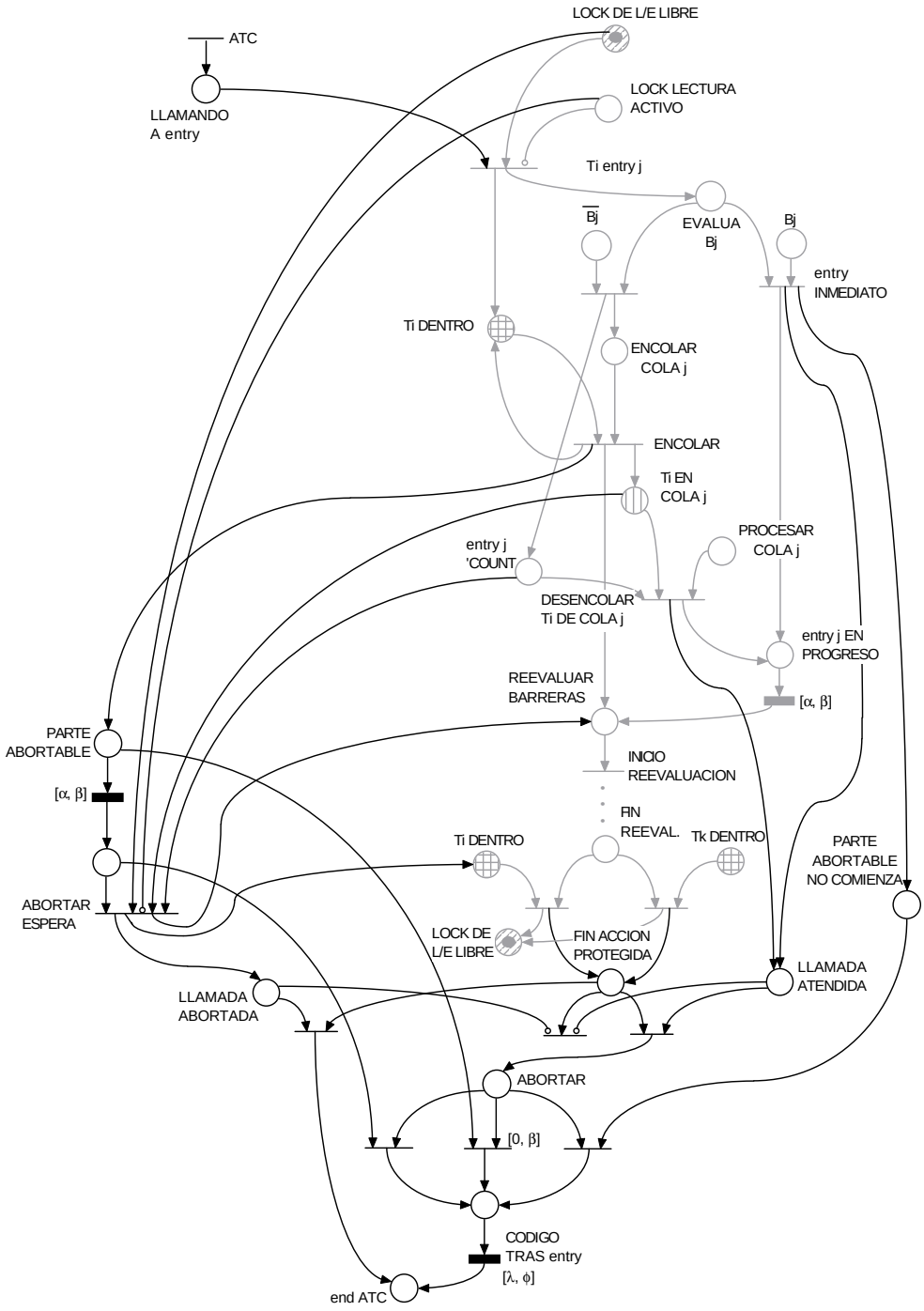


Figura 6.22: Modelo de una estructura ATC con una llamada a un punto de entrada de un objeto protegido como alternativa de disparo

implementación software descritas en el capítulo 5. Sin embargo, como avanzábamos en la introducción, el trabajo puede ser entendido como contribución en dos líneas de trabajo suplementarias:

1. Completar la definición de modelos que muestren con precisión y sin ambigüedad la semántica de las primitivas de concurrencia y tiempo real de Ada 95, completando y en su caso, y desde nuestro punto de vista, mejorando los trabajos conocidos hasta el momento ([MZGT85] y [GS97]).

2. Ayudar en la línea de trabajo propuesta en [STMD96], que persigue la generación de redes de Petri (*Ada nets*) que modelen programas Ada. De esta forma es posible el estudio de las propiedades de un programa a través del análisis de las propiedades de la red que lo modela. En este trabajo se presentan modelos, que previa reducción (como se ha hecho en el capítulo 5), pueden servir para la construcción de modelos de análisis para programas Ada 95 más complejos que los tratados en las referencia citada.

Capítulo 7

Resumen de resultados, conclusiones y líneas abiertas

Al comenzar la redacción de esta tesis se planteó como objetivo general el desarrollo de un método, basado en el empleo de una extensión temporal de las RdP, para la elaboración del software de control de sistemas de tiempo real (STR), que cubriera todas las fases del ciclo de vida de los mismos, desde la especificación de requisitos hasta la generación de código. Es el momento de recapitular y analizar qué es lo que se ha conseguido, y en qué se debe seguir trabajando.

Resumen de resultados

Antes de la elaboración de esta tesis ya se conocían trabajos, citados a lo largo del texto, que relacionaban las redes de Petri y el tiempo real, en su mayoría dedicados al análisis de las propiedades de este tipo de sistemas. También se contaba con trabajos sobre implementación de redes de Petri generalizadas. Lo novedoso en esta tesis es la convergencia de ambos mundos para conseguir una generación automática, o casi automática, de código para STR. Además se situa ésta dentro de un entorno uniforme que tiene a las RdP como formalismo base, que propone utilizarlas para el modelado, análisis e implementación del sistema objetivo.

Ya de partida reconocimos que la magnitud del trabajo a abordar era demasiado grande para una tesis doctoral, motivo por el cual se recortaron las pretensiones iniciales hasta centrarlas en las fases de modelado e implementación del sistema, en las cuales se contaba con mayor experiencia de partida.

Pese a ello, en el capítulo 2 (Introducción) se hizo un planteamiento general de las distintas etapas del desarrollo del software de control de un STR, con el fin de que el lector enmarcara la tesis en su entorno adecuado. Así mismo, se repasaron exhaustivamente los formalismos existentes de redes de Petri y sus extensiones temporales, razonando la elección de las *redes de Petri con tiempo* [BD91]. Elegida la herramienta, a continuación se presentaron los conceptos básicos sobre modelado de sistemas de tiempo real con RdPT, que se apoyan sobre las clásicas técnicas de *modelado* con RdP. Para redondear este capítulo de introducción, se esbozaron en él los métodos de planificación y de análisis basado en RdPT, de una manera superficial, pues hasta el momento el grado de desarrollo de las técnicas no permite mayores explicaciones. Sin embargo, en este punto se reconocen posibles e interesantes líneas de investigación futura. Para terminar, se resumieron una serie de conceptos necesarios sobre implementación de RdP, que han sido básicos para los siguientes capítulos.

Centradas las ideas, e introducidos los conceptos elementales sobre modelado, el resto de la tesis se ha dedicado a la generación de código, o *implementación* en terminología de RdP. Se han repasado, adaptado, ampliado y formalizado técnicas conocidas de implementación de RdP, y se

ha propuesto un método de "traducción"software para obtener el código Ada 95 que corresponde a la red.

En el capítulo 3 se adaptaron las técnicas tradicionales de implementación centralizada a las RdPT, afrontando la dificultad que la gestión del tiempo añade a los métodos conocidos. Se trató de técnicas interpretadas, basadas, en el uso de un proceso *coordinador* que gestionara la evolución de la red, la supervisara temporalmente y controlara la ejecución de un conjunto de procesos (*procesos-CODE*). Como consecuencia de esta arquitectura, las partes de control y operativa del sistema fueron separadas en la implementación. La primera correspondió al coordinador y la segunda a los procesos-CODE. En el capítulo se presentó el algoritmo general que ejecuta el coordinador en su *ciclo de tratamiento*, presentando dos particularizaciones clásicas: los métodos de los *lugares representantes marcados* y de las *transiciones sensibilizadas*, adaptados al caso de las RdPT. Todo ello condujo a un método de implementación muy simple y fácil de automatizar, ya que la implementación es tan sencilla como construir una estructura de datos representativa de la red y hacer que ésta sea ejecutada por el coordinador. Por ello, la técnica es muy adecuada en las fases de diseño y prototipado, en las que se necesita flexibilidad para realizar cambios. Sin embargo, no todo han sido ventajas, ya que la utilización de un único proceso coordinador responsable del control de la red introduce problemas que van en detrimento de las prestaciones de la implementación, sobre todo considerando que su campo de aplicación han de ser los sistemas de tiempo real. Entre los problemas detectados se encuentran la sobrecarga introducida por el coordinador; la pérdida de concurrencia en la evolución de la red; la reducida tolerancia a fallos; y el gran número de procesos en la implementación.

La solución a estos problemas pasó por abordar la implementación de manera descentralizada, sin utilizar ningún proceso coordinador, y haciendo que cada parte de la implementación fuera responsable de su propio control. Éste es el objetivo tradicional de las implementaciones descentralizadas, que fueron tratadas en los capítulos 4 y 5. La idea básica, que no era nueva [Vil90], consistió en el reconocimiento en la RdPT de una serie de procesos concurrentes (que nosotros llamamos *protoprocesos*), compuestos de un conjunto de *unidades de ejecución* en *exclusión mutua*, y comunicados a través de primitivas de comunicación síncrona o asíncrona. Se formalizó el método mediante el uso de *elementos de partición*, conjuntos de lugares en exclusión mutua, que cubrían un conjunto de transiciones también en exclusión mutua, y relajando las restricciones de vivacidad y 1-limitación que se impusieron en [Vil90]. Se sistematizó, por medio del uso de *p-invariantes monomarcados*, la construcción del flujo de control de los protoprocesos encontrados, lo cual fue clave para la posterior codificación en Ada 95. Así mismo se ampliaron los tipos de comunicación entre protoprocesos conocidos hasta entonces, presentando una nueva primitiva de comunicación síncrona resultante de compartir un conjunto de lugares y transiciones conexos. Se repasó el concepto de *tubería*, relacionándolo con el de p-invariante monomarcado. Y, sobre todo, se realizó un completo estudio del reconocimiento de procesos en redes con conflictos (de libre elección, simple o acoplado), estudiando la posibilidad de la distribución de las transiciones en conflicto entre distintos protoprocesos (en contra de lo que había sido la regla hasta el momento). Por último, se constató que las técnicas de implementación descentralizada existentes para RdP pueden ser aplicadas a las RdPT, aunque todavía deben ser ampliadas para contemplar las restricciones temporales aportadas por las últimas.

Tras la fase de reconocimiento, los protoprocesos deben ser codificados en Ada 95 (capítulo 5). La estrategia seguida se basó en concatenar instrucciones Ada 95 correspondientes a la configuración de unidades de ejecución que integran el protoproceso, utilizando como referencia el flujo de control calculado. Se propusieron estructuras de implementación para las unidades de ejecución básicas, así como para las situaciones de conflicto, realizándose una revisión sistemática de los casos que se pueden presentar. Se justificó, cuando la complejidad de la estructura generada no lo impidió, que la subred implementada y las primitivas Ada 95 propuestas para su codificación tenían el mismo comportamiento, lo que ha garantizado que la implementación ha sido correcta. Para ello nos ayudamos de RdPT que modelan las primitivas Ada 95 utilizadas (capítulo 6). De

especial interés es el hecho de que es posible automatizar la etapa de codificación del sistema a partir de la red que lo modela. Esto elimina los errores de codificación, reduce el tiempo de implantación y por lo tanto reduce el coste del sistema. Otra de las aportaciones del capítulo ha sido la proposición de técnicas de implementación mixta, para resolver situaciones en las que, debido a limitaciones del lenguaje de codificación (Ada 95), no ha sido posible realizar la implementación software con las técnicas descritas.

Además de todo ello, se puede considerar otra aportación más, que en principio no era objetivo de la tesis. Se han propuesto modelos, en términos de RdPT, de las primitivas de concurrencia y tiempo real de Ada 95. Esto se puede considerar una contribución en dos sentidos: ayuda a la clarificación de la semántica de las citadas primitivas, y puede ser utilizado en la generación de redes de Petri que permitan el análisis de programas Ada, como en la línea de trabajo propuesta en [STMD96].

Conclusiones

El uso de métodos formales en el desarrollo de un sistema de tiempo real redunda en el aumento de la fiabilidad del mismo, ya que permiten la verificación de requisitos funcionales y temporales, y la detección de errores en etapas tempranas del ciclo de vida. Ésta ha sido la motivación de esta tesis, en la cual, se ha propuesto un formalismo para todo el ciclo de vida de los sistemas de tiempo real: las redes de Petri con Tiempo. Además se ha tratado en profundidad su uso en las etapas de implementación.

Sobre su utilización podemos extraer las siguientes conclusiones globales:

1. Las RdPT permiten el modelado de multitud de situaciones propias de los sistemas de tiempo real, de manera sencilla, pues se trata de un formalismo fácil de entender debido a su naturaleza gráfica. La inclusión de estos modelos en una metodología permite abarcar todo el ciclo de vida de dichos sistemas.

2. Las RdPT permiten abordar de forma automática, o casi automática, la generación de código de la parte de control de un sistema de tiempo real, reduciendo, de este modo, los errores de la etapa de codificación. En este sentido se han presentado varias técnicas que permiten:

   - el prototipado del sistema, por medio de la utilización de la técnica de implementación centralizada, basada en la ejecución de un coordinador que interpreta la estructura de la red que modela el sistema. Realizar cambios en el código es tan sencillo como cambiar la estructura de datos interpretada;
   - el reconocimiento de las actividades (procesos) concurrentes que conforman el sistema, a través de la técnica de implementación descentralizada. Así mismo es posible determinar los mecanismos de comunicación entre los citados procesos, permitiéndose la elección de diferentes primitivas síncronas y asíncronas;
   - La generación de código Ada 95 para los procesos obtenidos de la etapa anterior.

   Todo ello para una clase muy grande de RdPT, sin algunas de las limitaciones de otros trabajos relacionados.

3. Las técnicas de codificación propuestas son correctas, al haberse justificado que el comportamiento de la primitiva Ada 95 propuesta y la estructura de red implementada por ella es el mismo.

4. Las mencionadas técnicas se han utilizado en un sistema real (véase el anexo F), donde se ha demostrado la utilidad de las mismas.

5. No menos importante ha sido la consecución de un objetivo paralelo, como lo ha sido mostrar que las RdPT permiten la especificación del comportamiento y la semántica de las primitivas de concurrencia y tiempo real de Ada 95 (véase capítulo 6).

Líneas de investigación abiertas

Básicamente las líneas de investigación abiertas están encaminadas hacia el estudio de las fases del ciclo de vida que no se han recogido en esta tesis, y a profundizar en algunos aspectos de las etapas cubiertas. Podemos destacar:

1. El estudio de técnicas de planificación basadas en redes de Petri, que puede consistir en la adaptación de técnicas existentes o desarrollo de otras nuevas. Ello permitirá el análisis de la planificabilidad y de las propiedades de sistemas de tiempo real a través del estudio de propiedades de las RdPT, profundizando en lo avanzado en el capítulo 2. Conviene estudiar la relación entre planificabilidad y exclusión mutua temporal (ver anexo A) de las transiciones de una RdPT.

2. El mismo concepto, exclusión mutua temporal, puede aplicarse para conseguir reducir el número de procesos de la implementación de una RdPT. Este estudio pasaría por encontrar técnicas para la determinación de este tipo de exclusión mutua. Otra línea de interés estriba en el estudio de la planificabilidad de sistemas de tiempo real y su relación con la exclusión mutua temporal de las transiciones de la RdPT que los modelen.

3. Estudio del modelado, análisis e implementación de técnicas de tolerancia a fallos, por ejemplo las excepciones, que no han sido consideradas en esta tesis.

4. Ampliación de las técnicas de modelado e implementación para la utilización de redes de alto nivel con tiempo (por ejemplo, redes TER [GMMP91]). Esto aumentará la potencia de la técnica, permitiendo el tratamiento de situaciones más complejas.

Apéndice A

Redes de Petri

A.1. Estructura

Definición A.1.1 (Red de Petri) *Una Red de Petri (RdP),* $\mathbf{N}$*, es una cuádrupla* $\mathbf{N} < P, T; Pre, Post >$*, donde:*

1. P *es un conjunto finito y no vacío de lugares*
2. T *es un conjunto finito y no vacío de transiciones*
3. $P \cap T = \emptyset$
4. $Pre : P \times T \rightarrow \mathbb{N}$ *es la función de incidencia previa, donde* $\mathbb{N}$ *es el conjunto de números naturales*
5. $Post : T \times P \rightarrow \mathbb{N}$ *es la función de incidencia posterior, donde* $\mathbb{N}$ *es el conjunto de números naturales*

Una red de Petri se representa gráficamente por un grafo bipartido orientado. Los lugares se representan por medio de círculos y las transiciones por medio de barras. Existe un *arco* que une el lugar p_i a la transición t_j, sii $Pre(p_i, t_j) \neq 0$. Análogamente, existe un arco que une la transición t_k al lugar p_i, sii $Post(t_k, p_i) \neq 0$. Cada arco se etiqueta con un número natural, $Pre(p, t)$ o $Post(t, p)$ que se denomina *peso del arco*. Por convenio, un arco no etiquetado posee peso unitario.

Definición A.1.2 (Red de Petri ordinaria) *Una RdP* $\mathbf{N}$ *es ordinaria si sus funciones de incidencia pueden tomar únicamente valores 0 y 1. Es decir:* $\forall t \in T$, $\forall p \in P$:

$$\begin{cases} Pre(p,t) & \in & \{0,1\} \\ Post(t,p) & \in & \{0,1\} \end{cases}$$

Definición A.1.3 *Sea* $\mathbf{N}$ *una RdP,* $t \in T$ *y* $p \in P$*. Se definen los siguientes conjuntos:*

1. *Conjunto de lugares de entrada a* t*:* ${}^\bullet t = \{p \in P \mid Pre(p,t) > 0\}$
2. *Conjunto de lugares de salida de* t*:* $t^\bullet = \{p \in P \mid Post(t,p) > 0\}$
3. *Conjunto de transiciones de entrada a* p*:* ${}^\bullet p = \{t \in T \mid Post(t,p) > 0\}$
4. *Conjunto de transiciones de salida de* p*:* $p^\bullet = \{t \in T \mid Pre(p,t) > 0\}$

Definición A.1.4 (Representación matricial) *Una RdP* **N** *se encuentra definida matricialmente por medio de dos matrices. Sea* $n = |P|$ *(número de lugares de* P*) y* $m = |T|$ *(número de transiciones de* T*). Se denomina*

1. *matriz de incidencia previa a la matriz* $C^- = [c_{ij}^-]_{n \times m}$ *en la que* $c_{ij}^- = Pre(p_i, t_j)$.
2. *matriz de incidencia posterior a la matriz* $C^+ = [c_{ij}^+]_{n \times m}$ *en la que* $c_{ij}^+ = Post(t_j, p_i)$.
3. *matriz de incidencia de* **N** *a:* $C = C^+ - C^-$.

Es decir, en las matrices de incidencia las filas representan a los lugares y las columnas a las transiciones.

Definición A.1.5 (Red pura) *Una RdP* **N** *es una red pura si no existe ninguna transición que tenga un lugar que sea al mismo tiempo de entrada y salida de la transición:*

$$\forall t_j \in T \, \forall p_i \in P, \; Pre(p_i, t_j) Post(t_j, p_i) = 0$$

Definición A.1.6 (Grafo de estados) *Un grafo de estado (GE) o máquina de estados (ME) es una RdP que cumple:*

$$\forall t \in T, \; |{}^\bullet t| = |t^\bullet| = 1$$

Definición A.1.7 (Grafo de marcados) *Un grafo de marcados (GM) o grafo de sincronización es una RdP que cumple:*

$$\forall p \in P, \; |{}^\bullet p| = |p^\bullet| = 1$$

Definición A.1.8 (Red libre elección) *Una RdP libre elección (RLE) es una RdP que cumple:*

$$\forall p \in P, \;\; si \; |p^\bullet| > 1, \;\; entonces \; \forall t \in p^\bullet, |{}^\bullet t| = 1$$

Definición A.1.9 (Red simple) *Una RdP simple (RS) es una RdP que cumple que toda transición tiene como máximo un lugar de entrada compartido con otras transiciones:*

$$\forall t_i, t_j \in T, \; i \neq j, \; |{}^\bullet t_i \cap {}^\bullet t_j| \leq 1$$

A.2. Marcado y comportamiento

Definición A.2.1 (Marcado) *El marcado* M *de una RdP de una red* **N** *es una aplicación:*

$$M : P \rightarrow \mathbb{N}$$

Es decir el marcado M asigna un número natural a cada lugar de la red. En el grafo asociado a **N**, este número ($M(p)$) se indica por medio de $M(p)$ objetos denominados *marcas*, que son representadas por puntos dentro de los lugares.

