UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS FÍSICAS

Departamento de Arquitectura de Computadores y Automática

SÍNTESIS FORMAL DE ALTO NIVEL POR DERIVACIÓN AUTOMÁTICA. ASPECTOS TEÓRICOS, METODOLÓGICOS Y PRÁCTICOS

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

José Manuel Mendías Cuadros

Madrid, 2003

ISBN: **978-84-669-1586-1**

Síntesis formal de alto nivel por derivación automática

Aspectos teóricos, metodológicos y prácticos

José M. Mendías Cuadros

TESIS DOCTORAL

Universidad Complutense de Madrid
Dpto. de Arquitectura de Computadores y Automática

Síntesis formal de alto nivel por derivación automática: aspectos teóricos, metodológicos y prácticos.

Memoria presentada por José Manuel Mendías Cuadros para optar al grado de Doctor en Ciencias Físicas por la Universidad Complutense de Madrid, realizada bajo la dirección de D. Román Hermida Correa.

Madrid, 18 de Mayo de 1998

Este trabajo ha sido posible gracias a la Comisión Interministerial de Ciencia y Tecnología, por las ayudas recibidas a través de los proyectos CICYT TIC 92/0088, CICYT TIC 94/0725-C03-02 y CICYT TIC 96/1071 y también gracias a la Comunidad Europea a través del proyecto HCM CHRX-CT94-0259.

AGRADECIMIENTOS

Agradezco a Román Hermida, no sólo el esfuerzo dedicado a dirigir esta tesis, sino principalmente la apuesta que realizó al confiar en unas ideas 'extrañas' en un comienzo, y que han quedado felizmente recogidas en esta memoria.

Agradezco a Olga Peñalba la reciente pero definitiva ayuda que me ha prestado en una de las más arduas tareas: la lectura, corrección y discusión de estas quinientas páginas.

Agradezco individualmente a Pablo Rupérez, a Juan Antonio Maestro, a Hortensia Mecha, a Katzalín Olcoz, a Silvia Molina, a Daniel Mozos y a Julio Septién todos estos largos años de convivencia.

Agradezco colectivamente a mis compañeros del que fuera el Departamento de Informática y Automática por acogerme entre ellos, en especial a los que hoy forman el Departamento de Arquitectura de Computadores y Automática, destacando a Milagros Fernández y a Francisco Tirado.

Agradezco a unos tales Stoy, Wirsing, Ehrig y tantos otros que, sin conocerlos yo y sin saberlo ellos, contribuyeron a que esta tesis tuviera sentido.

Agradezco ese error de Synopsys que tras 4 días de convalecencia me abrió los ojos para comprobar que las máquinas también son humanas y se equivocan.

Agradezco a Justi, a Ma, a Rafa, a Asun y Ana Mari su ayuda que, aunque no fue en lo técnico, si lo fue en lo emocional. Agradezco a Antonio su Toshiba con el que escribí en los trenes y los autobuses, las más notables páginas de esta memoria. Y finalmente agradezco enormemente a todos mis amigos que no me hayan olvidado en estos años de encierro.

a ella, que lo sabe

Contenido

Glosario de notación

$\mathbf{N}_+$	el conjunto de los números naturales distintos del 0
$(D_1 \rightarrow D_2)$	el conjunto de funciones de dominio D_1 y codominio D_2
$f : D_1 \rightarrow D_2$	una función (total) de dominio D_1 y codominio D_2
$f : D_1 \rightharpoonup D_2$	una función parcial de dominio D_1 y codominio D_2
$\lambda x.e$	una λ-expresión de tipo abstracción que representa a una función de un parámetro
$(D_1 \times ... \times D_n)$	el conjunto de tuplas de n elementos pertenecientes a los conjuntos $D_1 ... D_n$
$\lambda(x_1...x_n).(e_1...e_m)$	una λ-expresión de tipo abstracción que representa a m funciones de n parámetros
δ	un elemento de retardo
D^∞	el conjunto de todas las cadenas infinitas de elementos del conjunto D
D^*	el conjunto de todas las cadenas finitas de elementos del conjunto D
ε	la cadena vacía
$u.w$	la concatenación de las cadenas u y w
(S, Σ)	una signatura heterogénea, donde S es el un conjunto de géneros y Σ es una familia de símbolos de operación

Σ	notación abreviada de una signatura heterogénea
$\Sigma_{w,s}$	el conjunto de símbolos de aridad w y género *s* de una signatura heterogénea
σ	un símbolo genérico de operación
X	un conjunto genérico de variables
X_s	un conjunto genérico de variables de género *s*
$T_\Sigma(X)$	el conjunto de términos de la signatura Σ
$T_{\Sigma,s}(X)$	el conjunto de términos de género *s* de la signatura Σ
T_Σ	el conjunto de términos cerrados de la signatura Σ
var(*t*)	el conjunto de variables que ocurren en el término *t*
A	un álgebra genérica
A_s	el conjunto soporte del género *s* perteneciente al álgebra *A*
$A_\sigma^{w,s}$	la función soporte del símbolo de operación σ perteneciente al álgebra *A*
Z	el álgebra de los números enteros
Z_{op}	la función entera denotada habitualmente por el símbolo *op*
B	el álgebra de los booleanos
B_{op}	la función booleana denotada habitualmente por el símbolo *op*
$\mu : X \to A$	una valoración del conjunto de variables *X* sobre el álgebra *A*
$\hat{\mu} : T_\Sigma(X) \to A$	la interpretación del conjunto de términos de la signatura Σ inducida por la valoración μ
t^A	la interpretación del término cerrado *t* en el álgebra *A*

(Y, t_L, t_R)	una ecuación o una regla de reescritura
$t_L = t_R$	notación abreviada de una ecuación
$t_L \rightarrow t_R$	notación abreviada de una regla de reescritura
$A \models e$	el álgebra *A* satisface la ecuación *e* (o la ecuación *e* es válida en *A*
#	el símbolo comodín que denota indiferencia
$\approx$	la relación de compatibilidad
$\Sigma^{\#}$	extensión para denotar indiferencia de la signatura Σ
$A^{\#}$	extensión para denotar indiferencia del álgebra *A*
$\perp$	el símbolo fondo que denota indefinición
$\sqsubseteq$	una relación de orden parcial
$A_{\perp}$	extensión para denotar indefinición del álgebra *A*
$(\mathbf{N}_{+} \rightarrow A)$	extensión temporal del álgebra *A*
fix	el operador punto fijo
fby	el operador temporal de retardo
$Lu(\Sigma)$	la signatura obtenida por la adición a la signatura Σ de los símbolos #, *fby*, *next*, >>, <<, y ||
$(\Sigma, X, Ins, Outs, \varphi)$	Una especificación ecuacional donde Σ es una signatura, *X* es el conjunto de señales, *Ins* es el conjunto de puertos de entrada, *Outs* es el conjunto de puertos de salida y φ es el cuerpo ecuacional (puede encontrarse con una especificación algebraica en lugar de con una signatura)
$(x, \varphi(x))$	una definición perteneciente al cuerpo ecuacional de una especificación ecuacional
$x = \varphi(x)$	notación abreviada de una definición
$Lu(A)$	el álgebra obtenida por la extensión temporal del

	álgebra alcanzada tras la extensión de *A* para denotar indiferencia y para denotar indefinición
$(x_1, ..., x_n)$	una tupla de *n* elementos
$\downarrow$	operador restricción de tuplas
n..m	el subconjunto de los números naturales comprendido entre *n* y *m*, ambos inclusive
C[*ds* **]**	la semántica de la especificación ecuacional *ds*
E[*t* **]**	la semántica del término *t* incluido en una definición perteneciente a una especificación ecuacional
pos(*t*)	el conjunto de posiciones válidas dentro del término *t*
t/u	el subtérmino ubicado en la posición *u* del término *t*
t[*u*]	el símbolo ubicado en la posición *u* del término *t*
$pos_0(t)$	el conjunto de posiciones ocupadas por variables del término *t*
t[*u*←*s*]	el reemplazamiento del subtérmino ubicado en la posición *u* del término *t* por el término *s*
$\rho : Y \rightarrow T_\Sigma(X)$	la substitución del conjunto de variables *Y* por términos del conjunto de términos de la signatura Σ
(*z*, *t*)	un término recursivo
$\hat{\mu}_{rec}$	la interpretación del conjunto de términos recursivos de la signatura Σ inducida por la valoración μ
$ds_1 \rightarrow ds_2$	la especificación ecuacional ds_2 se deriva de la especificación ecuacional ds_1
$ds_1 \rightarrow_{RT} ds_2$	la especificación ecuacional ds_2 se deriva de la especificación ecuacional ds_1 mediante la aplicación de la regla de transformación *RT*
$ds_1 =_Z ds_2$	la equivalencia en conducta de las especificaciones ecuacionales ds_1 y ds_2 para un conjunto común de

	señales Z
$ds_1 \approx_Z ds_2$	la equivalencia débil en conducta de especificaciones ecuacionales ds_1 y ds_2 para un conjunto común de señales Z
next	el operador temporal de anticipación
sample	el operador temporal de muestreo
>>	el símbolo de operación que denota al operador temporal de muestreo
replicate	el operador temporal de replicación
<<	el símbolo de operación que denota al operador temporal de replicación
interleave	el operador temporal de multiplexación
||	el símbolo de operación que denota al operador temporal de multiplexación
« x_1, x_2, ... »	una secuencia infinita de valores
Ops_{FU}	conjunto de operaciones no temporales y no constantes de una especificación ecuacional
Ops_{CTE}	conjunto de operaciones no temporales y constantes de una especificación ecuacional
Ops_{ST}	conjunto de operaciones temporales de retardo de una especificación ecuacional
$Recs_{FU}$	conjunto de instancias funcionales de un circuito
$Recs_{ST}$	conjunto de instancias de retardador de un circuito
$Recs_{MUX}$	conjunto de instancias de multiplexor de un circuito
λ	latencia de una planificación
τ	planificación de operaciones y actualizaciones
α_{FU}	asignación de recursos funcionales

α_{ST}	asignación de recursos de almacenamiento
α_{MUX}	asignación de recursos de encaminamiento
α_{ALGN}	alineación de operaciones
(S, Σ, E)	una especificación algebraica, donde S es el un conjunto de géneros, Σ es una familia de símbolos de operación y E es un conjunto de ecuaciones
Alg_{SPEC}	el conjunto de todas las álgebras que satisfacen todas las ecuaciones de la especificación algebraica *SPEC*
$\sim$	equivalencia de términos cerrados
T_{SPEC}	el álgebra de términos cociente definida a partir de la especificación algebraica *SPEC*
Con_s	conjunto de constructores del género s
Gen_s	conjunto de generadores del género s
Mod_s	conjunto de modificadores del género s
Obs_s	conjunto de observadores del género s
$t_L \rightarrow_r t_R$	el término t_L se reescribe como t_R vía la regla de reescritura r
$t_L \rightarrow_e t_R$	el término t_L se reescribe como t_R vía la regla de reescritura inducida por la ecuación e
(Σ, E)	una implementación algebraica, donde Σ es una familia de símbolos de operación y E es un conjunto de ecuaciones
$Th(A)$	la teoría ecuacional del álgebra A
$E \vdash e$	la ecuación e se deriva a partir del conjunto de ecuaciones E

Capítulo 1

Corrección y diseño

(...) I can remember the exact
instant when I realized that a large
part of my life from then on was
going to be spent in finding
mistakes in my own programs
Maurice Wilkes

It isn't that they can't see
the solution. It is that they
can't see the problem
G. K. Chesterton

En cualquier actividad humana la confianza que se deposita en la corrección de logros pasados es uno de los pilares para afrontar nuevos desafíos. Si cada vez que se intentara solventar un nuevo problema fuera necesario partir desde cero, llegaría un momento en el que el progreso sería imposible. Así, un arquitecto no necesita reinventar el hormigón cada vez que desea proyectar un edificio, ni un cirujano redescubrir la anatomía humana antes de comenzar una intervención. Tanto el uno como el otro, se apoyan en

resultados obtenidos por aquellos que le precedieron en el oficio, confiando en que lo hicieran con corrección.

El diseño hardware, como actividad humana que es, también basa sus éxitos en la confianza que todo diseñador pone en la corrección de ciertas técnicas, de ciertos algoritmos o de ciertas componentes que interconecta. Confianza que no se circunscribe al ámbito científico sino que transciende a una sociedad que, cada vez más dependiente de la tecnología digital, se está edificando en base al buen funcionamiento de los sistemas informáticos.

Uno de los grandes hitos en la mejora de la calidad de los diseños y en el aumento de la confianza en su corrección, ha venido marcado por la incorporación de herramientas de diseño asistido a los ciclos clásicos de diseño manual. Gracias a esta incorporación, es posible que muchas tareas repetitivas y propensas a error se realicen mecánicamente y que un gran número de técnicas de diseño puedan ser efectuadas por algoritmos automáticos de optimización sin casi ninguna mediación humana. Dicha incorporación ha tenido tanto impacto en la reducción de tiempos de diseño, en el crecimiento de la complejidad de los circuitos alcanzables y en el aumento del nivel de abstracción requerido en las especificaciones, que en la actualidad no puede concebirse el diseño digital sin el soporte, aunque sea mínimo, de herramientas, que van desde simples capturadores de esquemas hasta complicados entornos de diseño automático.

En cualquier caso este éxito ha desencadenado una euforia (muchas veces avivada por la floreciente industria del diseño asistido) que ha propiciado que se acepten sin demasiada reflexión afirmaciones tan aventuradas como el marchamo de **correcto por construcción** que, acuñado a principios de los 80, hoy se otorgan sin pudor las herramientas de diseño automático. Sin embargo, cualquier diseñador que utilice estas herramientas, acabará por comprobar que la herramienta que le vendieron por infalible falla, y que un diseño que es *correcto por construcción* se convierte en *incorrecto por*

ingenuidad. Y así de repente, junto con el diseño perdido, se desvanece la confianza y, en cierto modo, la ciencia y el propio progreso sufren un nuevo revés.

¿Por qué fallan las herramientas?.

El desarrollo de las herramientas de diseño asistido ha sido casi tan espectacular como el desarrollo que ha sufrido la propia tecnología microelectrónica. En menos de 20 años se ha pasado de tener simples herramientas de emplazamiento y trazado (*placement & routing*) a disponer de la primera generación de herramientas de codiseño hardware-software, a un ritmo de progresión que se acelera a la par que aumenta la demanda de productos cada vez más potentes.

Como consecuencia de esto, los algoritmos de síntesis han tenido que evolucionar muy rápidamente adaptándose a dominios de aplicación de complejidad creciente. En muchos casos este desarrollo, fuertemente influenciado por criterios mercadotécnicos, se ha hecho sin las garantías que ofrece una sólida base formal, por lo que en un corto espacio de tiempo se han desarrollado complicadas herramientas, basadas en algoritmos muy sofisticados que manipulan intrincadas estructuras de datos. Herramientas formadas por cientos de miles de líneas de código escritas separadamente por grandes equipos de programadores, que dada su complejidad, sólo habrán podido ser parcialmente testadas y en ningún caso verificadas formalmente.

Viendo este panorama es fácil intuir que las herramientas automáticas (al igual que cualquier software complejo) son un caldo de cultivo inmejorable en donde pueden crecer los pequeños descuidos. Pequeños descuidos que si bien para un programador pueden tener una limitada importancia, desde la perspectiva del diseñador tienen efectos demoledores, ya que cuando se

ubican en el código de una herramienta de diseño se multiplican en cientos de circuitos erróneos que nunca podrán salir a un mercado hardware, mucho más exigente que el software[†].

Si a estos problemas típicamente software (que con una buena metodología podrían minimizarse) se añaden otros más profundamente enraizados con el diseño hardware, como son el tradicional distanciamiento que se mantiene respecto de los modelos formales de los lenguajes de especificación (cuando no su total ignorancia), las continuas conversiones que se hacen de dichos lenguajes a las más variadas representaciones internas que (en la mayor parte de las veces) se manipulan sin atender a implicaciones semánticas y la nula formalización que se hace de las decisiones de diseño, hacen que el paradigma de *corrección por construcción* pueda ser justificadamente tachado de falaz.

¿Cómo enfrentarse a los errores?.

La comunidad investigadora, consciente de la importancia que tiene la corrección de los diseños, va prestando poco a poco más atención a este aspecto y resulta sorprendente comprobar cómo el estudio de temas relacionados con la corrección en particular y con los métodos formales en general, que hace algunos años se limitaba a simposios específicos, hoy se abre paso en conferencias tradicionalmente dedicadas a temas más 'prácticos'.

En cualquier caso, aún hoy en día, el método más extendido para validar la corrección de una implementación es la **simulación post-síntesis**, un método heredado de los primeros tiempos del diseño manual sobre el que se

[†] Recuérdese el famoso incidente que supuso la detección de un error en el algoritmo de división en coma flotante del microprocesador Pentium una vez fabricado. Un error de diseño que costó a Intel $200.000 en depurar y $480.000.000 en reemplazar las unidades defectuosas que ya había vendido [SSSF97].

sigue publicando [HaKR98]. La razón de ello es su consustancialidad con el diseño: la simulación y el diseño han estado unidos desde los albores de la informática y por el momento resulta inconcebible que se pueda modificar este hábito aunque todos los investigadores coincidan que para diseños medianamente complicados resulta ineficaz.

La razón para no aceptar la simulación como un método de prueba de corrección es que sólo es capaz de asegurarla si todas las posibilidades de ejecución del circuito se comprueban. Obviamente en cuanto el circuito realice una función ligeramente complicada y sobre todo cuando ésta sea secuencial, la simulación exhaustiva será imposible por lo que, parafraseando a E.W. Dijkstra, este método sólo podrá probar la presencia de errores pero nunca la ausencia de ellos. Para solventar esta gran deficiencia se está comenzando a utilizar la verificación formal [McFa93][SaMS96].

La **verificación formal** trata de demostrar matemáticamente que la implementación satisface la especificación. Para ello se formalizan matemáticamente tanto la especificación como la implementación, y dentro de algún cálculo formal se demuestra que la función calculada por el circuito es la misma que la que requiere la especificación. El problema que en el pasado dificultó su incorporación al ciclo de diseño, es que requería el aprendizaje de técnicas formales que el diseñador medio estimaba no justificables. En la actualidad, conforme el problema de la simulación se hace más evidente† y los sistemas de verificación más accesibles [Kurs97], la reticencia inicial va desapareciendo. No obstante, lejos de ser la verificación formal la solución a todos los problemas, algunos autores [Gupt92] muestran que estas técnicas pueden alcanzar complejidades exponenciales e incluso ser indecibles para cierto tipo de diseños.

---

† Para la validación de una nueva generación del procesador MIPS, se utilizaron durante 7 meses más de 100 *workstations* de última generación para la simulación del nuevo diseño [HeHe96], y todo este esfuerzo no pudo prevenir de algunos errores cruciales que permanecieron ocultos hasta la fabricación del primer prototipo.

No obstante, para demostrar la corrección de los diseños existe una tercera alternativa: la **síntesis formal** [KBES96]. La diferencia de este tipo de síntesis respecto a la verificación formal es que mientras que esta última trata de demostrar que la implementación y la especificación describen la misma función, la síntesis formal trata de derivar la implementación a partir de la especificación (tal y como se realiza en cualquier cálculo en las matemáticas). A costa de simplificar en exceso, podría decirse que la verificación formal obtiene la función especificada a partir de la implementada mediante sucesivos pasos de abstracción, mientras que la síntesis formal obtiene la función implementada a partir de la especificada mediante sucesivos pasos de refinamiento. En cualquier caso ambos métodos son igual de seguros o, si se prefiere, igual de inseguros: ya que ambos exigen formalizaciones de la especificación y de la implementación, ambos realizan todos pasos dentro de un cálculo formal, ambos realizan dicho cálculo mediante la aplicación de un muy pequeño núcleo de derivaciones muy simples y ambos son tan fiables como fiable sea la implementación software de dicho núcleo.

Por otro lado, aunque la síntesis formal siga una metodología similar a la síntesis convencional, las diferencias entre ambas son notables. En un sistema convencional el hardware se representa mediante estructuras arbitrarias de datos y no existen restricciones sobre el tipo de manipulaciones que pueden hacerse sobre ellas. Por su parte, en los sistemas formales el hardware se representa mediante términos y fórmulas, y solamente se permiten transformaciones que se hayan demostrado previamente que conservan el comportamiento dentro de cierto cálculo lógico. La formalización de las estructuras junto con las restricciones en la transformación de las mismas son decisivas para que estos sistemas puedan garantizar la corrección del proceso de síntesis ya que, implícitamente, el propio proceso deberá constituir la prueba de corrección del circuito.

Tradicionalmente la síntesis formal ha permanecido latente en el ámbito teórico de los métodos formales ya que constituía un método elegante para formalizar las técnicas de diseño. Esto ha hecho que en la mayor parte de los casos los métodos desarrollados no se hayan materializado en ningún sistema real y sólo pueden encontrarse unos pocos autores que hayan apostado por la realización de prototipos. En cualquier caso, el principal problema que se ha encontrado para incorporar estos prototipos al ciclo de diseño y para que supongan una alternativa aceptable por la comunidad diseñadora, es que la mayor parte de ellos no son automáticos, sino que son mecanizados, es decir, que si bien cualquier transformación puede realizarse automáticamente sobre un soporte simbólico procesable por computador, la selección de qué transformaciones realizar para conseguir ciertos objetivos de optimización, o incluso para alcanzar un circuito según cierto modelo de cálculo, debe hacerse interactivamente mediante guiado del diseñador.

La investigación que se recoge en esta memoria está orientada a afrontar la verificación de diseños desde esta tercera perspectiva pero tratando de solventar su principal problema: la falta de automatización. Por ello se desarrollará un sistema de síntesis formal orientado al diseño algorítmico que pueda ser guiado tanto por un diseñador como por un algoritmo (potencialmente incorrecto) de optimización, de manera que si el proceso de diseño finaliza con éxito, el circuito obtenido sea realmente correcto por construcción, y si no es así, el sistema informe sobre las decisiones de diseño incorrectamente realizadas por el algoritmo externo de optimización.

Para que el lector pueda poner en perspectiva la investigación realizada, a continuación describiré algunos sistemas de síntesis formal (y no formal) que han sido desarrollados por otros autores. Asimismo, en §1.2 y en §1.3 concretaré, respectivamente, los objetivos de mi trabajo y la estructura de esta memoria.

1.1 Propuestas para la síntesis correcta.

Tanto en el frente de los métodos formales como en el frente del diseño clásico han aparecido propuestas para idear sistemas de síntesis correcta. Sin embargo, dada la tradicional separación de ambos mundos, los resultados que se obtienen en un campo suelen ignorarse en el otro, por lo que hasta la fecha no se ha desarrollado ninguna alternativa válida desde todos los puntos de vista.

Las propuestas que se realizan desde el campo de los métodos formales suelen primar los aspectos teóricos respecto de los prácticos. Por ello, si bien obtienen sistemas elegantes y correctos, suelen ignorar cuestiones fundamentales para una implantación realista. Así, la mayor parte de las investigaciones suelen limitarse a la definición de un formalismo más o menos adecuado (que suele ser poco conveniente para el diseñador medio), a la elección de un método de cálculo y a la comprobación con pequeños ejemplos de que pueden obtenerse formalmente ciertas implementaciones. En cualquier caso, no clasifican el tipo de transformaciones típicas a realizar para obtener metódicamente cierto tipo de resultados, ni estudian el rango de diseño abarcables, ni suelan considerar aspectos de eficiencia y complejidad para la implantación del método en un computador.

Por su parte, las propuestas que se realizan desde el campo de la síntesis clásica adolecen siempre del mismo problema: la falta de corrección. En general, aunque obtienen sistemas muy versátiles de síntesis, caen habitualmente en el mismo error: creer que sin un soporte semántico es posible asegurar la corrección. Así, construyen sistemas transformativos que sintetizan circuitos paso a paso mediante la aplicación de pequeñas transformaciones que pretendidamente conservan el comportamiento. Sin embargo, dado que no formalizan ni la especificación, ni la implementación ni las transformaciones, los sistemas no pueden asegurar que conservan el

comportamiento y por tanto tampoco son capaces de demostrar la corrección de los circuitos que obtienen.

Para conseguir una idea general del estado de las cosas, a continuación pasaré a comentar los sistemas formales más representativos en mi opinión, al igual que repasaré brevemente los sistemas transformacionales no formales más populares.

1.1.1 T-Ruby.

T-Ruby [ShR93a][ShR95a] es un sistema semiautomático de síntesis desarrollado en la universidad técnica de Dinamarca que permite la transformación formal de circuitos especificados en un subconjunto del lenguaje *Ruby*, llamado *Pure Ruby.*

Ruby [JoSh90] es un lenguaje que permite la especificación de una conducta digital síncrona en términos de su abstracción relacional. Así, el comportamiento de un circuito se describe mediante una relación binaria definida por la composición de un conjunto de *relaciones básicas* (predefinidas en el lenguaje o descritas mediante la especificación funcional de sus funciones características) que se realiza utilizando un conjunto de *combinadores* (funciones de orden superior que pueden definirse explícitamente de manera funcional).

Por su parte Pure Ruby [Ros90a][Ros90b] es un subconjunto del anterior lenguaje que restringe las relaciones (circuitos) y combinadores válidos a aquellos que se puedan construir (utilizando una sintaxis similar al del λ-cálculo con tipos) a partir de cuatro únicos elementos: dos relaciones y dos combinadores. Las relaciones son: *spread f*, que permite extender punto a punto sobre *streams* la función característica *f* de una relación, por lo que ofrece un método para definir circuitos combinacionales muestreados síncronamente, y la relación *D* que denota un elemento de retardo. Los

combinadores son: la composición serie y la composición paralela de dos relaciones.

El sistema T-Ruby ofrece un soporte mecanizado para que el diseñador realice las transformaciones formales que requiere un proceso de diseño y para que pueda simular el comportamiento de una relación. El circuito obtenido será correcto siempre y cuando las transformaciones realizadas sean correctas, por lo que para comprobar la corrección de cualquier transformación que se efectúe, el sistema facilita un interfaz con un demostrador de teoremas externo. El ciclo típico de diseño en T-Ruby involucra 3 tareas: transformación, prueba y análisis de causalidad.

Durante la transformación [ShR93b], el sistema permite al diseñador elegir interactivamente qué transformaciones básicas (conocidas como *tactics*) o qué grupos de transformaciones (conocidos como *tacticals)* aplicar sobre un diseño. Una transformación básica puede ser una regla de reescritura definida explícitamente, una expansión de la definición de una relación, una expansión de una definición de un combinador o un lema (en forma de regla de reescritura) obtenido en un proceso anterior de transformación. Por su parte, las transformaciones básicas pueden agruparse en tacticals de manera conjuntiva, de manera disyuntiva o de manera repetitiva.

Una vez finalizada la transformación de una conducta, toda regla de reescritura explícita que se haya utilizado en dicha fase debe ser probada. Para ello se utiliza una versión modificada [Rasm96] del demostrador de teoremas Isabelle [Paul94], que permite la demostración semiautomática de dichas reglas.

Finalmente, dado que el enfoque relacional permite la definición de comportamientos que no son implementables, el análisis de causalidad permite extraer aquellas partes de una relación que pueden ser implementadas en un circuito.

En general usando el lenguaje Ruby, y utilizando el sistema T-Ruby en particular, es posible reproducir formalmente algunas técnicas de síntesis sobre clases especificas de circuitos. Así es posible encontrar ejemplos de derivación que obtienen implementaciones serie de sumadores a partir de descripciones paralelas [JoS91a], procesos de transformación de redes iterativas [JoS91b][Luk93], o métodos para la derivación de circuitos sistólicos [Sand94].

La mayor crítica que se puede hacer al sistema T-Ruby es que, aparte de no ser automático, está basado en un lenguaje que si bien tiene una innegable elegancia desde el punto de vista matemático, es completamente inadecuado desde el punto de vista de un diseñador. Primero, porque para poderlo aplicar en diseño hardware ha sido necesario reducir enormemente sus capacidades (a dos relaciones y dos combinadores). Y segundo, aún habiéndolo reducido, es necesario un profundo conocimiento de métodos formales para poder dirigir cualquier proceso de diseño.

EJEMPLO 1.1

Como ejemplo de la poca conveniencia del lenguaje Pure Ruby desde el punto de vista de un diseñador medio, mostraré la especificación de un simple filtro FIR que tiene el siguiente comportamiento temporal:

$$y(t) = \sum_{i=0}^{N-1} x(t-i) * c_i(t)$$

Esta funcionalidad queda descrita en Pure Ruby [ShR95b] como:

$$\text{Fst } \text{shift}_n \; ; \; \text{zip}_n \; ; \; (\; \text{map}_n \; * \;) \; ; \; \text{sum}_n$$

donde algunas de las funciones utilizadas deben definirse como:

$\text{Fst} \equiv \lambda H : \alpha \sim \beta \; . \; [\; H, \; i \;]$

$\text{shift}_n \equiv \text{fork}_n \; ; \; (\; \text{tri}_n \; D \;)$

$\text{zip} \equiv \lambda n : \text{int} \; . \; (\text{ if } n=0 \text{ then } [\text{NNIL},\text{NNIL}]; \pi_1$

$\quad \text{else } [\text{apl}_{n-1}{}^{-1}, \text{apl}_{n-1}{}^{-1}];$

(reorg;Snd(reorg^{-1};(Fst cross);reorg);reorg^{-1});Snd(zip$_{n-1}$);apl$_{n-1}$)

map ≡ λn :int, R:α~β . (if n=0 then NNIL

else (apr$_{n-1}$)$^{-1}$;[(map$_{n-1}$R),R];apr$_{n-1}$)

∗ ≡ spread(λ(x,y):int×int, z:int . (z = x∗y))

quedando aún por definir *suml*, *i*, *fork*, *tri*, *NNIL*, π_1, *apl*, $^{-1}$, *reorg*, *Snd*, *cross*, y *apr*, que lo hacen en función de +, *row*, *mapf*, ^{i}, *dub*, *nlist*, *ncons*, *Lwir*, *Rwir*, ∧ y *nsnoc*. Y el proceso de especificación continúa hasta la definición de todos los símbolos en base a los cuatro elementos básicos de Pure Ruby.

Como puede observarse, aparte de que la necesidad de tal complejidad sea cuestionable en sí misma, es inconcebible que este método de especificación de conductas pueda ser aceptado por la comunidad diseñadora.

1.1.2 DDD.

El sistema DDD (Digital Design Derivation) desarrollado en la Universidad de Indiana [JoBo91][JoBB88] permite la derivación interactiva de circuitos digitales síncronos de nivel RT descritos en *Schema* [ReCl86] (un dialecto de primer orden del lenguaje Lisp), a partir de especificaciones de alto nivel descritas en el mismo lenguaje.

Un proceso de diseño en DDD parte de un comportamiento iterativo descrito mediante composición de funciones recursivas que operan sobre tipos compuestos. A continuación lo serializa obteniendo una descripción formada por un conjunto de definiciones simultáneas de señales que utilizan operaciones y retardos [John84], seguidamente aplica las llamadas factorizaciones [John89] para reducir el número de subexpresiones comunes

y encapsular comportamientos complejos, y finaliza obteniendo un conjunto de subfunciones que manipulan tipos simples.

Para realizar este proceso aplica una serie de transformaciones basadas en álgebra funcional (transformaciones que no ha sido probado que sean correctas), que permiten añadir, agrupar y eliminar definiciones, reemplazar expresiones por otras equivalentes y reemplazar identificadores por su definición. La base de la factorización es la aplicación de la propiedad de distributividad de las operaciones de selección (que modelan multiplexores) respecto del resto de los operadores de una descripción. Algunos ejemplos de su uso para el diseño transformativo pueden encontrarse en [BoJo89][BoJo93][ZhJo93].

Actualmente este sistema ha evolucionado adoptando como representación intermedia un lenguaje tabular que describe una máquina de estados algorítmica y en el que las transformaciones, que antes podían hacerse directamente sobre Schema, se han redefinido para que manipulen filas y columnas de dicha representación [RaTJ93][John97].

La principal deficiencia que muestra es que realizando un proceso de diseño que podría catalogarse como síntesis de alto nivel, ignora completamente los desarrollos realizados por otros autores en este campo, limitándose a realizar planificaciones y reusos triviales aplicando siempre a mano técnicas de transformación que, además, en ningún momento se demuestran que sean correctas.

1.1.3 HASH.

El sistema HASH (Higher order logic Applied to Synthesis of Hardware) desarrollado en la universidad de Karlsruhe, es un conjunto de normas que definen cómo utilizar el demostrador de teoremas HOL [GoMe93] para

verificar formalmente los resultados de algunas técnicas de diseño de alto nivel.

La principal novedad de este sistema es que diferencia claramente dos tareas en todo proceso de diseño: la exploración de soluciones y la transformación formal. Mientras que la exploración de soluciones determina aspectos relativos a la calidad del circuito frente a ciertas ligaduras, la transformación formal determina la corrección del diseño. Así este sistema, partiendo de la especificación de un comportamiento no iterativo descrito mediante una λ-expresión no recursiva de primer orden, confía en herramientas externas para que realicen la exploración de soluciones. Una vez obtenido un circuito, define normas para la formalización de las decisiones de diseño adoptadas durante dicha exploración mediante una λ-expresión de orden superior y define un mecanismo automático para verificar formalmente el diseño utilizando HOL.

Las técnicas de diseño que pueden verificarse en HASH incluyen, la retemporización de circuitos de nivel RT [EiKB97], las optimizaciones de máquinas de estado [EiK95a] y algunas fases de la síntesis de alto nivel tales como planificación y asignación [EiK95b][EiBK96].

Sin embargo, nuevamente es un sistema que, aún teniendo una elegancia teórica notable, impone limitaciones que hacen que sea difícil su aceptación práctica. El primer problema es que sólo pueden especificarse y verificarse comportamientos lineales. Esto se traduce en que sea imposible sintetizar especificaciones iterativas o recursivas y que, desde el punto de vista hardware, no sea posible representar directamente circuitos con memoria o con realimentación de cálculos. Las consecuencias de esta limitación son claras: cualquier cómputo iterativo o cualquier circuito realimentado (que son los más, tras un proceso de SAN) debe describirse de una manera 'especial' que por sí sola sólo es capaz de describir una porción incompleta del comportamiento, por lo que siempre queda un núcleo sin verificar. El segundo

problema es que el paradigma de corrección sólo se asegura a costa de que el proceso de verificación tenga complejidad exponencial. Esto es así ya que, tal y como los autores reconocen en [BlEi97], el proceso de verificación formal sobre HOL estándar tiene una complejidad exponencial, por ello para reducirla a una polinomial aceptable, proponen modificar el núcleo del demostrador de teoremas para que trate eficientemente algunos problemas. Si se analiza críticamente esta alternativa, la propuesta viene a ser equivalente al desarrollo de una nueva herramienta específica cuya corrección no queda resuelta en ningún momento y que deja de ser estándar (uno de los argumentos que los autores abanderan para justificar el uso del demostrador).

EJEMPLO 1.2

Para ilustrar el alcance del sistema HASH, sea la siguiente especificación de un comportamiento simple (formalizada utilizando el lenguaje nativo que adopta el sistema):

```
Original ≡
λ( a, b, c ).
   let p = a * b in
      let q = inc(c) in
         let r = p * q in
            let s = b + c in
               let t = p - s in
                  let x = r + t in
                     let y = r * t in
                        ( x, y )
```

Como puede observarse, se utiliza una notación funcional para describir un flujo de datos lineal de manera que, a partir de las dependencias de datos inducidas por la anidación de sentencias *let*, pueda extraerse el orden parcial de ejecución de las operaciones.

Supongamos que se desea verificar el resultado de un proceso de planificación realizado por una herramienta externa. Para ello, el sistema HASH propone que dado que una planificación en k ciclos permite descomponer el conjunto de operaciones de la especificación inicial en k

subconjuntos, el resultado de una planificación puede representarse como una función construída por la composición sucesiva de *k* funciones, cada una de las cuales incluye las operaciones que forman cada uno de dichos subconjuntos. De este modo, si *Original* se planifica en 4 ciclos, la asignación de operaciones a ciclos se formaliza mediante las funciones g_0, g_1, g_2 y g_3, que se agrupan en la siguiente expresión.

```
Planificado ≡
let
   g0 = λ( a, b, c ).
         let s = b + c in
            ( a, b, s, c )
   and g1 = λ( a, b, s, c ).
         let p = a * b in
            let q = inc(c) in
               ( p, q, s )
   and g2 = λ( p, s, q ).
         let r = p * q in
            let t = p - s in
               ( r, t )
   and g3 = λ( r, t ).
         let x = r + t in
            let y = r * t in
               ( x, y )
in g3 ∘ g2 ∘ g1 ∘ g0
```

Para verificar su corrección el sistema HASH, una vez que ha construido las expresiones *Original* y *Planificado*, invoca al demostrador de teoremas HOL para que las normalice automáticamente. Si se alcanza una forma común, la planificación es correcta, en caso contrario incorrecta.

Otro tipo de procesos de síntesis se verifican del mismo modo. Así, para verificar los resultados de una fase de selección (*binding*) de registros se trata de normalizar la siguiente expresión:

```
SeleccionadosRegistros ≡
let
   g0 = λ( a, b, c ).
         let r1' = a
            and r2' = b
            and r3' = b + c
```

```
            and r4' = c
        in ( r1, r2, r3, r4 )
and g1 = λ( r1, r2, r3, r4 ).
        let r1' = r1 * r2
            and r2' = inc(r4)
            and r3' = r3
            and r4' = z1
        in ( r1, r2, r3, r4 )
and g2 = λ( r1, r2, r3, r4 ).
        let r1' = r1 * r2
            and r2' = r1 - r3
            and r3' = z2
            and r4' = z3
        in ( r1, r2, r3, r4 )
and g3 = λ( r1, r2, r3, r4 ).
        let x = r1 + r2 in
            let y = r1 * r2 in
                ( x, y )
in g3 ∘ g2 ∘ g1 ∘ g0
```

Como puede observarse, la selección sólo queda reflejada mediante la aparición de argumentos ficticios en las funciones g_i y mediante la inclusión de definiciones no operativas (de copia de argumentos de entrada a la salida), pero no en base a la introducción de nuevas funciones de memorización que modelen el nuevo comportamiento temporal de la función.

1.1.4 Lambda-Dialog.

El sistema Lambda-Dialog [FoMa89][FiFM91][MaFo91] de Abstract Hardware Limited, ha sido la primera herramienta de síntesis formal desarrollada en el ámbito comercial. Este sistema permite que el diseñador refine interactivamente una especificación en lógica de orden superior mediante la interacción gráfica con un demostrador de teoremas. El interfaz gráfico se llama Dialog, mientras que el demostrador de teoremas se llama Lambda.

Básicamente durante el ciclo de diseño se trata de refinar la tautología: 'la especificación satisface la especificación' en la aserción 'algunos componentes bajo ciertas condiciones de temporización satisfacen la especificación'. Para ello el diseñador debe instanciar interactivamente los componentes preverificados que facilita la herramienta, o partir la especificación en subobjetivos, o aplicar patrones de refinamiento genéricos.

El principal problema que presenta es que incluso para pequeños circuitos, el usuario debe demostrar una gran cantidad de lemas intermedios completamente ajenos al propio proceso de diseño.

1.1.5 Sistemas de síntesis transformacional no formal.

La mayor aportación a la corrección que se realiza desde campos de investigación no relacionados con los métodos formales, es el del desarrollo de sistemas de síntesis transformacional, que reemplazan el diseño clásico basado en un algoritmo monolítico por una sucesiva aplicación de pequeñas transformaciones que presumiblemente conservan el comportamiento.

Sin embargo debe decirse que la mayor parte de estos sistemas, lejos de preocuparse de la corrección (que suponen resuelta mediante la elección de un conjunto simple de transformaciones cuya corrección sea 'evidente'), adoptan las técnicas transformacionales como un método para aumentar la flexibilidad de los sistemas de síntesis. Así, es posible comprobar como muchos autores tras desarrollar una herramienta convencional monolítica, ocasionalmente publican sus experiencias con subproductos transformacionales. De este modo, pueden encontrarse referencias que discuten la integración de algunas técnicas transformacionales en el sistema CAMAD desarrollado en la universidad de Linkoping [PeKu94], en el sistema Cathedral desarrollado en el IMEC [JaCM94][SaCM93][FNS+94], en el sistema SAW desarrollado en la universidad Carnegie-Mellon [WaTh89].

Quizás sea el sistema HYPER, desarrollado en la universidad de Berkeley, el que recientemente más ha apostado por esta metodología alternativa, y pueden encontrarse referencias de síntesis transformativa aplicada a mejorar el reuso hardware [PoRa94], al diseño de baja potencia [CPM+95], al diseño para testabilidad [PoDR95] y al diseño de alto nivel en general [WaPa95]. Las transformaciones típicas que utiliza bajo dirección de algoritmos estocásticos son retemporización, conmutatividad, asociatividad, ley del elemento inverso, desenrollado de bucles, propagación de constantes, segmentación y reducción de las anchuras de las señales.

Por otro lado, también pueden encontrarse sistemas transformacionales que se justifican en base a la fiabilidad que consiguen. Así, el sistema TRADES (Transformational Design System) [MiRa96] es una herramienta desarrollada en la universidad de Twente para la síntesis por transformación de circuitos descritos en VHDL. Si bien propone una rica metodología pseudo-formal, que en un futuro permitiría asistir eficazmente el diseño manual por refinamiento de comportamientos complejos, presenta importantes carencias teóricas que imposibilitan considerarlo como un sistema de síntesis correcta.

La metodología general de diseño comienza con la compilación del comportamiento descrito en VHDL para generar una estructura intermedia descrita en SIL [KMN+92], un lenguaje gráfico con una intuitiva, pero informal, semántica declarativa. A continuación todo el proceso de diseño se reduce a la aplicación interactiva de transformaciones a elegir entre un conjunto de 50 diferentes. Los principales problemas que presenta son tres: en la fase de compilación no se tienen en cuenta ningún aspecto semántico del lenguaje VHDL sino que simplemente se realiza una traducción de las construcciones sintácticas, el lenguaje que se manipula no tiene una semántica formal (sólo un subconjunto de él en base a algebras relacionales [HuKr94]) y no existe una noción de equivalencia formalmente fundamentada. Finalmente, la corrección de las transformaciones se basa en un análisis de la especificación

informal de las mismas pero nunca en una verificación formal de su corrección.

1.2 Objetivos.

El propósito fundamental de la investigación que se recoge en esta memoria es desarrollar un sistema de síntesis formal de alto nivel que pueda ofrecer una alternativa viable a los sistemas convencionales. Mediante el calificativo de viable me refiero a que pueda ser aceptada por una comunidad diseñadora que se muestra recelosa de los métodos formales, por lo que la propuesta deberá, aún siendo completamente rigurosa, ser ante todo intuitiva en sus formas y eficiente en sus implementaciones.

Sin embargo, he deseado que mi investigación sea audaz, por lo que no me he limitado a proponer y desarrollar una alternativa, sino que apuesto convencidamente por ella, no sólo como un método para resolver los problemas de corrección de las herramientas de síntesis automática, sino también como un elemento que active el debate acerca del estado en que se encuentran los sistemas de síntesis, con vistas a plantear soluciones menos rígidas y más fiables que las que en la actualidad se imponen.

Esto será así ya que el entorno desarrollado es tal que, bajo un soporte simbólico uniforme, simple y muy intuitivo, es posible expresar todas las semánticas implicadas en un proceso de diseño de tal manera que pueda razonarse sobre ellas de una manera puramente simbólica. De este modo podrán convivir bajo un mismo techo, una gran cantidad de técnicas diferentes de diseño que podrán realizarse formalmente sin modificar la estructura de la herramienta.

1.3 Organización de esta memoria.

Además del capítulo de introducción que está finalizando, esta memoria se ha estructurado en seis capítulos más que a continuación pasaré a desglosar.

El **capítulo 2** se dedica a la búsqueda de un método de especificación de conductas que posea como características la de ser formal, la de ser intuitivo, la de ser fácilmente manipulable y la de poseer un amplio espectro descriptivo. La primera característica como una exigencia sin la cual no es posible hablar de corrección, la segunda para que pueda ser utilizado por cualquier diseñador con mínimos conocimientos de métodos formales, la tercera para que pueda plantearse un sistema en donde las derivaciones formales sean fácilmente implementables y la cuarta para que, bajo un único formalismo, sean representables no sólo la especificación y la implementación sino también los estados intermedios de un proceso de síntesis por derivación.

Dicho capítulo comenzará estudiando los modelos conductuales que hay que describir y los métodos de especificación informales más habituales entre diseñadores para proponer, en base a ello, un primer método formal de especificación, denominado especificación ecuacional, que será progresivamente completado en los capítulos 4 y 6. De este método se formalizará su sintaxis abstracta y se concretará denotacionalmente su semántica en términos de funciones sobre cadenas infinitas de valores, cadenas que se obtienen mediante un proceso semántico de extensión de álgebras estáticas, que también se define. Por último, el capítulo muestra la simulabilidad del formalismo, propone un algoritmo perezoso para simularlo (una de cuyas posibles implementaciones se muestra en el apéndice D) y presenta un nutrido conjunto de ejemplos prácticos de su uso.

El **capítulo 3** propone un cálculo para la derivación de especificaciones ecuacionales. Este cálculo está compuesto por 11 reglas de transformación que se especifican axiomáticamente y que constituyen el único método permitido de manipulación sintáctica de especificaciones ecuacionales. Su objetivo es facilitar un medio para que cualquier conocimiento que se haya expresado mediante una fórmula universal de primer orden pueda aplicarse convenientemente sobre una especificación ecuacional. Dado que es un método simbólico de transformación, es necesario demostrar su corrección respecto al modelo semántico establecido en el anterior capítulo. Por ello se definen un par de nociones de corrección basadas en la compatibilidad de comportamientos y a continuación se demuestra que cada una de las reglas que forman el cálculo es correcta respecto a dichas nociones. El capítulo se completa con un estudio de complejidad de cada una de las reglas y se muestra cómo reproducir formalmente (o si se prefiere verificar formalmente) un variado número de técnicas de diseño clásico mediante derivación.

El **capítulo 4** se centra en la síntesis de alto nivel: en cómo expresar sus bases ecuacionalmente y en cómo utilizar el conjunto de reglas de transformación definido en el capítulo 3 para realizarla formalmente. De este modo, el capítulo comienza con un estudio de las características especificas que posee un comportamiento que ha sido síntetizado respecto del comportamiento original y propone un conjunto de 4 operadores que se incorporan al mecanismo de especificación presentado en el capítulo 2. A continuación, y utilizando los nuevos operadores, se formalizan ecuacionalmente algunas de las fases de síntesis de alto nivel, y se demuestra la corrección de las mismas. Para finalizar se estudia la complejidad simbólica de dichas ecuaciones y se ilustra cómo utilizarlas para realizar formalmente cada una de las fases de un típico proceso de síntesis de alto nivel.

El **capítulo 5** especifica un sistema formal de síntesis de alto nivel mediante la adecuada aplicación de las reglas propuestas en el capítulo 3. Para ello comienza con la descripción de un primer prototipo capaz de realizar síntesis formal mecanizada (un sistema comparable a cualquiera de los presentados en §1.1), para que en base a una discusión sobre su uso, pueda plantearse el segundo prototipo de síntesis formal automática. Este segundo sistema, que puede ser dirigido por un algoritmo de optimización convencional, no sólo es un sistema de síntesis por derivación sino también un sistema de verificación de la corrección de decisiones de diseño externamente adoptadas. Para finalizar se realiza un estudio teórico de la complejidad del sistema automático, tras el cual se obtiene un conjunto de predicciones que se contrastan experimentalmente sobre una implementación del mismo.

El **capítulo 6** se dedica a realizar extensiones del mecanismo de especificación presentado en el capítulo 2, para que el sistema pueda aceptar no sólo especificaciones del comportamiento, sino también especificaciones de dominios, operadores, representaciones de datos y bibliotecas de componentes. El objetivo que se pretende aumentando el espacio descriptivo del formalismo (y que no requiere una ampliación del sistema de transformación) es aumentar el espacio de soluciones alcanzable por un diseño formal, a la par que ofrecer un método seguro de desarrollo de futuras herramientas que no dependan de técnicas de diseño prefijadas de antemano, sino que puedan evolucionar a la vez que se utilizan. Por ello se incorpora al mecanismo de especificación ecuacional, el mecanismo de especificación algebraica de tipos abstractos de datos y se estudian las posibilidades del mecanismo de implementación algebraica. Asimismo se analiza cómo utilizando éstos nuevos métodos puede especificarse cualquier objeto de diseño presente en cualquier nivel de abstracción de un proceso de síntesis de alto nivel. A continuación se estudia la deducción en teorías ecuacionales y se evalúa la conveniencia de la interacción del sistema de síntesis formal con un demostrador de teoremas y para finalizar se muestran nuevos

ejemplos de reproducción formal de técnicas de diseño clásicas mediante derivación.

Por último el **capítulo 7** reflexiona sobre lo conseguido y sobre lo que todavía queda por conseguir, y la memoria finaliza con cuatro **apéndices** que desarrollan algunos aspectos teóricos (A y B) y algunos prácticos (C y D).

Capítulo 2

Especificación conductual de sistemas

A notation is important
for what it leaves out.
Joseph E. Stoy

La elección de un buen mecanismo de especificación en un entorno de diseño automático es una tarea de crucial importancia que, en muchos casos, es infravalorada en el momento de desarrollar un sistema de síntesis. Su importancia radica en que permite al diseñador tanto definir la relación de entrada-salida que debe verificar un sistema a lo largo del tiempo como comunicar dicha relación a una herramienta. Una elección inadecuada no sólo dificultará esta comunicación, sino que también podrá restringir indirectamente el tipo de conductas especificables y, por tanto, el número de diseños alcanzables.

La tendencia generalizada, al menos en el sector industrial, pasa por adoptar subconjuntos de lenguajes de descripción hardware (HDLs) que continuamente se van ampliando para facilitar la especificación de un número creciente de comportamientos. Esta tónica, si se analiza críticamente, resulta un tanto artificiosa ya que, al estar impulsada por las necesidades puntuales de comunidades concretas de diseñadores, transforma a los lenguajes en

meros grupos disjuntos de construcciones especializadas que se ajustan a problemas particulares, pero que difícilmente pueden generalizarse a otros ámbitos de descripción. Así, por un lado, muchos usuarios terminan siendo incapaces de utilizar eficazmente el número creciente de alternativas expresivas (cuando no de conocerlas) y, por otro, los algoritmos de síntesis, a la hora de tomar decisiones de diseño, son incapaces de obtener un provecho real de todo el potencial expresivo de la especificación.

Este capítulo se dedica a proponer un mecanismo de especificación formal que facilite tanto el trabajo del diseñador como el de la herramienta. Un mecanismo que será utilizado a lo largo de la presente memoria como base del sistema de síntesis formal objeto de la investigación realizada. Los criterios utilizados para su elección han sido los siguientes:

- **Alto poder expresivo**. Para que abarque, a distintos niveles de abstracción, los dominios conductual y estructural de un sistema y todos los estadios intermedios que atraviesa durante su diseño.
- **Alto poder manipulativo**. Para que pueda ser interpretado y directamente transformado (sin necesidad explícita de compilación ni de estructuras intermedias) tanto de un modo manual como de un modo automático.
- **Formal**. Para que posea tanto una sintaxis como una semántica precisa que permita realizar pruebas de corrección y definir sistemas formales de transformación.
- **Compensado**. Para que, conservando todo su rigor matemático, posea una apariencia intuitiva que lo haga fácilmente accesible a diseñadores profanos en métodos formales.

El capítulo comienza (§2.1) con un estudio de los modelos de conducta que pueden ser especificados en un entorno de diseño automático de alto nivel. Continúa (§2.2) con un repaso de los métodos más comunes de especificación de conductas y con una reflexión crítica (§2.3) sobre los problemas que conlleva la implantación de cada uno de ellos. A continuación

(§2.4), se propone el mecanismo de especificación ecuacional. De éste se detalla su sintaxis y su semántica, se presenta su calidad de simulable y se muestran algunos ejemplos prácticos de su uso.

2.1 Modelos conductuales.

Cuando se concreta un modelo se establece un conjunto de objetos a los que referirse. Mediante un modelo se abstraen todos aquellos aspectos de la realidad no relevantes para un cierto problema. En esta sección se presenta un modelo conductual único, que es capaz de expresar cualquier objeto presente en los dominios de interés de un entorno de síntesis de alto nivel (SAN): desde una pura conducta algorítmica hasta un circuito a nivel de transferencia entre registros (RT). Con él se pretende fijar cuál es el conjunto de objetos que cualquier mecanismo de especificación debe denotar, es decir, qué comportamientos son especificables.

2.1.1 Modelo conductual del hardware a nivel RT.

Si queremos describir a nivel RT la evolución de las señales de cualquier circuito síncrono a un único reloj común (tipo de circuito resultado de un proceso de SAN), bastará con detallar los valores que toman las señales en los flancos activos de ese reloj. Esto es así ya que, si el circuito está correctamente diseñado, el período de reloj será mayor que el retardo de cualquier camino combinacional, o lo que es lo mismo, el tiempo transcurrido entre dos flancos de reloj es suficiente para que todas las señales se estabilicen.

Según este argumento una señal puede ser modelada como una secuencia infinita de valores pertenecientes a cierto conjunto. Estos valores están ordenados temporalmente y se corresponden con los valores estables que

adopta la señal en sucesivos flancos. En cuanto al conjunto, será siempre el conjunto de vectores de bits de cierta anchura ya que es posible abstraer cualquier efecto eléctrico, pero si además, se abstrae también la representación, es posible admitir cualquier conjunto discreto. Por todo ello y conociendo que toda sucesión de valores es una aplicación de $\mathbb{N}_+$ sobre cierto conjunto y que el orden en la sucesión denota el instante temporal (medido en pulsos de reloj) en el que se produjo el valor, es posible alcanzar el siguiente modelo RT de señal.

2.1 **DEFINICIÓN.** El **modelo a nivel RT de una señal** es una aplicación ($\mathbb{N}_+ \to A$), donde A es un conjunto discreto que representa el tipo de los datos transportados por la señal y $\mathbb{N}_+$ denota el ciclo en que la señal transporta cierto valor.

Toda señal puede definirse como una función de un parámetro temporal $t \in \mathbb{N}_+$ y expresarse de forma nominal $f(t) = e$, o de la forma anónima clásica de las λ-notaciones† $\lambda t.e$.

Como puede verse en la fig. 2.1, con este modelo no se describe exactamente el comportamiento temporal de la señal pero sí el comportamiento temporal relevante a nivel RT.

Todo dispositivo hardware acepta los valores que transportan las señales de entrada y genera nuevos valores que vuelca sobre las señales de salida. Así, si una señal puede modelarse como una función de variable natural, el comportamiento a nivel RT de cualquier dispositivo hardware con p entradas y q salidas puede ser modelado por una única función que toma como argumento una p-tupla de funciones de variable natural y que genera una q-tupla de funciones de variable natural. Además, conociendo la naturaleza física de todo dispositivo hardware y que éste siempre está compuesto de un número fijo y finito de elementos de memoria, es posible caracterizar un par

† Véase el apéndice A.

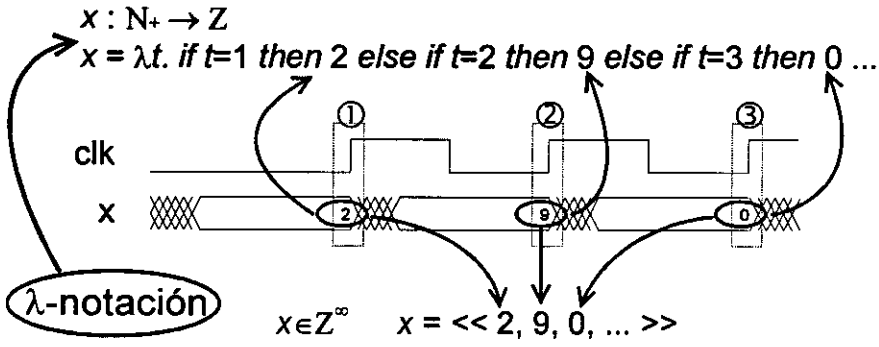


Fig. 2.1: Modelo RT de una señal.

de propiedades que posee toda función modelo de un circuito: causalidad y memoria finita. Es decir, la función deberá ser **causal** ya que todo valor de una señal de salida depende sólo de valores presentes o pasados de las señales de entrada, y deberá tener **memoria finita** ya que los valores que toma una salida sólo dependen de un conjunto finito de clases de equivalencia de los valores de las señales de entrada. Este razonamiento nos lleva a la siguiente definición.

2.2 **Definición.** El **modelo a nivel RT de un dispositivo hardware** con p entradas y q salidas es una función causal y con memoria finita del tipo:

$$(\mathbf{N}_+ \to A_1) \times \overset{p}{\ldots} \times (\mathbf{N}_+ \to A_p) \to (\mathbf{N}_+ \to B_1) \times \overset{q}{\ldots} \times (\mathbf{N}_+ \to B_q)$$

La definición de esta función, que toma y devuelve un vector de señales, puede expresarse de la forma nominal $f(x_1, \overset{p}{\ldots}, x_p) = (e_1, \overset{q}{\ldots}, e_q)$, o de la forma anónima $\lambda(x_1, \overset{p}{\ldots}, x_p).(e_1, \overset{q}{\ldots}, e_q)$.

Nuevamente, con este modelo tampoco se describe el comportamiento temporal de un dispositivo pero sí el comportamiento en situación estable del mismo.

2.1.2 Modelo conductual de los sistemas de tiempo discreto.

Los sistemas de tiempo discreto o sistemas de datos muestreados, son sistemas dinámicos en los cuales las variables pueden variar solamente en ciertos instantes. En los sistemas convencionales, estos instantes son equiespaciados y están fijados por un período de muestreo. El valor que toman las variables dentro de cada uno de los intervalos temporales es constante y puede ser representado por una única cantidad. Por ello, la evolución de las variables en los sistemas de tiempo discreto puede representarse también por secuencias ordenadas de valores. Esto nos lleva a los siguientes modelos de variable y de operador en un sistema muestreado.

2.3 **DEFINICIÓN.** El **modelo de una variable de un sistema muestreado** es una aplicación ($\mathbf{N}_+ \to A$), donde A es un conjunto no necesariamente discreto que representa el tipo de la variable y $\mathbf{N}_+$ denota el ciclo de muestreo en que la variable toma cierto valor.

2.4 **DEFINICIÓN.** El **modelo de un operador de un sistema muestreado** con p entradas y q salidas en un sistema muestreado es una función no necesariamente causal ni con memoria finita, del tipo:

$$(\mathbf{N}_+ \to A_1) \times \overset{p}{\cdots} \times (\mathbf{N}_+ \to A_p) \to (\mathbf{N}_+ \to B_1) \times \overset{q}{\cdots} \times (\mathbf{N}_+ \to B_q)$$

2.1.3 Modelo conductual de un algoritmo en continuo funcionamiento.

Todo algoritmo representa a una función computable que depende de ciertos argumentos de entrada. Si se asignan unos valores concretos a dichas entradas, el algoritmo permite calcular el valor de la función. Si este proceso se repite varias veces, es posible establecer un orden completo entre los valores que sucesivamente toman las entradas y también entre los valores

que sucesivamente calcula el algoritmo. De este modo, es posible representar dicha evolución de valores como secuencias ordenadas cronológicamente, de manera que todos los valores ubicados en una misma posición dentro de la secuencia, se corresponden con un mismo cálculo concreto.

Aunque desde el punto de vista software pueda resultar extraño aceptar el concepto de computación infinita (ya que tiende a asociarse con algoritmos que fallan), desde el punto de vista hardware, no solo es aceptable sino también deseable, ya que todo circuito se diseña para que funcione indefinidamente repitiendo el mismo cálculo una y otra vez y si éste cesa de realizar dicha actividad, se considera que falla. Así llegamos a los siguientes modelos.

2.5 **DEFINICIÓN.** El **modelo de un argumento de un algoritmo en continuo funcionamiento** es una aplicación ($\mathbf{N}_+ \to A$), donde A es un conjunto no necesariamente discreto que representa el tipo del argumento y $\mathbf{N}_+$ denota el orden en que se realizó su uso.

2.6 **DEFINICIÓN.** El **modelo de un algoritmo en continuo funcionamiento** con p argumentos de entrada y q de salida es una función no necesariamente causal ni con memoria finita, del tipo:

$$(\mathbf{N}_+ \to A_1) \times \overset{p}{\ldots} \times (\mathbf{N}_+ \to A_p) \to (\mathbf{N}_+ \to B_1) \times \overset{q}{\ldots} \times (\mathbf{N}_+ \to B_q)$$

2.1.4 Equivalencia de modelos.

Como puede observarse, los modelos que caracterizan a un circuito a nivel RT, a un sistema muestreado y a un algoritmo en continuo funcionamiento son muy parecidos, de hecho pueden ser equivalentes si se realizan algunas matizaciones.

El modelo de una señal y el de una variable de sistema muestreado pueden ser equivalentes si se iguala la frecuencia del reloj del circuito con la

frecuencia de muestreo del sistema y si se restringen los dominios de las variables a conjuntos discretos. El modelo de un dispositivo hardware será equivalente al de un operador, si se restringen las posibles funciones características de los operadores a aquellas que sean causales y con memoria finita.

Las restricciones sobre dominios y funciones no tienen discusión si se desea implementar físicamente el sistema muestreado, en cuanto a la igualación de frecuencias es una alternativa de diseño que lleva usándose durante décadas para realizar implementaciones hardware directas de sistemas muestreados, son las que llamaré **implementaciones monociclo**. Este tipo de implementación no es único y, de hecho, cuando se sintetiza un sistema muestreado mediante técnicas de SAN, la frecuencia de muestreo es un múltiplo de la frecuencia de reloj (en donde el valor del múltiplo es el número de ciclos de la planificación).

Por otro lado, los modelos de un sistema muestreado y de un algoritmo en continuo funcionamiento también pueden hacerse equivalentes (y por transitividad equivalentes a un circuito RT) si se admite que las secuencias que describen los parámetros de un algoritmo no sólo establecen un orden temporal, sino que indican además posiciones equiespaciadas en el tiempo real de ejecución, de modo que el momento de aplicación de valores esté implícitamente marcado por un cierto reloj. Esta asunción obliga a asegurar que todos los algoritmos a modelar tienen un tiempo máximo de cálculo para cualquier dato de entrada (tiempo que permitiría fijar el período del reloj virtual) pero esto es algo que no se puede garantizar de modo general si el cálculo es iterativo (o recursivo), por lo que si aceptamos dicha asunción cabría pensar que es necesario limitar el tipo de funciones especificable. Sin embargo esto no es del todo cierto, un cálculo iterativo puede ser expresado mediante secuencias: basta con repartir cada una de las iteraciones del

mismo en instantes temporales consecutivos y con permitir la realimentación en un cálculo de los valores calculados en ciclos previos.

Así, esta asunción lleva utilizándose en diseño hardware también durante décadas para realizar cálculos iterativos mediante la realimentación de circuitos secuenciales. Estos circuitos, en cada pulso de reloj, realizan una iteración que se repite una y otra vez hasta que el propio circuito detecta que el cálculo ha finalizado.

De esta discusión no sólo es interesante concluir bajo qué condiciones los tres modelos conductuales pueden llegar a ser el mismo, sino también extraer algunos beneficios de la existencia de un modelo común. Gracias a éste, es posible exportar sin problemas metodologías y resultados de un área de aplicación a otra, así en la §2.4 utilizaremos resultados de la teoría de sistemas muestreados para optimizar circuitos RT. Además, si existe un modelo común, es posible definir un único formalismo de especificación para describir cualquier tipo de conducta, lo que permitirá plantear la SAN sin el cambio de nivel de abstracción que habitualmente se acepta que existe entre el nivel RT y el nivel conductual. En ambos niveles los valores que toman las variables podrán ordenarse temporalmente, y por tanto, modelarse del mismo modo: la única diferencia radica en que a nivel RT cambiarán muchas más veces.

2.2 Métodos habituales de especificación conductual a nivel algorítmico.

Los modelos anteriormente descritos definen la relación entrada-salida que debe verificar un sistema a lo largo del tiempo. Cada una de las distintas maneras de describir dichos modelos es un método de especificación. En esta sección se repasarán los mecanismos de especificación más comunes

sin reparar en si se usan en un entorno automático o simplemente como medio en una comunicación entre diseñadores. El objetivo que se persigue no es realizar un estudio detallado de distintos lenguajes particulares, sino más bien clasificar las filosofías de especificación para que, tras estudiar los pros y los contras de su implantación en un entorno automático, permita resolver cuál es la que mejor se adapta a lo que un diseñador necesita y a lo que una herramienta requiere. Una vez elegida la filosofía, no será difícil crear un mecanismo de especificación adecuado.

2.2.1 Especificación temporal.

Dadas las características de los sistemas de datos muestreados en los que los valores sólo cambian en instantes discretos, es común especificar estos sistemas mediante un conjunto de variables que dan nombre a entradas, salidas y señales intermedias y que denotan un conjunto de funciones discretas en el tiempo. El proceso de definición de estas funciones, en donde el orden de las expresiones no influye, utiliza la indexación temporal explícita y puede ser mutuamente recursivo.

Por **mutuamente recursivo** se entiende que en la definición de una función pueden aparecer referencias a otras funciones que a su vez referencian a la función que se está definiendo (la recursividad simple puede considerarse un caso particular de recursividad mutua en donde la función se autoreferencia). Por **indexación temporal explícita** se entiende que toda aparición de una variable que denote a una función debe ser acompañada de una referencia explícita a un instante temporal. Dicho instante puede ser genérico y estar, además, determinado por una expresión que utilice índices temporales (variables especiales que sólo pueden tomar valores enteros).

Así, para definir una función x, se declara el valor que toma en un instante genérico t en función de sus valores pasados y de los valores pasados de

cualquier otra variable del sistema, y todo ello se completa con la definición de los valores iniciales, o sea, los que toma en instantes 'negativos'.

La fig. 2.2 muestra una posible especificación temporal de un filtro recursivo de segundo orden. Como puede observarse, en el aspecto sintáctico, se utiliza un sistema de ecuaciones en las que en el lado izquierdo aparecen las variables indexadas a definir y en el lado derecho aparecen expresiones formadas por símbolos de operación, símbolos constantes y variables indexadas.

En el aspecto semántico, los símbolos de constante denotan funciones cuyo valor es constante en el tiempo y los símbolos de operación denotan operadores que transforman funciones. La solución del sistema de ecuaciones (en el más estricto sentido matemático) suele ser la semántica de la especificación dada.

Este mecanismo de especificación, tal y como ha sido presentado, no es aceptado como entrada por ningún sistema de síntesis automática conocido. Sin embargo, es uno de los métodos más naturales de especificación de sistemas muestreados del que pueden encontrarse infinidad de referencias [SyMa93][PaMa93][ShPa93] que lo utilizan como método de comunicación

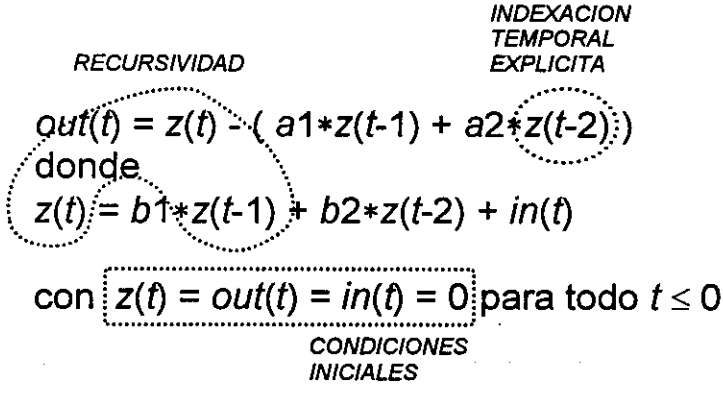


Fig. 2.2: Especificación temporal.

entre diseñadores. No obstante, pueden encontrarse en la literatura lenguajes que tienen un cierto 'sabor' de especificación temporal. Por ejemplo *Silage* [Hilf85] utilizado extensivamente por la saga de herramientas *Cathedral* [DRSC86][NGCD91] desarrolladas en el *IMEC* o por el sistema *Piramid* [WBMN90] de *Philips*, *Schema* [John84] utilizado por el sistema de síntesis formal *DDD* [JoBo91] de la Universidad de Indiana o, desde un punto de vista más teórico, *Stream* [Delg87][Broy94] que aún no ha tenido una aplicación directa en el campo del diseño automático.

2.2.2 Especificación estructural.

Otro tipo de especificación habitual es el que utiliza diagramas de bloques para especificar la estructura del sistema. En ellos se describe básicamente un grafo de flujo de datos basado en el modelo de **flujo de un único token**. En este modelo todo nodo sólo puede modelar el comportamiento del sistema durante un único ciclo de muestreo, quedando implícito el comportamiento del sistema a lo largo del tiempo. Así en lugar de definir el sistema en cada ciclo individual, la misma definición genérica se reutiliza para cada uno de los ciclos describiendo, por tanto, las transformaciones invariantes en el tiempo que se aplican a las cadenas de tokens que llegan a las entradas. La fig. 2.3 muestra una especificación estructural del filtro especificado temporalmente en la fig. 2.2.

En toda especificación estructural existen, al menos, dos tipos de nodos primitivos: el operativo o combinacional en donde el token producido sólo depende de los tokens consumidos y denota, por tanto, funciones con un comportamiento que no depende del instante temporal en que se aplique, y el de retardo (representado por z^{-1} o por δ) que almacena la información de estado y que se usa para modelar el flujo de datos entre ciclos de muestreo.

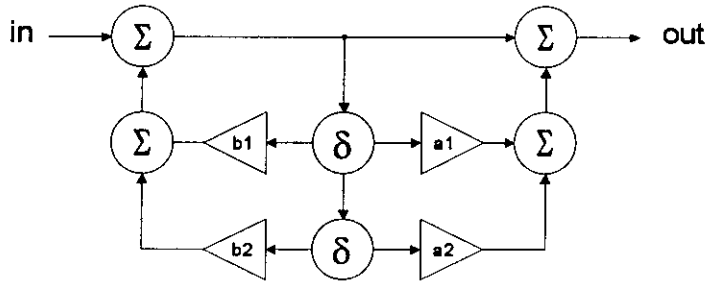


Fig. 2.3: Especificación estructural.

Las redes de nodos suelen agruparse para definir nodos jerárquicos más complejos y que pueden contener lazos de realimentación siempre y cuando en todo lazo exista un nodo de retardo.

Las especificaciones estructurales son un caso particular y simple de un conjunto más amplio de redes, las redes puras de flujo encauzado de datos. En un **flujo encauzado de datos** (establecido por primera vez por Conway [Conw63]) los datos son consumidos por los nodos en el orden en que se producen y por tanto no necesitan etiquetarse con información temporal. En un **flujo encauzado de datos puro**, además, la historia completa de las salidas está completamente determinada por la historia completa de las entradas. Es claro que la existencia implícita de un ciclo de muestreo global hace que una especificación estructural sea encauzada y que la simplicidad de la función que realizan los nodos primitivos hace que también sea pura.

El método de especificación estructural es, al igual que el de especificación temporal, un método de comunicación de uso generalizado entre la comunidad de diseñadores pero, a diferencia de éste, si que ha sido utilizado en el ámbito del diseño automático. Esto es así ya que algunos sistemas de síntesis aceptan como mecanismo de especificación grafos basados en el modelo de token único. Entre ellos podemos destacar el sistema HAL [PaKG86] desarrollado en la Universidad de Carleton y el sistema DAGAR de

la universidad de Texas en Austin. Además también pueden encontrase referencias de lenguajes puramente estructurales, tal como lo es SIL [KMN+92] utilizado por el sistema de síntesis semiautomática TRADES [MiRa96] desarrollado en la universidad de Twente.

2.2.3 Especificación procedural.

Una especificación procedural es una colección de sentencias (en cierto lenguaje imperativo) que describen como realizar secuencialmente una computación dada. Para ello, aparte de las típicas construcciones de control, se utilizan un conjunto de variables para almacenar temporalmente valores intermedios utilizados tanto en presentes como en futuras iniciaciones del algoritmo. Cualquier variable que almacene valores que vayan a ser utilizados en una futura iniciación se marca para que su valor permanezca estático. Esta marca será posteriormente utilizada por la herramienta de síntesis para prevenir las posibles antidependencias creadas por la asignación de dicha variable a un registro.

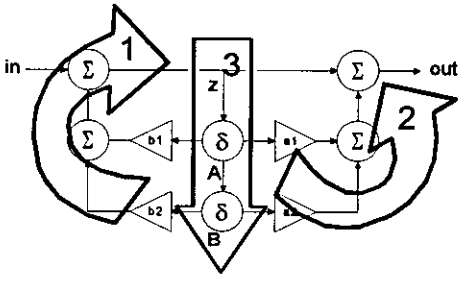
En este tipo de especificaciones, el orden de las sentencias es crítico y el comportamiento especificado no sólo es distinto si las sentencias lo son, sino que también varía según el orden que éstas adopten. Así, cuando se parte de otros tipos de especificación no procedurales (tales como la temporal o la estructural) es necesario realizar una fase previa de serialización.

La **serialización** o generación de una secuencia de operaciones que describa una computación, involucra, en general, la obtención de una especificación estructural sin jerarquía en donde todos los nodos sean primitivos (combinacionales o retardos). Entonces, se deben ordenar las operaciones de tal manera que los argumentos de cada operador hayan sido calculados por una sentencia anterior o se lean de una variable marcada (o sea, hayan sido calculados en una anterior iteración del algoritmo).

Finalmente, cada una de las variables marcadas deben ser actualizadas en un orden concreto fijado por la topología global de la especificación. Debe destacarse que partir de una especificación no jerárquica es esencial, ya que la composición secuencial de bloques serializados no es funcionalmente equivalente a la serialización global de una especificación sin jerarquía.

Un análisis del comportamiento del filtro de segundo orden mostrado en las anteriores figuras revela cuál es la correcta secuencia de operaciones que lo describe: la expresada mediante flechas en la fig. 2.4-a. La fig. 2.4-b, por su parte, muestra el correspondiente código procedural (VHDL). En dicho código se han enmarcado los bloques de sentencias que se corresponden con la computación fijada por las flechas.

Sin duda alguna, los lenguajes procedurales son los más populares en el ámbito del diseño automático. De hecho, la mayor parte de los sistemas de SAN adoptan subconjuntos procedurales de los lenguajes de descripción hardware más extendidos como *VHDL* [IEEE87], *Verilog* [Veri91] o *Hardware-C* [KuMi90].



a

```
ENTITY SecondOrder IS
   PORT( inport : IN INTEGER; outport : OUT INTEGER );
END SecondOrder;

ARCHITECTURE Spec OF SecondOrder IS
BEGIN
   PROCESS
      CONSTANT a1, a2, b1, b2 :INTEGER := ...;
      VARIABLE t1, t2, t3, t4, t5, t6, z, A, B :INTEGER;
      ATTRIBUTE IsArchRegister OF A :VARIABLE IS TRUE;
      ATTRIBUTE IsArchRegister OF B :VARIABLE IS TRUE;
   BEGIN
      t6 := b2*B;
      t5 := b1*A;
      t4 := t5 + t6;                    1
      z := inport + t4;
      t3 := a2*B;
      t2 := a1*A;
      t1 := t2 + t3;                    2
      outport <= z - t1;
      B := A;
      A := z;                           3
   END PROCESS;
END Spec;
```

b

Fig. 2.4: Especificación procedural.

2.2.4 Especificación heterogénea.

Para finalizar, cabe reseñar otro estilo de especificación que puede encontrarse: el enfoque mixto. En éste la estructura gruesa del sistema se especifica utilizando diagramas de bloques y cada bloque, dependiendo de su complejidad, se especifica usando nuevamente bloques o mediante algún otro mecanismo de especificación (temporal o procedural).

El enfoque mixto que mezcla estructura y ecuaciones temporales suele utilizarse en la comunicación hombre-hombre. Para la comunicación hombre-máquina pueden encontrarse algunos lenguajes que mezclan estructura y bloques procedurales tales como *SpecCharts* [NaVG91] o *StateCharts* [Hare87].

2.3 Discusión sobre la adecuación en síntesis de los métodos habituales de especificación conductual.

Si se desea implantar un mecanismo de especificación para que sea procesado por una herramienta automática es necesario, como apuntara Boute [Bout87], que satisfaga tanto a teóricos como a prácticos, siendo tan elegante como útil y tan riguroso como intuitivo. Básicamente se proponen dos criterios para calificarlo: poder expresivo y poder manipulativo. Entendiendo por lo primero la capacidad del formalismo de expresar los sistemas que pretende describir y entendiendo por lo segundo la facilidad del mismo para ser interpretado y transformado tanto automáticamente como manualmente.

2.3.1 Sobre la implantación de la especificación temporal.

Si bien este mecanismo es claramente intuitivo (de lo contrario no sería utilizado por los diseñadores para comunicarse entre ellos) posee algunas características que lo hacen de difícil implantación.

Para empezar, las variables necesitan ser indexadas en ambos lados de una ecuación, lo que hace que el mecanismo (aún siendo declarativo) no tenga transparencia referencial, o sea, que un identificador de variable (e indirectamente los propios operadores) no siempre representa la misma cosa ya que su denotación depende de su índice, y si ese índice es una expresión todo se complica aún más. La falta de transparencia referencial reduce, en general, la manipulabilidad de los formalismos.

EJEMPLO 2.1

Como ilustración de la anterior observación considérense independientemente las siguientes definiciones:

$$z(t) = x(t) + y(t)$$
$$z(t) = x(t) + y(t\text{-}1)$$
$$z(t\text{+}1) = x(t) + y(t)$$
$$z(t\text{+}1) = x(t) + y(t\text{-}1)$$
$$z(t) = x(t) + y(t\text{+}1)$$

Son notacionalmente muy parecidas, pero describen sistemas bien diferentes. Mientras la primera describe un sistema combinacional que suma, la segunda describe un sistema secuencial que suma secuencias desfasadas, la tercera un sistema secuencial que retrasa una suma combinacional, la cuarta un sistema secuencial que retrasa la suma de secuencias desfasadas y la quinta un sistema no causal. El número de elementos de memoria requeridos es también diferente 0, 1, 1, 2 respectivamente para los cuatro primeros, mientras que el último no es implementable. Como puede observarse dependiendo del índice de las variables tanto éstas como el

operador + tienen un significado distinto, lo que hace este mecanismo potencialmente difícil de entender y manipular.

La especificación con subíndices tiende a oscurecer el hecho de que una variable denota la historia de una única actividad dinámica. Así, cuando se indexa una variable se tiende a interpretarla como un objeto individual en lugar de como una parte de la historia del verdadero objeto individual que es la historia completa. Es decir, se tiende a asimilar variables con arrays y a considerar que el conjunto de todas las variables indexadas componen una historia, en lugar de que es una historia la que puede descomponerse en un conjunto de sucesos individuales. Esto, en general, puede dar lugar a que el diseñador realice razonamientos incorrectos.

EJEMPLO 2.2

Sea la secuencia a = « 1, 2, 3, 4, 5 ... », definida por la expresión:

$$a(\ t\ ) = t$$

Si se descompone en dos definiciones parciales de términos pares e impares se obtiene:

$$a(\ 2*t\ ) = 2*t$$
$$a(\ 2*t\text{-}1\ ) = 2*t\text{-}1$$

De la segunda ecuación se puede deducir que:

$$a(\ 2*t\text{-}1\ ) = 2*t\text{-}1 = a(\ 2*t\ )\text{-}1 \qquad \textit{primera deducción}$$
$$a(\ 2*t\text{-}1\ ) + 1 = 2*t\text{-}1+1 = 2*t = a(\ 2*t\ ) \qquad \textit{segunda deducción}$$

Luego según ese razonamiento cabría escribir:

$$a(\ 2*t\ ) = a(\ 2*t\text{-}1\ )+1 \qquad \textit{según la segunda deducción}$$
$$a(\ 2*t\text{-}1\ ) = a(\ 2*t\ )\text{-}1 \qquad \textit{según la primera deducción}$$

Pero, si se estudia, se comprobará que este par de ecuaciones lejos de ser equivalente a la primera, dejan completamente indefinido el valor de a, esto

es, a = « ?, ?, ?, ?, ? ... ». Así la semántica del sistema es ambigua, ya que existen infinitas secuencias que son solución del mismo.

Además si un sistema de ecuaciones involucra un elevado número de variables y de índices y en éstos incluyen expresiones aritméticas arbitrarias (que son necesarias para poder hacer descripciones finitas de secuencias infinitas) será difícil interpretar su significado, sin hablar de su posible manipulación para fines de diseño.

EJEMPLO 2.3

Intente el lector averiguar qué secuencia de valores queda descrita por este sistema en donde sólo se ha definido una variable, utilizado un único índice temporal y los operadores + y -.

$$a(2*t) = \textit{if } t{=}1 \textit{ then } 2 \textit{ else } a(2*t\text{-}1) + a(2*t\text{-}1) + a(t{+}1) - a(t)$$
$$a(2*t\text{-}1) = \textit{if } t{=}1 \textit{ then } 1 \textit{ else } a(2*t\text{-}2) + 1$$

El principal problema del mecanismo de especificación temporal es que la indexación temporal explícita es excesiva para el propósito para el cual se necesita, es decir, que permite al diseñador expresar más cosas de las necesarias aumentando el riesgo de especificar funciones ambiguas e incrementando las posibilidades de cometer errores. La indexación temporal explícita es a otros esquemas más abstractos de especificación como el lenguaje máquina es a los lenguajes de alto nivel.

No obstante, como conclusión debe extraerse que si fuera posible eliminar de algún modo la indexación temporal explícita (substituyéndola por algún operador de mayor nivel de abstracción) este mecanismo sería, desde el punto de vista del diseñador, bastante adecuado.

2.3.2 Sobre la implantación de la especificación estructural.

Existen dos principales inconvenientes que hacen que este mecanismo fuertemente intuitivo no resulte excesivamente atrayente para su implantación. Primero, obliga a facilitar una información topológica que, a este nivel de abstracción, no es relevante para la síntesis (la información topológica sólo es útil en fases de mucho más bajo nivel) por lo que se malgasta el tiempo dedicado a maquetar los diagramas (proceso típico en las herramientas de captura de esquemas). Y segundo, la semántica completa del sistema no queda fijada únicamente por la estructura, siempre es necesario definir con un mecanismo de especificación alternativo el comportamiento de los nodos primitivos.

Pero la verdadera razón que permite desestimarlo, al menos como mecanismo de especificación primario, es que la especificación estructural no aporta más capacidad expresiva que mecanismo temporal. Esto se concluye a partir de un resultado establecido por Kahn [Kahn74] y demostrado por Faustini [Faus82] que determina que el comportamiento de una red pura de flujo encauzado de datos (una especificación estructural) está exactamente descrito por la función solución del sistema de ecuaciones asociado a la red.

La correspondencia entre la red y el sistema de ecuaciones se establece de la siguiente manera: cada nodo de procesado se corresponde con un operador, cada una de sus salidas con una variable y cada una de sus entradas libres (que no estén conectadas a ningún otro nodo y por consiguiente son entradas del sistema) se corresponde también con una variable. Así cada nodo se corresponde con una ecuación en donde en el lado izquierdo aparecen las variables que define y en el lado derecho una expresión simple formada por la aplicación de un operador a ciertas variables que usa como argumentos. Un sistema de *n* nodos se corresponderá, por tanto, con un sistema de *n* ecuaciones. Los arcos tienen también una relación

directa con las variables, ya que su labor es conectar la salida de un nodo con las entradas de uno o varios nodos y esa labor ahora se realiza por múltiples apariciones de la misma variable en los lados derechos de las ecuaciones. Los índices concretos de las variables dependerán de la funcionalidad de los nodos primitivos.

2.3.3 Sobre la implantación de la especificación procedural.

En el campo del desarrollo software existe abundante literatura que desaconseja el uso de lenguajes procedurales para muchos propósitos. Pero también es cierto que existe otra tanta que continuamente tacha a los lenguajes no-procedurales (sin asignaciones ni flujo de control) de ineficientes para cualquier aplicación práctica. Sin embargo, tal polémica no surge cuando se habla de lenguajes de especificación ya que hay un cierto consenso en aceptar que un buen lenguaje de especificación debe poseer características declarativas no-procedurales. La razón es que una descripción procedural, en lugar de una especificación, lo que expresa es una implementación secuencial particular que resulta sobre-específica frente a una pura especificación de la relación entrada-salida.

Esta postura contrasta con la adoptada en el campo del diseño hardware donde se insiste en adoptar HDLs (en su mayor parte procedurales) como formalismos válidos de especificación. Incluso el estándar de facto, el VHDL, que posee características tanto procedurales como no-procedurales, se restringe a un subconjunto procedural cuando debe ser aceptado por un sistema de síntesis conductual.

Por ello, últimamente algunos autores [MiLD92][RoBu95] han alertado de los problemas que pueden derivarse del uso de los HDLs procedurales en aplicaciones de síntesis. Problemas que actualmente vienen sufriendo los

diseñadores de herramientas cuando tratan de ampliar el alcance de las mismas en aspectos tales como: reuso, corrección, interoperabilidad, etc.

El primero de los problemas viene por herencia de los lenguajes de programación. La mayor parte de los HDLs son evoluciones de lenguajes software, por lo que han sido diseñados para ser ejecutados, es decir, interpretados por un simulador. Esto impone que los resultados intermedios de la síntesis, que no tienen por qué ser ejecutables, no puedan ser expresados e impone también que las especificaciones sólo posean una semántica de simulación fija (que no formal) para determinar cuando dos 'programas' modelan el mismo circuito: cuando ambos responden igual al mismo conjunto de estímulos.

El problema está en que dicha semántica de simulación es completamente inútil para un diseñador o una herramienta, ya que éste no considera las formas de onda como especificación del sistema sino que acepta la descripción textual e intenta interpretarla. Sin embargo, la poca relación que existe entre la mayor parte de las construcciones del lenguaje y el hardware real, además de la infinidad de 'programas' que pueden simular el mismo comportamiento, hacen que la interpretación sea imposible.

La solución que se adopta es restringir el modo descriptivo de manera que sólo se permitan aquellas construcciones que mejor se correspondan con la arquitectura objetivo. Así, junto con esta solución nacen multitud de dialectos adaptados para cada propósito y para cada herramienta particular, que hacen que un lenguaje estándar tenga una semántica de síntesis, e incluso un subconjunto de sintaxis, parcial y arbitrario. En muchos casos estos dialectos llegan a ser tan estrechos, que el proceso de especificación llega a reducirse a una mera captura y rellenado de formularios.

Por ello, es posible encontrar una cantidad enorme de publicaciones describiendo trucos y reglas de estilo para especificar circuitos mediante este tipo de lenguajes [DeOd93][Meye89][CaST91][MHM+94] y, de hecho, no

existe aún un subconjunto consensuado de VHDL para la especificación conductual a nivel algorítmico. A más bajos niveles se está consiguiendo [ViES95], pero muchos años después de la aparición del lenguaje.

El segundo problema viene originado por la carencia de semántica formal que poseen los HDLs (aunque existan serios esfuerzos en definirla para alguno de ellos [DeBr95]). Esto hace que sea imposible establecer cuando dos descripciones modelan a un mismo circuito. A lo sumo, si se exceptúan las simulaciones exhaustivas (inviables para descripciones de tamaño mediano), es posible tener sospechas razonables de equivalencia si la batería de estímulos es representativa, pero jamás certeza.

Además esta carencia de semántica formal hace que un diseño transformacional (o simplemente diseño correcto) basado directamente en un HDL sea imposible. Esa es la razón por la cual siempre es necesario un formato intermedio más fácilmente manipulable que contenga sólo la información relevante de la especificación. Así todas las herramientas basadas en HDLs, necesitan de compiladores que construyan la representación interna del HDL, y de generadores de código que, a partir de la representación interna, obtengan descripciones usando HDLs.

El tercer problema se origina por la poca relación que suelen tener los HDLs con el ámbito del problema y, en general, con los modos naturales de especificación de alto nivel. Este problema obliga a que el diseñador adapte la especificación original al HDL usado y que deba asegurarse que en la adaptación no se perdió ni ganó información (tal como se ha descrito en el proceso de serialización de §2.2.3). Sin embargo, como a continuación se mostrará, algunas de esas adaptaciones son innecesarias en un entorno de SAN, ya que la herramienta las deshace para obtener una representación interna que termina siendo similar a la especificación original.

Una herramienta convencional de SAN toma el código procedural mostrado en la fig. 2.4-b y lo compila para obtener una representación interna en forma

de grafo de flujo de datos (DFG). Este grafo, revelará tanto las ligaduras de secuenciamiento reales que se derivan de las dependencias de datos, como el paralelismo potencial del algoritmo (véase fig. 2.5-a). Por otro lado si se observa el grafo de la fig. 2.5-b, que se ha obtenido redibujando la especificación estructural de la fig. 2.3, podrá comprobarse que ambos grafos son esencialmente equivalentes.

De todo ello se pueden obtener dos conclusiones. Primero, la secuencialización de una especificación, que permite escribir código procedural, es innecesaria, dado que el trabajo realizado por el diseñador es inmediatamente deshecho por la herramienta para generar el grafo de flujo de datos original. Y segundo, no hay razón por la cual una herramienta no pueda adaptar su entrada al modo natural de especificación (temporal o estructural) ya que, al igual que la procedural, contienen toda la información necesaria.

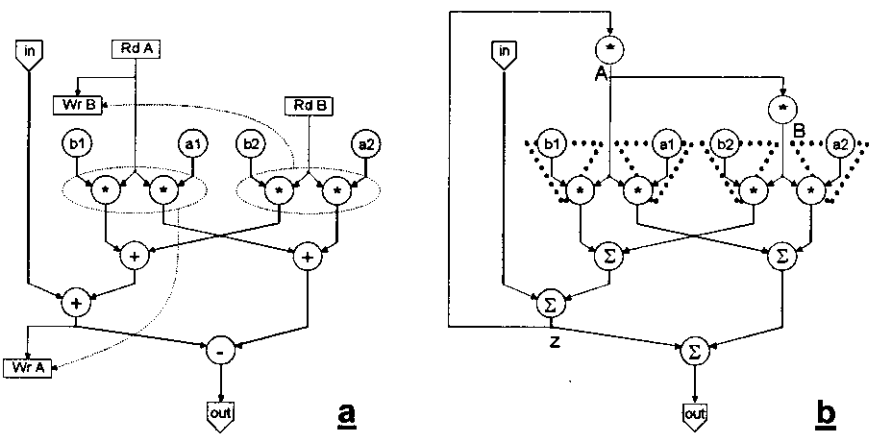


Fig. 2.5: DFG vs. especificación estructural.

2.4 Especificación ecuacional de sistemas.

En esta sección se presenta un método de especificación inspirado en la especificación temporal† (que se demostró equivalente a la estructural) pero que posee un modo de abstraer la indexación temporal lo que lo hace, por tanto, referencialmente transparente. Así este mecanismo continuará siendo intuitivo pero será mucho más manipulable.

En el aspecto sintáctico, el mecanismo propuesto estará compuesto de un conjunto de variables que dan nombre a entradas, salidas y señales intermedias y que denotan un conjunto de funciones discretas en el tiempo. Pero ahora en el proceso de definición de estas funciones, que seguirá siendo mutuamente recursivo y utilizará operadores comunes, no existirán índices temporales sino que se utilizarán operadores abstractos que transforman el tiempo.

En el aspecto semántico, se le dotará de una semántica formal que permita resolver ambigüedades y demostrar equivalencias de descripciones. En posteriores capítulos, gracias a esta semántica formal, podrá desarrollarse un sistema de transformación puramente sintáctico que realice síntesis de alto nivel correcta.

El desarrollo de esta sección será como sigue. Inicialmente (§2.4.1) se asumirá la existencia de una colección suficiente de dominios estáticos y se presentará un esquema que permitirá asociarles un soporte sintáctico. Dicho soporte sintáctico será utilizado para la construcción de expresiones que denoten funciones invariantes en el tiempo (combinacionales). La especificación explícita de dominios se retrasará hasta el capítulo 6. Tras esto, se realizarán tres extensiones sucesivas de los anteriores dominios y de

† Y sobre todo en el lenguaje data-flow LUCID [WaAs85] que es el que en realidad me inspiró a estudiar otros métodos de especificación distintos del procedural.

su soporte sintáctico (§2.4.2). Las extensiones pretenden aumentar el poder expresivo de los dominios añadiendo nuevos elementos y extendiendo las operaciones primitivas: una primera para soportar indiferencia, que permitirá que el diseñador aumente el grado de libertad de decisión de la herramienta (algo parecido a los *don't-care* de nivel lógico pero a más alto nivel), una segunda para soportar indefinición, que permitirá detectar descripciones que se corresponden a circuitos inestables y una tercera que permitirá definir secuencias infinitas de valores. Seguidamente (§2.4.3) se abordará la definición del único operador temporal básico necesario para la especificación: el operador *fby* (a este operador se unirán otros cuatro en el capítulo 4, que permitirán realizar síntesis transformacional de alto nivel). Además se presentará el concepto de especificación ecuacional. Para finalizar en §2.4.4 y §2.4.5 se demostrará la calidad de simulable del mecanismo propuesto y se presentarán algunos aspectos de su utilización.

2.4.1 Algebra estática.

Dado que las principales características del tipo de especificación que se va a proponer son polimórficas, es decir, independientes de cualquier consideración en cuanto a tipos y operaciones, se va a utilizar una notación que en sí no exige ni tiene incorporados unos tipos de datos primitivos o particulares, sino que puede ser utilizada con cualesquiera de ellos. Podría decirse que la notación que será usada define una familia de notaciones bien formada, en donde cada miembro está determinado por el conjunto de estructuras de datos sobre las que trabaja. Tal y como propuso Landin [Land66] se separará 'la manera de expresar objetos en términos de otros objetos' de 'el conjunto básico de objetos dado'.

Las estructuras de datos particulares del problema, que son una colección de conjuntos de elementos y de operaciones sobre esos conjuntos, serán

modeladas mediante un álgebra particular. Y será esta álgebra dependiente del problema, la que determine en cada momento la notación concreta a utilizar.

No obstante, a partir de ahora, se supondrá que existen ciertas álgebras elementales. Por ello se asumirá la existencia de conjuntos tales como el que forman los números naturales o los números enteros o los booleanos, y se asumirá también la existencia de un conjunto suficiente de operaciones aritméticas y relacionales. El problema de cómo especificar dominios arbitrarios y de cómo especificar el comportamiento de funciones particulares será tratado en detalle en el capítulo 6, en donde, para conservar el enfoque no se presentará una nueva teoría sino que se complementará la que en este capítulo se presenta parcialmente.

Por el momento, si se asume la existencia de dichas álgebras, el objetivo es darles un soporte sintáctico coherente que permita el razonamiento simbólico sobre ellas. Para dar un soporte sintáctico simultáneo a un conjunto de álgebras utilizo el concepto de signatura heterogénea, que es una colección de identificadores de clases de datos (llamados géneros) y una colección de identificadores de operación.

2.7 **DEFINICIÓN.** Una **signatura heterogénea** es el par (S, Σ), donde:

- S es el conjunto de **géneros** de la signatura.
- Σ es una familia $S^* \times S$-indexada de conjuntos $\{ \Sigma_{w,s} \mid w \in S^* \wedge s \in S \}$[†] tal que los elementos de $\Sigma_{w,s}$ son **símbolos de operación**.

Cuando el conjunto de géneros está claro, se escribirá únicamente Σ para indicar la signatura completa (S, Σ).

Todo símbolo de operación $\sigma \in \Sigma_{w,s}$ se dice que tiene **perfil** (w, s), **aridad** w y **género** s. La aridad expresa el género de cada uno de los argumentos y

---

[†] S^* es el conjunto de cadenas finitas de géneros, incluyendo la cadena vacía ε.

el orden relativo que adoptan. El género de un símbolo de operación expresa el género del resultado. Un símbolo de operación se llama **constante** cuando la aridad de la operación es nula, es decir, la cadena de géneros w es la cadena vacía ε.

Obsérvese que la calidad de $S^* \times S$-indexado permite agrupar a los símbolos de operación por los géneros del perfil, lo que permite usar el mismo nombre de operación para nombrar símbolos presentes en distintas declaraciones (cada uno con distinto número o género de argumentos), es lo que se conoce como **sobrecarga de símbolos de operación**.

En los ejemplos, y por conveniencia se utilizará el siguiente esquema sintáctico: los géneros se listarán tras la palabra reservada **sorts** y los símbolos de operación tras la palabra reservada **operations**. En la definición de estos últimos se usará la notación $\sigma : w \rightarrow s$, donde σ, aparte del nombre de la operación, podrá indicar mediante $\square$ la posición relativa de los argumentos respecto a dicho nombre.

EJEMPLO 2.4

Demos soporte sintáctico aceptable al conjunto de los booleanos, B, al de los números enteros, Z, y a un subconjunto de sus operaciones mediante una signatura heterogénea.

sorts
Bool, Ent
operations

cierto	:	$\rightarrow$ Bool	*constante (aridad nula)*
falso	:	$\rightarrow$ Bool	
no	:	Bool $\rightarrow$ Bool	
$\square + \square$	:	Ent, Ent $\rightarrow$ Ent	*símbolo de operación infijo*
$\square - \square$	:	Ent, Ent $\rightarrow$ Ent	
-	:	Ent $\rightarrow$ Ent	*símbolo de operación sobrecargado*
$\square > \square$	:	Ent, Ent $\rightarrow$ Bool	

Como puede observarse hemos declarado un género por cada conjunto y siete símbolos de operación. Los dos primeros son constantes y el resto son

operaciones convencionales. Obsérvese que en algunos de los símbolos de operación se ha utilizado el símbolo □ para denotar la posición infija del identificador. Nótese como existe un identificador sobrecargado que aparece con distinto perfil en dos declaraciones distintas.

Más formalmente, si (S, Σ) es la anterior signatura, S y Σ han sido definidos como:

- $S \equiv \{$ Bool, Int $\}$
- $\Sigma \equiv \{ \Sigma_{\varepsilon,\ Bool}, \Sigma_{Bool,\ Bool}, \Sigma_{Ent,\ Ent}, \Sigma_{Ent.Ent,\ Ent}, \Sigma_{Ent.Ent,\ Bool} \}$ donde
  $\Sigma_{\varepsilon,\ Bool} \equiv \{$ cierto, falso $\}$
  $\Sigma_{Bool,\ Bool} \equiv \{$ no $\}$
  $\Sigma_{Ent,\ Ent} \equiv \{$ - $\}$
  $\Sigma_{Ent.Ent,\ Ent} \equiv \{$ +, - $\}$
  $\Sigma_{Ent.Ent,\ Bool} \equiv \{$ > $\}$

Toda signatura permite definir un conjunto de expresiones sintácticamente correctas, llamadas términos, que pueden ser formadas a partir de la asociación de constantes y símbolos de operación. Si además definimos un conjunto de variables (conjunto de símbolos disjunto con el de operaciones) este número de expresiones válidas se multiplica.

2.8 **Definición.** Sea una signatura heterogénea (S, Σ) y una familia S-indexada de conjuntos de variables, $X = \{ X_s \mid s \in S \}$, disjuntos dos a dos y respecto a Σ. Se define el **conjunto de términos** como la familia S-indexada $T_\Sigma(X) = \{ T_{\Sigma,s}(X) \mid s \in S \}$, donde cada $T_{\Sigma,s}(X)$ (**conjunto de términos de género s**) se define inductivamente como sigue:

- $X_s \cup \Sigma_{\varepsilon,s} \subset T_{\Sigma,s}(X)$ *término simple*
- Si $\sigma \in \Sigma_{w,s}$ con $w = s1 \ldots sn$ y $t_i \in T_{\Sigma,si}(X)$ con $i \in \{1,\ldots,n\}$ entonces $\sigma(t_1, \ldots, t_n) \in T_{\Sigma,s}(X)$ *término compuesto*

Todo término sin variables se llama **término cerrado**. Cuando en la anterior definición $X \equiv \varnothing$, se habla del **conjunto de términos cerrados** y se escribe T_Σ. Obsérvese nuevamente como la calidad de *S*-indexado permite clasificar por géneros tanto variables como términos.

En los ejemplos se usará indistintamente la notación prefija anteriormente definida u otra más conveniente para aquellos símbolos de operación en cuya signatura se haya explicitado una posición concreta del identificador respecto de los argumentos. Dado que no se pueden concretar las asociatividades de los símbolos, todo término que use una notación distinta de la prefija aparecerá entre paréntesis.

EJEMPLO 2.5

Para la signatura del ejemplo 2.4 y, definiendo el conjunto de variables X como $\{ X_{Bool} \equiv \{ b \}, X_{Ent} \equiv \{ x, y \} \}$, las siguientes expresiones son términos:

x	*término simple de género Ent*
cierto	*término cerrado simple de género Bool*
(x + y)	*término compuesto de género Ent en notación infija*
+(x, y)	*término compuesto de género Ent en notación prefija*
no(no(falso))	*término cerrado compuesto de género Bool*
no (-(x) > (x + y))	*término compuesto de género Bool en notación mixta*

Clasifiquemos más formalmente los términos, nótese que en esta clasificación sólo utilizamos la notación infija precisada por la definición 2.8:

- $\{ x, +(x, y) \} \subset T_{\Sigma,Ent}(X) \subset T_\Sigma(X)$
- $\{ \text{cierto}, \text{no}(\text{no}(\text{falso})) \} \subset T_{\Sigma,Bool} \subset T_{\Sigma,Bool}(X) \subset T_\Sigma(X)$
- $\{ \text{no}(>(-(x), +(x, y))) \} \subset T_{\Sigma,Bool}(X) \subset T_\Sigma(X)$

Una vez conseguido el soporte sintáctico, es necesario ponerlo explícitamente en relación con su significado, o sea, con el álgebra que

soporta. Esto es necesario para asegurar la correspondencia entre géneros y conjuntos y entre símbolos de operación y funciones.

2.9 DEFINICIÓN. Sea una signatura heterogénea (S, Σ). Se dice que un álgebra A es una **Σ-álgebra** si está formada por:

- Una familia S-indexada de **conjuntos soporte** no vacíos, $A = \{ A_s \mid s \in S \}$.
- Un **conjunto de funciones** tal que para cada símbolo de operación $\sigma \in \Sigma_{w,s}$ existe una función $A_\sigma^{w,s} : A_w \rightarrow A_s$ donde $A_w = A_{s1} \times \ldots \times A_{sn}$ si $w = s1 \ldots sn$, y donde si $w = \varepsilon$, $A_\sigma^{\varepsilon,s}$ es un elemento de A_s.

Como puede observarse, una misma signatura puede dar soporte sintáctico a muchas álgebras distintas, ya que la exigencia impuesta por la definición 2.9 es que conjuntos y géneros sean iguales en número y que los símbolos de operación y las funciones también lo sean, además de que los parámetros de las funciones sean coherentes en número y género con los argumentos de los símbolos de operación. De esto se debe extraer una conclusión interesante, los símbolos en sí mismos no significan nada (al menos hasta el capítulo 6) y cualquier manipulación que se haga sobre ellos debe ser interpretada en base a un álgebra soporte.

EJEMPLO 2.6

Una posible Σ-álgebra de la signatura del ejemplo 2.4 puede ser:

$$Alg1 \equiv (\mathrm{B}, \mathrm{Z}, \mathrm{t}, \mathrm{f}, \neg, +, -, -, >)$$

es decir, el conjunto de los booleanos como soporte del género *Bool*, $Alg1_{\mathrm{Bool}} \equiv \mathrm{B}$, y de los enteros como soporte del género *Ent*, $Alg1_{\mathrm{Ent}} \equiv \mathrm{Z}$, junto con las funciones siguientes:

- el *cierto lógico* como soporte del símbolo *cierto*
  $Alg1_{\mathrm{cierto}}^{\varepsilon,\,\mathrm{Bool}} \equiv \mathrm{t} \in \mathrm{B}$
- el *falso lógico* como soporte del símbolo *falso*
  $Alg1_{\mathrm{falso}}^{\varepsilon,\,\mathrm{Bool}} \equiv \mathrm{f} \in \mathrm{B}$
- la *negación lógica* como soporte del símbolo *no*

$Alg1^{Bool,\,Bool}_{no} \equiv B_{\neg} : B \rightarrow B$

- la *suma de enteros* como soporte del símbolo +

  $Alg1^{Ent.Ent,\,Ent}_{+} \equiv Z_{+} : Z \times Z \rightarrow Z$

- la *resta de enteros* como soporte del símbolo binario -

  $Alg1^{Ent.Ent,\,Ent}_{-} \equiv Z_{-} : Z \times Z \rightarrow Z$

- la *negación entera* como soporte del símbolo monario -

  $Alg1^{Ent,\,Ent}_{-} \equiv Z_{-} : Z \rightarrow Z$

- la comparación de enteros como soporte del símbolo >

  $Alg1^{Ent.Ent,\,Ent}_{>} \equiv Z_{>} : Z \times Z \rightarrow B$

Otra posible Σ-álgebra podría ser:

$$Alg2 \equiv (Z, B, -1, 1, -, \wedge, \vee, \neg, 7)$$

es decir, el conjunto de los enteros como soporte del género *Bool*, $Alg2_{Bool} \equiv Z$, y el de los booleanos como soporte del género *Ent*, $Alg2_{Ent} \equiv B$, junto con las funciones siguientes:

- el *menos uno* como soporte del símbolo *cierto*

  $Alg2^{\varepsilon,\,Bool}_{cierto} \equiv -1 \in Z$

- el *uno* como soporte del símbolo *falso*

  $Alg2^{\varepsilon,\,Bool}_{falso} \equiv 1 \in Z$

- la *negación entera* como soporte del símbolo *no*

  $Alg2^{Bool,\,Bool}_{no} \equiv Z_{-} : Z \rightarrow Z$

- la *disyunción lógica* como soporte del símbolo +

  $Alg2^{Ent.Ent,\,Ent}_{+} \equiv B_{\wedge} : B \times B \rightarrow B$

- la *conjunción lógica* como soporte del símbolo binario -

  $Alg2^{Ent.Ent,\,Ent}_{-} \equiv B_{\vee} : B \times B \rightarrow B$

- la *negación lógica* como soporte del símbolo monario -

  $Alg2^{Ent,\,Ent}_{-} \equiv B_{\neg} : B \rightarrow B$

- la *función constante siete* como soporte del símbolo >

  $Alg2^{Ent.Ent,\,Ent}_{>} \equiv f_7 : B \times B \rightarrow Z \mid \forall x,y \in B,\ f_7(x, y) = 7$

Como puede verse la relación signatura-álgebra es completamente arbitraria. Sin embargo no, debe considerarse una deficiencia del enfoque ya

que es comparable al adoptado por la mayor parte de las herramientas de síntesis, que asocian a cada símbolo que aparece en la especificación un significado implícito inalterable. Con este enfoque, al menos, podremos cambiar el conjunto de símbolos aceptable, y en el capítulo 6 comprobaremos como también podemos concretar qué modelos son válidos para cierta signatura.

Si todo género tiene asociado un conjunto soporte y todo símbolo de operación una función, sería interesante poder evaluar el valor que denota un término cualquiera. Para ello primero se define la noción de valoración que permite asignar valores a un conjunto de variables y a continuación la noción de interpretación que, dada una valoración, permite calcular el valor de cualquier término. Obsérvese que, por el momento, no se dará significado a los términos con variables libres, todas deberán estar ligadas por una valoración.

2.10 Definición. Dada una signatura heterogénea (S, Σ), una familia S-indexada de conjuntos de variables X, y una Σ-álgebra A, definimos **valoración** como toda asignación de valores de A a las variables de X, $\mu : X \to A$, donde μ representa al conjunto de valoraciones S-indexado $\mu = \{ \mu_s : X_s \to A_s \mid s \in S \}$.

2.11 Definición. Dada una signatura heterogénea (S, Σ), una familia S-indexada de conjuntos de variables libres X, una Σ-álgebra A, y una valoración $\mu : X \to A$, definimos **Interpretación** como la asignación de valores a todo el conjunto de términos, $\hat{\mu} : T_\Sigma(X) \to A$, definida recursivamente sobre la estructura del término como sigue:

- $\hat{\mu}(x) = \mu(x), \quad \forall x \in X$
- $\hat{\mu}(\sigma) = A_\sigma^{\varepsilon,s}, \quad \forall \sigma \in \Sigma_{\varepsilon,s}$
- $\hat{\mu}(\sigma(t_1, \ldots, t_n)) = A_\sigma^{w,s}(\hat{\mu}(t_1), \ldots, \hat{\mu}(t_n)), \quad \forall \sigma \in \Sigma_{w,s}$ con $t_i \in T_\Sigma(X)$, $i \in \{1,\ldots,n\}$

Cuando $X \equiv \emptyset$, existe una única valoración μ llamada **valoración vacía**. Además puede comprobarse que la **interpretación de un término cerrado** es independiente de valoraciones, dicha interpretación es fija para un álgebra soporte dada, *A*, y se representa por t^A.

EJEMPLO 2.7

Evaluemos los términos del ejemplo 2.5 para distintas valoraciones realizadas sobre las álgebras soporte del ejemplo 2.6. Adoptemos inicialmente el álgebra *Alg*1 del ejemplo 2.6 y hagamos la siguiente valoración del conjunto de variables *X*, $\mu \equiv \{ \mu_{Bool} \equiv \{ (b,t) \}, \mu_{Ent} \equiv \{ (x,2), (y,3) \} \}$ (recuérdese que *t*, 2 y 3 son elementos abstractos del álgebra, mientras que *b*, *x* e *y* son símbolos de variable):

- $\hat{\mu}(x) = \mu(x) = 2$
- $\hat{\mu}(\text{cierto}) = Alg1^{\varepsilon,\ Bool}_{cierto} \equiv t$
- $\hat{\mu}(+(x, y)) = Alg1^{Ent.Ent,\ Ent}_{+}(\hat{\mu}(x), \hat{\mu}(y)) =$
  $= Alg1^{Ent.Ent,\ Ent}_{+}(\mu(x), \mu(y)) =$
  $= Alg1^{Ent.Ent,\ Ent}_{+}(2, 3) \equiv$
  $\equiv Z_{+}(2, 3) = 5$
- $\hat{\mu}(no(no(falso))) = Alg1^{Bool,\ Bool}_{no}(\hat{\mu}(no(falso))) =$
  $= Alg1^{Bool,\ Bool}_{no}(Alg1^{Bool,\ Bool}_{no}(\hat{\mu}(falso))) =$
  $= Alg1^{Bool,\ Bool}_{no}(Alg1^{Bool,\ Bool}_{no}(Alg1^{\varepsilon,\ Bool}_{falso})) \equiv$
  $\equiv B_{\neg}(B_{\neg}(f)) = f$
- $\hat{\mu}(no (-(x) > (x + y))) = \ldots =$
  $= Alg1^{Bool,\ Bool}_{no}(Alg1^{Ent.Ent,\ Bool}_{>}(Alg1^{Ent,\ Ent}_{-}(2),Alg1^{Ent.Ent,\ Ent}_{+}(2,3))) \equiv$
  $\equiv B_{\neg}(Z_{>}(Z_{-}(2), Z_{+}(2, 3))) = t$

Si por el contrario, adoptamos el algebra *Alg*2 del ejemplo 2.6 y hacemos la valoración $\mu \equiv \{ \mu_{Bool} \equiv \{ (b,1) \}, \mu_{Ent} \equiv \{ (x,f), (y,f) \} \}$, las interpretaciones son considerablemente distintas ya que el comportamiento de las funciones soporte es bien distinto.

- $\hat{\mu}(x) = \mu(x) = f$

- $\hat{\mu}$(cierto) = $Alg2^{\varepsilon,\ Bool}_{cierto} \equiv -1$
- $\hat{\mu}$(+(x, y)) = ... =
  $= Alg2^{Ent.Ent,\ Ent}_{+}$(f, f) ≡
  ≡ $B_{\wedge}$(f, f) = f
- $\hat{\mu}$(no(no(falso))) = ... =
  $= Alg2^{Bool,\ Bool}_{no}$($Alg2^{Bool,\ Bool}_{no}$($Alg2^{\varepsilon,\ Bool}_{falso}$)) ≡
  ≡ Z_{-}(Z_{-}(1)) = 1
- $\hat{\mu}$(no (-(x) > (x + y))) = ... =
  $= Alg2^{Bool,Bool}_{no}(Alg2^{Ent.Ent,Bool}_{>}(Alg2^{Ent,Ent}_{-}(f), Alg2^{Ent.Ent,Ent}_{+}(f,f)))$ ≡
  ≡ Z_{-}(f_{7}($B_{\neg}$(f), $B_{\wedge}$(f, f))) =
  = Z_{-}(f_{7}(t, f)) = Z_{-}(7) = -7

Para finalizar obsérvese que, de modo natural, un par de términos son capaces de expresar una ecuación. Las ecuaciones son útiles ya que permiten establecer equivalencias algebraicas y ofrecer un mecanismo simple de razonamiento simbólico (como se comprobará en el capítulo 3).

2.12 Definición. Sea una signatura (S, Σ) y una familia S-indexada de conjuntos de variables Y. Se define **Σ-ecuación de género s** como la tripleta (Y, t_L, t_R) donde $t_L, t_R \in T_{\Sigma,s}(Y)$ tal que representa a la fórmula universal de primer orden siguiente:

$$\forall x_1 \in s_1 \ ... \ \forall x_n \in s_n \ (t_L = t_R) \text{ donde } x_i \in Y \wedge s_i \in S$$

Por conveniencia, si los identificadores de variables y sus géneros son claros, una ecuación suele notarse simplemente por $t_L = t_R$.

2.13 Definición. Sea una signatura (S, Σ) y una familia S-indexada de conjuntos de variables Y. Se dice que una ecuación (Y, t_L, t_R) es **válida** en una Σ-álgebra A, y se escribe $A \models$ (Y, t_L, t_R), si para toda valoración $\mu : Y \rightarrow A$ se

cumple que $\hat{\mu}(t_L) = \hat{\mu}(t_R)$. Si una ecuación es válida en un álgebra también puede decirse indistintamente que el álgebra **satisface** la ecuación.

Esta definición viene a decir que cuando una ecuación es válida, las interpretaciones de los términos t_L y t_R son iguales independientemente del valor que tomen las variables. Nótese que la noción de validez es otro concepto que también depende del álgebra soporte, por lo que una misma ecuación puede ser válida en un álgebra e inválida en otra.

EJEMPLO 2.8

Sea la signatura del ejemplo 2.4. La siguiente tripleta es una ecuación de género *Ent*:

$$(Y \equiv \{ Y_{Ent} \equiv \{ x, y \} \}, x - y, x + (-y))$$

que representa a la fórmula universal de primer orden:

$$\forall x \in Ent, \forall y \in Ent, (x - y = x + (-y))$$

Comprobemos si es válida para cada una de las álgebras del ejemplo 2.6. Sea el álgebra *Alg*1, y sea $\mu \equiv \{ \mu_{Ent} : Y_{Ent} \rightarrow Z \}$ una valoración genérica.

$\hat{\mu}(x - y) = ... =$

$= Alg1_{-}^{Ent.Ent,\ Ent}(\mu(x), \mu(y)) \equiv$ *definición 2.11*

$\equiv Z_{-}(\mu(x), \mu(y)) = Z_{+}(\mu(x), Z_{-}(\mu(y))) \equiv$ *propiedad de la resta entera*

$\equiv Alg1_{-}^{Ent.Ent,\ Ent}(\mu(x), Alg1_{-}^{Ent,\ Ent}(\mu(y))) = ... =$

$= \hat{\mu}(x + (- y))$ *definición 2.11*

Luego esta álgebra satisface la ecuación. Sea ahora el álgebra soporte *Alg*2, y sea esta vez la valoración genérica $\mu \equiv \{ \mu_{Ent} : Y_{Ent} \rightarrow B \}$ (ya que el álgebra soporte del género *Ent* es el conjunto de los booleanos).

$\hat{\mu}(x - y) = ... =$

$= Alg2_{-}^{Ent.Ent,\ Ent}(\mu(x), \mu(y)) \equiv$ *definición 2.11*

$= B_{\vee}(\mu(x), \mu(y)) \neq B_{\wedge}(\mu(x), B_{\neg}(\mu(y))) \equiv ... =$ *según el álgebra de Boole*

$\equiv Alg2_{-}^{Ent.Ent,\ Ent}(\mu(x), Alg2_{-}^{Ent,\ Ent}(\mu(y))) = ... =$

$= \hat{\mu}(x + (- y))$ *definición 2.11*

Como puede comprobarse, esta segunda álgebra no satisface la ecuación. Como consecuencia nunca debe confiarse completamente en un soporte sintáctico ya que los símbolos (cuando no tienen una semántica fija) pueden inducir a razonamientos incorrectos.

2.4.2 Extensiones del álgebra estática.

A continuación se formalizarán las extensiones tanto sintácticas como semánticas que deben realizarse respectivamente sobre signaturas y álgebras para aumentar su expresividad a la hora de describir circuitos digitales. A diferencia de la elección de álgebras y signaturas, que puede ser distinta para cada diseño, estas extensiones son fijas y, en un enfoque automático, deben ser asumidas por el diseñador ya que se realizan implícitamente. La razón es que definen un proceso necesario que, por su complejidad, es conveniente que sea ocultado. Este proceso permite construir, a partir de álgebras estáticas simples, álgebras dinámicas complejas sobre las que puedan definirse sin ambigüedad funciones recursivas. Básicamente este mecanismo es el que permite que el diseñador pueda abstraerse de conceptos dinámicos a la hora de especificar, ya que sobrecarga sucesivamente la sintaxis por él definida, con conceptos semánticos más complejos.

Extensión para soportar indiferencia.

El objetivo de la síntesis no es obtener un diseño que se comporte igual que la especificación, sino obtener un sistema cuyo comportamiento en ciertas ventanas temporales sea el especificado. Al igual que dos implementaciones se consideran equivalentes a nivel lógico si, al cabo de cierto tiempo de cálculo, los puertos del sistema se estabilizan en los mismos valores

independientemente de cual haya sido la evolución seguida para alcanzarlos, dos implementaciones serán equivalentes a nivel algorítmico si, en los momentos en que tiene que transferir datos con el entorno, los puertos tienen los mismos valores independientemente de los valores intermedios calculados.

Para poder expresar esa independencia que un cálculo tiene respecto a los valores que temporalmente tome cierta señal, se debe añadir a cada uno de los dominios de valores un elemento que denote indiferencia. Este elemento aumentará el poder expresivo del especificador y permitirá al diseñador optimizar los resultados. En nuestro caso, dado que el objetivo es la síntesis algorítmica, la indiferencia será utilizada para marcar todos aquellos valores calculados o almacenados pero no utilizados en posteriores cálculos. Téngase en cuenta que en un circuito con recursos multiplexados en el tiempo, los recursos nunca dejan de hacer cálculos aunque algunos de los resultados generados no se utilicen.

2.14 Definición. Sea un conjunto D, definimos $D^{\#}$ como el conjunto resultante de añadir un nuevo elemento llamado **comodín**, para denotar la indiferencia.

El significado de este elemento (la indiferencia) permite definir de modo natural una relación binaria, reflexiva y simétrica que será llamada compatibilidad. Esta permite relacionar elementos que no son contradictorios, o sea, que si transportan información relevante, esta nunca es distinta. Obviamente esta definición puede resultar artificiosa sobre dominios elementales pero, como se comprobará en próximas secciones, cobra un significado excepcional cuando se trata con dominios de secuencias de valores.

2.15 Definición. Se dice que dos elementos pertenecientes al mismo conjunto $x,y \in D^{\#}$ son **compatibles**

$$x \approx y \Leftrightarrow (x \equiv \#) \vee (y \equiv \#) \vee (x \equiv y)$$

Un álgebra no sólo es un conjunto de valores sino también un conjunto de funciones: no basta con extender el dominio, sino también es necesario extender el comportamiento de las funciones para que operen coherentemente sobre el dominio extendido. Nuevamente el significado del elemento comodín nos guía para definir el modo en que deben extenderse dichos comportamientos: si al menos uno de los argumentos de una función es indiferente, es razonable pensar que la función está involucrada en un cálculo que no será utilizado, por lo que el resultado también deberá ser indiferente. Este tipo de extensión es la que se denomina extensión estricta.

2.16 Definición. Sea la función $f : D_1 \times ... \times D_n \to D$. Se define la **extensión estricta** de f respecto de los nuevos elementos $\#_1, ..., \#_n, \#$ como la función $g : D_1^{\#} \times ... \times D_n^{\#} \to D^{\#}$ que se define como:

$$g(x_1, ..., x_n) = \# \Leftrightarrow \exists i \in \{1..n\} \mid x_i \equiv \#_i$$

$$g(x_1, ..., x_n) = f(x_1, ..., x_n) \Leftrightarrow \nexists i \in \{1..n\} \mid x_i \equiv \#_i$$

Finalmente ya es posible definir la extensión de signaturas para soportar indiferencia (extensión sintáctica) y la extensión de Σ-álgebras para soportar indiferencia (extensión semántica).

2.17 Definición. Sea (S, Σ) una signatura heterogénea. Se define su **extensión (sintáctica) para soportar indiferencia** como la nueva signatura $(S, \Sigma^{\#})$, tal que $\Sigma^{\#}$ es la familia $S^* \times S$-indexada de conjuntos que verifica:

- $\Sigma^{\#}_{w,s} \equiv \Sigma_{w,s}$ si $w \neq \varepsilon$
- $\Sigma^{\#}_{\varepsilon,s} = \Sigma_{\varepsilon,s} \cup \{ \# \}$.

Obsérvese que cada género es ampliado con un símbolo de comodín distinto, aunque usualmente por simplicidad, se escribirá sin subindexar si es posible extraer su género del contexto.

2.18 Definición. Sea $\Sigma^{\#}$ la signatura definida a partir de (S, Σ), y sea A una Σ-álgebra. Se define **la extensión (semántica) para soportar indiferencia** de A, $A^{\#}$, como la $\Sigma^{\#}$-álgebra que cumple:

- El universo de $A^{\#}$ es la familia S-indexada de conjuntos soporte ampliados con el elemento comodín $A^{\#} = \{ (A_s)^{\#} \mid s \in S \}$.
- Cualquier símbolo de operación $\sigma \in \Sigma$ que denota la función $A_{\sigma}^{w,s}$, permite definir la denotación del mismo símbolo $\sigma \in \Sigma^{\#}$, como la extensión estricta de $A_{\sigma}^{w,s}$ respecto al elemento comodín.

Extensión para soportar indefinición.

Dado que el formalismo de especificación que se va a proponer permite definiciones recursivas explícitas, es necesario añadir un nuevo elemento para expresar cuando la descripción de una computación recursiva no tiene sentido†. Desde el punto de vista operacional este nuevo elemento denotará la no terminación del algoritmo especificado (o más propiamente de la simulación del algoritmo especificado). Desde el punto de vista de diseñador hardware este símbolo denotará la no estabilización del correspondiente circuito RT y, por tanto, la invalidez del modelo RT para tratar el problema.

EJEMPLO 2.9

Analicemos la siguiente especificación temporal:

$$x(t) = x(t) + 1$$

Desde el punto de vista matemático existe una inconsistencia: no existe ninguna secuencia de valores pertenecientes a ningún dominio útil que sea igual a si misma incrementada en 1. Por consiguiente la solución de esta ecuación, que es la semántica de la especificación, debe ser una secuencia infinita de elementos indefinidos.

