BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

TRABAJO DE TESIS

ESPECIFICACIÓN Y VERIFICACIÓN DE CIRCUITOS LÓGICOS MEDIANTE EL ÁLGEBRA DE PROCESOS CIRCAL

QUE PRESENTA PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN.

EL ALUMNO:
ABRAHAM NIEVA DE LA HIDALGA.

ASESOR:
JOSÉ DE JESÚS LAVALLE MARTINEZ.

Contenido

Introducción

Un álgebra de procesos es un formalismo matemático que se emplea para resolver problemas que implican la representación de sistemas concurrentes de tiempo real, como son el flujo de tráfico vehicular, el flujo de líquidos, el ritmo cardiaco, los protocolos de red o los circuitos digitales, por mencionar algunos. Exiten varias álgebras de procesos, las cuales son aplicables para resolver el problema de representar los sistemas antes mencionados, entre ellas podemos mencionar a LOTOS, CCS (Calculus of Comunicating Systems), CSP (Comunicating Sequential Processes) y Circal (Circuit Calculus).

Los motivos para usar un álgebra de procesos, especificamente Circal, para modelar diseños de Circuitos Digitales puede resumirse en dos puntos principales:

- Verificar formalmente que el hardware que deseamos implantar está correctamente diseñado y que responde a las necesidades para las que se crea.
- Como consecuencia de lo anterior, se obtiene un ahorro considerable en el costo de fabricación de hardware, pues permite detectar las falla antes de siquiera elaborar un prototipo físico del producto.

Para poder estudiar la aplicación de las álgebras de procesos en el la especificación y verificación de sistemas digitales, es necesario disponer de una herramienta apropiada para tal efecto. Es posible obtener copia gratuita de una implantación del álgeba de procesos Circal,via ftp, del sitio de internet de la Universidad del Sur de Australia (UNISA), el **Sistema Circal**. Éste Sistema es la implantación del lenguajes XCircal (eXtended Circal), el cual es un lenguaje de alto nivel para especificación de sistemas concurrentes y se basa en el álgebra de proceos Circal. Sin Embargo este sistema requiere entre otras cosas de una máquina mas potente que una PC comercial. Por lo anterior el objetivo de este trabajo de tesis es diseñar e implantar un prototipo basado en Circal que sirva para modelar y verificar circuitos digitales, utilizando las técnicas y ejemplos descritos en la bibliografía.

El contenido se encuentra estructurado como se describe a continuación.

En el primer capítulo se presenta una breve introducción a la modelación de sistemas concurrentes, de la cual se deriva una explicación de como se pueden modelar circuitos de hardware usando un algebra de procesos. Asimismo se establecen los fundamentos teóricos. Se presenta una introducción al Álgebra de Procesos Circal, que incluye la descripción de sus operadores, su semántica operacional, así como los Axiomas que permiten la transformación de procesos equivalentes. Además se incluye el álgoritmo de reescritura que se aplicará en el capítulo 2.

El segundo capítulo trata de las estrategias aplicadas para la elaboración del prototipo en si, con la descripción y justificación del uso de algunos operadores adicionales a los originales de circal.

El tercer capítulo muestra algunos ejemplos de aplicación del prototipo, primero utilizando ejemplos hallados en la bilbiografía propia de circal, para a continuación proceder a probarlo con ejemplos independientes.

Al final se presentan las conclusiones junto con las sugerencias de posibles mejoras y las expectativas para la aplicación del prototipo.

Capítulo 1

Modelación de Sistemas Concurrentes.

El presente capítulo se divide en dos partes, en la primera se presentan los antecedentes para la modelación de sistemas concurrentes y se describe la forma en que Circal ha evolucionado como lenguje de modelación y especificación de sistemas concurrentes. La segunda parte presenta la teoría básica de Circal necesaria para construir el prototipo del lenguaje Circal.

1.1 Antecedentes

La solución de un problema específico con la ayuda de una computadora puede realizarse de distintas maneras, una de ellas es el uso de un modelo que nos permita simular el problema del mundo real y su posible solución. Con este modelo podemos hacer pruebas, para obtener información que nos ayuda a resolver el problema en cuestión. En la actulidad existen diversos productos que utilizando técnicas de modelación son aplicados para resolver problemas reales en distintas áreas de aplicación, entre los mas populares tenemos a las herramientas CAD (Computer Aided Desing), que se aplican en una amplia gama de campos científicos y tecnológicos. Asimismo existen también paquetes muy especializados, que se enfocan a la modelación y solución de probemas complejos y muy especializados.

En general estas aplicaciones nos dan soluciones con un determinado nivel de abstracción, es decir que para solucionar problemas de un tipo especifico, existen distintas aplicaciones para distintos niveles de precisión del mismo problema.

1.1.1 Sistemas Concurrentes.

Al estudiar el comportamiento de sistemas concurrentes complejos como los protocolos de red, el tráfico vehicular, el ritmo cardiaco, el flujo de líquidos o los sistemas digitales, debemos tomar en cuenta que los componentes de estos sistemas interactuan entre si, pero al mismo tiempo mantienen su individualidad dentro del sistema. Por esto la comunicación entre ellos es un aspecto muy importante, pues es mediante ésta que se mantiene la unicidad del sistema como un todo.

Lo ideal sería desarrollar herramientas que permitan modelar sistemas concurrentes complejos en diferentes niveles de abstracción. Una opción para realizar este tipo de herramientas es la aplicación de métodos formales, como las álgebras de procesos, que al mismo tiempo que permiten la modelación de los sistemas nos permiten validar la presición del comportamiento de los mismos. Un álgebra de procesos es un formalismo matemático que puede aplicarse en el análisis riguroso de sistemas concurrentes, que consiste en modelar a los sistemas concurrentes mediante procesos que se comunican entre sí, para cambiar de estado o simplemente para servir como medio de comunicación con otros procesos o incluso con otros sistemas.

Entre los investigadores que iniciaron el estudio de las teorías de modelación mediante procesos se encuentra Robin Milner [Mlr 89]. Para él, los conceptos de comunicación y concurrencia se complementan mutuamente, siendo ambos escenciales para el entendimiento de sistemas dinámicos complejos. De acuerdo con Milner un sistema puede ser modelado como una red de agentes que se comunican entre sí y actúan concurrentemente; a su vez cada uno de estos agentes puede descomponerse en una red similar a la mencionada anteriormente. De esta forma es posible, por ejemplo, visulizar a una empresa como una red de departamentos; del mismo modo cada departamento esta formado por una red de personas que trabajan en él y se comunican entre si para llevar a cabo un trabajo determinado, este trabajo puede ser transmitido a los otros departamentos o incluso a otras empresas, por los canales de comunicación predeterminados para este fin. De igual manera podemos pensar en un sistema de electrónico digital como una red de compuertas lógicas, mismas que se encuentran formadas internamente por redes de transistores, unidas entre si por un sistema de comunicación similar al anterior. El limite para las descomposiciones que podemos hacer respecto a un sistema se encuentra determinado por el nivel de abstracción que deseamos obtener, por ejemplo, mientras para un médico es importante tratar a una persona como un conjunto de organos y tejidos, esta apreciación tal vez no sea tan significativa si lo que estamos estudiando es a la persona como parte de un departamento en una empresa.

Las álgebras de procesos han evolucionado de una manera natural en lenguajes de programación que nos proporcionan una sintaxis con la cual podemos describir algoritmos y codificar secuencias de acciones. Aún así existen diversas formas de aplicar estos lenguajes para modelar sistemas concurrentes.

1.1.2 Modelación de Sistemas Digitales

El proceso de diseño de circuitos digitales se divide en distintos pasos, en la Figura 1.1 se muestran los principales, cada paso describe el mismo circuito de manera distinta, es por esto que hablamos de diferentes niveles de abstracción.

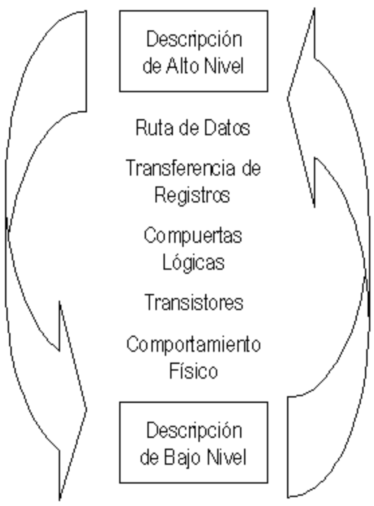


Figura 1.1: Esquema de Niveles de Diseño de Sistemas Digitales

Los diseñadores se auxilian de distintas herramientas en cada paso del diseño. Sin embargo se han creado lenguajes de descripción de hardware (VHDL, XCircal, Tangram, entre otros). El desarrollo de estos lenguajes, su eficiencia demostrada en la aplicación práctica, así como el creciente interés de los investigadores estan encaminados en la actualidad a la adecuación de estos lenguajes a fin de que el proceso de diseño sea completamente automático, lo cual culminaría con la creación de los llamados *Compiladores de Silicón*.

George G. Milne propuso un modelo simple, mediante el cual se podrían modelar sistemas con diferentes niveles de abstracción auxiliandonos de un álgebra de procesos, como medio para realizar las transciciones entre los diferentes niveles de abstracción [Mil+ 85], a continuación se presenta el esquema creado por él (Figura 1.2), que más tarde daría origen al Lenguaje de Descripción de Hardware XCircal.

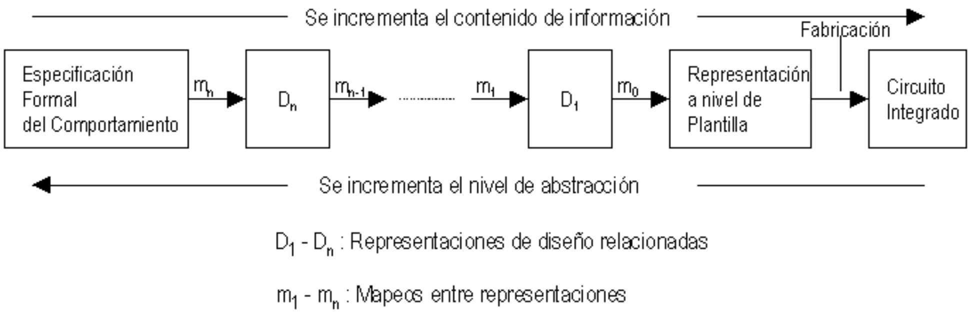
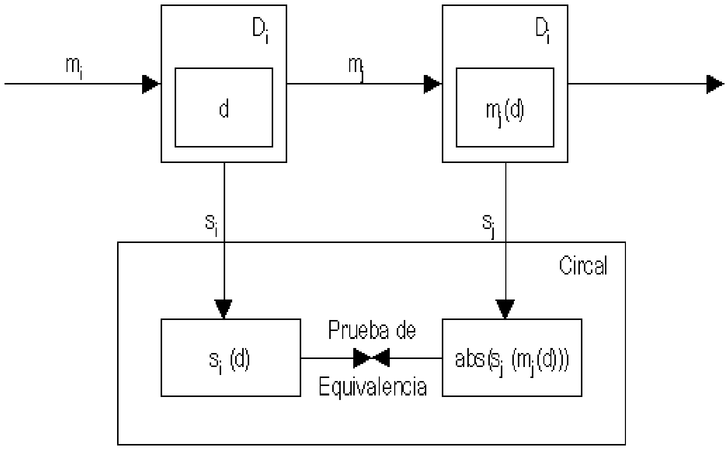


Figura 1.2: Esquema de Mapeos entre Niveles de Representación

En este esquema cada caja representa un nivel de abstracción determinado ($D_1 ... D_n$), cada nivel de abstracción se relaciona con el siguiente por un mapeo ($m_1 ... m_n$), de manera natural el diseño de circuitos encaja en este esquema de diseño. En cada mapeo se debe verificar que la representación del siguiente nivel de abstracción se comporte de la misma manera que su antecesora, en este punto es donde el aplicar el álgebra de procesos nos permite asegurar que dicho comportamiento es el mismo, sin necesidad de realizar un conjunto de pruebas exhaustivas para determinarlo.

En el siguiente gráfico (Figura 1.3) es posible ver como se integra Circal en un paso específico del proceso de verificación de un diseño en un nivel de abstracción D_i al ser modelado como un diseño en un nivel de abstracción consecutivo D_j, para determinar si la conversión se ha hecho o no de forma correcta.



Símbolo	Significado
D_i, D_j	: Niveles de Abstracción i y j
m_i	: Mapeo de D_i a D_j
s_i, s_j	: Funciones Semánticas
$s_i(d)$	: Expresión Circal que representa a d
$abs(s_j(m_j(d)))$	: Diseño en D_j

Figura 1.3: Detalle de Mapeos entre Niveles

Aquí el mapeo m_j aplicado al diseño d, que modela un proceso en el nivel D_i representa la creación de un modelo equivalente de $m(d)$ en el siguiente nivel de abstracción, es decir en D_j. Las funciones semánticas s_i y s_j dan significado a los diseños en los niveles D_i y D_j, respectivamente, al expresar su comportamiento como un modelo circal. Los modelos circal de ambos diseños creados mediante estas funciones semánticas se expresan como $s_i(d)$ y $s_j(m_j(d))$.

Por último se aplica la prueba de equivalencia: $s_i(d) \equiv abs(s_j(m_j(d)))$, la cual indica que m_j mapea correctamente al diseño d en el nivel D_i en un diseño en D_j $(m_j(d))$. Dicho en otros terminos $m_j(d)$ conserva el comportamiento de d en un nivel de abstracción distinto.

Este proceso de modelación puede repetirse tantas veces como sea necesario para realizar transiciones entre los distintos niveles de abstracción y permitir comprobar en cada paso que el criterio de mapeo ha sido aplicado correctamente.

Con las nociones expresadas hasta el momento es posible tener una idea de la complejidad de los esquemas de diseño y verificación que deben usarse, los cuales serán desarrollados y aplicados a partir de la construcción del prototipo tomando como base las definiciones teóricas que a continuación se presentan.

1.1.3 Antecedentes de Circal

La validación del diseño de un circuito, previa a su fabricación, es de gran importancia para los diseñadores por muchas razones, debido a la aplicación cada vez mayor de circuitos en el control de industrias, maquinaria, sistemas de seguridad, etc. Se requiere de circuitos con alto grado de confiabilidad; el costo de fabricación es otro factor importante, pues un circuito que se fabrica erroneamente implica, además del gasto hecho en su fabricación, el alto costo de rediseñar y producir el circuito una vez corregido; por último podemos mencionar la creciente complejidad de los sistemas diseñados, lo que hace que sea cada vez más difícil el control de calidad en el proceso de diseño.

En 1983 Robin Milner y George G. Milne trabajaron en el desarrollo de una teoría para representar sistemas concurrentes complejos mediante procesos, los resultados de este trabajo se publicaron bajo el título de *Concurrent Processes and Their Sintax*. Este primer trabajo más tarde dió origen a la creación de las álgebras de procesos CCS y SCCS (Calculus for Comunicating Systems y Sincronus CCS) por parte de Milner y Circal por Milne [Mil 95]. Desde su concepción, Circal fue diseñado con el propósito de servir como una herramienta básica en la especificación y verificación de circuitos digitales. Muchos investigadores han contribuido al desarrollo y enriquecimiento de Circal. Aunque se hicieron muchos trabajos y prototipos de sistemas despues de la presentación formal de Milne [Mil+ 85], no es sino hasta el trabajo de Faron Moller, *The Semantics of Circal* [Mol 94], donde se revisa la semántica y se determina la necesidad de corregirla, debido a ciertas inconsistencias (La introducción de bloqueos en sistemas donde originalmente no los hay). Es a partir de la publicación de este trabajo que Circal cuenta con una descripción semántica completa y consistente con respecto a la original establecida por Milne.

Además del trabajo de Moller, existe la semántica de bisimulación propuesta por Wenbo Mao [Mao 92]. La semántica desarrollada por Moller es la que más ampliamente ha sido aplicada, al grado que se emplea como base fundamental para el desarrollo del Sistema Circal. Sin embargo, en los trabajos relacionados con ésta, no establece en forma concreta la aplicación de las leyes de Circal, para la verificación y transformación de procesos. En cambio la semántica de bisismulación de Mao, establece las bases teóricas para el desarrollo de un sistema de reescritura basado completamente en la sintaxis y la aplicación de las leyes de Circal.

Las diferencias entre ambas descripciones semánticas son mínimas, obedecen a la necesidad de especificar claramente el funcionamiento de Circal desde distintos puntos de análisis formal. El uso de una u otra, para la elaboración de un sistema de verificación, es indistinto, en cuanto a los resultados de la aplicación, debido a que son equivalentes.

En este trabajo se utilizará la semántica de bisimulación de Mao, por ser la más clara en cuanto a su descripción de la aplicación de las leyes de Circal para la reescritura de términos, ya que describe el algoritmo completo de reescritura y proporciona una alternativa a la reescritura que fácilita la comparación de Procesos para su verificación.

1.2 Fundamentos Teóricos

En esta segunda parte se hace la presentación formal de Circal como un álgebra de procesos, su composición básica; la sintaxis y semántica de sus operadores; los axiomas de igualdad que permiten la transformación de términos equivalentes; la normalización de los términos Circal en base a estos axiomas; una alternativa a la normalización total de los términos mediante el uso de árboles de derivación; y por último la definición del lenguaje que se utilizará como base del prototipo.

1.2.1 Circal

De acuerdo con lo establecido en las secciones previas se desea modelar un tipo específico de sistemas concurrentes mediante procesos, de tal manera que cada sistema concurrente sea representado por un proceso, y a su vez los elementos componentes del sistema, es decir los subsistemas, puedan igualmente ser modelados mediante procesos.

Como se menciono anteriormente los sistemas interactuan con el medio a través de los elementos de comunicación. De igual forma se estableció que un sistema considerado como atómico en algun caso, en otro puede dividirse y estudiarse cada una de sus partes, subsistemas, como un partes integrales de un sistema complejo. Asi mismo se dijo que los sistemas complejos mantienen la unidad entre sus componentes mediante las entidades de comunicación.

En Circal la representación de cada sistema es un proceso. Del mismo modo los elementos o canales de comunicación (tambien conocidos como puertos) se modelan mediante Acciones Atómicas.

El comportamiento y construcción de sistemas estan determinados por las posibles combinaciones de procesos y acciones, para realizar estas combinaciones Circal cuenta con un conjunto básico de operadores. A partir de las acciones, procesos y operadores se pueden construir expresiones más complejas que corresponden a su vez a procesos que son modelos de sistemas más complejos.

A continuación se presentan las definiciones básicas de los elementos de Circal. En la bibliografía que se dispuso para el presente trabajo, no existen definiciones formales para todos los elementos de Circal (por ejemplo la definición de acción), sin embargo con base en las descripciones de los reportes técnicos, y tomando como modelo las específicaciones formales de Milner [Mlr 89], se crean las definiciones para ellos, por lo que si no se establece claramente antes de una definición su procedencia, es por que cae en este rango.

Comenzando por el elemento más sencillo, la acción atómica.

Definición 1.1 Acciones Atómicas
Sea S el Conjuto de Acciones Atómicas, llamado Orden, tal que para cada

$a \in S$, a puede usarse para nombrar uno y solo un puerto de comunicación a la vez.

Una acción atómica es la representeación de un elemento de comunicación que puede ser compartido con más de un proceso. La acción puede ser externa para modelar la interacción con otros procesos. La acción puede ser interna, para modelar la comunicación entre los componentes del sistema.

Definición 1.2 Acción
Para un cojunto de Orden S, al conjunto de todas las acciones posibles se le llama Act_S, tal que para cada $m \in Act_S$, $m \subset S$.

El conjunto de todas las acciones posibles se llama Act_S, y es igual al conjunto potencia de S, es decir 2^S.

La comunicación en general puede transmitirse simultaneamente por más de un puerto (cada puerto es modelado por una acción atómica), es decir se forman subconjuntos del orden, a estos subconjuntos se les llama simplemente acciones. Estas acciones son también conocidas como eventos, y en lo subsecuente se usarán ambas denominaciones de manera indistinta.

Definición 1.3 Procesos
Sea Proc un conjunto de procesos, tal que para cada $P \in Proc$, P puede usarse para nombrar uno y solo un sistema a la vez

Un Proceso es el modelo de un sistema cuyo comportamiento se compone o puede ser descrito mediante acciones discretas.

Cada Proceso Circal puede descomponerse a su vez en Procesos más simples, cada uno de los cuales mantiene una identidad propia, el nivel de abstracción que se desea para resolver un problema es el que determina al mismo tiempo la forma y número de descomposiciones que se den a un Proceso.

Los operadores de Circal tienen la función de ser los conectores entre procesos, que dan significado al comportamiento y estructuración de los mismos. A continuación se describe a estos operadores, esta descripción se basa en la que hace Wenbo Mao [Mao 92], con la adición de los operadores de Definición ($\leftarrow$) y Reetiquetamiento ($P[a/b]$). Donde P, Q y X son procesos, a y b acciones atómicas y m una acción.

- Δ Delta, que modela un proceso terminado, es decir que no responderá a acción alguna.

- $P + Q$ Elección determinística, que modela un proceso que puede comportarse como P o como Q, dependiendo del contexto en que se encuentra.

- $P \oplus Q$ Elección no determinística, que modela un proceso que puede comportarse como P o como Q, independientemente del contexto en que se encuentra.

- mP Guardia o Espera, que modela a un proceso que ante la ocurrencia de la acción m se transforma en el proceso P.

- $X \leftarrow P$ Definición, denota que la variable X modela un proceso que se comportará como P, es decir que responderá a las mismas acciones.

- $P * Q$ Composición, que denota a un proceso que es el resultado de la sincronización de las acciones de los procesos P y Q, es decir siempre que P se transforma ante la ocurrencia de alguna acción, Q se transforma también.

- $P - a$ Abstracción, denota al proceso que se comporta exactamente como P, con excepción de aquellas acciones donde se encuentre el elemento a.

- $P[a/b]$ Reetiquetación, implica que el proceso P al sincronizarse con el evento b , actuará exactamente de la misma manera que lo haría si se sincronizará con el evento a.

De acuerdo con Milne [Mil 94], [Mil 95], los operadores se pueden agrupar en dos conjuntos: Operadores de Comportamiento y Operadores Estructurales.

Los Operadores de Comportamiento, se usan para modelar los cambios e interacción de un proceso determinado. En este conjunto se encuentran los operadores: Delta (Δ), Guardia (mP), Elección Determinística ($+$), Elección Indeterminística ($\oplus$) y Definición ($\leftarrow$).

Los Operadores Estructurales, permiten comunicar procesos distintos, así como establecer relaciones entre ellos en diferentes niveles de representación.

Dentro de este conjunto estan los operadores: de Abstracción ($-$), de Composición Concurrente ($*$) y de reetiquetamiento ($P[a/b]$).

Esta presentación de los operadores de Circal incluye a los operadores de Definición y Reetiquetamiento, que no forman en sí parte de la semántica formal definida por Mao. Sin embargo la necesidad de aplicar el álgebra de procesos en la forma de un lenguaje de programación justifica su uso. Además la base de Circal se mantiene, ya que su semántica no es modificada, es decir no eliminan o substituyen regla alguna.

Los procesos complejos construidos a partir de las Acciones, Procesos y Operadores de Circal puede usarse para modelar completamente el comportamiento de un sistema concurrente. Sin embargo es necesario agrupar estos procesos de acuerdo al conjunto Orden (conjunto de acciones a las que responden dichos procesos) para poder combinarlos entre si. Para ello Mao [Mao 92], define formalmete a los conjuntos de procesos finitos Circal .

Definición 1.4 Procesos Finitos Circal

Un conjunto de procesos finitos Circal con un orden S es construido por los operadores de Circal, dentro del contexto de Circal $\{\Delta_S, +, \oplus, *, -\} \cup Act_S$*, De la siguiente manera:*

i) El símbolo Δ_S pertenece al conjunto.

ii) Si el símbolo P está en el conjunto, entonces para cada elemento a en 2^S, el símbolo aP está en el conjunto.

iii) Si los símbolos P y Q están en el conjunto, entonces $P + Q$, $P \oplus Q$ y $P * Q$ están en el conjunto.

iv) Si el símbolo P está en el conjunto, entonces para cada elemento λ en S, $P - \lambda$ está en el conjunto.

v) Si el símbolo P está en el conjunto, entonces el símbolo (P) está en el conjunto

Denotamos al conjunto así definido como $Proc_S$.

Cuando un proceso responde a una acción determinada, es decir efectúa una transición ante la ocurrencia de una acción específica, se dice que el proceso se sincroniza con tal acción. La sincronización puede o no provocar un cambio en el estado del proceso.

Para modelar paso a paso la evolución de un sistema a partir de su sincronización con una acción determinada, Mao [Mao 92] presenta la noción de Sistema de transición etiquetado.

Definición 1.5 Sistema de Transiciones Etiquetadas
Un sistema de transiciones etiquetadas es un triple de la forma (Act, Proc, $\rightarrow$), donde:

i) Act es el conjunto de todas las acciones.

ii) Proc es el conjunto de todos los procesos

iii) $\rightarrow$ es la relación $Act \times Proc \times Proc$.

Se escribe $P \overset{a}{\rightarrow} P'$ en lugar de $(a,\ P,\ P') \in\rightarrow$. Y debe interpretarse como: el proceso P ejecuta la acción a y se transforma en el proceso P'.

Para cada $a \in Act$, $\overset{a}{\rightarrow}$ representará la menor transición sobre $Proc$.

Así por ejemplo la transición descrita por $AND0 \overset{\{a1,b1,x1\}}{\rightarrow} AND1$, implica que el proceso representado por $AND0$, ante la ocurrencia del evento $\{a1, b1, x1\}$ se transforma en el evento AND1, de modo que pordriamos decir que la definición de AND0 es igual a $(a1, b1, x1)AND1$.

Esta definición proporciona una manera sencilla de entender y definir la evolución por sincronización de los procesos Circal. Su importancia radica en que mediante ella se define la semántica operacional de Circal.

1.2.2 La Semántica de Circal

Formalmente la semántica operacional para los operadores de Circal que Mao proporciona [Mao 92], está dada en la siguiente definición.