Definición A.2.2 (Red de Petri marcada) *Una RdP marcada es el par* $< \mathbf{N}, M_0 >$*, donde* **N** *es una RdP y* M_0 *el marcado inicial de la red.*

Definición A.2.3 (Transición sensibilizada) *Una transición* $t \in T$ *está sensibilizada para un marcado* M *dado, sii* $\forall p \in {}^\bullet t$ *se verifica* $M(p) \geq Pre(p, t)$.

Definición A.2.4 (Grado de sensibilización de una transición) *El grado de sensibilización de una transición indica el número máximo de veces que la transición puede ser disparada concurrentemente. El grado de sensibilización de una transición $t \in T$ para un marcado M dado, viene dado por la expresión*

$$E_M(t) = min_{p \in {}^\bullet t} \lfloor \frac{M(p)}{Pre(p,t)} \rfloor$$

Definición A.2.5 (Regla de evolución del marcado) *Si t está sensibilizada para un marcado M entonces t puede disparar. En la operación se alcanza un nuevo marcado M', y se denota por $M[t > M'$, resultante de quitar $Pre(p,t)$ marcas de cada lugar $p \in {}^\bullet t$ y añadir $Post(t,p)$ marcas a cada lugar $p \in t^\bullet$. El disparo viene descrito por la ecuación:*

$$M'(p) = M(p) - Pre(p,t) + Post(t,p), \forall p \in P$$

Definición A.2.6 (Secuencia de disparos) *Una secuencia finita de transiciones, $\sigma = t_1, t_2, ..., t_k$ es una secuencia de disparos de la RdP marcada $< \mathbf{N}, M_0 >$, sii existe una secuencia $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} ... \xrightarrow{t_k} M_k$ tal que $\forall i, i : 1...k : M_{i-1}[t_i > M_i$. Se dice que el marcado M_k es un marcado alcanzable de M_0 mediante el disparo de $\sigma : M_0[\sigma > M_k$.*

Definición A.2.7 (Vector característico) *Se llama vector característico asociado a una secuencia de disparos σ, al vector $\overrightarrow{\sigma} \in \mathbb{N}^m (|T| = m)$, cuya i-ésima componente es el número de ocurrencias de la transición t_i en la secuencia σ.*

Definición A.2.8 (Lenguaje de disparos) *El conjunto de secuencias disparables desde un marcado inicial M_0 es un lenguaje:*

$$L(\mathbf{N}, M_0) = \{\sigma | M_0[\sigma > M\}$$

Definición A.2.9 (Conjunto de marcados alcanzables) *El conjunto $R(\mathbf{N}, M_0) = \{M | \exists \sigma \in L(\mathbf{N}, M_0) \wedge M_0[\sigma > M\}$ se denomina conjunto de marcados alcanzables a partir de M_0.*

Definición A.2.10 (Ecuación de estado) *Sea $< \mathbf{N}, M_0 >$ una RdP marcada. Sea $\sigma \in L(\mathbf{N}, M_0)$ una secuencia de disparos y $M_0[\sigma > M$. Es fácil de obtener la regla de disparo (ecuación de estado) que rige el disparo de las transiciones de la red:*

$$M = M_0 + C\overrightarrow{\sigma}$$

A.3. Propiedades funcionales

Definición A.3.1 (Transición viva) *Sea $< \mathbf{N}, M_0 >$ una RdP marcada, $t \in T$ es viva para el marcado inicial M_0 sii para cualquier marcado alcanzable M existe una secuencia de disparos que incluye a t:*

$$\forall M \in R(\mathbf{N}, M_0)\ \exists \sigma : M[\sigma > M', M' \in R(\mathbf{N}, M_0) \wedge t \in \sigma$$

Definición A.3.2 (Red viva) *Una RdP marcada $< \mathbf{N}, M_0 >$ es viva para el marcado inicial M_0 sii todas sus transiciones son vivas.*

Definición A.3.3 (Lugar k-limitado) *Sea $< \mathbf{N}, M_0 >$ una RdP marcada, $p \in P$ es k-limitado para el marcado inicial M_0 sii $\forall M \in R(\mathbf{N}, M_0) \exists k \in \mathbb{N}$ tal que $M(p) \leq k$.*

Definición A.3.4 (Red k-limitada) *Una RdP marcada $< \mathbf{N}, M_0 >$ es k-limitada para M_0 sii todos sus lugares son k-limitados para M_0: $\forall p \in P$ y $\forall M \in R(\mathbf{N}, M_0)\ \exists k \in \mathbb{N}$ tal que $M(p) \leq k$.*

Definición A.3.5 (Red binaria) *Una RdP* $\mathbf{N}$ *es binaria (safe) sii* $\forall p \in P$ *y* $\forall M \in R(\mathbf{N}, M_0)$, $M(p) \leq 1$.

Definición A.3.6 (Conflicto estructural) *Se dice que en una RdP* $\mathbf{N}$ *existe un conflicto estructural cuando* $\exists p \in P$ *tal que* $p^\bullet > 1$.

Definición A.3.7 (Conflicto efectivo) *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada. Sean* $t_i, t_j \in T$ *dos transiciones de la red. Se dice que existe un conflicto efectivo entre* t_i, t_j *para* M_0 *sii:*

1. $\exists M \in R(\mathbf{N}, M_0)$ *que sensibiliza a* t_i, t_j
2. *al disparar* t_i (t_j) *el marcado obtenido no sensibiliza la transición* t_j (t_i).

Definición A.3.8 (Conflicto acoplado) *Sea* $\mathbf{N}$ *una RdP. Sean* $t_i, t_j \in T$ *dos transiciones de la red. Se dice que* t_i, t_j *están en relación de conflicto acoplado,* $t_i \, \mathcal{CA} \, t_j$*, sii:* ${}^\bullet t_i \cap {}^\bullet t_j \neq \emptyset$ *ó* $\exists t_k \in T$ *t.q.* $t_i \, \mathcal{CA} \, t_k \wedge t_k \, \mathcal{CA} \, t_j$.

Definición A.3.9 (Lugares en exclusión mutua) *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada. Sean* $p_i, p_j \in P$ *dos lugares de la red. Se dice que* p_i, p_j *están en exclusión mutua para* M_0*, y se denota* $p_i \, \mathcal{EM} \, p_j$*, sii no pueden estar marcados simultáneamente en los marcados alcanzables a partir de* M_0*:* $\nexists M \in R(\mathbf{N}, M_0)$ *tal que* $M(p_i) > 0 \wedge M(p_j) > 0$.

Definición A.3.10 (Transiciones en exclusión mutua) *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada. Sean* $t_i, t_j \in T$ *dos transiciones de la red. Se dice que* t_i, t_j *están en exclusión mutua para* M_0*, y se denota* $t_i \, \mathcal{EM} \, t_j$*, sii ocurre alguna de las siguientes situaciones:*

1. $\nexists M \in R(\mathbf{N}, M_0)$ *tal que* t_i, t_j *estén sensibilizadas a la vez, o*
2. $\exists M \in R(\mathbf{N}, M_0)$ *que sensibiliza a* t_i, t_j*, pero al disparar* t_i (t_j) *el marcado obtenido no sensibiliza la transición* t_j (t_i).

Proposición A.3.1 *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada. Sean* $t_i, t_j \in T$ *dos transiciones de la red. Si existe un conflicto efectivo entre* $t_i, t_j \Rightarrow t_i \, \mathcal{EM} \, t_j$.

La demostración se deduce directamente de la definición de conflicto efectivo. Obsérvese que lo contrario no es cierto.

Definición A.3.11 (P-semiflujo) *Sea* $\mathbf{N} < P, T; Pre, Post >$ *una RdP. Sea* C *su matriz de incidencia y* $n = |P|$*. Un vector* $Y \in \mathbb{N}^n, Y \neq 0$ *es un p-semiflujo de* $\mathbf{N}$ *sii* $Y^T C = 0$.

Definición A.3.12 (Soporte de un p-semiflujo) *Se denomina soporte de un p-semiflujo* Y*, y se denota* $\|Y\|$ *al conjunto de lugares asociados a los elementos no nulos de* Y.

Proposición A.3.2 *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada e* $Y \in \mathbb{N}^n$ *un P-semiflujo de* $\mathbf{N}$*. Entonces* $\forall M \in R(\mathbf{N}, M_0) : Y^T M = Y^T M_0$.

Esta expresión se denomina *invariante (estructural) de marcado*, y si $Y^T M_0 = 1$, *invariante monomarcado*. Para un p-invariante monomarcado se cumple $\forall M \in R(\mathbf{N}, M_0) : \sum_{p_i \in \|Y\|} M(p_i) = 1$.

Definición A.3.13 (Red conservativa) *Sea* $\mathbf{N} < P, T; Pre, Post >$ *una RdP.* $\mathbf{N}$ *es conservativa sii existe un P-semiflujo* $Y \in \mathbb{N}^n$ *tal que* $\|Y\| = P$.

Una red es conservativa si tiene un invariante de marcado global, $Y^T M_0 = P^T M_0$.

Definición A.3.14 (Red ampliada) *La RdP* $\mathbf{N}_p =< P', T; Pre', Post' >$ *se denomina red ampliada de* $\mathbf{N} < P, T; Pre, Post >$ *mediante el lugar* p *sii* $P' = P \cup \{p\}, p \notin P$, *y* $Pre' : P' \times T \rightarrow \mathbb{N}$ *y* $Post' : T \times P' \rightarrow \mathbb{N}$.

Definición A.3.15 (Lugar comportamentalmente implícito) *Sea* $< \mathbf{N}, M_0 >$ *una RdP marcada y* $< \mathbf{N}_p, M_{0p} >$ *su red ampliada con* p. *El lugar* $p \in P'$ *en implícito (comportamentalmente) en* $< \mathbf{N}_p, M_{0p} >$ *sii* $L(\mathbf{N}_p, M_{0p}) = L(\mathbf{N}, M_0)$.

Esto es, la adición de p preserva el lenguaje.

Definición A.3.16 (Lugar estructuralmente implícito) *Sea* $\mathbf{N}_p$ *una RdP. El lugar* p *es estructuralmente implícito en* $\mathbf{N}_p$ *sii existe un vector* $Y \in Q^n, Y \geq 0$ *tal que* $Y^T \cdot C \leq l_p$, *siendo* l_p *la línea de la matriz de incidencia correspondiente al lugar* p.

Teorema A.3.1 *Un lugar* p *es estructuralmente implícito en* $\mathbf{N}_p$ *sii existe* M_p *tal que* p *es implícito (comportamentalmente) en* $< \mathbf{N}_p, M_p >$.

Definición A.3.17 (Lugar complementario) *Sea* $\mathbf{N}$ *una RdP y* $p \in P$ *un lugar de la red. El lugar denotado por* $\overline{p}$ *se dice complementario de* p *sii* $C[p] = -C[\overline{p}]$ *(esto es, los arcos de salida de* p *pasan a ser los de entrada de* $\overline{p}$ *y los arcos de entrada de* p *pasan a ser los de salida de* $\overline{p}$*).*

A.4. Redes de Petri con Tiempo

Esta sección da una definición formal de las redes de Petri con Tiempo según se definen en [BD91].

Definición A.4.1 (Red de Petri con Tiempo) *Una Red de Petri con Tiempo (RdPT),* $\mathbf{N}$, *es una tupla* $\mathbf{N} < P, T; Pre, Post; M_0; CIS >$, *donde:*

1. P *es un conjunto finito y no vacío de lugares*
2. T *es un conjunto finito y no vacío de transiciones*
3. $P \cap T = \emptyset$
4. $Pre : P \times T \rightarrow \mathbb{N}$ *es la función de incidencia previa, donde* $\mathbb{N}$ *es el conjunto de números naturales*
5. $Post : T \times P \rightarrow \mathbb{N}$ *es la función de incidencia posterior, donde* $\mathbb{N}$ *es el conjunto de números naturales*
6. M_0 *es la función de marcado inicial* $M_0 : P \rightarrow \mathbb{N}$
7. CIS *es una correspondencia de intervalos estáticos,* $CIS : T \rightarrow Q^+ \times (Q^+ \cup \infty)$, *donde* Q^+ *es el conjunto de los números racionales positivos junto con el cero* [1].

La última función asocia a cada transición un par $(CIS(t_i) = (\alpha_i, \beta_i))$, *que define un intervalo temporal, por lo que se debe verificar:* $0 \leq \alpha_i < \infty$, $0 \leq \beta_i \leq \infty$, *y* $\alpha_i \leq \beta_i$ *si* $\beta_i \neq \infty$ *ó* $\alpha_i < \beta_i$ *si* $\beta_i = \infty$.

[1] En la elección de $\mathbb{Q}^+$ radica la diferencia entre la definición propuesta por Merlin en [Mer74, MF76] y la de [BD91]. En la propuesta de Merlin la función CIS se define sobre los $\mathbb{R}^+$. En nuestro caso hemos elegido los $\mathbb{Q}^+$ ya que nuestro propósito último es la implementación en Ada 95, y en ella utilizaremos el tipo de datos `TIME`, que es racional, no real

Definición A.4.2 (Regla de disparo) *Este intervalo permite definir la regla de disparo de una transición en una RdPT: suponiendo que la transición t_i comienza a estar sensibilizada en el instante τ_i, y que continúa sensibilizada, el disparo de la transición se producirá no antes del instante $\tau_i + \alpha_i$, y no más tarde del instante $\tau_i + \beta_i$. El intervalo de tiempos de disparo válidos para t_i será, por tanto, $(\tau_i + \alpha_i, \tau_i + \beta_i)$. La semántica de este disparo es del tipo de tiempo de sensibilización, por lo que las marcas permanecen en los lugares de entrada durante el tiempo necesario, y una vez que se produce el disparo, éste no consume tiempo: es instantáneo.*

Definición A.4.3 *Sea* $\mathbf{N} < P, T; Pre, Post; M_0; CIS >$ *una RdPT,* $\mathbf{N}$*. Se llama:*

- Intervalo estático de disparo *al intervalo* (α_i, β_i).
- Instante de disparo más cercano *(*earliest firing time*, EFT) al valor* α_i.
- Instante de disparo más lejano *(*latest firing time*, LFT) al valor* β_i.

Definición A.4.4 (Estado) *El estado de una RdPT se define como un par* $S = (M, I)$*, donde:*

- M *es el marcado*
- I *es un conjunto de intervalos de disparo, que será un vector de todos los posibles intervalos de disparo de todas las transiciones sensibilizadas por el marcado* M*. El tamaño del vector podrá cambiar dependiendo del marcado de la red, ya que el número de transiciones sensibilizadas en uno u otro marcado pueden cambiar. Se supondrá que los intervalos almacenados en* I *están ordenados respecto del índice de la transición a la que corresponden.*

Definición A.4.5 (Exclusión mutua temporal) *Sea* $\mathbf{N} < P, T; Pre, Post; M_0; CIS >$ *una RdPT. Sean* $t_i, t_j \in T$ *dos transiciones de la red. Se dice que* t_i, t_j *están en exclusión mutua temporal para* M_0*, y se denota* $t_i \, \mathcal{EMT} \, t_j$*, sii ocurre alguna de las siguientes situaciones:*

1. *las transiciones están en exclusión mutua en la red de Petri subyacente (sin considerar el tiempo), o*
2. $\nexists \, S(M, I)$ *estado alcanzable de la red que sensibiliza a* t_i, t_j*, y que cumpla que* $I(t_i) \cap I(t_j) \neq \emptyset$ *(es decir no existe ningún instante en el que ambas transiciones puedan ser disparadas simultáneamente)*
3. $\exists \, S(M, I)$ *que sensibiliza a* t_i, t_j*, y que cumple que* $I(t_i) \cap I(t_j) \neq \emptyset$*, pero al disparar* t_i (t_j) *el marcado obtenido no sensibiliza la transición* t_j (t_i)*.*

Apéndice B

Algoritmo de obtención de clases máximas de compatibles

Las implementaciones descentralizadas se basan en el cálculo de conjuntos de transiciones en exclusión mutua (EM). Esta propiedad es una relación de compatibilidad, ya que cumple las propiedades reflexiva y simétrica: una transición está en exclusión mutua consigo misma (recordar que consideramos para su implementación redes binarias) y dadas $t_i, t_j \in T$ si t_i EM $t_j \Rightarrow t_j EM t_i$. Por tanto el cálculo de conjuntos de transiciones en EM se transforma en el cálculo de clases de compatibilidad de transiciones en EM. Como se pretende encontrar una partición de la red con el mínimo número de procesos, las clases de compatibilidad de transiciones deberán ser máximas.

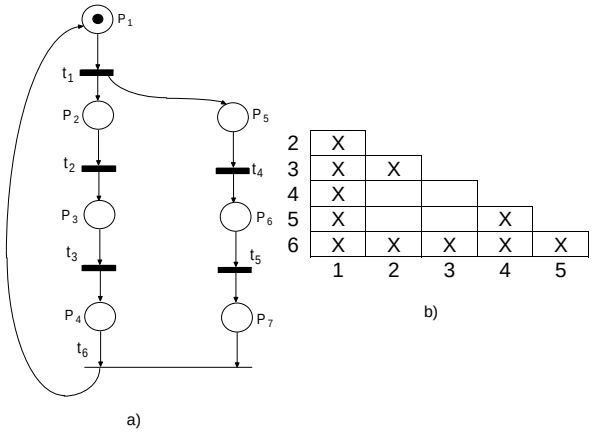


Figura B.1: a) Red de Petri y b) su tabla de compatibilidad de transiciones en exclusión mutua

El algoritmo está basado en [Sil85]. Para facilitar el cálculo de los compatibles máximos se utiliza una representación en forma de tabla que se denomina *tabla de compatibilidad*. Se parte de un conjunto de N elementos a los que nos referiremos por sus índices (de 1 a N). Debido al cumplimiento de las propiedades reflexiva y simétrica las dimensiones de la tabla serán $N(N-1)/2$ casillas. No es necesario mostrar las compatibilidades de i con i, ni de j con i si ya se muestra la de i con j. En la tabla de compatibilidad se indica si los pares de elementos del conjunto son o no compatibles. El valor del elemento ij de la tabla indica la compatibilidad entre los elementos i y j del conjunto de elementos. Aplicado al caso del conjunto de transiciones en exclusión mutua,

la casilla ij indica que la transición t_i está en exclusión mutua con t_j. Considérese por ejemplo el caso de la figura B.1.

Sea $\mathcal{C}_i$ el subconjunto de elementos del conjunto definido por las casillas no nulas de la columna i de la tabla de compatibilidad. Así, en el ejemplo $\mathcal{C}_2 = \{3, 6\}$, que representa a $\{t_3, t_6\}$. Sea k el mayor índice de elemento cuya columna en la tabla contenga al menos un par compatible. En el ejemplo $k = 5$. El algoritmo para el cálculo de compatibles máximos es el siguiente:

(1) Crear una lista ($\mathcal{L}$) con los pares de elementos compatibles definidos por la columna k (la que está más a la derecha con al menos un par compatible)
(2) **Para** $i := k - 1$ **hasta** 1
    Si $\mathcal{C}_i \neq \emptyset$ **entonces**
        2.1 Realizar la intersección de $\mathcal{C}_i$ con los elementos de $\mathcal{L}$
        2.2 Añadir a $\mathcal{L}$ los compatibles formados por el resultado de las intersecciones no nulas más el elemento i, así como los pares de compatibles definidos por la columna i-ésima
        2.3 Suprimir de $\mathcal{L}$ todos los compatibles contenidos en otros
    fsi
    fpara
(3) Añadir a $\mathcal{L}$ todos los elementos del conjunto de partida que no están contenidos en ningún elemento de $\mathcal{L}$.

La aplicación del anterior algoritmo a la tabla del ejemplo da como resultado los siguientes compatibles: $\mathcal{L} = \{\{1, 2, 3, 6\}, \{1, 4, 5, 6\}\}$.

Apéndice C

Reglas de reducción de redes de Petri y redes de Petri con tiempo

Tradicionalmente, el propósito de las reglas de reducción ha sido disminuir el tamaño de un modelo construido con una red de Petri para facilitar su análisis. Esto se consigue eliminando determinados lugares y/o transiciones que resultan ser superfluos. Durante el proceso de reducción, se ha de garantizar que la propiedad que se desea estudiar no se vea afectada por la reducción. Por ejemplo, si queremos comprobar que un lugar es k-limitado, la reducción no podrá eliminar el lugar bajo estudio. La trivialidad del ejemplo anterior no es tal cuando se desean mantener propiedades más complejas como vivacidad, 1-limitación, u otras. Más aún, en el caso de las redes de Petri con tiempo, además de las citadas, se han de preservar las propiedades temporales.