Desde el punto de vista operacional, un simulador convencional entrará en un ciclo de simulación infinito y el algoritmo no terminará. Suponiendo una simulación bajo demanda, obsérvese el comportamiento del simulador: para

† Véase el apéndice B, para una discusión detallada.

calcular el valor de $x(0)$ debe evaluar el lado derecho de su definición en $t = 0$, eso le hace que demande el valor de $x(0)$ lo que obliga a intentar evaluar nuevamente el lado derecho de su definición en $t = 0$ y así una y otra vez hasta el infinito. Obviamente un simulador no tan convencional, avisará que la especificación es no simulable ya que este tipo de no terminación es fácilmente detectable. El símbolo indefinido puede denotar el resultado de simulaciones fallidas.

Finalmente desde el punto de vista de un diseñador hardware, esta especificación es directamente implementable mediante un sumador combinacional realimentado. Esta implementación deja de ser síncrona y, en general, no permitirá que el valor de salida se estabilice haciendo el modelo inaplicable. Nuevamente el símbolo indefinido permite que el formalismo pueda expresar cuándo no es aplicable.

Además será necesario añadir una relación de orden parcial y dotar al conjunto extendido con estructura de retículo. Esta estructura será posteriormente heredada por el espacio de funciones construido sobre el dominio extendido y permitirá localizar sin ambigüedad cual es la función que denota toda definición recursiva. El enfoque adoptado ya es clásico en la formalización de la semántica de los lenguajes de programación [Watt93] [Alli86] [Stoy81] y en el apéndice B se incluye un resumen del núcleo teórico.

2.19 **Definición.** Sea un conjunto D. Definimos el **dominio plano** $D_\perp$ como la tupla $(D, \sqsubseteq, \perp)$ donde $\perp$ (léase **fondo**) es un nuevo elemento que denota indefinición y $\sqsubseteq$ (léase **aproximación**) es una relación de orden parcial definida como:

$$\forall x,y \in D_\perp,\ x \sqsubseteq y \Leftrightarrow (x \equiv \perp \vee x \equiv y)$$

En esta extensión, a diferencia de la anterior, no se amplia el soporte sintáctico, ya que ni el elemento fondo ni la relación de orden parcial son

útiles para el diseñador: una función indefinida debe ser el resultado de una especificación incorrecta pero nunca de una especificación explícitamente indefinida. La extensión semántica, sin embargo, si que se realizará ya que el nuevo elemento y la relación de orden son conceptos puramente semánticos.

2.20 DEFINICIÓN. Sea Σ la signatura definida a partir de (S, Σ), y sea A una Σ-álgebra. Se define la **extensión (semántica) para soportar indefinición** de A, $A_\perp$, como la Σ-álgebra que cumple:

- El universo de $A_\perp$ es la familia S-indexada de dominios planos $A_\perp = \{ (A_s)_\perp \mid s \in S \}$.
- Cualquier símbolo de operación $\sigma \in \Sigma$ que denota la función $A_\sigma^{w,s}$, permite definir la denotación del mismo símbolo como la extensión estricta de $A_\sigma^{w,s}$ respecto al elemento fondo.

Extensión para modelar el tiempo: extensión temporal.

Hasta ahora hemos supuesto que todas las álgebras estaban formadas por dominios cuyos elementos eran atómicos, en cierto modo eran álgebras estáticas. Sin embargo para modelar el comportamiento de un sistema digital en el que los valores evolucionan en el tiempo, necesitamos álgebras algo más complejas. Este es el objetivo de la extensión temporal: definir un álgebra de secuencias infinitas de elementos pertenecientes a un álgebra estática, o lo que es lo mismo, el álgebra formada por todas las aplicaciones de $\mathbb{N}_+$ a cierta álgebra estática.

Esta álgebra temporal posee un número infinito de posibles operadores, pero existen algunos de ellos que se definen de un modo natural a partir de los operadores que poseía el álgebra estática primitiva, estos operadores son los que se llamarán **no temporales** o **combinacionales**. La característica que los distingue (como explica la fig. 2.6) es que cuando manipulan secuencias

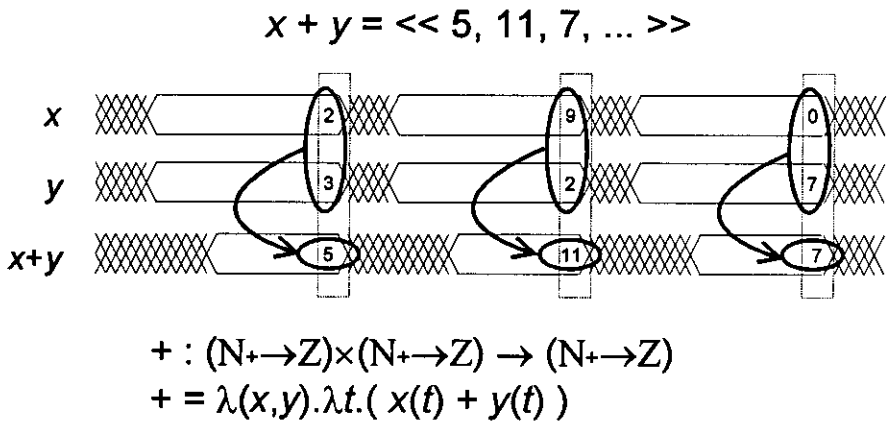


Fig. 2.6: Comportamiento de un operador no temporal.

de valores lo hacen punto a punto, o sea, que para calcular el resultado de la posición *n* de una secuencia sólo necesitan los valores en la posición *n* de sus argumentos. Del resto de los operadores, que se llamarán **temporales** o **secuenciales**, sólo nos interesará un pequeño conjunto que definiremos en el capítulo 4.

Desde el punto de vista sintáctico no es necesario hacer ninguna extensión para soportar operadores no temporales ya que los símbolos que servían para expresar operaciones estáticas pueden reutilizarse para denotar operaciones punto a punto. El soporte sintáctico de operadores temporales se presentará en la próxima sección.

Para formalizar la extensión semántica (que sí es necesaria, al igual que otros muchos conceptos que se presentarán en próximos capítulos), puede utilizarse indistintamente uno de los dos enfoques equivalentes siguientes: el basado en secuencias infinitas o el basado en funciones temporales. Si bien trabajar con secuencias es más intuitivo, hacerlo con funciones es más compacto y más fácilmente manipulable. La elección concierne más al gusto

que a la ciencia y de ahora en adelante se adoptará para el resto de la presente memoria la segunda alternativa. No obstante, el lector podrá alternar una visión u otra según qué opción le sea más fácil en cada contexto.

Para ilustrar las diferencias notacionales entre ellas, a continuación se presentarán dos definiciones equivalentes de **extensión (semántica) temporal**.

2.21 Definición. Sea Σ la signatura definida a partir de (S, Σ), y sea A una Σ-álgebra. Se define A^{∞} como la Σ-álgebra que cumple:

- El universo de A^{∞} es la familia S-indexada de conjuntos soporte A^{∞}_{s} cada uno de los cuales es la colección de todas las secuencias infinitas de objetos de género s pertenecientes al universo de A_s.
- Cualquier símbolo de operación $\sigma \in \Sigma$ que denota la función $A^{w,s}_{\sigma}$, permite definir la denotación del mismo símbolo como la función $(A^{\infty})^{w,s}_{\sigma}$ tal que para toda secuencia $x_i \in A^{\infty}$ y para todo índice $t \in \mathbb{N}_+$ se cumple que:

$$((A^{\infty})^{w,s}_{\sigma}(x_1, \ldots, x_n))_t = A^{w,s}_{\sigma}((x_1)_t, \ldots, (x_n)_t)$$

2.22 Definición. Sea Σ la signatura definida a partir de (S, Σ), y sea A una Σ-álgebra. Se define $(\mathbb{N}_+ \rightarrow A)$ como la Σ-álgebra que cumple:

- El universo de $(\mathbb{N}_+ \rightarrow A)$ es la familia S-indexada de conjuntos soporte $(\mathbb{N}_+ \rightarrow A)_s$ cada uno de los cuales es la colección de todas las funciones de dominio natural y codominio A_s, o sea, $(\mathbb{N}_+ \rightarrow A_s)$.
- Cualquier símbolo de operación $\sigma \in \Sigma$ que denota la función $A^{w,s}_{\sigma}$, permite definir la denotación del mismo símbolo como la función $(\mathbb{N}_+ \rightarrow A)^{w,s}_{\sigma}$ tal que para toda función $x_i \in (\mathbb{N}_+ \rightarrow A)$ y para todo $t \in \mathbb{N}_+$ se cumple:

$$((\mathbb{N}_+ \rightarrow A)^{w,s}_{\sigma}(x_1, \ldots, x_n))(t) = A^{w,s}_{\sigma}(x_1(t), \ldots, x_n(t))$$

Obsérvese que la sintaxis permanece inalterada y que es la semántica la que se extiende. Esto es posible gracias a la explícita separación entre signatura y Σ-álgebra.

EJEMPLO 2.10

Sea la signatura del ejemplo 2.4, y consideremos la nueva Σ-álgebra que se obtiene como extensión temporal del Σ-álgebra *Alg*1 definida en el ejemplo 2.6. Según la definición 2.22, $(\mathbf{N}_+ \rightarrow Alg1)$ tendrá como conjuntos soporte el conjunto de todas las secuencias infinitas de booleanos, $(\mathbf{N}_+ \rightarrow Alg1)_{Bool} = (\mathbf{N}_+ \rightarrow Alg1_{Bool}) \equiv (\mathbf{N}_+ \rightarrow \mathbf{B})$ y el conjunto de todas las secuencias infinitas de enteros, $(\mathbf{N}_+ \rightarrow Alg1)_{Ent} = (\mathbf{N}_+ \rightarrow Alg1_{Ent}) \equiv (\mathbf{N}_+ \rightarrow \mathbf{Z})$, conjuntos denotados respectivamente por los géneros *Bool* y *Ent*.

Asimismo, de todas las operaciones que pueden existir en los nuevos dominios, la extensión temporal concreta cuáles de ellas (todas no temporales) serán las denotadas por cada símbolo de operación de la signatura†. Así:

- $(\mathbf{N}_+ \rightarrow Alg1)^{\varepsilon,\ Bool}_{cierto}$ *secuencia infinita de elementos cierto lógico*

  $\lambda k : \mathbf{N}_+ .\ t : \mathbf{B} \equiv \langle\langle\ t, t, t \ldots \rangle\rangle$

- $(\mathbf{N}_+ \rightarrow Alg1)^{\varepsilon,\ Bool}_{falso}$ *secuencia infinita de elementos falso lógico*

  $\lambda t : \mathbf{N}_+ .\ f : \mathbf{B} \equiv \langle\langle\ f, f, f \ldots \rangle\rangle$

- $(\mathbf{N}_+ \rightarrow Alg1)^{Bool,\ Bool}_{no}$ *negación booleana punto a punto sobre secuencias infinitas*

  $\lambda x : (\mathbf{N}_+ \rightarrow \mathbf{B}).\ (\ \lambda t : \mathbf{N}_+ .\ B_{\neg}(\ x(t)\ ) : \mathbf{B}\ ) : (\mathbf{N}_+ \rightarrow \mathbf{B})$

- $(\mathbf{N}_+ \rightarrow Alg1)^{Ent.Ent,\ Ent}_{+}$ *suma de enteros punto a punto sobre secuencias infinitas*

  $\lambda(x,y) : (\mathbf{N}_+ \rightarrow \mathbf{Z}) \times (\mathbf{N}_+ \rightarrow \mathbf{Z}).\ (\ \lambda t : \mathbf{N}_+ .\ Z_+(\ x(t), y(t)\ ) : \mathbf{Z}\ ) : (\mathbf{N}_+ \rightarrow \mathbf{Z})$

- $(\mathbf{N}_+ \rightarrow Alg1)^{Ent.Ent,\ Ent}_{-}$ *resta de enteros punto a punto sobre secuencias infinitas*

  $\lambda(x,y) : (\mathbf{N}_+ \rightarrow \mathbf{Z}) \times (\mathbf{N}_+ \rightarrow \mathbf{Z}).\ (\ \lambda t : \mathbf{N}_+ .\ Z_-(\ x(t), y(t)\ ) : \mathbf{Z}\ ) : (\mathbf{N}_+ \rightarrow \mathbf{Z})$

- $(\mathbf{N}_+ \rightarrow Alg1)^{Ent,\ Ent}_{-}$ *negación entera punto a punto sobre secuencias infinitas*

  $\lambda x : (\mathbf{N}_+ \rightarrow \mathbf{Z}).\ (\ \lambda t : \mathbf{N}_+ .\ Z_-(\ x(t)\ ) : \mathbf{Z}\ ) : (\mathbf{N}_+ \rightarrow \mathbf{Z})$

- $(\mathbf{N}_+ \rightarrow Alg1)^{Ent.Ent,\ Bool}_{>}$ *comparación entera punto a punto de secuencias infinitas*

  $\lambda(x,y) : (\mathbf{N}_+ \rightarrow \mathbf{Z}) \times (\mathbf{N}_+ \rightarrow \mathbf{Z}).\ (\ \lambda t : \mathbf{N}_+ .\ Z_>(\ x(t), y(t)\ ) : \mathbf{B}\ ) : (\mathbf{N}_+ \rightarrow \mathbf{B})$

† Se utilizará una λ-notación con tipos para expresar las funciones abstractas. El apéndice A repasa los conceptos fundamentales del λ-cálculo.

Bajo la nueva álgebra soporte es también posible evaluar términos. Para la valoración $\mu \equiv \{ \mu_{Bool}, \mu_{Ent} \}$ donde (asumiendo que $\mathbf{N}_+ \subset Z \equiv Alg1_{Ent}$):

$$\mu_{Bool} \equiv \{ (b, \lambda k{:}\mathbf{N}_+.t{:}B \equiv \langle\langle t, t, t \ldots \rangle\rangle) \}$$

$\mu_{Ent} \equiv \{ (x, \lambda t :\mathbf{N}_+.\ t :Z \equiv \langle\langle 1, 2, 3 \ldots \rangle\rangle), (y, \lambda t :\mathbf{N}_+.\ t\text{-}2 :Z \equiv \langle\langle -1, 0, 1 \ldots \rangle\rangle) \}$

la interpretación de los términos del ejemplo 2.7 es:

- $\hat{\mu}(x) = \mu(x) = \lambda t :\mathbf{N}_+.\ t :Z \equiv \langle\langle 1, 2, 3 \ldots \rangle\rangle$
- $\hat{\mu}(\text{cierto}) = (\mathbf{N}_+ \rightarrow Alg1)^{\varepsilon,\ Bool}_{cierto} \equiv \lambda k :\mathbf{N}_+.\ t :B \equiv \langle\langle t, t, t \ldots \rangle\rangle$
- $\hat{\mu}(+(x, y)) = \ldots =$

  $= (\mathbf{N}_+ \rightarrow Alg1)^{Ent.Ent,\ Ent}_{+}(\lambda t :\mathbf{N}_+.\ t :Z, \lambda t :\mathbf{N}_+.\ t\text{-}2 :Z) =$

  $= \lambda t :\mathbf{N}_+.\ Z_-(t, t\text{-}2) :Z =$ *asumiendo que '-' ≡ Z_-*

  $= \lambda t :\mathbf{N}_+.\ (-2) :Z \equiv \langle\langle -2, -2, -2 \ldots \rangle\rangle$ *operando sobre enteros*
- $\hat{\mu}(\text{no}(\text{no}(\text{falso}))) = \ldots =$

  $= (\mathbf{N}_+ \rightarrow Alg1)^{Bool,\ Bool}_{no}((\mathbf{N}_+ \rightarrow Alg1)^{Bool,\ Bool}_{no}(\lambda t :\mathbf{N}_+.\ f :B) \equiv$

  $= \ldots = \lambda t :\mathbf{N}_+.\ f :B \equiv \langle\langle f, f, f \ldots \rangle\rangle$
- $\hat{\mu}(\text{no} (-(x) > (x + y))) = \ldots =$

  $= (\mathbf{N}_+ \rightarrow Alg1)^{Bool,\ Bool}_{no}($

  $(\mathbf{N}_+ \rightarrow Alg1)^{Ent.Ent,\ Bool}_{>}($

  $(\mathbf{N}_+ \rightarrow Alg1)^{Ent,\ Ent}_{-}(\lambda t{:}\mathbf{N}_+.t{:}Z),$

  $(\mathbf{N}_+ \rightarrow Alg1)^{Ent.Ent,\ Ent}_{+}(\lambda t :\mathbf{N}_+.\ t :Z, \lambda t :\mathbf{N}_+.\ t\text{-}2 :Z))) =$

  $= \ldots =$ *operando sobre enteros*

  $= \lambda t :\mathbf{N}_+.\ f :B \equiv \langle\langle f, f, f \ldots \rangle\rangle$

Principio de extensión temporal.

Como muestra de las potenciales ventajas que pueden obtenerse gracias a la sobrecarga de símbolos que se realiza en una extensión temporal, propondré el **principio de extensión temporal**. Este principio permite, en algunas ocasiones, abstraer el tiempo cuando se razona sobre secuencias ya

que establece que toda deducción que se realice sobre un álgebra estática, es también válida sobre la correspondiente álgebra dinámica: simplificando algunas de las pruebas que se realicen sobre dominios dinámicos. Asimismo, este resultado se utilizará en la próxima sección para la definición de un principio más general (el principio de *Lu*-extensión) que será decisivo para edificar parte del sistema de síntesis formal que se presentará en esta memoria.

2.23 PROPOSICIÓN. Si una Σ-álgebra, *A*, satisface una Σ-ecuación *e*, entonces la Σ-álgebra $(\mathbf{N}_+ \to A)$ también la satisface.

DEMOSTRACIÓN. Sea la ecuación $e \equiv (X, t_L, t_R)$. Toda valoración sobre el álgebra dinámica $\mu : X \to (\mathbf{N}_+ \to A)$ permite definir un conjunto de valoraciones sobre el álgebra estática *A*, donde cada una de ellas representa el valor que adoptan las variables en un instante concreto:

$$M = \{ \mu_t : X \to A \mid \forall t \in \mathbf{N}_+, \forall x \in X, \mu_t(x) = \mu(x)(t) \}$$

Según la definición 2.13, $A \models e \Leftrightarrow \forall(\mu : X \to A), \hat{\mu}(t_L) = \hat{\mu}(t_R)$ luego, en particular, $A \models e \Rightarrow \forall \mu \in M, \hat{\mu}(t_L) = \hat{\mu}(t_R)$, lo que viene a significar que en todo instante temporal se satisface la ecuación *e*, es decir, que $(\mathbf{N}_+ \to A) \models e$.

□

2.4.3 Especificación ecuacional.

El ejemplo 2.10 muestra cómo mediante una λ-notación es posible describir compactamente algunos de los elementos pertenecientes al álgebra dinámica $(\mathbf{N}_+ \to A)$. Dado que, atendiendo a los modelos establecidos por las definiciones 2.2, 2.4 y 2.6, tanto la especificación como la implementación de un circuito digital denotan una función en dicha álgebra, cabría preguntarse por qué no adoptar como mecanismo de especificación, tal como hacen algunos sistemas de síntesis formal [BlEi97], el propio λ-cálculo junto con algunas adiciones sintácticas que simplifiquen su uso.

Así, el comportamiento de cualquier sistema combinacional podría definirse en base a su comportamiento estático tal como se hizo en la definición 2.22 para los operadores no temporales. Para especificar sistemas secuenciales, aquéllos que para calcular el resultado en cierto instante necesitan conocer instantes pasados de los argumentos, sería necesario utilizar el llamado operador punto fijo *fix*†, que permite definir funciones recursivas anónimas o, lo que es lo mismo, sistemas realimentados.

EJEMPLO 2.11

Mostremos algunos ejemplos de descripción de sistemas digitales:

- Un multiplexor 2 a 1 polimórfico, es decir, done el tipo concreto de los puertos de datos, aún siendo el mismo, es irrelevante. Para expresar esto se usa la variable de tipo *'a*:

  $mux2a1 = \lambda(\ x_1,\ x_0,\ sel\ ) :(\mathbf{N}_+\rightarrow 'a)\times(\mathbf{N}_+\rightarrow 'a)\times(\mathbf{N}_+\rightarrow \mathrm{Bit}).$

  $(\ \lambda t{:}\mathbf{N}_+.\ if\ sel(t){=}0\ then\ x_0(t)\ else\ x_1(t)\ :'a\ ) :(\mathbf{N}_+\rightarrow 'a)$

- Un retardador polimórfico:

  $ret = \lambda(\ init,\ x\ ) :(\mathbf{N}_+\rightarrow 'a)\times(\mathbf{N}_+\rightarrow 'a).$

  $(\ \lambda t :\mathbf{N}_+.\ if\ t{=}1\ then\ init(1)\ else\ x(t{-}1)\ :'a\ ) :(\mathbf{N}_+\rightarrow 'a)$

- Un registro polimórfico con carga síncrona activa en alta:

  $reg = \lambda(\ init,\ load,\ x\ ) :(\mathbf{N}_+\rightarrow 'a)\times(\mathbf{N}_+\rightarrow \mathrm{Bit})\times(\mathbf{N}_+\rightarrow 'a).$

  $fix(\ \lambda z :(\mathbf{N}_+\rightarrow 'a).\ (\ \lambda t :\mathbf{N}_+.\ if\ t{=}1\ then\ init(1)\ else$

  $if\ load(t{-}1) = 1\ then\ x(t{-}1)\ else\ z(t{-}1)\ :'a\ ) :(\mathbf{N}_+\rightarrow 'a)\ ) :(\mathbf{N}_+\rightarrow 'a)$

- El mismo registro pero descrito mediante la composición recursiva (realimentación) de un multiplexor y un retardador:

  $reg = \lambda(\ init,\ load,\ x\ ) :(\mathbf{N}_+\rightarrow 'a)\times(\mathbf{N}_+\rightarrow \mathrm{Bit})\times(\mathbf{N}_+\rightarrow 'a).$

  $fix(\ \lambda z :(\mathbf{N}_+\rightarrow 'a).\ ret(\ init,\ mux2a1(\ x,\ z,\ load\ )\ ) :(\mathbf{N}_+\rightarrow 'a)\ ) :(\mathbf{N}_+\rightarrow 'a)$

- Un multiplicador-acumulador:

  $mac = \lambda(\ x,\ y\ ) :(\mathbf{N}_+\rightarrow 'a)\times(\mathbf{N}_+\rightarrow 'a).$

† Para una discusión detallada sobre dicho operador véase §B.2.

$$fix(\ \lambda z :(\mathbf{N}_+ \rightarrow 'a).\ ret(\ 0,\ (\ \lambda t :\mathbf{N}_+.\ z(t)+x(t)*y(t) :'a\ )\ ) :(\mathbf{N}_+ \rightarrow 'a)\ ) :(\mathbf{N}_+ \rightarrow 'a)$$

Este formalismo poseería un conjunto de propiedades que podrían colocarlo en ventaja frente a otros lenguajes utilizados para especificar circuitos: es versátil, es abstracto, es formal, es fácilmente manipulable e incluso, por ser quizás el sistema de reescritura más estudiado a lo largo de la historia, es 'estándar'.

Sin embargo, el propósito de esta discusión no es proponer la adopción de una λ-notación como mecanismo de especificación, sino que se comprenda la filosofía que ha inspirado, en particular el formalismo de especificación propuesto y, en general la investigación cuya memoria aquí se presenta: la sencillez. Todo un conjunto de propiedades no son suficientes si el formalismo que las sustenta no es sencillo para el usuario que lo utiliza y para la herramienta que lo manipula.

Así que esta sección propondrá un mecanismo de especificación [MeHe97][MeHe98] que, pareciéndose a los mecanismos naturales de especificación, oculte la mayor cantidad de aspectos complicados de su semántica formal. Las claves de este nuevo lenguaje serán dos: una aceptación implícita de la recursividad y un manejo implícito del tiempo. La primera estará fundamentada por la idea de transparencia referencial: toda aparición de un mismo identificador en un conjunto de ecuaciones (ya sea en el lado izquierdo como en el derecho) siempre denotará el mismo objeto. La segunda estará supeditada a la posibilidad de encontrar un único operador temporal básico tal que, componiéndolo con él mismo y con operadores combinacionales, permita describir cualquier sistema secuencial sin necesidad de índices temporales.

La existencia de dicho operador, que llamaremos *fby* (léase *followed by*) y cuyo comportamiento se muestra en la fig. 2.7, está asegurada desde varios

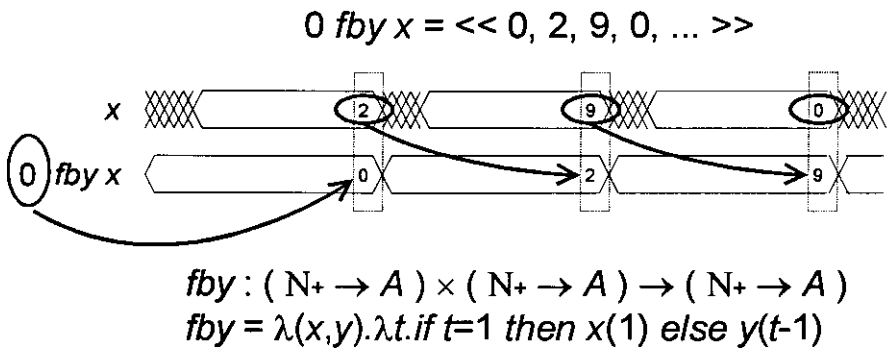


Fig. 2.7: Comportamiento del operador temporal fby.

puntos de vista. Desde el punto de vista matemático cualquier secuencia infinita de valores que caracterice el comportamiento de una señal, puede definirse por recurrencia, o sea, a partir de la concatenación de un valor y otra secuencia infinita. Desde el punto de vista del diseñador de hardware cualquier sistema secuencial se implementa finalmente mediante retardadores (que toman cierto valor inicial) y elementos combinacionales. Obsérvese el símil de una definición por recurrencia y el uso de un retardador: si una señal modelada como una secuencia infinita de valores llega a la entrada de un retardador, la salida de éste será el resultado de concatenar el valor inicial del retardador a la secuencia de entrada.

EJEMPLO 2.12

Asumiendo que todo identificador denota una secuencia infinita de valores pertenecientes a cierto dominio genérico '*a*, que todo símbolo de operación (exceptuando *fby*) denota una función combinacional y que todo identificador libre representa a un puerto de entrada, obsérvese la simplificación notacional que puede alcanzarse respecto al ejemplo 2.11:

- *mux2a1 = if sel=0 then* x_0 *else* x_1
- *ret = init fby x*
- *reg = init fby (if load=1 then x else reg)*

- $mac = 0$ *fby* ($mac + x*y$)

Como puede observarse si se sustituye la recursividad anónima por explícita y se eliminan las referencias temporales, sólo permanece el fragmento 'relevante' de cada una de las λ-expresiones.

Sintaxis y notación.

Formalicemos, primero, el soporte sintáctico de los términos que permitirán definir funciones: partiendo de una signatura dada, se extiende para soportar indefinición y se añade un símbolo que denotará a la función secuencial básica.

2.24 DEFINICIÓN. Sea (S, Σ) una signatura heterogénea. Se define $Lu(\Sigma)$† como la signatura (S, Σ') tal que Σ' es la familia $S^*\times S$-indexada de conjuntos que verifican para todo género s:

- $\Sigma'_{w,s} \equiv \Sigma^{\#}_{w,s}$ si $w \neq s.s$
- $\Sigma'_{s.s,s} = \Sigma^{\#}_{s.s,s} \cup \{ \text{fby} \}$

Obsérvese como *fby* es un símbolo sobrecargado ya que se define para cada uno de los géneros de la signatura de partida. Con ello se consigue que a efectos prácticos pueda ser considerado polimórfico, aunque en realidad cada aparición tenga un género concreto definido por sus argumentos.

A continuación, se formaliza la sintaxis abstracta de la especificación ecuacional.

---

† En consideración al lenguaje dataflow Lucid, que utiliza una extensión similar.

2.25 DEFINICIÓN. Se define **especificación ecuacional de un sistema digital**† como la tupla (Σ, X, *Ins*, *Outs*, φ), donde Σ es una signatura, X es una familia S-indexada de conjuntos de variables disjuntos dos a dos y respecto a Σ, *Ins* y *Outs* son subconjuntos propios de X tales que $Ins \cap Outs = \varnothing$, y φ es una función que proyecta, respetando géneros, elementos de la familia diferencia *X-Ins* sobre el conjunto de términos de la signatura *Lu*-extendida:

$$\varphi : X\text{-}Ins \rightarrow T_{Lu(\Sigma)}(X\text{-}Outs) \mid x \in X_s\text{-}Ins_s \Leftrightarrow \varphi(x) \in T_{Lu(\Sigma),s}(X_s\text{-}Outs_s)$$

Llamaremos **cuerpo** de la especificación a la función φ. Llamaremos **señal** a todo $x \in X$. Llamaremos **puerto de entrada** a todo $x \in Ins$. Llamaremos **puerto de salida** a todo $x \in Outs$. A cada par (x, $\varphi(x)$) perteneciente al cuerpo de la especificación lo llamaremos **definición**.

Obsérvese como la signatura Σ es la que permite resolver los símbolos del cuerpo ecuacional. A todos los efectos funciona como las bibliotecas de diseño de cualquier sistema de síntesis ya que éstas permiten dar significado a los componentes con las que se construyen las conductas (de hecho esta idea se utilizará en el capítulo 6 para formalizarlas). Desde un punto de vista pragmático, no es estrictamente necesario que sea el propio diseñador el que siempre defina Σ, puede existir un conjunto de signaturas predefinidas entre las cuales el diseñador únicamente tenga que elegir una.

Por conveniencia, toda definición será representada por una ecuación de la forma $x = \varphi(x)$, de modo que el cuerpo de la especificación estará formado por un conjunto finito de ecuaciones:

$$\varphi \equiv \{ x_1 = t_1, \ldots, x_n = t_n \} \text{ siendo } t_i \equiv \varphi(x_i)$$

Además, en los ejemplos, mientras que las señales se listarán tras la palabra reservada **signals** usando la notación $x : s$ para indicar el género de cada señal, los puertos de entrada, los de salida y el conjunto de definiciones se agruparán respectivamente tras las palabras clave **inports**, **outports** y **body**.

† Esta definición será ampliada en el capítulo 4.

EJEMPLO 2.13

Especifiquemos ecuacionalmente un filtro recursivo de segundo orden:

sorts
Ent
operations
0, a1, a2, b1, b2 : → Ent
□ + □, □ - □, □ * □ : Ent, Ent → Ent
signals
out, in, z : Ent
inports
in
outports
out
body
out = z - (a1 *(0 fby z) + a2 *(0 fby 0 fby z))
z = (b1 *(0 fby z) + b2 *(0 fby 0 fby z)) + in

Como puede observarse, hemos definido mediante la signatura tanto todos los posibles tipos de datos que el circuito manipula (1 sólo en este caso), como todos los símbolos que se utilizan para definir el comportamiento del sistema (a excepción del símbolo *fby* que está definido implícitamente por la *Lu*-extensión de la signatura). En dicha signatura hemos definido símbolos para que puedan denotar constantes concretas (0), constantes genéricas (a1, a2, b1 y b2) y operaciones combinacionales (+, - y *). La denotación concreta, por el momento, no se especifica† y puede ser cualquier $Lu(\Sigma)$-álgebra que esté en la mente del diseñador.

También hemos definido tres señales del mismo género de las cuales una es un puerto de entrada, por lo que no está definida, y otras dos, que si lo están, representan un puerto de salida y una señal intermedia.

Más formalmente si ((S, Σ), X, *Ins*, *Outs*, φ) es la anterior especificación, cada una de las componentes de la tupla han sido definidas como:

- $S \equiv \{ \text{Ent} \}$
- $\Sigma \equiv \{ \Sigma_{\varepsilon,\, \text{Ent}} \equiv \{ 0, a1, a2, b1, b2 \},\ \Sigma_{\text{Ent.Ent},\, \text{Ent}} \equiv \{ +, -, * \} \}$

† En el capítulo 6 se definirá un método para hacerlo.

- $X \equiv \{ X_{Ent} \equiv \{ \text{out, in, z} \} \}$
- $Ins \equiv \{ Ins_{Ent} \equiv \{ \text{in} \} \}$
- $Outs \equiv \{ Outs_{Ent} \equiv \{ \text{out} \} \}$
- $\varphi : \{\text{out,z}\} \rightarrow T_{Lu(\Sigma)}(\{\text{in,z}\})$ definida como:

  $\varphi(\text{ out }) \equiv \text{z} - (\text{ a1} * (0 \text{ fby z}) + \text{a2} * (0 \text{ fby } 0 \text{ fby z}))$

  $\varphi(\text{ z }) \equiv (\text{ b1} * (0 \text{ fby z}) + \text{b2} * (0 \text{ fby } 0 \text{ fby z})) + \text{in}$

Semántica.

En primer lugar se dará soporte semántico a los $Lu(\Sigma)$-términos: partiendo de una Σ-álgebra estática, sucesivamente se extiende para denotar indiferencia, se extiende para denotar indefinición y se extiende temporalmente.

2.26 Definición. Sea $Lu(\Sigma)$ la signatura definida a partir de (S, Σ), y sea A una Σ-álgebra. Se define $Lu(A)$ como la $Lu(\Sigma)$-álgebra que cumple:

- El universo de $Lu(A)$ es $(\mathbf{N}_+ \rightarrow (A^{\#})_{\perp})$
- Cualquier símbolo de operación $\sigma \in \Sigma_{w,s}$ que denota la función $A_{\sigma}^{w,s}$, permite definir la denotación del mismo símbolo como la función:

  $Lu(A)_{\sigma}^{w,s} = (\mathbf{N}_+ \rightarrow (A^{\#})_{\perp})_{\sigma}^{w,s}$

  o lo que es lo mismo†:

  $Lu(A)_{\sigma}^{w,s} = \lambda(x_1, ..., x_n) : Lu(A_{s1}) \times ... \times Lu(A_{sn}).$

  $(\lambda t : \mathbf{N}_+ . ((A^{\#})_{\perp})_{\sigma}^{w,s}(x_1(t), ..., x_n(t)) : (A_s^{\#})_{\perp}) : Lu(A_s)$
- Para todo género s, el símbolo *fby* denota a la función $Lu(A)_{\text{fby}}^{s,s,s}$, que se define como:

  $Lu(A)_{\text{fby}}^{s,s,s} = \lambda(x,y) : (\mathbf{N}_+ \rightarrow (A_s^{\#})_{\perp}) \times (\mathbf{N}_+ \rightarrow (A_s^{\#})_{\perp}).$

  $(\lambda t : \mathbf{N}_+ . \textit{ if } t{=}1 \textit{ then } x(1) \textit{ else } y(t\text{-}1) : (A_s^{\#})_{\perp}) : (\mathbf{N}_+ \rightarrow (A_s^{\#})_{\perp})$

---

† Véase la definición 2.22.

En §2.4.1 se definió la noción de interpretación con la que era posible poner un único término en relación con el valor que denota en cierta álgebra. Sin embargo, dicha noción no es capaz de relacionar un conjunto de términos, interpretarlos como mutuamente recursivos y obtener la función entre álgebras dinámicas que denota (la que se intenta especificar mediante una especificación ecuacional y que responde a los modelos definidos en §2.1).

A continuación se definirá una función semántica **C** (de circuito), que a toda especificación (sintaxis) le asocia su correspondiente comportamiento (semántica). Esta utiliza una familia de funciones auxiliares **E** (de expresión) que para cada género asocia términos con comportamientos.

2.27 **DEFINICIÓN.** Sea la especificación ecuacional $(\Sigma, X, Ins, Outs, \varphi)$, se define su semántica como el resultado de la función **C** que se define como:

$$\mathbf{C} : (\Sigma, X, Ins, Outs, \varphi) \to (Lu(A)_{s1} \times \overset{p}{\ldots} \times Lu(A)_{sp} \to Lu(A)_{t1} \times \overset{q}{\ldots} \times Lu(A)_{tq})$$

$$\mathbf{C}[\![(\Sigma, (in_1, \overset{p}{\ldots}, in_p, out_1, \overset{q}{\ldots}, out_q, z_1, \overset{m}{\ldots}, z_m), (in_1, \overset{p}{\ldots}, in_p), (out_1, \overset{q}{\ldots}, out_q), \varphi)]\!] =$$

$$= \lambda(in_1, \overset{p}{\ldots}, in_p) : Lu(A)_{s1} \times \overset{p}{\ldots} \times Lu(A)_{sp} .$$

$$(\mathrm{fix}(\lambda(out_1, \overset{q}{\ldots}, out_q, z_1, \overset{m}{\ldots}, z_m) : Lu(A)_{t1} \times \overset{q}{\ldots} \times Lu(A)_{tq} \times Lu(A)_{u1} \times \overset{m}{\ldots} \times Lu(A)_{um} .$$

$$(\mathbf{E}[\![\varphi(out_1)]\!], \overset{q}{\ldots}, \mathbf{E}[\![\varphi(out_q)]\!], \mathbf{E}[\![\varphi(z_1)]\!], \overset{m}{\ldots}, \mathbf{E}[\![\varphi(z_m)]\!])$$

$$: Lu(A)_{t1} \times \overset{q}{\ldots} \times Lu(A)_{tq} \times Lu(A)_{u1} \times \overset{m}{\ldots} \times Lu(A)_{um}$$

$$) \downarrow 1..q) : Lu(A)_{t1} \times \overset{q}{\ldots} \times Lu(A)_{tq}$$

donde $in_i \in Ins$, $out_i \in Outs$, $z_i \in X\text{-}Ins\text{-}Outs$, p es el número de puertos de entrada, q el número de puertos de salida, m el número de señales auxiliares, A es cierta Σ-álgebra, *fix* es el operador punto fijo, $\downarrow$ es el operador de restricción de tuplas, definido como $(x_1,\ldots,x_n) \downarrow a..b = (x_a,\ldots,x_b)$ con $1 \le a \le b \le n$, y **E** se define como:

$$\mathbf{E} : T_{Lu(\Sigma),s}(X) \to Lu(A)_s$$

$$\mathbf{E}[\![\sigma(e_1, \ldots, e_n)]\!] = Lu(A)^{w,s}_{\sigma}(\mathbf{E}[\![e_1]\!](t), \ldots, \mathbf{E}[\![e_n]\!](t)), \quad \forall \sigma \in Lu(\Sigma)_{w,s}$$

$$\mathbf{E}[\![x]\!] = \lambda t{:}\mathbb{N}_+ .\ x(t) : (A_s^{\#})_{\perp}, \quad \forall x \in X_s$$

Obsérvese que bajo la primera definición de **E** se da semántica a los operadores no temporales (ya que por la extensión temporal $\Sigma \subset Lu(\Sigma)$), a las constantes (ya que para todo género *s*, $\Sigma_{\varepsilon,s} \subset \Sigma \subset Lu(\Sigma)$) y al operador temporal *fby* (ya que para algún género *s*, $fby \in Lu(\Sigma)_{s,s,s}$).

EJEMPLO 2.14

Sea la signatura *Enteros* que ofrece soporte sintáctico al álgebra de los números enteros y que se muestra a continuación:

Enteros ≡
sorts
Ent
operations
0 : → Ent
□ + □ : Ent, Ent → Ent
□ * □ : Ent, Ent → Ent

Una posible especificación ecuacional de un multiplicador-acumulador, *Mac*, puede ser:

Mac ≡
... la signatura Enteros ...
signals
out, x, y, z : Ent
inports
x, y
outports
out
body
out = z
z = 0 fby (z + x*y)

Compárese el cuerpo ecuacional con el ejemplo 2.12 para comprobar su similitud. Calculemos la función que denota, que deberá ser también muy similar a la mostrada en el ejemplo 2.11 (si en él reemplazamos la aparición de *ret* por su correspondiente definición).

C[*Mac*] =
= $\lambda(x, y) : Lu(Z)_{Ent} \times Lu(Z)_{Ent} \cdot$
(*fix*($\lambda(out, z) : Lu(Z)_{Ent} \times Lu(Z)_{Ent} \cdot$

$$(\mathbf{E}[\![z]\!], \mathbf{E}[\![0 \text{ fby } (z + x*y)]\!]) : Lu(Z)_{Ent} \times Lu(Z)_{Ent}$$
$$)\downarrow 1..1) : Lu(Z)_{Ent}$$

si calculamos, por ejemplo, la denotación de la segunda de las definiciones:

$$\mathbf{E}[\![0 \text{ fby } (z + x*y)]\!] = \ldots =$$
$$= \lambda t : \mathbf{N}_+ .\ \textit{if } t{=}1 \textit{ then } \mathbf{E}[\![0]\!](1) \textit{ else } \mathbf{E}[\![z + x*y]\!](t\text{-}1) : (Z_{Ent}^{\#})_{\perp} = \ldots =$$
$$= \lambda t : \mathbf{N}_+ .\ \textit{if } t{=}1 \textit{ then } ((Z^{\#})_{\perp})_0^{\varepsilon, Ent} \textit{ else}$$
$$((Z^{\#})_{\perp})_+^{Ent.Ent,\ Ent}(\mathbf{E}[\![z]\!](t\text{-}1),\ \mathbf{E}[\![x*y]\!](t\text{-}1)) : (Z_{Ent}^{\#})_{\perp} = \ldots =$$
$$= \lambda t : \mathbf{N}_+ .\ \textit{if } t{=}1 \textit{ then } (Z^{\#})_{\perp})_0^{\varepsilon, Ent} \textit{ else}$$
$$((Z^{\#})_{\perp})_+^{Ent.Ent,\ Ent}(z(t\text{-}1),\ ((Z^{\#})_{\perp})_*^{Ent.Ent,\ Ent}(x(t\text{-}1),\ y(t\text{-}1))) : (Z_{Ent}^{\#})_{\perp}$$

de manera que si la substituimos, y abreviamos la representación de las funciones estáticas:

$$\mathbf{C}[\![\textit{Mac}]\!] =$$
$$= \lambda(x, y) : Lu(Z)_{Ent} \times Lu(Z)_{Ent} \cdot$$
$$(\textit{fix}(\lambda(\textit{out}, z) : Lu(Z)_{Ent} \times Lu(Z)_{Ent} \cdot$$
$$(\lambda t : \mathbf{N}_+ .\ z(t) : (Z_{Ent}^{\#})_{\perp}, \lambda t : \mathbf{N}_+ .\ \textit{if } t{=}1 \textit{ then } 0 \textit{ else } z(t\text{-}1) + x(t\text{-}1) * y(t\text{-}1) : (Z_{Ent}^{\#})_{\perp}$$
$$) : Lu(Z)_{Ent} \times Lu(Z)_{Ent}$$
$$)\downarrow 1) : Lu(Z)_{Ent}$$

Sea, ahora, una especificación ecuacional incorrecta de un multiplicador-acumulador, *Mac'*, en donde al eliminar el operador *fby* de la definición de *z*, aparece un lazo combinacional:

Mac' ≡
... la signatura Enteros ...
signals
out, x, y, z : Ent
inports
x, y
outports
out
body
out = z
z = z + x*y

Esta especificación denota a la función:

$$\mathbf{C}[\![\textit{Mac'}]\!] =$$

$= \lambda(\, x,\, y\,) : Lu(Z)_{Ent} \times Lu(Z)_{Ent} \cdot$

$(\, fix(\, \lambda(\, out,\, z\,) : Lu(Z)_{Ent} \times Lu(Z)_{Ent} \cdot$

$(\, \lambda t : \mathbf{N}_+ .\ z(t) : (Z_{Ent}^{\#})_\perp,\ \lambda t : \mathbf{N}_+ .\ z(t) + x(t) * y(t) : (Z_{Ent}^{\#})_\perp$

$) : Lu(Z)_{Ent} \times Lu(Z)_{Ent}$

$)\downarrow 1\,) : Lu(Z)_{Ent}$

Utilicemos la definición B.28 del operador punto fijo para una representación más clara de la función que está denotando. Recordemos que este operador aplicado a una función se definía como el límite de una sucesión de funciones cada vez más definidas, obtenidas mediante la aplicación sucesiva de la función argumento a la anterior aproximación†. Sea *f* la función que toma como argumento el operador *fix* incluido en la denotación de *Mac'*.

- $f_0 = (\, \lambda t : \mathbf{N}_+ .\ \perp : (Z_{Ent}^{\#})_\perp,\ \lambda t : \mathbf{N}_+ .\ \perp : (Z_{Ent}^{\#})_\perp\,)$
- $f_1 = f(\, f_0\,) = \ldots =$
  $= (\, \lambda t : \mathbf{N}_+ .\ \perp : (Z_{Ent}^{\#})_\perp,\ \lambda t : \mathbf{N}_+ .\ \perp + x(t) * y(t) : (Z_{Ent}^{\#})_\perp\,) =$
  $= (\, \lambda t : \mathbf{N}_+ .\ \perp : (Z_{Ent}^{\#})_\perp,\ \lambda t : \mathbf{N}_+ .\ \perp : (Z_{Ent}^{\#})_\perp\,) \equiv$
  $\equiv f_0$
- $f_2 = f(\, f(\, f_0\,)\,) = \ldots =$
  $= (\, \lambda t : \mathbf{N}_+ .\ \perp : (Z_{Ent}^{\#})_\perp,\ \lambda t : \mathbf{N}_+ .\ \perp : (Z_{Ent}^{\#})_\perp\,) \equiv$
  $\equiv f_0$

Claramente el supremo de la cadena de funciones $(\, f_0,\, f_1,\, f_2,\, \ldots\,)$ es la propia función f_0, lo que permite concluir aplicando el operador restricción que:

$$\mathbf{C}[\![\, Mac' \,]\!] = \lambda(\, x,\, y\,) : Lu(Z)_{Ent} \times Lu(Z)_{Ent} \cdot (\, \lambda t : \mathbf{N}_+ .\ \perp : (Z_{Ent}^{\#})_\perp\,) : Lu(Z)_{Ent}$$

O sea, que la semántica de este multiplicador-acumulador mal especificado es la función de dos argumentos que para todo instante temporal está indefinida. Semántica que está en concordancia con el significado que dimos

† Véase el ejemplo B.2.

al símbolo fondo en §2.4.2 y con idea de un circuito que está inestable en todo momento.

Para finalizar debe destacarse que toda función especificada ecuacionalmente es necesariamente causal y tiene memoria finita. Su comprobación es simple. Es causal por ser composición de operadores causales (ya que los no-temporales y el *fby* lo son) y tiene memoria finita ya que el único elemento semántico con memoria es el operador *fby* y éste sólo podrá aparecer un número finito de veces en una especificación ecuacional por ser ésta un mecanismo finito.

Principio de *Lu*-extensión.

En la definición de la sintaxis y semántica del mecanismo de especificación ecuacional se han definido sendos procesos de extensión: tanto de signaturas como de álgebras. El propósito perseguido era, mediante la sobrecarga de símbolos, ocultar los aspectos complicados de las *Lu*-álgebras para que el diseñador pudiera manejar conceptos dinámicos con la misma comodidad con la que pueden manipularse los conceptos estáticos. Sin embargo, siempre que existe ocultación en un formalismo, se corre el peligro de que el que lo usa pueda incurrir en errores conceptuales difíciles de asimilar. Así en este caso, y al contrario de lo que sucedía en la extensión temporal, no todas las ecuaciones válidas en un álgebra estática son válidas en su correspondiente *Lu*-extensión.

EJEMPLO 2.15

Asumiendo la interpretación habitual de los símbolos de operación, la ecuación $x*0 = 0$ es válida en el álgebra de los enteros Z pero no lo es en

Lu(Z). Para comprobarlo basta con considerar la valoración μ que asigna $\lambda t.\#$ a x, e interpretar ambos términos de la ecuación. Por un lado, según la definición estricta de la multiplicación extendida a partir de la multiplicación entera (véase definición 2.18), $\hat{\mu}(x * 0) = \lambda t.\#$. Por otro lado $\hat{\mu}(0) = \lambda t.0$.

Por ello, para evitar razonamientos incorrectos, a continuación establezco una razón sintáctica (y por tanto simple de chequear) que sea necesaria y suficiente para poder asegurar cuándo una ecuación válida en el modelo estático es válida en el modelo *Lu*-extendido y que he denominado **principio de *Lu*-extensión**.

2.28 PROPOSICIÓN. Si una Σ-álgebra A, satisface una Σ-ecuación $e \equiv (X, t_L, t_R)$, entonces la $Lu(\Sigma)$-álgebra $Lu(A)$ también la satisface[†] si y sólo si $var(t_R) = var(t_L)$. Donde si t es un término, $var(t)$ es el conjunto de variables incluidas en el t (véase la definición 3.1 para una definición más formal).

DEMOSTRACIÓN. Según el principio de extensión demostrado en la proposición 2.23 y según la definición de $Lu(A)$, para demostrar este enunciado basta con comprobar que:

$$var(t_R) = var(t_L) \Leftrightarrow (A \models e \Rightarrow (A^{\#})_{\perp} \models e)$$

($\Rightarrow$) Según la definición 2.13, que un álgebra satisfaga una ecuación es equivalente a que para toda valoración de variables, la interpretación de ambos términos de la ecuación sea la misma. Así sea la valoración genérica $\mu: X \to (A^{\#})_{\perp}$. Por un lado, si μ es tal que $\forall x \in X$, $\mu(x) \in A$, se tiene (en virtud de que $A \models e$) que $\hat{\mu}(t_L) = \hat{\mu}(t_R)$ sin más consideraciones. Por otro lado, si μ es tal que $\exists x_0 \in X$, $\mu(x_0) \equiv \#$ (respectivamente $\mu(x) \equiv \perp$), dado que $t_R, t_L \in T_{\Sigma}(X)$ se tendrá que si $x_0 \in var(t_R) = var(t_L)$ entonces $\hat{\mu}(t_L) = \hat{\mu}(t_R) = \#$ (respectivamente

---

[†] Recuérdese que gracias a la sobrecarga de símbolos que ocurre durante un proceso de *Lu*-extensión, toda Σ-ecuación es también una $Lu(\Sigma)$-ecuación.

$\perp$) por denotar todos los operadores que se incluyen en t_L y t_R funciones estrictas respecto del elemento # (respectivamente $\perp$).

($\Leftarrow$) Sea la valoración genérica $\mu: X \to (A^{\#})_{\perp}$ tal que $\exists x_0 \in X$, $\mu(x_0) \equiv \#$ y $\forall x \in X\text{-}\{x_0\}$, $\mu(x) \in A$. Supongamos que $x_0 \in var(t_L)$ y que $x_0 \notin var(t_R)$, por lo que claramente $var(t_L) \neq var(t_R)$, así $\hat{\mu}(t_R) \in A$ y $\hat{\mu}(t_L) = \# \notin A$ y por consiguiente $(A^{\#})_{\perp}$ no satisface la ecuación *e*.

□

2.4.4 Simulación de especificaciones ecuacionales.

La propia definición de la semántica formal de una especificación ecuacional establece implícitamente un esquema válido de simulación. Este esquema está basado en la interpretación iterativa del operador punto fijo (como consecuencia del teorema B.29) y en la noción de reducción del λ-cálculo (véase §A.3). Así una posible implementación de este algoritmo abstracto podría usar como núcleo operativo, aparte de algún conocimiento sobre la semántica de los operadores, la conversión β (para simplificar expresiones) y la propiedad *fix f = f fix f* (para simular la recursividad). Mostremos mediante un ejemplo la calidad de simulable del mecanismo de especificación propuesto.

EJEMPLO 2.16

Sea la especificación ecuacional de un filtro recursivo de primer orden:

FirstOrder ≡
sorts
Ent
operations
0, a1, b1 : → Ent
□ + □, □ - □, □ * □ : Ent, Ent → Ent
signals
in, out, z : Ent
inports
in

outports
out
body
out = z - (a1 ∗(0 fby z))
z = in + (b1 ∗(0 fby z))

Asumiendo el álgebra soporte habitual y, por simplicidad, utilizando una λ-notación sin tipos, la función que denota la anterior especificación es:

C[*FirstOrder*] = λ(*in*).(*fix*(λ(*out*, z) (

λ*t*. z(*t*) - a1 ∗*if t*=1 *then* 0 *else* z(*t*-1),

λ*t*. *in*(*t*) + b1 ∗*if t*=1 *then* 0 *else* z(*t*-1)))↓1)

Realicemos una **simulación de valores**. Para ello basta con aplicar a la anterior expresión un estímulo de entrada y a la expresión que resulte un instante temporal concreto. Tras reducir la expresión obtendremos como resultado el valor que toma el circuito en el instante y bajo el estímulo dados.

Supongamos que la entrada del circuito está permanentemente estimulada por un 1, esto es la función λ*t*.1. Calculemos el valor que calcula el circuito en el tercer ciclo.

λ*in*.(*fix*(λ(*out*, z). (

λ*t*. z(*t*) - a1 ∗*if t*=1 *then* 0 *else* z(*t*-1),

λ*t*. *in*(*t*) + b1 ∗*if t*=1 *then* 0 *else* z(*t*-1)))↓1) (λ*t*.1) (3) =

= *fix*(λ(*out*, z). (*β conversión*

λ*t*. z(*t*) - a1 ∗*if t*=1 *then* 0 *else* z(*t*-1),

λ*t*. (λ*t*.1)(*t*) + b1 ∗*if t*=1 *then* 0 *else* z(*t*-1)))↓1 (3) =

= *fix*(λ(*out*, z). (*β conversión*

λ*t*. z(*t*) - a1 ∗*if t*=1 *then* 0 *else* z(*t*-1),

λ*t*. 1 + b1 ∗*if t*=1 *then* 0 *else* z(*t*-1)))↓1 (3) =

= (λ(*out*, z). (*propiedad de fix donde f abrevia el*

λ*t*. z(*t*) - a1 ∗*if t*=1 *then* 0 *else* z(*t*-1), *argumento actual del operador*

λ*t*. 1 + b1 ∗*if t*=1 *then* 0 *else* z(*t*-1)) (*fix f*))↓1 (3) =

= (λ*t*. (*fix f*)↓2(*t*) - a1 ∗*if t*=1 *then* 0 *else* (*fix f*)↓2(*t*-1), *β conversión*

λ*t*. 1 + b1 ∗*if t*=1 *then* 0 *else* (*fix f*)↓2(*t*-1))↓1 (3) =

$= (\lambda t.\ (\ fix\ f\ )\downarrow 2(t) - a1 * if\ t{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(t{-}1)\ )(\ 3\ )$ = *semántica de* $\downarrow$

$= (\ fix\ f\ )\downarrow 2(3) - a1 * if\ 3{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(2) =$ *β conversión*

$= (\ fix\ f\ )\downarrow 2(3) - a1 * (\ fix\ f\ )\downarrow 2(2) =$ *semántica de if-then-else*

$= (\ f\ fix\ f\ )\downarrow 2(3) - a1 * (\ f\ fix\ f\ )\downarrow 2(2) =$ *propiedad de fix*

$= (\lambda t.\ 1 + b1 * if\ t{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(t{-}1)\ )(3) -$ *β conversiones*

$\quad - a1 * (\lambda t.\ 1 + b1 * if\ t{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(t{-}1)\ )(2) =$

$= 1 + b1 * if(\ 3{=}1\ then\ 0\ else\ (fix\ f)\downarrow 2(2)\ ) -$ *β conversiones*

$\quad - a1 * (\ 1 + b1 * if(\ 2{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(1)\ )\ ) =$

$= 1 + b1 * (\ fix\ f\ )\downarrow 2(2) - a1 * (\ 1 + b1 * (\ fix\ f\ )\downarrow 2(1)\ ) =$ *semántica de if-then-else*

$= 1 + b1 * (\ f\ fix\ f\ )\downarrow 2(2) - a1 * (\ 1 + b1 * (\ f\ fix\ f\ )\downarrow 2(1)\ ) =$ *propiedad de fix*

$= 1 + b1 * (\lambda t.\ 1 + b1 * if\ t{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(t{-}1)\ )(2) -$ *β conversiones*

$\quad - a1 * (\ 1 + b1 * (\lambda t.\ 1 + b1 * if\ t{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(t{-}1)\ )(1)\ ) =$

$= 1 + b1 * (\ 1 + b1 * if\ 2{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(1)\ ) -$ *β conversiones*

$\quad - a1 * (\ 1 + b1 * (\ 1 + b1 * if\ 1{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(0)\ )\ ) =$

$= 1 + b1 * (\ 1 + b1 * (\ fix\ f\ )\downarrow 2(1)\ ) - a1 * (\ 1 + b1 * (\ 1 + b1 * 0\ )\ ) =$ *semántica de if*

$= 1 + b1 * (\ 1 + b1 * (\ f\ fix\ f\ )\downarrow 2(1)\ ) - a1 * (1 + b1) =$ *semántica de* $*$ *y propiedad de fix*

$= 1 + b1 * (\ 1 + b1 * (\lambda t.\ 1 + b1 * if\ t{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(t{-}1)\ )(1)\ ) -$

$\quad - a1 * (\ 1 + b1\ ) =$ *β conversión*

$= 1 + b1 * (\ 1 + b1 * (\ 1 + b1 * if\ 1{=}1\ then\ 0\ else\ (\ fix\ f\ )\downarrow 2(0)\ )\ ) -$

$\quad - a1 * (\ 1 + b1\ ) =$ *β conversión*

$= 1 + b1 * (\ 1 + b1 * (\ 1 + b1 * 0\ )\ ) - a1 * (\ 1 + b1\ ) =$ *semántica de if-then-else*

$= 1 + b1 * (\ 1 + b1\ ) - a1 * (\ 1 + b1\ )$ *semántica de* $*$

Si bien la idea utilizada por el esquema de simulación mostrado en el ejemplo anterior es válida, requiere de un soporte simbólico importante. Por ello, aunque el objetivo de la investigación cuya memoria aquí se presenta no sea el desarrollo de simuladores, cabe preguntarse si podría existir algún algoritmo de simulación más simple y potencialmente menos costoso.

Pues bien, también la semántica formal de la especificación ecuacional (y más concretamente la semántica formal de las definiciones) permite considerar un esquema de simulación dirigido por demanda. A diferencia de las simulaciones dirigidas por eventos, en donde el cambio de los estímulos es el que dispara un árbol de cálculos, en una simulación dirigida por demanda, son los valores requeridos los que disparan dicho árbol. Así, si se desea calcular un valor de cierta señal en cierto instante, el simulador dirigido por demanda recurre a la definición de la señal y evalúa los subtérminos que la componen en otros instantes lo que dispara a su vez nuevas demandas. Cuando los valores demandados se conocen se vuelve hacia atrás operando con ellos hasta que se obtenga el valor inicialmente solicitado.

EJEMPLO 2.17

Considérese nuevamente la especificación ecuacional del filtro de primer orden presentado en el ejemplo 2.16. Para realizar una simulación dirigida por demanda es necesario asumir la semántica recursiva que posee toda especificación ecuacional y tener en cuenta la semántica de cada una de las definiciones que forman el cuerpo de la misma:

$\mathbf{E}[\ \varphi(\text{out})\ ] = \lambda t.\ z(t) - a1 * \textit{if}\ t{=}1\ \textit{then}\ 0\ \textit{else}\ z(t{-}1)$

$\mathbf{E}[\ \varphi(\text{z})\ ] = \lambda t.\ in(t) + b1 * \textit{if}\ t{=}1\ \textit{then}\ 0\ \textit{else}\ z(t{-}1)$

Simulemos el valor del puerto *out* en el instante 3 si la entrada del circuito está permanentemente estimulada por un 1. O sea se crea la primera demanda, *out*(3):

$\mathbf{E}[\ \varphi(\text{out})\ ](3) \equiv (\ \lambda t.\ z(t) - a1 * \textit{if}\ t{=}1\ \textit{then}\ 0\ \textit{else}\ z(t{-}1)\ )\ (3) =$

$= z(3) - a1 * \textit{if}\ 3{=}1\ \textit{then}\ 0\ \textit{else}\ z(2) =$

$= z(3) - a1 * z(2)$ *se demanda el cálculo de z(3) y z(2)*

$\mathbf{E}[\ \varphi(\text{z})\ ](3) \equiv (\ \lambda t.\ in(t) + b1 * \textit{if}\ t{=}1\ \textit{then}\ 0\ \textit{else}\ z(t{-}1)\ )\ (3) =$

$= in(3) + b1 * \textit{if}\ 3{=}1\ \textit{then}\ 0\ \textit{else}\ z(2) =$

$= in(3) + b1 * z(2)$ *se demanda el cálculo de in(3) y z(2)*

$in(3) \equiv (\ \lambda t.1\ )\ (3) = 1$ *se obtiene el valor de in(3) = 1*

$\mathbf{E}[\,\varphi(z)\,](2) \equiv (\,\lambda t.\ in(t) + b1 * \mathit{if}\ t{=}1\ \mathit{then}\ 0\ \mathit{else}\ z(t\text{-}1)\,)\,(2) =$

$= in(2) + b1 * \mathit{if}\ 2{=}1\ \mathit{then}\ 0\ \mathit{else}\ z(1) =$

$= in(2) + b1 * z(1)$ *se demanda el cálculo de in(2) y z(1)*

$in(2) \equiv (\,\lambda t.1\,)\,(2) = 1$ *se obtiene el valor de in(2) = 1*

$\mathbf{E}[\,\varphi(z)\,](1) \equiv (\,\lambda t.\ in(t) + b1 * \mathit{if}\ t{=}1\ \mathit{then}\ 0\ \mathit{else}\ z(t\text{-}1)\,)\,(1) =$

$= in(1) + b1 * \mathit{if}\ 1{=}1\ \mathit{then}\ 0\ \mathit{else}\ z(0) =$

$= in(1) + b1 * 0 =$

$= in(1)$ *se demanda el cálculo de in(1)*

$in(1) \equiv (\,\lambda t.1\,)\,(1) = 1$ *se obtiene el valor de in(1) = 1*

$= 1$ *se obtiene el valor de z(1) = 1*

$= 1 + b1 * 1 =$

$= 1 + b1$ *se obtiene el valor de z(2) = 1+b1*

$= 1 + b1 * (\,1 + b1\,)$ *se obtiene el valor de z(3) = 1+b1*(1+b1)*

$\mathbf{E}[\,\varphi(z)\,](2) = \ldots. =$ *esta es la demanda pendiente de z(2) hecha por out(3)*

$= 1 + b1$ *se vuelve a obtener el valor de z(2) = 1+b1*

$= 1 + b1 * (\,1 + b1\,) - a1 * (\,1 + b1\,)$ *se obtiene el valor de out(3)*

A continuación se define un algoritmo secuencial que es capaz de reproducir el comportamiento mostrado en el ejemplo (no es necesario que sea secuencial ya que cada demanda puede procesarse por separado).

Un algoritmo para la simulación de valores de especificaciones ecuacionales.

Se define la función *simular* que recursivamente devuelve la interpretación de cierto término en cierto instante temporal. Para ello toma como argumentos una especificación ecuacional $ds \equiv (\,\Sigma, X, Ins, Outs, \varphi\,)$, una valoración *estimulo* : $Ins \rightarrow Lu(A)$, donde $Lu(A)$ es una cierta $Lu(\Sigma)$-álgebra y que define los valores que toman los puertos de entrada, el $termino \in T_{Lu(\Sigma)}(X)$ que

queremos simular (recuérdese una señal es también un término) y el *ciclo*$\in \mathbb{N}_+$ en que queremos simular.

simular(*ds*, *estimulo*, *termino*, *ciclo*)
inicio
  si *termino* **es**:
    $x \in Ins$ **entonces devolver** *estimulo* (*x*) (*ciclo*)
    $x \in X\text{-}Ins$ **entonces devolver** simular(*ds*, *estimulo*, $\varphi(x)$, *ciclo*)
    σ **entonces devolver** $((A^{\#})_{\perp})^{\varepsilon,s}_{\sigma}$
    $\sigma(t_1, ..., t_n)$, con $\sigma \in \Sigma$ **entonces**
      para cada t_i **hacer** $valor_i$ = simular(*ds*, *estimulo*, t_i, *ciclo*)
      devolver $((A^{\#})_{\perp})^{w,s}_{\sigma}(valor_1, ..., valor_n)$
    t_1 fby t_2 **entonces**
      si *ciclo* = 1 **entonces devolver** simular(*ds*, *estimulo*, t_1, 1)
        sino devolver simular(*ds*, *estimulo*, t_2, *ciclo*-1)
fin

El algoritmo, tal como se ha descrito, es inaceptable desde un punto de vista práctico. El problema es que los valores calculados no se almacenan, por lo que deberán ser recomputados cada vez que sean demandados. Sin embargo, esta manifiesta ineficiencia puede ser resuelta si se mantiene un almacén de tuplas *señal-valor-ciclo* (o incluso de *subtérmino-valor-ciclo*). Cada vez que se demande el valor de una señal (o subtérmino), el simulador deberá chequear en el almacén por si ya ha sido calculado, si es así, obtendrá el valor inmediatamente, en caso contrario, procederá a calcularlo. De este modo cada valor sólo se calcula una vez. Además para conseguir un uso eficiente de la memoria usada como almacén, debería existir algún mecanismo (basado en antigüedad u otras métricas) que ocasionalmente eliminara tuplas que no fueran a ser necesitadas en el futuro.

Obsérvese cómo un lazo combinacional da lugar a continuas demandas de las mismas señales lo que hace entrar al simulador en un lazo sin final que

corresponde al significado que dimos al elemento fondo (inténtese simular *Mac'* del ejemplo 2.14).

Conceptos de traza, estímulo y simulación de una especificación ecuacional.

Si bien la semántica formal es un concepto claro y compacto que sería válido para establecer equivalencias entre especificaciones y para razonar sobre ellas, es algo complejo para poder trabajar cómodamente con él por ser, como definimos, una función sobre funciones. Por ello vamos a introducir un concepto muy intuitivo que, de modo natural, puede abstraerse de experiencia cotidiana con simuladores de valores: la noción de traza.

Una traza es la colección de secuencias de valores que adoptan cada una de las señales de una especificación ecuacional en una ejecución particular, o sea, un conjunto de funciones temporales. Cada vez que ponemos un circuito en funcionamiento sus señales también siguen una traza particular. Así el comportamiento de un circuito, aparte de como función, puede ser contemplado como el conjunto de todas las trazas que eventualmente podrían caracterizarlo en cualquiera de sus ejecuciones. Esta nueva noción permitirá simplificar las pruebas de equivalencia y compatibilidad de circuitos (que se definirán en el capítulo 3), además de permitir unir elegantemente la idea semántica de comportamiento, con otras tan comunes para un diseñador hardware como lo son las de simulación de valores y estímulo.

2.29 DEFINICIÓN. Sea una especificación ecuacional $ds = (\Sigma, X, Ins, Outs, \varphi)$, y una $Lu(\Sigma)$-álgebra $Lu(A)$. Llamaremos **traza** de la especificación ecuacional ds, a toda valoración, $\mu : X \to Lu(A)$ que cumpla:

$$\forall x \in X\text{-}Ins,\ \mu(x) = \hat{\mu}(\varphi(x))$$

Es decir, una traza es una aplicación que empareja cada una de las señales de una especificación con una secuencia particular de valores tal que

sea equivalente a la secuencia calculada por la interpretación del término que define a dicha señal.

EJEMPLO 2.18

Volvamos a la especificación *Mac* del ejemplo 2.14. Si tomamos como álgebra soporte de la signatura *Enteros* el conjunto de los números enteros, Z, una posible traza, μ, puede ser:

- $\mu(x) = « 2, 2, 2, 2, 2 \ldots »$
- $\mu(y) = « 1, 2, 3, 4, 5 \ldots »$
- $\mu(z) = « 0, 2, 6, 12, 20, 30 \ldots »$
- $\mu(out) = « 0, 2, 6, 12, 20, 30 \ldots »$

ya que para las dos únicas definiciones del cuerpo se cumple la equivalencia establecida por la definición 2.29:

- $\hat{\mu}(\varphi(out)) \equiv \hat{\mu}(z) = \mu(z) = \mu(out)$ *por definición de* $\hat{\mu}$
- $\hat{\mu}(\varphi(z)) \equiv \hat{\mu}(0 \text{ fby } (z + x*y)) = \ldots =$
  $= \textit{if } t{=}1 \textit{ then } 0 \textit{ else } (« 0, 2, 6 \ldots » + « 2, 2, 2 \ldots » * « 1, 2, 3 \ldots ») =$
  $= « 0, 2, 6, 12, 20, 30 \ldots » \equiv$ *operando sobre secuencias*
  $\equiv \mu(z)$

Por el contrario, toda valoración para la que, por ejemplo, $\mu(out) \neq \mu(z)$ no podrá ser traza ya que violará la exigencia de $\hat{\mu}(\varphi(out)) = \mu(out)$.

Para finalizar obsérvense dos aspectos. Primero, que existirán al menos tantas trazas como secuencias distintas de entrada y segundo, que la noción de traza es otro concepto semántico que depende del álgebra que dé soporte a las definiciones por lo que nunca debe suponerse un significado implícito de los símbolos. Así, bajo la anterior álgebra soporte la especificación estaba modelando un multiplicador-acumulador abstracto, pero si el álgebra soporte es la formada por los vectores de bits de longitud 8 y las operaciones soporte se realizan en módulo 8, la especificación está modelando un multiplicador-acumulador concreto de 8 bits.

El concepto de traza no es un concepto vacuo, ya que la existencia de al menos una traza para toda especificación ecuacional está asegurada por la existencia de puntos fijos en las álgebras dinámicas (gracias a la estructura de dominio inducida por la extensión para denotar indefinición).

2.30 PROPOSICIÓN. Toda especificación ecuacional tiene al menos una traza.

DEMOSTRACIÓN. Primero definiremos la noción auxiliar de definición cerrada, que generaliza sobre especificaciones ecuacionales la noción de término cerrado (véase definición 2.8). Una definición se dice que es cerrada si el término que la compone es cerrado o, siendo abierto, todas las señales que utiliza están definidas como cerradas. Como puede observarse, al igual que los términos cerrados, las definiciones cerradas tienen una interpretación independiente de valoraciones y lo representaremos por $\mu(x)^{Lu(A)}$. Desde el punto de vista hardware esta noción es más clara: las señales definidas como cerradas no dependen de los puertos de entrada por lo que para todo estímulo verificarán el mismo comportamiento.

Para una especificación ecuacional genérica (Σ, X, *Ins*, *Outs*, φ) y una $Lu(\Sigma)$-álgebra $Lu(A)$, la siguiente valoración, μ, es una traza:

- $\forall x \in Ins$, $\mu(x) = \lambda t : \mathbf{N}_+ . \perp : (A^{\#})_{\perp}$
- $\forall x \in X\text{-}Ins$ no definida como cerrada, $\mu(x) = \lambda t : \mathbf{N}_+ . \perp : (A^{\#})_{\perp}$
- $\forall x \in X\text{-}Ins$ definida como cerrada, $\mu(x) = \varphi(x)^{Lu(A)}$

Para comprobar que dicha valoración es una traza, basta con calcular la interpretación de las definiciones:

- $\forall x \in X\text{-}Ins$ no definida como cerrada, $\hat{\mu}(x) = \lambda t : \mathbf{N}_+ . \perp : (A^{\#})_{\perp} = \mu(x)$ ya que todas las funciones soporte se han definido estrictas.
- $\forall x \in X\text{-}Ins$ definida como cerrada, $\hat{\mu}(x) = \mu(x)$ por tener un valor fijo que depende sólo del álgebra soporte y no de valoraciones.

□

Como puede comprobarse en el ejemplo 2.18 la noción de traza tiene una fuerte componente intuitiva que la pone en clara relación con el

comportamiento observable de un circuito en funcionamiento o con una simulación del mismo. Obviamente por simulación no podemos obtener una traza completa pero si que podemos obtener aproximaciones tan precisas como deseemos: basta con aumentar cuanto se quiera el tiempo de simulación.

A continuación relacionaremos los conceptos de simulación y estímulo con el de traza. Así definiremos estímulo a toda traza parcial definida sólo para los puertos de entrada. Llamaremos simulación bajo cierto estímulo, a toda traza que, restringida a los puertos de entrada, sea equivalente al estímulo.

2.31 DEFINICIÓN. Sea una especificación ecuacional ds = (Σ, X, Ins, $Outs$, φ), y una $Lu(\Sigma)$-álgebra $Lu(A)$. Llamaremos **estímulo** de la especificación ecuacional ds, a toda valoración, stm: $Ins \rightarrow Lu(A)$, definida solamente para los puertos de entrada de la especificación ecuacional.

2.32 DEFINICIÓN. Sea una especificación ecuacional ds = (Σ, X, Ins, $Outs$, φ), una $Lu(\Sigma)$-álgebra $Lu(A)$, y un estímulo stm. Llamaremos **simulación** de la especificación ecuacional ds bajo el estímulo stm, a toda traza μ, que sea equivalente a stm para todo puerto de entrada, o sea, $\forall x \in Ins\ \mu(x) \equiv stm(x)$.

La definición de simulación presentada es declarativa pero no operativa, es decir, define que relación cumplen una simulación y el estímulo que la genera, pero no fija cómo obtener una simulación a partir de un estímulo, que es lo que se conoce como simular. Por ello, sea una especificación ecuacional:

ds = (Σ, ($in_1,\overset{p}{..},in_p,out_1,\overset{q}{..},out_q,z_1,\overset{m}{..},z_m$), ($in_1,\overset{p}{..},in_p$), ($out_1,\overset{q}{..},out_q$), φ) una $Lu(\Sigma)$-álgebra $Lu(A)$, y un estímulo stm. La traza μ definida a continuación es una simulación de la especificación ecuacional ds bajo el estímulo stm[†]:

- $\forall x \in Ins$, $\mu(x) = stm(x)$

[†] Se ha omitido, por simplicidad, la declaración explícita de tipos que puede encontrarse en la definición 2.27.

- $\forall out_i \in Outs,\ \mu(out_i) = (\ \mathbf{C}[\![\ ds\ ]\!](\ stm(in_1), ..., stm(in_p)\ )\ )\downarrow i =$
  $= \lambda(\ in_1, \overset{p}{...}, in_p\ ).$
  $(\ fix(\ \lambda(\ out_1, \overset{m}{...}, out_q, z_1, \overset{m}{...}, z_m\ )$
  $(\ \mathbf{E}[\![\ \varphi(out_1)\ ]\!], \overset{q}{...}, \mathbf{E}[\![\ \varphi(out_q)\ ]\!], \mathbf{E}[\![\ \varphi(z_1)\ ]\!], \overset{m}{...}, \mathbf{E}[\![\ \varphi(z_m)\ ]\!]\ )$
  $)\downarrow i\ )\ (\ stm(in_1), ..., stm(in_p)\ )$
- $\forall z_i \in X\text{-}Ins\text{-}Outs,\ \mu(z_i) =$
  $= \lambda(\ in_1, \overset{p}{...}, in_p\ ).$
  $(\ fix(\ \lambda(\ out_1, \overset{m}{...}, out_q, z_1, \overset{m}{...}, z_m\ ).$
  $(\ \mathbf{E}[\![\ \varphi(out_1)\ ]\!], \overset{q}{...}, \mathbf{E}[\![\ \varphi(out_q)\ ]\!], \mathbf{E}[\![\ \varphi(z_1)\ ]\!], \overset{m}{...}, \mathbf{E}[\![\ \varphi(z_m)\ ]\!]\ )$
  $)\downarrow q+i\ )\ (\ stm(in_1), ..., stm(in_p)\ )$

2.4.5 Ejemplos de especificaciones ecuacionales.

A continuación se mostrará como este mecanismo de especificación aún siendo simple, es lo suficientemente versátil para adaptarse a las necesidades básicas de especificación de muchas aplicaciones. Se comprobará que quizás es demasiado austero y que podrían concebirse ampliaciones para facilitar la especificación de jerarquías o de comportamientos repetitivos. Esto no se ha hecho ya que el objetivo que se pretende es demostrar cómo este nuevo paradigma de especificación es tan válido como otros, y cómo además presenta ventajas en muchos aspectos: es intuitivo, es formal, se adapta mucho mejor a los modos naturales de especificación, puede construirse sobre él un sistema de síntesis formal completo, es ampliable para soportar todas las necesidades de especificación de un entorno de SAN (desde conductas hasta bibliotecas pasando por protocolos), puede expresar estadios intermedios en un proceso de SAN (circuitos semiplanificados o semiasignados), etc.

Así en esta sección se mostrarán, mediante ejemplos, los aspectos más relevantes del formalismo en relación a su potencialidad en describir

conductas. Se comenzará explicando las pautas para obtener una especificación ecuacional a partir de otros métodos de especificación, a continuación se darán las directrices para la especificación ecuacional directa de sistemas con y sin protocolos de comunicación y para finalizar se mostrarán, ecuacionalmente especificados, algunos benchmarks clásicos de SAN.

Obtención de especificaciones ecuacionales a partir de otros métodos de especificación.

La obtención de una especificación ecuacional a partir de una temporal es inmediata y queda recogida esquemáticamente en la fig. 2.8.

Los contenidos de la signatura y de los conjuntos de señales y puertos deben formalizarse en base a conocimientos implícitos que asume la especificación temporal. Así toda variable u operador que aparezca en una especificación temporal deberá aparecer respectivamente en el conjunto de señales o en la signatura de la especificación ecuacional.

Para obtener la definición de las señales, utilizando el operador fby en lugar de la indexación temporal, bastará con darse cuenta de que la historia completa de valores que transporta cada variable que aparece en una especificación temporal, $x(t)$ con $t \in \mathbf{N}_+$, puede ser ordenada cronológicamente en una secuencia de valores « $x(1)$, $x(2)$, $x(3)$, ... ». Según el enfoque

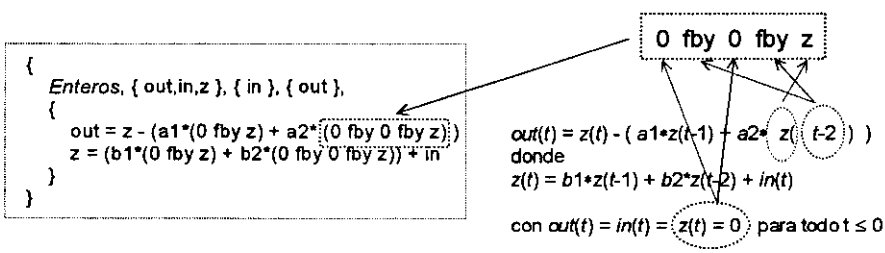


Fig. 2.8: De temporal a ecuacional.

ecuacional, esta secuencia puede referenciarse por una única señal, x, de uno de los tipos de datos *Lu*-extendidos. Si $x(t)$ se referencia por x, entonces la historia completa de valores transportados por la variable $x(t\text{-}1)$, con $t \in \mathbf{N}_+$ y $x(t')=c$ para todo $t'<0$, es decir, la secuencia de valores « c, $x(1)$, $x(2)$, $x(3)$ ... », puede referenciarse por el término c fby x. A su vez, toda variable del tipo $x(t\text{-}2)$ podrá ser referenciada por c' fby c'' fby x y así sucesivamente. Si aparecen otro tipo de índices temporales, la mejor alternativa es realizar una traslación temporal que modificará uniforme y simultáneamente todos los índices para que todos queden de la forma anteriormente explicada.

Por otro lado, obtener una especificación ecuacional a partir de una estructural es, si cabe, más simple (véase la fig. 2.9). Para ello, primero es necesario dar nombre a cada arco que aparezca en el diagrama de bloques. El conjunto de dichos nombres compondrán el conjunto de señales. Después se debe detallar en la signatura el interfaz de cada uno de los nodos primitivos (número y género de sus argumentos). Para finalizar se necesita obtener la definición de cada una de las señales.

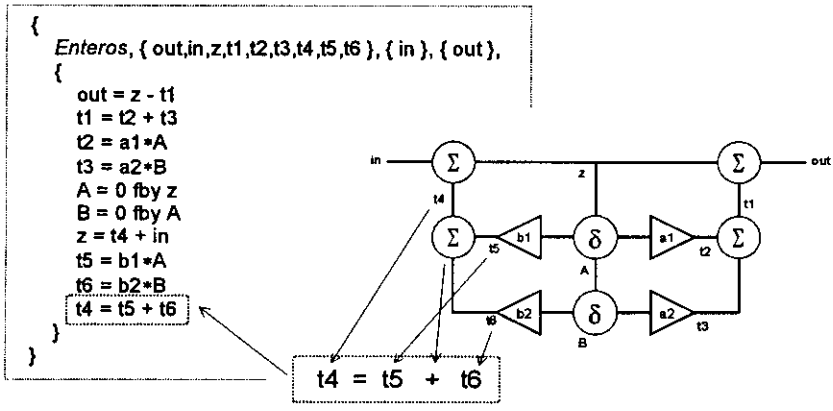


Fig. 2.9: De estructural a ecuacional.

Toda señal generada por un nodo primitivo será definida como:

'señal de salida' = *operador*(*'lista de señales de entrada'*)

y toda señal generada por un nodo de retardo (si se recuerda el significado temporal del operador *fby*) se definirá como:

'señal de salida' = *'valor inicial'* fby *'señal de entrada'*

Es importante destacar que si la especificación combina estructura (incluso jerárquica) y comportamiento temporal, el enfoque puede aplicarse del mismo modo ya que ambos estilos descriptivos pueden transformarse al mismo formalismo común.

Para finalizar reseñaré la relación que existe entre una especificación procedural y una ecuacional. En §2.3.3 se explicó cómo la primera tarea de una herramienta de SAN era compilar el código fuente y generar el correspondiente grafo de flujo de datos. Tras esto se comprobó cómo el grafo resultante era equivalente a cierta especificación estructural. Por consiguiente el camino para obtener una especificación ecuacional a partir de una procedural quedó descrito. Basta con compilar el código y tratar a la estructura resultante como si de una especificación estructural se tratara.

Aunque se ha simplificado este proceso, ya que existen representaciones internas muy complejas, no debe olvidarse que es posible hacerlo. Por muy complejas que sean las notaciones, el modelo que las sustenta no es distinto del que abarca una especificación ecuacional.

Especificación directa de conductas: método.

Cualquier cálculo lineal es fácil de especificar ecuacionalmente: sólo se necesitan operadores no-temporales y un número arbitrario de señales intermedias que hagan más simples las expresiones. Dada la transparencia referencial del formalismo, la definición de las mismas podrá hacerse en cualquier orden, ya que éste no influye en el comportamiento. Si el

comportamiento necesita construcciones selectoras (selección condicional o múltiple) bastará con añadir el símbolo que corresponda a la signatura.

EJEMPLO 2.19

Especifiquemos un sistema que continuamente tome como entradas los coeficientes *a*, *b*, y *c* de un polinomio de 2º grado, calcule los valores de las raíces y las vuelque sobre la salidas *out*1 y *out*2. Además deberá detectar cuándo las raíces son imaginarias en cuyo caso devolverá sendos ceros.

Suponiendo que existe una signatura en la que están definidos todos los símbolos de operación (*, +, /, sqrt, if-then-else), un posible cuerpo ecuacional puede ser:

```
body
out1 = if ( d < 0 ) then 0 else ( ( (-b) + e ) / f )
out2 = if ( d < 0 ) then 0 else ( ( (-b) - e ) / f )
e = sqrt( d )
d = ( b*b ) - ( 4*(a*c) )
f = (2*a)
```

Especificación que se correspondería con el siguiente cuerpo de descripción procedural (obsérvese que como el ecuacional, también se ha descrito para estar en continuo funcionamiento).

```
loop
   d := ( b*b ) - ( 4*(a*c) );
   if ( d < 0 ) then
      e := sqrt( d );
      f := 2*a;
      out1 <= ( (-b) + e ) / f;
      out2 <= ( (-b) + e ) / f;
   else
      out1 <= 0;
      out2 <= 0;
   end if;
end loop;
```

Sin embargo, la principal dificultad de utilizar directamente el mecanismo de especificación ecuacional radica en comprender cómo puede expresarse la iteratividad.

Existen dos tipos de iteratividad para describir dos tipos de cálculos diferentes: los que se repiten un número fijo de veces y los que lo hacen un número variable. La primera clase no define en realidad un cálculo iterativo sino que abrevia la representación sintáctica de un cálculo lineal. De este modo, dado que el formalismo ecuacional no posee aún ese tipo de abreviaturas, se podrá especificar pero repitiendo explícitamente la misma definición en el cuerpo ecuacional tantas veces como número de iteraciones tenga el bucle. La segunda, que si que define en realidad un cálculo iterativo (o su equivalente recursivo), necesita utilizar el operador *fby*.

Para entender el uso del *fby*, es necesario recurrir a la interpretación iterativa de los elementos sintácticos. Según esta interpretación, las variables (y expresiones) que aparecen en un cuerpo ecuacional representan valores que cambian con el tiempo. Así la expresión 0 *fby x* puede ser interpretada como un objeto que tiene un valor inicial 0 y que en cada iteración toma como valor el que en la anterior iteración tenía *x*, o la expresión *x* + *y* como el objeto que en toda iteración toma como valor la suma del valor que en esa iteración toman *x* e *y*.

En general, cuando se tiene un sistema de ecuaciones puede pensarse que en cada iteración, cada una de las variables se actualiza simultáneamente en función de los valores de otras variables.

EJEMPLO 2.20

Considérese el siguiente fragmento de código procedural que calcula, por el algoritmo de Euclides, el máximo común divisor (MCD) de dos números que toma de los puertos de entrada *in*1 e *in*2.

```
begin
   a := in1;
   b := in2;
   while ( a /= b ) loop
      if ( a > b ) then
         a := a - b;
      else
         b := b - a;
      end if;
   end loop;
   out <= a;
end;
```

Para conseguir la correspondiente especificación ecuacional se deben detectar cuántas variables están involucradas en el cálculo iterativo, en este caso *a* y *b*. Estas variables tomarán a lo largo del tiempo valores aproximados que se estabilizarán cuando el MCD se alcance. Después, utilizando el significado iterativo de *fby*, se definirá el comportamiento de las mismas.

```
body
a = in1 fby if ( a > b ) then ( a - b ) else a
b = in2 fby if ( b > a ) then ( b - a ) else b
out = a
```

Obsérvese cómo ambas especificaciones son quizás poco realistas por no definir ningún protocolo con el exterior. Así ninguna de ellas espera por una señal de inicio, por lo que definen un circuito que en cuanto empiece a funcionar deberá leer inmediatamente el valor de los puertos.

Además, tampoco avisan cuándo están los resultados disponibles, sino que cierto número de ciclos después de empezar el cálculo (número que depende de los datos de entrada) volcará el resultado. El valor del puerto de salida hasta que termine el cálculo, queda implícitamente no definido por la primera especificación, mientras que por la segunda se especifica que evolucione al ritmo de *a*.

Para finalizar, ambas especificaciones sólo realizan el cálculo para la primera muestra. Mientras que en la primera especificación queda implícito, ya que no se especifica qué hacer tras volcar en valor calculado (acaba el

algoritmo con la palabra **end**) en la segunda se explícita que permanezca en funcionamiento sin hacer nada. Obviamente si queremos realizar un nuevo cálculo será necesario reiniciar el sistema.

Especificación simultánea de conductas y protocolos de interfaz: método.

Cualquier especificación realista de un sistema digital debe, junto al puro comportamiento, acompañar una especificación del protocolo básico de comunicación con el exterior. Justo cuando se añade esta especificación, es cuando el mecanismo de especificación ecuacional se hace más útil. A continuación se añadirán al algoritmo básico mostrado en el ejemplo 2.19 distintos protocolos de comunicación. En el apéndice C, se muestra a modo de comparación las correspondientes especificaciones procedurales escritas en VHDL que verifican el mismo comportamiento y que son aceptadas por la que es, quizás, la herramienta comercial de SAN más ampliamente utilizada, el *Behavioral Compiler* de *Synopsys*. Compruebe el lector, mediante dicha comparación, la simplicidad notacional que permite el mecanismo de especificación ecuacional y lo intuitiva que es su semántica.

EJEMPLO 2.21

Supongamos, primero, que el circuito tiene dos puertos de entrada de datos, *ain* y *bin*, por los que cuando cierto puerto *start* valga uno, deberán leerse simultáneamente los argumentos del algoritmo. Además, por un puerto de salida *done*, el circuito deberá indicar cuando hay un resultado válido en el puerto de salida de datos *outp*. Si en mitad de un cálculo *start* se activa, deberán abandonarse los cálculos y volver a leer los puertos de entrada.

Un posible cuerpo ecuacional que verifica dicho comportamiento es:

```
body                                                    cuerpo A
a = 0 fby ( if (start=1) then ain else xa )
b = 0 fby ( if (start=1) then bin else xb )
xa = if ( a > b ) then ( a - b ) else a
xb = if ( b > a ) then ( b - a ) else b
done = if ( a = b ) then 1 else 0
outp = a
```

Como puede verse, primero se han definido *a* y *b*, que son las señales sobre las que se realiza el cálculo iterativo. En el primer ciclo valen 0 y en el resto dependerá del valor de *start*: si es 1 cargarán el valor presente en los puertos, si es 0, almacenarán el valor calculado por *xa* y por *xb*. Estas dos últimas señales, que se definen según el valor de *a* y *b*, implementan el algoritmo de Euclides. En la definición de *done* puede verse cuándo el circuito indicará que el calculo ha terminado, cuando *a* y *b* sean iguales. Una posible simulación del algoritmo puede verse en la fig. 2.10.

Supongamos, ahora, que existe un único puerto de entrada *inp*, lo que obliga a multiplexar su uso para leer los dos argumentos que precisa el algoritmo. Especifiquemos un protocolo que considera que ambos valores llegan en instantes sucesivos cuando *start* vale 1, primero *a* y después *b*. Además *start* deberá estar activo durante un único ciclo, esto hará que si vale 1 durante más de un ciclo cada uno de los valores leídos de *inp* deberán

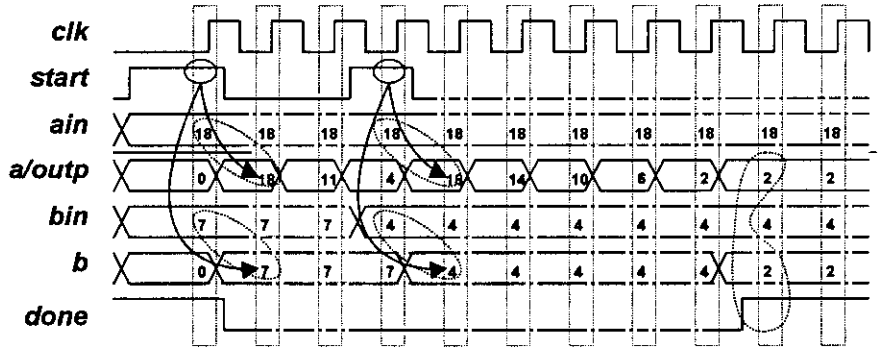


Fig. 2.10: MCD con protocolo de arranque/fin.

sobreescribir al valor anterior de *a*. Para finalizar también se especificará de manera que los cálculos puedan interrumpirse en respuesta a las activaciones de *start*.

Para especificar fácilmente este comportamiento vamos a utilizar un nuevo operador temporal que en el capítulo 4 se añadirá, junto a otros 3, al mecanismo de especificación ecuacional. Es el operador *next*. Su comportamiento es opuesto al de *fby*: elimina el primer valor de una secuencia. Si *x* representa a la secuencia « *x*(1), *x*(2), *x*(3), ... », el término *next x* representa la secuencia « *x*(2), *x*(3), *x*(4), ... ». Gracias a él el comportamiento del circuito queda como:

cuerpo B

```
body
a = 0 fby ( if (start=1) then inp else xa )
b = 0 fby ( if (start=1) then (next inp) else xb )
xa = if ( a > b ) then ( a - b ) else a
xb = if ( b > a ) then ( b - a ) else b
done = if ( a = b ) then 1 else 0
outp = a
```

Como puede verse, la única variación respecto de la anterior especificación es la lectura de puertos. Así cuando *start* vale 1, *a* cargará el valor presente en *inp* y *b* el siguiente valor que *inp* transporte. Este comportamiento se simula en la fig. 2.11. La fig. 2.12 muestra como efectivamente si *start* permanece activa durante más de un ciclo, el valor de *a* se sobreescribe.

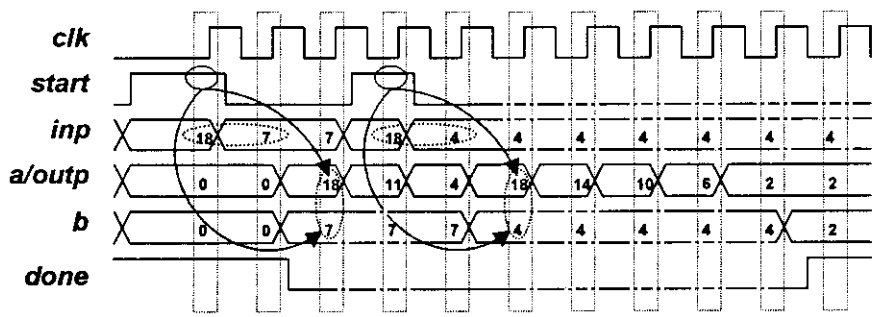


Fig. 2.11: MCD con un puerto multiplexado.

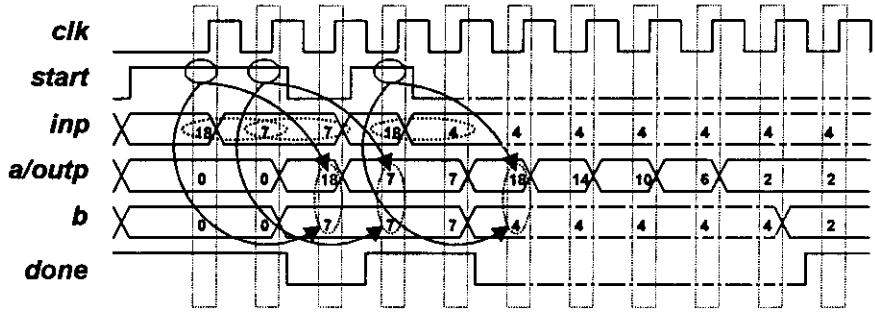


Fig. 2.12: Una interrupción de cálculos.

Si deseamos que el cálculo no pueda interrumpirse hasta que no termine, nada más fácil que especificar que la carga se realice cuando *start* y *done* (que indica cálculo finalizado) estén activadas.

body *cuerpo C*
a = 0 fby (if ((start=1) and (done=1)) then inp else xa)
b = 0 fby (if ((start=1) and (done=1)) then (next inp) else xb)
xa = if (a > b) then (a - b) else a
xb = if (b > a) then (b - a) else b
done = if (a = b) then 1 else 0
outp = a

Si, por otro lado, queremos que *out* no refleje los cálculos internos y, entre un cálculo y otro, permanezca estable con el último resultado calculado, deberemos modificar el cuerpo B como sigue:

body *cuerpo D*
a = 0 fby (if (start=1) then inp else xa)
b = 0 fby (if (start=1) then (next inp) else xb)
xa = if (a > b) then (a - b) else a
xb = if (b > a) then (b - a) else b
done = if (a = b) then 1 else 0
z = 0 fby (if (done=1) then a else z)
outp = z

La especificación usando *next*, no es estrictamente necesaria ya que es posible hacerla simplemente con *fby* aunque resulte menos intuitiva. Bastará

hacer una traslación temporal en la lectura de todos los puertos y obtener, a partir del cuerpo B, la siguiente especificación equivalente:

cuerpo E

```
body
a = 0 fby ( if ((0 fby start)=1) then (0 fby inp) else xa )
b = 0 fby ( if ((0 fby start)=1) then inp else xb )
xa = if ( a > b ) then ( a - b ) else a
xb = if ( b > a ) then ( b - a ) else b
done = if ( a = b ) then 1 else 0
outp = a
```

Esta puede leerse como: si en cierto instante el valor anterior de *start* es 1, entonces *a* toma el valor anterior de *inp* y *b* el valor actual de *inp*.

Para finalizar un último ejemplo que muestre como es posible especificar las formas de onda que deben adoptar las señales de control de un protocolo. Para ello vamos a especificar el mismo protocolo que el anterior (con un puerto de entrada multiplexado) pero ahora suponiendo que *start* debe permanecer en alta durante los dos ciclos que dura la carga de datos. Este comportamiento se muestra en la fig. 2.13. La especificación ecuacional correspondiente es como sigue.

cuerpo F

```
body
a = 0 fby aux
aux = if ( (start=1) and ((next start)=1) ) then inp else xa
b = 0 fby ( if ( ( next start ) = 1 ) then ( next inp ) else xb )
xa = if ( a > b ) then ( a - b ) else a
xb = if ( b > a ) then ( b - a ) else b
```

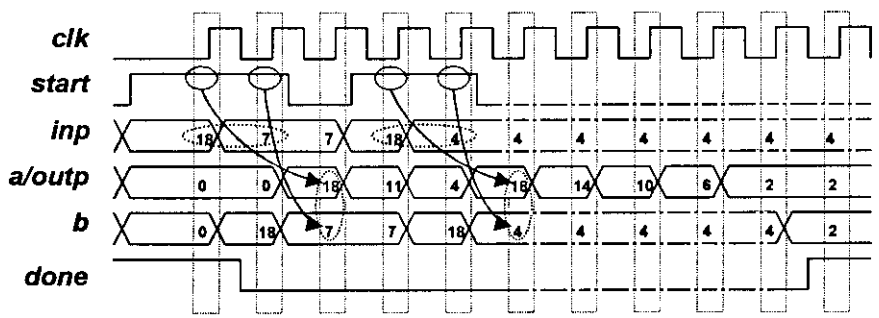


Fig. 2.13: MCD modificado con un puerto multiplexado.

```
done = if ( a = b ) then 1 else 0
outp = a
```

O, sin utilizar *next*.

body *cuerpo G*

```
a = 0 fby aux
aux = if ( ((0 fby start)=1) and (start=1) ) then (0 fby inp) else xa
b = 0 fby ( if (start=1) then inp else xb )
xa = if ( a > b ) then ( a - b ) else a
xb = if ( b > a ) then ( b - a ) else b
done = if ( a = b ) then 1 else 0
outp = a
```

Capítulo 3

Transformación de especificaciones ecuacionales

... metamathematics [formerly] was
not considered as a science
describing objective mathematical
states of affairs, but rather as a
theory of the human activity of
handling symbols.
Kurt Gödel

Un sistema formal es un conjunto de símbolos junto con una colección de reglas para emplearlos. La conjunción de ambas definiciones da lugar, de modo natural, a la definición de conjuntos de reglas de manipulación formal para transformar grupos de símbolos. En el capítulo 2, se presentaron los símbolos, cómo agruparlos para que formaran especificaciones ecuacionales (sintaxis) y cuál era el significado de éstas últimas (semántica). En este capítulo se presentará el conjunto de reglas de inferencia que permita derivar unas especificaciones ecuacionales a partir de otras. Desde el punto de vista operativo, estas reglas de manipulación definirán un procedimiento finito para transformar unas especificaciones ecuacionales en otras y, dado que el objetivo que se persigue es realizar síntesis correcta, serán pocas, intuitivas

y simples para reducir el riesgo de error en su implementación efectiva y facilitar una eventual verificación del código implementado.

El capítulo comienza (§3.1) definiendo un conjunto de reglas de manipulación sintáctica de especificaciones ecuacionales que permiten transformarlas según cualquier conocimiento formalizado mediante fórmulas universales de primer orden. Estas reglas se construyen mediante un conjunto de observaciones y manipulaciones básicas que se realizan sobre los términos que forman las definiciones de la especificación. A continuación (§3.2), se definen un par de nociones de equivalencia conductual en base a las cuales se demuestra la corrección (desde el punto de vista semántico) del sistema de derivación presentado. Seguidamente se discute sobre su completitud y se estudia su complejidad temporal. Para finalizar (§3.3), se ilustra la utilización del sistema formal para la reproducción metódica de un conjunto de técnicas de diseño hardware sobre el mecanismo de especificación ecuacional.

3.1 Un conjunto de reglas para la transformación de especificaciones ecuacionales.

Como se ha dicho, una regla de transformación define un mecanismo que permite realizar un cambio puramente sintáctico sobre cierto formalismo. En esta sección se establece un conjunto básico de normas de manipulación simbólica que puedan ser aplicadas a una especificación ecuacional de manera que el comportamiento de la misma no cambie (o que conserve ciertos invariantes). Además, dado que todo algoritmo (incluidos los de síntesis) define una manipulación sobre ciertas estructuras sintácticas de datos, no debe olvidarse que cuando se define una regla de transformación, se está especificando a la vez un cierto algoritmo.

3.1.1 Observación y transformación de términos.

Una especificación ecuacional está compuesta de definiciones y una definición es un par señal-término. De este modo, muchas de las manipulaciones sintácticas que se realicen sobre una especificación ecuacional, se traducirán finalmente en transformaciones de términos. A continuación se define un conjunto de observaciones y de manipulaciones básicas sobre términos. Éstas serán utilizadas, en posteriores secciones, para definir cómodamente el sistema de transformación de especificaciones ecuacionales.

Como puede verse en la fig. 3.1, es claro que todo término puede asimilarse a un árbol que represente la relación jerárquica operador-operando. A su vez, en todo árbol es posible identificar cada nodo como una palabra $u \in \mathbf{N}_+^*$ que indique la posición de dicho nodo en un recorrido primero en anchura. De este modo, el nodo raíz queda identificado por la palabra vacía ε. Cada uno de los nodos hijos de la raíz, por el número que indica su posición de izquierda a derecha. Cada uno de los nodos nietos de la raíz, como la concatenación de sus respectivas posiciones a las posiciones que ocupan sus padres, y así sucesivamente. Gracias a ello es posible introducir la idea de posición en un término [Rose73] como la posición equivalente en el árbol asociado. El conjunto de posiciones válidas en un término t, se expresará por $pos(t)$. El subtérmino de t en la posición u, se expresará mediante t/u. Y el símbolo que aparece en la posición u del término t será expresado por $t[u]$.

3.1 **Definición.** Sea (S, Σ) una signatura heterogénea, X una familia S-indexada de conjuntos de variables y sea el término $t \in T_\Sigma(X)$. Se define recursivamente el **conjunto de variables** de t, $var(t)$, el **conjunto de posiciones** de t, $pos(t)$, el **subtérmino** de t en la posición u, t/u, y el **símbolo** que aparece en la posición u del término t, $t[u]$, como:

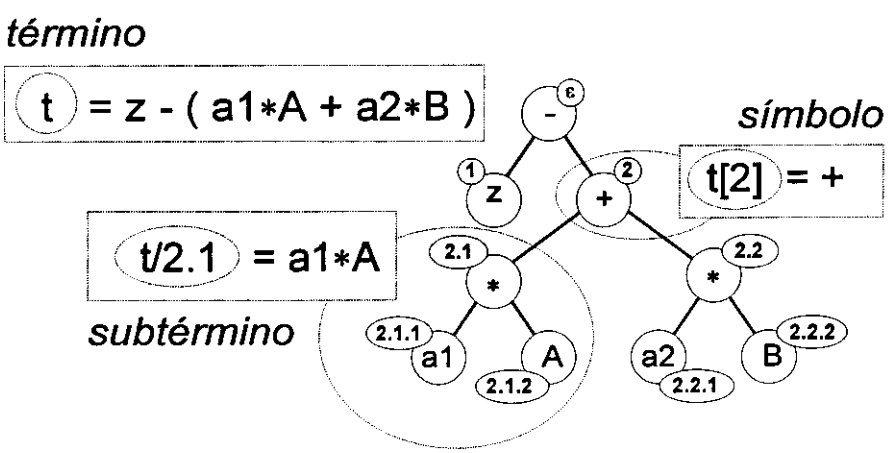


Fig. 3.1: Observaciones sobre términos.

- si t es un término simple, entonces

  $var(t) = t$ si $t \in X$, en caso contrario $var(t) = \emptyset$

  $pos(t) = \{ \varepsilon \}$

  $t/\varepsilon = t$

  $t[\varepsilon] = t$

- si t es el término compuesto $\sigma(t_1, ..., t_n)$, entonces

  $var(t) = \bigcup_{i=1}^{n} var(t_i)$

  $pos(t) = \{ \varepsilon \} \cup (\bigcup_{i=1}^{n} i.pos(t_i))$

  $t/\varepsilon = t$, y $t/i.u = t_i/u$

  $t[\varepsilon] = \sigma$, y $t[i.u] = t_i[u]$

Además resulta útil definir un subconjunto de $pos(t)$: el **conjunto de posiciones variables** de t, $pos_0(t)$, como el conjunto de posiciones de t ocupadas por un símbolo de variable:

$$pos_0(t) = \{ u \in pos(t) \mid t/u \in X \}$$

Todos los conceptos hasta ahora definidos permiten 'observar' los términos. A continuación, se definen unos nuevos que permiten modificarlos. El primero de ellos es el concepto de reemplazamiento.

3.2 **DEFINICIÓN.** Sea (S, Σ) una signatura heterogénea, X una familia S-indexada de conjuntos de variables y sean los términos $t,t' \in T_\Sigma(X)$. Se define recursivamente el **reemplazamiento** del subtérmino t/u por otro término t', $t[u \leftarrow t']$, como:

- $t[\varepsilon \leftarrow t'] = t'$ si t y t' son del mismo género, en caso contrario $t[\varepsilon \leftarrow t'] \mp t$)
- $\sigma(\ t_1, \ ..., \ t_n\ )[i.u \leftarrow t'] = \sigma(\ t_1, \ ..., t_i[u \leftarrow t'], \ ..., \ t_n\ )$

Obsérvese que gracias a la comprobación de géneros que se realiza en (1), la noción de reemplazamiento está bien definida ya que produce términos bien formados, es decir, que siempre $t[u \leftarrow t'] \in T_\Sigma(X)$. Asimismo, nótese que el resultado de cualquier reemplazamiento 'anómalo' (aquel que trate de reemplazar un subtérmino de cierto género por un término de género distinto) es el término original.

Para poder fijar compactamente un conjunto de reemplazamientos no 'anómalos' de variables se usará la noción de substitución.

3.3 **DEFINICIÓN.** Sea (S, Σ) una signatura heterogénea y X e Y dos familias S-indexadas de conjuntos de variables. Una **substitución** es cualquier aplicación $\rho : Y \rightarrow T_\Sigma(X)$ que respete los géneros, esto es, que si $y \in Y_s$ se tenga que $\rho(y) \in T_{\Sigma,s}(X)$ con $s \in S$. Si la familia Y es finita, ρ admite la siguiente representación como conjunto de vínculos:

$$\rho \equiv \{\ x_1/t_1, \ x_2/t_2, \ ..., \ x_n/t_n\ \} \text{ siendo } t_i \equiv \rho(x_i)$$

Obsérvese que dado que, aparte de conjunto de términos, $T_\Sigma(X)$ es también una Σ-álgebra, toda substitución es también una valoración (véase definición 2.10). Por ello, la acción de hacer efectiva una substitución sobre un término cualquiera $t \in T_\Sigma(Y)$, que se realiza mediante el reemplazamiento de toda aparición de una variable en t por el correspondiente término indicado por la substitución, se representa mediante $\hat{\rho}(t)$. Esto es así ya que $\hat{\rho}(t)$ resulta ser la interpretación del término t inducida por la valoración ρ sobre el conjunto de términos $T_\Sigma(X)$ (véase definición 2.11). Así, toda substitución

establece una aplicación sobre dicho conjunto, es decir, que dada una valoración ρ, es posible definir recursivamenrte $\hat{\rho} : T_\Sigma(Y) \rightarrow T_\Sigma(X)$ como:

- $\hat{\rho}(x) = \rho(x), \forall x \in X$
- $\hat{\rho}(\sigma(t_1, ..., t_n)) = \sigma(\hat{\rho}(t_1), ..., \hat{\rho}(t_n))$

EJEMPLO 3.1

Sea la especificación ecuacional (Σ, X, *Ins*, *Outs*, φ) del filtro recursivo de primer orden mostrado en el ejemplo 2.16. Sea uno de los términos que forma una de las definiciones:

$$\varphi(\text{out}) \equiv z - (a1 * (0 \text{ fby } z)) \in T_{Lu(\Sigma)}(X)$$

Para este término, los conjuntos de variables, de posiciones válidas y de posiciones variables son:

$var(\varphi(out)) = \{ out \}$

$pos(\varphi(out)) = \{ \varepsilon, 1, 2, 2.1, 2.2, 2.2.1, 2.2.2 \}$

$pos_0(\varphi(out)) = \{ 1, 2.2.2 \}$

Algunos ejemplos de subtérminos y símbolos pueden ser:

$\varphi(out)/\varepsilon \equiv z - (a1 * (0 \text{ fby } z))$	$\varphi(out)[\varepsilon] \equiv -$
$\varphi(out)/2.1 \equiv a1$	$\varphi(out)[2.1] \equiv a1$
$\varphi(out)/2.2 \equiv 0 \text{ fby } z$	$\varphi(out)[2.2] \equiv \text{fby}$

Asimismo, algunos ejemplos de reemplazamientos pueden ser:

$\varphi(out)[\varepsilon \leftarrow b1] = b1$

$\varphi(out)[2.1 \leftarrow (0 \text{ fby } z)] = z - ((0 \text{ fby } z) * (0 \text{ fby } z))$

$\varphi(out)[2.2 \leftarrow b1 * in] = z - (a1 * (b1 * in))$

Para finalizar, sea la substitución expresada mediante conjuntos de vínculos $\rho \equiv \{ z/(x+y) \}$, el resultado de aplicarla sobre $\varphi(out)$ es:

$$\hat{\rho}(\varphi(out)) \equiv (x+y) - a1 * (0 \text{ fby } (x+y))$$

A continuación, se define un mecanismo para identificar dos términos a través de una substitución.

3.4 DEFINICIÓN. Sea (S, Σ) una signatura heterogénea, X e Y dos familias de conjuntos S-indexados de variables y sean los términos $t \in T_\Sigma(X)$ y $t' \in T_\Sigma(Y)$. Se dice que t se **ajusta sintácticamente** a t', si existe una substitución (que por definición respeta los géneros, véase la definición 3.3) $\rho : Y \rightarrow T_\Sigma(X)$ tal que $\hat{\rho}(t') \equiv t$. A ρ se le llama **substitución de ajuste** y se dice que t se ajusta a t' vía ρ.

El proceso de ajuste es decidible, y en caso de que para dos términos $t \in T_\Sigma(X)$ y $t' \in T_\Sigma(Y)$ exista una substitución que ajuste t a t', ésta es única y puede calcularse efectivamente. Así, sea el conjunto de posiciones de t' en las que hay un símbolo de variable, esto es, $pos_0(t')$. El término t se ajusta al término t' si y sólo si:

- $\forall u \in pos_0(t'),\ u \in pos(t)$ *(1)*
- $\forall u \in pos_0(t'),$ si $t'[u] \in Y_s$ entonces $t/u \in T_{\Sigma,s}(X)$ *(2)*
- $\forall u \in (pos(t') - pos_0(t')),\ t'[u] \equiv t[u]$ *(3)*
- $(\forall u,v \in pos_0(t') \mid u \neq v),\ t'[u] \equiv t'[v] \Rightarrow t/u \equiv t/v$ *(4)*

En (1) se establece que toda posición de t' en donde haya una variable, debe ser una posición válida en t. En (2), que el género de cada uno de los subtérminos de t ubicados en una posición que en t' esté ocupada por una variable, debe ser el mismo que el género de la variable correspondiente. En (3), que los símbolos de t' que ocupen posiciones no variables deben ser los mismos que los símbolos de t ubicados en dichas posiciones. Recuérdese que, dado que es posible que exista sobrecarga de identificadores, esta equivalencia debe ser en símbolo y perfil (véase definición 2.7). Finalmente en (4) se establece que todas las ocurrencias de la misma variable en t', deben asociarse respectivamente con subtérminos de t que sean equivalentes (en símbolos y perfiles). Si estas condiciones se cumplen, la sustitución de ajuste se construye como:

$$\forall u \in pos_0(t'),\ \rho(\ t'[u]\ ) = t/u$$

EJEMPLO 3.2

Sean la signatura Σ y el conjunto de señales X presentes en la especificación ecuacional del ejemplo 2.16, el conjunto de variables Y = { x, y, z }, y los términos $t_1 \equiv$ (0 fby x) - y, $t_2 \equiv$ x - (b1 *y), $t_3 \equiv$ x - (y *y), $t_4 \equiv$ x - (y *z), todos ellos incluidos en $T_{Lu(\Sigma)}(Y)$. Intentemos ajustar el término φ(out) $\equiv$ z - (a1 *(0 fby z)) $\in T_{Lu(\Sigma)}(X)$ con cada uno de ellos.

- no se ajusta a t_1 por no cumplir la condición (1): el conjunto de posiciones de t_1 en las que hay variables es { 2, 1.2 } y la posición 1.2 no pertenece a *pos*(φ(out)).
- no se ajusta a t_2 por no cumplir la condición (2): ya que t_2 y φ(out) poseen símbolos de operación distintos en la posición 2.1, t_2[2.1] $\equiv$ b $\neq$ a1 $\equiv \varphi$(out)[2.1].
- no se ajusta a t_3 por no cumplir la condición (3): t_3 tiene la variable *y* en las posiciones 2.1 y 2.2, pero φ(out)/2.1 $\equiv$ a1 $\neq$ 0 fby z $\equiv \varphi$(out)/2.2.
- se ajusta a t_4 vía la substitución $\rho \equiv$ { x/z, y/a1, z/(0 fby z) }.

Tras todas estas definiciones es posible definir el único mecanismo de manipulación de términos que se permitirá en el sistema de síntesis formal que va a definirse en §3.1.2 y en §3.1.3: la reescritura.

3.5 DEFINICIÓN. Sea una signatura (S, Σ) y una familia S-indexada de conjuntos de variables Y. Se define una **Σ-regla de reescritura de género *s*** como la tripleta (Y, t_L, t_R) donde $t_L, t_R \in T_{\Sigma,s}(Y)$ tal que $var(t_R) \subseteq var(t_L)$.

Por conveniencia, si los identificadores de variables y sus géneros son claros, una regla de reescritura suele notarse por $t_L \rightarrow t_R$

Toda regla de reescritura permite definir una relación entre términos que asocia a un término con su correspondiente término reescrito. Esta 'aplicación' de una regla de reescritura define un proceso computacional que puede

realizarse sobre la estructura del término en base a ajustes y reemplazamientos. Así si t y t' son dos términos y r es una regla de reescritura, se dice que t' se deriva de t vía r, esto es $t \rightarrow_r t'$ si, siendo $u \in pos(t)$, existe una substitución ρ tal que:

- $t/u \equiv \hat{\rho}(t_L)$ *t/u se ajusta a* t_L
- $t' \equiv t[\, u \leftarrow \hat{\rho}(t_R)\,]$

Cuando r se sobrentienda se escribirá simplemente $t \rightarrow t'$. Nótese que una regla de reescritura puede aplicarse en distintas posiciones, dando lugar a términos distintos.

EJEMPLO 3.3

Sea la regla de reescritura:

$$r \equiv (\ \{x,y,z\},\ x - (\ y + z\ ),\ (\ x + (-y)\ ) + (-z)\ )$$

y sean los términos:

$$t \equiv z - (\ a1*A + a2*B\ ) \quad \text{y} \quad t' \equiv (\ z + (-(a1*A))\ ) + (\ -(a2*B)\ )$$

Se dice que $t \rightarrow t'$ vía r (véase fig. 3.2) ya que, para cierta posición $\varepsilon \in pos(t)$, existe una substitución $\rho = \{\ x/z,\ y/(a1*A),\ z/(a2*B)\ \}$ que cumple:

- $t/\varepsilon \equiv (\ z - (\ a1*A + a2*B)\ )/\varepsilon \equiv$
  $\equiv z - (\ a1*A + a2*B\ ) \equiv$
  $\equiv \hat{\rho}(\ x - (\ y + z\ )\ )$
- $t' \equiv (\ z + (-(a1*A))\ ) + (\ -(a2*B)\ ) \equiv$
  $\equiv (\ z - (\ a1*A + a2*B\ )\ )[\ \varepsilon \leftarrow (\ (\ z + (-(a1*A))\ ) + (\ -(a2*B)\ )\ )\ ] \equiv$
  $\equiv (\ z - (\ a1*A + a2*B\ )\ )[\ \varepsilon \leftarrow \hat{\rho}(\ (\ x + (-y)\ ) + (-z)\ )\ ]$

La restricción $var(t_R) \subseteq var(t_L)$, no permite el uso de reglas que pudieran añadir ambigüedad al término reescrito por adición de variables extra que no hayan sido enlazadas por el proceso de ajuste. Por otro lado, toda ecuación $(\ Y,\ t_L,\ t_R\ )$ define de modo natural, si $var(t_L) = var(t_R)$, las dos reglas de

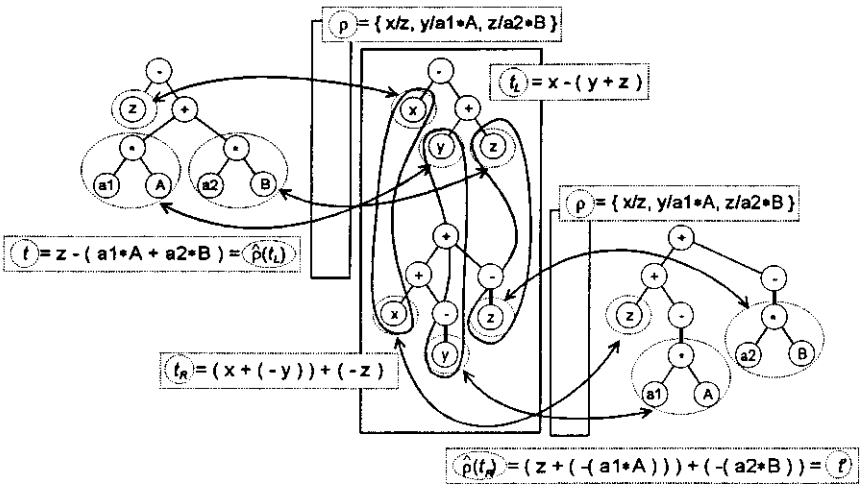


Fig. 3.2: Reescritura de un término.

reescritura (Y, t_L, t_R) y (Y, t_R, t_L). Si $var(t_L) \neq var(t_R)$ sólo una de ellas será utilizable.

3.1.2 Reglas estructurales de transformación de especificaciones ecuacionales.

El objetivo de las reglas estructurales es reorganizar la distribución de términos dentro de las definiciones que forman el cuerpo de una especificación ecuacional. Estas reglas no modifican ni el comportamiento de una especificación ni la manera en que los cálculos se realizan. Su objetivo es facilitar la aplicación del otro tipo de reglas que sí lo hacen: las conductuales.

Regla de substitución.

La primera regla es una consecuencia directa de la transparencia referencial del formalismo. Es decir, dado que toda aparición de una misma señal en el cuerpo de una especificación ecuacional denota el mismo objeto, ésta puede ser reemplazada por su definición sin alterar el comportamiento especificado. Más formalmente podemos definir la regla de substitución como:

Sea (Σ, X, *Ins*, *Outs*, φ) una especificación ecuacional y sean dos señales $x_1, x_2 \in X\text{-}Ins$. Si existe una ocurrencia de x_1 en la definición de x_2 en la posición $u \in Pos(\varphi(x_2))$, esto es $\varphi(x_2)[u] \equiv x_1$, la anterior especificación ecuacional puede transformarse en otra (Σ, X, *Ins*, *Outs*, φ')†, donde φ' se define como:

$$\varphi'(x_2) = \varphi(x_2)[\, u \leftarrow \varphi(x_1) \,] \wedge \forall x \in X\text{-}Ins\text{-}\{x_2\},\ \varphi'(x) = \varphi(x)$$

A partir de ahora, por conveniencia, las reglas no serán definidas así. Se utilizará una notación más compacta que es habitualmente utilizada en la especificación de sistemas de inferencia. Esta notación presenta el siguiente aspecto:

$$\frac{\textit{'Especificación ecuacional original'}}{\textit{'Especificación ecuacional transformada'}} \quad \textit{'Condiciones que debe cumplir la especificación original para poder ser transformada utilizando esta regla'}$$

Esta notación permite definir axiomáticamente el comportamiento e interfaz del algoritmo que realiza la transformación, estableciendo qué se puede asegurar tras la aplicación de la regla en términos de lo que era cierto antes de su aplicación. Así:

† Si son conductualmente equivalentes o no se demostrará en §2.3

Regla de substitución:

$$\frac{(\ \Sigma,\ X,\ I,\ O,\ \varphi \cup \{\ x_1 = \varphi(x_1),\ x_2 = \varphi(x_2)\ \}\ )}{(\ \Sigma,\ X,\ I,\ O,\ \varphi \cup \{\ x_1 = \varphi(x_1),\ x2 = \varphi(x_2)[u \leftarrow \varphi(x_1)]\ \}\ )} \quad \exists u \in pos(\varphi(x_1)) \mid \varphi(x_2)[u] \equiv x_1$$

Obsérvese cómo el símbolo de unión de conjuntos asume implícitamente que los conjuntos implicados son disjuntos, ya que lo que intenta expresar es un subconjunto particular de definiciones. Así, la anterior regla, expresa que de todo cuerpo del que puedan extraerse dos definiciones, de las cuales, una contenga a la señal definida por la otra, puede ser reemplazado por otro cuerpo compuesto por la primera definición transformada y por la segunda sin transformar. Por otro lado, es obvio que esta transformación respeta los géneros ya que parte de una especificación ecuacional que por definición los respeta (véase definición 2.25).

Un ejemplo de su aplicación se muestra en la fig. 3.3. Como puede verse, la regla de substitución puede aplicarse en muchos lugares dentro de una especificación ecuacional definiendo, cada uno de ellos, una transformación

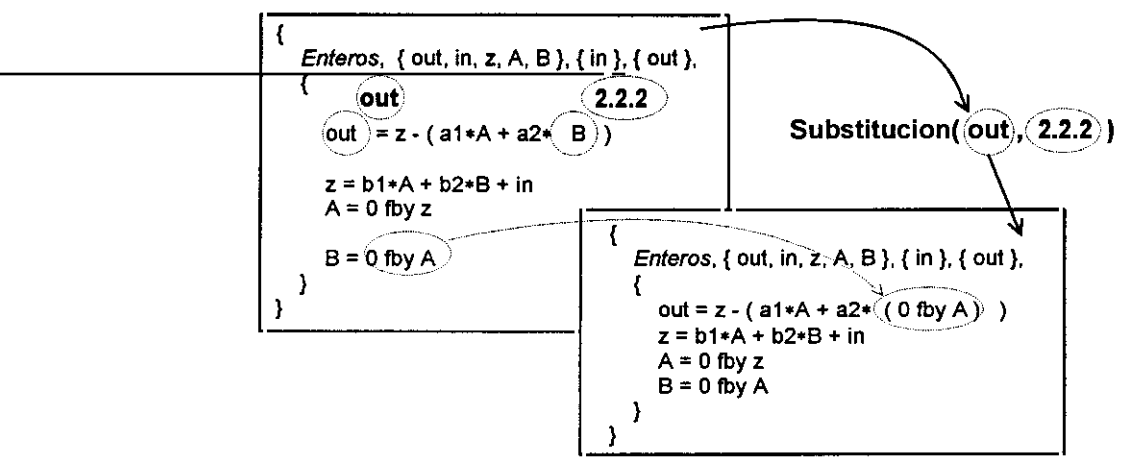


Fig. 3.3: Regla de substitución.

distinta. Para elegir de modo único una de las posibles transformaciones, será necesario que se parametrice el uso de la regla, fijando una señal particular y una posición determinada dentro de la definición. En la fig. 3.3 se refleja el resultado de aplicarla sobre la definición de *out* en la posición 2.2.2. Más adelante, en el capítulo 5, se estudiará cómo decidir en cada momento de un flujo de diseño cual es la elección más adecuada.