Definición 1.6 Semántica de los Operadores de Circal

Los operadores de Circal y su semántica operacional son especificados por el sistema de transiciones etiquetadas y las reglas de transición de la siguiente manera:

Terminación. *Un operador* Δ *de aridad 0 (constante), modela una máquina en su estado terminal. Por esto, no se da una semántica operacional para este proceso. Aún así se utiliza* $\Delta_S \in Proc_S$ *para denotar que este proceso modela el estado terminal de aquellas máquinas modeladas por procesos en* $Proc_S$*.*

Espera o Guardia. *Para cada acción* $a \subseteq S$*, un operador* $G_a : Proc_S \rightarrow Proc_L$ *de aridad 1 mapea cada proceso* $P \in Proc\_S$ *en un proceso aP. El proceso aP modela a una máquina en un estado tal que la máquina solo se sincroniza con un evento estimulo modelado por la acción a. Al hacer esto puede evolucionar a otro estado, modelado por el proceso P. La semántica operacional de este modelo de espera es:*

$$aP \xrightarrow{a} P$$

cuando $a = \emptyset$ *tenemos:*

$$\emptyset P \xrightarrow{\emptyset} P$$

que puede ser abreviada como la transformación ociosa:

$$P \rightarrow P$$

Notese que por esta transfomación vacía podemos de hecho tener una semántica operacional para el proceso Δ*, la cual es* $\Delta \rightarrow \Delta$*. Esta es la unica transformación que puede ocurrir desde* Δ*.*

Elección. *Dados los procesos P, Q* $\in Proc_S$*, un operador +:* $Proc_S \times Proc_S \rightarrow Proc_S$ *de aridad 2, mapea los procesos P, Q en un proceso P + Q. El proceso P + Q modela a una máquina en un estado tal que la máquina puede sincronizarse con aquellos eventos con los que la máquina modelada por el proceso P se sincroniza, o con aquellos con los que la máquina modelada por Q lo hace. En el primer caso, la máquina evoluciona a aquellos estados modelados por la máquina P; en el segundo caso, la máquina evoluciona a aquellos estados modelados por el proceso Q. La semántica operacional para este modelo de elección es:*

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \quad (elección\ 1)$$

$$\frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'} \quad (elección\ 2)$$

Cuando $P \xrightarrow{a} P'$ *y* $Q \xrightarrow{a} Q'$*, al aplicar las reglas de elección,* $P + Q$ *puede transformarse en* $P + Q \xrightarrow{a} P'$*, o bien puede actuar como* $P + Q \xrightarrow{a} Q'$*. Los resultados de la transformación no son determinables por la sincronización entre el evento* a *y el proceso. Es decir* $P + Q$ *es ahora un proceso no determinístico denotado por* $P' \oplus Q'$*:*

$$\frac{P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} P' \oplus Q'} \quad (elección\ 3)$$

Indeterminismo. *Dados los procesos* $P,\ Q \in Proc_S$*, un operador* $\oplus : Proc_S \times Proc_S \to Proc_S$ *de aridad 2, mapea los procesos* P*,* Q *en un proceso* $P \oplus Q$*. El proceso* $P \oplus Q$ *modela una máquina con un estado inicial que es el conjunto unión de aquellos de la máquina modelada por* P *y de aquellos de la máquina modelada por* Q*. Sabemos que la máquina que tiene más de un estado inicial es una máquina no determinística. Supongase que estimulamos tal máquina con un evento dado, la máquina puede o no sincronizarse con él. Esto es la máquina no puede ser controlada por este evento. En general, la máquina modelada por* $P \oplus Q$ *tiene más de un estado inicial, en tal caso, definitivamente no existe evento alguno con el cual se sincronize (más adelante se presentará el caso de un estado inicial más adelante). Como se usa el conjunto no vacio para modelar un evento, en general, no existe acción* $a \neq \emptyset$*, tal que* $P \oplus Q \xrightarrow{a}$ *ocurre (el caso excepcional se presenta posteriormente). Por otra parte, como una máquina de estado no determinística puede aleatoriamente permanecer en uno de sus estados iniciales sin que ocurra una sincronización con un evento, tal comportamiento puede ser modelado exactamente por:*

$$P \oplus Q \to P \ \ (P \oplus Q \xrightarrow{\emptyset} P) \qquad (Indeterminismo\, 1)$$

$$P \oplus Q \to Q \;\; (P \oplus Q \xrightarrow{\emptyset} Q) \qquad (Indeterminismo\, 2)$$

Notas:

i) *La transformación $P \oplus Q \to Q$ no significa que la máquina modelada por $P \oplus Q$ se encuentra en sincronización con algún "evento vacio". En lugar de esto, solo significa que la máquina puede ser prevenida de sincronizarse con aquellos eventos con los cuales la máquina modelada por P puede sincronizarse. Del mismo modo, la máquina puede ser prevenida de sincronizarse con aquellos eventos con los cuales la máquina modelada por Q puede sincronizarse. Llamamos transformaciones incontrolables a tales transformaciones sin etiqueta.*

ii) *Por medio de las transformaciones incontrolables, un proceso no determinístico puede eventualmente convertirse en un proceso determinístico y por tanto puede ocurrir alguna transformación etiquetada. Por lo tanto fijamos la siguiente restricción: las transformaciones incontroladas tienen mayor prioridad que las transformaciones etiquetadas por una acción no vacia. Las siguientes dos reglas describen esta restricción:*

$$\frac{P \;\to\; P'}{P \;\oplus\; Q \;\to\; P' \;+\; Q} \qquad (Indeterminismo\, 4)$$

$$\frac{Q \;\to\; Q'}{P \;\oplus\; Q \;\to\; P \;+\; Q'} \qquad (Indeterminismo\, 5)$$

iii) *Cuando $P \xrightarrow{a}$ P' y $Q \xrightarrow{a}$ Q', el proceso $P \oplus Q$ debe ejecutar la acción a. Esto es, la máquina modelada por $P \oplus Q$ definitivamente se sincronizará con el evento modelado por a, si la máquina es estimulada por tal evento. Por tanto es necesario introducir una nueva regla de transición para el proceso $P \oplus Q$ en este caso. Por un razonamiento similar al de la nota de las reglas de transición de la elección, se da la siguiente nueva regla de transición:*

$$\frac{P \xrightarrow{a} P' \wedge \; Q \xrightarrow{a} Q'}{P \;\oplus\; Q \xrightarrow{a} P' \;\oplus\; Q'} \qquad (Indeterminismo\, 3)$$

Concurrencia. *Dados los Procesos* $P \in Proc_{S_1}$ *y* $Q \in Proc_{S_2}$, *un operador* $* : Proc_{S_1} \times \Pr oc_{S_2} \to \Pr oc_{S_1 \cup S_2}$ *de aridad 2, mapea a los procesos* P *y* Q *en un proceso* $P * Q$. *El proceso* $P * Q$ *modela a una máquina que puede sincronizarse con aquellos eventos, cada uno de los cuales es la ocurrencia concurrente de dos eventos, tales que las máquinas modeladas por los procesos* P *y* Q *pueden sincronizarse con ellos respectivamente. La semántica operacional para este modelo de concurrencia es:*

$$\frac{P \overset{a}{\to} P',\ Q \overset{b}{\to} Q',\ a \cap S_2 = b \cap S_1}{P * Q \overset{a \cup b}{\to} P' * Q'} \quad (Concurrencia)$$

Nota:

La condición $a \cap S_2 = b \cap S_1$ *en la regla de concurrencia es llamada condición de concurrencia. Con esta condición, la transformación* $P * Q \overset{a \cup b}{\to} P' * Q'$ *puede de hecho considerarse como dos transformaciones ocurriendo simultaneamente en forma separada. Las dos transformaciones son:*

$$P \overset{a \cup (b \cap S_1)}{\to} P' \text{ y } Q \overset{b \cup (a \cap S_2)}{\to} Q'$$

La condición de concurrencia garantiza que las dos trasnsformaciones anteriores ocurran. esto es debido a que por la condición tenemos que:

$$a \cup (b \cap S_1) = a \cup (a \cap S_2) = a \text{ y } b \cup (a \cap S_2) = b \cup (b \cap S_1) = b$$

por tanto ambas transformaciones son de hecho:

$$P \overset{a}{\to} P' \text{ y } Q \overset{b}{\to} Q'$$

pero las dos transformaciones ocurren exactamente al mismo tiempo. En este caso extremo, la condición de concurrencia puede satisfacerse ya sea por $a \cap S_2 = \emptyset$ *o por* $b \cap S_1 = \emptyset$. *Como los procesos pueden transformarse incontrolablemente o permanecer ociosos, entonces por ejemplo, en el caso de* $a \cap S_2 = \emptyset$, *podemos permitir* $Q \to Q'$ *(*Q' *puede ser* Q *mismo para la transición vacia) lo que significa* $b = \emptyset$ *y por lo tanto satisfacer la condición de concurrencia con* $a \cap S_2 = \emptyset = \emptyset \cap S_1$. *La transformación concurrente para el proceso* $P * Q$ *en tal caso es:* $P * Q \overset{a}{\to} P' * Q'$.

Abstracción: *Dado un proceso $P \in Proc_S$ y un símbolo $\lambda \in S$, un operador* $-_\lambda : \Pr oc_S \to \Pr oc_{S-\{\lambda\}}$ *de aridad 1, mapea al proceso P en un proceso $P - \lambda$, que modela una máquina en un estado tal que su comportamiento es similar al de la máquina modelada por P, excepto que sólo puede sincronizarse con eventos en los cuales no aparece el símbolo λ concurrentemente. La semántica operacional para este modelo de abstracción es:*

$$\frac{P \xrightarrow{a} P'}{P - \lambda \overset{a - \{\lambda\}}{\rightarrow} P' - \lambda} \quad (Abstracción)$$

Nota:

Cuando $\{\lambda\} = a$, entonces a - $\{\lambda\} = \emptyset$. Por tanto P - λ $\to$P' - λ. Esta transformación no es la misma que la transformación ociosa, sino una transición incontrolable. Esto significa que la operación de abstracción puede introducir indeterminismo en un sistema. Tomando en cuenta que Circal utiliza el orden de los procesos para modelar los puertos de comunicación de las máquinas de estados, entonces al abstraer una etiqueta del orden de un proceso estamos indicando que no se dará información alguna al puerto modelado por dicha etiqueta. El indeterminismo se presenta en el sistema debido a la insuficiencia de información.

Falta solo definir la semántica de los operadores de Definición y Reetiquetación, de acuerdo con los trabajos de McCaskill, Milne, et. al. [McC 91], [McC+ 91] y [McC+ 92], y tomando como modelo la forma en que se Mao presentó la anterior definición (1.6), a continuación se presenta la definición de la semántica de estos dos operadores.

Definición 1.7 Semántica para los Operadores de Definición y Reetiquetación

Los operadores de Definición y Reetiquetación, y su semántica operacional son especificados por el sistema de transiciones etiquetadas y las reglas de transición de la siguiente manera:

Definición. *Dado el Proceso $P \in Proc_S$, $X \in Proc$ un operador $\leftarrow$:$Proc \times Proc_S \leftarrow Proc_S$ de aridad 2, mapea al proceso X en un proceso P, de tal manera que X y P modelan la misma máquina. La semántica operacional para este modelo de definición es:*

$$\frac{P \xrightarrow{a} P', X \leftarrow P}{X \xrightarrow{a} P'} \quad (Definición)$$

Nota:

Cuando se da nombre a un proceso Circal con el uso del operador de definición es posible usar recursivamente el identificador que esta siendo definido dentro de la definición misma, por ejemplo: $P \leftarrow (a,b)P$.

Reetiquetación. *Dado el Proceso $P \in Proc_S$, $a, b \in S$ un operador $[a/b]$:$Proc_S \leftarrow Proc_S$ de aridad 1, mapea al proceso aP en un proceso bP, de tal manera que aP y bP modelan la misma máquina. La semántica operacional para este modelo de Reetiquetación es:*

$$\frac{P \xrightarrow{a} P'}{(aP)[b/a] \xrightarrow{b} P'} \quad (Reetiquetación)$$

Estos dos operadores complementan Circal permitiendo que su uso en la implantación de un lenguaje de programación sea mas sencillo. El operador de Definición permite dar nombre a cualquier proceso; tambien establece la posibilidad de utilizar la recursión para describir el comportamiento de los sistemas. Por otro lado el operador de reetiquetamiento permite cambiar el conjunto de acciones asociado a un proceso, por lo que es posible generar una nueva instancia del proceso pero en diferente contexto.

En la implantación de la semántica definida por Mao [Mao 92], se aplica el concepto de bisimilaridad, que fue definido por Robin Milner [Mlr 89], a fin de construir una relación de equivalencia. Posteriormente demuestra que esta relación genera una congruencia y por lo tanto es posible desarrollar un conjunto de axiomas de igualdad.

Básicamente la bisimilaridad es una relación binaria, por la cual dos procesos P y Q se dicen bisímiles si, encontrandose dentro del mismo conjunto de procesos finitos, responden al mismo conjunto de acciones (es decir que su orden S es el mismo), y ante una acción $a \in Act$, P se transforma en P' y Q se transforma en Q'. Y a su vez P' y Q' son bisímiles entre sí dentro del mismo conjunto de Orden.

Para abarcar todos los estados internos posibles de un proceso, los cuales ocurren despues de todas las transformaciones incontrolables posibles (definición 1.6), Mao crea el concepto de Conjunto de Acepatación.

Definición 1.8 Conjunto de Aceptación

El conjunto de aceptación de un proceso se define como $A(P) = \{S(P') \mid P \Rightarrow P'\}$

donde $S(P) = \{a \mid a \neq \emptyset, P \stackrel{a}{\Rightarrow}\}$ [1] *es llamado conjunto de acciones de sucesores del proceso P.*

El Conjunto de Aceptación de un proceso es de hecho el conjunto de todas las acciones con las que tal proceso puede sincronizarse.

En general cuando se tienen relaciones donde la transformación se indica como $\stackrel{a}{\rightarrow}$, se entiende que para que exista ésta, el proceso debe ejecutar la acción a.

Formalmente Mao [Mao 92] presenta su definición de bisimilaridad de la siguiente manera.

Definición 1.9 Relación de Equivalencia $\sim$

Una relación binaria $\sim \subseteq Proc \times Proc$ *se define : para* $P,\ Q \in Proc$, $P \sim Q$ *si y solo si:*

i $A(P) = A(Q)$

ii Para todo $a \in Act, a \neq \emptyset$,

siempre que $P \stackrel{a}{\mapsto} P'$, [2] *entonces* $\exists Q,\ Q \mapsto Q'$ *y* $P' \sim Q'$;
siempre que $Q \stackrel{a}{\mapsto} Q'$, *entonces* $\exists P,\ P \mapsto P'$ *y* $P' \sim Q'$.

[1] La transición $P \stackrel{a}{\Rightarrow}$ indica que el proceso P puede o no aceptar la sincronización con la acción a y cambiar su estado [Mao 92].

[2] $P \stackrel{a}{\mapsto}$ Indica que en la relación de transición no nos importa si P puede o debe efectuar la acción a.

$(+_0)$	$x + \Delta =_\sim x$
$(+_1)$	$x + x =_\sim x$
$(+_2)$	$x + y =_\sim y + x$
$(+_3)$	$(x + y) + z =_\sim x + (y + z)$
$(\oplus_1)$	$x \oplus x =_\sim x$
$(\oplus_2)$	$x \oplus y =_\sim y \oplus x$
$(\oplus_3)$	$(x \oplus y) \oplus z =_\sim x \oplus (y \oplus z)$
$(+\oplus)$	$x + (y \oplus z) =_\sim (x + y) \oplus (x + z)$
$(+\oplus_{\alpha 1})$	$ax + ay =_\sim ax \oplus ay =_\sim a(x \oplus y)$
$(+\oplus_{\alpha 2})$	$(ax + u) \oplus (ay + v) =_\sim (ax + v) \oplus (ay + u)$

Tabla 1.1: Axiomas de Igualdad

Como se aprecia por la condición i, esta es una relación de bisimulación, en la que se forza a que los conjuntos de aceptación de los procesos que estan probandose sean iguales. Por lo anterior Mao la llama **Bisimulación de Aceptación**.

Una vez obtenidos todos los conceptos teorícos básicos con los que se pueden construir y comparar los procesos Circal a partir de sus elementos, es posible pasar a la presentación de la axiomatización que se desarrolla a partir de estas definiciones.

1.2.3 Axiomas de Igualdad

El conjunto de axiomas nos permite razonar de manera más simple acerca de sistemas complejos, ya que manipulamos los terminos de acuerdo con ellos en lugar de recurrir constantemente la semántica operacional. En la tabla 1.1 se presentan dichos axiomas (donde u, v, x, y, z representan procesos y a representa una acción).

La primera regla $(+_0)$ indica que el operador Delta (Δ) es el neutro aditivo respecto al operador de elección determinística.

$(+_1)$, $(+_2)$, $(+_3)$, $(\oplus_1)$, $(\oplus_2)$ y $(\oplus_3)$, enuncian que ambos operadores ($+$ y $\oplus$) cumplen con las propiedades de idempotencia, conmutatividad y asociación.

$(+\oplus)$ establece la propiedad distributiva de $\oplus$ respecto a $+$.

$$
\begin{array}{ll}
(D_1) & x \oplus y =_\sim x \oplus y \oplus (x+y) \\
(D_2) & (ax+u) \oplus (ay+v) =_\sim (a(x \oplus y)+u) \oplus (a(x \oplus y)+v)
\end{array}
$$

Tabla 1.2: Teoremas Derivados

La regla $(+\oplus_{\alpha 1})$ se deriva directamente de la definición (elección 3).

Por último la regla $(+\oplus_{\alpha 2})$ implica la posibilidad de conmutar miembros con guardias iguales entre sumatorias de elección determinística asociadas mediante operadores indeterminísticos.

A partir de estos axiomas se derivan dos teoremas D_1 y D_2 (Tabla 1.2).

El Axioma D_1 se deriva de la siguiente manera [Mao 92]:

$$
\begin{array}{ll}
x \oplus y & \\
=_\sim (x \oplus y) + (x \oplus y) & por(+_1) \\
=_\sim ((x \oplus y) + x) \oplus ((x \oplus y) + y) & por\ \ + \oplus \\
=_\sim (x + (x \oplus y)) \oplus (y + (x \oplus y)) & por +_2 \\
=_\sim ((x+x) \oplus (x+y)) \oplus ((y+x) \oplus (y+y)) & por\ \ + \oplus \\
=_\sim (x \oplus (x+y)) \oplus ((y+x) \oplus y) & por\ +_1 \\
=_\sim (x \oplus (x+y)) \oplus ((x+y) \oplus y) & por\ +_2 \\
=_\sim (x \oplus ((x+y) \oplus (x+y)) \oplus y) & por\ \oplus_3 \\
=_\sim (x \oplus (x+y) \oplus y) & por\ \oplus_1 \\
=_\sim x \oplus y \oplus (x+y) & por\ \oplus_2
\end{array}
$$

D_1 se utilizará para determinar la máxima expansión de los términos Circal, es decir para alcanzar la forma saturada del término o dicho de otra forma todas las posibles combinaciones que pueden existir al expandir un término, lo que conduce a obtener la Forma Normal Saturada del término. Más aún, si se aplica el teorema en la otra dirección podemos encontrar la forma no saturada, es decir la mínima expresión de un término que conserva el comportamiento del término original.

De igual manera es posible derivar el Axioma D_2 como en la siguiente demostración [Mao 92]:

$$
\begin{array}{ll}
(ax+u) \oplus (ay+v) & \\
=_{\sim} ((ax+u) \oplus (ay+v)) \oplus ((ax+u) \oplus (ay+v)) & \textit{por } \oplus_1 \\
=_{\sim} ((ax+u) \oplus (ay+v)) \oplus ((ax+v) \oplus (ay+u)) & \textit{por } +\oplus_{\alpha 2} \\
=_{\sim} ((u+ax) \oplus (v+ay)) \oplus ((v+ax) \oplus (u+ay)) & \textit{por } +_2 \\
=_{\sim} (u+ax) \oplus ((v+ay) \oplus (v+ay)) \oplus (u+ax) & \textit{por } \oplus_3 \\
=_{\sim} ((u+ax) \oplus (u+ay)) \oplus ((v+ax) \oplus (v+ay)) & \textit{por } \oplus_2 \\
=_{\sim} (u+(ax \oplus ay)) \oplus (v+(ax \oplus ay)) & \textit{por } +\oplus \\
=_{\sim} (u+a(x \oplus y)) \oplus (v+a(x \oplus y)) & \textit{por } +\oplus_{\alpha 1} \\
=_{\sim} (a(x \oplus y)+u) \oplus (a(x \oplus y)+v) & \textit{por } +_2
\end{array}
$$

El teorema D_2 es un refinamiento del axioma ($+\oplus_{\alpha 2}$), que permite que los procesos en espera de la ocurrencia de un evento a, que estén en distintas sumatorias de elección determinística sean iguales.

A partir de estas reglas sencillas es posible manipular todos los términos creados como modelos Circal, ya que como se muestra a continuación (Tablas 1.3 y 1.4) los operadores de concurrencia y abstracción pueden derivarse en términos de sumatorias determinísticas ($+$) y no determinísticas ($\oplus$).

En adelante, la sumatoria

$$\sum_{1 \leq i \leq m} a_i x_i$$

representa una sumatoria de elección determinística, es decir: $a_1x_1 + a_2x_2 + ... + a_mx_m$.

De manera similar la expresión:

$$\bigoplus_{1 \leq i \leq m} a_i x_i$$

representa una sumatoria de elección no determinística ($a_1x_1 \oplus a_2x_2 \oplus ... \oplus a_mx_m$).

Los axiomas $*_1$ y $*_3$, establecen la propiedad conmutativa y asociativa para el operador de concurrencia. $*_2$ establece la propiedad distributiva del operador de Elección Indeterminística respecto al operador de concurrencia.

El axioma $*_4$ define la manera en que una operación concurrente puede ser expresada por medio de sumatorias de elección. Las condiciones inicales

$$
\begin{array}{ll}
(*_1) & x * y =_{\sim} y * x \\
(*_2) & x * (y \oplus z) =_{\sim} (x * y) \oplus (x * z) \\
(*_3) & x * (y * z) =_{\sim} (x * y) * z
\end{array}
$$

$$(*_4)\ para\ x = \sum_{1 \leq i \leq m} a_i x_i \in Proc_L,\ y = \sum_{1 \leq j \leq n} b_j y_j \in Proc_M,$$

$$x * y =_{\sim} \Delta_{L \cup M} + \sum_{\substack{1 \leq i \leq m \\ a_i \cap M = \emptyset}} a_i(x_i * y) + \sum_{\substack{1 \leq j \leq n \\ b_j \cap L = \emptyset}} b_j(x * y_j) + \sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n \\ a_i \cap M = b_j \cap L}} (a_i \cup b_j)(x_i * y_j)$$

Tabla 1.3: Axiomas para el Operador de Concurrencia

expresadas aquí permiten combinar términos de distintos Conjuntos de Procesos Fínitos Circal (aunque $Proc_L$ y $Proc_M$ pueden ser iguales), lo cual implica basicámente que los conjuntos de aceptación pueden ser distintos. Esto en base a la condición de concurrencia expresada en la definición 1.6.

En cuanto a los axiomas presentados en la tabla 1.4, el axioma $-_1$ establece la propiedad de distribución del operador de Abstracción con respecto al de Elección Indeterminística.

El segundo axioma ($-_2$)es muy importante, ya que indica que la aplicación del operador de abstracción puede ocasionar la introducción de indeterminación en un sistema estable, debido a que permite quitar incluso todos los componentes del orden de un proceso.

En los axiomas definidos hasta ahora es importante notar la fuerza del operador de indeterminación ($\oplus$), respecto a los otros operadores, pues de hecho todos son distributivos con respecto a él. Esto determinará la precedencia de los operadores.

Los axiomas de igualdad permiten la manipulación algebraica de los términos Circal, su correcta aplicación nos permite determinar si un proceso determinado es bisimilar a otro (de serlo se entiende que tienen el mismo comportamiento). Sin embargo se debe usar una estrategía determinada que

$$(-_1)\ (x \oplus y) - \lambda =_{\sim} (x - \lambda) \oplus (y - \lambda)$$

$$(-_2)\ (\sum_{1 \leq i \leq m} a_i x_i) - \lambda =_{\sim} \begin{cases} (D + N) \oplus N & si\ \exists\ i\ tal\ que\ a_i = \{\lambda\} \\ D & de\ otro\ modo \end{cases}$$

donde

$$D = \sum_{\substack{1 \leq i \leq m \\ \{\lambda\} \neq a_i}} (a_i - \{\lambda\})(x_i - \lambda)$$

y

$$N = \bigoplus_{\substack{1 \leq i \leq m \\ \{\lambda\} = a_i}} (x_i - \lambda)$$

Tabla 1.4: Axiomas para el Operador de Abstracción

impida la repetición de la aplicación de los axiomas, o que por lo menos la limite; además es necesario saber lo que esperamos obtener de la aplicación de las mismas. En pocas palabras es necesario contar con una forma canónica, de manera que si dos términos distintos en forma pueden reducuirse a la misma forma canónica, podemos decir entonces que son bisimilares. Para este fin Mao [Mao 92] define una forma normal, probando además que todo proceso puede transformarse a la forma normal y que es única. Esto implica que si dos procesos distintos tienen formas normales iguales, entonces son bisimilares entre sí.

Por ello la siguiente sección presenta la Forma Normal Circal definida por Mao, junto con algunas definiciones importantes, que serán de utilidad más adelante durante la implementación del prototipo.

1.2.4 Reescritura de Términos

Antes de la definición de la Forma Normal Circal, es necesario conocer los conceptos de Forma Disyuntiva y Orden Léxico de la Forma Disyuntiva. Por lo que a continuación se presentan ambas, y se justifica su uso para obtener la forma normal.

Definición 1.10 Forma Disyuntiva
Un proceso está en forma disyuntiva, sí es Δ o tiene la forma

$$\bigoplus_{1\leq j\leq m} \Big(\sum_{1\leq i\leq n} a_i^j P_i^j \Big)$$

donde para $1 \leq j \leq m, 1 \leq i \leq n_j, a_i^j \in Act$ *y* P_i^j *es un proceso en forma disyuntiva;*

Recordando que:

$$\sum_{1\leq i\leq n} E_i \ \text{ y } \ \bigoplus_{1\leq i\leq n} Ei$$

denotan las sumatorias de elección deterministica e indeternistica de todas las expresiones E_i mientras i está en el rango de 1 a n, esto es, $E_1 + E_2 + ... + E_n$ y $E_1 \oplus E_2 \oplus ... \oplus E_n$ respectivamente.

Como se puede apreciar por la definición, un proceso en forma disyuntiva es aquel compuesto por expresiones que sólo contienen los operadores, $+$ y $\oplus$. Usando los axiomas es posible llevar cualquier proceso a la forma disyuntiva.

Para entender la definición de Orden Léxico de la Forma Disyuntiva, primero es necesario definir el Orden Léxico de Dos Acciones. Inmediatamente despues se introduce la definición del Orden Léxico de la Forma Disyuntiva.

