BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Universidad de Valladolid

Arquitectura de software dinámica basada en reflexión

Carlos Enrique Cuesta Quintero

Tesis de Doctorado

Facultad: E. T. S. Informática

Director: Dr. Pablo de la Fuente Redondo

2002

Universidad de Valladolid
Departamento de Informática (ATC, CCIA, LSI)

Arquitectura de Software Dinámica Basada en Reflexión

Tesis Doctoral

D. Carlos Enrique Cuesta Quintero

Director:

Dr. Pablo de la Fuente Redondo

Julio de 2002

A Pilar

BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Agradecimientos

Aunque es casi un tópico, no por ello es menos cierto que un trabajo de tesis implica siempre a muchísimas más personas que su autor. En mi caso, puedo decir que el apoyo y confianza de todas esas personas ha resultado determinante para que esta tesis llegase a buen término.

Son muchas las contribuciones que he recibido, de un modo u otro; es imposible recordar ahora todas ellas. Pero aun así, no quiero dejar de dar las gracias a algunas personas cuya aportación ha sido para mí especialmente importante:

Ante todo, a Pablo, por su apoyo incondicional, su infinita paciencia, su interés, su apertura de ideas, sus buenos consejos y su integridad y calidad como científico y como persona. También, por haberme dirigido hacia un tema que ha llegado a apasionarme, y por confiar en que sería capaz de llevarlo a buen término; ha sido para mí un verdadero honor trabajar a su lado.

A Manolo, que sin pretenderlo me ha impulsado siempre a hacer todo aquéllo que yo en un principio no creía posible, y que ha resuelto siempre mis dudas más profundas, a menudo simplemente haciendo la pregunta precisa o diciendo la palabra adecuada. También, por su sentido del humor, por su ánimo constante y por ser un magnífico compañero.

A mis padres y hermanas, en especial, por su apoyo a lo largo de todos estos años, y a mis dos familias, en general, por su paciencia, confianza y ánimo.

A mis amigos, todos ellos, y en especial a Alberto, Blas, Carlos, Chema, David, Diego, Julio, Juan Carlos, Óscar, Rosa y Santiago, que han seguido ahí, todo el tiempo, esperando a que yo terminase de una vez, siempre dispuestos a ofrecer su apoyo.

A todos los miembros de los grupos de investigación Arco y Grinbd, del Área de Lenguajes y Sistemas Informáticos, en especial, y del Departamento de Informática, en general, por todas las veces que me han echado una mano, en especial en estos últimos meses.

Y, por supuesto, a Pilar, que ha estado siempre a mi lado, dentro y fuera de esta tesis, en todas y cada una de las etapas y momentos de su desarrollo. Son demasiadas las cosas que tengo que agradecerle, tanto en el plano humano, como en el científico y académico. Es completamente cierto que no tengo palabras para darle las gracias; sin ella, simplemente, no lo hubiera conseguido.

A todos, de nuevo, gracias.

Índice General

Dedicatoria iii

Agradecimientos v

I Introducción y Objetivos 1

1 Introducción 3
- 1.1 Introducción . . . 3
- 1.2 Objetivos . . . 4
- 1.3 Estructura de la Tesis . . . 5

II Arquitectura de Software Dinámica 9

2 Arquitectura de Software 11
- 2.1 Introducción . . . 11
- 2.2 Definición de Arquitectura de Software . . . 11
- 2.3 Relevancia de la Arquitectura de Software . . . 13
- 2.4 Conceptos Fundamentales . . . 15
  - 2.4.1 Lenguajes de Descripción de Arquitectura . . . 16
  - 2.4.2 Componente . . . 17
  - 2.4.3 Conector . . . 18
  - 2.4.4 Puerto . . . 21
  - 2.4.5 Estilo Arquitectónico . . . 22
  - 2.4.6 Vista . . . 23
- 2.5 Conclusión . . . 25

3 Arquitectura de Software Dinámica 27
- 3.1 Introducción . . . 27
- 3.2 Arquitectura de Software Dinámica . . . 28
- 3.3 Tipos de Dinamismo . . . 29
- 3.4 Reconfiguración Dinámica . . . 31
  - 3.4.1 Tipos de Reconfiguración . . . 31

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

3.4.2 Operaciones Básicas de Reconfiguración . . . 32
3.5 Clasificación: por Origen . . . 33
3.6 Sistemas Clásicos . . . 34
3.6.1 Conic . . . 34
3.6.2 Durra . . . 35
3.7 Basados en Álgebras de Procesos . . . 37
3.7.1 Wright Dinámico . . . 37
3.7.2 Darwin . . . 39
3.7.3 Leda . . . 42
3.7.4 Derivados de Acme . . . 44
Armani . . . 45
Hot Acme . . . 45
3.7.5 AML . . . 46
3.8 Basados en Eventos . . . 47
3.8.1 Rapide . . . 48
3.8.2 C2 (C2sadel) . . . 50
3.9 Basados en Grafos . . . 53
3.9.1 Precedentes . . . 53
Gramáticas-Δ: Garp . . . 53
Teoría Sintáctica de Dean & Cordy . . . 54
Aproximaciones de Holt . . . 54
3.9.2 Gramáticas de Le Métayer . . . 54
3.9.3 Hipergrafos de Hirsch . . . 56
Modelo Inicial . . . 57
Modelo de Mosaicos . . . 58
3.9.4 Modelo Categórico de Wermelinger . . . 60
3.10 Modelos Químicos . . . 62
3.10.1 Precedentes . . . 62
3.10.2 Doble Cham de Wermelinger . . . 63
3.11 Coordinación Dirigida por Datos . . . 64
3.11.1 El Modelo Linda . . . 65
3.11.2 Los Sistemas PoliS y MobiS . . . 67
3.12 Coordinación Dirigida por Control . . . 68
3.12.1 El Modelo Iwim: Manifold . . . 68
3.13 Otros Lenguajes . . . 70
3.13.1 Lógica de Reescritura . . . 70
3.13.2 El Modelo de Actores . . . 71
3.14 Modelos de Dinamismo Arquitectónico . . . 72
3.15 Conclusiones . . . 75
BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

III Reflexión **77**

4 El Concepto de Reflexión **79**
4.1 Introducción . . . 79
4.1.1 Definición . . . 80
4.2 El Problema de la Autorreferencia . . . 81
4.2.1 Meta-Programación . . . 84
4.3 Conceptos Fundamentales . . . 86
4.3.1 Hipótesis de Reflexión . . . 86
4.3.2 Reificación . . . 87
4.3.3 Arquitectura de Meta-Nivel . . . 89
Protocolo de Meta-Objeto . . . 90
4.3.4 Meta-Espacio . . . 91
4.4 Clasificaciones de la Reflexión . . . 92
4.4.1 Por Efecto . . . 93
4.4.2 Por Tiempo . . . 94
4.5 Reflexión y Orientación al Objeto . . . 94
4.5.1 Modelos Reflexivos de Objetos . . . 95
4.6 Reflexión Aplicada: Envolventes y Aspectos . . . 97
4.7 Reflexión y Arquitectura de Software . . . 98
4.8 Conclusiones . . . 103

IV Arquitectura de Software Dinámica mediante Reflexión **105**

5 El Modelo Marmol **107**
5.1 Introducción . . . 107
5.2 Arquitectura de Software Dinámica con Reflexión . . . 108
5.3 Marmol: Modelo Informal . . . 110
5.4 Sobre el Concepto de Nombre . . . 112
5.5 Marmol: Modelo Formal . . . 114
5.5.1 Modelo Arquitectónico General ($\aleph_\Sigma$) . . . 115
Conceptos Básicos (Primitivos) . . . 116
Dimensión Vertical: Composición . . . 117
Dimensión Horizontal: Interacción . . . 124
Sistema de Tipos . . . 131
5.5.2 Modelo Marmol General ($\aleph_M$) . . . 147
5.5.3 Modelo Marmol Restringido ($\aleph_\Pi$) . . . 162
5.6 Conclusión . . . 164

6 El Lenguaje PiLar: Sintaxis **167**
6.1 Introducción . . . 167
6.2 Visión General del Lenguaje . . . 169
6.2.1 El Concepto de Componente . . . 170

6.3 Lenguaje Estructural . . . 172
6.4 Declaración de Componente . . . 172
6.4.1 Declaración de Interfaz . . . 174
6.4.2 Declaración de Instancia . . . 175
6.4.3 Declaración de Conexiones . . . 175
6.4.4 Reificaciones y Restricciones . . . 176
6.4.5 Estructuras Sintácticas Adicionales . . . 177
Alternativa . . . 177
Iteración . . . 177
Parametrización . . . 178
Recursión . . . 178
6.5 Reificación: Reflexión Arquitectónica . . . 179
6.5.1 Anotaciones de Meta-Nivel . . . 181
6.6 Lenguaje Dinámico . . . 183
6.6.1 ¿Por qué no cálculo-π? . . . 185
6.7 Estructura de una Restricción . . . 186
6.7.1 Canales . . . 186
6.7.2 Variables . . . 187
6.7.3 Procesos . . . 187
6.8 Constructores de Proceso . . . 188
6.8.1 Regla de Ámbito . . . 190
6.8.2 Replicación . . . 190
6.9 Acciones Elementales . . . 191
6.10 Comandos Reflexivos . . . 193
6.10.1 Operandos Reflexivos . . . 194
6.10.2 Operadores Dinámicos . . . 196
6.10.3 Operadores Reflexivos . . . 197
6.11 Conectores de Nivel Meta . . . 199
6.12 Extensiones Específicas . . . 202
6.12.1 Soporte para Envolventes . . . 203
6.13 Conclusión . . . 206

7 El Lenguaje PiLar: Semántica **209**
7.1 Introducción . . . 209
7.2 Semántica Intuitiva . . . 210
7.2.1 Concepto Semántico Básico . . . 210
7.2.2 Discusión: sobre las Semánticas Algebraicas . . . 211
7.2.3 Componentes como Procesos Abstractos . . . 214
7.2.4 Conceptos Arquitectónicos de PiLar . . . 217
Puertos . . . 217
Componentes . . . 220
Conexiones . . . 222
7.2.5 Intuición para un Proceso Reflexivo . . . 223
7.3 Semántica de los Aspectos Estructurales . . . 236

7.3.1 Puerto . . . 236
7.3.2 Componentes (Instancias) . . . 239
7.3.3 Reificación . . . 244
7.3.4 Componentes en el Nivel Meta . . . 246
7.3.5 Conexión . . . 254
Conexión Jerárquica . . . 259
7.3.6 Estructuras Sintácticas Adicionales . . . 260
Conjuntos . . . 260
Estructura Alternativa . . . 261
Bucles . . . 262
Nota sobre Componentes Recursivos y Paramétricos . . . 263
7.4 Semántica de los Aspectos Dinámicos . . . 264
7.4.1 Sobre la Notación Semántica . . . 265
7.4.2 Restricciones . . . 265
Canales . . . 265
Variables . . . 266
7.4.3 Procesos: Constructores . . . 266
Replicación . . . 266
Recursión . . . 268
7.4.4 Primitivas de Comunicación . . . 268
7.4.5 Primitivas Dinámicas . . . 270
7.4.6 Primitivas Reflexivas . . . 273
Operandos Reflexivos . . . 273
Operadores Reflexivos . . . 275
7.5 Extensión: Envolventes . . . 279
7.5.1 Puerto . . . 279
7.6 Conclusión . . . 281

8 Arquitectura Dinámica en PiLar 283
8.1 Introducción . . . 283
8.2 Ejemplo de la Gasolinera . . . 284
8.2.1 Variante: una Gasolinera Dinámica . . . 286
8.3 Un Sistema Cliente/Servidor [2|3]-*tier* . . . 288
8.4 Ejemplo de Los Filósofos Evolutivos . . . 290
8.5 El Modelo Linda: Espacios de Tuplas . . . 296
8.6 Sistema con Componentes Envolventes . . . 299
8.7 Conclusiones . . . 303

V Conclusiones 305

9 Conclusiones y Trabajo Futuro 307
9.1 Conclusiones . . . 307
9.2 Líneas de Trabajo Futuro . . . 309

Bibliografía **314**

VI Apéndices **353**

A Gramática de PiLar **355**

B Sintaxis del Cálculo-π **363**
B.1 El Cálculo-π 366

C Dialecto del Cálculo-π **371**
C.1 Versión del Cálculo 371
C.2 Extensiones Gramaticales 374
C.2.1 Estructura Alternativa 375
C.2.2 Composición Secuencial 375
C.3 Tipos Básicos de Datos 376
C.3.1 Booleanos [Bool] 376
C.3.2 Enteros [Int] 377
C.3.3 Cadenas [String] 378
C.4 Tipos Abstractos de Datos 379
C.4.1 Tuplas y Vectores 379
C.4.2 Registros 380
C.4.3 Pares 382
C.4.4 Listas 383
Listas Nombradas 386
Notación Abreviada para Listas 389
C.5 Notación de Tipos 392

D Definiciones Semánticas Auxiliares **395**
D.1 Bucles: Iteradores 395
D.1.1 Sobre Listas 396
D.1.2 Sobre Listas Nombradas 397
D.1.3 Sobre Rangos de Enteros 397
D.2 Concatenación de Listas 398
D.3 Conversión de Enumeraciones 398
D.3.1 De Listas a Vectores 399
D.3.2 De Vectores a Listas 399

Índice de Tablas

4.1 Metaprogramación: Clasificación de Rideau (adaptada) 85

6.1 Constructores Básicos (Algebraicos) 188
6.2 Constructores Avanzados (Algorítmicos) 189
6.3 Acciones Elementales 191
6.4 Operaciones sobre Conjuntos 193
6.5 Operandos Reflexivos 194
6.6 Operadores Dinámicos 196
6.7 Operadores Reflexivos 197
6.8 Soporte para Envolventes 204

B.1 Gramática del Cálculo-π Poliádico 368

C.1 Operaciones de Listas: Definición Abreviada. 390
C.2 Operaciones de Listas Nombradas: Definición Abreviada. 391

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

Índice de Figuras

2.1 Hitos en Arquitectura de Software (1989–2002) 14
2.2 Taxonomía de Estilos 23
2.3 Evolución de Vistas en el Modelo 4+1 de Kruchten 24

3.1 El clásico Sistema de Monitorización de Pacientes 35
3.2 Estructura básica de un componente (tarea) en Durra 36
3.3 Fragmento de un *Configuror* de Wright Dinámico 38
3.4 Una Tubería Extensible en $\delta arwin$ 40
3.5 Establecimiento de un Enlace $\delta arwin$ en cálculo-π 42
3.6 Un Sistema Cliente-Servidor con Equilibrio de Carga en LEDA 43
3.7 Descripción del Estilo C2 como un *kind* de AML 47
3.8 El Problema de los Productores-Consumidores en Rapide 49
3.9 Una configuración sencilla en C2SADEL 51
3.10 Arquitectura Cliente-Servidor de Le Métayer 55
3.11 Arquitectura Cliente-Servidor como Multiconjunto 55
3.12 Definición del Coordinador para el Ejemplo Anterior 56
3.13 Una Regla de Producción de Grafos 57
3.14 Una Transformación en Notación-λ Doble 59
3.15 Aplicación de una Producción de Grafo sobre G 60
3.16 Un Conector Binario para Programas COMMUNITY 61
3.17 Un Conector Binario entre los Componentes P_1 y P_2 61
3.18 Reconfiguración por Superposición 61
3.19 CHAMs de Creación y de Evolución para el Estilo Clt/Srv/Mgr 64
3.20 Descripción de un Sistema Reconfigurable en Manifold 69
3.21 Modelos de Dinamismo en ADLs 72
3.22 Configurador 73
3.23 Espacio Común 73
3.24 Control Externo 74
3.25 Oráculo 74
3.26 Componente Mixto 75
3.27 Conector Mixto 75

4.1 Modelo de Reflexión en Dos Niveles 88
4.2 Tipos de Reflexión según el Efecto 93

4.3 Modelos de Reflexión Orientada al Objeto 95

5.1 Correspondencia entre Arquetipos e Instancias 145

6.1 Esquema de un Componente en $\mathcal{P}i\mathcal{L}ar$ 173
6.2 Filtro-Tubería: Iterativo 177
6.3 Filtro-Tubería: Recursivo 177
6.4 Componente Infinitamente Recursivo – Ilegítimo 178
6.5 Esquema de una Restricción en $\mathcal{P}i\mathcal{L}ar$ 186

7.1 Estructura Cuádruple de una Instancia Reificada 229
7.2 Reificaciones de un Meta-Componente 230
7.3 Reificaciones sobre un mismo Avatar 231
7.4 Reificación Múltiple para un Proceso con $\eth = 2$ 233
7.5 Estructura Interna de un Puerto $\mathcal{P}i\mathcal{L}ar$ 237
7.6 Estructura Interna de un Componente $\mathcal{P}i\mathcal{L}ar$ 240
7.7 Dualidad de un Componente Reflexivo 245
7.8 Estructura de un Meta-Componente 246
7.9 Dependencias entre Procesos sobre Componentes 248
7.10 Estructura Interna de una Conexión $\mathcal{P}i\mathcal{L}ar$ 255

8.1 Sistema de la Gasolinera 284
8.2 La Gasolinera, versión Dinámica 287
8.3 Sistema Cliente/Servidor (2|3)-*tier* 289
8.4 Definición Inicial y Etapas Dinámicas del Ejemplo 290
8.5 El Problema de los Filósofos Comiendo: Enfoque Estático 291
8.6 Los Filósofos Apresurados: Enfoque Recursivo 292
8.7 Los Filósofos Apresurados, versión Evolutiva 294
8.8 Los Filósofos Apresurados, versión Evolutiva (simplificada) 295
8.9 Modelo de Espacio de Tuplas: una descripción en $\mathcal{P}i\mathcal{L}ar$ 297
8.10 Representación Gráfica del Modelo de Espacio de Tuplas 298
8.11 Envolvente con Protocolo Explícito en $\mathcal{P}i\mathcal{L}ar$ 300
8.12 Envolvente con Protocolo Implícito en $\mathcal{P}i\mathcal{L}ar$ 302
8.13 Entradas Base Ligadas por Meta 303

A.1 Tabla de Equivalencias Léxicas en $\mathcal{P}i\mathcal{L}ar$ 361

B.1 El Cálculo-π y los Cálculos Nominales 365

Parte I

Introducción y Objetivos

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

Capítulo 1

Introducción

1.1 Introducción

BIBLIOTECA VIRTUAL

La complejidad y tamaño de los sistemas software se ha incrementado de manera espectacular en los últimos años. Su diseño y desarrollo ha pasado de una concepción monolítica y uniforme a una visión hetereogénea y distribuida. Esto ha puesto de manifiesto la relevancia de un estudio específico de la estructura del software, un aspecto que ha ido adquiriendo importancia de manera progresiva, y que ha dado lugar a la definición de su propio campo de estudio.

Se denomina *Arquitectura de Software* a la parte de la Ingeniería de Software que se dedica al estudio, análisis y descripción de esta estructura –que recibe, también, el nombre de *arquitectura de software*–. Se trata, en definitiva, de formalizar el esquema global de un sistema, poniendo el énfasis en el estudio de las interacciones entre sus elementos básicos, denominados *componentes*. Todo sistema, a cualquier nivel de granularidad, puede verse como una composición de tales elementos; por tanto, todo sistema tendrá una arquitectura.

Tradicionalmente, el estudio de la arquitectura de software se ha centrado en sus aspectos *estáticos*, esto es, en la descripción y análisis de estructuras inalterables. Sin embargo, existen sistemas, aún más complejos, en los que la estructura evoluciona a lo largo del tiempo, a menudo siguiendo patrones preestablecidos. Cuando estos patrones no son aleatorios, sino que están predefinidos, son una parte integral de la arquitectura, como lo son las propias relaciones estructurales. Se dice entonces que se tiene una *arquitectura dinámica*.

El estudio de arquitecturas dinámicas es un aspecto particularmente complejo, ya que la estructura suele ser, precisamente, el marco de referencia en el que se apoyan las descripciones de dinamismo. Esto ha hecho que se trate de un aspecto normalmente poco tratado; sin embargo, esto no ha de hacer pensar que su importancia es menor. Por el contrario, existe un buen número de sistemas –abiertos, distribuidos, adaptativos, evolutivos– que no pueden ser descritos de manera adecuada sin una especificación dinámica.

Las distintas propuestas para la descripción de Arquitectura de Software Dinámica utilizan aproximaciones muy diferentes al problema. Aunque hay algunos enfoques que muestran una versatilidad notable, la mayoría de ellos tienen una serie de limitaciones importantes, y en general sólo han conseguido alcanzar su objetivo de manera parcial.

En este trabajo se plantea si existe algún concepto unificador, que permita plantear la des-

cripción de dinamismo arquitectónico desde una perspectiva común. De este modo, se podrían reunir y recoger las aportaciones de las distintas propuestas existentes, integrándolas en un modelo uniforme. Es presumible que un modelo de este tipo superaría las limitaciones de los distintos enfoques particulares, dando una mejor solución al problema.

En esta tesis se sugiere que ese concepto existe, y es el de *Reflexión*. La Reflexión se define como la capacidad de un sistema (computacional) para *razonar y actuar sobre sí mismo*, proporcionando una forma controlada de auto-modificación, basada en sólidos fundamentos formales. Se ha aplicado en una gran variedad de ramas de la Informática, desde la Inteligencia Artificial a los Sistemas Operativos, pasando especialmente por los Lenguajes de Programación. Sin embargo, apenas se ha utilizado en la Arquitectura de Software, que por tanto carece de una perspectiva adecuada de esta noción.

Y sin embargo, una arquitectura dinámica es, por definición, reflexiva. En efecto: una arquitectura es una descripción de una estructura y aspectos de su comportamiento, mientras que una arquitectura dinámica es aquélla que cambia y evoluciona, e incluye una descripción de los patrones que dirigen este cambio. En definitiva, una parte de la estructura descrita está indicando el modo en que cambia esta misma estructura: es decir, una arquitectura dinámica se modifica a sí misma, y es por tanto *reflexiva*.

No obstante, el concepto de reflexión es más genérico, y manifiesta una estructura propia, que puede ser a su vez formalizada. Así pues, se plantea que la Arquitectura de Software puede ser ampliada con los conceptos adecuados, de modo que sea capaz de describir una estructura reflexiva de manera organizada. Puesto que una arquitectura dinámica es de hecho una estructura reflexiva, se asume que un modelo arquitectónico con reflexión sería un marco adecuado para la descripción de cualquier tipo de dinamismo.

La hipótesis de trabajo de esta tesis es, por tanto, que *la introducción de Reflexión en la Arquitectura de Software proporciona un marco unificado para la descripción de Dinamismo*. La demostración de esta hipótesis constituye el objetivo global de esta tesis. Este será acometido mediante la consecución de una serie de objetivos parciales, que se enumeran a continuación.

1.2 Objetivos

Con el fin de mostrar que la introducción de Reflexión en la Arquitectura de Software conduce al dinamismo, se desarrollará un trabajo en cinco etapas, cuya estructura se describe a continuación:

- En primer lugar, se hará un estudio general del campo de la Arquitectura de Software Dinámica. Esto incluye una definición de sus conceptos fundamentales, así como una revisión detallada de los distintos enfoques existentes. A lo largo de este estudio se identificarán las estructuras inherentes a dichos enfoques, que son las que conducen a la descripción de dinamismo. De este modo, se obtendrá una definición de los distintos modelos de dinamismo existentes, independiente de las aproximaciones parciales utilizadas.

- En segundo lugar, se hará una revisión general de las nociones *estructurales* implicadas en el concepto de Reflexión. Se extraerán, de entre éstas, aquéllas que se consideren relevantes en el contexto de la Arquitectura de Software, teniendo en cuenta los resultados del punto anterior. La introducción de estas nociones en un esquema general permitirá la

elaboración de un *modelo arquitectónico reflexivo*, denominado MARMOL, que se construirá con independencia de cualquier lenguaje o modelo existente, con el fin de ser totalmente general. De este modo, las conclusiones de este punto podrían ser posteriormente aplicadas a cualquier ADL particular, con independencia de las conclusiones del resto de este trabajo. El modelo será definido en términos de teoría de conjuntos y de relaciones; de este modo se logrará hacer una descripción puramente estructural, elaborada sobre un formalismo totalmente independiente de cualquier enfoque particular.

- En tercer lugar, se ha de considerar que el modelo anterior sólo proporciona la *infraestructura* para una descripción, pero de hecho no permite realizarla. Para llegar hasta ese punto, es necesario aplicar el modelo sobre un lenguaje concreto. En lugar de utilizar un lenguaje preexistente, que ya tuviera una orientación particular, se ha preferido utilizar un lenguaje nuevo, específicamente diseñado como un ADL reflexivo, al que se ha dado el nombre de $\mathcal{P}i\mathcal{L}ar$. La conjunción de este lenguaje y el modelo anterior proporcionan ya un marco reflexivo para la descripción arquitectónica, en el que basar el estudio de la hipótesis de partida. No obstante, este lenguaje será diseñado desde un punto de vista genérico, ampliando de este modo su aplicabilidad.

- En cuarto lugar, se ha de acometer una formalización completa del lenguaje diseñado. Esto cumple dos funciones principales: en primer lugar, da una definición precisa de sus nociones, estructuras básicas e instrucciones, con lo que el lenguaje queda descrito sin ninguna ambigüedad. En segundo lugar, se demuestra que el lenguaje no sólo puede ser concebido, sino que puede ser de hecho llevado a la práctica, e implementado sobre un contexto no reflexivo. El sistema formal elegido para su formalización ha sido el cálculo-π tipado polimórfico, convenientemente adaptado al contexto de uso. Sobre este cálculo, se pretende obtener un correlato formal de cada una de las nociones utilizadas en el diseño del lenguaje, comprobando además que en efecto se ha elaborado un sistema reflexivo, en cuyo interior se puede realizar la descripción de dinamismo.

- Finalmente, se comprobará la aplicabilidad del sistema elaborado, esto es, la capacidad de $\mathcal{P}i\mathcal{L}ar$ para acometer la descripción de arquitecturas de software, y en particular la de arquitecturas dinámicas. De este modo se obtendrá una perspectiva de la verdadera utilidad del esquema –modelo, lenguaje y semántica–, que permita, en términos generales, determinar la bondad de la hipótesis de partida.

En este punto, se habrá demostrado, al menos, que un enfoque reflexivo de la Arquitectura de Software proporciona un entorno apto para la descripción de dinamismo, y que la construcción y utilización de dicho entorno es en efecto posible.

1.3 Estructura de la Tesis

Este trabajo de tesis se estructura en nueve capítulos, agrupados en cinco partes, y cuatro apéndices. El esquema se organiza de modo que se parte de los aspectos más genéricos, y se van concretando poco a poco los más específicos.

La primera parte, *Introducción*, consta únicamente de un capítulo, éste mismo. En él se introduce brevemente el contexto general de esta tesis, centrándolo en el problema de la descripción de arquitectura dinámica, y sugiriendo la hipótesis de que este problema puede ser acometido mediante la introducción de nociones reflexivas.

La segunda parte, *Arquitectura de Software Dinámica*, consta de dos capítulos. El primero de ellos es una introducción parcial a la Arquitectura de Software, cuyo objetivo es proporcionar un contexto adecuado para el resto de la exposición. De este modo, se define el campo y se resalta su importancia, para luego centrarse en sus aspectos descriptivos. Por fin, se revisan con cierto detalle sus conceptos fundamentales; la mayoría de ellos serán utilizados con frecuencia a lo largo de los capítulos restantes.

El capítulo tres se centra ya específicamente en la cuestión de la descripción de dinamismo en la Arquitectura de Software. Consiste, en realidad, en una revisión global de este aspecto –la Arquitectura Dinámica–, que no sólo sitúa a este trabajo en un contexto, sino que resulta de interés en sí mismo. Se considera el problema, se justifica su interés, y se pone en relación con el término vinculado de Reconfiguración Dinámica, que define un contexto a partir del cual se enumeran las operaciones fundamentales del dinamismo. Se pasa entonces a hacer un estudio detallado de los distintos lenguajes y enfoques utilizados en la literatura para la descripción de arquitecturas dinámicas. En una primera etapa, estos enfoques se clasifican según su origen conceptual; posteriormente, se identifican siete modelos básicos de dinamismo arquitectónico, y se los relaciona con los lenguaje examinados con anterioridad.

La tercera parte, *Reflexión*, consta únicamente de un capítulo, el cuarto. Su objetivo es tan sólo dar una breve perspectiva global del concepto de Reflexión y sus derivados, con el fin de establecer una base adecuada de la que luego extraer las nociones que serán integradas en un contexto arquitectónico. Tras considerar brevemente el problema general de la autorreferencia, y contrastar sus diferencias con el concepto de meta-programación, se pasan a detallar sus principales nociones derivadas, en particular la de *reificación*. Se hace también especial énfasis en los distintos tipos de Reflexión, y se describe el modelo de capas de cebolla, uno de sus usos más importantes. Finalmente, se examinan las combinaciones existentes entre Reflexión y Arquitectura de Software, de nuevo con el fin de situar a este trabajo en contexto.

La cuarta parte, *Arquitectura Dinámica mediante Reflexión*, consta de cuatro capítulos, y constituye el verdadero núcleo de este trabajo. En el capítulo quinto se justifica la aproximación reflexiva a la descripción de arquitecturas dinámicas mediante una comparación con los modelos identificados en el capítulo tres. A partir de ésta, se identifican las nociones reflexivas que se consideran de utilidad en el contexto de la Arquitectura de Software, y se elabora con ellas un modelo de descripción arquitectónica denominado globalmente Marmol. De éste se hace primero una exposición informal, que se completa después con una definición formal, hecha en términos de teoría de conjuntos y relaciones. Ésta se divide en tres etapas: en la primera ($\aleph_\Sigma$) se hace una definición de los conceptos básicos de la Arquitectura de Software en términos de relaciones. En la segunda ($\aleph_\mathrm{M}$), se introducen las nociones reflexivas, integrándolas con las anteriores. Finalmente, en la tercera ($\aleph_\Pi$) se realiza un colapso entre las relaciones de reificación y tipado, con el fin de simplificar la especificación de reflexión estructural.

En el sexto capítulo se toman los conceptos básicos de Marmol desde una perspectiva lingüística, con el fin de mostrar su verdadero impacto en la descripción arquitectónica. De este modo se elabora el lenguaje $\mathcal{P}i\mathcal{L}ar$, concebido específicamente como un Adl reflexivo, pero de

propósito general. El capítulo se dedica esencialmente a describir su sintaxis, identificando los dos sublenguajes de los que se compone –Estructural y Dinámico–, así como cada una de sus estructuras de control. Se hace especial hincapié en la visión lingüística del concepto de reificación, y se justifica el enfoque algebraico utilizado en la elaboración del Lenguaje Dinámico. Una vez descrito el lenguaje básico, se discute en detalle el modo en que la reificación de conexiones da lugar a la concepción de *conectores de nivel meta*. Finalmente, se describe una extensión específica del lenguaje que proporciona un soporte para la definición de *envolventes*.

En el séptimo capítulo, se proporciona una semántica formal para $\mathcal{P}i\mathcal{L}ar$ en términos de una variante del cálculo-π. Se justifica la elección de este cálculo, y se esbozan algunas de las consecuencias que tendrá el enfoque reflexivo sobre éste. Tras una discusión sobre los problemas derivados de la definición de otros ADLs algebraicos, se justifica el nivel de abstracción utilizado, y se pasa a describir con detalle la estructura que adquieren las nociones del lenguaje desde una perspectiva semántica; en particular, el complejo esquema derivado de la definición de procesos "reflexivos". A partir de aquí, se definen formalmente todos los conceptos del Lenguaje Estructural y todos los comandos del Lenguaje Dinámico, dando una traducción completa y consistente de $\mathcal{P}i\mathcal{L}ar$, que demuestra su viabilidad.

En el capítulo octavo se examinan, a la luz de todo lo anterior, una serie de ejemplos de aplicación de $\mathcal{P}i\mathcal{L}ar$, con el doble objetivo de mostrar el modo en que puede ser usado, y demostrar su potencia y versatilidad. Tras un ejemplo inicial anecdótico, que hace patente su simplicidad en condiciones normales, se examinan una serie de casos particularmente interesantes o conflictivos.

Finalmente, la quinta parte, *Conclusiones*, consta de un único capítulo, el noveno. En éste se recogen las conclusiones del trabajo realizado, y se sugieren varias líneas de trabajo futuro, a través de las cuales se podrían aclarar algunos de los aspectos menos tratados de esta tesis, así como ampliar en gran medida su posible campo de aplicación.

Todos los capítulos están fuertemente interrelacionados, y dan en conjunto una visión completa del campo, y del enfoque utilizado. Aunque la aportación esencial de este trabajo de tesis se concentra en la parte cuarta –y especialmente en los capítulos cinco, seis y siete–, el razonamiento global se extiende y justifica en la totalidad del mismo.

Por su parte, los cuatro Apéndices se ocupan de una serie de aspectos particulares relacionados con puntos concretos de este trabajo de tesis, cuyo tratamiento detallado en un contexto particular hubiera resultado excesivo.

El Apéndice A incluye una especificación completa de la gramática formal del lenguaje $\mathcal{P}i\mathcal{L}ar$ en notación EBNF, que complementa a la definición informal del Capítulo 6. Se comentan brevemente algunos aspectos de la misma, incluyendo ciertos detalles de la fase léxica.

El Apéndice B hace una introducción somera a los conceptos básicos de un *cálculo nominal*, centrándose de manera clara en el cálculo-π, principal formalismo utilizado en este trabajo, que es también el más importante de todos ellos. Se proporciona también una breve descripción de su sintaxis, así como una semántica intuitiva.

El Apéndice C describe el dialecto concreto del cálculo-π tipado que se utiliza en las definiciones formales del Capítulo 7, al que se denomina abreviadamente cálculo-$\hat{\pi}$. Se trata, específicamente, de una variante poliádica, síncrona, de primer orden, tipada, polimórfica y no extendida, inspirada directamente en el cálculo-π polimórfico de Turner [Tur96]. A éste se le

añade, además, un soporte básico para tipos de datos primitivos y una sintaxis específica para pares, registros y listas, así como un tipo dinámico (*Dyn*).

Finalmente, el Apéndice D describe una serie de definiciones auxiliares, desarrolladas en términos del cálculo-$\hat{\pi}$, que completan algunas de las definiciones semánticas del Capítulo 7. Se considera que no tienen entidad suficiente para ser descritas como parte del lenguaje, pero algunas de ellas serán, no obstante, muy habituales.

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

Parte II

Arquitectura de Software Dinámica

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

Capítulo 2

Arquitectura de Software

2.1 Introducción

La Arquitectura de Software es la parte de la Ingeniería de Software que se ocupa de la descripción y el tratamiento de la estructura de un sistema como una serie de componentes, con el fin de organizar adecuadamente los distintos subsistemas, y permitir la integración de diferentes grupos de desarrollo en el mismo proyecto. Habitualmente se la vincula con la fase de Diseño, aunque este detalle no es estrictamente necesario [MT98]; su principal objetivo es hacer énfasis en la importancia de la descripción estructural de los sistemas software, un aspecto bien conocido pero habitualmente poco tratado.

Incluso hoy, con la disciplina relativamente establecida pero aún joven, no es posible dar una definición o descripción de Arquitectura de Software que resulte completa, o cuando menos suficientemente integradora. Hay, no obstante, un *corpus* conceptual común, en torno al cual existe cierto grado de consenso. Por ello, lo que se hace en este Capítulo es proporcionar una breve introducción a los conceptos fundamentales del campo, centrada exclusivamente en los aspectos *lingüísticos* de la Descripción Arquitectónica.

Este Capítulo, por tanto, no pretende ser una revisión del campo, ni dar un resumen adecuado de las distintas aproximaciones que lo componen; este aspecto es ya cubierto, por otra parte, por una serie de artículos generales sobre el tema [Per97, Sar97, Gar00, MT00, Sha01], así como por la creciente bibliografía disponible [SG96, BCK98, HNS00, JRvdL00, MM00, Bos00]. Su único objetivo es proporcionar una mínima base conceptual, así como una nomenclatura unificada, que sirva de apoyo al resto de esta tesis. Así pues, en las secciones siguientes únicamente se hará una definición general del campo, se señalará su relevancia, y se desarrollarán brevemente sus nociones principales. Se dará entonces por concluido este Capítulo, para entrar ya a considerar el problema específico tratado en esta tesis, esto es, la descripción de dinamismo (§3).

2.2 Definición de Arquitectura de Software

El significado intuitivo de la expresión *Arquitectura de Software* resulta, en general, bastante claro. Tal vez por ello, no existe una definición única y aceptada de la misma; distintos autores utilizan distintas definiciones, según el énfasis que quieran hacer en distintos puntos. Es muy

habitual, de hecho, citar varias de ellas de manera simultánea, con el fin de proporcionar una visión de conjunto. En la misma línea, en esta sección se mencionan las dos probablemente más completas, y luego se discuten brevemente otras aproximaciones.

La definición siguiente tiene la ventaja de ser breve y concisa, y es sin lugar a dudas la más utilizada en la literatura:

> *La **arquitectura de software** está compuesta por la estructura de los componentes de un programa o sistema, sus interrelaciones, y los principios y reglas que gobiernan su diseño y evolución a lo largo del tiempo.*
>
> *David Garlan y Dewayne Perry [GP95]*

Aunque es incompleta, poco explícita y en parte ambigua, resulta ser una definición general, abierta, no excluyente, amplia y flexible. También es unificadora, en el sentido que es válida tanto para la visión descriptiva –representada por Garlan– como para la visión de proceso –representada por Perry–. En conclusión, es habitualmente aceptada como un mínimo común denominador. De hecho, la definición proporcionada en la Práctica Recomendada sobre Arquitectura de Software de la IEEE [AWG99] no es sino una versión más imprecisa de ésta.

La definición siguiente fue propuesta con anterioridad, y resulta mucho más detallada, aunque no es, en sentido estricto, una definición de arquitectura de software, sino una justificación del enfoque, que incluye la formulación de sus objetivos:

> *Más allá de los algoritmos y estructuras de datos de la computación, el **diseño y especificación de la estructura global** del sistema emerge como un nuevo tipo de problema. Los detalles estructurales incluyen la organización general y la estructura de control global; los protocolos de comunicación, sincronización y acceso a datos; la asignación de funcionalidad a los elementos de diseño; la distribución física; la composición de los elementos de diseño; su escalabilidad y los aspectos de rendimiento; y la selección entre alternativas de diseño.*
>
> *David Garlan y Mary Shaw [GS93]*

Lo cierto es que esta enumeración delimita adecuadamente los aspectos de interés de una descripción arquitectónica. Sin embargo, no ha de ser considerada en un sentido estricto, ya que muchos de ellos sólo han sido considerados de manera parcial en los desarrollos existentes.

En resumen, siempre que se define *Arquitectura de Software* se tiende a identificar la noción de estructura, refiriéndose tanto a su concepción como a su especificación. Sin embargo, la expresión no es utilizada únicamente en ese sentido, sino que a medida que se desarrolla el campo se han podido identificar al menos cuatro significados diferentes, todos ellos interrelacionados, a saber:

- *Arquitectura como Producto.* Éste es el significado por excelencia, y al que se refieren las definiciones proporcionadas. En este sentido, la *arquitectura* es el conjunto formado por la organización, la estructura y la infraestructura de un sistema software. Desde esta perspectiva, interesa ante todo su *descripción*, evaluación y análisis.

- *Arquitectura como Proceso.* Habitualmente se considera que el *proceso arquitectónico* se constituye como el principal interés del campo. En este sentido, se hace referencia tanto

al método a seguir para deducir la arquitectura de un sistema, existente o no, como a la elaboración de una metodología de desarrollo que considere explícitamente el impacto de la arquitectura de software [Kru95].

- *Arquitectura como Campo de Estudio.* En los últimos años, el estudio de la estructura del software ha llegado a ser un interés en sí mismo, hasta el punto de que comienza a aparecer como una disciplina propia. En este sentido, se deben considerar tanto los estudios específicos existentes, como toda una serie de aspectos fuertemente relacionados, que abarcan desde los métodos formales, constituidos en herramienta básica, hasta los patrones de diseño, cuyo objetivo, origen e inspiración es similar.

- *Arquitectura como Profesión.* Aunque este último aspecto es muy discutible, lo cierto es que con cierta frecuencia comienza a aparecer la figura del *arquitecto de software*, definido como el rol encargado de diseñar y mantener la descripción de la arquitectura, por lo que implícitamente es el que proporciona la visión integral del producto. Se establece, por tanto, como una figura clave en el proceso de desarrollo.

En esta tesis, el término se considera fundamentalmente en el primer sentido, y dentro de éste se hace especial énfasis en los aspectos de *descripción*, particularmente desde un punto de vista lingüístico. Esta elección viene determinada por la convicción de que sólo cuando se alcance un nivel de descripción lo suficientemente preciso, se podrán concretar los objetivos de la Arquitectura de Software. De otro modo, el análisis requerido no es posible, y la información obtenida durante el proceso se pierde en gran medida.

2.3 Relevancia de la Arquitectura de Software

Algunos autores, y en particular Mary Shaw [Sha80, Sha90], consideran que el actual énfasis en Arquitectura de Software es una consecuencia lógica de la evolución en el desarrollo de sistemas software, que siempre ha tendido a elevar el nivel de abstracción utilizado. Shaw identifica ciclos de aproximadamente diez años, en los que el concepto básico del campo ha ido cambiando, adquiriendo una forma progresivamente más abstracta: de la instrucción a la función, luego al módulo, y por fin al objeto. En esta visión, la fase actual se corresponde precisamente con la Arquitectura de Software, y su noción fundamental es la de componente arquitectónico (§2.4.2).

Es obvio, por otra parte, que en los últimos años la importancia del campo y su difusión han crecido de manera considerable. En la Figura 2.1 de la página 14 puede verse una cronología aproximada de los principales hitos del campo, así como de los grandes proyectos –financiados– que han influido de manera decisiva en su evolución.

El punto de partida elegido para esta cronología ha sido 1989, fecha en la que comenzó a difundirse el artículo seminal de Dewayne Perry y Alexander Wolf, posteriormente publicado como [PW92], y que suele considerarse como el principio del actual auge en Arquitectura de Software. No obstante, pueden citarse precedentes mucho más antiguos: de hecho, en ocasiones algunos autores se han remontado hasta los trabajos iniciales de David Parnas [Par72] sobre encapsulamiento, aunque probablemente esto sea excesivo.

En la Figura puede verse con claridad cómo la actividad en el campo ha crecido de manera considerable a lo largo de la última década, alcanzando su máxima expresión en el cambio de

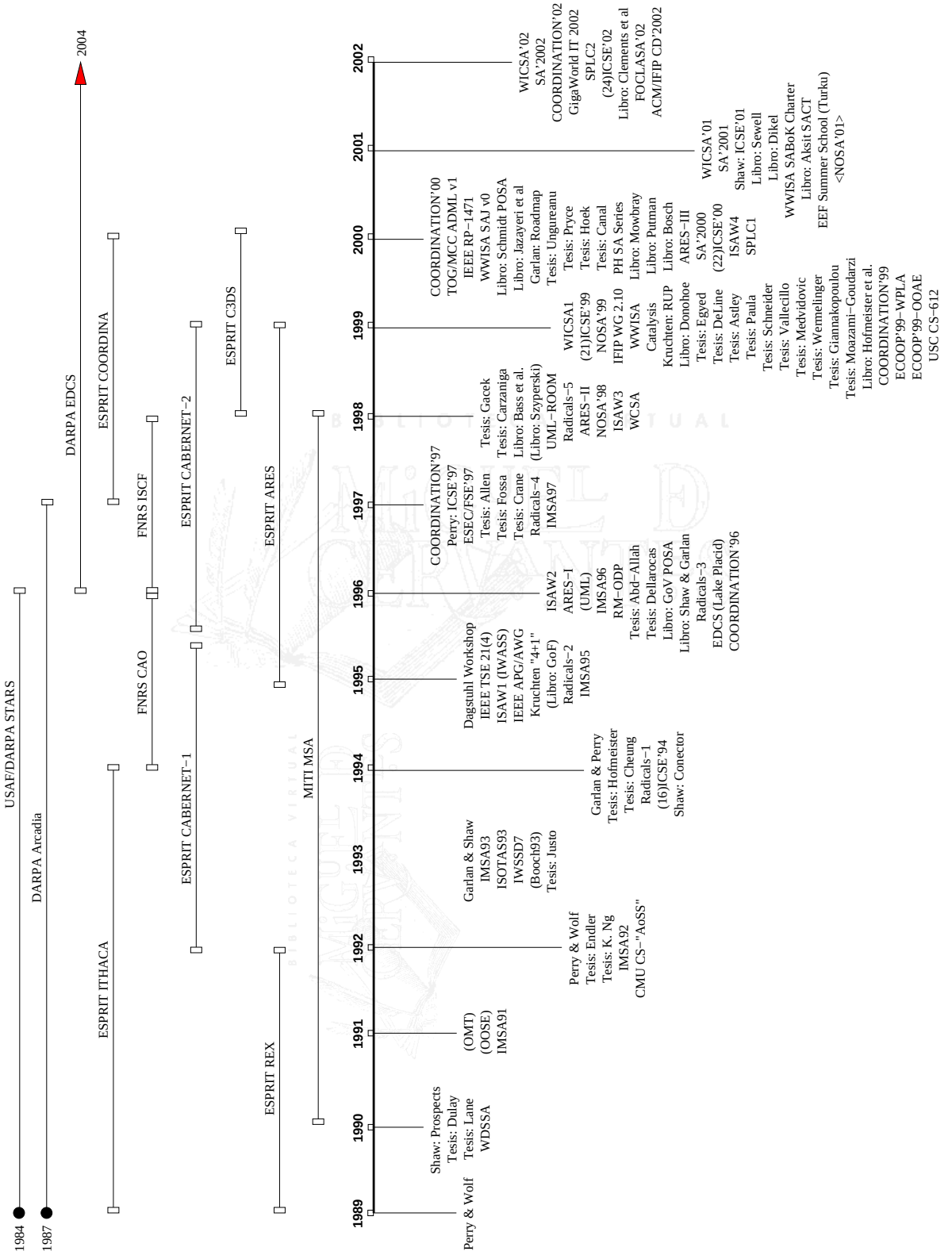

Figura 2.1: La Arquitectura de Software en el Tiempo: Hitos y Proyectos

milenio. No debe pensarse, sin embargo, que este impulso ha remitido en los últimos dos años; muy por el contrario, lo que ocurre es que la Arquitectura de Software se ha integrado plenamente en la Ingeniería de Software, y que apenas existe foro general en el que no se hayan incluido trabajos relacionados con el campo. El mejor ejemplo es la propia Conferencia Internacional de Ingeniería de Software (ICSE), que ha tratado el tema de una manera especial en los últimos cuatro años. Por otra parte, la consolidación de una conferencia específica, la Conferencia IFIP sobre Arquitectura de Software (WICSA) hace cada vez menos necesaria la dispersión en reuniones de menor envergadura.

Conviene resaltar el papel de los proyectos financiados, en particular en los Estados Unidos, donde el máximo impulsor ha sido sin duda la investigación militar. El proyecto STARS (*Software Technology for Adaptable, Reliable Systems*), tal vez uno de los que mayor influencia ha tenido en la Ingeniería de Software en general, fue también el que introdujo el concepto de *arquitectura de software*, y la causa original del interés de la industria en el mismo. Una vez finalizado éste, sus conclusiones dieron lugar a otro proyecto, el Programa EDCS (*Evolutionary Design of Complex Systems*), específicamente centrado en el concepto de arquitectura, y en el que participa un enorme número de empresas y universidades. Su objetivo es plantear una nueva visión de los sistemas software complejos como entidades sometidas a continuos cambios; el papel de la arquitectura en este contexto es el de servir como elemento unificador.

La Arquitectura de Software también ha conocido una aceptación plena, tanto en el plano científico como en el industrial. La IEEE inició en 1995 un proceso de estandarización [EHIP+96] de las ideas del campo, que cristalizaron en 2000 con la elaboración de la *Práctica Recomendada* RP-1471 [AWG99]. Por otra parte, el Open Group ha publicado la primera versión de una norma para la descripción arquitectónica, denominada ADML, con base en XML. No obstante, el detalle más significativo es sin duda la constitución de un grupo de trabajo específico dentro de la IFIP, concretamente el WG 2.10, que reune a algunas de las figuras más importantes del campo, tanto del mundo académico como del industrial.

No se ha de dejar de mencionar, siquiera desde un punto de vista anecdótico, la fundación del Instituto Mundial de Arquitectos de Software (WWISA) [Sew00], que pretende consolidar la visión de la Arquitectura de Software como profesión, y en el que participan también cierto número de personalidades vinculadas al Diseño de Software, tales como Grady Booch, Philippe Kruchten, Tom Mowbray, Ian Graham o Terry Winograd.

2.4 Conceptos Fundamentales

A lo largo de esta sección, se revisarán los conceptos fundamentales del campo, con el fin de proporcionar una visión de conjunto razonable, que sirva como una base adecuada para la comprensión de los capítulos posteriores.

Como ya se ha dicho, esta revisión se centra únicamente en los aspectos de *descripción* arquitectónica, por lo que el principal interés es el estudio de los distintos lenguajes utilizados para este propósito, conocidos como *Lenguajes de Descripción de Arquitectura* o ADLs. Las características que los definen son también brevemente discutidas en este apartado.

Desde el punto de vista lingüístico, una arquitectura de software viene determinada por los *componentes* –elementos básicos caracterizados por una *interfaz* segmentada en *puertos*– y

conectores que la constituyen, así como por una serie de *conexiones* o enlaces específicos, que definen la unión de todos ellos formando una estructura. A esta estructura se le da el nombre de *configuración*, y suele considerarse insertada en una *jerarquía*, independientemente de su granularidad. Ocasionalmente, la configuración no se describe de manera monolítica, sino que se estructura en diferentes *vistas*, cada una de las cuales se concentra en un aspecto diferente. Cuando lo que interesa no es obtener una configuración concreta, sino los patrones genéricos que definen a una familia de sistemas, se habla de *estilos arquitectónicos* [CFBS98]. En definitiva, es la definición de todos estos aspectos la que determina una visión concreta de la Arquitectura de Software; a este fin se dedicarán los apartados siguientes.

2.4.1 Lenguajes de Descripción de Arquitectura

Se define un *Lenguaje de Descripción de Arquitectura* (*Architecture Description Language*, o ADL) como una notación que permite una descripción y análisis preciso de las propiedades observables de una arquitectura de software, dando soporte a distintos estilos arquitectónicos a diferentes niveles de abstracción [Sch99].

La necesidad de utilizar una notación específica para la especificación arquitectónica, que permita separar con claridad los aspectos vinculados a la estructura del resto de los detalles del desarrollo, ha sido identificada casi desde el principio. Esto no significa, no obstante, que todo lenguaje que haga una descripción específica de la estructura sea propiamente un ADL. En las etapas iniciales de desarrollo del campo, han recibido esta denominación [Cle96] todo tipo de lenguajes, desde sistemas puramente formales, hasta lenguajes de especificación algebraica, de programación concurrente o de definición de interfaces. En particular, ha resultado confusa su relación con otros lenguajes específicos de dominio, como los Lenguajes de Configuración y de Interconexión de Módulos (MILS), algunos de los cuales han evolucionado posteriormente hasta llegar a ser auténticos ADLs (§3.7.2).

Con el fin de clarificar la situación, Shaw y Garlan [SG94] enunciaron las seis propiedades que, según su punto de vista, debe tener todo ADL:

- **Composición.** El lenguaje debe ser tal que el arquitecto podrá dividir un sistema complejo en partes más pequeñas, de manera jerárquica, o construir un sistema a partir de los elementos que lo constituyen.

- **Abstracción.** La arquitectura expresa una abstracción que permite identificar a los distintos elementos en una estructura de alto nivel, así como su papel en la misma. Esta abstracción es específica, y se diferencia de otras utilizadas en el desarrollo.

- **Reutilización.** Un objetivo fundamental de la especificación arquitectónica es hacer posible la reutilización posterior de componentes, conectores, estilos y arquitecturas, incluso en un contexto diferente a aquél en el que fueron definidos.

- **Configuración.** El lenguaje debe separar con claridad la descripción de elementos individuales y de la de las estructuras –elementos compuestos– en las que éstos participen. Para Medvidovic [MT00], ésta es la característica fundamental que distingue a un ADL.

- **Hetereogeneidad.** El ADL ha de ser independiente del lenguaje en que se implemente cada uno de los componentes que manipula; y debe diseñarse de modo que pueda combinar patrones arquitectónicos diferentes en un único sistema complejo.

- **Análisis.** La descripción arquitectónica debe poder ser analizada, de modo que se puedan determinar sus propiedades con independencia de una implementación concreta, así como verificarlas después de cualquier modificación. Esto suele vincularse al uso de métodos formales, que permitan una definición sin ambigüedad semántica.

En realidad, la mayoría de los ADLs existentes carecen de alguna de estas propiedades, incluyendo a los definidos por los propios Garlan y Shaw; pero, tomada en un sentido amplio, esta caracterización resulta suficiente para delimitar con claridad qué es lo que *no* constituye un ADL propiamente dicho, a pesar del parecido que pueda mostrar en otros aspectos.

Por otra parte, Nenad Medvidovic [MT00] ha elaborado, con base en esta enumeración, un marco de comparación y clasificación de ADLs, que permite realizar una verdadera taxonomía. El marco es muy detallado, e incluye una gran variedad de puntos, de modo que se puede estructurar adecuadamente cualquier propuesta existente. También se identifican qué aspectos se consideran esenciales para determinar que un lenguaje es, efectivamente, un ADL: concretamente, la estructuración en componentes, la especificación de conectores, la definición de sus interfaces, y, sobre todo, la descripción *explícita* de las configuraciones compuestas por todos ellos.

A pesar de todo lo dicho, en los últimos años ha habido cierta tendencia a intentar expresar una arquitectura de software utilizando notaciones de propósito general, sobre todo UML y su combinación con ROOM [SR98]. Más todavía, el Proceso Unificado de Rational [Kru99] se basa en un uso intensivo de arquitecturas descritas mediante UML, e incluso existe una propuesta completa [HNS00] para la adaptación de UML a este propósito. No obstante, y aunque es cierto de los mecanismos de *estereotipos* permiten una modificación de la notación estándar más cercana a las necesidades del campo, debe de reconocerse con Medvidovic y Rosenblum [MR99] que este enfoque resulta, en términos generales, contraintuitivo e inadecuado.

2.4.2 Componente

El concepto fundamental de la Arquitectura de Software es el de *componente*. Esto se refiere, en términos globales, a cada una de las partes o *unidades de composición* –por definición– en las que se subdivide la funcionalidad de un sistema, y cuya unión da lugar al sistema completo. En su sentido más genérico, puede hacer referencia a cualquier tipo de elemento estructural, esto es, integrado en una estructura; y es precisamente con este significado con el que habitualmente se le utiliza en la Arquitectura de Software.

El término de *componente* no se limita a la arquitectura, sino que es de hecho utilizado en múltiples campos de la Ingeniería de Software desde la propuesta realizada por Douglas McIlroy [McI69] en la célebre conferencia de la OTAN en Garmisch-Patenkirchen. En realidad, su connotación más habitual hace referencia más bien a aspectos de implementación, vinculados a los estudios de Desarrollo Basado en Componentes. Lo cierto es que, en todo caso, el término suele ir acompañado de cierta confusión: aunque existe cierto consenso en torno a una idea intuitiva, resulta muy difícil obtener una definición satisfactoria [BDH+98]. Ni siquiera los

distintos intentos por proporcionar una descripción formal del concepto [Bro96] han tenido un éxito claro, de modo que el significado preciso de la noción es aún un tanto difuso.

No obstante, la siguiente definición ha alcanzado cierta popularidad y se considera comúnmente aceptada:

> *Un* componente *es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio.*
>
> *Clemens Szyperski [Szy98]*

Aunque se sustenta, de nuevo, en aspectos de implementación, lo cierto es que es lo bastante general como para ser aplicada a cualquiera de los contextos en los que la palabra es normalmente utilizada, incluyendo a la Arquitectura de Software. Desde este punto de vista, un componente es por tanto una parte de la arquitectura fuertemente encapsulada y con un interfaz bien definido, que se concibe con independencia del resto del sistema, y que se integra en el mismo mediante mecanismos de composición e interacción.

En realidad, lo único en lo que están de acuerdo todos los autores es en que un componente ha de presentar una interfaz definida al resto del sistema, en la que se indica qué proporciona y, habitualmente, también lo que necesita. Desde este punto de vista, un componente es, prácticamente por definición, una *caja negra*, en la que el encapsulamiento llega a ser incluso más rígido que en el concepto de objeto. Esto se aplica por igual a los componentes de implementación, de especificación y arquitectónicos.

En el contexto del desarrollo de componentes, las ventajas de tal grado de encapsulamiento –sea por abstracción, por seguridad o por motivos comerciales– son innegables; pero las posibilidades de adaptación quedan limitadas, de modo que se hace necesario el uso de *envolventes* [Höl93], *adaptadores* o *mediadores* [GHJV94] (ver sección 4.6). Ese papel ha sido asumido en Arquitectura de Software, normalmente, mediante el uso de *conectores* desarrollados específicamente para la ocasión. Por ello, no es extraño que cada vez más autores propongan un cambio en este sentido, de modo que todo componente conste de una *interfaz privilegiada* que permita algún tipo de acceso, con grados relativos de seguridad, a detalles internos. Esto define lo que se ha dado en llamar una *caja gris* [BW97], un concepto ya propuesto con anterioridad para la Orientación al Objeto [KLM+97, HL95].

De hecho, la propuesta para la descripción arquitectónica que se presenta en esta tesis puede enmarcarse en esta línea. Aunque el objetivo de la introducción de Reflexión en el modelo –como se verá en el Capítulo 5– es facilitar la descripción de dinamismo, debe de tenerse en cuenta que los esquemas reflexivos suponen, precisamente, la especificación de un interfaz privilegiado para los elementos situados en el meta-nivel. De este modo, se obtendrá la definición de componentes como una caja gris, como un simple efecto secundario de la incorporación de Reflexión.

2.4.3 Conector

El concepto de conector procede principalmente de los trabajos de Mary Shaw [SG96], a partir de su experiencia en Unicon [SDK+95a]. En un célebre artículo [Sha94], propuso considerar por

separado las abstracciones relativas a la funcionalidad (el componente) y a su interacción (el conector). De este modo, se realiza una clara *separación de intereses*, que permite ampliar el nivel de abstracción y aumentar la modularidad del sistema.

Sin embargo, lo que se propone no es simplemente disponer de dos tipos de componentes, sino de distinguir dos elementos diferenciados, con funciones muy dispares. Los componentes normales –elementos de computación– realizan una tarea sin preocuparse de cómo se relacionan con el resto del sistema; por su parte, los elementos e interacción, denominados *conectores*, son los que se encargan de resolver todas las cuestiones relativas a la comunicación de los primeros. Shaw insiste en que los conectores deben ser considerados como *elementos de primera clase*, que tienen significado por sí mismos. Esto quiere decir que no serán definidos en función de otros elementos, ni diseñados específicamente para un componente, sino que podrán ser extraídos y considerados en otro contexto.

La forma más sencilla de ver a un conector es como la encarnación de un protocolo de comunicación, entendido en su sentido más amplio. En general, cualquier artificio –o artefacto– que permita comunicarse a dos o más elementos es un conector. Por ejemplo, la llamada de procedimiento es un tipo clásico de conector.

No obstante, la definición del concepto supone que hay *dos* tipos de vínculo entre los distintos elementos de una descripción arquitectónica: por una parte, el propio *conector* expresa la interacción existente entre varios componentes; pero por otra, esto exige establecer, a su vez, el tipo de enlace que relaciona a cada componente con un conector determinado: este enlace recibe normalmente el nombre de *adjunción* o *attachment* (§3.7.1, §3.7.3, §3.8.2). Sin embargo, a lo largo de esta tesis le denominaremos simplemente *conexión*, término que consideramos más intuitivo y menos forzado, y cercano a su uso convencional en español[1].

Es importante señalar que existen ciertas diferencias de matiz en cuanto al uso de la palabra *conector*. Podría decirse, incluso, que se utiliza con dos sentidos diferentes, aunque relacionados. Por una parte, suele entenderse que la propuesta original de Shaw exige la definición de un nuevo tipo de elemento, análogo a un componente, y que se describe del modo indicado. Sin embargo, también se puede mencionar la palabra *conector*, de modo general, como haciendo referencia a cualquier interacción explícita entre dos componentes, lo que incluye a los *bindings* de *δarwin* (§3.7.2) o a las *conexiones* de Rapide (§3.8.1). Este es, por ejemplo, el sentido en que lo usa Medvidovic en [MT00], cuando señala que la definición de conectores es uno de los rasgos que caracterizan a un ADL (§2.4.1). Sin embargo, en pro de la claridad, en esta tesis se utilizará la palabra únicamente en el primer sentido, es decir, como elemento de primera clase, haciendo siempre una distinción explícita entre *conector* y *conexión*.

Incluso en su forma más básica, esto es, cuando se los considera como simples conexiones, la enumeración de enlaces o asociaciones explícitas entre los elementos de un sistema proporciona una descripción, siquiera parcial, de su estructura; es decir, de su *arquitectura*. Puede decirse con Medvidovic [MT00], pues, que por el mero hecho de disponer de una noción de conector, un lenguaje ya está reforzando su capacidad para especificar configuraciones, lo que constituye la tarea principal de cualquier ADL. De hecho, uno de los primeros trabajos sobre Rapide [LVM95], uno de los lenguajes con un modelo de conexión más simple, demuestra sin embargo que esta

[1] En inglés, del mismo modo, preferimos el término más general de *binding*, utilizado en algunos lenguajes sin conectores, como *δarwin*, aunque es tal vez más cercano a aspectos formales o de implementación.

característica, por sí sola, resulta suficiente para distinguir con claridad un ADL de un lenguaje orientado al objeto convencional. Tal vez, sugerir la posibilidad de una confusión entre estos dos tipos de lenguajes resulte ahora extraño; sin embargo, y debido al énfasis que ambos hacen en el concepto de encapsulación, éste es un punto que fue ampliamente debatido en su momento [Cle96], e incluso reaparece, todavía hoy, de manera ocasional.

Ha de tenerse en cuenta que se puede ver a los conectores de dos maneras, a menudo antagónicas, pero que no tienen por qué serlo: como una especificación, o como una simple implementación. El mejor ejemplo de lo primero son los conectores de Wright (§3.7.1), mientras que los de Unicon son un adecuado ejemplo de lo segundo. En un lugar intermedio podrían situarse los de C2 (§3.8.2). Para Wright, los conectores son ante todo especificaciones que indican *qué es lo que se espera* de un componente en una interacción dada. Es decir, se trata ante todo de indicar el papel de cada uno de los componentes en cada uno de los protocolos; si la especificación del conector se corresponde con las de los componentes, se puede verificar la corrección del sistema. Para Unicon, en cambio, un conector no es más que una implementación de un protocolo; su objetivo es, ante todo, evitar que el diseñador del componente tenga que preocuparse de los aspectos de interacción. Por tanto, se plantea como un elemento reutilizable que puede ser conectado a un componente en un momento dado, y se encarga desde ese momento de realizar las interacciones apropiadas.

Nikunj Mehta [MMP00] ha hecho un estudio exhaustivo de todo aquello que ha sido o es considerado un conector; aunque es el principio de una taxonomía que se hace claramente necesaria, no constituye más que un trabajo preliminar, y lamentablemente no parece haber tenido continuidad. No obstante, un trabajo de este tipo, completamente desarrollado, será imprescindible para que el concepto puede llegar a alcanzar todo su potencial. Eventualmente, se espera que una comprensión adecuada de la naturaleza del conector permita la definición de un *álgebra de conectores*, que permita su combinación, permitiendo la descripción de interacciones aún más elaboradas, en las que se ha elevado el nivel de abstracción. Los conectores así obtenidos son los que han sido llamados *conectores de orden superior* [Gar98]; aunque en los últimos años ha habido cierto progreso en esta línea [SG01, LWF01], todavía está en sus etapas iniciales, y es aún poco más que una idea.

Debe tenerse en cuenta que el propio concepto de conector está en discusión. Aunque se trata de una idea generalizada y es aceptada de manera común, no todos los autores están plenamente convencidos de su necesidad. Algunos, entre los que destaca Jeff Kramer, opinan que la existencia de los conectores *distorsiona* la naturaleza compositiva de una arquitectura de software, que queda afectada de manera negativa. Ciertamente, mientras que la composición de dos elementos de estructura similar –dos componentes– resulta fácil de expresar, sea formal o informalmente, es claro que la introducción de un segundo tipo de elemento –el conector– complica la situación. No se trata simplemente de una yuxtaposición de funcionalidades, sino que ha de considerarse el tipo de composición que define el conector. Existe además toda una serie de problemas derivados, como es la diferencia precisa entre componente y conector, o cuál es la naturaleza exacta de un compuesto intermedio, en el que alguno de los extremos –o *roles*– de un conector quedase libre. Se ha argumentado que un "componente de interacción", concebido de manera equivalente a un conector, pero definido de forma análoga a otros componentes, mantiene las posibilidades de reutilización sin perder la composición. Aunque esto rechaza la posibilidad que lo consideraba como una abstracción básica, no contradice explícitamente a la

propuesta de Shaw: el mero hecho de plantear la diferencia entre las dimensiones de *composición* e *interacción* es ya importante, y constituye su verdadera esencia.

En definitiva, la noción de conector se ha convertido en uno de los rasgos definitorios del campo: sea como un componente específico, como una conexión compleja, o como noción por propio derecho, aparecerá siempre, de algún modo, en toda descripción arquitectónica.

2.4.4 Puerto

El concepto de *puerto* es cercano al de conector, pero no debe ser confundido con éste bajo ningún concepto. Se denomina con este nombre a cada uno de los puntos por los que un componente puede realizar cualquier tipo de interacción; dicho de otro modo, es cada uno de los fragmentos en los que se *segmenta* el interfaz de un componente.

En definitiva, hace referencia a un punto de entrada o de salida de la caja negra que es, de hecho, el componente. Distintos autores lo ven y denominan de distinto modo, y la analogía que implícitamente se establece no es idéntica en todos los casos. Así, si el componente se ve como un objeto, un puerto definiría un método; mientras que si se ve como un proceso, se tiene un canal de mensajes. Ha de tenerse en cuenta, sin embargo, que los puertos de un componente no sólo expresan los servicios que éste oferta, sino *también* los requisitos que precisa; esto es, aquellas condiciones que necesita que cumpla el entorno para funcionar correctamente. Por todo ello, la analogía con los métodos de un objeto es simple, pero peligrosa.

Los puertos se han denominado de distintos modos en distintos lenguajes; así, en Wright se llaman *puertos* en los componentes, pero *roles* en los conectores, mientras que en Unicon reciben el nombre de *jugadores*. En $\delta arwin$ se conocían inicialmente como *puertos*, para después pasar a llamarse *nombres de interfaz*, de manera consistente con su semántica en cálculo-π. Actualmente conocen como *portales*, un nombre que expresa total neutralidad respecto del signo de la interacción: es decir, pueden ser de entrada o de salida.

Habitualmente los puertos se agrupan definiendo una interfaz; en algunos lenguajes se permite, incluso, que definan más de una. Incluso en algunos sistemas se asume que el puerto está subestructurado en varios puntos de entrada, y por tanto se define todo él como una interfaz completa. Ése es el caso, por ejemplo, de Koala [vOvdLKM00].

En los lenguajes que carecen de conectores, el concepto de puerto es aún más importante. En la segunda versión de $\delta arwin$ (§3.7.2), por ejemplo, los puertos están fuertemente tipados, de modo que el tipo expresa su posible comportamiento. La complejidad puede llegar hasta tal punto que los niveles de abstracción resulten confusos. Así, por ejemplo, en Unicon el mecanismo de RPC es un conector, mientras que en $\delta arwin$ esto define simplemente el tipo *port* de un portal. Algo análogo sucede con los mecanismos de difusión de eventos y el tipo *event* de un puerto. Además, los portales de $\delta arwin$ son móviles, pueden separarse del componente e incluso representarlo de manera remota como un *proxy*, lo que supone una función habitualmente reservada ya no a un conector, sino incluso a un componente.

En todo caso, su definición es un aspecto fundamental, ya que configura la naturaleza externa de los componentes, lo que a su vez condiciona la estructura de la arquitectura.

2.4.5 Estilo Arquitectónico

La noción de *estilo* es otro de los conceptos fundamentales de la Arquitectura de Software, y posiblemente el más importante de todos ellos. Tiene tanto de aspiración como de realidad, y el futuro del campo depende de la forma que llegue a tomar definitivamente. Describe y proporciona las propiedades básicas de una arquitectura, y también es el encargado de imponer los límites a su evolución, lo que lo relaciona estrechamente con el dinamismo arquitectónico.

Se denomina **configuración**, siguiendo la nomenclatura de Garlan, a la arquitectura de software de un sistema *concreto*, esto es, a la estructura específica constituida por unos componentes y conectores determinados. Por contraposición, se denomina **estilo arquitectónico** a cada uno de los patrones genéricos que pueden identificarse en un grupo de sistemas relacionados [CFBS99]. Es decir, una configuración es una instancia concreta de un estilo. Este concepto, con ser el más original del campo, lo relaciona directamente con el estudio de los *patrones de diseño*. Ésta no es su única coincidencia: ambos nacen de la analogía con la Arquitectura clásica, y en especial con los trabajos de Christopher Alexander [Ale81, AIS+80, Lea94]

En general, se puede definir *estilo arquitectónico* como la descripción de una familia de sistemas en términos de un patrón de organización estructural [Gar95, SG95]. Más específicamente, la especificación de un estilo define:

- Un **vocabulario** de tipos de componentes –y conectores– de los que se dispone.
- Un conjunto de **restricciones** sobre el modo en que éstos pueden ser combinados.
- Opcionalmente, un **modelo semántico** que determine cómo se pueden deducir las propiedades globales del sistema a partir de las propiedades de cada uno de sus elementos.

Existe un consenso generalizado en la idea de que cualquier descripción razonable de estilos arquitectónicos se ha de basar en la combinación de estos elementos; más aún: únicamente en ellos. Sin embargo, ninguna de las propuestas concretas actuales parece ser satisfactoria, o mostrar suficiente expresividad como para capturar de manera adecuada la idea *intuitiva* de estilo. El único lenguaje que ha intentado la definición de estilos con un enfoque generalista ha sido Wright [All97]. Sin embargo, sus estilos están todavía a demasiado bajo nivel, acercándose demasiado a la noción de configuración. De manera análoga, Padl intenta describir simplemente un concepto intermedio, al que denomina, con poca fortuna, *tipo arquitectónico*. Otros lenguajes, como Aesop [GAO94] o Asdl [RS96] resultan aún más limitados.

Aunque todavía no existe un método satisfactorio para describir estilos en ningún lenguaje, se han realizado intentos diversos que, en alguna medida, parecen ir bien encaminados. Todos estos enfoques, de algún modo, se aproximan al objetivo:

- Mediante una *arquitectura abierta*, esto es, un fragmento de una arquitectura que ha de ser completada para dar lugar a un sistema completo; esto define una gran parte de sus propiedades y proporciona implícitamente un estilo. Esto, de nuevo, puede considerarse en relación tanto con los patrones de diseño como con los *frameworks* [Bos98].
- Mediante una *arquitectura conceptual*, es decir, una serie de ideas y abstracciones que definen el vocabulario básico del sistema, y sus relaciones [Ran99].

- Mediante una *gramática de grafos*, considerando a la arquitectura como un grafo. Este es un enfoque más limitado de lo que parece, ya que restringe en exceso el rango de sistemas posibles. Existen, no obstante, diferentes aproximaciones (§3.9).
- Mediante la definición de unas *restricciones lógicas* que indican las características básicas que la arquitectura ha de mantener en todo momento. Es el caso, por ejemplo, de los estilos de Wright [All97].

- Estilo Arquitectónico
  - Sistemas de Flujo de Datos
    - Secuenciales por Lotes
    - Filtro/Tubería
    - Sistemas de Control Clásico
  - Sistemas de Llamada-Respuesta
    - Árbol de Módulos (Subrutina)
    - Sistemas Orientados a Objetos
    - Capas Jerárquicas
    - Cliente-Servidor
  - Componentes Independientes
    - Procesos Comunicantes
    - Sistemas de Eventos
  - Máquinas Virtuales
    - Intérpretes
    - Sistemas Basados en Reglas
  - Repositorios
    - Bases de Datos
    - Sistemas de Hipertexto
    - Pizarras

Figura 2.2: Taxonomía de Estilos, adaptada de [SG95]

Aunque el soporte lingüístico no está aún claro, el estudio de los estilos ha sido acometido desde un punto de vista informal, y se han identificado un buen número de ellos, así como algunas de sus características más significativas. No se entrará aquí a comentarlos en detalle; no obstante, en la Figura 2.2 se incluye una taxonomía preliminar, con el fin de mostrar el tipo de conceptos que se manipulan en este contexto.

No se profundizará más aquí en este concepto, ni en sus relaciones con otras nociones cercanas, tales como la de *línea de producto* [JRvdL00] o la de familia de sistemas. Aunque se trata de uno de los aspectos más interesantes y fundamentales de la Arquitectura de Software, su relación con la descripción de arquitectura dinámica, que constituye el tema principal de esta tesis, es más bien indirecta. En este sentido, lo único que hace es *restringir* las posibilidades de evolución, imponiendo límites. No obstante, es un tema que apenas ha sido estudiado, salvo de modo marginal (§3.9.2, §3.9.3), y todavía está pendiente de recibir un tratamiento adecuado.

2.4.6 Vista

La noción de *vistas*, aunque fue mencionada por los primeros autores del campo, ha sido la última en ser propuesta de manera explícita, y su éxito ha sido inmediato, ya que, por ejemplo, es el único aspecto obligatorio en la Práctica Recomendada RP-1471 sobre Arquitectura de Software de la IEEE [AWG99], que adopta explícitamente un enfoque multi-vista de la Arquitectura de Software.

El concepto de *vista* se limita a señalar la posibilidad de que una arquitectura de software pueda examinarse desde diferentes perspectivas –literalmente, puntos de vista–, de modo que cada una de estas arquitecturas "parciales" constituye una vista. Luego, la visión completa del sistema se adquiere por combinación de todas ellas. Se mantiene, de nuevo, la metáfora de la arquitectura clásica, por analogía con los distintos planos de un mismo edificio.

El concepto no asume que el ADL utilizado haya de tener un soporte de múltiples vistas: habitualmente, se asume que se tienen varias descripciones del mismo sistema, típicamente en el mismo lenguaje, y la visión global se obtiene de modo puramente heurístico. Lo cierto es que ningún lenguaje existente ha definido un soporte explícito para vistas; el tratamiento ha sido siempre muy informal, por lo que este soporte resultaría muy conveniente. Admite, además, una interesante analogía con la denominada *separación de intereses avanzada*, esto es, la generalización de la AOP (§4.6), traducida en conceptos parcialmente sinónimos como el de sujeto, dimensión o aspecto.

Los primeros en sugerir un enfoque multi-vista fueron los propios Perry y Wolf, que ya desde el principio [PW92] identificaron que una arquitectura habría de tener una *vista de flujos de datos*, otra de *control*, y otra de *uso de recursos*. Las tres consisten, por tanto, en enfoques muy tradicionales, cercanos incluso a los desarrollos contemporáneos en metodologías orientadas al objeto, como OMT [RBP+91] o Booch'93 [Boo96].

Sin embargo, sin duda la propuesta más difundida es el denominado *Modelo 4+1* de Philippe Kruchten, que fue diseñado específicamente dentro del contexto de la Arquitectura de Software [Kru95], pero ha sido posteriormente popularizado por su adaptación a UML y su difusión dentro del campo de la orientación al objeto. La Figura 2.3 muestra el modo en que el modelo fue modificado para obtener una correspondencia más directa con UML.

Vista Lógica	$\longrightarrow$	Vista Lógica
Vista de Procesos	$\longrightarrow$	Vista de Procesos
Vista de Desarrollo	$\longrightarrow$	Vista de Implementación
Vista Física	$\longrightarrow$	Vista de Despliegue
Vista de Escenarios	$\longrightarrow$	Vista de Caso de Uso

Figura 2.3: Evolución de Vistas en el Modelo 4+1 de Kruchten

Aunque es un aspecto que no será considerado a lo largo de esta tesis, no debe dejarse de señalar que la combinación de Reflexión en un ADL crea una infraestructura particularmente adecuada para el tratamiento de múltiples vistas. Obviamente, la identificación de los puntos comunes –los *puntos de unión* de la AOP– se haría en base a componentes concretos, y la influencia de las distintas vistas puede expresarse mediante reificaciones sobre estos puntos. De este modo, la combinación se produciría por simple combinación paralela, con un método similar a la propuesta de Andrews [And01] para los fundamentos de la AOP. No obstante, este enfoque requiere un estudio más detallado, y será considerado como trabajo futuro.

2.5 Conclusión

Como ha podido verse en este mismo capítulo, el campo de estudio de la Arquitectura de Software se ha extendido de manera considerable en los últimos años, a la par que su importancia a continuado creciendo. Aquí se han revisado apenas algunos de sus conceptos fundamentales, desde el punto de vista de la *descripción* arquitectónica, con el fin de proporcionar una base adecuada para la comprensión de capítulos posteriores. Algunas nociones, como la de componente, conector o puerto serán decisivas; otras, como la de estilo o vista, serán sólo mencionadas de manera general, ya que no se relacionan directamente con los objetivos de esta tesis.

Una vez se ha proporcionado una mínima introducción al campo, la exposición se centrará ya en la problemática principal a acometer en esta tesis, esto es, la especificación de dinamismo en la Arquitectura de Software, y la unificación de las distintas aproximaciones a este problema, con base en el concepto de reflexión.

BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Capítulo 3

Arquitectura de Software Dinámica

Tratar de entender [la estructura de ejecución de un programa]
a partir de [la estructura de su código] es como
tratar de entender el dinamismo de los ecosistemas vivos
a partir de la taxonomía estática de plantas y animales, y viceversa.
Erich Gamma [GHJ+95b]

3.1 Introducción

En este Capítulo se proporciona un estudio general sobre la Arquitectura de Software Dinámica o, para ser más exactos, sobre los distintos enfoques utilizados por aquellos ADLs dotados de capacidades dinámicas. Existen al menos dos motivos de importancia para incluir aquí un estudio de estas características. En primer lugar, como ya se ha comentado en la Introducción (§1), la especificación de dinamismo es uno de los objetivos primarios de esta tesis, y el interés principal durante el diseño del lenguaje $\mathcal{P}i\mathcal{L}ar$ (§6), si bien no el único. En segundo lugar, resulta oportuno realizar dicho estudio debido al hecho de que, hasta la fecha, no existía ninguno similar. Las revisiones existentes del campo se reducían a breves recensiones en algunos artículos –apenas [Med96a, OT98], o trabajos propios como [CFBSB00, CFBSB01]–, y resúmenes en algunas tesis doctorales [Cra97, MG99, Med99, Pry00, Wer99], normalmente incompletos. En realidad, el tratamiento más amplio –si bien no el más extenso– disponible se limitaba a los cinco párrafos dedicados al tema en las secciones 4.3.7 y 4.4.6 de la conocida clasificación de ADLs de Medvidovic [MT00]. En resumen, una revisión detallada del campo era aún necesaria.

Se pretende que la revisión incluida en este Capítulo sea lo suficientemente completa; no obstante, esto no quiere decir que se incluya en ella *todo* lenguaje que haya afirmado ser un ADL dinámico. De hecho, interesa hacer un tratamiento lo más preciso posible, realmente centrado en el dominio arquitectónico; por tanto, se evitará en gran medida discutir todos aquellos lenguajes que no puedan considerarse un auténtico ADL; en este sentido se utilizará un criterio bastante abierto, aunque se tendrán en cuenta los criterios establecidos por Medvidovic.

En particular, se dejarán de considerar todas aquellas aproximaciones a la Arquitectura de Software Dinámica que no definen una contrapartida lingüística, y aquellos lenguajes que se basan únicamente en la existencia de una implementación concreta. Esto excluye a un buen

número de lenguajes de composición y configuración, y también a los Lenguajes de Descripción de Interfaces (IDLs), entre los que merece especial mención el de CORBA.

No obstante, a lo largo de este capítulo se comentarán brevemente algunos Lenguajes de Interconexión de Módulos (MILs), como Conic o Durra; pero no se considera necesario hacer una revisión detallada de todos ellos, ya que, de nuevo, no se trata de ADLs propiamente dichos. Por este motivo no se entran a detallar algunos sistemas indudablemente interesantes, y que muestran cierto grado de dinamismo, tales como Gerel [EW92, End94], Polylith [HP93, Pur94], CL [JC92] o Aster [IB96, BBI+00].

3.2 Arquitectura de Software Dinámica

El rápido desarrollo que ha conocido la Arquitectura de Software a lo largo de la última década no es en absoluto casual: el software siempre ha tenido una estructura, y algunos de los sistemas clásicos ya mostraban una arquitectura elaborada [Dij68]. Sin embargo, esta estructura nunca ha sido tan importante como ahora: si no en tamaño, los sistemas han crecido en diversidad y complejidad. Esto se hace especialmente patente en los actuales sistemas abiertos, distribuidos, concurrentes, y en red.

Dichos sistemas son altamente *dinámicos*, y muestran una estructura *evolutiva*. Continuamente se añaden, eliminan, o reemplazan componentes, a veces incluso como parte del funcionamiento normal. Estos detalles ya no son coyunturales, sino estructurales, y deben tenerse en cuenta durante el Diseño del sistema. Por tanto, para ser realmente útil, un Lenguaje de Descripción de Arquitectura (ADL) debe ser capaz de especificarlos. Sin embargo, esto no es lo normal en los ADLs existentes, cuya naturaleza es mayoritariamente estática. Los pocos que muestran capacidades dinámicas muestran también limitaciones, que dependen en gran medida de sus orígenes particulares.

Cuando la evolución de una arquitectura es continua, y los cambios en su interior siguen un patrón predefinido, entonces ha de considerarse que este patrón *forma parte intrínseca de la estructura*. El objetivo de la Arquitectura de Software es describir la estructura de los sistemas, y esto debería incluir tanto a sus partes fijas –estáticas–, como a las cambiantes –dinámicas–. De hecho, en muchos sistemas, el rasgo diferencial, que lo distingue de otros sistemas similares, es el comportamiento dinámico de su arquitectura; a menudo, puede resultar aún más importante que el esquema estático de partida. Algunos ejemplos de sistemas para los que la definición de dinamismo es esencial son los siguientes:

- Los sistemas *abiertos* y *distribuidos*, cuya estructura cambia de manera constante, a medida que los elementos se incorporan o se disocian de él, o son sustituidos.
- Los sistemas *evolutivos* de varias clases, que se ajustan a un cierto patrón.
- Los sistemas *continuos*, *de tiempo real* y *tolerantes a fallos*, en los que a menudo se requieren cambios, pero que no pueden ser detenidos para ser actualizados.
- Los sistemas *adaptativos* o auto-organizados, que modifican su propia estructura para adaptarla a cada entorno concreto.

- Los *sistemas multi-agente* (MAS), dentro y fuera del contexto de Inteligencia Artificial; y en general todos los sistemas capaces de expresar algún grado de *movilidad*.
- O, por ejemplo, los sistemas *reflexivos*, que son interesantes en sí mismos, pero que además constituyen la base de la propuesta final de este trabajo.

Claramente, muchos de estos sistemas están de plena actualidad, lo que ha hecho más patente la necesidad de una especificación clara de sus patrones dinámicos. Para cumplir sus objetivos, la Arquitectura de Software debería ser capaz de acometer las descripciones de este tipo.

En los últimos años, el estudio específico de las *arquitecturas dinámicas* ha adquirido una importancia cada vez mayor dentro de la propia Arquitectura de Software. La mejor muestra es, sin duda, el hecho de que los sistemas dinámicos y evolutivos sean el objeto de estudio fundamental del Proyecto EDCS (*Evolutionary Design of Complex Systems*) [Tra96] de DARPA, y en especial de su *cluster* de Arquitectura y Tecnología de Generación [Ger96], probablemente el mayor impulsor, en términos financieros, de la investigación en este campo.

Hace tan sólo cinco años, los trabajos específicos en Arquitectura de Software Dinámica eran escasos y aislados. La lista de ADLs –reconocidos como tales– con capacidades dinámicas era muy corta, y podía limitarse a considerar los casos de *δarwin* (§3.7.2), Rapide (§3.8.1), C2 (§3.8.2), la versión dinámica de Wright (§3.7.1) y el enfoque gráfico de Le Métayer (§3.9.2). Hoy en día, todas estas propuestas han sido desarrolladas, y muchas otras se han ido incorporando; muchos sistemas anteriores han sido adaptados o reconocidos como ADLs dinámicos, y se han desarrollado varias tesis doctorales en todo el mundo [Cra97, MG99, Med99, dP99, Sch99, Wer99, Pry00, CV00] dedicadas a este tema de algún modo. En definitiva, el desarrollo e interés creciente de este campo es innegable, y se presenta, sin duda alguna, como uno de las líneas de trabajo más fructíferas de la próxima década.

3.3 Tipos de Dinamismo

Todo sistema software cuya estructura es descrita como una arquitectura tiene siempre asociado algún tipo de comportamiento dinámico. Ahora bien, es obvio que este dinamismo no siempre indica el mismo grado de variabilidad. Algunos sistemas muestran una estructura inalterable, en la que los distintos elementos se limitan a comunicarse; en otros, la topología interna se altera continuamente, e incluso se incorporan nuevos tipos de elementos.

Resulta fundamental, para poder realizar una comparación adecuada entre las distintas aproximaciones, delimitar los diferentes tipos de dinamismo que pueden aparecer en un sistema software. La clasificación que se incluye más abajo ha sido planteada con este único objetivo. No es exacta, ni pretende ser determinante; el límite entre las distintas categorías puede aparecer difuminado en algunos casos. Sin embargo, parece servir adecuadamente a su objetivo, esto es, aclarar el tipo de actividad al que se hace referencia en cada caso; y ya ha sido publicada en dos ocasiones [CFBSB00, CFBSB01].

Según esto, entonces, pueden identificarse los siguientes tres tipos de dinamismo, en orden creciente de potencial para el cambio:

- **Dinamismo Interactivo**. Este término se refiere al movimiento de *datos* entre componentes, el dinamismo inherente en la comunicación e interacción usual. Se relaciona con

cuestiones tales como la sincronización y los protocolos, y afecta, no a la estructura, sino al *comportamiento* de un sistema computacional. Está presente de manera implícita en todo sistema, pero resulta particularmente obvio en los concurrentes y paralelos.

- **Dinamismo Estructural**. Hace referencia a los cambios que afectan a los propios componentes y sus interacciones, y que normalmente se expresan como la creación y destrucción de instancias y enlaces entre componentes. Por tanto, la arquitectura de software del sistema resulta alterada, porque *lo que cambia es la estructura*. Este tipo de dinamismo es natural en sistemas distribuidos.

- **Dinamismo Arquitectónico**. Se refiere a los cambios en los *tipos* de componente e interacción, modificando por lo tanto incluso el significado y "vocabulario" del sistema. Se expresa como la inserción de nuevos tipos de componente (y conector), y el borrado de algunos ya existentes. Ahora, lo que cambia es la *infraestructura*, el sustrato en el que las arquitecturas se definen, su meta-nivel. Éste es el nivel definitivo de dinamismo, y también el más difícil de capturar y utilizar. Aún hay que demostrar su verdadera importancia y utilidad, pero está claramente presente en los (auténticos) sistemas *abiertos*.

Cada uno de estos niveles incluye, obviamente, a los anteriores. El primero existe *entre* elementos arquitectónicos, luego la arquitectura no es afectada. El segundo manipula componentes arquitectónicos, produciendo por lo tanto variaciones (*configuraciones*) de una única descripción arquitectónica (*estilo*). El tercero modifica *arquitecturas* como un todo, y por tanto crea otras completamente nuevas.

O a la inversa, esto significa que si se divide a los sistemas en categorías de acuerdo con esta taxonomía, los que manifiestan menor dinamismo incluyen a aquéllos con mayor dinamismo conceptual. Cualquier sistema que se considere, incluso los clásicos –secuenciales y monolíticos–, muestra el del primer tipo (*interactivo*). El segundo tipo (*estructural*) se encuentra en los sistemas compuestos (lo que incluye los distribuidos) y configurables. El tercer tipo (*arquitectónico*) es el menos común, y se relaciona con el grado de apertura, las capacidades reflexivas o incluso la intervención externa (humana).

En todo caso, los tres niveles son interesantes. Una descripción arquitectónica no está verdaderamente completa hasta que expresa *toda* la dinámica de un sistema. La mayoría de los ADLs existentes son capaces de expresar la del primer tipo, incluso utilizando notaciones formales, como en Wright (§3.7.1). Muy pocos son capaces de manipular el segundo, siendo $\delta arwin$ (§3.7.2) el más conocido de ellos. Pero, hasta donde alcanza nuestro conocimiento, ninguno de ellos considera el del tercer tipo, y sólo Rapide (§3.8.1) es parcialmente compatible con él.

Esta taxonomía muestra con claridad los problemas con los que se enfrenta el tratamiento del dinamismo durante el diseño de software. Diferentes aspectos, correspondientes a distintos niveles de dinamismo, se han mezclado muy a menudo, creando construcciones confusas y complejas. Por ejemplo, la sincronización (interactiva) y la distribución (estructural) son a menudo consideradas de manera simultánea; en el fondo, esto es poco natural, y genera problemas. La intuición que sugiere esta aproximación es engañosa, porque es debida a una confusión en la causa de su naturaleza dinámica.

3.4 Reconfiguración Dinámica

Hay al menos dos aproximaciones a la especificación de dinamismo en sistemas software. Una de ellas trata de encontrar un conjunto adecuado de comandos **imperativos**, capaces de expresar cualquier configuración deseable. La otra utiliza una sintaxis **declarativa**, que es generalmente más potente e intuitiva, pero no siempre tiene la flexibilidad deseada. El trabajo sobre este tema se centró inicialmente en el primer enfoque; sin embargo, a medida que se alcanza una mejor comprensión del dinamismo estructural, la presencia del segundo tiende a aumentar.

A menudo, la denominación de *Arquitectura de Software Dinámica* ha sido discutida. Muchos autores prefieren el término *Reconfiguración Dinámica*, que ha sido consagrado por años de trabajo previo en el campo específico de la configuración de sistemas distribuidos. Con esa expresión se hace referencia, en general, a los cambios producidos en la *topología* de un sistema compuesto (esto es, a cualquier alteración en el número y orden de los nodos y los enlaces que lo constituyen), que es el grado de dinamismo que actualmente se está alcanzando en los Lenguajes de Descripción de Arquitecturas. Según Jeff Kramer, el nombre de *Arquitectura Dinámica* debería reservarse para referirse a los sistemas en los que incluso el *tipo* de componentes implicados puede cambiar. Este es, precisamente, el nivel de descripción al que se pretende llegar, por lo que a lo largo de este trabajo esa expresión será utilizada a menudo, sin mayores consecuencias.

Hay que tener en cuenta, no obstante, que una buena parte de la investigación que se ha realizado en el campo de la Configuración se ha llevado a cabo al margen de la Arquitectura de Software, y tiene muy poco que ver con ella [Wer99]. La máxima preocupación, en muchos casos, es determinar cómo construir un sistema que realice reconfiguraciones seguras, o desarrollar un sistema formal que preserve el *estado* de la computación, o que al menos lo tenga en cuenta a la hora de reconfigurar. Por supuesto, incluso una buena parte del trabajo que sí se describe en este Capítulo tiene que ver también con estas cuestiones, pero haciendo énfasis en cómo esto afecta a la *Arquitectura*, no a una cierta implementación o sistema operativo. Esto se puede aplicar a toda una línea de investigación basada en *δarwin* (§3.7.2), o a propuestas más abstractas como el enfoque categórico de Wermelinger (§3.9.4).

Por supuesto, muchas de esos estudios externos a la Arquitectura de Software extraen conclusiones que podrían aplicarse a este campo. Afortunadamente, no hay aquí necesidad de describirlos, ya que se puede encontrar un estudio reciente, muy claro y bastante completo, en la tesis de Kaveh Moazami-Goudarzi [MG99].

3.4.1 Tipos de Reconfiguración

Hay varios tipos distintos de reconfiguraciones posibles, que manifiestan problemáticas muy diferentes. Con el fin de determinarlas con mayor exactitud, se han introducido una serie de clasificaciones. En este trabajo se mencionarán dos de ellas, a saber:

- **Según el tiempo**. Uno de los aspectos críticos en un cambio es el momento en que se produzca. La dificultad para realizarlo de manera adecuada crece según se avanza en el proceso de desarrollo.

  - **Antes de la Compilación.** Los cambios en la estructura prevista del sistema se producen antes de su construcción, por lo que sólo hay que asegurar su consistencia.
  - **Antes de la Ejecución.** El sistema ya ha sido elaborado, pero aún no está funcionando, por lo que introducir un cambio sólo requiere, de nuevo, asegurar la consistencia interna del sistema. Puede equipararse sin problemas con el caso anterior.
  - **Durante la Ejecución.** Esta última es la auténtica *Reconfiguración Dinámica*: los cambios se producen con el sistema en marcha: los componentes tienen un estado, y sus enlaces soportan cierto número de transacciones. La problemática generada por este tipo de situaciones ha sido el campo de investigación de la comunidad de Configuración en los últimos diez años.

- **Según el origen**. Es decir, según cómo se inicie el proceso; en definitiva, quién toma la iniciativa. Existen dos posibilidades, que son las siguientes:

  - **Reconfiguración Programada.** Reciben este nombre [EW92, MG99] las operaciones que son iniciadas por el propio sistema, mediante una serie de reglas proporcionadas como parte de su especificación. También se ha mencionado como *reconfiguración restringida en tiempo de ejecución* [Ore96], y es el nivel de descripción que se desea alcanzar en la Arquitectura de Software.
  - **Reconfiguración *Ad-Hoc*.** Además de este nombre [EW92], ha recibido los de reconfiguración *evolutiva* [MG99] o *en tiempo de ejecución* [Ore96][1]. Hace referencia a los cambios que son introducidos manualmente por el usuario, y que por tanto requieren en concurso de algún tipo de herramienta, que haga corresponderse a la arquitectura con un sistema real. Obviamente, no pueden ser previstos, aunque en general sí pueden ser controlados.

La primera clasificación es obvia, y ha sido planteada entre otros autores por Wermelinger [Wer99]. La segunda se debe inicialmente a los trabajos de Markus Endler en Gerel [EW92], un lenguaje ligado a $\delta arwin$, pero ha sido redescubierta en múltiples ocasiones por otros autores; esto explica la gran variedad de terminologías utilizadas.

3.4.2 Operaciones Básicas de Reconfiguración

Desde casi el principio, se ha determinado que toda configuración ha de poder ser descrita en función de cuatro operaciones básicas, cuyo establecimiento procede de Conic (§3.6.1). En realidad, no es muy difícil deducirlas de la simple observación de un grafo, considerando en este caso al grafo como la descripción más simple posible de la topología de cualquier sistema, en la que los componentes se expresan como nodos y sus enlaces como aristas.

Las cuatro operaciones mencionadas son las siguientes:

- **Crear (*create*).** Se refiere a la creación de nuevos componentes; para ser exacto, la instanciación de un nuevo ejemplar de un tipo de componente conocido.

[1] Sin embargo, no ha de ser confundido con el mencionado con este nombre en la clasificación anterior.

- **Destruir (*destroy*).** La inversa de la anterior: permite destruir un componente ya existente; para ser más exacto, eliminarlo del sistema.

- **Enlazar (*link*).** Crea un enlace simple entre dos elementos; no se trata de un conector, sino de lo que se ha denominado como *conexión* o *adjunción*.

- **Desligar (*unlink*).** La inversa de la anterior: destruye un enlace preexistente.

No se quiere decir con esto que todo sistema de reconfiguración haya de tener estas cuatro operaciones indicadas de manera explícita; al contrario, muchos las expresan mediante estructuras declarativas, y ni siquiera tienen un equivalente claro para algunas de ellas; lo único que se quiere decir es que *toda* reconfiguración, por compleja que sea, puede realizarse mediante la aplicación sucesiva de estas cuatro operaciones. Del mismo modo, si un sistema no es capaz de expresar una transformación que, sin embargo, puede ser descrita en función de estas cuatro operaciones, entonces el sistema no es completo, en lo que a este tema respecta.

Oreizy [Ore96] cita un buen número de operaciones no básicas; sin embargo, todas ellas son reducibles a combinaciones de éstas. En más de una ocasión se ha propuesto la **sustitución** de un componente por otro del mismo tipo –o de un tipo compatible– como una operación común. Ciertamente, lo es, pero no es básica; puede ser fácilmente emulada por una serie de *unlink*s de los enlaces del elemento original, el *destroy* de éste, el *create* del nuevo, y la regeneración de los enlaces mediante los *link*s correspondientes.

En realidad, estas operaciones sólo proporcionan lo que aquí se ha denominado *dinamismo estructural* (§3.3), ya que únicamente permiten alterar la *topología* del sistema. Sin embargo, constituyen lo que podría considerarse el límite mínimo exigible a cualquier sistema que pretenda expresar arquitecturas dinámicas. No obstante, este aspecto no es trivial; de hecho, algunos de los sistemas descritos en este Capítulo (§3.9.2, §3.9.3) no son capaces de indicar operaciones destructivas, lo que reduce la expresividad a la mitad de este "mínimo".

3.5 Clasificación: por Origen

A lo largo de las secciones siguientes –desde la 3.6 hasta la 3.13– se realiza una revisión general de los principales enfoques utilizados para la descripción arquitectónica. Concretamente, se consideran tan sólo aquéllos que explícitamente consideran un *lenguaje*, siempre que éste pueda ser o haya sido explícitamente considerado como un ADL, y muestre algún grado de dinamismo.

Con el fin de organizar la exposición, se ha decidido agrupar estas propuestas en base a su *origen* conceptual, con un criterio abierto. No se intenta establecer aquí una clasificación tajante, o absolutamente rígida: aunque el criterio básico de agrupación sigue siendo el concepto básico –típicamente formal– en el que se fundamentan, a veces han primado los aspectos de semejanza o de cercanía histórica. Es por esta razón por la que un sistema basado en el modelo Entidad-Relación, como AML, se agrupa con los lenguajes basados en álgebras de procesos; o por la que se discuten brevemente algunos ADLs cuyo dinamismo es más bien potencial, como ACME.

A lo largo de la revisión quedará claro, no obstante, que hay dos modelos conceptuales que predominan claramente sobre cualquier otro; esto no es solamente cierto en los ADLs dinámicos, sino incluso en los convencionales. Se trata de los *lenguajes algebraicos*, fundamentados en

álgebras de procesos (§3.7) –si bien en este grupo pueden incluirse también, sin problemas, los enfoques basados en eventos (§3.8)–, y los *lenguajes gráficos*, esto es, aquellos que se basan en conceptos de teoría de grafos (§3.9), o incluso teoría de categorías. Otros enfoques, como los modelos químicos (§3.10) o los basados en coordinación dirigida por datos (§3.11) han alcanzado también cierta popularidad, pero su difusión es claramente mucho menor.

3.6 Sistemas Clásicos

En este apartado se describe, con cierta brevedad, la estructura y forma de algunos lenguajes y sistemas que definieron los conceptos básicos de lo que todavía se denomina Reconfiguración, y que aquí se propone bajo el nombre de Arquitectura de Software Dinámica. En aquel momento, estos lenguajes fueron descritos bajo la denominación de "Lenguajes de Configuración" o "Lenguajes de Interconexión de Módulos" (MILs), en el sentido de que su origen se relacionaba con la posibilidad de configurar una herramienta o sistema software determinado, o de proporcionar una estructura general para reunir elementos independientes. En su momento, la novedad de estas propuestas fue proporcionar una *separación de intereses*, entre la Configuración y la Computación, permitiendo por vez primera disociar los aspectos puramente algorítmicos de un sistema (distribuido) de aquellos relacionados con su estructura; por ello, se los considera un claro precedente de la Arquitectura de Software, y muchos de ellos han recibido posteriormente –o reclamado para sí– la denominación de ADLs. Dado que muchos de ellos mostraban capacidades para alterar, incluso en tiempo de ejecución, la topología de un sistema, no sólo describen arquitecturas, sino incluso arquitecturas *dinámicas*.

En la actualidad, ninguno de estos sistemas sigue en uso, aunque a diferencia de otros, no han caído por completo en el olvido; sin embargo, muestran en fechas muy tempranas unas capacidades bastante avanzadas. Su abandono se debe, en muchos casos, al hecho de que la mayoría estaban ligados a un lenguaje de programación concreto, como Pascal o Ada, que en muchos casos ha caído también en desuso.

3.6.1 Conic

Conic [KM85, MKS89, KM90] es uno de esos sistemas legendarios cuya vigencia pervive mucho tiempo después de ser abandonados por sus propios creadores. Éstos fueron el núcleo original de la sección de Ingeniería de Software Distribuida del Imperial College de Londres: Jeff Kramer, Jeff Magee y Morris Sloman. Se podría decir que fue el *primer* sistema distribuido con un Lenguaje de Configuración propio. El mismo concepto de Lenguaje de Configuración fue definido en su contexto, así como la descripción de las cuatro operaciones básicas (§3.4.2), y algunos de los problemas más famosos del campo [KM90].

El componente básico de Conic es el *tipo de módulo de tarea*, que consiste en la definición de una tarea secuencial y autocontenida, a partir de la cual el sistema genera las correspondientes instancias, que se comunican mediante el intercambio de mensajes. Éstos son emitidos a través de *puertos* fuertemente tipados, que reciben el nombre de *exitport* para las salidas (emisores) y *entryport* para las entradas (receptores). El interfaz del módulo se define siempre en función de estos elementos sintácticos, y la especificación de su funcionalidad se realiza en el interior de estos módulos, en un lenguaje propio inspirado en Pascal.

```
system sala;
    create
        cama1: paciente at maquina1;
        enfermera: ats at maquina2;
    link
        cama1: alarma to enfermera.alarma[1];
        enfermera.cama[1] to cama1.cama;
end.
```

```
manage sala;
    create
        cama2: paciente at maquina1;
    unlink
        cama1: alarma from enfermera.alarma[1];
        enfermera.cama[1] from cama1.cama;
    link
        cama2: alarma to enfermera.alarma[1];
        enfermera.cama[1] to cama2.cama;
end.
```

Figura 3.1: El clásico Sistema de Monitorización de Pacientes [KM85]

Los módulos ignoran por completo con quién están conectados en un momento dado, ya que de eso se ocupa el propio Conic, de manera externa a su definición. Los *enlaces* entre los puertos se llevan a cabo mediante un mecanismo implícito de paso de mensajes. Se permiten interacciones asíncronas unidireccionales, o síncronas bidireccionales. Por tanto, se separan los aspectos computacionales, indicados en el dialecto de Pascal, de los aspectos de interacción, definidos en Conic propiamente dicho. Así se define el comportamiento de Conic como un Lenguaje de Configuración, en el que los módulos que manipula son completamente reutilizables, ya que no tienen referencias directas a ningún otro módulo.

El sistema se especifica de manera constructiva, tal como se ve en la Figura 3.1, en el bloque **system**. Los componentes se crean uno a uno, y a continuación se definen los enlaces entre ellos, generándose una configuración completa. Esta configuración es almacenada en un elemento externo, implícito, denominado el *gestor de configuraciones (configuration manager)*, que existía como tal en la implementación real de Conic. Este gestor será, eventualmente, el encargado de realizar las correspondientes *reconfiguraciones*, en respuesta a una secuencia de instrucciones, que se agrupan en el bloque **manage**, tal como se puede ver en la Figura. Estas instrucciones son precisamente las cuatro operaciones básicas, que tal como se ha dicho (§3.4.2) fueron definidas precisamente en este contexto.

Aunque Conic alcanzó una gran popularidad, terminó siendo abandonado por sus propios diseñadores, que intentaron renovar el concepto especificando la configuración de forma *declarativa*, en lugar de *imperativa*. Esto terminaría dando lugar al diseño de *δarwin* (§3.7.2).

3.6.2 Durra

Durra [BDWL91] fue diseñado a principios de los 90 en el SEI, como pieza integral de una metodología que combinaba aspectos de especificación, modelado y prototipado, coordinando a múltiples grupos de desarrollo que podrían estar utilizando, incluso, diferentes notaciones. El dominio de aplicación principal eran los sistemas de tiempo real, lo cual se percibe claramente en el diseño del lenguaje, y en particular en el sublenguaje de especificación del comportamiento.

Durra se define como un *lenguaje de descripción de tareas*, cuyo objetivo es facilitar el desarrollo de aplicaciones *distribuidas* implementadas por tareas de grano grueso [BDWL91]. Se concibe como un lenguaje no procedural, completamente separado de cualquiera de los lenguajes

de programación específicos que se utilicen para definir las tareas, típicamente Ada[2]. Durra fue concebido como un lenguaje de integración al nivel de lo que Barbacci denominaba la *Arquitectura Técnica*, y que definía como "una arquitectura de software aumentada con la descripción formal del comportamiento de las tareas" [BDWL91]. Es decir, no sólo se describe la estructura del sistema, sino que se llega a especificar el modo en que se comporta cada uno de los componentes, *respecto de su papel en la arquitectura*. Algo muy similar a lo que posteriormente sería la especificación de los protocolos en los puertos y roles de los componentes y conectores de Wright, aunque utilizando un formalismo diferente.

task *nombre_tarea (parámetros)*
  ports
    declaración_puerto
  behavior
    especificación_comportamiento
  attributes
    pares_atributo-valor
  components
    declaración_subcomponentes
  structures
    conexiones_subcomponentes
  reconfigurations
    pares_condición-acción
  clusters
    asociaciones_cluster-componente
end *nombre_tarea*

Figura 3.2: Estructura básica de un componente (tarea) en Durra

En una especificación Durra hay dos tipos de elementos: *tareas (***task***)* y *canales (***channel***)*. Es fácil ver que se corresponden, respectivamente, con los componentes y conectores de un ADL actual, a pesar de haber sido definidos con anterioridad. La estructura de una tarea puede verse en la Figura 3.2: tiene puertos por los que realiza interacciones, la especificación de comportamiento mencionada, y una serie de atributos que permiten clasificarlas y ordenarlas en un repositorio. Además, puede estar compuesta por sub-componentes unidos en una serie de estructuras internas, y se puede alterar la topología de esas estructuras mediante reconfiguraciones predefinidas. El número de *cluster* hace referencia a la localización física de la tarea, algo que depende de la implementación concreta. La sintaxis de un canal es idéntica, pero sin las secciones de componentes, estructuras y reconfiguraciones, ya que los canales son siempre primitivos.

La comunicación entre las tareas de Durra se realiza siempre a través de un canal. Un sistema se organiza como una tarea compuesta (es decir, una jerarquía de componentes), en el que los puertos de las tareas se conectan a los de los canales, y cada canal transporta esta información del modo indicado en su comportamiento (por defecto, actúa como un bus común).

Durra es particularmente notable por mostrar, en una fecha tan temprana como 1991, una arquitectura completa, con jerarquías de componentes, y capacidad para reconfiguración *basada en eventos*. Los eventos son generados explícitamente como *señales* en el lenguaje de *expresiones temporales* de la sección de comportamiento, y son capturados por la sección de reconfiguración, donde desencadenan una serie de actividades (típicamente, la activación de una tarea, o de

[2]La implementación de Durra generaba código Ada, y es evidente su influencia en el diseño de la sintaxis. No obstante, es cierto que la especificación resulta completamente independiente de éste o cualquier otro lenguaje.

una secuencia de tareas). Aunque las secuencias de tareas están predefinidas, dependen de condiciones concretas, y por tanto son distintas en cada momento. Es decir, constituyen un ejemplo de *dinamismo*.

Durra es algo más que un precedente de los ADLs modernos; podría decirse que es incluso superior a algunos de ellos. Por otra parte, se puede advertir una filosofía muy similar en Wright (§3.7.1). Sin embargo, la especificación de los protocolos de los puertos es mucho más formal en este último, pero mucho más flexible (y dinámica) en Durra.

3.7 Basados en Álgebras de Procesos

En este apartado se revisarán una serie de ADLs muy conocidos, que constituyen buena parte de lo que hoy en día se entiende por Descripción Arquitectónica. El nombre de este grupo se justifica en la medida en que todos ellos se relacionan, de un modo u otro, con un álgebra de procesos, en especial desde un punto de vista semántico; por este motivo, a lo largo de esta tesis recibirán con cierta frecuencia el nombre de *"ADLs algebraicos"*, a menudo incluyendo también a otros sistemas no dinámicos, como Unicon [SDK+95a] o PADL [BCD01].

El primero de los lenguajes revisados, Wright, es también el ejemplo clásico de un ADL de este tipo, aunque es precisamente uno de los que muestra menores capacidades dinámicas. Su fundamento formal es el clásico CSP, lo que se muestra de manera explícita en la propia sintaxis del lenguaje. Otros sistemas tratados, como Armani, HOT ACME o AML, no tienen un fundamento explícitamente algebraico, y aparecen aquí únicamente por su vinculación con Wright. Por otra parte, El fundamento de los sistemas restantes es el cálculo-π, un álgebra aún más potente: en el caso de LEDA, está tan íntimamente ligado al lenguaje como en el caso de Wright; en cambio, el origen de *δarwin* es anterior, y su relación con este formalismo deriva únicamente de la formalización de su semántica.

3.7.1 Wright Dinámico

El lenguaje Wright [All97] es uno de los ADLs más conocidos, y posiblemente el que tiene la sintaxis más formal de todos los existentes. También es el más estático de todos ellos, como se ha señalado en multitud de ocasiones [CFBS99]. Plantear la presencia de este lenguaje en un Capítulo sobre Arquitectura Dinámica parece casi una contradicción, pero en efecto existe una propuesta para una versión de Wright Dinámico [ADG97, ADG98].

Una descripción Wright convencional consiste en una *configuración* compuesta por una serie de *componentes*, cuyo interfaz está segmentado en *puertos*. Los distintos puertos se relacionan entre sí, internamente, a través de la *computación* definida por el componente. Por su parte, la comunicación entre componentes requiere la existencia de *conectores*, cuyo interfaz se segmenta en *roles*, relacionados entre sí mediante un *pegamento (glue)*. La combinación de puertos y computación especifica las posibilidades de comunicación de un componente; la de roles y pegamento, el protocolo proporcionado por un conector. La unión del puerto y el rol adecuado recibe el nombre de *adjunción (attachment)* y es lo que hace posible la comunicación. Todos estos detalles son descritos con el lenguaje CSP, un álgebra de procesos [Hoa85]. Además, Wright es uno de los pocos lenguajes que admite también la especificación de estilos, mediante el uso de una variante de la Lógica de Predicados (§2.4.5).

La descripción de Wright Dinámico es en principio completamente idéntica, y tan sólo añade un detalle, que es el que sigue: en todo sistema dinámico se asume la existencia de un componente especial denominado *Configuror*, que se encargará de realizar las reconfiguraciones requeridas por el sistema. Para poder hacerlo, el lenguaje es ampliado con las cuatro operaciones básicas de la reconfiguración (§3.4.2), que aquí reciben los nombres de *new, del, attach* y *detach*, y cuya correspondencia con las originales resulta evidente. Ha de tenerse en cuenta que estas operaciones sólo pueden usarse desde el *Configuror*, ya que éste elemento será el que contenga la especificación completa de los aspectos dinámicos del sistema.

La Figura 3.3 proporciona parte de la definición de un Configuror (o más bien de la definición *del* Configuror, pues sólo se conoce un ejemplo) para un sistema Cliente-Servidor Tolerante a Fallos. En él se dispone de dos servidores: el primario (*Primary*) es rápido, pero no es totalmente seguro; el secundario (*Secondary*) es más lento, y se utiliza para dar respaldo al primero. El fragmento de la Figura muestra cómo un nuevo Cliente, al llegar, se une al Servidor primario. La parte que no se muestra proporciona la definición del proceso *WaitforDown*, en el que se detecta una caída del Servidor primario, y se transfieren *todos* sus enlaces al secundario, con el fin de proporcionar una mínima Tolerancia a Fallos. Cuando el primario vuelve a estar activo, el secundario le "devuelve" toda la carga que había tomado. En ningún momento, por supuesto, se intenta hacer ningún tipo de equilibrio de cargas.

Configuror DynamicClient-Server
  Style Fault-Tolerant-Client-Server
    new.C:Client
    → new.Primary : FlakyServer
    → new.Secondary : SlowServer
    → new.L : FTLink
    → attach.C.p.to.L.c
    → attach.Primary.p.to.L.s → WaitForDown
    where ...

Figura 3.3: Fragmento de un *Configuror* de Wright Dinámico [ADG98]

Dado que el objetivo de Wright es traducir una descripción arquitectónica a una especificación formal en Csp [AG94a, AG94b, AG94c, All97], la introducción de dinamismo en el lenguaje plantea un problema. Csp es un álgebra de procesos clásica, y sólo es capaz de describir configuraciones *estáticas* de procesos. No tiene soporte para movilidad, por lo que, a diferencia de lo que ocurre en los cálculos nominales –véase al respecto el Apéndice B–, no es posible crear nuevos procesos y canales de comunicación. ¿Cómo se puede entonces proporcionar una semántica coherente para las nuevas acciones *attach, new*, etc.?

Con el fin de lograrlo, se introducen dos ideas. La primera es que se restringen los sistemas que se pueden representar: *solamente se admiten aquellos para los que el número de configuraciones posibles es finito*, aunque pueda ser grande. La segunda es que en la traducción de Wright a Csp se añade una *etiqueta* a todos los eventos, que expresa en qué configuración ocurren los mismos. La restricción de finitud se da, precisamente, para poder disponer de un número finito de estas etiquetas. Por tanto, el efecto de una acción de reconfiguración consiste únicamente

en la selección de un subconjunto de las expresiones de Csp: las que estén señaladas con la etiqueta adecuada. De este modo, en cada momento se producen solamente las acciones de una configuración concreta, ya que aquellas que tienen una etiqueta diferente no se seleccionarán nunca de manera simultánea, y por tanto no pueden llegar a interactuar.

Así pues, un evento e, que se comunica desde el puerto p del componente C, hasta el rol r del conector N es manejado por el sistema como el evento $C.p.e.N.r$; y un componente es identificado por su nombre, seguido de la especificación de la conexión de *todos* sus puertos, de modo que se pueda saber en qué configuración concreta se encuentra. Es decir, el componente C será, en un momento dado, el $C.go.p_1.N_1.r_1 \ldots p_n.N_n.r_n$, donde los distintos p_i son sus puertos, y los r_i los roles de los conectores N_i a los que éstos están conectados.

Por tanto, las operaciones *new, del, attach* y *detach* tan sólo cambian el etiquetado; un componente no se crea, sino que se le da la etiqueta adecuada; ni tampoco se destruye, sino que simplemente se le da un prefijo diferente. En definitiva, la secuencia siguiente, que aparentemente es dinámica,

$$new.L \rightarrow attach.C.p.to.L.c \rightarrow attach.Primary.p.to.L.s \rightarrow \ldots$$

en realidad no pasa de la fase de traducción, y se convierte en la definición estática de un componente, que se especifica como:

$$L.go.c.C.p.s.Primary.p \rightarrow \ldots$$

Lo cierto es que es un método realmente complicado, y sigue siendo muy limitado. Se restringe excesivamente el número de posibilidades, se complica innecesariamente la especificación, y se pierde gran parte de la elegancia conceptual del Wright original. Más aún, la semántica basada en etiquetas resulta mucho más complicada, en realidad, que la del propio álgebra de procesos. Además, provoca una explosión combinatoria que hace dudar de que sea realmente viable en sistemas de más de tres componentes, como éste.

En realidad, el método es casi idéntico a una propuesta que Dean y Cordy tan sólo esbozaban [DC95] (§3.9.1), y en el que también se habían de conocer de antemano todas las configuraciones. Se cambia el papel de un autómata finito externo, que se encargase de realizar las transiciones, por el del *configuror* interno, que expresa los cambios de estado con el mismo formalismo (Csp) que se usa para especificar las interacciones en el sistema.

Lo cierto es que los propios autores parecen haber reconocido la falta de interés de este sistema. Nunca han descrito otro ejemplo que el del sistema Cliente-Servidor tolerante a fallos, y no han vuelto a escribir sobre el tema desde 1998 [ADG98]. Ni siquiera han desarrollado su propia sugerencia acerca de que este método podría ser mejorado mediante la adición de propiedades, descritas en alguna Lógica Temporal, con toda probabilidad CTL*.

3.7.2 Darwin

Existen, en líneas generales, tres versiones del Adl conocido como *δarwin*. La primera, denominada simplemente Darwin, nació como una evolución del ya mencionado Conic, durante el desarrollo del proyecto ESPRIT Rex. Fue elaborada en la primera mitad de los años 90 por Naranker Dulay [Dul90], un miembro del grupo de Kramer, Sloman y Magee.

Aún más importante es la segunda, conocida como *δarwin* 2. En el plano sintáctico es tan sólo un refinamiento de la anterior, pero supuso una reelaboración completa en el plano semántico. En esta etapa el lenguaje fue formalizado en cálculo-π [MDEK95], y analizado en contextos concurrentes [Che94]; se elaboraron varias herramientas visuales [Ng92] e interactivas [CDF+95, Fos97], así como un sistema distribuido completo, denominado Regis [MKS94, Cra97]. Incluso se desarrolló una implementación compatible con CORBA [CD97], por entonces más eficiente que algunas comerciales [Cra97]. Ésta será, por defecto, la versión a la que se hará referencia siempre que se hable del ADL, ya que es la que ha sido utilizada con mayor frecuencia en trabajos relacionados con la Arquitectura de Software [KM96b, MK96]. Su autor principal fue de nuevo Naranker Dulay, con notorias aportaciones de John Stephen Crane.

La tercera versión, que podría denominarse *δarwin* 3, fue elaborada dentro del proyecto Tracta, y es debida mayoritariamente a Dimitra Giannakopoulou [Gia99]. Se realiza una simplificación notable de la parte estructural, para combinarla después con un modelo de comportamiento completamente nuevo, basado en Sistemas de Transiciones Etiquetadas. En esta versión se fundamentan algunos trabajos posteriores [MKG99, Pry00], entre los que cabe destacar el magnífico estudio de Moazami-Goudarzi sobre la consistencia frente al cambio [MG99], que continúa la línea iniciada por Jeff Kramer una década antes [KM90].

Por otra parte, existe también una variante de *δarwin* llamada Koala [vOvdLKM00], que es empleada exclusivamente como herramienta de desarrollo en el interior de la compañía Philips. Aunque tiene sus propias peculiaridades, ya que ha sido especialmente adaptado a la descripción de líneas y familias de productos [JRvdL00], en líneas generales es prácticamente el mismo lenguaje, por lo que no se abundará más en este punto.

δarwin ha sido definido como un lenguaje Minimal Orientado a Objetos, Concurrente, de Configuración, de Coordinación, de Composición y, por supuesto, de Descripción de Arquitecturas. Es uno de los vértices sobre los que se define el propio campo, y representa a una visión muy concreta de sus conceptos que resulta, por ejemplo, completamente opuesta a la de Wright (§3.7.1). De hecho, estos dos lenguajes han sido durante años los ejemplos por excelencia de la visión de la arquitectura de software orientada al objeto –*δarwin*– frente a la arquitectura de software orientada al proceso –Wright– [BF97].

```
component filtro {
    provide prev <port,int>;
    require next <port,int>,
            output <port,int>;
}

component pipeline (int n) {
    provide input;
    require output;
    array F[n]:filtro;
    forall k:0..n-1 {
        inst F[k];
        bind F[k].output -- output;
        when k < n-1 bind
            F[k].next -- F[k+1].prev;
    }
bind
    input -- F[0].prev;
    F[n-1].next -- output;
}
```

Figura 3.4: Una Tubería Extensible en *δarwin* [MK96]

δarwin admite dos notaciones, una textual y otra gráfica. La segunda es más intuitiva, mientras que la primera es, obviamente, mucho más potente. Aquí se considerará exclusivamente esta última. Sintácticamente, todo en *δarwin* es un componente. De hecho, Jeff Kramer es uno de los principales detractores del concepto de conector. Según su visión, tener dos tipos de elementos altera la estructura del sistema, y fuerza a un estilo de descripción en ocasiones poco natural. Las especificaciones de protocolos se indicarán al nivel de un portal, o en un componente específico si resultan demasiado complejas.

Los componentes pueden ser primitivos o compuestos; en el primer caso se corresponden directamente con una implementación. En el segundo, se construyen jerárquicamente a partir de otros componentes. Dado que no hay conectores, tampoco se definen *configuraciones*; el conjunto total de la arquitectura se ve simplemente como un componente compuesto.

Dentro de un componente jerárquico se realizan dos tipos de enlaces (*bindings*): entre dos subcomponentes (instancias) del mismo nivel, y entre distintos niveles (enlaces jerárquicos). Se facilita en lo posible la definición de patrones, por lo que se admite la parametrización de los componentes, así como el uso de bucles (**forall**), condicionales (**when**), estructuras de tipo array e incluso especificaciones recursivas.

Todo componente tiene unos puntos de interacción definidos, en todo equivalentes al concepto de *puerto*. De hecho, la versión original de *δarwin* los designaba precisamente con este término. Ahora se denominan *portales*, y en la *vista de comportamiento* (ver abajo) su descripción es arbitraria. En cambio, en la versión tradicional siempre han podido ser de dos clases, en las que se definen como asimétricos de manera intencionada: se trata de los *servicios* (**provide**) y los *requisitos* (**require**). Todo enlace en *δarwin* va desde un requisito hasta un servicio, aunque una vez establecido es completamente bidireccional. Los requisitos expresan solicitudes, mientras que los servicios *proporcionan* la respuesta a esas solicitudes; por tanto, se admite una cardinalidad de $1 : N$. Los portales tienen además asociado un *tipo*, que indica el esquema de interacción que son capaces de realizar. Los tres básicos son el *puerto (port)*, el *flujo (stream)* y el *evento*, aunque es perfectamente posible añadir otros nuevos.

El dinamismo de *δarwin* se manifiesta con claridad en los dos usos de la palabra clave **dyn**, que define declarativamente dos tipos de reconfiguración: la *instanciación perezosa* y la *instanciación dinámica* [MK96, MDEK95]. En el primer caso, un componente es declarado normalmente con su nombre; pero no se *crea* realmente hasta el momento en que es utilizado. En el segundo caso, ni siquiera se da nombre al componente, al que sólo se alude mediante su tipo. Se crea dinámicamente en el momento en que otro componente intenta activar uno de sus enlaces; y él mismo genera un nuevo enlace, desde el que tomará toda iniciativa de comunicación en adelante –los demás no pueden invocarlo, ya que no conocen un nombre con el que hacerlo–. En definitiva, el dinamismo de *δarwin* consiste en un tipo de *reconfiguración restringida*, que expresa unos patrones fijos, pero muy potentes, en los que se admite la creación de componentes y enlaces. Por otra parte, la evolución dinámica de los enlaces produce de manera habitual alguna reconfiguración trivial (al respecto, revisar el enfoque basado en cálculo-π, que se describe más abajo). Con todo, su versatilidad sigue siendo inferior a la de Conic.

Aparte de esto, en [KM96b] se propone la definición de *arquitecturas auto-organizadas*, en las que una serie de restricciones globales (replicadas localmente en cada componente) controlan la evolución del sistema, sin que tenga que haber un elemento específico que las aplique. Este enfoque, muy citado en la Bibliografía, no ha sido sin embargo explorado desde entonces.

Uno de los aspectos más interesantes de *δarwin* ha sido la especificación formal de la semántica de sus enlaces en cálculo-π. Esto abarca tanto a los enlaces "estáticos" como a las reconfiguraciones dinámicas citadas [MDEK95, MK96, Cra97]. Como muestra, se indica en la Figura 3.5 la semántica de un requisito, un servicio y un enlace simple, y se indica cómo interaccionan. Tras una evolución inicial (el establecimiento del protocolo), puede comprobarse que los distintos procesos quedan ligados (cada uno de ellos conoce ya a los canales del otro), mientras el servicio (Prov) mantiene la capacidad de seguir atendiendo peticiones.

$\mathrm{Prov}(p,s) \stackrel{\mathrm{def}}{=} !(p(x).\bar{x}s)$	$\mathrm{Req}(r,o) \stackrel{\mathrm{def}}{=} r(y).\bar{y}o$	$\mathrm{Bind}(r,p) \stackrel{\mathrm{def}}{=} \bar{r}p$
Semántica de un servicio	**Semántica de un requisito**	**Semántica de un enlace**

$$(\,\mathrm{Req}(r,o) \,|\, \mathrm{Bind}(r,p) \,|\, \mathrm{Prov}(p,s)\,) \equiv \ldots \xrightarrow{r} \ldots \xrightarrow{p} \underline{\underline{\bar{o}s \,|\, \mathrm{Prov}(p,s)}}$$

Figura 3.5: Establecimiento de un Enlace *δarwin* en cálculo-π

Como ya se ha dicho, la última versión de *δarwin* parece haber derivado más hacia la parte de especificación. A la perspectiva de este nuevo enfoque se le denomina *vista de comportamiento*, y a la tradicional, *vista de construcción*. La descripción de los protocolos ha abandonado parcialmente el cálculo-π en favor de los sistemas de transiciones etiquetadas (LTS), que facilitan la construcción de herramientas de verificación, como el *Labelled Transition Systems Analyzer (LTSA)* [MKG97, Gia99]. Su mayor problema, la explosión de estados, se controla mediante una técnica desarrollada por Jeff Kramer y Shing-Chi Cheung, el análisis de alcanzabilidad composicional (CRA). Por su parte, Kaveh Moazami-Goudarzi ha utilizado esta técnica para comprobar los estados del sistema antes y después de una reconfiguración [GK96, MG99]. Aunque el sistema de LTS tiene la ventaja de su simplicidad, parece no obstante demasiado limitado para llegar a plantearse en sistemas de cierto tamaño.

En resumen, *δarwin* es uno de los ADLs más potentes y flexibles. Se puede utilizar en varios niveles de descripción, y a pesar de no ser formal, tiene una semántica formalizada. Sus mayores defectos son precisamente su relación ambigua con los aspectos formales, su habitual dependencia de las distintas versiones de Regis, y el hecho de tener un dinamismo limitado, aunque considerable, en su versión más declarativa.

3.7.3 LEDA

LEDA es uno de los ADLs más recientes y avanzados que se incluyen en esta revisión, y tal vez el que alcanza mayores cotas de dinamismo. Su autor principal ha sido Carlos Canal, de la Universidad de Málaga, que ha desarrollado tanto sus aspectos sintácticos como los semánticos [CPT99, CPT01, CV00], incluyendo las nociones formales necesarias.

LEDA es lo que Wright podría haber sido, si en lugar de CSP hubiera utilizado cálculo-π como su formalismo base. El supuesto de partida es muy similar: cómo construir un ADL cuyas descripciones sean luego directamente traducibles a una especificación formal completa. De nuevo, se tiene la arquitectura descrita como un conjunto de componentes cuya interacción

se indica directamente de manera formal, pero en este caso se utiliza cálculo-π. El hecho de seleccionar un álgebra de procesos mucho más potente permite obtener un lenguaje mucho más potente, con enormes capacidades dinámicas.

En LEDA, la arquitectura se expresa como un conjunto de *componentes*, cuyo interfaz se segmenta en *roles*. La estructura de cada rol se especifica (*spec*) como una descripción en cálculo-π, en la que los nombres hacen referencia a los elementos del componente (sus roles) o son creados como variables auxiliares (*names*). Como es habitual, un componente puede ser atómico o compuesto; y puede estar también parametrizado. Cada componente se conecta con otros componentes por medio de *adjunciones (attachment)* directas; esto es, en LEDA no existen conectores. La Figura 3.6 muestra la estructura de un sistema Cliente-Servidor sencillo que utiliza parte de las características descritas.

```
component Client {
    interface
        request: Request(request){
        spec is
        (reply)request!(reply).
        reply?(service).Request(request);
    }
composition
    service[]:any
}

component Server {
    interface
        serve:Serve(request){
        spec is
        request?(reply).
         (new service)reply!(service).
        n++.Serve(request);
    }
composition
    service[]:any
}

component ReconfigurableClientServer{
    interface none;
    composition
        client: Client;
        server[2]: Server;
    attachments
        client.request(r) <>
            if ( server[1].n <= server[2].n )
            then server[1].serve(r)
            else server[2].serve(r);
}
```

Figura 3.6: Un Sistema Cliente-Servidor con Equilibrio de Carga en LEDA

LEDA define tres tipos de adjunciones: las *estáticas* (<>) son idénticas a las existentes en otros lenguajes. Las *reconfigurables* admiten la especificación de varias posibilidades, que se indican mediante un **if** –véase, por ejemplo, la Figura–. Finalmente, las *múltiples* (>>) describen patrones de interacción entre arrays de componentes, y pueden ser privadas, para uso de un par determinado, o compartidas por todos los participantes de la interacción.

El lenguaje admite la *extensión* de componentes y roles con posterioridad a su definición (*adding*), así como la definición de múltiples niveles de abstracción, vistos como un mecanismo de subtipado; existe pues una forma de herencia (*extends*) y un tipo básico neutral (*any*). Éste es uno de los aspectos más notables del lenguaje, en particular por su tratamiento formal [CPT01], que lo vincula a una noción de *compatibilidad*. De hecho, se puede afirmar que LEDA tiene el desarrollo de subtipado más completo del campo, complementario a los estudios de Rosenblum sobre C2SADEL [MRT98] e incluso superior al esquema de Rapide [KLM94].

Finalmente, se admite la construcción de un componente como envolvente (§4.6) de otro ya existente, de modo que captura las interacciones de sus roles sin necesidad de redefinirlos como

propios; estas construcciones se denominan *adaptadores*.

No se dispone aquí de operaciones explícitas de reconfiguración, ya que no son necesarias en absoluto. Las manipulaciones de los enlaces son perfectamente naturales como paso de nombres en cálculo-π. Nuevos componentes (procesos) pueden ser creados mediante el operador de restricción (ν)[3]; la destrucción de componentes y enlaces sucede de forma natural al salir los nombres del ámbito del proceso. LEDA, como la mayoría de los sistemas basados en cálculo-π, y en especial el lenguaje PICT, utiliza una sintaxis tipo Occam, en lugar de la notación algebraica original. Esto es preciso para poder utilizar el lenguaje sobre un teclado convencional, y de hecho ocurrirá lo mismo en el desarrollo de $\mathcal{P}i\mathcal{L}ar$ (§6.6).

La potencia expresiva de LEDA es enorme, y no ha sido aún explorada hasta sus últimas consecuencias. El hecho de utilizar cálculo-π dota ya de por sí al lenguaje de una enorme flexibilidad, y de gran capacidad para el dinamismo. Las características propias del lenguaje, tales como el tratamiento de subtipado, la definición de adaptadores, o los distintos tipos de adjunciones, se constituyen pues como un valor añadido. En definitiva, se trata de uno de los lenguajes más elegantes del campo, y sin duda uno de los que muestran mayor potencial para la especificación de dinamismo.

3.7.4 Derivados de ACME

El lenguaje ACME [GMW97] fue diseñado originalmente no como un ADL, sino como un *Lenguaje de Intercambio Arquitectónico* que sirviese de vehículo intermedio entre múltiples lenguajes. Sin embargo, existe cierta tendencia a utilizarlo también como un ADL por sí mismo.

Se pretendía que el lenguaje proporcionase el mínimo común denominador necesario para expresar todas descripciones arquitectónicas. Se define pues el lenguaje como un vocabulario *no interpretado* de siete elementos: *componentes*, *conectores*, *puertos*, *roles*, *sistemas*, *representaciones*, y *mapas de representación (rep-maps)*[4]. Estos elementos se corresponden con los análogos de otros ADLs. Los sistemas son configuraciones, una representación es la descomposición jerárquica de un componente o conector, y un *rep-map* expresa la correspondencia entre las interfaces de un componente y una representación suya. Por supuesto, existen también, y definidos de manera independiente, *enlaces (attachments)* entre los puertos y roles de los componentes y conectores, respectivamente. También pueden definirse plantillas (*templates*) de componentes, que se resuelven de manera estática.

Los elementos de los siete tipos citados pueden tener asociadas una serie de especificaciones, que se denominan *propiedades*, y que pueden ser indicadas en cualquier lenguaje, ya que ACME tan sólo define lo que se denomina un *marco semántico abierto*. Este marco sólo supone, en última instancia, que las propiedades se expresarán en términos de Lógica de Primer Orden y relaciones predefinidas, y como pares de asignaciones atributo-valor. Sin embargo, no se proporciona una definición cerrada para las mismas.

ACME se concibe, ante todo, como una enumeración de los conceptos básicos del campo. Se ha utilizado, por ejemplo, para traducir descripciones en Wright, Aesop, C2, MetaH, Rapide o

[3]En la Figura puede verse el uso de este operador, expresado en sintaxis PICT como **new**. No se trata, por tanto, de una primitiva de reconfiguración, sino de un operador normal de cálculo-π.

[4]Es decir, se proporcionan de manera puramente sintáctica las características de lenguajes como Aesop o Wright, lo que no parece corresponderse demasiado con el objetivo de actuar como un lenguaje neutral.

SADL. Sin embargo, cada vez se insiste más en utilizarlo para describir arquitecturas *directamente*, en lugar de simplemente intercambiarlas. Es en este sentido en el que se deben entender los desarrollos siguientes, que han afirmado en algún momento ser dinámicos.

Armani

Bob Monroe ha desarrollado Armani [Mon98] como una extensión de Acme que no pretende describir solamente la Arquitectura, sino abarcar la fase de Diseño en su totalidad.

La aportación más interesante del lenguaje es la adición de un Lenguaje de Predicados, que permite expresar las restricciones de la arquitectura y las heurísticas de diseño, de un modo más claro que el que utilizaba Acme. Se trata, en suma, de una variante de la Lógica de Predicados, a la que se proporcionan una serie de funciones predefinidas sobre tipos, conjuntos, parentescos y conectividad de grafos. A esto se añaden una serie de operadores y los clásicos cuantificadores (**forall** y **exists**), así como la posibilidad de usar definir funciones propias de usuario (**analysis**), y de especificar invariantes (**invariant**). El lenguaje está definido de manera semi-formal, y se inspira de manera reconocida en el OCL de Uml [WK99].

Se ha dicho que las restricciones de Armani, basadas en este lenguaje de predicados, podrían llegar a proporcionar cierto grado de dinamismo. Sin embargo, el único intento conocido de este tipo fue la traducción del bus de eventos Jedi a Armani, realizada por Elisabetta di Nitto en [DNR99]. Resultó finalmente en un fracaso, ya que eventualmente se llegó a una definición recursiva, lo que es explícitamente prohibido por el lenguaje.

Hot Acme

Hot Acme fue un desarrollo realizado en Yale por John Peterson y Martin Sulzmann, bajo la dirección de Paul Hudak [PS98, PS99]. Consiste la integración de Acme en un sistema de tipos de orden superior –precisamente, Hot significa *Higher-Order Typed*–. No se trata aquí de proporcionar soporte para polimorfismo en el lenguaje, sino de poder traducir una descripción Acme a una especificación verificable por el sistema HM(X), un comprobador automático de tipos al estilo Hindley-Milner basado en Haskell.

En este sistema se parte de la descripción de un determinado elemento arquitectónico, esto es, un componente o un puerto, *con sus propiedades*, definidas del modo usual en Acme. Ésta es tratada como la definición de un tipo polimórfico, y de igual modo se hace con todos los elementos de una configuración. Entonces, el mecanismo de comprobación de tipos es capaz de determinar si la misma es consistente.

Este tipo de verificación puede aplicarse a cualquier aspecto que se desee, y Peterson y Sulzmann plantean explícitamente la definición de arquitecturas dinámicas. Para ello, añaden al lenguaje una construcción nueva, la *abstracción de arquitectura*, esto es, la definición de una estructura parametrizada sobre un conjunto de componentes concreto.

```
Architecture nombre (c1, ..., cn) = { body }
```

Lo notable es que esta abstracción puede ser manejada sin que las referencias a los componentes hayan sido especificadas completamente. Esto significa que los parámetros terminan siendo inferidos por el sistema de tipos *en tiempo de ejecución*, esto es, se crean nuevos enlaces

de manera dinámica. La relación entre los tipos de los argumentos inicialmente desconocidos permite deducir la topología de la arquitectura, una vez que ésta ha sido instanciada.

Lamentablemente, este enfoque parece haber sido completamente abandonado, y no ha conocido mayor desarrollo tras la marcha de Martin Sulzmann de Yale.

3.7.5 AML

AML fue diseñado por David Wile [Wil99] como un Lenguaje de Intercambio alternativo a ACME, del que fue también uno de los autores originales. Se plantea como un *metalenguaje* neutral en el que poder describir las características de los distintos ADLs, para poder luego realizar los intercambios de descripciones sobre una base común. De ahí la denominación de AML (**A***rchitectural* **M***eta*-**L***anguage*), y el hecho de que nunca se haya planteado para un uso directo.

Se toma como base del lenguaje a uno de los modelos más difundidos y mejor comprendidos de cuantos existen: el Modelo Entidad-Relación. En el contexto que éste define, AML intenta representar arquitecturas de la manera más básica, como un conjunto de elementos que mantienen una serie de relaciones topológicas que son estrictamente descritas y restringidas. Aparte de esto, y de plantear otras relaciones no-topológicas, como las temporales o las específicas de dominio, no se pretende llegar más lejos.

El concepto fundamental del lenguaje es el de *relación (relationship)*, que puede ser topológica o de dominio, y a la que se da un nombre y unos parámetros. Esta relación va asociada a una serie de *suposiciones (assume)* que expresan sus propiedades en una notación basada en la Lógica de Primer Orden, y en la que las propias relaciones son predicados. Estas relaciones se definen entre *elementos (element)* concretos, que se equiparan con una constante de Skolem, y quedan identificados por definición, es decir: se trata de instancias, no de tipos. Las restricciones sobre estas configuraciones vienen dadas por las suposiciones ya citadas (que han de ser necesariamente ciertas) y por *derivaciones (derive)*, que han de poder ser deducidas en un momento dado del sistema, *dinámicamente*. Para poder definir elementos genéricos, que permitan expresar relaciones genéricas, se utiliza la construcción *kind*, que se define como una combinación de un tipo de componente y un estilo arquitectónico. Esta es, sin duda, la construcción más potente del lenguaje, y permite la creación de nuevos tipos definidos por usuario, que pueden luego utilizarse como si de palabras clave se tratase, en cualquier otro lugar de la descripción. El ejemplo de la Figura 3.7 muestra cómo varios tipos pueden ser definidos y utilizados en la misma especificación. Se trata de un ejemplo particularmente interesante, en el que se describe la totalidad del estilo C2 (§3.8.2) en apenas unas líneas. A partir de este momento, se podría traducir cualquier especificación de C2SADEL a AML sin mayor complicación [Wil99].

Wile dice explícitamente que uno de los objetivos de AML es la especificación de dinamismo. Éste se introduce a través de las restricciones denominadas *derivaciones*, ya citadas, cuyo valor de verdad no se asegura, sino que ha de ser deducido a partir del sistema. En estas restricciones se admite el uso de *lógica temporal*, con cuantificaciones exactas de tiempo y modalidades del tipo de *alguna vez (sometimes)* y *siempre (always)*. Este camino aún tiene que ser explorado con mayor detenimiento, ya que por el momento sólo se dispone de una breve mención al mismo.

```
kind C2-system
    kinds m-top, m-bottom, n-top, n-bottom;
        relationship C2-element(x)
        == component(x) | connector(x);
        relationship part-of(1..1 C2-system, C2-element);
        relationship attaches-at(t: C2-element, b: C2-element);
        kind component
            attaches-at ! { 0..1 m-top t; 0..1 m-bottom b };
        kind connector
            attaches-at ! { m-top t; n-bottom b };
        relationship attaches-at(element, p: connector);
```

Figura 3.7: Descripción del Estilo C2 como un *kind* de AML

BIBLIOTECA VIRTUAL

3.8 Basados en Eventos

Aunque su vinculación con la Informática es ya muy antigua, en los últimos años los paradigmas basados en *eventos* han alcanzado una importancia aún mayor. El enfoque conocido como *invocación implícita* [GN91], en especial, muestra una clara capacidad para el dinamismo, en el sentido de que define una arquitectura cuyo grado de acoplamiento es muy débil. Esto es aún más cierto en su versión distribuida, conocida como el paradigma *publicar/suscribir* [Car98, Eug01], actualmente uno de los más populares en la industria.

No obstante, todo lo dicho más arriba se refiere únicamente a un *estilo* concreto (§2.4.5), que podrá ser representado, con mayor o menor fortuna, por todo ADL lo suficientemente potente. En este apartado, sin embargo, no se pretende utilizar el paradigma para describir la estructura del sistema, sino la infraestructura del propio lenguaje; es decir, se utilizan los *eventos* como concepto básico, y se asume que el lenguaje realiza una gestión global de los mismos a lo largo de toda la especificación.

La noción de evento es muy intuitiva, y admite una aproximación tanto formal como informal. De hecho, existe un representante de cada caso en este grupo. C2SADEL utiliza una noción informal de evento, y de hecho obliga a la descripción de todo sistema en una versión restringida del estilo de invocación implícita. Las formalizaciones posteriores, basadas en Z [Med99] no han realizado ninguna contribución en este aspecto. Por su parte, el fundamento formal de Rapide es la noción de *poset*, es decir, la de conjunto parcialmente ordenado, u orden parcial de eventos: se trata de un formalismo sencillo, pero de gran potencia y muy bien estudiado; muestra, además, capacidad para la expresión de *concurrencia verdadera*.

Este grupo se puede considerar vinculado, en cierto modo, con el anterior. Debe recordarse que las álgebras de procesos han sido planteadas, en teoría de la concurrencia, como la definición de patrones de emisión y recepción de eventos. De hecho, se puede establecer formalmente la correspondencia entre la especificación algebraica de un sistema concurrente, y el orden parcial de mensajes asociado, aunque la semántica asociada no es siempre compatible. Por otra parte, el modelo de C2SADEL ha sido comparado en numerosas ocasiones con el de Wright, en particular

en sus respectivas –e incompatibles– nociones de conector.

3.8.1 Rapide

Rapide [LKA+95, LV95, Ken95] es un ADLdesarrollado por el grupo de David Luckham en la Universidad de Stanford, y basado en la idea de modelar las computaciones e interacciones del sistema mediante *conjuntos de eventos parcialmente ordenados (posets)* [Pra86, LVB+93].

El concepto de arquitectura de software del grupo de Rapide difiere ligeramente del de otros investigadores [LVM95]. Para ellos, los componentes *no* forman parte de la arquitectura, que es una estructura completamente independiente. Esta estructura define los patrones de interacción entre unos *interfaces*, que básicamente son unos tipos [KLM94] de componentes, descritos aquí como una colección de *eventos* de comunicación totalmente abstracta. Posteriormente, los componentes podrán integrarse *con* la arquitectura, *enchufándose* a un interfaz determinado.

En principio, el concepto de arquitectura de Rapide y el habitual se corresponden casi exactamente sin más que equiparar a un interfaz con lo que otros lenguajes llamarían un *tipo de componente*, tal como se ha hecho en el párrafo anterior. En realidad, un interfaz no representa a un componente, sino a los *requisitos de interacción* sobre un componente, pero la mayor parte del tiempo esta sutileza carece de importancia. Sin embargo, sí tiene una consecuencia, de tipo práctico: al ser el interfaz lo que forma parte de la arquitectura, no el componente asociado, no hay ningún problema en sustituir dicho componente (esto es, una implementación) por cualquier otro que cumpla las especificaciones de ese interfaz. Esto aporta una gran flexibilidad desde el punto de vista del uso del lenguaje (en especial, para la simulación), pero en realidad no supone ningún tipo de dinamismo para la arquitectura, ya que esta no cambia en absoluto, y el único elemento que conoce en realidad –el interfaz– no ha sido alterado en absoluto. Es decir, las modificaciones no se realizan *en la arquitectura*, sino simplemente en el sistema descrito.

Como se ha dicho, un interfaz (un tipo) de Rapide está compuesto de eventos. Los eventos son complejos, admiten parámetros e incluso un valor de retorno; de modo que habitualmente se piensa en ellos como si de métodos se tratase. De nuevo, la diferencia existe, pero puede obviarse la mayor parte del tiempo. Se distinguen dos tipos de eventos: aquellos que son *observados* por el interfaz, y los que son *generados* por él. Se denominan, respectivamente, *acciones externas* y *acciones públicas*, y se puede pensar en ellos como métodos ajenos y métodos propios, respectivamente. La computación en Rapide se define relacionando la observación de acciones externas con la iniciación de acciones públicas, mediante la especificación de un *orden parcial*.

Los eventos de Rapide se difunden por todo el sistema mediante un mecanismo de difusión global (*broadcast*). Sin embargo, cada interfaz sólo observa aquellos que tiene definidos como acciones externas. La relación entre los eventos localizados en distintos puntos de la especificación puede tomar dos formas: utilizando *conexiones* y mediante *restricciones*. Dos eventos que se han comparado se consideran totalmente equivalentes, esto es, *son el mismo evento*. Así, partes separadas de la arquitectura, que habrían podido incluso definirse independientemente, quedan ligadas de manera directa. Más flexible e interesante es la unión mediante restricciones. En este caso, se especifican relaciones *causales* entre dos eventos: uno es la causa del otro y la aparición del primero implica la iniciación del segundo; pero se les sigue considerando eventos distintos. En realidad, lo que se ha hecho es definir un *orden parcial* entre ellos.

```
type Productor(Max:Positive) is interface
    action out Send (N:Integer);
    action in Reply (N:Integer);
behavior
    Start => Send(0);
    (?X in Integer) Reply(?X)
    where ?X < Max => Send(?X+1);
end Productor;

type Consumidor is interface
    action in Receive (N:Integer);
    action out Ack (N:Integer);
behavior
    (?X in Integer) Receive(?X) => Ack(?X):
end Productor;

architecture ProdCons()
    return SomeType is
        Prod: Productor(100);
        Cons: Consumidor;
connect
        (?n in Integer)
        Prod.Send(?n) => Cons.Receive(?n);
        Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCons;
```

Figura 3.8: El Problema de los Productores-Consumidores en Rapide

Con esto se permite la descripción de patrones de interacción de eventos a lo largo de la arquitectura; sin embargo, dichos patrones quedan entremezclados con la definición de las interfaces, y en general son poco reutilizables. Las conexiones son poco flexibles; en definitiva, si una de las acciones públicas de A es una acción externa de B, lo único que se está diciendo es que B puede invocar a uno de los métodos de A. Las restricciones son más flexibles, pero pierden parte de la modularidad del sistema, al estar definidas de forma global.

La Figura 3.8 muestra la formulación de una variante del célebre problema de los Productores-Consumidores en Rapide. En ella, los símbolos del tipo `?X` hacen referencia a los parámetros de entrada de un evento ("método") cualquiera, antes y después de ser ligado por el correspondiente evento externo. Por su parte, los símbolos del tipo `=>` expresan el establecimiento de un orden parcial entre sus extremos. Dicho de otro modo, se indica que su parte izquierda es la *causa* de su parte derecha. El esquema descrito es muy potente y no demasiado complicado, aunque queda un poco oscurecido por la enorme cantidad de detalles que se acumulan en las distintas partes de la especificación del lenguaje. En realidad, reducido a sus mecanismos básicos, resulta bastante intuitivo y razonablemente cómodo.

Las arquitecturas de Rapide fueron diseñadas pensando en la *simulación*. Toda descripción Rapide puede ser reducida automáticamente a un simple *poset*, esto es, una secuencia (parcialmente) ordenada de eventos. Sin embargo, la especificación del orden parcial normalmente no es determinista[5], lo que significa que una simulación concreta (una *ejecución de la arquitectura*) sólo tomará uno de los caminos posibles, y el resto serán ignorados. El *poset* resultante, por tanto, describe el encadenamiento de eventos en una ejecución determinada, lo que no agota todas las posibilidades de la especificación. Por tanto, las herramientas de Rapide (muchas, muy utilizadas y de gran potencia) no permiten la *verificación* de la arquitectura, sino una *simulación*, que prueba su corrección, pero no la demuestra.

Ha de tenerse en cuenta que el orden parcial se define en la especificación, y por lo tanto es totalmente *estático*. El dinamismo en Rapide viene dado por la sentencia **where** (puede verse

[5]Es perfectamente posible definir especificaciones del tipo `A => (B and C) or (B and D)`. Una ejecución determinada de la arquitectura ha de optar, en esta bifurcación, por un camino u otro; y esta elección afectará a los encadenamientos de eventos posteriores.

un ejemplo en la Figura), que puede crear (o no) un cierto enlace –y por tanto, desencadenar o no una serie de eventos o acciones–, según se cumplan (o no) las condiciones que ésta especifica. Por tanto, se tiene un tipo de reconfiguración programada, basada en restricciones, equiparable en potencia a las de *δarwin* (§3.7.2).

Recientemente [VPL99] se ha añadido a Rapide un conjunto de comandos de reconfiguración, más directos que los anteriores, y que se mantienen *integradas* en el lenguaje. Su forma es análoga a la de las cuatro operaciones básicas de la reconfiguración (§3.4.2), lo que las hace más intuitivas, al mantener una sintaxis imperativa. Por otra parte, pueden ser invocadas de manera interactiva a través de una herramienta (reconfiguración *ad hoc*), o situadas como parte de una restricción, para que se disparen en el momento adecuado (reconfiguración programada). Se denominan, concretamente, *CreateModule, CreatePathway, DeleteModule, DeletePathway* y *ChangeParent*. Sus significados son obvios, mediante la oportuna correspondencia con las cuatro operaciones básicas. Por supuesto, pueden ser (y son) definidos en función de *posets*. Por ejemplo, un enlace (*pathway*) no es sino una conexión causal (un orden parcial) entre dos eventos dados. A diferencia del enfoque anterior, no obstante, pueden ser creados y destruidos *dinámicamente*.

Rapide es uno de los lenguajes más versátiles y utilizados del campo, y no ha dejado de evolucionar en los últimos cinco años. En su contexto se plantea una comparativa muy interesante, pendiente aún de estudio: la relación, a este nivel, entre un modelo de orden parcial de eventos y el enfoque algebraico de lenguajes como Darwin o LEDA; se ha de tener en cuenta que ambos tipos de formalismo están vinculados, teóricamente y en su origen.

3.8.2 C2 (C2SADEL)

La situación del ADL C2SADEL (*C2* **S***oftware* **A***rchitecture* **D***escription and* **E***volution* **L***anguage*) es prácticamente única en el campo de la descripción arquitectónica. Se ha de señalar, para empezar, que es uno de los ADLs más estudiados y mejor conocidos que existen, y que en él se han producido algunos de los avances más importantes de la Arquitectura de Software. Es y ha sido el objeto principal de estudio de investigadores de la importancia de Nenad Medvidovic, Peyman Oreizy o David Rosenblum.

Inspirado originalmente en el diseño de una herramienta para el Diseño de interfaces gráficas [TMA+96], ha sido él mismo dotado de toda una serie de herramientas software que pretenden facilitar su uso, entre las que destacan Argo, ArchShell, ArchStudio o, más recientemente, DRADEL [MRT99]. Se ha creado, incluso, una biblioteca Java que implementa el estilo, de modo que las especificaciones C2SADEL pueden utilizarse para la generación de código. Esta operación se realiza, bien a mano mediante el uso directo de la biblioteca, bien de manera automática usando el entorno DRADEL. Todo este sistema se apoya en la definición de una *teoría de tipos* completa [MRT98], fundamentada principalmente en los trabajos de Amy Zaremski y Jeannette Wing sobre tipos de comportamiento [ZW95], pero adaptado de manera explícita al entorno de la arquitectura de software, y en especial a su noción de componente.

Además, es un magnífico ejemplo del buen uso del concepto de conector [MMP00], pues aunque lo aplica de un modo bastante inusual, le saca mucho más partido que otros lenguajes supuestamente más versátiles. Basándose en este concepto, ha servido para el estudio de la integración de componentes *off-the-shelf* en arquitecturas generales [DNR99]. También se ha trabajado, muy recientemente, en la conversión de sus especificaciones a un formato intermedio

basado en XML [KGO+00], siempre con el mismo fin de lograr el mayor grado de integración posible con otros sistemas y métodos.

Con todo este soporte, C2 se presenta como uno de los lenguajes con mayores capacidades para la expresión de *evolución* de sistemas a través de la arquitectura de software [Med99, MRT99], además de contar con algunos de los mejores estudios sobre el tema específico de la Arquitectura de Software Dinámica [Med96a, Ore96, OMT98, OT98, OGT+99], que ha sido desde el principio uno de sus intereses principales.

Y, sin embargo, tiene una limitación inherente: ha sido diseñado para ser capaz de representar *únicamente* **un** estilo arquitectónico: el *estilo C2*. Se puede argumentar, con parte de razón, que precisamente el hecho de haber acotado su expresividad es lo que ha permitido concretar todos los aspectos anteriores, y ha sido fundamental a la hora de diseñar las herramientas, plantear la integración de COTS o traducirlo a otros entornos. Sin embargo, los estudios sobre dinamismo y la teoría de tipos son independientes del estilo, y podrían aplicarse a cualquier otro ADL.

C2 es un estilo de "alto nivel", ya que describe un patrón ampliamente utilizable, por sistemas muy diversos. Así pues, aunque fue concebido para el diseño de interfaces gráficas de usuario, no se ha quedado en este campo, y ha sido aplicado, por ejemplo, en un juego de ordenador en red [Med99], o en un sistema de control de misiles aire-tierra [OGT+99]. Sin embargo, el hecho de estar restringido a un estilo concreto no deja de ser una limitación, por lo que hay sistemas que no pueden expresarse en él de modo natural [Wol97].

```
architecture StackArch is {
    components {
        StackADT;
        StackArtist;
        StackPieArtist;
        Graphics; }
    connectors {
        bus1;
        bus2; }
topology {
    connector bus1 connections {
        top_ports {
            StackADT filter no-filtering; }
        bottom_ports {
            StackArtist filter no_filtering;
            StackPieArtist filter no_filtering; }}
    connector bus2 connections {
        top_ports {
            StackArtist filter no_filtering;
            StackPieArtist filter no_filtering; }
        bottom_ports {
            Graphics filter no_filtering; }}
}}
```

Figura 3.9: Una configuración sencilla en C2SADEL

Toda descripción en C2SADEL[MTW96] consta de tres partes: un conjunto de *componentes*, otro de *conectores*, y una especificación de la *topología* que conforman, tal como puede verse en la Figura 3.9. Los componentes tienen un comportamiento propio, definido por su código, que no es conocido a priori por la arquitectura. Los conectores son básicamente *buses de eventos*, y manifiestan lo que se ha denominado *reflexión contextual*: no tienen un interfaz fijo, sino que éste depende de los componentes que estén conectados a ellos, esto es, de su contexto. Además, en un rasgo que los diferencia de cualquier otro ADL, *se permiten los enlaces de conector a conector*, directamente. Esto tendrá un sentido claro en cuanto se explique su funcionamiento:

- Todo componente y conector tiene dos puntos de conexión, que se denominan *arriba (top)* y *abajo (bottom)*. Éstos se proporcionan de manera implícita, y son los únicos que existen.

- La parte de arriba de un componente puede unirse a la parte de abajo de un conector. La parte de abajo de un componente puede unirse a la parte de arriba de un conector. *Las uniones inversas no pueden realizarse.* Un componente sólo puede estar conectado a un conector por cada extremo, como máximo; en cambio, el conector admite cualquier número de componentes. Todas estas restricciones quedan mucho más claras sin más que echar un vistazo a su representación gráfica.

- Dos componentes no pueden conectarse de manera directa, sino que siempre han de hacerlo a través de un conector. En cambio, como se ha dicho, no hay problema en conectar entre sí a dos conectores. Por otra parte, no se admiten ciclos.

La comunicación es totalmente basada en eventos, y no se permite ningún otro tipo de interacción, por ejemplo una RPC. Los componentes nunca se comunican directamente con otros elementos, sino que se limitan a emitir eventos. Un componente toma la iniciativa emitiendo un evento de *petición*, que siempre se envía hacia el conector que se encuentra *arriba*. Desde allí, el conector lo difunde a todos los componentes situados por encima. Si está enlazado con otro conector, se difunde por éste de la misma manera. Tarde o temprano, algún componente producirá una respuesta, que tomará la forma de un evento de *notificación*. Por supuesto, la causa de una notificación podría ser totalmente independiente. La notificación se envía al conector que está situado *abajo*, el cual lo difunde a los componentes y conectores que están por debajo de él, y así sucesivamente. En resumen, las peticiones *suben* y las notificaciones *bajan* por la arquitectura. En la unión entre cada componente y conector puede introducirse un filtro, en el que se indicará, por ejemplo, de qué componentes se quiere recibir eventos, de modo que los que tengan otro origen serán simplemente ignorados. En la configuración de la figura se ha mantenido el filtro implícito: todo se acepta.

C2SADEL es más "dinámico" que otros ADLs, en el sentido de que la forma de sus conectores no está predeterminada, y que en cada momento sólo se activan algunos de los enlaces; pero la configuración no puede cambiar en ningún momento, ya que todas las conexiones son fijas y explícitas. En sí, por tanto, el ADL no es dinámico.