De ahora en adelante, una aplicación particular de esta regla vendrá representada en notación funcional, esto es, si x_2 es un identificador de señal y u un identificador de posición, el intento de substituir en la definición de x_2 la señal ubicada en la posición u estará representado por:

Substitucion(x_2, u **)**

Esta aplicación podrá efectuarse, o no. Todo dependerá de si la correspondiente especificación ecuacional sobre la que se aplique, cumple las condiciones fijadas por la regla o no. Esta notación pretende reflejar la forma que adoptará una posterior implementación de la regla que, en cualquier lenguaje de programación, será una llamada a procedimiento.

Regla de renombrado.

La segunda regla establece la posibilidad de renombrar cualquier señal que no sea puerto, siempre y cuando no se provoque una colisión de nombres. Para ello, definimos una substitución ρ que reemplaza el identificador a renombrar por el nuevo identificador en cada una de las definiciones del cuerpo de la especificación.

Regla de renombrado:

$$\frac{(\ \Sigma,\ X \cup \{x_1\},\ I,\ O,\ \varphi \cup \{\ x_1 = \varphi(x_1)\ \}\ )}{(\ \Sigma,\ X \cup \{x_2\},\ I,\ O,\ \hat{\rho} \circ \varphi \cup \{\ x_2 = \hat{\rho}(\varphi(x_1))\ \}\ )} \quad x_1 \notin I \cup O \wedge x_2 \notin X \cup Lu(\Sigma) \wedge \rho = \{x_1/x_2\}$$

La restricción de no aceptar que se renombren puertos es una concesión al pragmatismo. Desde el punto de vista teórico, no habría problema en cambiarles el nombre, pero si en un entorno de síntesis se permite el renombrado arbitrario de puertos sería necesario ir recordando la correspondencia de los puertos de la especificación original con los sucesivos nombres que adopta durante su diseño, y esto es una complicación añadida que no aporta ninguna ventaja real.

Nótese cómo se evita la colisión de nombres: poniendo como condición que la nueva señal no exista en el conjunto de señales y que no aparezca como símbolo de operación en la signatura *Lu*-extendida. Además, obsérvese que no existe ningún problema en componer φ y $\hat{\rho}$, ya que:

- $\varphi : X \cup \{x_1\}\text{-}I \rightarrow T_{Lu(\Sigma)}(X \cup \{x_1\}\text{-}O)$
- $\hat{\rho} : T_{Lu(\Sigma)}(X \cup \{x_1\}\text{-}I\text{-}O) \rightarrow T_{Lu(\Sigma)}(X \cup \{x_2\}\text{-}I\text{-}O)$

Un ejemplo de la aplicación de la regla de renombrado se muestra en la fig. 3.4 y, de ahora en adelante, una aplicación particular de esta regla vendrá representada por:

Renombrado(x_1, x_2 **)**

donde x_1 es el antiguo identificador y x_2 es el nuevo.

Regla de expansión.

La tercera regla permite simplificar las definiciones del cuerpo de una especificación ecuacional a la vez que amplia su número. Básicamente establece que cualquier subtérmino de una definición puede ser reemplazado por una nueva señal, siempre que esta señal se defina como el subtérmino a reemplazar y la nueva definición se añada al cuerpo de la especificación ecuacional sin producir confusión. Obviamente el género de la nueva señal deberá ser el género del subtérmino a expandir.

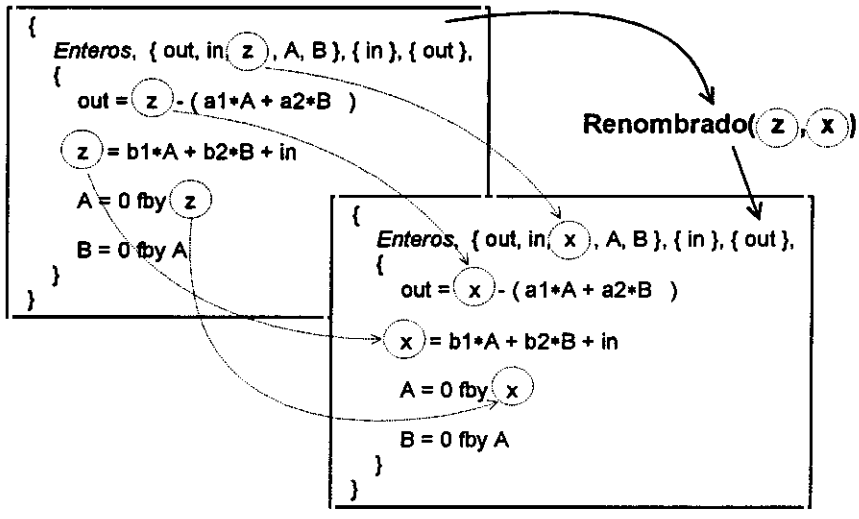


Fig. 3.4: Regla de renombrado.

Regla de expansión:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1) \})}{(\Sigma, X \cup \{x_2\}, I, O, \varphi \cup \{ x_1 = \varphi(x_1)[u \leftarrow x_2],\ x_2 = \varphi(x_1)/u \})} \quad u \in pos(\varphi(x_1)) \wedge x_2 \notin X \cup Lu(\Sigma)$$

Obsérvese cómo pueden generarse señales de paso (señales cuya definición es otra señal) si expandimos una definición formada por una única señal o expandimos la raíz de cualquier definición. Para evitar esto, bastaría con añadir a las condiciones: $x_1 \neq \varepsilon$ y $x_1/u \notin X$.

El efecto de la aplicación de la regla de expansión puede analizarse en la fig. 3.5, donde la acción de expandir bajo nombre x_2, el subtérmino ubicado en la posición u de la definición de la señal x_1, se expresa por:

Expansion(x_1, u, x_2 **)**

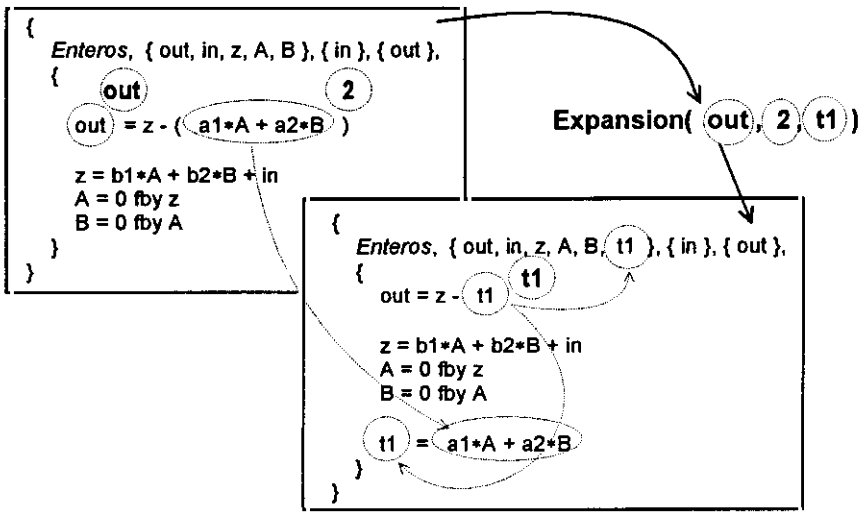


Fig. 3.5: Regla de expansión.

Regla de eliminación.

La cuarta regla permite, por el contrario, reducir el número de definiciones que aparecen en el cuerpo de una especificación mediante la eliminación del llamado código muerto, es decir, aquellas definiciones que no se utilizan en otras definiciones. Básicamente establece que cualquier definición de una señal que no sea puerto de salida y que no ocurra en ninguna otra definición del cuerpo de una especificación ecuacional, puede ser eliminada.

Regla de eliminación:

$$\frac{(\Sigma, X \cup \{x_1\}, I, O, \varphi \cup \{x_1 = \varphi(x_1)\})}{(\Sigma, X, I, O, \varphi)} \quad x_1 \notin O \wedge \wedge \forall x \in X\text{-}I\text{-}\{x_1\}, \forall u \in pos(\varphi(x)), \varphi(x)[u] \neq x_1$$

Esta vez, la restricción de no eliminar un puerto de salida si que es necesaria ya que si no fuera así, según la sintaxis definida por 2.25 que no

permite que un puerto de salida forme parte de una definición, ésta caería en la categoría de código muerto sin serlo en realidad. Obsérvese que esta regla no elimina posibles puertos de entrada no utilizados, y que para poder eliminar definiciones recursivas muertas no busca ocurrencias de la señal a eliminar dentro de su propia definición.

Para eliminar todo el código muerto de una especificación ecuacional no basta una aplicación de esta regla, es necesario aplicarla varias veces. La razón es que por sí misma no hace una búsqueda exhaustiva de todos los cálculos no utilizados, sino que detecta aquellos que se limiten a una única definición. Si un cálculo muerto se reparte por varias definiciones, será necesario aplicarla tantas veces como definiciones involucradas haya. Además si dicho cálculo está involucrado en un lazo recursivo, será necesaria la utilización de otras reglas para eliminarlo.

La fig. 3.6 muestra una aplicación de esta regla. La sintaxis que denota la aplicación de esta regla para intentar eliminar la señal x_1 es:

Eliminacion(x_1)

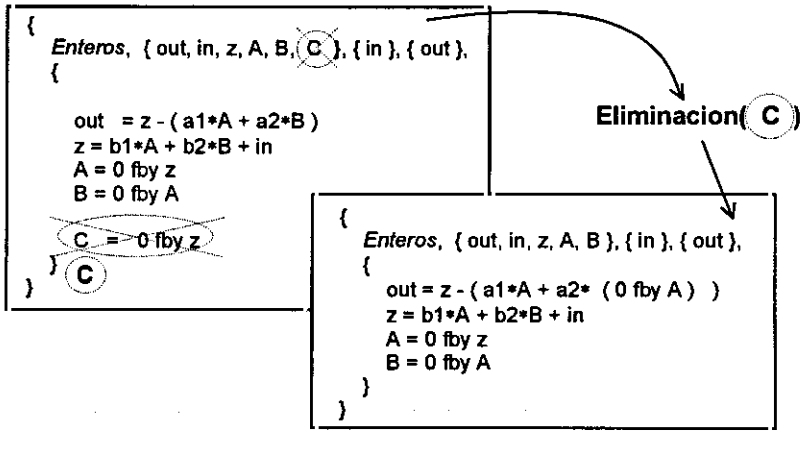


Fig. 3.6: Regla de eliminación.

Regla de limpieza.

La quinta regla también permite reducir el número de definiciones pero, esta vez, eliminando redundancias. Establece que cualquier definición de una señal que no sea puerto de salida, puede ser eliminada del cuerpo de una especificación ecuacional, si cada una de las apariciones de la señal se reemplaza por una aparición de otra señal que esté definida de igual modo.

Regla de limpieza:

$$\frac{(\ \Sigma, X \cup \{x_1\},\ I,\ O,\ \varphi \cup \{\ x_1 = \varphi(x_1),\ x_2 = \varphi(x_2)\ \}}{(\ \Sigma,\ X,\ I,\ O,\ \hat{\rho} \circ \varphi \cup \{\ x_2 = \hat{\rho}(\varphi(x_2))\ \}\ )} \quad x_1 \notin O \wedge \varphi(x_1) \equiv \varphi(x_2) \wedge \rho = \{x_1/x_2\}$$

Nótese que esta regla no define ningún procedimiento de búsqueda para encontrar dos definiciones equivalentes, sino que asume que se le indican. Esta regla solamente comprueba que es cierta dicha equivalencia (que debe ser en símbolos y perfiles) y elimina una de las definiciones. Será algún agente externo (ya sea el diseñador o un algoritmo) el que realizará esta búsqueda cuando sea necesaria.

Debe destacarse que ésta es una regla de limpieza sintáctica, ya que en las precondiciones se exige la equivalencia de los términos $\varphi(x_1)$ y de $\varphi(x_2)$ y no la igualdad. Así, por ejemplo, si en el cuerpo de una definición se encuentran las definiciones $a = c + 1$ y $b = 1 + c$ no podrá ser limpiada ninguna de ellas. Esto, que pudiera parecer una desventaja, en realidad no lo es ya que puede evitar muchos errores típicos en las herramientas de síntesis debidos tanto a la confusión entre sintaxis y semántica, como a la asunción implícita de semánticas no formales; explicaré la razón.

Para que se pudiera limpiar alguna de las anteriores definiciones sería necesario demostrar previamente que ambas denotan la misma cosa. Para

poder demostrar eso, sería necesario recurrir al álgebra soporte pero, por el momento, nuestro formalismo trabaja con semánticas totalmente genéricas: cualquier Σ-álgebra. Esto hace que para algunos modelos el operador denotado por el símbolo '+' pudiera ser conmutativo, pero para otros no. Así, la única manera de asegurar que dos términos son iguales para cualquier modelo es comprobar que son sintácticamente equivalentes.

La solución para poder limpiar dos definiciones semánticamente equivalentes vendrá por dos caminos: o bien transformando las definiciones hasta hacerlas sintácticamente equivalentes mediante las reglas conductuales de transformación (véase §3.1.3), o bien facilitando al sistema de síntesis un mecanismo para que pueda conocer el significado de los términos y pueda razonar sobre él (véase capítulo 6).

Un ejemplo gráfico del resultado de la regla de limpieza puede verse en la fig. 3.7. En ella se asume la notación que se utilizará para expresar la limpieza de cierta señal x_1 por otra definida de igual modo x_2:

Limpieza(x_1, x_2)

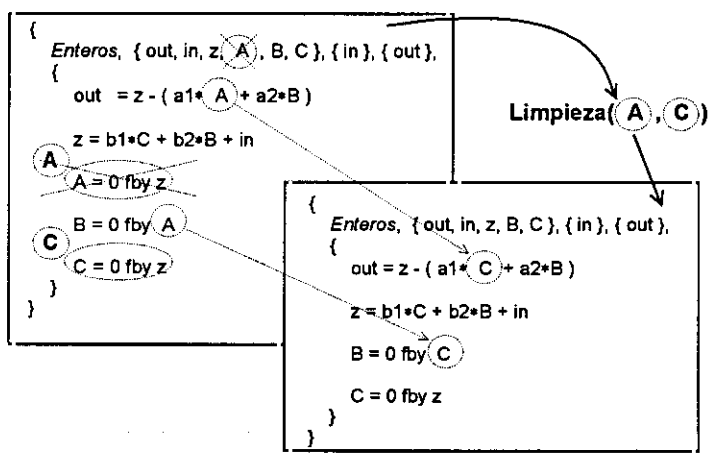


Fig. 3.7: Regla de limpieza.

Regla de limpieza de definiciones recursivas.

Esta es la sexta y última regla estructural, que viene a solventar un problema que posee la regla de limpieza debido a la naturaleza recursiva del mecanismo de especificación ecuacional. Si solamente se permitiera la regla de limpieza anterior, ninguna de las dos definiciones $z = 0$ *fby* z y $x = 0$ *fby* x podría limpiarse, aún siendo semánticamente equivalentes (ambas denotan la secuencia infinita de ceros) y, salvo renombrado, equivalentes desde el punto de vista sintáctico.

De este modo, esta regla establece que cualquier definición de una señal que no sea puerto de salida, puede ser eliminada del cuerpo de una especificación ecuacional, si cada una de las apariciones de la señal se reemplaza por una aparición de otra señal que esté definida, salvo renombrado, de igual modo. Una aplicación particular puede encontrarse en la fig. 3.8.

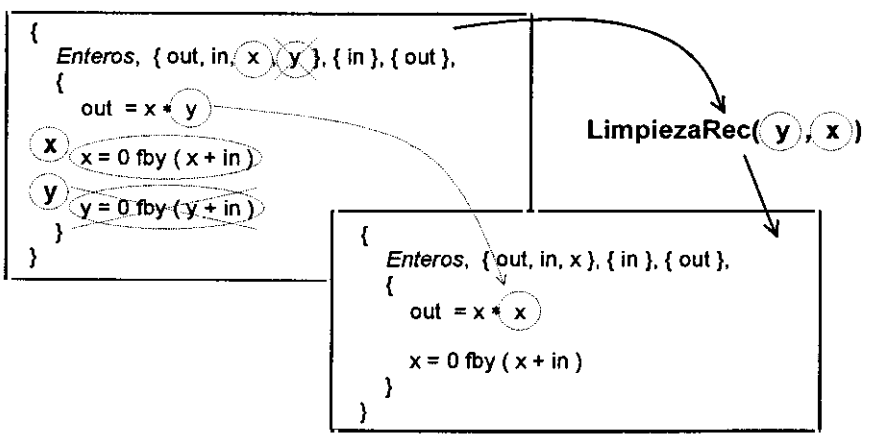


Fig. 3.8: Regla de limpieza recursiva.

Regla de limpieza de definiciones recursivas:

$$\frac{(\ \Sigma,\ X \cup \{x_1\},\ I,\ O,\ \varphi \cup \{\ x_1 = \varphi(x_1),\ x_2 = \varphi(x_2)\ \}\ )}{(\ \Sigma,\ X,\ I,\ O,\ \hat{\rho} \circ \varphi \cup \{\ x_2 = \hat{\rho}(\varphi(x_2))\ \}\ )} \qquad \begin{array}{l} x_1 \notin O \wedge \rho = \{x_1/x_2\} \wedge \\ \wedge\ \varphi(x_2) \equiv \hat{\rho}(\varphi(x_1)) \end{array}$$

Obsérvese que esta regla podría reemplazar a la de limpieza ya que, si una definición no contiene ocurrencias de la señal que define, la aplicación de la substitución ρ no tiene efecto, es decir, si $\forall u \in pos(\varphi(x_1)),\ \varphi(x_1)[u] \neq x_1$ se cumple que $\hat{\rho}(\varphi(x_1)) \equiv \varphi(x_1)$, por lo que el conjunto de condiciones que permiten aplicarla queda reducido al de la regla de anterior. Sin embargo no se ha hecho por conveniencia, por considerar que es más fácil encontrar el significado a esta regla mediante su contraste con las deficiencias de la anterior.

Obsérvese también que no es necesario añadir más reglas para limpiar una colección de definiciones que estén involucradas en un lazo mutuamente recursivo. Para hacer esto basta con encadenar la regla de limpieza de definiciones recursivas con otras reglas estructurales.

EJEMPLO 3.4

Sea la siguiente especificación ecuacional:

```
... la signatura Enteros ...
signals
out, in, x, y, t1, t2 :  Ent
inports
in
outports
out
body
out = x*y
x = 0 fby ( t1 + in )
y = 0 fby ( t2 + in )
t1 = 0 fby x
t2 = 0 fby y
```

Si aplicamos las regla de substitución en las posición 2.1 sobre las definiciones de *x* e *y*, y eliminamos *t*1 y *t*2, ya es posible aplicar la regla de limpieza recursiva sobre *x* e *y* quedando la especificación como:

... *la signatura Enteros* ...
signals
out, in, x : Ent
inports
in
outports
out
body
out = x*x
x = 0 fby (0 fby x + in)

La aplicación de esta regla para limpiar cierta señal x_1 por otra definida recursivamente de igual modo x_2 se expresará por:

LimpiezaRec(x_1, x_2 **)**

3.1.3 Reglas conductuales de transformación de especificaciones ecuacionales.

El objetivo de las reglas conductuales es transformar la manera en que una especificación ecuacional define los cálculos a realizar: ya sea cambiando los operadores que se utilizan, o alterando las dependencias de datos entre dichos operadores.

Para asegurar que una transformación de este tipo conserva el comportamiento, es necesario que la sintaxis tenga un soporte semántico que permita decidir qué cálculos son equivalentes. Dado que, en cualquier ámbito de aplicación, la manera más general de expresar equivalencias es mediante el uso de ecuaciones algebraicas, las transformaciones conductuales

establecerán un mecanismo de manipulación simbólica que permitirá aplicar sobre una especificación ecuacional conocimientos formalizados mediante ecuaciones (en el más estricto sentido matemático).

En el capítulo 2 (definición 2.12) se presentó, dada una signatura Σ, una formalización de ecuación útil para expresar conocimientos válidos en cualquier Σ-álgebra. Sin embargo, dada la naturaleza infinita del modelo que sustenta a toda señal de una especificación ecuacional (un elemento de cierta Σ-álgebra Lu-extendida), dicha formalización puede no ser lo suficientemente expresiva si se desean describir conceptos recursivos, esto es, si se desea referenciar algún elemento del álgebra soporte que sea el *punto fijo* de cierta función.

Por ello, es necesario extender aquella formalización y permitir que los términos que la formen posean, al menos, una variable ligada sobre la que construir la recursividad y ampliar la noción de interpretación para que el valor denotado por uno (o los dos) términos de una ecuación pueda ser el **punto fijo** que se desea especificar.

3.6 **Definición.** Sea una signatura heterogénea Σ y una familia de variables X. Se define **Lu(Σ)-término recursivo de género *s*** como el par (z, t) donde z es un símbolo no incluido en X ni en Σ y $t \in T_{Lu(\Sigma),s}(X \cup \{z\})$. Dada una Σ-álgebra A y una valoración $\mu : X \to Lu(A)$ su interpretación se define como:

$$\hat{\mu}((z, t)) = fix(\lambda z : Lu(A)_s . \hat{\mu}_{rec}(t) : Lu(A))$$

donde $\hat{\mu}_{rec}$ se define recursivamente sobre la estructura del término como:

- $\hat{\mu}_{rec}(x) = \mu(x), \forall x \in X$
- $\hat{\mu}_{rec}(z) = z$
- $\hat{\mu}_{rec}(\sigma) = Lu(A)^{\varepsilon,s}_{\sigma}$
- $\hat{\mu}_{rec}(\sigma(t_1, ..., t_n)) = Lu(A)^{w,s}_{\sigma}(\hat{\mu}_{rec}(t_1), ..., \hat{\mu}_{rec}(t_n)) \ \forall \sigma \in \Sigma_{w,s}$

Obsérvese que z es un símbolo ligado (no es una variable) y que a efectos del término t puede ser considerado como una constante. Además si el término t no posee ocurrencias de z, un $Lu(\Sigma)$-término recursivo puede

reducirse a un $Lu(\Sigma)$-término y su interpretación a la interpretación definida en 2.11. Además, nótese que esta definición restringe explícitamente la recursividad a una por término, decisión que, si bien cabe pensar que limita los conceptos expresables, es suficiente (según mi experiencia y tal como demostraré en el capítulo 4) para formalizar todos los conceptos que sustentan a la síntesis de alto nivel, y es necesaria para que cualquier término recursivo pueda identificarse con una única definición del cuerpo de una especificación ecuacional: si el término permitiera más de una recursividad sería necesario ajustar simultáneamente varias definiciones para aplicar el conocimiento que formaliza†. No obstante, si estas razones no parecieran suficientes, no habría problema en ampliar la definición 3.6 para que el término *t* pueda ser también recursivo.

Es posible generalizar la definición de ecuación (definición 2.12) de manera que uno o ambos de sus términos sea recursivo. La noción de validez establecida por la definición 2.13 sigue siendo aplicable a dichas ecuaciones recursivas si se utiliza la noción de interpretación de un término recursivo establecida en la definición 3.6.

EJEMPLO 3.5

Sea la signatura del ejemplo 2.4. Formalicemos, mediante una ecuación de género *Ent* que posee un término recursivo, el siguiente hecho hardware: si conectamos la salida de un retardador inicializado a 0 a su entrada, éste, en todo ciclo, muestra dicho valor inicial:

(∅, 0, (z, 0 fby z)) ó, abreviadamente 0 = (z, 0 fby z)

Comprobemos (ignorando géneros) si es efectivamente válida en el álgebra de secuencias de enteros y, dado que ambos términos son cerrados, no será necesario considerar ninguna valoración para realizarla.

† Se discutirá con más detalle en la definición de las reglas que aplican ecuaciones con términos recursivos.

$\hat{\mu}$((z, 0 fby z)) =

= *fix*(λz. $\hat{\mu}_{rec}$(0 fby z)) = *definición 3.6*

= *fix*(λz. λt. *if* t=1 *then* $\hat{\mu}_{rec}$(0)(1) *else* $\hat{\mu}_{rec}$(z)(t-1)) = *definición de* μ_{rec}

= *fix*(λz. λt. *if* t=1 *then* 0 *else* z(t-1)) = ... = *definición de* μ_{rec}

= « 0, 0, 0, ... » *calculando el punto fijo*

Como puede verse, este hecho es imposible expresarlo con una *Lu*(Σ)-ecuación convencional si no se utiliza expresamente un símbolo que denote al operador punto fijo (cosa que deseo evitar para facilitar el uso de las ecuaciones).

Regla de aplicación de izquierda a derecha de ecuaciones.

La primera regla conductual establece que cualquier *Lu*(Σ)-ecuación válida para la *Lu*(Σ)-álgebra que soporte a una especificación ecuacional podrá ser aplicada de izquierda a derecha sobre cualquier definición de su cuerpo siempre y cuando no añada símbolos no definidos.

Dado que el mecanismo utilizado para aplicarla será la reescritura (véase definición 3.5), es necesario que sea una *Lu*(Σ)-ecuación para poder ajustar sintácticamente el término izquierdo de la ecuación con la definición apropiada. Para asegurar que no se añaden símbolos no definidos basta con comprobar que el conjunto de variables del término derecho sea un subconjunto del conjunto de variables del término izquierdo†.

---

† La misma restricción que se hizo en la definición 3.5 para limitar el tipo de reglas de reescritura (y de ecuaciones) que eran utilizables.

Regla de aplicación de izquierda a derecha de ecuaciones:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1)\ \}\ )}{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1)[u \leftarrow \hat{\rho}(t_R)]\ \}\ )} \quad Lu(A) = (Y, t_L, t_R) \wedge var(t_R) \subseteq var(t_L) \wedge \wedge u \in pos(\varphi(x_1)) \wedge \exists \rho \mid \varphi(x_1)/u \equiv \hat{\rho}(t_L)$$

Obsérvese que la condición de existencia de cierta substitución ρ, tal que aplicada al lado izquierdo de la ecuación $\hat{\rho}(t_L)$, sea equivalente a algún subtérmino de alguna definición $\varphi(x_1)/u$, es que exista alguna definición que se ajuste al lado izquierdo de la ecuación, en cuyo caso la definición transformada será la que se obtenga por reescritura de la original vía la regla $t_L \rightarrow t_R$. (obtenida por la orientación de izquierda a derecha de la ecuación que se aplica)

Por otro lado, es importante destacar que debe ser la misma álgebra la que dé soporte tanto a la especificación ecuacional como a la ecuación, y que la demostración de que la ecuación es válida para dicha álgebra es algo a realizar externamente a la regla y al propio sistema de transformación. Esto hace que el lector pueda plantearse que la idea es formalmente admisible pero impracticable por requerir que los diseñadores demuestren qué $Lu(\Sigma)$-ecuaciones son válidas para cierta Σ-álgebra Lu-extendida.

Sin embargo, la realidad de la síntesis de alto nivel puede ser más simple de lo que parece.

- Primero, porque en el capítulo 4 se establecerá un conjunto de fórmulas que resumen el conocimiento de los aspectos básicos que conciernen a la síntesis de alto nivel. En dicho capítulo se mostrará, además, cómo expresarlas como $Lu(\Sigma)$-ecuaciones y cómo utilizar las reglas de aplicación para alcanzar un gran número de circuitos equivalentes con distintos rendimientos. Por ello, puede suponerse que estas reglas estarán almacenadas en la base de conocimiento del eventual sistema de síntesis formal.

- Segundo, gracias al principio de *Lu*-extensión (formalizado en el capítulo 2), la complejidad de algunas de las demostraciones se reduce notablemente al eliminar todos los conceptos añadidos durante las sucesivas extensiones del modelo estático, lo cual facilita la potencial labor del diseñador.
- Tercero, en el capítulo 6 se comprobará cómo esa demostración puede incluirse en el sistema de síntesis general y realizarse de modo casi automático bajo ciertas restricciones.

La aplicación de izquierda a derecha de una ecuación *e*, sobre la posición *u* de la definición de cierta señal x_1 será expresada utilizando la notación:

AplicacionID(x_1, u, *e* **)**

y un ejemplo de dicha aplicación se muestra en la fig. 3.9.

Regla de aplicación de derecha a izquierda de ecuaciones.

Esta segunda regla conductual es análoga a la anterior y permite aplicar una ecuación en el sentido contrario.

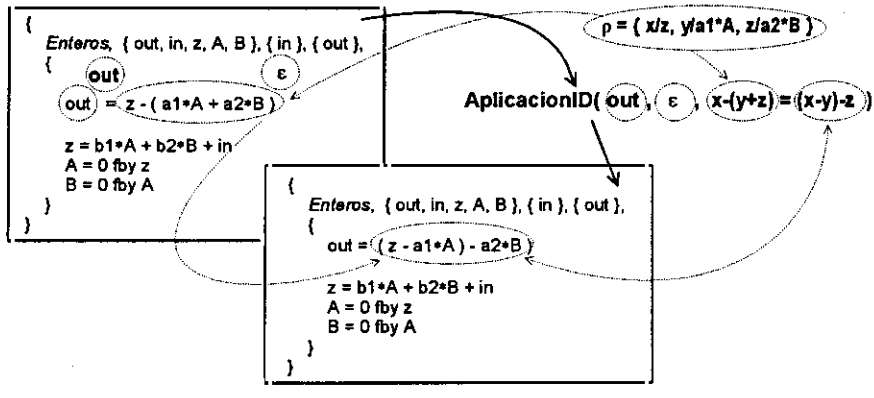


Fig. 3.9: Regla de aplicación de izq. a der.

Regla de aplicación de derecha a izquierda de ecuaciones:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1)\ \}\ )}{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1)[u \leftarrow \hat{\rho}(t_L)]\ \}\ )} \quad Lu(A) \vdash (Y, t_L, t_R) \wedge var(t_L) \subseteq var(t_R) \wedge u \in pos(\varphi(x_1)) \wedge \exists \rho \mid \varphi(x_1)/u = \hat{\rho}(t_R)$$

Obsérvese que gracias a la separación entre ecuaciones sintácticas y fórmulas ecuacionales semánticas el sistema de transformación presentado es válido independientemente del área en que se trabaje. Será el conjunto de ecuaciones mediante las que se formalice el conocimiento particular de un problema el que fijará la mayor o menor habilidad del sistema en transformar adecuadamente la especificación ecuacional según ciertos criterios. Que sea un mecanismo sintáctico de transformación hace, además, que sea directamente implantable en un computador, ya que especifica lo único que un computador puede hacer, esto es, manipular símbolos que representan conceptos que 'él desconoce'.

Un ejemplo de la aplicación de esta regla se muestra en la fig. 3.10. Si *e* es la ecuación que se aplica de derecha a izquierda, sobre la posición *u* de la definición de cierta señal x_1 esta regla se expresa como:

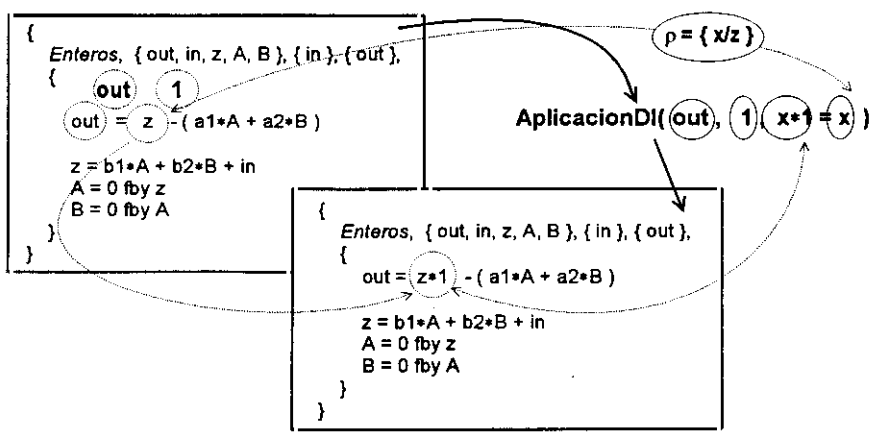


Fig. 3.10: Regla de aplicación de der. a izq..

AplicacionDI(x_1, u, e)

Regla de aplicación de izquierda a derecha de ecuaciones con un término recursivo.

La tercera regla conductual permite, al estilo de las dos anteriores, aplicar una ecuación. En este caso de izquierda a derecha y suponiendo que el término derecho es recursivo.

Regla de aplicación de izquierda a derecha de ecuaciones con un término recursivo:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1)\ \}\ )}{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \hat{\rho}(\hat{\sigma}(t_R))\ \}\ )} \quad Lu(A) \vdash (Y, t_L, (z, t_R)) \wedge var(t_R) \subseteq var(t_L) \wedge \wedge \exists \rho \mid \varphi(x_1) \equiv \hat{\rho}(t_L) \wedge \sigma = \{z/x_1\}$$

Obsérvese que se ha supuesto, sin pérdida de generalidad, que el término derecho es el recursivo. Podría asumirse que lo fuera el izquierdo, pero en ese caso esta regla sería equivalente a la siguiente y la siguiente a esta.

Por otro lado, obsérvese que a la hora de ajustar el término recursivo con la definición elegida, aparece una nueva componente: la aplicación σ. Si se estudia el conjunto de vínculos que la caracteriza, $\{z/x_1\}$, se podrá comprobar que su función es reemplazar toda ocurrencia en t_R de la variable muda z, por ocurrencias de la señal x_1 sobre la que se aplica la ecuación. Con esto se consigue que al ajustar el término se identifique al mismo tiempo x_1 como la nueva variable sobre la que construir la recursividad. Esta aplicación no es formalmente una substitución como la definida en 3.3 ya que según la definición 3.6, z no debe ser considerada una variable sino una constante muda, sin embargo he utilizado la misma notación ya que lo que se está definiendo es también un conjunto de reemplazamientos. Por esa misma razón, es necesario destacar que el ajuste ρ no debe intentar enlazar el símbolo x_1 ya que es el renombrado del símbolo mudo z.

Además, nótese que el requisito de que el término t_L sea no recursivo (según la definición 3.6) permite que pueda ajustarse con una única definición, si t_L fuera recursivo, necesitaríamos ajustar simultáneamente tantas definiciones de φ como recursividades contuviera el lado derecho de la ecuación. Esto es así ya que cada definición de un cuerpo ecuacional define una única función.

Para finalizar obsérvese que esta regla sólo permite ajustar la definición completa, por lo que no aparece ninguna posición *u* que localice un subtérmino particular de la definición como en las anteriores reglas.

La aplicación de esta regla, siendo *e* una ecuación con un término recursivo y siendo x_1 un identificador de señal, se expresa mediante:

AplicacionRecID(x_1, *e* **)**

Regla de aplicación de derecha a izquierda de ecuaciones con un término recursivo.

La cuarta regla conductual es análoga a la tercera, permitiendo, ahora, aplicar la ecuación en el sentido contrario.

Regla de aplicación de derecha a izquierda de ecuaciones con un término recursivo:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1) \})}{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \hat{\rho}(t_L) \})} \quad Lu(A) \vdash (Y, t_L, (z, t_R)) \wedge var(t_L) \subseteq var(t_R) \wedge \wedge \sigma=\{z/x_1\} \wedge \exists \rho \mid \varphi(x_1) \equiv \hat{\rho}(\hat{\sigma}(t_R))$$

Nótese cómo también en este caso siempre se ajusta la definición completa (desde la raíz) y cómo asume, a la hora de ajustar el término recursivo, la semántica recursiva de la especificación ecuacional.

La aplicación de esta regla, siendo *e* una ecuación con un término recursivo y siendo x_1 un identificador de señal, se expresa mediante:

AplicacionRecDI(x_1, e)

Otras reglas de aplicación incluidas por ortogonalidad.

Aunque no se han utilizado para ninguna de las técnicas de síntesis estudiadas y cuyos resultados se presentan en los capítulos 3, 4 y 5 de la presente memoria, conviene por ortogonalidad y previendo futuros desarrollos incluir un par más de ellas: las reglas de aplicación de ecuaciones cuyos términos sean ambos recursivos.

Regla de aplicación de izquierda a derecha de ecuaciones con ambos términos recursivos:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1)\ \}\ )}{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \hat{\rho}(\hat{\sigma}(t_R))\ \}\ )} \quad Lu(A) \vdash (Y,(z,t_L),(z,t_R)) \wedge var(t_R) \subseteq var(t_L) \wedge \sigma=\{z/x_1\} \wedge \exists \rho \mid \varphi(x_1)=\hat{\rho}(\hat{\sigma}(t_L))$$

Regla de aplicación de derecha a izquierda de ecuaciones con ambos términos recursivos:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1)\ \}\ )}{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \hat{\rho}(\hat{\sigma}(t_L))\ \}\ )} \quad Lu(A) \vdash (Y,(z,t_L),(z,t_R)) \wedge var(t_L) \subseteq var(t_R) \wedge \sigma=\{z/x_1\} \wedge \exists \rho \mid \varphi(x_1)=\hat{\rho}(\hat{\sigma}(t_R))$$

Ambas, si e es la ecuación y x_1 la señal sobre cuya definición se aplica e, estarían representadas, respectivamente, por:

AplicacionRecRecID(x_1, e)

AplicacionRecRecDI(x_1, e)

Regla de reemplazo de comodines.

Para finalizar, una regla conductual que permite utilizar las indiferencias que aparezcan en la especificación ecuacional. Esta regla establece que siempre que exista un comodín, podrá ser reemplazado sin peligro por cualquier otro término que denote cualquier concepto. Será tarea del diseñador decidir por qué término substituirlo.

Regla de reemplazo de comodines:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1) \})}{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1)[u \leftarrow t] \})} \quad \exists u \in pos(\varphi(x_1)),\ \varphi(x_1)[u] \equiv \# \wedge \wedge\ t \in T_{Lu(\Sigma)}(X\text{-}Outs)$$

La aplicación de esta regla se ejemplifica en la fig. 3.11. Toda aplicación de la regla de reemplazo de un comodín ubicado en la posición u de la definición de la señal x_1 por el término t será, a partir de ahora, expresada por:

Reemplazo(x_1, u, t)

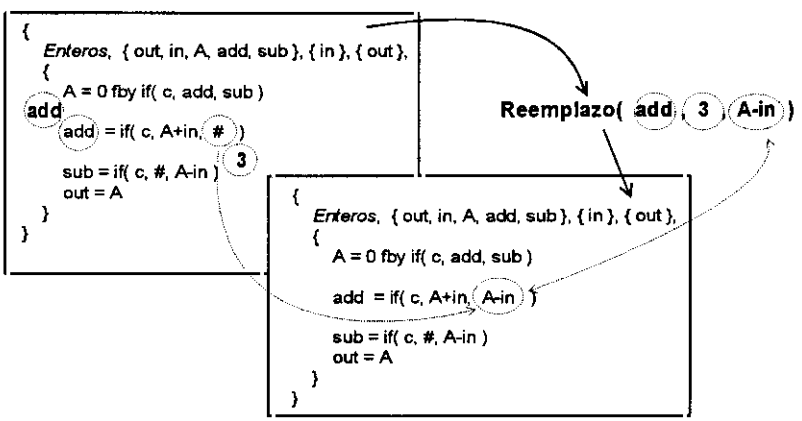


Fig. 3.11: Regla de reemplazo de comodines.

3.2 Un sistema para la síntesis formal por derivación.

Las anteriores reglas constituyen un sistema mínimo de inferencia para la transformación de especificaciones ecuacionales y especifican axiomáticamente el comportamiento e interfaz del núcleo básico de algoritmos de un sistema de síntesis formal. El patrón de comportamiento que siguen dichos algoritmos es similar: comprueban las condiciones de disparo de la regla y las precondiciones de la descripción inicial; si no se cumplen, no hacen nada y si se cumplen, construyen una descripción que verifique las postcondiciones. El interfaz lo constituyen aquellos elementos que permiten referirse a una transformación concreta (por ejemplo, la regla de renombrado define cómo realizar uno genérico, mientras que una operación de renombrado concreto cambiará el nombre de una señal específica).

A continuación se define lo que entenderemos por un proceso de síntesis formal: una concatenación de aplicaciones de cualquiera de las reglas presentadas en la anterior sección.

3.7 **DEFINICIÓN.** Se dice que la especificación ecuacional ds_2 es **derivable** a partir de la especificación ecuacional ds_1 vía la regla de transformación RT, escribiéndose $ds_1 \rightarrow_{RT} ds_2$, si ds_2 puede obtenerse aplicando RT sobre ds_1, donde RT forma parte del conjunto de reglas de transformación presentadas en §3.1.2 y §3.1.3.

3.8 **DEFINICIÓN.** Un **proceso de síntesis formal por derivación** es una secuencia finita de especificaciones ecuacionales tal que cada una de ellas (excepto la primera) es derivable a partir de la que la precede.

3.2.1 Equivalencia y compatibilidad de conductas.

Para poder demostrar que un proceso de síntesis formal es correcto, es necesario establecer qué conceptos no deben alterarse durante dicho proceso y demostrar efectivamente que ninguna de las transformaciones que lo forman, lo hace. Obviamente, en todo proceso de síntesis conductual, se desea que el comportamiento del circuito diseñado sea equivalente a la especificación de partida. Por ello, seguidamente, se definen dos conceptos que, basados en la noción de traza, nos permiten comparar especificaciones ecuacionales. En ellos se establece que si no podemos distinguir a dos descripciones en base a sus trazas (independientemente de cómo se calculen), pueden ser consideradas equivalentes.

3.9 **DEFINICIÓN.** Dos especificaciones ecuacionales $ds_1 \equiv (\Sigma, X_1, I, O, \varphi_1)$ y $ds_2 \equiv (\Sigma, X_2, I, O, \varphi_2)$ son **equivalentes en conducta** para un conjunto común de señales $Z \subseteq X_1 \cap X_2$, escribiéndose $ds_1 =_Z ds_2$, si para toda traza de la primera especificación, μ_1, existe una traza de la segunda especificación, μ_2, tal que ambas trazas son equivalentes si se restringen al conjunto Z. Es decir, siendo A una Σ-álgebra se tiene:

$$(\Sigma, X_1, I, O, \varphi_1) =_Z (\Sigma, X_2, I, O, \varphi_2) \Leftrightarrow$$

$$\Leftrightarrow \forall \mu_1 : X_1 \to Lu(A),\ \exists \mu_2 : X_2 \to Lu(A),\ \forall z \in Z,\ \mu_1(z) = \hat{\mu}_1(\varphi_1(z)) = \hat{\mu}_2(\varphi_2(z)) = \mu_2(z)$$

Obsérvese que en esta definición la igualdad clave es $\hat{\mu}_1(\varphi_1(z)) = \hat{\mu}_2(\varphi_2(z))$ que quiere decir que dos especificaciones serán equivalentes si para toda señal z del conjunto elegido, las interpretaciones de sus correspondientes definiciones son iguales, independientemente de cómo se calculen las trazas (mediante $\varphi_1(z)$ y $\varphi_2(z)$, respectivamente). Las otras dos igualdades $\mu_i(z) = \hat{\mu}_i(\varphi_i(z))$ son consecuencias de la definición de traza (véase definición 2.29).

Nótese también que la anterior definición permite explicitar qué señales deben ser equivalentes. La razón es que las señales de un circuito

completamente diseñado y las de su especificación original pueden ser completamente distintas, por ello es necesario poder restringir cuáles de ellas son realmente interesantes. Desde un punto de vista práctico, sólo suele interesar la equivalencia conductual como **caja negra**, en donde el conjunto común de señales es el formado por los puertos de entrada y de salida. No obstante, a veces, puede ser interesante considerar algunas equivalencias como **caja blanca**. Por ejemplo la equivalencia arquitectónica, entendiendo por tal la que toma como conjunto de señales común aquel formado por los puertos y por las salidas de los registros arquitectónicos.

Debe destacarse que para poder comparar dos especificaciones ecuacionales es necesario, como puede verse en la definición, que ambas utilicen la misma signatura, que ambas tengan la misma álgebra soporte y el mismo conjunto de puertos de entrada y salida. Ninguna de estas restricciones es irrealista ni arbitraria. La equivalencia de puertos se debe respetar en todo flujo de diseño para asegurar que el circuito diseñado puede adaptarse sin problemas al entorno para el que fue especificado. La equivalencia de signaturas es necesaria para asegurar un entorno simbólico coherente (teóricamente no es estrictamente necesario, pero no se obtiene ninguna ventaja práctica si no se permite, ya que cualquier problema puede resolverse ampliando la signatura común). La equivalencia de álgebras es necesaria para que los valores de las trazas puedan compararse sin ambigüedad (supóngase comparar las trazas de una especificación cuya álgebra soporte sean los reales y otra cuya álgebra sean vectores de bits de cierta anchura, para hacerlo serían necesarios conceptos más complicados tales como el de margen aceptable de error de representación) . Para eliminar esta restricción (que si que tiene aplicaciones prácticas) será necesario el concepto de morfismo algebraico que será utilizado en el capítulo 6.

La definición de equivalencia conductual es demasiado fuerte para muchos casos prácticos de diseño: obliga a que las trazas sean completamente

iguales. En el capítulo 2, §2.4.2, se introdujo el elemento comodín para denotar indiferencia, éste podía ser utilizado por el diseñador para expresar que el resultado concreto de cierto cálculo no afecta a los valores realmente importantes que calcula un circuito. Por ello, si la traza de una especificación en algunos instantes temporales posee comodines, la anterior definición exigiría que cualquier otra especificación derivada también los tuviera, siendo imposible utilizar el grado de libertad que facilitan dichos elementos.

La nueva definición permitirá aceptar como conductualmente equivalentes a dos especificaciones que no se contradigan, es decir, que sean iguales en todos aquellos instantes temporales en los que transporten valores concretos, pudiendo ser distintas en aquellos ciclos en los que una de ellas transporte un comodín. Esto es lo mismo que decir, según la definición 2.15, que las trazas sean compatibles.

3.10 Definición. Dos especificaciones ecuacionales $ds_1 \equiv (\Sigma, X_1, I, O, \varphi_1)$ y $ds_2 \equiv (\Sigma, X_2, I, O, \varphi_2)$ son **débilmente equivalentes en conducta** o, simplemente compatibles, para un conjunto común de señales $Z \subseteq X_1 \cap X_2$, escribiéndose $ds_1 \approx_Z ds_2$, si para toda traza de la primera especificación, μ_1, existe una traza de la segunda especificación, μ_2, tal que ambas trazas son compatibles si se restringen al conjunto Z. Es decir, siendo A una Σ-álgebra se tiene:

$$(\Sigma, X_1, I, O, \varphi_1) \approx_Z (\Sigma, X_2, I, O, \varphi_2) \Leftrightarrow$$

$$\Leftrightarrow \forall \mu_1 : X_1 \to Lu(A),\ \exists \mu_2 : X_2 \to Lu(A),\ \forall z \in Z,\ \mu_1(z) = \hat{\mu}_1(\varphi(z)) \approx \mu_2(z) = \hat{\mu}_2(\varphi(z))$$

Nótese que en la definición 2.15 sólo se definió la noción de compatibilidad sobre elementos atómicos, pero que es gracias al principio de *Lu*-extensión por lo que puede también aplicarse punto a punto sobre secuencias.

Obsérvese que dos especificaciones equivalentes en conducta también son débilmente equivalentes en conducta, es decir:

$$(\Sigma, X_1, I, O, \varphi_1) =_Z (\Sigma, X_2, I, O, \varphi_2) \Rightarrow (\Sigma, X_1, I, O, \varphi_1) \approx_Z (\Sigma, X_2, I, O, \varphi_2)$$

3.2.2 Corrección del sistema de síntesis formal.

En esta sección se aborda el aspecto más importante del presente capítulo: los teoremas que establecen la corrección del sistema formal presentado. Mediante ellos se demuestra que todas las transformaciones mostradas en §3.1.2 y §3.1.3 preservan la conducta como caja negra de las especificaciones ecuacionales. Este resultado se desglosa en dos teoremas: el 3.11, para todas las reglas excepto la de reemplazo de comodines por conservar fuertemente la conducta, y el 3.13, para la regla de reemplazo ya que es la única que conserva la conducta débilmente.

3.11 **TEOREMA.** Toda derivación (excepto la generada por la regla de reemplazo de comodines) es correcta, si entendemos por corrección la equivalencia en conducta como caja negra de la especificación original y de la especificación derivada.

DEMOSTRACIÓN. La demostración se descompone en el chequeo de la corrección de cada regla por separado y se realiza de manera constructiva. Es decir, para toda traza de la especificación original se describe otra traza de la especificación derivada que, restringida a ciertas señales, es equivalente a la dada. Pero, antes de comenzar, es necesario demostrar un lema que simplificará las posteriores demostraciones.

3.12 **LEMA.** Sea A una Σ-álgebra, X una familia de variables, μ una valoración y t, t' y t'' tres términos incluidos en $T_\Sigma(X)$. Se cumple que:

$$\text{si } \hat{\mu}(t') = \hat{\mu}(t'') \wedge \exists u \in pos(t) \mid t/u = t' \Rightarrow \hat{\mu}(t[u \leftarrow t']) = \hat{\mu}(t[u \leftarrow t''])$$

DEMOSTRACIÓN. Se realiza por inducción sobre la estructura de t.

Caso base, $u \equiv \varepsilon$:

$\hat{\mu}(t[\varepsilon \leftarrow t'])$ =

$= \hat{\mu}(t')$ = *definición de reemplazamiento*

$= \hat{\mu}(t'')$ = *premisa*

$= \hat{\mu}(t[\varepsilon \leftarrow t''])$ *definición de reemplazamiento*

Paso de inducción, $u \equiv i.w$:

$\hat{\mu}(t[i.w \leftarrow t']) \equiv$

$\equiv \hat{\mu}(\sigma(t_1, ..., t_i, ..., t_n)[i.w \leftarrow t']) =$ *si* $u \equiv i.w \in Pos(t) \Rightarrow t[i] \equiv \sigma \in \Sigma_{w,s}$

$= A_\sigma^{w,s}(\hat{\mu}(t_1), ..., \hat{\mu}(t_i[w \leftarrow t']), ..., \hat{\mu}(t_n)) =$ *definición de* $\hat{\mu}$

$= A_\sigma^{w,s}(\hat{\mu}(t_1), ..., \hat{\mu}(t_i[w \leftarrow t'']), ..., \hat{\mu}(t_n)) =$ *hipótesis de inducción*

$= \hat{\mu}(\sigma(t_1, ..., t_i, ..., t_n)[i.w \leftarrow t'']) \equiv \hat{\mu}(t[i.w \leftarrow t''])$ *definición de* $\hat{\mu}$

□

En todas las demostraciones asumiré que $ds_1 \equiv (\Sigma, X_1, I, O, \varphi_1)$ es la especificación ecuacional original y que $ds_2 \equiv (\Sigma, X_2, I, O, \varphi_2)$ es la especificación ecuacional derivada. Obsérvese que Σ, I y O son iguales en ambas ya que ninguna regla de transformación modifica ni la signatura ni el conjunto de puertos. Además, supondré que la valoración $\mu_1 : X_1 \rightarrow Lu(A)$, es una traza de ds_1, donde A es una Σ-álgebra completamente genérica†. Para finalizar recuérdese que por ser μ una traza, según la definición 2.29, cumple:

$$\forall x \in X_1\text{-}I,\ \mu_1(x) = \hat{\mu}_1(\varphi(x))$$

(1) **Corrección de la regla de substitución**. Dado que esta regla no altera el conjunto de señales demostraré que:

$$ds_1 \rightarrow_{\text{Substitución}} ds_2 \ \Rightarrow\ ds_1 =_{X,} ds_2 \ \Rightarrow ds_1 =_{I \cup O} ds_2$$

Para ello, siendo x_2 la señal en cuya definición se substituye la ocurrencia de x_1 en la posición u, demostraré que μ_1 también es una traza de ds_2:

si $x \in X_1\text{-}I\text{-}\{x_2\}$ entonces $\hat{\mu}_1(\varphi_2(x)) =$

$= \hat{\mu}_1(\varphi_1(x)) =$ *la regla de substitución sólo altera la definición de* x_2

$= \mu_1(x)$ μ_1 *es una traza de* φ_1

si $x \equiv x_2$ entonces $\hat{\mu}_1(\varphi_2(x_2)) =$

$= \hat{\mu}_1(\varphi_1(x_2)[u \leftarrow \varphi_1(x_1)]) =$ *según la definición de la regla de substitución*

$= \hat{\mu}_1(\varphi_1(x_2)[u \leftarrow x_1]) =$ *(*) y Lema 3.12*

$= \hat{\mu}_1(\varphi_1(x_2)) =$ *precondición de ocurrencia de* x_1 *en* $\varphi_1(x_2)[u]$

$= \mu_1(x_2)$ μ *es un comportamiento de* φ_1

---

† Lo que hace que el sistema de transformación sea correcto independientemente del significado de los símbolos (álgebra soporte) y que sea, por tanto, un mecanismo de manipulación puramente simbólico.

(*) Al ser μ_1 una traza de φ_1 se cumple que $\hat{\mu}_1(\varphi_1(x_1)) = \mu_1(x_1)$. Además, al ser x_1 una señal, la definición 2.11 establece que $\mu_1(x_1) = \hat{\mu}_1(x_1)$. Así se concluye que $\hat{\mu}_1(\varphi_1(x_1)) = \hat{\mu}_1(x_1)$.

(2) **Corrección de la regla de renombrado**. Dado que esta regla modifica el conjunto de señales demostraré que, siendo x_1 la señal (no puerto) que cambia su nombre a x_2, se cumple:

$$ds_1 \rightarrow_{\text{Renombrado}} ds_2 \Rightarrow ds_1 =_{X_1-\{x1\}} ds_2 \Rightarrow ds_1 =_{I\cup O} ds_2$$

Para ello, demostraré que la valoración $\mu_2 : X_1-\{x_1\}\cup\{x_2\} \rightarrow Lu(A)$, definida como: $\forall x \in X_1-\{x_1\}$, $\mu_2(x) = \mu_1(x) \wedge \mu_2(x_2) = \mu_1(x_1)$, es una traza de ds_2:

si $x \in X_1-I-\{x_1\}$ entonces $\hat{\mu}_2(\varphi_2(x)) =$

$= \hat{\mu}_2(\hat{\rho}(\varphi_1(x))) =$ *según la definición de la regla de renombrado*

$= \hat{\mu}_1(\varphi_1(x)) =$ *(*)*

$= \mu_1(x) =$ *μ_1 es una traza de ds_1*

$= \mu_2(x)$ *según la definición de μ_2*

si $x \equiv x_2$ entonces $\hat{\mu}_2(\varphi_2(x_2)) =$

$= \hat{\mu}_2(\hat{\rho}(\varphi_1(x_1))) =$ *según la definición de la regla de renombrado*

$= \hat{\mu}_1(\varphi_1(x_1)) =$ *(*)*

$= \mu_1(x_1) =$ *μ_1 es una traza de ds_1*

$= \mu_2(x_2)$ *según la definición de μ_2*

(*) Para las trazas μ_1 y μ_2 y para la substitución ρ fijadas en esta prueba, demostraré, por inducción sobre la estructura del término t, que $\hat{\mu}_2(\hat{\rho}(t)) = \hat{\mu}_1(t)$:

Casos base:

$t \equiv \sigma \in \Sigma_{\varepsilon,s}$, $\hat{\mu}_1(\sigma) =$

$= Lu(A)_{\sigma}^{\varepsilon,s} =$ *según la definición 2.11*

$= \hat{\mu}_2(\sigma) =$ *según la definición 2.11*

$= \hat{\mu}_2(\hat{\rho}(\sigma))$ *según la definición 3.3*

$t \equiv x \in X_1-\{x_1\}$, $\hat{\mu}_1(x) =$

$= \mu_1(x) =$ *según la definición 2.11*

$= \mu_2(x) =$ *según la definición de μ_2*

$= \mu_2(\rho(x)) =$ *según la definición de ρ*

$= \hat{\mu}_2(\hat{\rho}(x))$ *según las definiciones 3.3 y 2.11*

$t \equiv x_1$, $\hat{\mu}_1(x_1) =$

$= \mu_1(x_1) =$ *según la definición 2.11*

$= \mu_2(x_2) =$ *según la definición de μ_2*

$= \mu_2(\rho(x_1)) =$ *según la definición de ρ*

$= \hat{\mu}_2(\hat{\rho}(x_1))$ *según las definiciones 3.3 y 2.11*

Paso de inducción, $t = \sigma(t_1, ..., t_n)$ con $\sigma \in \Sigma_{w,s}$ y $t_i \in T_{Lu(\Sigma)}(X_1)$

$\hat{\mu}_1(\sigma(t_1, ..., t_n)) =$

$= Lu(A)_{\sigma}^{w,s}(\hat{\mu}(t_1), ..., \hat{\mu}(t_n)) =$ *según la definición 2.11*

$= Lu(A)_{\sigma}^{w,s}(\hat{\mu}_2(\hat{\rho}(t_1)), ..., \hat{\mu}_2(\hat{\rho}(t_n))) =$ *hipótesis de inducción*

$= \hat{\mu}_2(\sigma(\hat{\rho}(t_1), ..., \hat{\rho}(t_n))) =$ *según la definición 2.11*

$= \hat{\mu}_2(\hat{\rho}(\sigma(t_1, ..., t_n)))$ *según la definición 3.3*

(3) **Corrección de la regla de expansión**. Esta regla modifica el conjunto de señales original añadiendo una nueva señal x_2. Así que dado que $X_1 \subset X_2$ demostraré que se cumple:

$$ds_1 \rightarrow_{\text{Expansión}} ds_2 \Rightarrow ds_1 =_{X_1} ds_2 \Rightarrow ds_1 =_{I \cup O} ds_2$$

Para ello, demostraré que la valoración $\mu_2 : X_1 \cup \{x_2\} \rightarrow Lu(A)$, definida como: $\forall x \in X_1$, $\mu_2(x) = \mu_1(x) \wedge \mu_2(x_2) = \hat{\mu}_1(\varphi_1(x_1)/u)$, donde x_1 es la señal cuya definición ha sido expandida en la posición u, es una traza de ds_2:

si $x \in X_1 - I - \{x_1\}$ entonces $\hat{\mu}_2(\varphi_2(x)) =$

$= \hat{\mu}_2(\varphi_1(x)) =$ *según la definición de la regla de expansión*

$= \hat{\mu}_1(\varphi_1(x)) =$ *$\varphi_1(x)$ no contiene ocurrencias de x_2 ya que $x_2 \notin X_1$*

$= \mu_1(x) =$ *μ_1 es una traza de ds_1*

$= \mu_2(x)$ *según la definición de μ_2*

si $x \equiv x_2$ entonces $\hat{\mu}_2(\varphi_2(x_2)) =$

$= \hat{\mu}_2(\varphi_1(x_1)/u) =$ *según la definición de la regla de expansión*

$= \hat{\mu}_1(\varphi_1(x_1)/u) =$ *$\varphi_1(x_1)$ no contiene ocurrencias de x_2 ya que $x_2 \notin X_1$*

$= \mu_2(x_2)$ *según la definición de μ_2*

si $x \equiv x_1$ entonces $\hat{\mu}_2(\varphi_2(x_1)) =$

$= \hat{\mu}_2(\varphi_1(x_1)[u \leftarrow x_2]) =$ *según la definición de la regla de expansión*

$= \hat{\mu}_2(\varphi_1(x_1)[u \leftarrow \varphi_1(x_1)/u]) =$ *(*) y Lema 3.12*

$= \hat{\mu}_2(\varphi_1(x_1)) =$ *según las definiciones 3.1 y 3.2*

$= \hat{\mu}_1(\varphi_1(x_1)) =$ *$\varphi_1(x_1)$ no contiene ocurrencias de x_2 ya que $x_2 \notin X_1$*

$= \mu_1(x_1) =$ *μ_1 es una traza de ds_1*

$= \mu_2(x_1)$ *según la definición de μ_2 ($x_1 \in X_1$)*

(*) En la definición de μ_2 se establece que $\mu_2(x_2) = \hat{\mu}_1(\varphi_1(x_1)/u)$. Dado que, según la precondición de la regla de expansión, $\varphi_1(x_1)$ no contiene ninguna ocurrencia de x_2, puede decirse que $\hat{\mu}_1(\varphi_1(x_1)/u) = \hat{\mu}_2(\varphi_1(x_1)/u)$. Además, al ser x_2 una señal, la definición 2.11 establece que $\hat{\mu}_2(x_2) = \mu_2(x_2)$. Con todo ello puede concluirse que $\hat{\mu}_2(\varphi_1(x_1)/u) = \hat{\mu}_2(x_2)$.

(4) **Corrección de la regla de eliminación**. Esta regla modifica el conjunto de señales original eliminando una señal x_1 (no puerto). Así que dado que $X_2 \subset X_1$ demostraré que se cumple:

$$ds_1 \rightarrow_{\text{Eliminación}} ds_2 \Rightarrow ds_1 =_{X_1-\{x1\}} ds_2 \Rightarrow ds_1 =_{I \cup O} ds_2$$

Para ello, demostraré que la valoración $\mu_2 : X_1-\{x_1\} \rightarrow Lu(A)$, definida como: $\forall x \in X_1-\{x_1\}$, $\mu_2(x) = \mu_1(x)$ es una traza de ds_2:

si $x \in X_1-I-\{x_1\}$ entonces $\hat{\mu}_2(\varphi_2(x)) =$

$= \hat{\mu}_2(\varphi_1(x)) =$ *según la definición de la regla de eliminación*

$= \hat{\mu}_1(\varphi_1(x)) =$ *si la regla se aplica, $\varphi_1(x)$ no contiene ocurrencias de x_1*

$= \mu_1(x) =$ *μ_1 es una traza de ds_1*

$= \mu_2(x)$ *según la definición de μ_2*

(5) **Corrección de la regla de limpieza**. Esta regla también modifica el conjunto de señales original eliminando una señal x_1 (no puerto). Así que dado que $X_2 \subset X_1$ demostraré que se cumple:

$$ds_1 \rightarrow_{\text{Limpieza}} ds_2 \Rightarrow ds_1 =_{X_1-\{x1\}} ds_2 \Rightarrow ds_1 =_{I \cup O} ds_2$$

Para ello, demostraré que la valoración $\mu_2 : X_1-\{x_1\} \rightarrow Lu(A)$, definida como: $\forall x \in X_1-\{x_1\}$, $\mu_2(x) = \mu_1(x)$ es una traza de ds_2:

si $x \in X-I-\{x_1\}$ entonces $\hat{\mu}_2(\varphi_2(x)) =$

$= \hat{\mu}_2(\hat{\rho}(\varphi_1(x)) =$ *según la definición de la regla de limpieza*

$= \hat{\mu}_1(\varphi_1(x)) =$ (*)

$= \mu_1(x) =$ *μ_1 es una traza de ds_1*

$= \mu_2(x)$ *según la definición de μ_2*

(*) Para las trazas μ_1 y μ_2 y para la substitución ρ fijadas en esta prueba, demostraré, por inducción sobre la estructura del término t, que $\hat{\mu}_2(\hat{\rho}(t)) = \hat{\mu}_1(t)$:

Casos base:

$t \equiv \sigma \in \Sigma_{\varepsilon,s}$, $\hat{\mu}_1(\sigma) =$

$= Lu(A)_{\sigma}^{\varepsilon,s} =$ *según la definición 2.11*

$= \hat{\mu}_2(\sigma) =$ *según la definición 2.11*

$= \hat{\mu}_2(\hat{\rho}(\sigma))$ *según la definición 3.3*

$t \equiv x \in X_1\text{-}\{x_1\}$, $\hat{\mu}_1(x) =$

$= \mu_1(x) =$ *según la definición 2.11*

$= \mu_2(x) =$ *según la definición de μ_2*

$= \mu_2(\rho(x)) =$ *según la definición de ρ*

$= \hat{\mu}_2(\hat{\rho}(x))$ *según las definiciones 3.3 y 2.11*

$t \equiv x_1$, $\hat{\mu}_1(x_1) =$

$= \mu_1(x_1) =$ *según la definición 2.11*

$= \hat{\mu}_1(\varphi_1(x_1)) =$ *μ_1 es una traza de ds_1*

$= \hat{\mu}_1(\varphi_2(x_2)) =$ *si la regla se aplica $\varphi_1(x_1) \equiv \varphi_2(x_2)$*

$= \hat{\mu}_1(x_2) =$ *μ_1 es una traza de ds_1*

$= \mu_1(x_2) =$ *según la definición 2.11*

$= \mu_2(x_2) =$ *según la definición de μ_2*

$= \mu_2(\rho(x_1)) =$ *según la definición de ρ*

$= \hat{\mu}_2(\hat{\rho}(x_1))$ *según las definiciones 3.3 y 2.11*

Paso de inducción, $t = \sigma(t_1, ..., t_n)$ con $\sigma \in \Sigma_{w,s}$ y $t_j \in T_{Lu(\Sigma)}(X_1)$

$\hat{\mu}_1(\sigma(t_1, ..., t_n)) =$

$= Lu(A)_{\sigma}^{w,s}(\hat{\mu}_1(t_1), ..., \hat{\mu}_1(t_n)) =$ *según la definición 2.11*

$= Lu(A)_{\sigma}^{w,s}(\hat{\mu}_2(\hat{\rho}(t_1)), ..., \hat{\mu}_2(\hat{\rho}(t_n))) =$ *hipótesis de inducción*

$= \hat{\mu}_2(\sigma(\hat{\rho}(t_1), ..., \hat{\rho}(t_n))) =$ *según la definición 2.11*

$= \hat{\mu}_2(\hat{\rho}(\sigma(t_1, ..., t_n)))$ *según la definición 3.3*

(6) **Corrección de la regla de limpieza de definiciones recursivas**. Esta regla es muy similar a la regla de limpieza, por lo que podré reutilizar parte de su demostración. La diferencia entre ambas reglas radica en las precondiciones, mientras que la de limpieza exige que $\varphi_1(x_2) \equiv \varphi_1(x_1)$, la de limpieza recursiva exige que $\varphi_1(x_2) \equiv \hat{\rho}(\varphi_1(x_1))$. Si se observa en qué parte de la demostración (5) se utiliza dicha precondición se observa que se hace en la demostración de uno de los casos base de la demostración por inducción estructural marcada con (*), concretamente en la demostración de que $\hat{\mu}_1(x_1) = \hat{\mu}_2(\hat{\rho}(x_1))$, en el paso en el que se prueba $\hat{\mu}_1(\varphi_1(x_1)) = \hat{\mu}_1(\varphi_1(x_2))$.

En este caso, para demostrar:

$$ds_1 \rightarrow_{\text{LimpiezaRec}} ds_2 \Rightarrow ds_1 =_{X-\{x1\}} ds_2 \Rightarrow ds_1 =_{\wedge O} ds_2$$

comprobaré que, conociendo $\varphi_1(x_2) \equiv \hat{\rho}(\varphi_1(x_1))$, se cumple directamente $\mu_1(x_1) = \mu_1(x_2)$, el resto de la demostración es igual que en (5).

Asumiendo que se conociera μ_1 para toda señal distinta de x_1, $\mu_1(x_1)$ sería equivalente, conociendo la semántica recursiva de las definiciones que forman toda especificación ecuacional, a la interpretación del término recursivo $(x_1, \varphi(x_1))$. El mismo razonamiento puede aplicarse a la señal x_2. Es decir, tenemos que $\mu_1(x_1) = \hat{\mu}_1((x_1,\varphi(x_1)))$ y que $\mu_1(x_2) = \hat{\mu}_1((x_2,\varphi(x_2)))$.

Por otro lado, si $\varphi_1(x_2) \equiv \hat{\rho}(\varphi_1(x_1))$ entonces $\forall u \in \{ pos(\varphi_1(x_1)) \mid \varphi_1(x_1)[u] \neq x_1 \}$ se tiene que $\varphi_1(x_1) \equiv \varphi_2(x_2)$. Es decir, que ambos términos son equivalentes excepto en la variable muda cuya traza es el punto fijo del término recursivo, por lo que puede concluirse que $\mu_1(x_1) = \mu_1(x_2)$.

(7) **Corrección de la regla de aplicación de izquierda a derecha de ecuaciones**. Dado que esta regla no altera el conjunto de señales demostraré que:

$$ds_1 \rightarrow_{\text{AplicacionID}} ds_2 \Rightarrow ds_1 =_{X_1} ds_2 \Rightarrow ds_1 =_{\wedge O} ds_2$$

Para ello, siendo x_1 la señal sobre cuya definición se aplica la ecuación en la posición u, demostraré que μ_1 también es una traza de ds_2:

si $x \in X_1\text{-}I\text{-}\{x_1\}$ entonces $\hat{\mu}_1(\varphi_2(x)) =$

$= \hat{\mu}_1(\varphi_1(x)) =$ *la regla de substitución sólo altera la definición de* x_1

$= \mu_1(x)$ *μ es un comportamiento de* φ_1

si $x \equiv x_1$ entonces $\hat{\mu}_1(\varphi_2(x_1)) =$

$= \hat{\mu}_1(\varphi_1(x_1)[u \leftarrow \hat{\rho}(t_R)]) =$ *según la definición de la regla de aplicación*

$= \hat{\mu}_1(\varphi_1(x_1)[u \leftarrow \hat{\rho}(t_L)]) =$ *(*) y Lema 3.12*

$= \hat{\mu}_1(\varphi_1(x_1)[u \leftarrow \varphi_1(x_1)/u]) =$ *según la precondición de que $\varphi(x_1)/u$ se ajuste a t_L*

$= \hat{\mu}_1(\varphi_1(x_1)) =$ *según las definiciones 3.1 y 3.2*

$= \mu_1(x_1)$ *μ es un comportamiento de* φ_1

(*) Si $Lu(A) \rightarrowtail t_L = t_R$, según la definición 2.13, se tiene que $Lu(A) \rightarrowtail \hat{\rho}(t_L) = \hat{\rho}(t_R)$, luego $\hat{\mu}(\hat{\rho}(t_L)) = \hat{\mu}(\hat{\rho}(t_R))$.

(8) **Corrección de la regla de aplicación de derecha a izquierda de ecuaciones**. Para demostrar que:

$$ds_1 \rightarrow_{\text{AplicacionDI}} ds_2 \Rightarrow ds_1 =_{X_1} ds_2 \Rightarrow ds_1 =_{I \cup O} ds_2$$

basta con cambiar t_L por t_R y t_R por t_L en la demostración (7).

(9) **Corrección de la regla de aplicación de izquierda a derecha de ecuaciones con un término recursivo**. Dado que esta regla no altera el conjunto de señales demostraré que:

$$ds_1 \rightarrow_{\text{AplicacionRecID}} ds_2 \Rightarrow ds_1 =_{X_1} ds_2 \Rightarrow ds_1 =_{I \cup O} ds_2$$

Para ello, siendo x_1 la señal sobre cuya definición se aplica la ecuación, demostraré que μ_1 también es una traza de ds_2.

si $x \in X_1\text{-}I\text{-}\{x_1\}$ entonces $\hat{\mu}_1(\varphi_2(x)) =$

$= \hat{\mu}_1(\varphi_1(x)) =$ *la regla de substitución sólo altera la definición de* x_1

$= \mu_1(x)$ *μ es un comportamiento de* φ_1

si $x \equiv x_1$ entonces $\hat{\mu}_1(\varphi_2(x_1)) =$

$= \hat{\mu}_1((x_1, \varphi_2(x_1))) =$ *según lo establecido en la demostración de (6)*

$= \hat{\mu}_1((x_1, \hat{\rho}(\hat{\sigma}(t_R)))) =$ *según la definición de la regla de aplicación*

$= \hat{\mu}_1(\hat{\rho}(t_L)) =$ *(*)*

$= \hat{\mu}_1(\varphi_1(x_1)) =$ *según la precondición de que $\varphi(x_1)$ se ajuste a t_L*

$= \mu_1(x_1)$ *μ es un comportamiento de* φ_1

(*) Si $Lu(A) \rightarrowtail t_L=(z,t_R)$, se tiene que $\hat{\rho}(t_L)=\hat{\rho}((z,t_R))$. Dado que z es una variable muda, es posible decir que $\hat{\rho}(t_L)=\hat{\rho}((x_1,\hat{\sigma}(t_R)))$. Finalmente ya que ρ es un ajuste en cuyo domino no aparece x_1 (ya que x_1 no es una variable), se concluye $\hat{\rho}(t_L)=(x_1,\hat{\rho}(\hat{\sigma}(t_R)))$.

(10) **Corrección de la regla de aplicación de derecha a izquierda de ecuaciones con un término recursivo**. Para demostrar:

$$ds_1 \rightarrow_{\text{AplicacionRecDI}} ds_2 \Rightarrow ds_1 =_{X,} ds_2 \Rightarrow ds_1 =_{I \cup O} ds_2$$

reutilizaré la primera parte de la demostración (10), la segunda, queda así:

si $x \equiv x_1$ entonces $\hat{\mu}_1(\varphi_2(x_1)) =$

$= \hat{\mu}_1(\hat{\rho}(t_R)) =$ *según la definición de la regla de aplicación*

$= \hat{\mu}_1((x_1,\hat{\rho}(\hat{\sigma}(t_R)))) =$ *según el argumento (*) de la demostración (9)*

$= \hat{\mu}_1((x_1,\varphi_1(x_1))) =$ *según la definición de la regla de aplicación*

$= \mu_1(x_1)$ *según lo establecido en la demostración de (6)*

(11) **Corrección de la regla de aplicación de izquierda a derecha de ecuaciones con ambos términos recursivos**. La demostración:

$$ds_1 \rightarrow_{\text{AplicacionRecRecID}} ds_2 \Rightarrow ds_1 =_{X,} ds_2 \Rightarrow ds_1 =_{I \cup O} ds_2$$

utiliza las misma ideas presentadas en las demostraciones (9) y (10). Reutilizando la primera parte de (9), la nueva segunda es:

si $x \equiv x_1$ entonces $\hat{\mu}_1(\varphi_2(x_1)) =$

$= \hat{\mu}_1((x_1,\varphi_2(x_1))) =$ *según lo establecido en la demostración de (6)*

$= \hat{\mu}_1((x_1,\hat{\rho}(\hat{\sigma}(t_R)))) =$ *según la definición de la regla de aplicación*

$= \hat{\mu}_1((x_1,\hat{\rho}(\hat{\sigma}(t_L)))) =$ *según el argumento (*) de la demostración (9)*

$= \hat{\mu}_1((x_1,\varphi_1(x_1))) =$ *según la precondición de que $\varphi(x_1)$ se ajuste a t_L*

$= \hat{\mu}_1(\varphi_1(x_1)) =$ *según lo establecido en la demostración de (6)*

$= \mu_1(x_1)$ *μ es un comportamiento de φ_1*

(*) Si $Lu(A) \rightarrowtail (z,t_L)=(z,t_R)$, se tiene que $\hat{\rho}((z,t_L)) = \hat{\rho}((z,t_R))$. Dado que z es una variable muda, es posible decir que $\hat{\rho}((x_1,\hat{\sigma}(t_L))) = \hat{\rho}((x_1,\hat{\sigma}(t_R)))$. Finalmente ya que ρ es un ajuste en cuyo domino no aparece x_1 (ya que x_1 no es una variable), se concluye $(x_1,\hat{\rho}(\hat{\sigma}(t_L))) = (x_1,\hat{\rho}(\hat{\sigma}(t_R)))$.

(12) **Corrección de la regla de aplicación de derecha a izquierda de ecuaciones**. Para demostrar que:

$$ds_1 \rightarrow_{\text{AplicacionRecRecDI}} ds_2 \Rightarrow ds_1 =_X ds_2 \Rightarrow ds_1 =_{I \cup O} ds_2$$

basta con cambiar t_L por t_R y t_R por t_L en la demostración (11).

□

3.13 TEOREMA. La regla de reemplazo de comodines es correcta, entendiendo por corrección la equivalencia débil en conducta como caja negra de la especificación original y de la especificación derivada.

DEMOSTRACIÓN. Esta demostración, a diferencia de las realizadas en el teorema 3.11, no es constructiva. La razón es que si no se conoce la especificación ecuacional concreta, los efectos de reemplazar un comodín por un término son impredecibles sobre la traza. Sin embargo, para demostrar:

$$ds_1 \rightarrow_{\text{Reemplazo}} ds_2 \Rightarrow ds_1 \approx_X ds_2 \Rightarrow ds_1 \approx_{I \cup O} ds_2$$

demostraré simplemente lo que exige la definición 3.10, que si μ_1 existe, siempre existe otra μ_2 compatible con la primera.

Así, supóngase que el comodín a ser reemplazado por cierto término t, está ubicado en la posición u de la definición de la señal x_1, esto es, $\varphi_1(x_1)[u] \equiv \#$. Según esto, cualquier traza μ_1 de ds_1 tendrá en algunos de sus ciclos elementos comodín ya que $\hat{\mu}_1(\varphi_1(x_1)/u) = \lambda t.\#$.

Si se reemplaza dicho comodín, la nueva definición cumplirá $\varphi_2(x_1)/u \equiv t$ y la traza μ_1 pasará a ser otra μ_2 que será igual a la original sólo que en algunos de los ciclos en que μ_1 valía #, μ_2 vale otro cierto valor, ya que ahora $\hat{\mu}_2(\varphi_2(x_1)/u) = \lambda t.e$. En cualquier caso, cualquiera que sea la interpretación de $\varphi_2(x_1)/u$, esto es, $\lambda t.e$, se cumple que $\lambda t.\# \approx \lambda t.e$ y por tanto que $\hat{\mu}_1(\varphi_1(x_1)/u) \approx \hat{\mu}_2(\varphi_2(x_1)/u)$. Dado que la relación de compatibilidad es claramente monótona, por ser (según la definición 2.16) todas las funciones soporte de la signatura estrictas respecto a #, es posible concluir que $\hat{\mu}_1 \approx \hat{\mu}_2$ y por consiguiente que $\mu_1 \approx \mu_2$.

□

3.2.3 Sobre la completitud del sistema de síntesis formal.

Lo ideal sería que todo sistema de inferencia fuera, además de correcto, completo. La completitud en un sistema de síntesis formal es una característica que, cuando se cumple, permite que a partir de cualquier descripción pueda alcanzarse cualquier otra que sea conductualmente equivalente a la primera, es decir, que no existan dos descripciones conductualmente equivalentes que no estén unidas por un camino de derivación.

Sin embargo, demostrar esto puede ser demasiado difícil en sistemas con un gran número de reglas complejas. El sistema presentado las tiene, por ello, hasta que consiga demostrar si es o no completo, adoptaré un enfoque más pragmático y buscaré 'sospechas razonables' de que, para un cierto propósito, el sistema de síntesis es capaz de alcanzar todas las descripciones 'interesantes'. Para poner de manifiesto que el sistema de transformación presentado es lo suficientemente completo (y además adecuado) para el ámbito de la síntesis conductual, se dedicarán tres secciones (la §3.3, la §4.5 y la §6.4) y un capítulo de la presente memoria (el 5).

No obstante, compárese por un momento el enfoque presentado con otros: en un sistema real de síntesis exigir la completitud parece por el momento demasiado ambicioso pero, además, en las herramientas de síntesis convencional exigir la corrección es inconcebible. Así que el sistema aquí presentado aún no garantizando la completitud, supera notablemente las posibilidades del estado actual de la síntesis conductual.

3.2.4 Estudio de la complejidad temporal de las reglas de transformación.

Dada la orientación eminentemente práctica de la investigación realizada, es necesario, para finalizar, reseñar algunas consecuencias que pueden derivarse de la puesta en práctica del sistema de síntesis formal propuesto. Por ello se dedicará esta sección a estudiar la complejidad temporal de las reglas propuestas en §3.1.2 y en §3.1.3.

Sea una especificación ecuacional (Σ, X, I, O, φ). Calculemos los tiempos de aplicación y la complejidad temporal de la aplicación de cada una de las reglas, en función de los siguientes parámetros:

- $|\Sigma|$ *número de símbolos de operación de la signatura*
- $|X|$ *número de señales de la especificación ecuacional*
- $|\varphi|$ *número de definiciones de la especificación ecuacional*
- $|\varphi_{sim}|$ *número de símbolos utilizados en todas las definiciones de la especificación ecuacional*
- $|\varphi_{med}|$ *número de símbolos medio por definición*

Por un lado, $|\Sigma|$ y $|\varphi_{med}|$ al no variar demasiado de un diseño a otro, pueden considerarse constantes. Por otro, $|X|$, $|\varphi|$ y $|\varphi_{sim}|$ crecen linealmente con el tamaño del problema, cumpliendo siempre la inecuación $|\varphi|<|X|<|\varphi_{sim}|$, pudiendo llegar a ser considerados iguales si la descripción está aplanada, es decir, si cada definición posee un único símbolo de operación.

Comenzaré analizando cada una de las acciones atómicas que componen las reglas de inferencia.

Si se analiza la especificación de las reglas, todas ella toman como argumento, al menos, un nombre de señal. Este nombre de señal indica sobre qué definición se debe aplicar la regla. Conceptualmente las definiciones (pares señal-término) formarán una lista, y será la

implementación concreta de esa lista la que determine cual será la complejidad en el peor caso de la búsqueda: si dicha lista se implementa enlazada $O(|X|)$, si se implementa como árbol binario de búsqueda $O(log_2|X|)$ si se implementa como tabla hash podría llegar a $O(1)$ y si la señal indica una referencia en lugar de un nombre se obtendría $O(1)$.

Además, muchas de ellas también toman como argumento una posición dentro del término que forma la definición, lo que hará que el tiempo de búsqueda del subtérmino concreto (asumiendo una implementación en forma de árbol que refleje la estructura del término) sea en el peor caso (un término degenerado) un factor de $|\varphi_{med}|$. Por consiguiente la complejidad de la búsqueda de subtérminos será $O(1)$.

Todo reemplazamiento conlleva la copia de un término sobre una posición de otro término. Suponiendo que el tamaño del término a reemplazar sea del orden del tamaño de cualquier definición y que el propio reemplazamiento tenga que buscar el subtérmino a reemplazar, el tiempo de realización de esta transformación en el peor de los casos puede depender de $|\varphi_{med}|^2$, en cuyo caso la complejidad de un reemplazamiento será $O(1)$.

Una comprobación de que un nuevo nombre no colisiona implica un recorrido entre todos los símbolos de la signatura y entre todas las señales de la especificación ecuacional, luego su ejecución será lineal con $|\Sigma|+|X|$, y su complejidad $O(|X|)$.

Un renombrado de una señal también depende de la estructura de datos que sustente a la especificación ecuacional. Si el nombre de la señal aparece explícitamente en la estructura de los términos, el renombrado de una señal implicará recorrer todos los símbolos del cuerpo ecuacional, teniendo por complejidad $O(|\varphi_{sim}|)$. Sin embargo, si el nombre aparece como una referencia a un elemento de una lista de señales, el renombrado tendrá la misma complejidad que la búsqueda de una señal. Asumiré que se adopta esta última opción en cualquier implementación 'razonable'.

Con las anteriores medidas, repasemos las complejidades de las reglas de transformación. Así:

- **Regla de substitución**: realiza dos búsquedas de señal y un reemplazamiento. De las dos búsqueda la segunda no incrementa la complejidad si se asume que los términos no contienen nombres de señales sino referencias y que señales y definiciones comparten una estructura común. Así la complejidad total puede variar, dependiendo de la implementación de la lista de señales, entre $O(|X|)$ y $O(1)$.
- **Regla de renombrado**: realiza un chequeo de no colisión y un renombrado. Luego tendrá una complejidad $O(|X|)$.
- **Regla de expansión**: realiza un chequeo de no colisión, una búsqueda de señal y un reemplazamiento. Dado que la búsqueda de la señal se puede simultanear con el chequeo de no colisión, la complejidad queda como: $O(|X|)$.
- **Regla de eliminación**: recorre todos los símbolos del cuerpo para chequear si aparece el símbolo a eliminar, por tanto tiene una complejidad $O(|\varphi_{sim}|) \approx O(|X|)$.
- **Reglas de limpieza**: realizan dos búsquedas de señal, un chequeo de igualdad de términos y un renombrado, luego la complejidad variará entre $O(|X|)$ y $O(1)$.
- **Reglas de aplicación**: realizan una búsqueda de señal, un reemplazamiento y un ajuste, sabiendo que el ajuste conlleva el recorrido de un término, obtenemos que la complejidad variará nuevamente entre $O(|X|)$ y $O(1)$.
- **Regla de reemplazo**: involucra una búsqueda de señal y un reemplazo, luego la complejidad variará entre: $O(|X|)$ y $O(1)$

Como puede observarse la complejidad temporal de todas las transformaciones es lineal. De ellas, las de substitución, limpieza, aplicación y reemplazo pueden hacerse de complejidad logarítmica o casi constante utilizando una estructura de datos adecuada para implementar la lista de

pares señal-definición. Las de expansión y renombrado también pueden hacerse de complejidad constante si hacemos que el nuevo símbolo esté prechequeado por algún mecanismo de generación de identificadores únicos, en cuyo caso la única regla que continuaría conservando complejidad lineal sería la de eliminación.

3.3 Ejemplos de la implantación de técnicas de diseño sobre el sistema de síntesis formal.

En la literatura existen una colección de técnicas [Leis83] para transformar grafos que representan computaciones de manera que puedan obtenerse otros equivalentes cuyas implementaciones tengan mejores rendimientos. En esta sección aplicaremos esas técnicas sobre especificaciones ecuacionales ya que, tal y como se mostró en §2.4.5, el formalismo de especificación ecuacional tiene una equivalencia directa con los grafos de flujo. El objetivo que se persigue es comprobar la versatilidad tanto del mecanismo de especificación como del sistema de transformación para adaptarse a un gran número de técnicas distintas de diseño conductual pero con una notable ventaja respecto a sistemas convencionales: si dichas técnicas se implantan en el sistema formal propuesto, puede asegurarse que se realizan diseños libres de error.

Habitualmente, para comprobar la corrección de la aplicación particular de una técnica, o bien se recurre a una simulación más o menos exhaustiva, o se recurre a la semántica informal de los grafos para verificar que las descripciones de origen y destino denotan el mismo cálculo o conservan cierta relación. Sin embargo, si expresamos el cálculo mediante una especificación ecuacional se tiene, aparte de dichas alternativas, otra: la de encontrar un camino de derivación que una (o transforme) la especificación original y la final, representando dicho camino, el flujo de diseño verificado.

En esta sección, tomando como ejemplo un filtro IIR de segundo orden, se mostrará cómo dicha derivación siempre existe y cómo incluso sigue un esquema fácilmente automatizable. Así cada una de las técnicas presentadas podrá concebirse como una secuencia de transformaciones pertenecientes al conjunto básico.

Previo a la presentación de las técnicas, es necesario definir algunos conceptos básicos que sirven para comprender la utilidad de las mismas. Se han definido para grafos de flujo, sin embargo, no existe problema en extenderlas para especificaciones ecuacionales, dada su ya conocida equivalencia.

- **Camino**: es una sucesión de nodos adyacentes en la que todos, excepto el último, deben ser distintos.
- **Ciclo**: es un camino en el que el primer nodo y el último son el mismo.
- **Latencia computacional de un camino**: suma de los tiempos de ejecución de todos los nodos operativos que componen un camino o, si se mide en número de nodos, número de ellos que lo forman.
- **Latencia computacional de un grafo**: es el máximo del conjunto de latencias computacionales de los caminos sin retardos que componen el grafo, lo que viene a ser equivalente a la latencia computacional del camino crítico.
- **Período de iteración de un ciclo**: se define como la latencia computacional del ciclo, dividido por el número de retardos.
- **Período mínimo de iteración de un grafo**: es el máximo de los períodos de iteración de los ciclos que componen el grafo. Este período límite de iteración determina el período de iniciación mínimo con el que cualquier circuito que implemente el grafo puede operar.

3.3.1 Retemporización (retiming).

Se entiende por retemporización, el cambio de posición y número de los retardos que componen un grafo, tal que el grafo resultante represente el mismo algoritmo pero tenga una implementación monociclo con un rendimiento distinto. A nivel lógico, la retemporización tiene un significado equivalente: consiste en reagrupar biestables de un circuito para que, sin modificar su comportamiento como caja negra, mejore su rendimiento. Un ejemplo de retemporización se ilustra en la fig. 3.12.

Los objetivos habituales que justifican el retiming para una implementación monociclo ('período de muestreo' = 'ciclo de reloj') son: reducir la latencia computacional del camino crítico, gracias a lo cual puede reducirse el período de muestreo y por consiguiente el ciclo de reloj; minimizar el número de retardos, reduciendo así el número final de registros hardware; y finalmente conseguir una agrupación de los operadores que, al ser implementados

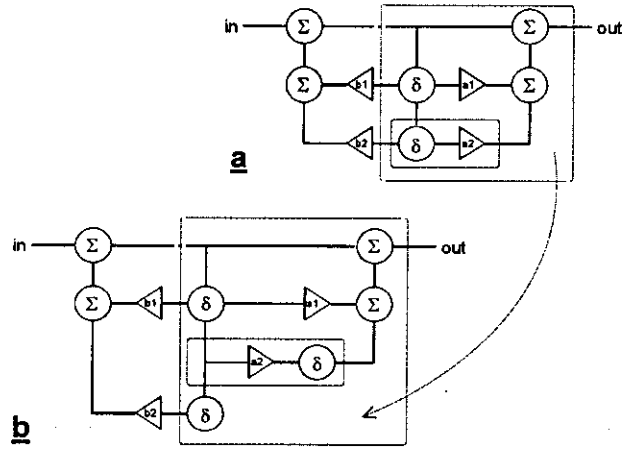


Fig. 3.12: Ejemplo de retemporización.

mediante lógica combinacional, permite abrir el espacio de soluciones de la posterior fase de síntesis lógica.

Algunos autores [PoRa94][ChLS93] también han aplicado técnicas de retemporización en el campo de síntesis de alto nivel (generando implementaciones multiciclo en las que 'período de muestreo' > 'ciclo de reloj'). En estos casos los objetivos eran otros: por un lado, reducir la longitud (medida en nodos o en carga computacional) del camino crítico en grafos de operaciones; y por otro, modificar estructuralmente las movilidades de las operaciones para ampliar el espacio de soluciones de los algoritmos de planificación.

En cualquier caso, la técnica de retemporización se basa en la siguiente idea: si todas las entradas de un nodo operacional están retrasadas *n* ciclos, esos *n* retardos pueden ser eliminados de las entradas si se colocan en cada una de sus salidas.

En el formalismo propuesto, esta idea puede formularse como la calidad de distributivo que tiene el operador *fby* respecto a cualquier operador no temporal. Esta propiedad, junto a otras muchas, se demostrará en el capítulo 4 y puede formalizarse, para cualquier símbolo de operación $\sigma \in \Sigma$, como la $Lu(\Sigma)$-ecuación siguiente:

$$\sigma(\ (\ z_1\ fby\ x_1\ ),\ \overset{n}{\ldots},\ (\ z_n\ fby\ x_n\ )\ ) = \sigma(\ z_1,\ \overset{n}{\ldots},\ z_n\ )\ fby\ \sigma(\ x_1,\ \overset{n}{\ldots},\ x_n\ )$$

EJEMPLO 3.6

Sea el cuerpo de la especificación ecuacional de un filtro recursivo de segundo orden (la especificación completa puede verse en el ejemplo 2.13 y equivale al estructural mostrado en la fig. 3.12-a).

```
body
out = z - ( a1*(0 fby z) + a2*(0 fby 0 fby z) )
z = ( b1*(0 fby z) + b2*(0 fby 0 fby z) ) + in
```

Intentemos reproducir, mediante derivación, la retemporización mostrada en la fig. 3.12. Para ello, es necesario formalizar mediante *Lu*(*Ent*)-ecuaciones algunos conocimientos básicos sobre el comportamiento de los operadores (suponiendo que el álgebra soporte sea el conjunto de los enteros, Z:

$$(x_1 \; fby \; y_1) * (x_2 \; fby \; y_2) = (x_1 * x_2) \; fby \; (y_1 * y_2) \qquad (eq1)$$

$$a2 \; fby \; a2 = a2 \qquad (eq2)$$

$$x * 0 = 0 \qquad (eq3)$$

La ecuación *eq*1 establece la distributividad del operador *fby* respecto de la multiplicación de enteros. La ecuación cerrada *eq*2 describe que la concatenación de una secuencia infinita de valores *a*2 al primer valor de otra secuencia infinita de valores *a*2, es igual a la secuencia infinita de valores *a*2 (recuérdese que *a*2 no es una señal ni una variable sino una constante declarada en signatura *Ent* y que, por consiguiente, denota algún número entero). La ecuación *eq*3 establece una propiedad bien conocida de los números enteros.

Si queremos retemporizar uno de los multiplicadores, es necesario que todas sus entradas estén retardadas. Por ello, debemos aplicar primeramente la ecuación *eq*2 utilizando la regla de aplicación de derecha a izquierda sobre la posición 2.2.1 de la definición de *out*. Así, obtenemos la siguiente especificación:

body
out = z - (a1 * (0 fby z) + (a2 fby a2) * (0 fby 0 fby z))
z = (b1 * (0 fby z) + b2 * (0 fby 0 fby z)) + in

A continuación, cambiamos la posición de los retardos aplicando de izquierda a derecha la ecuación *eq*1 sobre la posición 2.2 de la definición de *out*, obteniendo:

body
out = z - (a1 * (0 fby z) + ((a2 * 0) fby (a2 * (0 fby z))))
z = (b1 * (0 fby z) + b2 * (0 fby 0 fby z)) + in

Finalmente, simplificamos los cálculos eliminando la multiplicación que se realiza sobre constantes. Para ello, se aplica de izquierda a derecha la

ecuación *eq*3 nuevamente sobre la posición 2.2.1 del término que define la señal *out*, obteniendo:

body
out = z - (a1 ∗(0 fby z) + (0 fby (a2 ∗(0 fby z))))
z = (b1 ∗(0 fby z) + b2 ∗(0 fby 0 fby z)) + in

Centrémonos por un momento en esta última transformación para destacar un detalle que subraye las ventajas de los sistemas de síntesis formal frente a los clásicos. En esa última derivación, hemos tenido que transformar *a*2 ∗0 en 0, es decir, hemos tenido que tener en cuenta (por la naturaleza del propio sistema formal) los valores iniciales de los registros a retemporizar para obtener el valor inicial del registro retemporizado. Los sistemas clásicos anteriormente referenciados que utilizan técnicas de retemporización en SAN 'olvidan' incluir este aspecto en sus algoritmos no formales debido, supongo, a que generalmente estos valores iniciales suelen ser 0. Pero esta simplificación, que no consta explícitamente entre las precondiciones del algoritmo y que se suele realizar por analogía con el nivel lógico, no es admisible a nivel algorítmico en donde los valores iniciales de los retardos arquitectónicos son cuidadosamente elegidos en especificaciones complejas [CCIT88].

Presentemos abreviadamente la derivación formal realizada y que reproduce la síntesis esquematizada en la fig. 3.13 (esta notación será utilizada a partir de ahora).

body
out = z-(a1 ∗(0 fby z)+a2 ∗(0 fby 0 fby z))
z = (b1 ∗(0 fby z)+b2 ∗(0 fby 0 fby z))+in

AplicacionDI(out, 2.2.1, *eq2*)
AplicacionID(out, 2.2, *eq1*)
AplicacionID(out, 2.2.2, *eq3*)

body
out = z-(a1 ∗(0 fby z)+(0 fby (a2 ∗(0 fby z))))
z = (b1 ∗(0 fby z)+b2 ∗(0 fby 0 fby z))+in

Como se dijo anteriormente, las técnicas de retemporización no son solamente útiles a nivel RT o lógico, sino que pueden serlo también en alto nivel para ampliar el espacio de soluciones alcanzable de los algoritmos de planificación. Además, gracias a la equivalencia de modelos, es posible aplicar retemporización a cualquier especificación ecuacional que contenga retardos, provenga o no de un grafo. Pero sobre todo, puede resultar interesante si se recuerda que el equivalente procedural a una definición con *fby* es un bucle (véase §2.4.5). De este modo es posible reproducir mediante simple retemporización los resultados obtenidos por algunas técnicas de exploración de dependencias en el espacio de iteraciones. Comprobemos esto mediante un ejemplo.

EJEMPLO 3.7

Estúdiese la fig. 3.13. En ella se muestra a la izquierda, el conocido grafo del filtro recursivo de segundo orden en el que se han marcando con línea continua y punteada, respectivamente, el período mínimo de iteración (PMI) y la latencia computacional (LC) de la especificación original. Si utilizamos como unidad para medir estas cantidades el número de nodos (una

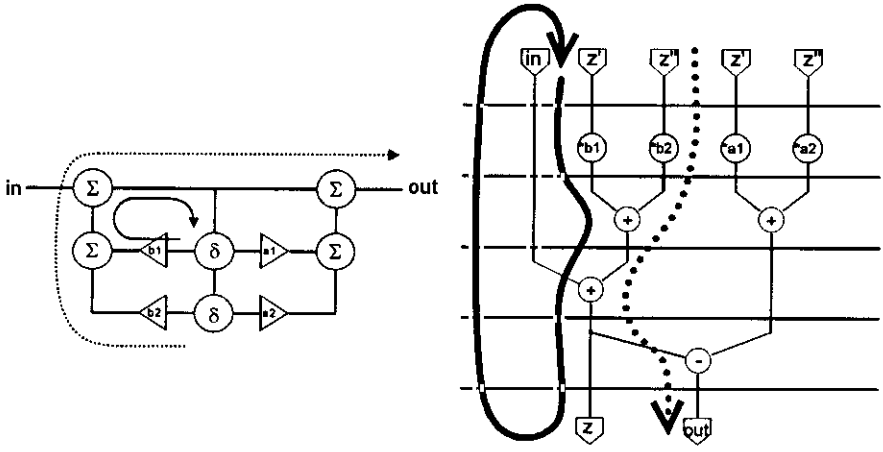


Fig. 3.13: ASAP del filtro original.

simplificación utilizada habitualmente en SAN) comprobamos que valen 3 y 4 respectivamente. La LC viene a medir el número de ciclos mínimo en que podría planificarse dicho grafo y el PMI define el número de ciclos mínimo que deben transcurrir entre dos muestreos de datos en cualquier circuito que implementara dicha computación (independientemente de la técnica usada para planificarlo).

Por su parte, el lado derecho de la fig. 3.13 muestra una planificación ASAP de dicho grafo. En ella se marcan la LC y el PMI. Como era de esperar se tiene un camino crítico igual a la LC, así si se utilizan algoritmos de planificación convencionales no será posible obtener una planificación con menos de 4 ciclos (que necesitará como mínimo 2 multiplicadores y 2 sumadores). Para que la LC alcance el PMI es necesario utilizar técnicas de planificación complejas que realicen exploraciones en el espacio de iteraciones. Muchos algoritmos de este tipo necesitan partir de especificaciones procedurales que incluyen bucles y son incapaces de realizarlas en comportamientos con retardos arquitectónicos. ¿Será completamente necesario partir de una especificación secuencial para aplicar estas técnicas?, la respuesta es no: esto puede realizarse por derivación, aplicando a la especificación ecuacional original sucesivas retemporizaciones formales como la mostrada en el anterior ejemplo.

Sean las siguientes *Lu*(*Ent*)-ecuaciones que añadimos a las propuestas en el ejemplo 3.6:

$$(x_1 \; fby \; y_1) + (x_2 \; fby \; y_2) = (x_1+x_2) \; fby \; (y_1+y_2) \qquad (eq4)$$

$$b1 \; fby \; b1 = b1 \qquad (eq5)$$

$$b2 \; fby \; b2 = b2 \qquad (eq6)$$

A continuación se presenta el resumen de transformaciones (15, de las cuales las últimas 5 son estéticas) a realizar para obtener formalmente una especificación ecuacional planificable en un mínimo de 3 ciclos y que, en caso de hacerlo en 4, sólo requeriría 1 multiplicador y 1 sumador. El resultado obtenido se muestra gráficamente en la fig. 3.14 (en el lado izquierdo el grafo

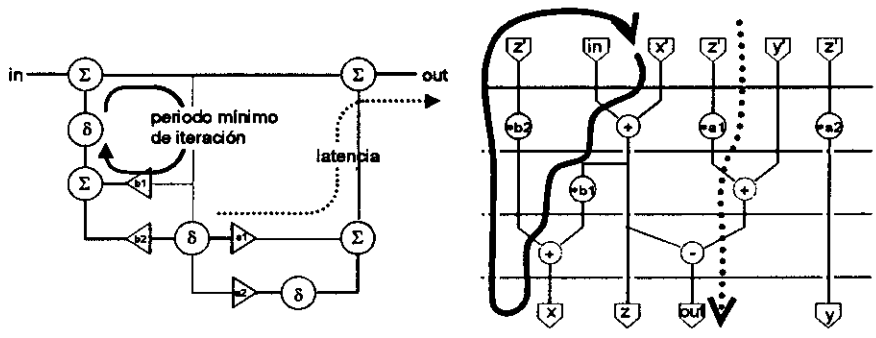


Fig. 3.14: ASAP del filtro retemporizado.

retemporizado equivalente y en el derecho la planificación ASAP del algoritmo).

body
out = z-(a1 ∗(0 fby z)+a2 ∗(0 fby 0 fby z))
z = (b1 ∗(0 fby z)+(b2 ∗(0 fby 0 fby z))+in

AplicacionDI(out, 2.2.1, *eq2*)
AplicacionID(out, 2.2, *eq1*)
AplicacionID(out, 2.2.2, *eq3*)
AplicacionDI(z, 1.1.1, *eq5*)
AplicacionID(z, 1.1, *eq1*)
AplicacionID(z, 1.1.1, *eq3*)
AplicacionDI(z, 1.2.1, *eq6*)
AplicacionID(z, 1.2, *eq1*)
AplicacionID(z, 1.2.1, *eq3*)
AplicacionID(z, 1, *eq4*)
Expansion(out, 2.1.2, A)
Expansion(out, 2.2.2.2, t1)
Expansion(z, 1.2.2.2, t2)
Limpieza(t1, t2)
Limpieza(t2, A)

body
out = z-(a1 ∗A+(0 fby (a2 ∗A)))
z = (0 fby (b1 ∗z+b2 ∗A))+in
A = 0 fby z

Obsérvese cómo aplicando éstas sencillas reglas, junto con el uso de un conjunto de $Lu(\Sigma)$-ecuaciones que resuman el conocimiento necesario sobre esta técnica, es posible realizar una retemporización formalmente correcta (o verificar que cualquiera otra obtenida por otros medios es correcta).

En este punto, es interesante recordar que existe un resultado en teoría de procesado digital de señal que establece que el número de retardos en un

ciclo, no puede cambiarse vía retemporización. Por ello, dado que el PMI es 3 una vez alcanzada una LC igual al PMI es inútil seguir buscando mediante esta técnica posibles soluciones que reduzcan el número de ciclos de la planificación.

3.3.2 Segmentación (pipelining).

La segmentación consiste en la inserción de nodos de retardo en caminos de un grafo que no estén incluidos en ningún ciclo del mismo. El efecto es que varias iteraciones del algoritmo pueden ser computadas simultáneamente, a condición de que el comportamiento del circuito resultante esté retrasado respecto al comportamiento original.

El principal objetivo que justifica la segmentación en implementaciones monociclo es reducir la latencia computacional del camino crítico y por tanto el período de muestreo y el ciclo de reloj. Para ello existen dos tipos de segmentación que dependen del período de muestreo requerido: la segmentación a nivel de palabra, cuando los retardos de los nodos son menores que el período requerido y la segmentación a nivel de bit, cuando el retardo de los nodos es mayor que el período requerido. La primera es una técnica de diseño a nivel RT y la segunda a nivel lógico. Aquí se ilustrará la primera.

Además, como se verá en el próximo capítulo, la segmentación tiene una aplicación práctica en alto nivel: lo que en este apartado se mostrará como una planificación de un grafo segmentado, se conoce habitualmente en SAN como planificación con plegado de bucles (*loop-folding*). Este tipo de algoritmos de planificación avanzada, generan implementaciones con las mismas características anteriormente expuestas: permiten ejecutar

simultáneamente varias iteraciones del algoritmo a una frecuencia de entrada de datos mayor (intervalo de iteración) a costa de cierto intervalo de iniciación inicial en el que no se generan valores válidos.

Según el formalismo propuesto, la segmentación se concibe como la calidad de inverso que tiene el operador *next*† (que será formalmente presentado en el capítulo 4, y que lo fue informalmente en §2.4.5) respecto del operador *fby*, que puede formularse como:

$$next\ (\ 0\ fby\ y\ ) = y \qquad (eq7)$$

Así, se puede aplicar dicha propiedad sobre cualquier señal que se desee segmentar y, utilizando la distributividad del operador *next* respecto a cualquier operador no-temporal, desplazarlo hasta los puertos de salida. No obstante, para evitar el desplazamiento de *next* y los intentos de segmentar ciclos, esta ecuación será aplicada siempre sobre las salidas (tantas veces como etapas se deseen) y después se retemporizarán los retardos añadidos hasta colocarlos en el lugar deseado. Un operador *next* como operador raíz en la definición de un puerto de salida es la formalización que adopta el retraso que sufren los grafos segmentados (ya que ateniéndose a la semántica del operador, viene a significar que el primer y sucesivos valores que toma el puerto del circuito segmentado son el segundo y sucesivos que toma la especificación original sin segmentar).

EJEMPLO 3.8

Segmentemos, mediante derivación, el camino crítico de la especificación original del filtro recursivo de 2º orden. El objetivo es partirlo en dos para reducir la LC del grafo a 3 operadores. La razón de por qué no partirla en dos caminos iguales y reducirla a 2 viene por la exigencia de que la segmentación sólo debe aplicarse fuera de ciclos si queremos que conserve el comportamiento, es decir, no podemos segmentar las partes recursivas del

† Si $x \equiv \langle\langle\ x_1, x_2, x_3, x_4\ \ldots\ \rangle\rangle$ entonces $next\ x \equiv \langle\langle\ x_2, x_3, x_4\ \ldots\ \rangle\rangle$.

grafo: dado que parte del camino crítico esta involucrado en un ciclo, la LC del mismo medirá la LC del grafo completo.

Dado que pretendemos hacer dos etapas del camino crítico, aplicamos la ecuación *eq7* de izquierda a derecha sobre la posición ε de la definición de *out*, obteniendo:

body
out = next(0 fby (z - (a1∗(0 fby z) + a2∗(0 fby 0 fby z))))
z = (b1∗(0 fby z) + b2∗(0 fby 0 fby z)) + in

A continuación se retemporiza la resta que aparece en la definición de *out* para ubicar los nuevos retardos a las puertas del ciclo que forma la señal *z*. Para ello utilizo las ecuaciones:

$$(x_1 \ fby \ y_1) - (x_2 \ fby \ y_2) = (x_1 - x_2) \ fby \ (y_1 - y_2) \qquad (eq8)$$

$$0-0 = 0 \qquad (eq9)$$

El resumen de todas las derivaciones se muestra a continuación (13, de las cuales las 10 últimas son estéticas). El resultado que se obtiene se ha representado en la fig. 3.15 (en el lado izquierdo el grafo segmentado equivalente y en el derecho la planificación ASAP del algoritmo). Puede verse como efectivamente el camino crítico se ha reducido 1 ciclo respecto a la especificación original.

body
out = z-(a1∗(0 fby z)+a2∗(0 fby 0 fby z))
z = (b1∗(0 fby z)+(b2∗(0 fby 0 fby z))+in

AplicacionDI(out, ε, *eq7*)
AplicacionDI(out, 1.1, *eq9*)
AplicacionDI(out, 1, *eq8*)
Expansion(out, 1.1, A)
Expansion(out, 1.2.2.1.2, t1)
Limpieza(t1, A)
Expansion(out, 1.2.2.2.2, B)
Expansion(z, 1.1.2, t2)
Limpieza(t2, A)
Expansion(z, 1.2.2, t3)
Limpieza(t3, B)
Expansion(B, 2, t4)
Limpieza(t4, A)

body
out = next(A - (0 fby (a1∗A + a2∗B)))
z = (b1∗A + b2∗B) + in
A = 0 fby z
B = 0 fby A

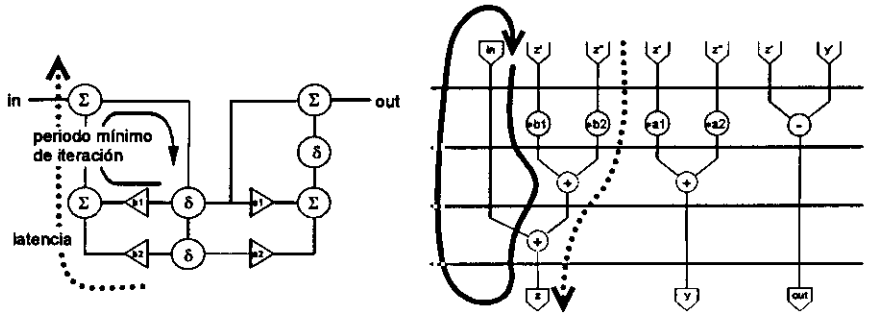


Fig. 3.15: ASAP del filtro segmentado.

Dado que en un grafo el PMI es la razón entre la LC y el número de retardos del lazo crítico, si se desea mejorar el rendimiento del circuito (disminuir PMI), se necesita o bien aumentar el numero de retardos, o bien disminuir la LC. Eso es lo que se proponen las dos siguientes técnicas que van a ser formalizadas.

3.3.3 Reducción de bucles (loop-shrinking).

Esta técnica [MaMT92] consigue, sin modificar la funcionalidad del circuito, disminuir el período mínimo de iteración a través la disminución de la latencia computacional del bucle crítico, y utiliza simplemente las propiedades aritméticas básicas de los nodos operativos, estas son: conmutatividad, asociatividad y distributividad. Mediante la aplicación de estas propiedades sobre operadores del bucle crítico se consigue sacarlos del lazo de cálculo disminuyendo, por tanto, su LC. Véase en la fig. 3.16, cómo usando la asociatividad de las sumas que involucran la entrada se reduce a 2 el PMI.

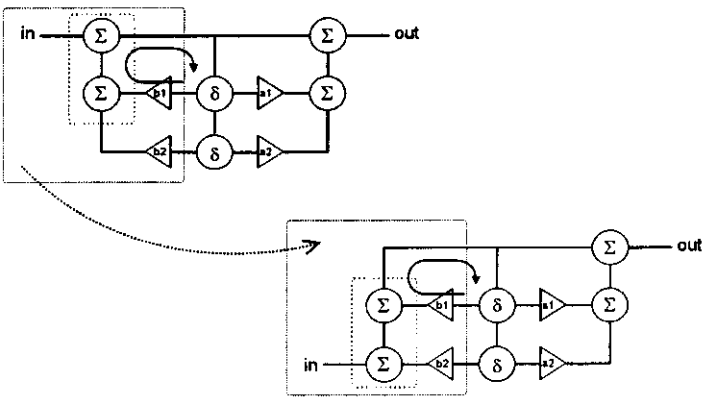


Fig. 3.16: Ejemplo de reducción de bucles.

Como en anteriores técnicas, lo que se intenta es reducir aún más el período de muestreo del sistema, lo que desde una perspectiva de alto nivel, se traduce en una disminución de la longitud del camino crítico más allá del alcance de las dos técnicas anteriores.

Como puede intuirse, formalizar la técnica de reducción de bucles es notablemente sencillo, ya que para cualesquiera símbolos de operación σ_1 y σ_2 las propiedades aritméticas pueden formalizarse (sin pérdida de generalidad, para operadores binarios) según los esquemas siguientes:

$$\sigma_1(x, y) = \sigma_1(y, x) \qquad \textit{conmutatividad}$$

$$\sigma_1(x, \sigma_1(y, z)) = \sigma_1(\sigma_1(x, y), z) \qquad \textit{asociatividad}$$

$$\sigma_1(\sigma_2(x, y), \sigma_2(z, w)) = \sigma_2(\sigma_1(x, z), \sigma_1(y, w)) \qquad \textit{distributuvidad}$$

EJEMPLO 3.9

Intentemos aplicar esta técnica como un proceso de derivación de especificaciones ecuacionales. Con ella conseguiremos un grafo que pueda ser planificable en sólo dos ciclos. Para ello primero formalizamos la asociatividad de la suma de enteros mediante una *Lu*(*Ent*)-ecuación:

$$x + (y + z) = (x + y) + z \qquad (eq10)$$

Seguidamente aplicamos esta ecuación de derecha a izquierda sobre la raíz de la definición de *z*, resultando:

body
out = z - (a1 ∗(0 fby z) + a2 ∗(0 fby 0 fby z))
z = b1 ∗(0 fby z) + (b2 ∗(0 fby 0 fby z) + in)

Esta especificación ecuacional corresponde con el grafo transformado en la fig. 3.16. No obstante, aunque hemos reducido a 2 el PMI, la LC continúa siendo 4 así que retemporizamos convenientemente los retardos hasta alcanzar una LC de 2. El resumen de la derivación a partir de la especificación original se presenta a continuación. La figura 3.17, muestra el estructural correspondiente a la especificación final y una planificación ASAP de la misma.

	body out = z-(a1 ∗(0 fby z)+a2 ∗(0 fby 0 fby z)) z = (b1 ∗(0 fby z)+(b2 ∗(0 fby 0 fby z))+in
AplicacionDI(z, ε, *eq10*)	
AplicaciónID(z, 2.2, x = x fby next x)	
AplicacionDI(z, 2.1.1, *eq6*)	
AplicacionDI(z, 2.1, *eq1*)	
AplicacionID(z, 2.1.1, *eq3*)	
AplicacionID(z, 2, *eq4*)	
AplicacionID(z, 2.1, 0+x = x)	
AplicacionDI(z, 1.1, *eq5*)	
AplicacionDI(z, 1, *eq1*)	
AplicacionID(z, 1.1, *eq3*)	
AplicacionDI(out, 2.2.1, *eq2*)	
AplicacionID(out, 2.2, *eq1*)	
AplicacionID(out, 2.2.2, *eq3*)	
AplicacionDI(out, 2.1.1, a1 fby a1=a1)	
AplicacionID(out, 2.1, *eq1*)	
AplicacionID(out, 2.2.2, *eq3*)	**body** out = z-(((0 fby (a1 ∗z))+(0 fby (a2 ∗A))) z = (0 fby (b1 ∗z))+(in fby (b2 ∗A+next in)) A = 0 fby z
Expansion(out, 2.2.2.2, A)	
Expansion(z, 2.2.1.2, t1)	
Limpieza(t1, A)	

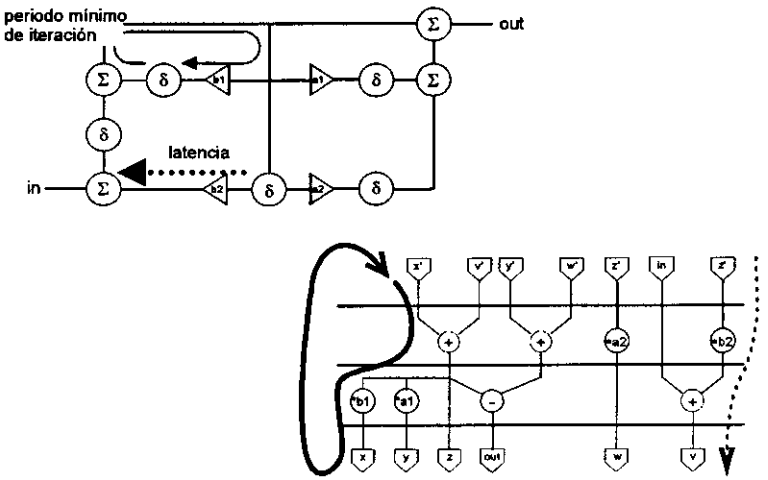


Fig. 3.17: ASAP del filtro reducido.

3.3.4 Anticipación (lookahead).

La técnica de anticipación [Mees88][PaMe89] consigue, sin modificar la funcionalidad del circuito, disminuir el período mínimo de iteración mediante el aumento del número de retardos en el ciclo crítico. Sus objetivos son los mismos que los de la técnica de reducción de bucles, por lo que esta técnica puede ser útil en alto nivel para reducir la longitud del camino crítico a costa de un aumento notable del número de operaciones.

Una anticipación de grado *m* de un ciclo, permite insertar m retardos dentro del mismo y, replicando operadores, calcular fuera de él (anticipando) los *m* primeras iteraciones que eran realizadas por el ciclo primitivo. La fig. 3.16 mostraba el ciclo crítico del filtro de 2º orden, la fig. 3.18 muestra el resultado de anticipar dicho ciclo en grado 1. Como puede observarse la complejidad de esta técnica (y el posible chequeo de su corrección) es superior a la de las

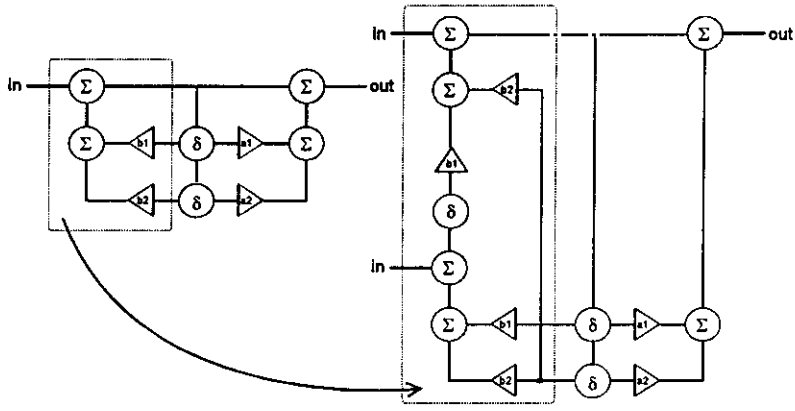


Fig. 3.18: Ejemplo de anticipación.

presentadas anteriormente, lo que justifican aún más el enfoque transformacional.

En el formalismo ecuacional, una anticipación de grado *m* se realiza mediante la substitución (repetida *m* veces), de la señal que define el ciclo a anticipar, por su propia definición. Esta substitución replica los operadores que aparecen en un lazo de modo que, extrayendo fuera de él los operadores combinacionales, se consigue incrementar la relación entre retardadores y operadores que postula esta transformación.

EJEMPLO 3.10

Repliquemos mediante derivación la anticipación mostrada en la figura 3.18. Primero, detectamos el lazo crítico (definido por la primera realimentación de *z*) y eliminamos el resto de subexpresiones comunes que aparecen en la especificación ecuacional original.

	body out = z-(a1∗(0 fby z)+a2∗(0 fby 0 fby z)) z = (b1∗(0 fby z)+(b2∗(0 fby 0 fby z))+in
Expansion(out, 2.1.2, A) Expansion(out, 2.2.2, B) Expansion(z, 1.2.2, t1) Limpieza(t1, B) Expansion(B, 2, t2) Limpieza(t2, A)	**body** out = z-(a1∗A+a2∗B) z = (b1∗(0 fby z)+(b2∗B))+in A = 0 fby z B = 0 fby A

A continuación realizamos la anticipación de grado 1, substituyendo la aparición de *z* en la definición de *z* por ella misma:

body
out = z-(a1∗A+a2∗B)
z = (b1∗(0 fby ((b1∗(0 fby z)+(b2∗B))+in))+(b2∗B))+in
A = 0 fby z
B = 0 fby A

Seguidamente se retemporiza el nuevo retardo internándolo en el bucle inducido por la recursividad de *z*, resultando la especificación siguiente:

body
out = z-(a1∗A+a2∗B)
z = (((b1∗b1∗B+b1∗b2∗C)+ b1∗(0 fby in)) + b2∗B)+in
A = 0 fby z
B = 0 fby A
C = 0 fby B

Para finalizar se reordenan las multiplicaciones y las sumas mediante distributividad y conmutatividad, obteniendo:

body
out = z-(a1∗A+a2∗B)
z = ((b1∗b1∗B+b1∗b2∗C) + b2∗B) + (b1∗(0 fby in) + in)
A = 0 fby z
B = 0 fby A
C = 0 fby B

Una vez obtenido un PMI de 2 ciclos reducimos, mediante una segmentación en dos etapas, la LC del camino crítico a 2 ciclos:

body
out = next(A - 0 fby (a1∗A + 0 fby (a2∗A))))

$z = (0\ fby\ (b1 * b1 * A + b1 * b2 * B) + 0\ fby\ (b2 * A)) + (0\ fby\ (b1 * in) + in)$
$A = 0\ fby\ z$
$B = 0\ fby\ A$

La representación esquemática de las derivaciones realizadas, junto con la implicaciones que en SAN pudieran tener (una planificación ASAP en 2 ciclos) se muestra en la fig. 3.19.

3.3.5 Reflexiones sobre el alcance práctico del sistema de síntesis formal.

Aparte de comprobar la utilidad y versatilidad del sistema formal propuesto debe extraerse, como consecuencia de esta sección, la filosofía que debe seguirse para hacer un uso eficiente del mismo.

Para reproducir mediante derivación cualquier técnica de diseño conductual se requieren dos condiciones: primero, encontrar un conjunto de ecuaciones que sean capaces de formalizar todo el conocimiento que sustenta a dicha técnica y segundo, encontrar una serie regular de aplicaciones de reglas estructurales que permitan utilizar convenientemente (mediante reglas conductuales) dichas ecuaciones. Si además se desea que dicha derivación sea automática deberá diseñarse un algoritmo que, para toda especificación, sea capaz de disparar convenientemente las reglas que forman esa serie regular anteriormente referida. Nótese que, en cualquier caso, si el algoritmo de diseño (el que dispara reglas) falla, el circuito resultante podrá no ser el deseado pero, al haber sido obtenido por derivación, al menos siempre será correcto.

El objetivo fundamental de la investigación realizada es, aparte de construir un sistema formal, sintonizarlo para reproducir formalmente las técnicas de

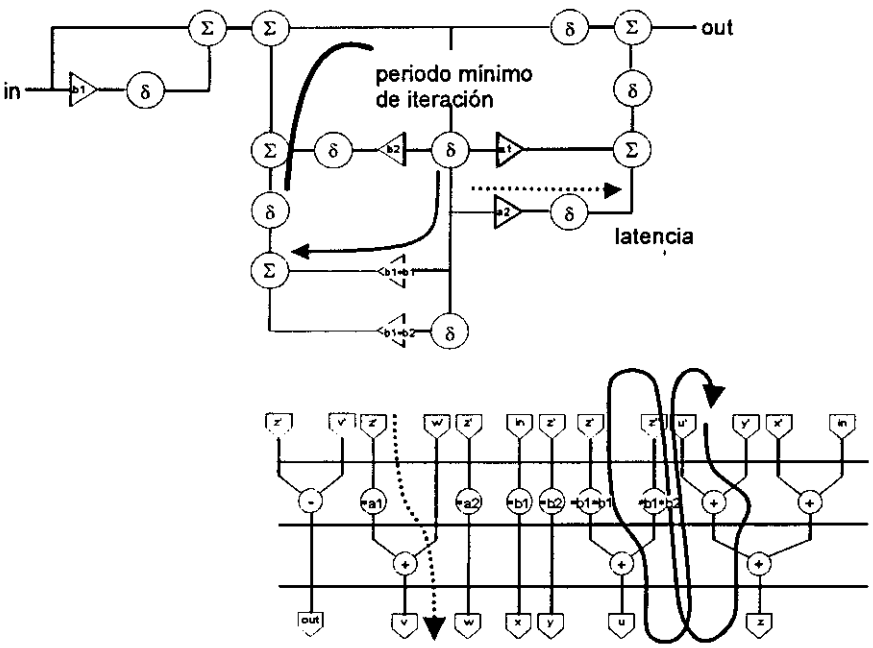


Fig. 3.19: ASAP del filtro anticipado.

síntesis de alto nivel. Por ello, según la norma anteriormente expuesta, ahora es necesario encontrar el conjunto de ecuaciones que formalicen los conocimientos que subyacen en un proceso de SAN y encontrar un algoritmo que los aplique utilizando como núcleo de transformación el sistema de inferencia presentado en este capítulo. Y eso será la tarea de los dos próximos capítulos.