Definición 1.11 Orden Léxico de Dos Acciones:

Para las acciones $a, b \subseteq L$*, decimos que la acción* a *tiene mayor orden léxico que la acción* b*, si al acomodar ambas acciones en forma de dos palabras con sus miembros en orden alfabético la palabra formada a partir de la acción* a *puede ser puesta antes de la palabra formada de la acción* b *mediante el uso del orden léxico común del diccionario.*

Definición 1.12 Orden Léxico de la Forma Disyuntiva:

Se dice que una forma disyuntiva $\bigoplus_{1\leq j\leq m}(\sum_{1\leq i\leq n_j} a_i^j P_i^j)$ *está en orden léxico, si:*

1. *Para Cualesquiera dos acciones* a_k^j *y* a_l^j *en cada sumatoria de elección* $\sum_{1\leq i\leq n_j} a_i^j P_i^j$ *(para* $1 \leq j \leq m$*) si* $k < l$ *entonces* a_k^j *tien mayor orden léxico que* a_l^j*;*

2. *Dejemos que la palabra* $A^j = (a_1^j, a_2^j ... a_k^j)$ *denote la sumatoria de elección* $a_1^j P_1^j + a_2^j P_2^j + ... + a_k^j P_k^j$ *(para* $1 \leq j \leq m$*), entonces para cualquier* $j_1 < j_2$*, la palabra* A^{j_1} *tiene mayor orden léxico que la palabra* A^{j_2}*.*

De acuerdo con las definiciones anteriores, una Forma disyuntiva está en orden léxico cuando:

- Cada uno de los procesos dentro de las sumatorias determinísticas se encuentran ordenados alfabéticamente con respecto a las acciones que los Guardan.

- Cada una de las sumatorias determinísticas que componen a la sumatoria no determinística se ordenan alfabéticamente con respecto al conjunto de aceptación de cada una de ellas.

A partir de las definiciones anteriores y de la semántica operacional de Circal es posible la definición de la Forma Normal Circal, como se presenta a continuación.

Definición 1.13 Forma Normal Circal

Un proceso Circal está sintácticamente en forma normal, si es una forma disyuntiva $\bigoplus_{1\leq j\leq m} P_j$ *donde para* $1 \leq j \leq m, P_j = \sum_{1\leq i\leq n_j} a_i^j Q_i^j$ *tal que satisface las siguientes condiciones:*

a) Para $1 \leq j \leq m$ *en cada sumatoria de elección* $P_j = \sum_{1\leq i\leq n_j} a_i^j Q_i^j$ *si* $n_{j>1}$ *entonces para* $1 \leq i \leq n_j, a_i^j \neq \emptyset$ *, siempre que* $k \neq l, a_k^j \neq a_l^j$*.*

b) Para $1 \leq j_1,\ j_2 \leq m,\ j_1 \neq j_2$*, siempre que existan* k*,* l *tales que la acción* $a_k^{j_1}$ *sea guardia de un subtérmino en una sumatoria de elección* P_{j_1} *y la acción* $a_l^{j_2}$ *sea guardia de un subtérmino en otra sumatoria de elección* P_{j_2}*, y sean tales que* $a_k^{j_1} = a_l^{j_2}$*, entonces ambos subtérminos en espera son el mismo.*

c) *Para* $1 \leq j_1, j_2 \leq m$, *siempre que* $j_1 \neq j_2$, *entonces* $S(P_{j_1}) \neq S(P_{j_2})$; *y sí* $S(P_{j_1}) \nsubseteq S(P_{j_2}) \nsubseteq S(P_{j_1})$, *entonces existe* l, $1 \leq l \leq m, l \neq j_1, l \neq j_2$ *tal que* $P_l =_{\sim} P_{j_1} + P_{j_2}$*es también una sumatoria de elección en* $\bigoplus_{1 \leq j \leq m} P_j$.

d) *El término* $\bigoplus_{1 \leq j \leq m} P_j$ *está en orden léxico.*

e) *Para* $1 \leq j \leq m, 1 \leq i \leq n_j$, *cada* Q_i^j *está en forma normal.*

La condición *a)* sirve para limitar la existencia de acciones terminales en las sumatoria de elección determinística. Porque $\emptyset P = P = \Delta$, ya que un proceso que no espera acción alguna (un proceso ocioso) puede considerarse como terminado, se puede eliminar de cualquier elección por la aplicación del axioma $+_0$.

b) es usado para garantizar que no hay ambiguedad en cuanto a los procesos que estan en espera de la misma acción. Esta parte se cumple con la aplicación del axioma $+_2$, $\oplus_2$ y el teorema D_2.

En la tercera condición *c)*, basada en la aplicación del teorema D_1, se busca obtener la maxima expansión del término.

En *d)*, simplemente se ordena el término resultante en orden léxico a fin de estandarizar la forma normal.

Por último *e)*, implica que todo subtérmino dentro de la Forma Normal debe también ser a su vez una Forma Normal.

Para abreviar el proceso de reescritura en orden léxico y reducir la expansión de los términos resultantes, Mao propone aquí una alternativa al orden léxico de la forma normal, que es la creación de un árbol de mapeo. En la siguiente sección se define como se construye este árbol.

1.2.5 Árbol de Mapeo para la Forma Normal.

De acuerdo con Mao, el Árbol de Mapeo de la Forma Normal (AMFN) se construye de la siguiente manera:

Definición 1.14 Árbol de Mapeo de la Forma Normal
El árbol de mapeo de un proceso se construye como:

- *La raíz del árbol es el conjunto de aceptación del proceso.*
- *Cada acción a, que está en guardia o espera, etiqueta una rama que va de la raíz a un subárbol que se forma a partir de un proceso, que a su vez es el resultado de la indeterminación ($\oplus$) de todos los procesos en espera de esa misma acción a.*

La construcción del árbol de mapeo que sugiere Mao **??** como una alternativa para la representación y comparación de procesos Circal en Forma Normal es realmente sencilla, a la vez que facilita grandemente la comparación de términos, y mantiene la propiedad de bisimilaridad de los mismos.

1.2.6 Definición Formal del Lenguaje.

A partir de las nociones y conceptos adquiridos con la teoría previa, sólo resta definir formalmente el lenguje que usará e interpretará el prototipo. Formalmente el lenguaje Circal se define de acuerdo con las siguientes reglas.

Definición 1.15 El Lenguaje Circal
El Lenguaje Circal se define de acuerdo con los elementos que lo conforman de la siguiente manera: **Símbolos:**

- *Letras minúsculas $\{a...z\}$.*
- *Letras mayúsculas $\{A...Z\}$.*
- *Números $\{0...9\}$.*
- *Operadores Δ, $+$, $\oplus$, $\leftarrow$, $*$, $-$ y $/$.*
- *Agrupadores (,), [,], { y }.*
- *Delimitadores coma (,) y punto (.).*

La combinación de estos símbolos para construir cadenas bajo Circal obedece a lo siguiente:

Acciones y Procesos

- **Acciones Atómicas :** *Cualquier cadena alfanumérica que inicie con una letra minúscula (i.e. in0, out1, carga).*

- **Acciones :** *Cualquier conjunto de acciones atómicas agrupadas entre corchetes y separadas por comas (i.e. [in0, in1, out0, out1], [a, b]).*

- **Procesos :** *Cualquier cadena alfanumérica que inice con una letra mayuscula (i.e. AND1, Or2, Sumador).*

Operaciones

Sean a, b acciones atómicas, m una acción y P, Q, X procesos, la forma general de las cadenas para describir operaciones son:

- Δ : *Terminación.*

- $P + Q$: *Elección Determinística (Elección).*

- $P \oplus Q$: *Elección No Determinística (Indeterminismo).*

- mP : *Guardia o Espera.*

- $X \leftarrow Q$: *Definición.*

- $P * Q$: *Composición Concurrente.*

- $P - a$: *Abstracción.*

- $P[b/a]$: *Reetiquetación.*

Sólo son cadenas del Lenguaje Circal las construidas de acuerdo con esta definición.

1.2.7 Algoritmo de Conversión a forma normal

La conversión a forma normal de un término Circal cualquiera, con la aplicación de los axiomas igualdad, se realiza de la siguiente manera.

1. Primero se ejecuta la reetiquetación de acciones, si es que la hay.

2. Se aplican las leyes distributivas del operador de elección no determinística ($\oplus$), con respecto a los operadores de elección ($+$) y concurrencia ($*$).

3. Se transforma el término expandiendo el operador de concurrencia a una secuencia de sumas determinísticas (de acuerdo con el axioma $*_4$).

4. Se realiza la reducción del término con la aplicación de los axiomas $-_1$ y $-_2$ del operador de abstracción.

5. Después de la aplicación de los cuatro pasos anteriores el término queda expresado mediante sumas de elección determinística y no determinística, por lo que a partir de aqui la conversión a la Forma Normal Circal se desarrolla directamente de acuerdo con la definición de Mao.

Para la conversión a la forma normal es necesario considerar que la forma normal es una forma disyuntiva, que solo tiene las operaciones de guardia, elección determinística e indeterminística, por lo que el primer paso es la conversión de las operaciones, con el uso de los axiomas en procesos a sumatorias de elección determinísticas e indeterminísticas. Los cuatro primeros puntos del algoritmo descrito llevan a cabo estas operaciones.

Es necesario que el orden de los procesos que se van a convertir este completamente definido, al mismo tiempo los nombres de las acciones (puertos de comunicación) sean iguales, de otra forma no se puede hablar de comunicación entre procesos. Por ello se realiza el reetiquetamiento de acciones. En general se reetiquetan solo las acciones que serán comunes a más de un proceso de esta forma es posible representar la comunicación. Para esta operación no existen axiomas en la semántica de Mao, sin embargo el realizarla no afecta las propiedades de los procesos, o su semántica operacional, se debe tener cuidado solamente que la reetiquetación alcance los niveles mas internos en los procesos, a fin de que se mantenga la consistencia del sistema.

A continuación es necesaria la distribución de los operadores de elección ($+$) y concurrencia ($*$), respecto al operador de elección no determinística, esto reponde a la necesidad de agrupar, por una parte, a los procesos en sumatorias determinísticas, y por otra parte para estar en posición de realizar la expansión de las operaciones de concurrencia en sumatorias de elección eterminística.

El tercer paso se refiere a la expansión de los operadores de concurrencia, ya agrupados, en sumatorias de elección eterminística. Este paso requiere que los conjuntos de aceptación de los procesos que se estan componiendo tengan acciones en común, lo cual justifica el uso del operador de reetiquetamiento.

El cuarto paso consiste en la reducción de acciones mediante la conversión del operador de abstracción. Esta debe ser posterior a la expansión del operador de concurrencia debido a que es necesario primero realizar la conexión de los puertos que realizarán la comunicación ya que en general la abstracción se utiliza para hacer transparentes estas conexiones, de modo que los procesos compuestos se puedan ver como un proceso unico.

Una vez realizado este paso es posible aplicar directamente la definición 1.11 de la forma normal de Mao [Mao 92].

Al igual que para construir la forma normal, la construcción del árbol de derivación se realiza llevando a cabo los cuatro pasos descritos en el algoritmo anterior. Una vez conseguido el término que consiste sólo de operadores de elección e indeterminación es posible construir el árbol de derivación aplicando la definición 1.11.

La construcción y diseño del prototipo se llevan a cabo en la siguiente sección, tomando estrictamente como base la teoría presentada en este capítulo.

Capítulo 2

Desarrollo del Prototipo

Este Capítulo trata del desarrollo del prototipo para la verificación de circuitos basado en el álgebra de procesos Circal. El desarrollo se dividió en 3 partes, en la primera se crean los procedimietos que permiten llevar un proceso Circal a su forma normal; en la segunda se crean los procedimientos que permiten la comparación entre dos procesos, es decir se establece la bisimilaridad de dos procesos; por último en la tercera parte se perfeccionó la integración de los dos procesos anteriores para conseguir que el prototipo funcione de una manera similar al sistema XCircal, incluyendo las cláusulas que permiten leer archivos directamente del disco, así como las que generan la salida de las opciones de normalización, creación de árboles y comparación de árboles, como archivos de texto en disco; tales archivos de entrada y salida podrán ser, con una pequeña modificación, leidos y procesados en ambiente Latex, para su posterior impresión.

El lenguaje de programación elegido para la implantación del prototipo es Prolog (concretamente SWI Prolog), debido a la eficiencia para el manejo simbólico de expresiones aritméticas y por las facilidades que da la recursión en el analisis de las cadenas que conforman la entrada de datos del prototipo.

2.1 Implantación de la Forma Normal Circal.

El objetivo en esta sección es describir como se realizó la implantación de las cláusulas que convierten una expresión que representa un proceso Circal a su forma normal. Se define la representación de los procesos Circal en prolog y se definen los pasos a seguir para la conversión de un proceso a forma normal.

```
Definición de Acciones
accion(a).
accion(b).
accion(uno).
accion(entrada_1).
Definición de Procesos
proceso(q).
proceso(p).
proceso(and).
proceso(or).
```

Tabla 2.1: Ejemplo de Definición de Acciones y Procesos

Los nombres de procesos y acciones atómicas se representan con cadenas de caracteres alfanuméricos (además del carácter "_") , observando que siempre deben iniciar con una letra y que las letras deben ser minúsculas. Las palabras reservadas que no se pueden usar para denominar acciones o procesos son `oplus, leftarrow, delta,` y `p_PXXXX`; las palabras `oplus, leftarrow` y `delta` se reservan por definir operadores de Circal que identificarán a: Elección No Determinística $\oplus$, Definición $\leftarrow$ y Delta Δ respectivamente; las palabras de la forma `p_PXXXX`, donde `X` se refiere a cualquier carácter numérico (0,1,...,9), es decir `p_P0000, p_P0001, ... , p_P9999`, se reservan por ser utilizadas durante la reescritura de procesos como etiquetas Para definir un nuevo proceso se deben primero definir los procesos y acciones que lo componen como hechos de Prolog, como se muestra en la tabla 2.1.

Prolog permite la modificación de los operadores definidos así como la adición de nuevos operadores [Bar 98] y [Wie 99], para aprovechar esta característica en la especificación de los operadores de Circal se hizó la definición de la tabla 2.2.

Los operadores propios de Circal representados por estas claúsulas se ajustan a las siguientes condiciones, y se presentan en orden descendente respecto a su precedencia.

Para definir guardia o espera se utiliza $\sim$, dentro de los operadores de comportamiento éste tiene la mayor precedencia (indicada por el primer argumento de la claúsula), debido a la fuerte relación entre un proceso y su orden. La razón para utilizar este operador para la guardia ($m \sim P$),

```
:-op(400, xfx, leftarrow).
:-op(450, xfy, ~).
:-op(470, xfy, ^).
:-op(500, xfy, oplus).
:-op(550, xfy, *).
:-op(600, xfy, +).
:-op(650, xfy, -).
```

Tabla 2.2: Definición de Operadores

.

cuando en la sección anterior se definió la guardia como mP es por que Prolog requiere un operador para cualquier operación que se realiza, generando un error ante la ausencia del mismo, sin embargo este operador será transparente al usuario, quien codificará su entrada simplemente como se indica en la sección de integración final del prototipo. La misma razón determina la inclusión del operador de reetiquetamiento (^).

El simbolo `oplus` representa al operador de elección no determinística o de indeterminación ($\oplus$), es el siguiente en precedencia debido a que la forma normal Circal definida por Mao tiene como base una forma disyuntiva.

La composición concurrente es representada por su operador original, es decir *.

El operador de elección determinística o simplemente de elección es $+$, tal como se definio inicialmente.

Del mismo modo el símbolo - representa al operador de abstracción.

Estos operadores, junto con `delta` (Δ), componen el conjunto completo de los operadores de Circal, sin embargo para poder desarrollar la aplicación como un pseudo lenguaje de descripción de hardware fue necesario agregar los operadores de definición `leftarrow` ($\leftarrow$), comparación ($=$) y reetiquetamiento (^). Estos operadores son necesarios para la aplicación práctica de la base teórica de Circal, representada por sus operadores originales.

Utilizar la redefinición de operadores, permitida por Prolog, representa un ahorro considerable al no tener que definir, ademas de su comportamiento, la precedencia de cada operador. Al mismo tiempo facilita la escritura de términos de manera infija, que es más natural y clara para el usuario.

Los eventos se representan por medio de listas de la forma `[accion_1, accion_2, ... , accion_n]`, los eventos que constan de una sola acción pueden representarse sin los corchetes.

Para la reetiquetación se utilizarán listas de la forma `[accion_1/accion_1a, ... , accion_n/accion_na]`, lo que implica simplemente que en cualquier lugar que se encuentre la acción `accion_1` será reemplazada por la `accion_1a`.

La precedencia de los operadores se verá afectada sólo por el uso de paréntesis.

De esta forma los procesos Circal definidos por el usuario se expresan internamente como en los siguientes ejemplos:
La entrada del ususario:
`bP[b/e]+aQ[a/e]`
se representa internamente como:

```
b ~ p^[b/e] + a ~ q ^ [a/e]
```

del mismo modo la entrada:
`[a, c] ([a, c] P + [d] Q) * ( [a, b] R` $\oplus$ `[c] S) - [b, c] )`
se representa internamente como:

```
[a, b] ~ ([a, c] ~ p_P + [d] ~ p_Q)*([a, b] ~ p_R oplus [c] ~ p_S) - [b, c]
```

Es importante hacer estas consideraciones, debido a que los términos a ser estudiados, deberán ajustarse a las especificaciones descritas.

Para dar claridad a la lectura de los términos transformados a su forma normal se utiliza la reetiquetación de los subtérminos más internos con respecto al nivel de abstracción en el cual se esta trabajando, por lo que los resultantes de los pasos de la aplicación de algunos axiomas que provocan estos casos, se ven seguidas por la inmediata reetiquetación del proceso resultante, sin afectar a las acciones que lo preceden (su guardia).

Esta forma de proceder se desarrolla tomando como base el ejemplo presentado en [Mil 94](p.p. 185 - 189), toda vez que este reetiquetamiento

```
0001 normal(([a0,x1,t]~p_IN0+[a1,x0,t]~p_IN1)^[a0/p0,a1/p1,x0/q0,x1/q1]*
0002 ([a0,x1,t]~p_IN0+[a1,x0,t]~p_IN1)^[a0/q0,a1/q1,x0/r0,x1/r1]-[q0,q1],X).
0003 ([a0,x1,t]~p_IN0+[a1,x0,t]~p_IN1)^[a0/p0,a1/p1,x0/q0,x1/q1]*
0004 ([a0,x1,t]~p_IN0+[a1,x0,t]~p_IN1)^[a0/q0,a1/q1,x0/r0,x1/r1]-[q0,q1]
0005 definido(p_P0001 leftarrow p_IN0^[a0/p0, a1/p1, x0/q0, x1/q1]).
0006 definido(p_P0002 leftarrow p_IN1^[a0/p0, a1/p1, x0/q0, x1/q1]).
0007 definido(p_P0003 leftarrow p_IN0^[a0/q0, a1/q1, x0/r0, x1/r1]).
0008 definido(p_P0004 leftarrow p_IN1^[a0/q0, a1/q1, x0/r0, x1/r1]).
0009 definido(p_P0005 leftarrow  (p_P0001*p_P0004)).
0010 definido(p_P0006 leftarrow  (p_P0002*p_P0003)).
0011 definido(p_P0007 leftarrow p_P0005-[q0, q1]).
0012 definido(p_P0008 leftarrow p_P0006-[q0, q1]).
0013 X = [p0, r0, t]~p_P0007+[p1, r1, t]~p_P0008
```

Tabla 2.3: Conversión Interna a la Forma Normal

implica simplemente la definición temporal de los procesos en función de los nombres de variables reservados como se especificó anteriormente (p_PXXXX).

En el apéndice A se incluye el código completo del prototipo, en el se puede ver con detalle el proceso de conversión a Forma Normal Circal, sin embargo para mostrar brevemente su funcionamiento se presenta a continuación la ejecución paso a paso de la conversión del término Circal del ejemplo antes mencionado. En el ejemplo original el término se expresa como:

```
(a.0 x.1 t)IN0 [p/a, q/x ] + (a.1 x.0 t)IN1[p/a,q/x] *
(a.0 x.1 t)IN0 [q/a, r/x ] + (a.1 x.0 t)IN1[q/a,r/x] - q
```

Después de la conversión a la forma normal con la aplicación de los axiomas este término se reduce a:

```
(p.0 r.0 t)SYS00+(p.1 r.1 t)SYS11
```

De acuerdo con la sintaxis del prototipo el término original se expresa internamente como lo muestran las lineas 3 y 4 del ejemplo en la tabla 2.3, en éste se realiza la conversión del término a la forma normal.

Las líneas 1 y 2 son el llamado a la consulta normal, con dos argumentos, el primero es el proceso a reescribir y el segundo es la variable que contendrá

```
0621          normal_1(Proc,ProcF):-
0622                  print(Proc),
0623                  reet(Proc, ProcR),
0624                  dist(ProcR, ProcD),
0625                  comp(ProcD, ProcC),
0626                  abst(ProcC, ProcA),
0627                  quitadelta(ProcA, ProcE),
0628                  r_proc(ProcE, ProcF),
0629                  listing(definido).
```

Tabla 2.4: Cláusula de Forma Normal Circal

la salida. Las líneas 3 y 4 presentan la impresión del término que se va a convertir. A continuación desde la linea 5 hasta la 12 se presentan las definiciones de las etiquetas que se utilizan durante el proceso de conversión. La linea 14 presenta la salida de la consulta, en ella se asigna a la variable X el valor resultante.

A continuación se detalla paso a paso como se construye la forma normal del proceso presentado en este ejemplo, describiendo en cada uno de los pasos el proceso de reescritura y las cláusulas que realizan las operaciones correspondientes. El fragmento del listado fuente del programa en la tabla 2.4 muestra la cláusula que toma como entrada un proceso (`Proc`), y proporciona como salida la forma normal de dicho proceso (`ProcF`).

En línea 622 se manda a imprimir la cláusula de entrada. A partir de aqui se llama sucesivamente a las cláusulas que realizan: la reetiquetación de Acciones, la distribución de los operadores respecto al operador de elección no determinística, la composición de procesos, la abstracción de acciones, la eliminación de las Δ's generadas por los pasos anteriores y la reescritura del proceso en orden léxico. La última línea (629) manda a imprimir las cláusulas donde se definen los procesos auxiliares creados durante la reescritura. En las siguientes secciones se describe cada uno de los pasos de la reescritura.

2.1.1 Reetiquetación de Acciones.

La línea 623 del listado (tabla 2.4) invoca a la cláusula reet, la cual realiza la reetiquetación de las acciones del proceso si es que las hay. El listado de la tabla 2.5 presenta las cláusulas que definen esta operación.

Las cuatro primeras cláusulas (líneas 17 a 30) son para el caso en que la

```
reet(P + Q, PR + QR):-
        reet(P, PR),
        reet(Q, QR).

reet(P oplus Q, PR oplus QR):-
        reet(P, PR),
        reet(Q, QR).

reet(P * Q, PR * QR):-
        reet(P, PR),
        reet(Q, QR).

reet(P - Q, PR - Q):-
        reet(P, PR).

reet(Acc ~ Proc_1^Vars, AccR~Et_Proc) :-
        reet(Proc_1^Vars, Proc_1R),
        evento(Acc, AccE),
        reemp(Vars,AccE,AccR),
        renombra(Proc_1R^Vars,Et_Proc).

reet((Proc_1 + Proc_2)^Vars,Proc_1R + Proc_2R):-
        reet(Proc_1^Vars, Proc_1R),
        reet(Proc_2^Vars, Proc_2R).

reet((Proc_1 * Proc_2)^Vars,Proc_1R * Proc_2R):-
        reet(Proc_1^Vars, Proc_1R),
        reet(Proc_2^Vars, Proc_2R).

reet((Proc_1 oplus Proc_2)^Vars,Proc_1R oplus Proc_2R):-
        reet(Proc_1^Vars, Proc_1R),
        reet(Proc_2^Vars, Proc_2R).

reet((Proc_1 - Acc)^Vars,Proc_1R -Acc):-
        reet(Proc_1^Vars, Proc_1R).

reet(Proc_1 ^ _ , Proc_1):-
        proceso(Proc_1).

reet(Proceso,Proceso).
```

Tabla 2.5: Reetiquetación de Acciones

operación de reetiquetación se encuentre agrupada dentro de alguno de los operandos de los otros operadores, aplicando recursivamente la operación.

La quinta cláusula (líneas 32 a la 36) define la forma en la que se hace la reetiquetación en sí, esto cuando se llega a un proceso de la forma $mP[a/b...]$; La cláusula es invocada recursivamente para verificar, la existencia de más operaciones internas de reetiquetación; a continuación se llama al proceso que verifica que m es una acción válida; una vez comprobado esto se sustituye cualquier miembro de m con su correspondiente definido en el conjunto de reetiquetación ($[a/b...]$). Finalmente se llama al proceso que creará la etiqueta temporal correspondiente, renombra el proceso y adiciona la cláusula que lo define, es esta etiqueta de proceso la que se devolverá junto con el evento modificado.

Las siguientes cuatro cláusulas (líneas 38 a la 51 realizan la distribución del operador de reetiquetación respecto a las otras operaciones a fin de alcanzar todos los niveles de anidación de procesos.

Las cláusulas 10 y 11 (líneas 53 a 56) cierran los ciclos de recursión, la primera para el caso que existe retiquetamiento, pero se alcanza una etiqueta de proceso terminal; la segunda en caso de que se agote la recursión y no exista reetiquetación.

Esta operación no es propia de la semántica de Circal, sin embargo su utilidad práctica en la construcción y definición de procesos dentro del lenguaje, justifica su utilización. Al mismo tiempo su aplicación no modifica las propiedades del leguaje o de los otros operadores.

La aplicación de esta cláusula al proceso del ejemplo (tabla 2.3) da como resultado:

```
([p0, t, q1]~p_P0001+[p1, t, q0]~p_P0002) *
([q0, t, r1]~p_P0003+[q1, t, r0]~p_P0004)-[q0, q1]
```

Donde p_P0001, p_P0002, p_P0003 y p_P0004 se definen como etiquetas auxiliares de procesos como se presenta a continuación.

```
p_P0001 leftarrow p_IN0^[a0/p0, a1/p1, x0/q0, x1/q1]
p_P0002 leftarrow p_IN1^[a0/p0, a1/p1, x0/q0, x1/q1]
p_P0003 leftarrow p_IN0^[a0/q0, a1/q1, x0/r0, x1/r1]
p_P0004 leftarrow p_IN1^[a0/q0, a1/q1, x0/r0, x1/r1]
```

La finalidad de crear estas etiquetas es principalmente la de dar claridad a los procesos resultantes, pues aunque estos se pueden escribir en forma extendida, es más comodo utilizar esta forma de abreviación.

2.1.2 Distribución de Operadores Respecto a $\oplus$.

El siguiente paso para la conversión a la forma normal requiere efectuar la distribución de $\oplus$ con respecto a los demás operadores, ya que como se describió en la sección 1.2.4, la Forma Normal Circal es una forma disyuntiva. El listado de la tabla 2.6 detalla la forma en que se hace la distribución.

La distribución de operadores respecto al operador de Elección no determinística $\oplus$, tiene como finalidad dejar estos operadores en la parte más externa posible dentro de cada nivel de anidación de los procesos, ya que como se ha establecido es necesario obtener primero la forma disyuntiva del proceso, en ella todos sus componentes deben ser a su vez formas disyuntivas también.

Las seis cláusulas buscan encontrar la operación de indeterminación dentro de los operandos del proceso, incluso en los operandos del mismo operador de elección no determinística $\oplus$ (líneas 70 a la 72).