Oreizy fue el primero en proponer la idea de disponer de una triada de lenguajes [Ore96]: un ADL para la descripción, un AML (*Architecture Modification Language*) para los cambios, y un ACL (*Architecture Constraint Language*) para las restricciones. Acaba de describirse el primero, y el tercero aún no ha sido descrito –por el momento, el grupo de C2 usa el Lenguaje de Predicados de Armani–. En cuanto al segundo, fue definido por Medvidovic y Oreizy [Med96a, OMT98], y es conocido como el C2 AML. En realidad, consta simplemente de las cuatro operaciones básicas (§3.4.2), con la salvedad de que al tener dos tipos de elementos (componentes y conectores), se diferencian seis: *AddComponent, AddConnector, RemoveComponent, RemoveConnector, weld* y *unweld*. El significado es obvio, aunque tal vez sea conveniente señalar que las dos últimas se corresponden, respectivamente, con los *link* y *unlink* clásicos.

Dado que se introducen de manera interactiva, no tienen parámetros, y es el propio sistema el que los solicita (por ejemplo, el tipo del componente a crear, o el lugar en que se va a crear un enlace). Sólo admite, pues, reconfiguraciones *ad hoc*, basadas en sus herramientas, esto es: Argo,

ArchShell y Dradel. La consistencia final de la arquitectura queda garantizada por un *Gestor de Evolución de la Arquitectura (AEM, Architecture Evolution Manager)*, que es un elemento integrado en la herramienta [OMT98]. Como alternativa, se puede crear un *script* que las realice, en el que cada operación se invoca como un método de la clase del componente a modificar. Esto es sólo posible, por supuesto, cuando se actúa directamente sobre la implementación Java, con el apoyo de la biblioteca definida para C2 [OT98, Med99].

3.9 Basados en Grafos

Se ha dicho en multitud de ocasiones que la representación más típica de una arquitectura de software se basa en el típico diagrama *de cajas y líneas*, que refleja el esquema del sistema de manera evidente. De hecho, es cierto que una buena parte de las propuestas de Descripción en Arquitectura de Software han partido del enfoque gráfico. No obstante, toda notación de este tipo tiene dos problemas con los que ha de tratar: la falta de rigor y la poca escalabilidad. Ambos suelen solucionarse de la misma manera: construyendo un sistema formal que permita la especificación de los diagramas. Normalmente, el formalismo se basa en teoría de grafos, aunque también es posible un enfoque más complejo, basado en teoría de categorías (§3.9.4).

Entre todos los enfoques posibles basados en grafos, el que más veces ha sido aplicado, y no solamente en la Arquitectura de Software, ha sido el de las Gramáticas de Grafos. Se trata de un sistema formal, gráfico o simbólico (o ambos), que a partir de una configuración inicial (un grafo concreto) puede elaborar distintas versiones del mismo, manteniendo un enfoque constructivo. De este modo, se tiene una versión controlada de la evolución del sistema.

El enfoque arquitectónico más clásico hace una analogía muy sencilla, y considera que los grafos expresan configuraciones, y las gramáticas completas, estilos. Otra posibilidad es verlas como arquitecturas dinámicas; las reglas de la gramática expresan el modo en que puede ser modificada la configuración actual. Normalmente, no se habla de dinamismo muy a la ligera: como podrá verse, los sistemas que lo expresan de hecho son notablemente elaborados.

3.9.1 Precedentes

En esta sección, se revisarán brevemente tres propuestas ligadas a la descripción gráfica de estructuras arquitectónicas, y que han introducido de algún modo la idea de dinamismo en este contexto, aunque no de modo tan explícito como las siguientes.

Gramáticas-Δ: Garp

El sistema Garp fue desarrollado en el seno de la Universidad de Illinois como parte de la tesis doctoral de Steven Göring sobre las *Gramáticas-Δ* [Gör90], un tipo de gramáticas de grafos. Proponía, incluso antes que Conic (§3.6.1), la separación de los aspectos de computación y configuración de un sistema mediante el uso de lenguajes diferentes. Los módulos eran elaborados mediante un lenguaje de programación convencional, y a continuación eran ensamblados para formar un sistema completo. La composición concreta, esto es, la *configuración* se realizaba con el concurso de una herramienta visual (Garp) en la que se expresaba mediante un grafo la relación

entre los distintos módulos. El sistema se encargaba después de gestionar la comunicación entre los módulos. Por lo tanto, la descripción de la configuración era puramente gráfica.

El sistema carecía apenas de dinamismo: la configuración podía ser modificada, pero de manera estrictamente manual. Se trataba, pues, un caso más de reconfiguración *ad hoc*.

Teoría Sintáctica de Dean & Cordy

En un conocido artículo, Thomas Dean y James Cordy [DC95] propusieron lo que denominaron como "una teoría sintáctica de la Arquitectura de Software". Consistía en definir una notación para la arquitectura, basada en grafos y en el concepto de correspondencia de patrones (*pattern matching*). Esta correspondencia se define de manera abstracta en una primera fase, se apoya en la definición de expresiones regulares en la segunda, y termina utilizando gramáticas de grafos en la tercera. En todo momento la notación es totalmente gráfica, y define únicamente pictogramas; los únicos símbolos utilizados sirven para expresar el orden parcial de las expresiones regulares, y siempre van unidos a una flecha.

En ningún momento este sistema muestra capacidades dinámicas. Sin embargo, hacia el final se ocupa de manera explícita de la *Arquitectura Dinámica*, esto es, de los cambios dinámicos en la estructura de un sistema. Su propuesta es modelar los posibles cambios de interfaz de cada uno de los subsistemas como un estado, e incluir todas las posibles configuraciones en un autómata de estados finitos. Las transiciones entre estados podrían ser provocadas, por ejemplo, por eventos predefinidos del sistema. No obstante, esta idea nunca llegó a ser desarrollada.

Aproximaciones de Holt

Ric Holt, uno de los primeros autores en interesarse por la Arquitectura de Software, ha realizado dos propuestas para un lenguaje de descripción basado en grafos. La primera, realizada en cooperación con Spiros Mancoridis, describe un sistema basado en el concepto de Grafos-Tubo [HM94]. La segunda propuesta es posterior, y se concibe como una mejora sobre la primera, a la que dota de mayor flexibilidad y rigor. Los grafos-tubo son ahora un caso más en un sistema de grafos coloreados, que se manipulan mediante el álgebra de relaciones binarias de Tarski [Hol96a, Hol98]. Ninguno de estos enfoques incluye dinamismo; aunque resulta bastante sencillo imaginar cómo podría ser añadido al sistema, los autores no llegan a plantearlo en ningún momento.

3.9.2 Gramáticas de Le Métayer

El sistema que se describe a continuación fue propuesto por Daniel Le Métayer [LM96, LM98], y guarda también cierta relación con sus trabajos sobre el modelo Gamma.

Es el ejemplo arquetípico del enfoque basado en grafos. Usa una aproximación directa, en la que los componentes se describen como *nodos* y las conexiones entre ellas como *arcos*. El conjunto forma un grafo completo, que expresa una *configuración* concreta del sistema. Las reglas con las que se construye el grafo son expresadas mediante una Gramática de Grafos, en este caso Independiente del Contexto, que describe, en definitiva, un *estilo arquitectónico*.

La Figura 3.10 muestra el ejemplo más famoso de Le Métayer, en el que se describe un sistema Cliente Servidor, con múltiples clientes (C) y servidores (S), caracterizado por la existencia de un gestor (M), que hace las funciones de conector, aunque este punto no es puesto de relieve

por el autor. Cuando un cliente llega al sistema, se enlaza con el gestor, y éste se encarga de transportar sus mensajes hasta el servidor requerido. El componente X es externo al sistema, pero podrá afectarlo.

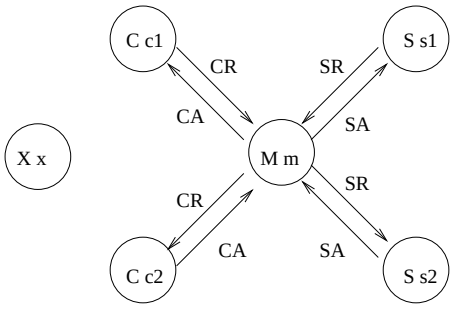

Figura 3.10: Arquitectura Cliente-Servidor de Le Métayer

El verdadero punto clave de este trabajo no está en la representación del sistema como un grafo, que es lo más evidente, sino en el hecho de que este grafo puede expresarse como un *multiconjunto*. La representación del sistema anterior en esta forma se muestra en la Figura 3.11. Todos los elementos muestran la misma estructura, como una expresión en la que se indica su *tipo*, y se incluye su identificación entre paréntesis. Esto vale tanto para los nodos como para los arcos; los primeros tienen un único parámetro, indicando el nombre del componente; mientras que los segundos tienen dos parámetros, reflejando cuáles son los dos elementos que conectan.

$$\{X(x), M(m), C(c_1), C(c_2), S(s_1), S(s_2),$$
$$CR(c_1, m), CR(c_2, m), CA(m, c_1), CA(m, c2),$$
$$SR(m, s_1), SR(m, s_2), SA(s_1, m), SA(s_2, m)\}$$

Figura 3.11: Arquitectura Cliente-Servidor como Multiconjunto

En realidad, el método descrito no es aplicable *sólo* a grafos, sino que puede ser utilizado, sin apenas cambios, en una estructura de hipergrafos (§3.9.3), tal como indica el propio autor ya desde el principio [LM96]. Resulta evidente que, al expresarlos como elementos de un multiconjunto, basta con aumentar el número de parámetros de los elementos para expresar adecuadamente los hiperarcos. La Gramática de Grafos utilizada ha de ser definida específicamente para cada sistema, y expresa el *estilo* del mismo, en el sentido de que toda configuración posible puede ser expresada como una derivación sintáctica de dicha Gramática. Ésta es independiente del contexto[6], lo que significa que puede plantearse como una reescritura de nodos o una reescritura de arcos [LM96]. Se toma la segunda opción, aunque ambas son equivalentes.

El dinamismo en este enfoque viene dado por la existencia de un elemento más, que se incluye de manera implícita en toda configuración. Este elemento se denomina *coordinador*

[6]Lo que introduce unas limitaciones idénticas a las que se mencionan en la exposición del modelo de Hirsch (§3.9.3), y por idénticos motivos.

(coordinator), y se define como una serie de reglas de reescritura que indican los cambios que pueden producirse en el sistema ante la presencia de ciertas situaciones. La Figura 3.12 muestra la definición del coordinador para el sistema ya conocido. Cuando se producen las condiciones de la parte izquierda, se reestructura el sistema de la forma indicada en la parte derecha.

$$
\begin{array}{lcl}
Coo_{CS} = & & \\
\quad X(x), M(m), x.newc = true & \rightarrow & X(x'), M(m), CR(c,m), CA(m,c), C(c) \\
\quad C(c), c.dejar = true, CR(c,m), CA(m,c) & \rightarrow & \emptyset
\end{array}
$$

Figura 3.12: Definición del Coordinador para el Ejemplo Anterior

Ha de tenerse en cuenta que las reglas del coordinador no son independientes del contexto, y ni siquiera gramaticales, por lo que no se puede garantizar su consistencia con el estilo de la arquitectura. Por ello, tras cada reconfiguración han de lanzarse una serie de algoritmos de chequeo, que comprueban si la configuración actual sigue siendo válida. Otro aspecto fundamental es que normalmente la definición de las reglas de reescritura depende de aspectos internos de la definición de los componentes –básicamente, de los *valores* de sus atributos, como el *newc* de la entidad externa X–, que sigue a su vez una sintaxis diferente. Esta última está claramente inspirada en Csp, con algunas ligeras extensiones.

En resumen, el enfoque es interesante, pero poco flexible y con poca consistencia interna, ya que mezcla dos o tres especificaciones. Uno de los detalles más curiosos es que, una vez que se traduce el sistema a un multiconjunto, Le Métayer no intenta usar ningún tipo de máquina química. Sin embargo, precisamente éste será el origen de la propuesta de Wermelinger (§3.10.2) para reconfiguraciones basadas en un par de Chams.

3.9.3 Hipergrafos de Hirsch

Las representaciones arquitectónicas basadas en grafos, como la que acaba de examinarse, consideran habitualmente un modelo en el que los nodos son componentes y los arcos son enlaces. La conectividad que se proporciona de este modo es solamente punto a punto y por parejas; esto exige a menudo la creación de nodos auxiliares –equivalentes a conectores– que permitan la expresión de ciertas estructuras de interacción complejas. Es claro que en multitud de ocasiones se tienen interacciones complejas que involucran a dos o más actores, por lo que cabría expresar una representación menos indirecta. Al plantearse este problema, Dan Francisco Hirsch llegó a la conclusión de que la causa era que la propia definición de grafo resultaba demasiado restringida; por tanto, propone utilizar en su lugar la noción de *hipergrafo*.

Una vez que se plantea esta utilización, sin embargo, decide usar un enfoque completamente opuesto al habitual. Así, en el modelo de Hirsch son los *componentes* los que se expresan como *hiperarcos*, y los *nodos* pueden verse como sus *puertos*, o como *enlaces*. Dos componentes (hiperarcos) se enlazan en el momento en el que comparten un nodo (puerto), por simple yuxtaposición. Dado que un mismo nodo puede participar en varios hiperarcos, la interacción múltiple ya resulta posible. No se definen, pues, ni adjunciones (*attachment*) propiamente dichas, ni conectores de primera clase. Un hiperarco puede hacer funciones de conector, pero en

tal caso simplemente actúa como un componente específico.

Modelo Inicial

La formulación del modelo de hipergrafos de Hirsch ha pasado por dos etapas. En la primera [HIM99] se define el marco original de descripción, en el que una arquitectura se define como una gramática de grafos. El objetivo, en este caso, no es tanto la descripción de un estilo, como en el modelo de Le Métayer (§3.9.2), sino la especificación de la *evolución* de la arquitectura. Esto hace que su intención sea eminentemente dinámica desde un principio. Esto no es extraño, ya que la utilización del modelo gráfico se hace desde la perspectiva de la teoría de la concurrencia, en la línea de los trabajos de Ugo Montanari.

Siguiendo el esquema más habitual, la gramática se presenta como un Sistema de Reescritura de Grafos (GRS). Cada regla de reescritura es aquí una *producción de grafo*, que transforma (reescribe) un grafo en otro grafo, manteniendo algunos de sus elementos (nodos o arcos), borrando otros, y creando otros nuevos. La definición es la estándar, adaptada a hipergrafos; pero además se añade la restricción de que se trate de producciones *independientes del contexto*. Es decir, puede generar un grafo arbitrario a partir de un *único* hiperarco, con la única restricción de mantener su conectividad. Dicho de otro modo, un arco se transformará en un sub-grafo sin necesidad de considerar su entorno, esto es, los arcos adyacentes; pero todos sus nodos habrán de mantenerse ligados. Los hiperarcos están etiquetados; por tanto, basta encontrar un arco cuya etiqueta coincida para poder aplicar la regla de reescritura. Es fundamental tener en cuenta de que en este modelo sólo se permite este tipo de producciones; ésta es la fuente de las mayores limitaciones del modelo, como se indica más abajo.

Se parte, pues, de un hipergrafo inicial G_0, que va evolucionando por sustitución sucesiva de hiperarcos, según la gramática, hasta llegar a la descripción de una arquitectura. Cada derivación concreta proporciona una configuración concreta.

La primera formulación del método de Hirsch definía tanto el hipergrafo inicial como todas sus producciones de manera exclusivamente gráfica, sin proporcionar ningún enfoque simbólico. En la Figura se da un ejemplo de una regla de producción. Ha de tenerse en cuenta que las "cajas" representan a hiperarcos, y los "círculos" que se unen a ellas representan a nodos. La parte izquierda expresa el hiperarco que va a ser reescrito, y la derecha el grafo producido.

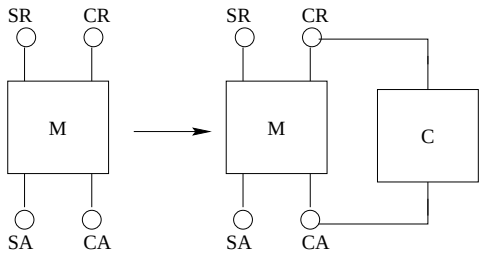

Figura 3.13: Una Regla de Producción de Grafos [HIM99]

Al ser puramente gráfico, el modelo de Hirsch es muy visual, pero poco práctico. Cualquier especificación resulta demasiado grande, y ciertas expresiones resultan poco naturales. Cada

arco puede ser visto como un arco terminal, que no necesita evolucionar más, o como un arco no terminal, que ha de ser reescrito, y se ubica por tanto en una configuración no definitiva.

Cada una de las producciones puede ser vista como una regla de la gramática, definiendo el estilo del sistema, o como una producción dinámica, que permite realizar cambios sobre el mismo. Básicamente la diferencia es que las primeras reescriben arcos no terminales, y las segundas arcos terminales. Por ejemplo, la producción mostrada en la Figura 3.13 es una regla de dinamismo, en la que se genera un nuevo componente, pero no se elimina el original. No se proporciona, en principio, ningún método para disparar una de estas reglas; por tanto, se está describiendo un método de reconfiguración *ad hoc* o externo.

El hecho de que sólo se consideren producciones libres de contexto limita más de lo que parece la aplicabilidad del modelo. Una consecuencia inmediata es que no se pueden eliminar *nodos*, ya que las producciones han de preservar la conectividad del hiperarco que transforman. Esto plantea la curiosa situación, traducida a términos arquitectónicos, de que en un momento dado se admite la creación o destrucción de componentes, *pero no la de sus puertos*. Esto tiene cierta lógica, en el sentido de que siguen siendo los puertos de otros; pero un nodo podría llegar a quedar aislado, con lo que carecería de significado real. Tampoco pueden realizarse operaciones tales como intercambiar dos componentes (arcos) que se crearon a la vez, porque en cada momento sólo puede reescribirse uno de ellos.

El modelo de Hirsch es ciertamente prometedor, aunque bastante incómodo, y su dinamismo es bastante restringido. Consciente de ello, Hirsch propuso una segunda versión, apoyándose en el modelo de mosaicos de Roberto Bruni, Fabio Gaducci y Ugo Montanari.

Modelo de Mosaicos

La segunda versión del modelo de Hirsch [HM99] añade una semántica más precisa, y una notación simbólica. Es una ampliación del modelo descrito, en la que todo lo que se ha dicho hasta el momento sigue siendo cierto; su mayor objetivo es facilitar la descripción de dinamismo, mediante lo que se denominan *transformaciones*. La semántica de las transformaciones se basa en el modelo de mosaicos, su notación (la notación-λ doble) y la lógica derivada (la Lógica de Mosaicos), que fueron definidas en la tesis doctoral de Roberto Bruni [Bru99].

El concepto de *transformación* se describe del modo siguiente. Toda arquitectura descrita mediante un hipergrafo H_1, y generada (a partir de un grafo inicial G_0) mediante un GRS del modo que acaba de verse, tiene asociada una derivación de la forma $G_0 \Rightarrow_{p1} G_1 \Rightarrow_{p2} \ldots G_n \equiv H_1$. Al demostrarse que el grafo es gramatical, también se demuestra que pertenece a un estilo determinado. Se quiere reconfigurar este grafo para obtener uno nuevo, H_2, lo que puede requerir la introducción de cambios no expresables en una gramática independiente del contexto. Por ejemplo, se quiere eliminar un nodo, o permutar dos arcos dados. Para poder hacerlo, antes ha de poder encontrarse una derivación capaz de generar H_2, lo que demostrará que dicho grafo es gramatical. Se toman entonces las dos derivaciones; a la original se le llama *prueba inicial*, y a la recién obtenida, *prueba final*. La operación capaz de pasar de la prueba inicial a la prueba final es lo que se denomina una *transformación*, y expresa una reconfiguración.

Cada una de estas dos pruebas puede verse como una única regla de producción derivada, obtenida mediante composición secuencial, es decir, mediante el emparejamiento consecutivo de los arcos no terminales de las producciones sucesivas. Pero además se puede representar una

producción mediante un único grafo, fusionando los dos lados de la regla a partir de sus nodos comunes. En definitiva, se puede expresar *toda la prueba* como un grafo de este tipo.

Por tanto, la *transformación* se expresará como una regla que reescribe el grafo de prueba inicial en el de prueba final. Con este fin se recurre al Modelo de Mosaico.

Un *mosaico (tile)* se define como una regla mediante la que una configuración inicial g se transforma en una final h, en presencia de dos observaciones, denominadas respectivamente el *disparador* y el *efecto*. Estos mosaicos, representados en dos dimensiones, se combinan según las reglas de la Lógica de Mosaicos, es decir, en secuencia (vertical) o en paralelo (horizontal). Dos aspectos particularmente interesantes de los mosaicos son su representación simbólica y la categoría derivada. Todo mosaico puede representarse mediante la denominada *notación-λ doble*, que es un híbrido del cálculo-λ simplemente tipado y algunas de las características del cálculo-π. Esta notación se corresponde con una categoría cartesiana cerrada doble, lo que le garantiza ciertas propiedades. Por tanto, una transformación, descrita como un mosaico, puede ser representada de manera simbólica mediante la notación-λ doble.

BIBLIOTECA VIRTUAL

$$
\begin{array}{lcl}
\mathit{Hijo} & : & (1 \to *) \times (2 \to *) \to (1 \to *) \\
\mathit{Hijo} & \stackrel{\text{def}}{=} & \lambda(x_L, x_F).\, \lambda n.\, \nu m.\, h(n,m) \,|\, x_L m \,|\, x_F(m,n)
\end{array}
$$

x_L, x_F	Hiperarcos No Terminales
n, m	Nodos
h	Hiperarcos Terminales

Figura 3.14: Una Transformación en Notación-λ Doble

La Figura 3.14 muestra una regla de producción –un mosaico– utilizando la notación-λ doble. Aunque no es obvio en absoluto, este ejemplo describe cómo un grafo formado por un nodo (n) y un arco no terminal (x_L) se transforma en un grafo formado por dos nodos, dos arcos no terminales y un arco terminal (h). La regla puede verse además como una función-λ de segundo orden, cuyo *tipo* se indica en la línea superior. Gracias a esto último, el sistema es susceptible de tratamiento automático.

Una transformación, por tanto, se describe como la reescritura de una función de este tipo en otra; es decir, se trata de un mosaico en la que tanto la configuración inicial como la final se expresan de este modo. Dado el modo en que se obtienen las transformaciones, los distintos grafos *siempre* están dentro del estilo correcto; no es necesario aplicar ningún tipo de algoritmo para comprobarlo, como ocurría en el modelo de Le Métayer (§3.9.2).

En resumen, el nuevo enfoque es mucho más flexible que el anterior, aunque también resulta engañoso. Las transformaciones no se aplican sobre el grafo que representa al sistema actual, sino sobre el que indica la derivación gramatical. En términos arquitectónicos, por tanto, esto equivale a hacer que el sistema *retroceda en el tiempo*, para deshacer producciones que ya fueron aplicadas. Por otra parte, deducir la transformación deseada puede ser en ocasiones complicado e incluso imposible. No obstante, este modelo puede ser interesante como soporte teórico de otros sistemas más prácticos, si bien este camino no ha sido aún explorado.

3.9.4 Modelo Categórico de Wermelinger

Este modelo mezcla conceptos formales procedentes de tres fuentes diferentes, aunque ligados entre sí: la teoría de categorías, las gramáticas de grafos y los derivados del lenguaje UNITY [CM88]. Una exposición completa del modelo sería en consecuencia realmente larga, por lo que aquí se intentará hacer sólo una revisión general; para mayores detalles, se puede consultar la documentación original en [WF99, Wer99].

Wermelinger, siguiendo a Fiadeiro y Maibaum [FM96], parte del concepto categórico de *pushout*. Éste se define como el colímite de un diagrama –esto es, un grafo dirigido–, en el cual existe un morfismo (arco) de todos los objetos (nodos) a otro objeto; éste último es el que recibe el nombre de *pushout*, y simbolizará al grafo completo. La Figura 3.15 muestra la unión de dos diagramas, cada uno de los cuales es un *pushout*.

A continuación, plantea la definición de una transformación de grafos, como una producción en una Gramática de Grafos que *no* es independiente del contexto. Las producciones tienen la forma $L \xleftarrow{l} K \xrightarrow{r} R$, en la que L es un grafo origen (parte izquierda) y R es el grafo transformado (parte derecha). K expresa un interfaz entre ambos, y las operaciones l y r indican correspondencias entre las etiquetas. La Figura 3.15 muestra cómo se aplica una regla de esta forma sobre un grafo G, obteniendo como resultado el grafo H (y un complemento de *pushout*, D). Este tipo de producciones expresan todas las interacciones del modelo.

$$\begin{tikzcd}
L \arrow[d, "m"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d"] & R \arrow[d, "m^*"] \\
G & D \arrow[l, "l^*"'] \arrow[r, "r^*"] & H
\end{tikzcd}$$

Figura 3.15: Aplicación de una Producción de Grafo sobre G

A continuación Wermelinger introduce el lenguaje COMMUNITY, desarrollado también por Fiadeiro y Maibaum [FM97] como una variante de UNITY [CM88]. Éste lenguaje se usa a título de ejemplo, pero su sintaxis y semántica formal permiten igualmente considerarlo como una parte esencial de la propuesta, ya que quedará perfectamente integrado en el esquema arquitectónico. No se describe aquí la estructura del lenguaje: sus programas constan simplemente de la declaración de variables externas e internas, y de una secuencia de asignaciones guardadas de la forma [*guarda* $\rightarrow$ *asignación*], en la que la guarda es una etiqueta que origina el disparo alternativo de una u otra asignación.

Se define pues una *configuración* como un diagrama formado por programas COMMUNITY, cuya relación se expresa como arcos de reescritura que indican las distintas sincronizaciones. Para diseñar una verdadera arquitectura, es necesario definir *conectores*. Éstos son grafos con un cierto número de roles (aridad), donde se define un "pegamento" (*glue*) G, que expresa la relación entre dichos roles. Cada uno de los roles es también un programa COMMUNITY. Por ejemplo, la Figura 3.16 muestra un conector binario.

Los roles *Far* y *Near* hacen referencia al ejemplo concreto, un sistema de carretas móviles [RMP97]. Cada uno de ellos puede unirse directamente a un programa; de este modo, el conector controlará la interacción entre los dos programas. La Figura 3.17 muestra la aplicación

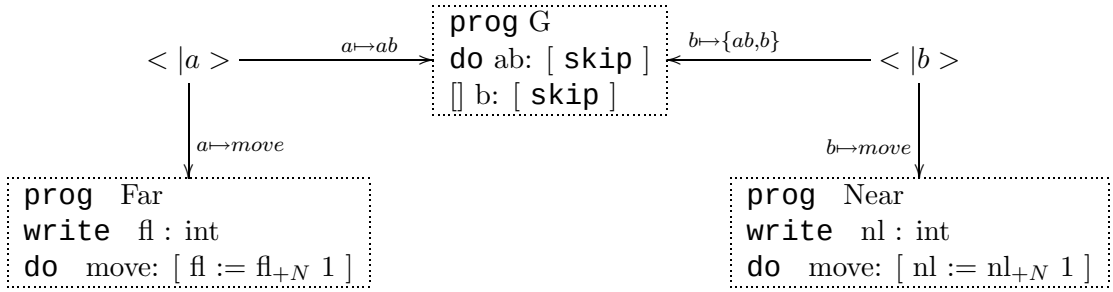

Figura 3.16: Un Conector Binario para Programas CommUnity

completa de este conector a dos programas. En ella, P_1 y P_2 son los programas, R_1 y R_2 los roles, y los C_i son *programas degenerados* de la forma $< |a >$, cuya única función es servir de interfaz entre el pegamento G y los roles.

$$P_1 \xleftarrow{\iota_1} R_1 \xleftarrow{\rho_1} C_1 \xrightarrow{\gamma_1} G \xleftarrow{\gamma_2} C_2 \xrightarrow{\rho_2} R_2 \xrightarrow{\iota_2} P_2$$

Figura 3.17: Un Conector Binario entre los Componentes P_1 y P_2

El conjunto completo forma un diagrama. Sin más que seguir la teoría, se puede obtener automáticamente, como un *pushout* de dicho diagrama, un único programa CommUnity que resume toda la computación. De este modo, la arquitectura de software se diseña de modo que es posible calcular un programa monolítico que la resume.

En este modelo, el dinamismo se introduce mediante la adición de *reglas de reconfiguración*, que son simplemente producciones independientes del contexto, como las ya consideradas. Las reconfiguraciones se plantean como la sustitución de un objeto, esto es, un programa CommUnity, por otro. Para ello, se apoya en el concepto de *superposición*, procedente de teoría de la concurrencia [Kat93]. Se define como la relación existente entre dos procesos concurrentes cuando uno de ellos se define como una extensión del otro, esto es, incluye la totalidad de sus tareas, y puede por tanto sustituirle. En CommUnity se puede expresar la superposición como una correspondencia entre las etiquetas de guarda que disparan las asignaciones.

Por tanto, una reconfiguración se basa simplemente en aplicar una regla a un diagrama ya completo, de modo que se realice una superposición sobre el nodo deseado. Los diagramas quedan perfectamente integrados, y el *pushout* refleja de inmediato la nueva situación: la reconfiguración se ha producido. La Figura 3.18 muestra una relación de superposición, que podría ser aplicada a un diagrama del modo que de ha indicado.

$$\mathrm{Cart}_i \bullet 1 \xleftarrow{1\mapsto 1} \mathrm{Cart}_i \bullet 1 \xrightarrow{1\mapsto 1} \mathrm{Cart}_i \bullet 1 \xrightarrow{\iota} \mathrm{CartLoop}_i \bullet 2$$

Figura 3.18: Reconfiguración por Superposición

Todo lo descrito es válido para diagramas *sin* estado. La ampliación a sistemas con estado

es completamente análoga, y no requiere mayor explicación.

La mayor ventaja de este modelo es, sin duda, su rigor. Cualquier operación sobre la arquitectura, y en particular la configuración, obedece a unas reglas precisas y carentes de ambigüedad. En contra tiene, como es evidente, su gran complejidad. Por otra parte, la evaluación de su capacidad para el dinamismo puede resultar engañosa. En principio, podría parecer bastante limitada, en especial por restringirse a la noción de superposición. Pero no es cierto: la superposición permite definir un comportamiento totalmente nuevo, por lo que permite sustituciones arbitrarias; tan sólo está obligando a dar un criterio que lo relacione con el antiguo proceso. En cuanto a su aplicabilidad sobre la arquitectura, el hecho de considerar producciones dependientes del contexto lo hace más potente que cualquiera de los otros modelos gráficos considerados, ya que permite la modificación de *cualquiera* de sus elementos.

3.10 Modelos Químicos

El modelo de la Máquina Química Abstracta (Cham) es bien conocido en Concurrencia, y ha sido utilizado con frecuencia para describir la semántica operacional de un sistema o lenguaje. Fue desarrollado por Berry y Boudol [BB90] como una evolución del modelo Gamma [BLM96] de Banâtre y Le Métayer (§3.9.2). Ambos son esencialmente sistemas para la manipulación de multiconjuntos, y se basan en la denominada *metáfora química*, una analogía entre la concurrencia y el comportamiento de las reacciones químicas.

La Cham fue el primer modelo puramente formal utilizado de manera *directa* para la descripción arquitectónica. Tal vez por este motivo ha sido uno de los formalismos más populares del campo, conociendo un buen número de variantes. En lo que respecta a dinamismo, su forma resulta engañosa: como se verá a continuación, sólo el modelo de Cham dual de Wermelinger es propiamente dinámico, mientras que el resto únicamente lo aparenta.

3.10.1 Precedentes

La introducción de la Cham en el campo es debida a un célebre artículo de Paola Inverardi y Alexander Wolf [IW95]. En realidad, estos autores sólo pretendían utilizar un formalismo ya conocido para expresar la semántica operacional de una arquitectura de software. Esto es, se trataba en definitiva de realizar una simple especificación formal. Sin embargo, este intento se hizo muy popular, y es la causa de que la Cham haya sido mencionada en múltiples ocasiones como si se tratase de un Adl, sin serlo en absoluto.

Puede llegar a parecer que el esquema de Inverardi y Wolf describe una arquitectura *dinámica*, ya que define una estructura de enlaces entre componentes que aparentemente se crean y destruyen a medida que el sistema evoluciona. Pero esto no es así en absoluto; tan sólo es una confusión generada por la dispersión con la que se describe una arquitectura en este esquema.

Una Cham se define como una serie de reglas de reacción –reescritura– entre moléculas. Una *molécula* es un término que representa a una parte del estado del sistema, y cuya forma viene determinada por una gramática que varía en cada caso. En la propuesta de Inverardi y Wolf, una *arquitectura* se describe como una solución –multiconjunto– de tales moléculas. Tanto los componentes como los tipos de datos manipulados se expresan solamente como *fragmentos de la sintaxis* de estas moléculas. Una correspondencia adecuada entre la gramática y las reglas

de reescritura hace posible que las distintas reacciones químicas expresen interacciones entre componentes. Estas reacciones dependen de las moléculas presentes, lo que significa que ciertas interacciones son sólo posibles bajo ciertas condiciones.

Sin embargo, esto no supone dinamismo, ya que los flujos de datos que producen las conexiones no se crean ni destruyen. Por el contrario, recorren enlaces definidos explícitamente por las reglas de reacción, que son totalmente inamovibles. El esquema es, en realidad, totalmente *estático*, y su dinamismo es sólo aparente. La descripción resulta poco clara: no sólo se mezclan los aspectos de computación e interacción, sino que incluso se pierde la visión global de la arquitectura, que se diluye en una "sopa" de moléculas.

Daniel Yankelevich [IWY97, IWY98, IWY00] intenta un enfoque completamente nuevo: se utiliza una CHAM completa para cada componente. Cada una de ellas se ocupará únicamente de éste, y describirá solamente su evolución computacional. De este modo, los aspectos funcionales y estructurales vuelven a estar separados. Estas máquinas son independientes entre sí; para poder comunicarlas, Yankelevich propone un mecanismo basado en *meta-variables*, que reflejan cada punto de interacción. El papel de una meta-variable es expresar el comportamiento que un componente concreto *espera* del componente externo al que se conecta; el comportamiento global de todo el sistema se infiere después por composición.

Sin embargo, las conexiones entre distintas CHAMs son totalmente *estáticas*, al ser definidas de manera externa; y resultan por ello totalmente inamovibles, por lo que este enfoque tampoco proporciona ninguna auténtica capacidad dinámica.

Cabría aún la mención de otros modelos químicos, relacionados de manera indirecta con la Arquitectura de Software. Los más interesantes son aquéllos que dotan al Gamma original de cierta estructura y capacidad compositiva, como pueden ser el Cálculo-Γ [HLMS92], el Gamma Estructurado [FM98] o, muy especialmente, el HOGamma o Gamma de Orden Superior [LM94], cuyo potencial para la especificación arquitectónica es notable. Ninguno de ellos, sin embargo, ha sido específicamente estudiado dentro del campo, por lo que se trata simplemente de una posibilidad. Su uso directo, por otra parte, tendría los mismos inconvenientes que el modelo original de Inverardi y Wolf.

3.10.2 Doble CHAM de Wermelinger

Este enfoque [Wer98, Wer99] es una variante del modelo de Le Métayer, descrito en la sección 3.9.2. El sistema se describe exactamente del mismo modo, y se traduce a una notación de multiconjuntos de manera análoga.

Wermelinger señala el peor inconveniente del enfoque de Le Métayer: la mezcla de distintos lenguajes. Así, mientras la estructura del sistema era generada por una gramática de grafos, su evolución se describía en función de una reescritura de términos. El objetivo de Wermerlinger es simplificar su uso mediante el uso del mismo formalismo para ambas tareas; este será, obviamente, la CHAM.

En lugar de la gramática de grafos, se dispone de una CHAM *de creación*, que construye el sistema a partir de una configuración de partida. Las reconfiguraciones se describen mediante las reacciones de una segunda máquina, la CHAM *de evolución*, que utiliza unas reglas muy similares.

Al igual que ocurre con otros enfoques químicos, la expresión de la arquitectura depende directamente de la sintaxis de las moléculas. La gramática es distinta para cada sistema concreto; pero Wermelinger también acomete este problema con la definición de una notación –un lenguaje de moléculas– específica para la descripción de arquitecturas. Los componentes y enlaces ya no están entremezclados, como en el enfoque de Inverardi y Wolf, sino que se expresarán como moléculas independientes, lo que permite relacionar directamente cada reacción de la CHAM de evolución con una de las cuatro operaciones básicas de reconfiguración (§3.4.2).

Wermelinger utiliza el mismo ejemplo de un sistema Cliente-Servidor basado en un Gestor de enlaces (Clt/Srv/Mgr) descrito por Le Métayer y ya detallado en la sección 3.9.2. La clave es la sintaxis definida para las moléculas, que no solamente define componentes y enlaces, sino también *comandos* de reconfiguración, tales como *cc* y *rc*, que significan, respectivamente, *Crear-Componente* y *Borrar-Componente*. De este modo, el trabajo de la CHAM de evolución es muy sencillo: cuando un comando de este tipo coincide con una molécula-componente en la situación adecuada, se produce la reacción química que expresa la reconfiguración requerida. No son necesarios comandos que alteren los enlaces; la CHAM está diseñada de forma que se hace cargo de las moléculas que los expresan, en el momento en que se indica una creación o un borrado, tal como puede comprobarse en su definición.

$$
\begin{array}{rcl}
cc(m:M) & \rightarrow & c:C, c-m, cc(m:M) \\
cc(m:M) & \rightarrow & s:S, m-s, cc(m:M) \\
s:S, cc(m:M) & \rightarrow & s:S, m:M
\end{array}
$$

$$
\begin{array}{rcl}
cc(s:S), m:M & \rightarrow & s:S, m-s, m:M \\
rc(c), c:C, c-m & \rightarrow & \\
s':S, rc(s), s(S), m-s & \rightarrow & s':S \\
m:M, rc(m), cc(m':M) & \rightarrow & m':M \\
m-s, m':M & \rightarrow & m'-s, m':M \\
c-m, m':M & \rightarrow & c-m', m':M
\end{array}
$$

$$
cc(c:C), m:M \quad \rightarrow \quad c:C, c-m, m:M
$$

Figura 3.19: CHAMs de Creación y de Evolución para el Estilo Clt/Srv/Mgr

La Figura 3.19 expresa las dos CHAMs adecuadas para el ejemplo en discusión. A partir de aquí, Wermelinger [Wer99] demuestra la potencia de este enfoque extendiendo el mismo ejemplo para mostrar dinamismo arquitectónico (manipulación de enlaces) y capacidades de auto-organización, sin más que alterar las reglas que constituyen a la CHAM de evolución, y sin necesidad de afectar para nada a la CHAM de creación.

El propio Wermelinger expone que no toda arquitectura se puede describir de manera clara con este enfoque. Las condiciones negativas, en particular, son muy difíciles de expresar. No obstante, esta propuesta tiene el mérito de haber logrado unificar las de Le Métayer e Inverardi, tomando lo mejor de cada una, y produciendo un sistema más sencillo y potente.

3.11 Coordinación Dirigida por Datos

El término *coordinación* se utiliza para designar a una clase de modelos, formalismos y mecanismos para la descripción de actividades concurrentes y distribuidas. Básicamente, la coordinación

se logra bien mediante comunicación generativa, a través de un espacio de datos compartido, como ocurre en el caso de Linda y Gamma, o manipulando de manera dinámica las interconexiones entre los procesos, como consecuencia de la observación de sus cambios de estado, como ocurre en el caso de Manifold [Zav98].

Matthias Radestock [RE96] describió un modelo de cuatro capas para los sistemas distribuidos, en el que las dos superiores son la Configuración (que él identifica con la arquitectura) y la Coordinación. Según esta visión, la Configuración indica *dónde* se ha de producir la comunicación, mientras que la Coordinación, definida por encima, indica *cuándo*. Por su parte, las premisas de la Arquitectura 4C [AF01] describen otro modelo de cuatro capas, que invierte este orden. La Coordinación, definida ahora por debajo, hace posible la adición y eliminación de componentes; por encima se define la Configuración, que identifica las estrategias globales de reconfiguración. En realidad, se trata pues de cuatro abstracciones diferentes. Una auténtica descripción de la estructura de un sistema ha de proporcionar todos estos detalles: la simple comunicación y sus patrones, y la reconfiguración y los suyos. Reuniendo de este modo las cuatro abstracciones, se da lugar a lo que aquí se denomina *Arquitectura Dinámica*.

Existe una clara proximidad conceptual entre la Coordinación y la Arquitectura de Software, y es natural comparar la expresividad de los Lenguajes de Coordinación y los ADLs, o mejor aún, la de sus respectivos modelos. Se pueden considerar análogos en muchos sentidos [CFBSB02a], y a menudo han sido descritos en ambos contextos. De este modo, muchos modelos propuestos originalmente para Coordinación son igualmente aplicables a Arquitectura, y viceversa.

Existe un buen número de estudios recientes sobre la noción de coordinación en general [Cia97, BLM96, Arb98, MC94, ACH98, PA98a, PA98b]. En éstos se distingue habitualmente ente Modelos de Coordinación *Dirigidos por Datos*, o basados en Comunicación Generativa, y los *Orientados a Control*. En esta sección se revisan someramente los primeros; en la próxima se hará lo propio con los segundos. En ambos casos se podría realizar un tratamiento mucho más extenso; sin embargo, el único objetivo de estos apartados es situar estos trabajos y esbozar su relevancia para el contexto de la Arquitectura de Software Dinámica.

3.11.1 El Modelo Linda

La *comunicación generativa* se define como la comunicación asíncrona entre procesos, basada en una estructura de datos compartida [DMSMTH96]. Recibe este nombre debido al hecho de que cuando un proceso inicia la comunicación, no sabe ni a quién se la dirige, ni cómo llegar hasta él. Es el propio mensaje el que sabe encontrar (*generar*) este camino; dicho de otro modo, la información se recupera y comunica según *su contenido*. El origen y ejemplo arquetípico de comunicación generativa es el modelo Linda [Gel82, Gel85, CG86, GC92], concebido inicialmente como una extensión paralela para lenguajes de programación. El concepto ha evolucionado de manera notable, pero su esencia sigue siendo la misma.

La abstracción básica de Linda es el *espacio de tuplas*, a menudo mencionado simplemente como TS (*tuplespace*). Hace las funciones de medio de coordinación, y es una estructura de datos compartida en que contiene registros de múltiples campos, que se denominan *tuplas*. El modelo de Linda consiste, básicamente, en la definición de este concepto, y la de las cuatro primitivas que lo manejan. Las tuplas, en el modelo original, pueden ser pasivas (datos) o activas (procesos); este segundo caso se da cuando alguno de los campos es una llamada a función. Las tuplas

pasivas, las únicas que pueden ser accedidas, son secuencias finitas de campos; el acceso se realiza mediante un mecanismo de correspondencia de patrones (*pattern matching*).

Linda, en su forma original, consta únicamente de cuatro primitivas, que se combinan con un lenguaje anfitrión para hacerlo capaz de manipular las tuplas del espacio. Son:

- ***out(t).*** Se coloca una tupla pasiva t en el espacio de tuplas. Esta operación puede verse también como la *creación* de una tupla.
- ***in(t).*** Se extrae una tupla t del espacio, con el fin de *eliminarla*. La tupla se selecciona por su similitud con un patrón de partida, o *anti-tupla*, de manera no determinista.
- ***rd(t).*** También citado a menudo como ***read(t)***. Selecciona una tupla de manera similar, y lee su contenido en t, pero *no* elimina a la tupla del espacio.
- ***eval(t).*** Se crea una tupla activa, cuyo contenido se indica en t; al menos uno de sus campos ha de ser una función a ejecutar.

Dado que todas las operaciones son atómicas, nunca se puede producir por su causa ningún tipo de inconsistencia en el espacio de tuplas; por ello, resulta ser un modelo adecuado para la resolución de problemas de concurrencia, que es el objetivo para el que fue diseñado. Su naturaleza no determinista le hace aún más interesante, en especial porque se permite simular el determinismo sin más que dar un *nombre* a los procesos involucrados. Aunque su definición original es semiformal, se han desarrollado varios sistemas formales para Linda, que proporcionan una semántica precisa a sus operaciones; todos ellos se basan en las nociones y sintaxis de un álgebra de procesos. La primera formalización fue descrita por Paolo Ciancarini, y se conoce como *álgebra de Linda*, o *álgebra de comunicación generativa* [CGZ95b, CGZ96]. Se inspira directamente en CCS, y tan sólo cambian las operaciones. Posteriormente, Nadia Busi realizó un estudio aún más profundo, elaborando para ello un álgebra similar llamado *LinPA* [BGZ97b, BGZ97a, BGZ98]. Basándose en un un subconjunto de éste, denominado $\mathcal{L}$, Gianluigi Zavattaro pudo establecer formalmente la falta de equivalencia entre Linda y Gamma [Zav98], es decir, demostrar que las operaciones de Linda no son de naturaleza "química".

Existen numerosísimas variantes de Linda, que amplían el modelo o lo ven desde otra perspectiva. En un estudio reciente, Robert Tolksdorf [TRG02] ha identificado una veintena de propuestas sustancialmente distintas, y aún existen más. Así, se ha ampliado el conjunto de operaciones, considerando predicados y transferencias múltiples, como en Bonita [RW97]. El espacio de tuplas adquiere estructura; así, se trata como un objeto o un multiconjunto, y se define como simple o múltiple, plano o jerárquico. Bauhaus [CGZ95a], por ejemplo, utiliza un esquema de múltiples espacios de tuplas anidados. También se han tratado detalles tales como la distribución, la seguridad o incluso la temporización. Se ha considerado, además, la combinación del concepto con el marco proporcionado por otros entornos dinámicos, tales como los Sistemas Multi-Agente [OZ99] o el Modelo de Actores [Cal94].

Especialmente relevantes resultan los aspectos relacionados con la activación de procesos concurrentes desde el espacio de tuplas; éstos pueden llegar a alterar el estilo de comunicación generativa, y llegar a algo mucho más complejo. En primer lugar, están las distintas versiones de la primitiva *eval*, en sus vertientes funcional y orientada al objeto. Pero son más interesantes las reacciones provocadas mediante la emisión de eventos, como en JavaSpaces [FHA99], o definidas desde el propio espacio. Éste último es el caso del modelo TuCSoN [OD01, OZ99], que admite reacciones síncronas y asíncronas; del modelo OpenSpaces [DHN00], que las invoca *antes* de

cualquier operación sobre las tuplas; y, muy especialmente, del modelo de *Linda Gobernada por Leyes*, LGL [MR87, MR89, MUWZ96, MU97, Min97, MU99, Ung00, MU00], que permite la expresión de restricciones lógicas *globales* sobre el sistema, aplicadas desde el espacio, y que ha conocido su expresión más avanzada en el sistema Moses [MU97, MU00].

Es evidente que Linda no es un ADL. Lo que hace, más bien, es proporcionar una estructura en la que los componentes se establecen de manera independiente, y en la que la ligadura entre ellos viene dada por el propio modelo. En este caso, sí se trata de una *arquitectura de software dinámica*, ya que los componentes no están conectados entre sí –a menudo ni siquiera se conocen–, y la comunicación entre ellos se establece independientemente en cada interacción. No obstante, el potencial del modelo va más allá de la descripción de una arquitectura concreta; algunas de sus variantes más avanzadas han sido explícitamente presentadas y utilizadas como ADLs. Los casos más notables son los de LGL [MUWZ96, Min97] y MobiS [CM99], que tienen además cierto grado de semejanza. A continuación se describe este último con mayor detalle.

3.11.2 Los Sistemas PoliS y MobiS

PoliS –abreviatura de *PoliSpace*– es un modelo de coordinación híbrido, inspirado igualmente por la metáfora química de Gamma (§3.10) y el espacio de tuplas de Linda, y con influencias de la programación lógica. Fue desarrollado por Paolo Ciancarini [Cia91, CFM98], y es posiblemente el mejor estudiado de los modelos basados en *Espacios de Tuplas Múltiples Anidados*.

El modelo PoliS puede verse como una extensión del modelo básico de Bauhaus [CGZ95a], que se implementa sobre ESP (*Extended Shared Prolog*) [Cia93]. El sistema describe la totalidad del sistema como una colección de espacios de tuplas anidados, a los que se denomina *lugares*, y que son identificados mediante su nombre. En los lugares puede haber dos tipos de tuplas: las denominadas *tuplas-programa* o *agentes*, y tuplas normales, es decir, registros de un número finito de campos con un tipo y valor asociado.

Una tupla-programa es un proceso con su propio hilo de ejecución, y tiene dos partes diferenciadas: *cabecera* y cuerpo. La cabecera es una tupla convencional, mientras que el cuerpo es una secuencia de *operaciones de tuplas*, totalmente análogas a las del modelo Linda básico. Los agentes son completamente independientes entre sí, y pueden operar sobre las tuplas de cualquier lugar. Se activan cuando en un lugar dado coinciden una tupla-programa y una tupla normal, que se corresponde (*match*) con la cabecera de la primera. Una vez iniciado, el agente ejecuta la secuencia de operaciones indicada, y finalmente termina.

Tanto los agentes como las tuplas pueden ser enviadas hacia o recibidas desde otro lugar, que es referenciado mediante su nombre. Tanto la *comunicación entre lugares* como la *creación de nuevos lugares* es gestionada por un *Meta Espacio de Tuplas*. Éste es responsable tanto de encaminar los mensajes enviados al lugar correspondiente, como de almacenarlos cuando el lugar aún no existe –y, por supuesto, de entregarlos una vez que éste es creado–. Tanto los lugares como las tuplas que contienen pueden ser creados por los agentes de manera dinámica.

Hasta aquí, se ha descrito el comportamiento completo del paradigma, sin necesidad de remitirse al modelo químico, como si se tratase de una variante más de Linda. Sin embargo, es obvio cómo se puede interpretar en términos químicos: todas las tuplas de datos son *moléculas*, y las tuplas-programa son *reglas de reacción*. La cabecera de la tupla-programa es la condición, y el cuerpo es la acción; la ejecución de una tupla-programa constituye un paso de reescritura.

Las tuplas-programa se activan exactamente igual que una regla de reacción química: cuando se encuentra una molécula que cumple la condición indicada en su cabecera.

PoliS ha sido propuesto en más de una ocasión como un ADL por derecho propio [CM99]. Esto no resulta sorprendente, ya que su propia capacidad para manejar jerarquías de espacios invita a hacerlo. Resulta sencillo modelar un componente como un multiconjunto, y sus interacciones como reglas PoliS. Basta esto para ver cómo una jerarquía de espacios se convierte en una *jerarquía de componentes anidados*, esto es, una auténtica arquitectura de software, en la que las interacciones se desarrollan de manera menos explícita. Las capacidades dinámicas añadidas por el concepto de *meta espacio de tuplas* no son desdeñables en absoluto, y dan forma a una de las propuestas más cercanas a la de esta tesis.

Por su parte, **MobiS** es una variante de PoliS que resulta más apta para la expresión de *movilidad*. Ha sido desarrollado por Cecilia Mascolo [CM98, Mas99] bajo la supervisión de Ciancarini y, al igual que a PoliS, también se la ha utilizado para la descripción de *arquitecturas de software*, en este caso ya totalmente dinámicas.

Es prácticamente idéntico a PoliS, tanto en el modelo (múltiples espacios de tuplas anidados) como en la notación, la división entre tuplas y tuplas-programa o el mecanismo de activación de estas últimas. Su única diferencia es la presencia de un tercer tipo de tupla: la tupla-espacio, que representa a un *espacio de tuplas anidado*. Dicho de otro modo, en MobiS los espacios anidados son accesibles al programa en la forma de tuplas; por tanto, pueden ser manipulados libremente por las tuplas-programa, y creados, desplazados o destruidos al igual que cualquier otra tupla. Esto le proporciona la posibilidad de expresar movimiento, y uno de los mayores grados de dinamismo encontrados en un ADL existente.

3.12 Coordinación Dirigida por Control

La otra gran familia de la Coordinación muestra una cohesión menor, ya que no nace de un modelo patrón, como Linda, sino que se establece como una amalgama de métodos similares, pero con distinto origen. Se incluyen en este grupo lenguajes de muy diversas procedencias, identificados como lenguajes de programación concurrente, de configuración, de interconexión de módulos (MILS), e incluso ADLS. Por ejemplo, tanto *δarwin* (§3.7.2) como Rapide (§3.8.1) o Durra (§3.6.2) han sido considerados dentro de esta clase [PA98a]. No obstante, ésta agrupa también a otros lenguajes más específicos, como ConCoord [Hol96b], ColaS [CD99] o Toolbus [BK96]. Pero el ejemplo arquetípico, y la máxima expresión de este enfoque, es el modelo IWIM, encarnado en el lenguaje Manifold [PA01]; ambos se describen con mayor detalle a continuación.

3.12.1 El Modelo IWIM: Manifold

El modelo IWIM (*Idealized Worker Idealized Manager*: Trabajador Idealizado, Gestor Idealizado) [Arb96] es un modelo abstracto de comunicación, diseñado para construir sistemas en los que haya poca dependencia entre los procesos. Se trata de emular una situación en la que cada proceso –el *trabajador ideal*– intenta cumplir su tarea, sin preocuparse de la organización, y una serie de gestores –los *gestores ideales*– se encargan exclusivamente de manipular las relaciones

entre los procesos, es decir, sólo se ocupan de cuestiones de organización. Este modelo hace posible la separación de aspectos y la comunicación anónima, esto es: la separación estricta entre computación y coordinación, y que dos procesos puedan comunicarse sin saber nada el uno del otro. La interacción puede realizarse por difusión de eventos, o bien mediante a través de un canal punto a punto establecido por el gestor.

Manifold [AHS93, AHR93, PA97, Arb98, PA00] es un lenguaje desarrollado específicamente para construir a los gestores de este modelo. La idea es que los procesos pueden describirse en cualquier lenguaje, mientras que el lenguaje de los gestores tiene un funcionalidad muy concreta.

```
export manifold ejemplo()
    port in input;
    port out output;
{
    process A is tipo_A;
    process B is tipo_B;
    process C is tipo_C;
    start: (
        activate A, activate B, activate C; )
        do begin.
    begin:  (
        A → B, output → C, input → output).
    ev1: (
        B → input, C → A, A → B,
        output → A, B → C, input → output).
    ev2: ( C → B )
}
```

Figura 3.20: Descripción de un Sistema Reconfigurable en Manifold

Manifold es fuertemente tipado, estructurado en bloques y orientado a eventos. Sólo reconoce cuatro tipos de entidad: *procesos*, que son los trabajadores; *puertos*, que expresan sus entradas y salidas, y pueden ser incluso las propias; *eventos*, que se erigen como base del sistema; y *flujos* (*streams*), definidos como canales privados entre los puertos de dos procesos. No tiene estructuras de control (al menos, en primera instancia), por lo que ha de realizar toda su tarea mediante una dinámica de transición de estados dirigida por eventos, en la que se crean con total libertad nuevos procesos y conexiones entre ellos. De este modo, en respuesta a cada evento, se produce una reconfiguración en el sistema, especificada por uno de los gestores, denominados *coordinadores*. La Figura 3.20 muestra la estructura de un coordinador en Manifold.

Un proceso es una caja negra con puertos. Desde el punto de vista de otro proceso, no es posible distinguir a un coordinador de un trabajador; por tanto, un gestor puede estar coordinando a otros gestores, es decir, actuando como *meta-coordinador*. Por tanto, se pueden llegar a establecer protocolos de gran nivel de complejidad, y a múltiples niveles de abstracción.

Manifold ha sido siempre un lenguaje con gran capacidad dinámica, dado que su única función es, precisamente, hacer reconfiguraciones como respuesta a eventos. Aunque su uso en

este sentido era obvio desde hace tiempo, sólo ha sido presentado como un ADL en fechas muy recientes [PA01]. Se constituye, en cualquier caso, como uno de los enfoques para Arquitectura de Software Dinámica más versátiles y competentes.

3.13 Otros Lenguajes

Aparte de todos los descritos, existe un buen número de sistemas, lenguajes y modelos que podrían ser considerados de un modo u otro como ADLs dinámicos. Una enumeración exhaustiva abarcaría desde lenguajes de programación concurrente hasta sistemas de eventos distribuidos, e incluso algunos modelos formales de concurrencia. Algunos de ellos, tal vez de manera impropia, han sido incluso mencionados explícitamente como ADLs, aun cuando no todos cumplen los criterios de Medvidovic [MT00], ni aun otros menos restrictivos. No obstante, algunos muestran rasgos interesantes, por lo que serán mencionados de la manera más breve posible.

3.13.1 Lógica de Reescritura

La lógica de reescritura [Mes92] es un sistema formal desarrollado sobre el concepto de institución lógica, como una evolución de los modelos de especificaciones algebraicas, y en particular de OBJ. Su característica fundamental es la adición explícita de reflexión [CM96a, CM96b, MOM96], que le dota de un elevado grado de generalidad. Se establece como un modelo de *concurrencia verdadera*, y en este contexto ha demostrado ser capaz de subsumir en su interior prácticamente cualquier otro modelo existente [Mes96]. En realidad, la Lógica de Reescritura es una familia de lógicas; esta sección se centra explícitamente en la que utiliza lógica de membresía en su parte ecuacional, y se considera cuando es preciso que se dispone de la variante condicional.

La Arquitectura de Software es uno de los entornos en los que se ha pretendido demostrar la aplicabilidad general de la Lógica de Reescritura. En este sentido, José Meseguer y Carolyn Talcott han elaborado una propuesta [DM98, MT97c, MT97b] para la especificación de *dinamismo* arquitectónico, que justifica su inclusión en este apartado. En realidad, no se está planteando la definición de un auténtico ADL, sino que sólo se está proponiendo el uso de un formalismo ya existente para el problema de la descripción arquitectónica.

Las tres partes del método pueden resumirse como sigue. En primer lugar, se define la Lógica de Reescritura como un marco semántico común, al que se pretende *traducir* las especificaciones realizadas en un ADL clásico. Dada la generalidad de la lógica, se asume que existe una versión en su interior del formalismo base del ADL utilizado; por ejemplo, una descripción Wright se traduce a CSP, y de ahí a la lógica. De este modo, se reduce todo sistema a un lenguaje común y un tratamiento uniforme: por tanto, la Lógica de Reescritura se está usando como lenguaje de intercambio. En segundo lugar, se considera el formato utilizado en esta traducción. éste se concibe de modo que coincide con la formulación del Modelo de Objetos Distribuidos [Mes90], que es también el de Actores [Tal96]. De este modo, se tiene una representación uniforme para los componentes arquitectónicos dentro de la lógica. En tercer lugar, por fin, se puede manipular libremente el esquema resultante con las reglas de la lógica; de este modo se pretende expresar comportamiento, e incluso proporcionar *dinamismo*. El proceso es, en teoría, reversible. Sin embargo, la reducción de cualquier ADL a una forma unificada supone pérdida de información estructural, por lo que este extremo no resulta en absoluto trivial.

Este enfoque tiene un inconveniente obvio: para realizar todo el proceso, se reduce toda la especificación a una sintaxis de bajo nivel. La visión global, que era precisamente el objetivo principal de todo el proceso, se pierde casi por completo. Por tanto, esta aproximación puede utilizarse, en el mejor de los casos, como un modelo semántico de gran potencia; pero no resulta adecuada para descripción arquitectónica *directa*, al menos mientras no se le asocie algún tipo de descripción *de alto nivel*, más cercana a la tradición de los ADLs clásicos.

3.13.2 El Modelo de Actores

Se trata de un modelo teórico de innegable importancia, establecido originalmente por Carl Hewitt [Hew77] y ampliamente desarrollado, tanto formalmente como en la práctica, por múltiples autores, entre los que destaca Gul Agha [Agh97]. Su relación con la Arquitectura de Software es ante todo implícita, y abarca múltiples aspectos; no sólo se describe como el fundamento teórico de propuestas relacionadas [VM99], sino que los patrones que define aparecen, por sí solos, al desarrollar otros modelos [Mes90, PT00, Sch99].

A lo largo de los años se han descrito múltiples variantes del modelo de actores, algunas de las cuales podrían haberse planteado como ADLs por derecho propio. Esto incluye a los ActorSpaces de Callsen [CA94, Cal94], y a los *sincronizadores* y *conjuntos de mensajes* de Svend Frølund [FA93, FA96]. Ambas propuestas tienen el mismo fundamento teórico, que consiste en el establecimiento de una infraestructura sobre el modelo básico, concebida como una meta-arquitectura definida por *meta-actores* [Tal96]. Sin embargo, no se trata de un modelo reflexivo, sino de un esquema en dos niveles, particularizado siempre en un estilo concreto.

Éste es también el fundamento del trabajo de Mark Christopher Astley [Ast99], que tiene como objetivo principal la especificación de estructuras distribuidas. Para ello, define una arquitectura estructurada en *grupos de actores*, cada uno de los cuales tiene asociado uno o varios meta-actores. Algo similar ocurre con las conexiones entre ellos, a las que denomina explícitamente *conectores*. Sobre esta estructura, Astley define un lenguaje denominado DCL (*Distributed Connection Language*), que establece las conexiones entre *módulos*, indicando los *protocolos* entre ellos, así como las *políticas* de configuración aplicables, estructuradas en actores.

Un desarrollo lógico de las ideas de Astley hubiera mostrado un buen número de coincidencias con el enfoque presentado en esta tesis, aunque limitándose a un meta-nivel. Sin embargo, lo que se hace realmente es describir *una* arquitectura concreta, sobre la que se establece DCL como lenguaje de configuración. No desarrolla todo su potencial, y alcanza únicamente dinamismo estructural. Aunque Astley lo concibe como un ADL, tal vez no es lo bastante abstracto, ya que queda demasiado ligado a un entorno de programación, el *Java Actor Foundry*.

Por otra parte, cabe mencionar que el propio Astley ha evolucionado con posterioridad, elaborando a partir de DCL un nuevo lenguaje, denominado DIL [ASA01]. En éste se mantiene la perspectiva basada en protocolos y la infraestructura de meta-objetos, aunque se aumenta el énfasis en el tratamiento de eventos. Sin embargo, el lenguaje no se describe ya como un ADL; de hecho, es prácticamente un lenguaje de programación, integrado en un entorno específico de actores, llamado *Broadway*.

3.14 Modelos de Dinamismo Arquitectónico

En este apartado se presenta una breve clasificación de los distintos enfoques existentes para la descripción de arquitecturas de software dinámicas, basada en este caso ya no en el origen, sino en la *arquitectura* interna de dichas propuestas, concretamente en los aspectos referidos al modo en que se introduce el dinamismo.

La clasificación considera a los sistemas a alto nivel, ya que su objetivo es identificar las coincidencias entre los enfoques de los distintos lenguajes, abstrayendo los detalles menores. No ha de considerarse, sin embargo, una clasificación rígida, ni una propuesta definitiva. Su objetivo principal es agrupar conceptualmente las distintas aproximaciones para la descripción de dinamismo, con el fin de facilitar su comparación y discusión. En principio, parece claro que ningún ADL, haya sido o no explícitamente considerado, queda fuera de ella. Esto significa que, en última instancia, su estructura interna –su *meta-arquitectura*, en definitiva– podrá ser identificada como perteneciente a alguna de las seis categorías definidas.

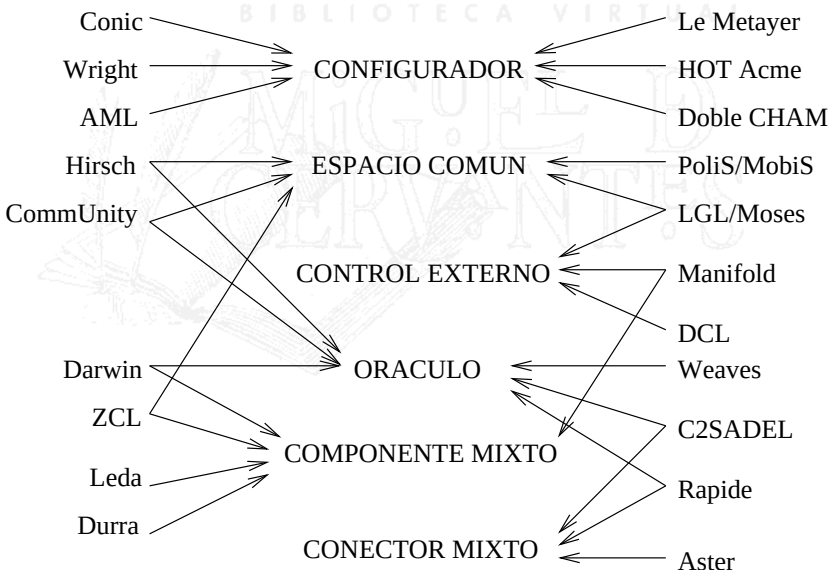

Figura 3.21: Modelos de Dinamismo en ADLs

La Figura 3.21 muestra en cuáles, de entre los modelos aquí descritos, podrían enmarcarse cada uno de los ADLs considerados en los apartados anteriores. Como ya se ha dicho, la asignación no ha de considerarse rígida, ni estricta. Por otra parte, algunos de ellos manifiestan simultáneamente más de una aproximación, debido a la riqueza del propio lenguaje, o de su entorno: en ese caso, aparecerán vinculados a más de un modelo.

Pasemos, pues, a hacer una descripción detallada de los seis modelos identificados, que a su vez podrán agruparse en tres bloques:

1. **Supervisor**. La característica que define a este grupo es la presencia, explícita o implícita, de algún tipo de entidad "omnisciente", que gestiona la dinámica de la arquitectura, y

decide cómo y cuándo han de realizarse los cambios relevantes. Este enfoque es, con mucho, el más utilizado, pues utiliza una perspectiva centralizada que resulta mucho más evidente que otros enfoques. No obstante, en esta centralización se adoptan estrategias diversas, que permiten identificar tres grupos diferentes, hasta un total de cuatro modelos distintos.

(a) **Configurador**. En este grupo, el supervisor se modela de manera explícita, y se incluye como un componente *especial* en la arquitectura. Este componente tiene sólo una función: modificar y gestionar la propia arquitectura. Ejemplos de este supervisor son el *configurator* en las gramáticas de grafos de Le Métayer (§3.9.2), el *configuror* en la versión dinámica de Wright (§3.7.1), el *changer* de Stuurman [SVK98], el tipo *architecture* en Hot Acme, la *máquina de evolución* en el modelo de doble Cham de Wermelinger (§3.10.2), o el bloque *manage* de Conic (§3.6.1). El caso de Aml (§3.7.5) es especial, pues usa una aproximación diferente; pero sus meta-descripciones son explícitas, por lo que se incluyen en este grupo.

En la Figura 3.22 se muestra un esquema simplificado de este modelo de dinamismo. En éste, los iconos en forma de caja representan a componentes, los círculos a conectores y las líneas a conexiones directas. El componente marcado es el que expresa dinamismo: en este caso, el supervisor. Las líneas de trazos expresan la influencia de éste sobre otros elementos del esquema.

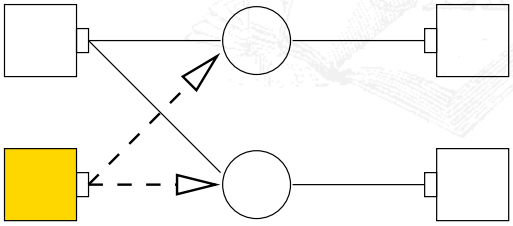

Figura 3.22: Configurador

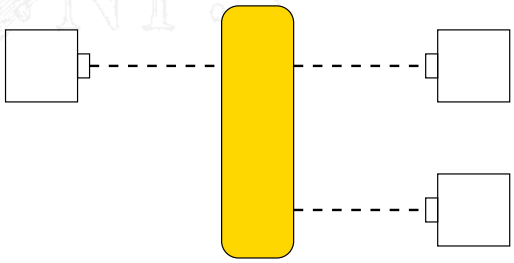

Figura 3.23: Espacio Común

(b) **Coordinación**. En este modelo se supone la existencia de un ámbito o espacio externo, con ciertas *reglas* que condicionan la evolución dinámica de la arquitectura. El supervisor que se encarga de hacer cumplir estas reglas es ahora, sin embargo, *implícito*: a diferencia del grupo anterior, no se puede identificar un único elemento que ejerza esta función. Este enfoque es complejo, y puede a su vez ser subdividido en dos enfoques distintos, aunque relacionados:

- **Espacio Común**. En este esquema la coordinación se realiza sobre un ámbito común, sea a toda la arquitectura o a alguna de sus partes. Toma la forma de una región de comunicación, o de la jurisdicción de un conjunto de reglas. Por tanto, *no* se modela explícitamente. El ejemplo obvio son las propuestas basadas en espacios de tuplas, lo que engloba a todas las variantes de Linda (§3.11.1), y en especial a Lgl [MUWZ96] y los modelos PoliS y MobiS (§3.11.2). También se han incluido en este grupo los modelos en los que las reconfiguraciones se aplican

como un conjunto de normas genéricas, no asociadas a ningún elemento, como ocurre con las *transformaciones* de Hirsch (§3.9.3) y las reglas de reconfiguración por superposición de programas CommUnity de Wermelinger (§3.9.4).

La Figura 3.23 muestra una simplificación de este esquema, en la que se muestra un espacio "dinámico" cuya influencia sobre los distintos componentes, y en particular sobre su interacción, es siempre indirecta.

- **Control Externo**. Este grupo engloba a los esquemas en los que no existe un supervisor explícito, pero el dinamismo sobre el sistema sí es modelado, y controlado por una entidad externa. Por ejemplo, en Manifold (§3.12.1) el dinamismo es consecuencia de una emisión global de eventos, pero se distribuye entre los coordinadores; y en Dcl (§3.13.2) el control sobre los protocolos se modela de manera genérica, pero se distribuye en meta-actores.

  La Figura 3.24 muestra el esquema de este modelo, en el que algún tipo de entidad externa condiciona y controla de manera activa la evolución, y en especial la interacción, de los componentes.

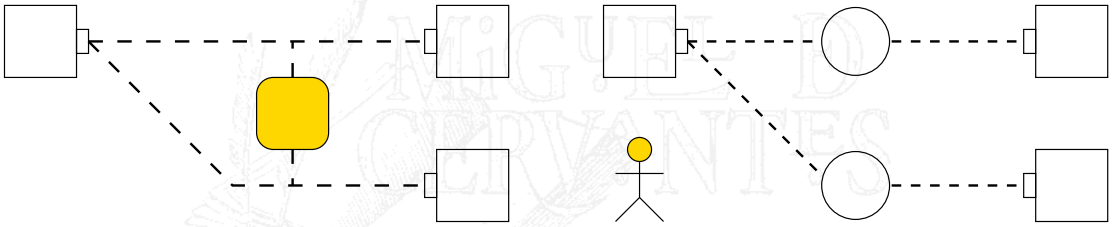

Figura 3.24: Control Externo

Figura 3.25: Oráculo

(c) **Oráculo**. En este enfoque, la entidad externa es completamente ajena al sistema y no puede ser modelada. Puede intervenir de en cualquier momento y de cualquier modo, por lo que su influencia es impredecible. Si pudiese incluirse en un sistema, tendría el papel de un *oráculo*, en el sentido de Turing. Este grupo se considera por completud, reuniendo todas las soluciones de reconfiguración *ad hoc*, basadas en la intervención humana. Esto incluye la actividad de herramientas software, como ArchShell y Dradel en C2sadel (§3.8.2), Sysman en *δarwin* (§3.7.2) o Jacquard en Weaves [GQ94]; pero también puede referirse a la aplicación de modelos puramente teóricos, como en los casos ya mencionados del modelo de Hirsch y el esquema categórico sobre CommUnity de Wermelinger.

La Figura 3.25 expresa la presencia de este oráculo sobre un esquema arquitectónico, indicado con el mismo icono que un actor en los casos de uso de Uml.

2. **Componente Mixto**.

En este grupo, la definición del Adl proporciona a los componentes una serie de primitivas que les permiten alterar la arquitectura del sistema, o al menos una parte de ella. Esto significa que los componentes realizan una tarea doble, ya que aparte de su propia computación se encargan de tareas de reconfiguración. Los ejemplos de esta clase incluyen algunos de los Adls con mayor capacidad para el dinamismo, como *δarwin* (§3.7.2) o

Leda (§3.7.3) o Zcl [dPJC98], que no hace mixtos a los componentes, sino a las configuraciones. Se puede considerar también a Manifold (§3.12.1) dentro de este grupo, por el papel ambivalente de sus coordinadores, que sin embargo no son exactamente mixtos.

La Figura 3.26 muestra un posible esquema de este tipo, en el que se considera que cada componente puede tener una parte dinámica; ésta podrá afectarle a él mismo, o al resto de elementos. Se ha dibujado un modelo sin conectores, que es el caso más típico, pero éstos podrían ser igualmente considerados.

3. **Conector Mixto**. Este grupo reune las propuestas que provocan cambios en una arquitectura utilizando o definiendo alguna característica de su medio de comunicación, y en particular de los conectores. Muestra rasgos comunes con la clase denominada *Espacio Común*, pero se distingue por evitar una aproximación centralizada. Es sin duda el grupo más difícil de clasificar. Aquí se incluyen los *sistemas de eventos* en los que se impone algún tipo de efecto dinámico, como Rapide (§3.8.1), pero también los sistemas con un *bus común*, como Polylith [Pur90] o Aster [IB96, BBI+00], e incluso una mezcla de ambas ideas, encarnada en los conectores de C2Sadel (§3.8.2) y su reflexión contextual.

La Figura 3.27 muestra un esquema que intenta resumir este enfoque, de modo que uno de los conectores tiene un comportamiento dinámico, que incluso puede afectar al resto. Sin embargo, a diferencia del segundo grupo (Figura 3.23), el medio está estructurado, y puede incluir conectores no dinámicos.

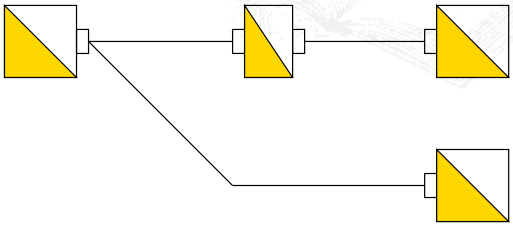

Figura 3.26: Componente Mixto

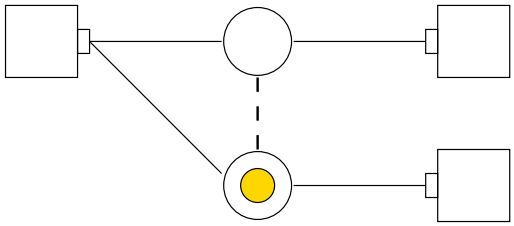

Figura 3.27: Conector Mixto

Esta clasificación no pretende ser definitiva, aunque ha sido publicada con anterioridad en dos ocasiones [CFBSB00, CFBSB01]. Sin embargo, proporciona un criterio coherente para agrupar los distintos enfoques utilizados, lo que posteriormente facilitará la discusión de sus equivalentes reflexivos: de este modo se justificará la afirmación –hipótesis de trabajo de esta tesis– de que todo sistema de dinamismo arquitectónico existente puede ser descrito en términos de Reflexión. Esta tarea, no obstante, se difiere hasta la sección 5.2, de modo que antes (§4) se proporcione una base adecuada sobre los conceptos reflexivos.

3.15 Conclusiones

A lo largo de este Capítulo se han revisado los distintos enfoques propuestos para la descripción de Arquitectura Dinámica, y han sido clasificados, atendiendo tanto a su origen conceptual

como a la estructura interna de los sistemas propuestos. En general, se puede apreciar una gran diversidad: pero también existen una serie de coincidencias significativas entre todos ellos.

Aparentemente, la clasificación según el origen resulta mucho más dispersa. Sin embargo, un estudio más detallado revela que, en realidad, las distintas sintaxis suelen coincidir en *dos* estrategias básicas. En primer lugar, las comunicaciones se expresan, en general, con una forma análoga a la de un álgebra de procesos: esto no es sólo cierto en los ADLs algebraicos, sino también en los modelos químicos, e incluso en algunos de los basados en grafos (§3.9.2, §3.9.3). En segundo lugar, la mayoría de las reconfiguraciones se expresan como una reescritura; esto es puede verse en los sistemas gráficos, los químicos y el basado en lógica de reescritura; pero también en la versión dinámica de Wright (§3.7.1). Tan sólo los sistemas de eventos parecen tener un esquema independiente, posiblemente debido a que plantean la concurrencia desde un enfoque totalmente *dual* [Pra95] al del resto de los sistemas descritos.

La clasificación de los modelos de dinamismo, por otra parte, muestra un claro conflicto entre la *centralización* y la *distribución* del control del dinamismo. Así, los modelos basados en un supervisor manifiestan un control central, mientras que los basados en componentes mixtos resultan totalmente distribuidos; éste es también el caso de los conectores mixtos, donde el control no se ubica en ningún punto concreto de la estructura, sino en los espacios que ésta define. Además, el conflicto se repite en las subdivisiones del esquema de supervisor: de este modo, el configurador es totalmente centralizado, al igual que el espacio común, que simplemente tiene menor cohesión; en cambio, el modelo de control externo distribuye de nuevo el control entre múltiples agentes, cada uno de los cuales ejerce un control centralizado sobre su área de influencia. Parece evidente, en definitiva, que la estructuración de cada posible alternativa permite identificar conflictos similares.

En definitiva, existe una gran diversidad de aproximaciones para la descripción de arquitecturas dinámicas, cada una de las cuales muestra una potencia expresiva muy diferente; sin embargo, existe también toda una serie de coincidencias significativas a bajo nivel entre ellas, lo que pone de manifiesto una fuerte coherencia interna en el campo.

Parte III

Reflexión

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

Capítulo 4

El Concepto de Reflexión

κρητες αει πσευσται
Epiménides de Creta

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

4.1 Introdu

El concepto de Re exión– es una de las

[1]Existe cierto debate en torno a la traducción, en este contexto, del sustantivo inglés *reflection* y sus derivados, el adjetivo *reflective* y el verbo *to reflect*. En primera instancia, parece indiscutible que el sustantivo se debe traducir, tal como se ha hecho, como *reflexión*. No sólo es la voz más natural, sino que además mantiene el juego de palabras, igualmente existente en inglés, con el resto de sus significados. Son aquí relevantes –por sus asociaciones autorreferenciales– tanto el relativo al pensamiento, que sugiere *introspección*, como el habitual en óptica y física, que connota *retorno* y repetición. No obstante, en este punto surge ya el conflicto, al considerar el verbo correspondiente. En el primer sentido, éste es *reflexionar*, mientras que en el segundo son correctos tanto *reflejar* como *reflectar*, si bien éste último es considerado más técnico. Estas tres formas se traducen, además, como el mismo verbo en inglés, precisamente el citado *to reflect*. La única razón para elegir alguna de ellas es la metáfora asociada a *reflejar* como "dejarse ver una cosa en otra", que se ajusta perfectamente al significado deseado, y de hecho es justamente la razón por la que Smith [Smi82] utilizó este término.

Por otra parte, en principio sólo hay una traducción posible del adjetivo, esto es, *reflexivo*. Parece ideal, pues no sólo se aplica a las dos connotaciones consideradas, sino que añade por su cuenta otras dos igualmente válidas: la de los verbos reflexivos, en los que el sujeto actúa sobre sí mismo, y la de la propiedad reflexiva del álgebra, en la que un elemento se relaciona consigo mismo. Sin embargo, aquí aparece un nuevo conflicto, ya que mientras las dos primeras se traducen al inglés como *reflective*, las dos últimas lo hacen como *reflexive*, manteniendo una forma más arcaica. Esto permite establecer en inglés una distinción [MNS95] que en español no existe, ya que la calidad de reflexivo (*reflexiveness*) se realiza en un único plano, mientras que la reflexión (*reflection*) siempre implica al menos dos, incluso cuando actúa como principio metalógico [Har95]. Con ánimo de recuperar este matiz, para el segundo caso se ha utilizado a menudo el adjetivo *reflectivo*, inexistente en castellano. Esta palabra sugiere un sustantivo claramente inadecuado (*reflección*), por lo que en su lugar se ha empleado otro término inexistente, *reflectividad*. Tanto éste como el adjetivo han conocido, sin embargo, cierta difusión. La confusión es aún mayor, pues por sobrecorrección este último ha degenerado ocasionalmente en *reflexividad*, carente de sentido en tanto que pierde incluso la distinción buscada.

Etimológicamente, tanto en español como en inglés, estas palabras proceden del verbo latino *reflectĕre*, "volver hacia atrás", de cuyo supino, *reflexum*, se deriva el sustantivo *reflexĭo*. Por tanto, las dos raíces estaban ya presentes en latín, lo que hace patente que su evolución natural en español es el primer término indicado. Por esto, y por todo lo dicho, en esta tesis se usarán las traducciones *reflexión, reflexivo* y *reflejar*, excepto allí donde explícitamente se indique lo contrario.

nociones fundamentales que se manejarán a lo largo de esta tesis, por los motivos ya expuestos en la Introducción. Se trata de un concepto con una larga historia, definido por Brian C. Smith a principios de la década de los 80, y que hace referencia a los sistemas que *tratan y actúan sobre sí mismos*. Tradicionalmente, ha estado vinculado al ámbito de la Inteligencia Artificial, en el contexto de los lenguajes funcionales –Lisp, en especial– y declarativos –donde destacan Prolog y sus variantes, y Gödel–, así como a los sistemas basados en reglas.

No obstante, los trabajos de Pattie Maes [Mae87a] han conseguido popularizarlo fuera de ese ámbito, y desde entonces ha conocido una difusión muy rápida. En los últimos quince años se ha aplicado de manera intensiva en el diseño de Lenguajes de Programación y Sistemas Operativos, así como en la Orientación al Objeto; pero ha demostrado también su importancia en muchos otros campos, y todavía no se ha expandido por completo.

En los últimos tres años, específicamente, el desarrollo del concepto ha experimentado un impulso aún mayor, debido a su relación con otros campos que lo utilizan de algún modo. Esto incluye a modelos y marcos de integración de componentes, tales como Corba, .Net o incluso el entorno de Java; a los propios sistemas de *middleware* reflexivo; o al auge de los modelos basados en aspectos; así como a la normalización y asimilación de nociones vinculadas como la de *metadato* y *metamodelo* –incluido de manera explícita en Uml–. Este renacimiento ha traído consigo, además, la recuperación de algunas ideas relacionadas, como las de prototipo y superposición, que habían decaído durante la última década.

Este Capítulo contiene tan sólo una introducción muy general al campo de la Reflexión, y su intención es proporcionar una mínima base conceptual para sustentar la estructura y argumentos definidos en el Capítulo 5. No pretende ser un resumen completo del tema, dado que esto no es preciso y su extensión sería, además, intolerable; en particular, se ha evitado entrar en aspectos específicamente relacionados con programación, puesto que la mayor parte de ellos no resultan relevantes en este contexto. Por tanto, hay toda una serie de conceptos adicionales, que no serán tratados, por no estar vinculados ni ser transportables al modelo conceptual de la Arquitectura de Software. Para una adecuada revisión de los mismos, esta tesis se remite a los ya numerosos estudios generales existentes sobre el tema. Algunos de los más interesantes son [DM95, MJD96, Caz98, OCGB98, Ste94, KMS98, Gol97, CM96a, CM96b, BL97b, Sie92, BL97a, DS99, BC98, Mar01], aunque hay muchos otros. Algunos de ellos se centran en un área o modelo particular, o en un aspecto concreto; pero el conjunto formado por todos ellos proporciona una visión que ninguno de ellos da en solitario.

4.1.1 Definición

En este apartado se dará una definición inicial de reflexión, con el objeto de situar a este capítulo en contexto. No obstante, ninguna de ellas se ha de considerar concluyente. La definición más completa se difiere hasta la sección 4.3.1, momento en el cual ya se habrán introducido algunas de las consideraciones previas necesarias.

El concepto de Reflexión es en realidad muy simple; se puede dar una definición muy concisa que proporcione una idea bastante aproximada. Ésta es la siguiente:

> ***Reflexión*** *es el proceso de razonar y actuar sobre uno mismo.*
>
> *Takuo Watanabe & Akinori Yonezawa [WY88]*

Su principal ventaja –e inconveniente– es la de no restringirse al ámbito informático. En líneas generales, resulta bastante imprecisa. En su lugar, se prefiere la siguiente, debida a Pattie Maes [Mae87a], verdadera responsable de la actual difusión del concepto:

> ***Reflexión Computacional*** *es la actividad desarrollada por un sistema computacional al realizar computaciones sobre –y, posiblemente, afectando a– su propia computación.*
>
> *Pattie Maes [Mae87a]*

Que se concreta luego, más detalladamente, como sigue:

> *Definimos* ***reflexión computacional*** *como el comportamiento que exhibe un sistema reflexivo; donde un* ***sistema reflexivo*** *es un sistema computacional que trata sobre sí mismo, manteniendo una* ***conexión causal****.*
>
> *Pattie Maes [Mae87a]*

Esta definición es también muy general; no obstante, se ubica ya en un contexto computacional y tiene varios puntos notables, en particular la mención a la *conexión causal*, que expresa la relación existente entre un sistema reflexivo y la parte de él mismo que lo utiliza como objeto de su computación. Esta idea proporciona ya una primera estructuración del concepto, que hasta este momento era estrictamente autorreferencial; y proporciona la base para todo su desarrollo posterior, que será tratado a lo largo de la sección 4.3 –y posteriores–.

4.2 El Problema de la Autorreferencia

El pensamiento científico y filosófico tradicional ha rechazado siempre los razonamientos circulares; en concreto, los problemas vinculados a la *autorreferencia* se han considerado particularmente patológicos, expresiones sin sentido[2] que sólo expresaban, en palabras de Bertrand Russell, un *círculo vicioso*. Y sin embargo, es improbable que exista jamás otro concepto tan fecundo en cualquiera de sus formas y variantes, y de tan amplias consecuencias e implicaciones en todos los órdenes del pensamiento moderno, en especial en el último siglo.

En realidad, la autorreferencia no es en sí misma una fuente de contradicción; de hecho, en ciertos contextos y en teorías científicas de todos los órdenes, su aparición resulta inevitable [DCS76]. Sin embargo, es claro que siempre debe plantearse dentro de unos límites, los que vienen marcados por resultados tan fundamentales como el teorema de incompletud de Gödel, el de indecibilidad de Turing, o el de la verdad de Tarski. Los sistemas tradicionales la rechazan en gran medida; otros, como las lógicas paraconsistentes y parciales, los principios de reflexión de Kreisel, la teoría libre de tipos de Feferman o la de la verdad de Kripke, admiten su presencia, pero controlan cuidadosamente sus consecuencias.

La autorreferencia se encarna, a grandes rasgos, en tres nociones fundamentales: la *realimentación*, la *recursión* y la *reflexión*. Otras ideas básicas, como la auto-organización, la repetición o la autorreplicación, no son sino variantes de éstas. Con el objetivo de controlarla y acotarla

[2] Como la Paradoja del Mentiroso o $\Pi\sigma\epsilon\upsilon\delta o\mu\epsilon\nu o\varsigma$, formulada por Eubúlides de Mileto, que consiste simplemente en la oración *"Esta oración es falsa"*, a la que no se le puede asignar un valor de verdad consistente.

surge una cuarta noción fundamental: la de *tipo*, estrato u *orden* –todas ellas variantes de la misma idea–, que por lo tanto se vincula también fuertemente a este contexto.

La diferencia entre *recursión* y *reflexión* es muy sutil, y no es extraño que con frecuencia ambos conceptos hayan sido confundidos, ya que están fuertemente interrelacionados. En ambos casos se tiene el mismo patrón de autorreferencia; pero así como el primero tiene una naturaleza *sintáctica*, el segundo tiene una naturaleza *semántica*. El primero en hacer esta distinción fue Frank Ramsey, cuando identificó dos tipos diferentes de paradojas, a las que denominó respectivamente *lógicas* –sintácticas– y *epistemológicas* –semánticas–.

De igual modo, posiblemente la mejor forma de describir la diferencia entre ambos conceptos sea apoyándose en dos paradojas muy similares, cada una de las cuales se corresponde con uno de los conceptos discutidos. Son las siguientes [FML55]:

Paradoja 4.1 (de las Propiedades o de Russell) *Se dice que una propiedad es* predicable *si se aplica a sí misma, e* impredicable *en el caso contrario. La propiedad de ser impredicable, ¿es predicable o impredicable?*

Por ejemplo, la propiedad de ser concebible es concebible, y por tanto es predicable, mientras que la propiedad de ser verde no es verde, y por tanto es impredicable. Esta paradoja no es más que otra versión de la célebre paradoja de Russell sobre la teoría de conjuntos; del mismo modo que ésta, queda explícitamente prohibida por las teorías de tipos convencionales, con las que se relaciona en más de un sentido [KL01]. Puede ser planteada en términos de lógica de orden superior, produciendo obviamente un resultado inconsistente.

Paradoja 4.2 (de la Denotación o de Grelling) *Se dice que un adjetivo es* homológico *si se describe o denota a sí mismo, y* heterológico *en el caso contrario. El adjetivo "heterológico", ¿es homológico o heterológico?*

Por ejemplo, el adjetivo "español" está en español, luego se describe a sí mismo, y por tanto es homológico; mientras que el adjetivo "inglés" no está en inglés, y por tanto es heterológico. Esta paradoja no es formulable en términos de una lógica de orden superior, sino que sólo puede ser planteada en lenguajes –lógicas– con soporte para *nominalización* [Kam92]; tampoco puede ser evitada por una teoría de tipos, por lo que para evitarla es preciso construir una *jerarquía de lenguajes* –sugerida en primer lugar por Russell [Rus22] y desarrollada después por Alfred Tarski [Tar35]–, que permita hacer distinción entre un lenguaje y un meta-lenguaje.

La diferencia entre recursión y reflexión, por tanto, es la misma que hay entre una propiedad impredicable y un adjetivo heterológico: mientras que los primeros se refieren a los conceptos *en sí*, los segundos se refieren a su *nombre*, esto es, a la palabra con la que se los designa. La diferencia puede parecer trivial, pero de hecho es absolutamente fundamental. De hecho, este detalle es precisamente la clave del celebérrimo artículo de Kurt Gödel [Göd31] sobre la incompletud de la aritmética. En la primera parte del artículo, Gödel identifica el concepto de recursión, y define cualquier función computable en términos recursivos. En la segunda parte, sin embargo, y apoyándose en la estructura recursiva definida, introduce el concepto de *aritmetización*, mediante el que construye un número que representa a un predicado que habla sobre sí mismo. Este predicado es, por tanto, reflexivo, y el modo en que evita producir una inconsistencia es, precisamente, el que demuestra la incompletud.

Es significativo señalar que la diferencia entre ambas es en realidad la clásica *distinción uso/mención* de la lógica: un objeto no es referenciado del mismo modo si él mismo está involucrado (*uso*) en un acto, que cuando sólo se alude a él de manera indirecta (*mención*). La distinción parece trivial, pero es fundamental: se encuentra, entre otros aspectos, en el núcleo de las lógicas modales, donde plantea un conflicto [JF90].

Como se ha dicho, el intento de controlar estos aspectos ha derivado en el desarrollo de las teorías de tipos. La noción es debida a Bertrand Russell, pero su desarrollo actual deriva más bien de la Teoría Simple de Tipos de Alonzo Church [Chu40]. Consiste, básicamente, en una estratificación de los valores utilizados en un término lógico, de tal modo que éste sólo puede utilizar como parámetros a valores de un tipo inferior, hasta llegar a un tipo o tipos primitivos. De hecho, se ha afirmado [KL01, Per88] con fundamento que las teorías axiomáticas de conjuntos –como la ZFC de Zermelo y Frænkel, la de Ackermann, la NBG de von Neumann, Bernays y Gödel, la NF de Quine y Wang, o la más reciente de Aczel– pueden ser consideradas también como teorías de tipos. En definitiva, se distinguen lógicas y teorías de tipos de primer orden, que sólo admiten manipulaciones sobre tipos básicos; de segundo orden, que admiten actuar sobre un tipo indefinido; o de orden superior, que pueden desplegarse indefinidamente.

No deja de ser relevante resaltar que la noción de *órdenes* fue también propuesta por Russell como parte de la Teoría Ramificada de Tipos [Rus08], con anterioridad a todos estos planteamientos, y en esta definición se diferenciaba explícitamente de la de *tipos*. Sin embargo, Frank P. Ramsey demostró [Ram25] que el concepto de orden no era estrictamente necesario para evitar antinomias, por lo que fue excluido –aunque no ignorado– por Alonzo Church al desarrollar la Teoría Simple de Tipos. Esto ha supuesto su ausencia de los sistemas de tipos habituales, lo que no ha impedido que se haya seguido utilizando de manera implícita, aunque siempre de modo informal. No obstante, tiene una influencia decisiva sobre la *predicatividad* de un sistema lógico [KL01], que determina su consistencia o su expresividad, lo que muestra con claridad su importancia. No obstante, no se considerará aquí esta distinción, ya que actualmente sólo se maneja de manera explícita en una de las teorías constructivas de tipos de Per Martin-Löf y en los estudios sobre el tema de Fairouz Kamareddine.

Sin embargo, aun asumiendo una identificación entre orden y tipo, ambos conceptos se refieren exclusivamente a la noción de *recursión*: controlan el comportamiento de funciones y predicados que hacen referencia a sí mismos. De este modo, un predicado que trata acerca de predicados es de segundo orden; una función recursiva puede ser de orden superior. Sin embargo, esto no debe plantear una confusión con la noción de *reflexión*. En ésta, un predicado puede tratar sobre otro, pero no lo *usa*, sino que solamente lo *menciona*. Típicamente, el predicado objeto ha debido ser transformado previamente en un tipo básico, sea un número, una cadena de caracteres o un nombre; a este proceso se le denomina *reificación*. En tal caso, el predicado sujeto es simplemente de primer orden, ya que manipula al otro como un *dato*, y de dice entonces que es un *meta-predicado*. Puede parecer que el enfoque es más limitado, pero en realidad ocurre exactamente lo contrario. De hecho, la potencia expresiva es mayor, ya que el meta-predicado tiene acceso a todos los detalles internos del dato que manipula. Así pues, *no se puede hacer una identificación* inconsciente entre una meta-entidad y una entidad de segundo orden, si bien es cierto que en la mayoría de los casos la distinción se relaja sin mayores consecuencias.

En realidad, suele plantearse que el prefijo *meta-* tiene una connotación lingüística, y que

implica la presencia de una jerarquía de lenguajes. Esto no es necesariamente así, y se debe principalmente a razones históricas. El prefijo *meta-* fue reintroducido en el contexto científico a partir de las tesis formalistas de David Hilbert, que él mismo agrupó bajo el nombre de *metamatemática*. Cuando otros autores –incluyendo a los propios Kurt Gödel y Alfred Tarski– desarrollaron este campo, lo hicieron siempre bajo la suposición de que el prefijo implicaba una relación de tipo lingüístico. El mayor defensor de esta tesis fue Stephen C. Kleene, que la consagró definitivamente al escribir su clásica compilación sobre el tema [Kle74]. Esta interpretación fue discutida por algunos autores, en particular Haskell B. Curry, que propugnaba un uso más amplio del prefijo, especialmente en lo relativo al término *metateoría*, al que de hecho daba el mismo significado que tiene hoy en día en lógica [Hun81]. No obstante, el propio Curry renunció temporalmente a este uso, admitiendo la popularidad del enfoque de Kleene, e introduciendo en su lugar el término *epiteoría* [CF67], que resulta ser etimológicamente más correcto, aunque finalmente ha desaparecido. Lo cierto es que tanto el uso de Kleene como el de Curry se agrupan hoy bajo el prefijo *meta-*, pero la connotación lingüística se mantiene, con mayor fuerza si cabe; la influencia de los trabajos de Tarski sobre el concepto semántico de verdad y las jerarquías de lenguajes [Tar35, Tar44] ha sido determinante en este sentido.

Finalmente, se ha de señalar brevemente que a menudo se da un *colapso*, consciente o inconsciente, entre algunas de las nociones y relaciones implicadas. Esto significa que, sin tratarse necesariamente del mismo concepto, a menudo se ha identificado a la reflexión con otras relaciones estructurales, con el fin de facilitar su comprensión. Ya se ha mencionado la equivalencia que normalmente se asume entre la reflexión y las entidades de orden superior. Pero ésta no es la única confusión existente. Colin Atkinson [AKHS00] señala que la difusión de los metamodelos provoca a menudo un mal uso de sus posibilidades –por ejemplo, los *estereotipos*– en detrimento de las clásicas relaciones de generalización y subtipado. Sin embargo, él mismo asume una identificación entre la reificación y la instanciación (clasificación), derivada de la noción de *meta-clase*. La propia Pattie Maes [Mae87a] señala que en realidad esta equivalencia es también incorrecta: una meta-clase no ha de ser necesariamente la clase de una clase –una clase de segundo orden–, sino una entidad que contiene información sobre una clase. No obstante, debido probablemente a la influencia de Smalltalk, esta última idea ha perdurado, expresando un colapso muy habitual entre las relaciones de reificación y tipado[3], y dando lugar a la descripción de entidades –metaobjetos, metaclases– con una naturaleza claramente dual [CRC98]. Este colapso tiene también, sin embargo, sus ventajas, y ha conducido directamente al concepto de reflexión estructural, que será descrito en la sección 4.4.1.

4.2.1 Meta-Programación

Teniendo en cuenta todo lo anterior, y entrando ya en un contexto más explícitamente informático, se define *metaprogramación* como "el arte de escribir programas que leen, transforman o escriben otros programas" [Rid99a]. Se trata de una noción que no es autorreferencial, por lo que no se identifica con la reflexión. Sin embargo, la incluye, en el sentido de que la programación reflexiva es una forma de meta-programación que, además, actúa sobre sí misma.

[3] De hecho, se debería considerar también la distinción entre clasificación y tipado [LP91], pero ésta será obviada a lo largo de este capítulo.

En este contexto, se asumirá como normal general que el prefijo *meta-* hace referencia al *nivel de abstracción en el que se definen las reglas que gobiernan el sistema*, asumiendo que éste adopta también la forma de un sistema. De este modo, se define un **meta-sistema** como aquel sistema que actúa sobre otro sistema. Se trata, por tanto, de una definición derivada; un sistema no es meta-sistema por sí mismo, sino *en relación a otro*. No se dice, pues, que A es un meta-sistema, sino que A es un *meta-sistema de* B, es decir, que A controla el comportamiento de B.

En el dominio de la Informática, se puede definir de manera más precisa a un *meta-sistema* como un sistema computacional cuyo dominio [Mae87a] de aplicación es otro sistema computacional, al que se denomina su **sistema base** o **sistema objeto**. Un meta-sistema ha de tener una representación del sistema objeto como parte de sus datos, a la que se denomina *meta-representación* o *metainformación estructural*. Desde el punto de vista del sistema base, el programa del meta-sistema es un *meta-programa*, y realiza una *meta-computación*. A partir de cierto punto, el uso abusivo del prefijo puede llegar a parecer reiterativo, y hasta tedioso; pero una vez que se asume el sentido preciso en el que es empleado, resulta bastante claro.

El paso de este punto a la reflexión resulta casi trivial, aunque sus implicaciones son enormes [Mae87a, Ste94]: a un meta-sistema conectado causalmente (§4.3.1) con un sistema objeto que es parte de sí mismo, se le denomina **sistema reflexivo**, y al acto de razonar y actuar sobre sí mismo se le denomina **computación reflexiva**.

Estos aspectos, a los que ya se ha aludido con anterioridad (§4.1.1), serán considerados con mayor detalle en la sección 4.3.1, donde se proporciona una formulación definitiva del concepto de Reflexión.

$E \rightarrow S$	**Clase**	**Sistemas de Ejemplo**
$0 \rightarrow 0$	No Meta	—
$1 \rightarrow 0$	Intérprete	Intérprete, Inspector, Analizador, ...
$0 \rightarrow 1$	Generador	Precompilador, Generador de Código
$1 \rightarrow 1$	Traductor	Traductor, Compilador, Optimizador, Estilizador, ...
$2 \rightarrow 0$	MetaIntérprete	MetaIntérprete, Analizador Diferencial, Co-Ejecutor, ...
$2 \rightarrow 1$	MetaTraductor	MetaCompilador, Co-Optimizador, Tejedor de Aspectos, ...
$1 \rightarrow 2$	Divisor de Fase	Divisor de Fase

Tabla 4.1: Metaprogramación: Clasificación de Rideau (adaptada)

No ha de pensarse, sin embargo, que la metaprogramación es un concepto extraño o inusual. En realidad, sigue un razonamiento muy simple: si la programación tiene como objetivo facilitar el trabajo de los hombres, relevándolos de ciertas tareas rutinarias, del mismo modo las tareas más rutinarias de la programación pueden ser realizadas por la máquina, facilitando el trabajo del programador. Esta idea se ha aplicado, conscientemente o no, desde hace mucho tiempo, y se encuentra en la misma raíz de conceptos como el de *compilador* o *programación visual*.

Rideau [Rid99a] proporciona una clasificación de todos los tipos de sistemas posibles, con particular atención a los que manifiestan *algún tipo* de meta-programación. Esto permite comprobar que ésta se encuentra en el mismo rango que los sistemas habituales, y que no resulta

en absoluto una idea exótica. La Figura 4.1 muestra esta clasificación. Es particularmente interesante la columna marcada como $E \rightarrow S$, esto es, *entradas* $\rightarrow$ *salidas*. En ella se indica el número de programas que se toman como entradas al sistema (E), y el número de programas que se obtienen como resultado (S). Allí donde $S \geq 1$, el sistema realiza algún tipo de generación de código; cuando $E = 2$, se tiene auténtica metaprogramación. La reflexión se encontrará, entonces, en este último contexto.

4.3 Conceptos Fundamentales

A lo largo de esta sección se revisarán algunos de los conceptos fundamentales del campo, comenzando por una redefinición más detallada de la noción de reflexión, y prestando una especial atención a los aspectos estructurales.

4.3.1 Hipótesis de Reflexión

La primera definición de reflexión, históricamente hablando, es la del trabajo original de Brian Cantwell Smith. Casi veinte años después, sigue siendo una de las más claras, y permite delimitar el verdadero alcance del concepto, evitando las complejidades de otras más modernas.

> ***Hipótesis de la Reflexión****: Del mismo modo que un proceso computacional puede ser construido para razonar sobre un mundo externo en virtud de la inclusión de un proceso ingrediente (intérprete) que manipule formalmente representaciones de ese mundo, también así puede hacerse a un proceso computacional razonar sobre sí mismo en virtud de la inclusión de un proceso ingrediente (intérprete) que manipule formalmente representaciones de sus propias operaciones y estructuras.*
>
> *Brian C. Smith [Smi82]*

Esta definición implica poco más que la existencia de múltiples niveles de abstracción (meta-intérpretes) y la conexión causal; lógicamente, ignora toda la serie de conceptos relacionados, como el de *reificación*, que han ido surgiendo a lo largo del tiempo. Tal vez por eso mismo, resulta ser no sólo la más "pura", sino incluso una de las más potentes y flexibles. Algunos autores, como Rémi Douence [DS99] han llegado a afirmar que puede resultar más beneficioso volver a examinar este enfoque –el original– a la luz de los actuales conocimientos sobre lenguajes de programación, tecnología de compilación y modelos reflexivos, que continuar reelaborando los complejos sistemas ya existentes.

No obstante, esta definición resulta aún demasiado general; para comprender por completo cuáles son sus implicaciones, es necesario proporcionar una definición más técnica. Sin duda, la mejor definición, aunque también la más restrictiva, es la elaborada por Jacques Malenfant:

> ***Reflexión*** *es la habilidad integral de un programa para observar o cambiar su propio código, así como aspectos de su lenguaje de programación (sintaxis, semántica, o implementación), incluso en tiempo de ejecución. Se dice que un lenguaje de programación es* ***reflexivo*** *cuando proporciona a sus programas la capacidad para realizar reflexión (completa).*
>
> *Malenfant, Jacques & Demers [MJD96]*

En esta definición se señala un detalle particularmente importante, y es el hecho de que la capacidad reflexiva involucra al propio lenguaje que permite realizarla; como se verá más adelante, a veces hasta el punto de poder alterar dinámicamente su propio significado.

La característica esencial que define a un sistema reflexivo y lo distingue de cualquier otro meta-sistema es el concepto, ya mencionado en varias ocasiones, de **conexión causal**. Se designa con este nombre a la conexión existente entre el sistema base y el meta-sistema. Ha de ser tal que todo cambio en el sistema base ha de inducir el correspondiente cambio en su propia representación (meta-representación) ante el meta-sistema; y, a su vez, todo cambio realizado por el meta-sistema sobre la meta-representación ha de tener un efecto instantáneo sobre el sistema base. Esto quedará aún más claro cuando se hable de un único sistema, y se hable en términos de *nivel base* y *nivel meta*.

4.3.2 Reificación

La definición original de Smith postulaba un intérprete capaz de manipular las operaciones del intérprete, es decir, un meta-intérprete. Éste es a su vez un intérprete, luego puede disponer de su propio meta-interprete; es decir, un meta-meta-intérprete. Este proceso puede continuar indefinidamente, en una sucesión o torre de intérpretes que ha dado en ser conocida como la *Torre Reflexiva Infinita*. Por supuesto, la torre no es realmente infinita: en algún momento, el meta-intérprete correspondiente es sustituido por un *intérprete por defecto*, proporcionado por el sistema, que no es reflexivo. De este modo se detiene la auto-referencia, y la torre queda limitada a un número concreto de intérpretes, conocido como *grado de introspección*. Cada uno de los intérpretes define un subsistema con sus propias normas, que recibe el nombre de *nivel*.

El sistema y el meta-sistema, tal como fueron definidos en la sección 4.2.1, pueden por tanto ser considerados como subsistemas de un único sistema compuesto, en el que cada uno de ellos es un nivel de este tipo. Se habla, pues, de *nivel base* o *nivel objeto*, para hacer referencia al sistema, y de *nivel meta*, para hacer referencia al meta-sistema.

Sin embargo, el mismo concepto de la Torre Reflexiva es incómodo, y la forma de eliminarlo es poco elegante. Era necesario evitar su existencia de algún modo, y para ello Mitchell Wand y Daniel Friedman [WF88, FW84] introdujeron el concepto de **reificación** o **materialización**.

El sentido del concepto queda claro en la siguiente definición:

> *La* **Reificación** *es el proceso por el cual un programa de usuario $\mathcal{P}$, o cualquier aspecto de un lenguaje de programación $\mathcal{L}$, que estaban implícitos en el programa traducido y en el sistema de ejecución, se hacen explícitos, utilizando una representación (estructuras de datos, procedimientos, etc.) que se expresa en el propio lenguaje $\mathcal{L}$ y es puesta a disposición del programa $\mathcal{P}$, que podrá inspeccionarla como si se tratase de datos ordinarios. En los lenguajes reflexivos, los datos obtenidos mediante la reificación están* ***conectados causalmente*** *con la información reificada correspondiente, de modo que una modificación en uno de ellos afecta al otro. De este modo, los datos obtenidos por la reificación son siempre una representación fidedigna del correspondiente aspecto reificado.*
>
> *Malenfant, Jacques & Demers [MJD96]*

Existe cierta confusión en torno a la reificación. Por una parte, se llama igual al proceso, la relación entre los dos bloques de datos, y ocasionalmente al propio bloque de datos obtenido mediante este proceso. Por otra parte, no se debe confundir la reificación con la posibilidad de disponer de una *entidad de primera clase*, algo que es factible en un buen número de lenguajes. La entidad de primera clase –por ejemplo, una función– puede ser manejada por el lenguaje como si se tratase de un dato, pero una característica clave de la reificación es el hecho de que los datos obtenidos han de ser tales que puedan ser *inspeccionados* y modificados por el programa, algo que no está implícito en el concepto de entidad de primera clase. Se trata, en realidad, de una distinción similar, pero no idéntica, a la que se hacía en la sección 4.2 entre las nociones de meta-entidad y entidad de orden superior.

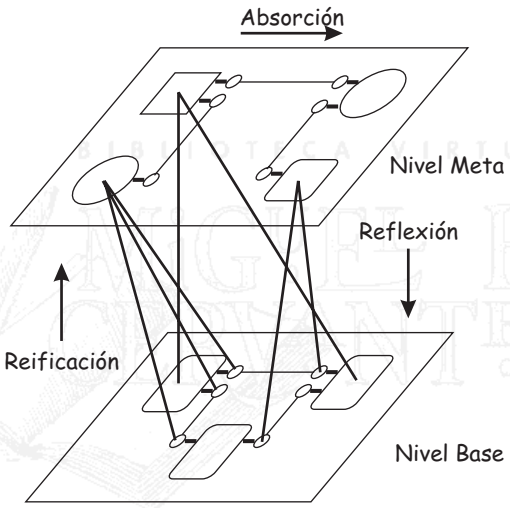

Figura 4.1: Modelo de Reflexión en Dos Niveles

Con este proceso se elimina la Torre Reflexiva, pues ahora ésta sólo se va desarrollando a medida que se ordena explícitamente la reificación de una entidad. La reificación es pues el proceso que permite a una cierta entidad "subir" (*shift-up*) del nivel base al nivel meta. Dado que todo cambio en el nivel meta ha de reflejarse en el nivel base, se dispone del proceso inverso que permite "bajar" (*shift-down*) del uno al otro. La conexión causal es realizada por el conjunto de ambas operaciones, tal como puede verse en la Figura 4.1.

Ocasionalmente, se denomina *absorción* al conjunto de operaciones realizadas en el nivel meta; con esto se quiere expresar que una parte de la computación ha sido "absorbida" por la meta-computación. Teniendo en cuenta esto, cada operación reflexiva puede verse como un proceso en tres etapas: la reificación de la entidad, la absorción de una tarea dada, y la reflexión final que permite *reflejar* los resultados en el sistema base[4]. De hecho, a menudo el nivel meta *no existe* como una estructura persistente, definida explícitamente como un fragmento del sistema,

[4]Algunos autores prefieren no mencionar a la operación de *reflexión*, con el fin de evitar confusiones con el término global de *Reflexión*. Es cierto que, dada la noción de *absorción*, esto resulta posible; pero es también mucho menos claro, ya que se pierde la simetría. En ocasiones, asimismo, se han utilizado denominaciones alternativas –aunque inadecuadas–, como el propio nombre de *absorción* para la operación de descenso, o el uso

sino que sólo se produce como un esquema o proceso de reificación, absorción y reflexión que realiza, en definitiva, una meta-computación intercalada en la ejecución del sistema.

No resulta ocioso señalar que resulta completamente *imposible* disponer de un procedimiento de *reificación universal*. Un procedimiento universal de este tipo permitiría obtener una estructura de datos concreta (d) a partir de cualquier programa abstracto (p). No obstante, toda estructura de datos puede verse como un conjunto de instrucciones en cierto lenguaje, esto es, como un fragmento de código fuente. Esto significa que disponer del procedimiento equivaldría a tener un método universal capaz de generar el código fuente correspondiente a cualquier función computable abstracta. O en términos llanos, que bastaría con poder plantear formalmente un problema dado, para generar automáticamente un algoritmo que lo solucionase.

Sin embargo, como se ha dicho, esto es totalmente imposible. Rideau [Rid99b] lo plantea formalmente del siguiente modo:

Teorema 4.1 (Teorema de Limitación) *Dada una estructura de datos $\mathbb{D}$ (es decir, un conjunto recursivamente enumerable con un test de igualdad computable), y un dominio computacional $\mathbb{C}$ equivalente-Turing (es decir, el conjunto dado por la semántica definida sobre una cierta estructura de datos o lenguaje fuente, capaz de expresar todas las funciones recursivas parciales),* ***no existe*** *una correspondencia inyectiva total computable $\mathcal{R}$ de $\mathbb{C}$ a $\mathbb{D}$.*

La demostración intuitiva del Teorema es muy sencilla. Si existiese la correspondencia $\mathcal{R}$, se podrían obtener las estructuras de datos correspondientes a dos funciones recursivas parciales cualesquiera. Se puede determinar si las dos estructuras así obtenidas son equivalentes, ya que el test de igualdad de $\mathbb{D}$ es computable. Por tanto, se dispondría de un procedimiento finito para detectar la equivalencia de dos funciones recursivas parciales, lo que solucionaría el Problema de la Detención, y por tanto violaría el Teorema Fundamental de Turing.

No obstante, la reificación es posible cuando no se da alguna de las hipótesis del Teorema, o no se considera la correspondencia entre un programa concreto y la función *abstracta* que éste representa. En resumen, la reificación puede realizarse sin problemas en la mayor parte de los casos normales, y de hecho con una amplitud considerable.

4.3.3 Arquitectura de Meta-Nivel

Existe cierta confusión, en la literatura existente, entre varias descripciones asociadas con la infraestructura de un sistema reflexivo, y especialmente entre los términos *arquitectura de meta-nivel*, *meta-arquitectura* y *arquitectura reflexiva*. En este apartado se expone brevemente el distinto enfoque de cada uno de ellos, y se clarifican sus diferencias.

En primer lugar, se denomina **meta-arquitectura** a la arquitectura –estructura– utilizada para dar soporte a la definición y construcción de otras arquitecturas. Coincide, por tanto, con su significado etimológico, y se puede considerar sinónima del término *infraestructura*. Puede hacer referencia a aspectos físicos, organizativos o conceptuales, y no tiene por qué ir asociada a un sistema reflexivo; de hecho, no es necesario hacerla explícita. En el esquema reflexivo descrito, se trata simplemente del *contenido* del nivel meta –o de todos los niveles meta–.

de los términos de *introspección* e *intercesión* para referirse a las operaciones [KFRGC98a, Caz00], cuando en realidad se trata de algo completamente distinto (§4.4.1). En esta tesis se emplea la convención de denotar al campo con mayúscula, y a la operación con minúscula, al menos cuando exista potencial para la confusión.

En segundo lugar, se considera la expresión *arquitectura de meta-nivel*, que a pesar de ser muy habitual, tiene sin embargo una forma contraintuitiva. Una interpretación literal haría pensar que se refiere a la estructura –arquitectura– constituida por los elementos ubicados en el nivel meta; es decir, la meta-arquitectura que acaba de describirse. Sin embargo, no es así.

En realidad, el término **arquitectura de meta-nivel** hace referencia a la estructura mediante la que el nivel base accede al nivel meta: es decir, la arquitectura que describe la *relación* entre el nivel base y el meta, o los mecanismos que utiliza el sistema reflexivo para proporcionar la reflexión. En un sistema orientado al objeto, donde la arquitectura de meta-nivel se presenta al programador como un conjunto de objetos que definen una interfaz compleja, ésta recibe el nombre de *Protocolo de Meta-Objeto*. El subapartado siguiente describe este último término de forma más detallada, ya que constituye, sin duda, la forma más habitual.

Finalmente, se denomina **arquitectura reflexiva** a la estructura de un sistema reflexivo, esto es, al conjunto formado por el nivel base, el nivel meta, y la relación entre ellos [BL97b]. Este término se confunde a menudo con el anterior, ya que ambos describen prácticamente el mismo esquema, esto es, la colección de elementos que proporcionan reflexión.

No obstante, estrictamente hablando existe una diferencia entre las dos expresiones. En general, la primera es aplicable a todo meta-sistema, mientras que la segunda sólo tiene sentido cuando se refiere a un sistema reflexivo:

> *La diferencia principal entre una* arquitectura de meta-nivel *y una* arquitectura reflexiva *es que la arquitectura de meta-nivel sólo proporciona* acceso estático *a la representación del sistema computacional, mientras que la arquitectura reflexiva también proporciona un* acceso dinámico [*a la misma*].
>
> *Pattie Maes [Mae87a]*

Es decir, una arquitectura de meta-nivel, estrictamente, sólo proporciona introspección (§4.4.1), mientras que una arquitectura reflexiva da soporte a cualquier tipo de reflexión.

Lo cierto es que, normalmente, ambas expresiones –y en especial la primera– se utilizan en los dos sentidos, y la distinción teórica se tiene pocas veces en cuenta. Es incluso más frecuente una cierta connotación, según la cual la *arquitectura de meta-nivel* se refiere al soporte genérico de reflexión en un sistema dado, mientras que una *arquitectura reflexiva* es, más bien, la estructura de una instancia de ese sistema, y el uso concreto que hace de dicho soporte genérico.

Protocolo de Meta-Objeto

Se define un **protocolo de meta-objeto** (Mop), en sentido estricto, como la especificación de la implementación de un modelo de objetos reflexivo [GC96]. Asumiendo que el nivel meta describe la arquitectura del modelo de objetos, lo que describe el Mop es *qué* meta-objetos son necesarios para implementar dicho modelo, de modo que se proporciona soporte para la reflexión. Por tanto, el protocolo describe más bien una interfaz, que viene dada por la combinación de las interfaces de todos los meta-objetos instanciados.

En resumen, se trata del conjunto concreto de meta-objetos disponibles en un determinado sistema reflexivo, pero no considerados como la meta-arquitectura del mismo, sino como su arquitectura de meta-nivel. Desde el Mop de Clos, que fue el primero y es probablemente el

más complejo [MJD96], hasta el modelo multi-MOP de Iguana [GC96], se han realizado multitud de implementaciones e interpretaciones. En sentido informal, suele utilizarse el término para referirse, de manera genérica, al modo en que se proporciona reflexión en un sistema orientado al objeto, así como al *modo en que se invocan* los meta-objetos.

Desde el punto de vista del programador, un meta-objeto puede ser invocado de cuatro maneras diferentes. No todas ellas constituyen, en sentido estricto, un protocolo de meta-objeto; pero es cierto que se han definido MOPs utilizando cada uno de estos enfoques:

- Dada una asociación entre un objeto base y una meta-entidad –sea un metaobjeto o una metaclase–, se produce una invocación en esta última por cada acceso al objeto reificado. Es el enfoque más habitual, y tal vez el más coherente con la teoría original [DS99].

- El nivel base invoca explícitamente a una entidad del nivel meta, a la que se refiere por su nombre. Este tipo especial de invocación puede hacerse mediante el uso de comandos específicos [GC96, RC00], o a través de la definición de abstracciones especiales, análogas a los *procedimientos reflexivos* [dRS84] o los *predicados reflexivos* [DM95].

- Como respuesta a la invocación de un evento (*meta-evento*) específico, que puede ser provocada por un objeto-base, otro meta-objeto, o el propio sistema [OIT93, ASA01].

- Mediante la definición de estructuras con puntos de extensión (*hooks*), a través de los cuales se puede introducir nueva meta-funcionalidad en el sistema; éste es el caso del *Python Object Protocol* [vR01] o de los *fallbacks* de Lua [IdFF96].

Esto se relaciona directamente con otro de los aspectos tratados en el célebre artículo original de Pattie Maes [Mae87a]. En él se describe una distinción entre dos tipos de reificación que, en realidad, se refiere más bien a la política de invocación de los meta-objetos. Precisamente por ese motivo, no fue comentado en la sección 4.3.2, ya que en realidad no describe un aspecto conceptual de la reificación. Maes distingue, entonces, entre *reificación implícita*, cuando la invocación de un meta-objeto es producida por el propio sistema reflexivo, y *reificación explícita*, cuando la invocación es realizada de manera consciente por un objeto base. Su correspondencia con las cuatro aproximaciones descritas más arriba resulta obvia en la mayoría de los casos, aunque con cierta frecuencia se puede dar una mezcla de ambas.

En definitiva, existe una gran diversidad en lo relativo a estos aspectos, ya que suelen depender, no sólo de cada lenguaje, sino de cada posible perspectiva del sistema reflexivo.

4.3.4 Meta-Espacio

La división de un sistema reflexivo en *meta-niveles* surge como una necesidad puramente conceptual: cada nivel describe, en realidad, un grado de abstracción mayor, que se incrementa sobre el del nivel anterior. No se trata, por tanto, de una modularización consciente de la meta-computación, realizada por causas prácticas.

No obstante, cuando cada uno de los meta-niveles crece en complejidad, algún tipo de modularización se hace necesaria, y no solamente por motivos prácticos. En este sentido ha de enmarcarse la propuesta original de Maes [Mae87a], que no sólo hizo posible la inclusión de

reflexión en la orientación al objeto, sino que, también, propició la estructuración del nivel meta como un conjunto de meta-objetos interrelacionados.

Sin embargo, la división en meta-objetos puede resultar, en ocasiones, demasiado elemental. En un sistema reflexivo convencional, su número suele ser muy grande, y su interacción múltiple; la estructura que se define, en definitiva, resulta aún demasiado compleja. Por eso se hace deseable algún tipo de abstracción intermedia, ni tan gruesa como el nivel completo, ni tan fina como un meta-objeto. En este contexto, se han propuesto dos enfoques diferentes, ambos vinculados indirectamente a los trabajos de Mario Tokoro.

El primero de ellos es el concepto de *meta-espacio*, debido a Yasuhiko Yokote y el propio Tokoro, como parte de su trabajo en el sistema Apertos, un sistema operativo reflexivo [Yok92]. Su estructura conceptual, basada en el *modelo de campo computacional* [Tok90] define el meta-espacio como una agrupación de todos los meta-objetos que reifican *algún aspecto* de un objeto (base) concreto. De este modo, el nivel meta se puede considerar dividido en un conjunto de metaespacios, que no tienen por qué ser disjuntos; los meta-objetos comunes, de hecho, son los que permiten su integración. Esta misma idea será utilizada a lo largo de esta tesis, concretamente en el Capítulo 5, durante el desarrollo del modelo Marmol.

Asociado a éste se encuentra el concepto de *reflector*, descrito como un único meta-objeto que reune toda la influencia de un meta-espacio concreto, y se encarga de mantener la comunicación con el respectivo objeto base, haciendo por tanto funciones de mediador.

El segundo enfoque se basa en el concepto de *modelos* [OIT93] o *planos* [Mar01], planteado exactamente desde la perspectiva contraria. En lugar de agrupar los meta-objetos, se plantea una *descomposición* explícita, en base a aspectos concretos, del nivel meta. En esta aproximación, todo meta-nivel se considera implícitamente dividido en una serie de planos, que a su vez se estructuran como una red de meta-objetos. Un objeto base podrá ser reificado simultáneamente por distintos planos, y un plano podrá reflejarse en múltiples objetos. El enfoque tiene obvias similitudes con la programación orientada al aspecto (§4.6), y ha conocido cierta difusión, especialmente en el contexto del *middleware* reflexivo [BBI+00, BCA+01].

También podrían enmarcarse aquí las *categorías de reificación* de Iguana [GC96]; sin embargo, éstas plantean una división puramente semántica, en el sentido de que sólo agrupan conceptualmente las abstracciones de las que dispone el programador. Pero una vez que éste las utiliza, la invocación de los meta-objetos se produce de manera totalmente independiente; por tanto, no suponen en realidad una auténtica división del nivel meta.

4.4 Clasificaciones de la Reflexión

Existen toda una serie de subdivisiones de la Reflexión, atendiendo a las distintas posibilidades del campo; aquí se hará una breve reseña de las más importantes, con el fin de proporcionar una visión más general de las distintas alternativas posibles.

Existen otras clasificaciones, aparte de las que vemos aquí; las más importantes se relacionan con el paradigma orientado a objetos, y hacen referencia al método de intercepción y al modelo general definido por su meta-arquitectura. Una clasificación de este último tipo se revisará en la sección 4.5.1. Otras, como la distinción entre reflexión explícita e implícita, no tienen aplicación posible en el tema que nos ocupa, por lo que no serán consideradas.

4.4.1 Por Efecto

Atendiendo al efecto que tiene el meta-programa sobre el programa, se pueden distinguir dos grandes grupos, el segundo de los cuales se subdivide a su vez. La Figura 4.2 muestra esta clasificación, justamente considerada como la más importante dentro del campo.

$$\text{Reflexión} \begin{cases} \text{Introspección} \\ \text{Intercesión} \begin{cases} \text{Reflexión Estructural (Lingüística)} \\ \text{Reflexión de Comportamiento} \end{cases} \end{cases}$$

Figura 4.2: Tipos de Reflexión según el Efecto

Introspección. Se designa con este nombre a la capacidad que tiene un programa para *observar*, y por tanto razonar, sobre su propio estado.

Intercesión. Por contraposición al anterior, se define como la capacidad que tiene un programa para *modificar* su propio estado de ejecución, o *alterar su propia interpretación* o significado.

La *introspección* consiste solamente en *observar*. No puede alterar ningún aspecto de su propio código, sino solamente examinar su propia estructura. Sorprendentemente, algo tan simple manifiesta una potencia enorme, ya que se permite tomar decisiones en función de esta información (meta-información). Dado que se trata de información que normalmente no está accesible más que al propio compilador, la cantidad de manipulaciones inusuales que se pueden hacer es numerosa. Debe tenerse en cuenta que el célebre Api de Reflexión de Java sólo proporciona este tipo de acceso, y aun así ha conocido una difusión y aplicabilidad considerables.

Dentro de la *intercesión* se distinguen dos tipos de reflexión, denominados reflexión estructural o lingüística, y reflexión de comportamiento. Brevemente, se denomina *reflexión estructural* a aquella que reifica los aspectos estructurales de un programa, tales como la jerarquía de herencia, la estructura de las clases, o los tipos de datos. A su vez, la *reflexión de comportamiento* se ocupa de la reificación de las propia computación y su comportamiento; por ejemplo, del mecanismo de invocación de un método [Gol97]. Sin embargo, esta idea intuitiva es sólo aproximada, y resulta interesante dar aquí las definiciones más precisas, y comúnmente aceptadas, proporcionadas por Demers y Malenfant [DM95, MJD96].

- **Reflexión Estructural.** Se denomina con este nombre a la habilidad de una lenguaje para proporcionar una reificación completa del programa que se está ejecutando, así como una reificación completa de sus tipos abstractos de datos. También se le denomina **Reflexión Lingüística**, debido al hecho de que es capaz de generar nuevos fragmentos del programa en funcionamiento, e integrarlos en su ejecución [KMS98].

- **Reflexión de Comportamiento.** Se refiere a la habilidad del lenguaje de proporcionar una reificación completa de su propia semántica e implementación, así como una reificación completa de los datos y la implementación del propio *sistema de ejecución* dentro del cual él mismo se ejecuta.

Ha de tenerse en cuenta que ambos tipos de reflexión *no son* mutuamente excluyentes; ni la segunda, aparentemente más potente, implica necesariamente a la primera.

4.4.2 Por Tiempo

Según el momento en que se aplique la Reflexión, se puede distinguir entre Reflexión en tiempo de compilación, de carga, de enlace o de ejecución. Normalmente, se atiende sólo a la distinción entre los dos siguientes:

Reflexión en Tiempo de Compilación. En esta opción, un meta-programa M altera el modo en que se compila el programa P correspondiente, de modo que el código ejecutable que se genera es distinto del que inicialmente se habría creado. Con distintos meta-programas, se puede lograr que el mismo programa P tenga comportamientos diferentes, sin que se haya introducido cambio alguno en su código fuente.

Reflexión en Tiempo de Ejecución. Esta es, en realidad, la auténtica Reflexión. Un programa ejecutable lleva a cabo su ejecución, y durante la misma sus operaciones son *interceptadas* por un meta-programa, que puede ser independiente, o bien una parte de su propio código[5]. Habitualmente se ha supuesto que realizar reflexión de este tipo requería que ambos programas trabajasen sobre una cierta *máquina virtual* [Gol97] o directamente sobre un intérprete, y de ahí el considerable éxito de las versiones Java; sin embargo, Malenfant [MJD96] ha demostrado que eso no es estrictamente necesario, dados los avances actuales en compilación dinámica.

En realidad, el primer caso tiene más de meta-programación que de reflexión; sin embargo, existe una diferencia: algunos sistemas admiten también que al programa P –que será modificado– se le proporcionen primitivas que le permitan invocar de algún modo al meta-programa M –que lo modificará–, alterando el modo en que éste le afecta. Con esto, el flujo ya no es totalmente unidireccional, puesto que no sólo M modifica el resultado de P, sino que el propio P pueden influir en el modo en que él mismo es modificado.

El ejemplo más significativo de reflexión en tiempo de compilación es el del sistema Open C++ [Chi95] y sus derivados. Por su parte, los ejemplos más populares de reflexión en tiempo de ejecución se han implementado en Java, aprovechando las características de su máquina virtual: es el caso de MetaXa [Gol97], Dalang, Meta/J [DS99] o Guaraná [OCGB98].

4.5 Reflexión y Orientación al Objeto

Como ya se ha dicho, la Reflexión nació con Smith en el ámbito de los lenguajes de programación funcional, concretamente en el seno de Lisp. Esto la confinó durante años al ámbito de los lenguajes de programación que habitualmente se relacionan con la Inteligencia Artificial, tales

[5]En un sistema secuencial, el meta-programa se inserta como parte del programa, y altera su ejecución del modo conveniente, capturando sus interacciones. En un sistema concurrente, los dos programas pueden ser independientes, pero uno de ellos supervisa el comportamiento del otro y le obliga a seguir una evolución determinada. Esto coincide con el concepto de *superposición* [Kat93], definido en teoría de la concurrencia, y que no partía de consideraciones reflexivas. No obstante, se plantea una interesante correspondencia entre ambos.

como Lisp o Prolog –y sus múltiples derivados–, así como a multitud de sistemas basados en reglas, entre los que cabe destacar el caso de Teiresias [Sie92].

Todo esto cambió gracias al trabajo de Pattie Maes [Mae87a, Mae87b], que decidió introducir estos conceptos en el contexto de la orientación al objeto. Para ello, diseñó el lenguaje 3-Krs sobre un sistema basado en conocimiento, y propuso toda una serie de mecanismos que en su mayoría siguen vigentes hoy en día. El sistema de Maes fue rápidamente adoptado, y se difundió con especial rapidez al campo de los Sistemas Concurrentes Orientados al Objeto, donde el desarrollo de la variante reflexiva de Abcl, llamada Abcl/R [WY88], se ha convertido en una referencia tan importante como la primera.

Los conceptos vistos hasta ahora se mantienen con gran propiedad; sin embargo, en una organización orientada a objetos, cada uno de los niveles reflexivos estará estructurado como una red de objetos que se comunican; de ahí surge el concepto de *meta-objeto*, entendido como un objeto situado en el nivel meta. Este objeto tendrá su propia clase, que se situará también en el nivel meta, y recibe el nombre de *metaclase* –véase, no obstante, la sección 4.2–. Existe una gran variedad de posibles relaciones entre objetos, clases, metaobjetos y metaclases; algunas se esbozan muy rápidamente en la sección 4.5.1.

4.5.1 Modelos Reflexivos de Objetos

La Reflexión puede tomar formas muy diferentes en la Orientación al Objeto; éstas definen estructuras diferentes, por lo que son particularmente interesantes desde la perspectiva de la Arquitectura de Software. En líneas generales, se pueden distinguir dos grandes familias, las cuales a su vez se subdividen, dando lugar a cuatro modelos. La Figura 4.3 muestra esta taxonomía, debida a Walter Cazzola [Caz98].

- Reflexión
  - Reificación de Objeto
    - Modelo de Meta-Clase (Mmc)
    - Modelo de Meta-Objeto (Mmo)
  - Reificación de Comunicación
    - Modelo de Reificación de Mensaje (Mrm)
    - Modelo de Reificación de Canal (Mrc)

Figura 4.3: Modelos de Reflexión Orientada al Objeto [Caz98]

La familia compuesta por los dos primeros utiliza un enfoque basado en la reificación del tipo del objeto, mientras que en la segunda se reifica la propia comunicación entre los objetos [Fer89]. A continuación, se detallará brevemente cada una de ellas.

Mmc**: Modelo de Meta-Clase.** En el modelo de Meta-Clase, la torre reflexiva es realizada por el enlace de instanciación entre un objeto y su clase. Normalmente, en un lenguaje orientado a objetos, la clase es el patrón que se copia en la creación de cada objeto, y sólo se define como una entidad real cuando se dispone de atributos y métodos de clase. En el este modelo, la clase del objeto es *un meta-objeto*, y existe realmente. Cada vez que un objeto haga referencia a las características de su clase, el meta-objeto se interpone, y realiza las manipulaciones deseadas.

Al ser el meta-objeto también un objeto, tendrá su propia clase, que será a su vez otro meta-objeto. Por tanto, ésta será una *meta-clase*; se pueden desarrollar niveles de reflexión, sucesivamente: sólo se detienen cuando una clase se utiliza como patrón de sí misma. El concepto de *metaclase* fue introducido en el contexto de Smalltalk, incluso con anterioridad a la definición de Reflexión, y ha influido poderosamente en el desarrollo del campo, así como en la concepción de Reflexión Estructural.

MMO**: Modelo de Meta-Objeto.** Ahora el meta-objeto tiene su propia clase –sea, por ejemplo, *MetaObject*–, y su existencia es independiente de la del objeto. Se establece una relación de *clientela* entre este meta-objeto y un objeto concreto, al que se denomina su *referente*. Los mensajes que se dirijan a éste son interceptados por el meta-objeto, del modo que ya se ha detallado.

Cada meta-objeto puede hacer referencia a múltiples objetos, y cada objeto puede estar conectado (causalmente) a lo largo del tiempo a distintos metaobjetos. Éste es el modelo habitual en los sistemas de Capas de Cebolla (§4.6).

Algunos autores [Mar01] distinguen de manera explícita un esquema de Reflexión de Método, e incluso uno de Reflexión de Método-Objeto, que asigna un meta-objeto distinto a cada uno de los *métodos* de un objeto. Sin embargo, ha de considerarse que si bien el mecanismo de selección es distinto, éste no es más que un caso particular de este Modelo de Meta-Objeto, ya que finalmente ambos definen la misma estructura.

MRM**: Modelo de Reificación de Mensaje.** En este modelo, las meta-entidades son objetos especiales, denominados *mensajes*, que reifican a las acciones que deberían ser realizadas por los métodos de la entidad-base. La selección se hace en base al *tipo del mensaje* enviado; es decir, los objetos involucrados no tienen nada que ver, y no juegan ningún papel.

Por tanto, lo que ocurre es que los mensajes son entidades de primera clase, a las que se les ha asociado un meta-comportamiento. Este enfoque ha sido incluso generalizado a actores [Tan91], por lo que no depende en ningún sentido de la orientación al objeto.

MRC**: Modelo de Reificación de Canal.** Es una extensión del anterior, muy poco difundida. En lugar de reificar los mensajes indiscriminadamente, según su tipo, se tiene en cuenta a los objetos involucrados. Se define un *canal* como el enlace que existe (virtualmente) entre dos objetos cuando éstos se comunican; por tanto, viene definido por su origen, su destino, y el tipo del mensaje enviado.

Su objetivo es, básicamente, no perder la información relacionada con los extremos de la comunicación, que se perdía en el Modelo de Reificación de Mensaje. Tanto su definición original [ACDG95] como su máximo desarrollo [Caz00] se deben fundamentalmente a Walter Cazzola. Obviamente, existe cierta relación conceptual entre este modelo y la expresión reflexiva de un conector arquitectónico (§6.11), aunque este extremo no ha sido explorado con anterioridad, ni siquiera por el propio Cazzola.

Aunque fueron concebidos para el caso concreto de los sistemas orientados al objeto, estos cuatro modelos son lo bastante generales como para ser utilizados en otros entornos. De he-

cho, serán utilizados con libertad en el contexto de la Arquitectura de Software a lo largo del resto de esta tesis, explotando implícitamente la analogía entre componente y objeto. Se puede considerar, en resumen, que se han definido también los modelos análogos, esto es:

MMC $\mapsto$ Modelo de Meta-Arquetipo (Meta-Tipo)
MMO $\mapsto$ Modelo de Meta-Componente (Meta-Instancia)
MRM $\mapsto$ Modelo de Reificación de Mensaje
MRC $\mapsto$ Modelo de Reificación de Conexión (Conector)

Como puede comprobarse, se han mantenido por claridad las mismas siglas que en las versiones orientadas al objeto. Las diferencias son triviales: el modelo MRM, en concreto, no se modifica en absoluto. A su vez, el esquema reflexivo definido en MARMOL, como se verá en el Capítulo 5, deriva desde una mezcla de MMO y MRC, que constituye el modelo $\aleph_{\mathrm{M}}$, hasta una combinación de MMC y MRC, que define el modelo $\aleph_{\Pi}$ definitivo. Este último planteamiento, que no obstante es compatible con el anterior, es el que será empleado también en $\mathcal{P}i\mathcal{L}ar$.

4.6 Reflexión Aplicada: Envolventes y Aspectos

Como es bien sabido, la reutilización de componentes no siempre es inmediata; cuando se desea insertar un componente en un sistema dado, no siempre encaja a la perfección, y ha de ser *adaptado*. Sin embargo, el encapsulamiento es la principal característica de un componente, por lo que normalmente no se tiene acceso a los detalles de implementación. En tal caso, la adaptación ha de hacerse de manera externa, mediante lo que se conoce como un *envolvente* o *wrapper* [Höl93]. Éste se define como una entidad software que recibe los mensajes originalmente destinados al componente, así como los que éste envíe, y los manipula de la manera adecuada, para hacer posible la correcta comunicación.

Un envolvente no sólo se crea por esta causa: también se usa para añadir características –por ejemplo, seguridad–, o para intentar modificar el comportamiento del componente en respuesta a nuevos requisitos. Schneider [Sch99] identifica al menos cuatro tipos diferentes de envolvente, también reconocidos como patrones de diseño [GHJ+95b]. DeLine [DeL99] va aún más allá, llegando a establecer *ocho* esquemas diferentes.

Normalmente, un envolvente se inserta en el sistema sustituyendo en el código del cliente del componente la llamada a éste por una llamada al envolvente. Sin embargo, esto no es siempre posible. Por otra parte, la tarea del envolvente –la intercepción y manipulación de mensajes– guarda un estrecho parecido con los métodos de interposición habitualmente utilizados para realizar reflexión de comportamiento. Por tanto, resulta natural la idea de utilizar una meta-entidad –típicamente, un meta-actor o un meta-objeto– para realizar las funciones de un envolvente, con la ventaja de que esto habitualmente podrá hacerse en cualquier momento, sin necesidad de tener acceso al código de los componentes.

La gran mayoría de los usos "prácticos" de la reflexión se basan en esta idea, que surgió por vez primera en el contexto del modelo de actores, donde las funciones de envolvente eran realizadas por los *meta-actores* de Callsen [Cal94]. También fue él quien hizo por primera vez una metáfora con las capas de cebolla: un actor es envuelto por un meta-actor, que a su vez es un actor y puede ser envuelto por un meta-meta-actor, y así sucesivamente. Esto no es arbitrario, sino muy

interesante, ya que permite superponer múltiples adaptaciones sobre un mismo componente. Se puede representar el conglomerado resultante como un objeto "rodeado" de múltiples capas, de modo que cuando se "quita" una de ellas, lo que se encuentra es otra, de manera similar a cuando se está pelando una cebolla. Desde entonces, el nombre se ha consagrado, y se conoce a este sistema como el *Modelo de Capas de Cebolla* [VM99, DMT99].

Existen numerosos sistemas, tanto teóricos como prácticos, de este tipo. Los más famosos entre los orientados al objeto son, sin duda, los *Filtros de Composición* de Sina [AWB+93, Ber94]. Una forma avanzada de los mismos es el concepto de *Capas* del modelo LAYOM de Jan Bosch [Bos96c]. Por otra parte, entre los trabajos basados en meta-actores destacan el desarrollo original de Callsen [Cal94], ya citado, y los trabajos posteriores de Svend Frølund en las nociones de *Sincronizadores* [FA93] y *Conjuntos de Mensajes* [FA96].

En cierto modo relacionado con esta idea, pero en un contexto completamente diferente, surge el concepto del *desarrollo de software orientado a aspectos* (AOSD) [OK02], también conocido como *separación de intereses avanzada*. Se trata de otra forma de afrontar el problema de la reutilización de componentes, en este caso cambiando el criterio inicial de descomposición: de modo que los sistemas no se modularizan sólo en base a las nociones de objetos, sino que se realiza también una división ortogonal en los denominados *aspectos* [KLM+97], *sujetos* [OKH+95] o *dimensiones* [OT00], entre otros nombres.

Las distintas encarnaciones de la *programación orientada a aspectos* (AOP) permiten definir un sistema como la unión de estos aspectos, cada uno de los cuales se centra en una característica global (*cross-cutting*) del sistema. De este modo, un aspecto sólo proporciona un *fragmento* de cada uno de distintos componentes del sistema, tanto de sus datos como de su comportamiento. Los distintos aspectos se especifican de manera independiente, y cada uno constituye una visión parcial del sistema. El propio sistema se construye mediante un proceso de *composición de aspectos*, que es realizado por un elemento global denominado *tejedor de aspectos* o *weaver*. Este proceso de combinación permite disponer de software más legible, ya que las especificaciones son modulares; y también más fáciles de reutilizar, pues los aspectos pueden recombinarse en otro sistema. Por supuesto, todas las formas de AOP son meta-programación; pero además, los enfoques más interesantes han adoptado una perspectiva basada en reflexión. Este punto no ha sido aún adecuadamente explotado, aunque ha sido muy tratado en los últimos cuatro años. Cabe esperar un desarrollo considerable en esta línea, en un futuro cercano.

El concepto de *aspecto* va más allá de la simple AOP: constituye un marco de trabajo con unas posibilidades y unos problemas mucho mayores de lo que aquí se ha dado a entender. Su influencia reciente es enorme, aunque sólo fuera por el modo en que ha sabido hacer atractiva la tantas veces reiterada idea de la separación de intereses [HL95]. Desde el punto de vista de Arquitectura de Software, resulta obvia su analogía con el modelo de múltiples vistas [AWG99]. Sin embargo, esto se refiere más bien a la combinación de "aspectos formales" [BB98], más que a la idea de disponer de un sistema generativo basado en la arquitectura.

4.7 Reflexión y Arquitectura de Software

A lo largo de este capítulo se ha intentado dar una breve visión general de la Reflexión, considerada dentro del contexto de la Informática, pero totalmente independiente. Sin embargo,

debe recordarse que la Reflexión se introduce en este trabajo con el objetivo de combinarla posteriormente con la Arquitectura de Software, dando lugar a un nuevo modelo de descripción arquitectónica; esta tarea será de hecho realizada en el capítulo 5 y subsiguientes.

No obstante, hasta el momento no se ha considerado, específicamente, si existe ya alguna combinación de este tipo. Como ha podido verse durante esta exposición, algunos de los conceptos de Reflexión, como las arquitecturas de meta-nivel, los modelos de capas de cebolla, la reflexión estructural o el modelo de reificación de canal, resultan perfectamente compatibles con la Arquitectura de Software; por tanto, una propuesta en esta línea no resultaría extraña. De hecho, aunque no son muy habituales, sí se han realizado algunas aproximaciones en este sentido, como podrá verse más abajo; pero prácticamente todas ellas se han centrado en aspectos laterales, o han sido muy poco desarrolladas.

En este apartado, por tanto, se revisarán muy brevemente las distintas combinaciones existentes –aun cuando lo sean simplemente en potencia– entre Reflexión y Arquitectura, con el objetivo de enmarcar adecuadamente el trabajo realizado en capítulos posteriores.

Estas propuestas puede agruparse en cinco categorías; todas ellas se relacionan con Reflexión, pero sólo cuatro de ellas utilizan explícitamente técnicas de Arquitectura, y la que *no* lo hace es, precisamente, la más extensa.

□ 1. **Arquitectura con Reflexión**. *Modelos de Arquitectura de Software que utilizan y describen Reflexión de manera explícita.*

Se trata, por tanto, de modelos y lenguajes de Arquitectura de Software que incluyen de manera explícita soporte para conceptos reflexivos, su descripción y su manipulación. En este grupo se engloba, precisamente, la propuesta de esta tesis; en ella se describe un modelo reflexivo para Arquitectura de Software, denominado Marmol (§7), y un Adl parcialmente basado en dicho modelo, al que se denominará $\mathcal{P}i\mathcal{L}ar$ (§6).

Aparte de ésta, en realidad sólo se ha hecho *una* propuesta en este sentido: concretamente, los trabajos de Walter Cazzola y Francesco Tisato en el concepto de Reflexión Arquitectónica, en cuyo contexto se describe tanto un modelo como un lenguaje de descripción, que reciben la denominación global de PiL (*Programming-in-the-Large*) [CSST98, CSST99].

La expresión *Reflexión Arquitectónica* no hace referencia, como podría pensarse, a una perspectiva de la Reflexión desde el dominio de la Arquitectura de Software, sino a la propuesta de una técnica reflexiva basada en la descripción arquitectónica. Ésta consiste en la inclusión de dicha descripción como parte de un sistema en ejecución, de manera que los cambios en uno se reflejen en la otra, y viceversa, asegurando de este modo la consistencia entre ambos.

El lenguaje PiL se define con el fin de poder especificar adecuadamente esta estructura. Se trata de un Adl que describe los conceptos usuales, pero al que se le añade además una noción explícita de *metacomponente* y una relación de reflexión. No obstante, la definición del concepto es *sintáctica*, lo que significa que hay una distinción sintácticamente explícita entre los componentes de distintos niveles, lo que se deriva en una cierta rigidez. Por otra parte, el vínculo reflexivo –entre componentes y metacomponentes– es también permanente; y la descripción de comportamiento está basada en *reglas* asociadas a cada componente, lo que a priori define un esquema con poco potencial para el dinamismo. En todo caso, este lenguaje ha sido apenas desarrollado, y su definición no ha tenido continuidad.

Lo cierto es que Cazzola ha sido uno de los principales impulsores de la difusión de la Reflexión a otros contextos. No sólo ha estudiado su relación con la Ingeniería de Software [CST00], sino que ha participado en la más reciente revisión de la noción de arquitectura de meta-nivel [CCL00b]. Es además el autor de la clasificación de los modelos reflexivos orientados al objeto [Caz98] examinada en la sección 4.5.1, así como del propio modelo de *reificación de canal* (MRC) [ACDG95], que exploró posteriormente en su tesis mediante la definición –en el contexto de Java– del sistema mChaRM [Caz00]. Sin embargo, su aplicación a la Arquitectura de Software es bastante incompleto, y no deja de resultar sorprendente que ni siquiera intente definir una noción reflexiva de conector, basada en su propio modelo MRC.

El enfoque de Cazzola es probablemente el único comparable al que se hace en esta tesis; tiene muchos puntos de coincidencia y, no obstante, también tiene un buen número de limitaciones. Parece claro que PiL fue concebido únicamente para la descripción de un modelo reflexivo de objetos, y el desarrollo de la noción de reflexión arquitectónica. Sin embargo, no ha sido nunca utilizado para la descripción de dinamismo, ni se ha pretendido nunca explotar su potencial reflexivo. Finalmente, el desarrollo ha carecido totalmente de continuidad, hasta el punto de que los trabajos posteriores del propio Cazzola siguen una línea diferente.

□ 2. **Reflexión de la Arquitectura**. *Uso de técnicas reflexivas en modelos de Arquitectura de Software, sin que éstos desarrollen su potencial descriptivo.*

Se trata, por tanto, de modelos de Arquitectura de Software que utilizan en su interior, conscientemente o no, algún tipo de técnica reflexiva. Sin embargo, esta técnica no se utiliza con fines de descripción, sino que aparece tan sólo como la solución particular utilizada para algún tema concreto, en especial la implementación de conectores.

Este grupo se divide, principalmente, en tres tipos de propuestas:

- Trabajos relacionados con la definición de *conectores reflexivos*, cuyo único objetivo es facilitar la implementación de una arquitectura, o proporcionar un mecanismo de generación. En este sentido han enmarcarse los *conectores abiertos* de Uwe Aßmann [ALP99], los *conectores instrumentados* de Bob Balzer [Bal97], o el modelo FLO para *conectores ejecutables* de Stéphane Ducasse y Tamar Richner [DR97].

- Trabajos relacionados con el concepto de *meta-actores*, una noción reflexiva introducida en el Modelo de Actores [Agh97] con el fin de proporcionar flexibilidad. Dentro de éste, los trabajos específicamente orientados a la descripción arquitectónica, en particular los de Mark C. Astley, que ya fueron comentados en la sección 3.13.2.

- Finalmente, modelos vinculados al campo del *middleware* reflexivo, pero que han alcanzado suficiente nivel descriptivo como para acercarse al de un ADL. Los ejemplos más significativos son la poderosa combinación de Aster y OpenORB concebida por Valèrie Issarny y Gordon Blair [BBI+00], y los trabajos de Orlando Loques en el lenguaje de configuración del sistema R-RIO [LSLL00].

Aunque no están directamente vinculados con la Arquitectura de Software, no debe dejar de señalarse que muchos de los restantes modelos de capas de cebolla podrían agruparse en la

primera de estas tres categorías. Esto incluye algunos esquemas ya mencionados (§4.6), como el modelo de *Filtros de Composición* de Sina, definido por Mehmet Akşit [AWB+93, Ber94, BA00]; o el modelo de *capas* del sistema LAYOM, desarrollado por Jan Bosch [Bos96b, Bos96c, Bos96a] a partir del anterior. Ambos se vinculan más bien a la programación concurrente orientada al objeto, pero utilizan una notación específica, y podrían ser fácilmente adaptados para cumplir objetivos similares a los descritos por el modelo FLO.

☐ 3. **Arquitectura de la Reflexión**. *Uso de los modelos de Arquitectura de Software para la descripción de la estructura de esquemas reflexivos.*

La estructura de la mayoría de los esquemas reflexivos es a menudo compleja, lo que contribuye en gran medida a dificultar su difusión y utilización. En este sentido, sería interesante proporcionar una descripción precisa de estas estructuras en términos de alguna notación de Arquitectura de Software, con el fin de facilitar su comprensión.

Sin embargo, este tipo de tarea no ha sido acometida prácticamente nunca. La explicación más lógica para esto es que los ADLs clásicos no están adecuadamente equipados para ello. Esto se demuestra plenamente en el breve trabajo de Ian Welch [WS98] al respecto, posiblemente el único de este tipo que ha conocido cierta difusión. Su desarrollo resulta realmente impreciso, y su principal mérito es intentar plantear la especificación formal de un MOP en términos de Wright. Sin embargo, en este sentido, se mezclan niveles de abstracción al intentar modelar la reificación –o estrictamente, la conexión causal– mediante un *conector*, como si se tratase de una interacción convencional, lo cual es claramente inadecuado.

☐ 4. **Reflexión con Arquitectura**. *Tratamiento específico de Reflexión, en un contexto que también considera aspectos de Arquitectura de Software.*

Las propuestas de este grupo incluyen a trabajos que, utilizando reflexión de manera explícita, consideran también de algún modo detalles relacionados con la Arquitectura de Software, siquiera indirectamente, y realizan por tanto una combinación informal de ambos.

Aquí se engloban, pues, los estudios del grupo de Oscar Nierstrasz sobre *lenguajes de composición*, y en particular los trabajos de Theo Dirk Meijler [Mei95, MN97]. Su enfoque introduce explícitamente la noción de meta-objetos en un paradigma de componentes –objetos activos–, con el fin de poder expresar la composición al nivel de los meta-objetos. El lenguaje recibe, además, la influencia directa de Wright (§3.7.1), como lenguaje de descripción de arquitecturas, y de las GSP de Cíaran McHale [McH94], como modelo de coordinación.

Esta línea de trabajo tuvo su continuidad [LSN96, SL97] en los desarrollos posteriores de Markus Lumpe [Lum99] y Jean-Guy Schneider [Sch99], que derivaron en la elaboración del *lenguaje de composición* Piccola [ALSN00, AKN00, AN00], participando tanto en su implementación como en su soporte formal, el cálculo-$\pi\mathcal{L}$, del que existen dos variantes. Piccola se concibe, por definición, como un sistema que considera la coordinación, configuración y adaptación de componentes, dando lugar a la construcción de arquitecturas de software. Además, su estructura interna hace un uso notable de la reflexión, a través de la definición de un meta-nivel explícito. Sin embargo, su perspectiva es ante todo la de un lenguaje de *scripting*, y su principal aplicación ha sido el modelado de objetos y los conceptos relacionados.

Por otra parte, se ha de considerar en este grupo también el trabajo de Claudia Marcos [Mar01] sobre la construcción reflexiva de patrones de diseño; si bien no utiliza un enfoque explícitamente arquitectónico, si considera en todo momento conceptos análogos, aunque fuertemente ligados a aspectos de implementación. Este desarrollo ha conocido continuidad en los trabajos de Jane Pryor [PBC00, VRMP01] sobre orientación a aspectos.

Finalmente, pueden enmarcarse también en esta línea los trabajos iniciales sobre el lenguaje de coordinación MeldC [KHL+93], que también tenía una naturaleza reflexiva; no obstante, éstos no tuvieron continuidad, y en la actualidad éste ha sido completamente olvidado.

□ 5. **Reflexión sin Arquitectura**. *Tratamiento específico de Reflexión, donde se manipulan conceptos* análogos *a los de la Arquitectura de Software.*

Las propuestas de este grupo incluyen a trabajos que, a pesar de concentrarse exclusivamente en el dominio de Reflexión, dan especial importancia a aspectos estructurales, o se relacionan estrechamente con campos muy cercanos a la Arquitectura de Software. En todos ellos se manejan conceptos que serán relevantes para una perspectiva arquitectónica de la reflexión; algunos de ellos serán tratados de manera específica en el Capítulo 5.

- En primer lugar, cabe mencionar a los sistemas de programación concurrente orientada al objeto que definen un modelo subyacente reflexivo, en algunos casos particularmente complejo. El primero de estos desarrollos, históricamente hablando, fue el lenguaje ABCL/R de Akinori Yonezawa [WY88]. Entre sus sucesores, cabe mencionar dos líneas de investigación principales: la desarrollada en Japón por el grupo de Yukata Ishiwaka sobre el lenguaje AL/D-1 [OIT93], en el que se define el primer marco de reificación multi-modelo, denominado MMRF; y la desarrollada en Dublín por el grupo de Vinny Cahill sobre el modelo Iguana [RC00], donde resulta especialmente interesante el trabajo de Brendan Gowing sobre el concepto de categorías de reificación [GC96]. Este último trabajo está evolucionando hacia su integración con los modelos de componentes, en el Modelo-K de Jim Dowling [DSC+00, DC01b, DC01a].

- En segundo lugar, son especialmente interesantes los desarrollos vinculados a sistemas operativos reflexivos. De entre la amplísima variedad de trabajos existentes, cabe mencionar en especial el desarrollo del sistema ApertOS de Yasuhiko Yokote [Yok92, YT87], fundamentado en el Modelo de Campo Computacional de Mario Tokoro [Tok90], que introdujo el concepto de meta-espacio. Entre los restantes, posiblemente los más difundidos son los trabajos de Jeff McAffer sobre el sistema CodA [McA95, McA96].

- En tercer lugar, se deben mencionar los trabajos vinculados a sistemas de *middleware* reflexivo, cuya difusión ha sido enorme en los últimos años, y que son parcialmente responsables del interés actual en el campo. En buena medida, la existencia esta línea ha sido posible gracias al marco proporcionado por CORBA, que en su encarnación básica es ya en parte reflexivo. El primer sistema de este tipo plenamente reflexivo fue el OpenCorba de Thomas Ledoux [Led99, Led98]. En un contexto similar se enmarcan los trabajos del grupo de Jean-Charles Fabre [KFRGC98b, KFRGC98a, Mar99] sobre tolerancia a fallos; en éstos se incluye uno de los escasos estudios formales existentes sobre el tema, junto a la adaptación

del modelo de actores denominada TLAM [VT01]. Pero sin duda el tratamiento más completo y potente es el del OpenORB de Gordon Blair [BC98, BCRP98, BBI+00, BCA+01], que se basa en un esquema multi-modelo análogo al de AL/D-1, ya mencionado.

- En cuarto lugar, merecen una mención aparte los trabajos sobre persistencia ortogonal del grupo de la Universidad de St. Andrews, y en particular los estudios de Graham Kirby [Kir92, KMS98], que proporcionan una de las caracterizaciones más completas del concepto de reflexión estructural –o lingüística– en un contexto semiformal.

- En quinto lugar, no deben dejar de mencionarse los trabajos pioneros en la adaptación de la reflexión a la orientación al objeto, que ya dan una descripción muy completa de sus aspectos estructurales. Entre ellos cabe mencionar, de nuevo, los trabajos de Pattie Maes [Mae87a, MN88], así como la tesis doctoral de Patrick Steyaert [Ste94].

- Especialmente significativos han resultado en los últimos años las distintas adaptaciones del concepto en el contexto del lenguaje Java, completando el enfoque del API de Reflexión, que solamente proporciona introspección. Entre éstos destacan, por su diversidad, el sistema MetaXa de Michael Golm [Gol97], el Dalang de Ian Welch y Robert Stroud, el Guaraná de Luiz Buzato y Alexandre Oliva [OB98, OCGB98, Oli98], el Meta/J de Rémi Douence [DS99], basado en el modelo original de Smith, o el OpenJava de Shigeru Chiba [CTKI00], que adapta sus resultados anteriores en OpenC++, donde introdujo el concepto de meta-hélice [Chi95, CKL96]. Todos ellos han desarrollado una serie de conceptos estructurales; por otra parte, el estudio general de Julien Vayssière [Vay00] realiza una breve comparación de sus diferentes enfoques.

- Finalmente, cabe destacar algunos de los estudios comparativos realizados en los últimos años, en los que se resaltan algunos de los aspectos más interesantes de la noción de reflexión, desde una perspectiva inusual. Cabe mencionar los resúmenes de algunas conferencias, como [CCL00b] y [BCCK01]; pero sin duda los más interesantes son la revisión general del campo de François-Nicola Demers [DM95], y el estudio de modelos orientados al objeto de Walter Cazzola, ya mencionado [Caz98].