Capítulo 4

Síntesis formal de alto nivel por derivación

The real utility of abstract general knowledge is that just a few general principles are sufficient for answering innumerable questions.
Robert J. Baum

Como quedó patente en la última sección del capítulo anterior, el sistema de inferencia propuesto puede ser utilizado para realizar cualquier técnica de síntesis por derivación formal, siempre y cuando se puedan expresar ecuacionalmente los conocimientos que sustentan dicha técnica. El objetivo fundamental de esta investigación es realizar síntesis de alto nivel (SAN), por lo tanto, será necesario encontrar un conjunto de propiedades generales que, formalizadas como ecuaciones, puedan ser aplicadas a una especificación ecuacional y la transformen en otra en la que sus componentes se reutilicen.

Para formalizar conocimientos se utilizan operadores. Si bien el operador *fby* y los operadores no temporales (heredados de las álgebras soporte de las signaturas) fueron suficientes para describir cualquier circuito síncrono, así como un gran número de computaciones y muchas de sus propiedades, sin

embargo existen conceptos y etapas intermedias en un proceso de SAN que no pueden expresarse convenientemente. Por ello, antes de formalizar los principios que sustentan la SAN, es necesario ampliar el conjunto de operadores para poder hacerlo con comodidad. En esta ampliación se usarán las dos ideas clave que la diferencian respecto de la síntesis monociclo RT: el uso multiplexado (o compartido) de los recursos hardware en el tiempo, y la disparidad entre las frecuencias de adquisición de datos (de muestreo) y de procesado de los mismos (de reloj).

De este modo el capítulo se estructurará en cuatro bloques. Un primero (§4.1) que tendrá como objetivo la ampliación del conjunto de operadores temporales. Un segundo (§4.2) que presentará los principios básicos de la SAN. Un tercero (§4.3 y §4.4) que estudiará cómo formalizarlos mediante $Lu(\Sigma)$-ecuaciones. Y un cuarto (§4.5) que mostrará cómo el sistema de inferencia puede reproducir, utilizando las anteriores ecuaciones, los resultados de cualquier algoritmo de SAN.

4.1 Un conjunto de operadores temporales para la formalización de conductas intermedias en un proceso de síntesis de alto nivel.

Según el modelo establecido hasta el momento, todas las señales de una especificación ecuacional (o del correspondiente circuito en una implementación monociclo) trabajan a la misma frecuencia. Esto hace que todo valor en una misma posición de cualquier secuencia acontezca en el mismo instante temporal. Sin embargo, si se desea ampliar el modelo para permitir la descripción de señales a distintas frecuencias†, sucederá que los

† Tal como sucede en todo circuito en el que la frecuencia de adquisición de datos sea menor que la frecuencia de las transferencias entre registros.

valores transportados dejan de ser comparables en el tiempo y, por tanto, cualquier operador que los procese no 'sabrá' como emparejar los datos para calcular los resultados. Estudiemos este fenómeno en mayor profundidad, pero desde un punto de vista hardware.

Supónganse dos sistemas síncronos, A y B, funcionando a distintas frecuencias, f_A y f_B, que se comunican sin ningún tipo de protocolo a través de dos señales comunes: una, $x_{A\to B}$, por la que circula información desde A hacia B y otra, $x_{B\to A}$, por la que circula información en sentido opuesto. Supóngase también que la frecuencia del primer módulo es múltiplo de la frecuencia del segundo, es decir, $f_A = k*f_B$. Y, para finalizar, supóngase que ambos sistemas están continuamente leyendo y escribiendo (cada uno a su frecuencia) sobre las correspondientes señales. Intentemos modelarlas como secuencias de valores.

Si nos centramos en las escrituras, el sistema A en cada pulso de reloj efectúa una sobre $x_{A\to B}$. Por su lado, el sistema B hace lo propio sobre $x_{B\to A}$. Desde el punto de vista de A, el comportamiento de $x_{A\to B}$ está caracterizado por la secuencia « a_1, a_2, a_3, a_4 ... ». Desde el punto de vista de B, el comportamiento de $x_{B\to A}$ lo está por la secuencia « b_1, b_2, b_3, b_4 ... ».

Si, por el contrario, nos centramos en las lecturas, puede observarse que el sistema A lee valores de $x_{B\to A}$ a una frecuencia f_A que es k veces mayor que la frecuencia f_B a la que son colocados. Esto hace que A lea en sucesivas ocasiones un mismo valor que evoluciona a un ritmo más lento del requerido. ¿Cuántas veces leerá el mismo?: k veces, ya que por cada k valores que lee A, B sólo produce 1. De este modo, desde el punto de vista de A, la señal $x_{B\to A}$ estará caracterizada por la secuencia « $b_1, \overset{k}{\ldots}, b_1, b_2, \overset{k}{\ldots}, b_2$, ... ».

Por su lado, el sistema B lee valores de $x_{A\to B}$ a una frecuencia f_B que es k veces menor que la frecuencia f_A a la que son colocados, el efecto es que B es incapaz de leer todos los valores que A genera. ¿Cuántos valores

perderá?: k-1 valores o, de otra manera, B leerá 1 de cada k valores transportados por la señal. Así, desde el punto de vista de B, la señal $x_{A\to B}$ podrá modelarse como la secuencia « a_k, a_{2*k}, a_{3*k}, a_{4*k}, ··· ».

¿Qué está sucediendo?, que una misma señal tiene dos comportamientos distintos que dependen de la relación entre las frecuencias de los sistemas que la utilicen.

Para solucionar esta ambigüedad, algunos sistemas asignan atributos de frecuencia de transmisión a señales y de frecuencia de funcionamiento a operadores. De este modo, toda señal puede ser descrita mediante varias secuencias y es la frecuencia del dispositivo que accede a ella la que condiciona cual hay que elegir. Sin embargo, si bien este enfoque es válido, no es suficientemente conveniente en un sistema de síntesis formal.

Cuando una señal está caracterizada por múltiples secuencias, ¿cuál debe ser considerada como 'verdadera' y cuál es la que esta condicionada por la frecuencia de acceso? o, en el ejemplo, ¿es A o es B el sistema que está caracterizando correctamente la señal $x_{A\to B}$?. En algunos sistemas [IMEC92], tras un análisis global de todas las frecuencias presentes en una especificación, se toma la secuencia cuya frecuencia sea igual a la mayor frecuencia global. Obsérvese que, aunque este problema sea local a un conjunto de módulos que se comunican, estos sistemas requieren de un estudio global para solucionarlo y de un conocimiento expreso de las frecuencias absolutas cuando, para cualquier aplicación de diseño de alto nivel, basta con conocer los factores que relacionan a unas frecuencias con otras.

Por otro lado, desde un punto de vista descriptivo, si se permite que un identificador posea atributos sucederá que distintas apariciones del mismo tendrán distintas interpretaciones (un mismo objeto sintáctico, distintas semánticas). Así toda la transparencia referencial conseguida por el

mecanismo de especificación ecuacional se pierde y parte del sistema de inferencia deja de ser aplicable.

Para finalizar y desde un punto de vista hardware, si las señales físicas no van acompañadas por ningún atributo que determine cómo hay que leerlas (ya que son los dispositivos hardware los que extraen de ellas la información que pueden según su capacidad de procesarlas) no hay razón que obligue a imponerlo en un mecanismo de especificación.

La solución que propongo está basada en un modelado distinto del fenómeno: en lugar de aceptar que una señal puede tener múltiples descripciones, asumiré la existencia de múltiples señales, cada una con su propia y única descripción, y tales que unas son la transformación de las otras. Este enfoque es totalmente análogo al realizado con la indexación temporal y el operador *fby*. En los sistemas que usan indexación, una misma señal puede tener distintas descripciones: la normal *z*, la retrasada un ciclo $z(t\text{-}1)$, dos ciclos $z(t\text{-}2)$, etc (en *Silage* *z*, *z*@1, *z*@2 respectivamente). Sin embargo, según el enfoque ecuacional, la señal primitiva y la retardada son dos señales distintas, tal que la segunda es una traslación temporal de la primera que se realiza usando el operador *fby*.

Así, propongo dos operadores temporales afines, llamados *replicate* y *sample*, que permitirán variar la granularidad con la que puede ser manipulado el tiempo. El primero, cuyo comportamiento se muestra en la fig. 4.1, permitirá modelar el fenómeno observado por el sistema *A* al leer la señal $x_{B\rightarrow A}$: *replicate* será la función que, tomando como argumentos un número natural *k* y una señal « x_1, x_2, x_3, x_4 ... », devuelve otra señal cuyos valores son cada uno de los observados repetidos *k* veces, es decir, « x_1, $\overset{k}{\ldots}$, x_1, x_2, $\overset{k}{\ldots}$, x_2, ... ». El segundo, que se muestra en la fig. 4.2, será utilizado para modelar el fenómeno observado por el sistema *B* al leer la señal $x_{A\rightarrow B}$: *sample* será una función que tomando como argumentos un número natural *k* y una señal « x_1, x_2, x_3, x_4 ... », devuelve otra señal cuyos valores son 1 de

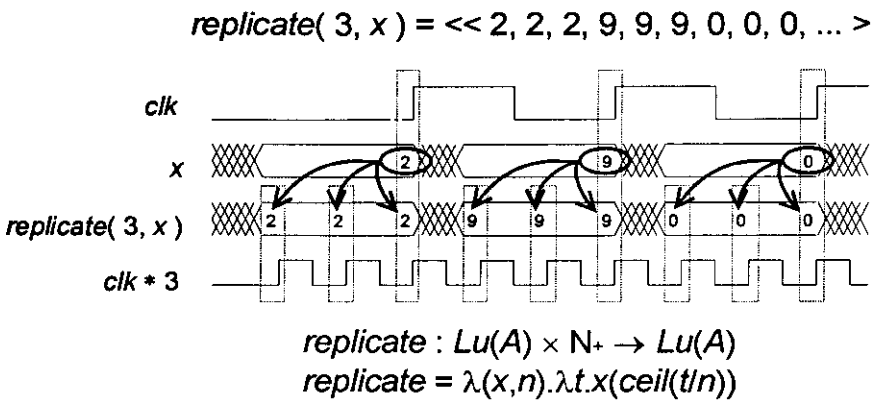


Fig. 4.1: Comportamiento del operador replicate.

cada k valores de los de entrada, es decir, « x_k, x_{2*k}, x_{3*k}, x_{4*k}, ... ».

Gracias a ellos el anterior ejemplo pudiera ser descrito indistintamente (ya que no existen frecuencias privilegiadas) por cualquiera de los cuatro siguientes conjuntos de definiciones (a la derecha de los cuales se ha indicado la relación entre las frecuencias de transmisión de las señales y las frecuencias de funcionamiento de los sistemas A y B):

body

$x_{A\to B}$ = A(replicate(k, $x_{B\to A}$)) — *frecuencia de* $x_{A\to B} = f_A$

$x_{B\to A}$ = B(sample(k, $x_{A\to B}$)) — *frecuencia de* $x_{B\to A} = f_B$

body

$x_{A\to B}$ = sample(k, A($x_{B\to A}$)) — *frecuencia de* $x_{A\to B} = f_B$

$x_{B\to A}$ = replicate(B($x_{A\to B}$), k) — *frecuencia de* $x_{B\to A} = f_A$

body

$x_{A\to B}$ = sample(k, A(replicate(k, $x_{B\to A}$))) — *frecuencia de* $x_{A\to B} = f_B$

$x_{B\to A}$ = B($x_{A\to B}$) — *frecuencia de* $x_{B\to A} = f_B$

body

$x_{A\to B}$ = A($x_{B\to A}$) — *frecuencia de* $x_{A\to B} = f_A$

$x_{B\to A}$ = replicate(k, B(sample(k, $x_{A\to B}$))) — *frecuencia de* $x_{B\to A} = f_A$

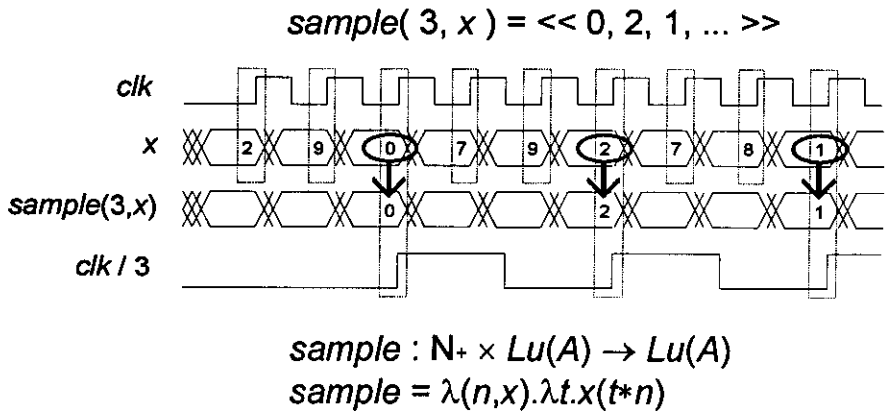


Fig. 4.2: Comportamiento del operador sample.

Obsérvese como es posible, usando las reglas de expansión, substitución, eliminación y renombrado, transformar unos conjuntos de definiciones en otros.

La aplicación de estos operadores en síntesis de alto nivel es inmediata. Si sobre una especificación ecuacional se realiza este tipo de síntesis, el reloj interno del circuito resultante debe funcionar a una frecuencia múltiplo de la frecuencia de muestreo, donde ese múltiplo es el llamado número de ciclos de la planificación o latencia del circuito. Por ello, para indicar la latencia de un circuito sintetizado, la especificación ecuacional que lo describa deberá contener necesariamente a los nuevos operadores: un operador *replicate* por cada uno de los puertos de entrada y un operador *sample* por cada uno de los puertos de salida. De hecho, como veremos más adelante, una de las ideas que sustentará la SAN por derivación estará basada en la cualidad de que un operador es inverso del otro. Así, se colocarán parejas de ellos en lugares estratégicos de la especificación y se irán propagando unos hacia las entradas y otros hacia las salidas. Cuando todos lleguen a su destino, la especificación ecuacional estará describiendo un circuito que funciona a mayor frecuencia.

Una vez formalizado el concepto de diferencia de frecuencias, queda por encontrar un mecanismo para formalizar la idea del uso multiplexado de los recursos en el tiempo. Por un uso multiplexado se entiende que un mismo recurso, en distintos instantes de tiempo, puede estar aceptando datos generados por distintas fuentes. Si dicho recurso es un operador combinacional, se hablará de multiplexado de recursos operativos. Si es un operador *fby*, se estará hablando de multiplexado de elementos de almacenamiento. Si es simplemente una señal, podrá hablarse de multiplexado de caminos de comunicación.

Para expresar esta circunstancia todas las herramientas de síntesis consideran imprescindible el marcado de los operadores. Así, por ejemplo, cuando se realiza una planificación de operaciones, se marca cada operación con un atributo que fija el momento en que se ejecuta. Cuando se realiza asignación, se vuelve a marcar con otro atributo que le asocia un recurso particular. De todos estos atributos las herramientas extraen otros para expresar, por ejemplo, las necesidades de almacenamiento o las necesidades de comunicación y poco a poco la representación interna va creciendo y va haciéndose más difícil de manipular. De hecho, esta proliferación de atributos conlleva a una pérdida de transparencia que obliga a que los sistemas aseguren su robustez mediante complejos chequeos de consistencia de sus representaciones internas [GoCK98].

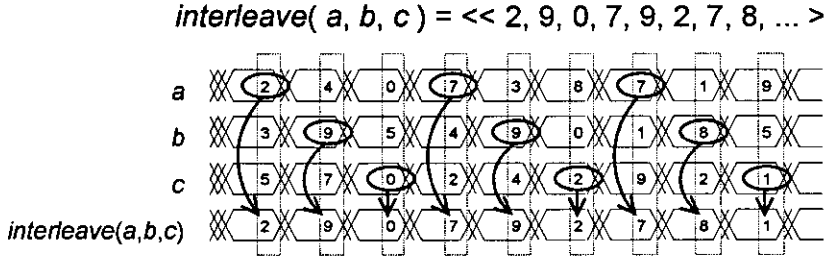
Para evitar este problema volveré a aplicar el paradigma funcional: un objeto sintáctico, una interpretación. En un circuito sintetizado sabemos que una señal puede transportar valores que han sido generados por distintas fuentes, sabemos el número de esas fuentes es finito y sabemos que el orden de acceso a las fuentes, si bien cambia con el tiempo, debe seguir un patrón regular que está gobernado por un controlador hardware que tiene un número de estados finito. Por ello, para modelar el intercalado regular de valores en una señal, utilizaré una familia de operadores temporales que se llamarán

interleave[†]. Un operador genérico *interleave*, véase fig. 4.3, toma como argumento n señales, $x_i \equiv \langle\langle (x_i)_1, (x_i)_2, (x_i)_3 \ldots \rangle\rangle$, y genera una señal que intercala algunos de los valores que transportan las entradas. La secuencia valores de salida será $\langle\langle (x_1)_1, (x_2)_2, \overset{n}{\ldots}, (x_n)_n, (x_1)_{n+1}, (x_2)_{n+2}, \ldots \rangle\rangle$.

Obsérvese que este operador hace que se pierdan n-1 valores de cada n valores que transportan cada una de las señales, por lo que no altera la relación entre las frecuencias de entrada y de salida. La razón es que en un circuito, los módulos están siempre ofreciendo valores independientemente de si son o no utilizados. Así, el operador *interleave* va escogiendo alternativamente sólo uno de los n valores que le ofrecen las n señales x_i.

Nótese también que los parámetros no tienen por qué ser todos distintos. De hecho, si se quieren describir patrones de selección complejos, será necesario repetir el mismo argumento en distintas posiciones.

La aplicación del operador en síntesis de alto nivel no es difícil de encontrar: basta pensar que una especificación, tras ser sintetizada en k ciclos, permite que sus recursos se multiplexen según patrones de longitud



$$interleave : Lu(A) \times \overset{k}{\ldots} \times Lu(A) \rightarrow Lu(A)$$
$$interleave = \lambda(x_1,\ldots,x_k).\lambda t.(x_1(t),\ldots x_k(t))\downarrow(((t\text{-}1)\ mod\ k)+1)$$

Fig. 4.3: Comportamiento del operador interleave.

[†] Utilizaré indistintamente la noción de familia formada por un conjunto de operadores que poseen un número fijo de argumentos y la noción de operador con un número genérico de argumentos.

k. Así, por ejemplo, para expresar que en una planificación de *k* ciclos, una operación se planifica en el ciclo *j*, se utilizará un operador *interleave* de *k* argumentos que tomará a la operación planificada como parámetro de la posición *j*.

Para finalizar, propondré un último operador que ya ha sido utilizado informalmente en anteriores capítulos y que completa el conjunto. Este operador no tiene significado hardware ya que es un operador no causal, sin embargo, juega un papel fundamental en la formalización de etapas intermedias de la SAN y facilita notablemente la especificación de algunas conductas. El nuevo operador es *next* y su función es inversa a la de *fby*, es decir, toma como argumento una señal « x_1, x_2, x_3, ... » y genera otra anticipada en el tiempo que es « x_2, x_3, x_4, ... ». Su comportamiento y definición se muestra en la fig. 4.4.

Por conveniencia podrían existir un mayor número de operadores implícitos del lenguaje que facilitaran su uso, sin embargo, he preferido definir solamente el conjunto mínimo necesario para poder realizar síntesis de alto nivel por derivación. El resto de operadores†, que son extensiones que no se

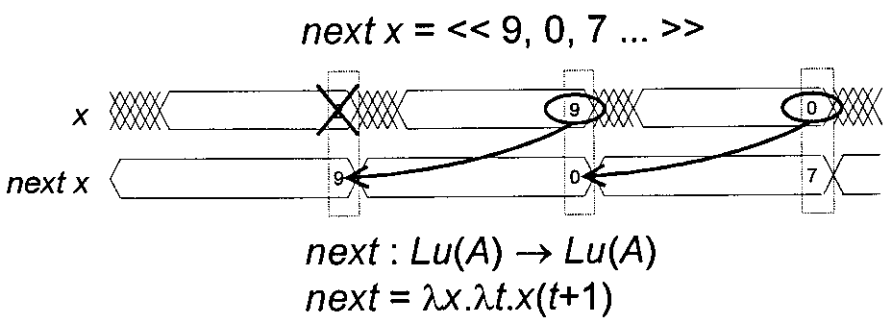


Fig. 4.4: Comportamiento del operador next.

† Operadores de selección condicional, de selección múltiple, de repetición, e incluso un operador que modele el comportamiento de un multiplexor.

utilizan en todo diseño, asumiré que estarán declarados en la signatura de la especificación cuando sean precisos.

4.2 Propiedades de los operadores temporales.

Esta sección está completamente dedicada a establecer y demostrar un conjunto de propiedades que verifican los cinco operadores temporales presentados hasta el momento. Estas propiedades [MeHe98][MeHF98], una vez concretadas como $Lu(\Sigma)$-ecuaciones (§4.4), permitirán realizar síntesis formal de alto nivel utilizando el sistema de inferencia propuesto en el anterior capítulo.

Dado que la mayor parte de las propiedades describen realidades muy genéricas de las álgebras temporales, he preferido no utilizar directamente $Lu(\Sigma)$-ecuaciones para formularlas, en su lugar he usado esquemas con metavariables para poder referir, de un modo compacto, una colección (a veces infinita) de ecuaciones. En su enunciado se presupone la existencia de cierta signatura Σ, de cierta Σ-álgebra soporte A y de su correspondiente Lu-extensión $Lu(A)$. El comportamiento de algunos de los operadores de esta última álgebra se caracterizará en esta sección.

Dado que todos los operadores temporales propuestos son polimórficos y el tipo, tanto de cada uno de sus argumentos, como del objeto que calculan, es el mismo, utilizaré sin pérdida de generalidad las siguientes definiciones, como λ-expresiones sin tipo, de los mismos:

$fby = (\lambda(x,y).(\lambda t.(\mathit{if}\ t{=}1\ \mathit{then}\ x(1)\ \mathit{else}\ y(t\text{-}1))))$

$next = (\lambda x.(\lambda t.(x(t{+}1))))$

$replicate = (\lambda(x,n).(\lambda t.(x(\mathit{ceil}(t/n)))))$

$sample = (\lambda(n,x).(\lambda t.(x(t{*}n))))$

$interleave = (\lambda(x_1, \overset{n}{\ldots}, x_n).(\lambda t.((x_1(t), \overset{n}{\ldots}, x_n(t)) \downarrow((t\text{-}1)\ mod\ n){+}1))))$

donde $\downarrow$ es el operador de restricción de tuplas, *mod* es el resto de la división entera de dos números naturales, *ceil* es la función techo definida como:

$ceil(\ x, y\ ) = if\ x\ mod\ y = 0\ then\ (\ x\ div\ y\ )\ else\ (\ (\ x\ div\ y\ ) + 1\ )$

y *di* es la división entera de naturales.

Estas definiciones me permitirán realizar la mayor parte de las demostraciones como un simple proceso de reducción de λ-expresiones a una forma común utilizando las reglas clásicas de conversión (véase §A.3), por lo que su demostración, e incluso la demostración de un conjunto mayor de ellas, podría realizarse utilizando un demostrador automático†.

4.2.1 Operadores inversos.

Son un conjunto de propiedades que establecen que algunos operadores temporales son inversos de algunos otros. Su utilidad, desde la perspectiva del diseño por derivación, es permitir la aparición de pares de operadores temporales en cualquier posición de una especificación temporal o hacerlos desaparecer, también a pares, de la misma.

IFBY: función del operador *next* como operador inverso de *fby*.

$$\forall x,y \in Lu(A)_s,\ next(\ fby(\ y,\ x\ )\ ) = x$$

DEMOSTRACIÓN. Se realiza reduciendo la expresión izquierda a la derecha. En este caso se realizará con todo detalle para ilustrar expresamente el proceso mecánico de la demostración. Posteriores demostraciones serán más esquemáticas, utilizarán menos paréntesis (sobrentendiendo la asociatividad

† Creando un lazo de unión, que no es nuevo, entre el mundo de la demostración automática y el diseño hardware.

natural de los operadores) y realizarán varias transformaciones en un solo paso.

$next(\ fby(\ y,\ x\ )\ ) =$ *definición de next*

$= (\ \lambda x.(\ \lambda t.(\ x(t+1)\ )\ )\ )(\ fby(\ y,\ x\ )\ ) =$ *conversión* β

$= (\ \lambda t.(\ fby(\ y,\ x\ )(t+1)\ )\ ) =$ *definición de fby*

$= (\ \lambda t.(\ (\ \lambda(x,y).(\ \lambda t.(\ if\ t{=}1\ then\ x(1)\ else\ y(t{-}1)\ )\ )\ )(\ y,\ x\ )(t+1)\ )\ ) =$ *conv.* β

$= (\ \lambda t.(\ (\ \lambda t.(\ if\ t{=}1\ then\ y(1)\ else\ x(t{-}1)\ )\ )(t+1)\ )\ ) =$ *conversión* β

$= (\ \lambda t.(\ if\ t{+}1{=}1\ then\ y(1)\ else\ x(t{+}1{-}1)\ )\ ) =$ *t+1 con* $t \in \mathbb{N}_+$ *nunca será igual a 1*

$= (\ \lambda t.(\ x(t)\ )\ ) =$ *conversión* η

$= (\ x\ )$

□

INEXT: función del operador *fby* como operador inverso de *next*

$$\forall x \in Lu(A)_s,\ fby(\ x,\ next\ x\ ) = x$$

DEMOSTRACIÓN. Se realiza reduciendo la expresión izquierda a la derecha.

$fby(\ x,\ next\ x\ ) =$ *definición de fby, conversión* β

$= \lambda t.(\ if\ t{=}1\ then\ x(1)\ else\ (next\ x)(t{-}1)\ ) =$ *definición de next, conversión* β

$= \lambda t.(\ if\ t{=}1\ then\ x(1)\ else\ x(t{-}1{+}1)\ ) =$

$= \lambda t.(\ if\ t{=}1\ then\ x(1)\ else\ x(t)\ ) =$ *semántica de if-then-else*

$= \lambda t.(\ x(t)\ ) =$ *conversión* η

$= x$

□

IREP: función del operador *sample* como operador inverso de *replicate*.

$$\forall k \in \mathbb{N}_+,\ \forall x \in Lu(A)_s,\ sample(\ k,\ replicate(\ x,\ k\ )\ ) = x$$

DEMOSTRACIÓN. Se realiza reduciendo la expresión izquierda a la derecha.

$sample(\ k,\ replicate(\ x,\ k\ )\ ) =$ *definición de sample, conversión* β

$= \lambda t.(\ replicate(\ x,\ k\ )(t*k)\ ) =$ *definición de replicate, conversión* β

$= \lambda t.(\ x(ceil((t*k)/k)\ ) =$ $k \in \mathbb{N}_+ \Rightarrow ceil(t) = t$

$= \lambda t.(\ x(t)\ ) =$ *conversión* η

$= x$

□

4.2.2 Distributividad de los operadores temporales respecto a los no temporales.

Si con las anteriores propiedades era posible colocar pares de operadores temporales en cualquier lugar de una especificación ecuacional, la utilidad de estas propiedades es poder distribuirlos a conveniencia entre las definiciones del cuerpo ecuacional. Por ejemplo, cuando se trata del operador *fby*, esta redistribución se denomina en el ámbito del diseño hardware como retemporización (véase §3.3.1).

Obsérvese que las propiedades se fijan para toda función $A_\sigma^{w,s}$ soporte de cualquier símbolo de operación de cualquier signatura Σ, lo que viene a significar, que son válidas para toda función que sea una *Lu*-extensión de una función estática.

DFBY: ditributividad del operador *fby*.

$$\forall A_\sigma^{w,s},\ \forall x_i,y_i \in Lu(A)_{si},\ Lu(A)_\sigma^{w,s}(\ fby(\ y_1,\ x_1\ ),\ \overset{n}{\ldots},\ fby(\ y_n,\ x_n\ )\ ) =$$
$$= fby(\ Lu(A)_\sigma^{w,s}(\ y_1,\ \overset{n}{\ldots},\ y_n\ ),\ Lu(A)_\sigma^{w,s}(\ x_1,\ \overset{n}{\ldots},\ x_n\ )\ )$$

donde n es el número de argumentos que toma la función no temporal $A_\sigma^{w,s}$ y que está fijado en la signatura por la aridad w.

DEMOSTRACIÓN. Se realiza demostrando por casos, que en todo instante temporal $t \in \mathbf{N}_+$, las expresiones izquierda y derecha se reducen a una forma común.

(1) En caso de que $t = 1$:

(1.1) Reducción de la expresión izquierda:

$Lu(A)^{w,s}_{\sigma}(\ fby(y_1,x_1), \overset{n}{\ldots}, fby(y_n,x_n)\ )(1) =$ *definición de Lu-extensión, conversión β*

$= \lambda t.A^{w,s}_{\sigma}(\ fby(y_1,x_1)(t), \overset{n}{\ldots}, fby(y_n,x_n)(t)\ )(1) =$ *conversión β*

$= A^{w,s}_{\sigma}(\ fby(y_1,x_1)(1), \overset{n}{\ldots}, fby(y_n,x_n)(1)\ ) =$ *definición de fby, conversiones β*

$= A^{w,s}_{\sigma}(\ (\mathit{if}\ 1{=}1\ \mathit{then}\ y_1(1)\ \mathit{else}\ x_1(1\text{-}1)), \overset{n}{\ldots}, (\mathit{if}\ 1{=}1\ \mathit{then}\ y_n(1)\ \mathit{else}\ x_n(1\text{-}1))\ ) =$

$= A^{w,s}_{\sigma}(\ y_1(1), \overset{n}{\ldots}, y_n(1)\ )$

(1.2) Reducción de la expresión derecha:

$fby(\ Lu(A)^{w,s}_{\sigma}(y_1,\overset{n}{\ldots},y_n), Lu(A)^{w,s}_{\sigma}(x_1,\overset{n}{\ldots},x_n)\ )(1) =$ *definición de fby, conversiones β*

$= (\mathit{if}\ 1{=}1\ \mathit{then}\ Lu(A)^{w,s}_{\sigma}(y_1,\overset{n}{\ldots},y_n)(1)\ \mathit{else}\ Lu(A)^{w,s}_{\sigma}(x_1,\overset{n}{\ldots},x_n)(1\text{-}1)) =$

$= Lu(A)^{w,s}_{\sigma}(\ y_1, \overset{n}{\ldots}, y_n\ )(1) =$ *definición de Lu-extensión, conversión β*

$= \lambda t.A^{w,s}_{\sigma}(\ y_1(t), \overset{n}{\ldots}, y_n(t)\ )(1) =$ *conversión β*

$= A^{w,s}_{\sigma}(\ y_1(1), \overset{n}{\ldots}, y_n(1)\ )$

(2) En caso de que $t > 1$:

(2.1) Reducción de la expresión izquierda:

$Lu(A)^{w,s}_{\sigma}(\ fby(y_1,x_1), \overset{n}{\ldots}, fby(y_n,x_n)\ )(t) =$ *definición de Lu-extensión, conversión β*

$= \lambda t.A^{w,s}_{\sigma}(\ fby(y_1,x_1)(t), \overset{n}{\ldots}, fby(y_n,x_n)(t)\ )(t) =$ *conversión β*

$= A^{w,s}_{\sigma}(\ fby(y_1,x_1)(t), \overset{n}{\ldots}, fby(y_n,x_n)(t)\ ) =$ *definición de fby, conversiones β*

$= A^{w,s}_{\sigma}((\mathit{if}\ t{=}1\ \mathit{then}\ y_1(1)\ \mathit{else}\ x_1(t\text{-}1)),\overset{n}{\ldots},(\mathit{if}\ t{=}1\ \mathit{then}\ y_n(1)\ \mathit{else}\ x_n(t\text{-}1))) =$

$t{>}1 \Rightarrow t{\neq}1$

$= A^{w,s}_{\sigma}(\ x_1(t\text{-}1), \overset{n}{\ldots}, x_n(t\text{-}1)\ )$

(2.2) Reducción de la expresión derecha:

$fby(\ Lu(A)^{w,s}_{\sigma}(y_1,\overset{n}{\ldots},y_n), Lu(A)^{w,s}_{\sigma}(x_1,\overset{n}{\ldots},x_n)\ )(t) =$ *definición de fby, conversiones β*

$= (\mathit{if}\ t{=}1\ \mathit{then}\ Lu(A)^{w,s}_{\sigma}(y_1,\overset{n}{\ldots},y_n)(1)\ \mathit{else}\ Lu(A)^{w,s}_{\sigma}(x_1,\overset{n}{\ldots},x_n)(t\text{-}1)) =$ $t{>}1 \Rightarrow t{\neq}1$

$= Lu(A)^{w,s}_{\sigma}(\ x_1, \overset{n}{\ldots}, x_n\ )(t\text{-}1) =$ *definición de Lu-extensión, conversión β*

$= \lambda t.A^{w,s}_{\sigma}(\ x_1(t), \overset{n}{\ldots}, x_n(t)\ )(t\text{-}1) =$ *conversión β*

$= A_{\sigma}^{w,s}(x_1(t\text{-}1), \overset{n}{\ldots}, x_n(t\text{-}1))$

□

DNEXT: distributividad del operador *next*.

$$\forall A_{\sigma}^{w,s}, \forall x_i \in Lu(A)_{si}, Lu(A)_{\sigma}^{w,s}(next(x_1), \overset{n}{\ldots}, next(x_n)) = \\ = next(Lu(A)_{\sigma}^{w,s}(x_1, \overset{n}{\ldots}, x_n))$$

DEMOSTRACIÓN. Se realiza reduciendo las expresiones izquierda y derecha a una forma común.

(1) Reducción de la expresión izquierda:

$Lu(A)_{\sigma}^{w,s}(next(x_1), \overset{n}{\ldots}, next(x_n)) =$ *definición de Lu-extensión, conversión β*

$= \lambda t.A_{\sigma}^{w,s}(next(x_1)(t), \overset{n}{\ldots}, next(x_n)(t)) =$ *definición de next, conversión β*

$= \lambda t.A_{\sigma}^{w,s}(x_1(t+1), \overset{n}{\ldots}, x_n(t+1))$

(2) Reducción de la expresión derecha:

$next(Lu(A)_{\sigma}^{w,s}(x_1, \overset{n}{\ldots}, x_n)) =$ *definición de next, conversión β*

$= \lambda t.Lu(A)_{\sigma}^{w,s}(x_1, \overset{n}{\ldots}, x_n)(t+1) =$ *definición de Lu-extensión, conversión β*

$= \lambda t.(\lambda t.A_{\sigma}^{w,s}(x_1(t), \overset{n}{\ldots}, x_n(t))(t+1) =$ *conversión β*

$= \lambda t.A_{\sigma}^{w,s}(x_1(t+1), \overset{n}{\ldots}, x_n(t+1))$

□

DREP: Distributividad del operador *replicate*.

$$\forall A_{\sigma}^{w,s}, \forall k \in \mathbf{N}_+, \forall x_i\ Lu(A)_{si}, \\ Lu(A)_{\sigma}^{w,s}(replicate(x_1, k), \overset{n}{\ldots}, replicate(x_n, k)) = \\ = replicate(Lu(A)_{\sigma}^{w,s}(x_1, \overset{n}{\ldots}, x_n), k)$$

DEMOSTRACIÓN. Se realiza reduciendo las expresiones izquierda y derecha a una forma común.

(1) Reducción de la expresión izquierda:

$Lu(A)^{w,s}_{\sigma}(\ replicate(x_1,k),\ \overset{n}{\ldots},\ replicate(x_n,k)\ ) =$ *definición de Lu-extensión, conv.* β

$= \lambda t.A^{w,s}_{\sigma}(\ replicate(x_1,k)(t),\ \overset{n}{\ldots},\ replicate(x_n,k)(t) =$ *definición de replicate, conv.* β

$= \lambda t.A^{w,s}_{\sigma}(\ x_1(ceil(t/k)),\ \overset{n}{\ldots},\ x_n(ceil(t/k))\ )$

(2) Reducción de la expresión derecha:

$replicate(\ Lu(A)^{w,s}_{\sigma}(\ x_1,\ \overset{n}{\ldots},\ x_n\ ),\ k\ ) =$ *definición de replicate, conversión* β

$= \lambda t.Lu(A)^{w,s}_{\sigma}(\ x_1,\ \overset{n}{\ldots},\ x_n\ )(ceil(t/k)) =$ *definición de Lu-extensión, conversión* β

$= \lambda t.(\ \lambda t.A^{w,s}_{\sigma}(\ x_1(t),\ \overset{n}{\ldots},\ x_n(t)\ )\ )(ceil(t/k)) =$ *conversión* β

$= \lambda t.A^{w,s}_{\sigma}(\ x_1(ceil(t/k)),\ \overset{n}{\ldots},\ x_n(ceil(t/k))\ )$

□

DSAM: distributividad del operador *sample*.

$$\forall A^{w,s}_{\sigma},\ \forall k \in \mathbf{N}_+,\ \forall x_i\ Lu(A)_{si},$$
$$Lu(A)^{w,s}_{\sigma}(\ sample(\ k,\ x_1\ ),\ \overset{n}{\ldots},\ sample(\ k,\ x_n\ )\ ) =$$
$$= sample(\ k,\ Lu(A)^{w,s}_{\sigma}(\ x_1,\ \overset{n}{\ldots},\ x_n\ )\ )$$

DEMOSTRACIÓN. Se realiza reduciendo las expresiones izquierda y derecha a una forma común.

(1) Reducción de la expresión izquierda:

$Lu(A)^{w,s}_{\sigma}(\ sample(k,x_1),\ \overset{n}{\ldots},\ sample(k,x_n)\ ) =$ *definición de Lu-extensión, conversión* β

$= \lambda t.A^{w,s}_{\sigma}(\ sample(k,x_1)(t),\ \overset{n}{\ldots},\ sample(k,x_n)(t)\ ) =$ *definición de sample, conversión* β

$= \lambda t.A^{w,s}_{\sigma}(\ x_1(t*k),\ \overset{n}{\ldots},\ x_n(t*k)\ )$

(2) Reducción de la expresión derecha:

$sample(\ k,\ Lu(A)^{w,s}_{\sigma}(\ x_1,\ \overset{n}{\ldots},\ x_n\ )\ ) =$ *definición de sample, conversión* β

$= \lambda t.Lu(A)^{w,s}_{\sigma}(\ x_1,\ \overset{n}{\ldots},\ x_n\ )(t*k) =$ *definición de Lu-extensión, conversión* β

$= \lambda t.(\ \lambda t.A^{w,s}_{\sigma}(\ x_1(t),\ \overset{n}{\ldots},\ x_n(t)\ )\ )(t*k) =$ *conversión* β

$= \lambda t.A^{w,s}_{\sigma}(\ x_1(t*k),\ \overset{n}{\ldots},\ x_n(t*k)\ )$

□

DINT: distributividad del operador *interleave*.

$$\forall A_\sigma^{w,s},\ \forall k\in\mathbf{N}_+,\ \forall x_{i,j}\in Lu(A)_{si},$$
$$Lu(A)_\sigma^{w,s}(\ interleave(\ x_{1,1},\overset{k}{\ldots},x_{1,k}\ ),\overset{n}{\ldots},\ interleave(\ x_{n,1},\overset{k}{\ldots},x_{n,k}\ )\ ) =$$
$$= interleave(\ Lu(A)_\sigma^{w,s}(\ x_{1,1},\overset{n}{\ldots},x_{n,1}\ ),\overset{k}{\ldots},\ Lu(A)_\sigma^{w,s}(\ x_{1,k},\overset{n}{\ldots},x_{n,k}\ )\ )$$

Obsérvese, como se dijo anteriormente, que k no es una variable del formalismo sino una variable del metalenguaje (o sea, del lenguaje informal que utilizo para hablar sobre el formalismo). Esto permite que esta fórmula no describa una única definición formal, sino una representación esquemática de un conjunto infinito de definiciones del formalismo.

DEMOSTRACIÓN. Se realiza demostrando que en todo instante temporal $t\in\mathbf{N}_+$, las expresiones izquierda y derecha se reducen a una forma común.

(1) Reducción de la expresión izquierda para cierto t:

$Lu(A)_\sigma^{w,s}(\ interleave(x_{1,1},\overset{k}{\ldots},x_{1,k}),\overset{n}{\ldots},\ interleave(x_{n,1},\overset{k}{\ldots},x_{n,k})\ )(t) =$

definición de Lu-extensión, conversiones β

$= A_\sigma^{w,s}(\ interleave(x_{1,1},\overset{k}{\ldots},x_{1,k})(t),\overset{n}{\ldots},\ interleave(x_{n,1},\overset{k}{\ldots},x_{n,k})(t)\ ) =$

definición de interleave, conversiones β

$= A_\sigma^{w,s}((x_{1,1}(t),\overset{k}{\ldots},x_{1,k}(t))\downarrow((t\text{-}1)\ mod\ k)+1),\overset{n}{\ldots},(x_{n,1}(t),\overset{k}{\ldots},x_{n,k}(t))\downarrow((t\text{-}1)\ mod\ k)+1)) =$

dado que t está fijado, sea $\alpha = ((t\text{-}1)\ mod\ k)+1) \Rightarrow t = \alpha*k\text{-}k+1$

$= A_\sigma^{w,s}(\ x_{1,\alpha}(\alpha*k\text{-}k+1),\overset{n}{\ldots},\ x_{n,\alpha}(\alpha*k\text{-}k+1)\ )$

(2) Reducción de la expresión derecha para cierto t:

$interleave(\ Lu(A)_\sigma^{w,s}(\ x_{1,1},\overset{n}{\ldots},x_{n,1}\ ),\overset{k}{\ldots},\ Lu(A)_\sigma^{w,s}(\ x_{1,k},\overset{n}{\ldots},x_{n,k}\ )\ )(t) =$

definición de interleave, conversiones β

$= (\ Lu(A)_\sigma^{w,s}(\ x_{1,1},\overset{n}{\ldots},x_{n,1}\ )(t),\overset{k}{\ldots},\ Lu(A)_\sigma^{w,s}(\ x_{1,k},\overset{n}{\ldots},x_{n,k}\ )(t)\ )\downarrow((t\text{-}1)\ mod\ k)+1) =$

sea nuevamente $\alpha = ((t\text{-}1)\ mod\ k)+1)$

$= Lu(A)_\sigma^{w,s}(\ x_{1,\alpha},\overset{n}{\ldots},x_{n,\alpha}\ )(\alpha*k\text{-}k+1) =$ *definición de Lu-extensión, conversiones β*

$= A_\sigma^{w,s}(\ x_{1,\alpha}(\alpha*k\text{-}k+1),\overset{n}{\ldots},\ x_{n,\alpha}(\alpha*k\text{-}k+1)\ )$

□

4.2.3 Existencia de elementos neutros.

Estas propiedades establecen que toda secuencia constante (formada por la repetición del mismo elemento) es inmune al efecto de los operadores temporales y, en el caso del operador *interleave*, se puede ser más general demostrando que el resultado de intercalar cualquier señal consigo misma es la propia señal.

NFBY: caracterización de los elementos neutros del operador *fby*.

$$\forall c \in (A_{\perp}^{\#})_s,\ fby(\ \lambda t.c,\ \lambda t.c\ ) = \lambda t.c$$

DEMOSTRACIÓN. Se realiza por reducción de la expresión izquierda a la derecha.

$fby(\ \lambda t.c,\ \lambda t.c\ ) =$ *definición de fby, conversión β*

$= \lambda t.if\ t{=}1\ then\ \lambda t.c(1)\ else\ \lambda t.c(t{-}1) =$ *conversión β*

$= \lambda t.if\ t{=}1\ then\ c\ else\ c =$ *semántica de if-then-else*

$= \lambda t.c$

□

NNEXT: caracterización de los elementos neutros del operador *next*.

$$\forall c \in (A_{\perp}^{\#})_s,\ next(\ \lambda t.c\ ) = \lambda t.c$$

DEMOSTRACIÓN. Se realiza por reducción de la expresión izquierda a la derecha.

$next(\ \lambda t.c\ ) =$ *definición de next, conversión β*

$= \lambda t.\lambda t.c(t{+}1) =$ *conversión β*

$= \lambda t.c$

□

NREP: caracterización de los elementos neutros del operador *replicate*.

$$\forall c \in (A_{\perp}^{\#})_S,\ \forall k \in \mathbf{N}_+,\ replicate(\ \lambda t.c,\ k\ ) = \lambda t.c$$

DEMOSTRACIÓN. Se realiza por reducción de la expresión izquierda a la derecha.

$replicate(\ \lambda t.c,\ k\ ) =$ *definición de replicate, conversión β*

$= \lambda t.\lambda t.c(ceil(t/n)) =$ *conversión β*

$= \lambda t.c$

□

NSAM: caracterización de los elementos neutros del operador *sample*.

$$\forall c \in (A_{\perp}^{\#})_S,\ \forall k \in \mathbf{N}_+,\ sample(\ k,\ \lambda t.c\ ) = \lambda t.c$$

DEMOSTRACIÓN. Se realiza por reducción de la expresión izquierda a la derecha.

$sample(\ k,\ \lambda t.c\ ) =$ *definición de sample, conversión β*

$= \lambda t.\lambda t.c(t*n) =$ *conversión β*

$= \lambda t.c$

□

NINT: caracterización de los elementos neutros del operador *interleave*.

$$\forall k \in \mathbf{N}_+,\ \forall x \in Lu(A)_S\ interleave(\ x,\ \overset{k}{\ldots},\ x\ ) = x$$

DEMOSTRACIÓN. Se realiza demostrando que en todo instante temporal $t \in \mathbf{N}_+$, la expresión izquierda se reduce a la derecha.

$interleave(\ x,\ \overset{k}{\ldots},\ x\ )(t) =$ *definición de interleave, conversión β*

$= (\ \lambda t.(\ x(t),\ \overset{k}{\ldots},\ x(t)\ )\downarrow((t\text{-}1)\ mod\ k)+1\ )(t) =$ *conversión β*

$= (x(t), \overset{k}{\ldots}, x(t))\downarrow((t\text{-}1) \bmod k)+1 =$ *semántica de* $\downarrow$

$= x(t)$

□

4.2.4 Teoremas de síntesis.

La siguiente colección de propiedades dejan de establecer hechos generales sobre el comportamiento de los operadores temporales y se centran a formalizar aspectos claves de algunas fases de la síntesis de alto nivel.

TMT: teorema de multiplexación temporal.

Este teorema resume, mediante una única fórmula, la base de la planificación de operaciones, es decir, permite asignar una operación a un ciclo. La idea que utiliza es la siguiente: si los valores que transporta una señal son leídos (para realizar cierto cálculo) a una frecuencia menor que la frecuencia a la que son producidos, muchos de ellos no tendrán efecto alguno sobre el cómputo a realizar ya que el lector es incapaz de procesarlos. Dado que estos valores son calculados por un cierto operador rápido, todos aquellos ciclos en los que se calcule un valor que no pueda ser leído, podrán ser utilizados para realizar cualquier otro cálculo más útil. Esto podrá hacerse, siempre y cuando, el operador reciba sus argumentos en el momento adecuado y el valor leído sea almacenado el número de ciclos suficiente para que llegue a tiempo a su destino.

Dicha idea (véanse §4.5.1 y §4.5.2 para encontrar ejemplos de su utilización) se formaliza seguidamente:

$$\forall x \in Lu(A)_s,\ \forall k \in \mathbf{N}_+,\ \forall m \in \mathbf{N} \mid m<k,$$
$$sample(\ k,\ x\ ) =$$
$$= sample(\ k,\ fby(\lambda t.\#)^m(interleave(\ \lambda t.\#,\ \overset{k-m-1}{\ldots},\ \lambda t.\#,\ next^m(x),\ \lambda t.\#,\ \overset{m}{\ldots},\ \lambda t.\#\ ))\ )$$

Obsérvese cómo se han utilizando los operadores temporales. Primero, mediante el operador *sample*, se ha expresado la razón *k*, entre las frecuencias de lectura y escritura. Desde el punto de vista de la SAN, esta razón no es otra cosa que la latencia de la planificación. Después, la indiferencia sobre los valores transportados pero no leídos queda formalizada mediante las secuencias de elementos comodín†. La asignación de la lectura del valor que transporta x_m al ciclo *k-m*, utiliza el operador *interleave*. Finalmente, mediante operadores *fby* con valores iniciales cualesquiera (notados con elementos comodín), se ha fijado el número de ciclos que como máximo es necesario que el valor calculado permanezca almacenado (*m* ciclos, es decir, hasta el último ciclo de la planificación). De estos *fby* muchos desaparecerán (vía la propiedad IFBY) con los operadores *next* que aparezcan en la planificación de los sucesores de x_m.

Obsérvese que mediante *fby* y *next* se fijan los requisitos de temporización de dicha planificación individual suponiendo que el resto de las operaciones no están planificadas. Esa es la razón para que, a la hora de elegir el número de *next* y *fby* que forman una expresión concreta, se tome como referencia el último ciclo de la planificación, por corresponderse en el tiempo con el final del ciclo de muestreo.

---

† Esta misma circunstancia podría ser expresada mediante una colección de variables universales que sólo aparecieran en el lado derecho de la fórmula:

$$\forall x_i, z_i \in Lu(A)_s,\ \forall k \in \mathbf{N}_+,\ \forall m \in \mathbf{N} \mid m<k,$$
$$sample(\ k,\ x_m\ ) =$$
$$= sample(\ k,\ fby(\ z_i\ )^m(\ interleave(\ x_1,\ \overset{k-m-1}{\ldots},\ x_{m-1},\ next^m(\ x_m\ ),\ x_{m+1},\ \overset{m}{\ldots},\ x_k\ )\ )\ )$$

Sin embargo, esta formulación traería problemas una vez que se formalizarse como una Lu(Σ)-ecuación y se intentara aplicar de izquierda a derecha usando el sistema de transformación ya que, claramente, var(R) ⊄ var(L).

En cuanto a la notación, se ha utilizado la versión currificada† del operador *fby* y la potenciación de funciones. Así *fby*($\lambda t.\#$) representa a un operador de un único argumento obtenido a partir de la definición de *fby* y que antepone a cualquier secuencia un elemento comodín. Por su parte, si *f* es una función de un argumento, f^n es la aplicación sucesiva de *f*, *n* veces, indicando f^0 la no aplicación de la función (o la aplicación de la función identidad).

Para facilitar la demostración tanto de este teorema como de otros teoremas de síntesis, demostraré dos lemas bastante obvios que clarifican el significado de las potencias de *fby* (lema 4.1) y de *next* (lema 4.2).

4.1 **LEMA.** $\forall x,y \in Lu(A)_s$, $\forall m \in \mathbb{N}$, $fby(x)^m(y) = \lambda t.\, if\ t \leq m\ then\ x(1)\ else\ y(t-m)$

DEMOSTRACIÓN. Por inducción.

(1) Caso base:

$fby(x)^0(y) =$ — *definición de potencias de funciones*

$= y =$ — *conversión* η

$= \lambda t.y(t) =$ — $t \in \mathbb{N}_+$

$= \lambda t.\, if\ t \leq 0\ then\ x(1)\ else\ y(t)$

(2) Paso de inducción:

$fby(x)^{n+1}(y) =$ — *definición de potencias de funciones*

$= fby(x)^n(fby(x,y)) =$ — *hipótesis de inducción*

$= \lambda t.\, if\ t \leq n\ then\ x(1)\ else\ fby(x,y)(t-n) =$ — *definición de fby, conversiones* β

$= \lambda t.\, if\ t \leq n\ then\ x(1)$

$\quad else\ (if\ t-n=1\ then\ x(1)\ else\ y(t-n-1)) =$ — *despejando t de la condición del if*

$= \lambda t.\, if\ t \leq n\ then\ x(1)$

$\quad else\ (if\ t=n+1\ then\ x(1)\ else\ y(t-n-1)) =$ — *simplificando la anidación de if*

$= \lambda t.\, if\ t \leq n+1\ then\ x(1)\ else\ y(t-n-1)$

□

† Véase §A.4.

4.2 **LEMA.** $\forall x \in Lu(A)_s \; \forall m \in \mathbb{N} \; next^m(x) = \lambda t.x(t+m)$

DEMOSTRACIÓN. Por inducción.

(1) Caso base:

$next^0(x) =$ *definición de potencias de funciones*

$= x =$ *conversión* η

$= \lambda t.x(t)$

(2) Paso de inducción:

$next^{n+1}(x) =$ *definición de potencias de funciones*

$= next^n(\, next(x)\,) =$ *hipótesis de inducción*

$= \lambda t.next(x)(t+n) =$ *definición de next, conversiones* β

$= \lambda t.x(t+n+1)$

□

Pasemos, entonces, a la demostración del teorema de multiplexación temporal.

DEMOSTRACIÓN. Se realiza reduciendo los términos izquierdo y derecho a una forma común.

(1) Reducción de la expresión izquierda:

$sample(\, k, x\,) =$ *definición sample, conversión* β

$= \lambda t.x(t*k)$

(2) Reducción de la expresión derecha:

$sample(\, k, fby(\lambda t.\#\,)^m(interleave(\, \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, next^m(x), \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\,))\,) =$

definición de sample, conversión β

$= \lambda t.(\, fby(\lambda t.\#)^m(interleave(\, \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, next^m(x), \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\,))\,)(t*k) =$

Lema 4.1, conversiones β

$= \lambda t.(\, if\ t*k \le m\ then\ \#\ else$ $m<k,\ t \in \mathbb{N}_+ \Rightarrow m<t*k$

$\quad interleave(\, \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, next^m(x), \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\,)(t*k-m)\,) =$

$= \lambda t.(\, interleave(\, \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, next^m(x), \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\,)(t*k-m)\,) =$ (*)

$= \lambda t.next^m(\, x\,)(t*k-m) =$ *Lema 4.2, conversión* β

$= \lambda t.x(t*k)$ □

(*) Demostraré, por reducción de la expresión izquierda a la derecha, un resultado algo más general:

$\forall x_i \in Lu(A)_s,\ \forall k \in \mathbf{N}_+,\ \forall m \in \mathbf{N} \mid m<k,\ interleave(\ x_1, \overset{k}{\ldots}, x_k\ )(t*k-m) = x_{k-m}(t*k-m)$

$interleave(\ x_1, \overset{k}{\ldots}, x_k\ )(t*k-m) =$ *definición de interleave, conversión β*

$= (\ x_1(t*k-m), \overset{k}{\ldots}, x_k(t*k-m)\ )\downarrow(((t*k-m)-1)\ mod\ k)+1 =$

0<m<k ⇒ k-m-1<k ⇒ (k-m-1) mod k = k-m-1

$= (\ x_1(t*k-m), \overset{k}{\ldots}, x_k(t*k-m)\ )\downarrow(k-m) =$ *semántica de ↓*

$= x_{k-m}(t*k-m)$ □

□

ADRET: teorema de reemplazo de retardos arquitectónicos.

Un retardo arquitectónico es aquel que conserva valores entre dos iniciaciones de un algoritmo (cuando se necesita conservar un valor entre más de dos iniciaciones, se encadenan varios de ellos). Este tipo de retardos se caracterizan por tener una frecuencia de carga igual a la frecuencia de muestreo del sistema y aparecen inicialmente, en forma de operadores *fby*, en toda especificación ecuacional que describa un cálculo recursivo o iterativo.

Sin embargo, cuando se realiza síntesis de alto nivel, es necesario un nuevo tipo de retardo auxiliar que almacene valores temporalmente dentro de una misma iniciación y que tenga, por tanto, una frecuencia de carga mayor. Este tipo de retardos, además de no conservar valores entre iniciaciones, pueden reutilizarse para almacenar en distintos intervalos de tiempo valores producidos por distintos operadores de la especificación.

Este teorema formaliza la base de la reutilización de los retardos arquitectónicos para almacenar valores temporales en aquellos ciclos en los que no son observables. Para plasmar esta idea, convierte los registros

arquitectónicos, cuya lectura ha podido ser planificada con el anterior teorema, en cadenas de retardos auxiliares de cierta longitud, longitud que cuando sus sucesores se planifiquen, podrá ser menor que la latencia de la planificación e indicará en qué ciclos puede ser reutilizado el retardo arquitectónico.

$$\forall x,y \in Lu(A)_S,\ \forall k \in \mathbb{N}_+ \mid m<k,$$
$$interleave(\lambda t.\#,\overset{k-m-1}{\ldots},\lambda t.\#,next^m(replicate(fby(y,sample(k,x)),k)),\lambda t.\#,\overset{m}{\ldots},\lambda t.\#) =$$
$$= fby(\lambda t.\#)^{k-m-1}(fby(y)(fby(\lambda t.\#)^m interleave(\lambda t.\#,\overset{k-m-1}{\ldots},\lambda t.\#,next^m(x),\lambda t.\#,\overset{m}{\ldots},\lambda t.\#)))$$

Como puede observarse, este teorema permite iluminar un aspecto oscuro de la síntesis conductual. En muchos de los sistemas convencionales de síntesis, los retardos arquitectónicos son tratados sin necesidad de un modo especial: sobre ellos no se realiza el clásico análisis de tiempo de vida de sus contenidos y, para evitar problemas de antidependencia, se les asigna un registro dedicado planificando su carga en el último ciclo. El efecto es que no pueden reutilizarse. Gracias a este teorema, los retardos arquitectónicos pueden tratarse como conjuntos de retardos convencionales independientes y las antidependencias como conjuntos de dependencias, permitiendo una solución unificada al problema del reuso de los elementos de memoria.

DEMOSTRACIÓN. Obtiene una definición por partes de las funciones que describen cada una de las expresiones que forman la ecuación y, comparando dichas definiciones, puede concluirse la equivalencia de las mismas.

(1) Estudio el valor en el tiempo de la función descrita por la expresión izquierda de la ecuación:

$$interleave(\lambda t.\#,\overset{k-m-1}{\ldots},\lambda t.\#,next^m(replicate(fby(y,sample(k,x)),k)),\lambda t.\#,\overset{m}{\ldots},\lambda t.\#)(t) =$$

definición de interleave, conversión β

$$= (\#,\overset{k-m-1}{\ldots},\#,next^m(replicate(fby(y,sample(k,x)),k))(t),\#,\overset{m}{\ldots},\#)\downarrow(((t\text{-}1)\ mod\ k)+1) =$$

Lema 4.2, conversión β

$= (\#,\overset{k-m-1}{\ldots},\#,replicate(fby(y,sample(k,x)),k)(t+m),\#,\overset{m}{\ldots},\#)\downarrow(((t\text{-}1)\ mod\ k)+1) =$

definición de replicate, conversiones β

$= (\#,\overset{k-m-1}{\ldots},\#,fby(y,sample(k,x))(ceil((t+m)/k)),\#,\overset{m}{\ldots},\#)\downarrow(((t\text{-}1)\ mod\ k)+1) =$

$= (\ \#,\overset{k-m-1}{\ldots}\ ,\ \#\ ,\ if\ ceil((t+m)/k)=1\ then\ y(1)\ else$ *def. fby, conversiones β*

$sample(k,x)(ceil((t+m)/k)\text{-}1)\ ,\ \#\ ,\overset{m}{\ldots}\ ,\ \#\ )\downarrow(((t\text{-}1)\ mod\ k)+1) =$

$= (\ \#,\overset{k-m-1}{\ldots},\ \#,\ if\ ceil((t+m)/k)=1\ then\ y(1)\ else$ *def. sample, conversiones β*

$x((ceil((t+m)/k)\text{-}1)*k),\ \#,\overset{m}{\ldots},\ \#\ )\downarrow(((t\text{-}1)\ mod\ k)+1)$

Como puede observarse, el resultado es una k-tupla indexada por la expresión $((t\text{-}1)\ mod\ k)+1$ cuyo valor claramente varía entre 1 y k. En dicha k-tupla sólo existen dos tipos de componentes: por un lado, la componente $k\text{-}m$ que vale $if\ ceil((t+m)/k)=1\ then\ y(1)\ else\ x((ceil((t+m)/k)\text{-}1)*k)$ y, por otro, las primeras $k\text{-}m\text{-}1$ y las últimas m componentes que valen todas #.

(1.1) Calculo los valores de t en los que se indexa la única componente distinta de #, es decir, aquellos que satisfacen $((t\text{-}1)\ mod\ k)+1 = k\text{-}m$:

$$((t\text{-}1)\ mod\ k)+1 = k\text{-}m \Leftrightarrow t = (\alpha+1)*k\text{-}m,\ \text{con}\ \alpha\in\mathbb{N}$$

Substituyendo este valor de t sobre la expresión que se indexa, podrán conocerse los valores que toma la función en esos instantes temporales:

$if\ ceil(((\alpha+1)*k\text{-}m+m)/k)=1\ then\ z(1)\ else\ x((ceil(((\alpha+1)*k\text{-}m+m)/k)\text{-}1)*k) =$

$\alpha\in\mathbb{N} \Rightarrow ceil(\alpha+1) = \alpha+1$

$= if\ \alpha=0\ then\ z(1)\ else\ x(\alpha*k)$

Que valdrá $z(1)$ si $t=k\text{-}m$ y $x(\alpha*k)$ si $t = (\alpha+1)*k\text{-}m$, con $\alpha\in\mathbb{N}_+$, o lo que es lo mismo $x((\alpha+1)*k)$ si $t = (\alpha+2)*k\text{-}m$, con $\alpha\in\mathbb{N}$.

(1.2) En resumen, la función definida por el término izquierdo es:

$interleave(\lambda t.\#,\overset{k-m-1}{\ldots},\lambda t.\#,next^m(replicate(fby(y,sample(x,k)),k)),\lambda t.\#,\overset{m}{\ldots},\lambda t.\#)(t) =$

$= y(1)$ si $t = k\text{-}m$

$= x((\alpha+1)*k)$ si $t = (\alpha+2)*k\text{-}m$, con $\alpha\in\mathbb{N}$

$= \#$ en otro caso

(2) Estudio el valor en el tiempo de la función descrita por la expresión derecha de la ecuación.

$fby(\lambda t.\#)^{k-m-1}(fby(y)(fby(\lambda t.\#)^m interleave(\lambda t.\#,\overset{k-m-1}{\ldots},\lambda t.\#,next^m(x),\lambda t.\#,\overset{m}{\ldots},\lambda t.\#)))(t)=$

Lema 4.1, conversiones β

$= \textit{if } t \le k\text{-}m\text{-}1 \textit{ then } \# \textit{ else}$ *definición de fby, conversiones β, aritmética*

$(\ fby(y)(fby(\lambda t.\#)^{m} interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, next^{m}(x), \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)))(t\text{-}k\text{+}m\text{+}1) =$

$= \textit{if } t \le k\text{-}m\text{-}1 \textit{ then } \# \textit{ else}$ *Lema 4.1, conversiones β, aritmética*

$\textit{if } t\text{-}k\text{+}m\text{=}0 \textit{ then } y(t\text{-}k\text{+}m\text{+}1) \textit{ else}$

$(\ fby(\lambda t.\#)^{m} interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, next^{m}(x), \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)\ )(t\text{-}k\text{+}m) =$

$= \textit{if } t \le k\text{-}m\text{-}1 \textit{ then } \# \textit{ else}$ *definición de interleave, conversiones β*

$\textit{if } t\text{-}k\text{+}m\text{=}0 \textit{ then } y(t\text{-}k\text{+}m\text{+}1) \textit{ else}$

$\textit{if } t\text{-}k \le 0 \textit{ then } \# \textit{ else } interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, next^{m}(x), \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(t\text{-}k) =$

$= \textit{if } t \le k\text{-}m\text{-}1 \textit{ then } \# \textit{ else}$ *definición de next, conversiones β*

$\textit{if } t\text{-}k\text{+}m\text{=}0 \textit{ then } y(t\text{-}k\text{+}m\text{+}1) \textit{ else}$

$\textit{if } t\text{-}k \le 0 \textit{ then } \# \textit{ else } (\#, \overset{k-m-1}{\ldots}, \#, next^{m}(\ x\ )(t\text{-}k), \#, \overset{m}{\ldots}, \#) \downarrow ((t\text{-}k\text{-}1) \bmod k)\text{+}1) =$

$= \textit{if } t \le k\text{-}m\text{-}1 \textit{ then } \# \textit{ else}$

$\textit{if } t\text{-}k\text{+}m\text{=}0 \textit{ then } y(t\text{-}k\text{+}m\text{+}1) \textit{ else}$

$\textit{if } t\text{-}k \le 0 \textit{ then } \# \textit{ else } (\ \#, \overset{k-m-1}{\ldots}, \#, x(t\text{-}k\text{+}m), \#, \overset{m}{\ldots}, \#\ ) \downarrow ((t\text{-}k\text{-}1) \bmod k)\text{+}1)$

Como puede observarse, el resultado es una nueva k-tupla anidada dentro de 3 expresiones *if*. Dicha tupla sólo es evaluada cuando $t > k$ y, en ese caso, vale distinto de # sólo cuando se indexa la componente k-m.

(2.1) Calculo los valores de t que hacen que se seleccione la expresión $x(t\text{-}k\text{+}m)$, es decir, los que satisfacen la ecuación $((t\text{-}k\text{-}1) \bmod k)\text{+}1 = k\text{-}m$:

$$((t\text{-}k\text{-}1) \bmod k)\text{+}1 = k\text{-}m \Leftrightarrow t = (\alpha\text{+}2) * k\text{-}m, \text{ con } \alpha \in \mathbb{N}$$

Substituyendo este valor de t en la expresión que se indexa podrán conocerse los valores que toma la función en esos instantes temporales:

$$x((\alpha\text{+}2) * k\text{-}m\text{-}k\text{+}m) = x((\alpha\text{+}1) * k), \text{ con } \alpha \in \mathbb{N}$$

(2.2) En resumen, la función descrita por el término derecho de la ecuación se define por partes como sigue:

$fby(\lambda t.\#)^{k-m-1}\ fby(y)\ fby(\lambda t.\#)^{m}\ interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, next^{m}(x), \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(t) =$

$= \#$ si $t \le k\text{-}m\text{-}1$

$= y(1)$ si $t = k\text{-}m$

$= \#$ si $k\text{-}m\text{+}1 \le t \le k$

$= x((\alpha+1)*k)$ si $t = (\alpha+2)*k-m$, con $\alpha \in \mathbf{N}$

$= \#$ en otro caso

□

FRAG: teorema de fragmentación de cadenas de retardos.

Cuando la lectura de un retardo arquitectónico ha sido planificada en un ciclo posterior al ciclo en que se ha planificado su actualización, un único registro no es suficiente para implementar correctamente este comportamiento. La razón es que dado que dichas acciones (escritura y lectura) deben acontecer en iniciaciones distintas y consecutivas del algoritmo, el valor arquitectónico tiene una vida mayor que la latencia de la planificación. Para implementar correctamente esta conducta es necesario partir dicha vida en dos de tal manera que cada fragmento se asigne a un registro separado, el primero de ellos tendrá todos sus ciclos ocupados y el segundo tantos ciclos ocupados como número de ciclos separen las acciones de lectura y escritura.

Dicho fenómeno aparece, en el formalismo propuesto, como una cadena de retardos con una longitud mayor que el número de argumentos del operador *interleave* que indica el ciclo en el que se efectúa la planificación de la actualización del retardo arquitectónico.

$$\forall x \in Lu(A)_s,\ \forall k \in \mathbf{N}_+,\ \forall n,m \in \mathbf{N} \mid m<k,$$

$$fby(\lambda t.\#)^n fby(\lambda t.\#)^{k-m-1} fby(y)\ fby(\lambda t.\#)^m interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#) =$$

$$= fby(\#)^n\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x', \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ )$$

$$\text{donde } x' = fby(\#)^{k-m-1}\ fby(y)\ fby(\lambda t.\#)^m\ interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)$$

DEMOSTRACIÓN. Obtiene una definición por partes de las funciones que describen cada una de las expresiones que forman la ecuación y, comparando dichas definiciones, concluye la equivalencia de las mismas.

(1) Estudio el valor en el tiempo de la función descrita por la expresión izquierda de la ecuación:

$fby(\lambda t.\#)^n fby(\lambda t.\#)^{k-m-1} fby(y)\ fby(\lambda t.\#)^m interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#) =$

Lema 4.1, $fby(\lambda t.\#)^n fby(\lambda t.\#)^{k-m-1} = fby(\lambda t.\#)^{n+k-m-1}$

$= \lambda t.$ *if* $t \le n+k-m-1$ *then* # *else* *definición de fby, conversiones* β

$fby(y)\ fby(\lambda t.\#)^m\ interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(t-n-k+m+1) =$

$= \lambda t.$ *if* $t \le n+k-m-1$ *then* # *else if* $t=n+k-m$ *then* $y(1)$ *else* *Lema 4.1*

$fby(\lambda t.\#)^m\ interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(t-n-k+m) =$

$= \lambda t.$ *if* $t \le n+k-m-1$ *then* # *else if* $t=n+k-m$ *then* $y(1)$ *else* *def. de interleave*

if $t \le n+k$ *then* # *else* $interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(t-n-k) =$

$= \lambda t.$ *if* $t \le n+k-m-1$ *then* # *else if* $t=n+k-m$ *then* $y(1)$ *else*

if $t \le n+k$ *then* # *else* $(\#, \overset{k-m-1}{\ldots}, \#, x(t-n-k), \#, \overset{m}{\ldots}, \#) \downarrow ((t-n-k-1) \bmod k)+1$

(1.1) Calculo los valores de t en los que se indexa $x(t-n-k)$, que serán los instantes en los que la función valga distinto de #

$$((t-n-k-1) \bmod k)+1 = k-m \Leftrightarrow t = (\alpha+2)*k-m+n, \text{ con } \alpha \in \mathbf{N}$$

Substituyendo este valor de t en la expresión que se indexa, obtengo el valor de la función en dichos instantes temporales:

$$x(\ (\alpha+2)*k-m+n-n-k\ ) = x(\ (\alpha+1)*k-m\ )$$

(1.2) En resumen, la función descrita por el término izquierdo es:

$fby(\lambda t.\#)^n fby(\lambda t.\#)^{k-m-1} fby(y) fby(\lambda t.\#)^m interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(t) =$

$= \#$ si $t \le n+k-m-1$

$= y(1)$ si $t = n+k-m$

$= \#$ si $n+k-m+1 \le t \le n+k$

$= x((\alpha+1)*k-m)$ si $t = (\alpha+2)*k-m+n$, con $\alpha \in \mathbf{N}$

$= \#$ en otro caso

(2) Estudio el valor en el tiempo de la función descrita por la expresión derecha de la ecuación:

$fby(\lambda t.\#)^n\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x', \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ ) =$

$= \lambda t.$ *if* $t \le n$ *then* # *else* $interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x', \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ )(t-n) =$

$= \lambda t.$ *if* $t \le n$ *then* # *else* $(\#, \overset{k-m-1}{\ldots}, \#, x'(t-n), \#, \overset{m}{\ldots}, \#) \downarrow ((t-n-1) \bmod k)+1$

(2.1) Calculo los valores de t en los que se indexa $x'(t\text{-}n)$, que serán los instantes en los que la función valga distinto de #

$$((t\text{-}n\text{-}1) \bmod k)+1 = k\text{-}m \Leftrightarrow t = (\alpha+1)*k+n\text{-}m, \text{ con } \alpha \in \mathbf{N}$$

(2.2) Reducción de la expresión $x'(t\text{-}n)$ en $t = (\alpha+1)*k+n\text{-}m$, con $\alpha \in \mathbf{N}$

$fby(\ \lambda t.\#\ )^{k\text{-}m\text{-}1}\ fby(\ y\ )\ fby(\ \lambda t.\#\ )^{m}$ *Lema 4.1, conversiones β*

$interleave(\ \lambda t.\#, \overset{k\text{-}m\text{-}1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ )((\alpha+1)*k\text{-}m) =$

$=$ *if* $\alpha*k+1 \leq 0$ *then* # *else* *para ningún α se cumple la condición, def. fby*

$fby(y)\ fby(\lambda t.\#)^{m}\ interleave(\lambda t.\#, \overset{k\text{-}m\text{-}1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(\alpha*k+1) =$

$=$ *if* $\alpha*k+1=1$ *then* $y(1)$ *else* *Lema 4.1, conversiones β*

$fby(\lambda t.\#)^{m}\ interleave(\lambda t.\#, \overset{k\text{-}m\text{-}1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(\alpha*k) =$

$=$ *if* $\alpha=0$ *then* $y(1)$ *else* *si m<k, para ningún α se cumple la segunda condición*

if $\alpha*k \leq m$ *then* # *else* $interleave(\lambda t.\#, \overset{k\text{-}m\text{-}1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#)(\alpha*k\text{-}m) =$

$=$ *if* $\alpha=0$ *then* $y(1)$ *else* *si* $\alpha>0 \Rightarrow ((\alpha*k\text{-}m\text{-}1) \bmod k\ )+1 = k\text{-}m$

$(\ \#, \overset{k\text{-}m\text{-}1}{\ldots}, \#, x(\alpha*k\text{-}m), \#, \overset{m}{\ldots}, \#\ )\downarrow((\alpha*k\text{-}m\text{-}1) \bmod k\ )+1 =$

$=$ *if* $\alpha=0$ *then* $y(1)$ *else* $x(\alpha*k\text{-}m)$

(2.3) En resumen, la función descrita por la expresión derecha es:

$fby(\lambda t.\#)^{n}\ interleave(\ \lambda t.\#, \overset{k\text{-}m\text{-}1}{\ldots}, \lambda t.\#, x', \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ ) =$

$= \#$ si $t \leq n$

$= y(1)$ si $t = n+k\text{-}m$ *α=0*

$= x(\ (\alpha+1)*k\text{-}m\ )$ si $t = (\alpha+2)*k\text{-}m+n$, con $\alpha \in \mathbf{N}$ *con cambio de variable*

$= \#$ en otro caso

□

MEMT: teorema de memorización intra-iniciaciones.

La idea que formaliza este teorema es muy simple: si un valor se calcula en cierto ciclo $k\text{-}m$, pero debe retrasarse n+1 unidades de tiempo para ser utilizado, es porque se necesita n ciclos más tarde del ciclo en que se calcula. Así, este teorema establece la compatibilidad entre el array de n+1 retardadores y un único retardador realimentado.

En un primer caso para cálculos que se realizan y se consumen en la misma iniciación del algoritmo:

$$\forall x \in Lu(A)_S,\ \forall k \in \mathbf{N}_+,\ \forall m \in \mathbf{N} \mid m<k \wedge n \le m,$$
$$fby(\ y\ )\ fby(\ \lambda t.\#\ )^n\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ ) \approx$$
$$fix(\ \lambda z.\ fby(\ y\ )\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, z, \overset{n}{\ldots}, z, \lambda t.\#, \overset{m-n}{\ldots}, \lambda t.\#\ )\ )$$

DEMOSTRACIÓN. Obtiene una definición por partes de las funciones que describen cada una de las expresiones que forman la ecuación y, comparando dichas definiciones, puede concluirse la compatibilidad de las mismas.

(1) Estudiaré el valor en el tiempo de la función descrita por la expresión izquierda de la ecuación:

$fby(y)\ fby(\ \lambda t.\#\ )^n\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ )(t) =$

definición de fby, conversiones β

$= if\ t=1\ then\ y(1)\ else$ *Lema 4.1, conversiones β*

$fby(\ \lambda t.\#\ )^n\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ )(t-1) =$

$= if\ t=1\ then\ y(1)\ else$ *definición de interleave, conversiones β*

$if\ t-1 \le n\ then\ \#\ else\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ )(t-1-n) =$

$= if\ t=1\ then\ y(1)\ else$

$if\ t \le n+1\ then\ \#\ else\ (\ \#, \overset{k-m-1}{\ldots}, \#, x(t-n-1), \#, \overset{m}{\ldots}, \#\ )\downarrow((t-n-2)\ mod\ k)+1)$

Como puede observarse, el resultado es una k-tupla anidada dentro de 2 expresiones *if*. Dicha tupla sólo es evaluada cuando $t > n+1$ y, en ese caso, vale distinto de # sólo cuando se indexa la componente $k-m$.

(1.2) Calculo los valores de t que hacen que se seleccione la expresión $x(t-n-1)$, es decir, los que satisfacen la ecuación $((t-n-2)\ mod\ k)+1 = k-m$:

$$((t-n-2)\ mod\ k)+1 = k-m \Leftrightarrow t = (\alpha+1)*k-m+n+1,\ \alpha \in \mathbf{N}$$

Substituyendo este valor de t en la expresión que se indexa podrán conocerse los valores que toma la función en esos instantes temporales:

$$x(\ (\alpha+1)*k-m+n+1-n-1) = x((\alpha+1)*k-m),\ \text{con}\ \alpha \in \mathbf{N}$$

(1.3) En resumen, la función descrita por la expresión izquierda de la ecuación se define por partes como sigue:

$fby(\ y\ )\ fby(\ \lambda t.\#\ )^n\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, \lambda t.\#, \overset{m}{\ldots}, \lambda t.\#\ )(t) =$

$= y(1)$ si $t = 1$

$= \#$ si $1 < t \leq n+1$

$= x((\alpha+1)*k-m)$ si $t = (\alpha+1)*k-m+n+1$, con $\alpha \in \mathbf{N}$

$= \#$ en otro caso

(2) Estudiaré ahora, también por casos, el valor en el tiempo de la función descrita por la expresión derecha de la ecuación. Para simplificar las expresiones, llamaré Z al subtérmino que toma como argumento el operador punto fijo:

$fix(\ \lambda z.fby(\ y\ )\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, z, \overset{n}{\ldots}, z, \lambda t.\#, \overset{m-n}{\ldots}, \lambda t.\#\ )\ )(t) =$

aplicando la propiedad fix(x) = x(fix(x)), conversión β

$= fby(\ y\ )\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, fix(\ Z\ ), \overset{n}{\ldots}, fix(\ Z\ ), \lambda t.\#, \overset{m-n}{\ldots}, \lambda t.\#\ )(t) =$

definición de fby, conversiones β

$= \lambda t.if\ t=1\ then\ y(1)\ else$ *definicion de interleave, conversiones β*

$interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, fix(\ Z\ ), \overset{n}{\ldots}, fix(\ Z\ ), \lambda t.\#, \overset{m-n}{\ldots}, \lambda t.\#\ )(t-1) =$

$= \lambda t.if\ t=1\ then\ y(1)\ else$

$(\ \#, \overset{k-m-1}{\ldots}, \#, x(t-1), fix(\ Z\ )(t-1), \overset{n}{\ldots}, fix(\ Z\ )(t-1), \#, \overset{m-n}{\ldots}, \#\ )\downarrow((t-2)\ mod\ k)+1$

Ahora el resultado es una k-tupla anidada dentro de 1 expresión *if*. Dicha tupla sólo se evalúa cuando $t > 1$ y, en ese caso, vale distinto de # sólo cuando se indexan las componentes comprendidas entre $k-m$ y $k-m+n$.

(2.1) En caso que $t > 1$ y $((t-2)\ mod\ k)+1 = k-m$ se indexa la componente $k-m$ de la k-tupla. Calculo los valores de t que hacen cumplir dicha ecuación:

$$((t-2)\ mod\ k)+1 = k-m \Leftrightarrow t = (\alpha+1)*k-m+1, \text{ con } \alpha \in \mathbf{N}$$

Substituyendo la t de la expresión $x(t-1)$ por la t calculada podrán conocerse los valores que toma la función en esos instantes temporales:

$$x(t-n-1) = x((\alpha+1)*k-m+1-1) = x((\alpha+1)*k-m), \text{ con } \alpha \in \mathbf{N}$$

(2.2) En caso que $t > 1$ y $k\text{-}m+1 \le ((t\text{-}2) \bmod k)+1 \le k\text{-}m+n$ se indexan las componentes comprendidas entre las posiciones $k\text{-}m+1$ y $m\text{-}n\text{-}1$. Calculo los valores de t que hacen cumplir la anterior inecuación:

$$k\text{-}m+1 = ((t\text{-}2) \bmod k)+1 \Leftrightarrow t = (\alpha+1)*k\text{-}m+2,\ \alpha \in \mathbb{N}$$

$$k\text{-}m+n = ((t\text{-}2) \bmod k)+1 \Leftrightarrow t = (\alpha+1)*k\text{-}m+n+1,\ \alpha \in \mathbb{N}$$

de modo que:

$$t > 1 \wedge k\text{-}m+1 \le ((t\text{-}2) \bmod k)+1 \le k\text{-}m+n \Leftrightarrow$$

$$\Leftrightarrow (\alpha+1)*k\text{-}m+2 \le t \le (\alpha+1)*k\text{-}m+n+1, \text{ con } \alpha \in \mathbb{N}$$

En este intervalo la función vale $fix(\ Z\ )(t\text{-}1)$ con t cumpliendo la anterior inecuación. Sin embargo, para poder comparar esta función con la definida en (1.3), es necesario poner este valor en función de x. Así que demostraré por inducción, algo que la intuición nos dice: que la función vale igual dentro de cada uno de los intervalos temporales definidos por α y este valor es:

$$x(\ (\alpha+1)*k\text{-}m\ )$$

(2.2.1) Caso base, $t = (\alpha+1)*k\text{-}m+2$:

$fix(\ Z\ )(t\text{-}1) =$

$= fix(\ Z\ )((\alpha+1)*k\text{-}m+1) =$ *aplicando la propiedad fix(x) = x(fix(x)), conversiones β*

$= fby(y)interleave(\lambda t.\#, \overset{k\text{-}m\text{-}1}{\ldots}, \lambda t.\#, x, fix(Z), \overset{n}{\ldots}, fix(Z), \lambda t.\#, \overset{m\text{-}n}{\ldots}, \lambda t.\#)((\alpha+1)*k\text{-}m+1) =$

definición de fby, conversiones β

$= if\ (\alpha+1)*k\text{-}m=0\ then\ y(1)\ else$ *t>1 y para ningún α se cumple la segunda condición*

$interleave(\lambda t.\#, \overset{k\text{-}m\text{-}1}{\ldots}, \lambda t.\#, x, fix(Z), \overset{n}{\ldots}, fix(Z), \lambda t.\#, \overset{m\text{-}n}{\ldots}, \lambda t.\#)((\alpha+1)*k\text{-}m) =$

$= interleave(\ \lambda t.\#, \overset{k\text{-}m\text{-}1}{\ldots}, \lambda t.\#, x, fix(\ Z\ ), \overset{n}{\ldots}, fix(\ Z\ ), \lambda t.\#, \overset{m\text{-}n}{\ldots}, \lambda t.\#\ )((\alpha+1)*k\text{-}m) =$

definición de interleave, conversiones β

$= (\#, \overset{k\text{-}m\text{-}1}{\ldots}, \#, x((\alpha+1)*k\text{-}m), fix(Z)((\alpha+1)*k\text{-}m), \overset{n}{\ldots}, fix(Z)((\alpha+1)*k\text{-}m), \#, \overset{m\text{-}n}{\ldots}, \#)$

$\downarrow(((\alpha+1)*k\text{-}m\text{-}1) \bmod k)+1 =$

$= (\#, \overset{k\text{-}m\text{-}1}{\ldots}, \#, x((\alpha+1)*k\text{-}m), fix(Z)((\alpha+1)*k\text{-}m), \overset{n}{\ldots}, fix(Z)((\alpha+1)*k\text{-}m), \#, \overset{m\text{-}n}{\ldots}, \#)\downarrow(k\text{-}m) =$

$= x((\alpha+1)*k\text{-}m)$

(2.2.2) Paso de inducción $t = (\alpha+1)*k\text{-}m+2+j+1$ (conociendo que estará dentro del intervalo)

$fix(\ Z\ )(t\text{-}1) =$

$= fix(\ Z\ )((\alpha+1)*k\text{-}m+2+j) =$ *aplicando la propiedad fix(x) = x(fix(x)), conversiones β*

$= fby(y)\ interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, fix(Z), \overset{n}{\ldots}, fix(Z), \lambda t.\#, \overset{m-n}{\ldots}, \lambda t.\#)((\alpha+1)*k\text{-}m+2+j) =$

definición de fby, conversiones β

$=$ *if* $(\alpha+1)*k\text{-}m+1+j=0$ *then* $y(1)$ *else* *t>1 y para ningún α se cumple la condición*

$interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, fix(Z), \overset{n}{\ldots}, fix(Z), \lambda t.\#, \overset{m-n}{\ldots}, \lambda t.\#)((\alpha+1)*k\text{-}m+1+j) =$

$= interleave(\lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, fix(Z), \overset{n}{\ldots}, fix(Z), \lambda t.\#, \overset{m-n}{\ldots}, \lambda t.\#)\ ((\alpha+1)*k\text{-}m+1+j) =$

definición de interleave, conversiones β

$= (\ \#, \overset{k-m-1}{\ldots}, \#, x((\alpha+1)*k\text{-}m+1+j), fix(Z)((\alpha+1)*k\text{-}m+1+j), \overset{n}{\ldots},$

$fix(Z)((\alpha+1)*k\text{-}m+1+j), \#, \overset{m-n}{\ldots}, \#\ )\downarrow(((\alpha+1)*k\text{-}m+j)\ mod\ k)+1 =$

$= (\ \#, \overset{k-m-1}{\ldots}, \#, x((\alpha+1)*k\text{-}m+1+j), fix(Z)((\alpha+1)*k\text{-}m+1+j), \overset{n}{\ldots},$

$fix(Z)((\alpha+1)*k\text{-}m+1+j), \#, \overset{m-n}{\ldots}, \#\ )\downarrow(k\text{-}m+j+1) =$

$= fix(\ Z\ )((\alpha+1)*k\text{-}m+1+j) =$

$= fix(\ Z\ )(t\text{-}2) =$ *hipótesis de inducción*

$= x((\alpha+1)*k\text{-}m)$

(2.3) En resumen, la función descrita por la expresión derecha de la ecuación se define por partes como sigue:

$fix(\ \lambda z.fby(\ y\ )(\ interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, \lambda t.\#, x, z, \overset{n}{\ldots}, z, \lambda t.\#, \overset{m-n}{\ldots}, \lambda t.\#\ )\ )\ )(t) =$

$= y(1)$ si $t = 1$

$= x((\alpha+1)*k\text{-}m)$ si $t = (\alpha+1)*k\text{-}m+1$, con $\alpha \in \mathbf{N}$

$= x((\alpha+1)*k\text{-}m)$ si $(\alpha+1)*k\text{-}m+2 \leq t \leq (\alpha+1)*k\text{-}m+n+1$, con $\alpha \in \mathbf{N}$

$= \#$ en otro caso

(3) Para finalizar falta por demostrar que ambas funciones son compatibles. Para ello, basta con comprobar visualmente que lo son en aquellos instantes en los que ambas son distintas de #.

□

MEMT2: teorema de memorización inter-iniciaciones.

Es un segundo caso del teorema anterior, formaliza que sucede con aquellos cálculos que se realizan en una iniciación pero se consumen en

alguna posterior (es decir, aquellos cálculos que provienen de especificaciones ecuacionales recursivas o iterativas).

$$\forall x \in Lu(A)_S,\ \forall k \in \mathbf{N}_+,\ \forall m \in \mathbf{N} \mid m<k \wedge n<k \wedge n > m,$$
$$fby(\ y\ )\ fby(\ \lambda t.\#\ )^n\ interleave(\ \lambda t.\#,\ \overset{k-m-1}{\ldots},\ \lambda t.\#,\ x,\ \lambda t.\#,\ \overset{m}{\ldots},\ \lambda t.\#\ ) \approx$$
$$fix(\ \lambda z.\ fby(\ y\ )\ interleave(\ z,\ \overset{n-m}{\ldots},\ z,\ \lambda t.\#,\ \overset{k-n-1}{\ldots},\ \lambda t.\#,\ x,\ z,\ \overset{m}{\ldots},\ z\ )\ )$$

DEMOSTRACIÓN. Sigue el esquema anterior: obtener una definición por partes de las funciones descritas por ambos términos de la ecuación y compararlas para concluir su compatibilidad.

(1) El valor en el tiempo de la función denotada por el término izquierdo de la ecuación, ya estudiado en la anterior demostración, es:

$fby(\ y\ )\ fby(\ \lambda t.\#\ )^n\ interleave(\ \lambda t.\#,\ \overset{k-m-1}{\ldots},\ \lambda t.\#,\ x,\ \lambda t.\#,\ \overset{m}{\ldots},\ \lambda t.\#\ )(t) =$

$= y(1)$ si $t = 1$

$= \#$ si $1 < t \leq n+1$

$= x((\alpha+1)*k-m)$ si $t = (\alpha+1)*k-m+n+1$, con $\alpha \in \mathbf{N}$

$= \#$ en otro caso

(2) Estudiaré, del mismo modo que en la anterior demostración, el valor en el tiempo de la función descrita por la expresión derecha de la ecuación.

$fix(\ \lambda z.\ fby(\ y\ )\ interleave(\ z,\ \overset{n-m}{\ldots},\ z,\ \lambda t.\#,\ \overset{k-n-1}{\ldots},\ \lambda t.\#,\ x,\ z,\ \overset{m}{\ldots},\ z\ )\ ) =$

aplicando la propiedad fix(x) = x(fix(x)), conversión β

$= fby(\ y\ )\ interleave(\ fix(Z),\ \overset{n-m}{\ldots},\ fix(Z),\ \lambda t.\#,\ \overset{k-n-1}{\ldots},\ \lambda t.\#,\ x,\ fix(Z),\ \overset{m}{\ldots},\ fix(Z)\ )(t) =$

definición de fby, conversiones β

$= \lambda t.if\ t=1\ then\ y(1)\ else$ *definición de interleave, conversiones β*

$interleave(\ fix(Z),\ \overset{n-m}{\ldots},\ fix(Z),\ \lambda t.\#,\ \overset{k-n-1}{\ldots},\ \lambda t.\#,\ x,\ fix(Z),\ \overset{m}{\ldots},\ fix(Z)\ )(t-1) =$

$= \lambda t.if\ t=1\ then\ y(1)\ else$

$(fix(Z)(t-1),\overset{n-m}{\ldots},fix(Z)(t-1),\#,\overset{k-n-1}{\ldots},\#,x(t-1),fix(Z)(t-1),\overset{m}{\ldots},fix(Z)(t-1))\downarrow((t-2) mod k)+1$

Como puede verse, el resultado es una k-tupla anidada dentro de una expresión *if*. Dicha tupla sólo se evalúa cuando $t > 1$ y, en ese caso, vale #

cuando se indexan las componentes comprendidas entre n-m+1 y k-m-1. Calcularé el valor que toma el resto de las componentes.

(2.1) En caso que $t > 1$ y $((t\text{-}2) \ mod \ k)+1 = k\text{-}m$ se indexa la componente k-m de la k-tupla. Calculo los valores de t que hacen cumplir dicha ecuación:

$$((t\text{-}2) \ mod \ k)+1 = k\text{-}m \Leftrightarrow t = (\alpha+1)*k\text{-}m+1, \text{ con } \alpha \in \mathbf{N}$$

Substituyendo la t de la expresión $x(t\text{-}1)$ por la t calculada podrán conocerse los valores que toma la función en esos instantes temporales:

$$x(t\text{-}n\text{-}1) = x((\alpha+1)*k\text{-}m+1\text{-}1) = x((\alpha+1)*k\text{-}m), \text{ con } \alpha \in \mathbf{N}$$

(2.2) En caso que $t > 1$ y $k\text{-}m+1 \leq ((t\text{-}2) \ mod \ k)+1 \leq k$ se indexan las últimas m componentes. Calculo los valores de t que hacen cumplir dicha inecuación:

$$k\text{-}m+1 = ((t\text{-}2) \ mod \ k)+1 \Leftrightarrow t = (\alpha+1)*k\text{-}m+2, \text{ con } \alpha \in \mathbf{N}$$

$$k = ((t\text{-}2) \ mod \ k)+1 \Leftrightarrow t = (\alpha+1)*k+1, \text{ con } \alpha \in \mathbf{N}$$

de modo que:

$$t > 1 \wedge k\text{-}m+1 \leq ((t\text{-}2) \ mod \ k)+1 \leq k \Leftrightarrow$$

$$\Leftrightarrow (\alpha+1)*k\text{-}m+2 \leq t \leq (\alpha+1)*k+1, \text{ con } \alpha \in \mathbf{N}$$

En este intervalo la función vale $fix(\ Z\ )(t\text{-}1)$ con t cumpliendo la anterior inecuación que, utilizando el resultado obtenido en el apartado (2.2) de la anterior demostración con $n = m$, resulta ser:

$$x((\alpha+1)*k\text{-}m)$$

(2.3) En caso que $t > 1$ y $1 \leq ((t\text{-}2) \ mod \ k)+1 \leq n\text{-}m$ se indexan las primeras n-m componentes. Calculo los valores de t que hacen cumplir dicha inecuación:

$$1 = ((t\text{-}2) \ mod \ k)+1 \Leftrightarrow t = \alpha*k+2, \text{ con } \alpha \in \mathbf{N}$$

$$n\text{-}m = ((t\text{-}2) \ mod \ k)+1 \Leftrightarrow t = \alpha*k+n\text{-}m+1, \text{ con } \alpha \in \mathbf{N}$$

de modo que:

$$t > 1 \wedge 1 \leq ((t\text{-}2) \ mod \ k)+1 < k\text{-}m \Leftrightarrow \alpha*k+2 \leq t \leq \alpha*k+n\text{-}m+1, \text{ con } \alpha \in \mathbf{N}$$

En este intervalo la función vuelve a valer $fix(\ Z\ )(t\text{-}1)$ con t cumpliendo la anterior inecuación. Calcularé este valor en función de x demostrando por inducción algo que la intuición nos dice: que la función vale igual dentro de cada uno de los intervalos temporales definidos por α y este valor es $x(\alpha*k\text{-}m)$

cuando $\alpha>1$ e $y(1)$ si $\alpha=0$, es decir, el valor almacenado en la anterior iniciación.

(2.3.1) Caso base $t = \alpha*k+2$

$fix(\ Z\ )(t\text{-}1) =$

$= fix(\ Z\ )(\alpha*k+1) =$ *aplicando la propiedad fix(x) = x(fix(x)), conversiones β*

$= fby(y)\ interleave(fix(Z),\overset{n-m}{\ldots},fix(Z),\lambda t.\#,\overset{k-n-1}{\ldots},\lambda t.\#,x,fix(Z),\overset{m}{\ldots},fix(Z))(\alpha*k+1) =$

definición de fby, conversiones β

$= \lambda t.if\ \alpha*k=0\ then\ y(1)\ else$ *si α=0 demostrado, si α≠0 es necesario evaluar interleave*

$interleave(\ fix(\ Z\ ),\overset{n-m}{\ldots},fix(\ Z\ ),\lambda t.\#,\overset{k-n-1}{\ldots},\lambda t.\#,\ x,\ fix(\ Z\ ),\overset{m}{\ldots},fix(\ Z\ )\ )(\alpha*k) =$

$= interleave(\ fix(\ Z\ ),\overset{n-m}{\ldots},fix(\ Z\ ),\lambda t.\#,\overset{k-n-1}{\ldots},\lambda t.\#,\ x,\ fix(\ Z\ ),\overset{m}{\ldots},fix(\ Z\ )\ )(\alpha*k) =$

definición de interleave, conversión β

$= (\ fix(Z)(\alpha*k),\overset{n-m}{\ldots},fix(Z)(\alpha*k),\#,\overset{k-n-1}{\ldots},\#,x(\alpha*k),fix(Z)(\alpha*k),\overset{m}{\ldots},fix(Z)(\alpha*k)\ )$

$\downarrow((\alpha*k\text{-}1)\ mod\ k)+1 =$

$= (fix(Z)(\alpha*k),\overset{n-m}{\ldots},fix(Z)(\alpha*k),\#,\overset{k-n-1}{\ldots},\#,x(\alpha*k),fix(Z)(\alpha*k),\overset{m}{\ldots},fix(Z)(\alpha*k))\downarrow k =$

$= fix(\ Z\ )(\alpha*k) =$

$= x(\alpha*k\text{-}m)$

(2.3.2) Paso de inducción $t = \alpha*k+2+j+1$ (conociendo que estará dentro del intervalo)

$fix(\ Z\ )(t\text{-}1) =$

$= fix(\ Z\ )(\alpha*k+2+j) =$ *aplicando la propiedad fix(x) = x(fix(x)), conversiones β*

$= fby(y)\ interleave(fix(Z),\overset{n-m}{\ldots},fix(Z),\lambda t.\#,\overset{k-n-1}{\ldots},\lambda t.\#,x,fix(Z),\overset{m}{\ldots},fix(Z))(\alpha*k+2+j) =$

definición de fby, conversiones β

$= \lambda t.if\ \alpha*k+1+j=0\ then\ y(1)\ else$ *para ningún α se cumple la condición*

$interleave(\ fix(Z),\overset{n-m}{\ldots},fix(Z),\lambda t.\#,\overset{k-n-1}{\ldots},\lambda t.\#,\ x,\ fix(Z),\overset{m}{\ldots},fix(Z)\ )(\alpha*k+1+j) =$

$= interleave(\ fix(Z),\overset{n-m}{\ldots},fix(Z),\lambda t.\#,\overset{k-n-1}{\ldots},\lambda t.\#,\ x,\ fix(Z),\overset{m}{\ldots},fix(Z)\ )(\alpha*k+1+j) =$

definición de interleave, conversión β

$= (\ fix(Z)(\alpha*k+1+j),\overset{n-m}{\ldots},fix(Z)(\alpha*k+1+j),\#,\overset{k-n-1}{\ldots},\#,x(\alpha*k+1+j),$

$fix(Z)(\alpha*k+1+j),\overset{m}{\ldots},fix(Z)(\alpha*k+1+j)\ )\downarrow((\alpha*k+j)\ mod\ k)+1 =$

$= fix(\ Z\ )(\alpha*k+1+j) =$

$= fix(\ Z\ )(t\text{-}2) =$ *hipótesis de inducción*

$= x(\alpha * k\text{-}m)$

(2.4) En resumen, la función definida por la expresión derecha de la ecuación se define por partes como sigue:

$fix(\ \lambda z.fby(\ y\ )\ interleave(\ z, \overset{n-m}{\ldots}, z, \lambda t.\#, \overset{k-n-1}{\ldots}, \lambda t.\#, x, z, \overset{m}{\ldots}, z\ )\ )(t) =$

$= y(1)$ si $1 \le t \le n\text{-}m+1$

$= x((\alpha+1)*k\text{-}m)$ si $t = (\alpha+1)*k\text{-}m+1$, con $\alpha \in \mathbf{N}$

$= x((\alpha+1)*k\text{-}m)$ si $(\alpha+1)*k\text{-}m+2 \le t \le (\alpha+1)*k+1$, con $\alpha \in \mathbf{N}$

$= x((\alpha+1)*k\text{-}m)$ si $(\alpha+1)*k+2 \le t \le (\alpha+1)*k+n\text{-}m+1$, con $\alpha \in \mathbf{N}$

$= \#$ en otro caso

(3) Para finalizar falta por demostrar que ambas funciones son compatibles. Para ello, basta con comprobar visualmente que lo son en aquellos instantes en los que ambas son distintas de #.

□

DET: teorema de descomposición de acciones.

Cuando se planifica una operación a un ciclo utilizando el teorema TMT, no sólo se fija el momento en que se realiza la operación sino que indirectamente también se fija el instante en que se realiza la lectura de las fuentes, la escritura sobre el destino y la reserva de los caminos de comunicación. Este teorema permite separar las planificaciones de las distintas acciones RT anteriormente referidas para que, en una posterior fase, puedan reusarse separadamente los distintos tipos de recursos que las implementan.

$$\forall x_i \in Lu(A)_s,\ \forall k \in \mathbf{N}_+,\ \forall m \in \mathbf{N} \mid m<k,\ interleave(\ x_1, \overset{k-m-1}{\ldots}, x_{k-m}, \overset{m}{\ldots}, x_k\ ) =$$
$$= interleave(\ x_1, \overset{k-m-1}{\ldots}, interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, x_{k-m}, \overset{m}{\ldots}, \lambda t.\#\ ), \overset{m}{\ldots}, x_k\ )$$

DEMOSTRACIÓN. Dado que todos los argumentos del operador *interleave*, excepto el ubicado en la posición *k-m*, coinciden en uno y otro lado de la

ecuación, bastará con demostrar que las expresiones representan a la misma función en aquellos valores de t en los que se indexa dicha componente:
$((t\text{-}1)\ mod\ k)+1 = k\text{-}m \Leftrightarrow \alpha*k+k\text{-}m\text{-}1 = t\text{-}1 \Leftrightarrow t = (\alpha+1)*k\text{-}m$, con $\alpha \in \mathbb{N}$

Así que basta con demostrar que:

$$x_{k-m}((\alpha+1)*k\text{-}m) = interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, x_{k-m}, \overset{m}{\ldots}, \lambda t.\#\ )((\alpha+1)*k\text{-}m)$$

Lo haré por reducción de la expresión derecha a la izquierda

$interleave(\ \lambda t.\#, \overset{k-m-1}{\ldots}, x_{k-m}, \overset{m}{\ldots}, \lambda t.\#\ )((\alpha+1)*k\text{-}m) =$

definición de interleave, conversiones β

$= (\ \#, \overset{k-m-1}{\ldots}, x_{k-m}((\alpha+1)*k\text{-}m), \overset{m}{\ldots}, \#\ )\downarrow(((\alpha+1)*k\text{-}m\text{-}1)\ mod\ k)+1 =$

*a mod b = (a + n*b) mod b*

$= (\ \#, \overset{k-m-1}{\ldots}, x_{k-m}((\alpha+1)*k\text{-}m), \overset{m}{\ldots}, \#\ )\downarrow(k\text{-}m) =$ *semántica de ↓*

$= x_{k-m}((\alpha+1)*k\text{-}m)$

□

INAT: teorema de anticipación de entradas.

Este teorema establece qué sucede si pretendemos anticipar los valores de una señal 'lenta', cuya lectura ha sido planificada en cierto ciclo. Desde el punto de vista hardware este teorema presupone que es posible la implementación monociclo de la especificación original, y por tanto, se puede asumir que las entradas están estables durante todo el ciclo de muestreo†.

$$\forall x, x_i \in Lu(A)_s,\ \forall k \in \mathbb{N}_+,\ \forall m \in \mathbb{N} \mid m<k,$$
$$interleave(\ x_1, \overset{k-m-1}{\ldots}, x_{k-m-1}, next^m(\ replicate(\ x,\ k\ )\ ), x_{k-m+1}, \overset{m}{\ldots}, x_k\ ) =$$
$$= interleave(\ x_1, \overset{k-m-1}{\ldots}, x_{k-m-1}, replicate(\ x,\ k\ ), x_{k-m+1}, \overset{m}{\ldots}, x_k\ )$$

DEMOSTRACIÓN. Como en la demostración anterior, los argumentos de los operadores *interleave* presentes en las expresiones izquierda y derecha de

† Si no lo fueran, bastaría con añadir registros externos.

la ecuación son iguales excepto el ubicado en la posición *k-m*. Por ello simplemente demostraré, por reducción de la expresiones derecha e izquierda a una forma común, que:

$next^m(\ replicate(\ x,\ k\ )\ )((k\text{-}m)+\alpha*k) = replicate(\ x,\ k\ )((k\text{-}m)+\alpha*k)$, con $\alpha \in \mathbf{N}$

(1) Reducción de la expresión izquierda:

$next^m(\ replicate(\ x,\ k\ )\ )(\ (k\text{-}m)+\alpha*k\ ) =$ *Lema 4.2, conversiones β*

$= replicate(\ x,\ k\ )(\ (\alpha+1)*k\ ) =$ *definición de replicate, conversiones β*

$= x(\ ceil(\ ((\alpha+1)*k)/k\ )\ ) =$

$= x(\ ceil(\ \alpha+1\ )\ ) =$ $\alpha+1 \in \mathbf{N}$

$= x(\alpha+1)$

(2) Reducción de la expresión derecha:

$replicate(\ x,\ k\ )(\ (k\text{-}m)+\alpha*k\ ) =$ *definición de replicate, conversiones β*

$= x(\ ceil(\ (\alpha+1)\text{-}(m/k)\ )\ ) =$ $m<k \Rightarrow m/k<1,\ \forall n \in \mathbf{N}_+ \wedge \forall \varepsilon \in \mathbf{R} \mid 0 \leq \varepsilon < 1,\ ceil(\ n-\varepsilon\ ) = n$

$= x(\ ceil(\ \alpha+1\ )\ ) =$ $\alpha+1 \in \mathbf{N}$

$= x(\alpha+1)$

□

4.2.5 Teoremas de proyección RT.

Hasta el momento todos los operadores presentados pueden considerarse abstractos (no poseen señales de control y los tipos que manipulan no son aún vectores de bits), por lo que si deseamos obtener una especificación ecuacional que refleje más fielmente un circuito hardware a nivel RT, es necesario reemplazarlos por operadores con una mayor afinidad a los módulos hardware. A la cuestión de cómo proyectar operadores abstractos no-temporales sobre módulos hardware combinacionales dedicaré el capítulo 6 de esta memoria. Esta sección se dedica a presentar simplemente cómo proyectar los operadores temporales.

De los cinco presentados, el operador *next* es no implementable y no podrá, por tanto, proyectarse sobre ningún módulo hardware real (esto no es preocupante ya las especificaciones ecuacionales obtenidas tras todo proceso de síntesis correcto no lo utilizan). Los operadores *sample* y *replicate* simplemente indican relaciones entre las frecuencias de distintos sucesos, por lo que para implementarlos no es necesario ningún módulo especifico sino una correcta conexión de los elementos secuenciales a los relojes adecuados. En cuanto al operador *fby*, la relación con un retardador hardware† es inmediata por lo que no añadiré ningún nuevo símbolo††. Finalmente sólo queda el operador *interleave*.

El comportamiento de un operador *interleave* de k entradas permite conocer cuál es su correspondencia hardware: un multiplexor que, en el caso más desfavorable, tendrá k entradas de datos y una línea de control que sigue cierto patrón regular que se repite cada k ciclos. Esta correspondencia asume que la selección de fuentes en un circuito sintetizado por técnicas de SAN se implementa usando multiplexores. Otro tipo de técnicas de selección (por ejemplo, vía buses) también podrían ser formalizables. En cualquier caso, asumiré la existencia de una biblioteca (léase signatura para el formalismo presentado) que posea las descripciones de un variado número de multiplexores con distintas anchuras y número de puertos y que el comportamiento de cualquiera de ellos puede describirse por la λ-expresión:

$$mux = (\lambda(sel,x_0,\overset{n}{\ldots},x_{n-1}).(\lambda t.((x_0(t) \overset{n}{\ldots} x_{n-1}(t))\downarrow(sel(t) \bmod n))))$$

---

† El primer argumento es el valor inicial tras el reset, y el segundo argumento es el puerto de entrada de datos.

†† Podría, no obstante, pensarse que es deseable poder proyectar definiciones del tipo $x\ fby\ (y_1 \mid\mid \ldots \mid\mid y_n)$ sobre un registro. No lo he hecho porque desde el punto de vista hardware un registro es un par retardador-multiplexor 2 a 1, que requiere una línea de control y en cuyo puerto de entrada suele añadirse, tras un proceso de SAN, otro multiplexor que requiere a su vez una nueva línea de control. El resultado es que, en cualquier caso, una implementación que utilice sólo retardadores y multiplexores siempre es más barata que una implementación análoga que utilice registros y multiplexores.

MUXI: teorema de implementación con multiplexores.

Esta propiedad establece la equivalencia entre un operador *interleave* cuyos k argumentos son n fuentes de datos que pueden estar repetidas ($n \le k$), y un multiplexor n a 1, cuyas fuentes de datos no se repiten y cuyo control está generado por un operador *interleave* de k argumentos constantes que genera los patrones fijos de selección de fuentes.

$$\forall x_i \in Lu(A)_s,\ \forall k,i \in \mathbf{N}_+ \mid i \le k,$$

$$interleave(\ x_{j1}, \overset{k}{\ldots}, x_{jk}\ ) = mux(\ interleave(\ \lambda t.j1, \overset{k}{\ldots}, \lambda t.jk\ ),\ x_1, \overset{n}{\ldots}, x_n\ )$$

donde $n \le k$ y $(\ j1, \overset{k}{\ldots}, jk\ )$ es una variación con repetición del conjunto de índices $(\ 1, \overset{n}{\ldots}, n\ )$.

EJEMPLO 4.1

Esta propiedad permite establecer por ejemplo que *interleave*(*a*, *a*, *b*, *c*, *b*) es equivalente, entre otras, a cualquiera de las siguientes expresiones:

mux(*interleave*(0, 0, 1, 2, 1), *a*, *b*, *c*)

mux(*interleave*(1, 1, 0, 2, 0) *b*, *a*, *c*)

mux(*interleave*(2, 2, 1, 0, 1) *c*, *b*, *a*)

DEMOSTRACIÓN. Se realiza demostrando que en todo instante las expresiones pueden reducirse a una forma común.

(1) Reducción de la expresión izquierda para cierto instante $i+\alpha*k$, con $i,\alpha \in \mathbf{N} \wedge i \le k$:

$mux(\ interleave(\ \lambda t.j1, \overset{k}{\ldots}, \lambda t.jk\ ),\ x_1, \overset{n}{\ldots}, x_n\ )(i+\alpha*k) =$ *def. de mux, conv.* β

$= (\ x_1(i+\alpha*k) \overset{n}{\ldots} x_n(i+\alpha*k)\ )\downarrow(interleave(\ \lambda t.j1, \overset{k}{\ldots}, \lambda t.jk\ )(i+\alpha*k)\ mod\ n) =$

definición de interleave, conversión β

$= (\ x_1(i+\alpha*k) \overset{n}{\ldots} x_n(i+\alpha*k)\ )\downarrow(\ (\ (\ j1, \overset{k}{\ldots}, jk\ )\downarrow((i+\alpha*k-1)\ mod\ k)+1)\ )\ mod\ n) =$

$= (\ x_1(i+\alpha*k) \overset{n}{\ldots} x_n(i+\alpha*k)\ )\downarrow(\ (\ j1, \overset{k}{\ldots}, jk\ )\downarrow i\ ) =$ $i \le k$

$= (x_1(i+\alpha*k) \overset{n}{\ldots} x_n(i+\alpha*k))\downarrow ji =$ *por ser (j1, $\overset{k}{\ldots}$, jk) una variación de (1, $\overset{n}{\ldots}$, n)*

$= x_{ji}(i+\alpha*k)$

(2) Reducción de la expresión derecha para el mismo instante:

interleave$(x_{j1}, \overset{k}{\ldots}, x_{jk})(i+\alpha*k) =$ *definición de interleave, conversión β*

$= (x_{j1}(i+\alpha*k), \overset{k}{\ldots}, x_{jk}(i+\alpha*k))\downarrow((i+\alpha*k\text{-}1) \ mod \ k)+1 =$

$= (x_{j1}(i+\alpha*k), \overset{k}{\ldots}, x_{jk}(i+\alpha*k))\downarrow i =$

$= x_{ji}(i+\alpha*k)$

□

4.3 Incorporación de los operadores temporales al formalismo de especificación ecuacional.

Esta sección redefine dos conceptos introducidos en el capítulo 2, de manera que indirectamente se amplíe la sintaxis y la semántica del mecanismo de especificación ecuacional para que pueda utilizar los nuevos operadores temporales. Gracias a ello podrá describirse mediante una especificación ecuacional cualquier conducta en un estado intermedio de un proceso de SAN y podrá, igualmente, realizarse dicho proceso de síntesis por derivación formal.

4.3 DEFINICIÓN. (Reemplaza a la definición 2.24). Sea (S, Σ) una signatura heterogénea. Definimos $Lu(\Sigma)$ como la signatura $(S \cup \{N\}, \Sigma_N \cup \Sigma')$ tal que Σ_N es una signatura capaz de dar soporte sintáctico a los números naturales y Σ' es la familia $S^*\times S$ indexada de conjuntos que verifica para todo género s:

- $\Sigma'_{N,s} = \Sigma^{\#}_{N,s} \cup \{ >> \}$
- $\Sigma'_{s,N} = \Sigma^{\#}_{s,N} \cup \{ << \}$
- $\Sigma'_{s,s} = \Sigma^{\#}_{s,s} \cup \{ \text{next} \}$
- $\Sigma'_{s.s,s} = \Sigma^{\#}_{s.s,s} \cup \{ \text{fby}, (\square \,||\, \square) \}$
- $\Sigma'_{w,s} = \Sigma^{\#}_{w,s} \cup \{ (\square \,||\, \overset{n}{\ldots} \,||\, \square) \}$ con $w \in \{ s.s.s, s.s.s.s, \ldots \}$

Obsérvese como esta definición, al igual que lo estaba la que reemplaza, está completamente condicionada por la signatura Σ. Recuérdese que esta signatura Σ se facilita junto con la especificación ecuacional del circuito, por lo que el número y género de los operadores temporales varía de un proyecto de diseño a otro.

4.4 DEFINICIÓN. (Reemplaza a la definición 2.26) Sea $Lu(\Sigma)$ la signatura definida a partir de (S, Σ), y sea A una Σ-álgebra. Definimos $Lu(A)$ como la $Lu(\Sigma)$-álgebra que cumple:

- El universo de $Lu(A)$ es la familia $S\cup\{N\}$-indexada de conjuntos soporte $(\mathbf{N}_+\to(A^{\#})_\perp) \cup \mathbf{N}$.
- Cualquier símbolo de operación $\sigma \in \Sigma_N$ denota a una función natural de dominio(s) natural(es).
- Cualquier símbolo de operación $\sigma \in \Sigma_{w,s}$ que denota la función $A_\sigma^{w,s}$, permite definir la denotación del mismo símbolo como la función:

  $Lu(A)_\sigma^{w,s} = (\mathbf{N}_+\to(A^{\#})_\perp)_\sigma^{w,s}$

  o lo que es lo mismo[†]:

  $Lu(A)_\sigma^{w,s} = \lambda(x_1, ..., x_n) : Lu(A_{s1})\times...\times Lu(A_{sn}).$
  $(\lambda t : \mathbf{N}_+. ((A^{\#})_\perp)_\sigma^{w,s}(x_1(t), ..., x_n(t)) : (A_s^{\#})_\perp) : Lu(A_s)$
- Para todo género s, el símbolo >> denota a la función $Lu(A)_{>>}^{\mathbf{N},s,s}$, que se define como:

  $Lu(A)_{>>}^{\mathbf{N},s,s} = \lambda(n,x) : \mathbf{N}_+\times(\mathbf{N}_+\to(A_s^{\#})_\perp).$
  $(\lambda t : \mathbf{N}_+. x(t\dot{-}n) : (A_s^{\#})_\perp) : (\mathbf{N}_+\to(A_s^{\#})_\perp)$
- Para todo género s, el símbolo << denota a la función $Lu(A)_{<<}^{\mathbf{N},s,s}$, que se define como:

  $Lu(A)_{<<}^{\mathbf{N},s,s} = \lambda(x,n) : (\mathbf{N}_+\to(A_s^{\#})_\perp)\times\mathbf{N}_+.$
  $(\lambda t : \mathbf{N}_+. x(ceil(t/n)) : (A_s^{\#})_\perp) : (\mathbf{N}_+\to(A_s^{\#})_\perp)$
- Para todo género s, el símbolo *next* denota a la función $Lu(A)_{next}^{\varepsilon,s}$, que se define como:

[†] Véase la definición 2.22.

$$Lu(A)^{\varepsilon,s}_{\mathbf{next}} = \lambda x :(\mathbf{N}_+\rightarrow(A_s^{\#})_\perp).$$
$$(\ \lambda t :\mathbf{N}_+.\ x(t+1) :(A_s^{\#})_\perp\ ) :(\mathbf{N}_+\rightarrow(A_s^{\#})_\perp)$$

- Para todo género *s*, el símbolo *fby* denota a la función $Lu(A)^{s.s,s}_{\mathbf{fby}}$, que se define como:

$$Lu(A)^{s.s,s}_{\mathbf{fby}} = \lambda(x,y) :(\mathbf{N}_+\rightarrow(A_s^{\#})_\perp)\times(\mathbf{N}_+\rightarrow(A_s^{\#})_\perp).$$
$$(\ \lambda t :\mathbf{N}_+.\ \textit{if } t{=}1 \textit{ then } x(1) \textit{ else } y(t\text{-}1) :(A_s^{\#})_\perp\ ) :(\mathbf{N}_+\rightarrow(A_s^{\#})_\perp)$$

- Para todo género *s*, y para todo w∈{ s.s, s.s.s, s.s.s.s, ... }, el símbolo || denota a la familia de funciones $Lu(A)^{s.s,s}_{||}$, $Lu(A)^{s.s.s,s}_{||}$, ... tal que cualquiera de sus componentes se define como:

$$Lu(A)^{s\ \cdots\ s,s}_{||} = \lambda(x_1 \overset{n}{\ldots} x_n) :(\mathbf{N}_+\rightarrow(A_s^{\#})_\perp)\times \overset{n}{\ldots} \times(\mathbf{N}_+\rightarrow(A_s^{\#})_\perp).$$
$$(\ \lambda t :\mathbf{N}_+.\ (\ x_1(t) \overset{n}{\ldots} x_n(t)\ )\downarrow((t\text{-}1)\ mod\ n)+1) :(A_s^{\#})_\perp\ ) :(\mathbf{N}_+\rightarrow(A_s^{\#})_\perp)$$

Nótese cuáles serán los símbolos sintácticos que de ahora en adelante se utilizarán para denotar algunas de las funciones temporales: >> (que lo escribiré infijo) para el operador *sample*, << (también infijo) para el operador *replicate* y || (infijo) para el operador *interleave*.

Como puede observarse, solamente se han ampliado las nociones de $Lu(\Sigma)$ y $Lu(A)$, sin embargo, al ser éstas utilizadas en muchas otras definiciones†, muchos conceptos quedan redefinidos. El sistema de transformación presentado en el capítulo anterior sigue también siendo aplicable, ya que se definió sin hacer ninguna asunción respecto del tipo de operadores que pudieran aparecer en la especificación ecuacional.

† Sintaxis de una especificación ecuacional (definición 2.25), semántica de una especificación ecuacional (definición 2.27), traza (definición 2.28), estímulo (definición 2.19) y simulación (definición 2.31).

4.4 Formalización de las propiedades de los operadores temporales como *Lu*(Σ)-ecuaciones.

Para que las propiedades presentadas en §4.2 puedan aplicarse vía el sistema de transformación propuesto en el capítulo 3 es necesario que, desde el aspecto sintáctico, se formalicen como *Lu*(Σ)-ecuaciones de cierto género. Recuérdese que, según la definición 2.12, este tipo de ecuaciones permiten representar fórmulas universales de primer orden fuertemente tipificadas. Sin embargo, dado que no todas las propiedades presentadas son de primer orden y que, además, han sido propuestas de una manera independiente a los tipos concretos de los argumentos, dicha formalización no puede realizarse directamente.

Esta sección se dedica a realizar algunas consideraciones para alcanzar una formalización adecuada de las propiedades para que puedan aplicarse sobre una especificación ecuacional.

La primera dificultad nace de la naturaleza polimórfica de los operadores temporales. Esto ya obligó, en la definición 4.3, a definir familias de símbolos sobrecargados para que, mediante un mismo símbolo de operación común, pudieran denotarse operadores temporales distintos que operasen sobre conjuntos soporte distintos. Como consecuencia de ello, cada una de las propiedades en las que intervenga un operador temporal, también deberá expresarse mediante una familia de *Lu*(Σ)-ecuaciones con una cardinalidad igual al número de géneros de la signatura. Dicha familia, al igual que el resto de las nociones presentadas, variará de un proyecto de diseño a otro a merced de la signatura que en cada uno de ellos se especifique. Obviamente, dado que el conjunto de géneros de toda signatura es finito, la familia de ecuaciones también será finita.

Nótese que si bien, las propiedades establecidas en el capítulo 3 son más genéricas que los conjuntos finitos de ecuaciones que propongo, esto no supone ningún problema práctico ya que, para un ciclo de diseño particular, basta con la particularización de las mismas a los géneros concretos con los que el circuito trabaja.

EJEMPLO 4.2

Para la signatura del ejemplo 2.4, dado que sólo se declaran dos géneros (*Bool* y *Ent*), las propiedades IFBY e IREP pueden formalizadas, cada una de ellas, por una familia de dos ecuaciones:

{ *propiedad IFBY*

({ X_{Ent} ≡ { y, x } }, next(y fby x), x), *particularizada al género Ent*

({ X_{Bool} ≡ { y, x } }, next(y fby x), x) *particularizada al género Bool*

}

{ *propiedad IREP*

({ X_{Ent} ≡ { x }, X_N ≡ { k } }, k >> (x << k), x), *particularizada al género Ent*

({ X_{Bool} ≡ { x }, X_N ≡ { k } }, k >> (x << k), x) *particularizada al género Bool*

}

Obsérvese que es imprescindible explicitar el género de las variables para poder resolver el género de la ecuación. Esto evita caer en ambigüedades cuando se realicen los ajustes que requieren las reglas de aplicación.

El segundo problema nace de la exigencia impuesta en las condiciones de las reglas de aplicación de que el conjunto de variables del término a ajustar de una ecuación, sea un superconjunto del conjunto de variables del otro término que la forma. El objeto de esta restricción (véase §3.1.1) era evitar la ambigüedad que provoca la adición sobre el término reescrito de variables no ajustadas. Esta restricción, por ejemplo, evitaría que las reglas IFBY o

IREP pudieran aplicarse de derecha a izquierda por quedar sin ajustar *y* ó *k* respectivamente, lo que reduciría el alcance de las propiedades.

Para solucionarlo y sólo cuando quieran aplicarse en el sentido prohibido, bastará con reemplazar cada una de las variables libres por un término cerrado concreto. No obstante, este término cerrado, si se deseara formalizar por completo el concepto de variable universal no ligada y ésta sólo ocurriera una vez en el término, podría ser el símbolo comodín.

EJEMPLO 4.3

Así, la primera propiedad del ejemplo anterior podría formalizarse, para que pudiera ser aplicada en ambos sentidos, como el conjunto de ecuaciones siguientes:

{ *propiedad IFBY*

({ $X_{Ent} \equiv$ { x } }, next(# fby x), x), *particularizada al género Ent*

({ $X_{Bool} \equiv$ { x } }, next(# fby x), x) *particularizada al género Bool*

}

Una vez aplicadas, el usuario podría reemplazar los comodines por el término que más le convenga utilizando la regla de reemplazo. Sin embargo, este método no puede aplicarse a la propiedad IREP por existir 2 ocurrencias de la variable *k* en su término izquierdo. Si dichas ocurrencias se reemplazaran por comodines, el usuario podría a su vez reemplazar cada uno de ellos por términos diferentes violando la semántica de la ecuación.