En nuestro caso, utilizaremos reglas de reducción para lograr demostrar que la implementación software de determinadas subredes del modelo original es correcta, constatando que las primitivas Ada 95 propuestas para su implementación, tiene la misma semántica que la subred, tal como se muestra en el capítulo 5. Para nosotros, serán principios a seguir y propiedades a preservar:

- No eliminar ninguna de las transiciones TIME o CODE, ya que en ellas se refleja la parte operativa y de supervisión temporal del sistema.
- No introducir estados no presentes en la dinámica propia de la red
- Preservar el lenguaje de disparo de las transiciones TIME y SYCO, de forma que las situaciones en las que estas transiciones están sensibilizadas y son disparadas no se vean afectadas.
- Preservar las propiedades de exclusión mutua en el acceso a recursos comunes, modelados mediante lugares. Estos lugares, tampoco deben ser eliminados

Enumeramos seguidamente las reglas más habituales. En todas ellas la red reducida tiene el mismo lenguaje de disparos, restringido a las transiciones no eliminadas, y se mantienen las propiedades temporales y la vivacidad. Un esquema que ilustra la aplicación de las reglas se muestra al final de este anexo.

Regla C.1 (Prefusión de transiciones) [Ber85, STMD96] (ver figura C.1)

Condiciones de aplicación Sean:

- Un lugar $p \in P$
- Una transición $t_e \in T$ de entrada a p
- Un conjunto de transiciones $\mathcal{S} \subset T$, $\mathcal{S} \neq \emptyset$, de salida de p

1. $t_e^\bullet = \{p\}$, y $p \notin {}^\bullet t_e$, p es el único lugar de salida de t_e y no pertenece a su entrada.
2. $\forall t_s \in \mathcal{S},\ p \in {}^\bullet t_s \wedge p \notin t_s^\bullet$, toda transición de $\mathcal{S}$ tiene a p como entrada pero ninguna lo tiene como salida
3. $\forall t \notin (\mathcal{S} \cup t_e),\ p \notin t^\bullet \wedge p \notin {}^\bullet t$, p sólo se conecta a t_e y a las transiciones de S.
4. $\forall p' \in {}^\bullet t_e,\ |p'^\bullet| = 1$, t_e no comparte sus lugares de entrada con ningún otro lugar
5. $M_0(p) = 0$

Regla Fundir t_e con toda transición $t_s \in \mathcal{S}$, eliminando el lugar p, y poner como entrada de las t_s los lugares ${}^\bullet t_e \cup {}^\bullet t_s$

Regla C.2 (Prefusión de transiciones en redes de Petri con tiempo) [SB96] (ver figura C.1)

Condiciones de aplicación Sean:

- Un lugar $p \in P$
- Una transición $t_e \in T$ de entrada a p
- Un conjunto de transiciones $\mathcal{S} \subset T$, $\mathcal{S} \neq \emptyset$, de salida de p

1. La red subyacente es binaria
2. $t_e^\bullet = \{p\}$, y $p \notin {}^\bullet t_e$, p es el único lugar de salida de t_e y no pertenece a su entrada.
3. $\forall t_s \in \mathcal{S},\ p \in {}^\bullet t_s \wedge p \notin t_s^\bullet$, toda transición de $\mathcal{S}$ tiene a p como entrada pero ninguna lo tiene como salida
4. $\forall t \notin (\mathcal{S} \cup t_e),\ p \notin t^\bullet \wedge p \notin {}^\bullet t$, p sólo se conecta a t_e y a las transiciones de S.
5. $\forall p' \in {}^\bullet t_e,\ |p'^\bullet| = 1$, t_e no comparte sus lugares de entrada con ningún otro lugar
6. $M_0(p) = 0$
7. El intervalo temporal de t_e es puntual, $\mathcal{F}(t_e) = [d,\ d]$
8. El intervalo temporal de t_e debe ser $\mathcal{F}(t_e) = [0,\ 0]$ o si es $[d,\ d]$ con $d > 0$, entonces se debe cumplir $\forall t_s \in \mathcal{S},\ \forall q \in {}^\bullet t_s,\ q \neq p$, se tiene un *nice-tree*(t_s, q, d), siendo un *nice-tree*(t_s, q, d):
   - $q^\bullet = \{t_s\}$, las transiciones de $\mathcal{S}$ no comparten ningún lugar de entrada
   - $({}^\bullet q)^\bullet = \{q\}$, el lugar no comparte ninguna de sus transiciones de entrada
   - Para cada $t' \in {}^\bullet q$, o bien $d \leq EFT(t')$, o para cada $q' \in {}^\bullet t'$ tenemos un *nice-tree*$(t', q', d - EFT(t'))$.

   o, lo que es lo mismo, para cada rama de este árbol que acaba en t_s se verifica que la suma de los EFT de las transiciones debe ser $\geq d$.

Regla

1. Fundir t_e con toda transición $t_s \in \mathcal{S}$, eliminando p, y poner como entrada de las t_s los lugares ${}^\bullet t_e \cup {}^\bullet t_s$
2. Si $\mathcal{F}(t_e) \neq [0,\ 0]$ añadir $[d,\ d]$ a todos los intervalos temporales de todas las $t_s \in \mathcal{S}$ y ajustar los tiempos de las transiciones del *nice-tree*(t_s, q, d) : $\forall t \in {}^{\bullet\bullet}\mathcal{S}$ restar $m = min\{EFT(t),\ d\}$ de $\mathcal{F}(t)$. Si $m < d$, $\forall t' \in {}^{\bullet\bullet}t$ restar $m' = min\{EFT(t'),\ d - m\}$ de $\mathcal{F}(t')$. Continuar así hasta alcanzar las hojas del *nice-tree* o se haya restado toda la cantidad d.

Regla C.3 (Postfusión de transiciones) [Ber85, STMD96] (ver figura C.1)

Condiciones de aplicación Sean:

- Un lugar $p \in P$
- Un conjunto de transiciones $\mathcal{E} \subset T$, $\mathcal{E} \neq \emptyset$, de entrada de p
- Un conjunto de transiciones $\mathcal{S} \subset T$, $\mathcal{S} \neq \emptyset$, de salida de p

1. $\forall t_s \in \mathcal{S},\ {}^\bullet t_s = \{p\} \wedge p \notin t_s^\bullet$, la única entrada de toda transición de $\mathcal{S}$ es p, pero ninguna lo tiene como salida
2. $\forall t_e \in \mathcal{E},\ p \in t_e^\bullet \wedge p \notin {}^\bullet t_e$, toda transición de $\mathcal{E}$ tiene a p como salida pero ninguna lo tiene como entrada
3. $\forall t \notin (\mathcal{E} \cup \mathcal{S}),\ p \notin t^\bullet \wedge p \notin {}^\bullet t$, p sólo se conecta a las transiciones de $\mathcal{E}$ y a las de $\mathcal{S}$.
4. $M_0(p) = 0$

Regla Fundir cada $t_e \in \mathcal{E}$ con toda transición $t_s \in \mathcal{S}$, eliminando p y formando un conjunto de transiciones t_{es} de modo que ${}^\bullet t_{es} = {}^\bullet t_e$ y $t_{es}^\bullet = t_s^\bullet \cup (t_e^\bullet - \{p\})$

Regla C.4 (Postfusión de transiciones en redes de Petri con tiempo) [SB96] (ver figura C.1)

Condiciones de aplicación Sean:

- Un lugar $p \in P$
- Un conjunto de transiciones $\mathcal{E} \subset T$, $\mathcal{E} \neq \emptyset$, de entrada de p
- Un conjunto de transiciones $\mathcal{S} \subset T$, $\mathcal{S} \neq \emptyset$, de salida de p

1. La red subyacente es binaria
2. $\forall t_s \in \mathcal{S},\ {}^\bullet t_s = \{p\} \wedge p \notin t_s^\bullet$, la única entrada de toda transición de $\mathcal{S}$ es p, pero ninguna lo tiene como salida
3. $\forall t_e \in \mathcal{E},\ p \in t_e^\bullet \wedge p \notin {}^\bullet t_e$, toda transición de $\mathcal{E}$ tiene a p como salida pero ninguna lo tiene como entrada
4. $\forall t \notin (\mathcal{E} \cup \mathcal{S}),\ p \notin t^\bullet \wedge p \notin {}^\bullet t$, p sólo se conecta a las transiciones de $\mathcal{E}$ y a las de $\mathcal{S}$.
5. $M_0(p) = 0$
6. $\forall t_s \in \mathcal{S},\ \mathcal{F}(t_s) = [0, 0]$

Regla Fundir cada $t_e \in \mathcal{E}$ con toda transición $t_s \in \mathcal{S}$, eliminando p y formando un conjunto de transiciones t_{es} de modo que ${}^\bullet t_{es} = {}^\bullet t_e$ y $t_{es}^\bullet = t_s^\bullet \cup (t_e^\bullet - \{p\})$ y $\mathcal{F}(t_{es}) = \mathcal{F}(t_e)$

Regla C.5 (Fusión serie de transiciones) [Ber85, STMD96] (ver figura C.2)

Condiciones de aplicación Sean:

- Un lugar $p \in P$
- Una transición $t_e \in T$ de entrada a p
- Una transición $t_s \in T$ de salida de p

1. $t_e^\bullet = \{p\}$, y $p \notin {}^\bullet t_e$, p es el único lugar de salida de t_e y no pertenece a su entrada.
2. ${}^\bullet t_s = \{p\}$ y $p \notin t_s^\bullet$, p es el único lugar de entrada de t_s y no pertenece a su salida.
3. $|{}^\bullet p| = 1 = |p^\bullet|$, p no tiene más transiciones de entrada ni de salida.
4. $M_0(p) = 0$

Regla Eliminar la estructura t_e - p - t_s y sustituirla por una transición t_{es} de modo que ${}^\bullet t_{es} = {}^\bullet t_e$ y $t_{es}^\bullet = t_s^\bullet$.

Regla C.6 (Fusión serie de transiciones en redes de Petri con tiempo) [SB96] (ver figura C.2)

Condiciones de aplicación Sean:

- Un lugar $p \in P$
- Una transición $t_e \in T$ de entrada a p
- Una transición $t_s \in T$ de salida de p

1. La red subyacente es binaria
2. $t_e^\bullet = \{p\}$, y $p \notin {}^\bullet t_e$, p es el único lugar de salida de t_e y no pertenece a su entrada.
3. ${}^\bullet t_s = \{p\}$ y $p \notin {}^\bullet t_s$, p es el único lugar de entrada de t_s y no pertenece a su salida.
4. $|{}^\bullet p| = 1 = |p^\bullet|$, p no tiene más transiciones de entrada ni de salida.
5. $M_0(p) = 0$
6. O bien $({}^\bullet t_e)^\bullet = \{t_e\}$, es decir, t_e no comparte ninguno de sus lugares de entrada, o el intervalo temporal de t_s es $[0, 0]$, $\mathcal{F}(t_s) = [0, 0]$

Regla Eliminar la estructura t_e - p - t_s y sustituirla por una transición t_{es} de modo que ${}^\bullet t_{es} = {}^\bullet t_e$ y $t_{es}^\bullet = t_s^\bullet$ y $\mathcal{F}(t_{es}) = \mathcal{F}(t_e) + \mathcal{F}(t_s)$.

Regla C.7 (Fusión lateral de transiciones en redes de Petri) [Ber85] (ver figura C.2)

Condiciones de aplicación Sean:

- Un conjunto de transiciones $\mathcal{S} \subset T$, $\mathcal{S} \neq \emptyset$
- Una transición $t_c \notin \mathcal{S}$
- Un lugar p_{ei} distinto para cada $t_i \in \mathcal{S}$

Las transiciones de $\mathcal{S}$ son lateralmente fusionables en una transición t_f si

1. $\forall p_{ei},\ {}^\bullet p_{ei} = \{t_c\}$, la única transición de entrada de los lugares p_{ei} es t_c
2. $\forall p_{ei},\ p_{ei}^\bullet = \{t_i\}$, los lugares no tiene más salida que su transición t_i correspondiente
3. $\forall t_i \in \mathcal{S},\ {}^\bullet t_i = \{p_{ei}\}$, las transiciones de $\mathcal{S}$ no tienen más lugar de entrada que su correspondiente p_{ei}
4. $\forall p_{ei}, p_{ej},\ M_0(p_{ei}) = M_0(p_{ej})$, todos los lugares p_{ei} tienen el mismo marcado inicial

Regla Cada transición de $\mathcal{S}$ es reemplazada por una única transición t_f, que tiene como lugares de entrada y salida la unión de los correspondientes a las transiciones de $\mathcal{S}$.

Regla C.8 (Fusión lateral de transiciones en redes de Petri con tiempo) [SB96] (ver figura C.2)

Condiciones de aplicación Sean:

- Un conjunto de transiciones $\mathcal{S} \subset T$, $\mathcal{S} \neq \emptyset$
- Una transición $t_c \notin \mathcal{S}$
- Un lugar p_{ei} distinto para cada $t_i \in \mathcal{S}$

Las transiciones de $\mathcal{S}$ son lateralmente fusionables en una transición t_f si

1. La red subyacente es binaria
2. $\forall p_{ei},\ {}^\bullet p_{ei} = \{t_c\}$, la única transición de entrada de los lugares p_{ei} es t_c
3. $\forall p_{ei},\ p_{ei}^\bullet = \{t_i\}$, los lugares no tiene más salida que su transición t_i correspondiente
4. $\forall t_i \in \mathcal{S},\ {}^\bullet t_i = \{p_{ei}\}$, las transiciones de $\mathcal{S}$ no tienen más lugar de entrada que su correspondiente p_{ei}
5. $\forall p_{ei}, p_{ej},\ M_0(p_{ei}) = M_0(p_{ej})$, todos los lugares p_{ei} tienen el mismo marcado inicial
6. Todas las transiciones de $\mathcal{S}$ tienen el mismo intervalo temporal, que además es de la forma $\mathcal{F}(t_i) = [a, a]$

Regla Cada transición de $\mathcal{S}$ es reemplazada por una única transición t_f, que tiene el mismo intervalo temporal que las de $\mathcal{S}$. Los lugares de entrada y salida de t_f son la unión de los correspondientes a las transiciones de $\mathcal{S}$.

Regla C.9 (Nodos paralelos redundantes) [Mur89, STMD96, SB96] (ver figura C.2)

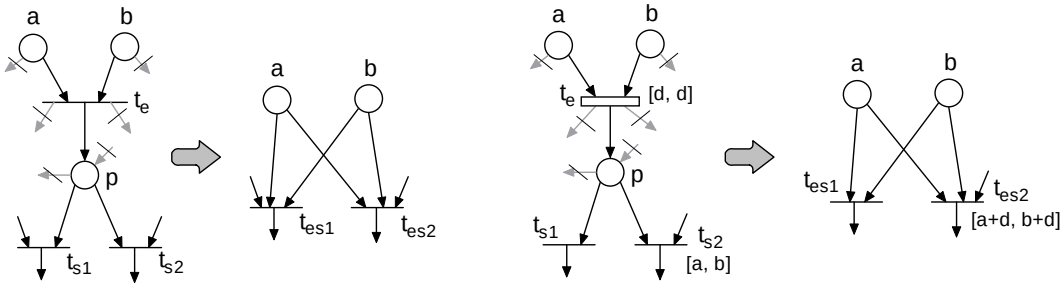
Condiciones de aplicación Sean dos nodos (transiciones o lugares desmarcados) de la red que tienen idénticas entradas y salidas (posiblemente vacías),

Regla Uno de los nodos puede ser eliminado.

Regla C.10 (Lugares implícitos) [Ber85, Sil85, Col89, Mur89]

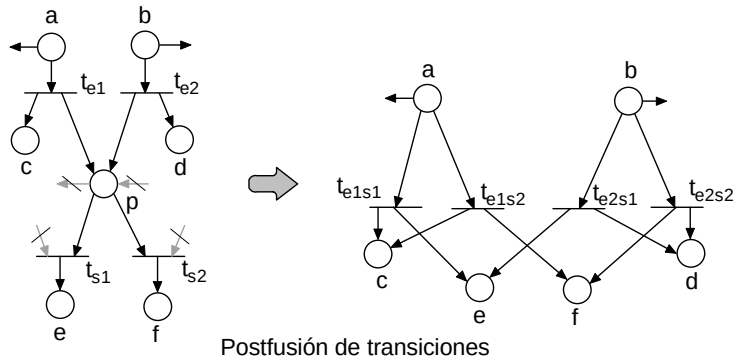
Condiciones de aplicación En las referencias propuestas pueden encontrarse varios conjuntos de condiciones para la determinación de lugares implícitos.

Regla Los lugares implícitos pueden ser eliminados de las redes de Petri. Como un lugar implícito no puede ser el único responsable de que una transición no esté sensibilizada, en la red de Petri con tiempo un lugar implícito no añadirá ningún estado nuevo, no hará que se sensibilice ni desensibilice ninguna transición. Por ello tampoco influirá en las clases de estados de las redes de Petri con tiempo, y, en consecuencia también podrá ser eliminado de las redes de Petri con tiempo.

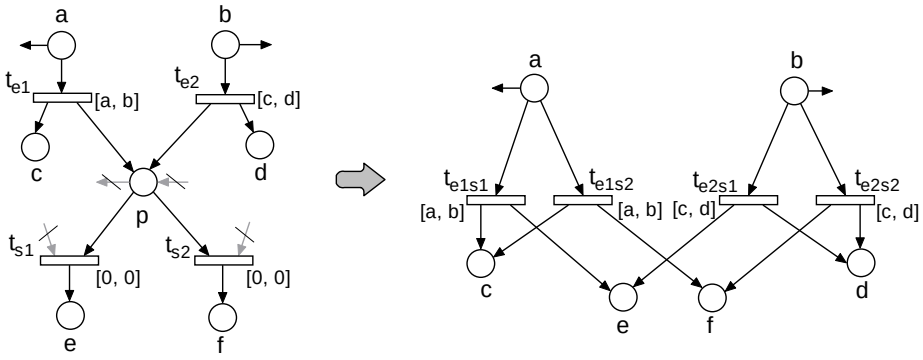


Prefusión de transiciones

Prefusión de transiciones con tiempo



Postfusión de transiciones



Postfusión de transiciones con tiempo

Figura C.1: Reglas de reducción de redes de Petri y redes de Petri con tiempo

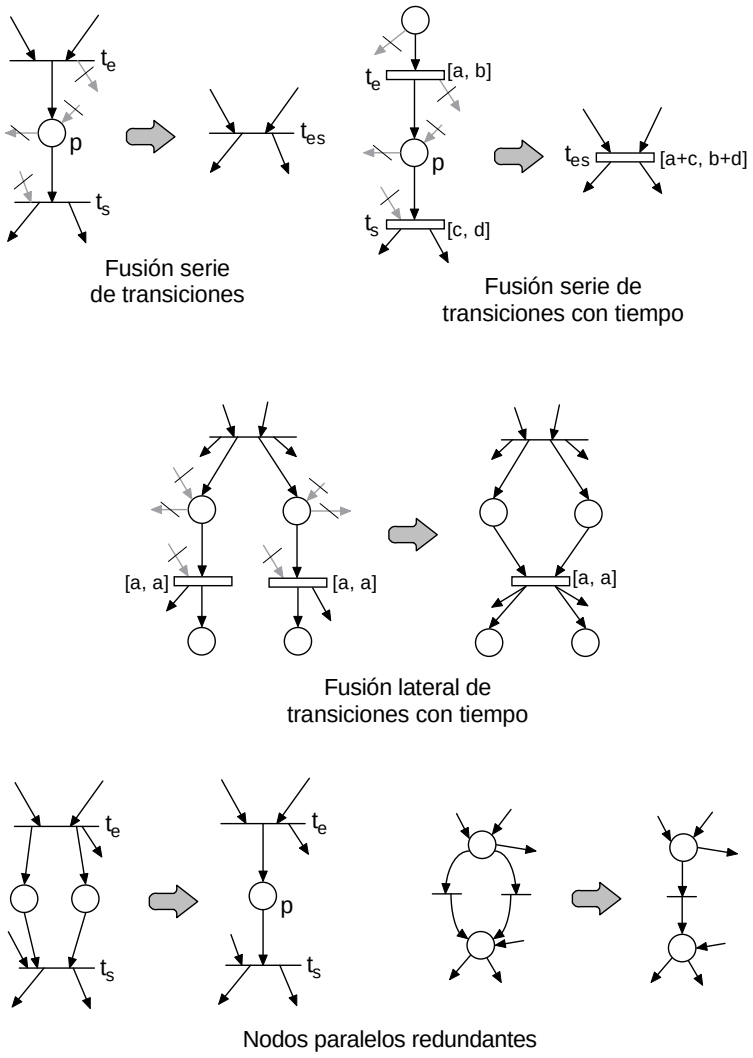


Figura C.2: Reglas de reducción de redes de Petri y redes de Petri con tiempo (II)

Apéndice D

Ejemplo de implementación mixta

Consideraremos para su implementación software la red de la figura D.1.