En el caso de los operadores de elección ($+$ líneas 74 a 77) y de composición concurrente ($*$ líneas 83 a 86), se realiza la distribución respecto a $\oplus$, usando las cláusulas: mixup y mixup2, basadas en los axiomas $+\oplus$[1] y $*_2$[2], respectivamente (tabla 2.7).

Por ejemplo con un proceso de la forma $(P \oplus Q) + (R \oplus S)$ la llamada a dist dentro del programa tiene la forma:

```
dist(p_P oplus p_Q + p_R oplus p_S, p_Salida).
```

Esto genera dos llamadas a dist:

[1] $x + (y \oplus z) =_{\sim} (x + y) \oplus (x + z)$
[2] $x * (y \oplus z) =_{\sim} (x * y) \oplus (x * z)$

```
        dist(PRO_A oplus PRO_B, PRO_AR oplus PRO_BR):-
                dist(PRO_A, PRO_AR),
                dist(PRO_B, PRO_BR).

        dist(PRO_A + PRO_B, PRO_R):-
                dist(PRO_A, PRO_AR),
                dist(PRO_B, PRO_BR),
                mixup(PRO_AR, PRO_BR, PRO_R).

        dist(EV ~ PRO, EV_R ~ PRO_R):-
                dist(PRO, PRO_R),
                evento(EV, EV_R).

        dist(PRO_A * PRO_B, PRO_R):-
                dist(PRO_A, PRO_AR),
                dist(PRO_B, PRO_BR),
                mixup(PRO_AR, PRO_BR, PRO_R).

        dist(Pro_A-Ev, Pro_R-Ev_R):-
                dist(Pro_A,Pro_R),
                evento(Ev,Ev_R).

        dist(PRO, PRO):-
                proceso(PRO).
```

Tabla 2.6: Distribución de los Operadores Respecto a $\oplus$

```
mixup(X oplus Y,Z,XZ oplus YZ):-mixup(X,Z,XZ),mixup(Y,Z,YZ).
mixup(X,Y oplus Z,XY oplus XZ):-mixup(X,Y,XY),mixup(X,Z,XZ).
mixup(X,Y,X + Y).

mixup2(X oplus Y,Z,XZ oplus YZ):- mixup2(X, Z, XZ), mixup2( Y, Z, YZ).
mixup2(X,Y oplus Z,XY oplus XZ):- mixup2(X, Y, XY), mixup2( X, Z, XZ).
mixup2(X,Y,X * Y).
```

Tabla 2.7: Distribución de $+$ y $*$ Respecto a $\oplus$

```
dist(p_P oplus p_Q, p_Salida1).
dist(p_R oplus p_S, p_Salida2).
```

donde:

`p_Salida1 = p_P oplus p_Q` y `p_Salida2 = p_R oplus p_S`.

Una vez resueltas estas dos llamadas se procesa la tercera que es una llamada a mixup:

```
mixup(p_P oplus p_Q, p_R oplus p_S, p_Salida).
```

La cláusula mixup es resuelta recursivamente de la siguiente manera:

```
mixup(p_P, p_R oplus p_S, p_Salida1a)
mixup(p_Q, p_R oplus p_S, p_Salida2a)
mixup(p_P, p_R, p_Salida1b)
mixup(p_P, p_S, p_Salida1c)
mixup(p_Q, p_R, p_Salida2b)
mixup(p_Q, p_S, p_Salida2c)
```

La primera llamada a mixup corresponde a la cláusula de la línea 95 del listado 2.4 [3], la descomposición del proceso es posible por la aplicación de las propiedades conmutativa [4] y asociativa [5], del operador $\oplus$.

Del mismo modo las llamadas 2 y 3 a mixup, que corresponden a la cláusula de la línea 96[6] del mismo listado, se basan en estas propiedades y en la propiedad distributiva de + respecto a $\oplus$[7].

Las cuatro últimas llamadas a mixup corresponden a la cláusula de la línea 97[8], en ella solo se denota que se han alcanzado los puntos de cierre y se inicia la solución de la recursión de manera ascendente. La solución de la recursión genera:

---

[3] `mixup(X oplus Y, Z, XZ oplus YZ):- mixup(X, Z, XZ), mixup(Y, Z, YZ)`

[4] $(\oplus_2): \quad x \oplus y =_{\sim} y \oplus x$

[5] $(\oplus_3): \quad x \oplus (y \oplus z) =_{\sim} (x \oplus y) \oplus z$

[6] `mixup(X, Y oplus Z, XY oplus XZ):-mixup(X, Y, XY),mixup(X, Z, XZ).`

[7] $(+\oplus): \quad x + (y \oplus z) =_{\sim} (x + y) \oplus (x + z)$

[8] `mixup(X,Y,X + Y).`

```
p_Salida1b = p_P + p_R
p_Salida1c = p_P + p_S
p_Salida2b = p_Q + p_R
p_Salida2c = p_Q + p_S
p_Salida1a = p_Salida1b oplus p_Salida1c =
(p_P + p_R) oplus (p_P + p_S)
p_Salida2a = p_Salida2b oplus p_Salida2c =
(p_Q + p_R) oplus (p_Q + p_S)
p_Salida = p_Salida1a oplus p_Salida =
(p_P + p_R) oplus (p_P + p_R) oplus (p_Q + p_R) oplus (p_Q + p_R)
```

Una vez formateada correctamente esta salida equivale a:

$(P + R) \oplus (P + R) \oplus (Q + R) \oplus (Q + R)$

El proceso que se sigue para la distribución de $*$ respecto a $\oplus$ es el mismo.

Es evidente, a partir de este ejemplo, la gran ayuda que proporciona el uso de Prolog para la construcción del prototipo. En el regreso de las llamadas recursivas Prolog coloca los paréntesis correspondientes de acuerdo con la precedencia de los operadores (por ello se obtienen las elecciones determinísticas escritas correctamente), y los omite cuando su uso es irrelevante. Asimismo implantar los axiomas de reescritura se hace más sencillo, como se ve esta es casí directa. Sin embargo el usuario esta obligado a usar los paréntesis cuidadosamente para expresar los procesos, ya que por ejemplo los procesos $S * P \oplus Q + R$, $S * (P \oplus Q) + R$ y $(S * (P \oplus Q)) + R$ son equivalentes entre sí (la forma normal para los tres es la misma), mientras que el proceso $S * (P \oplus Q + R)$ no lo es. Esto por el resultado de la distribución y posterior composición de los procesos. Aplicar la distribución a los primeros da como resultado: $((S * P + R) \oplus (S * Q + R)$, y aplicandolo al otro se obtiene: $((S * (P + R)) \oplus (S * (Q + R))$, debido a que el orden de precedencia de estos tres operadores es $+ < * < \oplus$.

La distribución respecto al operador de abstracción se realiza más adelante al aplicar la cláusula abst.

Aplicar esta cláusula al proceso resultante del ejemplo 2.3, no tiene efecto alguno, ya que éste proceso no tiene operaciones de indeterminación y el resultado del paso anterior se conserva intacto.

2.1.3 Expansión del Operador de Composición.

A continuación se procede a convertir las operaciones de composición concurrente a sumatorias de elección. Esto se realiza dividiendo el proceso en dos partes, con la cláusula comp (tabla 2.8), que busca los operadores de composición, una vez que los encuentra pasa el control a la cláusula compone (tabla 2.9), la cual lleva en a cabo la expansión en sí.

La primera cláusula (líneas 108 a la 111), realiza la composición de los operandos, para posteriormente realizar mediante compon la mezcla de los resultados de esa operación. El resto de las cláusulas busca la operación de composión anidada dentro de las otras operaciones.

Para realizar la expansión del operador de composición se utiliza la cláusula auxiliar compone; primero se obtiene el orden de cada uno de los procesos; posteriormente se pasa el orden de cada proceso y los procesos mismos a la cláusula compone; la cláusula compone realiza la composición de los procesos dados.

En la primera etapa se adelgazan las sumatorias de elección, para realizar la composición proceso a proceso (líneas 130 a 138). Cuando se alcanzan los elementos de la sumatoria (procesos de la forma mP) se realizan las operaciones de composición de acuerdo con lo establecido en el axioma $*_4$ [9].

Así cuando se cumple la condición de concurrencia para dos procesos (Definición 1.6)[10]. Se realiza la composición de acuerdo con lo que indica la tercera sumatoria de este axioma, misma que se expresa en el listado de la tabla 2.9 en las líneas 140 a 145.

La otras tres posibilidades se presentan en las siguientes tres cláusulas y se dan cuando la Intersección de la guardia del primero y el orden del segundo es nula (líneas 147 a la 149), que corresponde a la primera sumatoria; cuando la Intersección de la guardia del segundo y el orden del primero es nula (líneas

---

9

$$(*_4) : \; para\ x = \sum_{1 \leq i \leq m} a_i x_i \in Proc_L \;, y = \sum_{1 \leq j \leq n} b_j y_j \in Proc_M$$

$$x*y =_\sim \Delta_{L \cup M} + \sum_{\substack{1 \leq i \leq m \\ a_i \cap M = \emptyset}} a_i(x_i*y) + \sum_{\substack{1 \leq j \leq n \\ b_j \cap L = \emptyset}} b_j(x*y_j) + \sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n \\ a_i \cap M = b_j \cap L}} (a_i \cup b_j)(x_i*y_j)$$

10 Sean P y Q Procesos con ordenes L y M respectivamente tales que para $a \in L$ y $b \in M$ se tiene $a \cap M = b \cap L$

```
0107    comp(P_1*P_2,Compuesto):-
0108            comp(P_1,P_1c),
0109            comp(P_2,P_2c),
0110            compon(P_1c,P_2c,Compuesto).
0111
0112    comp(P_1 oplus P_2, P_1c oplus P_2c):-
0113            comp(P_1,P_1c),
0114            comp(P_2,P_2c).
0115
0116    comp(P_1+P_2, P_1c+P_2c):-
0117            comp(P_1,P_1c),
0118            comp(P_2,P_2c).
0119
0120    comp(P_1-Ev, P_1c-Ev):-
0121            comp(P_1,P_1c).
0122
0123    comp(Ev~P_1, Ev2~P_1c):-
0124            evento(Ev,Ev2),
0125            comp(P_1,P_1c).
0126
0127    comp(P_1,P_1):-
0128            proceso(P_1).
```

Tabla 2.8: Composición Respecto a otros Operadores

151 a la 153), segunda sumatoria; y cuando no se cumple alguna de las tres condiciones anteriores se devuelve solamente delta (líneas 155), corresponde al primer sumando $\Delta_{L\cup M}$.

Esta operación puede generar Δ's, mismas que son eliminadas con la aplicación de la cláusula la_quita, como estas se generan con sumas de elección determinística, su eliminación se fundamenta en los axiomas de idempotencia $+_0$[11] y de conmutatividad $+_1$ [12] del operador $+$ como en el listado de la tabla 2.10.

Cuando se aplica esta cláusula al problema de estudio (ejemplo 2.1) se obtiene como resultado el siguiente proceso.

```
[p0, q1, r0, t]~p_P0005+[p1, q0, r1, t]~p_P0006-[q0, q1])
```

donde p_P0005 y p_P0006 estan dados por:

[11] $(+_0): \; x + \Delta =_\sim x$

[12] $(+_1): \; x + y =_\sim y + x$

```
p_P0005 leftarrow (p_P0001*p_P0004)
p_P0006 leftarrow (p_P0002*p_P0003)
```

2.1.4 Abstracción de Acciones.

La abstracción se realiza de acuerdo con los axiomas $-_1$[13] de distribución de $-$ respecto a $\oplus$ y $-_2$[14] de expansión de $-$, tomando también en cuenta la precedencia de los operadores. El fragmento de listado de las tablas 2.11 y 2.12 muestran la forma en que se implantó esta operación.

Por la naturaleza del axioma de expansión ($-_2$) del operador $-$, la implantación del mismo se dividió en dos partes, en el listado de la tabla 2.11, la primera cláusula (líneas 171 a 173) se determina efectúa la distribución de $-$ respecto a $\oplus$.

En la segunda cláusula (líneas 171 a 173) se busca si existe un proceso tal que su guardia sea igual al evento que se esta abstrayendo; mediante una llamada a act_p se obtiene el Act (conjunto de acciones con los que el proceso puede sincronizarse) del proceso en cuestión; acto seguido member comprueba si dentro de este conjunto (Act) se encuentra una acción igual a la que se esta abstrayendo, de ser así se pasa el control a abst2(tabla 2.12). en caso contrario se continua la expansión y el resultado será simplemente D (como se define en $-_2$).

Las siguientes cláusulas realizan la distribución del operador de abstracción respecto a $+$ (líneas 180 a 182), ya que por la precedencia de los operadores, a menos que se específique mediante paréntesis, $P+Q-a$ equivale a $P-a+Q-a$.

Cuando la descomposición alcanza procesos de la forma $mP-a$, realiza la abstracción del proceso, aplicando la resta de conjuntos $m-a$ con la cláusula

[13] $(-_1)\ (x \oplus y) - \lambda =_\sim (x-\lambda) \oplus (y-\lambda)$

[14]

$$(-_2)\ (\sum_{1\leq i\leq m} a_i x_i) - \lambda =_\sim \begin{cases} (D+N)\oplus N & si\ \exists\ i\ tal\ que\ a_i = \{\lambda\} \\ D & de\ otro\ modo \end{cases}$$

donde

$$D = \sum_{\substack{1\leq i\leq m \\ \{\lambda\}\neq a_i}} (a_i - \{\lambda\})(x_i - \lambda) \quad y \quad N = \bigoplus_{\substack{1\leq i\leq m \\ \{\lambda\} = a_i}} (x_i - \lambda)$$

subtract (líneas 184 a 188). Si se alcanza un proceso de la forma $P - a$, se procede a renombrar y se devuelve el proceso renombrado (líneas 192 a 194).

En el caso de que el proceso sea de la forma $\Delta - a$, se devuelve sólo *Delta*, ya que por su definición Δ no realiza transicion alguna (línea 190).

La última cláusula se ocupa cuando no existe la operación de abstracción (línea 196).

Cuando existe una guardia que sea igual al conjunto de abstracción, se tiene el primer caso del axioma $-_2$, En el que se obtienen dos procesos, D, que contiene la suma determinística de los procesos donde no se cumple esta condición y N que es la suma no determinística de los procesos que si cumplen esta condición. Para los primeros el proceso de reducción del operador es el mismo de la tabla 2.11, abst.

La operación de construir $(D + N) \oplus N$ se realiza con el auxilio de las cláusulas abst2e y pon_operador.

La cláusula abst2e localiza los procesos cuya guardia es igual a la acción que se abstrae (líneas 210 y 211) y los que son diferentes (líneas 213 y 214). En el primer caso devuelve una lista vacía y una lista que contiene a los procesos cuya guardia es vacía, por la aplicación del axioma $-_2$. En el segundo se devuelve una lista con los procesos con guardia no vacía y una lista vacía, aplicando el mismo axioma.

Estas listas se unen en la primera cláusula (líneas 204 y 208), para devolver dos listas, la primera contiene a los procesos que formarán D y la segunda a los que construirán N. Posteriormente en abst2, con la ayuda de pon_operador se intercalan los operadores correspondientes a cada lista, obteneniendo finanlmente el proceso de la forma $(D + N) \oplus N$

Esta cláusula aplicada al problema de estudio (ejemplo 2.1) da como resultado el siguiente proceso.

```
[p0, r0, t]~p_P0007+[p1, r1, t]~p_P0008)
```

donde p_P0007 y p_P0008 están definidos como:

```
p_P0007 leftarrow p_P0005-[q0, q1]
p_P0008 leftarrow p_P0006-[q0, q1]
```

2.1.5 Reducción de Deltas

Al llegar a este punto se puede tener un cierto número de Δ's en las sumatorias de elección determinística, que pueden ser eliminadas mediante la aplicación del axioma $+_0$. Esta reducción se describe en el listado de la tabla 2.13.

Este proceso es relativamente sencillo en comparación con los anteriores, debido a que en este paso ya solo se tienen sumatorias de operadores $+$ y $\oplus$.

En la primera cláusula se buscan las Δ's más internas en las sumatorias elección determinística, ya que no existe una regla para elminar las que se generan con la forma $P \oplus \Delta$.

El resto de las cláusulas buscan las Δ's y las elimina al momento de encontrarlas.

Nuevamente el proceso descrito en el ejemplo de estudio no cambia ante la aplicación de esta cláusula, pues no contiene Δ alguna.

2.1.6 Normalización del Proceso.

Hecho esto es posible la conversión a la Forma Normal ordenando la forma disyuntiva obtenida como lo indica la definición Mao, mediante la consulta r_proc.

La tabla 2.14, presenta el fragmento del listado correspondiente a la cláusula r_proc, que realiza la reescritura del proceso, como en este paso el proceso ya es una forma disyuntiva, compuesto solo por operadores de Elección ($+$) e Indeterminación ($\oplus$), resta el procesarlo de acuerdo con el teorema de Mao para obtener la Forma Normal Circal.

Los procesos se reducen, hasta llegar a procesos de la forma mP, donde P es un identificador de proceso que no puede descomponerse y m puede ser incluso una acción vacía; una vez alcanzada esta condición se inicia la solución de la recursión.

Si el proceso regresa a una llamada de la cláusula 1 (líneas 265 a 267), se obtiene la reescritura de un sólo término de la forma mP.

La segunda cláusula (líneas 269 a 272) realiza la reescritura de los términos con la forma $P + Q$, primero obtiene el resultado de reescribir los operandos. Posteriormente los acomoda en orden léxico, con el auxilio de la cláusula pon_proc, esta recibe a los operandos, al operador respecto al que esta ordenando y devuelve el proceso reescrito correctamente.

La tercera cláusula realiza la reescritura de los términos $P \oplus Q$. Utilizando también la cláusula pon_proc.

La reescritura y ordenamiento real de los términos se realizan dentro de pon_proc, la tabla 2.15 muestra la implantación para el operador de elección $+$. pon_proc busca el lugar de cada proceso dentro de la sumatoria de elección con las primeras cinco cláusulas (líneas 284 a 299), mediante la comparación de las guardias (tomando como base a los axiomas $+_2$ [15] y $+_3$[16]).

También realiza la reducción de términos iguales (líneas 301 a303) con base en el axioma $+_0$[17]. Para los procesos distintos que tienen guardias iguales se aplica el axioma $+\oplus_{\alpha 1}$[18]

De igual forma la cláusula pon proc se usa para acomodar los procesos de las sumatorias no determinísticas en orden léxico y hacer las reducciones necesarias. El listado de la tabla 2.16 muestra como se implanta esta reescritura.

Inicialmente se busca el lugar de cada elemento de la sumatoria indeterminística, mediante las primeras 5 cláusulas, comparando el orden de cada proceso, de acuerdo con la definición de orden léxico dada por Mao [19]

las propiedades conmutativa[20] y asociativa[21] del operador $\oplus$ posibilitan esta reescritura. Al encontrar procesos iguales, se reducen aplicando el axioma $(\oplus_1)$[22], de acuerdo con lo que muestra el listado en las líneas 345 a 348.

---

[15] $(+_2): \quad x + y =_\sim y + x$

[16] $(+_3): \quad (x + y) + z =_\sim x + (y + z)$

[17] $(+_1): \quad x + x =_\sim x$

[18] $(+\oplus_{\alpha 1}): \quad ax + ay =_\sim ax \oplus ay =_\sim a(x \oplus y)$

[19] *Se dice que una forma disyuntiva* $\bigoplus_{1 \le j \le m}(\sum_{1 \le i \le n_j} a_i^j P_i^j)$ *está en orden léxico, si:*

1. *Para Cualesquiera dos acciones* a_k^j *y* a_l^j *en cada sumatoria de elección* $\sum_{1 \le i \le n_j} a_i^j P_i^j$ *(para* $1 \le j \le m$*) si* $k < l$ *entonces* a_k^j *tien mayor orden léxico que* a_l^j*;*

2. *Dejemos que la palabra* $A^j = (a_1^j, a_2^j ... a_k^j)$ *denote la sumatoria de elección* $a_1^j P_1^j + a_2^j P_2^j + ... + a_k^j P_k^j$ *(para* $1 \le j \le m$*), entonces para cualquier* $j_1 < j_2$*, la palabra* A^{j_1} *tiene mayor orden léxico que la palabra* A^{j_2}*.*

[20] $(\oplus_2): \quad x \oplus y =_\sim y \oplus x$

[21] $(\oplus_3): \quad (x \oplus y) \oplus z =_\sim x \oplus (y \oplus z)$

[22] $(\oplus_1): \quad x \oplus x =_\sim x$

Para los procesos distintos que tienen guardias iguales se aplica el axioma $+\oplus_{\alpha 1}$[23] (líneas 350 a 354).

Aplicar esta reescritura, da como resultado un proceso en la Forma Normal Circal.

Así, el paso final del ejemplo de aplicación, da como resultado el proceso de la forma:

```
[p0, r0, t]~p_P0007+[p1, r1, t]~p_P0008)
```

que formateado para la salida da como resultado:
$(\mathtt{p0}, \mathtt{r0}, \mathtt{t})\mathtt{P0007} + (\mathtt{p1}, \mathtt{r1}, \mathtt{t})\mathtt{P0008}$.

Una vez conseguida la reescritura de procesos en Forma Normal, es posible proceder a implementar el árbol de mapeo de la forma normal (AMFN).

2.2 Construcción del Árbol de Mapeo para la Forma Normal.

Para la representación del AMFN en el prototipo se utilizan listas anidadas, cuya estructura se describe a continuación.

El árbol es una lista en donde el primer elemento es la raíz del mismo, formada por una sublista que contiene el conjunto de aceptación del proceso; el segundo elemento es una sublista que contiene las ramas del árbol, en esta cada elemento es una sublista en la cual el primer elemento es la etiqueta que corresponde a dicha rama, mientras que el segundo es el subárbol a que dicha acción conduce; a su vez cada subárbol se construye de la misma forma que el árbol inicial. Las hojas del árbol son subárboles cuyo conjunto de acepatación es vacio (primer elemento) y la sublista de ramas contiene un solo elemento que corresponde a la etiqueta de un proceso.

La cláusula básica para la construcción del AMFN se encuentra en la tabla 2.17, se utiliza el método de parametro acumulante, donde `Arbol_c` es la lista temporal, donde se va acumulando el resultado. cada rama se crea mediante la cláusula ramas, que devuelve la rama formada a partir de la

[23] $(+\oplus_{\alpha 1}): \quad ax + ay =_{\sim} ax \oplus ay =_{\sim} a(x \oplus y)$

sumatoria de elección que representa P, una vez creada la rama es colocada en orden dentro del árbol (`Arbol_c`) con la cláusula p_ramas. Si la rama corresponde a la última sumatoria de elección de la forma disyuntiva, el árbol obtenido de p_ramas se devuelve como resultado, de no ser así se continúa la recursión llamando a árbol con el resto de la forma disyuntiva y el árbol obtenido de p_ramas como parametro acumulante.

Las ramas se crean a partir de las sumatorias de elección determinística, sin embargo se debe buscar si hay anidadas más sumatorias indeterminística, por ello se llama al proceso arbol. En cualquier caso se obtendrá un árbol, si existen más sumatorias anidadas el árbol tendrá varias ramas, en caso contrario el árbol que se obtien es una hoja, que corresponde a la representación de un proceso terminal (tabla 2.18).

Las cláusulas p_ramita y pones_r sirven para colocar los componentes de la rama en su lugar, aunque no es necesario que las ramas esten ordenadas, por claridad y comodidad se orenan, usando las cláusulas sort (tabla 2.19).

Lar ramas creadas mediante la cláusula rama son integradas al árbol utilizando la cláusula p_ramas (Tabla 2.20). Esto se realiza verificando si el orden del nuevo proceso no esta incluido entre las ramas del parametro acumulante, de ser así solo se mezclan las ramas y se devuelve el resultado con el mismo orden del parametro acumulante. En caso contrario se unen y ordenan los conjunto de aceptación y se mezclan las ramas, obteniendo así el nuevo árbol.

La mezcla de las ramas se hace con el fin de evitar incluir ramas ya formadas en el árbol. Este proceso debe recorrer todas las ramas del árbol, por ello icluir de paso las nuevas ramas en orden respecto a su conjunto de aceptación no implica mayor trabajo (tabla 2.21).

Puede apreciarse en la figura 2.1 un ejemplo sencillo que muestra como se forma inrternamente el AMFN de un proceso Circal.

Las consultas de Prolog que ayudan a construir el AMFN, son más sencillas que las utilizadas cuando se ordena el proceso lexicamente. Debido a que el trabajo con los operadores es mínimo. La mayor parte del trabajo se hace con listas y sublistas, que son aun más fáciles de manejar que las cadenas de operaciones con operadores infijos. Además al aplicar esta obtenemos la forma normal compacta, es decir el menor de todos los procesos.

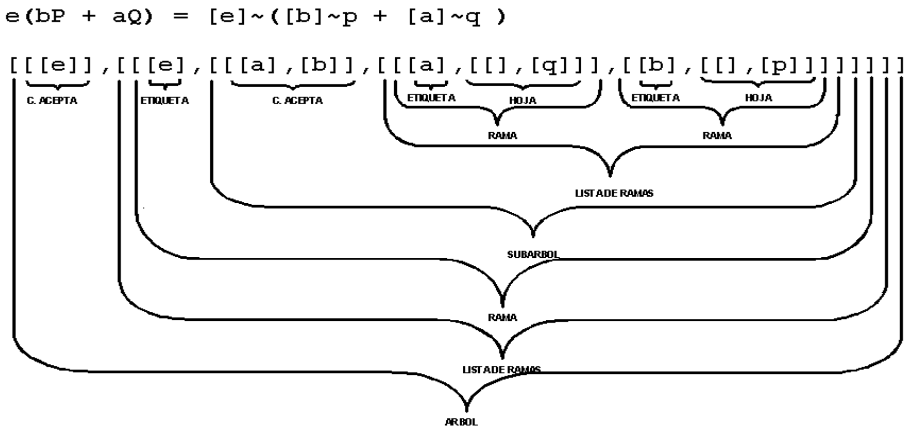


Figura 2.1: Árbol de Mapeo de la Forma Normal

Integrar la creación del AMFN a la normalización se realiza con la simple sustitución de la última cláusula, ordena, por la cláusula arbol. Así el resultado de dicha sustitución aplicado al proceso utilizado para ejemplificar la conversión a forma normal es el siguiente:

```
X =[[[p0, r0, t], [p1, r1, t]], [[[p0, r0, t], [[], [p0007]]],
[[p1, r1, t], [[], [p0008]]]]]
```

Bisimilaridad (Comparación de Procesos)

La bisimilaridad de dos procesos se comprueba al comparar los AMFN de cada uno de ellos, por esto se hace necesario crear las cláusulas para la comparación de dos árboles. A continuación se detalla la forma en que se realiza esta comparación.

De acuerdo con la tabla 2.22, la comparación de dos árboles inicia al comparar el primer elemento de cada uno de los AMFNs, es decir los conjuntos de aceptación, por unificación de los primeros elementos de cada lista (líneas 463 a 468).