El tercer inconveniente, para una formalización directa de las propiedades de los operadores temporales, proviene del operador *interleave*. Como se dijo en §4.1, este operador o bien puede considerarse como una única función con un número variable de argumentos, o bien puede ser contemplado como una familia infinita de funciones con número de argumentos fijo. Esto ya

obligó en §4.2 a utilizar metavariables para que, mediante un único esquema, pudiera establecerse un conjunto infinito de propiedades válidas para cualquier número de argumentos que tuviera dicho operador. Es esta infinitud, o lo que es lo mismo, la ausencia de metavariables en el mecanismo de especificación, lo que trae problemas a la hora de formalizar algunas propiedades como *Lu*(Σ)-ecuaciones, ya que se requeriría un número infinito de ellas para plasmar todo ese conocimiento que resume una única fórmula.

Sin embargo, a fines prácticos, no es necesario tener en un ciclo de diseño tal cantidad de ecuaciones ya que, para un circuito particular, el conjunto de ecuaciones aplicable siempre es finito. Concretando, si el circuito se planifica en *k* ciclos, sólo deberán aparecer dentro del cuerpo de la especificación ecuacional operadores *interleave* con k argumentos, de este modo sólo serán necesarias particularizaciones de las propiedades en las que intervenga el operador *interleave* con dicho número de argumentos.

EJEMPLO 4.4

El esquema NINT que utiliza la metavariable *k*, representa a un conjunto infinito de fórmulas de primer orden que no utilizan metavariables:

{

$\forall x \in Lu(A)_S$, *interleave*(x) = x *para k = 1*

$\forall x \in Lu(A)_S$, *interleave*(x, x) = x *para k = 2*

$\forall x \in Lu(A)_S$, *interleave*(x, x, x) = x *para k = 3*

$\forall x \in Lu(A)_S$, *interleave*(x, x, x, x) = x *para k = 4*

...

}

Este conjunto y para la signatura del ejemplo 2.4, es directamente formalizable mediante un conjunto también infinito de conjuntos de *Lu*(Σ)-ecuaciones:

{

{ *propiedad NINT con k=2*

$(\{ X_{Ent} \equiv \{ x \} \}, (x \| x), x),$ *particularizada al género Ent*

$(\{ X_{Bool} \equiv \{ x \} \}, (x \| x), x)$ *particularizada al género Bool*

}

{ *propiedad NINT con k=3*

$(\{ X_{Ent} \equiv \{ x \} \}, (x \| x \| x), x),$ *particularizada al género Ent*

$(\{ X_{Bool} \equiv \{ x \} \}, (x \| x \| x), x)$ *particularizada al género Bool*

}

{ *propiedad NINT con k=4*

$(\{ X_{Ent} \equiv \{ x \} \}, (x \| x \| x \| x), x),$ *particularizada al género Ent*

$(\{ X_{Bool} \equiv \{ x \} \}, (x \| x \| x \| x), x)$ *particularizada al género Bool*

}

...

}

Sin embargo, para un ciclo de diseño particular en el que se realice una planificación particular en un número concreto de ciclos, la propiedad NINT basta que sea formalizada como uno sólo de los anteriores conjuntos de ecuaciones.

El cuarto problema aparece en las propiedades distributivas de los operadores temporales. La razón es que no son fórmulas universales de primer orden ya que algunas variables ($A_{\sigma}^{W,S}$) no toman como valores a los elementos de un dominio sino que toman como valores funciones entre dominios. La solución, en lugar de modificar el sistema para que acepte $Lu(\Sigma)$-ecuaciones de orden superior, pasa por la aplicación de una idea aceptable desde el punto de vista práctico: dado que sólo es posible expresar aquello para lo que se tiene soporte sintáctico, nunca será necesario aplicar una de dichas propiedades de segundo orden sobre un operador cualquiera, sino sólo sobre aquellos operadores relevantes para el problema, es decir, los que están definidos por la signatura (que podrá variar de un diseño a otro,

pero que para cada diseño particular es fija). Así cada una de las propiedades de orden superior podrá expresarse como un conjunto finito de $Lu(\Sigma)$-ecuaciones, cuya cardinalidad será el número de operadores de aridad mayor que 0 que posee la signatura. Nuevamente la propiedad es más general que el conjunto, pero téngase en cuenta, por ejemplo, que en un circuito en el que sólo se suma y se resta, resulta indiferente a fines prácticos si el operador *fby* es o no distributivo sobre la división.

EJEMPLO 4.5

Para la signatura del ejemplo 2.4, que posee 5 símbolos de operación con aridad distinta de 0, la propiedad DNEXT queda completamente formalizada mediante el conjunto siguiente:

{ *propiedad DNEXT*

({ $X_{Bool} \equiv$ { x } }, next(no(x)), no(next(x))), *particularizada para el símbolo no*

({ $X_{Ent} \equiv$ { x, y } }, next(x)+next(y), next(x+y)), *particularizada para el símbolo +*

({ $X_{Ent} \equiv$ { x, y } }, next(x)-next(y), next(x-y)), *particularizada para el símbolo -*

({ $X_{Ent} \equiv$ { x } }, -next(y) , next(-x)), *particularizada para el símbolo -*

({ $X_{Ent} \equiv$ { x, y } }, next(x)>next(y), next(x>y)) *particularizada para el símbolo >*

}

La quinta dificultad aparece en las propiedades de existencia de elementos neutros. En ellas se establece que cualquier secuencia formada por la repetición de un mismo elemento constante no es afectada por los operadores temporales. El problema es que en el formalismo propuesto, todo término cerrado denota un elemento constante y, dado que el número de términos cerrados que pueden generarse a partir de una signatura es en general infinito (aunque el conjunto soporte no lo sea), nuevamente es necesario un conjunto infinito de $Lu(\Sigma)$-ecuaciones para expresar dicha propiedad.

Para solucionarlo, utilizaré una idea que permita alcanzar resultados similares a los descritos por dichas propiedades. La idea está basada en que todo término cerrado no posee símbolos de señal. De este modo, si cada una de las propiedades se expresa como un conjunto de $Lu(\Sigma)$-ecuaciones en las que sólo aparezcan símbolos constantes (operadores de aridad 0), los resultados de las propiedades pueden obtenerse mediante la aplicación de una de esas ecuaciones y una posterior y sucesiva aplicación de las ecuaciones de distributividad.

EJEMPLO 4.6

Para la signatura del ejemplo 2.4, que posee 3 símbolos constantes, la propiedad NNEXT podría formalizarse mediante el conjunto de ecuaciones:

{ *propiedad NNEXT*

(∅, next(cierto), cierto), *particularizada para el símbolo cierto*

(∅, next(falso), falso), *particularizada para el símbolo falso*

(∅, next(0), 0) *particularizada para el símbolo 0*

}

Sólo con este conjunto de ecuaciones sería imposible aplicar la propiedad NNEXT sobre un término cerrado genérico (que la cumpliría por denotar a una secuencia constante). Sin embargo, si junto con ellas se utilizan también las que en el ejemplo 4.5 formalizan la propiedad DNEXT, podrían alcanzarse resultados equivalentes a los que se alcanzarían con la propiedad NNEXT general. Así, sea el término cerrado:

no(no(falso))

Este término en función del álgebra soporte denotará cierto elemento que será constante, por lo que según la propiedad NNEXT, la acción del operador *next* sobre él lo deja inalterado. Obsérvese como dicha conclusión puede obtenerse mediante la aplicación sucesiva de algunas $Lu(\Sigma)$-ecuaciones:

next(no(no(falso))) *DNEXT particularizada para el símbolo no*

= no(next(no(falso))) *DNEXT particularizada para el símbolo no*

= no(no(next(falso))) *DNEXT particularizada para el símbolo no*

= no(no(falso)) *NNEXT particularizada para el símbolo falso*

De ahora en adelante, para referirme a una $Lu(\Sigma)$-ecuación concreta de las incluidas en cierto conjunto que formaliza cierta propiedad de las presentadas en §4.2, utilizaré una notación funcional al estilo de la utilizada en el capítulo 3 para abreviar la referencia a una regla concreta de transformación. Así para referirme a cierta $Lu(\Sigma)$-ecuación, en lugar de formularla, utilizaré el nombre de la propiedad junto a una colección de argumentos actuales que fijen los valores concretos de cada una de las metavariables presentes en la propiedad.

Así las propiedades de operadores inversos (donde t es el término que reemplazará la aparición de la variable libre y s el género de la ecuación) se abreviarán como:

IFBY(t, s **), INEXT(** s **), IREP(** t, s **)**

las de distributividad de los operadores temporales respecto a un cierto operador no temporal σ (que implícitamente fija el género de la ecuación), como:

DFBY(σ **), DNEXT(** σ **), DREP(** σ **), DSAM(** σ **), DINT(** k, σ **)**

las de existencia de elementos neutros respecto a un cierto símbolo constante σ, como:

NFBY(σ **), NNEXT(** σ **), NREP(** k, σ **), NSAM(** k, σ **), NINT(** k **)**

los teoremas de síntesis (en donde s es el género de la ecuación):

TMT(k, m, s **), ADRET(** k, m, s **), FRAG(** k, m, n, s **),**

MEMT(k, m, n, s **), MEMT2(** k, m, n, s **), DET(** k, m, s **), INAT(** k, m, s **)**

y los teoremas de proyección RT:

MUXI($x_{j1}, \overset{k}{\ldots}, x_{jk}$, s **)**

Obsérvese que los argumentos que concretan una ecuación particular tienen un claro significado desde el punto de vista de la SAN. El parámetro *k*, en todos los casos, determina la latencia de la planificación; el parámetro *m*, fija un ciclo concreto de la planificación (el ciclo *k-m*, ya que *m* mide el número de ciclos que separan a un ciclo concreto del último); y *n* la vida medida en ciclos que, para cierta planificación, tienen los valores que se calculan.

Por otro lado, y por conveniencia, dado que el género de una propiedad puede extraerse claramente del contexto sobre el que se aplique, de ahora en adelante **no aparecerá explícitamente el género al referirnos a una propiedad** por lo que su notación abreviada quedará como:

IFBY(*t* **), INEXT, IREP(** *t* **)**

TMT(*k*, *m* **), ADRET(** *k*, *m* **), FRAG(** *k*, *m*, *n* **),**

MEMT(*k*, *m*, *n* **), MEMT2(** *k*, *m*, *n* **), DET(** *k*, *m* **), INAT(** *k*, *m* **)**

MUXI($x_{j1}, \overset{k}{\ldots}, x_{jk}$ **)**

4.4.1 Estudio de la complejidad de la generación de conjuntos de Lu(Σ)-ecuaciones para un proceso de síntesis concreto.

La formalización de las propiedades de los operadores temporales como conjuntos de *Lu*(Σ)-ecuaciones puede tener consecuencias importantes desde el punto de vista práctico. La razón es que el tamaño de los conjuntos, al igual que la complejidad de los términos que forman cada una de las ecuaciones, pueden variar de un diseño a otro. Así, dado que para cada proceso concreto de diseño formal, es necesario generar un conjunto de ecuaciones particular, dicho proceso de generación debe tenerse en cuenta a la hora de evaluar la complejidad espacial y temporal del sistema de síntesis por derivación.

Si se decide realizar una implementación que genere las ecuaciones conforme se vayan necesitando y libere espacio tras su uso, la complejidad espacial de la implementación será del orden de la complejidad de la ecuación más grande, y la complejidad temporal se diluirá dentro del propio algoritmo de síntesis. Si, por el contrario, se decide generar todas las ecuaciones en una fase previa a la síntesis, será necesario tener en cuenta el espacio necesario para almacenar todas las ecuaciones, el tiempo necesario para generarlas e incluso, dentro del proceso de síntesis, el tiempo de búsqueda de la ecuación que se necesita.

Como puede comprobarse en la discusión de la anterior sección, existen dos parámetros que condicionan de modo directo el tamaño de los conjuntos de ecuaciones, estos son:

- $|S|$ *número de géneros de la signatura*
- $|\Sigma|$ *número de operadores de la signatura*

además, existe un tercero que condiciona tanto el tamaño de éstos como la complejidad de los términos, que es:

- λ *latencia de la planificación*

Comenzaré estudiando el crecimiento, en función de los anteriores factores, del tamaño de los conjunto de ecuaciones.

- **Propiedades de operadores inversos**. Sólo se verán afectados por la polimorfía de los operadores temporales, por lo que se necesitara de cada ellas una por género:

$$| \text{IFBY} | = | \text{INEXT} | = | \text{IREP} | = |S|$$

- **Propiedades distributivas**. Serán necesarias una particularización de cada una de ellas para cada operador de la signatura con aridad distinta de 0. En el caso de DINT, dado que la latencia es fija para cada proceso de diseño, la latencia no afecta al tamaño del conjunto:

$$| \text{DFBY} | = | \text{DNEXT} | = | \text{DREP} | = | \text{DSAM} | = | \text{DINT} | = |\Sigma|$$

- **Propiedades de existencia de elementos neutros**. Será necesaria particularizar cada una de ellas para cada operador de la signatura con aridad 0, así:

  $$| NFBY | = | NNEXT | = | NREP | = | NSAM | = | NINT | = |\Sigma|$$

- **TMT, ADRET, DET, INAT y MUXI**. Será necesario una por género y dentro de cada género una por cada posible valor de *m*. Dado que *m* indica un ciclo particular dentro de la planificación que varía entre 1 y λ, resulta que:

  $$| TMT | = | ADRET | = | DET | = | INAT | = | MUXI | = \lambda * |S|$$

- **FRAG y MEMT**. Nuevamente serán necesarias una por género y ciclo *m* particular, pero además cada una de ellas deberá particularizarse para un posible número de ciclos de retardo *n*. El número de retardos *n* indica el número de ciclos que separan las operaciones de producción y lectura de un valor particular† y, para toda operación y en todo proceso de diseño, nunca supera el número de ciclos de la planificación, por lo que el número de ecuaciones totales resulta:

  $$| FRAG | = | MEMT | = \lambda * (\lambda - 1) * |S|$$

En cuanto a la complejidad de la generación de una ecuación particular es posible distinguir únicamente dos categorías, aquellas que su complejidad es constante:

- **IFBY, INEXT, IREP, DFBT, DNEXT, DREP, DSAM, NEFBY, NNEXT, NREP, NSAM**

y aquellas cuya complejidad crece linealmente con la latencia de la planificación:

- **DINT, NINT, TMT, ADRET, FRAG, MEMT, DET, INAT, MUXI**

Para finalizar, y como consecuencia de este estudio, puede obtenerse una conclusión clara sobre la implementación del sistema de síntesis. De las dos opciones propuestas, la segunda de pregeneración de ecuaciones es

† En el próximo capítulo se discutirá en detalle.

totalmente inadecuada, ya que en algunos conjuntos de ecuaciones (FRAG y MEMT) se alcanzan complejidades cúbicas, y la experiencia dice que sólo un pequeño número de las ecuaciones generadas son utilizadas (para que se utilizaran todas, sería necesario que en todos los ciclos hubiera operaciones con dependencias de datos respecto a operaciones planificadas en todos los ciclos restantes). Por ello se aconseja adoptar la otra alternativa que propone la implementación de generadores de ecuaciones bajo demanda. Opción que ha sido desarrollada con éxito en el prototipo que se discute en el capítulo 5.

4.5 Ejemplos de la implantación de técnicas de diseño de alto nivel sobre el sistema de síntesis formal.

Al igual que en el pasado capítulo (§3.3) se dedicó una sección al estudio de cómo reproducir los resultados de algunas técnicas de diseño mediante secuencias de transformaciones, esta sección ilustra la utilización de los operadores temporales y de sus propiedades para reproducir, también por derivación, los resultados de técnicas específicas de alto nivel. Nuevamente el objetivo es demostrar la versatilidad del mecanismo de especificación y del sistema de transformación para realizar síntesis de alto nivel correcta.

Puede resultar extraño que aplique técnicas de SAN directamente sobre especificaciones ecuacionales, cuando en todo proceso de SAN se asume una fase de compilación previa para convertir la especificación de entrada en una representación intermedia más fácilmente manipulable por el resto de las fases. Sin embargo, dado que esta representación suele tener la forma de grafo de dependencias y tal y como se comprobó en §2.2.3 una especificación ecuacional es equivalente a un grafo de flujo, no es necesaria dicha fase para manipular el formalismo propuesto. No obstante en las

siguientes secciones mostraré, por claridad, los grafos que son tan habituales en SAN y que equivalen a las especificaciones ecuacionales que se vayan obteniendo por derivación.

4.5.1 Planificación de operaciones y planificación del ciclo de actualización de los retardos arquitectónicos.

La planificación es la tarea de determinar el ciclo de reloj de comienzo de cierta operación de manera que se respeten las ligaduras de precedencia impuestas por un grafo de secuenciamiento. Este grafo de secuenciamiento (que suele obtenerse tras la compilación de la especificación inicial), determina qué operaciones requieren como operandos los resultados calculados por otras operaciones. Se dice que una planificación tiene una latencia de k ciclos de reloj, si el circuito planificado resultante es capaz de realizar todo el algoritmo inicial en ese número de ciclos.

Para reproducir la planificación de un nodo particular en un ciclo concreto, es necesario recordar que, en general, el período de reloj de un circuito planificado en *k* ciclos es *k* veces menor que el período de muestreo del sistema. Por ello, para planificarla en cierto ciclo *j* será necesario aumentar, primero, la frecuencia de operación de dicho operador (vía la aplicación de IREP(*k*)) y segundo, asignarle un ciclo (vía la propiedad TMT(*k*,*k-j*)).

EJEMPLO 4.7

Sea el cuerpo de la especificación ecuacional del filtro recursivo de segundo orden expandida parcialmente y simplificada (usando repetidas veces las reglas de expansión y limpieza):

```
body
out = z - ( a1*A + t1 )
z = ( b1*A + b2*B ) + in
A = 0 fby z
B = 0 fby A
```

t1 = a2*B

Asignemos la operación de multiplicación ubicada en la definición de *t*1, al ciclo 2 de una planificación en 4 ciclos. Dicha tarea se muestra esquemáticamente en la fig. 4.5.

	body
	out = z - (a1*A + t1)
	z = (b1*A + b2*B) + in
	A = 0 fby z
AplicacionDI(t1,ε,IREP(4))	B = 0 fby A
AplicacionID(t1,ε,TMT(4,2))	t1 = 4>>(# fby # fby (#||next next ((a2*B)<<4)||#||#))

Como puede observarse se ha aplicado IREP con el valor actual de la latencia y TMT con el mismo valor de latencia y con *m*=2 por ser *m* el número de ciclos que separa al ciclo por planificar respecto del último ciclo. Nótese

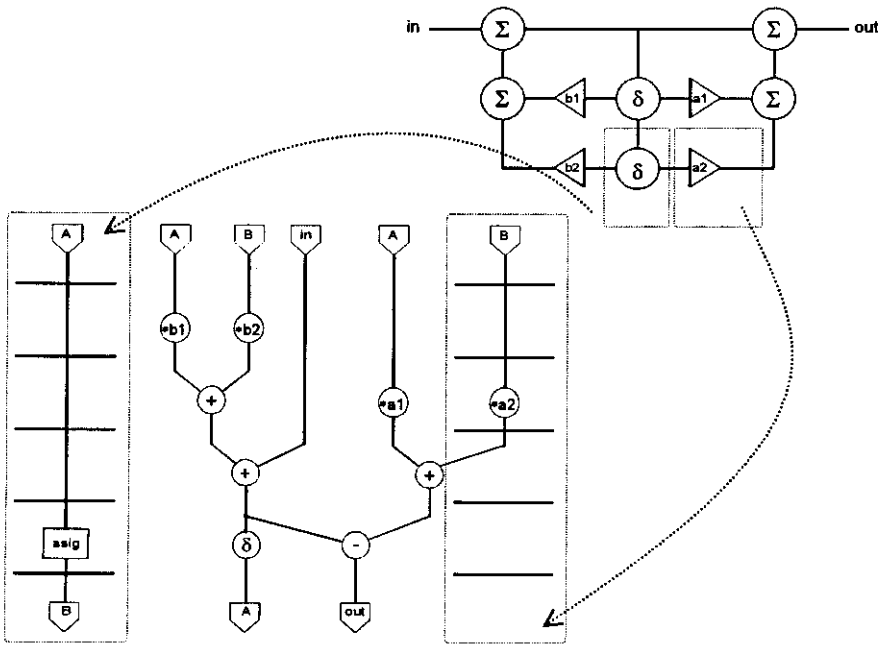


Fig. 4.5: Ejemplos de planificación.

cómo se ha planificado individualmente una única operación y el resto sigue funcionando a la frecuencia de muestreo. Por ello, el valor calculado se retrasa dos ciclos de reloj para que coincida con el ciclo de muestreo y mediante *next* se exige que los operandos estén calculados al menos dos ciclos de reloj antes del final del ciclo de muestreo. Mediante *replicate* y *sample* con el mismo parámetro 4, se indica la relación entre frecuencias de reloj y muestreo.

Cuando lo que se desea planificar es el ciclo en que se actualizará el valor de un retardo arquitectónico, se procede del mismo modo sobre el operador *fby* que se desea planificar:

	body
	out = z - (a1*A + t1)
	z = (b1*A + b2*B) + in
	A = 0 fby z
AplicacionDI(B, ε, IREP(4))	B = 4>> (# || # || # || (0 fby A) << 4)
AplicacionID(B, ε, TMT(4,0))	t1 = a2*A

Una reproducción esquemática de esta planificación también se muestra en la fig. 4.5.

4.5.2 Planificación encadenada (chaining).

Cuando dos o más operaciones con ligaduras de precedencia se planifican en el mismo ciclo, se dicen que están encadenadas. Una planificación encadenada es la que permite dos o más operaciones de ese tipo.

El esquema para realizar una planificación con encadenamiento dentro del formalismo de especificación ecuacional, no es distinto del mostrado anteriormente, la única diferencia radica en que deben planificarse a la vez grupos de operaciones con dependencias de datos.

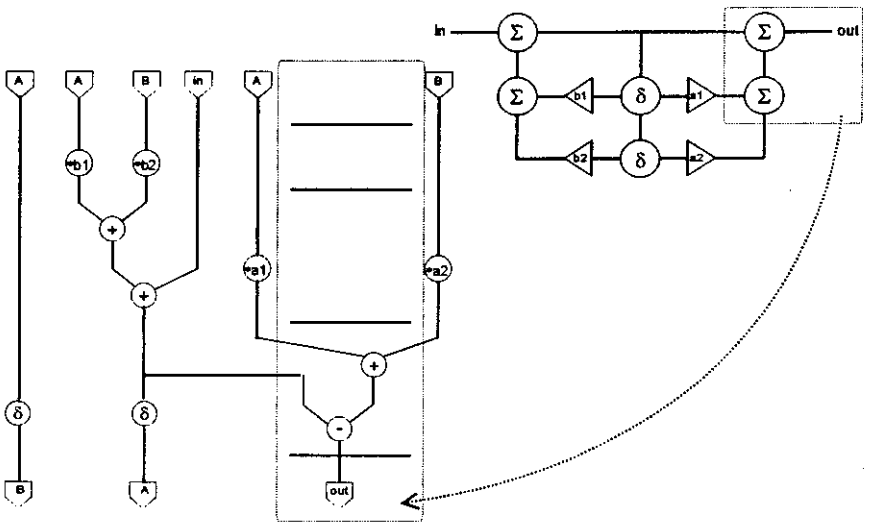


Fig. 4.6: Ejemplo de planificación encadenada.

EJEMPLO 4.8

Reproduzcamos la planificación encadenada mostrada en la fig. 4.6, en la que una suma y una resta se encadenan en el tercer ciclo de una planificación con latencia 3. Las transformaciones a realizar son:

```
                                    body
                                    out = z - ( t1 + t2 )
                                    z = ( b1*A + b2*B ) + in
                                    A = 0 fby z
                                    B = 0 fby A
                                    t1 = a1*A
                                    t2 = a2*B
```

```
                                    body
                                    out = 3 >>( # || # || ( z - ( t1 + t2 ) ) << 3 )
                                    z = ( b1*A + b2*B ) + in
                                    A = 0 fby z
                                    B = 0 fby A
AplicacionDI( out, ε, IREP(3) )     t1 = a1*A
AplicacionID( out, ε, TMT(3,0) )    t2 = a2*B
```

4.5.3 Planificación completa de conductas: corrección de una planificación.

Las anteriores secciones han fijado un mecanismo para planificar individualmente una única operación. Para poder chequear que no se violan las dependencias de datos es necesario realizar la planificación de todas las operaciones que forman una especificación ecuacional. A continuación definiré un posible esquema de transformación que permite planificar completamente cualquier conducta. Este esquema además de ser fácilmente automatizable (tal y como se verá en el próximo capítulo), permite chequear sin dificultad si la planificación realizada es correcta y, en caso de no serlo, permite localizar cual fue la decisión de diseño errónea.

El esquema de transformación comienza aplicando la propiedad IREP sobre todas las fuentes de datos (puertos de entrada, constantes y retardos arquitectónicos) y, utilizando las reglas de expansión, sustitución y eliminación junto con las propiedades DSAM y NSAM, va propagando el operador *sample* de unas definiciones a otras. El objeto de dicha propagación es ir aumentando el número de operaciones que pueden planificarse vía el teorema TMT. Este proceso continúa hasta que el operador *sample* alcanza todos los destinos de datos (puertos de salida y retardos arquitectónicos). Tras esto se reemplazan los retardos arquitectónicos por retardos convencionales utilizando ADRET y se extraen las vidas de las variables intermedias intentando eliminar todos los operadores *next*. Para esto último se utilizan las propiedades IFBY, DNEXT, NNEXT e INAT. Si la especificación ecuacional resultante no tiene ocurrencias del operador *next* (único operador no implementable), será correcta, si las tiene se han violado en dicho proceso algunas dependencias de datos.

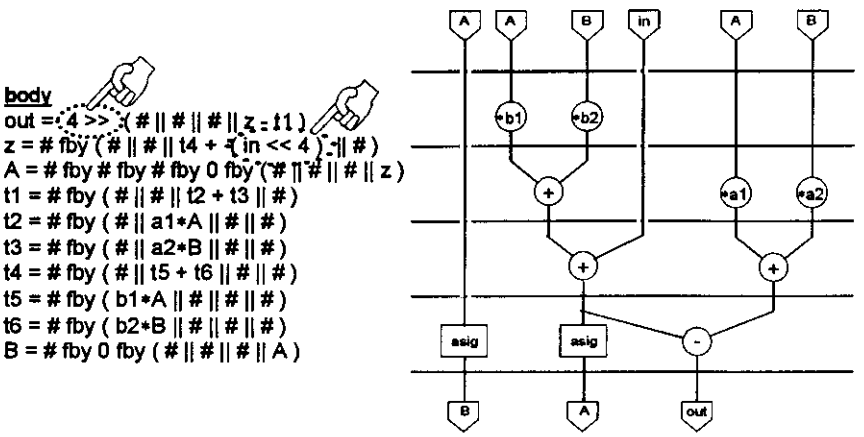


Fig. 4.7: Comportamiento correctamente planificado.

Nótese que cuando se dice que en el proceso de diseño se han violado algunas dependencias de datos, quiere decirse que el circuito obtenido no es implementable, aunque sí que sea conductualmente equivalente a la especificación original. Esto es posible gracias a la existencia del operador *next*.

EJEMPLO 4.9

Realicemos la planificación en 4 ciclos que se muestra en el lado derecho de la fig. 4.7. Para ello, partiendo de la especificación ecuacional inicial del ejemplo 4.7, se aplica la ecuación IREP(4) sobre cada una de las fuentes de datos:

body
out = z - (a1*A + a2*B)
z = (4 >> (in << 4)) + (b1*A + b2*B)
A = 4 >> ((0 fby z) << 4)
B = 4 >> ((0 fby A) << 4)

A continuación se planifican las actualizaciones de los registros arquitectónicos (*A* y *B*) en el cuarto ciclo, usando TMT(4, 0) y se propaga el operador *sample* a través de la especificación:

```
                                  body
                                  out = z - ( a1*A + a2*B )
                                  z = ( 4 >> ( in << 4 ) ) + ( b1*A + b2*B )
                                  A = 4 >> ( # || # || # || ( 0 fby z ) << 4 )
                                  B = 4 >> ( # || # || # || ( 0 fby A ) << 4 )
                                  ..........................................
Expansion( A, 2, t1 )             |
Substitucion( out, 2.1.2 )        |
Substitucion( z, 2.1.2 )          |
Substitucion( B, 2.4.1.2 )        |
Eliminacion( A )                  | body
Renombrado( t1, A )               | out = z - ( t2 + a2*B )
Expansion( out, 2.1, t2 )         | z = ( 4 >> ( in << 4 ) ) + ( b1*(4>>A) + b2*B )
AplicacionDI( t2, 1, NSAM(4,a1) ) | B = 4 >> ( # || # || # || ( 0 fby (4>>A) ) << 4 )
AplicacionID( t2, ε, DSAM(*) )    | A = ( # || # || # || ( 0 fby z ) << 4 )
                                  | t2 = 4 >> ( a1*A )
```

Como puede verse, ahora sobre la definición de *t2* puede aplicarse TMT(4, 2) para planificar la multiplicación en el segundo ciclo. Repitiendo el proceso de propagación y de aplicación sucesiva de TMT(4, *ciclo*) hasta que el operador *sample* alcance los destinos de datos, se alcanza la siguiente especificación ecuacional:

```
body
  out = 4 >> ( # || # || # || z - t1 )
  z = # fby ( # || # || next ( ( in << 4 ) + t4 ) || # )
  A = ( # || # || # || ( 0 fby ( 4 >> z ) ) << 4 )
  B = ( # || # || # || ( 0 fby ( 4 >> A ) ) << 4 )
  t1 = # fby ( # || # || next (t2 + t3) || # )
  t2 = # fby # fby ( # || next next (a1*A) || # || # )
  t3 = # fby # fby ( # || next next (a2*B) || # || # )
  t4 = # fby # fby ( # || next next (t5 + t6) || # || # )
  t5 = # fby # fby # fby ( next next next (b1*A) || # || # || # )
  t6 = # fby # fby # fby ( next next next (b2*B) || # || # || # )
```

Aplicando ADRET(4, 0) sobre las definiciones de *A* y *B* transformamos los retardos arquitectónicos en cadenas de retardos convencionales.

```
body
  out = 4 >> ( # || # || # || z - t1 )
  z = # fby ( # || # || next ( ( in << 4 ) + t4 ) || # )
  A = # fby # fby # fby 0 fby ( # || # || # || z )
  B = # fby # fby # fby 0 fby ( # || # || # || A )
  t1 = # fby ( # || # || next (t2 + t3) || # )
```

t2 = # fby # fby (# || next next (a1 ∗A) || # || #)
t3 = # fby # fby (# || next next (a2 ∗B) || # || #)
t4 = # fby # fby (# || next next (t5 + t6) || # || #)
t5 = # fby # fby # fby (next next next (b1 ∗A) || # || # || #)
t6 = # fby # fby # fby (next next next (b2 ∗B) || # || # || #)

Para finalizar, expandiendo las subexpresiones en las que ocurren los operadores *next*, substituyendo en ellas cada señal por su definición y utilizando principalmente la propiedad IFBY para eliminar pares *fby-next*, es posible alcanzar a la especificación ecuacional mostrada en la figura 4.7.

Como puede observarse, el significado de los operadores temporales en una especificación ecuacional planificada es claro desde el punto de vista hardware. Hay un operador *interleave* por operación, de manera que la posición que ésta última ocupa, indica el ciclo en que ha sido planificada. El resto de los argumentos de *interleave* son comodines para indicar la 'disponibilidad' del operador en los demás ciclos. El número de operadores *fby* que aparecen en cada definición indican la vida, medida en ciclos, del valor que se calcula en ella. Por su parte, los operadores *sample* y *replicate*, sobre los puertos de salida y entrada respectivamente, indican la relación entre las frecuencias de muestreo y de reloj del circuito. Dado que no hay ninguna ocurrencia del operador *next* la especificación ecuacional es realizable: bastaría con implementar los operadores *fby* e *interleave* con registros y multiplexores, los operadores no temporales, con módulos combinacionales y aumentar la frecuencia de reloj respecto de la frecuencia de muestreo de partida.

Si se intenta reproducir el proceso descrito para realizar una planificación que viole las dependencias de datos (como la mostrada en el lado derecho de la fig. 4.8) se alcanza una especificación ecuacional válida pero no implementable. Obsérvese cómo en la especificación ecuacional de la fig. 4.8, aparece un único operador *next* en la definición de *t*3. Este operador indica que el resultado de la multiplicación *a*2 ∗*B* se necesita, al menos, un ciclo

antes del ciclo en que es calculado (recuérdese que el significado de *next* es adelantar un ciclo las componentes de una secuencia).

Esta última consideración permite encontrar una nueva ventaja del mecanismo de especificación: es un buen soporte para la verificación formal de diseños ya que, al permitir expresar planificaciones incorrectas, no sólo puede detectarse que lo son, sino que también permite localizar el lugar del error e indicar una manera inmediata de corregirlo.

4.5.4 Planificación con plegado de bucles (loop-folding).

Cuando la latencia de un circuito planificado con métodos convencionales es demasiado alta, existe un método de planificación que permite solapar iniciaciones sucesivas del algoritmo para que, a costa de retardar la salida de los primeros resultados, se reduzca el tiempo que transcurre entre dos resultados consecutivos. Esta idea es la misma que utiliza la técnica de segmentación (§3.3.2) en implementaciones monociclo y que, dentro del

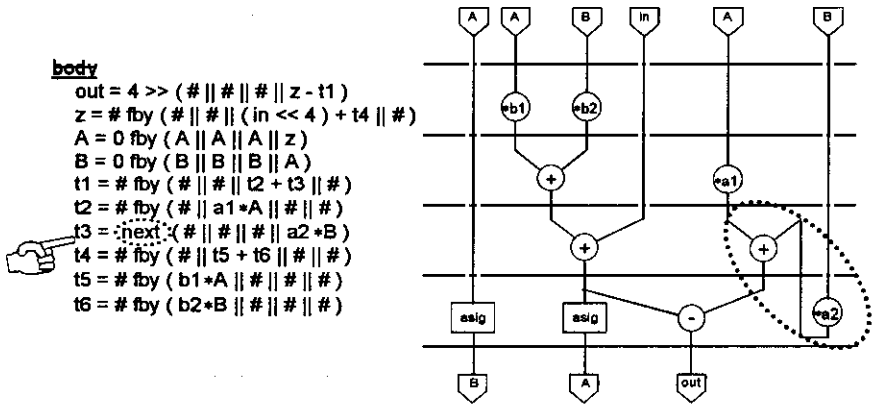


Fig. 4.8: Comportamiento incorrectamente planificado.

retardos-arquitectónicos-en-cadenas-de-retardos-convencionales.

```
body
  out = 4 >> ( # || # || # || z - t1 )
  z = # fby ( # || # || next ( ( in << 4 ) + t4 ) || # )
  A = # fby # fby # fby 0 fby ( # || # || # || z )
  B = # fby # fby # fby 0 fby ( # || # || # || A )
  t1 = # fby ( # || # || next (t2 + t3) || # )
```



```
t2 = # fby # fby ( # || next next (a1 *A) || # || # )
t3 = # fby # fby ( # || next next (a2 *B) || # || # )
t4 = # fby # fby ( # || next next (t5 + t6) || # || # )
t5 = # fby # fby # fby ( next next next (b1 *A) || # || # || # )
t6 = # fby # fby # fby ( next next next (b2 *B) || # || # || # )
```

Para finalizar, expandiendo las subexpresiones en las que ocurren los operadores *next*, substituyendo en ellas cada señal por su definición y utilizando principalmente la propiedad IFBY para eliminar pares *fby-next*, es posible alcanzar a la especificación ecuacional mostrada en la figura 4.7.

Como puede observarse, el significado de los operadores temporales en una especificación ecuacional planificada es claro desde el punto de vista hardware. Hay un operador *interleave* por operación, de manera que la posición que ésta última ocupa, indica el ciclo en que ha sido planificada. El resto de los argumentos de *interleave* son comodines para indicar la 'disponibilidad' del operador en los demás ciclos. El número de operadores *fby* que aparecen en cada definición indican la vida, medida en ciclos, del valor que se calcula en ella. Por su parte, los operadores *sample* y *replicate*, sobre los puertos de salida y entrada respectivamente, indican la relación entre las frecuencias de muestreo y de reloj del circuito. Dado que no hay ninguna ocurrencia del operador *next* la especificación ecuacional es realizable: bastaría con implementar los operadores *fby* e *interleave* con registros y multiplexores, los operadores no temporales, con módulos combinacionales

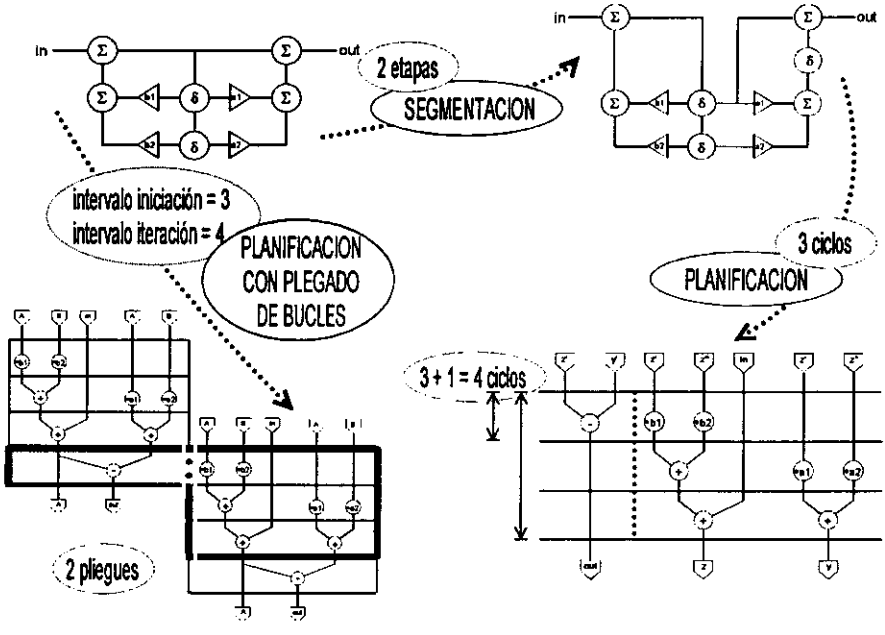


Fig. 4.9: Planificación con plegado de bucles.

número de etapas de la especificación segmentada se corresponde con el número de pliegues de la especificación plegada, y cómo el intervalo de iniciación se corresponde con la latencia de la planificación de la especificación segmentada. Asimismo, el intervalo de iteración vendrá determinado por el ciclo en que se planifique la última operación de la última etapa de la especificación segmentada. Obsérvese además, que la separación entre pliegues es ficticia, esto hace que, aunque se utilicen, sean innecesarias técnicas específicas para planificar operaciones pertenecientes a distintos pliegues o para asignar posteriormente hardware común a dichas operaciones: es posible realizarlo mediante técnicas convencionales.

Concretando: en una planificación realizada mediante técnicas de plegado de bucles que tenga *k* ciclos de intervalo de iteración y *p* ciclos de intervalo de iniciación, se solapan *ceil(k/p)* iniciaciones del algoritmo, por lo que es un problema equivalente a planificar en *p* ciclos una especificación

segmentada en *ceil(k/p)* etapas. O, viéndolo desde el otro punto de vista, la planificación en *n* ciclos de una especificación segmentada en *m* etapas, es equivalente a la planificación mediante técnicas de plegado de bucles de la misma especificación pero sin segmentar con $[1..n]*m$[†] ciclos de intervalo de iteración y con *n* ciclos de intervalo de iniciación.

Una vez conocido cómo se relacionan los parámetros, realizar una planificación con plegado por derivación, se reduce a concatenar las transformaciones mostradas en §3.3.2 y §4.5.3.

4.5.5 Planificación compartida de operaciones en caminos de ejecución mutuamente exclusivos.

La planificación compartida de operaciones en caminos de ejecución mutuamente exclusivos, es una técnica avanzada de planificación que permite planificar en un mismo ciclo y sin recargo en el coste hardware, un conjunto de operaciones que no sean ejecutadas en una misma iniciación del algoritmo. El caso más habitual es la planificación compartida de operaciones presentes en ramas distintas de una misma construcción de selección.

Esta sección muestra cómo pueden reproducirse mediante derivación formal los resultados obtenidos por estas técnicas de planificación. Además se comprobará cómo un único esquema de transformación es lo suficientemente versátil para abordar la solución de problemas de planificación compartida que muchos algoritmos [LiGu98] no son capaces de tratar por las deficiencias de sus representaciones internas.

El esquema de transformación que propondré es similar al utilizado para la planificación con plegado, en lugar de realizar el análisis de exclusión

---

† Donde el valor que se tome desde 1 hasta n, depende del ciclo en donde se planifique la última operación de la última etapa.

mutua a la vez que la planificación, primero se transforma la especificación ecuacional en otra más adecuada cuyas operaciones se correspondan a conjuntos de operaciones mutuamente exclusivas de la especificación inicial, para que, después, ésta pueda ser planificada de modo convencional.

Aunque en el mecanismo de especificación ecuacional no existen construcciones de selección condicional ni de selección múltiple implícitas†, éstas pueden ser declaradas en la signatura como cualquier otro operador no temporal y, como éstos, puede ser manipulada mediante un conjunto de $Lu(\Sigma)$-ecuaciones. Ese conjunto de ecuaciones, como a continuación se mostrará, no requiere ser demasiado grande para permitir la reproducción del tipo de técnicas que trataré en esta sección.

EJEMPLO 4.10

Sea la signatura del ejemplo 2.4 que es capaz de dar soporte al conjunto de los números enteros y al de los booleanos. Para permitir especificar ecuacionalmente conductas que utilicen, por ejemplo, selecciones condicionales y selecciones múltiples de 3 opciones, bastaría con añadir a la dicha signatura un par de símbolos de operación que denotasen dichas funciones sobre cierto género.

sorts
Bool, Ent
operations
...
if □ then □ else □ : Bool, Ent, Ent → Ent
case □ of
□ : □ ;
□ : □ ;
default : □ : Ent, Ent, Ent, Ent, Ent, Ent → Ent

† Es una las consecuencias de la austeridad sintáctica tomada como norma a la hora de definir el mecanismo de especificación (véase §2.4.5). No obstante, es una las extensiones a realizar como trabajo futuro.

Un par de $Lu(\Sigma)$-ecuaciones válidas sobre el modelo 'natural' de la signatura del anterior ejemplo y que permiten la manipulación de la construcción *if-then-else*, son las siguientes:

(*distributividad del operador if respecto al operador +*

$\{ X_{Bool} \equiv \{ c \}, X_{Ent} \equiv \{ x_1, x_2, x_3, x_4 \} \}$, *(análoga a DINT(+), pero sobre if)*

if c then (x_1+x_2) else (x_3+x_4), (if c then x_1 else x_3) + (if c then x_2 else x_4)

)

(*descomposición del operador if*

$\{ X_{Bool} \equiv \{ c \}, X_{Ent} \equiv \{ x, y, x_2, y_2 \} \}$, *(análoga a DET, pero sobre if)*

if c then x else y, if c then (if c then x else x_2) else (if c then y_2 else y)

)

Con ellas es posible reducir el número de operadores de una especificación ecuacional de manera que un único operador pueda realizar varias operaciones mutuamente exclusivas de la especificación ecuacional. La primera, DIF(+), puede usarse para mover la posición de los símbolos *if* dentro de una especificación ecuacional: si se aplica de izquierda a derecha se reducirá el número de operadores suma que aparecen. La segunda, DEIF, se utilizará para reducir selectivamente grupos de operaciones presentes en distintas ramas de condicionales complejos.

EJEMPLO 4.11

Sea una especificación ecuacional, tal que en su cuerpo contenga una definición condicional.

```
body
...
z = if ( a > b ) then ( ( a + b ) + c ) else ( d + e )
```

Como puede observarse en ella (al igual que en la representación gráfica central de la fig. 4.10) la rama *then* (rama izquierda) posee dos operaciones suma cuya ejecución es mutuamente exclusiva con la única operación suma de la rama *else* (rama derecha). Así tenemos dos alternativas: o compartir la

operación más externa de la rama *then* (fig. 4.10-a) o compartir la operación más interna (fig. 4.10-b).

En el primer caso sólo hay que aplicar la propiedad DIF(+):

	body
	...
Expansion(z, 1, x)	**z = t1 + t2**
AplicacionID(z, ε, DIF(+))	**x = a > b**
Expansion(z, 1, t1)	**t1 = if x then (a+b) else d**
Expansion(z, 2, t2)	**t2 = if x then c else e**

Como puede observarse el nuevo sumador realiza las sumas mutuamente exclusivas de la especificación original, una planificación convencional por derivación de esta especificación transformada siguiendo el esquema propuesto en §4.5.3, reproducirá con éxito los resultados de una planificación con análisis de caminos mutuamente exclusivos de la especificación original.

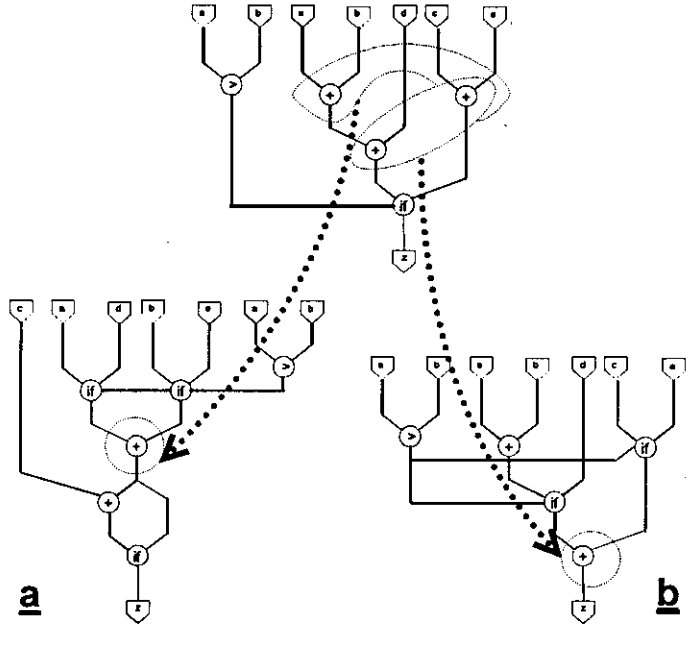


Fig. 4.10: Uso compartido de operaciones.

En el caso de que se desee que se planifiquen compartidamente la operación más interna de la rama *then* con la operación de la rama *else*, será necesario aplicar la ecuación DEIF de izquierda a derecha. Dado que $var(t_R) \neq var(t_L)$, reemplazaremos las variables libres por comodines†.

	body
	...
Expansion(z, 1, x)	z = if x then t1 else t2
AplicacionID(z, ε, DEIF(#, #))	x = a > b
Expansion(z, 2, t1)	t1 = if x then ((a+b)+c) else c+#
Expansion(z, 3, t2)	t2 = if x then # else (d+e)
Reemplazo(t1, 3, c+#)	

A continuación deberán aplicarse la ecuación DIF(+) y reemplazar algunos comodines:

AplicacionID(t1, ε, DIF(+))	
Expansion(t1, 1, t3)	**body**
Reemplazo(t2, 2, a+b)	...
Reemplazo(t3, 3, d+e)	z = if x then t1 else t2
Limpieza(t3, t2)	x = a > b
AplicacionID(t1, 2, if x then y else y = y)	t1 = t2 + c
AplicacionID(t2, ε, DIF(+))	t2 = t3 + t4
Expansion(t2, 2, t3)	t3 = if x then a else d
Expansion(t2, 3, t4)	t4 = if x then b else e

El esquema presentado es también aplicable sobre construcciones condicionales distintas que compartan una misma condición, es decir, sobre operaciones presentes en distintas ramas de ifs distintos. Y también puede aplicarse sobre construcciones condicionales anidadas, e incluso, si existe un número suficiente de ecuaciones, puede serlo sobre operaciones distintas o

† Esto obliga a que cuando se formalice la semántica de if ésta sea parcialmente estricta respecto al símbolo comodín.

sobre construcciones condicionales distintas cuya condición sea opuesta†. Compárese este enfoque tan simple, con los utilizados por los algoritmos de síntesis convencionales.

Para el tratamiento de la selección múltiple puede hacerse un enfoque equivalente o tratarlo como una colección de selecciones condicionales anidadas.

EJEMPLO 4.12

Un ejemplo simple de uso compartido de operaciones en condicionales anidados, puede ser:

```
                          body
                          ...
                          z = if c1 then ( a+b ) else y
                          y = if c2 then ( a+d ) else ( e+f )
```

AplicacionID(y, ε, DIF(+))	
Expansion(y, 1, t1)	body
Expansion(y, 2, t2)	...
Substitucion(z, 3)	z = t3 + t4
Eliminacion(y)	t1 = if c2 then a else e
AplicacionID(z, ε, DIF(+))	t2 = if c2 then d else f
Expansion(z, 1, t3)	t3 = if c1 then a else t1
Expansion(z, 2, t4)	t4 = if c1 then b else t2

4.5.6 Reutilización de recursos.

La *asignación de hardware* consiste en decidir qué módulo hardware del circuito final realiza cada una de las operaciones de la especificación original. Dentro de este proceso suelen distinguirse varios subproblemas: la *selección de recursos*, que determina el tipo de recurso hardware que realizará cierta

† Se tratará en el capítulo 6.

operación, la *asignación de recursos*, que determina el número de cada tipo de recurso de que constará el circuito final, y la *asignación de instancias*, que determina qué operaciones comparten el mismo recurso.

En cualquier caso la asignación de hardware está basada en la reutilización de los operadores presentes en una especificación ecuacional para realizar operaciones que han sido, o serán, planificadas en distintos ciclos. Dado que en el formalismo ecuacional se indicaban mediante comodines los ciclos que los que un operador estaba inactivo, para reutilizar dicho operador en distintos ciclos, bastará con reemplazar cierto comodín por la operación que vaya a reutilizarse.

El esquema general de transformación para reutilizar hardware una vez que se ha realizado la planificación es el siguiente: inicialmente se separan las acciones de cálculo y de almacenamiento de resultados utilizando la propiedad DET y se separan también las acciones de selección de fuentes y de cálculo mediante la propiedad DINT. Una vez separadas las distintas acciones RT implicadas en cada una de las definiciones si se desea reusar operadores, se reemplazan algunos comodines de su definición por señales y se eliminan definiciones redundantes. Si, por el contrario, lo que se desea es reusar registros, primero se reemplazan cadenas de retardos por retardos realimentados (vía la propiedad TMEM) y después se reemplazan comodines por señales y se eliminan definiciones redundantes mediante la regla de limpieza. También es posible reutilizar multiplexores a la vez que eliminar multiplexaciones innecesarias: ambas cosas se consiguen aplicando la propiedad NINT y eliminando definiciones redundantes.

EJEMPLO 4.13

Intentemos reproducir el reuso de un sumador para ejecutar 2 sumas que han sido planificadas respectivamente en los ciclos 2 y 3, tal como indica la fig. 4.11-a.

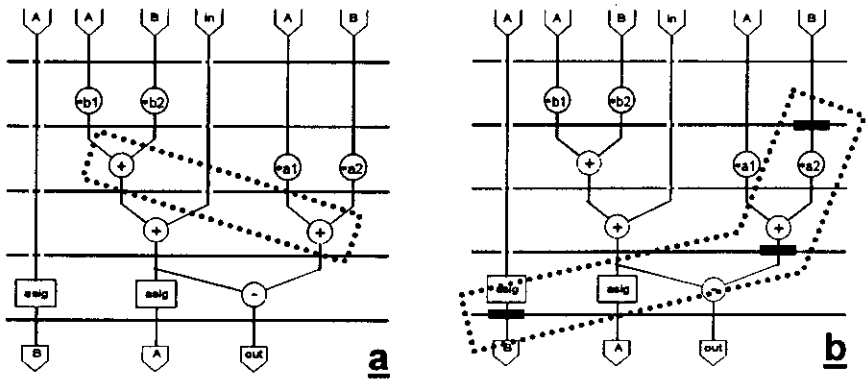


Fig. 4.11: Ejemplos de reutilización de recursos.

Tomaremos como especificación de partida la especificación mostrada por la fig. 4.8 que, como sabemos, fue correctamente planificada en una sección anterior. La primera fase es separar las acciones de cálculo de las de almacenamiento, para ello aplicamos la propiedad DET sobre las definiciones que contienen las operaciones a reusar.

body
...
t1 = # fby (# || # || t2 + t3 || #)
t4 = # fby (# || t5 + t6 || # || #)

	body
	...
AplicacionID(t1, 2, DET(4,3))	t1 = # fby (# || # || t1' || #)
Expansion(t1, 2.3, t1')	t4 = # fby (# || t4' || # || #)
AplicacionID(t4, 2, DET(4,2))	t1' = (# || # || t2 + t3 || #)
Expansion(t4, 2.2, t4')	t4' = (# || t5 + t6 || # || #)

Seguidamente separamos las acciones de selección de operandos y de cálculo, utilizando la ecuación DINT.

```
Reemplazo( t1', 1, #+# )
Reemplazo( t1', 2, #+# )
Reemplazo( t1', 4, #+# )
AplicacionDI( t1', ε, DINT(4,+) )
Reemplazo( t4', 1, #+# )
Reemplazo( t4', 3, #+# )
Reemplazo( t4', 4, #+# )
AplicacionDI( t4', ε, DINT(4,+) )
```

```
body
...
t1 = # fby ( # || # || t1' || # )
t4 = # fby ( # || t4' || # || # )
t1' = ( # || # || t2 || # ) + ( # || # || t3 || # )
t4' = ( # || t5 || # || # ) + ( # || t6 || # || # )
```

Si lo que queremos es reutilizar ambos sumadores, deberemos reemplazar algunos comodines y eliminar el código redundante.

```
Reemplazo( t1', 1.2, t5 )
Reemplazo( t1', 2.2, t6 )
Reemplazo( t4', 1.3, t2 )
Reemplazo( t4', 2.3, t3 )
Limpieza( t4', t1' )
```

```
body
...
t1 = # fby ( # || # || t1' || # )
t4 = # fby ( # || t1' || # || # )
t1' = ( # || t5 || t2 || # ) + ( # || t6 || t3 || # )
```

Reproduzcamos ahora, el reuso de elementos de almacenamiento mostrado en la fig. 4.11-b donde se reutiliza un retardo arquitectónico para almacenar los valores temporales calculados por la suma.

Nuevamente, tomando como especificación inicial la especificación mostrada en la fig. 4.8, volvemos a separar las acciones de cálculo y almacenamiento de las definiciones implicadas que sean operativas y reemplazamos cadenas de retardos por retardos realimentados, vía MEMT, esto es:

```
body
...
t1 = # fby ( # || # || t2 + t3 || # )
B = # fby 0 fby ( # || # || # || A )
```

```
AplicacionID( t1, 2, DET(4,3) )
Expansion( t1, 2.3, t1' )
AplicacionID( B, ε, MEMT( 4, 0, 1) )
```

```
body
...
t1 = # fby ( # || # || t1' || # )
B = 0 fby ( B || # || # || A )
t1' = ( # || # || t2 + t3 || # )
```

Para reutilizar los registros, reemplazamos algunos comodines y eliminamos redundancias.

```
Reemplazo( t1, 1, 0 )
Reemplazo( t1, 2.1, B )
Reemplazo( t1, 2.4, A )
Reemplazo( B, 2.3, t1' )
Limpieza( t1, B )
```

```
body
...
B = 0 fby ( B || # || t1' || A )
t1' = ( # || # || t2 + t3 || # )
```

Obsérvese que el esquema para reproducir una asignación completa de recursos, consistirá en repetir el proceso anteriormente explicado tantas veces como recursos haya. Además nótese cual es el mecanismo que permite detectar si una asignación es o no correcta: el proceso explicado para reproducir un reuso tiende a producir redundancias entre las definiciones implicadas, una vez producidas éstas se eliminan aplicando la regla de limpieza para hacer efectivo el reuso. En el caso de que la asignación sea incorrecta, la redundancia no se produce y, por tanto, la regla de limpieza no es efectiva. De este modo, si se ordena un reuso incorrecto, podrá comprobarse que lo es, si las dos definiciones que debían compartir operador continúan existiendo tras aplicar el esquema de transformación.

En el lado izquierdo de la fig. 4.12 puede verse una especificación ecuacional en la que todos los recursos funcionales y de almacenamiento se han reutilizado correctamente, reproduciendo las asignaciones mostradas en los esquemas que aparecen en el lado derecho. Como puede observarse, para implementar este diseño son necesarios 4 retardadores (recuérdese que 2 ya estaban fijados por la especificación), 2 multiplicadores (que realizan 4 multiplicaciones), 2 sumadores (que realizan 3 sumas) y 1 restador (que realiza la resta).

Hay que destacar que el enfoque propuesto para el reuso es independiente del tipo de planificación que se haya realizado, es decir, si la planificación realizó plegado de bucles o encadenamiento, las ecuaciones que genera son del mismo tipo de las aquí mostradas, esto hace que el esquema de reuso

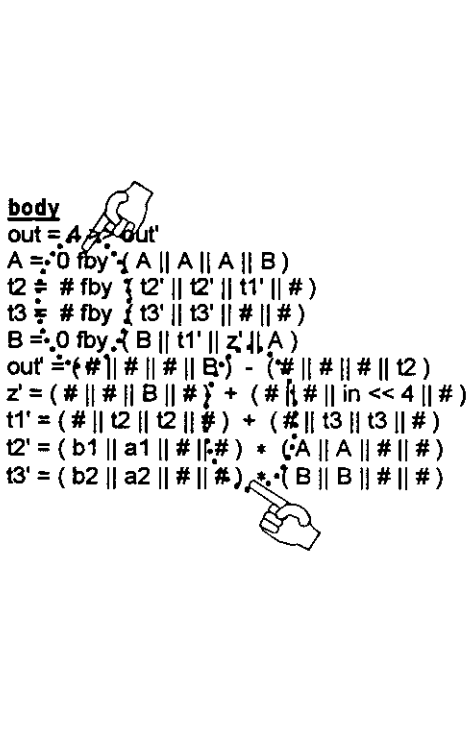
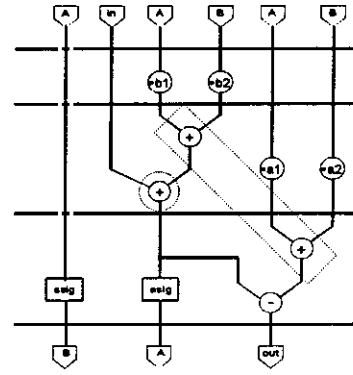


Fig. 4.12: Comportamiento correctamente asignado.

sea también válido para el reuso de componentes en ese tipo de especificaciones. Así, por ejemplo, la fig. 4.13 muestra dos reusos distintos de especificaciones planificadas con encadenamiento, en donde los cada una de las componentes de la cadena se reusan por separado. En un primer caso es la primera suma de la cadena la que se reusa, en un segundo caso es la segunda suma, lo que obligará a insertar multiplexores entre módulos operacionales.

body
out = 3 >> out'
A = 0 fby (A || A || B)
B = 0 fby (B || z' || A)
t2 = # fby (t2' || t2' || #)
t3 = # fby (t3' || t3' || #)
out' = (# || # || B) - (# || # || t7)
z' = (# || t7 || #) + (# || in << 3 || #)
t2' = (b1 || a1 || #) * (A || A || #)
t3' = (b2 || a2 || #) * (B || B || #)
t7 = (# || t2 || t2) + (# || t3 || t3)



body
out = 3 >> out'
A = 0 fby (A || A || B)
B = 0 fby (B || z' || A)
t2 = # fby (t2' || t2' || #)
t3 = # fby (t3' || t3' || #)
out' = (# || # || B) - (# || # || z')
z' = (# || t8 || t2) + (# || in << 4 || t3)
t2' = (b1 || a1 || #) * (A || A || #)
t3' = (b2 || a2 || #) * (B || B || #)
t8 = (# || t2 || #) + (# || t3 || #)

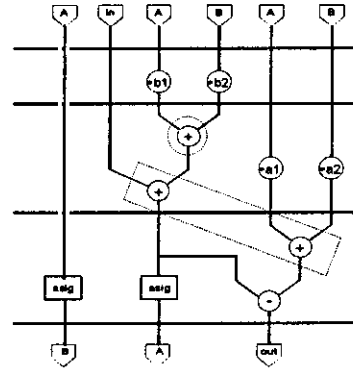


Fig. 4.13: Reuso entre operaciones encadenadas.

4.5.7 Otros aspectos del reuso hardware.

Una vez realizada las fase de planificación y de asignación, puede existir una fase llamada de **alineación de operadores**. El objetivo de este tarea es determinar el orden que toman las fuentes de datos en cada una de los multiplexores que las leen. El objetivo perseguido es reducir el coste de los multiplexores (reusando aquellos que lean de las mismas fuentes en el mismo orden) y reducir el número de secuencias de control distintas (reusando aquellas líneas de control que sigan una misma secuencia)

En el sistema de transformación propuesto la determinación del orden de las entradas de los multiplexores, a la vez que la generación de las secuencias de control que gobiernan dichas multiplexaciones, se realizan vía la aplicación de la propiedad MUXI. Para aplicarla con comodidad deben separarse las selecciones de fuentes en definiciones diferentes mediante la regla de expansión, eliminar redundancias para realizar posibles reusos (según el esquema mostrado en §4.5.6) y aplicar MUXI. Con la aplicación de esta última propiedad se extraen las secuencias de control que, en una última fase, también se pueden reusar.

EJEMPLO 4.14

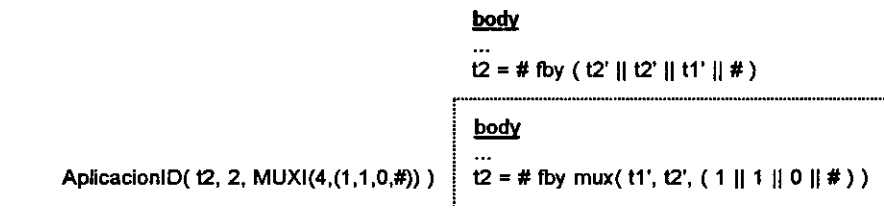
Realicemos algunas manipulaciones sobre la especificación ecuacional mostrada en la fig. 4.12, relacionadas con la técnica de alineación de operadores.

Cuando en distintos instantes temporales se leen distintos valores generados por la misma fuente de datos no es necesario ningún multiplexor, ya que puede reusarse un único camino de transferencia para comunicar en distintos instantes varios datos. La secuencia de derivación para reproducir dicha idea utiliza la regla de reemplazo y la propiedad NINT:

```
                                  body
                                  ...
                                  t3 = # fby ( t3' || t3' || # || # )
                                  ------------------------------------
Reemplazo( t3, 2.3, t3' )         body
Reemplazo( t3, 2.4, t3' )         ...
AplicacionID( t3, 2, NINT(4) )    t3 = # fby t3'
```

Obsérvese, ahora, la definición de *t2*. Tal como indica el operador *interleave*, necesita leer de *t2'* (la salida de uno de los multiplicadores) en los ciclos 1 y 2, mientras que necesita leer de *t1'* (la salida de uno de los sumadores) en el ciclo 3. Al leer solamente dos señales será necesario un multiplexor de 2 entradas para implementar el operador *interleave*, sin

embargo, existen dos alternativas de ordenar dichas señales a las entradas del multiplexor, cada una de las cuales, requerirá distintas secuencias de control. Probemos a alinear las señales en orden inverso al que aparece en el operador *interleave* usando la propiedad MUXI.



Una especificación ecuacional con la fase de alineación de operadores realizada se muestra a continuación. Obsérvense los recursos de multiplexación que necesita 4 multiplexores 2 a 1 y 1 multiplexor 4 a 1.

```
body
out = 4 >> out'
A = 0 fby mux( A, B, c2 )
t2 = # fby mux( t2', t1', c3 )
t3 = # fby t3'
B = 0 fby mux( B, t1', z', A, c4 )
out' = B - t2
z' = B + ( in << 4 )
t1' = t2 + t3
t2' = mux( b1,a1, c5 ) * A
t3' = mux( b2, a2, c1 ) * B
c1 = ( 0 || 1 || # || # )
c2 = ( 0 || 0 || 0 || 1 )
c3 = ( 0 || 0 || 1 || # )
c4 = ( 0 || 1 || 2 || 3 )
c5 = ( 0 || 1 || # || # )
```

Para finalizar obsérvese como pueden eliminarse líneas de control redundantes y cómo pueden reutilizarse aquellas que transporten secuencias de control compatibles.

body
...
c1 = (0 || 1 || # || #)
c2 = (0 || 0 || 0 || 1)
c3 = (0 || 0 || 1 || #)
c4 = (0 || 1 || 2 || 3)
c5 = (0 || 1 || # || #)

	body
Limpieza(c5, c1)	...
Reemplazo(c1, 3, 2)	c1 = (0 || 1 || 2 || 3)
Reemplazo(c1, 4, 3)	c2 = (0 || 0 || 0 || 1)
Limpieza(c4, c1)	c3 = (0 || 0 || 1 || #)

Capítulo 5

Síntesis formal de alto nivel por derivación automática

System development should be
as formal as necessary,
but not more formal.
A. Einstein

El conjunto de todas las posibles especificaciones ecuacionales que describen un mismo algoritmo (véase capítulo 2) o, desde la óptica hardware, el conjunto de todos los circuitos que tienen el mismo comportamiento, junto con el sistema de transformación (véase capítulo 3) y el conjunto de propiedades temporales (véase capítulo 4) forman lo que se denomina un **sistema de reducción**. Sin embargo, este sistema carece del conjunto de propiedades que facilitarían su implantación inmediata en forma de sistema automático de síntesis:

- **No posee formas normales**, es decir, no existen especificaciones ecuacionales que no puedan ser transformadas.
- Es **cíclico**, ya que es posible seguir una secuencia de transformaciones que comience y acabe en la misma especificación ecuacional. Por consiguiente es **no terminante**, por existir cadenas infinitas de derivación.

- No es **confluente**, es decir, si a partir de una única especificación ecuacional se siguen dos caminos de derivación distintos que lleguen a dos especificaciones ecuacionales diferentes, puede que no existan caminos que transformen ambas especificaciones a otra especificación común. Esto hace que tampoco sea **convergente**.

La ausencia de estas propiedades hace que el sistema necesite ser guiado para alcanzar cualquier especificación ecuacional concreta, en particular, aquellas especificaciones que puedan ser consideradas soluciones de cierto proceso de diseño. La aplicación libre (o indeterminista) de las reglas de transformación pasará por muchas descripciones equivalentes pero, en general, sin interés práctico para el diseñador. Por tanto, a la hora de implementar un sistema de derivación automática útil, es necesario estudiar las alternativas de guía.

Para guiar un sistema de transformación caben, al menos, dos alternativas. La primera es la de guiado manual, en la que el usuario decide en cada momento qué regla de transformación disparar[†] y en la que el sistema responde efectuando, si ello tiene sentido, la manipulación simbólica ordenada. Si se le añaden facilidades para la declaración de macros esta alternativa puede llegar a considerarse como parcialmente automática. La segunda alternativa es la de guiado automático, en donde la tarea de disparar cada regla de transformación individual es decidida por un algoritmo y el usuario se limita a marcar las directrices del proceso de derivación, directrices que parametrizan el comportamiento del algoritmo de guía. En esta última es posible concebir un paso más en pos de la automatización: reemplazar al propio usuario por un algoritmo basado en heurísticas que evalúe como parametrizar al algoritmo de guía en cada proceso de diseño.

[†] Recuérdese que no basta con seleccionar la regla de transformación sino que es necesario explicitar el lugar de la especificación ecuacional en donde aplicarla e incluso, si la regla lo requiere, concretar que $Lu(\Sigma)$-ecuación debe utilizarse.

Este capítulo está dedicado a especificar un algoritmo de guiado automático para realizar síntesis formal de alto nivel por derivación. Para obtenerlo se estudiará primero un sistema con guiado manual (§5.1) y de las conclusiones de dicho estudio, podrá formularse el algoritmo buscado (§5.2).

5.1 Un sistema de derivación con guiado manual.

Los sistemas de derivación con guiado manual o con guiado parcialmente automático, aunque puedan resultar extraños dentro del marco de la SAN, no lo son en otros ámbitos de aplicación. Pueden encontrarse ejemplos en el campo de la demostración automática de teoremas [StGG92] en el campo de la programación automática [HoKr93] e incluso en el campo del diseño formal de hardware (véase §1.1). La simplicidad de su implementación, una vez resueltos todos los problemas teóricos, hace que sea una alternativa rápida para obtener un sistema real sobre el que evaluar cómodamente el alcance y viabilidad de un enfoque formal.

Un sistema de este tipo es el que se ha venido usando implícitamente en las secciones §3.3 y §4.5 para demostrar la versatilidad tanto del mecanismo de especificación como de las reglas de transformación. Su arquitectura es simple y algunas de sus componentes han sido también implícitamente mostradas en §3.1.2, §3.1.3 y §4.4 y evaluadas en complejidad en §3.2.4 y §4.4.1. Básicamente pueden distinguirse cuatro módulos operativos en este tipo de sistema:

- **Cargador**†: lee una especificación ecuacional chequeando su corrección sintáctica y semántica.

† Nótese que es este módulo es considerablemente más simple que el compilador de un sistema de síntesis de alto nivel, que no sólo chequea y carga, sino que también debe crear una representación interna que verifica un modelo computacional distinto del que informalmente asume la especificación procedural de partida.

- **Núcleo de transformación**: implementa cada una de las reglas de transformación presentadas en §3.1.2 y §3.1.3.
- **Generador de ecuaciones temporales**: genera el conjunto de $Lu(\Sigma)$-ecuaciones correspondientes a las propiedades temporales tal y como se ha descrito en §4.4.
- **Interfaz de usuario**: acepta las órdenes de transformación que el diseñador dicta en un formato textual que puede ser equivalente al utilizado en §3.3 y §4.5.

Respetando esta arquitectura desarrollé un prototipo† que sirvió para comprobar efectivamente qué tipo de conductas eran especificables, cuántas técnicas de diseño eran aplicables y qué proporción del espacio de diseño era alcanzable mediante el enfoque transformativo. Así, gracias a él, se han podido efectuar cada uno de los ejemplos mostrados en los capítulos anteriores.

Sin embargo, la utilidad de este prototipo ha sido más amplia ya que ha permitido extraer, observando el uso que de él se hacía, algunas consecuencias sobre el modo natural de realizar procesos de síntesis formal. Consecuencias que han resultado muy valiosas a la hora de desarrollar el sistema de guiado automático que será presentado en la próxima sección. Las consecuencias principales han sido las siguientes:

- El nivel de abstracción de las reglas de transformación es, en general, excesivamente bajo para que cada una de sus aplicaciones individuales sean disparadas manualmente. Esto es así ya que el número de transformaciones a realizar en un proceso de diseño es desproporcionado en relación a las decisiones típicas que acostumbra a tomar un diseñador. Sirva de ejemplo que, para realizar cualquier

† Utilizando PROLOG y con un coste de programación aproximado de 1 hombre/semana y un tamaño aproximado de 80 predicados y 650 líneas de código (repartidas en generador de ecuaciones: 38%, núcleo de transformación: 32% y resto: 30%).

planificación del filtro recursivo de segundo orden mostrado en anteriores ejemplos, deben realizarse alrededor de 300 transformaciones de las cuales sólo 10 afectan realmente al rendimiento del circuito final (aquellas que asignan un ciclo a cada una de las 10 operaciones de la especificación).

- Se observó que el diseñador para hacer frente a tal número de posibilidades y tras alcanzar cierta pericia en manejar el sistema, tendía a agrupar las transformaciones de manera que, mientras que el orden de uso de los grupos tenía una influencia notable sobre el resultado obtenido, el orden de aplicación de las reglas de transformación individuales pertenecientes a cada grupo, era totalmente irrelevante.
- Se notó además, que muchos de los grupos de transformaciones anteriormente referidos tenían un claro significado hardware, y era este significado el que permitía al diseñador decidir el orden de aplicación de los mismos.
- Finalmente, pudo también observarse que los grupos de transformaciones eran independientes de la especificación ecuacional inicial, pero eran fuertemente dependientes de la técnica de diseño que intentaran reproducir.

Tras este trabajo de campo, y basándome en la idea de agrupar transformaciones, me propuse desarrollar un prototipo de algoritmo de guía para síntesis de alto nivel de manera que, sin que realizase exploración en el espacio de soluciones, permitiera reproducir por derivación cualquier proceso de diseño realizado externamente.

5.2 Un sistema de derivación automático.

El propósito de esta sección es presentar un sistema de derivación que sea capaz de realizar síntesis de alto nivel y que tenga como únicos puntos de

control las decisiones típicas de un proceso de diseño. Así este sistema, aplicando únicamente el conjunto de reglas de transformación mostradas en el capítulo 3 (para asegurar la corrección), será capaz de transformar por derivación una especificación ecuacional que describa una computación a nivel algorítmico en otra especificación ecuacional que describa un circuito RT capaz de implementarla, tomando como entrada la especificación de partida e información sobre la planificación y la asignación de hardware realizada†.

La arquitectura de este sistema es una ampliación de la arquitectura mostrada en la anterior sección. De hecho a los módulos *cargador*, *núcleo de transformación*, *generador de ecuaciones temporales* e *interfaz de usuario*, sólo hay que añadir el módulo **controlador** que, a partir de la información de un proceso de síntesis particular, dirige el disparo de las reglas de transformación individuales.

Al estudio de los módulos transformador y generador ya se han dedicado los dos capítulos anteriores, a continuación estudiaré el algoritmo de guiado.

5.2.1 Un algoritmo para el guiado automático de un proceso de síntesis formal de alto nivel por derivación.

Como se ha dicho anteriormente una de las entradas del algoritmo es un informe sobre los resultados de un proceso concreto de síntesis de alto nivel. Para formalizar compactamente dichos resultados es necesario llegar a los convenios que seguidamente se detallan.

Descripción de los resultados de un proceso de síntesis de alto nivel.

---

† Recuérdese que el objeto de esta investigación no es realizar una herramienta de síntesis capaz de tomar sus propias decisiones, sino desarrollar una herramienta capaz de reproducir formalmente un proceso de diseño ya efectuado (o que se está efectuando), para asegurar matemáticamente si es correcto.

Para caracterizar el resultado de un proceso de síntesis de alto nivel realizado sobre cierta especificación ecuacional de partida $ds \equiv (\Sigma, X, I, O, \varphi)$, utilizaré los siguientes conjuntos:

- Ops_{FU}: cada una de las ocurrencias de un símbolo de operación perteneciente a la signatura Σ y de aridad distinta de 0 dentro del cuerpo ecuacional φ. En un enfoque convencional este conjunto es equivalente al conjunto de vértices operativos del grafo de dependencias obtenido por compilación de cierta especificación. Para distinguir cada una de dichas ocurrencias puede utilizarse un par señal-posición, (x, u), donde $x \in X$ y $u \in pos(\varphi(x))$. Sin embargo, por conveniencia y a costa de ambigüedad, a veces utilizaré el propio símbolo de operación para referirme a una de sus ocurrencias dentro del cuerpo ecuacional. A cada uno de los elementos de Ops_{FU} se le llamará **operación funcional**. Obsérvese que las operaciones de selección condicional o de selección múltiple se engloban dentro de esta categoría.
- Ops_{CTE}: cada una de las ocurrencias de un símbolo de operación perteneciente a la signatura Σ y de aridad 0 dentro del cuerpo ecuacional φ. En un enfoque convencional este conjunto es equivalente al conjunto de vértices constantes del grafo de dependencias. Para distinguir cada ocurrencia utilizaré el mismo convenio que en Ops_{FU}. A cada una de sus componentes se le llamará **constante**.
- Ops_{ST}: cada una de las ocurrencias del símbolo de operación *fby* dentro del cuerpo ecuacional φ. En un enfoque convencional este conjunto es equivalente al conjunto de vértices de asignación de pre-registros del grafo de dependencias e incluso puede ser equivalente al de nodos que marcan el comienzo de bucle en especificaciones iterativas (véase §2.4.5). Nuevamente utilizaré el mismo convenio que en Ops_{FU} para distinguir cada ocurrencia individual. A cada una de sus componentes se le llamará **operación de retardo arquitectónico**.

- $Recs_{FU}$: conjunto de instancias funcionales del circuito sintetizado a nivel RT. Para identificar cada una de las componentes de este conjunto puede usarse cualquier método. Sin embargo, dado que todo circuito puede ser descrito por cierta especificación ecuacional $ds' \equiv (\Sigma, X', I, O, \varphi')$, es posible utilizar sin pérdida de generalidad pares (x, u) pertenecientes a ds', esto es $x \in X'$ y $u \in pos(\varphi(x))$. Obsérvese que de esta manera este conjunto viene a especificarse de una forma equivalente a Ops_{FU}, la diferencia radica en que las señales pertenecen a especificaciones ecuacionales distintas a distinto nivel de abstracción.
- $Recs_{ST}$: conjunto de instancias de retardador del circuito sintetizado a nivel RT. Podrá usarse también un par señal-posición para distinguir cada una de ellas, dada su correspondencia con ocurrencias de operadores *fby* en la especificación ecuacional que describa dicho circuito de nivel RT.
- $Recs_{MUX}$: conjunto de instancias de multiplexor del circuito sintetizado a nivel RT. Podrá usarse el convenio de notación acostumbrado.

Así, el resultado de cada una de las fases de un proceso de SAN, puede formalizarse como una aplicación entre algunos de los anteriores conjuntos:

- Planificación de operaciones y planificación del ciclo de actualización de los retardos arquitectónicos (donde λ es la latencia de la planificación):

$$\tau : Ops_{FU} \cup Ops_{ST} \rightarrow [1..\lambda]$$

- Asignación de recursos funcionales:

$$\alpha_{FU} : Ops_{FU} \rightarrow Recs_{FU}$$

- Asignación de recursos de almacenamiento†:

† Obsérvese que se requieren recursos de almacenamiento tanto para almacenar valores arquitectónicos como para almacenar los valores generados por cada una de las operaciones hasta el momento de su uso. Nótese que esta aplicación es parcial ya que pueden existir algunas operaciones que no necesiten que sus resultados se almacenen (por ejemplo, operaciones encadenadas). Además se asumen, por simplicidad y no por limitación del enfoque ecuacional, dos aspectos obvios que todo

$$\alpha_{ST} : Ops_{FU} \cup Ops_{ST} \rightarrow Recs_{ST}$$

- Asignación de recursos de encaminamiento†:

$$\alpha_{MUX} : (Ops_{FU} \times \mathbf{N}_+) \cup Ops_{ST} \cup Ops_{FU} \rightarrow Recs_{MUX}$$

- Alineamiento de operaciones††:

$$\alpha_{ALGN} : (Ops_{FU} \times \mathbf{N}_+) \cup Ops_{ST} \cup Ops_{FU} \rightarrow \mathbf{N}$$

EJEMPLO 5.1

Sea el cuerpo de una especificación ecuacional del filtro recursivo de segundo orden expandido y simplificado (lo que posteriormente llamaremos *normalizado*):

```
body
out = z - t1
t1 = t2 + t3
t2 = a1 * t12
t3 = a2 * t11
```

algoritmo razonable de síntesis debe adoptar:

- Distintas transferencias del mismo valor a distintos consumidores, comparten elemento de almacenamiento (por ello sólo se asocia un elemento de almacenamiento por operación).
- Cuando un valor arquitectónico debe almacenarse durante un número de ciclos mayor que la latencia, deberá implementarse con varios elementos de almacenamiento de los cuales sólo uno podrá reusarse, es ése el que fija esta aplicación que por ello es una aplicación que no será sobreyectiva.

† Nótense los siguientes aspectos (cada uno relacionado con un conjunto de los que forman el dominio de esta aplicación):

- que cada uno de los argumentos de una operación funcional, puede leerse a través de uno de los posibles multiplexores conectados a cada uno de los puertos de entrada del recurso funcional que implemente dicha operación.
- que el valor que debe actualizar un retardo arquitectónico, puede leerse también a través del posible multiplexor conectado al puerto de entrada del retardador que implemente dicha operación de almacenamiento.
- que la salida de una operación cualquiera (si necesita almacenarse temporalmente), podrá estar asociada a un multiplexor conectado al puerto de entrada del registro que almecene temporalmente el valor calculado.
- que implícitamente las operaciones que comparten recursos funcionales o de retardo comparten multiplexores.

†† Esta aplicación complementa a la anterior asignando a cada transferencia de información que atraviese un multiplexor, un puerto concreto por donde hacerlo.

```
z = t7 + in
t7 = t8 + t9
t8 = b1 * t12
t9 = b2 * t11
t11 = 0 fby t12
t12 = 0 fby z
```

Sea también la especificación ecuacional obtenida a partir de la anterior, tras un proceso concreto de SAN (cuyas principales decisiones de diseño se muestran gráficamente en las fig. 5.1 y fig. 5.2):

```
body
out = 5 >> t48
t48 = t50 - t47
t47 = mux( t39, t34, t67 ) + mux( in << 5, t34, t27, t68 )
t44 = mux( b2, b1, a2, a1, t65 ) * mux( t27, t50, t67 )
t50 = 0 fby mux( t50, t47, t64 )
t34 = # fby mux( t34, t44, t47, t63 )
t39 = # fby mux( t44, t50, t64 )
t27 = 0 fby mux( t27, t44, t39, t68 )
t63 = ( 1, 0, 2, 1, # )
t65 = ( 0, 1, 2, 3, # )
t68 = ( 0, 0, 1, 0, 2 )
t64 = ( 0, 0, 0, 1, 0 )
t67 = ( 0, 1, 0, 1, 1 )
```

Para dichas especificaciones los conjuntos anteriormente definidos son:

- Ops_{FU} = { (out,ε), (t1,ε), (t2,ε), (t3,ε), (z,ε), (t7,ε), (t8,ε), (t9,ε) }
- Ops_{ST} = { (t11,ε), (t12,ε) }
- Ops_{CTE} = { (t2,1), (t3,1), (t8,1), (t9,1) }
- $Recs_{FU}$ = { (t48,ε), (t47,ε), (t44,ε) }
- $Recs_{ST}$ = { (t50,ε), (t34,ε), (t39,ε), (t27,ε) }
- $Recs_{MUX}$ = { (t47,1), (t47,2), (t44,1), (t44,2), (t50,2), (t34,2), (t39,2), (t27,2) }

Describamos el proceso de SAN llevado a cabo mediante aplicaciones entre los anteriores conjuntos:

- Planificación (véase fig. 5.1):

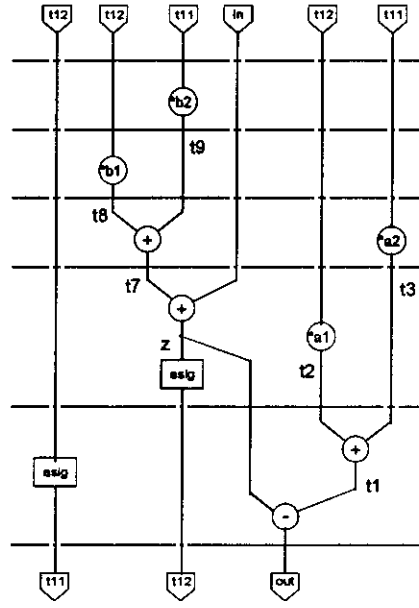


Fig. 5.1: Planificación.

$\tau : Ops_{FU} \cup Ops_{ST} \rightarrow [1..5]$

τ((out,ε))	= 5	τ((t7,ε))	= 3
τ((t1,ε))	= 5	τ((t8,ε))	= 2
τ((t2,ε))	= 4	τ((t9,ε))	= 1
τ((t3,ε))	= 3	τ((t12,ε))	= 4
τ((z,ε))	= 4	τ((t11,ε))	= 5

- Asignación de recursos funcionales (véase fig. 5.2-a, donde las operaciones que comparten una misma unidad funcional están rodeadas por una misma línea):

$\alpha_{FU} : Ops_{FU} \rightarrow Recs_{FU}$

α_{FU}((out,ε))	= (t48,ε)	α_{FU}((z,ε))	= (t47,ε)
α_{FU}((t1,ε))	= (t47,ε)	α_{FU}((t7,ε))	= (t47,ε)
α_{FU}((t2,ε))	= (t44,ε)	α_{FU}((t8,ε))	= (t44,ε)
α_{FU}((t3,ε))	= (t44,ε)	α_{FU}((t9,ε))	= (t44,ε)

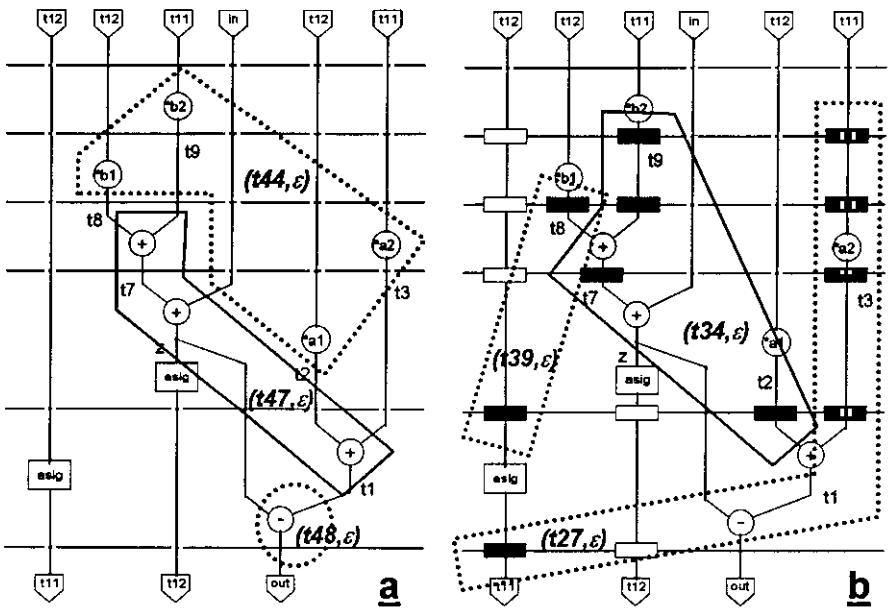


Fig. 5.2: Reuso.

- Asignación de recursos de almacenamiento (véase fig. 5.2-b, en donde mediante rectángulos negros se marcan cada uno de los valores distintos que requieren almacenarse entre ciclos y que pueden reutilizar elementos de memoria y con rectángulos blancos aquellos que requieren un elemento de memoria dedicado†):

† Obsérvese que la actualización del retardo arquitectónico *f12* se realiza en el ciclo 4 y la última lectura del valor que memoriza se ha planificado en el ciclo 5 (de la siguiente iniciación). Esto obliga a que dicho valor deba permanecer almacenado durante 6 ciclos y que por tanto sean necesarios dos registros encadenados: uno con todos sus ciclos ocupados (y que no puede reutilizarse) y otro con un único ciclo ocupado (que si que puede serlo). Por ello en la asignación de recursos de almacenamiento sólo se explicitará el segundo de ellos, que en este caso, se reutiliza para almacenar el resultado de *f8*.

$\alpha_{ST} : Ops_{FU} \cup Ops_{ST} \rightarrow Recs_{ST}$

α_{ST}((out,ε))	= ×××†	α_{ST}((t7,ε))	= (t34,ε)
α_{ST}((t1,ε))	= ×××††	α_{ST}((t8,ε))	= (t39,ε)
α_{ST}((t2,ε))	= (t34,ε)	α_{ST}((t9,ε))	= (t34,ε)
α_{ST}((t3,ε))	= (t27,ε)	α_{ST}((t11,ε))	= (t27,ε)
α_{ST}((z,ε))	= ×××†††	α_{ST}((t12,ε))	= (t39,ε)

- Asignación de recursos de encaminamiento (véase la especificación ecuacional sintetizada):

$\alpha_{MUX} : (Ops_{FU} \times \mathbf{N}_+) \cup Ops_{FU} \cup Ops_{ST} \rightarrow Recs_{MUX}$

α_{MUX}((out,ε), 1)	= ×××	α_{MUX}((t11,ε))	= (t27,1)
α_{MUX}((out,ε), 2)	= ×××	α_{MUX}((t12,ε))	= (t39,1)
α_{MUX}((t1,ε), 1)	= (t47,1)	α_{MUX}((out,ε))	= ×××
α_{MUX}((t1,ε), 2)	= (t47,2)	α_{MUX}((t1,ε))	= ×××
α_{MUX}((t2,ε), 1)	= (t44,1)	α_{MUX}((t2,ε))	= (t34,1)
α_{MUX}((t2,ε), 2)	= (t44,2)	α_{MUX}((t3,ε))	= (t27,1)
α_{MUX}((t3,ε), 1)	= (t44,1)	α_{MUX}((z,ε))	= ×××
α_{MUX}((t3,ε), 2)	= (t44,2)	α_{MUX}((t7,ε))	= (t34,1)
α_{MUX}((z,ε), 1)	= (t47,1)	α_{MUX}((t8,ε))	= (t39,1)
α_{MUX}((z,ε), 2)	= (t47,2)	α_{MUX}((t9,ε))	= (t34,1)
α_{MUX}((t7,ε), 1)	= (t47,1)		
α_{MUX}((t7z,ε), 2)	= (t47,2)		
α_{MUX}((t8,ε), 1)	= (t44,1)		
α_{MUX}((t8,ε), 2)	= (t44,2)		
α_{MUX}((t9,ε), 1)	= (t44,1)		
α_{MUX}((t9,ε), 2)	= (t44,2)		

- Alineamiento de operaciones (véase la especificación ecuacional sintetizada):

---

† La resta no necesita registro temporal dado que está conectada a un puerto de salida que, según el modelo establecido en el capítulo 2, es leído en el último ciclo.

†† Al estar la suma encadenada con la resta no necesita registro para almacenar el valor que calcula.

††† Asumo que la herramienta de SAN es lo suficientemente inteligente como para reusar el primero de los registros que implementa el retardo arquitectónico $t12$ (el que tiene todos sus ciclos ocupados).

$$\alpha_{ALGN} : (Ops_{FU} \times \mathbf{N}_+) \cup Ops_{ST} \cup Ops_{FU} \rightharpoonup \mathbf{N}$$

α_{ALGN}((out,ε), 1)	= ×××	*las entradas del restador no tienen mux*
α_{ALGN}((out,ε), 2)	= ×××	*las entradas del restador no tienen mux*
α_{ALGN}((t1,ε), 1)	= 1	*t2 entra por el puerto 1 del mux (t47,1)*
α_{ALGN}((t1,ε), 2)	= 2	*t3 entra por el puerto 2 del mux (t47,2)*
α_{ALGN}((t2,ε), 1)	= 3	*a1 entra por el puerto 3 del mux (t44,1)*
α_{ALGN}((t2,ε), 2)	= 1	*t12 entra por el puerto 1 del mux (t44,2)*
α_{ALGN}((t3,ε), 1)	= 2	*a2 entra por el puerto 2 del mux (t44,1)*
α_{ALGN}((t3,ε), 2)	= 0	*t11 entra por el puerto 0 del mux (t44,2)*
α_{ALGN}((z,ε), 1)	= 1	*t7 entra por el puerto 1 del mux (t47,1)*
α_{ALGN}((z,ε), 2)	= 0	*in entra por el puerto 0 del mux (t47,2)*
α_{ALGN}((t7,ε), 1)	= 0	*t8 entra por el puerto 0 del mux (t47,1)*
α_{ALGN}((t7,ε), 2)	= 1	*t9 entra por el puerto 1 del mux (t47,2)*
α_{ALGN}((t8,ε), 1)	= 1	*b1 entra por el puerto 1 del mux (t44,1)*
α_{ALGN}((t8,ε), 2)	= 1	*t12 entra por el puerto 1 del mux (t44,2)*
α_{ALGN}((t9,ε), 1)	= 0	*b2 entra por el puerto 0 del mux (t44,1)*
α_{ALGN}((t9,ε), 2)	= 0	*t11 entra por el puerto 0 del mux (t44,2)*
α_{ALGN}((t11,ε))	= 3	*la act. entra por el puerto 3 del mux (t27,1)*
α_{ALGN}((t12,ε))	= 1	*la act. entra por el puerto 1 del mux (t39,1)*
α_{ALGN}((out,ε))	= ×××	*z-t1 no se memoriza*
α_{ALGN}((t1,ε))	= ×××	*t2+t3 no se memoriza*
α_{ALGN}((t2,ε))	= 1	*a1 ∗t12 entra por el puerto 1 del mux (t34,1)*
α_{ALGN}((t3,ε))	= 1	*a2 ∗t11 entra por el puerto 1 del mux (t27,1)*
α_{ALGN}((z,ε))	= ×××	*t7+in reusa implícitamente un registro*
α_{ALGN}((t7,ε))	= 2	*t8+t9 entra por el puerto 2 del mux (t34,1)*
α_{ALGN}((t8,ε))	= 0	*b1 ∗t12 entra por el puerto 0 del mux (t39,1)*
α_{ALGN}((t9,ε))	= 1	*b2 ∗t11 entra por el puerto 1 del mux (t34,1)*

Obsérvese que lo importante de estas aplicaciones no es su definición concreta en base a posiciones de ciertas especificaciones ecuacionales sino su capacidad expresiva de definir agrupaciones entre operaciones de la especificación original.

Esquema general del algoritmo de guía.

El algoritmo que a continuación se propone toma como entrada una especificación ecuacional y un conjunto de aplicaciones que describen unívocamente un proceso de SAN concreto. El resultado será un informe determinando si el proceso se SAN es correcto o no, y una especificación ecuacional conductualmente equivalente a la de partida† que, en caso de que el proceso de SAN sea correcto, describirá un circuito a nivel RT y, en caso de no serlo, permitirá localizar los errores de dicho proceso.

En su especificación se usarán las conclusiones establecidas en §5.1. Así, cada uno de los grupos de transformaciones que podían distinguirse en todo proceso de derivación manual, se presentará como una fase del algoritmo de guía, de manera que cada una de las fases tengan que realizarse en orden. Los límites de dichas fases se han elegido con el objetivo tanto de mostrar tales grupos, como de que las fases posean un intuitivo significado hardware.

Por otro lado, y atendiendo a otra de las conclusiones, cada una de las fases será especificada como un conjunto de reducciones (formuladas cada una de ellas como composición de las reglas de transformación del capítulo 3) tal que puedan aplicarse de forma indeterminista, es decir en cualquier orden y tantas veces como veces se cumplan sus condiciones de disparo. La razón de hacerlo así es doble: por un lado reflejar la irrelevancia que supone el orden de aplicación de las reglas de transformación a la hora de conseguir cierto objetivo y, por otro, presentar una solución lo más abstracta posible que no adopte innecesariamente un paradigma de programación concreto, con ello no se limita el número de implementaciones válidas de este algoritmo, de

† Recuérdese que el formalismo ecuacional es capaz de representar comportamientos erróneos (aunque conductualmente equivalentes a otras especificaciones correctas) mediante definiciones no implementables.

manera que pueda ser desarrollado fácilmente usando cualquier tipo de lenguaje de programación: procedural, no procedural, serie o paralelo.

El esquema del algoritmo de guía (en el que solo se detallan las fases que deben realizarse secuencialmente) es:

sintetizar($(\Sigma, X, I, O, \varphi)$, λ, τ, α)
inicio
  normalización
  detección de especificaciones no soportadas
  multiplexación de fuentes
  planificación
  separación de acciones RT
  eliminación del predictor *next*
  chequeo de la corrección de la planificación
  realimentación de retardos
  reuso implícito de retardadores
  aplanado
  reuso de recursos
  chequeo de la corrección de la asignación
  síntesis del encaminamiento
  reuso de líneas de control
fin

A continuación, en distintas secciones, se especificará la funcionalidad de cada fase como un conjunto de reducciones, cada uno de los cuales deberá ser interpretado como sigue. Una misma reducción que pueda aplicarse en distintos lugares de una especificación ecuacional puede considerarse como caminos de computación distintos e independientes. Siempre que exista más de una reducción que sea aplicable, el orden de aplicación es indiferente y si el resultado de efectuar una reducción desencadena que un mayor número de reducciones sea aplicable, el orden vuelve a ser indistinto entre antiguas y nuevas reducciones. Finalmente, cuando ninguna reducción pueda aplicarse o alguna devuelva FALLO, se considerará que la fase ha terminado y en este segundo caso (cuando devuelva FALLO) también termina el algoritmo completo.

En la ejecución de las reducciones, recuérdese el comentario realizado en §3.2 sobre el comportamiento de las reglas de transformación: **la aplicación de una regla cuando no se cumplen las precondiciones, no tiene efecto alguno sobre la especificación ecuacional** que pretende transformar.

El criterio utilizado tanto para seleccionar el número de reducciones que forman cada fase, como el número de reglas de transformación que se disparan en cada reducción ha sido en cierto modo discrecional. Se ha preferido primar, frente a otros aspectos, la simplicidad del enunciado de las reducciones para que puedan tener un significado hardware claro, ya que lo que se intenta plasmar en esta sección es ante todo una idea en lugar de la mejor implementación de esa idea: seguramente, a costa de perder la intuición, es posible especificar sistemas de reducción mucho más efectivos pero, quizás, solamente comprensibles por su creador.

En cualquier caso, desde el punto de vista teórico, los sistemas de reducción que se especifican en cada una de las fases del algoritmo poseen todas las características que no poseía el sistema de transformación general: son confluentes, es decir, fuertemente terminantes (no existen cadenas infinitas de reducción) y convergentes (toda divergencia confluye y por tanto existe una única forma normal). Así, las definiciones que forman cualquier especificación ecuacional resultado de cualquiera de las fases tienen una forma sintáctica determinada que, como comprobaremos, también posee un claro significado hardware.

En cuanto a la notación usada, se utilizará la variable x, con o sin sub/superíndices para referenciar señales de la especificación ecuacional, c para referenciar constantes, σ para referenciar operaciones funcionales, t para referenciar cualquier $Lu(\Sigma)$-término y p para referenciar puertos.

Por último, destacar que tras el enunciado de cada una de las fases se muestran y discuten las formas normales que se obtienen cuando la fase finaliza. El objetivo perseguido con ello es doble. Por un lado, mostrar el

significado hardware de las definiciones que aparecen durante un proceso de diseño, lo que ayudará a completar la comprensión de las reducciones. Y por otro lado, establecer las bases para el estudio teórico de complejidad que se realizará en §5.2.2.

Normalización.

Su objetivo es restringir la forma que pueden adoptar las definiciones del cuerpo de una especificación ecuacional para simplificar la descripción del resto de las fases. De este modo, transforma una especificación ecuacional en otra caracterizada por:

- Toda definición solo posee un símbolo de operación.
- No existen señales de paso, es decir, señales cuya definición sea simplemente otra señal o una constante.
- No existen definiciones redundantes, de este modo el cuerpo ecuacional es mínimo en símbolos de operación por no existir subexpresiones comunes†.
- No existe código muerto.

Aplanado

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1) \})}{\text{Expansion}(x_1, u, x_2)} \quad u \notin pos_0(\varphi(x_1)) \cup \{\varepsilon\} \wedge x_2 \notin X \cup Lu(\Sigma)$$

† Nótese que esto ya puede considerarse una optimización que, en las herramientas convencionales, suele ser realizada por el compilador junto a otras tales como la propagación de constantes o la reducción de profundidad de los árboles de operaciones. No obstante, si se quisiera reproducir un proceso de diseño que no la realizase bastaría con quitar de esta fase la reducción de *eliminación de definiciones redundantes.*

Eliminación de definiciones de paso†

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1), x_2 = t \})}{\begin{array}{c}\text{Substitucion}(x_1, u) \\ \text{Eliminacion}(x_2)\end{array}} \quad \exists u \in pos_0(\varphi(x_1)),\ \varphi(x_1)[u] \equiv x_2 \wedge t \in X \cup \Sigma_{\varepsilon,s}$$

Eliminación de definiciones muertas

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1) \})}{\text{Eliminacion}(x_1)} \quad \begin{array}{l} x_1 \notin O \wedge \forall x \in X - I - \{x_1\},\ \forall u \in pos_0(\varphi(x)), \\ \varphi(x)[u] \neq x_1 \end{array}$$

Eliminación de definiciones redundantes

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1), x_2 = \varphi(x_2) \})}{\text{LimpiezaRec}(x_2, x_1)} \quad x_2 \notin O \wedge \varphi(x_1) \equiv \varphi(x_2)$$

Obsérvese que éste sistema de reducción, aunque la aplicación de las reducciones sea indeterminista, termina, es decir, que llega un momento en que ninguna reducción es aplicable. Esto es así, ya que la aplicación de la *eliminación de definiciones redundantes*, de la *eliminación de definiciones muertas* y de la *eliminación de definiciones de paso* reducen el tamaño de la especificación ecuacional y el *aplanado* se aplica como máximo un número de veces igual al de operaciones de la especificación original del sistema.

EJEMPLO 5.2

La primera especificación ecuacional del ejemplo 5.1. es una especificación normalizada que puede obtenerse, por ejemplo, de la siguiente (ya mostrada en el ejemplo 2.13):

† Obsérvese que puede dejar una señal muerta (la definición de x_2). Así, por si acaso, se aplica la regla de eliminación que la eliminará si efectivamente lo es.

body
out = z - (a1 *(0 fby z) + a2 *(0 fby 0 fby z))
z = (b1 *(0 fby z) + b2 *(0 fby 0 fby z)) + in

Como puede observarse, para normalizar esta especificación sólo se disparan 2 de la reducciones que componen esta fase: la de *aplanado* que lo hace 12 veces y la de *eliminación de definiciones redundantes* que, dependiendo del orden de disparo, lo hace 3 o 4 veces.

Detección de especificaciones no soportadas.

Evita la síntesis de especificaciones que contengan características no soportadas por la versión actual de este algoritmo de guiado. Las especificaciones ecuacionales que permite pasar (que no disparan ninguna de las siguientes reducciones que finalizan todas en FALLO) se caracterizan por:

- El único operador temporal que aparece en el cuerpo ecuacional es *fby*, con lo que se asegura que la especificación de partida sea de nivel algorítmico.
- El primer argumento de todo operador *fby* es una constante, con esto se asegura que dicho argumento tenga un claro significado hardware (el valor inicial fijo que adopta un elemento de memoria tras un *reset* asíncrono)†. Si este argumento pudiera ser un valor variable sería difícil de hacer una interpretación hardware simple del término.
- Que ningún puerto ni constante esté retrasado. La razón es simplificar la descripción del algoritmo de guía prohibiendo aspectos de la especificación que no pueden optimizarse mediante un proceso de SAN.

† Nótese que cualquier término cerrado sería igualmente válido, se ha restringido a que sea constante por simplificar la especificación del sistema de reducción.

Obsérvese que estos retrasos pueden ser fácilmente implementados externamente al diseño y si se permitiera su integración dentro del sistema, requerirían de un retardador que estaría ocupado durante toda la latencia de la planificación, por lo que no podría ser reusado.

Chequeo del primer argumento de fby:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = c \text{ fby } x_2 \})}{\text{FALLO}} \quad c \notin \Sigma_{\varepsilon,s}$$

Chequeo del segundo argumento de fby:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = c \text{ fby } x_2 \})}{\text{FALLO}} \quad x_2 \in I \cup \Sigma_{\varepsilon,s}$$

Chequeo de que los puertos de salida no estén retrasados:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = c \text{ fby } x_2 \})}{\text{FALLO}} \quad x_2 \in O$$

Detección de next:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \text{next } x_2 \})}{\text{FALLO}}$$

Detección de sample:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = n >> x_2 \})}{\text{FALLO}}$$

Detección de replicate:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = x_2 << n \})}{\text{FALLO}}$$

Detección de interleave:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x = (x_1 \,||\, \ldots \,||\, x_n) \})}{\text{FALLO}}$$

Obsérvese que esta fase no realiza chequeos sintácticos (como podrían ser la detección de que no se escriben puertos de entrada, o de que no se leen puertos de salida, o de que todas las señales están declaradas, o de que no existen colisión de nombres) ni semánticos (como la detección de lazos combinacionales o de tautologías del tipo $x = x$) ya que de eso debe encargarse el módulo cargador. Al igual que tampoco detecta especificaciones no razonables (como aquellas que contengan puertos de salida definidos como constantes o definidos redundantemente). El objetivo de esta fase es filtrar especificaciones ecuacionales correctas, pero cuya síntesis no puede ser adecuadamente guiada por este algoritmo.

Multiplexación de fuentes.

El propósito de esta fase es aumentar la frecuencia de lectura de las fuentes de datos que, en cualquier diseño, son los puertos de entrada, las constantes y salidas de retardos arquitectónicos.

Multiplexación de puertos de entrada:

$$\frac{(\ \Sigma,\ X,\ I \cup \{p\},\ O,\ \varphi \cup \{\ x = \varphi(x)\ \}\ )}{\text{AplicacionDI}(\ x,\ i,\ \text{IREP}(\ \lambda\ )\ )} \quad \exists i \in \mathbf{N}_+,\ \varphi(x)[i] \equiv p \wedge \varphi(x)[\varepsilon] \in \Sigma_{w,s}$$

Multiplexación de constantes:

$$\frac{(\ \Sigma,\ X,\ I,\ O,\ \varphi \cup \{\ x = \varphi(x)\ \}\ )}{\begin{array}{c}\text{AplicacionDI}(\ x,\ i,\ \text{IREP}(\ \lambda\ )\ )\\ \text{AplicacionID}(\ x,\ i.2,\ \text{NREP}(\ \lambda,\ \varphi(x)[i]\ )\ )\end{array}} \quad \exists i \in \mathbf{N}_+,\ \varphi(x)[i] \in \Sigma_{\varepsilon,s} \wedge \varphi(x)[\varepsilon] \in \Sigma_{w,s}$$

Multiplexación de retardos arquitectónicos:

$$\frac{(\ \Sigma,\ X,\ I,\ O,\ \varphi \cup \{\ x_1 = c\ \text{fby}\ x_2\ \}\ )}{\text{AplicacionDI}(\ x_1,\ \varepsilon,\ \text{IREP}(\ \lambda\ )\ )}$$

Obsérvese cómo el número de veces que se aplica cada una de las reducciones está limitado por el número de puertos de entrada, el número de constantes y el número de retardos arquitectónicos que existan, respectivamente, en el cuerpo ecuacional. Tras esta fase la especificación (dado que fue previamente normalizada) sólo posee definiciones de la forma:

- $\lambda >> (\ (\ c\ \text{fby}\ x\ ) << \lambda\ )$ *un retardo arquitectónico multiplexado*
- $\sigma(\ t_1, \ldots, t_n\ )$, donde cada t_i podrá ser:
  - x *una señal convencional*
  - $\lambda >> c$ *una constante multiplexada*
  - $\lambda >> (\ in << \lambda\ )$ *un puerto de entrada multiplexado*

En cuanto a la definición de las reducciones, nótese cómo el hecho de partir de una especificación normalizada simplifica el chequeo de las posiciones en donde buscar puertos y constantes a las posiciones sucesoras

de la raíz, evitando que la aplicación indeterminista de las mismas entre en un lazo de reducción.

Asimismo, obsérvese que la *multiplexación de constantes* chequea que el símbolo raíz de las definiciones sea una operación funcional (chequeando que está definida en la signatura) para evitar tanto que se multiplexen las potenciales constantes que los operadores *fby* pudieran tomar como argumento izquierdo, como que se multiplexen las constantes añadidas por anteriores multiplexaciones de los retardos arquitectónicos (que ocupan en la definición justamente una posición del tipo *i*, por ejemplo el primer 4 del término 4 >> ((0 fby *x*) << 4)).

EJEMPLO 5.3

Tras la finalización de esta fase sobre la especificación normalizada mostrada en el ejemplo 5.1, obtenemos la siguiente especificación ecuacional (recuérdese que dado que vamos a reproducir formalmente los pasos de síntesis mostrados en dicho ejemplo, se tiene que λ=5):

```
body
out = z - t1
t1 = t2 + t3
t7 = t8 + t9
t2 = ( 5 >> a1 ) * t12
t3 = ( 5 >> a2 ) * t11
t8 = ( 5 >> b1 ) * t12
t9 = ( 5 >> b2 ) * t11
z = t7 + ( 5 >> ( in << 5 ) )
t11 = 5 >> ( ( 0 fby t12 ) << 5 )
t12 = 5 >> ( ( 0 fby z ) << 5 )
```

Para conseguirla deben realizarse 4 *multiplexaciones de constantes*, 2 *multiplexaciones de retardos arquitectónicos* y 1 *multiplexación de puertos de entrada*.

Planificación.

Esta fase asigna, a cada una de las operaciones de la especificación normalizada, el ciclo en que se ejecutará según lo establecido en la planificación definida por la aplicación τ.

Planificación del ciclo de actualización de los retardos arquitectónicos:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \lambda >> (\ (\ c \text{ fby } x_2\ ) << \lambda\ )\ \}\ )}{\begin{array}{c}\text{AplicacionID}(\ x_1, \varepsilon, \text{TMT}(\ \lambda, \lambda\text{-}\tau(\text{fby})\ )\ ) \\ \text{Expansion}(\ x_1, 2, x_3\ )\end{array}} \quad x_3 \notin X \cup Lu(\Sigma)$$

Planificación de operaciones:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x = \lambda >> \sigma(\ t_1, ..., t_n\ )\ \}\ )}{\begin{array}{c}\text{AplicacionID}(\ x, \varepsilon, \text{TMT}(\ \lambda, \lambda\text{-}\tau(\sigma)\ )\ ) \\ \text{Expansion}(\ x, 2, x_0\ ) \\ \forall i \in \{\lambda\text{-}\tau(\sigma)\text{-}1..0\},\ \text{AplicacionDI}(\ x_0, [2.]^{\lambda-\tau(\sigma)}.\tau(\sigma)[.1]^i, \text{DNEXT}(\sigma)\ ) \\ \forall i \in \{1..n\},\ \text{Expansion}(\ x_0, [2.]^{\lambda-\tau(\sigma)}.\tau(\sigma).i, x_i\ )\end{array}} \quad \begin{array}{l}\sigma \in \Sigma_{w,s} \wedge \\ x_0, x_i \notin X \cup Lu(\Sigma)\end{array}$$

Exportación del operador sample:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1),\ x_2 = \lambda >> x_3\ \}\ )}{\begin{array}{c}\text{Substitucion}(\ x_1, u\ ) \\ \text{Eliminacion}(\ x_2\ )\end{array}} \quad \exists u \in pos(\varphi(x_1)),\ \varphi(x_1)[u] \equiv x_2$$

Extracción del operador sample:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x = \sigma(\ \lambda >> t_1, ..., \lambda >> t_n\ )\ \}\ )}{\text{AplicacionID}(\ x, \varepsilon, \text{DSAM}(\ \sigma\ )\ )} \quad \sigma \in \Sigma_{w,s}$$

Homogeneización de los retardos arquitectónicos:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = [\#\ \text{fby}\ ]^m([\#\ ||\ ]^{\lambda-m-1}\ (c\ \text{fby}\ (\lambda >> x_2))) << \lambda\ [||\ \#]^m)\ \}\ )}{\begin{array}{c}\text{AplicacionID}(\ x_1,\ \varepsilon[.2]^m,\ \text{ADRET}(\ \lambda,\ m\ )\ ) \\ \text{Expansion}(\ x_1,\ \varepsilon[.2]^m.\lambda\text{-}m,\ x_3\ )\end{array}}\ x_3 \notin X \cup Lu(\Sigma)$$

Tras la finalización de esta fase, la especificación ecuacional sólo posee definiciones de la forma:

(i) $x = [\ \#\ \text{fby}\ ]^m\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ \sigma(\ x_1, ..., x_n\ )\ [\ ||\ \#\ ]^m\ )$. Una por cada una de las apariciones de una operación funcional en la especificación normalizada. Mediante m se indica tanto el ciclo en que se ha planificado (que es λ-m), como el número máximo de ciclos que potencialmente necesitará estar almacenado el valor que genera, para que sea usado por posteriores operaciones (efectivamente m ciclos, es decir, hasta el último ciclo de la planificación).

(ii) $x_1 = [\ \#\ \text{fby}\ ]^m\ [\ \#\ \text{fby}\ ]^{\lambda-m-1}\ c\ \text{fby}\ [\ \#\ \text{fby}\ ]^m\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_2\ [\ ||\ \#\ ]^m\ )$. Una por cada una de las apariciones de un retardo arquitectónico en la especificación normalizada. Mediante m se indica tanto el ciclo en que se ha planificado su actualización (que es λ-m), como el número máximo de ciclos que potencialmente necesitará estar almacenado el valor arquitectónico que memoriza para que sea usado por posteriores operaciones (efectivamente $m+\lambda$ ciclos, es decir, el número de ciclos existentes hasta el final de la planificación más todos los ciclos que componen la siguiente iteración completa del algoritmo).

(iii) $x_1 = \lambda >> x_2$. Una por cada puerto de salida.

(iv) $x_1 = [\ \text{next}\ ]^m\ x_2$. Una por cada lectura que se realice de los valores generados por una operación funcional o de los valores almacenados por un registro arquitectónico, o lo que es lo mismo, una por cada argumento que tome como entrada cada una de las operaciones de

la especificación normalizada. Mediante *m* se indica el ciclo en que se realiza la lectura (que es λ-*m*). Obsérvese que si *m*=0 este tipo de definición es una señal de paso. Así mismo nótese que pueden también existir definiciones redundantes, tantas como lecturas que, planificadas en un mismo ciclo, sean realizadas sobre la misma fuente por operaciones distintas. Si estas lecturas se efectúan en ciclos diferentes, el numero de *next* de las definiciones no será el mismo.

(v) x = [next]^m c. Una por cada lectura que se realice de una constante. Al igual que antes pueden existir definiciones de paso y definiciones redundantes.

(vi) x_1 = [next]^m (x_2 << λ). Una por cada lectura que se realice de un puerto de entrada. Pueden existir también redundancias.

La reducción de *planificación del ciclo de actualización de los retardos arquitectónicos*, se aplicará tantas veces como retardos arquitectónicos posea la especificación ecuacional. En dicho proceso se generan definiciones de la forma λ >> x que disparan *exportaciones del operador sample* y definiciones con la forma:

$$\#\ \text{fby}\ (\ [\ \#\ ||\ ]^{k-m-1}\ \text{next}^m\ (\ (\ c\ \text{fby}\ x\ )\ <<\ \lambda\ )\ [\ ||\ \#\ ]^m\ )$$

que no disparan, por el momento, ninguna otra reducción.

Por su parte la reducción de *planificación de operaciones*, se aplicará tantas veces como operaciones funcionales posea el cuerpo ecuacional. Estudiemos en detalle cada una de las reglas de transformación que se aplican en esta reducción. La aplicación de TMT transforma la definición original en otra de la forma:

$$\lambda >>\ [\ \#\ \text{fby}\ ]^{\lambda-\tau(\sigma)}\ (\ [\ \#\ ||\ ]^{\tau(\sigma)-1}\ \text{next}^{\lambda-\tau(\sigma)}\ \sigma(\ t_1, \ldots, t_n\ )\ [\ ||\ \#\ ]^{\lambda-\tau(\sigma)}\ )$$

que tras la primera expansión se divide en dos, una de la forma λ >> x que dispara una *exportación del operador sample* (si la operación planificada no ataca a un puerto de salida) y otra que se continúa transformando vía la

aplicación de DNEXT λ-$\tau(\sigma)$ veces. Mediante esta aplicación se consigue pasar la cadena de *next* de la salida de σ a cada una de sus n entradas, quedando la definición de la forma:

$$[\ \#\ \text{fby}\ ]^{\lambda-\tau(\sigma)}\ (\ [\ \#\ ||\ ]^{\tau(\sigma)-1}\ \sigma(\ \text{next}^{\lambda-\tau(\sigma)}\ t_1,\ \ldots,\ \text{next}^{\lambda-\tau(\sigma)}\ t_n\ )\ [\ ||\ \#\ ]^{\lambda-\tau(\sigma)}\ )$$

Por último las últimas n expansiones transforman esta definición en una de tipo (*i*) generando otras de los tipos (iv), (v) y (vi) que no disparan ninguna otra reducción.

Obsérvese que si $\tau(\sigma)=\lambda$, DNEXT no se aplica ninguna vez. Por otro lado, para evitar que si m=0 se generen señales de paso, podría descomponerse dicha reducción en dos: una especial en caso de planificaciones al último ciclo, y otra para el resto de los ciclos. No se ha hecho por simplicidad.

Nótese también cómo afecta al proceso de transformación la complejidad de las $Lu(\Sigma)$-ecuaciones utilizadas. Así, en general, conforme crece la complejidad de las ecuaciones decrece la complejidad del sistema de reducción. Por ejemplo, en la *planificación de operaciones*, la aplicación sucesiva λ-$\tau(\sigma)$-1 veces de DNEXT(σ) podría reemplazarse por una única aplicación de una ecuación más compleja DNEXT(σ,λ-$\tau(\sigma)$) donde pudiera parametrizarse el número de *next* a retemporizar.

Por último destacar otra vez que la agrupación de transformaciones en reducciones se ha hecho atendiendo a su significado hardware ya que, por ejemplo, esta reducción podría haberse descompuesto en varias por ser el orden en que se realicen las expansiones completamente irrelevante. Así mismo decir que no existe problema en diseñar implementaciones que simultáneamente realicen varias reducciones. Por ejemplo, a la vez que se aplica la propiedad DNEXT sobre distintas ecuaciones, es posible estar realizando la exportación del operador *sample* que dispara la expansión previa.

En cuanto a la reducción de *exportación del operador sample*, ésta necesitará aplicarse tantas veces como lecturas del resultado de una

operación se realicen en el cuerpo ecuacional†. Si el resultado de una misma operación es leído varias veces por operaciones distintas, la regla de eliminación solo tendrá efecto cuando exportemos el operador *sample* sobre el último de ellos, el resto no tendrá efecto. Esta reducción genera ecuaciones con subtérminos de la forma $\lambda >> x$. Cuando todos los operandos de una operación funcional sean de dicho tipo, podrá dispararse la reducción de *extracción del operador sample*. Por otro lado, cuando se exporte a un registro arquitectónico se generarán definiciones con la forma:

$$\#\ \text{fby}\ (\ [\ \#\ ||\ ]^{k-m-1}\ \text{next}^{m}\ (\ (\ c\ \text{fby}\ (\ \lambda >> x\ )\ )\ <<\ \lambda\ )\ [\ ||\ \#\ ]^{m}\ )$$

que dispararán la reducción de *homogeneización de los retardos arquitectónicos*.

El objetivo de la *extracción del operador sample* es aumentar la frecuencia de funcionamiento de una operación funcional cuyos predecesores ya lo hagan a mayor frecuencia y hayan sido planificados. Se dispara tantas veces como operaciones funcionales haya, generando ecuaciones del tipo $\lambda >> \sigma(\ t_1, \ldots, t_n\ )$ que dispararán la *planificación* de dicha operación.

Para finalizar, la *homogeneización de los retardos arquitectónicos*, se aplica tantas veces como retardos de este tipo posea la especificación ecuacional. Su objetivo es transformar registros que se cargan a baja frecuencia pero que son leídos a alta frecuencia en retardos convencionales que leen y cargan a la misma frecuencia. Se dispara cuando una definición, con un retardo arquitectónico ya planificado, es alcanzada por el operador *sample* de la operación cuya salida carga. Como consecuencia genera definiciones del tipo (ii) y del tipo (iv) que son acordes con las generadas por la *planificación de operaciones*. Ninguna de estas definiciones dispara ninguna otra reducción

† Se pueden hacer lecturas de operaciones (funcionales o de retardo), de constantes, de puertos de entrada y pueden realizar lecturas (o lo que es lo mismo, pueden ser escritos) las operaciones y los puertos de salida (aunque estas últimas no serán tenidas en cuenta excepto que se explicite lo contrario).

de esta fase. Al igual que en la *planificación de operaciones*, las actualizaciones planificadas en el último ciclo generan señales de paso.

EJEMPLO 5.4

El efecto de realizar esta fase sobre la especificación ecuacional del ejemplo 5.3, según las directrices la aplicación τ del ejemplo 5.1, es (donde en el lado derecho se muestra la correspondencia entre las nuevas definiciones y las definiciones originales de la especificación normalizada):

```
body
out = 5 >> t38
t38 = ( # || # || # || # || t37 - t36 )                    z - t1
t37 = t35
t36 = t29                                                  ------------
t29 = ( # || # || # || # || t28 + t27 )                    t2 + t3
t28 = t17
t27 = t20                                                  ------------
t17 = # fby ( # || # || # || t16 * t15 || # )              a1 * t12
t16 = next a1
t15 = next t14                                             ------------
t20 = # fby # fby ( # || # || t19 * t18 || # || # )        a2 * t11
t19 = next next a2
t18 = next next t13                                        ------------
t35 = # fby ( # || # || # || t34 + t33 || # )              t7 + in
t34 = next t32
t33 = next in << 5                                         ------------
t32 = # fby # fby ( # || # || t31 + t30 || # || # )        t8 + t7
t31 = next next t23
t30 = next next t26                                        ------------
t23 = # fby # fby # fby ( # || t22 * t21 || # || # || # )  b1 * t12
t22 = next next next b1
t21 = next next next t14                                   ------------
t26 = # fby # fby # fby # fby ( t25 * t24 || # || # || # || # )  b2 * t11
t25 = next next next next b2
t24 = next next next next t13                              ------------
t13 = # fby # fby # fby # fby 0 fby ( # || # || # || # || t39 )  0 fby t12
t39 = t14                                                  ------------
t14 = # fby # fby # fby # fby 0 fby # fby ( # || # || # || t40 || # )
t40 = next t35                                             0 fby z
```

Dado que en la especificación normalizada había 8 operadores funcionales, se han realizado 8 *planificaciones de operaciones* y 8 *extracciones del operador sample*. Además las 2 operaciones de retardo arquitectónico han hecho que se disparasen 2 *planificaciones del ciclo de actualización de los retardos arquitectónicos* y 2 *homogeneizaciones de retardos arquitectónicos*. La existencia en la especificación normalizada de 13 lecturas de valores generados por operaciones (sin contar las 4 lecturas de constantes y la única lectura del puerto de entrada) han hecho que en esta fase se realicen 13 disparos de la reducción de 1 *exportación del operador sample*.

Como consecuencia en el cuerpo ecuacional aparecen 29 definiciones que pueden clasificarse:

- Por las 8 operaciones funcionales, 8 definiciones del tipo (i): t38, t29, t17, t20, t35, t32, t23 y t26.
- Por las 2 operaciones de retardo arquitectónico, 2 definiciones del tipo (ii): t13 y t14.
- Por el único puerto de salida, 1 definición del tipo (iii): *out*.
- Por las 13 lecturas de resultados calculados por los operadores funcionales (8) o de valores almacenados en operadores de retardo arquitectónico (5), 13 definiciones del tipo (iv): t37, t36, t28, t27, t15, t18, t34, t31, t30, t21, t24, t39 y t40.
- Por las 4 lecturas de constantes, 4 definiciones del tipo (v): t16, t19, t22 y t25.
- Por la única lectura del puerto de entrada, 1 definición del tipo (vi): t33.

Separación de acciones RT.

En todo cálculo con recursos multiplexados hay 4 acciones implicadas: la interconexión de cada una de las fuentes de operandos con la operación, el

cálculo en sí, la interconexión de la salida de la operación con el recurso de memoria que almacenará el resultado hasta su uso y la memorización. Esta fase aísla en definiciones separadas las acciones de cálculo y almacenaje, y en subtérminos separados cada una de las acciones de selección de fuentes implicadas en las anteriores acciones. Esto permitirá posteriormente reutilizar por separado cada uno de los elementos RT que son necesarios para implementar cada una de dichas acciones.