En definitiva, puede concluirse que, aunque existen numerosos puntos de contacto entre muchas nociones y modelos de la Reflexión y la Arquitectura de Software, sobre todo desde su introducción de la primera en la orientación al objeto, lo cierto es que nunca hasta ahora se ha planteado una plena integración de los dos campos. Los únicos intentos importantes en este sentido han sido el sistema PiL de Cazzola, que fue apenas esbozado y no ha conocido continuidad, y los trabajos sobre lenguajes de composición basados en el cálculo-$\pi\mathcal{L}$–que, curiosamente, también ha sido designado ocasionalmente con el nombre de PiL- de Schneider y Lumpe, del cual este trabajo de tesis es en cierto modo deudor, al menos en origen [LSN96].

4.8 Conclusiones

A lo largo de todo este capítulo se ha dado una perspectiva general del campo de la Reflexión, poniendo especial énfasis en sus conceptos estructurales, y resaltando aquéllas de entre sus

aplicaciones que tienen mayor interés en el contexto de esta tesis, o se relacionan de algún modo con las ideas y perspectiva de la Arquitectura de Software.

Se han identificado adecuadamente nociones como reificación, reflexión, introspección, intercesión, conexión causal, meta-nivel, meta-espacio o arquitectura reflexiva, todas las cuales serán utilizadas en capítulos siguientes. También resultan particularmente importantes las diferentes clasificaciones expuestas (§4.4.1), y en particular la distinción entre Reflexión Estructural –o Lingüística– y Reflexión de Comportamiento. Ambos conceptos tendrán una influencia decisiva en el desarrollo de los capítulos 5 y 7.

Por supuesto, no se ha dado una visión completa del tema. Incluso sin considerar muchas de sus vinculaciones con la Inteligencia Artificial, o sus aplicaciones en sistemas operativos y lenguajes de programación, se trata de un campo muy amplio. No se han mencionado, tampoco, los distintos enfoques formales existentes [Har95], algunos de los cuales parten de posturas filosóficas radicalmente opuestas; esto incluye, por ejemplo, a la lógica policontextual de Gotthard Gunther, que introduce el concepto de relación proemial [KM96a], las aplicaciones de Manuel G. Clavel en lógica de reescritura [CM96a, CM96b], o incluso la reflexión monádica de Andrezj Filinski [Fil94]. Hay, en definitiva, toda una serie de detalles suplementarios, suficientes por sí solos para multiplicar varias veces la extensión de este capítulo. No obstante, lo único que se pretendía era dar una visión lo suficientemente completa de la complejidad del concepto, así como de las diferencias existentes entre las múltiples perspectivas del mismo.

CERVANTES

Parte IV

Arquitectura de Software Dinámica mediante Reflexión

BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Capítulo 5

El Modelo Marmol

En esencia, son [*los* nombres] (...) *los que nos dan acceso a* (...) *todo: pero tendemos a pensar en* variables, direcciones, lugares, punteros*, o* canales *de manera diferente en contextos diferentes, porque dan acceso a cosas diferentes.*
Robin Milner [Mil93a]

5.1 Introducción

El objetivo final de esta tesis es la incorporación de Reflexión en Arquitectura de Software, con el fin de permitir la especificación de dinamismo de una manera natural y unificada, y manteniendo además la descripción de la estructura arquitectónica a todos los niveles. Sin embargo, se desea hacer que esta incorporación se haga en términos generales, con independencia de un lenguaje de descripción (Adl) concreto. Debido a esto, los aspectos principales de la misma serán descritos en primer lugar en el contexto de un modelo genérico, adaptable a cualquier enfoque lingüístico particular, que recibe el nombre global de Marmol. Sólo después se planteará su aplicación a un lenguaje concreto, que será descrito en los Capítulos siguientes.

Marmol es un acrónimo de M*eta* AR*chitectural* MO*de*L. Se trata de un modelo básico para Arquitectura de Software, al cual se le han incorporado las nociones reflexivas de manera explícita. Hay dos aspectos fundamentales en este modelo: en primer lugar, la selección concreta de conceptos reflexivos que se consideran relevantes para la descripción arquitectónica; en segundo lugar, la expresión de estos conceptos en el contexto adecuado, y con la precisión requerida. Ninguna de las nociones incorporadas en el modelo es, entonces, totalmente nueva; sin embargo, tanto el planteamiento realizado como el desarrollo concreto de las mismas es totalmente original, y plantea explícitamente una serie de distinciones poco habituales en los tratamientos informales de la Reflexión. Algunas de ellas serán fundamentales en Capítulos posteriores.

Este Capítulo comienza con una revisión de los distintos esquemas de Arquitectura de Software Dinámica desde el punto de vista de Reflexión, apoyando la hipótesis de partida de esta tesis. Continúa con una definición informal del modelo Marmol, descrito aquí como un enfoque para la descripción arquitectónica que incorpora explícitamente reflexión. Se manifiesta la necesidad de pasar de un modelo informal a uno formal; tras debatir la importancia de establecer el concepto de *nombre* como noción fundamental de la formalización, se pasa a realizar la

descripción del modelo formal, en términos de álgebra de conjuntos y de relaciones.

El modelo se describe entonces en tres etapas: en primer lugar se plantea un modelo básico ($\aleph_\Sigma$), que describe las nociones comunes de la Arquitectura de Software en términos de teoría de relaciones. A continuación se incorpora el soporte de reflexión ($\aleph_M$), lo que constituye la primera encarnación de Marmol propiamente dicha. Finalmente, se elabora una versión restringida de este modelo ($\aleph_\Pi$), que colapsa parcialmente las nociones de tipado y reificación, y que hace más sencilla la descripción de ciertas ideas. Ésta es la versión a la que se hará referencia en el resto de la tesis, cuando se hable del modelo Marmol.

5.2 Arquitectura de Software Dinámica con Reflexión

En la sección 4.7 se han discutido los vínculos existentes entre la Reflexión y la Arquitectura de Software, abarcando desde la descripción arquitectónica de la reflexión, hasta los modelos reflexivos de arquitectura. Estos últimos han planteado, siquiera de manera tímida, las posibilidades de la combinación de los dos conceptos, si bien éstas no han sido nunca explícitamente desarrollada. En particular, ninguna de estas propuestas ha planteado explícitamente la idea de enfocar el *dinamismo* en Arquitectura de Software desde un punto de vista reflexivo.

La hipótesis de partida de esta tesis es, precisamente, que cualquier tipo de dinamismo arquitectónico puede expresarse mediante Reflexión. Así, en lugar de incorporar un conjunto de comandos extra, o una estructura particular, el modelo que aquí se describe incorpora únicamente un nuevo concepto, que permite la introducción de todos ellos de manera simultánea y unificada; sus efectos en el modelado aparecen con posterioridad.

El objetivo, tanto de este Capítulo como de toda la tesis, es plantear la introducción de Reflexión en Arquitectura de Software de una forma coherente. Aunque el objetivo de partida era únicamente la expresión de dinamismo, el resultado es lo bastante general para traer consigo toda una serie de posibilidades extra: precisamente, la mayor ventaja de la Reflexión es el modo en que facilita la extensibilidad. Así, por ejemplo, en un esquema arquitectónico reflexivo será mucho más sencilla la incorporación de seguridad, la integración de múltiples vistas, e incluso la descripción de un sistema en base a aspectos (§4.6).

No obstante, en esta tesis nos centraremos básicamente en el dinamismo. Con el fin de justificar, siquiera de manera intuitiva, la afirmación de que la reflexión es suficiente para expresarlo en cualquiera de sus formas, se tomará de nuevo la clasificación de modelos de dinamismo, desarrollada en la sección 3.14, y se revisará a la luz de los conceptos reflexivos. Podrá comprobarse, de este modo, que todos aquellos modelos pueden ser descritos, sin cambios, en términos de Reflexión. Obviamente, esto no constituye una demostración formal, pero apoya, con una argumentación sólida, la afirmación inicial de que la incorporación de este concepto es suficiente para la especificación de dinamismo.

Por tanto, se revisarán brevemente los modelos allí descritos, mencionando únicamente cómo habrán de considerarse desde un punto de vista de reflexión. Se pueden utilizar, incluso, las mismas figuras; las mismas partes que antes se señalaban como *dinámicas* pueden verse ahora como *reflexivas*. Sea entonces:

1. **Supervisor**. El supervisor se definía como una entidad externa que actúa sobre el sistema: es entonces, por definición, una *meta-entidad* del mismo, que adopta distintas formas según

la variante del modelo.

(a) **Configurador**. El configurador es un componente con la capacidad de gestionar el sistema, o parte de él. Es entonces, de nuevo por definición, un *componente de nivel meta*, a menudo incluso un *metacomponente*. Como puede verse en la Figura 3.22, su relación con el elemento dinámico tiene una naturaleza diferente a la de una conexión: desde este punto de vista, podría tratarse con naturalidad de una reificación, si bien no tiene por qué tratarse necesariamente de una relación directa.

(b) **Coordinación**. Esta opción, en cualquiera de sus dos variantes, supone la existencia de cierta estructura, capaz de capturar la información relevante sobre un sistema y garantizar que éste cumple ciertas leyes. Esta estructura será, en términos reflexivos, un *meta-espacio* (§4.3.4), dentro del cual se podrán describir las reglas de distintas maneras. Concretamente:

- **Espacio Común.**. En este caso, el meta-espacio definirá una influencia común a todos los elementos dinámicos. El hecho de concebirlo como un meta-espacio permite considerarlo alternativamente como un único metacomponente complejo, o bien como una estructura más elaborada.
- **Control Externo.**. En este caso, el meta-espacio puede ser único o múltiple: la característica esencial que lo distingue del anterior es que en este caso estará explícitamente estructurado, es decir, que se dividirá en módulos, distribuyendo su influencia en múltiples puntos.

(c) **Oráculo**. Dado que no puede ser modelado, el oráculo tampoco puede ser descrito por medios reflexivos. No obstante, se sabe que estará situado en un espacio externo, desde donde mantiene la capacidad de manipular la arquitectura. Por tanto, es también por definición una *meta-entidad*, pero sin embargo *no es* reflexiva.

2. **Componente Mixto**. Se trata, en este caso, de componentes normales que se están comportando, en realidad, como *meta-componentes*, ya que alteran y modifican su propia arquitectura. Normalmente, en su descripción se están mezclando aspectos que deberían permanecer separados, relativos respectivamente a dinamismo interactivo y estructural. La Figura 3.26 refleja esta naturaleza dual, de manera esquemática. Obviamente, sería interesante recuperar dicha separación, ya que simplificaría su definición: la parte dinámica sería en ese caso descrita como un metacomponente.

3. **Conector Mixto**. En este caso, los conectores adquieren mayor responsabilidad que simplemente gestionar la comunicación, ya que son también los encargados de controlar la evolución del sistema. Han de alterar la estructura del sistema en función de las interacciones que interceptan. Son, de nuevo, análogos al caso anterior, ya que están usando la propia arquitectura de software como un conjunto de datos; por tanto se comportan, igualmente, como *meta-componentes*.

En resumen, cualquiera de los seis modelos de dinamismo arquitectónico puede ser expresado como una relación reflexiva entre una o varias meta-entidades y los elementos de la arquitectura. Se postula, por tanto, que dotar a un modelo con la capacidad para describir estas meta-entidades

es suficiente para lograr, asimismo, la descripción de cualquiera de estos esquemas, excepto los aspectos relativos al oráculo, que no son modelables.

5.3 Marmol: Modelo Informal

En la sección anterior (§5.2) se ha mantenido y justificado la hipótesis de que una descripción arquitectónica en la que se hagan explícitos tanto los meta-niveles como las relaciones reflexivas inherentes debería ser capaz de expresar, por lo menos, unos grados de dinamismo similares, si no mayores, a los de las propuestas revisadas en el Capítulo 3.

Con el objetivo de formalizar esta intuición, se propone en este capítulo un nuevo modelo de especificación arquitectónica, que recibe la denominación de Marmol, y se concibe con independencia de cualquier lenguaje. En el capítulo siguiente este modelo será utilizado de hecho para la definición de un Adl reflexivo, que desarrolla esta misma idea y es uno de los temas centrales de esta tesis. Sin embargo, el modelo es lo suficientemente genérico como para poder ser aplicado, con las modificaciones relevantes, a cualquier lenguaje ya existente[1].

La idea de fondo es disponer de dos descripciones, no necesariamente separadas de manera explícita, que muestren dos perspectivas diferentes de la arquitectura de un sistema. La primera, que se corresponde con un nivel base, consistirá en la especificación de la estructura, del modo convencional en el campo. La segunda, el nivel meta, describe las estrategias globales de control y modificación desarrolladas sobre esta estructura. Estos dos niveles estarían *conectados casualmente*, de modo que los cambios en uno de ellos afectarían al otro. El esquema se extiende de manera obvia desde dos hasta cualquier número de niveles. Sin embargo, habitualmente no habrá más de tres; más allá se alcanzan unos niveles de abstracción poco razonables.

Marmol es un modelo de *especificación*, no un estilo arquitectónico. No intenta forzar la construcción de sistemas en una (infra-)estructura particular. Define un modo para *describir* arquitecturas, que intenta ser lo bastante flexible como para poder incluir a cualquier sistema concebible, en particular un sistema reflexivo. Es, en definitiva, una *plantilla* que sirve para organizar las descripciones arquitectónicas, no un patrón limitado para una familia concreta. En términos globales, su objetivo principal es identificar cuáles, de entre las principales nociones reflexivas, son relevantes para la descripción arquitectónica.

El modelo Marmol establece, entonces, los siguientes puntos:

- **No hay límite al número de meta-niveles**. Cualquier arquitectura dada combina como mínimo un sistema base y un meta-sistema, pero un meta-nivel podría tener su propio meta-meta-sistema, y de ahí en adelante. Más formalmente, *hay una torre reflexiva infinita, estructurada en pares base-meta.* Por supuesto, la recursión no necesita ser infinita; sólo es explorada hasta donde sea necesario, describiendo solamente los niveles precisos, con una profundidad que se denomina *grado de introspección.*

- **No se supone nada sobre la semántica de la relación base-meta**. Ésta debería ser definido por cada Adl que quisiera utilizar el modelo. En particular, no se fuerza el uso

[1] Esto es, cualquier lenguaje que admite al menos los principios conceptuales más básicos del campo. Este criterio es mucho menos restrictivo, por ejemplo, que los utilizados en la clasificación de Adls de Medvidovic [MT00] o la caracterización de lenguajes de alto nivel de Garlan y Shaw [SG94].

de un modelo tipo-Mmc (§4.5.1), que colapse esta relación con la de *tipado*. Tampoco se asume una interpretación metacircular, aunque en la práctica ésta será la más razonable; cualquier otra exigiría un proceso intermedio de traducción. En resumen, esto significa que un meta-nivel no está expresando necesariamente las abstracciones básicas (el vocabulario) de su nivel base, ni en el mismo lenguaje. Por supuesto, ambas ideas podrán ser expresadas dentro del marco; pero también podrán serlo otras no necesariamente ortogonales, como por ejemplo la definición del comportamiento de los componentes (base) a partir de la composición de aspectos o políticas definidas en el meta-nivel.

- **Todo componente (base) tiene exactamente un meta-espacio asociado**, que reune toda la meta-información asociada con él. Este meta-espacio puede hacerse explícito, en la forma de un metacomponente compuesto, o no. La relación entre este meta-espacio y el componente base asociado indica una *conexión causal*, que normalmente adopta la forma de una operación de *reificación*, de base a meta, o *reflexión*, de meta a base. Para simplificar, se asume una relación bidireccional, isomorfa al concepto de conexión causal, pero que se denominará también, por metonimia y abuso del lenguaje, *reificación*, término que se considera el más significativo.

- **Un meta-espacio se compone de uno o más meta-componentes y cero o más componentes-meta**. En primer lugar, se denomina *componente de nivel meta* a todo componente arquitectónico situado en un cierto meta-nivel. Dentro de éstos, se denomina *metacomponente* a todo aquél que mantiene un vínculo de reificación directa, en el sentido de reflexión, hacia un componente-base dado, que recibe el nombre de *referente* o *avatar*. Finalmente, se dice que todo componente que interactúa con un metacomponente en su mismo nivel es un *componente meta* del avatar implicado. El meta-espacio se forma, pues, por la reunión de elementos arquitectónicos de ambos tipos. Se define un meta-espacio por cada avatar, lo que significa que normalmente habrá varios. Es obvio que no tienen por qué ser disjuntos: un mismo componente-meta puede interactuar con varios metacomponentes diferentes, y pertenecer por tanto a varios meta-espacios.

- **La relación de reificación tiene cardinalidad múltiple**. El mismo metacomponente puede reflejarse en varios avatares, y a su vez un avatar puede ser reificado por varios metacomponentes, que se situarán entonces en el mismo meta-espacio. Cada correspondencia se indica, para mayor claridad, como un *enlace* de reificación independiente; pero la relación que los reune, y que expresa la correlación entre ambos, es por tanto múltiple en ambos sentidos.

- Con las herramientas conceptuales adecuadas, puede suponerse que la influencia de un metaespacio sobre un avatar concreto puede ser agrupada en un único meta-componente, al que se denomina **reflector**. Para poder hacer esto, es necesario haber utilizado un formalismo –composicional– adecuado, por lo que el marco no puede asumir que esto vaya a ser posible. El concepto existe, simplemente, para proporcionar la abstracción pertinente en el caso de que esta composición sea en efecto posible.

- Los Adls convencionales especifican primero un conjunto de tipos de componente, a los que desde aquí se denominará *arquetipos*, y a continuación agrupan sus instancias en

una configuración. Entonces, **todo arquetipo puede ser reificado como un meta-componente**; desde un punto de vista reflexivo, se puede considerar que una descripción de esta forma, en realidad, está mezclando dos niveles: el vocabulario básico del sistema y sus ejemplares, respectivamente meta y base. Esto supone el colapso de las relaciones de reificación y tipado (§4.2), y desarrolla un *enfoque de metaclase*, y como se ha dicho, es en todo caso opcional en Marmol. Sin embargo, utilizarlo hace notablemente sencillo expresar el dinamismo estructural, e incluso el arquitectónico: en el nivel meta, un nuevo arquetipo no sería más que un nuevo componente. Se trata, pues, de una opción que no debería ser –ni será– ignorada.

- Por un razonamiento similar, en Marmol se pueden concebir *conectores* como un concepto derivado, obtenido como la reificación de las *conexiones* simples en un arquetipo complejo. Este punto será ampliamente desarrollado, en el contexto del lenguaje $\mathcal{P}i\mathcal{L}ar$, a lo largo de la sección 6.11. Sin embargo, no será apenas considerado durante este capítulo, al no tratarse de una noción general del modelo.

Como ya se ha dicho, Marmol no depende de ningún lenguaje, y ha sido diseñado para ser lo más general posible. Su objetivo principal es aislar las características reflexivas que tendrían interés en la descripción arquitectónica, dando un marco de referencia neutral. En teoría, cualquier Adl podría utilizar su infraestructura, sin más que incorporar las abstracciones necesarias.

Aunque Marmol fue diseñado pensando en dinamismo, no hay ningún motivo para limitar su aplicabilidad a esta idea. El modelo reflexivo abre las puertas a otra serie de conceptos aún no muy tratados por la Arquitectura de Software, y se constituye como un marco ideal para su integración. El enfoque muestra claras analogías con la *orientación al aspecto*, en especial con el enfoque de Andrews [And01]. Algunas de las ideas más interesantes incluyen la especificación de seguridad, la fusión de las múltiples vistas (§2.4.6), e incluso la simple descripción de los propios sistemas software reflexivos.

Obviamente, la descripción informal que acaba de hacerse resulta insuficiente, en especial ante la incorporación de un concepto con este nivel de complejidad. El resto de este capítulo se dedica precisamente al establecimiento de un modelo formal que desarrolle adecuadamente el marco descrito. Para ello, se utilizarán herramientas matemáticas muy habituales, como la teoría de conjuntos y la de relaciones; sin embargo, se ha de partir de un concepto inicial; en nuestro caso, éste será el concepto de *nombre*.

5.4 Sobre el Concepto de Nombre

Como ya ha quedado establecido, lo que intenta describir un modelo de Arquitectura de Software es, ante todo, la *estructura* de un sistema. Esta estructura consta siempre de un conjunto de entidades primitivas o atómicas, tales como los puertos o las instancias de componente; pero lo que la define, en realidad, es el modo en que se *relacionan* entre sí estas entidades, y la manera en que se *accede* e identifica a cada una de ellas. Esta es precisamente la razón por la que el modelo será descrito (§5.5) utilizando esencialmente teoría de relaciones.

En cualquier caso, la arquitectura queda completamente definida una vez que se identifican estas relaciones: algunos elementos se crean por *composición* de otros más primitivos; otros definen distintos tipos de *interacción* entre ellos. Pero todos, en cualquier caso, contribuyen a definir una estructura compleja de relaciones entre componentes. Otros aspectos presentes en el sistema resultan, en última instancia, irrelevantes para la descripción arquitectónica. La *funcionalidad*, por ejemplo, resulta totalmente ortogonal y es abstraída en cualquier Adl; por su parte, el *comportamiento* se refiere fundamentalmente a los protocolos de interacción, que se desarrollan precisamente sobre la estructura de interrelaciones de la que ya se dispone.

Sin embargo, una vez que se ha abstraído la funcionalidad, y que la estructura interna o externa ha sido reflejada como un conjunto de relaciones, todo lo que queda de un componente (o puerto) es su *nombre*. Es ese nombre, en definitiva, lo que permite identificar y localizar a cada uno de las entidades de la arquitectura, desde los simples puertos hasta las configuraciones más complejas. Por tanto, refleja el otro aspecto requerido, esto es, la definición de los *caminos de acceso* a cada elemento. Éstos vendrán dados de nuevo por la arquitectura, pero el nombre será necesario, al menos, para decidir cuándo se ha llegado al lugar requerido.

Debe recordarse que se está tratando con *componentes*, y que éstos definen, ante todo, un encapsulamiento rígido. Dos componentes del mismo tipo son, en principio, idénticos, y resultan por tanto completamente intercambiables. A diferencia de los objetos, carecen por completo de *identidad*. Lo único que distingue a dos instancias dadas es, por tanto, el *lugar* que ocupan en la arquitectura. En definitiva, lo que expresa el nombre de un componente (o puerto) es *cuál* es este lugar. Así pues, los nombres forman parte básica de la estructura, ya que identifican a los lugares de su interior; de hecho, las interrelaciones que la constituyen se describirán, esencialmente, como dependencias entre estos nombres.

En resumen, una vez que se abstraen totalmente los detalles, lo único que queda de cada entidad arquitectónica es una representación atómica, que consiste básicamente en su *nombre*. Este enfoque nominalista no es nuevo en absoluto: de hecho, constituye una de las dos alternativas principales ante casi cualquier problema filosófico, y sus consecuencias en este sentido pueden ser enormes. No obstante, no debe pensarse que esto implica ningún tipo de reduccionismo, ya que simplemente se trata de una abstracción conveniente: por supuesto, no se está afirmando que un componente (o puerto) se reduzca únicamente a su propio nombre, sino tan sólo que, estructuralmente, éste es lo único que es necesario considerar.

Esta misma intuición ha sido reflejada en un buen número de tratamientos formales, vinculados o no a la Informática. El mejor ejemplo, sin duda, es el cálculo-π [Mil99], en cuyo contexto se enmarca la cita de Robin Milner incluida al principio de este Capítulo; pero no es sin embargo, el único. Incluso excluyendo a todos sus derivados, los llamados cálculos nominales –véase el Apéndice B–, existe toda una serie de formalismos basados en el concepto de nombre, tales como el cálculo-λ-N de Laurent Dami [Dam95] o la lógica nominal de Andrew Pitts [Pit01]. Debe tenerse en cuenta, además, que la *nominalización* [Kam92], concebida como una forma general de la gödelización, se plantea como uno de los problemas fundamentales a afrontar en un sistema formal con autorreferencia; esto indica, por sí solo, la importancia de realizar un tratamiento explícito del concepto de nombre.

Además de todo lo dicho, la elección de un enfoque basado en nombres resultará especialmente adecuada de cara a la combinación con el formalismo utilizado en la semántica de $\mathcal{P}i\mathcal{L}ar$, esto es, el cálculo-$\pi$, ya que éste realiza la misma abstracción de partida. Sin embargo, el hecho de

que la correspondencia entre la estructura definida por Marmol y la semántica establecida para $\mathcal{P}i\mathcal{L}ar$ sea conceptualmente casi trivial, no ha hacer pensar que la utilización del modelo será mucho más compleja en un contexto diferente. Por el contrario, la abstracción nominal es tan genérica que resulta compatible con casi cualquier modelo conceptual concebible. Obviamente, disponer de una base común simplifica la tarea; pero incluso cuando se trata con un modelo distinto, la correspondencia es, en la mayoría de los casos, bastante sencilla.

En resumen, se considera que el enfoque nominalista es lo bastante genérico, y que permite expresar con mayor claridad los detalles estructurales, por lo que se considera justificado su uso como abstracción fundamental, no sólo en este Capítulo, sino en la totalidad de esta tesis.

5.5 Marmol: Modelo Formal

Como ya se ha dicho, es evidente que la descripción realizada hasta el momento de Marmol es insuficiente. Aunque resulta intuitiva y clara, una definición meramente verbal nunca es lo bastante precisa. Esto es aún más notorio en un contexto tan complejo como el que se obtiene al desarrollar un esquema reflexivo, donde las distintas referencias a metacomponentes, avatares, meta-espacios o componentes meta pueden llegar a resultar confusas.

Incluso cuando los únicos aspectos que se pretende poner de relieve son los de tipo *estructural*, es obvio que el rigor necesario para poder producir una manipulación consistente hacen necesario llegar a establecer una definición *formal*. Más aún, y puesto que se desea que Marmol sea un modelo totalmente genérico, no se utilizará ningún formalismo específico, como sí se hará al describir la semántica de $\mathcal{P}i\mathcal{L}ar$ (§7). En lugar de ello, se ha optado por utilizar el lenguaje casi universal de la matemática discreta, de manera que se pueda hacer corresponder con cualquier sistema o lenguaje particular, sea cual sea su base conceptual.

Una de las cualidades esenciales del modelado arquitectónico es su visión modular (§2), que puede remontarse incluso hasta los trabajos originales de David L. Parnas [Par72]. Es precisamente esta característica la que hace más lógico optar por un enfoque netamente algebraico, con preferencia a otras opciones. Por tanto, Marmol será definido como una estructura algebraica, es decir, como un *modelo*, desarrollado en términos de teoría de conjuntos y teoría de relaciones (binarias). Se ha considerado que el enfoque relacional no sólo es lo suficientemente potente, sino también el más general posible. Por supuesto, esto no excluye la posibilidad de un tratamiento en términos de notaciones equivalentes, y en especial la teoría de grafos; sin embargo, cualquier desarrollo ulterior sería simplemente complementario al aquí realizado.

Antes de describir formalmente el modelo, conviene establecer desde el principio una clara diferenciación entre los conceptos que éste utiliza. Algunos de ellos son nociones estándar en el estudio de la Arquitectura de Software; mientras que otros proceden de la integración en el campo de las ideas de la Reflexión, y constituyen la aportación específica de Marmol como modelo general. Hay además un tercer grupo de conceptos, que delimitan una variante concreta de Marmol; es ésta la que será usada como base para la definición semántica de $\mathcal{P}i\mathcal{L}ar$, que se tratará en el capítulo 7. Con el objetivo de alcanzar una mayor claridad, se mantendrá esta distinción en la definición del modelo. Así pues, éste se describirá en tres etapas, que pueden ser consideradas a su vez como tres modelos independientes, aunque relacionados entre sí.

El primer modelo, al que denominaremos $\aleph_\Sigma$, describe un *Modelo Arquitectónico General*,

que expresa las nociones estándar de la Arquitectura de Software. En términos de este modelo se podrían describir la práctica totalidad de los lenguajes y modelos arquitectónicos existentes, al menos en lo que respecta a su aspecto estructural. El segundo modelo, denominado $\aleph_{\mathrm{M}}$, es una extensión del primero, al que se le añaden los conceptos propios de Reflexión, integrándolos con los anteriores. Se obtiene así un marco reflexivo para la descripción arquitectónica totalmente genérico: es a éste al que llamaremos, sin excesiva rigidez, *Modelo Meta-Arquitectónico*, o simplemente MARMOL (M*eta* AR*chitectural* MO*de*L). Por último, el tercer modelo, denominado $\aleph_{\Pi}$, es a su vez también una extensión del segundo, en el sentido de que le añade una serie de restricciones. Constituye así una variante concreta del modelo general, que pretende ganar en simplicidad –por economía conceptual– sin sacrificar expresividad. Como ya se ha dicho, este MARMOL *Restringido* es el que servirá de soporte para la semántica al lenguaje $\mathcal{P}i\mathcal{L}ar$ y es, por tanto, el que se utilizará implícita y explícitamente a lo largo de esta tesis.

A continuación se describen formalmente estos modelos, comentando detalladamente su relación con los conceptos arquitectónicos (y reflexivos) relevantes en cada punto.

BIBLIOTECA VIRTUAL

5.5.1 Modelo Arquitectónico General ($\aleph_{\Sigma}$)

Se define el modelo $\aleph_{\Sigma} = \langle V, S_V \rangle$ como la colección formada por un *conjunto universal de nombres* V y una signatura S_V de operaciones sobre los elementos de éste. Se asume que V es un conjunto infinito, aunque el uso que se hará de él será siempre finito[2]. Las operaciones de la signatura toman la forma de un conjunto de relaciones binarias, no necesariamente homogéneas, definidas siempre entre elementos o subconjuntos del conjunto de nombres V.

En la definición de nuestro modelo arquitectónico se parte de una serie de nociones o categorías semánticas. Se trata, en todo caso, de conceptos ya existentes y comúnmente aceptados en el campo; todos ellos ya fueron descritos en el capítulo 2. Con su inclusión se pretenden dos objetivos: por una parte, alcanzar un alto grado de coherencia con la práctica existente; por otra, situar las aportaciones del marco (en los modelos $\aleph_{\mathrm{M}}$ y $\aleph_{\Pi}$) dentro de un contexto natural y adecuado a la Arquitectura de Software.

En concreto, se considerará un marco arquitectónico en el que una configuración se forma por la agregación de componentes tipados, primitivos o –a su vez– compuestos, encapsulados en una interfaz segmentada en puertos. Cada componente se relaciona en dos dimensiones, a saber: la *composición*, estrictamente jerárquica (aun cuando posiblemente parametrizada y/o recursiva); y la *interacción*, marcada explícitamente mediante conexiones –o enlaces– nombrados, que se establecen estrictamente entre componentes del mismo nivel. Se considera a la primera como la dimensión *vertical*, puesto que se expande como una jerarquía; y a la segunda como la dimensión *horizontal*, ya que se desarrolla entre elementos de un mismo plano.

Como rasgo significativo, no se incorpora (aún) el concepto de *conector*, dado que se pretende que $\aleph_{\Sigma}$ sea un modelo general y, como ya se ha explicado, existen multitud de ADLs que no lo consideran. Como se verá en el apartado correspondiente, es posible considerar que, en cierto modo, la noción de *conexión* utilizada en el modelo ya lo incluye. Esto nos permitirá, en todo caso, explorarlo desde una perspectiva completamente distinta, tal como se verá al aplicarlo en $\mathcal{P}i\mathcal{L}ar$ en la sección 6.11. Con esta decisión se simplifica notablemente el modelo; sin embargo,

[2] Este detalle es coherente con la definición de canal o nombre atómico que se hace en cálculo-π [MPW92a].

sigue siendo suficientemente expresivo para la descripción de arquitecturas de software. Esto no quiere decir que la noción de conector no sea conveniente: simplemente, muestra que no es estrictamente *necesaria* como elemento de primera clase.

Conceptos Básicos (Primitivos)

Del marco descrito se extraen una serie de nociones significativas, en función de las cuales se define todo el modelo. Son las de *puerto*, *instancia* de componente, tipo de componente (*arquetipo*) y *conexión* o ligadura. Aunque no todas son estrictamente primitivas –en particular, las conexiones no lo son–, todas ellas deben poder ser manipuladas como tales, por lo que es preciso asignarles un identificador, es decir, proporcionarles un *nombre*.

Por este motivo, se particiona el conjunto universal V en una serie de subconjuntos, en los que se agrupan los *nombres* que identifican (unívocamente) a cada una de las entidades pertenecientes a cada una de las categorías semánticas. Se definen, por tanto:

V_P – Conjunto de nombres de *puerto* (punto de interacción).
V_I – Conjunto de nombres de *instancia de componente*.
V_T – Conjunto de nombres de *arquetipo* (tipo de componente).
V_B – Conjunto de nombres de *conexión* (interacción entre dos instancias).

A éstos se debe añadir un conjunto V_E de nombres especiales, que no identifican a ninguna entidad del modelo, y que –por el momento– carecerán de significado. A estos nombres inertes los denominaremos *etiquetas* y, como se verá más adelante, su función será la de actuar como selectores; por tanto, nunca tendrán una semántica propia, sino que ésta vendrá dada solamente por su uso –o sus usos, ya que podrán ser múltiples–.

Todos estos subconjuntos son totalmente disjuntos. Sin embargo, no constituyen necesariamente una partición completa de V, ya que siempre podrán existir categorías de nombres, aún no incorporadas, que eventualmente podrían resultar significativas para nuestro modelo (o sus variantes). En tal caso, estos nombres "ajenos" deberán formar parte igualmente de V, ya que éste se define como universal. Al subconjunto que los agrupa le denominaremos *conjunto residual*, y lo denotaremos por $\mho_S$. En realidad, tan sólo está expresando la posibilidad de que el modelo pueda ser ampliado –como de hecho se hará en la sección 5.5.2–, por lo que a todos los efectos se podrá considerar equivalente al conjunto vacío ($\mho_S \approxeq \varnothing$).

De este modo, se tiene una *partición* de V, tal que:

$$\begin{gathered} V = V_P \cup V_I \cup V_T \cup V_B \cup V_E \cup \mho_S \\ \bigcap_{\varphi}(V_\varphi) = \varnothing \ / \ V_\varphi \in \{V_P, V_I, V_T, V_B, V_E, \mho_S\} \end{gathered} \tag{5.1}$$

Y por tanto, para cada nombre n se cumple:

$$n \text{ nombre} \iff \begin{cases} n \text{ nombre puerto} & \Rightarrow n \in V_P \subset V \\ n \text{ nombre instancia} & \Rightarrow n \in V_I \subset V \\ n \text{ nombre arquetipo} & \Rightarrow n \in V_T \subset V \\ n \text{ nombre conexión} & \Rightarrow n \in V_B \subset V \\ n \text{ etiqueta} & \Rightarrow n \in V_E \subset V \\ \text{otro} & \Rightarrow n \in \mho_S \subset V \end{cases}$$

Dadas estas nociones como primitivas, todos los demás conceptos del modelo se podrán construir como derivados, a partir de su definición intuitiva; en el proceso se va elaborando una estructura compleja que constituye, de hecho, la *arquitectura* de cada sistema descrito. A continuación se define cada uno de estos conceptos.

Dimensión Vertical: Composición

Se describirá en primer lugar la noción de *instancia*. Ésta es, a un tiempo, la más sencilla y la más compleja del modelo. Aunque se la ha identificado como primitiva, esto no excluye que de hecho manifieste una estructura, especialmente en su relación con el resto de conceptos. Su complejidad nace del hecho de que, al expandirse, da lugar a la *jerarquía de composición* de una arquitectura, que ha sido asignada como el eje *vertical* de la misma.

Se identifican, pues, dos clases de instancias: *primitivas* y *compuestas*; las primeras se consideran como un bloque atómico e indivisible, mientras que las segundas se corresponden con una *configuración*. No obstante, externamente las instancias de ambas clases son idénticas. Todas ellas están encapsuladas y se relacionan con su contexto mediante una *interfaz*. Ésta está segmentada, esto es, se divide en una serie de puntos de interacción o *puertos*. Por tanto, toda instancia se *corresponde* con una interfaz, que se define como un conjunto de puertos.

La descripción de la interfaz cubre completamente el caso de los componentes primitivos. En lo que respecta a los compuestos, es necesario considerar además que esta interfaz encapsula a lo que en realidad es una configuración. Una *configuración* se define como una amalgama –literalmente, una *composición*– de instancias, que interactúan entre sí a través de conexiones. Estas conexiones se establecen, bien entre las instancias internas a la configuración –en realidad, entre los puertos de éstas–, o bien exportan servicios hacia el exterior a través de la interfaz. En este último caso se denominan *conexiones jerárquicas*; serán detalladas en la sección siguiente, al tratar la interacción –ver definición 5.20–.

Por tanto, una instancia *compuesta* contiene solamente tres tipos de elementos: puertos, instancias y conexiones. Éstos se están considerando aquí como atómicos, aunque más tarde podrán ser adecuadamente desarrollados. Por su parte, una instancia *primitiva* sólo contiene puertos, lo que en realidad puede verse como un caso particular de la definición de una compuesta; y de hecho, así es como será considerada a partir de este momento. Conviene plantear ya este razonamiento en términos del modelo. Sea entonces:

Definición 5.1 (Configuración) *Se define una* configuración C_i *como un conjunto heterogéneo de nombres de tres tipos: los de los* puertos *de su interfaz, los de las* instancias *que contiene, y los de las* conexiones *que relacionan éstas entre sí y con la interfaz.*

$$n \in C_i \iff \begin{cases} n \in V_P \Rightarrow n \text{ nombre puerto} \\ n \in V_I \Rightarrow n \text{ nombre (sub-)instancia} \quad \forall C_i \in KF \\ n \in V_B \Rightarrow n \text{ nombre conexión} \end{cases} \tag{5.2}$$

Se define asimismo el conjunto KF como la agrupación de todas estas configuraciones –que constituyen tanto instancias simples como compuestas–. Es fácil deducir, por tanto:

$$C_i \subset (V_P \cup V_I \cup V_B),\ \forall C_i \in KF \implies KF \subset \wp(V_P \cup V_I \cup V_B) \tag{5.3}$$

Además, todas las configuraciones son disjuntas: ningún puerto, instancia o conexión puede formar parte simultáneamente de dos configuraciones. Debe tenerse en cuenta en todo momento que los nombres de V *identifican* a los elementos unívocamente, y que **en estas definiciones no se contemplan (aún) los tipos**. Por tanto, dos configuraciones del mismo tipo son completamente distintas, y no guardan entre sí relación alguna. En este punto, *cada puerto, instancia y conexión es único* y totalmente diferenciable del resto.

$$C_i \cap C_j = \varnothing \quad \forall C_i, C_j \in KF \tag{5.4}$$

Definición 5.2 (Frontera) *Dada una configuración C_i, se define el* operador frontera $\mathcal{F}$ *como un operador que proporciona su* interfaz*, seleccionando los* puertos *de entre los elementos que la componen. Se representa como una relación binaria que proyecta una configuración sobre el conjunto de sus puertos ($\mathcal{F} : (KF \leftrightarrow \wp(V_P))$). Se denominará* conjunto frontera*, o simplemente* frontera*, al obtenido como resultado de su aplicación.*

Como ocurre en general con las relaciones binarias deterministas –sólo hay una salida para cada entrada–, resulta más cómodo usarla como una función (total[3]) de la forma:

$$\begin{array}{rcl} \mathcal{F} & : & KF \longrightarrow \wp(V_P) \\ \mathcal{F}(C_i) & = & C_i \cap V_P \qquad \forall C_i \in KF \end{array} \tag{5.5}$$

Obviamente, cuando se da que $C_i = \mathcal{F}(C_i)$, es decir, cuando la instancia coincide con su interfaz, estamos ante un componente *primitivo*. En todos los demás casos se tratará de un componente *compuesto*. Por tanto, y como se había dicho, los primeros se consideran así como como un caso particular de los segundos. Se definen por conveniencia los conjuntos KP y KC de componentes primitivos y compuestos, respectivamente, del modo evidente.

$$\left.\begin{array}{rcl} KP & \doteq & \{C_j \ / \ \mathcal{F}(C_j) = C_j \ \wedge \ C_j \in KF\} \\ KC & \doteq & \{C_j \ / \ \mathcal{F}(C_j) \subsetneq C_j \ \wedge \ C_j \in KF\} \end{array}\right\} \Longrightarrow KF = KP \cup KC \tag{5.6}$$

En este punto, se ha de poder establecer ya la relación entre el nombre de una instancia y la configuración que realiza su desarrollo estructural. En general:

Definición 5.3 (Relación de Nombrado) *Se define* relación de nombrado *como el tipo de relación que se establece entre el* nombre *(atómico) de una entidad de tipo φ y una estructura (no atómica) que implementa a dicha entidad. Se representa siempre como una relación binaria $\mathcal{N}_\varphi$ entre un (sub-)conjunto de nombres y el conjunto de tales entidades.*

Existen tantas relaciones de nombrado como tipos de entidades primitivas define el modelo; la única excepción son los *puertos*, que se consideran totalmente atómicos. Eventualmente se describirán todas ellas; la más importante es, sin duda, el caso que nos ocupa ahora:

Definición 5.4 (Nombrado de Instancias) *Se define el* nombrado de instancias *como una relación binaria $\mathcal{N}_I \subset (V_I \times KF)$ establecida entre cada nombre de instancia ($i \in V_I$) y la configuración ($C_i \in KF$) a la que hace referencia. Sea pues:*

[3] Además sería inyectiva, si no fuese por el caso especial de los componentes sin interfaz, es decir, los *sistemas cerrados*. En ellos, la frontera resulta ser la interfaz vacía ($\varnothing$), que ya se incluye por definición en $\wp(V_P)$.

$$\mathcal{N}_I \triangleq \{(i, C_i) \ / \ i \in V_I \ \wedge \ C_i \in KF\} \tag{5.7}$$

De nuevo, puede utilizarse más cómodamente como una función. Es decir:

$$\mathcal{N}_I : \begin{array}{ccc} V_I & \longrightarrow & KF \\ i & \longrightarrow & C_i \end{array} \quad \text{esto es, } \mathcal{N}_I(i) = C_i \in KF \tag{5.8}$$

En este caso, dado que cada configuración tendrá exclusivamente un nombre, y al menos uno, la función será biyectiva total[4].

Es sencillo ahora, a partir solamente del nombre de una instancia, determinar si es primitiva ($\mathcal{N}_I(i) \in KP$) o compuesta ($\mathcal{N}_I(i) \in KC$). Igualmente es fácil obtener su interfaz, por simple composición de las dos funciones (relaciones) definidas: $(\mathcal{F} \circ \mathcal{N}_I(i)) \equiv (\mathcal{F}(\mathcal{N}_I(i)))$. Dado esto, para una instancia primitiva k, es también evidente que $\mathcal{F}(\mathcal{N}_I(k)) = \mathcal{N}_I(k)$.

No obstante, la relación $\mathcal{N}_I$ proporciona por sí sola, además, una información mucho más fundamental, ya que en realidad está describiendo la *jerarquía* de componentes: la dimensión de composición de la arquitectura del sistema. Para mostrarlo con mayor claridad, se utilizará la siguiente relación auxiliar:

Definición 5.5 (Reunión de Instancias) *Se define la* reunión de instancias *como una relación binaria de la forma $\mathcal{J}_I \subset (V_I \times \wp(V_I))$, que indica, dada una instancia, el conjunto de instancias que ésta reune en su interior. Se deduce de $\mathcal{N}_I$ de la manera obvia.*

De nuevo, resulta más sencillo manejarla como una función (total[5]):

$$\mathcal{J}_I : \begin{array}{ccc} V_I & \rightarrow & \wp(V_I) \\ i & \rightarrow & \mathcal{N}_I(i) \cap V_I \end{array} \quad \text{de modo que } \mathcal{J}_I(i) = \mathcal{N}_I(i) \cap V_I, \ \forall i \in V_I \tag{5.9}$$

Es decir: dada una instancia i, se obtiene la configuración $\mathcal{N}_I(i)$ correspondiente, y de ésta se extraen, por intersección, los nombres de las instancias *interiores* a la primera.

Se debe recordar, además, que para dos instancias compuestas i, j cualesquiera, las respectivas reuniones $\mathcal{J}_I(i), \mathcal{J}_I(j)$ son totalmente disjuntas, tal como se deduce de la ecuación 5.4.

$$\mathcal{J}_I(i) \cap \mathcal{J}_I(j) = \varnothing \quad \forall i, j \in V_I \ \wedge \ i \neq j \ \wedge \ \mathcal{N}_I(i), \mathcal{N}_I(j) \in KC \tag{5.10}$$

Cuando la instancia es compuesta (esto es, $\mathcal{N}_I(i) \in KC$), todo $\mathcal{J}_I(i)$ contendrá al menos un elemento. Obsérvese que la relación, en realidad, está también definida para las instancias primitivas; sin embargo, es evidente que, en tal caso, éstas no contienen a ninguna otra.

$$\begin{array}{ll} \exists j \in V_I \ / \ j \in \mathcal{J}_I(i) \subset V_I & \forall i \ / \ i \in V_I \ \wedge \ \mathcal{N}_I(i) \in KC \\ \mathcal{J}_I(i) = \varnothing & \forall i \ / \ i \in V_I \ \wedge \ \mathcal{N}_I(i) \in KP \end{array}$$

Esto permitirá que su comportamiento, en lo que resta, sea exactamente el deseado. Basándose en $\mathcal{J}_I$, es posible definir, por fin:

[4] V_I es potencialmente infinito, pero sólo se consideran los nombres de instancia efectivamente utilizados.

[5] Véase de nuevo la nota 4, en la página 119. Si $\mathcal{J}_I(i)$ se aplicase sólo a las instancias compuestas, sería además inyectiva, ya que como se deduce de la ecuación 5.4, una instancia sólo puede estar contenida en otra.

Definición 5.6 (Contención directa) *Se define la relación de* contención directa *entre dos instancias* $(i, j \in V_I)$ *dadas como una relación binaria* homogénea $\ni \subset (V_I \times V_I)$*, de modo que se tiene que* $(i, j) \in \ni$ *si el primer nombre* (i) *designa a la configuración que contiene a la instancia referenciada con el segundo nombre* (j)*.*

Esta relación *no* puede ser utilizada como una función, puesto que no es determinista[6]; en lugar de ello, se admitirá para ella la notación infija $i \ni j$. Su definición formal resulta notablemente sencilla en términos de la relación auxiliar $\mathcal{J}_I$, de modo que:

$$\ni \;\stackrel{\circ}{=}\; \{(i, j) \;/\; j \in \mathcal{J}_I(i)\} \;=\; \{(i, j) \;/\; j \in (\mathcal{N}_I(i) \cap V_I)\} \tag{5.11}$$

Obsérvese que $\ni$ sólo proporciona, en realidad, información que ya estaba presente en la relación heterogénea $\mathcal{N}_I$, y de hecho tan sólo considera una parte de la misma. Sin embargo, permite caracterizar con mayor facilidad algunas propiedades de aquélla; al centrarse exclusivamente en los nombres de instancia, expresa de manera inmediata la jerarquía de composición ya mencionada. Por otra parte, su homogeneidad simplificará enormemente la expansión de ésta.

Puede comprobarse intuitivamente que la relación no sólo será *irreflexiva* y *asimétrica*, sino que incluso será *intransitiva*. En efecto, es (i) *irreflexiva*, ya que ninguna instancia puede contenerse a sí misma, y (ii) *asimétrica*, puesto que si una instancia contiene a otra, ésta no puede, a su vez, contener a la primera. Formalmente, ambas propiedades se indican como:

$$\begin{array}{llll} (i) & \forall i \in V_I, & (i, i) \notin \ni & \equiv \quad i \not\ni i \\ (ii) & \forall i, j \in V_I, & (i, j) \in \ni \Rightarrow (j, i) \notin \ni & \equiv \quad i \ni j \Rightarrow j \not\ni i \end{array} \tag{5.12}$$

donde $\not\ni$ es obviamente la relación complementaria, esto es, $\not\ni = \overline{\ni} = (V_I^2 - \ni)$.

Pero, además, la relación no se limita a *no* ser transitiva, sino que, debido a que expresa una relación directa, es de hecho explícitamente (iii) *intransitiva*. Es claro que si una instancia dada contiene a otra $(i \ni j)$, y ésta a su vez a una tercera $(j \ni k)$, puede deducirse que esta última (k) está ubicada en el interior de la primera (i), de manera indirecta. Sin embargo, esta idea no es reflejada por la relación, ya que no es así como fue concebida. Pero por el contrario, hay una intuición distinta que *sí* debe expresar. Se sabe que la segunda instancia contiene a la tercera, y que una instancia no puede pertenecer al mismo tiempo a dos configuraciones (recuérdese la ecuación 5.10), luego se deduce obviamente que la primera *no* contiene a la tercera. Es decir, se obtiene la *negación* de la relación. Formalmente:

$$(iii) \quad \forall i, j, k \in V_I, \quad \{(i, j), (j, k)\} \in \ni \Rightarrow (i, k) \notin \ni \quad \equiv \quad i \ni j \wedge j \ni k \Rightarrow i \not\ni k \tag{5.13}$$

Sin embargo, es evidente que la información indirecta es también interesante, y de hecho resulta fundamental, ya que está directamente vinculada a la jerarquía. Por eso, representarla con naturalidad resulta especialmente importante. Afortunadamente, gracias a la homogeneidad de la relación $\ni$, resulta fácil realizar una expansión del concepto:

Definición 5.7 (Contención General) *Se define la* contención general *o* indirecta *como una relación binaria homogénea* $\Ni$ *entre instancias, que viene dada por el* cierre transitivo *de* $\ni$*, esto es, por la menor relación binaria transitiva sobre* V_I *que contiene a* $\ni$*.*

[6] Para un valor dado en el dominio, la relación puede proporcionar más de un valor en el codominio.

$$\begin{array}{lcl} \Supset & : & V_I \leftrightarrow V_I \\ \Supset & \doteq & \supset^+ \quad = \bigcup_{i=1}^{n}(\supset^i) \quad \text{para } n \leq \binom{|V_I|}{2} \end{array} \tag{5.14}$$

Al igual que en el caso anterior, se admitirá para esta relación la notación infija equivalente, de modo que $i \Supset j \equiv (i,j) \in \Supset$. Como es sabido, el cierre transitivo no afecta a la reflexividad ni a la simetría; por tanto, la nueva relación es *irreflexiva*, *asimétrica* y, obviamente, *transitiva*. Son las mismas propiedades de la inclusión estricta de la teoría de conjuntos ($A \subset B$), y de su inversa ($B \supset A$). Esto es lógico, dado que la idea de fondo es análoga a la de esta última, aunque no idéntica –aparte de la indirección dada por $\mathcal{N}_I$, la contención $\supset$ introduce restricciones de cardinalidad en niveles adyacentes que no se dan entre subconjuntos estándar–.

Como todo cierre transitivo, la relación $\Supset$ también puede ser definida por construcción, a partir de la relación $\supset$. De la propia definición se deduce que la primera "etapa" de la transitividad se obtiene por composición de la relación consigo misma; esto es, $(\supset \circ \supset)$ o, en forma de potencia, $\supset^2$. Para no perder el punto de partida, esta etapa ha de unirse a la relación original, de modo que se obtiene $(\supset \cup \supset^2)$. Este proceso se repite iterativamente, hasta que la relación resultante no incorpore ninguna nueva pareja de instancias. El cierre es, por tanto, la unión de todas las potencias de $\supset$ hasta el grado n necesario[7], tal como ya se indicaba en la ecuación 5.14.

Tanto de lo dicho como de la definición se infiere que la relación expresa efectivamente la idea deseada: se cumple que $i \Supset k$ si i contiene a k, directa o indirectamente. Es decir, se da una secuencia de contenciones, tal que i contiene alguna instancia l_1, que a su vez contiene a otra l_2, y así sucesivamente hasta llegar a k. Esta secuencia puede ser tan corta o larga como sea preciso, y se corresponde con lo que en teoría de relaciones se denomina un *camino*, por analogía con la teoría de grafos. Este punto será puesto de relieve un poco más abajo, en la definición 5.8.

Resulta significativo representar esta idea semiformalmente, de modo que:

$$i \Supset k \;\simeq\; i \supset l_1 \supset \ldots \supset l_n \supset k, \text{ tal que } n \in \mathbb{N}^* \;\wedge\; \{l_j\}_{j=1}^{n} \subset V_I$$

Obsérvese que incluso esta expresión aproximada se ha definido de forma que n puede ser cero ($\mathbb{N}^*$). En se caso, no existiría ninguna instancia l_j intermedia, por lo que la expresión se reduciría simplemente a $i \supset k$. Esto ya se deducía de la definición, y significa tan sólo que la contención directa es un caso particular de la general ($\supset \subset \Supset$). También es sencillo mostrar sobre esta expresión que la relación es irreflexiva, es decir, que sólo es posible cuando $i \neq k$. Puede verse que el caso menor se reduce a $\supset$, que ya era irreflexiva, y que cualquier otra secuencia (camino) será mayor. Si fuese posible un par reflexivo, el camino sería un *ciclo*, lo que es imposible debido a la asimetría de la relación original.

Las relaciones inversas (o recíprocas) para ambos tipos de contención también son particularmente interesantes, ya que estas últimas establecen una estructura que aquéllas permiten recorrer. Se definen de la manera obvia como:

$$\begin{array}{lclcl} \in & \doteq & \supset^{-1} & = & \{(j,i) \;/\; i \supset j\} \quad = \quad \{(j,i) \;/\; j \in \mathcal{J}_I(i)\} \subset V_I^2 \\ \Subset & \doteq & \Supset^{-1} & = & \{(j,i) \;/\; i \Supset j\} \quad = \quad \{(j,i) \;/\; j(\in)^n i\} \subset V_I^2 \quad 1 \leq n \leq \binom{|V_I|}{2} \end{array} \tag{5.15}$$

Ambas tienen características interesantes, en la mayoría de los casos análogas a las de las relaciones directas. En primer lugar, es evidente que éstas, al igual que aquéllas, se pueden

[7] Con un límite, dado por el número de todas las posibles combinaciones de instancias, tomadas dos a dos.

definir en función de $\mathcal{N}_I$ (o de su versión reducida, $\mathcal{J}_I$). En segundo lugar, tienen las mismas propiedades que sus recíprocas: las dos son irreflexivas y asimétricas, mientras que la primera ($\Subset$) es intransitiva, y la segunda ($\Supset$) es transitiva.

Es particularmente importante señalar que, dado que las reuniones de instancias son siempre disjuntas (ecuación 5.10), resulta posible deducir que la relación $\Subset$ es determinista, esto es, puede representarse como una función:

$$\left.\begin{array}{rccc} \Subset: & V_I & \rightarrow & V_I \\ & i & \rightarrow & j \end{array}\right\} \Rightarrow \left.\begin{array}{ccc} \Subset(i) = j & \Longleftrightarrow & j \Supset i \\ \equiv & & \\ i \Subset j & \Longleftrightarrow & j \Supset i \end{array}\right\} \wedge \ |\Subset(i)| = 1 \tag{5.16}$$

Que es, simplemente, una consecuencia natural del hecho de que cada instancia puede estar contenida exclusivamente en otra, o en términos gráficos, que cada instancia *sólo puede tener un padre*. Esta nomenclatura deriva del hecho de que una relación como $\Subset$ define obviamente una estructura arbórea; aunque este enfoque no será desarrollado, se utilizará ocasionalmente la notación asociada, por ser lo suficientemente clara y conocida.

En el mismo grupo de nociones "gráficas" se incluye el concepto siguiente:

Definición 5.8 (Camino) *Dada la relación de contención $\Supset$, sea una secuencia de instancias $c_k \in V_I$, ordenada según un índice $k \in \mathbb{N}^*$, y elegida de modo que la instancia $(k+1)$-ésima está contenida en la k-ésima ($c_k \Supset c_{k+1}$). Se define entonces el* camino de longitud n, *desde la* instancia origen c_0 *hasta la* instancia terminal c_n, *como una relación binaria $\mathcal{C}_j \subset \Supset$ formada por los n primeros pares ordenados de la forma (c_k, c_{k+1}) extraídos de esa secuencia.*

$$\mathcal{C}_j = \{(c_k, c_{k+1}) \ / \ (0 \leq k < n) \ \wedge \ (c_k \Supset c_{k+1})\} \tag{5.17}$$

Por supuesto, éste es un concepto de teoría de grafos; sin embargo, no es necesario desarrollar la analogía, ya que puede ser concebido, como acaba de verse, utilizando únicamente términos de teoría de relaciones. Esta última definición se ha inspirado parcialmente en el modo en que Whitmire [Whi97] adapta el concepto a un lenguaje puramente relacional.

Conocida la relación de contención general ($\Supset\!\!\!\!=$), es sencillo determinar el conjunto de instancias contenidas, directa o indirectamente y a cualquier nivel, en una dada. Ocasionalmente será útil poder disponer de este conjunto, que recibe el nombre de *expansión de $\mathcal{N}_I$*, o, utilizando de nuevo el símil gráfico, *subrama* de un componente i. Por tanto, para obtenerlo con facilidad se define la función siguiente:

Definición 5.9 (Función de Expansión) *Se define la* función de expansión, *respecto de la relación de nombrado de instancias $\mathcal{N}_I$, como aquella que, dada una instancia $i \in V_I$, proporciona el conjunto de todas las instancias contenidas en ella, en el sentido general. Es decir:*

$$\begin{array}{rcl} \Xi\left[\mathcal{N}_I\right] & : & V_I \longrightarrow \wp(V_I) \\ \Xi\left[\mathcal{N}_I\right](i) & \stackrel{\circ}{=} & \{j \in V_I \ / \ i \Supset\!\!\!\!= j\} \end{array} \tag{5.18}$$

A partir de lo visto hasta el momento, es sencillo deducir toda una serie de propiedades de la dimensión compositiva en $\aleph_\Sigma$. No se desarrolla aquí, ya que la mayoría de ellas se hacen evidentes incluso con una simple argumentación verbal; el mérito del modelo es que permite

expresar dichas argumentaciones y propiedades de un modo preciso y conciso. Así, intuiciones tales como "una instancia no puede contenerse a sí misma, ni siquiera indirectamente" se pueden expresar, demostrar y manipular fácilmente bajo formas tales como

$$i \notin \Xi\left[\mathcal{N}_I\right](i)\,\forall i \in V_I$$

aunque en este caso concreto resulta más sencillo decir que $i \not\ni i$, o simplemente recordar que la relación $\Supset$ es –como ya se sabía– antirreflexiva.

Para terminar, se consideraran brevemente las versiones simétricas de las relaciones de contención, que serán utilizadas en los apartados siguientes. Su utilidad deriva del hecho de que, en ocasiones, no interesa tanto el sentido de la contención, como simplemente el hecho de que esta relación existe. Sea entonces:

Definición 5.10 (Contención Simétrica) *Dada la relación de contención directa $\ni$, se define la* contención simétrica *como una relación binaria homogénea simétrica $\ominus \in (V_I \leftrightarrow V_I)$, obtenida como el cierre simétrico de la relación $\ni$.*

$$\begin{array}{lcl} \ominus & : & V_I \leftrightarrow V_I \\ \ominus & \stackrel{\circ}{=} & \ni^S \quad = \ni \cup \in \end{array} \tag{5.19}$$

Es decir, entre dos instancias $i, j \in V_I$ existe una contención simétrica si alguna de ellas contiene a la otra, es decir, $i \ominus j \Longleftrightarrow (i \ni j) \vee (i \in j)$. Determina, en definitiva, si dos instancias están en niveles adyacentes de la jerarquía de composición, sin considerar el sentido de la relación.

De manera equivalente, puede definirse también una versión simétrica de la relación de contención general, de modo que se tiene:

Definición 5.11 (Contención General Simétrica) *Sea la relación de contención general $\Supset$, se define la* contención general simétrica *como una relación binaria homogénea reflexiva, simétrica y transitiva $\circledcirc \in V_I \leftrightarrow V_I$, obtenida como el cierre simétrico de la relación $\Supset$. Es decir:*

$$\begin{array}{lcl} \circledcirc & : & V_I \leftrightarrow V_I \\ \circledcirc & \stackrel{\circ}{=} & \Supset^S \quad = \Supset \cup \Subset \end{array} \tag{5.20}$$

Debe recordarse que, al considerar el cierre simétrico de un cierre transitivo, se obtiene el cierre reflexivo transitivo de la relación original, de modo que puede concluirse que $\circledcirc = \ni^*$. Del mismo modo, se podría haber obtenido la contención general simétrica como el cierre transitivo de la contención simétrica, esto es, $\circledcirc = \ominus^+$. Es fácil ver que, de este modo, se llega igualmente al cierre reflexivo transitivo de la relación original, y por tanto se demuestra la equivalencia.

En realidad, esta última relación es muy amplia, pero proporciona información de utilidad. Expresa, en primer lugar, si una instancia determinada pertenece o no a *alguna* jerarquía de composición. Más todavía, permite separar dicha jerarquía del resto de la arquitectura: en lenguaje de teoría de grafos, extrae las *componentes conexas* de la dimensión de composición. En una arquitectura convencional, por supuesto, todas las instancias forman parte de la misma jerarquía, por lo que la relación $\circledcirc$ abarca a todo par de instancias, de modo que resulta equivalente al producto cartesiano ($\circledcirc = V_I \times V_I$).

Aunque se podrían extraer muchas más consecuencias a partir de las definiciones de este apartado, especialmente si se utilizase el lenguaje de la teoría de grafos –donde evidentemente

la jerarquía de contención adopta la forma de un árbol–, se considera que lo tratado hasta el momento resulta suficiente. Por tanto, finaliza aquí el recorrido por la dimensión de composición, y pasa a considerarse la interacción, como segundo eje estructural de una arquitectura.

Dimensión Horizontal: Interacción

En un modelo arquitectónico como el que estamos adoptando, toda interacción entre dos instancias se realiza a través de una *conexión* (asociación o enlace) explícita. Precisamente, esta es una de las características que definen a la Arquitectura de Software. En $\aleph_\Sigma$, todas estas conexiones tienen además un *nombre*, es decir, todas ellas están identificadas de manera unívoca. No obstante, en muchos casos –de hecho, en la mayoría de los modelos– habrá un buen número de conexiones que se manipulen como si fueran *anónimas*. Sin embargo, esto no significa que carezcan de nombre –de otro modo, nunca podrían ser referenciadas–, sino que simplemente éste no se hace explícito desde el exterior. En $\aleph_\Sigma$, esto será simplemente irrelevante.

Como ya se ha dicho, el modelo agrupa los nombres de conexión en el subconjunto primitivo $V_B \subset V$. A diferencia de lo que ocurría con las instancias, todas las conexiones tienen la misma naturaleza: su única función es expresar grupos de interacción, por lo que no desarrollan una estructura interna. Se pueden considerar, en cierto modo, análogas a un componente primitivo; de hecho, esta comparación será utilizada a menudo en los apartados siguientes.

La única diferencia significativa entre las conexiones será tan sólo la cardinalidad de su interfaz. Las conexiones en Marmol serán n-arias, mientras que en los modelos habituales se limitan a ser binarias; por tanto, resulta interesante hacer explícito este dato. Sin embargo, deberá tenerse en cuenta que esto no supondrá diferencia alguna en su tratamiento.

Por tanto, la noción básica de interacción se define como:

Definición 5.12 (Conexión) *Se define una* conexión $L_i^{\prec g}$ *como un conjunto homogéneo de nombres de puerto, que expresa la existencia de algún tipo de interacción entre todos ellos. Se dice, entonces, que estos puertos están* ligados *o* conectados. *El superíndice* (g) *expresa la cardinalidad de este conjunto, que se denomina* grado de la conexión*, y podrá ser cualquier número natural finito mayor o igual que dos.*

$$L_i^{\prec g} \subseteq V_P \ / \ |L_i^{\prec g}| = g \ \wedge \ g \geq 2 \tag{5.21}$$

Una conexión puede indicarse como $L_i^{\prec g}$ o simplemente como L_i, omitiendo en este último caso la referencia al grado de la conexión, cuya finalidad es puramente informativa. Por su parte, el subíndice (i) tan sólo se utiliza para distinguir entre sí las diferentes conexiones.

Se define asimismo, para cada grado de conexión h, el conjunto de todas las conexiones con dicho grado, y se denota como $\mathcal{A}^{\prec h}$. Todos estos *conjuntos de grado* se reunen en un único *conjunto de conexiones* global, que se denota como $\mathcal{A}^*$. Por tanto:

$$\begin{array}{c} \left.\begin{array}{l} L_i^{\prec g} \subseteq V_P \\ L_i^{\prec g} \in \mathcal{A}^{\prec g} \end{array}\right\} \Longrightarrow \mathcal{A}^{\prec g} \subset \wp(V_P) \\ \mathcal{A}^* \doteq \bigcup_{i=2}^{n} (\mathcal{A}^{\prec i}) \subset \wp(V_P) \end{array} \tag{5.22}$$

Obviamente, los distintos conjuntos $\mathcal{A}^{\prec g}$ definen una partición del conjunto global $\mathcal{A}^*$, de modo que cada uno de ellos agrupa a todos los subconjuntos del mismo tamaño.

El hecho de considerar conexiones n-arias es una generalización propia de MARMOL, que no resulta habitual en el modelo semántico de los ADLs existentes. Lo cierto es que la mayoría de ellos emplean únicamente las conexiones del primer subconjunto de $\mathcal{A}^*$, esto es, las *conexiones binarias*. Incluso en un modelo más amplio, como éste, sigue resultando más natural utilizar éstas, con preferencia a conexiones de cualquier otro grado. Dada su frecuencia, pues, se proporciona el siguiente caso particular de la definición de conexión:

Definición 5.13 (Enlace) *Se define* enlace *como una conexión binaria $L_i^{\prec 2}$, que supone implícitamente una interacción bidireccional entre los dos puertos que contiene. No hay ninguna diferencia estructural entre un* enlace *y cualquier otra* conexión binaria*, por lo que ambos términos se utilizarán, en adelante, como sinónimos.*

Por tanto, el conjunto de *enlaces* binarios $\mathcal{A}^{\prec 2}$ será particularmente importante. Sin embargo, no será preciso hacer para éste definiciones especiales; por el contrario, podrá ser tratado exactamente del mismo modo que cualquier otro. La noción básica de interacción, en MARMOL, es la de conexión n-aria: ésta no se plantea como una extensión del concepto de enlace; por el contrario, es éste el que se define como un caso particular de la primera.

No obstante, una vez que se ha identificado la idea de enlace, merece la pena señalar que *sí* existe en Arquitectura de Software otro concepto relacionado, que había sido explícitamente excluido de $\aleph_\Sigma$: es, por supuesto, el de *conector*. Dado que MARMOL define un modelo puramente estructural, en principio no hay nada que impida establecer una analogía entre una conexión n-aria y un conector. De hecho, dependiendo de la interpretación que se utilice, el papel de un conector podrá ser representado, alternativamente, por una instancia de componente (primitivo) o por una conexión. El modelo admite ambos enfoques, sin optar por ninguno de ellos. En definitiva, este aspecto deberá ser definido para cada lenguaje concreto; en la sección 6.11 se describe el modo en que se ha planteado, dentro de la sintaxis específica de $\mathcal{P}i\mathcal{L}ar$.

En cualquier caso, una vez conocida la estructura de una conexión, puede establecerse la relación de nombrado correspondiente, de modo que se define:

Definición 5.14 (Nombrado de Conexiones) *Se define el* nombrado de conexiones *como una relación binaria $\mathcal{N}_B \subset (V_B \times \mathcal{A}^*)$, establecida entra cada nombre de conexión $(b \in V_B)$ y la conexión $(L_i^{\prec g} \in \mathcal{A}^{\prec g})$ a la que hace referencia. Sea pues:*

$$\mathcal{N}_B : \begin{array}{ccc} V_B & \rightarrow & \mathcal{A}^* \\ b & \rightarrow & L_i^{\prec g} \end{array} \quad \text{es decir, } \mathcal{N}_B(b) = L_i^{\prec g} \in \mathcal{A}^{\prec g} \subseteq \mathcal{A}^* \tag{5.23}$$

Al igual que en casos anteriores, y como puede verse, la relación de nombrado puede utilizarse como una función total biyectiva[8], que por supuesto resulta completamente independiente de la noción de grado. No obstante, es muy sencillo utilizar esta relación para identificar *nombres* de conexión n-aria; en particular, para el caso de un enlace:

$$b \in V_B \ \text{ es enlace } \iff \mathcal{N}_B(b) \in \mathcal{A}^{\prec 2}$$

La relación de nombrado (de conexiones) resulta fundamental para poder desarrollar el esquema de instancias completo, a partir de la noción de configuración –definición 5.1–. De allí se

[8] Aplíquese aquí un comentario análogo al de la nota 4, pero para el caso de V_B.

deduce que una instancia puede contener a puertos, otras instancias o incluso conexiones. Éste último detalle no había sido aún desarrollado, ya que no se había dado un significado para esa noción; pero puede serlo, sin embargo, a partir de ahora.

Puede verse que las conexiones son análogas a una instancia primitiva, y por tanto no contienen a ningún otro elemento, ni establecen una jerarquía; pero, en cambio, ellas mismas están contenidas en instancias compuestas, a través de las configuraciones, por lo que forman parte del esquema de contención descrito por éstas. Esto puede expresarse con un concepto, análogo al de reunión de instancias ($\mathcal{J}_I$), que se define como:

Definición 5.15 (Reunión de Conexiones) *Se define la* reunión de conexiones *como una relación binaria de la forma $\mathcal{J}_B \subset (V_I \times \wp(V_B)$ que indica, dada una instancia, el conjunto de conexiones que ésta reune en su interior. Es análoga a la* reunión de instancias $\mathcal{J}_I$*, y se deduce de $\mathcal{N}_I$ por intersección, de la manera obvia.*

De nuevo, y al igual que ocurría con la reunión de instancias, resulta más sencillo manejarla como una función (total). En este caso, la suprayectividad no está asegurada, ni siquiera aunque se limitase el dominio de la función a las instancias compuestas. Por tanto:

$$\mathcal{J}_B : \left.\begin{array}{rcl} V_I & \rightarrow & \wp(V_B) \\ i & \rightarrow & \mathcal{N}_I(i) \cap V_B \end{array}\right\} \Longrightarrow \mathcal{J}_B(i) = \mathcal{N}_I(i) \cap V_B,\ \forall i \in V_I \tag{5.24}$$

A partir de aquí, podría establecerse una relación de *contención de conexiones* análoga a la de instancias ($\ni$); sin embargo, dado que una conexión no define a su vez una sub-estructura, no se podría desarrollar una versión transitiva de dicha contención, por lo que en realidad no resulta necesaria. Esto no es extraño, ya que lo que se está explorando es la interacción como dimensión *horizontal*, y por tanto no sería lógico identificar en el proceso una jerarquía de contención, que definiría de nuevo un esquema vertical.

Por otra parte, aunque una *conexión* se ha definido simplemente como una agrupación de puertos, en realidad está describiendo una relación mucho más compleja, esto es, una *interacción entre instancias*. La idea de fondo es que todo nombre de puerto estará contenido en (la frontera de) una instancia compuesta, y por tanto "pertenece" a ella. Así pues, cuando una conexión utiliza un puerto determinado, en realidad *también* está haciendo referencia a la instancia que lo contiene. En definitiva, se establece de este modo una relación entre dos[9] o más instancias, que se reflejará a continuación en el concepto de *ligadura* y sus derivados.

Para asegurar la consistencia de esta aproximación, es preciso eliminar la posibilidad de utilizar un puerto aislado, que no pertenezca a ninguna instancia, lo cual se expresará mediante la condición siguiente:

$$\forall L_i \in \mathcal{A}^* : \forall p \in L_i : \exists C_j \in KF \ / \ p \in C_j \ \vee \ p = \varepsilon \tag{5.25}$$

Que puede, desde ahora, considerarse *incorporada* a la definición de conector. Se desea, por generalidad, permitir la posibilidad de que una conexión no utilice alguno de sus puertos, dejándolo libre. Para ello se introduce el símbolo ε, que tan sólo expresará que existe algún

[9] En realidad, nada impide que un enlace utilice dos puertos de la misma instancia, por lo que la interacción podría incluso involucrar a *una* sola.

puerto no ligado. Esta idea es análoga a la del puerto especial **void** de $\mathcal{P}i\mathcal{L}ar$. Este aspecto no será explícitamente considerado en el resto de este Capítulo, ya que se tratará de una situación poco habitual.

Teniendo esto en cuenta, puede considerarse ya una relación que expresa la existencia de una interacción entre dos instancias, y que se define como:

Definición 5.16 (Ligadura de Instancias) *Se define la* ligadura de instancias *como una relación binaria homogénea* simétrica $\hookrightarrow \subseteq (V_I \times V_I)$ *sobre el conjunto de instancias, que expresa la existencia de conexiones entre pares de ellas. Dos instancias $i, j \in V_I$ forman un par de la relación $\hookrightarrow$ si existe* al menos una *conexión $L_k^{\prec g} \in \mathcal{A}^*$ que contenga* al menos un *puerto de cada una de ellas –esto es, de sus interfaces–. Es decir:*

$$(i,j) \in \hookrightarrow \iff \begin{cases} \exists L_k^{\prec g} \in \mathcal{A}^* : \\ \quad (\exists p \in L_k^{\prec g} \,/\, p \in (\mathcal{F} \circ \mathcal{N}_I(i))), \\ \quad (\exists q \in L_k^{\prec g} \,/\, q \in (\mathcal{F} \circ \mathcal{N}_I(j))) : \\ \qquad\qquad p \neq q \end{cases} \tag{5.26}$$

Donde obviamente $i, j \in V_I$, y $p, q \in V_P$. De manera equivalente, se puede expresar:

$$(i,j) \in \hookrightarrow \iff \begin{cases} \exists L_k^{\prec g} \in \mathcal{A}^* : \\ \qquad (L_k^{\prec g} \cap \mathcal{F}(\mathcal{N}_I(i)) \neq \varnothing) \\ \qquad\qquad \wedge \\ \qquad (L_k^{\prec g} \cap \mathcal{F}(\mathcal{N}_I(j)) \neq \varnothing) \\ \qquad\qquad \wedge \\ \quad \left| \, (L_k^{\prec g} \cap \mathcal{F}(\mathcal{N}_I(i))) \cap \mathcal{F}(\mathcal{N}_I(j))) \, \right| > 1 \end{cases} \tag{5.27}$$

Obsérvese que estas definiciones tan sólo especifican que dos instancias cualesquiera están en la relación si existe al menos una conexión que contiene puertos de ambas instancias. Además, se pone especial cuidado en asegurar que se utilizan al menos *dos* puertos diferentes; de otro modo, toda instancia estaría "ligada" consigo misma, cayendo en el caso trivial. Sin embargo, téngase en cuenta que esto no impide que una instancia se conecte consigo misma, utilizando dos puertos diferentes; pero en este caso se estará haciendo de manera consciente.

Como en ocasiones anteriores, se admite para esta relación la notación infija equivalente, de modo que $i \hookrightarrow j \equiv (i,j) \in \hookrightarrow$. Se dirá, pues, que dos instancias de componente están *ligadas* o *conectadas* cuando existe una relación de ligadura entre ellas, es decir, cuando $i \hookrightarrow j$.

De la definición se sigue inmediatamente que la relación de ligadura es *irreflexiva*, ya que la reflexividad se evita explícitamente; y *simétrica*, dado que es obvio que si i tiene una conexión en común con j, entonces j tiene esa misma conexión en común con i. Por tanto, $(i \hookrightarrow j) = (j \hookrightarrow i)$. Evidentemente, la relación no es transitiva, ya que sólo expresa interacciones *directas*. Sin embargo, es sencillo desarrollar una versión transitiva de la misma, que exprese la existencia de una relación *indirecta* entre dos instancias:

Definición 5.17 (Encadenamiento) *Se define el* encadenamiento *o* ligadura débil *como una relación binaria homogénea simétrica $\looparrowright \subset (V_I \times V_I)$ entre instancias de componente, definida como el cierre transitivo de la relación de ligadura $\hookrightarrow$. Es decir:*

$$\begin{array}{lcl} \looparrowright & : & V_I \leftrightarrow V_I \\ \looparrowright & \stackrel{\circ}{=} & (\hookrightarrow^+) \quad = (\hookrightarrow^*) \end{array} \tag{5.28}$$

Es decir, dos instancias $(i, j) \in V_I$ estarán *encadenadas* si están directamente ligadas entre sí, o existe otra instancia –o incluso una secuencia de instancias encadenadas– que esté simultáneamente ligada con ambas. Obsérvese que, dado que la relación de ligadura es simétrica, su cierre transitivo es también reflexivo, por lo que la relación $\looparrowright$ es reflexiva, simétrica y transitiva.

Utilizando esta relación, se puede determinar la influencia que tiene una instancia en todo el esquema, según su interacción. Idealmente, esta influencia abarcará a la arquitectura completa, a través de las relaciones de interacción –téngase en cuenta que existirán conexiones jerárquicas, que serán definidas más abajo–; no obstante, esto no es obligatorio, y se permite la existencia de instancias aisladas. Será útil, en cualquier caso, poder determinar el ámbito de influencia *directa* de una instancia, que constituye su entorno. Sea pues:

Definición 5.18 (Contexto de una Instancia) *Sea una instancia de componente* $\underline{i} \in V_I$. *Se define su* contexto *–es decir, en las que participa– en el es*

$$))\} \tag{5.29}$$

Su deducción, a partir de la definición de conexión, es inmediata. Se trata, tan sólo, de obtener el conjunto de conexiones que utilicen alguno de los puertos de la frontera de la instancia actual. Describe, en definitiva, la conectividad de la misma.

Resulta también interesante la definición del concepto dual, centrado en una conexión:

Definición 5.19 (Contexto de una Conexión) *Sea una conexión* $L_i^{\prec c} \in \mathcal{A}^*$, *referenciada por el nombre* $b \in V_B$. *Se define su* contexto *como el conjunto de todas las instancias ligadas por ella en el esquema, y se representa como* $\divideontimes(b) \subset V_I$.

$$\begin{array}{lcl} \divideontimes & : & V_B \to \wp(V_I) \\ \divideontimes(b) & \stackrel{\circ}{=} & \{i \in V_I \mid \exists p \in (\mathcal{F} \circ \mathcal{N}_I(i)) \,/\, p \in \mathcal{N}_B(b)\} \end{array} \tag{5.30}$$

Cuya deducción es, tan igualmente obvia como la anterior. Obsérvese que se puede dar una nueva definición de ligadura (directa) a partir de este concepto, de modo que:

$$i \hookrightarrow j \iff \exists b \in V_B \,/\, i \in \divideontimes(b) \wedge j \in \divideontimes(b) \tag{5.31}$$

Existe una correlación implícita entre la jerarquía de composición y el contexto de interacción. Esta se hará explícita mediante una regla de ámbito –Ley 5.1–; no obstante, conviene antes especificar el siguiente concepto relacionado:

Definición 5.20 (Conexión Jerárquica) *Sea una conexión binaria* $L_i^{\prec 2} = (l_i, m_i) \in \mathcal{A}^{\prec 2}$, *referenciada por un nombre* $b \in V_B$. *Se dice que es una* conexión jerárquica *cuando los puertos que la componen pertenecen, respectivamente, a una instancia cualquiera y a* la *instancia que contiene* directamente *a ésta, y por tanto a sus interfaces. Es decir:*

$$\begin{array}{l} \text{Sea } b \in V_B \,/\, \mathcal{N}_B(b) = L_i^{\prec 2} = \{p, q\} \in \mathcal{A}^{\prec 2} : \\ \text{b conexión jerárquica} \iff \left\{ \begin{array}{l} \exists j \in V_I \,/\, p \in \mathcal{F}(\mathcal{N}_I(j)), \\ \exists k \in V_I \,/\, q \in \mathcal{F}(\mathcal{N}_I(k)) : \\ \qquad j \ominus k \end{array} \right. \end{array} \tag{5.32}$$

Como es bien sabido, el objetivo de estas conexiones es únicamente exportar el ámbito de interacción de un puerto hasta el nivel superior de la jerarquía de composición. Por tanto, esta noción involucra únicamente a dos puertos, y no tiene sentido generalizarlo al caso de una conexión n-aria; se trata siempre, por tanto, de enlaces.

Se define, asimismo, el conjunto de conexiones jerárquicas $\mathcal{AJ}$, como un subconjunto del conjunto global de enlaces; se tiene, por tanto, que $\mathcal{AJ} \subseteq \mathcal{A}^{\prec 2} \subseteq \mathcal{A}^*$.

El caso de una conexión jerárquica es especial, ya que supone sólo una extrusión de ámbito. Por esto, a menudo se desea no tenerlas en cuenta cuando se habla del contexto de interacción de una instancia. En este caso, interesa conocer únicamente el conjunto de instancias *de su propio nivel* con las que interactúa, excluyendo explícitamente a su contenedor, o a sus propios contenidos. Se define, entonces:

Definición 5.21 (Ligadura en Nivel) *Sean dos instancias $i, j \in V_I$. Se dice que están* ligadas *o* conectadas en nivel*, y se expresa como $i \widehat{\hookrightarrow} j$, si están ligadas y ninguna de ellas está contenida en la otra. Es decir:*

$$\begin{array}{c} i \widehat{\hookrightarrow} j \iff ((i \hookrightarrow j) \wedge (\neg(i \ominus j))) \\ \equiv \\ \widehat{\hookrightarrow} \stackrel{\circ}{=} (\hookrightarrow - \ominus) \end{array} \tag{5.33}$$

Y a su vez, se expande esta definición con su cierre transitivo:

Definición 5.22 (Encadenamiento en Nivel) *Se define el* encadenamiento en nivel *como una relación binaria homogénea reflexiva, simétrica y transitiva $\widehat{\looparrowright} \in (V_I \times V_I)$, obtenida como el cierre transitivo de la relación de ligadura de nivel $\widehat{\hookrightarrow}$. Es decir:*

$$\widehat{\looparrowright} = (\widehat{\hookrightarrow}^+) = (\widehat{\hookrightarrow}^*) \tag{5.34}$$

Obsérvese que esta relación no puede obtenerse a su vez como la diferencia entre el encadenamiento $\looparrowright$ y la contención simétrica $\ominus$, ya que la expansión del primero puede alcanzar a todos los antecesores y descendientes de una instancia, así como a todas las instancias ligadas con ellos; en cambio, el encadenamiento en nivel $\widehat{\looparrowright}$ se expandirá únicamente en el nivel actual de la jerarquía de composición.

Las dos definiciones que acaban de hacerse sólo tienen sentido si se considera que existe la correlación mencionada entre las dimensiones de composición e interacción. Esto se expresa mediante la siguiente regla general, que se aplica a todo el esquema:

Ley 5.1 (Regla de Ámbito) *En un esquema arquitectónico, sólo pueden existir conexiones entre instancias del mismo nivel, o entre instancias de niveles adyacentes en la jerarquía de composición. Es decir:*

$$\forall i, j \in V_I : \ (i \hookrightarrow j) \implies ((i \in\ni j) \vee (i \ominus j)) \tag{5.35}$$

En definitiva, dos instancias sólo podrán estar ligadas si están contenidas en la misma ("son hijos del mismo padre"), o si una de ellas es, precisamente, la que contiene a la otra ("el padre"). En ausencia de una norma de este tipo, las instancias podrían conectarse, de manera arbitraria, con elementos de otros niveles, rompiendo por completo la idea de *encapsulación* que constituye la esencia de la noción de componente.

Teniendo en cuenta esta restricción, se puede explorar por fin la mencionada correlación entre las dos dimensiones, comenzando por la definición siguiente:

Definición 5.23 (Vínculo) *Se define el* vínculo *como una relación binaria homogénea simétrica* $\rightrightarrows \subset (V_I \times V_I)$ *entre instancias de componente, que expresa que entre sus antecesores en la jerarquía de contención existe* al menos *una ligadura.*

Se dice en tal caso que estas instancias están vinculadas.

$$i \rightrightarrows j \iff \exists k_i, k_j \in KF \ : \ (k_i \Supset i) \wedge (k_j \Supset j) \wedge (k_i \hookrightarrow k_j) \tag{5.36}$$

Obsérvese que esta relación permite utilizar las conexiones jerárquicas, ya que en la definición se emplea la ligadura convencional. Esto produce una mezcla compleja entre las dos dimensiones: en el caso más normal, toda instancia estará *vinculada* con la mayoría de sus hermanos, antecesores, e incluso consigo misma; para esto bastaría con que existiese una conexión jerárquica entre dos antecesores suyos cualesquiera.

Si se excluye el uso de las conexiones jerárquicas, se obtiene un subconjunto cuyo significado intuitivo es totalmente diferente:

Definición 5.24 (Vínculo Puro) *Se define el* vínculo puro *como una relación binaria homogénea simétrica* $\models\!\!\rightrightarrows \subset (V_I \times V_I)$ *entre instancias de componente, que expresa que entre sus antecesores en la jerarquía de contención existe* al menos *una ligadura en nivel.*

$$i \models\!\!\rightrightarrows j \iff \exists k_i, k_j \in KF \ : \ (k_i \Supset i) \wedge (k_j \Supset j) \wedge (k_i \widehat{\hookrightarrow} k_j) \tag{5.37}$$

Es decir, ahora dos instancias están *puramente vinculadas* solamente si existe una conexión (no jerárquica) entre dos de sus antecesores. Esto permite, por ejemplo, expresar la relación que existe entre dos sub-instancias contenidas en otras dos instancias, que a su vez interactúan entre sí. Esto no sólo se da, además, en niveles adyacentes, sino en todos los de la jerarquía. Esta noción de "interacción indirecta" es muy útil en Arquitectura de Software; en los grafos-tubo de Holt [HM94, Hol98] se define una relación similar, desde una perspectiva gráfica.

Ahora bien, no resulta lógico limitar esta noción a las conexiones directas entre dos componentes, sino que deberá expandirse para abarcar a toda interacción:

Definición 5.25 (Vínculo Débil) *Se define el* vínculo débil *como una relación binaria homogénea simétrica* $\models\!\!\Rrightarrow \subset (V_I \times V_I)$ *entre instancias de componente, que expresa que entre sus antecesores en la jerarquía de contención existe* al menos *un encadenamiento en nivel.*

$$i \models\!\!\Rrightarrow j \iff \exists k_i, k_j \in KF \ : \ (k_i \Supset i) \wedge (k_j \Supset j) \wedge (k_i \widehat{\hookrightarrow} k_j) \tag{5.38}$$

Obsérvese que este vínculo excluye, de nuevo, a las conexiones jerárquicas. Sin embargo, se expande completamente en cada nivel de una jerarquía de composición, hasta el punto que, con interacción suficiente, abarca a la práctica totalidad de los miembros de dicha jerarquía, excluyendo solamente al primero (la "raíz").

Téngase en cuenta que, en el caso más habitual, en el que se tendrá al menos una conexión jerárquica en cada configuración, esta relación y la de *encadenamiento* ($\looparrowright$) son prácticamente coincidentes, aunque no idénticas. Aun así, se define:

Definición 5.26 (Vínculo General) *Se define el* vínculo general *como una relación binaria homogénea simétrica* $\Rrightarrow \subset (V_I \times V_I)$ *entre instancias de componente, que expresa que entre sus antecesores en la jerarquía de contención existe* al menos *un encadenamiento.*

$$i \Rrightarrow j \iff \exists k_i, k_j \in KF \,:\, (k_i \Supset i) \wedge (k_j \Supset j) \wedge (k_i \looparrowright k_j) \tag{5.39}$$

Éste abarca, por tanto, a toda conexión jerárquica, y alcanza por tanto a todo elemento vinculado *de cualquier modo* con uno dado. En cualquier caso, normalmente será más interesante considerar la diferencia entre vínculo débil y encadenamiento, y no su unión.

En resumen, las dimensiones de composición e interacción están fuertemente relacionadas, pero su conjunción es más compleja y expresiva que su simple yuxtaposición. Su combinación se define según unas reglas específicas, que ninguna de ellas puede mantener por sí sola. Es precisamente cuando estas estructuras entran en conflicto, cuando se demuestra la importancia de disponer de un esquema arquitectónico apto para el análisis.

En su reciente propuesta sobre *bigrafos* [Mil01][10], que es en muchos sentidos análoga a esta parte de nuestro modelo, Robin Milner ha decidido, sin embargo, eliminar cualquier tipo de restricción en la relación entre anidamiento (composición) y conectividad (interacción). Obviamente esto permite el modelado de un rango más amplio de sistemas, pero resulta poco práctico, en especial cuando el objetivo principal es la descripción de la estructura resultante. El propio Milner había introducido este tipo de restricciones en sus cálculos de acción [Mil96], y reconoce que eliminarlas es *"contrario a la realidad"*, pero manifiesta que su motivo es que pretende *"modelar no sólo la realidad (...), sino también la ficción"* [Mil01].

Aunque es posible concebir el potencial de su enfoque –sobre todo en relación con el concepto de código móvil–, parece razonable no asumirlo en nuestro modelo, máxime cuando aún no ha sido explorado y es contrario a la práctica existente en otros modelos arquitectónicos. Por otra parte, si eventualmente se considerase interesante, adaptar nuestro modelo a esta idea tan sólo requeriría ignorar la Ley 5.1 –Regla de Ámbito– y las definiciones subsiguientes.

Sistema de Tipos

En los dos apartados anteriores se han recorrido completamente las dos dimensiones que definen la estructura de una arquitectura de software convencional, e incluso se ha considerado su combinación. Sin embargo, esto se ha hecho de manera única para cada puerto, componente y conexión; en realidad, no se está describiendo una arquitectura, sino una *instancia de arquitectura* concreta. En su forma actual, no habría forma de comparar los esquemas correspondientes a

[10] En teoría de grafos, este término es sinónimo de *grafo bipartito*; aquí se usa en un sentido diferente.

dos arquitecturas intuitivamente idénticas, salvo mediante el establecimiento de un isomorfismo entre ellos, esto es, una biyección entre sus nombres que preserve la estructura.

Sin embargo, precisamente el objetivo de la Arquitectura de Software es la identificación de *patrones* estructurales, por lo que este enfoque no parece adecuado. En los ADLs de tipo algebraico, estos patrones se expresan, mediante la noción de *tipo de componente* o *arquetipo*. Precisamente, el mayor inconveniente de buena parte de los ADLs de tipo gráfico, que constituyen el otro enfoque habitual, es la ausencia de un tratamiento explícito de este concepto.

En este apartado, se establecerá por tanto un *sistema de tipos*, desarrollado de manera análoga a la estructura definida para las instancias, pero con dos diferencias fundamentales: en primer lugar, ya no se puede garantizar que cada elemento se utilice una única vez, por lo que no se puede afirmar que un nombre de tipo ($t \in V_T$) *identifique* a un elemento. Debido a esto, todo tipo irá asociado a una *etiqueta* ($e \in V_E$), de modo que se podrán distinguir dos usos diferentes del mismo, sin necesidad de utilizar una estructura de multiconjunto. Tanto los tipos como sus etiquetas podrán ser reutilizados de manera indefinida, siempre que sea en contextos diferentes; por otra parte, los respectivos conjuntos V_T y V_E eran ya conocidos, pues fueron descritos como subconjuntos de V al principio de la descripción del modelo $\aleph_\Sigma$.

En segundo lugar, cada vez que se utiliza un tipo determinado, se repite con él toda su subestructura. Esto significa que, a diferencia de lo que ocurría con las instancias, no será preciso desarrollar una estructura completa, sino que bastará con la combinación de una serie de definiciones de tipo, que se desplegará por sí sola.

El objetivo de este sistema de tipos es comprobar si la arquitectura definida por las instancias y conexiones respeta los patrones estructurales declarados por sus arquetipos. Se ha de comenzar, ante todo, definiendo la relación que, en última instancia, establecerá la correspondencia entre las instancias y sus tipos. El resto de este apartado se dedicará, simplemente, a describir la estructura del sistema de tipos, para finalmente plantear su comparación con el esquema definido por las instancias. Cuando esta comparación sea consistente con la relación de tipado, podrá afirmarse que el sistema está *bien tipado*.

Por tanto, se empieza definiendo la relación de tipado:

Definición 5.27 (Tipado) *Se define el* tipado *o* esquema de tipos *como una relación binaria hetereogénea* $\mathcal{T} \subset ((V_I \cup V_B) \times V_T)$*, que se establece entre todo nombre de instancia o conexión y un determinado nombre de tipo. Sea pues:*

$$\begin{array}{rccc} \mathcal{T}: & (V_I \cup V_B) & \longrightarrow & V_T \\ & c & \longrightarrow & t \end{array} \tag{5.40}$$

Al igual que en ocasiones anteriores, puede utilizarse esta relación como una función, en este caso total suprayectiva[11]. Se introduce además la notación abreviada siguiente, que recibirá la denominación de **asignación de tipo**, y se define como:

$$(c :: t) \iff \mathcal{T}(c) = t \iff (c, t) \in \mathcal{T}$$

[11] Es total porque toda instancia o conexión deberá estar tipada. Se considera suprayectiva en un sentido similar al de la nota 4 de la página 119: aunque V_T es potencialmente infinito, sólo se consideran los tipos que hayan sido efectivamente instanciados, asumiendo que esto ocurre al menos una vez para todo tipo declarado.

Ha de observarse que el tipado de los *puertos* no ha sido tratado. Esto se debe a que el conjunto V_T sólo contiene, como se dijo en su definición, nombres de *arquetipos*. En realidad, en cualquier modelo coherente de Arquitectura de Software se debe considerar que los puertos están tipados. Más aún, en la mayoría de los casos lo estarán de manera estática. Sin embargo, estos tipos, sean simples o compuestos, se relacionan ante todo con la naturaleza de los *datos*, y no tienen nada que ver con aspectos estructurales, o con los tipos de componente. En particular, no se concibe la posibilidad de *transmitir* una instancia a través de un puerto. Por tanto, al ser tipos totalmente diferentes, acumularlos en V_T carecería de sentido. Obviamente, podría considerarse un nuevo conjunto de nombres $V_D \subset V$, que permitiera referirse a los *tipos de puerto*, un concepto que también debería de tener en cuenta sus posibles combinaciones.

Sin embargo, éste es uno de los detalles que se han decidido abstraer en el modelo, ya que no se vincula de manera directa a la descripción arquitectónica. En $\aleph_\Sigma$, por tanto, los puertos se consideran atómicos y no tipados; esta suposición es compatible con cualquier sistema de tipos concebible, incluso cuando éste es dinámico. En cualquier caso, este aspecto concreto habrá de ser definido, o abstraído también, por cada ADL particular.

Dicho esto, merece la pena considerar la división de la relación de tipado en dos sub-relaciones que tienen en cuenta la naturaleza de las entidades –o nombres– implicadas:

Definición 5.28 (Tipado de Componentes) *Se define el* tipado de componentes *como una relación binaria, hetereogénea y asimétrica* $\mathcal{T}_I \subset (V_I \times V_T)$*, establecida entre todo nombre de instancia y el nombre del tipo que se le asigna. Por tanto:*

$$\begin{array}{rccc} \mathcal{T}_I : & V_I & \longrightarrow & V_T \\ & i & \longrightarrow & t \end{array} \tag{5.41}$$

Y, análogamente, se define también:

Definición 5.29 (Tipado de Conexiones) *Se define el* tipado de conexiones *como una relación binaria, hetereogénea y asimétrica* $\mathcal{T}_B \subset (V_B \times V_{TB})$*, establecida entre cada nombre de conexión y el nombre del tipo que se le asigna, de modo que:*

$$\begin{array}{rccc} \mathcal{T}_B : & V_B & \longrightarrow & V_T \\ & b & \longrightarrow & t \end{array} \tag{5.42}$$

Del mismo modo, y con un objetivo similar, se puede establecer una distinción entre los *tipos de componente* y los *tipos de conexión*, agrupados respectivamente en dos conjuntos V_{TI} y V_{TB}, de modo que definen una partición del V_T original. Resulta todavía más útil, como podrá verse más adelante, considerar una subdivisión del primero, de modo que se separan los tipos de componente *primitivo* (V_{TIP}) de los tipos de componente *compuesto* (V_{TIC}).

La partición queda, por tanto, definida finalmente como:

$$\begin{gathered} V_T = V_{TB} \cup V_{TI} = V_{TB} \cup (V_{TIP} \cup V_{TIC}) \\ \bigcap_\varphi (V_\varphi) = \varnothing \ / \ V_\varphi \in \{V_{TB}, V_{TIP}, V_{TIC}\} \end{gathered} \tag{5.43}$$

No obstante, en el contexto general de MARMOL debe tenerse en cuenta que la distinción entre tipos de componente y de conexión resulta un tanto artificiosa, a diferencia de la existente

entre instancias y conexiones. Aunque no resulta relevante en este momento, una vez que se introduzcan los esquemas de reificación (en el modelo $\aleph_{\mathrm{M}}$), ambos conceptos podrán ser intercambiados con libertad casi total, a partir del primer meta-nivel (§5.5.2).

Precisamente por esto, el criterio que se utiliza para establecer la separación entre los subconjuntos de V_T es exactamente el contrario. Dado entonces un tipo t, se dirá si es de componente o de conexión según la instancia a la que sea asignado:

$$\begin{array}{rcl} V_{TI} & \stackrel{\circ}{=} & \{t \in V_T \,/\, \exists i \in V_I : \mathcal{T}_I(i) = t\} \\ V_{TB} & \stackrel{\circ}{=} & \{t \in V_T \,/\, \exists b \in V_B : \mathcal{T}_B(b) = t\} \end{array} \tag{5.44}$$

Del mismo modo, para las dos clases de tipos de componente:

$$\begin{array}{rcl} V_{TIP} & \stackrel{\circ}{=} & \{t \in V_{TI} \,/\, \exists i \in V_I : (\mathcal{T}_I(i) = t) \ \wedge \ (\mathcal{N}_I(i) \in KP)\} \\ V_{TIC} & \stackrel{\circ}{=} & \{t \in V_{TI} \,/\, \exists i \in V_I : (\mathcal{T}_I(i) = t) \ \wedge \ (\mathcal{N}_I(i) \in KC)\} \end{array} \tag{5.45}$$

Recapitulando todo lo dicho, puede comprobarse que se cumple globalmente que:

$$\left.\begin{array}{rcrcl} \mathcal{T} & : & (V_I \cup V_B) & \rightarrow & V_T \\ \mathcal{T}_I & : & V_I & \rightarrow & V_{TI} \\ \mathcal{T}_B & : & V_B & \rightarrow & V_{TB} \end{array}\right\} \wedge \quad \mathcal{T} = \mathcal{T}_I \cup \mathcal{T}_B \tag{5.46}$$

Por lo que a partir de una asignación de tipo, se puede deducir de manera trivial la categoría semántica de los nombres implicados.

$$(c :: t) \implies \left\{\begin{array}{lcl} t \in V_{TI} & \wedge & c \in V_I \\ t \in V_{TB} & \wedge & c \in V_B \end{array}\right. \tag{5.47}$$

Sin embargo, un arquetipo ($\in V_T$) no describe más que un nombre, y en cambio todo tipo tiene asociada una estructura, similar a lo que habitualmente se denomina un tipo abstracto. A esta estructura, que desarrolla el contenido de un tipo, se le denomina *declaración de tipo*, y tiene una forma análoga a la de una configuración de instancias, combinada además con el desarrollo de las ligaduras entre las conexiones de su interior. En definitiva:

Definición 5.30 (Declaración de Tipo) *Se define una* declaración de tipo DT_j *como la terna formada por una* declaración de interfaz DF_j*, una* declaración de configuración DC_j*, y un* esquema de conexión E_j*. Se denomina* $\mathcal{D}_{\mathcal{T}}$ *al conjunto de estas declaraciones de tipo.*

$$DT_j \in \mathcal{D}_{\mathcal{T}} \implies DT_j = (DF_j, DC_j, E_j) \tag{5.48}$$

Es preciso definir a su vez los tres constituyentes mencionados, de modo que:

Definición 5.31 (Declaración de Interfaz) *Se define una* declaración de interfaz DF_j *como un conjunto de etiquetas* $e \in V_E$*, de modo que* $DF_j \in \wp(V_E)$*.*

Este conjunto carece *a priori* de significado; posteriormente, será relacionado con un tipo de componente ($t \in V_T$). Desde ese momento, las etiquetas representarán a cada uno de los *puertos* que constituyen la interfaz de este arquetipo. La única restricción que se aplica deriva de la

propia definición de conjunto, y consiste en que las etiquetas de una interfaz no pueden estar repetidas. Esto significa que todo *puerto de un tipo* podrá ser identificado de manera unívoca.

Sin embargo, esto debe distinguirse con claridad de los nombres de puerto ($p \in V_P$) que constituían la interfaz o frontera de una configuración, tal como se estableció en la definición 5.2. Allí, los elementos de una interfaz son explícitamente designados como *nombres*, mientras que aquí son declarados como *etiquetas*. La diferencia es que los primeros *identifican* a un puerto de manera unívoca, de modo que puede ser únicamente utilizado en una ocasión; mientras que las segundas tan sólo señalan el *lugar* donde, eventualmente, habrá de ubicarse un puerto determinado. La unicidad de las etiquetas sólo se asegura dentro de cada declaración, de modo que la misma podría ser utilizada en otra, sin que esto implique ningún tipo de relación.

Definición 5.32 (Declaración de Configuración) *Se define una* declaración de configuración DC_j *como un conjunto de pares ordenados, formados por una etiqueta y un nombre de tipo, y seleccionados de modo que una etiqueta no puede aparecer en más de un par. Es decir:*

$$\begin{gathered} DC_j \triangleq \{(e,t) \,/\, e \in V_E \ \wedge \ t \in V_T\} \iff DC_j \subset (V_E \times V_T) \\ \wedge \\ (e,t) \in DC_j \implies \nexists u \in V_T : (e,u) \in DC_j \end{gathered} \tag{5.49}$$

Obviamente, cada uno de estos conjuntos puede verse como una relación binaria heterogénea, que asigna a cada etiqueta e un tipo t determinado. Más aún, teniendo en cuenta la condición establecida, es claro que esta correspondencia es unívoca, por lo que resulta sencillo considerarlo como una función suprayectiva, de modo que:

$$\left.\begin{array}{rccc} DC_j : & V_E & \rightarrow & V_T \\ & e & \rightarrow & t \end{array}\right\} \implies DC_j(e) = t \tag{5.50}$$

Este conjunto, por tanto, proporciona una enumeración de los distintos elementos que forman una configuración, y designa a cada uno de ellos mediante una etiqueta única, por razones análogas a las ya indicadas en el caso de los puertos. Por definición, pues, a cada una de estas etiquetas se le asigna un *arquetipo*, y sólo uno.

Obsérvese, no obstante, que las condiciones desarrolladas sólo garantizan la unicidad de las etiquetas *dentro de cada declaración* de configuración. No hay nada que impida utilizar una misma etiqueta en distintas declaraciones; de hecho, esto se hará normalmente con cierta frecuencia. Sin embargo, debe de tenerse en cuenta que esto *no* supone la existencia de relación alguna entre los elementos designados con la misma etiqueta.

En esta definición solamente se ha hecho referencia al conjunto de nombres de tipo V_T. Por tanto, los tipos utilizados pueden ser tanto de componente como de conexión. Esto significa, en resumen, que puede redefinirse una **declaración de configuración** como la secuencia etiquetada de los tipos de instancias (sub-instancias) y conexiones (sub-conexiones) que *componen la estructura* de una configuración, es decir, un componente compuesto.

Sin embargo, a menudo será interesante tener en cuenta la distinción entre componentes y conexiones, dentro de una configuración. Con este fin, se definen los siguientes subconjuntos para *cada* configuración DC_j, en la que además definen una partición:

$$\left.\begin{array}{rcl} DI_j & \triangleq & \{(e,t) \in DC_j \,/\, t \in V_{TI}\} \\ DB_j & \triangleq & \{(e,t) \in DC_j \,/\, t \in V_{TB}\} \end{array}\right\} \iff (DC_j = DI_j \cup DB_j) \ \wedge \ (DI_j \cap DB_j = \varnothing) \tag{5.51}$$

Donde, evidentemente, DI_j reune la secuencia de tipos de las sub-instancias de una configuración, y DB_j hace lo propio con las conexiones de la misma. Con el objetivo de establecer una distinción similar entre las etiquetas, se define ahora:

Definición 5.33 (Proyección de Etiquetas) *Dada una declaración de configuración DC_j, o cualquiera de sus subconjuntos, se define la* proyección de etiquetas *como un operador π_E que devuelve todas las etiquetas de esa declaración, es decir, que crea un conjunto tomando (proyectando) el primer miembro de cada uno de los pares ordenados de la misma.*

$$\begin{gathered} \pi_E : \wp(V_E \to V_T) \to \wp(V_E) \\ \pi_E(DC_j) \triangleq \{e \in V_E \ / \ (e,t) \in DC_j\} \end{gathered} \tag{5.52}$$

Gracias a este operador, resulta sencillo considerar el conjunto de las etiquetas de una (declaración de) configuración, así como el conjunto de las etiquetas correspondientes a sus instancias (sub-instancias) o conexiones (sub-conexiones). Por brevedad, se admitirá en adelante la nomenclatura siguiente para las etiquetas de cada declaración de tipo:

$$\text{Sea } e \in V_E \Longrightarrow \left\{ \begin{array}{lcl} e \in DF_j & \Rightarrow & e \text{ etiqueta de puerto} \\ e \in \pi_E(DI_j) & \Rightarrow & e \text{ etiqueta de instancia} \\ e \in \pi_E(DB_j) & \Rightarrow & e \text{ etiqueta de conexión} \end{array} \right\} \text{ para cada } DT_j \in \mathcal{D}_\mathcal{T}$$

Además de la proyección de etiquetas, se admitirá el uso del operador convencional de *proyección* de una relación, π_k, que extrae el k-ésimo elemento de cada una de las n-tuplas ordenadas que la componen. Esto incluye, como caso inusual, a los elementos de las ternas que constituyen una declaración de tipo, de modo que:

$$\text{Sea } DT_j \in \mathcal{D}_\mathcal{T} \Longrightarrow \left\{ \begin{array}{lcl} \pi_1(DT_j) & = & DF_j \\ \pi_2(DT_j) & = & DC_j = (DI_j \cup DB_j) \\ \pi_3(DT_j) & = & E_j \end{array} \right.$$

Con esto, estamos por fin en condiciones de definir el último de los elementos con los que se elabora una declaración de tipo, y que se compone exclusivamente de etiquetas:

Definición 5.34 (Esquema de Conexión) *Se define un* esquema de conexión E_j *como una relación binaria hetereogénea, establecida entre un conjunto de etiquetas y un conjunto de* secuencias *de pares ordenados de etiquetas. Más concretamente, se trata de una relación que hace corresponder a cada* etiqueta de conexión *una secuencia de pares ordenados, compuestos por una* etiqueta de instancia *y una* etiqueta de puerto. *Es decir:*

$$\begin{aligned} &\text{Sea } DT_j = (DF_j, (DI_j \cup DB_j), E_j) \in \mathcal{D}_\mathcal{T}, \text{ entonces:} \\ &E_j \triangleq \Big\{(e_b, S) \ / \ \ e_b \in \pi_E(DB_j) \ \wedge \ \Big(S = \{s_k\} \ \Big\} \\ &\qquad (\forall k \in [1..n] \ \wedge \ n \geq 2\,) \ \wedge \\ &\qquad (\, s_k = (e_i, e_p) \ / \ \ e_i \in (\pi_E(DI_j) \cup \Upsilon) \ \wedge \\ &\qquad\qquad (\exists DT_m \in \mathcal{D}_\mathcal{T} : e_p \in DF_m)) \Big) \end{aligned} \tag{5.53}$$

En esta ecuación, el símbolo Υ es una etiqueta especial ($\Upsilon \in V_E$) que representa a la declaración de tipo actual, esto es, a la DC_j en la que está contenida la propia E_j definida. Sin embargo, no se trata en este caso de ningún tipo de autorreferencia problemática: su único objetivo será permitir la definición de enlaces jerárquicos, como se describe más abajo.

Lo que hace este esquema, en definitiva, es relacionar entre sí a los distintos elementos que constituyen una declaración de tipo, es decir, los puertos de la declaración de interfaz y los tipos de componente y conexión de la declaración de configuración. En este sentido, se mimetiza en parte la estructura de apartados anteriores, pero aplicada ahora a tipos y etiquetas.

La estructura que aquí se define pretende expresar la conectividad entre las sub-instancias del tipo que se está definiendo; para esto, se consideran únicamente las *conexiones* que éste contiene. Cada una de ellas es referenciada en el tipo mediante una etiqueta; a ésta última se le relaciona, entonces, con una secuencia de pares ordenados S, cuya longitud es indeterminada. La idea es que cada uno de los pares $s_k \in S$ representa a un puerto, de modo que la secuencia completa expresará el conjunto de puertos ligados por la conexión con la que se empareja. Es por esto que la longitud es como mínimo igual a dos, ya que una conexión ha de ser al menos binaria.

Finalmente, cada par s_k está constituido por dos términos. El primero ha de ser una de las *etiquetas de instancia* del tipo actual, o bien el símbolo especial Υ. Esto último permite hacer referencia a los puertos del tipo contenedor, permitiendo por tanto definir conexiones jerárquicas. Del segundo término sólo se dice que ha de ser una *etiqueta de puerto* en alguna declaración de tipo. Obsérvese que, en este punto, no es todavía posible determinar si el puerto es, efectivamente, uno de los contenidos en la instancia, ya que aún no se dispone de una relación de *nombrado* que proporcione la correspondencia entre el nombre de un tipo y su estructura. Esta relación será introducida de inmediato; sin embargo, se difiere el desarrollo de este último aspecto hasta la definición 5.39, donde se garantizará la *consistencia* de un esquema de conexión.

Con esto, queda (casi) completamente especificada la estructura de una declaración de tipo. Debe recordarse, no obstante, que se había establecido una distinción entre tipo de componente *primitivo* y tipo de componente *compuesto*. Igualmente, y del mismo modo que ocurría con las instancias –véase la ecuación 5.6–, podrá determinarse una distinción entre una declaración de tipo primitivo, que solamente constará de interfaz, y una de tipo compuesto, para la que se describirá la estructura completa. Sea pues:

$$\left.\begin{array}{lcl} \mathcal{D}_{\mathcal{TP}} & \mathrel{\hat{=}} & \{DT_k \in \mathcal{D}_{\mathcal{T}} \,/\, DT_k = (DF_k, \varnothing, \varnothing)\} \\ \mathcal{D}_{\mathcal{TC}} & \mathrel{\hat{=}} & \{DT_k \in \mathcal{D}_{\mathcal{T}} \,/\, DT_k = (DF_k, DC_k, E_k)\} \end{array}\right\} \Rightarrow \mathcal{D}_{\mathcal{T}} = \mathcal{D}_{\mathcal{TP}} \cup \mathcal{D}_{\mathcal{TC}} \tag{5.54}$$

Donde obviamente $\mathcal{D}_{\mathcal{TP}}$ es el conjunto de declaraciones de tipos primitivos, y $\mathcal{D}_{\mathcal{TC}}$ el de las de tipos compuestos. Dicho todo esto, estamos por fin en condiciones de establecer la única relación de *nombrado* que quedaba por definir, esto es:

Definición 5.35 (Nombrado de Tipos) *Se define el* nombrado de tipos *como una relación binaria $\mathcal{N}_T \subset (V_T \times \mathcal{D}_{\mathcal{T}})$ establecida entre cada uno de los nombres de tipo ($t \in V_T$) y la declaración de tipo ($DT_j \in \mathcal{D}_{\mathcal{T}}$) que describe su estructura.*

Como de costumbre, resultará más cómodo utilizar la sintaxis de una función, aunque en

este caso será preciso hacer ciertas puntualizaciones:

$$\mathcal{N}_T : \left.\begin{array}{ccc} V_T & \rightarrow & \mathcal{D}_{\mathcal{T}} \\ t & \rightarrow & DT_j \end{array}\right\} \Longrightarrow \mathcal{N}_T(t) \eqcirc DT_j \in \mathcal{D}_{\mathcal{T}} \tag{5.55}$$

Como función, $\mathcal{N}_T$ es obviamente total[12]; sin embargo, no se puede asegurar que sea *determinista*. Si no lo es, entonces no puede considerarse una función propiamente dicha. En realidad, la inmensa mayoría de los modelos de Arquitectura de Software habrán de asumir que a cada arquetipo le corresponde una única declaración, esto es, una única estructura. Sin embargo, existen también sistemas más complejos, en los que una estructura puede ser variable; éste es el caso de $\delta arwin$, Koala o el propio $\mathcal{P}i\mathcal{L}ar$. El modo más sencillo de indicar esta variabilidad es hacer corresponder a un único nombre de tipo, simultáneamente, con varias estructuras.

En ese caso, la aplicación de la función $\mathcal{N}_T$ no tendría como respuesta *una* única declaración de tipo, sino un *conjunto* de ellas. Por tanto, la expresión que debería aparecer en la ecuación 5.55 sería $\mathcal{N}_T(t) \ni DT_j$, para cada una de las estructuras alternativas. No obstante, y aunque éste es el caso más general, no es el más habitual; éste último se expresaría simplemente como $\mathcal{N}_T(t) = DT_j$, y manifiesta mejores propiedades. Se ha utilizado el símbolo $\eqcirc$, en definitiva, con el objetivo de no optar explícitamente por la pertenencia o la igualdad.

En cualquier caso, y aunque es sin duda un aspecto interesante, no es uno de los que se pretendía tratar en este modelo, por lo que $\aleph_\Sigma$ se deja en este punto conscientemente *abierto*. Tratarlo adecuadamente requeriría la definición de algún tipo de correspondencia parcial, de una noción explícita de subtipado, o incluso de un sistema de tipos dinámico[13]. Todas estas alternativas van mucho más allá de los objetivos de este Capítulo, por lo que no serán desarrolladas.

En cualquier caso, puede verse que la relación de nombrado $\mathcal{N}_T$ que acaba de definirse se aplica sobre el conjunto V_T en su totalidad, abarcando pues a todos sus subconjuntos. Por tanto, se refiere igualmente a tipos de componente o de conexión, y dentro de los primeros, se cubren tanto los primitivos como los compuestos. Teniendo en cuenta que la estructura correspondiente a un tipo de conexión no podrá ser compuesta[14], queda claro entonces que constará únicamente de puertos, por lo que se agrupa, por definición, en el conjunto $\mathcal{D}_{\mathcal{TP}}$ de declaraciones de tipos primitivos –véase la ecuación 5.54–. En resumen, se cumple que:

$$\mathcal{N}_T(t) \eqcirc DT_j \Rightarrow \left\{\begin{array}{lcl} t \in V_{TB} & \Longrightarrow & DT_j \in \mathcal{D}_{\mathcal{TP}} \\ t \in V_{TIP} & \Longrightarrow & DT_j \in \mathcal{D}_{\mathcal{TP}} \\ t \in V_{TIC} & \Longleftrightarrow & DT_j \in \mathcal{D}_{\mathcal{TC}} \end{array}\right. \tag{5.56}$$

Precisamente por esta razón, se asumirá en líneas generales que se pueden asimilar sin problemas los tipos de conexión y de componente primitivo, ya que en $\aleph_\Sigma$ ambos se corresponderán

[12] De nuevo, aunque V_T es potencialmente infinito, sólo se consideran los tipos que son efectivamente utilizados, al igual que se indicaba en la nota 11 de la página 132. Por tanto, esto sólo supone decir que todo arquetipo del sistema ha sido declarado, y tiene por tanto una estructura.

[13] Es decir, uno en que a cada nombre de tipo se corresponde con una única estructura, y donde $\mathcal{N}_T$ es determinista; pero donde una instancia o conexión podría tener asignados varios nombres de tipo, de modo que ahora la no determinista sería la relación de tipado ($\mathcal{T}$), lo que definiría un esquema aún más complejo.

[14] En $\aleph_{\mathrm{M}}$, una conexión podrá manifestar una estructura compleja, pero lo hará a través de una relación de reificación. En el modelo básico, no tiene sentido plantear la definición de sub-instancias dentro de una conexión, como no se ha hecho tampoco en el apartado correspondiente.

únicamente con una interfaz, definida como un conjunto de etiquetas de puerto, que además *no* estarán tipadas. Por tanto, sus estructuras serán totalmente indistinguibles.

En lo que sigue, se denominará abreviadamente como *arquetipo primitivo* a todo nombre de tipo $t \in V_T$ que designe, a través de la relación de nombrado $\mathcal{N}_T$, a una declaración de tipo primitivo ($\mathcal{N}_T(t) \in \mathcal{D}_{\mathcal{TP}}$), y *arquetipo compuesto* a todo nombre de tipo $u \in V_T$ que haga lo propio con una de tipo compuesto ($\mathcal{N}_T(u) \in \mathcal{D}_{\mathcal{TC}}$). Obviamente, y por lo que acaba de verse, el primer grupo coincidirá con el conjunto unión ($V_{TB} \cup V_{TIP}$), y el segundo, de manera aún más intuitiva, con el subconjunto V_{TIC}.

Hasta ahora, se había introducido la estructura de tipos como algo independiente del resto del modelo; sin embargo, su objetivo es precisamente comprobar que la arquitectura, definida en términos de instancias y conexiones (V_I, V_B) se organiza en un esquema consistente. Para ello, por tanto, se ha de establecer una correspondencia entre ambas. Si esto es efectivamente posible y se cumplen ciertas restricciones, se habrá garantizado la coherencia de la arquitectura.

Se comienza, entonces, con una serie de nociones de compatibilidad:

Definición 5.36 *$\in V_I$ y un arquetipo $\underline{t} \in V_{TI}$, se dice $\underline{i} \leftrightharpoons_F \underline{t}$, si se puede establecer* al menos una *correspondencia biunívoca (esto es, una biyección) entre los* nombres de puerto *en la frontera de la configuración nombrada por $\underline{i}$, y las* etiquetas *en la interfaz de la declaración de tipo nombrada por $\underline{t}$. Es decir:*

$$\underline{i} \leftrightharpoons_F \underline{t} \iff \left\{ \begin{array}{c} \exists \mathcal{B} \text{ biyección } \subset (V_P \leftrightarrow V_E) : \\ \forall e \in \pi_1(\mathcal{N}_T(\underline{t})),\ \forall p \in \mathcal{F}(\mathcal{N}_I(\underline{i})) : \\ (e,p) \in \mathcal{B} \end{array} \right. \tag{5.57}$$

Donde $\pi_1(\mathcal{N}_T(\underline{t}))$ es, como ya se ha dicho, el primer elemento de la terna que constituye una declaración de tipo, esto es, la declaración de interfaz DF_j correspondiente a la estructura del tipo designado por $\underline{t}$, y obtenida a través de la relación de nombrado $\mathcal{N}_T$.

En realidad, lograr este tipo de compatibilidad es muy sencillo, ya que al haberse abstraído la información de tipo relacionada con los puertos, se puede encontrar una correspondencia siempre que haya el mismo número de elementos en ambas interfaces. Por tanto, hubiera bastado con comparar las cardinalidades de los dos conjuntos:

$$\underline{i} \leftrightharpoons_F \underline{t} \iff |\pi_1(\mathcal{N}_T(\underline{t}))| = |\mathcal{F}(\mathcal{N}_I(\underline{i}))|$$

Por otra parte, la misma definición puede aplicarse a conexiones, sin más que cambiar los conjuntos referenciados en la misma. Es decir, se hablaría de la compatibilidad de interfaz entre una conexión $\underline{b} \in V_B$ y un arquetipo $\underline{t} \in V_{TB}$, y se habría de tener en cuenta que la interfaz de la primera viene dada por el conjunto frontera $\mathcal{F}(\mathcal{N}_B(\underline{b}))$, pero por lo demás la definición sería totalmente idéntica. Además, ya se ha dicho –ecuación 5.56– que en este contexto, los tipos de conexión y de componente primitivo se consideran completamente análogos; por tanto, se considera que no merece la pena establecer una distinción entre la compatibilidad de interfaz en componentes o en conexiones. Así pues, en lugar de dar una nueva definición, se amplía la

anterior para abarcar a ambos tipos de entidad, de modo que ésta queda finalmente como:

$$\underline{c} \leftrightharpoons_F \underline{t} \iff \begin{cases} \exists \mathcal{B} \text{ biyección } \subset (V_P \leftrightarrow V_E) : \\ \quad \forall e \in \pi_1(\mathcal{N}_T(\underline{t})),\ \forall p \in \mathcal{F}(\mathcal{N}_I(\underline{c})) : \qquad \text{si } \underline{c} \in V_I \\ \qquad\qquad (e,p) \in \mathcal{B} \\ \exists \mathcal{B} \text{ biyección } \subset (V_P \leftrightarrow V_E) : \\ \quad \forall e \in \pi_1(\mathcal{N}_T(\underline{t})),\ \forall p \in \mathcal{F}(\mathcal{N}_B(\underline{c})) : \qquad \text{si } \underline{c} \in V_B \\ \qquad\qquad (e,p) \in \mathcal{B} \end{cases} \tag{5.58}$$

Para simplificar la notación en lo sucesivo, se proporciona la siguiente definición auxiliar, que desarrolla una variante del operador de selección del álgebra relacional:

Definición 5.37 (Selección por Etiqueta) *Sea una relación binaria $\mathcal{R}$, definida como un conjunto de pares etiquetados, es decir, en los que el primer elemento es una etiqueta. Se define entonces el operador derivado de* selección por etiqueta*, y se expresa mediante $\widetilde{\sigma}_{\underline{e}}$, como la operación que* [illegible] *determinada, y* proyecta *después el resultado, devolviendo únicamente el segundo miembro de cada par.*

$$\widetilde{\sigma}_{\underline{e}}(\mathcal{R}) = \pi_2(\sigma_{(1=\underline{e})}(\mathcal{R})) \quad (\forall \mathcal{R} \subset (V_E \times \Psi)) \ \wedge \ (\forall \underline{e} \in V_E) \tag{5.59}$$

Donde Ψ puede ser cualquier conjunto, y $\sigma_{(1=\underline{e})}$ es el operador convencional de selección relacional, en el que se especifica como *condición de selección* que el primer atributo, indicado aquí simplemente como 1, ha de ser igual a la etiqueta especificada, $\underline{e}$.

En definitiva, lo que hace este operador es buscar todos los pares de la relación que han sido marcados con una determinada etiqueta, y devolver el correspondiente campo de *contenido*. En términos generales, el resultado es un conjunto, esto es, un subconjunto de Ψ. Sin embargo, dado que será generalmente aplicado a relaciones en las que se ha asegurado la unicidad de las etiquetas, este conjunto será con frecuencia unitario.

Conocido este operador, se puede definir ya la relación siguiente:

Definición 5.38 (Compatibilidad de Composición) *Dada una instancia compuesta $\underline{i} \in V_I$ y un arquetipo compuesto $\underline{t} \in V_{TIC}$, se dice que definen* composiciones *o* compuestos compatibles*, y se expresa como $\underline{i} \leftrightharpoons_C \underline{t}$, si se puede establecer* al menos una *biyección entre los* nombres de instancia *y* de conexión *contenidos en la configuración nombrada por $\underline{i}$, y las* etiquetas *de la declaración de configuración DC_j obtenida a partir de $\underline{t}$, de modo que el tipo de cada instancia contenida en $\underline{i}$ coincida con el tipo seleccionado, en dicha declaración de configuración, por la etiqueta con la que se le empareja.*

Es decir: dada una instancia compuesta $\underline{i}$, se toma el conjunto de todas sus sub-instancias y sub-conexiones $(\mathcal{J}_I(\underline{i}) \cup \mathcal{J}_B(\underline{i}))$. Por otra parte, dado un arquetipo compuesto $\underline{t}$, se obtiene la declaración de configuración correspondiente $(\pi_2(\mathcal{N}_T(\underline{t})))$, y se toma el conjunto de todas sus etiquetas $(\pi_E(\pi_2(\mathcal{N}_T(\underline{t}))))$. Se define entonces una biyección $\mathcal{B}$ entre estos dos conjuntos, constituida por pares de la forma (c, e), en los que el primer término es una instancia o conexión c del primer conjunto, y el segundo término es una etiqueta e del segundo conjunto.

Sin embargo, no se considera cualquier biyección $\mathcal{B}$, sino solamente una que respete las anotaciones de tipo ya existentes. Así, un par (c, e) sólo es válido si el tipo obtenido a partir de

la instancia o conexión ($\mathcal{T}(c)$) coincide con el tipo seleccionado por la etiqueta en la declaración de configuración ($\widetilde{\sigma}_e(\pi_2(\mathcal{N}_T(\underline{t})))$). En resumen:

$$\text{Sean} \quad \underline{i} \in V_I \,/\, \mathcal{N}_I(\underline{i}) \in KC, \quad \underline{t} \in V_{TIC} \,/\, \mathcal{N}_T(\underline{t}) \in \mathcal{D}_{\mathcal{TC}} :$$

$$\underline{i} \leftrightharpoons_C \underline{t} \iff \begin{cases} \exists \mathcal{B} \text{ biyección } \subset ((V_I \cup V_B) \leftrightarrow V_E) : \\ \quad \forall c \in (\mathcal{J}_I(\underline{i}) \cup \mathcal{J}_B(\underline{i})), \forall e \in \pi_E(\pi_2(\mathcal{N}_T(\underline{t}))) : \\ \qquad (c, e) \in \mathcal{B} \ \wedge \ (\mathcal{T}(c) = \widetilde{\sigma}_e(\pi_2(\mathcal{N}_T(\underline{t})))) \end{cases} \tag{5.60}$$

Obsérvese que este es uno de los casos en los que la selección por etiqueta ($\widetilde{\sigma}$) devuelve un único elemento (un nombre de tipo), ya que la unicidad de la etiqueta está asegurada por la propia estructura de declaración de configuración –definición 5.32–. Téngase en cuenta, asimismo, que para los elementos interiores no se considera diferencia alguna respecto a tipos de conexión o de componente, primitivos o compuestos.

Antes de pasar al siguiente tipo de compatibilidad, es preciso poder determinar cuándo un esquema de conexión –contenido en una declaración de tipo– define una estructura válida, en el sentido de que no contradice a la relación de nombrado:

Definición 5.39 (Consistencia de un Esquema) *Dado un esquema de conexión E_j, contenido en una declaración de tipo $DC_j \in \mathcal{D}_{\mathcal{TC}}$, se dice que es* consistente *si todos los pares de cada secuencia definida en E_j agrupan una* etiqueta de instancia *con una* etiqueta de puerto *de modo que se respeta la estructura definida por la relación de nombrado $\mathcal{N}_T$. Es decir:*

$$\begin{aligned} &\forall\, eb \in \pi_E(DB_j) : \\ &\quad \forall\, (ei, ep) \in \widetilde{\sigma}_{eb}(E_j) : \\ &\qquad \begin{cases} ep \in \pi_1(\mathcal{N}_T(\widetilde{\sigma}_{ei}(DC_j))) & \text{si } ei \neq \Upsilon \\ ep \in DF_j & \text{si } ei = \Upsilon \end{cases} \\ &\quad \wedge \quad |\pi_2(\widetilde{\sigma}_{eb}(E_j))| = |\mathcal{N}_B(\widetilde{\sigma}_{eb}(DB_j))| \end{aligned} \tag{5.61}$$

La condición que expresa esta ecuación garantiza la consistencia mencionada, integrando la estructura del esquema de conexión con la relación de nombrado. Tal como se dijo en la definición 5.34, a cada etiqueta de conexión eb le corresponde una secuencia de pares (ei, ep), en los que el primer elemento es una etiqueta de instancia, y el segundo una etiqueta de puerto. Lo que aquí se comprueba es que la (declaración de) interfaz del arquetipo referenciado por ei contiene, efectivamente, la etiqueta de puerto ep.

Se parte, pues, de la declaración de tipo (DT_j) que contiene el esquema a considerar (E_j), y se examina, para cada etiqueta de conexión eb, cada uno de los pares (ei, ep) de la secuencia correspondiente. Cuando la etiqueta de instancia ei es el símbolo especial Υ, entonces basta con comprobar que la etiqueta de puerto está en la interfaz DF_j del tipo actual.

Cuando la etiqueta es otra, se la emplea para seleccionar el arquetipo correspondiente en la declaración de configuración DC_j. Se usa entonces la relación de nombrado $\mathcal{N}_T$ para obtener la declaración de este nuevo tipo, y se extrae, mediante proyección, el conjunto de etiquetas de puerto que define su interfaz. Si este conjunto incluye la etiqueta de puerto ep buscada, se está *respetando* la estructura de nombres de tipo. La segunda parte de la condición indica, simplemente, que para cada etiqueta de conexión, la longitud de la secuencia de pares es igual

a la cardinalidad de la (declaración de) interfaz correspondiente al arquetipo asignado. De este modo, se tiene el mismo número de puertos en ambas definiciones, lo que implica que, en efecto, *el esquema es consistente*. Se ha comprobado, en definitiva, que cada par de etiquetas hace referencia a un puerto concreto que efectivamente existe en la interfaz del arquetipo con el que se empareja, identificándolo pues de manera unívoca.

Asumiendo, por tanto, que todo esquema es consistente en el sistema, se puede pasar a definir la noción de compatibilidad correspondiente:

Definición 5.40 (Compatibilidad de Interacción) *Dada una instancia compuesta* $\underline{i} \in V_I$ *y un arquetipo compuesto* $\underline{t} \in V_{TIC}$*, se dice que definen* esquemas de interacción compatibles*, o que son* compatibles por interacción*, y se expresa como* $\underline{i} \leftrightharpoons_I \underline{t}$*, si se puede establecer* al menos una *biyección entre los nombres de instancia y conexión contenidos en la instancia* $\underline{i}$*, y las etiquetas de instancia y conexión contenidas en el tipo* $\underline{t}$*, de modo que cada secuencia del esquema de conexión de* $\underline{t}$ *define la misma conectividad que la conexión correspondiente. Es decir:*

$$\begin{array}{l} \text{Sean} \quad \underline{i} \in V_I \,/\, \mathcal{N}_I(\underline{i}) \in KC, \quad \underline{t} \in V_{TIC} \,/\, \mathcal{N}_T(\underline{t}) \in \mathcal{D}_{TC} : \\ \underline{i} \leftrightharpoons_C \underline{t} \iff \left\{ \begin{array}{l} \exists \mathcal{B} \text{ biyección } \subset ((V_I \cup V_B) \leftrightarrow V_E) : \\ \quad \forall c \in (\mathcal{J}_I(\underline{i}) \cup \mathcal{J}_B(\underline{i})), \forall e \in \pi_E(\pi_2(\mathcal{N}_T(\underline{t}))) : \\ \qquad (c,e) \in \mathcal{B} \,/ \\ \left(\begin{array}{l} \forall k \in \pi_1(\widetilde{\sigma}_e(\pi_3(\mathcal{N}_T(\underline{t})))) : \\ \left\{ \begin{array}{ll} \mathcal{B}^{-1}(k) \in \circledast(c) & \text{si } c \in V_I \\ \mathcal{B}^{-1}(k) \in \divideontimes(c) & \text{si } c \in V_B \end{array} \right. \end{array} \right) \end{array} \right. \end{array} \tag{5.62}$$

Obsérvese que, a pesar de su aparente complejidad, esta definición resulta muy similar a la de la compatibilidad de composición. Las restricciones que definen la biyección $\mathcal{B}$ son idénticas; de hecho, en el caso ideal es *la misma*. La única diferencia es la condición asociada a cada par ordenado (c, e), ya que hace referencia a la conectividad de ambos elementos.

Esta condición se desarrolla del siguiente modo: se toma, en primer lugar, el esquema de conexión correspondiente al arquetipo $\underline{t}$, a través de la relación de nombrado $(\pi_3(\mathcal{N}_T(\underline{t})))$. Se toma entonces la etiqueta e del par, y se utiliza para seleccionar $(\widetilde{\sigma})$ en este esquema la secuencia correspondiente. Esta secuencia puede verse como una relación, de modo que al proyectar sobre su primer atributo (π_1) se obtiene el conjunto de *etiquetas de instancia* que se "conectan" con la etiqueta e del par. Se determina, entonces, para cada una de las etiquetas de este conjunto, el elemento correspondiente según la biyección $\mathcal{B}$, y se comprueba que está incluido en el *contexto* del término c del par, sea éste una instancia o conexión. Dado que esto ha de cumplirse para todo elemento contenido en ambas estructuras, es evidente que describen la misma estructura de interacción, y su conectividad es por tanto la misma.

Con esto queda descrita esta compatibilidad, la última de las básicas. Pueden ahora reunirse todas las definidas hasta aquí en un único concepto global, de modo que:

Definición 5.41 (Compatibilidad Débil) *Dada una instancia (o conexión)* $\underline{c} \in (V_I \cup V_B)$ *y un arquetipo* $\underline{t} \in V_T$*, se dice que son* débilmente compatibles*, y se expresa como* $\underline{c} \leftrightharpoons_1 \underline{t}$*, si y sólo si ambos son compatibles en interfaz y, simultáneamente, compatibles por composición e interacción, cuando estas relaciones están definidas. Es decir:*

$$\underline{i} \leftrightharpoons_1 \underline{t} \iff \begin{cases} \underline{c} \leftrightharpoons_F \underline{t} & \text{si } \underline{c} \in V_I \wedge \underline{t} \in V_{TIP} \\ \underline{c} \leftrightharpoons_F \underline{t} & \text{si } \underline{c} \in V_B \wedge \underline{t} \in V_{TB} \\ (\underline{c} \leftrightharpoons_F \underline{t}) \wedge (\underline{c} \leftrightharpoons_C \underline{t}) \wedge (\underline{c} \leftrightharpoons_I \underline{t}) & \text{si } \mathcal{N}_I(\underline{c}) \in KC \wedge \underline{t} \in V_{TIC} \end{cases} \tag{5.63}$$

Esto significa que, en el caso de instancias y tipos primitivos, o equivalentemente conexiones, la compatibilidad débil se reduce a compatibilidad de interfaz, ya que los otros dos aspectos no están definidos; sólo en el caso de instancias y arquetipos compuestos se habrán de tener en cuenta las tres relaciones. De hecho, la igualdad:

$$\leftrightharpoons_1 = ((\leftrightharpoons_F) \cap (\leftrightharpoons_C) \cap (\leftrightharpoons_I))$$

es cierta si se aplica exclusivamente sobre la intersección de sus dominios de definición, es decir, sobre los conjuntos de instancias y arquetipos compuestos.

Esta noción supone ya un nivel de correspondencia más que razonable entre los arquetipos y sus instancias. Sin embargo, para asegurar que la totalidad de la arquitectura es consistente, es preciso desarrollar el concepto a lo largo de la estructura jerárquica definida por los tipos e instancias. Para ello, será preciso hacer todavía una serie de definiciones auxiliares:

Definición 5.42 (Contención de Tipos) *Se define la relación de* contención de tipos*, entre dos arquetipos* $(t, u \in V_T)$ *dados, como una relación binaria homogénea* $\sqsupset \subset (V_T \times V_T)$*, de modo que se tiene que* $(t, u) \in \sqsupset$ *si el primer nombre (t) hace referencia a una declaración de configuración en la que al menos un par etiquetado contiene al segundo nombre (u).*

Se admite, como de costumbre, la notación infija $t \sqsupset u \equiv (t, u) \in \sqsupset$. Entonces:

$$t \sqsupset u \iff u \in (\pi_2(\pi_2(\mathcal{N}_T(t)))) \tag{5.64}$$

En definitiva, se toma el tipo *contenedor* (t), y se obtiene la declaración de tipo asociada $(\mathcal{N}_T(t))$. Dentro de ésta, se considera exclusivamente el segundo miembro, es decir, la declaración de configuración, que es una relación binaria; se proyecta entonces el segundo elemento de cada par de esta relación. De este modo, se ha obtenido el conjunto de los nombres de tipo utilizados en las sub-instancias (y sub-conexiones) de t. Cualquier arquetipo u perteneciente a este conjunto estará por tanto (directamente) *contenido* en t.

Esta relación define, por tanto, una relación de agregación clásica, tal como se considera habitualmente (al nivel de tipos). Es también evidente que existe un gran parecido entre ésta y la contención de instancias ($\ni$). Sin embargo, hay entre ambas diferencias cualitativas de gran importancia. La contención de instancias se definía sobre elementos individuales, identificados de manera única; y por tanto, era irreflexiva, asimétrica e intransitiva. En cambio, la de tipos se define sobre patrones estructurales, que no tienen por qué utilizarse en un único lugar de la jerarquía; por tanto, no tendrá *ninguna* de las tres propiedades, ni tampoco sus opuestas. Un tipo podrá (o no) contenerse a sí mismo, contener a un tipo que le contenga a él, o contener directa e indirectamente –a través de otro– a un mismo tipo.

Precisamente, la noción de contención indirecta que acaba de mencionarse resulta especialmente interesante; al ser la contención de tipos una relación homogénea, es muy sencillo definir ésta como su cierre transitivo, de modo que:

Definición 5.43 (Contención General de Tipos) *Se define la* contención general *o* indirecta *sobre tipos como una relación binaria homogénea* $\sqsupset\!\!\!\sqsupset$*, que viene dada por el cierre transitivo de de* $\sqsupset$*, esto es, por la menor relación binaria transitiva sobre* V_T *que contiene a* $\sqsupset$*.*

$$\begin{array}{lcll} \sqsupset\!\!\!\sqsupset & : & V_T \leftrightarrow V_T & \\ \sqsupset\!\!\!\sqsupset & \doteq & \sqsupset^+ & = \bigcup_{i=1}^{n}(\sqsupset^i) \quad \text{para } n \leq \binom{|V_T|}{2} \end{array} \tag{5.65}$$

Su desarrollo es obvio, y establece una analogía similar con la relación de contención de instancias $\Supset$. Sin embargo, de nuevo, se debe tener en cuenta que su forma es diferente; al igual que ocurría en la anterior, esta contención de tipos no es ni irreflexiva ni asimétrica, aunque sí será –por construcción– transitiva. Además, para establecer una correlación razonable con la contención de instancias, será preciso incluir también a las conexiones; es por tanto preciso ampliar la definición original, de modo que:

Definición 5.44 (Contención Extendida) *Se define la relación de* contención extendida *entre una instancia* $(i \in V_I)$ *y otra instancia o conexión* $(j \in (V_I \cup V_B))$ *como una relación binaria hetereogénea* $\widetilde{\Supset} \subset (V_I \times (V_I \cup V_B))$*, de modo que se tiene que* $(i, j) \in \widetilde{\Supset}$ *si el primer nombre* (i) *designa a la configuración que contiene a la instancia o conexión referenciada con el segundo nombre* (j)*. Se admite, de nuevo, la notación infija* $i\widetilde{\Supset}j \equiv (i, j) \in \widetilde{\Supset}$*. Entonces:*

$$i\widetilde{\Supset}j \iff \begin{cases} j \in \mathcal{J}_I(i) & \text{si } j \in V_I \\ j \in \mathcal{J}_B(i) & \text{si } j \in V_B \end{cases} \tag{5.66}$$

La definición resulta obvia a estas alturas, y no requiere de mayor explicación. Se trata tan sólo de una amalgama que permite hablar simultáneamente de instancias y conexiones. Puede comprobarse que el fragmento de $\widetilde{\Supset}$ referido a instancias coincide totalmente con $\Supset$; ambas muestran, además, las mismas propiedades.

Del mismo modo, interesa definir una versión transitiva de $\widetilde{\Supset}$. Sin embargo, ahora no es posible establecer el cierre, ya que la relación es hetereogénea. Más aún, las conexiones son siempre primitivas y no definen contención alguna, por lo que sólo aparecen como elementos terminales en un camino de instancias –véase la definición 5.8–. Por tanto, se trata tan sólo de aprovechar la estructura creada por la contención entre instancias, y ampliar sus extremos para incluir las conexiones. Esto puede expresarse como:

Definición 5.45 (Contención Extendida General) *Se define la relación de* contención extendida general*, representada por* $\widetilde{\Supset\!\!\!\Supset}$*, como la composición de la relación de contención general de instancias* $(\Supset\!\!\!\Supset)$*, y la relación de contención extendida recién definida. Es decir:*

$$\begin{array}{lcl} \widetilde{\Supset\!\!\!\Supset} & : & V_I \to (V_I \times (V_I \cup V_B)) \\ \widetilde{\Supset\!\!\!\Supset} & \doteq & (\Supset\!\!\!\Supset \circ \widetilde{\Supset}) \end{array} \tag{5.67}$$

Por tanto, una instancia o conexión (j) estará $\widetilde{\Supset\!\!\!\Supset}$-contenida en una instancia (i) si y sólo si está $\widetilde{\Supset}$-contenida en ella, o bien en otra instancia (k) que, a su vez, esté $\widetilde{\Supset\!\!\!\Supset}$-contenida en i.

Conocidas todas las relaciones de contención, y la noción de compatibilidad débil, se puede definir, por fin, la relación siguiente, que será ya la definitiva:

Definición 5.46 (Compatibilidad Fuerte) *Dada una instancia (o conexión)* $\underline{c} \in (V_I \cup V_B)$ *y un arquetipo* $\underline{t} \in V_T$*, se dice que son* fuertemente compatibles*, o simplemente* compatibles*, si y sólo si son débilmente compatibles, y existe una función suprayectiva entre los nombres* contenidos *por cada uno de ellos, de modo que los elementos que ésta empareja son, a su vez,* fuertemente compatibles *entre sí. Es decir:*

$$\underline{c} \leftrightharpoons \underline{t} \iff (\underline{c} \leftrightharpoons_1 \underline{t}) \wedge \begin{cases} \exists \mathcal{S} \text{ suprayección } \subset ((V_I \cup V_B) \times V_T) : \\ \quad \forall k \in (V_I \cup V_B) \,/\, \underline{c} \,\widetilde{\ni}\, k,\ \forall u \in V_T \,/\, \underline{t} \sqsupset u : \\ \qquad (k, u) \in \mathcal{S} \wedge k \leftrightharpoons u \end{cases} \tag{5.68}$$

Obviamente, se trata de una definición recursiva, lo que es legítimo pero a menudo conflictivo. Esta relación es hetereogénea, al igual que la de compatibilidad débil, de la que procede. Por tanto, no puede plantearse su cierre transitivo, que es el modo más habitual de eliminar la recursión. Sin embargo, sí que puede desplegarse en términos de los cierres transitivos de ambas contenciones; de modo que se reestructurará la definición, de manera no recursiva, como:

$$\underline{c} \leftrightharpoons \underline{t} \iff (\underline{c} \leftrightharpoons_1 \underline{t}) \wedge \begin{cases} \exists \mathcal{S} \text{ suprayección } \subset ((V_I \cup V_B) \times V_T) : \\ \quad \forall k \in (V_I \cup V_B) \,/\, \underline{c} \,\widetilde{\ni\!\!\ni}\, k,\ \forall u \in V_T \,/\, \underline{t} \,\sqsupset\!\!\sqsupset u : \\ \qquad (k, u) \in \mathcal{S} \wedge k \leftrightharpoons_1 u \end{cases} \tag{5.69}$$

Obsérvese que la definición ya no utiliza la compatibilidad fuerte, sino la débil. Ésta resulta ahora suficiente, ya que no sólo se plantea para los elementos directamente contenidos ($\widetilde{\ni}$, $\sqsupset$) en los dos implicados, sino en todos los contenidos en ésta de manera *indirecta* y a cualquier nivel de profundidad, desplegándose a través de las relaciones transitivas ($\widetilde{\ni\!\!\ni}$, $\sqsupset\!\!\sqsupset$).

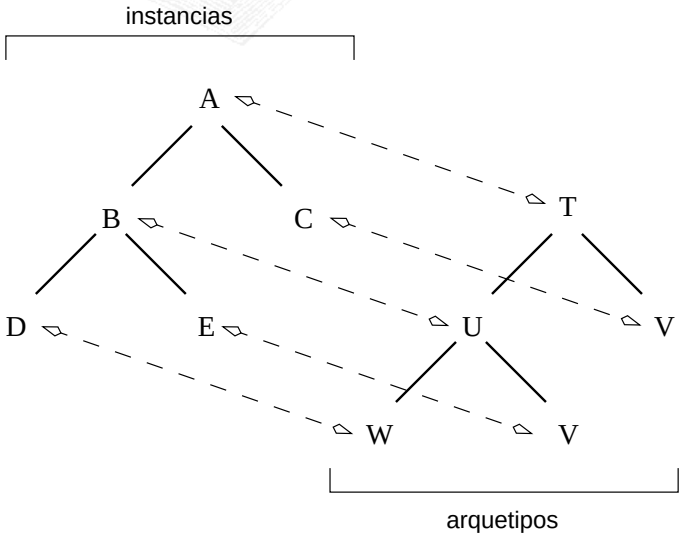

Figura 5.1: Correspondencia entre Arquetipos e Instancias

Ha de señalarse el hecho de que aquí ya no se exige una biyección entre los elementos, como en las definiciones anteriores, sino solamente una suprayección, es decir, una función (determinista

y total, en este caso) tal que a todo elemento del codominio (es decir, a todo tipo) le corresponde al menos un elemento del dominio (una instancia o conexión). Lo que ocurre, en realidad, es que en una declaración de conexión sólo se había garantizado la unicidad de las etiquetas, y éstas han sido eliminadas por proyección; por tanto, es probable –e incluso frecuente– que los arquetipos se utilicen más de una vez. Esto elimina la posibilidad de plantear una correspondencia uno a uno, y resulta preciso conformarse con una suprayección. En realidad, en esencia ésta sigue siendo la misma relación: si se tuviesen en cuenta las repeticiones de los nombres de tipo –usando, por ejemplo, un multiconjunto–, se tendría de hecho una biyección.

La Figura 5.1 expresa de manera esquemática un ejemplo de la clase de correspondencia que se ha de lograr para que una instancia compuesta A y un arquetipo T sean fuertemente compatibles. Por supuesto, el esquema está simplificado, ya que sólo muestra las relaciones de contención[15] y la propia correspondencia (función suprayectiva). Debe de tenerse en cuenta que entre los nodos *hermanos* de los árboles definidos por ambas jerarquías existen todas las relaciones de interacción definidas por $\mathcal{A}^*$ y cada esquema de conexión E_j, y que éstas influyen de manera determinante en la suprayección, a través de la noción de compatibilidad débil.

Con esto, se ha definido por fin la estructura completa con la que se pretendía verificar la consistencia interna de una descripción arquitectónica. Una vez establecida esta estructura, se puede resumir su influencia en el concepto siguiente:

Definición 5.47 (Buen Tipo) *Se dice que una arquitectura está* bien tipada*, o que muestra un* buen tipo*, si el esquema de tipos dado por $\mathcal{T}$ es tal que a toda instancia o conexión se le asigna un arquetipo (fuertemente) compatible con ella.*

$$\text{Buen tipo} \equiv \Big((\, c \leftrightharpoons \mathcal{T}(c)\,)\ \forall c \in (V_I \cup V_B) \Big) \tag{5.70}$$

Con este concepto se llega, por tanto, al objetivo final de este apartado; en resumen, una arquitectura sólo describirá una estructura válida si ésta está *bien tipada*.

No obstante, y casi como mero comentario, ha de señalarse que a partir de la estructura establecida por las definiciones anteriores, pueden identificarse todavía una serie de nociones interesantes y particularmente complejas, como por ejemplo:

Definición 5.48 (Recursión Directa) *Sea un tipo $\underline{t} \in V_T$. Se dice que es* (directamente) recursivo *si se contiene (directamente) a sí mismo. Es decir:*

$$\underline{t} \text{ dir. recursivo} \iff \underline{t} \sqsupset \underline{t} \tag{5.71}$$

Es decir, es su propio padre, y también su propio hijo. Análogamente:

Definición 5.49 (Recursión) *Sea un tipo $\underline{t} \in V_T$. Se dice que es* (indirectamente) recursivo*, o simplemente* recursivo *si se contiene a sí mismo, en el sentido más general. Es decir:*

$$\underline{t} \text{ recursivo} \iff \underline{t} \,\widetilde{\sqsupset}\, \underline{t} \tag{5.72}$$

Y, finalmente, por extensión:

[15] Es decir, la relación $\sqsupset$ (entre tipos) y la versión extendida de la relación de contención, $\widetilde{\ni}$, referida tanto a instancias como a conexiones.

Definición 5.50 (Recursión Mutua) *Sean dos tipos* $\underline{t}, \underline{u} \in V_T$. *Se dice que son* mutuamente recursivos *si cada uno de ellos contiene (indirectamente) al otro. Es decir:*

$$\underline{t} \text{ mut. rec. con } \underline{u} \iff (\underline{t} \sqsupseteq \underline{u}) \wedge (\underline{u} \sqsupseteq \underline{t}) \tag{5.73}$$

Evidentemente, estas tres definiciones corresponden a estructuras con ciclos, y por tanto a esquemas de tipo *infinitos*. Como es sabido, el esquema de instancias será siempre finito, por lo que parece imposible, en principio, encontrar una correspondencia entre ambos. Esto equivaldría a decir que cualquier estructura en la que se incluyese un tipo recursivo, en cualquiera de las tres formas descritas, no podría estar bien tipada.

En realidad, esta restricción resulta excesiva. Es cierto que no todo sistema de tipos, en Arquitectura de Software o fuera de ella, admite recursión; pero en todo el que la admite, se establece un tratamiento especial para este caso. El más habitual hace referencia al despliegue de la recursión como un punto fijo, desarrollado bien como un árbol regular, como un hiperconjunto [JR97], o incluso como una relación de bisimulación [Tur96]. No resultaría, en principio, demasiado complicado integrar cualquiera de estos enfoques en nuestro sistema, en particular teniendo en cuenta que la noción de *compatibilidad* se basa en una función suprayectiva, y que en el caso de las instancias primitivas –precisamente, allí donde deberá detenerse toda recursión–, se reduce simplemente a compatibilidad de interfaces. De hecho, el punto clave se basaría en determinar si una instancia *primitiva* es compatible (en interfaz) con un arquetipo *compuesto*, y además *recursivo*. Dado que este tipo de relación no está definida, bastaría con una ligera ampliación para lograr un sistema de tipos nuevamente consistente.

En resumen, estos conceptos muestran los límites del sistema de tipos que aquí se ha establecido. Resulta necesario plantearse, en este punto, si es razonable seguir ampliándolo, para hacerlo capaz de manejar nociones complejas como ésta, o utilizar, de manera alternativa, una auténtica teoría de tipos. En realidad, no hay tanta diferencia entre ambos enfoques, dado que a menudo las teorías de tipos utilizan también teoría de relaciones como base. Pero, en todo caso, el sistema actual resulta suficiente para examinar los aspectos realmente relevantes, y ha servido, por lo tanto, a su propósito.

Así pues, no se considera necesario seguir tratándolo, y en lo sucesivo se entenderá que el sistema de tipos de MARMOL se limita a lo especificado en este apartado; respecto a los tipos recursivos, no obstante, se asumirá simplemente que son consistentes dentro de algún punto fijo, sin que se proporcionen más detalles sobre el particular.

5.5.2 Modelo MARMOL General ($\aleph_{\mathrm{M}}$)

Se define el modelo $\aleph_{\mathrm{M}}$ como una extensión del modelo $\aleph_{\Sigma}$ ya descrito, de modo que:

$$\aleph_{\mathrm{M}} = \langle V, S_M \rangle \ / \ \aleph_{\Sigma} = \langle V, S_V \rangle \ \wedge \ S_V \subset S_M$$

Es decir, todas las definiciones y operaciones del modelo original siguen estando presentes, pero en la nueva signatura S_M se incorpora toda una serie de nuevas operaciones, que son las que se definen en este apartado. El objetivo de $\aleph_{\mathrm{M}}$ es, como ya se ha dicho, incorporar en $\aleph_{\Sigma}$ todos los conceptos necesarios para la introducción de *Reflexión* en un contexto arquitectónico. De hecho, este es el objetivo principal, tanto de MARMOL, como de esta tesis, y es aquí donde se encarna de manera explícita.

Aunque el conjunto universal de nombres V no ha cambiado, sí se distingue una nueva categoría semántica, a añadir a las cinco ya existentes. Esta categoría es la noción de *reificación*, que obviamente no tenía sentido en el modelo básico.

No obstante, esta noción no debe entenderse aquí como la operación de reificación habitual, descrita en la sección 4.3.2, sino más bien como la relación de *conexión causal*. Aunque el primer término es menos preciso, se prefiere su uso para evitar confusiones con el concepto habitual de *conexión*. Además, en todo caso coincide con con la noción filosófica de reificación, aunque no en todas las interpretaciones [Qui85, Qui92]. De todos modos, esto sólo se aplica a la denominación general: en el modelo formal, la operación de reificación ($\curlywedge$) sí tendrá un sentido asignado, e incluso podrá verse cómo la conexión causal ($\diamond$) se deduce trivialmente de ésta.

Se considera, entonces, el siguiente nuevo conjunto de nombres:

- V_R – Conjunto de nombres de *enlace de reificación*, de modo que $r \in V_R \subset V$.

Ya se ha comentado, sin embargo, que la identificación de V_R no supone un cambio en V. Esto se debe a que, dado que los nombres de reificación no tenían sentido en $\aleph_\Sigma$, se encontraban acumulados dentro del conjunto residuo $\mho_S$, del cual se extraen ahora. Entonces:

$$\mho_S = V_R \cup \mho_M$$

De este modo, se obtiene el conjunto preciso, y se obtiene un nuevo conjunto residuo ($\mho_M$). Nunca se puede descartar la idea de tener un conjunto residuo, donde se puedan incorporar los nombres de nuevas categorías semánticas a incorporar. Por ejemplo, cuando se incorporan metaespacios, se definen simplemente como un conjunto –véase la definición 5.85–; pero si se desease considerar la existencia de *planos* o *modelos* (§4.3.4), se haría necesario identificarlos mediante un nombre, y para ello sería necesario incorporar un nuevo subconjunto de V, sea V_M. En tal caso, se tendría $\mho_M = V_M \cup \mho'_M$, y en definitiva:

$$V = V_P \cup V_I \cup V_T \cup V_B \cup V_E \cup (V_R \cup (V_M \cup \mho'_M))$$

En cualquier caso, este aspecto no será considerado en la presente versión de este modelo, ya que se considera que la definición de metaespacio permite obtener el mismo efecto, con un aparato conceptual menor. No obstante, esto no disminuye su utilidad práctica.

Como se sabe, el concepto de reflexión es muy complejo, y sus consecuencias formales son muy amplias. No obstante, lo que aquí se desea es definir estrictamente sus aspectos *estructurales*. Como ya se ha visto en la sección 4.5.1, en la práctica la noción de reflexión adopta distintas estrategias; sin embargo, siempre se expresa como el acceso privilegiado de una entidad especial –una meta-entidad– a las entidades del nivel base. En un modelo arquitectónico, las entidades de ambos tipos serán representadas como *componentes*; por tanto, para completar la descripción, lo único que deberá indicarse es la clase de vínculo creado entre estos componentes.

La relación entre un componente y su metacomponente se distingue cualitativamente tanto de la composición, dada por la contención, como de la interacción, dada por la ligadura. Esto significa que la *reificación* se establece como la **tercera dimensión** de la descripción arquitectónica, siempre que se considere un modelo reflexivo. En Marmol, y concretamente en $\aleph_{\mathrm{M}}$,

por tanto, la reificación se establece como un tercer tipo de relación, que fragmenta y proyecta tridimensionalmente el esquema descrito por el modelo básico $\aleph_\Sigma$.

No es necesario, desde este punto de vista, considerar la naturaleza de la relación establecida: ésta puede considerarse como reflexión, en abstracto, y más concretamente, sobre un contexto concurrente, como una superposición (§7.2.5). Sin embargo, esto no es más que una *interpretación*, desde el punto de vista del modelo. En $\aleph_{\mathrm{M}}$, la reificación es simplemente un concepto estructural, encarnado en una relación. Es esta abstracción la que lo hace tratable.

Definición 5.51 (Reificación Estricta) *Se define la* reificación estricta*, o simplemente* reificación*, como una relación binaria homogénea, asimétrica e intransitiva* $\curlywedge \subset (V_I \times V_I)$ *entre instancias de componente, de modo que para un par ordenado* $(\underline{m}, \underline{a}) \in \curlywedge$*, se dice que* $\underline{m}$ reifica a $\underline{a}$*, o que* $\underline{a}$ *es* reificado por $\underline{m}$*. Se admite, como de costumbre, la notación infija* $\underline{m} \curlywedge \underline{a}$*.*

Análogamente, se define a su vez:

Definición 5.52 (Reflexión Estricta) *Se define la* reflexión estricta*, o simplemente* reflexión*, como una una relación binaria homogénea, asimétrica e intransitiva* $\curlyvee \subset (V_I \times V_I)$ *entre instancias de componente, de modo que para un par ordenado* $(\underline{a}, \underline{m}) \in \curlyvee$*, se dice que* $\underline{a}$ refleja a $\underline{m}$*, o que* $\underline{m}$ se refleja en $\underline{a}$*. Se admite también la notación infija* $\underline{m} \curlyvee \underline{a}$*.*

Por definición, ésta es la relación inversa a la anterior, de modo que $\curlyvee = (\curlywedge)^{-1}$*.*

En resumen, se cumple, por tanto, que:

$$(a, m) \in \curlyvee \quad \equiv \quad a \curlyvee m \qquad \Longleftrightarrow \qquad m \curlywedge a \quad \equiv \quad (m, a) \in \curlywedge \tag{5.74}$$

Sin embargo, no sólo interesa reificar (respectivamente, reflejar) a las instancias de componente; sino que, como ya se ha dicho en la sección 5.3, también se pueden reificar las conexiones entre ellas. Por ello, se definen las siguientes relaciones análogas:

Definición 5.53 (Reificación de Conexión) *Se define la* reificación de conexión*, como una relación binaria hetereogénea* $\curlywedge \subset (V_I \times V_B)$ *entre instancias de componente y conexiones, de modo que para un par ordenado* $(\underline{m}, \underline{c}) \in \curlywedge$ *se dice que* $\underline{m}$ reifica a $\underline{c}$*, o que* $\underline{c}$ *es* reificada por $\underline{m}$*. Se admite igualmente la notación infija* $\underline{m} \curlywedge \underline{c}$*.*

Análogamente, se define a su vez:

Definición 5.54 (Reflexión de Conexión) *Se define la* reflexión de conexión*, como una relación binaria hetereogénea* $\curlyvee \subset (V_B \times V_I)$ *entre conexiones e instancias de componente, de modo que para un par ordenado* $(\underline{c}, \underline{m}) \in \curlyvee$ *se dice que* $\underline{c}$ refleja a $\underline{m}$*, o que* $\underline{m}$ se refleja en $\underline{c}$*. Se admite igualmente la notación infija* $\underline{c} \curlyvee \underline{m}$*.*

También por definición, esta relación es la inversa de la anterior, es decir, $\curlyvee = (\curlywedge)^{-1}$*.*

El objetivo principal de estas dos relaciones el de servir de puente entre el universo de conexiones y el de componentes, ya que –como se verá– la gran mayoría de las manipulaciones reflexivas se realizan, a distintos niveles, entre instancias de componente. De este modo, todas las conclusiones se aplicarán igualmente a conexiones.