Una vez realizada esta verificación se procede a comparar las listas de ramas de cada AMFN, con base en las etiquetas de cada rama que conducen a cada subárbol, también esta comparación se realiza por unificación (líneas 470 a 477). Al llegar a los subárboles se llama recursivamente a la consulta que compara los árboles, pues cada subárbol es a su vez un AMFN del proceso guardado por la etiqueta de la rama.

La condición de paro del proceso recursivo se da al encontrar un subárbol que tenga conjunto de aceptación vacio, es decir al alcanzar las hojas de los árboles.

En la siguiente sección se describe la integración final del prototipo para el análisis automático de procesos descritos de acuerdo con las reglas hasta aqui presentadas, para posteriormente realizar las pruebas del mismo con ejemplos detallados de la utilización del prototipo en la verificación de algunos diseños de circuitos.

2.3 Integración Final del Prototipo

A fin de lograr que la utilización de este prototipo sea lo más cercano posible a un lenguaje formal de descripción de sistemas concurrentes, en esta sección se detallan las cláusulas y reglas para la representación y comparación de procesos.

2.3.1 Consideraciones de Representación Interna

En primer lugar se debe tomar en cuenta el que los procesos complejos se definen a partir de una descripción de los estados en los que se puede encontrar el proceso al sincronizarse con una acción determinada; es decir, si estamos realizando el modelo de una compuerta NOT, tomamos en consideración que esta responde a (se sincroniza con) dos acciones, las cuales son descritas de acuerdo con su tabla de verdad como (a0, x1) y (a1, x0). De acuerdo con el ejemplo de [MLN 95], esto se lleva a cabo como sigue:

```
INV ← (a0, x1)INV + (a1, x0)INV
```

Del mismo modo una compuerta AND es modelada como:

```
AND ← (a0, b0, x0)AND + (a0, b1, x0)AND +
(a1, b0, x0)AND + (a1, b1, x1)AND
```

En estas representaciones se toma la convención por la cual al ocurrir una de estas acciones la compuerta se sincronizará con ella y una vez hecho esto regresa a su estado inicial, en el cual espera nuevamente la ocurrencia de cualquiera de los eventos antes mencionados.

Esto cambia al modelar un proceso que ante la ocurrencia de un evento determinado cambia a un estado distinto al inicial, como por ejemplo un Flip-Flop D, el cual puede retener en su salida un valor 0 o 1, es decir permanece o conmuta entre un estado determinado y otro, esto se modela en Circal mediante dos procesos como se muestra:

```
D0 ← (d0,q0)D0 + (d1,q0)D0 + (d0, clk, q0)D0 + (d1, clk, q1)D1
D1 ← (d1,q1)D1 + (d0,q1)D1 + (d1, clk, q1)D1 + (d0, clk, q0)D0
```

En los ejemplos anteriores se intuye que la representación de procesos es recursiva. A su vez el ejemplo del Flip - Flop D, introduce un concepto muy importante en el diseño digital, la posibilidad de representar más de un estado para un mismo proceso. Mientras los dos primeros ejemplos muestran la representación basada en los cambios en los niveles de voltaje, el ejemplo del Flip-Flop resalta la creación de modelos en base a las transiciones entre los estados de cada proceso. Este modo de representación toma en cuenta el que los procesos no cambian automáticamente a su estado inicial, se encuentran conmutando de un estado a otro mientras se sincronizan con distintos eventos. Así por ejemplo la representación de la compuerta NOT es ahora más compleja, como se ve a continuación:

```
 INV ← (a0, x1, t)INV0 + (a1, x0, t)INV1
INV0 ← (a0, x1, t)INV0 + (a1, x1, t)INV1
INV1 ← (a0, x0, t)INV0 + (a1, x0, t)INV1
```

La representación recursiva de los procesos es utilizada para definir el comportamiento de un proceso de una forma más detallada a fin de conseguir una mayor precisión al realizar la verificación de un diseño.

Para utilizar de manera práctica el prototipo creado en Prolog, aplicando directamente de la Semántica de Bisimulación de Mao, la cual se basa en la comparación sintáctica de los procesos, con una aplicación directa de las definiciones y axiomas de la misma y, al mismo tiempo, aprovechar las facilidades que nos da el ambiente de prolog, la creación de modelos Circal, se dará tomando en cuenta las siguientes convenciones.

1. Las acciones que se utilizan deben ser declaradas previamente.

2. El proceso se modela como una consulta en la que se tiene una lista de parametros de tamaño n+1.

3. Los parametros de 1 hasta n, son la lista de puertos del proceso (acciones átomicas), con todos sus posibles valores.

4. El parametro n representa el proceso en sí.

5. El nombre de la consulta será el nombre al proceso del que se este hablando.

Así por ejemplo la compuerta not puede ser representada como:

```
inv(A0, A1, X0, X1, T,([A0, X1, T] ~ p_INV1 + [A1, X0, T] ~ p_INV0)).
```

De este modo al ejecutar la consulta:

```
inv(a0, a1, x0, x1, t, INV).
```

El resultado que se obtiene es:

```
INV = ([a0, x1, t]~ p_INV1 + [a1, x0, t]~ p_INV0)
```

De este modo el buffer de retardo (DEL) con el que se ha estado trabajando puede construirse como:

```
accion(p0).
accion(p1).
del(A0,A1,X0,X1, DEL):-
 inv(A0, A1, p0, p1, t, INV1),
 inv(p0, p1, X0, X1, t, INV2),
 norma(INV1*INV2-[p0,p1],DEL).
```

Y el resultado de la consulta:

```
del(a0,a1,x0,x1, DEL)
```

sera:

```
DEL = ([a0, x0, t]~ p_P0007 + [a1, x1, t]~ p_P0008)
```

Este resultado concuerda con el comportameniento definido para el elemento DEL.

2.3.2 Entrada y Salida con Formato

Con base en las consideraciones de representación interna de los procesos, es posible determinar las acciones a seguir para crear especificaciones de entrada en formato Latex que al ser leidas se conviertan para poder ser procesadas por el prototipo, generando una salida que del mismo modo será traducida de manera que se pueda guardar en un archivo, para su posterior inclusión en un texto Latex.

Las reglas de codificación son las siguientes:

1. Toda acción que se utilize debe ser declaradas previamente, utilizando cadenas alfanuméricas cuya primera letra debe ser minúscula, e ir precedida por la etiqueta `Accion` (por ejemplo `Accion a1, b0, c1.`).

2. Los nombres de procesos deben declararse antes de poder ser utilizados, se deben usar cadenas alfanuméricas cuyo primer caracter debe ser una literal mayúscula, e ir precedido por la etiqueta `Proceso` (por ejemplo `Proceso P, AND, X.`).

3. El proceso se modela como un parrafo en modo matemático de latex, es decir delimitada por signos `$`, entre el proceso y su descripción se debe tener al identificador $\leftarrow$. Antes de cerrar la descripción con el segundo `$`, se debe tener un punto (.).

4. Los parametros del proceso pueden ser variables, en cuyo caso se representan con cadenas alfanuméricas de letras mayusculas, agrupadas en listas entre corchetes y separadas entre sí por comas. Su posición se encuentra entre nombre del proceso y el identificador $\leftarrow$.

5. El nombre del proceso será igual al nombre de la función que lo declara y deben ser cadenas de caracteres alfanuméricos donde las letras deben ser todas minúsculas.

Así los procesos de prueba de la tabla 2.23 al ser incluido en un documento Latex toma la forma:

Accion a, b, c, d, e, f, g.
Proceso P, Q, R, S.
$pr00 \leftarrow [b]P[b/e] + [a]Q[a/e].$
$pr01 \leftarrow [b]P + [a]Q.$

$pr02 \leftarrow [e]([b]P + [a]Q)$.
$pr03 \leftarrow [e]([b,d]P + [a,c]Q)$.
$pr04 \leftarrow [e]S \oplus ([b,d]P + [a,c]Q)$.

El archivo se carga mediante la consulta `carga`, la cual recibe un sólo argumento, dicho argumento es la localización del archivo que se desea leer desde disco. Este nombre debe ir delimitado entre comillas sencillas (') y los separadores de subdirectorios serán diagonales(/). Así la consulta para cargar el archivo anterior, que tiene como nombre que `prueba01.tex` y se encuentra en el directorio `c:/circal/programa` y debe tener la forma:

```
carga('c:/circal/programa/prueba01.tex').
```

Internamente, el programa queda cargado como se detalla en la tabla 2.24

Una vez cargado en memoria se pueden efectuar operaciones sobre los procesos por ejemplo la consulta:

```
?- pr00(Proceso),
normal(Proceso, FNCP),
salida('c:/circal/programa/salida1.tex', [Proceso, FNCP]).
```

Realiza la conversión del proceso pr00 a su forma normal FNCP, así como la salida con formato del proceso y su forma normal, en un archivo llamado `salida.tex`, en el mismo directorio de origen.

La salida con formato de esta consulta es:

```
$[b]P[b/e] + [a]Q[a/e].$

$[e] P0003.$

$P0001 \leftarrow P[b/e].$

$P0002 \leftarrow Q[a/e].$

$P0003 \leftarrow P0001 \oplus P0002.$
```

que al ser incluida en un documento Latex se ve como:

$[b]P[b/e] + [a]Q[a/e]$.
$[e]P0003$.

$P0001 \leftarrow P[b/e].$
$P0002 \leftarrow Q[a/e].$
$P0003 \leftarrow P0001 \oplus P0002.$

Del mismo modo es posible efectuar operaciones con los procesos definidos, de tal forma que a partir de ellos se pueden construir otros más complejos. Por ejemplo para hacer la composición de los procesos pr02 y pr03, basta con hacer una consulta de la forma:

```
 ?- pr02(Proceso1), pr03(Proceso2),
normal(Proceso1*Proceso2, FNCP),
salida('c:/circal/programa/salida2.tex', [Proceso1*Proceso2, FNCP]).
```

Al igual que el ejemplo anterior, esta consulta produce una salida con formato en disco que al ser incluida en el texto Latex aparece como:

$[e]([b]P + [a]Q) * [e]([b, d]P + [a, c]Q).$
$[e]P0004.$
$P0004 \leftarrow (([b]P + [a]Q) * ([b, d]P + [a, c]Q)).$

Toda operación válida puede efectuarse de la misma manera. Cabe hacer notar que la salida incluirá todos los procesos temporales creados durante las operaciones con procesos desde la última salida a disco, una vez hecha ésta, los procesos temporales son borrados de memoria.

En adelante los ejemplos se presentarán incluidos dentro del texto Latex, tanto para entradas como salidas, sin presentar el código latex de cada uno de ellos.

Con base en estas especificaciones, en el siguiente capítulo se desarrollarán los ejemplos de aplicación del prototipo, con los que se prueba la eficiencia de Circal como auxiliar en el diseño, modelación y verificación de circuitos digitales.

```
compon(P_1,P_2,Compuesto):-
        orden_p(P_1,P_1o),
        orden_p(P_2,P_2o),
        compone(P_1,P_1o,P_2,P_2o,Compuesto).

compone(P_1+Res,P_1o,P_2,P_2o, Compuesto):-
        compone(P_1,P_1o,P_2,P_2o,Comp_1),
        compone(Res,P_1o,P_2,P_2o,Comp_2),
        la_quita(Comp_1,Comp_2,Compuesto).

compone(P_1,P_1o,P_2 + Res,P_2o, Compuesto):-
        compone(P_1,P_1o,P_2,P_2o,Comp_1),
        compone(P_1,P_1o,Res,P_2o,Comp_2),
        la_quita(Comp_1,Comp_2,Compuesto).

compone(A_1~P_1,P_1o,A_2~P_2,P_2o,A_c~Pro_Ren):-
        intersection(A_1,P_2o,I_1),
        intersection(A_2,P_1o,I_1),
        union(A_1,A_2,A_x),
        sort(A_x,A_c),
        renombra(P_1*P_2,Pro_Ren).

compone(A_1~P_1,_,A_2~P_2,P_2o,A_1~Pro_Ren):-
        intersection(A_1,P_2o,[]),
        renombra(P_1*A_2~P_2,Pro_Ren).

compone(A_1~P_1,P_1o,A_2~P_2,_,A_2~Pro_Ren):-
        intersection(A_2,P_1o,[]),
        renombra(A_1~P_1*P_2,Pro_Ren).

compone(_,_,_,_,delta).
```

Tabla 2.9: Composición de Procesos

```
la_quita(X,delta,X).
la_quita(delta,X,X).
la_quita(X,Y,X+Y).
```

Tabla 2.10: Eliminación de Δ.

```
abst(Proc_1 oplus Proc_2-Vars,Proc_1R oplus Proc_2R):-
        abst(Proc_1-Vars, Proc_1R),
        abst(Proc_2-Vars, Proc_2R).

abst(Proc_1-Vars,Proc_1R):-
        act_p(Proc_1, Act),
        member(Vars,Act),
        abst2((Proc_1)-Vars,Proc_1R).

abst(Proc_1 + Proc_2-Vars,Proc_1R + Proc_2R):-
        abst(Proc_1-Vars, Proc_1R),
        abst(Proc_2-Vars, Proc_2R).

abst(Acc ~ Proc_1-Vars, AccR~Proc_1R) :-
        abst(Proc_1-Vars, Proc_1R),
        evento(Acc, AccE),
        evento(Vars,VarsE),
        subtract(AccE,VarsE,AccR).

abst(delta-_,delta).

abst(Proc_1-Vars, Proc_Ren):-
        proceso(Proc_1),
        renombra(Proc_1-Vars,Proc_Ren).

abst(Proc,Proc).
```

Tabla 2.11: Operador de Abstracción 1

```
abst2(Proc_1-Vars,(DS + NS) & NS):-
        abst2e(Proc_1-Vars,D1,N1),
        pon_operador(D1, N1, DS, NS).

abst2e(Proc_1 + Proc_2-Vars, DS, NS):-
        abst2e(Proc_1-Vars, D1, N1),
        abst2e(Proc_2-Vars, D2, N2),
        union(D1, D2, DS),
        union(N1, N2, NS).

abst2e(Acc ~ Proc_1-Acc, [], [[]~Proc_1R]):-
        abst(Proc_1-Vars, Proc_1R).

abst2e(Acc ~ Proc_1-Vars,  [Proc_1R], []):-
        abst(Acc~Proc_1-Vars, Proc_1R).
```

Tabla 2.12: Operador de Abstracción 2

```
quitadelta(X&Y,X1&Y1):-
        quitadelta(X,X1),
        quitadelta(Y,Y1).

quitadelta(X+Y,Z):-
        quitadelta(X,X1),
        quitadelta(Y,Y1),
        la_quita(X1, Y1, Z).

quitadelta(X + delta, X1):-
        quitadelta(X,X1).

quitadelta(delta + Y, Y1):-
        quitadelta(Y,Y1).

quitadelta(X~Y,X1~Y):-
        evento(X,X1),
        proceso(Y).

quitadelta(X~Y,X1~Y1):-
        evento(X,X1),
        quitadelta(Y,Y1).

quitadelta(X,X):-
        proceso(X).

quitadelta(delta,delta).
```

Tabla 2.13: Reducción de Δ.

```
r_proc(EV ~ PRO, EV_R ~ PRO_R):-
        r_proc(PRO, PRO_R),
        evento(EV, EV_R).

r_proc(PRO_A + PRO_B, PRO_R):-
        r_proc(PRO_A, PRO_AR),
        r_proc(PRO_B, PRO_BR),
        pon_proc(PRO_AR, +, PRO_BR, PRO_R).

r_proc(PRO_A oplus PRO_B, PRO_R):-
        r_proc(PRO_A,PRO_AR),
        r_proc(PRO_B,PRO_BR),
        pon_proc(PRO_AR, oplus, PRO_BR,PRO_R).

r_proc(PRO, PRO):-
        proceso(PRO).
```

Tabla 2.14: Normalización 1.

```
pon_proc(Eve~Proc + Resto,+, Proceso, Resultado):-
        pon_proc(Eve~Proc,+,Proceso, Res_tmp),
        pon_proc(Resto,+, Res_tmp, Resultado).

pon_proc(Eve~Proc, +, Eve2~Proc2 + Resto, Eve2~Proc2+Res ):-
        Eve @>Eve2,
        pon_proc(Eve~Proc,+, Resto, Res).

pon_proc(Eve~Proc, +, Eve2~Proc2, Eve2~Proc2+Eve~Proc):-
        Eve @> Eve2.

pon_proc(Eve~Proc, +, Eve2~Proc2 + Res, Eve~Proc + Eve2~Proc2+Res):-
        Eve @<Eve2.

pon_proc(Eve~Proc, +, Eve2~Proc2 , Eve~Proc + Eve2~Proc2):-
        Eve @< Eve2.

pon_proc(Eve~Proc, +, Eve~Proc + Res, Eve~Proc+Res).

pon_proc(Eve~Proc, +, Eve~Proc , Eve~Proc).

pon_proc(Eve~Proc1, +, Eve~Proc + Res, Eve~X+Res):-
        renombra(Proc_1 oplus Proc, X).

pon_proc(Eve~Proc1, +, Eve~Proc , Eve~X):-
        renombra(Proc_1 oplus Proc, X).
```

Tabla 2.15: Orden Léxico de la Elección.

```
pon_proc(Proc_1 oplus Resto, oplus, Proc_2, Resultado):-
        pon_proc(Proc_1, oplus, Proc_2,R_temp),
        pon_proc(Resto, oplus, R_temp, Resultado).

pon_proc(Proc_1, oplus, Proc_2 oplus Resto, Proc_2 oplus Result):-
        act_p(Proc_1,Act_1),
        act_p(Proc_2,Act_2),
        Act_1 @> Act_2,
        pon_proc(Proc_1, oplus, Resto, Result).

pon_proc(Proc_1, oplus, Proc_2 , Proc_2 oplus Proc_1):-
        act_p(Proc_1,Act_1),
        act_p(Proc_2,Act_2),
        Act_1 @> Act_2.

pon_proc(Proc_1, oplus, Proc_2 oplus Resto, Proc_1 oplus Proc_2 oplus Resto):-
        act_p(Proc_1,Act_1),
        act_p(Proc_2,Act_2),
        Act_1 @< Act_2.

pon_proc(Proc_1, oplus, Proc_2 , Proc_1 oplus Proc_2):-
        act_p(Proc_1,Act_1),
        act_p(Proc_2,Act_2),
        Act_1 @< Act_2.

pon_proc(Proc_1, oplus, Proc_1 oplus Resto, Proc_1 oplus Resto).

pon_proc(Proc_1, oplus, Proc_1 , Proc_2).

pon_proc(Act~Proc,oplus,Act~Proc_1oplus Resto,Act~X oplus Resto):-
        renombra(Proc oplus Proc_1, X).

pon_proc(Act~Proc,oplus, Act~Proc_1 , Act~X):-
        renombra(Proc oplus Proc_1, X).
```

Tabla 2.16: Orden Léxico de la Indeterminación

```
arbol(P&Q,Arbol_c, Arbol_r):-
        ramas(P,Ramas_1),
        p_ramas(Arbol_c,Ramas_1,Arbol_c1),
        arbol(Q,Arbol_c1,Arbol_r).

arbol(P,Arbol_c,Arbol_r):-
        ramas(P,Ramas_1),
        p_ramas(Arbol_c, Ramas_1, Arbol_r).
```

Tabla 2.17: Árbol de Mapeo de la Forma Normal

```
ramas(P+Q,La_rama):-
        ramas(P, Rama_1),
        ramas(Q, Rama_2),
        p_ramita(Rama_1,Rama_2,La_rama).

ramas(A~P,[[A],[[A,Arbol]]]):-
        arbol(P,[],Arbol).

ramas(P, [[],[P]]):-
        proceso(P).
```

Tabla 2.18: Creación de una Rama

```
p_ramita([[A1], R1],[A2, R2], [A2, Rama]):-
        member(A1,A2),
        pones_r(R1,R2,Rama).

p_ramita([A1, R1],[A2, R2], [Acc1, Rama]):-
        union(A1,A2,Acc),
        sort(Acc,Acc1),
        union(R1,R2,Ro),
        sort(Ro,Rama).

pones_r([[Ac,P]], [[Ac,Q]|Rest], [[Ac,X]|Rest]):-
        proc_temp(X),
        assert(definido(X <- P & Q)).

pones_r([[Ac,P]], [[Ax,Q]|Rest], [[Ax,Q]|Res]):-
        p_ramita([[Ac,P]], Rest, Res).
```

Tabla 2.19: Creación de una Rama 2

```
p_ramas([C_acepta,Ramas],[C_acepta_1,Ramas_1],[C_acepta, Ramas_n]):-
        member(C_acepta_1,C_acepta),
        mezcla_r(Ramas,Ramas_1,Ramas_n).

p_ramas([C_acepta,Ramas],[C_acepta_1,Ramas_1],[C_acepta_n, Ramas_n]):-
        sort([C_acepta_1|C_acepta],C_acepta_n),
        mezcla_r(Ramas,Ramas_1,Ramas_n).

p_ramas([],[X,Y],[[X],Y]).

p_ramas(X,[],X).
```

Tabla 2.20: Adición de Ramas al Árbol

```
mezcla_r([[A,Arbol_1]|Resto],[[A,Arbol_1]|Resto_1], Ramas_n):-
        mezcla_r(Resto, [[A,Arbol_1]|Resto_1], Ramas_n).

mezcla_r([[A,Arbol_1]|Resto],[[A,Arbol_n1]|Resto_1], Ramas_n):-
        mezcla_r(Resto, [[A,[Arbol_1,Arbol_n1]]|Resto_1], Ramas_n).

mezcla_r([[A,Arbol_1]|Resto],[[A_1,Arbol_n1]|Resto_1],
        [[A_1,Arbol_n1]|Ramas_n]):-
        A@>A_1,
        mezcla_r([[A,Arbol_1]|Resto],Resto_1, Ramas_n).

mezcla_r([[A,Arbol_1]|Resto], [[A_1,Arbol_n1]|Resto_1],
        [[A,Arbol_1]|Ramas_n]):-
        A@<A_1,
        mezcla_r(Resto, [[A_1,Arbol_n1]|Resto_1], Ramas_n).

mezcla_r([],Algo,Algo).

mezcla_r(Algo,[],Algo).
```

Tabla 2.21: Mezcla de las Ramas en el Árbol

```
c_arbol([Acepta_1, Ramas_1], [Acepta_1, Ramas_2]):-
        c_ramas(Ramas_1,Ramas_2).

c_arbol([Acepta_1,Ramas_1], [Acepta_2, Ramas_2]):-
        assert(error(orden,Acepta_1,no_igual_a,Acepta_2)),
        c_ramas(Ramas_1,Ramas_2).

c_ramas([[Eti_1, Arbol_1]|Resto_1], [[Eti_1, Arbol_2]|Resto_2]):-
        c_arbol(Arbol_1,Arbol_2),
        c_ramas(Resto_1,Resto_2).

c_ramas([[Eti_1, Arbol_1]|Resto_1], [[Eti_2, Arbol_2]|Resto_2]):-
        assert(error(guardia, Eti_1, no_igual_a, Eti_2)),
        c_arbol(Arbol_1,Arbol_2),
        c_ramas(Resto_1,Resto_2).

c_ramas([X],[Y]):-
        proceso(X),
        proceso(Y).

c_ramas([],[]).
```

Tabla 2.22: Comapara Dos Árboles

```
% Ejemplos de prueba

Accion a, b, c, d, e, f, g.

Proceso P, Q, R, S, T.

$pr00 \leftarrow [b]P[b/e] + [a]Q[a/e].$

$pr01 \leftarrow [b]P + [a]Q.$

$pr02 \leftarrow [e]([b]P + [a]Q).$

$pr03 \leftarrow [e]([b,d]P + [a,c]Q).$

$pr04 \leftarrow [e]S \oplus  ([b,d]P + [a,c]Q).$
```

Tabla 2.23: Ejemplo de Codificación de Procesos

```
accion(a).
accion(b).
accion(c).
accion(d).
accion(e).
accion(f).
accion(g).

procesos(p_P).
procesos(p_Q).
procesos(p_R).
procesos(p_S).
procesos(p_P).

pr00([b]~p_P^[b/e] + [a]~p_Q^[a/e]).

pr01([b]~p_P + [a]~p_Q).

pr02([e]([b]~p_P + [a]~p_Q).

pr03([e]([b,d]~p_P + [a,c]~p_Q).

pr04([e]~p_S oplus  ([b,d]~p_P + [a,c]~p_Q)).
```

Tabla 2.24: Representación Interna de los Procesos del Ejemplo

Capítulo 3

Aplicación en Ejercicios Prácticos

En este capítulo se desarrollan la pruebas prácticas, con las cuales se verifican los beneficios y facilidades que brinda Circal como auxiliar en el análisis y diseño de circuitos digitales. En la primera sección se define y crea un archivo con las compuertas lógicas básicas, al mismo tiempo se prueban los resultados de la composición sencilla de las compuertas. La segunda sección se refiere al análisis y diseño de compuertas mas complicadas (que se forman a partir de la composición de cuatro o más compuertas). La tercera sección aborda la representación usando distintos niveles de abstracción, mediante el metodo top - down, para la verificación completa de un diseño sencillo. Por último la cuarta sección se refiere a la exploración de la utilidad del presente prototipo de Circal en la representación y estudio de otros sistemas concurrentes, distintios a los circuitos digitales.

3.1 Diseño de Circuitos Digitales

Las compuertas lógicas básicas que se crean para verificar y diseñar circuitos se detallan en la tabla de la tabla 3.1, la creación de cada una de ellas se basa fundamentalmente en la representación de las compuertas como elementos que conmutan entre distintos estados, los estados se definen de acuerdo con sus tablas de verdad establecidas para cada una de las compuertas, tomando como base las compuertas de dos entradas. Posteriormente se verifica la aplicación de las compuertas básicas para la representación de funciones lógicas,

```
Proceso AND.
and2 [A0, A1, B0, B1, X0, X1] ←
[A0, B0, X0] AND +
[A0, B1, X0] AND +
[A1, B0, X0] AND +
[A1, B1, X1] AND.
```

Tabla 3.1: Proceso AND

así como la construcción de las mismas compuertas a través de la composición de otras distintas.

3.1.1 Compuertas Lógicas Básicas.

Se representan las ocho compuertas básicas con las cuales es más común trabajar en el diseño de circuitos digitales [Man 87], en la figura 3.1 se presentan dichas compuertas con las características necesarias para la creación del modelo circal para cada una de ellas.

De acuerdo con los criterios establecidos en la sección anterior, las compuertas pueden ser construidas del mismo modo que la compueta NOT, sin embargo para esta primera parte se implementaron utilizando el enfoque de diseño basado en el nivel de entrada, no en las transiciones, por lo tanto sólo se utiliza un proceso; así la compuerta AND puede ser representada como en el programa de la tabla 3.1.