Separación de las acciones de cálculo y de almacenamiento de resultados:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x = [\ \#\ \text{fby}\ ]^m\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ \sigma(\ x_1, \ldots, x_n\ )\ [\ ||\ \#\ ]^m\ )}{\begin{array}{c}\text{AplicacionID}(\ x,\ \varepsilon[.2]^m,\ \text{DET}(\ \lambda,\ m\ )\ )\\ \text{Expansion}(\ x,\ \varepsilon[.2]^m.\lambda\text{-}m,\ x_0\ )\end{array}} \quad \begin{array}{l}\sigma \in \Sigma_{w,s} \wedge \\ x_0 \notin X \cup Lu(\Sigma)\end{array}$$

Separación de las acciones de cálculo y de selección de operandos:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x = (\ [\ \#\ ||\ ]^{\lambda-m-1}\ \sigma(\ x_1, \ldots, x_n\ )\ [\ ||\ \#\ ]^m\ )\ \}\ )}{\begin{array}{c}\forall i \in \{1..m\},\ \text{Reemplazo}(\ x,\ i,\ \sigma(\ \#, \ldots, \#\ )\ )\\ \text{AplicacionDI}(\ x,\ \varepsilon,\ \text{DINT}(\ \lambda,\ \sigma\ )\ )\end{array}} \quad \sigma \in \Sigma_{w,s}$$

Tras la finalización de esta fase, la especificación ecuacional sólo posee definiciones de la forma (algunas de las cuales ya aparecían tras la anterior fase):

(i) $x_1 = [\ \#\ \text{fby}\ ]^m\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_2\ [\ ||\ \#\ ]^m\ )$. Una por cada una de las apariciones de una operación funcional en la especificación normalizada. Esta definición representa la acción de memorización de un resultado calculado por una definición del tipo (ii).

(ii) $x = \sigma(\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_1\ [\ ||\ \#\ ]^m\ ), \ldots, (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_n\ [\ ||\ \#\ ]^m\ )\ )$. Una por cada una de las apariciones de una operación funcional en la especificación normalizada. Esta definición representa la acción de cómputo que se almacena en una definición del tipo (i).

(iii) x_1 = [# fby]^m [# fby]$^{\lambda-m-1}$ c fby [# fby]^m ([# ||]$^{\lambda-m-1}$ x_2 [|| #]^m). Una por cada una de las apariciones de un retardo arquitectónico en la especificación normalizada.

(iv) $x_1 = \lambda >> x_2$. Una por cada puerto de salida.

(v) x_1 = [next]^m x_2. Una por cada lectura que se realice de los valores generados por cualquier clase de operación.

(vi) x = [next]^m c. Una por cada lectura que se realice de una constante.

(vii) x_1 = [next]^m ($x_2 << \lambda$). Una por cada lectura que se realice de un puerto de entrada.

La primera de las reducciones, la *separación de las acciones de cálculo y de almacenamiento de resultados*, se aplicará tantas veces como operaciones funcionales existan. Su aplicación genera pares de definiciones, donde cada par está integrado por una definición de tipo (i) y otra de la forma:

$$([\# ||]^{\lambda-m-1} \sigma(t_1, ..., t_n) [|| \#]^m)$$

que a su vez, dispara *la separación de las acciones de cálculo y de selección de operandos*. Obsérvese que si la operación está planificada en el ultimo ciclo (m=0) no se necesita memorizar nada y, por tanto, no sería necesario ninguna separación sin embargo, nuevamente se conserva por regularidad.

La segunda reducción, la *separación de las acciones de cálculo y de selección de operandos*, se aplicará también tantas veces como operaciones funcionales posea el cuerpo ecuacional. Obsérvese que uno de los reemplazamientos no tiene efecto, ya que el término a reemplazar (cuando el índice i vale λ-m) no es un comodín. Esta reducción generará definiciones del tipo (ii).

EJEMPLO 5.5

Tras aplicar esta fase sobre la especificación ecuacional del ejemplo 5.4, obtenemos:

body
out = 5 >> t38

```
t38 = ( # || # || # || # || t48 )
t48 = ( # || # || # || # || t37 ) - ( # || # || # || # || t36 )        z - t1
t37 = t35
t36 = t29                                                              ---
t29 = ( # || # || # || # || t45 )
t45 = ( # || # || # || # || t28 ) + ( # || # || # || # || t27 )        t2 + t3
t28 = t17
t27 = t20                                                              ---
t17 = # fby ( # || # || # || t41 || # )
t41 = ( # || # || # || t16 || # ) * ( # || # || # || t15 || # )        a1 * t12
t16 = next a1
t15 = next t14                                                         ---
t20 = # fby # fby ( # || # || t42 || # || # )
t42 = ( # || # || t19 || # || # ) * ( # || # || t18 || # || # )        a2 * t11
t19 = next next a2
t18 = next next t13                                                    ---
t35 = # fby ( # || # || # || t47 || # )
t47 = ( # || # || # || t34 || # ) + ( # || # || # || t33 || # )        t7 + in
t34 = next t32
t33 = next in << 5                                                     ---
t32 = # fby # fby ( # || # || t46 || # || # )
t46 = ( # || # || t31 || # || # ) + ( # || # || t30 || # || # )        t8 + t9
t31 = next next t23
t30 = next next t26                                                    ---
t23 = # fby # fby # fby ( # || t43 || # || # || # )
t43 = ( # || t22 || # || # || # ) * ( # || t21 || # || # || # )        b1 * t12
t22 = next next next b1
t21 = next next next t14                                               ---
t26 = # fby # fby # fby # fby ( t44 || # || # || # || # )
t44 = ( t25 || # || # || # || # ) * ( t24 || # || # || # || # )        b2 * t11
t25 = next next next next b2
t24 = next next next next t13                                          ---
t13 = # fby # fby # fby # fby 0 fby ( # || # || # || # || t39 )        0 fby t12
t39 = t14                                                              ---
t14 = # fby # fby # fby # fby 0 fby # fby ( # || # || # || t40 || # )
t40 = next t35                                                         0 fby z
```

Al existir en la especificación normalizada 8 operaciones funcionales, en esta fase se han realizado 8 *separaciones de las acciones de cálculo y almacenamiento* y 8 *separaciones de las acciones de cálculo y selección de operandos*, que han aumentado a 37 el número de definiciones, que pueden clasificarse en:

- Para el almacenamiento del resultado de las 8 operaciones funcionales, 8 definiciones del tipo (i): t38, t29, t17, t20, t35, t32, t23 y t26. De las cuales 2 de ellas (t38 y t29) no tienen ocurrencias del operador *fby* para denotar que en realidad no necesitan almacenarse, la primera por ser un resultado leído en el último ciclo por un puerto de salida y la segunda por encadenarse con otra operación.
- Para el cálculo de las 8 operaciones funcionales , 8 definiciones del tipo (ii): t48, t45, t41, t42, t47, t46, t43 y t44.
- Por las 2 operaciones de retardo arquitectónico, 2 definiciones del tipo (iii): t13 y t14.
- Por el único puerto de salida, 1 definición del tipo (iv): *out*.
- Por las 13 lecturas de resultados calculados por operadores, 13 definiciones del tipo (v): t37, t36, t28, t27, t15, t18, t34, t31, t30, t21, t24, t39 y t40.
- Por las 4 lecturas de constantes, 4 definiciones del tipo (vi): t16, t19, t22 y t25.
- Por la única lectura del puerto de entrada, 1 definición del tipo (vii): t33.

Eliminación del predictor *next*

El objetivo de esta fase es eliminar pares *next-fby* para que la especificación ecuacional esté formada solamente por operadores causales y por tanto pueda implementarse. Esto, además, se utilizará en la siguiente fase para chequear fácilmente que todo dato sólo es leído después de ser generado. Recuérdese que en la fase de planificación una operación indicaba el ciclo en que realizaba las lecturas de sus operandos mediante cadenas de *next*, y mediante cadenas *fby* el número de ciclos que como máximo podía estar almacenado el valor que genera, por ello el número de cadenas de *next* será

igual al número de lecturas de operandos que se realice, y el número de cadenas de *fby* será igual al número de resultados que se generen.

Preparación de eliminaciones†:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = [\ \text{next}\ ]^m\ x_2\ \}\ )}{\begin{array}{c}\text{Substitucion}(\ x_1,\ \varepsilon[.1]^m\ )\\ \text{Eliminacion}(\ x_2\ )\end{array}}$$

Eliminación de pares next-fby††:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x = [\ \text{next}\ ]^{m-1}\ \text{next}\ (\ c\ \text{fby}\ t\ )\ \}\ )}{\text{AplicacionID}(\ x,\ \varepsilon[.1]^{m-1},\ \text{IFBY}(\ c\ )\ )}$$

Eliminación de predicciones sobre constantes:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x = [\ \text{next}\ ]^{m-1}\ \text{next}\ c\ \}\ )}{\text{AplicacionID}(\ x,\ \varepsilon[.1]^{m-1},\ \text{NNEXT}(\ c\ )\ )} \qquad c \in \Sigma_{\varepsilon,s}$$

Eliminación de predicciones sobre puertos de entrada:

$$\frac{\begin{array}{c}(\ \Sigma, X, I \cup \{p\},\ O,\\ \varphi \cup \{\ x_1 = [\text{next}]^{m-1}\text{next}(p{<<}\lambda),\ x_2{=}\sigma(t_1,\ldots,t_n)\ \}\ )\end{array}}{\begin{array}{c}\text{Substitucion}(\ x_2,\ i.\lambda\text{-}m\ )\\ \text{Eliminacion}(\ x_1\ )\\ \text{AplicacionID}(\ x_2,\ i,\ \text{INAT}(\ \lambda,\ m\ )\ )\\ \text{Expansion}(\ x_2,\ i.\lambda\text{-}m,\ x_3\ )\end{array}} \qquad \begin{array}{l}\exists i,\ t_i = (\ [\#\ ||]^{\lambda-m-1}\ x_1\ [||\#]^m\ )\\ \quad \wedge\ x_3 \notin X \cup Lu(\Sigma)\end{array}$$

† Si hay varias lecturas de x_2, sólo la última eliminación es efectiva.

†† La exigencia en la precondición de la existencia de al menos un *next* en la definición evita que el sistema de reducción cicle.

Tras la finalización de esta fase, la especificación ecuacional sólo posee definiciones de la forma:

(i) $x_1 = [\ \#\ \text{fby}\ ]^n\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_2\ [\ ||\ \#\ ]^m\ )$. Tantas como lecturas se efectúen de los valores calculados por una operación funcional (incluidas las realizadas por un puerto de salida), indicando que el valor se calcula en el ciclo λ-m (por la operación que aparece en la definición de x_2) y se lee n ciclos después, es decir en el ciclo λ-m+n, por lo que es necesario retrasarlo n ciclos. Si n=0 indica que la lectura se realiza en el mismo ciclo que el cálculo por lo que la operación que genera el valor y la que lo lee están encadenadas (el que lo lee puede ser otra operación, un retardo arquitectónico o un puerto de salida).

(ii) $x_1 = [\ \#\ \text{fby}\ ]^n\ c\ \text{fby}\ [\ \#\ \text{fby}\ ]^m\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_2\ [\ ||\ \#\ ]^m\ )$. Tantas como lecturas se efectúen de los valores almacenados por los retardos arquitectónicos, donde c es el valor inicial, λ-m es el ciclo en donde se planificó su actualización y n es el ciclo en donde se efectúa la lectura de dicha actualización. Obsérvese cómo retrasa m ciclos y después n, para indicar que se produce en una iteración del algoritmo y se consume en la siguiente. Obsérvese además que n puede ser mayor que λ-m-1 lo que indica que además de actualizarse en una iniciación y consumirse en la siguiente, puede que se consuma en un ciclo posterior a su actualización por lo que deberá almacenarse durante más de λ ciclos hasta un máximo $2*\lambda$-1 ciclos (actualizado en el primer ciclo de una iniciación y consumido en el último de la siguiente).

(iii) $x = \sigma(\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_1\ [\ ||\ \#\ ]^m\ ),\ \ldots,\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_n\ [\ ||\ \#\ ]^m\ )\ )$. Una por cada una de las apariciones de una operación funcional en la especificación normalizada.

(iv) $x_1 = \lambda >> x_2$. Una por cada puerto de salida.

(v) $x = c$. Una por cada lectura que se realice de una constante.

(vi) $x_1 = x_2 << \lambda$. Una por cada lectura que se realice de un puerto de entrada.

(vii) $[next]^n$ next $\sigma(([\ \#\ ||\ ]^{\lambda-m-1}\ x_1\ [\ ||\ \#\ ]^m), ..., ([\ \#\ ||\ ]^{\lambda-m-1}\ x_n\ [\ ||\ \#\ ]^m)\ )$. Tantas como lecturas incorrectamente planificadas se realicen de los valores calculados por una operación funcional, donde el número de *next* indica el número de ciclos que como mínimo debe retrasarse dicha lectura o, lo que es lo mismo, el número de ciclos que como mínimo debe adelantarse la planificación de la operación que la genera. Para conocer la operación que hace incorrectamente la lectura, basta con comprobar qué operación tiene por argumento la señal así definida.

(viii) $[\ next\ ]^n$ next $(\ [\ \#\ ||\ ]^{\lambda-m-1}\ x\ [\ ||\ \#\ ]^m\ )$, tantas como actualizaciones (lecturas realizadas por un retardo arquitectónico) incorrectamente planificadas se realicen de los valores calculados por una operación funcional, donde el número de *next* indica el número de ciclos que como mínimo debe retrasarse dicha actualización o, lo que es lo mismo, el número de ciclos que como mínimo debe adelantarse la planificación de la operación que la genera. Obsérvese que sólo pueden planificarse incorrectamente la actualización del primer retardo de una cadena de retardadores, es decir, que no se puede planificar incorrectamente actualizaciones de valores almacenados por otros retardos arquitectónicos ya que todos esperan λ ciclos.

La reducción de *preparación de eliminaciones* se aplicará tantas veces como lecturas se realicen de los resultados generados por una operación. Si m=0 elimina señales de paso, si m<>0 junta en una misma definición una cadena de operadores *next* con su correspondiente cadena de operadores *fby*, generando definiciones de la forma:

$$[\ next\ ]^m\ [\ \#\ fby\ ]^{m'}\ (\ [\ \#\ ||\ ]^{\lambda-m'-1}\ x\ [\ ||\ \#\ ]^{m'}\ )$$

y de la forma:

$$[\ next\ ]^m\ [\ \#\ fby\ ]^{m'}\ [\ \#\ fby\ ]^{\lambda-m'-1}\ c\ fby\ [\ \#\ fby\ ]^{m'}\ (\ [\ \#\ ||\ ]^{\lambda-m'-1}\ x\ [\ ||\ \#\ ]^{m'}\ )$$

que, si m<>0, dispararán *eliminaciones de pares next-fby.*

Por otra parte, cada *eliminación de pares next-fby* genera nuevas definiciones de la forma:

$$[\ \text{next}\ ]^{m}\ [\ \#\ \text{fby}\ ]^{m'}\ (\ [\ \#\ ||\ ]^{\lambda-m''-1}\ x\ [\ ||\ \#\ ]^{m''}\ )$$

y de la forma:

$$[\ \text{next}\ ]^{m}\ [\ \#\ \text{fby}\ ]^{m'}\ [\ \#\ \text{fby}\ ]^{\lambda-m''-1}\ c\ \text{fby}\ [\ \#\ \text{fby}\ ]^{m''}\ (\ [\ \#\ ||\ ]^{\lambda-m''-1}\ x\ [\ ||\ \#\ ]^{m''}\ )$$

que a su vez, si m<>0 vuelven a disparar nuevas *eliminaciones de pares.*

Cuando existen lecturas de constantes o puertos, la reducción de *preparación de eliminaciones* genera definiciones de la forma:

$$[\ \text{next}\ ]^{m}\ \sigma$$

o de la forma:

$$[\ \text{next}\ ]^{m}\ (\ p << \lambda\ )$$

que respectivamente disparan cadenas de *eliminación de predicciones sobre constantes* y de *eliminación de predicciones sobre puertos de entrada.*

EJEMPLO 5.6

La aplicación de esta fase sobre la especificación ecuacional del ejemplo 5.5 elimina por completo todos los predictores *next*, quedando la especificación como se muestra a continuación:

```
body
out = 5 >> t38
t38 = ( # || # || # || # || t48 )                                   z - t1
t48 = ( # || # || # || # || t37 ) - ( # || # || # || # || t36 )    ---------
t36 = ( # || # || # || # || t45 )                                   t2 + t3
t45 = ( # || # || # || # || t28 ) + ( # || # || # || # || t27 )    ---------
t28 = # fby ( # || # || # || t41 || # )
t41 = ( # || # || # || t16 || # ) * ( # || # || # || t15 || # )     a1 * t12
t16 = a1                                                           ---------
t27 = # fby # fby ( # || # || t42 || # || # )
t42 = ( # || # || t19 || # || # ) * ( # || # || t18 || # || # )     a2 * t11
t19 = a2                                                           ---------
t40 = ( # || # || # || t47 || # )
t37 = # fby ( # || # || # || t47 || # )
t47 = ( # || # || # || t34 || # ) + ( # || # || # || t49 || # )     t7 + in
```

```
t49 = in << 5                                                          ──────────
t34 = # fby ( # || # || t46 || # || # )                                t8 + t9
t46 = ( # || # || t31 || # || # ) + ( # || # || t30 || # || # )        ──────────
t31 = # fby ( # || t43 || # || # || # )
t43 = ( # || t22 || # || # || # ) * ( # || t21 || # || # || # )        b1 * t12
t22 = b1                                                               ──────────
t30 = # fby # fby ( t44 || # || # || # || # )
t44 = ( t25 || # || # || # || # ) * ( t24 || # || # || # || # )        b2 * t11
t25 = b2                                                               ──────────
t24 = 0 fby ( # || # || # || # || t39 )                                0 fby t12
t18 = # fby # fby 0 fby ( # || # || # || # || t39 )                    ──────────
t21 = # fby 0 fby # fby ( # || # || # || t40 || # )
t15 = # fby # fby # fby 0 fby # fby ( # || # || # || t40 || # )        0 fby z
t39 = # fby # fby # fby # fby 0 fby # fby ( # || # || # || t40 || # )  ──────────
```

En la ejecución de esta fase se han realizado 13 *preparaciones de eliminación*, una por cada una de las lecturas de los resultados que calculan o almacenan los 10 operadores de la especificación normalizada. Para eliminar las 27 ocurrencias del operador *next* en la especificación del ejemplo 5.5, se han disparado 16 *eliminaciones de pares next-fby*, 10 *eliminaciones de predicciones sobre constantes* y 1 *eliminación de predicciones sobre puertos.*

Como puede observarse el número de definiciones se ha reducido a 28, que pueden clasificarse en:

- Por las 9 lecturas que se realizan de los valores calculados por los 8 operadores funcionales†, 9 definiciones del tipo (i): t38, t36, t28, t27, t40, t37, t34, t31 y t30.
- Por las 5 lecturas que se realizan de los valores almacenados por las 2 operaciones de retardo arquitectónico, 5 definiciones del tipo (ii): t24, t18, t21, t15 y t39.
- Por las 8 operaciones funcionales, 8 definiciones del tipo (iii): t48, t45, t41, t42, t47, t46, t43 y t44.

† El valor que en la especificación normalizada se transmitía a través de *z* y se definía como *t7+in*, es leído dos veces: una para el cálculo de la salida en esa iteración y otra para actualizar uno de los retardos arquitectónicos.

- Por el único puerto de salida, 1 definición del tipo (iv): *out*.
- Por las 4 lecturas de constantes, 4 definiciones del tipo (v): t16, t19, t22 y t25.
- Por la única lectura del puerto de entrada, 1 definición del tipo (vi): t49.
- Como no existen lecturas incorrectamente planificadas de valores calculados por operadores funcionales, 0 definiciones del tipo (vii).
- Como tampoco existen lecturas incorrectamente planificadas de valores almacenados por retardos arquitectónicos, 0 definiciones del tipo (viii).

Chequeo de la corrección de la planificación.

Si detecta algún *next* dentro del cuerpo de la especificación ecuacional la planificación es incorrecta ya que describe un circuito no implementable. Los errores que se han cometido pueden extraerse de la forma que adoptan las definiciones según lo establecido en la anterior fase al explicar los tipos (vii) y (viii).

Chequeo de la corrección de la planificación:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x = \text{next } t \})}{\text{FALLO}}$$

Obsérvese que por las reducciones realizadas por el sistema de reducción completo, el chequeo de ocurrencias de *next* sólo debe hacerse en la raíz de las definiciones.

Realimentación de retardos.

Tras la anterior fase, la diferencia existente entre el ciclo en que un valor se calcula y es utilizado, queda representada por una cadena de operadores *fby* de longitud igual a dicha diferencia. Su significado hardware es claro y puede ser directamente implementado mediante una cadena de retardadores encadenados. Sin embargo, al ser ésta una opción de diseño costosa que no aporta ventajas frente a otras, ninguna herramienta de SAN la adopta. En su lugar se escogen alternativas de implementación que utilizan registros (elementos de memoria con control selectivo de carga) y/o retardadores que se realimentan a través de un multiplexor 2 a 1 y con cuya entrada de control es posible seleccionar en cada ciclo si el retardador carga su salida (manteniendo el mismo valor) o carga un nuevo dato.

He decidido que el sistema de síntesis formal realice esta última alternativa por ser un poco menos costosa que la solución basada en registros (que también podría formalizarse). De este modo esta fase reemplaza cadenas de operadores *fby*, por un único operador *fby* realimentado. Además no sólo se asume esta decisión de diseño (implementar dependencias de datos mediante retardadores realimentados) sino que también se asume que la memorización de un valor no se reparte entre varios retardadores sino que permanece en el mismo elemento de memoria desde el ciclo en que se almacena hasta el ciclo en que se usa (siempre que el número de ciclos que tenga que estar memorizado sea menor o igual que la latencia de la planificación). Algoritmos de síntesis que no verificaran esta última asunción, también podrían formalizarse modificando levemente la especificación de esta fase.

Fragmentación de cadenas de retardos con longitud superior a la latencia:

$$\frac{\begin{array}{c}(\ \Sigma, X, I, O, \varphi \cup \\ \{x_1 = [\#\ \text{fby}]^{n-1} \#\ \text{fby}\ [\#\ \text{fby}]^{\lambda-m-1} c\ \text{fby}\ [\#\ \text{fby}]^{m}(\ [\#\ ||]^{\lambda-m-1}\ x_2\ [||\ \#\ ]^{m}\ )\}\ )\end{array}}{\begin{array}{c}\text{AplicacionID}(\ x_1, \varepsilon, \text{FRAG}(\ \lambda, m, n\ )\ ) \\ \text{Expansion}(\ x_1, \varepsilon[.2]^{n}.\lambda\text{-}m, x_3\ )\end{array}} \quad x_3 \notin X \cup Lu(\Sigma)$$

Reemplazamiento de cadenas de retardos generadas por dependencias de datos:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = c\ \text{fby}\ [\ \#\ \text{fby}\ ]^{n}\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_2\ [\ ||\ \#\ ]^{m}\ )\ \}\ )}{\text{AplicacionRecID}(\ x_1, \varepsilon, \text{MEMT}(\ \lambda, m, n\ )\ )} \quad n \leq \lambda\text{-}1$$

Reemplazamiento de cadenas de retardos generadas por la homogeneización de los retardos arquitectónicos:

$$\frac{\begin{array}{c}(\ \Sigma, X, I, O, \\ \varphi \cup \{\ x_1 = [\ \#\ \text{fby}\ ]^{n}\ c\ \text{fby}\ [\ \#\ \text{fby}\ ]^{m}\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_2\ [\ ||\ \#\ ]^{m}\ )\ \}\ )\end{array}}{\text{AplicacionRecID}(\ x_1, \varepsilon, \text{MEMT2}(\ \lambda, m, n\ )\ )} \quad n \leq k\text{-}m\text{-}1$$

Tras la finalización de esta fase, la especificación ecuacional sólo posee definiciones de la forma:

(i) $x_1 = \#\ \text{fby}\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_2\ [\ ||\ x_1\ ]^{n}\ [\ ||\ \#\ ]^{m-n-1}\ )$. Tantas como lecturas que, planificadas en ciclos distintos del último, se efectúen de los valores calculados por una operación funcional. Las lecturas y los cálculos deberán haber sido planificados en ciclos distintos para que aparezcan definiciones de este tipo.

(ii) $x_1 = (\ [\ \#\ ||\ ]^{\lambda-1}\ x_2\ )$. Tantas como lecturas que, planificadas en el último ciclo, se efectúen de los valores calculados por una operación funcional.

(iii) $x_1 = ([\# \|]^{\lambda-m-1} x_2 [\| \#]^{m-1})$. Tantas como lecturas se efectúen de los valores calculados por una operación funcional, tal que la lectura y el cálculo se realicen en el mismo ciclo.

(iv) $x_1 = c$ fby $([\| x_1]^{n-m} [\# \|]^{\lambda-n-1} x_2 [\| x_1]^m)$. Tantas como lecturas se efectúen de los valores almacenados por los retardos arquitectónicos, tal que la actualización del retardo esté planificada en un ciclo igual o posterior al ciclo en que ha sido planificada su lectura.

(v) $x_1 = \#$ fby $([\# \|]^{\lambda-m-1} x_2 [\| x_1]^n [\| \#]^{m-n})$. Tantas como lecturas se efectúen de los valores almacenados por los retardos arquitectónicos, tal que la actualización del retardo esté planificada en un ciclo anterior al ciclo en que ha sido planificada su lectura.

(vi) $x_1 = c$ fby $([\| x_1]^{\lambda-m-1} x_2 [\| x_1]^m)$. Tantas como lecturas se efectúen de los valores almacenados por los retardos arquitectónicos, tal que la actualización del retardo esté planificada en un ciclo anterior al ciclo en que ha sido planificada su lectura.

(vii) $x = \sigma(([\# \|]^{\lambda-m-1} x_1 [\| \#]^m), ..., ([\# \|]^{\lambda-m-1} x_n [\| \#]^m))$. Una por cada una de las apariciones de una operación funcional en la especificación normalizada.

(viii) $x_1 = \lambda >> x_2$. Una por cada puerto de salida.

(ix) $x = c$. Una por cada lectura que se realice de una constante.

(x) $x_1 = x_2 << \lambda$. Una por cada lectura que se realice de un puerto de entrada.

La reducción de *fragmentación de cadenas de retardos de longitud superior a la latencia* se aplica una vez por cada lectura de un retardo arquitectónico se realice en un ciclo posterior y distinto del ciclo que se actualice. Cada una de ellas genera términos de la forma:

$$[\# \text{ fby }]^{\lambda-m-1} c \text{ fby } [\# \text{ fby }]^m ([\# \|]^{\lambda-m-1} x [\| \#]^m)$$

y de la forma:

$$[\# \text{ fby }]^n ([\# \|]^{\lambda-m-1} x [\| \#]^m)$$

que disparan reemplazamientos de arrays de distinto tipo. La primera modela retardadores ocupados en todos los ciclos, la segunda retardadores con ciclos libres.

A su vez, la reducción de *reemplazamiento de cadenas de retardos generadas por dependencias de datos* genera definiciones del tipo (iv) y (v), disparándose tantas veces como lecturas se realicen de los valores calculados por un operador funcional, y que estén planificadas a más de un ciclo de diferencia con respecto al ciclo en que esté planificado su cálculo.

Por su parte la reducción de *reemplazamiento de cadenas de retardos generadas por la homogeneización de los retardos arquitectónicos* se aplica tantas veces como lecturas se realicen de los valores almacenados por un retardo arquitectónico, y que estén planificadas a más de un ciclo de diferencia con respecto al ciclo en que esté planificada su actualización. La aplicación de esta reducción genera definiciones del tipo (iv).

EJEMPLO 5.7

La finalización de esta fase genera la siguiente especificación ecuacional (partiendo de la mostrada en el ejemplo 5.6):

body

out = 5 >> t38
t38 = (# || # || # || # || t48) — *z - t1*
t48 = (# || # || # || # || t37) - (# || # || # || # || t36) — *t2 + t3*
t36 = (# || # || # || # || t45)
t45 = (# || # || # || # || t28) + (# || # || # || # || t27) — *a1 * t12*
t28 = # fby (# || # || # || t41 || #)
t41 = (# || # || # || t16 || #) * (# || # || # || t15 || #) — *a2 * t11*
t16 = a1
t27 = # fby (# || # || t42 || t27 || #)
t42 = (# || # || t19 || # || #) * (# || # || t18 || # || #) — *t7 + in*
t19 = a2
t40 = (# || # || # || t47 || #)
t37 = # fby (# || # || # || t47 || #)
t47 = (# || # || # || t34 || #) + (# || # || # || t49 || #) — *t8 + t9*
t49 = in << 5
t34 = # fby (# || # || t46 || # || #)

```
t46 = ( # || # || t31 || # || # ) + ( # || # || t30 || # || # )
t31 = # fby ( # || t43 || # || # || # )
t43 = ( # || t22 || # || # || # ) * ( # || t21 || # || # || # )      b1 * t12
t22 = b1
t30 = # fby ( t44 || t30 || # || # || # )
t44 = ( t25 || # || # || # || # ) * ( t24 || # || # || # || # )      b2 * t11
t25 = b2
t24 = 0 fby ( # || # || # || # || t39 )                              0 fby t12
t18 = 0 fby ( t18 || t18 || # || # || t39 )
t21 = 0 fby ( t21 || # || # || t40 || t21 )
t15 = 0 fby ( t15 || t15 || t15 || t40 || t15 )
t39 = # fby ( # || # || # || t50 || # )                              0 fby z
t50 = 0 fby ( t50 || t50 || t50 || t40 || t50 )
```

Dado que la actualización de uno de los retardos arquitectónicos (véase fig. 5.1) se ha planificado en el ciclo 4 y uno de sus lecturas se realiza 6 ciclos después para actualizar el otro retardo arquitectónico (en el ciclo 5 de la siguiente iniciación), la reducción de *fragmentación de cadenas de fby con longitud superior a la latencia* se ha disparado 1 vez para tener en dos definiciones separadas el comportamiento de los 2 retardadores que implementan la dependencia de datos referida. Así mismo dado que 2 de las multiplicaciones están planificadas en ciclos con una diferencia superior a 1 respecto de los ciclos en los que están planificadas sus lecturas, la reducción de *reemplazamiento de cadenas de retardos generadas por dependencias de datos* se dispara 2 veces. La reducción de *reemplazamiento de cadenas de retardos generadas por la homogeneización de los retardos arquitectónicos* se dispara 4 veces para reemplazar 4 de las cadenas de retardadores que implementan las diferencias en ciclos mayores que 1 entre la actualización y la lectura de los valores arquitectónicos.

Como consecuencia de la finalización de esta fase podemos clasificar las definiciones generadas en:

- Por las 6 lecturas de valores (planificadas en ciclos distintos del último) que se realizan en ciclos distintos del ciclo en que se planifica su cálculo, 6 definiciones del tipo (i): t28, t27, t37, t34, t31 y t30.

- Por las 2 lecturas que se realizan de valores calculados en el último ciclo, 2 definiciones del tipo (ii): t38 y t36.
- Por la lectura que se encadena a la actualización de uno de los retardos arquitectónicos, 1 definición del tipo (iii): t40.
- Por las 4 lecturas de los retardos arquitectónicos (utilizadas como operandos de las multiplicaciones) que se planifican en ciclos anteriores o iguales a los ciclos en que se planifican sus actualizaciones, 4 definiciones de tipo (iv): t24, t18, t21 y t25.
- Por la lectura del retardo arquitectónico que se realiza en un ciclo posterior a su actualización (realizada por el otro retardo arquitectónico), 1 definición de tipo (v): t39, y 1 definición de tipo (vi): t50.
- Por las 8 operaciones funcionales de la especificación normalizada, 8 definiciones del tipo (vii): t48, t45, t41, t42, t47, t46, t43 y t44.
- Por el único puerto de salida, 1 definición del tipo (viii): *out*.
- Por las 4 lecturas de constantes, 4 definiciones del tipo (ix): t16, t19, t22 y t25.
- Por la única lectura del puerto de entrada, 1 definición del tipo (x): t49.

Reuso implícito de retardadores.

Tras la finalización de la anterior fase, cada uno de los pares producción-lectura (o lo que es lo mismo cada una de las dependencias de datos de la especificación normalizada) aparece en una definición diferente con un operador *fby* en la raíz, si la producción y la lectura se realizan en ciclos distintos. Su significado hardware es que cada transferencia de información tiene asociado un camino propio que en muchos casos atraviesa un elemento de memoria. Así, todo valor producido que sea consumido por varias

operaciones distintas (en el mismo o en distintos ciclos) sigue caminos diferentes y en una implementación directa utilizaría retardadores diferentes.

Sin embargo, la mayor parte de las herramientas de SAN apuestan por utilizar un único retardador para memorizar un mismo valor leído en distintos instantes por distintos consumidores, de manera que el valor permanezca almacenado hasta el ciclo en que se haya planificado la lectura más lejana al ciclo en que está planificada su producción, así dicho valor estará siempre presente a la salida de un único retardador que podrá ser leído en todos los ciclos intermedios por todas aquellas operaciones que lo requieran. Esta fase realiza dicha reutilización de retardadores y dado que todas las herramientas de SAN asumen que debe efectuarse, he decidido llamarla de **reuso implícito**.

Además, al igual que la fase de reuso 'explícito', dado que existen retardadores que almacenan valores arquitectónicos y retardadores que almacenan valores temporales (todos representados por *fby*), he preferido distinguir entre **reusos endogámicos**, cuando se reusan elementos de memoria que almacenan el mismo tipo de valor y **exogámicos**, cuando el reuso se hace para almacenar valores de distinto tipo.

Esta misma discusión puede extenderse a caminos sin retardo como pueden ser los que tienen origen en constantes o en puertos de entrada, o a caminos entre operaciones encadenadas.

Obsérvese que por primera vez comienzan a reemplazarse comodines, por lo que la especificación resultante deja de ser conductualmente equivalente a la original para pasar a ser débilmente equivalente, es decir, que calcula todos los valores relevantes del problema pero también algunos más que se derivan del reuso de los recursos.

Reuso implícito endogámico de retardadores que implementan retardos no arquitectónicos†:

$$\frac{\begin{array}{c}(\ \Sigma, X, I, O, \qquad\qquad n1 \geq n2\\ \varphi \cup \{\ x_1 = \#\ \text{fby}\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x\ [\ ||\ x_1\ ]^{n1}\ [\ ||\ \#\ ]^{m-n1-1}\ ),\\ x_2 = \#\ \text{fby}\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x\ [\ ||\ x_2\ ]^{n2}\ [\ ||\ \#\ ]^{m-n2-1}\ )\ \}\ )\end{array}}{\begin{array}{c}\forall i \in \{\lambda-m+n2+1..\lambda-m+n1\},\ \text{Reemplazo}(\ x_2,\ 2.i,\ \varphi(x_1)/2.i\ )\\ \text{LimpiezaRec}(\ x_1,\ x_2\ )\end{array}}$$

Reuso implícito endogámico de retardadores que implementan retardos arquitectónicos:

$$\frac{\begin{array}{c}(\ \Sigma, X, I, O, \qquad\qquad n1 \geq n2\\ \varphi \cup \{\ x_1 = c\ \text{fby}\ (\ [\ x_1\ ||\ ]^{n1-m}\ [\ \#\ ||\ ]^{\lambda-n1-1}\ x\ [\ ||\ x_1\ ]^{m}\ ),\\ x_2 = c\ \text{fby}\ (\ [\ x_2\ ||\ ]^{n2-m}\ [\ \#\ ||\ ]^{\lambda-n2-1}\ x\ [\ ||\ x_2\ ]^{m}\ )\ \}\ )\end{array}}{\begin{array}{c}\forall i \in \{1..n1-m\},\ \text{Reemplazo}(\ x_2,\ 2.i,\ \varphi(x_1)/2.i\ )\\ \text{LimpiezaRec}(\ x_1,\ x_2\ )\end{array}}$$

Reuso implícito exogámico de retardadores:

$$\frac{\begin{array}{c}(\ \Sigma, X, I, O,\\ \varphi \cup \{\ x_1 = c\ \text{fby}\ (\ [\ x_1\ ||\ ]^{n1-m}\ [\ \#\ ||\ ]^{\lambda-n1-1}\ x\ [\ ||\ x_1\ ]^{m}\ ),\\ x_2 = \#\ \text{fby}\ (\ [\ \#\ ||\ ]^{\lambda-m-1}\ x\ [\ ||\ x_2\ ]^{n2}\ [\ ||\ \#\ ]^{m-n2-1}\ )\ \}\ )\end{array}}{\begin{array}{c}\text{Reemplazo}(\ x_2,\ 1,\ \varphi(x_1)[1]\ )\\ \forall i \in \{1..n1-m\},\ \text{Reemplazo}(\ x_2,\ 2.i,\ \varphi(x_1)/2.i\ )\\ \forall i \in \{\lambda-m+n2+1..\lambda\},\ \text{Reemplazo}(\ x_2,\ 2.i,\ \varphi(x_1)/2.i\ )\\ \text{LimpiezaRec}(\ x_1,\ x_2\ )\end{array}}$$

† Nótese que la condición $n1 \geq n2$ no es una restricción ya que el orden de las definiciones en una especificación ecuacional no es relevante y por tanto x_1 y x_2 son dos definiciones genéricas que en caso de que $n1 < n2$ bastaría con tomarlas en el orden contrario.

Reuso implícito de retardadores que implementan fragmentos de retardos arquitectónicos:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = c \text{ fby } (\ [\ ||\ x_1\ ]^{\lambda-m-1}\ x\ [\ ||\ x_1\ ]^{m}\ ),\ x_2 = c \text{ fby } (\ [\ ||\ x_2\ ]^{\lambda-m-1}\ x\ [\ ||\ x_2\ ]^{m}\ )\ \}\ )}{\text{LimpiezaRec}(\ x_1,\ x_2\ )}$$

Eliminación de las lecturas redundantes de constantes y puertos de entrada:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = \varphi(x_1),\ x_2 = \varphi(x_2)\ \}\ )}{\text{Limpieza}(\ x_1,\ x_2\ )} \qquad \varphi(x_1) \equiv \varphi(x_2) \wedge \wedge\ \varphi(x_1) \in I \cup \Sigma_{\varepsilon,s}$$

Eliminación de lecturas encadenadas:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = (\ t_1\ ||\ \ldots\ ||\ t_\lambda\ )\ \}\ )}{\begin{array}{c}\forall i \in \{1..\lambda\},\ \text{Reemplazo}(\ x_1,\ 1,\ x_2\ ) \\ \text{AplicacionID}(\ x_1,\ \varepsilon,\ \text{NINT}(\ \lambda\ )\ ) \\ \text{Substitucion}(\ x_1,\ \varepsilon\ ) \\ \text{Limpieza}(\ x_1,\ x_2\ )\end{array}} \qquad \forall i,\ (t_i \not\equiv \# \vee t_i \not\equiv x_2)$$

Tras esta fase el cuerpo ecuacional solamente posee definiciones del siguiente tipo:

(i) $x_1 = \#$ fby $(\ [\ \#\ ||\ ]^{\lambda-m-1}\ x_2\ [\ ||\ x_1\ ]^{n}\ [\ ||\ \#\ ]^{m-n-1}\ )$. Una por cada operación funcional no encadenada con ninguna otra operación, y que haya sido planificada en un ciclo distinto del último

(ii) $x_1 = (\ [\ \#\ ||\ ]^{\lambda-1}\ x_2\ )$. Una por cada operación funcional que haya sido planificada en el último ciclo.

(iii) $x_1 = c$ fby $(\ [\ ||\ x_1\ ]^{n-m}\ [\ \#\ ||\ ]^{\lambda-n-1}\ x_2\ [\ ||\ x_1\ ]^{m}\ )$. Una por cada retardo arquitectónico, tal que su actualización esté planificada en un ciclo igual o posterior a toda lectura del valor que almacena.

(iv) x_1 = # fby ([# ||]$^{\lambda-m-1}$ x_2 [|| x_1]^n [|| #]$^{m-1}$). Una por cada retardo arquitectónico, tal que su actualización esté planificada en un ciclo anterior a alguna lectura del valor que almacena.

(v) x_1 = c fby ([|| x_1]$^{\lambda-m-1}$ x_2 [|| x_1]^m). Una por cada retardo arquitectónico, tal que su actualización esté planificada en un ciclo anterior a alguna lectura del valor que almacena.

(vi) $x = \sigma($ ([# ||]$^{\lambda-m-1}$ x_1 [|| #]^m), ..., ([# ||]$^{\lambda-m-1}$ x_n [|| #]^m)). Una por cada operación funcional de la especificación normalizada.

(vii) $x_1 = \lambda >> x_2$. Una por cada puerto de salida.

(viii) $x = c$. Una por cada constante.

(ix) $x_1 = x_2 << \lambda$. Una por cada puerto de entrada.

Cada una de las reducciones deberá aplicarse tantas veces como duplicidades en la lectura de cierto valor existan en la especificación normalizada.

EJEMPLO 5.8

Tras el reuso de retardos y de caminos de comunicación, la especificación ecuacional del ejemplo 5.7 se transforma en:

```
body
out = 5 >> t48
t48 = ( # || # || # || # || t50 ) - ( # || # || # || # || t45 )      z - t1
t45 = ( # || # || # || # || t28 ) + ( # || # || # || # || t27 )      t2 + t3
t28 = # fby ( # || # || # || t41 || # )
t41 = ( # || # || # || t16 || # ) * ( # || # || # || # || t50 || # )      a1 * t12
t16 = a1
t27 = # fby ( # || # || t42 || t27 || # )
t42 = ( # || # || t19 || # || # ) * ( # || # || t18 || # || # )      a2 * t11
t19 = a2
t47 = ( # || # || # || t34 || # ) + ( # || # || # || t49 || # )      t7 + in
t49 = in << 5
t34 = # fby ( # || # || t46 || # || # )      t8 + t9
t46 = ( # || # || t31 || # || # ) + ( # || # || t30 || # || # )
t31 = # fby ( # || t43 || # || # || # )
t43 = ( # || t22 || # || # || # ) * ( # || t50 || # || # || # )      b1 * t12
t22 = b1
```

```
t30 = # fby ( t44 || t30 || # || # || # )
t44 = ( t25 || # || # || # || # ) * ( t18 || # || # || # || # )      b2 * t11
t25 = b2                                                            ----------
t18 = 0 fby ( t18 || t18 || # || # || t39 )                          0 fby t12
t50 = 0 fby ( t50 || t50 || t50 || t47 || t50 )                      ---------
t39 = # fby ( # || # || # || t50 || # )                              0 fby z
                                                                    ----------
```

Como en la especificación original ningún valor calculado por un operador funcional es leído por más de un operador funcional, no existen duplicidades generadas por dependencias de datos y la reducción de *reuso implícito endogámico de retardadores que implementan retardos no arquitectónicos* no se dispara ninguna vez. Por otra parte, como uno de los retardos arquitectónicos es leído 2 veces y el otro 3, se generan un total de 3 duplicidades (1 y 2) que hacen disparar la reducción de *reuso implícito endogámico de retardadores que implementan retardos arquitectónicos* 3 veces. El valor que en la especificación normalizada es transportado por la señal *z*, es leído 2 veces, una para actualizar un retardo arquitectónico y otra para realizar otro cálculo, esto hace que se aplique 1 vez el *reuso implícito exogámico de retardadores*. Dado que sólo existe una transferencia de valores con una longitud superior a la latencia (la que permite actualizar un retardo arquitectónico con el valor almacenado por el otro), no existen fragmentos duplicados de retardos arquitectónicos por lo que la reducción de *reuso implícito de retardadores que implementan fragmentos de retardos arquitectónicos* no se llega a aplicar ninguna vez. Al no existir tampoco lecturas redundantes de puertos ni de constantes, la eliminación de lecturas redundantes de constantes y puertos de entrada no llega aplicarse. Para finalizar las 2 operaciones funcionales encadenadas en la planificación (una con otra operación funcional y otra con la actualización de un retardo arquitectónico) hacen que la *eliminación de lecturas encadenadas* se dispare 2 veces.

Como efecto de la aplicación de las anteriores reducciones, el número de definiciones se hace menor y éstas pueden clasificarse en:

- Por las 5 operaciones funcionales no encadenadas y no planificadas en el último ciclo, 5 definiciones de tipo (i): t28, t27, t34, t31 y t30.
- Por las 2 operaciones funcionales planificadas en el último ciclo, 2 definiciones del tipo (ii): t48 y t45.
- Por el retardo arquitectónico cuya actualización se realiza en un ciclo igual o posterior a cualquier lectura, 1 definición del tipo (iii): t18.
- Por el retardo arquitectónico que se lee en un ciclo posterior al ciclo en que se planifica su actualización, 1 definición de tipo (iv): t39, y 1 definición de tipo (v): t50.
- Por las 8 operaciones funcionales de la especificación normalizada, 8 definiciones del tipo (vi): t48, t45, t41, t42, t47, t46, t43 y t44.
- Por el único puerto de salida, 1 definición del tipo (vii): *out*.
- Por las 4 constantes, 4 definiciones del tipo (ix): t16, t19, t22 y t25.
- Por el único puerto de entrada, 1 definición del tipo (x): t49.

Aplanado.

Para facilitar el enunciado de la siguiente fase y para poder reusar independientemente los caminos de transferencia, es necesario aplanar la especificación ecuacional.

Aplanado:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \varphi(x_1) \})}{\text{Expansion}(x_1, u, x_2)} \quad u \notin pos_0(\varphi(x_1)) \cup \{\varepsilon\} \wedge x_2 \notin X \cup Lu(\Sigma)$$

Reuso de recursos.

Esta fase reutiliza (bajo el dictado de la aplicación $\alpha^{\dagger}$) los símbolos de operación para realizar las operaciones de la especificación normalizada (este uso compartido de los símbolos de operación es el equivalente simbólico al reuso hardware en un circuito real).

Debe destacarse que en esta fase se incluyen reducciones para reproducir un amplio conjunto de alternativas de diseño diferentes. Que todas ellas se apliquen o no, dependerá de las capacidades de la herramienta de optimización externa que construya la aplicación α. Así, por ejemplo, si la herramienta de diseño no es capaz de reutilizar multiplexores entre elementos funcionales distintos, α_{MUX} no estará definida y por tanto la reducción de *reuso de caminos de comunicación entre recursos funcionales diferentes* no llegará nunca a dispararse.

Reuso de recursos funcionales:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \sigma_1(x_1^1, \ldots, x_1^n),\ x_2 = \sigma_2(x_2^1, \ldots, x_2^n) \})}{\begin{array}{c} \forall j \in \{1..n\},\ \forall i \in \{1..\lambda\},\ \text{Reemplazo}(x_1^j, i, \varphi(x_2^j)/i) \\ \forall j \in \{1..n\},\ \forall i \in \{1..\lambda\},\ \text{Reemplazo}(x_2^j, i, \varphi(x_1^j)/i) \\ \forall j \in \{1..n\},\ \text{Limpieza}(x_1^j, x_2^j) \\ \text{Limpieza}(x_1, x_2) \end{array}} \quad \alpha_{FU}(\sigma_1) = \alpha_{FU}(\sigma_2)$$

† Que por simplicidad, tomará por argumento el propio símbolo de operación en lugar de la ocurrencia de dicho símbolo en la especificación original.

Reuso endogámico de retardadores que implementan retardos no arquitectónicos[†]:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \# \text{ fby } x_1', x_2 = \# \text{ fby } x_2' \})}{\begin{array}{l} \forall i \in \{1..\lambda\}, \text{ Reemplazo}(x_1', i, \varphi(x_2')/i) \\ \forall i \in \{1..\lambda\}, \text{ Reemplazo}(x_2', i, \varphi(x_1')/i) \\ \quad \text{Limpieza}(x_1', x_2') \\ \quad \text{Limpieza}(x_1, x_2) \end{array}} \quad \alpha_{ST}(\text{fby}_1) = \alpha_{ST}(\text{fby}_2)$$

Reuso endogámico de retardadores que implementan retardos arquitectónicos:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = c \text{ fby}_1 x_1', x_2 = c \text{ fby}_2 x_2' \})}{\begin{array}{l} \forall i \in \{1..\lambda\}, \text{ Reemplazo}(x_1', i, \varphi(x_2')/i) \\ \forall i \in \{1..\lambda\}, \text{ Reemplazo}(x_2', i, \varphi(x_1')/i) \\ \quad \text{Limpieza}(x_1', x_2') \\ \quad \text{Limpieza}(x_1, x_2) \end{array}} \quad \alpha_{ST}(\text{fby}_1) = \alpha_{ST}(\text{fby}_2)$$

Reuso exogámico de retardadores:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = c \text{ fby}_1 x_1', x_2 = \# \text{ fby}_2 x_2' \})}{\begin{array}{l} \forall i \in \{1..\lambda\}, \text{ Reemplazo}(x_1', i, \varphi(x_2')/i) \\ \quad \text{Reemplazo}(x_2, 1, \varphi(x_1)/1) \\ \forall i \in \{1..\lambda\}, \text{ Reemplazo}(x_2', i, \varphi(x_1')/i) \\ \quad \text{Limpieza}(x_1', x_2') \\ \quad \text{Limpieza}(x_1, x_2) \end{array}} \quad \alpha_{ST}(\text{fby}_1) = \alpha_{ST}(\text{fby}_2)$$

[†] Obsérvese que en esta reducción, α toma como argumento un operador *fby* en lugar de la operación funcional cuyo resultado memoriza. Se ha hecho por simplicidad ya que siempre es posible conocerla.

Reuso de caminos de comunicación entre recursos funcionales diferentes[†]:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \sigma_1(x_1^1, \ldots, x_1^n),\ x_2 = \sigma_2(x_2^1, \ldots, x_2^n) \})}{\begin{array}{c} \forall i \in \{1..\lambda\},\ \text{Reemplazo}(x_1^{j1}, i, \varphi(x_2^{j2})/i) \\ \forall i \in \{1..\lambda\},\ \text{Reemplazo}(x_2^{j2}, i, \varphi(x_1^{j1})/i) \\ \text{Limpieza}(x_1^{j1}, x_2^{j2}) \end{array}} \quad \begin{array}{l} \alpha_{MUX}(\sigma_1 j_1) = \\ \alpha_{MUX}(\sigma_2 j_2) \end{array}$$

Reuso de caminos de comunicación entre retardadores y recursos funcionales:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = c\ \text{fby}_1\ x_1',\ x_2 = \sigma_2(x_2^1, \ldots, x_2^n) \})}{\begin{array}{c} \forall i \in \{1..\lambda\},\ \text{Reemplazo}(x_1', i, \varphi(x_2^{j2})/i) \\ \forall i \in \{1..\lambda\},\ \text{Reemplazo}(x_2^{j2}, i, \varphi(x_1')/i) \\ \text{Limpieza}(x_1', x_2^{j2}) \end{array}} \quad \begin{array}{l} \alpha_{MUX}(\text{fby}_1) = \\ \alpha_{MUX}(\sigma_2 j_2) \end{array}$$

El esquema de todas las reducciones es similar: si dos operaciones que se realizan sobre dos operadores distintos deben compartir un único recurso, la aplicación de la reducción consigue (vía reemplazo de comodines) que ambos operadores realicen las dos operaciones. De este modo sus definiciones se tornan redundantes (sintácticamente iguales) y puede eliminarse una de ellas. Además, dado que dos operaciones que comparten un recurso comparten también caminos de comunicación, no sólo se hace redundante el operador sino también todas las señales que toma como argumento. Obsérvese que por este razonamiento, no tiene sentido que exista una reducción de *reuso de caminos de comunicación entre retardadores diferentes* ya que no tiene sentido que dos operaciones de retardo compartan un mismo camino de comunicación sin compartir el propio retardador.

[†] Desde el punto de vista hardware esto es interesante cuando entradas distintas de operadores diferentes lean las mismas fuentes (pudiendo utilizar un único multiplexor para ambas) o cuando las leídas por un operador sean un subconjunto de las leídas por el otro (pudiendo utilizar las entradas libres del multiplexor asociado al primer operador).

Tras esta fase el cuerpo ecuacional sólo poseerá definiciones con las siguiente forma:

(i) $x = c$ fby x_1. Tantas como retardadores necesite el circuito final, donde c es el valor inicial del registro y que en caso de ser # podrá ser cualquiera. Si la asignación ha sido correcta el número de tales definiciones deberá ser igual a la cardinalidad de $Recs_{ST}$, si no será mayor.

(ii) $x = \sigma(x_1, ..., x_n)$. Tantas como recursos funcionales requiera el circuito final. Si la asignación ha sido correcta el número de tales definiciones deberá ser igual a la cardinalidad de $Recs_{FU}$, si no será mayor.

(iii) $x = (t_1 \,||\, ... \,||\, t_\lambda)$. Tantas como caminos de comunicación necesite el circuito final. Si la asignación ha sido correcta el número de tales definiciones deberá ser igual a la cardinalidad de $Recs_{MUX}$ mas el número de conexiones punto a punto entre recursos del circuito final.

(iv) $x_1 = \lambda >> x_2$. Una por cada puerto de salida.

(v) $x = \sigma$. Una por cada constante.

(vi) $x_1 = x_2 << \lambda$. Una por cada puerto de entrada.

Cada una de las reducciones se deberá aplicar tantas veces como reusos se hayan especificado mediante la aplicación α.

EJEMPLO 5.9

Siguiendo las directrices de la aplicación α del ejemplo 5.1 y tras aplanar la especificación ecuacional del ejemplo 5.8 alcanzamos la siguiente descripción:

```
body
out = 5 >> t48
t16 = a1
t19 = a2
t22 = b1
t25 = b2
t49 = in << 5
```

```
t48 = t51 - t52
t44 = t57 * t58
t47 = t59 + t60
t34 = # fby t54
t39 = # fby t55
t50 = 0 fby t53
t27 = 0 fby t56
t51 = ( # || # || # || # || t50)
t52 = ( # || # || # || # || t47)
t53 = ( t50 || t50 || t50 || t47 || t50 )
t54 = ( t44 || t34 || t47 || t44 || # )
t55 = ( # || t44 || # || t50 || # )
t56 = ( t27 || t27 || t44 || t27 || t39 )
t57 = ( t25 || t22 || t19 || t16 || # )
t58 = ( t27 || t50 || t27 || t50 || # )
t59 = ( # || # || t39 || t34 || t34 )
t60 = ( # || # || t34 || t49 || t27 )
```

Para conseguirla se han aplicado 5 veces la reducción de *reuso de recursos funcionales*, 2 veces la de *reuso endogámico de retardadores que implementan retardos no arquitectónicos* y 2 veces la de *reuso exogámico de retardadores*. El resto de las reducciones no se han llegado a aplicar. Como puede observarse el número de aplicaciones está en conformidad con los reusos mostrados en la fig. 5.2.

Como resultado el número de definiciones se acerca al número de elementos funcionales del circuito final. Definiciones que se clasifican en:

- 4 definiciones del tipo (i): t34, t39, t50 y t27.
- 3 definiciones del tipo (iii): t48, t44 y t47.
- 10 definiciones del tipo (iii): t51, t52, t53, t54, t55, t56, t57, t58, t59 y t60.
- 1 definición del tipo (iv): *out*.
- 4 definiciones del tipo (v): t16, t19, t22 y t25.
- 1 definición del tipo (vi): t49.

Chequeo de la corrección de la asignación.

Una condición suficiente para demostrar que una asignación es incorrecta, es comprobar que el número de operadores que aparecen en la especificación ecuacional que se obtiene tras la anterior fase, no coincide con la cardinalidad de los codominios de las distintas aplicaciones α. No obstante podrían darse situaciones en que aún siendo iguales, la asignación fuera incorrecta, por ello esta fase se enuncia en base a 4 reducciones que chequean por separado cada uno de los 3 tipos de reusos posibles: de elementos funcionales, de elementos de memoria y de elementos de encaminamiento.

Al igual que en la fase de *chequeo de la corrección de la planificación*, los errores cometidos pueden extraerse estudiando las definiciones que forman la especificación ecuacional resultante. Sin embargo, a diferencia de dicha fase, la especificación ecuacional resultante sí que es directamente implementable sin necesidad de corregirla. Esto es posible ya que tal como se vio con anterioridad, el mecanismo para reproducir formalmente un reuso se basa en generar redundancias dentro de la especificación y hacerlas desaparecer vía la aplicación de la regla de eliminación. Dado que esta regla es correcta, su aplicación sólo transforma la especificación ecuacional si efectivamente la redundancia existe. De este modo, si un reuso es incorrecto nunca llegará a hacerse efectivo, por lo que la especificación ecuacional resultante simplemente describirá un circuito con más hardware del previsto por la asignación incorrecta descrita por α.

Es justamente esta última cualidad la que permite chequear la corrección de una asignación: comprobar que efectivamente para todas las operaciones de la especificación original que debían compartir hardware, solamente existe un único símbolo de operación en la especificación ecuacional obtenida tras la anterior fase, ya que este es el síntoma de que las redundancias se produjeron con éxito y por consiguiente pudieron eliminarse.

Chequeo del reuso de recursos funcionales:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \sigma_1(\ldots), x_2 = \sigma_2(\ldots) \})}{\text{FALLO}} \quad \alpha_{FU}(\sigma_1) = \alpha_{FU}(\sigma_2)$$

Chequeo del reuso de retardadores:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = t_1 \text{ fby } x_1', x_2 = t_2 \text{ fby } x_2' \})}{\text{FALLO}} \quad \alpha_{ST}(\text{fby}_1) = \alpha_{ST}(\text{fby}_2)$$

Chequeo del reuso de caminos de comunicación:

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = \sigma_1(\ldots), x_2 = \sigma_2(\ldots) \})}{\text{FALLO}} \quad \begin{array}{l} \alpha_{cnx}(\sigma_1, i_1) = \alpha_{cnx}(\sigma_2, i_2) \\ \varphi(x_1)[\varepsilon.i_1] \neq \varphi(x_2)[\varepsilon.i_2] \end{array}$$

$$\frac{(\Sigma, X, I, O, \varphi \cup \{ x_1 = c \text{ fby}_1 x, x_2 = \sigma_2(\ldots) \})}{\text{FALLO}} \quad \begin{array}{l} \alpha_{cnx}(\text{fby}_1) = \alpha_{cnx}(\sigma_2, i_2) \\ \varphi(x_2)[\varepsilon.i_2] \neq x \end{array}$$

EJEMPLO 5.10

Un estudio de las definiciones que forman la especificación ecuacional del ejemplo 5.9 permite concluir que la asignación definida en el ejemplo 5.1 era correcta. Efectivamente si dicha especificación se hace pasar por esta fase ninguna de las reducciones se dispara y el algoritmo de verificación puede continuar sin terminar en fallo.

Obsérvese cómo el número de definiciones de tipo (i) y (ii) está en conformidad con el número de elementos de los conjuntos de recursos $Recs_{ST}$ y $Recs_{FU}$ respectivamente. Además el número de definiciones de tipo

(iii) es igual a la cardinalidad de $Recs_{MUX}$ más el número de conexiones punto a punto entre recursos.

Síntesis del encaminamiento.

Esta fase ilustra cómo puede llegar a alcanzarse por derivación especificaciones ecuacionales que describan estructurales puros a nivel RT. En particular aquí se muestran las reducciones para obtener implementaciones que utilicen retardadores para memorizar y multiplexores para interconectar. La formalización de fases de síntesis con modelos de hardware diferentes (utilizando registros, buses, etc) podrían formalizarse de un modo parecido.

Esta fase puede completarse con otras que proyecten operaciones funcionales sobre módulos hardware concretos. A la discusión de dichos aspectos se dedicará el capítulo 6.

Eliminación de multiplexaciones innecesarias:

$$(\Sigma, X, I, O, \varphi \cup \{ x = (x_1 \,||\, \ldots \,||\, x_\lambda) \})$$

$\forall i \in \{1..\lambda\}$, Reemplazo(x, i, x_0)
AplicacionID(x, ε, NINT(λ))
Substitucion(x, ε)
Limpieza(x_1, x_0)

$\forall i, (t_i = \# \vee t_i = x_0)$

Implementación de caminos de comunicación con multiplexores:

$$\frac{(\ \Sigma,\ X,\ I,\ O,\ \varphi \cup \{\ x = (\ x_1 \parallel \ldots \parallel x_\lambda\ )\ \}\ )}{\begin{array}{c}\text{AplicacionID}(\ x,\ \varepsilon,\ \text{MUXI}(\ \lambda,\ \alpha_{ALGN}(x_1),\ \ldots,\ \alpha_{ALGN}(x_\lambda)\ )\ )\\ \text{Expansion}(\ x,\ 1,\ x_0\ )\end{array}} \qquad x_0 \notin X \cup Lu(\Sigma)$$

La primera reducción se aplicará tantas veces como caminos de comunicación dedicados existan, es decir, tantos como caminos que no sean reutilizados por más de un recurso. La segunda reducción se aplicará tantas veces como caminos de comunicación estén compartidos. Esta reducción, además, utiliza la aplicación α_{ALGN} para asignar un puerto particular del multiplexor a cada una de las transferencias de información que confluyen en un recurso.

Tras esta fase el cuerpo ecuacional sólo poseerá definiciones con la siguiente forma:

(i) $x = c$ fby x_1. Tantas como retardadores tenga el circuito final.

(ii) $x = \sigma(\ x_1,\ \ldots,\ x_n\ )$. Tantas como recursos funcionales tenga el circuito final.

(iii) $x = \text{mux}(\ x_1,\ \ldots,\ x_n\ )$. Tantas como multiplexores tenga el circuito final.

(iii) $x = (\ t_1 \parallel \ldots \parallel t_\lambda\ )$. Tantas como secuencias de control sean necesarias para controlar los multiplexores (dado que el modelo de circuito usado utiliza retardadores, no se requieren secuencias de control para el control de carga).

(iv) $x_1 = \lambda >> x_2$. Una por cada puerto de salida.

(v) $x = \sigma$. Una por cada constante.

(vi) $x_1 = x_2 << \lambda$. Una por cada puerto de entrada.

EJEMPLO 5.11

La aplicación de esta fase sobre la especificación ecuacional del ejemplo 5.9, finaliza con la obtención de la siguiente descripción.

```
body
out = 5 >> t48
t16 = a1
t19 = a2
t22 = b1
t25 = b2
t49 = in << 5
t48 = t50 - t47
t44 = t57 * t58
t47 = t59 + t60
t34 = # fby t54
t39 = # fby t55
t50 = 0 fby t53
t27 = 0 fby t56
t53 = mux( t61, t50, t47 )
t61 = ( 0, 0, 0, 1, 0 )
t54 = mux( t63, t34, t44, t47 )
t63 = ( 1, 0, 2, 1, # )
t55 = mux( t64, t44, t50 )
t64 = ( #, 0, #, 1, # )
t56 = mux( t62, t27, t44, t39 )
t62 = ( 0, 0, 1, 0, 2 )
t57 = mux( t65, t25, t22, t19, t16 )
t65 = ( 0, 1, 2, 3, # )
t58 = mux( t66, t27, t50 )
t66 = ( 0, 1, 0, 1, # )
t59 = mux( t67, t39, t34 )
t67 = ( #, #, 0, 1, 1 )
t60 = mux( t68, t49, t34, t27 )
t68 = ( #, #, 1, 0, 2 )
```

Reuso de líneas de control.

Para finalizar mostraré cómo es posible formalizar tareas de más bajo nivel tales como optimización del controlador para reducir el número de secuencias de control que debe generar. Así esta fase reutiliza las líneas de control para controlar más de un multiplexor a la vez.

La idea vuelve a ser la misma utilizada en otras reducciones de reuso: crear redundancias y después eliminarlas. Obsérvese que esta fase también necesitaría ser guiada ya que formaliza un proceso de diseño que afecta al rendimiento del circuito final. Sin embargo, dado que no todas las herramientas de síntesis realizan esta optimización, he optado por describir solamente el proceso de derivación formal sin entrar en detalles de cómo deben especificarse externamente las decisiones de síntesis.

Reuso de líneas de control:

$$\frac{(\ \Sigma, X, I, O, \varphi \cup \{\ x_1 = (\ x_1^1 \parallel \ldots \parallel x_1^\lambda\ ),\ x_2 = (\ x_2^1 \parallel \ldots \parallel x_2^\lambda\ )\ \}\ )}{\begin{array}{c}\forall i \in \{1..\lambda\},\ \text{Reemplazo}(\ x_1,\ i,\ \varphi(x_2)/i\ ) \\ \forall i \in \{1..\lambda\},\ \text{Reemplazo}(\ x_2,\ i,\ \varphi(x_1)/i\ ) \\ \text{Limpieza}(\ x_1,\ x_2\ )\end{array}}$$

EJEMPLO 5.12

Aplicando 3 veces la reducción de *reuso de líneas de control* es posible transformar la especificación del ejemplo 5.11 en la que a continuación se muestra.

```
body
out = 5 >> t48
t16 = a1
t19 = a2
t22 = b1
t25 = b2
t49 = in << 5
t48 = t50 - t47
t44 = t57 * t58
t47 = t59 + t60
t34 = # fby t54
t39 = # fby t55
t50 = 0 fby t53
t27 = 0 fby t56
t53 = mux( t64, t50, t47 )
```

```
t54 = mux( t63, t34, t44, t47 )
t55 = mux( t64, t44, t50 )
t56 = mux( t68, t27, t44, t39 )
t57 = mux( t65, t25, t22, t19, t16 )
t58 = mux( t67, t27, t50 )
t59 = mux( t67, t39, t34 )
t60 = mux( t68, t49, t34, t27 )
t63 = ( 1, 0, 2, 1, # )
t64 = ( 0, 0, 0, 1, 0 )
t65 = ( 0, 1, 2, 3, # )
t67 = ( 0, 1, 0, 1, 1 )
t68 = ( 0, 0, 1, 0, 2 )
```

Si para finalizar aplicamos (manualmente) algunas veces las reglas de substitución y limpieza, es posible llegar a una especificación más compacta igual a la mostrada en el ejemplo 5.1. Recíprocamente si ésta última se aplana, se obtendrá (salvo renombrado y señales de paso) la aquí mostrada.

5.2.2 Estudio de la complejidad temporal de un proceso de síntesis de alto nivel por derivación automática.

En la especificación del algoritmo de guía se han explicitado cada una de las reglas de transformación que deben aplicarse para reproducir formalmente cualquier proceso completo de síntesis de alto nivel. De este modo, si se tienen en cuenta los resultados de los dos estudios de complejidad llevados a cabo hasta el momento (sobre la complejidad de la aplicación de reglas de transformación, §3.2.4 y sobre la complejidad de la generación de las ecuaciones temporales, §4.4.1) es posible conocer la complejidad del proceso de síntesis de alto nivel por derivación automática.

El algoritmo está compuesto por un conjunto de fases que deben ejecutarse secuencialmente, por ello su complejidad será igual a la de la fase

más compleja. A su vez, cada fase está formada por una colección de reducciones que pueden aplicarse en cualquier orden, así la complejidad de una fase podrá ser calculada como el mayor producto de la complejidad de cada reducción por las veces que debe ejecutarse. Por último, cada reducción está compuesta por un chequeo de precondiciones, y una aplicación secuencial de un conjunto de reglas de transformación, de este modo la complejidad de una reducción será la tarea mas compleja de las llevadas a cabo dentro de la misma.

A la hora de aplicar una reducción, es necesario realizar una reflexión previa. En §3.2.4, se vio como la complejidad de la mayor parte de las reglas de transformación dependía linealmente del número de definiciones de la especificación ecuacional, ya que una aplicación concreta se realizaba sobre una definición concreta que era necesario buscar dentro del cuerpo ecuacional. Sin embargo, en cualquier implementación razonable de las reducciones, será necesario realizar una búsqueda de las definiciones que hacen disparar cierta transformación, por lo que a la hora de aplicar una regla de transformación particular el tiempo de búsqueda de la definición sólo deberá ser tenido una vez en cuenta.

Así mismo si se idea un mecanismo de generación de nombres de señal únicos que no colisionen con los símbolos que ocurren en una especificación, no es necesario realizar los chequeos de no colisión de las reglas de renombrado y expansión. Quedando como única regla compleja la de eliminación que sigue requiriendo un recorrido completo de la especificación.

En cualquier caso, el estudio se llevará a cabo en función de los dos factores que independientemente influencian las complejidades de las reglas de transformación y de las ecuaciones temporales, que son:

- $|\varphi_{sim}|$ *número de símbolos de operación de una especificación ecuacional*
- λ *latencia de la planificación*

Es fácil comprobar que el resto de los factores que parecen afectar a la complejidad del sistema de derivación pueden ponerse en función de los anteriores. Así, por ejemplo, pudiera pensarse que el número de transferencias de información es un factor a tener en cuenta, sin embargo, ya que el número de argumentos de una operación es independiente del tamaño de una especificación (y por lo general fijo de un diseño a otro), el número de transferencias crece al mismo ritmo que crece el número de operaciones.

Dado que las fases de *normalización* y de *detección de especificaciones no soportadas* no cumplen funciones propiamente de diseño sino que realizan adaptaciones y chequeos de la especificación original, independientes del proceso de diseño y que pueden ubicarse en el módulo cargador, comenzaré el estudio con la fase de *multiplexación de fuentes*.

La máxima complejidad de la fase de *multiplexación de fuentes* puede llegar a ser $O(|\varphi_{sim}|)$, ya que las complejidades de todas las aplicaciones de cada una de las reducciones que lo forman son:

- **Multiplexación de puertos de entrada**. En el peor caso en que todas las operaciones tomaran como argumentos puertos de entrada, la complejidad de la búsqueda de las definiciones que hacen disparar esta regla sería proporcional al número de operaciones (pudiendo considerar fijo el número de argumentos por operador). Dado que tanto la aplicación de la regla de transformación como la generación de la ecuación temporal IREP tienen complejidades constantes, la complejidad de la aplicación de todas las posibles reducciones de este tipo es $O(|\varphi_{sim}|)$.
- **Multiplexación de constantes**. Utilizando el mismo razonamiento expuesto en la anterior reducción, pero considerando que todos los argumentos fueran constantes, la complejidad de todas las reducciones de este tipo es $O(|\varphi_{sim}|)$.

- **Multiplexación de retardos arquitectónicos**. En el peor de los casos en que todas las operaciones de una especificación ecuacional fueran de retardo, la complejidad nuevamente no sería mayor de $O(|\varphi_{sim}|)$.

Si consideramos que por regla general la latencia de una planificación es menor que el número de operaciones a planificar, la complejidad de la fase de *planificación* no es mayor que $O(|\varphi_{sim}|^2)$, ya que las complejidades de todas las posibles aplicaciones de las reducciones que la forman son:

- **Planificación del ciclo de actualización de los retardos arquitectónicos**. En el peor de los casos en que todas las operaciones de una especificación ecuacional fueran de retardo y suponiendo que el nombre de la nueva variable que se añade no tuviera que chequearse†, la complejidad sería $O(\lambda * |\varphi_{sim}|)$, donde λ se deriva de la complejidad de generación de la ecuación TMT.
- **Planificación de operaciones**. En el peor de los casos en que todas las operaciones se planificaran en el primer ciclo, la ecuación DNEXT debería aplicarse λ-1 veces por cada operación, luego la complejidad sería $O(\lambda * |\varphi_{sim}|)$.
- **Exportación del operador *sample***. Se repite tantas veces como lecturas se realicen y dado que estas lecturas crecen proporcionalmente a las operaciones y que para cada lectura se aplica la regla de eliminación (que requiere recorrer toda la especificación para comprobar que no hay ocurrencias de la señal que elimina), resulta una complejidad de $O(|\varphi_{sim}|^2)$.
- **Extracción del operador *sample***. Dado que la generación de la ecuación DSAM tiene complejidad constante, la complejidad de esta reducción, que se aplica tantas veces como operadores haya, es $O(|\varphi_{sim}|)$.

---

† Si no se pudiera hacer esta asunción sobre la regla de expansión, la complejidad alcanzaría $O(\lambda * |\varphi_{sim}|^2)$

- **Homogeneización de los retardos arquitectónicos**. Se ejecuta tantas veces como retardos haya en la especificación original, y dado que la complejidad de generación de la ecuación ADRET es lineal con la latencia, resulta que en el peor de los casos se alcanzará una complejidad de $O(\lambda*|\varphi_{sim}|)$.

La fase de *separación de acciones RT* puede alcanzar una complejidad de $O(\lambda*|\varphi_{sim}|)$, ya que las complejidades de la aplicación de sus reducciones son:

- **Separación de las acciones de cálculo y de almacenamiento de resultados**. Por aplicar DET (que tiene complejidad lineal respecto a λ) una vez por operación funcional resulta una complejidad de $O(\lambda*|\varphi_{sim}|)$.
- **Separación de las acciones de cálculo y de selección de operandos**. Nuevamente la complejidad es $O(\lambda*|\varphi_{sim}|)$, ya que al poder considerarse el número de argumentos de una operación como una constante, la transformación que influencia la complejidad es la aplicación de DINT, que tiene complejidad lineal respecto a la latencia y que se aplica tantas veces como operaciones tenga la especificación original.

En el peor de los casos la complejidad de la *eliminación del predictor next* es $O(\lambda*|\varphi_{sim}|^2)$ ya que la complejidad de sus reducciones es:

- **Preparación de eliminaciones**. La componente que contribuye a la complejidad de esta reducción es la aplicación de la regla de eliminación que tiene una complejidad lineal respecto al número de símbolos totales de la especificación ecuacional. Sin embargo, al comienzo de esta fase este número no es $|\varphi_{sim}|$ sino que es una cantidad proporcional, en el peor de los casos, al producto entre la latencia (que determina la longitud de las cadenas de *next* o *fby* que aparecen en las definiciones) y el número de lecturas de argumentos (que crecen linealmente con el

número de operaciones de la especificación normalizada). Dado que esta reducción se aplica tantas veces como lecturas, la complejidad de esta reducción puede llegar a ser: $O(\lambda * |\varphi_{sim}|^2)$. En el mejor de los casos en que todas las operaciones se planifiquen en el último ciclo, el número de símbolos se reduce, reduciendo la complejidad total de la reducción a $O(|\varphi_{sim}|^2)$.

- **Eliminación de pares *next-fby***. En el peor de los casos en que todas las operaciones se planificaran en el primer ciclo, generando cadenas de operadores *next* y *fby* de longitud λ-1, esta reducción tendría que realizarse λ veces por cada lectura que se hiciera de un resultado, por lo que resulta una complejidad de $O(\lambda * |\varphi_{sim}|)$.
- **Eliminación de predicciones sobre constantes**. Suponiendo que los argumentos de todas las operaciones fuesen constantes y todas las operaciones se planificasen en el primer ciclo no se tendría una complejidad mayor que $O(\lambda * |\varphi_{sim}|)$
- **Eliminación de predicciones sobre puertos de entrada**. La complejidad nuevamente será como máximo de $O(\lambda * |\varphi_{sim}|)$, aún siendo todos los argumentos de las operaciones puertos de entrada.

La fase de *chequeo de la corrección de la planificación*, al estar formada por una única reducción, posee una complejidad que es función del número de veces que ésta tenga que aplicarse:

- **Chequeo de la corrección de la planificación**. Dado que sólo hay que chequear el primer símbolo de cada definición de la especificación ecuacional (por la forma normalizada que adopta la especificación ecuacional tras la anterior fase), y el número de ellas crece linealmente con el tamaño de la especificación original, la complejidad es $O(|\varphi_{sim}|)$.

La fase de *realimentación de retardos* tiene una complejidad de $O(\lambda^2 * |\varphi_{sim}|)$ según el estudio siguiente:

- **Fragmentación de cadenas de retardos de longitud superior a la latencia**. En el peor de los casos tendrá que aplicarse $|\varphi_{sim}|$ veces la ecuación FRAG cuya complejidad de generación es $O(\lambda)$. Sin embargo, la complejidad de la detección de los parámetros que condicionan la forma del teorema FRAG crecen también linealmente con la latencia. De este modo, la complejidad resulta $O(\lambda^2*|\varphi_{sim}|)$. No obstante, si se utiliza el conocimiento que se tiene de la aplicación τ (que fija la planificación a realizar), dichos parámetros se conocen y la complejidad se reduce a $O(\lambda*|\varphi_{sim}|)$.
- **Reemplazamiento de cadenas de retardos generadas por dependencias de datos**. El número de dependencias de datos crece linealmente con el número de operaciones de la especificación normalizada. Así puede concluirse, siguiendo un razonamiento análogo al anterior, que la aplicación de todas las posibles reducciones de este tipo tiene una complejidad de $O(\lambda^2*|\varphi_{sim}|)$.
- **Reemplazamiento de cadenas de retardos generadas por homogeneización de los retardos arquitectónicos**. Por razonamientos análogos a los anteriores puede concluirse que la complejidad será también $O(\lambda^2*|\varphi_{sim}|)$.

La complejidad que alcanza la fase de *reuso implícito de retardadores* es $O(\lambda*|\varphi_{sim}|^2)$:

- **Reuso implícito endogámico de retardadores que implementan retardos no arquitectónicos**. En el hipotético caso de que se reusaran todos los retardos no arquitectónicos de cada una de las operaciones de la especificación original, la búsqueda de los pares de definiciones a reusar no tendría complejidad mayor de $O(|\varphi_{sim}|^2)$. Además dado que cada reuso implicaría λ reemplazos se obtiene una complejidad igual a $O(\lambda*|\varphi_{sim}|^2)$. Nótese que la búsqueda del ciclo en que cada una de las memorizaciones está planificada no incrementa esta complejidad.

- **Reuso implícito endogámico de retardadores que implementan retardos arquitectónicos**. El mismo razonamiento anterior puede volverse a aplicar obteniendo una complejidad de $O(\lambda*|\varphi_{sim}|^2)$.
- **Reuso implícito exogámico de retardadores**. Nuevamente por la misma razón, todas las aplicaciones de esta reducción pueden alcanzar una complejidad de $O(\lambda*|\varphi_{sim}|^2)$.
- **Reuso implícito de retardadores que implementan fragmentos de retardos arquitectónicos**. El mismo razonamiento permite calcular la complejidad como $O(\lambda*|\varphi_{sim}|^2)$. Donde ahora λ proviene del chequeo de igualdad de las definiciones.
- **Eliminación de las lecturas redundantes de constantes y puertos de entrada**. Aún siendo todas las lecturas redundantes, la complejidad no sería mayor de $O(|\varphi_{sim}|^2)$.
- **Eliminación de lecturas encadenadas**. En el peor de los casos en que todas las operaciones estén encadenadas, para cada definición la regla de reemplazo debe aplicarse λ veces (o generarse 1 vez la ecuación NINT), por lo que la complejidad es $O(\lambda*|\varphi_{sim}|)$.

La fase de *aplanado* tiene una complejidad $O(|\varphi_{sim}|)$:

- **Aplanado**. El número de definiciones que forman el cuerpo ecuacional antes de la aplicación de esta fase es proporcional al número de operaciones de la especificación inicial y al estar formadas todas definiciones por no más de 2 operadores, la complejidad resultante es $O(|\varphi_{sim}|)$.

En la fase de *reuso de recursos* se alcanza nuevamente la complejidad $O(\lambda*|\varphi_{sim}|^2)$, ya que:

- **Reuso de recursos funcionales**. En el caso de que todas las operaciones compartan un único recurso, la búsqueda de las definiciones a reusar tiene complejidad cuadrática respecto a $|\varphi_{sim}|$ ya que el número de ellas crece linealmente con el número de operaciones de la especificación

original. Dado que además, en cada reuso debe aplicarse la regla de reemplazo un número de veces múltiplo de λ, la complejidad resultante es $O(\lambda*|\varphi_{sim}|^2)$.

- **Reuso endogámico de retardadores que implementan retardos no arquitectónicos**. Por el mismo razonamiento anterior la complejidad puede llegar a ser $O(\lambda*|\varphi_{sim}|^2)$.
- **Reuso endogámico de retardadores que implementan retardos arquitectónicos**. La complejidad de todas las posibles aplicaciones de esta reducción también es $O(\lambda*|\varphi_{sim}|^2)$.
- **Reuso exogámico de retardadores**. Nuevamente se tiene una complejidad máxima de $O(\lambda*|\varphi_{sim}|^2)$.
- **Reuso de caminos de comunicación entre recursos funcionales diferentes**. Tiene una complejidad de $O(\lambda*|\varphi_{sim}|^2)$.
- **Reuso de caminos de comunicación entre retardadores y recursos funcionales**. La complejidad puede también llegar a ser $O(\lambda*|\varphi_{sim}|^2)$.

Es claro ver que la complejidad de la fase *chequeo de la corrección de la asignación* nunca será mayor de la complejidad de la anterior fase de hecho, en el peor de los casos, podrá llegar a ser $O(|\varphi_{sim}|^2)$.

- **Chequeo del reuso de recursos funcionales**. En el peor de los casos, es decir, si se optara por comprobar que cada uno de los pares de operaciones presentes en la especificación que no deberían compartir hardware, efectivamente no lo hacen, la complejidad no será mayor de $O(|\varphi_{sim}|^2)$.
- **Chequeo del reuso de retardadores**. Aplicando el mismo razonamiento anterior se concluye que la complejidad es $O(|\varphi_{sim}|^2)$.
- **Chequeo del reuso de caminos de comunicación**. Del mismo modo la complejidad de todas las aplicaciones de esta reducción es $O(|\varphi_{sim}|^2)$.

La fase de *síntesis del encauzamiento* tiene una complejidad de $O(\lambda*|\varphi_{sim}|)$, según lo siguiente:

- **Eliminación de multiplexaciones innecesarias**. Esta reducción es equivalente a la de eliminación de lecturas encadenadas así que su complejidad es $O(\lambda*|\varphi_{sim}|)$.
- **Implementación de caminos de comunicación con multiplexores**. El número de caminos de comunicación es proporcional a $|\varphi_{sim}|$ y la complejidad de generación de MUXI es $O(\lambda)$, luego la complejidad total es $O(\lambda*|\varphi_{sim}|)$.

Dado que la única reducción que compone la fase de *reuso de líneas de control* tiene complejidad $O(\lambda*|\varphi_{sim}|^2)$, esa fase hereda dicha complejidad:

- **Reuso de líneas de control**. Como en otras reducciones de reuso, la complejidad puede alcanzar $O(\lambda*|\varphi_{sim}|^2)$.

En resumen, si suponemos que $\lambda < |\varphi_{sim}|$, el estudio llevado a cabo permite concluir que la complejidad máxima del sistema de síntesis de alto nivel por derivación automática es $O(\lambda*|\varphi_{sim}|^2)$ y si se supone lo contrario es $O(\lambda^2*|\varphi_{sim}|)$.

No obstante, este estudio también permite concluir cómo pueden eliminarse cualquiera de las componentes cuadráticas de las anteriores complejidades. En el primer caso, la componente $|\varphi_{sim}|^2$ (que puede encontrarse en las reducciones de *exportación del operador sample*, de *preparación de eliminaciones* y en todas las reducciones que forman las fases de *reuso*), se deriva de la complejidad lineal respecto a $|\varphi_{sim}|$ de la *regla de eliminación*. Así, si se consigue que la comprobación de no ocurrencia de cierta señal dentro del cuerpo ecuacional tenga complejidad constante (vía contadores de ocurrencia) el término cuadrático desaparece.

Por otro lado, la componente λ^2 (que aparece en algunas de las reducciones que forman la fase de *realimentación de retardos*) se basa en la necesidad de extraer de las propias definiciones los parámetros característicos de las ecuaciones a aplicar, extracción que tiene complejidad lineal respecto de λ. Si en lugar de llevar a cabo esta extracción se utiliza, al

igual que en la fase de planificación, la información de la aplicación α, dicha complejidad se hace constante y el término cuadrático desaparece (nótese que la no extracción de parámetros no afecta a la corrección del sistema ya que en el peor de los casos intentará aplicar una ecuación no aplicable y el resultado será que la definición no se modifica).

En cualquier caso obsérvese que ningún algoritmo de síntesis de alto nivel que obtenga soluciones razonables tiene una complejidad inferior a cuadrática†, por lo que la fase de verificación formal de sus resultados, según el esquema propuesto, nunca recargará el ciclo de diseño.

Por otro lado, una última reflexión sobre el límite mínimo de complejidad alcanzable por un enfoque transformacional puramente simbólico (como lo es el presentado). Este límite será en el mejor de los casos $O(\lambda * |\varphi_{sim}|)$. Por un lado, la dependencia lineal respecto de $|\varphi_{sim}|$ es natural ya que al crecer el número de operaciones, crece el número de definiciones a procesar y por consiguiente el tiempo de manipularlas. Por otro lado, la dependencia lineal respecto de λ se extrae de que el resultado de una planificación se representa en el formalismo propuesto mediante operadores *interleave* con un número de argumentos igual a la latencia, así a mayor latencia mayores son los términos que forman las definiciones.

5.2.3 Comprobación experimental del estudio teórico de la complejidad.

Para comprobar la viabilidad del algoritmo de guiado automático propuesto con anterioridad y poder contrastar experimentalmente la complejidad teórica

† Por ejemplo, las planificaciones por listas pueden alcanzar complejidades cuadráticas y las planificaciones por fuerzas, complejidades cúbicas.

calculada, diseñé un nuevo prototipo† (extendiendo el descrito en §5.1), para que aparte del cargador, del núcleo de transformación, del generador de ecuaciones y del interfaz de usuario, incluyese un módulo controlador que dirigiese al sistema para realizar SAN formal.

Dada la dependencia lineal respecto a λ y cuadrática respecto a $|\varphi_{sim}|$ que tiene la complejidad del sistema de síntesis formal, comencé llevando a cabo 2 series de experimentos para detectar cada una de estas componentes por separado. En ambas series utilicé colecciones de filtros de 2do. orden en paralelo, que como ya se ha podido comprobar, cada uno de ellos está compuesto de 1 entrada, 1 salida, 4 constantes y 10 operaciones (2 retardos arquitectónicos, 4 multiplicadores, 3 sumadores y 1 restador).

El primer experimento estaba orientado a medir cómo crecía el tiempo†† de diseño formal en relación al crecimiento del tamaño de la especificación. Para ello, manteniendo la latencia de la planificación fija, iba incrementando el número de etapas de 2do. orden que formaban la especificación a diseñar (lo que se traducía en incrementos de 10 operaciones). Se realizó para especificaciones que poseían desde 10 hasta 200 operaciones y para 4 latencias de planificación fijas (4, 8, 16 y 32 ciclos). Los resultados se muestran gráficamente en la fig. 5.3 y tabularmente a continuación.

	latencia de la planificación			
núm. de operaciones	4 ciclos	8 ciclos	16 ciclos	32 ciclos
10	0,1 s	0,2 s	0,5 s	1,4 s
20	0,3 s	0,6 s	1,3 s	3,4 s

† Utilizando PROLOG y con un coste de desarrollo aproximado de 1 hombre/mes y de un tamaño aproximado de 170 predicados y 1350 líneas de código (repartidas en, generador de ecuaciones: 19%, núcleo de transformación: 15%, algoritmo de control: 52% y resto: 14%).

†† Ejecutando el código sobre una máquina Ultra-Sparc.

30	0,6 s	1,1 s	2,3 s	6,0 s
40	1,0 s	1,8 s	3,6 s	8,9 s
50	1,5 s	2,6 s	5,2 s	12,4 s
60	2,0 s	3,5 s	7,0 s	16,3 s
70	2,7 s	4,7 s	9,2 s	20,7 s
80	3,4 s	5,9 s	11,4 s	25,8 s
90	4,2 s	7,4 s	14,0 s	31,3 s
100	5,1 s	8,9 s	16,8 s	37,3 s
110	6,1 s	10,6 s	20,0 s	43,7 s
120	7,2 s	12,5 s	23,4 s	51,1 s
130	8,4 s	14,5 s	27,1 s	58,2 s
140	9,6 s	16,6 s	31,1 s	66,3 s
150	11,0 s	19,0 s	35,2 s	74,8 s
160	12,4 s	21,4 s	39,8 s	83,6 s
170	14,0 s	24,0 s	44,3 s	92,9 s
180	15,6 s	26,9 s	49,7 s	102,6 s
190	17,1 s	29,7 s	54,2 s	112,8 s
200	18,7 s	32,6 s	59,1 s	121,0 s

Puede parecer que los tiempos de ejecución son excesivamente largos para problemas de complejidad media (200 nodos $\approx$ 2 min.)†, pero téngase en cuenta que no se está evaluando el rendimiento del sistema sino su complejidad, es decir, como crece el tiempo de ejecución conforme crece el tamaño del problema. El rendimiento siempre puede mejorarse realizando implementaciones más óptimas o ejecutándolas sobre máquinas más potentes, la complejidad solo se mejora cambiando el algoritmo. No obstante la explicación de los altos tiempos de ejecución están justificados ya que se obtuvieron de una ejecución interpretada de un programa PROLOG, una implementación en un lenguaje procedural compilado obtendría resultados notablemente mejores.

† Que extrapolando, por ejemplo, para 2000 nodos tardaría unas 3 horas y para 20000 nodos unos 14 días.

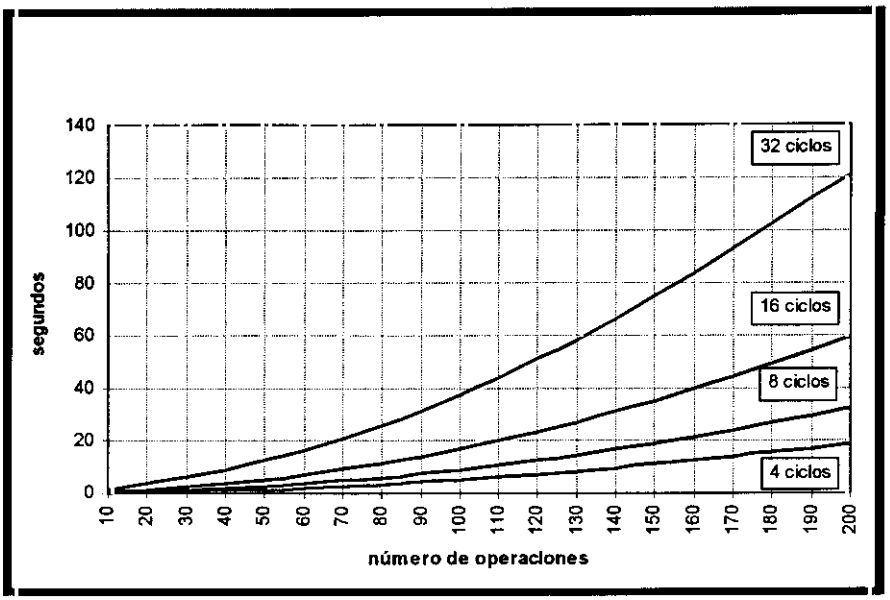


Fig. 5.3: Efecto del tamaño de la espec.

En cualquier caso, una vez finalizados los experimentos se efectuó un análisis de regresión para comprobar qué correlación verificaban los resultados. De este modo, pudo concluirse que cada una de las curvas obtenidas a latencia fija, tiene crecimiento cuadrático respecto del número de operaciones que formen la especificación original.

El objetivo del segundo experimento fue medir el crecimiento del tiempo de diseño formal respecto al crecimiento de la latencia de la planificación. Para ello, una especificación con un número fijo de operaciones, se diseñó repetidas veces con distintas longitudes de planificación. El experimento se realizó con especificaciones formadas por 40, 80 y 160 operaciones y cada una de ellas se diseñó bajo panificaciones que variaban entre 4 y 80 ciclos en intervalos de 4 ciclos. Los resultados se muestran gráficamente en la fig. 5.4. y tabularmente a continuación.

número de ciclos	tamaño de la planificación		
	40 ops.	80 ops.	160 ops.
4	1,1 s	3,3 s	12,7 s
8	1,7 s	5,6 s	21,2 s
12	2,6 s	7,8 s	29,9 s
16	3,6 s	10,5 s	38,8 s
20	4,5 s	13,7 s	50,1 s
24	5,8 s	18,0 s	61,7 s
28	7,3 s	21,9 s	74,1 s
32	8,9 s	25,4 s	86,7 s
36	10,6 s	30,2 s	98,8 s
40	12,7 s	34,9 s	114,3 s
44		40,3 s	123,3 s
48		45,4 s	142,8 s
52		51,1 s	157,1 s
56		55,9 s	172,9 s
60		61,2 s	194,5 s
64		70,7 s	208,6 s
68		74,7 s	228,7 s
72		85,0 s	245,6 s
76		91,5 s	257,1 s
80		98,7 s	272,3 s

Nuevamente el análisis de regresión de los resultados fue concluyente: las curvas a número de operaciones fijo, verifican un crecimiento casi lineal (la componente cuadrática es casi nula) respecto de la latencia de la planificación con que se diseñen.

Para finalizar, se realizó un tercer experimento para poner de manifiesto algo sutilmente oculto en el primero: cómo afectaba a los tiempos de diseño formal, la 'forma' de la planificación. Dado que el formalismo presentado es simbólico y el proceso de diseño se realiza por derivación, es natural que éstos tiempos de diseño se vean afectados por el número de símbolos de la especificación que manipulan. El efecto del número de símbolos de la

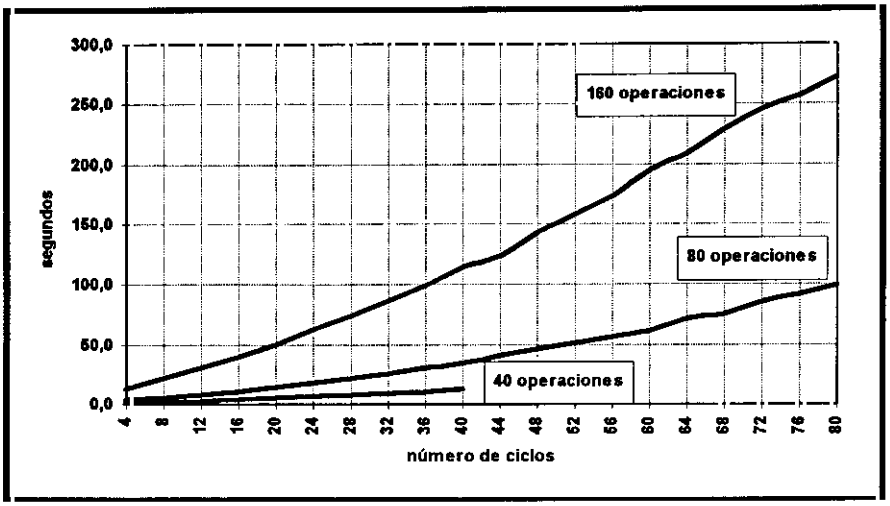


Fig. 5.4: Efecto de la latencia.

especificación original ya fue medido en el primer experimento, sin embargo, el número de símbolos que tienen las especificaciones intermedias, depende fuertemente del tipo de planificación realizada.

Si se estudia como se formaliza la asignación de una operación a un ciclo (mediante cadenas de operadores *fby* y *next* de igual longitud que la latencia menos el ciclo en que se planifique) podrá comprobarse que cuanto mayor sea el número de nodos planificados en ciclos tempranos, mayor número de símbolos se crearán durante el proceso de diseño, y cuanto mayor sea el número de nodos planificados en ciclos tardíos, menor número de símbolos serán necesarios. Así, en el límite, los tiempos mínimos de diseño se obtendrán con planificaciones que encadenen todas las operaciones en el último ciclo, y los tiempos máximos de diseño se obtendrán con planificaciones que encadenen todas las operaciones en el primer ciclo.

De este modo el tercer experimento muestra, para especificaciones con número variable de operaciones (de 10 a 200) y para una latencia de planificación fija (32 ciclos), la relación entre los tiempos máximos y mínimos

de planificación, mostrando también el tiempo de una planificación ASAP. Obviamente, la planificación ASAP se encuentra más cerca del máximo que del mínimo ya que, al realizarse sobre etapas de 2do. orden en paralelo, planifica todas las operaciones en los 4 primeros ciclos por ser, en cualquier caso, la longitud del camino crítico igual a 4. Los resultados se muestran gráficamente en la fig. 5.5 y tabularmente a continuación.

	'forma' de la planificación		
num. de operaciones	máximo	ASAP	mínimo
10	1,8 s	1,4 s	0,1 s
20	4,1 s	3,4 s	0,3 s
30	7,1 s	6,0 s	0,5 s
40	10,5 s	8,9 s	0,8 s
50	14,6 s	12,4 s	1,1 s
60	19,0 s	16,3 s	1,5 s
70	24,3 s	20,7 s	2,0 s
80	29,9 s	25,8 s	2,6 s
90	36,1 s	31,3 s	3,2 s
100	42,8 s	37,3 s	3,9 s
110	50,3 s	43,7 s	4,7 s
120	58,3 s	51,1 s	5,5 s
130	67,0 s	58,2 s	6,4 s
140	76,4 s	66,3 s	7,3 s
150	85,4 s	74,8 s	8,4 s
160	96,2 s	83,6 s	9,4 s
170	106,3 s	92,9 s	10,6 s
180	119,2 s	102,6 s	11,8 s
190	129,6 s	112,8 s	13,1 s
200	142,8 s	121,0 s	14,4 s

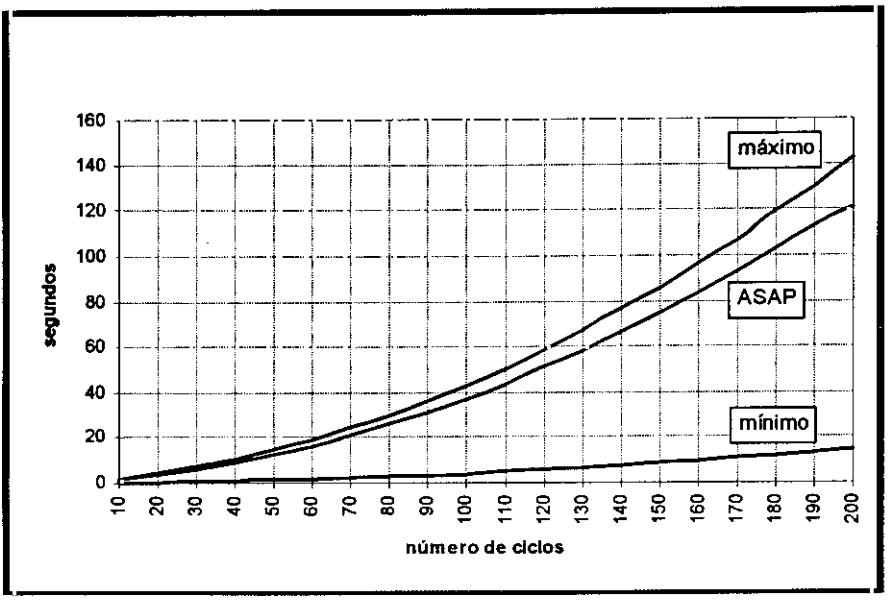


Fig. 5.5: Efecto del tipo de planificación.

5.2.4 Reflexiones sobre el alcance práctico de un sistema de síntesis de alto nivel por derivación automática.

Como se ha podido comprobar, es posible concebir sistemas de síntesis formal tales que su uso, no requiera conocimiento alguno de sus bases matemáticas. Esto se ha conseguido reduciendo al mínimo el número de decisiones que deben tomarse para controlar un proceso de diseño por derivación. Según el esquema mostrado, es posible diseñar un algoritmo de baja complejidad, que tomando como entrada sólo aquellas alternativas que tienen efecto directo sobre el rendimiento del circuito (que no son otras que las que debe resolver un diseñador o un algoritmo de síntesis convencional), sea capaz de encontrar automáticamente un camino de derivación que ligue la especificación ecuacional de nivel algorítmico, con la especificación

ecuacional de nivel RT que describe el circuito sintetizado. Este circuito, a diferencia del obtenido por otros medios, será correcto y la prueba de su corrección es el propio proceso de síntesis por derivación, es decir, será un circuito que nace formalmente verificado.

Por otro lado, se ha propuesto un método seguro para desarrollar herramientas formales que sean capaces de abordar los diferentes problemas incluidos en la síntesis conductual. Este método se sustenta sobre 3 pilares: primero, formalización del conocimiento de la técnica de diseño como un conjunto de ecuaciones; segundo, demostración de la corrección de dichas ecuaciones; y tercero, búsqueda de un orden de uso de las reglas de transformación que permita aplicar convenientemente sobre una especificación ecuacional las ecuaciones demostradas. Como puede observarse, este método de desarrollo, si bien necesita algunos conocimientos del soporte matemático, no requiere que el diseñador de herramientas formales elabore una teoría distinta para cada problema, sino que simplemente aplique una y otra vez el mismo esquema formal, un esquema que, como se comprobó en §3.3 y en §4.5, es suficientemente versátil.

De hecho, el sistema presentado en este capítulo no puede considerarse como un sistema universal para la síntesis formal de alto nivel, sino que se ha adaptado para formalizar un conjunto relativamente general de algoritmos de síntesis. Es decir, podrá ocurrir que el sistema presentado requiera más información del proceso de diseño del que una herramienta convencional le pueda ofrecer y no será porque el sistema formal necesite un control excesivo, sino porque el algoritmo de síntesis convencional no es capaz de diferenciar y optimizar algunos aspectos que el sistema formal sí es capaz de resolver. Por otro lado, pudiera también suceder lo contrario: que existan optimizaciones que el sistema formal no solicita y que el algoritmo convencional realiza. Sin embargo, esto no se debe a que el soporte formal

sea incapaz de prever dicha optimización, sino a que por simplificar el algoritmo de control, éste se ha diseñado para que efectúe dichas tareas de un modo rígido e implícito. Ilustraré mediante ejemplos estos dos comentarios.

El sistema formal presentado, por ejemplo, solicita información sobre el ciclo en que debe planificarse cada una de las operaciones de actualización de los registros arquitectónicos y el modo en que deben reusarse éstos para almacenar valores temporales. Asimismo, requiere información sobre el modo de asociar los distintos valores que lee un recurso funcional, con las entradas de los multiplexores a los que se conecta (esto lo hace para reutilizar las entradas libres de los multiplexores o para reducir el número de secuencias de control diferentes). Del mismo modo también se requiere conocer si se reusan dichos multiplexores entre distintos recursos funcionales o incluso si se fusionan en una, varias líneas de control que siguen secuencias compatibles.

No todos los sistemas de síntesis son capaces de optimizar en todos estos frentes. Así existen sistemas que no permiten la existencia de retardos arquitectónicos o que asignan elementos de memoria dedicados para almacenar este tipo de variables. Existen sistemas en los que el orden de las entradas de datos en los multiplexores no se optimiza, sino que viene dado por el orden en que han sido planificadas las operaciones que generan los operandos. Y existen sistemas que no consideran la optimización del controlador como una fase de SAN. Incluso es posible llegar más lejos: existen sistemas que no soportan encadenamiento de operaciones, por lo que muchas de las reducciones que forman el sistema de síntesis formal presentado no llegan nunca a aplicarse.

En relación al otro aspecto a ejemplificar, debe comentarse que a la hora de diseñar el sistema formal se han restringido explícitamente el número de decisiones de diseño a controlar. Por ejemplo, se ha asumido que todo

resultado intermedio que deba almacenarse durante más de un ciclo se memoriza en un único recurso, cuando en realidad pudiera repartirse entre varios de ellos y en distintos ciclos estar dicho valor presente a la salida de distintos registros. Asimismo, se ha asumido que un mismo valor, independientemente del número de veces que sea leído, de los ciclos en que se haga dicha lectura, y de quién sea el que la realiza, está siempre almacenado en un único elemento de memoria, cuando en realidad podría estar simultáneamente en varios de ellos. Igualmente, se ha adoptado que el modelo de interconexionado esté basado en multiplexores (en lugar de en buses o de un esquema mixto), que los elementos de memoria sean todos retardadores (en lugar de registros o de una arquitectura que tenga elementos de distinto tipo) y que el reuso sólo sea posible entre operaciones funcionalmente equivalentes (cuando, como veremos en el capítulo 6, el soporte formal permite reusos entre operaciones que poseen funcionalidades compatibles).

En general y tal como se comprobó en anteriores capítulos, el formalismo ecuacional es capaz de representar infinidad de posibilidades, serán los algoritmos de control los que limiten el conjunto de posibilidades alcanzable. Pero, en cualquier caso, la conclusión que debe extraerse es que existen buenas razones para pensar que siempre será posible idear un esquema de derivación automático que, bajo un conjunto mínimo de parámetros, pueda reproducir cualquier técnica de diseño de alto nivel.

Capítulo 6

Especificación de dominios, operadores, representaciones y bibliotecas de componentes hardware

Mathematicians first used the sign $\sqrt{-1}$ without in the least knowing what it could mean, because it shortened work and led to correct results. People naturally tried to find out why this happened and what $\sqrt{-1}$ really meant. After two hundred years they succeeded.
W.W. Sawyer

Cabría pensar que tras el anterior capítulo en el que se mostraba cómo reproducir formal y automáticamente todas y cada una de las tareas de un proceso de síntesis de alto nivel convencional, no queda otra opción que exponer las conclusiones y finalizar. Sin embargo, el proceso de síntesis formalizado es excesivamente simplista ya que se limita (al igual que la mayor parte de los sistemas automáticos actuales) a un mero cambio del modelo temporal de un cálculo o, desde la perspectiva de la herramienta, a repartir

símbolos entre un conjunto de ciclos de manera que se respeten ciertas dependencias y a reducir el número de símbolos iguales si éstos han sido repartidos entre ciclos distintos.

Para recordar cómo ha sido esto posible y para comprender sus deficiencias, debe analizarse en perspectiva la trayectoria seguida en los capítulos precedentes. En un comienzo se supuso la existencia de cierta álgebra estática a la que se asociaba una cierta signatura Σ. Dicha Σ-álgebra se extendía sucesivamente hasta alcanzar la $Lu(\Sigma)$-álgebra con la que se construía la semántica del formalismo de especificación ecuacional. Mediante la anterior signatura era posible definir $Lu(\Sigma)$-ecuaciones que, vía las reglas de aplicación, permitían transformar la especificación ecuacional de manera que conservara el comportamiento si y solo si, la ecuación era válida en el álgebra original. A continuación, se demostró un conjunto de ecuaciones que eran correctas para toda álgebra de partida y que formalizaban todas las fases de un proceso de síntesis de alto nivel convencional. Gracias a ellas pudo desarrollarse un sistema completamente simbólico que, ignorando el significado real de los símbolos que manipulaba (significado que dependía del álgebra soporte), era capaz de hacer planificación y reuso con seguridad.

Sin embargo, si se deseara realizar otro tipo de transformaciones que no fueran estrictamente temporales, el papel del álgebra soporte sería ineludible y dado que dicha álgebra es totalmente ajena a la herramienta (que sólo conoce su signatura), no quedaría otra alternativa que condicionar la corrección de los resultados al buen hacer del diseñador, confiando en que éste sólo aplique ecuaciones válidas.

Así, el propósito de este capítulo será encontrar un mecanismo para que la herramienta pueda conocer de algún modo los objetos que está manipulando, y pueda verificar por sí misma que las ecuaciones que aplica son ciertas. Este mecanismo deberá permitir construir un modelo simbólico isomorfo al álgebra que el diseñador tiene en su cabeza, de manera que todo

razonamiento abstracto pueda reproducirse en el computador, mediante razonamiento simbólico.

Los tres tipos de objetos que deberán quedar descritos serán los que debería distinguir un sistema de síntesis: tipos abstractos de datos (dominios y operadores), bibliotecas de componentes hardware y representaciones. Los dos primeros serán modelados mediante álgebras y los últimos mediante homomorfismos entre álgebras.

Para ello, este capítulo presentará el concepto de **especificación algebraica** como un método para la formalización simbólica (vía ecuaciones) de la semántica de los géneros y símbolos de operación incluidos en una signatura (§6.1). Presentará el concepto de **implementación algebraica** como un método para la formalización (también vía ecuaciones) de las funciones de abstracción de tipos (§6.2). Discutirá los métodos de prueba de validez de ecuaciones, apuntando las posibles vías de automatización para su inclusión en el sistema de síntesis formal (§6.3) y finalizará mostrando una colección de técnicas de diseño que pueden ser reproducidas gracias a los nuevos conceptos introducidos (§6.4).

Debe destacarse, que la mayor aportación de este capítulo es la integración que hace de un conjunto de técnicas ya existentes, pero dispersas. Un conjunto de técnicas, que reunidas para el diseño de hardware, ofrecen un método uniforme para solucionar algunos problemas de los que adolecen las herramientas actuales de síntesis y que no parecen tener una conexión evidente entre sí: bajo nivel de abstracción de las operaciones de la especificación (generalmente *bit-true*), inconsistencias semánticas en el tratamiento de los datos, bajo grado de reuso de los módulos hardware cuando son ligeramente diferentes, etc. Debe también mencionarse que detrás de dicha recopilación se esconde una gran cantidad de desarrollo que por sí solo merecería incluirse en esta memoria. Sin embargo (por motivos de espacio) he optado simplemente por apuntar las principales conclusiones y

por reseñar las nociones de la metodología adoptada para que dicho desarrollo pueda ser reproducido por el lector interesado.

6.1 Especificación de tipos abstractos de datos.

Conceptualmente un comportamiento se caracteriza tanto por un algoritmo, como por los tipos de datos que utiliza en él. Estos tipos de datos determinan qué objetos manipula (dominios) y qué operaciones pueden utilizarse para manipularlos (operadores). Por su parte, el algoritmo describe cómo componer los anteriores operadores para realizar cierto cálculo. A la manera de especificar algoritmos, se dedicó el capítulo 2 al completo. Esta sección, por tanto, se dedicará a presentar un método para especificar tipos de datos que sea, además, compatible con el método de especificación ecuacional que se propuso entonces.

Comencemos concretando los términos. Un **tipo de datos** es una colección de dominios y de operaciones sobre dichos dominios, tal que todo objeto de datos pueda ser generado (o alcanzado si se prefiere) a partir de cierto conjunto de objetos básicos mediante la aplicación repetida de las operaciones, que es al fin y al cabo, la única manera de que puedan ser creados por un sistema físico, ya sea software o hardware. Dichas operaciones no sólo permiten obtener unos objetos de datos a partir de otros, sino que también, permiten organizar y estructurar los dominios que relacionan por lo que, desde un punto de vista matemático, todo tipo de datos puede modelarse mediante un álgebra†.

Obsérvese que lo contrario no es cierto, es decir, toda álgebra no es un tipo de datos, ya que una exigencia de esta definición es que los dominios sean numerables para que todos los objetos puedan ser alcanzados por un

† Asunción que ya fue utilizada en el capítulo 2.

proceso finito de cálculo, y es claro que existen muchas álgebras que no lo son (por ejemplo, los números reales).

A partir de esta definición de tipo de datos, es posible proponer una noción algo más abstracta que generaliza la anterior y que recoge lo que la comunidad investigadora entiende por tipo abstracto de datos [LiZi74][GoTW78].

6.1 **DEFINICIÓN.** Un **tipo abstracto de datos** es una clase de tipos de datos cerrada bajo renombrado de dominios, objetos y operaciones, que es independiente de la representación que adopte cada uno de los anteriores elementos.

Una vez conocido lo que se pretende especificar, esta sección comenzará definiendo en §6.1.1 un método algebraico para hacerlo, seguidamente se dedicará §6.1.2 al estudio de la corrección de las especificaciones y §6.1.3 a la propuesta de un método sistemático de especificación. En §6.1.4, se comprobará la utilidad práctica de la propuesta para la especificación de los tipos de datos implicados en un proceso de síntesis de alto nivel (tipos de nivel RT como bit y vectores de bit y tipos de nivel algorítmico como naturales, enteros, etc.) y para la especificación de bibliotecas de componentes hardware. Finalmente en §6.1.5, se incorporará al mecanismo de especificación ecuacional.

6.1.1 Especificación algebraica de tipos abstractos de datos.

La idea en que se basa el enfoque algebraico para la especificación de tipos abstractos de datos consiste en describirlos facilitando únicamente los nombres de los diferentes dominios, los nombres de las distintas operaciones y las propiedades que caracterizan el comportamiento de estas últimas. Los nombres de dominios y operaciones se describen vía signaturas heterogéneas (véase definición 2.7) y las propiedades se establecen mediante una

colección de ecuaciones (véase definición 2.12) que deben ser invariantes bajo cualquier representación del tipo.

6.2 DEFINICIÓN. Una **especificación algebraica de un tipo abstracto de datos**, $SPEC \equiv (S, \Sigma, E)$ consiste en una signatura (S, Σ) y un conjunto de Σ-ecuaciones, E.

Como puede observarse es nuevamente un método de especificación no procedural y, dado que está basado en signaturas y ecuaciones, podrá ser incorporado con facilidad al mecanismo de especificación ecuacional, a la vez que podrá adaptarse sin problemas al sistema de síntesis formal desarrollado.

En la literatura esta noción puede ser algo más general. Así, es posible encontrar, bajo el mismo nombre, otros tipos de especificación no basados en lógicas ecuacionales, sino en otras lógicas más generales como lógicas ecuacionales por cláusulas de Horn [Pada88][EhMa90], lógicas de primer orden completas o lógicas ecuacionales de orden superior [Poig86]. Sin embargo, he preferido adoptar este método (que algunos autores denominan **especificación ecuacional de tipos abstractos de datos**) por ser simple, por ser con diferencia el más utilizado y por ser lo suficientemente expresivo para una primera aproximación con propósitos de diseño hardware.

Al igual que en el capítulo 2, el propósito de la signatura que aparece en la anterior definición es dar soporte sintáctico a cierto modelo algebraico, proporcionando un medio de dar nombre a los elementos de los dominios vía términos cerrados (los términos abiertos, recuérdese, eran maneras genéricas de expresar subconjuntos de dichos dominios alcanzados tras cierto cálculo). Sin embargo, dado que una signatura era capaz de dar soporte sintáctico a una gran variedad de álgebras distintas de la que se desea especificar (véase ejemplo 2.6), la adición de un conjunto de ecuaciones, junto con la noción de satisfacción (véase definición 2.13) establece un mecanismo simple para limitar sintácticamente el número de álgebras que deben considerarse: sólo se aceptarán por especificadas aquellas álgebras que satisfagan el conjunto

completo de propiedades descritas por las ecuaciones.

6.3 DEFINICIÓN. Sea una especificación algebraica *SPEC* = (*S*, Σ, *E*). Se dice que una Σ-álgebra es una ***SPEC*-álgebra** si satisface todas las ecuaciones que forman *E*. La clase de todas las *SPEC*-álgebras de cierta especificación se notará por Alg_{SPEC}.

Para poder describir especificaciones algebraicas en los ejemplos, ampliaré la notación establecida en el capítulo 2 para la declaración de signaturas, con la palabra reservada **equations** tras la cual se listarán los símbolos de variable y las ecuaciones que formen la especificación algebraica. Para la declaración de variables se usará la convención **for all** *x* : *s*, donde *x* será una lista de identificadores de variable y *s* el género de cada una de ellas. Dicha lista de variables podrá ser compartida por todas las ecuaciones.

EJEMPLO 6.1

Realicemos una primera especificación algebraica muy simple del conjunto de los booleanos. En la signatura declaramos dos operaciones constantes para expresar los dos valores lógicos y una operación monaria para denotar la negación. A esto añadimos dos ecuaciones que establecen la tabla de verdad de esta última operación.

BOOLEANOS ≡
sorts
Bool
operations
cierto : → Bool
falso : → Bool
no : Bool → Bool
equations
no(cierto) = falso *(eq1)*
no(falso) = cierto *(eq2)*

Según la anterior definición podrán ser *BOOLEANOS*-álgebras cualquiera de las siguientes $\Sigma_{BOOLEANOS}$-álgebras:

- (B, t, f, ¬), es decir, el conjunto de los *booleanos* como soporte del

género *Bool*, *cierto lógico* como soporte del símbolo *cierto*, *falso lógico* como soporte del símbolo *falso* y la *negación booleana* como soporte del símbolo *no*, ya que este modelo satisface *eq*1 y *eq*2:

$$\text{no(cierto)}^{B} = \neg\, t = f = \text{falso}^{B}$$

$$\text{no(falso)}^{B} = \neg\, f = t = \text{cierto}^{B}$$

- (Z, -1, 1, -), es decir, el conjunto de los *enteros* como soporte del género *Bool*, *menos uno entero*, como soporte del símbolo *cierto*, *uno entero* como soporte del símbolo *falso* y la *negación entera* como soporte del símbolo *no*, ya que esta álgebra también satisface *eq*1 y *eq*2:

$$\text{no(cierto)}^{Z} = -(\,-1\,) = 1 = \text{falso}^{Z}$$

$$\text{no(falso)}^{Z} = -(\,1\,) = \text{cierto}^{Z}$$

No obstante, obsérvese que este modelo posee infinitos elementos que no tienen ninguna representación sintáctica, ya que la interpretación de cualquier término será 1 ó -1. Cuando un modelo posee este tipo de elementos se dice que es **no accesible**.

- $X \equiv (\,\{\,u\,\},\, u,\, u,\, id\,)$, es decir, el conjunto $\{\,u\,\}$ como soporte del género Bool, *u* como soporte del símbolo *cierto*, *u* como soporte del símbolo *falso* y la *función identidad* como soporte del símbolo *no*, ya que, nuevamente este modelo satisface ambas ecuaciones:

$$\text{no(cierto)}^{X} = id(\,u\,) = u = \text{falso}^{X}$$

$$\text{no(falso)}^{X} = id(\,u\,) = u = \text{cierto}^{X}$$

Sin embargo, este modelo vuelve a ser anómalo (desde el punto de vista de la intuición) ya que esta álgebra satisface más ecuaciones entre términos cerrados de las que estrictamente aparecen en la especificación. Por ejemplo la ecuación *cierto* = *falso* es también válida, ya que:

$$\text{cierto}^{X} = u = \text{falso}^{X}$$

Cuando un modelo presenta esta característica se dice que no está **libre de confusión**.

En cualquier caso, la adición de un conjunto de ecuaciones a la signatura permite no aceptar cualquier Σ-álgebra sino sólo aquellas que cumplan ciertas propiedades. Por ello el modelo (**N**, 1, 0, +1), es decir, el conjunto de los *naturales* como soporte del género *Bool*, *uno natural*, como soporte del símbolo *cierto*, *cero* como soporte del símbolo *falso* y la *operación incremento* como soporte del símbolo *no*, no puede ser una *BOOLEANOS*-álgebra por no cumplir las ecuaciones de la especificación:

$$\text{no}(\text{ cierto })^{\mathbf{N}} = (1) + 1 = 2 \neq \text{falso}^{\mathbf{N}}$$
$$\text{no}(\text{ falso })^{\mathbf{N}} = (0) + 1 = 1 = \text{cierto}^{\mathbf{N}}$$

Como se ha podido constatar en el anterior ejemplo, incluso con una selección adecuada de las ecuaciones que forman una especificación algebraica *SPEC*, es posible que existan muchas *SPEC*-álgebras distintas que no cumplan las condiciones de ser un tipo abstracto de datos o que simplemente no sean el modelo que se pretende especificar. Por ello, para evitar ambigüedades en la interpretación de una especificación algebraica, será necesario restringir aún más su semántica.

Según esto, de ahora en adelante, se adoptará siempre el modelo inicial. Este modelo o álgebra, permitirá dejar fuera modelos lejanos a la intuición como los mostrados en el anterior ejemplo, ya que cumple las condiciones de ser generado por sus operaciones, es decir, de no contener más elementos de los que pueden obtenerse mediante la interpretación de términos, y de ser típico, es decir, de satisfacer una ecuación cerrada si y sólo si dicha ecuación es válida en toda *SPEC*-álgebra (nuevamente se evidencia la importancia de los términos cerrados, por el papel que desempeñan representando sintácticamente a los objetos que forman el tipo abstracto de datos).

Como a continuación se definirá, el modelo inicial buscado es el álgebra cociente del conjunto de términos cerrados T_Σ construida según la relación de

equivalencia inducida por las ecuaciones de la especificación.

6.4 DEFINICIÓN. Sea una especificación algebraica *SPEC* = (*S*, Σ, *E*). Se llama **equivalencia de términos cerrados** a la relación *S*-indexada, ~, que se define para todo género $s \in S$ y para todo par de términos $t_1, t_2 \in T_{\Sigma,s}$ como:

$t_1 \sim t_2$, si y sólo si para toda *SPEC*-álgebra, *A*, se cumple que $t_1^A = t_2^A$

Obviamente es una relación de equivalencia que está definida únicamente sobre el conjunto de términos cerrados.

Para disponer de un criterio operacional que permita determinar de un modo más claro si dos términos cerrados son equivalentes, resulta útil conocer que la relación anteriormente definida, además de las propiedades de reflexividad, simetría y transitividad, cumple:

- $\forall \sigma \in \Sigma,\ \forall t_i \in T_\Sigma,\ t_1 \sim t_1', \ldots, t_n \sim t_n' \Rightarrow \sigma(t_1, \ldots, t_n) \sim \sigma(t_1', \ldots, t_n')$ *congruencia*
- Si $t_1, t_2 \in T_\Sigma$ son dos términos cerrados tales que $t_1 \rightarrow t_2$ vía una regla de reescritura definida a partir de cualquier ecuación de *E* se tiene que $t_1 \sim t_2$.
- Si existe una *SPEC*-álgebra *A* para la que dos términos cerrados $t_1, t_2 \in T_\Sigma$ cumplen $t_1^A \neq t_2^A$, entonces $t_1 \not\sim t_2$.

6.5 DEFINICIÓN. Dada una especificación *SPEC* = (*S*, Σ, *E*). Se define constructivamente el **álgebra de términos cociente**, T_{SPEC}, como:

- Para cada género $s \in S$, su conjunto soporte se define como
$$(T_{SPEC})_s = \{ [\, t \,] \mid t \in T_{\Sigma,s} \}$$
donde la clase de equivalencia $[\, t \,]$ es el conjunto $\{ t' \in T_{\Sigma,s} \mid t' \sim t \}$.
- Para cada símbolo de operación $\sigma \in \Sigma_{w,s}$ se define $(T_{SPEC})_\sigma^{w,s}$ como:
$$(T_{SPEC})_\sigma^{w,s} : (T_{SPEC})_{s1} \times \ldots \times (T_{SPEC})_{sn} \rightarrow (T_{SPEC})_s$$
tal que $\forall t_i \in (T_{SPEC})_{si},\ (T_{SPEC})_\sigma^{w,s}([\, t_1 \,], \ldots, [\, t_n \,]) = [\, \Sigma_{w,s}(t_1, \ldots, t_n) \,]$ y donde si $w \equiv \varepsilon$, $(T_{SPEC})_\sigma^{\varepsilon,s} = [\, \sigma \,]$.

Una vez definida T_{SPEC} merecen destacarse algunas de sus propiedades características. Las demostración de las mismas puede encontrarse en

[EhMa85].

6.6 **TEOREMA.** El álgebra de términos cociente T_{SPEC} de una especificación algebraica (S, Σ, E) posee las siguientes propiedades:

- La interpretación en T_{SPEC} de todo término cerrado es una aplicación sobreyectiva que cumple $\forall t \in T_\Sigma$, $t^{T_{SPEC}} = [\, t \,]$.
- Toda ecuación cerrada ($\varnothing$, t_L, t_R) con $t_L, t_R \in T_\Sigma$ es válida en T_{SPEC} si y sólo si es válida en toda *SPEC*-álgebra A.
- T_{SPEC} es una *SPEC*-álgebra.