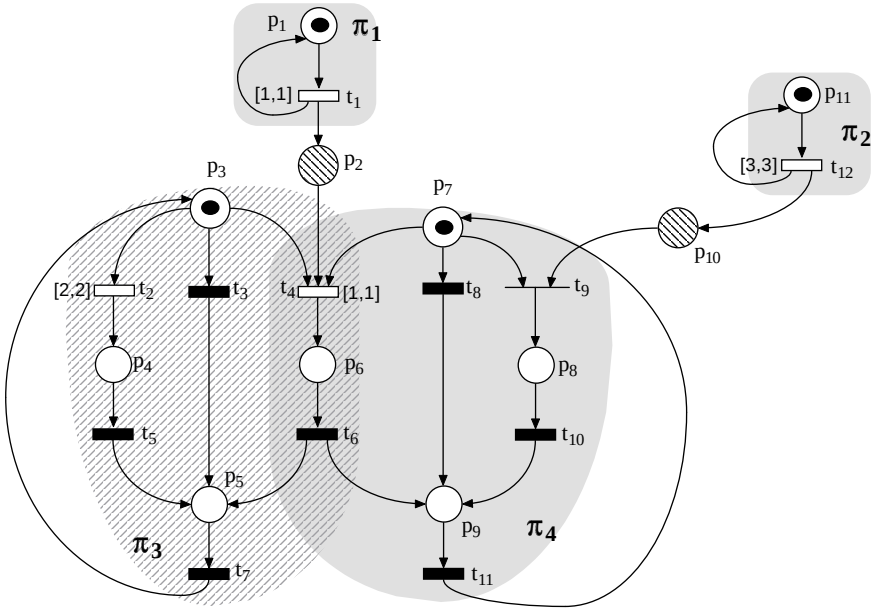


Figura D.1: Red de Petri para su implementación

Mediante un cálculo de p-invariantes monomarcados, se han determinado cuatro protoprocesos en la red, los cuales se muestran destacados en la figura. En este caso, las transiciones del conflicto son de tipo CODE, TIME y SYCO, y una de ellas tiene más de un lugar de entrada que no pertenece al protoproceso que la contiene. Es el caso de la transición t_4, para cuya implementación sería necesario, tal como se indica en la sección 5.7, forzar la distribución de las transiciones en conflicto mediante la adición de lugares implícitos. Sin embargo, en esta red no es posible añadir lugares implícitos para formar elementos de partición que no incluyan lugares comunes, por lo que la implementación no sería posible.

Así mismo, la transición t_4 está modelando una cita a tres, pues además de estar compartida entre π_3 y π_4, desmarca un lugar de comunicación asíncrona (p_2). En este caso la transición debería ser asignada a uno de los dos protoprocesos, y cerrar el otro con algún lugar implícito. Como se razonó en el párrafo anterior, esto no es posible en este caso.

Por tanto, la única opción para implementar la red es recurrir a una implementación mixta, tal como se indica en la sección 5.10. El gestor de conflicto deberá encapsular los lugares p_2, p_3, p_7 y p_{10}. Además dispondrá de las entradas adecuadas para *marcar* lugares, para aceptar el *inicio de la ejecución* del código en conflicto (familia de dos entradas, una para cada proceso descendente, π_3 y π_4), para aceptar la *finalización de la ejecución* del código en conflicto (familia de dos entradas) y, cuando se produzca el disparo de una de las transiciones, para *desmarcar* lugares del gestor de conflicto (familia de dos entradas), lo que supondrá el aborto del resto de transiciones no disparadas.

La implementación del gestor de conflicto que se muestra a continuación es una adaptación para la red en estudio de la arquitectura de un coordinador centralizado.

```
  TT1  : constant Duration := 1.0;
  TT2  : constant Duration := 2.0;
  TT4  : constant Duration := 1.0;
  TT12 : constant Duration := 3.0;

  type Transiciones is (t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12);
  type Transiciones_salida is (t2, t3, t4, t8, t9);
  type Transiciones_entrada is (t1, t7, t11, t12);
  type Procesos_descendentes is (Proceso3, Proceso4);
  type Lugares_gestor is (p2, p3, p7, p10);

task Gestor_Conflicto is
  entry Desmarcar(Procesos_descendentes) (t: out Transiciones_salida);
  entry Marcar(t: in Transiciones_entrada);
  entry Comienza_codigo(Procesos_descendentes);
  entry Finaliza_codigo(Procesos_descendentes);
  pragma Priority(25);
end;

task body Gestor_Conflicto is
  TIME_a_disparar : Transiciones_salida;
  TIME_TT2, TIME_TT4: TIME;
  CT3_Activo, CT8_Activo, TT2_Activo, TT4_Activo: boolean := FALSE;
  Marcado: array (Lugares_gestor) of ...
  function TIME_mas_Cercano return TIME is
  begin
    -- devuelve el tiempo de vencimiento del evento activo más urgente
    -- Hace las veces de lista de eventos temporales
  end;
begin
 loop
  -- Disparo transiciones SYCO
  if Marcado(p7) and Marcado(p10) then
    accept Desmarcar(Proceso4)(t: out Transiciones_salida) do
      t := t9;
      Decrementar(p7); Decrementar(p10); TT4_Activo := FALSE; CT8_Activo := FALSE;
    end;
  end if;


  -- Disparo transición TIME
```

```
if Marcado(p3) then
  if Marcado(p7) and Marcado(p2) then
    if not TT4_Activo then
      TT4_Activo := TRUE; TIME_TT4 := CLOCK + To_Time_Span(TT4);
    end if;
  end if;
  if not TT2_Activo then
    TT2_Activo := TRUE; TIME_TT2 := CLOCK + To_Time_Span(TT2);
  end if;
end if;

-- Disparo transición CODE
if Marcado(p3) and not CT3_Activo then
  accept Comienza_codigo(Proceso3); CT3_Activo := TRUE;
end if;
if Marcado(p7) and not CT8_Activo then
  accept Comienza_codigo(Proceso4); CT8_Activo := TRUE;
end if;

loop
  select
    accept Marcar (t: in Transiciones_entrada) do
      case t is
        when t1  => Incrementar(p2);
        when t7  => Incrementar(p3);
        when t11 => Incrementar(p7);
        when t12 => Incrementar(p10);
        when others => null;
      end case;
    end;
  or
    accept Finaliza_codigo(Proceso3) do
      Decrementar(p3); TT2_Activo := FALSE;TT4_Activo := FALSE;CT3_Activo := FALSE;
    end;
    exit;
  or
    accept Finaliza_codigo(Proceso4) do
      Decrementar(p7); TT4_Activo := FALSE;CT8_Activo := FALSE;
    end;
    exit;
  or
    when TT4_Activo or TT2_Activo =>
      delay until TIME_mas_cercano;
      case TIME_a_disparar is
        when t2 => accept Desmarcar(Proceso3)(t: out Transiciones_salida) do
                     t := t2;
                     Decrementar(p3);
                     TT4_Activo := FALSE; TT2_Activo := FALSE; CT3_Activo := FALSE;
                   end;
        when t4 => accept Desmarcar(Proceso4)(t: out Transiciones_salida) do
                     t := t4;
```

```
                        Decrementar(p3); Decrementar(p7); Decrementar(p2);
                        TT4_Activo := FALSE; CT8_Activo := FALSE;
                        TT2_Activo := FALSE; CT3_Activo := FALSE;
                      end;
                      accept Desmarcar(Proceso3)(t: out Transiciones_salida) do
                        t := t4;
                      end;
          when others => null;
        end case;
        exit;
    end select;
    exit when Marcar'COUNT = 0;
  end loop;
 end loop;
end;
```

La implementación de los procesos Π_1 y Π_2 se recoge a continuación. Se trata de procesos que periódicamente marcan los lugares de comunicación p_2 y p_{10}.

```
  task Proc_1 is
    pragma Priority(20);
  end;
  task Proc_2 is
    pragma Priority(5);
  end;
  task body Proc_1 is
    Last: TIME := CLOCK;
  begin
    loop
      delay until Last + To_Time_Span(TT1);
      Last := Last + To_Time_Span(TT1);
      Gestor_Conflicto.Marcar(t1);
    end loop;
  end;
  task body Proc_2 is
    Last: TIME := CLOCK;
  begin
    loop
      delay until Last + To_Time_Span(TT12);
      Last := Last + To_Time_Span(TT12);
      Gestor_Conflicto.Marcar(t12);
    end loop;
  end;
```

Resta la implementación software de los procesos Π_3 y Π_4, que implementan el conflicto acoplado. En cada proceso puede reconocerse la estructura ATC que se mencionó en la sección 5.10. Llamamos la atención sobre la forma de implementar la subred compartida t_4 - p_6 - t_6. Si la transición t_6 es disparada, se debe abortar la ejecución del código de los dos procesos que la comparten. Tras ello los procesos deben ejecutar en común el código de t_6, por lo que se deberá mantener una cita entre ambos.

```
  task Proc_3 is
```

```
    entry t4;
    pragma Priority(15);
  end;
  task body Proc_3 is
    t_aborta: Transiciones_salida;
    P4, P6: boolean := FALSE;
  begin
    loop
      select
        Gestor_Conflicto.Desmarcar(Proceso3)(t_aborta);
        case t_aborta is
          when t2 => P4 := TRUE; P6:=FALSE;
          when t4 => P6 := TRUE; P4:=FALSE;
          when others => null;
        end case;
      then abort
        Gestor_Conflicto.Comienza_Codigo(Proceso3);
        Codigo(t3);
        Gestor_Conflicto.Finaliza_Codigo(Proceso3);
      end select;
      if P4 then Codigo(t5);end if;
      if P6 then
        accept t4 do
          Codigo(t6);
        end;
      end if;
      Codigo(t7);
      Gestor_Conflicto.Marcar(t7);
    end loop;
  end;

  task Proc_4 is
    pragma Priority(10);
  end;
  task body Proc_4 is
    t_aborta: Transiciones_salida;
    P8, P6: boolean;
  begin
    loop
      select
        Gestor_Conflicto.Desmarcar(Proceso4)(t_aborta);
        case t_aborta is
          when t9 => P8 := TRUE; P6:=FALSE;
          when t4 => P6 := TRUE; P8:=FALSE;
          when others => null;
        end case;
      then abort
        Gestor_Conflicto.Comienza_Codigo(Proceso4);
        Codigo(t8);
        Gestor_Conflicto.Finaliza_Codigo(Proceso4);
      end select;
```

```
    if P8 then Codigo(t10);end if;
    if P6 then Proc_3.t4;end if;
    Codigo(t11);
    Gestor_Conflicto.Marcar(t11);
  end loop;
end;
```

Apéndice E

Implementación de un lugar de comunicación mediante un objeto protegido

Este anexo corresponde a la justificación de la proposición 5.3.1, del capítulo 5 (sección 5.3.1):

La implementación de un lugar de comunicación de destino único, o cuyo destino son transiciones SYCO, será de la siguiente manera:

```
protected Lugar_Comunicacion_Simple is
  entry Desmarcar;
  procedure Marcar;
private
  Marcado: integer := ...; -- Valor inicial de marcado
end Lugar_Comunicacion_Simple;

protected body Lugar_Comunicacion_Simple is
  entry Desmarcar when Marcado > 0 is
  begin
    Marcado := Marcado - 1;
  end;
  procedure Marcar is
  begin
    Marcado := Marcado + 1;
  end;
end Lugar_Comunicacion_Simple;
```

JUSTIFICACIÓN

Supondremos, por simplicidad, que el lugar puede ser marcado por un único protoproceso, y desmarcado por otro único protoproceso. La generalización al caso de varios protoprocesos no implica mayor complejidad que la de la explosión exponencial de estados.

Para la justificación partiremos de la red de la figura E.1, que modela el objeto protegido propuesto en una situación general: dos protoproceso esquemáticos (T1 y T2), que tratan de desmarcar (T1) y marcar (T2) el lugar de comunicación. Del comportamiento de los protoprocesos se ha eliminado cualquier otra actividad no relacionada con estas dos que afectan al estado del lugar

de comunicación, que es lo que nos interesa estudiar. El modelo se ha construido particularizando los modelos generales para objetos protegidos que se muestran en el capítulo 6.

Trataremos de justificar que la red de la figura E.1, que modela el objeto protegido anterior en la situación desccrita, es equivalente, en términos de tener el mismo comportamiento observable, al de la figura E.2, en el que se muestra la típica configuración de lugar de comunicación que podemos encontrar en las RdPT objeto de nuestro estudio.

En el proceso de justificación recurriremos a comparar los grafos de estados de ambas redes (figuras E.3 y E.4). No se trata de grafos de estados habituales, sino que en ellos los nodos representan varios estados simples. Los estados están parametrizados dependiendo del valor del número de marcas del lugar de comunicación (LC). Hemos determinado que los grafos así construidos son más compactos e igualmente representativos del comportamiento dinámico que aquellos que mostrasen los estados reales. Los arcos muestran cómo se modifica el valor del parámetro n en la transición de uno a otro grupo de estados.

El significado de los acrónimos que nominan los lugares y transiciones en las figuras es el siguiente:

- LL, lugar lock libre,
- $T1_D$, lugar T1 quiere desmarcar,
- $T2_M$, lugar T2 quiere marcar,
- $T1_F$, lugar T1 final desmarcar,
- $T2_F$, lugar T2 final marcar,
- $T1_C$, lugar T1 en cola,
- LC, lugar lugar de comunicación,
- EB, lugar evalúa barrera,
- PC, lugar procesa cola,
- D, lugar desmarcar,
- t_1INI, transición T1 inicia desmarcar,
- t_1D, transición T1 va a desmarcar,
- t_1, transición T1 desmarca,
- t_1ENC, transición T1 va a ser encolada,
- t'_1, transición T1 vuelve,
- t_2, transición T2 marca,
- t_2NPC, transición T2 no procesa cola,
- t_2PC, transición T2 procesa cola,
- t'_2, transición T2 vuelve

Lo que nos interesa comprobar es que el comportamiento observable del modelo de implementación (figura E.1) con respecto al lugar LC, es el mismo que el del lugar de comunicación (figura E.2). Es decir que las transiciones entre estados del modelo de E.2, se producen de igual manera entre los estados equivalentes del modelo de E.1. Como puede apreciarse por inspección, cualquier camino que una estados en el grafo E.3 modifica el número de marcas del lugar LC (aumentándolo, $n = n + 1$, o disminuyéndolo, $n = n - 1$) en la misma mediada que el camino equivalente en el grafo E.4. Por ejemplo, cualquier camino entre el estado (t_1D, LL, t_2M, LC^n) al estado (t_1F, LL, t_2F, LC^n), aumenta en uno el número de marcas ($n = n + 1$) y lo disminuye ($n = n - 1$).

Para facilitar aún más el seguimiento de los caminos en el grafo, en las figuras E.5 a E.12 se muestran una serie de reducciones y simplificaciones [1] que eliminan los nodos del grafo E.3 que no aparecen en el E.4. Finalmente en la figura E.12 pueden apreciarse los caminos simplificados, y como, en todos ellos, la variación de marcas de LC (valor de n) es el mismo que en el grafo E.4.

[1] El proceso elimina estados o arcos que unen estados, de acuerdo al siguiente procedimiento (los estados o arcos que van a ser eliminados están dibujados con líneas punteadas):

- Los estados eliminados son sustituidos por arcos que enlazan los estados entre los que estaba el eliminado, etiquetándolos con la secuencia de transiciones que corresponden a los eliminados. Este es el caso de los pasos 1, 2, 3, 5 y 7.
- Los arcos son eliminados si representan un "atajo"entre los estados que unen, es decir, si existe otro camino entre estos estados, compuesto por varios arcos, que represente la misma secuencia de transiciones que el que se quiere eliminar. Es el caso de los pasos 4 y 6.

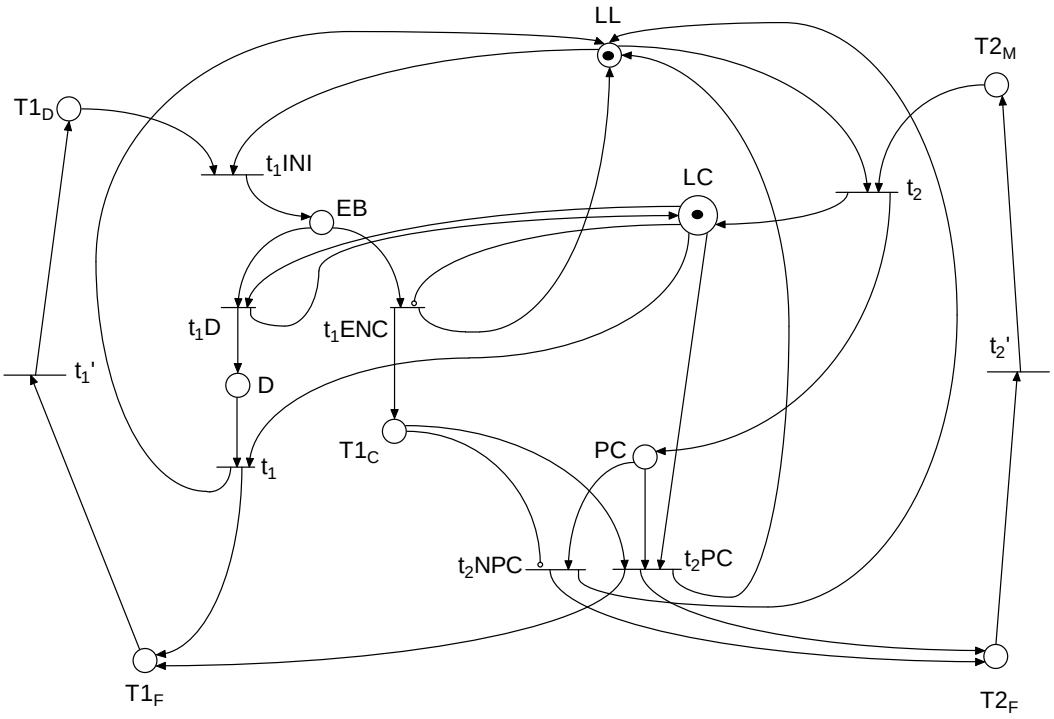


Figura E.1: Modelo completo de un lugar de comunicación implementado con un objeto protegido

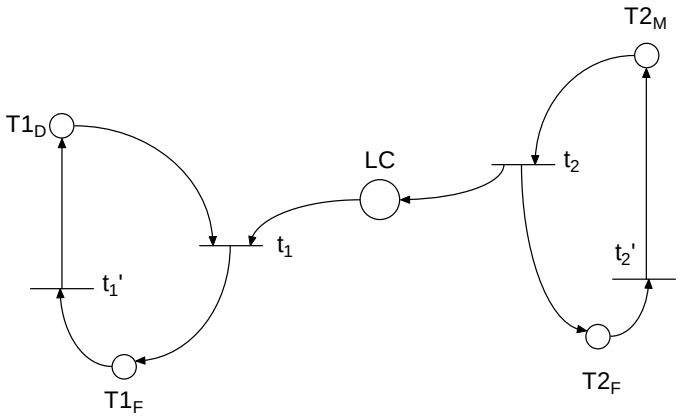


Figura E.2: Modelo reducido de un lugar de comunicación implementado con un objeto protegido