El nombre del proceso and2, nos indica que se trata de una compuerta AND de dos entradas, la descripción se basa completamente en su tabla de verdad. Las variables utilizadas para denominar los puertos o conexiones(`A0, A1, B0, B1, X0, X1`), permiten crear instancias de compuertas and de dos entradas, cuyos puertos pueden ser nombrados de acuerdo con las necesidades del usuario.

Al ser cargado, el proceso de la tabla 3.1 toma la forma:

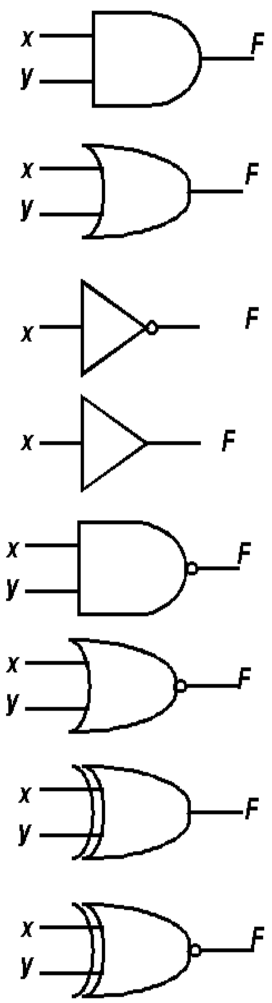
```
and2(A0, A1, B0, B1, X0, X1, AND).
```

Por lo que al ejecutar la consulta:

```
?- and2(a0, a1, b0, b1, x0, x1, AND),
salida('c:/circal/programa/and2.tex', [AND]).
```

Se obtiene como resultado:

$[a0, b0, x0]AND + [a0, b1, x0]AND + [a1, b0, x0]AND + [a1, b1, x1]AND.$



$F = xy$

x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

$F = x + y$

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

$F = x'$

x	F
0	1
1	0

$F = x$

x	F
0	0
1	1

$F = (xy)'$

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

$F = (x + y)'$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

$F = xy' + x'y$

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

$F = xy + x'y'$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

Figura 3.1: Compuertas Básicas

Entrada de archivo

```
accion a0, a1, b0, b1, x0, x1, p0, p1.
Proceso AND, INV. and2 [A0, A1, B0, B1, X0, X1] ← [A0, B0, X0] AND + [A0,
B1, X0] AND + [A1, B0, X0] AND + [A1, B1, X1] AND.
inv [A0, A1, X0, X1] ← [A0, X1] INV + [A1, X0] INV.
```

Consulta

```
?- and2(a0, a1, b0, b1, p0, p1, AND),
inv(p0, p1, x0, x1, INV),
normal(AND*INV-[p0, p1], NAND),
salida('c:/circal/programa/nand2.tex', [AND, INV, NAND]).
```

Salida en archivo nand2.tex

```
[A0, B0, X0] AND + [A0, B1, X0] AND + [A1, B0, X0] AND + [A1, B1, X1]
AND.
[A0, X1] INV + [A1, X0] INV.
[A0, B0, X1] P0002 + [A0, B1, X1] P0002 + [A1, B0, X1] P0002 + [A1, B1,
X0] P0002 .
P0001 ← (AND * INV) P0002 ← (P0001 - [p0, p1])
```

Tabla 3.2: Proceso NAND, modelado con INV y AND

Las demás compuertas se declaran de un modo similar. Los procesos que las definen son: `and2, or2, inv, buf, nand2, nor2, xor2, xnor2`.

3.1.2 Composición de Compuertas Básicas.

Para ejemplificar como se hace la composición de procesos la tabla 3.2 se muestra el modelo de una compuerta NAND, formada con la composición de una compuerta AND y una compuerta NOT, la consulta que realiza la composición y normalización de las compuertas y la salida en archivo que se obtiene con ella.

Las acciones `p0` y `p1` representan los posibles valores que puede tomar la conexión interna p, que se ve gráficamente en la figura 3.2, como esta conexión es interna se abstrae del resultado de la composición, para obtener el resultado deseado, es decir una compuerta con tres puertos (`a,b,x`), cuyo comportamiento pueda ser comparado, mediante la construcción de su AMFN, con el de la compuerta NAND definida por nand2.

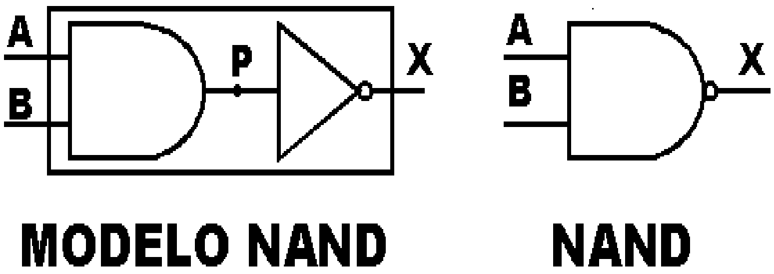


Figura 3.2: Dispositivo NAND

Así el resultado de la consulta es un proceso con un compartamiento identico al de la compuerta NAND definida por nand2, como se puede ver en la tercera parte de la tabla 3.2.

Los resultados pueden ser comparados con la secuencia:

```
?- and2(a0, a1, b0, b1, p0, p1, AND),
inv(p0, p1, x0, x1, INV),
arbol(AND*INV-[p0, p1], [], NAND),
nand2(a0, a1, b0, b1, p0, p1, NAND2),
arbol(NAND2, [], NAND2A),
c_arbol(NAND, NAND2A).
salida('c:/circal/programa/nand2.tex', [NAND, NAND2A]).
```

El resultado contenido en el archivo nand2.tex es:

```
[[[[a0, b0, x1], [a0, b1, x1], [a1, b0, x1], [a1, b1, x0]]], [[[a0, b0,
x1], [[[]], [P0002]]], [[a0, b1, x1], [[[]], [P0002]]], [[a1, b0, x1], [[[]],
P0002]]], [[a1, b1, x0], [[[]], P0002]]]]]
[[[[a0, b0, x1], [a0, b1, x1], [a1, b0, x1], [a1, b1, x0]]], [[[a0, b0,
x1], [[[]], [NAND]]], [[a0, b1, x1], [[[]], [NAND]]], [[a1, b0, x1], [[[]],
[NAND]]], [[a1, b1, x0], [[[]], [NAND]]]]]
```

$\mathtt{P0001} \leftarrow (\mathtt{AND} * \mathtt{INV})$
$\mathtt{P0002} \leftarrow (\mathtt{P0001} - [\mathtt{p0}, \mathtt{p1}])$

Del mismo modo, es posible construir compuertas de 3 o mas entradas, como la compuerta AND del siguiente ejemplo:

```
and2(A0, A1, B0, B1, p0, p1, AND1),
and2(C0, C1, p0, p1, X0, X1, AND2),
normal(AND1 * AND2 - [p0, p1], AND3).
```

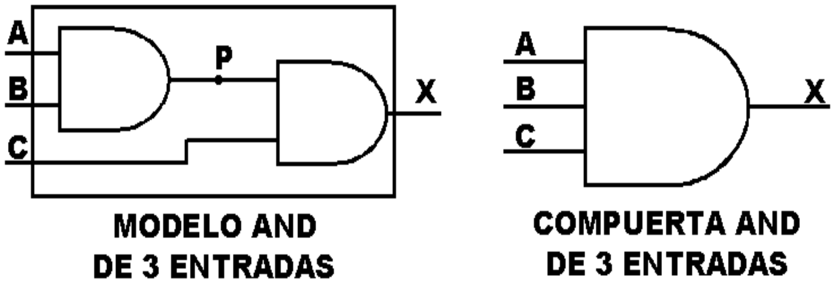


Figura 3.3: Composición AND-AND = AND3

En el diseño de circuitos, así como en el diseño en general de cualquier sistema concurrente, deben considerarse las propiedades de los elementos que se estan modelando, en este caso las compuertas lógicas; ya que por ejemplo las compuertas nand y nor no son asociativas, por lo que el resultado de la combinación de 2 nand de 2 entradas para formar una nand de 3 no produce el resultado esperado para el comportamiento de una compuerta nand de 3 entradas, en tal caso es mejor construirla como la and de 3 entradas anterior, agregando solamente un negador a la salida como se representa en la figura 3.4 que corresponde a:

```
and2(A0, A1, B0, B1, p0, p1, AND1),
and2(C0, C1, p0, p1, X0, X1, AND2),
inv(q0,q1,X0,X1,INV),
normal(AND1 * AND2 * INV - [p0, p1, q0, q1], NAND3).
```

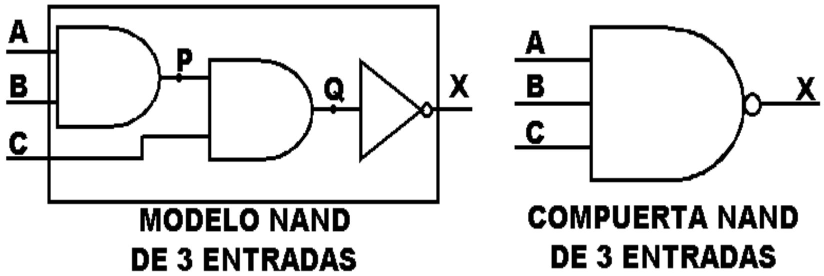


Figura 3.4: Composición AND-AND-INV = NAND3

En la siguiente sección se describe y analiza mas detalladamente la construcción de circuitos mas complejos, a partir de las compuertas básicas.

3.2 Circuitos Combinacionales y Secuenciales.

De acuerdo con Morris Mano [Man 87], los circuitos lógicos digitales pueden ser combinacionales o secuenciales. Él define a los circuitos combinacionales como circuitos que se componen de compuertas lógicas cuyas salidas estan determinadas por la combinación de las entradas sin tomar en cuenta el estado anterior de las mismas. A su vez un circuito secuencial es el que emplea elementos de memoria (celdas binarias), por esto sus salidas estan determinadas por las entradas actuales y por el estado de los elementos de memoria, que a su vez es una funcion determinada por las entradas anteriores.

Por lo anterior esta sección se divide a su vez en dos partes, la primera trata de la representación y verificación de circuitos combinacionales, y la segunda se refiere a los circuitos secuenciales.

3.2.1 Circuitos combinacionales

La principal carácteristica de los circuitos combinacionales es la realización de operaciones específicas de procesamiento de información, especificada por completo en forma lógica por un conjunto de funciones booleanas [Man 87]. Por esto modelar los circuitos combinacionales con procesos Circal mediante el metodo de modelación basada en el nivel de abstracción ([McC+ 95], [Mil 94] y [Mil 95]), para ejemplificar la validación del diseño de circuitos combinacionales se presentan dos ejemplos, el primero es un sumador de rizo con acarreo y el segundo es una memoria de sólo lectura (ROM).

Sumador de Rizo con Acarreo.

El ejemplo del sumador de rizo con acarreo se toma de [Mil 94], para seguir paso a paso el análisis del diseño de dicho sumador. El ejemplo de la tabla 3.3 muestra el código para la creación de un sumador completo.

Se definen dos procesos: `sumador_completo` y `sumador_completo2`; el primero se construye con dos medios sumadores (el medio sumador es definido previamente) y una compuerta OR de 2 entradas; mientras que el segundo

```
Accion a0, a1, b0, b1, cin0, cin1, cout0, cout1, sum0, sum1, s10, s11,
s20, s21, s30, s31.

 Proceso Suma.

 medio_sumador[X0, X1, Y0, Y1, SUM0, SUM1, COUT0, COUT1] ←
xor2[X0, X1, Y0, Y1, SUM0, SUM1] ← XOR,
and2[X0, X1, Y0, Y1, COUT0, COUT1] ← AND.

 sumador_completo[X0,X1, Y0, Y1, CIN0, CIN1, COUT0, COUT1,SUM0, SUM1] ←
MEDSUM1 * MEDSUM2 * OR - [s10,s11,s20,s21,s30,s31],
medio_sumador[X0, X1, Y0, Y1, s10,s11,s20,s21] ← MEDSUM1,
medio_sumador[CIN0, CIN1, s10, s11, SUM0, SUM1, s30, s31] ← MEDSUM2,
or2[s20, s21, s30, s31, COUT0, COUT1] ← OR.

 sumador_completo2[X0, X1, Y0, Y1, CIN0, CIN1, COUT0, COUT1,SUM0, SUM1] ←
[X0, Y0, CIN0, COUT0, SUM0]Suma + [X0, Y0, CIN1, COUT0, SUM1]Suma + [X0,
Y1, CIN0, COUT0, SUM1]Suma + [X0, Y1, CIN1, COUT1, SUM0]Suma + [X1, Y0,
CIN0, COUT0, SUM1]Suma + [X1, Y0, CIN1, COUT1, SUM0]Suma + [X1, Y1, CIN0,
COUT1, SUM0]Suma + [X1, Y1, CIN1, COUT1, SUM1]Suma.
```

Tabla 3.3: Sumador Completo.

se construye en base a la definición de su comportameniento, es decir de la misma manera que se definieron las compuertas básicas en la sección anterior en base a su tabla de verdad.

Con la siguiente consulta se compara si los árboles resultantes de los dos procesos son bisimilares, es decir su comportamiento es el mismo. Al hacer la consulta, simplemente se obtiene como resultado `Yes`, lo cual implica que los procesos son bisimilares.

```
   ?-sumador_completo(a0, a1, b0, b1, cin0, cin1, cout0, cout1,sum0,
sum1, Suma1),
n_arbol(Suma1, Suma1a),
sumador_completo2(a0, a1, b0, b1, cin0, cin1, cout0, cout1,sum0,
sum1, Suma2),
n_arbol(Suma2, Suma2a),
c_arbol(Suma1a, suma1).
```

A continuación se crea un sumador de rizo de 2 bits. Empleando dos sumadores completos se implementa un sumador de numeros binarios de dos dígitos con acarreo, este puede crearse utilizando `sumador_completo`, `sumador_completo2` o una combinación de ambos, como se aprecia en el ejemplo de la tabla 3.4.

```
sumador_rizo2[A0_0, A0_1, A1_0, A1_1, B0_0, B0_1, B1_0, B1_1,
CIN0, CIN1, COUT0, COUT1, S0_0, S0_1, S1_0, S1_1] ←
(SUMA1A * SUMA2A) - [co0_0, co0_1],
sumador_completo(A0_0, A0_1, B0_0, B0_1, CIN0, CIN1, co0_0, co0_1, S0_0,
S0_1, SUMA1),
sumador_completo(A1_0, A1_1, B1_0, B1_1, co0_0, co0_1, COUT0, COUT1, S1_0,
S1_1, SUMA2).
 sumador_rizo2a[A0_0, A0_1, A1_0, A1_1, B0_0, B0_1, B1_0, B1_1,
CIN0, CIN1, COUT0, COUT1, S0_0, S0_1, S1_0, S1_1] ←
(SUMA1 * SUMA2) - [co0_0, co0_1] ),
sumador_completo2[A0_0, A0_1, B0_0, B0_1, CIN0, CIN1, co0_0, co0_1, S0_0,
S0_1]← SUMA1),
sumador_completo2[A1_0, A1_1, B1_0, B1_1, co0_0, co0_1, COUT0, COUT1, S1_0,
S1_1]← SUMA2.
 sumador_rizo2b[ A0_0, A0_1, A1_0, A1_1, B0_0, B0_1, B1_0, B1_1,
CIN0, CIN1, COUT0, COUT1, S0_0, S0_1, S1_0, S1_1] ←
(SUMA1A * SUMA2) - [co0_0, co0_1],
sumador_completo[A0_0, A0_1, B0_0, B0_1, CIN0, CIN1, co0_0, co0_1, S0_0,
S0_1] ← SUMA1,
sumador_completo2[A1_0, A1_1, B1_0, B1_1, co0_0, co0_1, COUT0, COUT1, S1_0,
S1_1] ← SUMA2.
```

Tabla 3.4: Sumador de Rizo con Acarreo

Este ejemplo complementa al anterior (tabla 3.3). En este se crean tres sumadores de rizo, el primero `sumador_rizo2` utilizando `sumador_completo`, el segundo `sumador_rizo2a` utilizando `sumador_completo3` y por último `sumador_rizo2b`, para el que se utilizas una combinación de ambos sumadores. Con la siguiente consulta se comparan los tres sumadores de rizo entre sí, el resultado de ejecutar esta consulta es `Yes`, lo que demuestra que los procesos son bisimilares y a su vez los procesos creados a partir combinaciones de procesos bisimilares son bisimilares también.

```
    sumador_rizo2(a0_0, a0_1, a1_0, a1_1, b0_0, b0_1, b1_0, b1_1, cin0, cin1,
cout0, cout1, s0_0, s0_1, s1_0, s1_1, X),
   n_arbol(X,Y),
   sumador_rizo2a(a0_0, a0_1, a1_0, a1_1, b0_0, b0_1, b1_0, b1_1, cin0, cin1,
cout0, cout1, s0_0, s0_1, s1_0, s1_1, X1),
   n_arbol(X1,Y1),
```

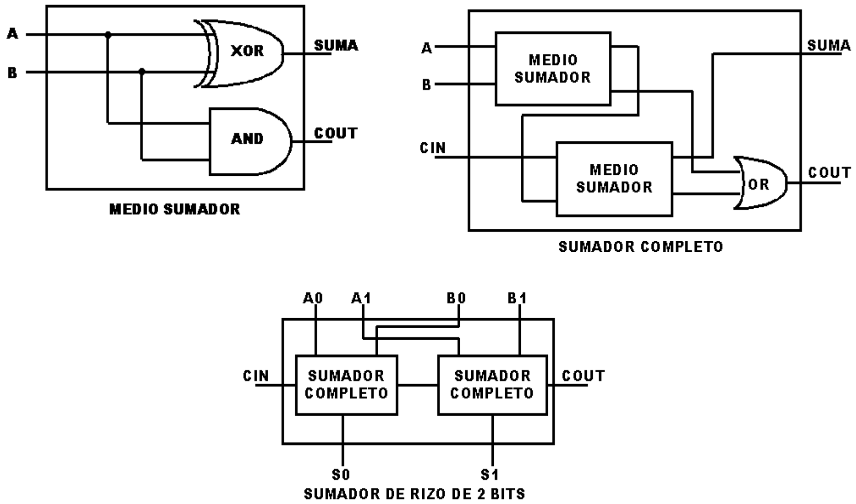


Figura 3.5: Sumador de Rizo de 2 Bits

```
   sumador_rizo2b(a0_0, a0_1, a1_0, a1_1, b0_0, b0_1, b1_0, b1_1, cin0, cin1,
cout0, cout1, s0_0, s0_1, s1_0, s1_1, X2),
   n_arbol(X2,Y2),
   c_arbol(Y,Y1),
   c_arbol(Y1,Y2),
   c_arbol(Y2,Y).
```

En este ejemplo se utilizó el metodo de diseño bottom-up, como muestra la figura 3.5, aquí se aprecia el desarrollo del diseño en 4 niveles de abstracción. Comenzando por el diseño con compuertas lógicas del medio sumador hasta la integración del sumador de rizo de 2 bits. En este método de diseño la complejidad del modelo aumenta a medida que se avanza para llegar al circuito deseado.

Memoria ROM

Este diseño se realizó utilizando el método de diseño top-down, la figura 3.6 ilustra este procediemiento.

En la figura 3.6, se ve como el circuito es dividido en dos partes, un decodificador de 2 x 4, el cual se define en el código, y un arreglo de compuertas OR; del mismo modo cada uno de estos elementos es a su vez descompuesto, el

Figura 3.6: Detalle Memoria ROM

decodificador es representado en este nivel por medio de compuertas lógicas; en tanto que el arreglo de compuertas OR debe ser todavía subdividido en dos partes mas, cada una de ellas es una compuerta OR de cuatro entradas que dan como salida f0 y f1, respectivamente; cada una de ellas es construida mediante compuertas OR de dos entradas.

La complejidad de este circuito es mayor que la del ejemplo anterior, tan solo el número de compuertas involucradas proporciona un claro ejemplo de idea de ello.

El listado de la tabla 3.5, muestra como se llevo a cabo la construcción de cada uno de los componentes del circuito, así como su integración final, misma que se compara posteriormente con su específicación, para comprobar la veracidad del modelo, con la consulta:.

```
 rom42x(a0, a1, b0, b1, f00, f01, f10, f11, X),
n_arbol(X,Y),
```

```
Accion a0, a1, b0, b1, f00, f01, f10, f11, ai0, ai1, bi0, bi1, p0, p1, q0,
q1.

 Proceso Rom.

 dec_2_x_4[A0, A1, B0, B1, D00, D01, D10, D11, D20, D21, D30, D31] ←
NOT0*NOT1*AND0*AND1*AND2*AND3-[ai0, ai1, bi0, bi1],
inv[A0,A1,ai0,ai1] ← NOT0,
inv[B0,B1,bi0,bi1] ← NOT1,
and2[ai0, ai1, bi0, bi1, D00, D01] ← AND0,
and2[ai0, ai1, B0, B1, D10, D11] ← AND1,
and2[A0, A1, bi0, bi1, D20, D21] ← AND2,
and2[A0, A1, B0, B1, D30, D31] ← AND3.

 or3[A0,A1,B0,B1,C0,C1,X0,X1] ← (OR1*OR2)-[p0,p1],
or2[A0,A1,B0,B1,p0,p1] ← OR1],
or2[C0,C1,p0,p1,X0,X1] ← OR2].

 or4[A0,A1,B0,B1,C0,C1,D0,D1,X0,X1] ← (OR1*OR2*OR3) - [p0,p1,q0,q1],
or2[A0, A1, B0,B1, p0, p1] ← OR1,
or2[C0, C1, p0,p1, q0, q1] ← OR2,
or2[D0, D1, q0,q1, X0, X1] ← OR3.

 rom42x[A00, A01, A10, A11, F00, F01, F10, F11] ←
DEC4X2 * OR41 * OR42 - [d00, d01, d10, d11, d20, d21, d30, d31],
dec_2_x_4(A00, A01, A10, A11, d00, d01, d10, d11, d20, d21, d30, d31] ←
DEC4X2),
or4[d00, d01, d10, d11, d20, d21, d30, d31, F00, F01] ← OR41,
or4[d00, d01, d10, d11, d20, d21, d30, d31, F10, F11] ← OR42.

 rom42a[A00, A01, A10, A11, _X, F01, _Y, F11] ← [A00,A10,F01,F11] ROM +
[A00,A11,F01,F11] ROM + [A01,A10,F01,F11] ROM + [A01,A11,F01,F11] ROM.
```

Tabla 3.5: Memoria ROM.

```
rom42a(a0, a1, b0, b1, f00, f01, f10, f11, X1),
n_arbol(X1,Y1),
c_arbol(Y,Y1).
```

La memoria sin programar es comparada con su tabla de verdad, la cual presenta ambas salidas en 1, ante cualquier entrada. Los modelos construidos con ésta implantación Circal no se adaptan a modificaciones en tiempo de ejecución, por lo que en el sentido estricto no es posible programar la memoria omitiendo sus conexiones internas. Lo que se puede hacer es sustituir las compuertas OR de 4 entradas, por compuertas de 2 o 3 entradas según sea necesario. Los ejemplos de las tablas 3.6 y 3.7 se construyerón de esta manera

```
rom42y[A00, A01, A10, A11, F00, F01, F10, F11] ←
DEC4X2 * OR41 *OR42 - [d00, d01, d10, d11, d20, d21, d30, d31],
dec_2_x_4[A00, A01, A10, A11, d00, d01, d10, d11, d20, d21, d30, d31] ←
DEC4X2,
or3[d10, d11, d20, d21, d30, d31, F00, F01] ← OR41,
or2[d00, d01, d20, d21, F10, F11] ← OR42.
rom42b[A00, A01, A10, A11, F00, F01, F10, F11] ←
[A00,A10,F00,F11] ROM +
[A00,A11,F01,F10] ROM +
[A01,A10,F01,F11] ROM +
[A01,A11,F01,F10] ROM).
```

Tabla 3.6: Memoria ROM, programada como OR e INV

El primer ejemplo (tabla 3.6) se crea de acuerdo con el que aparece en el texto de Morris Mano, en él la memoria ROM es programada para simular el comportamiento de dos compuertas, una OR cuyos valores se verifican en f0 y una NOT cuyos valores se otienen en f1, y reflejan sólo el comportamiento respecto a la entrada b. Como ya se mencionó las conexiones no pueden ser omitidas y alambradas en tiempo de ejecución, por lo que se sustituyen las dos compuertas OR de cuatro entradas por una compuerta OR de 3 y otra de 2 entradas.

La consulta que sigue compara los dos modelos de memoria ROM creados, al ejecutarla se obtiene un resultado favorable.

```
 ?- rom42y(a0, a1, b0, b1, f00, f01, f10, f11, X),
n_arbol(X,Y),
rom42b(a0, a1, b0, b1, f00, f01, f10, f11, X1),
n_arbol(X1,Y1),
c_arbol(Y,Y1).
```

Del mismo modo que el anterior, el ejemplo en la tabla 3.7 muestra como se implementa la programación de la memoria ROM para que ante las distintas entradas posibles de a y b se comporte como una compuerta AND cuya salida es f1 y una compueta NAND representada por f2. Para la compuerta AND se requiere sólo una salida en alto, por lo que la OR de 4 entradas es simplemente reemplazada por un Buffer; mientras que para la compuerta NAND la programación se simula sustituyendo con una compuerta OR de 3 entradas.

```
rom42z[A00, A01, A10, A11, F00, F01, F10, F11] ←
DEC4X2 * BUF * OR42 - [d00, d01, d10, d11, d20, d21, d30, d31],
dec_2_x_4[A00, A01, A10, A11, d00, d01, d10, d11, d20, d21, d30, d31] ←
DEC4X2,
buf[d30, d31, F00, F01] ← BUF,
or3[d00, d01, d10, d11, d20, d21, F10, F11] ← OR42.
 rom42c[A00, A01, A10, A11, F00, F01, F10, F11] ←
[A00,A10,F00,F11]ROM +
[A00,A11,F00,F11]ROM +
[A01,A10,F00,F11]ROM +
[A01,A11,F01,F10]ROM.
```

Tabla 3.7: Memoria ROM programada como AND y NAND

Nuevamente al verificará la validez del modelo construido (`rom42z`), comparándolo con el modelo del comportamiento (`rom42c`), mediante la consulta el resultado es positivo.

```
 rom42z(a0, a1, b0, b1, f00, f01, f10, f11, X),
n_arbol(X,Y),
rom42c(a0, a1, b0, b1, f00, f01, f10, f11, X1),
n_arbol(X1,Y1),
c_arbol(Y,Y1).
```

Hasta aquí se han mostrado dos ejemplos, los cuales explican la manera en que se puede llevar a cabo la representación y verificación de circuitos combinacionales en Circal.

3.2.2 Circuitos Secuenciales

La diferencia básica entre los ciruitos combinacionales y los secuenciales radica en que los primeros responden y se comportan de modo que sus salidas son una función directa de las entradas, en cambio las salidas de los segundos son una función de sus entradas y del estado anterior de salida, es decir tienen memoria.