La sobreyectividad de la aplicación interpretación (que tiene dominio T_Σ y codominio T_{SPEC}, véase definición 2.11) establece que T_{SPEC} sólo está compuesta por elementos que pueden ser generados mediante la interpretación de términos, por lo que es accesible. La segunda propiedad significa que T_{SPEC} es **típica**, por lo que las únicas igualdades que se pueden establecer entre términos cerrados son las que se deducen de las ecuaciones, es decir, está libre de confusión por no poseer propiedades que no sean necesariamente verdad para todas las realizaciones de la especificación. La última propiedad simplemente afirma que T_{SPEC} no es sólo una Σ-álgebra sino que es en realidad una *SPEC*-álgebra.

EJEMPLO 6.2

Construyamos el álgebra de términos cociente de la especificación algebraica del ejemplo 6.1. Las clases de equivalencia inducidas en $T_{\Sigma_{BOOLEANOS}}$ por las ecuaciones *eq*1 y *eq*2 son:

[cierto] = { cierto } $\cup$ { no^n(falso) | n es impar } $\cup$ { no^n(cierto) | n es par }

[falso] = { falso } $\cup$ { no^n(cierto) | n es impar } $\cup$ { no^n(falso) | n es par }

ya que, por ejemplo y según las propiedades de la equivalencia de términos cerrados establecidas tras la definición 6.4:

$$no(no(no(falso))) \rightarrow_{eq2} no(no(cierto)) \rightarrow_{eq1} no(falso) \rightarrow_{eq2} cierto$$

De este modo el conjunto soporte de *Bool* y el comportamiento de cada

una de las funciones soporte es:

- $(T_{BOOLEANOS})_{Bool}$ = { [cierto], [falso] }
- $(T_{BOOLEANOS})^{\varepsilon,\ Bool}_{cierto}$ = [cierto]
- $(T_{BOOLEANOS})^{\varepsilon,\ Bool}_{falso}$ = [falso]
- $(T_{BOOLEANOS})^{Bool,\ Bool}_{no}$([cierto]) = [no(cierto)] ≡ [falso]
  $(T_{BOOLEANOS})^{Bool,\ Bool}_{no}$([falso]) = [no(falso)] ≡ [cierto]

Como puede observarse $T_{BOOLEANOS}$, al menos para las operaciones definidas, es isomorfo a B.

Antes de finalizar el ejemplo, estudiaremos más en detalle el doble papel que juegan las ecuaciones en la definición del comportamiento de las operaciones y en la restricción de los modelos de una especificación. Para ello añadamos una nueva operación a la anterior especificación, una operación binaria que deseamos que denote la *conjunción lógica*:

BOOLEANOS2 ≡
sorts
Bool
operations
cierto : → Bool
falso : → Bool
no : Bool → Bool
□ * □ : Bool, Bool → Bool
equations for all x : Bool
no(cierto) = falso
no(falso) = cierto

Si no se añade ninguna ecuación, el álgebra cociente queda invadida por una colección infinita de nuevos términos cerrados que, al no poderse hacer congruentes a ningún otro obligan, según el modelo inicial, a ser considerados como elementos distintos. Así tenemos infinitas clases de equivalencia que hacen que esta especificación no describa a B:

$$[cierto] = \{ cierto \} \cup \{ no^n(falso) \mid n \text{ es impar} \} \cup \{ no^n(cierto) \mid n \text{ es par} \}$$

$$[falso] = \{ falso \} \cup \{ no^n(cierto) \mid n \text{ es impar} \} \cup \{ no^n(falso) \mid n \text{ es par} \}$$

$$[cierto * cierto] = \{ (a * b) \mid a \in [cierto] \wedge b \in [cierto] \}$$

$$[cierto * falso] = \{ (a * b) \mid a \in [cierto] \wedge b \in [falso] \}$$

...

$[(\text{cierto} * \text{cierto}) * \text{cierto}] = \{ ((a*b)*c) \mid a \in [\text{cierto}] \wedge b \in [\text{cierto}] \wedge c \in [\text{cierto}] \}$

$[\text{cierto} * (\text{cierto} * \text{cierto})] = \{ (a*(b*c)) \mid a \in [\text{cierto}] \wedge b \in [\text{cierto}] \wedge c \in [\text{cierto}] \}$

...

Bastará con añadir un par de ecuaciones que describan el comportamiento de $*$, para que también las clases de equivalencia se reduzcan a dos, pudiendo corresponderse con los dos valores de B.

BOOLEANOS3 ≡
sorts
Bool
operations
cierto : → Bool
falso : → Bool
no : Bool → Bool
□ * □ : Bool, Bool → Bool
equations for all x : Bool
no(cierto) = falso
no(falso) = cierto
cierto * x = x
falso * x = falso

Tras la inclusión de las nuevas ecuaciones, las clases de equivalencia de términos cerrados se reducen a dos, que pueden definirse recursivamente como:

$$[\text{cierto}] = \{ \text{cierto} \} \cup \{ \text{no}^n(\text{falso}) \mid n \text{ es impar} \} \cup \{ \text{no}^n(\text{cierto}) \mid n \text{ es par} \} \cup \cup \{ (a*b) \mid a \in [\text{cierto}] \wedge b \in [\text{cierto}] \}$$

$$[\text{falso}] = \{ \text{falso} \} \cup \{ \text{no}^n(\text{cierto}) \mid n \text{ es impar} \} \cup \{ \text{no}^n(\text{falso}) \mid n \text{ es par} \} \cup \cup \{ (a*b) \mid a \in [\text{falso}] \wedge b \in [\text{cierto}] \} \cup \{ (a*b) \mid a \in [\text{falso}] \wedge b \in [\text{falso}]) \} \cup \cup \{ (a*b) \mid a \in [\text{cierto}] \wedge b \in [\text{falso}] \}$$

Tras las anteriores definiciones y teoremas, ya es posible definir la semántica que se asumirá que tiene una especificación algebraica.

6.7 **DEFINICIÓN.** La **semántica inicial** de una especificación algebraica *SPEC* es

la clase de todas aquellas álgebras isomorfas al álgebra de términos cociente de *SPEC*, T_{SPEC}. Dada una Σ-álgebra *A*, la especificación *SPEC* se dice que es **inicialmente correcta** respecto de *A*, si *A* es isomorfa a T_{SPEC}.

De esta definición deben extraerse algunas conclusiones muy importantes. La primera es que se establece una semántica unívoca para toda especificación algebraica de manera que pueda existir una interpretación unánime por todos los usuarios del tipo especificado (inclusive una herramienta de diseño hardware). La segunda es que dicha interpretación posee un soporte simbólico sólido, T_{SPEC}, el cual permite razonar sobre el tipo abstracto de datos de una manera puramente simbólica (en concordancia con la postura que defiendo en la presente memoria). Obviamente según la definición 6.4, el mecanismo de razonamiento privilegiado será la reescritura, mecanismo sobre cuya adecuación para la deducción automática, se discutirá en §6.3. La tercera es que establece un modo formal para chequear que cierto modelo se ha especificado correctamente: la demostración de isomorfía respecto a T_{SPEC}, tema al que se dedicará el siguiente apartado.

Como siempre que ocurre cuando se asocian sintaxis y semántica, la elección del modelo inicial debe considerarse simplemente como un convenio que deberá respetarse de ahora en adelante. Sin embargo, no es la única alternativa válida y en la literatura pueden encontrarse autores que adoptan otros enfoques diferentes [BrWi84]. Existe la llamada **semántica final** [Wand81], en donde los conjuntos soporte están formados por elementos que se consideran equivalentes a menos que su desigualdad pueda ser deducida a partir de las ecuaciones, o la **semántica laxa** [BDP+79], que considera semántica de una especificación a la clase formada por todas las *SPEC*-álgebras generadas por términos.

La razón por la que he adoptado la semántica inicial es por ser la que actualmente mejor se integra en sistemas automáticos de demostración de teoremas y en sistemas automáticos de desarrollo software, por lo que la

hace una buena candidata a incluirse también con éxito, en una herramienta de síntesis hardware como la que en esta memoria se presenta.

6.1.2 Corrección de una especificación algebraica.

Como se comentó con anterioridad, la definición 6.7 propone un método para comprobar si una especificación algebraica con semántica inicial es correcta: chequear que el modelo 'mental' del tipo abstracto de datos que se pretende formalizar mediante la especificación algebraica *SPEC* es isomorfo a T_{SPEC}. Obviamente para demostrar esa isomorfía será necesario concebir dicho modelo 'mental' en forma de álgebra y formalizarla de alguna manera rigurosa para que pueda compararse con T_{SPEC}.

El método que a continuación se expone, ha sido propuesto en [EhMa85] y requiere de algunas definiciones previas que permitan clasificar a los operadores de una signatura por sus efectos sobre los elementos de un cierto género.

6.8 **DEFINICIÓN.** Dada una signatura (S, Σ) y un género $s \in S$, se distinguen los siguientes subconjuntos de Σ:

- **Constructores de *s* (Con_s)**: conjunto formado por todas las operaciones incluidas en Σ que tengan género *s*.
- **Generadores de *s* (Gen_s)**: conjunto de constructores con la propiedad de que sólo mediante su aplicación es posible alcanzar cualquier valor del tipo a especificar o, de otro modo, conjunto de constructores tal que si se excluye cualquiera de ellos quedan valores del álgebra soporte que no son interpretación de ningún término cerrado formado sólo por operadores de dicho conjunto. Además se dice que Gen_s es un **conjunto libre de generadores**, cuando todo término incluido en T_{Gen_s} denota un valor diferente del tipo de datos asociado al género *s*. Por otro lado, se dice que Gen_s es un **conjunto no libre de generadores**, cuando dos o

más términos distintos de T_{Gen_s} denotan un mismo valor del tipo de datos.

- **Modificadores de *s*, (Mod_s)**: conjunto de todos los constructores que no pertenecen al conjunto de generadores.
- **Observadores de *s*, (Obs_s)**: conjunto formado por todas las operaciones de género distinto de *s*, en cuya aridad se incluye el género *s*.

EJEMPLO 6.3

Sea la siguiente especificación algebraica de los números enteros:

ENTEROS ≡
sorts
Ent
operations
0 : → Ent
s : Ent → Ent
p : Ent → Ent
add : Ent, Ent → Ent
equations for all n, m : Ent
s(p(n)) = n *(eq1)*
p(s(n)) = n *(eq2)*
add(n, 0) = n *(eq3)*
add(n, s(m)) = s(add(n, m)) *(eq4)*
add(n, p(m)) = p(add(n, m)) *(eq5)*

Si el soporte del símbolo 0 es el *número cero*; el del símbolo s, la función *incremento*; el del símbolo p, la función *decremento*; y el del símbolo add, la *suma entera*. Podemos clasificar las operaciones según la definición 6.8 en:

- $Cons_{Ent}$ = { 0, s, p, add }
- Gen_{Ent} = { 0, s, p }, ya que a partir del numero cero y mediante incrementos o decrementos se puede alcanzar cualquier valor entero. Además las ecuaciones *eq*1 y *eq*2 hacen que este conjunto sea un conjunto no libre de generadores.
- Mod_{Ent} = { add }
- Obs_{Ent} = ∅

Para demostrar si una especificación $SPEC \equiv (S, \Sigma, E)$ es una especificación correcta del tipo abstracto de datos modelado como el álgebra A, debe encontrase en primer lugar un conjunto de términos cerrados canónicos, C_Σ, que sean representantes de cada una de las clases de equivalencia inducidas por las ecuaciones de E. Para ello se localizan en la signatura Σ las operaciones constructoras y se identifican todos los términos que pueden alcanzarse a partir de ellas. Si no existen ecuaciones en las que intervengan constructores, entonces C_Σ estará formado por todos los términos cerrados obtenidos únicamente por composición de constructores. En caso contrario, se designan adecuadamente algunos subconjuntos de dichos términos como términos canónicos. Una vez fijado C_Σ deberá demostrarse que se cumplen las siguientes tres afirmaciones:

- La restricción $\mu_c : C_\Sigma \to A$ de $\mu : T_\Sigma \to A$, es biyectiva. *(1)*
- $\forall t \in T_\Sigma, \exists c \in C_\Sigma, c \sim t$ *(2)*
- $\forall c_1, c_2 \in C_\Sigma, c_1 \sim c_2 \Rightarrow c_1 \equiv c_2$ *(3)*

En general, no existe un algoritmo general para la construcción de C_Σ y la elección de términos canónicos se realiza en función de lo fácil que sea la demostración de (1). Por otro lado, los criterios suficientes para demostrar (2) y (3) son las propiedades de la equivalencia de términos cerrados mostradas tras la definición 6.4.

EJEMPLO 6.4

A continuación demostraremos que la especificación algebraica del ejemplo 6.3 es una especificación correcta de los números enteros, donde el soporte de los símbolos de operación es el definido también en dicho ejemplo, es decir, que *ENTEROS* es una especificación correcta de (*Z*, *0*, *+1*, *-1*, +).

La primera tarea es encontrar un conjunto de términos cerrados tal que sean capaces de representar sintácticamente a todos los números enteros. Dado que a partir del número cero y mediante incrementos y decrementos

puede alcanzarse cualquier número positivo o negativo, la intuición nos dice que si las funciones soporte de los símbolos *s* y *p* son el incremento y el decremento, un buen conjunto de términos canónicos puede ser:

$$C_\Sigma = \{ 0 \} \cup \{ s^n(0) \mid n > 1 \} \cup \{ p^n(0) \mid n > 1 \} \subset T_\Sigma$$

Para demostrar que la interpretación de los términos de este conjunto $\mu_c : C_\Sigma \to Z$ es una aplicación biyectiva, basta con comprobar como está definida (recuérdese que la interpretación de términos cerrados, queda completamente definida cuando se elige el álgebra soporte de la signatura, véase definición 2.11):

- $\mu_c(0) \equiv 0^Z = 0$
- $\forall n \in \mathbb{N}_+, \mu_c(s^n(0)) \equiv (s^n(0))^Z = n$
- $\forall n \in \mathbb{N}_+, \mu_c(p^n(0)) \equiv (p^n(0))^Z = -n$

A continuación se debe demostrar que todo término cerrado es equivalente a un término canónico. Para hacer esto demostraré por inducción estructural que todo término cerrado puede reescribirse en un término canónico (utilizando las propiedades de la equivalencia de términos cerrados), es decir:

$$\forall t \in T_\Sigma, \exists c \in C_\Sigma, t \to c$$

(1) Caso base: $t \equiv 0$, trivial ya que 0 es canónico.

(2) Hipótesis de inducción $\exists c_1, c_2 \in C_\Sigma$, $t_1 \to c_1$ y $t_2 \to c_2$

(3) Utilizando la propiedad de congruencia con cada uno de los símbolos de operación sobre la hipótesis de inducción, bastará con demostrar:

(3.1) $s(t_1) \to s(c_1)$, dado que c_1 es canónico sólo hay tres alternativas:

(3.1.1) si $c_1 \equiv 0$, entonces $s(t_1) \to s(0) \in C_\Sigma$

(3.1.2) si $c_1 \equiv s^n(0)$ para cierto n, entonces $s(t_1) \to s(s^n(0)) \equiv s^{n+1}(0) \in C_\Sigma$

(3.1.3) si $c_1 \equiv p^n(0)$ para cierto n, entonces $s(t_1) \to s(p^n(0)) \to_{eq1} p^{n-1}(0) \in C_\Sigma$

(3.2) $p(t_1) \to p(c_1)$, según el mismo argumento anterior:

(3.2.1) si $c_1 \equiv 0$ entonces $p(t_1) \to p(0) \in C_\Sigma$

(3.2.2) si $c_1 \equiv s^n(0)$ para cierto n, entonces $p(t_1) \to p(s^n(0)) \to_{eq2} s^{n-1}(0) \in C_\Sigma$

(3.2.3) si $c_1 \equiv p^n(0)$ para cierto p, entonces $p(t_1) \to p(p^n(0)) \equiv p^{n+1}(0) \in C_\Sigma$

(3.3) $add(t_1,t_2) \to add(c_1,c_2)$, volvería a repetirse el mismo argumento para

cada una de las distintas combinaciones posibles de valores canónicos de c_1 y c_2. Lo demuestro sólo para una ya que el resto se harían igual:

(3.3.*i*) Si $c_1 \equiv s^n(0)$ para cierto n, entonces se tienen 3 alternativas para c_2:

(3.3.*i*.1) si $c_2 \equiv 0$ entonces $add(t_1,t_2) \rightarrow add(s^n(0),0) \rightarrow_{eq3} s^n(0) \in C_\Sigma$

(3.3.*i*.2) si $c_2 \equiv s^m(0)$ para cierto m, entonces:

$add(t_1,t_2) \rightarrow add(s^n(0),s^m(0)) \rightarrow_{eq4} s(add(s^n(0),s^{m-1}(0))) \rightarrow_{eq4} \cdots \rightarrow_{eq4}$
$\rightarrow_{eq4} s^m(add(s^n(0),0)) \rightarrow_{eq3} s^m(s^n(0)) \equiv s^{m+n}(0) \in C_\Sigma$

(3.3.*i*.2) si $c_2 \equiv p^m(0)$ para cierto m, entonces:

$add(t_1,t_2) \rightarrow add(s^n(0),p^m(0)) \rightarrow_{eq5} p(add(s^n(0),p^{m-1}(0))) \rightarrow_{eq4} \cdots \rightarrow_{eq4}$
$\rightarrow_{eq4} p^m(add(s^n(0),0)) \rightarrow_{eq3} p^m(s^n(0)) \rightarrow_{eq2} p^{m-1}(s^{n-1}(0)) \rightarrow_{eq2} \cdots$

que terminará convergiendo o bien en 0 si $m=n$, en $s^{n-m}(0)$ si $n>m$ o en $p^{m-n}(0)$ si $n<m$, siendo en cualquier caso todos ellos términos canónicos.

Para finalizar es necesario demostrar que dos términos canónicos son equivalentes si y solo si son el mismo. Sean c_1, $c_2 \in C_\Sigma$, por la definición de equivalencia dos términos son equivalente si sus interpretaciones son iguales, luego $c_1 \sim c_2$ implica que $(c_1)^Z = (c_2)^Z$ que es equivalente a $\mu_c(c_1) = \mu_c(c_2)$ y por ser μ_c biyectiva se concluye que $c_1 \equiv c_2$.

□

6.1.3 Método sistemático de especificación algebraica.

La definición 6.7 proponía un método de verificación de especificaciones algebraicas que se ha implementado en el anterior apartado. Sin embargo, este método sólo es útil cuando se tiene una imagen clara del tipo de datos a especificar. Cuando el tipo de datos evoluciona a la par que se especifica, la única manera de asegurar la corrección, es decir, que se especifica lo que se desea y no otra cosa, es utilizar un método sistemático de especificación.

El método sistemático de especificación algebraica que aquí se estudia es un extracto del método propuesto en [Peña93] y ha sido utilizado con éxito para la especificación de todos los tipos de datos de interés en un sistema de síntesis de alto nivel (véase §6.1.4).

La idea fundamental del método es que debe especificarse paso a paso. Así, se comienza con la elección de un conjunto suficiente de operaciones generadoras que permitan construir los conjuntos soporte de cada uno de los géneros según el modelo inicial. A continuación, si dicho conjunto no está libremente generado, se añade una colección de ecuaciones cuyos términos estén formados solamente por generadores. Seguidamente, se va enriqueciendo la especificación con nuevas operaciones cuyo comportamiento se define en base a cómo afectan a las operaciones generadoras. Esta extensión deberá ser conservativa, es decir, no deberá añadir nuevos elementos a la semántica inicial y consistente, no deberá identificar ningún elemento antiguo.

Concretando, las normas generales para la elección de las ecuaciones con las que definir el comportamiento de cualquier operador son:

- Si Gen_s es un conjunto libre de generadores, las únicas ecuaciones a escribir son las relacionados con los modificadores y los observadores. Si no los hubiera la especificación algebraica no poseería ecuaciones.
- Si Gen_s es un conjunto no libre de generadores, serán necesarias algunas ecuaciones para hacer congruentes algunos términos de $T_{Gen,}$. Dichas ecuaciones sólo estarán formadas por generadores y su objetivo será hacer congruente todo término no canónico de $T_{Gen,}$ con el representante canónico de su clase.
- Para cada modificador, se escribirán tantas ecuaciones como sean necesarias para garantizar que todo término de $T_{Cons,}$ sea congruente a algún término de $T_{Gen,}$. La estrategia más habitual es escribir ecuaciones en las que el término izquierdo esté formado por un

modificador en su raíz que tenga por argumentos de género *s*, distintos patrones de operaciones generadoras y en las que el término derecho contenga también a dicho modificador pero no ubicado en la raíz. Estas ecuaciones proporcionarán, si se orientan de izquierda a derecha, un método eficiente de cómputo por reescritura.

- Para cada observador se escribirán tantas ecuaciones como sean necesarias para garantizar que toda observación de términos de género s, sea congruente con algún término de $T_{Cons_{s'}}$ (donde *s'* es el género con el que se mide la observación). El problema de la construcción de este tipo de ecuaciones que involucran operaciones de distintos géneros es que si no se ponen suficientes, el tipo descrito por el género *s'* podría verse invadido por infinitos nuevos valores procedentes de las observaciones del género *s* que no son congruentes con los valores previamente construidos a partir de $Cons_{s'}$. Por otro lado, si se ponen demasiadas, podrían hacerse congruentes valores de género *s* que previamente no lo eran. En cualquier caso las recomendaciones dadas para los modificadores son también aquí válidas.

Para finalizar debe decirse que en ocasiones, una operación modificadora (u observadora) puede especificarse exclusivamente en términos de otra u otras operaciones modificadoras (u observadoras), sin que aparezcan en las ecuaciones operaciones generadoras del tipo de interés. Asimismo, no todas las operaciones han de estar especificadas respecto al mismo conjunto de generadores, ya que lo realmente importante es que, en cualquier caso, todo término cerrado pueda ser reducido a un término canónico, aunque el proceso requiera de la intervención de otros modificadores (u observadores) que serán los que finalmente sean reducidos.

Además de estas normas, la literatura propone algunas restricciones en la manera de redactar las ecuaciones para que en el caso de que se orienten de izquierda a derecha (el modo más simple de orientar una ecuación) se

obtenga un sistema eficiente de cálculo por reescritura de términos cerrados, para ello véase [HoDo82].

EJEMPLO 6.5

Vamos a poner en práctica el método propuesto para la especificación simultánea de los números naturales y de los números enteros, utilizando esta vez una signatura más intuitiva que la usada en el ejemplo 6.3 (que estaba basada en los conceptos de sucesor y predecesor). La nueva signatura tratará de reproducir la notación posicional con signo explícito en base 2. Se ha elegido la base 2 para economizar el número de símbolos necesario para elaborar un ejemplo representativo.

El esquema general que seguiremos comenzará especificando los naturales en notación posicional y una vez especificados, continuará añadiendo el signo explícito a la notación obtenida para especificar los enteros. Los nombres de los géneros utilizados serán *unsigned* para los naturales y *signed* para los enteros.

La primera tarea es elegir un conjunto de generadores suficiente para alcanzar todos los datos del tipo que se desea especificar, en nuestro caso los naturales. El conjunto que elegiré reproducirá perfectamente el proceso que cualquier humano sigue para escribir un numeral binario: comienza escribiendo un 1 ó un 0 y a continuación va colocando sucesivamente nuevos dígitos a la derecha de éste. De este modo elegiré un conjunto de generadores formado por las constantes 1, 0 y los operadores monarios postfijos 1 y 0. Esto permitirá que el numeral binario '101' se represente mediante un término a primera vista idéntico: 101, aunque si ponemos paréntesis podrá distinguirse mejor el lugar que ocupan las operaciones ((1)0)1.

sorts
unsigned
operations
1 : → unsigned

0 : → unsigned
□0 : unsigned → unsigned
□1 : unsigned → unsigned

Sin embargo, este conjunto de generadores es demasiado potente ya que permite construir más términos cerrados diferentes que números naturales existen. Así, son términos válidos el 00 y el 000 y el 0000 ..., términos que la intuición nos dice que deben ser maneras equivalentes de representar la misma cantidad. Por tanto, será necesario añadir alguna ecuación para hacer que los términos no puedan ser libremente generados, ecuaciones que deberán reflejar un concepto elemental sobre equivalencia de numerales: la inutilidad de los ceros a la izquierda.

equations
00 = 0 *(eq1)*
01 = 1 *(eq2)*

Obsérvese que en estas ecuaciones sólo intervienen constructores y cómo no pueden reemplazarse por una única ecuación $0x = x$ (ya que esto requería permitir variables de segundo orden y en ese caso la ecuación sería inconsistente por denotar el lado izquierdo un valor y el derecho una función). Compruébese también que el conjunto de términos canónicos inducido por estas dos ecuaciones es:

$$C_\Sigma = \{ 0 \} \cup \{ 1x \mid x \text{ son sucesivas aplicaciones de 0 ó 1} \}$$

y que dicho conjunto es equivalente al de los números naturales, pero no equivalente al de los vectores de bits (que sería equivalente al generado libremente a partir de la especificación sin ecuaciones).

Definamos a continuación una operación modificadora: la suma. Para ello primero aumentamos la signatura con el nuevo símbolo de operación:

operations
...
□ + □ : unsigned, unsigned → unsigned

Y a continuación definimos un conjunto de ecuaciones que simulen los cálculos que realizamos cuando operamos en base 2. Así, primero debemos

especificar cómo sumar dígitos (es decir, la tabla de sumar):

equations
...
0 + 0 = 0 *(eq3)*
0 + 1 = 1 *(eq4)*
1 + 0 = 1 *(eq5)*
1 + 1 = 10 *(eq6)*

Para después especificar cómo sumar numerales compuestos. Sumando dígito a dígito de derecha a izquierda y 'llevándose' los acarreos (tal y como se hace a mano).

equations for all a, b : unsigned
...
a0 + b0 = (a + b)0 *(eq7)*
a0 + b1 = (a + b)1 *(eq8)*
a1 + b0 = (a + b)1 *(eq9)*
a1 + b1 = ((a + b) + 1)0 *(eq10)*

Obsérvese, si se me permite el símil, que al igual que cuando nosotros aprendimos a sumar, no aprendimos a sumar cantidades (eso ya lo sabíamos de antes) sino que aprendimos a manipular numerales (representantes de cantidades) de manera que el numeral obtenido representara a la cantidad suma, mediante esta especificación estamos enseñando a 'sumar' a la herramienta de síntesis. Así por ejemplo (omitiendo algunos paréntesis):

$$(11+10) \rightarrow_{eq9} (1+1)1 \rightarrow_{eq6} 101 \qquad 3+2 = 5$$

o, en el caso de que los términos tengan distinta anchura, se rellena con ceros por la izquierda el numeral más corto:

$$(11+1) \rightarrow_{eq2} (11+01) \rightarrow_{eq10} ((1+0)+1)0 \rightarrow_{eq5} (1+1)0 \rightarrow_{eq6} 100 \qquad 3+1 = 4$$

A continuación especificaremos los enteros. Para ello, en lugar de volver a repetir un proceso parecido al realizado con los naturales, utilizaremos el trabajo hecho y aplicaremos la idea intuitiva de que para obtener un entero basta con añadir un signo a un natural. Por ello se añade a la especificación un nuevo género y dos nuevas operaciones que denoten los signos:

sorts
..., signed

operations

...
+□ : unsigned → signed
-□ : unsigned → signed

Nuevamente el conjunto de términos cerrados de género signed es mayor que el conjunto de números enteros (de hecho tiene un elemento más) por ello es necesaria una ecuación que haga que el término extra se haga congruente con otro. Obviamente el término extra es uno de los dos que resultan de ponerle signo al 0. Por ello dado que no tiene sentido que el 0 tenga signo, debemos hacer ambos términos equivalentes:

equations ...
-0 = +0

El proceso podría continuar definiendo la suma de enteros en base a la suma de naturales, mediante ecuaciones que reflejaran las leyes del signo (en general serían necesarias operaciones extra, llamadas **ocultas**, que permitieran comparar los valores absolutos de los términos y tomar una decisión sobre el resultado). Y se irían añadiendo más y más operaciones, definiendo su comportamiento siempre en base de constructores.

Para finalizar, un comentario. La decisión de cómo especificar los tipos de datos puede depender de muchos factores. Por ejemplo, de lo intuitivo que sea el soporte sintáctico: así el soporte aquí presentado es más intuitivo que el del ejemplo 6.3 para especificar los enteros; o de el número de ecuaciones necesarias para especificar un comportamiento: en el aquí presentado se necesitan 10 ecuaciones para definir la suma, en el ejemplo 6.3 sólo 3; o en lo que faciliten las pruebas de nuevas propiedades (tal y como se discutirá en §6.3).

6.1.4 Especificación algebraica de objetos de diseño.

Una vez descrito el método de especificación de tipos abstractos de datos, es necesario comprobar si es útil para el propósito que se buscaba: la especificación de los objetos de diseño implicados en un proceso de síntesis de alto nivel.

En un sistema de síntesis de alto nivel existen tres categorías fundamentales de objetos que pueden catalogarse como tipos abstractos de datos. La primera son los tipos de nivel algorítmico con los que se definen los algoritmos abstractos que suelen tomarse como punto de partida del proceso de diseño. La segunda son los tipos de nivel RT con los que se construyen algoritmos *bit-true* que tanto pueden ser punto de partida como de destino de un proceso de síntesis. La tercera son las bibliotecas de componentes hardware que, constituyendo una manera alternativa de describir los algoritmos de nivel RT, básicamente se suelen utilizar para construir especificaciones que describan con claridad las estructuras de módulos que se obtienen como resultado de la síntesis.

Este apartado trata de recoger mis experiencias en la especificación algebraica de los tipos de datos anteriormente referidos, y tal como se dijo al introducir el capítulo, el objetivo de esta sección no es realizar un listado detallado de dichas especificaciones, sino más bien mostrar qué se ha podido especificar y ofrecer algunos apuntes sobre el método (que en su mayor parte ha sido ya mostrado en §6.1.3). Este estudio tratará de dejar constancia de la versatilidad y utilidad de la propuesta y será completado en §6.3.2 con un estudio de las posibilidades que ofrece para la automatización de pruebas en el modelo inicial que especifican.

Especificación de tipos de nivel RT.

Los únicos tipos admisibles en una descripción a nivel RT deben ser el tipo **bit** y el tipo **vector de bits** y deben especificarse con capacidad de describir únicamente lo que sucede en un circuito físico a este nivel de abstracción, es decir, sólo deben ser capaces de describir transformaciones de cadenas de ceros y unos sin más información que el valor de los bits que forman la cadena.

Este detalle es importante ya que cabría pensar que a este nivel de abstracción, tal y como ocurre en algunas herramientas de nivel RT, sería conveniente que los vectores de bits pudieran contener alguna información sobre la representación de datos usada (con signo, sin signo, etc.) para poder chequear que se manipulan correctamente. Pero eso es una falacia: los módulos RT no tienen conocimiento alguno de la codificación que adoptan los datos que transforman, por lo que dicha información debe haber desaparecido de cualquier descripción razonable de un circuito a ese nivel. Habrá sido una fase de diseño anterior la encargada de proyectar con corrección las funcionalidades algorítmicas, que trabajan con múltiples tipos que se representan de diversas formas, sobre una colección de elementos que sólo manipulan un tipo homogéneo de datos: cadenas de bits. El método para formalizar las diferentes maneras de proyectar dichas funcionalidades se estudiará en §6.2.

De este modo, elegí como generadores del tipo bit los dos valores lógicos y los vectores de bits se generaron concatenando bits (véase ejemplo 6.8). Seguidamente se definió un conjunto suficiente de funciones lógicas que se utilizaron posteriormente para definir las funcionalidades sobre vectores. Además, para soportar observaciones tales como la anchura de un vector o extraer campos de un vector de bits, fue necesario especificar previamente el tipo natural.

Debe destacarse que el proceso algebraico que seguí fue muy parecido al que se sigue para el diseño manual de hardware real: se comienza definiendo algunas celdas básicas de nivel lógico que posteriormente se agrupan para formar módulos más complejos que trabajan con vectores. No obstante, no debe extrapolarse el símil más allá de lo necesario. Cuando se diseña hardware se trata de componer funcionalidades simples para construir una funcionalidad más compleja, pero no debe olvidarse que también se trata de obtener una estructura real con ciertas características físicas (área, retardo, consumo). Sin embargo, cuando se especifica algebraicamente un tipo a partir de otro, sólo se hace para reutilizar algo ya hecho con anterioridad, por lo que no se debe extraer ninguna conclusión de tipo físico de la estructura que adoptan los términos en las ecuaciones.

Una especificación algebraica especifica aspectos puramente funcionales de un comportamiento y, en general, se hace de manera que se cumplan objetivos muy distantes del hardware, objetivos tales como la simplificación del posterior proceso de demostración de nuevas propiedades. Así que no es de extrañar que (a menos que se dé una interpretación añadida a la configuración que adoptan los términos [Shee90][JoSh90]) pueda describirse la funcionalidad de una suma con anticipación de acarreo mediante una composición de términos que recuerde al algoritmo de suma con propagación.

Para finalizar, a continuación se repasan los principales tipos de funcionalidades de nivel RT que pudieron especificarse algebraicamente sin dificultad (sin contar las operaciones de transformación de bits):

- **Operaciones de adaptación de anchuras**: truncamiento, extensión, concatenación y extracción de bits y rangos.
- **Operaciones de reordenación de bits**: desplazamiento lógico, desplazamiento aritmético, rotación lógica y rotación aritmética.
- **Operaciones lógicas bit a bit**, que generan un vector de bits calculado a partir de la operación lógica de cada uno de los bits colocados en la

misma posición dentro de los vectores argumento.

- **Operaciones lógicas de palabra completa**, que generan un bit a partir de la operación lógica de todos los bits de un vector argumento.
- **Operaciones lógicas con operandos de distinto tipo**, que generan un vector de bits a partir de la operación lógica de un único bit con cada uno de los bits de un vector.
- **Operaciones de selección de operandos**, es decir, multiplexores.
- **Aritmética sin signo**: suma, resta y multiplicación.
- **Aritmética con signo**†: suma, resta y multiplicación en complemento a dos, complemento a uno y magnitud y signo.

Especificación de tipos de nivel algorítmico.

Como tipos de nivel algorítmico, se ha comprobado que pueden especificarse algebraicamente con éxito, los tipos **booleano**, **natural**, **entero**, **racional**, **alfanumérico** y, como aproximación a los reales, la **representación posicional binaria con punto fijo y signo explícito**.

El modo general de especificar los tipos booleano, natural y entero ya se ha mostrado en los ejemplos 6.1 y 6.3. En cuanto al tipo racional, éste ha sido generado a partir de dos enteros que denotan el numerador y el denominador de la fracción. Por su parte, el tipo alfanumérico fue generado mediante un conjunto de operaciones constantes cada una de las cuales denotaba una letra distinta. Sin embargo, la aproximación a los reales merece

† Es importante destacar que la aritmética con signo define manipulaciones de bits distintas de las realizadas por la aritmética sin signo, por ello se describen aparte. Esto, sin embargo, no quiere decir que a este nivel de abstracción se tenga conocimiento alguno de cómo están codificados los datos, ya que la manipulación se realizará en cualquier caso independientemente de cómo estén codificados en realidad los operandos. La selección de dicha codificación, de cómo proyectarla correctamente sobre funcionalidades lógicas y de asegurar que los operandos se manipulan en coherencia con su representación, es una tarea de diseño que será formalizada más adelante mediante una implementación algebraica de los tipos de nivel algorítmico.

algunos comentarios.

La imposibilidad de especificar algebraicamente los números reales surge del incumplimiento que hacen de la condición de generabilidad exigida a cualquier tipo abstracto de datos especificable. Esta exigencia previene de que se trate de especificar un álgebra que contenga objetos que sean inalcanzables mediante un proceso finito de cálculo y, por consiguiente, que se especifique algo irrealizable (no existe ningún sistema físico, ni analógico ni digital, que pueda realizar cálculos con una precisión infinita).

Este mismo problema surge en campos ajenos a la informática y se soluciona distinguiendo en el espectro no numerable de los reales un conjunto numerable de intervalos, cada uno de los cuales, se representa por una única cantidad. Tras esto, se define un sistema de numeración que establece una correspondencia biunívoca entre dichas cantidades representantes y un conjunto de numerales (símbolos sintácticos). El sistema de numeración más común entre los humanos, aunque no el único, es el de notación posicional en base decimal con punto fijo y signo explícito.

Esta misma solución es la que se ha tratado de especificar algebraicamente. Sin embargo aún queda por explicar el motivo por el cual se ha elegido la base 2 en lugar de la base 10. La razón es que, excepto en casos muy particulares de codificaciones BCD, los módulos aritméticos se suelen diseñar para operar en bases potencias de 2. Por lo que si un cálculo se especifica asumiendo que los operandos están expresados en base 10 y que tienen parte fraccionaria, será imposible diseñar un circuito digital que se comporte exactamente igual que dicha especificación, ya que existirán cantidades que, expresadas con un número finito de dígitos en base 10, requieran un número infinito de dígitos para expresarse en base 2 (por ejemplo, $0.3_{10} = 0.0101010..._2$).

Según esto, para diseñar un circuito correcto a partir de una especificación en base 10 o bien se codifica siempre en BCD (dejando fuera la mayor parte

de las implementaciones válidas) o bien se incluyen en el formalismo matemático propuesto, todos los mecanismos de análisis de precisión. Dado que esta última alternativa queda por el momento fuera del alcance de la investigación que en esta memoria se reporta, decidí adoptar la base 2†. No obstante, el uso de la base 10 podrá aceptarse siempre y cuando las cantidades a representar sean enteras ya que, en ese caso, el problema de diferencia de bases desaparece.

Para especificar algebraicamente la representación posicional binaria con punto fijo y signo explícito, se han especificado como pasos intermedios un total de cuatro tipos abstractos de datos. El **tipo binario** (isomorfo a los naturales), el **tipo binario con signo** (isomorfo a los enteros), el **tipo binario** con punto fijo y el **tipo binario con punto fijo y signo**. El método para especificar los dos primeros se ha mostrado en el ejemplo 6.5. En cuanto el tipo binario con punto fijo, se ha generado como un par formado por un objeto de género binario y un objeto de género natural. El objeto binario indica los dígitos utilizados, mientras que el objeto natural indica la posición del punto en relación a los dígitos contando desde la derecha de éstos. El tipo binario con punto fijo y signo se construye añadiendo el signo al anterior tipo.

EJEMPLO 6.6

A continuación se muestra un extracto de la especificación del tipo binario con punto fijo denotado por el género *unsignedFxp*.

sorts
natural, unsigned, unsignedFxp
operations

† Puede que esta decisión no agrade a un diseñador clásico acostumbrado a trabajar con el tipo real facilitado por los HDLs, pero antes o después el problema de la numerización, acotación y codificación de dominios debe ser resuelto en todo proceso de síntesis que tenga cierto nivel de abstracción. Lo único que hacen los HDLs es ocultar un problema que la herramienta de diseño deberá afrontar por sí sola, modo de resolver que al no estar contemplado en la especificación, hará que el circuito final acumule errores de representación no previstos por el diseñador y, por consiguiente, con implicaciones imprevistas en los resultados.

```
...
[ □, □ ]:  unsigned, natural → unsignedFxp
□ + □  :  unsignedFxp,unsignedFxp → unsignedFxp
equations for all n : natural; a, b : unsigned;
...
[ a0, succ(n) ] = [ a, n ]
[ 0, n ] = [ 0, 0 ]
[ a, n ] + [ b, n ] = [ a+b, n ]
```

A modo de resumen, apuntaré las operaciones de nivel algorítmico especificadas (cada tipo de datos tiene un subconjunto de ellas coherente con su naturaleza):

- **Operaciones booleanas**: negación, conjunción, disyunción e implicación.
- **Operaciones aritméticas**: negación, incremento, decremento, suma, resta, multiplicación, división entera, resto entero, potenciación, raíz cuadrada, factorial, máximo, mínimo, valor absoluto y signo.
- **Operaciones de comparación**.
- **Operaciones de selección**: condicional y múltiple.
- **Operaciones de clasificación**: es negativo, es par, es impar, es múltiplo, es divisible, es entero y es fraccionario.
- **Operaciones de conversión de tipo**.
- **Operaciones de redondeo**: techo, suelo, parte entera, parte decimal y redondeo.
- **Operaciones de manipulación alfanumérica**: siguiente en orden alfabético, anterior en orden alfabético, poner en mayúscula y poner en minúscula.

Aunque podría hacerse, el método de especificación algebraica propuesto no es el adecuado para la especificación de tipos no atómicos (estructuras, arrays, listas, colas), para ello es conveniente una ampliación de esta noción para que soporte la parametrización [TaWW82]. Sin embargo, el problema de

estos tipos, que podrían considerarse de nivel sistema, no está en el método de especificación, sino en que sus operaciones no se adaptan al paradigma de implementación combinacional que se asume para todo operador (véase §2.1). Pero esto no es de extrañar ya que al ser operaciones relativamente complicadas, se salen incluso del ámbito de la síntesis de alto nivel.

Especificación de recursos funcionales.

En anteriores capítulos (tal y como se hace en la mayor parte de los sistemas de síntesis) se ha aceptado tácitamente que un módulo hardware combinacional puede modelarse mediante una función. Esto conduce a que no fuera necesaria hacer una distinción explícita entre la especificación algebraica de módulos hardware y la de funciones de nivel RT. Sin embargo este modelado simplista es incompleto: un módulo hardware suele ser capaz de realizar simultáneamente más de una función†, además de estar caracterizado por un conjunto de atributos físicos no conductuales (área, retardo, consumo...).

Por otra parte, desde el punto de vista descriptivo, una estructura de nivel RT no está formada por una composición de funciones sino por una interconexión de instancias concretas que pertenecen a cierta clase. Luego si se desea que, tras un proceso de diseño por derivación, pueda describirse mediante una especificación ecuacional una estructura hardware real, será necesario dar un soporte sintáctico a la instanciación.

Ofrecer una solución uniforme al problema de la especificación y uso de bibliotecas de componentes e incorporarlo al sistema de síntesis formal no es trivial, por ello previamente realizaré matizaciones para centrar la solución de

† Por ejemplo, un sumador calcula la suma a la vez que calcula el acarreo, por lo que un proceso de diseño podría proyectar funciones de alto nivel sobre cualquiera de ellas: usando la primera efectivamente para sumar, pero la segunda para comparar.

los tres problemas anteriormente descritos.

Una solución habitual para poder agrupar bajo un mismo nombre una colección de funciones es el uso de tuplas. De este modo es posible definir funciones que tengan como género tuplas de géneros. Sin embargo, en mi opinión, la incorporación de tuplas a un formalismo (y en especial a un formalismo orientado al hardware) aparte de añadir una colección amplia de nuevos operadores polimórficos (concatenación, indexación, etc), añade un conjunto de complicaciones técnicas ajenas al diseñador, que tienen una solución más complicada que el problema que pretenden resolver. Piénsese, por ejemplo, en la diferencia ficticia impuesta entre una función de tres argumentos atómicos de otra con el mismo comportamiento que tenga un único argumento que sea una tripleta de géneros atómicos. Además, desde el punto de vista del diseñador hardware, el uso de un único símbolo con varios argumentos de salida complica innecesariamente la expresión de un uso parcial de las funcionalidades y la expresión de un uso simultáneo de todas ellas, pero cada una para diferentes propósitos.

Por otro lado, si se asimila un módulo hardware con una función, la atribución de características físicas implica la atribución de funciones, por lo que dichas atribuciones deberán modelarse mediante funciones de orden superior. Es posible concebir especificaciones algebraicas de orden superior, sin embargo, resultaría chocante que solamente para solucionar un problema tan particular fuera necesario modificar todo el paradigma de primer orden en el que esta basado el sistema de síntesis propuesto hasta el momento. Por otra parte, la atribución de funciones no capta el verdadero sentido que tienen para un diseñador los atributos físicos: para un diseñador dos módulos con igual comportamiento pero, por ejemplo, con distinto retardo son módulos diferentes, que incluso pueden llegar a considerarse funciones diferentes por no ser intercambiables.

Finalmente, aunque parezca una mera cuestión estética, es necesario

destacar que en una estructura de nivel RT existe una diferencia clara entre lo que es una señal, que se caracteriza completamente por su anchura y se implementa mediante una interconexión, y lo que es una instancia, que se caracteriza por su interconexionado a otras instancias y por su pertenencia a una cierta clase. Por ello, si un formalismo sólo permite la declaración de señales y el uso de funciones, será incapaz de distinguir entre los nombres de las referencias, los nombres de las instancias y los nombres de las señales que las interconectan, siendo incapaz de reflejar el resultado de cualquier proyección tecnológica, incluyendo la fase de selección de implementaciones de un proceso de síntesis de alto nivel.

Aunque tras esta discusión, pudiera parecer que la propuesta algebraica es insuficiente para dar una solución uniforme al problema de la especificación y uso de bibliotecas de componentes, a continuación demostraré que con un modelado adecuado esto no es cierto.

La idea (véase fig. 6.1) es modelar un módulo hardware no como una función, sino como tipo abstracto de datos completo. En donde el conjunto soporte está formado por cada uno de los posibles grupos de valores que

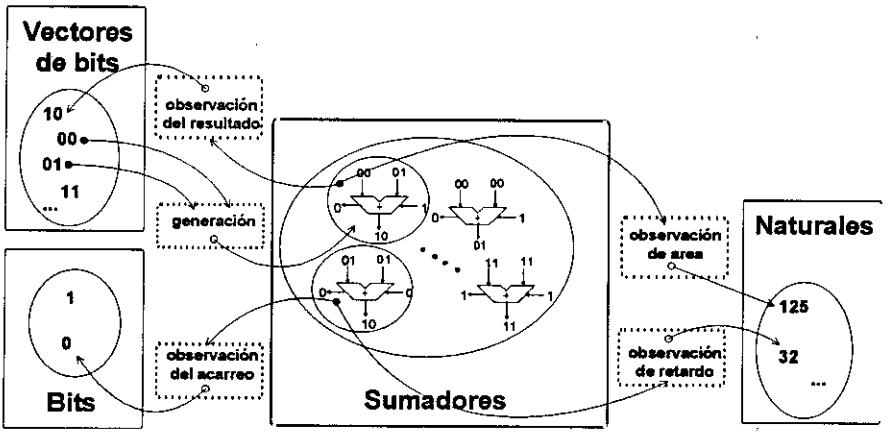


Fig. 6.1: Modelo de un sumador.

pueden encontrarse en los puertos del módulo a especificar, junto con las características físicas que lo describen, de manera que cada uno de dichos aspectos parciales (ya sea funcional como no funcional) pueda ser observado a través de un operador diferente y sea activado a través de un único generador.

Para especificar algebraicamente ese modelo [MHF96a][MHF96b][MeHF97] propongo un método en tres fases (suponiendo especificados los tipos bit y vector de bits):

- **Especificación del interfaz y de las características observables**: mediante una signatura se asocia un género por módulo a especificar, se declara un constructor del tipo cuya aridad describa el género de los puertos de entrada del módulo y se declaran tantos observadores, como puertos de salida y atributos no funcionales tenga.
- **Asignación de funcionalidades**: mediante ecuaciones se define el comportamiento del módulo, igualando cada término formado por un constructor y un observador funcional distinto, a una función de nivel RT concreta.
- **Asignación de atributos físicos**: mediante ecuaciones se definen las características físicas del módulo, igualando cada término formado por un constructor y un observador no funcional diferente, a cierto término general.

EJEMPLO 6.7

Supongamos que se desea especificar una biblioteca formada por un único sumador. Primero se asocia un identificador de género (*sumador*) al único módulo a especificar. A continuación se declara el constructor del tipo (*sumadorOp*) que permitirá describir el hecho de colocar una instancia de *sumador* en el circuito y de conectar sus puertos de entrada a través de ciertas señales. Seguidamente, cada una de las características observables del sumador se declaran como operadores, en este caso: el área (*verArea*),

el retardo (*verRetador*) y los valores en los puertos de salida (*verSuma* y *verAcarreo*). Estos últimos permitirán describir el hecho de conectar por separado cada uno de los puertos de salida de una instancia.

```
sorts
sumador
operations
sumadorOp  :  bitVector, bitVector, bit → sumador
verSuma    :  sumador → bitVector
verAcarreo :  sumador → bit
verRetardo :  sumador → natural
verArea    :  sumador → natural
```

Mediante ecuaciones establezco el comportamiento de los anteriores operadores. En el caso de lectura de puertos de salida se establecen correspondencias con manipulaciones de bits (tales como *add* o *carry*) definidas en la especificación algebraica de los tipos de nivel RT. En el caso de características físicas se establecen maneras de calcular un número que pueda ser utilizado por los algoritmos de optimización de herramienta de síntesis. Obsérvese cómo puede parametrizarse ese cálculo en función de características tales como la anchura de los datos (e incluso en función de los valores concretos que adoptaran las entradas).

```
equations for all c : bit; l, r : bitVector;
verSuma( sumadorOp( l, r, c ) ) = add( l, r, c )
verAcarreo( sumadorOp( l, r, c ) ) = carry( l, r, c )
verArea( sumadorOp( l, r, c ) ) = width( l )*8
verRetardo( sumadorOp( l, r, c ) ) = width( l )*3
```

Si el método propuesto se amplía en la misma línea, es posible describir bibliotecas que tengan en cuenta aspectos tecnológicos de los módulos hardware:

```
sorts
sumador, tecnologia
operations
es2_10m    :  → tecnologia
xilinx_7000 :  → tecnologia
sumadorOp  :  bitVector, bitVector, bit → sumador
verArea    :  tecnologia, sumador → natural
```

```
...
equations for all c : bit; l, r : bitVector;
verArea( es2_10m, sumadorOp( l, r, c ) ) = width( l )*8
verArea( xilinx_7000, sumadorOp( l, r, c ) ) = width( l )*40
```

o que describan distintas estructuras de un mismo comportamiento:

```
sorts
carryRipple, carryLookahead, piramid
operations
carryRippleOp      : bitVector, bitVector, bit → carryRipple
carryLookaheadOp   : bitVector, bitVector, bit → carryLookahead
piramid            : bitVector, bitVector, bit → piramid
verSuma            : carryRipple → bitVector
verSuma            : carryLookahead → bitVector
verSuma            : piramid → bitVector
...
equations for all c : bit; l, r : bitVector;
verSuma( carryRippleOp( l, r, c ) ) = add( l, r, c )          ( eq1 )
verSuma( carryLookaheadOp( l, r, c ) ) = add( l, r, c )       ( eq2 )
verSuma( piramidOp( l, r, c ) ) = add( l, r, c )
...
```

o que describan módulos multifuncionales:

```
sorts
sumRestador
operations
sumRestadorOp  : bit, bitVector, bitVector, bit → sumRestador
verResultado   : sumRestador → bitVector
verExceso      : sumRestador → bit
equations for all c : bit; l, r : bitVector
verResultado( sumRestadorOp( 0, l, r, c ) ) = add( l, r, c )
verResultado( sumRestadorOp( 1, l, r, c ) ) = sub( l, r, c )
verExceso( sumRestadorOp( 0, l, r, c ) ) = carry( l, r, c )
verExceso( sumRestadorOp( 1, l, r, c ) ) = borrow( l, r, c )
```

Por otra parte, este método de especificación de bibliotecas permite soportar la instanciación dentro del formalismo de especificación ecuacional, de manera que pueda quedar descrito el resultado de una cierta selección de implementaciones. Así, sea el siguiente fragmento de una especificación ecuacional:

```
signals
```

```
t1, t2, t3, t4 : bitVector
...
body
t1 = add( t2, t3, 0 )
t4 = add( t1, t3, 0 )
...
```

Este fragmento puede pertenecer a un comportamiento descrito a nivel RT. Si una posterior fase de diseño decidiera proyectar la primera de las sumas sobre un sumador con propagación de acarreo y la segunda sobre un sumador con anticipación de acarreo (ambos módulos pertenecientes, por ejemplo, a una de las anteriores bibliotecas), bastaría con aplicar algunas ecuaciones:

```
                          signals
                          t1, t2, t3, t4 : bitVector
                          a1 : carryRipple
                          a2 : carryLookahead
                          ...
                          body
                          a1 = carryRippleOp( t2, t3, 0 )
AplicacionDI( t1, ε, eq1 ) a2 = carryLookaheadOp( t1, t3, 0 )
AplicacionDI( t4, ε, eq2 ) t1 = verSuma( a1 )
   Expansion( t1, 1, a1 )  t4 = verSuma( a2 )
   Expansion( t4, 1, a2 )  ...
```

Como puede observarse, ahora queda claro qué son señales (*t1*, *t2*, *t3* y *t4*), qué son sumadores (*a1* y *a2*), de qué tipo son (*carryRipple* y *carryLookahead*) y cuáles de sus puertos de salida se utilizan (solamente los de resultado, quedando abiertos los de acarreo).

6.1.5 Incorporación del mecanismo de especificación algebraica al formalismo de especificación ecuacional.

Una vez mostrada la utilidad del mecanismo de especificación algebraica,

sólo queda integrarlo con el de especificación ecuacional. Para ello bastará con reemplazar la signatura que forma parte de la especificación ecuacional por una especificación algebraica completa.

6.9 DEFINICIÓN. (Reemplaza a la definición 2.25). Se define especificación ecuacional de un sistema digital como la tupla (*SPEC*, *X*, *Ins*, *Outs*, φ), donde *SPEC* ≡ (*S*, Σ, *E*) es una especificación algebraica, *X* es una familia *S*-indexada de conjuntos de variables disjuntos dos a dos y respecto a Σ, *Ins* y *Outs* son subconjuntos propios de *X* tales que $Ins \cap Outs = \varnothing$, y φ es una función que proyecta, respetando géneros, elementos de la familia *X-Ins* sobre el conjunto de términos de la signatura *Lu*-extendida:

$$\varphi : X\text{-}Ins \rightarrow T_{Lu(\Sigma)}(\, X\text{-}Outs\,) \mid x \in X_s\text{-}Ins_s \Leftrightarrow \varphi(\, x\,) \in T_{Lu(\Sigma),s}(\, X_s\text{-}Outs_s\,)$$

Dado que una especificación algebraica permite concretar un álgebra determinada (el modelo inicial isomorfo a T_{SPEC}), la semántica de la especificación ecuacional ya puede definirse en base a un modelo que tiene una interpretación unánime y no, como se hizo en §2.4.3, en base a un modelo externo que se asumió existía y que sólo el diseñador podría conocer.

6.10 DEFINICIÓN. (Reemplaza a la definición 2.27) Dada la especificación ecuacional (*SPEC*, *X*, *Ins*, *Outs*, φ), se define su semántica como el resultado de la función **C** que se define como:

$$\mathbf{C} : (\, SPEC, X, Ins, Outs, \varphi\,) \rightarrow$$

$$\rightarrow (\, Lu(T_{SPEC})_{s1} \times \overset{p}{\ldots} \times Lu(T_{SPEC})_{sp} \rightarrow Lu(T_{SPEC})_{t1} \times \overset{q}{\ldots} \times Lu(T_{SPEC})_{tq}\,)$$

$$\mathbf{C}[\,(\, \Sigma, (\, in_1, \overset{p}{\ldots}, in_p, out_1, \overset{q}{\ldots}, out_q, z_1, \overset{m}{\ldots}, z_m\,), (\, in_1, \overset{p}{\ldots}, in_p\,), (\, out_1, \overset{q}{\ldots}, out_q\,), \varphi\,)\,] =$$

$$= \lambda(\, in_1, \overset{p}{\ldots}, in_p\,) : Lu(T_{SPEC})_{s1} \times \overset{p}{\ldots} \times Lu(T_{SPEC})_{sp} \cdot$$

$$(\, \mathrm{fix}(\, \lambda(\, out_1, \overset{q}{\ldots}, out_q, z_1, \overset{m}{\ldots}, z_m\,)$$

$$: Lu(T_{SPEC})_{t1} \times \overset{q}{\ldots} \times Lu(T_{SPEC})_{tq} \times Lu(T_{SPEC})_{u1} \times \overset{m}{\ldots} \times Lu(T_{SPEC})_{um} \cdot$$

$$(\, \mathbf{E}[\, \varphi(out_1)\,], \overset{q}{\ldots}, \mathbf{E}[\, \varphi(out_q)\,], \mathbf{E}[\, \varphi(z_1)\,], \overset{m}{\ldots}, \mathbf{E}[\, \varphi(z_m)\,]\,)$$

$$: Lu(T_{SPEC})_{t1} \times \overset{q}{\ldots} \times Lu(T_{SPEC})_{tq} \times Lu(T_{SPEC})_{u1} \times \overset{m}{\ldots} \times Lu(T_{SPEC})_{um}$$

$$)\downarrow 1..q\,) : Lu(T_{SPEC})_{t1} \times \overset{q}{\ldots} \times Lu(T_{SPEC})_{tq}$$

donde $in_i \in Ins$, $out_i \in Outs$, $z_i \in X\text{-}Ins\text{-}Outs$, *p* es el número de puertos de

entrada, q el número de puertos de salida, m el número de señales auxiliares, *fix* es el operador punto fijo, $\downarrow$ es el operador de restricción de tuplas, definido como $(x_1,...,x_n)\downarrow a..b = (x_a,...,x_b)$ con $1 \le a \le b \le n$, y **E** se define como:

$\mathbf{E} : T_{Lu(\Sigma),s}(X) \rightarrow Lu(T_{SPEC})_s$

$\mathbf{E}[\![\sigma(e_1, ..., e_n)]\!] = Lu(T_{SPEC})_\sigma^{w,s}(\mathbf{E}[\![e_1]\!](t), ..., \mathbf{E}[\![e_n]\!](t)),\ \forall \sigma \in Lu(\Sigma)_{w,s}$

$\mathbf{E}[\![x]\!] = \lambda t{:}\mathbf{N}_+.\ x(t) :(T_{SPEC_*}^{\#})_\perp,\ \forall x \in X_s$

La incorporación de la semántica de los símbolos dentro del propio mecanismo de especificación de conductas, aunque no tiene grandes implicaciones prácticas por el momento (espérese hasta §6.3) sí que tiene una importante conclusión teórica: permite evitar las inconsistencias que surgen cuando distintas herramientas tratan de interpretar los operadores.

Por ejemplo, si bien hay cierto consenso en la comunidad investigadora en cómo tratar las distintas construcciones secuenciales del lenguaje VHDL (*while*, *wait*, ...), ese consenso no es tan claro para la interpretación que deben darse a los operadores aritméticos (que por otra parte desde el punto de vista de la síntesis de alto nivel es un aspecto casi más importante). En general, un diseñador suele utilizar en una misma descripción, los propios operadores que facilita el estándar [IEEE87], un conjunto de operadores facilitados por la herramienta que sobrecargan los símbolos predefinidos por el estándar y una colección de funciones facilitadas también por la herramienta para aquellas funciones para las que el lenguaje no pose un símbolo específico (máximo, mínimo, etc).

El primer problema puede surgir cuando el diseñador desea simular su especificación, el estándar sólo establece unos mínimos en los rangos y precisiones de los tipos numéricos†. De este modo, las simulaciones de una

† IEEE Std 1076-1987, sección 3.1.2:
'The range of INTEGER is implementation-dependent, ...'
IEEE Std 1076-1987, sección 3.1.4:
'The range of REAL is host-dependent, ...'
'The representation of floating point types must include a minimum of six decimal

misma especificación pueden ser diferentes de un simulador a otro, e incluso del mismo simulador que corra sobre máquinas diferentes. Cabría pensar que podría evitarse no utilizando los tipos predefinidos, pero no debe olvidarse que aunque sólo se utilicen los operadores que facilita la herramienta, estos finalmente también se definen como composición de los operadores estándar.

El segundo problema (que tiene peor solución) surge cuando el diseñador intenta sintetizar su especificación. En la actualidad la asociación que se hace de operadores de la especificación con módulos hardware es completamente sintáctica. Es decir, se basa en enlaces arbitrarios que aparecen en las bibliotecas de componentes que facilitan las herramientas de síntesis†, bibliotecas que además están descritas con un formalismo completamente diferente al usado para definir conductas. En cualquier caso, esta asociación se hace siempre sin tener en cuenta aspectos semánticos, es decir, sin tener en cuenta cómo está definido el comportamiento del operador para simulación, por lo que diferentes herramientas pueden obtener estructuras RT con comportamientos diferentes, que además pueden ser diferentes de la propia especificación original.

Con la solución aquí propuesta todos esos problemas desaparecen, ya que el propio significado de los símbolos acompaña siempre a la especificación de la conducta, todos los símbolos que pueden aparecer durante el ciclo de diseño pueden formalizarse de un modo uniforme y éstos sólo aparecen si su semántica ha sido definida en la especificación algebraica. Obviamente, si ésta última es incorrecta el circuito que se obtenga no se comportará como el usuario espera. Pero nunca por culpa de la herramienta, ya que lo que ésta

---

digits of precision.'

† El entorno de *Cathedral-2nd* [LCG+90][NGCD91][NCGD92][GeCD93] reserva en cada entrada de la *Library Data-Base* un campo en donde establece el nombre del módulo más abstracto capaz de realizar una operación. Por su parte las *Synthetic Libraries* (o *DesignWare Libraries*) de *Synopsys* [Syno95a][Syno95b] permiten realizar *binding declarations* en donde asociar el nombre de una operación y el nombre de sus argumentos con un nombre de un módulo y el nombre de sus pines.

puede asegurar es que el circuito que se obtenga se comportará correctamente según el modelo inicial de la especificación que le facilitó el diseñador. Sólo si este modelo inicial ha sido correctamente especificado (véase §6.2.1) el circuito será válido para las expectativas del diseñador.

Por otro lado, como anteriormente se decía, las implicaciones prácticas de la incorporación del mecanismo de especificación algebraica son menores que las teóricas, sin embargo merecen destacarse un par de ellas.

La primera es que si se desea que sea útil el algoritmo de simulación presentado en §2.4.4 no basta con reemplazar, tal y como se ha hecho en la definición 6.10, la aparición de A por T_{SPEC}. Lo que un diseñador espera tras una simulación de valores es justo eso, valores, y no términos genéricos incluidos en T_{SPEC}. Por ello, debe ampliarse el algoritmo para que sea capaz de normalizar términos cerrados para que los resultados de una simulación sean elementos de cierto conjunto canónico (por ejemplo, el conjunto C_Σ referido §6.1.2). La normalización automática de términos cerrados se discutirá en §6.3.2.

La segunda, es cómo limitar las ecuaciones que pueden utilizarse vía cualquier regla de aplicación para que el sistema de síntesis por derivación pueda asegurar la corrección de lo que hace. Recuérdese que una de las condiciones de disparo de estas reglas era la condición de que la ecuación fuera válida en la Lu-álgebra soporte, es decir $Lu(A) \models t_L=t_R$. Dado que ahora esta álgebra está completamente especificada, esta condición debería reemplazarse por $Lu(T_{SPEC}) \models t_L=t_R$. Sin embargo, esta condición es demasiado ambiciosa ya que no se ha incorporado a la herramienta ningún mecanismo para demostrar cuándo una ecuación es satisfecha por el modelo inicial. Esta cuestión se discutirá en §6.3. Hasta entonces, la manera más simple de hacerlo es restringir las ecuaciones aplicables solamente a aquéllas que aparecen en la especificación algebraica (que por definición satisface el

modelo inicial), tarea que se reduce a la mera comprobación de pertenencia $t_L=t_R \in E$.

6.2 Implementación de tipos abstractos de datos.

Intuitivamente, implementar un tipo abstracto de datos mediante otro, consiste en reproducir cada uno de los valores del primero mediante valores del segundo y en simular cada una de las operaciones del primero por medio de operaciones del segundo. Es claro que todo circuito de nivel RT reproduce mediante manipulaciones de vectores de bits, operaciones aritméticas especificadas sobre tipos más abstractos de datos. Por ello, si se desea que el paso de especificaciones abstractas a especificaciones más concretas sobre vectores de bits pueda ser tratado matemáticamente, es necesario estudiar cómo formalizar la implementación de tipos.

Para formalizar una implementación de tipos abstractos de datos es necesario establecer una correspondencia entre los valores del tipo implementado, *Abst*, y los valores del tipo implementador, *Impl*. Esta aplicación, que se denomina **función de abstracción**, determina qué elemento del tipo implementador (que llamaremos elemento concreto) representa a cada uno de los objetos del tipo implementado (elementos que llamaremos abstractos). El perfil de toda función de abstracción es:

$$abstr : Impl \rightarrow Abst$$

y, en general, cumple ser:

- **Sobreyectiva**: ya que todos los valores abstractos deben estar representados por algún valor concreto.
- **No necesariamente inyectiva**: ya que pueden existir varios valores concretos que representen a un mismo valor abstracto.
- **Parcial**: ya que pueden existir valores concretos que no se utilicen para representar a ningún valor abstracto.

- **Homomórfica**: la función de abstracción debe conmutar con las operaciones de los tipos, para aquellos valores para los que esté definida.

Obsérvese que, aunque el proceso de refinamiento de tipos siga una línea que comienza en lo abstracto y termina en lo concreto, las anteriores propiedades obligan a que el perfil de esta función tenga dominio *Impl* y codominio *Abst* en lugar de al contrario (ya que entonces no podría modelarse mediante una aplicación).

Esta sección se dedica a la formalización de funciones de abstracción. Así en §6.2.1 se presenta un mecanismo algebraico compatible con el método, también algebraico, de especificación de tipos que se mostró en la sección anterior. En §6.2.2 se aplica dicho mecanismo para la especificación de las representaciones numéricas más habituales en los sistema digitales. Finalmente, §6.2.3 discute aspectos prácticos sobre la incorporación de la propuesta en un sistema de síntesis general y en un sistema de síntesis por derivación en particular.

6.2.1 Implementación algebraica de tipos abstractos de datos.

Se puede hablar de implementación algebraica de tipos abstractos de datos [EKMP82] cuando el tipo abstracto a implementar y el tipo abstracto implementador, se hayan ambos especificados algebraicamente. En ese caso, la propia función de abstracción, si se concretan algunos aspectos semánticos, se puede especificar como si de una operación observadora se tratase.

6.11 **DEFINICIÓN.** Dadas dos especificaciones algebraicas $SPEC_A \equiv (S_A, \Sigma_A, E_A)$ y $SPEC_C \equiv (S_C, \Sigma_C, E_C)$, una **implementación algebraica** de $SPEC_A$ mediante $SPEC_C$ es el par $IMPL \equiv (\Sigma_{IMPL}, E_{IMPL})$ donde:

- Σ_{IMPL} es una familia $S_C^* \times S_A$-indexada de conjuntos de símbolos de

operación.

- E_{IMPL} es una familia de $(\Sigma_A \cup \Sigma_C \cup \Sigma_{IMPL})$-ecuaciones.

El objeto de Σ_{IMPL} es dar soporte sintáctico a las funciones de abstracción, mientras que el de E_{IMPL} es permitir la definición mediante ecuaciones de los comportamientos de dichas funciones, es decir, de los procedimientos para simular las operaciones de Σ_A en términos de las operaciones de Σ_C y Σ_{IMPL}.

Obsérvese que todas las operaciones pertenecientes a Σ_{IMPL} tienen géneros pertenecientes a la especificación a implementar y aridad que varía entre los géneros de la especificación implementadora. No obstante, existen extensiones a este esquema que permiten ampliarlo para que permita operaciones auxiliares que no denoten ninguna función de abstracción [EKMP82].

El perfil más habitual que tomarán las operaciones de Σ_{IMPL} será de tipo proyección, es decir $s_1 \rightarrow s_0$, con $s_1 \in S_C$ y $s_0 \in S_A$, no obstante podrán tomar también perfiles de la forma $s_1 \ldots s_n \rightarrow s_0$, o de la forma $s_1 \ldots s_n.s_0 \rightarrow s_0$, con $s_i \in S_C$ y $s_0 \in S_A$.

EJEMPLO 6.8

Implementemos algebraicamente los números enteros (especificados algebraicamente en el ejemplo 6.5, bajo el nombre de *signed*) mediante el tipo vector de bits que a continuación se especifica.

```
sorts
bit, bitVector
operations
0, 1    :  → bit
not     :  bit → bit
[ □ ]   :  bit → bitVector
□ & □   :  bitVector, bitVector → bitVector
tc      :  bitVector → bitVector
equations for all b : bit; bv1, bv2, bv3 : bitVector
not( 0 ) = 1
not( 1 ) = 0
( ( bv1 & bv2 ) & bv3 ) = ( bv1 & ( bv2 & bv3 ) )
```

not([b]) = [not(b)]
not(bv1 & [b]) = not(bv1) & [not(b)]
tc([b]) = [b] *(eq1)*
tc(bv1 & [0]) = tc(bv1) & [0] *(eq2)*
tc(bv1 & [1]) = not(bv1) & [1]

Para comenzar se da nombre a las funciones de abstracción que permitirán representar mediante vectores de bits los números enteros. Dado que en el ejemplo 6.5 se construyó el género *signed* (que denota al conjunto de los números enteros) a partir del género *unsigned* (que denota al conjunto de los números naturales), definiré una primera función de abstracción auxiliar (*binAbstr*) que permite implementar el género *unsigned* mediante el *bitVector*. A continuación otras dos (*smAbstr* y *tcAbtrs*) para implementar de dos maneras diferentes el género *signed* mediante el *bitVector*.

operations
binAbstr : unsigned → bitVector
smAbstr : signed → bitVector
tcAbstr : signed → bitVector

Una vez declaradas las funciones, debemos definir su comportamiento conociendo que *binAbstr* deberá denotar la representación en binario puro de los números naturales, que *smAbstr* y *tcAbstr* deberán denotar, respectivamente, las representaciones en magnitud y signo y en complemento a dos de los números enteros (suponiendo especificada la operación). El criterio para la elección de las ecuaciones es similar al utilizado en anteriores ejemplos: definir los comportamientos de los operadores en base a cómo transforman a los generadores del género.

equations for all bv : bitVector
binAbstr([0]) = 0
binAbstr([1]) = 1 *(eq3)*
binAbstr(bv & [0]) = binAbstr(bv)0 *(eq4)*
binAbstr(bv & [1]) = binAbstr(bv)1
smAbstr([0] & bv) = +binAbstr(bv)
smAbstr([1] & bv) = -binAbstr(bv)
tcAbstr([0] & bv) = +binAbstr(bv)
tcAbstr([1] & bv) = -binAbstr(tc([1] & bv)) *(eq5)*

Obsérvense, dos aspectos. Primero que la funciones de abstracción definidas para los números enteros son parciales: nótese que no se ha especificado la manera de interpretar el vector [0] ni el vector [1] por considerar que un vector con un único bit no puede representar una cantidad entera. Sin embargo, si se deseara hacerlo bastaría con especificar mediante ecuaciones el convenio a usar:

equations for all bv : bitVector
...
smAbstr([0]) = +0
smAbstr([1]) = -0
tcAbstr([0]) = +0
tcAbstr([1]) = -1

Segundo, merece destacarse cómo la asimetría en los rangos de valores presente en la representación en complemento a dos, queda también especificada mediante la implementación algebraica efectuada:

tcAbstr([1]&[0]) $\rightarrow_{eq5}$ - binAbstr(tc([1]&[0])) $\rightarrow_{eq2}$ - binAbstr(tc([1])&[0]) $\rightarrow$
$\rightarrow_{eq1}$ - binAbstr([1]&[0]) $\rightarrow_{eq4}$ -binAbstr([1])0 $\rightarrow_{eq3}$ -10

Aún siendo bastante gráfica, la definición 6.11 solamente describía el aspecto sintáctico de una implementación algebraica. Sin embargo, esto no basta. Como se discutió a lo largo de §6.1.3, la simple incorporación de una función observadora entre géneros puede provocar que las semánticas de los tipos implicados se corrompan, por ello es necesario describir completamente el proceso semántico que implica una implementación algebraica, para que quede claro lo que debe entenderse.

Básicamente la semántica de una implementación algebraica es un proceso que partiendo de la semántica del tipo implementador $SPEC_C$, que en el enfoque inicial propuesto es T_{SPECc}, alcanza cierta $SPEC_A$-álgebra que llamaremos A_{IMPL} que, si la implementación propuesta es correcta, será isomorfa a la semántica del tipo implementado $SPEC_A$, que en el modelo

inicial no es otra que T_{SPEC_A}.

6.12 Definición. Sea la implementación algebraica $IMPL \equiv (\Sigma_{IMPL}, E_{IMPL})$ que implementa $SPEC_A \equiv (S_A, \Sigma_A, E_A)$ mediante $SPEC_C \equiv (S_C, \Sigma_C, E_C)$. La semántica de dicha implementación, SEM_{IMPL}, se construye mediante la composición de los tres siguientes functores:

$SEM_{IMPL} : T_{SPEC_C} \rightarrow_{SINTESIS} T_{IMPL} \rightarrow_{RESTRICCION} R_{IMPL} \rightarrow_{IDENTIFICACION} A_{IMPL}$

donde cada una de las semánticas implicadas son:

- T_{SPEC_C} es la semántica inicial de la especificación implementadora, $SPEC_C$.
- T_{IMPL} es la semántica inicial de:

  $$SPEC_C + (S_A, \Sigma_{IMPL}, \varnothing) + (\varnothing, \Sigma_A, E_{IMPL})^{\dagger}.$$

- R_{IMPL} es la parte de T_{IMPL} que es generada por las operaciones de Σ_A.
- S_{IMPL} es el álgebra de términos cociente inducida sobre R_{IMPL} por el conjunto de ecuaciones E_A.

La idea del functor de síntesis, es construir los nuevos géneros y operaciones de $SPEC_A$ a partir de los que existen en $SPEC_C$ según se indica en *IMPL*. Por ello primero se enriquece el tipo implementador (el primer término de la suma) con los géneros S_A del tipo a implementar y se generan los nuevos dominios de datos mediante las operaciones de abstracción definidas en Σ_{IMPL} (el segundo término de la suma). Seguidamente se vuelve a enriquecer el tipo implementador, pero esta vez con las operaciones Σ_A del tipo a implementar, definiendo el comportamiento de éstas en base a las operaciones presentes en $SPEC_C$ mediante las ecuaciones E_{IMPL} (el tercer término de la suma). El resultado es un álgebra T_{IMPL} (generada a partir de T_{SPEC_C}) que da soporte tanto a la especificación del tipo implementador $SPEC_C$, como a los nuevos géneros S_A y operaciones Σ_A sin recurrir a la

---

† Donde la suma de dos especificaciones algebráicas es la especificación algebráica cuyos conjunto de géneros, operaciones y ecuaciones son la unión de los respectivos conjuntos de cada una de las dos operaciones originales.

semántica T_{SPEC_A} del tipo a implementar. Esto es así ya que aún no se han añadido las ecuaciones de $SPEC_A$, por lo que la semántica de las operaciones Σ_A está definida por las ecuaciones E_C y E_{IMPL}.

El functor de restricción limita el nuevo tipo de datos obtenido tras la fase de síntesis a todas aquellas operaciones y objetos de datos que son alcanzables mediante operaciones de $SPEC_A$, es decir se 'olvidan' los valores de $SPEC_C$ que no sean accesibles mediante operaciones de $SPEC_A$, al igual que las operaciones de $SPEC_C$ que no se usen para simular las operaciones de $SPEC_A$. El resultado es un álgebra que puede simular el álgebra T_{SPEC_A} pero que puede conservar múltiples representaciones de un mismo valor abstracto.

Para finalizar, el functor de identificación hace congruentes las múltiples representaciones que puedan tener los objetos de $SPEC_A$. Por ello se define A_{IMPL} como el álgebra de términos cociente inducida en R_{IMPL} con respecto a las ecuaciones E_A que existen en la especificación del tipo a implementar. No obstante, el resultado final pudiera contener más datos identificados de los que tenía T_{SPEC_A}, pero esto será un efecto de una incorrecta especificación de *IMPL* y no por una incorrecta definición del proceso semántico de construcción de A_{IMPL}.

6.13 Definición. Se dice que la implementación algebraica $IMPL \equiv (\Sigma_{IMPL}, E_{IMPL})$ que implementa $SPEC_A \equiv (S_A, \Sigma_A, E_A)$ mediante $SPEC_C \equiv (S_C, \Sigma_C, E_C)$ es correcta si:

- $\forall t \in T_{\Sigma_A}, \exists t' \in T_{\Sigma_C+\Sigma_{IMP}}, t \sim_{E_C+E_{IMP}} t'$ (sean congruentes bajo la relación de equivalencia inducida por E_C+E_{IMP}).
- El álgebra A_{IMPL} obtenida por el proceso definido en la definición 6.12 es isomorfa a T_{SPEC_A}.

Obsérvese que la primera condición exige que toda operación de $SPEC_A$ pueda ser simulada mediante la función de abstracción y un conjunto de operaciones de $SPEC_C$. La segunda condición establece que el tipo de datos

A_{IMPL}, una vez finalizado el proceso de implementación, sea indistinguible del tipo de datos a implementar.

6.2.2 Implementación algebraica de objetos de diseño.

En un ciclo de diseño de alto nivel sólo puede distinguirse un paso de refinamiento de tipos, ya que sólo están implicados dos niveles de abstracción: el algorítmico y el RT. Por ello, para comprobar la validez de las técnicas de implementación algebraica basta con especificar las representaciones numéricas.

Para hacerlo, aunque pudiera describirse vía la implementación algebraica de las operaciones algorítmicas mediante módulos de biblioteca (que también fueron especificados algebraicamente), dados los problemas teóricos que se discutirán en el próximo apartado he preferido realizar esa tarea en dos fases: una primera para proyectar tipos algorítmicos sobre tipos de nivel RT mediante implementación algebraica y una segunda para proyectar operaciones de nivel RT sobre módulos hardware mediante especificación algebraica (véase el ejemplo 6.7). Los resultados de la primera de estas fases son la que a continuación se reseñan.

Implementación algebraica de los tipos de nivel algorítmico mediante los tipos de nivel RT.

Utilizando el mecanismo de implementación algebraica se han especificado un total de 9 clases diferentes de representaciones de datos. Las representaciones como **lógica inversa** y como **lógica directa** para implementar el tipo booleano mediante el tipo bit. Las representaciones numéricas **binaria pura**, **one-hot**, **BCD-8421** y **BCD-exceso-3** para implementar el tipo natural y el tipo binario mediante el tipo vector de bits. Las representaciones numéricas

complemento a dos, complemento a uno y magnitud y signo para implementar el tipo entero, el tipo binario con signo y el tipo binario con punto fijo y el tipo binario con punto fijo y signo mediante el tipo vector de bits. Finalmente se ha especificado la representación **ASCII** para implementar el tipo alfanumérico mediante el tipo vector de bits.

Mediante las implementaciones algebraicas realizadas no sólo se ha especificado cómo se proyectan los diferentes objetos de datos de nivel algorítmico sobre cadenas de 0s y 1s, sino que también se ha concretado con qué operación (u operaciones) de manipulación de bits se puede realizar cada funcionalidad abstracta. Un aspecto, éste último, de gran importancia desde la perspectiva de alto nivel, ya que formaliza los caminos para bajar realmente el nivel de abstracción de las especificaciones y abre la puerta a que un sistema de síntesis pueda realizar muchas de las tareas que los diseñadores manuales realizan constantemente: alineaciones de operandos, adaptaciones de anchura, etc.

6.2.3 Sobre el uso del mecanismo de implementación algebraica en un sistema de síntesis formal por derivación.

Desde la perspectiva del diseñador de hardware, el mecanismo de implementación algebraica es un método poderoso para formalizar la proyección de funcionalidades abstractas sobre manipulaciones concretas de bits, de manera que pueda demostrarse la corrección de las mismas. Desde el punto de vista de la síntesis de alto nivel, es la capacidad de formalización la cualidad más interesante del mecanismo ya que, en caso de incorporarse a un sistema automático, permitiría cubrir un importante vacío que existe en las herramientas de síntesis actuales, que no es otro que el primitivo tratamiento que hacen de los tipos numéricos y el escaso aprovechamiento en optimización de las representaciones numéricas.

Los enfoques actuales al problema o bien lo ignoran o bien lo eluden. Los primeros son sistemas que presuponen una especificación homogénea en la que sólo existe un único tipo de datos (y a lo sumo dos si se incluyen los booleanos). Especificación que, una vez diseñada, da lugar a un circuito cuyas señales tienen una anchura común y que comparten una misma representación. Este enfoque requiere que el diseñador, para tratar circuitos reales, homogeinice previamente la especificación o que, tras la fase de síntesis de alto nivel, realice manualmente un proceso de rediseño a nivel RT que tengan en cuenta las posibilidades de una implementación con múltiples anchuras y múltiples representaciones (por ejemplo, realizar simultáneamente en un operador 'largo' dos operaciones 'cortas'). El segundo tipo de sistemas requieren que la especificación de partida sea *bit-true*, es decir, que utilice directamente vectores de bits que serán interpretados según una representación fija (generalmente complemento a dos). Esto hace que tanto la anchura como la representación no sean alteradas por el proceso de síntesis de alto nivel, por lo que los operadores de la especificación resultan ser meras referencias a módulos hardware concretos, referencias que sólo se alteran en sistemas que consideren módulos multifuncionales o múltiples implementaciones de nivel lógico del mismo comportamiento. Por ello, si el diseñador desea optimizar dichos aspectos debe hacerlo manualmente, ya sea antes de especificar o después de diseñar.

Como puede verse, en cualquier caso, la síntesis de alto nivel tal y como se conoce, no es de alto nivel en cuanto a tipos de datos se refiere, ya que si bien acepta descripciones algorítmicas, estas deben construirse usando tipos y operaciones de nivel RT, por lo que la tarea de estos sistemas se reduce a un mero cambio del modelo temporal de un cálculo, pero no a un cambio en el nivel de abstracción de los propios cálculos. Por todo ello, sería interesante elevar realmente el nivel de abstracción de las especificaciones, para poder automatizar algunas tareas que en la actualidad se hacen a mano y, por tanto, abrir el espacio de soluciones en una nueva dirección.

Un posible esquema para acoplar el refinamiento de tipos [KWCM98] dentro del flujo de diseño de alto nivel, sería partir de una especificación que utilice tipos de datos de nivel algorítmico (enteros, caracteres, subconjuntos de reales, etc.) para obtener una especificación *bit-true* sobre la cual comenzar un proceso de síntesis de alto nivel convencional. En nuestro caso, todas las especificaciones serían ecuacionales, el proceso de síntesis de alto nivel convencional sería el presentado en el capítulo 5 y para la fase de refinamiento de tipos habría que idear un mecanismo que utilizara la información descrita por una implementación algebraica.

La tarea de un algoritmo de refinamiento de tipos sería seleccionar óptimamente la representación de cada señal de un diseño y seleccionar con qué operación de manipulación de bits realizar cada operación abstracta, con el objetivo de que exista un conjunto mayor de hardware reusable. La labor de una herramienta formal sería asegurar que todas las elecciones son válidas y que son coherentes entre sí, de manera que la especificación bit-true realice el mismo cálculo que la especificación abstracta.

Para demostrar que una especificación ecuacional *bit-true* de nivel algorítmico implementa a una especificación ecuacional abstracta también de nivel algorítmico existe un problema teórico para la mayor parte de los circuitos interesantes: la fase de refinamiento de tipos no conserva en general el comportamiento, es decir, no es correcta según las nociones de equivalencia y compatibilidad conductual definidas en 3.9 y 3.10.

Este problema nace porque mientras que los dominios de cualquier tipo abstracto no son acotados, los que adopta cualquier señal de un circuito sí lo son. De modo que siempre que se implementa un cálculo abstracto sobre un circuito, existe el riesgo de que aparezcan resultados abstractos que no puedan ser correctamente representados por sus correspondientes valores concretos. ¿Cómo podría evitarse esta eventualidad?: asegurando que el resultado de un cierto cálculo está siempre acotado. Sin embargo, cuando el

cálculo es recursivo o es iterativo, en general es imposible realizar dicha aseveración mediante un estudio estático del algoritmo, es decir, se necesita tener información de la secuencia de estímulos. Siendo más concreto, aún cuando la entrada de un algoritmo esté acotada, el resultado del cálculo puede no estarlo y, que lo esté o no lo esté, depende de los datos de entrada.

Para ilustrar este razonamiento, utilizaré un ejemplo muy simple. Sea la especificación de un acumulador. Un acumulador es un sistema cuya salida en todo ciclo es la suma de todas las entradas al circuito hasta dicho ciclo, por lo que podría especificarse ecuacionalmente como:

```
...
body
z = in + x
x = 0 fby z
out = z
```

Obviamente si se determina que la entrada es entera, cualquier análisis estático del algoritmo permitiría concluir que la salida será homogénea y por tanto también entera. Sin embargo, si se acota la entrada asegurando que los valores de la misma siempre estarán comprendidos entre +100 y -100, no habrá manera, sin conocer la secuencia de entrada, de conocer si existe una cota para los valores de la salida. Desde el punto de vista hardware, esto se traduce en una incapacidad de dimensionar las señales y por tanto de obtener una especificación equivalente *bit-true*.

Por ejemplo, si se asumen las anteriores cotas, la entrada podrá codificarse correctamente con al menos 7 bits, esto permite formular la hipótesis de que será necesario un sumador de 7 bits para implementar correctamente a nivel RT el anterior comportamiento. Por el mismo argumento, dado que el circuito RT tiene que ser conductualmente equivalente a la especificación original, el resultado de la suma (y por consiguiente el tamaño de las señales *z* y *out*) deberá ser de 8 bits para no

perder información. Esto obligará a utilizar un retardador de 8 bits, y dado que la salida del retardador se realimenta a la entrada del sumador este deberá ser también de 8 bits, un resultado que contradice la hipótesis.

En general, para dimensionar correctamente una especificación de nivel RT partiendo de una más abstracta es necesario realizar estudios de convergencia de los algoritmos o estudios de los efectos de los errores de precisión en el rendimiento del sistema. Estos estudios están del todo fuera del alcance de las herramientas de diseño de alto nivel, sin embargo sus resultados sí que pueden incorporarse a la especificación para que la herramienta los utilice. Así, esa información suele aparecer en las especificaciones temporales como un conjunto de ligaduras sobre las representaciones a usar en algunas de las señales intermedias, por lo que la herramienta de diseño tiene libertad para elegir las representaciones del resto de las señales y para elegir la implementación de las operaciones. Además esta información a veces se completa con la especificación de mecanismos de actuación en caso de desbordamiento, en cuyo caso la herramienta debe incorporar hardware extra de detección y corrección de desbordamientos.

Una vez estudiado el problema teórico, es necesario estudiar sus implicaciones prácticas en un sistema de síntesis formal. Está claro que, dado que el sistema de transformación presentado en el capítulo 3 es un sistema correcto (teoremas 3.11 y 3.13), será imposible alcanzar por derivación una especificación ecuacional *bit-true* partiendo de su correspondiente especificación abstracta (ya que como se ha comprobado con anterioridad no son equivalentes). Sin embargo, dado que la implementación algebraica utiliza una sintaxis compatible con la de la especificación algebraica (símbolos de operación y ecuaciones) y esta última, se ha podido integrar sin problemas en el sistema formal de síntesis (§6.1.5), cabría pensar si es posible encontrar algún 'truco' para utilizar dicho sistema formal en la fase de implementación de tipos y realizarla por derivación de un modo coherente, es decir, que si las

asunciones que se hacen antes de comenzar el proceso son ciertas (que las señales abstractas pueden codificarse según cierta representación), los resultados que se obtengan sean aceptables. Pues bien, ese 'truco' existe y a continuación lo expondré.

Sea una implementación algebraica (Σ_{IMPL} , E_{IMPL}). Es posible realizar un proceso de implementación de tipos mediante derivación, si se conoce la representación que debe adoptar cada fuente de datos de una especificación ecuacional (puertos de entrada y retardos arquitectónicos) y si se asume que para toda función de abstracción $\sigma \in (\Sigma_{IMPL})_{s1,s2}$ que implementa elementos del género $s2$ mediante elementos del género $s1$, existe una función $\sigma^{-1} : s2 \rightarrow s1$ tales que ambas son mutuamente inversas, es decir que se cumplen las siguientes propiedades:

$$\forall x_1 \in T_{SPEC,s1},\ \sigma^{-1}(\ \sigma(\ x_1\ )\ ) = x_1 \qquad (1)$$

$$\forall x_2 \in T_{SPEC,s2},\ \sigma(\ \sigma^{-1}(\ x_2\ )\ ) = x_2 \qquad (2)$$

El método es simple. Se parte de una especificación algebraica que incluye los tipos de datos abstractos, los tipos de datos concretos (bit y vector de bits) y las operaciones y ecuaciones de las implementaciones algebraicas. Con dicha especificación algebraica se construye la especificación ecuacional del circuito, de manera que sólo utilice tipos de datos abstractos. A continuación sobre las fuentes de datos, de las que la herramienta debe conocer la representación que adoptan, se aplica la ecuación (2) de derecha a izquierda y se propaga la operación a lo largo de toda la especificación utilizando la homomorfía de las funciones de abstracción (distributividad respecto al resto de operaciones). Cuando dicha función alcance los destinos de datos se aplicará, si es posible, la ecuación (1) de izquierda a derecha.

Como puede observarse el proceso va modificando progresivamente el tipo de datos con el que trabaja el circuito, y es un proceso análogo al utilizado para planificar un circuito (véase §4.5.3) en donde aprovechando la calidad de inversos de los operadores temporales *sample* y *replicate* se va

modificando progresivamente la frecuencia a la que se realizan los cálculos. Obsérvese también que sólo exige conocimiento de cómo implementar las fuentes de datos, por lo que si una especificación no es iterativa (no está realimentada) es posible deducir los tipos de cualquier señal intermedia, tal y como exigía la discusión teórica previa. Por otro lado, en caso de que sea realimentada sólo requiere conocer la representación que deben tener las entradas y los valores que almacenan los retardos arquitectónicos.

En los ejemplos, la información de representación de las fuentes de datos que se obtiene externamente por un análisis del algoritmo, aparecerá por comodidad junto con la declaración del tipo de la señal entre paréntesis. Dicha información indicará la función de abstracción a utilizar y una colección de parámetros que caractericen la proyección particular (por ejemplo el número de bits). He considerado no añadir explícitamente esta información al mecanismo de especificación ecuacional, ya que la propuesta aquí discutida es aún parcial y está adaptada a casos particulares de implementaciones mediante vectores de bits. Su objetivo es mostrar la aplicabilidad de las técnicas de implementación algebraica a un proceso de diseño formal. Cuando futuras investigaciones completen la teoría† y confirmen todas las conjeturas aquí discutidas se incorporará definitivamente al formalismo.

EJEMPLO 6.9

Como ejemplo de los beneficios potenciales que pueden obtenerse de la incorporación del mecanismo de implementación algebraica dentro del sistema de síntesis formal, realicemos por derivación una fase de implementación de tipos, es decir, obtengamos una especificación que utilice

† Por ejemplo, en aspectos tales como el tratamiento de incoherencias, es decir, cuando la especificación de una representación de tipos de una señal contradice una deducción obtenida por el sistema formal a partir de la especificación de las representaciones de otras señales.

tipos de nivel RT partiendo de otra 'equivalente' que utilice tipos de nivel algorítmico.

El sistema a implementar realiza una de las subfunciones que se distinguen en el algoritmo de transcodificación ADPCM (*Adaptative Differential Pulse Code Modulation*) descrito en la recomendación G.721 del CCITT [CCIT88]. Este subsistema se encarga de actualizar uno de los factores de escala que regula la velocidad de adaptación del algoritmo y verifica la siguiente especificación temporal:

$$y_u(t) = (1\text{-}2^{-5}) * y(t) + 2^{-5} * w_i(t)$$

Además la descripción de alto nivel (ya que la recomendación ofrece dos especificaciones, una de alto nivel y otra a nivel RT) acota explícitamente los valores que pueden adoptar algunas de las señales:

$$1.06 \leq y_u(t) \leq 10.00,\ -0.75 \leq w_i(t) \leq 70.13$$

e impone que tanto y_u como y se codifiquen en punto fijo sin signo utilizando 4 bits para la parte decimal y 9 para la fraccionaria, y que w_i se codifique en punto fijo con signo bajo la representación de complemento a dos y utilizando 1 bit de signo, 7 bits para la parte decimal y 4 para la fraccionaria.

Comencemos con una especificación ecuacional del anterior comportamiento temporal:

```
signals
yu, y : fxpUnsigned( fxpAbstr, 13, 9 )
wi : fxpSigned( fxpTcAbstr, 12, 4 )
inports
y, wi
outports
yu
body
yu = abs( ( ( (+(1.0)) - exp(+(10.0),-(101.0)) )*(+y) )
        + ( exp(+(10.0),-(101.0))*wi ) )
```

Obsérvese que se han añadido algunos operadores (signos y valor absoluto) que no aparecían en la especificación temporal. Esto se debe a que en la especificación original existían pequeñas inconsistencias de tipos que deben resolverse al formalizar la ecuación. Así en la especificación original

mientras que w_i se especificaba como con signo, tanto y_u como y se hacían sin él, no dejando clara la manera de realizar las operaciones. Por ello se ha optado por especificarlas del modo menos restrictivo, es decir, haciendo una conversión explícita de todos los operandos fuente, operando con signo y realizando finalmente un valor absoluto del resultado.

Antes de realizar la implementación de tipos, conviene aplicar algunas ecuaciones aritméticas que cumple el tipo *fxpSigned* y que son independientes del tipo de codificación particular que adopten los valores:

$$(x1-x2) * x3 = (x1*x3) - (x2*x3)$$

$$((+(1.0)) * x) = x$$

$$(x1-x2) + x3 = x1 + (x3-x2)$$

$$(x1*x2) - (x1*x3) = x1*(x2 - x3)$$

$$exp(x1,-x2) * x3 = x3 / exp(x1,x2)$$

$$exp(+(10.0),+(101.0)) = +(100000.0)$$

Tras aplicarlas convenientemente (de izquierda a derecha y en ese orden) y expandir la definición de *yu*, se obtiene la siguiente especificación ecuacional:

```
signals
yu, y : fxpUnsigned( fxpAbstr, 13, 9 )
wi : fxpSigned( fxpTcAbstr, 12, 4 )
t1, t2, dif, difsx : fxpSigned
inports
y, wi
outports
yu
body
yu = abs( t1 )
t1 = t2 + difsx
t2 = +y
difsx = dif / (+(100000.0))
dif = wi - t2
```

A continuación, se comienza la implementación de tipos ubicando sobre las fuentes de datos la función de abstracción que se corresponda con la representación elegida en la especificación original, para ello se aplican las

siguientes ecuaciones sobre los puertos de entrada:

$$fxpAbstr(fxpAbstr^{-1}(x, 9), 9) = x$$

$$fxpTcAbstr(fxpTcAbstr^{-1}(x, 4), 4) = x$$

y se fija la representación de las constantes, de manera que, sin perder precisión, sea la más económica en anchura (en este caso punto fijo con signo bajo representación en complemento a dos, con 1 bit de signo, 6 de parte decimal y 0 de parte fraccionaria), para lo que se aplica:

$$+(100000.0) = fxpTcAbstr(0100000, 0)$$

resultando la especificación ecuacional siguiente:

signals
yu, y : fxpUnsigned(fxpAbstr, 13, 9)
wi : fxpSigned(fxpTcAbstr, 12, 4)
t1, t2, dif, difsx : fxpSigned
inports
y, wi
outports
yu
body
yu = abs(t1)
t1 = t2 + difsx
t2 = + (fxpAbstr($fxpAbstr^{-1}$(y, 9), 9))
difsx = dif / fxpTcAbstr(0100000, 0)
dif = fxpTcAbstr($fxpTcAbstr^{-1}$(wi, 4), 4) - t2

Seguidamente se va propagando cada una de las funciones de abstracción dentro de la especificación ecuacional. Con ello será posible reemplazar, según describe la implementación algebraica, cada una de las operaciones abstractas por las correspondientes operaciones de manipulación de vectores de bits. Durante este proceso la representación y anchura de los operandos irá homogeneizándose y la posición de los puntos se alineará de manera que toda la información sobre codificaciones vaya progresivamente desapareciendo de la especificación ecuacional. Por ejemplo si aplicamos la ecuación:

$$+(fxpAbstr(x, n)) = fxpTcAbstr(0 \& x, n)$$

conseguimos implementar el operador signo mediante una concatenación de

un bit por la izquierda. Aplicando ecuaciones análogas, pero notablemente más complicadas†, es posible llegar a la especificación ecuacional siguiente:

```
signals
yu, y : fxpUnsigned( fxpAbstr, 13, 9 )
wi : fxpSigned( fxpTcAbstr, 12, 4 )
t1, t2 : bitVector                    con 14 bits de anchura
diff : bitVector                      con 17 bits de anchura
difsx : bitVector                     con 12 bits de anchura
inports
y, wi
outports
yu
body
yu = fxpAbstr( tcAbs( t1 ), 9 )
t1 = t2 + tcSignExt( difsx, 14 )
t2 = 0 & fxpAbstr^-1( y, 9 )
difsx = rightCut( dif, 5 )
dif = rightFill( 0, fxpTcAbstr^-1( wi, 4 ), 5 ) - tcSignExt( t2, 17 )
```

En donde, como puede observarse, las funciones de abstracción han alcanzado el puerto de salida y han adoptado una forma equivalente a la fijada en la especificación original.

---

† Como por ejemplo:

```
fxpTcAbstr( x, n ) - fxpTcAbstr( y, m ) =
   fxpTcAbstr(
      tcSignExt(
         rightFill( 0, x, abs( n - m ) ),
         width( x ) + abs( n - m ) + max( width( x ), width( y ) ) )
   - tcSignExt(
         rightFill( 0, y, abs( n - m ) ),
         abs( n - m ) + max( width( x ), width( y ) ) ),
      max( n, m ) )
```

que, en nuestro caso, puede llegar a reducirse a:

```
fxpTcAbstr( x, 4 ) - fxpTcAbstr( y, 9 ) =
   fxpTcAbstr( rightFill( 0, x, 5 ) - tcSignExt( y, 17 ), 9 )
```

6.3 Deducción en teorías ecuacionales.

Decidí elegir el formalismo de especificación algebraica no sólo por ser un método que permite describir con rigor las propiedades que caracterizan a un tipo abstracto de datos, sino porque también puede considerarse como una axiomatización de la teoría que verifica dicho tipo. Es decir, por ser un mecanismo que de modo natural permite la prueba formal de cualquier propiedad no especificada explícitamente, pero que sea válida en el tipo abstracto de datos que se especifica.

Esto tiene una clara utilidad desde el punto de vista del diseño hardware, ya que abre una puerta a que el propio sistema de síntesis verifique las ecuaciones que aplica en un proceso de diseño y a que el propio diseñador según su experiencia (y quién sabe si en el futuro la propia herramienta) proponga nuevas conjeturas, que puedan ser comprobadas y que en caso de ser ciertas, puedan añadirse dinámicamente a la base de conocimiento de la herramienta de diseño. Y todo ello sin modificar el formalismo ni el planteamiento global de la herramienta formal.

A continuación se extraerán algunos resultados desarrollados en el campo de la deducción en teorías ecuacionales, se pondrán en relación con la especificación algebraica de tipos abstractos de datos con semántica inicial y se presentarán algunas ideas que se están desarrollando en el campo de automatización de pruebas con vistas a evaluar la posibilidad de su futura integración en el entorno de diseño formal propuesto. En §6.3.1, se discutirán algunas técnicas de deducción específicas dentro del modelo inicial, finalizando con una discusión en §6.3.2 sobre algunas experiencias realizadas en la interoperación del sistema formal de síntesis presentado con un sistema de demostración automática de teoremas.

6.14 **Definición.** Si A es una Σ-álgebra, se define la teoría ecuacional de A, $Th(A)$,

como el conjunto de Σ-ecuaciones que son válidas en *A*. Si *C* es una clase de Σ-álgebras, se define la teoría ecuacional de *C*, *Th*(*C*), como el conjunto de Σ-ecuaciones que son válidas en todas las álgebras que componen dicha clase.

El mecanismo más conocido para decidir formalmente si una ecuación pertenece o no a una teoría ecuacional, es el clásico **cálculo ecuacional**. Dicho cálculo, partiendo de un conjunto *E* de Σ-ecuaciones válidas en un cierto modelo, permite demostrar la validez de otra ecuación si ésta puede derivarse en un número finito de pasos, aplicando cualquiera de las siguientes reglas (donde $t, t', t_i \ldots \in T_\Sigma(X)$, $\sigma \in \Sigma$ y ρ es una substitución cualquiera):

- $t = t$ es derivable *reflexividad*
- si $t = t'$ es derivable, entonces $t' = t$ es derivable *simetría*
- si $t = t'$ y $t' = t''$ son derivables, entonces $t = t''$ es derivable *transitividad*
- si $t_1 = t'_1, \ldots, t_n = t'_n$ son derivables, entonces $\sigma(t_1, \ldots, t_n) = \sigma(t'_1, \ldots, t'_n)$ es derivable *congruencia*
- si $t_L = t_R \in E$ entonces $\hat{\rho}(t_L) = \hat{\rho}(t_R)$ es derivable *aplicación*

Si una ecuación *e* es derivable a partir de un conjunto de ecuaciones *E* utilizando el anterior cálculo, se suele escribir como $E \vdash e$.

Dado que según la definición 6.3, toda especificación algebraica $SPEC \equiv (S, \Sigma, E)$ permite definir la clase de Σ-álgebras Alg_{SPEC} (formada por todas las Σ-álgebras que satisfacen todas las ecuaciones de *E*), el cálculo ecuacional es un mecanismo válido para realizar deducciones en $Th(Alg_{SPEC})$ si se adopta como conjunto de axiomas el conjunto de Σ-ecuaciones *E*.

Otro mecanismo de prueba utilizado habitualmente, es el llamado **cálculo por reescritura**. Este cálculo, a partir de un conjunto de Σ-ecuaciones *E*, construye el conjunto *R* de reglas de reescritura inducidas por la orientación en ambos sentidos de cada una de las ecuaciones de *E* (véase §3.1.1).

Para demostrar la validez de una ecuación bastará con encontrar un

camino de derivación, vía la aplicación de cualquiera de la reglas de R, que conecte a los dos términos que forman la ecuación. En general esta prueba puede tener los llamados **picos y valles**, es decir, puede que para realizarla se tenga que recurrir a ciertos términos intermedios. Una representación gráfica de este efecto en la demostración de $t_1 = t_2$ puede ser: $t_1 \leftarrow ... \leftarrow t_i \rightarrow ... \leftarrow t_j \rightarrow ... \rightarrow t_2$.

Si dos términos t_1 y t_2 están unidos por un camino de derivación por reescritura se suele escribir como $t_1 \leftrightarrow_R^* t_2$, donde $\leftrightarrow_R^*$ representa al cierre reflexivo, simétrico y transitivo de derivaciones vía las reglas de reescritura incluidas en el conjunto R.

A continuación formalizaré algunos resultados útiles sobre los dos métodos de prueba presentados. La demostración puede encontrarse en [EhMa85].

6.15 Teorema. Dada una especificación algebraica $SPEC \equiv (S, \Sigma, E)$, dos términos $t_1, t_2 \in T_\Sigma(X)$ y el conjunto de reglas de reescritura R, inducidas por el conjunto de ecuaciones E, las siguientes afirmaciones son equivalentes:

- $E \vdash (X, t_1, t_2)$ *(1)*
- $\forall A \in Alg_{SPEC}, A \models (X, t_1, t_2)$ *(2)*
- $t_1 \leftrightarrow_R^* t_2$ *(3)*

La equivalencia entre (1) y (2) es un resultado conocido como teorema de Birkhoff, que si bien fue demostrado para especificaciones algebraicas mono-género, puede ser generalizado a especificaciones multi-genero si se asume que los géneros son no vacíos ($\forall s \in S, T_{\Sigma,s} \neq \emptyset$). La implicación (1) $\Rightarrow$ (2) se denomina **corrección del cálculo ecuacional**, y la implicación (2) $\Rightarrow$(1) se denomina **completitud del cálculo ecuacional**.

La equivalencia entre (1) y (3) establece que toda prueba ecuacional puede también realizarse mediante una prueba por reescritura, por lo que ambos métodos son igualmente potentes y pueden utilizarse indistintamente.

La equivalencia entre (2) y (3) garantiza que cualquier proceso de

reescritura sólo produce términos equivalentes, y demuestra la corrección y completitud del cálculo por reescritura.

Si bien los dos mecanismos de deducción propuestos son sencillos y válidos para demostraciones a mano, la completa ausencia de guía tanto en el uso de las reglas (para el caso del cálculo ecuacional) como en la selección de la reescritura a realizar (para el caso del cálculo por reescritura) hace que ambos métodos de prueba no puedan ser eficazmente construidos en la práctica, y por tanto, no sean convenientes para la deducción automática.

En la actualidad, una gran parte del esfuerzo realizado para la automatización de pruebas en teorías ecuacionales, pasa por encontrar técnicas que, partiendo de un conjunto de ecuaciones E, sean capaces de encontrar un conjunto de reglas de reescritura convergente que sea equivalente a E. Un conjunto de reglas de reescritura se llama **convergente** si no existen secuencias de derivación infinitas y si el orden de aplicación de las reglas no importa, es decir, si un término se alcanza tras una secuencia de derivación, cualquier otra secuencia de derivación llegará también al mismo término si tiene el mismo punto de partida. Por otro lado, un conjunto de reglas de reescritura se dice que es equivalente a un conjunto de ecuaciones, si todo lo que se puede probar ecuacionalmente también se puede probar por reescritura.

La diferencia entre las pruebas por reescritura que puede realizar un sistema convergente, respecto de las que puede realizar uno que no lo es (como en general, el presentado en el teorema 6.15), es que toda prueba que contenga picos y valles se transforma en una prueba que contiene a lo sumo un único valle, por lo que su automatización es directa. Si se desea demostrar $t_1 = t_2$, bastará con transformar cada término a su respectiva forma normal (que existe y es única gracias a la convergencia) y después realizar un simple chequeo de equivalencia sintáctica.

Estas técnicas tienen el nombre genérico de *técnicas de complección*. Básicamente toman un conjunto de ecuaciones E, las orientan según un orden de reducción dado (que establece implícitamente la forma de los términos normales), y van produciendo sistemáticamente nuevas reglas, llamadas pares críticos, que son consecuencias ecuacionales válidas del sistema de ecuaciones. Si el procedimiento de complección termina, entonces el conjunto de reglas de reescritura obtenido facilita un procedimiento de decisión para la teoría ecuacional definida por E.

El problema principal para el uso de estas técnicas es que son muy sensibles a la elección del orden de reducción adoptado y que incluso para sistemas de ecuaciones muy simples, los procedimientos de complección pueden no terminar por requerirse un conjunto de reglas infinito, o fallar por haber encontrado un par de pares críticos no comparables según el orden prefijado. No obstante en la literatura, pueden encontrarse infinidad de enfoques que, poniendo algunas restricciones al criterio de confluencia, pueden dar soluciones eficientes en teorías particulares.

6.3.1 Deducción en el modelo inicial.

En los párrafos precedentes se han discutiendo métodos para decidir si una ecuación es válida en la clase formada por todas las *SPEC*-álgebras. Sin embargo, tal y como se estableció en §6.1 se pretende adoptar como semántica de una especificación algebraica el modelo inicial T_{SPEC} y no el modelo laxo Alg_{SPEC}, por lo que los métodos presentados pueden ser insuficientes.

Profundicemos en la cuestión desgranando el verdadero significado de la definición 6.14. Es claro que al ser T_{SPEC} una *SPEC*-álgebra (véase teorema 6.6), toda ecuación válida en la clase Alg_{SPEC} es en particular válida para T_{SPEC}. Sin embargo, es posible que T_{SPEC} satisfaga ecuaciones que no sean

válidas en alguna *SPEC*-álgebra concreta, por lo que la hace inválida en Alg_{SPEC} y por consiguiente (en virtud del teorema 6.15) indemostrable mediante prueba ecuacional o prueba por reescritura.

A continuación resumiré en un teorema algunos de los resultados más interesantes que pueden encontrarse sobre deducciones inductivas (así llamadas a las que se realizan sólo para el modelo inicial). La demostración puede encontrarse en [EhMa85].

6.16 Teorema. Sea una especificación algebraica $SPEC \equiv (S, \Sigma, E)$. Las siguientes afirmaciones son ciertas:

- $Th(Alg_{SPEC}) \subseteq Th(T_{SPEC})$ *(1)*
- $\forall t_1,t_2 \in T_\Sigma,\ T_{SPEC} \models (\varnothing, t_1, t_2) \Leftrightarrow E \vdash (\varnothing, t_1, t_2)$ *(2)*
- $\forall t_1,t_2 \in T_\Sigma(X),\ T_{SPEC} \models (X, t_1, t_2) \Leftrightarrow \forall \rho:X \to T_\Sigma,\ E \vdash (\varnothing, \hat{\rho}(t_1), \hat{\rho}(t_2))$ *(3)*

La afirmación (1) viene a significar que pueden existir ecuaciones válidas en el modelo inicial (llamadas inductivas) que no lo son para cualquier *SPEC*-álgebra. Su implicación directa, como se dijo anteriormente, es que el cálculo ecuacional no es suficiente para demostrar todas las ecuaciones que cumple el modelo inicial. Para solventar ese problema se proponen las dos siguientes afirmaciones.

Por un lado, la propuesta (2) asegura que toda ecuación cerrada que sea válida en el modelo inicial puede ser demostrada usando el cálculo ecuacional. Esto no es de extrañar ya que el modelo inicial está caracterizado por ser típico (véase teorema 6.6), por lo que toda ecuación cerrada cierta para él es cierta para toda *SPEC*-álgebra y por consiguiente deducible ecuacionalmente.

Por otro lado, la afirmación (3) ofrece el método para poder demostrar una propiedad en T_{SPEC} que no sea demostrable vía cálculo ecuacional: demostrarla para todas sus instancias cerradas, es decir para todas las posibles substituciones de sus variables por términos cerrados. Esto es

posible ya que al ser el modelo inicial generado por términos (véase teorema 6.6) sólo interesa que la ecuación se satisfaga para aquellos valores que tengan representación sintáctica, que como se sabe, se corresponden con términos cerrados.

No obstante, cabría pensar que dado que el número de instancias cerradas de una ecuación es en general infinito, la prueba mediante cálculo ecuacional de cada una de ellas por separado es del todo imposible, haciendo inservible la afirmación (3) desde el punto de vista práctico. Sin embargo, esta afirmación esconde una solución alternativa: materializarla mediante inducción estructural sobre los términos cerrados de la substitución, inducción que estará bien definida por ser T_{SPEC} un modelo generado por términos.

EJEMPLO 6.10

En general demostrar propiedades inductivas es más difícil que demostrar propiedades que no lo son. Para ilustrar el proceso que requieren, sea una especificación algebraica de los números naturales con la suma:

sorts
Natural
operations
0 : ⟶ Natural
s : Natural → Natural
□ + □ : Natural, Natural → Natural
ecuations for all n, m : Natural
(n+0) = n *(eq1)*
(n+s(m)) = s(n+m) *(eq2)*

Intentaremos demostrar que el símbolo + es conmutativo para el modelo inicial, en concordancia con la conmutatividad de la suma de números naturales. Pero antes, probaré mediante un contraejemplo cómo ese símbolo no tiene que ser conmutativo en general para toda *SPEC*-álgebra y por tanto al no ser una ecuación válida en Alg_{SPEC}, no puede ser demostrada por simple reescritura.

Sea la siguiente *SPEC*-álgebra, (Z, *∗1*, -), es decir: el soporte del género

Natural, el conjunto de los enteros; el de *s*, la multiplicación por 1; y el + la resta de enteros. Que este modelo es una *SPEC*-álgebra, se deduce de que satisface todas las ecuaciones de la especificación:

- $\forall \mu: X \rightarrow Z,\ \hat{\mu}(n+0) = \mu(n)-0 = \mu(n) = \hat{\mu}(n)$
- $\forall \mu: X \rightarrow Z,\ \hat{\mu}(n+s(m)) = \mu(n)-\mu(m)*1 = \mu(n)-\mu(m) = (\mu(n)-\mu(m))*1 = \hat{\mu}(s(n+m))$

Sin embargo, para esta álgebra el símbolo + no es conmutativo por no ser conmutativa la resta de enteros.

A continuación utilizaré el método de inducción estructural para demostrar que $(n+m) = (m+n)$ es una ecuación válida en el modelo inicial:

(1) Caso base: $(0+m) = (m+0)$ por inducción estructural sobre m:

(1.1) Caso base: $(0+0) = (0+0)$ trivial

(1.2) Hipótesis de inducción: $(0+m_c) = (m_c+0)$ *(hip1)*

(1.3) Demostrar: $(0+s(m_c)) = (s(m_c)+0)$

$(0+s(m_c)) \rightarrow_{eq1} s(0+m_c) \rightarrow_{hip1} s(m_c+0) \rightarrow_{eq1} s(m_c) \leftarrow_{eq1} (s(m_c)+0)$

(2) Hipótesis de inducción $(n_c+m) = (m+n_c)$ *(hip2)*

(3) Demostrar: $(s(n_c)+m) = (m+s(n_c))$ por inducción sobre m

(3.1) Caso base: $(s(n_c)+0) = (0+s(n_c))$ trivial según lo demostrado en (1)

(3.2) Hipótesis de inducción: $(s(n_c)+m_c) = (m_c+s(n_c))$ *(hip3)*

(3.3) Demostrar: $(s(n_c)+s(m_c)) = (s(m_c)+s(n_c))$

$(s(n_c)+s(m_c)) \rightarrow_{eq2} s(s(n_c)+m_c) \rightarrow_{hip3} s(m_c+s(n_c)) \rightarrow_{eq2} s(s(m_c+n_c)) \rightarrow_{hip2}$
$s(s(n_c+m_c)) \leftarrow_{eq2} s(n_c+s(m_c)) \leftarrow_{hip2} s(s(m_c)+n_c) \leftarrow_{eq2} (s(m_c)+s(n_c))$

□

En resumen, cuando se desee demostrar la validez de una ecuación en T_{SPEC} se tienen dos alternativas: o intentar demostrarla vía cálculo ecuacional (o equivalentes) conociendo que existe la posibilidad de no conseguirlo, o recurrir a la inducción estructural sobre los términos de T_{Σ}, conociendo que dentro de esa última alternativa de prueba podrá usarse el cálculo ecuacional

o volver a utilizar la inducción estructural si es necesario.

Para la automatización de pruebas en teorías inductivas, en la literatura existen una colección de particularizaciones de técnicas usadas en teorías más generales ya que aquí el criterio de confluencia se relaja, dejando de ser necesaria la confluencia general y siendo suficiente la confluencia para términos cerrados. Así, pueden encontrarse las técnicas de complección inductiva que en lugar de transformar cualquier prueba ecuacional en una prueba por reescritura, sólo buscan un sistema de reglas de reescritura que lo hagan para pruebas ecuacionales sin variables.

6.3.2 Experiencias en la demostración automática de propiedades de los objetos de diseño.

Dado que la deducción completamente automática en teorías genéricas, tanto ecuacionales como inductivas, son objeto de investigaciones en curso para los que no parece que haya soluciones a corto plazo, la incorporación de un mecanismo de demostración automática dentro del sistema de síntesis formal descrito en esta memoria queda por el momento abierta.

No obstante, para despejar dudas sobre la conveniencia de continuar esta línea de investigación, abordé un objetivo menos ambicioso pero con posibilidades más tempranas de éxito: la integración del sistema de síntesis formal, con un sistema ya existente de demostración semi-automática de teoremas.

El objetivo principal estaba claro: demostrar que las técnicas algebraicas no son un mero artificio teórico sino que son un mecanismo poderoso y simple que permitirá que las herramientas de diseño de alto nivel (y de niveles más altos de abstracción) salgan del status actual. Concretando, el objetivo era demostrar la viabilidad práctica de un entorno de diseño 'semántico' en el que fuera posible especificar los objetos que manipula un

circuito; en el que, para optimizar dicho circuito, fuera posible proponer conjeturas no especificadas, pero que desde la óptica del diseñador parecieran interesantes; en el que dichas conjeturas pudieran ser demostradas automáticamente únicamente en base al conocimiento especificado; y en el que una vez demostradas pudieran integrarse con seguridad en el sistema de síntesis formal para optimizar efectivamente el circuito especificado.

No obstante este no era el único objetivo. Como objetivo secundario me propuse comprobar que era posible automatizar la simulación de valores, es decir, que a partir de una pura especificación de cualquier conjunto de objetos de diseño, era posible normalizar cualquier término cerrado a un representante canónico (elegido en la propia especificación según la manera de escribir las ecuaciones).

El medio para realizar esto era encontrar un demostrador externo que aceptara especificaciones algebraicas como entrada, y que permitiera realizar demostraciones inductivas en base a ellas: el demostrador que hallé fue el *Larch Prover*.

El demostrador de teoremas Larch Prover [GuHo93], es un sistema de demostración semiautomática basado en reescritura ecuacional y desarrollado para la deducción sobre un subconjunto de la lógica de primer orden. Acepta especificaciones formales escritas en Larch [GuHo86], un lenguaje que permite la introducción directa de especificaciones algebraicas. Entre las propiedades que posee, las más interesantes para nuestro propósito son tres:

- Soporta una gran variedad de métodos de prueba, entre los que destaco: normalización, complección, complección módulo conmutatividad, complección módulo asociatividad e inducción estructural. La normalización es útil para la reducción de términos (y en particular cerrados) a sus formas canónicas. La complección permite encontrar automáticamente (si existe) un conjunto de reglas de

reescritura convergente que pueda reproducir cualquier cálculo ecuacional. La complección módulo conmutatividad y asociatividad, permite la complección de teorías que posean estas propiedades (cualquier teoría aritmética) y que por naturaleza poseen ecuaciones que no pueden ser orientadas en un conjunto de reglas de reescritura terminante. La inducción estructural permite la demostración de propiedades específicas del modelo inicial.

- Está diseñado para trabajar eficientemente con grandes conjuntos de ecuaciones (varios cientos de ellas).
- No está completamente automatizado, por lo que permite el diseño de pruebas específicas y la depuración de éstas. De este modo es posible analizar por qué falla una demostración y así entender por qué una conjetura no es cierta o por qué los propios mecanismos automáticos de prueba no encuentran la solución.

No es la primera vez que este demostrador es utilizado en propósitos hardware, de hecho, pueden encontrase referencias de su utilización para la verificación formal de diseños a nivel lógico en [GaGS88][StGG92].

Una vez presentado el medio pasaré a exponer, al igual que en anteriores apartados de desarrollo (§6.1.4 y §6.2.2), cual fue la metodología seguida y la clase de resultados que se obtuvieron.

Así, para conseguir utilizar el demostrador de modo eficiente en los propósitos descritos, se siguieron las siguientes fases:

- Definición de la signatura básica del tipo mediante la declaración de los generadores y propuesta de un conjunto de ecuaciones que delimiten el modelo inicial de la especificación algebraica al de un álgebra isomorfa a la del tipo abstracto de datos que se desea especificar.
- Demostración de que el anterior conjunto de ecuaciones es completo para definir el modelo inicial sobre generadores, lo que se traduce en una demostración de la corrección de la especificación y en una

elección de los términos canónicos representantes de cada una de las clases de equivalencia inducidas por las ecuaciones.

- Incorporación progresiva de nuevos operadores no generadores a la signatura del tipo.
- Definición mediante ecuaciones (orientables de izquierda a derecha, que es el orden de reducción por defecto que adopta el Larch Prover) del comportamiento operacional de los operadores anteriormente incorporados. Este comportamiento operacional debe definir la manera en que cada operador manipula a los generadores del tipo, lo que permitirá que pueda realizarse por normalización cualquier cálculo concreto sobre términos cerrados. Este conjunto de ecuaciones, en general, podrá definir un sistema de reescritura confluente sobre términos cerrados pero no confluente sobre términos cualesquiera. Para asegurar la terminación sobre términos abiertos, deberán probarse otros órdenes de reducción diferentes de la mera orientación de las ecuaciones de izquierda a derecha.
- Validación de la definición operacional de los nuevos operadores. Esta puede realizarse por simulación simbólica, es decir, por normalización de conjuntos suficientes de términos en los que dichos operadores intervengan, o por demostración de un conjunto de propiedades que verifiquen. Si se realiza por simulación esta validación facilitará, además, criterios razonables para demostrar la confluencia sobre términos cerrados (recuérdese que en cualquier caso, esta demostración puede hacerse por métodos no automáticos).
- Propuesta de un conjunto de propiedades de los operadores, formalizadas en términos de ecuaciones en general no cerradas, que sean interesantes desde el punto de vista de diseño.
- Demostración de las anteriores propiedades utilizando la definición operacional de los operadores, la mayor parte de las veces vía inducción estructural y otras veces por normalización de términos

abiertos.

- Incorporación de las propiedades demostradas a la especificación algebraica del tipo para poderlas aplicar posteriormente en un proceso de transformación de especificaciones ecuacionales, o para utilizarlas en futuras demostraciones de nuevas propiedades (que en lugar de hacerse por inducción, puede que entonces se realicen por normalización).

Los resultados que se obtuvieron con esta metodología fueron muy alentadores. Se logró que el demostrador aceptara todas las especificaciones e implementaciones algebraicas descritas en este capítulo. Se consiguió que el cálculo de formas normales de términos cerrados fuera resuelto por el demostrador de manera completamente automática por normalización, por lo que sería posible utilizarlo directamente como núcleo del simulador discutido en §6.1.5 (posible aunque no práctico).

En cuanto a la demostración de nuevas propiedades, los resultados fueron algo menos concluyentes: aunque pudieron demostrarse una gran variedad de propiedades y todas ellas por mecanismos rutinarios de inducción estructural o normalización, la interacción humana fue decisiva para conseguirlo. Esto es así, ya que en muchas de las demostraciones es necesaria la elección de un conjunto adecuado de lemas, tras cuya demostración, es posible la demostración automática de la conjetura. No obstante, es necesario hacer dos puntualizaciones sobre este tema. La primera es que la elección de dichos lemas casi siempre se ha podido hacer en base a un informe de fallo generado por el propio demostrador, por lo que dicha intervención manual no requiere de un excesivo trabajo. La segunda es que dichos informes de fallo pueden producirse tanto durante la demostración de conjeturas ciertas, como durante el intento estéril de demostración de conjeturas falsas; según lo cual, si no se consigue demostrar una propiedad, puede ser tanto porque no se haya encontrado el camino de hacerlo, como porque no exista dicho camino.

Para ilustrar las anteriores conclusiones, mostraré mediante un ejemplo una sesión típica de uso del demostrador, de manera que sea posible hacerse una idea de las posibilidades reales de ese futuro entorno de diseño 'semántico'.

EJEMPLO 6.11

Supóngase que deseamos diseñar un algoritmo que, entre otras operaciones, utilice las de suma y máximo sobre números naturales. Dado que el mecanismo de especificación ecuacional propuesto en esta memoria no posee operadores predefinidos (aparte de los temporales), en una primera fase es necesario o bien utilizar una especificación algebraica ya existente en donde se defina la conducta de dichas operaciones, o bien construir una especificación algebraica nueva. En cualquier caso supongamos que dicha especificación algebraica define las operaciones de la forma más sencilla, es decir, operacionalmente, o lo que es lo mismo, en base a cómo se comportan cuando toman como argumentos a los generadores del tipo (que como vimos en el ejemplo 6.10, pueden ser el cero y el sucesor). Sea por tanto dicha especificación que por conveniencia está descrita en Larch:

```
declare sorts
natural
..
declare operators
0          : -> natural
s          : natural -> natural
__ + __    : natural, natural -> natural
max        : natural, natural -> natural
..
declare variables
n, m, p, q : natural
..
assert
sort natural generated by 0, s;
n + 0 = n;
n + s(m) = s(n + m);
max( 0, n ) = n;
max( n, 0 ) = n;
```

```
max( s( n ), s( m ) ) = s( max( n, m ) );
..
```

Como puede observarse la sintaxis de Larch es completamente compatible con la que hemos estado utilizando en el resto de los ejemplos. Las únicas diferencias respecto a la notación acostumbrada son el símbolo '__', que es equivalente al símbolo □ utilizado para indicar la posición de los argumentos respecto del símbolo de operación; el símbolo '..', que se utiliza como marca de fin de bloque declarativo; y la declaración 'sort natural generated by 0, s;' que se utiliza para indicar al demostrador sobre qué símbolos debe realizar la inducción estructural, que no deben ser otros que los generadores.