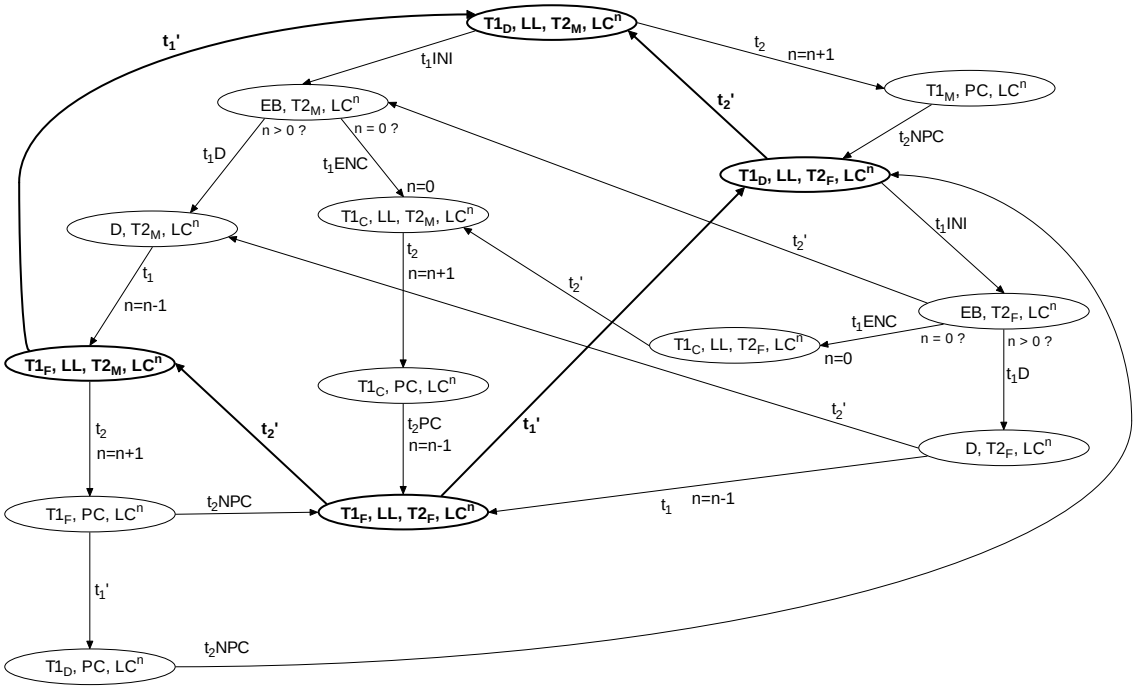


Figura E.3: Grafo de estados de la red de la figura E.1

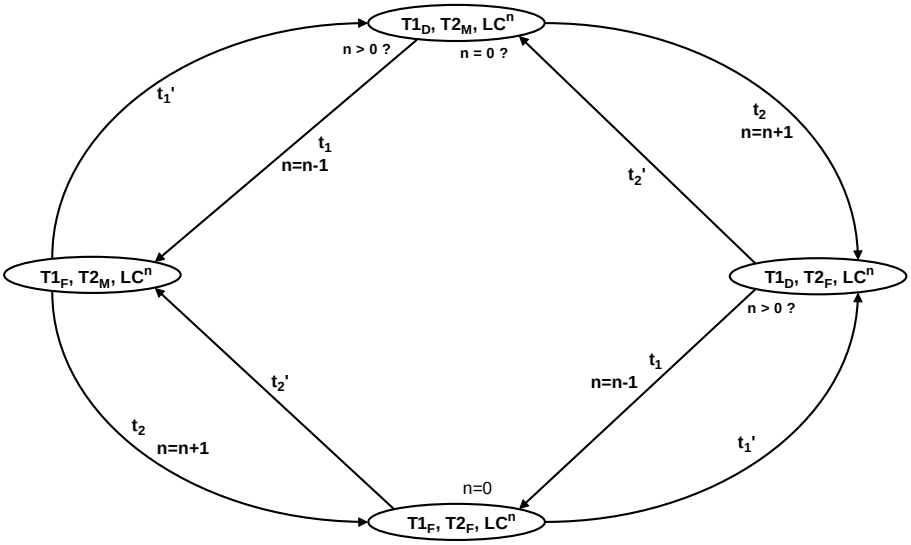


Figura E.4: Grafo de estados de la red de la figura E.2

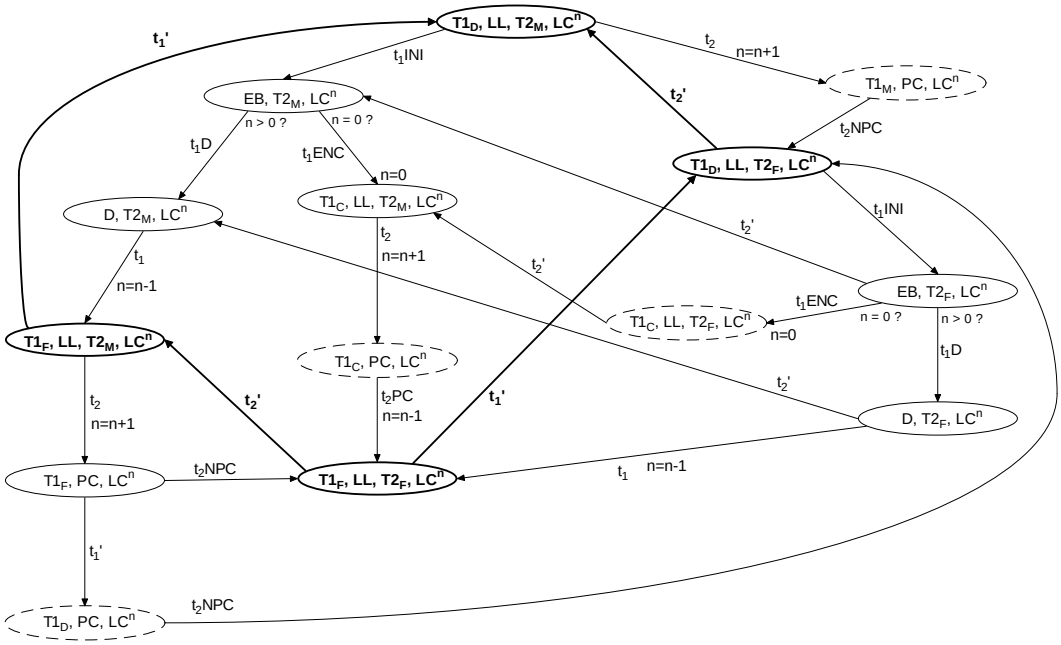


Figura E.5: Paso 1 en la reducción

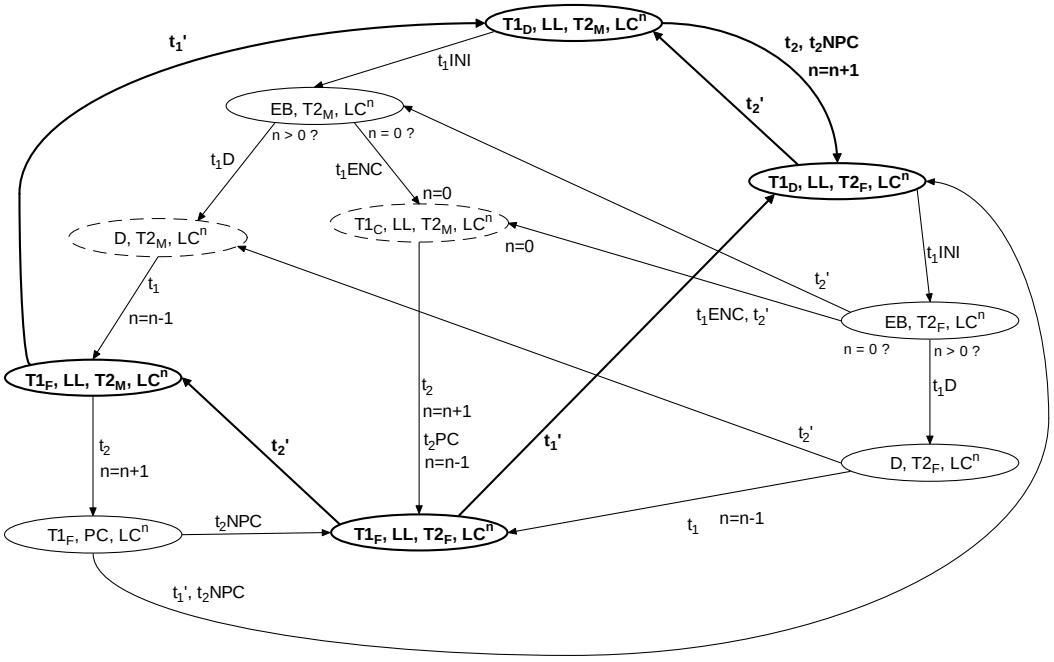


Figura E.6: Paso 2 en la reducción

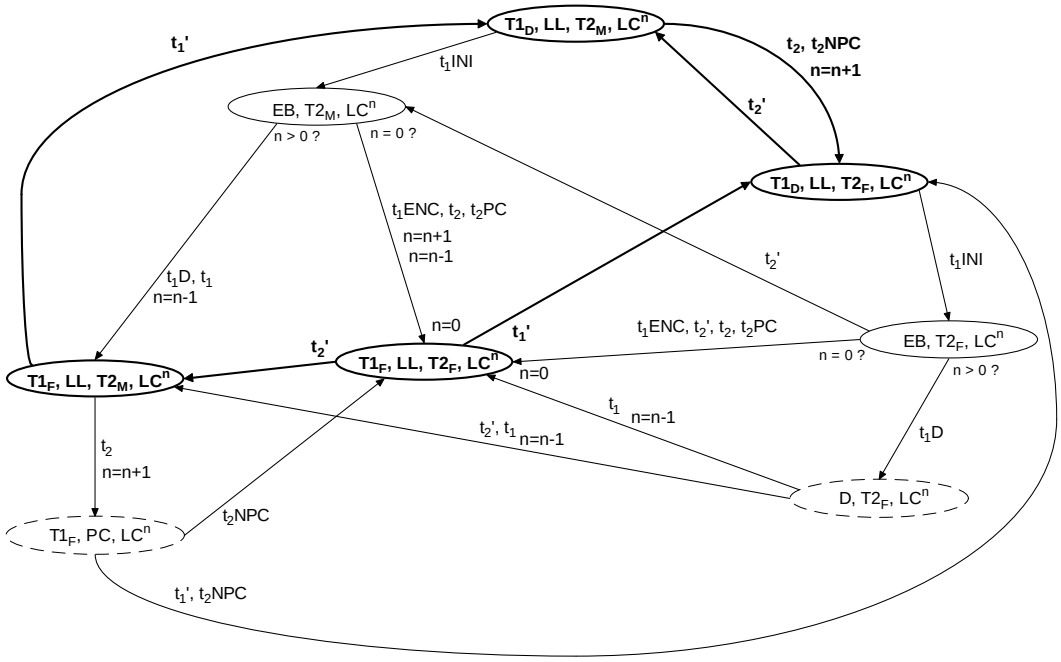


Figura E.7: Paso 3 en la reducción

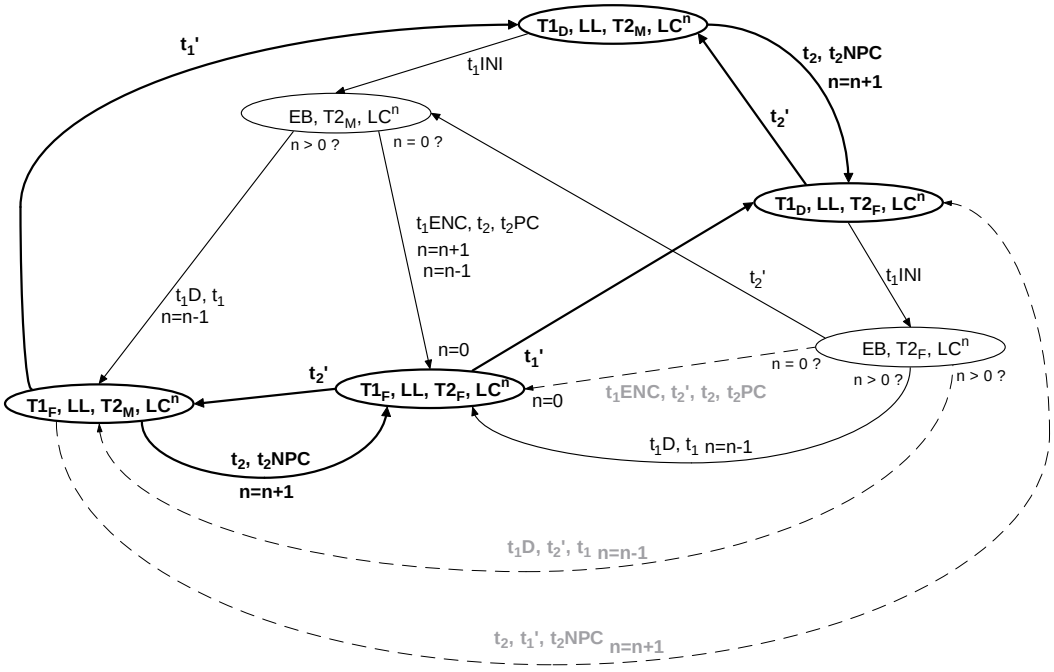


Figura E.8: Paso 4 en la reducción

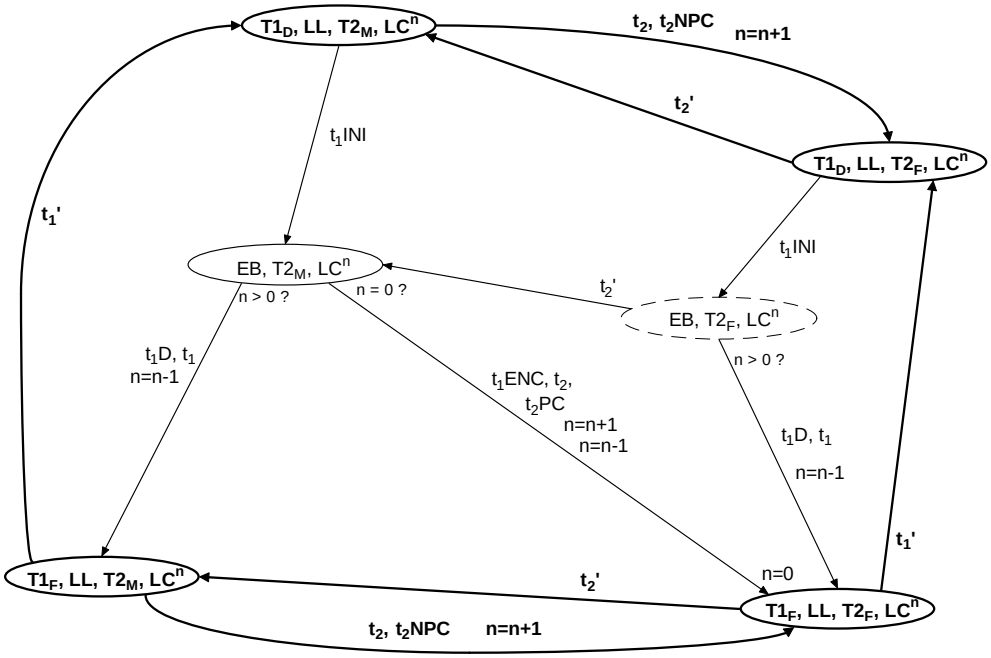


Figura E.9: Paso 5 en la reducción

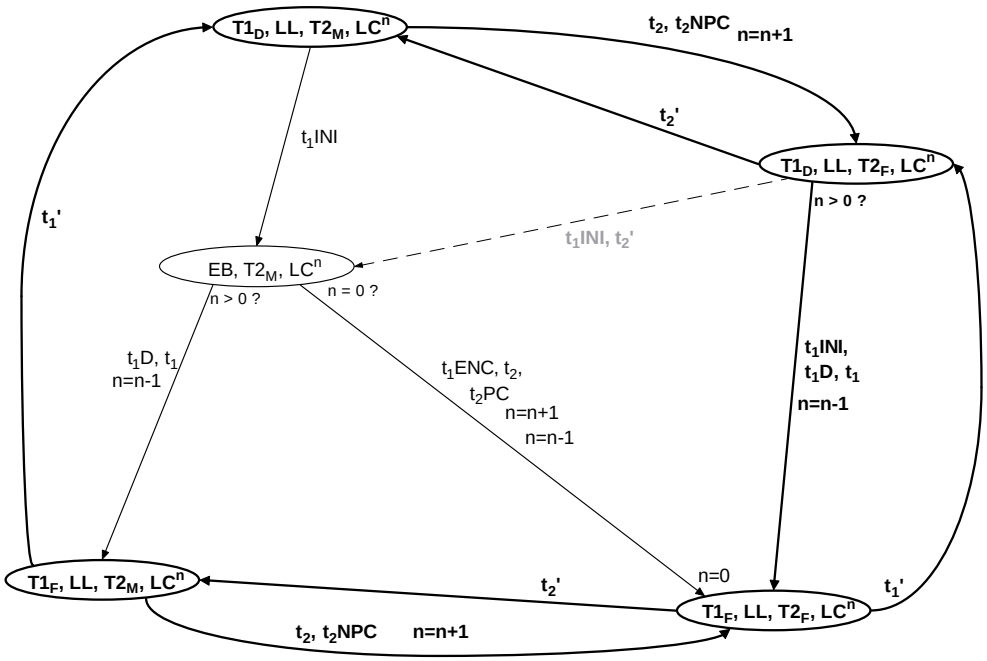


Figura E.10: Paso 6 en la reducción

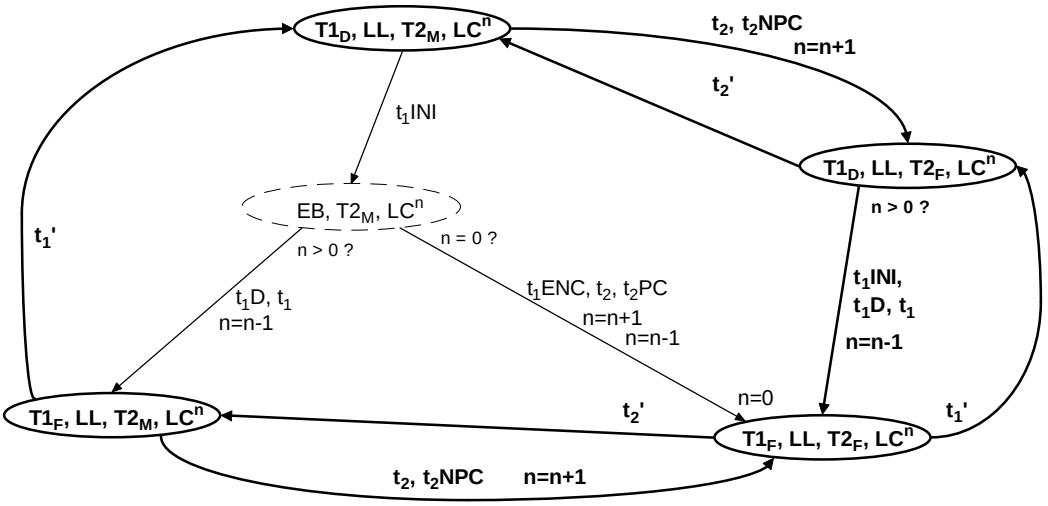


Figura E.11: Paso 7 en la reducción

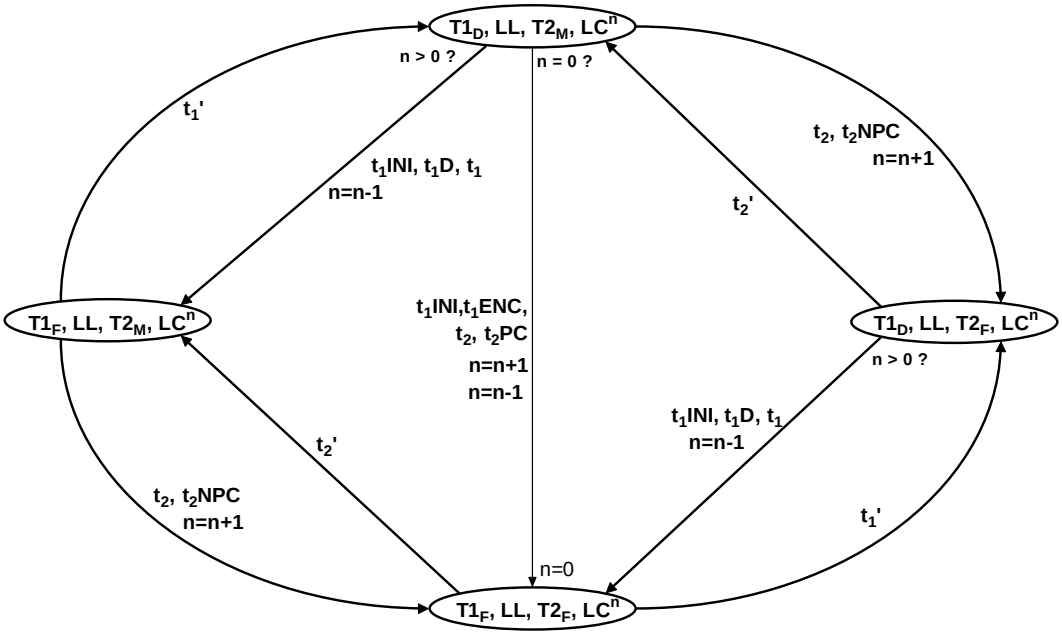


Figura E.12: Paso 8 en la reducción

Apéndice F

Ejemplo de aplicación

En este anexo se aplicarán las ideas expuestas en esta tesis al desarrollo de un sistema de control robótico, constituido por una plataforma móvil y varios sensores que captan información del entorno. Hemos elegido este ejemplo porque en él aparecerán varias actividades que deben ser ejecutadas de forma concurrente. El principal objetivo del análisis (ver sección F.3) será encontrar el mínimo período que garantice el cumplimiento de plazos en todos los procesos involucrados. El método también permite la adición de procesos concurrentes, modificando incrementalmente el sistema de control de una manera sencilla y uniforme.