Dicho esto, ya se pueden definir formalmente los dos conceptos siguientes:

Definición 5.55 (Meta-Componente) *Se dice que una instancia* $\underline{m}$ *es un* metacomponente *o un* rohatar[16] de *otra instancia* $\underline{a}$ *cuando se tiene que* $\underline{m} \curlywedge \underline{a}$*, es decir, cuando existe una* reificación estricta *desde* $\underline{m}$ *hasta* $\underline{a}$*. Esta definición se generaliza a las conexiones, de modo que* $\underline{m}$ *es también un* metacomponente para $\underline{c}$ *cuando se cumple que* $\underline{m} \curlywedge \underline{c}$*.*

Definición 5.56 *Por extensión, se dice que una instancia* $\underline{m}$ *es un* metacomponente *o un* rohatar*, en sentido general, cuando lo es de* alguna *instancia o conexión* $\underline{a}$*, es decir:*

$$\underline{m}(\in V_I) \text{ metacomponente} \iff \begin{cases} \exists \underline{i} \in V_I & : \quad (\underline{m} \curlywedge \underline{i}) \quad \equiv \quad (\underline{i} \curlyvee \underline{m}) \\ \quad \vee & \\ \exists \underline{c} \in V_B & : \quad (\underline{m} \curlywedge \underline{c}) \quad \equiv \quad (\underline{c} \curlyvee \underline{m}) \end{cases} \tag{5.75}$$

Definición 5.57 (Avatar) *Se dice que una instancia* $\underline{a}$ *es un* componente base*, un* componente objeto*, o un* avatar[17] de *otra instancia* $\underline{m}$ *cuando se tiene que* $\underline{a} \curlyvee \underline{m}$*, es decir, cuando existe una reflexión estricta desde a hasta m. Esta definición se generaliza a conexiones, de modo que una conexión* $\underline{c}$ *es también un* avatar de *una instancia* $\underline{m}$ *cuando* $\underline{a} \curlyvee \underline{m}$*.*

Es decir: $\underline{a}$ es un avatar de $\underline{m}$ cuando $\underline{m}$ es un metacomponente para él.

Definición 5.58 *Por extensión, se dice que una instancia o conexión* $\underline{a}$ *es un* avatar*, en sentido general, cuando lo es de* alguna *instancia* $\underline{m}$*, es decir:*

$$\underline{a}(\in V_I \cup V_B) \text{ avatar} \iff \exists \underline{m} \in V_I \ : \begin{cases} (\underline{a} \curlyvee \underline{m}) \quad \equiv \quad (\underline{a} \curlywedge \underline{m}) & \text{si } \underline{a} \in V_I \\ (\underline{a} \curlyvee \underline{m}) \quad \equiv \quad (\underline{a} \curlywedge \underline{m}) & \text{si } \underline{a} \in V_B \end{cases} \tag{5.76}$$

Una vez definido el vocabulario básico, se pueden establecer una serie de definiciones derivadas, comenzando por la equivalente estructural de la noción de conexión causal:

Definición 5.59 (Meta-Ligadura) *Se define la* meta-ligadura *o* conexión causal *como una relación binaria homogénea simétrica* $\diamond \subset (V_I \times V_I)$ *entre instancias de componente, de modo que se dice que dos instancias i y j están* meta-ligadas *o* conectadas causalmente *si existe una reificación (resp. reflexión) estricta entre ellas. Es decir:*

$$(i, j) \in \diamond \equiv (i \diamond j) \iff (i \curlywedge j) \ \vee \ (i \curlyvee j) \tag{5.77}$$

Obsérvese que la definición se aplica solamente a *instancias*; se ha preferido no extenderla a conexiones, ya que éstas carecen de contenido computacional explícito. No obstante, la generalización sería en cualquier caso la obvia.

Se hace evidente, además, que la relación de conexión causal expresa simplemente el *cierre simétrico* de la relación de reificación (resp. reflexión) estricta:

$$\diamond = (\curlywedge \cup \curlyvee) = (\curlywedge \cup (\curlywedge)^{-1}) = (\curlywedge)^S = (\curlyvee)^S \tag{5.78}$$

Es fácil igualmente deducir que, al igual que las dos relaciones que la originan, la meta-ligadura es también intransitiva, ya que la simetría no afecta a este aspecto.

[16] Del sánscrito *rohana-tar*, que significa literalmente *el que asciende.*

[17] Del sánscrito *avatara-tar*, que significa literalmente *el que desciende.*

Resulta obvio que esta noción no es estrictamente necesaria; se puede partir simplemente de la relación de reificación (resp. reflexión) estricta, y obtener todas las demás como derivadas suyas. En cambio, la operación inversa no es posible: dada la relación de conexión causal, simétrica, no se dispone de un criterio para obtener los fragmentos asimétricos correspondientes a las otras dos: es por ello que se considerará a la *reificación* (estricta y de conexión) como la única relación primitiva; el resto se infieren a partir de ésta.

En resumen, los *enlaces de reificación* se definen simplemente como pares ordenados de instancias en las relaciones de reificación estricta ($\curlywedge$) y de conexión ($\curlywedge$). De hecho, se constituyen como la entidad más simple del esquema, lo que no es un obstáculo, sin embargo, para que sean utilizados en la definición de algunas de las estructuras más complejas del mismo.

En cualquier caso, y como ya se dijo, estos enlaces de reificación se identifican mediante un nombre, lo que hace preciso la definición de la relación siguiente:

Definición 5.60 (Nombrado de Reificaciones) *Se define el* nombrado de reificaciones *como una relación binaria* $\mathcal{N}_R \subset (V_R \times (\curlywedge \cup \curlywedge))$*, establecida entre un nombre de reificación* ($r \in V_R$) *y cada uno de los pares que constituyen ambas relaciones de reificación. Es decir:*

$$\begin{array}{rcl} \mathcal{N}_R : \; V_R & \rightarrow & ((V_I \times V_I) \cup (V_I \times V_B)) \\ r & \rightarrow & \left\{ \begin{array}{lcl} (m,i) \in (V_I \times V_I) & / & (m \curlywedge i) \\ (m,b) \in (V_I \times V_B) & / & (m \curlywedge b) \end{array} \right. \end{array} \tag{5.79}$$

La relación definida puede tratarse como una función total biyectiva, ya que todo nombre de V_R se refiere a una reificación, y todo par de $\curlywedge$ y $\curlywedge$ recibirá un nombre. La expresión $\mathcal{N}_R(r)$, por tanto, denota a uno de estos pares. Este uso, en cualquier caso, no será muy habitual, ya que el tratamiento asociado a una reificación suele ir más vinculado a sus aspectos estructurales. De hecho, aunque sigue siendo necesario que estén identificadas, las reificaciones son el concepto del modelo para el que el *nombre* asociado tiene una importancia menor.

Se definen ahora una serie de relaciones (derivadas) para la manipulación de instancias, que nos permitirán llegar hasta la noción de *meta-niveles*. Sea, entonces:

Definición 5.61 (Reificación Débil) *Se define la* reificación débil *como una relación binaria homogénea asimétrica* $\hat{\curlywedge} \subset (V_I \times V_I)$ *entre instancias de componente, obtenida como el* cierre transitivo *de la relación de reificación estricta. Es decir:*

$$\begin{array}{lcl} \hat{\curlywedge} & : & V_I \leftrightarrow V_I \\ \hat{\curlywedge} & \stackrel{\circ}{=} & \curlywedge^+ = \bigcup_{i=1}^{n} (\curlywedge^i) \end{array} \tag{5.80}$$

Análogamente, se define a su vez:

Definición 5.62 (Reflexión Débil) *Se define la* reflexión débil *como una relación binaria homogénea asimétrica* $\check{\curlyvee} \subset (V_I \times V_I)$ *entre instancias de componente, obtenida como el* cierre transitivo *de la relación de reflexión estricta. Es decir:*

$$\begin{array}{lcl} \check{\curlyvee} & : & V_I \leftrightarrow V_I \\ \check{\curlyvee} & \stackrel{\circ}{=} & \curlyvee^+ = \bigcup_{i=1}^{n} (\curlyvee^i) \end{array} \tag{5.81}$$

Es evidente por construcción que, al igual que sus respectivas relaciones de origen, la reificación débil y la reflexión débil son relaciones inversas, es decir, que $\check{\curlyvee} = (\hat{\curlywedge})^{-1}$.

De ahí se infiere con naturalidad la relación estructural siguiente:

Definición 5.63 (Meta-Encadenamiento) *Se define el* meta-encadenamiento*, que también admite la denominación de* meta-ligadura débil *o* conexión causal débil*, como una relación binaria homogénea simétrica* $\hat{\diamond} \subset (V_I \times V_I)$*, obtenida como el* cierre transitivo *de la relación de meta-ligadura. Es decir:*

$$\begin{array}{lcl} \hat{\diamond} & : & V_I \leftrightarrow V_I \\ \hat{\diamond} & \stackrel{\circ}{=} & \diamond^+ = \bigcup_{i=1}^{n} (\diamond^i) \end{array} \tag{5.82}$$

Resulta evidente ahora, por construcción, que el meta-encadenamiento no sólo es el cierre simétrico de la reificación (resp. reflexión) débil, sino que además es, en consecuencia, el *cierre reflexivo transitivo* de la relación de reificación (resp. reflexión) estricta.

$$\hat{\diamond} = \hat{\curlywedge} \cup \check{\curlyvee} = (\hat{\curlywedge})^S = (\check{\curlyvee})^S = (\curlywedge^+)^S = \curlywedge^* = \curlyvee^* \tag{5.83}$$

Se dice pues, en definitiva, que dos instancias están *meta-encadenadas* si existe una cadena posible de pares de reificación o reflexión entre ellas. Por tanto, de nuevo es posible inferir toda la información a partir solamente de la relación de reificación (resp. reflexión) estricta.

Aunque ya se ha mencionado, resulta significativo señalar que toda instancia reificada está meta-encadenada consigo misma, ya que es un avatar para cualquiera de sus metacomponentes. Es decir, la relación de meta-encadenamiento es reflexiva[18]; esto hace que, en la práctica, sólo sea útil para determinar el área de influencia (reflexiva) de una instancia dada. La reificación (resp. reflexión) débil será, en cambio, mucho más importante.

Dada la hetereogeneidad de las relaciones de reificación y reflexión de conexiones, no se pueden hacer definiciones análogas basadas en su cierre transitivo. Sin embargo, lo que realmente interesa es considerar su composición con las de instancias, de modo que:

Definición 5.64 (Reificación Débil de Conexión) *Se define la* reificación débil de conexión *como una relación binaria hetereogénea* $\hat{\curlywedge} \subset (V_I \times V_B)$ *entre instancias y conexiones, obtenida mediante la composición de la reificación débil y la reificación de conexión. Es decir:*

$$\begin{array}{lcl} \hat{\curlywedge} & : & V_I \times V_B \\ \hat{\curlywedge} & \stackrel{\circ}{=} & \hat{\curlywedge} \circ \curlywedge \end{array} \tag{5.84}$$

De este modo, se recorre también la estructura de reificaciones, ya que en realidad es definida únicamente por las instancias; pero este recorrido termina ahora en una conexión. Esta relación expresa, en definitiva, cómo la influencia de un metacomponente puede llegar, mediante una cadena de reificaciones de longitud indefinida, incluso hasta una conexión.

Se define también, equivalentemente, la relación recíproca:

[18] Hay aquí cierto potencial para la confusión, aunque el significado es claro por contexto. La relación de meta-encadenamiento $\hat{\diamond}$ expresa un concepto reflexivo (*reflective*), en el sentido de Reflexión, y es ella misma reflexiva (*reflexive*) en el sentido usual en álgebra, dado que:

$$(i \curlyvee j) \rightarrow (j \curlywedge i) \rightarrow (i \diamond j) \wedge (j \diamond i) \rightarrow (i \hat{\diamond} j \hat{\diamond} i) \rightarrow (i \hat{\diamond} i)$$

Ya se ha dicho, usando ambos sentidos en una única frase, que el meta-encadenamiento es el cierre *reflexivo* transitivo de la relación de *reflexión* estricta. Recuérdese la nota al pie de la página 79.

Definición 5.65 (Reflexión Débil de Conexión) *Se define la* reflexión débil de conexión *como una relación binaria hetereogénea* $\mathfrak{Y} \subset (V_B \times V_I)$ *entre conexiones e instancias, obtenida mediante la composición de la reflexión de conexión y la reflexión débil. Es decir:*

$$\begin{array}{lcl} \mathfrak{Y} & : & V_B \times V_I \\ \mathfrak{Y} & \stackrel{\circ}{=} & \ddot{\Upsilon} \circ \check{\Upsilon} \end{array} \tag{5.85}$$

Nuevamente, puede demostrarse por construcción que ambas nociones definen relaciones inversas, de manera que $\mathfrak{Ꙗ} = (\mathfrak{Y})^{-1}$. Por tanto, no sólo cumplen su objetivo principal, que es generalizar las relaciones débiles al caso de conexiones, sino que incluso manifiestan las mismas propiedades. Debido a esto, todo lo siguiente será igualmente aplicable a reificaciones (resp. reflexiones) sobre ambos tipos de entidad.

Como acaba de verse, la secuencia definida por las relaciones débiles permite realizar un recorrido por la estructura de reificaciones; este recorrido constituye un *meta-camino*, que se define de manera análoga al camino –definición 5.8– de contenciones, cambiando únicamente la relación implicada. Es decir, se tendrá que:

Definición 5.66 (Meta-camino) *Dada la relación de reificación estricta* $\curlywedge$*, sea una secuencia de instancias* $m_k \in V_I$*, ordenada según un índice* $k \in \mathbb{N}^*$*, y elegida de modo que la instancia* k*-ésima reifica a la* $(k+1)$*-ésima* $(m_k \curlywedge m_{k+1})$*. Se define entonces el* meta-camino de longitud n*, desde una instancia* m_0 *hasta una instancia* m_n*, como una relación binaria* $\mathcal{K}_j \subset \curlywedge$*, formada por los* n *primeros pares ordenados de la forma* (m_k, m_{k+1})*, extraídos de esa secuencia.*

$$\mathcal{K}_j = \{(m_k, m_{k+1}) \ / \ (0 \leq k < n) \ \wedge \ (m_k \curlywedge m_{k+1})\} \tag{5.86}$$

Esta noción puede extenderse de la manera obvia para incluir a una conexión en sus extremo terminal. En cualquier caso, conocidos el concepto de (meta-)camino y su longitud, resulta interesante definir los conceptos siguientes, cuya influencia será determinante:

Definición 5.67 (Meta-altura Relativa) *Sea un par ordenado de instancias* (i, j) *contenido en la relación de reificación (o reflexión) débil, de modo que* $i \,\hat{\curlywedge}\, j$*. Se define la* meta-altura relativa *de* i *sobre* j*, y se expresa como* $\mathfrak{h}(i, j)$*, como la longitud de un meta-camino existente entre las instancias* i *y* j*. El concepto se extiende de la manera obvia al par formado por una instancia y una conexión, a través de la relación* $\mathfrak{Ꙗ}$*.*

Es importante resaltar aquí que, a priori, el meta-camino entre dos instancias relacionadas puede ser múltiple, esto es, que pueden establecerse, sin problemas, varias secuencias de reificaciones distintas entre los mismos elementos. Así, por ejemplo:

$$\left.\begin{array}{lcl} i \curlywedge k & \wedge & k \curlywedge j \\ i \curlywedge l & \wedge & l \curlywedge j \end{array}\right\} \Longrightarrow i \,\hat{\curlywedge}\, j$$

En principio, esto no supone ninguna dificultad; sin embargo, puede pensarse que de este modo podrían llegar a establecerse en el sistema estructuras inicialmente imprevistas. De hecho, si existieran simultáneamente varios meta-caminos de longitud diferente entre las mismas instancias, la meta-altura relativa $\mathfrak{h}(i, j)$ habría de definirse como un dato múltiple. Sin embargo,

esto no es cierto: como podrá verse más adelante (corolario 5.67), una situación de este tipo será explícitamente prohibida por la Primera Regla de Estratificación (Ley 5.2).

En cualquier caso, este detalle no afectará todavía a las definiciones siguientes, aunque sí puede considerarse en su interpretación.

Definición 5.68 (Meta-altura) *Por extensión, se define la* meta-altura (absoluta) *de una instancia dada, y se expresa como* $\mathfrak{H}(i)$*, como la máxima meta-altura relativa que alcanza esta instancia sobre cualquier otra instancia –o conexión–.*

$$\mathfrak{H}(i) \triangleq \max(\mathfrak{h}(i,j)) \quad \forall j \in V_I : i \,\hat{\curlywedge}\, j \tag{5.87}$$

Obsérvese que la meta-altura se define únicamente para instancias que participan de la relación de reificación ($\curlywedge$). Más todavía, solamente para aquéllas que actúan como *sujeto* de la misma, esto es, para las que reifican. En todas las demás, la meta-altura *no está definida*, al menos de momento. Esto se aplica incluso a las instancias que sólo actúan como *objeto* de la relación, esto es, las que son reificadas pero nunca reifican[19].

El concepto será extendido, abarcando a todos los elementos del esquema, a partir del corolario 5.80, como consecuencia de las dos Reglas de Estratificación (leyes 5.2 y 5.3).

Análogamente, se define el concepto dual, de modo que:

Definición 5.69 (Meta-profundidad Relativa) *Sea un par ordenado de instancias contenido en la relación de reflexión (o reificación) débil, de modo que* $i \,\check{\curlyvee}\, j$*. Se define la* meta-profundidad relativa *de* i *respecto de* j*, y se expresa como* $\mathfrak{d}(i,j)$*, como la longitud de un meta-camino existente entre las instancias* i *y* j*. Se define para conexiones de la manera análoga.*

Y, a su vez, se puede determinar el máximo de este último:

Definición 5.70 (Meta-profundidad) *Por extensión, se define la* meta-profundidad (absoluta) *de una instancia dada, y se expresa como* $\mathfrak{D}(i)$*, como la máxima meta-profundidad relativa que tiene esta instancia respecto de cualquier otra.*

$$\mathfrak{D}(i) \triangleq \max(\mathfrak{d}(i,j)) \quad \forall j \in V_I : i \,\check{\curlyvee}\, j \tag{5.88}$$

La meta-profundidad admite consideraciones idénticas a las ya expresadas para la meta-altura. Las dos nociones se reunen en un concepto estructural fundamental, presente en la teoría estándar de Reflexión Computacional [MJD96] (§4.3.2), a saber:

Definición 5.71 (Grado de Introspección) *Se define el* grado de introspección *de un esquema (del modelo* $\aleph_{\mathrm{M}}$*) como la máxima meta-altura (absoluta) del mismo, esto es, la mayor meta-altura alcanzada por una instancia del esquema.*

[19] Se podría pensar que en este caso la meta-altura debería ser *cero*, pero no es cierto. Podría tratarse simplemente de un elemento que se incorpora al esquema "a media altura", siendo reificado por una instancia de un meta-nivel mayor que uno. Suponer sin más que pertenece al nivel base iría contra las dos Reglas de Estratificación.

$$\Im \triangleq \max(\mathfrak{H}(i)) \quad \forall i \in V_I \tag{5.89}$$

Este es, por supuesto, un dato fundamental del sistema, y uno de los más importantes que se pueden extraer directamente del modelo. Es evidente, además, que el grado de introspección coincidirá igualmente con la máxima meta-profundidad del esquema, ya que esta noción y la meta-altura se definen sobre relaciones inversas; por tanto, el meta-camino (o meta-caminos) más largo será exactamente el mismo, y se cumple entonces que:

$$\Im = \max(\mathfrak{D}(i)) \quad \forall i \in V_I$$

Como se verá de inmediato, este dato ($\Im$) indicará el número de *meta-niveles* del esquema reflexivo. Siguiendo la denominación de Mitchell Wand y Daniel Friedman [FW84, WF88], esto puede verse también como el tamaño, altura o profundidad de la *torre reflexiva*, que en este caso es finita (§4.3.2). Con todo lo dicho, se puede definir ya el concepto estructural más importante en cualquier sistema reflexivo, es decir:

Definición 5.72 (Meta-Nivel) *Se define un* meta-nivel*, también denominado* meta-capa*,* meta-estrato*, o simplemente* nivel*, como el conjunto de todas las instancias de un esquema que tienen la misma meta-altura (absoluta). Se designa mediante la notación $\mathfrak{N}_H$, donde H es el número entero positivo que expresa la meta-altura mencionada. Por tanto:*

$$\mathfrak{N}_H \triangleq \{i \in V_I \,/\, \mathfrak{H}(i) = H\} \qquad \text{tal que } H \in \mathbb{N}^* \tag{5.90}$$

De esta definición se infieren, inmediatamente, los siguientes corolarios:

Corolario 5.73 *Un mismo elemento del esquema no puede pertenecer, simultáneamente, a dos* meta-niveles *diferentes, incluso sin tener en cuenta la Primera Regla de Estratificación.*

La demostración es trivial: dado que el meta-nivel ($\mathfrak{N}_H$) se define a partir de la meta-altura absoluta ($\mathfrak{H}(i)$), y que ésta se define a su vez como un valor *máximo* para cada elemento, es evidente que la asignación de niveles ha de ser única.

Corolario 5.74 *Todas las instancias de un nivel $\mathfrak{N}_x$ dado tienen también la misma meta-profundidad (absoluta), incluso sin tener en cuenta la Primera Regla de Estratificación.*

La demostración es obvia, dado que:

$$\forall i \in \mathfrak{N}_x : \mathfrak{D}(i) = \Im - x \tag{5.91}$$

Dado que la instancia i sólo puede pertenecer a un nivel, que éste especifica la máxima meta-altura (x) que alcanza i, y que el grado de introspección $\Im$ es un máximo para todo el esquema, se deduce que la máxima profundidad que puede alcanzar i es su distancia al nivel más alto en la torre reflexiva. Se puede objetar que no siempre tiene por qué existir un meta-camino desde este nivel hasta cada instancia del esquema, pero esta dificultad será soslayada tras el establecimiento de la Segunda Regla de Estratificación, concretamente en el corolario 5.80.

La definición de niveles crea por tanto una partición de V_I, de modo que los niveles van desde cero hasta $\Im$, sin que pueda excluirse ningún valor intermedio:

$$V_I = \bigcup_{i=0}^{\Im}(\mathfrak{N}_i) \;\; \text{tal que} \;\; \bigcap_{i=0}^{\Im}(\mathfrak{N}_i) = \varnothing \tag{5.92}$$

No obstante, esta ecuación no estará plenamente justificada hasta que se apliquen las dos Reglas de Estratificación, una vez que se desarrollen el corolario 5.80 y la definición 5.81.

Dada su importancia y frecuencia, resulta conveniente introducir una notación especial para expresar el meta-nivel asignado a una instancia o conexión determinada. Sea, por ejemplo, un nombre de instancia x, cuyo nivel asociado es a. Entonces:

$$x\rceil^a \;\; \equiv \;\; x \in \mathfrak{N}_a \;\; \equiv \;\; (\,\mathfrak{H}(x) = a\,)$$

Establecida la partición de niveles, se pueden proporcionar ya las dos definiciones siguientes, que se establecen de manera relativa al nivel actual. Entonces:

Definición 5.75 (Ananivel) *Dado un meta-nivel $\mathfrak{N}_H$, se define su* ananivel *como el nivel inmediatamente superior en la jerarquía* $[\mathfrak{N}_{(H+1)}]$.

Obviamente, este concepto está definido para todo nivel excepto el último ($\mathfrak{N}_\Im$).

Definición 5.76 (Catanivel) *Dado un meta-nivel $\mathfrak{N}_H$, se define su* catanivel *como el nivel inmediatamente inferior en la jerarquía* $[\mathfrak{N}_{(H-1)}]$.

De nuevo, este concepto está definido para todo nivel excepto el primero ($\mathfrak{N}_0$), que será denominado *nivel base*, y cuya especificación se difiere hasta la definición 5.81, debido a que depende conceptualmente de la Segunda Regla de Estratificación.

En este punto, precisamente, se está ya en condiciones de enunciar el siguiente principio fundamental, que determinará la consistencia interna de todo el modelo:

Ley 5.2 (Primera Regla de Estratificación) *Sólo se pueden establecer pares de reificación y reflexión estricta entre elementos de meta-niveles adyacentes.*

$$\begin{array}{rcrcl} m\rceil^i & \curlywedge & a\rceil^j & \longrightarrow & i = j+1 \\ a\rceil^i & \curlyvee & m\rceil^j & \longrightarrow & i = j-1 \end{array} \tag{5.93}$$

E igualmente, para conexiones:

$$\begin{array}{rcrcl} m\rceil^i & \curlywedge & b\rceil^j & \longrightarrow & i = j+1 \\ b\rceil^i & \curlyvee & m\rceil^j & \longrightarrow & i = j-1 \end{array}$$

La necesidad de la regla es evidente. Podría parecer, de hecho, que es innecesaria; pero en realidad no estaba implicada en las definiciones anteriores, que sólo establecían relaciones entre los meta-niveles en base a máximo. Esto es ante todo una condición sobre la reificación estricta (y de conexión); hasta el momento, ésta sólo estaba limitada por su asimetría e intransitividad.

La Regla implica también que se puede encontrar una conexión en cualquier meta-nivel. Aunque sólo puede aparecer como extremo terminal de una meta-camino, introducido mediante un paso de reificación específico ($\curlywedge$), nada impide, de hecho, que sea reificada por un metacomponente de un nivel n intermedio. En tal caso, será lógico concebirla como una conexión entre dos instancias del nivel $n-1$, si bien esto último sólo será obligatorio tras el establecimiento de la Segunda Regla de Estratificación (Ley 5.3).

Por tanto, el papel de la *estratificación* es en cierto modo análogo al del tipado, en el sentido de que asegura la consistencia de las cadenas de reificaciones. Evita explícitamente la aparición de situaciones conflictivas, como por ejemplo que una instancia m sea a la vez un metacomponente y un meta-metacomponente para una misma instancia a. Se da a este principio el nombre de estratificación por analogía con el concepto homónimo de la teoría axiomática de conjuntos conocida como NF –Nuevos Fundamentos–, que fue establecida por Willard V.O. Quine [For98] en 1937. En ambos casos, la idea de fondo es la misma: a cada elemento del sistema se le asigna un número de nivel según su relación con el resto. Si esta asignación es única, el sistema está estratificado, y es por tanto consistente.

A partir de la regla, se puede deducir, por tanto, el siguiente:

Corolario 5.77 *La meta-altura relativa* $\mathfrak{h}(i,j)$*, si está definida, es* única *para cualquier par de elementos* (i,j) *del esquema. Es decir, todo meta-camino existente entre ellos tendrá siempre la misma longitud. Esto mismo se aplica a la meta-profundidad relativa* $\mathfrak{d}(j,i)$*.*

Obsérvese que ni en la definición de reificación estricta ($\curlywedge$) ni de conexión ($\curlywedge$), ni por tanto en ninguna de las subsiguientes, se ha impuesto límite alguno al número de pares en los que puede participar una instancia dada, ni como sujeto –reificador– ni como objeto –reificado–. Esto quiere decir, utilizando la terminología habitual, que dichas relaciones, en particular, y todas las demás, en general, muestran una *cardinalidad* $(n:n)$, de muchos a muchos: en resumen, un metacomponente m puede reificar a varios avatares, y un avatar a puede reflejar a varios metacomponentes diferentes. Esto se mantiene, por supuesto, en todos los meta-niveles.

Precisamente el hecho de que esta cardinalidad sea múltiple es el que hace interesante poder obtener el conjunto de todos los avatares o metacomponentes de una instancia dada. Con este fin se definen, como se ha hecho para otras relaciones, las funciones siguientes:

Definición 5.78 (Función de Descenso) *Se define la* función de descenso*, y se representa como* $\mathfrak{R}_{\curlyvee}$*, como aquella que, para un metacomponente* m *dado, proporciona el conjunto de todas las instancias que reifica (estrictamente); es decir, el conjunto de todos sus avatares.*

$$\begin{array}{rcl} \mathfrak{R}_{\curlyvee} & : & V_I \longrightarrow \wp(V_I) \\ \mathfrak{R}_{\curlyvee}(i) & \doteq & \{j \in V_I \,/\, (j \curlyvee i) = (i \curlywedge j)\} \end{array} \tag{5.94}$$

A su vez, de manera análoga:

Definición 5.79 (Función de Ascenso) *Se define la* función de ascenso*, y se representa como* $\mathfrak{R}_{\curlywedge}$*, como aquella que, para un avatar* a *dado, proporciona el conjunto de todas las instancias que la reifican (estrictamente); es decir, el conjunto de todos sus metacomponentes.*

$$\begin{array}{rcl} \mathfrak{R}_{\curlywedge} & : & V_I \longrightarrow \wp(V_I) \\ \mathfrak{R}_{\curlywedge}(i) & \doteq & \{j \in V_I \,/\, (j \curlywedge i) = (i \curlyvee j)\} \end{array} \tag{5.95}$$

Del mismo modo, podrían hacerse definiciones análogas basándose en la relación de reificación de conexión, si bien son menos interesantes, ya que se establecería en un único nivel.

No existe una correlación directa –aunque sí indirecta– entre las funciones de ascenso y descenso, ya que en cada momento ambas proyectan fragmentos diferentes de la relación de reificación.

Una vez que se ha desarrollado por completo la estructura creada por la relación de reificación, que se había concebido como la tercera dimensión, se puede por fin plantear su combinación con las otras dos dimensiones, teniendo en cuenta especialmente el concepto de meta-nivel. La mezcla podría ser totalmente arbitraria; sin embargo, con el fin de lograr una estructura coherente, se regula mediante el siguiente principio general del modelo:

Ley 5.3 (Segunda Regla de Estratificación) *Sólo se pueden establecer relaciones de contención y ligadura entre elementos del mismo meta-nivel.*

$$\begin{array}{rclcrcl} a\rceil^i \supset b\rceil^j & \rightarrow & i = j & \Longrightarrow & a\rceil^i \circledcirc c\rceil^k & \rightarrow & i = k \\ a\rceil^i \hookrightarrow d\rceil^l & \rightarrow & i = l & \Longrightarrow & a\rceil^i \looparrowright e\rceil^n & \rightarrow & i = n \end{array} \tag{5.96}$$

Obviamente, aunque esta regla se define sólo sobre la contención directa y la ligadura, el propio desarrollo de éstas extiende su influencia, por sí sola, hasta la contención general simétrica, el encadenamiento, e incluso su combinación global, dada por la relación de vínculo débil. Por otra parte, dada la Regla de Ámbito (ley 5.1), hubiera sido suficiente definirla para la contención; la forma en que ambas se combinan implica ya que se cumple también para la ligadura. De este modo, se puede garantizar en un modelo reflexivo la consistencia de la reificación con las dos relaciones del fragmento básico del modelo ($\aleph_\Sigma$). Debido a esto, también se la puede denominar *regla de integración* o de *no ortogonalidad*.

Esto permite extender la definición de los conceptos de meta-altura y meta-nivel, abarcando ya a todos los elementos del sistema, que tengan *cualquier tipo de relación con alguien*, en alguna de las tres dimensiones. Esto es interesante, en particular, para abarcar a aquellos elementos para los que no se define ningún enlace de reificación explícita.

Corolario 5.80 (Meta-nivel Generalizado) *Toda instancia del esquema, reificada o no, pertenece al mismo meta-nivel que cualquier otra en la que esté contenida o con la que esté encadenada, y al mismo que cualquier elemento contenido en su interior.*

A su vez, toda conexión del esquema, reificada o no, pertenece al mismo meta-nivel que cualquiera de las instancias que conecta o encadena, y que aquélla que la contiene.

Este corolario implica pues la expansión del concepto de meta-altura absoluta (definición 5.68) a todo elemento del esquema, y por ende supone también la del concepto de meta-altura relativa. De hecho, este último podría ahora redefinirse como la diferencia entre las meta-alturas (absolutas) de dos elementos, estén o no reificados. Es decir:

$$\mathfrak{h}(i, j) = |\mathfrak{H}(i) - \mathfrak{H}(j)|$$

Lo mismo puede decirse para los dos tipos de meta-profundidad, teniendo en cuenta lo dicho en el corolario 5.74. Además, la difusión de estas medidas a toda entidad hace por fin posible definir otro de los conceptos fundamentales, esto es:

Definición 5.81 (Nivel Base) *Se define el* nivel base *o* nivel objeto *como el conjunto de todos aquellos elementos –instancias o conexiones– cuya meta-altura asociada es* cero*, una vez se han aplicado las dos Reglas de Estratificación.*

Por tanto, se asume que el nivel base ocupa la posición del meta-nivel cero. Por convención, se considera que todo elemento sin meta-altura asignada está igualmente incluido en éste.

Esta definición es necesaria para caracterizar el límite inferior de la torre de meta-niveles, de modo que se mantenga la consistencia con la Primera Regla de Estratificación, que lo hacía implícito en cierto modo. La convención pretende asegurar, además, que todo elemento del esquema tiene una meta-altura (es decir, un nivel) asociada.

La estratificación tenía ya un punto de partida, esto es, su límite superior, proporcionado por el grado de introspección ($\Im$), que es una constante del esquema. Por definición, éste indica la meta-altura asociada al meta-nivel más alto. A partir de éste, dada la Primera Regla, se puede calcular fácilmente dicha meta-altura para cada uno de los estratos restantes: basta con decrementar el valor en una unidad por cada reflexión. Debido a la Segunda Regla, todo elemento débilmente vinculado a otro incluido en la cadena de reificaciones, sea por contención general o por encadenamiento, pertenecerá también a alguno de dichos estratos.

De la propia definición se deduce, también, que la meta-altura del nivel más bajo ha de ser necesariamente igual a cero. Esto es así porque este nivel se obtiene a partir del superior, siguiendo la secuencia de reflexiones más larga; es decir, precisamente la que permite calcular el grado de introspección. Se infiere de la Primera Regla que todo avatar que participa como sujeto de la $\Im$-ésima reflexión a partir del último meta-nivel ha de tener una meta-altura nula. De nuevo por la Segunda Regla, todo elemento débilmente vinculado a (contenido en, continente de, o encadenado con) alguno de estos avatares tiene igualmente un nivel igual a cero.

Con todos estos elementos se construye, pues, el *nivel base*. Todo elemento sin asignación de nivel, en este punto, está en realidad *aislado*. No participa activamente del esquema, ya que no se conecta, incluye, ni reifica a ningún otro. Así pues, se le podría asociar sin consecuencias una meta-altura arbitraria; pero por simplicidad se prefiere incluirlo en el estrato más bajo.

Una vez que se tienen definidos ambos extremos, y que todo elemento del esquema tiene un nivel asociado, se pueden definir, por fin, los siguientes conceptos derivados:

Definición 5.82 (Nivel Meta) *Por abuso de lenguaje, se dice que un meta-nivel es el* nivel meta *para cualquier elemento del nivel inmediatamente inferior en la torre reflexiva.*

Por analogía, se podría admitir un uso similar para la expresión *nivel base de*; pero esto provocaría cierta confusión con *el* nivel base del esquema, ya que la inversión de adjetivos no parece aquí posible. Por tanto, se ha preferido no considerarla.

Definición 5.83 (Componente de Nivel Meta) *Se dice que una instancia* $\underline{m}$ *es un* componente de nivel meta *para un elemento* $\underline{a}$ *cualquiera, si pertenece a su* nivel meta*, esto es, al meta-nivel inmediatamente superior.*

Obsérvese que esta expresión incluye tanto a todos los metacomponentes de un avatar dado, como a todas las instancias vinculadas, en sentido general, con éste. Es decir, a todas las que tienen influencia, directa o indirecta, sobre él. Sin embargo, la definición abarca también a todas las instancias que influencian a otros elementos del mismo nivel que mi avatar. Ciertamente éstas también le afectan, de algún modo, pero resulta evidente que el concepto es todavía demasiado amplio. Por ello, se proporciona otro aún más acotado:

Definición 5.84 (Componente Meta) *Se dice que una instancia* $\underline{m}$ *es un* componente meta *o* niyatar[20] *de un avatar* $\underline{a}$ *cualquiera, si mantiene un vínculo general, sea por contención general o por encadenamiento, con algún* metacomponente *de* $\underline{a}$.

Se denomina $\mathfrak{S}(\underline{a}) \subset V_I$ *al conjunto de dichos* componentes meta.

$$\mathfrak{S}(\underline{a}) \doteq \begin{cases} \{y \in V_I \,:\, \neg(y \curlywedge \underline{a}) \wedge \exists \underline{m} \curlywedge \underline{a} \,/\, \underline{m} \Rrightarrow y\} & \text{si } \underline{a} \in V_I \\ \{y \in V_I \,:\, \neg(y \curlywedge \underline{a}) \wedge \exists \underline{m} \curlywedge \underline{a} \,/\, \underline{m} \Rrightarrow y\} & \text{si } \underline{a} \in V_B \end{cases} \tag{5.97}$$

Como puede verse, la definición excluye explícitamente la posibilidad de considerar que un metacomponente sea un componente meta. Por tanto, la definición de componentes meta abarca a todos los elementos con una "meta-influencia" directa sobre el avatar, pero excluye a los propios metacomponentes. Sin embargo, resultará interesante reunir todos estos elementos en un conjunto, de manera que se obtiene una de las definiciones estándar en la teoría de Reflexión Computacional [Yok92] (§4.3.4), a saber:

Definición 5.85 (Meta-Espacio) *Dado un avatar* $\underline{a}$*, se dice que su* meta-espacio *es el conjunto de todos los* metacomponentes *que lo reifican y de todos los* componentes meta *débilmente vinculados a éstos. Es decir, el conjunto todas las instancias del nivel superior que tienen influencia sobre* $\underline{a}$*. Se representa como* $\Lambda(\underline{a}) \subset V_I$.

$$\Lambda(\underline{a}) \doteq \begin{cases} \{x \in V_I \,:\, (x \curlywedge \underline{a}) \vee (x \in \mathfrak{S}(\underline{a}))\} & \text{si } \underline{a} \in V_I \\ \{x \in V_I \,:\, (x \curlywedge \underline{a}) \vee (x \in \mathfrak{S}(\underline{a}))\} & \text{si } \underline{a} \in V_B \end{cases} \tag{5.98}$$

Se denomina *paratar*[21] a todo elemento del meta-espacio de un avatar dado; es decir, es un término global para abarcar tanto a metacomponentes como a componentes meta.

Definición 5.86 (Meta-Espacio Conexo) *Dado un avatar* $\underline{a}$*, de define su* meta-espacio conexo *como el conjunto de todas las instancias de su meta-espacio, más todas las conexiones existentes entre ellas. Se representa como* $\Lambda'(\underline{a}) \subset (V_I \cup V_B)$.

$$\Lambda'(\underline{a}) \doteq \Big\{ \{x \in V_I \,:\, x \in \Lambda(\underline{a})\} \,\cup\, \{y \in V_B \,:\, y \in \circledast(z) \,/\, z \in \Lambda(\underline{a})\} \Big\} \tag{5.99}$$

En realidad, el auténtico *meta-espacio* es éste. Puede verse que el definido por $\Lambda(a)$ proporciona solamente un conjunto de instancias dispersas, mientras que éste extrae un fragmento de arquitectura completo, en el que la influencia de los distintos elementos se combina mediante criterios de composición e interacción.

Sin embargo, todas estas definiciones se limitan al ananivel, cuando es obvio que la influencia sigue expandiéndose a lo largo de los distintos meta-niveles. Todas estas nociones pueden extenderse, por tanto, a un número arbitrario de niveles, de modo que:

[20] Del sánscrito *niya-tar*, que significa literalmente *el que controla* ($\kappa\iota\beta\varepsilon\rho\nu o\varsigma$).

[21] Del sánscrito *para-tar*, que significa literalmente *el que está encima*.

Definición 5.87 (Meta-metacomponente) *Se denomina* meta-metacomponente *de un avatar* $\underline{a}$ *dado a cualquier metacomponente de un metacomponente de* $\underline{a}$*. De forma análoga, se puede añadir cualquier número de prefijos (o sufijos)* meta*, según sea necesario.*

En resumen, el término se generaliza para abarcar a cualquier número de reificaciones, a razón de un paso en la jerarquía por cada uso de la palabra *meta*.

No se proporciona ningún término general para abarcar a todos las instancias de la cadena de reificaciones, para un avatar dado. Sin embargo, sí se proporciona su conjunto, que estaba ya implícito en la definición de reificación débil:

Definición 5.88 (Función de Ascenso Débil) *Se define la* función de ascenso débil*, y se representa como* $\mathfrak{M}(\underline{a})$*, como aquella que, para un avatar* $\underline{a}$ *dado, proporciona el conjunto de todas las instancias que la reifican (débilmente).*

$$\mathfrak{M}(\underline{a})(\subset V_I) \triangleq \int \{x \in V_I \,:\, x \,\hat{\curlywedge}\, \underline{a}\} \quad \text{si } \underline{a} \in V_I \\ \in V_B \tag{5.100}$$

Esta función es la expansión natural de la función de ascenso $\mathfrak{R}_{\curlywedge}$, definida de manera análoga. De manera similar, podría establecerse una función de descenso débil, análoga a su vez a $\mathfrak{R}_{\curlyvee}$. No se ha considerado necesario incluirla aquí, dado que su definición resulta evidente.

Si lo que se pretende es determinar *toda* la influencia reflexiva, a cualquier nivel, sobre un avatar $\underline{a}$ concreto, sin embargo, la noción adecuada no es ésta. En su lugar se utiliza la expansión del concepto de meta-espacio, incluyendo no sólo a los metacomponentes, sino también a los componentes meta a cualquier nivel.

Definición 5.89 (Meta-Espacio Extendido) *Se define el* meta-espacio extendido *correspondiente a un avatar* $\underline{a}$ *dado, como la unión del meta-espacio conexo de* $\underline{a}$*, y los* meta-espacios extendidos *de cada uno de los elementos de éste. Se representa como* $\Delta(a) \subset (V_I \cup V_B)$*.*

Esta definición es, por supuesto, recursiva, y mucho más amplia de lo que pudiera pensarse en un principio. No sólo abarca, como podría parecer, a todos los elementos de $\mathfrak{M}(\underline{a})$, los componentes meta débilmente vinculados con ellos y sus conexiones; sino que se expande en cada una de sus ramas. Así, se incluyen también los metacomponentes que reifican (débilmente) a dichos componentes meta y conexiones, los componentes meta vinculados a éstos, y las conexiones que los encadenan; de nuevo, los metacomponentes que reifican a todos éstos, y así sucesivamente. Afortunadamente, el proceso es finito, dado que la torre reflexiva solamente tiene $\Im$ niveles. En principio, la mayor expansión se produce en *anchura*, aunque en el caso más normal los meta-espacios obtenidos de cada una de las ramas se solaparán con facilidad.

Por tanto, se define el meta-espacio extendido como:

$$\Delta(\underline{a}) \triangleq \Lambda'(\underline{a}) \cup \bigcup_{x}^{\Lambda'(\underline{a})} (\Delta(x)) \tag{5.101}$$

La estructura de esta definición es tal vez menos habitual, pero es igualmente válida. No obstante, puede hacerse una formulación equivalente, pero menos conflictiva, partiendo de la semejanza entre la definición de meta-espacio y una función. Así pues, sea:

$$\begin{array}{rcccl} \Lambda' & : & (V_I \cup V_B) & \longrightarrow & \wp(V_I \cup V_B) \\ & & \underline{a} & \longrightarrow & \Lambda'(\underline{a}) \end{array}$$

Se (re)define como una función (o relación), entonces:

$$\Delta \;\mathring{=}\; (\Lambda')^+ \;=\; \bigcup_{i=1}^{\Im}((\Lambda')^i) \tag{5.102}$$

Es decir, el meta-espacio extendido equivale al cierre transitivo no reflexivo del meta-espacio conexo. Es necesario partir directamente de la forma conexa, ya que las conexiones también pueden ser reificadas; cualquier expansión del meta-espacio que no las incluyera –y a partir de ellas, sus respectivos meta-espacios (extendidos)– estaría incompleta.

Con esto concluye la exploración de la meta-estructura desplegada simplemente a partir de la relación de *reificación*, en el sentido aquí descrito. Como puede verse, es posible describir esquemas de una gran complejidad; sin embargo, esto no significa que esta complejidad sea gratuita. En circunstancias normales, nunca habrá más de tres meta-niveles, y cada reificación tendrá una justificación detallada. No obstante, $\aleph_{\mathrm{M}}$ pretende ser un modelo general, y como tal se ha descrito para ser capaz de abarcar a cualquier esquema reflexivo concebible.

5.5.3 Modelo Marmol Restringido ($\aleph_{\Pi}$)

Se define el modelo restringido $\aleph_{\Pi}$ como una extensión del modelo general $\aleph_{\mathrm{M}}$ ya descrito, de modo que se cumple que:

$$\aleph_{\Pi} = \langle V, S_{\Pi} \rangle \;/\; \aleph_{\mathrm{M}} = \langle V, S_M \rangle \;\wedge\; S_M \subset S_{\Pi}$$

Su definición será mucho más sencilla que en los dos casos anteriores, ya que las modificaciones respecto a $\aleph_{\mathrm{M}}$ serán mínimas. La estructura del conjunto de nombres V no se modifica en absoluto, y la nueva signatura S_{Π} tan sólo incorpora una relación esencialmente nueva, que impone una restricción extra sobre la estructura del sistema, obligando a la definición de cierto número de metacomponentes. En resumen, se pretende *colapsar* (§4.2) parcialmente las relaciones de tipado y reificación, de modo que a toda instancia le corresponda al menos un rohatar, relacionado de algún modo con su tipo.

Puede parecer que con esto se pierde generalidad: en parte es cierto, pero lo que en realidad se está haciendo es integrar de manera más estrecha la estructura básica ($\aleph_{\Sigma}$) con las extensiones de $\aleph_{\mathrm{M}}$. Ninguna de las dos relaciones implicadas desaparece: lo único que se hace es establecer un vínculo entre ellas. De este modo, se facilitará directamente la expresión de Reflexión Estructural (§4.4.1), complementando el enfoque de $\aleph_{\mathrm{M}}$, que favorecía a su vez la definición de Reflexión de Comportamiento.

Como ya se ha dicho, la signatura S_{Π} sólo define una nueva relación, junto a una serie de conceptos derivados. Sea entonces:

Definición 5.90 (Metalepsis) *Se define la* metalepsis *como una relación binaria hetereogénea* $\mathcal{M} \subset (V_T \times V_I)$*, equiparable a una función total inyectiva, establecida de modo que a todo*

arquetipo $\underline{t}$ *del sistema se le asigna una instancia* $\underline{m}$*, tal que toda instancia o conexión tipada por* $\underline{t}$ *es también reificada por* $\underline{m}$*. Es decir:*

$$\mathcal{M}: \left.\begin{array}{rcl} V_T & \rightarrow & V_I \\ \underline{t} & \rightarrow & \underline{m} \end{array}\right\} \wedge \left\{\begin{array}{lcl} \forall j \in V_I & : & \mathcal{M}(\mathcal{T}_I(k)) \curlywedge j \\ \forall k \in V_B & : & \mathcal{M}(\mathcal{T}_B(k)) \curlywedge k \end{array}\right. \tag{5.103}$$

Es decir, cualquier instancia o conexión k tendrá un tipo, tal que $(k :: \underline{t})$. A este tipo se le asigna por metalepsis[22] una instancia $\underline{i} = \mathcal{M}(\underline{t})$, de modo que esta última mantiene una relación de reificación con la primera, sea un componente $(\underline{i} \curlywedge k)$ o una conexión $(\underline{i} \curlywedge k)$.

Por tanto, la metalepsis declara la existencia un metacomponente, al menos, por cada arquetipo del sistema, de modo que toda instancia del arquetipo es también un avatar del metacomponente. En definitiva, el tipado $\mathcal{T}$ y la reificación $(\curlywedge \cup \curlywedge)$ *se colapsan* parcialmente. Téngase en cuenta que, al ser la metalepsis inyectiva, a todo tipo le corresponde una reificación, pero el inverso no es cierto. Por tanto, puede haber un número indeterminado de reificaciones totalmente independientes del sistema de tipos.

La relación se define en este orden para poder expresar mejor la restricción que da lugar al modelo $\aleph_\Pi$. Sin embargo, una vez que la estructura está establecida, y dada la naturaleza inyectiva de la metalepsis, lo realmente interesante será saber si un metacomponente dado corresponde a un arquetipo, o por el contrario es totalmente independiente. Por tanto, se manejará con frecuencia la relación inversa a la de metalepsis, que se define como:

$$\left.\begin{array}{lcl} \mathfrak{B} & : & V_I \rightarrow V_T \\ \mathfrak{B} & = & \mathcal{M}^{-1} \end{array}\right\} \Longrightarrow \mathfrak{B}(\underline{m}) = \underline{t} \tag{5.104}$$

Obviamente, se trata de una función parcial que, dado un metacomponente, determina si éste corresponde, por metalepsis, a un arquetipo. En tal caso, se dice que el arquetipo es el **brhátar**[23] del metacomponente. Por el contrario, cuando un metacomponente no tiene brhátar, se dice que es *independiente* o *no dual*. El concepto de brhátar será ampliamente utilizado en la semántica de $\mathcal{P}i\mathcal{L}ar$, como puede verse, por ejemplo, en la sección 7.2.5.

Tan sólo existe un problema: dado que toda instancia está tipada y que a todo tipo le corresponde un metacomponente, que también es una instancia, puede verse que la metalepsis obliga a la definición de una *torre reflexiva infinita*, reproduciendo el clásico problema de Wand y Friedman. La solución será la misma que se emplea en la mayoría de los sistemas de reflexión estructural: se considera, al llegar a ciertos tipos, que el esquema de reificación se recircula, de modo que un *arquetipo se refleja a sí mismo.*

Por supuesto, esto no quiere decir que $\underline{t} \curlyvee \underline{t}$, algo explícitamente prohibido por el modelo. Lo que ocurre es que el comportamiento y estructura del arquetipo $\underline{t}$, que en el caso típico será un tipo unitario (*singleton*), no necesitarán ser descritos por un nuevo metacomponente, puesto que se considera que la (única) *instancia* de $\underline{t}$ ya cumple esa función.

Obviamente, esto va en contra de la definición de metalepsis. Si se asume que $(k :: \underline{t})$ y se considera que $\mathcal{M}(\underline{t}) = k$, se llegaría de nuevo a una relación ilegítima $k \curlywedge k$. Por tanto, en estos

[22] El significado literal de este término se refiere al tipo de metonimia que intercambia el antecedente y el consecuente de una oración. La palabra se utiliza aquí como una metáfora, ya que el objetivo de la relación $\mathcal{M}$ es poder utilizar el metacomponente como si fuera el arquetipo, esto es, su brhátar.

[23] Literalmente del sánscrito, donde significa *hermano*. Existen términos etimológicamente más ajustados, tales como *prânamâkosha*, pero se ha considerado que son excesivamente complejos, y menos claros.

casos se asume que la metalepsis *no asocia ningún componente a ese arquetipo*. Esto significa que, contrariamente a lo que se había dicho, $\mathcal{M}$ ya no será una función total sobre V_T, sino parcial. Todos los nombres de tipo excluidos de la relación por esta causa recibirán el nombre de **arquetipos anudados**, y se agruparán en el conjunto $\circlearrowright[V_T]$. Se tiene, por tanto:

$$\mathcal{M} : (V_T - \circlearrowright[V_T]) \longrightarrow V_I$$

De modo que la metalepsis se define, finalmente, sobre el conjunto de los nombres de tipo *no* anudados, recuperando de este modo, en parte, su carácter de función total.

En resumen, $\aleph_{\Pi}$ es prácticamente idéntico a $\aleph_{\mathrm{M}}$, y ambos tienen la misma expresividad. Su única diferencia es la presencia de la restricción que obliga a la existencia de un metacomponente para cada arquetipo, debido a la metalepsis. Por tanto, $\aleph_{\mathrm{M}}$ es más general, en el sentido de que permite definir meta-estructuras que no respeten esta relación; pero todo comportamiento que pueda definirse sobre un modelo, puede igualmente definirse sobre el otro.

El colapso (parcial) de las relaciones de tipado y reificación no aumenta, por tanto, la potencia expresiva del modelo, pero hace mucho más sencillo expresar mecanismos de Reflexión Estructural (§4.4.1), obviamente la más útil en Arquitectura de Software. Por ello, $\aleph_{\Pi}$ será la versión de Marmol implícitamente utilizada en $\mathcal{P}i\mathcal{L}ar$.

5.6 Conclusión

A lo largo de este Capítulo, se han presentado, principalmente, tres resultados, todos ellos dirigidos a la combinación de Reflexión con Arquitectura de Software. En primer lugar, se han descrito los principales modelos de dinamismo arquitectónico en términos reflexivos, justificando así la motivación principal de esta combinación. En segundo lugar, se han considerado los principales conceptos reflexivos en el contexto de la Arquitectura de Software; se han seleccionado aquéllos que se consideraban especialmente relevantes, y se ha elaborado con ellos un modelo informal, concebido como una extensión para cualquier esquema de descripción preexistente.

En tercer lugar, finalmente, se ha planteado la definición formal de este modelo reflexivo, comenzando en este caso por la teoría más básica. El lenguaje utilizado ha sido, fundamentalmente, la teoría de relaciones; aunque no es la primera vez que ésta se utiliza para la descripción arquitectónica [Hol98], sí es la primera en la que se hace con este grado de detalles. Además, es también la primera vez en la que las nociones de reflexión, y en especial las derivadas del concepto de reificación, se plantean desde este punto de vista.

Aunque se ha intentado hacer un desarrollo lo suficientemente completo, resulta evidente que estos modelos están lejos de haberse explotado –matemáticamente– en su totalidad. El hecho de haber utilizado la teoría de conjuntos y de relaciones como base formal tiene varias ventajas obvias, aunque no está exento de inconvenientes. Entre las ventajas cabe destacar la naturalidad y difusión de las nociones básicas, incluso dentro de un entorno netamente informático; su facilidad de integración con otros métodos formales, habituales en el contexto de la Ingeniería de Software, muchos de los cuales tienen también una base algebraica; o su enfoque fundamentalmente modular, que resulta análogo al de la propia Arquitectura de Software. Entre sus inconvenientes están la inevitable complejidad del modelo final obtenido, y el hecho de que, a pesar de su potencia, los conjuntos y relaciones resultan en ocasiones demasiado básicos.

A lo largo de la descripción del modelo formal se han planteado ocasionalmente correspondencias entre estructuras complejas, usualmente descritas en términos de biyecciones. Es evidente que en este caso, el haber utilizado teoría de relaciones como formalismo base plantea, de algún modo, una limitación. Sin duda, un enfoque basado en teoría de categorías, como el planteado por Fiadeiro y Maibaum [FM96], que fue desarrollado después por Wermelinger [Wer99], resulta mucho más prometedor en este aspecto, aunque ciertamente más complejo. En este sentido, la consideración de Marmol en el contexto de la teoría de categorías, así como su comparación explícita con el modelo arquitectónico de Wermelinger, se plantea como una de las líneas de trabajo futuro con mayores posibilidades.

Finalmente, se debe mencionar que la elección de la noción de *nombre* como único concepto atómico, inspirada en buena medida por el cálculo-π, tiene varias consecuencias notables. Sin duda, la más interesante es la forma en que se facilita la descripción de la reificación como un concepto estructural, que en esta representación puede ya comenzar a concebirse como asimilable a la noción de *superposición*, una idea que será brevemente desarrollada en capítulos posteriores (§7.2.5).

BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Capítulo 6

El Lenguaje PiLar: Sintaxis

La Sabiduría ha edificado su propia casa,
ha tallado sus Siete Pilares.
Proverbios 9, 1

Sapientia Ædificavit Sibi Domum
Lema de la Universidad de Valladolid

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

6.1 Introducción

El modelo Marmol, descrito en el Capítulo anterior, proporciona una referencia común para la introducción de Reflexión en Arquitectura de Software. Sin embargo, por sí solo no es suficiente, ya que únicamente define un marco *estructural* para la especificación de arquitecturas de software. Para poder llegar a hacer una verdadera descripción arquitectónica, es necesario aplicar sus conceptos a un lenguaje, que proporcione las construcciones sintácticas necesarias. En principio, cualquier Adl convencional podría ser adaptado para este fin, ya que Marmol se ha definido como un modelo general, y utilizando un formalismo absolutamente neutral.

No obstante, cada Adl particular nace con un objetivo específico, y está concebido desde un punto de vista particular. Esto hace que, incluso cuando la adaptación a Marmol es factible, el lenguaje resultante estará generalmente condicionado por su propio punto de partida. En esta tesis se desea, por el contrario, tener una perspectiva independiente de lo que significa tener un Adl reflexivo, sin depender de ningún otro aspecto. Por ello, se ha decidido diseñar un lenguaje completo y específico, cuya estructura esté influenciada ante todo por su enfoque reflexivo y, por supuesto, por su orientación final a la especificación de dinamismo.

Por supuesto, la aplicación de Marmol a un Adl no es totalmente lineal; intentar hacerlo de ese modo hubiera llevado a la definición de un lenguaje rígido y poco versátil. Por el contrario, se ha intentado diseñar un lenguaje lo más general posible, que tenga en cuenta las experiencias previas de otros Adls, tomando las características más interesantes de cada uno de ellos. Por tanto, el diseño del lenguaje no es una mera adaptación sintáctica de Marmol, sino que por el contrario es también una síntesis –parcial, por supuesto– de las estructuras y construcciones ya existentes en otros lenguajes, a las que incluso se añade alguna nueva aportación. La reflexión se incorpora después, de forma suave, y sin realizar una ruptura con la sintaxis definida.

El lenguaje desarrollado recibe el nombre de $\mathcal{P}i\mathcal{L}ar$, que aparte de sus significados obvios representa a un acrónimo recursivo –"Pi*Lar es un* L*enguaje de descripción* Ar*quitectónica"*–, tanto en español como en inglés –"P*iLar* I*s a* L*anguage for* Ar*chitectural description"*–, y hace referencia, además, al hecho de que su semántica será expresada en cálculo-π (§7). Como ya se ha dicho, $\mathcal{P}i\mathcal{L}ar$ está basado en Marmol, aunque no de manera rígida; concretamente, realiza una implementación del modelo $\aleph_{\Pi}$, es decir, colapsa la reificación y el tipado.

El lenguaje ha sido diseñado con el objetivo de ser útil, incluso independientemente de su enfoque reflexivo. No obstante, no deja de ser un lenguaje experimental, cuya verdadera aplicabilidad sólo será demostrada tras un uso prolongado. En cualquier caso, la sintaxis definida es apta para un uso genérico, tal como podrá comprobarse en capítulos sucesivos; y en cualquier caso resulta suficiente para los propósitos definidos en esta tesis.

En este capítulo se describe con detalle la estructura y sintaxis del lenguaje $\mathcal{P}i\mathcal{L}ar$. No se pretende proporcionar aquí una descripción formal completa del lenguaje; ésta se proporciona en el Apéndice A, donde se incluye la correspondiente gramática. Lo que se hace, más bien, es una introducción gradual al mismo, en el que la estructura de la sintaxis se va desarrollando poco a poco, de modo que cada nuevo elemento se construye sobre la base de los conceptos ya conocidos. Pero no se revisa aquí, sin embargo, ningún ejemplo completo, que dé una perspectiva global del lenguaje: este tipo de tarea se difiere hasta el Capítulo 8.

Dado que $\mathcal{P}i\mathcal{L}ar$ se plantea como un Adl reflexivo, se asume que para su correcto uso es necesario comprender al menos los aspectos más generales del modelo Marmol, descrito en el Capítulo 5. Sin embargo, no se ha de pensar por ello que el lenguaje depende de éste; por el contrario, los detalles más complejos quedan relegados al plano semántico, y en líneas generales resultan bastante independientes. No se exige, pues, asimilar la totalidad del modelo formal $\aleph_{\Pi}$, o de la perspectiva general incluida en el apartado 5.3. En realidad, tan sólo es importante entender qué tipo de nociones se están manejando, y en particular aquéllas que el modelo aporta a la teoría habitual de descripción arquitectónica –es decir: qué conceptos introduce el modelo reflexivo $\aleph_{\mathrm{M}}$ que no estaban ya presentes en el modelo canónico $\aleph_{\Sigma}$–. En resumen, de lo que se trata es de comprender conceptos tales como meta-componente, conexión causal o avatar, para así poder asimilar la estructura de niveles que $\mathcal{P}i\mathcal{L}ar$ define implícitamente.

Por otra parte, $\mathcal{P}i\mathcal{L}ar$ se define también como un Adl del tipo proceso-algebraico, lo que le sitúa en la tradición de algunos de los Adl mejor conocidos y más avanzados, como por ejemplo Wright [All97], *δarwin* [MDEK95], Leda [CPT99] o Padl [BCD01]. Por tanto, cierta familiaridad previa con cualquier álgebra de procesos, sea del tipo Ccs (en especial), Csp, Acp o incluso Lotos[1], resultará muy conveniente para una comprensión plena del sentido en que se especifican las acciones de las reglas $\mathcal{P}i\mathcal{L}ar$. No obstante, este requisito no es tan restrictivo como pudiera parecer, ya que en última instancia estas álgebras sólo agrupan en una notación compacta conceptos básicos de concurrencia, y pueden describirse de manera aproximada en un contexto informático general como un lenguaje de comunicación síncrona basada en paso de mensajes entre pares. En definitiva, se supone que el lenguaje es lo suficientemente simple para poder ser rápidamente asumido por cualquier arquitecto de software que conozca nociones tales como sincronización o paralelismo. De todos modos, con el fin de facilitar el uso, la sintaxis final

[1] O incluso lenguajes de programación formales o semiformales con sintaxis similar, tales como Occam, CML, Oz, Facile, Pict, Join, TyCO o Piccola, ligados en su mayoría a los cálculos del Apéndice B.

intenta evitar las notaciones más algebraicas en favor de un formato más familiar, cercano al de un lenguaje de programación. No obstante, esto último es algo que tampoco pretende ser en ningún caso; un Adl es un tipo de lenguaje mucho más específico.

6.2 Visión General del Lenguaje

Posiblemente la característica más llamativa del lenguaje, en un primer vistazo, es el hecho de que está dividido de manera implícita en dos sublenguajes relacionados, pero claramente diferenciados, a los que se designará como *Lenguaje Estructural* y *Lenguaje Dinámico*. Ha de quedar claro desde el principio, sin embargo, que se trata de un *único* lenguaje, con nociones comunes y notaciones consistentes para ellas. La citada división sólo pretende facilitar la descripción, comprensión y uso de dos aspectos distintos, pero íntimamente ligados, de una arquitectura: su estructura básica esencial, y su evolución dinámica en el tiempo.

El primero proporciona el armazón estático de una descripción arquitectónica a través de la definición declarativa de componentes; el segundo especifica la evolución dinámica del sistema mediante un conjunto de restricciones o reglas de carácter imperativo, que se asocian con dichos componentes. No obstante, como se ha dicho, el lenguaje es único: su sintaxis ha sido diseñada para ser mutuamente consistente, y se diferencian sobre todo en la orientación –declarativa o imperativa– de cada uno de ellos, que se adapta a sus distintos cometidos.

En este punto, $\mathcal{P}i\mathcal{L}ar$ sigue la tradición de los demás Adls, por una parte, y se aleja de ella, por otra. En efecto, los Adls convencionales hacen, o bien una descripción puramente estructural, como Unicon [SDK+95b], o bien especifican *además* el comportamiento mediante un lenguaje auxiliar, típicamente formal, como ocurre en Wright [All97] o Rapide [LV95]. En ambos casos, la forma es similar a la del Lenguaje Estructural de $\mathcal{P}i\mathcal{L}ar$. Sin embargo, los Adls dinámicos suelen mezclar los aspectos de evolución con los de estructura, como Darwin [MK96] o Manifold [PA01], o mantenerlos completamente separados, como C2sadel [OMT98]. En esto nuestro Lenguaje Dinámico supone cierta ruptura, pues ni se mezcla ni se separa por completo. Existe, no obstante, un caso intermedio, el de LEDA [CPT99], que dispone también de un lenguaje *de comportamiento*, que es básicamente cálculo-π, y describe el dinamismo dentro de éste. El caso de $\mathcal{P}i\mathcal{L}ar$ es análogo, pero la orientación del Lenguaje Dinámico está en líneas generales a un nivel más arquitectónico. De hecho, en cierto modo llega a incluir al Estructural, si bien con distinto enfoque y menor naturalidad. Esto permite, a un tiempo, mantener la coherencia entre los dos sublenguajes sin perder la *separación de intereses*, que es uno de los conceptos que de los que nace la propia Arquitectura de Software.

Por otra parte, como ya se ha dicho, $\mathcal{P}i\mathcal{L}ar$ se enmarca dentro de la más clásica tradición de la descripción arquitectónica: la de los Adls originados o fundamentados en álgebras de procesos. Esto no sólo abarca a los directamente implicados, como los ya citados Wright, $\delta arwin$, Leda o Padl, sino también a lenguajes estrechamente relacionados, como Aesop, Acme (Armani), Koala o Unicon; y otros no tan cercanos, pero igualmente fundamentados en teoría de la concurrencia, como Rapide o C2sadel. La otra tradición más importante –la de los lenguajes basados de algún modo en el concepto de gramáticas de grafos– no ha alcanzado aún un grado de difusión o de interrelación semejante; y en el resto de los casos se trata más bien de heurísticas, adaptaciones de otros contextos –algunas de ellas notablemente interesantes– o esfuerzos aislados.

En este sentido, se debe enfatizar que $\mathcal{P}i\mathcal{L}ar$ se diseñó con el ánimo de sintetizar –en la medida de lo posible– los logros de los sistemas ya existentes, y aumentar su potencia expresiva con las menores adiciones necesarias. No se pretende, pues, reinventar los conceptos clásicos, sino integrarlos; y en ese sentido ha de entenderse la sintaxis del Lenguaje Estructural, al igual que las definiciones del modelo canónico $\aleph_\Sigma$ (§5.5.1).

Por tanto, en $\mathcal{P}i\mathcal{L}ar$ se incluye la gran mayoría de las nociones tópicas de la descripción arquitectónica, dentro de la tradición mencionada, con su significado habitual u otro muy similar. Por supuesto, la correspondencia concreta con cada uno de los ADLs citados no es siempre directa, pero desde luego manejan conceptos comparables. Lo único novedoso en este punto, aparte de la sintaxis específica, es precisamente la selección y unificación de dichos conceptos, así como los detalles de su interpretación semántica, que se difieren hasta el Capítulo 7. Sin embargo, no se ha de pensar que esta tarea es trivial: la inclusión de ciertas construcciones lingüísticas, presentes en algunos de estos lenguajes, tales como iteración, recursión, parametrización o estructuras alternativas, es notoriamente útil, pero ha sido también muy discutida en varias ocasiones, y no es, ni mucho menos, aceptada por todos los ADLs citados. En general, se ha intentado no renunciar a (casi) ninguna idea cuya utilidad sea manifiesta.

Por otra parte, como también se ha dicho, no se debe olvidar que $\mathcal{P}i\mathcal{L}ar$ es todavía un lenguaje experimental, cuyo objetivo principal es permitir la descripción y manipulación de esquemas arquitectónicos reflexivos; sin embargo, se ha intentado desde el principio que el lenguaje tenga todas las características deseables en un ADL de uso general, sin que sus aspectos más específicos comprometan su aplicabilidad. Se puede garantizar, desde luego, que la descripción de los ejemplos estáticos clásicos –véase, por ejemplo, la sección 8.2– no resulta más difícil que en otros ADLs; de hecho, a menudo ocurre incluso al contrario.

6.2.1 El Concepto de Componente

En la sección 2.4.2 ya se ha proporcionado una perspectiva global de lo que significa la noción de componente en la Arquitectura de Software. Por supuesto, los componentes de $\mathcal{P}i\mathcal{L}ar$ siguen esta misma idea. No obstante, conviene examinar brevemente el enfoque particular que se utiliza en este caso, delimitando los aspectos concretos seguidos en su definición.

Como ya es sabido, el concepto de componente –instancia de componente– toma como básica una de las abstracciones fundamentales más utilizadas en Informática: el *encapsulamiento* u *ocultación de información*. Por ésta y otras razones, no resulta extraño que los trabajos primigenios sobre esta abstracción de David Lorge Parnas [Par72] hayan sido citados, con cierta frecuencia, como la primera gran inspiración de la Arquitectura de Software.

Frente al encapsulamiento, otras abstracciones fundamentales como la composición y la interacción, aun siendo todavía muy importantes, pasan a un segundo plano. Otros conceptos, como la genericidad o la herencia, tradicionalmente vinculados al modelo de orientación al objeto, sólo aparecen ocasionalmente, y son considerados por una minoría de lenguajes.

En cambio, el tipado sigue siendo una abstracción fundamental: de hecho, Medvidovic [MT00] sitúa a la especificación de tipos como una de las características que definen a un ADL. La distinción entre *instancias* de componente –a menudo denominadas simplemente *componentes*– y *tipos* de componente –aquí abreviado como *arquetipos*– es respetada por la práctica totalidad de los ADLs existentes; en el caso particular de $\mathcal{P}i\mathcal{L}ar$, dada su vinculación con el modelo $\aleph_\Pi$,

resulta además totalmente imprescindible.

Dado que el componente se define en primer lugar como una unidad de encapsulamiento, se ha de asumir que se comporta ante todo como una *caja negra*. Esto afecta sobre todo al modo en que se relaciona su *especificación* dentro de una arquitectura con su verdadera naturaleza o *implementación* en un sistema real. Está claro que ambos han de manifestar el mismo comportamiento externo; por tanto, y dado el encapsulamiento, la única equivalencia que se puede considerar es una *equivalencia observacional*. Ésto, en el contexto de un lenguaje algebraico como $\mathcal{P}i\mathcal{L}ar$, con una semántica basada en cálculo-π, conduce directamente hacia la noción de *bisimulación (débil)*, como relación de equivalencia por observación [Mil89, Mil99].

Por tanto, en $\mathcal{P}i\mathcal{L}ar$ no se pretende describir en ningún momento la *funcionalidad* de un componente, sino sólo su comportamiento externo. Un componente cualquiera –sea implementado, o simplemente especificado mediante una definición formal– podrá ser utilizado en un lugar concreto solamente si es *bisimilar* con la descripción $\mathcal{P}i\mathcal{L}ar$ del componente allí ubicado. Es decir, un artefacto dado podrá situarse en una arquitectura si su comportamiento externo es equivalente –bisimilar– al esperado. Por tanto, en el lenguaje, la descripción de un componente sólo proporciona una definición de este comportamiento, y en particular de su interacción con el resto de la arquitectura. Esta definición es restrictiva, pero más amplia incluso que la de otros lenguajes. En muchos sentidos, su filosofía es similar a la de las *interfaces* de Rapide, y resulta prácticamente idéntica a la utilizada en Padl, aunque a otro nivel de abstracción. Leda es, probablemente, el único lenguaje que admite una equivalencia aún más amplia.

La elección de la equivalencia observacional implica con naturalidad el hecho de que la arquitectura se concibe como concurrente. También se deriva con facilidad la idea de que un componente ha de ser representado semánticamente como un proceso, tal como se verá en el capítulo siguiente (§7.2.3). Por otra parte, esto es una consecuencia obvia, una vez que se ha elegido una sintaxis algebraica para el lenguaje.

Por otra parte, la Arquitectura de Software puede verse, ante todo, como un método básico para *modularizar* el diseño de un sistema. En este sentido, su abstracción fundamental es la de *composición*: de aquí deriva el hecho de que su concepto esencial sea el de *componente*. Todos los demás conceptos se presentan como derivados de éste, esto es, como conectivos, combinaciones, o fragmentos suyos. El único elemento esencialmente distinto es el *puerto*, que se concibe como estrictamente atómico, y en realidad sólo es cada uno de los puntos por los que se *observa* el comportamiento de un componente.

Por tanto, al elegir una equivalencia observacional para el concepto de componente, se está extendiendo la misma idea a todos los demás elementos del lenguaje; y, de hecho, se está implicando para toda la arquitectura –que de hecho, se define en $\mathcal{P}i\mathcal{L}ar$ como un único gran componente compuesto–. El caso de los componentes compuestos es particularmente interesante: al igual que ocurre al propio nivel de la descripción, una configuración tiene un comportamiento global, resultado de la combinación de todas sus partes; pero este comportamiento puede ser comparado con el de un único componente que se considere equivalente, y esta equivalencia puede ser determinada formalmente por bisimulación.

En resumen, un *componente* $\mathcal{P}i\mathcal{L}ar$ se concibe como una unidad de encapsulamiento y composición, definida por bisimilitud (débil) con su especificación, y caracterizada por su interacción con el resto del sistema. Es el concepto fundamental del lenguaje y, como podrá verse, toda la

descripción de una arquitectura se desarrolla en la definición de sus componentes.

6.3 Lenguaje Estructural

El *Lenguaje Estructural* es, como ya se ha dicho, la parte de $\mathcal{P}i\mathcal{L}ar$ que guarda más parecido con un Adl clásico. De hecho, el diseño de su sintaxis pretende mantener cierto grado de continuidad con los lenguajes existentes, conservando las características principales del marco metalingüístico de Shaw y Garlan [SG94], para extenderlo después de la manera más concisa posible mediante la adición de conceptos reflexivos.

En líneas generales, puede verse a $\mathcal{P}i\mathcal{L}ar$ como un heredero muy evolucionado de Wright, en especial en que, al igual que éste, se plantea como un recubrimiento de alto nivel sobre un álgebra de procesos. Su sintaxis, sin embargo, está menos ligada a este origen que la de, por ejemplo, Padl, ya que se intenta proporcionar una perspectiva más netamente arquitectónica; y ha sido enriquecida con adiciones procedentes de otros lenguajes, en especial de $\delta arwin$, tal vez la referencia más importante en Arquitectura Dinámica.

Los elementos del Lenguaje Estructural pueden considerarse por tanto análogos a los de Wright, en líneas generales, aunque con ciertos matices. De manera similar a lo que ocurre en $\delta arwin$, Rapide o Leda, se ha preferido unificar algunos de ellos, en tanto que otros –en particular el concepto de *conector*– han sido completamente abstraídos.

En definitiva, en $\mathcal{P}i\mathcal{L}ar$ hay únicamente cuatro tipos de elementos, no sólo en el Lenguaje Estructural, sino en todo el lenguaje: *componentes*, *puertos*, *conexiones* y *reificaciones*. Como ya se ha dicho, los más importantes son los primeros, y todos los demás se describen en relación a ellos. Los puertos forman parte de su interfaz, las conexiones les permiten comunicarse, y las reificaciones los vinculan con los niveles superiores.

Es lógico, pues, que la estructura sintáctica más importante del Lenguaje Estático sea precisamente la definición de componentes. De hecho, con la excepción de las anotaciones de nivel –que a menudo no proporcionan más que información redundante–, una descripción $\mathcal{P}i\mathcal{L}ar$ se compone, exclusivamente, de *declaraciones de componente*. Todos los demás elementos se describen en su interior, bien como parte de su interfaz, bien como parte de una configuración; incluso las restricciones del Lenguaje Dinámico se indicarán siempre asociadas con un componente. Más aún, como ya se ha dicho, en $\mathcal{P}i\mathcal{L}ar$ una arquitectura se describe –como en $\delta arwin$ o Leda, por ejemplo– como un gran componente compuesto.

Así pues, la sección siguiente describe la estructura de una declaración de componente, desplegando de este modo la sintaxis de *todo* el Lenguaje Estático. Por otra parte, este no se diferencia en exceso de un Adl convencional –en gran medida, esta es la parte "clásica" del lenguaje– por lo que tampoco requerirá excesivos comentarios.

6.4 Declaración de Componente

La estructura sintáctica básica en $\mathcal{P}i\mathcal{L}ar$ es, como se ha dicho, la *declaración de componente*. No es ocioso reiterar aquí la distinción entre tipos e instancias de componente, ya que existe cierta confusión al respecto. Por abuso de lenguaje, es habitual referirse a las *instancias* de un componente como *componentes*, y a los componentes en sí como *tipos de componente*. Esto crea

la falsa impresión de que los componentes están a nivel de objeto, cuando en realidad están –por definición– al nivel de una clase (parametrizada). No obstante, para evitar confusiones, se usará por convenio *arquetipo* para referirse al tipo, y *componente* o *instancia* para las instancias. Este uso no es estrictamente correcto, pero es desde luego el más habitual.

En $\mathcal{P}i\mathcal{L}ar$ un componente (*arquetipo*) se define sintácticamente como una unidad básica de composición, encapsulada y caracterizada por una o varias interfaces (segmentadas), y representada en un sistema o configuración concreta mediante múltiples ejemplares o instancias. Su estructura, el aspecto fundamental que lo define, se especifica mediante una *declaración de componente*, que se constituye como el armazón básico de toda descripción $\mathcal{P}i\mathcal{L}ar$. La Figura 6.1 muestra el esquema básico de una declaración de este tipo.

```
\component nombre [ < parámetros > ] (
  { [ \interface ]
    ( definición de interfaces ) }
  [ \config (
    { declaración de instancia } |
    [ \bind (
      { declaración de conexión }
    ) ] ) ]
  { \reify reificación (explícita) }
  [ \constraint (
     { restricción en el lenguaje dinámico }
    ) ]
  ) /– ... fin del componente –/
```

Figura 6.1: Esquema de una Declaración de Componente en $\mathcal{P}i\mathcal{L}ar$

Sintácticamente, un componente se divide en cuatro partes: *interfaces*, *configuración*, *reificaciones* y *restricciones*; cada una de ellas expresa, respectivamente, su esquema de comunicación externo, su estructura interna, sus relaciones reflexivas y su posible comportamiento. En realidad, todas ellas son opcionales. Esto es, se pueden definir componentes sin interfaz, en cuyo caso se tiene un sistema cerrado; sin configuración, dando lugar a un tipo primitivo; sin reificaciones explícitas, aunque siempre habrá al menos una implícita; y sin restricciones, esto es, sin comportamiento definido.

Como ya se ha dicho, el aspecto fundamental en todo caso es la encapsulación; desde el exterior, lo único que se percibe es su *interfaz*. Según esto, un componente puede ser *primitivo* o *compuesto*. En el primer caso, se corresponde con algún módulo existente, y tan sólo se declara la interfaz que éste exporta. En el segundo caso, su estructura interna se describe como una *configuración*. En una configuración se combinan una serie de instancias de componentes, que interactúan entre sí, y se considera a este agregado como un nuevo componente. Externamente ambos tipos de componente son idénticos; esto permite que un componente compuesto sea, a su vez, combinado con otros, dando forma a la *jerarquía de componentes* que es característica de la descripción arquitectónica.

Una configuración se compone, a su vez, de *instancias* de componente y *conexiones*. En $\mathcal{P}i\mathcal{L}ar$ no hay conectores; por tanto, se asume que las instancias se conectan entre sí directa-

mente, a través de conexiones entre sus puertos. Como se verá, éstas pueden manifestar una estructura compleja, pero en general se asimilan a simples *enlaces* entre los puertos.

A continuación se describe brevemente la estructura de una declaración de componente estudiando sus distintos elementos; lo cierto es que no será necesaria una explicación excesivamente prolija, ya que en su mayoría se manejan conceptos habituales en cualquier Adl.

6.4.1 Declaración de Interfaz

Se define un *puerto* como cada uno de los puntos por los que un componente interactúa con el resto del sistema. Expresa tanto los servicios que exporta, como las peticiones que realiza. Estos puertos pueden estar tipados, bien con un tipo primitivo, bien como un tipo-tupla definido mediante una cláusula **deftype**, que se proporciona únicamente con este fin –véase el Apéndice A para más detalles–. En el caso habitual, no obstante, no se indicará este tipo, que a menudo podrá deducirse de la propia especificación.

Se define a su vez una *interfaz* como una agrupación de puertos lógicamente coherente. Cada puerto se declara como un nombre precedido de la palabra reservada **port**, mientras que una interfaz se declara como una enumeración de tales puertos, separados mediante el símbolo de composición paralela (|). Se utiliza éste para señalar que los distintos puertos pueden ser activados en paralelo; sin embargo, no se ha de asumir por ello que la enumeración es conmutativa: en el caso particular de una *conexión tipada* –que como se verá, también es un componente–, la ligadura de puertos –es decir, la correspondencia entre los puertos enlazados por una conexión concreta, indicados durante su instanciación, y los nombres de puerto declarados en la interfaz de su tipo– se realiza precisamente según este orden.

La sintaxis de $\mathcal{P}i\mathcal{L}ar$ admite múltiples declaraciones de interfaz en un mismo componente. Se trata, simplemente, de enumeraciones de puertos que no guardan relación entre sí. Esto no plantea problema alguno, ya que *a priori* no se afectan entre sí, y combinarlas en una sola interfaz equivale a yuxtaponerlas en paralelo. Cuando hay varias interfaces, se puede marcar una de ellas como privilegiada, mediante la palabra clave **interface**. Cuando sólo hay una interfaz, el uso de esta palabra está permitido, pero resulta irrelevante. En una conexión tipada, ésta es la única interfaz que será seleccionada para la mencionada ligadura de puertos.

Como se verá, todo arquetipo será reificado como un metacomponente. Se asume, en tal caso, que tiene una copia de la interfaz para uso propio, es decir: que al igual que hay un puerto P en cada instancia, habrá también un puerto P en el metacomponente, formando parte del *interfaz de nivel meta*. Estos "meta-puertos" –valga la expresión– podrán ser citados específicamente mediante el cualificador **self** del Lenguaje Dinámico, que se describe en la sección 6.10.1. Salvo en lo que respecta a la coincidencia de nombre, no existirá relación alguna entre éstos y los puertos homónimos del nivel base.

No obstante, cuando se desee disponer de una interfaz explícitamente diferente para el metacomponente, ésta podrá declararse del mismo modo que cualquier otra, precedida de la palabra clave **metaface**. Esta interfaz de nivel meta recibe el nombre abreviado de *metafaz*, y no afecta en modo alguno a las interfaces del nivel base.

Finalmente, se ha de señalar que en $\mathcal{P}i\mathcal{L}ar$ se admite la definición de *arrays* de puertos; se trata simplemente de azúcar sintáctico, que permite unificar identificadores, y ocasionalmente dotarlos de un comportamiento común.

6.4.2 Declaración de Instancia

Como ya se ha dicho, se define una *configuración* como un conjunto de *instancias* de componentes, interrelacionadas mediante *conexiones* o *enlaces*. Se utiliza para describir la estructura de un subsistema, que eventualmente se encapsula como un componente compuesto.

Las instancias de componente se declaran con la sintaxis habitual *nombre:tipo*. Esto describe la creación de una instancia dada, en un tipo concreto. No obstante, se admiten también otro tipo de definiciones, relacionadas también con otros aspectos sintácticos.

En primer lugar, se admite la declaración de un *array* de instancias. Al igual que en el caso de los puertos, se trata simplemente de un modo de agrupar nombres, o de definir varias instancias de manera simultánea. No obstante, pueden resultar muy útiles, en especial si el tamaño del array está parametrizado.

Se admite también la declaración de *instancias parametrizadas*, es decir, aquellas que corresponden a un arquetipo paramétrico. El parámetro habrá de ser siempre proporcionado: por tanto, la instancia se crea una vez que el parámetro ha sido ya sustituido, y no tiene de hecho diferencia alguna con una instancia convencional. Además, pueden declararse instancias de *componente con puertos ligados*, esto es, de un arquetipo al que se le proporcionan como parámetro especial una lista de puertos. En realidad, en este caso se tratará de una conexión tipada, por lo que será descrita en el punto siguiente.

Finalmente, existe un elemento que puede ser utilizado como una instancia, aunque no lo es en realidad: se trata de un *tipo reificado*; se define brevemente en la sección 6.4.4.

6.4.3 Declaración de Conexiones

Las conexiones son el único medio de expresar interacción entre instancias en $\mathcal{P}i\mathcal{L}ar$. Se declaran como parte de una configuración, dentro del ámbito de una sección marcada por la palabra clave **bind**. Existen tres tipos de conexiones:

Enlace. Se trata de un enlace directo entre dos componentes, que se asume como una identificación de puertos. Se indica simplemente como $A = B$, donde A y B son dos puertos. Cuando se crea este enlace, se considera que éstos comparten un canal; toda emisión de A es recibida en B, y viceversa. No obstante, se admiten múltiples enlaces sobre el mismo puerto: en tal caso, sólo uno recibe el mensaje emitido, de manera totalmente indeterminista –y coherente con la semántica de cualquier álgebra de procesos–.

Enlace Jerárquico. Se define como un caso particular del anterior, y se expresa con la misma sintaxis. Hace referencia al enlace entre un puerto de una instancia y un puerto de la *interfaz del componente que lo contiene*, es decir, de su contenedor. Su efecto es, simplemente, expandir el ámbito de ese puerto, permitiendo que pueda interactuar con elementos externos, desde el interfaz del contenedor.

Conexión Tipada. Equivalente a los componentes con puertos ligados, ya mencionados. En este caso, la conexión siempre tiene nombre, y está tipado por un componente, que define su comportamiento. A diferencia de los anteriores, no está restringido a dos puertos, sino que puede conectar un número arbitrario de ellos. Se corresponde, por tanto, con las conexiones no binarias definidas en Marmol.

Los puertos ligados por esta conexión se indicarán como argumentos para el tipo, durante la declaración; esto es, con la forma:

$$nombre_conexión{:}arquetipo(puertos)$$

de modo que el arquetipo recibe esa lista de puertos, la contrasta con su propia interfaz, y los liga según su orden. Si tiene varias, elige la interfaz privilegiada. Ha de tenerse en cuenta que el tipo será reificado (§6.5), por lo que a toda conexión tipada le corresponde una descripción en el nivel meta. Es a ésto a lo que se denomina un *conector de nivel meta*. Será discutido en detalle en la sección 6.11.

En realidad, todas las conexiones se definen como conexiones tipadas; los enlaces –simples o jerárquicos– simplemente tienen un tipo implícito, denominado **link**, que recibe dos parámetros y les da este comportamiento. Su sintaxis habitual no es más que una forma de azúcar sintáctico. Por lo demás, estas conexiones –los enlaces– son completamente idénticos a una *adjunción* pura, tal como se define en otros lenguajes. De hecho, se ha demostrado –a partir de su definición semántica– que ambos conceptos son observacionalmente equivalentes por bisimulación débil [Mil99], esto es, la misma usada en todo este contexto.

6.4.4 Reificaciones y Restricciones

Poco se dirá aquí sobre la estructura y sintaxis de estas dos construcciones, ya que serán desarrolladas en apartados posteriores. La noción de reificación –ya conocida de Marmol– se define en un contexto lingüístico en la sección 6.5; por su parte, las restricciones constituyen el marco en el que se desarrolla la sintaxis del Lenguaje Dinámico, por lo que se remite su exposición hasta la sección 6.6 y posteriores, donde ésta será tratada en detalle.

No obstante, hay dos comentarios que merece la pena hacer ya en el contexto del Lenguaje Dinámico, por lo que se mencionan brevemente en este apartado.

En primer lugar, el efecto de las reificaciones implícitas (§6.5) hace que todo arquetipo sea reificado como un metacomponente, aunque no siempre explícitamente utilizado. Sin embargo, debe tenerse en cuenta que, una vez que el metacomponente existe, puede ser empleado *como una instancia más* en las configuraciones arquitectónicas existentes en el nivel meta. Es decir, puede interactuar con otras instancias del nivel meta –los *componentes meta*– e incluso ser compuesto como parte de otro componente. En tales circunstancias, se sigue haciendo referencia a él por su nombre –el nombre del *tipo* de componente–, pero sintácticamente ocupa el lugar de una instancia. Esto es lo que en terminología $\mathcal{P}i\mathcal{L}ar$ se ha denominado un *tipo reificado*, y es en todo equivalente a cualquier instancia, aunque por supuesto siempre dentro de construcciones del nivel adecuado. Puede ser incluso reificada a su vez por un meta-metacomponente, aunque nunca deberá declararse como una instancia de otro tipo.

Ha de tenerse en cuenta, además, que un tipo reificado no representa a la entidad como arquetipo, sino como un *metacomponente*. Esto quiere decir que su comportamiento será el definido por su restricción y reificaciones, pero que el interfaz que a través del que se conecta es su interfaz de nivel meta –metafaz, si es explícito–.

En lo que respecta a las restricciones, se debe señalar que *no existe* ningún medio para indicar una regla que no vaya asociada a un componente. Toda restricción formará parte, pues, de una

declaración de componente, y será aplicada en su contexto. Sin embargo, no ha de pensarse que esto implica que no pueden expresarse restricciones *globales* sobre el sistema: dado que toda la arquitectura se describe, igualmente, como un componente compuesto, es perfectamente posible asociar una restricción a éste, que por tanto será aplicada a la totalidad del sistema.

Por tanto, la vinculación de toda restricción a un componente no supone una limitación, sino simplemente un criterio para la modularización de las descripciones de dinamismo.

6.4.5 Estructuras Sintácticas Adicionales

Además de todo lo dicho, el lenguaje describe también una serie de estructuras condicionales –o alternativas– e iterativas, análogas a la de cualquier lenguaje de programación, y similares en particular a las definidas en algunos Adls, particularmente en *δarwin*.

Alternativa

Sólo existe una estructura condicional, que utiliza una sintaxis análoga a la de un lenguaje de programación clásico, marcada mediante la palabra clave **if**. Puede ser simple o doble, y permite definir una estructura que se despliega o no, según una condición. La Figura 6.3 muestra un ejemplo de estructura alternativa doble –que, en su forma actual, no necesita siquiera de una etiqueta **else**–, en el contexto de una recursión.

Iteración

Se definen dos tipos de estructura iterativa. Ambos se conciben básicamente como iteradores sobre un conjunto; cuando este conjunto es una secuencia numérica, adoptan la forma de un típico bucle contador. La variable de control puede indicarse explícitamente o no; en este último caso puede ser sustituida por la palabra reservada **now**.

```
\component Filter (
 \interface (
   port left | port right ))

\component PipeIter <N:int> (
  \interface (
    port into | port out )
  \config (
    F[1 .. N]:Filter |
    \bind (
      into = F[1]. left |
      out  = F[N].right |
      \for i (2 .. N−1)
        (F[i−1]. right =
                F[i] . left)) ))
```

Figura 6.2: Filtro-Tubería: Iterativo

```
\component PipeRec <N:int> (
 \interface (
   port into | port out )
 \config (
   head:Filter |
   \if (N>1)
    ( tail:PipeRec<N−1> |
      \bind (
        into = head. left |
        out  = tail .out |
        head. right = tail . into ))
   /− else −/
    ( \bind (
        into = head. left |
        out  = head.right )) ))
```

Figura 6.3: Filtro-Tubería: Recursivo

Las iteraciones sobre conjuntos sólo se introducen con el fin de abreviar patrones de creación de instancias y conexiones; no pueden utilizarse, de hecho, con fines de cálculo. Pueden desplegarse como una *secuencia*, en cuyo caso se tendrá un bucle **for**. Esto significa, en resumen, que las instancias y conexiones serán creadas en un orden determinado. Pero también pueden desplegarse *en paralelo*, en cuyo caso se tendrá un bucle **loop**. Intuitivamente, su significado es realizar todas las creaciones e instanciaciones de manera simultánea, lo que puede afectar a su comportamiento. En particular, no se asume ninguna secuencia entre sus distintos ciclos.

Parametrización

Al igual que otros Adls, $\mathcal{P}i\mathcal{L}ar$ permite cierto grado de *parametrización* de los componentes; esto dota al lenguaje de gran potencia, en particular cuando se combina con estructuras tales como los *arrays*, la iteración, o la *recursión*. No obstante, los parámetros indicados a un tipo consisten únicamente en tipos primitivos –básicamente, números–, y tan sólo se definen con el fin de permitir una generalización de las declaraciones de tipo. En particular, no se admiten como parámetros otros tipos más complejos, concretamente otros arquetipos. En ningún caso se plantea la posibilidad de utilizar este mecanismo para definir *arquetipos de orden superior*. Éstos no existen en ningún Adl conocido, y de hecho no parecen tener mucho sentido en una descripción arquitectónica, donde las configuraciones derivadas se construyen por composición. En su lugar parece mucho más lógico utilizar una construcción análoga a los *fragmentos arquitectónicos* de Jan Bosch [Bos98], cuyo enfoque es radicalmente diferente.

Recursión

Finalmente, $\mathcal{P}i\mathcal{L}ar$ admite *recursividad*, esto es, la definición de arquetipos recursivos. Por supuesto, esta recursión ha de ser finita, ya que en cada paso de la misma se estarán *creando* instancias. Por tanto, se indicará siempre su fin mediante una estructura alternativa, y además irá normalmente asociada a algún tipo de parámetro. Sin embargo, la recursividad no controla *sintácticamente* su corrección. Así, por ejemplo, la construcción de la Figura 6.4 tiene una sintaxis correcta, pero define una recursión infinita, y por tanto no está describiendo a ningún sistema existente. La semántica, de hecho, será capaz de asignarle un tipo, por lo que no puede detectar esta incorrección de manera sintáctica –como no lo hace, por otra parte, ningún lenguaje conocido–; sin embargo, no podrá generar una representación interna para él, debido a su infinitud, y por tanto no define una estructura válida.

```
\component TipoRec
( ( ... )
  \config ( InstRec:TipoRec ... ) )
```

Figura 6.4: Componente Infinitamente Recursivo – Ilegítimo

Las Figuras 6.2 y 6.3 muestran varias posibles combinaciones de los elementos descritos en este apartado, sobre el ejemplo de un clásico sistema filtro-tubería, esto es, una secuencia de filtros unidos mediante conexiones. La primera muestra una versión iterativa, parametrizada

sobre el tamaño de la tubería y realizada como un bucle **for**. La segunda está igualmente parametrizada, pero da una solución recursiva, de modo que la recursión se detiene cuando el número de elementos "pendientes" de conexión se hace cero.

6.5 Reificación: Reflexión Arquitectónica

La mayor diferencia entre $\mathcal{P}i\mathcal{L}ar$ y cualquier otro ADL es el concepto de *reificación*. Éste es el modo en que se introduce la reflexión en el lenguaje, y se presenta como la única abstracción adicional que es necesario incorporar. Por tanto, es su uso lo que permite decir que el lenguaje es reflexivo, y lo que le dota de una mayor expresividad. Se trata de un concepto estructural, pero resulta determinante para el dinamismo, ya que las reglas evolucionan según el esqueleto que describe; y de aquí derivan su significado e importancia.

En $\mathcal{P}i\mathcal{L}ar$, como en MARMOL, la reificación se define como un tipo de relación, la tercera dimensión de la descripción arquitectónica junto a la composición y la interacción. Expresa la *conexión causal* entre dos componentes situados en distintos niveles de abstracción, que se denominan respectivamente *avatar* (componente base) y *meta-componente* o *rohatar*. Se considera que el segundo controla al primero o, de modo más preciso, que es el *contexto* –intérprete– en el que se éste se ejecuta. Cualquier alteración en uno de ellos se refleja de inmediato en el otro.

Desde otra perspectiva, se puede ver al avatar como un *dato* que es manipulado por el metacomponente, o al metacomponente como un elemento que *materializa* –literalmente, reifica– alguna de las abstracciones del avatar. Esto permite que el metacomponente altere libremente al avatar, o que el avatar influya en su propia concepción; de ahí su gran expresividad.

La relación de reificación ha sido descrita como parte de MARMOL, por lo que no es necesario repetir aquí sus características. En todo caso, aunque se dispone de un resumen completo en la sección 5.3, se reiteran a continuación algunos de sus aspectos fundamentales:

- En virtud de la reificación, el sistema se divide en múltiples meta-niveles. No hay límite a su número (*grado de introspección*), si bien usualmente no será mayor que tres.

- La reificación es una relación muchos a muchos: un avatar puede reificar simultáneamente a varios meta-componentes, y un metacomponente puede reflejarse en varios avatares.

- Un metacomponente, en su mismo nivel, puede interactuar libremente con otros componentes. Éstos influyen indirectamente en los avatares, y desde su perspectiva son *componentes meta* o *niyatares*.

Un punto particularmente significativo es que no existe distinción sintáctica entre elementos base y meta; la diferencia procede únicamente de su posición en la descripción, relativa a las reificaciones que mantengan. Esto facilita que, a su vez, un metacomponente pueda ser ser reflejado por un *meta*-metacomponente, dando lugar a la estructura de múltiples niveles mencionada más arriba, que también ha sido caracterizada durante la descripción de MARMOL.

De entre las distintas versiones de MARMOL, en $\mathcal{P}i\mathcal{L}ar$ se utiliza la última de las definidas, esto es, el modelo $\aleph_{\Pi}$. Esto significa que la reificación se colapsa con el tipado: es decir, el arquetipo de un componente se reifica como un metacomponente. El inverso, como es sabido, no es cierto: la reificación de un componente no equivale a una definición de tipo. En cualquier

caso, esto establece que $\mathcal{P}i\mathcal{L}ar$ utiliza el análogo de un modelo Mmc, tal como fue descrito en la sección 4.5.1; es decir, tiene una estructura similar a la definida por un esquema de metaclases reflexivas, aquí sustituidas por la noción de *arquetipos*, a distintos niveles.

Esto significa que en el lenguaje existen dos tipos de reificación: *explícita* e *implícita*[2]. En realidad se trata del mismo concepto, introducido con dos sintaxis diferentes. En la *reificación explícita* se utiliza una construcción sintáctica específica, indicada por la palabra clave **reify**, e incluida en la definición de un componente:

reify [*nombre_enlace*] [**as**] *metacomponente*

Cada instancia del (tipo de) componente en el que aparece esta construcción es un avatar del metacomponente indicado. La reificación crea pues un enlace, de naturaleza muy diferente a la de las conexiones ya mencionadas, que puede recibir un nombre. Este último detalle permite que pueda ser modificado por alguna de las restricciones dinámicas, pudiendo ser incluso borrado mediante una instrucción explícita (§6.10.3).

La *reificación implícita*, por su parte, no se indica mediante sintaxis alguna: simplemente, cuando se declara (C:T) una instancia C de un cierto tipo de componente T, no sólo se crea dicha instancia, sino también un metacomponente que reifica al arquetipo T –siempre, por supuesto, que no existiera previamente–, y recibe ese mismo nombre. Es decir: *en todo momento, los arquetipos declarados pueden ser accedidos como metacomponentes* en el nivel inmediatamente superior –ananivel–. Esto permite realizar, en definitiva, algo análogo a la Reflexión Estructural: dado que las definiciones de tipos son elementos concretos del sistema, pueden ser alterados dinámicamente, proporcionando un elevado grado de flexibilidad.

Como se ha dicho, la reificación de un metacomponente da lugar a la aparición de un nuevo meta-nivel. Puede pensarse que la identificación implícita entre ésta y el tipado genera una secuencia indefinida de meta-niveles –una *torre reflexiva infinita*–. En realidad, esto no es así: en primer lugar, un arquetipo sólo será un metacomponente activo cuando el sistema así lo requiera. Si no tiene restricciones dinámicas o no se relaciona con el resto del sistema, puede ser considerado igual que un arquetipo tradicional. En segundo lugar, cuando un arquetipo es reificado como un metacomponente, no tiene *a priori* un tipo definido, salvo el suyo propio, de manera análoga a lo que se denominaba un *arquetipo anudado* en Marmol. Sólo en el caso de que se lo declare a su vez como instancia de otro tipo –algo muy poco lógico– o se lo reifique explícitamente en el Lenguaje Dinámico –mucho más habitual– se tendrá un auténtico despliegue del meta-nivel superior, siempre que éste no existiera ya previamente.

Finalmente, dado que en $\mathcal{P}i\mathcal{L}ar$ las conexiones pueden ser nombradas y tipadas, es fácil ver que también pueden ser reificadas. Es decir, lo que a nivel base es un simple enlace –una *conexión tipada*–, toma la forma de un componente en el nivel meta, que puede interactuar y manipularse como cualquier otro componente. Mediante este mecanismo se permite pues la definición de *conectores* en $\mathcal{P}i\mathcal{L}ar$, y se hace innecesaria la creación de una categoría sintáctica aparte, como ocurre en otros Adls. Las consecuencias del uso de metacomponentes como *conectores de nivel meta* serán tratadas con mayor detalle en la sección 6.11.

[2]Sin relación con la terminología análoga de Maes [Mae87a], que determinaba cuál de los actores del sistema iniciaba un proceso de reificación.

Como es sabido, desde un punto de vista teórico, la introducción de la noción de reflexión en un lenguaje plantea muchos problemas. En general, su control supone la existencia de una *jerarquía de lenguajes* (§4.2), según la cual cada lenguaje L_n es definido por un meta-lenguaje L_{n+1}, desde el cual existe el potencial para alterar las nociones del primero. Este razonamiento es, precisamente, el que implica la existencia de una *torre infinita* de intérpretes, como la ya mencionada. Esto ocurre incluso cuando se asume que los intérpretes a todos los niveles definen el mismo lenguaje, en un proceso que se denomina *metacircularidad* ($L_n \equiv L_{n+1}$). Como ya se expuso en la sección 4.3.2, la reificación fue introducida, precisamente, como un modo de controlar la expansión de esta torre.

Sin embargo, Maes [Mae87a] demostró que este esquema podía ser simplificado en sistemas fuertemente modulares, como los sistemas orientados al objeto. Asumiendo cada módulo –en nuestro caso, cada componente– como independiente, se puede considerar la agrupación de la meta-información relativa al mismo como un meta-módulo; esto es lo que se denomina habitualmente un *meta-objeto* –en nuestro caso, un metacomponente–. Puede considerarse, incluso, que cada uno de estos meta-módulos actúa como un intérprete específico para cada módulo. En el caso de un modelo Mmc, como el que se usa en $\mathcal{P}i\mathcal{L}ar$, el tipado y la reflexión se colapsan, y se asume por definición que cada uno de los arquetipos declarados se presenta como un meta-módulo o intérprete de esta especie.

$\mathcal{P}i\mathcal{L}ar$ traslada estas ideas a componentes, y mediante la relación de reificación admite todos los esquemas conocidos. No basta, sin embargo, con definir esta relación sintácticamente, sino que se necesita una caracterización semántica precisa, que sea coherente con el resto del lenguaje y le dé el significado pretendido. Como se verá en el Capítulo 7, la semántica de $\mathcal{P}i\mathcal{L}ar$ concibe al sistema como un conjunto de procesos concurrentes, comunicados mediante canales, en el que cada componente equivale a un proceso abstracto (§7.2.3).

6.5.1 Anotaciones de Meta-Nivel

En todo el esquema anterior, los niveles se infieren por sí solos. Es fácil determinar, a partir de una descripción arquitectónica, el meta-nivel en el que se sitúa cada uno de los elementos de la misma. El proceso es similar al de una inferencia de tipo *a la* Curry [Bar92] o, más aún, al de una estratificación en la teoría de conjuntos de Quine.

Se agrupan, en primer lugar, los elementos que se relacionan entre sí, sea por composición o por interacción; todos los miembros de cada grupo estarán ubicados en el mismo meta-nivel, tal como indica la Segunda Regla de Estratificación (ley 5.3) de Marmol. A continuación se parte de aquellos elementos que, *a priori*, sólo aparezcan como instancias o sean reificados; se asume, temporalmente, que todos éstos son elementos-base. Los grupos que estén formados exclusivamente por componentes de este tipo, constituyen en efecto el *nivel base*.

A partir de aquí, todo arquetipo o componente que reifique a un miembro del nivel base estará en el nivel meta, es decir, en el primer meta-nivel. Por tanto, el conjunto de todos los grupos que contengan a alguno de estos elementos constituye, a su vez, el *nivel meta*. De nuevo, todo el que reifique a un miembro del nivel meta estará en el *nivel meta-meta* –es decir, en el segundo meta-nivel–, lo que abarca también a todos sus grupos, y así sucesivamente. Cualquier conflicto en esta asignación de niveles indicaría una mala estratificación, lo que implicaría que la descripción arquitectónica analizada es incorrecta.

No obstante, en ocasiones este mecanismo podría revelar alguna ambigüedad, dificultando la asignación de nivel de algún elemento. Esto es muy inusual, pues implicaría, en principio, que la descripción se compone de fragmentos aislados, que no se relacionan ni por composición, ni por interacción, ni por reificación. Con el fin de solucionar estos conflictos, de facilitar la inferencia de niveles y, sobre todo, como documentación, se introduce en el lenguaje el concepto de *anotación de nivel*. Su rasgo más notable es que es el único elemento del lenguaje, junto a la definición de tipos primitivos, que no forma parte de una declaración de componente.

Una anotación de nivel no es más que una sentencia formada por una etiqueta especial y una secuencia de nombres de componentes. Las etiquetas pueden ser de dos tipos, a saber:

base-level: (*secuencia*)
meta-level[n]: (*secuencia*)

E indican, simplemente, que los componentes enumerados en la secuencia están en el nivel base, o tienen un meta-nivel determinado. No es preciso enumerar *todos* los miembros de un nivel, siempre que el resto puedan ser deducidos a partir de los indicados. Normalmente, se indica sólo la primera etiqueta (**base-level**), con el fin de proporcionar fácilmente el punto de partida para el proceso de inferencia descrito. De nuevo, se ha de asumir que los niveles asignados por las anotaciones de tipo son consistentes con la estructura del sistema especificado. De otro modo, se considera de nuevo que la descripción arquitectónica es incorrecta.

Toda la estructura de niveles se basa en las dos reglas de estratificación de Marmol, que se mantienen de manera explícita en el modelo $\aleph_{\Pi}$, y por tanto en $\mathcal{P}i\mathcal{L}ar$. Concretamente, la Primera Regla de Estratificación limitaba el conjunto de elementos con los que se podía establecer una reificación, asegurando la consistencia entre esta relación y la definición de meta-niveles. Por su parte, la Segunda Regla de Estratificación aseguraba explícitamente que toda relación de composición e interacción se realizaba siempre dentro del mismo meta-nivel, asegurando la coherencia estructural entre las tres dimensiones de la descripción arquitectónica.

Con el objetivo de realizar un control estructural similar, desde la perspectiva del lenguaje, se introducen en $\mathcal{P}i\mathcal{L}ar$ una serie de normas relativas a la reflexión, que se agrupan en un único bloque. A éste se le denomina globalmente, por analogía con las reglas del modelo, *principio de estratificación*. Se enuncia como sigue:

Ley 6.1 (Principio de Estratificación) *Toda descripción arquitectónica en* $\mathcal{P}$i$\mathcal{L}$ar *se estratifica en dos o más niveles, y en ellos siempre se cumple que:*

- *Toda entidad se ubica en un nivel, y sólo en uno.*
- *Los meta-niveles (adyacentes) se desarrollan según relaciones de reificación.*
- *Toda interacción y composición se define entre entidades del mismo nivel.*
- *Sólo se pueden manipular dinámicamente los elementos de niveles inferiores.*

Como puede comprobarse, los tres primeros puntos se refieren a nociones básicamente estáticas, que garantizan la congruencia con las dos reglas homónimas del modelo $\aleph_{\mathrm{M}}$ de Marmol. Por ejemplo, se garantiza que una entidad base –ubicada en el nivel cero– no puede a su vez reificar a otro elemento, lo que implicaría un *ascenso* de nivel.

Por su parte, el último punto impone en cambio una restricción –valga la redundancia– sobre el modo en que un proceso puede manipular a los nombres correspondientes a los niveles inferiores, restringiendo explícitamente a los del nivel actual. Éste será sólo importante, por tanto, en el contexto del Lenguaje Dinámico.

6.6 Lenguaje Dinámico

Como ya se ha dicho, se denomina *Lenguaje Dinámico* al subconjunto de $\mathcal{P}i\mathcal{L}ar$ que especifica la sintaxis de las *reglas* o *restricciones* asociadas a los componentes, y que describen tanto las características de su interacción, como la evolución dinámica de su estructura –incluyendo a la estructura reflexiva– a lo largo del tiempo.

La sintaxis del Lenguaje Dinámico es de carácter *imperativo*, en contraposición con el carácter declarativo del Estructural. Dado que pretende expresar dinamismo, esto es inevitable; pero no ha de pensarse por ello que describe una "ejecución" de tipo procedimental. En realidad, su concepción es también parcialmente declarativa; de ahí el nombre de ***restricciones***. Se trata, más bien, de un conjunto de *reglas* que se asocian al armazón estructural ya descrito, asegurando que ciertas propiedades se conservan, o reconfigurando el sistema en respuesta a situaciones concretas. En cierto modo, pueden recordar, en su filosofía, a las reglas Eca de las Bases de Datos Activas [PD99], ya que se trata de acciones que se disparan cuando se producen ciertos eventos bajo ciertas condiciones.

La estructura del Lenguaje Dinámico o, más exactamente, la del subconjunto *no reflexivo* de su sintaxis, está fuertemente influida por la de un álgebra de procesos [BPS01], en concreto por las álgebras del tipo Ccs [Mil80, Mil89], entre las que se incluye, por supuesto, el propio cálculo-π (véase el Apéndice B). Esto es completamente intencional, y parte tanto de la ya citada tradición de Adls algebraicos como de la convicción de que este tipo de notación es capaz de expresar todas las abstracciones necesarias de la forma más precisa y compacta posible.

Tanto Ccs como –especialmente– el cálculo-π son formalismos completos-Turing, lo que justifica en parte la afirmación anterior. Sin embargo, no han sido elegidas por esta razón, sino por su especial adecuación a las necesidades del dominio. El uso tradicionalmente más extendido de las álgebras de procesos ha sido sin duda la especificación de protocolos, en lo que han tenido un éxito notable. Precisamente, ésta es la razón por la que fueron introducidas en Wright, que de este modo describía con total exactitud los flujos de datos entre componentes y conectores. Hoy resulta prácticamente indiscutible la afirmación de que constituyen el mejor medio para especificar lo que hemos denominado dinamismo interactivo (§3.3).

Sin embargo, no es éste el único tipo de actividad que se pretende describir en $\mathcal{P}i\mathcal{L}ar$. Para ser cuando menos equivalente a un Adl dinámico clásico, ha de tener al menos la capacidad de un sistema de reconfiguración dinámica (§3.4), lo que se concreta típicamente en las cuatro primitivas básicas (§3.4.2), esto es, *crear, destruir, ligar* y *desligar*. Una actividad de este tipo da lugar a sistemas con topologías cambiantes, que a diferencia de lo que ocurre con los protocolos, no pueden ser especificados mediante cualquier álgebra. Por el contrario, sólo el cálculo-π y sus herederos –los denominados *cálculos nominales*, véase de nuevo el Apéndice B– tienen la expresividad necesaria. Se considera, en líneas generales, que son los únicos formalismos que definen un soporte explícito para *movilidad*, palabra que se utiliza aquí en su sentido más

formal [Mil99], referido precisamente a la variabilidad de las topologías. No resulta extraño que la semántica de $\delta arwin$ y Leda, tal vez los dos Adls algebraicos dinámicos más potentes, se apoye precisamente en el cálculo-π, de manera mucho más notoria en el segundo caso.

En $\mathcal{P}i\mathcal{L}ar$ también se describe la semántica mediante cálculo-π (§7). Sin embargo, a la hora de especificar la sintaxis se parte de la siguiente hipótesis: *la combinación de reflexión con un álgebra de procesos tradicional es suficiente para expresar cualquier tipo de dinamismo.* Ha de recordarse que el objetivo principal en el diseño de $\mathcal{P}i\mathcal{L}ar$ es demostrar que un Adl cualquiera, al que se le añada un soporte para reflexión, es suficiente para especificar dinamismo. Si la sintaxis del Lenguaje Dinámico estuviera basada en el del cálculo-π, entonces éste tendría capacidad dinámica *per se*, sin necesidad de considerar reflexión. Por este motivo –que no es el único– se prefiere utilizar un álgebra de procesos clásico. Por su parecido con el cálculo-π, y por consistencia con el enfoque utilizado en esta tesis, se ha elegido Ccs como inspiración básica de la sintaxis del Lenguaje Dinámico.

Debe tenerse en cuenta que Ccs y Csp son formalismos prácticamente equivalentes. Esto significa que, limitando el lenguaje de comportamiento de $\mathcal{P}i\mathcal{L}ar$ a reglas Ccs, se está restringiendo su capacidad dinámica –*a priori*– a los niveles de Wright. Por tanto, todo dinamismo en el lenguaje vendrá dado por su enfoque reflexivo. Esto permite una comparación ajustada entre el sistema aquí desarrollado y un Adl esencialmente estático, como es Wright.

La extensión del lenguaje a múltiples meta-niveles se relaciona con los aspectos arquitectónicos, más que con la sintaxis del lenguaje. Pero tiene no obstante unas consecuencias notables; como especifica el Principio de Estratificación (ley 6.1) ya establecido, una acción, a cualquier nivel, podrá afectar a cualquiera de los elementos del nivel inferior. Esto significa, por ejemplo, que en lugar de cuatro primitivas dinámicas, basta simplemente con *dos*: la creación y la destrucción. Dado que éstas pueden aplicarse indistintamente a instancias y conexiones –del nivel inferior, es decir, de un avatar propio–, no tiene sentido introducir una sentencia explícita para ligar o desligar componentes. Lo mismo puede aplicarse a puertos, e incluso a reificaciones.

Por otra parte, aunque no están al mismo nivel de abstracción, resulta bastante lógico suponer que la potencia expresiva de Ccs en un entorno reflexivo será similar a la del cálculo-π. En el momento en que la reificación permite a Ccs manipular elementos del nivel inferior como si fueran datos, esto significa que el álgebra tiene ya la posibilidad de transmitir *canales* durante una comunicación. Ésta es, precisamente, la extensión del álgebra original que dio lugar a la definición del Eccs de Engberg y Nielsen [EN00], que constituyó la inspiración original del cálculo-π. No obstante, esta posibilidad no es ilimitada: la capacidad para movilidad, proporcionada por la estructura reflexiva, queda no obstante limitada por ésta. Es decir, siempre se habrá de respetar la estructura de meta-niveles, tal como indica el Principio de Estratificación. Esto garantiza cierto nivel de control –deseable– sobre el Lenguaje Dinámico.

Finalmente, existen dos sintaxis distintas para este Lenguaje. La primera es una notación de tipo algebraico, básicamente inspirada en Ccs, al que se le añade una serie de primitivas extra. Es compacta, pero poco intuitiva, y posiblemente no muy adecuada para un Adl de uso general. Por ello, se desarrolló una segunda notación, más cercana a la de un lenguaje de programación, pero totalmente equivalente. Su único objetivo es hacer a las reglas más fácilmente legibles. En general se utilizará esta última, por considerarla más comprensible; no obstante, a lo largo de este capítulo se describen ambas.

6.6.1 ¿Por qué no cálculo-π?

No se pretende aquí negar las inmensas capacidades y potencial del cálculo-π. Por el contrario, toda esta tesis se fundamenta en él, y constituye una demostración más de sus enormes posibilidades como modelo semántico de concurrencia. Más aún, llega incluso a adoptar el mismo elemento básico –el nombre– en su modelo semántico y conceptual. Y sin embargo, no se considera que sea el lenguaje más apto para un uso directo por parte del arquitecto de software, ni siquiera cuando se está defendiendo que este lenguaje sea un álgebra de procesos.

Los inconvenientes del cálculo-π como lenguaje de especificación directa son dos: en primer lugar, el posible uso inconsciente de la movilidad; y en segundo, el hecho de que los procesos-π adoptan con facilidad la forma de un *estilo con paso de continuación*. Más concretamente, el primero deriva del hecho de que todo acto de comunicación en el cálculo, excepto las sincronizaciones puras, supone una transmisión de nombres; esto puede implicar que un canal dado realice una *extrusión* de ámbito, esto es, un cambio en la topología. En un uso libre del cálculo, se pueden realizar este tipo de operaciones inadvertidamente, con lo que la arquitectura del sistema se alteraría imprevista e incontroladamente. En cuanto al segundo problema, es aún más complejo. Como se sabe, un proceso del cálculo-π es perfectamente capaz de emular el comportamiento de una función [Mil92], mediante el artificio de utilizar un *canal de respuesta* como parte de su entrada, a través del cual se envía después el resultado. De hecho, esta es la forma normal de trabajo en en cálculo-π. Fue Davide Sangiorgi el primero en señalar que esta formulación resulta análoga al estilo por paso de continuación (*continuation-passing style*, habitualmente abreviado como CPS) del cálculo-λ [Plo75a], en el que una función-λ recibe un parámetro de *continuación*, que representa al "resto" de la computación a realizar, y que puede verse como una función que la actual debe *invocar*, con su propio resultado como parámetro. La elegancia formal del estilo-CPS es grande, pero su uso práctico resulta totalmente contraintuitivo para un programador convencional, ya que altera el orden normal en el que se utilizan las abstracciones funcionales. Ésta es la misma razón, sin saberlo, por la que el uso del cálculo-π resulta, en general, más extraño para el usuario no experimentado que el de cualquier otro álgebra de procesos.

El mismo arquitecto de software que asumiría con cierta facilidad las estructuras de Csp o Ccs, concibiéndolas como formas de especificar un protocolo de comunicación, podría sin embargo amilanarse ante la complejidad inicial del cálculo-π. De hecho, a pesar de su elegancia y difusión, es cierto que el cálculo sigue siendo menos popular, entre los propios estudiosos de la teoría de la Concurrencia, que otras álgebras sin movilidad. Tampoco resulta ajeno a esto la mayor dificultad que entraña a la hora de realizar verificaciones automáticas o semiautomáticas, como le ocurre también al Ccs con paso de valores; pero en todo caso no se considera que éste sea un factor determinante. En lo que respecta a su uso como base de la sintaxis del Lenguaje Dinámico, se piensa que el argumento psicológico es mucho más importante.

Por estos motivos, se considera que cualquier otro álgebra de procesos resulta más fácil de asimilar, para el no especialista, que el cálculo-π. Sin embargo, éste sigue siendo, de todos modos, el formalismo ideal para el desarrollo de la semántica y, como ya se ha dicho, será de hecho el utilizado. Conviene, pues, que la sintaxis de $\mathcal{P}i\mathcal{L}ar$ guarde cuando menos cierto grado de similitud con los operadores del cálculo-π. Esto hace casi obvia la elección de Ccs como fundamento sintáctico del Lenguaje Dinámico de $\mathcal{P}i\mathcal{L}ar$. No sólo está en el origen conceptual del cálculo, sino que muestra una estructura similar, y tiene una sintaxis simple, precisa y fácil

de extender, que se constituye como un punto de partida idóneo.

Por supuesto, el lenguaje no es idéntico a Ccs. Aparte de añadir algunos de sus operadores derivados –básicamente, la composición secuencial–, no puede dejar de tener un operador de *creación*, e incluso otro de borrado; ninguno de éstos existe en el álgebra de procesos. De hecho, el operador de creación está reconocidamente inspirado en el operador de restricción (ν) del cálculo-π. Sin embargo, no ha de pensarse que esto amplía en exceso la capacidad del lenguaje, anulando la analogía con Wright. En realidad, y teniendo en cuenta la regla de ámbito de las restricciones dinámicas (§6.8.1), se podría emular su comportamiento –aunque con cierta dificultad– manipulando los nombres libres y el operador de renombrado de Ccs. En cualquier caso, no se incluye –salvo a través de la reflexión– la característica definitoria del cálculo-π, esto es, la capacidad para transmitir *canales* en una interacción, provocando en el proceso una extrusión de ámbito. Por tanto, en ningún momento se obtiene –salvo por reflexión– un lenguaje de potencia equivalente al cálculo-π.

6.7 Estructura de una Restricción

La estructura del Lenguaje Dinámico se desarrolla por completo como una serie de *restricciones*. Todas ellas están asociadas a componentes, y hay exactamente *una* por cada componente, que se corresponde con la sección **constraint** de su declaración de tipo. Sin embargo, cada restricción puede ser tan compleja como se desee. Su estructura se desarrolla –véase la Figura 6.5– como tres tipos de elementos: definiciones de *canal*, definiciones de *variable* y definiciones de *proceso*. Las realmente importantes son, obviamente, éstas últimas. Estrictamente hablando, una *restricción* o *regla* se compone de uno o más procesos; pero por abuso de lenguaje, a menudo estos últimos recibirán también la denominación de restricciones o reglas.

```
\constraint (
    { chan  canal:tipo }
    { var   variable:tipo }
    { proceso def= ( ... ) }
  ) /- fin de regla -/
```

Figura 6.5: Esquema de una Definición de Restricción en $\mathcal{P}i\mathcal{L}ar$

Las declaraciones de canal y variable son opcionales, y de hecho no muy habituales. En toda regla hay al menos un proceso –de otro modo no tiene sentido definirla–. Cuando solamente existe un proceso, su nombre puede omitirse, salvo en el caso de una definición recursiva.

Los apartados siguientes presentan brevemente a estos tres tipos de elementos.

6.7.1 Canales

Dado que según la regla de ámbito (§6.8.1) todos los canales definidos en un proceso son privados, es preciso disponer de algún elemento común, que haga posible realizar una comunicación entre los distintos procesos. Con este objeto se definen los *canales* como puntos de interacción independientes, comunes a todos los procesos de la restricción en la que se declaran.

Las declaraciones de canal van precedidas de la palabra reservada **chan**; pueden declararse uno o más canales, y se admite la definición de su tipo, sea como un tipo primitivo o como una tupla –tipo unión– de los mismos. Un canal definido sin tipo no carece, sin embargo, de éste. En estos casos se asume, simplemente, que el tipo del canal puede ser deducido trivialmente a partir de su uso, o que es en realidad irrelevante en una descripción dada.

Sobre un canal se admiten operaciones de escritura [**!**] y lectura [**?**], con la misma sintaxis y significado que se da en los canales privados, o en cualquier término Ccs convencional. Toda emisión deposita un dato, que será recogido y *consumido* por una recepción; el único mérito de esta definición concreta es, por tanto, la ampliación de ámbito mencionada. Se admite también el uso de operaciones **when**, que lo equiparan en gran medida a una variable.

Se asume en general que la comunicación a través de estos canales es asíncrona; no obstante, se siguen las reglas del lenguaje, por lo que si este detalle es esencial, ha de indicarse de manera sintáctica. En tal caso, se ubicará como parte de una composición paralela.

La principal diferencia entre un canal y un puerto es que el primero es estrictamente interno; esto no sólo se refiere al ámbito de su definición, sino también a la forma de uso. Toda emisión sobre un puerto sale hacia el exterior. Toda recepción procede también del exterior. En cambio, toda emisión sobre un canal produce un dato, que podrá ser recogido por cualquier otro proceso; una recepción sobre el mismo lo consume si existe; si no, pasa inmediatamente a un estado de espera hasta la próxima emisión.

6.7.2 Variables

El concepto de variable es completamente idéntico al de canal, tanto en ámbito como en operaciones. Sólo hay dos diferencias entre ambos conceptos. La primera, obviamente, es el uso de la palabra reservada **var** en la declaración, donde se admite igualmente una especificación de tipos. La segunda, por su parte, hace referencia al comportamiento de las operaciones sobre ella.

Una emisión [**!**] sobre una variable equivale a una escritura o asignación de la misma. A partir de este momento, cualquier recepción [**?**] podrá leer el dato emitido, tantas veces como sea preciso. El dato contenido en la variable no se consume, y sólo puede ser modificado mediante una nueva asignación, es decir, una nueva escritura.

Puede pensarse que el uso de una variable es similar al de un canal con una lectura-**when**. Sin embargo, hay una diferencia de matiz: en el caso del canal, basta una *lectura* para que el dato sea consumido, momento en el cual toda lectura-**when** ha de quedar a la espera. En cambio, en el caso de una variable, el dato no se consume en ningún caso, y sólo puede ser alterado mediante una orden de *escritura* – es decir, emisión–.

6.7.3 Procesos

Una *definición de proceso* en una restricción $\mathcal{P}i\mathcal{L}ar$ es totalmente análoga a una definición de proceso en Ccs o cualquier otro álgebra, por lo que no requiere excesivos comentarios.

En toda restricción puede haber un número indeterminado de definiciones de proceso; siempre habrá al menos una. Se asume que se trata simplemente de definiciones, esto es, no se asume que todos los procesos se activen simultáneamente. De hecho, el único que se activa durante la instanciación del componente –y, por tanto, el que define el subsistema– es *el primero* de los

definidos. Los demás se activan después, mediante invocaciones explícitas. Un proceso invocado no es, sin embargo, una subrutina: se ha de tener en cuenta que se activa de manera *concurrente*. Sólo podrá dársele esa consideración cuando esté contenido como parte de una composición *secuencial*, ya que en este caso el resto del proceso habrá de esperar a su terminación.

Obviamente, si se necesita activar inicialmente varios procesos de manera simultánea para describir el comportamiento, esto puede hacerse de manera trivial, indicando como primera regla una composición paralela, que contenga una invocación a cada uno de ellos.

La estructura de un proceso consiste, simplemente, en la declaración de un *nombre de proceso*, seguido del operador de definición (**def=**) y de una secuencia de constructores del Lenguaje Dinámico, encerrada o no entre paréntesis. Opcionalmente, un proceso puede estar *parametrizado*, esto es, tener uno o más parámetros que recibe al comienzo de su ejecución; aquí se sigue también la notación e interpretación convencional de Ccs.

Cuando sólo hay un proceso en la restricción, y éste no es recursivo ni parametrizado, puede omitirse la declaración de su nombre, de modo que simplemente se desarrolla su ejecución en el momento en el que se instancia un componente. Ésto sólo será habitual cuando se utiliza una estructura de replicación, pero no hay límite en este sentido.

Las secciones siguientes se dedican específicamente a la descripción de los constructores y operadores de un proceso de este tipo, por lo que no se dirá aquí nada más sobre el particular.

6.8 Constructores de Proceso

Los constructores de proceso en el Lenguaje Dinámico son análogos a los definidos en Ccs o cualquier álgebra convencional. Su interpretación es, por lo demás, completamente estándar, salvo en un caso –el de la replicación–.

Los constructores pueden agruparse en dos categorías; por una parte, los constructores básicos de cualquier álgebra; por otra, una serie de constructores avanzados, de naturaleza más algorítmica. En ambos casos se admite un libre uso de paréntesis como constructor de bloque, evitando cualquier posible ambigüedad sobre el alcance de las estructuras indicadas.

Constructor	**Significado**
A **;** B	Composición Secuencial
$A \mid B$	Composición Paralela
$A \mid\!+ B$	Selección No Determinista

Tabla 6.1: Constructores Básicos (Algebraicos)

Los constructores básicos son perfectamente conocidos. Se trata de los tres que pueden verse en la Tabla 6.1: la *secuencia*, la *composición paralela* y la *selección*. El primero describe el constructor secuencial clásico, según el cual cada proceso ha de esperar a la terminación del anterior para comenzar su ejecución. Se utiliza como separador el punto y coma, aunque se admite también –por simplicidad– el salto de línea.

El segundo describe la activación simultánea de todos los subprocesos indicados, dado lugar a una ejecución concurrente. Se indica mediante el operador clásico de composición paralela (|),

y se trata de una composición tipo-Ccs, es decir, no se asume sincronización alguna, ni se fuerza a la conjunción sobre un vocabulario predefinido.

Finalmente, el tercer constructor describe un operador de suma o selección, en el que se *elige* a uno de los subprocesos para su ejecución, y se descarta el resto. Se trata de una selección tipo-Ccs, lo que significa que puede ser totalmente indeterminista, en situación de ambigüedad, o desarrollarse como una selección *guardada*, esto es, en la que se selecciona una de las alternativas según su primera acción –típicamente, una recepción–. A pesar de que este constructor resulte más extraño en un desarrollo convencional, es completamente imprescindible para un modelado coherente. Se utiliza como operador el combinado (|+), para evitar confusiones con la suma aritmética convencional. Todos ellos pueden combinarse con total libertad.

Constructor	**Significado**
if (*condición*) (*A*)	Alternativa Simple
if (*condición*) (*A*) (*B*)	Alternativa Doble
when *recepción* (*A*)	Guarda No Destructiva
for [*vcb*] (*conjunto*) (*A*)	Bucle en Secuencia
loop [*vcb*] (*conjunto*) (*A*)	Bucle en Paralelo
rep (*A*)	Replicación

Tabla 6.2: Constructores Avanzados (Algorítmicos)

La Tabla 6.2 describe el resto de constructores del Lenguaje Dinámico. La mayoría de ellos son perfectamente conocidos, e incluso han sido descritos ya en el Lenguaje Estructural; la única diferencia es que en este caso componen *procesos*, en lugar de instancias y conexiones. Los dos primeros describen la clásica estructura *condicional* o *alternativa*, simple o doble, en función de una condición lógica (booleana). El cuarto y el quinto son, de nuevo, dos tipos de *iteración*, cuya interpretación es la misma que en el caso estático, aunque ahora resulta mucho más importante. Así, los distintos ciclos de un bucle **for** se ejecutan *en secuencia*, de modo que cada uno de ellos ha de esperar a la finalización del anterior. En cambio, en el bucle **loop** todos estos ciclos se disparan *en paralelo*, asumiendo en todo caso que la variable de control del bucle se ha incrementado. En este último caso, se pueden producir combinaciones sumamente complejas.

La tercera estructura (**when**) es totalmente análoga a una condicional simple, pero en ella se sustituye la condición por una recepción no destructiva. En realidad, se trata de una acción básica, y como tal será descrita en la sección 6.9.

Existe, por fin, un constructor de replicación o reiteración, que es descrito un poco más abajo. Ha de señalarse aquí que también se admite –por supuesto– la *recursión*, concebida del mismo modo que en cualquier álgebra de procesos. Toda definición de proceso puede ser recursiva, e incluso ilimitada; el lenguaje se encarga de proporcionar un punto final a las recursiones cuando se destruye una instancia. No obstante, la terminación también puede ser controlada desde la propia definición del proceso, utilizando la selección o la estructura alternativa.

A lo largo de toda esta sección se ha asumido que los constructores presentados realizaban una composición de *subprocesos*. Sin embargo, éstos no tienen por qué estar definidos como

una declaración aparte: puede tratarse simplemente de un bloque de términos del Lenguaje, encerrados o no entre paréntesis. En el caso mínimo, se tendrá al menos *un* término. De hecho, todo subproceso se construye mediante composición de tales términos, utilizando para ello los constructores aquí descritos. En este punto, también se sigue las mismas reglas que en un álgebra de procesos convencional.

6.8.1 Regla de Ámbito

Los nombres utilizados en los procesos de una regla siguen unas reglas de ámbito similares a las descritas para Ccs, con dos grandes diferencias. En primer lugar, se pueden *crear* nuevos nombres durante la ejecución del proceso, mediante la orden **new**. En este sentido, se sigue la misma estrategia del cálculo-π, pero sin extrusión de nombres. En segundo lugar, se asume por definición que todos los canales usados en un proceso dado son privados. Esto se expresa más adecuadamente en la regla siguiente:

Ley 6.2 (Regla de Ámbito) *Todos los canales (nombres) usados en un proceso son estrictamente* privados*; los únicos canales comunes a los distintos procesos son aquellos definidos como parte de la restricción (**chan**), así como los* puertos*, que se definen como* públicos*.*

La única excepción a esta norma se deriva de la aplicación de dos de las operaciones dinámicas: una sentencia **alias** puede hacer público a algún canal o puerto privado, expandiendo su ámbito. Por su parte, la sentencia opuesta (**hide**) permite ocultar –hacer privado– a un puerto, sin que esta reducción de ámbito afecte a las conexiones ya establecidas.

En el caso de que se produzca una colisión de nombres, siempre prevalece aquel nombre que haya sido definido en el ámbito más cercano. A diferencia del cálculo-π, no se admite la *extrusión* de ámbito: un canal, existente o recién creado, no puede ser incluido como parámetro de una comunicación –en su mismo nivel–, aunque sí en una invocación explícita de proceso. En éste último caso, se considera simplemente que el proceso invocado se ejecuta dentro del mismo ámbito que tiene el proceso que lo invoca.

6.8.2 Replicación

El constructor de replicación, señalado en $\mathcal{P}i\mathcal{L}ar$ mediante las palabras **rep** o **bang**, no existe en Ccs, sino que fue definido específicamente para el cálculo-π poliádico [Mil93b]. Sin embargo, su definición en Ccs es perfectamente posible, ya que es en cierta medida equivalente a su operador de punto fijo **fix**, introducido en el álgebra con fines semánticos [Mil89], y que no ha sido incluido en esta sintaxis. Los detalles de esta relación se discuten al describir su semántica, en la sección 7.1 del capítulo siguiente.

La replicación expresa la repetición *infinita* de una secuencia de acciones determinada; en $\mathcal{P}i\mathcal{L}ar$ se asume que está *guardada*, esto es, que cada ejecución de esta secuencia se produce como respuesta al disparo de un *evento* de guarda, típicamente la recepción de un mensaje –aunque en un esquema síncrono, como éste, puede ser igualmente un envío–. Lo notorio es que las distintas secuencias se disparan *en paralelo*; es decir, no es necesario esperar a la terminación de la secuencia anterior para comenzar la siguiente. En este sentido, puede considerarse a la replicación como una "versión paralela" de la recursión.

Al igual que ocurría con la recursión, la replicación se define como infinita –y en el cálculo-π lo es de hecho–, pero sin embargo no lo es en $\mathcal{P}i\mathcal{L}ar$. Una replicación puede terminar, bien como consecuencia de la destrucción de la instancia que la define, bien como consecuencia de la destrucción de la reificación que indujo su ejecución. En cualquier caso, esto no afecta desde un punto de vista sintáctico; de nuevo, los detalles sobre este aspecto particular se discuten en la sección 7.1 de la especificación semántica.

Se ha de tener en cuenta que si una regla $\mathcal{P}i\mathcal{L}ar$ se define como puramente secuencial o paralela, esto es, si no es recursiva, ni está replicada, ni describe algún tipo de bucle, se ejecuta estrictamente *una vez*. Esto puede resultar particularmente útil en algunas circunstancias, como por ejemplo al realizar una inicialización. Una vez que el proceso ha terminado, el comportamiento de la instancia correspondiente se asume como totalmente inerte.

6.9 Acciones Elementales

Las acciones elementales de $\mathcal{P}i\mathcal{L}ar$, con la excepción de una, son las mismas de cualquier álgebra de procesos: básicamente, se trata de la emisión y recepción de mensajes a través de los *puertos* del componente. Su interpretación será la convencional en un álgebra de este tipo, asumiendo no obstante que toda emisión sobre un puerto se dirige al exterior, y todo mensaje recibido procede igualmente de su exterior. Es decir, un componente no puede *comunicarse* consigo mismo.

Un aspecto fundamental en este punto –y los siguientes– es el modo en que se realiza el acceso a los puertos. En este caso, como en el Lenguaje Estructural, se utiliza la clásica *notación punto* habitual en programación, que resulta suficientemente intuitiva. Así, cada componente *cualifica* a todos los puertos e instancias que contiene, y la notación puede aplicarse de manera reiterada. De este modo, una regla puede referirse al puerto P de una subinstancia C del componente actual, sin más que indicarlo como $C.P$. Cuando un puerto no está cualificado, se asume que está contenido en la interfaz del componente actual. Esta notación es, por lo demás, habitual en todo Adl, y totalmente convencional.

Acción	**Notación**	**Significado**
*puerto***?**(*variables*)	$p?(v)$	Recepción (Entrada)
when *puerto***?**(*variables*)	$\omega\,(p?(v))(P)$	Recepción No Destructiva
*puerto***!**(*datos*)	$p!(d)$	Emisión (Salida)
tau (*variables*)	$\tau\,(v)$	Silencio (Acción Interna)
P<*parámetros*>	$P\langle p\rangle$	Invocación de Proceso

Tabla 6.3: Acciones Elementales

Las acciones elementales de $\mathcal{P}i\mathcal{L}ar$ se describen en la Tabla 6.3. Se asume que todas, excepto la última, son atómicas, aunque pueden tener una duración asociada. En todas ellas, por otra parte, los parámetros son opcionales. Revisando concretamente cada una de ellas:

◊ *Recepción*: Se trata de la convencional recepción de un mensaje en una variable, o tupla de variables. La sintaxis utiliza la notación de Csp, muy común en los lenguajes concurrentes [PT00], y más cómoda en este contexto que la de Ccs. Como en cualquier álgebra, se

trata de un operador síncrono, lo que significa que el proceso espera a que el mensaje esté disponible, por lo que normalmente se puede usar esta sentencia como una *guarda*, esto es, como el evento que controla el disparo de una acción.

Se puede hacer también una recepción sobre un canal o variable interna, tal como fueron descritas en la sección 6.7.

◊ **when**: Este operador es, en cambio, completamente nuevo. Ya fue mencionado en la sección anterior, como un constructor similar a la estructura condicional clásica, que sustituye la condición por una recepción. Debe observarse que una recepción convencional puede describir igualmente una estructuras de este tipo, ya que establece –como se ha dicho– una guarda sobre el fragmento de proceso subsiguiente.

Su diferencia básica con la recepción convencional es que *no consume* el dato leído: es decir, obtiene el dato recibido en el puerto, pero éste no es eliminado, y queda aún disponible para una lectura posterior. Del mismo modo, tampoco se produce una sincronización con el proceso emisor. Esta acción puede realizarse a cualquier nivel, pero será especialmente interesante desde la perspectiva de un metacomponente sobre un avatar, ya que le permite seguir su interacción, sin afectarla en absoluto.

A diferencia de la anterior, esta operación sólo puede ser realizada sobre un puerto.

◊ *Emisión*: Se trata de la operación opuesta a la de recepción: se emite un mensaje, contenido en una variable o tupla, hacia el exterior a través de un puerto. También es síncrono, aunque puede producirse fácilmente una variante asíncrona utilizando el operador de composición paralela. Utiliza igualmente la notación Csp.

Se puede hacer también una emisión sobre un canal o variable interna, tal como fueron descritas en la sección 6.7.

◊ **tau**: Se trata de la acción *silenciosa*, que no produce ningún efecto visible exterior. Puede ser perfectamente eliminada de cualquier especificación, ya que no define ningún comportamiento: tan sólo se utiliza para expresar que en un punto determinado se produce una acción o proceso interno. Los parámetros (opcionales) se incluyen en $\mathcal{P}i\mathcal{L}ar$ –no forman parte de la sintaxis Ccs original–, y su único objetivo es expresar qué nombres resultan afectados por la acción interna, con fines de documentación y ligadura.

◊ *Invocación*: Se trata de una simple invocación de proceso, que permite iniciar la ejecución del mismo. Se admiten las invocaciones recursivas. En el caso de haber parámetros, se indican a continuación del nombre del proceso, encerrados entre ángulos.

Además de estas operaciones básicas, $\mathcal{P}i\mathcal{L}ar$ asume un soporte sintáctico para la *aritmética entera*, con sus respectivos operadores de suma, resta, producto, cociente y resto; y para el *álgebra de Boole* clásica, lo que incluye los operadores lógicos de conjunción, disyunción y negación, aparte de las constantes lógicas –**true** y **false**–. Junto a éstos, se definen igualmente los operadores relacionales convencionales; todos ellos pueden utilizarse de la manera usual en las estructuras condicionales e iterativas descritas en la sección anterior.

Operando	Notación	Significado
dato **in** *conjunto*	$d \in C$	El predicado de pertenencia tradicional
card(*conjunto*)	$\|C\|$	Cardinalidad (tamaño) del conjunto

Tabla 6.4: Operaciones sobre Conjuntos

Por otra parte, también se manejarán con cierta frecuencia operaciones sobre conjuntos: por ejemplo, los bucles definen una iteración sobre un conjunto de cualquiera de los tipos permitidos. No se proporcionan operadores de álgebra de conjuntos, pero sí dos operaciones sobre un conjunto, cuya sintaxis se describe en la Tabla 6.4. Son los siguientes:

◊ **in**: Se trata del clásico operador de pertenencia; puede verse como una función booleana que devuelve verdadero si el elemento pertenece al conjunto, y falso en el caso contrario.

◊ **card**: Proporciona la *cardinalidad*, esto es, el tamaño de un conjunto dado. Esto permite, por ejemplo, pasar de una iteración sobre un conjunto a una iteración numérica.

Aunque ambas operaciones son ocasionalmente útiles, son no obstante secundarias; su importancia es mucho menor que la de las restantes acciones descritas en esta sección.

6.10 Comandos Reflexivos

El Lenguaje Dinámico describe la evolución del sistema como la ejecución concurrente de las acciones indicadas en sus restricciones. Esto garantiza cierto grado de dinamismo; pero es la reificación la que permite desarrollar todo su potencial, al permitir que los metacomponentes manipulen a las instancias y enlaces de niveles inferiores *como si de datos se tratase*. Dado que las propias reificaciones pueden ser creadas y destruidas, la estructura se puede ampliar y alterar de manera indefinida; esto dota a $\mathcal{P}i\mathcal{L}ar$ de un dinamismo desconocido en otros Adls.

Se asume que el proceso se describe *desde la perspectiva del nivel meta* sobre el nivel base. Es decir, lo que se define en una restricción del Lenguaje Dinámico es el comportamiento que un metacomponente (**self**) impone sobre sus avatares, en primer lugar, y sobre sí mismo, en segundo lugar. Probablemente es importante reseñar aquí que éste no es, ni mucho menos, el modo habitual con el que se expresa reflexión en los lenguajes de programación. De hecho, en éstos es más frecuente encontrar el planteamiento opuesto, según el cual una entidad-base es la que provoca una reacción en niveles superiores.

Sin embargo, utilizar este último enfoque en $\mathcal{P}i\mathcal{L}ar$ sería totalmente incongruente. Debe tenerse en cuenta que el comportamiento de una instancia se define junto con su arquetipo, por la propia concepción de tipo; pero además, este arquetipo es también un metacomponente para esa instancia. Por tanto, una estrategia *ascendente*, de base a meta, resulta en un extraño circunloquio, según el cual se debería definir el comportamiento de las instancias de manera independiente, pero sin embargo dentro del arquetipo, y éste sólo caracteriza su propia operación como una reacción a las acciones de aquéllas.

Por su parte, el planteamiento *descendente* elegido hace exactamente lo mismo, pero presenta una conceptualización mucho más coherente. Según ésta, las restricciones de un tipo definen el comportamiento de sus instancias, lo que significa que cada una de ellas tiene una copia independiente del mismo proceso. Las acciones propias del metacomponente se entrelazan en esa misma definición de proceso; de este modo, el protocolo entre niveles queda establecido sin ninguna clase de ambigüedad. Así, una restricción puede leerse como una descripción del comportamiento que un tipo impone sobre *cada una* de sus instancias, y de la relación que establece para sí mismo con respecto a ellas.

Puede argumentarse, por supuesto, que este enfoque liga directamente al metacomponente con sus instancias, y describe su comportamiento como la composición de múltiples definiciones concurrentes; pero podría ser necesario definir parte de dicho comportamiento de manera unívoca, e independiente. Sin embargo, la respuesta es clara: si esa parte no depende de la relación con las instancias, entonces no es realmente *meta*-comportamiento, sino operaciones propias del componente, visto ahora como *instancia de su propio tipo*. Por tanto, deberán ser descritas como restricciones de un meta-arquetipo, esto es, de un meta-metacomponente.

La divergencia con la práctica habitual se vincula, simplemente, al hecho de que $\mathcal{P}i\mathcal{L}ar$ no es un lenguaje de programación, sino de *especificación*. La visión ascendente de la reflexión resulta más sencilla cuando lo que se pretende es poder modificar fácilmente aspectos concretos de un sistema ya construido; pero en un Adl el objetivo es describir la estructura del sistema desde una perspectiva global, por lo que un enfoque descendente es mucho más coherente.

Los comandos de este tipo han sido divididos en tres grupos: *operandos* reflexivos, operadores dinámicos y *operadores* reflexivos. Éstos son descritos en las secciones siguientes:

6.10.1 Operandos Reflexivos

En este grupo se reunen una serie de palabras clave que actúan como *identificadores* o *cualificadores*, de modo que permiten recorrer el esquema de meta-niveles, siempre de arriba a abajo, y con la perspectiva que ya se ha descrito.

Operando	Notación	Significado
avatar	α	Elemento del avatar (instancia)
self	γ	Elemento del metacomponente (arquetipo)
avatarSet($\underline{m}$)	$\Sigma\alpha(\underline{m})$	Conjunto de avatares de $\underline{m}$
portSet($\underline{c}$)	$\Sigma\pi(\underline{c})$	Conjunto de puertos públicos de $\underline{c}$
bound($\underline{a}$)	$\Sigma\beta(\underline{a})$	Conjunto de instancias enlazadas con $\underline{a}$

Tabla 6.5: Operandos Reflexivos

Los operadores se recogen en la Tabla 6.5, y se dividen en dos grupos: el de aquellos que designan a una entidad, y el de los que hacen referencia a todo un conjunto. Son los siguientes:

◊ **avatar**: Es un identificador que hace referencia a *cada uno* de los avatares del metacomponente actual, esto es, el que define esta regla. Implica, pues, un descenso de nivel. Se utiliza habitualmente como cualificador, prefijando el nombre de un puerto, instancia o

conexión, de modo que se indique explícitamente que está en el nivel inferior, de modo que, según el Principio de Estratificación, es manipulable por cualquiera de los operadores del apartado siguiente, en particular los de creación y borrado.

Dado que en el caso más habitual las reglas se refieren solamente a cada una de las instancias de su tipo, *este prefijo puede omitirse*; en tal caso, el significado del nombre cualificado es el intuitivo, esto es, hace referencia a un puerto o sub-instancia de la instancia actual –es decir: del propio avatar–.

Este prefijo puede aplicarse un número indeterminado de veces; cada vez que se usa, implica un descenso de nivel. De este modo, por ejemplo, avatar.avatar hace referencia a un avatar de un avatar del tipo definido; es decir, el arquetipo es un meta-metacomponente de la entidad referenciada. Al igual que antes, hace referencia en realidad a *cada* uno de los avatares de *cada* avatar del arquetipo actual.

◊ **self**: Por contra, este prefijo hace referencia al propio metacomponente, es decir, a la parte activa del arquetipo, visto como una instancia en el nivel meta. Se utiliza, igualmente, como un cualificador para los elementos del metacomponente: así, permite acceder a los puertos del metafaz, o a cualquier subinstancia en el propio nivel meta. En cierto sentido, estos elementos pueden verse como los análogos de una *variable de clase*, y en cualquier caso definen un espacio común para todos sus avatares.

En definitiva, una restricción se lee desde la perspectiva del metacomponente pero se aplica individualmente a cada avatar. Puede considerarse que cada avatar –instancia– tiene su propia copia de la regla –de hecho, así será–, de modo que en cada caso, el cualificador **self** se refiere a su rohatar, y el **avatar** a sí mismo. En resumen, se está aplicando la misma regla a todos los avatares, desde la perspectiva del metacomponente, pero en cada caso se distingue el avatar al que se está aplicando.

Por otra parte, los referidos a conjuntos son:

◊ **avatarSet**: Su parámetro es opcional; si se omite, se refiere al metacomponente actual. Dado un metacomponente, devuelve como resultado el conjunto de *todos* sus avatares. Este conjunto podrá después ser utilizado en una iteración, típicamente, o se le aplica alguna de las operaciones de conjuntos ya definidas.

◊ **portSet**: Dado un componente cualquiera, devuelve el conjunto de todos sus puertos públicos, esto es, su interfaz. A pesar de su sencillez, sigue siendo un operando reflexivo, ya que lo que hace es proporcionar meta-información sobre la estructura del componente.

◊ **bound**: Puede aplicarse a un puerto, a una instancia o a una conexión, manifestando ante estos dos últimos un comportamiento prácticamente idéntico. Cuando es un puerto, puede serlo de una instancia o de una conexión: en el primer caso, devuelve el conjunto de conexiones que ligan a la instancia en este puerto; en el segundo, devuelve el conjunto de instancias enlazadas por la conexión a través de este puerto. En ambos casos, el conjunto puede serlo de más de un elemento, ya que $\mathcal{P}i\mathcal{L}ar$ admite las conexiones múltiples.

Cuando se aplica a una instancia, el significado es el mismo, extendido a toda la entidad: devuelve el conjunto de conexiones que ligan a esta instancia, a partir de cualquiera de sus

puertos. Análogamente, cuando se aplica a una conexión, proporciona el conjunto de todas las instancias ligadas por ella, a través de cualquiera de sus puertos. Es obvio que este significado más amplio puede deducirse simplemente a partir del relativo a los puertos, por una simple unión de conjuntos –que en $\mathcal{P}i\mathcal{L}ar$ debería desplegarse como una iteración–. En cualquier caso, es claramente un operando reflexivo, ya que proporciona al sistema un medio para estudiar su propia conectividad.

El objetivo y uso de cada uno de ellos es obvio. Todos ellos admiten una enorme variedad de usos. Se podrían definir muchos más operandos de este tipo; pero la mayoría de ellos pueden definirse simplemente como derivados de éstos.

6.10.2 Operadores Dinámicos

Estos operadores, en realidad, son mayoritariamente independientes de la reflexión; proceden en su mayoría de la tradición de los Lenguajes de Configuración y Mils, donde se describen por sí solos. No obstante, se incluyen en este grupo porque, según el Principio de Estratificación, sólo pueden ser aplicados a una entidad de nivel inferior; y porque, desde esta perspectiva, pueden ser aplicados a *cualquier* entidad de este nivel, sea una instancia, un puerto o una conexión. Esto permite obviar la distinción, por ejemplo, entre *borrar* y *desligar*.

Operador	Notación	Significado
new($\underline{a} : \underline{T}$)	$\nu(\underline{a} : \underline{T})$	Creación de una nueva entidad $\underline{a}$ de tipo $\underline{T}$
del $\underline{a}$	$\delta\, \underline{a}$	Destrucción de una entidad $\underline{a}$
alias p **as** q	$\Delta p(q)$	Extrusión (publicación) de un puerto p
hide p	∇p	Ocultación (privatización) de un puerto p
shift *puerto*(*datos*)	$p \downarrow (d)$	Emisión hacia el puerto p del nivel inferior

Tabla 6.6: Operadores Dinámicos

Se agrupan en la Tabla 6.6, y se describen como sigue:

◊ **new**: Esta operación permite *crear* un nuevo elemento. Éste tendrá asignado un tipo, según el cual será un puerto, un componente o una conexión –incluso un enlace básico–. Si se crea sin tipo, será simplemente un canal, que se define como privado al proceso actual, y que no puede ser difundido por extrusión –es decir, se cumple la regla de ámbito–.

◊ **del**: Esta operación *destruye* –borra– a cualquier elemento que se le pase como parámetro, sea un puerto, instancia o conexión. Si es un componente compuesto, se destruyen también todos los elementos de su interior.

De hecho, como se verá en el apartado siguiente, el operador de destrucción (**del**) puede aplicarse también a reificaciones; no obstante, en este caso está a un nivel de abstracción diferente, por lo que no considera en este punto. Aparte de los dos operadores básicos, existen también otros operandos "menores", que pueden servir a propósitos diversos:

◊ **alias**: Permite realizar la extrusión de ámbito de un puerto dado, e incluso cambiar su nombre en el proceso. De este modo, un puerto es incorporado a la interfaz del componente actual, bien porque no pertenecía a la misma, bien porque estaba oculto.

◊ **hide**: Hace el proceso inverso: dado un puerto ubicado en la interfaz del componente actual, lo *oculta*, de modo que no está disponible para nuevas conexiones. Sin embargo, las conexiones ya existentes no se destruyen; en este sentido, pueden seguir interactuando.

◊ **shift**: Se trata de un comando esencialmente algebraico, con una naturaleza enteramente distinta. Se incluye en este grupo porque sólo puede aplicarse en un esquema reflexivo. Su función es *emitir* un mensaje hacia un puerto del avatar, que se indica como parámetro; pero este mensaje, en lugar de dirigirse hacia el exterior, como en una emisión normal, se recibirá en el nivel inferior como una entrada. En definitiva, permite simular *falsas entradas* desde el nivel meta. Su utilidad será diversa; pero se concibe especialmente para la definición de envolventes (§6.12.1).

Ninguno de estos comandos es básico, pero todos son útiles de algún modo, y perfectamente coherentes con la estructura reflexiva definida. En cualquier caso, no suponen un rasgo esencial del lenguaje, aunque todos ellos son una aportación específica de $\mathcal{P}i\mathcal{L}ar$.

6.10.3 Operadores Reflexivos

Finalmente, bajo el nombre de *operadores reflexivos* se agrupan todos aquellos que no utilizan, sino que *manipulan* a la estructura de reificaciones que define los meta-niveles, y por ende todo el comportamiento reflexivo del sistema.

Operador	Notación	Significado
reify $\underline{r}(\underline{a} : \underline{m})$	$\rho\, \underline{r}(\underline{a} : \underline{m})$	Crea una reificación $\underline{r}$ entre $\underline{a}$ y el metacomponente $\underline{m}$
findr $\underline{r}(\underline{a} : \underline{m})$	$\phi\, \underline{r}(\underline{a} : \underline{m})$	Busca una reificación entre $\underline{a}$ y el metacomponente $\underline{m}$
nullr $\underline{r}$	$\varepsilon(\underline{r})$	Booleana: verdadero si $\underline{r}$ existe y es una reificación
del $\underline{r}$	$\delta\ \underline{r}$	Borra un elemento, en este caso una reificación.

Tabla 6.7: Operadores Reflexivos

Se recogen en la Tabla 6.7, y se describen como sigue:

◊ **reify**: Crea un enlace de reificación entre un avatar $\underline{a}$ y un rohatar $\underline{m}$ ya existente. Su efecto es el mismo que el de su homónimo del Lenguaje Estructural. La reificación puede alterar simplemente el comportamiento del avatar, o incluso inducir la creación de nuevos elementos en su interior, desde puertos a sub-instancias. Todas las reificaciones reciben específicamente un nombre, aunque pueden ser también identificadas por el par avatar-rohatar formado, ya que sólo puede haber una entre dos elementos concretos.

Al igual que en Marmol, la existencia de reificaciones previas no supone ningún problema: un avatar puede ser reificado múltiples veces por metacomponentes distintos.

◊ **del**: Se trata del mismo comando dinámico ya citado. Sin embargo, se repite aquí porque puede aplicarse igualmente a reificaciones, siempre que se conozca su nombre. Esto no sólo incluye a las creadas con una orden **reify** en una restricción dinámica, sino también a todas aquéllas que hayan sido creadas *con nombre* en la sección correspondiente de una declaración –estática– de componente. Esto excluye, por supuesto, a las reificaciones implícitas procedentes de una definición de tipo.

Su único efecto es romper el vínculo entre el avatar y el metacomponente, de modo que este último deja de ejercer su influencia sobre el primero. No implica la destrucción de ninguno de ellos, pero sí la de todo elemento –puertos, subinstancias– cuya existencia en el avatar se debiera a esta reificación. Por supuesto, se mantiene el efecto de las acciones ya realizadas sobre otros elementos.

◊ **findr**: Permite determinar si existe una reificación entre un avatar $\underline{a}$ y un metacomponente $\underline{m}$. Es decir, permite averiguar si $\underline{a}$ es un avatar de $\underline{m}$. El resultado se recibe en una variable de reificación, que podrá luego ser utilizada.

◊ **nullr**: Es un predicado booleano, que pretende determinar la nulidad de la variable de reificación obtenida en la operación anterior. Devuelve verdadero si la reificación es *nula*, esto es, si no había reificación entre ellos, y falso si se trata de una auténtica reificación, que puede ser en tal caso usada posteriormente.

El uso de las dos primeras resulta obvio a la luz de todo lo dicho con anterioridad. En cambio, el de las dos últimas es menos claro. En resumen, la idea es que si se establece una cierta analogía –no estricta– entre tener un tipo y estar reificado, en muchos aspectos preguntar a un avatar por sus reificaciones equivale a preguntarle por su tipo, lo cual tendrá una indudable utilidad en un esquema dinámico como el definido por $\mathcal{P}i\mathcal{L}ar$.

Resulta relevante mencionar que, en este punto, existen *dos* opciones diferentes en lo que respecta al diseño del lenguaje. Una vez que se plantea la tarea de localizar un enlace de reificación entre dos instancias de niveles adyacentes, puede verse con claridad que ésta consta en realidad de dos partes. En primer lugar, ha de determinarse si dicho enlace *existe* siquiera; sólo entonces, en el supuesto de que así sea, debe ser efectivamente obtenido.