Un circuito secuencial de modo simplificado es la union de un circuito combinacional con una unidad de memoria. Estas unidades de memoria estan por lo general construidas a partir de flip-flops. Por esto la primera parte

Figura 3.7: Flip-Folp's Básicos

de esta sección se dedica al diseño y representación de flip-flops y la segunda muestra un ejemplo de un ciruito secuencial muy sencillo. Los modelos de estos ejemplo se tomaron en su totalidad del libro de Morris Mano.

Diseño con Flip-Flops.

Los flip-flops básicos se muestran en la figura 3.7, cada uno de ellos puede construirse a partir del flip-flop RS o a partir de las compuertas básicas como se muestra a continuación.

El flip-flop básico RS, se puede construir a partir de compuertas NAND o NOR, la diferencia es mínima, y rádica en el hecho de que para el que se construye con compuertas NAND el poner sus dos entradas en 0 genera una salida ambigua, mientras que para el construido con compuertas NOR ésta se genera al tener ambas entradas en 1.

```
Accion s0,s1,r0,r1,qi0,qi1,qo0,qo1,clk0,clk1, ri0,ri1,si0,si1.

 ffrs_nor[S0, S1, R0, R1, Q0, Q1, QI0, QI1] ← NOR1*NOR2,
nor2[S0,S1,QI0,QI1,Q0,Q1] ← NOR1,
nor2[R0,R1,Q0,Q1,QI0,QI1] ← NOR2.

 ffrs_nand[S0, S1, R0, R1, Q0, Q1, QI0, QI1] ← NAND1*NAND2,
nand2[S0,S1,QI0,QI1,Q0,Q1] ← NAND1,
nand2[R0,R1,Q0,Q1,QI0,QI1] ← NAND2.

 ffrs_clk[S0, S1, R0, R1, Q0, Q1, QI0, QI1, CLK0, CLK1] ←
(AND1 * AND2 * FFRS) - [ri0,ri1,si0,si1]
and2[R0,R1,CLK0,CLK1,ri0,ri1] ← AND1,
and2[S0,S1,CLK0,CLK1,si0,si1] ← AND2,
ffrs_nor[si0,si1,ri0,ri1,Q0,Q1,QI0,QI1] ← FFRS.

 ffrs_clka[S0, S1, R0, R1, Q0, Q1, QI0, QI1, CLK0, CLK1] ←
(AND1 * AND2 * NOR1* NOR2) - [ri0,ri1,si0,si1]
and2[R0,R1,CLK0,CLK1,ri0,ri1] ← AND1,
and2[S0,S1,CLK0,CLK1,si0,si1] ← AND2,
nor2[si0,si1,QI0,QI1,Q0,Q1] ← NOR1,
nor2[ri0,ri1,Q0,Q1,QI0,QI1] ← NOR2.
```

Tabla 3.8: Flip - Flop RS.

El flip-flop RS es un circuito secuencial asíncrono, por lo que es necesario agregar un pulso de reloj a la entrada, a fin incrementar el control sobre la entrada y salida (como el `ffrs_clk`), que por otra parte se puede construir también utilizando sólo las compuertas básicas como en `ffrs_clka`.

Por último como en los ejemplos anteriores se procede a comparar ambas implementaciones a fin de determinar si son bisimilares, lo cual es cierto.

```
 ffrs_clk(s0,s1,r0,r1,qi0,qi1,qo0,qo1,clk0,clk1,FF),
n_arbol(FF,FF_A),
ffrs_clka(s0,s1,r0,r1,qi0,qi1,qo0,qo1,clk0,clk1,FF1),
n_arbol(FF1,FF1_A),
c_arbol(FF_A,FF1_A).
```

El mismo proceso de diseño se repite con los flip-flops D que se presentan a continuación, en el se puede ver como se deriva el Flip-Flop D a partir del Flip-Flop RS. La figura 3.8 muestra el detalle de este diseño.

Figura 3.8: Modelo de Flip - Flop D

```
ffd_clk[D0, D1, Q0, Q1, QI0, QI1, CLK0, CLK1, ← (INV * FFRSCLK) - [r0,r1]
inv[D0,D1,r0,r1] ← INV, ffrs_clk[D0, D1, r0, r1, Q0, Q1, QI0, QI1, CLK0,
CLK1] ← FFRSCLK.
```

```
ffd_clka[D0, D1, Q0, Q1, QI0, QI1, CLK0, CLK1] ← (INV * AND1 * AND2
* NOR1* NOR2) - [ri0,ri1,si0,si1,r0,r1] inv[D0,D1,r0,r1] ← INV,
and2[D0,D1,CLK0,CLK1,ri0,ri1] ← AND1, and2[r0,r1,CLK0,CLK1,si0,si1] ←
AND2, nor2[ri0,ri1,QI0,QI1,Q0,Q1] ← NOR1, nor2[si0,si1,Q0,Q1,QI0,QI1] ←
NOR2.
```

Tabla 3.9: Flip - Flop D.

Reutilizar los circuitos creados con anterioridad permite apreciar la característica de bisimilaridad de los procesos Circal, ya que en el presente ejemplo utilizar el ffrs_clk, implica compactar la definición a del flip flop, pues esta sola línea sustituye a las cuatro correspondientes en ffd_clka . El ejemplo de la tabla 3.9, en el que se construyen Flip -Flops Maestro Esclavo, permite apreciar esta misma característica.

Al igual que el ejemplo anterior aquí el Flip-Flop T es creado a partir del Flip-Flop JK , alambrado como T como lo muestra la figura 3.9, y también como el Flip-Flop JK, detallando todos sus componentes.

Hasta aqui se han presentado sólo ejemplos del uso de Circal como auxiliar en el diseño y verificación de sistemas digitales de Hardware. En la última sección de este cápitulo se muestra un ejemplo de la aplicación de Circal en el análisis de otro tipo de sistemas concurrentes de tiempo real.

```
ffjkms_clk[J0, J1, K0, K1, Q0, Q1, QI0, QI1, CLK0, CLK1] ←
(NAND31 * NAND32 * RSNAND1 * INV * NAND1 * NAND2 * RSNAND2)
- [ri0, ri1, si0, si1, rj0, rj1, sj0, sj1, clki0, clki1,
q_0, q_1, q_c0, q_c1] nand3[K0,K1,CLK0,CLK1,QI0,QI1,ri0,ri1]
← NAND31, nand3[J0,J1,CLK0,CLK1,Q0,Q1,si0,si1] ← NAND32,
ffrs_nand[ri0,ri1,si0,si1,q_0,q_1,q_c0,q_c1] ← RSNAND1,
inv[CLK0,CLK1,clki0,clki1] ← INV, nand2[q_0,q_1,clki0,clki1,rj0,rj1]
← NAND1, nand2[q_c0,q_c1,clki0,clki1,sj0,sj1] ← NAND2,
ffrs_nand[rj0,rj1,sj0,sj1,Q0,Q1,QI0,QI1] ← RSNAND2.
```

```
 fftms_clk[T0, T1, Q0, Q1, QI0, QI1, CLK0, CLK1] ← FFJKCLK,
ffjkms_clk[T0, T1, T0, T1, Q0, Q1, QI0, QI1, CLK0, CLK1] ← FFJKCLK.
```

```
fftms_clka[T0, T1, Q0, Q1, QI0, QI1, CLK0, CLK1] ← (NAND31 * NAND32 *
RSNAND1 * INV * NAND1 * NAND2 * RSNAND2) - [ri0, ri1, si0, si1, rj0,
rj1, sj0, sj1, clki0, clki1, q_0, q_1, q_c0, q_c1] nand3[T0, T1, CLK0,
CLK1, QI0, QI1, ri0, ri1] ← NAND31, nand3[T0, T1, CLK0, CLK1, Q0, Q1,
si0, si1] ← NAND32, ffrs_nand[ri0, ri1, si0, si1, q_0, q_1, q_c0, q_c1]
← RSNAND1, inv[CLK0, CLK1, clki0, clki1] ← INV, nand2[q_0, q_1, clki0,
clki1, rj0, rj1] ← NAND1, nand2[q_c0, q_c1, clki0, clki1, sj0, sj1] ←
NAND2, ffrs_nand[rj0, rj1, sj0, sj1, Q0, Q1, QI0, QI1] ← RSNAND2.
```

Tabla 3.10: Flip - Flop Maestro Esclavo JK y T.

Figura 3.9: Modelo del Flip - Flop T, a partir de un JK

3.3 Simulador de Tráfico Vehicular.

Este ejercicio se construyó tomando como base el texto "Simulating Vehicular Traffic Flows Using the Circal System" [Rus 94]. Con el fin de ilustrar la capacidad de Circal para verificar otros sistemas concurrentes de tiempo real, como lo es en este caso el flujo del tráfico vehicular.

De acuerdo con el ejemplo mencionado es necesario crear cuatro elementos básicos, los cuales al ser combinados nos permiten la creación de modelos de porciones de camino en distintas formas. Estos elementos son: una Célula de Camino (Road Cell), una señal de tráfico simple (Traffic Light), un distribuidor de 2 a 1 (Splitter) y un componete de arbitraje (Arbitration Component). Con estos elementos se contruyen de manera jerarquica componetes cada vez mas complejos para conseguir al final el modelo de un cruce de dos calles de doble sentido.

Figura 3.10: Elementos básicos de representación de un camino

Se requiere de un modelo de sistema cerrado, para que se generen eventos determinísticos en cada ciclo de reloj. Para ello es necesario agregar dos nuevos componentes a los elemetos básicos: Un Generador de Autos (Source) y un Destructor de Autos (Sink).

Primero ser define, construye y prueba cada uno de los cuatro elementos básicos (La Célula de Camino, el Generador, el Destructor).

La célula de camino es el único elemento que tiene mas de un estado, la célula de camino es un componente que puede contener un auto o estar vacio, suponiendo que no puede haber menos de una célula de distancia entre auto y auto. El modelo completo de esta y de los otros componentes se presenta en la figura 3.10.

La Célula de Camino tiene como proposito modelar un tramo de camino de una determinada longitud (suficiente para contener un vehiculo a la vez, por ejemplo 5 metros), el cual puede estar lleno o vacio. Se compone con cinco puertos, un pulso de reloj (tick) que sincroniza el modelo, dos puertos para comunicar el ingreso de un auto en la celda (req_in, ack_in) y los dos restantes para cooordinar la salida de un vehiculo de la Célula (req_out, ack_out). Asi como de 2 estados lleno y vacío (representados por los procesos del mismo nombre). El Generador de Autos (origen) consta de 3 puertos, un pulso de

Figura 3.11: Representación de tramo de camino.

reloj para sinconizarse con los otros elementos del modelo, y dos puertos que determinan el ingreso de un auto a la vez en el sistema (req_out, ack_out). Al igual que el elemento anterior el Destructor de autos consta de tres puertos, uno de sincronización (tick), así como dos para controlar la salida de autos desde el sistema (req_in, ack_in). El trabajo que realiza la señal de tráfico (semáforo), se reduce a la de detener el cambio del sistema en ausencia de la señal de siga (go), o ser transparente al elemento previo si ésta aparece. Para demostrar el funcionamiento de las células de camino se desarrolla el ejemplo de la figura 3.11, en el que intervienen 3 de los elemtos descritos: la Célula de Camino, el Elemento que Genera y el que Destruye autos.

Como se observa el tramo de un camino es construido sencillamente con la composición de los elementos necesarios, dos células de camino, un generador y un destructor de autos. El diseño completo del sistema está dado en el listado de la tabla 3.11.

El manejo de un proceso con estados se realiza repartiendo el comportamiento a su vez en dos estados distintos (lleno y vacio), representados por las transiciones entre los procesos vacía y Llena. Así el comportamiento de la celda vacía indica que el estado vacío puede cambiar a lleno ante la ocurrencia de los eventos [t, REQ_IN] o [t, REQ_IN, ACK_OUT] o permanecer en vacío si ocuren [t] o [t, ACK_OUT]. El comportamiento de la celda cuando está llena se representa mediante el predicado lleno, el cual refleja un comportamiento en el cual puede cambiar a vacío si ocurren los eventos [t, ACK_IN,

```
Accion t,req,ack,req1,ack1,req2,ack2.
Proceso Vacia, Llena, Crea, Destruye.
```

```
vacio[t, REQ_IN, ACK_OUT] ← [t, REQ_IN]Llena + [t, REQ_IN, ACK_OUT]Llena
+ [t, ACK_OUT]Vacia+ [t] Vacia.
```

```
lleno[t, REQ_IN, ACK_IN, REQ_OUT, ACK_OUT] ← [t, ACK_IN, REQ_OUT]Vacia +
[t, REQ_IN, ACK_IN, REQ_OUT]Vacia + [t, ACK_IN, REQ_OUT, ACK_OUT]Llena +
[t, REQ_IN, ACK_IN, REQ_OUT, ACK_OUT]Llena).
```

```
origen[t, REQ_OUT, ACK_OUT] ← [t, REQ_OUT, ACK_OUT]Crea+ [t,
REQ_OUT]Crea+ [t, ACK_OUT]Crea).
```

```
final[t, REQ_IN, ACK_IN] ← [t, REQ_IN, ACK_IN]Destruye+ [t,
REQ_IN]Destruye+ [t, ACK_IN]Destruye+ [t]Destruye).
```

Consulta que crea y normaliza los tramos de camino

```
tramo:- origen(t, req, ack, INICIA),
vacio(t, req, ack1, CELDA_1),
vacio(t, req1, ack2, CELDA_2),
lleno(t, req, ack, req1, ack1, CELDA_1A),
lleno(t, req1, ack1, req2, ack2, CELDA_2A),
final(t, req2, ack2, TERMINA),
normal(INICIA*CELDA_1*CELDA_2*TERMINA,UNO),
normal(INICIA*CELDA_1A*CELDA_2*TERMINA,DOS),
normal(INICIA*CELDA_1*CELDA_2A*TERMINA,TRES),
normal(INICIA*CELDA_1A*CELDA_2A*TERMINA,CUATRO).
```

Tabla 3.11: Tramo de Camino formado por dos Células

REQ_OUT] o [t, REQ_IN, ACK_IN, REQ_OUT], y permanece sin cambio ante los eventos [t, ACK_IN, REQ_OUT, ACK_OUT] o [t, REQ_IN, ACK_IN, REQ_OUT, ACK_OUT].

Esta forma de representar la transición entre estados de un mismo elemento origina que se tengan que construir cuatro procesos, derivados de la composición de los elementos básicos. Son sólo cuatro debido a que los únicos elementos con estado son las células de camino, se combinan sus posibles estados a fin de obtener los cuatro posibles estados del sistema completo (vacío-vacío, lleno-vacío, vacío-lleno y lleno - lleno), éstos se representan como la claúsula tramo.

Los resultados obtenidos de este ejemplo, es decir, las formas normales contenidas por las variables UNO, DOS, TRES y CUATRO, proporcionan el comportamiento del sistema. Al ejecutar la consulta de tramo, resultan cuatro valores, que son los procesos P0007, P0008, P0009 y P0010, cuyos valores se presentan al hacer la consulta `listing(definido)`, que proporciona los siguientes valores.

```
definido(P0001 leftarrow  (Llena*Destruye)).
definido(P0002 leftarrow  (Vacia*Destruye)).
definido(P0003 leftarrow  (Llena*P0001)).
definido(P0004 leftarrow  (Llena*P0002)).
definido(P0005 leftarrow  (Vacia*P0001)).
definido(P0006 leftarrow  (Vacia*P0002)).
definido(P0007 leftarrow  (Crea*P0003)).
definido(P0008 leftarrow  (Crea*P0004)).
definido(P0009 leftarrow  (Crea*P0005)).
definido(P0010 leftarrow  (Crea*P0006)).
```

Los últimos cuatro valores al ser expandidos resultan ser los tres estados descritos con anteriridad, y son los mismos que aparecen en las formas normales obtenidas de la consulta tramo. Por ellas podemos decir hacia que estado evolucionará el sistema, dependiendo de las entradas y el estado presente. En las siguientes tablas se muestran estas evoluciones.

Actual	Acciones	Procesos		Evoluciona a
1	[ack, ack1, ack2, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack, ack1, ack2, req, req1, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack, ack1, ack2, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack, ack1, ack2, req, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack, ack1, ack2, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
1	[ack, ack1, ack2, req1, t]	P0009	3	Crea*Vacia*Llena*Destruye
1	[ack, ack1, ack2, req2, t]	P0010	1	Crea*Vacia*Vacia*Destruye
1	[ack, ack1, ack2, t]	P0010	1	Crea*Vacia*Vacia*Destruye
1	[ack, ack1, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack, ack1, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack, ack1, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
1	[ack, ack1, req2, t]	P0010	1	Crea*Vacia*Vacia*Destruye
1	[ack, ack2, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack, ack2, req, req1, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack, ack2, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack, ack2, req, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack, ack2, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
1	[ack, ack2, req1, t]	P0009	3	Crea*Vacia*Llena*Destruye
1	[ack, ack2, req2, t]	P0010	1	Crea*Vacia*Vacia*Destruye
1	[ack, ack2, t]	P0010	1	Crea*Vacia*Vacia*Destruye
1	[ack, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
1	[ack, req2, t]	P0010	1	Crea*Vacia*Vacia*Destruye
1	[ack1, ack2, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack1, ack2, req, req1, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack1, ack2, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack1, ack2, req, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack1, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack1, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack2, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack2, req, req1, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[ack2, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[ack2, req, t]	P0008	2	Crea*Llena*Vacia*Destruye
1	[req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
1	[req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye

Esta primera tabla muestra como del estado 1 (vacío*vacío) se puede pasar a cualqiera de los estados del sistema, incluso a si mismo. De igual forma se puede observar cuales son los eventos a los que responde este estado.

Actual	Acciones	Procesos		Evoluciona a
2	[ack, ack1, ack2, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
2	[ack, ack1, ack2, req, req1, t]	P0007	4	Crea*Llena*Llena*Destruye
2	[ack, ack1, ack2, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
2	[ack, ack1, ack2, req1, t]	P0007	4	Crea*Llena*Llena*Destruye
2	[ack, ack1, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
2	[ack, ack1, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
2	[ack, ack2, req, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
2	[ack, ack2, req, req1, t]	P0009	3	Crea*Vacia*Llena*Destruye
2	[ack, ack2, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
2	[ack, ack2, req1, t]	P0009	3	Crea*Vacia*Llena*Destruye
2	[ack, req, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
2	[ack, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye

Del estado 2 (Lleno - vacío), se puede pasar al estado 3 (vacío-lleno) o permanecer en el mismo estado.

Actual	Acciones	Procesos		Evoluciona a
3	[ack, ack1, ack2, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
3	[ack, ack1, ack2, req, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
3	[ack, ack1, ack2, req1, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
3	[ack, ack1, ack2, req2, t]	P0009	3	Crea*Vacia*Llena*Destruye
3	[ack, ack1, req, req1, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
3	[ack, ack1, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
3	[ack, ack1, req1, req2, t]	P0010	1	Crea*Vacia*Vacia*Destruye
3	[ack, ack1, req2, t]	P0010	1	Crea*Vacia*Vacia*Destruye
3	[ack1, ack2, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
3	[ack1, ack2, req, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
3	[ack1, req, req1, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye

Del estado 3 (vacío - Lleno) se puede evolucionar a cualquier estado del sistema, incluso a si mismo.

Actual	Acciones	Procesos		Evoluciona a
4	[ack1, req, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
4	[ack, ack1, ack2, req, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
4	[ack, ack1, ack2, req1, req2, t]	P0007	4	Crea*Llena*Llena*Destruye
4	[ack, ack1, req, req1, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye
4	[ack, ack1, req1, req2, t]	P0008	2	Crea*Llena*Vacia*Destruye

Por último del estado 4 (lleno - lleno) sólo se puede evolucionar al estado 2, o permanecer en el mismo estado 4.

Este ejercicio permite reconocer la capacidad descriptiva de Circal como un álgebra de procesos. Aqui no se trato de determinar si el sistema diseñado es correcto. El esfuerzo se orientó a conseguir una especificación que permitiera estudiar un sistema concurrente real, modelarlo con circal y estudiar los resultados obtenidos de la normalización del problema.

Con este ejemplo se concluye el último cápitulo del presente trabajo de tesis. Las experiencias obtenidas en la práctica durante la elaboración del mismo aportan nuevas ideas y refuerzan los conceptos descritos en los 2 primeros capítulos.

4

Conclusiones

El objetivo general de este trabajo de tesis fue diseñar, implantar y probar un prototipo para auxiliar en el diseño de circuitos digitales mediante la aplicación de un álgebra de procesos. El álgebra de procesos elegida para realizar esta tarea fue Circal, debido a su orientación original hacia el diseño de circuitos digitales.

El desarrollo paso a paso del trabajo permitio cumplir cada uno de los objetivos planteados, como se detalla a continuación.

Las definiciones y conceptos planteados en los capítulos 1 y 2, fueron la base fundamental para el diseño del prototipo, cuya construcción se desarrolla detalladamente en el capítulo 3. Debido al uso de prolog para la implantación del prototipo, los procesos de diseño e implantación se redujeron a un simple paso, que es reescribir como claúsulas de prolog las leyes de circal y los teoremas y axiomas para la reescritura de procesos en la forma normal circal, así como para utilizar el árbol de mapeo de la forma normal propuesto por Mao.

Aplicar el diseño a ejemplos de la bibliografia de circal sirvio para constatar que el prototipo es útil en la verificación de sistemas digitales de hardware, así como para la descripción y análisis de otros sistemas concurrentes de tiempo real, como es el caso del ejemplo de simulación de tráfico vehicular.

La aplicación del prototipo en el análisis permite determinar los estados a los que evolucionará un proceso, las acciones ante las cuales responde y sus relaciones con otros procesos.

Una de las principales contribuciones del presente trabajo de tesis es el aplicar la semántica de Bisimilaridad definida por Mao, quien a su vez la construyó basandose en el trabajo de Robin Milner. Dado que esta no había sido probada antes en la construcción de un prototipo similar al presente.

Como otra aportación es el uso de los árboles de mapeo de la forma normal para determinar la bisimilaridad de dos procesos. Este concepto no es nuevo, sin embargo su aplicación en una herramienta de verificación de sistemas concurrentes si lo es.

Este trabajo puede ser la base para la creación de una herramienta práctica de diseño y modelación de sistemas concurrentes de tiempo real. Para ello se requiere construir las herramientas adecuadas para permitir la modelación mas sencilla de, por ejemplo, procesos con más de un estado (como la célula de camino del ejercicio sobre el tráfico vehicular), o procesos asincronos, los cuales requieren de un poco mas de trabajo para lograr una representación satisfactoria.

Asimismo se puede complementar con la creación de una interfaz más expresiva, que reporte la información con más detalle, como en el caso de la evolución de los procesos y sus composiciones internas.

El proceso de verificación entre las transiciones es sólo una ayuda para el diseñador, ya que en un sistema complejo no es suficiente el determinar si existe un error o no durante la transición, de hecho es deseable que al mismo tiempo indique la causa del mismo. Aunque este proceso es ajeno a Circal, el prototipo puede darnos algunas pistas respecto a las causas de las diferencias entre el comportamiento de una especificación y su predecesora.

Apéndice A

Listado

En la presente sección se incluye una copia del listado final del prototipo de circal creado para ejecutarse con el interprete SWI-Prolog, La finalidad del mismo es detallar la manera en que se construyó.

Para Facilitar su comprensión el listado incluye comentarios, los cuales en SWI se hacen igual que en C.

```
/* Prototipo de Circal */

/* Definicin de Operadores */

          :-op(400, xfx, <-).
          :-op(450, xfy, ~).
          :-op(470, xfy, ^).
          :-op(500, xfy, &).
          :-op(550, xfy, *).
          :-op(600, xfy, +).
          :-op(650, xfy, -).

/******************************/
/* Reetiquetacion de acciones */
/******************************/

          reet(P + Q, PR + QR):-
                reet(P, PR),
                reet(Q, QR).

          reet(P & Q, PR & QR):-
                reet(P, PR),
                reet(Q, QR).

          reet(P * Q, PR * QR):-
                reet(P, PR),
                reet(Q, QR).

          reet(P - Q, PR - Q):-
                reet(P, PR).

          reet(Acc ~ Proc_1^Vars, AccR~Et_Proc) :-
                reet(Proc_1^Vars, Proc_1R),
                evento(Acc, AccE),
                reemp(Vars,AccE,AccR),
                renombra(Proc_1R^Vars,Et_Proc).

          reet((Proc_1 + Proc_2)^Vars,Proc_1R + Proc_2R):-
                reet(Proc_1^Vars, Proc_1R),
                reet(Proc_2^Vars, Proc_2R).
```

```

        reet((Proc_1 * Proc_2)^Vars,Proc_1R * Proc_2R):-
                reet(Proc_1^Vars, Proc_1R),
                reet(Proc_2^Vars, Proc_2R).

        reet((Proc_1 & Proc_2)^Vars,Proc_1R & Proc_2R):-
                reet(Proc_1^Vars, Proc_1R),
                reet(Proc_2^Vars, Proc_2R).

        reet((Proc_1 - Acc)^Vars,Proc_1R -Acc):-
                reet(Proc_1^Vars, Proc_1R).

        reet(Proc_1 ^ _ , Proc_1):-
proceso(Proc_1).

        reet(Proceso,Proceso).

        reemp(Lista, [A|Resto],[AR|RestoR]):-
                member(A/AR, Lista),
                reemp(Lista, Resto, RestoR).
        reemp(Lista, [A|Resto],[A|RestoR]):-
                reemp(Lista, Resto, RestoR).
        reemp(_, [],[]).

/*******************************************/
/* aplicacion de la propiedad distributiva */
/* de & respecto a otros operadores        */
/*******************************************/

        dist(PRO_A & PRO_B, PRO_AR & PRO_BR):-
                dist(PRO_A,PRO_AR),
                dist(PRO_B,PRO_BR).

        dist(PRO_A + PRO_B, PRO_R):-
                dist(PRO_A,PRO_AR),
                dist(PRO_B,PRO_BR),
                mixup(PRO_AR, PRO_BR,PRO_R).

        dist(EV ~ PRO, EV_R ~ PRO_R):-
                dist(PRO, PRO_R),
                evento(EV, EV_R).

        dist(PRO_A * PRO_B, PRO_R):-
                dist(PRO_A,PRO_AR),
                dist(PRO_B,PRO_BR),
                mixup2(PRO_AR, PRO_BR,PRO_R).

        dist(Pro_A-Ev,Pro_R-Ev_R):- dist(Pro_A,Pro_R),evento(Ev,Ev_R).

        dist(PRO, PRO):-
proceso(PRO).

        mixup(X&Y,Z,XZ&YZ):-mixup(X,Z,XZ),mixup(Y,Z,YZ).
        mixup(X,Y&Z,XY&XZ):-X\=(_&_),mixup(X,Y,XY),mixup(X,Z,XZ).
        mixup(X,Y,X + Y):-X\=(_&_),Y\=(_&_).

        mixup2(X&Y,Z,XZ&YZ):-mixup2(X,Z,XZ),mixup2(Y,Z,YZ).
        mixup2(X,Y&Z,XY&XZ):-X\=(_&_),mixup2(X,Y,XY),mixup2(X,Z,XZ).
        mixup2(X,Y,X * Y):-X\=(_&_),Y\=(_&_).

/***************************/
/* Composicion de porcesos */
/***************************/


        /* compone dos procesos, manda primero a componer cada elemeto */
/* posteriormente mezcla el resultado de estps dos */
        comp(P_1*P_2,Compuesto):-
                comp(P_1,P_1c),
                comp(P_2,P_2c),
                compon(P_1c,P_2c,Compuesto).

/* La composicin respecto a los otros operadores de circal */
        comp(P_1&P_2, P_1c&P_2c):- comp(P_1,P_1c), comp(P_2,P_2c).

        comp(P_1+P_2, P_1c+P_2c):- comp(P_1,P_1c), comp(P_2,P_2c).