F.1. Especificación de requisitos

En este anexo se mostrará como un diseño basado en RdPT hace fácil el análisis y la implementación de un sistema de control para un robot móvil, en el cuál varios procesos relacionados con el control del movimiento y los sistemas sensoriales están en concurrencia.

Para centrar el problema consideraremos que el robot móvil cuenta con un sensor telémetro láser con dos grados de libertad, usado en tareas de navegación con evitación de obstáculos (figura F.1). La información del láser permite modificar la trayectoria nominal del robot mientras éste se mueve en las proximidades de un obstáculo. En la navegación existen varias tareas involucradas:

- El *control del robot*, que controla su movimiento ejecutando periódicamente su bucle de control.
- El *controlador del láser*, que proporciona información de proximidad usada por el proceso de control para evitar los obstáculos. Gestiona los datos que periódicamente recoge el láser, por lo que el comportamiento de este subsistema es también periódico.
- El *subsistema de supervisión*, que supervisa todas las tareas robóticas para detectar si una meta o submeta de la trayectoria se ha alcanzado, actualiza el punto meta actual y gestiona el sistema de alarmas. Este subsistema actúa cada vez que se actualiza el estado del sistema, o se produce una alarma, por lo que su comportamiento no es periódico.

En este anexo no se mostrarán los detalles meramente robóticos del ejemplo: navegación, evitación de obstáculos, integración de muestras del láser, etc. Solamente comentaremos que la trayectoria a seguir se especifica en base a una serie de submetas que conducen a una meta final. El resto de detalles pueden ser consultados en [MGV99].

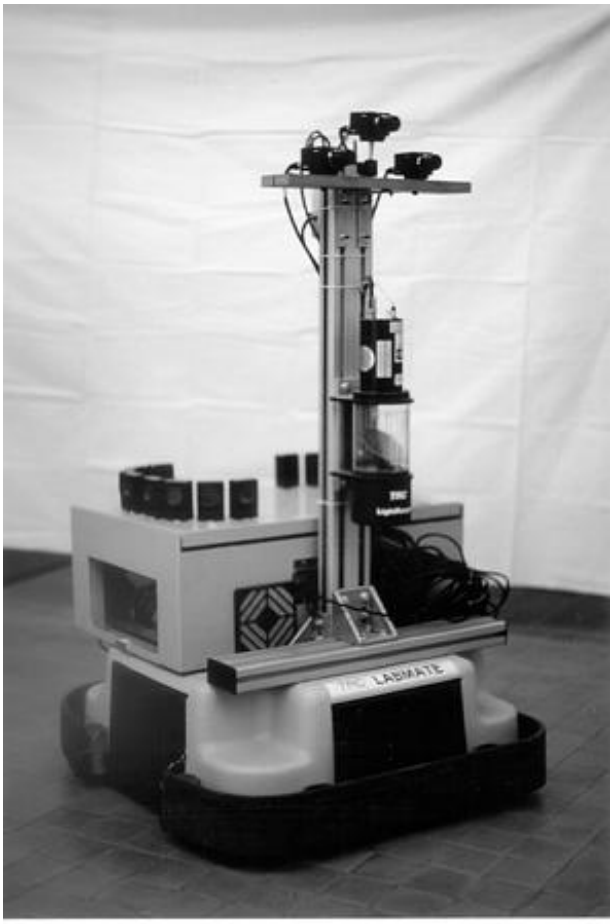


Figura F.1: Robot móvil objeto del ejemplo

F.1.1. Restricciones temporales

Las restricciones de tiempo real del sistema son las siguientes:

- El período del bucle de control será establecido durante el análisis. Sin embargo, podemos dar una cota inferior para él: debido a restricciones internas del robot (ver sección F.3), debe ser mayor que 0.18 segundos.

- Por razones de seguridad, se asociarán varios timeouts a las comunicaciones entre robot y controlador: 0.1 sg. en la lectura de la posición y 0.1 sg. en el envío de la acción de control (*setpoint*). De esta forma se facilitará la detección de fallos en las comunicaciones.

- El disparo de un timeout en una comunicación debe parar el sistema en 0.1 sg.

- El láser 3D envía una nueva muestra cada 0.1 sg. El controlador debe ser capaz de aceptar y procesar los datos del sensor a esa velocidad.

- La comunicación con el láser tiene un timeout de 0.2 sg.

F.2. Modelado

En la sección anterior hemos comentado que el robot estaba compuesto por tres subsistemas principales: el control del robot, el controlador del láser y el de supervisión. El primer paso será la construcción de una subred para cada uno de ellos, para después establecer el mecanismo de interconexión.

F.2.1. Subsistema de control del robot

El período de control del robot no se ha fijado todavía (se hará en la etapa de análisis, sección F.3), pero debe ser lo más pequeño posible. En cada período, el bucle de control debe realizar las siguientes actividades:

1. *Lectura de la posición* (*Read Position*). El control se comunica con el robot para obtener la posición actual del mismo. Las comunicaciones se realizan vía línea serie RS-232.

2. *Integración de muestras* (*Scans Integration*). Se integran los datos recogidos en las últimas diez vueltas del láser, para tener una información completamente actualizada de los obstáculos del entorno (ver [MGV99]).

3. *Cálculo de la acción de control* (*Compute Setpoint*). Basado en la posición actual del robot, los puntos de obstáculo detectados por el láser y la meta a alcanzar, se calcula la acción a enviar al robot, utilizando la técnica del campo de potencial (ver [MGV99]).

4. *Envío de la acción de control* (*Send Setpoint*). El proceso de control se comunica con el robot para enviarle un nuevo valor de velocidad (acción desde el punto de vista del control).

Para integrar los puntos detectados por el láser, se debe poder acceder a los datos del sensor una vez corregido el efecto de la velocidad del robot sobre el muestreo de los mismos. Esta información también debe ser accesible por el subsistema controlador del láser, que debe actualizarla. Por ello, se debe proporcionar acceso en exclusión mutua a los datos de estas muestras. Para minimizar el bloqueo en el acceso, la integración de las muestras será dividida en dos acciones: lectura de las muestras (*Read Scans*), que accede a los datos y realiza una copia local, y la propia integración (*Scans Integration*), que trabajará sobre la copia de los datos. Por la misma razón, el cálculo de la acción de control (*Compute Setpoint*) ha sido dividida en la lectura de la meta (*Read Goal*) y el cálculo de la acción de control (*Compute Setpoint*). La meta actual es compartida por la actividad de control y la de supervisión. Cuando se obtiene la posición actual del robot, una variable global protegida, compartida por todas las actividades, es actualizada. Esto es, siguiendo a la lectura de la posición (*Read Location*), se ejecuta la actividad almacenar estado (*Store State*). Un posible bloqueo de la actividad de control causado por una comunicación perdida con el robot puede ser evitada asociando timeouts a la lectura de la posición (*Read Location*) y al envío de la acción de control (*Send Setpoint*). La duración de estos timeouts ha sido especificada en la sección F.1.1.

F.2.2. Subsistema de supervisión

El subsistema de supervisión no es activado por un evento temporal periódico, sino que debe ser ejecutado cada vez que se recibe una nueva posición del robot. Sus actividades pueden desglosarse:

1. *Almacenar meta* (*Store Goal*). Actualiza la meta que va a ser usada por el controlador.

2. *Trazado de la posición* (*Trace Location*). Almacena el estado actual del robot en el sistema de ficheros. Para hacer esta actividad se debe acceder a la variable compartida *estado* (*State*). Por ello la acción *Supervisor Read State* es ejecutada antes de crear una copia local del estado actual.

3. *Test de meta* (*Goal Test*). Comprueba si el robot ha alcanzado la meta o una submeta de la trayectoria. Si se ha alcanzado una submeta, se debe calcular una nueva como siguiente objetivo. Si es la meta final la alcanzada, el robot debe ser parado.

4. *Gestión de alarmas* (*Alarm*). Ejecuta un código de tratamiento de alarma si se recibe un evento de alarma desde el controlador del láser. No consideramos la recuperación de errores en el sistema, sino que éste es detenido inmediatamente.

F.2.3. Subsistema controlador del láser

Finalmente, el subsistema controlador del láser tampoco es activado periódicamente. Sin embargo, se ejecuta cada vez que una nueva muestra llega del sensor, y éstas lo hacen según un patrón periódico. Por tanto, se comporta como si fuera periódico. Sus cometidos son:

1. *Leer muestra* (*Read Scan*). Lee una muestra comunicándose con el sensor láser. Esta comunicación tiene un timeout asociado de 0.1 sg.

2. *Corregir muestra* (*Correct Scan*). Calcula el tiempo de corrección de los datos muestreados por el láser para tener en cuenta el efecto del movimiento del robot. Esta acción accede al estado del robot y a las muestras, que son variables compartidas. Por ello, para minimizar el bloqueo, esta acción se ha dividido en *Laser Read State*, *Correct Scan* y *Store Scan*.

La figura F.2 muestra los modelos de RdPT de los subsistemas aislados. El próximo paso en el modelado ha de ser el establecimiento de las comunicaciones y sincronizaciones entre ellos.

F.2.4. Comunicación y sincronización entre subsistemas

A la hora de comunicar los subsistemas descritos se deben tener en cuenta los siguientes puntos:

- Las variables compartidas deben ser protegidas contra el acceso concurrente. Un lugar monomarcado, State, Scans y Goal, modela un semáforo que preserva la exclusión mutua en las primitivas de acceso.

- La llegada de una nueva posición del robot al subsistema de control debe ser señalada al subsistema de supervisión. Esto se ha realizado mediante un semáforo binario, lugar CC4, que comunica el control con la supervisión.

- Cada vez que un timeout es completado, se debe enviar una señal de alarma al subsistema de supervisión. Esto se ha realizado mediante dos lugares de comunicación CC3 y LC2.

- Cuando una señal de alarma es procesada por el supervisor, todos los subsistemas deben ser detenidos. Para ello, se envía una señal a cada uno de los procesos activos y, entonces, el supervisor finaliza. La señalización es implementada mediante los lugares CC1, para parar la activación periódica del control, CC2 para parar el subsistema de control y LC1, para parar el subsistema controlador del láser.

- Cuando el supervisor detecta que el robot ha alcanzado la meta final de su trayectoria, envía una señal de parada al resto de subsistemas, del mismo modo que el punto anterior. Sin embargo, la parada del subsistema de control es realizada mediante el lugar CC5, porque, para alcanzar de forma efectiva la meta antes de la parada definitiva del robot, es necesario un movimiento final.

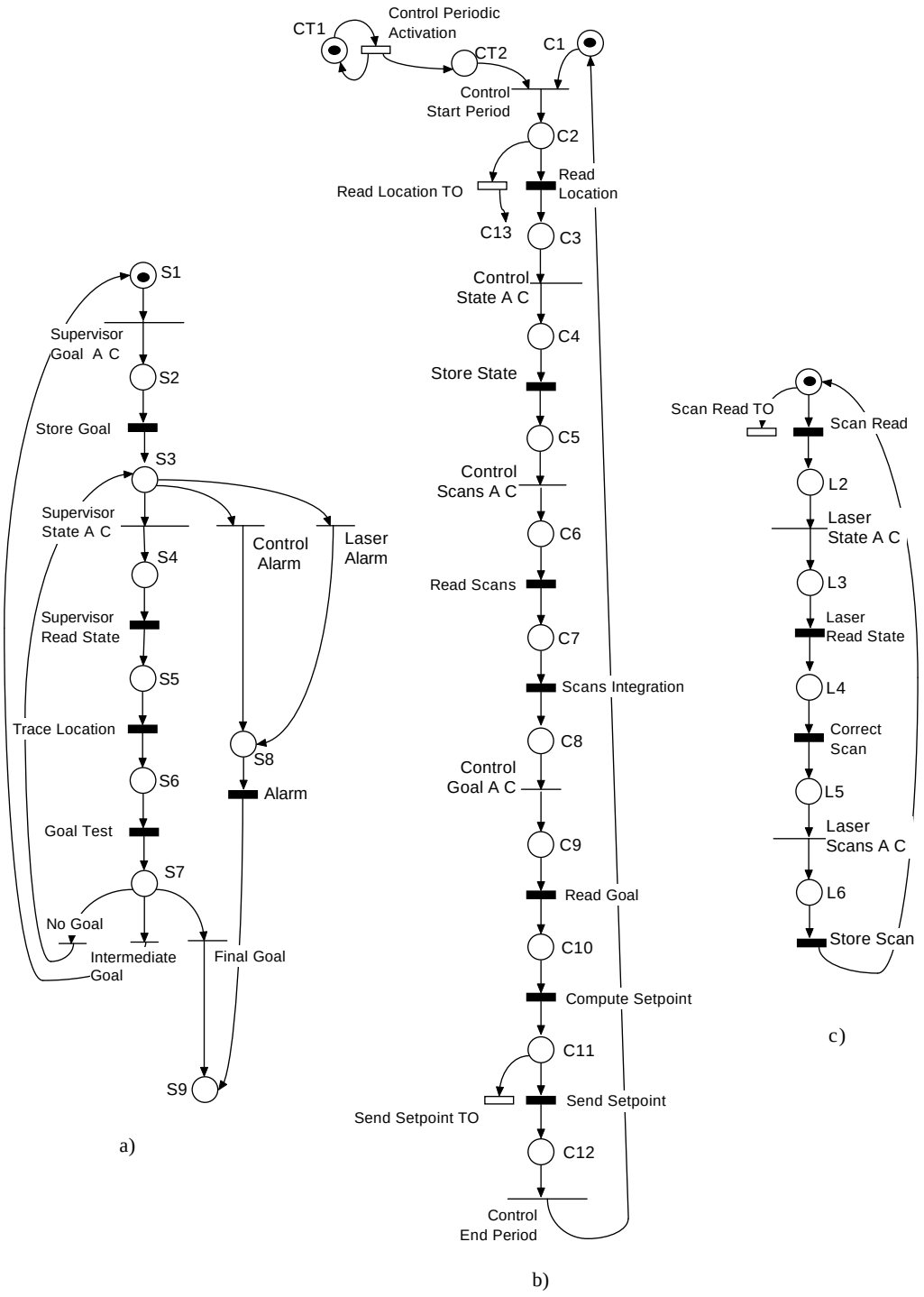


Figura F.2: RdPT que modelan los subsistemas del ejemplo: a) subsistema de supervisión; b) subsistema de control del robot; c) subsistema del controlador del láser

Para completar el modelo es necesario asociar los tiempos de ejecución de mejor y peor caso a las transiciones CODE. El cálculo de estos tiempos ha sido realizado ejecutando directamente los trozos de código de cada transición sobre la plataforma destino (una workstation SPARC CPU-5V ejecutando SOLARIS en modo de tiempo real). La figura del final del anexo (F.5) muestra la RdPT que especifica completamente el sistema del ejemplo (la información temporal no se ha incluido por razones de claridad). Esta red puede ser usada directamente para la generación de código, pero para el realizar el análisis de la misma en un sistema monoprocesador es preciso realizar ciertas modificaciones.

F.3. Planificación y análisis de tiempo real

El objetivo final de este ejemplo es la generación de código para un sistema monoprocesador. Como se argumentó en la sección 2.3, la plataforma de ejecución hace necesaria una planificación de las actividades del sistema, para que todas ellas puedan cumplir sus requisitos temporales. No ha sido objetivo de esta tesis el planteamiento de una técnica de planificación basada en RdPT. En su lugar, hemos utilizado un método heurístico basado en *Rate Monotonic Scheduling* (RMS).

Sabemos desde la especificación de requisitos, que en el sistema hay dos subsistemas periódicos (el de control del robot y el del controlador del láser) y uno esporádico (el supervisor). El período de control del robot es un parámetro de diseño, pero que debe ser mayor de 0.18 sg. a causa de las comunicaciones con el robot (hay tres comunicaciones involucradas, cada una de 0.06 sg). Desde el punto de vista del RMS, las transiciones involucradas en el subsistema controlador del láser deben ser más prioritarias que las del control (ya que su período es de 0.1 sg). Para reducir el tiempo de respuesta del subsistema de supervisión, haremos que sus transiciones tengan mayor prioridad que las demás.

El siguiente paso en la planificación será realizar el análisis del comportamiento de la RdPT, con las restricciones de ejecución en un sistema monoprocesador y con las prioridades asignadas, para verificar las restricciones temporales.

El análisis de RdPT (ver sección 2.4) se realiza casi exclusivamente por medio de técnicas enumerativas, que hacen uso del *grafo de clases de estado* [BD91]. En nuestra aplicación robótica hay dos clases de actividades: aquellas que cuentan con un tiempo de cómputo fijo ($[\alpha, \alpha]$); y las que están involucradas en una comunicación externa. Si no hay ningún problema, la comunicación dura un tiempo fijo, pero puede bloquearse, por lo que su intervalo asociado será $[\alpha, \infty)$.

Estas características permiten la simplificación del análisis de RdPT. La red puede ser transformada en otras objeto de análisis: una con los tiempos de ejecución fijos (que responderá al caso en el que no hay problemas con las comunicaciones), y otra para cada posible timeout de comunicación que pueda bloquearse (ver figura F.3). El generador y analizador de grafos de estados que se ha desarrollado para este análisis tiene en cuenta las prioridades estáticas asociadas, y el comportamiento del núcleo de Ada 95 y del sistema operativo.

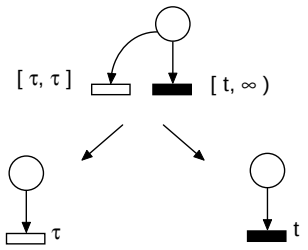


Figura F.3: Una actividad con tiempo de cómputo fijo, pero que puede bloquearse y es protegida por un timeout, puede dividirse en dos casos de prueba con intervalos puntuales

Para llevar a cabo el análisis de tiempo real del sistema se han realizado algunas modificaciones sobre el modelo. Principalmente:

- Se ha añadido un activador periódico para el subsistema controlador del láser. Éste modela la llegada periódica de muestras del sensor y hace el modelo autónomo.
- Se ha añadido un lugar implícito a ambos subsistemas periódicos. Como se explicó en la sección 2.4, este lugar es marcado cuando comienza la actividad periódica y desmarcado cuando termina.

El objetivo principal del proceso de análisis ha sido el de encontrar un límite inferior para el período del subsistema de control. Para alcanzar este objetivo, el analizador de grafos de estados ha sido programado para detectar lugares no binarios y pares de lugares marcados simultáneamente. La aparición de un lugar no binario supone la existencia de un evento en el sistema que no ha podido ser procesado a la frecuencia requerida. Por otra parte, el marcado simultáneo de un lugar de activación y del correspondiente lugar representante de una actividad (el implícito añadido) significa que la actividad periódica no ha terminado antes de la siguiente activación, esto es, no se han cumplido los plazos de ejecución estrictos.

Como ya se ha explicado, el mínimo período del subsistema de control es 0.18 sg. El sistema ha sido analizado iterando desde ese punto de partida, en pasos de 10 ms. Se ha encontrado que, para las prioridades impuestas, el mínimo período para el cual se cumplen todas las restricciones temporales es 0.2 sg. Basándonos en este resultado, tomaremos como período de la actividad de control 0.25 sg.

F.4. Implementación

F.4.1. Partición en protoprocesos

Se ha realizado un cálculo de p-invariantes monomarcados para el reconocimiento de los protoprocesos del sistema. Hay siete p-invariantes: $(I_1) = \{S_1..S_9\}$, $(I_2) = \{C_1..C_{15}\}$, $(I_3) = \{CT_1, CT_3\}$, $(I_4) = \{L_1..L_8\}$, $(I_5) = \{Goal, S_2, C_9\}$, $(I_6) = \{State, L_3, C_4, S_{10}\}$ e $(I_7) = \{Scans, L_6, C_6\}$. Con estos p-invariantes se puede resolver el problema de cobertura. Las transiciones a cubrir se muestran en la figura F.4. En esta tabla se puede apreciar que existen cuatro p-invariantes esenciales (I_1, I_2, I_3, I_4), que cubren todas las transiciones de la red. El primer protoproceso (generado por I_1) corresponde al subsistema de supervisión, el segundo (generado por I_2) al de control, el tercero (I_3) al activador periódico del subsistema de control, y el cuarto (I_4) al controlador del láser.

Además, se han encontrado varios lugares de comunicación, de destino único (como CC_4, CC_3, LC_2, CC_1, CT_2, LC_1, CC_2 y CC_5), y de destino múltiple (con conflictos simples de salida), como *Goal*, *State* y *Scans*. En la última figura del anexo se muestra una distribución de los protoprocesos y lugares de comunicación descritos.