Hay dos formas de dividir esta tarea. La primera, aparentemente la más obvia, es definir cada subtarea como una acción independiente. De este modo, se dispondría de una operación booleana, capaz de establecer la existencia de la reificación, y de una búsqueda propiamente dicha, que localizase el enlace requerido. No obstante, resulta patente que ambas operaciones no son ortogonales, sino que en realidad hacen el mismo recorrido. La segunda alternativa, que es la que de hecho se ha utilizado, toma en consideración este hecho, y define una única operación de búsqueda (**findr**), que siempre obtiene una reificación como resultado. Pero si el enlace buscado no existía, el dato devuelto es entonces la *reificación nula*, esto es, un resultado no válido, que no admitirá el tratamiento habitual. Se debe proporcionar, por tanto, otra operación (**nullr**), que permita determinar si la reificación localizada es o no la reificación nula.

La razón por la que $\mathcal{P}i\mathcal{L}ar$ se decanta por la segunda opción es en realidad semántica, no sintáctica. Puede verse que en el primer caso la misma búsqueda se realiza dos veces, mientras que en el segundo tan sólo se hace una vez, y se determina luego la validez de su resultado;

por tanto, éste último es internamente mucho más eficiente. Esta afirmación será examinado con mayor detenimiento durante la descripción semántica de este aspecto, contenida en la sección 7.4.6. Por contra, los argumentos sintácticos a favor de una u otra opción son bastante débiles. Ambas constan de dos operaciones, y son igualmente incómodas: en la primera, toda búsqueda debe ir precedida de la comprobación de la existencia; en la segunda, toda búsqueda ha de ir seguida de una verificación de su resultado. La alternativa elegida tiene, sin embargo, otra ventaja: siempre se devuelve un valor significativo, lo que no está garantizado en el caso contrario.

6.11 Conectores de Nivel Meta

Se ha dicho en varias ocasiones que $\mathcal{P}i\mathcal{L}ar$ ha sido diseñado, no sólo con el objetivo de introducir las nociones reflexivas en la Arquitectura de Software, sino también con el de sintetizar en un único lenguaje las capacidades ya existentes en otros ADLs, al menos en parte. Con este planteamiento, puede resultar sorprendente el hecho de que tanto en MARMOL (ya desde el modelo $\aleph_\Sigma$) como en $\mathcal{P}i\mathcal{L}ar$ se haya suprimido un concepto tan popular en el campo como el de *conector*. Sin embargo, este es posiblemente uno de los enfoques más interesantes del lenguaje, aunque no resulta totalmente ajeno a cierta polémica.

Como ya se señalaba en la sección 2.4.3, Mary Shaw fue la primera en concebir a los conectores como entidades *de primera clase* en la descripción arquitectónica [Sha94]. Esta idea ha sido ampliamente aceptada dentro del campo, hasta el punto de ser considerada a menudo como una de las características definitorias de un ADL [Med96b, MT97a, MT00], así como una noción prácticamente indiscutible. Sin embargo, como también se ha dicho (§2.4.3), muchos de los ADLs de primera línea no la utilizan, o incluso la rechazan de manera explícita. Más aún, entre ellos se encuentran algunos de los ADLs dinámicos más relevantes, como $\delta arwin$, Rapide o LEDA. Se puede afirmar, en realidad, que el concepto no es sólo uno de los más conocidos de la Arquitectura de Software, sino también, con toda seguridad, el más discutido.

Como se ha dicho (§2.4.3), varios ADLs distinguen entre *conector* (interacción compleja) y *adjunción* o *conexión* (enlace directo). Otros no utilizan conectores, y en su lugar emplean componentes normales ("componentes de interacción") adaptados a este fin. Wright, C2 o Unicon consideran a los conectores esenciales para la arquitectura; pero otros lenguajes ($\delta arwin$, LEDA) los rechazan completamente, aduciendo que complican los aspectos composicionales.

En $\mathcal{P}i\mathcal{L}ar$ –como en MARMOL–, los componentes y conectores tienen la misma forma, pero se mantienen diferenciados. De este modo, se conserva tanto la Composición como la Separación de Intereses. Para hacerlo, se apoya en la estructura multinivel que define el propio marco: los *componentes* normales y las *conexiones* o enlaces se definen en el nivel-base. Estos enlaces se reifican como estructuras "complejas" en el nivel meta. Así pues, *los conectores son meta-espacios*, o en su forma mínima, incluso *un* único meta-componente.

Varias conexiones se pueden corresponder con un único metaespacio, interactuando internamente de modo transparente. Una única conexión puede ser reificada por distintos metaespacios a lo largo del tiempo, adquiriendo en cada caso distintas propiedades. Un enlace puede ser destruido, pero su metaespacio puede regenerarlo, proporcionado tolerancia a fallos. La flexibilidad que se logra de este modo no es igualada por los conectores de ningún otro ADL. En realidad,

$\mathcal{P}i\mathcal{L}ar$ no pierde generalidad al hacer esta elección; incluso si se prefiere el enfoque tradicional de conectores –por ejemplo, el de Wright–, siempre se pueden utilizar componentes normales para expresar los detalles de interacción, y enlazarlos utilizando conexiones "planas".

Es importante señalar que, en el meta-espacio, no hay necesidad de hacer nada especial con los conectores: se les representa exactamente igual que a cualquier otro componente. Esto no es sólo por coherencia, sino que también hay una razón importante para hacerlo así: de este modo, *todo es un componente* en nuestro modelo. El distinto significado de los distintos componentes viene dado únicamente por *su posición* en la arquitectura.

Este enfoque de *conectores de nivel meta* no es en absoluto extraño: por el contrario, es posiblemente el que mejor refleja la realidad. Según la propuesta de Shaw, un conector puede ser tanto una simple llamada a procedimiento local como una Rpc compleja. El diseñador de un componente no ha de preocuparse de ese aspecto, tan sólo ha de conocer el protocolo necesario. Sin embargo, al hacer los conectores explícitos, en muchos casos se están especificando al mismo nivel de la aplicación conceptos mucho más generales, que en realidad se sitúan en un nivel de abstracción diferente. Para un programador, el mecanismo de una Rpc es "parte del lenguaje", y no lo considera nunca como un elemento propio de la aplicación. Lo usa, de hecho, pero implícitamente, como parte de su vocabulario: en resumen, en su *nivel meta*.

En realidad, no hay una verdadera diferencia entre la representación clásica de un conector y la que se utiliza en $\mathcal{P}i\mathcal{L}ar$, salvo la pérdida de la dualidad innecesaria. La especificación del comportamiento de un conector –por ejemplo, en Wright– es muy similar a su análogo en $\mathcal{P}i\mathcal{L}ar$, esto es, la definición de las restricciones en el metacomponente de una conexión compleja. El único aspecto significativo es que en Wright se trataba de una especificación de un protocolo abstracto, mientras que en $\mathcal{P}i\mathcal{L}ar$ es de hecho el comportamiento de un componente efectivo, y se enfatiza que está *ubicado en el nivel meta*.

Por otra parte, esta propuesta no contradice la de Shaw. Su motivo al proponer conectores de primera clase era hacer explícita la interacción, que hasta entonces se indicaba simplemente como "una línea de conexión". El objetivo no era, pues, situarlos como una entidad diferenciada de los componentes. En $\mathcal{P}i\mathcal{L}ar$, al hacer explícito el nivel meta, las propiedades de los elementos base pueden hacerse también explícitas: esto incluye a las características de interacción de una conexión. Por tanto, no se necesita introducir a los conectores como entidades *de primera clase*, cuando ya es posible representarlos como conceptos *del nivel superior*.

Dado que no se tienen conectores, sino únicamente componentes –del nivel meta–, estos pueden ser *compuestos* como sea preciso. Esto hará posible, a medio plazo, la definición de un *álgebra de conectores*, capaz de componer conectores primitivos para crear *conectores de orden superior* [Gar98]. Con el enfoque clásico, la composición de conectores necesitaba un componente intermedio; y la naturaleza de tal agrupación siempre resultaba confusa[3]. En $\mathcal{P}i\mathcal{L}ar$, este problema desaparece; la cuestión se reduce a elaborar un vocabulario de conectores lo bastante completo, y a hacer un estudio de sus posibilidades de interacción, basado en la experiencia.

El resultado final de la aproximación de $\mathcal{P}i\mathcal{L}ar$ al concepto de conector resulta, finalmente, algo paradójico. Inicialmente era rechazado, debido a su ambigüedad semántica. Posteriormente se lo recuperaba, no como una noción *de primera clase*, sino como un meta-concepto; sin em-

[3]Un ejemplo muy claro, debido a Carlos Canal, es el siguiente: si una cola es un componente y un canal de mensajes es un conector, ¿qué es una cola de mensajes, o un canal con memoria?.

bargo, dado que en $\mathcal{P}i\mathcal{L}ar$ el nivel meta se hace explícito, el conector recupera su importancia anterior. En este punto, el lenguaje tiene lo mejor de ambos enfoques: sin hacer forzosa la definición de conectores, se mantienen todas las ventajas derivadas de su especificación.

Pero todavía se puede llegar más lejos en este razonamiento. Como se verá en el Capítulo 7, la definición semántica de *cualquier* conexión en $\mathcal{P}i\mathcal{L}ar$ toma una forma análoga a la de una instancia de componente; de hecho, idéntica a nivel de tipos. Esto es cierto incluso para las más sencillas, esto es, los enlaces simples y jerárquicos. Se podría decir que en cierto modo una arquitectura no consta, como parece, de un conjunto de componentes ligados mediante conexiones establecidas entre sus puertos: sino que, en realidad, lo que se tiene es *un conjunto de componentes, de distintos tipos, unidos* directamente *por sus puertos*. Curiosamente, esta visión no es totalmente nueva: es muy similar a la que aplica Hirsch [HIM99, HM99] en su modelo de hipergrafos, que ya fue descrito en la sección 3.9.3. En él, todos los elementos son también componentes –hiperarcos– unidos a través de sus puertos –nodos–. Existe también cierto parecido con las semánticas de otros ADLs algebraicos sin conectores, como PADL o LEDA. En ambos casos, sin embargo, el motivo es exactamente el contrario: aquí se identifican los puertos con canales, en lugar de los conectores con componentes –esto es, procesos–.

Como ya se ha expuesto con anterioridad (§4.7), la introducción de *conectores reflexivos* no es una sugerencia totalmente nueva. Por el contrario, ya ha sido utilizada con anterioridad, tanto en la propia Arquitectura de Software como en el contexto de los sistemas generativos y los modelos de componentes; incluso, en términos generales, en muchos de los planteamientos concurrentes de la orientación al objeto [Hol99].

Algunos ejemplos notables de este tipo de conectores incluyen, entre otros, a los *conectores instrumentados* de Bob Balzer [Bal97], los *conectores abiertos* de Uwe Aßmann [ALP99], los *conectores ejecutables* del modelo FLO de Stéphane Ducasse y Tamar Richner [DR97], los *multicanales* del sistema mChaRM de Walter Cazzola [Caz00], los *controladores* del modelo $\mathcal{SC}$ de Antonio Vallecillo [VM99], los *roles coordinados* de Juan Manuel Murillo [MHSA98], los *filtros de composición* de Sina, desarrollados por Mehmet Akşit y Ludwig Bergmans [AWB+93, Ber94, BA00], o las *capas* del modelo LAYOM de Jan Bosch [Bos96c], basadas en éstos últimos. En realidad, no en todos los casos hay una correlación perfecta entre el tipo de intermediario introducido y el concepto arquitectónico de conector, pero en líneas generales hay una correspondencia bastante clara. De hecho, incluso podrían considerarse en este contexto algunos enfoques conceptualmente más distantes, como las implementaciones de los *contratos de reutilización* [Luc97], o la gran mayoría de los modelos de capas de cebolla (§4.6).

Sin embargo, ha de tenerse en cuenta que en casi todas las ocasiones, éstos consisten simplemente en *implementaciones* que, bien utilizan técnicas reflexivas para cumplir su función, bien generan su propio código a partir de una descripción *arquitectónica*. En todo caso, no utilizan las capacidades reflexivas de las que disponen *al nivel de la descripción*, por lo que no son comparables a la aproximación que aquí se ha utilizado.

Merecen una mención aparte los *conectores* de C2SADEL [OMT98] (§3.8.2), ya que constituyen un caso particularmente curioso. Según su propia definición, manifiestan lo que se denomina *reflexión contextual*: su interfaz se redefine en cada momento según su contexto actual, esto es, según los componentes que tenga conectados. No se trata, en realidad, de conectores reflexivos, sino más bien de una especie de *bus común* para la difusión de eventos. En todo caso, están

muy restringidos, ya que quedan limitados por el estilo C2.

En resumen, se podría terminar concluyendo que, aunque existen otros conectores definidos en el nivel meta, las conexiones reificadas de $\mathcal{P}i\mathcal{L}ar$ son, posiblemente, los únicos *conectores de nivel meta* propiamente dichos.

6.12 Extensiones Específicas

Durante el diseño de $\mathcal{P}i\mathcal{L}ar$ se ha utilizado un criterio lo más amplio y genérico posible, con el objetivo de introducir únicamente el número preciso de abstracciones y operaciones, e intentando con ello obtener el mayor grado de expresividad. En este sentido, consta únicamente de una serie de abstracciones fundamentales: configuración e instanciación, conexión e interacción, y reificación; y de unas acciones elementales, que se reducen al envío y recepción de mensajes, la invocación de procesos, y la creación y destrucción de elementos.

No se han tenido en cuenta, por tanto, consideraciones de índole más particular, referidas a algún tipo de operación o estilo de comunicación concreta. Debe recordarse, además, que ésta es parte de la tarea del propio $\mathcal{P}i\mathcal{L}ar$; ha de ser capaz de describir estas interacciones y estilos en su propio lenguaje, sintetizar con ellos una definición de componente –o conector–, y de este modo incorporarlas al vocabulario básico utilizable.

Sin embargo, es también evidente que la incorporación de algunas de estas operaciones como elementos primitivos del lenguaje facilitaría considerablemente la descripción, no ya de un estilo concreto, sino de un amplio rango de sistemas. Operaciones de este tipo podrían ser, por ejemplo, un soporte para la difusión implícita de eventos, análogo a los mecanismos de Rapide (§3.8.1) o Manifold (§3.12.1), o un tipo de enlace con interacción privada, similar a los que se establecen durante el uso de $\delta arwin$ (§3.7.2) o Leda (§3.7.3).

No se plantea, en esta tesis, la incorporación de este tipo de extensiones. Sin embargo, es evidente que esto es perfectamente posible, máxime cuando la propia descripción $\mathcal{P}i\mathcal{L}ar$ se podría utilizar como base de la correspondiente definición semántica –hecha, a su vez, en el lenguaje del cálculo-π–. No obstante, existe al menos un caso en el que una incorporación de este tipo resulta interesante, ya que guarda cierta relación con el objetivo general de este trabajo.

Esta incorporación es la de un soporte explícito para *envolventes* en el lenguaje. Se trata, por una parte, de una abstracción bastante difundida, en la que se elabora una estructura compleja; por tanto, es el tipo de sistema en el que una descripción arquitectónica adecuada resulta especialmente beneficiosa. Por otra parte, la construcción de envolventes reflexivos, basados en el modelo de capas de cebolla (§4.6), es posiblemente el uso más popular de la Reflexión; así pues, parece razonable suponer que su especificación en $\mathcal{P}i\mathcal{L}ar$, definido como un Adl reflexivo, sea una tarea habitual. En consecuencia, cualquier extensión que vaya en la línea de facilitar esta tarea será particularmente interesante.

En resumen, en este apartado se considerará sólo *una* extensión de la sintaxis base de $\mathcal{P}i\mathcal{L}ar$, es decir, la que se refiere al soporte para envolventes reflexivos. El objetivo principal es mostrar el modo en que el lenguaje se adapta fácilmente a una nueva estructura; de este modo se sugiere que la incorporación de otras extensiones no habría de resultar excesivamente compleja. En cuanto a la extensión en sí, por otra parte, se sitúa solamente como un objetivo secundario. Sin embargo, ha sido seleccionada por su interés específico, su frecuencia y su sencillez: como se

verá, su impacto final en la sintaxis del lenguaje será mínimo.

6.12.1 Soporte para Envolventes

En este apartado, se describe una extensión de $\mathcal{P}i\mathcal{L}ar$ que facilita la construcción de *envolventes* reflexivos, basados en el modelo de capas de cebolla, cuya filosofía es similar a la ya expuesta en la sección 4.6. Se trata, simplemente, de "rodear" a un componente, de modo que todas sus interacciones externas hayan de pasar previamente por un intermediario. Al igual que en el modelo informal, la relación entre un componente y su envolvente se establece por reificación: un avatar –apto para ser envuelto– es *reificado* por un metacomponente, y en este proceso es también *envuelto* por éste, que desde este momento interceptará –e intervendrá– todas sus recepciones, y capturará –y alterará– todas sus emisiones.

Ha de señalarse que el concepto de conector –incluso el de conexión reificada– ofrece una potencia expresiva similar; más aún, porque permite particularizar un envolvente en cada puerto. Sin embargo, el enfoque de diseño es diferente, ya que un envolvente es concebido como un módulo completo, mientras que en un componente estaría disperso. Por tanto, es interesante disponer de una sintaxis para ambos, lo que permite –entre otras cosas– poder compararlos.

En realidad, como ya se ha dicho, la definición de envolventes en $\mathcal{P}i\mathcal{L}ar$ no necesita de un soporte externo; el lenguaje es, por sí solo, lo suficientemente potente como para poder describirlos, en cualquiera de las formas que adoptan. Esto resulta perfectamente claro por comparación con el enfoque de Robert DeLine, que en su tesis doctoral [DeL99] elabora un soporte general para *wrappers* basado en Unicon. En su modelo, denominado *Empaquetamiento Flexible*, se identifican dos tipos de entidades: los *wares*, o componentes a envolver, y los *empaquetadores* (*packagers*), que realizan la función de envolventes. Para ello, éstos últimos han de poder especificar de manera explícita los protocolos de interacción de los componentes, un aspecto que en Unicon no era considerado [SDK+95a]. Por tanto, se hace preciso definir a los empaquetadores en un lenguaje alternativo, denominado *Ciao*, que simplemente aporta tres construcciones adicionales, a saber: operaciones de envío y recepción de mensajes (*out*, *in*), y una estructura selectiva no determinista (*alt*). Las tres se sitúan de manera explícita en un contexto concurrente, y están reconocidamente inspiradas en la especificación de puertos de Wright [All97]; se trata, en definitiva, de las acciones de salida, entrada y selección de Csp, expresadas en la sintaxis de Occam y analizadas posteriormente mediante la herramienta Fdr.

DeLine muestra que su modelo es suficiente para describir un catálogo completo de –ocho– tipos de envolvente [DeL99], cuando en realidad se limita a rodear un componente Unicon con una capa externa, análoga a las definiciones de Wright. Es obvio que en $\mathcal{P}i\mathcal{L}ar$, al igual que en el propio Wright, no es necesario recurrir a un segundo lenguaje, ya que todas las construcciones necesarias están en la sintaxis de partida; de este modo, se obtiene una mayor coherencia. Más todavía, no sólo se puede indicar el envolvente como un recubrimiento sobre el componente original, sino que puede ser incorporado dinámicamente en la especificación mediante reificación; obviamente, esto no es posible en el modelo de DeLine, ni en el propio Wright.

No obstante, la definición de envolventes en $\mathcal{P}i\mathcal{L}ar$ tiene sus propios problemas, derivados por otra parte de la naturaleza intrínsecamente concurrente del lenguaje. Como podrá verse en el ejemplo 8.6, el lenguaje dinámico permite especificar de manera detallada la interacción entre el componente envuelto y su envolvente; sin embargo, y como allí se indica, existe en este

caso un elevado potencial para el indeterminismo, que habrá de ser evitado de manera explícita mediante la definición de un protocolo específico.

Este protocolo puede alcanzar un notable grado de complejidad, y sobre todo no es fácilmente escalable. Por este motivo, resulta interesante plantear la definición de una sintaxis específica para esta abstracción, de modo que los posibles conflictos entre niveles se resuelvan en el plano semántico, y que se pueda asegurar la consistencia de las suposiciones de partida.

Operador	Notación	Significado
wrap	—	Declara el componente actual como *envuelto*
catch *puerto*(*variables*)	$p \uparrow (v)$	Captura la salida en el puerto p del nivel inferior
letin p	$\Pi\, p$	Habilita *una* recepción en el puerto p
letout p	$\amalg\, p$	Habilita *un* envío a través del puerto p

Tabla 6.8: Soporte para Envolventes

Como ya se ha dicho, el impacto del soporte para envolventes en la sintaxis es mínimo. Tan sólo consta de *cuatro* palabras reservadas, tres de las cuales tienen un comportamiento casi trivial, especialmente a efectos de la sintaxis. Estas palabras clave, así como la notación algebraica correspondiente –si procede– se muestran en la Tabla 6.8. Junto a ellas, una de las instrucciones del lenguaje básico (**shift**) tendrá también un papel fundamental; no obstante, su alcance va más allá de esta extensión.

A continuación se proporciona una descripción más detallada de esta sintaxis:

◊ **wrap**: No se trata de un operador, sino de un cualificador y, a diferencia de los otros tres comandos, forma parte del *Lenguaje Estático*. Se utiliza, por tanto, dentro de éste, como una etiqueta asociada a una declaración de (tipo de) componente, del modo siguiente:

$$\backslash\textbf{component}\ \backslash\textbf{wrap}\ \textit{nombre} < \textit{parámetros} > (\ \ldots)$$

Su efecto es declarar a este componente como *apto para ser envuelto* o, mejor aún, simplemente *envuelto* (*wrapped*). De hecho, cualquier instancia de este tipo no será capaz de interactuar con otros componentes a menos que sea envuelta, o que su propio arquetipo tenga en cuenta este aspecto. La declaración afecta a un componente *bloqueando cualquier interacción externa* a través de todos sus puertos, de modo que una instancia de este tipo no podrá enviar o recibir ningún mensaje, a menos que sea previamente autorizado por algún envolvente. Conceptualmente, puede considerarse que cada uno de los puertos tiene asociado un *cerrojo*, que está implícitamente cerrado, y que habrá de ser abierto –por un envolvente– para que la interacción a través del puerto sea posible.

◊ **letin**: Operador del Lenguaje Dinámico. Recibe como parámetro un nombre de puerto, que deberá pertenecer a la interfaz de algún componente envuelto. Su efecto es permitir que se realice *una* interacción de *entrada* a través de dicho puerto. Es decir, se asume que el cerrojo se abre para una única entrada.

◊ **letout**: Operador del Lenguaje Dinámico. De manera análoga al anterior, recibe como parámetro el nombre de un puerto en un componente envuelto. Su efecto es permitir que se realice *una* interacción de *salida* a través de dicho puerto. Es decir, se asume que el cerrojo se abre para una única salida.

◊ **catch**: Operador del Lenguaje Dinámico. Se puede considerar como el análogo en la salida de lo que es un **shift** en la entrada. Se trata de una operación de lectura, y su efecto es *capturar* la salida emitida por el componente envuelto, que es recibida en el envolvente como una entrada. Al igual que en una lectura, la signatura esperada ha de coincidir con la del mensaje emitido, y podrá variar desde una sincronización pura hasta una tupla. Semánticamente, su definición es independiente del soporte para envolventes, por lo que en realidad la instrucción puede utilizarse en otros contextos; sin embargo, en ese caso la captura se realizaría de manera asíncrona –e indeterminista–, por lo que probablemente este comportamiento carecería de sentido.

Su descripción sintáctica es, como puede verse, muy sencilla, mientras que su semántica (§7.5) resulta casi trivial. El aspecto más complejo es la declaración de un componente como *apto para ser envuelto*; sus diferencias con respecto a un componente convencional son mínimas, pero la estrategia de traducción utilizada es diferente. Por esta razón, no existe una instrucción específica, en el Lenguaje Dinámico, capaz transformar a un componente convencional en uno envuelto, y viceversa. No obstante, esto no supone un verdadero problema, ya que basta con declarar las dos versiones de un tipo, y es posible cambiar dinámicamente –por reificación– entre ellas. Por supuesto, el acto de *envolver* a un componente también puede ser realizado de manera dinámica, y de hecho a menudo lo será.

El uso conjunto de estas instrucciones resulta evidente. No obstante, se proporciona un ejemplo explícito en la ya mencionada sección 8.6, concretamente en la Figura 8.12. Así pues, se comienza con una instancia de un componente apto para ser envuelto, y se *reifica* esta instancia con un envolvente concreto. Todos los mensajes enviados hacia la instancia son bloqueados, pero pueden ser recogidos por el envolvente, mediante una sentencia de lectura normal. Éste manipula entonces el dato a voluntad, y produce una entrada alterada –si procede– para el avatar envuelto, que se introduce en el puerto afectado mediante un **shift**. A continuación, se autoriza una lectura en el puerto (**letin**), de modo que el avatar recibe el dato alterado. Realiza entonces su comportamiento normal, hasta el momento en que intenta realizar una operación de salida, que está nuevamente bloqueada. El envolvente puede entonces capturar la salida emitida (**catch**), manipularla a voluntad, y enviar a su vez el resultado modificado. No obstante, esta última emisión habrá de ser, también, previamente autorizada (**letout**).

Por supuesto, esta secuencia puede ser alterada de cualquier manera; no toda entrada tiene por qué ser interceptada, ni toda salida ha de ser capturada. Simplemente, esta sintaxis proporciona la capacidad de que un envolvente actúe *antes* y *después* de cada interacción del componente envuelto. Debe tenerse en cuenta, además, que el metacomponente envolvente puede interactuar libremente a través de los puertos de su avatar; esto significa que sigue siendo posible realizar lecturas no destructivas (**when**) desde el envolvente, lo cual permite desarrollar un rango aún más amplio de posibles interacciones.

Para concluir este apartado, debe señalarse que, si bien esta sintaxis facilita en gran medida la construcción de cierto tipo de sistemas, no se pretende afirmar que sea una solución definitiva,

ni un soporte genérico para cualquier tipo de envolvente. Tal como se muestra en el ejemplo 8.6, la sintaxis resulta suficiente para emular a un modelo de capas de cebolla, siempre que se liguen entre sí las entradas de los distintos niveles –véase la Figura 8.13–. Sin embargo, este enfoque no resulta totalmente transparente, por lo que cabría considerar un refinamiento ulterior de la sintaxis de $\mathcal{P}i\mathcal{L}ar$ para envolventes, posiblemente mediante una ligera ampliación.

No obstante, el principal objetivo de esta sección era mostrar el modo en que podría definirse algún tipo de extensión de la sintaxis básica de $\mathcal{P}i\mathcal{L}ar$. Como ha podido comprobarse, esto resulta realmente sencillo, y no reviste mayor complejidad.

6.13 Conclusión

A lo largo de este Capítulo se han descrito brevemente las distintas construcciones y características del lenguaje $\mathcal{P}i\mathcal{L}ar$, concebido de manera primordial como un Adl reflexivo. De este modo ha podido comprobarse el modo en que el modelo Marmol puede adaptarse a la sintaxis específica de un lenguaje de descripción, y que de hecho la introducción de reflexión en el lenguaje proporciona capacidades dinámicas.

$\mathcal{P}i\mathcal{L}ar$ ha sido concebido, además, como un Adl dinámico del tipo algebraico, cuya sintaxis realiza una síntesis de las aportaciones de los distintos Adls existentes –y en particular, por supuesto, de los Adls algebraicos–. No obstante, se ha diseñado también como un lenguaje *mínimo*, en el que sólo se incluyen las mínimas abstracciones e instrucciones necesarias.

En $\mathcal{P}i\mathcal{L}ar$ sólo hay cuatro conceptos básicos: componente, puerto, conexión y reificación. Todos los restantes se conciben como combinaciones suyas: así, una configuración es un componente compuesto –de instancias y conexiones–, y un conector es una conexión reificada. Las estructuras se desarrollan en las tres dimensiones identificadas en el modelo $\aleph_{\mathrm{M}}$ de Marmol: composición, interacción y reificación. Se asume un colapso entre la reificación y el tipado de componentes –y conexiones–, de modo que se asume que a todo arquetipo le corresponde un metacomponente, que puede definirse o no como activo.

El lenguaje se divide en dos fragmentos: el Lenguaje Estructural, que describe el armazón estático de una descripción, especificando la estructura del sistema según los conceptos ya mencionados; y el Lenguaje Dinámico, que describe la forma y estructura de las restricciones indicadas sobre los componentes, que son las que definen su comportamiento dinámico.

De nuevo, la sintaxis de este último sigue un enfoque minimal. Tan sólo define los constructores básicos de cualquier álgebra de procesos –secuencia, selección, composición paralela, alternativa y repetición–, y las operaciones básicas para interacción –enviar, recibir, escuchar, callar–, dinamismo –crear, destruir, mostrar, ocultar– y reflexión –reificar, localizar reificación–. El resto del lenguaje tan sólo define los cualificadores adecuados, que permiten recorrer libremente la estructura de niveles y componentes. No es preciso hacer distinciones entre conceptos de distinto tipo o nivel: gracias al esquema reflexivo, una instancia y una conexión se manipulan de modo similar; y una reificación se borra del mismo modo.

Finalmente, se ha de señalar que, aunque la introducción de Reflexión en Arquitectura de Software se ha hecho con el objetivo de especificar dinamismo, se obtienen toda una serie de resultados secundarios interesantes. Éstos incluyen la posibilidad de definir *conectores de nivel meta*, un enfoque completamente distinto de la noción clásica, o la definición de un marco

especialmente adecuado para la descripción de *aspectos* (§4.6) y *vistas* múltiples (§2.4.6); si bien este último extremo no será desarrollado con detenimiento en esta tesis.

BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Capítulo 7

El Lenguaje PiLar: Semántica

No hay problema en informática que no pueda solucionarse mediante la adición de un nivel de indirección.
Maurice Wilkes

7.1 Introducción

En el capítulo anterior se ha definido la sintaxis de $\mathcal{P}i\mathcal{L}ar$, y se ha dado una semántica informal para sus construcciones. Sin embargo, ésta no es suficiente: siempre es preciso dar una semántica clara para un lenguaje completamente nuevo, pero mucho más cuando éste es tan complejo como $\mathcal{P}i\mathcal{L}ar$. Una adecuada comprensión y definición de los elementos reflexivos es absolutamente imprescindible, o de otro modo la definición del lenguaje sería totalmente ambigua.

Por tanto, se pretende aquí proporcionar una semántica formal para $\mathcal{P}i\mathcal{L}ar$. En este caso, se ha preferido expresarla como una *semántica por traducción*, es decir, en el que las construcciones y estructuras del lenguaje se definen mediante su traducción a un lenguaje formal. Se considera que este método resulta mucho más claro que una semántica operacional, más habitual en álgebras de procesos, aunque también usada en Arquitectura de Software para la definición de Padl [BCD01]; o que una semántica denotacional, que resulta poco adecuada y particularmente complicada en un contexto concurrente [Mil93a, AJ94] como el definido por un lenguaje de este tipo.

Por otra parte, el formalismo elegido, esto es, aquél al que se traducirán los términos de $\mathcal{P}i\mathcal{L}ar$, es el cálculo-$\pi$; o para ser más preciso, una variante concreta del cálculo-π, que se describe detalladamente en el Apéndice C.

No se pretende hacer aquí una traducción completa del lenguaje, detallada paso a paso sobre la estructura gramatical de su sintaxis. Esto resultaría excesivamente farragoso, y en definitiva poco ilustrativo. En lugar de ello, y en pro de la claridad y la brevedad, lo que se proporciona es la descripción y traducción al cálculo-π de *cada concepto del lenguaje*, considerado por separado del resto, aunque sin ignorar las traducciones previas. Incluso con este enfoque parcial, el Capítulo ya es, de por sí, suficientemente extenso, y uno de los más largos de esta tesis.

El modo en que todas estos conceptos se integran para formar un conjunto coherente es tan sólo, finalmente, una cuestión de simple composición. Por tanto, puede deducirse de ma-

nera trivial a partir de las definiciones formales que aquí se proporcionan. En el proceso de traducción, pueden distinguirse con nitidez dos tareas distintas. Por una parte, la descripción del Lenguaje Estructural, que hace preciso definir cada uno de los elementos y nociones básicas –arquitectónicas– de $\mathcal{P}i\mathcal{L}ar$. Por otra, la especificación del Lenguaje Dinámico, concebido ante todo como un medio para definir reglas de comportamiento, y que requiere una traducción término a término y constructor a constructor, y que resulta sin embargo mucho más sencilla, ya que se basa en la estructura definida para el sublenguaje anterior.

En todo caso, no será necesario deducir la semántica concurrente de $\mathcal{P}i\mathcal{L}ar$ solamente a partir de su especificación formal. Por el contrario, en la primera mitad de este Capítulo se discute con detalle el razonamiento que ha conducido a la elaboración de toda su infraestructura, proporcionando de este modo una perspectiva global de la misma.

Al final de este Capítulo no sólo se habrá demostrado que el diseño y definición de $\mathcal{P}i\mathcal{L}ar$ sigue una pauta y desarrollo consistentes, sino también que es posible la descripción de un Adl reflexivo y concurrente en términos puramente estructurales, abstrayendo el esquema reflexivo subyacente, del modo que se había postulado durante la definición de Marmol (§5).

7.2 Semántica Intuitiva

Las especificaciones formales, a pesar de su precisión, tienen a menudo el inconveniente de alcanzar una gran longitud, lo que dificulta la obtención de la *visión de conjunto* según la cual fueron diseñadas. Las de tipo semántico, además, suelen ser consideradas particularmente complejas, lo que no hace sino aumentar este efecto. Aunque la forma reducida que aquí se ha adoptado contribuye a mitigarlo –proporciona, al menos, una clara distinción entre estructuras de datos y procesos–, no resulta suficiente, sin embargo, para evitarlo por completo.

Por ello, antes de comenzar la descripción formal en cálculo-π de la semántica de $\mathcal{P}i\mathcal{L}ar$, se ha considerado conveniente incluir una somera exposición del enfoque utilizado, que proporcione la mencionada visión de conjunto. A este fin se dedica todo este apartado. Dado que es aquí donde se obtiene una perspectiva global, parece ser asimismo el lugar idóneo para detallar los *motivos* que han determinado la elección de las distintas opciones de diseño. Por ello, éstos también se discuten aquí, y se contrastan brevemente con algunas de las alternativas utilizadas en otros lenguajes, en particular en otros Adls de tipo algebraico.

7.2.1 Concepto Semántico Básico

En la descripción sintáctica del lenguaje se han asumido una serie de conceptos estructurales de alto nivel, tales como *componente*, *puerto*, *conexión* y *reificación*. Esto es perfectamente adecuado al nivel de la sintaxis, ya que ésta da forma a las abstracciones que va a manejar y manipular el arquitecto de software, y éste debe trabajar en todo momento precisamente en dichos términos. Sin embargo, ya desde el modelo $\aleph_\Sigma$ (§5.5.1), con el que se pretendía describir la estructura básica de cualquier especificación arquitectónica, es obvio que estas nociones no pueden ser tomadas como primitivas; en especial cuando sabemos que es posible, e incluso necesario, definir algunas de ellas en función del resto.

Una definición semántica razonable ha de partir necesariamente, pues, de algún concepto más simple. Resulta evidente, a partir de todos los razonamientos anteriores (§5.4), que en nuestro

caso ese concepto será el de *nombre*. Esto no sólo facilitará su manipulación mediante cálculo-π, sino que además hará posible establecer una correspondencia directa con las definiciones de Marmol. Más aún, gracias a esta elección la semántica puede concentrarse en los aspectos estructurales, que son los que en definitiva afectan a un Adl, sin que por ello se produzca pérdida de significado –o de representatividad– alguna.

Así pues, la semántica parte de que todo elemento de una especificación tiene un nombre asociado, incluso cuando esto no se indica sintácticamente. No existen, pues, elementos *anónimos*; cuando se necesite simular dicha situación, simplemente el nombre no se hace explícito –o es desconocido en un ámbito dado–. Por supuesto, el nombre no es la única información asociada a una noción dada, ya que de otro modo la semántica sería realmente trivial. A toda construcción de cierta complejidad –lo que, como veremos, quiere decir a *todas* las del lenguaje– se le asocia además una descripción funcional o estructural. En un álgebra de procesos, como es el cálculo-π, esto último significa, literalmente, que toda construcción se corresponde con un *proceso*.

Por tanto, a cada elemento concreto de una descripción arquitectónica la semántica le asocia un nombre, que lo designa, y un proceso, que lo representa. Disponer del primero es especialmente relevante en un álgebra de este tipo, donde un proceso puede ser recursivo, estar replicado, o construirse por composición de otros procesos. Los límites que lo demarcan son, cuando menos, difusos, y más aún en el cálculo-π, donde en cualquier momento pueden crearse nuevos canales y la topología puede alterarse con cualquier comunicación. Sin embargo, este problema desaparece al tener un nombre, que ejerce la función de un *canal de acceso*. La frontera del proceso en cuestión vendrá dada ahora, simplemente, por el *ámbito accesible* a partir de este canal.

7.2.2 Discusión: sobre las Semánticas Algebraicas

Partir de que todo componente o conexión se representa mediante un proceso no es una premisa muy original. En realidad, es compartida por (casi) todos los Adls algebraicos. De nuevo, resulta especialmente interesante la comparación con Wright, el primero en apoyarse en este planteamiento. De hecho, la intuición en la que se originan las estructuras de $\mathcal{P}i\mathcal{L}ar$ nació, en parte, como una reacción a la complejidad conceptual de Wright.

En Wright, como ya se ha comentado, se tienen *componentes*, cuya interfaz se divide en *puertos*, que se comunican a través de *conectores*, cuya interfaz se segmenta en *roles*. Éstos se unen mediante *adjunciones* o *attachments*, y el conjunto conforma una *configuración*, que puede a su vez exportar una interfaz, actuando así como un componente *compuesto*. Esto da lugar a la correspondiente jerarquía de composición.

Semánticamente, todos estos elementos se corresponderán con procesos Csp. Nunca se escribió una descripción detallada de la semántica de Wright, aunque sí llegó a implementarse un traductor para la herramienta Fdr. Sin embargo, no es difícil deducirla, ya que la correspondencia notacional resulta suficientemente clara.

En primer lugar, un *puerto* –o rol– se corresponde con un proceso, cuya definición se incluye explícitamente en la declaración del (tipo de) componente. Este proceso manipula uno o más nombres libres, esto es, canales por los que podrá comunicarse, sea hacia el interior o hacia el exterior. Un *componente* se forma mediante la composición paralela de estos procesos-puerto; no obstante, en principio éstos no están relacionados entre sí, por lo que Wright exige incluir

la definición explícita de otro proceso Csp, que describe el comportamiento del componente, y que se conoce como la *computación*. Dicho esto, se podría esperar que la definición de un *conector* fuese similar, pero no lo es. En efecto los roles son procesos análogos a los puertos, que describen "el otro extremo" de un protocolo; y de nuevo se exige la definición explícita de un proceso Csp que vincule los procesos-rol, conocido como el *pegamento*. Sin embargo, la relación entre el pegamento y los roles es exactamente la contraria de la que había entre la computación y los puertos. Peor aún, es el arquitecto el que ha de tener presente esta distinción, ya que es el que debe encargarse de la definición de ambos procesos –pegamento y computación–.

Las *adjunciones*, tal vez el elemento más desconcertante (§2.4.3), son en cambio las que tienen una semántica más simple. Para empezar, no se establecen, como puede parecer, entre un componente y un conector, sino entre un puerto y rol concretos, aprovechando así la modularidad del álgebra. Precisamente, su labor es hacer posible la comunicación entre estos procesos complementarios, para lo cual es necesario que dispongan de un vocabulario común. Por tanto, resulta ahora obvio que una adjunción es tan sólo una correspondencia entre los nombres libres del puerto y del rol: en resumen, se está usando el operador de composición $({}_K\|_L)$ de Csp, el mismo que en la terminología Ccs se denomina *conjunción* [Mil89]. A partir de aquí, la semántica es trivial; las *configuraciones* son tan sólo una reunión de los elementos descritos, que encajan por sí mismos describiendo el sistema como un único *proceso*, cada vez más complejo.

A pesar de la obvia elegancia del resultado, la sensación general es que este esquema es demasiado complejo. Cuando se considera en su conjunto, se puede comprobar que el lenguaje, en realidad, no ha aportado *nada*: todos los detalles del sistema han tenido que ser especificados formalmente en Csp, desde el protocolo de cada puerto hasta la actividad del componente. Wright es un Adl casi canónico, que describe un sistema al más alto nivel de abstracción –el arquitectónico–. Y sin embargo en definitiva no es más que una *plantilla* que en cada sistema debe ser rellenada con una serie de definiciones de proceso. Una vez que se dispone de éstas, la plantilla ya no es necesaria: el conjunto de procesos constituye por sí solo una especificación formal completa del sistema concurrente –la arquitectura– descrito.

¿Cuál es la ventaja de utilizar Wright, pues?. Dado que el arquitecto tiene que especificar formalmente todo, ¿no podría haber utilizado directamente Csp desde un principio?. La respuesta es la del propio Allen [All97]: lo que proporciona el Adl son las abstracciones arquitectónicas. Gracias a ellas, resulta más sencillo organizar las descripciones formales, de modo que en cada momento sólo se tienen en cuenta los aspectos que afectan a un componente concreto. La modularidad, inherente a un álgebra de procesos, llega incluso a intensificarse.

¿Es cierto esto?. En parte, desde luego. Pero es posiblemente más cierto en algunos Adls posteriores que en el propio Wright. Éste tiene un inconveniente cuando menos significativo: a saber, que en la mayoría de los casos el sistema obtenido a partir del esquema arquitectónico es *más complejo* que una especificación formal directa del mismo problema. Considérese, por ejemplo, un par cliente-servidor típico. Se puede describir en cualquier álgebra mediante dos procesos, comunicados a través de uno (o dos) canales. En cambio, la descripción Wright necesita dos componentes, un conector, dos puertos, dos roles y dos adjunciones. En total, el arquitecto debe especificar, como mínimo, *siete* procesos Csp.

Por otra parte, en el campo de la teoría de la Concurrencia se ha mantenido con frecuencia que la propia noción de *proceso*, como tal abstracción algebraica, es por sí misma lo suficientemente intuitiva, y podría ser utilizada *directamente* en un modelo conceptual. Esta afirmación es, tal

vez, exagerada: la noción es, en efecto, simple; pero posiblemente también demasiado primitiva. Es obvio, además, que no resulta conveniente descartar sin más los conceptos arquitectónicos, pues es cierto que simplifican la concepción de la estructura de un sistema.

Aún así, la sugerencia no deja de resultar atractiva. Y esto es porque hay, de hecho, una similitud más que notable entre las ideas intuitivas de *proceso* y *componente (instancia)*. Ambos se definen como entidades computacionales de tamaño indeterminado, con una serie de puntos de interacción –canales o puertos– bien definidos, mediante los que los que se relacionan con otras entidades similares, obteniendo por *composición* un sistema completo. De hecho, es este parecido el que justifica la extensión del enfoque algebraico en la Arquitectura de Software, ya comentada en la sección 6.2. No es extraño, pues, que varios trabajos posteriores hayan intentado recuperar la "pureza" de la analogía original, simplificando el esquema propuesto en Wright.

Entre éstos, Padl presenta la aproximación más sencilla; al mismo tiempo, es también la menos lograda. No intenta simplemente recuperar la analogía, sino que la hace explícita. En una arquitectura Padl, los *elementos arquitectónicos* se definen como procesos, cuyos canales se denominan *interacciones*, y se declaran explícitamente como de entrada o de salida. La topología viene dada por una serie de emparejamientos o *attachments* entre dichas interacciones. No existe el concepto de conectores. El problema de este método es que en ningún momento se abstrae del modelo formal al arquitectónico. Más que en Padl, la arquitectura se está especificando directamente en Pa, el álgebra de procesos definido por Marco Bernardo [BCD01] a tal efecto. Se trata, además, de un lenguaje totalmente estático. Sin embargo, es un Adl muy reciente, y todavía podría evolucionar; últimamente, ha añadido soporte para extensibilidad [BF02].

Posiblemente el lenguaje que mejor logra el equilibrio es Leda. Se trata de un Adl algebraico sin conectores, cuya álgebra subyacente es el cálculo-π; fue ya descrito en la sección 3.7.3. Tiene las ventajas de Wright sin sus inconvenientes: el arquitecto trabaja siempre al nivel de abstracción correcto, pero la traducción semántica de una descripción, además de ser casi literal, es mucho más cercana a la que se lograría con una descripción directa. Además del dinamismo proporcionado por el cálculo-π, añade soporte para *adaptadores*, conexiones (*attachments*) de cardinalidades diversas, y el tratamiento más completo de *extensibilidad* del campo [CPT01], junto con el de C2sadel. Lo único que se le puede objetar, en realidad, es una correspondencia demasiado estrecha con el cálculo-π, que ha de ser utilizado por el arquitecto como lenguaje de especificación. Esto tiene los problemas ya señalados en la sección 6.6: el tratamiento inconsciente de la movilidad y el estilo-CPS de los procesos-π. Aunque Leda hace un buen trabajo a la hora de mantener a raya el primero, no puede evitar la aparición del segundo, ya que forma parte de la propia concepción del cálculo.

No se puede dejar de señalar un enfoque radicalmente opuesto, y en parte anterior al de Wright: el de *δarwin*, lenguaje que también fue ya descrito en la sección 3.7.2. Es considerado un Adl algebraico porque su semántica ha sido especificada en un álgebra de procesos (concretamente cálculo-π [MDEK95]); sin embargo, esto no se refleja en su sintaxis, dado que su origen es anterior. De hecho se lo ha clasificado como un Adl orientado al objeto, más que orientado al proceso [BF97]; y es esto lo que le hace adoptar una estrategia diferente a la de los demás Adls mencionados. Los elementos del lenguaje (preexistentes, por tanto) tienen una expresión equivalente en cálculo-π, pero una vez definidos se usan como si fueran atómicos. El arquitecto, pues, no necesita (ni puede) especificar el protocolo completo, sino que tiene acceso

a un número limitado de primitivas, con una estructura claramente inspirada en el paradigma de llamada de procedimiento. Se maneja, pues, una sintaxis de más alto nivel; pero obviamente la correspondencia con la semántica de procesos es mucho menos directa. Esto facilita, por una parte, la descripción de la arquitectura, pero por otra limita el rango de posibilidades.

Un ejemplo claro de esto último es el modo en que se usan las interfaces. Los puertos de Wright son especificados formalmente en cada declaración de componente, y se corresponden semánticamente con el proceso así descrito. Los *portales* de $\delta arwin$ se definen como elementos del lenguaje, con un comportamiento predeterminado, que es diferente según se trate de interfaces de provisión (*provide*) o de petición (*require*) de servicios. Aunque semánticamente se corresponden con un proceso-π completo, sintácticamente se utilizan como un simple punto de interacción. Curiosamente, esto no significa que sean menos potentes, ya que los portales de $\delta arwin$ son capaces de soportar comunicaciones que no pueden ser expresadas en Wright; pero sí son mucho menos versátiles. Esto se hace más aparente en la última revisión de $\delta arwin$[1], donde se pone mayor énfasis en la que sus autores [MKG99] denominan la *vista de comportamiento*. Los tipos de portal, antes omnipresentes, llegan incluso a descartarse, con el fin de poder flexibilizar la especificación de los protocolos. Pero esta especificación ya no se realiza en cálculo-π, sino que se usa directamente un Sistema de Transiciones Etiquetadas (LTS), que se considera más simple. En realidad, es un formalismo sencillo y potente, pero está mucho menos evolucionado.

Los tres casos comentados tienen algo en común: parte de la simplicidad de la descripción arquitectónica se recupera eliminando por completo la noción de *conectores*. De esto se deduce que, si bien un intermediario es muy útil cuando es preciso, no parece que forzar siempre a su utilización sea conveniente. En la sección 6.11 ya se han discutido las ventajas e inconvenientes de este aspecto. El diseño de $\mathcal{P}i\mathcal{L}ar$, como también se ha descrito, sigue esta misma idea: se permiten conectores (o sus análogos) cuando son necesarios, y se suplen con una definición por defecto cuando esto es suficiente.

7.2.3 Componentes como Procesos Abstractos

Dicho todo lo anterior, conviene ante todo plantearse el enfoque global que ha de manifestar una semántica basada en procesos. A primera vista, parece claro que en los ADLs algebraicos existe un difícil equilibrio entre la sintaxis de la descripción arquitectónica, capaz de manipular conceptos de alto nivel, y la especificación formal pura, concisa y elegante, pero situada a "bajo" nivel. Podría llegarse a pensar, incluso, que sus objetivos son opuestos. Se ha visto que cuando el lenguaje se acerca demasiado a una notación procesual, como ocurría con PADL, la visión arquitectónica se diluye. Por el contrario, cuando se utiliza un vocabulario más cercano al dominio de aplicación, como ocurre en el caso de $\delta arwin$, no sólo se pierde versatilidad, sino que la analogía en la que se basaba la semántica se difumina.

Esa conjetura, sin embargo, es falsa. No sólo sus objetivos no son opuestos, sino que de hecho ambos persiguen la misma meta: tan sólo se debe encontrar el nivel de abstracción adecuado. No se trata, pues, de un conflicto entre las nociones de proceso y componente; al contrario, la clave está en determinar sus coincidencias, y aprovecharlas. En parte, el problema está también en que se asume una correspondencia cercana entre los usos sintáctico y semántico de la idea de

[1] Para ser exactos, la que en la sección 3.7.2 fue denominada $\delta arwin$ 3.

proceso; como se verá de inmediato, esto no es necesario.

Como ya se ha dicho en el apartado anterior, en más de una ocasión se ha afirmado que el modelo de procesos es lo bastante abstracto como para ser usado directamente. La idea, desde luego, es sugestiva. Se dispondría de un esquema en que todo componente es un proceso, y toda conexión un canal; los elementos fundamentales están ya representados, y el esquema de manipulación parece mucho más simple. No sería necesario, como en Wright o Leda, construir un proceso completo para cada puerto, sino que simplemente se usaría como un canal; no sería preciso separar el protocolo de comunicación y la actividad interna –la "computación"– del componente, sino que se podría concebir, simplemente, como un todo.

Y sin embargo, ninguno de los Adls descritos llega a este punto, ni siquiera cuando se construyen con el objetivo de recuperar la "pureza" de la analogía. La razón se hace evidente en cuanto se examina con detenimiento el problema: la idea de proceso es en realidad demasiado simple, y no se comporta de la manera inicialmente prevista. La correspondencia directa que se estaba buscando, en definitiva, no existe ni es posible.

Para ilustrar lo engañoso de esta simplicidad, se propone el ejemplo siguiente, que describe un sistema conscientemente mal diseñado[2].

$$
\begin{aligned}
\mathrm{A}(a) &\triangleq \overline{a}\langle\, x\,\rangle.\, A_1 \\
\mathrm{B}(b) &\triangleq b(y).\, B_1 \\
\mathrm{C}(c_1, c_2) &\triangleq (\; c_1(z).\, \overline{c_2}\langle\, z\,\rangle.\, \mathrm{C}\langle\, c_1, c_2\,\rangle \mid c_2(w).\, \overline{c_1}\langle\, w\,\rangle.\, \mathrm{C}\langle\, c_1, c_2\,\rangle\;) \\
Sys &\triangleq (\nu\, s, t)(\mathrm{A}\langle\, s\,\rangle \mid \mathrm{B}\langle\, t\,\rangle \mid \mathrm{C}\langle\, s, t\,\rangle) \\
&\equiv (\nu\, s, t)(\overline{s}\langle\, x\,\rangle.\, A_1 \mid s(z).\, \overline{t}\langle\, z\,\rangle.\, \mathrm{C}\langle\, s, t\,\rangle \mid t(w).\, \overline{s}\langle\, w\,\rangle.\, \mathrm{C}\langle\, s, t\,\rangle \mid t(y).\, B_1)
\end{aligned}
$$

Los tres primeros procesos describen tres agentes, concebidos de manera independiente. El primero (A) emite un dato y continúa su actividad; a su vez, el segundo (B) recibe un dato y continúa su actividad. El más complejo es el tercero (C), que manipula dos canales: cuando recibe un dato por uno de ellos, lo reenvía por el otro, y repite el ciclo. El cuarto proceso (Sys) describe un sistema como la composición de estos agentes; la idea es que los dos primeros realizan el papel de "componentes", y el tercero actúa como un "conector" entre ellos. Para hacer posible la comunicación, se crean los canales necesarios (s, t), de A a C y de C a B.

La idea es que A envía el dato x; C lo recoge y lo reenvía, y finalmente B lo recibe. Pero, en realidad, las cosas no están tan claras: el sistema es claramente indeterminista. En efecto, la interacción prevista es posible; *pero no es la única.* El inicio es claro: A envía x a través de s, y C lo recibe. Pero cuando C lo reenvía a través de t, surge el conflicto. La intención es que llegue hasta B, pero igualmente podría ocurrir que fuese *el propio* C el que lo capturase, por su otra rama –subproceso paralelo–. Tanto B como C tienen una lectura $[t(y)]$ sobre el canal implicado, y no hay nada que dé prioridad a una sobre la otra. Peor aún: si es C el que recoge el dato, lo reenvía a través de s. Pero ya no hay nadie allí para recogerlo, excepto *él mismo*, que vuelve a capturarlo, para enviarlo de nuevo sobre t. Se ha regresado a la situación original, y el mismo conflicto se reproduce: podría ocurrir que B recibiese por fin el dato, pero igualmente

[2]El lenguaje utilizado es cálculo-π, pero se está usando como un álgebra de procesos no móviles. De hecho, podría tratarse igualmente de Ccs con paso de valores, excepto por la creación de canales en Sys, introducida sólo por comodidad, y que podría sustituirse con facilidad sin más que usar el operador de renombrado.

que C volviese a capturarlo. De hecho, nada impide que C caiga en un bucle infinito, ni asegura que el mensaje vaya a llegar hasta B en algún momento del futuro.

La situación parece patológica, y sin embargo se trata de un ejemplo mínimo. Además, se ha presentado con un álgebra síncrona, en la que el indeterminismo está parcialmente controlado. Con un enfoque asíncrono, en el que todo envío es instantáneo y no bloqueante, el conflicto podría surgir incluso en procesos más sencillos.

Ya se había dicho que el ejemplo estaba mal planteado, y sin embargo las definiciones de los agentes *parecían* razonables, si se consideraban desde una perspectiva "arquitectónica". ¿Dónde falla, pues, la analogía?. Precisamente, en que es demasiado literal: las nociones de puerto y canal se han utilizado como sinónimas; y aunque es cierto que son semejantes, no son sin embargo idénticas. En realidad, es fácil diagnosticar el fallo del ejemplo anterior: se está usando un mismo canal $[t]$ en dos sentidos diferentes, y éstos entran en conflicto[3] al combinar los procesos. La idea era que sirviera como un puerto de acceso a B, y este puerto sería bidireccional. Desde fuera, todo envío se dirige hacia el proceso, y toda recepción procede de él; desde dentro, todo envío se dirige al exterior, y toda recepción procede de allí. Pero la distinción que implícitamente acaba de hacerse, no existe en realidad: tanto el proceso como su entorno pueden capturar *sus propios mensajes*. La ambigüedad solamente desaparece cuando el uso del canal es unidireccional. Entonces, todo envío exterior se corresponde con una recepción interior, y viceversa. Esto hace obvio que se necesitan, al menos, dos canales: uno de *entrada* y otro de *salida*.

De hecho, es perfectamente habitual, en un álgebra de procesos, distinguir entre canales de entrada y de salida. No sólo por este tipo de problemas, sino incluso por cuestiones de *tipado* [BS95]: no hay motivo alguno para suponer que los datos de salida, generados por el proceso, tengan que coincidir en tipo con los datos de entrada que éste recibe. La distinción entre canales de entrada, de salida o mixtos ha demostrado ser fructífera en el cálculo-π, donde ha incluso ha originado la definición de una noción de subtipado [PS96, SW01].

Pero, si bien resulta razonable al nivel del cálculo, no lo parece tanto desde un punto de vista arquitectónico. Al contrario, como ya se ha dicho, un puerto se concibe habitualmente como bidireccional, y acepta interacciones tanto de entrada como de salida. Esto implica que, más que con un canal, un puerto se corresponde con *dos*. Ésta es, precisamente, la razón de la definición del *par* de canales como un tipo básico en nuestra versión del cálculo-π (véase la sección C.4.3). Sin embargo, lo realmente importante de este punto es que demuestra que **una correlación directa de puerto a canal, o de componente a proceso, es de hecho ilógica**.

Un uso ingenuo de la analogía entre componente y proceso resulta pues demasiado forzado, y es en definitiva inadecuado. Sin embargo, la analogía en sí sigue siendo atractiva, y puede de hecho recuperarse. La clave, como se dijo al principio, está en separar con cuidado los aspectos sintácticos y semánticos. *Semánticamente*, lo que se se pretende lograr una representación de las nociones arquitectónicas como procesos del cálculo-π; es aquí donde, como se ha visto, *no* conviene basar el planteamiento en una correspondencia directa. En cambio, no hay ninguna razón para no mantenerla desde un punto de vista *sintáctico*. Se ha dicho varias veces que la idea de proceso ofrece un modo natural de definir el comportamiento de un componente: eso significa que las construcciones y primitivas de la sintaxis algebraica son un fundamento adecuado para el diseño del *lenguaje* de especificación, como ya se argumentó en la sección 6.2, y como se vino

[3] Se produce, en definitiva, una *interferencia* [Jon96, Mil93a].

aplicando a lo largo de todo el Capítulo 6.

El enfoque lingüístico de $\mathcal{P}i\mathcal{L}ar$ se basa pues en la concepción de un componente como un *proceso abstracto*[4]: una entidad que se describe y manipula de manera parecida a un proceso, pero cuya representación semántica es, en ciertos puntos, mucho más compleja. Su lenguaje de comportamiento (el Lenguaje Dinámico) tiene una forma similar a la de un álgebra, pero con todos los elementos a nivel arquitectónico. Los puertos se manejan como canales, pero tienen una serie de mecanismos internos que aumentan sus posibilidades y evitan la aparición de condiciones inesperadas. En resumen, se recupera sintácticamente la pureza de la analogía, sin forzar por ello, semánticamente, a una traducción literal.

Se adopta, pues, una postura intermedia entre las de Wright y $\delta arwin$. En el primero se tiene tanto una sintaxis como una semántica procesual, con una correlación demasiado cercana; en el segundo las nociones del lenguaje se abstraen en la semántica, pero se basa la sintaxis en conceptos diferentes. En $\mathcal{P}i\mathcal{L}ar$, en cambio, la correlación es mucho más abstracta que en el primer caso; y la estructura sintáctica, a diferencia del segundo caso, respeta la analogía. El lenguaje gana, de este modo, en flexibilidad, ya que no es limitado por una semántica demasiado rígida, ni por una sintaxis excesivamente restringida.

7.2.4 Conceptos Arquitectónicos de PiLar

Conviene retomar ahora las nociones arquitectónicas de alto nivel mencionadas al principio de esta sección: *puertos*, *componentes*, *conexiones* y *reificaciones*. Se dejarán estas últimas para más adelante. Sin embargo, se pueden esbozar ya unas ideas generales sobre la representación que se hará en cálculo-π, o más bien en lenguaje procesual, de las tres primeras, una vez consideradas como conceptos al nivel de un proceso abstracto.

Puertos

En principio, el concepto de *puerto* sigue siendo el más sencillo. Sintácticamente, se manipula como si fuera atómico; pero como acaba de verse, semánticamente esto no es suficiente, y se exige definir algún tipo de estructura. Ya se ha justificado que ésta debe constar, como mínimo, de un *canal de entrada* al componente, y de otro *de salida* del mismo.

Puede comenzarse, pues, con la representación inicial de un puerto como un *par* de canales. Esto puede expresarse en el cálculo mediante un registro de tipo $(Par\,\alpha\,\beta)$, tal como se detalla en el Apéndice C. Aunque la versión final será mucho más compleja, de momento puede utilizarse ésta como hipótesis de trabajo. En cualquier caso, no afecta al razonamiento que sigue, que de hecho es aplicable a todas las estructuras de datos de la semántica.

Partiendo ahora de que un puerto ya no es atómico, es evidente que las distintas menciones que se hagan de él a lo largo de una descripción arquitectónica (en $\mathcal{P}i\mathcal{L}ar$) harán referencia a distintas partes de su estructura. De este modo, cuando un puerto P es utilizado para una

[4]Este término se utiliza de manera informal, e intenta expresar la idea de que un componente $\mathcal{P}i\mathcal{L}ar$ conserva la analogía, y es sintácticamente parecido a un proceso, pero está situado a un nivel de abstracción mucho mayor. No se debe confundir, pues, con otros usos ocasionales de la misma expresión existentes en la literatura. Por ejemplo, en el contexto de las redes de Petri, que es un modelo de *concurrencia verdadera*, hace referencia a la representación de la causalidad como un orden parcial entre las transiciones de la red [Cas01].

recepción, lo que en realidad se está usando es su canal de entrada; por el contrario, si se trata de un envío, el elemento involucrado es su canal de salida.

No obstante, no puede considerarse que un puerto se reduce a un simple registro. Ocurre a menudo que no se utiliza como sujeto de una acción, sino como objeto de la misma; así pues, sigue siendo preciso disponer de un *nombre* para el puerto, que permita referirse a él como un todo. Por otra parte, la necesidad de definir un nombre tal para cada entidad era una de las premisas de nuestro modelo semántico, tal como fue esbozado en la sección 7.2.1. Este nombre será, por tanto, el tercer canal relacionado con un puerto.

Como ya se dijo, a partir de este nombre debe ser posible obtener toda la información asociada con un elemento arquitectónico: así, en el caso de un puerto, los dos canales que (hipotéticamente) lo componen. Puede parecer entonces que basta con establecer el nombre como un canal que apunta hacia el registro adecuado. Esto no es cierto, por tres motivos. En primer lugar, un canal no es un puntero, sino un punto de interacción; con este planteamiento, por tanto, la primera lectura sobre el canal *consumiría* el registro, el cual se supone que debería seguir existiendo. Sin embargo, este no es un inconveniente grave: se podría establecer un protocolo adecuado –por ejemplo, un esquema de petición-reenvío, similar al que se utilizará con las estructuras de tipo lista–, que asegurase que el registro vuelve a estar disponible tras cada interacción.

En segundo lugar, es preciso que el nombre mantenga un vínculo permanente con la estructura a la que designa. Esto no es trivial, dada la movilidad del cálculo-π. Cuando un registro es transmitido a un contexto distinto del original, éste puede transmitirlo a su vez a través de otro canal. Este nuevo canal podría usarse también como un nombre para el registro, con la consiguiente confusión. En realidad, la situación es peor aún: no solamente se tienen nombres múltiples para un registro, sino para múltiples *copias* de éste.

En tercer lugar, y relacionado con el punto anterior: se ha de tener en cuenta que el nombre del registro es parte de la información esencial asociada con el mismo. Una misma regla $\mathcal{P}i\mathcal{L}ar$, o un proceso de la semántica, puede usar un puerto concreto como sujeto y como objeto; por tanto, es en general necesario que se conozca tanto el registro como su nombre. Obviamente, es posible transmitir ambos en cada interacción; pero resulta mucho más lógico incluir el nombre *dentro del propio registro* al que designa.

La estructura así obtenida no solamente es recursiva, sino que manifiesta una forma aún más pura de autorreferencia. Resuelve además el segundo problema planteado, ya que la contención supone, sin duda, un vínculo *permanente* entre el registro y su nombre. De hecho, a partir de este instante es irrelevante de cuál de los dos se disponga en un momento dado: si se tiene la estructura, ésta contiene al nombre; si se tiene el canal, se obtiene el registro mediante una simple lectura.

Sin embargo, ¿cómo es posible que un canal sea una referencia hacia la misma estructura que lo contiene?. Más aún, se exige que lo sea de manera indefinida, esto es, que admita cualquier número de peticiones de dicha estructura. Además, no se proporciona ningún tipo de protocolo que garantice este suministro; por tanto, éste debe estar asegurado de antemano.

La respuesta es la siguiente: el vínculo es posible porque se introduce *después* de la creación del nombre. En primer lugar, toda entidad del lenguaje será generada por un proceso. Este proceso creará en primer lugar el canal que actúa como nombre, y luego elaborará el registro

que constituye el puerto, incluyendo dicho canal como parte de la estructura. Sólo después, una vez que los dos han sido construidos, se establece la relación entre ellos.

Esto se logra invocando un proceso especial, que recibe ambos como argumento. Éste se define como una secuencia recursiva que emite continuamente el registro (*dato*) a través del canal (*me*). De este modo, cualquier lectura sobre el canal permite obtener el registro completo, y éstos quedan por fin ligados de la manera deseada. La definición del proceso se muestra a continuación. Se trata, como puede verse, de una definición polimórfica. Esto es particularmente conveniente, ya que este proceso no será aplicado únicamente a los puertos, sino que será ampliamente utilizado sobre categorías semánticas de todo tipo a lo largo de todo este Capítulo.

$$\begin{array}{rcl} \mathrm{RepSelf} & : & (\sharp\gamma;\ \uparrow\gamma,\ \gamma,\ \mathit{Sync}) \\ \mathrm{RepSelf}(me, dato, die) & \triangleq & \overline{me}\langle\, dato\,\rangle.\,\mathrm{RepSelf}\langle\, me, die, dato\,\rangle + die.\,\mathbf{0} \end{array}$$

Obsérvese que la definición incluye un tercer argumento, *die*, que se declara como una sincronización pura (*Sync*). Puede verse que éste define una recepción, que se ubica como alternativa a la recursión en una estructura selectiva. Esto significa que cuando se recibe una señal en este extremo, la recursión finaliza: se trata, por tanto, de una señal de terminación. En definitiva, de este modo se hace posible *destruir* la entidad involucrada, ya que al morir el proceso, el suministro de estructuras de tipo-γ queda interrumpido. De otro modo, si se hubiera definido como una recursión pura, o incluso una replicación, éste hubiera continuado indefinidamente. Este tipo de esquema será también muy habitual durante la traducción de las restricciones recursivas de $\mathcal{P}i\mathcal{L}ar$, tal como se describirá en la sección 7.4.3.

Este razonamiento se ha expuesto para los puertos; sin embargo, como ya se ha dicho, es igualmente aplicable a todas las entidades de alto nivel del lenguaje. Todas ellas muestran idéntico conflicto entre nombre y estructura; todas ellas –sean puertos, componentes o conexiones– incluyen a su propio nombre, y por tanto manifiestan el mismo grado de autorreferencia.

Volviendo al caso específico de un puerto, se han de hacer algunos comentarios más sobre su representación interna. En principio, su hipotética forma como un par de canales resulta suficiente para describir el envío y recepción convencional de mensajes. Sin embargo, el Lenguaje Dinámico de $\mathcal{P}i\mathcal{L}ar$ considera una serie de operadores no habituales en un álgebra de procesos, tales como **when**, **catch** o **shift**, que exigen un comportamiento adicional. En todos ellos se distingue entre el momento en que se *inicia* una acción de comunicación –sea una recepción o un envío– y el momento en que ésta, efectivamente, se *produce*.

Esto implica diferenciar entre el interfaz *interno* y *externo* del proceso, en lo que respecta al puerto considerado; y es obvio que tal distinción no se puede hacer con un par de canales, por lo que la estructura ha de ser ampliada, al menos, a *dos pares*. No obstante, éstos no son aún suficientes: como se verá, la dinámica de las primitivas mencionadas exige el concurso de otros dos pares de canales. La adición de reflexión –véase la sección 7.2.5– aprovecha en parte esta estructura, pero añade a su vez más complejidad. La versión final de la misma se compone, en definitiva, de cinco procesos, una lista nombrada y once canales, que incluyen a los ya citados. Pero a pesar de lo que pueda parecer, no es demasiado complicada: de hecho, se trata de una estructura estática, con un comportamiento fijo, definido por procesos conocidos de antemano.

No se comentarán aquí los detalles concretos sobre esta representación, ya que a su descripción se dedica la totalidad del apartado 7.3.1, al cual nos remitimos.

Componentes

La estructura de un componente es, por definición, la más compleja de la semántica, en especial si no se consideran las nociones reflexivas. En realidad, esto no supone decir demasiado: la estructura de un puerto es notablemente simple, y la de una conexión será, como se verá más adelante, un subconjunto de la aquí definida.

Sin embargo, parte de esta complejidad no es inherente a la propia estructura, sino que nace del mismo concepto, o más bien de las confusiones que se dan en torno al mismo. Para obtener una idea clara, hay que establecer al menos tres distinciones significativas.

- En primer lugar, la clásica distinción entre *tipo* de componente e *instancia* de componente, que ya se ha puesto de relieve, tanto en el planteamiento matemático del modelo Marmol, ya desde su forma básica ($\aleph_\Sigma$), como en la definición de la sintaxis.

  Esta distinción es fundamental desde un punto de vista lingüístico. Las instancias son los auténticos actores de una configuración real, aquéllos que tienen un comportamiento definido, y los que en realidad simulan –bisimulan– la dinámica de un sistema. Por su parte, los tipos definen los patrones según los que se crean estas instancias, y plantean las relaciones a nivel estructural. No obstante, toda la importancia de estos últimos se diluye en la semántica. Ésta sólo trabaja con procesos, concebidos como entidades activas; éstos se construyen para cada instancia. Los arquetipos, en cambio, se utilizan solamente como los *esquemas de definición* de estos procesos, esto es, las plantillas sobre las que elaboran las definiciones de las instancias. Sin embargo, una vez que éstas han sido desarrolladas y adquieren una forma independiente, los tipos no juegan papel alguno. De hecho, en principio no existe en la semántica ningún tipo de estructura que los represente.

  Dicho de otro modo: en la semántica, los únicos tipos relevantes son los de los procesos y estructuras concurrentes que ésta define; y éstos no guardan relación alguna, salvo indirectamente, con los *tipos arquitectónicos* definidos sintácticamente.

  Por tanto, en todo este Capítulo cuando se habla de componentes se habla en realidad de *instancias*. Como ocurre habitualmente con los esquemas basados en procesos o actores, una vez desplegados, la estructura de clasificación se difumina en gran medida. De hecho, en ciertos aspectos, el planteamiento guarda muchos parecidos con un sistema prototipado. Ha de advertirse, sin embargo, que en $\mathcal{P}i\mathcal{L}ar$ los tipos tienen de hecho un papel más activo; pero éste estará ligado a su naturaleza reflexiva, y obedece a consideraciones diferentes, que no se desarrollarán aquí, sino en el apartado siguiente (§7.2.5).

- En segundo lugar, precisamente, se plantea la distinción entre componentes y metacomponentes, esto es, entre los aspectos puramente estructurales y los específicamente reflexivos. Buena parte de la complejidad inherente al concepto de componente en $\mathcal{P}i\mathcal{L}ar$ tiene mucho más que ver con la presencia de reflexión que con su fundamento arquitectónico. Este apartado se centrará exclusivamente en este último, con el fin de definir un esquema básico y fácil de comprender. Los detalles restantes se añadirán luego, en la ya mencionada sección 7.2.5, al considerar la introducción de reflexión en el modelo.

- En tercer lugar, por fin, se encuentra una distinción mucho más sutil, y que sí será relevante en este momento: la que hay entre los aspectos estáticos y dinámicos de una instancia.

Se ha dicho desde el principio (§7.2.1) que toda entidad se plantea como un proceso. No obstante, éste será normalmente compuesto, y reunirá subprocesos de naturalezas muy diferentes. En concreto, una parte de ellos se dedicará al establecimiento y mantenimiento de la *estructura* que define la instancia; el resto, en cambio, describirá el *comportamiento* de la misma. Estos se corresponden, respectivamente, con los mencionados aspectos estáticos y dinámicos, y también con los correspondientes sublenguajes de $\mathcal{P}i\mathcal{L}ar$. El resto de este apartado se dedicará a esbozar brevemente esta configuración.

Se puede resumir esta última distinción, con un ligero abuso de lenguaje, diciendo que una instancia se compone de una *estructura* y un *proceso*. La primera es, en líneas generales, parecida a la descrita para un puerto: esto significa que consta de un registro con cierto formato, y de una serie de procesos predefinidos que le dotan de un comportamiento básico, ligado a la forma de dicho registro. El proceso, por su parte, procede de las restricciones especificadas con el Lenguaje Dinámico, y describe el comportamiento de la instancia. La relación entre ambos es directa: los elementos manipulados por el proceso son los definidos por la estructura.

El aspecto fundamental, al considerar la traducción al cálculo-π tanto de la estructura como del proceso, es la naturaleza inherentemente *dinámica* de la que se ha dotado a $\mathcal{P}i\mathcal{L}ar$. Si no se tuviera esto en cuenta, sólo sería necesario realizar una traducción casi literal, que no tendría por qué ser, en principio, demasiado compleja. En tal caso, la estructura consistiría en una simple representación de los elementos que constituyen un componente –su interfaz, las instancias que contiene y las conexiones entre ellas–, y el proceso sería simplemente una versión de bajo nivel de las reglas indicadas, que recibiría el conjunto de dichos elementos como parámetro.

Sin embargo, el dinamismo inherente en $\mathcal{P}i\mathcal{L}ar$ hace que el esquema final sea, en realidad, mucho más complejo. Es cierto que la *estructura* procede, en principio, de la especificación estática del Lenguaje Estructural; pero no permanece inalterada, ya que puede ser modificada por el proceso: en su interior, pueden crearse y destruirse instancias, puertos o conexiones, sin límite alguno. En el *proceso* ocurre algo similar: el conjunto de los elementos a los que afecta se altera de manera continua, por lo que no se puede asegurar la consistencia en el número de sus parámetros, ni siquiera entre sucesivas llamadas recursivas. Además, debe tenerse en cuenta que, aunque el sistema de tipos definido para el cálculo-$\hat{\pi}$ es polimórfico, sigue siendo estático[5]; por tanto, esto plantea un conflicto incluso de cara a la simple *definición*, tanto de la estructura de un componente como del proceso asociado.

La solución más sencilla a este conflicto pasa por agrupar los distintos tipos de elementos en bloques, de modo que se pueda hacer una referencia individual al *conjunto* de los mismos. La propia estructura se definirá como una agrupación de tales conjuntos. A su vez, éstos se conciben inicialmente como registros; pero de nuevo el dinamismo hace esto imposible, por lo que habrán de ser representados como *listas* (§C.4.4). Por su parte, el proceso recibirá *toda* la estructura como un parámetro –en principio único, aunque esto cambiará más adelante–, evitando de este modo los problemas derivados de plantear una recepción individual de sus contenidos.

[5]Como se indica en el Apéndice C, concretamente en la sección C.5, en el cálculo-$\hat{\pi}$ sí se dispone, en realidad, de un tipo dinámico (*Dyn*). Sin embargo, éste sólo será utilizado como sustituto para un tipo primitivo –o una tupla de éstos–, de manera que la definición de un puerto pueda hacerse con independencia de los tipos concretos de datos que transporten sus canales. Utilizarlo en este contexto supondría renunciar a comprobar la consistencia del sistema de tipos, que en tal caso perdería todo su interés.

No obstante, cuando toda la estructura es cambiante, y se tiene plena libertad para crear y destruir cualquiera de sus elementos, ¿qué es lo que realmente la define?. Obviamente, aquello que la dota de identidad, es decir, su *nombre*. Afortunadamente, la unicidad de este nombre está asegurada por las premisas del cálculo-π, del mismo modo que lo estaba en Marmol, donde se definía un conjunto universal de nombres –V_I, en este caso– de potencia análoga.

Precisamente por esto, se reproducen aquí –en principio– todos los problemas derivados de la movilidad de nombres del cálculo, que ya fueron identificados en el caso de un puerto: en particular, la necesidad de establecer un vínculo permanente entre el nombre de un componente y la estructura que lo define. De nuevo, y tal como se advirtió en su momento, la solución es *exactamente* la misma: así pues, se incluye el nombre de instancia como parte de la propia estructura, y ambos se relacionan mediante la invocación de un subproceso RepSelf, creando de este modo una una referencia –autorreferencia– persistente.

Por lo demás, una vez que se asumen los aspectos dinámicos y autorreferenciales, la estructura final que describe a un componente –instancia– resulta bastante obvia, y es en muchos aspectos idéntica a la que se definía en Marmol mediante la noción de *configuración* y la relación de nombrado de instancias ($\mathcal{N}_I$). Constará, por tanto, de tres subestructuras, definidas como listas –en realidad, listas nombradas–. Estas listas proporcionarán el conjunto de sus puertos, es decir, la *interfaz*; el conjunto de las instancias contenidas en su interior, que conforma la estructura jerárquica; y el conjunto de las conexiones definidas entre ellas, que hace posible la interacción. Por supuesto, esta estructura sólo estará completa en una instancia compuesta; en un tipo primitivo, tanto el conjunto de instancias como el de conexiones estarán vacíos.

La definición de un componente es básicamente ésta; no obstante, habrá de ser ampliada debido a su naturaleza reflexiva (§7.2.5). La estructura final resultante es, no obstante, muy similar, y será la que se desarrolle en el apartado 7.3.2.

Conexiones

La definición de las conexiones es totalmente análoga a la de los componentes; de hecho su estructura es, a todos los efectos, un *subconjunto* de la estructura de un componente. Más todavía, en realidad no hay nada, a nivel semántico, que distinga a una conexión, una vez establecida, de una instancia de componente *primitivo*. El tipo definido para ambos conceptos, en términos del cálculo-π, será diferente; no obstante, el de una conexión no es sino un fragmento del de un componente, por lo que frecuentemente se considerará que ambos son perfectamente compatibles. Esto no se refiere al hecho de que mantengan una relación de subtipado –aunque, estrictamente, el tipo de una estructura-componente (*TComp*) será por definición un subtipo del de una estructura-conexión (*TBind*)–, sino que se asumirá la posibilidad de realizar una conversión implícita entre ellos.

La única diferencia realmente importante –y que de hecho será determinante en el desarrollo de la semántica– no se da tanto en el esquema final de su estructura, como en el *modo* en que ésta se construye. Ha de tenerse en cuenta que la tarea principal de una conexión es expresar una *interacción* entre dos –o más– componentes; para ello, su traducción en cálculo-π ha de poder expresar una comunicación entre los procesos que los representan.

Ahora bien, toda comunicación se realiza a través de canales, identificados por su nombre; por tanto, para que dos procesos puedan interactuar, han de manipular los mismos canales, y

conocer los mismos nombres. En un componente, estos canales son parte de los puertos, y se *crean* como parte de éstos, durante la instanciación del componente; así pues, están vinculados a una instancia concreta. Por su parte, una conexión siempre habrá de ser creada *después* que los componentes a los que conecta, ya que se define en función de éstos. De hecho, y dado que ha de ejercer como intermediaria entre ellos, es *ella* la que debe conocer los nombres comunes: esto es, los canales involucrados en la comunicación, que *pertenecen* a los respectivos puertos que los definen, pero habrán de ser utilizados por los suyos propios.

Por supuesto, el hecho de tener que utilizar un canal definido en un proceso diferente no supone problema alguno en el cálculo-π. Sin embargo, sí supone una diferencia con respecto al modo en que normalmente se crea un componente. Al instanciarse, éste crea simplemente sus propios puertos; en cambio, en una conexión, la creación de los puertos ha de utilizar nombres *ya definidos*; y es este acto, en realidad, el que verdaderamente establece una conexión.

Toda la complejidad de la parte estructural, de hecho, derivará únicamente de este detalle, como podrá verse en la sección 7.3.5. Esto se relaciona con dos aspectos; por una parte, se habrá de considerar una forma diferente de crear un puerto, en el caso de que uno de sus extremos exista previamente. Por otra, se debe tener especial cuidado en el modo en que los nombres –privados– de los canales implicados serán transmitidos a lo largo de la estructura, de modo que lleguen hasta la definición de una conexión sin provocar en el proceso una involuntaria extrusión de ámbito. Estos detalles son, en realidad, mucho más sencillos de lo que podría parecer, y serán examinados en la sección correspondiente.

7.2.5 Intuición para un Proceso Reflexivo

El esquema descrito hasta este momento se ha concentrado en diseñar una semántica lo más directa posible para los elementos arquitectónicos –componentes, conexiones y puertos– en términos de procesos abstractos, intentando evitar algunos de los inconvenientes encontrados en otros lenguajes. Por tanto, el rasgo más característico de $\mathcal{P}i\mathcal{L}ar$, es decir, su naturaleza reflexiva, ha sido explícitamente ignorado hasta ahora. Con esta omisión, se ha elaborado un modelo bastante sencillo, comparable en muchos sentidos a la versión inicial de Marmol ($\aleph_\Sigma$). En resumen, se dispone de tres tipos de entidades, fuertemente ligadas entre sí. Dos de ellas muestran un carácter dual, lo que significa que se componen de una *estructura* pasiva y de un *proceso* activo. La tercera, por su parte, solamente tiene estructura.

En esta sección, se considera por fin la adición de reflexión, introducida a la manera de $\mathcal{P}i\mathcal{L}ar$, y el impacto que ésta tiene sobre el esquema básico descrito. De este modo, se pretende separar con claridad los aspectos reflexivos de los meramente arquitectónicos, y determinar hasta qué punto crece la complejidad por esta causa. Por otra parte, la exposición incremental de la semántica ha de contribuir a facilitar su comprensión: la versión final del modelo se construye sobre la base de los procesos abstractos ya conocidos.

Para comenzar, se debe considerar qué es lo que implica el concepto de *reflexión* en un contexto concurrente como el que se plantea en un Adl arquitectónico. Para ello, merece la pena retomar la taxonomía de tipos de reflexión presentada en la sección 4.4.1, que distingue entre *introspección*, reflexión *estructural* y reflexión *de comportamiento*. Cada uno muestra un estilo de autorreferencia diferente; dado que $\mathcal{P}i\mathcal{L}ar$ fue diseñado con el objetivo de ser lo bastante general como para poder expresarlos todos, su semántica debe tener en cuenta a los tres.

El primero requiere que, en su fase activa, el sistema tenga acceso a cierta información de tipo estructural y evolutivo –meta-información, en definitiva– que de otro modo no estaría disponible. El segundo, que esa información incluya a los tipos abstractos de datos, así como un control efectivo de los esquemas de generación y destrucción de sus instancias. Y el tercero, finalmente, define la posibilidad de alterar la dinámica de un proceso a voluntad, en cualquier momento y de cualquier manera imaginable.

Los dos primeros exigen, simplemente, que *lo que era implícito se haga explícito*. Esto hace preciso introducir algunas definiciones nuevas, pero en realidad implica una extensión de la semántica bastante directa. Datos que antes se manipulaban de manera transitoria se hacen ahora persistentes; información que sólo estaba presente de manera indirecta emerge en la forma de estructuras complejas; e incluso ciertos procesos, antes definidos como internos y limitados a una fase concreta, se hacen accesibles y pueden ser invocados de manera genérica.

La reflexión de comportamiento muestra unas implicaciones aún mayores, incluso desde un punto de vista teórico. En el caso más habitual, como ya se ha dicho, este tipo de reflexión se lleva a cabo mediante un mecanismo de *interposición*: un determinado elemento "se pone en medio", de modo que interfiere en una invocación dada. Las llamadas –y respuestas– se capturan, desvían o difieren; las entradas –y salidas– se alteran, eliminan o sustituyen. Esto ocurre normalmente, aunque no únicamente, en un contexto secuencial. Ahora bien, esta descripción resulta insuficiente en un modelo concurrente, en especial cuando está basado en un álgebra de procesos. Por el contrario, el encadenamiento de las acciones debe asegurarse mediante el establecimiento de un protocolo[6] definido de manera explícita. Pero además este protocolo podrá intervenir en la interacción de cualquier modo; un meta-proceso no tiene por qué limitarse a intervenir *antes* o *después* de un proceso determinado, ya que ambos pueden desarrollar su ejecución en paralelo: las limitaciones por secuencialidad han desaparecido.

Para tener el control absoluto que la reflexión exige, el meta-proceso debe poder acceder en todo momento a cada uno de los aspectos que definen el comportamiento del proceso-base. Esto implica tener acceso, tanto en lectura como en escritura, a todos sus posibles estados y elementos internos. Sin embargo, esto resulta plenamente coincidente con un concepto preexistente en teoría de la concurrencia, a saber, la noción de *superposición* [BF88, Bos99, Kat93]. Se considerará pues, a todos los efectos, que *reflexión de comportamiento y superposición son equivalentes* en un contexto concurrente. Ésta es una conclusión personal; pero no es, sin embargo, la primera vez que ha sido presentada. De hecho, ha sido sugerida en fechas muy recientes por autores muy vinculados a la definición original del concepto, tales como Shmuel Katz [KG99] e Ira Forman [For00], que ha llegado incluso a señalarlo de manera explícita.

La superposición ha sido definida como una operación especial en álgebras de procesos del tipo Csp [BF88]. Sin embargo, parece posible expresarla de manera indirecta en una variante del cálculo-π. Como se ha dicho, para ello es necesario que el proceso superpuesto, o meta-proceso, tenga acceso tanto a los estados como a los elementos internos del proceso original. Lo primero viene dado por el protocolo meta-base ya citado: al poder intervenir en cualquier punto de la definición, es obvio que puede detectar y alterar cualquier estado. Lo segundo es aún más

[6] El papel de este protocolo es análogo, pero no idéntico, al de un Protocolo de Meta-Objeto (Mop) en un sistema reflexivo orientado al objeto (§4.3.3). En especial, no vendrá predeterminado por las reglas del lenguaje, sino que deberá ser especificado como parte de la definición del proceso concreto involucrado.

sencillo, ya que los elementos mencionados pueden ser únicamente *canales*. Por tanto, se trata solamente de que todos estos canales estén disponibles para ambos procesos; de hecho, algunos de ellos deberían haberlo estado en cualquier caso, o de otro modo la comunicación no sería posible. En resumen, esto implica poco más que una interacción muy estrecha y una *fusión de ámbitos*. Todo esto cristalizará eventualmente en una fusión de procesos, que será examinada con mayor detenimiento un poco más adelante.

No es la primera vez que se propone el uso de superposición en Arquitectura de Software. Michel Wermelinger ha utilizado el concepto tanto para introducir dinamismo en su modelo categórico [WF99] (§3.9.4), como para extender el concepto de conector [WLF00]. Sí es, sin embargo, la primera vez que se introduce desde este punto de vista. La propuesta puede parecer compleja, pero define un modelo mucho más sencillo que su equivalente secuencial. Es, sin embargo, mucho más potente y flexible; de hecho, la interposición clásica, que da lugar a un modelo *de capa de cebolla*, puede describirse fácilmente como un caso particular del protocolo de superposición. Esto ya se había implicado, de manera indirecta, durante la definición de la sintaxis, al definir la variedad *envolvente* (**wrap**) de un componente (§6.12.1). Allí ya se señalaba que un esquema de capa de cebolla puede ser descrito simplemente mediante su protocolo, y que la extensión sintáctica se proporcionaba únicamente con el fin de facilitar la especificación y utilización de un caso particularmente habitual.

En cualquier caso, el desarrollo de $\mathcal{P}i\mathcal{L}ar$ viene guiado fundamentalmente por su perspectiva reflexiva, que no se reduce además a los aspectos de comportamiento; por tanto, la terminología de reflexión se mantendrá en adelante, como se ha venido haciendo hasta ahora. En resumen, se parte de que la arquitectura de software consiste, ante todo, en información estructural. En nuestro modelo, esta información se hace explícita, y se dotan los medios para alterarla, de modo que lo que cambiará, en definitiva, es la propia estructura. Esto es reflexión, sobre todo del tipo estructural; pero, como ya se ha dicho, es al mismo tiempo una forma de dinamismo, precisamente la que se está abordando en esta tesis.

Reuniendo todo lo que se ha dicho, y aplicándolo al modelo inicial descrito en los apartados anteriores, se va obteniendo una serie de consecuencias, encadenadas entre sí de manera sucesiva, que van dibujando, paso a paso, un nuevo esquema; éste puede resultar mucho más complejo, pero es ya totalmente reflexivo. A continuación se examinarán estas consecuencias:

☐ 1. **Puertos No Reificados**. *Como ya se indicó en la sintaxis, los puertos no admiten reificación; su estructura es fija y no puede ser alterada.*

Desde un punto de vista sintáctico, los puertos se consideraban atómicos. Como ya se ha dicho, semánticamente tienen una estructura, pero ésta es idéntica para todos ellos: está totalmente predeterminada y no depende de aspectos particulares. De hecho, una de las funciones de los puertos será la de actuar como "ancla" conceptual de la semántica: son el único elemento cuya función y comportamiento es perfectamente conocida en todo momento.

Esto no significa que la reflexión les sea ajena. Por el contrario, son el medio que hace posible muchos de los comportamientos reflexivos del sistema, tales como la detección, captura o simulación de interacciones. Además, al igual que el resto de elementos, almacenarán también la meta-información correspondiente, contribuyendo así al funcionamiento de los operadores introspectivos. Todo esto se habrá tenido ya en cuenta durante la definición de la estructura de

puerto, que por tanto se concibe como totalmente integrada en el esquema reflexivo. Lo único que aquí se está indicando es que no existe modo alguno de alterar dicha estructura.

□ 2. **Arquetipos Explícitos**. *Los tipos de componente, que eran tratados de manera implícita, se hacen ahora explícitos, y definen una nueva estructura.*

En la sección anterior (§7.2.4), al hablar de los componentes, se indicó sólo era necesario modelar como procesos a las instancias, y que los arquetipos actuaban meramente como patrones estructurales. No obstante, esto ya no es cierto al introducir reflexión. Por definición, la reflexión estructural exige un acceso completo a los tipos abstractos de datos (§4.4.1); esto significa, en este contexto, que las definiciones de arquetipo han de hacerse explícitas, esto es, que deben de quedar accesibles como una entidad para el resto del sistema.

Esto significa, en resumen, que es preciso definir un nuevo tipo de *estructura* para modelar a los arquetipos. En principio, esta estructura guardará cierta semejanza con la de una instancia de componente, pero en realidad será mucho más compleja. Debe de tenerse en cuenta que almacenará toda la información necesaria para realizar una instanciación; esto incluye información que no es preciso mantener en la estructura de una instancia, porque ya habrá sido procesada durante su creación. De nuevo, podrá establecerse una analogía bastante cercana entre esta estructura de arquetipo –que será descrita en la sección 7.3.4– y la forma que tomaba una declaración de tipo en Marmol, detallada en la definición 5.30 y subsiguientes. En realidad, hay en ésta toda una serie de detalles suplementarios que allí no eran considerados: pero todos ellos se refieren a la parte activa, y no guardan relación con este punto, sino con el siguiente.

Los arquetipos no son la única metainformación que se hace explícita. De hecho, ya se ha mencionado que parte de la misma estará contenida en los puertos. Sin embargo, éste es el único aspecto que exige la creación de una estructura completamente nueva. Por razones que quedarán claras en los dos puntos siguientes –y que básicamente coinciden con los postulados del modelo $\aleph_\Pi$–, el arquetipo, y por tanto su estructura, será asimilado con un metacomponente. Este es el motivo por el que, a partir de este momento, se utilizará este último nombre –metacomponente– para referirse a la totalidad de la entidad.

En definitiva, para todo arquetipo se define una estructura de metacomponente, que se utilizará para crear instancias, con las que quedará de algún modo ligada. Esto implica que toda instancia del sistema tendrá al menos *tres* partes: su propia estructura, ya conocida; un proceso, que define su comportamiento; y la estructura de su metacomponente. Por supuesto, ésta última es única, y será por tanto un elemento común a todas las instancias del mismo tipo.

Sin embargo, esta estructura ternaria, con ser ya bastante compleja, puede complicarse aún más; entre otras cosas, podrá ser sometida a múltiples reificaciones, como se verá más abajo.

□ 3. **Dualidad de los Meta-Componentes**. *Todo metacomponente define una parte estructural pasiva, que ejerce una influencia sobre sus avatares; pero puede desarrollar, además, una parte activa, que lo define como un componente del nivel superior.*

Como se sabe desde $\mathcal{P}i\mathcal{L}ar$, algunos componentes podrán tener una naturaleza dual. En el caso normal, todo arquetipo define solamente la estructura y comportamiento de sus instancias. Sin embargo, como ya se ha dicho en múltiples ocasiones, en $\mathcal{P}i\mathcal{L}ar$ se produce un colapso

implícito entre las relaciones de tipado y de reificación: de modo que un arquetipo es también un metacomponente, y podrá tener un comportamiento propio, actuando como no sólo como un patrón estructural, sino también como una *instancia situada en el nivel meta*.

En último término, un metacomponente puede ser tan complejo como se desee; pero en principio, se asume que se le declara como una meta-instancia primitiva, y por tanto sólo tiene interfaz. Esta interfaz puede declararse de dos modos diferentes: en el caso más habitual, se define implícitamente como una *copia* de la interfaz principal de sus avatares; de modo que por cada puerto P declarado en un arquetipo no sólo habrá un puerto ($\alpha.P$) en cada avatar, sino también un "meta-puerto" ($\gamma.P$) en el metacomponente. Por otra parte, puede definirse también de manera explícita –mediante la palabra clave **metaface**–, constituyendo lo que en la sintaxis se denominaba una interfaz de nivel meta, o abreviadamente *metafaz*. En cuanto a su comportamiento, en principio se deduce a partir de las reglas –restricciones– especificadas en el arquetipo, tomando las partes que se refieren específicamente al metacomponente.

La naturaleza dual de un metacomponente, por tanto, se expresa como una *parte pasiva*, formada por la estructura de meta-información asociada con su arquetipo, y una *parte activa*, constituida por la instancia de nivel meta, que es el metacomponente propiamente dicho. Sin embargo, la entidad no tiene por qué ser siempre dual: cuando no se le define ningún tipo de comportamiento específico, y ninguno de sus (meta-)puertos es accedido, la parte activa simplemente *no existe*. El potencial para crearla sigue estando presente, pero no es explotado; de este modo se evita duplicar entidades innecesariamente.

Por tanto, la mitad activa puede existir o no; pero cuando exista, irá siempre asociada a la estructura pasiva, de modo que conforman una pareja con un esquema inusual. Se tiene, por un lado, a la parte activa, que es una instancia con su propia estructura, pero que *a priori* no tiene asociado un arquetipo –salvo que sea, a su vez, explícitamente declarado; pero esto no es obligatorio–. Por otro lado, se tiene a la estructura pasiva; ahora bien, desde el punto de vista de la instancia, ésta no es ni su propia estructura, ni la de su arquetipo. En realidad, son dos aspectos de la misma entidad, "gemelos" de distinta naturaleza situados al mismo nivel. Para expresar esta relación se dirá, del mismo modo que en Marmol (concretamente en el modelo $\aleph_{\Pi}$, ecuación 5.104), que la estructura pasiva es el **brhátar** de la instancia activa.

En principio, ambas mitades están ligadas, y cada una de ellas conoce a la otra, si ésta existe. Sin embargo, la relación es más importante en el sentido indicado –es decir, la activa conoce a la pasiva–, ya que le permite deducir su propia naturaleza. Ocurre que toda instancia del sistema, sea cual sea el meta-nivel al que esté definida, podrá ahora saber si forma parte de un metacomponente sin más que comprobar si tiene –o no– un *brhátar*. En tal caso, éste será el arquetipo de los avatares de dicho metacomponente.

Resumiendo, el esquema triple del punto anterior se hace ahora quíntuple, en el caso de que el metacomponente tenga un comportamiento específico. Los cinco elementos que lo constituyen son la estructura y proceso de la instancia, la estructura de su arquetipo, y la estructura y proceso de la meta-instancia activa. Sin embargo, este esquema será simplificado más adelante mediante una fusión de los dos procesos, de modo que el esquema final será cuádruple.

□ 4. **Enlace de Reificación**. *Se define la noción de* reificación *como una estructura explícita, que expresa las relaciones entre meta-niveles.*

Una vez que se ha introducido –en el punto anterior– un metacomponente, se hace necesario proporcionar una organización coherente para todo el esquema. Con este fin se introduce la noción de *reificación*, concebida como un enlace bidireccional entre un componente (instancia) y su metacomponente. No se repetirá aquí la descripción del esquema definido por las reificaciones, ya que es el mismo que fue expuesto con todo detalle en el modelo $\aleph_{\mathrm{M}}$ de Marmol. Tan sólo se recordará brevemente parte de la terminología utilizada, según la cual:

- Una *reificación* es una estructura de datos que expresa una relación bidireccional entre dos elementos de distinto nivel de abstracción. En este caso, es que relaciona a (la estructura de) una instancia con (la estructura de) su arquetipo.
- Se denomina *avatar*, desde el punto de vista de un metacomponente, a cada uno de las instancias a las que reifica.
- Se denomina *metacomponente* o *rohatar*, desde el punto de vista de una instancia, a cada uno de los elementos en los que se refleja, esto es, que mantienen una reificación hacia ella.
- El esquema de reificaciones divide al sistema en *niveles* o *meta-niveles*, de modo que todo elemento reificado está en un nivel inferior al del elemento que lo reifica. Se asume que las asignaciones de nivel son consistentes.

Por tanto, la relación entre avatares y rohatares se distribuye en una red de reificaciones, que estratifica el sistema en metaniveles.

No obstante, en los dos puntos anteriores se mostró que en $\mathcal{P}i\mathcal{L}ar$, como es sabido, *el tipado se* ***colapsa*** *con la reificación*, con el fin de facilitar la reflexión estructural. Esto significa que, como se ha visto, todo arquetipo se expresa como una estructura, que *reifica* a cada una de sus instancias. Sin embargo, debe tenerse en cuenta que el inverso no es cierto: una vez definida la relación, no hay nada que impida establecer una reificación entre un componente y cualquier otro del nivel superior, aunque éste no sea –esté unido a– su arquetipo. Este aspecto será considerado como séptimo punto de este apartado.

Al igual que todo lo anterior, este esquema es perfectamente consistente con las premisas del modelo $\aleph_{\Pi}$ de Marmol, en el que se basa la definición de $\mathcal{P}i\mathcal{L}ar$.

□ 5. **Superposición por Ámbito Compartido**. *La reflexión de comportamiento se expresa como una superposición no explícita, cuyo resultado es un proceso cuyo ámbito abarca a las entidades implicadas en distintos meta-niveles.*

En la estructura –quíntuple– descrita hasta el momento, existen en principio *dos* procesos activos. Uno se corresponde con la instancia de nivel base; el otro con la de nivel meta. Sin embargo, ambos habrán de ser generados a partir de las reglas $\mathcal{P}i\mathcal{L}ar$ definidas para el mismo arquetipo. Por otra parte, y tal como se ha dicho al principio de este apartado, el segundo proceso –meta– se *superpone* al primero –base–, para expresar reflexión de comportamiento.

No obstante, considerando ambos aspectos en detalle, se hace evidente que el esquema es, en realidad, más simple. En primer lugar, como ya se dijo durante la definición de la sintaxis, las reglas de $\mathcal{P}i\mathcal{L}ar$ de describen *desde la perspectiva del nivel meta*, y actúan entonces sobre su nivel base. Debido a esto, toda referencia de una regla a un elemento del rohatar se hace en primera persona (**self**), mientras que toda referencia a la estructura del nivel base se hace en

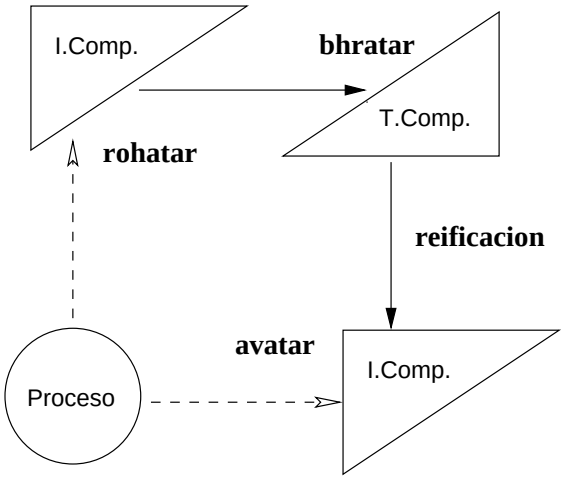

Figura 7.1: Estructura Cuádruple de una Instancia Reificada

tercera persona (**avatar**). Sin embargo, la reificación se define como bidireccional, por lo que en realidad ambos niveles se afectan mutuamente.

En una regla $\mathcal{P}i\mathcal{L}ar$, las distintas referencias a elementos de los dos niveles están intercaladas, definiendo un protocolo que pasa explícitamente el control de uno a otro. Extraer *dos* procesos a partir de esta regla exigiría separar las menciones explícitas a cada nivel, obteniendo dos secuencias diferentes; pero el protocolo definido habrá de ser respetado. Esto significa que en cada una de las secuencias se habrán de insertar las adecuadas sincronizaciones y comunicaciones, de modo que el control sea transferido de una a otra, manteniendo el mismo esquema. De este modo, se habría realizado una conversión bastante compleja, para obtener finalmente un par de procesos cuya composición es "bisimilar" con la regla original. Por tanto, todo esta transformación es en el fondo ilógica, ya que deja todo como estaba.

En segundo lugar, la superposición exige que un proceso tenga total acceso a los variables y estados del otro. Como ya se ha dicho al principio, en un lenguaje como el cálculo-π esto significa, simplemente, que *todo canal* del otro proceso es accesible, ya que los estados vienen determinados únicamente por la secuencia de términos de una fórmula concreta. Esto equivale a decir que los dos ámbitos se han fusionado, o que los dos procesos comparten el mismo ámbito.

En definitiva, si los dos procesos tienen el mismo ámbito, y su comportamiento conjunto es idéntico al del proceso original, queda claro que *no hay motivo alguno para dividir dicho proceso* en dos; dicho de otro modo, el proceso definido para el avatar y el establecido para el rohatar son el mismo. En cierto modo, se ha realizado una fusión entre ambos procesos, aunque lo que ocurre realmente es que nunca han llegado a estar separados.

Se tiene, en definitiva, un único proceso, que accede por igual a elementos del nivel base y del nivel meta, contenidos respectivamente en las estructuras del avatar y el rohatar. El modo más sencillo de ver este aspecto es considerar que el proceso recibe dos parámetros, que son precisamente estas dos estructuras, ubicadas en distintos niveles. Sin embargo, una vez que éstas están dentro de su ámbito, puede manipularlas con entera libertad.

En definitiva, el esquema quíntuple pasa ahora a adquirir una estructura cuádruple; ésta es la que puede verse en la Figura 7.1, y ya es, por fin, la definitiva. Podrá crecer por efecto de otras reificaciones y elementos externos, pero en su forma más básica será siempre ésta.

Puede parecer que esta "fusión" tiene un aspecto problemático, y es que el comportamiento del metacomponente se vincula estrechamente a un avatar concreto. Aparentemente, esto podría provocar un conflicto cuando hubiese varios avatares; pero como se verá a continuación, esto no es cierto en absoluto.

☐ 6. **Multiplicidad de Avatares**. *Todo metacomponente podrá ser reflejado en muchos avatares, y supondrá una referencia común para todos ellos.*

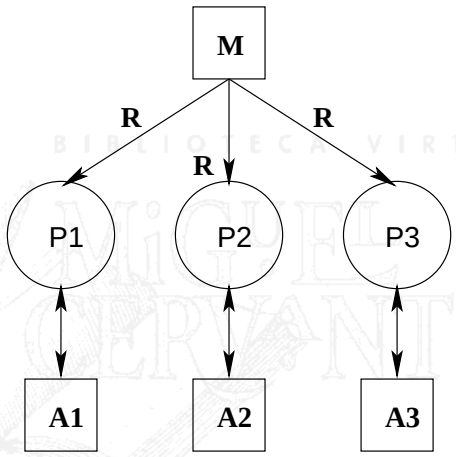

Figura 7.2: Reificaciones de un Meta-Componente

Como es sabido, todo metacomponente puede reificarse en un número indeterminado de avatares. Ahora bien, acaba de verse que el comportamiento de un metacomponente estará combinado con en el mismo proceso que describe el comportamiento de un avatar. ¿Qué ocurre, entonces, cuando se tienen varios avatares?

La solución es sencilla, y es la que se muestra en la Figura 7.2. Simplemente, se crea una *copia* del mismo proceso para cada avatar. En realidad, esto no puede ser más lógico: todos los avatares del mismo arquetipo han sido creados con base en el mismo patrón, por lo que su proceso habría de ser en cualquier caso idéntico. Esto es idéntico, además, a lo que ya se dijo al definir la sintaxis: el uso en una regla de la palabra **avatar** se refiere a *cada uno* de los avatares, considerado individualmente.

Todos estos procesos idénticos se diferencian únicamente en sus parámetros: cada uno accede a la estructura de su propio avatar. Sin embargo, todos acceden al mismo rohatar, ya que metacomponente no hay más que uno. Lo único que se multiplica es el *proceso*, no la estructura. Esto permite utilizar dicha estructura –es decir, a los puertos y elementos del rohatar– como referencia común entre todos los avatares, que de este modo definen un conjunto.

Esto implica un acceso concurrente de múltiples procesos a la misma estructura. A menudo esta concurrencia no será conflictiva; en otros casos, tendrá un comportamiento indeterminista;

y en otros, por fin, será explícitamente buscada y desarrollada, y permitirá explotar el hecho de que esta estructura sirve de referencia común a todos los miembros de un conjunto cambiante –el **avatarSet**–. Esto permite plantear esquemas muy complejos de manera sencilla.

No obstante, éste no es el único tipo de multiplicidad existente en el sistema reflexivo. De hecho, podrá darse igualmente el enfoque contrario:

☐ 7. **Multiplicidad de Reificaciones**. *Todo avatar podrá ser reificado por múltiples rohatares; su comportamiento queda descrito como la composición paralela de los distintos procesos que define cada uno de ellos.*

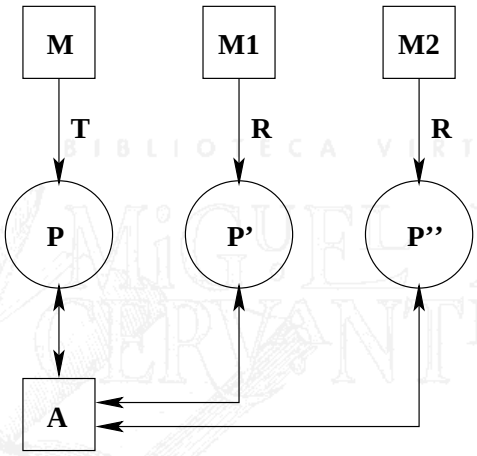

Figura 7.3: Reificaciones sobre un mismo Avatar

Hasta ahora sólo se ha considerado la reificación procedente del tipado. No obstante, como es sabido, un avatar puede ser reificado, a lo largo del tiempo e incluso de manera simultánea, por múltiples metacomponentes, cada uno de los cuales realiza su propia aportación.

Como acaba de verse, un rohatar influencia directamente a su avatar, debido al hecho de que comparten el mismo proceso –punto 5–, y la existencia de varios avatares implica múltiples copias del mismo proceso –punto 6–, cada una de las cuales se refiere a un avatar diferente. Esto implica que, por cada reificación de un avatar, existe un rohatar que intenta expresar su influencia, aportando una copia de su propio proceso. No obstante, esto significa que habrá varios procesos de este tipo. ¿Cómo se integra, entonces, esta influencia múltiple?

La respuesta es sencilla: por simple composición paralela. En efecto, dado que cada metacomponente aporta su propio proceso, lo que se tiene es múltiples procesos actuando simultáneamente sobre el mismo avatar, tal como se muestra en la Figura 7.3. La situación es exactamente la inversa a la del punto anterior: ahora, cada proceso tiene como parámetro un rohatar distinto –aquél que lo define–, pero todos actúan sobre el mismo avatar. No obstante, ambos esquemas son perfectamente compatibles; de hecho, normalmente se darán de manera simultánea.

Por tanto, los distintos procesos tienen de nuevo un elemento común, que es el avatar. Sin embargo, a diferencia del caso anterior, este aspecto no será habitualmente explotado de manera

consciente, ya que cada proceso tiene un origen diferente. No obstante, es evidente que, al desarrollar cada proceso su propia dinámica sobre el *mismo* elemento, se producirán toda una serie de sincronizaciones –e interferencias– entre los ellos, que darán lugar a un comportamiento conjunto. A menudo la mezcla podrá ser incluso indeterminista, que es precisamente el efecto deseado: de este modo, podrán reflejarse las situaciones conflictivas nacidas de la mezcla de distintos estilos, esto es, el desajuste arquitectónico [GAO95].

Es relevante señalar que todo rohatar, aparte del proceso, tiene también su propio brhátar, que consiste en una definición de arquetipo. En éste se definen, de hecho, los *nombres* –de nivel base– que serán utilizados en la definición del proceso. Obviamente, algunos podrán coincidir con los definidos por otros arquetipos, dando lugar a las sincronizaciones ya mencionadas. Sin embargo, cada uno podría incluir toda una serie de elementos nuevos: normalmente puertos, pero también sub-instancias y conexiones. Todos estos elementos han de existir en el nivel base, por lo que han de integrarse en el avatar. Por tanto, la parte dinámica de la reificación afecta a su proceso, pero la parte estática afecta a su *estructura*. El método seguido para su integración es el obvio: todo elemento nuevo se *añade* a la estructura, mientras que todo elemento común se mantiene. De este modo, cada uno de los procesos tiene acceso al conjunto de elementos necesario, pero los distintos "vocabularios" se solapan en sus puntos comunes.

□ 8. **Grado de Reflexión**. *La definición de un metacomponente puede involucrar directamente a elementos de meta-niveles no adyacentes. Esta influencia se expresa mediante la noción de* grado de reflexión*, que extiende el esquema conceptual al caso más general.*

La estructura de reificación no se limita a dos niveles. Aunque implícitamente se ha estado suponiendo esta idea hasta este momento, y ciertamente es la situación más habitual, no es en modo alguno la única configuración posible. Pero aquí no se trata, únicamente, de tener en cuenta que pueden existir múltiples niveles –algo ya conocido desde el punto 4–, sino de considerar que la relación entre elementos no tiene por qué limitarse a involucrar a niveles adyacentes.

Como es sabido desde la definición de la sintaxis, las restricciones de un componente se conciben *desde el nivel meta*, esto es, desde la perspectiva del metacomponente asociado. En el caso más habitual, sólo hacen referencia a los elementos de un avatar al que reifican –en en nivel inferior–. Sin embargo, una restricción puede también manipular a un avatar de éste –es decir, un avatar de un avatar–, esto es, dos niveles más abajo. De hecho, no hay límite al número de niveles que se pueden descender, siempre que el nivel exista y contenga avatares ligados al metacomponente a través de algún meta-camino (véase la definición 5.66).

Desde este punto de vista, lo esencial de un metacomponente no es su nivel, como tal valor absoluto –su meta-altura–, sino *el número de niveles que necesita descender*, según su especificación, en la cadena de reificaciones. Este número es un dato fijo, derivado de la propia definición del componente, y que puede obtenerse por simple inspección de las reglas, seleccionando el acceso más profundo que se produzca en cualquiera de ellas, es decir, la cadena de cualificadores *avatar.avatar...* más larga. Obviamente, basta *un* acceso de este tipo para que sea necesario considerar el nivel implicado, incrementando el número como corresponda.

A este número se le denomina *grado de reflexión*, y se denotará mediante el símbolo $\Rho$. Su nombre deriva del hecho de que expresa un *descenso* en la cadena de reificaciones, en la dirección comúnmente conocida como reflexión. Sobre todo, no se le ha de confundir con el

grado de introspección ($\Im$) definido en Marmol, que no expresaba una propiedad de cada metacomponente, sino del sistema completo. De hecho, al concepto que más se parece es al de *meta-profundidad* absoluta de una instancia –véase la definición 5.70 y los corolarios 5.77 y 5.80–, ya que expresa el mayor número de reflexiones (estrictas) recorridas[7].

Ahora bien, ha de considerarse cómo afecta esto a la multiplicidad de procesos descrita en los puntos anteriores –concretamente en el sexto–. Se había determinado que la influencia de un metacomponente sobre su avatar se concretaba en un proceso, cuyo ámbito abarcaba a ambos niveles. Por otra parte, todo avatar tenía su propia copia del proceso: de modo que puede considerarse que el proceso es "del avatar", y que simplemente tiene acceso a los nombres el metacomponente. Sin embargo, en la situación actual, el proceso puede involucrar a elementos de varios niveles: en realidad, de cualquier entidad que participe en la cadena de reificaciones que lleva del metacomponente al avatar más profundo accedido. Parece lógico mantener el esquema de un único proceso: esto implica, por tanto, que el *ámbito* de ese proceso abarca a *todos los componentes del meta-camino*.

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

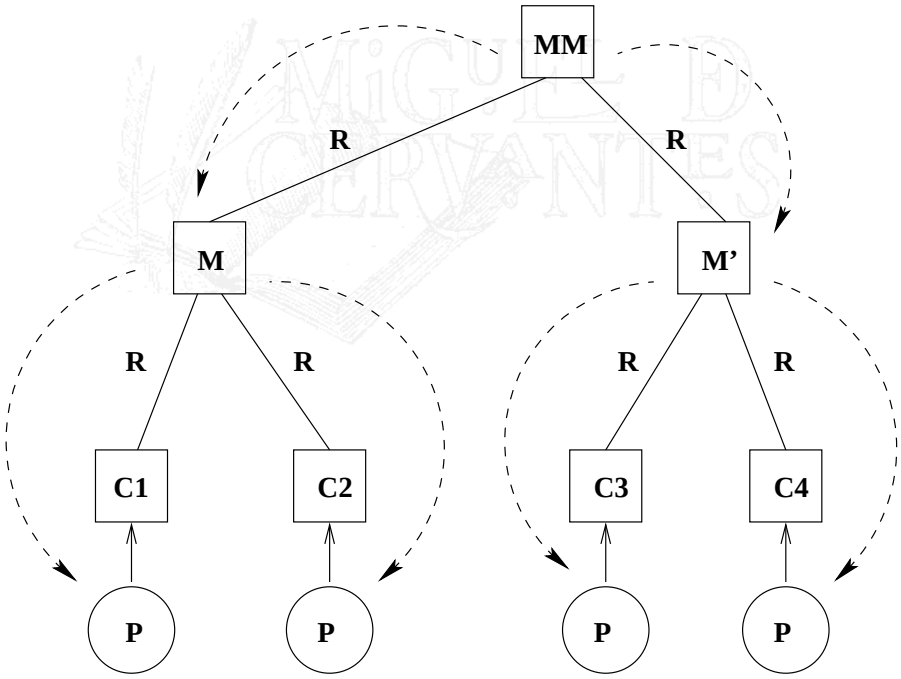

Figura 7.4: Reificación Múltiple para un Proceso con $\eth = 2$

Para mayor claridad, se explicará este punto mediante un ejemplo, que se muestra en la Figura 7.4. En éste, se limita el grado de reflexión $\eth$ a dos: es decir, se trata del caso en el que una restricción accede al avatar de su avatar. El elemento *MM* situado en la raíz del árbol es,

[7] Pero no son idénticos, ya que las reglas de un componente no tienen por qué recorrer toda reflexión establecida; y además, dada la Segunda Regla de Estratificación, se debe considerar también cuál es la meta-altura del nivel más bajo accedido. No obstante, son conceptos relacionados: $\eth \leq \mathfrak{H}(\underline{c})$.

por tanto, un meta-metacomponente.

Como se acaba de decir, la extensión a tres –o más– niveles no afecta a los argumentos expuestos en el punto quinto: por tanto, se ha de tener todavía un proceso para cada avatar, cuyo ámbito es compartido con el de su rohatar. Sin embargo, ahora un proceso puede hacer referencia, simultáneamente, a elementos del rohatar (**self**), del avatar de éste (**avatar**) o del avatar de este último (**avatar.avatar**). Por tanto, y dado que sigue siendo un único proceso, su ámbito ha de fusionar el de los tres niveles. Se puede considerar que para $\eth = 2$, entonces, las fusiones no son ya de dos, sino de tres procesos.

Obsérvese que, como antes, los argumentos de cada proceso son diferentes: un proceso asociado al componente C_1 de la Figura habrá de poder acceder al rohatar M y al meta-metacomponente *MM*. En cambio, el correspondiente al componente C_3 habrá de acceder al rohatar M', aunque finalmente llegará también hasta *MM*. En definitiva, cada proceso ha de tener en cuenta el meta-camino a través del cual fue creado, ya que en todo momento conservará el acceso a cada una de las instancias que participan en el mismo.

En todo lo dicho arriba se implica, por tanto, que la expansión de procesos dada por la multiplicidad de avatares se desarrolla a lo largo de todos los niveles involucrados. Ya no se tiene que tener un proceso por cada avatar, sino un proceso por cada *avatar del nivel más profundo*. En el ejemplo de la Figura, *MM* tiene dos avatares, cada uno de los cuales tiene a su vez otros dos: por tanto, la reificación con $\eth = 2$ genera *cuatro* procesos. Debe de tenerse en cuenta que el esquema ya está simplificado: gracias a la fusión de ámbitos, no es necesario tener un proceso por cada uno de los componentes del esquema –los siete de la Figura–, sino sólo por cada uno de los avatares del nivel inferior –es decir, las hojas de la cadena de reificaciones–.

Un último comentario hace referencia al modo en que se reciben los parámetros en el proceso. Obviamente, la definición abstracta de un proceso reificado asume tantos parámetros como sea necesario. Sin embargo, lo que acaba de decirse asume que un proceso con $\eth = 1$ tendrá dos parámetros –su avatar y su rohatar–, uno con $\eth = 2$ tendrá tres, y así sucesivamente. Tener procesos con distinta signatura complica innecesariamente la traducción: por tanto, se agruparán todos los parámetros variables en una única estructura. De este modo, todo proceso reificado tendrá únicamente un parámetro[8].

La estructura que reune estos parámetros será, obviamente, una lista, que podrá verse como una "torre" de meta-instancias. El primer elemento de la lista –el más alto de la torre– será la meta-instancia de mayor nivel, esto es, el que define el proceso. En la Figura, se corresponde con *MM*. El último se corresponderá con el punto en donde se detiene el descenso, esto es, el avatar más bajo; en la Figura, cualquiera de los componentes-hoja. Esta estructura, que sólo existe desde la perspectiva del proceso, recibirá ocasionalmente la denominación de *torre reflexiva*, por analogía con el término de Wand y Friedman [WF88, FW84]. Sin embargo, no se trata en realidad del mismo concepto, ya que aquél era una torre de meta-intérpretes, y éste, en cambio, no es más que la torre particular de meta-componentes de un proceso dado.

Se asume, por supuesto, que toda la estructura es consistente. Podría plantearse qué ocurriría si un componente con $\eth = 2$ se aplicase a una instancia del primer meta-nivel (es decir, $\mathfrak{H}(c) = 1$); o si el nivel fuese adecuado, pero no hubiese avatares ubicados a la suficiente profundidad. Pero

[8] En realidad tendrá dos; pero el otro no es más que su señal de terminación, y no afecta en absoluto a este razonamiento, aunque habrá de tenerse en cuenta al desarrollar formalmente este punto.

en realidad, la pregunta carece de sentido: en tal caso, la restricción definida se estaría aplicando en un contexto inadecuado, y por tanto la arquitectura que se define sería simplemente *errónea*. En definitiva, sería una mala especificación, y por tanto no ha de ser considerada.

Como puede verse, la estructura resultante puede mostrar una gran complejidad, incluso cuando no se están considerando las influencias indirectas –niyatares–. No debe pensarse, sin embargo, que ésta es excesiva, ya que será inusual encontrar estructuras con un grado de reflexión elevado ($\Rho \geq 3$), y en éstas la complejidad será inherente al propio sistema. La elegancia del esquema resultante deriva de su consistencia con la noción intuitiva: así, toda entidad puede influenciar a sus avatares; cada uno de éstos actúa de manera independiente, y un proceso puede moverse por los distintos niveles con entera libertad.

Con esto se completa la descripción intuitiva de un modelo reflexivo basado en procesos abstractos. Es obvio que la configuración final obtenida es notablemente más compleja que el esquema del que se partía al principio de este apartado. En primer lugar, todas las entidades implicadas –incluso los puertos– tienen ahora una estructura mucho más elaborada, capaz de almacenar la meta-información requerida. En segundo lugar, se incorporan dos nuevos tipos de entidad, y uno de ellos muestra, además, una naturaleza claramente dual. En tercer lugar, el modelo se estratifica en niveles, creando una complicada red de enlaces de reificación cuya influencia no es sólo directa, sino también indirecta. En cuarto lugar, esta influencia rompe el emparejamiento preexistente entre estructuras y procesos, de modo que un proceso puede actuar sobre múltiples estructuras, y viceversa. En quinto lugar, y precisamente por ésto, se observa que cada componente puede adoptar múltiples roles, según los niveles implicados. En sexto lugar, finalmente, se debe considerar que el comportamiento de una entidad ya no se define de manera unívoca durante su creación, sino que se compone a partir de la influencia de múltiples fuentes, y puede evolucionar de manera significativa a lo largo del tiempo.

Puede parecer, entonces, que la complejidad ha crecido de manera excesiva. En realidad, no es así: es cierto que puede alcanzar un grado notable, pero esto sólo ocurrirá de manera excepcional. Un esquema sólo será complicado si lo es *en sí* la configuración que describe; mientras que un sistema convencional será casi tan sencillo como lo hubiera sido en el modelo inicial, aunque contendrá mayor cantidad de (meta)-información.

Se debe tener en cuenta, además, que apenas se han añadido dos nociones, y sólo la de reificación es totalmente nueva. La de tipo era ya conocida, y lo único que se ha hecho es dotarla de un papel activo. La dificultad conceptual, pues, no crece en exceso; aunque el impacto sobre el modelo es grande, no es más que una consecuencia natural de su expresividad.

Por otra parte, debe señalarse que, incluso cuando un esquema dado pueda parecer demasiado elaborado, su comportamiento queda determinado de modo totalmente preciso. No importa lo complicada que sea la interrelación de procesos y estructuras: una vez establecida la configuración global de un sistema, su dinámica se desarrolla por sí sola. El problema se ha reducido, por tanto, a un conjunto de procesos que interactúan entre sí siguiendo las reglas del álgebra. No es preciso, por tanto, pretender concebir de antemano cualquier posible evolución de una descripción dada; por el contrario, el objetivo de esta formalización es hacer posible determinar sus puntos conflictivos *a posteriori*, para así poder resolverlos del modo preciso.

Para terminar, resulta interesante indicar que el esquema obtenido es, en el caso más general, indeterminista. Esto no es más que una consecuencia natural de su concurrencia, y de hecho

podría darse igualmente en el modelo inicialmente propuesto; pero dado que lo que se pretende describir ahora es un sistema reflexivo, parece significativo resaltar que esto hace que sea totalmente congruente con el Teorema de Limitación de Rideau (§4.3.2). O, dicho de otro modo, que esta construcción de reificación no plantea, en principio, ninguna contradicción teórica.

7.3 Semántica de los Aspectos Estructurales

Los aspectos estructurales se constituyen como la parte más importante de la semántica. No sólo dan forma a todos los elementos que conforman la sintaxis del Lenguaje Estructural, sino que definen el esqueleto sobre el que se desarrollan los aspectos dinámicos. Esto llega hasta el punto de que, una vez conocidos éstos, aquellos resultarán muy sencillos. Ocasionalmente serán incluso triviales, pues lo único que harán será desplegar con naturalidad la dinámica ya implícita en los procesos "estructurales" aquí definidos.

Como en la parte sintáctica, hay cuatro elementos básicos: puertos, componentes, conexiones y reificaciones. De hecho, por efecto de la reificación, y como ya es sabido, a los componentes se les asocian dos grandes estructuras: una en el nivel base –el *componente* propiamente dicho– y otra en el nivel meta –el *metacomponente*–. Por tanto, la parte estructural de la semántica ha de considerar cinco estructuras diferentes, aparte de los procesos asociados, para dar una caracterización completa del esquema básico descrito en la sección 7.2.5.

En los apartados siguientes se examinarán una a una estas cinco estructuras. Una vez que se hayan revisado, se habrá dado una definición formal casi completa de los aspectos estáticos de la sintaxis, y sólo habrá de considerarse la traducción de procesos.

7.3.1 Puerto

Desde el punto de vista de la sintaxis, como ya se sabe, un *puerto* $\mathcal{P}i\mathcal{L}ar$ es totalmente atómico. Sólo puede ser creado, utilizado, y destruido. Es el único elemento que nunca se reduce a un componente; más aún, el único que no admite reificación. Sin embargo, ya se ha explicado que la estructura semántica de un puerto no es atómica. De hecho, es en muchos sentidos la estructura más compleja del lenguaje, en lo que se refiere a su comportamiento (predefinido), aunque también la de menor tamaño. Para facilitar la explicación de la misma, en la Figura 7.5 se proporciona una representación gráfica aproximada de los elementos que la constituyen.

Como en todo elemento estructural, comenzaremos por indicar el *tipo* del cálculo-$\hat{\pi}$ con el que se define el puerto, y que da una idea de su estructura. Así, un puerto $\mathcal{P}i\mathcal{L}ar$ se concibe como un *registro* con siete campos, parametrizado con dos tipos (α y β) que representan, respectivamente, al tipo de datos de entrada y de salida asociados a este puerto.

$$
\begin{aligned}
(\mathit{TPort}\,\alpha\,\beta) \quad \triangleq \quad \mu P.(\,&\dagger I\colon\ (\mathit{Par}\,\alpha\,\beta),\\
&\dagger O\colon\ (\mathit{Par}\,\beta\,\alpha),\\
&\dagger W\colon (\mathit{Par}\ \uparrow\alpha\ \uparrow\alpha),\\
&\dagger M\colon (\mathit{Par}\,\alpha\ \uparrow\beta),\\
&\dagger D\colon\ \mathit{Sync},
\end{aligned}
$$

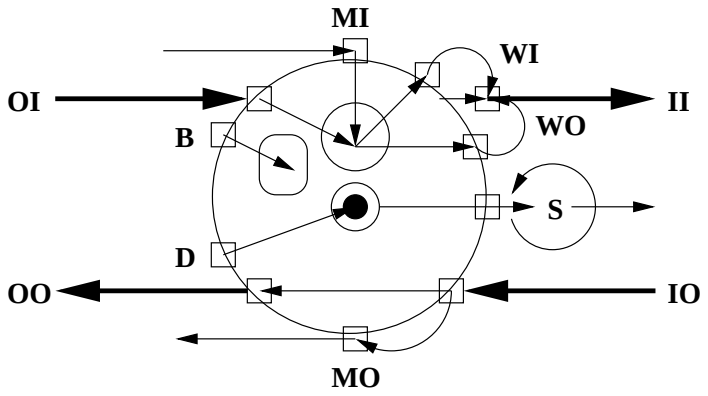

Figura 7.5: Estructura Interna de un Puerto $\mathcal{P}i\mathcal{L}ar$

$$\dagger B : \uparrow(List \uparrow TComp),$$
$$\dagger S : \uparrow P\,)$$

A partir de este tipo y de la Figura, se puede dar ya una primera caracterización del significado con el que se dota a cada elemento de este registro, y que se resume a continuación:

- ☐ **I** $\mapsto$ Se trata de un *par*, por lo que define dos nombres, que en la Figura han sido indicados como **II** e **IO**. En conjunto representan la interfaz interna de un puerto, según lo ya descrito en la sección 7.2.4. Concretamente, toda emisión desde el interior de un componente se hará a través de **IO**. En principio, toda recepción en este mismo punto se hará a través de **II**. Pero en realidad esto último no es obligatorio. Como se ve en la Figura, y como se mostrará en la sección 7.4.4, en realidad el puerto **II** no está vinculado a la entrada, sino que sólo es usado por conveniencia, si así se selecciona en **WI** o **WO**. En cualquier caso, aunque el proceso definido no lo garantiza, normalmente la traducción sí lo hará.

- ☐ **O** $\mapsto$ Análogamente, es un par con dos nombres: **OI** y **OO**. En este caso su interpretación es trivial: **OI** recibe toda entrada desde el exterior, y la introduce en el puerto. En cambio, **OO** es el punto por el que se emitirán todas las salidas. Obsérvese que sus tipos están invertidos con respecto al par anterior.

- ☐ **W** $\mapsto$ Se trata, nuevamente, de un par, en este caso de referencias a canal. Los nombres asociados se llaman nuevamente **WI** y **WO**. Sin embargo, en este caso ambos se orientan hacia la entrada. Como puede verse, todo dato recibido por **OI** entra en una célula de memoria, definida por el proceso CellPort. A partir de ahí, se solicita su recepción, a través de un canal u otro. Si se hace desde **WI**, el dato se emite desde la célula, pero aún permanece en la misma. Si se hace desde **WO**, el dato es borrado tras la emisión. En definitiva, y como se verá en la sección 7.4.4, el primero realiza una lectura-**when**, mientras que el segundo hace una lectura convencional.

- ☐ **M** $\mapsto$ En este caso, el par definido distingue de nuevo entre entrada y salida, tomando un canal en el primer caso y una referencia a canal en el segundo. En el caso de **MI**, se

recibe una entrada que puede afectar a la célula de memoria mencionada más arriba, y que contiene el último dado de entrada recibido; por tanto, este nombre se utiliza para implementar un **shift**. Análogamente, el **MO** se utiliza para implementar un **catch**: recibe un nombre de canal, por el cual se *desvía* una salida emitida por este puerto.

☐ **D** ↦ Se trata de una señal –una sincronización pura–, cuya recepción provoca la destrucción de toda la estructura; es el nombre designado como *kill* en la definición de un puerto. No sólo provoca la finalización del proceso RepSelf, sino también la de todos los procesos. Todas las estructuras tendrán este campo.

☐ **B** ↦ Da acceso a una lista de nombres de componente. Concretamente, esta lista será la de las *instancias conectadas a este puerto* a través de alguna conexión. Esta meta-información se corresponde con la solicitada por el operador **bound**, y su actualización es garantizada por las conexiones (§7.3.5).

☐ **S** ↦ Se trata de una emisión recursiva del nombre del puerto actual. Es, en definitiva, el canal por el que se hacen peticiones al proceso RepSelf ya mencionado (§7.2.4). Al igual que en el **D**, todas las estructuras tendrán este campo.

Como puede verse, la estructura tiene un tipo recursivo, ya que hace referencia a sí mismo en el proceso RepSelf. Dado que todas las estructuras usan este proceso, todas ellas se definirán igualmente con un tipo recursivo. Además, el campo **B** hace referencia al tipo de un componente (*TComp*), que a su vez contendrá referencias a puertos. Por tanto, ambos tipos serán mutuamente recursivos.

Por supuesto, esta funcionalidad no se debe simplemente al tipo, sino que se implementa como un conjunto de procesos, que se describen a continuación, y que sólo serán descritos de manera superficial:

$$
\begin{array}{rcl}
\text{Port} & : & (\sharp\alpha,\, \sharp\beta;\, \uparrow(\mathit{TPort}\,\alpha\,\beta)) \\
\text{Port}(p) & \triangleq & (\nu\, ii, io, oi, oo, wi, wo, sh, ct, bp, kill, me, k_1, k_2) \\
& & (\, \overline{bp}\langle\, (\texttt{nil})\,\rangle.\,\mathbf{0} \mid kill.\,\overline{k_1}.\,\overline{k_2}.\,\mathbf{0} \\
& & \mid me(p_1).\,\overline{p}\langle\, p_1\,\rangle.\,\mathbf{0} \\
& & \mid \text{RepSelf}\langle\, me, (\dagger I \prec [ii, io], \dagger O \prec [oi, oo], \\
& & \qquad \dagger W \prec [wi, wo], \dagger M \prec [sh, ct], \\
& & \qquad \dagger D \prec kill, \dagger B \prec bp, \\
& & \qquad \dagger S \prec me\ ), k_1\rangle \\
& & \mid \text{BehPort}\langle\, [ii, io], [oi, oo], [wi, wo], [sh, ct], k_2\,\rangle \\
& &)
\end{array}
$$

El proceso Port es, obviamente, el proceso que define la construcción de una estructura de puerto. Básicamente crea todos los canales, inicializa la lista, y construye el registro, con el que alimenta al proceso RepSelf. El comportamiento de todos los elementos se despliega a partir del proceso BehPort, que se describe a continuación, con todos sus subprocesos:

$$
\begin{array}{rcl}
\text{BehPort} & : & (\sharp\alpha,\, \sharp\beta;\, (Par\,\alpha\,\beta),\, (Par\,\beta\,\alpha),\, (Par\ \uparrow\alpha\ \uparrow\alpha), \\
 & & \quad (Par\,\alpha\ \uparrow\beta),\, Sync) \\
\text{BehPort}(\widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, die) & \triangleq & (\nu\, d_1, d_2)(\ die.\, \overline{d_1}.\, \overline{d_2}.\, \mathbf{0}\ | \\
 & & \text{OutPort}\langle \widehat{\imath}, \widehat{o}, \widehat{m}, d_2 \rangle \mid \text{InPort}\langle \widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, d_1 \rangle\ ) \\
\text{OutPort} & : & (\sharp\alpha,\, \sharp\beta;\, (Par\,\alpha\,\beta),\, (Par\,\beta\,\alpha), \\
 & & \quad (Par\,\alpha\ \uparrow\beta),\, Sync) \\
\text{OutPort}(\widehat{\imath}, \widehat{o}, \widehat{m}, die) & \triangleq & i_o(x).\, \overline{o_o}\langle\, x\,\rangle.\, \text{OutPort}\langle \widehat{\imath}, \widehat{o}, \widehat{m}, die \rangle \\
 & & +\ m_o(r).\, i_o(x).\, \overline{r}\langle\, x\,\rangle.\, \text{OutPort}\langle \widehat{\imath}, \widehat{o}, \widehat{m}, die \rangle \\
 & & +\ die.\, \mathbf{0} \\
\text{InPort} & : & (\sharp\alpha,\, \sharp\beta;\, (Par\,\alpha\,\beta),\, (Par\,\beta\,\alpha),\, (Par\ \uparrow\alpha\ \uparrow\alpha), \\
 & & \quad (Par\,\alpha\ \uparrow\beta),\, Sync) \\
\text{InPort}(\widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, die) & \triangleq & o_i(x).\, \text{CellPort}\langle \widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, die, x \rangle \\
 & & +\ m_i(z).\, \text{CellPort}\langle \widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, die, z \rangle \\
 & & +\ die.\, \mathbf{0} \\
\text{CellPort} & : & (\sharp\alpha,\, \sharp\beta;\, (Par\,\alpha\,\beta),\, (Par\,\beta\,\alpha),\, (Par\ \uparrow\alpha\ \uparrow\alpha), \\
 & & \quad (Par\,\alpha\ \uparrow\beta),\, Sync,\, \alpha) \\
\text{CellPort}(\widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, die, x) & \triangleq & w_i(r).\, \overline{r}\langle\, x\,\rangle.\, \text{CellPort}\langle \widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, die, x \rangle \\
 & & +\ w_o(r).\, \overline{r}\langle\, x\,\rangle.\, \text{InPort}\langle \widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, die \rangle \\
 & & +\ m_i(z).\, \text{CellPort}\langle \widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, die, z \rangle \\
 & & +\ die.\, \mathbf{0}
\end{array}
$$

El comportamiento definido por BehPort es muy sencillo. Su tarea es, básicamente, repartir a los canales entre un proceso de salida (OutPort) y otro de entrada (InPort). El primero de ellos simplemente transporta un dato desde la salida interior **IO** hasta la exterior **OO**, o bien permite que éste sea capturado. La estructura del segundo es más compleja, ya que se remite a una célula de memoria, definida por CellPort. Esta célula podrá ser alimentada por datos procedentes de las dos vías ya indicadas, **OI** y **MI**. Como también se ha dicho, la lectura de los datos de esta célula podrá ser realizada a través de dos canales diferentes, siendo en uno de los casos destructiva (**WO**) y en el otro, no destructiva (**WI**).

7.3.2 Componentes (Instancias)

El elemento más importante –y la estructura más grande, aunque no la más compleja– del lenguaje es, como es sabido, el componente; en este caso, cada una de sus instancias. Su estructura se describe en el dibujo de la Figura 7.6.

El tipo que define a la estructura de un puerto es mucho más complejo, y se define a continuación. Obsérvese que los puertos se definen haciendo uso del tipo dinámico Dyn, tal como se describe en la sección C.5 del Apéndice correspondiente. El motivo es que, de otro modo, el

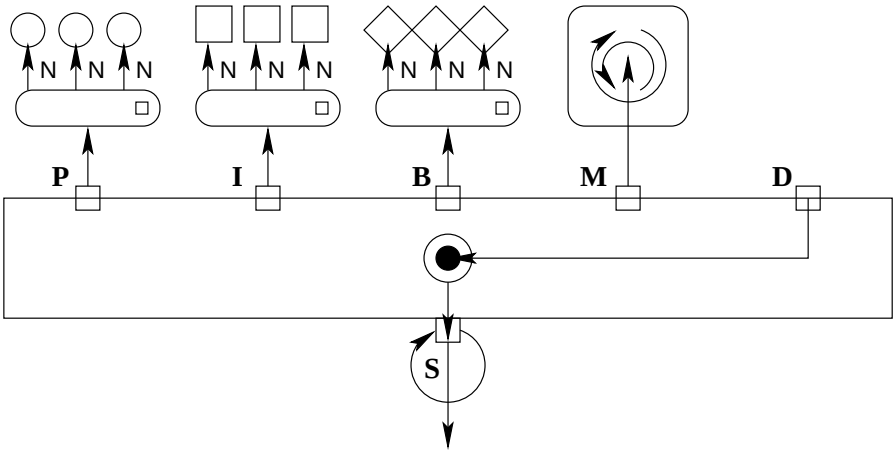

Figura 7.6: Estructura Interna de un Componente $\mathcal{P}i\mathcal{L}ar$

componente tendría que estar parametrizado según todos los tipos de cada uno de sus puertos, cuyo tipo es variable; por tanto, no se podría hacer una definición genérica de la estructura. Este tipo de artificio es habitual en circunstancias similares: el tipo, por ejemplo, fue definido en Nomadic Pict [WS00] en una situación análoga. En cualquier caso, la estructura se expresa, de nuevo, como un registro de seis campos:

$$
\begin{array}{rcl}
DPort & \triangleq & (\,TPort\,\mathrm{Dyn}\,\mathrm{Dyn}) \\
Tier & \triangleq & (\,\dagger C\!:\!Label,\ \dagger P\!:\!Label) \\
\\
TComp & \triangleq & \mu C.(\,\dagger P\!:\!\uparrow\!(NList\ \uparrow\!DPort), \\
& & \quad \dagger I\!:\!\uparrow\!(NList\ \uparrow\!C), \\
& & \quad \dagger B\!:\!\uparrow\!(NList\ \uparrow\!TBind), \\
& & \quad \dagger M\!:\!\uparrow\!\mathcal{M}Comp, \\
& & \quad \dagger D\!:\ Sync, \\
& & \quad \dagger S\!:\!\uparrow\!C\,)
\end{array}
$$

Como ya se indicó antes, este tipo no sólo es recursivo, sino que es mutuamente recursivo con el de un puerto. Esto no plantea, por supuesto, ningún problema desde un punto de vista teórico, aunque hace compleja su traducción a una implementación real del cálculo.

El significado intuitivo de cada una de los campos del registro es el que se indica a continuación. Obsérvese que en este caso se usan con frecuencia estructuras de *listas nombradas*, que como se comenta en su definición (§C.4.4) son el equivalente dinámico de un registro.

- □ **P** $\mapsto$ Se trata de un canal hacia una lista nombrada de puertos, es decir, la lista que define la interfaz del componente.

☐ **I** $\mapsto$ También es un canal hacia una lista nombrada, pero en este caso es de instancias: es decir, se trata de la lista de sub-instancias contenidas por este componente –cuando es compuesto–. Por supuesto, esto define al tipo como recursivo.

☐ **B** $\mapsto$ De nuevo es un canal hacia una lista nombrada, en este caso de conexiones. Es decir, se trata de la lista de conexiones entre las sub-instancias contenidas en este componente, cuando es compuesto.

☐ **M** $\mapsto$ Cuando la instancia sea un metacomponente, este canal apunta hacia una estructura de tipo $\mathcal{M}Comp$, que contiene la representación del arquetipo a través del cual se realizan reificaciones (7.2.5). Es decir, este elemento apunta a su *brhátar*, si éste existe. Los detalles sobre este particular se indicarán con mayor detenimiento en la sección 7.3.3.

☐ **D** $\mapsto$ Se trata, nuevamente, de la señal de terminación.

☐ **S** $\mapsto$ Se trata, nuevamente, del canal recursivo, asociado a RepSelf, que vincula esta estructura a su nombre.

La estructura del proceso que crea un componente (Comp) es mucho más compleja que en el caso anterior, aunque resulta bastante lógica. Se debe tener presente en todo momento la diferencia entre el contenido de las listas nombradas y sus etiquetas. Cada etiqueta describe el nombre de un elemento, tal como se le ve desde la perspectiva de la sintaxis: es decir, es el identificador que el arquitecto utiliza para referenciarlo. En cambio, el contenido es un *nombre* del cálculo, cuya unicidad está garantizada, y que permite distinguir a un puerto, instancia o conexión de cualquier otro, de manera análoga a como ocurría en Marmol.

$$
\begin{aligned}
\mathrm{Comp} \quad &: \quad (\sim Label,\ \sim Label,\ \sim\!\uparrow\!\mathcal{M}Comp,\ \sim Label,\ \sim\!\uparrow\!\mathcal{M}Comp,\\
&\qquad \approx Tier,\ \uparrow\!\mathcal{M}Comp,\ \uparrow\!TComp)\\
\mathrm{Comp}(\widetilde{p}, \widetilde{s}, \widetilde{T}, \widetilde{b}, \widetilde{L}, \widetilde{\widetilde{E}}, m, c) \quad &\triangleq \quad (\nu\, lp, ls, lb, kill, me, k_1)\\
&\quad ((\ (\nu\, v_p, e_p)(\ \overline{lp}\langle\, (\mathtt{nil})\,\rangle.\, \mathrm{NFor}\langle\, |\widetilde{p}|, v_p, e_p\,\rangle\\
&\qquad\quad |\ \mathrm{MkPort}_F\langle\, \widetilde{p}, v_p, e_p, lp\,\rangle\ )\\
&\quad |\ (\nu\, v_s, e_s)(\ \overline{ls}\langle\, (\mathtt{nil})\,\rangle.\, \mathrm{NFor}\langle\, |\widetilde{s}|, v_s, e_s\,\rangle\\
&\qquad\quad |\ \mathrm{MkInst}_F\langle\, \widetilde{s}, \widetilde{T}, v_s, e_s, ls\,\rangle\ ));\\
&\quad (\ me(c_1).\, \overline{c}\langle\, c_1\,\rangle.\, \mathbf{0}\\
&\quad |\ \ \mathrm{RepSelf}\langle\, me, (\dagger P \prec lp,\\
&\qquad\quad \dagger I \prec ls, \dagger B \prec lb,\\
&\qquad\quad \dagger M \prec m,\\
&\qquad\quad \dagger D \prec kill,\\
&\qquad\quad \dagger S \prec me\ ), k_1\rangle\\
&\quad |\ (\nu\, v_b, e_b)(\ \overline{lb}\langle\, (\mathtt{nil})\,\rangle.\, \mathrm{NFor}\langle\, |\widetilde{b}|, v_b, e_b\,\rangle\\
&\qquad\quad |\ \mathrm{MkBind}_F\langle\, me, \widetilde{b}, \widetilde{L}, \widetilde{\widetilde{E}}, v_b, e_b, lb\,\rangle\ )
\end{aligned}
$$

$$
\begin{array}{rcl}
 & & \mid \mathit{kill}.\,(\,\mathrm{SubKill}\langle\, lp, ls, lb \,\rangle \mid \overline{k_1}.\,\mathbf{0}\,) \\
 & &))
\end{array}
$$

En este caso, el constructor de componente recibe muchos más parámetros. Se asume que este proceso es invocado directamente a partir de la traducción de una descripción del Lenguaje Estructural, por lo que todos estos parámetros se reciben en la forma de vectores. Los parámetros son, respectivamente, la secuencia de etiquetas de puerto; la de etiquetas de instancias, y sus arquetipos; la de etiquetas de conexión, junto a sus arquetipos y un esquema de conexión E, indicado como un vector bidimensional de pares de etiquetas *Tier*; finalmente, la referencia al brhátar, si existe –en caso contrario, aquí se le proporciona la referencia a su propio rohatar, para producir un nulo–, y un canal de retorno.

La creación del componente invoca la creación de cada uno de sus elementos; una vez todos han sido creados, se construye la estructura, y se liga en una recursión RepSelf. Los distintos elementos se construyen a partir de un bucle NFor sobre cada uno de los vectores proporcionados, según el esquema descrito en la sección D.1 del Apéndice correspondiente. Por tanto, y tal como allí se explica, cada *invocación* de un proceso marcado con el subíndice $(_F)$ se refiere a *uno* de los elementos de dicho vector, esto es, expresa *un ciclo* del bucle.

$$
\begin{array}{rcl}
\mathrm{MkPort_F} & : & (\sim \mathit{Label},\ \uparrow \mathit{Int},\ \mathit{Sync},\ (\mathit{NList\ DPort})) \\
\mathrm{MkPort_F}(\widetilde{p}, vcb, end, lp) & \triangleq & (\nu\, np)(\ vcb(ix).\, lp(lis).\,(\ \mathrm{Port}\langle\, np \,\rangle \mid np(m_p). \\
 & & \quad \overline{lp}\langle\, (\mathsf{ncons}\ p_{ix}\ m_p{\dagger}S\ lis)\,\rangle.\,\mathrm{MkPort}_F\langle\, \widetilde{p}, vcb, end, lp \,\rangle\ ) \\
 & & +\ end.\,\mathbf{0}\ ) \\
\mathrm{MkInst_F} & : & (\sim \mathit{Label},\ \sim\!\uparrow \mathcal{M}\mathit{Comp},\ \uparrow \mathit{Int},\ \mathit{Sync},\ (\mathit{NList\ TComp})) \\
\mathrm{MkInst_F}(\widetilde{s}, \widetilde{T}, vcb, end, ls) & \triangleq & (\nu\, nc)(\ vcb(ix).\, ls(lis).\,(\ \mathrm{NewComp}\langle\, T_{ix}, nc \,\rangle \mid nc(m_c). \\
 & & \quad \overline{ls}\langle\, (\mathsf{ncons}\ s_{ix}\ m_c{\dagger}S\ lis)\,\rangle.\,\mathrm{MkInst}_F\langle\, \widetilde{s}, \widetilde{T}, vcb, end, ls \,\rangle\ ) \\
 & & +\ end.\,\mathbf{0}\ ) \\
\mathrm{MkBind_F} & : & (\uparrow \mathit{TComp},\ \sim \mathit{Label},\ \sim\!\uparrow \mathcal{M}\mathit{Comp},\ \approx \mathit{Tier},\ \uparrow \mathit{Int}, \\
 & & \quad \mathit{Sync},\ (\mathit{NList\ TBind})) \\
\mathrm{MkBind_F}(c, \widetilde{b}, \widetilde{L}, \widetilde{\widetilde{E}}, vcb, end, lb) & \triangleq & (\ vcb(ix).\, lb(lis).\,(\nu\, nb)(\\
 & & \quad \mathrm{OneBind}\langle\, c, L_{ix}, E_{ix}, nb \,\rangle \\
 & & \quad \mid\ nb(n_1).\,\overline{lb}\langle\, (\mathsf{ncons}\ b_{ix}\ (n_1{\dagger}S)\ lis)\,\rangle. \\
 & & \qquad \mathrm{MkBind}_F\langle\, c, \widetilde{b}, \widetilde{L}, \widetilde{\widetilde{E}}, vcb, end, lb \,\rangle\ ) \\
 & & +\ end.\,\mathbf{0}\ ) \\
\mathrm{OneBind} & : & (\uparrow \mathit{TComp},\ \uparrow \mathcal{M}\mathit{Comp},\ \sim \mathit{Tier},\ \uparrow \mathit{TBind}) \\
\mathrm{OneBind}(c, L, \widetilde{E}, b) & \triangleq & (\nu\, v_1, e_1, r_b)(\ \mathrm{NFor}\langle\, |\widetilde{E}|, v_1, e_1 \,\rangle \\
 & & \quad \mid\ \mathrm{UnTier}_F\langle\, c, \widetilde{E}, (\mathtt{nil}), (\mathtt{nil}), v_1, e_1, r_b \,\rangle \\
 & & \qquad \mid\ r_b(ls, lp).\,\mathrm{NewBind}\langle\, L, ls, lp, b \,\rangle \\
 & & \ )
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{UnTier} & : & (\uparrow \mathit{TComp},\ \sim \mathit{Tier},\ (\mathit{List}\ \uparrow \mathit{TComp}),\ (\mathit{List}\ \uparrow \mathit{DPort}), \\
 & & \quad \uparrow \mathit{Int},\ \mathit{Sync}) \\
\mathrm{UnTier_F}(c, \widetilde{E}, ls, lp, vcb, end, r) & \triangleq & (\ vcb(ix).\, c(c_1). \\
 & & (\nu\, s)(\ \overline{s}\langle\, c_1 \bullet (E_{ix}\dagger C)\,\rangle \mid s(s_1). \\
 & & \quad (\nu\, p)(\ \overline{p}\langle\, s_1 \bullet (E_{ix}\dagger P)\,\rangle \mid p(p_1). \\
 & & \qquad \mathrm{UnTier}_F\langle\, c, \widetilde{E}, (\mathsf{cons}\ (s_1\dagger S)\ ls), \\
 & & \qquad\quad (\mathsf{cons}\ (p_1\dagger S)\ lp), vcb, end, r\,\rangle\ \ )) \\
 & & +\ end.\, \overline{r}\langle\, ls, lp\,\rangle.\, \mathbf{0}\ ) \\
\mathrm{SubKill} & : & ((\mathit{NList\ DPort}),\ (\mathit{NList\ TComp}),\ (\mathit{NList\ TBind})) \\
\mathrm{SubKill}(lp, ls, lb) & \triangleq & (\ ((\nu\, v_b, e_b)(\ lb(lis).\, \mathrm{NLFor}\langle\, lis, v_b, e_b\,\rangle \\
 & & \qquad \mid \mathrm{Die}_F\langle\, v_b, e_b\,\rangle\ ));\\
 & & (\ (\nu\, v_n, e_n)(\ lp(lis).\, \mathrm{NLFor}\langle\, lis, v_n, e_n\,\rangle
\end{array}
$$

BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

$$
\begin{array}{rcl}
 & & \mathit{Comp},\ \mathit{TBind}) \\
] & & k.\, \mathrm{Die}\langle\, vcb, end\,\rangle + end.\, \mathbf{0}\ )
\end{array}
$$

Se definen cuatro tipos de procesos, así como los subprocesos necesarios. El primero es MkPort_F, que crea un puerto –utilizando el proceso Port ya conocido– por cada etiqueta del vector de puertos, y acumula ambos en la lista nombrada designada por †**P**. El segundo es MkInst_F, que crea una sub-instancia por cada etiqueta del vector correspondiente, utilizando el tipo correspondiente, y acumula el resultado en la lista nombrada de †**I**. Para crear el componente, se remite en última instancia al proceso NewComp, que será definido en la página 250, y que se basa internamente en el propio proceso Comp. El tercero es MkBind_F, y es el más complejo: crea una conexión por cada una de las etiquetas del vector, según el tipo y el esquema de conexión proporcionado, y se acumula el resultado en la lista nombrada de †**B**. Para crear la conexión, se utiliza en última instancia el subproceso NewBind, que será definido en la página 259. El efecto de sus subprocesos (OneBind y UnTier) es básicamente el de transformar los parámetros, de su forma de vectores a la de listas.

Resulta interesante señalar el modo en que se recorre un esquema de conexión, ya que se utilizará en varias ocasiones. En resumen: dados los tres vectores iniciales (b, L y E), a cada etiqueta de b le corresponde un tipo de L, y un elemento de E; ya que E es un vector bidimensional, este elemento es un vector. Este vector contiene una secuencia de elementos *Tier*, que representa a *todos los puertos ligados por la conexión* que ha de crearse. Cada puerto es representado, en un tipo *Tier*, como un par de etiquetas, que simbolizan respectivamente a la instancia que lo contiene (†C) y el propio puerto en ella (†P). De este modo, una vez se han creado las instancias necesarias, pueden seleccionarse en ellas los puertos, para luego enlazarlos.

Finalmente, el cuarto proceso, SubKill, recorre con un bucle de listas nombradas NLFor (§D.1) cada una de las listas de puertos, instancias y conexiones, y emite en cada uno de sus elementos una señal de terminación (†**D**) a través del subproceso Die_F.

Obsérvese que todas las listas definidas –nombradas o no– son en realidad un *canal* hacia una lista. Como se verá a menudo en los procesos subsiguientes, una lectura sobre este canal *consume* a la lista, por lo que es preciso definir un protocolo explícito de recepción-envío: todo aquél que la lea tendrá luego que volver a emitirla. Esto tiene dos ventajas: permite la modificación de la lista, y evita los problemas derivados de un acceso concurrente, ya que de este modo se garantiza la exclusión mutua: dos procesos no pueden acceder a la lista simultáneamente.

Por supuesto, el proceso SubKill no restaura las listas una vez que las lee, ya que su propósito es destruir la estructura.

7.3.3 Reificación

La estructura que representa a una reificación es la más simple de todas las utilizadas en el Lenguaje Estructural. Al igual que en Marmol, una reificación se describe simplemente como un enlace entre un avatar y un rohatar. Sin embargo, dado que la referencia a este enlace será contenida –como ya es sabido– en uno de estos extremos, concretamente dentro de la estructura de un metacomponente, bastará con reducir su definición al tipo siguiente:

$$(\mathit{TReif}\ \alpha) \quad \triangleq \quad (\dagger A{:}\uparrow\alpha,\ \dagger D{:}\ \mathit{Sync})$$

La reificación está parametrizada para distinguir entre tipos de componente y de conexión, cuando sea preciso. En la práctica, se utilizará siempre sobre tipos *TComp*, ya que la estructura de las conexiones puede verse como un fragmento suyo, y a menudo se utilizará el tipo *TComp* para hacer referencia a ambos. En definitiva, se puede considerar que una reificación será siempre una estructura de tipo (*TReif TComp*), que consta simplemente del enlace a un avatar y una señal de terminación. Esta última es, como se verá más adelante, necesaria para desarrollar todo el proceso de destrucción, en caso de borrado de la reificación.

Debe tenerse en cuenta que un metacomponente sólo puede reificar *una vez* a un mismo componente (instancia) y al mismo tiempo. De otro modo, se encontraría en la situación absurda de entrar en conflicto consigo mismo, intentando duplicar la misma estructura, y describir dos veces el mismo comportamiento. Por tanto, entre un metacomponente y un avatar concreto sólo puede existir *una* reificación.

Como cualquier otra entidad del modelo, una reificación debe poder ser referenciada de manera unívoca, ya que podrá ser manipulada e incluso destruida en un momento dado (§7.4.6). Sin embargo, y a diferencia del resto, no necesita tener un nombre propio que la designe de manera explícita. Como se acaba de ver, un enlace de reificación supone una interrelación única, lo que significa que queda identificada por los identificadores –los nombres– de sus extremos. Es decir, basta con conocer en cada caso el nombre del metacomponente y el del avatar, para caracterizar unívocamente a la reificación implicada.

El nombre del avatar está incluido como parte de la propia estructura de reificación. A su vez, como se verá a continuación (§7.3.4), ésta estará contenida en la estructura de un metacomponente, que también incluye su propio nombre. Por tanto, los dos nombres necesarios estarán disponibles en todo momento, y por tanto la reificación queda siempre identificada. Así pues, no es preciso definir un concepto de nombre para esta categoría semántica[9].

[9] Sí se hacía, sin embargo, en la sintaxis; pero no es necesario traducirlo, por las razones expuestas.

En resumen, la reificación es, paradójicamente, la única entidad del modelo que *no* tiene una estructura autorreferencial, dado que ni siquiera necesita un nombre propio.

La reificación, en conjunción con la estructura del metacomponente, que viene dada por el tipo $\mathcal{M}Comp$ y se describe en el apartado siguiente, define el modo en que se realiza el enlace entre el nivel base y el nivel meta. Recapitulando, puede recordarse que la estructura descrita en la sección 7.2.5, que fue resumida en la Figura 7.1, adoptaba la forma de un esquema cuádruple compuesto por un avatar, un metacomponente, el brhátar correspondiente a éste, el enlace de reificación, y el proceso compartido por ellos. En este punto, y a la luz de lo dicho en este apartado, se puede reconstruir dicha Figura utilizando términos concretos de la traducción, teniendo pues en cuenta los tipos y nombres de enlace ya definidos, o que se definen a continuación. El resultado se expresa en la Figura 7.7.

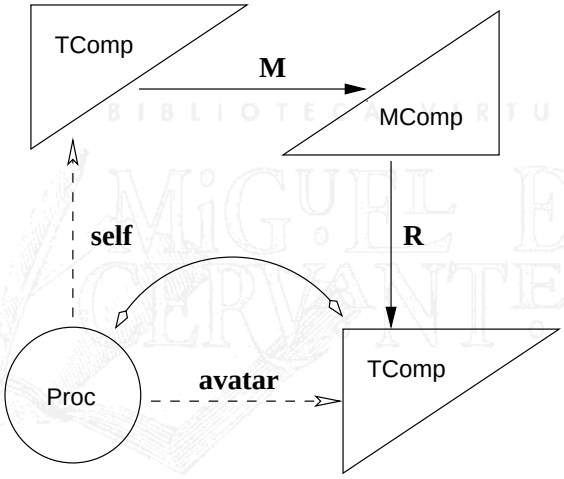

Figura 7.7: Dualidad de un Componente Reflexivo

De este modo, el campo †**M** de un tipo *TComp* señala al brhátar de un componente del nivel meta, mientras que el campo †**R** de un $\mathcal{M}Comp$ contendrá, como se verá a continuación, una listas de todas las reificaciones de un rohatar.