```

```
          comp(P_1-Ev, P_1c-Ev):- comp(P_1,P_1c).

          comp(Ev^P_1, Ev2^P_1c):- evento(Ev,Ev2), comp(P_1,P_1c).

          comp(P_1,P_1):- proceso(P_1).

/* realiza la composicin de dos procesos dados */
          compon(P_1,P_2,Compuesto):-
                 orden_p(P_1,P_1o),
                 orden_p(P_2,P_2o),
                 compone(P_1,P_1o,P_2,P_2o,Compuesto).

/* Se adelgazan las sumatorias y compone proceso a proceso */

          compone(P_1+Res,P_1o,P_2,P_2o, Compuesto):-
                 compone(P_1,P_1o,P_2,P_2o,Comp_1),
                 compone(Res,P_1o,P_2,P_2o,Comp_2),
                 la_quita(Comp_1,Comp_2,Compuesto).

          compone(P_1,P_1o,P_2 + Res,P_2o, Compuesto):-
                 compone(P_1,P_1o,P_2,P_2o,Comp_1),
                 compone(P_1,P_1o,Res,P_2o,Comp_2),
                 la_quita(Comp_1,Comp_2,Compuesto).


/* Cuando nos encontramos ante procesos con guardia y la interseccin */
/* de sus guardias con sus ordenes es igual */
          compone(A_1^P_1,P_1o,A_2^P_2,P_2o,A_c^Pro_Ren):-
                 intersection(A_1,P_2o,I_1),
                 intersection(A_2,P_1o,I_1),
                 union(A_1,A_2,A_x),
                 sort(A_x,A_c),
                 renombra(P_1*P_2,Pro_Ren).

/* Cuando la Interseccin de la guardia del primero y el orden del segundo */
/* es nula */
          compone(A_1^P_1,_,A_2^P_2,P_2o,A_1^Pro_Ren):-
                 intersection(A_1,P_2o,[]),
                 renombra(P_1*A_2^P_2,Pro_Ren).

/* Cuando la Interseccin de la guardia del Segundo y el orden del primero */
/* es nula */

          compone(A_1^P_1,P_1o,A_2^P_2,_,A_2^Pro_Ren):-
                 intersection(A_2,P_1o,[]),
                 renombra(A_1^P_1*P_2,Pro_Ren).

          compone(_,_,_,_,delta).

/**************************/
/* Abstraccion de procesos */
/**************************/

          abst(Proc_1-Vars,Proc_1R):-
                 act_p(Proc_1, Act),
                 member(Vars,Act),
                 abst2((Proc_1)-Vars,Proc_1R).

          abst(Proc_1 + Proc_2-Vars,Proc_1R + Proc_2R):-
                 abst(Proc_1-Vars, Proc_1R),
                 abst(Proc_2-Vars, Proc_2R).

          abst(Proc_1 & Proc_2-Vars,Proc_1R & Proc_2R):-
                 abst(Proc_1-Vars, Proc_1R),
                 abst(Proc_2-Vars, Proc_2R).

          abst(Proc_1 * Resto-Vars,((Proc_1 * Resto)-Vars)).

          abst(Acc ^ Proc_1-Vars, AccR^Proc_1R) :-
                 abst(Proc_1-Vars, Proc_1R),
                 evento(Acc, AccE),
                 evento(Vars,VarsE),
                 subtract(AccE,VarsE,AccR).

          abst(delta-_,delta).

          abst(Proc_1-Vars, Proc_Ren):-
```

```
                 proceso(Proc_1),
                 renombra(Proc_1-Vars,Proc_Ren).

         abst(Proc,Proc).

         abst2((Proc_1)-Vars,(D1 + N1) & N1):-
                 abst2e((Proc_1)-Vars,D1,N1).

         abst2e((Proc_1 + Proc_2)-Vars, D1+D2, N1&N2):-
                 abst2e(Proc_1-Vars, D1, N1),
                 abst2e(Proc_2-Vars, D2, N2).

         abst2e((Proc_1 & Proc_2)-Vars,D1+D2, N1+N2):-
                 abst2e(Proc_1-Vars, D1, N1),
                 abst2e(Proc_2-Vars, D2, N2).

         abst2e(Proc_1 * Resto-Vars,((Proc_1 * Resto)-Vars),((Proc_1 * Resto)-Vars)).

         abst2e(Acc ~ Proc_1-Vars, delta, (Proc_1R)):-
                 abst(Proc_1-Vars, Proc_1R),
                 evento(Acc, AccE),
                 evento(Vars,AccE).

         abst2e(Acc ~ Proc_1-Vars,  AccR~(Proc_1R), delta):-
                 abst(Proc_1-Vars, Proc_1R),
                 evento(Acc, AccE),
                 evento(Vars,VarsE),
                 subtract(AccE,VarsE,AccR).

/*************************************/
/*     Proceso que elimina deltas    */
/*************************************/

         quitadelta(X&Y,X1&Y1):-
                 quitadelta(X,X1),
                 quitadelta(Y,Y1).

         quitadelta(X+Y,Z):-
                 quitadelta(X,X1),
                 quitadelta(Y,Y1),
                 la_quita(X1, Y1, Z).

         quitadelta(X + delta, X1):-
                 quitadelta(X,X1).

         quitadelta(delta + Y, Y1):-
                 quitadelta(Y,Y1).

         quitadelta(X~Y,X1~Y):-
                 evento(X,X1),
proceso(Y).

         quitadelta(X~Y,X1~Y1):-
                 evento(X,X1),
                 quitadelta(Y,Y1).


         quitadelta(X,X):-proceso(X).

         quitadelta(delta,delta).


         la_quita(X,delta,X).
         la_quita(delta,X,X).
         la_quita(X,Y,X+Y).

/*********************************************/
/*   Orden lexico de los procesos            */
/*********************************************/

         r_proc(EV ~ PRO, EV_R ~ PRO_R):-
                 r_proc(PRO, PRO_R),   %10
                 evento(EV, EV_R).

         r_proc(PRO_A + PRO_B, PRO_R):-
                 r_proc(PRO_A, PRO_AR),
                 r_proc(PRO_B, PRO_BR),
```

```
                pon_proc(PRO_AR, +, PRO_BR, PRO_R).

        r_proc(PRO_A & PRO_B, PRO_R):-
                r_proc(PRO_A,PRO_AR),
                r_proc(PRO_B,PRO_BR), % 20
                pon_proc(PRO_AR, &, PRO_BR,PRO_R).

        r_proc(PRO, PRO):-
proceso(PRO).

/* acomoda el proceso en una sumatoria de eleccion */

        pon_proc(Eve~Proc + Resto,+, Proceso, Resultado):-
                pon_proc(Eve~Proc,+,Proceso, Res_tmp),  % 40
                pon_proc(Resto,+, Res_tmp, Resultado).

        pon_proc( Eve ~ Proc, +, Eve2 ~ Proc2 + Resto, Eve2 ~Proc2 + Res ):-
                Eve @>Eve2,
                pon_proc(Eve~Proc,+, Resto, Res).

        pon_proc( Eve ~ Proc, +, Eve2 ~ Proc2, Eve2 ~ Proc2 + Eve ~ Proc):-
Eve @> Eve2.

        pon_proc( Eve ~ Proc, +, Eve2 ~ Proc2 + Res, Eve~Proc + Eve2 ~Proc2 + Res ):-
Eve @<Eve2.

        pon_proc( Eve ~ Proc, +, Eve2 ~ Proc2 , Eve ~ Proc + Eve2 ~ Proc2 ):-
Eve @< Eve2.

/* regla P + P =~ P*/
        pon_proc( Eve ~ Proc, +, Eve ~ Proc + Res, Eve ~Proc + Res ).

        pon_proc( Eve ~ Proc, +, Eve ~ Proc , Eve ~ Proc).

/* regla aP + aQ =~ a(P&Q) */
        pon_proc( Eve ~ Proc1, +, Eve ~ Proc + Res, Eve ~ X + Res ):-
                proc_temp(X),
assert(definido(X <- Proc1 & Proc)).

        pon_proc( Eve ~ Proc1, +, Eve ~ Proc , Eve ~ X):-
                proc_temp(X),
assert(definido(X <- Proc1 & Proc)).


/* Acomoda procesos en una sumatoria no determinstica */

        pon_proc(Proc_1 & Resto, &, Proc_2, Resultado):-
                pon_proc(Proc_1, &, Proc_2,R_temp),
                pon_proc(Resto, &, R_temp, Resultado).

        pon_proc(Proc_1, &, Proc_2 & Resto, Proc_2 & Result):-
                act_p(Proc_1,Act_1),
                act_p(Proc_2,Act_2),
                Act_1 @> Act_2,
                pon_proc(Proc_1, &, Resto, Result).

        pon_proc(Proc_1, &, Proc_2 , Proc_2 & Proc_1):-
                act_p(Proc_1,Act_1),
                act_p(Proc_2,Act_2),
Act_1 @> Act_2.

        pon_proc(Proc_1, &, Proc_2 & Resto, Proc_1 & Proc_2 & Resto):-
                act_p(Proc_1,Act_1),
                act_p(Proc_2,Act_2),
Act_1 @< Act_2.

        pon_proc(Proc_1, &, Proc_2 , Proc_1 & Proc_2):-
                act_p(Proc_1,Act_1),
                act_p(Proc_2,Act_2),
Act_1 @< Act_2.

/* P & P = P */

        pon_proc(Proc_1, &, Proc_2 & Resto, Proc_2 & Resto):-
Proc_1 = Proc_2.

        pon_proc(Proc_1, &, Proc_2 , Proc_2):-
```

```
                Proc_1 = Proc_2,

        pon_proc(Act~Proc,&,Act~Proc & Resto,Act~Proc & Resto).

        pon_proc(Act~Proc,&, Act~Proc , Act~Proc).

/* aP&aQ= a(P&Q)*/

        pon_proc(Act~Proc,&,Act~Proc_1& Resto,Act~(Proc&Proc_1) & Resto).

        pon_proc(Act~Proc,&, Act~Proc_1 , Act~(Proc&Proc_1)).


/****************************************************/
/* Crea un rbol de aceptacin para un proceso dado */
/****************************************************/

        arbol(P&Q,Arbol_c, Arbol_r):-
                ramas(P,Ramas_1),
                p_ramas(Arbol_c,Ramas_1,Arbol_c1),
                arbol(Q,Arbol_c1,Arbol_r).

        arbol(P,Arbol_c,Arbol_r):-
                ramas(P,Ramas_1),
                p_ramas(Arbol_c, Ramas_1, Arbol_r).

        arbol(P,[],[P]):-
proceso(P).

/* Correccin 03/05/2000 */

        ramas(P+Q,La_rama):-
                ramas(P, Rama_1),
                ramas(Q, Rama_2),
                p_ramita(Rama_1,Rama_2,La_rama).

        ramas(A~P,[[A],[[A,Arbol]]]):-
                arbol(P,[],Arbol).

        ramas(P, [[],[P]]):-
proceso(P).

/* Correccin 03/05/2000 */

        p_ramita([[A1], R1],[A2, R2], [A2, Ramona]):-
                member(A1,A2),
                pones(R1,R2,Ramona).

        p_ramita([A1, R1],[A2, R2], [Acc1, Ramona]):-
                union(A1,A2,Acc),
                sort(Acc,Acc1),
                union(R1,R2,Ro),
                sort(Ro,Ramona).

        pones_r([[Ac,P]], [[Ac,Q]|Rest], [[Ac,X]|Rest]):-
                proc_temp(X),
assert(definido(X <- P & Q)).

        pones_r([[Ac,P]], [[Ax,Q]|Rest], [[Ax,Q]|Res]):-
                p_ramita([[Ac,P]], Rest, Res).


/***************************************************************/
/* Coloca las ramas ya formadas en su posicion correspondiente */
/***************************************************************/

        p_ramas( [[],Ramas], [C_acepta_1,Ramas_1], [C_acepta_1, Ramas_1]):-
                assert(error(la rama, [[],Ramas], no es valida)).

        p_ramas( [C_acepta_1,Ramas_1], [[],Ramas], [C_acepta_1, Ramas_1]):-
                assert(error(la rama, [[],Ramas], no es valida)).

        p_ramas([C_acepta,Ramas],[C_acepta_1,Ramas_1],[C_acepta, Ramas_n]):-
                member(C_acepta_1,C_acepta),
                mezcla_r(Ramas,Ramas_1,Ramas_n).

        /*p_ramas([[C_acepta],Ramas],[C_acepta_1,Ramas_1],[C_acepta_n, Ramas_n]):-
```

```
                  sort([C_acepta, C_acepta_1],C_acepta_n),*/

         p_ramas([C_acepta,Ramas],[C_acepta_1,Ramas_1],[C_acepta_n, Ramas_n]):-
                  sort([C_acepta_1|C_acepta],C_acepta_n),
                  mezcla_r(Ramas,Ramas_1,Ramas_n).


         p_ramas([],[X,Y],[[X],Y]).

         p_ramas(X,[],X).

/**************************************************/
/* Mezcla las ramas ya creadas (de forma ordenada) */
/**************************************************/

         mezcla_r([[A,Arbol_1]|Resto],[[A,Arbol_1]|Resto_1], Ramas_n):-
                  mezcla_r(Resto, [[A,Arbol_1]|Resto_1], Ramas_n).

         mezcla_r([[A,Arbol_1]|Resto],[[A,Arbol_n1]|Resto_1], Ramas_n):-
                  mezcla_r(Resto, [[A,[Arbol_1,Arbol_n1]]|Resto_1], Ramas_n).

         mezcla_r([[A,Arbol_1]|Resto],[[A_1,Arbol_n1]|Resto_1], [[A_1,Arbol_n1]|Ramas_n]):-
                  A@>A_1,
                  mezcla_r([[A,Arbol_1]|Resto],Resto_1, Ramas_n).

         mezcla_r([[A,Arbol_1]|Resto], [[A_1,Arbol_n1]|Resto_1], [[A,Arbol_1]|Ramas_n]):-
                  A@<A_1,
                  mezcla_r(Resto, [[A_1,Arbol_n1]|Resto_1], Ramas_n).

         mezcla_r([],Algo,Algo).

         mezcla_r(Algo,[],Algo).

/*********************************************/
/*          Compara dos rboles             */
/*********************************************/

         c_arbol([Acepta_1, Ramas_1], [Acepta_1, Ramas_2]):-
                  c_ramas(Ramas_1,Ramas_2).

         c_arbol([Acepta_1,Ramas_1], [Acepta_2, Ramas_2]):-
                  assert(error(orden,Acepta_1,no_igual_a,Acepta_2)),
                  c_ramas(Ramas_1,Ramas_2).

         c_ramas([[Eti_1, Arbol_1]|Resto_1], [[Eti_1, Arbol_2]|Resto_2]):-
                  c_arbol(Arbol_1,Arbol_2),
                  c_ramas(Resto_1,Resto_2).

         c_ramas([[Eti_1, Arbol_1]|Resto_1], [[Eti_2, Arbol_2]|Resto_2]):-
                  assert(error(guardia, Eti_1, no_igual_a, Eti_2)),
                  c_arbol(Arbol_1,Arbol_2),
                  c_ramas(Resto_1,Resto_2).

         c_ramas([X],[Y]):-
                  proceso(X),
proceso(Y).

         c_ramas([],[]).

/*********************************************/
/* Verifica si un conjunto dado es un evento */
/*********************************************/

         evento(EV, EVR):-
                  is_set(EV),
                  c_ev(EV),
                  sort(EV, EVR).

         evento(Ev, Ev1):-
                  list_to_set(Ev,Ev1),
                  c_ev(Ev1),
                  assert(advertencia(Ev,'no es un evento', 'variables duplicadas')).

         evento(EV, [EV]):-
 accion(EV).

c_ev([ELEM|REST]):-
```

```
                  accion(ELEH),
c_ev(REST).

c_ev([]).


/*******************************************/
/* Obtienen el orden y el Act de un proceso */
/*******************************************/

/* tanto el orden como el act de los procesos son importantes en la reescritura */
/* pero solo a partir de la relacin  que guardan con el operador de eleccin.  */

          orden_p(Acc^_+Resto,Orden):-
                  orden_p(Resto, Orden_r),
                  union(Acc,Orden_r,Orden_t),
                  sort(Orden_t, Orden).

          orden_p((X)+Resto,Orden):-
                  orden_p(X, Orden_X),
                  orden_p(Resto, Orden_r),
                  union(Orden_X,Orden_r,Orden_t),
                  sort(Orden_t, Orden).

          orden_p(Acc^_,Acc).

          orden_p(delta,[]).

/*********************************************/
/*                                           */
/*********************************************/
          act_p(Acc^_+Resto,Act_1):-
                  act_p(Resto,Act_1),
                  member(Acc, Act_1).

          act_p(Acc^_+Resto,[Acc, Act_1|R_act]):-
                  act_p(Resto,[Act_1|R_act]).

          act_p(Acc^_, [Acc]).

/***********************************************/
/*   Renombra Procesos si hay reetiquetacin    */
/***********************************************/

/* si existe la etiqueta para el proceso la devuelve*/
          renombra(Proc_1 ^ Vars, Et_Proc):-
                  definido(Et_Proc <- Proc_1 ^ Vars ),
proceso(Et_Proc).

/* si no existe la crea */

          renombra(Proc_1 ^ Vars, Et_Proc):-
                  proc_temp(Et_Proc),
assert(definido(Et_Proc <- Proc_1 ^ Vars )).

/***********************************************/
/*    Renombra Procesos si hay concurrencia     */
/***********************************************/

          renombra(Proc_1 * Proc_2, Et_Proc):-
                  definido(Et_Proc <- (Proc_1 * Proc_2) ),
proceso(Et_Proc).

          renombra(Proc_1 * Proc_2, Et_Proc):-
                  proc_temp(Et_Proc),
assert(definido(Et_Proc <- (Proc_1 * Proc_2 ))).

/***********************************************/
/*    Renombra Procesos si hay abstraccin      */
/***********************************************/

          renombra(Proc_1-Vars,Et_Proc):-
                  definido(Et_Proc <- Proc_1 - Vars ),
proceso(Et_Proc).

          renombra(Proc_1 - Vars, Et_Proc):-
                  proc_temp(Et_Proc),
```

```
assert(definido(Et_Proc <- Proc_1 - Vars )).

/**************************************************/
/* Obtiene el conjunto de aceptacin de un proceso */
/**************************************************/

           c_acepta(Proc,[]):-
proceso(Proc).

           c_acepta( Proc & Resto, Algo):-
                   c_acepta(Proc, Acp1),
                   c_acepta(Resto, Acp2),
                   union(Acp1,Acp2,Algo).

/* cuando el act de una sumatoria de eleccion ya ha sido */
/* aceptado no lo vuelve a incluir */

           c_acepta(Proc_1 & Resto,Act_Res):-
                   act_p(Proc_1,Act_1),
                   c_acepta(Resto,Act_Res),
                   member(Act_1,Act_Res).

           c_acepta(Proc_1 & Resto,Resultado):-
                   act_p(Proc_1,Act_1),
                   c_acepta(Resto,[C_1|RC_1]),
                   sort([Act_1,C_1|RC_1],Resultado).

           c_acepta(Resto, [Act_1]):-
                 act_p(Resto,Act_1).

/************************************************************/
/* Integra las consultas para devolver la Forma Normal Circal*/
/************************************************************/

           normal(Proc,ProcF):-
                   dist(Proc, ProcD),
                   comp(ProcD, ProcC),
                   abst(ProcC, ProcA),
                   quitadelta(ProcA, ProcE),
                   r_proc(ProcE, ProcF).

           normal_1(Proc,ProcF):-
                   print(Proc),
                   reet(Proc, ProcR),
                   dist(ProcR, ProcD),
                   comp(ProcD, ProcC),
                   abst(ProcC, ProcA),
                   quitadelta(ProcA, ProcE),
                   r_proc(ProcE, ProcF),
listing(definido).

/************************************************************/
/* Integra las consultas para devolver el Arbol de Mapeo de  */
/* la Forma Normal Circal                                   */
/************************************************************/

           n_arbol(Proc,ProcF):-
                   dist(Proc, ProcD),
                   comp(ProcD, ProcC),
                   abst(ProcC, ProcA),
                   arbol(ProcA, [], ProcF).

           n_arbol1(Proc,ProcF):-
                   print(Proc),
                   reet(Proc, ProcR),
                   dist(ProcR, ProcD),
                   comp(ProcD, ProcC),
                   abst(ProcC, ProcA),
                   quitadelta(ProcA, ProcE),
                   arbol(ProcE, [], ProcF),
listing(definido).

/***********************************************************/
/*   Crea las etiquetas para la reescritura de procesos     */
/***********************************************************/

proc_temp(Etiqueta):-
```

```
                        proceso(H),
                        concat(p_,X,H),
                        atom_chars(X,[Y1,Y2,Y3,Y4]),
                        plus(Y1,-48,Y11),
                        plus(Y2,-48,Y21),
                        plus(Y3,-48,Y31),
                        plus(Y4,-48,Y41),
                        la_lista([Y11,Y21,Y31,Y41],[Y12,Y22,Y32,Y42]),
                        concat_atom([p_,Y12,Y22,Y32,Y42],Etiqueta),
asserta(proceso(Etiqueta)).

            la_lista([A, B, C, D],[A, B, C, D1]):-
                        D<9,
                        plus(D, 1, D1).

            la_lista([A, B, C, _],[A, B, C1, 0]):-
                        C<9,
                        plus(C, 1, C1).

            la_lista([A, B, _, _],[A, B1, 0, 0]):-
                        B<9,
                        plus(B, 1, B1).

            la_lista([A, _, _, _],[A1, 0, 0, 0]):-
                        A<9,
                        plus(A, 1, A1).

            la_lista([ _, _, _, _],[0, 0, 0, 0]):-
assert(error('demasiadas variables')).

inicial_cc:-
                        assert(definido(X)),
                        retract(definido(X)),
                        asserta(proceso(p_0000)),
asserta(proceso(delta)).
```

Bibliografía

[Bai+ 91] Andrew Bailey & George Milne. **Using Circal to Analyse Sutherland's Asynchronous Micropipeline Design Style.** *Research Report HDV-13-91. University of Strathclyde, Departament of Computer Science, Glasgow U.K. Mayo 1991.*

[Bai+ 91a] Andrew Bailey, George McCaskill, Jim McIntosh, George Milne. **The Description and Automatic Verification of Digital Circuits in Circal.** *Research Report HDV-15-91. University of Strathclyde, Departament of Computer Science, Glasgow U.K. Mayo 1991.*

[Bar 98] Roman Barták. **Guide to Prolog Programming.** *Checoeslovaquia, Septiembre 1998.*

[Coc 90] W. P. Cockshott. **Circal and Quantum Phenomena.** *Research Report HDV-9-90. University of Strathclyde, Departament of Computer Science, Glasgow U.K. Abril 1990.*

[Hop+ 79] Jhon E. Hopcroft, Jeffrey D. Ullman. **Introduction to Automata Theory, Languages, and Computation.** *Addison - Wesley, Estados Unidos, 1979.*

[Lop 98] Miguel López Robles. **Prototipo para la Notación Z.** *Tesis de Licenciatura, Benemérita Universidad Autónoma de Puebla, Facultad de Ciencias de la Computación. Puebla, México. 1998.*

[Man 87] M. Morris Mano. **Diseño Digital.** *Prentice Hall. México. 1987.*

[Mao 92] Wenbo Mao. **The Re-Examination of Circal Bisimulation Semantics.** *Research Report HDV-10-90. University of*

Strathclyde, Departament of Computer Science, Glasgow U.K. Agosto 1992.

[McC 91] George A. McCaskill. **The XCircal User Guide and Reference Manual.** *Research Report HDV-18-91. University of Strathclyde, Departament of Computer Science, Glasgow U.K. Octubre 1991.*

[McC+ 91] George A. McCaskill, Jim McIntosh, Andrew Bailey. **Formal Circuit Design and Analysis with an Implementation of Circal.** *Research Report HDV-11-91. University of Strathclyde, Departament of Computer Science, Glasgow U.K. Mayo 1991.*

[McC+ 92] George A. McCaskill, George J. Milne. **Hardware Description and Verification Using the Circal - System.** *Research Report HDV-24-92. University of Strathclyde, Departament of Computer Science, Glasgow U.K. Junio 1992.*

[McC+ 95] George A. McCaskill,George J. Milne. **Sequential Circuit Analysis with a BDD Based Process Algebra System.** *Quality Systems and Software Limited, Edimburg Scotland U.K. and University of South Australia, School of Computer and Information Science. Australia. Junio 1995.*

[Mil+ 85] George Milne, P. A. Subrahmanyam. **Formal Aspects of VLSI Design** *North Holland. Holanda. 1985*

[Mil+ 88] George J. Milne, Mauro Pezzé. **Typed Circal: A High Level Framework for Hardware Verification, The Fusion of Harware Design and Verification.** *Proc. II IP Working Conf., Glasgow U.K.. Julio 1988*

[Mil 94] George Milne. **Formal Specification and Verification of Digital Systems.** *McGraw-Hill. Inglaterra. 1994*

[Mil 95] George J. Milne. **Modeling Digital Hardware in a Process Algebra.** *University of South Australia, School of Computer and Information Science. Australia. Mayo 1995.*

[Mlr 89] Robin Milner. **Comunication and Concurrency.** *Prentice Hall. Inglaterra. 1989.*

[Mol 94] Faron Moller. **The Semantics of Circal.** *Technical Report HDV-3-89. University of Strathclyde, Departament of Computer Science, Glasgow U.K. Mayo 1994.*

[Rus 94] Gordon Russell, Alex Cowie, John McInnes, Martin Bate, George Milne. **Simulating Vehicular Traffic Flows Using the Circal System.** *Research Report /94/157 [AISCIA-1-94]. University of Strathclyde, Departament of Computer Science, Glasgow U.K. University of South Australia, School of Computer and Information Science. Australia. Mayo 1994.*

[Wie 99] Jan Wielemaker. **SWI-Prolog 3.2.0 Reference Manual.** *Dept. of Social Science Informatics (SWI), University of Amsterdam. Holanda. 1999.*