F.4.2. Implementación software

Aplicando las propuestas de implementación del capítulo 5 puede obtenerse el código Ada 95 que implementa el ejemplo del sistema de control del robot. Cada protoproceso será implementado en una tarea Ada 95, por lo que la implementación contará con cuatro tareas: `Supervisor`, `Control`, `Laser`, y `Control_Activation`.

Los lugares de comunicación CC_4, CC_3, LC_2, CC_1, CT_2, CC_2 y CC_5 serán implementados según la versión activa de lugar de comunicación propuesta en la sección 5.3.1, pues se trata de lugares cuyo destino son conflictos internos sin código (ver sección 5.4.2.c). Sin embargo el destino del lugar LC_1 es un conflicto interno con código, por lo que debe ser implementado (ver 5.4.1.e)

	I_1	I_2	I_3	I_4	I_5	I_6	I_7
Store_Goal	x				x		
Supervisor_Read_State	x					x	
Trace_Location	x						
Goal_Test	x						
Alarm	x						
Supervisor_Goal_Access_Control	x				x		
Supervisor_Wait_State	x					x	
Supervisor_State_Access_Control	x						
No_Goal	x						
Intermediate_Goal	x						
Final_Goal	x						
Control_Alarm	x						
Laser_Alarm	x						
Move_To		x					
Robot_Stop		x					
Read_Location		x					
Store_State		x				x	
Read_Scans		x					x
Scans_Integration		x					
Read_Goal		x			x		
Compute_Setpoint		x					
Send_Setpoint		x					
Control_Scans_Access_Control		x					x
Control_Goal_Access_Control		x			x		
Control_State_Access_Control		x				x	
Control_Start_Period		x					
Stop_Control		x					
Control_End_Period		x					
End_of_trajectory_Control		x					
Read_Location_TO		x					
Send_Setpoint_TO		x					
Stop_Act_Control			x				
Control_Periodic_Activation			x				
Scan_Read				x			
Laser_Read_State				x		x	
Correct_Scan				x			
Store_Scan				x			x
Stop_Scan				x			
Laser_State_Access_Control				x		x	
Laser_Scans_Access_Control				x			x
Stop_Laser				x			
Scan_Read_TO				x			

Figura F.4: Tabla de cobertura de las transiciones del ejemplo

como un objeto protegido (sección 5.3.1). Los lugares *Goal*, *State* y *Scans* son lugares de destino múltiple generadores de conflictos simples, por lo que serán implementados como se indica en 5.5.1.

El código del sistema de control del robot es el siguiente:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Queuing_Policy (Priority_Queuing);

with System, Laser_Navigation_Procedures, Ada.Text_IO, Ada.Real_Time, Global_Data;
use Laser_Navigation_Procedures, Ada.Text_IO, Ada.Real_Time, Global_Data;

procedure Laser_Navigation is

  -- ####                                    ####
  --        PROCESOS DE LA RdPT
  -- ####                                    ####

  task Supervisor is
    pragma Priority (HIGH);
    entry Start;
    entry New_Robot_State;
    entry Control_Alarm;
    entry Laser_Alarm;
  end Supervisor;

  task Control is
    pragma Priority (LOW);
    entry Start;
    entry Control_Star_Period;
    entry Stop_Control;
    entry End_Of_Trajectory_Control;
  end Control;

  task Laser is
    pragma Priority (MEDIUM);
    entry Start;
  end Laser;

  task Control_Activation is
    pragma Priority (MAXIMUM);
    entry start;
    entry Stop_Act_Control;
  end Control_Activation;

  task body Supervisor is separate;
  task body Control is separate;
  task body Laser is separate;
  task body Control_Activation is separate;

  -- ####                                    ####
  --        LUGARES DE COMUNICACION
  -- ####                                    ####
```

```
-- Implementación de lugares Goal, State, Scans, LC1
protected type Communication_Place is
  pragma Priority (MAXIMUM);
  entry Demark;
  procedure Mark;
private
  Marking: integer := 1; -- Inital marking value
end Communication_Place;

protected body Communication_Place is
  entry Demark when Marking > 0 is
  begin
    Marking := Marking - 1;
  end;
  procedure Mark is
  begin
    Marking := Marking + 1;
  end;
end Communication_Place;

Goal, State, Scans, LC1: Communication_Place;

-- Implementación de lugares CC1, CC2, CC3, CC4, CC5, LC2, CT2
generic
  with procedure Demark;
package Relay_Places is
  procedure Mark;
end Relay_Places;

package body Relay_Places is
  task Relay is
    pragma Priority (MAXIMUM);
    entry MarkMe;
  end Relay;

  procedure Mark is
  begin
    Relay.MarkMe;
  end;

  task body Relay is
  begin
    select
      accept MarkMe;
    or
      terminate;
    end select;
    Demark;
  end Relay;
end Relay_Places;
```

```
  package CC4 is new Relay_Places(Supervisor.New_Robot_State);
  package CC3 is new Relay_Places(Supervisor.Control_Alarm);
  package LC2 is new Relay_Places(Supervisor.Laser_Alarm);
  package CC2 is new Relay_Places(Control.Stop_Control);
  package CC5 is new Relay_Places(Control.End_Of_Trajectory_Control);
  package CT2 is new Relay_Places(Control.Control_Star_Period);
  package CC1 is new Relay_Places(Control_Activation.Stop_Act_Control);

  -- ####                                    ####
  --        COMIENZO PROGRAMA PRINCIPAL
  -- ####                                    ####

begin
  Init_Laser;
  Init_Robot;
  Supervisor.        Start;
  Control.           Start;
  Laser.             Start;
  Control_Activation.Start;
end Laser_Navigation;


  -- ####                                    ####
  --            PROCESO DE CONTROL
  -- ####                                    ####

separate (Laser_Navigation)
task body Control is
  Read_Location_TO: constant duration := 0.1;
  Send_Set_Point_TO: constant duration := 0.1;
begin
  accept start;
  loop
    select
      accept Control_Star_Period;
        select
          delay Read_Location_TO;
          CC3.Mark;
          exit;
        then abort
          Read_Location;
        end select;
        State.Demark;
        Store_State;
        State.Mark;
        CC4.Mark;
        Scans.Demark;
        Read_Scans;
        Scans.Mark;
        Scans_Integration;
```

```
        Goal.Demark;
        Read_Goal;
        Goal.Mark;
        Compute_Setpoint;
        select
          delay Send_Set_Point_TO;
          CC3.Mark;
          exit;
        then abort
          Send_Setpoint;
        end select;
    or
      accept Stop_Control;
      exit;
    or
      accept End_Of_Trajectory_Control;
      Move_To;
      exit;
    end select;
  end loop;
  Stop_Robot;
end;


  -- ####                                    ####
  --      PROCESO DE ACTIVACION DE CONTROL
  -- ####                                    ####

separate (Laser_Navigation)
task body Control_Activation is
  T_last_firing : time;
  PERIOD : time_span := To_Time_Span (0.25);
begin
  accept start;
  T_last_firing := CLOCK;
  loop
    select
      accept Stop_Act_Control;
      T_last_firing := CLOCK;
      exit;
    or
      delay until T_last_firing + PERIOD;
      T_last_firing := T_last_firing + PERIOD;
      CT2.Mark;
    end select;
  end loop;
end;

  -- ####                                    ####
  --            PROCESO DEL LASER
  -- ####                                    ####
```

```
separate (Laser_Navigation)
task body Laser is
  Scan_Read_TO: constant duration := 0.2;
begin
  accept start;
  loop
    select
      LC1.Demark;
      exit;
    then abort
      select
        delay Scan_Read_TO;
        exit;
      then abort
        Scan_Read;
      end select;
    end select;
    State.Demark;
    Laser_Read_State;
    State.Mark;
    Correct_Scan;
    Scans.Demark;
    Store_Scan;
    Scans.Mark;
  end loop;
  Stop_Scan;
end;

  -- ####                                    ####
  --          PROCESO DE SUPERVISION
  -- ####                                    ####

separate (Laser_Navigation)
task body Supervisor is
begin
  accept start;
  S1: loop
        Goal.Demark;
        Store_Goal;
        Goal.Mark;
  S3:   loop
          select
            accept New_Robot_State;
              State.Demark;
              Supervisor_Read_State;
              State.Mark;
              Trace_Location;
              Goal_Test;
              if Test_Result = No_Goal then
                null;
```