Conviene señalar que ya se está en condiciones de averiguar si un componente dado tiene o no un brhátar, esto es, si actúa como un metacomponente. Como se indicó en su momento, si esta estructura no existe, en el proceso Comp se proporciona el nombre de un rohatar del componente, de modo que se pueda utilizar como un valor nulo. Por tanto:

$$
\begin{array}{rcl}
\text{NoBhratar} & : & (\mathit{TComp}, \uparrow \mathit{Bool}) \\
\text{NoBhratar}(ava, b) & \triangleq & (\nu\, r)(\; ava\dagger M(m).\, m\dagger R(lr).\, \text{AvIn}\langle\, ava, lr, r\,\rangle \\
& & \quad \mid\; r(ki).\, \overline{m\dagger R}\langle\, lr\,\rangle.\, \textbf{if}\;\; (ki = ava\dagger D)\;\; \textbf{then}\;\; \overline{b}\langle\, true\,\rangle.\, \mathbf{0} \\
& & \quad \textbf{else}\;\; \overline{b}\langle\, false\,\rangle.\, \mathbf{0}
\end{array}
$$

La descripción es evidente: se comprueba si el componente indicado por †**M** tiene como avatar al componente actual; si no es así, entonces no es uno de sus metacomponentes, y por

tanto es su brhátar. La definición utiliza al subproceso AvIn, que se define a su vez en la sección 7.1, concretamente en la página 275. Su única función es determinar si un avatar dado es mencionado entre las reificaciones de un metacomponente.

En cualquier caso, éste es un detalle menor, salvo en el hecho de que permite averiguar si un componente dado es o no un metacomponente activo. Por lo demás, su mayor interés está, precisamente, en la concreción de la estructura descrita en la Figura.

7.3.4 Componentes en el Nivel Meta

Como ya se ha explicado en la sección 7.2.5, y como acaba de reiterarse en el apartado anterior, a todo arquetipo le corresponde una estructura interna, que expresa su influencia sobre las instancias. Esta estructura es compleja, y se muestra de manera gráfica en la Figura 7.8.

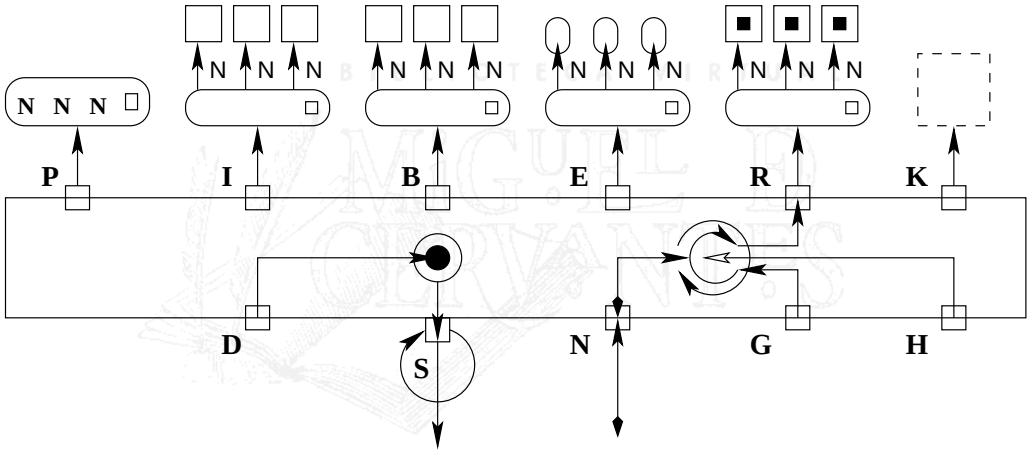

Figura 7.8: Estructura de un Meta-Componente

De modo análogo a casos anteriores, el tipo correspondiente se define como:

$$
\begin{aligned}
Tower &\triangleq (List\ TComp) \\
Proc &\triangleq (Tower,\ \uparrow Sync)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{M}Comp \triangleq\ & \mu M.(\dagger P{:}\uparrow(List\ Label), \\
& \quad \dagger I{:}\uparrow(NList\ \uparrow M), \\
& \quad \dagger B{:}\uparrow(NList\ \uparrow M), \\
& \quad \dagger E{:}\uparrow(NList\,(List\ Tier)), \\
& \quad \dagger R{:}\uparrow(List\,(TReif\ TComp)), \\
& \quad \dagger N{:}\uparrow TComp,\ \dagger K{:}\uparrow TComp, \\
& \quad \dagger G{:}\uparrow Proc,\ \dagger H{:}\ Int,
\end{aligned}
$$

$$\dagger D\colon\ Sync,$$
$$\dagger S\colon \uparrow M\ )$$

Se proporciona, entonces, la descripción intuitiva de cada uno de los campos del registro definido, que constituye la estructura más grande de toda la semántica:

☐ **P** ↦ Lista de etiquetas de puertos: define cuál es la interfaz esperada en el arquetipo para cada uno de sus avatares. Obsérvese que esto excluye explícitamente a la metafaz (§6.4.1), que será la interfaz de la *instancia* asociada con éste arquetipo.

☐ **I** ↦ Lista nombrada de metacomponentes para sub-instancias: define el conjunto de subinstancias que se contienen en cada avatar de este arquetipo, indicando no sólo su etiqueta, sino también su tipo, expresado como una estructura $\mathcal{M}Comp$.

☐ **B** ↦ Lista nombrada de metacomponentes para sub-conexiones: define el conjunto de conexiones contenidas en cada avatar, indicando tanto su etiqueta como su tipo, de nuevo como una estructura $\mathcal{M}Comp$.

☐ **E** ↦ Lista nombrada de esquemas de conexión. Indica la conectividad de cada una de las conexiones incluidas en el esquema anterior, como una secuencia de puertos, expresados como pares *Tier*, esto es, como pares de etiquetas de instancia-puerto. El modo en que se manipula una lista de este tipo fue ya descrito en la sección 7.3.2, y su estructura es análoga a la de un esquema de conexión de Marmol (véase la definición 5.34).

☐ **R** ↦ Lista de reificaciones: contiene un enlace hacia cada uno de los avatares de este arquetipo, en una estructura de reificación.

☐ **K** ↦ Expresa una referencia desde una estructura $\mathcal{M}Comp$ hasta la instancia que conforma la parte activa de este metacomponente. Es decir, si el metacomponente es el brhátar de una instancia –lo que es cierto en todo arquetipo no anudado–, este campo permite localizar a dicha instancia. Es el inverso al †M de un *TComp*, y sólo será utilizado durante la generación de los procesos reificados.

☐ **N** ↦ Se trata de un canal generador de reificaciones. Se recibe a través de él una referencia hacia una instancia, y ésta *se reifica* con el arquetipo que mi estructura representa. Es decir todo componente enviado hacia este punto es reificado por este arquetipo.

☐ **G** ↦ Vinculado a **N**. Proporciona una referencia hacia la definición de proceso g asociada con este arquetipo, de modo que este proceso podrá ser duplicado para toda instancia a reificar, según la estructura descrita en la sección 7.2.5.

☐ **H** ↦ Vinculado a **N**. Expresa el grado de reflexión $\eth$ asociado con el proceso referenciado por **G**, indicando por tanto cuántos niveles han de descenderse en la red de reificaciones, para crear los duplicados correspondientes, según la estructura descrita en la sección 7.2.5.

☐ **D** ↦ Se trata, nuevamente, de la señal de terminación.

☐ **S** ↦ Se trata, nuevamente, del canal recursivo, asociado a RepSelf, que vincula esta estructura a su nombre.

Al igual que las otras estructuras, ésta es obviamente recursiva, y de nuevo su tipo es mutuamente recursivo con el tipo *TComp*, ya que cada uno de ellos menciona al otro.

La complejidad de las relaciones entre los tipos $\mathcal{M}Comp$ y *TComp*, así como del tipo de proceso Proc asociado con una regla, y todos los procesos utilizados para su creación, es muy grande. Con el fin de clarificar las distintas dependencias, teniendo en cuenta quién invoca a quién, se ha elaborado el grafo de la Figura 7.9.

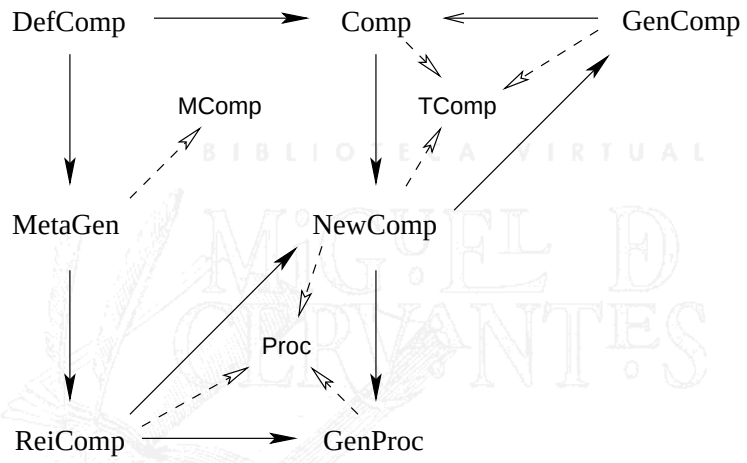

Figura 7.9: Dependencias entre Procesos sobre Componentes

En este esquema, se comienza a partir de una definición de componente en la sintaxis de $\mathcal{P}i\mathcal{L}ar$. A partir de ésta se define la invocación de dos procesos: el proceso Comp, que creará a cada una de sus instancias, y el proceso MetaGen, que crea a la estructura de su arquetipo. El primero (Comp) crea simplemente una estructura de componente *TComp*, con todos sus elementos. Ahora bien, algunos de estos elementos serán también componentes, cuyo tipo es conocido: en tal caso, se construyen mediante el proceso NewComp, que utiliza a su vez al proceso GenComp, para generar la estructura *TComp* correspondiente, y el proceso GenProc, que crea una copia del proceso adecuado. Finalmente, cada vez que se envía una instancia a través del puerto †**N** de una estructura $\mathcal{M}Comp$, ésta es reificada por ReiComp, que crea igualmente el proceso vinculado –usando también a GenProc– y a todas las estructuras subyacentes –usando también a NewComp para las subinstancias–.

Pasemos, pues, a examinar cada uno de ellos muy brevemente, comenzando por el proceso que crea una estructura $\mathcal{M}Comp$, y que es el siguiente:

$$\begin{aligned}\text{MetaGen} \quad : \quad & (\sim Label,\ \sim Label,\ \sim\!\uparrow\mathcal{M}Comp,\ \sim Label,\ \sim\!\uparrow\mathcal{M}Comp,\\ & \approx Tier,\ \uparrow TComp,\ \uparrow Proc,\ Int,\ \uparrow\mathcal{M}Comp)\end{aligned}$$

$$
\begin{array}{rcl}
\mathrm{MetaGen}(\widetilde{p}, \widetilde{s}, \widetilde{T}, \widetilde{b}, \widetilde{L}, \widetilde{\widetilde{E}}, c, g, h, m) & \triangleq & (\nu\, lp, ls, lb, lle, lr, do, kill, me, k_1, k_2) \\
& & (\ (\ \mathrm{VecLis}\langle\, \widetilde{p}, lp\,\rangle \mid \mathrm{VecNLis}\langle\, \widetilde{s}, \widetilde{T}, ls\,\rangle \\
& & \quad \mid \mathrm{VecNLis}\langle\, \widetilde{b}, \widetilde{L}, lb\,\rangle\ );\\
& & (\ \mathrm{VecDual}\langle\, \widetilde{b}, \widetilde{\widetilde{E}}, lle\,\rangle \\
& & \quad \mid \overline{lr}\langle\, (\mathtt{nil})\,\rangle.\,\mathbf{0} \\
& & \quad \mid kill.\,\overline{k_1}.\,\overline{k_2}.\,\mathrm{KillRei}\langle\, me\,\rangle.\,\mathbf{0} \\
& & \quad \mid me(m_1).\,\overline{m}\langle\, m_1\,\rangle.\,\mathbf{0} \\
& & \quad \mid \mathrm{RepSelf}\langle\, me, (\dagger P \prec lp, \\
& & \qquad \dagger I \prec ls, \dagger B \prec lb, \\
& & \qquad \dagger E \prec lle, \dagger R \prec lr, \\
& & \qquad \dagger N \prec do, \dagger K \prec c, \\
& & \qquad \dagger G \prec g, \dagger H \prec h, \\
& & \qquad \dagger D \prec kill, \\
& & \qquad \dagger S \prec me), k_1\rangle \\
& & \quad \mid \mathrm{Reifier}\langle\, me, do, k_2\,\rangle\ ))
\end{array}
$$

La estructura del proceso MetaGen es por tanto muy sencilla: se traducen sus parámetros del formato de vectores –tal como se lo indica el proceso de traducción– al de listas, usando las técnicas descritas en la sección D.3.2 del Apéndice correspondiente, y una vez que han sido traducidas a ese formato, se crea el registro con la forma adecuada. Se vincula, además, el campo †**N** con el subproceso Reifier, que invoca a ReiComp bajo demanda.

$$
\begin{array}{rcl}
\mathrm{Reifier} & : & (\uparrow\mathcal{M}Comp,\ \uparrow TComp,\ Sync) \\
\mathrm{Reifier}(me, do, die) & \triangleq & (\ do(c).\,(\ \mathrm{ReiComp}\langle\, me, c\dagger S\,\rangle \\
& & \quad \mid \mathrm{Reifier}\langle\, me, do, die\,\rangle\ ) \\
& & +\ die.\,\mathbf{0}\ ) \\
\mathrm{KillRei} & : & (\uparrow\mathcal{M}Comp) \\
\mathrm{KillRei}(me) & \triangleq & me\ddagger R(lr).\,\mathrm{KReif}\langle\, lr\,\rangle \\
\mathrm{KReif} & : & (\alpha;\ (List\,(TReif\ \alpha))) \\
\mathrm{KReif}(lis) & \triangleq & \mathbf{if}\ (\mathtt{null}\ lis)\ \mathbf{then}\ \mathbf{0} \\
& & \quad \mathbf{else}\ (\nu\, r)(\ \overline{r}\langle\, (\mathtt{car}\ lis)\,\rangle \mid r(top).\,\overline{top\dagger D}. \\
& & \qquad \mathrm{KReif}\langle\, (\mathtt{cdr}\ lis)\,\rangle\ ) \\
\mathrm{VecDual} & : & (\sim Label,\ \approx Tier,\ \uparrow(NList\,(List\,Tier)) \\
\mathrm{VecDual}(\widetilde{l}, \widetilde{\widetilde{E}}, r) & \triangleq & (\nu\, v_1, e_1, \widetilde{vec}, r_1)(\ \mathrm{NFor}\langle\, |\widetilde{v}|, v_1, e_1\,\rangle \\
& & \quad \mid \mathrm{VDual}_F\langle\, \widetilde{\widetilde{E}}, \widetilde{vec}, v_1, e_1, r_1\,\rangle \mid r_1(\widetilde{vE}). \\
& & \qquad \mathrm{VecNLis}\langle\, \widetilde{l}, \widetilde{vE}, r\,\rangle\ )
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{VDual_F} & : & (\approx Tier,\ \sim(List\,Tier),\ \uparrow Int,\ Sync, \\
 & & \quad \uparrow \sim(List\,Tier)) \\
\mathrm{VDual_F}(\widetilde{\widetilde{E}}, \widetilde{vec}, vcb, end, r) & \triangleq & (\ vcb(ix).\,(\nu\, r_1)(\ \mathrm{VecLis}\langle\, E_{ix}, r_1\,\rangle \\
 & & \quad \mid\ r_1(vec_{ix}).\,\mathrm{VDual}_F\langle\, \widetilde{\widetilde{E}}, \widetilde{vec}, vcb, end, r\,\rangle\ ) \\
 & & +\ end.\,\overline{r}\langle\, \widetilde{vec}\,\rangle.\,\mathbf{0}\ )
\end{array}
$$

Los procesos restantes se encargan simplemente de completar la transformación de estructuras –VecDual– o de preparar el camino para la destrucción de una reificación –KillRei– en el momento adecuado, como respuesta a una señal recibida en †**D**.

Como ya se ha comentado, en el caso de tener que crear una sub-instancia, conocido su tipo –su arquetipo $\mathcal{M}Comp$–, se utilizará el proceso NewComp, que elabora la estructura $TComp$ correspondiente, y general el proceso adecuado. Se define como:

BIBLIOTECA VIRTUAL

$$
\begin{array}{rcl}
\mathrm{NewComp} & : & (\uparrow\mathcal{M}Comp,\ \uparrow TComp) \\
\mathrm{NewComp}(km, kc) & \triangleq & (\nu\, rc, rd)(\ km(mt).\,\mathrm{GenComp}\langle\, mt\dagger S, rc\,\rangle \\
 & & \quad \mid\ \mathrm{GenProc}\langle\, rc\ddagger S, mt\dagger S, rd\,\rangle \\
 & & \quad \mid\ rd(die).\,rc(nco).\,mt\dagger M(lr).\,\overline{kc}\langle\, nco\,\rangle. \\
 & & \qquad \mathrm{NewReif}\langle\, lr, nco, die, mt\dagger M\,\rangle\ )
\end{array}
$$

Como puede verse, apenas si es una llamada a los subprocesos GenComp y GenProc, que se describen algo más abajo. El subproceso NewReif tan sólo actualiza la lista de reificaciones del arquetipo $\mathcal{M}Comp$ (km) utilizado.

Por otra parte, el proceso ReiComp, cuya definición estaba ya pendiente, se encarga de *reificar* a un componente ya existente, conocido su tipo. Esto significa que se elabora el proceso de control y las subestructuras adecuadas. Se define, por tanto, como:

$$
\begin{array}{rcl}
\mathrm{ReiComp} & : & (\uparrow\mathcal{M}Comp,\ \uparrow TComp) \\
\mathrm{ReiComp}(km, kc) & \triangleq & kc(c).\,km(m). \\
 & & (((\ (\nu\, v_p, e_p)(\ c\dagger P(lc).\,m\dagger P(lm).\,\overline{m\dagger P}\langle\, lm\,\rangle. \\
 & & \quad \mathrm{LFor}\langle\, lm, v_p, e_p\,\rangle \mid \mathrm{AddPort}_F\langle\, lc, v_p, e_p, c\dagger P\,\rangle\ ) \\
 & & \mid\ (\nu\, v_s, e_s)(\ c\dagger I(lc).\,m\dagger I(lm).\,\overline{m\dagger I}\langle\, lm\,\rangle. \\
 & & \quad \mathrm{LFor}\langle\, lm, v_s, e_s\,\rangle \mid \mathrm{AddInst}_F\langle\, lc, v_s, e_s, c\dagger I\,\rangle\ )); \\
 & & (\ (\nu\, v_b, e_b)(\ c\dagger B(lc).\,m\dagger B(lb).\,\overline{m\dagger B}\langle\, lb\,\rangle. \\
 & & \quad m\dagger E(lle).\,\overline{m\dagger E}\langle\, lle\,\rangle.\,\mathrm{LFor}\langle\, lb, v_b, e_b\,\rangle \\
 & & \qquad \mid\ \mathrm{AddBind}_F\langle\, lc, lle, v_b, e_b, c\dagger B\,\rangle\ )); \\
 & & \quad (\ (\nu\, kd)(\ \mathrm{GenProc}\langle\, kc, km, kd\,\rangle \mid kd(die). \\
 & & \qquad m\dagger R(lr).\,\mathrm{NewReif}\langle\, lr, c, die, m\dagger R\,\rangle.\,\mathbf{0}\ ))
\end{array}
$$

Como puede verse, se toma la estructura definida en el arquetipo, y se revisa la lista de sus puertos, instancias y conexiones. Todos los elementos necesarios se crean, y se añaden –$\mathsf{AddPort}_F$, $\mathsf{AddInst}_F$, $\mathsf{AddBind}_F$– en las listas correspondientes de la estructura ($TComp$) de la instancia reificada. Finalmente, se crea el subproceso con su comportamiento, mediante el proceso GenProc. A continuación se desarrolla la estructura de todos estos procesos:

$$
\begin{array}{rcl}
\mathrm{NewReif} & : & (\alpha;\ (List\,(TReif\ \alpha)),\ \alpha,\ Sync,\ \uparrow(List\,(TReif\ \alpha))) \\
\mathrm{NewReif}(lis, ava, die, r) & \triangleq & \overline{r}\langle\,(\mathtt{cons}\ (\dagger A \prec ava\ \dagger D \prec die)\ lis)\,\rangle \\
\mathrm{AddPort_F} & : & ((NList\ \uparrow DPort),\ \uparrow Label,\ Sync,\ \uparrow(NList\ \uparrow DPort)) \\
\mathrm{AddPort_F}(lis, vcb, end, r) & \triangleq & (\nu\, np)(\ vcb(tag).\,(\ \mathrm{Port}\langle\, np\,\rangle \mid np(m_p). \\
 & & \quad \mathrm{AddPort}_F\langle\,(\mathtt{ncons}\ tag\ (m_p\dagger S)\ lis), vcb, end, r\,\rangle\ ) \\
 & & +\ end.\,\overline{r}\langle\, lis\,\rangle.\,\mathbf{0}\ ) \\
\mathrm{AddInst_F} & : & ((NList\ \uparrow TComp),\ \uparrow(Named\,\mathcal{M}Comp), \\
 & & \quad Sync,\ \uparrow(NList\ \uparrow TComp)) \\
\mathrm{AddInst_F}(lis, vcb, end, r) & \triangleq & (\nu\, nc)(\ vcb(sub).\,(\ \mathrm{NewComp}\langle\, sub\dagger d, nc\,\rangle \mid nc(m_c). \\
 & & \quad \mathrm{AddInst}_F\langle\,(\mathtt{ncons}\ sub\dagger l\ (m_c\dagger S)\ lis), vcb, end, r\,\rangle\ ) \\
 & & +\ end.\,\overline{r}\langle\, lis\,\rangle.\,\mathbf{0}\ ) \\
\mathrm{AddBind_F} & : & ((NList\ \uparrow TBind),\ (NList\,(List\,Tier)), \\
 & & \quad \uparrow(Named\,\mathcal{M}Comp),\ Sync,\ \uparrow(NList\ \uparrow TBind)) \\
\mathrm{AddBind_F}(lis, lle, vcb, end, r) & \triangleq & (\nu\, nb, rc, rp)(\ vcb(lk). \\
 & & (\ \mathrm{TwoTier}\langle\, lle \bullet (lk\dagger l), rc, rp\,\rangle \mid rc(lc).\, rp(lp). \\
 & & \quad (\ \mathrm{NewBind}\langle\, lk\dagger d, lc, lp, nb\,\rangle \mid nb(m_b). \\
 & & \qquad \mathrm{AddBind}_F\langle\,(\mathtt{ncons}\ (lk\dagger l)\ (m_b\dagger S)\ lis), \\
 & & \qquad\quad lle, vcb, end, r\rangle\ )) \\
 & & +\ end.\,\overline{r}\langle\, lis\,\rangle.\,\mathbf{0}\ ) \\
\mathrm{TwoTier} & : & ((List\,Tier),\ \uparrow(List\ \uparrow TComp),\ \uparrow(List\ \uparrow DPort)) \\
\mathrm{TwoTier}(lt, rc, rp) & \triangleq & (\nu\, vcb, end)(\ \mathrm{LFor}\langle\, lt, vcb, end\,\rangle \\
 & & \mid\ \mathrm{TwoT}_F\langle\,(\mathtt{nil}), (\mathtt{nil}), vcb, end, rc, rp\,\rangle\ ) \\
\mathrm{TwoT_F} & : & ((List\ \uparrow TComp),\ (List\ \uparrow DPort),\ \uparrow Tier,\ Sync, \\
 & & \quad \uparrow(List\ \uparrow TComp),\ \uparrow(List\ \uparrow DPort)) \\
\mathrm{TwoT_F}(lc, lp, vcb, end, rc, rp) & \triangleq & (\ vcb(tier). \\
 & & \quad \mathrm{TwoT}_F\langle\,(\mathtt{cons}\ (tier\dagger C)\ lc), (\mathtt{cons}\ (tier\dagger P)\ lp), \\
 & & \qquad vcb, end, rc, rp\rangle \\
 & & +\ end.\,\overline{rc}\langle\, lc\,\rangle.\,\overline{rp}\langle\, lp\,\rangle.\,\mathbf{0}\ )
\end{array}
$$

Obsérvese que, una vez creado el elemento necesario –mediante Port, NewComp o NewBind–, la única complejidad del consiste en incorporarlas a la estructura de lista correspondiente. Los procesos restantes, de nuevo –TwoTier– se encargan básicamente de recorrer la estructura de

un esquema de conexión, de forma análoga a la ya expuesta, pero en este caso sobre una lista nombrada, en lugar de sobre un vector bidimensional.

Casi para finalizar, queda proporcionar la estructura del proceso GenComp, que crea una estructura de componente *TComp* a partir de la estructura de su arquetipo –$\mathcal{M}Comp$–, pero no así el proceso asociado. En tal caso, se define como:

$$
\begin{array}{rcl}
\mathrm{GenComp} & : & (\uparrow\mathcal{M}Comp,\ \uparrow TComp) \\
\mathrm{GenComp}(mc, r) & \triangleq & (\nu\, r_1)(\ mc(m).\,\mathrm{LisVec}\langle\, m{\dagger}P, r_1\,\rangle \mid r_1(\widetilde{p}). \\
& & \quad (\nu\, r_2)(\ \mathrm{LLisVec}\langle\, m{\dagger}I, r_2\,\rangle \mid r_2(\widetilde{s}). \\
& & \quad (\nu\, r_3)(\ \mathrm{NLisVec}\langle\, m{\dagger}I, r_3\,\rangle \mid r_3(\widetilde{T}). \\
& & \quad\ (\ \mathrm{LLisVec}\langle\, m{\dagger}B, r_2\,\rangle \mid r_2(\widetilde{b}). \\
& & \quad (\nu\, r_4)(\ \mathrm{NLisVec}\langle\, m{\dagger}B, r_4\,\rangle \mid r_4(\widetilde{L}). \\
& & \quad (\nu\, r_5)(\ \mathrm{Dual}\langle\, m{\dagger}E, r_5\,\rangle \mid r_5(\widetilde{\widetilde{E}}). \\
& & \quad\ \mathrm{Comp}\langle\, \widetilde{p}, \widetilde{s}, \widetilde{T}, \widetilde{L}, \widetilde{\widetilde{E}}, (m{\dagger}S), r\,\rangle\ ) \\
& &)))))
\end{array}
$$

Como puede verse, su única labor es recoger la estructura básica de los campos de un $\mathcal{M}Comp$, y pasarla del formato de vector al formato de lista, utilizando los procesos definidos a tal efecto –LLisVec, NLisVec– en el Apéndice D. Una vez se ha adaptado el formato de los datos, se invoca simplemente al proceso Comp original, que es el que construye al componente. La definición se completa con el fragmento siguiente:

$$
\begin{array}{rcl}
\mathrm{Dual} & : & ((NList\,(List\,Tier)),\ \uparrow\approx Tier) \\
\mathrm{Dual}(llis, r) & \triangleq & (\nu\, r_1, \widetilde{\widetilde{E}})(\ \mathrm{NLisVec}\langle\, llis, r_1\,\rangle \mid r_1(\widetilde{lE}). \\
& & \quad (\nu\, v_1, e_1)(\ \mathrm{NFor}\langle\, |\widetilde{lE}|, v_1, e_1\,\rangle \\
& & \quad \mid\ \mathrm{Dual}_F\langle\, \widetilde{lE}, \widetilde{\widetilde{E}}, v_1, e_1, r\,\rangle\ )) \\
\mathrm{Dual_F} & : & (\sim(List\,Tier),\ \approx Tier,\ \uparrow Int,\ Sync,\ \uparrow\approx Tier) \\
\mathrm{Dual_F}(\widetilde{lE}, \widetilde{\widetilde{E}}, vcb, end, r) & \triangleq & (\ vcb(ix).\,(\nu\, r_1)(\ \mathrm{LisVec}\langle\, lE_{ix}, r_1\,\rangle \\
& & \quad \mid\ r_1(E_{ix}).\,\mathrm{Dual}_F\langle\, \widetilde{lE}, \widetilde{\widetilde{E}}, vcb, end, r\,\rangle \\
& & +\ end.\,\overline{r}\langle\, \widetilde{\widetilde{E}}\,\rangle.\,\mathbf{0}\ )
\end{array}
$$

El proceso Dual era el único restante, y su funcionalidad es similar a la de los procesos definidos en el Apéndice: pasar de una estructura de doble lista a una de doble vector. Se define aquí solamente porque tiene una estructura menos general.

Finalmente, el generador de un proceso GenProc es un proceso que parte de un avatar y un metacomponente, y construye a partir de ellos tantas copias del proceso correspondiente

–indicado en †**G** por el arquetipo– como sea preciso, según el grado de reflexión $\eth$, aquí representado por el campo †**H** del arquetipo. Se define como:

$$
\begin{array}{rcl}
\text{GenProc} & : & (\uparrow \mathit{TComp},\ \uparrow \mathcal{M}\mathit{Comp},\ \uparrow \mathit{Sync}) \\
\text{GenProc}(ka, km, kd) & \triangleq & ka(ava).\, km(mt). \\
& & \quad \text{Descend}\langle\, mt{\dagger}H, mt{\dagger}G, \\
& & \qquad (\texttt{cons}\ ava{\dagger}K\ (\texttt{nil})), ava, kd\rangle
\end{array}
$$

Donde, obviamente, el aspecto clave es el proceso recursivo Descend. Éste, simplemente, recibe la estructura correspondiente a los niveles superiores del meta-camino que está recorriendo, y comprueba si ha descendido lo suficiente. Si es así, finaliza la estructura *Tower* que se pasará como parámetro al proceso –la *torre reflexiva* descrita en la sección 7.2.5–, e invoca el proceso, mandando una señal en el puerto especial *gen*, recogiendo además su señal de terminación para poder destruirlo en el momento requerido. En caso contrario, si no se ha profundizado lo suficiente, Descend aumenta la estructura de la torre, pasa del proceso activo a su brhátar, y desciende otro nivel, según el proceso ya explicado.

$$
\begin{array}{rcl}
\text{Descend} & : & (\mathit{Int},\ \uparrow \mathit{Proc},\ \mathit{Tower},\ \mathit{TComp},\ \uparrow \mathit{Sync}) \\
\text{Descend}(h, gen, Tr, ava, kd) & \triangleq & \textbf{if}\ \ (h = 1)\ \ \textbf{then} \\
& & \quad \overline{gen}\langle\, (\texttt{cons}\ ava\ Tr), kd\,\rangle.\, \mathbf{0} \\
& & \quad \textbf{else}\ \ (\nu\, v_r, e_r)(\ ava{\dagger}M(mc).\, \overline{ava{\dagger}M}\langle\, mc\,\rangle. \\
& & \qquad mc{\dagger}R(lr).\, \overline{mc{\dagger}R}\langle\, lr\,\rangle.\, \text{LFor}\langle\, lr, v_r, e_r\,\rangle \\
& & \qquad\quad \mid\ (\nu\, rd)(\ \text{Desc}_F\langle\, h, gen, (\texttt{cons}\ ava\ Tr), \\
& & \qquad\qquad\quad v_r, e_r, (\texttt{nil}), rd\rangle \\
& & \qquad\qquad \mid\ (\nu\, kill)(\ rd(lk).\, \overline{kd}\langle\, kill\,\rangle. \\
& & \qquad\qquad\quad kill.\, \text{LSync}\langle\, lk\,\rangle.\, \mathbf{0}\ )) \\
& & \qquad\quad)
\end{array}
$$

$$
\begin{array}{rcl}
\text{Desc}_{\text{F}} & : & (\mathit{Int},\ \uparrow \mathit{Proc},\ \mathit{Tower},\ \uparrow(\mathit{TReif}\ \mathit{TComp}), \\
& & \quad \mathit{Sync},\ (\mathit{List}\ \mathit{Sync}),\ \uparrow(\mathit{List}\ \mathit{Sync})) \\
\text{Desc}_{\text{F}}(h, gen, Tr, vcb, end, lk, rd) & \triangleq & vcb(rei).\, (\nu\, r)(\\
& & \quad \text{Descend}\langle\, (h-1), gen, Tr, (rei{\dagger}A), r\,\rangle \\
& & \qquad \mid\ r(k_s).\, \text{Desc}_F\langle\, h, gen, Tr, vcb, end, \\
& & \qquad\qquad (\texttt{cons}\ k_s\ lk), rd\rangle\ \ ) \\
& & \quad +\ end.\, \overline{rd}\langle\, lk\,\rangle.\, \mathbf{0}\ )
\end{array}
$$

$$
\begin{array}{rcl}
\text{LSync} & : & ((\mathit{List}\ \mathit{Sync})) \\
\text{LSync}(lis) & \triangleq & (\nu\, vcb, end)(\ \text{LFor}\langle\, lis, vcb, end\,\rangle \\
& & \quad \mid\ \text{LSy}_F\langle\, vcb, end\,\rangle\ ) \\
\text{LSy}_{\text{F}} & : & (\uparrow \mathit{Sync},\ \mathit{Sync})
\end{array}
$$

$$\mathrm{LSy_F}(vcb, end) \triangleq (\ vcb(die).\,\overline{die}.\,\mathrm{LSy}_F\langle\, vcb, end\,\rangle + end.\,\mathbf{0}\ )$$

El único aspecto particular de esta definición es el hecho de que el proceso correspondiente se invoque mediante una señal en un canal especial (*gen*). Esto, sin embargo, no es nada extraño en el cálculo-π, una vez que existe la estructura de replicación.

Se asume que el proceso de traducción incorpora el proceso siguiente. Su uso es global, y expresa a la suma replicada de un conjunto de procesos que procede de las definiciones de las distintas restricciones, cada una ligada a un identificador g_M diferente, generado también por el traductor. Esto está en el mismo espíritu que la traducción estándar del cálculo-π de primer orden al HO-π, y evita tener que remitirse a la variante extendida del cálculo.

$$\mathrm{Dispatch}(\widetilde{g}) \triangleq\ !\left(\sum_M g_M(Tr, kd).\,(\nu\, die)(\ \overline{kd}\langle\, die\,\rangle.\,\mathrm{Proc}_M\langle\, Tr, die\,\rangle\ \right)$$

En realidad, se trata de un mero disparador, que por cada señal recibida selecciona el proceso correspondiente, y genera una nueva copia del mismo. Esta estructura puede parecer muy grande, pero en el caso normal se limitará a unos pocos procesos.

Todos los procesos se generarán, además, con una forma análoga a:

$$\begin{aligned} \mathrm{Proc_N} &: (Tower,\ Sync) \\ \mathrm{Proc_N}(Tr, die) &\triangleq (\ \ldots + die.\,\mathbf{0}\ ) \end{aligned}$$

No obstante, esta representación esquemática es sólo una aproximación. En realidad, la selección no determinista con la señal de terminación *die* no adopta esa forma, sino que se plantea como una alternativa a la definición recursiva del proceso. Esto es totalmente análogo al esquema con el que se traduce una *replicación* de $\mathcal{P}i\mathcal{L}ar$, tal como se describe en la sección 7.4.3, concretamente en la página 267, donde se examina con mayor detenimiento.

Todo este esquema, como la definición de componente en general, se aplica igualmente a las conexiones, aunque de una forma mucho más sencilla. De hecho, como se verá en la sección siguiente, la estructura de éstas puede verse como un fragmento del tipo de componente *TComp*. Por este motivo, como ya se ha dicho, éstas serán tratadas ocasionalmente como un tipo especial de componente, y en ciertos momentos se considerará que el tipo *TComp* hace referencia a ambas estructuras. No obstante, las conexiones tienen sus propias particularidades, en especial en lo relativo al comportamiento; éstas se describen en la sección siguiente.

7.3.5 Conexión

La Figura 7.10 muestra la estructura de una conexión, que como puede verse es idéntica a la de un componente primitivo, en el que se hubieran eliminado los campos irrelevantes para la concepción de un enlace. Tan sólo consta de interfaz, y de una referencia al arquetipo –que ejerce el papel de *conector de nivel meta*, en este caso– correspondiente, que lo integra en una estructura similar a la descrita en el apartado anterior, tal como ya se había señalado.

El tipo correspondiente es, como en todos los demás casos, recursivo, y se puede considerar como un fragmento de la estructura *TComp*. Se define del modo siguiente:

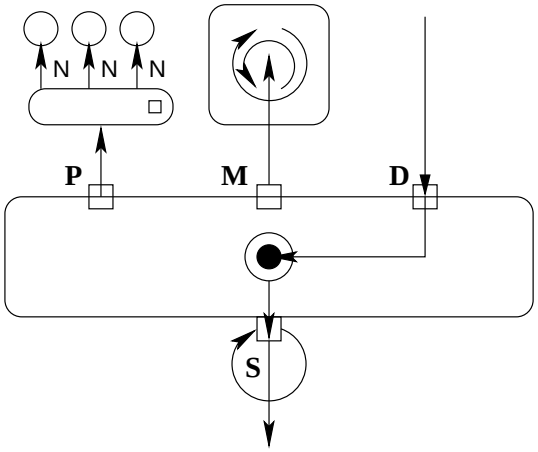

Figura 7.10: Estructura Interna de una Conexión $\mathcal{P}i\mathcal{L}ar$

$$
\begin{aligned}
TBind \quad \triangleq \quad & \mu E.(\,\dagger P{:}\uparrow(NList\ \uparrow DPort),\\
& \dagger M{:}\uparrow\mathcal{M}Comp,\\
& \dagger D{:}\ Sync,\ \dagger S{:}\uparrow E\,))
\end{aligned}
$$

Por tanto, su estructura es la más sencilla de todas, ya que consta únicamente de cuatro campos. Éstos se definen de manera análoga a sus homónimos de *TComp*, con un significado similar. Éste, en todo caso, se resume igualmente a continuación.

- ☐ **P** $\mapsto$ Se trata de la lista de sus puertos, es decir, la interfaz de la conexión, que indica qué instancias se conectan.

- ☐ **M** $\mapsto$ Se trata de una referencia al arquetipo correspondiente, que en este caso tendrá funciones de conector de nivel meta.

- ☐ **D** $\mapsto$ Se trata, nuevamente, de la señal de terminación.

- ☐ **S** $\mapsto$ Se trata, nuevamente, del canal recursivo, asociado a RepSelf, que vincula esta estructura a su nombre.

La clave de la creación de una conexión es la forma en que se generan los puertos de su interfaz, ya que a diferencia del caso de los componentes, estos están conectando a elementos ya existentes. Por tanto, estos puertos generan sólo la estructura interna, pero en su interfaz *externa* –el campo †**O** del *TPort* clásico–, se utilizan los canales preexistentes, correspondientes a dicho interfaz en el puerto de un componente. El proceso de creación de un puerto ligado es, por tanto, prácticamente idéntico al de un puerto normal, exceptuando ese único detalle.

$$
\begin{array}{rcl}
\text{BPort} & : & (\sharp\alpha,\ \sharp\beta;\ (\mathit{Par}\,\alpha\,\beta),\ \uparrow\mathit{TComp},\ \uparrow(\mathit{TPort}\,\alpha\,\beta)) \\
\text{BPort}(\widehat{e}, c, p) & \triangleq & (\nu\, ii, io, wi, wo, sh, ct, bp, kill, me, k_1, k_2) \\
& & (\ \overline{bp}\langle\,(\texttt{cons}\ c\ (\texttt{nil}))\,\rangle.\,\mathbf{0} \mid kill.\,\overline{k_1}.\,\overline{k_2}.\,\mathbf{0} \\
& & \mid me(p_1).\,\overline{p}\langle\,p_1\,\rangle.\,\mathbf{0} \\
& & \mid \text{RepSelf}\langle\, me, (\dagger I \prec [ii, io], \dagger O \prec [e_o, e_i], \\
& & \qquad \dagger W \prec [wi, wo], \dagger M \prec [sh, ct], \\
& & \qquad \dagger D \prec kill, \dagger B \prec bp, \\
& & \qquad \dagger S \prec me\ ), k_1\rangle \\
& & \mid \text{BehPort}\langle\,[ii, io], [e_o, e_i], [wi, wo], [sh, ct], k_2\,\rangle)
\end{array}
$$

Para mantener una conexión, de modo que el comportamiento sea el deseado, tan sólo es necesario, por tanto, *cruzar* los canales externos, de modo que se mantenga la compatibilidad de tipos, y las interacciones sigan el camino adecuado. Por lo demás, el sistema no cambia en absoluto, y no es necesario redefinir ni añadir ningún proceso; una vez creado, este puerto es completamente idéntico a cualquier otro existente.

Para comprender mejor el comportamiento de una conexión, se planteará el caso ideal de una conexión binaria, esto es, un enlace. Debe de tenerse en cuenta que esta no será la verdadera definición, en realidad, ya que en último término todas las conexiones –incluidas las binarias– serán traducidas siguiendo el mismo patrón de proceso. Pero el esquema aquí descrito no solamente es muy similar, sino mucho más fácil de comprender, al tratarse del caso reducido de únicamente dos extremos. De hecho, la definición que realmente se utiliza, esto es, la del proceso Bind, desarrollada en la página 257, se ha construido a partir de la generalización al caso n-ario de esta descripción para una conexión binaria.

$$
\begin{array}{rcl}
\text{Link} & : & (\uparrow\mathit{TComp},\ \uparrow\mathit{DPort},\ \uparrow\mathit{TComp},\ \uparrow\mathit{DPort},\ \uparrow\mathit{TBind}) \\
\text{Link}(c_1, p_1, c_2, p_2, b) & \triangleq & (\nu\, mp, bp, kill, me, k_1, k_2, k_3) \\
& & p_1(e_1).\,p_2(e_2). \\
& & \quad (\ \text{BPort}\langle\, e_1\dagger O, c_1, mp\,\rangle \mid \text{BPort}\langle\, e_2\dagger O, c_2, mp\,\rangle \\
& & \quad \mid\ e_1\dagger B(l_1).\,\overline{e_1\dagger B}\langle\,(\texttt{cons}\ me\ l_1)\,\rangle \\
& & \quad \mid\ e_2\dagger B(l_2).\,\overline{e_2\dagger B}\langle\,(\texttt{cons}\ me\ l_2)\,\rangle \\
& & \quad \mid\ mp(ip_1).\,mp(ip_2). \\
& & \qquad (\ kill.\,\overline{k_1}.\,\overline{k_2}.\,\overline{k_3}.\,\overline{ip_1\dagger D}.\,\overline{ip_2\dagger D}.\,\mathbf{0} \\
& & \qquad \mid\ \overline{bp}\langle\,(\texttt{ncons}\ \texttt{“O”}\ ip_2\ (\texttt{ncons}\ \texttt{“I”}\ ip_1\ (\texttt{nil})))\,\rangle.\,\mathbf{0} \\
& & \qquad \mid\ \text{BehLink}\langle\, ip_1\dagger I, ip_2\dagger I, k_1\,\rangle \\
& & \qquad \mid\ \text{BehLink}\langle\, ip_2\dagger I, ip_1\dagger I, k_2\,\rangle \\
& & \qquad \mid\ \text{RepSelf}\langle\, me, (\dagger P \prec bp, \dagger D \prec kill, \\
& & \qquad\quad \dagger M \prec \mathbf{TLink}, \dagger S \prec me), k_3\,\rangle\ )
\end{array}
$$

$$
\begin{array}{rcl}
 & & \mid\ me(b_1).\,\overline{b}\langle\, b_1 \,\rangle.\,\mathbf{0} \\
 & &) \\
\text{BehLink} & : & (\sharp\alpha,\ \sharp\beta;\ (Par\,\alpha\,\beta),\ (Par\,\beta\,\alpha),\ Sync) \\
\text{BehLink}(\widehat{a},\widehat{b},die) & \triangleq & a_i(z).\,\overline{b_o}\langle\, z \,\rangle.\,\text{BehLink}\langle\, \widehat{a},\widehat{b},die \,\rangle \\
 & & +\ die.\,\mathbf{0}
\end{array}
$$

La definición es trivial: simplemente se crean los puertos, ya ligados, según el proceso descrito. Éstos se acumulan en la lista que representa a la interfaz, y con ésta como elemento principal, se elabora el registro correspondiente a la estructura de conexión. Éste, finalmente, se vincula consigo mismo a través de una estructura RepSelf, como es el caso habitual. Además, durante el proceso, la referencia a la conexión (me) se acumula en la lista señalada por el campo †**B** de los dos puertos de instancia utilizados, de modo que se actualiza la información necesaria para el operador **bound**, que se describe en la sección 7.4.6.

Lo único digno de ser señalado en este caso es que, al tratarse de un enlace simple, está utilizando (por definición) el tipo predefinido **link** de $\mathcal{P}i\mathcal{L}ar$. Se puede asumir entonces la existencia de una estructura tipo $\mathcal{M}Comp$ específica, que describa al metacomponente de un enlace. Ésta ha sido designada internamente como **TLink**, y así ha sido indicado en la definición –aunque, en realidad, no es necesario designarlo como una constante, y bastaría con disponer de una referencia a dicha estructura–. Por otra parte, al ser un enlace, tiene un comportamiento (BehLink) predeterminado, que consiste simplemente en el envío de los mensajes de un extremo a otro. En una conexión tipada, en cambio, el comportamiento habrá de ser específicamente definido en cada caso, mediante una regla del Lenguaje Dinámico.

Se ha demostrado formalmente –aunque no se desarrollará aquí, por ser una demostración larga e innecesaria– que esta estructura es débilmente bisimilar al simple *cruce de puertos* que se realiza en la definición de un proceso BPort. Por tanto, se demuestra con esto que un *enlace* es totalmente equivalente –por observación– a una conexión algebraica pura, a pesar de disponer también de una estructura completa.

Pero esto sólo es el desarrollo del caso simple. El de la conexión genérica es más complicado, pero la explicación en realidad no varía; tan sólo hay que definir un bucle que recorre a los puertos, ya que ahora podrán ser más de dos. Además, aquí el protocolo del enlace no está predefinido (no hay ya un BehLink), sino que vendrá dado por la traducción de las restricciones dinámicas, como en cualquier componente.

$$
\begin{array}{rcl}
\text{Bind} & : & (\uparrow\mathcal{M}Comp,\ \sim\!\uparrow TComp,\ \sim\!\uparrow DPort,\ \sim Label,\ \uparrow TBind) \\
\text{Bind}(T,\widetilde{c},\widetilde{p},\widetilde{L},b) & \triangleq & (\nu\,\widetilde{mp}, bp, kill, me, k_1) \\
 & & (\ (\nu\,v_1,e_1)(\ \text{NFor}\langle\, |\widetilde{p}|, v_1, e_1 \,\rangle \\
 & & \quad \mid\ \text{NewPort}_F\langle\, \widetilde{c},\widetilde{p},v_1,e_1,me,\widetilde{mp} \,\rangle\ ) \\
 & & \mid (\nu\,v_2,e_2)(\ \text{NFor}\langle\, |\widetilde{L}|, v_2, e_2 \,\rangle \\
 & & \quad \mid\ \text{ListP}_F\langle\, (\texttt{nil}), \widetilde{L}, \widetilde{mp}, v_2, e_2, bp \,\rangle\ ) \\
 & & \mid kill.\,\overline{k_1}.\,(\ \text{KProc}\langle\, T, me \,\rangle \mid (\nu\,v_3,e_3) \\
 & & \quad (\ bp(lis).\,\text{LFor}\langle\, lis, v_3, e_3 \,\rangle \mid \text{Kill}_F\langle\, v_3, e_3 \,\rangle\ ))
\end{array}
$$

$$
\begin{array}{l}
\mid \mathrm{RepSelf}\langle\, me, (\dagger P \prec bp, \dagger D \prec kill, \\
\qquad \dagger M \prec T, \dagger S \prec me), k_1 \rangle \\
\mid me(b_1).\, \overline{b}\langle\, b_1 \,\rangle.\, \mathbf{0})
\end{array}
$$

Los parámetros de la definición son los mismos que ya se han manejado en varias ocasiones en las estructuras del apartado anterior: el tipo de la conexión, la lista de los componentes a conectar, los puertos concretos de esos componentes que van a ser utilizados, y la lista de etiquetas correspondientes a los nombres de puerto.

Por lo demás, el esquema es totalmente análogo al de una conexión binaria, pero generalizado al caso n-ario. Por tanto, el proceso simplemente recorre las listas de puertos a ligar, crea los puertos ligados correspondientes, los acumula en una lista, construye el registro necesario, y lo vincula a su nombre con un RepSelf, después de haber preparado las señales de terminación de la manera adecuada, a través del subproceso KProc. Sea pues:

$$
\begin{array}{rcl}
\mathrm{NewPort_F} & : & (\sim\!\uparrow TComp,\ \sim\!\uparrow DPort,\ \uparrow Int,\ Sync,\ \uparrow TBind,\ \sim\!\uparrow DPort) \\
\mathrm{NewPort_F}(\widetilde{c}, \widetilde{p}, vcb, end, b, \widetilde{r}) & \triangleq & (\ vcb(ix).\, p_{ix}(e_{ix}).\, (\ \mathrm{BPort}\langle\, e_{ix}\dagger O, c_{ix}, r_{ix}\,\rangle. \\
 & & \qquad e_{ix}\dagger B(bp).\, \overline{e_{ix}\dagger B}\langle\, (\mathsf{cons}\ b\ bp)\,\rangle \\
 & & \qquad \mid \mathrm{NewPort}_F\langle\, \widetilde{c}, \widetilde{p}, vcb, end, b, \widetilde{r}\,\rangle\ ) \\
 & & + end.\, \mathbf{0} \\
\mathrm{ListP_F} & : & ((NList\ \uparrow DPort),\ \sim Label,\ \sim\!\uparrow DPort,\ \uparrow Int, \\
 & & \qquad Sync,\ \uparrow(NList\ \uparrow DPort)), \\
\mathrm{ListP_F}(lis, \widetilde{L}, \widetilde{mp}, vcb, end, r) & \triangleq & (\ vcb(ix). \\
 & & \qquad \mathrm{ListP}_F\langle\, (\mathsf{ncons}\ L_{ix}\ mp_{ix}\ lis), \widetilde{L}, \widetilde{mp}, vcb, end, r\,\rangle \\
 & & + end.\, \overline{r}\langle\, lis\,\rangle.\, \mathbf{0} \\
\mathrm{KProc} & : & (\uparrow\mathcal{M}Comp,\ \uparrow TBind) \\
\mathrm{KProc}(mc, bin) & \triangleq & (\nu\, r)(\ bin(ava).\, mc(m).\, m\dagger R(lr).\, \mathrm{AvIn}\langle\, ava, lr, r\,\rangle \\
 & & \qquad \mid\ r(die).\, \mathbf{if}\ \ (die \neq\ ava\dagger D)\ \ \mathbf{then} \\
 & & \qquad\quad \mathrm{DelRei}\langle\, ava, die, lr, m\dagger R\,\rangle \\
 & & \qquad \mathbf{else}\ \ \mathbf{0} \\
\mathrm{DelRei} & : & (TComp,\ Sync,\ (List\,(TReif\ TComp)), \\
 & & \qquad \uparrow(List\,(TReif\ TComp))) \\
\mathrm{DelRei}(ava, die, lr, r) & \triangleq & \overline{r}\langle\, (era, (\dagger A \prec (ava\dagger S), \dagger D \prec die)\ lr)\,\rangle.\, \overline{die}.\, \mathbf{0} \\
\mathrm{Kill_F} & : & (\sharp\alpha;\ \uparrow(Named\,\alpha),\ Sync) \\
\mathrm{Kill_F}(vcb, end) & \triangleq & (\ \overline{vcb\ddagger d\dagger D}.\, \mathrm{Kill}_F\langle\, vcb, end\,\rangle + end.\, \mathbf{0}\ )
\end{array}
$$

Como ha podido verse, en este caso el proceso $\mathsf{NewPort}_F$ se encarga de crear un nuevo puerto ligado, aprovechando también la ocasión para actualizar el campo †**B** del puerto de instancia correspondiente, de modo que se mantenga la meta-información necesaria para el operador **bound**. Por su parte, ListP_F construye la lista de puertos de la conexión, que define

su interfaz. El resto de los procesos definidos tan sólo describen el modo en que se destruye toda esta estructura en respuesta a una señal de terminación. Particularmente significativo, entre estos últimos, es el proceso –DelRei– que se encarga de destruir la reificación entre la conexión que va a ser destruida y su rohatar, evitando así dejar señales en el meta-nivel.

Para los efectos del apartado anterior, se define también un proceso NewBind, similar al NewComp de los componentes. Pasa por una fase de creación, análoga pero más sencilla que el comportamiento de GenComp, y el proceso se genera con una invocación a GenProc, exactamente de la misma manera que ya se conoce.

$$
\begin{array}{rcl}
\text{NewBind} & : & (\uparrow\mathcal{M}Comp,\ (List\ \uparrow TComp),\ (List\ \uparrow DPort),\ \uparrow TBind) \\
\text{NewBind}(T, lc, lp, nb) & \triangleq & (\nu\, r_b, r_1)(\ \text{LisVec}\langle\, lc, r_1\,\rangle \mid r_1(\widetilde{c}). \\
& & \quad (\nu\, r_2)(\ \text{LisVec}\langle\, lp, r_2\,\rangle \mid r_2(\widetilde{p}). \\
& & \quad (\nu\, r_3)(\ \text{LisVec}\langle\, T{\ddagger}P, r_3\,\rangle \mid r_3(\widetilde{L}). \\
& & \qquad \text{Bind}\langle\, T, \widetilde{c}, \widetilde{p}, \widetilde{L}, r_b\,\rangle\ )) \\
& & \quad \mid\ (\nu\, rd)(\ r_b(b).\,(\ \text{GenProc}\langle\, b{\dagger}S, T, rd\,\rangle \\
& & \qquad \mid\ rd(die).\, T{\dagger}R(lr).\, \overline{nb}\langle\, b\,\rangle. \\
& & \qquad\quad \text{NewReif}\langle\, lr, b, die, T{\dagger}R\,\rangle\ )))
\end{array}
$$

Con estos detalles puede darse por terminada esta sección, excepto en lo relativo al otro tipo especial de enlace, esto es, la conexión jerárquica. Para éste se harán también algunos comentarios especiales, aunque en definitiva será tratado, al igual que el enlace simple, como un caso particular de la estructura genérica de conexión.

Conexión Jerárquica

En realidad, una conexión jerárquica es prácticamente idéntica a un enlace simple, por lo que no se ha de decir demasiado. La única diferencia significativa es que, en este caso, cuando la conexión se vincula a un puerto del componente *contenedor* –esto es, un componente compuesto que engloba a la propia conexión– lo está haciendo *desde dentro*. Por tanto, el puerto creado en la conexión no ha de ligarse a la parte externa del puerto contenedor, como hasta ahora, sino a la *interna*. Esto se refleja en la definición incluida a continuación.

Ésta, por lo demás, consiste exactamente en el mismo proceso que se ha proporcionado para la definición de un enlace simple (Link). La única diferencia es la ya señalada, que se expresa en el fragmento de proceso siguiente, correspondiente al momento en que se crean los puertos ligados, y en el que se utiliza el interfaz interno **†I** del puerto contenedor.

$$
\begin{array}{rcl}
\text{HBind}(c_C, p_C, c_S, p_S, b) & \triangleq & (\ \ldots\ p_C(e_C).\, p_S(e_S). \\
& & \quad (\ \text{BPort}\langle\, e_C{\dagger}I, c_C, mp\,\rangle \mid \text{BPort}\langle\, e_S{\dagger}O, c_S, mp\,\rangle \\
& & \quad \ldots\ ))
\end{array}
$$

Aunque se podrían haber mantenido los nombres del proceso Link, los parámetros se indican para mayor claridad de otro modo: así, c_C, p_C son el componente y puerto *contenedor*, mientras

que c_S, p_S son el subcomponente contenido, y su puerto. Por lo demás, el proceso es idéntico a un *TLink*, tanto en su comportamiento como en su rohatar ($\dagger M$).

Con esto se da por concluido el apartado correspondiente a la definición de conexiones, así como el bloque relativo a las grandes estructuras semánticas del lenguaje. En lo que resta, se comentarán brevemente algunos de los detalles "menores" del Lenguaje Estático.

7.3.6 Estructuras Sintácticas Adicionales

Aparte de las grandes estructuras que se acaban de describir, la definición del lenguaje estático incluye algunos otros aspectos y construcciones de menor importancia. Algunos detalles son muy sencillos, como las cuestiones relativas a la cualificación y localización de nombres, la traducción de las etiquetas o la definición de vectores de instancias y puertos. No obstante, hay una serie de construcciones sintácticas de cierta envergadura que merecen un breve comentario. En este apartado se tratan con algún detalle las más importantes de ellas.

Dado que en este caso no se trata tanto de estructuras de proceso como de traducciones de esquemas sintácticos, se adopta en este apartado la misma notación que será empleada, de manera más general, durante la descripción semántica del Lenguaje Dinámico. Ésta se describe en la sección 7.4.1, donde ya se proporcionan todas las explicaciones necesarias.

Conjuntos

La estructura de conjunto es la única que es manejada de manera implícita por $\mathcal{P}i\mathcal{L}ar$, tanto por el Lenguaje Estructural como por el Dinámico. De hecho, sobre la misma se construyen tres operadores básicos, procedentes de la teoría de conjuntos, que son los que le dotan de cierta importancia. El primero es cualquiera de los bucles que iteran sobre sus elementos, y que serán descritos más abajo. Los otros dos son el predicado de pertenencia (**in**) y la cardinalidad (**card**), que permite determinar su tamaño.

En realidad, es habitual que una semántica considere a los conjuntos como primitivos, al ser entidades matemáticas básicas. Sin embargo, esto no es estrictamente necesario aquí, ya que pueden ser integrados como estructuras del cálculo-π. Así, pueden verse como vectores de tamaño indefinido, o incluso como listas; esto último es lo que se hace a la hora de implementarlos *dentro* del cálculo, en cualquiera de los lenguajes derivados, y particularmente en Pict. Por tanto, en adelante se considera que un conjunto viene implementado como una lista o tupla de cualquier tamaño, para los efectos de este apartado, aunque se considerará como un tipo primitivo cuando ésto resulte conveniente, sin pérdida de generalidad.

Una vez que se dispone en el cálculo de operadores condicionales y lógicos, así como de la noción de conjunto, resulta casi trivial elaborar un predicado de pertenencia. Por ejemplo, Zhang y Potter [ZP97] lo elaboran a partir del operador de emparejamiento (*matching*), que es, como se sabe, la forma más primitiva de condicional existente. De modo que:

$$[a \in \tilde{S}].P \quad \triangleq \quad \prod_{i=1}^{n} [a = s_i].P \quad \text{tal que } \tilde{S} = \{s_1, s_2, \ldots, s_n\}$$

No obstante, este enfoque no parece aquí interesante, ya que mientras ellos lo orientan a la definición de una estructura condicional compleja, lo que se pretende en $\mathcal{P}i\mathcal{L}ar$ es disponer de

su resultado como un valor lógico –booleano– que permita utilizarlo *después* en las estructuras alternativas del propio lenguaje. De modo que se utiliza la codificación siguiente:

$$[\![\, \underline{a} \textbf{ in } \underline{S} \,]\!] \;\stackrel{\circ}{=}\; \begin{cases} \blacktriangleright\ (\texttt{ein}\ \underline{a}\ \underline{S}) & \text{si } \underline{S} \text{ lista} \\ \blacktriangleright\ (\texttt{nein}\ \underline{a}\ \underline{S}) & \text{si } \underline{S} \text{ lista nombrada} \\ \blacktriangleright\ \bigvee_{i=1}^{n}(\underline{a} = s_i) & \text{si } \underline{S} = \widetilde{s} \text{ tupla} \end{cases}$$

Se trata de una definición por partes: las distintas ramas expresan los distintos tipos de datos que, dentro del cálculo, pueden estar representando en cada momento lo que en $\mathcal{P}i\mathcal{L}ar$ es un conjunto abstracto. No obstante, la definición del predicado de pertenencia es en los tres casos trivial: así, cuando se trata de una lista, simplemente se utiliza la función de búsqueda; mientras que si es una tupla o vector, se compara el valor con cada uno de sus elementos.

Por su parte, el operador de cardinalidad es igualmente muy sencillo, y su traducción se plantea de un modo muy similar:

$$[\![\, \textbf{card}(\underline{S}) \,]\!] \;\stackrel{\circ}{=}\; \begin{cases} \blacktriangleright\ (\texttt{length}\ \underline{S}) & \text{si } \underline{S} \text{ lista} \\ \blacktriangleright\ (\texttt{length}\ \underline{S}) & \text{si } \underline{S} \text{ lista nombrada} \\ \blacktriangleright\ |\widetilde{s}| & \text{si } \underline{S} = \widetilde{s} \text{ tupla} \end{cases}$$

En realidad, esta traducción podría ser mucho más simple, ya que en este caso no ha lugar hacer distinciones entre listas y listas nombradas; pero se ha preferido indicarlo así para mantener la analogía con el operador anterior. Por lo demás, las tres definiciones hacen uso de las respectivas funciones primitivas.

Eventualmente, podría ser interesante disponer de operadores para operaciones más complejas como la unión o la intersección. Sin embargo, no son estrictamente necesarias, ya que pueden expresarse en $\mathcal{P}i\mathcal{L}ar$ mediante la combinación de condicionales, bucles y el predicado de pertenencia, sin necesidad de anidamiento.

Estructura Alternativa

La estructura alternativa es uno de los dos constructores fundamentales del Lenguaje Estructural, junto a la iterativa; y se aplica además, e incluso con la misma semántica, en el Lenguaje Dinámico. No obstante, y a pesar de su importancia, su definición es absolutamente trivial, desde el momento el que el cálculo –esto es, el dialecto definido en el Apéndice C– dispone de esta misma estructura como primitiva. La única diferencia se relaciona con el grado de abstracción: mientras la condicional del cálculo-$\hat{\pi}$ es consciente de que manipula *procesos*, la de $\mathcal{P}i\mathcal{L}ar$ manipula *estructuras*, que son a su vez representadas mediante procesos.

En $\mathcal{P}i\mathcal{L}ar$ se dispone tanto de la estructura alternativa simple como de la doble, y ambas se definen como se indica a continuación.

$$\begin{aligned} [\![\backslash\textbf{if}\,(co)\,(\underline{P_1})\,]\!] \;&\stackrel{\circ}{=}\; \textbf{if}\ [\![co]\!]\ \textbf{then}\ (\ \underline{P_1}\ ) \\ &\qquad\quad \textbf{else}\ \textbf{0} \\ [\![\backslash\textbf{if}\,(\underline{co})\,(\underline{P_1})\,(\underline{P_2})\,]\!] \;&\stackrel{\circ}{=}\; \textbf{if}\ [\![co]\!]\ \textbf{then}\ (\ \underline{P_1}\ ) \\ &\qquad\quad \textbf{else}\ (\ \underline{P_2}\ ) \end{aligned}$$

En definitiva, ambas estructuras son prácticamente idénticas, y no requieren de mayor explicación. Por otra parte, en $\mathcal{P}i\mathcal{L}ar$ no se ha definido ninguna estructura alternativa múltiple, aunque proporcionarla sería, en cualquier caso, trivial.

Bucles

Como puede comprobarse en el Apéndice correspondiente (§C.2.1), la traducción de un bucle convencional a los términos del cálculo-π es muy sencilla, una vez que se dispone de condiciones booleanas y valores lógicos. No obstante, los bucles de $\mathcal{P}i\mathcal{L}ar$ tienen características concretas, que hacen interesante detenerse brevemente en su definición.

En primer lugar, estos bucles se aplican únicamente sobre estructuras de tipo conjunto, lo que en este caso incluye también, de manera específica, a rangos de números enteros. En definitiva, se plantean como *iteradores* sobre estos conjuntos, que describen un recorrido sobre sus elementos de distinto modo según el tipo subyacente.

El bucle más sencillo es el iterador secuencial **for**, que define un recorrido del conjunto elemento a elemento, y en un orden determinado. Se garantiza, por tanto, que el elemento i-ésimo sólo es accedido cuando se han recorrido ya los $(i-1)$ anteriores, y tan sólo varía el criterio que se utiliza para definir *anterior*.

Como puede verse, lo que se define en este punto no son procesos concretos, sino *esquemas* de traducción, representados como un par de procesos: el que realiza el bucle en sí, que será un proceso predefinido –como LFor o NFor–, y el *cuerpo* del bucle, marcado con el subíndice $_F$, y que expresa el conjunto de acciones que se realizan en cada una de sus vueltas. La relación entre los iteradores y el proceso-cuerpo se describe en la sección D.1, donde se especifica también la estructura de los procesos predefinidos necesarios.

$$
\begin{array}{rcl}
[\![\backslash\mathbf{for}\ \underline{v}\,(\underline{S})\,(\underline{P})\,]\!] & \stackrel{\circ}{=} & \left\{\begin{array}{ll}
(\nu\, v_1, end)(\ \mathrm{LFor}\langle\, \underline{S}, v_1, end\,\rangle & \\
\quad \mid\ \mathrm{Cuerpo}_F\langle\, v_1, end\,\rangle\ ) & \text{si } \underline{S} \text{ lista} \\
(\nu\, v_1, end)(\ \mathrm{LLFor}\langle\, \underline{S}, v_1, end\,\rangle & \\
\quad \mid\ \mathrm{Cuerpo}_F\langle\, v_1, end\,\rangle\ ) & \text{si } \underline{S} \text{ lista nombrada} \\
(\nu\, v_1, end)(\ \mathrm{NFor}\langle\, |\widetilde{s}|, v_1, end\,\rangle & \\
\quad \mid\ \mathrm{Cuerpo}_F\langle\, v_1, end\,\rangle\ ) & \text{si } \underline{S} = \widetilde{s} \text{ tupla} \\
(\nu\, v_1, end)(\ \mathrm{IFor}\langle\, inf, sup, v_1, end\,\rangle & \\
\quad \mid\ \mathrm{Cuerpo}_F\langle\, v_1, end\,\rangle\ ) & \text{si } \underline{S} = inf..sup
\end{array}\right. \\
[\![\nu]\!]\ \mathrm{Cuerpo_F}(vcb, end) & \triangleq & (\ vcb(\underline{v}).\,(\ \underline{\mathrm{P}}\langle\, \underline{v}\,\rangle\ ;\ \mathrm{Cuerpo}_F\langle\, vcb, end\,\rangle\ ) \\
& & +\ end.\,\mathbf{0}\ )
\end{array}
$$

Como puede verse, la única diferencia entre las distintas variantes es el uso, en cada caso, de un proceso distinto según el tipo de conjunto utilizado. Todos ellos –LFor, LLFor, NFor e IFor– son definidos formalmente en la sección D.1 del Apéndice D, y muchos de ellos han sido también utilizados con frecuencia a lo largo de todo este apartado.

En realidad, la existencia de Cuerpo_F es un truco del traductor. Hace más bien el papel de un proceso parametrizado que tuviera estas mismas variables, más el nombre libre $\underline{v}$ y el proceso $\underline{P}(\underline{v})$. Es perfectamente posible expresar una construcción de este tipo utilizando el cálculo-π

extendido, pero no el cálculo-$\hat{\pi}$. No obstante, eso no es aquí necesario: basta con suponer que el traductor crea en cada ocasión un proceso Cuerpo_F con un nombre diferente. En realidad, más que disponer de un subproceso $\underline{P}$ como parámetro, lo que se hace es *construir* el proceso necesario, a partir de estos ingredientes.

Por su parte, la definición de un bucle paralelo **loop** es prácticamente idéntica. De hecho, el recorrido de los conjuntos se plantea del mismo modo, utilizando incluso los mismos procesos predefinidos. Lo único que cambia es el modo en el que se genera el subproceso que expresa el *cuerpo* de dicho bucle. Por tanto:

$$
\begin{array}{rcl}
[\![\backslash\mathbf{loop}\,\underline{v}\,(\underline{S})\,(\underline{P})\,]\!] & \stackrel{\circ}{=} & \left\{\begin{array}{ll}
(\nu\, v_1, end)(\ \mathrm{LFor}\langle\, \underline{S}, v_1, end\,\rangle & \\
\quad \mid \mathrm{Cuerpo}_L\langle\, v_1, end\,\rangle\ ) & \text{si } \underline{S} \text{ lista} \\
(\nu\, v_1, end)(\ \mathrm{LLFor}\langle\, \underline{S}, v_1, end\,\rangle & \\
\quad \mid \mathrm{Cuerpo}_L\langle\, v_1, end\,\rangle\ ) & \text{si } \underline{S} \text{ lista nombrada} \\
(\nu\, v_1, end)(\ \mathrm{NFor}\langle\, |\widetilde{s}|, v_1, end\,\rangle & \\
\quad \mid \mathrm{Cuerpo}_L\langle\, v_1, end\,\rangle\ ) & \text{si } \underline{S} = \widetilde{s} \text{ tupla} \\
(\nu\, v_1, end)(\ \mathrm{IFor}\langle\, inf, sup, v_1, end\,\rangle & \\
\quad \mid \mathrm{Cuerpo}_L\langle\, v_1, end\,\rangle\ ) & \text{si } \underline{S} = inf\,..\,sup
\end{array}\right. \\
[\![\nu]\!]\,\mathrm{Cuerpo_L}(vcb, end) & \triangleq & (\ vcb(\underline{v}).\,(\ \underline{\mathrm{P}}\langle\, \underline{v}\,\rangle \mid \mathrm{Cuerpo}_L\langle\, vcb, end\,\rangle\ ) \\
& & \quad + end.\,\mathbf{0}\ )
\end{array}
$$

Como puede verse, el esquema del bucle **loop** es totalmente idéntico, pero ahora el cuerpo Cuerpo_F ya no se define como una *secuencia* de subprocesos, sino que éstos se van invocando en paralelo. Por supuesto, el recorrido del conjunto sigue siendo secuencial –se debe seguir algún orden–, pero eso es en realidad irrelevante, ya que el orden de terminación de los subprocesos constituye de todos modos una mezcla indeterminista. Ningún orden está, en definitiva, garantizado; y ésta es, precisamente, la característica que define a este tipo de bucle.

En ambos casos, sea un **for** o un **loop**, la traducción del bucle, cuando no se indica la variable de control $\underline{v}$, es idéntica al caso ya descrito. La única diferencia es que el nombre interno que se utiliza en el proceso $\underline{P}$, esto es, en el cuerpo del bucle, se corresponderá ahora con el identificador **now** de $\mathcal{P}i\mathcal{L}ar$; no obstante, esto supone tan sólo una distinción de carácter meramente sintáctico, y semánticamente resulta totalmente irrelevante.

Nota sobre Componentes Recursivos y Paramétricos

La aparición de recursión en el interior de una instancia, como parte de sus restricciones, es perfectamente natural; pero ese es un aspecto que compete exclusivamente al Lenguaje Dinámico, por lo que se remite su explicación a la sección 7.4.3. Sin embargo, sí existe dentro del Lenguaje Estructural un tipo de recursión puramente estática, que se ha de considerar aquí: la definición de componentes recursivos en $\mathcal{P}i\mathcal{L}ar$, según el esquema descrito en la sección 6.4.5.

En realidad, este tipo de definición no tiene dificultad alguna, ya que se maneja exactamente igual que la de cualquier otro componente. Simplemente, el proceso Comp crea la interfaz de la instancia compuesta, y ordena la creación de sus subinstancias; si una de éstas es de su mismo tipo, la estructura es recursiva, pero esto es indiferente a la hora de crear el proceso. Tampoco

afecta a la definición del tipo, pues éste no depende de la estructurta del componente: todos ellos tienen internamente el mismo tipo, *TComp*.

Como pudo verse en la Figura 6.4, la semántica de $\mathcal{P}i\mathcal{L}ar$, como la de cualquier lenguaje similar, no necesita encargarse de controlar la corrección de una definición recursiva: si esta es ilegítima, caerá con facilidad en el caso infinito, con lo que será imposible obtener una traducción completa para el componente mal concebido –debe recordarse que el proceso que crea a un componente no termina hasta que se han creado todas sus instancias: por tanto, en una recursión infinita, simplemente no termina–.

En lo que respecta a los arquetipos paramétricos, la definición resulta igualmente sencilla. Dado que los parámetros pueden tener solamente tipos básicos –primitivos–, en especial numéricos, se pueden manipular semánticamente como un dato más. En términos generales, un parámetro puede ser utilizado directamente como parte de una expresión, y en principio no es necesario almacenarlo como un campo en la estructura de componente; dado que se liga de manera estática, sólo será desarrollado estructuralmente *una* vez, durante la definición de un vector o en la condición de una alternativa. Dicho de otro modo, en principio no hay necesidad alguna de que la infraestructura revele que el arquetipo es paramétrico, una vez que ya ha sido construido.

No obstante, puede ser interesante tener acceso a los parámetros de manera permanente, de modo que pueda ser modificado desde el Lenguaje Dinámico, tal como se hace, por ejemplo, en la sección 8.4 del Capítulo 8. En este sentido, pueden afectar únicamente a las condiciones y operaciones descritas en las restricciones dinámicas; por tanto, el dato correspondiente sí se almacena, pero como un elemento del Lenguaje Dinámico. No afecta, pues, a la definición estructural del arquetipo. En términos del Lenguaje Dinámico, un parámetro se traduce *exactamente igual* que una *variable* del mismo, con la misma semántica y comportamiento que serán descritos en la sección 7.4.2. Es decir, se concibe como un simple dato, adecuadamente tipado, y contenido en una célula de memoria, accesible a todos los procesos de la restricción.

Con esto se da por terminada, definitivamente, la descripción del Lenguaje Estructural. En lo que resta, la complejidad del sistema se concentra, simplemente, en los aspectos relativos a la traducción de una restricción de comportamiento; es esta traducción, precisamente, la que elabora los procesos Proc_N, generados por GenProc y activados por Dispatch (§7.3.4), utilizados implícitamente a lo largo de todo este apartado. Es ésto, precisamente, lo que se corresponde con la semántica del Lenguaje Dinámico.

7.4 Semántica de los Aspectos Dinámicos

Aunque a primera vista pueda parecer sorprendente, la descripción de los aspectos dinámicos resultará mucho más sencilla que la de los estáticos. Esto se debe, por supuesto, a que la infraestructura en la que se desarrollan ya ha sido descrita a lo largo de todo el apartado anterior. Las estructuras de datos son las ya conocidas; no es necesario aportar nuevas estructuras, sino simplemente plantear la correspondencia de los términos del lenguaje. Por tanto, tan sólo será necesario indicar un *esquema de traducción*, que indique la especificación que describe la función y comportamiento de cada uno de los elementos sintácticos del Lenguaje Dinámico.

7.4.1 Sobre la Notación Semántica

Como ya se dijo en la introducción, en ningún momento se proporciona una traducción completa de cada estructura del lenguaje, que conforme una semántica totalmente definida. Sin embargo, cada traducción se realizará de la manera más precisa posible, y se indica con una notación aproximada que pretende, ante todo, ser clara y concisa.

Se utilizan, pues, las siguientes convenciones:

- Cuando una estructura o notación es rodeada entre dobles corchetes, esto es, indicada como $[\![A]\!]$, se está haciendo referencia al modo en que se define su traducción.

- Un elemento aparece subrayado como $\underline{a}$, en general, para indicar que ya ha sido desarrollado; es decir, si es un nombre cualificado, o viene contenido en una variable, se asume que en este punto la traducción ya proporciona el nombre correcto.

- Se asume la existencia de una función `cad(x)`, que permite a la traducción referirse, no al nombre-π de un elemento, sino al nombre –cadena de caracteres o etiqueta– que se utiliza desde un punto de vista sintáctico.

- El símbolo ▶ indica que el valor indicado a continuación es "el resultado" de la definición semántica, en el caso de que se esté identificando a un nombre. Seguido de un paréntesis, indica simplemente que el ámbito de una definición se prolonga.

- Finalmente, por brevedad, se asumirá en general que se dispone directamente del nombre del componente (avatar) implicado en una sentencia, con el símbolo $\underline{C}$, y del mismo modo, el metacomponente relevante se referencia con el símbolo $\underline{M}$. Obviamente, ambos se deducen directamente de la "torre" de parámetros recibida por todo proceso (*Tower*).

En cuanto al resto de la notación, como se ha venido haciendo, utiliza los símbolos del cálculo-$\hat{\pi}$ y su teoría de tipos, tal como fueron definidos en el Apéndice C.

7.4.2 Restricciones

Como ya se ha dicho en la descripción sintáctica (§6.7), una restricción consta básicamente de tres elementos: canales, variables y procesos. Los últimos se describen a partir del apartado siguiente, mientras que en éste se resumen brevemente los dos primeros.

Canales

Se definen, simplemente, como un canal cualquiera del cálculo. Su único aspecto particular es que se declaran como globales, en el ámbito exterior del primer proceso, y por tanto pueden ser utilizados por la restricción completa. No necesitan, siquiera, de una estructura dual subyacente, ya que se usan *exactamente igual* que en el propio cálculo-π.

Variables

Se usan igual que los canales, pero tienen memoria; esto se podría proporcionar fácilmente mediante una restricción $\mathcal{P}i\mathcal{L}ar$, en realidad, pero se proporciona, únicamente, como "azúcar sintáctico". En cualquier caso, se pueden definir de manera trivial como:

$$[\![\, \mathbf{var}\ v : \underline{T} \,]\!] \ \stackrel{\circ}{=}\ (\nu\, \widehat{v})\mathrm{NewCell}\langle\, \widehat{v}\,\rangle \ \blacktriangleright\ \widehat{v}$$

Donde se crean un par de canales y se los envía al proceso NewCell; éste sólo realiza una inicialización, y a continuación se remite a una definición clásica de célula de memoria, dada por el proceso Cell.

$$\begin{array}{rcl}
\mathrm{NewCell} & : & (\alpha;\ (Par\, \alpha\, \alpha)) \\
\mathrm{NewCell}(\widehat{v}) & \triangleq & v_i(x).\, \mathrm{Cell}\langle\, \widehat{v}, x\,\rangle \\
\mathrm{Cell} & : & (\alpha;\ (Par\, \alpha\, \alpha),\ \alpha) \\
\mathrm{Cell}(\widehat{v}, d) & \triangleq & (\ v_i(x).\, \mathrm{Cell}\langle\, \widehat{v}, x\,\rangle + \overline{v_o}\langle\, d\,\rangle.\, \mathrm{Cell}\langle\, \widehat{v}, d\,\rangle\ )
\end{array}$$

De modo que, según se emita o reciba por un canal u otro, se obtiene el dato contenido en la célula, o se sustituye. Esto implica que la traducción de la sintaxis –que utiliza el mismo mecanismo de emisión y recepción definido para puertos y canales– se define como:

$$\begin{array}{rcll}
[\![\, v!(\underline{x}) \,]\!] & \stackrel{\circ}{=} & \overline{v_i}\langle\, \underline{x}\,\rangle & \text{si } v \text{ variable } (\widehat{v}) \\
[\![\, v?(\underline{x}) \,]\!] & \stackrel{\circ}{=} & v_o(\underline{x}) & \text{si } v \text{ variable } (\widehat{v})
\end{array}$$

Lo que obviamente realiza el comportamiento descrito.

7.4.3 Procesos: Constructores

Los constructores básicos de $\mathcal{P}i\mathcal{L}ar$ se traducen literalmente a su forma en cálculo-$\hat{\pi}$: esto se aplica tanto a la secuencia, como a la composición paralela, como a la selección. El único aspecto a resaltar en este sentido es que, en cumplimiento de la Regla de Ámbito (§6.8.1), todo canal de un proceso se define como privado; se asume, por tanto, la existencia de todos los términos de restricción (ν) necesarios al principio de cada proceso.

Hay una sutil diferencia entre el comportamiento de la replicación y la recursión en el cálculo-π y en $\mathcal{P}i\mathcal{L}ar$. Dado que es un aspecto determinante, en lo que respecta a la terminación de un proceso, ésta será discutida en mayor detalle a continuación.

Replicación

Como ya se ha dicho, existe total libertad en cuanto a la estructura de los procesos que constituyen una restricción. Sin embargo, su naturaleza esencial será de tipo reactivo: cada interacción sobre un puerto obtendrá normalmente una respuesta, mientras el componente exista. Por tanto, aunque es posible definir procesos finitos, éstos serán poco habituales. En lugar de eso, abundarán los procesos recursivos y, muy especialmente, los replicados. Éstos últimos son

particularmente adecuados en este contexto, ya que permiten describir un protocolo como una secuencia cerrada de pasos, sin necesidad de mayores consideraciones; pero al mismo tiempo suponen un suministro infinito de los mismos, ubicados además en un contexto concurrente. Su ventaja principal sobre el típico esquema recursivo es, precisamente, que no se implica secuencialidad alguna entre invocaciones distintas, ya que todas las copias del proceso se generan en paralelo. Esto facilita enormemente la expresión de los conflictos debidos a interferencias entre interacciones concurrentes, que se constituyen como uno de los problemas básicos que se pretende detectar mediante la descripción arquitectónica de un sistema software.

Como ya se explicó (§6.8.2), el constructor de replicación (**rep**) se concibe como el análogo en $\mathcal{P}i\mathcal{L}ar$ de su homónimo en el cálculo-π. En líneas generales son de hecho idénticos, y por tanto la traducción ingenua $\backslash\mathbf{rep}(A) = !A$, donde A representa a cualquier abstracción procesual, se puede considerar como correcta a efectos de una especificación normal.

Sin embargo, esto no es exacto. La replicación supone una repetición explícitamente infinita, mientras que una restricción siempre va ligada a la vida de la instancia –o más exactamente, la reificación– en la que se define. Por tanto, una restricción replicada no puede implicar un proceso infinito: cuando la instancia es borrada, o la reificación destruida, todas sus restricciones, replicadas o no, deben dejar de interactuar y morir con él.

Como se ha visto a lo largo de la descripción de los elementos del Lenguaje Estático, y en particular en la sección relativa a la generación de un proceso –véanse las secciones 7.2.4 y 7.3.4, en particular la página 253 y subsiguientes–, todo proceso tiene asociada una señal de terminación única, proporcionada como parámetro por el generador, que se designará abreviadamente como *die*. Por tanto, un esquema repetitivo, sea recursivo o replicado, se debe plantear en cada vuelta como una selección (una suma) entre su cuerpo y la señal *die*.

En el caso de la replicación, la expresión más clara y compacta de su traducción semántica es, curiosamente, su definición como un punto fijo, con la forma siguiente:

$$[\![\backslash\mathbf{rep}\,(A)\,]\!] \;\;\doteq\;\; \mathbf{fix}\,X.\,(\;(A \mid X) + die\;)$$

En este caso, A vuelve a ser cualquier agente o abstracción de proceso, que constituye la parte de la restricción que se infiere inmediatamente de su definición sintáctica. El punto fijo define una selección entre la señal de terminación y este agente; en el caso de elegirse este último, se invocará en paralelo con una nueva ocurrencia del punto fijo, simbolizada por el agente ligado X. Por supuesto, esta ocurrencia se plantea de nuevo como una selección; esto garantiza que la eventual terminación del proceso se produce efectivamente como respuesta a la señal *die*.

Esta definición es poco habitual, y tal vez no deja una idea clara de cuál puede ser la estructura que adopta realmente un proceso de este tipo. Se ha dicho ya que por cada restricción descrita sintácticamente existe una definición de proceso Proc_N, que declara como argumentos a la torre reflexiva y señal de terminación correspondientes. Cuando un proceso es replicado, entonces, esta definición adoptará cualquiera de las dos formas siguientes:

$$\begin{aligned}
\mathrm{Proc_N}(Tr, die) &\triangleq (\;(\;A \mid \mathrm{Proc}_N\langle\, Tr, die\,\rangle\;) + die\;)\\
\mathrm{Proc_N}(Tr, die) &\triangleq (\nu\, a)(\;\overline{a} \mid !(\;die + a.\,(A \mid \overline{a}))\;)
\end{aligned}$$

Ambas son totalmente equivalentes: la primera es obviamente recursiva y bastante intuitiva; se describe de manera similar a como se ha hecho para el caso del punto fijo. La segunda

es bastante más exótica e inusual, y se plantea de modo que utiliza únicamente el operador de replicación. De este modo, se expresa explícitamente cuál es la verdadera relación entre el constructor del cálculo-π y su análogo en $\mathcal{P}i\mathcal{L}ar$.

Recursión

Los mismos comentarios pueden aplicarse a cualquier definición de restricción recursiva. De nuevo, puede considerarse, en líneas generales, que la recursión se trata de manera idéntica en $\mathcal{P}i\mathcal{L}ar$ y en cálculo-π. De nuevo, se plantea el mismo problema con la señal de terminación del proceso. La única diferencia significativa es que en este caso las distintas copias no se disparan en paralelo, sino secuencialmente. Precisamente por eso, la traducción resulta mucho más obvia. Basta con suponer que la selección es el primer paso del proceso recursivo, de manera similar a como se hace en el esquema de los bucles de esta semántica (véase el Apéndice D).

De nuevo, la traducción en forma de punto fijo es la más sencilla y compacta:

$$[\![\, \underline{P} \ \mathbf{def}{=}\ A.\underline{P} \,]\!] \ \stackrel{\circ}{=}\ \mathbf{fix}\, X.\, (\ A.X + die\ )$$

Su estructura es totalmente intuitiva, y resulta además similar al caso anterior. Del mismo modo, la correspondiente definición del proceso genérico Proc_N es mucho más parecida:

$$\mathrm{Proc_N}(Tr, die) \ \stackrel{\triangle}{=}\ (\ A.\mathrm{Proc}_N\langle\, Tr, die\,\rangle\ + die.\mathbf{0}\ )$$

Con este punto queda completa la descripción semántica de los constructores no finitos del Lenguaje Dinámico de $\mathcal{P}i\mathcal{L}ar$. El caso finito, por su parte, puede considerarse trivial. Todo esto se aplica a todas las restricciones, de aquí en adelante. No obstante, como ya se ha dicho, en general puede considerarse que se trabaja con unas estructuras prácticamente idénticas a las de un álgebra de procesos normal. En realidad, y dado que lo dicho se refiere solamente a la destrucción de una instancia o reificación, puede obviarse en el caso habitual; por tanto, no se hará hincapié en este aspecto salvo cuando sea estrictamente necesario.

7.4.4 Primitivas de Comunicación

Dado que las primitivas de comunicación básicas son las propias *acciones* atómicas del cálculo, cabe esperar que su traducción semántica sea muy sencilla. En realidad, las cosas no son tan simples, como ya se explicó en la sección 7.2.3, pero en cualquier caso no serán tampoco demasiado complicadas, pues vienen inducidas en gran medida por la estructura con la que se ha dotado a un puerto, que se ha elaborado desde esta perspectiva.

En primer lugar, se considera una *emisión*: se distingue entre la realizada sobre un puerto o sobre un canal interno, aunque ambas serán muy similares.

$$[\![\, \underline{a}!(\widetilde{\underline{x}}) \,]\!] \ \stackrel{\circ}{=}\ \begin{cases} \overline{\underline{a}\dagger I_o}\langle\, \widetilde{\underline{x}}\,\rangle & \text{si } \underline{a} \text{ puerto} \\ \overline{\underline{a}}\langle\, \widetilde{\underline{x}}\,\rangle & \text{si } \underline{a} \text{ canal interno} \end{cases}$$

En el segundo caso es, por supuesto, trivial. En el primero, simplemente se envía al canal correspondiente (**IO**) de un puerto.

Visto esto, cabría esperar que la recepción siguiese un esquema similar, a saber:

$$[\![\, \underline{a}?(\widetilde{\underline{x}}) \,]\!] \quad \not\Rightarrow \quad \begin{cases} \underline{a}\dagger I_i(\widetilde{\underline{x}}) & \text{si } \underline{a} \text{ puerto} \\ \underline{a}(\widetilde{\underline{x}}) & \text{si } \underline{a} \text{ canal interno} \end{cases}$$

Sin embargo, esto no es cierto en absoluto, y ya lo sabíamos, pues en la definición de puerto (§7.3.1) ya quedó claro que el canal de entrada interna ($a\dagger I_i$) no está en realidad ligado a ningún proceso del puerto, y sólo se proporciona por conveniencia: de este modo, se dispone siempre de un canal del tipo adecuado en el que recibir el mensaje:

$$[\![\, \underline{a}?(\widetilde{\underline{x}}) \,]\!] \quad \doteq \quad \begin{cases} \overline{\underline{a}\dagger W_o}\langle\, \underline{a}\dagger I_i \,\rangle.\, \underline{a}\dagger I_i(\widetilde{\underline{x}}) & \text{si } \underline{a} \text{ puerto} \\ \underline{a}(\widetilde{\underline{x}}) & \text{si } \underline{a} \text{ canal interno} \end{cases}$$

No obstante, se podría haber utilizado cualquier puerto de este tipo que se hubiera indicado como parámetro sobre $\underline{a}\dagger W_o$. Como se ha visto en la definición de un puerto, esta recepción consume el dato contenido en la célula CellPort.

Sin embargo, esta definición sólo tiene sentido una vez que se considera la traducción de una lectura-**when**, que precisamente por este motivo sólo puede aplicarse a puertos.

$$[\![\backslash\mathbf{when}\, \underline{a}?(\widetilde{\underline{x}})\, (\underline{P})]\!] \quad \doteq \quad \overline{\underline{a}\dagger W_i}\langle\, \underline{a}\dagger I_i \,\rangle.\, \underline{a}\dagger I_i(\widetilde{\underline{x}}).\, \underline{P} \qquad \text{si } \underline{a} \text{ puerto}$$

Al igual que una recepción convencional, el **when** hace una espera pasiva, hasta que existe un dato en la célula. Su única diferencia es que, posteriormente, este dato no se consume.

En cuanto al **tau**, realmente no hace nada; de hecho, eso es lo que se supone que hace una transición τ. Sin embargo, cuando recibe parámetros, está indicando –de manera meramente informativa– cuáles son los nombres que están siendo afectados. Esto es irrelevante si son nombres ya ligados; pero si son nuevos, han de crearse por mera coherencia. Se define, entonces:

$$\begin{aligned} [\![\, \mathbf{tau}]\!] \quad &\doteq \quad \tau \\ [\![\, \mathbf{tau}(\widetilde{x})]\!] \quad &\doteq \quad \tau.\, (\nu\, \widetilde{x_n}) \qquad \text{donde } \widetilde{x_n} = \{\widetilde{x}\} - \mathrm{bn}(\widetilde{x}) \end{aligned}$$

Por su parte, las dos sentencias especiales, aunque fueron introducidas para ejercer su influencia desde el nivel superior, se definen de manera muy sencilla con base en el puerto, por lo que resulta más lógico incluirlas aquí. En la definición sintáctica, la primera se indicó como un operador dinámico, y la segunda como parte de la extensión propia de un envolvente. Sin embargo, se trata ante todo, en ambos casos, de primitivas de comunicación.

El **shift** simplemente introduce un mensaje en la célula de un puerto, a través del canal especial (**MI**) definido a tal efecto.

$$[\![\backslash\mathbf{shift}\, \underline{\mathrm{P}}(\widetilde{\underline{x}})]\!] \quad \doteq \quad \overline{\underline{P}\dagger M_i}\langle\, \widetilde{\underline{x}} \,\rangle$$

De manera análoga, un **catch** intercepta una salida, mediante el mecanismo proporcionado en un puerto a través del canal **MO**, en una de las alternativas del proceso OutPort:

$$[\![\backslash\mathbf{catch}\, \underline{\mathrm{P}}(\widetilde{\underline{x}})]\!] \quad \doteq \quad (\nu\, r)\overline{\underline{P}\dagger M_o}\langle\, r \,\rangle.\, r(\widetilde{y}) \;\blacktriangleright\; \widetilde{\underline{x}} = \widetilde{y}$$

Como puede verse, la traducción, si bien no es directa, tampoco reviste mayor dificultad.

7.4.5 Primitivas Dinámicas

Hay una clara diferencia entre **new** y **del**, por una parte, y **alias** y **hide**, por otro. Las dos primeras son más complejas, porque pueden aplicarse, y con distinto comportamiento, a puertos, componentes (instancias) y enlaces. En cambio las otras dos son más sencillas, porque sólo se aplican a puertos. Por tanto, se comenzará describiendo primero a estas últimas.

El **hide** es casi trivial: simplemente supone que un nombre ya no se encuentra cuando se busca por su etiqueta. Sin embargo, ha de recordarse que no se destruye ninguno de sus enlaces, ni su comportamiento –esto puede hacerse de manera explícita en $\mathcal{P}i\mathcal{L}ar$, si es preciso, así que la semántica no necesita encargarse de ello–. Por tanto, basta con proporcionarle un nombre "imposible", de modo que no pueda ser localizado. Para ello se utiliza el artificio de añadirle una cifra al principio, aunque cualquier otro método hubiera sido igualmente válido:

$$
\begin{array}{rcl}
[\![\backslash\mathbf{hide}\ \underline{x}\,]\!] & \mathrel{\hat{=}} & \underline{C}\dagger P(lp).\,\mathbf{if}\ (\mathtt{lein}\ \mathtt{cad}(\underline{x})\ lp)\ \mathbf{then} \\
& & \quad (\nu\, rx)(\ \mathrm{Encode}\langle\, \mathtt{cad}(\underline{x}), rx\,\rangle \mid rx(xc). \\
& & \quad\ \ \overline{\underline{C}\dagger P}\langle\, (\mathtt{ncons}\ xc\ (\mathtt{ndot}\ \mathtt{cad}(\underline{x})\ lp)\ (\mathtt{lera}\ \mathtt{cad}(\underline{x})\ lp)\,\rangle\ ) \\
& & \mathbf{else}\ \overline{\underline{C}\dagger P}\langle\, lp\,\rangle
\end{array}
$$

La forma de modificar el nombre se expresa en el proceso funcional Encode. Es obvio que puede cambiarse el método de codificación sin más que modificar este proceso.

$$
\begin{array}{rcl}
\mathrm{Encode} & : & (\mathit{Label},\ \uparrow\mathit{Label}) \\
\mathrm{Encode}(L, r) & \triangleq & \overline{r}\langle\, \mathtt{"1"}\frown L\,\rangle
\end{array}
$$

Como se sabe, el **alias** es más complejo, ya que tiene tres posibilidades: añadir un nuevo nombre a la lista, conociendo o no la etiqueta a utilizar; mostrar de nuevo un nombre que estaba oculto; e incluso cambiar el nombre, si éste se proporciona explícitamente con un sufijo **as**. En resumen, se trata simplemente de:

$$
\begin{array}{rcl}
[\![\backslash\mathbf{alias}\ \underline{x}\,]\!] & \mathrel{\hat{=}} & \underline{C}\dagger P(lp).\,\mathbf{if}\ (\mathtt{lein}\ \mathtt{cad}(\underline{x})\ lp)\ \mathbf{then}\ \overline{\underline{C}\dagger P}\langle\, lp\,\rangle \\
& & \quad \mathbf{else}\ (\nu\, rx)(\ \mathrm{Encode}\langle\, \mathtt{cad}(\underline{x}), rx\,\rangle \mid rx(xc). \\
& & \quad\ \ \mathbf{if}\ (\mathtt{lein}\ xc\ lp)\ \mathbf{then}\ \overline{\underline{C}\dagger P}\langle\, (\mathtt{ncons}\ \mathtt{cad}(\underline{x})\ \underline{x}\ (\mathtt{lera}\ xc\ lp)\,\rangle \\
& & \quad\ \ \mathbf{else}\ \overline{\underline{C}\dagger P}\langle\, (\mathtt{ncons}\ \mathtt{cad}(\underline{x})\ \underline{x}\ lp)\,\rangle\ )
\end{array}
$$

$$
\begin{array}{rcl}
[\![\backslash\mathbf{alias}\ \underline{x}\ \mathbf{as}\ L]\!] & \mathrel{\hat{=}} & \underline{C}\dagger P(lp).\,\mathbf{if}\ (\mathtt{lein}\ L\ lp)\ \mathbf{then}\ \overline{\underline{C}\dagger P}\langle\, lp\,\rangle \\
& & \quad \mathbf{else}\ (\nu\, rx)(\ \mathrm{Encode}\langle\, L, rx\,\rangle \mid rx(Lc). \\
& & \quad\ \ \mathbf{if}\ (\mathtt{lein}\ Lc\ lp)\ \mathbf{then}\ \overline{\underline{C}\dagger P}\langle\, (\mathtt{ncons}\ L\ \underline{x}\ (\mathtt{lera}\ Lc\ lp)\,\rangle \\
& & \quad\ \ \mathbf{else}\ \overline{\underline{C}\dagger P}\langle\, (\mathtt{ncons}\ L\ \underline{x}\ lp)\,\rangle\ )
\end{array}
$$

Es decir, se comprueba si un nombre existe en la interfaz –codificado–, se elimina de ésta, y se introduce de nuevo con el nombre adecuado. Obsérvese que se evita explícitamente definir un método de decodificación, de modo que sólo se depende del definido en la función Encode.

Por su parte, la creación (**new**) es sencilla, pues es idéntica al caso estático. El único aspecto complejo hace referencia al hecho de que los elementos se crean dentro de un contexto, luego han de añadirse a la estructura en el momento de su creación.

Así, en el caso de la creación de un puerto:

$$\begin{aligned}[\![\backslash\mathbf{new}\ \underline{p} : \mathbf{Port}\,]\!] \ \doteq\ & (\nu\,\underline{p})(\ \underline{C}\dagger P(lp).\,\mathrm{Port}\langle\,\underline{p}\,\rangle \\ & \mid\ \underline{p}(p_n).\,\overline{\underline{C}\dagger P}\langle\,(\mathsf{ncons}\ \mathsf{cad}(\underline{p})\ (p_n\dagger S)\ lp)\,\rangle\ \blacktriangleright\ )\end{aligned}$$

Por su parte, la creación de un componente es totalmente análoga:

$$\begin{aligned}[\![\backslash\mathbf{new}\ \underline{c} : \underline{T}\,]\!] \ \doteq\ & (\nu\,\underline{c})(\ \underline{C}\dagger I(ls).\,\mathrm{NewComp}\langle\,\underline{T},\underline{c}\,\rangle \\ & \mid\ \underline{c}(c_n).\,\overline{\underline{C}\dagger I}\langle\,(\mathsf{ncons}\ \mathsf{cad}(\underline{c})\ (c_n\dagger S)\ ls)\,\rangle\ \blacktriangleright\ )\end{aligned}$$

El único caso ligeramente más complejo es el de las conexiones: por una parte, es prácticamente idéntico al proceso NewBind, ya conocido; sin embargo, puede remitirse directamente al proceso Bind, que lo hace más sencillo.

$$\begin{aligned}[\![\backslash\mathbf{new}\ \underline{b} : \underline{\mathrm{T}}(\widetilde{\underline{c}},\widetilde{\underline{p}})\,]\!] \ \doteq\ & (\nu\,r_L,\underline{b})(\ \mathrm{LisVec}\langle\,\underline{T}\ddagger B, r_L\,\rangle \mid r_L(\widetilde{tag}). \\ & \underline{C}\dagger B(lb).\,(\ \mathrm{Bind}\langle\,\underline{T},\widetilde{c},\widetilde{p},\widetilde{tag},\underline{b}\,\rangle \mid \underline{b}(b_n). \\ & \overline{\underline{C}\dagger B}\langle\,(\mathsf{ncons}\ \mathsf{cad}(\underline{b})\ (b_n\dagger S)\ lb)\,\rangle\ )\ \blacktriangleright\ )\end{aligned}$$

Esto es aplicable, por supuesto, a enlaces simples y jerárquicos, que se definen como un caso particular de éstos, y no requieren por tanto una definición especial.

Por su parte, la destrucción (**del**) es más compleja, ya que suele extenderse a varios elementos; esto ocurría ya en la construcción, pero en este caso los procesos ya se encargaban de ello.

El borrado de un puerto es sencillo, ya que no se intentan destruir los enlaces asociados, que por tanto quedan *al aire*. No obstante, esto puede hacerse explícitamente desde $\mathcal{P}i\mathcal{L}ar$.

$$\begin{aligned}[\![\backslash\mathbf{del}\,\underline{P}\,]\!] \ \doteq\ & \underline{C}\dagger P(lp).\,\mathbf{if}\ (\mathtt{lein}\ \mathsf{cad}(\underline{P})\ lp)\ \mathbf{then} \\ & \quad (\ \overline{\underline{C}\dagger P}\langle\,(\mathtt{lera}\ \mathsf{cad}(\underline{P})\ lp)\,\rangle \mid (lp \bullet \mathsf{cad}(\underline{P}))(p_d).\,\overline{p_d\dagger D}.\,\mathbf{0}\ ) \\ & \mathbf{else}\ \mathbf{0}\end{aligned}$$

En cambio, borrar un componente supone borrar todos sus puertos, así como todas sus instancias internas, enlaces internos y reificaciones, cuando estos elementos existen. Afortunadamente, la definición de un componente definía el subproceso SubKill, que propaga automáticamente esta orden, de manera que ésta se propaga en cascada, e incluso en el orden correcto –primero los enlaces, y luego el resto–. Entonces, basta con:

$$\begin{aligned}[\![\backslash\mathbf{del}\,\underline{s}\,]\!] \ \doteq\ & \underline{C}\dagger I(ls).\,\mathbf{if}\ (\mathtt{lein}\ \mathsf{cad}(\underline{s})\ ls)\ \mathbf{then} \\ & \quad (\overline{\underline{C}\dagger I}\langle\,(\mathtt{lera}\ \mathsf{cad}(\underline{s})\ ls)\,\rangle \mid (ls \bullet \mathsf{cad}(\underline{s}))(s_d).\,\overline{s_d\dagger D}.\,\mathbf{0}\ ) \\ & \mathbf{else}\ \mathbf{0}\end{aligned}$$

Pero sin duda, el comportamiento más complejo es el de los enlaces, que se ven obligados a *actualizar la información reflexiva* de los puertos ligados, con el fin de que la instrucción **bound** funcione correctamente. Entonces:

$$
\begin{array}{rcl}
[\![\backslash\mathbf{del}\,\underline{L}]\!] & \triangleq & \underline{C}\dagger B(lb).\,\mathbf{if}\ (\mathtt{lein}\ \mathtt{cad}(\underline{L})\ lb)\ \mathbf{then} \\
& & \quad (\,\overline{\underline{C}\dagger B}\langle\,(\mathtt{lera}\ \mathtt{cad}(\underline{L})\ lb)\,\rangle \mid \mathrm{KillBind}\langle\, lb \bullet \mathtt{cad}(\underline{L})\,\rangle\,) \\
& & \mathbf{else}\ \mathbf{0}
\end{array}
$$

Donde el proceso KillBind se despliega como puede verse a continuación:

$$
\begin{array}{rcl}
\mathrm{KillBind} & : & (\uparrow TBind) \\
\mathrm{KillBind}(kb) & \triangleq & kb(b).\,b\dagger P(lp).\,\mathrm{LimRef}\langle\, lp, b\dagger S\,\rangle.\,\overline{b\dagger D}.\,\mathbf{0} \\
\mathrm{LimRef} & : & ((NList\ \uparrow DPort),\ \uparrow TBind) \\
\mathrm{LimRef}(lp, bin) & \triangleq & (\nu\, v_1, e_1)(\ \mathrm{NLFor}\langle\, lp, v_1, e_1\,\rangle \\
& & \quad \mid\ (\nu\, rc)(\ \mathrm{LCom}_F\langle\, v_1, e_1, (\mathtt{nil}), rc\,\rangle \\
& & \qquad \mid\ (\nu\, v_2, e_2)(\ rc(lc).\,(\ \mathrm{LFor}\langle\, lc, v_2, e_2\,\rangle \\
& & \qquad\quad \mid\ \mathrm{LimCom}_F\langle\, bin, v_2, e_2\,\rangle\ )))) \\
\mathrm{LCom_F} & : & (\uparrow\uparrow DPort,\ Sync,\ (List\ \uparrow TComp),\ \uparrow(List\ \uparrow TComp)) \\
\mathrm{LCom_F}(vcb, end, lis, r) & \triangleq & (\ vcb(kp).\,kp(p).\,p\dagger B(lc_p). \\
& & \quad \overline{p\dagger B}\langle\, lc_p\,\rangle.\,(\nu\, rc)(\ \mathrm{ApNDup}\langle\, lc_p, lis, rc\,\rangle \\
& & \qquad \mid\ rc(lc_n).\,\mathrm{LCom}_F\langle\, vcb, end, lc_n, r\,\rangle\ ) \\
& & +\ end.\,\overline{r}\langle\, lis\,\rangle.\,\mathbf{0}\ ) \\
\mathrm{LimCom_F} & : & (\uparrow TBind,\ \uparrow\uparrow TComp,\ Sync) \\
\mathrm{LimCom_F}(bin, vcb, end) & \triangleq & (\ vcb(kc).\,kc(c).\,c\dagger P(lp_c). \\
& & \quad \overline{c\dagger P}\langle\, lp_c\,\rangle.\,(\nu\, v_p, e_p)(\ \mathrm{NLFor}\langle\, lp_c, v_p, e_p\,\rangle \\
& & \qquad \mid\ \mathrm{LRef}_F\langle\, bin, v_p, e_p\,\rangle\ ) \\
& & +\ end.\,\mathbf{0}\ ) \\
\mathrm{LRef_F} & : & (\uparrow TBind,\ \uparrow\uparrow DPort,\ Sync) \\
\mathrm{LRef_F}(bin, vcb, end) & \triangleq & (\ vcb(kp).\,kp(p).\,p\dagger B(lb_o), \\
& & \quad (\nu\, rb)(\ \mathrm{LimLis}\langle\, bin, lb_o, rb\,\rangle \mid rb(lb_n).\,\overline{p\dagger B}\langle\, lb_n\,\rangle. \\
& & \qquad \mathrm{LRef}_F\langle\, bin, vcb, end\,\rangle\ ) \\
& & +\ end.\,\mathbf{0}\ ) \\
\mathrm{LimLis} & : & (\alpha;\ \uparrow\alpha,\ (List\,\alpha),\ \uparrow(List\,\alpha)) \\
\mathrm{LimLis}(ele, lis, r) & \triangleq & \mathbf{if}\ (\mathtt{nein}\ ele\ lis)\ \mathbf{then}\ \overline{r}\langle\,(\mathtt{nera}\ ele\ lis)\,\rangle \\
& & \quad \mathbf{else}\ \overline{r}\langle\, lis\,\rangle
\end{array}
$$

Básicamente, se puede resumir como sigue: se da la orden de borrar un enlace, y por tanto de eliminar toda referencia a él. Entonces, se toma cada uno de sus puertos: se obtiene el compo-

nente ligado a este puerto, y a partir de él se determina *cuál* es el puerto de este componente al que la conexión hacía referencia. Se elimina entonces la referencia a esta conexión, y el proceso finaliza. Una vez eliminada toda referencia, la conexión se elimina.

Finalmente, el **del** también puede aplicarse a reificaciones, pero ese aspecto se examinará en la sección dedicada a primitivas reflexivas, esto es, a continuación.

7.4.6 Primitivas Reflexivas

Básicamente, se trata de un sistema para *encontrar nombres*, en el caso de los operandos, y de una serie de operaciones sobre reificaciones, en el caso de los operadores. De nuevo, el proceso es grandemente facilitado por el hecho de disponer de una infraestructura adecuada: en este caso, la *torre* de parámetros que recibe todo proceso.

Operandos Reflexivos

Una vez que se sabe que todo proceso recibe una torre reflexiva con las estructuras del metacamino empleado, la semántica de de **avatar** y **self** resulta prácticamente trivial, ya que simplemente se trata de recorrer la torre, que es, en definitiva, una lista.

De este modo, el avatar de enésimo nivel es el enésimo elemento de esa lista; típicamente, el avatar considerado es la cabeza de la lista; el metacomponente de mayor nivel, al que corresponde el **self**, será siempre la *cola* de la lista. Por tanto, el enésimo **avatar** y el **self** se definen como se ve a continuación, asumiendo en este caso que $\underline{T}$ representa a la torre de parámetros.

$$[\![\, \mathbf{avatar}_N \,]\!] \triangleq \blacktriangleright\ (\mathtt{nth}\ ((\mathtt{length}\ \underline{T}) - N)\ \underline{T})$$

$$[\![\, \mathbf{self} \,]\!] \triangleq (\nu\, r)\mathrm{Tail}\langle\, \underline{T}, r\,\rangle\ \blacktriangleright\ r$$

$$\begin{array}{rcl} \mathrm{Tail} & : & (\,\sharp\alpha;\ (\mathit{List}\,\alpha),\ \uparrow\alpha\,) \\ \mathrm{Tail}(l, r) & \triangleq & \overline{r}\langle\, (\mathtt{nth}\ (\mathtt{length}\ l)\ l)\,\rangle \end{array}$$

Más interesante es la estructura de los operandos que obtienen los conjuntos asociados; así, en primer lugar, el **avatarSet**, con sus dos sintaxis:

$$\begin{array}{rcl} [\![\, \mathbf{avatarSet} \,]\!] & \mathrel{\mathring{=}} & (\nu\, v_r, e_r, r)(\ \underline{M}\dagger R(avs).\, \overline{\underline{M}\dagger R}\langle\, avs\,\rangle. \\ & & \quad \mathrm{LFor}\langle\, avs, v_r, e_r\,\rangle \mid \mathrm{LAva}\langle\, (\mathtt{nil}), v_r, e_r, r\,\rangle \\ & & \quad \mid r(S_a)\ \blacktriangleright\ S_a\ \blacktriangleright\ ) \end{array}$$

$$\begin{array}{rcll} [\![\, \mathbf{avatarSet}(\underline{c}) \,]\!] & \mathrel{\mathring{=}} & (\nu\, v_r, e_r, r)(\ \underline{c}\dagger R(avs).\, \overline{\underline{c}\dagger R}\langle\, avs\,\rangle. & \text{si } \mathfrak{H}(\underline{c}) \geq 1 \\ & & \quad \mathrm{LFor}\langle\, avs, v_r, e_r\,\rangle \mid \mathrm{LAva}\langle\, (\mathtt{nil}), v_r, e_r, r\,\rangle & \\ & & \quad \mid r(S_a)\ \blacktriangleright\ S_a\ \blacktriangleright\ ) & \end{array}$$

Es decir, se examina el conjunto de reificaciones del metacomponente, y se extrae de éste el nombre de todos los avatares. Si se especifica un componente, ha de ser un elemento de –al menos– el primer nivel meta. Se utiliza, además, la definición auxiliar siguiente:

$$\begin{array}{rcl} \mathrm{LAva} & : & (\alpha;\, (\mathit{List}\,\alpha),\, \uparrow(\mathit{TReif}\,\alpha),\, \mathit{Sync},\, \uparrow(\mathit{List}\,\alpha)) \\ \mathrm{LAva}(l, vcb, end, r) & \triangleq & (\ vcb(rei).\,\mathrm{LAva}\langle\,(\mathtt{cons}\ (rei{\dagger}A)\ l), vcb, end, r\,\rangle \\ & & \quad + \mathit{end}.\,\overline{r}\langle\,l\,\rangle.\,\mathbf{0}\ ) \end{array}$$

Obsérvese que aquí se utiliza la meta-altura, no el $\ni$, porque lo que se está indicando es una condición general de la semántica, no un paso del proceso de traducción. Obviamente, si la $\mathfrak{H}$ es cero, no se puede obtener el conjunto de avatares, porque no existe. Por otra parte, dada la estructura de $\mathcal{P}i\mathcal{L}ar$, esta condición es cierta *siempre*, porque las definiciones se hacen siempre a nivel de arquetipo, y éste está, por tanto, en un meta-nivel.

De este modo se obtiene únicamente el conjunto de avatares de primer nivel, esto es, los inmediatos. Para profundizar más, se ha de descender en la petición, cualificando el parámetro de **avatarSet** como la correspondiente secuencia de palabras **avatar**.

El caso de un **portSet** es mucho más sencillo, y de hecho resulta obvio.

$$\begin{array}{rcl} [\![\,\mathbf{portSet}\,]\!] & \doteq & \underline{C}{\dagger}P(\mathit{lport}).\,\overline{\underline{C}{\dagger}P}\langle\,\mathit{lport}\,\rangle \ \blacktriangleright\ \mathit{lport} \\ [\![\,\mathbf{portSet}(\underline{c})\,]\!] & \doteq & \underline{c}{\dagger}P(\mathit{lport}).\,\overline{\underline{c}{\dagger}P}\langle\,\mathit{lport}\,\rangle \ \blacktriangleright\ \mathit{lport} \quad \text{si } \mathfrak{H}(\underline{c}) \geq 1 \end{array}$$

Aunque es el meta quien lo ejecuta, son los puertos del *avatar*. Para obtener la metafaz, basta simplemente con aplicar este operando sobre el *arquetipo*.

Sin duda, el más complejo es el **bound**, responsable –entre otras cosas– de las complejidades del borrado de conexiones, revisada en el apartado anterior. En el caso de un puerto, es trivial, pues la información está presente *a priori*. En el caso de un componente, en cambio, se ha de recorrer la estructura.

$$[\![\mathbf{bound}(\underline{n})]\!] \ \doteq\ \begin{cases} \underline{n}{\dagger}B(\mathit{lcomp}).\,\overline{\underline{n}{\dagger}B}\langle\,\mathit{lcomp}\,\rangle \ \blacktriangleright\ \mathit{lcomp} & \text{si } \underline{n} \text{ puerto} \\ \underline{n}{\dagger}P(\mathit{lport}).\,\overline{\underline{n}{\dagger}P}\langle\,\mathit{lport}\,\rangle. & \\ \quad (\nu\,\mathit{lco})(\ \mathrm{Gather}\langle\,\mathit{lport}, \mathit{lco}\,\rangle \ \blacktriangleright\ \mathit{lco}\ \blacktriangleright\,) & \text{si } \underline{n} \text{ instancia} \end{cases}$$

De este modo, el proceso Gather recorre la lista de cada puerto, y va acumulando los nombres de instancia obtenidos, eliminando los duplicados en el proceso. Es decir:

$$\begin{array}{rcl} \mathrm{Gather} & : & ((\mathit{NList}\ \uparrow\mathit{DPort}),\, \uparrow(\mathit{List}\ \uparrow\mathit{TComp})) \\ \mathrm{Gather}(lp, lc) & \triangleq & (\nu\,vcb, end)(\ \mathrm{NLFor}\langle\,lp, vcb, end\,\rangle \\ & & \ \mid\ (\nu\,lc_m)(\ \mathrm{Gath}_F\langle\,(\mathtt{nil}), vcb, end, lc_m\,\rangle \\ & & \quad \mid\ lc_m(lc_1).\,\overline{lc}\langle\,lc_1\,\rangle.\,\mathbf{0}\ ) \end{array}$$

$$
\begin{array}{rcl}
 & &) \\
\mathrm{Gath_F} & : & ((\mathit{List}\ \uparrow\mathit{TComp}),\ \uparrow\mathit{DPort},\ \mathit{Sync},\ \uparrow(\mathit{List}\ \uparrow\mathit{TComp})) \\
\mathrm{Gath_F}(\mathit{lis}, \mathit{vcb}, \mathit{end}, \mathit{lc}) & \triangleq & (\ \mathit{vcb}(\mathit{port}).\, \mathit{port}\dagger B(\mathit{lcp}).\, \overline{\mathit{port}\dagger B}\langle\, \mathit{lcp}\,\rangle. \\
 & & \quad (\nu\, \mathit{lct})(\ \mathrm{ApNDup}\langle\, \mathit{lcp}, \mathit{lis}, \mathit{lct}\,\rangle \\
 & & \qquad \mid\ \mathit{lct}(\mathit{lt}).\, \mathrm{Gath}_F\langle\, \mathit{lt}, \mathit{vcb}, \mathit{end}, \mathit{lc}\,\rangle\ ) \\
 & & +\ \mathit{end}.\, \overline{\mathit{lc}}\langle\, \mathit{lis}\,\rangle.\, \mathbf{0}\ )
\end{array}
$$

Se podrían definir, de manera análoga, otros procesos de este tipo, pero el lenguaje, al menos de momento, no proporciona más.

Operadores Reflexivos

Se consideran cuatro operadores, los mismos que fueron definidos en la sección 6.10.3 de la especificación sintáctica. En este caso, será conveniente adaptar ligeramente la definición de *reificación* que se

La búsqueda c e:

$$
\begin{array}{rcl}
[\![\backslash\mathbf{find} & & \overline{a\dagger R}\langle\, lr\,\rangle. \\
 & & \quad (\mathit{ava}).\, \mathrm{AvIn}\langle\, \mathit{ava}, \mathit{lr}, \mathit{pr}\,\rangle \\
 & & \quad \mid\ \mathit{pr}(r_1).\, \overline{F_1}\langle\, (\dagger A \prec \underline{a}, \dagger M \prec \underline{m}, \dagger D \prec r_1)\,\rangle\ ) \\
 & & \quad \blacktriangleright\ F = F_1\ \blacktriangleright\ )
\end{array}
$$

Que se apoya en el proceso auxiliar que sigue a continuación. Éste resulta particularmente interesante, pues tiene un uso más general, y de hecho ya ha sido utilizado con anterioridad.

$$
\begin{array}{rcl}
\mathrm{AvIn} & : & (\mathit{TComp},\ (\mathit{List}\,(\mathit{TReif}\ \mathit{TComp})),\ \uparrow\mathit{Sync}) \\
\mathrm{AvIn}(\mathit{ava}, \mathit{lrei}, \mathit{rr}) & \triangleq & \mathbf{if}\ (\mathtt{null}\ \mathit{lrei})\ \mathbf{then}\ \overline{\mathit{rr}}\langle\, \mathit{ava}\dagger D\,\rangle \\
 & & \quad \mathbf{else}\ (\nu\, \mathit{cr})(\ \overline{\mathit{cr}}\langle\, (\mathtt{car}\ \mathit{lrei})\,\rangle \mid \mathit{cr}(r). \\
 & & \qquad \mathbf{if}\ (r\dagger A = \mathit{ava}\dagger S)\ \mathbf{then}\ \overline{\mathit{rr}}\langle\, r\dagger D\,\rangle \\
 & & \qquad \mathbf{else}\ \mathrm{AvIn}\langle\, \mathit{ava}, (\mathtt{cdr}\ \mathit{lrei}), \mathit{rr}\,\rangle\ )
\end{array}
$$

Obviamente, el objetivo de este proceso es el mismo que el de la operación de búsqueda: averiguar si el avatar considerado ($\mathit{ava}\dagger S$) es alguno de los reificados por el metacomponente, es decir, alguno de los contenidos en su lista de reificaciones (lrei). Una vez que se dispone de estos datos, la tarea es sencilla. Sin embargo, el tipo del resultado plantea un conflicto. En primer lugar, se debe averiguar si el enlace existe o no; por tanto, el resultado ideal es un booleano. Pero, una vez que sabemos que existe, se necesita obtener la otra mitad del dato, es decir, la señal de terminación para el proceso reificado, de tipo *Sync*. Los tipos no coinciden, pero es evidente que el camino de búsqueda es en ambos casos exactamente el mismo.

Por supuesto, podrían definirse dos procesos distintos, pero esto supondría una repetición que es a todas luces innecesaria. En lugar de eso, se adopta una solución de compromiso: se devuelve siempre una señal, pero se utiliza de modo que *también* sirva para indicar la condición

booleana deseada. Para ello, en el caso que *no* exista una reificación, se envía una señal no válida, indicando que la búsqueda ha fracasado. De este modo, cuando la señal recibida es válida, se sabe que la reificación existe, y por tanto la condición es cierta.

Sin embargo, ¿cómo se puede tener una señal no válida?. En principio, toda señal es igualmente correcta, y no se dispone de ningún tipo de "valor nulo" que se diferencie con claridad del resto. Por tanto, la única opción es usar un canal que, en un sistema dado, *no pueda* ser la señal de terminación de un proceso reificado, porque ya esté cumpliendo otra función. El mejor candidato, que tiene además el tipo adecuado ($Sync$), es la señal de eliminación del propio avatar ($ava{\dagger}D$). No puede provocar ninguna confusión: no puede ocupar esa posición casualmente, ya que un avatar nunca se reifica a sí mismo. Además, resulta especialmente adecuada porque estará siempre disponible para cualquier futura comparación con la señal de respuesta, dado que finalmente ambas formarán parte de la misma estructura de datos.

Esa estructura es, precisamente, la que queda caracterizada como el tipo del dato F, que es del traductor –y equivale al nombre F_1 del primer proceso–. Se trata, concretamente, de un canal $\uparrow TReifM$, donde lo significativo es el valor asociado, que describe un tipo abstracto, esencial para la comprensión de todo este apartado, y cuya definición es la siguiente:

$$TReifM \quad \triangleq \quad (\,{\dagger}A{:}{\uparrow}TComp,\ {\dagger}M{:}{\uparrow}TComp,\ {\dagger}D{:}\,Sync)$$

Como resulta evidente, se construye como una extensión del tipo de una reificación convencional ($TReif\ \ TComp$). En ella se ha añadido un nuevo campo (${\dagger}M$), donde se indica el nombre del metacomponente utilizado en la búsqueda, y que presumiblemente mantiene un enlace de reificación hacia el avatar $\underline{a}$ considerado. Este dato debe ser almacenado, ya que será necesario en una hipotética operación futura de borrado de la reificación.

Precisamente por este motivo, es necesario asumir a partir de este momento que, en el contexto del Lenguaje Dinámico, **una *reificación* es un canal de tipo** $\uparrow TReifM$. Esta idea se aplica a todas las operaciones definidas en $\mathcal{P}i\mathcal{L}ar$ para las reificaciones, que se describen en este mismo apartado. Esto, por supuesto, establece una diferencia clara con respecto al Lenguaje Estático, donde una reificación consistía en un registro binario ($TReif\ \ TComp$). Sin embargo, no supone dificultad alguna, ya que la transición de un tipo a otro resulta inmediata. En un sentido, de estático a dinámico, se trata justamente de la operación de búsqueda, ya expuesta; en el contrario es aún más sencilla, puesto que consiste en una simple proyección.

En la sección 6.10.3 se establece que, junto a la operación de búsqueda de una reificación, la sintaxis ha de definir otra, capaz de comprobar la validez de su resultado. Concretamente, debe verificar si dicho resultado toma la forma de lo que allí se denominó, sin descripción alguna, una *reificación nula*. Si ése es el caso, el enlace buscado simplemente no existe.

La definición de esta operación resulta sencilla a la luz de todo lo dicho más arriba. De hecho, el razonamiento en ambos casos es prácticamente el mismo. Una reificación nula será, obviamente, la estructura $\uparrow TReifM$ procedente de una búsqueda fallida; y por tanto, la señal obtenida (${\dagger}D$) coincidirá con la de terminación del avatar (${\dagger}A{\ddagger}D$). Es por tanto evidente que la verificación de nulidad se define como:

$$\begin{aligned}[\![\,\mathbf{nullr}(\underline{R})\,]\!] \quad \mathrel{\hat{=}} \quad & \underline{R}(r).\,\overline{\underline{R}}\langle\, r\,\rangle.\,\mathbf{if}\ \ (r{\dagger}A{\ddagger}D = r{\dagger}D)\ \ \mathbf{then}\ \ \blacktriangleright\ \ true \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \blacktriangleright\ \ false\end{aligned}$$

Visto esto, ya es posible definir la *reificación*, que será aplicada a instancias y conexiones. Comenzando por el caso de los componentes, se tiene, entonces, que:

$$[\![\backslash\mathbf{reify}\ \underline{r}\,(\underline{c} : \underline{T})\,]\!] \ \stackrel{\circ}{=}\ (\nu\,\underline{r})(\ \mathrm{Reify}\langle\,\underline{T}, \underline{c}, \underline{r}\,\rangle\ \blacktriangleright\ )$$

Donde el proceso auxiliar Reify se define como:

$$\begin{array}{rcl}
\mathrm{Reify} & : & (\uparrow\mathcal{M}Comp,\ \uparrow TComp,\ \uparrow TReifM) \\
\mathrm{Reify}(km, kc, r) & \triangleq & kc(c).\,km(m). \\
& & ((\ (\nu\,v_p, e_p)(\ c\dagger P(lc).\,m\dagger P(lm).\,\overline{m\dagger P}\langle\,lm\,\rangle. \\
& & \quad \mathrm{LFor}\langle\,lm, v_p, e_p\,\rangle \mid \mathrm{AddPort}_F\langle\,lc, v_p, e_p, c\dagger P\,\rangle\ ) \\
& & \mid\ (\nu\,v_s, e_s)(\ c\dagger I(lc).\,m\dagger I(lm).\,\overline{m\dagger I}\langle\,lm\,\rangle. \\
& & \quad \mathrm{LFor}\langle\,lm, v_s, e_s\,\rangle \mid \mathrm{AddInst}_F\langle\,lc, v_s, e_s, c\dagger I\,\rangle\ ));\\
& & (\ (\nu\,v_b, e_b)(\ c\dagger B(lc).\,m\dagger B(lb).\,\overline{m\dagger B}\langle\,lb\,\rangle. \\
& & \quad m\dagger E(lle).\,\overline{m\dagger E}\langle\,lle\,\rangle.\,\mathrm{LFor}\langle\,lb, v_b, e_b\,\rangle \\
& & \quad \mid\ \mathrm{AddBind}_F\langle\,lc, lle, v_b, e_b, c\dagger B\,\rangle\ ));\\
& & \quad (\ (\nu\,kd)(\ \mathrm{GenProc}\langle\,kc, km, kd\,\rangle \mid kd(die). \\
& & \quad\quad (\ m\dagger R(lr).\,\mathrm{NewReif}\langle\,lr, c, die, m\dagger R\,\rangle \\
& & \quad\quad \mid\ \overline{r}\langle\,(\dagger A \prec c\dagger S, \dagger M \prec m\dagger S, \dagger D \prec die)\,\rangle.\,\mathbf{0}\ ))))
\end{array}$$

Es decir, se trata de una definición muy similar a la del proceso ReiComp, declarado en la semántica del Lenguaje Estructural. De hecho, ambos utilizan los mismos subprocesos, y la explicación es similar.

La reificación de una conexión es aún más sencilla, porque se trata tan sólo de crearla, sin que esto afecte a la conexión anterior, salvo en el hecho de que se amplía su conjunto de puertos. Sin embargo, en este caso no se tienen ni subinstancias ni subconexiones.

$$[\![\backslash\mathbf{reify}\ \underline{r}\,(\underline{b} : \underline{\mathrm{T}}(\widetilde{\underline{c}}, \widetilde{\underline{p}}))\,]\!] \ \stackrel{\circ}{=}\ (\nu\,\underline{r})(\ \mathrm{ReifyBind}\langle\,\underline{T}, \underline{b}, \widetilde{\underline{c}}, \widetilde{\underline{p}}, \underline{r}\,\rangle\ \blacktriangleright\ )$$

El despliegue del proceso ReifyBind creará los puertos correspondientes, que se unen a la interfaz. También se crea una copia del proceso reflexivo ligado al tipo, y se actualiza la información de ligadura asociada a los puertos.

$$\begin{array}{rcl}
\mathrm{ReifyBind} & : & (\uparrow\mathcal{M}Comp,\ \uparrow TBind,\ \sim\!\uparrow TComp,\ \sim\!\uparrow DPort, \\
& & \quad \uparrow TReifM) \\
\mathrm{ReifyBind}(km, kb, \widetilde{c}, \widetilde{p}, r) & \triangleq & kb(b).\,km(m). \\
& & \quad ((\nu\,v_p, e_p)(\ b\dagger P(lp).\,m\dagger P(lm). \\
& & \quad \overline{m\dagger P}\langle\,lm\,\rangle.\,\mathrm{LFor}\langle\,lm, v_p, e_p\,\rangle
\end{array}$$

$$
\begin{array}{rcl}
 & & \quad \mid \ \mathrm{AddBP}_F\langle\, lc, v_p, e_p, 1, \widetilde{c}, \widetilde{p}, b\dagger P \,\rangle\ );\\
 & & \quad ((\nu\, kd)(\ \mathrm{GenProc}\langle\, kc, km, kd \,\rangle \mid kd(die).\\
 & & \quad\ (\ m\dagger R(lr).\, \mathrm{NewReif}\langle\, lr, c, die, m\dagger R \,\rangle\\
 & & \quad\ \mid \overline{r}\langle\, (\dagger A \prec c\dagger S, \dagger M \prec m\dagger S, \dagger D \prec die)\,\rangle.\, \mathbf{0}\ )\\
 & &)))\\
\mathrm{AddBP_F} & : & ((\mathit{NList}\ \uparrow\mathit{DPort}),\ \uparrow\mathit{Label},\ \mathit{Sync},\ \mathit{Int},\\
 & & \quad \sim\!\uparrow\mathit{TComp},\ \sim\!\uparrow\mathit{DPort},\ \uparrow(\mathit{NList}\ \uparrow\mathit{DPort}))\\
\mathrm{AddBP_F}(lc, v, end, ix, \widetilde{c}, \widetilde{p}, r) & \triangleq & (\nu\, bp)(\ v(tag).\\
 & & (\ \mathrm{BPort}\langle\, p_{ix}\dagger O, c_{ix}, bp \,\rangle \mid bp(m_p).\\
 & & \quad \mathrm{AddBP}_F\langle\, (\mathtt{ncons}\ tag\ m_p\ lc),\\
 & & \quad\ v, end, (ix+1), \widetilde{c}, \widetilde{p}, r\,\rangle\ )\\
 & & +\ end.\, \overline{r}\langle\, lc \,\rangle.\, \mathbf{0}\ )
\end{array}
$$

Finalmente, se ha de considerar el borrado de reificaciones. En este sentido, la instrucción **del** se aplica del mismo modo que en componentes y conexiones; pero está sin embargo en en un nivel distinto, porque no actúa sobre una capa del sistema, sino sobre *aquello* que liga entre sí las capas del sistema.

$$
\begin{array}{rcl}
[\![\backslash\mathbf{del}\,\underline{R}]\!] & \doteq & \underline{R}(r).\, \mathbf{if}\ (r\dagger A\ddagger D \neq r\dagger D)\ \mathbf{then}\ \overline{r\dagger D}.\\
 & & (\ (\nu\, rx)(\ r\dagger M(m).\, r\dagger A(a).\, (\ \mathrm{ReiSim}\langle\, r, rx \,\rangle\\
 & & \quad \mid rx(r_1).\, m\dagger R(lr).\, \overline{m\dagger R}\langle\, (\mathtt{era}\ r_1\ lr)\,\rangle\\
 & & \mid\ m\dagger B(lb).\, (\nu\, v_1, e_1)(\ \mathrm{LLFor}\langle\, lb, v_1, e_1 \,\rangle\\
 & & \quad \mid \mathrm{DelBind}_F\langle\, a\dagger B, v_1, e_1 \,\rangle\ )\\
 & & \mid\ m\dagger P(lp).\, (\nu\, v_2, e_2)(\ \mathrm{LLFor}\langle\, lp, v_2, e_2 \,\rangle\\
 & & \quad \mid \mathrm{DelPort}_F\langle\, a\dagger P, v_2, e_2 \,\rangle\ )\\
 & & \mid\ m\dagger I(li).\, (\nu\, v_3, e_3)(\ \mathrm{LLFor}\langle\, li, v_3, e_3 \,\rangle\\
 & & \quad \mid \mathrm{DelInst}_F\langle\, a\dagger I, v_3, e_3 \,\rangle\ )))\\
 & & \mathbf{else}\ \mathbf{0}
\end{array}
$$

Es decir, se comprueba la reificación, y se obtiene el metacomponente asociado. A partir de ahí, se eliminan todas las conexiones, puertos e instancias del avatar cuya creación había sido inducida por la reificación. Los procesos auxiliares se definen como:

$$
\begin{array}{rcl}
\mathrm{ReiSim} & : & (\mathit{TReifM},\ \uparrow(\mathit{TReif}\ \mathit{TComp}))\\
\mathrm{ReiSim}(rm, r) & \triangleq & \overline{r}\langle\, (\dagger A \prec (rm\dagger A), \dagger D \prec (rm\dagger D))\,\rangle\\
\mathrm{DelPort_F} & : & (\uparrow(\mathit{NList}\ \uparrow\mathit{DPort}),\ \uparrow\mathit{Label},\ \mathit{Sync})\\
\mathrm{DelPort_F}(mp, vcb, end) & \triangleq & (\ vcb(tag).\, mp(lport).
\end{array}
$$

$$
\begin{array}{rcl}
 & & \textbf{if } (\texttt{lein}\ tag\ lport)\ \textbf{then}\ \overline{lport \bullet tag\dagger D}. \\
 & & \quad \overline{mp}\langle\,(\texttt{lera}\ tag\ lport)\,\rangle.\,\mathrm{DelPort}_F\langle\,mp, vcb, end\,\rangle \\
 & & \quad \textbf{else}\ \overline{mp}\langle\,lport\,\rangle.\,\mathrm{DelPort}_F\langle\,mp, vcb, end\,\rangle \\
 & & +\ end.\,\mathbf{0}\ ) \\
\mathrm{DelBind_F} & : & (\uparrow(NList\ \uparrow TBind),\ \uparrow Label,\ Sync) \\
\mathrm{DelBind_F}(mb, vcb, end) & \triangleq & (\ vcb(tag).\,mb(lbind). \\
 & & \textbf{if } (\texttt{lein}\ tag\ lbind)\ \textbf{then}\ \overline{lbind \bullet tag\dagger D}. \\
 & & \quad \overline{mb}\langle\,(\texttt{lera}\ tag\ lbind)\,\rangle.\,\mathrm{DelBind}_F\langle\,mb, vcb, end\,\rangle \\
 & & \quad \textbf{else}\ \overline{mb}\langle\,lbind\,\rangle.\,\mathrm{DelBind}_F\langle\,mb, vcb, end\,\rangle \\
 & & +\ end.\,\mathbf{0}\ ) \\
\mathrm{DelInst_F} & : & (\uparrow(NList\ \uparrow TComp),\ \uparrow Label,\ Sync) \\
\mathrm{DelInst_F}(mi, vcb, end) & \triangleq & (\ vcb(tag).\,mi(lins). \\
 & & \textbf{if } (\texttt{lein}\ tag\ lins)\ \textbf{then}\ \overline{lins \bullet tag\dagger D}. \\
 & & \quad \overline{mi}\langle\,(\texttt{lera}\ tab\ lins)\,\rangle.\,\mathrm{DelInst}_F\langle\,mi, vcb, end\,\rangle \\
 & & \quad \textbf{else}\ \overline{mi}\langle\,lins\,\rangle.\,\mathrm{DelInst}_F\langle\,mi, vcb, end\,\rangle \\
 & & +\ end.\,\mathbf{0}\ )
\end{array}
$$

En este caso, dado que todos los elementos implicados serán destruidos, no es necesario restaurar la meta-información; por tanto, se puede realizar un borrado directo, que por este motivo es mucho más sencillo.

7.5 Extensión: Envolventes

La extensión descrita en la sección 6.8 es casi trivial desde el punto de vista de la semántica; apenas se ha de introducir un pequeño cambio en la definición de un puerto, y las dos instrucciones de habilitación de entrada y salida definidas.

7.5.1 Puerto

El tipo de un puerto, en un componente *envuelto*, pasa a ser:

$$
\begin{array}{rcl}
(\,WPort\,\alpha\,\beta) & \triangleq & \mu P.(\,\dagger I\colon\ (Par\,\alpha\,\beta), \\
 & & \quad \dagger O\colon\ (Par\,\beta\,\alpha), \\
 & & \quad \dagger W\colon (Par\ \uparrow\alpha\ \uparrow\alpha), \\
 & & \quad \dagger M\colon (Par\,\alpha\ \uparrow\beta), \\
 & & \quad \dagger D\colon\ Sync, \\
 & & \quad \dagger B\colon\uparrow(List\ \uparrow TComp), \\
 & & \quad \dagger S\colon\uparrow P, \\
 & & \quad \dagger L\colon (Par\,No\,No)\,)
\end{array}
$$

Obsérvese que se trata tan sólo de una extensión de la definición original, en la que aparece un nuevo par ($\dagger\mathbf{L}$), compuesto por dos señales de sincronización puras. Además, dado que el tipo *WPort* es una extensión de *TPort*, cuando ambos se definen como registros, se trata de un *subtipo* suyo –según las reglas de Pierce y Turner para el subtipado de registros en cálculo-π, que son por otra parte estándar en teoría de tipos–.

El proceso que define a un *puerto envuelto* es, simplemente:

$$
\begin{array}{rcl}
\text{WraPort} & : & (\sharp\alpha,\, \sharp\beta;\, \uparrow(\mathit{WPort}\,\alpha\,\beta)) \\
\text{WraPort}(p) & \triangleq & (\nu\, ii, io, oi, oo, wi, wo, sh, ct, bp, kill, me, li, lo, k_1, k_2) \\
& & (\,\overline{bp}\langle\,(\mathtt{nil})\,\rangle.\,\mathbf{0} \mid kill.\,\overline{k_1}.\,\overline{k_2}.\,\mathbf{0} \\
& & \mid me(p_1).\,\overline{p}\langle\,p_1\,\rangle.\,\mathbf{0} \\
& & \mid \text{RepSelf}\langle\, me, (\dagger I \prec [ii, io], \dagger O \prec [oi, oo], \\
& & \quad \dagger W \prec [wi, wo], \dagger M \prec [sh, ct], \\
& & \quad \dagger D \prec kill, \dagger B \prec bp, \\
& & \quad \dagger S \prec me, \dagger L \prec [li, lo]\,), k_1\rangle \\
& & \mid \text{WBehPort}\langle\,[ii, io], [oi, oo], [wi, wo], [sh, ct], lo, k_2\,\rangle)
\end{array}
$$

Donde, aparte de incluir el nuevo par declarado en el tipo, se cambia también el proceso que define el comportamiento del puerto (WBehPort). Sin embargo, como podrá comprobarse, los cambios son mínimos, y sólo afectan al proceso de *salida*.

$$
\begin{array}{rcl}
\text{WBehPort} & : & (\sharp\alpha,\, \sharp\beta;\, (\mathit{Par}\,\alpha\,\beta),\, (\mathit{Par}\,\beta\,\alpha),\, (\mathit{Par}\ \uparrow\alpha\ \uparrow\alpha), \\
& & \quad (\mathit{Par}\,\alpha\ \uparrow\beta),\, \mathit{Sync},\, \mathit{Sync}) \\
\text{WBehPort}(\widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, lo, die) & \triangleq & (\nu\, d_1, d_2)(\, die.\,\overline{d_1}.\,\overline{d_2}.\,\mathbf{0} \mid \\
& & \text{WOutPort}\langle\,\widehat{\imath}, \widehat{o}, \widehat{m}, lo, d_2\,\rangle \mid \text{InPort}\langle\,\widehat{\imath}, \widehat{o}, \widehat{w}, \widehat{m}, d_1\,\rangle\,) \\
\text{WOutPort} & : & (\sharp\alpha,\, \sharp\beta;\, (\mathit{Par}\,\alpha\,\beta),\, (\mathit{Par}\,\beta\,\alpha),\, (\mathit{Par}\,\alpha\ \uparrow\beta), \\
& & \quad \mathit{Sync},\, \mathit{Sync}) \\
\text{WOutPort}(\widehat{\imath}, \widehat{o}, \widehat{m}, lo, die) & \triangleq & lo.\, i_o(x).\,\overline{o_o}\langle\,x\,\rangle.\,\text{WOutPort}\langle\,\widehat{\imath}, \widehat{o}, \widehat{m}, die\,\rangle \\
& & + m_o(r).\, i_o(x).\,\overline{r}\langle\,x\,\rangle.\,\text{WOutPort}\langle\,\widehat{\imath}, \widehat{o}, \widehat{m}, die\,\rangle \\
& & + die.\,\mathbf{0}
\end{array}
$$

En WBehPort sólo se modificarán los aspectos de salida, lo que requiere la definición de WOutPort. Pero, sin embargo, no es necesario modificar la entrada, que permanece con el InPort original. La razón de esto último es que forzar una sincronización sobre l_i en toda entrada forzaría igualmente a habilitar cada interacción en el nivel meta; esto, además de incómodo, podría provocar cierto indeterminismo, ya que podrían producirse condiciones de carrera durante la activación. Por tanto, es el *traductor* el que realiza este proceso, de modo que:

$$
[\![a?x]\!] \;\triangleq\; a\dagger L_i.\,\overline{a\dagger W_o}\langle\,a\dagger I_i\,\rangle.\,a\dagger I_i(x)
$$

Es decir, el traductor detecta que en este caso se está tratando con un componente envuelto –de ahí la necesidad de marcarlo con **wrap**–, y realiza ahora de manera diferente la traducción de una recepción del *nivel base*. En este caso, lo que ocurre es que la sincronización sobre l_i se fuerza en cada lectura: no es necesario, pues, imponerlo en la definición del puerto.

Una vez dicho esto, es obvio que las dos operaciones de habilitación resultan triviales, y se basan simplemente en realizar una sincronización sobre los dos canales definidos. Así pues, la que permite una entrada se traduce como:

$$[\![\backslash\mathbf{letin}\,\underline{P}\,]\!] \;\;\mathring{=}\;\; \overline{\underline{P}\dagger L_i}$$

Y correspondiene a una salida, análogamente:

$$[\![\backslash\mathbf{letout}\,\underline{P}\,]\!] \;\;\mathring{=}\;\; \overline{\underline{P}\dagger L_o}$$

Resulta significativo señalar aquí que la introducción del nombre lo –$\underline{P}\dagger L_o$–, tal como se incorpora en el subproceso WOutPort, no afecta en absoluto al comportamiento de una instrucción **catch**, cuya semántica fue descrita en la sección 7.4.4; del mismo modo que su dual li –$\underline{P}\dagger L_i$–, tampoco afectaría a una instrucción **shift**. Es precisamente el uso de éstas lo que permite que el componente envolvente evite, cuando sea preciso, los bloqueos –cerrojos– que la semántica impone sobre la interacción del componente envuelto; de aquí se deriva la importancia de ambas en este contexto específico.

Con este comentario, se concluye la descripción de la extensión sintáctica para envolventes; y se finaliza, en definitiva, la especificación de la semántica formal de $\mathcal{P}i\mathcal{L}ar$.

7.6 Conclusión

A lo largo de este Capítulo, se ha descrito formalmente la semántica asociada a cada uno de los conceptos utilizados en la definición sintáctica de $\mathcal{P}i\mathcal{L}ar$: tanto los elementos del Lenguaje Estructural, que definen el modo en el que se concibe una especificación arquitectónica concurrente y reflexiva, como los términos del Lenguaje Dinámico, que hacen posible una traducción casi directa de las restricciones a la notación propia de un álgebra de procesos, en este caso un dialecto concreto, tipado y polimórfico, del cálculo-π.

Tras todo este desarrollo pueden concluirse, en definitiva, dos detalles fundamentales, que también fueron brevemente esbozados en la introducción. Por una parte, se ha demostrado que, a pesar de su complejidad y de la dificultad de su enfoque reflexivo, la especificación de $\mathcal{P}i\mathcal{L}ar$ en términos formales –los del cálculo-π– es perfectamente posible, lo que demuestra la solidez y coherencia del planteamiento de partida. Más aún, la traducción se ha realizado a un cálculo *tipado*, lo que en principio resultaría aún más difícil; es cierto que se ha empleado un sistema de tipos con recursión, polimorfismo e incluso un tipo dinámico, pero en todo caso se ha podido comprobar que la traducción es, ante todo, *consistente*.

En segundo lugar, al realizar la traducción en términos de un formalismo no reflexivo, se demuestra también la auténtica simplicidad del enfoque utilizado: aunque la Reflexión ha demostrado ser potente y expresiva, en realidad no es estrictamente necesaria, ya que se puede

desarrollar como una estructura puramente interactiva. No obstante, es la concepción reflexiva la que ha guiado la construcción de esta estructura, y sigue apareciendo como la abstracción más adecuada desde un punto de vista sintáctico. Sería muy difícil describir de otro modo, igualmente compacto y consistente, el esquema –realmente complejo– de estructuras estáticas y procesos activos que se define en la sección 7.2.5, y que constituye el núcleo de esta semántica.

Finalmente, conviene también señalar que el hecho de reducir el uso del cálculo-π a una traducción interna garantiza que no se pueden producir alteraciones inesperadas o efectos secundarios a causa de la movilidad, dado que la semántica los controla implícitamente. En efecto, ésta es tal que nunca se transmite un nombre *fuera de su ámbito*: esto asegura que toda comunicación se hace estrictamente a través de conexiones, uno de los postulados básicos de un Adl [MT00]. Más aún, estas conexiones usan nombres creados de manera específica, no relacionados con los del otro extremo, por lo que ni siquiera en caso de ser capturados podrían ser utilizados para una interacción directa entre las instancias.

Además, una extrusión de ámbito ocasional no supondría en realidad ningún problema: incluso aunque un nombre lograse difundirse hasta un ámbito no previsto, el lenguaje no le permitiría aprovechar esta información. El proceso de traducción aplica siempre las instrucciones de $\mathcal{P}i\mathcal{L}ar$ de una manera controlada, lo que elimina cualquier posibilidad de encontrar, incluso en este caso, un comportamiento imprevisto.

En definitiva, con la definición de $\mathcal{P}i\mathcal{L}ar$ se ha logrado disponer de un lenguaje ubicado a otro nivel de abstracción, pero con una expresividad y coherencia comparables a las del cálculo-π, eliminando además en el proceso la posibilidad de introducir otro tipo de errores, puramente algebraicos, que serían claramente inconsistentes con la concepción de un Adl.

Capítulo 8

Arquitectura Dinámica en PiLar

8.1 Introducción

A lo largo de este Capítulo se presentan y explican brevemente algunos ejemplos de uso de $\mathcal{P}i\mathcal{L}ar$, con el fin de demostrar la aplicabilidad del lenguaje a especificaciones arquitectónicas concretas. Debe de tenerse en cuenta que, aunque la finalidad del lenguaje es ante todo experimental –su principal objetivo es estudiar la integración de reflexión en la Arquitectura de Software–, el interés de este enfoque vendrá dado por su capacidad para realizar descripciones coherentes y comprensibles, que determinen su viabilidad a medio plazo.

En esta tesis se ha preferido optar por elaborar una serie de ejemplos de menor tamaño, en lugar de utilizar un único gran ejemplo que mezclase en su interior todas las estructuras necesarias. De este modo resulta más sencillo mostrar el modo en que se pueden afrontar ciertas situaciones concretas en el seno del lenguaje, y se resaltan con mayor claridad sus puntos fuertes y debilidades. Lo cierto es que un ejemplo más cercano a la experiencia del mundo real sería, o bien demasiado sencillo, o bien demasiado complejo, dependiendo de la granularidad de los componentes descritos. Si ésta fuera lo bastante amplia, la especificación se limitaría a ser apenas una enumeración de los componentes implicados; en cambio, si fuera lo bastante fina, su longitud excedería con mucho los propósitos de este Capítulo.

De hecho, existe en la Arquitectura de Software, al igual que en otros campos –y en particular en los estudios sobre Concurrencia– una amplia tradición sobre el uso de ejemplos reducidos (*ejemplos de juguete*), algunos de los cuales son bastante conocidos. Sin embargo, los ya existentes describen en su mayoría sistemas de carácter estático, por lo que en general se han utilizado supuestos originales, o casos extrapolados de los contextos de Concurrencia y Coordinación. La única excepción es el primer ejemplo considerado (§8.2), que se introduce exclusivamente a efectos de comparación, y que describe en efecto un sistema estrictamente estático.

Ha de señalarse que el uso de estos ejemplos menores hace que, muy a menudo, la granularidad de los componentes no sea la deseable en una especificación situada en el nivel *arquitectónico*. No obstante, no es ése el aspecto que se considera importante a lo largo de este Capítulo, sino que se presta mucha mayor atención a la complejidad de las estructuras (abstractas) descritas. En este sentido, los sistemas examinados sí cumplen adecuadamente su función, ya que prueban los límites expresivos del lenguaje, al poner de manifiesto los patrones y esquemas arquitectónicos

que pueden especificarse con mayor naturalidad.

En este Capítulo se ha intentado introducir al menos un ejemplo de cada una de las características más notables del lenguaje; no obstante, no siempre se ha podido evitar entremezclar varias de ellas. Por otra parte, la extensión de este trabajo hace aconsejable limitar el número de ejemplos incluidos en el mismo; aunque una presentación adecuada del lenguaje admitiría con facilidad la inclusión de tres o cuatro más, lo cierto es que la longitud de la misma resultaría claramente excesiva. Aun así, no se puede dejar de señalar que existe toda una serie de sistemas cuyo planteamiento en $\mathcal{P}i\mathcal{L}ar$ resulta especialmente interesante: algunos de ellos serán mencionados en el Capítulo próximo como posibles líneas de trabajo futuro.

8.2 Ejemplo de la Gasolinera

El ejemplo de *La Gasolinera* es uno de los clásicos de la Arquitectura de Software. Fue definido originalmente en 1985 por Hembold y Luckham como ejemplo de aplicación de programas Ada, y fue reintroducido en el campo por Naumovich [NACO97], que describió su estructura –fuertemente estática– en Wright. El objetivo del ejemplo es mostrar que, gracias a la especificación formal (en su caso, en el dialecto de Csp usado en Wright) del protocolo de interacción, es posible comprobar que el sistema concurrente cumple una serie de propiedades, así como detectar las situaciones inesperadas debidas a condiciones de carrera, y corregirlas adecuadamente.

El problema ha sido retomado en varias ocasiones; destaca en especial la descripción del sistema en $\delta arwin$ hecha por Magee en [MKG99], que a un tiempo lo generaliza y simplifica, y que es en la que se inspira nuestra propia versión del ejemplo.

```
\component Vehiculo
( ( port prepago |
    port deposito )
  \constraint
  ( prepago!dinero ;
    deposito?gasolina
) )

\component Bomba
( ( port cobro |
    port surtidor )
  \constraint ( \rep
  ( cobro?dinero ; tau (dinero , gasolina) ;
    surtidor ! gasolina )
) )

\component Gasolinera<N:Int>
( \config
  ( veh[N]:Vehiculo |
    bom[N]:Bomba |
    \bind ( \for (1 .. N)
    ( veh[now].prepago =
        bom[now].cobro |
      veh[now].deposito =
        bom[now].surtidor )
) ) )
```

Figura 8.1: Sistema de la Gasolinera

El sistema describe una gasolinera como una arquitectura estática; es decir, en realidad está describiendo una *situación* dada, en un momento dado, en dicha gasolinera. En la descripción original, la arquitectura constaba de cuatro elementos: dos clientes, una bomba de gasolina (con dos surtidores) y un cajero. Cada cliente se conecta al cajero y a un surtidor, y el cajero a la

bomba. Se trata de un sistema de prepago: el cliente paga al cajero una determinada cantidad de dinero, y el surtidor emite en respuesta la correspondiente cantidad de gasolina. El enunciado presupone, por simplicidad, que siempre hay gasolina en las bombas, y que siempre se realiza un pago suficiente para obtener al menos cierta cantidad de combustible.

La versión original de Naumovich no permite más que dos clientes, y necesita para su descripción nada menos que una configuración de cuatro componentes y cinco conectores Wright. La versión de Magee es más general, y a la vez más simple. En primer lugar, se extiende el problema a un número indefinido (N) de clientes, y en lugar de una única bomba, se consideran también N bombas de gasolina, cada una (ahora) con un surtidor[1]. Además, se constata que la tarea del cajero es totalmente paralelizable: el pago de un cliente se relaciona con el bombeo de gasolina en su vehículo, de manera completamente independiente del resto. Así pues, el cajero centralizado desaparece y pasa a estar implícitamente *distribuido* entre las bombas: cada una controla sus propios pagos y habilita su propio surtidor. Debe recordarse, no obstante, que la arquitectura sigue siendo estática: la conexión entre cada uno de los vehículos y las bombas correspondientes está establecida ya desde el principio.

La Figura 8.1 muestra una descripción $\mathcal{P}i\mathcal{L}ar$ del sistema análoga a la solución de Magee. Consta de un componente parametrizado compuesto (Gasolinera), que a su vez contiene N instancias de los componentes Vehículo y Bomba, y liga estas instancias mediante enlaces simples, establecidos respectivamente entre los puertos de prepago y cobro, y los de depósito y surtidor. El comportamiento queda definido por las restricciones de los dos tipos de componente primitivos: cada vehículo paga una cantidad de dinero, y espera a recibir en su depósito la gasolina correspondiente. Por su parte, cada bomba espera a recibir un pago, momento en el cual calcula –mediante un proceso interno, que no es modelado salvo como **tau**– la cantidad de gasolina correspondiente, y la emite por su surtidor. Se asume que el vehículo sólo realiza esta operación una vez, mientras que una bomba puede realizarla muchas veces; aunque en esta arquitectura eso no supone diferencia alguna, ya que en cada caso sólo es posible un intercambio.

Toda esta actividad puede desplegarse en paralelo, sin interferencia alguna. El sistema final es muy sencillo, y tal vez lo más llamativo de la descripción es solamente el bucle **for** utilizado por el componente compuesto para describir la conexión entre los N pares vehículo-bomba presentes en el sistema. En este caso, se podría haber utilizado igualmente un bucle **loop**. La relación entre dinero y gasolina es matemática y se podría haber modelado en términos de operaciones aritméticas, que $\mathcal{P}i\mathcal{L}ar$ permite. Pero esto forma parte de la especificación *funcional*, más que de la propiamente arquitectónica, y no afecta para nada a la estructura del sistema. Por eso se indica como una acción **tau**, que como es sabido tiene una función meramente informativa.

Como comentario anecdótico cabe decir que, si bien este problema no se ha introducido aquí con el objetivo de estudiar las propiedades de consistencia descritas por Naumovich, de hecho no resultaría difícil demostrar que, en efecto, la versión $\mathcal{P}i\mathcal{L}ar$ sí las cumple. En particular, las dos condiciones de carrera más significativas de [NACO97] no pueden darse. La primera expresaba un conflicto en el acceso concurrente a las bombas por parte de los vehículos, que aquí no puede aparecer, debido a la estructura de *arrays* de instancias. La segunda se producía por el indeterminismo implícito en los conectores Wright; de nuevo, aquí esto no puede ocurrir,

[1] En realidad, esta generalización no es un mérito de $\delta arwin$. En la versión final de Wright [All97], ésta hubiera sido igualmente posible, ya que también dispone de *arrays* de instancias y puertos.

dado que los cinco conectores han desaparecido, y que los *enlaces* de $\mathcal{P}i\mathcal{L}ar$ no pueden tener este problema, al menos cuando su cardinalidad es *uno a uno*.

Este ejemplo se ha desarrollado para mostrar el uso de $\mathcal{P}i\mathcal{L}ar$ en una arquitectura estática clásica. Puede comprobarse que no es necesario hacer referencia alguna a reificaciones o avatares, y que el sistema se describe de manera análoga a como se hace en otros Adls; de hecho, con mayor facilidad que en algunos de ellos. Dado que los restantes ejemplos de este capítulo son de naturaleza dinámica, y que hacen un uso frecuente de las características reflexivas, se pretende evitar con éste una falsa impresión de complejidad en el lenguaje: $\mathcal{P}i\mathcal{L}ar$ es tan sencillo, o tan complejo, como cualquier otro Adl ante cualquiera de los problemas clásicos.

8.2.1 Variante: una Gasolinera Dinámica

Se puede argumentar (con razón) que ninguna de las tres versiones describe adecuadamente el funcionamiento de una gasolinera, dado que intenta hacerlo como un *sistema cerrado*. En la realidad, la gasolinera es un *sistema abierto*: tendrá un flujo indeterminado de clientes, que irán cambiando. En algunos momentos tendrá surtidores libres, y en otros habrá vehículos a la espera. Desde este punto de vista, se puede considerar una perspectiva totalmente distinta del sistema, hasta el punto de que podría decirse que se trata de un ejemplo completamente nuevo. Desde luego, en este caso lo que se describe es una *arquitectura dinámica*.

Por supuesto, no es posible especificar completamente un sistema abierto mediante una descripción arquitectónica, que por su propia naturaleza es cerrada y no puede incorporar nuevas instancias de Vehículo *a posteriori*. Sin embargo, se pueden utilizar diversos artificios para simular el suministro indefinido de clientes; el más sencillo es considerarlos contenidos en un array de longitud M indeterminada, que modelase –por ejemplo– cuántos clientes ha tenido la gasolinera a lo largo de un día. Luego, tan sólo sería necesario especificar el protocolo según el cual un vehículo se conecta a un surtidor libre, realiza la transacción y abandona finalmente la gasolinera, es decir: desaparece del sistema. En la realidad, este protocolo no viene dictado por el sistema, sino que forma parte del comportamiento de los clientes, o del modo en que éstos reaccionan ante una gasolinera llena o vacía, y de la manera más o menos aleatoria en que eligen un surtidor concreto de entre todos los que estén libres en ese momento.

Por tanto, para plantear una variante dinámica del problema de la gasolinera, todavía simplificada pero más realista, basta con abstraer el número de clientes, suponiéndolo indeterminado. Esto implica renunciar únicamente a una de las suposiciones del planteamiento original, a saber: que el número de clientes coincide con el número de surtidores. Ahora, con un suministro indefinido de vehículos, puede ocurrir fácilmente que la cantidad de los mismos supere a la de surtidores en la gasolinera, y por tanto ha de plantearse una estrategia –un patrón dinámico– que conecte vehículos a los surtidores libres, los mantenga a la espera cuando no hay ninguno disponible, y libere las conexiones una vez que la gasolina haya sido bombeada al vehículo.