Con estas 5 ecuaciones ya es posible evaluar cualquier término cerrado, así para evaluar la expresión natural 2 + *max*(6, *max*(3, 5) + 1) bastará con invocar al comando que calcula la forma normal (nótese que debe utilizarse la sintaxis que se ha definido en la signatura, así por ejemplo, el *2* natural deberá expresarse como el término *s*(*s*(0))):

```
LP6: show normal-form
   s(s(0)) + max( s(s(s(s(s(s(0))))))), max( s(s(s(0))), s(s(s(s(s(0))))) ) + s(0) )
```

A este comando, el demostrador responde calculando efectivamente el número 8. Obsérvese cómo aplica solamente las ecuaciones especificadas:

```
The sequence of reductions leading to the normal form of the term is:

1. s(s(0)) + max(s(s(s(s(s(s(0)))))), max(s(s(s(0))), s(s(s(s(s(0)))))) + s(0))
2. s(s(0)) + max(s(s(s(s(s(s(0)))))), max(s(s(s(0))), s(s(s(s(s(0)))))) + s(0))
3. s(max(s(s(s(s(s(s(0)))))), max(s(s(s(0))), s(s(s(s(s(0)))))) + s(0)) + s(0))
4. s(s(max(s(s(s(s(s(s(0)))))), max(s(s(s(0))), s(s(s(s(s(0)))))) + s(0)) + 0))
5. s(s(max(s(s(s(s(s(s(0)))))), max(s(s(s(0))), s(s(s(s(s(0)))))) + s(0))))
6. s(s(max(s(s(s(s(s(s(0)))))), s(max(s(s(s(0))), s(s(s(s(s(0)))))) + 0))))
7. s(s(s(max(s(s(s(s(s(0)))))), max(s(s(s(0))), s(s(s(s(s(0)))))) + 0))))
8. s(s(s(max(s(s(s(s(s(0)))))), max(s(s(s(0))), s(s(s(s(s(0))))))))))
9. s(s(s(max(s(s(s(s(s(0)))))), s(max(s(s(s(s(0))))), s(s(0))))))))
10. s(s(s(s(max(s(s(s(s(0)))))), max(s(s(s(s(0))))), s(s(0))))))))
11. s(s(s(s(max(s(s(s(s(0)))))), s(max(s(s(s(0))), s(0))))))))
12. s(s(s(s(s(max(s(s(s(0))))), max(s(s(s(0))), s(0)))))))))
13. s(s(s(s(s(max(s(s(s(0)))), s(max(s(s(0)), 0)))))))))
14. s(s(s(s(s(s(max(s(s(0)), max(s(s(0)), 0))))))))·
15. s(s(s(s(s(s(max(s(s(0)), s(s(0)))))))))
16. s(s(s(s(s(s(s(max(s(0), s(0)))))))))
17. s(s(s(s(s(s(s(s(max(0, 0))))))))))
18. s(s(s(s(s(s(s(s(s(0)))))))))
```

Supongamos sin pérdida de generalidad que con anterioridad se demostró, a partir de las 5 ecuaciones especificadas, la conmutatividad en el modelo inicial de las operaciones de suma y de máximo (recuérdese el ejemplo 6.10, en el que se comprobaba que estas propiedades sólo son demostrables por inducción estructural, ya que son propiedades que no cumplen todas las *SPEC*-álgebras). Una vez demostradas, se añadieron a la especificación algebraica original, para ello basta con una nueva sentencia *assert* que las establezca. No obstante, en lugar de fijarlas como ecuaciones, el demostrador *Larch Prover* permite declararlas mediante la palabra reservada *commutative* para que puedan realizarse demostraciones módulo conmutatividad (el problema de las teorías conmutativas fue comentado con anterioridad y básicamente es que no existe un conjunto terminante de reglas de reescritura que permita decidir dicha teoría).

```
assert
commutative +;          equivalente a: n+m = m+n
commutative max;        equivalente a: max(n,m) = max(m,n)
..
```

Una vez especificado el significado de los símbolos, la siguiente fase es utilizarlos para especificar el algoritmo. Supongamos que se construye dicha especificación y que comienza a sintetizarse. En una fase intermedia del diseño (por ejemplo, tras eliminar algunas redundancias) el diseñador observa que en la especificación obtenida existen operadores *max* que toman como argumento dos veces la misma señal. Obviamente, el diseñador piensa que el máximo de una cantidad con respecto a ella misma, es dicha cantidad, idea que podría formalizarse mediante la siguiente ecuación:

$$\max(n, n) = n$$

Sin embargo, dicha ecuación no aparece en la especificación algebraica, por lo que si se aplica directamente sobre la especificación ecuacional sin demostrarla, se corre el riesgo de que pudiera no ser correcta por no haberse tenido en cuenta todas las posibles alternativas, o porque se hubiera formalizado incorrectamente en forma de ecuación. Por ello, antes de

aplicarla se demuestra formalmente.

El método de prueba a utilizar en la mayor parte de los casos es la inducción estructural, así que se ordena al demostrador que intente probar la conjetura siguiendo dicho método y especificándole la variable sobre la que construir la inducción (en este caso la única que existe, la *n*):

```
LP7: prove max( n, n ) = n by induction on n
```

Comando al que el demostrador responde:

```
Attempting to prove conjecture user.9: max(n, n) = n
Creating subgoals for proof by structural induction on 'n'
Basis subgoal:
    Subgoal 1: max(0, 0) = 0
Induction constant: nc
Induction hypothesis:
    userInductHyp.1: max(nc, nc) = nc
Induction subgoal:
    Subgoal 2: max(s(nc), s(nc)) = s(nc)
Attempting to prove level 2 subgoal 1 (basis step) for proof by induction on n
Level 2 subgoal 1 (basis step) for proof by induction on n
[] Proved by normalization.                                              ( 1 )
Attempting to prove level 2 subgoal 2 (induction step) for proof by induction on n
Added hypothesis userInductHyp.1 to the system.                          ( 2 )
Level 2 subgoal 2 (induction step) for proof by induction on n
[] Proved by normalization.                                              ( 3 )
Conjecture user.9
[] Proved by structural induction on 'n'.
```

Como puede observarse ha logrado demostrar la conjetura de un modo completamente automático. Si se estudia el informe se comprobará que el caso base ha sido demostrado por normalización (1) y que tras añadir al sistema la hipótesis de inducción (2), el paso de inducción también se ha podido demostrar por normalización (3). Una vez demostrada, el propio Larch Prover añade dicha ecuación a las ecuaciones que anteriormente han sido especificadas, y el diseñador puede utilizarla sin riesgo, vía la regla de aplicación, para transformar la especificación ecuacional que tenía.

Intentemos ahora algo más complicado. Supóngase que continúa sintetizándose el circuito y nuevamente, en una fase intermedia, el diseñador observa que sería conveniente redistribuir la colocación de los operadores máximo respecto de los operadores suma para reducir el número global de

éstos. Esta idea puede formalizarse mediante una ecuación que describa la distributividad del operador máximo respecto de la suma. Sin embargo, esta conjetura ya no es tan evidente para el diseñador y tampoco ha sido especificada explícitamente con anterioridad, por lo que el uso del demostrador se hace imprescindible. Así que, tras algunas reflexiones, el diseñador decide proponer la siguiente ecuación:

max(max(n, m) + p, max(n, m) + q) = max(n, m) + max(p, q)

Como puede observarse esta ecuación expresa un resultado muy interesante desde el punto de vista de la optimización, ya que aplicada de izquierda a derecha reduciría la cantidad de hardware necesario para un diseño. Pues bien, la orden para probarla vuelve a ser:

```
LP8: prove max(max(n,m)+p,max(n,m)+q) = max(n,m)+max(p,q) by induction on q
```

Como respuesta a esta orden el demostrador comienza intentando demostrar el caso base, que no es otro que:

max(max(n, m) + p, max(n, m) + 0) = max(n, m) + max(p, 0)

sin embargo, en su demostración se encuentra aún con tres variables libres (*n*, *m* y *p*) y no consigue hacerlo, por lo que para la prueba y muestra el mensaje:

```
...
Suspending proof of level 2 subgoal 1 (basis step) for proof by induction on q
```

Este mensaje no significa que la conjetura sea falsa ni que se aborte la prueba, sino que se aplaza hasta que se suministre al demostrador más información de cómo continuarla. El diseñador consciente de que la conjetura poseía 4 variables libres, sabe que para probarla deberán anidarse al menos 4 procesos de inducción estructural (ya que cada proceso se construye sobre una única variable), por ello ordena sucesivamente que se continúe la demostración vía inducción sobre cada una de las variables libres restantes. Este proceso continúa hasta que o bien se demuestre la conjetura o bien se alcance un punto en que no queden variables libres porque todas estén ligadas por sucesivas hipótesis de inducción anidadas.

```
LP9: resume by induction on p
...
Suspending proof of level ...
LP10: resume by induction on m
...
Suspending proof of level ...
LP11: resume by induction on n
...
Suspending proof of level ...
LP12: resume by induction on n
...
Suspending proof of level 5 subgoal 1 (basis step) for proof by induction on n
```

Llegado ese punto, deberá preguntarse al demostrador en qué prueba intermedia se ha quedado bloqueado, para ello se teclea:

```
LP13: qed
Still attempting to prove level 5 subgoal 1 (basis step) for proof by induction on n:
   max(s(max(0, s(mc)) + pc), max(0, s(mc))) = s(max(0, s(mc)) + pc)
```

Como puede observarse, no quedan variables libres (el demostrador indica que una variable está ligada añadiéndole al final del nombre la letra *c*). Por ello, a efectos prácticos, ésta es una ecuación cerrada que no puede probarse. Para conocer el por qué ambos términos no son equivalentes, se solicita que calcule sus formas normales:

```
LP14: show normal-form max(s(max(0, s(mc)) + pc), max(0, s(mc)))
The sequence of reductions leading to the normal form of the term is:
...
6.  s(max(mc, s(pc + mc)))
LP15: show normal-form s(max(0, s(mc)) + pc)
The sequence of reductions leading to the normal form of the term is:
...
3.  s(s(pc + mc))
```

Viendo estas formas normales es sencillo concluir que hace falta demostrar un lema intermedio, que nuevamente se ordena que se pruebe mediante inducción:

```
LP16: prove max( n , s( n + m ) ) = s( n + m ) by induction on n
...
[] Proved by structural induction on 'n'.
Attempting to prove level 5 subgoal 1 (basis step) for proof by induction on n: ( 4 )
  max(s(max(0, s(mc)) + pc), max(0, s(mc))) = s(max(0, s(mc)) + pc)
  Current subgoal: s(max(mc, s(pc + mc))) = s(s(pc + mc))
Level 5 subgoal 1 (basis step) for proof by induction on n
[] Proved by normalization.
...
```

```
Suspending proof of level ...                                          ( 5 )
LP17: qed                                                              ( 6 )
Still attempting to prove level 2 subgoal 2 (induct. step) for proof by induction on q:
  max(max(n, m) + p, max(n, m) + s(qc)) = max(n, m) + max(p, s(qc))
```

Como puede observarse en (4), una vez demostrado el lema, el propio demostrador lo utiliza para continuar automáticamente la prueba que dejó a medias y la continúa hasta que vuelve a encontrar algún problema (5). Nuevamente el diseñador preguntará cual fue el problema (6), y al constatar que quedan variables libres vuelve a la dinámica de prueba por inducción.

```
LP18: resume by induction on p
...
Suspending proof of level ...
LP19: resume by induction on m
...
Suspending proof of level ...
LP20: resume by induction on n
...
Suspending proof of level ...
LP21: resume by induction on n
...
Suspending proof of level ...
LP22: prove max( n , s( n + m ) ) = s( n + m ) by induction on n                ( 7 )
...
Conjecture user.10: max(max(n, m) + p, max(n, m) + q) = max(n, m) + max(p, q)
[] Proved by structural induction on 'q'.
```

Finalmente la prueba finaliza con éxito, y dicha ecuación ya está lista para ser aplicada sobre la especificación ecuacional. Obsérvese que dado que los lemas que se demuestren en el curso de una demostración más general son locales, a veces puede ser necesario, como en (7), volver a demostrarlos en distintas fases de la demostración primaria.

Desde la perspectiva del ejemplo merecen hacerse tres últimas reflexiones. La primera es que si bien en el ejemplo se han probado propiedades relativamente simples, estos resultados están muy por encima de lo que en la actualidad se realiza en el ámbito de la síntesis de alto nivel: inténtese demostrar formalmente estas mismas propiedades a partir de una descripción

de la conducta de las operaciones realizada en cualquier HDL. No obstante, se han podido demostrar propiedades notablemente más complejas que las aquí mostradas.

La segunda reflexión es que en el ejemplo se ha identificado al diseñador con la misma persona que realiza las pruebas de conjeturas e incluso con la misma persona que potencialmente diseñó la herramienta de síntesis. Esta identificación ha sido provocada por afinidad con el autor de esta memoria que ha tenido que ser las tres personas a la vez. Sin embargo, en un mundo real, es razonable que cada una de las anteriores tareas sean realizadas por grupos diferentes de individuos.

La tercera reflexión es que si bien es cierto que en un mundo real existirán grandes bibliotecas de ecuaciones demostradas, aún en ese caso la demostración de nuevas conjeturas será necesaria, ya que a lo largo de ciclos específicos de diseño, podrán surgir nuevas ideas que deban demostrarse. Además, con bibliotecas de componentes en continuo cambio, es necesaria la propuesta y demostración también continua de sus propiedades.

6.4 Ejemplos de la implantación de técnicas algebraicas sobre el sistema de síntesis formal.

Como pasados capítulos, éste termina ilustrando cómo pueden reproducirse sobre el sistema de síntesis formal algunas técnicas de diseño habituales. Estas técnicas pueden realizarse gracias a la incorporación que se ha hecho de los mecanismos algebraicos sobre el mecanismo de especificación ecuacional.

La sección se divide en dos bloques, un primero (§6.4.1-4) en el que se exponen las técnicas más representativas y un segundo (§6.4.5) que

ejemplifica sobre un circuito real la aplicación de las técnicas descritas.

6.4.1 Evaluación de expresiones constantes (constant folding) y propagación de constantes.

La evaluación de expresiones constantes es una técnica que reemplaza en el grafo de flujo a todo operador (habitualmente aritmético) cuyos argumentos sean constantes, por el resultado de la operación. Cuando los resultados de la evaluación vuelven a utilizarse para evaluar las nuevas expresiones que se tornan constantes, el proceso se conoce como propagación de constantes. En sistemas convencionales estas dos técnicas se realizan habitualmente sólo sobre los operadores primitivos del sistema, y cuando todos los argumentos son constantes. Sistemas más complejos permiten evaluar expresiones en las que intervengan subfunciones definidas por el diseñador y expresiones en las que alguno de los argumentos no sea constante.

En un enfoque algebraico, la evaluación de expresiones constantes es equivalente a la normalización de términos cerrados; mientras que la propagación de constantes puede reproducirse sobre el mecanismo de especificación ecuacional, mediante una normalización de términos cerrados y una aplicación posterior de la regla de substitución.

Por otro lado, al no tener ningún operador predefinido en el sistema, las normalizaciones pueden hacerse siempre sobre cualquier operador que se defina. Sin embargo, la evaluación de expresiones parcialmente constantes requerirá en la mayor parte de las ocasiones de un proceso de demostración de la reducción a realizar.

Nótese que este tipo de transformaciones en un sistema convencional suelen ser realizadas por el compilador del lenguaje por lo que sólo pueden aplicarse como una primera fase de optimización. En el sistema aquí propuesto, esta fase no se distingue de cualquier otra ya que finalmente se

reduce a aplicar una ecuación, por lo que puede realizarse en cualquier momento.

Como aspecto curioso, debe destacarse que la inversa de la evaluación de expresiones constantes, también es posible en un sistema basado en especificación algebraica. Esta inversa podría ser útil en casos concretos para usar módulos operativos como caminos de paso y no aumentar el interconexionado de un circuito. Por ejemplo, podría usarse un sumador como 'interconexión' entre dos registros aplicando de derecha a izquierda la ecuación $x+0 = x$ (la misma ecuación que de izquierda a derecha permite evaluar una expresión parcialmente constante).

6.4.2 Reducción de operadores.

La técnica de reducción de operadores consiste en reemplazar un operador con algunos de sus argumentos constantes por otro operador con menor coste computacional, como por ejemplo, reemplazar multiplicaciones y divisiones por potencias de 2, por desplazamientos.

Desde el punto de vista algebraico, esta técnica no se distingue demasiado de la evaluación de expresiones parcialmente constantes. La única diferencia es que el resultado, en lugar de ser un término cerrado, es un término abierto, pero en cualquier caso deberá seguirse un proceso previo de demostración de la reducción que una vez finalizado, permitirá que ésta se aplique con seguridad en tantos diseños como se desee.

6.4.3 Reordenación de operadores.

La reordenación de operadores es una técnica de diseño que modifica las dependencias de datos presentes en un grafo de flujo vía la aplicación de las propiedades conmutativa, asociativa o distributiva que verifican los operadores

aritméticos. La más conocida es la reducción de profundidad en los árboles de operaciones, que transforma cadenas de operadores en árboles de operadores para extraer el máximo paralelismo contenido en una expresión.

La ventaja que aporta el enfoque algebraico frente al convencional, es que el número de operadores que puede demostrarse que es asociativo o conmutativo no se limita a los predefinidos en el lenguaje de partida. De este modo, las potenciales relaciones de distributividad entre operadores son mucho mayores que las que puede detectar un sistema convencional. Así, por ejemplo, el análisis de caminos de ejecución mutuamente exclusivos (véase §4.5.5) se basa en la reubicación del operador *if* y se consigue aplicando su distributividad demostrada respecto a otros.

6.4.4 Selección de tipos (binding) transformacional.

En la fase de *binding* de un proceso de SAN, se asigna a cada operación abstracta de la especificación, un tipo de módulo de biblioteca que se encargará de implementarla. Desde el punto de vista de síntesis, la utilidad de la información de binding es que permite determinar qué operaciones pueden reutilizar hardware. Así, cuantas más operaciones puedan proyectarse sobre un mismo módulo, más operaciones podrán reusarlo y potencialmente podrán ser más baratas las implementaciones que se consigan.

En los sistemas clásicos los posibles enlaces entre operadores y módulos se limitan a establecer una asociación entre nombres de operadores y argumentos con nombres de módulos y pines. En sistemas más complejos, esta asociación no tiene por qué ser 1 a 1 y se permite asociar ciertos atributos al enlace que permitan una mayor versatilidad a la hora de realizar binding. Por ejemplo en [Syn95b] dichos atributos permiten que sobre un sumador con una entrada constante a 1 sea posible proyectar una operación de incremento. En cualquier caso los mecanismos para definir enlaces se

basan en esquemas rígidos predeterminados por el lenguaje utilizado para especificarlos.

En un sistema basado en especificación algebraica, al ser un método que define semánticas y no enlaces, las posibilidades de *binding* se extienden más allá de los que hayan sido explícitamente propuestos en las ecuaciones. Esto permite, que puedan demostrarse enlaces entre operaciones y módulos sólo válidos en diseños particulares, y de esta manera reproducir formalmente lo que hacen los diseñadores manuales para lograr los altos grados de reuso que consiguen.

No obstante, aún en un sistema sin capacidades de demostración de teoremas, el método de expresión ecuacional es mucho más potente que los métodos habituales de expresión de enlaces. Esto se muestra en [MHF96a][MHF96b][MeHF97] y puede comprobarse en el siguiente ejemplo.

EJEMPLO 6.12

Mostremos cómo puede expresarse ecuacionalmente la manera de transformar unas operaciones en otras, de forma que las operaciones de una especificación ecuacional genérica puedan homogeneizarse para que después muchas de ellas puedan ser proyectadas sobre un mismo módulo (se ha hecho con operaciones, pero podría hacerse con módulos o con módulos y operaciones ya que el enfoque algebraico es el mismo para todos).

```
equations for all bv1, bv2 : bitVector
sub( bv1, bv2, 0 ) = add( bv1, not( bv2 ), 1 )
borrow( bv1, bv2, 0 ) = not( carry( bv1, not( bv2 ), 1 ) )
ge( bv1, bv2 ) = not( borrow( bv1, bv2, 0 ) )
ge( bv1, bv2 ) = carry( bv1, not( bv2 ), 1 )
min( bv1, bv2 ) = mux( bv2, bv1, sgn( sub( bv1, bv2, 0 ) ) )
max( bv1, bv2 ) = mux( bv1, bv2, sgn( sub( bv1, bv2, 0 ) ) )
abs( bv1 ) = mux( sub(zero,bv1,0), bv1, sign( sub(zero,bv1,0) )
```

6.4.5 Un ejemplo completo: QAM.

Como ejemplo de la aplicación de las técnicas algebraicas en un proceso de diseño, se sintetizará uno de los filtros transversales reales que forman el ecualizador digital adaptativo con coeficientes complejos que incluye todo bloque demodulador para QAM (*Quadrature Amplitude Modulation*: modulación de amplitud en cuadratura) en radio modems digitales.

Como puede verse en la fig. 6.2, el sistema a diseñar muestra una estructura gruesa formada por cuatro bloques:

- **Decisor**: calcula el valor de salida del circuito, la llamada señal decidida y una señal de error que es la diferencia entre la señal recibida y la señal decidida.

$$\hat{y}(k) = \begin{cases} sgn[y(k)] * \sqrt{M} & |y(k)| \geq \sqrt{M}-1 \\ \lfloor y(k)/2 \rfloor + 1 & en\ otro\ caso \end{cases}$$

$$\hat{e}(k) = y(k) - \hat{y}(k)$$

- **Filtro FIR**: pondera linealmente los valores actual y pasados de la señal de entrada x, usando un conjunto de coeficientes de ecualización c_j, que

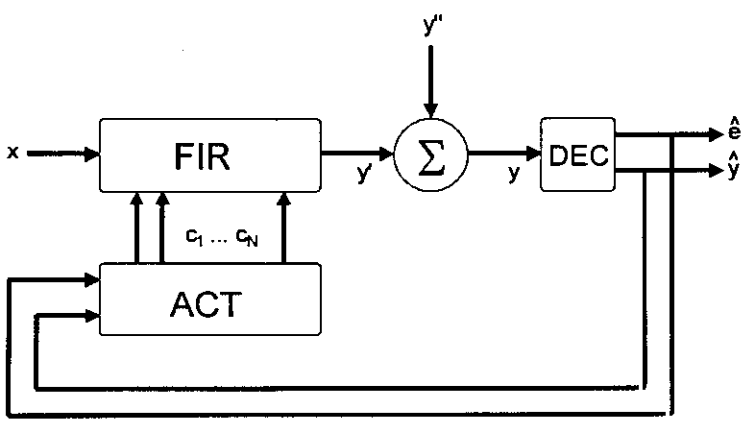


Fig. 6.2: Ecualizador para QAM.

se calculan adaptativamente.

$$y'(k) = \sum_{j=0}^{N-1} c_j(k) * x(k-j)$$

- **Actualizador**: ajusta recursivamente todos los coeficientes de ecualización del filtro FIR, utilizando una versión modificada del algoritmo ZF de Lucky extendido para un ecualizador con valores complejos (versión que en lugar de considerar la señal decidida completa para hacer la correlación con la señal de error, considera simplemente su signo), μ_0 es un factor de escala que controla la tasa de ajuste.

$$c_j(k+1) = c_j(k) - \mu_0 * \hat{e}(k) * sgn[\hat{y}(k-j)] \quad 0 \le j \le N-1$$

- **Sumador**: que permite la integración de este bloque con los otros 3 bloques que forman el ecualizador complejo completo.

$$y(k) = y'(k) - y''(k)$$

Todas las señales son enteras y deben ser representadas en complemento a 2 utilizando 8 bits. Además se asume que $\forall t \le 0$, $x(t)=0$.

Dado que por el momento el mecanismo de especificación ecuacional es plano, no es posible especificar el circuito de manera que refleje su estructura en 4 bloques. No obstante, dado que es declarativo, cada bloque puede especificarse por separado y después juntar en una única descripción y en cualquier orden, las ecuaciones que describen cada subsistema. Así, siguiendo las recomendaciones de §2.4.5 a partir de las anteriores ecuaciones, la especificación ecuacional parcialmente normalizada que describe este comportamiento es la siguiente (para unos valores de N=9, M=16 y μ_0=1).

... especificación algebraica de boolean ...
... especificación algebráica de bit ...
... implementación algebráica de boolean mediante bit ...

```
... especificación algebráica de signed ...
... especificación algebráica de bitVector ...
... implementación algebráica de signed mediante bitVector ...
signals
yg, eg, ygout, egout, a, b, y, yy, yyy, x, M, mu0,
c0, c1, c2, c3, c4, c5, c6, c7, c8,
t0, t1, t2, t3, t4, t5, t6,
x1, x2, x3, x4, x5, x6, x7, x8,
y1, y2, y3, y4, y5, y6, y7, y8 : signed( tcAbstr, 8 )
c : boolean( directLogic )
inports
x, yyy
outports
ygout, egout
body
M = +10000
mu0 = +1
ygout = yg                                              Decisor
egout = eg
yg = if c then a else b
c = abs( y ) >= ( sqrt( M ) - (+1) )
a = sgn( y ) *sqrt( M )
b = floor( y / (+10 ) ) + (+1)
eg = y - yg
yy = (c0 *x) + t0                                       Filtro FIR
t0 = (c1 *x1) + t1
t1 = (c2 *x2) + t2
t2 = (c3 *x3) + t3
t3 = (c4 *x4) + t4
t4 = (c5 *x5) + t5
t5 = (c6 *x6) + t6
t6 = (c7 *x7) + (c8 *x8)
x1 = (+0) fby x
x2 = (+0) fby x1
x3 = (+0) fby x2
x4 = (+0) fby x3
x5 = (+0) fby x4
x6 = (+0) fby x5
x7 = (+0) fby x6
x8 = (+0) fby x7
c0 = (+0) fby ( c0 - (mu0 *(eg *sgn( y ))) )            Actualizador
c1 = (+0) fby ( c1 - (mu0 *(eg *sgn( y1 ))) )
c2 = (+0) fby ( c2 - (mu0 *(eg *sgn( y2 ))) )
c3 = (+0) fby ( c3 - (mu0 *(eg *sgn( y3 ))) )
c4 = (+0) fby ( c4 - (mu0 *(eg *sgn( y4 ))) )
c5 = (+0) fby ( c5 - (mu0 *(eg *sgn( y5 ))) )
c6 = (+0) fby ( c6 - (mu0 *(eg *sgn( y6 ))) )
c7 = (+0) fby ( c7 - (mu0 *(eg *sgn( y7 ))) )
c8 = (+0) fby ( c8 - (mu0 *(eg *sgn( y8 ))) )
y1 = (+0) fby y
y2 = (+0) fby y1
y3 = (+0) fby y2
```

```
y4 = (+0) fby y3
y5 = (+0) fby y4
y6 = (+0) fby y5
y7 = (+0) fby y6
y8 = (+0) fby y7
y = yy + yyy
```

Sumador

Como puede observarse esta especificación contiene 90 operaciones que se reparten en: 25 retardos arquitectónicos, 11 restas, 28 multiplicaciones, 10 sumas, 1 valor absoluto, 2 raíces cuadradas, 1 función suelo, 1 comparación, 1 operación de selección condicional y 10 operaciones de extracción de signo. Obsérvese además, que para realizar una especificación directa, se requiere que algunas funciones se hayan especificado con perfiles peculiares. Por ejemplo, para que sea posible multiplicar un dato de género *signed* por el signo de otro dato del mismo género, es necesario que el resultado de la operación de extracción de signo sea también de género *signed*, luego:

```
sorts
unsigned, signed, ...
operations
...
sgn : signed → signed
equations for all a : unsigned
sgn( +a ) = +1
sgn( -a ) = +0
```

Además, para que después pueda ser correctamente implementada, es necesario que en la implementación algebraica aparezca un ecuación como la siguiente:

```
equations for all bv : bitVector
...
tcAbstr( index( bv, width( bv )-1 ) ) = sgn( tcAbstr( bv ) )
```

Realizamos primero un ejemplo de propagación de constantes y propaguemos *M* y *mu*0. Para ello se substituyen sus apariciones en el cuerpo ecuacional, y realiza externamente una evaluación de las expresiones que las utilizan para obtener las ecuaciones a aplicar (para dicha evaluación se puede

utilizar, por ejemplo, el demostrador de teoremas comentado en la sección anterior).

	body
	...
	c = abs(y) >= (+11)
Substitucion(a, 2.1)	a = sgn(y)*(+100)
AplicacionID(a, a, sqrt(+10000)=+100)	...
Susbtitucion(c, 2.1.1)	c0 = 0 fby (c0 - (eg*sgn(y)))
AplicacionID(c, 2.1, sqrt(+10000)=+100)	c1 = 0 fby (c1 - (eg*sgn(y1)))
AplicacionID(c, 2, (+100)-(+1)=(+11))	c2 = 0 fby (c2 - (eg*sgn(y2)))
Eliminacion(m)	c3 = 0 fby (c3 - (eg*sgn(y3)))
Substitucion(c0, 2.2.1)	c4 = 0 fby (c4 - (eg*sgn(y4)))
AplicacionID(c0, 2.2, 1*x = x)	c5 = 0 fby (c5 - (eg*sgn(y5)))
...	c6 = 0 fby (c6 - (eg*sgn(y6)))
Substitucion(c8, 2.2.1)	c7 = 0 fby (c7 - (eg*sgn(y7)))
AplicacionID(c8, 2.2, 1*x, x)	c8 = 0 fby (c8 - (eg*sgn(y8)))
Eliminacion(mu0)	...

A continuación, es posible eliminar la operación floor del cuerpo ecuacional vía la aplicación de una ecuación que relaciona la división con la división entera. También se adapta la comparación a una comparación con 0.

	body
	...
	c = abs(y) + (-11) >= (+0)
	...
AplicacionID(b, 1, floor(x / y)=(x div y))	b = (y div (+10)) + (+1)
AplicacionID(c, ε, (x>=(+a))=(x+(-a)>=(+0))	...

Seguidamente, se realiza la fase de implementación de tipos tomando como restricción que todas las señales se deben codificar en complemento a 2 utilizando 8 bits. Durante dicha fase (explicada en §6.2.3) se utilizan las ecuaciones siguientes (que aparecen en la implementación algebraica del tipo):

tcAbst(bv1) + tcAbst(bv2) = tcAbst(add(bv1, bv2, 0))
tcAbstr(bv1) * tcAbstr(bv2) = tcAbstr(leftCut(tcMult(bv1, bv2), width(bv1)))
sgn(tcAbstr(bv)) = tcAbstr(leftFill(0, [index(bv, width(bv) - 1)], width(bv) - 1))
tcAbst(bv1) - tcAbst(bv2) = tcAbst(sub(bv1, bv2, 0))
if directLogic(b) then tcAbstr(bv1) else tcAbstr(bv2) = tcAbstr(mux(b, bv2, bv1))
tcAbstr(bv1) div tcAbstr(bv2) = tcAbstr(tcDiv(bv1, bv2))
tcAbstr(bv1) >= tcAbstr(bv2) = tcAbstr(tcGe(bv1, bv2))
tcAbstr(+1) = [0] & ([0] & ([0] & ([0] & ([0] & ([0] & ([0] & [1]))))))

tcAbstr(+100) = [0] & ([0] & ([0] & ([0] & ([0] & ([1] & ([0] & [0]))))))
tcAbstr(+11) = [0] & ([0] & ([0] & ([0] & ([0] & ([0] & ([1] & [1]))))))
tcAbstr(+0) = [0] & ([0] & ([0] & ([0] & ([0] & ([0] & ([0] & [0]))))))
abs(tcAbstr(bv)) =
= tcAbstr(add(xor(bv,[index(bv,width(bv)-1)]),leftFill(0,[0],width(bv)-1),index(bv,width(bv)-1)))

Una vez aplicadas, se obtiene la especificación ecuacional de nivel RT (pero sin que se haya realizado SAN aún), que a continuación se muestra:

...
signals
x, yy : signed(tcAbstr, 8)
yg, eg, ygout, egout, a, b, y, yyy,
c0, c1, c2, c3, c4, c5, c6, c7, c8,
c0x, c1x, c2x, c3x, c4x, c5x, c6x, c7x, c8x, cx
t0, t1, t2, t3, t4, t5, t6,
x1, x2, x3, x4, x5, x6, x7, x8,
y1, y2, y3, y4, y5, y6, y7, y8 : bitVector
c : bit
inports
x, yy
outports
ygout, egout
body
ygout = tcAbstr(yg)
egout = tcAbstr(eg)
yg = mux(c, b, a)
c = tcGe(add(cx, "11111101", 0), "00000000")
cx= add(xor(y,[index(y,7)]),leftFill(0,[0],7),index(y,7))
a = leftCut(tcMult(leftFill(0,[index(y,7)],7), "00000100"), 8)
b = add(tcDiv(y,"00000010"), "00000001", 0)
eg = sub(y, yg, 0)
yy = add(leftCut(tcMult(c0 *tcAbstr^{-1}(x)),8), t0, 0)
t0 = add(leftCut(tcMult(c1,x1),8), t1, 0)
t1 = add(leftCut(tcMult(c2,x2),8), t2, 0)
t2 = add(leftCut(tcMult(c3,x3),8), t3, 0)
t3 = add(leftCut(tcMult(c4,x4),8), t4, 0)
t4 = add(leftCut(tcMult(c5,x5),8), t5, 0)
t5 = add(leftCut(tcMult(c6,x6),8), t6, 0)
t6 = add(leftCut(tcMult(c7,x7),8), leftCut(tcMult(c8,x8),8), 0)
x1 = "00000000" fby tcAbstr^{-1}(x)
x2 = "00000000" fby x1
x3 = "00000000" fby x2
x4 = "00000000" fby x3
x5 = "00000000" fby x4
x6 = "00000000" fby x5
x7 = "00000000" fby x6
x8 = "00000000" fby x7
c0 = "00000000" fby c0x

```
c0x = sub(c0,leftCut(tcMult(eg,leftFill(0,[index(y,7)],7)),8),0)
c1 = "00000000" fby c1x
c1x = sub(c1,leftCut(tcMult(eg,leftFill(0,[index(y1,7)],7)),8),0)
c2 = "00000000" fby c2x
c2x = sub(c2,leftCut(tcMult(eg,leftFill(0,[index(y2,7)],7)),8),0)
c3 = "00000000" fby c3x
c3x = sub(c3,leftCut(tcMult(eg,leftFill(0,[index(y3,7)],7)),8),0)
c4 = "00000000" fby c4x
c4x = sub(c4,leftCut(tcMult(eg,leftFill(0,[index(y4,7)],7)),8),0)
c5 = "00000000" fby c5x
c5x = sub(c5,leftCut(tcMult(eg,leftFill(0,[index(y5,7)],7)),8),0)
c6 = "00000000" fby c6x
c6x = sub(c6,leftCut(tcMult(eg,leftFill(0,[index(y6,7)],7)),8),0)
c7 = "00000000" fby c7x
c7x = sub(c7,leftCut(tcMult(eg,leftFill(0,[index(y7,7)],7)),8),0)
c8 = "00000000" fby c8x
c8x = sub(c8,leftCut(tcMult(eg,leftFill(0,[index(y8,7)],7)),8),0)
y1 = "00000000" fby y
y2 = "00000000" fby y1
y3 = "00000000" fby y2
y4 = "00000000" fby y3
y5 = "00000000" fby y4
y6 = "00000000" fby y5
y7 = "00000000" fby y6
y8 = "00000000" fby y7
y = add( tcAbstr^-1(yy), yyy, 0 )
```

Nuevamente sobre esta descripción pueden aplicarse muchas de las técnicas explicadas pero, esta vez, particularizadas sobre el tipo vector de bits. Así, pueden realizarse algunas reducciones de operadores, aplicando las ecuaciones siguientes:

$$tcMult(\ bv,\ "00000100"\ ) = leftShift(\ tcSignExt(\ bv,\ 8\ ),\ 2\ )$$
$$tcDiv(\ bv,\ "00000010"\ ) = signExtend(\ leftCut(\ rightShift(\ bv,\ 1\ ),\ 1\ ),\ 1\ )$$
$$tcMult(bv,leftFill(0,[b],7)) = and(\ tcSignExt(\ bv,\ 8\ ),\ b\ )$$
$$tcGe(\ bv,\ "00000000"\ ) = not(\ index(\ bv,\ 7\ )\ )$$

pueden homogeneizarse las operaciones y convertir las operaciones de comparación y resta en operaciones de suma:

$$tcGe(\ bv1,\ bv2\ ) = not(\ borrow(\ bv1,\ bv2,\ 0\ )\ )$$
$$sub(\ bv1,\ bv2,\ 0\ ) = add(\ bv1,\ not(\ bv2\ ),\ 1\ )$$
$$borrow(\ bv1,\ bv2,\ 0\ ) = not(\ carry(\ bv1,\ not(\ bv2\ )\ ),\ 1\ )\ )$$

pueden reordenarse las sumas obtenidas:

$$add(\ add(\ bv1,\ bv2,\ b1\ ),\ bv3,\ b2\ ) = add(\ bv1,\ add(\ bv2,\ bv3,\ b2\ ),\ b1\ )$$

y pueden evaluarse expresiones binarias:

$$add(\ add(xor(y,[index(y,7)]),leftFill(0,[0],7),index(y,7)),\ "11111101",\ 0\ )$$

add(xor(y,[index(y,7)]),"11111101", index(y,7))
mux(not(c), bv1, bv2) = mux(c, bv2, bv1)

Tras la aplicación de las anteriores ecuaciones, y de algunas otras para normalizar los operadores, se llega a obtener la siguiente especificación ecuacional optimizada (en donde por conveniencia las cadenas constantes se han escrito entre comillas en lugar de utilizando el operador &).

```
...
body
ygout = tcAbstr(yg)
egout = tcAbstr(eg)
yg = mux( c, a, b )
c = index(add(xor(y,[index(y,7)]),"11111101",index(y,7)),7)
a = "00000[index(y,7)]00"
b = add([index(y,7)]&rightCut(y,1),"00000001",0)
eg = add( y, not( yg ), 1 )
yy = add( leftCut(tcMult(c0,tcAbstr^-1(x)),8), t0, 0 )
t0 = add( leftCut(tcMult(c1,x1),8), t1, 0 )
t1 = add( leftCut(tcMult(c2,x2),8), t2, 0 )
t2 = add( leftCut(tcMult(c3,x3),8), t3, 0 )
t3 = add( leftCut(tcMult(c4,x4),8), t4, 0 )
t4 = add( leftCut(tcMult(c5,x5),8), t5, 0 )
t5 = add( leftCut(tcMult(c6,x6),8), t6, 0 )
t6 = add( leftCut(tcMult(c7,x7),8), leftCut(tcMult(c8,x8),8), 0 )
x1 = "00000000" fby tcAbstr^-1(x)
x2 = "00000000" fby x1
x3 = "00000000" fby x2
x4 = "00000000" fby x3
x5 = "00000000" fby x4
x6 = "00000000" fby x5
x7 = "00000000" fby x6
x8 = "00000000" fby x7
c0 = "00000000" fby add(c0,nand(eg,[index(y,7)]),1)
c1 = "00000000" fby add(c1,nand(eg,[index(y1,7)]),1)
c2 = "00000000" fby add(c2,nand(eg,[index(y2,7)]),1)
c3 = "00000000" fby add(c3,nand(eg,[index(y3,7)]),1)
c4 = "00000000" fby add(c4,nand(eg,[index(y4,7)]),1)
c5 = "00000000" fby add(c5,nand(eg,[index(y5,7)]),1)
c6 = "00000000" fby add(c6,nand(eg,[index(y6,7)]),1)
c7 = "00000000" fby add(c7,nand(eg,[index(y7,7)]),1)
c8 = "00000000" fby add(c8,nand(eg,[index(y8,7)]),1)
y1 = "00000000" fby y
y2 = "00000000" fby y1
y3 = "00000000" fby y2
y4 = "00000000" fby y3
y5 = "00000000" fby y4
y6 = "00000000" fby y5
y7 = "00000000" fby y6
y8 = "00000000" fby y7
y = add( tcAbstr^-1(yy), yyy, 0 )
```

Como puede observarse esta especificación contiene ahora 67 operaciones repartidas en: 25 retardos arquitectónicos, 21 sumadores, 9 multiplicadores, 1 multiplexor, 1 módulo xor vectorial, 1 módulo not vectorial y 9 módulos nand vectoriales. El proceso de optimización llevado a cabo gracias a las técnicas algebraicas se resume a continuación:

Operaciones de la especificación original	Operaciones de la especificación optimizada
11 restas	mediante sumadores y puertas not
2 raíces cuadradas	evaluadas y cableadas como constantes
9 multiplicaciones	evaluadas y cableadas como constantes
1 multiplicación	cableada como desplazamiento
9 multiplicaciones	mediante puertas nand
9 multiplicaciones	mediante multiplicadores
1 función suelo	cableada
1 división	cableada como desplazamiento
10 extracciones de signo	cableadas
1 valor absoluto	mediante sumador y puertas xor
9 sumas	mediante sumadores
1 comparación	cableada
1 selección condicional	mediante multiplexor

Finalmente sobre esta especificación es posible realizar un proceso de síntesis de alto nivel por derivación formal como el descrito en el capítulo 5, tras el que se obtiene la siguiente especificación ecuacional planificada en 23 ciclos:

```
...
body
ygout = 23 >> tcAbstr( t147 )
```

```
egout = 23 >> tcAbstr( t94 )
t133 = tcAbstr^-1( yyy ) << 23
t260 = tcAbstr^-1( x ) << 23
t72 = "00000000" fby mux( t72, t260, t328 )
t76 = "00000000" fby mux( t76, t72, t328 )
t79 = "00000000" fby mux( t79, t76, t328 )
t82 = "00000000" fby mux( t82, t79, t328 )
t85 = "00000000" fby mux( t85, t82, t328 )
t88 = "00000000" fby mux( t88, t85, t328 )
t91 = "00000000" fby mux( t91, t88, t328 )
t94 = "00000000" fby mux( t237, t252, t94, t91, t330 )
t138 = # fby mux( t252, t138, t313 )
t155 = "00000000" fby mux( t155, t138, t328 )
t158 = "00000000" fby mux( t158, t155, t328 )
t161 = "00000000" fby mux( t161, t158, t328 )
t164 = "00000000" fby mux( t164, t161, t328 )
t167 = "00000000" fby mux( t167, t164, t328 )
t170 = "00000000" fby mux( t170, t167, t328 )
t173 = "00000000" fby mux( t173, t170, t328 )
t62 = "00000000" fby mux( t62, t173, t328 )
t184 = "00000000" fby mux( t252, t184, t299 )
t188 = "00000000" fby mux( t252, t188, t300 )
t192 = "00000000" fby mux( t252, t192, t301 )
t196 = "00000000" fby mux( t252, t196, t302 )
t200 = "00000000" fby mux( t252, t200, t303 )
t204 = "00000000" fby mux( t252, t204, t304 )
t208 = "00000000" fby mux( t252, t208, t305 )
t92 = "00000000" fby mux( t252, t92, t314 )
t95 = "00000000" fby mux( t95, t252, t328 )
t147 = # fby mux( t237, t244, t147, t329 )
t244 = mux( index(t252,7), "00000[index(t138,7)]00", t94 )
t252 = add( t272, t273, t274 )
t272 = mux( t147, t94, [index(t94,7)]&rightCut(t94,1),
            xor(t138,[index(t138,7)]), t138, t184, t188, t192, t196,
            t200, t204, t208, t92, t95, t306 )
t273 = mux( t94, t133, "00000001", "11111101", not(t147),
            nand(t94,[index(t138,7)]), nand(t94,[index(t155,7)]),
            nand(t94,[index(t158,7)]), nand(t94,[index(t161,7)]),
            nand(t94,[index(t164,7)]), nand(t94,[index(t167,7)]),
            nand(t94,[index(t170,7)]), nand(t94,[index(t173,7)]),
            nand(t94,[index(t62,7)]), t306 )
t274 = mux( 0, index(t138,7), 1, t308 )
t237 = leftCut( tcMult( t275, t276 ), 8 )
t275 = mux( t95, t92, t208, t204, t200, t196, t192, t188, t184, t310)
t276 = mux( t94, t91, t88, t85, t82, t79, t76, t72, t260, t310 )
t308 = (#||#||00||0||0||0||0||0||0||0||0||0||1||2||2||2||2||2||2||2||2||2||2)
t306 = (#||#||0||0||0||0||0||0||0||0||0||1||2||3||4||5||6||7||8||9||10||11||12||13)
t310 = (0||1||2||3||4||5||6||7||8||#||#||#||#||#||#||#||#||#||#||#||#||#||#||#)
t299 = (1||1||1||1||1||1||1||1||1||1||1||1||1||1||0||1||1||1||1||1||1||1||1)
t300 = (1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||0||1||1||1||1||1||1||1)
t301 = (1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||0||1||1||1||1||1)
t302 = (1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||0||1||1||1||1)
```

```
t303 = (1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||0||1||1||1||1)
t304 = (1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||0||1||1||1)
t305 = (1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||0||1||1)
t313 = (#||#||#||#||#||#||#||#||#||#||0||1||1||1||1||1||1||1||1||1||1||1||1||#)
t314 = (1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||1||0||1)
t328 = (0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||1)
t329 = (#||0||0||0||0||0||0||0||0||0||#||#||#||1||2||2||2||2||2||2||2||2||2||#)
t330 = (0||2||1||1||1||1||1||1||1||1||1||1||1||#||1||2||2||2||2||2||2||2||2||3)
```

El coste final es de 27 retardadores (de los cuales 25 ya estaban preasignados), 1 sumador, 1 multiplicador, 1 módulo xor vectorial, 1 módulo not vectorial, 9 módulos nand vectoriales, y 33 multiplexores.

Capítulo 7

Sobre lo propuesto, lo logrado y lo posible: conclusiones y líneas abiertas

No hay nada más fecundo que la
ignorancia consciente de sí misma.
José Ortega y Gasset

... sobre lo propuesto y lo logrado

En líneas generales, las principales propuestas recogidas en esta memoria son: la definición de un modo simple de especificación formal de conductas (capítulo 2); la construcción de un sistema de transformación de especificaciones que permita sobre ellas el razonamiento ecuacional directo (capítulo 3); la definición de un conjunto de operadores y un conjunto de propiedades que permiten reproducir un proceso de SAN por derivación (capítulo 4); la automatización de dicho proceso de derivación junto con el estudio teórico y experimental de su complejidad (capítulo 5); y la incorporación al mecanismo de especificación de un conjunto de técnicas algebraicas que permitan la transformación semántica de las especificaciones (capítulo 6).

Estas propuestas han posibilitado la obtención de un mecanismo de especificación sencillo, versátil y formal que se adapta a los modos naturales de especificación de los diseñadores; que permite la especificación de la sintaxis y la semántica particular de los operadores que se utilizarán en dicha especificación, y realizarla en base a un conjunto de operadores completamente adaptable a las peculiaridades de la especificación original; que permite la especificación uniforme de conductas, de dominios, de operadores, de representaciones de datos, de proyecciones de operadores y de bibliotecas de módulos; que soporta la descripción de todo el rango de conductas comprendido entre un algoritmo que manipula tipos de datos abstractos y una estructura hardware del tipo 'ruta de datos + controlador' construido por la interconexión de módulos de biblioteca (incluyendo todos los estadios intermedios típicos de un proceso de SAN); que en cualquier caso es simulable; que es directamente manipulable; y que soporta la aplicación de un conjunto muy amplio de técnicas de diseño.

Asimismo se ha demostrado que los procesos de diseño y de verificación formal son tareas complementarias de cuya interacción pueden obtenerse grandes beneficios. De este modo si las decisiones que toma un algoritmo de optimización son adecuadamente utilizadas por un proceso de verificación formal, primero la complejidad de la prueba se reduce drásticamente (de exponencial a polinomial) y segundo el propio verificador puede conseguir un alto grado de resolución en la detección de la decisión incorrecta que permita una depuración de las decisiones de diseño (y no simplemente un mera aserción de corrección o incorrección).

Por otra parte, se ha ejemplificado un método general para el desarrollo de sistemas de síntesis formal que sean capaces de reproducir técnicas de diseño muy variadas. Método que escuetamente consiste en encontrar un conjunto suficiente de operadores que permitan describir ecuacionalmente las bases de una técnica de diseño y proponer un camino de derivación, que

guiado por un conjunto mínimo de decisiones de diseño adoptadas externamente, permita unir la especificación original con el circuito diseñado.

Se ha podido mostrar que para síntesis hardware de alto nivel no son necesarias complejas teorías matemáticas, ni complejos formalismos ya que, a diferencia del software, los problemas subyacentes en este tipo de síntesis son simples. Además se ha demostrado que lo formal no va reñido con lo intuitivo ni con lo práctico.

Se ha abierto una vía hacia un nuevo modo de concebir las herramientas de síntesis que permita trabajar con semánticas en lugar de con sintaxis (tendencia, esta última, que actualmente está siendo sobreexplotada por las herramientas comerciales y que se está traduciendo en una complicación innecesaria de los algoritmos subyacentes).

Y finalmente, no quisiera dejar de señalar que entre las principales aportaciones de esta tesis, seguramente se encuentra la gran cantidad de caminos de investigación que abre en un campo que algunos autores consideraban agotado. Caminos que a continuación paso a exponer.

... sobre lo posible

Aunque en cierto modo el enfoque propuesto es autocontenido, a lo largo de esta memoria se han ido dejando conscientemente bastantes aspectos abiertos que podrán convertirse en años venideros en líneas de investigación a seguir. Estos aspectos pueden clasificarse según los cuatro problemas principales que ha tratado de abordar esta investigación: especificación de conductas, derivación formal, optimización y diseño e integración del sistema.

Sobre la especificación de conductas.

El método de especificación ecuacional propuesto en el capítulo 2 puede considerarse como un método de especificación de nivel 0: aunque permite especificar directamente cualquier conducta, su uso podría hacerse más conveniente si se añadieran algunas facilidades para la estructuración de comportamientos. Así, desde el punto de vista práctico, sería interesante incorporar un conjunto de construcciones polimórficas predefinidas (ya que en este momento deben definirse vía especificación algebraica) que permitieran la compactación de las definiciones, es decir, operadores que facilitaran la descripción de aspectos repetitivos, alternativos o selectivos, o que hicieran menos explícita la anidación de iteraciones o que soportaran la recursividad local (vía términos recursivos). Por otro lado, aún a costa de los problemas teóricos añadidos, también resultaría interesante añadir la posibilidad de jerarquización y parametrización de especificaciones tanto ecuacionales como algebraicas, así como ampliar éstas últimas para que soporten explícitamente formulas universales condicionales de primer orden (actualmente lo hacen implícitamente vía la definición de condicionales) lo que se traduciría en una considerable reducción del tamaño de las mismas.

En la línea de la definición de nuevos operadores temporales con los que enriquecer los cinco presentados en el capítulo 4 (no para facilitar el uso del formalismo, sino para ampliar el número de conductas especificable y el número de técnicas de diseño abarcables), cabría pensar en la incorporación de operadores multifrecuencia (*multirate*) que permitieran la partición explícita de un algoritmo en diferentes procesos que funcionen a diferentes períodos de muestreo. Capacidad que si se amplía con la adición de operadores *sample* y *replicate* con argumentos fraccionarios (de manera que también pueda realizarse la partición implícita), ofrecería la posibilidad de formalizar aspectos tan dispares como la verificación de algoritmos de particionado y el

diseño de una conducta mediante varios circuitos con controladores locales intercomunicados por protocolos simples.

Continuando con la ampliación de operadores, otro conjunto de ellos que merece un estudio en detalle, sería el formado por los operadores de procesado de matrices (dadas las analogías existentes entre la especificación de cálculos espaciales y la especificación de cálculos temporales). Actualmente dichos cálculos se especifican vía indexación explícita (al igual que se hace con el tiempo en las especificaciones temporales). La idea es que puedan especificarse declarativamente vía operadores tipo *up*, *down*, *left*, *right* (equivalentes espaciales al *fby* temporal).

En cualquier caso, obsérvese que conforme crece el número de operadores predefinidos, el número de potenciales propiedades que cumplen y que pueden ser interesante demostrar crece exponencialmente, por lo que se abre un nuevo campo de aplicación para la interacción con los demostradores automáticos.

Asimismo, el modelo temporal asumido obliga a que toda conducta especificada algebraicamente sea combinacional, lo que limita innecesariamente el tipo de bibliotecas especificable. De este modo sería conveniente ampliarlo para que sea posible la existencia de bibliotecas generales formadas por conductas combinacionales y conductas secuenciales, en donde puedan incluirse desde componentes segmentados, hasta circuitos completos prediseñados sobre los que se puedan deducir formalmente sus propiedades.

Otro aspecto de crucial importancia que se ha dejado a medio resolver en el capítulo 6, ha sido el problema de las representaciones de datos. De este modo se impone estudiar las implicaciones teóricas de una incorporación definitiva del mecanismo de implementación algebraica al sistema, al igual que idear maneras de especificar las restricciones de representación que aparecen en una especificación real. Para ello será necesario redefinir los

conceptos de corrección presentados en el capítulo 3 para que toleren las potenciales pérdidas de precisión que suceden en un proceso de refinamiento de tipos.

En esa misma línea será conveniente estudiar una nueva definición de corrección generalizada que permita soportar incluso aquellas técnicas de diseño que no conservan estrictamente el comportamiento, pero cuyos resultados se consideran válidos (*scaling*, *loop-unfolding*, etc.).

Para intuir los potenciales caminos de investigación a más largo plazo, debe observarse que en este momento conviven en un único formalismo dos paradigmas diferentes de especificación de conductas: el funcional (representado por el mecanismo de especificación ecuacional sin tipos) y el axiomático o relacional (representado por el mecanismo de especificación algebraica de tipos). Mientras que el primero se utiliza para describir el algoritmo que debe realizar un circuito, el segundo se usa para describir las propiedades (e indirectamente el comportamiento) que verifican las funciones combinacionales con las que se construye la descripción del mismo. Esta distinción se ha realizado siguiendo los dictados de la conveniencia: el paradigma funcional se adapta eficientemente a los modos de especificar habituales entre diseñadores, y el paradigma axiomático se adapta a los requisitos de manipulabilidad de las herramientas de diseño. Sin embargo, es previsible que un futuro tiendan a converger en un único paradigma que, conociendo la trayectoria software, será probablemente el axiomático. De este modo, podrán especificarse conductas de un modo mucho más abstracto en términos de relaciones entre objetos de diseño mucho más complicados que los actuales.

Probablemente esta convergencia terminará por alcanzar desde abajo a las herramientas actuales de síntesis automática de software, lo que permitirá que, por compartir un único mecanismo de especificación, puedan acoplarse en un único entorno múltiples técnicas de refinamiento tanto software como

hardware y puedan concebirse herramientas de codiseño más generales, en donde pueda comenzarse aún sin tener el propio software desarrollado. Obsérvese que lo que actualmente se denomina codiseño y que parte de un programa procedural, es más reingeniería que codiseño hardware/software, ya que se parte de un software ya diseñado, del cuál extrae un fragmento que se rediseña mediante técnicas hardware.

Sobre derivación formal.

Aparte de todas las técnicas de diseño mostradas en §3.3, §4.5 y §6.4, es posible reproducir por derivación formal un número mayor de ellas, por ejemplo, se han tenido buenas experiencias en el diseño por derivación de circuitos con control segmentado y en la ingeniería inversa de diseños RT (obtención una especificación algorítmica a partir de un circuito RT planificado y asignado, sin tener ninguna información del proceso de diseño efectuado). En cualquier caso, estos logros permiten esperar que a medio plazo un gran número de técnicas de diseño puedan ser reproducidas formalmente, y lo que es más interesante, que todas ellas compartan un único formalismo y una manera común de efectuarse, por lo que su integración en un único flujo de diseño será sencilla. Un flujo de diseño del que habrán desaparecido algunas de las fronteras ficticias que en la actualidad se imponen.

Por otro lado, otro campo de posible desarrollo futuro se centra en la mejora del algoritmo de guiado automático propuesto en §5.2. En dicha sección, se especificó de manera que fuese intuitivo, pero a costa de perder eficiencia. Así que la búsqueda de nuevos teoremas más generales, a la vez que la búsqueda que nuevas formas de agrupar las transformaciones, concluirá necesariamente en métodos más eficientes de cálculo. Asimismo, la naturaleza declarativa del formalismo de especificación (que hace que el significado de cada definición no dependa de aspectos globales), permite que las transformaciones puedan implementarse en base a la aplicación

concurrente de transformaciones locales a conjuntos disjuntos de ecuaciones. Si a esto se une a que el modo de descripción del algoritmo de guiado es no determinista (las derivaciones pueden aplicarse en cualquier orden) hace pensar que una implementación paralela del sistema de derivación formal, sea una alternativa interesante a tener en cuenta.

También debe destacarse que en §5.2 se desarrolló únicamente un algoritmo de guiado de un proceso de SAN por derivación. Sin embargo, ninguna otra técnica de diseño por derivación, de las muchas presentadas en esta memoria, se ha automatizado. Luego si se desea ampliar el número de implementaciones que se puedan verificar formalmente, es imperativo que se apliquen técnicas similares a las presentadas en dicha sección, para cada uno de los métodos de diseño propuestos en §3.3, §4.5 y §6.4.

Para finalizar nótese que en esta memoria sólo se ha utilizado el sistema de transformación cómo un método de diseño formal, sin embargo, no habría mayor problema en considerarlo como un método de observación formal de las características de un comportamiento, o si se quiere como un método de prueba formal de que un comportamiento cumple ciertas propiedades. La investigación en este sentido consistiría tanto en encontrar una forma normal que decidiera dicha propiedad, como en encontrar un camino por derivación que la demuestre. Por ejemplo, aspectos que sería interesante observar pueden ser aquellos que permitan decidir qué operaciones son mútuamente exclusivas en un comportamiento. En cualquier caso, esta nueva aplicación del sistema de transformación es simplemente un cambio de perspectiva ya que, de hecho, la síntesis formal puede considerarse como un método de observación de la corrección de un diseño.

Sobre optimización.

Una sospecha que el capítulo 5 dejaba entrever, es que el sistema por derivación pone de manifiesto la existencia de muchos más aspectos que pueden optimizarse en un diseño, de los que un algoritmo de síntesis convencional tiene en cuenta. Y esto es así no porque el algoritmo convencional los considere indiferentes sino porque permanecen ocultos, o difíciles de expresar en las estructuras de datos que utilizan. Esta sospecha comienza a confirmarse en el capítulo 6 al comprobar el inabarcable espectro de soluciones que abre un entorno semántico de síntesis, y es previsible que futuros trabajos, que amplíen el número de técnicas que reproduce el sistema formal, terminen por confirmarla definitivamente. Esto permite dos reflexiones:

La primera es que a corto plazo serán necesarios algoritmos más versátiles de síntesis, presumiblemente estocásticos, que permitan discernir entre el número creciente de aspectos interrelacionados que podrán optimizarse. Piénsese, por ejemplo, en las consecuencias que podrían tener sobre las fases clásicas de SAN la incorporación de capacidades de retemporización o de proyección completamente libre de funcionalidades sobre comportamientos RT. En cualquier caso, es posible predecir que deberán ser algoritmos menos complejos pero más ágiles, de manera que permitan explorar simultáneamente las posibilidades globales de optimización, en lugar de tratar de obtener localmente buenos resultados en aspectos muy particulares.

La segunda es que el enfoque de derivación dirigido sólo será válido mientras que las herramientas sean simples. En cuanto sean capaces de tomar decisiones semánticas, la manera de expresarlas será el propio proceso de deducción efectuado, por lo que implícitamente los propios algoritmos de síntesis deberán soportar alguna clase de núcleo formal sobre el que razonar. Por ello considero interesante el estudio de esa nueva

generación de algoritmos, tomando como núcleo formal el aquí presentado, es decir, algoritmos de optimización que basen sus decisiones en conocimiento ecuacionalmente expresado.

Sobre integración de sistemas.

A lo largo de los capítulos precedentes se han ido subrayando los potenciales beneficios que pueden derivarse de la simbiosis del sistema de síntesis formal con otro tipo de herramientas. Por ello la investigación sobre mecanismos de interoperación parece un tema con prometedoras perspectivas.

A muy corto plazo, el mecanismo descrito en §5.2.1 posibilitará la verificación formal de los resultados obtenidos por herramientas comerciales basadas en algoritmos potencialmente incorrectos. Por otro lado, como se comentó en §6.2.3, si se desea formalizar el proceso de refinamiento de tipos numéricos, será necesaria la interacción del sistema formal con herramientas o algoritmos de análisis de precisión con los que, a partir las restricciones que facilita una especificación original, el sistema formal pueda derivar y verificar las representaciones de cada una de las señales intermedias de un diseño. Para finalizar, la tercera y última simbiosis deberá ser con demostradores de teoremas, gracias a la cual será posible desarrollar sistemas semánticos de síntesis como los ejemplificados en el anterior capítulo.

Apéndice A: Lambda notaciones†

El lambda cálculo de Church [Chur51] es un lenguaje matemático creado para el estudio de las funciones: de su definición y de su aplicación. Nació en el intento de encontrar una teoría coherente sobre la cual se pudieran derivar los fundamentos de las matemáticas y terminó siendo capaz de describir cualquier función computable. Mediante λ-cálculo es posible escribir y manipular formalmente expresiones construidas a partir de la aplicación de unas expresiones a otras, donde toda expresión denota una función computable que, generalmente, se suele identificar con la propia función.

La potencia expresiva del λ-cálculo hace que éste sea el fundamento de los meta-lenguajes usados en semántica denotacional. Estos meta-lenguajes, o λ-notaciones, se caracterizan por añadir al cálculo básico nuevas capacidades que facilitan su uso.

Este apéndice resume algunas nociones básicas que permiten entender la λ-notación utilizada en la presente memoria. Esta ha sido utilizada tanto para describir los comportamientos de los operadores temporales (véase §4.1), como para concretar la semántica formal de una especificación ecuacional (véase definición 2.27), como para fundamentar muchas de las demostraciones efectuadas (véase §4.2).

† Extractado de [Alli86].

A.1 Lambda cálculo sin tipos.

Suponiendo que existe un conjunto suficiente de identificadores de variable, la sintaxis del λ-cálculo sin tipos es:

ε ::=	ε ε	*aplicación*
	| λξ.ε	*abstracción*
	| ξ	*identificador de variable*
	| (ε)	

En una expresión del tipo **aplicación**, la expresión de la izquierda representa la función que toma como argumento actual a la expresión de la derecha. En una expresión de tipo **abstracción**, el identificador de variable representa el nombre del único argumento formal que toma la función representada por la expresión de la derecha. Dicha expresión incluirá habitualmente copias del argumento formal para fijar el lugar de aplicación del parámetro aún no facilitado.

Mediante abstracciones sólo es posible definir funciones anónimas. Sin embargo, y por conveniencia, se suele permitir la definición de expresiones con nombre. Una expresión con nombre siempre debe ser definida antes de usarse, quedando prohibida la recursión explícita.

Además, aunque mediante λ-cálculo puro es posible construir expresiones que se comporten como enteros o como sus operaciones básicas, comúnmente se suelen añadir algunos identificadores constantes para expresar cómodamente dichos valores y operaciones, operaciones que podrán aparecer dentro de la expresión en posiciones distintas de la prefija y que no será necesario definir explícitamente. Incluso es habitual que, para un propósito concreto, se fijen un conjunto de identificadores que denoten las operaciones más comunes, consiguiendo con ello simplificar la interpretación de las expresiones.

Dentro de una expresión una variable puede aparecer libre o ligada. Intuitivamente una variable ligada será aquella que esté afectada por una abstracción; cuando se aplique un argumento a dicha abstracción la variable ligada será reemplazada. Formalmente:

A.1 **DEFINICIÓN.** Se define variable libre como:

- ξ es libre en ξ (pero no en otra variable).
- ξ es libre en ε ε' si lo es en ε o en ε' (o en ambos).
- ξ es libre en $\lambda\xi'.\varepsilon$ si ξ y ξ' son diferentes y ξ es libre en ε.
- ξ es libre en (ε) si lo es en ε.

A.2 **DEFINICIÓN.** Se define variable ligada como:

- no hay variables ligadas en una expresión formada por una única variable.
- ξ está ligada en ε ε' si lo está en ε o en ε' (o en ambos).
- ξ está ligada en $\lambda\xi'.\varepsilon$ si ξ y ξ' la misma variable o si ξ está ligada en ε.
- ξ está ligada en (ε) si lo está en ε.

A.2 Conversiones.

La manipulación sintáctica de λ-expresiones está gobernada por las reglas de conversión. Estas definen formalmente un procedimiento para la aplicación de funciones y determinan las reglas de ámbito de variables ligadas.

A.3 **DEFINICIÓN.** Sea x una variable y e una expresión. La **substitución** de x por e, $[e/x]$, se define como:

$$
\begin{aligned}
\varepsilon\ \varepsilon'\ [e/x] &= (\ \varepsilon\ [e/x]\ )\ (\ \varepsilon'\ [e/x]\ ) \\
\lambda\xi.\varepsilon\ [e/x] &= \lambda\xi.\varepsilon && \text{si } \xi \equiv x && (2) \\
&= \lambda\xi'.(\ \varepsilon\ [\xi'/\xi]\ [e/x]\ ) && \text{si } \xi \neq x,\ x \text{ es libre } \varepsilon \text{ y } \xi \text{ es libre en } e, \\
& && \text{siendo } \xi' \text{ una nueva variable} && (3) \\
&= \lambda\xi.(\ \varepsilon\ [e/x]\ ) && \text{en otro caso}
\end{aligned}
$$

$\xi\ [e/x]$ $= e$ si $\xi \equiv x$, substituye la variable

$= \xi$ en otro caso

$(\ \varepsilon\ )\ [e/x]$ $= (\ \varepsilon\ [e/x]\ )$

La línea (2) indica que los enlaces locales son prioritarios, de manera que cuando se encuentra una abstracción en la que la variable ligada es la misma que la que debe ser reemplazada, el cuerpo no se modifica. La línea (3) previene una colisión de nombres entre $\lambda\xi.\varepsilon$ y e, de manera que la substitución no provoque que quede ligada una variable que no lo estaba antes de la transformación.

A.4 **DEFINICIÓN.** Se define la **conversión** α como:

α :- si y no es una variable libre en ε entonces $\lambda x.\ \varepsilon = \lambda y.\ \varepsilon\ [y/x]$

Con ella se permite el renombrado sistemático de parámetros formales, ya que simplemente denotan posiciones a reemplazar por valores dentro del cuerpo de una abstracción.

A.5 **DEFINICIÓN.** Se define la **conversión** β como:

$$\beta :- (\ \lambda x.\varepsilon\ )\ a = \varepsilon\ [a/x]$$

Con ella se define el procedimiento de aplicación de una función mediante la sustitución de un parámetro formal por uno actual.

A.6 **DEFINICIÓN.** Se define la **conversión** η como:

η :- si x no es una variable libre en ε entonces $\varepsilon = \lambda x.\ \varepsilon\ x$

Con ella se permite tratar cualquier expresión como una función.

Si la λ-notación permite definiciones o constantes, deberán añadirse otras reglas de conversión que aseguren su correcto uso. Estas nuevas reglas deberán replicar el comportamiento de las básicas en el caso de que las definiciones o constantes fueran reemplazadas por la λ-expresión que representan.

A.3 Evaluación.

Las reglas de conversión permiten evaluar o reducir λ-expresiones. Reducir una λ-expresión es intentar eliminar el mayor número de abstracciones posible. Una expresión que no puede ser reducida se dice que está expresada en **forma normal** y puede ser considerada como el resultado de una expresión. Aunque no todas las expresiones tienen forma normal, el **primer teorema de Church-Rosser** establece que no es posible que dos secuencias de reducciones terminen obteniendo resultados diferentes: o bien ambas terminan en formas normales equivalentes (salvo conversiones α), o al menos una de ellas no termina. Además, según el **segundo teorema de Church-Rosser**, existe un orden de reducción concreto llamado **orden normal** que siempre termina si cualquier otro orden lo hace.

El orden normal de reducción se realiza aplicando conversiones β sobre la abstracción más a la izquierda que esté involucrada en una aplicación, de manera que se substituyan copias no reducidas de la expresión argumento. La idea en que se fundamenta este orden es en retrasar la evaluación de los parámetros actuales hasta que sean efectivamente usados dentro de una función. Una idea en contraste con el **orden aplicativo** de reducción, que reduce primero los argumentos y después los aplica a la función.

A.4 Múltiples argumentos.

La sintaxis presentada no permite expresiones con múltiples argumentos, es decir, abstracciones con más de una variable enlazada ni aplicaciones simultáneas de más de una expresión. Sin embargo, ello no quiere decir que no puedan representarse funciones con más de un parámetro. Para ello basta conocer un mecanismo introducido por Schönfinkel [Schö24] y utilizado extensivamente por Curry [CuFe68], que permite corresponder cualquier

función de dos o más argumentos con otra equivalente de orden superior que los toma de uno en uno. Una función de orden superior es aquella que produce funciones como resultado o que toma como argumento funciones.

Este mecanismo, por ejemplo, permite corresponder la función de 2 argumentos $f = \lambda(x,y).e$ con su versión **currificada** $f = \lambda x.\lambda y.e$, donde f es una función de un argumento cuyo resultado, $(f\ x)$, es otra función de un argumento cuyo resultado, $((f\ x)\ y)$, es a su vez igual a la función original, $(f\ (x,y))$. Así, si se admite la anterior notación para representar las funciones de múltiples argumentos, es posible definir las funciones de currificación y descurrificación de funciones de 2 parámetros utilizando el propio λ-cálculo:

$$curry = \lambda f.\lambda x.\lambda y.(f\ (x,y))$$
$$curry^{-1} = \lambda f.\lambda (x,y).((f\ x)\ y)$$

Conociendo esta equivalencia, es indistinto utilizar una u otra notación, será el contexto de uso el que recomiende usar por conveniencia la versión de múltiples argumentos, $\lambda(x_1, \overset{n}{\ldots}, x_n).e$, o la versión currificada $\lambda x_1. \overset{n}{\ldots} \lambda x_n.e$

A.5 Recursividad.

Una función recursiva es aquella que se invoca a si misma dentro de su propia definición. Dado que el λ-cálculo expresa esencialmente funciones anónimas y las funciones con nombre exigen que sean definidas antes de ser usadas, parece difícil que mediante una expresión sea posible efectuar una autorreferencia.

Sin embargo, la autorreferencia explícita no es necesaria para expresar recursividad en λ-cálculo. En λ-cálculo pueden definirse funciones que permiten duplicar cualquier otra función, de manera que ésta pueda componerse consigo misma un número determinado de veces. Este tipo de

funciones se llaman combinadores y en especial existe un de ellos llamado **operador punto fijo**, *Y*, que se define como:

$$Y = \lambda h.((\lambda x.(h (x\ x))) (\lambda x.(h (x\ x))))$$

Este operador posee la propiedad de que para toda función de orden superior *F*, $Y\ F = F (Y\ F)$, lo que la hace capaz de encontrar un punto fijo de la función argumento si éste existe. En general, un **punto fijo de una función**† es un valor en el dominio de la función que es proyectado sobre sí mismo por dicha función. No todas las funciones tienen exactamente un punto fijo: pueden tenerlo, no tenerlo o tener infinitos.

A.6 Lambda-cálculo con tipos.

La sintaxis del lambda cálculo con tipos es:

ε ::=	$\varepsilon\ \varepsilon$	*aplicación*
	$\mid \lambda\xi :\tau\ .\ \varepsilon :\tau$	*abstracción con declaración de tipos*
	$\mid \xi$	*identificador de variable*
	$\mid (\varepsilon)$	

Una expresión de tipo abstracción permite ahora establecer el tipo de la expresión y el tipo del argumento formal. La sintaxis para la expresión del tipo, suponiendo que existen un número suficiente de identificadores para nombrar dominios elementales y variables de tipo, es la siguiente:

τ ::=	υ	*identificadores de dominios elementales*
	$\mid \tau \times \tau$	*producto directo de dominios*
	$\mid \tau + \tau$	*unión o suma disjunta de dominios*
	$\mid \tau \rightarrow \tau$	*dominio de funciones*
	$\mid {}'\xi$	*identificador de variable de tipo*
	$\mid \mu'\xi.\tau$	*operador punto fijo*

† En el apéndice B puede encontrarse una definición más exacta.

Una λ-expresión representa una función, y una expresión de tipo representa un domino. El operador μ sólo es aplicable a tipos y permite definir tipos recursivos. Los tipos recursivos se incluyen para responder a las necesidades de tipificación de λ-expresiones auto-aplicativas. Sólo así, por ejemplo, podemos asignar coherentemente tipos a las componentes del operador punto fijo:

$Y : (\tau \to \tau) \to \tau \qquad h : \tau \to \tau \qquad x : \mu'\xi.'\xi \to \tau$ *para algún tipo* τ

Además, mediante variables de tipo libres, podemos expresar funciones polimórficas en las que la variable libre se interpreta como universalmente cuantificada. Una función polimórfica es aquella que define un mismo 'esquema' de comportamiento que se realiza de un modo independiente del tipo concreto del argumento.

Para comprobar que una expresión está correctamente tipificada suelen existir algunas reglas de chequeo estático de tipos, tales como:

- $\varepsilon : \tau \to \tau' \wedge \varepsilon' : \tau \vdash (\varepsilon \, \varepsilon') : \tau'$
- $\xi : \tau \Rightarrow \varepsilon : \tau' \vdash (\lambda \xi : \tau \, . \, \varepsilon : \tau') : \tau \to \tau'$

Apéndice B: Dominios

Si se asume que los tipos de datos pueden denotar conjuntos arbitrarios de elementos, es posible que se obtengan contradicciones. Esto quiere decir que expresiones escritas utilizando cierta λ-notación, especialmente expresiones recursivas, pueden quedar vacuas o ambiguas en el sentido de que no representan a ninguna función concreta. Para corregir estas posibles inconsistencias, Scott [Scot71] desarrolló una teoría que ofrece un modelo consistente para el λ-cálculo. Esta teoría básicamente establece un conjunto de condiciones que permiten restringir la clase de funciones que una λ-expresión (en especial recursiva) puede denotar, permitiendo asociarle aquel modelo que de modo natural le daría significado: el que contiene la mínima cantidad de información necesaria.

EJEMPLO B.1

Considérese la siguiente definición recursiva que utiliza autorreferencia:

$$f(\,x\,) = \textit{if } x = 0 \textit{ then } 1 \textit{ else if } x = 1 \textit{ then } f(\,3\,) \textit{ else } f(\,x\text{-}2\,)$$

o, lo que es lo mismo, el punto fijo de la λ-expresión:

$$\lambda f.\ \lambda x.\ \textit{if } x = 0 \textit{ then } 1 \textit{ else if } x = 1 \textit{ then } f(\,3\,) \textit{ else } f(\,x\text{-}2\,)$$

La experiencia calculando funciones recursivas nos hace suponer que la definición denota una función que vale 1 si el argumento es par, e indefinido si el argumento es impar (ya que se queda atrapada en un lazo recursivo). Sin embargo, matemáticamente existen otras funciones que son soluciones

de dicha definición. Por ejemplo: la función constante 1 o cualquier otra que calcule valores arbitrarios en los puntos no definidos.

¿Cuál de ellas debe ser considerada como modelo de la definición?, obviamente la primera de todas que, respecto a las demás, tiene la particularidad de sólo contener la cantidad mínima de información implicada en la definición (no contiene elementos extra).

En este apéndice se presentarán las principales líneas de la teoría de Scott, que ha sido utilizada en la presente memoria para formalizar la semántica del mecanismo de especificación ecuacional (véase definición 2.27).

B.1 Ordenes parciales: cotas y extremales.

Para poder comparar la cantidad de información que contiene un valor (o función), es necesario que el espacio de valores este ordenado. A continuación se presentan algunos conceptos básicos.

B.1 **DEFINICIÓN.** Un conjunto, S, está **parcialmente ordenado** por la relación $\sqsubset$ (léase más débil que), si para todo $x,y,z \in S$ se cumplen las siguientes propiedades:

- $(x \sqsubset y \wedge y \sqsubset z) \Rightarrow x \sqsubset z$ *transitividad*
- $(x \sqsubset y \wedge y \sqsubset x) \Rightarrow x \equiv z$ *antisimetría*
- $x \sqsubset x$ *reflexividad*

Obsérvese que no se exige que todo par de elementos de S estén relacionados; de hecho, se dice que dos elementos $x, y \in S$ son incomparables si $x \not\sqsubset y \wedge y \not\sqsubset x$.

B.2 **Definición.** Sea S un conjunto parcialmente ordenado y A un subconjunto no vacío de S. Se dice que $z \in S$ es una **cota inferior** de A si $\forall x \in A,\ z \sqsubset x$.

B.3 **Definición.** Sea S un conjunto parcialmente ordenado y A un subconjunto no vacío de S. Se dice que $z \in S$ es una **cota superior** de A si $\forall x \in A,\ x \sqsubset z$.

Para conjuntos, ordenes parciales y subconjuntos arbitrarios pueden no existir cotas.

B.4 **Definición.** Sea S un conjunto parcialmente ordenado y A un subconjunto no vacío de S. Se dice que $z \in S$ es el **ínfimo** o extremo inferior de A, $\sqcap A$, si es la mayor de las cotas inferiores de A.

B.5 **Definición.** Sea S un conjunto parcialmente ordenado y A un subconjunto no vacío de S. Se dice que $z \in S$ es el **supremo** o extremo superior de A, $\sqcup A$, si es la menor de las cotas superiores de A.

Un conjunto que no está acotado superiormente (respectivamente, inferiormente) no puede tener supremo (respectivamente, ínfimo). Además, que un conjunto esté acotado no implica necesariamente que tenga extremos.

B.6 **Definición.** Sea S un conjunto parcialmente ordenado. Se dice que un subconjunto de S no vacío es una **cadena** si dos elementos cualesquiera de dicho subconjunto son comparables.

B.2 Retículos.

No sólo es necesario que el espacio de valores esté ordenado, sino que, además, posea cierta estructura que permita la existencia de puntos fijos.

B.7 **Definición.** Un **retículo** es un conjunto parcialmente ordenado en el que todo subconjunto finito no vacío tiene supremo e ínfimo.

Nótese que aún no es completo ya que no todos los subconjuntos tienen que tener extremales.

B.8 **DEFINICIÓN.** Un retículo es **completo** si todo subconjunto tiene supremo e ínfimo.

B.9 **COROLARIO.** Todo retículo finito es completo.

B.10 **COROLARIO.** Todo retículo completo tiene **fondo**, $\perp$, (un elemento más débil que ningún otro) y tiene *cima*, $\top$, (tal que cualquier otro elemento es más débil que él).

Dado que nuestro propósito es dar semántica a las λ-expresiones, la definición habitual de retículo completo resulta demasiado exigente. Para construir nuestro modelo basta con admitir conjuntos parcialmente ordenados en los cuales toda cadena tenga supremo. La razón es que, si bien desde un punto de vista semántico el elemento fondo nos es útil (ya que denota la ausencia de información), el elemento cima no es necesario (ya que denota información contradictoria). Así, de ahora en adelante, asumiremos una definición de retículo completo más simple y no completamente consensuada. Un desarrollo de los conceptos que a continuación se presentarán utilizando la definición general, pueden encontrarse en [Stoy81][Scot76].

B.11 **DEFINICIÓN.** Un **retículo completo** es un orden parcial en el que todo subconjunto tiene ínfimo y toda cadena tiene supremo.

B.12 **COROLARIO.** Todo retículo completo tiene fondo, $\perp$.

Cualquier domino elemental puede ser dotado de estructura de retículo. Basta con añadir un nuevo elemento y un orden parcial en donde todos los elementos originales sean incomparables entre sí.

B.13 **DEFINICIÓN.** Sea un conjunto S. Definimos el **dominio plano** $S_\perp$ como la tupla $(S, \sqsubset, \perp)$, donde $\perp$ es el elemento fondo y $\sqsubset$ es una relación de orden parcial llamada **aproximación** y definida como:

$$\forall x,y \in S_\perp \; x \sqsubset y \Leftrightarrow (x \equiv \perp \wedge x \equiv y)$$

B.2.1 Combinación de retículos completos.

Una vez introducidos los retículos completos, consideremos las maneras de combinarlos para generar nuevos retículos completos.

B.14 **Definición.** Sean los retículos completos $(A, \sqsubseteq_A, \bot_A)$ y $(B, \sqsubseteq_B, \bot_B)$. Se define **producto directo no estricto** como $(A\times B, \sqsubseteq_{A\times B}, \bot_{A\times B})$, donde:

$$A\times B = \{ (a,b) \mid a\in A \wedge b\in B \} \text{ y } (a,b) \sqsubseteq_{A\times B} (c,d) \Leftrightarrow a \sqsubseteq_A c \wedge b \sqsubseteq_B d$$

B.15 **Definición.** Se define **producto directo estricto** de los retículos $(A, \sqsubseteq_A, \bot_A)$ y $(B, \sqsubseteq_B, \bot_B)$ como el producto directo no estricto en el que además se han identificado las tuplas parcialmente definidas:

$$\forall a\in A,\ \forall b\in B,\ (a,\bot_A) = (b,\bot_B) = \bot_{A\times B}$$

B.16 **Definición.** Sean los retículos completos $(A, \sqsubseteq_A, \bot_A)$ y $(B, \sqsubseteq_B, \bot_B)$. Se define **suma disjunta coaligada** como $(A+B, \sqsubseteq_{A+B}, \bot_{A+B})$, donde:

$$A+B = \{ (a,1) \mid a\in A \} \cup \{ (b,2) \mid b\in B \},$$
$$(\bot_A,1) = (\bot_B,1) = \bot_{A+B},$$
$$(a,1) \sqsubseteq_{A+B} (a',1) \Leftrightarrow a \sqsubseteq_A a',$$
$$(b,1) \sqsubseteq_{A+B} (b',1) \Leftrightarrow b \sqsubseteq_A b'$$
$$\text{y } (a,1) \text{ y } (b,2) \text{ son incomparables}$$

B.17 **Definición.** Sean los retículos completos $(A, \sqsubseteq_A, \bot_A)$ y $(B, \sqsubseteq_B, \bot_B)$. Se define **suma disjunta separada** como $(A+B, \sqsubseteq_{A+B}, \bot_{A+B})$, donde:

$$A+B = \{ (a,1) \mid a\in A \} \cup \{ (b,2) \mid b\in B \} \cup \{ \bot_{A+B} \}$$
$$(a,1) \sqsubseteq_{A+B} (a',1) \Leftrightarrow a \sqsubseteq_A a',$$
$$(b,1) \sqsubseteq_{A+B} (b',1) \Leftrightarrow b \sqsubseteq_A b'$$
$$\bot_{A+B} \sqsubseteq_{A+B} (\bot_A,1) \text{ y } \bot_{A+B} \sqsubseteq_{A+B} (\bot_A,2)$$

B.18 **Proposición.** El producto directo no estricto, el producto directo estricto, la suma disjunta coaligada y la suma disjunta separada de retículos completos, son retículos completos.

Habitualmente, si no se dice lo contrario, se utiliza por defecto el producto no estricto y la suma separada.

Funciones sobre retículos completos.

Dado que para una misma definición recursiva existen varias funciones que pueden ser consideradas como modelo de la misma, ordenaremos dichas funciones para localizar una de ellas como el modelo que buscamos.

B.19 **DEFINICIÓN.** Si f y g son funciones de domino D y codominio D', donde D y D' son retículos completos, entonces $f \sqsubset g$ si para todo $x \in D$, $f(x) \sqsubset g(x)$.

Obsérvese que si D' es un domino plano, entonces $f \sqsubset g$ si g está definida para más argumentos que f (i.e. si para algún x, $g(x) \neq \perp$ y $f(x) = \perp$), y ambas devuelven los mismos resultados para aquellos argumentos para los cuales están definidas. La función más débil de todas (la menos definida), es aquella que está indefinida en todo su domino: la función $\lambda x.\perp$.

Si ordenamos el espacio de funciones atendiendo al contenido de información de sus elementos, podemos restringir el conjunto de funciones que debemos de considerar como candidatas a ser modelo de una definición recursiva. Así, es razonable asumir que sólo debemos admitir aquellas funciones tales que, si se conoce más información sobre los argumentos, es posible obtener más información sobre el resultado, es decir, sólo admitir funciones monótonas.

B.20 **DEFINICIÓN.** Una función f es **monótona** si $x \sqsubset y$ implica $f(x) \sqsubset f(y)$.

Matemáticamente una función es una proyección de argumentos sobre resultados y la mayor parte de ellas son proyecciones infinitas por estar definidas sobre dominios infinitos. Siendo infinitas no pueden estar representadas explícitamente mediante una tabla dentro de un dispositivo físico; por tanto son representadas por un algoritmo que es ejecutado. Un

algoritmo no puede dar explícitamente la proyección realizada por la función, pero si nos permite generar cualquier porción finita de la misma. De hecho, lo que se realiza cuando la ejecución avanza y se aplican más y más argumentos, es construir gradualmente una aproximación de la función real que es lo suficientemente buena para ciertos propósitos.

Así nuevamente debemos restringir nuestras funciones candidatas a aquellas cuyo comportamiento pueda ser predicho a partir del comportamiento de sus aproximaciones, es decir, aquellas que son continuas.

B.21 **Definición.** Una función $f: S \to S$ es **continua** si $f(\sqcup X) = \sqcup \{ f(x) \mid x \in X \}$ para toda cadena, X, de S.

B.22 **Proposición.** Toda función continua es monótona.

B.23 **Proposición.** La composición de funciones continuas es continua.

A partir de cualquier función definida sobre dominios elementales, es posible obtener otra función equivalente que sea continua sobre los correspondientes dominios planos. Dicho proceso se denomina extensión estricta.

B.24 **Definición.** Sea una función f, de dominio D y codominio D'. Se define la función $g : D_\perp \to D'_{\perp'}$ como extensión estricta de f respecto a los elementos $\perp$ y $\perp'$, como:

$$g(\perp) = \perp', \forall x \in D,\ g(x) = f(x)$$

B.25 **Proposición.** La extensión estricta de una función es continua.

Ya estamos en disposición de limitar el tipo de funciones a considerar cuando tratamos de dar semántica a una λ-expresión.

B.26 **Definición.** Sea el retículo completo $(S, \sqsubset, \perp)$. Se define $S \to S$ como el conjunto de funciones totales, monótonas y continuas que proyectan elementos de S en S, parcialmente ordenado por el orden inducido por $\sqsubset$.

B.27 **TEOREMA.** $S \to S$ es un retículo completo y toda función $f.S \to S$ tiene un punto fijo.

A continuación definimos el operador que nos permite encontrar dicho punto fijo.

B.28 **DEFINICIÓN.** Se define la función *fix* : $(S \to S) \to S$ como:

$$\forall f.S \to S,\ fix(f) = \sqcup_{n=0}^{\infty} f^n(\perp)$$

B.29 **TEOREMA.** Para toda f : $S \to S$, *fix* proyecta f sobre su punto fijo mínimo.

EJEMPLO B.2

Considérese nuevamente la λ-expresión:

$$F = \lambda f.\ \lambda x.\ \textit{if } x = 0 \textit{ then } 1 \textit{ else if } x = 1 \textit{ then } f(3) \textit{ else } f(x\text{-}2)$$

Calculemos su punto fijo según la definición B.28. Para ello, partiendo de la función completamente indefinida $\lambda x.\perp$, construiremos una cadena de funciones cada vez mejor definidas cuyo supremo es el punto fijo buscado.

- $F^0 = \lambda x.\perp \equiv$
  $\equiv \{ (0,\perp), (1,\perp), (2,\perp), (3,\perp), (4,\perp), (5,\perp), \ldots \}$
- $F^1 = F(F^0) =$
  $= (\lambda f.\ \lambda x.\ \textit{if } x{=}0 \textit{ then } 1 \textit{ else if } x{=}1 \textit{ then } f(3) \textit{ else } f(x\text{-}2))\ (\lambda x.\perp) =$
  $= \lambda x.\ \textit{if } x{=}0 \textit{ then } 1 \textit{ else if } x{=}1 \textit{ then } (\lambda x.\perp)(3) \textit{ else } (\lambda x.\perp)(x\text{-}2) =$
  $= \lambda x.\ \textit{if } x{=}0 \textit{ then } 1 \textit{ else if } x{=}1 \textit{ then } \perp \textit{ else } \perp =$
  $= \lambda x.\ \textit{if } x{=}0 \textit{ then } 1 \textit{ else } \perp \equiv$
  $\equiv \{ (0,1), (1,\perp), (2,\perp), (3,\perp), (4,\perp), (5,\perp), \ldots \}$
- $F^2 = F(F(F^0)) = F(F^1) =$
  $= ((\lambda f.\ \lambda x.\ \textit{if } x{=}0 \textit{ then } 1 \textit{ else if } \ldots))\ (\lambda x.\ \textit{if } x{=}0 \textit{ then } 1 \textit{ else } \perp) =$
  $= \ldots =$
  $= \lambda x.\ \textit{if } x{=}0 \textit{ then } 1 \textit{ else if } x{=}2 \textit{ then } 1 \textit{ else } \perp \equiv$
  $\equiv \{ (0,1), (1,\perp), (2,1), (3,\perp), (4,\perp), (5,\perp), \ldots \}$
- $F^3 = F(F(F(F^0))) = F(F^2)$
  $= \ldots =$

$= \lambda x.$ *if* $x=0$ *then* 1 *else if* $x=2$ *then* 1 *else if* $x=4$ *then* 1 *else* $\perp \equiv$

$\equiv \{ (0,1), (1,\perp), (2,1), (3,\perp), (4,1), (5,\perp), \ldots \}$

Como se puede observar cada una de las funciones F^i está más definida que sus predecesoras. Todas ellas forman una cadena infinita de funciones $\{ F^0, F^1, F^2, F^3, \ldots \}$ cuyo supremo es efectivamente la función que deseamos que la expresión denote: aquella que si el argumento es par, vale 1, y si es impar, vale indefinido.

Apéndice C: Especificaciones VHDL†

En este apéndice se ofrecen las especificaciones conductuales utilizando el subconjunto de VHDL aceptado por el *Behavioral Compiler* de Synopsys [Syn95a], que se corresponden con algunas de las especificaciones ecuacionales presentadas en el ejemplo 2.20.

La elección de este subconjunto, y no otro, no ha sido arbitraria. La primera razón es que esta herramienta de SAN es, posiblemente, la que tiene un uso más extendido dentro de la comunidad de diseñadores. La segunda razón queda clara conociendo cuál ha sido la evolución que ha seguido el subconjunto de VHDL para síntesis lógica-RT.

En un comienzo (y aún ahora) cada herramienta de síntesis lógica poseía su propio subconjunto de VHDL. Esto motivó al nacimiento, en el seno de los grupos de usuarios del lenguaje, de una corriente estandarizadora cuyo objetivo era encontrar el subconjunto mínimo de VHDL que debía ser aceptado por toda herramienta. Sin embargo, en paralelo con dichos esfuerzos, el subconjunto adoptado por Synopsys (que es y era también la herramienta más popular en ese ámbito) se convirtió en el estándar de facto para muchas otras herramientas que nacieron a su alrededor. Finalmente cuando los esfuerzos estandarizadores comenzaron a obtener conclusiones,

† Según las normas de estilo establecidas por el Behavioral Compiler de Synopsys.

terminaron coincidiendo en buena parte las construcciones que en un principio ya adoptara esta herramienta.

Por ello, según esta experiencia, no resulta aventurado decir que posiblemente el subconjunto de VHDL para síntesis conductual siga el mismo camino. Por ello este subconjunto se ha utilizado como piedra de toque para comparar su expresividad con el mecanismo de especificación ecuacional.

De todas la especificaciones ecuacionales presentadas en el capítulo 2 se reproducen las que se corresponden con los cuerpos A y G. Cada uno de ellos especificaba el mismo algoritmo de Euclides para el cálculo del máximo común divisor, pero con distinto interfaz y protocolo de comunicación con el exterior.

C.1 Especificación del *cuerpo A*.

El cuerpo A especificaba un circuito con dos puertos de entrada de datos, *ain* y *bin*, por los que cuando cierto puerto *start* valiera uno, debían leerse simultáneamente los argumentos del algoritmo. Además, por un puerto de salida *done*, el circuito debería indicar cuándo había un resultado válido en el puerto de salida de datos *outp*. Además si en mitad de un cálculo *start* se activaba, deberían abandonarse los cálculos y volver a leer los puertos de entrada. El cuerpo del código VHDL (sin declaración de entidad ni de datos ni de bibliotecas) podría ser como sigue.

cuerpo A

```
process
begin
   bucle_reset : loop
      done <= '1';
      outp <= 0;
      bucle_principal : loop
         bucle_start : loop
            wait until clk'event and clk='1';
```

```
            if ( reset = '1' ) then exit bucle_reset; end if;
            if ( start = '1' ) then
               a := ain;
               b := bin;
               done <= '0';
               wait until clk'event and clk='1';
               if ( reset = '1' ) then exit bucle_reset; end if;
               exit bucle_start;
         end loop bucle_start;
         bucle_calculo : while ( a /= b ) loop
            if ( a > b ) then
               a := a - b;
            else
               b := b - a;
            end if;
            wait until clk'event and clk='1';
            if ( reset = '1' ) then exit bucle_reset; end if;
            if ( start = '1' ) then
               a := ain;
               b := bin;
               done <= '0';
               wait until clk'event and clk='1';
               if ( reset = '1' ) then exit bucle_reset; end if;
               next bucle_calculo;
         end loop bucle_calculo;
         done <= '1';
      end loop bucle_principal;
   end loop bucle_reset;
end process;
```

C.2 Especificación del *cuerpo G*.

El cuerpo G especificaba un circuito con un único puerto multiplexado *inp* de manera que por él entraban en ciclos consecutivos los dos argumentos que el algoritmo necesita. Ahora el puerto de *start* deberá permanecer activo mientras que se leen ambos argumentos.

```
process
begin
   bucle_reset : loop
      done <= '1';
      outp <= 0;
      bucle_principal : loop
         bucle_start1 : loop
            wait until clk'event and clk='1';
            if ( reset = '1' ) then exit bucle_reset; end if;
            if ( start = '1' ) then
               a := inp;
               done <= '0';
               bucle_start2 : loop
                  wait until clk'event and clk='1';
                  if ( reset = '1' ) then exit bucle_reset; end if;
                  if ( start = '1' ) then
                     a := inp;
                  else
                     b := inp
                     wait until clk'event and clk='1';
                     if ( reset = '1' ) then exit bucle_reset; end if;
                     exit start1;
                  end if;
               end loop bucle_start2;
            end if;
         end loop bucle_start2;
         bucle_calculo : while ( a /= b ) loop
            if ( a > b ) then
               a := a - b;
            else
               b := b - a;
            end if;
            outp <= a;
            wait until clk'event and clk='1';
            if ( reset = '1' ) then exit bucle_reset; end if;
            if ( start = '1' ) then
               a := inp;
               done <= false;
               bucle_start3 : loop
                  wait until clk'event and clk='1';
                  if ( reset = '1' ) then exit bucle_reset; end if;
                  if ( start = '1' ) then
                     a := inp;
                  else
                     b := inp
```

```
                    wait until clk'event and clk='1';
                    if ( reset = '1' ) then exit bucle_reset; end if;
                    exit start3;
                  end if;
                end loop bucle_start3;
                next bucle_calculo;
              end if;
            end loop bucle_calculo;
            done <= '1';
          end loop bucle_principal;
        end loop bucle_reset;
      end process;
```

Apéndice D: Un simulador†

Como ilustración de la elegancia, validez y potencia del enfoque propuesto, en este apéndice se propone un simulador de especificaciones ecuacionales, en las que es posible obtener una simulación de la misma en cualquier estadio intermedio de la síntesis. Está implementado en GOFER y se desarrolló (prácticamente en 10 minutos) a partir de la semántica del formalismo, que como puede observarse, sólo hay que adaptarla a la sintaxis del lenguaje. Además se incluye un ejemplo en donde se muestra cómo puede expresarse la especificación ecuacional del filtro de segundo orden estimulado por una entrada nivel 1.0, que como también se observa, sólo hay que adaptarla por la condición prefija y asociativa por la izquierda de los operadores GOFER.

Para ver la traza, de longitud *n*, de cualquier señal, *s*, bastará con teclear, una vez cargado el programa, la expresión *map s* [1..*n*].

```
{- Definicion del indice temporal -}
type N = Int

{- Extension temporal de constantes -}
cons :: a -> ( N -> a )
cons x t = x
```

† 'Rápido' y 'sucio' usando Gofer.

```
{- Extension temporal de operadores de dos argumentos -}
lu :: ( a -> b -> c ) -> ( N -> a ) -> ( N -> b ) -> ( N -> c )
lu op left right t = op ( left t ) ( right t )

{- Definicion de operadores temporales -}
fby :: a -> ( N -> a ) -> ( N -> a )
   fby c s 1 = c
   fby c s t = s (t-1)
next :: ( N -> a ) -> ( N -> a )
   next s t = s (t+1)
replicate :: N -> ( N -> a ) -> ( N -> a )
   replicate n s t = s (ceil t n)
      where ceil l r = if (mod l r) == 0 then (div l r) else (div l r)+1
sample :: N -> ( N -> a ) -> ( N -> a )
   sample n s t = s (t*n)
interleave :: [ ( N -> a ) ] -> ( N -> a )
   interleave sx t = ( sx !! (mod (t-1) (length sx)) ) t


{- Estimulo de entrada -}
   input = cons 1.0

{- Second Order -}
   {- definicion de valores concretos de constantes -}
      a1 = cons 0.625
      a2 = cons 1.0
      b1 = cons 0.5
      b2 = cons 0.375
   {- definicion del circuito -}
      output = lu (-) z ( lu (+) ( lu (*) a1 a ) ( lu (*) a2 b ) )
      z = lu (+) input ( lu (+) ( lu (*) b1 a ) ( lu (*) b2 b ) )
      a = fby 0.0 z
      b = fby 0.0 a
{- Fin de Second Order -}
```

Bibliografía

[Alli86] L. Allison. *A practical introduction to denotational semantics.* Cambridge Computer Science Texts, nº 23. Cambridge University Press, 1986.

[BDP+79] M. Broy, W. Dosch, H. Partsch, P. Pepper, M. Wirsing. *Existential quantifiers in abstract data types*, en H.A. Maurer (ed.), *6th International Colloquium in Automata*, Languajes and Programming. Lecture Notes in Computer Science, nº 71, pp 75-87, Springer-Verlag, 1979

[BlEi97] C. Blumenröhr, D. Eisenbiegler. *An efficient representacion for formal synthesis.* Proc. International Symposium on System Synthesis, ISSS'97. 1997.

[BoJo89] C.D: Boyer, S.D Jonhson. *Usign the digital design derivation system: case study of a VLSI garbage collector*, en J. Darringer and F. Ramming (ed.), *Proc. International Symposium on Computer Hardware Description Languages*, CHDL'89. Elvesier, 1989.

[BoJo93] B. Bose and S.D. Johnson. *DDD-FM9001: derivation of a verified microprocessor.* Proc. Correct Hardware Design and Verification Methods, CHARME'93, 1993.

[Bout87] R.T. Boute. *Elements for the formal description of systems.* A. Kündig, R.E Bührer, J. Dähler (Eds.), *Embedded systems*, pp.

63-90. Lecture Notes in Computer Science, nº 284. Springer-Verlag, 1987.

[Broy94] M. Broy. *(Inter-)Action Refinement, the easy way*. J. Rozenblit, K. Buchenrieder (Eds.), *Codesign, computer-aided software/ hardware engineering*. IEEE Press, 1994.

[BrWi84] M. Broy and M. Wirsing. *A sistematic study of models of abstract data types*. Theoretical Computer Science, nº 33, pp. 139-174, 1984

[CaST91] R. Camposano, L.F. Saunders and R.M. Tabet. *VHDL as input for high-level synthesis*. Design and Test of Computers. pp. 43-55, 1991.

[CCIT88] CCITT. *Adaptive differential pulse code modulation (ADPCM)*, Rec. G.721, Fascicle III.4, 1988.

[ChLS93] L.F. Chao, A. LaPaugh, Sha. *Rotation scheduling: a loop pipelining algorithm*. Proc. 30th Design Automation Conference DAC'93. pp. 566-572. 1993.

[Chur51] A. Church. *The calculi of λ-conversion*. Annals of mathematical Studies 6. Princeton UP, 1951.

[Conw63] M.E. Conway. *Design of a separable transition-diagram compiler*. Comm. ACM 6. pp. 396-408. 1963.

[CPM+95] A.P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, R. Brodersen. *Optimizing power using transformations*. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, nº 1, pp. 12-31. January 1995.

[CuFe68] H.B. Curry, R. Feys. *Combinatory logic, vol. I*. Noth-Holland, 1968.

[DeBr95] C. Delgado, P.T. Breuer (ed.). *Formal semantics for VHDL*. Kluwer Academic Publishers, 1995.

[Delg87] C. Delgado. *Semantics of digital circuits*. Lecture Notes in Computer Science, nº 285. Springer-Verlag, 1987.

[DeOd93] A. Debreil and P. Oddo. *Synchronous designs in VHDL*. Proc. European Design Automation Conference, Euro-DAC'93. pp. 486-491, 1993.

[DRSC86] H. De Man, J. Rabaey, P. Six, L. Claesen. *Cathedral-II: A silicon compiler for digital signal processing.* IEEE Design and Test, Dec. 1986.

[EhMa85] H. Ehrig, B. Mahr. *Fundamentals of algebraic specification 1: equations and initial semantics*. EATCS Monographs on theoretical computer science nº 6, Springer-Verlag, 1985.

[EhMa90] H. Ehrig, B. Mahr. *Fundamentals of algebraic specification 2: module specifications and constraints*. EATCS Monographs on theoretical computer science nº 21, Springer-Verlag, 1990.

[EiBK96] D. Eisenbiegler, C. Blumenröhr, R. Kumar. *Implementation issues about the embedding of existing High Level Synthesis algorithms in HOL*. Proc. International Conference on Theorem Proving in Higher Order Logics, TPHOL'96. 1996.

[EiKB97] D. Eisenbiegler, R. Kumar, C. Blumenröhr. *A constructive approach towards correctness of synthesis - Application within retiming*. Proc. European Design and Test Conference, ED&TC'97. IEEE Press, 1997.

[EiK95a] D. Eisenbiegler, R. Kumar. *An automata theory dedicated towards formal circuit synthesis*, en E.T. Shubert, P.j: Windley, J. Alver-Foss (ed), *Proc. International workshop on Higher Order Logic theorem proving and its applications, HUG'95*. Lecture Notes in Computer Science nº 971, pp. 154-169. Springer-Verlag, 1995.

[EiK95b] D. Eisenbiegler, R. Kumar. *Formally embedding existing High Level Synthesis algorithms*. Proc. Correct Hardware Design and Verification Methods, CHARME'95. Lecture Notes in Computer Science nº 987, pp. 71-83. Springer-Verlag, 1995.

[EKMP82] H. Ehrig, J.J. Kreowski, B. Mhr and P. Padawitz. *Algebraic implementation of abstract data types.* Theoretical computer science, nº 20, pp. 209-263, North-Holland, 1982.

[Faus82] A. Faustini. *The equivalence of a denotational and an operational semantics for pure data flow.* Ph.D. dissertation, University of Warwick, Comp. Sci. Dept. Coventry. United Kingdom.

[FiFM91] S. Finn, M.P. Fourman, G. Musgrave. *Interactive synthesis in higher order logic.* Proc. International Workshop on the HOL theorem prover and its applications. IEEE Press. 1991.

[FNS+94] F. Franssen, L. Nachtergaele, H. Samsom, F. Catthoor, H. de Man. *Flow optimization for fast system simulation and storage minimisation.* Proc. European Design and Test Conference, ED&TC'94, pp. 20-24, IEEE Press, 1994.

[FoMa89] M.P. Fourman, E.M. Mayger. *Formally based systems design - interactive hardware scheduling.* Proc. VLSI'89. 1989.

[GaGS88] S.J. Garland, J.V. Guttag, J.A. Staunstrup. *Verification of VLSI circuits using LP*, en *The fussion of hardware design and verification*, pp. 329-345, IFIG WG 10.2, North Holland, 1988.

[GeCD93] W. Geurts, F. Catthoor, H. De Man, *Heuristic techniques for the synthesis of complex functional units*, Proc. European Design Automation Conference, EDAC'93, pp. 552-556, 1993.

[GoCK98] J. Gong, C.T. Chen, K. Küçükçakar. *Architectural rule checking for high-level synthesis.* Proc. Design, Automation and test in Europe, DATE'98. 1998.

[GoME93] M. Gordon, T. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GoTW78] J.A. Goguen, J.W. Thatcher, E.G. Wagner. *An initial algera approach to the specification, correctness and implementation of*

abstract data types, en J. Yeh (ed.), *Current Trends in Programming Methodology IV: Data Structuring*, pp. 80-144, Prentice-Hall 1978.

[GuHo86] J.V. Gutag and J.J Horning. *A Larch Shared language handbook. Science Computer Programming*, 6:2, pp. 103-157, March 1986.

[GuHo93] J.V. Guttag and J.J. Horning (Eds.). *Larch: languages and tools for formal specification.* EATCS Monographs on theoretical computer science, Springer-Verlag, 1993.

[Gupt92] A. Gupta. *Formal hardware verification methods: a survey.* Formal Methods in System Design. vol. 1, pp. 151-238, Kluwer Academic Publishers, 1992.

[HaKR98] C. Hansen, A. Kunzmann, W. Rosenstiel. Verification by simulation comparison using interface synthesis. Proc. Design Automation and Test in Europe, DATE'98, pp. 436-433, IEEE Press. 1998.

[Hare87] D. Harel. *StateCharts: a visual formalism for complex systems. Science of computer programming*, nº 8. 1987.

[HeHe96] J. Hennessy, M. Heinrich. *Hardware/software codesign of processors: concepts and examples*, en G. de Micheli, M. Sami (ed.), *Hardware/software codesign.* NATO ASI series E: Applied Sciences, vol. 310, pp. 29-44, Kluwer Academic Publishers. 1996.

[Hilf85] P.N. Hilfinger. *A high level language and silicon compiler for digital signal processing.* Proc. of the IEEE Custom Integrated Circuits Conference, pp. 213-216, 1985.

[HoDo82] C.M. Hoffmann and M.J. O'Donnell. *Programming with equations.* ACM Transactions on Programming Languages and Systems, vol. 4, nº 1, pp. 83-112, 1982.

[HoKr93] B. Hoffmann, B. Krieg-Brückner (ed.). *Program development by*

specification and transformation: the PROSPECTRA methodology, language family and system. Lecture Notes in Computer Science, nº 680. Springer-Verlag, 1993.

[HuKr94] C. Huijs, T. Krol. *A formal semantic model to fit SIL for transformational design*. Proc. Euromicro'94, pp. 100-108, 1994.

[IEEE87] *IEEE standard VHDL languaje reference manual*. IEEE Std 1076-1987, IEEE Inc. 1987.

[IMEC92] *The Cathedral-II silicon compiler for real time signal processing*. Report interno. IMEC. 1992.

[JaCM94] M. Janssen, F. Catthoor, H. de Man. *A specification invariant technique for operation cost minimisation in flow-graphs*. Proc. International Symposium on High-Level Synthesis, pp. 146-151. 1994.

[JoBB88] S.D. Johnson, B. Bose, C.D. Boyer. *A tactical framework for digital design*, en G. Birtwistle, P. Subrahmanyam (ed.), *VLSI specification, verification and synthesis*, pp. 349-383. Kluwer Academic Publishers, 1988.

[JoBo91] S.D. Johnson, B. Bose. *DDD - A system for mechanized digital design derivation*. Proc. ACM/SIGDA International Workshop on Fomal Methods in VLSI Design, 1991. (Nunca se llegó a publicar oficialmente).

[John84] S.D. Johnson. *Synthesis of digital designs from recursion equations*. The MIT Press, Cambridge, 1984.

[John89] S.D Johnson. Manipulating logical organization with system factorizations, en Leeser and Brown (ed.), Hardware specification, verification and synthesis: mathematical aspects. Lecture Notes in Computer Science, nº 408, pp. 260-281. Springer-Verlag, 1989.

[John97] S.D. Johnson. *A tabular language for system design*. Proc. NASA LaRC formal methods workshop, LFM'97, 1997.

[JoSh90] G. Jones and M. Sheeran. *Circuit design in Ruby*, en J. Staunstrup (ed.), *Formal Methods for VLSI Design*, pp. 13-70. Elvesier, 1990.

[JoS91a] G. Jones and M.Sheeran, *Deriving bit-serial circuits in Ruby.* Proc. VLSI'91, 1991.

[JoS91b] G. Jones and M.Sheeran, *Collecting butterflies.* Technical monograph, Oxford University Computing Laboratory, Oxford University, 1991.

[Kahn74] G. Kahn. *The semantics of a simple languaje for parallel processing.* Proc. IFIP Congress 74. pp. 471-475. Elvesier North-Holland.

[KBES96] R. Kumar, C. Blumenröhr, D. Eisenbiegler, D. Schmid. *Formal synthesis in circuit design - a classification and survey.* Proc. Formal Methods in Computer-Aided design, FMCAD'96. 1996.

[KMN+92] T. Krol, J.V. Meerbergen, C. Niessen, W. Smits, J. Huisken. *The Sprite input language. An intermediate format for High Level Synthesis.* Proc. European Design Automation Conference, EDAC'92, pp. 186-192, 1992.

[KuMi90] D. Ku, G. De Micheli. *HardwareC - a languaje for hardware design (version 2.0).* Stanford University CLS Technical Report, CSL-TR-90-419. 1990.

[Kurs97] R.P. Kurshan. *Formal verification in a commercial setting.* Proc. Design Automation Conference, DAC'97, pp. 258-262. IEEE Press. 1997.

[KWCM98] H. Keding, M. Willems, M. Coors, H. Meyr. *FRIDGE: a fixed point design and simulation environment.* Proc. Design, Automation and test in Europe, DATE'98. 1998.

[Land66] P.J. Landin. *The next 700 programming languajes.* Comm. ACM 9. pp. 157-166. 1966.

[LCG+90] D. Lanneer, F. Catthoor, G. Goossens, M. Pauwels, J. Van

Meerbergen, H. De Man. *Open-ended system for high-level synthesis of flexible signal processors*, Proc. European Design Automation Conference, EDAC'90, pp. 272-276, 1990.

[Leis83] C. Leiserson, J. Saxe. *Optimizing synchoronous systems.* Journal of VLSI and Computer Systems, vol. 1, issue 1, pp. 41-67, 1983.

[LiGu98] J. Li, R.K. Gupta. *An algorithm to determine mutually exclusive operations in behavioral descriptions.* Proc. Design, Automation and test in Europe, DATE'98, pp. 457-463. 1998.

[LiZi74] B. Liskov, S.N. Zilles. *Programming with abstract data types.* SIGPLAN Notices 9, nº 50-59, 1974.

[Luk93] W. Luk, *Systematic serialisation of array-based architectures.* Integration, The VLSI Journal, nº 3, pp. 333-360, Feb. 93.

[MaFo91] E.M. Mayger, M.P. Fourman. *Integration of formal methods with system design.* Proc. VLSI'91. 1991.

[MaMT92] F. Maciel, Y. Miyanaga, K. Tochinai. *Optimizing and scheduling DSP programs for high throughput VLSI designs.* IEICE Transactions - Japan, vol. E75-A, no. 10, pp. 1191-1201, October 1992.

[McFa93] M.C. McFarland. *Formal verification of sequential hardware: a tutorial.* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, nº 5, pp. 633-654. May 1993.

[Mees88] D. Messerschmitt. *Breaking the recursive bottleneck*, in Perfomance Limits in Communication Theory and Practice, pp. 3-19. Kluwer Publishers, 1988.

[MeHe97] J.M. Mendías, R. Hermida. *Usign equational specification for behavioral synthesis.* Proc. Conference Design of Integrated Circuits and Systems, DCIS'97. 1997.

[MeHe98] J.M. Mendías, R. Hermida. *Automatic formal derivation applied to high-level synthesis*, en J.C: López, R. Hermida, W.

Geisselhard (ed.), *Advances techniques for enbedded systems design and test*. Kluwer Academic Publishers, 1998.

[MeHF97] J.M. Mendías, R. Hermida, M. Fernández. *Formal Techniques for hardware allocation*. Proc. International conference on VLSI design, VLSI'97. 1997.

[MeHF98] J.M. Mendías, R. Hermida, M. Fernández. *Correct High-Level Synthesis: a formal perspective*. Proc. Design, Automation and test in Europe, DATE'98. 1998.

[Meye89] E. Meyer. *VHDL strives to cover both synthesis and modeling*. Computer Design, nº 10, 1989.

[MHF96a] J.M. Mendías, R. Hermida, M. Fernández. *Algebraic support for transformational hardware allocation*. Proc. European Design & Test Conference, ED&TC'96, 1996.

[MHF96b] J.M. Mendías, R.Hermida, M. Fernández. *A formal approach to improve hardware reusability in high level synthesis tools*. Proc. Euromicro Conference.1996.

[MHM+94] J.M. Mendías, R. Hermida, R. Moreno, M. Fernández, P. Rupérez. *Especificaciones de alto nivel usando VHDL*. Actas del IX Congreso Diseño de Circuitos Integrados, DCIS'94, pp. 18-23. 1994.

[MiLD92] P. Michel, U. Lauther and P. Duzy (Eds). *The synthesis approach to digital system design*. Kluwer Academic Publishers, 1992.

[MiRa96] P.F.A. Middelhoek, S.P. Rajan. *From VHDL to efficient and first-time-right designs: a formal approach*. ACM Trans. on Designs Automation of Electronic Systems. Vol. 1, nº 2, April 1996.

[NaVG91] S. Narayan, F. Vahid, D.D. Gajski. *System specification and synthesis with the SpecChart languaje*. Proc. International Conference on Computer Aided Design. 1991.

[NCGD92] S. Note, F. Catthoor, G. Goossens, H. De Man. *Combined*

hardware selection and pipelining in high-performance data-path design, IEEE Trans. on Computer-Aided Design, vol. 11, pp. 413-423, Apr. 1992.

[NGCD91] S. Note, W. Geurts, F. Catthoor, H. De Man. *Cathedral-III: architecture-driven high-level synthesis for high throughput DSP applications.* Proc. Design Automation Conference, pp. 597-602, 1991.

[Pada88] P. Padawitz. *Computing in Horn Clause Theories.* EATCS Monographs on theoretical computer science nº 16, Springer-Verlag, 1988.

[PaKG86] P.G. Paulin, J.P. Knight, E.F. Girczyc. *HAL: a multi-paradigm approach to automatic data path synthesis.* Proc. Design Automation Conference, DAC'86, pp. 263-270. 1986.

[PaMa93] M. Padmanabhan, K. Martin. *Feedback-based orthogonal digital filters.* IEEE Trans. on Circuits and Sysiitems II, vol. 40, pp. 512-521, Aug. 1993.

[PaMe89] K. Parhi, D. Messerschmitt. *Pipelining and parallelism in recursive digital filters: I and II.* IEEE Transactions on Signal Processing, pp. 1118-1135. July 1989.

[Paul94] L.C. Paulson. *Isabelle: a generic theorem prover.* Lecture Notes in Computer Science, nº 828. Springer-Verlag, 1994.

[PeKu94] Z. Peng, K. Kuchcinski. *Automated transformation of algorithms into register-transfer level implementations.* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, nº 2, pp. 150-166. Feb. 1994.

[Peña93] R. Peña. *Diseño de programas. Formalismo y abstracción.* Prentice-Hall. 1993.

[PoDR95] M. Potkonjak, S. Dey, R.K. Roy. *Considering testability at behavioral level: use of transformations for partial scan cost minimization under timing and area constraints.* IEEE Trans. on

Computer-Aided Design of Integrated Circuits and Systems, vol. 14, nº 5, pp. 531-546. May 1995.

[Poig86] A. Poigné. *On specification, theories and models with higher types. Information and Control*, nº 68, pp. 1-46, 1986.

[PoRa94] M. Potkonjak, J. Rabaey. *Optimizing resource utilization using transformations.* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, nº 3, pp. 277-292. March 1994.

[Rasm96] O. Rasmussen. *A Ruby proof system.* Technical Report. Dept. of Computer Science, Technical University of Denmark, 1996.

[RaTJ93] K. Rath, M.E. Tuna, S.D. Johnson. *Behavior Tables: a basis for system representation and transformational system synthesis.* Proc. International Conference on Computer-Aided Design, ICCAD'93, pp. 736-740. IEEE Press, 1993.

[ReCl86] J. Rees and W. Clinger. The revisited report on the algorithmic language Schema. ACM SIGPLAN Notices, 21(12), pp. 37-79, 1986.

[RoBu95] J. Rozenblit, K. Buchenrieder (Eds). *Codesign, computer-aided software/hardware engineering.* IEEE Press, 1995.

[Rose73] B.K. Rosen. *Tree-manipulating systems and Church-Rosser theorems.* Journal of the ACM, nº 20:1. 1973.

[Ros90a] L. Rossen. *Ruby algebra*, en G. Jones and M. Sheeran (ed.), *Designing Correct Circuits*, pp. 297-312, Springer Verlag, 1990.

[Ros90b] L. Rossen. *Formal Ruby algebra*, en J. Staunstrup (ed.), *Formal methods for VLSI design*, pp. 179-190. Elvesier Science Publishers, 1990.

[SaCM93] H. Samsom, L. Claesen, H. de Man. *SynGuide: an environment for doing interactive correctness preserving transformations*, en L.D.J. Eggermont, P. Dewilde, E. Deprettere, J. van Meerbergen (ed.) *Proc. VLSI signal processing VI*, pp. 269-277. IEEE

Press,1993.

[SaMS96] A.L Sangiovanni, P.C. McGeer, A. Saldanha. *Verification of electronic systems.* Proc. Design Automation Conference, DAC'96, pp. 106-111. IEEE Press. 1996.

[Sand94] O. Sandum. *Multiple clocks and Ruby.* Technical Report. Dept. of Computer Science, Technical University of Denmark, 1994.

[Schö24] M. Schönfinkel. *Uber die bausteine der mathematischen logik.* Mathematische Annalen 92, pp. 305-316, 1924.

[Scot71] D.S. Scott. *Continuous lattices.* PROG-7. Oxford University Programming Research Group, 1971.

[Scot76] D.S. Scott. *Data types as lattices.* The SIAM Journal on Computing 5, nº 3, pp. 522-587, Sept. 1976.

[Shee90] M. Sheeran. *Describing butterfly networks in Ruby*, en K. Davis and J. Hughes (ed.), *Functional Programming*, Springer Workshops in Computing, Springer-Verlag, 1990.

[ShPa93] N.R. Shanbhag, K.K. Parhi. *Relaxed look-ahead pipelined LMS adaptive filters and their application to ADPCM coder.* IEEE Trans. on Circuits and Systems II, vol. 40, pp. 753-765, Dec. 1993.

[ShR93a] R. Sharp and O. Rasmussen, *Transformational rewriting with Ruby*, en . D. Agnew, L. Claesen, R. Camposano (ed.), Proc. Computer Hardware Description Languages and their Applications, CHDL'93, pp. 243-260, Elvesier Science Publishers, 1993.

[ShR93b] R. Sharp and O. Rasmussen, *Rewriting with constraints in T-Ruby*, Proc. Correct Hardware Design and Verification Methods, CHARME'93, Lecture Notes in Computer Science, nº 663, pp. 226-241, Kluwer Academic Publishers, 1993.

[ShR95a] R. Sharp and O. Rasmussen, *The T-Ruby design system.* Proc. Computer Hardware Description Languages and their

Applications, CHDL'95, pp. 587-596, 1995.

[ShR95b] R. Sharp and O. Rasmussen, *An Introduction to Ruby*. Technical Report, Dept. of Computer Science, Technical University of Denmark, 1995.

[StGG92] J. Staunstrup, S.J. Garland, J.V. Guttag. *Mechanized verification of circuit descriptions using the Larch Prover*, en V. Stavridou, T.F. Melham, R.T. Boute (ed.), *Theorem provers in circuit design: theory, practice and experience*, pp. 277-299. IFIG TC10/WG 10.2, North Holland, 1992.

[Stoy81] J.E. Stoy. *Denotational semantics: The Scott-Strachey approach to programming languaje theory*. The MIT Press Series in Computer Science, nº 1. MIT Press, 1981.

[SyMa93] M.A. Syed, V.J. Mathews. *Lattice algorithms for recursive least squares adaptive second-order Volterra filtering*. IEEE Trans. on Circuits and Systems II, vol. 41, pp. 202-213, Mar. 1993.

[Syn95a] Synopsys Inc. *Behavioral compiler user's guide v3.3a*. Synopsys, 1995.

[Syn95b] Synopsys Inc. *DesignWare developer guide v3.3a*. Synopsys, 1995.

[SSSF97] B. Straube, J. Schönherr, I. Schreiber, E. Fordran. *An introduction to formal verification*. Notas del curso: Synthesis and test of digital systems from behavioural specification. Cursos de verano de la Universidad Complutense. 1997.

[TaWW82] J.W. Thatcher, E.G. Wagner and J.B. Wright. Data type specification: parameterization and the power of specification techniques. Trans. on Programming Languages and Systems, nº4, pp. 711-773, 1982.

[Veri91] *Verilog hardware description languaje reference manual v1.0*. Open Verilog International. 1991.

[ViES95] E. Villar, W. Ecker, M. Selz. *VHDL synthesis description*

portability: the need for Level-x synthesis subsets. Proc. VHDL-Forum Spring'95. 1995.

[WaAs85] W.W. Wadge and E.A. Ashcroft. *Lucid, the dataflow programming language.* APIC Studies in data processing nº 22, Academic Press, 1985.

[Wand81] M. Wand. *Final algebra semantics and data types extensions.* Journal of Computing and System Sciences, nº 19, pp. 27-44, 1981.

[WaPa95] C.Y. Wang, K.K. Parhi. *High-Level DSP Synthesis using concurrent transformations, scheduling and allocation.* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, nº 3, pp. 274-295. March 1995.

[WaTh89] R.A. Walker, D.E. Thomas. *Behavioral transformation for algorithmic level IC design.* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 8, nº 10, Oct. 1989.

[Watt93] D.A. Watt. *Programming language syntax and semantics.* C.A.R. Hoare series editor. Prentice Hall International. 1993.

[WBMN90] R. Woudsma, F. Beenker, J. van Meerbergen, C. Niessen. *Piramid: an architecture-driven silicon compiler for complex DSP applications.* Proc International Symposium on Circuits and Systems, pp. 2696-2700. 1990.

[ZhJo93] Z. Zhu, S.D. Johnson. *An example of interactive hardware transformation.* Proc. ACM/SIGDA International Workshop on Fomal Methods in VLSI Design, 1991. (Nunca se llegó a publicar oficialmente).