```
            elsif Test_Result = Intermediate_Goal then
              exit S3;
            elsif Test_Result = Final_Goal then
              CC1.Mark;
              LC1.Mark;
              CC5.Mark;
              exit S1;
            end if;
          or
            accept Control_Alarm;
              CC1.Mark;
              LC1.Mark;
              Alarm;
              exit S1;
          or
            accept Laser_Alarm;
              CC1.Mark;
              CC2.Mark;
              Alarm;
              exit S1;
          end select;
        end loop S3;
      end loop S1;
end Supervisor;
```

F.5. Resultados experimentales

Hemos probado el sistema de control del robot haciéndole navegar a lo largo de una trayectoria en el interior de un entorno desestructurado. El entorno se compone de dos corredores en los que hay dos puertas que el robot debe cruzar, así como varios obstáculos que debe evitar. Estos obstáculos son detectados por el telémetro láser que actúa corrigiendo el movimiento nominal.

Se ha estudiado el comportamiento en tiempo real del sistema de control, que satisfactoriament sincroniza los procesos que se ejecutan en paralelo en el sistema. Se ha probado y verificado el período mínimo teórico para el bucle de control obtenido en la fase de análisis.

Para verificar el cumplimiento de las restricciones de tiempo real en la ejecución del sistema de control se han incluido en el modelo de RdPT elementos de supervisión que permiten la detección y tratamiento de violaciones de deadline. La figura F.5 muestra la forma en que la que se detecta que un proceso periódico no cumple con su deadline (suponiendo que el deadline es igual al período). Basados en esta estructura de supervisión, se ha calculado experimentalmente el valor del período del bucle de control. Como en el método teórico, se ha iterado partiendo del límite inferior (0.18 sg.) hasta que no se ha detectado ninguna violación de período. Se han obtenido dos resultados diferentes: 0.25 sg. para la implementación centralizada, y 0.20 sg. para la descentralizada. En la figura F.6 se muestran dos trazas de ejecución para la implementación descentralizada. En la primera, todos los procesos cumplen sus requisitos temporales. En la segunda, se produce una violación de deadline debida al uso de un valor de período más pequeño que el teórico. El resultado obtenido para la implementación descentralizada demuestra la validez del análisis teórico. Sin embargo, para la implementación centralizada, se ha encontrado un período mínimo de 0.25 sg. La desviación de 50 ms. respecto del valor teórico es debida, en este caso, a la sobrecarga introducida por el coordinador de la implementación (ver sección 3.7), que reduce el factor de utilización.

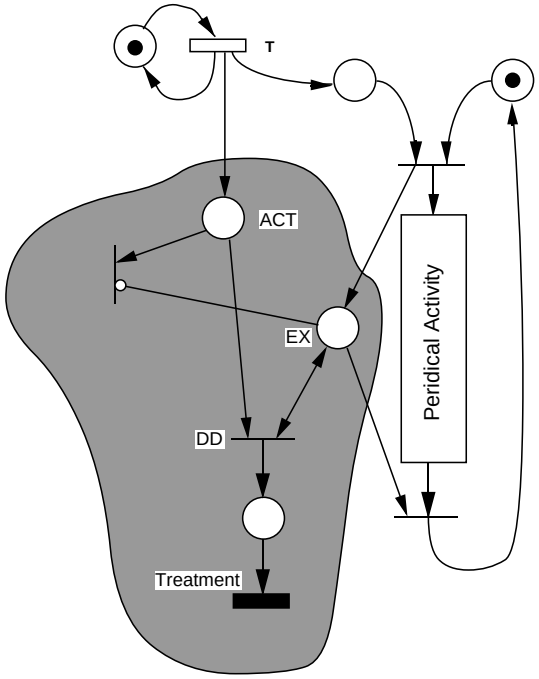
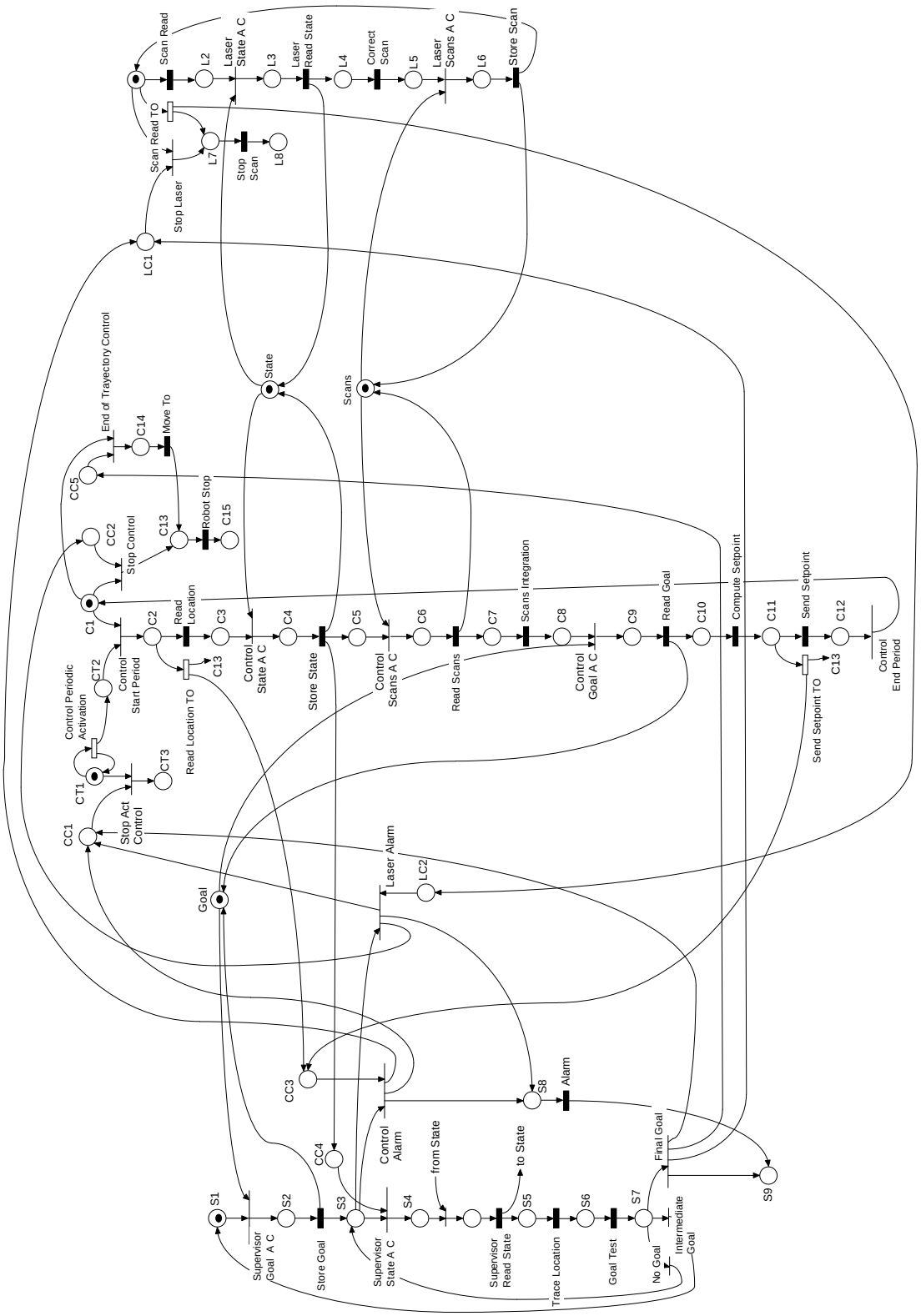
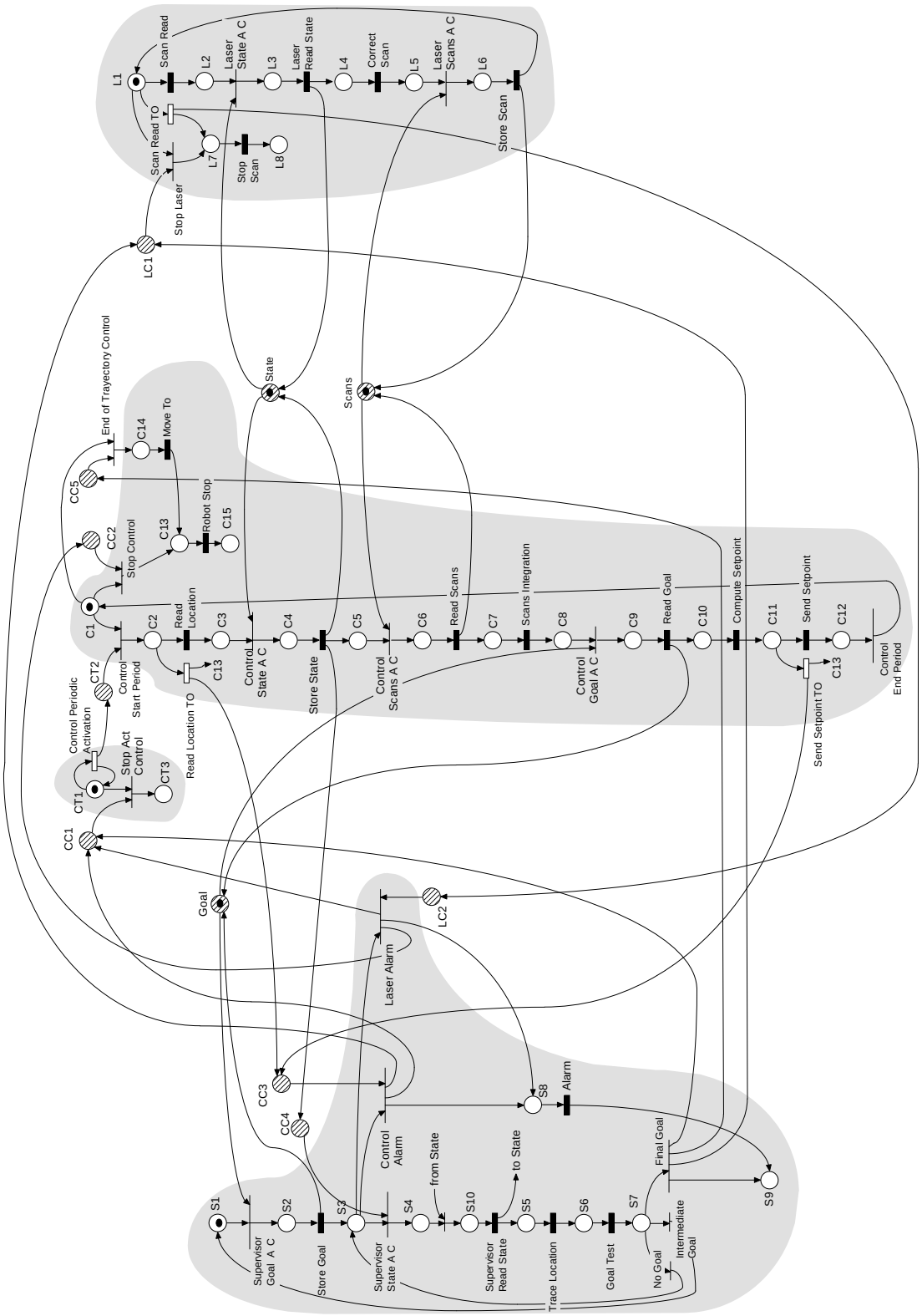


Figura F.5: Elemento de supervisión añadido para detectar violaciones de deadline en procesos periódicos en los que el deadline sea igual al período. La presencia simultánea de una marca en los lugares ACT *y* EX *evidencia que una nueva activación ha llegado mientras que la anterior todavía no ha finalizado. Con este marcado la transición* DD *es disparada y el tratamiento de la violación de deadline comienza.*

Figura F.6: Traza de ejecución de los procesos generados en la implementación descentralizada en dos situaciones distintas: a) con un período para el proceso de control de 0.25 sg. (mayor que el teórico); b) con un período de control de 0.19 sg. (menor que el teórico)


Scan Read
L2
Laser State A C
L3
Laser Read State
L4
Correct Scan
L5
Laser Scans A C
L6
Store Scan
Scan Read TO
L7
Stop Scan
L8
Stop Laser
LC1
State
Scans
End of Trayectory Control
C14
Move To
CC5
CC2
Stop Control
C13
Robot Stop
C15
C1
C2
Read Location
C3
Control State A C
C4
Store State
C5
Control Scans A C
C6
Read Scans
C7
Scans Integration
C8
Control Goal A C
C9
Read Goal
C10
Compute Setpoint
C11
Send Setpoint
C12
Control End Period
Control Periodic Activation
CT2
Control Start Period
Read Location TO
C13
CT1
CT3
Stop Act Control
CC1
Goal
Laser Alarm
LC2
Send Setpoint TO
CC3
S8
Alarm
CC4
Control Alarm
from State
to State
Final Goal
S9
S1
S2
S3
S4
S5
S6
S7
Supervisor Goal A C
Store Goal
Supervisor State A C
Supervisor Read State
Trace Location
Goal Test
No Goal
Intermediate Goal


L1
Scan Read
L2
Laser State A C
L3
Laser Read State
L4
Correct Scan
L5
Laser Scans A C
L6
Store Scan
Scan Read TO
L7
Stop Scan
L8
Stop Laser
LC1
State
Scans
End of Trayectory Control
C14
Move To
CC5
CC2
Stop Control
C13
Robot Stop
C15
C1
C2
Read Location
C3
Control State A C
C4
Store State
C5
Control Scans A C
C6
Read Scans
C7
Scans Integration
C8
Control Goal A C
C9
Read Goal
C10
Compute Setpoint
C11
Send Setpoint
C12
Control End Period
Send Setpoint TO
C13
Read Location TO
CT2
Control Start Period
Control Periodic Activation
Stop Act Control
CT1
CT3
CC1
Goal
LC2
Laser Alarm
CC3
CC4
S8
Alarm
Control Alarm
from State
to State
Final Goal
S9
S1
S2
S3
S4
S10
S5
S6
S7
Supervisor Goal A C
Store Goal
Supervisor State A C
Supervisor Read State
Trace Location
Goal Test
No Goal
Intermediate Goal

Bibliografía

[Aal93] W.M.P. van der Aalst. Interval timed coloured Petri nets and their analysis. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, number 691 in Lecture Notes in Computer Science, pages 453–472. Springer Verlag, Berlin, 1993.

[ABC+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* Wiley, 1995.

[ABCC87] M. Ajmone Marsan, G. Balbo, G. Chiola, and G. Conte. Generalized stochastic Petri nets revisited: Ramdom switches and priorities. In *Proc. of the Int. Workshop on Petri Nets and Performance Models*, pages 44–53, Madison, Washington DC., August 1987. IEEE-Computer Society Press.

[ABRW92] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling: the deadline-monotonic approach. In *Proc. of 8th. IEEE Workshop on Real-Time Operating Systems and Software*, May 1992.

[And91] G.R. Andrews. *Concurrent Programming. Principles and Practice.* The Benjamin-Cummings Publishing Company, 1991.

[AO95] W.M.P. van der Aalst and M.A. Odijk. Analysis of railway stations by means of interval timed coloured Petri nets. *Real-Time Systems*, 9(3):241–263, November 1995.

[ARM94] International Standard ISO/IEC 8652:1995. *Ada Reference Manual. Languaje and Standard Libraries*, November 1994.

[Ata93] Y. Atamna. Definition of the model stochastic timed well-formed coloured nets. In *Proc. of 5th. Int Workshop on Petri Nets and Performance Models (PNPM93)*, pages 24–33. IEEE Computer Society Press, 1993.

[Bak90] B. Bako. *Mise en oeuvre et simulation du niveau coordination de la commande des ateliers flexibles: une approche mixte reseaux de Petri et systemes de regles.* PhD thesis, Universite Paul Sabatier de Toulouse, 7, avenue du Colonel Roche, 31077 Toulouse Cedex, France, 1990.

[Bar87] D.S. Barbalho. *Conception et mise en oeuvre de la fonction coordination pour une commande distribu e d'atelier.* PhD thesis, Universite Paul Sabatier de Toulouse, December 1987.

[Bar95] J. Barnes. *Programming in Ada 95.* Addison Wesley, 1995.

[BC93] J.L. Briz and J.M. Colom. Implementation of Weighted Place /Transition Nets based on Linear Enabling Functions. Research Report 9314, Departamento Ing. Eléctrica e Informática, Univ. Zaragoza, 1993.

[BCMP92] G. Bruno, A. Castella, G. Macario, and M.P. Pescarmona. Scheduling hard real time systems using high-level Petri nets. In K. Jensen, editor, *Application and Theory of Petri Nets 1992*, pages 93–112. Springer Verlag, Berlin, June 1992.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE transactions on Software Engineering*, 17(3):259–273, March 1991.

[Ber85] G. Berthelot. Cheking properties of nets using transformations. In G. Rozenberg, editor, *Advances in Petri Nets 1985*, pages 19–40. Springer Verlag, Berlin, 1985.

[BH72] P. Brinch-Hansen. Structured-multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[BMS93] O. Babaoglu, K. Marzullo, and F.B. Schneider. A formalization of priority inversion. *Journal of Real-Time Systems*, 5(4):285–303, October 1993.

[BP94] F. Bréant and J.F. Peyre. An improved massively parallel implementation of colored Petri nets specifications. In *IFIP-WG 10.3 working conference on programming environments for massively parallel distributed systems, Ascona, Switzerland*, 1994.

[BP99] E. Bruneton and J.F. Pradat–Peyre. Automatic verification of concurrent ada programs. In M. Gonzalez Harbour and J.A. de la Puente, editors, *Reliable Software Technologies - Ada-Europe'99*, number 1622 in Lectures Notes in Computer Science, pages 146–157. Springer–Verlag, 1999.

[Bra83] G.W. Brams, editor. *Reseaux de Petri. Theorie et practique.* Masson, 1983.

[Bri95] J.L. Briz. *Técnicas de implementación de redes de Petri.* PhD thesis, Centro Politécnico Superior, Univ. Zaragoza, Ma. de Luna, 3. 50015 Zaragoza, Spain, 1995.

[BS93] U. Buy and R.H. Sloan. A Petri-net-based approach to real-time program analysis. In *Proc. 7th Int Workshop on Sofware Specification and Desing*, pages 56–60, December 1993.

[BS94] U. Buy and R.H. Sloan. Analysis of real-time programs with simple time Petri nets. In *Proc. Int. Symp. on Software Testing and Analysis*, pages 228–239, 1994.

[BS95] U. Buy and R.H. Sloan. Automatic real-time analysis of Ada tasking programs for embeded systems. Technical Report 95–1, Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, 1995.

[But98] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling, Algorithms and Applications.* Kluwer Academic Publishers, 1998.

[BV95] G. Bucci and E. Vicario. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE transactions on Software Engineering*, 21(12):969–992, December 1995.

[BW97] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, second edition, 1997.

[BW98] A. Burns and A. Wellings. *Concurrency in Ada.* Cambridge University Press, second edition, 1998.

[CC85] J. Carlier and Ph. Chretienne. Studies of solutions of scheduling problems associated with timed Petri nets. In *Proc. of 6th European Workshop on Application and Theory of Petri Nets*, pages 319–336, Espoo, Finland, June 1985.

[CC88] J. Carlier and P. Chretienne. Timed Petri net schedules. In G. Rozenberg, editor, *Advances in Petri Nets 1988*, number 340 in Lecture Notes in Computer Science, pages 62–84. Springer Verlag, Berlin, 1988.

[CCG83] J. Carlier, Ph. Chretienne, and C. Girault. Modelling scheduling problems with timed Petri nets. In *Proc. of 4th European Workshop on Application and Theory of Petri Nets*, pages 84–103, Toulouse, France, September 1983.

[CF93] G. Chiola and A. Ferscha. Distributed simulation of timed Petri nets: Exploiting the net structure to obtain efficiency. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, number 691 in Lecture Notes in Computer Science, pages 146–165. Springer Verlag, Berlin, 1993.

[CFGZ93] A. Camurri, P. Franchi, F. Gandolfo, and R. Zaccaria. Petri net based scheduling: A model of the control system of flexible manufacturing systems. *Journals of Intelligent and Robotic Systems*, 8:99–123, 1993.

[CMP87] L. Castellano, G. De Michelis, and L. Pomello. Concurrency vs interleaving: An instructive example. *Bulleting of the European Association for Theoretical Computer Science*, 31:12–15, September 1987.

[CMP98] A. Caloini, G.A. Magnani, and M Pezzé. A technique for the designing robotic control systems based en petri nets. *IEEE Transactions on Control Systems Technology*, 6(1):72–87, January 1998.

[Col89] J.M. Colom. *Análisis Estructural de Redes de Petri, Programación Lineal y Geometría Convexa*. PhD thesis, Universidad de Zaragoza., María de Luna 3 E-50015 Zaragoza, España., 1989.

[CR83] J.E. Coolahan and N. Roussopoulos. Timing requirements for time-driven systems using augmented Petri nets. *IEEE transactions on Software Engineering*, 9(5):603–616, September 1983.

[CSV86] J.M. Colom, M. Silva, and J..L. Villarroel. On software implementation of Petri Nets and Colored Petri Nets using high level concurrent languages. In *Proc. of 7th European Workshop on Application and Theory of Petri nets*, pages 207–241, Oxford, England, January 1986.

[DBDS94] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Transaction on Software Engeeniering Methodology*, 3(4):340–380, December 1994.

[Dij68] E.W. Dijkstra. Goto statements considered harmful. *Communications of the ACM*, 11(3):147–149, March 1968.

[FGP93] M. Felder, C. Ghezzi, and N. Piezzé. High-level timed Petri nets as a kernel for executable specifications. *Real-Time Systems*, (5):235–248, 1993.

[Gio90] R. Di Giovanni. Petri nets and software engineering: Hood nets. In *Proc. of 11th International Conference on Application and Theory of Petri Nets*, pages 123–138, Paris, France, June 1990.

[GJBL88] J.L.M. Grevet, L. Jandura, J. Brode, and A.H. Levis. Execution strategies for Petri net simulation. In *Proc. of the 12th IMACS World Congress on Scientific Computation*, pages 511–516, Paris, France, July 1988.

[GL81] H.J. Genrich and K. Lautenbach. System modeling with high level Petri nets. *Theoretical Computer Science*, (13):109–136, 1981.

[GMMP91] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A unified high-level Petri net formalism for time-critical systems. *IEEE transactions on Software Engineering*, 17(2):160–172, February 1991.

[GMP93] C. Ghezzi, S. Morasca, and M. Pezze. Validating timing requirements for TB net specifications. Internal report, Politecnico di MILANO, March 1993.

[GS88] J. Goodenough and L. Sha. The priority cieling protocol: a method for minimizing the blocking of high priority Ada tasks. In *Proc. of the second International Workshop on Real-Time Ada Issues*. ACM SIGAda, 1988.

[GS97] R.K. Gedela and S.M. Shatz. Modelling of advanced tasking in Ada-95: A Petri net perspective. In *Proc. 2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems, PSDE'97*, Boston, USA, 1997.

[GSW80] H.J. Genrich and E. Stankiewicz-Wichno. A dictionary of some basic notions of net theory. In *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Sciences*, pages 207–274. Springer-Verlag, Berlin, Germany, 1980.

[GV96] F.J. García and J.L. Villarroel. Modelling and Ada implementation of real-time systems using time Petri nets. In C.A. Pereira and W.A. Halang, editors, *Real Time Programming 1996*, pages 127–132. Pergamon, 1996.

[GV98] F.J. García and J.L. Villarroel. Decentralized implementation of real-time systems using time Petri nets. application to mobile robot control. In D.F. GarcíaÑocetti, editor, *Algorithms and Architectures for Real Time Control 1998*, pages 11–16. Pergamon, 1998.

[GV99] F.J. García and J.L. Villarroel. Translating time Petri net structures into Ada 95 statements. In M. Gonzalez Harbour and J.A. de la Puente, editors, *Reliable Software Technologies - Ada-Europe'99*, number 1622 in Lectures Notes in Computer Science, pages 158–169. Springer-Verlag, 1999.

[GWN91] G.R. Gao, Y.B. Wong, and Q.Ñing. A timed Petri-net model for loop scheduling. In *Proc. of 12th International Conference on Application and Theory of Petri Nets*, pages 22–41, Gjern, Denmark, June 1991.

[Han93] H.M. Hanisch. Analysis of place/transition nets with timed arcs and its application to batch process control. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, number 691 in Lecture Notes in Computer Science, pages 282–299. Springer Verlag, Berlin, 1993.

[Har80] D. Harel. On folk theorems. *Communications of the ACM*, 23(7):379–389, July 1980.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. In *Science of Computer Programing 8*, pages 231–274. North-Holland, 1987.

[HM95] C. Hanen and A. Munier. Cyclic scheduling problem: An overview. In P. Cheretienne, E.G. Coffman, J.K. Lengstra, and Z. Liu, editors, *Scheduling Theory and its Applications*. Wiley, 1995.

[Hoa74] C.A.R. Hoare. Monitors - An operating systems structuring concept. *Communications of the ACM*, 17(10):549–57, October 1974.

[IC87] J.M. Ilie and Ph. Chretienne. Synchronous p/t model. Internal report, MASI, Univ. Pierre-Marie Curie, 1987.

[Jen94] K. Jensen. Coloured petri nets: Basic concepts, analysis methods, and practicl use. In *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1994.

[JM86] F. Jahanian and A.K. Mok. Safety analysis of timing properties of real-time systems. *IEEE transactions on Software Engineering*, 12(9), September 1986.

[JR91] K. Jensen and G. Rozenberg, editors. *High-Level Petri Nets*. Springer-Verlag, 1991.

[Kor92] F Kordon. *Prototypage de systèmes parallèles à partir de rèseaux de Petri colorés.* PhD thesis, Institut Blaise Pascal, Univ. Paris VI, 4, Place Jussieu 75252 PARIS CEDEX 05, 1992.

[KP91] F. Kordon and J.F. Peyre. Process decomposition for Rapid Prototyping of Parallel systems. In *6th International Symposium on Computer and Information Science, Kener, Antalya, Turkey*, 1991.

[KRP+93] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Harbour. *A Practitioner's HandBOOK for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprograming in hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[LS87] N.G. Leveson and J.L. Stolzy. Safety analysis using Petri nets. *IEEE transactions on Software Engineering*, 13(3):386–397, March 1987.

[LSD89] J.P Lehoczky, L. Shaand, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. of th IEEE Real-Time Systems Symposium*, December 1989.

[LSS87] J.P Lehoczky, L. Shaand, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. of 8th IEEE Real-Time Systems Symposium*, pages 261–270, December 1987.

[LW80] J. Leung and J.W. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4), 1980.

[MBV98] P.R. Muro, J.A. Banares, and J.L. Villarroel. Kron: Knowlege representation oriented nets for discrete event systems applications. *IEEE Transactions on Systems Man and Cibernetics*, 28(2), March 1998.

[Mer74] P. Merlin. *A Stud of the Recoverability of Computer Systems*. PhD thesis, University of California, Irvine, CA, USA, 1974.

[MF76] P. Merlin and D.J. Farber. Recoverability of communication protocols. *IEEE transactions on Communication*, 24(9), September 1976.

[MGV99] L. Montano, F.J. Garcia, and J.L. Villarroel. Using the time petri net formalism for specification, validation and code generation in robot control applications. *International Journal of Robotic Research*, aceptado para su publicación, 1999.

[MMP+96] D. Mandrioli, A. Morzenti, M. Pezze, P. San Pietro, and S. Silva. A Petri net and logic approach to the specification and verification of real-time systems. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, number 5 in Trends in Software, pages 135–166. Wiley, 1996.

[Mol82] M.K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, 31(9):913–917, September 1982.

[Mur89] T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4), April 1989.

[Mur96] T. Murata. Temporal uncertainty and fuzzy-timing high level Petri nets. In D.F. Billington and W. Reisig, editors, *Applications and Theory of Petri Nets*, number 1091 in Lectures Notes in Computer Science, pages 11–28. Springer-Verlag, 1996.

[MZGT85] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato. Modeling the Ada task system by Petri nets. *Computer Languages*, 10(1):43–61, 1985.

[NHS83] R.A. Nelson, L.M. Haibt, and P.B. Sheridan. Casting Petri Nets into programs. *IEEE Transactions on Software Engineering*, 9(5):590–602, September 1983.

[NS87] K.Ñielsen and K. Shumate. *Designing Large Real-Time Systems with Ada.* Addison-Wesley, 1987.

[OSW93] K. Onaga, M. Silva, and T. Watanabe. Qualitative analysis of periodic schedules for deterministically timed Petri net systems. *IEICE transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E76-A(4):580–592, 1993.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice Hall, 1981.

[PM95] J.M. Proth and I. Minis. Planning and scheduling based on Petri nets. In M.C. Zhou, editor, *Petri Nets in Flexible and Agile Automation*, pages 109–148. Kluwer Academic Publishers, 1995.

[Pop91] L. Popova. On time Petri nets. *J. Inform. Process. Cybern.*, EIK 27(4):227–244, 1991.

[Ram74] C. Ramachandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD dissertation, Massachussets Inst. of Technology, 1974.

[Rat95] Intermetrics, Inc. *Ada 95 Rationale. The Languaje. The Standard Libraries*, January 1995.

[RCS93] A. Ramirez, J. Campos, and M. Silva. On optimal scheduling in deds. In *Proc. of 1993 IEEE Int. Conf. on Robotics and Automation*, pages 821–826, Atlanta, USA, May 1993.

[Rei88] W. Reisig. Concurrency is more fundamental than interleaving. *Bulleting of the European Association for Theoretical Computer Science*, 35:181–185, June 1988.

[Sac94] K.M. Sacha. Real-time software specification and validation with transnet. *Real-Time Systems*, 6(2):153–172, March 1994.

[SB96] R.H. Sloan and U. Buy. Reduction rules for time Petri nets. *Acta Informatica*, 43:687–706, 1996.

[SC88] S.M. Shatz and W.K. Cheng. A Petri net framework for automated static analysis of Ada tasking behaviour. *The Journal of Systems and Software*, 8:343–359, 1988.

[Sif79] J. Sifakis. Performance evaluation of systems using nets. In W. Brauer, editor, *Applications and Theory of Petri Nets*, number 84 in Lectures Notes in Computer Science, pages 307–319. Springer-Verlag, 1979.

[Sil85] M. Silva. *Las Redes de Petri en la Informática y en la Automática.* AC, Madrid, 1985.

[SLR86] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemtive scheduling. In *Proc. of the 7th IEEE Real-Time Systems Symposium*, pages 181–191, December 1986.

[SMBT90] S.M. Shatz, K. Mai, Ch. Black, and S. Tu. Design and implementation of a Petri net based toolkit for Ada tasking analysis. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):424–441, October 1990.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE transactions on Computers*, 39(9):1175–1185, September 1990.

[SSL89] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, 1989.

[STMD96] S.M. Shatz, S. Tu, T. Murata, and S. Duri. An application of Petri net reduction for Ada tasking deadlock analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1307–1322, December 1996.

[SV82] M. Silva and S. Velilla. Programmable logic controllers and Petri Nets: A comparative study. In *Proc. of the Third IFAC/IFIP Symposium*, Software for Computer Control 1982, pages 83–88. Pergamon Press, 1982.

[Tau88] D. Taubner. On the implementation of Petri Nets. In G. Rozenberg, H. Genrich, and G. Roucairol, editors, *Advances in Petri Nets 1988*, volume 340 of *Lecture Notes in Computer Sciences*, pages 418–439. Springer-Verlag, Berlin, Germany, 1988.

[TSM90] S. Tu, S.M. Shatz, and T. Murata. Applying Petri net reduction to support Ada-tasking deadlock analysis. In *Proc. 11th Int. Conf. on Distributed Computing Systems*, pages 961–103, Paris, France, 1990.

[TYC95] J.J.P. Tsai, S.J. Yang, and Y.H. Chang. Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications. *IEEE transactions on Software Engineering*, 21(1):32–49, January 1995.

[Val86] R. Valette. Nets in production systems. In G Goos and Hartmann, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Sciences*, pages 191–217. Springer-Verlag, Berlin, Germany, 1986.

[VB90] R. Valette and B. Bako. Software implementation of Petri Nets and compilation of rule-based systems. In *11th International Conference on Application and Theory of Petri Nets*, pages 27–29, Paris, France, June 1990.

[VCBA83] R. Valette, M. Courvoisier, J.M. Bigou, and J. Alburkerque. A Petri net based programmable logic controller. In *Proc. of IFIP Conference on Computer Applications in Production Engineering, CAP*, pages 103–116, 1983.

[Vil90] J.L. Villarroel. *Integración Informática del Control en Sistemas Flexibles de Fabricación.* PhD thesis, Universidad de Zaragoza, María de Luna 3 E-50015 Zaragoza, España, 1990.

[Wan90] Y. Wang. *Real-Time Behaviour of Asynchronous Agents.* Lecture Notes in Computer Science, 458. Springer Verlag, Berlin, 1990.

[WY93] T. Watanabe and M. Yamauchi. New prioroty-lists for schduling in timed Petri nets. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, number 691 in Lecture Notes in Computer Science, pages 493–512. Springer Verlag, Berlin, 1993.

[You82] S.J. Young. *Real-Time Languages: Design and Development.* Ellis Horwood, 1982.