La Figura 8.2 muestra un posible planteamiento dinámico del sistema según esta idea. En la descripción no se muestra la definición de una Bomba porque ésta no ha cambiado con respecto a la versión anterior. En cambio, sí cambia el comportamiento –aunque no la interfaz– de un Vehículo, y la configuración global de la gasolinera, especificada aquí como el componente GasoDin, en el que se han modificado tanto estructura como comportamiento.

Puede verse que la gasolinera comienza con un planteamiento similar al del caso anterior: se

```
\component Vehiculo
( ( port prepago |
    port deposito )
  \constraint
  ( prepago!dinero ; deposito?gasolina ;
    \loop ( bound (prepago) ) ( \del now );
    \loop ( bound (deposito) ) ( \del now );
    \del avatar ) )

\component GasoDin<N,M:Int>
( \configuration
  ( veh[M]:Vehiculo |
    bom[N]:Bomba |
    \bind
    ( \if (N>=M)
        ( \for i (1..M)
            ( veh[i].prepago =
              bom[i].cobro |
              veh[i].deposito =
              bom[i].surtidor ) )
  ) )
  \constraint
  ( \for i ( avatarSet (Vehiculo))
    ( \if ( card ( bound (i.prepago)) = 0 )
      ( \loop j ( avatarSet (Bomba))
        ( \if ( card ( bound (j.cobro)) = 0 and
                card ( bound (i.prepago)) = 0 )
          ( \new (L:(i.prepago=j.cobro)) |
            \new (M:(i.deposito=j.surtidor))
    ) ) ) )
) )
```

Figura 8.2: La Gasolinera, versión Dinámica

dispone de un conjunto fijo de N surtidores, y un flujo indefinido de M vehículos; como se ha dicho, el valor de estos dos parámetros puede ser ahora distinto. En el caso de que el número de bombas iguale o supere al de clientes, la situación es idéntica al caso estático: simplemente se enlazan todos los vehículos a los primeros N surtidores, utilizando el bucle **for** indicado. El comportamiento dinámico sólo afecta en el caso contrario.

La restricción dinámica de GasoDin expresa este comportamiento, y puede leerse como sigue. Se toman uno a uno, y por orden de llegada, todos los vehículos del sistema. Para ello, se recorre de manera secuencial, mediante un bucle **for**, el conjunto de avatares (**avatarSet**) del arquetipo Vehículo. Para cada uno de ellos, se examina su puerto prepago: si éste está libre, el vehículo no ha sido aún enlazado a un surtidor, por lo que está a la espera. Para saber que dicho puerto está libre se utiliza el operando reflexivo **bound**, que proporciona –en este caso– el conjunto de conexiones ligadas a éste. Por tanto, si el tamaño (**card**) de este conjunto es cero, ninguna conexión lo enlaza aún a una bomba.

En tal caso, se examina el conjunto de surtidores (**avatarSet**(Bomba)), buscando alguno que esté aún libre. En este caso el orden es indiferente, por lo que se usa un bucle **loop**, aunque esto no es estrictamente necesario. Si se encuentra algún surtidor libre, esto es, si el puerto cobro del mismo no está conectado –lo que se comprueba del mismo modo que antes–, se crean entonces dos enlaces simples L y M entre los puertos del vehículo veh y la bomba bom, con lo que pueden comenzar su interacción. Obsérvese que se comprueba de nuevo la condición de que el vehículo sigue desconectado, dado que el bucle **loop** se despliega en paralelo, y podría haber sido enlazado por cualquiera de las otras ramas del mismo.

Ha de considerarse, entonces, qué ocurre con un vehículo una vez que ha sido conectado al surtidor, lo que se define como el comportamiento del propio arquetipo Vehículo. Inicialmente, coincide con el del caso estático; la única diferencia se plantea cuando la gasolina ya ha sido

bombeada, y se ha de liberar el surtidor. Entonces, se examinan los puertos del vehículo, destruyendo (**del**) cada una de las conexiones ligadas a los mismos, que estaban contenidas en el conjunto **bound** correspondiente. De este modo, se libera la bomba utilizada y el vehículo "se va" de la gasolinera, lo que se expresa aquí destruyendo la instancia (**avatar**) que acaba de ser desconectada. Aunque esto último no es estrictamente necesario, evita que dicha instancia afecte, aunque sea a través del esquema reflexivo, al comportamiento del resto del sistema.

Por supuesto, esta especificación es sólo un esbozo de un desarrollo más completo. Por ejemplo, no se han considerado con detenimiento las posibles condiciones de carrera, como las que pueden darse cuando un llega un vehículo sin que haya ninguna bomba libre; este conflicto puede solucionarse con la inserción de un evento de espera, o mediante la adición de un bucle de prueba, cuya condición de terminación resultaría relativamente complicada. Pero en todo caso, el objetivo de este apartado era únicamente plantear el interés de una variante dinámica de este ejemplo clásico, observar sus diferencias con el original, y mostrar alguna de las maneras en las que $\mathcal{P}i\mathcal{L}ar$ puede acometer la labor de realizar su descripción.

BIBLIOTECA VIRTUAL

8.3 Un Sistema Cliente/Servidor [2|3]-*tier*

Este ejemplo muestra una de las descripciones dinámicas en $\mathcal{P}i\mathcal{L}ar$ más breves posibles, pero permite ilustrar de manera muy sencilla algunos de sus conceptos más significativos. En particular se examina el efecto de la reificación sobre las conexiones, origen de los ya comentados *conectores de nivel meta* (§6.11); el aquí presentado es también uno de los más simples.

La especificación se inspira vagamente en el ejemplo para Wright *Dinámico* (§3.7.1) propuesto por Robert Allen, Rémi Douence y David Garlan en [ADG97, ADG98]. Aquél, a pesar de su difusión, es uno de los ejemplos existentes que muestran menor grado de dinamismo. Éste supone una generalización mucho más compleja, y sin embargo su descripción necesita muchos menos detalles. Ha sido previamente expuesto, con ligeras variantes y utilizando la sintaxis de versiones anteriores del lenguaje, en [CFBSB00] y [CFBSB01]. Ésta es, no obstante, la primera presentación con la sintaxis actual de $\mathcal{P}i\mathcal{L}ar$.

La Figura 8.3 muestra la configuración $\mathcal{P}i\mathcal{L}ar$ utilizada en este ejemplo. El esquema podría expresarse de otra manera, incluso con el mismo lenguaje; la estructura podría ser mucho más simple, y hay detalles que podrían haberse ignorado. Sin embargo, se ha preferido utilizar esta descripción, ya que muestra ciertos aspectos de una manera particularmente clara.

El ejemplo describe un sistema Cliente/Servidor (*2-tier*) que evoluciona hasta convertirse en un sistema de tres niveles (*3-tier*); esto ocurre cuando el Servidor solicita la creación de un elemento Proxy que gestione sus interacciones con el Cliente. Para designar de manera sencilla esta solicitud, se asume que ésta se realiza mediante la emisión de un meta-evento o mensaje especial, h. Éste será recibido en la conexión a través del puerto correspondiente –situado en su metafaz–, momento en el que se desencadena la evolución del sistema. La Figura 8.4 muestra las distintas etapas por las que éste va pasando.

Inicialmente se dispone de dos componentes, C –Cliente– y S –Servidor–, unidos por una conexión, a la que se denomina L. Ésta se declara como tipada, por lo que es controlada por un metacomponente (arquetipo o conector) llamado Inter. Según su definición, inicialmente se reifica a L como un *enlace* simple, esto es, de tipo **link**, por lo que su comportamiento se limita

```
\component Cliente
( ( port C1 ) )

\component Servidor
( ( port S1 ) )

\component Proxy
( ( port CL | port SV )
  \constraint
  ( \rep ( CL?q; tau; SV!q;
           SV?r; tau; CL!r )
) )

\component System
( \config
  ( C:Cliente | S:Servidor |
    \bind
    ( L : Inter(C.C1, S.S1) )
) )

\component Inter
( \interface
  ( port A | port B )
  \constraint
  ( \reify R1 ( avatar : (A=B) );
    self.B?(h); \del R1;
    \reify R2 ( avatar : Tier(A,B) )
) )

\component Tier
( ( port A | port B )
  \config
  ( P:Proxy |
    \bind
    ( L1 : (A=P.CL) |
      L2 : (P.SV=B) )
) )
```

Figura 8.3: Un Sistema Cliente/Servidor [2|3]-*tier* en $\mathcal{P}i\mathcal{L}ar$

a transmitir mensajes de origen a destino.

Cuando Inter detecta que se ha recibido el evento especial –a través de su puerto B, por lo que presumiblemente procede del propio servidor S o alguno de sus metacomponentes, se destruye la anterior reificación de la conexión L, por lo que ésta deja de comportarse como un enlace. A continuación se crea una nueva relación de reificación con el tipo Tier, un conector complejo que se compone de un componente de tipo Proxy, denominado P, y dos enlaces simples que lo ligan al sistema. De este modo, el sistema ha evolucionado de una arquitectura de dos a tres extremos, creando un nuevo elemento –el proxy– en el proceso.

El dinamismo del sistema se deriva de la restricción de Inter, que puede expresarse, utilizando la notación algebraica –o abreviada– de $\mathcal{P}i\mathcal{L}ar$, como se indica a continuación:

$$\rho R_1(\alpha : (A = B)); \gamma.B?(h); \delta\, R_1; \rho R_2(\alpha : \mathrm{Tier}(A, B))$$

Esta expresión se traduce, en términos informales, como la secuencia de instrucciones siguiente, que se describirá en cuatro pasos:

1. En primer lugar, se reifica (ρR_1) a mi propio avatar (α) como un conexión simple o enlace; es decir, se crea una reificación con el el arquetipo **\link**, y se ligan los puertos ($\alpha.A, \alpha.B$) de la manera adecuada, según el orden de su declaración.
2. Después, se *espera* la recepción de un evento especial h a través del puerto B del metacomponente, esto es, de Inter. Se trata, por tanto, de un meta-evento. Se ha elegido el puerto B por suponer implícitamente que éste es el "extremo servidor", ya que en el nivel base lo es explícitamente. Sin embargo, en realidad no se ha indicado ningún tipo de restricción

sobre la conectividad del nivel meta, por lo que esto podría ser modificado con libertad, sin tener consecuencias sobre el comportamiento global del sistema.

3. Una vez haya llegado el evento, se destruye (δ) la reificación previa (R_1), de modo que la conexión ya no se comporta como un enlace simple.
4. Finalmente, se reifica de nuevo a mi avatar, en esta ocasión con el arquetipo Tier. Ha de tenerse en cuenta que esta reificación fuerza a la creación de un componente Proxy, que queda ligado a los extremos de la conexión mediante dos enlaces simples. Por tanto, esto no es simplemente un intercambio de tipos, sino que de hecho se crean y destruyen nuevos elementos; se trata, por tanto, de dinamismo de segundo nivel.

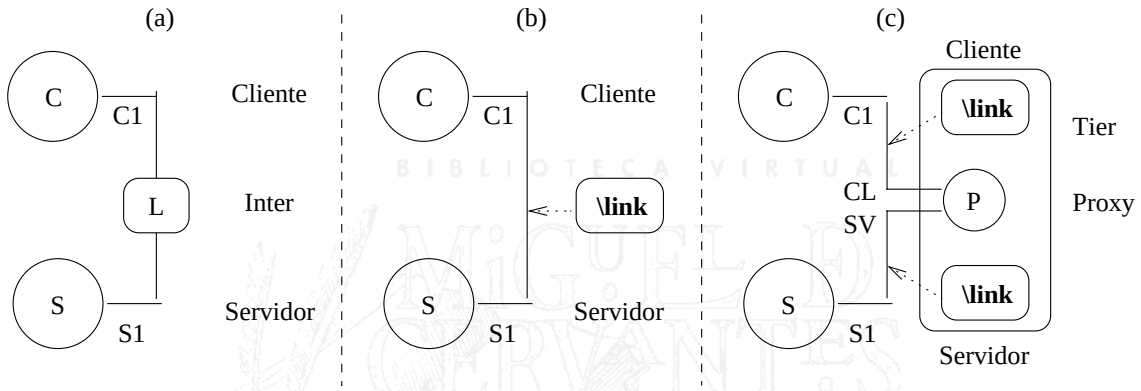

Figura 8.4: Definición Inicial y Etapas Dinámicas del Ejemplo

Obsérvese que Inter plantea una restricción que sólo se ejecuta una vez, sin plantear esquemas de replicación ni recursión. Por este motivo, la conexión comienza comportándose como un enlace y termina tomando la forma de un subsistema complejo; el propio proceso asociado a Inter termina tras esta última transformación.

Este ejemplo es muy sencillo, y tan sólo pretende mostrar cómo en $\mathcal{P}i\mathcal{L}ar$ es posible realizar las tareas dinámicas más habituales con una sintaxis bastante simple. En particular, se hace patente la potencia expresiva del concepto de *conector de nivel meta* (§6.11), dado que en este sistema Inter es el único elemento que utiliza sus capacidades reflexivas, y todo el dinamismo del sistema deriva de él. También resulta interesante comprobar cómo la estructuración en componentes permite separar incluso los detalles relativos al comportamiento, logrando con esto una mayor claridad en la especificación. Así, los tipos de conexión clásicos, ejemplificados por Tier y el propio **link**, se ocupan exclusivamente de aspectos de interacción, mientras que el conector Inter se concentra en los relativos a dinamismo, que en este caso de es tipo estructural, esto es, el identificado como segundo nivel en la sección 3.3.

8.4 Ejemplo de Los Filósofos Evolutivos

En esta sección se incluye la especificación de una posible solución para el remedo arquitectónico de una variante dinámica del clásico problema de *los Filósofos Comiendo*, descrito por primera

vez por Edsger W. Dijkstra en 1965. Como ya se ha dicho en la introducción, y como ocurre a menudo con los ejemplos clásicos de la teoría de la concurrencia, la arquitectura que aquí se describe no es, en realidad, un ejemplo de *buena* arquitectura. Tanto los componentes como los puertos tienen una granularidad muy baja, y el nivel de abstracción no resulta adecuado. Pero, en cualquier caso, resulta muy interesante.

Como es sabido, el problema de los Filósofos Comiendo describe el comportamiento de una comunidad de n filósofos sentados alrededor de una mesa redonda, que alternativamente *piensan* o *comen* spaghetti. Para poder comer, un filósofo necesita usar dos tenedores, uno con la mano izquierda y otro con la derecha; pero sólo hay n tenedores, por lo que no todos pueden comer al mismo tiempo. Una solución al problema debe proporcionar una estrategia para que los filósofos compartan los tenedores, evitando cualquier posibilidad de interbloqueo (*deadlock*) y maximizando la probabilidad de que se produzca una selección justa (*fairness*).

```
\component philosopher (
  ( port lhand |
    port rhand |
    port eat ))

\component fork (
  \interface (
    port T ))

\base-level : ( community<5> )
\component community <N:int> (
    \configuration (
      aphil[1 .. N] : philosopher |
      afork[1 .. N] : fork ;
      \for (k:1 .. N)
        ( \bind (
            aphil[k] .lhand = afork[k] .T |
            aphil[(k mod N) + 1].rhand
                                = afork[k] .T
    ))
  ))
```

Figura 8.5: El Problema de los Filósofos Comiendo: Enfoque Estático

La concepción arquitectónica más habitual de este problema [Dul92] hace uso de su naturaleza circular. El sistema se describe como un par de vectores de tamaño n, uno de filósofos y otro de tenedores. Cada filósofo se enlaza con los dos tenedores adyacentes: es decir, se crean dos *conexiones* por cada uno. De este modo, la estructura queda descrita, y lo único a desarrollar es el protocolo de comunicación, o más bien de sincronización. Éste será el que determine qué conexiones son *utilizadas* en un momento dado. Por tanto, la arquitectura es totalmente estática: no se trata de crear conexiones, sino de *activarlas* cuando sea necesario. Este enfoque es el que se muestra en la Figura 8.5; resulta muy sencillo, en especial porque se ha preferido no especificar el protocolo. En general, cualquiera de las soluciones conocidas [And91] del problema podría ser aquí aplicada, en cualquiera de sus puntos.

Sin embargo, no es esta descripción la que será tratada en mayor detalle. Con el fin de mostrar un esquema con mayor dinamismo, se utilizará a continuación un enfoque diferente. Así pues, cada vez que un filósofo coge (get) un tenedor, se *crea* una conexión; y una vez que lo libera (put), esta conexión es *destruida*. De este modo, el sistema evolucionará de manera intensiva, aunque obviamente ésta no es una solución muy eficiente al problema.

Sería muy sencillo adaptar el esquema de la Figura 8.5 para este enfoque. Bastaría con declarar únicamente los dos vectores, y crear las conexiones dinámicamente desde alguna regla.

En ese caso, la estrategia más sencilla sería cualquiera de las que recorre el anillo de filósofos y tenedores de manera iterativa, por ejemplo simulando el paso de un testigo. Sin embargo, con el objetivo de mostrar más aspectos sintácticos del lenguaje $\mathcal{P}i\mathcal{L}ar$, se ha preferido utilizar una aproximación recursiva. Ésta es la que se muestra en la Figura 8.6; aunque obviamente la complejidad del esquema ha aumentado, se considera que el problema es lo suficientemente bien conocido como para que la dificultad adicional sea asumible.

```
\component philosopher (
 ( port lhand | port rhand | port eat ))

\component fork (
 \interface ( port T )
 \constraint (
   P def= T?(get); T!(ok); Q
   Q def= T?(X); \if (X=get) (T!(no); Q)
                 \if (X=put) (P)))

\component group <n:int> (
 \interface ( port rhand | port lfork )
 \config (
   \if ( n=1 )
      ( aphil:philosopher |
        afork:fork |
       \bind ( aphil.rhand = rhand |
               afork.T = lfork ))
   \else ( aphil:philosopher |
           afork:fork |
          rest:group<n-1> |
          \bind ( rest.rhand = rhand |
                  afork.T = lfork ))
 \constraint (
   P def= \if ( n>1 )
            ( avatar.aphil.eat?()
              \if ( n mod 2 = 0)
                 ( Z; D; F )    % zurdo
              \else ( D; Z; F ))
                                 % diestro
   Z def= avatar.afork.T!(get)
          avatar.afork.T?(X)
          \if (X = ok)
            ( \new ( le:
                 ( avatar.afork.T =
                   avatar.aphil.lhand)))
          \else ( tau; Z )

   D def= avatar.rest.lfork!(get)
          avatar.rest.lfork?(X)
          \if (X = ok)
            ( \new ( ri:
                 ( avatar.rest.lfork =
                   avatar.aphil.rhand)))
          \else ( tau; D )

   F def= tau; le!(put); ri!(put)
          \del (le); \del (ri); P
 ))

\component system <n:int> (
  \config ( all:grupo<n> |
    \bind ( all.rhand = all.lfork )))

\base-level : ( system )
```

Figura 8.6: Los Filósofos Apresurados: Enfoque Recursivo

En primer lugar, se revisará la parte *estructural* de la descripción. El principal elemento recursivo es un componente compuesto denominado group, parametrizado mediante un número n que indica su tamaño. Este elemento representa a un grupo de n filósofos y tenedores puestos en secuencia, de modo que el extremo derecho es siempre la mano derecha (rhand) de un filósofo; y siempre hay un tenedor (lfork) en el extremo izquierdo. Se trata de un círculo, pero un círculo abierto; sólo cuando éste se cierra, se transforma en el sistema (system) que se pretende describir. El grupo *crece* a medida que se construye, de forma recursiva. Cada filósofo tiene dos manos (rhand, lhand), y se une al grupo trayendo consigo un tenedor (T). Por tanto, cuando un único

filósofo se sienta en la mesa y deja este tenedor a su izquierda, *sin cogerlo*, se ha creado un grupo de tamaño uno (group<1>). De este modo, los extremos del grupo son, como se había dicho, una mano derecha y un tenedor izquierdo.

El grupo crece cuando un nuevo filósofo, con su correspondiente tenedor, se sienta a la mesa. Lo hace de modo que siempre se sitúa a la izquierda del grupo existente; así, los extremos son, nuevamente, un tenedor a la izquierda, que acaba de incorporarse, y la mano derecha de un filósofo, concretamente el primero que se sentó. Como se ha dicho, el componente system tan sólo cierra el círculo, una vez que se ha formado un grupo del tamaño previsto. Lo único que ha de hacer, en este sentido, es dejar el último tenedor depositado dentro del alcance de la mano derecha del primer filósofo. Tal vez convenga resaltar que hasta este momento *sólo se han hecho enlaces jerárquicos*. Es decir, lo único que hace este esquema es situar a todos los puertos al mismo nivel, independientemente de la profundidad de la definición recursiva. De este modo, todos los tenedores pueden ser usados de inmediato.

Es preciso ahora especificar el protocolo utilizado. De entre todas las soluciones propuestas en la literatura, se ha elegido la conocida como *Los Filósofos Apresurados* [Lyn96], una de las más sencillas. Consiste en dividir a los filósofos en *diestros* y *zurdos*, lo que aquí significa estrictamente que siempre coge en primer lugar el tenedor derecho o izquierdo, respectivamente. Si dicho tenedor está ocupado, espera a que quede libre; no intenta coger el otro tenedor ni renuncia a comer; de aquí el adjetivo *apresurado*. Es fácil demostrar que cuando hay al menos *un* filósofo de cada tipo, no es posible producir una situación de bloqueo mutuo. En el sistema de la Figura 8.6, cada filósofo par es zurdo, y los impares son diestros; por tanto, la condición necesaria se mantiene siempre, y el sistema podrá evolucionar indefinidamente.

La descripción de la parte dinámica resulta ahora prácticamente trivial. Así, una vez que hay al menos dos tenedores, se espera a que un filósofo sienta el impulso de comer; cuando es par, es zurdo, y por tanto intenta comenzar por el tenedor izquierdo (Z). En otro caso, hace exactamente lo contrario. El acto de coger un tenedor se simboliza como un mensaje get: si se recibe una respuesta afirmativa, se crea un enlace entre la mano y el tenedor; si es negativa, el filósofo espera un momento (**tau**) y lo intenta de nuevo. Hace después lo mismo con la otra mano. Una vez que se han conseguido los dos tenedores, los usa el tiempo preciso (**tau**), para liberarlos a continuación. Esto se hace advirtiendo primero al tenedor con un mensaje put, y destruyendo después el enlace.

En esta definición hay dos aspectos especialmente curiosos. En primer lugar, puede observarse que todo este comportamiento no se define sobre un filósofo, sino sobre un *grupo*. Esto es debido a que es el grupo el que sabe si a su filósofo le corresponde una posición par o impar, puede así decidir si es zurdo o diestro. Ocurre pues que el grupo impone el comportamiento a su filósofo *desde fuera*, lo que no plantea problemas de concurrencia, pues gracias a la recursión hay un grupo por cada filósofo. En segundo lugar, y relacionado con esto, puede verse que el mensaje de captura (get) es recibido por el conector *antes* de crearse el enlace; pero en realidad no es el filósofo quien lo envía, sino el grupo.

En la Figura 8.7 se considera una extensión del problema; en este caso, el objetivo es mostrar cómo una descripción $\mathcal{P}i\mathcal{L}ar$ puede ser alterada sin ser apenas alterada, gracias a la reificación. La extensión descrita permite la incorporación dinámica de nuevos filósofos a la comunidad; de este modo, aumenta el grado de dinamismo del sistema. Se trata de la variante del problema conocida como *Los Filósofos Evolutivos* [KM90], que fue introducida explícitamente en el contexto

de la Configuración, y requiere de manera explícita dinamismo de segundo nivel (estructural). No se considera aquí la otra mitad del problema, esto es, la eliminación de filósofos, ya que resulta especialmente farragosa en el esquema recursivo utilizado. En una organización iterativa como la descrita en la Figura 8.5, sin embargo, esto sería muy sencillo.

```
\component plus <m:int> (
   ( port add )
   \constraint (
     \rep (add?();
     \if ( avatar.n = m )   % para recursión
       ( \new avatar.avatar. aphil:philosopher ;
         \new avatar.avatar. afork:fork ;
         ( \new avatar.avatar. ri:
             ( avatar.avatar. aphil .rhand =
               avatar.avatar.rhand ) |
           \new avatar.avatar.le:
             ( avatar.avatar.afork .T =
               avatar.avatar. lfork  )) ;
           avatar.n!(n+1); m!(m+1) )))

\component system <n:int> (
   ( port add )
   \config ( all:group<n> |
     \bind ( all .rhand = all . lfork  ))
   \constraint (
     \reify R1 ( group :plus<n>(add) )))
```

Figura 8.7: Los Filósofos Apresurados, versión Evolutiva

La Figura sólo introduce un cambio con respecto a la descripción original, concretamente en el componente system. En realidad, éste es prácticamente idéntico; tan sólo se añaden un puerto (add), por el que se recibirá la petición de incorporación de un filósofo, y una restricción, que crea una reificación entre el arquetipo group y un nuevo tipo parametrizado, plus. Por tanto, éste se define como un meta-metacomponente de cada instancia de grupo. Debe tenerse en cuenta que group es también un tipo paramétrico, lo que significa que se define un arquetipo para cada valor de n. Sin embargo, cada uno de estos arquetipos tiene únicamente una instancia, por lo que la complejidad disminuye en gran medida.

Además, el comportamiento de plus se define de modo que sólo afecta al grupo de mayor tamaño (n), es decir, al que describe al sistema completo. Este dato es, precisamente, el que el tipo plus recibe como parámetro, y su único objetivo es controlar la recursión. Así pues, cada vez que se recibe una petición en add para incluir a un nuevo filósofo, se selecciona el grupo de mayor tamaño –o, más bien, su arquetipo– y se crean, *dentro de éste, pero en el nivel base*, un nuevo filósofo, con su tenedor, y los enlaces jerárquicos necesarios con el grupo. Finalmente, se modifican las variables (m, n) que representan a los parámetros de ambos tipos; de este modo, se expresa que el tamaño ha aumentado, lo que afectará a las repeticiones sucesivas.

El detalle más importante es que todos los nuevos elementos se han creado dentro del avatar seleccionado: por tanto, se les aplican *las restricciones ya existentes*, lo que asegura un comportamiento global consistente. Por supuesto, ésta es de nuevo la estrategia más complicada, elegida a propósito para mostrar la creación de subinstancias en un avatar. El mismo efecto se hubiera podido lograr de manera mucho más sencilla creando una nueva instancia del tipo tras incrementar su parámetro. La complejidad de este enfoque deriva del hecho de que un arquetipo recursivo como group se despliega de manera descendente, mientras que aquí ha de *crecer* de manera ascendente, manteniendo la conexión con la construcción anterior.

De este mismo sistema existe una reformulación equivalente, aunque ligeramente más sen-

```
\component plus <m:int> (
  ( port add )
  \constraint (
    \rep (add?();
    \if ( n = m )   % para recursión
      ( \new avatar. aphil:philosopher ;
        \new avatar. afork:fork ;
        ( \new avatar. ri:
            ( avatar. aphil . rhand =
              avatar.rhand ) |
          \new avatar.le:
            ( avatar. afork . T =
              avatar. lfork  )) ;
          n!(n+1); m!(m+1) )))

\component system <n:int> (
  ( port add )
  \config ( all:group<n> |
    \bind ( all . rhand = all . lfork  ))
  \constraint (
    \reify R1 ( all  :plus<n>(add) )))
```

Figura 8.8: Los Filósofos Apresurados, versión Evolutiva (simplificada)

BIBLIOTECA VIRTUAL

cilla, publicada como parte del artículo referenciado como [CFBSB02b], e incluida aquí como Figura 8.8. Como puede verse, en ésta el metacomponente plus no reifica al arquetipo group, sino a la propia instancia all. De este modo, se sitúa como un rohatar directo, en lugar de como un meta-rohatar. La restricción es completamente idéntica, excepto en el uso de los prefijos **avatar**, que reflejan este cambio de meta-nivel: donde había dos, ahora hay solamente uno. De hecho, en la nueva descripción, todos los prefijos de este tipo pueden omitirse sin mayores consecuencias, ya que se refieren en todos los casos a un avatar directo.

Obsérvese que el comportamiento del sistema es, en líneas generales, exactamente el mismo. En el caso anterior (Figura 8.7), el proceso de un avatar era influido, de manera concurrente, por el de su arquetipo group, y el de su meta-arquetipo plus. En éste (Figura 8.8), tanto group como plus son rohatares ubicados al mismo nivel, pero influyen concurrentemente sobre el proceso del avatar, del mismo modo que antes. En realidad sólo hay una diferencia: el primero modifica el parámetro n del arquetipo, mientras que el segundo modifica el de la instancia. Esto afecta ligeramente al comportamiento definido por la restricción P de los componentes group, en el sentido de que se altera el orden según el que un filósofo se crea como zurdo o diestro. Pero esto no afecta, en cualquier caso, a la condición requerida por el esquema de *los Filósofos Apresurados*, esto es, que haya al menos uno de cada tipo. Por lo tanto, el comportamiento final del sistema completo mantiene los postulados de dicha solución, y por tanto la configuración descrita por esta especificación está igualmente libre de *deadlock*.

Puede comprobarse, en definitiva, que la modificación *dinámica* presentada en primer lugar es la más compleja. No obstante, ambas son igualmente interesantes, puesto que la primera muestra el modo en que se puede alterar el comportamiento de un avatar desde el nivel meta-meta, mientras que la segunda indica cómo se entremezclan las influencias de dos metacomponentes concurrentes sobre una misma entidad.

Debe tenerse en cuenta que el ejemplo no es ni mucho menos tan complejo como parece. Como ya se ha indicado, se podría haber tomado una decisión distinta en muchos lugares, con lo que se hubiera obtenido una solución mucho más sencilla. Si se ha elegido este enfoque, ha sido con el fin de mostrar otros modos de utilizar la sintaxis de $\mathcal{P}i\mathcal{L}ar$.

8.5 El Modelo Linda: Espacios de Tuplas

Con el fin de apoyar la afirmación de que $\mathcal{P}i\mathcal{L}ar$ puede ser también utilizado como un lenguaje de coordinación, la especificación $\mathcal{P}i\mathcal{L}ar$ de la Figura 8.9 describe una arquitectura que simula el paradigma del Espacio de Tuplas (§3.11.1), elegido como ejemplo arquetípico de un modelo de Coordinación. En este punto, se considerará únicamente el modelo Linda original [CG86], sin incluir ninguna de sus muchas extensiones. Por tanto, se dispondrá de un único espacio de tuplas, común a todos los agentes y de estructura plana; y se utilizarán solamente las primitivas básicas **in**, **rd**, y **out**[2]. En principio, se asume que la estructura de una tupla consta de una cabecera y un contenido, aunque este detalle resulta indiferente.

Hay varios modos de desarrollar este modelo en $\mathcal{P}i\mathcal{L}ar$, y algunos de ellos son incluso más simples. Por ejemplo, el espacio de tuplas podría ser descrito como un componente de nivel base, para luego conectarlo explícitamente con cada agente del sistema. Aunque esta aproximación es simple, no resulta inhabitual: de hecho, es muy frecuente en las versiones del modelo orientadas al objeto [FHA99]. Sin embargo, aquí se utiliza un enfoque distinto, más fiel al principio de *comunicación generativa* que constituye la esencia de Linda: los agentes no necesitan saber hacia dónde van o de dónde vienen las tuplas que manipulan.

El ejemplo muestra de manera sencilla, y en un entorno realista, algunas de las características de $\mathcal{P}i\mathcal{L}ar$ más significativas, tales como el modo en funciona la reificación, y su relevancia; el modo en que se despliega la estructura de meta-niveles; la diferencia entre metacomponente y componentes meta, o el aspecto que puede tener un conector de nivel meta.

En una primera lectura, puede verse que hay cinco definiciones de componente. Dos de éstos –System, SysTS– son componentes compuestos, ya que su declaración incluye una configuración. Los otros tres –Store, Triple, AgTS– son componentes primitivos: sólo se describe su interfaz, opcionalmente acompañada de restricciones. Como se verá, sus roles en el sistema serán sin embargo muy diferentes. Como ya se ha dicho, una descripción arquitectónica no incluye la especificación de la funcionalidad de los componentes primitivos, sino sólo sus restricciones de interacción; pero, para explicar mejor el ejemplo, se esbozará brevemente con el significado con el que se pretende utilizarlos.

Store Describe la interfaz de un almacén de datos ideal, por ejemplo un sistema de base de datos. El puerto insert recibe los datos que debe almacenar; query recibe un patrón que describe los datos se desea recuperar, que según esto son seleccionados y emitidos; y delete recibe un patrón que identifica qué contenido se desea eliminar. Se asume que hay algún tipo de control de concurrencia interno.

AgTS Es el arquetipo de cualquier agente situado *dentro* del ámbito de un espacio de tuplas, Los tres puertos tienen el significado habitual en el modelo: out envía una tupla hacia el espacio, rd la lee de dicho espacio, e in_ la lee también, pero eliminándola en el proceso. Las reglas indicadas en la definición se revisan más adelante.

[2]Como es usual en la práctica, no se tratará la primitiva **eval**, ya que resulta innecesaria en un esquema concurrente y modular como el que se está considerando, donde se pueden crear entidades activas (componentes) con entera libertad, incluso desde el propio ADL.

```
deftype Tuple = { hd, data }

\base−level : ( System )

\component System (
 \config (
   Agent1:AgTS |
   Agent2:AgTS ))

\component SysTS (
 \config (
   DB:Store | AgTS |
   \bind (
    AgTS.out = DB.insert |
    AgTS.rd  = DB.query |
    L1:Triple(  DB.query |
      DB.delete | AgTS.in_ ))
 ))

\component Store (
 ( port insert | port query |
    port delete ))

\component AgTS (
 ( port in_:Tuple |
   port rd: Tuple |
   port out:Tuple )
 \constraint (
  TP def= ( OP | RP | IP )
  OP def= \bang ( avatar.out?T;
      self.out!T )
  RP def= \bang ( avatar.rd?A;
      self.rd!A; self.rd?T
      avatar.rd!T )
  IP def= \bang ( avatar.in_?A;
      self.in_!A; self.in_?T
      avatar.in_!T )
  ))

\component Triple (
 ( port get | port dele |
   port res )
 \constraint (
  \bang ( res?Q; get!Q; get?A;
    ( res!A | dele!A ) )))
```

Figura 8.9: Modelo de Espacio de Tuplas: una descripción en $\mathcal{P}i\mathcal{L}ar$

Triple Define una conexión con tres extremos; el comportamiento previsto es descrito claramente en su restricción. Si se recibe una petición Q en el puerto res, la reenvía hacia get. Se espera entonces recibir una respuesta A, que será entonces enviada de nuevo a través de res y dele. Se asume que el primer dato es un resultado, mientras que el segundo es una orden para eliminar este dato del almacén. Este proceso se repite indefinidamente, ya que el proceso está replicado (**bang**).

El punto de partida para deducir la estructura es la etiqueta **base-level:**, que como se sabe indica qué componentes están en el nivel base, el más bajo de la jerarquía de meta-niveles. En la Figura, este nivel consiste solamente en el componente System, que a su vez es una composición de Agent1 y Agent2, instancias ambos de AgTS. El subsistema compuesto es notablemente sencillo: no se exporta ningún puerto hacia el exterior, ni se realiza conexión alguna; los agentes simplemente "están allí". En cualquier otro lenguaje, se comportarían como elementos aislados, desvinculados del resto de la arquitectura. Aquí, quedarán ligados al sistema, y entre sí, a través de las reificaciones y las relaciones establecidas en el nivel meta.

Pero su arquetipo, AgTS, es también un metacomponente suyo. Más aún, la restricción que define utiliza tanto la palabra **avatar** como la palabra **self**, lo que significa que en este caso se dispone de una verdadera meta-entidad *activa*. En resumen, esto significa que no solamente tenemos dos componentes en el nivel base, sino además *uno* en el nivel meta, vinculado a ellos por

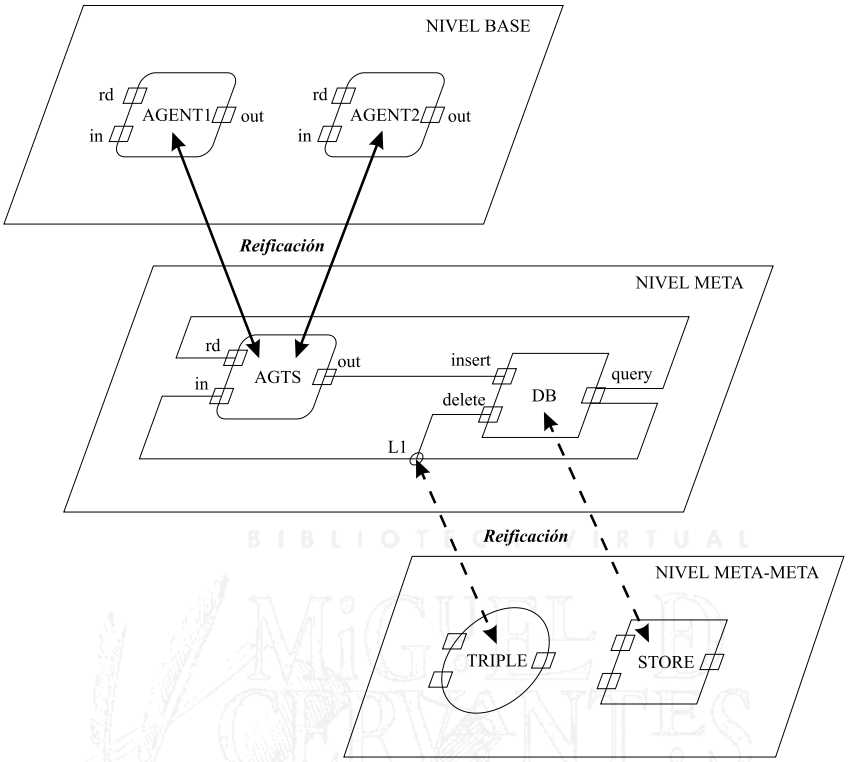

Figura 8.10: Representación Gráfica del Modelo de Espacio de Tuplas

reificación. Todos ellos tienen tres puertos, que como se sabe son completamente independientes. Esto significa que el puerto out de Agent1 no tiene nada que ver con el puerto out de AgTS: el metacomponente utiliza este último para interactuar con otros componentes de nivel meta. Sin embargo, en este caso concreto interesa que esos dos puertos estén conectados, por lo que se imponen las restricciones que lo aseguran.

Así pues, las reglas de AgTS asocian los puertos out, rd y in_, respectivamente, con sus homólogos en el meta-nivel. Por ejemplo, la regla OP de AgTS puede leerse desde el lado meta como "cada vez que un avatar reciba una tupla T en su puerto out, se envía la misma tupla a través de mi propio out"; o, desde el lado del avatar, "cada vez que reciba una tupla T en mi puerto out, se envía la misma tupla a través del out de mi metacomponente". En resumen, AgTS captura cada mensaje de cada avatar, y lo envía a través de sus propios puertos, es decir, los de su metafaz. De este modo, esta última queda vinculada a la interfaz de los avatares.

Es por tanto obvio que AgTS interactúa con otros componentes del nivel meta. Al examinar el otro componente compuesto del esquema –SysTS–, puede verse que la configuración contiene una instancia no tipada, AgTS. Se trata del propio arquetipo, usado como un componente (de nivel meta); es decir, lo que se denominó un *tipo reificado* en la sección 6.4. Por lo tanto, dado que AgTS es un metacomponente y SysTS lo contiene, este último está también en el nivel meta.

De hecho, *es* el meta-nivel completo, y consta de dos componentes: el propio AgTS y un

almacén, denominado DB. Dos de sus puertos están unidos mediante un enlace simple, mientras que el resto están ligados mediante una conexión ternaria, $L1$, de tipo Triple. Esto proporciona suficiente información para poder deducir el comportamiento resultante: las tuplas enviadas a través de out se insertan en el almacén; rd recibe consultas, en la forma de anti-tuplas, y devuelve el resultado correspondiente; y in_ hace lo mismo, pero borrando la tupla utilizada –leída– del almacén. De este modo el comportamiento del sistema ha quedado descrito. La estructura global se resume gráficamente en la Figura 8.10.

Volviendo al arquetipo Triple, puede verse que también actúa como un componente del nivel meta-meta, ya que reifica a la conexión $L1$, situada en el meta-nivel. Pero la restricción asociada sólo utiliza nombres de avatar, por lo que en realidad no actúa como una meta-entidad activa. En realidad, en este punto no parece que haya mucha diferencia con los conectores "clásicos" [All97, SG96, Sha94], por lo que aquí el término *conector de nivel meta* resulta quizás engañoso. Sin embargo, se podrían añadir restricciones que lo hicieran comunicarse con otros componentes (en el nivel meta-meta), creando de este modo un *conector compuesto*.

En resumen, el sistema tiene tres niveles. El nivel base consiste en un módulo compuesto System, formado por la composición de dos agentes, Agent1 y Agent2, ambos de tipo AgTS, que no se conocen entre sí. El nivel meta consiste en otro compuesto, SysTS, en este caso de dos componentes distintos: un almacén DB, de tipo Store, y el tipo reificado AgTS. El primero es un componente meta, mientras que el segundo es un metacomponente de los agentes. Ambos quedan conectados mediante dos enlaces y una conexión tipada –de tipo Triple–. Finalmente, el nivel meta-meta contiene los dos tipos mencionados: Store y Triple. Ambos son componentes inactivos, por tanto pueden ser ignorados sin mayor problema.

En resumen: cada vez que un agente cualquiera emita (out) una tupla, ésta será capturada por AgTS y almacenada en DB; y cada vez que un agente cualquiera intenta leer (rd, in_) una tupla, la consulta (anti-tupla) será capturada y enviada a DB, y su resultado será devuelto al agente. De este modo, Agent1 puede enviar tuplas a Agent2, así como recibirlas de él, sin necesidad de conocerlo en absoluto, ya que toda la comunicación (generativa) se realiza en el nivel meta. Esto tiene una interpretación clara y natural: los agentes del nivel base son definidos por el usuario, mientras que el nivel meta describe la implementación del sistema del Espacio de Tuplas.

Por supuesto, este ejemplo podría ser mucho más complejo; se podría considerar el formato de la tupla, o incluso se podría especificar el comportamiento del almacén, mediante el Lenguaje Dinámico, por ejemplo mediante una estructura de datos de tipo cola [Mil89].

En cualquier caso, el objetivo principal de este ejemplo es mostrar en detalle el uso del lenguaje, así como la facilidad con la que los modelos de coordinación pueden ser descritos con el mismo. El resultado final muestra una gran coherencia, lo que dota al esquema de cierta elegancia conceptual.

8.6 Sistema con Componentes Envolventes

En la sección 6.8 se la sintaxis se introdujo una extensión específica para la especificación de envolventes. Con el objetivo de examinar mejor este caso, se introduce aquí un ejemplo concreto.

El ejemplo de la Figura 8.11 muestra dos componentes $\mathcal{P}i\mathcal{L}ar$ diseñados específicamente

para actuar juntos, de manera que uno de ellos reifica al otro y lo *envuelve*, controlando sus interacciones. Es decir, en la definición de ambos se ha establecido un protocolo concreto, de modo que cuando se crea una reificación entre ellos, el avatar sólo puede interactuar *a través* de su metacomponente, que por tanto actúa como un envolvente. Este esquema es obviamente similar al de un modelo de capa de cebolla, en el sentido de que la relación entre los dos elementos se establece en el nivel meta de uno de ellos.

En este caso, el protocolo ha sido concebido para manipular las interacciones entre los dos niveles de manera explícita. Por separado, cualquiera de los componentes carece de sentido; combinados, crean un binomio en el que toda comunicación hacia o desde el avatar será previamente validada, y posiblemente incluso alterada, por su envolvente del nivel meta.

El sistema de la Figura muestra una arquitectura que consta tan sólo de estos dos elementos, con el objetivo de simplificar la descripción del ejemplo. Por tanto, se declara simplemente una instancia (A) del tipo a envolver (TA), y a continuación ésta misma se reifica como un avatar del tipo envolvente (TWrap). A partir de este momento, las interacciones de A vienen determinadas por la composición paralela de las dos partes del protocolo.

```
\component TA
( ( port P )
  \constraint
  ( chan u,d;
    Al def= P?x; u!x; d?z;
       tau (z); u!z; d?k;
       P!k; Al
) ) /- Comunicación síncrona -/

\component TWrap
( ( port W )
  \constraint
  ( Wl def= avatar.u?x; tau (x);
       avatar.d!x; avatar.u?z;
       tau (z); avatar.d!z; Wl
) )

\component Sys
( \config ( A:TA )
  \constraint
  ( \reify R1(A:TWrap)
) )
```

Figura 8.11: Envolvente con Protocolo Explícito en $\mathcal{P}i\mathcal{L}ar$

La instancia tiene un único puerto público (P), por el que se comunicará con el exterior. Sin embargo, define además dos canales internos (u, d), que serán cruciales para el establecimiento del protocolo entre niveles. Toda comunicación se inicia siempre en el avatar, ya que es el único elemento conocido en el exterior, y se pretende que el envolvente resulte transparente.

La secuencia de interacciones definida por el protocolo resulta entonces evidente: inicialmente, se recibe una comunicación a través del puerto P. Inmediatamente, el avatar envía el mensaje recibido a través del canal interno u, y espera algún tipo de respuesta en el otro canal, d. Al mismo tiempo, el metacomponente intenta leer en el canal u de su avatar, con lo que captura el mensaje recibido. Lo manipula del modo que desee (**tau**), y a continuación lo envía, modificado o no, a través del canal d de su avatar.

El avatar, que estaba esperando, reanuda su actividad y utiliza por fin la información recibida (**tau**) del modo previsto en su funcionalidad. Una vez procesada, intenta emitir un mensaje de respuesta; pero éste habrá de pasar, de nuevo, a través del envolvente. Por ello, se repite el esquema anterior, de modo que se envía la información por el canal interno u, y el metacompo-

nente la captura, procesa y reenvía en el canal d. Allí el avatar la recibe otra vez, modificada o no, y puede finalmente producir una salida hacia el exterior, a través del puerto P.

Puede parecer, y con cierto motivo, que esta estructura es demasiado rígida. De hecho, funciona exactamente al revés que un envolvente habitual: normalmente, éstos se diseñan para adaptar el funcionamiento e interacción de un módulo ya existente. En cambio, aquí el proceso interno ha de encargarse de emitir los datos en un lugar dado, esperando a que el metacomponente los capture. Sin embargo, se ha definido un sistema totalmente equivalente: independientemente de quién tome un papel activo en un momento dado, el flujo de datos es exactamente el mismo. De hecho, muchos sistemas reflexivos utilizan internamente una estrategia similar; la única diferencia real es que en este caso el protocolo se ha hecho explícito.

Se debe tener en cuenta, además, que la declaración del avatar es totalmente abstracta. A partir de ella, es evidente que éste no puede funcionar de manera aislada, y que necesita el concurso de otro componente; por tanto, lo que en realidad se está especificando es el esquema de interacción que el avatar *espera* tener con su envolvente. Esto, en definitiva, no es tan extraño: tan sólo se está estableciendo el *protocolo* (Mcp) entre los dos niveles, de modo que la comunicación se hace posible. No es más que lo mismo que se hace en cualquier interacción convencional, al especificar el protocolo entre dos componentes, o en un conector; la única diferencia es que la interacción no se produce a través del interfaz, sino por acceso directo (*caja gris*) a los canales internos del avatar. El metacomponente definido por el tipo TWrap debe verse, más bien, como una declaración de *interfaz*. Cualquier componente que respete el protocolo establecido, sea cual sea su funcionalidad, podrá actuar como un envolvente de una instancia de TA.

Sin embargo, tampoco se está sugiriendo que el sistema descrito sea especialmente adecuado. No obstante, lo único que se pretendía en este punto era expresar *cómo* podría afrontarse la descripción de un protocolo de este tipo en $\mathcal{P}i\mathcal{L}ar$; como era de esperar, sólo es posible por el hecho de que un metacomponente puede manipular cualquier elemento interno del avatar con total libertad, esto es, por el interfaz privilegiado que proporciona la reificación.

Los dos mayores inconvenientes del esquema descrito son su especificidad y, sobre todo, su falta de extensibilidad. El primer aspecto ya ha sido considerado, pero a pesar de todo lo dicho, es innegable que sería preferible especificar el avatar de manera independiente, y remitir los detalles reflexivos al metacomponente. El segundo aspecto hace referencia al hecho de que el esquema es difícilmente generalizable a más de dos niveles: prácticamente cualquier cambio exigiría el rediseño del protocolo completo.

Es por este motivo, y por lo habitual de este modelo, por lo que $\mathcal{P}i\mathcal{L}ar$ proporciona una sintaxis especializada para la definición de envolventes (§6.12.1). Una descripción de este tipo sigue manteniendo un estrecho control sobre los detalles del Mcp, pero abstrae los detalles triviales, y soluciona los dos problemas citados: en primer lugar, el avatar puede definir su protocolo con total libertad, sin considerar que si va a ser *envuelto*. En segundo lugar, el esquema resultante será en general mucho más fácil de extender a un número indefinido de niveles, por lo que esta vez se obtiene un verdadero esquema de capas de cebolla (§4.6).

La Figura 8.12 describe, pues, un sistema equivalente al ya examinado, pero utilizando la sintaxis extendida. El protocolo es ahora implícito, por lo que la especificación resulta de hecho mucho más sencilla: ahora no es preciso el uso de canales auxiliares, e incluso las restricciones del metacomponente adquieren una forma más legible.

```
\component \wrap TA
( ( port P )
  \constraint
  ( AI def= P?x; tau (x); P!x; AI
) )

\component TWrap
( ( port W )
  \constraint
  ( WI def= avatar.P?x; tau (x);
        \shift P(x); \letin P;
        \catch P(z); tau (z);
        \letout P; avatar.P!z; WI
) )

\component Sys
( \config ( A:TA )
  \constraint
  ( \reify R1(A:TWrap)
) )
```

Figura 8.12: Envolvente con Sintaxis Especializada en $\mathcal{P}i\mathcal{L}ar$

Como puede verse, la declaración del componente-base ya sólo ha de considerar su propia funcionalidad. Así pues, recibe un dato por su puerto, lo procesa, y emite una respuesta. El único detalle especial de la definición no se encuentra en sus restricciones, sino en su cabecera, señalada con el cualificador **wrap**. Tal como se estableció en la sintaxis (§6.12.1), esto indica que el componente es *interior* o *apto para ser envuelto*, lo que significa que cada una de sus interacciones con el exterior ha de ser previamente autorizada por un envolvente.

El protocolo entre los niveles, por tanto, depende solamente de las reglas especificadas en el metacomponente. De hecho, ahora es él quien comienza la actividad a partir de las entradas del avatar al que envuelve, ya que éste está inicialmente incomunicado. Así, la primera acción del envolvente es, precisamente, recoger la entrada recibida en el puerto P del avatar. Obsérvese que éste también intenta leer en ese puerto, pero no recibe nada; no ha sido aún autorizado, y por tanto no hay competencia por el dato. El envolvente no sólo lee el mensaje, sino que lo consume; luego lo procesa, modificándolo o no, y por fin vuelve a depositarlo (**shift**) en el puerto P del avatar, donde queda disponible para una nueva lectura. Sólo después de esto se autoriza (**letin**) una recepción. Esta comunicación inicia la actividad del avatar; por tanto, lo que éste lee en su puerto es el dato previamente procesado por el envolvente.

El componente interior manipula este dato (**tau**) e inmediatamente intenta enviarlo hacia el exterior. Sin embargo, no ha sido aún autorizado, por lo que esta salida es capturada (**catch**) por el envolvente. Éste procesa el dato capturado, modificándolo si es preciso, y en este momento autoriza (**letout**) la salida en P. Ahora bien, el dato enviado por el avatar ya ha sido consumido; por tanto, el que sale hacia el exterior –por este mismo punto– es el que emite el propio envolvente. El proceso vuelve a comenzar, recursivamente, en ambos niveles.

En un auténtico modelo de capas de cebolla, una nueva reificación sobre el metacomponente crearía otro envolvente sobre éste, que podría ser envuelto a su vez, repitiendo el esquema de manera indefinida. Sin embargo, esta generalización a varios niveles no resulta aquí obvia. En un sistema convencional, este recubrimiento múltiple funciona porque los puertos de cada capa *sustituyen* y *preceden* a los del elemento interior; aquí, el envolvente accede directamente a los puertos del avatar. Si un meta-metacomponente intentase hacer lo mismo, los dos competirían de manera *no determinista* por leer en dicho puerto.

En principio, esto es consciente: al igual que antes, se desea poder establecer cualquier

protocolo concurrente entre niveles. Sin embargo, la posibilidad de realizar un número indefinido de superposiciones resulta también interesante. En este caso, no se necesita ninguna sintaxis especial, sino un esquema de interacción concreto, que consiste en *ligar entre sí* las entradas de distintos niveles.

WI **def**= **self**.P?x; **avatar**.P?z; \ **shift** P(x+z); **letin** P ; ...

o, equivalentemente:

$$WI \stackrel{\text{def}}{=} \gamma.P?(x); \alpha.P?(z); P \downarrow (x+z); \Pi P; \ldots$$

Figura 8.13: Entradas a Nivel Base Ligadas por el Nivel Meta

Debe tenerse en cuenta que, si se declara un metacomponente como componente interior, lo que queda envuelto es su interfaz en el nivel meta, esto es, su *metafaz*. Por tanto, lo que queda bloqueado es su propia entrada, sin que esto afecte a sus interacciones en el nivel base. Resulta obvio, por tanto, que para lograr un comportamiento similar al del modelo, cada recepción en el nivel base ha de ser precedida por una interacción en el nivel meta. De este modo se obtiene la secuencia deseada entre las entradas, lo que hace posible la generalización a un número indeterminado de niveles, o de *capas* de cebolla.

Considérese que en el esquema de la Figura 8.12, la restricción WI se define del modo indicado en la Figura 8.13, donde se expresa en las dos notaciones del Lenguaje Dinámico. Se recibe una entrada en el nivel meta, y a continuación se captura otra en el nivel base: lo que se reenvía hacia el interior es la suma de ambas entradas, y a continuación se autoriza una lectura en el puerto. Es evidente que si un componente similar reificase a éste, se podría situar como una tercera capa; y este proceso puede repetirse de manera indefinida.

A partir de este punto, las manipulaciones que pueden hacerse con un componente envuelto resultan obvias. La combinación de entidades de ambos tipos puede dar lugar a esquemas muy complejos, pero cuya funcionalidad está, no obstante, perfectamente determinada. El único aspecto negativo de esta sintaxis es que un componente ha de ser declarado como *envuelto* de antemano, incluso cuando no ha sido aún reificado por un envolvente. Pero, dado que su semántica depende del modo en que se traducen las interacciones (§7.5), una instancia dada no puede pasar dinámicamente de una categoría a otra, por lo que esta limitación está justificada. Por otra parte, el esquema descrito es lo bastante flexible como para poder describir cualquier protocolo entre niveles, esté o no basado en un esquema de envolventes reflexivos.

8.7 Conclusiones

A lo largo de este Capítulo se han descrito varios ejemplos de descripción de arquitecturas en $\mathcal{P}i\mathcal{L}ar$. Éstos han sido seleccionados con el objetivo de ilustrar algunas de las características esenciales del lenguaje, tales como la definición de reificaciones, múltiples meta-niveles, o conectores de nivel meta, o el modo en que las restricciones de un metacomponente influyen sobre los elementos situados en niveles inferiores. Se ha mostrado, además, el modo en que se especifican

los aspectos convencionales, tanto estructurales como de comportamiento, asociados a cualquier Adl existente –incluyendo algunos menos habituales, como la recursión–.

El Capítulo comienza con la descripción de un sistema estático (§8.2), que sirva como una base adecuada para la comparación de $\mathcal{P}i\mathcal{L}ar$ con un Adl convencional, no necesariamente dinámico. Tras detallar brevemente una posible variante dinámica de dicho sistema, se pasa a la descripción de una serie de ejemplos más precisos. En el primero (§8.3) la evolución del sistema se desencadena desde una conexión, reificada por un sencillo conector de nivel meta (§6.11) asociado a un proceso finito que produce un comportamiento transitorio.

En el segundo (§8.4), se revisan varias formulaciones, estáticas y dinámicas, del clásico problema de *los Filósofos Comiendo*, haciendo especial hincapié en aspectos tales como las configuraciones iterativas y recursivas, las definiciones parametrizadas, el establecimiento dinámico de topologías de comunicación, o el modo en que una especificación ya terminada puede ser extendida *a posteriori* desde meta-niveles superiores.

En el tercero (§8.5), sin embargo, se opta por la descripción, mucho menos abstracta, de un Modelo de Coordinación, en concreto el de Espacios de Tuplas. Al tratarse de un sistema más realista, se mezclan con facilidad cuestiones tales como la ruptura del encapsulamiento, la complejidad que pueden alcanzar los conectores de nivel meta o los modos de interacción entre elementos de distinto nivel. El dinamismo es aquí menos obvio, ya que se trata simplemente de dinamismo de *primer* nivel (§3.3); pero sin embargo la interacción se realiza sin utilizar ninguna conexión-base. En cambio, la meta-arquitectura originada por las reificaciones resulta especialmente clara, desarrollándose a lo largo de tres niveles perfectamente definidos; su estructura es fundamental para determinar el comportamiento del sistema descrito.

Finalmente, en el último ejemplo (§8.6), se discuten en detalle las dificultades asociadas a la definición de *envolventes* reflexivos con la sintaxis básica de $\mathcal{P}i\mathcal{L}ar$, para mostrar luego las distintas alternativas que surgen de la aplicación de la sintaxis específica incorporada con este fin al núcleo del lenguaje. En la última etapa de este ejemplo se muestra, de hecho, el esquema básico que describe a un auténtico Modelo de Capas de Cebolla, que era precisamente uno de los objetivos secundarios planteados durante la definición del Adl.

En resumen, puede decirse que los distintos ejemplos desarrollados a lo largo de este Capítulo ilustran de manera significativa los posibles modos de utilizar el lenguaje $\mathcal{P}i\mathcal{L}ar$, y muestran que éste es perfectamente capaz de acometer la descripción de sistemas muy diversos, con distintos grados de complejidad, e incluso de un modo más sencillo que otros Adls. Más aún, puede afirmarse que la estructura de meta-niveles proporcionada es tan flexible que hace posible, en líneas generales, prácticamente cualquier tipo de descripción arquitectónica, incluso ante propuestas paradójicas[3]. En todo caso, el examen de los resultados de este Capítulo refuerza la suposición de partida de esta tesis, esto es, que la integración de reflexión en Arquitectura de Software no es sólo interesante, sino incluso relevante.

[3]Tales como hacer interactuar de manera implícita a dos elementos aparentemente aislados, precisamente dentro de un lenguaje diseñado para hacer explícita cualquier conexión entre los mismos, pues ésta es la función principal de un Adl [MT00]. Tal cosa ocurre en el ejemplo del Espacio de Tuplas (§8.5).

Parte V

Conclusiones

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

Capítulo 9

Conclusiones y Trabajo Futuro

9.1 Conclusiones

La premisa básica de la Arquitectura de Software Dinámica es que los aspectos dinámicos y evolutivos de un sistema son parte integrante de su arquitectura, por lo que han de ser modelados junto a los aspectos estructurales, y a su mismo nivel. Sin embargo, la mayoría de los ADLs tradicionales han ignorado este aspecto, o lo han tratado de manera insuficiente. Incluso cuando se incluye, la especificación de dinamismo es casi siempre tratada como un aspecto secundario, y en muchos casos está considerablemente limitada.

Inicialmente, este trabajo de tesis comenzó como la búsqueda de un modelo general de dinamismo arquitectónico, que fuese capaz –al menos– de englobar a todas las propuestas ya existentes en un marco común. El punto clave de esta búsqueda era encontrar un concepto unificador, que permitiera considerar a los distintos enfoques parciales desde una perspectiva uniforme.

La hipótesis de este trabajo de tesis es que este concepto es el de Reflexión. Por tanto, se plantea como objetivo principal la elaboración de un marco reflexivo de Arquitectura de Software, que permita demostrar dicha hipótesis, esto es, que *la introducción de Reflexión en la Descripción Arquitectónica proporciona un marco general para la especificación de Dinamismo*. Creemos que este trabajo ha justificado esta hipótesis con amplitud suficiente.

En el proceso seguido para la consecución de este objetivo, se han obtenido los siguientes resultados intermedios:

- ☑ Se ha realizado un estudio general sobre el problema de la Arquitectura de Software Dinámica, haciendo especial énfasis en las distintas propuestas existentes para su solución. Cada una de estas propuestas ha sido examinada en detalle, y relacionada con otros enfoques similares según su origen conceptual. Como resultado de este estudio, se han identificado siete modelos básicos de dinamismo, que resumen todas las aproximaciones utilizadas, haciendo posible la comparación incluso entre las más dispares.

- ☑ Se han revisado las nociones de Reflexión desde la perspectiva de la Arquitectura de Software, basándose en los modelos de dinamismo identificados; de este modo, se han extraído

aquéllas que se consideran especialmente relevantes en el contexto, elaborando a partir de ellas un *modelo reflexivo de descripción arquitectónica* de carácter general.

- ☑ Con el fin de dar una descripción precisa de este modelo, se ha elaborado una definición formal completa del mismo, basada en teoría de conjuntos y de relaciones. Esta definición se ha realizado en tres etapas: en la primera ($\aleph_\Sigma$) se han descrito todos los conceptos estándar de la Arquitectura de Software, incluyendo una formulación básica de tipos. En la segunda ($\aleph_\mathrm{M}$), se han introducido los conceptos reflexivos, incluyendo a la noción de *reificación* concebida como tercera dimensión de la descripción arquitectónica. Por fin, en la tercera ($\aleph_\Pi$) se han colapsado parcialmente las relaciones de tipado y reificación, con el fin de facilitar la expresión de reflexión estructural. Al conjunto de estos (tres) modelos se le ha dado el nombre común de Marmol. Incluso el primero de ellos constituye un trabajo original, pues es uno de los primeros estudios del campo desde esta perspectiva; los otros dos contribuyen a aclarar notablemente el impacto del concepto de reflexión desde un punto de vista estrictamente *estructural*.

- ☑ Con el fin de demostrar la aplicabilidad de este modelo, se ha construido un lenguaje de descripción con base en el mismo, al que se ha dado el nombre de $\mathcal{P}i\mathcal{L}ar$. Aunque se concibe en primer lugar como un Adl reflexivo, se ha diseñado de forma que sea lo más general posible. El planteamiento general de su sintaxis lo sitúa como un Adl de tipo algebraico, con un conjunto de instrucciones minimal. Sólo dos instrucciones son específicamente dinámicas: su potencia se desarrolla mediante el esquema reflexivo.

- ☑ Se ha desarrollado una *semántica formal* para este lenguaje, proporcionando de este modo una descripción precisa de su significado. Esta semántica ha sido planteada en términos de una variante del cálculo-π tipado, lo que demuestra la consistencia del enfoque utilizado. La reflexión ha sido planteada en términos de relaciones de reificación explícita entre estructuras –de manera similar al esquema descrito en Marmol– y de relaciones de *superposición* entre procesos, planteado en términos del cálculo-π como una fusión de ámbitos. El esquema global de la semántica es muy complejo, pero resulta totalmente coherente, tanto con la definición estructural del modelo, como con la teoría de tipos del cálculo.

- ☑ Se ha planteado, además, la posibilidad de incluir extensiones específicas en el lenguaje. Este aspecto se ha mostrado describiendo una de ellas, relacionada con la noción de *envolventes* reflexivos, que resulta especialmente adecuada en este contexto.

- ☑ Se ha comprobado la utilidad de $\mathcal{P}i\mathcal{L}ar$ como Adl de tipo general a través del desarrollo de una serie de ejemplos. De este punto se extraen las conclusiones siguientes:

  - Ante la descripción de sistemas estáticos, el uso de $\mathcal{P}i\mathcal{L}ar$ es tan sencillo como el de cualquier otro Adl existente. De hecho, sus características lingüísticas específicas lo hacen más potente que muchos de ellos.
  - En líneas generales, la descripción del dinamismo habitual en otros lenguajes –creación y destrucción de componentes y conexiones– resulta natural y bastante cómoda. Algunos esquemas, no obstante, muestran una solución más compleja, derivada del hecho de que parten de unas premisas diferentes.

- El lenguaje supone un marco especialmente adecuado para la especificación de esquemas reflexivos, no comparable al de ningún otro lenguaje. También extrae con naturalidad los patrones reflexivos implícitos en arquitecturas aparentemente convencionales. No obstante, puede alcanzar una gran complejidad, en especial cuando se manipulan explícitamente más de dos niveles.
- La reificación de conexiones permite un nuevo enfoque de la noción de conector –el *conector de nivel meta*–, que permite conciliar las posturas antagónicas existentes ante este concepto. Supone una aproximación totalmente diferente al mismo, y manifiesta un gran potencial que aún ha de ser explotado.

☑ Se ha de concluir, en definitiva, que el uso de reflexión no sólo permite establecer un modelo unificado para la expresión de dinamismo, sino que tiene aún más posibilidades que las inicialmente planteadas. Existe toda una serie de relaciones, incluso de tipo estático, que se expresan con mayor facilidad en la presencia de dinamismo. La descripción del concepto de *envolventes* es sólo una de ellas.

☑ Como objetivo secundario, se ha mostrado también que la inclusión de un álgebra de procesos tipo-CCS en un marco reflexivo dota al sistema de unas capacidades análogas a las del propio cálculo-π. Esto no es sorprendente, en realidad, ya que la potencia de este último deriva, entre otros aspectos, del modo en que difumina –controladamente– la distinción uso/mención.

En definitiva, los objetivos planteados han sido alcanzados. Se ha logrado construir, en efecto, un modelo reflexivo de descripción arquitectónica, y se ha mostrado su aplicabilidad al problema de la descripción de dinamismo. Más aún, el modelo obtenido es lo bastante genérico como para trascender estos objetivos, de modo que define un marco apto para la descripción de otras abstracciones interesantes, no necesariamente relacionadas con el dinamismo.

9.2 Líneas de Trabajo Futuro

El objetivo principal de este trabajo era llegar a la definición de un modelo arquitectónico general (MARMOL) y un lenguaje de descripción específico ($\mathcal{P}i\mathcal{L}ar$), completamente formalizado, que demostrasen la viabilidad de la aproximación reflexiva a la especificación de dinamismo. En este sentido, ambos aspectos han sido completamente desarrollados, y la potencia expresiva del enfoque reflexivo suficientemente justificada.

Sin embargo, una vez se ha elaborado este marco, resultará lógico utilizarlo para explorar otras abstracciones cercanas, así como algunos aspectos suplementarios de las aquí examinadas. De este modo, no sólo se alcanzará una mayor comprensión de las mismas, sino también de las posibilidades del entorno. En principio, parece que la expresividad del modelo reflexivo –y concurrente– lo hace apto para la descripción de casi cualquier estructura. Algunas de ellas son particularmente interesantes en sí mismas, y su estudio podría determinar tanto los límites como la adecuación del enfoque utilizado.

Por otra parte, hay aspectos, tanto de la descripción arquitectónica como del soporte formal empleado, que podrían haberse expandido en más de un sentido, llevando a algunos conceptos

hasta sus últimas consecuencias. Recorrer algunos de estos caminos permitiría ampliar el dominio de aplicación del modelo y el lenguaje definidos, así como adquirir una mayor comprensión de los aspectos implicados.

Estas consideraciones dividen a las líneas de trabajo futuro en dos grupos. En el primero, relativo al estudio de abstracciones en el marco, se considerarán las siguientes:

- Una abstracción para la que este modelo define un entorno particularmente adecuado es la de *interacción múltiple* en entornos concurrentes [FF96, JS96]. En realidad, podría decirse que, en este sentido, el concepto y el modelo siguen esquemas análogos. La interacción concurrente se plantea como primitiva, y se especifica después en términos de interacciones binarias. Del mismo modo, en $\mathcal{P}i\mathcal{L}ar$ se puede definir una conexión múltiple, y sólo se desarrolla su estructura –de nuevo, como interacciones binarias– en el nivel meta.

- Además de esto, se plantea el interés de realizar un estudio general de la noción de *intermediario*, aprovechando el marco especialmente adecuado que proporciona el concepto de conector de nivel meta. Esto incluiría desde los conectores clásicos de la Arquitectura de Software, hasta los *adaptadores* de Canal [CPT99], pasando por los *sincronizadores* y *meta-actores* de Frølund [FA93], los *envolventes* [Höl93] en cualquiera de sus encarnaciones, o en general todo tipo de *mediadores* [DeL99, GHJ+95a, Sch99, Sul94]. Un estudio de este tipo permitiría comprobar las analogías y diferencias entre todos ellos, y arrojaría nueva luz sobre el propio concepto de *conector*.

- Aún más interesante resulta la exploración de la analogía entre los componentes reflexivos de $\mathcal{P}i\mathcal{L}ar$ y la noción de *prototipos* [Bor86, Car96, DMC92, Foo89, US87, Ung88], en particular en lo que respecta al dinamismo de tercer nivel. Es habitual utilizar implícitamente una analogía entre componentes y objetos, que asimila a los arquetipos con las clases; sin embargo, en un enfoque dinámico como éste, en el que una instancia puede alterar su comportamiento e incluso "cambiar" de tipo por reificación, la analogía con el modelo de prototipos es aún más natural. No obstante, esta analogía no es total, dado que en $\mathcal{P}i\mathcal{L}ar$ las instancias siguen manteniendo un vínculo con su patrón estructural –su "clase"– a través de la reificación. En cualquier caso, un estudio de este tipo contribuiría a clarificar la naturaleza de las relaciones entre los elementos de $\mathcal{P}i\mathcal{L}ar$.

- En esta misma línea, uno de los aspectos más interesantes a explorar se encuentra en la definición semántica de $\mathcal{P}i\mathcal{L}ar$. En su forma actual, una reificación afecta a la estructura y proceso propios de cada componente. Si éste es un metacomponente, el efecto de la reificación se extiende a sus avatares. Sin embargo, las alteraciones no se propagan a su brhátar, esto es, a la definición pasiva del arquetipo correspondiente. Esto significa que los cambios afectan a las instancias existentes, pero no a las que se creen en el futuro. Se podría plantear el enfoque opuesto –de hecho, su especificación semántica sería notablemente sencilla–, pero en tal caso no podría recuperarse la definición de tipo original. Ambos enfoques tienen sus ventajas e inconvenientes, por lo que su combinación podría ser ventajosa: no obstante, esto implicaría tener dos formas de expandir la reificación, lo que sería sin duda complicado.

- Finalmente, un detalle que ya ha sido comentado es la particular adecuación de la estructura de reificaciones múltiples de $\mathcal{P}i\mathcal{L}ar$ a los conceptos del denominado *desarrollo de software orientado a aspectos* (AOSD) [OK02]. Es obvio que se puede plantear la definición de aspectos –sujetos, dimensiones– como componentes de nivel meta, y luego reificarlos simultáneamente sobre un mismo avatar, consiguiendo de este modo una combinación –tejido– de aspectos a partir de la propia dinámica concurrente de las reglas de $\mathcal{P}i\mathcal{L}ar$. Éste es, de hecho, el mismo enfoque de James Andrews [And01] para la especificación de aspectos en términos de un álgebra de procesos. Por otra parte, en términos más puramente arquitectónicos, se puede considerar que un esquema de este tipo constituiría la primera definición explícita de *vistas* (§2.4.6) en un ADL.

En cuanto a las posibles extensiones del trabajo realizado, que constituyen el segundo grupo, se considerarán las siguientes:

- Un concepto arquitectónico que ha sido explícitamente ignorado es el de estilo. Una de las primeras extensiones a plantear es, por tanto, la *inclusión de estilos*. Se ha dicho (§2.4.5) que en realidad este aspecto no ha sido satisfactoriamente acometido por ningún ADL, aunque sin duda el enfoque de Wright [All97] es uno de los más interesantes. Allí, las restricciones sobre el concepto de estilo se expresaban en lógica de predicados. Ese mismo enfoque podría ser aplicado en $\mathcal{P}i\mathcal{L}ar$; no obstante, en un contexto dinámico y concurrente, parecería más apropiado utilizar alguna forma de lógica modal, en especial una lógica temporal, ya que permite especificar con mayor facilidad las propiedades deseadas [ADG98]. La adecuación de éstas con un sistema de álgebra de procesos, como el utilizado en $\mathcal{P}i\mathcal{L}ar$, resulta especialmente apropiada [BS01], como demuestra el uso de una lógica modal específica en el MWB [VM94], o el del cálculo-μ modal en el CWB.

- En esta tesis, se define un componente en base al concepto de *bisimulación* débil. Esto significa, simplemente, que su comportamiento se define en base a las acciones que pueden observarse desde el exterior. Por tanto, toda entidad cuyo comportamiento observable respete –equivalga a– las restricciones definidas para un componente dado podrá ser utilizada en su contexto. No obstante, esta definición puede considerarse demasiado restrictiva, en el sentido de que no plantea la posibilidad de la extensión o sustitución de los componentes. En este sentido, resultaría interesante explorar una noción de subtipado para componentes. Un desarrollo inicial de este aspecto podría basarse en los estudios preliminares de Dinesh Katiyar [KLM94] y David Rosenblum [MRT98], y muy especialmente en la noción de *compatibilidad* descrita por Carlos Canal [CPT01] en el contexto de LEDA.

- En esta tesis, se ha discutido informalmente el modo en que la reflexión y la *superposición* se corresponden en un modelo concurrente, y se han citado otras opiniones similares [For00, KG99] en esta línea. Sin embargo, no se ha desarrollado todavía un estudio detallado en este sentido, que resultaría particularmente interesante. El modelo definido en esta tesis se muestra como un entorno particularmente adecuado para realizar tal comparación: $\mathcal{P}i\mathcal{L}ar$ ha sido descrito como un ADL reflexivo. No obstante, puede demostrarse fácilmente que además permite expresar una noción de superposición [Bos99, SK02] entre componentes, sin necesidad de cambio alguno en su estructura.

- Desde un punto de vista principalmente teórico, y una vez que se ha dado una noción de componente basada en la equivalencia observacional, resulta interesante plantear un estudio del modelo desde la perspectiva opuesta, utilizando un enfoque coalgebraico [JR97, RT94]. Esto supone, por una parte, un desarrollo completamente distinto; sin embargo, ambos muestran puntos de coincidencia notables, a través de la noción de bisimulación. Por otra parte, la perspectiva que se obtendría de la reflexión desde un punto de vista coinductivo es un aspecto particularmente intrigante. Algunos trabajos ya existentes [Bar00, Jac98] podrían ser utilizados como base inicial para un estudio de este tipo.

- De manera similar, resulta especialmente interesante el acometer un estudio de Marmol basado en teoría de categorías. Es sencillo pasar desde la descripción actual a una en términos de teoría de grafos; ésta podrá luego expandirse teniendo en cuenta las nociones categóricas. Algunos aspectos en particular, como la correspondencia entre instancias y tipos de componente o el colapso –metalepsis– de las nociones de reificación y tipado, se beneficiarían enormemente de esta perspectiva. En este sentido, resulta también interesante la comparación con el enfoque de Michel Wermelinger [Wer99], que constituye el trabajo más completo en esta línea.

- Finalmente, se ha de considerar que el esquema de reificaciones, tal como se establece en $\mathcal{P}i\mathcal{L}ar$, muestra una estructura parcialmente dispersa. Las reificaciones se establecen libremente entre componentes individuales, y no se adquiere una visión global de su estructura, salvo deduciéndola a partir de la especificación. Como señala Medvidovic [MT00], precisamente uno de los rasgos más significativos de la Arquitectura de Software es hacer patente los esquemas de interacción mediante una noción explícita de configuración. Del mismo modo, sería deseable describir un esquema similar para la reificación, definiendo una noción análoga de *meta-configuración*. Sin embargo, aún no está claro como acometer un planteamiento de este tipo, ya que cualquier construcción en este sentido va explícitamente en contra de la flexibilidad del modelo, que es su valor principal.

Las líneas de trabajo indicadas sólo son algunas de las posibles; por supuesto, podrían plantearse muchas otras. Por ejemplo, un enfoque interesante sería estudiar la sustitución del cálculo-π por otro cálculo nominal: algunas de las características del cálculo de ambientes, en particular, parecen adaptarse especialmente al contexto de la descripción arquitectónica. No obstante, realizar –duplicar– este trabajo supondría una extensión considerable, y se trataría, en realidad, de una aproximación conceptualmente distinta.

Bibliografía

[ACDG95] Massimo Ancona, Walter Cazzola, Gabriella Dodero y Vittoria Gianuzzi. Channel Reification: a Reflective Approach to Fault Tolerant Software Development. En *Proceedings of OOPSLA'95*, pág. 137. ACM, Octubre 1995.

[ACH98] Farhad Arbab, Paolo Ciancarini y Chris Hankin. Coordination Languages for Parallel Programming. *Parallel Computing*, 24(7):989–1004, 1998. Special Issue on Coordination.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce y Gordon Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, Abril 1991.

[ADG97] Robert Allen, Rémi Douence y David Garlan. Specifying Dynamism in Software Architectures. En *Proceedings of Foundations of Component-Based Systems Workshop*, Septiembre 1997.

[ADG98] Robert Allen, Rémi Douence y David Garlan. Specifying and Analyzing Dynamic Software Architectures. En *Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering*, volumen 1382 de *Lecture Notes in Computer Science*, págs. 21–37, Lisboa, Marzo 1998. Springer Verlag.

[AF01] Luis F. Andrade y José Luiz Fiadeiro. Coordination Technologies for Managing Information Systems Evolution. En K. Dittrich, A. Geppert y M. Norrie, editores, *Proceedings of CAiSE'01*, volumen 2068 de *Lecture Notes in Computer Science*, págs. 374–387. Springer Verlag, 2001.

[AG94a] Robert Allen y David Garlan. Beyond Definition/Use: Architectural Interconnection. *ACM SIGPLAN Notices*, 29(8), Agosto 1994. Proceedings of ACM IDL Workshop.

[AG94b] Robert Allen y David Garlan. Formal Connectors. Technical Report CMU-CS-94-115, Carnegie Mellon University, 1994.

[AG94c] Robert Allen y David Garlan. Formalizing Architectural Connection. En *Proceedings of the ICSE'16*, Mayo 1994.

[AG99] Martín Abadi y Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Journal of Information and Computation*, 148(1):1–70, 1999.

[Agh97] Gul A. Agha. Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems. En E. Najm y J.B. Stefani, editores, *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions. Chapman & Hall, 1997.

[AHR93] Farhad Arbab, Ivan Herman y Eric P. M. B. Rutten. The Skeleton of a Computer Farm in Manifold. En H. El-Rewini, T. Lewis y B.D. Schriver, editores, *Proceedings of the* 26th *Annual Hawaii International Conference on System Sciences (HICSS26)*, volumen II, págs. 347–356, Los Alamitos, California, 1993. IEEE CS Press.

[AHS93] Farhad Arbab, Ivan Herman y Per Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23–70, Febrero 1993. Also Report CS-R9142.

[AIS+80] Christopher Alexander, Sarah Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King y Shlomo Angel. *Un Lenguage de Patrones*. Gustavo Gili, 1980.

[AJ94] Samson Abramsky y Achim Jung. Domain Theory. En Samson Abramsky, Dov M. Gabbay y Tom S. E. Maibaum, editores, *Handbook of Logic in Computer Science*, volumen 3, págs. 1–168. Oxford University Press, 1994.

[AKHS00] Colin Atkinson, Thomas Kühne y Brian Henderson-Sellers. To Meta or Not to Meta– That Is the Question. *Journal of Object-Oriented Programming*, págs. 32–35, Diciembre 2000.

[AKN00] Franz Achemann, Stefan Kneubuehl y Oscar M. Nierstrasz. Scripting Coordination Styles. En Porto y Roman [PR00].

[Ale81] Christopher Alexander. *El Modo Intemporal de Construir*. Gustavo Gili, 1981.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. Tesis Doctoral, School of Computer Science, Carnegie Mellon University, Mayo 1997. Technical Report CMU-CS-97-144.

[ALP99] Uwe Aßmann, Andreas Ludwig y Daniel Pfeifer. Programming Connectors in an Open Language. En Clements et al. [CPR99].

[ALSN00] Franz Achermann, Markus Lumpe, Jean-Guy Schneider y Oscar M. Nierstrasz. Piccola – A Small Composition Language. En Howard Bowman y John Derrick, editores, *Formal Methods for Distributed Processing: an Object-Oriented Approach*. Cambridge University Press, 2000.

[AN00] Franz Achermann y Oscar M. Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. En Mehmet Aksit, editor, *Software Architecture and Component Technology*, Enschede, Netherlands, 2000. Kluwer Academic Publishers.

[And91] Gregory Andrews. *Concurrent Programming. Principles and Practice.* Benjamin-Cummings, 1991.

[And01] James H. Andrews. Process-Algebraic Foundations of Aspect-Oriented Programming. En Yonezawa y Matsuoka [YM01].

[Arb96] Farhad Arbab. The IWIM Model for Coordination of Concurrent Activities. En Ciancarini y Hankin [CH96], págs. 24–56.

[Arb98] Farhad Arbab. What Do You Mean, Coordination? *Bulletin of Dutch Association for Theoretical Computer Science (NVTI)*, Marzo 1998.

[ASA01] Mark Astley, Daniel C. Sturman y Gul A. Agha. Customizable Middleware for Modular Distributed Software. *Communications of the ACM*, 44(5):99–107, Mayo 2001.

[Ast99] Mark Christopher Astley. *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures.* Tesis Doctoral, University of Illinois at Urbana-Champaign, 1999.

[AT02] Farhad Arbab y Carolyn Talcott, editores. *Fifth International Conference on Coordination Models and Languages, COORDINATION 2002*, volumen 2315 de *Lecture Notes in Computer Science*, York, Abril 2002. Springer Verlag.

[AWB+93] Mehmet Akşit, Ken Wakita, Jan Bosch, Lodewijk Bergmans y Akinori Yonezawa. Abstracting Object Interactions using Composition Filters. En Rachid Guerraoui, Oscar Nierstrasz y Michel Riveill, editores, *Object-Based Distributed Programming*, volumen 791 de *Lecture Notes in Computer Science*, págs. 152–184. Springer-Verlag, 1993. ECOOP'93.

[AWG99] Recommended Practice for Architectural Description. IEEE Architecture Working Group, Diciembre 1999. Draft: Version 5.2.

[BA00] Lodewijk M. J. Bergmans y Mehmet Aksit. Aspects & Crosscutting in Layered Middleware Systems. En *Reflective Middleware 2000.* ACM/IFIP, 2000.

[Bal97] Bob Balzer. Instrumenting, Monitoring & Debugging Software Architectures. Technical report, USC/ISI, 1997.

[Bar92] Henk Barendregt. Lambda Calculi with Types. En Samson Abramsky, Dov M. Gabbay y Tom S.E. Maibaum, editores, *Handbook of Logic in Computer Science*, volumen 2, págs. 117–309. Oxford University Press, 1992.

[Bar00] Luis Manuel Soares Barbosa. Components as Processes: An Exercise in Coalgebraic Modeling. En Scott F. Smith y Carolyn L. Talcott, editores, 4th *Formal Methods for Open Object-Based Distributed Systems (FMOODS'2000)*, págs. 397–417, Stanford, 2000. Kluwer Academic Publishers.

[BB90] Gérard Berry y Gérard Boudol. The Chemical Abstract Machine. En *POPL '90. Proceedings of the* 17th *annual ACM symposium on Principles of Programming Languages*, págs. 81–94, New York, NY, USA, 1990. ACM Press.

[BB98] Lynne Blair y Gordon S. Blair. The Impact of Aspect-Oriented Programming on Formal Methods. En *Proceedings of ECOOP'98 Aspect-Oriented Programming Workshop*. Springer-Verlag, 1998.

[BBI+00] Gordon S. Blair, Lynne Blair, Valérie Issarny, Petr Tůma y Apostolos Zarras. The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. En Springer-Verlag, editor, *Proceedings of Middleware 2000*, Lecture Notes in Computer Science. IFIP/ACM, Abril 2000.

[BC98] Gordon S. Blair y Geoff Coulson. The Case for Reflective Middleware. Technical Report MPG-98-38, Distributed Multimedia Research Group, Department of Computing, Lancaster University, 1998.

[BCA+01] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas y Katia Saikoski. The Design and Implementation of Open ORB v2. *IEEE Distributed Systems Online*, 2(6), 2001.

[BCCK01] Gordon Blair, Fabio Costa, Roy Campbell y Fabio Kon. Reflective Middleware. *IEEE Distributed Systems Online*, 2(5), 2001.

[BCD01] Marco Bernardo, Paolo Ciancarini y Lorenzo Donatiello. Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems. En Kazman et al. [KKVvV01], págs. 77–86.

[BCK98] Len Bass, Paul Clements y Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BCRP98] Gordon S. Blair, Geoff Coulson, Phillipe Robin y Michael Papathomas. An Architecture for Next Generation Middleware. En N. Davies, K. Raymond y J. Seitz, editores, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lecture Notes in Computer Science, Lake District, UK, 1998. Springer Verlag.

[BDH+98] Manfred Broy, Anton Deimel, Jürgen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal y Clemens Szyperski. What Characterizes a Software Component? *Software Concepts and Tools*, 19(1), 1998. Software Research Lab, Technical Report # J010.

[BDWL91] Mario R. Barbacci, Dennis Doubleday, Charles B. Weinstock y Randall W. Lichota. Durra: An Integrated Approach to Software Specification, Modeling and Rapid Prototyping. Technical Report CMU/SEI-91-TR-21, Software Engineering Institute (SEI), Septiembre 1991.

[Ber94] Lodewijk Bergmans. *Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs.* Tesis Doctoral, TRESE, University of Twente, Enschede, Netherlands, Junio 1994.

[BF88] Luc Bougé y Nissim Francez. A Compositional Approach to Superimposition. En 15th *Annual ACM Symposium on Principles of Programming Languages, POPL'88*, págs. 240–249, San Diego, 1988. ACM Press.

[BF97] Manuel Barrio y Pablo de la Fuente. Software Architecture: Object vs. Process Approach. En *Proceedings of SCCC'97.* IEEE Computer Society Press, Noviembre 1997.

[BF02] Marco Bernardo y Francesco Franzè. Endogenous and Exogenous Extensions of Architectural Types. En Arbab y Talcott [AT02], págs. 40–55.

[BFP01] Jan A. Bergstra, Wan Fokkink y Alban Ponse. Process Algebra with Recursive Operations. En Bergstra et al. [BPS01], capítulo 5, págs. 333–389.

[BGZ97a] Nadia Busi, Roberto Gorrieri y Gianluigi Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. *Electronic Notes in Theoretical Computer Science*, 7, 1997. Express'97.

[BGZ97b] Nadia Busi, Roberto Gorrieri y Gianluigi Zavattaro. Three Semantics of the Output Operation for Generative Communication. En Garlan y Le Métayer [GLM97], págs. 205–219.

[BGZ98] Nadia Busi, Roberto Gorrieri y Gianluigi Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[BK96] J. A. Bergstra y P. Klint. The TOOLBUS Coordination Architecture. En Ciancarini y Hankin [CH96], págs. 75–88.

[BL97a] Maria Lúcia Blanck Lisbôa. Arquiteturas de Meta-Nível. Simpósio Brasileiro de Engenharia de Software, Fortaleza, Octubre 1997.

[BL97b] Maria Lúcia Blanck Lisbôa. Reflexão Computacional no Modelo de Objetos. II Simpósio Brasileiro de Linguagens de Programaçao, Campinas, Septiembre 1997.

[BLM96] Jean-Paul Banâtre y Daniel Le Métayer. Gamma and the Chemical Reaction Model: Ten Years After. En *Coordination Programming: Mechanisms, Models and Semantics*, págs. 3–41. World Scientific Publishing, IC Press, 1996.

[Boo96] Grady Booch. *Análisis y Diseño Orientado a Objetos con Aplicaciones.* Addison-Wesley/Díaz de Santos, 2 edición, 1996.

[Bor86] Alan H. Borning. Classes Versus Prototypes in Object-Oriented Languages. En *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, págs. 36–40, Dallas, TX, Noviembre 1986. IEEE.

[Bos96a] Jan Bosch. Composition through Superimposition. En *ECOOP '96 Workshop on Composability Issues in Object-Orientation (CIOO '96)*, 1996.

[Bos96b] Jan Bosch. Language Support for Component Communication in LayOM. En *ECOOP'96 Workshop on Component-Oriented Programming (WCOP)*, 1996.

[Bos96c] Jan Bosch. Relations as Object Model Components. *Journal of Programming Languages*, 4:39–61, 1996.

[Bos98] Jan Bosch. Specifying Frameworks and Design Patterns as Architectural Fragments. En Jian Chen, M. Li, Christine Mingis y Bertrand Meyer, editores, 27th *Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia '98)*, Beijing, China, Septiembre 1998. IEEE Computer Society Press.

[Bos99] Jan Bosch. Superimposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5):257–273, Marzo 1999.

[Bos00] Jan Bosch. *Design and Use of Software Architectures. Adopting and Evolving a Product-Line Approach.* Addison-Wesley, Mayo 2000.

[Bou98] Gérard Boudol. The π-Calculus in Direct Style. *Higher-Order and Symbolic Computation*, 11:177–208, 1998. Previous version in the *Proceedings of POPL'97*, ACM Press, pages 228–241.

[BPS01] Jan A. Bergstra, Alban Ponse y Scott A. Smolka, editores. *Handbook of Process Algebra.* Elsevier Science – North Holland, Amsterdam, 2001.

[Bro96] Manfred Broy. Towards a Mathematical Model of a Component and Its Use. En *Proceedings of Componentware Users Conference 1996.* SIGS Publications, 1996.

[Bru99] Roberto Bruni. *Tile Logic for Synchronized Rewriting of Concurrent Systems.* Tesis Doctoral, University of Pisa, 1999.

[BS95] Manuel Barrio Solórzano. *Estudio de Aspectos Dinámicos en Sistemas Orientados al Objeto.* Tesis Doctoral, Universidad de Valladolid, 1995.

[BS01] Julian C. Bradfield y Colin P. Stirling. Modal Logics and mu-Calculi: An Introduction. En Bergstra et al. [BPS01], págs. 293–330.

[BW97] Martin Büchi y Wolfgang Weck. A Plea for Grey-Box Components. Technical Report TUCS-TR-122, Turku Centre for Computer Science, Finlandia, Septiembre 1997.

[CA94] Christian Jürgen Callsen y Gul A. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21:289–300, 1994.

[Cal94] Christian Jürgen Callsen. *Open Distributed Hetereogeneous Computing.* Tesis Doctoral, Aalborg University, Marzo 1994.

[Car96] Luca Cardelli. Object-based vs. Class-based Languages. Tutorial at the 9th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96), Philadelphia, Pennsylvania, Abril 1996.

[Car97] Luca Cardelli. Type Systems. En Allen B. Tucker, Jr., editor, *The Handbook of Computer Science and Engineering*, capítulo 103, págs. 2208–2236. CRC Press, 1997.

[Car98] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* Tesis Doctoral, Politecnico di Milano, Diciembre 1998.

[Cas01] Ilaria Castellani. Process Algebras with Localities. En Bergstra et al. [BPS01], capítulo 15, págs. 945–1045.

[Caz98] Walter Cazzola. Evaluation of Object-Oriented Reflective Models. En *Proceedings of ECOOP98 Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, Julio 1998.

[Caz00] Walter Cazzola. *Communication-Oriented Reflection: a Way to Open Up the RMI Mechanism.* Tesis Doctoral, Università degli Studi di Milano, Milano, Italia, Noviembre 2000.

[CCL00a] Walter Cazzola, Shigeru Chiba y Thomas Ledoux, editores. *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, Nice, France, Junio 2000.

[CCL00b] Walter Cazzola, Shigeru Chiba y Thomas Ledoux. Reflection and Meta-Level Architectures: State of the Art, and Future Trends. En *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures* [CCL00a].

[CD97] Stephen Crane y Naranker Dulay. A Configurable Protocol Architecture for CORBA Environments. En *ISADS-97, Berlin*, 1997.

[CD99] Juan Carlos Cruz y Stéphane Ducasse. A Group-Based Approach for Coordinating Active Objects. En Ciancarini y Wolf [CW99], págs. 355–370.

[CDF+95] Stephen Crane, Naranker Dulay, Halldor Fosså, Jeff Kramer, Jeff Magee, Morris Sloman y Kevin Twidle. Configuration Management for Distributed Systems. En *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (ISINM 95)*, Mayo 1995.

[CF67] Haskell Brooks Curry y Robert Feys. *Lógica Combinatoria*, volumen 24 de *Estructura y Función.* Tecnos, 1967. Traducido por Manuel Sacristán.

[CFBS98] Carlos E. Cuesta, Pablo de la Fuente y Manuel Barrio-Solórzano. Una Perspectiva Arquitectónica de la Invocación Estática en CORBA. En Ambrosio Toval Álvarez y Joaquín Nicolás Ros, editores, *III Jornadas de Ingeniería del Software (JIS'98)*, págs. 219–230, Murcia, Noviembre 1998. Universidad de Murcia, Servicio de Publicaciones.

[CFBS99] Carlos E. Cuesta, Pablo de la Fuente y Manuel Barrio-Solórzano. An Architectural Perspective of the Static Invocation in CORBA. En Clements et al. [CPR99].

[CFBSB00] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano y Encarnación Beato. Arquitectura de Software Dinámica Basada en Reflexión. En Carlos Delgado Kloos, Esperanza Marcos y José Manuel Marqués Corral, editores, *V Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2000)*, págs. 203–214, Valladolid, Noviembre 2000. Universidad de Valladolid, Secretariado de Publicaciones.

[CFBSB01] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano y Encarnación Beato. Dynamic Coordination Architecture through the use of Reflection. En *Proceedings of* 16th *ACM Symposium on Applied Computing (SAC2001)*, ACM Proceedings, págs. 134–140, Las Vegas, NV, Marzo 2001. ACM Press.

[CFBSB02a] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano y Encarnación Beato. Coordination in a Reflective Architecture Description Language. En Arbab y Talcott [AT02], págs. 141–148.

[CFBSB02b] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio Solórzano y M. Encarnación Beato. Introducing Reflection in Architecture Description Languages. En Jan Bosch, Morven Gentleman, Christine Hofmeister y Juha Kuusela, editores, *Software Architecture: System Design, Development and Maintenance*, capítulo 9, págs. 143–156. Kluwer Academic Publishers, Montréal, Québec, Canada, Agosto 2002.

[CFM98] Paolo Ciancarini, F. Franzè y Cecilia Mascolo. A Coordination Model to Specify Systems Including Mobile Agents. En *Proceedings of the Ninth International Workshop on Software Specification and Design*, págs. 96–105. IEEE Computer Society Press, 1998.

[CG86] Nicholas Carriero y David Gelernter. How to Write Parallel Programs: a Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, 1986.

[CG98] Luca Cardelli y Andrew D. Gordon. Mobile Ambients. En Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, volumen 1378 de *Lecture Notes in Computer Science*, págs. 140–155. Springer, 1998.

[CG00] Luca Cardelli y Andrew D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[CGG00] Luca Cardelli, Giorgio Ghelli y Andrew D. Gordon. Secrecy and Group Creation. En Catuscia Palamidessi, editor, *CONCUR 2000 - Proceedings of* 11th *International Conference on Concurrency Theory*, volumen 1877 de *Lecture Notes in Computer Science*, págs. 365–379, University Park, PA, USA, Agosto 2000. Springer.

[CGZ95a] Nick Carriero, David Gelernter y Lenore Zuck. Bauhaus Linda. En Paolo Ciancarini, Oscar Nierstrasz y Akinori Yonezawa, editores, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, págs. 66–76. Springer-Verlag, 1995.

[CGZ95b] Paolo Ciancarini, Roberto Gorrieri y Gianluigi Zavattaro. Generative Communication in Process Algebra. Technical Report UBLCS-95-7, Universitá di Bologna, Febrero 1995.

[CGZ96] Paolo Ciancarini, Roberto Gorrieri y Gianlugi Zavattaro. Towards a Calculus for Generative Communication. En E. Najm y J. Stefani, editores, *Proceeding IFIP Conference on Formal Methods for Open Object-Based Distributed Systems*, págs. 289–306, Paris, France, 1996. IFIP Wg 6.1.

[CH96] Paolo Ciancarini y Chris Hankin, editores. *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, volumen 1061 de *Lecture Notes in Computer Science*, Cesena, Italia, Abril 1996. Springer Verlag.

[Che94] Shing-Chi Cheung. *Tractable and Compositional Techniques for Behaviour Analysis of Concurrent Systems.* Tesis Doctoral, Imperial College of Science, Technologie and Medicine. University of London, Febrero 1994.

[Chi95] Shigeru Chiba. A Meta-Object Protocol for C++. *ACM SIGPLAN Notices*, 30(10):285–299, Octubre 1995. Proceedings of OOPSLA'95.

[Chu40] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cia91] Paolo Ciancarini. PoliS: a Programming Model for Multiple Tuple Spaces. En Carlo Ghezzi y Gruia-Catalin Roman, editores, *Proceedings of* 6th *IEEE International Workshop on Software Specification and Design*, págs. 44–51, Como, Italia, Octubre 1991. IEEE Computer Society Press.

[Cia93] Paolo Ciancarini. Distributed Programming with Logic Tuple Spaces. Technical Report UBLCS-93-7, Universitá di Bologna, 1993.

[Cia97] Paolo Ciancarini. *Coordination Models, Languages, Architectures and Applications.* Università di Bologna, Octubre 1997.

[CKL96] Shigeru Chiba, Gregor Kiczales y John Lamping. Avoiding Confusion in Metacircularity: The Meta-Helix. En *Proceedings of Second International Symposium*

on Object Technologies for Advanced Software (ISOTAS'96), número 1049 en Lecture Notes in Computer Science, págs. 157–172. Springer Verlag, 1996.

[Cle96] Paul C. Clements. A Survey of Architecture Description Languages. En *Proceedings of* 8th *International Workshop on Software Specification and Design*, Marzo 1996.

[CM88] K. Mani Chandy y Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CM96a] Manuel G. Clavel y José Meseguer. Axiomatizing Reflective Logics and Languages. En *Proceedings of Reflection'96*, págs. 263–288, 1996.

[CM96b] Manuel G. Clavel y José Meseguer. Reflection and Strategies in Rewriting Logic. En *Proceedings of First International Workshop on Rewriting Logic and its Applications*, número 4 en Electronic Notes in Theoretical Computer Science. Elsevier Science, Septiembre 1996.

[CM98] Paolo Ciancarini y Cecilia Mascolo. Software Architecture and Mobility. En Jeff Magee y Dewayne E. Perry, editores, *Proceedings of the Third International Software Architecture Workshop*. ACM Press, Noviembre 1998.

[CM99] Paolo Ciancarini y Cecilia Mascolo. Specification and Analysis of Component Based Software Architectures. En Clements et al. [CPR99].

[CPR99] Paul C. Clements, Dewayne E. Perry y Alexander Ran, editores. *Electronic Proceedings of First Working IFIP Conference on Software Architecture (WICSA1)*. IFIP, Febrero 1999.

[CPT99] Carlos Canal, Ernesto Pimentel y José María Troya. Specification and Refinement of Dynamic Software Architectures. En *Software Architecture*, págs. 107–126. Kluwer Academic Publishing, Febrero 1999.

[CPT01] Carlos Canal, Ernesto Pimentel y José María Troya. Compatibility and Inheritance in Software Architectures. *Science of Computer Programming*, 41(2):105–138, 2001.

[Cra97] John Stephen Crane. *Dynamic Binding in Distributed Systems*. Tesis Doctoral, Imperial College of Science, Technologie and Medicine, University of London, Septiembre 1997.

[CRC98] José Angel Carsí, Isidro Ramos y J. Hilario Canós. La Metaclase OASIS. En *Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS'98)*, págs. 181–192, Torres, Brasil, Abril 1998.

[CSST98] Walter Cazzola, Andrea Savigni, Andrea Sosio y Francesco Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. En *Proceedings of* 6th *Reengineering Forum (REF'98)*, págs. 12–1–12–6. IEEE, Marzo 1998.

[CSST99] Walter Cazzola, Andrea Savigni, Andrea Sosio y Francesco Tisato. Architectural Reflection: Concepts, Design, and Evaluation. Technical Report RI-DSI 234-99, DSI, Universitá degli studi di Milano, Mayo 1999.

[CST00] Walter Cazzola, Robert J. Stroud y Francesco Tisato, editores. *Reflection and Software Engineering*, volumen 1826 de *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Alemania, Junio 2000.

[CTKI00] Shigeru Chiba, Michiaki Tatsubori, Marc-Olivier Killijian y Kozo Itano. OpenJava: a Class-based Macro System for Java. En Cazzola et al. [CST00], págs. 119–135.

[CV00] Carlos Canal Velasco. *Un Lenguaje para la Especificación y Validación de Arquitecturas de Software*. Tesis Doctoral, Universidad de Málaga, 2000.

[CW99] Paolo Ciancarini y Alexander L. Wolf, editores. *Third International Conference on Coordination Models and Languages (Coordination'99)*, volumen 1594 de *Lecture Notes in Computer Science*. Springer-Verlag, Abril 1999.

[Dam95] Laurent Dami. Functions, Records and Compatibility in the λN Calculus. En Oscar Marius Nierstrasz y Dionysios C. Tsichritzis, editores, *Object-Oriented Software Composition*, The Object-Oriented Series, capítulo 6, págs. 153–174. Prentice-Hall, 1995.

[DC95] Thomas R. Dean y James R. Cordy. A Syntactic Theory of Software Architecture. *IEEE Transactions on Software Engineering*, 21(4):302–313, Abril 1995.

[DC01a] Jim Dowling y Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Softare. En Yonezawa y Matsuoka [YM01], págs. 81–88.

[DC01b] Jim Dowling y Vinny Cahill. The K-Component Model and Dynamic Software Evolution. En Siobhán Clarke, Simon Dobson, Vinny Cahill y Robert Filman, editores, *Proceedings of the OOPSLA 2001 workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE'01)* , Tampa Bay, Florida, Octubre 2001.

[DCS76] Maria Luisa Dalla Chiara Scabia. *Lógica*. Labor, 1976.

[DeL99] Robert DeLine. *Resolving Packaging Mismatch*. Tesis Doctoral, Carnegie Mellon University, Julio 1999.

[DHN00] Stéphane Ducasse, T. Hofmann y Oscar Nierstrasz. OpenSpaces: an Object-Oriented Framework for Reconfigurable Coordination Spaces. En Porto y Catalin-Roman [PR00], págs. 1–19.

[Dij68] Edsger Wybe Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, Mayo 1968.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, Agosto 1975.

[DM95] François-Nicola Demers y Jacques Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. En *Proceedings of the IJCAI'95 workshop on Reflection and Meta-Level Architectures and their Applications in Artificial Intelligence*, págs. 29–38, Montreal, Agosto 1995.

[DM98] Francisco Durán y José Meseguer. Arquitecturas de Software desde una Perspectiva Formal. Conferencia en JIS'98, Noviembre 1998.

[DMC92] Christophe Dony, Jacques Malenfant y Pierre Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. En Andreas Paepcke, editor, *Proceedings of OOPSLA'92: Object-Oriented Programming Systems, Languages and Applications*, págs. 201–217, Vancouver, Canada, Octubre 1992. ACM, ACM Press. Also in SIGPLAN Notices 27(10).

[DMSMTH96] Giovanna Di Marzo Serugendo, Murhimanya Muhugusa, Christian Tschudin y Jürgen Harms. Survey of Theories for Mobile Agents. *Cahier du CUI*, 106, Noviembre 1996. Centre Universitaire d'Informatique, University of Geneva.

[DMT99] Grit Denker, José Meseguer y Carolyn Talcott. Rewriting Semantics of Distributed Meta Objects and Composable Communication Services, 1999.

[DNR99] Elisabetta Di Nitto y David S. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. En *Proceedings of ICSE'21*, Mayo 1999.

[Don99] Patrick Donohoe, editor. *Software Architecture – Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1).* Kluwer Academic Publishers, 1999.

[dP99] Virgínia Carvalho Carneiro de Paula. *Implementação de Linguagens de Configuração para Sistemas Distribuídos.* Tesis Doctoral, Universidade Federal de Pernambuco, Recife, Brasil, 1999.

[dPJC98] Virgínia Carvalho Carneiro de Paula, George Roger Ribeiro Justo y Paulo Roberto Freire Cunha. Specifying Dynamic Distributed Software Architectures. En *XII Simpósio Brasileiro de Engenharia de Software.* BCS Press, 1998.

[DR97] Stéphane Ducasse y Tamar Richner. Executable Connectors: Towards Reusable Design Elements. En *Software Engineering – Proceedings of ESEC/FSE'97*, volumen 1301 de *Lecture Notes in Computer Science*, págs. 483–500, 1997.

[dRS84] Jim des Rivières y Brian C. Smith. The Implementation of Procedurally Reflective Languages. En *Proceedings of ACM Symposium on Lisp and Functional Programming (L&FP'84)*, págs. 331–347, 1984.

[DS99] Rémi Douence y Mario Südholt. The Next 700 Reflective Object-Oriented Languages. Technical Report 99-1-INFO, École des Mines de Nantes, 1999.

[DSC+00] Jim Dowling, Tilman Schäfer, Vinny Cahill, Peter Haraszti y Barry Redmond. Using Reflection to Support Dynamic Adaptation of System Software: A Case Study Driven Evaluation. En Walter Cazzola, Robert J. Stroud y Francesco Tisato, editores, *Reflection and Software Engineering*, volumen 1826 de *Lecture Notes in Computer Science*, págs. 171–190. Springer Verlag, Junio 2000.

[Dul90] Naranker Dulay. *A Configuration Language for Distributed Programming*. Tesis Doctoral, Imperial College of Science, Technology and Medicine. University of London, Febrero 1990.

[Dul92] Naranker Dulay. δarwin Language Reference Manual. Technical report, Department of Computing, Imperial College, 1992.

[EHIP+96] Walter J. Ellis, Richard F. Hilliard III, Peter T. Poon, David Rayford, Thomas F. Saunders, Basil Sherlund y Ronald L. Wade. Toward a Recommended Practice for Architectural Description. En *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems*, Montreal, Quebec, Canadá, Octubre 1996. IEEE Architecture Planning Group.

[EN00] Uffe H. Engberg y Mogens Nielsen. A Calculus of Communicating Systems with Label Passing – Ten Years After. En Plotkin et al. [PST00].

[End94] Markus Endler. A Language for Implementing Generic Dynamic Reconfigurations of Distributed Programs. En *Proceedings of the* 12th *Brazilian Symposium on Computer Networks*, págs. 175–187, Curitiba, Mayo 1994.

[Eug01] Patrick Thomas Eugster. *Type-Based Publish/Subscribe*. Tesis Doctoral, École Polytechnique Fédérale de Lausanne, Diciembre 2001.

[EW92] Markus Endler y Jiawang Wei. Programming Generic Dynamic Reconfigurations for Distributed Applications. En *Proceedings of the* 1st *International Workshop on Configurable Distributed Systems*, págs. 68–79. IEE, 1992.

[FA93] Svend Frølund y Gul Agha. A Language Framework for Multi-Object Coordination. En Nierstrasz [Nie93], págs. 346–360.

[FA96] Svend Frølund y Gul A. Agha. Abstracting Interactions Based on Message Sets. En *Object-Based Models and Languages for Concurrent Systems*, volumen 924 de *Lecture Notes in Computer Science*, págs. 107–124. Springer-Verlag, 1996.

[Fer89] Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. En Meyrowitz [Mey89], págs. 317–326.

[FF96] Nissim Francez y Ira R. Forman. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*. Addison-Wesley, 1996.

[FG95] Cédric Fournet y Georges Gonthier. The reflexive CHAM and the join-calculus. En *Proceedings of POPL'96*, Octubre 1995.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget y Didier Rémy. A Calculus of Mobile Agents. En *Proceedings of CONCUR'96*, 1996.

[FHA99] Eric Freeman, Susanne Hupfer y Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Jini Technology Series. Addison-Wesley Publishing Co., Junio 1999.

[Fil94] Andrzej Filinksi. Representing Monads. En *Proceedings of ACM POPL'94*, págs. 446–457, 1994.

[FLMR00] Cédric Fournet, Cosimo Laneve, Luc Maranget y Didier Rémy. Inheritance in the Join Calculus. En *Foundations of Software Technology and Theoretical Computer Science*, volumen 1974 de *Lecture Notes in Computer Science*. Springer Verlag, Diciembre 2000.

[FM96] José Luiz Fiadeiro y Tom S.E. Maibaum. A Mathematical Toolbox for the Software Architect. En Jeff Kramer y Alexander Wolf, editores, *Proceedings* 8th *International Workshop on Software Specification and Design*, págs. 46–55. IEEE Computer Society Press, 1996.

[FM97] José Luiz Fiadeiro y Tom S. E. Maibaum. Categorial Semantics of Parallel Program Design. *Science of Computer Programming*, 28:111–138, 1997.

[FM98] Pascal Fradet y Daniel Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, Julio 1998.

[FML55] José Ferrater Mora y Hugues Leblanc. *Lógica Matemática*. Fondo de Cultura Económica, México, 1955.

[Foo89] Brian Foote. Class Warfare: Classes vs. Prototypes. En David Ungar, editor, *OOPSLA'89 Workshop on Objects without Classes*, New Orleans, Louisiana, Agosto 1989. ACM.

[For98] Thomas Forster. Quine's NF – 60 Years On. *American Mathematical Monthly*, Julio 1998.

[For00] Ira R. Forman. Superimposition: A Form of Separation of Concerns for Distributed Systems. En *Electronic Proceedings of First OOPSLA Workshop on Advanced Separation of Concerns (ASoC'00)*, Octubre 2000.

[Fos97] Halldor Fosså. *Interactive Configuration Management for Distributed Systems*. Tesis Doctoral, Imperial College of Science, Technology and Medicine. University of London, Abril 1997.

[FW84] Daniel P. Friedman y Mitchell Wand. Reification: Reflection Without Metaphysics. En *Proceedings of ACM Symposium on Lisp and Functional Programming*, págs. 348–355, 1984.

[GAO94] David Garlan, Robert Allen y John Ockerbloom. Exploiting Style in Architectural Design Environments. En *Proceedings of SIGSOFT: Foundations of Software Engineering (FSE'94)*, págs. 175–188, Diciembre 1994.

[GAO95] David Garlan, Robert Allen y John Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems Out of Existing Parts. En *Proceedings of ICSE'17*, Seattle, Washington, Abril 1995.

[Gar95] David Garlan. What is style? En *Proceedings of IWASS'1*, Abril 1995.

[Gar98] David Garlan. Higher-Order Connectors. En *Workshop on Compositional Software Architectures*, Enero 1998.

[Gar00] David Garlan. Software Architecture: a Roadmap. En Anthony Finkelstein y Jeff Kramer, editores, *The Future of Software Engineering*, págs. 91–101. ACM Press, Limerick, Ireland, Junio 2000. Proceedings of 22nd International Conference on Software Engineering (ICSE 2000), Vol. 2.

[Gay99] Simon Gay. Some Type Systems for the Pi Calculus. Technical report, Department of Computer Science, Royal Holloway, University of London, 1999.

[GC92] David Gelernter y Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, Febrero 1992.

[GC96] Brendan Gowing y Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. En *Proceedings of Reflection 96*, págs. 137–152, 1996.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[Gel82] David Gelernter. *An Integrated Microcomputer Network for Experiments in Distributed Programming.* Tesis Doctoral, State University of New York at Stony Brook, Octubre 1982.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Enero 1985.

[Ger96] Capt. Mark Gerken. Minutes of the First EDCS Architecture Cluster Workshop. Napa Valley, DARPA/SEI/RL, Octubre 1996.

[GHJ+95a] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides y Grady Booch. *Design Patterns.* Addison-Wesley, 1995.

[GHJ+95b] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides y Grady Booch. *Design Patterns*. Addison-Wesley, 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Octubre 1994.

[Gia99] Dimitra Giannakopoulou. *Model Checking for Concurrent Software Architectures*. Tesis Doctoral, Imperial College of Science, Technology and Medicine. University of London, Enero 1999.

[GK96] Kaveh Moazami Goudarzi y Jeff Kramer. Maintaining Node Consistency in the Face of Dynamic Change. En *Proceedings of the Third International Conference on Configurable Distributed Systems*, págs. 62–69. IEEE Computer Society Press, Mayo 1996.

[GLM97] David Garlan y Daniel Le Métayer, editores. *Second International Conference on Coordination Languages and Models (Coordination'97)*, volumen 1282 de *Lecture Notes in Computer Science*, Berlin, Septiembre 1997. Springer Verlag.

[GMW97] David Garlan, Robert T. Monroe y David Wile. ACME: An Architectural Interchange Language. En *Proceedings of CASCON'97*, Noviembre 1997.

[GN91] David Garlan y David Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. En *Proceedings of VDM'91: Formal Software Development Methods*, volumen 551 de *Lecture Notes in Computer Science*. Springer-Verlag, Octubre 1991.

[Gol97] Michael Golm. Design and Implementation of a Meta Architecture for Java. Tesis de Maestría, Institut für Mathematische und Datenverarbeitung der Friedrich-Alexander-Universität Erlangen-Nürnberg, Enero 1997.

[Gor00] Andrew D. Gordon. Notes on Nominal Calculi for Security and Mobility. Lecture Notes for FOSAD'00 Summer School, Julio 2000.

[GP95] David Garlan y Dewayne E. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), Abril 1995.

[GQ94] Michael M. Gorlick y Alex Quilici. Visual Programming in the Large versus Visual Programming in the Small. En *Proceedings of* 10th *IEEE Conference on Visual Languages*, págs. 137–144, St. Louis, Noviembre 1994.

[Gör90] Steven Kent Göring. *A Graph Grammar Approach to Concurrent Programming – Delta Grammars*. Tesis Doctoral, University of Illinois, Department of Computer Science, 1990. UIUC-DCS-R-90-1576.

[GS93] David Garlan y Mary Shaw. An Introduction to Software Architecture. Technical Report CMU/SEI-94-TR-21, Software Engineering Institute, Carnegie Mellon University, 1993.

[GW00] Philippa Gardner y Lucian Wischik. Explicit Fusions. En Mogens Nielsen y Branislav Rovan, editores, 25th *International Symposium on the Mathematical Foundations of Computer Science (MFCS 2000)*, volumen 1893 de *Lecture Notes in Computer Science*, págs. 373–382, Bratislava, Septiembre 2000. Springer.

[Har95] John Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report SRI-TR CRC-053, University of Cambridge, Computer Laboratory, SRI, Febrero 1995.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3), 1977.

[HIM99] Dan Francisco Hirsch, Paola Inverardi y Ugo Montanari. Modelling Software Architectures and Styles with Graph Grammars and Constraint Solving. En Donohoe [Don99], págs. 127–143.

[Höl93] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. En Nierstrasz [Nie93], págs. 36–56.

[HL95] Walter Hürsch y Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Febrero 1995.

[HLMS92] Chris Hankin, Daniel Le Métayer y David Sands. A Calculus of Gamma Programs. En *Fifth Workshop on Languages and Compilers for Parallel Computing*, volumen 757 de *Lecture Notes in Computer Science*, Yale, 1992. Springer Verlag.

[HM94] Richard C. Holt y Spiros Mancoridis. Using Tube Graphs to Model Architectural Designs of Software Systems. Technical Report 308, University of Toronto, CSRI, Octubre 1994.

[HM99] Dan Francisco Hirsch y Ugo Montanari. Consistent Transformations for Software Architecture Styles of Distributed Systems. En Gheorghe Stefanescu, editor, *Workshop on Distributed Systems*, volumen 28 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[HNS00] Christine Hofmeister, Robert Nord y Dilip Soni. *Applied Software Architecture*. The Object Technology Series. Addison-Wesley Longman, 2000.

[Hoa85] C. Anthony R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol96a] Ric C. Holt. Binary Relational Algebra Applied to Software Architecture. Technical Report CSRI TR-345, University of Toronto, Junio 1996.

[Hol96b] A. A. Holzbacher. A Software Environment for Concurrent Coordinated Programming. En Ciancarini y Hankin [CH96], págs. 249–266.

[Hol98] Richard C. Holt. Structural Manipulations of Software Architecture Using Tarski Relational Algebra. En 5th *Working Conference on Reverse Engineering*, Octubre 1998.

[Hol99] David Holmes. *Synchronisation Rings: Composable Synchronization for Object-Oriented Systems*. Tesis Doctoral, Department of Computing, Macquarie University, Sydney, Australia, 1999.

[HP93] Christine Hofmeister y James M. Purtilo. Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement. En *Proceedings of* 13th *International Conference on Distributed Computing Systems*, págs. 101–110. IEEE CS Press, Mayo 1993.

[HT91] Kohei Honda y Mario Tokoro. An Object Calculus for Asynchronous Communication. En Pierre America, editor, *ECOOP'91: European Conference on Object-Oriented Programming*, volumen 512 de *Lecture Notes in Computer Science*, págs. 133–147, Geneve, Switzerland, Julio 1991. Springer-Verlag.

[Hun81] Geoffrey Hunter. *Metalógica: Introducción a la Metateoría de la Lógica Clásica de Primer Orden*. Lógica y Teoría de la Ciencia. Paraninfo, 1981.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos y Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-based Programming. En Chris Hankin, editor, *Programming Languages and Systems: Proceedings of the* 7th *European Symposium on Programming (ESOP'98)*, volumen 1381 de *Lecture Notes in Computer Science*, págs. 122–138. Springer Verlag, Abril 1998.

[IB96] Valérie Issarny y Christophe Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. En *Proceedings of ICDCS'96*, Mayo 1996.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo y Waldemar Celes Filho. Lua – an Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[IW95] Paola Inverardi y Alexander L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, Abril 1995.

[IWY97] Paola Inverardi, Alexander L. Wolf y Daniel Yankelevich. Checking Assumptions in Component Dynamics at the Architecture Level. En Garlan y Le Métayer [GLM97], págs. 46–63.

[IWY98] Paola Inverardi, Alexander L. Wolf y Daniel Yankelevich. Behavioural Type Checking of Architectural Components Based on Assumptions. Technical Report CU-CS-861-98, University of Colorado at Boulder, Abril 1998.

[IWY00] Paola Inverardi, Alexander L. Wolf y Daniel Yankelevich. Static Checking of System Behaviors Using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 2000.

[Jac98] Bart Jacobs. Coalgebraic Reasoning About Classes in Object-Oriented Languages. *Electronic Notes in Theoretical Computer Science*, 11:1–12, 1998.

[JC92] George Roger Ribeiro Justo y Paulo Roberto Freire Cunha. Programming Distributed Systems with Configuration Languages. En *Proceedings of the* 1st *International Workshop on Configurable Distributed Systems*, págs. 118–127. IEE, Marzo 1992.

[JF90] Ramón Jansana Ferrer. *Una Introducción a la Lógica Modal.* Cuadernos de Filosofía y Ensayo. Tecnos, 1990.

[Jon96] Cliff B. Jones. Accommodating Interference in the Formal Design of Concurrent Object-Based Programs. *Formal Methods in System Design*, 8(2):105–122, Marzo 1996.

[JR97] Bart Jacobs y Jan J.M.M. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of EATCS*, 62:222–259, 1997.

[JRvdL00] Mehdi Jazayeri, Alexander Ran y Frank van der Linden, editores. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley, 2000.

[JS96] Yuh-Jzer Joung y Scott A. Smolka. A Comprehensive Study of the Complexity of Multiparty Interaction. *Journal of the ACM*, 43(1):75–115, 1996.

[Kam92] Fairouz Kamareddine. Set Theory and Nominalisation, Part I. *Journal of Logic and Computation*, 2(5):579–604, 1992.

[Kat93] Shmuel Katz. A Superimposition Control Construct for Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, Abril 1993.

[Ken95] John J. Kenney. *Executable Formal Models of Distributed Transaction Systems based on Event Processing.* Tesis Doctoral, University of Stanford, CSL, Diciembre 1995.

[KFRGC98a] Marc-Olivier Killijian, Jean-Charles Fabre, Juan-Carlos Ruiz-García y Shigeru Chiba. A Metaobject Protocol for Fault-Tolerant CORBA Applications. En *Proceedings of the* 17th *IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, págs. 127–134. IEEE Press, 1998.

[KFRGC98b] Marc-Olivier Killijian, Jean-Charles Fabre, Juan-Carlos Ruiz-García y Shigeru Chiba. A Metaobject Protocol for Fault-Tolerant CORBA Applications. Technical Report RR-98-139, Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique (LAAS-CNRS), Toulouse-Cedex, France, Abril 1998.

[KG99] Shmuel Katz y Yossi Gil. Aspects and Superimpositions. En *Electronic Proceedings of Third ECOOP Workshop on Aspect-Oriented Programming (AOP'99)*, Junio 1999.

[KGO+00] Rohit Khare, Michael Guntersdorfer, Peyman Oreizy, Nenad Medvidovic y Richard N. Taylor. xADL: Enabling Architecture-Centric Tool Integration With XML. Technical Report UCI-ICS-00-12, Department of Information and Computer Science, University of California, Irvine, Abril 2000.

[KHL+93] Gail E. Kaiser, Wenwey Hseush, James C. Lee, Shyhtsun F. Wu, Esther Woo, Eric Hilsdale y Scott Meyer. MELDC: A Reflective Object-Oriented Coordination Language. Technical Report CUCS-001-93, Department of Computer Science, Columbia University, Nueva York, Enero 1993.

[Kir92] Graham N. C. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems.* Tesis Doctoral, University of St. Andrews, Escocia, 1992. Department of Mathematical and Computational Sciences.

[KKVvV01] Rick Kazman, Philippe Kruchten, Chris Verhoef y Hans van Vliet, editores. *Proceedings of the Second Working IEEE/IFIP Conference on Software Architecture (WICSA'01).* IEEE Computer Society Press, Amsterdam, Agosto 2001.

[KL01] Fairouz Kamareddine y Twan Laan. A Correspondence between Martin-Löf Type Theory, the Ramified Theory of Types and Pure Type Systems. *Journal of Logic, Language and Information*, 10(3):375–402, 2001. Kluwer Academic Publishers.

[Kle74] Stephen Cole Kleene. *Introducción a la Metamatemática.* Estructura y Función. Tecnos, 1974.

[KLM94] Dinesh Katiyar, David C. Luckham y John Mitchell. A type system for prototyping languages. En *Proceedings of* 21st *ACM Symposium on Principles of Programming Languages (PoPL'94)*, Portland, Oregón, 1994. ACM Press.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier y John Irwin. Aspect-Oriented Programming. En Mehmet Aksit y Satoshi Matsuoka, editores, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, número 1241 en Lecture Notes in Computer Science, págs. 220–242, Jyvaskyla, Finland, Junio 1997. Springer Verlag.

[KM85] Jeff Kramer y Jeffrey N. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, 1985.

[KM90] Jeff Kramer y Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16, Noviembre 1990.

[KM96a] Rudolf Kaehr y Thomas Mahler. Introducing and Modeling Polycontextural Logics. En 13th *European Meeting on Cybernetics and Systems Research*, Abril 1996.

[KM96b] Jeff Kramer y Jeff Magee. Self Organising Software Architectures. En L. Vidal, A. Finkelstein, G. Spanoudakis y Alexander L. Wolf, editores, *Joint Proceedings of the SIGSOFT'96 Workshops*, págs. 35–38. ACM Press, 1996.

[KMS98] Graham Kirby, Ron Morrison y David Stemple. Linguistic Reflection in Java. Technical Report PL/9810027, University of St. Andrews, Scotland, Octubre 1998.

[Kru95] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6), Noviembre 1995.

[Kru99] Philippe Kruchten. *The Rational Unified Process - An Introduction*. Object Technology Series. Addison-Wesley, 1999.

[LAN00] Markus Lumpe, Franz Achermann y Oscar M. Nierstrasz. A Formal Language for Composition. En Gary T. Leavens y Murali Sitaraman, editores, *Foundations of Component-Based Systems*, págs. 69–90. Cambridge University Press, 2000.

[Lea94] Doug Lea. Christopher Alexander: An Introduction for Object-Oriented Designers. *ACM Software Engineering Notes*, Enero 1994.

[Led98] Thomas Ledoux. *Réflexion dans les Systémes Répartis: application à CORBA et SmallTalk*. Tesis Doctoral, Université de Nantes, École des Mines de Nantes, Marzo 1998.

[Led99] Thomas Ledoux. OpenCorba: a Reflective Open Broker. En Pierre Cointe, editor, *Proceedings of the Second International Conference on Reflection (Reflection'99)*, volumen 1616 de *Lecture Notes in Computer Science*, págs. 197–214, Saint Malo, France, Julio 1999. Springer Verlag.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan y Walter Mann. Specification and Analysis of System Architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Abril 1995.

[LM94] Daniel Le Métayer. Higher-Order Multiset Programming. En *DIMACS Workshop on Specifications of Parallel Algorithms*, volumen 18 de *DIMACS Series in Discrete Mathematics*. American Mathematical Society, 1994.

[LM96] Daniel Le Métayer. Software Architecture Styles As Graph Grammars. En *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, págs. 15–24. ACM Press, 1996.

[LM98] Daniel Le Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, Julio 1998.

[LP91] Wilf LaLonde y John Pugh. Subclassing ≠ Subtyping ≠ Is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, Enero 1991.

[LSLL00] Orlando Loques, Alexandre Sztajnberg, Julius Leite y Marcelo Lobosco. On the Integration of Configuration and Meta-Level Programming Approaches. En Cazzola et al. [CST00], págs. 191–210.

[LSN96] Markus Lumpe, Jean-Guy Schneider y Oscar Marius Nierstrasz. Using Metaobjects to Model Concurrent Objects with Pict. En *Proceedings of Languages et Modèles à Objets*, págs. 1–12, Leysin, Suiza, Octubre 1996.

[Luc97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. Tesis Doctoral, Vrije Universiteit Brussel, Septiembre 1997.

[Lum99] Markus Lumpe. *A π-Calculus Based Approach for Software Composition*. Tesis Doctoral, Institut für Informatik und angewandte Mathematik, Universität Bern, Switzerland, Enero 1999.

[LV95] David C. Luckham y James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Septiembre 1995.

[LVB+93] David C. Luckham, James Vera, Doug Bryan, Larry Augustin y Frank Belz. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems. *Journal of Systems and Software*, 21(3):253–265, Junio 1993.

[LVM95] David C. Luckham, James Vera y Sigurd Meldal. Three Concepts of System Architecture. Technical Report CSL-TR-95-674, University of Stanford, CSL, 1995.

[LW95] Xinxin Liu y David Walker. A Polymorphic Type System for the Polyadic π-calculus. En Insup Lee y Scott A. Smolka, editores, *Proceedings of Sixth International Conferencia on Concurrency Theory (CONCUR 1995)*, volumen 962 de *Lecture Notes in Computer Science*, págs. 103–116, Philadelphia, Agosto 1995. Springer.

[LW00] Xinxin Liu y David Walker. Concurrent Objects as Mobile Processes. En Plotkin et al. [PST00].

[LWF01] Antónia Lopes, Michel Wermelinger y José Luiz Fiadeiro. Higher-Order Architectural Connectors. Technical report, Universidade de Lisboa, 2001.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mae87a] Pattie Maes. Concepts and Experiments in Computational Reflection. En Meyrowitz [Mey87], págs. 147–155.

[Mae87b] Pattie Maes. *Concepts and Experiments in Computational Reflection*. Tesis Doctoral, Vrije Universiteit Brussel, 1987.

[Mar99] Eric Marsden. Description Formelle d'un Protocole à Métaobjets. Tesis de Maestría, Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique (LAAS-CNRS), Toulouse-Cedex, France, Junio 1999.

[Mar01] Claudia A. Marcos. *Patrones de Diseño como Entidades de Primera Clase*. Tesis Doctoral, Universidad Nacional del Centro de la Provincia de Buenos Aires, Instituto de Sistemas ISISTAN, Tandil, Argentina, Abril 2001.

[Mas99] Cecilia Mascolo. MobiS: A Specification Language for Mobile Systems. En Ciancarini y Wolf [CW99], págs. 37–52.

[MC94] Thomas W. Malone y Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, Marzo 1994.

[McA95] Jeff McAffer. Meta-Level Programming with CodA. En Walter Olthoff, editor, *Proceedings of the* 9th *European Conference on Object-Oriented Programming (ECOOP'95)*, volumen 952 de *Lecture Notes in Computer Science*, págs. 190–214. Springer Verlag, 1995.

[McA96] Jeff McAffer. Engineering the Meta-Level. En *Proceedings of Reflection'96*, págs. 39–61, Abril 1996.

[McH94] Ciaran McHale. *Synchronisation in Concurrent, Object-Oriented Languages: Expressive Power, Genericity and Inheritance*. Tesis Doctoral, Trinity College, University of Dublin, Octubre 1994.

[McI69] M. Douglas McIlroy. Mass-Produced Software Components. En Peter Naur y Brian Randell, editores, *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, págs. 138–150, Garmisch, Alemania, Octubre 1969.

[MDEK95] Jeff N. Magee, Naranker Dulay, Susan Eisenbach y Jeff Kramer. Specifying Distributed Software Architectures. En *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, Septiembre 1995.

[Med96a] Nenad Medvidovic. ADLs and Dynamic Architecture Changes. En *Proceedings of the Second International Software Architecture Workshop (ISAW'2/SIGSOFT'96)*, págs. 24–27, San Francisco, CA, 1996.

[Med96b] Neno Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, University of California Irvine, Febrero 1996.

[Med99] Nenad Medvidovic. *Architecture-Based Specification-Time Software Evolution*. Tesis Doctoral, University of California Irvine, 1999.

[Mei95] Theo Dirk Meijler. Viewing Software Composition as Composition of Metaobjects. Technical report, University of Bern, 1995.

[Mes90] José Meseguer. A Logical Theory of Concurrent Objects. En *Proceedings of ECOOP-OOPSLA'90 Conference on Object-Oriented Programming*, págs. 101–115, Ottawa, Canadá, Octubre 1990. ACM Press.

[Mes92] José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes96] José Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report. En *Proceedings of CONCUR'96*, número 1119 en Lecture Notes in Computer Science, págs. 331–372. Springer Verlag, 1996.

[Mey87] Norman Meyrowitz, editor. *OOPSLA'87 Conference Proceedings*, volumen 22(12) de *SIGPLAN Notices*. ACM Press, Diciembre 1987.

[Mey89] Norman Meyrowitz, editor. *OOPSLA'89 Conference Proceedings*, volumen 24(10) de *SIGPLAN Notices*. ACM Press, Octubre 1989.

[MG99] Kaveh Moazami-Goudarzi. *Consistency-Preserving Dynamic Reconfiguration of Distributed Systems*. Tesis Doctoral, Imperial College of Science, Technology and Medicine. University of London, Marzo 1999.

[MHSA98] Juan M. Murillo, Juan Hernández, Fernando Sánchez y Luis A. Alvarez. Roles y Coordinadores de Roles: Componentes de Coordinación para Objetos Activos. En *Actas de las III Jornadas de Ingeniería del Software*, págs. 231–242, Murcia, Noviembre 1998. Diego Marín.

[Mil80] A. J. Robin G. Milner. *A Calculus of Communicating Systems*, volumen 92 de *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil89] Arthur J. Robin G. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil92] Robin Milner. Functions as Processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil93a] Robin Milner. Elements of Interaction. *Communications of the ACM*, 36(1):78–89, Enero 1993. Turing Award Lecture.

[Mil93b] Robin Milner. The Polyadic π-calculus: A Tutorial. En F.L. Bauer, W. Brauer y H. Schwichtenberg, editores, *Logic and Algebra of Specification*, págs. 203–246. Springer Verlag, 1993.

[Mil96] Robin Milner. Calculi for Interaction. *Acta Informatica*, 33:707–737, 1996.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, Junio 1999.

[Mil01] Robin Milner. Bigraphical Reactive Systems. En Kim Guldstrand Larsen y Mogens Nielsen, editores, *CONCUR 2001 - Proceedings of* 12th *International Conference on Concurrency Theory*, volumen 2154 de *Lecture Notes in Computer Science*, págs. 16–35, Aalborg, Denmark, Agosto 2001. Springer.

[Min97] Naftaly Minsky. Towards Architectural Invariants of Evolving Systems. Technical report, Rutgers University, New Brunswick, NJ, Febrero 1997.

[MJD96] Jacques Malenfant, Marco Jacques y François-Nicola Demers. A Tutorial on Behavioural Reflection and its Implementation. En *Proceedings of Reflection 96*, págs. 1–20, 1996.

[MK96] Jeff Magee y Jeff Kramer. Dynamic Structure in Software Architectures. *ACM Software Engineering Notes*, 21(6):3–14, Noviembre 1996.

[MKG97] Jeff Magee, Jeff Kramer y Dimitra Giannakopoulou. Analysing the Behaviour of Distributed Software Architectures: a Case Study. En 5th *IEEE Workshop on Future Trends of Distributed Computing Systems*, Octubre 1997.

[MKG99] Jeff Magee, Jeff Kramer y Dimitra Giannakopoulou. Behaviour Analysis of Software Architectures. En *Software Architecture*, págs. 35–49. Kluwer Academic Publishing, Febrero 1999.

[MKS89] Jeff Magee, Jeff Kramer y Morris Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6), 1989.

[MKS94] Jeff Magee, Jeff Kramer y Morris Sloman. Regis: A Constructive Development Environment for Distributed Programs. *Distributed Systems Engineering*, 1(5):663–675, 1994.

[MM00] Raphael Malveau y Thomas Mowbray. *The Software Architect Bootcamp*. Prentice-Hall, Octubre 2000.

[MMP00] Nikunj R. Mehta, Nenad Medvidovic y Sandeep Phadke. Towards a Taxonomy of Software Connectors. En *Proceedings of the* 22nd *International Conference on Software Engineering (ICSE 2000)*, Junio 2000.

[MN88] Pattie Maes y Daniele Nardi, editores. *Meta-Level Architectures and Reflection*. Elsevier Science – North-Holland, Febrero 1988.

[MN97] Theo Dirk Meijler y Oscar Marius Nierstrasz. Beyond Objects: Components. En M.P. Papazoglou y G. Schlageter, editores, *Cooperative Information Systems: Current Trends and Directions*, págs. 49–78. Academic Press, Noviembre 1997.

[MNS95] Gail C. Murphy, David Notkin y Kevin Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. En *Proceedings of the Third ACM Symposium on the Foundations of Software Engineering (SIGSOFT-FSE'95)*, volumen 20 de *ACM Software Engineering Notes*, págs. 18–28, Washington, DC, Octubre 1995. ACM Press.

[MOM96] Narciso Martí-Oliet y José Meseguer. Rewriting Logic as a Logical and Semantic Framework. En *Proceedings of First International Workshop on Rewriting Logic and its Applications*, volumen 4 de *Electronic Notes in Theoretical Computer Science*, Septiembre 1996.

[Mon98] Robert T. Monroe. Capturing Software Architecture Design Expertise With Armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, Octubre 1998.

[MPW92a] Robin Milner, Joachim Parrow y David Walker. A Calculus of Mobile Processes, Part I. *Information and Computation*, 100:1–40, 1992.

[MPW92b] Robin Milner, Joachim Parrow y David Walker. A Calculus of Mobile Processes, Part II. *Information and Computation*, 100:41–77, 1992.

[MR87] Naftaly H. Minsky y David Rozenshtein. A Law-Based Approach to Object-Oriented Programming. En Meyrowitz [Mey87], págs. 482–493.

[MR89] Naftaly H. Minsky y David Rozenshtein. Controllable Delegation: An Exercise in Law-Governed Systems. En Meyrowitz [Mey89], págs. 371–380.

[MR99] Nenad Medvidovic y David S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. En *Software Architecture*, págs. 161–182. Kluwer Academic Publishing, Febrero 1999.

[MRT98] Nenad Medvidovic, David S. Rosenblum y Richard N. Taylor. A Type Theory for Software Architectures. Technical Report UCI-ICS-98-04, University of California Irvine, Febrero 1998.

[MRT99] Nenad Medvidovic, David S. Rosenblum y Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. En *Proceedings of ICSE'21*, Mayo 1999.

[MT97a] Nenad Medvidovic y Richard N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. En *Software Engineering – Proceedings of ESEC/FSE'97*, págs. 60–76. ACM Press, Septiembre 1997.

[MT97b] José Meseguer y Carolyn Talcott. Semantic Interoperation of Dynamic Heterogeneous Architectures. Part I: A Case Study in Semantic Interoperation of Dynamic Hetereogeneus Architectures. Seattle EDCS Workshop, Julio 1997.

[MT97c] José Meseguer y Carolyn Talcott. Semantic Interoperation of Dynamic Heterogeneous Architectures. Part I: Rewriting Logic as a Semantic Framework for Formal Interoperation of Architecture Styles and Languages. Santa Fe EDCS Architecture Cluster Meeting, Abril 1997.

[MT98] Nenad Medvidovic y Richard N. Taylor. Separating Fact from Fiction in Software Architecture. En *Proceedings of ISAW'3*, págs. 105–108. ACM Press, 1998.

[MT00] Nenad Medvidovic y Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Enero 2000.

[MTW96] Nenad Medvidovic, Richard N. Taylor y E. James Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. En *Proceedings of California Symposium 1996*, págs. 28–40, Abril 1996.

[MU97] Naftaly Minsky y Victoria Ungureanu. Regulated Coordination in Open Distributed Systems. En Garlan y Le Métayer [GLM97].

[MU99] Naftaly Minsky y Victoria Ungureanu. Law-Governed Interaction: A Coordination & Control Mechanism for Hetereogeneus Distributed Systems. Draft, 1999.

[MU00] Naftaly Minsky y Victoria Ungureanu. Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, Julio 2000.

[MUWZ96] Naftaly Minsky, Victoria Ungureanu, Wenhui Wang y Junbiao Zhang. Building Reconfiguration Primitives into the Law of a System. En *International Conference on Configurable Distributed Systems (ICCDS'96)*, Mayo 1996.

[NACO97] Gleb Naumovich, George S. Avrunin, Lori A. Clarke y Leon J. Osterweil. Applying Static Analysis to Software Architectures. En *Proceedings of the* 6th *European Software Engineering Conference and* 5th *ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97)*, volumen 1301 de *Lecture Notes in Computer Science*, págs. 77–93, Zurich, Septiembre 1997. Springer Verlag.

[Ng92] Keng Ng. *Visual Support for Distributed Programming.* Tesis Doctoral, Imperial College of Science, Technology and Medicine. University of London, 1992.

[Nie93] Oscar Marius Nierstrasz, editor. *ECOOP'93 – Object-Oriented Programming*, volumen 707 de *Lecture Notes in Computer Science*. Springer-Verlag, Julio 1993.

[NY93] Shojiro Nishio y Akinori Yonezawa, editores. *Object Technologies for Advanced Software*, volumen 742 de *Lecture Notes in Computer Science*, Kanazawa, Japón, Noviembre 1993. Springer Verlag.

[OB98] Alexandre Oliva y Luiz Eduardo Buzato. Composition of Meta-Objects in Guaraná. En *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998. Updates IC-98-33.

[OCGB98] Alexandre Oliva, Islene Calciolari Garcia y Luiz Eduardo Buzato. The Reflective Architecture of Guaraná. Technical Report IC-98-14, Instituto de Computaçao, Universidade Estadual de Campinas, Septiembre 1998.

[OD01] Andrea Omicini y Enrico Denti. From Tuple Spaces to Tuple Centres. *Science of Computer Programming*, 41(3):277–294, Noviembre 2001.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum y Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, Mayo 1999.

[OIT93] Hideaki Okamura, Yutaka Ishikawa y Mario Tokoro. Metalevel Decomposition in AL-1/D. En Nishio y Yonezawa [NY93], págs. 110–127.

[OK02] Harold Ossher y Gregor Kiczales, editores. *Proceedings of First International Conference on Aspect-Oriented Sofware Development (AOSD'02)*, Enschede, The Netherlands, Abril 2002. ACM Press.

[OKH+95] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz y Vincent Kruskal. Subject-Oriented Composition Rules. *ACM SIGPLAN Notices*, 30(10):235–250, Octubre 1995. Proceedings of OOPSLA'95.

[Oli98] Alexandre Oliva. Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java. Tesis de Maestría, Universidade Estadual de Campinas, Septiembre 1998.

[OMT98] Peyman Oreizy, Nenad Medvidovic y Richard N. Taylor. Architecture-Based Runtime Software Evolution. En *Proceedings of ICSE'98*, Kyoto, Japan, Abril 1998.

[Ore96] Peyman Oreizy. Issues in the Runtime Modification of Software Architectures. Technical Report UCI-ICS-96-35, University of California Irvine, Agosto 1996.

[OT98] Peyman Oreizy y Richard N. Taylor. On the Role of Software Architectures in Runtime System Reconfiguration. *IEE Proceedings-Software*, 145(8), Octubre 1998.

[OT00] Harold Ossher y Peri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. En *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[OZ99] Andrea Omicini y Franco Zambonelli. Coordination for Internet Application Development. *Journal of Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, Septiembre 1999. Special Issue on Coordination Mechanisms for Web Agents.

[PA97] George A. Papadopoulos y Farhad Arbab. Control-driven Coordination Programming in Shared Dataspace. En *Proceedings of PaCT'97*, volumen 1277 de *Lecture Notes in Computer Science*, págs. 247–261. Springer Verlag, 1997.

[PA98a] George A. Papadopoulos y Farhad Arbab. Coordination Models and Languages. Technical Report SEN-R9834, Centrum voor Wiskunde en Informatica (CWI), Diciembre 1998.

[PA98b] Georgis A. Papadopoulos y Farhad Arbab. Coordination Models and Languages. En *The Engineering of Large Systems*, volumen 46 de *Advances in Computers*. Academic Press, Agosto 1998.

[PA00] George A. Papadopoulos y Farhad Arbab. Dynamic Reconfiguration in Coordination Languages. En *Proceedings of High-Performance Computing and Networking 2000 (HPCN 2000)*, volumen 1823 de *Lecture Notes in Computer Science*, págs. 197–206, Amsterdam, 2000. Springer Verlag.

[PA01] George A. Papadopoulos y Farhad Arbab. Configuration and Dynamic Reconfiguration of Components Using the Coordination Paradigm. *Future Generation Computer Systems*, 17(8):1023–1038, Junio 2001.

[Par72] David Lorge Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, Diciembre 1972.

[Par01] Joachim Parrow. An Introduction to the π-calculus. En Bergstra et al. [BPS01], págs. 479–543.

[PBC00] Jane Pryor, Natalio Bastán y Marcelo Campo. A Reflective Approach to Support Aspect Oriented Programming in Java. En Álvaro Ortigosa y Claudia A. Marcos, editores, *Proceedings of First Argentine Symposium on Software Engineering (ASSE 2000), en Jornadas Argentinas de Informática e Investigación Operativa (29 JAIIO)*, Buenos Aires, Argentina, Septiembre 2000.

[PD99] Norman W. Paton y Oscar Díaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, Marzo 1999.

[Per85] Don Perlis. Languages with Self-reference I: Foundations (or: We can have everything in First-Order Logic!). *Artificial Intelligence*, 25:301–322, 1985.

[Per88] Don Perlis. Commonsense Set Theory. En Maes y Nardi [MN88].

[Per97] Dewayne E. Perry. State-of-the-Art: Software Architecture. En *Proceedings of the* 19th *International Conference on Software Engineering*, págs. 590–591. ACM Press, 1997.

[Pie98] Benjamin C. Pierce. *Programming in the Pi-Calculus. A Tutorial Introduction to Pict*. Indiana University, Marzo 1998. (Pict Version 4.1).

[Pit01] Andrew M. Pitts. Nominal Logic: A First Order Theory of Names and Binding. En Naoki Kobayashi y Benjamin C. Pierce, editores, *Fourth International Symposium on Theoretical Aspects of Computer Software (TACS 2001)*, volumen 2115 de *Lecture Notes in Computer Science*, págs. 219–242, Sendai, Japan, Octubre 2001. Springer-Verlag.

[Plo75a] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo75b] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PR00] António Porto y Gruia-Catalin Roman, editores. *Fourth International Conference on Coordination Models and Languages (Coordination'00)*, volumen 1906 de *Lecture Notes in Computer Science*, Limassol, Cyprus, Septiembre 2000. Springer Verlag.

[Pra86] Vaughan Pratt. Modelling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15:33–71, 1986.

[Pra95] Vaughan R. Pratt. Chu Spaces and their Interpretation as Concurrent Objects. En J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volumen 1000 de *Lecture Notes in Computer Science*, págs. 392–405. Springer Verlag, 1995.

[Pry00] Nathaniel G. Pryce. *Component Interaction in Distributed Systems*. Tesis Doctoral, Imperial College of Science, Technology and Medicine. University of London, Enero 2000.

[PS96] Benjamin C. Pierce y Davide Sangiorgi. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science*, 6(5), 1996.

[PS98] John Peterson y Martin Sulzmann. Analysis of Architectures using Constraint-Based Types. Technical Report YALEU/DCS/RR-1157, Department of Computer Science, University of Yale, 1998.

[PS99] John Peterson y Martin Sulzmann. Analysis of Architectures using Constraint-Based Types. En Clements et al. [CPR99].

[PS00] Benjamin C. Pierce y Davide Sangiorgi. Behavioural Equivalence in the Polymorphic π-calculus. *Journal of the ACM*, 47(5):531–584, 2000.

[PST00] Gordon Plotkin, Colin Stirling y Mads Tofte, editores. *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Foundations of Computing. MIT Press, Mayo 2000.

[PT98] Benjamin C. Pierce y David N. Turner. *Pict Libraries Manual*. Indiana University, University of Glasgow, Marzo 1998. Pict 4.1.

[PT00] Benjamin C. Pierce y David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. En Plotkin et al. [PST00].

[Pur90] James M. Purtilo. The POLYLITH Software Bus. Technical Report CS-TR-2469, University of Maryland, Institute for Advanced Computer Studies, Septiembre 1990.

[Pur94] James M. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[PW92] Dewayne E. Perry y Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM Software Engineering Notes*, 17(4), Octubre 1992.

[Qui85] Willard Van Orman Quine. Events and Reification. En E. Lepore y B. McLaughlin, editores, *Actions and Events*, págs. 162–171. Blackwell, 1985.

[Qui92] Willard Van Orman Quine. *La Búsqueda de la Verdad*. Crítica, 1992. Trad. Javier Rodríguez Alcázar.

[Ram25] Frank P. Ramsey. The Foundations of Mathematics. *Proceedings of the London Mathematical Society*, págs. 338–384, 1925.

[Ran99] Alexander Ran. Software Isn't Built From Lego Blocks. En *Proceedings of ACM Symposium on Software Reusability (SSR'99)*, 1999.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy y William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[RC00] Barry Redmond y Vinny Cahill. Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java. En Cazzola et al. [CCL00a].

[RE96] Matthias Radestock y Susan Eisenbach. Coordination in Evolving Systems. En O. Spaniol, C. Linnhoff-Popien y B. Meyer, editores, *Trends in Distributed Systems - CORBA and Beyond*, volumen 1161 de *Lecture Notes in Computer Science*, págs. 162–176. Springer-Verlag, Berlin, 1996.

[Rid99a] François-René Rideau. Métaprogrammation et Libre Disponibilité des Sources. Deux défis informatiques d'aujourd'hui. En *Autour du Libre 1999*, 1999.

[Rid99b] François-René Rideau. Reflection, Non-Determinism, and the λ-Calculus. Rapport de recherche à paraître, CNET, 1999.

[RMP97] Gruia-Catalin Roman, Peter J. McCann y Jerome Y. Plun. Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, Julio 1997.

[RRV98] António Ravara, Pedro Resende y Vasco T. Vasconcelos. Towards an Algebra of Dynamic Object Types. En *Workshop on Semantics of Objects as Processes*, número NS-98-5 en BRICS Notes Series, 1998.

[RS96] Michael D. Rice y Stephen B. Seidman. Using Z as a Substrate for an Architectural Style Description Language. Technical Report CS-96-120, Department of Computer Science, Colorado State University, Septiembre 1996.

[RT94] Jan Rutten y Daniele Turi. Initial Algebra and Final Coalgebra Semantics for Concurrency. En J.W. de Bakker, W.P. de Roever y G. Rozenberg, editores, *A Decade of Concurrency*, volumen 803 de *Lecture Notes in Computer Science*, págs. 530–582. Springer-Verlag, Berlín, 1994.

[Rus08] Bertrand Russell. Mathematical Logic as Based on the Theory of Types. *American Journal of Mathematics*, 30:222–262, 1908.

[Rus22] Bertrand Russell. Introduction. En *Ludwig Wittgenstein's Tractatus Logico-Philosophicus*. Keegan Paul, Trench & Tubner, Mayo 1922.

[RV00] António Ravara y Vasco T. Vasconcelos. Typing Non-uniform Concurrent Objects. En *Proceedings of Concur'00*, volumen 1877 de *Lecture Notes in Computer Science*, págs. 474–488. Springer Verlag, 2000.

[RW97] Anthony Rowstron y Alan Wood. Bonita: a set of tuple space primitives for distributed coordination. En *Proc. HICSS30, Sw Track*, págs. 379–388, Hawaii, 1997. IEEE Computer Society Press.

[San92] Davide Sangiorgi. From π-calculus to Higher-Order π-calculus — and Back. En *Proceedings of TAPSOFT'93*, volumen 668 de *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[San93] Davide Sangiorgi. *Expressing Mobility in Process Algebra*. Tesis Doctoral, University of Edinburgh, 1993.

[Sar97] Kamran Sartipi. A Survey on Software Architecture Domain. University of Waterloo, Ontario, Canadá, Febrero 1997.

[Sch99] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual Framework for Software Composition*. Tesis Doctoral, Institut für Informatik und angewandte Mathematik, Universität Bern, Octubre 1999.

[SDK+95a] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young y Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, Abril 1995.

[SDK+95b] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young y Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, Abril 1995.

[Sew00] Marc Sewell. Are You A Software Architect? *Enterprise Development*, 2(5):62–63, Mayo 2000. Fawcette Technical Publications.

[SG94] Mary Shaw y David Garlan. Characteristics of Higher-Level Languages for Software Architecture. Technical Report CMU-CS-94-210, SEI-94-TR-23, ESC-TR-94-023, Software Engineering Institute, Carnegie Mellon University, 1994.

[SG95] Mary Shaw y David Garlan. Formulations and Formalisms in Software Architecture. En *Computer Science Today*, volumen 1000 de *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[SG96] Mary Shaw y David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, New Jersey, 1996.

[SG01] Bridget Spitznagel y David Garlan. A Compositional Approach for Constructing Connectors. En Kazman et al. [KKVvV01], págs. 148–157.

[Sha80] Mary Shaw. The Impact of Abstraction Concerns on Modern Programming Languages. En *IEEE Proceedings*, Septiembre 1980. Reprinted: *IEEE Software*, Oct 1994.

[Sha90] Mary Shaw. Prospects for an Engineering Discipline of Software. *IEEE Software*, Noviembre 1990. Reprints: SEI Technical Report CMU/SEI-90-TR-020. *Software Management*, D. Reifer (ed.), IEEE Press 1993. *Enclyclopedia of Software Engineering*, Volume 2, pp. 930–940, John Wiley 1994.

[Sha94] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. En *Proceedings of Workshop on Studies of Software Design*, Enero 1994.

[Sha01] Mary Shaw. The Coming-of-Age of Software Architecture Research. En *Proceedings of the* 23rd *International Conference on Software Engineering (ICSE'01)*, Mayo 2001.

[Sie92] Carles Sierra. Estructuras de Control en Inteligencia Artificial. Arquitecturas Reflexivas y de Meta Nivel. En *Nuevas Tendencias en Inteligencia Artificial*, págs. 91–110. Universidad de Deusto, 1992.

[SK02] Marcelo Sihman y Shmuel Katz. A calculus of Superimpositions for Distributed Systems. En Ossher y Kiczales [OK02], págs. 28–40.

[SL97] Jean-Guy Schneider y Markus Lumpe. Synchronizing Concurrent Objects in the π-Calculus. En Roland Ducournau y Serge Garlatti, editores, *Proceedings of Languages et Modèles à Objets*, págs. 61–76, Roscoff, Octubre 1997. Hermes.

[Smi82] Brian Cantwell Smith. Reflection and Semantics in a Procedural Language. Technical Report MIT-LCS-TR-272, Massachusetts Institute of Technology, 1982.

[Smo94] Gert Smolka. A Foundation for Higher-Order Concurrent Constraint Programming. En Jean-Pierre Jouannaud, editor, *Proceedings of the First International Conference of Constraints in Computational Logics*, volumen 845 de *Lecture Notes in Computer Science*, págs. 50–72. Springer Verlag, 1994.

[SN99] Jean-Guy Schneider y Oscar M. Nierstrasz. Components, Scripts and Glue. En Leonor Barroca, Jon Hall y Patrick Hall, editores, *Software Architectures – Advances and Applications*, págs. 13–25. Springer Verlag, 1999.

[SR98] Bran Selic y Jim Rumbaugh. Using UML for Modelling Complex Real-Time Systems. ObjectTime Ltd., Marzo 1998.

[Ste94] Patrick Steyaert. *Open Design of Object-Oriented Languages: A Foundation For Specialisable Reflective Language Frameworks.* Tesis Doctoral, Vrije Universteit Brussel, 1994.

[Sul94] Kevin J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems.* Tesis Doctoral, University of Washington, 1994. UW-CSE-TR-94-08-01.

[SVK98] Sylvia Stuurman y Jan Van Katwijk. On-Line Change Mechanisms: The Software Architectural Level. En 6th *International Symposium on the Foundations of Software Engineering*, 1998.

[SW01] Davide Sangiorgi y David Walker. *The π-Calculus. A Theory of Mobile Processes.* Cambridge University Press, Junio 2001.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Enero 1998.

[Tal96] Carolyn Talcott. An Actor Rewriting Theory. En *Proceedings of First International Workshop on Rewriting Logic and its Applications*, volumen 4 de *Electronic Notes in Theoretical Computer Science*, Septiembre 1996.

[Tan91] Tomoyuki Tanaka. Actor Reflection without Meta-Objects. *OOPS Messenger*, 2(2), Abril 1991. ACM SIGPLAN.

[Tar35] Alfred Tarski. The Concept of Truth in Formalized Languages. En *Logic, Semantics, Metamathematics*, págs. 152–278. Clarendon Press, 1935. Traducido por J. H. Woodger.

[Tar44] Alfred Tarski. The Semantic Conception of Truth and the Foundations of Semantics. *Philosophical and Phenomenological Research*, 4, 1944.

[Tho90] Bent Thomsen. *Calculi for Higher Order Communicating Systems.* Tesis Doctoral, Imperial College, University of London, Septiembre 1990.

[TMA+96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy y Deborah L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, Junio 1996.

[Tok90] Mario Tokoro. Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment. Technical Report SCSL-TR-90-006, Sony Computer Science Laboratory, 1990.

[Tra96] Will Tracz. Minutes of the EDCS Lake Placid Kickoff Meeting. EDCS Community Pages, Julio 1996.

[TRG02] Robert Tolksdorf y Gregor Rojec-Goldmann. The SPACETUB Models and Framework. En Arbab y Talcott [AT02], págs. 348–363.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* Tesis Doctoral, University of Edinburgh, 1996.

[Ung88] David Ungar. Are Classes Obsolete? *ACM SIGPLAN Notices*, 23(12), 1988. OOPSLA'88.

[Ung00] Victoria Ungureanu. *A Mechanism for Supporting Communication Policies in Distributed Systems*. Tesis Doctoral, Faculty of Computer Science, Rutgers University, 2000.

[US87] David Ungar y Randall B. Smith. Self: The Power of Simplicity. En *ACM SIGPLAN Notices* [Mey87], págs. 227–242. OOPSLA'87.

[Vas94a] Vasco T. Vasconcelos. Predicative Polymorphism in the π-calculus. En *Parallel Architectures and Languages Europe: PARLE'94*, volumen 817 de *Lecture Notes in Computer Science*, págs. 425–437. Springer Verlag, 1994.

[Vas94b] Vasco Thudichum Vasconcelos. Typed Concurrent Objects. En Mario Tokoro y Remo Pareschi, editores, *ECOOP'94: European Conference on Object-Oriented Programming*, volumen 821 de *Lecture Notes in Computer Science*, págs. 100–117. Springer Verlag, 1994.

[Vas99] Vasco T. Vasconcelos. Processes, Functions, Datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[Vas01] Vasco Thudichum Vasconcelos. TyCO Gently. Technical Report DI-FCUL TR-01-4, Departamento de Informática, Universidade de Lisboa, Julio 2001.

[Vay00] Julien Vayssière. Security and Meta Programming in Java. En Cazzola et al. [CCL00a].

[vG01] Robert J. van Glabbeek. The Linear Time – Branching Time Spectrum I. The Semantics of Concrete, Sequential Processes. En Bergstra et al. [BPS01], págs. 3–99. Report CS-R9029, CWI, Amsterdam.

[VH93] Vasco Thudichum Vasconcelos y Kohei Honda. Principal Typing Schemes in a Polyadic Pi-Calculus. En *Proceedings of* 4th *International Conference on Concurrency Theory (CONCUR'93)*, volumen 715 de *Lecture Notes in Computer Science*, págs. 524–538, 1993.

[Vic95] Björn Victor. *The Mobility Workbench User's Guide. Polyadic version 3.122*. SICS, Estocolmo, Octubre 1995.

[Vic98] Björn Victor. *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes*. Tesis Doctoral, Uppsala University, Sweden, Junio 1998.

[VLS98] Vasco T. Vasconcelos, Luís Lopes y Fernando Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. *Electronic Notes in Theoretical Computer Science*, 16(3), 1998. Third International Workshop on High-Level Concurrent Languages.

[VM94] Björn Victor y Faron Moller. The Mobility Workbench — A Tool for the π-Calculus. En David Dill, editor, *CAV'94: Computer Aided Verification*, volumen 818 de *Lecture Notes in Computer Science*, págs. 428–440. Springer-Verlag, 1994.

[VM99] Antonio Vallecillo Moreno. *Un Modelo de Componentes para el Desarrollo de Aplicaciones Distribuidas*. Tesis Doctoral, Universidad de Málaga, Marzo 1999.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer y Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000.

[VPL99] James Vera, Louis Perrochon y David C. Luckham. Event-Based Execution Architectures for Dynamic Software Systems. En Donohoe [Don99], págs. 303–318.

[vR01] Guido van Rossum. *Python Reference Manual, release 1.5.2*. Stichting Mathematisch Centrum, Amsterdam, Marzo 2001.

[VRMP01] Federico Valentino, Andrés Ramos, Claudia A. Marcos y Jane Pryor. A Framework for the Development of Multi-Level Reflective Applications. En *Proceedings of Second Argentine Symposium on Software Engineering (ASSE 2001)*, Buenos Aires, Argentina, Septiembre 2001.

[VT93] Vasco Thudichum Vasconcelos y Mario Tokoro. A Typing System for a Calculus of Objects. En Nishio y Yonezawa [NY93], págs. 460–474.

[VT01] Nalini Venkatasubramanian y Carolyn Talcott. A Semantic Framework for Reasoning about Reflective Middleware. *IEEE Distributed Systems Online*, 2(6), 2001.

[Wal95] David Walker. Objects in the π-Calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.

[Wer98] Michel Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. *IEE Proceedings – Software*, 145(5):130–136, Octubre 1998.

[Wer99] Michel Alexandre Wermelinger. *Specification of Software Architecture Reconfiguration*. Tesis Doctoral, Universidade Nova de Lisboa, Septiembre 1999.

[WF88] Mitchell Wand y Daniel P. Friedman. The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower. *Lisp and Symbolic Computation*, 1, 1988.

[WF99] Michel Wermelinger y José Luiz Fiadeiro. Algebraic Software Architecture Reconfiguration. En *Software Engineering – Proceedings of ESEC/FSE'99*, volumen 1687 de *Lecture Notes in Computer Science*, págs. 393–409. Springer Verlag, 1999.

[Whi97] Scott A. Whitmire. *Object-Oriented Design Measurement*. John Wiley & Sons, Inc., 1997.

[Wil99] David Wile. AML: an Architecture Meta-Language. En *Proceedings of ASE'99*, págs. 183–190. Octubre, IEEE-CS Press, 1999.

[WK99] Jos Warmer y Anneke Kleppe. *The Object Constraint Language*. Object Technology Series. Addison-Wesley, 1999.

[WLF00] Michel Wermelinger, Antónia Lopes y José Luiz Fiadeiro. Superposing Connectors. En *Proceedings of* 10th *International Workshop on Software Specification and Design*, págs. 87–94. IEEE Computer Society Press, 2000.

[Woj00] Paweł T. Wojciechowski. The Nomadic Pict System. Documentation and User's Manual. Release 1.0a. Technical report, University of Cambridge, Computer Laboratory, Diciembre 2000.

[Wol97] Alexander L. Wolf. Succeedings of the Second International Software Architecture Workshop (ISAW-2). *ACM Software Engineering Notes*, 22(1):42–56, Enero 1997. ACM SIGSOFT'96, FSE-4.

[WS98] Ian S. Welch y Robert J. Stroud. Adaptation of Connectors in Software Architectures. En Wolfgang Weck, Jan Bosch y Clemens Szyperski, editores, *Proceedings of the Third International Workshop on Component-Oriented Programming (WCOP'98)*, volumen 10 de *TUCS General Publications*. Turku Centre for Computer Science (TUCS), Octubre 1998.

[WS00] Paweł T. Wojciechowski y Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. *IEEE Concurrency*, 8(2):42–52, 2000.

[WY88] Takuo Watanabe y Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. En *OOPSLA'88 Proceedings*, págs. 305–315, Septiembre 1988.

[YM01] Akinori Yonezawa y Satoshi Matsuoka, editores. *Reflection 2001: Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volumen 2192 de *Lecture Notes in Computer Science*, Kyoto, Japan, Septiembre 2001. AITO, ACM SIGSOFT, JSST, Springer Verlag.

[Yok92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. Technical Report SCLS-TR-92-014, Sony Computer Science Laboratory Inc., Tokyo, Japan, Junio 1992.

[YT87] Yasuhiko Yokote y Mario Tokoro. Experience and Evolution of ConcurrentSmalltalk. En Meyrowitz [Mey87], págs. 406–415.

[Zav98] Gianluigi Zavattaro. On the Incomparability of Gamma and Linda. Technical Report SEN-R9827, Centrum voor Wiskunde en Informatica (CWI), Octubre 1998.

[ZP97] Xiaogang Zhang y John Potter. Class-based Models in the π-Calculus. En Christine Mingins, Roger Duke y Bertrand Meyer, editores, *Proceedings of the* 25th *International Conference in Technology of Object-Oriented Languages and Systems (TOOLS Pacific '97)*, págs. 219–231, Melbourne, Australia, Noviembre 1997.

[ZP98] Xiaogang Zhang y John Potter. A Compositional Approach to Concurrent Objects. En Jian Chen, M. Li, Christine Mingis y Bertrand Meyer, editores, 27th *Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia '98)*, Beijing, China, Septiembre 1998. IEEE Computer Society Press.

[ZW95] Amy Moormann Zaremski y Jeannette Wing. Specification Matching of Sofware Components. En *Proceedings of the ACM SIGSOFT'95 Foundations of Software Engineering Symposium*. ACM Press, 1995.

BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Parte VI

Apéndices

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

Apéndice A

Gramática de PiLar

En el Capítulo 6 se describe en detalle la estructura de $\mathcal{P}i\mathcal{L}ar$, sus construcciones y el objetivo de las mismas; sin embargo, no se proporciona una definición formal de su sintaxis, ya que su inclusión en ese contexto se consideró poco ilustrativa.

Éste Apéndice pretende subsanar esa carencia; no sólo por afán de completitud, sino también con el fin de eliminar toda posible ambigüedad en cuanto al uso del lenguaje. Por tanto, se incluye aquí una gramática formal para $\mathcal{P}i\mathcal{L}ar$, expresada en la Notación de Backus-Naur Extendida. La gramática es del tipo *LALR-1*, de modo que resulta muy sencillo implementar un verificador automático. Carece además de conflictos de cualquier tipo, y no necesita establecer ninguna clase de reglas de precedencia.

PiLarLeng	::=	{ (TipoDef | Niveles | CompDef) }
TipoDef	::=	"**deftype**" *identif* "=" "[" LTipo "]"
LTipo	::=	{ *identif* "**,**" } *identif*
Niveles	::=	("**base−level**" | "**meta−level**" "[" *entero* "]") "**:**" "**(**" LComp "**)**"
LComp	::=	{ *identif* "**,**" } *identif*
CompDef	::=	"**component**" Wrap IdC "**(**" Faces Configs Reificar Reglas "**)**"
Wrap	::=	["**wrap**"]
IdC	::=	*identif* ["<" LParam ">"]
LParam	::=	LIden "**:**" VTipo
	|	LIden "**:**" "**Port**"
	|	LParam "**,**" LIden "**:**" VTipo
	|	LParam "**,**" LIden "**:**" "**Port**"
LIden	::=	{ *identif* "**,**" } *identif*
Faces	::=	{ Interfaz }
Interfaz	::=	["**interface**" | "**metaface**"] "**(**" Puertos "**)**"
IdP	::=	*identif* ["**:**" VTipo | "[" LTam "]" | "[" LTam "]" "**:**" VTipo]
Puertos	::=	{ "**port**" IdP "|" } "**port**" IdP
VTipo	::=	*identif*
	|	"**Int**"
	|	"**Bool**"
	|	"**String**"

Tam	::=	*entero*
	|	*identif*
LTam	::=	{ Tam “**,**” } Tam
Configs	::=	{ “**configuration**” “(” Topologia “)” }
Topologia	::=	Instancias
	|	Conectar
	|	Instancias “|” Conectar
Instancias	::=	LInst “**:**” Tipo
	|	MetaId
	|	Instancias “|” LInst “**:**” Tipo
	|	Instancias “|” MetaId
	|	Instancias “|” Bucle “(” Instancias “)”
	|	Instancias “|” Alter “(” Instancias “)”
	|	Instancias “|” Alter “(” Instancias “)” “(” Instancias “)”
MetaId	::=	*identif*
LInst	::=	{ IdVec “**,**” } IdVec
IdVec	::=	*identif* [“[” LDim “]”]
Tipo	::=	*identif*
	|	Tipo “<” LDim “>”
	|	Tipo “(” LIdRol “)”
IdN	::=	*entero*
	|	“**now**”
	|	QualId
	|	“**card**” “(” Conjunto “)”
LDim	::=	{ ExpN “**,**” } ExpN
QualId	::=	{ IdVec “**.**” } IdVec
IdRol	::=	“**now**”
	|	“**Void**”
	|	QualId
LIdRol	::=	{ IdRol “**,**” } IdRol
Alter	::=	“**if**” “(” Condicion “)”
Bucle	::=	(“**for**” | “**loop**”) Vcb “(” Conjunto “)”
Vcb	::=	[*identif*]
Condicion	::=	{ BoolExpr “**or**” | Condicion “**and**” | “**not**” } BoolExpr
BoolExpr	::=	“(” Condicion “)”
	|	ExpN “=” ExpN
	|	ExpN “>” ExpN
	|	ExpN “<” ExpN
	|	ExpN “>” “=” ExpN
	|	ExpN “<” “=” ExpN
	|	ExpN “<” “>” ExpN
	|	ExpN “**in**” Conjunto
	|	“**true**”
	|	“**false**”

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

Conjunto	::=	*identif*
	|	ExpN "**..**" ExpN
Conectar	::=	"**bind**" "(" Conexion ")"
Conexion	::=	IdCon "**:**" TConex
	|	Enlace
	|	Bucle "(" Conexion ")"
	|	Alter "(" Conexion ")"
	|	Alter "(" Conexion ")" "(" Conexion ")"
	|	Conexion "|" Enlace
	|	Conexion "|" IdCon "**:**" TConex
	|	Conexion "|" Bucle "(" Conexion ")"
	|	Conexion "|" Alter "(" Conexion ")"
	|	Conexion "|" Alter "(" Conexion ")" "(" Conexion ")"
IdCon	::=	*identif* ["[" LDim "]"]
TConex	::=	Enlace
	|	CnxTipad
CnxTipad	::=	*identif* ("(" LIdRol ")" | "<" LDim ">" "(" LIdRol ")")
Enlace	::=	IdRol "=" IdRol
	|	"(" IdRol "=" IdRol ")"
	|	"**link**" "(" IdRol "**,**" IdRol ")"
Reificar	::=	{ "**reify**" (IdR "**as**" Tipo | *identif* Tipo | Tipo) }
IdR	::=	[*identif*]
Reglas	::=	["**constraint**" "(" LengDin ")"]
LengDin	::=	Proceso
	|	ProcDef
	|	LengDin ChanDef
	|	LengDin VarDef
	|	LengDin ProcDef
VarDef	::=	"**var**" LVar ("**;**" | "**:**" VTipo "**;**")
LVar	::=	{ *identif* "**,**" } *identif*
ChanDef	::=	"**chan**" LChan ("**;**" | "**:**" VTipo "**;**")
LChan	::=	{ *identif* "**,**" } *identif*
ProcDef	::=	*identif* ("**def**=" Proceso | "(" LPar ")" "**def**=" Proceso)
LPar	::=	{ *identif* "**,**" } *identif*
Proceso	::=	SubProc
	|	Proceso "**;**" SubProc
	|	"(" Proceso "**;**" SubProc ")"
	|	Proceso "|" SubProc
	|	"(" Proceso "|" SubProc ")"
	|	Proceso "|+" SubProc
	|	"(" Proceso "|+" SubProc ")"
SubProc	::=	"(" SubProc ")"
	|	Accion
	|	Invocar

	|	"**rep**" "(" Proceso ")"
	|	"**when**" Recepcion "(" Proceso ")"
	|	LDAlter
	|	LDBucle
LDAlter	::=	"**if**" "(" LDCond ")" "(" Proceso ")" ["(" Proceso ")"]
LDBucle	::=	("**for**" | "**loop**") Vcb "(" LDConj ")" "(" Proceso ")"
LDCond	::=	{ LDBool "**or**" | LDCond "**and**" | "**not**" } LDBool
LDBool	::=	"(" LDCond ")"
	|	DExpN "=" DExpN
	|	DExpN ">" DExpN
	|	DExpN "<" DExpN
	|	DExpN ">" "=" DExpN
	|	DExpN "<" "=" DExpN
	|	DExpN "<" ">" DExpN
	|	DExpN "**in**" LDConj
	|	"**nullr**" "(" *identif* ")"
	|	"**true**"
	|	"**false**"
LDConj	::=	"**avatarSet**"
	|	"**avatarSet**" "(" Datum ")"
	|	"**portSet**"
	|	"**portSet**" "(" Datum ")"
	|	"**bound**" "(" Datum ")"
	|	Datum
	|	DExpN "**..**" DExpN
Canal	::=	["**self**" "**.**"] Datum
Datum	::=	{ IdDatum "**.**" } IdDatum
IdDatum	::=	*identif*
	|	*identif* "[" LDatN "]"
	|	"**now**"
	|	"**now**" "[" LDatN "]"
	|	"**avatar**"
LCanal	::=	{ Canal "**,**" } Canal
LDat	::=	{ Datum "**,**" } Datum
DatN	::=	*entero*
	|	Datum
	|	"**card**" "(" LDConj ")"
LDatN	::=	{ DExpN "**,**" } DExpN
Invocar	::=	IdProc ["<" LCanal ">"]
IdProc	::=	*identif*
Accion	::=	Emision
	|	Recepcion
	|	Silencio
	|	Crear

	|	Destruir
	|	Extrudir
	|	Ocultar
	|	LDReify
	|	HayReif
	|	Insertar
	|	Capturar
	|	AbrirEnt
	|	AbrirSal
Emision	::=	Canal "!" (Tupla | "(" ")")
Recepcion	::=	Canal "**?**" (NuTupla | "(" ")")
Tupla	::=	Datum
	|	"(" LDat ")"
NuTupla	::=	*identif*
	|	"(" LNNom ")"
LNNom	::=	{ *identif* "**,**" } *identif*
Silencio	::=	"**tau**" ["(" LNNom ")"]
Crear	::=	"**new**" "(" Datum "**:**" TCreado ")"
TCreado	::=	"**Port**"
	|	LinkComp
LinkComp	::=	"(" VDatum "=" VDatum ")"
	|	"**link**" "(" VDatum "=" VDatum ")"
	|	IdComp
VDatum	::=	"**Void**"
	|	Datum
LVDat	::=	{ VDatum "**,**" } VDatum
IdComp	::=	*identif* ["<" LDatN ">" | "(" LVDat ")" | "<" LDatN ">" "(" LVDat ")"]
Destruir	::=	"**del**" Datum
Extrudir	::=	"**alias**" Datum ["**as**" *identif*]
Ocultar	::=	"**hide**" Datum
LDReify	::=	"**reify**" *identif* "(" Datum "**:**" LinkComp ")"
HayReif	::=	"**findr**" *identif* "(" Datum "**:**" IdComp ")"
Insertar	::=	"**shift**" Canal "(" LDat ")"
Capturar	::=	"**catch**" Canal "(" LNNom ")"
AbrirEnt	::=	"**letin**" Canal
AbrirSal	::=	"**letout**" Canal
ExpN	::=	{ SubExp ("+" | "−" | "∗" | "/" | "%") } SubExp
SubExp	::=	IdN
	|	"(" ExpN ")"
DExpN	::=	{ DSubExp ("+" | "−" | "∗" | "/" | "%") } DSubExp
DSubExp	::=	DatN
	|	"(" DExpN ")"

BIBLIOTECA VIRTUAL MIGUEL DE CERVANTES

La regla inicial de la gramática se indica, por supuesto, en primer lugar (PilarLeng), y a partir de ella se despliega el resto. Por su parte, todos los símbolos no terminales se emplean aquí en su forma literal, por lo que su definición léxica no reviste mayor dificultad. Solamente dos de ellos requieren un comentario especial: los símbolos *identif* y *entero*.

El primero se corresponde con un identificador cualquiera. Se define léxicamente de la manera convencional, es decir, como una secuencia alfanumérica de longitud finita, que comienza siempre por una letra. Como es habitual, se considera al guión bajo (_) como una letra más. El segundo se corresponde, por su parte, con un número entero decimal, con o sin signo. En esta gramática es el único tipo numérico admitido, y por tanto los reales quedan excluidos.

A primera vista, puede parecer que la gramática definitiva es excesivamente larga y compleja. En realidad, no lo es en absoluto. Su estructura global es muy sencilla, y tiene un núcleo concentrado en muy pocas reglas. A partir de la inicial (PilarLeng) se define un lenguaje que consta casi en exclusiva de definiciones de componente (CompDef), en las que se concentra la estructura del denominado *Lenguaje Estructural*. Los componentes se desarrollan como un conjunto de interfaces (Faces) y reificaciones, de sintaxis más bien trivial, junto a cero o más configuraciones (Configs). La estructura de estas últimas es más compleja, y se divide a su vez en declaraciones de instancias (Instancias) y conexiones (Conexion). Éstas son apenas simples secuencias; su aparente complejidad deriva tan sólo de su combinación con las construcciones iterativas y alternativas, que a su vez están bastante limitadas.

La sintaxis de las restricciones asociadas a estos componentes (Reglas) constituye el denominado *Lenguaje Dinámico*, que se desarrolla, en líneas generales, como una serie de definiciones de proceso (ProcDef). Su estructura en este punto ha sido definida de manera análoga a la de un álgebra de procesos, y utiliza apenas siete constructores diferentes. En realidad, el auténtico núcleo de este sublenguaje es la docena de acciones atómicas definidas en la regla Accion. Sólo la forma de las tres primeras reviste cierta complejidad; la del resto es notablemente sencilla.

El resto de la gramática se dedica tan sólo a refinar aspectos menores de la sintaxis, comunes a muchos lenguajes y no relacionados de manera directa con la Arquitectura de Software. Se definen aspectos tales como los distintos modos de agrupar parámetros, indicar tipos, manipular y cualificar nombres, o definir conjuntos. Sin embargo, la mayor extensión de esta parte, casi una tercera parte del total, se dedica solamente al tratamiento sintáctico de los tipos de datos predefinidos, es decir, a la formulación de las expresiones aritméticas y lógicas. Si la gramática no hubiera incluido explícitamente estos detalles, su longitud sería mucho menor; en cualquier caso, se utiliza una notación estándar, por lo que la complejidad derivada es mínima.

En la gramática se han incluido algunas reglas semánticas, es decir, que establecen distinciones entre formas idénticas. En realidad, ninguna de ellas es estrictamente necesaria: sólo se proporcionan con el fin de aumentar la claridad en algunos puntos.

La sintaxis actual de $\mathcal{P}i\mathcal{L}ar$, descrita en la gramática aquí incluida, difiere en ciertos detalles con respecto a presentaciones anteriores, en especial las más antiguas [CFBSB00, CFBSB01]. No obstante, estas diferencias son en su mayoría meramente sintácticas; la coherencia conceptual entre las distintas versiones se ha mantenido en todo momento. A lo largo de la evolución del lenguaje, el vocabulario básico de *palabras clave* también ha cambiado ligeramente. No obstante, se sigue admitiendo por compatibilidad el uso de las formas antiguas allí donde no se produzca una inconsistencia. Para no complicar más la gramática, esta compatibilidad se introduce como

Palabra Clave Usada	*Equivalentes Léxicos Admitidos*
\alias	**\extrude**
\avatar	**\alpha**
\component	**\comp**
\configuration	**\config**, **\conf**
\del	**\delete**, **\destroy**, **\delta**
\findr	**\phi**
\new	**\create**, **\nu**
\port	**\role**
\reify	**\rho**
\rep	**\bang**
\self	**\gamma**

Figura A.1: Tabla de Equivalencias Léxicas en $\mathcal{P}i\mathcal{L}ar$

una serie de *sinónimos*, que se introducen en la forma de equivalencias léxicas, resumidas en la Tabla A.1. Esto significa que entre una palabra clave y cualquiera de sus sinónimos no hay ninguna diferencia sintáctica; a nivel gramatical, todas ellas son la misma palabra, y pueden aparecer por tanto en las mismas posiciones.

La mayoría de estos sinónimos no se utilizan desde hace tiempo, y se incluyen meramente por completud. Otros, como las formas abreviadas (**\config**, **\comp**) serán mucho más habituales. Alguno de ellos, incluso, es empleado en los ejemplos del Capítulo 8 de esta tesis.

La presentación de la gramática en este Apéndice ha sido generada para LaTeX 2_ε de manera automática, a partir de la versión de la misma utilizada en un verificador de sintaxis $\mathcal{P}i\mathcal{L}ar$. En el proceso de traducción se utilizó básicamente la herramienta `y2l` de Kris Van Hees, complementada con un procesado previo en `awk` y un ajuste posterior basado en `sed`.

Apéndice B

Sintaxis del Cálculo-π

El *cálculo-π* [MPW92a, MPW92b] es el máximo representante de una familia de cálculos, diseñados para la especificación formal de concurrencia, interacción y movilidad [Mil99], cuya noción fundamental es la de *nombre* o *referencia*. Los miembros de esta familia reciben por este motivo la denominación colectiva de *cálculos nominales*, debida a Andrew Gordon [Gor00], y se constituyen como una evolución, tanto histórica como conceptual, de las clásicas *álgebras de procesos* [BPS01], conjunto en el que ellos mismos están también incluidos.

Como se ha dicho, estos cálculos se conciben en torno al concepto de *nombre*. Bajo esta denominación común se agrupan todos aquellos términos que designan a una entidad del sistema, o proporcionan una referencia hacia la misma; así, un nombre puede ser una dirección, un identificador, una posición, un puntero o un canal. Desde este punto de vista, la interacción no puede darse en ausencia del nombrado: una entidad sólo puede comunicarse con otra si dispone de algún modo para hacer referencia a ésta. Por tanto, la conectividad de una entidad viene dada por el conjunto de nombres que conoce: el nombrado determina la *topología* de interacción. Del mismo modo, la capacidad para transmitir estos nombres permite alterar esta topología, por lo que determina la *movilidad*. Robin Milner fue el primero en intuir la importancia fundamental de esta idea en el campo de la Concurrencia, lo que le llevó a la definición original del cálculo-π como el punto de partida de una hipotética Teoría General del Nombrado [Mil93a]. Y en efecto, se ha constituido como un foco del que han surgido gran cantidad de propuestas, algunas de las cuales no se ciñen estrictamente a los límites del campo, como en el caso de la *lógica nominal* de Andrew Pitts [Pit01], que hace especial énfasis en el principio de sustitución.

En este Apéndice se habla primordialmente del cálculo-π [Mil99, SW01], que no es sólo el primero y más importante de estos cálculos, sino también el más sencillo y desarrollado. La próxima sección se dedica en su totalidad a describir brevemente su sintaxis y semántica informal; no obstante, aparte del original, existen otros cálculos nominales de especial relevancia e interés, algunos de los cuales se enumeran en los párrafos siguientes.

El más destacado de todos ellos es, sin duda, el *cálculo de ambientes* de Luca Cardelli y Andrew Gordon [CG98, CG00, Gor00]. En muchos aspectos, su enfoque es dual al del cálculo-π, en el que se sustituyen interacción y movilidad por *localización* y seguridad; ha sido desarrollado en gran detalle, y dispone incluso de su propia teoría de tipos –simplificada mediante una teoría de grupos [CGG00]– y su propia lógica modal –la lógica de ambientes–, descrita en la tradición de

las lógicas de Hennesy-Milner. Por otra parte, desde una perspectiva aún más teórica, Gérard Boudol [Bou98] definió el *cálculo azul* o *cálculo-π^**, base del sistema Blue, que lleva hasta el límite el isomorfismo entre el cálculo-π y el cálculo-λ en estilo-CPS (§6.6.1). Con el objetivo análogo de generalizar los postulados iniciales, el propio Milner planteó los denominados *cálculos de acción* [Mil96], que han adoptado su forma más evolucionada en el *cálculo-π_F* de Philippa Gardner y Lucian Wischik [GW00]; en éste se utiliza otra estrategia de ligadura de nombres, inspirada a su vez en el *cálculo de fusión* de Björn Victor y Joachim Parrow [Vic98].

La definición del *cálculo-join* o *cálculo de reunión* por Cédric Fournet, Georges Gonthier y otros [FG95, FGL+96, FLMR00] se ubica en una línea completamente distinta, ya que se obtiene a partir de una versión "reflexiva" de la CHAM, derivando en un cálculo nominal de potencia equivalente al cálculo-π. Se caracteriza por disponer de una sintaxis aún más compacta, y es la base del lenguaje Join, así como de sus variantes distribuida y orientada al objeto. Entre sus aspectos más notables sobresale el desarrollo de una completa teoría de tipos.

Otros formalismos significativos nacidos de la aplicación del cálculo-π en distintos entornos son el *cálculo-spi* de Martín Abadi y Andrew Gordon [AG99], diseñado para la descripción de protocolos de transmisión de claves criptográficas. O el *cálculo-γ* de Gert Smolka [Smo94], que incrementa la expresividad del cálculo-π original mediante la simple adición de variables *lógicas*, constituyéndose en el núcleo semántico del sistema Oz (Mozart), probablemente el más importante de los lenguajes de programación basados en restricciones concurrentes. También cabe destacar el *cálculo-$\pi\mathcal{L}$* de Markus Lumpe y Jean-Guy Schneider [Lum99, Sch99], planteado como una aproximación al cálculo-π a partir de la simple combinación de registros (*formularios*), y fundamento del lenguaje Piccola [SN99, LAN00, ALSN00, AKN00, AN00].

Finalmente, entre los cálculos tipados, aún más cercanos al cálculo-π original, merecen especial mención el *cálculo-π polimórfico* de David Turner [Tur96], base conceptual del lenguaje de programación concurrente Pict, probablemente el más conocido y potente en este contexto, desarrollado por Benjamin C. Pierce y el propio Turner [PT00]; el *cálculo-π nomádico*[1] de Peter Sewell y Paweł Wojciechowski [WS00, Woj00], una variante del anterior con un mayor énfasis en los aspectos de localidad y movilidad; y el sistema TyCO (*Typed Concurrent Objects*) de Vasco T. Vasconcelos [VT93, Vas94b, RV00], uno de los primeros entornos de orientación al objeto fundamentados en un cálculo nominal, junto a [Wal95] o [BS95], para el que se dispone también de una implementación [Vas01] con una difusión bastante amplia.

Todos ellos manifiestan una potencia expresiva similar; aunque la semántica de $\mathcal{P}i\mathcal{L}ar$ (§7) se describe en términos del cálculo-π polimórfico (§C), ésta podría haberse proporcionado igualmente, con mayor o menor dificultad, en cualquiera de los sistemas formales citados. Especialmente interesante resulta el planteamiento de la concepción arquitectónica desde la perspectiva del cálculo de ambientes, ya que su énfasis en el *encapsulamiento* se corresponde a la perfección con la noción intuitiva de componente. De hecho, los últimos trabajos de Milner [Mil01] se basan en la combinación de los postulados del cálculo-π y el cálculo de ambientes para la definición de *sistemas bigráficos reactivos*, los cuales no sólo muestran una analogía totalmente evidente con los términos de una especificación arquitectónica, sino que incluso manifiestan, de manera reconocida, una influencia directa de la investigación previa [HM99] en esta materia.

[1] El neologismo *nomádico* hace referencia a la naturaleza *nómada* de los procesos del cálculo. Por lo demás, el nombre es tan sólo un juego de palabras con respecto al cálculo-π (monádico) original.

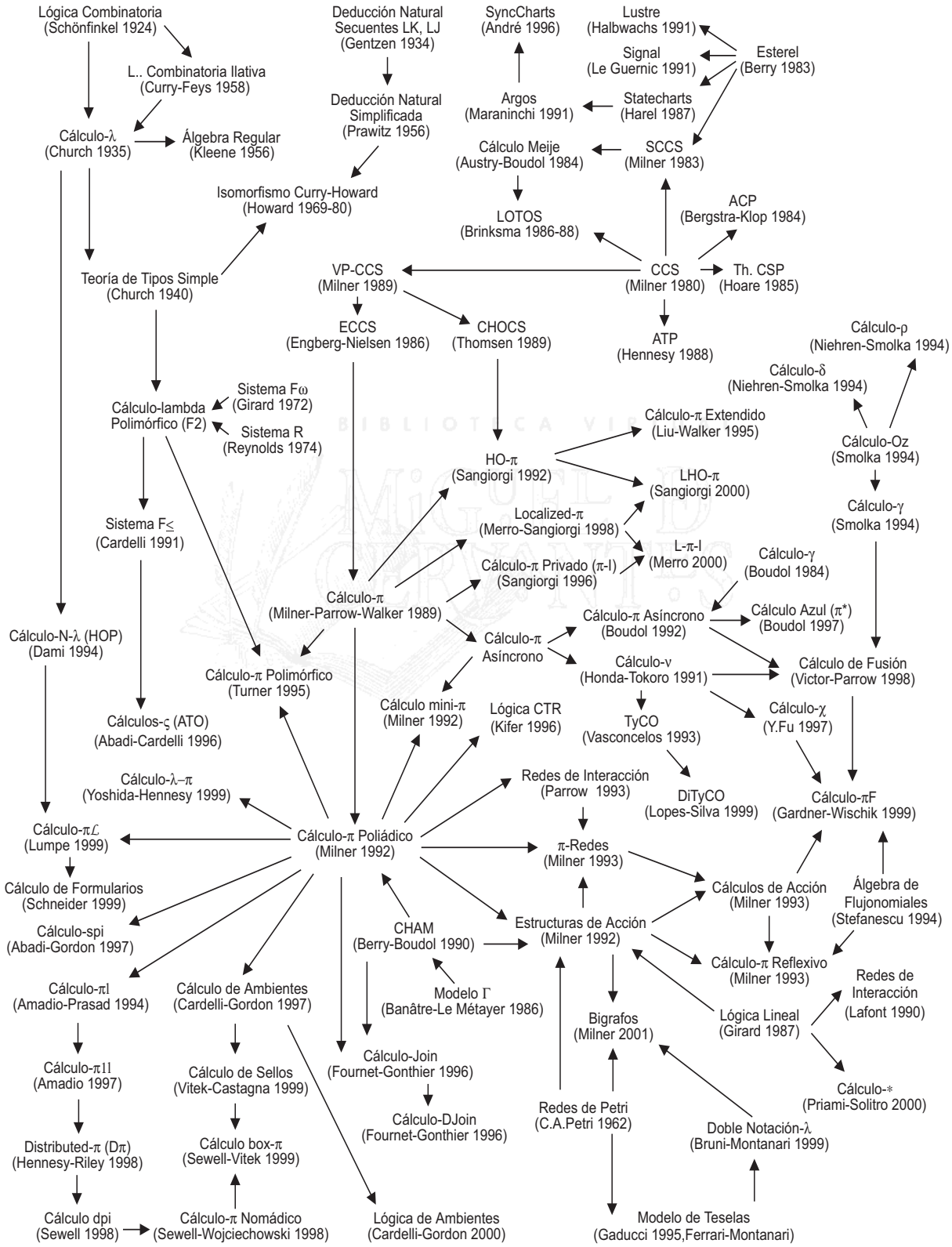

Figura B.1: Evolución del Cálculo-π y los Cálculos Nominales

La Figura B.1 muestra la evolución de distintos formalismos hacia y desde el cálculo-π, incluyendo a todos los cálculos nominales de importancia, y ligándolo con las teorías de tipos, el cálculo-λ, e incluso la lógica clásica e intuicionista. Todos los sistemas formales mencionados en la misma guardan algún tipo de relación con el cálculo-π o cualquiera de sus variantes. De hecho, el vínculo es con frecuencia incluso más directo de lo que la Figura puede dar a entender, algo que es especialmente cierto en las versiones tipadas. Es el caso, por ejemplo, de su –compleja– relación con la lógica lineal de Girard, los cálculos-ς de Abadi y Cardelli, el álgebra regular de Kleene, el modelo de teselas y la doble notación-λ de Gaducci y Montanari, las distintas formas del cálculo-λ, aquí apenas esbozadas [Bar92], o el isomorfismo de Curry-Howard.

Por razones de espacio y de relevancia, no se incluyen en la Figura otros formalismos relacionados con éstos en un grado similar, tales como sus variantes probabilísticas, estocásticas y temporizadas; las distintas lógicas modales –incluyendo la lógica dinámica– y temporales; o los múltiples derivados de algunos de los sistemas mencionados, como las redes de Petri, el cálculo-λ o la lógica lineal. Del mismo modo se han omitido menciones a otras aproximaciones algebraicas, gráficas, categóricas y lógicas, y en particular a otros modelos de concurrencia, como los *pomsets* y estructuras de eventos, los espacios de Chu, la lógica de reescritura o los autómatas de orden superior, cuya relación con los aquí presentados es aún más indirecta.

Aunque este Apéndice se centra de manera fundamental en el cálculo-π, ya que éste es el formalismo utilizado en esta tesis, la exposición de la familia de cálculos detallada en los párrafos anteriores obedece a dos objetivos principales: en primer lugar, situar a dicho cálculo en su contexto más amplio; y en segundo lugar, resaltar el hecho de que sus construcciones sintácticas se utilizan sin apenas variaciones en la práctica totalidad de los cálculos nominales. Las extensiones descritas en el Apéndice C se han realizado, igualmente, de modo que no entran en conflicto con ninguno de ellos. Por tanto, las formulaciones y conclusiones extraídas a lo largo de esta tesis podrán ser expresadas, en mayor o menor medida, en términos de otro cualquiera de los cálculos mencionados.

B.1 El Cálculo-π

En este Apéndice *no* se pretende proporcionar una introducción completa al cálculo-π, puesto que para éste existe ya una abundante bibliografía de referencia. No sólo se dispone de varios resúmenes y exposiciones más o menos extensas [Mil93b, Gor00, Par01], sino incluso de dos recientes libros [Mil99, SW01], que cubren en detalle, respectivamente, los aspectos semánticos asociados y las múltiples variedades y tipificaciones del cálculo –poliádico– básico.

No se tratarán aquí, pues, los principales aspectos formales del cálculo, tales como su semántica operacional (estructurada), la relación de reducción, la congruencia estructural, o las distintas nociones de bisimulación y equivalencia observacional [Mil89, Mil99, vG01]. El único objetivo de esta sección es aclarar la sintaxis concreta, de entre las múltiples presentaciones existentes, que se utiliza en esta tesis, con el fin de hacerla autocontenida, siquiera parcialmente.

Hay decenas de formulaciones más o menos equivalentes del cálculo; por ejemplo, en algunas de ellas se distingue sintácticamente entre variables y nombres, se evitan los paréntesis en las acciones monádicas, se utilizan distintas notaciones para los operadores de abstracción, restricción y replicación, o se eliminan algunos de los constructores básicos, como los de selección y replica-

ción. Las versiones más conocidas son la original [MPW92a, MPW92b], la de Milner [Mil93b], la de Pict [Pie98], o la del Mobility Workbench [Vic95].

Por otra parte, existen multitud de variantes del cálculo-π; en la parte introductoria del Apéndice C se comentan detalladamente algunas de ellas. En general, cuando se habla del cálculo, se hace referencia a alguna versión del cálculo-π poliádico –típicamente *sorted*– utilizando como referencia el artículo original de Milner [Mil93b] o incluso su libro más reciente [Mil99]. La sintaxis que aquí se incluye considera esta misma versión, junto a algunas de sus adiciones. La única diferencia significativa es el hecho de que se trata, finalmente, de un cálculo tipado; pero eso no afecta a los aspectos aquí desarrollados.

La formulación inicial del cálculo-π poliádico sólo considera un tipo de entidad activa, el *proceso*, que manipula un único tipo de elemento o dato, el *nombre* o *canal*, que normalmente se agrupa bajo la forma de *tuplas* o *vectores*, designadas mediante la notación $\tilde{x}$. Los procesos pueden estar parametrizados, pero en principio no se proporciona ningún tipo de construcción intermedia. Posteriores refinamientos de la sintaxis introducen la categoría de *agente* o término parcial, concebido como una generalización de la constante de proceso. No obstante, el concepto sólo tiene importancia cuando se consideran explícitamente los términos de abstracción y concreción, por lo que en general resulta innecesario tenerlo en cuenta. En general, se asimila la definición de un agente abstracto a la de un proceso parametrizado, no incluido en esta gramática, utilizándose ambas construcciones de manera totalmente equivalente.

En este Apéndice, como en toda esta tesis, se asume que no se dispone de ningún tipo de operador ni definición de orden superior; es decir, se está tratando con una variante de primer orden, no extendida. Al ser además poliádica, se asume que todos los términos y operadores pueden aplicarse sobre tuplas de nombres, incluyéndose el caso monádico de manera trivial. En esta sintaxis, toda operación de entrada –abstracción, recepción, restricción– exige el uso de paréntesis, mientras que en toda operación de salida –concreción, envío, aplicación de constante– se encierran los parámetros entre ángulos ($\langle, \rangle$). Por otra parte, no es preciso considerar en este punto si el cálculo es o no tipado; es decir, los términos no requieren el uso de anotaciones explícitas de tipo. Como se podrá verse en el Apéndice C, esta información sólo será necesaria en las declaraciones de proceso, al igual que ocurre en el cálculo-π polimórfico de Turner.

La semántica asociada a esta sintaxis es la que se describe de manera informal a continuación. Como puede verse, un proceso se despliega como un conjunto de secuencias de acciones, que se combinan entre sí mediante constructores de varios tipos. La gramática de la Tabla B.1 reune todos los términos en una única regla, la de definición de proceso; otras presentaciones separan explícitamente las acciones de los constructores, forzando en algunos casos la definición de procesos intermedios.

Como en cualquier álgebra de procesos tipo-CCS, las acciones vienen dadas en la forma de *prefijos*, que determinan los actos de comunicación e interacción. De este modo, las acciones *preceden* a todo el subproceso prefijado, actuando de este modo también como *guardas* [Dij75] del mismo. Las tres acciones básicas son la *entrada* $[x(y)]$, que expresa la recepción de un nombre o dato (y) a través de un canal (x), y realiza además una ligadura del primero cuyo ámbito se extiende hasta el final del proceso; la *salida* $[\overline{x}\langle y\rangle]$, que expresa la emisión de un nombre (y) a través de un canal (x), siendo ambos previamente conocidos. Por supuesto, el

$x, y, z \ldots$			Nombre (canal, puerto)
$\tilde{x}, \tilde{y}, \tilde{z} \ldots$			Vector de nombres (tupla)
$D, E, F \ldots$			Constante (Agente)
$P, Q, R \ldots$			Proceso
D	::=	P	Proceso (no parametrizado)
	|	$(\tilde{x})P$	Abstracción
	|	$\langle\tilde{x}\rangle P$	Concreción
P	::=	0	Proceso Nulo o Inacción
	|	$x(\tilde{y}).P$	Prefijo de Entrada o Acción de Recepción
	|	$\overline{x}\langle\tilde{y}\rangle.P$	Prefijo de Salida o Acción de Envío
	|	$\tau.P$	Prefijo Silencioso o Acción Interna
	|	$(\nu\,\tilde{x})P$	Restricción
	|	$P + Q$	Selección (Suma) No Determinista
	|	$P \mid Q$	Composición (Producto) Paralela
	|	$D\langle\tilde{x}\rangle$	Aplicación de Constante
	|	$!P$	Replicación (Rep. Guardada: $!\,\pi.P$)
	|	$[x = y]P$	Comparación o *Matching*
	|	$P\,;Q$	Composición Secuencial *(op. derivado)*
	|	**if** b **then** P **else** Q	E. Alternativa *(op. derivado)*

Tabla B.1: Gramática (Sintaxis) del Cálculo-π Poliádico

canal utilizado en los dos casos es también un nombre[2]. Finalmente, el *silencio* $[\tau]$ expresa la ocurrencia de una acción interna del proceso, que no produce efectos externos, y por tanto no puede ser observada. Se utiliza habitualmente para indicar que el subproceso prefijado no se activa de manera instantánea, sino que ha de ocurrir "algo" antes de su ejecución.

El término **0** hace referencia al proceso nulo, que indica *inacción*, esto es, la finalización del subproceso actual. Como es habitual, se admite omitirlo al final de una expresión, con el fin de simplificar la especificación. Por tanto, un subproceso de la forma $\overline{x}\langle y\rangle$ no describe, como podría parecer, una operación de salida asíncrona, sino que es una abreviatura de la fórmula $\overline{x}\langle y\rangle.\mathbf{0}$, que define una emisión síncrona, seguida de una inacción.

La operación de *restricción* $[(\nu\,x)P\,]$ es especialmente característica, e indica la creación de un nombre (x), cuyo ámbito se extiende hasta el final del subproceso actual. Produce también, por tanto, una ligadura de nombres, y de hecho en algunos cálculos nominales –como el cálculo de fusión o cualquiera de sus derivados– es la única forma de ligadura existente.

[2] A menudo se usa la terminología de CCS, que denomina *nombres* a los canales de entrada, y *conombres* a los de salida. También es usual el empleo de la notación de CSP, donde la entrada es $x?y$ y la salida es $x!y$.

Por su parte, los constructores son los operadores algebraicos que hacen posible la combinación de subprocesos en un proceso mayor. Los dos clásicos son la *composición paralela* $[P \mid Q]$ y la *selección no determinista* $[P + Q]$, que se corresponden, respectivamente, con el producto y la suma del álgebra. El primero simplemente yuxtapone ambos procesos y los activa de manera concurrente, haciendo posible la interacción entre ellos. El segundo, por su parte, permite optar por uno u otro de manera no determinista, descartando el subproceso no elegido. Se trata en este caso de una selección tipo-Ccs, lo que significa que aunque es estrictamente indeterminista cuando los distintos subprocesos están en igualdad de condiciones (selección interna), se comporta en cambio como una *suma guardada* si todos los prefijos involucrados definen acciones de entrada, en cuyo caso la elección es realizada por el entorno (selección externa).

La sintaxis descrita hasta aquí sólo permite la definición de procesos aislados, todos ellos finitos. La *aplicación de constante* $[P\langle x\rangle]$ soluciona ambos aspectos; en primer lugar, realiza el análogo de una llamada de procedimiento –literalmente, una invocación de proceso–, creando una nueva instancia del proceso designado por la constante, y transfiriendo además su hilo de control. En segundo lugar, introduce la posibilidad de realizar definiciones *recursivas*, dando lugar a definiciones potencialmente infinitas.

El operador de *replicación* $[!P]$, introducido por Milner en [Mil92], es equivalente al anterior, y permite sustituirlo por completo. Permite disponer de infinitas copias de un proceso, concurrentes entre sí, eliminando la necesidad de definiciones recursivas. Habitualmente se asume que el proceso replicado está prefijado por una acción de entrada, disponiendo así de una *replicación guardada*, en la que cada nueva copia del proceso se activa sólo en respuesta a una sincronización. Las dos construcciones son útiles en circunstancias diferentes, por lo que en esta sintaxis, como en [BS95], se admite el uso de ambas, sin que ésto tenga mayores consecuencias.

El operador de *comparación*, omitido muy a menudo, introduce un tipo muy básico de condicional, en el que un proceso se activa o no dependiendo de la igualdad de dos nombres. Es lo bastante potente para definir cualquier condición necesaria, a pesar de sus limitaciones; no obstante, normalmente se sustituye con ventaja por una auténtica estructura *alternativa* (doble), expresada aquí mediante la clásica sintaxis **if**...**then**...**else**, la más frecuente desde su inclusión en Ccs. En otras presentaciones se utilizan notaciones muy variadas, como el **if** $(b \triangleright P\,;\,Q)$ introducido por Liu y Walker en el cálculo-π extendido [LW95], o el $(b \hookrightarrow P)$, usado por Sangiorgi y Walker en [SW01], si bien generalizándolo a una estructura condicional múltiple. En todo caso, su significado es el habitual; y aunque realmente se trata de un operador derivado, se proporciona con frecuencia como primitivo. En la sección C.2.1 se discuten algunos detalles más sobre el modo en que este operador se utiliza a lo largo de esta tesis.

Sin embargo, el operador derivado más conocido es la *composición secuencial* $[P;Q]$ al estilo de Csp, cuya definición en términos del cálculo es inmediata, pero no trivial, tal como se explica en la sección C.2.2. Dada su utilidad y la frecuencia con la que será necesario, se introduce también como un operador primitivo, evitando con ello las engorrosas manipulaciones sintácticas que se producirían al definirlo como una construcción de segundo orden.

Finalmente, los operadores de *abstracción* y *concreción*, definidos en la parte superior, proceden de una generalización y refinamiento de la sintaxis, que pretende aumentar su correspondencia con los aspectos semánticos. La abstracción describe un agente en términos de un conjunto de nombres aún no especificados, identificando los aspectos comunes de las operaciones de recepción y restricción, relativos a la noción de *ligadura* de nombres. La concreción se define

como el operador dual a éste, que expresa la utilización de unos nombres concretos en un proceso abstracto, esto es, la consecuencia directa de un acto de sincronización, provocado por una emisión o una aplicación de constante. Se trata, en resumen, de la otra mitad de la unificación provocada por la reducción de dos términos que interactúan. Ninguno de los dos operadores es utilizado, salvo de manera indirecta, a lo largo de esta tesis, y sólo se incluyen aquí con el fin de proporcionar la definición sintáctica completa. En cualquier caso, este Apéndice se remite a la bibliografía [Mil93b, BS95] para mayores detalles sobre su uso y significado.

En resumen, esta sintaxis es básicamente la introducida por Milner en [Mil93b], que es también la que se emplea en [Mil99]. Sin embargo, al igual que se hacía en [BS95], se utiliza una versión ampliada que incluye todos los operadores que han sido descrito como básicos en alguna variante del cálculo, incluso cuando éstos son redundantes entre sí. La única excepción significativa es el operador de punto fijo (**fix**), introducido por Parrow [Par01] por analogía con la construcción homóloga de Ccs [Mil89]. Este detalle se discute brevemente en la sección D.1, concretamente en la nota al pie de la página 395. Aunque no se incluye explícitamente en esta sintaxis, se utiliza sin embargo, por precisión, en dos definiciones de la sección 7.4.3. En ambos casos esta tesis se remite a la definición de Parrow, mencionada más arriba.

En cuanto a los operadores derivados, se incluyen también los más importantes entre los clásicos. De entre los no considerados, merece la pena destacar el de concatenación de procesos ($P \frown Q$) y los de ocultación ($P \setminus x$) y conjunción paralela ($P \parallel Q$), análogos a sus homónimos de Csp. Ninguno de estos es estrictamente necesario, y ni siquiera resultan convenientes en el contexto de esta tesis, por lo que no serán utilizados en lugar alguno.

Apéndice C

Dialecto del Cálculo-π

Uno de los postulados sobre los que se basa el planteamiento general de esta tesis es el hecho de que en Informática, como en cualquier otra ciencia o técnica, para realizar el desarrollo de una teoría con el máximo rigor y precisión posibles resulta imprescindible un uso adecuado del lenguaje de las matemáticas, normalmente bajo la forma de algún sistema formal.

A lo largo de esta tesis se utilizan varios de estos formalismos, implícita o explícitamente. Es especialmente notorio el papel de tres de ellos, todos de naturaleza algebraica: la teoría de conjuntos, la teoría de relaciones, y el cálculo-π. Los dos primeros son sobradamente conocidos, y se emplean con su forma y notación habitual. El papel del tercero es posiblemente el más fundamental, en particular durante la definición de la semántica del lenguaje $\mathcal{P}i\mathcal{L}ar$, descrita en el Capítulo 7. En su uso se han adoptado ciertas convenciones, por lo que se requiere una serie de comentarios adicionales; a ellos se dedica este Apéndice.

C.1 Versión del Cálculo

En primer lugar, debe quedar claro que *no* se está usando el cálculo-π puro, si es que tal cosa pudiera decirse. En realidad, como ya se ha podido ver en el Apéndice B, y en particular en la Figura B.1, no hay *un* cálculo-π, sino muchos. Incluso excluyendo a todos los precedentes y a los sucesores menos directos, se dispone todavía de múltiples variantes del formalismo, y la diversidad de notaciones empleadas es aún mayor.

El cálculo se presenta en su forma más pura, como es lógico, en la versión original, esto es, el cálculo-π monádico (no tipado) de primer orden de Milner, Parrow y Walker [MPW92a, MPW92b]. Es el más básico, y el esencial en el sentido de que, de un modo u otro, todas las demás variantes del cálculo pueden ser descritas en función de éste. Sin embargo, su naturaleza monádica (unaria) complica en exceso la descripción de comunicaciones complejas, por lo que no es muy utilizado en la práctica.

Sin duda la variante más habitual es el cálculo-π poliádico de primer orden de Milner [Mil93b], que es considerada de algún modo como la versión canónica. Sin embargo, es habitual que distintos autores definan ligeras variaciones sobre éste, mediante la adición o eliminación de operadores, con el fin de facilitar un enfoque dado. El propio Milner lo ha hecho en más de una ocasión, incluso en la que hasta el momento es la última definición del cálculo [Mil99] –y que,

en esencia, sigue siendo la misma–. Por otra parte, no resulta nada extraña la introducción de operadores derivados, algunos de los cuales ya han sido mencionados.

La poliadicidad (aridad múltiple) introdujo una mayor capacidad expresiva en el cálculo, pero hizo también patente la necesidad de plantear cierta disciplina en el uso de los nombres. Con este objetivo, Milner [Mil93b] introdujo su célebre *sorting*, que es básicamente una teoría de tipos simplificada, sólo referida a los canales, y con una presentación poco habitual. No resulta extraño, sin embargo, que otros autores hayan profundizado aún más en el tema, desarrollando auténticas *teorías de tipos* para el cálculo-π, referidas tanto a procesos como a canales. Suponen una aplicación de la teoría habitual, en la tradición del cálculo-λ tipado, y los desarrollos análogos son de hecho perfectamente comparables. Se puede llegar a afirmar que, gracias a estos trabajos, la conocida correspondencia entre las versiones no-tipadas de ambos cálculos [Mil92, Mil93b] puede extenderse sin problemas a las versiones tipadas.

Existen numerosas versiones tipadas del cálculo-π. No siempre son estrictamente ortogonales, pero a menudo la combinación de sus características más interesantes resulta sencilla. Merece la pena destacar aquí tres de ellas, a saber:

- El *cálculo de objetos tipados TyCO* [Vas01]. Tiene su origen en el primer auténtico sistema de tipos propuesto para el cálculo-π poliádico, debido a Vasco Vasconcelos y Kohei Honda [VH93]. Era un sistema monomórfico con tipos recursivos, basado en el cálculo-ν del propio Honda y Mario Tokoro [HT91], que era a su vez una variante asíncrona no tipada del cálculo-π. De este sistema de tipos y la noción de entrada selectiva nació el cálculo de objetos original [VT93]. Éste fue reelaborado a continuación por el propio Vasconcelos, que le dotó de un sistema de tipos polimórfico [Vas94a], basado a su vez en una modificación del algoritmo de Damas-Milner. El cálculo resultante, conocido como TyCO [Vas94b], ha sido ampliamente utilizado desde entonces, y en la actualidad continúa evolucionando y siendo estudiado [RRV98, VLS98, Vas99, RV00, Vas01].
- El *cálculo-π polimórfico* de David N. Turner [Tur96]. En líneas generales, se trata de una traducción completa al cálculo-π de la teoría del cálculo-λ polimórfico de Girard y Reynolds. Al sistema resultante, de gran elegancia teórica, se le añade además un original esquema de tipos recursivos basado en la noción de bisimulación, obteniendo un cálculo de enorme potencia expresiva. Fue ésta la base sobre la que se diseñó el lenguaje Pict, tal vez la implementación más conocida del cálculo-π, desarrollada por Benjamin Pierce y el propio Turner [Pie98, PT00]. Este lenguaje ha sido el punto de partida de muchos estudios posteriores [Gay99], entre los que cabe destacar el trabajo de Paweł Wojciechowski y Peter Sewell en el denominado *cálculo-π nomádico* [WS00], que introduce colateralmente algunas interesantes aportaciones de carácter general.
- Los *cálculos-π tipados* descritos por Davide Sangiorgi y David Walker en [SW01]. Se trata del estudio detallado de un sistema de tipos que se va construyendo poco a poco, obteniendo versiones del cálculo cada vez más complejas. Comienzan por un esquema básico, al que denominan *cálculo-π con tipado simple* ($\pi^{\star,\sharp}$), consistente tan sólo en un conjunto de tipos-base –como mínimo, el tipo unidad– y los tipos de canal derivados. A éste se añaden después tipos-producto, uniones, registros, variantes y tipos recursivos, hasta llegar a un *cálculo-π poliádico* canónico ($\pi^{\sharp,\times,\mu}$), equivalente al *sorting* de Milner. A partir de aquí, se tratan por separado otros aspectos más avanzados: subtipado, linealidad

y polimorfismo. Interesa aquí hacer mención especial a este último, para el que se define un *cálculo-π polimórfico* canónico ($\pi^{\star,\sharp,\exists}$), utilizando de nuevo la analogía con el cálculo-λ de segundo orden de Girard y Reynolds. No es, sin embargo, equivalente al de Turner; por el contrario, se corresponde con un subconjunto estricto suyo.

La razón principal por la que se han considerado estas tres, de entre todas las variantes tipadas del cálculo-π, es el hecho de que comparten una característica fundamental: todas tienen soporte para *polimorfismo*. Este es un aspecto esencial, dado que se pretende utilizar el cálculo como base para la semántica de $\mathcal{P}i\mathcal{L}ar$. Las estructuras descritas en dicha semántica se sitúan a nivel arquitectónico, y por tanto manifiestan un elevado grado de abstracción. Su definición ha de ser, pues, lo más genérica posible; en concreto, debe ser independiente de detalles tales como el tipo de los datos manipulados, que sólo será conocido en cada instancia particular. El polimorfismo resulta pues decisivo a la hora de especificar estas estructuras abstractas.

Una vez identificada esta necesidad, se ha decidido optar, de entre las tres versiones citadas, por el cálculo-π polimórfico de Turner. Éste no es sólo el más conocido, sino también el más potente de todos ellos. El propio Turner postula [Tur96] que el sistema de Vasconcelos es un caso especial del suyo; además, debido a su uso de la entrada selectiva, adquiere un estilo sintáctico inusual, alejado de la práctica habitual del cálculo-π. Por su parte, Sangiorgi y Walker [SW01] tan sólo pretenden describir el fragmento de segundo orden del sistema de Turner. En sus propias palabras, si su cálculo equivale al Sistema F_2, el de Turner es comparable al Sistema F^ω de Girard. El polimorfismo natural en estos cálculos es, en cualquier caso, el de los *tipos existenciales*. En esto se acentúa la analogía entre el cálculo-π y el estilo de paso de continuación del cálculo-λ, que será brevemente comentada en la sección C.4.4. Es ésta la parte en la que todos ellos coinciden, y la única que será realmente utilizada.

Otro aspecto a tener en cuenta, y también fundamental para la definición de tipos abstractos, es la existencia de *tipos recursivos*. Como se ha dicho, Turner no utiliza la definición habitual, basada en árboles regulares, sino otra equivalente, que se fundamenta en la noción de bisimulación. Además, se admite el uso de tipos *mutuamente recursivos*, no muy frecuentes en las implementaciones [Pie98], pero que desde un punto de vista teórico son similares. Por otra parte, revisiones posteriores del cálculo-π polimórfico han introducido conceptos aún más avanzados, tales como los tipos lineales [HVK98, SW01] o los tipos-sesión. Ninguno de ellos es relevante en este contexto, y por tanto no serán considerados en absoluto.

Hasta este punto, se ha tratado exclusivamente con versiones de primer orden del cálculo[1]. Sin embargo, lo que se pretende en último término es describir un sistema reflexivo: por tanto, podría llegar a considerarse interesante utilizar alguna de sus variantes de orden superior. Por tanto,parece conveniente hacer aquí un breve comentario sobre las dos más significativas.

La primera es, por supuesto, el *cálculo-π de orden superior* o HO-π, definido en primer lugar por Davide Sangiorgi [San92, Mil93b, San93]. Se trata de una extensión natural del cálculo, en la que no sólo pueden comunicarse canales, sino incluso *procesos*. Aunque en apariencia esto

[1] A diferencia de lo que ocurre en el cálculo-λ, no parece lógico considerar al cálculo-π polimórfico como una versión *de segundo orden*. Lo es, en cierto modo, ya que en éste los canales tienen en efecto la capacidad de comunicar variables de tipo. No obstante, el concepto derivado por excelencia del cálculo no es el de tipo, sino el de *proceso*. Por tanto, el HO-π se define como un álgebra *sobre* procesos. En este sentido, lo más parecido a un cálculo-π de segundo orden es el CHOCS puro de Bent Thomsen [Tho90].

debería dotar al cálculo con mayor potencia expresiva, en realidad no es cierto. Como demostró el propio Sangiorgi, esta versión se puede codificar en términos del cálculo de primer orden, y por tanto son totalmente equivalentes. Esto significa que lo único que aporta la versión de orden superior es una sintaxis ampliada, que en ocasiones puede resultar conveniente. No es éste el caso de $\mathcal{P}i\mathcal{L}ar$, donde las abstracciones relacionadas apenas aparecen. De hecho, en este contexto, su uso tan sólo aumentaría la complejidad, y por tanto será totalmente descartado.

La segunda variante es el *cálculo-π extendido* de Liu y Walker [LW95]. Es una versión reducida del HO-π, que excluye la comunicación de procesos, pero sigue admitiendo la definición de agentes de orden superior. Fue diseñado específicamente para la realización de traducciones semánticas, y siempre ha sido utilizada con este fin [LW00, ZP97, ZP98]. Sin embargo, la semántica aquí descrita utiliza un enfoque diferente, para el que ya se proporcionan los operadores abstractos necesarios. No resulta conveniente, pues, remitirse a esta versión del cálculo.

En resumen, la versión del cálculo que se utilizará, especialmente en el Capítulo 7, es un **cálculo-π poliádico síncrono, tipado, de primer orden, no extendido, polimórfico, con tipos recursivos, registros y un tipo dinámico** Dyn. Más brevemente, se trata del cálculo-π polimórfico de Turner, sin considerar –pero sin rechazar– el subtipado, y ampliado con el tipo dinámico del cálculo-π nomádico. En la notación de Sangiorgi y Walker, se podría expresar como el sistema $\pi^{\sharp,\times,\oplus,\mu,\exists}$, unido al mencionado tipo *Dyn*. Aunque en teoría se deben tener en cuenta las diferencias existentes respecto a polimorfismo y recursividad, los efectos de las mismas en el uso final no serán significativos, al menos en esta tesis.

Este cálculo dispone además de los tipos-base habituales, y se amplía de dos maneras: con extensiones gramaticales específicas para algunas construcciones derivadas –secuencia, alternativa–, y con un soporte predefinido para ciertos tipos abstractos, concretamente pares, vectores, registros y listas. El sistema resultante es complejo, pero sólo usa construcciones estándar; todas han sido estudiadas con anterioridad, y su consistencia está sobradamente probada.

La versión del cálculo-π que se describe en este Apéndice no pretende, por tanto, introducir ninguna novedad. Tan sólo es original en el sentido de que reune una serie concreta de características, y en algunos detalles de su sintaxis. Por lo demás, es equivalente a una parte del cálculo-π polimórfico, o de su variante nomádica; y su biblioteca de tipos y funciones es mucho menos amplia, por ejemplo, que la de Pict [Pie98, PT98]. No obstante, se ha decidido designar a este sistema con un nombre propio, con el único objetivo de diferenciarlo fácilmente de otros dialectos. Por tanto, e inspirándose en la notación de los pares, en adelante se hablará de **cálculo-$\hat{\pi}$**. Este término no sólo abarca a la versión del formalismo, sino también al sistema de tipos, sus estructuras predefinidas y todas sus extensiones gramaticales.

C.2 Extensiones Gramaticales

En esta sección se describe la forma específica en la que se van a utilizar las principales construcciones derivadas del cálculo-π. Concretamente, se trata de la estructura alternativa y de la composición secuencial. Las dos pueden ser definidas en función del resto del cálculo; pero es muy habitual proporcionarlas como primitivas, y de hecho ya fueron incluidas en la gramática de la Tabla B.1. El cálculo-$\hat{\pi}$ también las considera como tales, pero se da aquí una justificación de su definición, al tiempo que se detalla brevemente su interpretación semántica.

C.2.1 Estructura Alternativa

Se utiliza aquí, como ya se ha visto, la clásica estructura alternativa doble (**if-then-else**), a partir de la cual es trivial definir el resto. Su traducción a un álgebra de procesos con selección es perfectamente conocida desde los tiempos de CCS [Mil89], y se basa en la definición de tres procesos: un selector y dos valores de verdad absolutos.

$$\begin{aligned} [\![\, \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q \,]\!] &\triangleq (\nu\, t, f)(\overline{b}\langle\, t, f\,\rangle.\,(t.\,P + f.\,Q)) \\ \mathrm{True}(b) &\triangleq b(t, f).\,\overline{t} \\ \mathrm{False}(b) &\triangleq b(t, f).\,\overline{f} \end{aligned}$$

Obviamente, esta construcción se verá grandemente simplificada por la adición de booleanos, que hará redundante la definición de los valores de verdad como derivados. Por ejemplo, usando las constantes lógicas se puede implementar fácilmente en términos del operador de comparación o *matching*, que también se define como derivado [Mil93b].

$$[\![\, \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q \,]\!] \triangleq (\,[\,b = \textbf{true}\,]\,P \mid [\,b = \textbf{false}\,]\,Q\,)$$

Sin embargo, una vez que se puede distinguir entre verdadero y falso, no parece razonable tener que hacerlo a través de canales, sino que es preferible proporcionar un operador lógico primitivo. El más sencillo es, precisamente, el propio **if**, que por este motivo es incluido en la sintaxis de todo álgebra con paso de valor desde CCS.

La integración de condicionales en el álgebra hace posible recuperar casi todas las estructuras de control tradicionales. Por ejemplo, un bucle-*mientras* se implementaría como sigue:

$$[\![\mathrm{while}(b, P)]\!] \triangleq \textbf{if}\ b\ \textbf{then}\ (P;\, [\![\mathrm{while}\langle\, b, P\,\rangle]\!])\ \textbf{else}\ \textbf{0}$$

Su rasgo más notable es el uso del operador de composición secuencial, que será definido a continuación; sin éste, se tendría una definición concurrente y fuertemente indeterminista. Sin embargo, normalmente será preferible definir bucles adaptados a cada situación concreta. De hecho, en el Capítulo 7 se utiliza un buen número de ellos, en la forma de iteradores.

C.2.2 Composición Secuencial

El operador de composición secuencial, habitualmente expresado por un punto y coma (;), suele ser primitivo en las álgebras de procesos del tipo-CSP [Hoa85]. Sin embargo, no suele aparecer como tal en las álgebras tipo-CCS [Mil89], entre las que se incluye el propio cálculo-π, ya que desde el principio quedó claro que era posible emularlo con el resto de construcciones. En particular, mediante la combinación de la sincronización (pura) y la composición paralela:

$$P; Q \triangleq (\nu\, done)(P.\,\overline{done} \mid done.\,Q) \quad done \notin \mathrm{fn}(P, Q)$$

Ésta no es la única forma de codificar la composición secuencial en términos del cálculo. De hecho, ni siquiera se sugiere que sea, necesariamente, una implementación razonable en un caso práctico. Pero sí resulta obvio, dada su brevedad, que la introducción de un constructor de *secuencia* resulta muy simple, casi trivial, en términos de un formalismo síncrono.

No obstante, es justo señalar que esta última definición lo es tan sólo en apariencia. En realidad, la expresión dada no es ni siquiera gramatical: la sintaxis del cálculo-π de primer orden no permite especificar ninguna acción tras una aplicación de constante, por lo que no se puede añadir el envío de la señal *done* tras la finalización del proceso P.

No obstante, el significado de la expresión es el obvio. Se puede dar una definición más formal asumiendo que P es en realidad una abstracción de proceso, que siempre termina enviando una señal de finalización por un canal parametrizado:

$$\begin{aligned} \mathrm{P}(w) &\triangleq (\ldots \overline{w}) \\ [\![\mathrm{P}(w); Q]\!] &\triangleq (\nu\, done)([\![\mathrm{P}\langle\, done\,\rangle]\!] \mid done.\, [\![Q]\!]) \quad done \notin \mathrm{fn}(Q) \end{aligned}$$

Este enfoque es válido incluso en las versiones asíncronas del cálculo-π; es, por ejemplo, el esquema utilizado en Pict [PT00]. Sin embargo, la necesidad de incluir la señal de terminación en cada definición de proceso resulta incómoda; por otra parte, no resulta en absoluto problemático asumirla como implícita. Por este motivo, se prefiere incluir la composición secuencial en el cálculo como primitiva, una vez que se ha demostrado su total coherencia. En realidad, no llega a ser un operador derivado, sino que más bien se trata simplemente de *azúcar sintáctico*.

C.3 Tipos Básicos de Datos

Todo sistema de tipos parte de un conjunto de tipos-base, que suele constar como mínimo del tipo vacío (§C.5). Es muy habitual, incluso dentro del propio cálculo-π [SW01], que este conjunto incluya una serie de tipos estándar de alto nivel, que traen consigo sus propios operadores. Esta adición simplifica notablemente el uso del lenguaje, ya que evita tener que prestar atención a detalles secundarios que de otro modo podrían llegar a resultar engorrosos.

Las abstracciones que se pretende proporcionar son casi tan básicas como el propio cálculo: datos lógicos y numéricos. En el cálculo-$\hat{\pi}$, como en $\mathcal{P}i\mathcal{L}ar$, se consideran sólo en sus formas más simples: datos *booleanos* y números *enteros*. A éstos se añaden las *cadenas* de caracteres, que inicialmente se plantean como un tipo inerte, pero a medio plazo serán fundamentales.

C.3.1 Booleanos [**Bool**]

Se ha de comenzar señalando que el cálculo-π es por sí solo lo bastante potente como para expresar las nociones lógicas, de manera similar a como ocurre con el cálculo-λ. Para esto, basta con definir los valores de verdad como procesos constantes a los que se envían señales; dado el emisor de una respuesta, se podrá determinar la veracidad o falsedad de la misma.

Esta idea ya ha sido expresada en la primera formulación de la estructura alternativa, en la sección C.2.1, que incluye la definición de los procesos True y False. En ambos se está emulando un dato lógico como un canal que envía dos señales síncronas, y es utilizado por un proceso capaz de seleccionar a una de ellas. Partiendo de esta expresión, es muy fácil inferir una codificación para el resto de las operaciones estándar.

Aunque esta codificación es bastante sencilla, parece claro que no resulta práctico tener que operar con valores lógicos en la forma de actos de comunicación sobre canales binarios. Por tanto, una vez que se ha mostrado su completa coherencia con el cálculo, se decide proporcionar

como primitivo un tipo para los valores lógicos. Este tipo se define como *booleano*, es decir, sigue las reglas y mantiene las propiedades de un álgebra de Boole.

Su descripción resulta trivial, pues se utiliza la forma y notación convencional. Se tienen por tanto las dos constantes lógicas, que reciben los nombres habituales de **true** y **false**. En última instancia, por supuesto, toda variable lógica remite su valor a uno de éstos. Se definen también los operadores clásicos de conjunción ($\wedge$), disyunción ($\vee$) y negación ($\neg$). Su función, uso y comportamiento serán los esperados. Se admitirá, además, una forma general abreviada de los dos primeros para las expresiones múltiples que involucren subíndices, como es el caso de los vectores. Esto es, se dispone de un **and**-atorio ($\bigwedge_i^n$) y un **or**-atorio ($\bigvee_i^n$).

En el contexto del cálculo, un booleano sólo podrá aparecer en una posición apta para un valor, es decir, como parámetro de una interacción. Esto se aplicará en general a todos los tipos predefinidos, como los enteros y cadenas; sin embargo, hay en este caso una excepción, ya conocida: un booleano podrá aparecer, además, como el selector de una estructura alternativa, es decir, como la condición de un constructor-**if**.

En definitiva, su comportamiento, sea como valor o como condición, no difiere del habitual, por lo que no se considera preciso entrar a detallarlo con mayor profundidad.

C.3.2 Enteros [Int]

Al igual que ocurría con los booleanos, es posible representar los números enteros directamente sobre el cálculo-π, siguiendo una técnica análoga a los numerales de Church [Mil93b, San92]. Un numeral n se codifica como el número de sincronizaciones sobre un canal dado:

$$[\![\, \underline{n} \,]\!] \quad \triangleq \quad (\lambda\, y, z)\, \overline{y}^n . \overline{z}$$

donde $\overline{y}^n$ simboliza a una secuencia de envíos $\overline{y} \ldots \overline{y}$ de longitud n. Luego, sólo es necesario proporcionar operadores capaces de manipular estos numerales, como por ejemplo la suma:

$$\text{Plus} \quad \triangleq \quad (\lambda\, M, N, y, z)\, (\nu\, x)(\, \langle\, y, x\, \rangle M \mid x . \langle\, y, z\, \rangle N\,)$$

donde Plus es una abstracción que toma dos numerales (M, N) como el definido más arriba, y los aplica en secuencia sobre el mismo canal (y), de modo que se obtiene una secuencia de co-nombres cuya longitud es evidentemente la suma.

Obsérvese que los numerales no son canales, sino abstracciones de proceso. Esto explica la complejidad del operador Plus, que recibe numerales como parámetros, y es por tanto él mismo una abstracción *de segundo orden*. Otros operadores más potentes, como el producto, tendrán órdenes aún mayores. No obstante, esto no supone dificultad alguna, incluso si se pretende limitarse a usar el cálculo-π de primer orden; como se recordará, se ha demostrado [San92, San93] que éste es equivalente al HO-π, y por lo tanto resulta suficiente.

El cálculo, por sí solo, es pues capaz de codificar números y emular su aritmética. Sin embargo, y al igual que ocurría con los booleanos, no parece práctico manejarlos en esta representación. Por el contrario, una vez que se sabe que son congruentes con el formalismo, se incorporan como un tipo primitivo, y se manipulan directamente con la notación matemática habitual.

Se proporciona un tipo Int de números enteros, incluido como uno de los tipos-base del sistema. Por tanto, se usa siempre como el valor intercambiado en una comunicación, bien en forma

de variable, bien como una constante literal. Se admite el uso de la notación numérica usual, tanto para estos literales como para los operadores, que son los tradicionales de la aritmética entera infija: suma y resta $(+, -)$, producto y cociente $(*, /)$, y el resto o módulo $(\%)$. Se incluyen además los operadores relacionales clásicos, tanto para la igualdad $(=)$ como para la desigualdad $(\neq, >, \geq, <, \leq)$. En ambos casos se mantienen las reglas de precedencia habituales, que podrán ser alteradas mediante el uso de paréntesis.

Se podrían incorporar de manera similar otros tipos numéricos, en particular los reales. Sin embargo, su papel es mucho menos importante, y no se ha considerado necesario incluirlos. Debe tenerse en cuenta que, al no haber en el cálculo ni variables ni asignaciones propiamente dichas, toda expresión aritmética habrá de aparecer como parámetro de una comunicación. Pero aparte de esto, su uso resulta completamente normal, y no requiere mayor comentario.

C.3.3 Cadenas [**String**]

En un formalismo tan básico como el cálculo-π, las *cadenas* de caracteres tienen un papel ambivalente. Es patente, por una parte, que su reducción a términos del cálculo puro no es en absoluto trivial, y de hecho tiene una estructura interna fácilmente identificable. Sin embargo, se trata también de un tipo de datos básico y muy habitual. Aquí será utilizado en una de sus formas más simples, y presentará unas posibilidades de manipulación muy limitadas, hasta el punto de que podrá considerarse como un dato prácticamente inerte.

Como en los casos anteriores, se justificará la inclusión de las cadenas en el cálculo demostrando previamente que éste es capaz de codificarlas. Como ya se ha dicho, esta codificación define una estructura de cierta complejidad; sin embargo, su descripción a nivel conceptual resulta bastante sencilla cuando ya se dispone de números, en particular los enteros. Basta entonces simplemente con definir un *código* de tipo alfabético, en el que cada número se corresponde con un carácter; la cadena se obtiene como una secuencia de tales caracteres, sea como una comunicación poliádica, o como una serie de comunicaciones sobre el mismo canal –ambas ideas son, en el fondo, lo mismo–. Es, en definitiva, análoga a una *tupla* de caracteres.

Sin embargo, el tipo cadena se incluye en el cálculo-$\hat{\pi}$ como *primitivo*. En particular, no se define ningún tipo carácter; por tanto, una cadena se utiliza como un bloque atómico de información. No se considera necesario en este contexto, y por tanto no se proporciona, ningún operador capaz de dividirla en fragmentos de menor tamaño. De hecho, se abstrae por completo el concepto de tamaño, ya que no se dispondrá de método alguno para comparar entre sí dos cadenas diferentes, o para determinar simplemente su longitud.

Sobre las cadenas únicamente se admitirán dos operaciones primitivas. La primera es una comparación de igualdad $(=)$, que averigua si dos cadenas son idénticas, sin considerar grados de similitud, y devuelve un resultado de tipo booleano. La segunda es un operador de *concatenación*, que toma dos cadenas-origen A y B, y produce una tercera con sus contenidos puestos en secuencia. Se expresa como $A^\frown B$, y dado que el resultado es atómico, puede considerarse como una simple fusión. Por el mismo motivo, es obvio también que es irreversible.

Como se comentó en el Capítulo 4, las posibilidades de un tipo cadena pueden ser casi ilimitadas en los lenguajes –formales o no– que disponen de un operador de entrecomillado (*quote*) reversible, hasta el punto de permitir varias formas de autorreferencia [Per85]. Este cálculo se sitúa, sin embargo, en el extremo opuesto. No sólo carece de tal operador, sino de otros

mucho más convencionales. Las comillas se admiten, simplemente, como el medio normal para expresar cadenas literales –v.g., `"cretense"`–. En definitiva, solamente se pretende utilizar las cadenas como datos atómicos, y no se explotarán el resto de sus capacidades.

El motivo principal para la inclusión de cadenas es la posibilidad de usarlas como *etiquetas*. Esto será especialmente relevante tras la definición de las *listas nombradas*, que las emplearán como selectores. Este aspecto será descrito en profundidad a lo largo de la segunda parte de la sección C.4.4, a la que me remito para más detalles.

C.4 Tipos Abstractos de Datos

Una vez establecidos los tipos básicos, conviene identificar las estructuras más habituales en el cálculo, con el fin de evitar tener que repetir su definición en cada especificación.

Todas las construcciones que se describen en esta sección han sido presentadas en alguna de las variantes del cálculo-π poliádico desde su definición original [Mil93b], que ya describe la mayoría de ellas. El único objetivo de esta sección es, por tanto, fijar la notación utilizada para las mismas; su coherencia teórica ya ha sido justificada en varias ocasiones.

En realidad, sólo dos de los tipos detallados en esta sección –los *pares* y las *listas*– son auténticos tipos abstractos de datos. El resto son simplemente estructuras; sin embargo, el conjunto de todas ellas, junto a la capacidad para el polimorfismo, constituye una base adecuada para la definición de las abstracciones necesarias, como aquéllas que dan forma a la semántica de $\mathcal{P}i\mathcal{L}ar$, tal como se describe en el Capítulo 7.

Se ha decidido no incorporar un tipo *conjunto* en términos del cálculo. En realidad, suele implementarse [Woj00] mediante una lista, tal como se hace en $\mathcal{P}i\mathcal{L}ar$. Sin embargo, se trata en realidad de un concepto más básico: el propio cálculo-π se define sobre la teoría de conjuntos, ya que consiste en una serie de reescrituras algebraicas sobre multiconjuntos de términos-π. En resumen, se asumirá cuando sea preciso la disponibilidad de esta noción, así como de sus operadores matemáticos. En cualquier caso, esto no será muy habitual.

C.4.1 Tuplas y Vectores

Las *tuplas*, es decir, los tipos-producto, son la aportación fundamental del cálculo-π poliádico, y el concepto que justifica el planteamiento de teorías de tipos. Las consecuencias de su inclusión son enormes, y han permitido, por ejemplo, la traducción de partes fundamentales de la teoría del cálculo-λ a contextos concurrentes. No obstante, se pueden codificar en el cálculo-π monádico como una secuencia de interacciones [Mil93b]; por tanto, se integran con total naturalidad en su definición conceptual, aunque con ciertos matices –en particular, la definición de errores–.

Se trata de un tipo estándar, admitido habitualmente en todas las versiones del cálculo, tanto tipadas como no tipadas. En su forma habitual, se describe como una secuencia de valores de cualquier tipo y tamaño, separados por comas. Su longitud no tiene límites preestablecidos; pero no puede variar una vez definida, por consistencia con la teoría de tipos.

Lo único realmente digno de mención es su notación abreviada, que fue proporcionada desde un principio [Mil93b]. Por tanto, se puede representar cualquier tupla mediante la forma $\widetilde{v}$, que expresa simplemente una secuencia de valores. Suele reservarse para fragmentos homogéneos, es decir, compuestos por datos del mismo tipo, pero esto no es estrictamente necesario.

El *vector* no es más que un tipo especial de tupla homogénea, cuyo tratamiento no introduce característica alguna, pero que es particularmente habitual. Normalmente se denota como una tupla más, aunque se puede indicar como $\vec{v}$ cuando se desea mostrar de manera explícita que el dato es concebido como un vector.

Se considera que la longitud de los vectores es potencialmente infinita, y por ello se manipula con especial libertad. De hecho, y aunque no es muy ortodoxo, resulta habitual utilizar un mismo vector con distintas longitudes en distintos lugares [Mil89, ZP97]. No se debe pensar por esto, sin embargo, que se trata de un tipo dinámico. Puede verse, más bien, como una tupla de tamaño indefinido de la que sólo se utiliza una parte; lo que varía en las distintas ocurrencias es solamente la longitud del fragmento significativo.

Tanto en tuplas como en vectores suele expresarse el acceso a sus elementos individuales mediante el uso de subíndices (v_i), que era propio, originalmente, de los segundos. Téngase en cuenta que un elemento designado de este modo puede ya utilizarse como un valor, o incluso como un canal, absolutamente independiente; y por lo tanto no tiene por qué asumirse ningún tipo de relación con el resto de la tupla.

C.4.2 Registros

Una vez que se dispone de tuplas, se considera prácticamente trivial la definición de *registros*. De hecho, suelen ir asociados en las teorías de tipos [SW01], y ambos son consideradas como distintas manifestaciones de un mismo tipo-producto. Por tanto, están incluidos por defecto en todas las variantes del cálculo-π poliádico tipado, aunque no siempre de manera explícita.

El concepto de registro es el habitual: se trata simplemente de una tupla en la que cada elemento, aquí denominado *campo*, tiene asociado una etiqueta que le sirve de selector. El objetivo es poder referenciar un campo mediante esta etiqueta, sin tener que considerar a los restantes, en una expresión dada; de este modo se ésta se simplifica de manera notable. De hecho, la importancia que adquieren los registros en un álgebra de procesos deriva del hecho de que proporcionan un medio estático para descomponer estructuras, sin hacerlas dependientes de una interacción. Esto se puede mostrar mejor con un ejemplo:

$$Punto \triangleq (\dagger x : Int,\ \dagger y : Int,\ \dagger z : Int)$$

Sea el tipo-registro *Punto*, definido de este modo, y un dato $p : Punto$. Entonces, para acceder a su segunda coordenada (y), en el cálculo puro sería necesario hacer:

$$\overline{a}\langle p\rangle.\mathbf{0} \mid a(px, py, pz).Q\langle py, \ldots\rangle$$

Mientras que, una vez definidos los registros, basta simplemente con:

$$Q\langle p\dagger y, \ldots\rangle$$

Incluso dejando a un lado el paso de comunicación, puede observarse que en el primer caso era necesario introducir al menos tres nombres nuevos como parámetros de entrada, cuando sólo se pretendía utilizar uno; en cambio, en el segundo caso, esto es totalmente innecesario.

La sintaxis de acceso a los registros resulta ligeramente conflictiva en un álgebra de procesos del tipo-CCS. En un lenguaje convencional, el acceso típico se realiza mediante la *notación*

punto: es decir, se utiliza el punto como separador entre el nombre del registro (cualificador) y el nombre del campo (selector). Sin ir más lejos, ésta es la notación empleada en $\mathcal{P}i\mathcal{L}ar$. Sin embargo, se ha de descartar su uso en el cálculo-π, donde provocaría una fuerte ambigüedad. Ha de tenerse en cuenta que en el cálculo el punto ya era utilizado para separar entre sí los términos de una secuencia de acciones; por tanto, el acceso a un campo de un registro podría confundirse con una serie de recepciones de señales, esto es, sincronizaciones puras.

Por tanto, se utilizará una notación diferente para los registros, la misma que ya ha sido empleada en el ejemplo anterior. En realidad, su estructura es idéntica a la de la notación punto, ya que sigue considerando cualificadores y selectores; tan sólo cambia el separador, que será el símbolo $\dagger$. Como es habitual, se admite la aparición de varios de estos símbolos en un mismo término, indicando el acceso a una secuencia de registros anidados. No obstante, ha de tenerse en cuenta que dicho término se considerará como un *único* nombre.

Se debe proporcionar además una sintaxis para el acceso en el sentido contrario, es decir, el envío de un valor como campo de un registro. Debe recordarse que el cálculo-π carece de operador de asignación, y que toda ligadura entre valores y variables se produce como parte de una interacción. En este caso, la notación más frecuente [SW01] para el acceso a un registro indica los valores como parámetros de una tupla, y exige además indicar explícitamente el nombre de los campos accedidos, utilizando como separador el símbolo de igualdad (=). No obstante, y dado que éste es ya utilizado en las comparaciones de números y cadenas, se ha preferido usar en su lugar el símbolo $\prec$, que indica además el sentido de la asignación. Con el objetivo de enfatizar que se trata de un registro, se marca de nuevo la etiqueta con el símbolo $\dagger$. Así, por ejemplo, una asignación para un término *Punto* quedaría como:

$$\overline{a}\langle \dagger x \prec 0,\ \dagger y \prec 0,\ \dagger z \prec 0\rangle.\mathbf{0} \mid a(p)\ldots \quad \text{donde } a :\uparrow \mathit{Punto} \text{ y } p : \mathit{Punto}$$

La utilización ortodoxa del tipo registro exige que todos los campos aparezcan siempre en la misma posición, y que ninguno de ellos sea excluido en ningún caso. No obstante, esta restricción no es demasiado importante, y puede relajarse sin más que suponer un paso previo que reordene los parámetros de la forma adecuada; sin embargo, esto no se hará en esta tesis.

Muy a menudo, se dará la situación en la que se debe acceder a un campo de un registro, pero no se dispone aún del registro, sino de un canal (un *puntero*) hacia él. La resolución es inmediata; sin embargo, es tal su frecuencia que se proporciona una *notación abreviada* para el acceso a un campo en estas circunstancias, es decir, a través de un canal:

$$a\ddagger y = a(p).p\dagger y \quad \text{donde } a :\uparrow \mathit{Punto} \text{ y } p : \mathit{Punto}$$

Obsérvese, no obstante, que aquí se produce una comunicación, y por tanto se consume un dato. Esto deberá tenerse en cuenta al considerar las interacciones, cuyo emparejamiento define la interpretación de un proceso. Por otra parte, y como es evidente, esta notación sólo es práctica cuando se necesita hacer un *único* acceso al registro intermedio.

Los registros resultan determinantes en la definición de cualquier tipo abstracto de datos, en particular la de aquéllos que proporcionan la estructura semántica de $\mathcal{P}i\mathcal{L}ar$. La notación se ha elegido, pues, de modo que no utilizase ningún símbolo habitual en ninguna variante del cálculo-π, para evitar confusiones, pero también de modo que resultase lo suficientemente llamativa.

Finalmente, se ha de insistir en que toda expresión de acceso a un registro –por ejemplo, un término $p\dagger x$– no es más que una forma abreviada para hacer referencia a un *nombre*. Éste no sólo podrá ser utilizado como un valor, sino también como un canal. Por tanto, dependiendo de su tipo, podrá aparecer incluso como sujeto activo de una interacción.

C.4.3 Pares

Una de las características fundamentales del cálculo-π, que lo distingue de otras álgebras de procesos, es su capacidad para representar funciones como procesos [Mil92, Mil99]. De hecho, la construcción resultante de esta analogía es utilizada con naturalidad y frecuencia en la descripción de sistemas concurrentes, y no sólo en sus fragmentos secuenciales, hasta el punto de ser la que da al cálculo su estilo característico. Éste será examinado con mayor detalle en el apartado C.4.4, y tiene ciertas dificultades, que ya fueron expuestas en la sección 6.6.

Los procesos de este tipo reciben cero o más entradas, y proporcionan al menos una salida. Se puede asumir, sin pérdida de generalidad, que las primeras se reciben a través de un único canal (poliádico); del mismo modo, se puede postular un único canal de salida, habitualmente monádico, aunque hay varias construcciones posibles. Hay varias razones para esta separación, que ya ha sido justificada en la sección 7.2.3; una de las más importantes es sin duda la coherencia con la teoría de tipos [BS95]: en el caso más normal, el número y tipo de los datos de entrada y salida será completamente diverso.

Será por tanto frecuente encontrar a los canales organizados en *pares* de entrada-salida. De hecho, es éste el primer aspecto que se puso de relieve en la semántica (§7.2.3), donde afecta especialmente a los puertos, definidos como entidades comunicantes que han de exportar una estructura estática. Parece razonable, por lo tanto, proporcionar un tipo abstracto de datos para esta construcción, que permita declararlos y manejarlos explícitamente como pares. Más aún, se le dotará incluso de una sintaxis especial que simplifique su uso:

$$\widehat{pa} \quad = \quad (\, pa_i, \, pa_o \,)$$

Se define por tanto un *par* como una tupla de dos canales, que se designa mediante un nombre. Este nombre se marca, opcionalmente, mediante un un acento circunflejo ($\widehat{pa}$), con el fin de hacer explícita su naturaleza. Por supuesto, puede ser también utilizado sin esta marca, como cualquier otro canal del cálculo. Por otra parte, cada uno de los canales puede ser referenciado mediante este mismo nombre –sin acento–, unido a un subíndice que indica, respectivamente, cuál es el de entrada (pa_i) y cuál el de salida (pa_o). No hay ninguna relación implícita entre ellos; se utilizan como dos nombres totalmente independientes, y sus tipos pueden ser diferentes.

También se proporciona una notación especial para indicar cuándo una tupla de dos valores debe ser considerada como un par. En este caso, se utilizarán *corchetes* como delimitadores, en lugar de los paréntesis de una tupla convencional. En realidad, esto sólo será necesario cuando el dato aparezca aislado, como en el acceso a un registro. Durante una comunicación normal, se asume como correcta la recepción de una tupla binaria en un par; en tal caso, la notación especial no sería necesaria, y de hecho se evitará explícitamente utilizarla.

En realidad, es claro que este tipo puede verse también como un caso particular de un registro, con un número y nombre de campos muy concretos y delimitados. Así pues, el par se

define formalmente como el tipo paramétrico (polimórfico) siguiente:

$$(Par\ \alpha\,\beta) \quad \triangleq \quad (\dagger i{:}\uparrow\alpha,\ \dagger o{:}\uparrow\beta)$$

De modo que la notación definida más arriba no es más que una forma abreviada para referirse a este tipo. Según esto, un par $\widehat{g}$ no es más que un registro declarado como $g : (Par\ \gamma\,\delta)$, y los subíndices no son más que formas abreviadas del acceso a sus campos, de modo que:

$$\begin{array}{rcl} g_i & = & \widehat{g}\dagger i \\ g_o & = & \widehat{g}\dagger o \\ [a,b] & = & (\dagger i \prec a, \dagger o \prec b) \end{array}$$

El tipo par es tal vez la construcción más característica de nuestra versión del cálculo-π. Es, por otra parte, notablemente sencilla, y tiene notables parecidos con estructuras definidas por algunos autores con estructura similar, aunque distinto propósito. Tal vez el mejor ejemplo sean los pares de Gordon [Gor00], definidos por analogía con el cálculo-λ tipado, que tienen una estructura parecida, pero cuya función es completamente distinta.

C.4.4 Listas

Sin duda, las *listas* son el tipo abstracto más utilizado en la semántica de $\mathcal{P}i\mathcal{L}ar$. Esto, en principio, no supone problema alguno, pues la definición de listas a partir de procesos se admite desde el principio en todas las variantes del cálculo-π poliádico. Sin embargo, es evidente que su manipulación directa con los medios habituales resulta bastante farragosa. Dada su frecuencia, se pretende simplificar lo más posible las expresiones que involucren listas; por ello, se dotará al cálculo-$\hat{\pi}$ de una sintaxis predefinida para las mismas.

Este apartado presenta, pues, esa sintaxis, inspirada por su difusión y compacidad en la de Lisp; en este particular coincidimos con la opinión de Pierce y Turner, que hacen lo propio al incluir listas en Pict [PT98]. Se demuestra, además, la consistencia de las operaciones introducidas mediante su traducción a procesos del cálculo-π polimórfico estándar.

Un aspecto significativo de las listas es que definen, por supuesto, un tipo recursivo. Sin embargo, esto no supone ningún problema, dado que se trata del tipo de recursión comúnmente admitida. Ya en la definición original del cálculo-π poliádico [Mil93b], Milner proporciona la siguiente codificación de listas, en la que se basan todas las posteriores:

$$\begin{array}{rcl} \mathrm{Nil} & \triangleq & (\lambda\, c, n)\, \overline{n} \\ \mathrm{Cons}(v,l) & \triangleq & (\lambda\, c, n)\, \overline{c}\langle\, v, l\,\rangle \end{array}$$

Estas dos abstracciones de proceso –cuyos nombres se basan, de nuevo, en los comandos de Lisp– constituyen la definición del tipo abstracto. La primera (Nil) define una lista vacía; la segunda es un constructor (Cons) que, dado un valor v y una lista l, elabora una nueva lista a partir de la anterior, en la que el valor indicado ha sido añadido al principio.

Por supuesto, esta definición es consistente con el sistema de *sorts* definido por el propio Milner. Por tanto, una lista será asociada a un *sort* determinado, el que habrán de tener todos

sus elementos. El *sorting* concreto para la codificación dada es el siguiente:

Definición del *sorting*			**Asignación de *sorts***		
VAL	$\mapsto$	*tipo elemento*	v	:	VAL
LIST	$\mapsto$	(CONS, NIL)	l	:	LIST
CONS	$\mapsto$	(VAL, LIST)	$Cons$	:	(VAL, LIST)$\frown$(CONS, NIL)
NIL	$\mapsto$	()	Nil	:	(CONS, NIL)

Obsérvese que los *sorts* LIST y CONS son (mutuamente) recursivos. Esto no es problema, pues como ya se ha dicho, el sistema lo admite. Sin embargo, lo que el *sorting* no soporta es el polimorfismo: una lista LIST sólo lo es de valores VAL. Para disponer de una lista de enteros sería necesario hacer una definición totalmente nueva, aunque prácticamente idéntica.

Como se dijo, son las estructuras abstractas de esta clase las que hacen especialmente interesante disponer de polimorfismo. De hecho, éste es el principal motivo por el que se eligió el cálculo-π polimórfico de Turner [Tur96], ya que nos permite disponer de las estructuras de segundo orden necesarias. De hecho, basándose en la codificación de Milner, Turner proporciona su propia definición de las listas, que es la que aquí utilizaremos:

$$\begin{aligned} \mathrm{Nil}(r) &\triangleq (\nu\, l)(\overline{r}(l) \mid !l(n,c).\,\overline{n}) \\ \mathrm{Cons}(hd, tl, r) &\triangleq (\nu\, l)(\overline{r}(l) \mid !l(n,c).\,\overline{c}(hd, tl)) \end{aligned}$$

Como puede verse, se introducen tres variaciones con respecto a la formulación de Milner. En primer lugar, las abstracciones de proceso han sido sustituidas por procesos parametrizados, que normalmente resultan más intuitivos. En segundo lugar, han sido transformados en procesos *funcionales*, esto es, capaces de emular el comportamiento de una función matemática [Mil92], mediante el artificio de enviar el resultado a través de un *canal de respuesta*, que en este caso es en ambos casos r. En tercer lugar, finalmente, se ha dotado a los procesos de persistencia mediante el uso del operador de replicación sobre el fragmento significativo. De hecho, podría decirse que lo que estos procesos-función envían por su canal de respuesta es en cada caso equivalente a la abstracción respectiva en la codificación de Milner.

Sin embargo, esta definición resultaría incompleta si no va acompañada de los tipos correspondientes. No obstante, sólo es necesario indicar los de los *constructores*, a saber:

$$\begin{aligned} (List\,\alpha) &\triangleq \mu L.\ \uparrow(\uparrow(),\ \uparrow(\alpha,\ L)) \\ \mathrm{Nil} &: (\sharp\alpha;\ \uparrow(List\,\alpha)) \\ \mathrm{Cons} &: (\sharp\alpha;\ \alpha,\ (List\,\alpha),\ \uparrow(List\,\alpha)) \end{aligned}$$

Conviene recordar que lo más interesante del sistema de tipos de Turner –o más bien de sus antecedentes, desde el Sistema de Tipos Simples de Church hasta el Sistema F^{ω} de Girard– es que a partir de una definición relativamente sencilla, como ésta, se pueden inferir con facilitad los tipos de todos los canales involucrados. Por ejemplo, lo primero que hacen ambos procesos es crear un canal nuevo (l), que luego se envía por el de respuesta (r). Es obvio, pues, que dicho canal tiene un tipo $List\,\alpha$, y por tanto se trata de una lista. De aquí se puede deducir también el tipo de sus parámetros: respectivamente, una señal síncrona $n : \uparrow()$, que indica una lista vacía,

y un canal de pares elemento-lista $c : \uparrow(\alpha,\, List\,\alpha)$, que es de hecho el auténtico constructor. Como puede verse, en este punto todos los canales tienen ya un tipo asignado.

La definición del tipo lista ($List\,\alpha$) que acaba de hacerse merece un comentario especial. Como cabía esperar, es al mismo tiempo un recursiva y polimórfica. El primer aspecto se indica con la notación-μ usual, y se interpreta de la manera habitual. De hecho, aunque se asume que se utiliza la semántica de bisimulación de Turner [Tur96], no habría diferencia alguna si se emplease la de árboles regulares [SW01], ya que estamos en el caso arquetípico. En lo que respecta al polimorfismo[2], la definición es obviamente paramétrica, y por tanto de segundo orden. El argumento (α) indica el tipo de los elementos de la lista; por tanto, podrá ser cualquiera, pero el mismo en todos ellos. Su interpretación es, pues, idéntica a la de un *tipo existencial*, como ya se había indicado. Obsérvese que esto no limita en modo alguno los valores que puede tomar la variable de tipo: podría ser una estructura compleja, o incluso otra lista.

Una vez se tiene el par de procesos que construyen el tipo abstracto lista, resulta sencillo utilizar el cálculo-π para elaborar las operaciones que lo manipulan. Para comenzar, se definen los procesos que implementan tres funciones clásicas de Lisp.

$$
\begin{array}{rcl}
\mathrm{Null} & : & (\sharp\alpha;\, (List\,\alpha),\, \uparrow Bool) \\
\mathrm{Null}(l,b) & \triangleq & (\nu\, n,c)(\overline{l}\langle\, n,c\,\rangle.\,(n().\,\overline{b}\langle\, \mathbf{true}\,\rangle + c(w).\,\overline{b}\langle\, \mathbf{false}\,\rangle) \\
\mathrm{Car} & : & (\sharp\alpha;\, (List\,\alpha),\, \uparrow\alpha) \\
\mathrm{Car}(l,e) & \triangleq & (\nu\, n,c)(\overline{l}\langle\, n,c\,\rangle.\,c(hd,tl).\,\overline{e}\langle\, hd\,\rangle) \\
\mathrm{Cdr} & : & (\sharp\alpha;\, (List\,\alpha),\, \uparrow(List\,\alpha)) \\
\mathrm{Cdr}(l,r) & \triangleq & (\nu\, n,c)(\overline{l}\langle\, n,c\,\rangle.\,c(hd,tl).\,\overline{r}\langle\, tl\,\rangle)
\end{array}
$$

Se trata de un test de lista vacía (Null), con forma de función booleana, y de dos proyecciones: la que obtiene el primer elemento de una lista (Car), y la que proporciona el resto de elementos, agrupados en una nueva lista (Cdr). Las tres son sobradamente conocidas, y su forma en el cálculo-π polimórfico mantiene el estilo habitual. El único detalle significativo, aplicable por otra parte a todas las definiciones de esta sección, es que de nuevo se trata de funciones simuladas como procesos, es decir, de *procesos-función* que envían explícitamente su resultado por un canal de respuesta. Por convención, dicho canal será siempre el último parámetro.

Estas operaciones son las únicas estrictamente necesarias; se puede definir cualquier otra en función de ellas. Sin embargo, resulta conveniente proporcionar también una serie de operadores auxiliares derivados, cuyo uso es muy habitual. Así pues, se definen:

$$\mathrm{Length} \quad : \quad (\sharp\alpha;\, (List\,\alpha),\, \uparrow Int)$$

[2]Cabe señalar que el cálculo-π polimórfico permite otras formulaciones de listas. Como curiosidad, admite incluso una versión *no recursiva*, puramente polimórfica, inspirada en la codificación de listas ideada por Church para el Sistema F de Girard-Reynolds. Los tipos de los constructores básicos son los siguientes:

$$
\begin{array}{rcl}
(List\,\alpha) & \triangleq & \uparrow(\sharp\beta;\, \beta,\, \uparrow(\alpha,\, \beta,\, \uparrow\beta),\, \uparrow\beta) \\
\mathrm{Nil} & : & (\sharp\alpha;\, \uparrow(List\,\alpha)) \\
\mathrm{Cons} & : & (\sharp\alpha;\, \alpha,\, (List\,\alpha),\, \uparrow(List\,\alpha))
\end{array}
$$

Son aún más complejas que las listas recursivas, y su uso no es práctico. Para obtener la codificación completa, se puede consultar el Capítulo 5 de la tesis de Turner [Tur96], donde fue originalmente planteada.

$$
\begin{array}{rcl}
\mathrm{Length}(l,t) & \triangleq & (\nu\, c,b)(\mathrm{Null}\langle\, l,b\,\rangle \mid b(r).\,\textbf{if}\ r\ \textbf{then}\ \overline{t}\langle\, 0\,\rangle \\
& & \quad \textbf{else}\ (\nu\, m)(\mathrm{Cdr}\langle\, l,m\,\rangle \\
& & \qquad \mid m(o).\,(\mathrm{Length}\langle\, o,c\,\rangle \mid c(z).\,\overline{t}\langle\, z+1\,\rangle)) \\
\mathrm{Nth} & : & (\sharp\alpha;\ \mathit{Int},\ (\mathit{List}\,\alpha),\ \uparrow\alpha) \\
\mathrm{Nth}(n,l,e) & \triangleq & \mathrm{Ith}\langle\, 1,n,l,e\,\rangle \\
\mathrm{Ith} & : & (\sharp\alpha;\ \mathit{Int},\ \mathit{Int},\ (\mathit{List}\,\alpha),\ \uparrow\alpha) \\
\mathrm{Ith}(i,n,l,e) & \triangleq & \textbf{if}\ (i=n)\ \textbf{then}\ \mathrm{Car}\langle\, l,e\,\rangle \\
& & \quad \textbf{else}\ (\nu\, m)(\mathrm{Cdr}\langle\, l,m\,\rangle \mid m(o).\,\mathrm{Ith}\langle\, i+1,n,o,e\,\rangle)
\end{array}
$$

Su significado es el evidente. La primera (Length) proporciona la longitud de una lista, en un momento dado, como un número entero. Por su parte, la segunda (Nth) es una función que devuelve el elemento n-ésimo, comenzando a contar por el último introducido. La tercera (Ith) es solamente un proceso auxiliar del anterior, y no será nunca usado directamente. Obsérvese que estas funciones permiten recorrer una lista como si se tratase de un *vector*. La definición del cálculo postula la existencia de vectores potencialmente infinitos, luego se dispone de ellos como primitivos; pero de no ser así, podrían ser fácilmente emulados mediante listas como éstas.

Por tanto, las últimas operaciones definidas tan sólo permiten utilizar una lista de manera diferente. Sin embargo, existen otras tareas, más habituales en el manejo normal del tipo, para las que sería igualmente beneficioso proporcionar operadores derivados, que evitasen la necesidad de un recorrido recursivo explícito. Concretamente, se considerarán dos de ellas:

$$
\begin{array}{rcl}
\mathrm{Era} & : & (\sharp\alpha;\ \alpha,\ (\mathit{List}\,\alpha),\ \uparrow(\mathit{List}\,\alpha)) \\
\mathrm{Era}(e,l,r) & \triangleq & (\nu\, d)(\mathrm{Car}\langle\, l,d\,\rangle \mid d(z).\,\textbf{if}\ (z=e)\ \textbf{then}\ \mathrm{Cdr}\langle\, l,r\,\rangle \\
& & \quad \textbf{else}\ (\nu\, m,p)(\mathrm{Cdr}\langle\, l,m\,\rangle \mid m(o).\,\mathrm{Era}\langle\, e,o,p\,\rangle \mid p(q).\,\mathrm{Cons}\langle\, z,q,r\,\rangle)) \\
\mathrm{Ein} & : & (\sharp\alpha;\ \alpha,\ (\mathit{List}\,\alpha),\ \uparrow\mathit{Bool}) \\
\mathrm{Ein}(e,l,b) & \triangleq & (\nu\, b)(\mathrm{Null}\langle\, l,b\,\rangle \mid b(r).\,\textbf{if}\ r\ \textbf{then}\ \overline{b}\langle\, \textbf{false}\,\rangle \\
& & \quad \textbf{else}\ (\nu\, d,m)(\mathrm{Car}\langle\, l,d\,\rangle \mid d(s).\,\textbf{if}\ (s=e)\ \textbf{then}\ \overline{b}\langle\, \textbf{true}\,\rangle \\
& & \quad \textbf{else}\ (\nu\, o)(\mathrm{Cdr}\langle\, l,o\,\rangle \mid o(p).\,\mathrm{Ein}\langle\, e,p,b\,\rangle)))
\end{array}
$$

La primera (Era) borra un elemento e de una lista l, asumiendo que dicho elemento está en la lista; como de costumbre, devuelve la lista resultante en r. La segunda (Ein) es un test booleano que comprueba precisamente la condición de pertenencia; es decir, si un elemento está o no en la lista. Curiosamente, es ligeramente más compleja que la anterior, puesto que tiene que considerar el caso especial de la lista vacía.

Se podrían incorporar muchas más definiciones, como varios tipos de iteradores o una función de concatenación. Sin embargo, y aunque algunas de ellas serán de hecho utilizadas, no parece necesario incluirlas en el vocabulario de partida. Así pues, serán definidas como procesos auxiliares allí donde sea preciso.

Listas Nombradas

Posiblemente, la principal utilidad de las listas deriva de su capacidad para actuar como el equivalente *dinámico* de otras estructuras de datos más rígidas. Así, en su forma normal, una

lista puede compararse a una tupla de longitud variable. Del mismo modo, y como ya se ha dicho, una vez se han definido los procesos Length y Nth puede también utilizarse como el análogo de un vector. Dada la flexibilidad que proporciona el polimorfismo, se pueden establecer otras analogías similares, e incluso potenciarlas mediante la definición de variantes adecuadas.

La más habitual de estas variantes es, con toda probabilidad, la denominada *lista nombrada* o *named list*. Se podría describir de manera aproximada como un registro de longitud variable, o mejor aún como un vector asociativo. Esto significa, en resumen, que se tiene una secuencia de datos, todos del mismo tipo, en el que cada uno tiene asignada una etiqueta que lo distingue del resto. Es obvio que si el tamaño de la lista es desconocido *a priori*, también ha de serlo la secuencia de etiquetas de sus elementos. Así pues, dichas etiquetas también deben ser proporcionadas como variables, para cuya definición se proporciona un tipo abstracto Label. Dada su función y la forma en que habrán de manipularse, parece lógico implementarlas mediante cadenas; por ello, a partir de este momento ambos se utilizarán como sinónimos.

Una lista nombrada será pues simplemente una lista, cuyos elementos serán los pares formados por una etiqueta –una cadena– y un dato del tipo base –polimórfico– correspondiente. Dichos pares se representan de la manera más cómoda como un registro de dos campos, la etiqueta (l) y el dato (d). En resumen, pues, una lista nombrada se implementará como una lista de registros de esa forma. Teniendo en cuenta todo lo dicho, se definen a continuación los tipos necesarios para describir esta estructura, que son:

$$\begin{array}{rcl} \mathit{Label} & \triangleq & \mathit{String} \\ (\mathit{Named}\,\alpha) & \triangleq & (\dagger l\colon \mathit{Label},\ \dagger d\colon \alpha) \\ (\mathit{NList}\,\alpha) & \triangleq & (\mathit{List}\,(\mathit{Named}\,\alpha)) \end{array}$$

Debe tenerse en cuenta que una lista nombrada no es solamente una variante del tipo lista, ya que en todo momento sigue siendo *también* una lista convencional. Puede verse, en efecto, que se tiene la equivalencia $(\mathit{NList}\,\alpha) \approx (\mathit{List}\,\beta)$ sin más que considerar a su vez que $\beta \approx (\mathit{Named}\ \alpha)$. Esto implica que todas las operaciones genéricas ya descritas son igualmente aplicables a una lista nombrada. No obstante, será también conveniente definir una serie de operaciones especiales para este tipo, que saquen ventaja de sus características particulares.

Algunas de las operaciones clásicas (Nil, Null, Cdr) se mantienen sin cambio alguno: no sólo se pueden utilizar sobre una lista nombrada, sino que su definición es de hecho la más adecuada para este tipo específico. Sin embargo, conviene considerar ligeras modificaciones de las restantes.

$$\begin{array}{rcl} \mathrm{NCons} & : & (\sharp\alpha;\ \mathit{Label},\ \alpha,\ (\mathit{NList}\,\alpha),\ \uparrow(\mathit{NList}\,\alpha)) \\ \mathrm{NCons}(t,e,l,r) & \triangleq & \mathrm{Cons}\langle\,(\dagger l \prec t, \dagger d \prec e), l, r\,\rangle \\ \mathrm{NCar} & : & (\sharp\alpha;\ (\mathit{NList}\,\alpha),\ \uparrow\alpha) \\ \mathrm{NCar}(l,e) & \triangleq & (\nu\, d)(\mathrm{Car}\langle\, l, d\,\rangle \mid d(z).\,\overline{e}\langle\, z\dagger d\,\rangle) \\ \mathrm{LCar} & : & (\sharp\alpha;\ (\mathit{NList}\,\alpha),\ \uparrow\mathit{Label}) \\ \mathrm{LCar}(l,t) & \triangleq & (\nu\, d)(\mathrm{Car}\langle\, l, d\,\rangle \mid d(z).\,\overline{e}\langle\, z\dagger t\,\rangle) \end{array}$$

Se trata de la operación análoga al constructor, y de dos formas de enfocar la proyección del primer elemento. La primera (Ncons) permite disciplinar la elaboración de listas nombradas.

Parte de un dato y su etiqueta, y los incorpora como el primer elemento de una lista preexistente. La segunda (NCar) es similar a Car; tras obtener el primer elemento, descompone el registro y devuelve como resultado el dato contenido. Por el contrario, la tercera (LCar) hace lo mismo, pero devuelve la *etiqueta* en lugar del dato, una manipulación que aquí será fundamental.

Pero, sin duda, la operación más importante es la siguiente. En realidad, es la que da sentido a estas listas, y la que permite usarlas como un registro extensible:

$$\begin{array}{rcl}
\mathrm{NDot} & : & (\sharp\alpha;\ \mathit{Label},\ (\mathit{NList}\,\alpha),\ \uparrow\alpha) \\
\mathrm{NDot}(t,l,e) & \triangleq & (\nu\, d)(\mathrm{Car}\langle\, l,d\,\rangle \,|\, d(z).\,\mathbf{if}\ \ (z\dagger l = t)\ \ \mathbf{then}\ \ \overline{e}\langle\, z\dagger d\,\rangle \\
& & \quad \mathbf{else}\ \ (\nu\, m)(\mathrm{Cdr}\langle\, l,m\,\rangle \,|\, m(o).\,\mathrm{NDot}\langle\, t,o,e\,\rangle))
\end{array}$$

Se trata, obviamente, de un *selector*. Dada una lista nombrada y una etiqueta, localiza el elemento marcado con dicha etiqueta y devuelve como resultado el *dato* correspondiente. La definición del proceso supone que dicho elemento existe en la lista; más adelante se proporciona un operador que permite comprobar previamente la condición de pertenencia.

Por su parte, los operadores que emulan el comportamiento de un vector apenas sufren cambios. No es necesario alterar la función de longitud (Length), que puede usarse sin alteraciones; y solamente la selección por posición experimenta un ligero cambio:

$$\begin{array}{rcl}
\mathrm{Nnth} & : & (\sharp\alpha;\ \mathit{Int},\ (\mathit{NList}\,\alpha),\ \uparrow\alpha) \\
\mathrm{Nnth}(n,l,e) & \triangleq & \mathrm{Nith}\langle\, 1,n,l,e\,\rangle \\
\mathrm{Nith} & : & (\sharp\alpha;\ \mathit{Int},\ \mathit{Int},\ (\mathit{NList}\,\alpha),\ \uparrow\alpha) \\
\mathrm{Nith}(i,n,l,e) & \triangleq & \mathbf{if}\ \ (i = n)\ \ \mathbf{then}\ \ \mathrm{NCar}\langle\, l,e\,\rangle \\
& & \quad \mathbf{else}\ \ (\nu\, m)(\mathrm{Cdr}\langle\, l,m\,\rangle \,|\, m(o).\,\mathrm{Nith}\langle\, i+1,n,o,e\,\rangle)
\end{array}$$

Obsérvese que el único cambio introducido en Nnth es similar al que se hizo en NCar: se modifica el proceso de modo que sólo devuelva la parte relevante de cada elemento.

Sin embargo, los operadores de borrado y búsqueda dependen totalmente de la naturaleza de una lista nombrada, y por tanto los dos campos resultan ahora significativos. Por tanto, cada uno de ellos adopta las dos variantes, de modo que se tiene:

$$\begin{array}{rcl}
\mathrm{NEra} & : & (\sharp\alpha;\ \alpha,\ (\mathit{NList}\,\alpha),\ \uparrow(\mathit{NList}\,\alpha)) \\
\mathrm{NEra}(e,l,r) & \triangleq & (\nu\, d)(\ \mathrm{Car}\langle\, l,d\,\rangle \mid d(z).\,\mathbf{if}\ \ (z\dagger d = e)\ \ \mathbf{then}\ \ \mathrm{Cdr}\langle\, l,r\,\rangle \\
& & \quad \mathbf{else}\ \ (\nu\, m,p)(\ \mathrm{Cdr}\langle\, l,m\,\rangle \mid m(o).\,\mathrm{NEra}\langle\, e,o,p\,\rangle \\
& & \qquad \mid p(q).\,\mathrm{Cons}\langle\, z,q,r\,\rangle\ )) \\
\mathrm{LEra} & : & (\sharp\alpha;\ \mathit{Label},\ (\mathit{NList}\,\alpha),\ \uparrow(\mathit{NList}\,\alpha)) \\
\mathrm{LEra}(t,l,r) & \triangleq & (\nu\, d)(\mathrm{Car}\langle\, l,d\,\rangle \,|\, d(z).\,\mathbf{if}\ \ (z\dagger l = t)\ \ \mathbf{then}\ \ \mathrm{Cdr}\langle\, l,r\,\rangle \\
& & \quad \mathbf{else}\ \ (\nu\, m,p)(\mathrm{Cdr}\langle\, l,m\,\rangle \,|\, m(o).\,\mathrm{LEra}\langle\, t,o,p\,\rangle \\
& & \qquad |\, p(q).\,\mathrm{Cons}\langle\, z,q,r\,\rangle)) \\
\mathrm{NEin} & : & (\sharp\alpha;\ \alpha,\ (\mathit{List}\,\alpha),\ \uparrow\mathit{Bool}) \\
\mathrm{NEin}(e,l,b) & \triangleq & (\nu\, b)(\mathrm{Null}\langle\, l,b\,\rangle \,|\, b(r).\,\mathbf{if}\ \ r\ \ \mathbf{then}\ \ \overline{b}\langle\, \mathbf{false}\,\rangle \\
& & \quad \mathbf{else}\ \ (\nu\, d,m)(\mathrm{NCar}\langle\, l,d\,\rangle \,|\, d(s).\,\mathbf{if}\ \ (s = e)\ \ \mathbf{then}\ \ \overline{b}\langle\, \mathbf{true}\,\rangle
\end{array}$$

$$
\begin{array}{rcl}
 & & \textbf{else}\ (\nu\, o)(\mathrm{Cdr}\langle\, l, o\,\rangle \mid o(p).\, \mathrm{NEin}\langle\, e, p, b\,\rangle))) \\
\mathrm{LEin} & : & (\sharp\alpha;\ \mathit{Label},\ (\mathit{List}\,\alpha),\ \uparrow\mathit{Bool}) \\
\mathrm{LEin}(t, l, b) & \triangleq & (\nu\, b)(\mathrm{Null}\langle\, l, b\,\rangle \mid b(r).\, \textbf{if}\ \ r\ \ \textbf{then}\ \ \overline{b}\langle\, \textbf{false}\,\rangle \\
 & & \textbf{else}\ \ (\nu\, d, m)(\mathrm{LCar}\langle\, l, d\,\rangle \mid d(s).\, \textbf{if}\ \ (s = t)\ \ \textbf{then}\ \ \overline{b}\langle\, \textbf{true}\,\rangle \\
 & & \textbf{else}\ \ (\nu\, o)(\mathrm{Cdr}\langle\, l, o\,\rangle \mid o(p).\, \mathrm{NEin}\langle\, t, p, b\,\rangle)))
\end{array}
$$

Su significado y función resultan de nuevo obvios. El primero (NEra) es el análogo a Era, y realiza un borrado según el contenido. Es decir, localiza el *primer* elemento cuyo dato coincida con el proporcionado, y lo elimina. Sin embargo, será mucho más frecuente el segundo (LEra), que realiza un borrado según la etiqueta, asumiendo de nuevo que ésta existe. En el caso de que esté duplicada, algo que no debería suceder, borraría simplemente su primera ocurrencia. Los otros dos operadores incluyen modificaciones análogas: NEin realiza una búsqueda por contenido, mientras que LEin comprueba la existencia de una etiqueta determinada. Este último test será esencial en la manipulación normal de las listas nombradas, como ya ha podido verse.

Notación Abreviada para Listas

Aunque la definición de todos estos procesos permite la manipulación de las listas, es evidente que ésta sigue tomando una forma excesivamente compleja, debido al estilo –un esquema de paso de continuación o CPS, como se ha dicho en el Capítulo 6– provocado por el uso del cálculo-π. Debido a esto, se pierde en buena medida la similitud con la sintaxis de Lisp, y con ella la flexibilidad que se pretendía proporcionar al elaborarlos.

Problemas similares han surgido siempre que se ha pretendido usar el cálculo-π como fundamento de un lenguaje de programación. Todos los autores que han afrontado esta tarea [ALSN00, Bou98, FGL+96, PT00, Vas01] han adoptado finalmente un compromiso sintáctico, el mismo que es habitual en las semánticas fundamentadas en el cálculo-λ con estilo CPS [Plo75b]. Éste se basa en la ocultación del término de continuación, mediante la definición sintáctica de los denominados *valores complejos*. Esto significa, básicamente, que se proporciona una sintaxis especial para los procesos funcionales, que acentúa su parecido con las funciones clásicas.

El objetivo es eliminar la manipulación explícita del *canal de respuesta*, que es de hecho el causante de este problema. Como ha podido verse en las definiciones previas, este canal ejerce el mismo papel que una *continuación*: se crea de manera explícita para cada invocación, y se sitúa como un prefijo de entrada en paralelo al proceso invocado. De este modo, no sólo recibe el valor calculado, sino que indica en qué punto ha de seguir la computación.

Eliminar este canal de las expresiones resulta sorprendentemente sencillo. Se debe suprimir toda mención explícita al mismo, luego desaparece tanto el término que lo crea como el que recibe el dato devuelto. También se elimina, y esto es fundamental, de la propia invocación del proceso; esto implica que la llamada se hará con *un parámetro menos* de los definidos. Precisamente ésta es la razón por la que se adoptó la convención de que el canal de respuesta fuera siempre el último: de este modo, se puede asumir que es éste el que ha sido apocopado, y se evita la confusión que, de otro modo, podría haberse producido.

Evidentemente, en circunstancias normales la ausencia de un parámetro se considera como un error de tipo. Para evitarlo, este tipo especial de invocación de proceso se realizará con una

sintaxis alterada, inspirada de nuevo en la notación abreviada de Church para el cálculo-λ. Por tanto, la llamada ya no se indica del modo convencional, con los parámetros encerrados entre ángulos y separados por comas, como en el término $P\langle a, b, c\rangle$; sino entre paréntesis, incluyendo el propio nombre de proceso, y separando los parámetros mediante espacios. El término anterior toma pues la forma $(P\, a\, b)$, donde el último argumento ha sido ya omitido.

Esto sólo nos deja con otro problema: al no haber ya un canal donde recibir la respuesta, y carecer el cálculo de asignaciones, no existe tampoco la variable d donde se recibía el resultado. Esto se soluciona utilizando en su lugar la propia invocación del proceso: al igual que en una función convencional, se asume que ésta, cuando se emplea con la notación abreviada, simboliza a su propio resultado. Se dice entonces que la invocación actúa como un *valor complejo*. De hecho, representa *siempre* a un dato, y sólo puede usarse como tal. No puede aparecer, por tanto, como un término independiente, sino que siempre debe ocupar la posición sintácticamente reservada a un valor. En definitiva, la de un argumento en la invocación de otro proceso.

En resumen: en el cálculo puro la invocación de un proceso funcional P sigue el esquema siguiente: $(\nu\, c)(\, P\langle ..., c\rangle \mid c(d).Q\langle d, ...\rangle\,)$, donde Q puede ser cualquier expresión-π que utilice el dato (d) calculado por P. Con la nueva sintaxis, esta misma fórmula se expresará como $Q\langle\,(P\, ...), ...\rangle$, donde el canal de respuesta ha desaparecido por completo, y la invocación se utiliza como un valor complejo allí donde se usaba el dato d, que también desaparece.

Se puede pensar que esta sintaxis resulta conflictiva, en especial porque *parece* una expresión de orden superior, en la que un proceso se usa como parámetro de la invocación de otro. En realidad, no plantea problema alguno, ya que sólo se envía el resultado del proceso; y menos aún en el cálculo-π, que en todo caso equivale a su propia versión de orden superior. Además, implícitamente se adoptando el mismo compromiso en todas las operaciones definidas para booleanos, números y cadenas, que también mantienen la notación funcional clásica.

$$\left[\begin{array}{lcl}
\mathrm{Length}(l,t) & \triangleq & \textbf{if } (\texttt{null}\ l) \textbf{ then } \overline{t}\langle\, 0\,\rangle \textbf{ else } \overline{t}\langle\,(\texttt{length}\ (\texttt{cdr}\ l)) + 1\,\rangle \\
\mathrm{Era}(e,l,r) & \triangleq & \textbf{if } ((\texttt{car}\ l) = e) \textbf{ then } \overline{r}\langle\,(\texttt{cdr}\ l)\,\rangle \\
 & & \quad \textbf{else } \overline{r}\langle\,(\texttt{cons}\ (\texttt{car}\ l)\ (\texttt{era}\ e\ (\texttt{cdr}\ l)))\,\rangle \\
\mathrm{Ein}(e,l,b) & \triangleq & \textbf{if } (\texttt{null}\ l) \textbf{ then } \overline{b}\langle\, false\,\rangle \textbf{ else} \\
 & & \textbf{if } ((\texttt{car}\ l) = s) \textbf{ then } \overline{b}\langle\, true\,\rangle \textbf{ else } \overline{b}\langle\,(\texttt{ein}\ e\ (\texttt{cdr}\ l))\,\rangle \\
\mathrm{Nth}(n,l,e) & \triangleq & \mathrm{Ith}\langle\, 1, n, l, e\,\rangle \\
\mathrm{Ith}(i,n,l,e) & \triangleq & \textbf{if } (i = n) \textbf{ then } \overline{e}\langle\,(\texttt{car}\ l)\,\rangle \textbf{ else } \mathrm{Ith}\langle\, i+1, (\texttt{cdr}\ l), e\,\rangle
\end{array}\right]$$

Tabla C.1: Operaciones de Listas: Definición Abreviada.

Se ha preferido limitar el alcance de esta sintaxis a los procesos definidos para *listas*, normales o nombradas. Esto significa que la *notación abreviada* sólo será aplicada a la veintena de funciones descritas en este Apéndice, y cualquier otro proceso habrá de utilizar el estilo-CPS convencional. La razón es que sólo se pretende facilitar el uso de listas, y no alterar la filosofía de especificación del cálculo-π. En lenguajes como Pict, que pretende establecer una correlación directa con el cálculo-λ, este tipo de transformación es permitida para todo proceso funcional [Pie98]. Sin embargo, parece claro que hacer lo mismo en esta versión del cálculo sólo provocaría confusión; es por esto que su uso se restringe exclusivamente a listas.

La acumulación de paréntesis, junto a la mezcla de las dos notaciones, puede resultar desconcertante. Para evitar este efecto, se indicará el nombre del proceso invocado con un tipo de letra diferente –`courier`– y en minúscula, de modo que se facilite la identificación visual inmediata del uso de la notación abreviada.

$$
\left[\begin{array}{lcl}
\mathrm{NCons}(t,e,l,r) & \triangleq & (\texttt{cons}\ (\dagger l \prec t, \dagger d \prec e)\ l) \\
\mathrm{NCar}(l,e) & \triangleq & (\nu\, d)\overline{d}\langle\, (\texttt{car}\ l)\,\rangle.\, d(z).\, \overline{e}\langle\, z\dagger d\,\rangle \\
\mathrm{LCar}(l,e) & \triangleq & (\nu\, d)\overline{d}\langle\, (\texttt{car}\ l)\,\rangle.\, d(z).\, \overline{e}\langle\, z\dagger t\,\rangle \\
\mathrm{NDot}(t,l,e) & \triangleq & \textbf{if}\ ((\texttt{lcar}\ l) = t)\ \textbf{then}\ \overline{e}\langle\, (\texttt{ncar}\ l)\,\rangle\ \textbf{else}\ (\texttt{ndot}\ t\ (\texttt{cdr}\ l)\ e) \\
\mathrm{Nnth}(n,l,e) & \triangleq & \mathrm{Nith}\langle\, 1, n, l, e\,\rangle \\
\mathrm{Nith}(i,n,l,e) & \triangleq & \textbf{if}\ (i = n)\ \textbf{then}\ \overline{e}\langle\, (\texttt{ncar}\ l)\,\rangle\ \textbf{else}\ \mathrm{Nith}\langle\, i+1, (\texttt{cdr}\ l), e\,\rangle \\
\mathrm{NEra}(e,l,r) & \triangleq & \textbf{if}\ ((\texttt{ncar}\ l) = e)\ \textbf{then}\ \overline{r}\langle\, (\texttt{cdr}\ l)\,\rangle \\
 & & \ \textbf{else}\ \overline{r}\langle\, (\texttt{cons}\ (\texttt{car}\ l)\ (\texttt{nera}\ e\ (\texttt{cdr}\ l)))\,\rangle \\
\mathrm{LEra}(t,l,r) & \triangleq & \textbf{if}\ ((\texttt{lcar}\ l) = t)\ \textbf{then}\ \overline{r}\langle\, (\texttt{cdr}\ l)\,\rangle \\
 & & \ \textbf{else}\ \overline{r}\langle\, (\texttt{cons}\ (\texttt{car}\ l)\ (\texttt{lera}\ t\ (\texttt{cdr}\ l)))\,\rangle \\
\mathrm{NEin}(e,l,b) & \triangleq & \textbf{if}\ (\texttt{null}\ l)\ \textbf{then}\ \overline{b}\langle\, \mathit{false}\,\rangle\ \textbf{else} \\
 & & \textbf{if}\ ((\texttt{ncar}\ l) = e)\ \textbf{then}\ \overline{b}\langle\, \mathit{true}\,\rangle\ \textbf{else}\ \overline{b}\langle\, (\texttt{nein}\ e\ (\texttt{cdr}\ l))\,\rangle \\
\mathrm{LEin}(t,l,b) & \triangleq & \textbf{if}\ (\texttt{null}\ l)\ \textbf{then}\ \overline{b}\langle\, \mathit{false}\,\rangle\ \textbf{else} \\
 & & \textbf{if}\ ((\texttt{lcar}\ l) = t)\ \textbf{then}\ \overline{b}\langle\, \mathit{true}\,\rangle\ \textbf{else}\ \overline{b}\langle\, (\texttt{lein}\ t\ (\texttt{cdr}\ l))\,\rangle
\end{array}\right]
$$

Tabla C.2: Operaciones de Listas Nombradas: Definición Abreviada.

La longitud de las expresiones-π que manipulan listas se acorta de manera significativa con el uso de la notación abreviada. Esto puede verse con claridad, por ejemplo, al aplicarla sobre la propia definición de las funciones ya conocidas; algunas reducen su extensión a casi la mitad, lo que es notable al tratarse de fórmulas breves. Para ilustrar este detalle, en las Tablas C.1 y C.2 se muestran las definiciones abreviadas de los operadores de listas y de listas nombradas, respectivamente. No se han incluido los procesos Nil, Cons, Null, Car y Cdr, dado que no sufren modificación alguna. Tampoco se indican los tipos, pues obviamente siguen siendo los mismos; es cierto que el canal de retorno ya no resulta visible, pero sigue estando presente.

Para finalizar, se añadirá un último detalle a la sintaxis. Como se recordará, se pretendía que las listas nombradas resultasen análogas a un registro dinámico. No obstante, aunque la notación abreviada hace mucho más flexible el uso de la función NDot, probablemente no resulta aún lo bastante cómoda. Con el objetivo de acentuar aún más el parecido con un registro, se introduce la equivalencia siguiente:

$$l \bullet t \;\doteq\; (\texttt{ndot}\ t\ l)$$

No tiene relevancia alguna, ya que se trata solamente de azúcar sintáctico, pero incrementa de manera notable la legibilidad de las expresiones complejas, y hace sus combinaciones mucho más flexibles, en especial cuando se relaciona con un registro (†).

C.5 Notación de Tipos

En esta sección tan sólo se pretende fijar la notación que se emplea en las asignaciones de tipos de los procesos. Básicamente es una notación estándar, con muy pocas adiciones especiales; no obstante, distintos autores han utilizado símbolos diferentes para el mismo concepto, por lo que conviene aclararla para evitar todo tipo de ambigüedades.

Existen dos tipos de ecuaciones de tipo: *definiciones* y *asignaciones*. Las primeras describen la estructura de un tipo compuesto, y utilizan el conector de definición $\triangleq$ como separador. Por su parte, las segundas simplemente indican el tipo que tiene asignado un determinado canal o proceso, y se marcan mediante los dos puntos (:) de la manera habitual.

En lugar de proporcionar un tipo unidad explícito, el cálculo tiene como tipo básico la tupla vacía, que se define simplemente como:

$$No \triangleq ()$$

Representa una comunicación en la que no se intercambia información alguna. No se utiliza, obviamente, nunca: pero sobre él se pueden definir toda una serie de derivados, y en particular las sincronizaciones puras. Al mismo nivel que éste se sitúan los otros tipos primitivos, ya conocidos: Int, Bool y String. Todos ellos pueden construirse a partir del anterior, como ya se ha comentado; sin embargo, se considerarán igualmente básicos.

El tipo derivado fundamental es el *tipo de canal*. Indica la clase de información que se puede enviar a través del canal considerado. Se puede aplicar a todo tipo, lo que incluye a otros tipos-canal; y puede dar lugar a estructuras muy complejas, con un nivel de anidamiento indefinido. La más importante, sin embargo, es también la más sencilla:

$$Sync \triangleq \uparrow() = \uparrow No$$

Describe una sincronización pura o *señal*, esto es, el tipo más básico de comunicación. En ciertas álgebras es el único permitido, e incluso en este cálculo su papel resulta fundamental. Se utiliza con relativa frecuencia en la especificación de la semántica. Como ha podido verse, un tipo-canal se denota como el símbolo $\uparrow$, seguido de la expresión que indica la clase de valor que transmite. En este particular se sigue la notación de Turner [Tur96], cercana a las propuestas de Pierce [PS00], por considerarla más clara que las de otros autores[3].

Otro tipo derivado básico es un *tipo de tupla*, o más estrictamente un *tipo producto*. Existen varias posibilidades para representarlo, pero se ha optado por seguir aquí también la notación de Turner. Por tanto, se expresa como la secuencia de los tipos de los elementos de la tupla, separados por comas y encerrados entre paréntesis; esta forma resulta totalmente intuitiva, y se puede combinar con la anterior para obtener resultados complejos, como la definición teórica de un booleano, descrita de manera implícita en la sección C.2.1:

$$Boolean \triangleq \uparrow(\uparrow(), \uparrow()) = \uparrow(Sync, Sync)$$

Asociado al tipo producto está el *tipo registro*. La notación clásica para este caso es similar a la del tipo producto, pero indicando para cada campo la etiqueta que lo designa, usualmente

[3] Sangiorgi y Walker, por ejemplo, utilizan el símbolo $\sharp$, que aquí recibirá un uso muy diferente.

marcada con el signo de igualdad. Aquí se ha preferido emplear una notación más cercana a la sintaxis definida en para los registros (§C.4.2), para mayor claridad. Por tanto, en el descriptor de tipo cada etiqueta va precedida del signo †, y seguida de dos puntos (:) y el tipo de su campo. El ejemplo más sencillo es el tipo de un par ($Par\ \alpha\ \beta$), ya conocido, cuya definición se incluye, siguiendo este formato, en la página 383.

Finalmente, se proporciona también una notación abreviada para los vectores. Éstos son tuplas homogéneas, y por tanto se definen como un tipo-producto. No obstante, ocasionalmente se desea abstraer su tamaño, o más bien dejarlo indefinido; en tal caso se describe únicamente como una secuencia de elementos de un mismo tipo. Esto se denota indicando el tipo-elemento, precedido del símbolo $\sim$. Esta representación no es estrictamente ortodoxa, pero resulta sin embargo bastante habitual. En general, las definiciones que implican vectores se usan con bastante libertad, especialmente cuando codifican una estructura [Mil89, Wal95, ZP97].

Con esto se han presentado ya todos los tipos convencionales. Sólo faltan por comentar los más avanzados, esto es: los *recursivos*, los *polimórficos* o de segundo orden, y el tipo *dinámico*, que será el más sencillo y se describirá en último lugar.

Los *tipos recursivos* se representarán igual que en el cálculo-λ, con una notación inspirada en la noción de punto fijo. Toda la definición quedará parametrizada por una variable de tipo, marcada con el símbolo μ, que representa al propio tipo definido. Luego se describe normalmente la estructura, utilizando con libertad la variable-μ. Obviamente, cada una de sus ocurrencias introduce una recursión. El sistema admite tipos *mutuamente recursivos*, pero no los señala de manera explícita: simplemente permite que dos tipos derivados se utilicen mutuamente. Esto no es muy habitual en las implementaciones, pero no supone ningún problema desde el punto de vista teórico, ya que pueden transformarse fácilmente en una expresión-μ sin más que desplegar (*unfold*) una vez el punto fijo, del mismo modo que en el cálculo-λ.

Por su parte, los tipos *polimórficos* se expresan como tipos *paramétricos*, ligados mediante variables de tipo; de modo que los términos asignados a dichas variables pueden tomar tipos diferentes en instanciaciones diferentes. Equivalen, pues, a las variables-$\exists$ del cálculo-λ de segundo orden [Car97], dado que el estilo del cálculo-π tiende con naturalidad hacia una cuantificación existencial. Por eso, los tipos paramétricos se representan como en el cálculo-λ, es decir, con el nombre de tipo y sus argumentos encerrados entre paréntesis y separados por espacios. Así, por ejemplo, el tipo de un puerto se define como ($TPort\ \alpha\ \beta$).

Dentro de la definición, las variables de tipo se indican siempre como los parámetros iniciales, habitualmente empleando letras griegas, separadas del resto mediante un punto y coma, y marcadas con el signo de almohadilla ($\sharp$). Después pueden usarse, como un tipo más, tantas veces como se quiera, y ya sin marcar. Se sigue por tanto la notación original de Turner [Tur96], pero ampliada del mismo modo que en Pict [PT00].

Por último, el tipo dinámico *Dyn* no requiere excesivos comentarios. Sigue la misma filosofía y notación que en su definición original [ACPP91], de modo que supone la única ampliación dinámica de un sistema de tipos estático. En definitiva, representa a *cualquier* tipo existente; pero a diferencia de un tipo polimórfico, no supone ligadura alguna. Por tanto, distintas ocurrencias pueden hacer referencia a tipos distintos, incluso dentro de la misma expresión. La resolución se hará explícitamente en tiempo de ejecución mediante un operador `typecase`; pero este detalle no será puesto de relieve en ningún otro lugar de esta tesis, ya que lo realmente importante es señalar los lugares donde la información de tipado se considera irrelevante. Su incorporación a

los sistemas de tipos de las álgebras de procesos, aunque sencilla, es muy reciente. Fue introducido en el cálculo-π nomádico por Paweł T. Wojciechowski, con un propósito prácticamente idéntico al que tiene en la semántica de $\mathcal{P}i\mathcal{L}ar$.

En esta tesis, todo proceso complejo va acompañado de su correspondiente anotación de tipos. Como ya se ha comentado, el tipo de todos los canales y subprocesos implicados es inmediatamente deducible. De hecho, todas las asignaciones de tipos son consistentes entre sí; algunas son particularmente complejas, en especial aquellas que definen estructuras mutuamente recursivas. Pero todas ellas son completamente coherentes con la definición teórica.

Ocasionalmente, el tipo no se indica de manera expresa en definiciones muy sencillas. Esto no significa que se esté usando una versión no tipada del cálculo, sino simplemente que esta información se considera obvia, y se prefiere omitirla por comodidad e incluso claridad. En resumen, se asume que toda expresión-$\hat{\pi}$ en este documento está correctamente tipada. Por esto, y por lo dicho más arriba, se puede afirmar que el sistema –la semántica de $\mathcal{P}i\mathcal{L}ar$– está completamente *bien tipado* según las reglas del cálculo.

BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Apéndice D

Definiciones Semánticas Auxiliares

A lo largo del Capítulo 7, durante la descripción formal de la semántica de $\mathcal{P}i\mathcal{L}ar$, aparecen con bastante frecuencia ciertas construcciones y esquemas comunes, más relacionadas con el modo en que se usa habitualmente el cálculo-π que con el sistema concreto descrito.

Dado que todas ellas pueden definirse independientemente de dicho sistema, se ha decidido agruparlas en este Apéndice. De este modo no sólo se reduce la longitud de la especificación semántica, ya de por sí bastante extensa, sino que además resulta simplificada, al centrarse exclusivamente en los aspectos relevantes. No obstante, ha de señalarse que existe también un conjunto de operaciones, en parte similares a éstas, pero mucho más específicas. Dado que no admiten una exposición coherente fuera de su contexto, se ha preferido mantenerlas en él; se incluyen, por tanto, como parte de la propia especificación.

La mayoría de las operaciones incluidas aquí se aplica sobre vectores de nombres o, especialmente, estructuras de tipo lista (véase al respecto el Apéndice C), por lo que su esquema general será bastante similar. No se ha considerado, sin embargo, que sean lo suficientemente básicas o habituales como para ser proporcionadas como primitivas. Es por esto que no se incorporan directamente en el dialecto del cálculo (§C.4.4), sino se conciben más bien como fragmentos autónomos de la definición de la semántica.

Las operaciones se han agrupado en tres categorías: una serie de iteradores que facilitan el establecimiento de bucles, una pareja de operaciones simples de manipulación de listas, y un conjunto de utilidades de conversión entre vectores y listas.

D.1 Bucles: Iteradores

El cálculo-π, como cualquier álgebra de procesos, es un formalismo esencialmente recursivo [BFP01]. Es cierto que, además de la recursión, se proporcionan otros mecanismos de *repetición* con potencia similar, tales como la replicación (**!**) o el punto fijo[1], pero ésta sigue siendo la construcción

[1]La presentación habitual del cálculo-π asume la definición de constantes recursivas como una noción básica [Mil99] de congruencia estructural. Por otra parte, este mismo concepto se justificaba en Ccs [Mil89] como una abreviatura para el operador de punto fijo ($\mathbf{fix}\,\widetilde{X}.P$). Aunque éste no es muy habitual en el contexto del cálculo-π, tanto el razonamiento como la sintaxis son perfectamente compatibles, y autores como Parrow [Par01] lo consideran sin cambios significativos. Además, la relación entre este operador y el de replicación es bastante

más natural, así como la más flexible y genérica. No obstante, aparecen con cierta frecuencia esquemas de comportamiento que muestran un patrón eminentemente *iterativo*, especialmente durante el tratamiento de estructuras complejas, como vectores o listas. Obviamente, y como ya se ha visto (§C.2.1, C.4.4), estos esquemas pueden definirse de manera recursiva. Sin embargo, resulta conveniente plantear una estrategia general para su desarrollo, simplificando de este modo tanto su especificación como su comprensión.

El esquema básico de esta estrategia supone que se está actuando sobre una estructura compleja. Normalmente, los elementos de esta estructura se recorren siguiendo un mismo patrón; lo único que diferencia a dos procesos de este tipo es el tratamiento concreto que se aplicará en cada caso. El objetivo es, por tanto, separar este tratamiento específico del patrón de recorrido. Para ello, se expresa este último mediante un proceso *iterador* predeterminado, diseñado de manera genérica para cada tipo de estructura. Junto a éste, se define para cada situación un proceso particular, relativo al problema en cuestión. La composición de ambos define un esquema iterativo, comparable a un *bucle* secuencial clásico.

El aspecto fundamental aquí es, sin duda, la definición del proceso específico, que actúa como *cuerpo* del bucle. Ha de seguir siempre el mismo esquema, que se resume como sigue:

$$\text{Cuerpo}_F(\widetilde{par}, vcb, end) \triangleq (\, vcb(x) \ldots \text{Cuerpo}_F\langle \widetilde{par}, vcb, end\rangle + end.\mathbf{0}\,)$$

Se trata, por tanto, de un esquema recursivo; pero diseñado de modo que cada definición se concentra en los detalles propios del tratamiento, mientras que la recursión en sí queda en un segundo plano. La idea es que este proceso –el cuerpo del bucle– describe solamente las operaciones a realizar sobre *cada elemento* de la estructura. Se asume que este elemento se recibe a través de un canal específico (vcb) y se manipula de la manera que se indique. A continuación se produce una nueva llamada recursiva, de modo que el proceso queda listo para recibir un nuevo dato. La iteración termina cuando se recibe una señal especial (end), que se define como una sincronización pura. Con este fin, el esquema se define como una selección entre la recepción de esta señal y el subproceso recursivo que opera sobre cada dato.

Todo proceso planteado como el cuerpo de un bucle utilizará un nombre marcado con el subíndice ($_F$), de modo que se distinga claramente del resto. Su interfaz constará de los dos canales mencionados (vcb, end), además de los parámetros propios de cada problema. Estos canales son, obviamente, los que le relacionan con el iterador correspondiente.

Por su parte, es claro que la interfaz del iterador ha de incluir al menos tres parámetros: los dos canales citados y la estructura sobre la que se produce el recorrido. Su comportamiento consistirá simplemente en una secuencia de mensajes, a razón de uno por cada elemento de la estructura, y la emisión de la señal de terminación cuando llega al final de la misma. La composición paralela de este iterador con un proceso-cuerpo como el descrito conforma, en definitiva, un bucle clásico, pero establecido sobre un contexto esencialmente concurrente.

Los apartados siguientes describen los iteradores utilizados a lo largo del Capítulo 7.

D.1.1 Sobre Listas

Este iterador es el más sencillo, y por ello es también el que muestra con mayor claridad el método utilizado en su diseño. En este caso, se sigue el esquema tópico de un recorrido recursivo

evidente [Par01], aunque éste último se considera mucho más intuitivo.

sobre listas: en cada paso, se considera –se emite– el primer elemento, y a continuación se invoca de nuevo el proceso sobre el resto de la lista. Cuando esta lista es la lista vacía, se genera la correspondiente señal de terminación.

$$\begin{array}{rcl} \text{LFor} & : & (\sharp\alpha;\ (\mathit{List}\,\alpha),\ \uparrow\alpha,\ \mathit{Sync}) \\ \text{LFor}(\mathit{lis}, \mathit{vcb}, \mathit{end}) & \triangleq & \textbf{if}\ (\texttt{null}\ \mathit{lis})\ \textbf{then}\ \overline{\mathit{end}} \\ & & \quad \textbf{else}\ (\ \overline{\mathit{vcb}}\langle\,(\texttt{car}\ \mathit{lis})\,\rangle.\,\text{LFor}\langle\,(\texttt{cdr}\ \mathit{lis}), \mathit{vcb}, \mathit{end}\,\rangle\ ) \end{array}$$

La definición es, como puede verse, muy simple. Tal vez sea necesario enfatizar, no obstante, que es totalmente polimórfica, y aplicable por tanto a listas de cualquier tipo; esto incluye a las listas nombradas, cuando se desea considerar cada elemento en su totalidad.

D.1.2 Sobre Listas Nombradas

Estos iteradores son totalmente análogos al anterior; tan sólo pretenden considerar el caso especial de las listas nombradas, descomponiendo cada uno de sus elementos –esto es, cada estructura del tipo $(\mathit{Named}\,\alpha)$– como sea preciso.

$$\begin{array}{rcl} \text{NLFor} & : & (\sharp\alpha;\ (\mathit{NList}\,\alpha),\ \uparrow\alpha,\ \mathit{Sync}) \\ \text{NLFor}(\mathit{lis}, \mathit{vcb}, \mathit{end}) & \triangleq & \textbf{if}\ (\texttt{null}\ \mathit{lis})\ \textbf{then}\ \overline{\mathit{end}} \\ & & \quad \textbf{else}\ (\ \overline{\mathit{vcb}}\langle\,(\texttt{ncar}\ \mathit{lis})\,\rangle.\,\text{NLFor}\langle\,(\texttt{cdr}\ \mathit{lis}), \mathit{vcb}, \mathit{end}\,\rangle\ ) \\ \text{LLFor} & : & (\sharp\alpha;\ (\mathit{NList}\,\alpha),\ \uparrow\mathit{Label},\ \mathit{Sync}) \\ \text{LLFor}(\mathit{lis}, \mathit{vcb}, \mathit{end}) & \triangleq & \textbf{if}\ (\texttt{null}\ \mathit{lis})\ \textbf{then}\ \overline{\mathit{end}} \\ & & \quad \textbf{else}\ \overline{\mathit{vcb}}\langle\,(\texttt{lcar}\ \mathit{lis})\,\rangle.\,\text{LLFor}\langle\,(\texttt{cdr}\ \mathit{lis}), \mathit{vcb}, \mathit{end}\,\rangle \end{array}$$

El primero (NLFor) recorre una lista nombrada, descartando en cada elemento el nombre, y emitiendo únicamente el *dato* concreto. Es decir, su comportamiento es externamente idéntico al del iterador anterior (LFor) sobre una lista convencional.

Por su parte, el segundo (LLFor) hace exactamente lo contrario, es decir, emite en cada caso la *etiqueta* del elemento, descartando el dato. A pesar de lo que podría parecer, se trata de una operación muy habitual, especialmente durante la construcción de una estructura.

D.1.3 Sobre Rangos de Enteros

Este iterador ha sido específicamente diseñado para emular al clásico bucle-**for** secuencial. Asume siempre que el extremo inferior del rango es uno, y se recibe como parámetro el superior (ul). Su estructura interna es totalmente convencional, y no precisa de mayor comentario. Obsérvese, no obstante, que su comportamiento externo es idéntico al que tendrían los anteriores sobre una lista de enteros que contuviera exactamente el mismo rango.

$$\begin{array}{rcl} \text{NFor} & : & (\mathit{Int},\ \uparrow\mathit{Int},\ \mathit{Sync}) \\ \text{NFor}(\mathit{ul}, \mathit{vcb}, \mathit{end}) & \triangleq & \text{IFor}\langle\,1, \mathit{ul}, \mathit{vcb}, \mathit{end}\,\rangle \\ \text{IFor} & : & (\mathit{Int},\ \mathit{Int},\ \uparrow\mathit{Int},\ \mathit{Sync}) \\ \text{IFor}(\mathit{dl}, \mathit{ul}, \mathit{vcb}, \mathit{end}) & \triangleq & \textbf{if}\ (\mathit{dl} = \mathit{ul})\ \textbf{then}\ \overline{\mathit{vcb}}\langle\,n\,\rangle.\,\overline{\mathit{end}} \\ & & \quad \textbf{else}\ (\overline{\mathit{vcb}}\langle\,\mathit{dl}\,\rangle.\,\text{IFor}\langle\,(\mathit{dl}+1), \mathit{ul}, \mathit{vcb}, \mathit{end}\,\rangle \end{array}$$

Su uso principal es el mismo que el del bucle clásico: proporcionar el índice para un recorrido secuencial sobre *vectores*. Éste es, precisamente, el motivo por el que se asume que el extremo inferior del rango es siempre la unidad. No obstante, si se necesita utilizar otro, basta con remitirse al proceso auxiliar (IFor), invocándolo ahora de manera directa.

Habitualmente, el extremo superior viene indicado por el tamaño (la cardinalidad) del vector; puede verse, por ejemplo, en la sección D.3.2 de este mismo Apéndice.

D.2 Concatenación de Listas

Los dos operaciones de esta sección realizan una concatenación entre dos listas del mismo tipo. Se trata, tal vez, de la única operación clásica que no ha sido incluida como primitiva (§C.4.4). La razón para esto es que no se utiliza en ningún punto de la especificación semántica, al menos en su forma pura. La definición siguiente muestra la estructura de un proceso análogo a la clásica función `append` de Lisp, que concatena dos listas en el mismo orden indicado:

$$
\begin{array}{rcl}
\text{Append} & : & (\sharp\alpha;\ (\mathit{List}\,\alpha),\ (\mathit{List}\,\alpha),\ \uparrow(\mathit{List}\,\alpha)) \\
\text{Append}(l_1, l_2, r) & \triangleq & \textbf{if}\ (\texttt{null}\ l_1)\ \textbf{then}\ \overline{r}\langle\, l_2 \,\rangle \\
& & \quad \textbf{else}\ (\nu\, r_m)(\ \text{Append}\langle\, (\texttt{cdr}\ l_1), l_2, r_m \,\rangle \\
& & \qquad \mid\ r_m(l_m).\,\overline{r}\langle\, (\texttt{cons}\ (\texttt{car}\ l_1)\ l_m) \,\rangle.\,\mathbf{0}\ )
\end{array}
$$

Esta definición se incluye solamente porque su forma es prácticamente idéntica a la siguiente. Se trata, en este caso, de una *concatenación sin duplicados*: los elementos de la primera lista se añaden al principio de la segunda, pero sólo en el caso de que no estuvieran ya contenidos en ésta. Se asume pues que la segunda lista carece a su vez de elementos repetidos.

$$
\begin{array}{rcl}
\text{ApNDup} & : & (\sharp\alpha;\ (\mathit{List}\,\alpha),\ (\mathit{List}\,\alpha),\ \uparrow(\mathit{List}\,\alpha)) \\
\text{ApNDup}(l_1, l_2, r) & \triangleq & \textbf{if}\ (\texttt{null}\ l_1)\ \textbf{then}\ \overline{r}\langle\, l_2 \,\rangle \\
& & \quad \textbf{else}\ (\nu\, r_m)(\ \text{ApNDup}\langle\, (\texttt{cdr}\ l_1), l_2, r_m \,\rangle \\
& & \qquad \mid\ r_m(l_m).\,\textbf{if}\ (\texttt{ein}\ (\texttt{car}\ l_1)\ l_m)\ \textbf{then}\ \overline{r}\langle\, l_m \,\rangle \\
& & \qquad\quad \textbf{else}\ \overline{r}\langle\, (\texttt{cons}\ (\texttt{car}\ l_1)\ l_m) \,\rangle.\,\mathbf{0}\ )
\end{array}
$$

Esta operación es más habitual que la anterior, y a diferencia de ésta, se utiliza en diversos puntos de la especificación de la semántica de $\mathcal{P}i\mathcal{L}ar$.

D.3 Conversión de Enumeraciones

En la semántica de $\mathcal{P}i\mathcal{L}ar$ se encuentran básicamente dos tipos de enumeraciones de longitud indeterminada: listas y vectores. Las primeras se construyen de manera explícita, y dan forma a un tipo de datos perfectamente conocido y controlado. Los vectores, en cambio, aunque más sencillos, tienen una configuración menos definida (§C.4.1). Se introducen en el lenguaje con el fin de expresar el modo en que se acomodan los parámetros recibidos desde la fase sintáctica, donde éstos se proporcionan de manera desestructurada. Por supuesto, podría asumirse una

fase de procesado previo, y suponer que todo son listas; pero este esquema es más cercano al comportamiento habitual.

Una vez que se dispone de ambos tipos de enumeraciones, es habitual tener que realizar conversiones entre ellas. Éstas se describen a continuación, en ambos sentidos:

D.3.1 De Listas a Vectores

La operación siguiente muestra un esquema muy simple: se recorre la lista, y cada uno de sus nodos se acumula como enésimo elemento del vector correspondiente. El proceso definido, así como los dos siguientes, sirven además como ejemplo del uso de los iteradores descritos en la sección D.1 de este mismo Apéndice.

$$
\begin{array}{rcl}
\mathrm{LisVec} & : & (\alpha;\, (\mathit{List}\,\alpha),\, \uparrow\!\sim\alpha) \\
\mathrm{LisVec}(\mathit{lis}, r) & \triangleq & (\nu\, v_1, e_1, \widetilde{\mathit{vec}})(\ \mathrm{LFor}\langle\, \mathit{lis}, v_1, e_1\,\rangle \\
 & & \mid \mathrm{LVec}_F\langle\, 1, \mathit{vec}, v_1, e_1, r\,\rangle\ ) \\
\mathrm{LVec_F} & : & (\alpha;\, \mathit{Int},\, \sim\alpha,\, \uparrow\alpha,\, \mathit{Sync},\, \uparrow\!\sim\alpha) \\
\mathrm{LVec_F}(n, \widetilde{\mathit{vec}}, \mathit{vcb}, \mathit{end}, r) & \triangleq & (\ \mathit{vcb}(\mathit{vec}_n).\, \mathrm{LVec}_F\langle\, n+1, \widetilde{\mathit{vec}}, \mathit{vcb}, \mathit{end}, r\,\rangle \\
 & & +\ \mathit{end}.\, \overline{r}\langle\, \widetilde{\mathit{vec}}\,\rangle.\, \mathbf{0}\ )
\end{array}
$$

Lo único reseñable de esta definición es el modo en que el vector va creciendo dinámicamente. Inicialmente, se declara en LisVec como un vector vacío ($\widetilde{\mathit{vec}}$); pero el que le dota de contenido es el proceso auxiliar LVec_F. En éste, cada elemento emitido por el iterador se recibe en una variable vec_n, que queda ligada en ese preciso instante. Estas variables se relacionan entre sí solamente a través de su subíndice; por lo demás, como en cualquier vector o tupla, se crean y manipulan de manera totalmente independiente.

Las dos definiciones siguientes hacen la misma operación sobre una lista nombrada, obteniendo en cada caso un vector de datos o de etiquetas; por lo demás, son totalmente análogos.

$$
\begin{array}{rcl}
\mathrm{NLisVec} & : & (\alpha;\, (\mathit{NList}\,\alpha),\, \uparrow\!\sim\alpha) \\
\mathrm{NLisVec}(\mathit{lis}, r) & \triangleq & (\nu\, v_1, e_1, \widetilde{\mathit{vec}})(\ \mathrm{NLFor}\langle\, \mathit{lis}, v_1, e_1\,\rangle \\
 & & \mid \mathrm{LVec}_F\langle\, 1, \mathit{vec}, v_1, e_1, r\,\rangle\ ) \\
\mathrm{LLisVec} & : & (\alpha;\, (\mathit{NList}\,\alpha),\, \uparrow\!\sim\mathit{Label}) \\
\mathrm{LLisVec}(\mathit{lis}, r) & \triangleq & (\nu\, v_1, e_1, \widetilde{\mathit{vec}})(\ \mathrm{LLFor}\langle\, \mathit{lis}, v_1, e_1\,\rangle \\
 & & \mid \mathrm{LVec}_F\langle\, 1, \mathit{vec}, v_1, e_1, r\,\rangle\ )
\end{array}
$$

Obsérvese que ambos procesos utilizan el mismo cuerpo (LVec_F) definido para el bucle normal. De hecho, el iterador utilizado es la única diferencia en cada uno de los tres casos.

D.3.2 De Vectores a Listas

El esquema de la operación siguiente es muy similar al de las anteriores; de hecho resulta, si cabe, aún más sencillo. Se itera sobre un rango de enteros dado por el tamaño del vector; éste se

recorre, usando los valores emitidos como índice, y cada uno de sus elementos se va acumulando en una lista inicialmente vacía. Es ésta la que se obtiene finalmente como resultado.

$$
\begin{array}{rcl}
\mathrm{VecLis} & : & (\alpha;\, {\sim}\alpha,\, \uparrow(\mathit{List}\,\alpha)) \\
\mathrm{VecLis}(\widetilde{v}, r) & \triangleq & (\nu\, v_1, e_1)(\ \mathrm{NFor}\langle\, |\widetilde{v}|, v_1, e_1\,\rangle \\
 & & \mid\ \mathrm{VLis}_F\langle\, \widetilde{v}, (\mathtt{nil}), v_1, e_1, r\,\rangle\ ) \\
\mathrm{VLis_F} & : & (\alpha;\, {\sim}\alpha,\, (\mathit{List}\,\alpha),\, \uparrow\mathit{Int},\, \mathit{Sync},\, \uparrow(\mathit{List}\,\alpha)) \\
\mathrm{VLis_F}(\widetilde{v}, \mathit{lis}, \mathit{vcb}, \mathit{end}, r) & \triangleq & (\ \mathit{vcb}(\mathit{ix}). \\
 & & \quad \mathrm{VLis}_F\langle\, \widetilde{v}, (\mathtt{cons}\ v_{ix}\ \mathit{lis}), \mathit{vcb}, \mathit{end}, r\,\rangle \\
 & & +\ \mathit{end}.\,\overline{r}\langle\, \mathit{lis}\,\rangle.\,\mathbf{0}\ )
\end{array}
$$

Obsérvese que el orden de los elementos en la lista es el inverso del utilizado en el vector. Esto no plantea problema alguno, ya que en todos los usos de esta función, el orden será irrelevante. No obstante, si éste fuera importante en un momento dado, bastaría con modificar ligeramente la definición de VLis_F, de modo que en lugar de acceder en cada caso al elemento en la posición ix-ésima, lo hiciera al de la posición $(|\tilde{v}| - ix + 1)$.

El siguiente proceso es análogo al anterior, pero debe producir una lista nombrada. Por tanto, necesita partir de dos vectores, uno de etiquetas y otro de datos; por tanto, no se realiza solamente una conversión, sino también una fusión de estructuras.

$$
\begin{array}{rcl}
\mathrm{VecNLis} & : & (\alpha;\, {\sim}\mathit{Label},\, {\sim}\alpha,\, \uparrow(\mathit{NList}\,\alpha)) \\
\mathrm{VecNLis}(\widetilde{l}, \widetilde{d}, r) & \triangleq & (\nu\, v_1, e_1)(\ \mathrm{NFor}\langle\, |\widetilde{l}|, v_1, e_1\,\rangle \\
 & & \mid\ \mathrm{VNLis}_F\langle\, \widetilde{l}, \widetilde{d}, (\mathtt{nil}), v_1, e_1, r\,\rangle\ ) \\
\mathrm{VNLis_F} & : & (\alpha;\, {\sim}\mathit{Label},\, {\sim}\alpha,\, (\mathit{NList}\,\alpha),\, \uparrow\mathit{Int},\, \mathit{Sync},\, \uparrow(\mathit{NList}\,\alpha)) \\
\mathrm{VNLis_F}(\widetilde{l}, \widetilde{d}, \mathit{lis}, \mathit{vcb}, \mathit{end}, r) & \triangleq & (\ \mathit{vcb}(\mathit{ix}). \\
 & & \quad \mathrm{VNLis}_F\langle\, \widetilde{l}, \widetilde{d}, (\mathtt{ncons}\ l_{ix}\ d_{ix}\ \mathit{lis}), \mathit{vcb}, \mathit{end}, r\,\rangle \\
 & & +\ \mathit{end}.\,\overline{r}\langle\, \mathit{lis}\,\rangle.\,\mathbf{0}\ )
\end{array}
$$

Se asume, por supuesto, que ambos vectores tienen el mismo tamaño y siguen el mismo orden. Obviamente, de no ser así, no existe modo alguno de fusionarlos en una única estructura.